View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarBank@NUS

SAFA: Stack And Frame Architecture

BY

Soo Yuen Jien

(B.Sc (Hon) NUS, M.Sc NUS)

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
2005

Acknowledgment

First and foremost, I would like to thank my supervisor, Professor Yuen Chung Kwong, for suggesting such an interesting research topic. His knowledge and insight on the subject has guided me through many thorny issues. More importantly, his kind words have given me more confidence in the research direction.

I wish to express my gratitude to my research review committee members, Professor Teo Yong Ming and Associate Professor Wong Weng Fai. They have frequently pointed out blind spots in my research method, steering the research from potential pitfalls.

Last but not least, I would like to thank my wife, my parent and family members for their unfailing support and encouragement.

Summary

Superscalar execution of computer instructions exists in many forms, which can be grouped roughly into two major camps: the hardware approach with examples like Alpha, PowerPC, x86 etc; the software approach with heavy reliance on compilers e.g. VLIW, EPIC etc. However, these approaches shares many characteristic and can be studied under a cohesive framework, which we termed as General Tagged Execution Framework. By exploiting the commonality of the approaches, it is possible to apply a combination of subsets of techniques under a different context.

Specifically, we investigated the feasibility of adapting some well studied techniques to a stack-oriented architecture. The research concentrates on two major areas of a stack architecture, namely high level language support and low level instruction execution. In the first area, improved control flow and data structure support are studied. For the low level instruction execution, superscalar and speculative execution techniques are incorporated. As a platform for experimenting with these mechanisms, we designed and implemented a simulator for a new stack architecture, named as SAFA (Stack And Frame Architecture).

Contents

1 Introduction **1**

1.1 General Tagged Execution Framework 4

1.2 The SAFA Architecture 6

1.3 Objectives of Our Work 8

1.4 Overview of Thesis 9

2 Literature Survey **10**

2.1 Introduction 10

2.2 Objectives 10

2.3 Stack Based Architecture 12

2.3.1 Burroughs Family B5000-B6700 12

2.3.2 Hewlett-Packard HP3000 13

2.3.3 Intel iAPX432 14

2.3.4 INMOS transputer 15

2.3.5 Java Virtual Machine and picoJava implementation 17

2.3.6 Conclusion 18

2.4 Register-Based Superscalar Architecture 20

2.4.1 Alpha Family 20
2.4.2 PowerPC Family 23
2.4.3 Conclusion 25
2.5 Summary 26

3 High Level Language Support 27

3.1 Control Flow 28
3.1.1 Procedure Activation 28
3.1.2 Repetitive Execution with Counter 33
3.2 Data Structure 38
3.2.1 Array 38
3.2.2 Linked List 41
3.3 Object Oriented Language 43
3.3.1 Object Representation 44
3.3.2 Dynamic Method Dispatching 46
3.4 Additional Benefits of Frame Register 52
3.4.1 Context Sensitivity 52
3.4.2 Prefetching 57
3.5 Summary 59

4 Low Level Execution Support 60

4.1 Overview of Instruction Dependencies 62
4.1.1 Data Dependence 62
4.1.2 Name Dependence 63

4.1.3 Control Dependence . . . 65
4.2 Coping with Data and Name Dependence . . . 66
4.2.1 Tomasulo's Scheme . . . 66
4.2.2 Adaptation for SAFA . . . 71
4.3 Coping with Control Dependence . . . 85
4.3.1 Branch Prediction and Speculative Execution in General . . . 85
4.3.2 Branch Prediction and Speculative Execution in SAFA . . . 88
4.3.3 Limitation of Speculative Execution in SAFA . . . 95
4.4 Coping with Frequent Memory Movements . . . 97
4.4.1 Local Data Access in SAFA . . . 100
4.5 Advances in Java Technology . . . 114
4.5.1 Comparison: SAFA vs Java Processors . . . 118
4.6 Influence of General Tagged Execution Framework . . . 120
4.7 Summary . . . 121

5 Benchmark Environment 122

5.1 Hardware - SAFA Simulator . . . 122
5.1.1 Fetch Unit . . . 125
5.1.2 Decode Unit . . . 125
5.1.3 Issue Unit . . . 126
5.1.4 Execution Units . . . 128
5.1.5 Frame Registers Unit . . . 128
5.1.6 Branch Predictor Unit . . . 129

5.1.7 Overall System . . . 130
5.1.8 Verification of SAFA Simulator . . . 131
5.2 Software - Assembler and Cross-Assembler . . . 134
5.3 Benchmark Programs . . . 136
5.3.1 Sieve of Erathosthense . . . 137
5.3.2 Bubble Sort . . . 138
5.3.3 Fibonacci Series . . . 139
5.3.4 Quick Sort . . . 140
5.3.5 Test Score Accumulation: Array and List . . . 141
5.3.6 Linpack - Gaussian Elimination . . . 142
5.4 Hardware Parameters . . . 144
5.5 Instruction Type and Execution Time . . . 146
5.5.1 Derivation of Instruction Execution Time . . . 146
5.6 Summary . . . 147

6 Benchmark Results 148

6.1 Benchmark Notation . . . 148
6.2 High Level Language Support . . . 151
6.2.1 Data Structure Support: Array . . . 151
6.2.2 Data Structure Support: Array of Records . . . 155
6.2.3 Data Structures Support: Linked List . . . 159
6.3 Low Level Instruction Support . . . 165
6.4 Various Benchmarks: Single Execution Unit . . . 166

6.4.1 Fibonacci Series . . . 167

6.4.2 Sieve of Erathosthense . . . 171

6.4.3 Bubble Sort . . . 175

6.4.4 Quick Sort . . . 177

6.4.5 Linpack: Gaussian Elimination . . . 180

6.5 Various Benchmarks: Multiple Execution Units . . . 184

6.5.1 Bubble Sort . . . 184

6.5.2 Linpack Benchmark . . . 187

6.6 Various Benchmarks: Local Data Access Optimization . . . 190

6.6.1 Fibonacci Series . . . 191

6.6.2 Sieve of Erathosthense . . . 195

6.6.3 Quick Sort . . . 199

6.6.4 Bubble Sort . . . 203

6.7 Conclusion . . . 207

7 Topical Benchmarks 208

7.1 Large Application . . . 209

7.1.1 Benchmark Result . . . 212

7.2 Instruction Folding . . . 215

7.2.1 SAFA vs Instruction Folding . . . 219

7.2.2 SAFA with Instruction Folding . . . 222

7.3 General Purpose Register Machine . . . 225

7.4 Conclusion . . . 230

8 Conclusion **231**

8.1 Contribution 231

8.2 Future Work 233

Appendices **245**

A SAFA Assembly Code and Assembler **245**

A.1 Frame Register Instructions 247

A.2 Direct Memory Access Instructions 251

A.3 Integer Instructions 252

A.4 Floating Point Instructions 254

A.5 Branching Instructions 257

A.6 Stack Manipulation Instructions 261

A.7 SAFA Assembler Introduction 264

A.7.1 Syntax for Procedure 264

A.7.2 Syntax for Data Values 265

A.7.3 Built in Assembly Macros 268

A.7.4 Sample Translation 270

A.7.5 Using the assembler 271

B SAFA Simulator **272**

B.1 Simulator in Plain Text 272

B.1.1 Configuration File 274

B.1.2 Statistic File 274

B.1.3 Memory Dump and CPU State 279

B.2 Simulator with GUI 281

B.2.1 Main Control Panel 283

B.2.2 Components Window 286

C SAFA Benchmark Programs 297

C.1 Sieve of Erathosthense 297

C.2 Bubble Sort 299

C.3 Bubble Sort: Frame Register Version 301

C.4 Fibonacci Series 303

C.5 Quick Sort 304

C.6 Student Array: Conventional Array Access 306

C.7 Student Array: Frame Register and Index 307

C.8 Student Array: Frame Register and Offset 308

C.9 Student List: Conventional Linked List Traversal 309

C.10 Student List: Frame Register and Index 310

C.11 Student List: Frame Register and Offset 311

C.12 Linpack Benchmark 312

List of Figures

1.1 Tagged Execution Framework. 4

3.1 Dynamic Dispatching in OOLs 50

3.2 Object Representation in SAFA 51

4.1 Simple Architecture without Tomasulo's Scheme 67

4.2 Simple Architecture with Tomasulo's Scheme 69

4.3 Control Dependence Example 1: *if-else* 86

4.4 Control Dependence Example 2: *while* loop 86

4.5 Prediction Level Example 88

4.6 Single Level Prediction 92

4.7 Multiple Level Prediction 94

4.8 Machine State before Branch 109

4.9 Machine State at Point A 109

4.10 Sun Microsystems picoJava Block Diagram 115

5.1 SAFA Components Diagram 124

6.1 Bubble Sort(50 Numbers): Comparison 152

6.2 Bubble Sort(50 Numbers): Conventional Array Access Instruction Composition . . . 153

6.3 Bubble Sort(50 Numbers): Frame Registers Version Instruction Composition . . . 153

6.4 Student Array (100 Records): Comparison . . . 156

6.5 Student Linked List (100 Records): Comparison . . . 162

6.6 Fibonacci Series. Fib(10) : Speed Up . . . 170

6.7 Fibonacci Series: Composition . . . 170

6.8 Sieve of Erathosthense (100 Numbers) : Speed Up . . . 173

6.9 Sieve of Erathosthense: Composition . . . 173

6.10 Bubble Sort (50 Numbers) : Speed Up . . . 176

6.11 Quick Sort (50 Numbers) : Speed Up . . . 178

6.12 Quick Sort: Composition . . . 178

6.13 Linpack Benchmarks : Speed Up . . . 181

6.14 Linpack Benchmarks: Composition . . . 181

6.15 Bubble Sort (50 Numbers) : Multiple Execution Units - Speed Up Comparison . . . 185

6.16 Linpack Benchmark (15 x 15): Multiple Execution Units - Speed Up Comparison . . . 188

6.17 Fibonacci Series: Local Variable Access - Speed Up Comparison . . . 192

6.18 Fibonacci Series: Local Variable Access - Execution Time Comparison 192

6.19 Fibonacci Series: Local Variable Access (Stack Frame) Instruction Composition . . . 194

6.20 Fibonacci Series: Local Variable Access (Operand Stack) Instruction Composition 194

6.21 Sieve of Erathosthense: Local Variable Access - Speed Up Comparison 195

6.22 Sieve of Erathosthense: Local Variable Access - Execution Time Comparison 196

6.23 Sieve of Erathosthense: Local Variable Access (Stack Frame) Instruction Composition 196

6.24 Sieve of Erathosthense: Local Variable Access (Operand Stack) Instruction Composition 198

6.25 Quick Sort: Local Variable Access - Speed Up Comparison 200

6.26 Quick Sort: Local Variable Access - Execution Time Comparison . . . 200

6.27 Quick Sort: Local Variable Access (Stack Frame) Instruction Composition 202

6.28 Quick Sort: Local Variable Access (Operand Stack) Instruction Composition 202

6.29 Bubble Sort: Local Variable Access - Speed Up Comparison 204

6.30 Bubble Sort: Local Variable Access - Execution Time Comparison . . 204

6.31 Bubble Sort: Local Variable Access (Stack Frame) Instruction Composition 206

6.32 Bubble Sort: Local Variable Access (Operand Stack) Instruction Composition 206

7.1 Compress (4000 bytes Text) - Speed Up Comparison 214

7.2 Compress (8 kbytes Binary) - Speed Up Comparison 214

7.3 Fibonacci Series : SAFA with Folding - Speed Up 223

7.4 Sieve of Erathosthense: SAFA with Folding - Speed Up 223

7.5 Quick Sort: SAFA with Folding - Speed Up 224

7.6 Bubble Sort: SAFA with Folding - Speed Up 224

8.1 Ideas Relationship in SAFA 234

A.1 Syntax for a Procedure in SAFA Assembly Code. 265

A.2 Layout of a Procedure Stack Frame 266

B.1 Sample Configuration File 275

B.2 Sample Statistic File (Part1) 276

B.3 Sample Statistic File(Part2) 277

B.4 Sample Statistic File (Part 3) 278

B.5 Sample Memory Dump File (Partial) 279

B.6 Sample CPU Trace File (Abridged) 280

B.7 SAFA Simulator GUI v1.5 Screen Shot 282

B.8 Main Control Panel GUI 283

B.9 Fetch Unit GUI 286

B.10 Decode Unit GUI 287

B.11 Issue Unit GUI 289

B.12 Frame Register Unit GUI 291

B.13 Branch Predictor Unit GUI 293

B.14 Execution Unit GUI 294

B.15 Memory Unit GUI 295

List of Tables

4.1 Speculative Consumption of Result 96

4.2 Confirmation of Prediction PL j 96

4.3 Handling Misprediction at PL j 96

6.1 Bubble Sort 50 Numbers: Conventional Array Access 154

6.2 Bubble Sort 50 Numbers: Using Frame Register 154

6.3 Student Array (100 records) Benchmark: Conventional Array Access 157

6.4 Student Array (100 records): Using Frame Register (version 1) 157

6.5 Student Array (100 records): Using Frame Register (version 2) 158

6.6 Student Linked List (100 records): Conventional Linked List Traversal 163

6.7 Student Linked List (100 records): Using Frame Register and Index . 163

6.8 Student Linked List (100 records): Using Frame Register and Offset . 164

6.9 Fibonacci(10) = 55. Total Recursive Calls = 177 169

6.10 Sieve of Erathosthense: 100 Numbers 174

6.11 Quick Sort: 50 Numbers. Total Recursive Calls = 43 179

6.12 Linpack(5): Solve 5 x 5 floating point matrix using Gaussian Elimination. 182

6.13 Linpack(10). Solve 10 x 10 floating point matrix using Gaussian Elimination. 182

6.14 Linpack(15). Solve 15 x 15 floating point matrix using Gaussian Elimination. 183

6.15 Bubble Sort (50 Numbers): Multiple Execution Units - Comparison . 186

6.16 Linpack Benchmark (15 x 15): Multiple Execution Units - Comparison 189

6.17 Fibonacci Series : Local Variable Access - Comparison 193

6.18 Sieve of Erathosthense: Local Variable Access - Comparison 197

6.19 Quick Sort: Local Variable Access - Comparison 201

6.20 Bubble Sort: Local Variable Access - Comparison 205

7.1 Compress (4000bytes Text): Summary 213

7.2 Compress (8kbytes Binary) - Summary 213

7.3 Folding Benchmarks without LDM: Summary 219

7.4 Folding Benchmarks with LDM: Summary 219

7.5 SAFA vs Instruction Folding (without LDM): Summary 221

7.6 SAFA vs Instruction Folding (with LDM): Summary 221

7.7 Bubble Sort(250) on SimpleScalar: Non-Optimized 229

7.8 Bubble Sort(250) on SimpleScalar: Optimized 229

7.9 Bubble Sort(250) on SAFA with LDM 229

Chapter 1

Introduction

"The number of transistors on an Integrated Chip will double every 18 months.", these are the words of the widely known *Moore's Law*[1] due to Gordon Moore in 1965. This observation, amid doubts and speculations, has held true for several decades, witnessing exponential growth of both component count and structural complexity of electronic chips. As an example, consider the first fully-electronic programmable computer *ENIAC* in 1940s, which had a mammoth foot print of 9 by 15 meters. Nowadays, even a handheld calculator of 9 by 15 *centimeters* has more computing power.

However, the ability to cramp more components into an ever decreasing space was only partially responsible for the increase in computing power. Transistors are just the raw building material that must must be harnessed into a meaningful design. Computer architecture completes the picture by imposing the structure on the raw components for better and more efficient computation, which usually takes the form of a set of machine instructions.

The execution of a machine instruction in a Von Neumann Machine[2] is frequently compared to a production line in the real world, for example, the automobile

[1] One of the many formulations.

[2] Computer with independent but interconnected memory and execution unit

assembly line. Just as a car undergoes several assembly stages, an instruction goes through several well defined stages as well, generally:

1. **Fetch:** To bring an instruction from the memory store into the execution core.
2. **Decode:** Determine the operation(s) to be performed as indicated by the instruction.
3. **Execution:** Execute the operation(s) required.
4. **Write Back:** The result of the execution is recorded.

The similarity between the real world assembly line and the minute one in the **C**entral **P**rocessing **U**nit allows many useful techniques to be shared. One good example would be the pipeline process. By splitting the procedure of car assembling into several stages, multiple cars at various stages can be worked on at the same time. Consider a simple scenario: a car assembly line with four stages where each stage takes one day can be expected to finish 12 cars in 15 days.

However, the pipelining in CPU does not yield such a speed up usually. There are two main reasons:

1. **Inter-Dependency between Instructions:** Unlike individual cars on the assembly line, machine instructions are usually inter-related. For example, an instruction may depend on the previous one for producing the needed data. In this case, the latter instruction must wait until the former instruction is executed before proceeding. Such relations restrict the order of the execution as well as impose delays in execution, and prevent many parallelizing techniques from running at full steam.
2. **Limited Resources:** Because of resource limitations, a CPU may not be able to accommodate more instructions running at the same time. These resources include registers (or similar structures to hold data), execution units, etc.

A large number of techniques have been proposed to mitigate these restrictions. The famous Tomasulo's Scheme [38] was proposed to enable dynamic scheduling of instructions, thereby curbing the dependency problem mentioned. By renaming registers (also known as tagging), the operands and result of an instruction are associated with a tag (or virtual register number) instead of a real physical register. Since real physical registers now can be utilized more freely by having different tags as needed, resource dependency problems would be less frequent. With dynamic scheduling and register renaming, it is now possible to process (issue) more than one instruction in a clock cycle. This technique has been the backbone for quite a number *superscalar* (multi-issue) architectures. Although the Tomasulo's Scheme requires relatively complicated hardware implementation, little special attention is needed from compilers.

Reminiscent of the heated debate of RISC[3] and CISC[4] in the 80's, another approach that requires more sophisticated compilers but relatively simple hardware has been proposed. The **V**ery **L**ong **I**nstruction **W**ord (VLIW) architecture depends on the compiler to extricate (disentangle) inter-dependent instructions and group independent instructions into a parallel package (also known as an instruction word/bundle). Since there is no dependency between instructions in a package, they can be executed simultaneously without further checking. As succinctly put by the Online Byte Magazine, "VLIW is basically a software- or compiler- based superscalar architecture. "

The two approaches mentioned spark off enthusiastic research into the respective areas with abundant results. At first glance, they seem quite different from each other, with distinct emphasis on separate part of the instruction execution. However, we feel that it would be beneficial to put them under a common cohesive framework. This conceptual framework is presented in the next section.

---

[3]Reduced Instruction Set Computer

[4]Complex Instruction Set Computer

1.1 General Tagged Execution Framework

By extracting the commonality between the approaches, we find that there is a underlying common conceptual framework, as shown in the Figure 1.1.

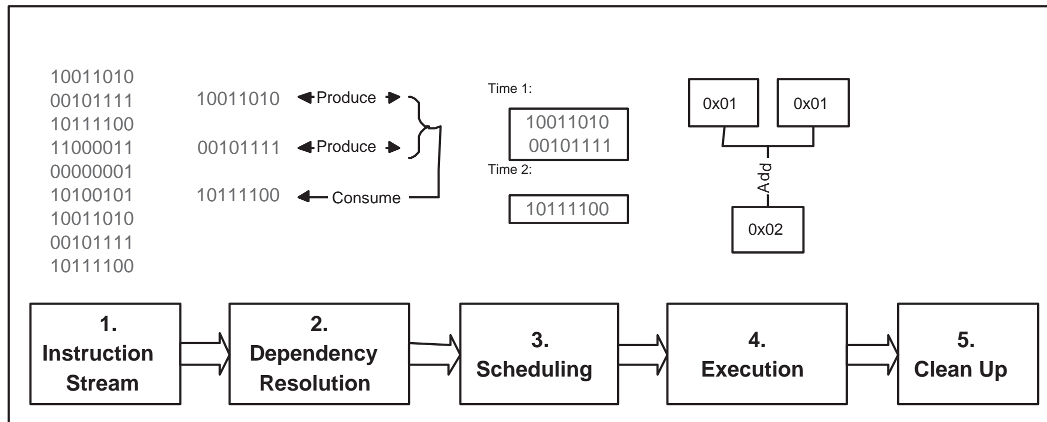


Figure 1.1: Tagged Execution Framework.

As can be seen in the framework, we start with a stream of instructions in stage one going into the framework. Instruction dependency checking is performed in stage two. Producer instructions pick up a fresh tag to identify their future results, while consumers instructions collect operands (identified by tags). Instructions can be said to have lost their original form at this stage, and become a more general execution package, which describes a manipulation based on tags. In stage three, an execution package that is considered ready based on a set of criteria get scheduled. The readiness criteria can differs from system to system. The actual execution happens in stage four. Finally, in stage five, execution results will be stored, tags and other resources will be released.

This conceptual framework captures quite a number of existing computer architectures. Since one or more stages preceding the execution stage in the figure above can be implemented either in hardware or software, a number of interesting models arise. For example, for a superscalar (multi-issue) machine that employs Tomasulo's Scheme (e.g. PowerPC, Alpha), the second and third stage would be implemented by a Reorder Buffer and Common Data Bus in hardware and the fourth

stage would be a superscalar pipelined execution engine.

For a VLIW architecture, (e.g. IA64 EPIC), the second and third stage would be performed in software (the compiler), with limited scheduling in hardware and the fourth stage would be a EPIC execution engine to process instruction bundles. A dynamically scheduled VLIW machine would have both 2nd and 3rd stage in hardware, with an EPIC-like execution engine.

Also, it is interesting to note that the type of instruction set *does not* matter in this framework. As instructions pass through the tagging stage and are transformed into an execution package as described previously, similar techniques at the later stages are equally valid. Traditionally, different types of instruction set (commonly known as 0-, 1- and 2- operands instructions) requires their own specialized hardware for execution. With this framework, however, it is possible to consider the possibility of utilizing previously used techniques on a wide range of instruction sets, all producing tagged instructions that produce/consume data via virtual registers.

Based on this observation, we decided to study the feasibility of applying tagging to the stack-oriented instruction set. The benefits for this choice are two fold:

1. Traditionally, stack-oriented machines suffered the most under the problems mentioned. The fading of stack machines from the computer architecture scene can be largely attributed to the fact that stack machines fail to incorporate new parallelizing techniques devised for other platforms.

2. Recent popularity of the programming language Java and its underlying virtual machine (JVM), which is a stack based machine, have rekindled interest in this area.

With this in mind, we introduce the **S**tack **A**nd **F**rame **A**rchitecture, SAFA.

1.2 The SAFA Architecture

Traditionally, a pure stack-based instruction set is also known as the 0-address or 0-operand instruction set. As opposed to the general-purpose register instruction set, where the operands of an operation (stored in registers) are stated explicitly in the instruction, or the accumulator instruction set, where one of the operands is stated explicitly and the other is assumed in the accumulator implicitly, the stack-based instruction set assume that the operands exist on a stack and consequently does not carry any explicit operand[1].

In the 70s, when main memory storage was a scarce and expensive resource, stack based machines enjoyed popular acceptance because of the compact binary code produced. Besides, the stack is also a natural data structure used frequently in high-level programming language (HLP) execution, e.g. activation records of procedural languages, simple variable scoping etc. However, the limitation of stack machines become apparent when better and more efficient instruction execution techniques, like superscalar execution, pipelining etc were found to be inapplicable. In [27], the limitation of the stack machine is summarized as:

> The stack oriented architectures .... has passed from the scene because it is difficult to speed execution of such a processor because the stack pointer manipulations become a bottleneck.

The other major disadvantage of the stack instruction set is the poor execution support for data structure, for example array indexing. Since the array is one of the most frequently used data structures, inefficient support of these operations seriously handicaps the stack architecture.

However, recent development in the field shows that a stack architecture still has its attractiveness. For example, *Java*, one of the fastest growing programming languages, is implemented on top of a virtual machine, the *Java Virtual Machine (JVM)*[14]. The designers have chosen the stack architecture for the *JVM*

because of the simplicity in design as well as the compact binary size produced[13]. Hardware implementation of JVM, the *picoJava*[10][11] architecture, shows that it is possible to overcome some of the inherent disadvantages of a stack architecture.

For our project, we have devised a set of mechanisms that concentrate on the two following areas:

1. **High Level Language Support:**
    - Instructions with hardware support for HLP execution, especially subroutine entrance and exit, variable scoping and stack frame accesses.
    - Improved data structure and control flow support for HLP.
2. **Low Level Instruction Support:**
    - Hardware stack structure that uses tagged execution to support Instruction Level Parallelism, ILP.
    - Speculative execution of stack instructions.
    - Retention of frequently accessed data in CPU core to minimize memory access.

Although these ideas/mechanisms are mostly self-contained and applicable to other suitable architectures, we need a flexible independent platform to introduce *all* of them for experimentation. The SAFA architecture is thus designed as a means to experiment with the ideas mentioned, as well as to study the interaction between them. Detailed explanations will be given in the relevant chapters, according to the categorization above: High Level Language support in Chapter 3, and Low Level Instruction execution in Chapter 4.

1.3 Objectives of Our Work

We shall see that the ideas in *SAFA* architecture are able to overcome the weaknesses of stack architectures, while strengthening the advantages. These mechanisms would be able to provide:

1. Good instruction level parallelism without the heavy compiler optimization usually needed in GPR machines.
2. Good support for high level programming languages, including procedure activation and array indexing.
3. More expressive instructions that allow compact binary code size.
4. Optimized local data access.

The SAFA architecture, as a complete package, has the following advantages:

1. General purpose yet providing hardware alternative to the Java Virtual Machine.
2. A possible choice as an embedded processor because of its good performance with simple hardware implementation and compact binary size.

Also, by showing that tagging stack-based instructions within a more general architecture, the usefulness of the general execution framework proposed in the last section could be established. With this framework in place, more cohesive studies of the topic can be made in the future.

1.4 Overview of Thesis

An overview of the remaining chapters in this thesis is given below:

Chapter 2 Gives a short related literature survey.

Chapter 3 Discusses the ideas we adapted in SAFA to improve high level language support.

Chapter 4 Explains the ideas we adapted to improve low level execution of stack-oriented instructions.

Chapter 5 Lay out the setup of the benchmarking.

Chapter 6 Presents the benchmark results of the SAFA simulator. Several representative programs are executed to exploit the various new hardware features.

Chapter 7 Presents several topical benchmark studies to provide a broader perspective.

Chapter 8 Concludes the thesis by summarizing the contribution of work done. Possible future continuation work is also discussed.

Chapter 2

Literature Survey

2.1 Introduction

This chapter summarizes the literature survey we have done as related to our research proposal. The methodology as well as the objectives of the survey is given in the Section 2.2. Detailed information for each of the included machines is presented, along with comparison of our proposed alternative. A brief conclusion that summary of the survey is presented in Section 2.5.

2.2 Objectives

As mentioned in Chapter 1, in addition to studying the general applicability and potential of a tagged execution, our project also aims to research the possibility and feasibility of designing a stack machine that is efficient at the instruction set level and provides good support for executing high level programming languages. Hence, a survey on past machine architectures would serve as both guideline and comparative framework for our design. With this in mind, we have cast our net into the past few decades to study a few architectures that have one or more of the following features:

0-Address Instruction Set As stated in [1], 0-address instruction set machine, which is usually considered as the pure stack machine, utilizes a stack for evaluating expressions. Most instructions assume the operands needed for carrying out the operations reside on a stack (whether in CPU hardware or memory). This makes stack machines very different from general purpose registers machines.

High Level Language Support As early as the 1970s, designers of machine architecture have realized the importance of good support for executing high level language programs[6]. Efficient instruction level execution cannot guarantee good overall performance of a CPU, if the support for high level language construct likes variable scoping, method/function invocation, information hiding/protection etc is lacking or poorly implemented.

Superscalar Register Based Machine If tagging can be applied to stack-oriented instructions, it is argued in Chapter 1 that normal execution mechanism and technique employed in register-based machines may be equally applicable to our design. It would be useful to study a few typical register-based superscalar machines to look for useful structure and/or technique.

The case studies will be grouped into:

1. Stack based machine, reported in Section 2.3.

2. Register based machine, reported in Section 2.4.

2.3 Stack Based Architecture

Information for stack machines proved to be scarce, mainly due to fact that stack machine has fallen out of the mainstream architectures for the past few decades. Four machines have been selected for our study.

2.3.1 Burroughs Family B5000-B6700

History

Brief Information: Developed by Burroughs Corporation, starting in 1961 (for B5000)[8].

Design of Instruction Set: Pure stack instruction set which takes 0 operand for most of the instructions.

Features of Processor Architecture

- Display registers to keep track of the activation records to reflect the current lexical scope of executing program. Facilitates non-local variable accessing.
- 4 registers to store top of stack data.
- Top and base of stack tracked by registers.

High Level Programming Language Support

- Influenced by Algol60 and Cobol.
- Operating System support. E.g. Linked-list search instruction, for ease of memory management, interrupt handling mechanism.
- Tagged Memory words that describe type/meaning of a memory word, facilitate Memory Protection.

- Descriptors that can used for array access mechanism and hardware bound checking. Also simplifies dynamic array allocation.
- Activation records stored on stacks. Both static and dynamic links are kept as a linked list and maintained by hardware when procedure is entered/exited to facilitate access to parent/caller information.
- Virtual Memory Support.
- Multitasking support.
- Data Structure support.
- Allows efficient process splitting/spawning (B6500/B7500), by establishing and maintaining a tree structure that stores multiple stacks (the Saguaro Stack System). Two independent jobs/process can share part of same stack.

2.3.2 Hewlett-Packard HP3000

History

Brief Information: Developed by Hewlett Packard, in 1976[27][9].

Design of Instruction Set:

- Takes in 1 operand and assume the other operands (if any) reside on stack. Can be considered a stack/accumulator hybrid.
- A number of addressing modes.
- A few instructions that does not conform to the stack paradigm (e.g. allowing execution results to be stored directly to memory etc).
- Does not give direct access to variables declared in enclosing blocks.

High Level Programming Language Support

- Influenced by Algol60 and Burroughs Family.
- Registers to keep track of stack (top of stack in memory, top of stack in registers etc).
- General linked list traversals instruction provided.
- Activation records kept as stack.

2.3.3 Intel iAPX432

History

Brief Information: Developed by Intel Corporation, in Year 1981[7].

Design of Instruction Set:

- No user addressable registers.
- Instruction fetches its input operands offset from an object (in Memory).
- 0-3 operands, expressed in 2 parts: object selector + displacement.
- Expression evaluation carried out on operand stack.

High Level Programming Language Support

- Influenced heavily by Ada
- Based on the observation that HLP relies heavily on a particular data structure, the directed graph. E.g. Object is a node, and reference to object is an arc to this node. Implements directed graph (akin to linked list) in hardware design.

- Object-oriented representation for program execution. Several key types of object below.
- Compiled code information encapsulated by a Domain Object.
- Context of a executing procedure, includes information of addressing info (scoping), operand stack for expression evaluation, static link (enclosing block of scope), dynamic link (caller's context) etc.
- A doubly linked list of context objects is maintained, with functionality similar to activation records.
- Process Object to store information of execution state of a program, so as to facilitate suspension and resumption of process easily.
- CPU internal registers to hold the current process, context, domain object descriptor for efficient access.
- Access rights are embedded in object descriptor and enforced by hardware.
- Refinement Object that implements public/private property of object attributes.
- Caters mainly for the Ada programming language, which organize program into package (similar to class in OOP). Supports easy implementation of OOP languages.

2.3.4 INMOS transputer

History

Brief Information: Developed by INMOS (now ST Microelectronics), starting from 1984[62][39]. A number of models were developed, which can be categorized into three groups:

1. 16-bit T2 series

2. 32-bit T4 series

3. 32-bit T8 series with 64-bit IEEE 754 floating-point support

Design of Instruction Set:

- 8 bits RISC instruction set with 4 bits opcode and 4 bits operand.
- Can be extended by interpreting the operand as extra opcode bits.

Features of Processor Architecture

- A single transputer consists of a RISC sequential processor, on chip memory and a 4-ways inter-processor communication system.
- Multiple transputer can be connected in different topology to form parallel system.
- Only 3 general registers A, B and C, which are treated as stack by the instruction set (A is the stack top). Arithmetic operations are performed using A and B implicitly.
- Other than general registers, there are also a workspace memory pointer, an instruction pointer and an operand pointer which refers to the on chip memory.
- High speed on chip memory helps to overcome the limited number of general registers in the INMOS transputers.

High Level Programming Language Support

- Intended to be programmed by the OCCAM programming language.
- Occam supported concurrency and channel-based inter-process or inter-processor communication as a fundamental part of the language.

- As such, the INMOS transputers are designed specifically with this language in mind [62].

2.3.5 Java Virtual Machine and picoJava implementation

History

JVM is the virtual machine designed by Sun Microsystem to execute Java byte code programs independently across different platforms. So far, two hardware implementations have been produced by Sun Microsystem[10][11]. There are a number of hardware extensions in recent years, for example, the ARM Jazelle[63], which have moderate success in embedded devices.

Brief Information: Developed by Sun Microsystem, in 1997 (picoJava I) and 1999 (picoJava II).

Processor Architecture

Design of Instruction Set:

- Pure 0-address instruction set.
- All instructions except memory load/store instructions take 0 data addresses and operate on top of stack.
- Specific set of instructions for different data types.
- Provides instructions that access local variables in a block directly.
- A few fairly high level instructions to facilitate method invocations.
- Byte size (8 bits).

Design of CPU:

- 6 stage pipeline with 64 entry stack cache
- Instruction folding for top of stack operations, to improve speed and efficiency.
- Hardware stack drizzle unit to load/store part of memory from/to memory automatically.
- Most common used instructions in hardware, complex instructions are microcoded. Only a few very complicated ones trapped and emulated in software.

High Level Programming Language Support

- Designed specifically for JAVA
- Thread synchronization, garbage collection support in hardware.
- Supports method invocation and hiding of loads from local variables.
- Utilizes stack frame to store information about executing threads, acts as activation record.
- Operand stack size is pre-calculated and space is allocated in stack frame to facilitate suspension/resumption of threads.
- Above items gives good support to OOP in general.

2.3.6 Conclusion

Stack machines surveyed showed a few common trends:

- Stack structure is very good at supporting certain high level programming language constructs, e.g. variables scoping, function/procedure entrance/exit.
- Although receiving praise (especially from the academic field), stack machines generally perform poorly in actual sales. For example, the Intel iAPX432 was

considered as the "machine of the future" by many [7], but it failed badly to sell. This is mainly due to the fact that stack machines are much more complicated than other machines, which usually shows in slow product development, higher price and/or poorer performance.

- Because of the complexity and difficulty in speeding the execution of stack instructions, most machine architecture designers prefer the alternative design (e.g. general purpose register architecture). In those architectures, dependency detection, pipelining, super scalar execution of instructions can be done much more easily[27].

2.4 Register-Based Superscalar Architecture

Since Register-Based Architectures has been the mainstream for almost as long as the history of computer architecture, a huge number of processor designs have been proposed and implemented. To narrow our search, we only concentrate on architectures that are:

RISC-Based RISC-based architecture has the added advantage of simple and uncluttered design compared to CISC based architecture. This allows us to concentrate on the main features that are relevant.

Superscalar We have chosen to implement a superscalar stack machine. Naturally, superscalar architecture will provide us with important ideas.

Speculative Another well-developed idea on register based architecture, which would shed light on our design.

Long Life Quite a number of architectures simply fade out of the main stream after a short period of time. Though not necessarily being the best designs, long lived processor families also allow us to compare each successive generation to see the evolution of certain ideas.

2.4.1 Alpha Family

History

The **DEC Alpha** (also known as **Alpha AXP**) is a 64-bit RISC microprocessor originally developed and fabricated by Digital Equipment Corp (DEC). This architecture family is frequently touted as the proof of superiority of manual design as opposed to automated design. The Alpha chips consistently showed that manual design can lead to a simpler and cleaner architecture [39]. Besides, the **Alpha AXP** also posted excellent performance that is almost unrivaled in its generation [20]. A

cluster of 4096 Alpha Processors currently (2004) powers the 6th fastest supercomputer in the world [26]. Sadly, the Alpha AXP family tree is finally ended at **EV7** in year 2004, where **HP** (who bought Compaq, which in turn bought DEC) officially announced the end of production line.

The DEC Alpha family includes the following chips (excluding chips that were never fabricated and minor variations):

1. Alpha 21064 (EV4) in Year 1992.
2. Alpha 21164 (EV5) in Year 1995.
3. Alpha 21264 (EV6) in Year 1998.
4. Alpha 21364 (EV7) in Year 2003.

This survey is mainly based on the older and simpler Alpha 21164.

Processor Architecture

The main features of the **Alpha AXP Architecture** is summarized in [17] as a scalable RISC architecture, supporting 64-bit addresses and data types, and deeply pipelined, superscalar designs that operate with a very high clock rate. The **AXP** designers strive for simplicity over functionality, such as eliminating branch delay slots, register windows etc, in exchange for efficient superscalar implementation.

Alpha 21164

The 21164 pipeline length varies from 7 stages for integer execution to 9 stages for floating point execution, up to 12 stages for on-chip memory access and a variable number of additional stages for off-chip memory access [18]. The first 4 stages (known as instruction pipeline in AXP architecture) which deals with instruction

decoding and issuing, are the same for all instructions. Since we are interested in Superscalar technique, this would be the part that we concentrate on.

Stage *S0* (the first stage in the instruction pipeline) fetches a blocks of four instructions from instruction cache and performs preliminary decoding. Stage *S1* mainly checks for flow control instruction (branching, subroutine enter/exit), calculates the new fetch address and updates the instruction cache accordingly.

In stage *S2*, instructions are steered to an appropriate function unit, a process called *instruction slotting*[19]. The slotter process can slot all four instructions in a single cycle if the block contains a mix of integer and floating point instructions that can be issued together. In other word, this stage resolves all structural hazards and issues as many as possible instruction to Stage *S3*. The slotting appears to be similar to the VLIW packaging process, albeit the former is dynamic, the latter static.

Stage *S3* performs dynamic conflict checks on the set of instructions advanced from *S2*. Basically, this stage contains a complex register scoreboard to check for read-after-write and write-after-write register conflicts. This stage also detects function-unit-busy conflicts.

Alpha 21264

According to [21], **Alpha 21264** has similar stages to **Alpha 21164**. However, there are a few notable differences. First, register renaming is deployed to expose instruction parallelism. This is stated as the fundamental to the 21264's out-of-order techniques.

Also, advanced branch prediction is added. A number of branch predictions methods are known that work pretty well. However, the accuracy of prediction is not universal and different algorithms work well on different types of branches. Hence, instead of using a fixed prediction algorithm, the **21264** employs a hybrid approach that combine two different algorithms, picking the better one dynamically[20]. It

is important to note that whenever prediction fails (the wrong path is taken), the **21264** enters a *mispredict trap*, which basically stops all in-flight instructions, flushes the instruction pipeline and restarts from the correct path.

2.4.2 PowerPC Family

History

The **PowerPC** (**P**ower **C**omputing) began life from the IBM's **POWER** (Power Optimization With Enhanced RISC) architecture, which was introduced with the RISC System/6000 in early1990 [39]. This architecture specification is the result of the three-way collaboration **AIM**, which involve three big names in the industry, **A**pple, **I**BM and **M**otorola. The first chip of the PowerPC family, **601** was released in Year 1994. A number of variations on the basic chip were later released as **PowerPC 602**, **603** and **604**. The first 64bit implementation, the **620**, was released in Year 1995. Later chips were used by the Apple Macintosh machine:

1. **750** (PowerPC G3) in Year 1997
2. **7400** (PowerPC G4) in Year 1999
3. **970** (PowerPC G5) in Year 2003

Besides from the Apple Macintosh machine, PowerPC chips are also a favorite choice for embedded computer designers, in particular the **PowerPC 620**.

Processor Architecture

The original **POWER** architecture incorporated common characteristics for RISC architectures: fixed length instructions, load/store only memory access, separate registers for integer and floating point operations. Also, the **POWER** architectures is functionally partitioned, which facilitated the implementation of superscalar designs[23].

When the **PowerPC** architecture was extended into the 64bits realm, there were several major changes:

1. The designers removed niche instructions that were deemed too complicated.
2. A set of simpler, single precision floating-point operations were added.
3. A more flexible memory model, allows software to specify how the system performs memory accesses.

PowerPC 620

The **620** pipeline has 5 stages for integer instruction: fetch, dispatch, execute, complete and write-back. For other type of instructions, a variable number of stages is needed, briefly Floating Point Instruction takes 8, Load Instruction takes 7, Store Instruction takes 9 and Branch Instruction takes 4. The main execution characteristic of the **620** is that Instructions are dispatched in program order, are executed out-of-order, and are completed in order [24]. As with the **Alpha AXP** architecture, we are concerned mainly with the fetch and dispatch stage.

The fetch stage access the instruction cache to bring up to 4 instructions into a 8-entry FIFO buffer. The first four (the older four) are referred to as dispatch buffer which is accessed by the dispatch stage directly, and the other four entries are the instruction buffer. The **620** also associates seven pre-decode bits with each instruction which contains executions information like GPR[1] file usage, execution unit needed etc. These pre-decode bits eliminates the need for a separate decode pipeline stage.

During each cycle, the dispatch stage examines the four instructions in the dispatch buffer and attempts to dispatch them to reservation stations in appropriate execution units. Inter-instruction dependencies are identified and an attempt is made to read the source operand from the architectural register files or from the

[1]General Purpose Register

rename buffers. Note that the **620** assigns rename buffer for results produced by an instruction. Subsequent instructions that depends on this result are given a tag to identify that particular rename buffer.

Since execution can occur out-of-order, a 16-entry reorder buffer is used to keep track of the instruction order (the program order) as well as the state of the instructions. A reorder buffer entry is associated with each dispatched instruction. A completion flag is recorded when the instruction finished its execution, which will be used to ensure program order.

The **620** employs a **Branch History Table** method for branch prediction. Prediction is made based on the past history of a particular branch instruction. Static branch prediction, where a bit in the instruction along with the direction of the branch determines the outcome of the prediction, can be turned on by software to take the place of the BHT method[24].

Speculative execution is made possible by maintaining the results in temporary storage, such as rename buffers, reorder buffers and shadow registers (registers that are invisible/unaccessible to programmer). When misprediction occurs, the completion logic kicks in and purges all speculative results.

2.4.3 Conclusion

The superscalar machines survey has taught us several important lessons:

1. **Multi-Issue (Superscalar) Execution** depends on the architecture ability to disentangle the static and dynamic dependencies between instructions. Tagging or renaming is a widely employed solution.

2. **Instruction Supply** is important in Superscalar design. Because of the multiple instructions dispatched every clock cycle, a constant supply of of instructions from the fetching stage is vital.

3. **Branch Prediction** in a superscalar setting is viable. However, two problems must be solved efficiently for this technique to be worthwhile: keeping track of speculative results and cleaning up misprediction.

These considerations have considerably influence our design for **SAFA**.

2.5 Summary

The two surveys outlined in this chapter have given us much material to help design our architecture. Although the architectures surveyed comes from radically different design, we found that there are complementing traits between the two groups. The stack machines survey has shown very clearly the disadvantages of the stack oriented architecture, but at the same time points out the undeniable fact that stack structure is one of most utilized data structure in high level language. The register machines survey, on the other hand, showed that there are some very well studied techniques that can improve instruction level parallelism. However, these machines have little specialized support for high level language. Guided by these observations, we have laid out our design of a new stack machine **SAFA**, in search for a stack architecture that provides good high level language support and performs well at the machine level. The architectural details which will be covered in the next few chapters.

Chapter 3

High Level Language Support

The original purpose of computer programming languages is to provide a more human friendly interface leading to the lowest level of hardware instructions. In the beginning, Assembly Languages were basically thinly veiled human readable machine code. The close relationship between the language and the hardware reflects the need of maximum efficiency in execution since processing power comes at a premium. However, that also renders the language to be restricted, awkward to use and machine dependent.

As computing power increased, the possibility of allowing more flexible and expressive programming language emerged. Programming language design shifted from machine-centric to human-centric. These languages, usually termed as High Level Programming Languages to distinguish them from the low level counterparts, represent richer paradigms to organize the data and control of a program. The HLPs, being further away from the machine code conceptually, relies on compilers to translate them into machine executable code. This frees the HLPs from the restriction of any single machine and further enhances the flexibility of the language.

With the growing importance of HLPs, new design in computer architecture must take high level language support into consideration. Providing specialized hardware for high level construct has greatly influenced the design of some computer

architectures. For example, from the older schools of architecture we have the Burrough architecture, which utilized the display registers (Section 2.3.1) for efficient procedure invocation. For the more recent architectures, the Intel Pentium x86 architecture provides the MMX (multimedia extension) for video, computer graphics processing. There are even attempts to based computer architecture on high level language paradigm, for example, the Intel AIX (Section 2.3.3) which actually embodies object oriented concepts.

For SAFA, we consider the HLP support in two main areas, control flow and data structure, which are discussed in the following sections.

3.1 Control Flow

There are two major forms of control flow in most HLPs, procedure[1] activation, and selective execution (branching and looping).

3.1.1 Procedure Activation

In HLPs, program code is usually organized into manageable modules e.g. *Procedures/Functions* in procedural imperative languages, *Methods* in object-oriented languages to allow better modularity and code re-usability. Hence, one of the most frequently used operations in HLPs is to transfer the thread of control from one module to another, e.g. function call, method invocation etc. According to [27], in a typical PASCAL program, the frequency of procedure activation (function call) can be as high as 15%, the third highest after assignment (45%) and branching (29%). Compared to function calls, assignment and *if-then-else* are relatively simpler operations . These operations can easily be mapped to a few low level instructions. By providing maximum efficiency in the instruction level, these operations can then be executed with satisfactory speed. On the other hand, function calls require more

[1] also known as module, coroutine or function

complicated mechanism for efficient execution.

There are two major tasks during procedure activation. In addition to the mechanism of transferring and bookkeeping of threads of control, accessing scoped variable also requires attention. For procedural HLPs, there are at least two level of scoping, one for the so called *Global Variables* (accessible in the whole program), and *Local Variables* (specific to each module)[2]. For object oriented HLPs, there are at least two more scopes: the class variables (specific to each class) and the object variables (specific to each object, i.e. instances of each class). The extra scopes in object oriented HLPs can be taken as an extension from the procedural counterparts, where class variable is similar to global variable in a class, and object variable is similar to local variable specific to each object. We will first consider the scoped variables in procedural HLPs.

During the execution of a procedure, both type of variables may be accessed. However, a local variable in a procedure is not visible or accessible from the outside. Hence, an efficient mechanism for accessing variables in different scopes, and also access protection is needed.

The information needed for the activation of a procedure is usually collected in a record called an *activation record* or *stack frame*[29]. This record is created at runtime when a procedure is about to be entered. Usually it is allocated on the memory stack or heap according to different designs. A number of the more important information items kept is listed as follows:

Control Link Also known as dynamic link, points to the stack frame of the run-time caller.

Access Link Also called as static link, points to the lexical host (parent) of the current procedure.

[2]Languages that support nested procedures will have more scoping levels, like Pascal, Algol60 etc

Saved State any process state information of the caller procedure, used for resumption of the caller procedure.

Parameters Actual parameters for the activation of the current procedure.

Local Variables Storage space for local variables of the current procedure.

The two links: dynamic and static, play important role in procedure activation. By following the control link, the return address and information of the caller relevant to the procedure can be determined when the current procedure finishes. By using the access link(s), the non-local variables in parent or global scope can be accessed. Since these two links are utilized heavily during execution, it is natural for architecture designer to provide hardware support for them. For most general purpose register machines, only a single register is used to store the address of the activation record. This register is given different name in different architecture, for example, the frame pointer in the x86 architecture. The two links are then accessed via offsets from this register.

Although a single register is sufficient for bookkeeping purpose, accessing out of scope variables would require extra computation since the access links need to be traversed repeatedly to reach the required scope. Other architecture designers tried to provide more sophisticated support to cut down the execution time. For example, the Burroughs line provides an array of *32 Display Registers*, which is used to store the addresses of activation records at each lexical scope level[37]. The content of each register is adjusted automatically upon entering/exiting to/from a procedure. Accessing scoped variable is then translated into a single access of the register array at the required lexical scope level[3].

SAFA follows a similar scheme, but with the following differences:

1. Less registers. The total number of registers (named as *frame register*) used for bookkeeping the activation frames is only four. Each of the four frame

[3] to be precise, an offset must also be added

registers has a specific role, which will be covered later.

2. More information is stored in each register. Most of the other architectures opt to store only the address of the starting (or ending) of the activation records in the registers. *SAFA* choose to encode more information, which includes the starting address, the total number of data elements and size of elements among others[4].

In most high level programming languages, there are four activation frames are of greater interest during execution. These four frames are:

Global Frame describes global information (the outermost lexical scope).

Caller Frame describes the caller of the current procedure. Also the destination of return when current procedure finishes.

Host Frame describes the lexical parent of current procedure, mostly used for accessing non local variables.

Own Frame describes the current running procedure.

As for languages that allows nested procedure, the information stored in the host frame can be used to locate the predecessor frames for non-local variables in various lexical levels. However, this scheme would be slower than other implementations, for example, the display registers scheme, since multiple traversals are needed. More frequent access to a non-local frame can however be accommodated by utilizing additional frame registers, as will be discussed later.

The decision of limiting the frame registers to 4 for procedure activation can be viewed as a compromise between flexibility and hardware economy. By providing more frame registers, the CPU may give better performance *only* when dealing with nested procedure languages. However, the complexity and size of the frame registers will certainly take their toll on CPU space and circuitry complexity. On the other

[4]other fields will be discussed in the next section

hand, the added information of each register provides added benefits like hardware detection of illegal access (by using the limit of each frame) and other checking with reasonable amount of register complexity.

To give a clearer picture of the scheme adopted by *SAFA*, consider the following *Pascal*-like program.

```
Program P
    Procedure A()
        Procedure B()
        begin
            ....          // Point Beta
        end               // end of B
    begin
        B()
    end                   // end of A

    Procedure C()
    begin
        ...               // Point Alpha
        A()
    end                   // end of C

begin                     // main procedure
    C()
end                       // end of P
```

Upon the initiation of the program **P**, the four frame registers have the following information:

Frame Register	Pointing to
Global	P
Caller	nil
Host	nil
Own	P

Since we are executing the main procedure, the current frame would point to **P**, which is also the global frame. When we reach *Point Alpha* in procedure **C**, the frame registers would now contain:

Frame Register	Pointing to
Global	P
Caller	P
Host	P
Own	C

Since the main procedure called **C**, the Caller Frame points to frame **P**. **P** also happens to be the lexical parent of **C**. When the *Point Beta* in procedure **B** is reached, we will now have:

Frame Register	Pointing to
Global	P
Caller	A
Host	A
Own	B

The procedure **A** is the caller as well as the host of procedure **B**. By using the Global Frame Register and the Host Frame Register, the procedure **B** can easily access global variables (defined in the program scope) or parent's variables (defined in procedure **A**).

Since the content of the frame registers is managed automatically by hardware upon procedure entry and exit, compiler writers can utilize the machine instructions provided to gives optimized access to variables in different scopes.

3.1.2 Repetitive Execution with Counter

Repetitive execution, commonly known as looping, represents another kind of major control flow in HLPs. It is no exaggeration to say most programs spend most time looping. For this section, we concentrate on a particular subclass of repetitive execution: loops with counter. Most loops will ends the execution upon meeting

some condition. For loops with counter, the ending condition is specified with numerical values, usually in the form of checking a variable against a fixed upper-bound. For example:

```
For I = 1 to 10     //Loop Header
    Do Something    //Loop Body
```

The *for-loop* above terminates when the variable I exceeds 10. This particular form of loop enjoyed heated research about optimizing it mainly because of its frequent appearance, usually to do with array/matrix manipulation. By examining the example, we can see that there are two major components in a loop with counter. The loop header is basically to maintain bookkeeping information about the number of iterations. The loop body, on the other hand, contains the main computations. Various methods are proposed to optimize each of the components.

In this section, we concentrate on the possible optimization of the loop header. For stack-oriented machine, the updating and validating of the loop header can be performed as follows:

```
//Assume:
//      I is a local variable in stack frame
//      I = 1 initially

start:
    Load I
    Load 10
    If_Greater done     //terminates the loop
    ...
    ...                 //loop body
    Load I              //load I again
    Add                 //Increment I
    Store I             //store the new value
    Goto start
done:
```

The multiple memory operations for updating I render the code above clumsy and slow. The *JAVA* programming language, trying to overcome this prob-

lem, specifically includes an instruction ***iinc***, which directly update a local variable in stack frame[5].

The design of the frame register in SAFA, although not aiming at this problem, provides a different kind of optimization. Before we delve any deeper, the complete description of all the fields in a frame register will be given. A frame register consists of five fields:

Field	**Usage**
Base	Starting address of a block of consecutive memory locations
Interval	Number of elements skipped for each iteration
Index	The position of the current element accessed
Limit	Upper bound of the memory region
Size	Size (in bytes) of each of the element

The three fields: *Interval*, *Limit*, *Index* was originally developed for data structures in HLPs (discussed in the next section). In loops with counters, the same three fields can be borrowed to keep track of the looping information. With the help of these fields, a simple loop in SAFA can be written as:

```
//Assume:
//     The fields in FR1 (frame register 1) is used.
//     limit is set to 10
//     index is set to 1
//     Interval is set to 0

start:
    Cmp_Index_Limit FR1 //compare the limit and index
    If_Greater done     //terminates the loop
    ...
    ...                 //loop body
    Increment_Index FR1 //increase index
    Goto start
done:
```

[5] *iinc* is more like a macro, it itself does not represent any execution speed up

The fields *Interval* can be used for loop counter that jump in stride. For example:

```
For I = 1 to 10, Step 2 //Loop Header, I = 1,3,5,7...
    Do Something        //Loop Body
```

Since the *Increment_Index* instruction actually performs

```
Index = Index + (Interval + 1)
```

The loop can be represented by the exact same code, except the *Interval* is changed to 1.

```
//Assume:
//     The fields in FR1 (frame register 1) is used.
//     limit is set to 10
//     index is set to 1
//     Interval is set to 1

start:
    Cmp_Index_Limit FR1 //compare the limit and index
    If_Greater done     //terminates the loop
    ...
    ...                 //loop body
    Increment_Index FR1 //increase index
    Goto start
done:
```

If the update of the variable does not follow a fixed stride, then the index can be loaded on the stack for manipulation directly. For example:

```
//Assume:
//     The fields in FR1 (frame register 1) is used.
//     limit is set to 10
//     index is set to 1

start:
    Cmp_Index_Limit FR1 //compare the limit and index
    If_Greater done     //terminates the loop
    ...
    ...                 //loop body
    Load_Index FR1      //load index to stack
    ...                 //manipulate the index
    Store_Index FR1     //store the index back to FR1
    Goto start
done:
```

There are two advantages of using the frame registers for bookkeeping in a loop. Firstly, the frame register reside in the CPU and offers much better access speed. Secondly, the resultant code is more compact. Admittedly, this would be an overkill if the three fields are specifically designed just for handling this type of loops. In this case however, the three fields are actually designed for multiple purposes (Section 3.2). So, this can be viewed as "side benefits" of the frame registers instead of the main reason.

The limitation of using the frame registers in this way can be summarized as follows:

1. The upper-bound of the *limit* and *index* fields is 65535.

2. The upper-bound of the *Interval* is 255.

3.2 Data Structure

The two major kind of data structures in HLPs are:

Array Easily the most commonly used data structure, provide a indexed access to a collection of homogeneous data in consecutive memory space. The number of elements in an array is fixed at the point of array creation.

Linked List A more flexible data structure, provide the ability to build an open-ended collection of possibly heterogeneous data via memory pointer (memory address). Memory dereferencing is used to traverse and access the desired data item.

Next, the two major data structures above will be inspected in the light of SAFA implementation.

3.2.1 Array

Arrays are frequently used for mathematical operations (e.g. matrix manipulations, fast Fourier transform etc), or as the building block for other data structure (e.g. stack, queues etc). This is also one of the area where the stack architectures of the past draw most criticism. Array indexing, i.e. accessing the elements in an array, requires frequent operation of a particular value (the base of the array), which is not suitable for a stack. Consider the case where the elements in an array are accessed sequentially, we will need to compute the location of the next element in the memory by computing $Base + (SizeOfElement \times PositionOfElement)$. In a GPR machine, these operations can be optimized by keeping the base of the array in a register and invoking addressing modes designed for array access. However, in a stack machine, the base address has to be loaded/moved to the top of the stack for each of the operations, resulting in great overhead.

Frame registers in SAFA can be used to cope with the array indexing problem. Recall that the five fields of a frame register capture the essential information of a memory region with consecutive elements: the starting address, the number and size of elements in the region, the current index, limit and interval of the element accessed. Any data structure with similar properties can be represented efficiently by a frame register. In this case, array would be a prime candidate.

With the hardware (frame register) in place, we will first look at the associated machine instructions. Major type of operations are listed below:

1. Load current array element to stack
2. Store element at top of the stack to the current position
3. Increase the index by one stride (i.e. take interval into consideration) by using the formula $NewIndex = Index + Interval + 1$.
4. Decrease the index by one stride.
5. Compare index to limit and leaves result on stack.

As can be seen, these operations provides a good foundation for building complex array accessing operations. We will briefly study some of these operations next.

Array of Records

The obvious extension from basic array of simple data items is array of records. A record is just a enumeration of heterogeneous data. For example, a student record would contain the name, matriculation number, age, and test score. To represent a cohort of students, an array of student records would be sufficient.

Assume the storage requirement for the record is as follows:

- Name = 20 bytes
- Matric Number = 4 bytes
- Age = 2 bytes
- Test score = 2 bytes

For an array of 20 students, we can calculate the average score by performing the following operations:

```
Total = 0
For I = 0 to 19
Total = Total + Student[I].Score
Average = Total / 20
```

The access of score *Student[I].Score* would require codes as follows in a stack-oriented architecture:

```
Load I
Load 28  //Size of a record
Multiply
Load 26 //Offset to test score
Add
Load Student //the base address of the array
Add
Derefenrece // Memory access
```

Surprisingly, for a GPR machine, the code is almost as wordy:

```
Multiply R1, 28, R2 //R1 stores I, R2 = R1* 28
Add 26, R2, R2 //Offset R2 = R2 + 26
Add R3, R2, R2 //R3 stores base address, R2 = R3 + R2
Load [R2], R4 //Load the score to register R4
```

For SAFA, there are at least two ways to handle this scenario and both are arguably better than either version described. The first version represents the student array as a array of 560 (28*20) elements, where each element is a single

byte. The index is initialized to 26, which is the test score for the first student. The interval can then be set to 28 (the length of 1 record). With these information, along with the base address stored in a frame register, the access code is much simpler:

```
Load_Halfword FR1   //read 2 bytes from frame register 1
Index_Increment FR1 //advance index, add 28 (size * interval)
```

The second version is based on the simple observation that each student records has the same layout. So the same offset will get the test score of any student provided the base address is adjusted accordingly. So, a frame register is used to represent a single student record, with size set to 28 bytes, and index set to 26 as offset for test score. The basic idea is to change the base of the second frame register to the next record for each iteration as explained. The SAFA pseudo code is as follows:

```
Load_Halfword FR1   //load halfword from current index of FR1
Add_Base FR1,28     //add 28 to the base address of FR1
```

Obviously, the first solution is faster and more economic in term of code size. However, the second version allows more general access pattern. For example, if random access is needed, i.e. no regular pattern to the record desired, then the second version can easily cope with the change by changing the base accordingly. More importantly, however, is the clear advantage of frame registers over its counterpart in other architectures.

3.2.2 Linked List

As opposed to array, where number of elements is fixed, linked list provides more flexibility by allowing easy insertion and deletion. Also, the number of elements can be extended at runtime as long as there are enough memory space. These advantages come at a price, elements in linked list must be allocated separately, and linked together via pointers (memory addresses). With the elements spread out in memory space, it is harder to traverse in linked list compared to array.

Consider the student result example in Section 3.2.1, we can add one more data item, a pointer to the next element, to accommodate linked list. The high level pseudo code to traverse and collect the test scores of all students is as follows:

```
Total = 0
StudentNum = 0
CurStudent = FirstStudent       //CurStudent is a pointer
While (CurStudent != NULL)
Total = Total + CurStudent->Score
    CurStudent = CurStudent->Next   //points to next student
    StudentNum = StudentNum + 1
Average = Total / StudentNum
```

We will look at both the access of the field *test score* as well as the more interesting and time consuming code, the traversing of records.

For a stack-oriented machine, the psuedocode is:

```
Load CurStudent     //Assume CurStudent stored as local variable
Load 26             //Offset to test score
Derefenrece         //Memory access

....                //Accumulate the total

Load CurStudent
Load 28             //Offset to test score
Add
Derefenrece         //Memory access
Store CurStudent
```

The weakness of stack machine is compounded by the need to repeatedly read in *CurStudent*. The GPR machine, on the other hand, fare slightly better by the ability to retain a frequently used value in register:

```
Add 26, R1, R2 //R2 stores the address. Offset R2 = R1 + 26
Load [R2],R3 //Load the score

....                //Accumulate the total

Add 28, R1, R2      //Next record, R2 = R1 + 28
Load [R2],R1        //load the next address into R1
```

For SAFA, the solution is based on the second version of code discussed in the array section. As a brief summary, a frame register is used to represent a single student record, with size set to 32 bytes (including the pointer), and index set to 26 as offset for test score. The basic idea is to change the base of the frame register to the next record in the linked list. The SAFA pseudo code is as follows:

```
Load_Halfword FR1       //load halfword from current index of FR1
....                    //Accumulate the total
Load_Word_Offset FR1,28 //load word from FR1 + 28, the next base
Set_Base FR1         //Set the current base.
```

The SAFA code is more compact compared to all other versions. Its improvement over conventional stack code is clear.

3.3 Object Oriented Language

Arguably, the object oriented paradigm is a significant advance in programming language design. By integrating data and the functions that operate on them into a tight, access controlled package, object oriented languages allows cleaner code organization compared to the conventional procedural languages. As its progenies like C++, CLOS, Java etc gain wide acceptance, it would be beneficial for a CPU architecture to provide support for Object Oriented Languages (OOLs).

A full treatment of this subject under the context of CPU architecture support is not done due to the scope and depth of the subject. We will concentrate on the two most important features in the OOLs:

- The representation of an object.

- The dynamic dispatching for polymorphic methods.

3.3.1 Object Representation

At the lowest level, the OOLs can be viewed as a extended version of simple record (or structure). In addition to data values, methods[6] are also grouped into the same representation. The declaration of this information is usually termed as *class*, where an actual instantiation a *class* is known as *object*. Consider the following class in a pseudo language:

```
class Simple
{
    int i_;
    void Method()
    {
        int local;

        local = i_ + 3;
        i_ = local *2;
    }
}
```

Since each object of the class *Simple* has its own version of the variable *i_* (known as object variables), the invocation of the method *Method* must explicitly state the associated object. For example:

```
Simple objSimple;

objSimple.Method();     //Method invocation
```

The code above shows one possible syntax: *Method* is invoked in the context of *objSimple*.

[6] similar to functions or procedure

For actual implementation of such languages, the objects are realized as records, which contains the variables (*i_* in the example). The method is simply converted to a normal procedure invocation, except that the object reference (the memory address of object) is passed in as parameter. As discussed previously (Section 3.1.1), in a procedural framework, the frame registers can be set up to access the stack frames of *Global*, *Caller*, *Host* and *Own*. In OOLs framework, the *Global* and *Caller* frames remained unchanged, while the *Host* frame can be altered to point to the associated *Object*. The frame pointer *Own* remains pointing to the stack frame of the current running method.

We will now give pseudo codes for the *Method()* on stack-oriented machine, GPR machine and also SAFA for comparison.

For stack-oriented machine:

```
Load This           //"This" is the object reference
Load 0              //offset for the object variable i_
Dereference         //load i_ from memory
Load 3
Add
Store local         //"local" is store as local variable
Load local
Load 2
Multiply            //the final result now on stack
Load This
Load 0              //offset for the object variable i_
Store_to_Memory     //store the final result to i_
```

For GPR machine:

```
//Assume: R1 = object reference
//        R2 = "local", memory location is FP+4
//        FP = frame pointer
//        R1+0 = i_ of this object

Load 0[R1],R3       //0[R1] = R1 + 0 (offset)
Add R3,3,R2
Store R2,4[FP]      //4[FP] = FP + 4 (offset)
Multiply R2,2,R3
Store R3,0[R1]      //Store the final result to i_
```

For SAFA:

```
//Assume: Host FP (HFP) = object reference
//        Own FP  (OFP) = method's stack frame
//        OFP + 4 = "local" memory location

Load_Word_Offset HFP,0  //Load i_
Load 3
Add
Store_Word_Offset OFP,4 //Store the result to "local"
Load_Word_Offset OFP,4
Load 2
Multiply
Store_Word_Offset HFP,0  //Store i_
```

In this case, the GPR and SAFA code is similar in term of execution steps, i.e. fewer than the conventional stack-oriented code. However, the SAFA code size, typical of a stack oriented code, is more compact as most instructions are 0-operand.

3.3.2 Dynamic Method Dispatching

The other important characteristic of OOLs is inheritance and polymorphism. In particular, method polymorphism, which gives OOLs a powerful feature over other languages, pose a serious challenge to language implementor. The premise of method polymorphism is simple: "The most specialized method according to the **class type of the object** must be invoked". A method is specialized if a subclass (children

class) implements its own version of a method that exist in the superclass (parent class). e.g.

```
class Parent
{
    void Method();

}

//Class Children inherit from Class Parent
class Children: public Parent
{
    void Method();    //specialized Method

}
```

For this simple class declaration, we have two different versions of *Method()* which requires *Method Resolution*. The resolution of method invocation in the form *Obj.Method()* is simple: the type of the object *Obj* selects the correct version of the method *Method*. This can done statically during compilation.

However, since reference to the object of *Parent* class are allowed to hold object reference of *Children* class (also known as subtype property), the method invocation resolution of the form *ObjRef->Method()*[7] is much harder. Consider the following two invocations:

```
Parent* ObjRef;
Parent p;

ObjRef = &p;         //ObjRef points to 'p' object
ObjRef->Method();    //Parent's Method() should be invoked

Children c;
ObjRef = &c;         //ObjRef points to 'c' object
ObjRef->Method();    //Children's Method() should be invoked
```

[7]Using C++ like pseudo code, where *ObjRef* is a pointer to object

The second invocation must call the *Method()* defined by the *Children* class[8] to conform with polymorphism. This resolution cannot be done statically, hence the name "dynamic method dispatching".

Method dispatching is a widely studied field. We consider only a subclass of the problem *single dispatching* which is implemented in the languages like C++, Java etc, where only methods with the same parameters type and return type (i.e. same method signature) are considered for dynamic method dispatching. The other type of dispatching *Multiple Dispatching* deals with polymorphism of different method signatures which is only supported in a few languages like CLOS.

In Java Virtual Machine implementation, dynamic dispatching is usually[9] achieved by *method table*[15]. The method table is a class specific data structure, usually implemented as array, that contains references to compiled method code. Since the method resolution for a class is *fixed*, this information can be generated during compilation time and will be loaded as part of the executable. An object now contains object variables as discussed previously *and also* the reference to the associated method table. When a method is invoked, the object reference is followed to reach the object representation. Then, the method table is used to determine the address of the appropriate method code.

Consider a slightly more elaborated example:

[8] For C++, the word *virtual* should be added in front of the method declaration

[9] The JVM specification did not give specific design on this

```
class Parent
{
    void Method();
    void MethodParent();
}

//Class Children inherit from Class Parent
class Children: public Parent
{
    void Method();    //specialized Method
    void MethodChildren();

    //Note: MethodParent() is also inherited
}
```

With the *method table* implementation, the object p and c in the previous example can be represented as shown in Figure 3.1.

Thus, the dynamic method dispatching boils down to the following requirements:

1. Representation of an object.
2. Dereferencing an object pointer.
3. Go through the method table to find the method code address.

As discussed, an object can be represented with the use of a frame register in SAFA. Since the method table is basically an array of records, the searching can be performed with the help of frame register(Section 3.2.1). To add the method table to an object representation, information like size of elements, number of elements must be stored along with other object information:

A *method record* should contains at least two parts:

1. The method signature, usually represented as string. In this example, an integer containing unique method index is used instead to facilitate discussion. Assume to be 4 bytes.

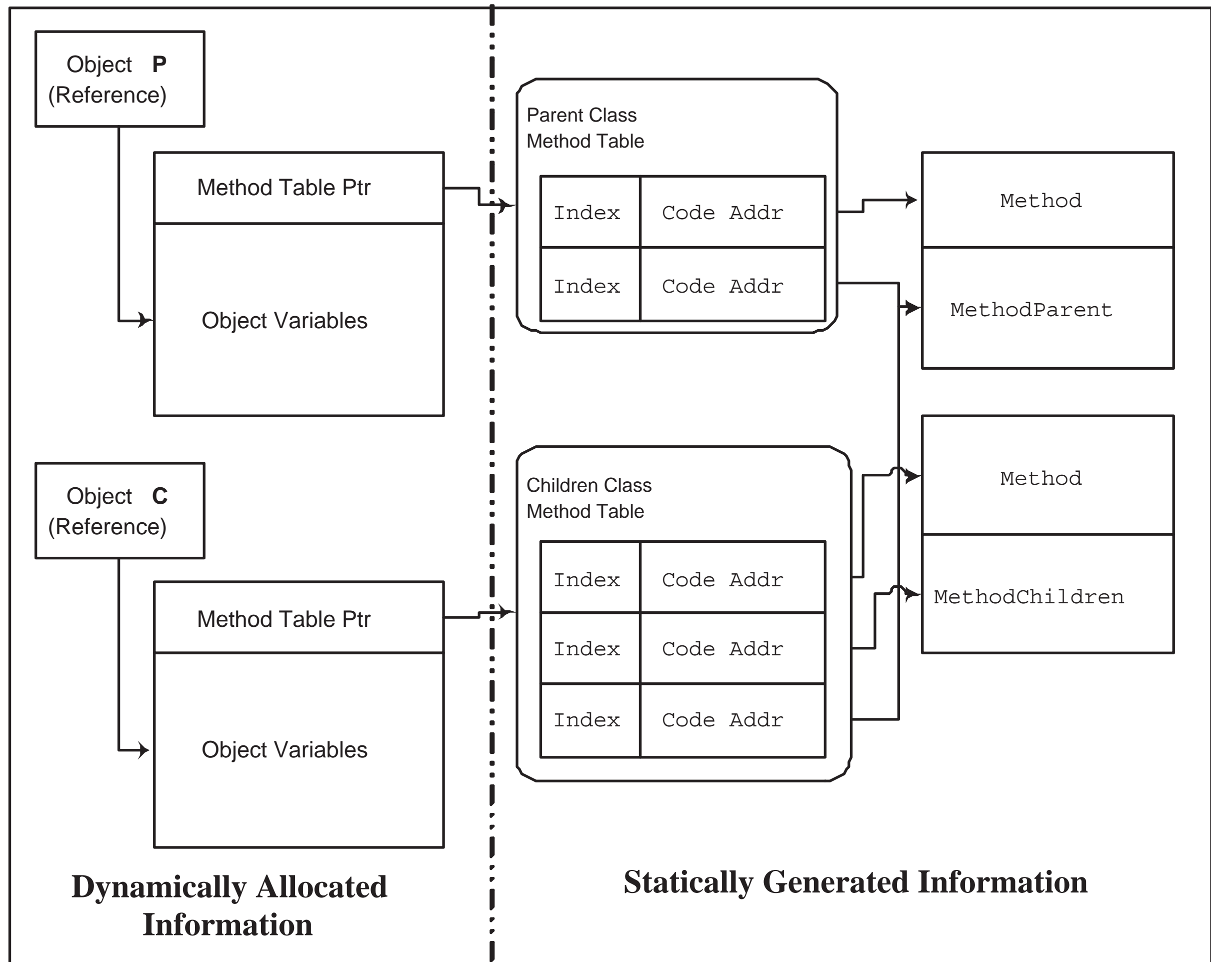


Figure 3.1: Dynamic Dispatching in OOLs

2. The memory address of the associated compiled method code. Assume to be 4 bytes.

Dynamic dispatching can be performed in SAFA via the following (psuedocode):

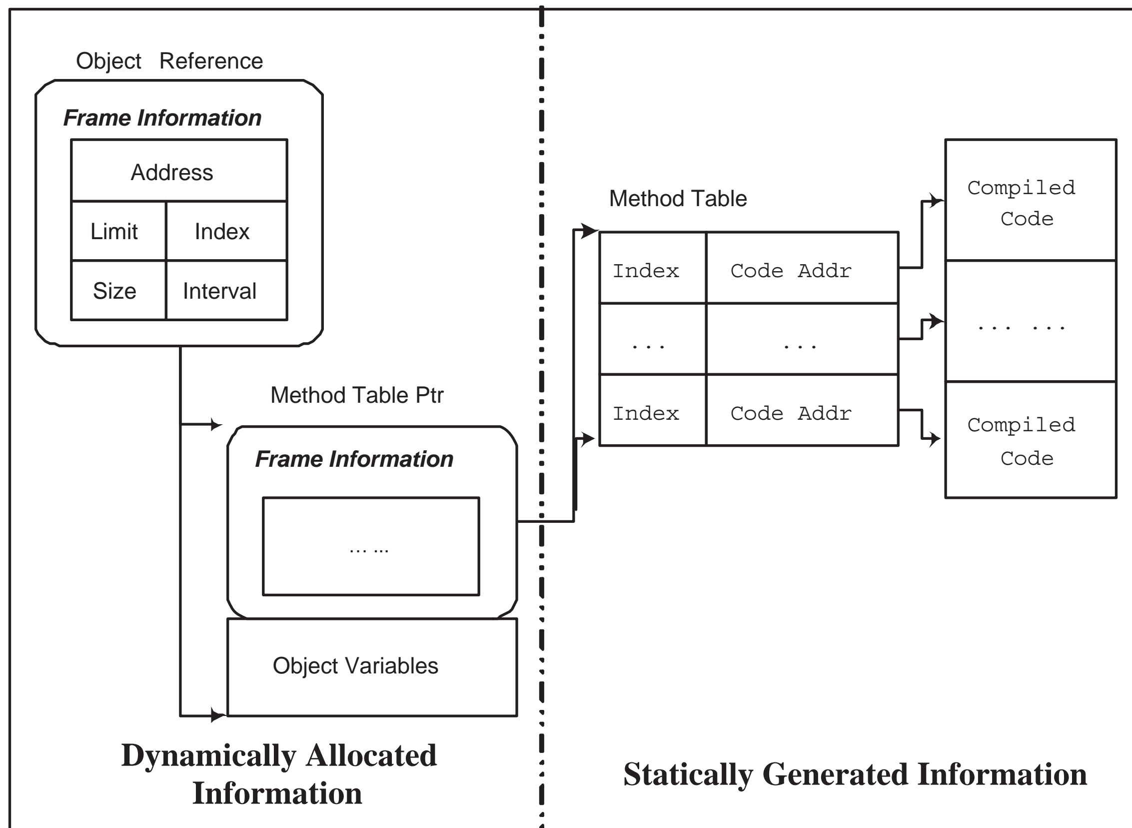


Figure 3.2: Object Representation in SAFA

```
//Assume:
//  FR1 = object reference
//      FR1 + 0 = Frame information for method table

//  FR2 = method table reference

//  Method table contains records of:
//      Offset 0:    Method Index
//      Offset 4:    Method Address
//  The desired MethodIndex, MI is on stack

    Load 0
    Load_Next_FRM FR1      //Load frame info for method table
    Store_FRM_Info FR2
    Duplicate              //duplicate MI for comparison
    Load_Word_Offset FR2,0 //load the method index
    If_equal done
    Add_Base FR2,8        //Advance FR2 to the next record
done:
    Load_Word_Offset FR2,4
```

As can be seen, most problems in OOLs involved some basic data structures like record and array. With good support for these structures in SAFA, the more advanced features can be implemented with little hassle.

3.4 Additional Benefits of Frame Register

In this section, several additional benefits of employing frame register are discussed.

3.4.1 Context Sensitivity

Context sensitivity, also known as context awareness, is the ability to understand and react accordingly to the environment. Just as a multi-meaning English word may only be interpreted correctly if we know the full sentence, computer instruction must also be executed according to the context. In most machines, the context is explicitly recorded/specified at any time. For example, from the stack frame, we would know the parameters, local variables, caller of the current executing procedure.

Since instructions are grouped together logically, it is clear that the context of adjacent instructions would most likely be the same. For example, consider the following code in a GPR machine:

```
                    //Calculating R1 = (4+R1) * 10
Add R1,4,R1
Multiply R1,10,R1
....
```

The R1 is repeatedly used, it would be useful if the instructions “know” implicitly that R1 is the register (the context) they should be working for. A context sensitive version of the same code would read:

```
                    //Calculating R1 = (4+R1) * 10
Add R1,4            //"know" that destination is R1
Multiply 10         //"know" that source and destination is R1
....
```

However, context sensitive instructions are not realistic in GPR machines for several reasons:

- The size of instruction is usually fixed in GPR machine. There is no real push to save instruction space.
- Context sensitive instruction is not needed, since the context is always explicitly spell out in the instruction.
- The register usage for adjacent instructions may be wildly different, although they form the same context.

The last point shows that it is important to capture a larger context that only changed slowly over time for context sensitivity instructions. In SAFA, the target is clearly instructions that deal with frame registers. Recall the discussion in the last few sections, frame registers are used to represent large data structure. The nature of these data structure, as well as the relatively higher cost of setting up a frame register, indicates that frame register instructions would be prime target for context awareness.

Consider the case of scaling the array elements by a factor:

```
//Scale all elements by 10
For I = 0 to 9
    Array[I] = Array[I] * 10
```

In SAFA, with the help of frame register, the code to scale one element can be written as:

```
//Assume:
//      FR1 = array reference

1.start:
2.    Load_Element FR1
3.    Load 10
4.    Multiply
5.    Store_Element FR1
6.    Increment_Index FR1     //advance to the next element
7.    Cmp_Index_Limit FR1     //compare the limit and index
8.    if_not_equal start      //jump back to start
```

Lines 2,5,6 and *7* all operates on the same frame register *FR1*. For line 2, it serves as a "trigger" that setup the context so that subsequent frame register instructions can follow suit. So, if the instructions at *line 6* and *7* (*line 5* is excluded for the reason discussed next) can be made context sensitive, then the code can be shortened to:

```
//Assume:
//      FR1 = array reference

1.start:
2.    Load_Element FR1
3.    Load 10
4.    Multiply
5.    Store_Element FR1
6.    Increment_Index       //advance to the next element of FR1
7.    Cmp_Index_Limit       //compare the limit and index of FR1
8.    if_not_equal start    //jump back to start
```

As opposed to the GPR scenario, context sensitivity in SAFA is worthwhile because:

- Frames in SAFA represent larger and more stable context that stay the same for a number of instructions.

- SAFA is a byte code architecture, instruction space is limited.

Since a context in SAFA is always referring to a particular frame, the context sensitivity is achieved via the following:

1. Two *Frame Register Pointers*: ***Current Frame Pointer*** *(CFP)* and ***Previous Frame Pointer*** *(PFP)*. A frame pointer simply records the Frame Register number, thus "pointing" to a particular frame register. Any context changing instruction, like loading element, setting frame number explicitly will change the CFP. While PFP is simply the previous value of CFP.

2. Other non-context changing frame register instructions are assumed to be implicitly acting on the frame register referred by CFP. These instructions are now aware of the context they are in.

An important decision for context sensitive instructions is to determine which are the instructions that trigger the change of context. Obviously, explicit triggers like "Set the context to Frame Register X" would be prime candidates. However, there are a few more subtle choices. For example, consider the same array scaling code fragment, but with the result store in another array:

```
//Scale all elements by 10

For I = 0 to 9
    Array_B[I] = Array_A[I] * 10
```

Assuming that *FR1* points to *Array_A* and *FR2* points to *Array_B*, the SAFA code is as follows:

```
1.start:
2.    Load_Element FR1
3.    Load 10
4.    Multiply
5.    Store_Element FR2
6.    Increment_Index FR1     //advance to the next element
7.    Cmp_Index_Limit FR1     //compare the limit and index
8.    if_not_equal start      //jump back to start
```

The main question to ask is should we make the "Load_Element" and "Store_Element" instructions context trigger? Guided by the spatial locality principle, loading an element from a frame is usually a good sign that more work would be done on the same frame. How about *storing* an element? Now suppose the answer is positive, the resultant code is a little awkward:

```
//Assume:
//      FR1 = array reference for Array_A
//      FR2 = array reference for Array_B

1. start:
2.    Load_Element FR1
3.    Load 10
4.    Multiply
5.    Store_Element FR2     //context changed to FR2
6.    Set_Frame FR1         //reset the context to FR1
7.    Increment_Index       //advance to the next element
8.    Cmp_Index_Limit       //compare the limit and index
9.    if_not_equal start    //jump back to start
```

The additional instruction at *line 6* must be added to "reset" the context correctly. Although this example in itself, does not warrant of reverting the earlier decision, common programming practice tell us that a result is only written after it is no longer needed. Hence, in SAFA, storing an element into a frame does not trigger context change, in other words, these instructions are context insensitive. So, the same code should be modified as:

```
//Assume:
//      FR1 = array reference for Array_A
//      FR2 = array reference for Array_B

1. start:
2.     Load_Element FR1
3.     Load 10
4.     Multiply
5.     Store_Element FR2      //no context change
6.     Increment_Index        //advance to the next element
7.     Cmp_Index_Limit        //compare the limit and index
8.     if_not_equal start     //jump back to start
```

With context sensitivity, the instruction space for the frame instructions can be cut down significantly. However, there are always programs that defy the common principles. When the context changes rapidly, explicit instruction to reset the context must be inserted. This is the main draw back of context sensitive instructions.

3.4.2 Prefetching

It is a well known fact that there is a huge gap between memory speed and CPU speed to the tune of several order of magnitudes. The speed of CPU become meaningless if the required instructions or data can not be brought into the CPU in time. This the main reason that drives many of the designs in computer architecture, like caching, burst memory access etc.

Prefetching is one of those attempts. There are many situations where prefetching is useful:

- Start fetching the instruction from a branch target.
- Read the next few records/array elements in an array.
- Read the next record in a memory access (linked list).

- etc

Prefetching is usually implemented in two forms:

1. Statically, compiler/programmer insert memory fetching code prior to memory operations. This requires specialized instructions.

2. Dynamically, cpu looks ahead and prefetches instruction/data.

The first form can gives good result but requires additional effort from the programmer or compiler writer. The second form already exists in most CPU in the form of prefetching instructions from branch target. However, dynamic prefetching requires additional logic for the CPU and thus only sees limited application. A CPU must be able to determine a memory access in advance to initiate the prefetching.

In SAFA, prefetching follows directly from the inclusion of frame registers. Unlike a GPR machine, where a register may contain either data value or data address, the frame register in SAFA *must* be a reference to a piece of memory space. With this distinction, frame register operations represents opportunities for prefetching in SAFA.

For example, consider the code for traversing a linked list in the previous section:

```
Load_Halfword FR1       //load halfword indexed by FR1
....                    //Accumulate the total
Load_Word_Offset FR1,28 //FR1's base + 28 =  base of next record
Set_Base FR1         //Prefetching opportunity
```

As soon as the base of a frame register is changed, the memory prefetching can begin from the new address. Since most frame register instructions set the *current frame pointer* automatically, this can be used as a signal to trigger the prefetching.

Although the simplistic dynamic prefetching in SAFA may not yield the best result compared to other prefetching schemes, it has the benefit of being ac-

commodated in the implementation, since it resulted directly from the central idea in SAFA, the frame registers.

3.5 Summary

With the inclusion of *frame register*, SAFA demonstrated qualitatively good support for HLPs features. In particular, the two major components of most HLPs: Control flow and data structures, received extra attention. The proposed frame registers and its associated operations can be added to any stack-oriented architecture to reap the same benefits as discussed. In Chapter 6, we have provided quantitative results of using frame register in HLPs.

Chapter 4

Low Level Execution Support

The biggest hurdle for improving *ILP*[1] in stack oriented architecture is the stack itself. Inherent dependency on the stack-top access stops many instructions from parallel execution, since operands required for an operation must reside on top of the stack. Whenever this condition fails, the execution has to be delayed. For example, a simple integer *add* instruction cannot be executed, if one of the two required operands is not ready (for example, cache miss). The whole execution pipeline has to wait for the memory operation that brings that missing operand onto the top of the stack, before new operands can be pushed, or lower elements can be popped.

It is natural to conclude that superscalar execution is not applicable for stack architecture, since each instruction assumes exclusive control of the top of the stack for the respective operands. So, even there are multiple execution units, it is still not possible to execute more than 1 instruction simultaneously. In short, stack machine exhibits a high level of data dependency. Moreover, unlike the register based machines, where the data dependency can be identified and resolved using tagged execution (register renaming) as seen in Section 2.4, there is no clear cut way of performing the same renaming for stack machine because the lack of source-destination (producer-consumer) information in the instructions. The succinctness

[1]Instruction Level Parallelism

of the 0-address stack instruction, which is essential for compact binary size, becomes its biggest enemy.

Although the odds against the stack architecture seem strong, there are quite a number of researches that indicate otherwise. For example, [36] shows that by mapping stack locations to registers thereby relaxing the data dependence, an ideal execution (perfect pipelining) can yield an average BLP[2] of 5.6 in Java Benchmark Suite (*SPECjvm98*). Additionally, under speculative execution, the average BLP reaches a high 19.8 (assuming perfect branch prediction). Besides, the popularity of the Java programming language has prompted heated research into augmenting the Java Virtual Machine and resulting in actual Java Processors. Some of these researches are summarized in Section 4.5 for comparison after our ideas are presented.

We believe that a simple mechanism devised for the register based machine can solve this problem quite satisfactorily. The Tomasulo's Scheme[30][38], can be adapted to transform stack operations into tagged execution as indicated in the execution framework in Section 1.1. By establishing linkage (relationship) between these tagged stack instructions, conventional superscalar techniques can be adapted to support and improve *ILP* for stack machines. The design and development of the SAFA scheme is largely guided by the General Tagged Execution Framework described in Section 1.1. To avoid frequent interruptions in the flow of the main explanation, a separate section (Section 4.6) is provided at the end of this chapter to describe the derivation process.

To facilitate discussion, a brief overview of instruction dependencies, which covers the various relationship between instructions will be provided in the next section. After which, the superscalar mechanisms deployed along with the necessary modifications will be presented.

[2]Bytecode Level Parallelism, i.e. ILP at Bytecode level

4.1 Overview of Instruction Dependencies

Instruction dependencies are the key stone in study of instruction level parallelism. A careful study of instruction relationship can determine not only the amount of parallelism, but also the exact way to extract and exploit this parallelism. According to [30], this study can tell us:

- If two instructions are parallel, they can be executed without causing any stall, provided there is enough resources.
- Instructions that are dependent are not parallel, hence cannot be reordered.

The three types of instruction dependences are:

1. Data Dependence: See Section 4.1.1
2. Name Dependence: See Section 4.1.2
3. Control Dependence: See Section 4.1.3

For most of these dependences, a simple program can go a long way in illustrating the problem. Hence, to better indicate the machine operations, a RISC-like register based format will be used:

```
op reg1,reg2,reg3
```

which means "apply *op* to *reg1* and *reg2*, then place the result in *reg3*".

4.1.1 Data Dependence

Data dependence, or more commonly known as the producer-consumer dependence, occurs whenever an instruction i produces a result that is used by instruction j. Instruction j is said to be *data dependent* on instruction i. This dependence is also

transitive. If instruction j is data dependent on instruction k, which in turn is data dependent on instruction i, then instruction j is also data dependent on i.

A simple example would be:

```
ADD R1, R2, R3      ; Instruction i
ADD R3, R3, R4      ; Instruction j
```

As can be seen, instruction i produces the result of addition in register **R3**, which is used by instruction j. Obviously, instruction i and j cannot be executed at the same time because of the dependence. An implicit execution order is imposed on the two instructions, where instruction i must be performed before j.

If we examine this problem from the view point of resource utilization (read or write a resource, e.g. register or memory location), data dependence will create a Read after Write (**RAW**) hazard[3]. In the example above, if instruction j read the content of **R3** before i manages to put the result in, then j will incorrectly read the old value.

It is important to know that data dependence is a *natural property* of computation (a value is repeatedly transform into a result we need). Hence, data dependence, or at least, the *effect* of the dependence must be preserved.

4.1.2 Name Dependence

Name dependence occurs when two instructions use the same register or memory location (i.e. resource with same *name*). However, there is no real data flow between the two instructions as opposed to data dependence. In another words, this dependence stems from the utilization conflict of resource, which partially is partially caused by scarcity of a particular resource. For example, name dependence may be created when limited number of registers forced the compiler to reuse the same register for unrelated instruction.

[3]Problem that prevent next instruction from executing in parallel[30]

Between an instruction i and j, there are two possible types of name dependences:

Anti-Dependence

When instruction j writes to a destination, which is read by an earlier instruction i, anti-dependence is created. The name *anti-dependence* comes from the fact that it is the opposite from data dependence.

The following code illustrates anti-dependence:

```
ADD R1, R1, R2      ; Instruction i
ADD R3, R3, R1      ; Instruction j
```

This also corresponds to the Write after Read (**WAR**) hazard. If incorrectly executed, instruction i may read the new value produced by instruction j.

Output Dependence

Output dependence is caused by instructions i and j writing to the same register or memory location. As shown in the example below:

```
MUL R1, R2, R3      ; Instruction i
ADD R4, R5, R3      ; Instruction j
```

This is also the cause of Write after Write (**WAW**) hazard. Incorrect execution may caused the wrong result to be stored. In the example shown, if instruction j finishes first, the result in **R3** will be overwritten by instruction i.

Since there is no real data dependence in both *antidependnce* and *output dependence*, the instructions involved can be executed in parallel or re-ordered, *provided* the name of the resource can be changed. This renaming can be performed easily enough for registers, which is called *register renaming*, which is used in virtually all architectures nowadays.

4.1.3 Control Dependence

As opposed to the previous two types of dependence,which deal mainly with data values and/or resources, the last type of dependence, *Control Dependence* study dependences created by program flow (control flow). In brief, the ordering of an instruction is study with respect to a branch instruction to ensure that execution only occurs for instructions in the correct control path.

The basic rules for control dependence are:

1. An instruction i that is control dependent on a branch *cannot* be moved before the branch. This movement breaks the dependence and allow instruction i to be executed regardless of the outcome of the branch instruction.

2. An instruction i that is not control dependent on a branch *cannot* be moved after the branch. Clearly, this rule is the reverse of the previous one.

Examine the example below (which is written in a C-like syntax):

```
s1;
if (condition){
    s2;
}
```

Moving the statement *s1* into the *if* block violate the first rule, whereas moving the second statement *s2* before or after the *if* block violates the second. The rules help to preserve the correctness of the execution by imposing a correct ordering of instructions.

Since most programs are non-linear, which involves multiple control paths, most instructions are under the influence of one branch instruction or the other. If control dependence can be weakened, more instructions will be available for execution. In particular, program loops represents the biggest potential source of speedup.

4.2 Coping with Data and Name Dependence

In this section, we concentrate on instructions other than control flow instructions (any instruction that causes the change of execution path). Since control dependence is out of the picture, only data and name dependences need to be handled. As explained in the previous section, true data dependence must be preserved, whereas name dependence can be removed. So, any mechanism that is designed to exploit ILP must be able to:

1. Recognize the existence of a dependency.
2. Preserve the ordering of execution in the case of true data dependence.
3. Allow for parallel execution by removing name dependence.

One such mechanism is the famous *Tomasulo's Scheme*, which is covered next. The adaptation for **SAFA** is presented right after that.

4.2.1 Tomasulo's Scheme

The original Tomasulo's scheme was devised in 1967 for facilitating execution of floating point operations in the computer *IBM System/360 Model 91*. In the famous paper[38], R.M.Tomasulo stated that the main idea of his mechanism is

> ..., the objective must be to preserve essential precedences while allowing the greatest possible overlap of independent operations.

The scheme proves to be flexible and effective enough to be extended to other instructions type and incorporated into other architectures.

The two main concepts in Tomasulo's Scheme are:

1. Incorporating the *reservation station*, which is an extension of the operand buffering.

2. The mechanism **CDB** (Common Data Bus) and a simple tagging scheme to preserve precedence while encouraging parallelism.

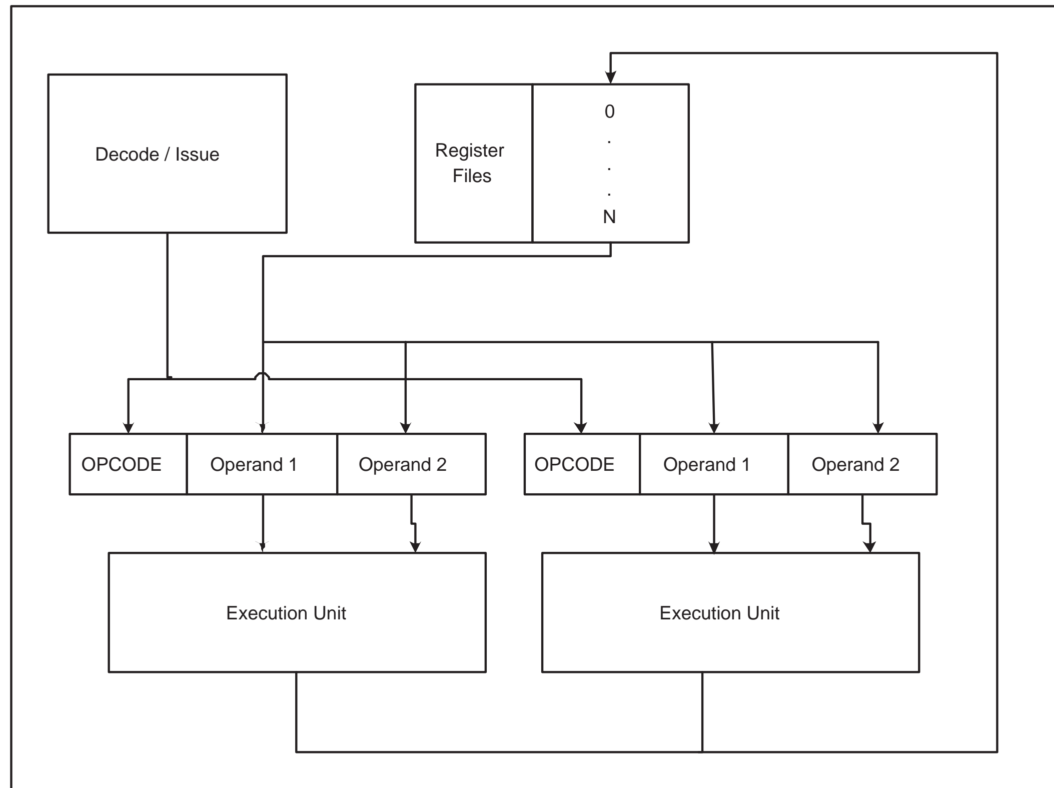


Figure 4.1: Simple Architecture without Tomasulo's Scheme

Figure 4.1 illustrates a simple architecture which does not deploy the Tomasulo's Scheme. In this architecture, an instruction goes through the following step for execution (starting from the decode/issue stage) :

1. Read the operand(s) from register.

2. The operation along with the operand(s) is passed to a execution unit if an appropriate unit is available. The operand(s) is then stored in the temporary storage in the execution unit.

3. Proceed with execution. If a result is produced at the end of the execution, it is passed back to the register file to be written into destination register.

The steps above seems simple enough, however, it itself cannot handle any of the RAW, WAW, or WAR hazards mentioned in Section 4.1. The situation may be improved if a busy bit is included with each register to indicate the availability. In the issue stage, an instruction would first check the availability of its source register(s) by checking the busy bit. If the register(s) is ready, then the destination register would be marked as busy before the instruction and its operands are sent to the execution unit. If any of the required registers is already marked as busy, then the instruction has to be stalled. When a result is produced, it is written back to the register file and the associated busy bit will be reset.

Although this scheme avoided the hazards, it is accomplished by imposing strict execution order. Any dependence between a pair of instructions will result in sequential execution (in-order execution). Also, execution may be stalled simply because the relevant execution unit is busy.

The Tomasulo's scheme, on the other hand, not only solve the problem but also allow out-of-order execution. Figure 4.2 shows the same architecture with Tomasulo's Scheme deployed.

Ignoring the tags and CDB for the moment, it can be seen that the *reservation stations* sitting on top on each execution unit are simply multiple set of the temporary storage from Figure 4.1. This straightforward addition relaxes the constraint on the availability of execution unit. Instructions now can be issued as long as *reservation stations* of the appropriate kind has vacant entry. Since there are more reservation station entries than execution units, more instructions can be issued.

The effect of the reservation stations is dampened by the fact that no instruction can be issued whenever a dependent instruction is decoded. The dependence must be resolved or relaxed if we expect better performance.

First, lets consider the problem of data dependence. As discussed, data dependence *cannot* be removed, for program correctness sake. The potential RAW

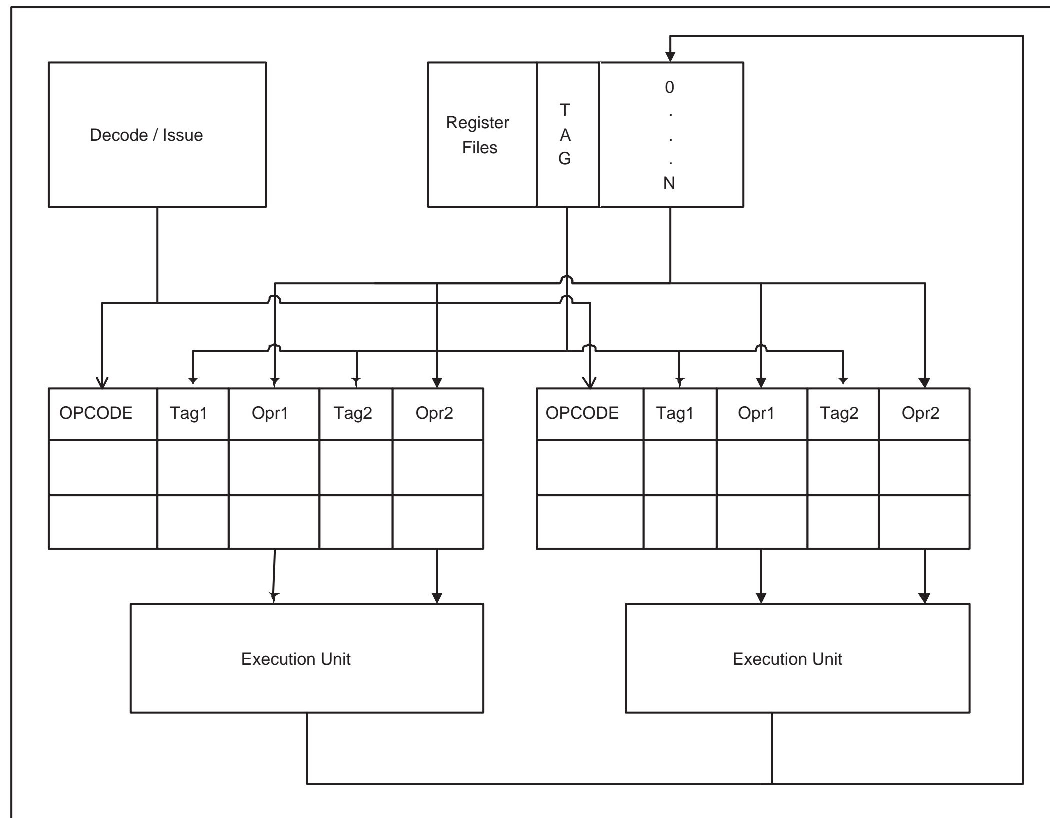


Figure 4.2: Simple Architecture with Tomasulo's Scheme

hazard is avoided by the busy bit scheme described. However, the delay can be further mitigated if the instruction that waits for operand(s) can be somehow issued without impeding the progress of later instructions. One simple solution is to allow the instruction to progress to the reservation stations and wait there. To allow for this solution, there are two additional problems:

1. The instruction must be able to identify the operand(s) even if it is waiting in the reservation station.

2. When the required operand(s) is ready, it must be brought into the corresponding ”slots” in the reservation stations.

The latter can be easily solved by adding a feedback connection from the output of execution unit back into the reservation station. This connection is given the name Common Data Bus (CDB) in Tomasulo's Scheme. The solution for the former will

become clear after the following discussion.

Second, lets concentrate on the two type of name dependences to see how can they be resolved. Both type of the name dependences can be resolved *if* we can keep track of the different *versions* of the value, which is created by each successive writing. In particular:

- **Antidependence**, which creates the WAR hazard. If the instruction write into a *new* version of the register, then preceding instruction will not get the wrong value, since it is reading from an *old* version.
- **Output Dependence**, which creates the WAW hazard. If an instruction produced a later version of a register content, then previous version(s) can be discarded without being written.

As discussed in Section 4.1, this versioning is nothing but *register renaming*. But giving a new *name* to a register, we are, in effect creating a new *version* of a register. Also, this new name can be used as identification for operands. In Tomasulo's Scheme, this renaming is achieve via *tagging*, which is to associate an alternative id to each register. Each tag can be a simple numerical value. With all the new addition, the execution steps are now:

1. The instruction is associated with a new tag, T to label its destination register (if any).
2. Check the availability of the operand(s), including source and destination.
   (a) If none of them are busy, the instruction can be issued to a free reservation station of the appropriate type, along with the required register(s) content.
   (b) If source register(s) is busy, the instruction will still be issued. However, instead of actual register content, the tag(s) associated with the register(s) is brought over.

3. Busy bit for the destination register is set.

4. The tag of the destination register is set to T.

5. Execution of an instruction can begin as soon as the execution unit is free *and* all the required operands arrived.

6. The result, along with the tag T is broadcast on the CDB. Any instruction that requires the it will be able to identify it using T and picks it up.

7. The register file receive the result and checks for T in the register file. If it can be found, the corresponding register content is overwritten with the result and the busy bit unset. The result is simply ignored (discarded) if the tag cannot be found.

It is simple to see why the Tomasulo's Scheme remain a powerful mechanism up to this day. The scheme skillfully solves the nagging problems caused by dependencies while adding minimum complexity to the architecture. In particular:

- Step 2a allows data dependent instruction to be issued without all the operands.
- Step 4 solves the WAR hazard by versioning.
- Step 4 and step 7 in conjunction, solves the WAW hazard.

Adapting Tomasulo's Scheme for Stack-Oriented execution is presented next.

4.2.2 Adaptation for SAFA

By adding additional information to the register file (with Tomasulo's Scheme deployed) discussed in the previous section, we have the centerpiece of the SAFA execution engine: the *Reorder Buffer* (RoB). Note that RoB may have different

usage in other architectures, such as PowerPC (used to reorder completed instructions according to program order [23]). In SAFA, it is essentially a modified logical registers file.

The reorder buffer is assumed to have a number of entries, each with a unique identification number (*tag* hereafter). Each entry comprises several fields:

Field	**Description**
Operator	The instruction to be executed.
Idle/Busy Flag	Indicates availability of this entry to store a new instruction. Abbreviated as **I/B**.
Free/Committed Flag	Indicates whether the result of this entry is committed to a consumer operator or not. Abbreviated as **F/C**.
Waiting/Available Flag	Indicates whether the result of the instruction is available or not. Abbreviated as **W/A**.
Destination Tag	The tag number of the consumer operator, if known.
Left/Right Operand Bit	Indicates the availability of the left and right operands of the instruction.
Value	The result of the instruction after execution. This field also doubles up as temporary storage for the left or right operand value before execution.

To maintain data dependencies between consumer/producer reorder buffer entries, a stack of reorder buffer tags(**O**perand **T**ag **S**tack, *OTS*) is used.

All instruction, upon entering execution, are assigned a reorder buffer entry. Suppose this instruction needs results from other preceding instructions (i.e. a consumer), tags are removed from the top of *OTS*. Likewise, if this instruction produces result for consumption, its tag is entered into the top of *OTS*. If an entry has all its operands ready, then it will be scheduled for execution. When the execution result is produced, it will be kept in the entry if it is not taken/consumed by the destination.

As an example, suppose the expression $(A-B)\times(C+D)$ is to be executed. To have a clearer picture of the relationship between the operators in the expression, it is converted to the polish notation (a.k.a. postfix notation): $AB - CD + \times$. Before execution, the 8-entry[4] reorder buffer as well as the operand tag stack[5] has the following stack:

tag	flags	Dest.Tag	Operator	Left	Right	Value
7	I F W	–	–	0	0	–
⋮	⋮	⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	I F W	–	–	0	0	–
0	I F W	–	–	0	0	–

Operand Tag Stack	[→]

After the instructions to load A and B have been executed, the state changes to the following. Take note of the changes for the various fields and operand tag stack.

tag	flags	Dest.Tag	Operator	Left	Right	Value
7	I F W	–	–	0	0	–
⋮	⋮	⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮	⋮	⋮
2	I F W	–	–	0	0	–
1	B F W	–	Load	1	1	@B
0	B F W	–	Load	1	1	@A

Operand Tag Stack	[0, →1]

Since both of the load instructions take only 1 operand (the address), which

[4]The number of entries is chosen only as example

[5]→ is used to indicate stack top

is also present with the instruction, they are ready for execution. In the mean time the subtraction operator comes in, and consumes the two tags, as follows.

tag	flags	Dest.Tag	Operator	Left	Right	Value
7	I F W	–	–	0	0	–
⋮	⋮	⋮	⋮	⋮	⋮	⋮
3	I F W	–	–	0	0	–
2	B F W	–	Subtract	0	0	–
1	B C W	2	Load	1	1	@B
0	B C W	2	Load	1	1	@A

Operand Tag Stack	[→2]

Suppose the two loads result in cache hits and return the value right away, then the subtraction operator will be able to proceed. After the load operations return, their entries are deallocated. When the result of the subtraction returns, then we will have the following state:

tag	flags	Dest.Tag	Operator	Left	Right	Value
7	I F W	–	–	0	0	–
⋮	⋮	⋮	⋮	⋮	⋮	⋮
3	I F W	–	–	0	0	–
2	B F A	–	Subtract	1	1	$(A - B)$
1	I F W	–	–	0	0	–
0	I F W	–	–	0	0	–

Operand Tag Stack	[→2]

The subsequent operations i.e. the loading and addition of C and D would be processed in similar fashion, producing the state:

tag	flags	Dest.Tag	Operator	Left	Right	Value
7	I F W	–	–	0	0	–
6	I F W	–	–	0	0	–
5	B F A	–	Add	1	1	$(C + D)$
4	I F W	–	–	0	0	–
3	I F W	–	–	0	0	–
2	B F A	–	Subtract	1	1	$(A - B)$
1	I F W	–	–	0	0	–
0	I F W	–	–	0	0	–

Operand Tag Stack	[2, →5]

When the final multiplication operator is decoded, it can get hold of the two operands quickly and proceed to execution. At the point in time, we have:

tag	flags	Dest.Tag	Operator	Left	Right	Value
7	I F W	–	–	0	0	–
6	B F W	–	Multiply	1	1	–
5	I F W	–	–	0	0	–
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	I F W	–	–	0	0	–
0	I F W	–	–	0	0	–

Operand Tag Stack	[6]

When the result of the multiplication returns, it will be kept in entry no.6 and be available for consumption.

Although a sequential scenario is presented in the execution above, it is obvious that this scheme can cater for irregularities too. For example, suppose the fist two loads (for A and B) results in cache misses, the subtraction that follows will have to wait, however, the loads for C and D can continue without problem. As an

illustration, let us suppose that the loading of C and D is successful which results in the following state:

tag	flags	Dest.Tag	Operator	Left	Right	Value
7	I F W	–	–	0	0	–
6	I F W	–	–	0	0	–
5	I F W	–	–	0	0	–
4	B F A	–	Load	1	1	(D)
3	B F A	–	Load	1	1	(C)
2	B F W	–	Subtract	0	0	–
1	B C W	2	Load	1	1	@B
0	B C W	2	Load	1	1	@A

Operand Tag Stack	[2, 3, →4]

At this point, when the addition operator comes in, it can read the operands and proceed for execution. Meanwhile, let's suppose that the value of A is finally loaded from memory. We will have:

tag	flags	Dest.Tag	Operator	Left	Right	Value
7	I F W	–	–	0	0	–
6	I F W	–	–	0	0	–
5	B F W	–	Add	1	1	–
4	I F W	–	–	0	0	–
3	I F W	–	–	0	0	–
2	B F W	–	Subtract	1	0	(A)
1	B C W	2	Load	1	1	@B
0	I F W	–	–	0	0	–

Operand Tag Stack	[2, →5]

As can be seen in the example above, irregularity in the execution pattern (e.g. cache misses) does not halt the machine, since other instructions can still

proceed without waiting. Also, it is clear that the above scheme allows simultaneous execution of more than 1 instruction, if such instructions are available. Hence, this scheme effectively transforms the reorder buffer into "multiple logical stacks", where each operator keeps track of its own operands in a similar fashion to a data-flow machine.

Under close scrutiny of the scheme detailed above, we find that there is still room for improvement. The reorder buffer serves a purpose very close to that of a register file in a *GPR* machine. As with its counterpart in *GPR* machine, it is a scarce resource that needs to be utilized efficiently. Two improvements are proposed to improve the utilization of reorder buffer:

1. Reservation Station and Virtual Tag

2. Preemptive Drizzling

Reservation Station and Virtual Tag

As discussed in the Section 4.2.1, reservation is a simple yet effective enhancement to lessen the pressure on the reorder buffer. The idea of the reservation station is to dispatch the instruction as soon as it manages to get hold of a reorder buffer entry. The reservation stations, essentially temporary storage for instructions sits on top of each execution unit, and will take care of the maintenance of the instructions assigned. Each entry in the reservation station contains the following information:

Field	**Description**
Own Tag	The corresponding tag in the reorder buffer.
Operator	The instruction to be executed.
Left/Right Operand Field	The field consists of: **Type** Whether it is a **V**alue or **T**ag. **Value** Store the actual value of operand or the tag of the operator that will produce the value.

Under this new scheme, when an instruction enters the reorder buffer, it will pick up its operands, either actual values, or tags of operators producing the values, and is dispatched to a corresponding execution unit. The reservation station will wait until an instruction acquires all its operands, before sending it into execution. When the execution result is ready, it is broadcast to all other reservation station (the inter-connection usually named as *Common Data Bus (CDB)*), along with the owner tag. So, any station entry waiting for this result to proceed will be able to pick it up.

With the help of reservation station, the demand on actual reorder buffer entry is lessened. However, since the tag number corresponds to physical location of reorder buffer entry (i.e. Tag number 0 is the first entry, Tag number 1 the second etc), the associated entry remains in use until the execution completes. If the tag number and physical ROB entry are reused as soon as a consumer instruction has picked up the tags before the result is ready, ambiguity may occur (two results may be sharing the same tag number) and cause incorrect execution. A closer look should reveal that this is nothing but the *name dependency* discussed in Section 4.1.2, where the Tag number behaves just like a register number. This false dependency may cause the execution to stall/wait even when there are technically free ROB entries.

As with all *name dependency* problems, this can be easily solved by attaching *virtual* tag numbers instead of physical ROB entry numbers to results, i.e. tag renaming. A pool of virtual tag numbers, usually twice the number of physical ROB entry is set up. Upon entering the *Issuing Stage*, a producer instruction grabs a new virtual tag and attach it to a free ROB entry, thereby creating a virtual-to-physical mapping. This virtual tag behaves exactly the same as the tag number discussed so far in other aspects. It is stored in the *Tag* field of a ROB entry as well as the OTS, ready to be picked up by consumer instruction. Whenever a virtual tag is picked up by a consumer, the associated ROB entry can be freed immediately regardless of the availability of the result, allowing later instructions to utilize it.

A particular virtual tag number will remain in used *until* the the result is produced. As discussed, the result will be broadcasted along with the virtual tag number. Depending on the execution pattern, one of the following scenarios must be true:

- The consumer for that result has already been dispatched, i.e. the corresponding ROB entry has been freed. The consumer instruction, sitting on reservation station will pick up the result. Hence, the virtual tag number is safe to be returned to the pool to be reused.
- The consumer instruction has not been dispatched. The result will be stored in the corresponding ROB entry. The virtual tag number can only be returned *after* the consumer instruction picks up the result.

The two extensions discussed affects the information stored in RoB entry as some of the information are now implied rather than explicit. The altered RoB entry now contains:

Field	**Description**
Virtual Tag	Stores associated virtual tag number.
Operator	Indicates the producer of this entry. Note: this is no longer needed in actual RoB entry, as all instruction are issued right away, this is kept for illustrative purposes.
Use Count	Indicates the total number of usages for this entry.
Idle/Busy Flag	Indicates availability of this entry to store a new instruction.
Waiting/Available Flag	Indicates whether the result of the instruction is available or not.
Value	Stores the produced result *if* the entry is not consumed.

The modified fields are pretty self explanatory, except the *Use Count* field. Since a value in RoB may be duplicated via stack manipulation instructions, it is not efficient to copy a exact replica to take up another RoB entry. The *Use Count* field is used instead to simulate this effect instead. When an value is duplicated, the *Use Count* of the corresponding RoB entry is simply incremented. At the same time, the same virtual tag number is pushed into the OTS. This ensures that the number of copies of a particular virtual tag in OTS matches the *Use Count* field. Conversely, when a virtual tag is popped, the *Use Count* would be decremented. When the *Use Count* reaches zero, a RoB entry can be safely removed.

As an example to show the interaction between RoB and reservation station, consider the expression $A \times A - B$, with the corresponding pseudo code:

```
load A          //I1
duplicate       //I2
multiply        //I3
load B          //I4
subtract        //I5
```

Assume that *I1* is executed successfully, then we have the following RoB:

RoB entry	VTag	Instruction	Use Count	Value
7	–	–	0	–
⋮	–	–	0	–
2	–	–	0	–
1	–	–	0	–
0	0x03	load	1	3.14

Operand Tag Stack	[→0x03]

Note that both virtual tag number *0x03* and the value *3.14* are randomly chosen. The next instruction *I2* will duplicates the value, resulting in the increment of *Use Count* and the extra copy of the virtual tag *0x03* in OTS.

RoB entry	VTag	Instruction	Use Count	Value
7	–	–	0	–
⋮	–	–	0	–
2	–	–	0	–
1	–	–	0	–
0	0x03	duplicate	2	3.14

Operand Tag Stack	[0x03, →0x03]

Instruction *I3* picks up both the operands and proceed to the reservation station of integer unit. This reduces the *Use Count* of *0x03* to zero and results in the removal of RoB entry 0.

RoB entry	VTag	Instruction	Use Count	Value
7	–	–	0	–
⋮	–	–	0	–
2	–	–	0	–
1	0x08	multiply	1	Waiting
0	–	–	0	–

Operand Tag Stack	[→0x08]

Own Tag	Instruction	Left Operand		Right Operand	
		Type	Value	Type	Value
0x08	multiply	V	3.14	V	3.14

In the value fields above, *V* denotes actual value. As the *multiply* instruction has all its operands ready, it can go into the execution on next time cycle.

Instruction *I4* will be executed next. Suppose the loading of *B* encounters a cache miss, then we have the following state:

RoB entry	VTag	Instruction	Use Count	Value
7	–	–	0	–
⋮	–	–	0	–
2	0x01	load	1	Waiting
1	0x08	multiply	1	Waiting
0	–	–	0	–

Operand Tag Stack	[0x08, →0x01]

Own Tag	Instruction	Left Operand		Right Operand	
		Type	Value	Type	Value
–	–	–	–	–	–

Note that the reservation is now empty as the *multiply* proceeds to execution. Suppose that the multiply result 9.85 is ready on the next time tick. Then we have the following state after we issue instruction *I5*:

RoB entry	VTag	Instruction	Use Count	Value
7	–	–	0	–
⋮	–	–	0	–
2	–	–	–	–
1	–	–	–	–
0	0x05	subtraction	1	Waiting

Operand Tag Stack	[→0x05]

Own Tag	Instruction	Left Operand		Right Operand	
		Type	Value	Type	Value
0x05	subtraction	V	9.85	T	0x01

The *T* in the type field represents a unavailable operand. As the operands of the *subtraction* instruction are not complete, it will sit in the reservation station waiting. When the memory unit (not shown) loaded the value of *B*, it will be broadcasted along the CDB with the tag *0x01*. The reservation station of the integer execution unit will then pick it up and proceed with the subtraction.

The example above shows that the combined effect of reservation station, virtual tag number and *Use Count* to a lesser degree can keep the utilization of actual RoB entry low.

Preemptive Drizzling

Stack overflowing is a common problem for stack machine. For SAFA, the reorder buffer faces the same problem. Even with all the improvements discussed, reorder buffer overflowing would still occur. Since expression may be nested to arbitrary length, it is always possible to find a computation that causes overflowing. Consider a ROB with 8 entries, this postfix expression "$ABCDEFGHI++++++++$" occupies all the ROB entries by loading the values of A to H. At this point, the value of I cannot be loaded because of lack of actual ROB entry. However, none

of the entries can be freed because the consumer (the addition operator) cannot be issued before I is loaded.

Stack overflowing is usually solved by moving the bottommost entry, which is less likely to be used immediately, to the memory. In our example, the ROB entry containing A will be moved to memory, making space for value of I. When the value of A is needed, after seven additions, the value of A is brought back from the memory. The restoration can also happen when the stack is empty.

Although this scheme is elegant and simple, there are still rooms for optimization. The timing of the memory operations can be further tightened to give better performance. Observe that the values in the memory are brought onto the stack only when the stack is empty or the value is needed. Also, the values are only moved to memory when the stack is totally full. Since all these value movements employ lengthy memory operations, it is best to anticipate their occurrence to start the operation as early as possible.

Following the footstep of picoJava I and II [10][11], we employ preemptive drizzling in SAFA. The scheme is basically a modification of the timing of the value movement. Instead of waiting the stack to be full, value can start moving to the main memory when the stack is 80% full for example. Similarly, values can be restored as soon as the utilization of stack fall below a threshold, e.g. 20%. These thresholds are appropriately named as “high water mark” and “low water mark” respectively in [13].

The effect of these frequent small memory movements (hence the name *drizzling*) may not be overwhelming for small reorder buffer (because the margin between the threshold and absolute is too small), the benefit is obvious when the reorder buffer gets larger. The latency and the cost of the memory operations is spread across several time cycles, therefore lessening the effect on execution.

4.3 Coping with Control Dependence

In this section, we will looking at the issue of introducing branch prediction and speculative execution in SAFA. Control dependence in the context of stack machine is not an overly well-studied field. Since the traditional stack machines lack the ability for multiple issuing and more importantly out-of-order execution, coping with control dependence seems to be the least of their inadequacies. As demonstrated by the last section, both multi-issue and out-of-order execution are now in the grasp of stack-oriented machine, it is now essential to look into the realm of speculative execution.

4.3.1 Branch Prediction and Speculative Execution in General

As mentioned in Section 4.1, most instructions in a program are under influence of a control decision. Usually, there are several mutually exclusive execution paths in a program, which are taken according to the result of a conditional branch instruction. For example, given the program below:

There can be four execution paths, depending on the combination of the conditions.

cond1	cond2	cond3	Instructions Executed
True	True	–	s1,s2,s3
True	False	–	s1,s2,s4
False	–	True	s1,s5,s6
False	–	False	s1,s5,s7

It is clear that only “free” instruction (*s1*) can be executed without depending on resolution of a condition, while all other instructions (*s2* to *s7*) have to wait on one or more conditions. This dependency also means that multi-issuing and out-of-order execution can not improve the situation in any way.

```
s1;
if (cond1){
    s2;
    if (cond2){
        s3;
    } else {
        s4;
    }
} else {
    s5;
    if (cond3){
        s6;
    } else {
        s7;
    }
}
```

Figure 4.3: Control Dependence Example 1: *if-else*

Other then *if-else* statement, control dependency also slows down execution of iterative loops. For example:

```
while (cond){
    s1;
    s2;
}
```

Figure 4.4: Control Dependence Example 2: *while* loop

If the conditions is true for a number of iterations, we would expect a 4-issues execution engine to executes close[6] to two iterations (two copies each of *s1* and *s2*) per cycle. However, since *s1* and *s2* are dependent on the *cond*, their executions cannot be started legally until *cond* is resolved. The resultant throughput therefore, is disappointing.

As shown by the two examples, instructions with control dependencies can stall the execution. The fact is that branches and loops are essential mechanics in

[6]the condition resolution also take up execution slot

most, if not all programs. In [43], it is found that there are about 10 to 30 percent branch instructions in a typical program. Additionally, the branch is taken 60 to 70 percent of the time. Confronted with this problem, computer scientists came up with quite a number of solutions, of which we will concentrate on one strategy: "guessing"[43][44].

By guessing (predicting) the execution path, the execution can continue without waiting on the resolution of the condition. For example, in Figure 4.3, we can assume that path one (where both *cond1* and *cond2* are true) is more likely to be taken, and executes *s2* and *s3* before *cond1* and *cond2* are resolved. Similar prediction can be applied for loops, for example (refer to Figure 4.4), we can assume that the next iteration is going to be executed and send both *s1* and *s2* into the execution again.

These predictions will not cause any trouble *as long as* they turned out to be correct. When a resolved condition shows that the wrong path has been taken, backtracking (or unrolling) is needed to undo all mispredicted executions. This is clearly the hardest aspect of the speculative execution. For instance, in *GPR* machines, instructions usually store result into register. If a speculative instruction overwrites the content of a register, a copy of the previous result must be kept in case the prediction is wrong. There are a number of well tested solution, for e.g. having a duplicate set of registers (the shadow registers) in PowerPC 620 (refer to Section 2.4.2). However, multi-level predictions (during the execution of nested *if-else* for example) is still not a simple task.

In a nutshell, there are three major requirements for a computer architecture to enable branch prediction and speculative execution:

1. Ability to indicate that an instruction is speculative.

2. Ability to confirm the instruction if the prediction is correct.

3. Ability to undo the instruction if the prediction is incorrect.

In addition, able to handle multiple level of prediction would be an advantage too. The solution we proposed for SAFA is presented in the next section.

4.3.2 Branch Prediction and Speculative Execution in SAFA

In some aspects, speculative execution for stack machine is somewhat easier than in *GPR* machine. Recall that one of the main hurdles of speculative execution is the restoration of previous register content should the prediction fails. This implies versioning of register content. However, in a stack machine, versioning is an inherent characteristic: each instruction is supposed to produce result to an entirely new location (the stack top). Should prediction fails, this new result can be simply removed from the stack.

Nevertheless, restoration after misprediction is still needed for values that are consumed by speculative instructions. This task can be made simpler by delaying the removal of a value until the relevant condition is resolved.

Before we delve into the details, lets define an important term that will see frequent use later on: *prediction level*. Prediction level is simply a number indicating the current nested level of speculation. For example:

```
s1;
if (cond1){
    s2;
    s3;
    if (cond2){
        s4;
        s5;
    }
}
```

Figure 4.5: Prediction Level Example

The instruction *s1* is not under influence of any condition, hence it has a prediction level of zero, notated as PL_0. For *s2* and *s3*, they have a prediction level

PL_1. If the *cond1* is still not resolved when *cond2* is predicted to be true, then *s4* and *s5* will have the prediction level PL_2. The two important observation about prediction level are:

1. It is a dynamic number showing the current number of conditions in speculation. In the example above, if *cond1* is resolved before the speculation of *cond2*, *s4* and *s5* will have predication level of one not two.

2. All instructions with the same control dependence will have the same prediction level, while the dependent condition is not resolved.

To identify speculative results, we devised the following modification for SAFA: each *Tag* in the *OTS* now has two more fields, a producer prediction level (**PPL**), and a consumer prediction level (**CPL**). The former indicates the prediction level of the instruction that produce that result, and the latter is the prediction level of the instruction that consume the result. Since a value on a stack can only be produced and consumed by one single instruction each, a single field to store the prediction level for the consumer and producer is sufficient.

A **PL** register is also added to record the current prediction level. It is increased whenever a branch (condition) is encountered and reduced to appropriate level whenever a branch has been resolved. The prediction level starts at zero to indicate no speculative instructions are underway. An instruction, upon issuing, will record the current prediction level in the PPL field of its result and mark the CPL fields of its operand(s). With these information, it is now possible to handle speculative execution properly.

Consider a result value with PPL of j, when its consumed by by instruction of k prediction level, there are only two possibilities:

1. $k = j$ implies that the result and the consumer are control dependent on the same condition. Hence, the result can be safely removed as in the basic scheme discussed in the previous section.

2. $k > j$ implies that that the consumer instruction is in a deeper control block. The CPL of the result should be marked with the k. However, it is *not* removed because the speculation at level k may be mispredicted.

Note that the condition $k < j$ cannot happen because of the way we handle prediction resolution, which is discussed later.

Observe that since there are now stack values consumed by speculative instruction, indicated by a non-zero CPL, consumer instruction have to look for the topmost *unconsumed* value, instead of just the topmost value.

Prediction Resolution

Eventually, the conditional branch associated with a prediction will be resolved. The result can either confirm or overturn the prediction, which would have radical effects on the state of execution. Below, each of the scenario is laid out in details.

Consider the case of single level prediction: speculative instructions have PL_1. During the period of speculation, there may be results produced (marked by PPL_1) or results consumed (marked by CPL_1) by these instructions. Upon the confirmation of the prediction, all PL_1 will be reverted to PL_0 (the prediction level of non-speculative execution). Through this act, some of the consumed results will now have PPL_0 and CPL_0, these results can then be removed normally from the OTS as well as any associated RoB entries. Besides, there may be speculative instructions sitting in reservation stations. The prediction level of these instructions will be reverted similarly to zero.

This basic scheme holds up surprisingly well even in the context of multiple level prediction. When a condition associated with PL_j is confirmed, instead of reverting back to PL_0, we fall back to the next highest unresolved PL_k that is smaller than j. Similar to the case above, all PL_j will be reverted to PL_k. This allows freeing up of more resource, while at the same time maintain the speculative execution at levels that are either greater or smaller than j.

Correct prediction traditionally is not hard to handle as opposed to misprediction, which require considerably more attention in most machines. In SAFA, on the other hand, misprediction is handled with similar process as in confirmation of prediction.

In the case of single level prediction, when the prediction of PL_1 is overturned, all results produced by the speculative instructions (marked with PPL_1), will be removed. Results consumed by these instructions (marked with CPL_1) will have to be "restored"[7] by simply clearing the CPL. Any speculative instructions caught on the reservation stations will be similarly flushed.

The situation become more delicate when multiple level prediction is considered. When a misprediction occurs at PL_j, speculative instructions as well as stack values may have PL that is greater or lesser than j. Any speculation with PL_k, where k is larger than j is invalidated by the misprediction, since they are indirectly control dependent on the condition associated with PL_j. Hence, all instructions and results that are associated with PL larger than j are processed in the same way with those with PL_j. The prediction level of the execution will then revert back to PL_k, which is the highest unresolved prediction level lesser than j. In this way, prediction that is not dependent on the mispredicted condition (distinguished by PL_i, where i is lesser than j) can continue the speculation undisturbed.

Example of Single Level Speculative Execution in SAFA

For all the subsequent examples, all conditions are predicted as *true*, i.e. the *if* block will be executed.

Consider the case of single level prediction:

Assuming each of the statements consume one and produce one result each, the Operand Tags Stack initially contains one result after statement *s1* is issued, results are tagged with the statement id for clarity:

[7] since it is not physically removed in the first place, it is not actually a full restoration

```
s1;
if (cond1){
    s2;
    s3;
}
```

Figure 4.6: Single Level Prediction

	Tag	PPL	CPL
Top→	s1	0	-

Upon execution of *cond1*, SAFA enter prediction level one. Statement *s2* will be executed with PL_1, giving OTS:

	Tag	PPL	CPL
Top→	s2	1	-
	s1	0	1

Note that the result of statement *s1* is marked but not removed. Statement *s3*, similarly, executes with PL_1. The OTS should looks like the following:

	Tag	PPL	CPL
Top→	s3	1	-
	s2	1	1
	s1	0	1

However, the result *s2* is clearly removable. Since it is consumed by instruction with the same prediction level, it will not be on the stack regardless of the prediction outcome. So, the actual OTS contains the following instead:

	Tag	PPL	CPL
Top→	s3	1	-
	s1	0	1

Assuming that the *cond1* is resolved and confirms our prediction, all PL_1 will be reverted to PL_0. Resulting in:

	Tag	PPL	CPL
Top→	s3	0	-
	s1	0	0

Observe that now the value *s1* is now safe to removed, producing the final OTS:

	Tag	PPL	CPL
Top→	s3	0	-

Suppose the prediction is wrong instead, all results with PL_1 will be purged. Any instructions consumed by instruction of PL_1 will be restored by clearing the CPL. The final OTS contains:

	Tag	PPL	CPL
Top→	s1	0	-

In both cases, the OTS contains the correct results as if the speculation never took place, which supports the validity of our scheme.

Example of Multiple Level Speculative Execution in SAFA

The following code fragment will be used to illustrate multiple level prediction in SAFA.

Since the first part up to the statement *if (cond2)* is similar to the previous example, discussion is resume at that point, with the OTS currently contain:

	Tag	PPL	CPL
Top→	s3	1	-
	s1	0	1

```
s1;
if (cond1){
    s2;
    s3;
    if (cond2){
        s4;
        s5;
    }
}
```

Figure 4.7: Multiple Level Prediction

Executing *cond2* elevate the prediction level to 2. Statements *s4* and *s5* will be executed in similar fashion as *s2* and *s3*, resulting in OTS:

	Tag	PPL	CPL
Top→	s5	2	-
	s3	1	2
	s1	0	1

If *cond1* is resolved first and confirms the prediction, PL_1 will revert back to the highest PL that is smaller than 1, which is 0 in this case. Resulting in:

	Tag	PPL	CPL
Top→	s5	2	-
	s3	0	2

The value *s1* is removed similar to the previous example. The prediction at PL_2 can continues undisturbed.

Suppose *cond2* is resolved first instead, and confirms the prediction, PL_2 will reverts back to PL_1, which gives:

	Tag	PPL	CPL
Top→	s5	1	-
	s1	0	1

Note that the result *s3* is removed, which is the correct behavior. Result *s3* will either be purged should the prediction of PL_1 be wrong, or consumed by *s4* which is already confirmed. Either way, it will not be on the stack. The reversion of the PPL of *s5* to 1 also shows that, with the *cond2* confirmed, it is control dependent on *cond1* instead.

Now, lets go back to the state where both of the conditions unresolved, and look at some alternative scenarios. Suppose *cond1* happens to be a misprediction, all prediction level greater than 1 will be purged. Results consumed by PL greater or equal to 1 will be restored, giving us the OTS:

	Tag	PPL	CPL
Top→	s1	0	-

The result of *s5* is correctly pruned because it is indirectly control dependent on *cond1*.

If the *cond2* is resolved first and contradicts the prediction, only PL_2 will be affected. With some of the results purged, while others are restored according to the scheme discussed, the OTS gives the following:

	Tag	PPL	CPL
Top→	s3	1	-
	s1	0	1

Again, the prediction of PL_1 can continue the execution uninterrupted. As summary to the speculative execution scheme in SAFA, several tables of appropriate operations are presented below:

4.3.3 Limitation of Speculative Execution in SAFA

The previous section gives a clear picture of the speculative execution in SAFA. However, there are some limitations in this scheme. Observe that the corner stone

Condition	Operation
PPL < CPL	Do nothing.
PPL = CPL	Remove the tag and clear associated RoB entry
PPL > CPL	Impossible.

Table 4.1: Speculative Consumption of Result

Result with	Operation
PPL $< j$	Not affected
PPL $= j$	Revert PL $< j$
PPL $> j$	Not affected
CPL $< j$	Not affected
CPL $= j$	Revert PL $< j$, remove if PPL = CPL
CPL $> j$	Not affected

Table 4.2: Confirmation of Prediction PL j

Result with	Operation
PPL $>= j$	Purged
PPL $< j$	Not affected
CPL $>= j$	CPL Cleared
CPL $< j$	Not affected

Table 4.3: Handling Misprediction at PL j

of SAFA speculative execution is that inherent “versioning” capability of the stack items. However, there are instructions that alter other parts of the CPU, for example a *memory store* instruction that changes the memory state, or a *index increment* instruction that changes the frame register. These operations do not have built in versioning capability, which make the required roll back operation in speculative execution hard to implement. Although there are well studied technique in GRP machines that can be adapted to SAFA, we decided to implement only stack related speculation. In this way, the benefit of speculation in a stack architecture can be better understood.

This decision also implies that there are two groups of instructions in SAFA:

those that can be speculated and others that can not be speculated. As mentioned before, the second group of instructions are those that change a CPU state that cannot be restored easily. These includes:

- Memory store instructions.
- Field-changing frame register instructions. Memory store instructions via frame register.
- Procedure entry and exit.
- Local Data Map (discussed in Section 4.4) *own_wstore* instruction, which modifies the LDM.

The instructions in this group (total 49 instructions) will stall the execution during speculative execution, as soon as they are decoded. Since majority of the instructions including memory load and computations are not in this group, there are usually substantial speculatable instruction after a branch.

4.4 Coping with Frequent Memory Movements

The previous two sections laid down a strategy to disentangle dependencies between stack instructions to allow parallel execution. However, there is a underlying assumption about the nature of the instruction stream, namely the instructions should be mostly stack-to-stack instructions for the discussed strategy to show its full potential. This implies that any temporary result laying on top of stack should be consumed by subsequent instructions without the need to be stored away. Among HLP paradigms, a pure-functional language[8] would be a perfect match.

However, given the popularity of imperative HLPs (including procedural and object oriented languages), the strategy may need to be reviewed. The obstacle

[8] Each function relies only on the result of previous function, i.e. there is no side effect, no need to store a intermediate result

posed by these language is the use of variables for value storage, which may be accessed and modified frequently throughout a program. As the usage of these variables does not form any discernible pattern, there may be a huge gap between the modification and utilization of a value. Consider the following example in C-Like pseudocode:

```
void f()
{
    float i,j;

    i = 3.14;       //Instruction A
    .....           //Intervene Code
    .....
    j = i * i * 123; //Instruction B
    .....
}
```

Instruction A modifies the value of variable i, which is later accessed in *Instruction B*. However, since there can be any number of instructions lying in the *Intervene Code*, it is not realistic to assume that the value of i can stay on top of stack without compiler's help. A stack compiler can try to do the following:

1. Leaves the value of i on top of stack.
2. Let any new values produced by *Intervene Code* to pile on top of the value i .
3. Manipulates the stack just before *Instruction B* to bring i value to the stack top for subsequent access.

This strategy requires the compilers to be able to keep track of the position of *all* temporary values at *any* time in execution. As can be imagined, this is not trivial and impossible under certain scenarios. For example, when the *Intervene Code* contains a loop or a branch, there is no easy way to keep track of the position of value i.

Hence, in conventional stack compiler, another approach is taken. All variables, including procedure parameters, are stored in a temporary memory location,

usually in the stack frame of the procedure. Any modification of these variables will be stored away to the corresponding memory location. Access of the variables will then be translated as memory load operation. In other words, as each instruction corresponds to an expression tree, the top of an expression tree will be a memory store operation. Using the popular Java compiler as an example, the code fragment above would correspond to the following Java assembly code:

```
Method f()
//i is store as local variable #0
//j is store as local variable #1

    ldc #2 <Real 3.14>  //Instruction A
    fstore_0                //store i
    ....                //Intervene Code
    ....
    fload_0             //Instruction B
    fload_0                 //load i
    fmul
    ldc #3 <Real 123.0>
    fmul
    fstore_1                //store j
    ....
```

By storing away all modifications on variables, Java compiler manages to avoid the complications posed by the *Intervene Code*. However, the ease of compilation comes with a huge cost, namely a huge amount of memory instructions is inserted into the program. In [45], by sampling across several programs in *SpecJVM98*[50] and *Java-Grande*[51] benchmark suites, it is shown that there are on average 34% of the total instructions in a program for the this purpose alone. The memory instructions can degrade the performance of a program in at least two ways. Firstly, the huge memory latency delays the execution pipeline, although this can be somewhat mitigated by data cache. Secondly, the memory access unit can only read or write limited number of memory locations (also known as access ports) at any time cycle. The two facts combined can easily force a parallel stream of instructions to be executed serially.

The most straightforward solution to this problem is to map variables to registers[45]. A number of different designs on this theme are discussed in Section 4.5. For example, the *picoJava* designed by *Sun Microsystems* utilize a register file (stack cache) for stack items, including local variables. A memory load/store of a local variable is translated to an equivalent read/write of a register. This basically transform the execution of Java bytecode into register-like execution (more in Section 4.5). However, there are several shortcomings:

- Multiple instructions that write to local variables cannot be executed in parallel as there is no renaming to resolve the data dependency.

- The stack cache is split into two areas: local variables and intermediate computation results. Two registers, **VARS** and **OPTOP** are used to keep track of the two areas respectively. Local variables are mapped as an offset of **VARS** register, which requires the local variables to be in consecutive slots in stack cache.

There are a number of researches aimed to improve on the *picoJava* design, some of which are also discussed in Section 4.5. One possible way to solve this under SAFA architecture will be presented next.

4.4.1 Local Data Access in SAFA

As SAFA has already a good foundation for tagged execution (register renaming), it is clear that the best way is to map the local variables to an entry in the RoB, which allows renaming and random access. With the main problem out of the way, there are only several minor problems to be solved:

1. A way to distinguish a local variable from intermediate computation.

2. A way to indicate a read/write to local variable.

3. A way to setup the local variable on RoB when the procedure is first entered.

4. A way to store these local variables into memory before calling another procedure.

The first problem can be solved by a simple observation: a temporary computation result will be consumed immediately, while values of local variables may not be needed immediately. In SAFA architecture, only entries on the OTS will be consumed by subsequent instruction. Previously, the virtual tag number of any value in RoB *must* exists on OTS. As soon as the virtual tag number is popped from OTS, the *Use Count* of the corresponding RoB entry is decremented. A RoB entry will be deleted if the *Use Count* reaches zero. So, firstly, we can easily keep a local variable in RoB by *not* putting its associated virtual tag number on OTS. Secondly, when a local variable is needed in computation, the corresponding virtual tag number can be loaded onto the OTS, and the *Use Count* of the RoB entry is incremented at the same time. In this way, the *Use Count* of a RoB entry that stores a local variable is always equal to *Number of appearances in OTS + 1*, which ensures the RoB entry will not be removed. Likewise, when a new value is to be stored into a local variable, we can achieve this by simply *changing* the mapping (local variable to virtual tag number) and decrement the *Use Count* of the old entry.

We follow the Java terminology, which groups the parameters and local variables of a procedure as local data (LD). The LDs, starting from parameters then followed by local variables, are labeled with a index number starting from zero. For example:

```
void f(int a, int b)
    int i;
    float j;
```

Then the parameters a and b are labeled as LD 0 and LD 1. The local variable i and j are labeled as LD 2 and LD 3.

Two new instructions are provided for LD access in SAFA:

Instruction	Parameter	Description
own_wload	N	Load the corresponding virtual tag number of LD *N* onto OTS. Increase *Use Count* of the corresponding RoB entry by 1.
own_wstore	N	Take the virtual tag number on top of OTS and change the mapping of LD N. Decrease the old RoB entry by 1.

The mapping between LD and its virtual tag number is kept in an array *Local Data Map (LDM)*, where *LDM* [***I***] gives the virtual tag number for LD I. A SAFA translation of the code example used previously is given below:

```
PROC f
//i is store as  LD #0
//j is store as  LD #1

    iwload <f3.14>      //I1: immediate load
    own_wstore 0        //I2: store to LD 0
    ....
    ....
    own_wload 0         //I3: load from LD 1
    own_wload 0         //I4: load from LD 1
    fmul                //I5: multiply
    iwload <f123.0>     //I6: immediate load
    fmul                //I7: multiply
    own_wstore 0        //I8: store to LD 1
    ....
```

Before the execution of instruction I1, the corresponding state of RoB, OTS and LDM are given next.

RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	–	–	0	–
1	–	–	0	–
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	–
1	–
⋮	⋮

Operand Tag Stack	[→]

We have the following states after execution of instruction I1 *iwload*:

RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	0x00	iwload	1	3.14
1	–	–	0	–
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	–
1	–
⋮	⋮

Operand Tag Stack	[→0x00]

Next would be the first example of local data map usage. Instruction I2 *own_wstore 0* changes the state and gives:

RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	0x00	own_wstore	1	3.14
1	–	–	0	–
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	0x00
1	–
⋮	⋮

Operand Tag Stack	[$\rightarrow$]

Note that the change on LDM and OTS. The RoB entry 0 is protected from deletion as the *Use Count* is still one. After execution of I3 *own_wload 0* and I4 *own_wload 0*:

RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	0x00	own_wstore	3	3.14
1	–	–	0	–
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	0x00
1	–
⋮	⋮

Operand Tag Stack	[0x00, $\rightarrow$0x00]

By loading the virtual tag number of `LD 0` onto OTS twice, we now effectively have two copies of the value 3.14. Note the change of *Use Count* of the corresponding RoB entry.

Instruction I5 *fmul* gives the following state:

RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	0x00	own_wstore	1	3.14
1	0x05	fmul	1	Waiting
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	0x00
1	–
⋮	⋮

Operand Tag Stack	[→0x05]

The instruction *fmul* will take the two copies of 3.14 by decreasing the corresponding *Use Count*, it is then dispatched to the floating point unit. Note that the virtual tag number *0x05* is chosen randomly to make the example more interesting.

After I6 and I7:

RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	0x00	own_wstore	1	3.14
1	–	–	0	–
2	–	–	0	–
3	0x0a	fmul	1	Waiting
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	0x00
1	–
2	–
⋮	⋮
⋮	⋮

Operand Tag Stack	[→0x0a]

The reservation entry for instruction I7 *fmul* will have the virtual tag *0x05* as the result is not ready, and the value 123.0 loaded by instruction I6. Note that both RoB entries for instruction I5 and instruction I6 have been deleted as the *Use Count* reaches zero.

Finally, the instruction I8 *own_wstore 0* give the following machine state:

RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	0x00	own_wstore	1	3.14
1	–	–	0	–
2	–	–	0	–
3	0x0a	fmul	1	Waiting
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	0x00
1	0x0a
2	–
⋮	⋮
⋮	⋮

Operand Tag Stack	[→]

Again, by popping the tag from OTS but leaving the *Use Count* intact, we have created another local data that is safe from deletion. Note also that the availability of a value does not affect the execution of *own_wload* or *own_wstore* as they depends only on virtual tag number but not actual value.

The described scheme gives a satisfactory answer to the first two problems stated at the beginning of this section. We can now look at the more technical problems posed.

The solution of setting up the local data for a procedure is related to the way that a stack frame is constructed during procedure activation. In SAFA, the caller is responsible for setting up the stack frame, filling in all the required data items (Section 3.1.1), including the procedure parameters. Hence, when a procedure is started, the parameters must be loaded from memory to setup the scheme above correctly. For example, to setup the procedure f above, a few more lines of code are needed:

```
PROC f
    cfb_wload x24       //I1: Load 1st parameter from memory
    own_wstore 0        //I2: store as LD 0
    cfb_wload x28       //I3: Load 2nd parameter
    own_wstore 1        //I4: store as LD 1
    ....
```

The instruction *I1* (refer to Chapter A) loads a word from current frame at offset *0x24* (more details in Section A.7), then *I2* will set the value up as *LD 0*. Similarly, the next pair of instructions *I3* and *I4* setup the *LD 1*. Obviously, as the number of parameters increase, more extra coding are needed. Whether this overhead is justifiable largely depends on the number of local variable accesses in a procedure.

One way to reduce these overhead is to modify the handling of a stack frame. The caller can leave all the actual parameters on the operand stack instead of storing them into memory and let the callee to set them up. This further reduce the memory traffic as more items are readily available on operand stack when a procedure is activated. These two ways of procedure activation are given a quantitative study in Chapter 6.

As the *LDM* is shared between all procedures, a caller has to protect its local data from the callee by sending them into the memory. The operations are analogous of the setup of local data. For example, to save the two local data of procedure *f*, the following code is needed:

```
....
own_wload 0         //I1: load  LD 0
cfb_wstore x24      //I2: store in memory
own_wload 1         //I3: load  LD 1
cfb_wstore x28      //I4: store in memory
....
```

The above code will transfer the current values of local data into the corresponding location in the stack frame. After calling a procedure, it is now necessary to re-setup the local data before continuing. Again, these overhead is only justifiable if there are frequent access of local data.

With the above scheme, it can be seen that all local variable accesses are now transformed into stack-to-stack instruction that bypass the memory unit. This should reduce the memory traffic as well as increase the ILP. However, there are a few drawbacks on this scheme:

1. As each of the local data will stay in the RoB for a long time, this reduce

the available RoB entries for intermediate computation results. However, the reservation station scheme should minimize the reliance on actual RoB entry.

2. The current implementation lacks the ability to spill some of the local data into memory, because the drizzling is based on the OTS entries.

3. As stated in Section 4.3.2, all state changing instructions are currently non-speculative. The instruction *own_wstore* is not speculatable, while *own_wload* has no problem.

4. Increase of complexity and on chip storage.

Speculative Execution and Local Data Map

As SAFA allows both speculative execution and local data map, it is important to study the interaction between these mechanisms to ensure correct program execution. As an informal proof, we will look at all possible prediction resolutions involving a LDM instruction.

Consider the following code fragment:

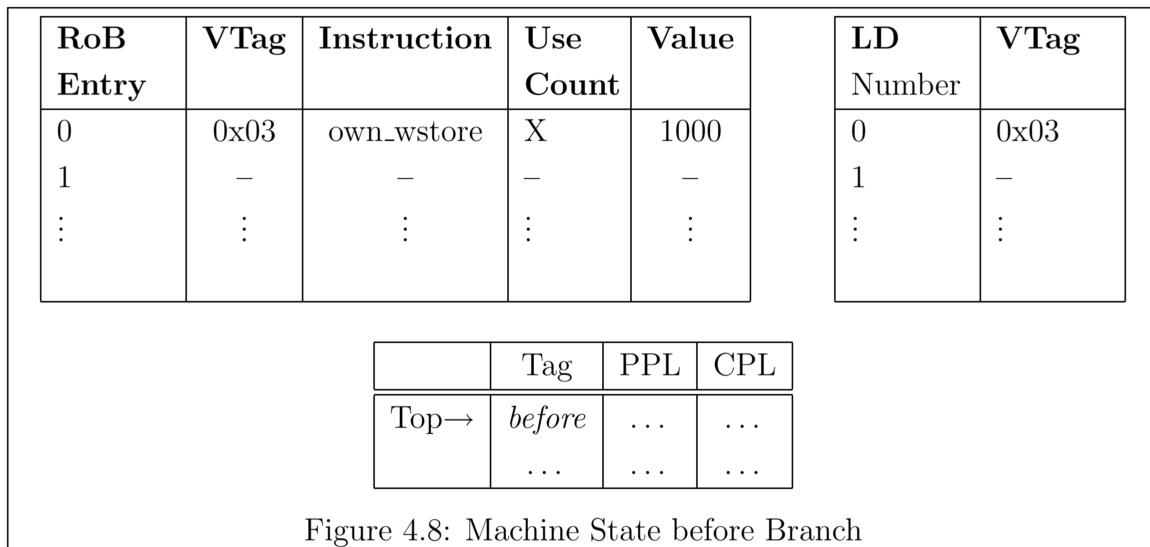
```
    ....
    ....                    //before speculation
    if_true end             //branch instruction
    own_wload 0             //SI1: load from LD 0
    ibload 234              //SI2: load value 234
    iadd                    //SI3: add integer
    ...                     //PointA
    ...
end:....                    //PointB: Branch Target
    ....
```

We assume the LDM is consistent before the speculation, which includes the following conditions:

- The LDM entries are setup correctly, i.e. each entry map to a existing RoB entry.

- The Use Count of the RoB entry associated with local data is X, where $X \geq 1$.

Depending on the outcome of the branching, speculative execution should result in the same machine state at either *PointA* or *PointB* in a non-speculative setting. The machine state before the branch instruction is as follows:



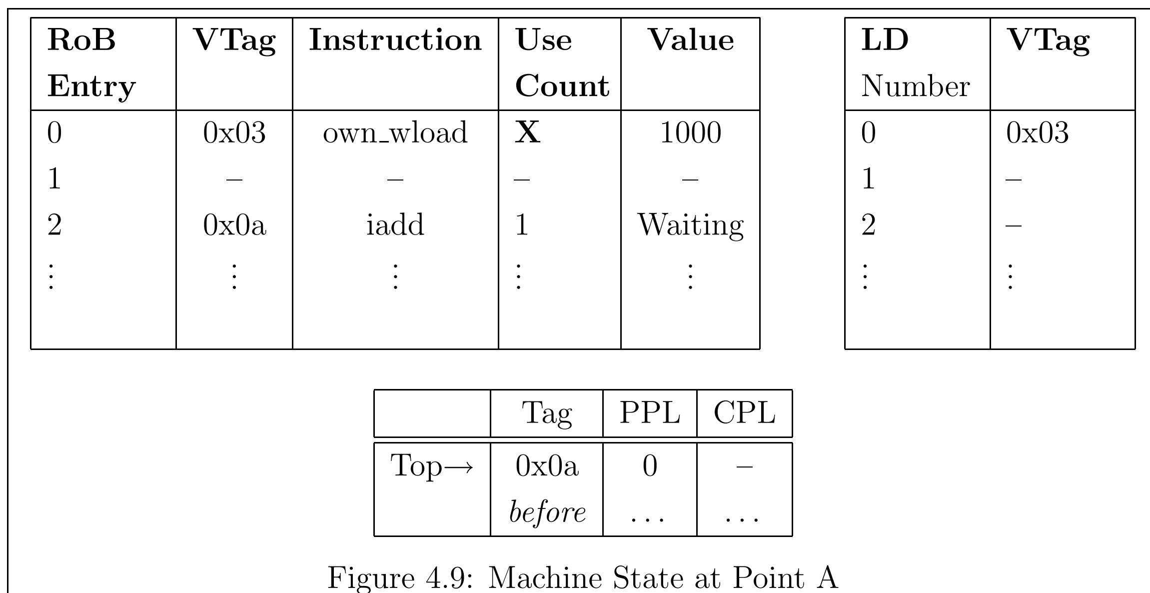
RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	0x03	own_wstore	X	1000
1	–	–	–	–
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	0x03
1	–
⋮	⋮

	Tag	PPL	CPL
Top→	*before*	...	...
	...	...	...

Figure 4.8: Machine State before Branch

If the branch *if_true* is **taken**, then the machine state at *PointB* would be the same as Figure 4.8. On the other hand, if the branch is **not** taken, then the machine state at *PointA* is shown in Figure 4.9.



RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	0x03	own_wload	**X**	1000
1	–	–	–	–
2	0x0a	iadd	1	Waiting
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	0x03
1	–
2	–
⋮	⋮

	Tag	PPL	CPL
Top→	0x0a	0	–
	before	...	...

Figure 4.9: Machine State at Point A

Suppose the branch *if_true* is predicted as *not taken*. The instructions *SI1*,*SI2* and *SI3* are executed speculatively. Upon encountering the branch instruction, the prediction level will be increased to 1. The machine state after the execution of instruction *SI1* is shown below:

RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	0x03	own_wload	**X+1**	1000
1	–	–	–	–
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	0x03
1	–
⋮	⋮

	Tag	PPL	CPL
Top→	**0x03**	**1**	–
	before	...	...
	...	...	...

Suppose the prediction is resolved at this point, we can then observe the following changes depending on the resolution outcome. In the case where prediction is *successful*, the prediction level PL_1 will be restored to PL_0.

RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	0x03	own_wload	**X+1**	1000
1	–	–	–	–
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	0x03
1	–
⋮	⋮

	Tag	PPL	CPL
Top→	0x03	**0**	–
	before	...	...
	...	...	...

The subsequent instructions *SI2* and *SI3* will then be executed non-speculatively, resulting in the following machine state:

RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	0x03	own_wload	**X**	1000
1	–	–	–	–
2	0x0a	iadd	1	Waiting
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	0x03
1	–
2	–
⋮	⋮

	Tag	PPL	CPL
Top→	**0x0a**	**0**	–
	before	...	...
	...	...	...

As can be seen, this is exactly the same machine state as in Figure 4.9. In particular, the *Use Count* is decreased back to *X*.

Suppose the prediction fails, all entries on OTS with PL_1 will be purged. As a VTag is removed from OTS, the corresponding *Use Count* will also be decreased. The resulting machine state is as follows:

RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	0x03	own_wstore	**X**	1000
1	–	–	–	–
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	0x03
1	–
⋮	⋮

	Tag	PPL	CPL
Top→	*before*	...	...
	...	...	...

This is exactly the same machine state as shown in Figure 4.8, which shows that any effects of speculative execution are removed correctly.

Since speculation can be resolved at any point in time, lets look at another two scenarios where the resolution come after the speculative execution of *SI3*. The machine state at that point is given below:

RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	0x03	own_wload	**X**	1000
1	–	–	–	–
2	0x0a	iadd	1	Waiting
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	0x03
1	–
2	–
⋮	⋮

	Tag	PPL	CPL
Top→	**0x0a**	**1**	–
	before	...	...
	...	...	...

It is a near replica of Figure 4.9, where instructions *SI2* and *SI3* are exe-

cuted non-speculatively, except the PPL of the *iadd* is one instead of zero.

In the case where the prediction is successful, the PL_1 is simply decreased to $PL - 0$. Resulting in the same machine state as in Figure 4.9.

On the other hand, if the prediction is wrong. The result of instruction *iadd* will be purged because of its PPL_1. The removal of the vtag also decrease the *Use Count* of the associated RoB entry, resulting in the machine state:

RoB Entry	**VTag**	**Instruction**	**Use Count**	**Value**
0	0x03	own_wload	**X**	1000
1	–	–	–	–
2	0x0a	iadd	**0**	Waiting
⋮	⋮	⋮	⋮	⋮

LD Number	**VTag**
0	0x03
1	–
2	–
⋮	⋮

	Tag	PPL	CPL
Top→	*before*	...	...
	...	...	...

The RoB entry is subsequently removed and restored the machine state as in Figure 4.8. Since virtual tag removal upon misprediction is handled just like normal consumption, the *Use Count* of the associated RoB entry will be kept in a consistent state. Besides, as the state changing instruction *own_wstore* is barred from execution, the *LDM* can be kept consistent during speculation. The scenario above should illustrate the process clearly.

4.5 Advances in Java Technology

The central promise of Java technology "Write once, Run anywhere" [13] has an unexpected impact on the computer architecture scene. To keep the promise, a Java binary (Java Bytecode Code **JBC**), unlike other executable, is interpreted and executed by Java Virtual Machine (JVM). This isolation from actual processor architecture ensures the portability of JBC, but at the same time, degrades the performance because of the interpretation overhead. As Java growing ubiquitous, the pressure for faster execution prompts active research.

One major direction of these researches is to provide an actual Java Processor. As surveyed in Section 2.3.5, Sun Microsystems developed and marketed two Java Processors, *picoJava I* and *picoJava II*. The central idea of these processors is the combination of *stack cache* and *instruction folding*[13].

The stack cache is a 64 entries register file that caches the top 64 entries of the stack. Frequently accessed data like local variables, parameters etc are first pushed onto the stack. Since the stack cache is actually a register file, this allows random access of any entry which eliminates the reliance on the stack pointer. The picoJava design further capitalize on stack cache by introducing the instruction folding mechanism.

To illustrate this mechanism, consider the JBC fragment:

```
iload_0     //load local variable 0
iload_1     //load local variable 1
iadd        //integer add
istore_2    //store to local variable 2
```

Assuming that each instruction can be executed in one time cycle, then the above would requires four time cycles. Since the two local variables are already in the stack cache, the execution above is grossly inefficient. The first two instructions did nothing useful, except moving values that is already in CPU core to another location (the top of stack). Instruction folding is the mechanism that detects such pattern

(folding group) and translate them into a single GPR-like instruction. For example, the above can be translated to:

```
add R0,R1,R2    //Assuming RX stores local variable X
```

This instruction can then be executed in one cycle. As multiple instructions are *folded* into a single instruction, this mechanism is aptly named instruction folding. The above example is also an example of *4-foldable* group as four instructions are folded.

For actual implementation, the folding logic is added to the decoding stage in picoJava core, as shown in Figure 4.10. Using a set of grouping rules, the decoded instructions (up to four instructions) are scanned for foldable sequence. The matched sequence is then folded into a single GPR-like operation. Because of the limitation of the matching windows, the 4-foldable is the biggest possible group.

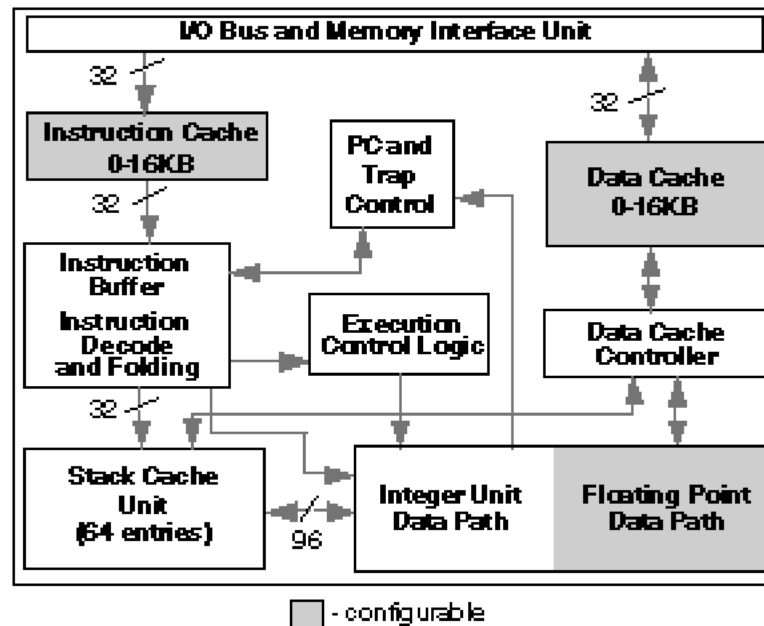


Figure 4.10: Sun Microsystems picoJava Block Diagram

This mechanism relies on the grouping rules, which heavily influence the possible folding. Since there are a huge number of foldable patterns, only a selected few are actually implemented because of hardware complexity. The grouping rules

implemented by picoJava achieves an average of 20% folding rate (percentage of instructions folded) for typical Java programs[13]. However, in [47], it is noted that there may be better grouping rules. The research [52] suggest instead of fixed grouping rules, instructions that modify either the value or the order of stack items can serves as *anchor instructions* of a folding group. As soon as an anchor instruction is encountered in the folding unit, the preceding instructions are scanned for producer(s). In the case that *anchor instruction* requires a consumer, the immediately following instruction is checked. A folding group is thus formed starting from the producer instruction(s) up to the consumer instruction, or in the case of missing consumer, up to the anchor instruction. After a folding group is formed in this way, the group is then removed to allow further folding of the remaining instructions. This technique, named as *Operand Extraction Based (OPEX)* allows more flexible folding groups with comparable hardware complexity.

The scanning and matching operations required by instruction folding come at a cost. In [54], it is found that the Instruction Folding Unit (IFU) falls in the critical path which decrease the possible clock rate. Both [54][53] suggest moving the complex logic off the critical path, such that the folding can occur in parallel with the main pipeline stages.

Adding in superscalar support is the natural next step for the Java Processors. [54] take a first look by allowing the picoJava *execute* pipeline stage to execute independent instructions from a basic block in parallel. The result gathered from five programs in the *specJVM98*[50] benchmark suite is not very encouraging. It is found that only 7.638% of the instructions on average can be executed in pair, with highest at 9.88% and lowest at 5.30%. Only negligible percentage of the instructions can be executed three at a time. The poor performance is attributed to the restriction that any instructions that write to the stack will stall the following instructions that read from the stack to preserve data dependency. As the stack cache actually serves two roles: storage of the local variables and frame data; storage of the operand stack. Since operand stack is allocated *after* the local variable area, they

are practically independent of each other. Performance can be improved by allowing operations acting on different *area* of the stack cache to proceed in parallel. This technique is given the name *stack disambiguation*[54] as it extricate the false dependency on the stack cache. The parallelism increased greatly by this simple addition: an average of 16.292% of the instructions (highest:21.06%, lowest:11.56%) now can be executed in pair, and average of 0.74% instructions (highest:0.17%, lowest:2.00%) can be executed in 3's.

Following the same line as the *stack disambiguation* technique, [53] proposed to adapt Tomasulo's Algorithm[38] for a instruction folding based Java Processors. The main ideas of the research are:

1. Use RISC-like register file.
2. Utilize the OPEX instruction folding strategy[52]. When the consumer of a folding group is missing, a tagged register entry is used as temporary storage. This tag is later picked by the consumer instruction. This renaming of register entries decouple the dependency between folding groups and allow multiple issues.
3. With instruction folding, this design dynamically translate JBC to typical RISC instructions. This allow utilization of RISC techniques.
4. Employ reservation station of the Tomasulo's Algorithm to allow instruction to be dispatched even when the operands are not ready.

As explained before, the OPEX folding strategy will remove a folded group to allow for recursive folding. However, this may cause pipeline hazards where a consumer may get issued *before* a precedent producer. This hazard can be detected by checking the not yet folded JBC to look for any preceding producer that uses the same local variable as the consumer of the current folding group. The precedent producer is then replaced by a folding group that produce the value into a tagged register entry T. This T is then added as producer in the JBC for further folding.

As an interesting side-note, this design also substantiates our proposed General Tag Execution framework (Section 1.1). This design opted for a JBC→GPR instruction translate for the the stage two of the framework (dependency checking). An adapted Tomasulo's Algorithm serves as scheduling and dispatching stage. The execution stage is simply a RISC-like GPR execution core. This shows the power of casting into existing execution model for well studied techniques.

With the relevant researches described, we present a comparison between the SAFA architecture and the Java Processors in the next section.

4.5.1 Comparison: SAFA vs Java Processors

Before a feature based comparison is presented, it is illuminating to briefly study the *design philosophy* between SAFA and Java Processors. This would give us a deeper understanding of the *choices* made. The SAFA design concentrates on a stack based architecture and try to rein in the inefficiency that such architecture holds. The Java Processors on the other hand, work under the assumption that register based architecture is more efficient and concentrates on ways to translate execution of JBC into register based execution. This is perhaps most apparent in the instruction folding mechanism, where each folding group map to one register instruction.

On closer inspection, it can be seen that the instruction folding mechanism is actually a special case of tagged execution in SAFA. A folding pattern re-establish the producer-consumer relation for the selected instructions group. As pointed out before, there are so many foldable patterns that only limited pattern can be fitted into the decoding logic. The rigidness of the folding pattern and also the limited number of instructions inspected prevent more instructions to be folded. In SAFA, on the other hand, the relation between procedure and consumer is reconnected as the instruction is issued, with the help of operand tag stack. Regardless of the depth or complexity of the relation, *all* instructions that reach the execution stage

would have picked up theirs operands automatically, i.e. "folded". Also, as pattern matching is not a trivial operation, it is understandable that the instruction folding in Java Processor (at least in the two actual hardware picoJava I and II) causes delay in the execution path. The relatively simpler operation proposed by SAFA would go toward in lowering the complexity.

The Java Processor architecture proposed by [53] answered some of the criticism above by introducing new folding strategy and also superscalar execution. However, as the register renaming is partial, additional logic must be added to detect and resolve pipeline hazard caused by folding. Register renaming is uniform in SAFA, where each producer instruction tags a new register entry, which prevents any extra hazard.

SAFA architecture also proposed a way to provide speculative execution that is consistent with the central tagged execution. As the speculation in SAFA is based on the inherent characteristic of stack, speculative execution can be added conveniently. This feature is not in the most of the Java Processor designs, mainly caused by power consumption and complexity issues.

Finally, with the local data map scheme, SAFA provides a more flexible mechanism for local data access as compared to the stack cache approach. The optimization proposed fit into the basic SAFA architecture easily, ensuring the applicability of the previously discussed techniques.

On the other hand, Java Processor have one major advantage as instruction folding is applied at the decode stage, which reduces the number of instructions issued. In SAFA, since there is no reduction of instruction count at the decode stage, more instruction must be issued. Hence, it is likely that a higher issue rate is needed to achieve good performance as compared to Java Processor.

In a nutshell, SAFA can be considered as a viable alternative design for stack architecture as opposed to Java Processors.

4.6 Influence of General Tagged Execution Framework

As promised in the opening of this chapter, we will briefly described the process of arriving at the current design for SAFA guided by the General Tagged Execution Framework (GTEF) before closing the chapter.

Recalls that Figure 1.1 in Section 1.1 depicted a general multi-stage instruction execution process, the General Tagged Execution Framework. This abstract framework shows that by associating the values/operands of an operation with a tag, we can gain a more general perspective. GTEF concerns with tags, instead of actual values. An instruction in this abstract machine manipulates tags, which can either be actual values or temporary identifiers for as yet uncompleted execution result.

One obvious benefit gained by working through this framework is that we can pick and match existing well studied mechanisms to complement the intended design. By mapping each stage in the GTEF with actual hardware mechanism, we will then get an instance of this framework. The SAFA architecture can serve as a good example of this process. As one of our main objectives is to derive a superscalar stack machine, the instruction stream (corresponds to stage one in Figure 1.1) is obviously going to be stack based. The execution core (stage four in GTEF) is directly fixed by our decision to allow superscalar execution. Since the instruction coming out of stage three, specify its operands and results as tags, the fact that we are dealing with stack based instructions ceased to make any difference. In short, any execution core capable of executing multiple tagged instructions can be picked. Looking back at the SAFA architecture, it can be seen that this is exactly what Tomasulo's Scheme provided: a superscalar execution core.

With the first and last few stages of the GTEF fixed, the rest of the pieces fall into place naturally. A dependency resolution and tagging mechanism is needed

at stage two, which should establish the dependency between stack based instructions by tagging. A scheduling mechanism is then needed to dispatch the transformed instruction into the execution core. Now, with hindsight, it is clear that this is exactly provided the combination of Reorder Buffer Scheme and Operand Tag Stack.

We do not claim that random picking any existing mechanisms for the GTEF would miraculously generates a viable design. Obviously, conscious effort must be taken to pick the correct ideas and iron out any incompatibility especially in between of two stages. GTEF serves as a guiding light for us to look in the correct direction and provide a more systematic approach at arriving at a plausible design.

4.7 Summary

This chapter present our main ideas to improve low level execution in a stack-oriented machine. Adapting well known superscalar ideas in general purpose register machines, we have shown that multi-issue, out-of-order execution is possible in a stack machine setting. With data dependency reined in, branch prediction and speculative execution are now powerful techniques to provide even more opportunity to improve the performance of our model cpu, SAFA.

Chapter 5

Benchmark Environment

This chapter describes the setup of the benchmarking process. The SAFA "hardware" is covered in Section 5.1 with details such as the hardware/software tools used. The description, as well as the rationale of the chosen benchmark programs can be found in Section 5.3.

5.1 Hardware - SAFA Simulator

As SAFA is a new cpu architecture, there is no actual fabricated chip to execute the SAFA programs. To provide a platform for experimentation, we have implemented a software simulator of the actual hardware. A *C++* program is written as a component level software simulator (a functional simulator [32]) for the SAFA architecture. This simulator provides accurate per time tick (also know as *CPU cycle*) view of the components in the CPU as well as its external memory (RAM) unit.

The benefits of choosing to implement a software simulator from scratch are:

- Provide the flexibility to implement/re-implement the components if needed.

- Give a clearer view of the low level details, which may spark off insights to improve the current design.

Instead of component level software simulator, there are other alternatives, for example, FPGA (Field Programmable Gate Arrays), VLSI (Very Large Scale Integration) simulator, Verilog HDL etc. These alternatives provides even more detailed information of the hardware, usually down to the logic gate level. However, we did not chose these alternatives because:

- The scale of the SAFA architecture is quite prohibitive to implement on these alternatives, with generating additional insight into the value or otherwise of our architectural ideas.

- There are low level details, such as logic gate fan-out, power consumption, interconnection, layout, clock signal distribution etc, which requires extensive experience and skill to handle correctly. However, it is important to separate implementation issues from design issues to keep our aim in view. SAFA architecture is essentially a new hardware design, which should be observed at a higher conceptual level to see the benefits of the proposal.

Figure 5.1 shows the Component Level Diagram for the SAFA architecture. There are a number of parameters (different settings) for each of the component, which can drastically change the execution in SAFA. Brief explanation for each component, with the associated parameters, is given next.

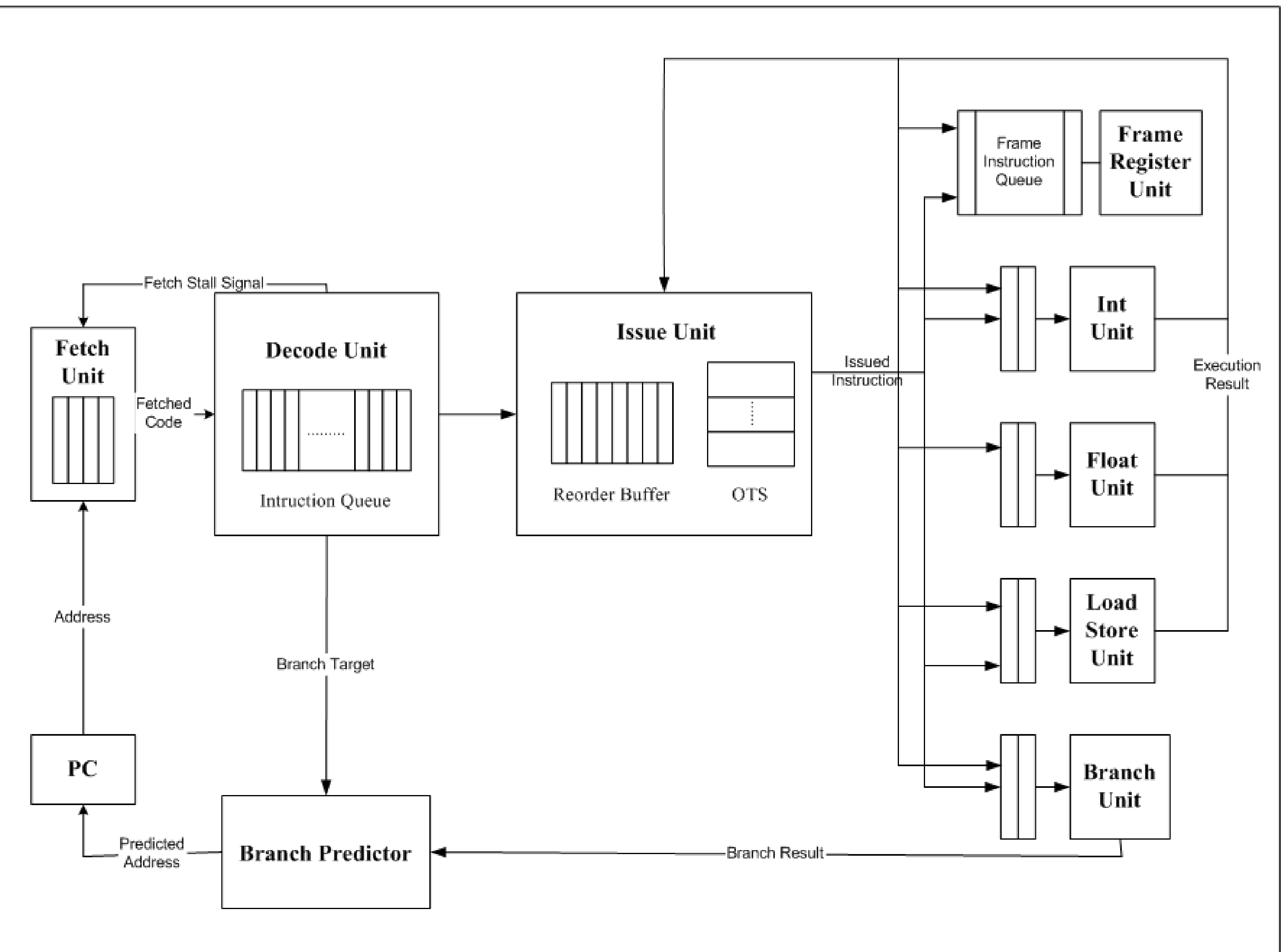


Figure 5.1: SAFA Components Diagram

5.1.1 Fetch Unit

Fetch Unit has a well-defined task, i.e. to retrieve instructions from the memory unit to be executed. Although instructions in SAFA are mostly byte size, a number of memory words (Fetch Size[1]) are fetched every time tick instead of a single byte. This allows more instructions to be fetched and stored at every time tick, to fully utilize the memory bandwidth. The fetch unit may stall under the following conditions:

1. When the next component (the decode unit) is unable to take in any more new instructions.
2. When a branch instruction is decoded by the decode unit. This behavior may be modified when branch prediction is turned on (refer to the section on *Branch Predictor*).

5.1.2 Decode Unit

After the raw code is fetched, the decoder unit proceeds to decode a number of instructions (Maximum Decode) and perform the following:

- A number of instructions have associated immediate operands (effectively making the instructions multiple byte), such operands are retrieved and passed along to the next stage. If an operand is not fetched yet, stall until it arrives.
- The unit calculates branch target for branching instructions. It also outputs a stall signal for the fetch unit if branch prediction is not in effect.

The decode unit may stall under the following conditions:

1. The instruction cannot be fully decoded, since some part of the instruction is yet not fetched.

[1] all setting (parameters) uses San Serif font

2. Run out of fetched raw code.

3. The next stage (*Issue*) is not yet ready.

To minimize the chance of the stated conditions, an instruction queue with Instruction Queue Size bytes serves as a temporary storage for fetched raw code. Since the fetch unit has a throughput of Fetch Size bytes per time tick, filling up the instruction queue does not take too long. If an immediate operand is needed, it is likely that it is already in the queue, allowing the decoder to proceed. Also, when "multiple decode" is enabled, having a instruction queue serves as buffer to lessen the chance of running out of raw code.

At the end of every cycle, the decoder will make sure that there are at least enough free space (a memory word, 4 bytes) in the instruction queue, anticipating the next output of the fetch unit. If the queue is almost full, a halt signal will be sent to the fetch unit to stall it for the next cycle. The signal will be cleared as soon as enough room in the queue is freed.

5.1.3 Issue Unit

This is the component that differs from conventional machines. It consists of the re-order buffer (RoB), modified to implement the *multiple logical stacks* idea discussed in Chapter 4, and the operand tag stack (OTS), to keep track of the input-output dependence between instructions.

This unit is in charge of issuing/dispatching a number of instructions (Maximum Issue) to respective execution units. The more important aspects of instruction dispatching are listed below:

- For the operand(s) of an instruction: Pop the require number of operand tag from OTS. Acquire actual value from RoB entry if the value is available, or virtual tag number if the value is in transit.

- For the output of an instruction: Acquire free reorder buffer (RoB) entry/entries and attach virtual tag number. Push the tag on to operand tag stack.

Since we follow the reservation station scheme (explained Section 4.2.2) for the execution units, the above steps essentially transforms an instruction into corresponding reservation station entry. When the execution unit produces result, it is broadcasted along with the owner tag. Upon receiving any result, the issue unit performs one of the following action:

- If reorder buffer entry containing the *Owner Tag* exists, then records the value into the *Value* field of the corresponding reorder buffer entry.

- If reorder buffer entry containing the *Ownder Tag* cannot be found i.e. already consumed, then the *Value* is discarded. The *Owner Tag* is also returned to the virtual tag pool.

There are four important parameters associated with the issue unit:

Reorder Buffer Size Number of actual entries in the reorder buffer.

Virtual Tags Number Amount of virtual tag (explained in Section 4.2.2) to associate with actual RoB entry.

Drizzling Out Threshold When the number of occupied RoB entry exceeds this number, the preemptive drizzling will triggers (Section 4.2.2) to flush out RoB entry into external memory space.

Drizzling In Threshold When the number of occupied RoB entry fall below this number, the flushed RoB entry in the memory (if any) will be brought back from memory.

5.1.4 Execution Units

There are a number of different execution units in the SAFA CPU, namely the integer, float, branch and load/store unit. Although they perform vastly different tasks, the overall structure is similar. So, only the overall general structure is explained in this section, without delving into the details.

Each execution unit has an associated reservation station, which is used to store the instruction dispatched from the issue unit (in the form of reservation station entry). The reservation station is capable of storing a number (Reservation Entries) of instructions. When an entry acquired all its operands, it is passed to the execution unit. If the reservation station has no more free entry at the end of a cycle, issue unit will be signaled to stop instruction dispatching.

Since different type of instruction takes varying amount of time to complete, we model this effect by giving each execution unit a pipeline stages number (Pipeline Stages), which represents the number of stages (each takes 1 cycle) an instruction has to pass through. For example, by giving the integer unit one pipeline stage, result will be available after 1 cycle.

When an execution result is produced, it is broadcast to all other execution units as well as the issue unit.

5.1.5 Frame Registers Unit

This unit can be taken as a specialized execution unit, which in charge of all frame registers (Chapter 3) related instructions. For these instructions, frame registers is the shared resource that has to be accessed in a strictly ordered fashion to avoid any input-output conflict. Because of this, the *reservation station* for the frame registers unit is implemented as an ordered queue of size Frame Instruction Queue Size, where instruction is always taken from the head of queue, regardless of whether there are other available instructions in other part of the queue. When a frame instruction is

taken out from the queue, the unit performs the following steps:

1. Mark related frame register(s) as busy. 2. Execute the instruction. 3. Unmark related frame register(s).

Clearly, the execution must ensures its exclusive hold of related frame registers under this scheme for correct execution. This scheme, although easy to implement, has the disadvantage of stalling ready and conflict free instruction in the queue. Taking this into consideration, the scheme has been modify to the following:

If the head of the queue is not ready, go through the rest of queue from head to tail to look for ready instruction I that satisfy the follows:

- No earlier instruction in the queue needs the same frame register as I.
- The frame register(s) required is free for I.

If such instruction I can be found, it is schedule for execution following the exact same steps explained above. This modified scheme is akin to a *multiple queues* approach that associates each frame register with an individual queue.

5.1.6 Branch Predictor Unit

As discussed (Section 4.3.2), speculative execution in a stack machine is entirely possible. The unit responsible for allowing branch prediction and confirmation in SAFA is the Branch Predictor Unit (BPU). When a branch instruction is decoded, it is passed along to the BPU, which will predict the outcome according to the prediction logic. Several prediction strategies are implemented in the simulator, which includes:

BTFN **B**ackward **T**aken **F**orward **N**ot taken[44]. One of the simplest static branch prediction strategy, where all backward brnaches are predicted as true, while forward branches are predicted as false.

BiModal Keep tracks of the behavior for each branch by a 2-bit counters, which is increased when the branch is taken, and decreased when the branch is not-taken. Prediction is based on the current value of the counter. If the counter is more than 1, the branch is predicted as true. Otherwise, the branch is predicted as false.

GShare One of the most widely used strategy[42]. The global branch pattern is *XOR* with the branch address to produce an index. This index, which captures both the global and local branch information is then used to access a counter (similar to the BiModal strategy). Again, if the counter is larger than 1, the branch is predicted as taken.

As most of our benchmarks programs are small to medium in term of execution time length, the **BiModal** stratgey is used for the results gathered. The BPU is also responsible of setting the appropriate system-wide prediction level.

When a branch instruction is resolved, the result is passed along to the BPU. Depending on the correctness of the prediction, the BPU will send signals to units involved to either confirm or purge a prediction (refer to Section 4.3.2 for details).

5.1.7 Overall System

The execution control unit is a single encompassing entity, serves as the linkage between all the components described. The main functionality of the control unit is as follows:

- Maintain time tick, where each tick corresponds to 1 CPU cycle.
- Maintain the temporary storage between components, e.g. the output of the fetch unit for decode unit. These temporary storages are similar to latches in real CPU.

- Maintain signal raised by all components.
- Maintain statistics like utilization of each components, instructions executed etc.
- Provide debug interface and user interface to the emulator.

The execution of the SAFA CPU follows the time-stepped simulation, where each time step corresponds to 1 cycle. During each time step, there are 2 phases:

1. **Clock Edge Up**: All components take input from previous stages temporary storage, and perform the necessary task.
2. **Clock Edge Down**: All components produce output to be stored in the temporary storage.

The main reasons of splitting into two phases is as follows:

- Since SAFA is a pipelined CPU, the temporary storage may be overwritten if we allow component to get input and produce output in a single phase.
- Good for debugging and execution monitoring, since each phase is well defined.
- This scheme is mimics real CPUs more closely.

5.1.8 Verification of SAFA Simulator

As the accuracy and reliability of the benchmarks hinges on the correctness of the simulator, it is important to give a thorough verification for the simulator. The verfication process is briefly outlined below.

Stage 1. Correctness of Single Instruction Execution

During the testing of single instruction, the simplest execution mode is chosen for clarity and ease of checking. The SAFA Simulator is configured without superscalar and speculative capabilities. The correctness of an instruction execution can be checked by the following criteria:

- Number of cycles: As each of the instruction has well defined execution time, the number of cycles can be checked.
- Machine State: The memory state and internal CPU state (values of various registers) before and after the execution can be used to ascertain the correctness.

Each SAFA instruction is tested in isolation, starting with the least dependent instructions. There are instructions that cannot stand alone, for example, the simple integer add instruction requires some ways to load the operands onto the stack. To isolate the checking, only instructions with no dependency are chosen at first, e.g. the various load instructions. After these instructions are checked, other dependent instructions can now be checked. In this fashion, the instructions are checked according to the depedency chain, which reduce the hassle in pinpointing the source in case of erroneous execution.

Stage 2. Correctness of Non-Superscalar Non-Speculative Program Execution

With the correctness of each instruction ensured, simple programs are then written to test the simulator. These programs contains a loop to stress several interlinked instructions. As the simulator is still under non-superscalar mode, the execution takes a regular pattern for each iteration, which helps to calculate the correct total execution cycles. The program is then set to loop arbitrary number of times and checked against the calculated number of cycles. Besides, the machine state before

and after execution are checked and recorded. After confirming the result of the execution, the machine state generated by the program under this configuration is important for the various test stages later.

Stage 3. Correctness of Non-Superscalar Speculative Program Execution

Speculation is turned on for this set of testings. The same set of programs used in the previous stage are executed under this new machine configuration. As speculative execution may generate irregular execution pattern, calculating the total number of cycles in advance is no longer possible. Instead, the machine state recorded for each of the program from previous stage is used to verify the correctness. The major source of error for speculative execution comes from the incorrect handling of undoing failed speculation. Using the machine state generated by non-speculative execution for comparison, any inconsistency is clearly visible.

Different speculation algorithms are tested separately to ensure they generate similar end state.

Stage 4. Correctness of Supersclar Non-Speculative Program Execution

Similar to the previous testing stage, there is no easy way to calculate the total cycles in advance, hence manual inspection is required to ascertain the validity of the simulator under this mode. Programs written for stage 2 are executed for a single iteration, where the end result for *each cycle* is inpected. As the simulator is capable of showing the changes in various units in the system, it is possible to locate error generated because of supersclar execution.

The programs are then executed with multiple iterations. Manual inspection is no longer feasible for these executions. Thus, only the final machine state is chcked. The machine states recorded in stage two proved to be a great help in weeding out problematic cases.

Stage 5. Correctness of Supersclar Speculative Program Execution

At this stage, the correctness of the simulator is reasonably verified. Hence, programs are simply executed under this mode and checked against the machine states from stage 2.

As with any other program, the correctness of the simulator cannot be verified by exhaustive checking. However, although there are infinetely many SAFA programs, they are composed by a much smaller, finite instruction set. So, it is reasonble to ensure the correctness for each instructions before other testing. Under the more advance modes of execution, the interplay between instructions are complicated and hard to test. As such, checking against verified results (machine states) are used as a main form of testing.

5.2 Software - Assembler and Cross-Assembler

To alleviate the difficulty of writing SAFA raw code directly, and also to minimize programming errors, a simple assembler *safaAs* is written using *yacc* (Yet Another Compilers Compiler) and *lex* (A lexical analyzer generator). The assembler supports a imperative procedural paradigm, with each procedure defined as a self-containing code package. Features of the assembler is summarized as follows:

- Support for symbolic flag for branching.
- Calculation of branch target and reporting error when target is out of range.
- Simple syntax error detection.

The syntax of the SAFA assembly program, as well as the usage of the *safaAs* assembler can be found in Chapter A.

Since we have access to the lowest level of architectural knowledge, there is the temptation to *hand optimize* the benchmark programs to produce favorable

execution result. To avoid this, we decided to take assembly programs compiled by commonly available compilers and convert (cross assembly) into SAFA raw code instead.

The most widely spread, and maybe even the only stack compiler available today, is the *Java Compiler*. Since the Java Virtual Machine is a stack oriented architecture (Section 2.3.5), there are a lot common instructions in the design which allow a mostly one-to-one Java-to-SAFA translation. The mapping is so well defined that a automatic cross-assembler is viable, which is implemented as *JaSa* (Java to SAFA cross-assembler). The limitations of this cross-assembler are:

- Only *static* methods are supported, since supporting object methods requires a fully fleshed out *JVM* emulation on SAFA.
- Have no knowledge of SAFA specific features, like frame registers.
- Array-based objects are represented differently in the two platforms, hence need manual translation.

Using the cross-compilation process, the SAFA program resembles more a real life program resulted from automatic compilation tools, which may contain abundant opportunities for optimization. As a simple illustration, consider the following function:

```
void f()
{
    int i = 0;

    i = 3 * i;
    i = (4 + i) * 5;

}
```

Java compiler would simply compiles the above into:

```
Method void f()
   0 iconst_0          //load constant zero
   1 istore_0          //store to the 0th local variable
   2 iconst_3
   3 iload_0           //load the 0th local variable, i.e. "i"
   4 imul
   5 istore_0          //store 3*i to "i"
   6 iconst_4
   7 iload_0           //load "i" again
   8 iadd
   9 iconst_5
  10 imul
  11 istore_0          //store "i" again
  12 return
```

It is easy to see that the repeated memory movements for the local variable “i” represent a major bottleneck. On stack machine and also GPR machine, such movements can easily be replaced by a single *store* at the end of all computation instead. However, unless otherwise specified, the benchmark programs picked for executions follows the actual java output and are not specifically optimized.

As an aside, a C to SAFA Compiler[41] has also been developed to guage the applicability of SAFA archietecture for general programming languages. That compiler attempts to utilize all special features in SAFA with moderate success.

5.3 Benchmark Programs

To test the various aspects of the SAFA architecture, a set of benchmark programs has been selected. Each program is picked to test a particular or a group of features proposed by the SAFA architecture, which are described in previous chapters. A brief overview and psuedocode for each of the benchmark programs is presented next. The actual SAFA assembly programs are provided in Chapter C for reference.

5.3.1 Sieve of Erathosthense

This is one of the most commonly used prime number finding algorithms. Given a range of positive numbers, we can find all the prime numbers in this range by repeatedly eliminating *multiples of known prime number*. After each round of elimination, the smallest un-eliminated number would be a prime number, which would then be used to cancel out others in the next round.

The pseudo-code of this algorithm is presented below:

```
//An array of N boolean values: True = Eliminated
boolean Range[N]

for i = 1 to N          //Initialize all to un-eliminated
    Range[i] = False

prime = 2
while prime < N
    i = 2;

//eliminate multiples
while i * prime < N
    Range[i*prime] = True
    i = i + 1

//look for the next prime, i.e.
//the smallest un-eliminated number
prime = prime + 1
while prime < N
    if Range[prime] == False
        break
    prime = prime + 1
```

This algorithm is chosen to represent a pure integer operation program. The main operations involved are integer multiplication and division. Refer to Section C.1 for actual SAFA assembly code.

5.3.2 Bubble Sort

This is a popular introductory $O(N^2)$ sorting algorithm which can be visualized easily: compare each number at position X in the array with its neighbor at $X+1$ and swap them if the neighbor is smaller. After one round of comparisons, the largest number will end up at the end of the array. So, the whole array can be sorted by repeating this procedure $N-1$ times. This algorithm can terminate earlier if no swapping occurs during any of the round, which implies that the numbers are already in order. SAFA assembly code for this benchmark is listed in Section C.2.

The pseudo-code of the bubble sort is as follows:

```
//Sort an array A, with N elements.
//Variable:
//   "swap" keep tracks of swapping operation
//   "end" keep track of the last element to compare
end = N - 1
do {
    swap = false
    for (i = 1 to end)
        if (A[i] > A[i+1])
            swap A[i] with A[i+1]
            swap = true

    //last element in place
    end = end + 1
//repeat if there is any swapping
} while (swap == true)
```

For this benchmark programs, we need a randomized initial array for the sorting algorithm to work through. The *Linear Congruential Generator* (LCG) is picked for random number generation because of its simplicity to use and to implement. The LCG basically rely on the formula $N_i = (a * N_{i-1} + c)\ \%\ m$ to generate new number every iteration. The psuedocode is also given for illustration purpose:

```
void LCG (int A[N], int N, int a, int c, int m)
//Fill array A with N random elements.
//Local Variable:
//      "N" use to keep track of the current
//      random number.
N = 1
for (i = 0 to N - 1)
    N = (a * N + c) % m
    A[i] = i;
```

5.3.3 Fibonacci Series

The Fibonacci Series, 0,1,1,2,3,5, ..., is frequently used as an introduction to recursive function. The Nth number in the Fibonacci Series can be calculated easily by summing the $(N-1)^{th}$ and the $(N-1)^{th}$ Fibonacci number. This can be expressed with the recurrence relation as: $Fib(N) = Fib(N-1) + Fib(N+2)$. For a computer program, the formula can be translated almost literally from its mathematical definition, shown in the following pseudo-code:

```
int Fibonacci(int N)
{
    //Fibonacci(0) = 0
    if (N == 0)
        return 0

    //Fibonacci(1) = 1
    if (N == 1)
        return 1



    //Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
    return Fibonacci(N-1) + Fibonacci(N-2)


}
```

The pseudo-code given is a recursive solution which requires exponential number of function calls as N grows. Hence it is a simple yet effective program to test

procedure activation capability. Look for the SAFA assembly code in Section C.4.

5.3.4 Quick Sort

Quick sort is another popular in-place sorting algorithm that gives $O(NlogN)$ in the best case. It employ the divide-and-conquer tactic to repeatedly shrink the problem into smaller but similar sub-problems. The pseudo-code is given as follows:

```
int Partition(int A[], int N)
{
    Pick a pivot P from the array A[]
    Split the A[] into two portions, < P or >=P
    return the position of P, i.e. middle point
}

void QuickSort(int A[], int Start, int End)
{
    if End <= Start
        return;

    //Split into two portion
    Middle = Partition(A, End-Start+1)

    //Apply quick-sort to the two portions
    QuickSort(A,Start,Middle-1)
    QuickSort(A,Middle+1,End)
}
```

The Quick Sort algorithm has linear number of recursive function calls in the worst case, however there are more computation per function calls compared to Fibonacci Series. Hence, it is used to represents typical programs, where there is a mix of function calls and computations. The SAFA assembly code can be found in Section C.5.

5.3.5 Test Score Accumulation: Array and List

To study the usefulness of frame registers in handling high level data structure, a simple benchmark is designed. Consider the scenario in a school where each student is associated with a record, which includes some information like matriculation number, test scores etc. A simple program can then be written to iterate through all the students and calculate the total test score for all students. There are at least two possible ways to group the records: using array or linked list.

To simplify the code, the student record is consider to have two fields: a matriculation number (an integer) and a test score (also an integer). For the array version, the pseudo code is given below:

```
void Accumulate( ){
    //assume to have 20 students
    StudentRecord sa[20]

    //Initialize the student record
    for i = 0 to 19
        sa[i].id = i
        sa[i].testScore = random number (0 to 100)

    //Accumulate the test score
    total = 0;
    for i = 0 to 19
        total = total + sa[i].testScore

}
```

The linked list version is similar, except now the student record contain one more field, a *next* pointer to the subsequent record:

```
public static void main(String args[]){
    //head and temp are pointers
    StudentRecPtr head, temp

    //initialize the head of list to null
    head = null
    for i = 0 to 19
        //Allocate a new student record
        temp = new StudentRecord
        temp.id = i;
        temp.testScore = random number (0 to 100)

        //chain up the new record
        temp.next = head;
        head = temp;

    total = 0;
    temp = head;

    //start from head until null is encountered
    while temp != null
        total = total + temp.testScore;
        temp = temp.next;
}
```

The optimizations using SAFA's unique feature are described in the respective sections.

5.3.6 Linpack - Gaussian Elimination

Linpack [40] is one of the predominantly used benchmarks. For example, the website "Top 500 Supercomputers Site" maintain a list of 500 computers [26] in the world that give highest theoretical peak performance based on this benchmark. It is basically a program to solve a dense system of linear equations via *Gaussian Elimination*. The simplified psuedocode is given as follows:

```
void Linpack(M, b, N)
{
    //Solve a system of linear equations  M * x = b
    //represented by a Matrix M of N x N.

    //First Step, generate a random matrix M
    M = Matgen(N)    //details omitted

    //Second Step, factorize M into
    //upper triangular Matrix, UTM
    UTM = Dgefa(M,N)

    //Third Step, solves UTM * x = b
    //Backward substitution
    Solution = Dgesl(UTM,b,N) //details omitted
}

Matrix Dgefa(M,N)
{
    //C is the column number
    for C = 0 to N-1
        R =  row entry with aboslute maximum in C
        move row R to row C
        for L = C+1 to N-1
            eliminate row L in C using M[R][C]
    return M
}
```

The Linpack is a computationally intensive program that is designed to stress the floating point calculation ability of a computer. For example, the factorization of the matrix (the *Dgefa* function in the pseudo code) involves repeatedly scaling and displacement of vectors of real numbers, which heavily lean on floating point calculation. This benchmark generates $O(N^2)$ floating point operations for solving a matrix of size $N \times N$. Actual SAFA assembly code is listed in Section C.12.

5.4 Hardware Parameters

As laid out in the previous section (Section 5.1), there are several component parameters that can adversely affect the execution. In this section, the parameters used for most of the benchmarks will be listed and explained.

Fetch Unit		
Parameter	Value Used	Explanation
Fetch Size	8	Memory access is aligned. However, 4 bytes is too small especially for multiple decode/issue.

Decode Unit		
Parameter	Value Used	Explanation
Instruction Queue Size	24	Able to store 3 times of the fetch size. So, fetch unit is less likely to be idle.

Issue Unit		
Parameter	Value Used	Explanation
Reorder Buffer Size	32	32 actual entries on the Reorder Buffer
Virtual Tag Number	64	64 Virtual tags to be assigned on the actual entries
Drizzling In Threshold	4	When more than 28 of the 32 entries is unoccupied
Drizzling Out Threshold	28	When more than 28 of the 16 entries is occupied

Integer Execution Unit		
Parameter	Value Used	Explanation
Reservation Entries	4	Able to receive 4 complete/incomplete instructions queuing up for execution
Pipeline Stages	2	All integer instructions take 2 time ticks for executions

Floating Point Execution Unit		
Parameter	Value Used	Explanation
Reservation Entries	4	Able to receive 4 complete/incomplete instructions queuing up for execution
Pipeline Stages	4	All floating point executions take 4 time ticks

Branch Predictor Unit		
Parameter	Value Used	Explanation
Maximum Prediction Level	3	Maximum number of branch prediction in flight. When the number of predictions exceed this number, the fetching and decoding are stalled to wait for the resolution of the previous predictions

5.5 Instruction Type and Execution Time

Each type of instruction in SAFA takes different amount of time ticks to execute. The table below summarize the execution time for each type of instruction, split into separate stages during execution life cycle. These parameters are used for most benchmark results, unless otherwise stated in the respective section.

Instruction Type	Fetch-Decode-Issue	Execute	Total
Integer	3	2	**5**
Float	3	4	**7**
Frame			
-Memory Operation	3	1 + Exec.Time of Memory Load/Store	5
-Others	3	1	**4**
Branch	3	1	**4**
Memory			
-Load	3	1	**4**
-Store	3	1	**4**

5.5.1 Derivation of Instruction Execution Time

Although the execution time for various type of instructions are freely adjustable, an informal derivation is nonetheless helpful in clarifying the parameters chosen. In particular, the execution time of integer versus floating point instruction warrants a careful inspection.

For the integer instruction, the number of cycles needed is quite uniform across most platforms which is in the range of 1 to 3 cycle per integer execution. As such, an average of 2 cycles was used.

On the other hand, the number of cycles needed for the floating point instructions has wildly different values in different archictecture. For example, the

80386 Intel processor used up to 35 cycles for floating point instructions (only the execution stage), while the Alpha 21264 processor uses only 2 cycles[21]. At this stage, there is no good reason to assume that SAFA would be a superpipelined architecture, hence a lower execution cycle is chosen. Among the architectures with low floating point execution cycle, the Alpha 21264 uses 2 cycles as mentioned, the PowerPC 620 uses 4 cycles[24]. Additionally, the ratio between integer and floating point execution cycles approaches 1:2 in these architectures. As the integer execution cycles have been chosen as 2, the floating point execution cycles takes should double this amount, which gives 4 cycles.

5.6 Summary

The information in this chapter serves as foundation to monitor the performance of SAFA architecture. The setup of the SAFA “hardware”, along with the hardware paramters were summarized. The benchmark programs were discussed together with pseudocode to prepare for benchmark result discussion presented in next chapter.

Chapter 6

Benchmark Results

To support the proposal of SAFA architecture, various benchmarks were performed to give a quantitative assessment of the benefits and pitfalls of the design. The results are split into two broad categories, correspond to the two major aspects of the SAFA architecture, namely high level language support and low level instruction parallelism. To ease the discussion, some notational details are first discussed in the next section.

6.1 Benchmark Notation

For each benchmark, a number of execution models are tested. These model are chosen to give contrast and highlight the benefits and/or pitfalls of the SAFA execution model. The table below summarize these models:

Execution Model	**Abbreviation**	**Explanation**
Strict Execution	Strict	Simulate conventional stack machine, which decode and issue 1 instruction per time tick. Also, the similar enforce the restriction that the top of the stack must be ready before the next instruction can be issued.
One decode/issue	1I	Execute using SAFA model. Only allow 1 instruction decode/issue per time tick.
Two decode/issue	2I	Execute using SAFA model. Allow 2 instruction decode/issue per time tick. Superscalar stack execution.
Four decode/issue	4I	Execute using SAFA model. 4 instruction decode/issue per time tick. Superscalar stack execution
One decode/issue & Speculative execution	1IBP	Similar to 1-Issue, except Branch Predictor is turned on to allow speculative execution.
Two decode/issue & Speculative execution	2IBP	Similar to above.
Four decode/issue & Speculative execution	4IBP	Similar to above.

There are a lot of data for each benchmarks, but only a few may shed lights on the particular aspect that we are interested in. To conserve space and avoid cluttering, abbreviations are used. The data reported and abbreviations used are listed below.

Data	**Abbreviation**	**Explanation**
Instruction Count	Inst.Count	Number of instructions decoded. Not necessary executed (e.g. during speculation).
Time Tick	-	Each time tick represent one CPU cycle.
Instruction Per Time Tick	IPT	Represents the number of instructions executed per time tick. i.e. $InstructionCount/TimeTick$
Decode Unit Idle Percentage	Dec.Idle (%)	Percentage of total time tick that decode unit is idle.
Issue Unit Stall Percentage	Iss.Stl (%)	Percentage of total time tick that issue unit is idle (i.e. stall).
Integer Execution Unit Idle (%)	Int Idle	Percentage of total time tick that integer unit is idle. An execution unit is idle if there is no instruction in pipeline and no instruction waiting in reservation station. Only relevant to programs that involve integer executions.
Float Execution Unit Idle (%)	Float Idle	Same as above. For Floating point execution unit.
Load/Store Unit (%)	Mem Idle	Same as above. For load and store instruction (i.e. memory operations).
Prediction Success Rate (%)	Pred.Success	Percentage of correct prediction.

6.2 High Level Language Support

6.2.1 Data Structure Support: Array

As discussed in Chapter 3, conventional stack machine performs poorly when dealing with array access. In SAFA, on the other hand, the proposed frame register provides easier management of the array access which should result in both reduction of memory access and also program footprint (program size). To support the claim, we set up a comparison between two sets of the bubble sort benchmark. One set of benchmark, which is translated as close as possible from a equivalent Java program, uses the conventional array access method. For the other set of benchmark, the bubble sort program is rewritten to utilize frame register for array access. The bubble sort is a suitable choice as it involve intensive array element access and swapping. The exponential growth of the algorithm also guarantee a huge number of operations with a relatively small array.

The respective results are posted in the two table: Table 6.1 for the conventional bubble sort program; Table 6.2 for the safa enhanced version (refer to Section C.3). Since low level execution support is not the topic of this section, we will ignore the significance of superscalar execution.

The first thing that leap to the eyes is the huge different of total instructions executed (*not* the total instruction in the program). The SAFA enhanced version represents a 40 percent reduction (from 75154 to 30770) in total number of instruction needed. Recall that in Chapter 3, we showed that the frame register allow much more concise coding. Although the saving for each element access is miniscule (around 3-5 instructions), the effects pile up quickly in array intensive program, as in this case.

With the reduction in number of instructions, the execution speed (number of time tick needed) naturally shows similar improvement. Comparing the two table, entry by entry, it clearly shows the advantage of frame register support. The result

is summarized in Figure 6.1 To understand the improvement, it is useful to look at the composition of the instruction type for each of the benchmarks. In Figure 6.2, it is shown that 10% of the instructions are direct memory operations (load/store from a stated memory address). Other than that, another 38% is spent on frame related memory operations, for e.g. updating local variables, accessing parameters etc. In total, a whopping 48% of the instructions are spent on memory operations. The memory operations on the SAFA simulator can be considered as very forgiving since cache hit is assumed. However, the huge number of memory instructions still take a toll on the overall performance.

On the other hand, the SAFA version (refer to Figure 6.3) contains negligible direct memory access (less than 1%) since all array accesses are handled through frame registers. Compound by the fact that the index of the array is stored directly in a frame register instead of a variable, the overall memory related operations has been reduced to only 26%. It it also interesting to note that the integer operations are also reduced (from 40% to 16%). This is mainly caused by the elimination of memory address calculation.

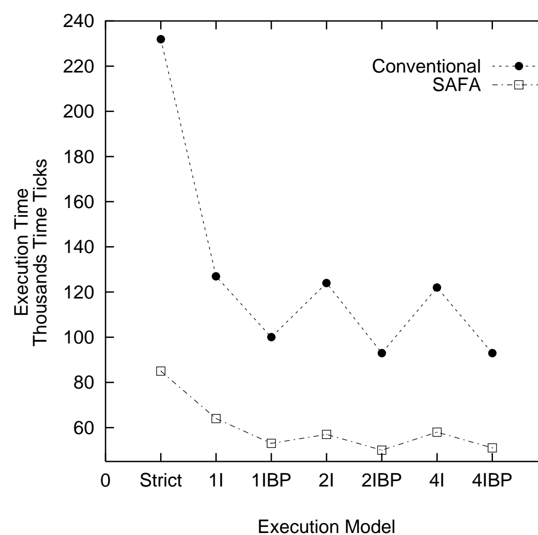


Figure 6.1: Bubble Sort(50 Numbers): Comparison

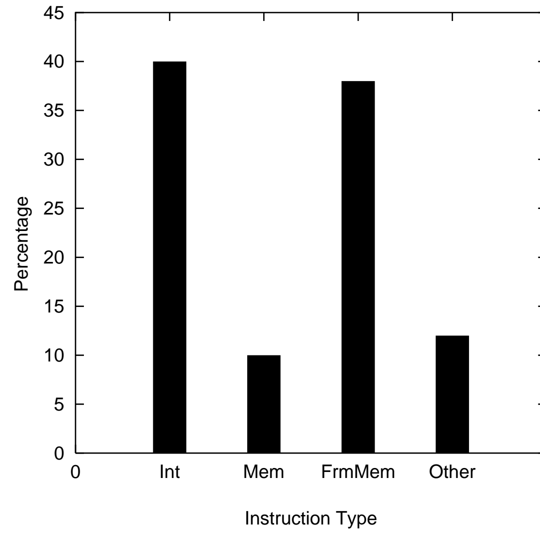


Figure 6.2: Bubble Sort(50 Numbers): Conventional Array Access Instruction Composition

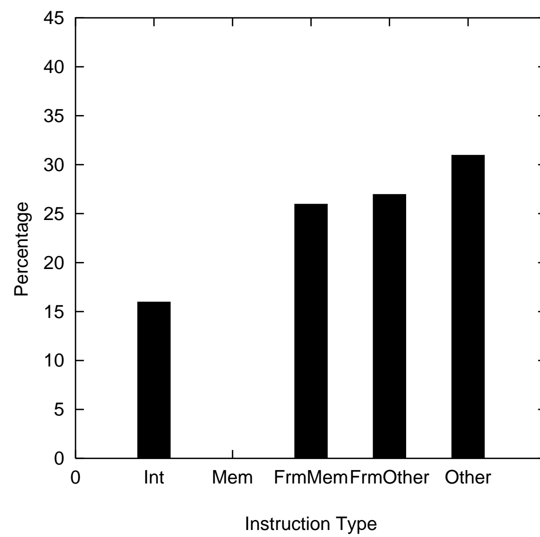


Figure 6.3: Bubble Sort(50 Numbers): Frame Registers Version Instruction Composition

	Inst.Count	Time Tick	IPT	Dec.Idle (%)	Iss.Stl(%)	Int Idle (%)	Mem Idle (%)	Pred.Success
Strict	75154	231597	0.325	63.722	0.000	88.216	75.184	Not Appl.
1-Issue	75154	128620	0.584	34.677	0.005	40.566	40.584	Not Appl.
2-Issue	75154	124445	0.604	60.732	0.420	26.283	30.930	Not Appl.
4-Issue	75154	122567	0.613	76.747	2.134	19.578	31.402	Not Appl.
1-Issue-BP	76689	100386	0.764	0.016	0.006	25.662	27.359	92.002
2-Issue-BP	77177	92521	0.834	11.337	7.001	14.537	11.993	91.100
4-Issue-BP	77283	92570	0.835	16.503	7.736	12.299	12.036	91.100

Table 6.1: Bubble Sort 50 Numbers: Conventional Array Access

	Inst.Count	Time Tick	IPT	Dec.Idle (%)	Iss.Stl(%)	Int Idle (%)	Mem Idle (%)	Pred.Success
Strict	30770	85315	0.361	53.544	0.000	94.134	89.729	Not Appl.
1-Issue	30770	64206	0.479	38.271	0.009	84.414	86.346	Not Appl.
2-Issue	30770	56948	0.540	56.413	0.000	69.493	81.545	Not Appl.
4-Issue	30770	57994	0.531	68.757	0.002	61.212	80.964	Not Appl.
1-Issue-BP	30771	53841	0.572	0.030	0.011	81.414	83.717	92.002
2-Issue-BP	30771	49509	0.622	0.103	0.000	63.758	78.772	92.002
4-Issue-BP	30771	51078	0.602	0.151	0.002	52.796	78.386	92.002

Table 6.2: Bubble Sort 50 Numbers: Using Frame Register

6.2.2 Data Structure Support: Array of Records

With the usefulness of frame register for array support as shown previously, it is natural to expect that SAFA also provides good support for record arrays. For this section, the accumulation of test scores across student records in an array is used as benchmark.

Unlike an array of simple data value, there are several ways to support access in an array of records as laid out in Section 3.2.1. To observe the difference between the solutions, three separate benchmarks are conducted. The first correspond to the conventional stack program (translated from a Java Program). The second and third respectively shows the two separate way to support array of record in SAFA. The second benchmark (corresponds to the version 1 described in Section 3.2.1) represents the whole array as a words (4 bytes) array. Each access to a record component is calculated/translated into an equivalent element access. The third benchmark (corresponds to the version 2 described in Section 3.2.1) make use of an extra frame register that "points" to the currently processed student record. To move to the next record, an offset (the size of a student record) is simply added to the base address of this frame register.

The results of the three benchmarks are summarized in the Table 6.3,Table 6.4 and Table 6.5 respectively.

The benchmarks show the similar trend as in the previous section. The code reduction achieved are 23% for first version and 22% for second version. As noted before, the instruction counts reduction scale proportionally with the number of array access. For this benchmark, the number of access is pretty low compared to bubble sort in the previous section, which results in a smaller reduction rate.

From the view point of execution speed, the SAFA enhanced versions fare better than the pure stack version, as is apparent from the instruction counts. The result is summarized in Figure 6.4. As the results showed, there is only very minor difference between the two ways of representing array of records using frame register.

As the benchmarks do not lean one way or the other, the better or preferred way to represent an array of records would depends on other issues, like ease of compilation etc.

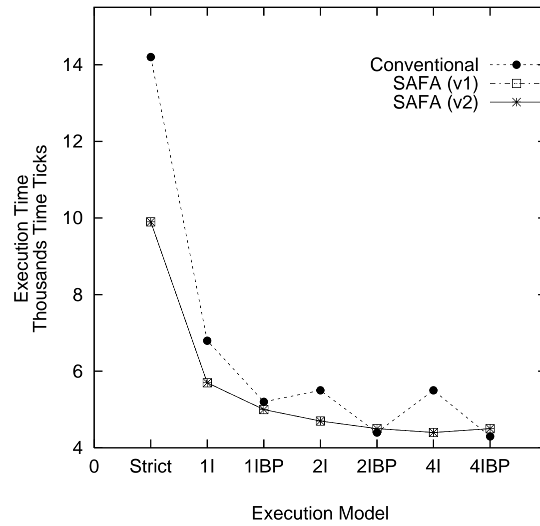


Figure 6.4: Student Array (100 Records): Comparison

	Inst.Count	Time Tick	IPT	Dec.Idle (%)	Iss.Stl(%)	Int Idle (%)	Mem Idle (%)	Pred.Success
Strict	5023	14248	0.353	61.896	0.000	89.458	70.410	Not Appl.
1-Issue	5023	6839	0.734	20.617	0.000	42.886	33.967	Not Appl.
2-Issue	5023	5532	0.908	45.445	0.000	18.492	22.035	Not Appl.
4-Issue	5023	5531	0.908	67.221	0.000	14.807	22.021	Not Appl.
1-Issue-BP	5032	5241	0.960	0.000	0.000	23.526	17.516	99.010
2-Issue-BP	5039	4432	1.137	0.000	0.000	4.941	7.040	99.010
4-Issue-BP	5039	4331	1.163	0.000	0.000	2.655	7.181	99.010

Table 6.3: Student Array (100 records) Benchmark: Conventional Array Access

	Inst.Count	Time Tick	IPT	Dec.Idle (%)	Iss.Stl(%)	Int Idle (%)	Mem Idle (%)	Pred.Success
Strict	3930	9856	0.399	56.006	0.000	92.877	69.399	Not Appl.
1-Issue	3930	5746	0.684	24.539	0.000	61.608	38.827	Not Appl.
2-Issue	3930	4738	0.829	48.902	0.000	28.008	25.791	Not Appl.
4-Issue	3930	4437	0.886	63.624	0.000	20.735	25.197	Not Appl.
1-Issue-BP	3932	5046	0.779	0.000	0.000	58.264	38.189	99.010
2-Issue-BP	3932	4538	0.866	0.000	0.000	27.038	24.680	99.010
4-Issue-BP	3932	4537	0.867	0.000	0.000	20.278	24.598	99.010

Table 6.4: Student Array (100 records): Using Frame Register (version 1)

	Inst.Count	Time Tick	IPT	Dec.Idle (%)	Iss.Stl(%)	Int Idle (%)	Mem Idle (%)	Pred.Success
Strict	3933	9860	0.399	55.994	0.000	92.880	69.412	Not Appl.
1-Issue	3933	5749	0.684	24.526	0.000	59.889	38.859	Not Appl.
2-Issue	3933	4740	0.830	48.882	0.000	28.017	25.802	Not Appl.
4-Issue	3933	4438	0.886	63.610	0.000	20.753	25.192	Not Appl.
1-Issue-BP	3935	5049	0.779	0.000	0.000	56.308	38.225	99.010
2-Issue-BP	3935	4540	0.867	0.000	0.000	24.846	24.692	99.010
4-Issue-BP	3935	4538	0.867	0.000	0.000	20.295	24.592	99.010

Table 6.5: Student Array (100 records): Using Frame Register (version 2)

6.2.3 Data Structures Support: Linked List

Another typical high level data construct would be linked list, which represents an almost exact opposite philosophy as compared to the array. For linked list, the number of records is not fixed at the point of execution or data structure creation. Each record is dynamically allocated, which may be scattered around the available memory space. The records are kept as a whole by *pointer* (memory address) linking one record to the next (refer to Section 3.2.2 for more details).

The nature of the linked list restricts the possible optimization. Each address has to be traversed to reach the next record. Hence, the only possible improvement would be the retainment of useful information in the CPU instead of reloading from memory as in conventional stack machine. For the linked list, the "useful information" would be the memory address of the currently inspected record. In conventional stack machine, stack is the only storage in the CPU core, hence, the memory address needs to be reloaded from memory whenever it is needed. In SAFA, as explained in Section 3.2.2, frame register can be used to retain this information.

Just like the previous benchmarks, a set of three programs are written for validating the idea of frame register in this aspect. These programs simply create a list of student records and then compute the total test score by following the linked list. The first follow closely to the conventional stack program, using a Java program as blueprint. The second and third both utilize frame register to retain the memory address, however, the second program uses the index in a frame register for element access while the third uses direct offset from the frame register base. The tables (Table 6.6 and Table 6.7) summarize the results of the benchmarks respectively.

First of all, the first SAFA version (i.e. program two) actually contains *more* instructions to accomplish the same task. In this particular case, a 6% increase (from 5018 instructions to 5324 instructions) is observed. The extra instructions are mainly dealing with frame register management, for example, setting up the *current frame register* (Section 3.4.1), managing the index etc. As an example, compare the

following two code fragments, taken from program one and two respectively, for the assignment of a pointer.

```
//Implementing the line
//        head = temp;
//Refer to the pseudo code

// Variables:
//      "head" at offset x34 of the execution frame
//      "temp" at offset x38 of the execution frame


    ...
    //Allocate and initialize a new student record
    //address stored in "temp" at x38
    ...
    cfb_wload x38       //load "temp" to stack
    cfb_wstore x34      //store to "head"
    ...
```

```
//Implementing the line
//        head = temp;
//Refer to the pseudo code

// Variables:
//      "head" stored in frame register 4, fr4
//      "temp" stored in frame register 5, fr5
    ...
    //Allocate and initialize a new student record
    //address stored in "temp" i.e. fr5
    ...
    cfset5          //set the current frame to fr5
    cfinfoload      //load the current frame info
    cfset4          //set the current frame to fr4
    cfinfostore     //store the info to fr4
    cfsetown        //set the current frame to the
                    //execution frame, "own" frame
```

The difference in instruction counts is small (2 compared to 5), however, since the above code is in a loop, the small difference accumulates and shows up in

the result.

The third version of the benchmark manages to be slightly more efficient by discarding all index management instructions. This results in a minor decrease in size (less than 1%) compared to the conventional version.

Lets move on to the next aspect of the benchmark: the execution time. This benchmark shows that total instruction count does not necessarily correlate to execution time. It is more important to see what *type* of instructions are more dominant, i.e. the composition of instructions. For example, referring to the two code fragments given, it is *not* definite that first code will run faster. This is because both instructions in the first code fragment are memory operations, while the second code fragment involve only frame register instructions. Although the memory operation latency is not severe in the SAFA model used, the conventional program eventually lose out when super scalar execution is enabled. Multiple memory operations will be serialized by the memory interface (the Load/Store Unit) which forms a performance bottleneck.

Both the SAFA programs executes faster than the conventional program under super scalar execution. The second version of SAFA program, benefiting from its more concise code and lesser memory operations, performs consistently better than the conventional version under all models. The results are summarized in Figure 6.5.

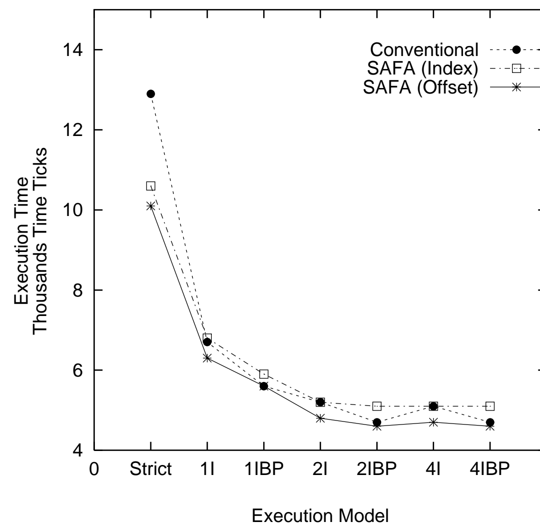


Figure 6.5: Student Linked List (100 Records): Comparison

	Inst.Count	Time Tick	IPT	Dec.Idle (%)	Iss.Stl(%)	Int Idle (%)	Mem Idle (%)	Pred.Success
Strict	5018	12941	0.388	58.087	0.000	93.038	66.649	Not Appl.
1-Issue	5018	6733	0.745	19.442	0.000	62.825	31.457	Not Appl.
2-Issue	5018	5228	0.960	42.330	0.000	34.889	23.259	Not Appl.
4-Issue	5018	5128	0.979	64.684	0.000	27.711	21.763	Not Appl.
1-Issue-BP	5022	5632	0.892	0.000	0.000	55.558	21.555	99.010
2-Issue-BP	5026	4728	1.063	0.000	0.000	21.595	15.102	99.010
4-Issue-BP	5026	4728	1.063	0.000	4.230	13.029	12.986	99.010

Table 6.6: Student Linked List (100 records): Conventional Linked List Traversal

	Inst.Count	Time Tick	IPT	Dec.Idle (%)	Iss.Stl(%)	Int Idle (%)	Mem Idle (%)	Pred.Success
Strict	5324	10644	0.500	46.167	0.000	95.293	76.400	Not Appl.
1-Issue	5324	6837	0.779	16.191	0.000	80.942	60.348	Not Appl.
2-Issue	5324	5231	1.018	40.394	0.000	42.554	32.900	Not Appl.
4-Issue	5324	5131	1.038	64.666	0.000	31.631	25.726	Not Appl.
1-Issue-BP	5327	5938	0.897	0.000	0.000	78.057	56.012	99.010
2-Issue-BP	5327	5131	1.038	0.000	0.000	41.434	27.694	99.010
4-Issue-BP	5327	5131	1.038	11.694	0.000	31.631	21.828	99.010

Table 6.7: Student Linked List (100 records): Using Frame Register and Index

	Inst.Count	Time Tick	IPT	Dec.Idle (%)	Iss.Stl(%)	Int Idle (%)	Mem Idle (%)	Pred.Success
Strict	4824	10144	0.476	48.442	0.000	95.061	71.293	Not Appl.
1-Issue	4824	6337	0.761	17.469	0.000	79.438	52.485	Not Appl.
2-Issue	4824	4831	0.999	39.598	0.000	41.937	31.484	Not Appl.
4-Issue	4824	4731	1.020	63.792	0.000	30.078	27.901	Not Appl.
1-Issue-BP	4827	5636	0.856	0.000	0.000	76.881	46.576	99.010
2-Issue-BP	4827	4631	1.042	0.000	0.000	41.589	19.845	99.010
4-Issue-BP	4827	4631	1.042	4.319	2.159	30.728	17.577	99.010

Table 6.8: Student Linked List (100 records): Using Frame Register and Offset

6.3 Low Level Instruction Support

In this section, we turn to inspect another aspect of the SAFA architecture: low level instruction execution support. For each of the benchmark result, we will concentrate on the improvement of SAFA architecture over conventional stack machines. The improvement is measured by *Speedup*, calculated as:

$$Speedup = \frac{ExecutionTime_{conventional}}{ExecutionTime_{SAFA}}$$

At the same time, as the result will most likely deviate from the expected (ideal) case, we will look at the reasons for the discrepancies and possible remedies.

The other important yardstick of CPU benchmark is the instruction per time tick (IPT). IPT basically measures the number of executed instructions during one time tick. For a single issue pipelined CPU, IPT is bounded by 1.0, since there is only a single instruction reaching the execution stage in any CPU cycle. Also, a number of problems can cause the pipeline to stall, e.g. branching, dependencies etc, further reducing the actual IPT.

Given a multiple issues (superscalar) CPU, we would expect the IPT to rise above 1.0, bounded by the maximum number of instructions issued. Again, this naive view is marred by the same problems mentioned above. Besides, the number of execution units plays a much more important role for super scalar CPU. Consider the scenario where four integer instructions are decoded and issued successfully, we can only expect all of them to be completed together *if* there are at least *four* available integer execution units.

However, the number of execution units is normally smaller, for example, the Alpha 620 consists of 3 integer units (2 for simple instruction, 1 for complex instruction). So, these issued instructions are forced to be executed serially, one after another by the limited number of execution units. In this scenario, the execution unit serializes the potentially parallel executions and reduce the IPT. Conversely, a stream of instructions of different types can perform much better, since the execution

can be performed in parallel by separate units. Since there is no easy way to quantify the dynamic mixture of instruction type during execution, we utilize the composition (spectrum of instructions type) of a program to give some inkling of the possible parallel execution opportunity.

Hence, the IPT for a superscalar machine is closely linked to the number of execution unit and also the composition of the program executed. We will inspect and comment on the IPT achieved as we go through the benchmark results.

6.4 Various Benchmarks: Single Execution Unit

In this section, we deploy a standard SAFA model, which consists of only four execution units, one each for integer, float, branch and memory instruction. For the reason described previously, the IPT rarely rise above 1.0 because of the serialization effect. Only benchmark with more spread out instruction composition can take advantage of multiple *different* execution units, and perform much better.

As mentioned before, all the benchmark results are accompanied by instruction composition graph. To ease discussions, the instruction types discussed are listed as follows:

Integer Integer instructions, like addition, subtraction etc

Float Floating point instructions.

Memory Direct memory access, such as loading, storing to memory space.

Frame-Memory Memory access through frame registers. For example, loading an element indexed by a frame register. To have a complete picture of memory access instruction, these instructions have to be considered along with the *Memory* instructions described above.

Frame-Other Non-memory related instructions that base on frame register, such as changing index, setting base address of a frame register etc. The previous

category (*Frame-Memory*) combined with this category form the total frame register based instruction.

Other Other type of instructions, such as stack manipulation instructions e.g. popping, swapping entries, and also branching instructions.

6.4.1 Fibonacci Series

From the algorithm description in the previous section, the *Fibonacci* benchmark obviously leans heavily on branching and function call, while light on calculation. As shown in the composition graph (Figure 6.7), integer calculation takes only a meager 9%, memory related operations, both directly and through frame instructions, take a big chunk at 29% (4% + 25%). Since function calls involve stack frame allocation and initialization, it is not surprising that the frame instructions (other than memory-related operations) dominates the graph at 34%.

Also, it is worth pointing out that the frame related instructions totally dwarfed other types of instructions at 59% (25% + 34%). This is a trend shared by all benchmark results, which arise simply because of the reliance of conventional stack compiler (Java compiler more specifically) relies heavily on the *current* stack frame (stack frame of the current executing method/procedure). The parameters and local variables of the current procedure are stored in the current stack frame, which see heavy usage. Since frame registers usage are serialized (refer to Section 5.1), this dominant usage of frame register instructions introduces a bottleneck which restrict the possible speed up.

The results are summarized in Table 6.9. As can be seen, the *strict* execution model which simulates conventional stack machine, gives appalling result. For the *strict* execution model, the stack top must be ready before subsequent instructions can be issued (as mentioned in Section 6.1). This unnecessarily stalls the execution, as shown by the poor execution time as well as low IPT (0.489). Just by removing this restriction such that consumer instruction can be issued and wait

for its operands on reservation stations, the resultant execution model (the *1-Issue* model) already shows a 1.39 speed up.

The *2-Issue* and *4-Issue* models shows the potential of superscalar execution. By relaxing the dependencies and adapting conventional superscalar techniques, the speed up gained are 1.60 and 1.63 respectively. Although the speed up gained by issuing 2 instructions up from 1 instruction is quite satisfactory, the jump to issuing 4 instructions is less so (only a 1.02 increase from 2-Issue to 4-Issue). One of the reason is that for all these execution models, execution stalls as soon as encountering a branch instruction. Since branch instructions (including conditional/unconditional branches and procedure entry/exit) occurs frequently in this program, the machine is forced to stall most of the time. As shown in Table 6.9, the decode unit is stalled *at least* half of the time and reaches a high 64% for *4-Issue* execution model.

One way to recoup some of the wasted time ticks is to enabled speculative execution. For all the execution models *X-Issue-BP*, branch prediction and speculation (Section 4.3.2) is turned on. Since speculative execution in SAFA is restricted to normal branches, i.e. no inter-procedure speculation, the expected benefit is minor. With the help of the *BiModal* adaptive branch prediction, SAFA predicted correctly for around 55% of the branches. The resultant speed up gained are 1.49, 1.75 and 1.77 respectively for the three speculative execution models.

The speed up of various execution models are summarized graphically in Figure 6.6. To round off the discussion, we should point out that this benchmark represent one of the worst possible scenario in low level execution. Short function body and intensive recursive function calls gives scant opportunity for instruction level parallelism. Even so, the result still shows some benefit of the ILP scheme.

	Inst.Count	Time Tick	IPT	Dec.Idle (%)	Iss.Stl(%)	Int Idle (%)	Mem Idle (%)	Pred.Success
Strict	8181	16720	0.489	44.061	0.000	95.449	72.895	Not Appl.
1-Issue	8181	12049	0.679	22.375	4.407	91.037	62.387	Not Appl.
2-Issue	8181	10457	0.782	48.829	0.000	86.621	46.505	Not Appl.
4-Issue	8181	10280	0.796	64.835	0.000	79.844	42.140	Not Appl.
1-Issue-BP	8339	11212	0.744	12.629	4.736	90.367	59.276	61.250
2-Issue-BP	8585	9546	0.899	35.229	0.000	83.836	39.535	55.367
4-Issue-BP	8648	9423	0.918	51.289	0.000	77.268	34.416	55.367

Table 6.9: Fibonacci(10) = 55. Total Recursive Calls = 177

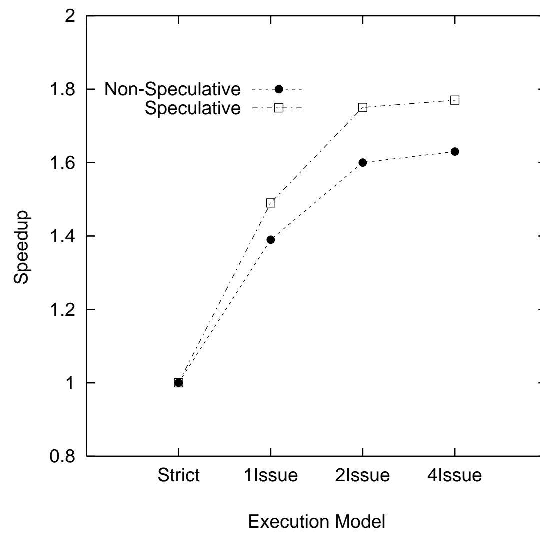


Figure 6.6: Fibonacci Series. Fib(10) : Speed Up

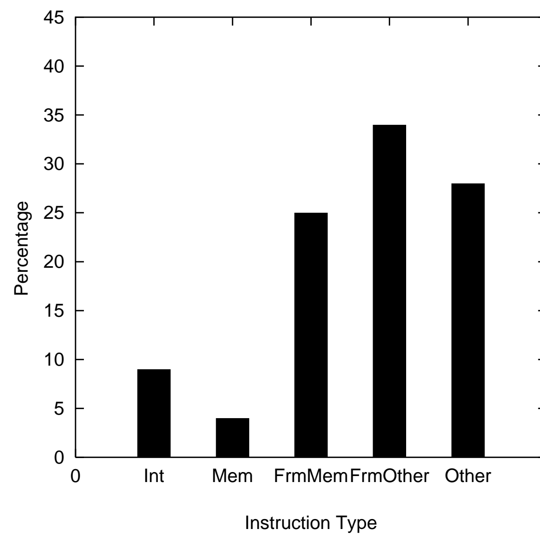


Figure 6.7: Fibonacci Series: Composition

6.4.2 Sieve of Erathosthense

For this benchmark, there are far fewer function calls compared to the Fibonacci benchmark. The majority of the execution time is spent on manipulating and updating local variables, e.g. the *prime*, the i variables in the pseudo-code shown in Section 5.3. Again, the reliance on current stack frame shows up as the heavy usage (47%) for the *Frame-Mem* category shown in Figure 6.9. There are more integer computation like calculating the multiples for prime numbers, increment of values etc, which contributes a substantial 22% to overall instruction counts. Branch instructions to handle the looping and condition checking contribute another 22%, with the array manipulation shown up as direct memory access taking the rest 7%.

Execution for the plain integer instructions can be considered as prime candidate for parallelization. This benchmark, with substantial amount of integer computation should yields better speed up. As can be seen in the graph (Figure 6.8), this simple analysis is pretty close to the mark. *2-Issue* and *4-Issue* both give speed up of 1.86. However, the speed up fails to gain any advantage from the higher issuing rate (from two to four), which shows a possible bottleneck.

One of the possible reason is the frequent occurrence of branching instructions, which detracts the CPU from good performance. Table 6.10 supports this reasoning, as shown by the high idling percentage of the decode unit (more than 59% during superscalar execution).

This bottleneck can be easily removed by enabling speculative execution. With good prediction success, the machine manage to perform much better, giving speed up of 2.54 for *1-Issue-BP* model and 2.77 for both *2-Issue-BP* and *4-Issue-BP* models. As most of the instructions can be speculated, the speculative execution consistently outperform the non-speculative counterpart. It is interesting to note that even the *1-Issue-BP* model can outperform the *4-Issue* model. The reason for the good performance under speculative execution comes mainly from the good success rate and also the way that a branch is compiled in Java. For example, below

is an excerpt from the assembly code (Section C.1):

```
forLoop:
    cfb_wload x2c   //load local variable
    cfb_wload x28   //load local variable
    ige
    iftrue forLoopEnd   //Branch Instruction
    cfb_wload x24 //load from current frame
    cfb_wload x2c //load from current frame
    iadd
    ibload 1
    bstore
    .....
forLoopEnd:
    .....
```

As the branch *iftrue* is based on two values that reside in memory, the resolution has to wait for the memory loads. This time window represent a good opportunity for speculative execution. Additionally, after the branch instruction "*iftrue*", there are several instructions that can be executed speculatively up to the instruction "*bstore*". With a good time window and plenty instructions, this allow the execution to continue while the branch is being resolved. Since the local variables are stored in the stack frame in Java model, the above code snippet occurs very frequently in Java bytecode.

However, good speed up does not detract from the fact the speed up again tapers out at *2-Issue-BP*. This strongly suggests another factor in play which limits the possible benefits. As mentioned earlier, the composition graph (Figure 6.9) shows a very heavy usage memory traffic via frame register instructions. Further, these instructions are serialized at the frame register unit which limits the possible gain from multiple issues. This major problem is rectified in Section 6.6.

In a nutshell, this benchmark shows that: Firstly, more computation allows better parallelization, and secondly, the time window and the amount of eligible instructions for speculation plays a major part in execution speed.

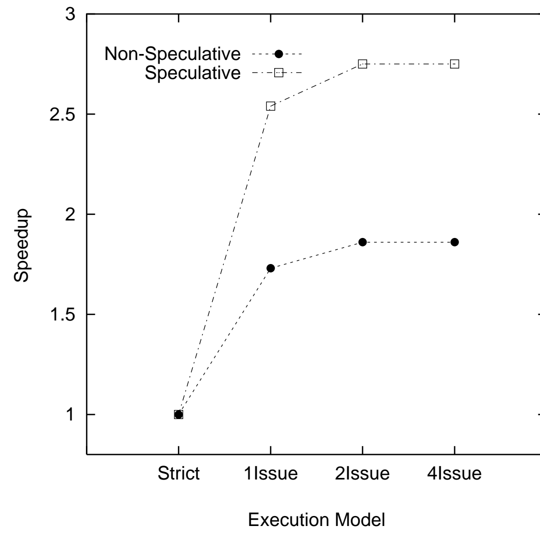


Figure 6.8: Sieve of Erathosthense (100 Numbers) : Speed Up

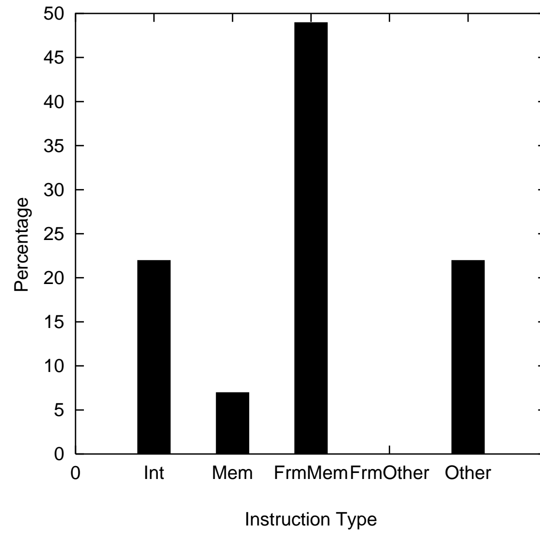


Figure 6.9: Sieve of Erathosthense: Composition

	Inst.Count	Time Tick	IPT	Dec.Idle (%)	Iss.Stl(%)	Int Idle (%)	Mem Idle (%)	Pred.Success
Strict	4943	15819	0.312	63.127	0.000	93.021	68.620	Not Appl.
1-Issue	4943	9150	0.540	36.251	0.033	60.634	41.191	Not Appl.
2-Issue	4943	8505	0.581	59.024	0.000	36.966	36.343	Not Appl.
4-Issue	4943	8504	0.581	70.967	0.000	31.456	36.324	Not Appl.
1-Issue-BP	5076	6233	0.814	0.128	0.048	46.286	22.798	90.121
2-Issue-BP	5095	5738	0.888	0.627	0.000	28.372	28.512	90.121
4-Issue-BP	5095	5738	0.888	0.575	0.000	18.404	28.494	90.121

Table 6.10: Sieve of Erathosthense: 100 Numbers

6.4.3 Bubble Sort

The bubble sort benchmark is previously used to show support for high level data structure. In this section, we are using the version with conventional array access. Please refer to Table 6.1 for detailed benchmark results and Figure 6.2 for instruction composition graph.

As noted before (Section 6.3), the instruction composition for bubble sort is pretty well spread, in particular there are 40% integer instructions and 48% memory related instructions. The huge amount of pure integer computation is largely responsible for the good 1.86 and 1.89 speed up under the superscalar execution models: *2-Issue* and *4-Issue* respectively as shown in Figure 6.10 .

With the same argument as in the previous benchmark "Sieve of Erathosthense", speculative execution again give additional speed up. Aided by more computational instructions and also higher prediction success rate, the gain in execution speed is pretty substantial. Both *2-Issue-BP* and *4-Issue-BP* gives speed up of 2.50. The *slightly longer* execution time of the *4-Issue-BP* model is an interesting artifact created by the larger issue rate and speculative execution. As the available parallelism hits a ceiling at two issuing model, faster issuing rate only increases the number of instructions executed when the execution is on the *wrong* predicted path. Again, it seems that the speed up encounters a ceiling at two issuing execution models. This ceiling of speed up can be pushed much further up as shown later in Section 6.6.

This benchmark demonstrates the possible benefits for SAFA architecture given unoptimized conventional stack program. As discussed in Section 6.2, it is possible to get even better performance by utilizing specialized SAFA features.

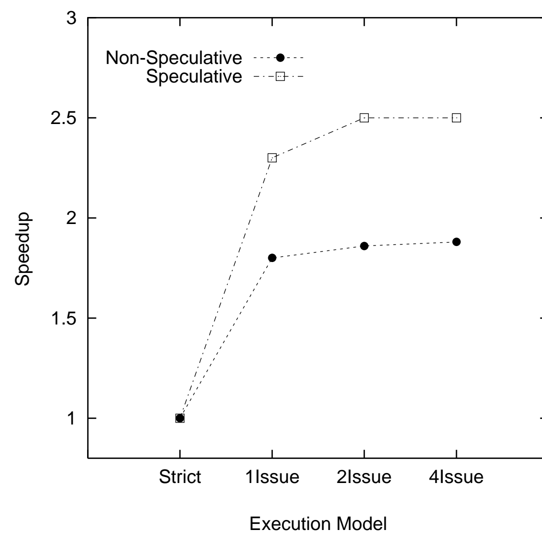


Figure 6.10: Bubble Sort (50 Numbers) : Speed Up

6.4.4 Quick Sort

The quick sort benchmark can be taken as a counterpoint of the Fibonacci benchmark discussed previously. Both of the benchmarks involve substantial function calls. However, the recursive function in quick sort contains a larger body of computation compared to those in Fibonacci benchmark. This contrast can be seen clearly in Figure 6.12, with the quick sort benchmark containing more integer instructions but having equally substantial frame register based instructions as compared to the Fibonacci benchmark (Figure 6.7).

The Fibonacci benchmark gives pretty poor performance especially for speculative execution. The quick sort benchmark, on the other hand, shows good performance under SAFA architecture. As shown in Figure 6.11, superscalar execution under the models *2-Issue* and *4-Issue* give speed up of 1.97 and 1.99 respectively.

Even with the mediocre prediction success rate (around 50% for superscalar execution), the speculative execution nevertheless squeeze extra performance out of the architecture. The speed up gained are 2.34 and 2.33 for execution models *2-Issue-BP* and *4-Issue-BP* respectively. The slightly longer execution time for *4-Issue-BP* model reprise the effect discussed in the Bubble Sort benchmark. The result also restates the trend that speculative execution can give better return compared to plain superscalar execution, with the execution model *1-Issue-BP* outperforming the *4-Issue* again.

This benchmark serves as a confirmation that as long as there are substantial body of computation, the parallelism extracted would overcome the serialization forced by other instructions (like function calls, branching etc).

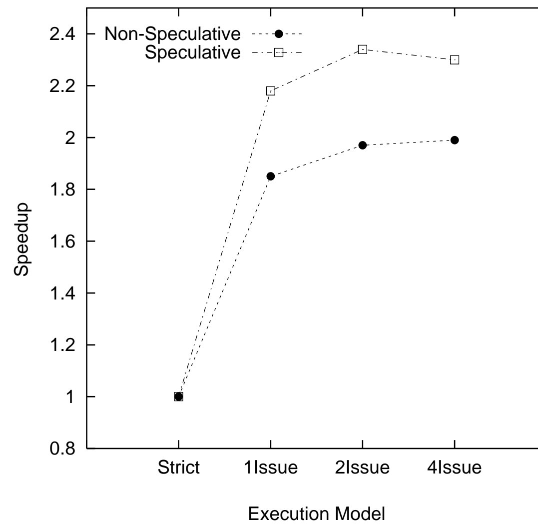


Figure 6.11: Quick Sort (50 Numbers) : Speed Up

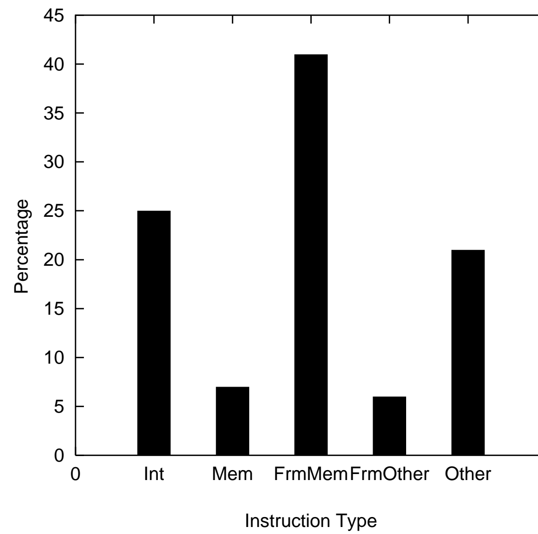


Figure 6.12: Quick Sort: Composition

	Inst.Count	Time Tick	IPT	Dec.Idle (%)	Iss.Stl(%)	Int Idle (%)	Mem Idle (%)	Pred.Success
Strict	11347	32860	0.345	61.519	0.000	91.272	70.350	Not Appl.
1-Issue	11347	17799	0.638	28.957	0.792	54.986	42.750	Not Appl.
2-Issue	11347	16691	0.680	57.013	1.917	38.823	31.778	Not Appl.
4-Issue	11347	16536	0.686	71.426	0.968	34.349	31.779	Not Appl.
1-Issue-BP	12207	15059	0.811	2.497	0.936	45.886	31.582	59.028
2-Issue-BP	13227	14035	0.942	15.155	3.684	30.737	18.725	49.190
4-Issue-BP	13309	14101	0.944	26.878	4.064	27.792	18.793	50.361

Table 6.11: Quick Sort: 50 Numbers. Total Recursive Calls = 43

6.4.5 Linpack: Gaussian Elimination

As the Linpack benchmark is the only floating point benchmark and also one of the more complicated benchmark compared to others, several set of tests were performed which solves simultaneous linear equation matrix of different size. The detailed result of the three tests, which solves the matrix of size 5, 10 and 15 are shown in Table 6.12, Table 6.13 and Table 6.14 respectively.

The composition graph (Figure 6.14) shows a good mix of instructions. Pure computation instructions (integer and floating point instructions combined) constitute 28%. Memory operations at 46% (9% + 37%) and other instructions take the rest 20%.

Figure 6.13 shows a steady increase for speed up as the matrix size increases. The smallest matrix (5×5) has the worst performance, which is caused by the limited amount of computations. The more intensive calculation in larger matrix proves to be a good source for parallel executions. The 15×15 achieve a good speed up 2.25 under the *4-Issue* model. In a nutshell, superscalar execution models *2-Issue* and *4-Issue* give speed up of 2.17 (average) and 2.18 (average) across the three tests.

For speculative execution models, the prediction success rate is fairly good (above 80% for all cases) and increase steadily as matrix size increases. This is because quite a number of functions in Linpack loop through all values in a row (for example, scaling the row by a constant etc), which tunes the *BiModal* predictor to give more accurate guesses.

With favorable prediction success and substantial computation, the speculative execution models *2-Issue-BP* and *4-Issue-BP* outperform their counterpart and give speed up of 2.57 (average) and 2.58 (average). In particular, the 15×15 matrix reaches a respectable speed up of 2.67. Similar to several previous benchmarks, all *1-Issue-BP* models outperforms the *4-Issue* model which has much higher issue rate.

Linpack benchmark is one of the more popular benchmark around. As the favorable results showed, SAFA architecture is capable of removing the constraints that restricts execution in conventional stack machine.

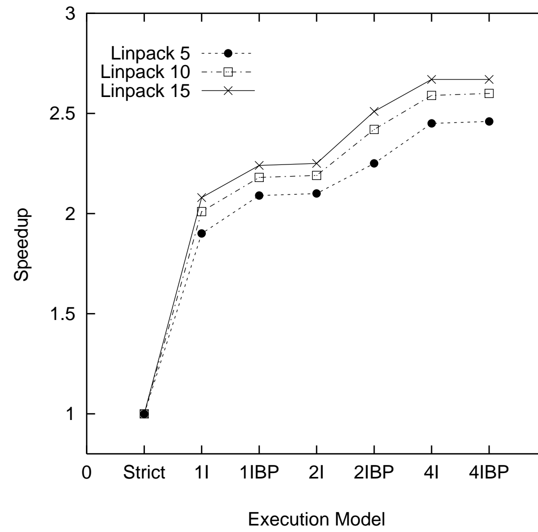


Figure 6.13: Linpack Benchmarks : Speed Up

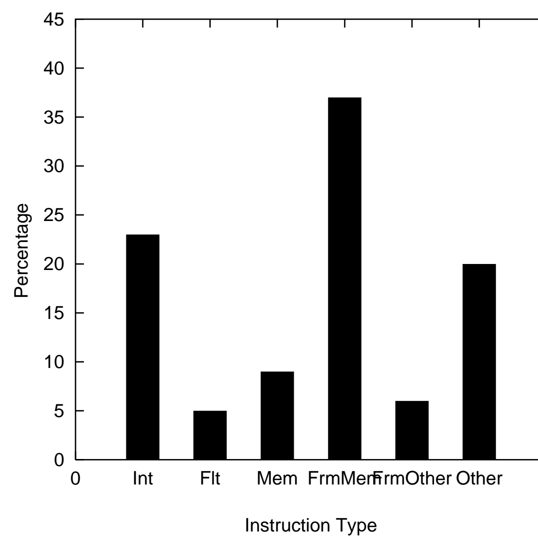


Figure 6.14: Linpack Benchmarks: Composition

	Inst.Count	Time Tick	IPT	Dec.Idle (%)	Iss.Stl(%)	Int Idle (%)	Float Idle (%)	Mem Idle (%)	Pred.Success
Strict	8020	22027	0.364	60.217	0.000	92.409	96.187	72.093	Not Appl.
1-Issue	8020	11599	0.691	24.450	1.138	59.729	84.266	40.391	Not Appl.
2-Issue	8020	10550	0.760	54.190	1.488	41.820	71.460	29.374	Not Appl.
4-Issue	8020	10490	0.765	71.678	2.555	37.807	65.672	29.066	Not Appl.
1-Issue-BP	8388	9806	0.855	3.692	1.448	48.226	79.737	28.880	80.902
2-Issue-BP	8651	8992	0.962	25.000	6.261	33.641	64.858	18.895	80.000
4-Issue-BP	8716	8962	0.973	35.729	7.097	30.852	57.744	18.623	80.000

Table 6.12: Linpack(5): Solve 5 x 5 floating point matrix using Gaussian Elimination.

	Inst.Count	Time Tick	IPT	Dec.Idle (%)	Iss.Stl(%)	Int Idle (%)	Float Idle (%)	Mem Idle (%)	Pred.Success
Strict	33117	94471	0.351	61.921	0.000	92.122	95.459	72.452	Not Appl.
1-Issue	33117	47055	0.704	23.549	0.886	55.019	79.917	37.199	Not Appl.
2-Issue	33117	43338	0.764	54.340	0.916	35.768	64.048	25.170	Not Appl.
4-Issue	33117	43106	0.768	72.136	1.735	32.021	56.570	24.922	Not Appl.
1-Issue-BP	34227	39107	0.875	3.150	1.373	38.648	71.412	23.564	88.616
2-Issue-BP	34790	36422	0.955	28.425	8.561	26.407	54.970	14.714	88.114
4-Issue-BP	35000	36332	0.963	39.076	8.590	24.279	45.990	14.585	88.114

Table 6.13: Linpack(10). Solve 10 x 10 floating point matrix using Gaussian Elimination.

	Inst.Count	Time Tick	IPT	Dec.Idle (%)	Iss.Stl(%)	Int Idle (%)	Float Idle (%)	Mem Idle (%)	Pred.Success
Strict	82409	240177	0.343	62.843	0.000	91.972	95.086	72.601	Not Appl.
1-Issue	82409	115643	0.713	22.830	0.737	52.098	77.245	35.082	Not Appl.
2-Issue	82409	107410	0.767	54.342	0.655	32.022	59.617	22.397	Not Appl.
4-Issue	82409	106915	0.771	72.331	1.321	28.440	51.114	22.189	Not Appl.
1-Issue-BP	84720	95642	0.886	2.951	1.466	32.602	65.192	19.973	90.647
2-Issue-BP	85572	90110	0.950	30.807	10.165	22.071	49.276	12.305	90.281
4-Issue-BP	86002	89936	0.956	41.590	9.659	20.332	39.150	12.198	90.281

Table 6.14: Linpack(15). Solve 15 x 15 floating point matrix using Gaussian Elimination.

6.5 Various Benchmarks: Multiple Execution Units

In this section, a brief exploration into multiple execution units under SAFA architecture is conducted. In contrast to the benchmarks in Section 6.4, multiple execution units *of the same type* are deployed. The result serves as a starting point to consider whether further performance can be gained under SAFA architecture.

6.5.1 Bubble Sort

This benchmark is picked because of its more widely spread instruction spectrum (Figure 6.2). The intensive integer computation should allow the Bubble Sort benchmark to make better use of the extra execution unit.

The results in Table 6.15 is gathered with a SAFA model that contains **two** integer execution units, with all other parameters remain the same. The table also contain a brief statistic from the single integer execution unit model discussed in Section 6.4 for comparison.

Intuitively, multiple execution units in general can only be exploited by *continuous* or at least *densely grouped* stream of instructions of the same type. Furthermore, since the execution units in SAFA architecture are pipelined, a new instruction (if ready) can be ushered into the execution stages every time tick. This further restrict the condition where multiple execution unit can be useful: firstly, that instructions of the same type must be issued within the same time cycle; and secondly, they must be ready to be executed.

Besides, the reservation station attached to a execution unit can serves as extra temporary storage for issued instruction. This would allow more instruction issue in the scenario where instruction issuing is stalled by inadequate reservation station entries.

With the reasoning above, it is unlikely that further speed up can be ob-

served when the issue count is low. Only when the issue count is high enough, the chance of satisfying the above conditions will be high enough for performance gain.

As the results in Table 6.15 shows, the above reasoning agrees with the outcome. The performance difference for any non-speculative execution models is negligible. Minor Performance gain (1.03) is observed for the *2-Issue-BP* and *4-Issue-BP* as compared to their single integer unit counterparts. The low speed up is consistent with the reasons mentioned: the conditions to fully utilize the multiple integer execution units only occurs infrequently, even when multiple instructions are issued every cycle.

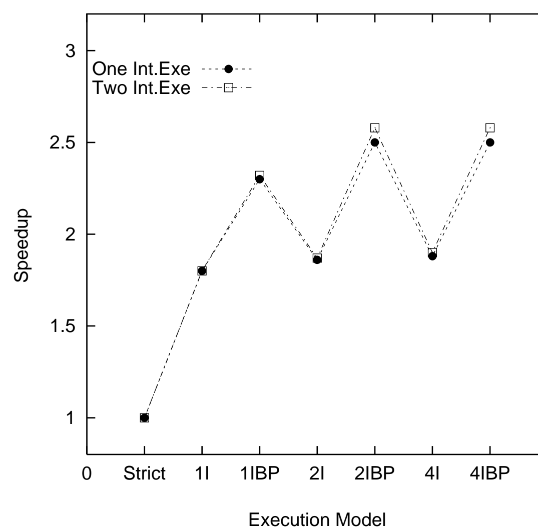


Figure 6.15: Bubble Sort (50 Numbers) : Multiple Execution Units - Speed Up Comparison

	Single Int.Exe			Two Int.Exe		
	Inst.Count	Time Tick	IPT	Inst.Count	Time Tick	IPT
Strict	75154	231597	0.325	75154	231597	0.325
1-Issue	75154	128620	0.584	75154	128620	0.584
2-Issue	75154	124445	0.604	75154	123922	0.606
4-Issue	75154	122567	0.613	75154	122044	0.616
1-Issue-BP	76689	100386	0.764	76689	99864	0.768
2-Issue-BP	77177	92521	0.834	77189	89794	0.860
4-Issue-BP	77283	92570	0.835	77283	89792	0.861

Table 6.15: Bubble Sort (50 Numbers): Multiple Execution Units - Comparison

6.5.2 Linpack Benchmark

The Linpack Benchmark, being the computational intensive and only floating point benchmark program serves as another suitable study. For this particular experiment, a matrix of 15×15 is used. As shown previously (Figure 6.14), this benchmark actually contains quite intensive floating point *and* integer instructions. To study the interaction of the multiple execution units, we set up three different execution models consists of *two integer execution units*, *two floating point execution units* and *two of integer and floating point execution units.* Refer to Table 6.16 for the trimmed results along with the single execution unit counterpart for comparison.

As noted in the previous test, multiple execution units is useless without sustained supply of instructions. For the Linpack benchmark, multiple integer units increase the performance only in *2-Issue*, *4-Issue* and *4-Issue-BP*. As with the Sieve of Erathothense and Bubble Sort benchmarks, the increase in integer execution units actually slightly degrades the performance as shown in the execution models *1-Issue-BP* and *2-Issue-BP*.

Even in the cases where execution gain is observed, the less than 1% (average) speedup is not encouraging given the high concentration of integer instructions (25%). It is perhaps not surprising that multiple floating point execution units *do not* increase the performance at all. With much lesser concentration (only 5%), it is not realistic to expect the instructions shows up in high enough frequency during one time cycle to make good use of the multiple execution units.

Since the floating point instructions in this benchmark could not utilize the multiple units fully, the SAFA model with two of integer and floating execution units can only perform on par with the model without the multiple floating point units. The diagram (Figure 6.16) shows only the speed up difference between the single integer unit and the two integer units.

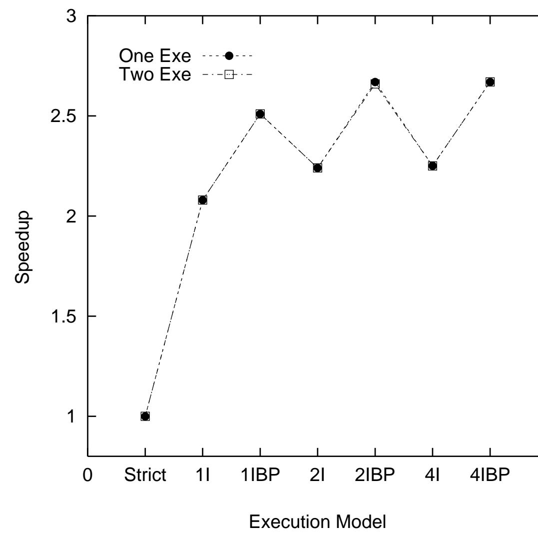


Figure 6.16: Linpack Benchmark (15 x 15): Multiple Execution Units - Speed Up Comparison

	Single Int.Exe		**Two Int.Exe**		**Two Flt.Exe**		**Two Int/Flt**	
	Time Tick	IPT	Time Tick	IPT	Time Tick	IPT	Time Tick	IPT
Strict	240177	0.343	240177	0.343	240177	0.343	240177	0.343
1-Issue	115643	0.713	115643	0.713	115643	0.713	115643	0.713
2-Issue	107410	0.767	107277	0.768	107410	0.767	107277	0.768
4-Issue	106915	0.771	106782	0.772	106915	0.771	106782	0.772
1-Issue-BP	95642	0.886	95657	0.886	95642	0.886	95657	0.886
2-Issue-BP	90110	0.950	90229	0.953	90110	0.950	90229	0.953
4-Issue-BP	89936	0.956	89803	0.958	89936	0.956	89803	0.958

Table 6.16: Linpack Benchmark (15 x 15): Multiple Execution Units - Comparison

6.6 Various Benchmarks: Local Data Access Optimization

As the previous benchmarks show, the over reliance on own stack frame for parameters and variables access hinders the full potential of superscalar execution in SAFA. Most of the benchmarks hit the performance ceiling as soon as *2-Issue-BP* execution models. In Section 4.4, we proposed a mechanism to minimize these frequent memory operations by turning them into stack-to-stack operations. In this section, several of the benchmark programs are re-written to utilize this feature. The results, shown later in respective sections, are very encouraging.

Section 4.4 noted that the possible overhead of using local data map can be reduced by modifying the construction method of stack frame. The usual method for stack frame building stores all the parameters into the stack frame before transferring the control to the activated procedure. The activated procedure is then required to setup the local data map by moving these items from memory. Obviously, this introduces additional memory movements. A better construction method (discussed in the same section) requires the caller to leave the parameters on the operand stack without storing them in stack frame of the callee. The callee procedure can simply store the parameters into the local data map when activated. The second method obviously cut out the redundant memory movements. To monitor the significance of this reduction, each of the benchmark presented consists of two set of programs, each utilizing one of the construction methods mentioned. The results marked with "**Stack Frame**" utilize the stack frame storage method, while those marked with "**Operand Stack**" leave the parameters on stack. The results reported in the previous are included in the "**Normal**" column for comparison.

To show the benefit of this optimization over the traditional stack machine, the speedup results are computed by comparing the execution time of the optimized binary with the execution time of the *unmodified* binary by the *Strict* model, which is reported in the previous section. For each of the benchmark, the execution time

for the *optimized* binary on the *Strict* model represents the benefit of introducing this optimization into traditional non-superscalar stack machine.

6.6.1 Fibonacci Series

The instruction composition of the binaries would be the first hint of whether the optimization is working. Comparing the two composition graphs (Figure 6.19 and Figure 6.20) shows a small reduction in frame register related operations, from 59% to 54% and 49% respectively. Because of the numerous recursive function calls in this benchmark, most of the frame instructions are used for *stack frame construction* instead of local data accesses, which explains the small reduction.

The extra instructions of setting up the local data map every function call is hardly justifiable because of the small function body and limited local variable access (at most three accesses). As can be seen in Table 6.17, the stack frame construction method caused a nearly 9% increase (from 8181 to 8887) in instruction counts. Although most of these extra instructions are stack-to-stack instruction, more instruction simply represents longer execution time for non-superscalar models (*Strict*, *1-Issue* and *1-Issue-BP*). These overhead are covered when superscalar execution is in action as most of these instructions are parallelizable, as can be seen by the higher IPT. Overall, the speed up is miniscule, with the *4-Issue* execution model posing the highest increase (1.04) with respect to the counterpart.

Using the second stack frame construction method, we see a small saving in instruction count (from 8181 to 8002) as the overhead of setting up the local data map is reduced. With smaller instruction count and better ILP, the program consistently outperform its unoptimized counterpart in *all* execution model (Figure 6.17). In particular, the *4-Issue* and *4-Issue-BP* outperform their counterparts by 1.24 and 1.18. Overall, the highest speedup (2.09) posted by the *4-Issue-BP* model compared to conventional stack machine, is quite satisfactory considering the nature of this benchmark.

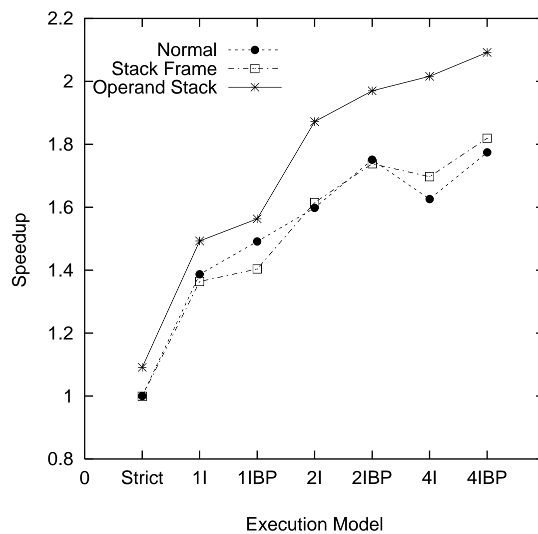


Figure 6.17: Fibonacci Series: Local Variable Access - Speed Up Comparison

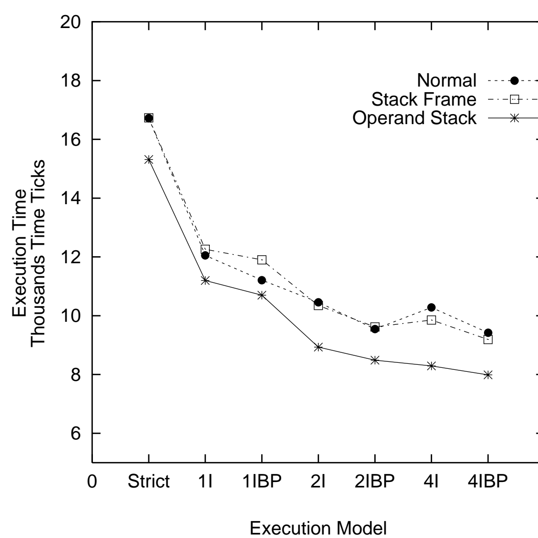


Figure 6.18: Fibonacci Series: Local Variable Access - Execution Time Comparison

	Normal			**Stack Frame**			**Operand Stack**		
	Inst. Count	Time Tick	IPT	Inst. Count	Time Tick	IPT	Inst. Count	Time Tick	IPT
Strict	8181	16720	0.489	8887	16733	0.531	8002	15317	0.522
1-Issue	8181	12049	0.679	8887	12258	0.725	8002	11196	0.715
2-Issue	8181	10457	0.782	8887	10347	0.859	8002	8931	0.896
4-Issue	8181	10280	0.796	8887	9851	0.902	8002	8292	0.965
1-Issue-BP	8339	11212	0.744	8922	11901	0.750	8037	10697	0.751
2-Issue-BP	8585	9546	0.899	9113	9619	0.947	8092	8486	0.954
4-Issue-BP	8648	9423	0.918	9147	9190	0.995	8092	7989	1.013

Table 6.17: Fibonacci Series : Local Variable Access - Comparison

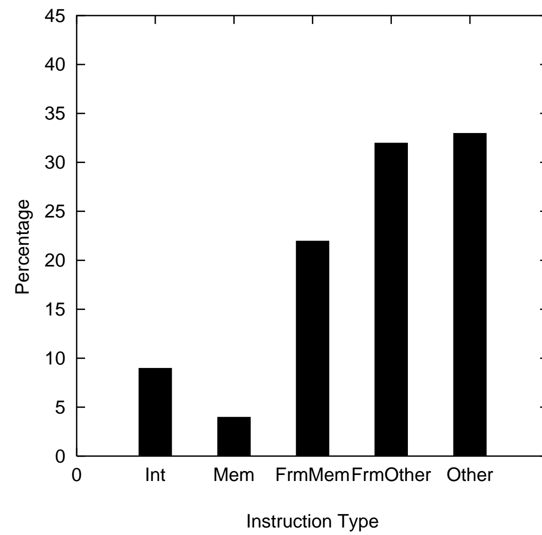


Figure 6.19: Fibonacci Series: Local Variable Access (Stack Frame) Instruction Composition

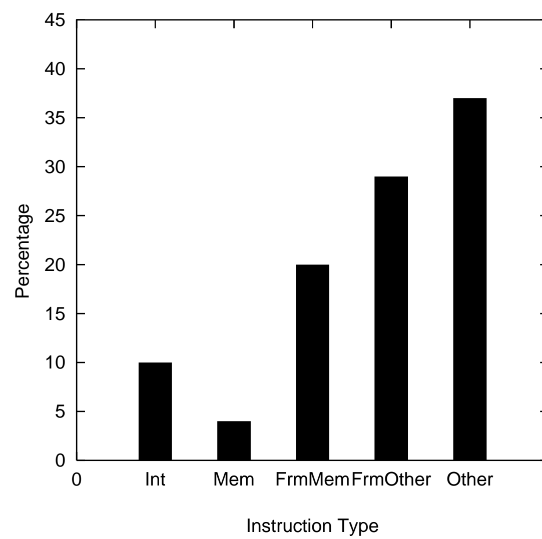


Figure 6.20: Fibonacci Series: Local Variable Access (Operand Stack) Instruction Composition

6.6.2 Sieve of Erathosthense

This benchmark shows that the local data map can have a drastic effect on removing redundant memory operations. The composition graph of the unoptimized program (Figure 6.9) shows a 49% of memory operations via frame register. The psuedocode in Section 5.3.1 further confirms that most of these operations are used for local data access. This huge concentration of memory operations are reduced to **less than 1%** using the local data map optimization, as shown in Figure 6.23 and Figure 6.24. As the Figure 6.21 shows, a traditional stack machine equipped with local data map already shows a speedup of 1.59.

Since there is only one procedure activation in this benchmark (*main* calling the *sieve* function), the overhead of setting up the local data map is negligible, resulting in much greater performance consistently (Table 6.18). As the serialization effect of the memory operations are removed, we get much better ILP in the optimized programs, achieving IPT of 1.47 in some of the execution models.

The net effect of the benefits mentioned is shown by the impressive speedup in Figure 6.21, reaching 4.58 for the *4-Issue-BP* model. This benchmark clearly shows the potential of local data map in program with heavy local data access.

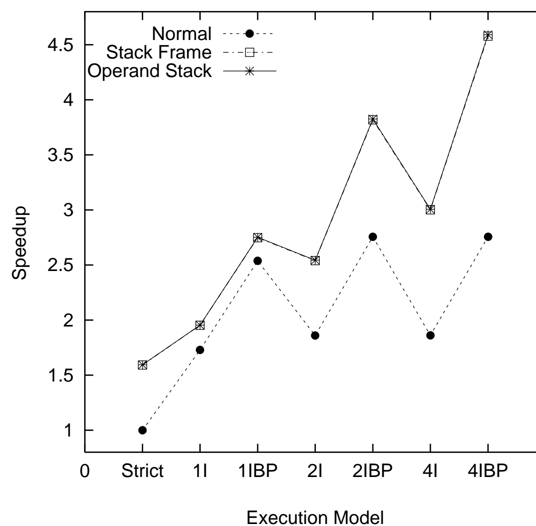


Figure 6.21: Sieve of Erathosthense: Local Variable Access - Speed Up Comparison

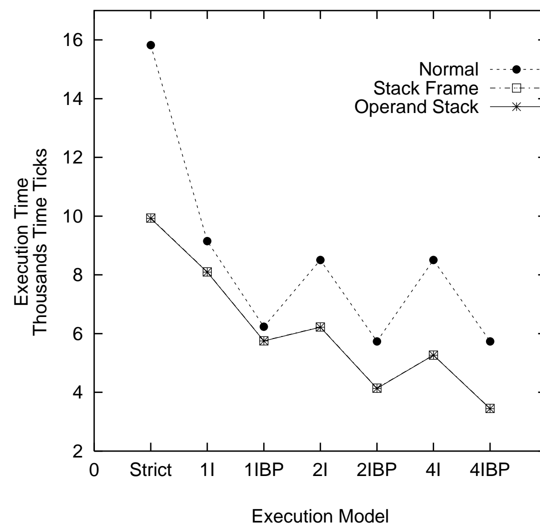


Figure 6.22: Sieve of Erathosthense: Local Variable Access - Execution Time Comparison

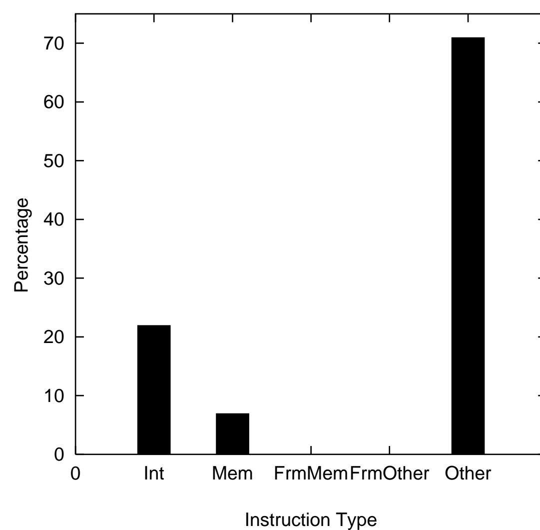


Figure 6.23: Sieve of Erathosthense: Local Variable Access (Stack Frame) Instruction Composition

	Normal			**Stack Frame**			**Operand Stack**		
	Inst. Count	Time Tick	IPT	Inst. Count	Time Tick	IPT	Inst. Count	Time Tick	IPT
Strict	4943	15819	0.312	4951	9935	0.498	4944	9922	0.498
1-Issue	4943	9150	0.540	4951	8101	0.611	4944	8093	0.611
2-Issue	4943	8505	0.581	4951	6227	0.795	4944	6219	0.795
4-Issue	4943	8504	0.581	4951	5271	0.939	4944	5262	0.940
1-Issue-BP	5076	6233	0.814	5018	5756	0.872	5011	5748	0.872
2-Issue-BP	5095	5738	0.888	5085	4143	1.227	5078	4135	1.228
4-Issue-BP	5095	5738	0.888	5101	3454	1.477	5094	3445	1.479

Table 6.18: Sieve of Erathosthense: Local Variable Access - Comparison

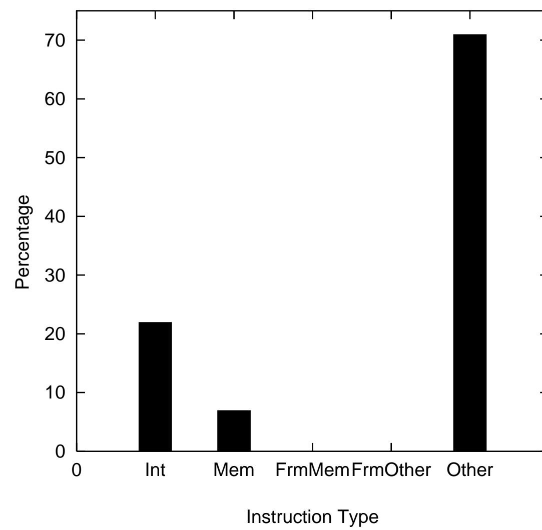


Figure 6.24: Sieve of Erathosthense: Local Variable Access (Operand Stack) Instruction Composition

6.6.3 Quick Sort

The Quick Sort benchmark combines the traits of the two previous program: numerous recursive calls and substantial local data access. The repercussions of these traits under the local data map optimization are clearly present. The recursive calls and the larger number of local data implies more instruction overhead for setting up, saving and restoring of the local data map, as can be seen in instruction count columns in Table 6.19. Balancing the scale is the removal of huge number of memory operations for local data access. Figure 6.27 and Figure 6.28 shows that the frame related instructions have been reduced to 15% and 12% from the original 47%(refer toFigure 6.12).

As noted before, the IPT of the optimized programs are much higher, helped by the better ILP between instructions. The speculative superscalar models benefit the most from this, showing IPT over 1.27.

The speedup gain stand in the middle ground compared to the two previous benchmarks. Figure 6.25 shows a near linear speed up for the optimized versions, with the "operand stack" version slightly better. The best overall speedup 3.73 is posted by the *4-Issue-BP* model using the "operand stack" scheme.

By studying the trend of the graph, we can see that the speedup of the unoptimized version taper off as soon as the *2-Issue-BP* execution model. Further increase in issuing count only gives diminishing return. The new data clearly shows that this is the caused by the serialization effect of the memory unit. With the help of *Local Data Map*, this bottleneck is partially removed.

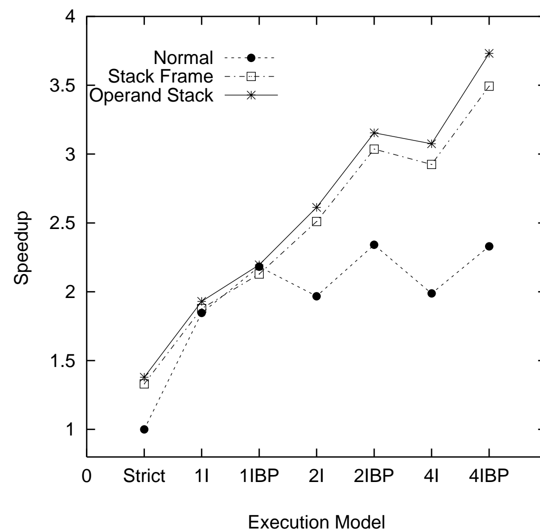


Figure 6.25: Quick Sort: Local Variable Access - Speed Up Comparison

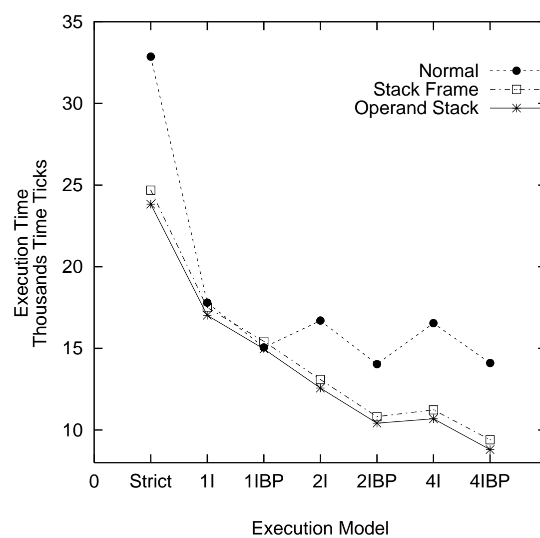


Figure 6.26: Quick Sort: Local Variable Access - Execution Time Comparison

	Normal			**Stack Frame**			**Operand Stack**		
	Inst. Count	Time Tick	IPT	Inst. Count	Time Tick	IPT	Inst. Count	Time Tick	IPT
Strict	11347	32860	0.345	12545	24683	0.508	12118	23827	0.509
1-Issue	11347	17799	0.638	12545	17492	0.717	12118	17018	0.712
2-Issue	11347	16691	0.680	12545	13089	0.958	12118	12572	0.964
4-Issue	11347	16536	0.686	12545	11232	1.117	12118	10685	1.134
1-Issue-BP	12207	15059	0.811	12985	15429	0.842	12563	14959	0.840
2-Issue-BP	13227	14035	0.942	13697	10820	1.266	13270	10417	1.274
4-Issue-BP	13309	14101	0.944	14192	9407	1.509	13817	8803	1.570

Table 6.19: Quick Sort: Local Variable Access - Comparison

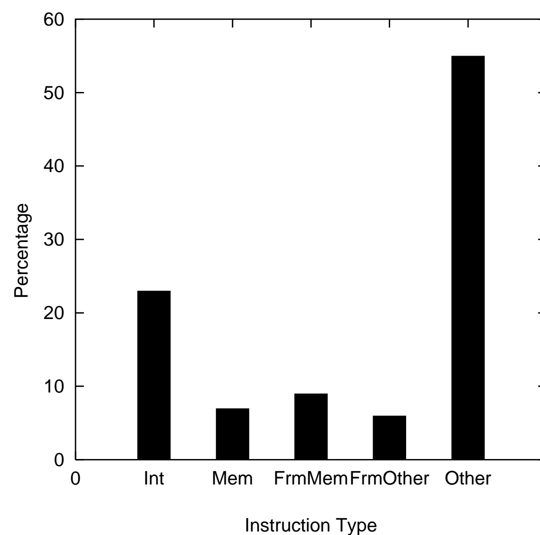


Figure 6.27: Quick Sort: Local Variable Access (Stack Frame) Instruction Composition

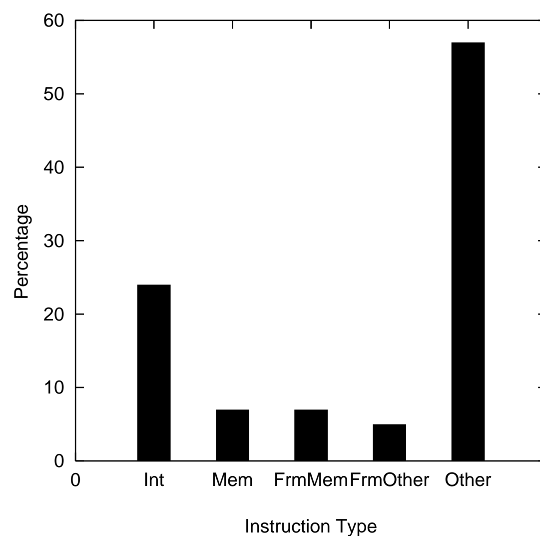


Figure 6.28: Quick Sort: Local Variable Access (Operand Stack) Instruction Composition

6.6.4 Bubble Sort

Table 6.20 summarizes the benchmark data for the bubble sort benchmark. Similar to the Sieve of Erathosthense benchmark, there are only a small number of function calls. Hence, nearly all of the frame related instructions are used for local data access. Figure 6.31 and Figure 6.32 shows the drastic reduction in the frame related instructions, the original heavy concentration of 49% (Figure 6.2) is virtually eliminated.

With most of the frame related instructions removed, combined with the big chunks of integer computation, the benchmark performs very well. As indicated in Figure 6.30, the execution time drops from 240 thousands time ticks to merely 52 thousands under the best execution model *4-Issue-BP* (over 75% reduction).

In Figure 6.29, a good speedup is observed for superscalar execution models. The *2-Issue* models and *4-Issue* models posted a speedup of 2.63 and 3.10 respectively, easily exceeds their counterpart for the unoptimized version. Even better performance are reported for the speculative superscalar models *2-Issue-BP* and *4-Issue-BP*. A linear[1] can seen in Figure 6.29, with speedup of 3.70 and 4.44 under the two models respectively.

The benchmark should further strengthens the case for the local data map optimization. In typical program with small number of function calls, the cost of maintaining the consistency of the local data map is repaid many times over by the performance gain.

[1] Slightly superlinear technically

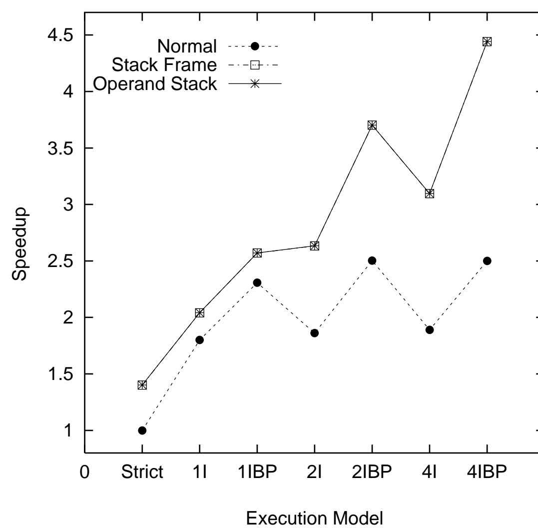


Figure 6.29: Bubble Sort: Local Variable Access - Speed Up Comparison

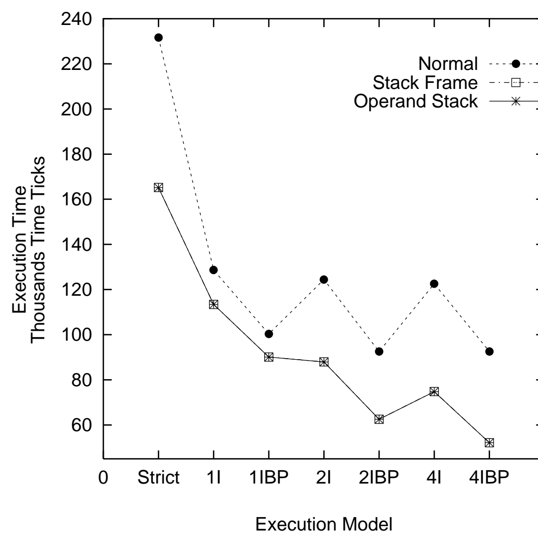


Figure 6.30: Bubble Sort: Local Variable Access - Execution Time Comparison

	Normal			**Stack Frame**			**Operand Stack**		
	Inst. Count	Time Tick	IPT	Inst. Count	Time Tick	IPT	Inst. Count	Time Tick	IPT
Strict	75154	231597	0.325	75180	165204	0.45	75160	165163	0.4555
1-Issue	75154	128620	0.584	75180	113420	0.66	75160	113398	0.6633
2-Issue	75154	124445	0.604	75180	87938	0.85	75160	87919	0.8555
4-Issue	75154	122567	0.613	75180	74790	1.00	75160	74771	1.0055
1-Issue-BP	76689	100386	0.764	76461	90084	0.84	76441	90062	0.8499
2-Issue-BP	77177	92521	0.834	76907	62558	1.22	76887	62539	1.2299
4-Issue-BP	77283	92570	0.835	77211	52139	1.48	77191	52119	1.4811

Table 6.20: Bubble Sort: Local Variable Access - Comparison

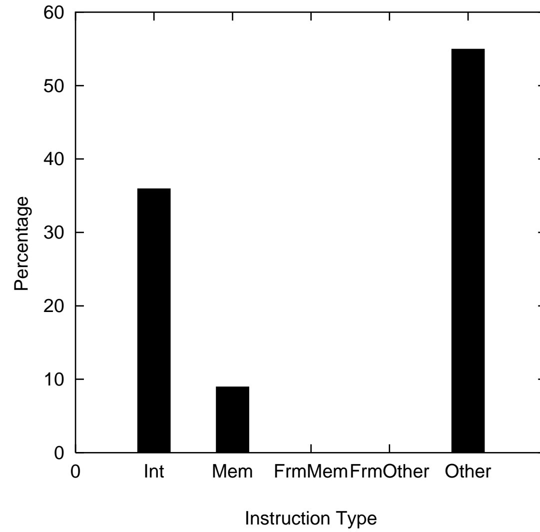


Figure 6.31: Bubble Sort: Local Variable Access (Stack Frame) Instruction Composition

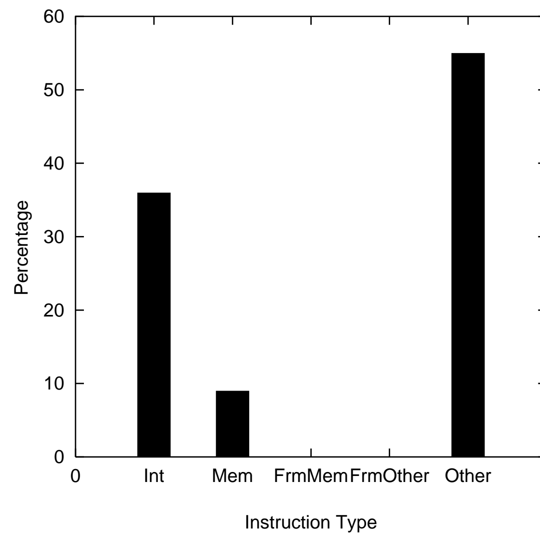


Figure 6.32: Bubble Sort: Local Variable Access (Operand Stack) Instruction Composition

6.7 Conclusion

The benchmarks results presented in this chapter validates our claims for SAFA architecture. In Section 6.2, the benchmarks shows that SAFA is capable of providing good support for high level language constructs. With frame registers, high level language programs can be expressed more concisely when dealing with data structures. Not only the footprint of the program is smaller, the execution time is smaller as well due to the elimination of redundant memory operations.

For the low level instruction execution, Section 6.3 supplies good supporting results that SAFA improves over conventional stack machine is substantial. Superscalar stack execution is shown to be possible and yields favorable speed up consistently. The results also strongly support speculative superscalar stack execution, which is capable of outperforming pure super scalar execution with higher instructions issue capability.

However, it is worth noting that the SAFA architecture does not (at least for the chosen benchmark programs) benefit much from multiple execution units in general. This could be caused by the sparse concentration of instructions of the same type. Also, the high reliance on current running frame for parameters and local variables cause a high reliance on frame register instructions. These instructions are serialized in SAFA, which translates to a huge loss in potential parallelization opportunities.

This problem is partially curbed by the local data map optimization, which fits into the SAFA design easily with a modest extra complexity. This optimization transforms most of the frame related instructions for local data access into stack-to-stack operations. These operations are easily parallelizable, fully realizing the potential of a speculative superscalar stack machine.

Chapter 7

Topical Benchmarks

In this chapter, we attempt to conduct several benchmarking studies into related topics. These forays should complement the more focused nature of the previous chapter to give broader perspective to evaluate the SAFA architecture.

The first topic, covered in Section 7.1, presents the behavior and performance of SAFA architecture during execution of large program. Since the previous benchmarks only observe SAFA with smaller programs, it is important to see whether the same performance trends can be sustained under larger application.

In Section 7.2, we give a quantitative study on the *Instruction Folding* technique (discussed in Section 4.5). As *Instruction Folding* is another well known execution optimization for stack machine, it is essential to study the pros and cons of this technique as compared to the SAFA architecture.

Lastly, we briefly compare SAFA architecture with General Purpose Machine in Section 7.3. Although it is premature to compare SAFA to the well established GPR machines at this point, it is nonetheless important to identify the possible weak points of our design for future development.

7.1 Large Application

The SpecJVM98 benchmark suite[50], being a well recognized yardstick for Java Virtual Machine performance, serves as a good repository for identifying a suitable large application for SAFA. As Java programs require mainly human translation into SAFA assembly, it is not feasible to pick programs that relies heavily on Java APIs. Out of the benchmarks programs in SpecJVM98 suite, the *Compress* benchmark is chosen. Following is a brief summary of this benchmark:

- Implements the LZW[59] adaptive compression algorithm.
- Replaces strings of characters with a single code.
- Build the common strings table on the fly.
- Decompression is similarly performed by constructing the common strings table during processing. Hence, no extra information (except a limited 3 bytes header) is included in the compressed file.

The direct implementations and derivatives of the LZW algorithm is widely available, for example, the *compress* and *uncompress* commands in UNIX. The implementation of the SAFA version follows closely to the SpecJVM98 version, with the following exceptions:

- As SAFA does not support Object natively, a Java object is simulated as a plain data structure instead. All methods take in an extra parameter, which is a memory pointer to the associated Object (simulated as structure). This is similar to the *this* reference.
- Several minor classes, for example the hash table, is changed to a direct array access. In this regard, the SAFA program is similar to the Compress Benchmark in SpecInt95[49].

- As there is no simulated I/O in SAFA, the input file is loaded into memory directly at program start up.

As the simulated object is heavily used in the benchmark, we will now give a brief example to illustrate the idea. Consider the following class definition in Java:

```
class C
{
    ...         //Other variables
    int a;      //object variable 'a'

    void M()
    {
        int b;  //local variable 'b'

        a = 1;
        b = a;
    }
    ...
}
```

The interaction between object variable and local variables can be observed by looking at the method *M( )* in Java Assembly Code:

```
Method void M()

   0 aload_0             //Load 1st parameter, i.e. 'this'
   1 iconst_1
   2 putfield #2 <Field int a>  //store 1 into field 'a'

   5 aload_0
   6 getfield #2 <Field int a>  //load value from field 'a'
   9 istore_1                   //store into local variable #1

  10 return
```

The same method is translated as follows in SAFA:

```
PROC M 1 1          //Procedure M with 1 parameter 1 local

s1: cfb_wload x24   //Load the address of simulated object
s2: ibload 8        //Assumes the offset to 'a' is 8 bytes
s3: iadd            //address of 'a'
s4: ibload 1
s5: wstore          //store 1 into 'a'

s6: cfb_wload x24
s7: ibload 8
s8: iadd
s9: wload           //load from 'a'
s10:cfb_wstore x28  //store into local variable 'b'

s11: EXIT
```

As can be seen an object of class *C* is simulated as a structure which contains all the fields in *C*. Statements *s1* to *s5* simulates the instructions *0* to *2* in the Java program. Note that the slightly longer coding in SAFA is to avoid micro-coding, which allows a more accurate instruction counts. In Java, the *putfield* instruction is usually expanded by trapping.

Although the translation is admittedly ad hoc, the execution behavior is very close. The access of fields (object variables) in Java is processed via memory address calculation followed by direct memory access. The translated program in SAFA exhibits the similar property. As we make no attempt to make SAFA a native JVM, this translation should give acceptable simulation of full-fledged OO program.

A direct result of this translation technique is the large amounts of memory dereferencing in the SAFA program, as the methods make frequent use of the object variables. The benchmark could potentially executes even faster if the code structure in Spec95, which uses C language, is used instead. The equivalent C program uses mainly global variables, or direct passing which reduces memory dereferencing.

7.1.1 Benchmark Result

The *Compress* benchmark compresses the input and subsequently decompressed the compressed information. Two set of benchmarks are performed with different input. A 4000 bytes text file is used for the first experiment, while a 8kb (8192) bytes of binary file is used for the second. As the LZW algorithm can adapt to the input on the fly, the execution metrics scales linearly, regardless of the input type. Table 7.1 summarizes the performance of SAFA as well as SAFA in conjunction with Local Data Map. Table 7.2 similarly summarizes the result for the second set of input.

The main concern for this study is to check whether the observed execution trends in Chapter 6 can be sustained for a larger and more complicated application. The graphs Figure 7.1 and Figure 7.2 restate several findings:

1. Without Local Data Map, the frequent memory operations restricts the potential speedup. The speedup flatten off even with higher issuing rate. Nevertheless, the recorded speedup is quite good: 2.2 highest speedup for non-speculative execution models, and 2.69 for speculative execution models for both benchmarks.

2. Using Local Data Map, the speedup is much better. The highest speedup achieved by non-speculative execution model stands at 3.50 for the first benchmark and 3.54 for the second. On the other hand, speculative models posed an impressive 4.65 and 4.62 for the two benchmarks respectively.

As the discrepancy in the two benchmarks is small enough, it is reasonable to believe that the trends would be similar for even larger input. These results should serve as a assurance for the capabilities of SAFA architecture.

	SAFA			SAFA with LDM		
	Inst. Count	Time Tick	IPT	Inst. Count	Time Tick	IPT
Strict	960065	2638446	0.364	1044976	2049352	0.510
1-Issue	960065	1325054	0.725	1044976	1320212	0.792
2-Issue	960065	1201257	0.799	1044976	890097	1.174
4-Issue	960065	1201235	0.799	1044976	743851	1.405
1-Issue-BP	979450	1088462	0.900	1051158	1138227	0.924
2-Issue-BP	1012306	978051	1.035	1069410	701275	1.525
4-Issue-BP	1014201	977232	1.038	1085440	567571	1.912

Table 7.1: Compress (4000bytes Text): Summary

	SAFA			SAFA with LDM		
	Inst. Count	Time Tick	IPT	Inst. Count	Time Tick	IPT
Strict	2026323	5562578	0.364	2209338	4326682	0.511
1-Issue	2026323	2794960	0.725	2209338	2787161	0.793
2-Issue	2026323	2532720	0.800	2209338	1878702	1.176
4-Issue	2026323	2532694	0.800	2209338	1570062	1.407
1-Issue-BP	2072873	2306219	0.899	2223518	2411606	0.922
2-Issue-BP	2150530	2072399	1.038	2267159	1487871	1.524
4-Issue-BP	2155472	2070946	1.041	2306494	1204813	1.914

Table 7.2: Compress (8kbytes Binary) - Summary

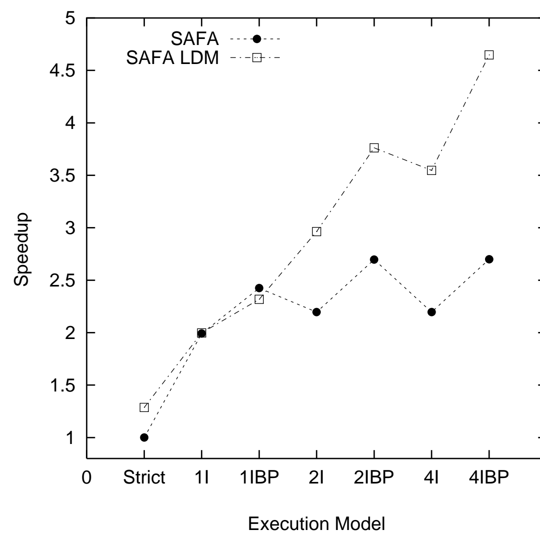


Figure 7.1: Compress (4000 bytes Text) - Speed Up Comparison

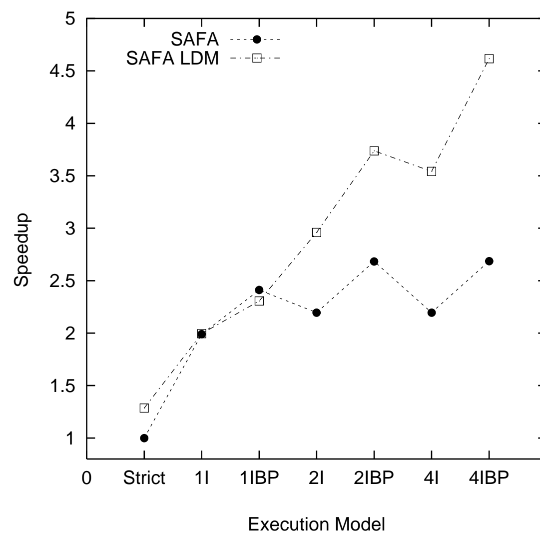


Figure 7.2: Compress (8 kbytes Binary) - Speed Up Comparison

7.2 Instruction Folding

Instruction Folding, as described briefly in Section 4.5, provides another attack angle to increase the performance of traditional stack machines. This section attempt to study this technique under the SAFA architecture. The two main requirements for incorporating Instruction Folding are:

1. Stack based instructions.

2. Ability to retain and specify information in register-like construct.

As the first requirement is easily satisfied, only the second requirement need to be checked. In SAFA, values in the execution core are identifiable with a tag. However, these values are only retained until the arrival of corresponding consumer. With the addition of Local Data Map, local data are retained in the Reorder Buffer and thus satisfying the second requirement. Both of the designs (with or without LDM) will be studied in this section.

The basic premise of Instruction Folding builds on top of a *foldable instructions group*, which is a stream of instructions that can be packed (folded) into a single execution package. The foldable groups are identified via folding type of the instructions. Each instructions can be categorized into one of the following folding types[12].

Acronym	**Meaning**	**JAVA Example**	**SAFA Example**
LV	Load from either local variable or global register or immediate operand (constant).	iload_0, iconst_0, etc	own_wload, ibload, etc
OP	Operation that uses the top two entries of stack	iadd, iand, etc	iadd, iand, etc
BG1	Operation that uses only the topmost entry of stack. This also breaks the folding group.	ifeq, ifge, etc	iftrue, iffalse, etc
BG2	Operation that uses the top two entries of stack. This also breaks the folding group.	if_cmpeq, iaload etc	wstore, hw-store, etc
MEM	Stores to local variable or global register and loading from memory.	istore, fstore etc	own_wstore
NF	Non-Foldable instruction.	dup2, i2f, invoke-virtual, etc	dup, enter, etc

There is a good correspondence in SAFA for most of the folding types. Although there is only one instruction in SAFA (own_wstore) in the *MEM* folding type, it actually covers all the equivalent instructions in Java (istore, fstore, etc) as the instruction in SAFA is not data type sensitive.

To incorporate Instruction Folding into the SAFA, additional processing is added at the decode stage. The decoded instructions is placed in a *Decode Instruction Queue* in program order. This queue (also known as *Decoded Cache Window* in picoJava architecture) is then inspected for foldable instruction group. Similar to the picoJava architecture, the following nine folding groups are supported.

1. *LV, LV, OP, MEM*
2. *LV, LV, OP*
3. *LV, LV, BG2*
4. *LV, OP, MEM*
5. *LV, BG2*
6. *LV, BG1*
7. *LV, OP*
8. *LV, MEM*
9. *OP, MEM*

The only 4-foldable (referring to the number of instructions folded) group is the first group, which make it highly desirable. In each time cycle, the first four instructions in the instruction queue are inspected. If these instructions correspond to any of the folding group listed, they are bundled into a single execution package, which will be issued in the next time tick. If the first instruction is non-foldable (*NF* type) or does not belong to any of the folding patterns, the folding operation will cease. Only this single instruction is issued next. As non-foldable instruction is removed one at a time, and the largest folding group is always produced by the folding logic, folding is always optimum (with respect to the available patterns).

Other than the above additions, the execution models used for the benchmark is different from the SAFA models only in the higher decoding rate. Four instructions are decoded every time tick and the decoded instructions queue contains up to eight decoded instructions. The issuing rate is fixed at one instruction (or one folded group). These hardware parameters are chosen to closely imitate the performance of picoJava architecture.

For the picoJava architecture, each folded group is assumed to be completed in one time cycle. Besides, neither superscalar nor speculative executions are supported. This can be closely simulated by the *Strict* SAFA execution model, which requires the stack top to be ready before next instruction can be issued. To provide a clear upper bound of the performance of instruction folding, we included the result under *1-Issue* execution model, in which one instruction is issued every time tick *regardless* of the readiness of stack top. In conventional picoJava architecture, with its strict stack cache implementation, does not allow for such executions. As such, the results can be taken as strict performance upper bound for instruction folding mechanism under SAFA setting.

Table 7.3 summarizes the results for various benchmarks without LDM. The difference of decoded and issued instruction counts gives the number of folded instructions. SAFA programs that do not use LDM relies on memory access for local data, i.e.there is no use of instructions like own_wload and own_wstore. This reduces the possible folding patterns and results in a low 7.68% folding average across the benchmarks. The speedup over conventional stack machine is at a equally low 1.029 (bounded by 1.727).

By introducing the LDM, the advantage of the Instruction Folding is clearly shown in Table 7.4. A healthy 29.39% average folding rate is achieved. Besides, this also supports accuracy of Instruction Folding simulation in SAFA, as the reported common folding rate lies between 23 to 37 percents, with an average of around 28 percents[13]. The speedup is similarly encouraging, with average of 1.584 (upper bound of 2.223).

The second benchmark shows that the Instruction Folding technique is quite effective, provided stack cache or similar construct is used to retain the execution values. Further comparisons will be conducted in subsequent sections.

Benchmark	Inst. Count		Folding	Speed Up	
	Decoded	Issued	Percentage (%)	Strict	1-Issue
Fibonacci(10)	8181	7508	8.23	1.042	1.452
Sieve(100)	4943	4672	5.48	1.017	1.777
QuickSort(50)	11347	10466	7.76	1.028	1.878
BubbleSort(50)	75154	68206	9.25	1.031	1.801
Average			7.680	1.029	1.727

Table 7.3: Folding Benchmarks without LDM: Summary

Benchmark	Inst. Count		Folding	Speed Up	
	Decoded	Issued	Percentage	Strict	1-Issue
Fibonacci(10)	8887	7718	13.15	1.074	1.484
Sieve(100)	4951	2527	48.96	2.106	2.774
QuickSort(50)	12545	8993	28.31	1.555	2.304
BubbleSort(50)	75180	54789	27.12	1.600	2.328
Average			29.39	1.584	2.223

Table 7.4: Folding Benchmarks with LDM: Summary

7.2.1 SAFA vs Instruction Folding

As both SAFA and Instruction Folding attempt to improve the conventional stack machine, it is natural to pits them against each other. Table 7.5 and Table 7.6 summarize the comparison between SAFA execution model with Instruction Folding. The result is calculated as speedup of SAFA execution time w.r.t Instruction Folding execution time:

$$Speedup = \frac{ExecutionTime_{InstructionFolding}}{ExecutionTime_{SAFA}}$$

The *lower* speedup is calculated against the *Strict* execution model for Instruction Folding, while the *upper* speedup is calculated against the *1-Issue* model.

In the closing of Section 4.5, we theorize that the SAFA architecture can compete with and overcome Instruction Folding when the issuing rate is high enough, since Instruction Folding effectively turns a single issue machine into a multiple issuing one. In Table 7.5 however, SAFA consistently outperforms Instruction Folding

if the lower bound is used. Even when competing with the higher bound of execution time in Instruction Folding, SAFA manages to gain the upper hand as early as the *1-Issue-BP*. As LDM is not utilized in this set of benchmarks, this certainly shows the reliance of the Instruction Folding technique on stack cache and local data manipulation.

By using the LDM (similar to stack cache), we can compare with the full potential of Instruction Folding. As shown in Table 7.6, the advantage for Instruction Folding is harder to overcome if we pick the upper bound performance. In particular, the Sieve of Erathosthense benchmark, with a high folding percentage, gives the only sublinear speedup for SAFA execution model *1-Issue*. By comparing with the upper bound of Instruction Folding performance, it takes a *2-Issue-BP* SAFA model in the worst case (Sieve of Erathosthense), a *1-Issue-BP* SAFA model in the best case to get better performance.

This comparison shows the SAFA architecture can be a better alternatives to improve stack machine performance. As discussed in Section 4.5, Instruction Folding only establishes specific consumer-producer relationships between the instructions. Each folding group basically includes one or more producers and a consumer. However, the limited folding patterns restricts the possible relationships that can be established. Besides, the lack of renaming in the picoJava stack cache implementation further restrict the possibility of superscalar execution. These reasons combined results in reduced potential of this technique.

The SAFA architecture on the other hand, implicitly “folded” each instruction at the issue stage by using the Operand Tag Stack. The ability to issue instructions regardless of the stack top further pushes SAFA ahead of Instruction Folding.

	SAFA Execution Models Speedup					
	1-Issue	**1-Issue BP**	**2-Issue**	**2-Issue BP**	**4-Issue**	**4-Issue BP**
Fibonacci						
lower	1.331	1.431	1.534	1.681	1.560	1.702
upper	*0.956*	1.027	1.101	1.206	1.120	1.222
Sieve						
lower	1.699	2.494	1.828	2.709	1.828	2.709
upper	*0.973*	1.428	1.046	1.551	1.047	1.551
QuickSort						
lower	1.796	2.123	1.915	2.278	1.933	2.267
upper	*0.982*	1.161	1.048	1.246	1.057	1.240
BubbleSort						
lower	1.746	2.237	1.805	2.428	1.832	2.426
upper	*0.999*	1.280	1.033	1.389	1.048	1.388

Table 7.5: SAFA vs Instruction Folding (without LDM): Summary

	SAFA Execution Models Speedup					
	1-Issue	**1-Issue BP**	**2-Issue**	**2-Issue BP**	**4-Issue**	**4-Issue BP**
Fibonacci						
lower	1.269	1.307	1.504	1.618	1.579	1.693
upper	*0.919*	*0.946*	1.088	1.171	1.143	1.225
Sieve						
lower	*0.927*	1.304	1.206	1.812	1.424	2.174
upper	*0.703*	*0.990*	*0.915*	1.376	1.081	1.650
QuickSort						
lower	1.208	1.369	1.614	1.952	1.881	2.246
upper	*0.815*	*0.924*	1.089	1.318	1.270	1.516
BubbleSort						
lower	1.276	1.607	1.646	2.314	1.935	2.776
upper	*0.840*	1.058	1.084	1.524	1.275	1.829

Table 7.6: SAFA vs Instruction Folding (with LDM): Summary

7.2.2 SAFA with Instruction Folding

The previous section presents SAFA and Instruction Folding as competing techniques, however, the two are not mutually exclusive. The Instruction Folding mainly concentrates on the decode stage, while SAFA works during the issue stage. Hence, it is possible to *combine* the two techniques instead of considering them individually. Although the hardware complexity by combining the two would be considerable, the speedup would be much better at lower issuing rate. Hence, the benchmark in this section serves as a brief investigation of the possible benefits.

To allow multiple issues for Instruction Folding, we assume the folding logic is capable to perform multiple foldings per time tick. The speedup comparisons for the four simple programs are presented in Figure 7.3, Figure 7.4, Figure 7.5 and Figure 7.6. The graphs exhibits two main commonalities:

1. The speedup collapsed at four issuing models, as the decode stage can only decode four instructions. Both SAFA, and "SAFA with Folding" are capable of churning through the available decoded instructions which results in stagnant speedup. This also shows that if no higher decoding rate is achievable, plain SAFA architecture is more desirable.

2. The biggest jump in performance occurs at using *1-Issue-BP* execution model. The branch prediction allows more instructions to be considered for folding, while the *1-Issue* model allow constant issuing of instructions. Hence, if a low issuing execution engine is more achievable, then the combination of SAFA and Instruction Folding can be considered.

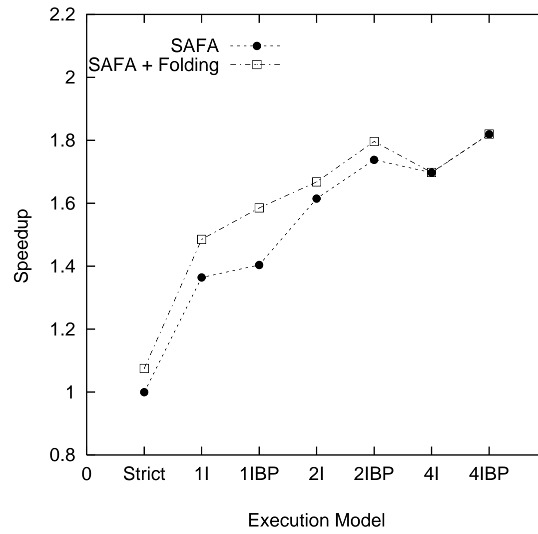


Figure 7.3: Fibonacci Series : SAFA with Folding - Speed Up

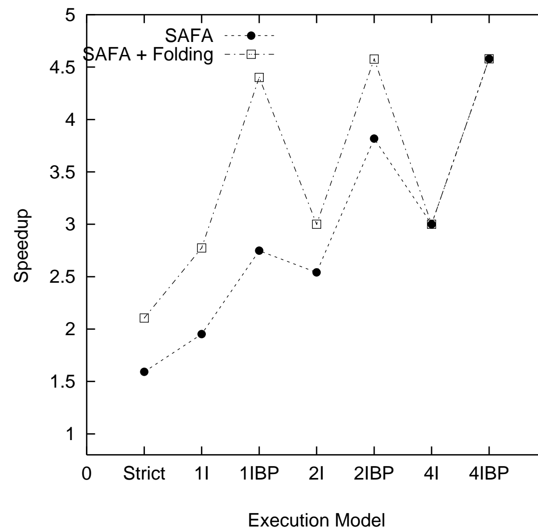


Figure 7.4: Sieve of Erathosthense: SAFA with Folding - Speed Up

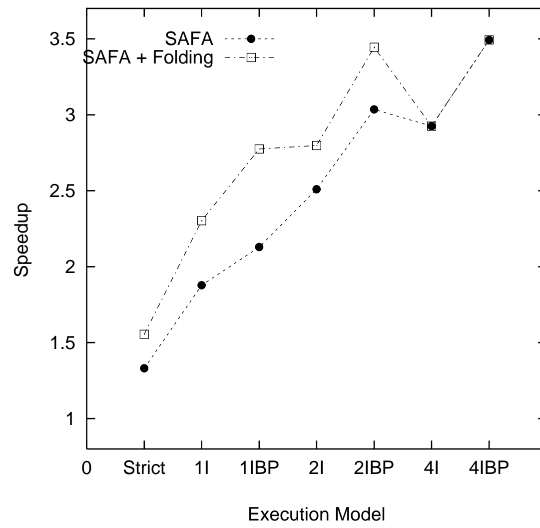


Figure 7.5: Quick Sort: SAFA with Folding - Speed Up

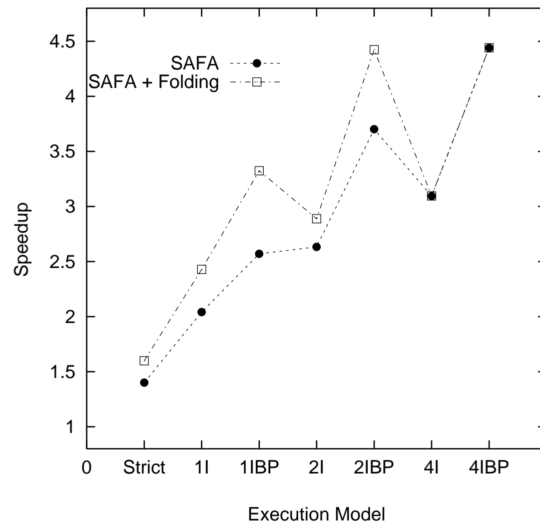


Figure 7.6: Bubble Sort: SAFA with Folding - Speed Up

7.3 General Purpose Register Machine

In this section, we present a comparison with superscalar GPR machine. A superscalar GPR simulator is chosen instead of an actual chip as we need more flexibility in setting up the hardware parameters. SimpleScalar[60] version 2.0 (Portable ISA mode) is used to generate the result under several machine configurations. The SimpleScalar simulator implements the Register Update Unit[61] scheme which allows out-of-order issuing and execution based on register renaming. The Portable Instruction Set Architecture uses a fixed four bytes instructions format that is similar to MIPS.

By pitting SAFA against GPR machine, which is well studied in computer architecture, would be helpful in understanding the pros and cons of the current design. As the program code generated would be totally different, no attempt is made to make sure they are comparable, except in ensuring the source code is as close as possible. A Bubble Sort of 250 numbers is picked because of its heavy computation and infrequent function calls. The binary for SimpleScalar is directly generated by a modified *gcc* compiler (provided by SimpleScalar package) without manual alteration.

The machine models under SimpleScalar is configured to be as close as the counterparts in SAFA. The same naming convention is used for these models for ease of comparison. As we only study speculative execution models, the acronym ***X***-*Issue-BP* represents the machine configuration with **X** decoding and issuing rate. Other than that, a few more important hardware parameters are listed below:

Fetch The fetch queue size is set to 8 bytes.

Integer Execution Unit Unlike SAFA, SimpleScalar contains a dedicated integer multiplication/division unit. To have a close comparison, integer multiplications and divisions are modified to use the normal integer execution unit instead.

Branch Predictor BiModal Predictor is used with 1024 entries.

Commit Rate Instruction commit rate is fixed at 4.

Also, to study the ability of multiple execution units utilization in GPR machine, benchmark for each execution models is repeated for different number of execution unit (integer execution units in this case). Table 7.7 summarize the result. As integer execution take only 1 cycle in SimpleScalar, the IPC is limited by the issuing rate. For example, there is no speedup improvement in the first row even when multiple execution units are present. The best performance is reported by *4-Issue-BP* model with 4 integer units, with IPC of 1.831. However, looking at the fetch rate mentioned previously, we can see that SimpleScalar could be hold back by the low fetch rate (8 bytes is equivalent to 2 instructions only). To see the full potential of GPR machine, a further model *4-Ideal* is setup, where the fetch rate is set to 16 bytes (4 instructions). Further improvement is observed, with the IPC standing at 2.128 for *4-Ideal* models with four integer units.

The results on SAFA with LDM is reported in Table 7.9. As noted before, SAFA fails to utilize the multiple execution units. The only improvement is observed in the *4-Issue-BP* model when the number of execution unit increased to 2. Comparing SimpleScalar with SAFA, we can see that SAFA has better IPC in all cases, except in *4-Ideal*. More notable is the lower instruction counts for SAFA programs compared to GPR programs. Hence, *if* the SAFA core can execute at the same clock as the GPR machine, then SAFA can outperform GPR machine with a lower fetch size and lesser execution units.

The rosy picture in the previous comparison no longer hold up if we take compiler optimization into consideration. Compilers for GPR machine are equally well studied, with many proven techniques. By turning on compiler optimization, the increase in performance is drastic as can be seen in Table 7.8. Much smaller dynamic instruction counts (over 70% reduction) and much better parallelism allows the *4-Ideal* model to achieve IPC of 2.4556 using four integer units. Assuming same

clock rate, this would represent a 2.5 speedup over the best SAFA execution.

To have a clearer picture of the source of such improvement, the assembly code generated for the non-optimized and optimized program are inspected. The following characteristics are found in the non-optimized version:

- This version relies heavily on stack frame for parameters and local variables. This result in frequent memory operation by relative addressing mode.
- Frequent register-to-register movements.
- Redundant calculation.
- Redundant branching code.

The optimized version correctly addresses the above shortcomings by:

- Frequently accessed variables are moved into registers. Less memory movements are needed.
- redundant register movements are removed.
- Commonly used values are recorded to avoid repeated calculation.
- The flow of the program is tightened and restructured and results in lesser branching.

These improvements come from several well studied techniques, like copy propagation, common subexpression elimination, jump optimization etc. As a particular revealing example, the instruction counts for the swapping operation in the unoptimized version is 31, which is reduced to only **2** in the optimized version. The reduced movements (both memory-to-register and register-to-register) increase the "useful" instructions to better utilize the hardware resources.

The SAFA assembly code, when compared to the two versions above, is found to be closer to the optimized version. The structure and flow of the program

is nearly identical. However, as it takes more stack instructions to accomplish a similar task expressed by GPR instruction, SAFA eventually lose out.

As expected, this study highlights the advantage and disadvantage of the SAFA architecture. Since the main SAFA logic is arguably similar to the out-of-order execution logic in GPR machine, SAFA architecture can be expected to perform better with lower hardware requirement (lower fetch rate, lesser execution units). As we know, compiler for stack machine is much simpler, then the SAFA architecture may fit in a niche where fast but efficient compilation is desirable, e.g. real-time HLP interpretation. All this points to a possible utilization in real-time embedded environment.

On the other hand, the disadvantages for SAFA is also very clear. SAFA programs, similar to conventional stack program, suffer from the overhead of stack manipulation instructions. These extra instructions also "dilutes" the density of pure computation instructions such that it is harder to sustain a stream of computation instructions to utilize multiple execution units. Several possible venues to combat this drawback are briefly discussed below.

Firstly, more aggressive bundling can be applied at the decode stage. For example, a VLIW-like approach can be taken to pack several self-contained execution package into a "Instruction Word". This would increase the density of computation instructions going to execution stage.

Secondly, more complicated instructions can be added. For example, the frame register instructions represent a step in this direction. The frame register instructions allows more compact code for memory manipulation, which reduce the overhead. Further instructions can added to allow whole frame interaction, e.g. whole array addition, array scaling etc. This harken back to the SIMD models of early supercomputers.

Last but not least, as SAFA allows superscalar and speculative execution for stack-oriented code, this allows us to reexamine compiler techniques devised for

GPR machine. These techniques were deem inapplicable for stack-oriented code due to the restrictions described in Chapter 4. As GRP machines gain respectable execution efficiency largely due to these well studied techniques, it is not unreasonable to assume these techniques may be adapted and improve the quality of compiler generated stack-oriented code.

	One Int		**Two Int**		**Four Int**	
	Inst. Count	IPC	Inst. Count	IPC	Inst. Count	IPC
1-Issue-BP	2180599	0.7546	2180599	0.7546	2180599	0.7546
2-Issue-BP	2185387	1.0611	2185128	1.4641	2185128	1.4641
4-Issue-BP	2185364	1.0722	2160980	1.8143	2160520	1.8311
4-Ideal					2182935	2.1285

Table 7.7: Bubble Sort(250) on SimpleScalar: Non-Optimized

	One Int		**Two Int**		**Four Int**	
	Inst. Count	IPC	Inst. Count	IPC	Inst. Count	IPC
1-Issue-BP	634192	0.8559	634192	0.8559	634192	0.8559
2-Issue-BP	687644	1.1406	655368	1.6142	655368	1.6142
4-Issue-BP	687633	1.1406	638837	1.7467	631436	1.7441
4-Ideal					686764	2.4556

Table 7.8: Bubble Sort(250) on SimpleScalar: Optimized

	One Int		**Two Int**		**Four Int**	
	Inst. Count	IPC	Inst. Count	IPC	Inst. Count	IPC
1-Issue-BP	1827588	0.895	1827588	0.895	1827588	0.895
2-Issue-BP	1855121	1.549	1855121	1.549	1855121	1.549
4-Issue-BP	1856485	1.859	1856480	2.036	1856480	2.036

Table 7.9: Bubble Sort(250) on SAFA with LDM

7.4 Conclusion

The first study in this chapter shows that larger application can be executed equally well in SAFA. The benchmark results agrees with the simpler, smaller benchmarks in previous chapter.

The study on Instruction Folding supports our claims that SAFA can outperform the technique by having higher issuing rate. In the worst case, a two issuing speculative SAFA execution model is needed to overcome the advantage of Instruction Folding if the folding rate is very high. In many other benchmarks, a one issuing speculative SAFA execution model is good enough.

The comparison between SAFA and GPR machine in the last study highlights the pros and cons of SAFA. As supported by the benchmark results, SAFA can perform well with less stringent hardware requirement. However, the well know stack manipulation overhead stop SAFA from utilizing more execution units.

Chapter 8

Conclusion

This chapter provides a closing to our work. In particular, the contribution of the study is summarized in Section 8.1. Possible future works are discussed in Section 8.2

8.1 Contribution

As mentioned in the previous section, the SAFA architecture is an experiment to support two claims. The more general aim of this work is to substantiate the **General Tagged Execution** framework, which provides a common ground to understand and reconcile various existing execution models. Our work showed that it is possible to place a stack execution core in a conventional superscalar CPU. Traditional superscalar mechanisms and techniques can be adapted to utilized with this core seamlessly.

The proposed SAFA architecture also introduces a simple yet useful addition: frame registers and execution context dependence. The frame registers, with its associated information fields, can be used to represents a piece of consecutive memory space to allow easier and well supported manipulation. This allow high level language construct like array, object etc, which traditionally do not sit well with stack based architecture, to be accessed efficiently with simple coding. In the

two array-based benchmarks, the highest reduction dynamic instruction counts reported is 40%. More importantly, the frame register is not a specialized mechanism that entwine with stack architecture. The idea can survive transplants into other execution model with minor or no modification.

On the low level instruction level execution, SAFA architecture overcomes the inherent bottleneck of stack based instructions. By adapting superscalar mechanism in General Purpose Register machine, SAFA is capable of relaxing the constraints and enable superscalar execution. Benchmarks under superscalar execution reports an average speedup of 1.95 (lowest: 1.39, highest:2.67) with respect to conventional stack execution. As superscalar execution demanding more eligible instructions per cycle to exploit instruction level parallelism, we devised a technique to allow speculative execution in stack based architecture. This technique exploits the inherent dependency of stack instruction to allow simple and clean resolution regardless of the outcome of the branch prediction. Additional benchmarks under speculative executions shows an average speedup of 2.249 (lowest: 1.49, highest 2.67).

Additionally, to cope with the heavy memory operations induced by local data access, a new mechanism *local data map* is utilized to transform these memory operations to more efficient stack-to-stack operations. By leaving the frequently access local data in the core (on the RoB more specifically), access time for these data can be improved. More importantly, ILP between these instructions are now free from the memory bottleneck, further supporting the superscalar capability of SAFA. Benchmarks under this configuration shows substantial improvements. Benchmarks under non-speculative models achieve an average speedup of 2.30 (lowest:1.36,highest:3.10), while benchmarks under speculative models reports an average speedup of 2.97 (lowest:1.404, highest:4.59).

Last but not least, we should point out that these ideas are not “SAFA dependent” in the sense that individual idea can be retrofitted into suitable platform as long as the prerequisites are satisfied. In Figure 8.1, the relationship between each

of the ideas and current existing technology are laid out clearly. For example, the "Reorder Buffer Scheme" can be used to allow superscalar execution in any machine that uses "Stack Based Instructions". With the superscalar stack execution, the "Speculative Scheme" can be added on top to facilitate speculative execution. From the figure, it is also clear that "Frame Register Instructions" is actually an independent path. Any instruction set can incorporate the idea to provide better support for high level languages. Whether the subsequent ideas like "Context Dependence", "Vector Support" etc are needed is totally up to the designer. As such, the SAFA architecture can be viewed as a "virtual" platform that conglomerate all the ideas for experimentation.

8.2 Future Work

Just as there are the two major aims in this work, there are also two possible directions to carry on the study.

For the more general direction, it is possible to further study the compatibility and expressing power of the **General Tagged Execution** framework by substituting part of the execution model with existing or new components. For example, by moving the VLIW packaging techniques into CPU core but retaining a stack based execution core, it is possible to study the possibility of dynamic VLIW execution. Currently, there is already one work pursuing this direction.

For the more specific direction, there are nooks and crannies in superscalar stack execution that awaits exploration. As shown by the benchmark result, binary code resulting from conventional compiler does not exploit the full potential of the SAFA architecture. Most compiler techniques devised in the past decades concentrates on optimization for register based architecture. Although some of these techniques would be equally valid for frame registers optimization for SAFA, the major feature namely superscalar stack execution is not well studied in the context

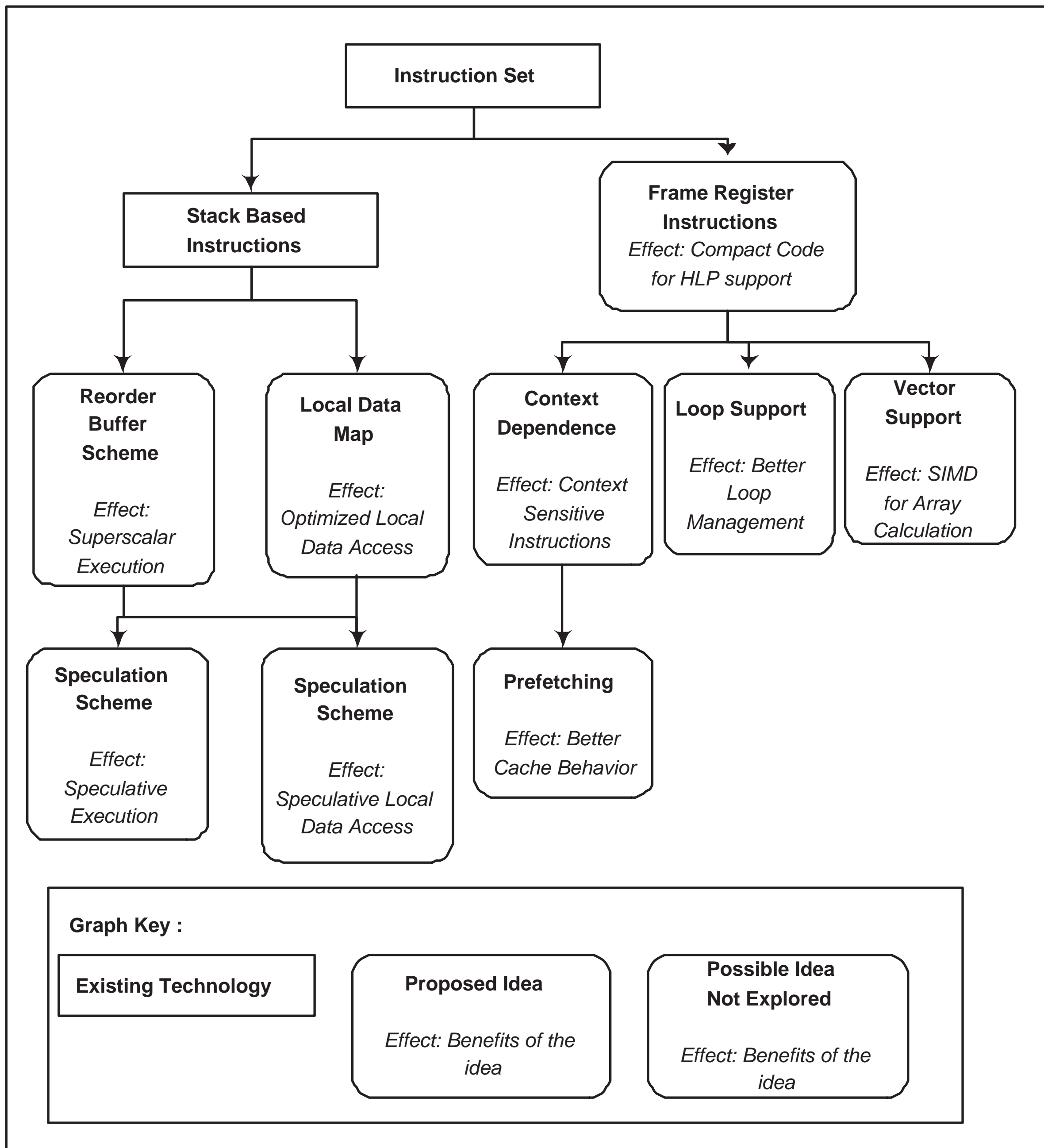


Figure 8.1: Ideas Relationship in SAFA

of compilation. Hence, compilers research is one possible path.

Other than that, the frame register can be further utilized. With its well defined structure and embedded representation of high level data structure, more complicated and specialized instructions can be implemented. For example, with two frame registers representing two separate arrays, we can use just one instruc-

tion to perform vector addition, reminiscent of the SIMD architecture as shown in Figure 8.1.

On the other hand, the context sensitive frame register instructions poses a potential bottle neck during superscalar execution. It is worthwhile to study ways to elevate this pressure on the frame registers, for example, by providing a separate set of instructions working on the "Current Frame Register" and "Previous Frame Register" to allow parallel executions.

Bibliography

[1] Philip J. Koopman, Jr.
Stack Computers - the new wave
Ellis Horwoord Limited, 1989

[2] Ilkka J. Haikala
"More design data for stack architectures"
Proceedings of the ACM '82 conference, January 1982.

[3] Peter Schulthess & Fritz Vonaesch
OPA: a new architecture for Pascal-like languages
ACM SIGARCH Computer Architecture News, Vol 10 Issue 6, December 1982.

[4] John R. Hayes, Martin E. Fraeman, Robert L. Williams & Thomas Zaremba
"An architecture for the direct execution of the Forth programming language"
Proceedings of the second international conference on Architectual support for programming languages and operating systems, Vol 22 Issue 10, October 1987

[5] William F. Keown, Philip Koopman & Aaron Collins
"Performance of the HARRIS RTX 2000 Stack Architecture versus the Sun 4 SPARC and the Sun 3 M68020 Architectures"
ACM SIGARCH Computer Architecture News, Volume 20 Issue 3, June 1992.

[6] Alastair J.W.Mayer
The Architecture of the Burroughs B5000 - 20 Years Later and Still Ahead of

the Times?

ACM Computer Architecture New, 1982

[7] Elliott I. Organick

A Programmer's View of The Intel 432 System

McGraw-Hill Book Company,1983

[8] Elliott I. Organick

Computer Systems Organization - The B5700/B6700 Series

Academic Press,

[9] Christopher Edler

"The Early History of HP3000"

The Analytical Engine, Newsletter of the Computer History Association of California, Vol 3 Number 1, November 1995.

[10] Sun Microsystem

picoJava I - Java Processor Core Data Sheet

Sun Microsystems, 1997

[11] Sun Microsystem

picoJava II - Java Processor Core Data Sheet

Sun Microsystems, 1999

[12] Sun Microsystem

picoJava II - Programmer's Reference Manual

Sun Microsystems, 1999

[13] Harlan McGhan & Mike O'Connor

"PicoJava: A Direct Execution Engine for Java Bytecode"

IEEE Computer, Volume 31 Issue 10, Pages 22-30, October 1998.

[14] Tim Lindholm and Frank Yellin

Java Virtual Machine Specification 2nd Edition

Addison Wesley, 1999

[15] Bill Venners

Inside the Java 2 Virtual Machine 2nd Edition

McGraw Hill, 1999

[16] The Kaffe Virtual Machine

http://www.kaffe.org

[17] R.L. Sites

“Alpha AXP Architecture”

Special Issue, Digital Technical Journal. Vol 4, No.4, 1992.

[18] Peter Bannon & Jim Keller

“Internal Architecture of Alpha 21164 Microprocessor”

IEEE CompCon 1995.

[19] John H.Edmondson, Paul Rubinfeld, Ronald Preston & Vidya Rajagopalan

“Superscalar Instruction Execution in the 21164 Alpha Microprocessor”

IEEE Micro April 1995, Vol 15.

[20] Linley Gwennap

“Digital 21264 Sets New Standard”

Microprocessor Report, October 1996.

[21] R. E. Kessler

“The Alpha 21264 Microprocessor”

IEEE Micro March-April 1999.

[22] James S.Evans & Richard H.Eckhouse

Alpha RISC Architecture for Programmers

Prentice Hall, 1999

[23] IBM Online PowerPC Archieve

“PowerPC Architecture: A high-performance architecture with a history”

http://wwww-1.ibm.com

[24] David Levitan, Thomas Thomas & Paul Tu
"The PowerPC 620(tm) Microprocessor: A High Performance Superscalar RISC Microprocessor" IEEE CompCon 1995.

[25] Apple Computer, Inc. "PowerPC G5 White Paper"
June 2004.

[26] Top500 Supercomputer Site
http://www.top500.org/lists/2004/11/

[27] Richard Y.Kain
Advanced Computer Architecture
Prentice Hall,1996

[28] William Stallings
Computer Organization and Architecture (4th Edition)
Prentice Hall, 1996

[29] Ravi Sethi
Programming Languages – Concepts and Constructs 2nd. Edition
Addion Wesley, 1996

[30] John L. Hennessy & David A. Patterson
Computer Architecture – A Quantitative Approach 2nd. Edition
Morgan Kaufman Publishers Inc, 1996

[31] Jurij Silc, Borut Robic & Theo Ungerer
Processor Architecture: From Dataflow to Superscalar and Beyond
Springer-Verlag 1999.

[32] John Paul Shen & Mikko H. Lipasti
Modern Processor Design: Fundamentals of Superscalar Processor (Beta Edition)
McGraw Hill, 2003.

[33] Tetsuya Hara, Hideki Ando, Chikako Nakanishi & Masao Nakaya
"Performance comparison of ILP machines with cycle time evaluation"
ACM SIGARCH Computer Architecture News , Proceedings of the 23rd annual international symposium on Computer architecture, Vol 24 Issue 2, May 1996.

[34] David W. Wall
"Limits of Instruction-Level Parallelism"
Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, Vol 26 Issue 4, April 1991.

[35] Narayan Ranganathan & Manoj Franklin
"An empirical study of decentralized ILP execution models"
Proceedings of the eighth international conference on Architectural support for programming languages and operating systems, Vol 33, 32 Issue 11, 5, October 1998.

[36] Kevin Scott & Kevin Skadron
"BLP: Applying ILP Techniques to Bytecode Execution"
In the Proceedings of the Second Annual Workshop on Hardware Support for Objects and Microarchitecturs for Java, September 2000.

[37] Daniel P. Siewiorek, C.Gordon Bell & Allen Newell
Computer Structures:Principles and Examples
McGraw-Hill, 1982

[38] R. M. Tomasulo
"An efficient algorithm for exploiting multiple arithmetic units."
IBM Journal of Research and Development,11(1):25-33. January 1967.

[39] Wikipedia, the free encyclopedia
http://en.wikipedia.org

[40] Jack J. Dongarray,Piotr Luszczeky,& Antoine Petitetz "The LINPACK Benchmark: the past, present and future"
December 2001.

[41] Gao YuGuang
"The Design and Implementation of a C to SAFA Compiler"
Master's Thesis. School of Computing, National University of Singapore. March 2005.

[42] Scott McFarling
"Combining Branch Predictors"
DEC WRL TN-36, June 1993.

[43] David J.Lilja
"Reducing the Branch Penalty in Pipelined Processors"
IEEE Computer Volume 21 Issue 7, Pages 47–55, July 1988.

[44] Augutus K.Uht, Vijay Sindagi & Sajee Somanathan
"Branch Effect Reduction Techniques"
IEEE Computer Volume 30 Issue 5, Pages 71–81, May 1997.

[45] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg & John Waldron
"The Case for Virtual Register Machines"
In Workshop on Interpreters, Virtual Machines and Emulators, ACM Press, San Diego, California, Pages 41–49, 2002.

[46] Yunhe Shi, David Gregg, Andrew Beatty & M. Anton Ertl
"Language representations: Virtual machine showdown: stack versus registers"
Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, June 2005

[47] Lee-Ren Ton, Lung-Chung Chang, Min-Fu Kao & Han Min Tseng
"Instruction Folding in Java Processor"

In the Proceedings of International Conference on Parallel and Distributed Systems, 1997.

[48] Lee-Ren Ton, Lung-Chung Chang & Chung-Ping Chung
"An Analytical POC stack operations folding for continuous and discontinuous Java Bytecodes
Journal of Systems Architecture, Vol 48 Issue 1-3, September 2002.

[49] The Spec95 Benchmark Suite
http://www.spec.org/osg/cpu95/

[50] The SpecJVM98 Benchmark Suite
http://www.spec.org/osg/jvm98/

[51] The Java Grande Forum Benchmark Suite
http://www.spec.org/osg/jvm98/

[52] M.Watheq El-Kharashi, Fayez Elguibaly, & Kin F.Li
"An operand extraction-based stack folding algorithm for Java processors"
Second Annual Workshop on Hardware Support for Objects and Microarchitectures for Java, Austin, Texas, USA, September 2000, Pages 22–26.

[53] M.Watheq El-Kharashi, Fayez Elguibaly, & Kin F.Li
"Adapting Tomsaulo's Algorithm for Bytecode Folding Based Java Processors"
ACM SIGARCH Computer Architecture News Volume 29 Issue 5, December 2001.

[54] Ramesh Radhakrishnan, Deependra Talla & Lizy Kurian John
"Allowing for ILP in an Embedded Java Processor"
In the Proceedings of the 27th International Symposium on Computer Architecture, Pages 294–305, June 2000.

[55] R. Radhakrishnan, N. Vijaykrishnan, L.K. John, A. Sivasubramaniam, J. Rubio, & J. Sabarinathan

"Java runtime systems: characterization and architectural implications"
IEEE Transactions on Computers, Vol 50, Issue 2, Feb. 2001, Pages 131-146.

[56] Ramesh Radhakrishnan & Ravi Bhargava, Lizy K. John
"Improving Java performance using hardware translation"
Proceedings of the 15th international conference on Supercomputing, June 2001.

[57] Iffat H. Kazi, Howard H. Chen, Berdenia Stanley & David J. Lilja
'Techniques for obtaining high performance in Java programs"
ACM Computing Surveys (CSUR), Vol 32 Issue 3, September 2000.

[58] T.Suganuma et al
"Evolution of a Java just-int-time compiler for IA-32 platforms"
IBM Journal of Research and Development, IBM Research in Asia Issue, Vol 48 No 5/6, Pages 767-795, 2004.

[59] Terry Welch
"A Technique for High-Performance Data Compression"
IEEE Computer 17, Pages 8-19, 1984

[60] SimpleScalar LLC
http://www.simplescalar.com

[61] Gurindar S. Sohi
"Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers."
IEEE transactions on Computers, Volume 39, Issue 3. Pages 349-359. March 1990.

[62] John Catsoulis
"Transputers - a short historical overview"
http://www.embedded.com.au/Reference/transputers.html

[63] Chris Porthouse

"Jazelle(tm) technology: ARM(tm) acceleration technology for the Java(tm) Platform"

White Paper, ARM Limited, September 2004.

Appendix A

SAFA Assembly Code and Assembler

This appendix gives a brief run through of all currently supported assembly code. Section A.1 to Section A.6 categorize and explain the assembly code according to the instruction type. The assembler written for SAFA is briefly described in Section A.7 along with the assembly code syntax.

To shorten the instruction description, the following notations are used throughout the sections:

Notation	Description
B	A prefix to denote the value take up 1 byte. e.g. *BOffset* means a 1 byte offset value.
HW	A prefix to denote the value take up half a memory word (2 bytes). e.g. *HWOffset* means a 2 bytes offset value.
W	A prefix to denote the value take up a memory word (4 bytes). e.g. *WOffset* means a 4 bytes offset value.

W1.W2	Two words forming one value, for example, a double word integer.
FInfo	Frame register information. The information for the frame register consists of three words: *FInfo1* contains the base address, *FInfo2* contains the item index and index limit, *FInfo3* contains the interval (stripe) for index increment and the size of the elements. Refer to Section 3.1.1 for detailed explanation. When the frame information is on the stack, the *FInfo1* is assumed to be the bottommost of the three words and the *FInfo3* the topmost.

A.1 Frame Register Instructions

The following abbreviations are used for the more frequently used frame registers. Refer to Section 3.1.1 for more details on the types and composition of a frame register.

CFP Current Frame Pointer, points to one of the Frame Registers. Some instructions can change the CFP as a side-effect.

PFP Previous Frame Pointer, points to one of the Frame Registers, automatically take the CFP value when CFP changes.

OFR Own Frame Register, the Frame Register that contains information of the current executing stack frame.

SAFA	Stack		Brief
Mnemonic	Before	After	Description
cfset*X*	...	...	Set CFP to *X*, ranging from 0 to 7.
cfsetglobal	...	...	Set CFP to the Global Frame Register.
cfsetcaller	...	...	Set CFP to the Caller Frame Register.
cfsethost	...	...	Set CFP to the Host Frame Register.
cfsetown	...	...	Set CFP to the Host Frame Register.
cfload	...	...,frm_no	Load the FR number stored in CFP to stack.
pfload	...	...,frm_no	Load the FR number stored in PFP to stack.
cfpfswitch	...	...	Switch CFP and PFP.
frload*X*	...	...,element	Load the current element indexed by FR *X*, ranging from 0 to 7. Does not affect CFP.

frstore*X*	...,element	...	Store the element to location indexed by FR *X*, ranging from 0 to 7. CFP changed to *X* after execution.
idxstore	...,idx	...	Store *idx* from stack to index of CFP.
idxload	...	...,index	Load index of CFP to stack.
itvstore	...,itv	...	Store *itv* from stack to the interval field of CFP.
cfincidx	...	...	Increment the index of CFP, calculated by $Index_{new} = Index_{old} + Interval + 1$.
idxlimitcmp	...	..., diff	Calculate the difference *diff* of index and limit of CFP.
cfdecidx	...	...	Decrement the index of CFP, calculated by $Index_{new} = Index_{old} - Interval - 1$.
cfb_wload *BOffset*	...	...,W	Load a word *W* from location $Base + BOffset$, where base is stored in CFP.
cfb_wstore *BOffset*	...,W	...	Store the word *W* to location $Base + BOffset$, where base is stored in CFP.
cfhw_wload *HWOffset*	...	...,W	Load a word *W* from location $Base + HWOffset$, where base is stored in CFP.
cfhw_wstore *HWOffset*	...,W	...	Store *W* to location $Base + HWOffset$, where base is stored in CFP.
baseloadidx	...,addr	...	Change CFP's base to new address *addr* and clear index.
baseload	...,addr	...	Change CFP's base to new address *addr*.

hw_addbase *HWOffset*	...	...	Add *HWOffset* to base of CFP.
b_addbase *BOffset*	...	...	Add *BOffset* to base of CFP.
newframe	...,W1,W2	...	Create a new frame, using *W1* as *FInfo2*, *W2* as *FInfo3*. The base of new frame is calculated automatically. CFP changed to this new frame after execution.
loadnextfrm	...,offset	..., FInfo1, FInfo2, FInfo3	Load Frame Information from $Base + offset$ to stack, where $Base$ is the base of CFP.
ldfrmcur	...,offset	..., FInfo1, FInfo2, FInfo3	Load Frame Information from $Base + offset$ to stack, where $Base$ is the base of CFP. The *index* is reset to zero.
cfinfoload	...	..., FInfo1, FInfo2, FInfo3	Load Frame Information of CFP to stack.
cfinfoload0	...	...,FInfo1, FInfo2, FInfo3	Load Frame Information of CFP to stack. The index field is set to zero.
cfinfostore	..., FInfo1, FInfo2, FInfo3	...	Store Frame Information from stack into CFP.

idxsetlimit	. . . , FInfo1, FInfo2, FInfo3	. . .	Store Frame Information from stack into CFP. Set the index field to equal to limit field.
frinfoload	. . . ,addr	. . . , FI-nof1, FInfo2, FInfo3	Load Frame Info from address *addr*.
newarray	. . . , E, S	. . . , FInfo1, FInfo2, FInfo3	Create an Array of E number of elements, each element with size S. The array is created as a new frame, with frame info push onto stack.

A.2 Direct Memory Access Instructions

SAFA Mnemonic	Stack		Brief Decription
	Before	After	
0load	...	...,0	Push Immediate Operand *0*.
1load	...	...,1	Push Immediate Operand *0*.
ibload *B*	...	...,B	Push Immediate Operand *B* onto stack.
nibload *B*	...,W	...,W2	Splice *B* onto *W* to get *W2*, i.e. the last byte (least significant byte) of *W* is replaced by *B*.
ihwload *HW*	...	...,HW	Push Immediate Operand *HW* onto stack.
nihwload *HW*	...,W	...,W2	Splice *HW* onto *W* to get *W2*.
iwload *W*	...	...,W	Push Immediate Operand *W* onto stack.
idwload *DW*	...	...,W1,W2	Push Immediate Operand *DW* onto stack.
*X*load	...,Addr	...,Value	Load value from absolute address *Addr*. *X* is size specifier: b, h, w and dw for byte, halfword, word and double-word respectively.
*X*store	...,Addr,V	...	Store value *V* to absolute address *Addr*. *X* is size specifier: b, h, w and dw.

A.3 Integer Instructions

SAFA Mnemonic	Stack Before	Stack After	Brief Decription
iadd	...,W1,W2	...,W	Integer (single word) Addition, $W = W1 + W2$.
diadd	..., WA1,WA2, WB1,WB2	..., WC1,WC2	Integer (double word) Addition, $WC1.WC2 = WA1.WA2 + WB1.WB2$, where $WC1.WC2$ represent a double word integer.
isub	...,W1,W2	...,W	Integer (single word) Subtraction, $W = W1 - W2$.
disub	..., WA1,WA2, WB1,WB2	..., WC1,WC2	Integer (double word) Subtraction, $WC1.WC2 = WA1.WA2 - WB1.WB2$, where $WC1.WC2$ represent a double word integer.
imul	...,W1,W2	...,W	Integer (single word) Multiplication, $W = W1 \times W2$.
idiv	...,W1,W2	...,Q,R	Integer (single word) Division, $Q = W1 \div W2$, $R = W1\%/W2$.
inc	...,W1	...,W	Increment: $W = W1 + 1$
dec	...,W1	...,W	Decrement: $W = W1 - 1$
ieq	...,W1,W2	...,W	Comparison: Equal, $W = (W1 == W2)$. True is represented as 1, false 0.
ineq	...,W1,W2	...,W	Comparison: Not Equal, $W = (W1 \neq W2)$.
igt	...,W1,W2	...,W	Comparison: Greater than, $W = (W1 > W2)$.

ilt	...,W1,W2	...,W	Comparison: Less than, $W = (W1 < W2)$.
ige	...,W1,W2	...,W	Comparison: Greater than or Equal, $W = (W1 \geq W2)$.
ile	...,W1,W2	...,W	Comparison: Less than or Equal, $W = (W1 \leq W2)$.
ieqsign	...,W1,W2	...,W	Comparison: Whether *W1* and *W2* has same sign.
ineqsign	...,W1,W2	...,W	Comparison: Whether *W1* and *W2* has different sign.
ineg	...,W1	...,W	Integer Negation (2s complement), $W = -W1$.
iand	...,W1,W2	...,W	Bitwise AND operation.
ior	...,W1,W2	...,W	Bitwise OR operation.
ixor	...,W1,W2	...,W	Bitwise XOR operation.
inotxor	...,W1,W2	...,W	Bitwise NOT-XOR operation.
iinv	...,W1	...,W	Bitwise Inversion.
imask	...,W1,W2	...,W	Bitwise Masking operation.
i2f	...,I	...,F	Convert Integer *I* to Floating Point Number *F*.
i2d	...,I	...,F1,F2	Convert Integer *I* to Double Floating Point Number *F1.F2*.

A.4 Floating Point Instructions

fadd	...,F1,F2	...,F	Floating Point (single word) Addition, $F = F1 + F2$.
fsub	...,F1,F2	...,F	Floating Point (single word) Subtraction, $F = F1 - F2$.
fmul	...,F1,F2	...,F	Floating Point (single word) Multiplication, $F = F1 \times F2$.
fdiv	...,F1,F2	...,F	Floating Point (single word) Division, $F = F1 \div F2$.
feq	...,F1,F2	...,W	Comparison: Floating Point Equal, $W = (F1 == F2)$. True is represented as 1, false 0.
fneq	...,F1,F2	...,W	Comparison: Floating Point Inequal, $W = (F1 \neq F2)$.
fgt	...,F1,F2	...,W	Comparison: Floating Point Greater, $W = (F1 > F2)$.
flt	...,F1,F2	...,W	Comparison: Floating Point Lesser, $W = (F1 < F2)$.
fge	...,F1,F2	...,W	Comparison: Floating Point Greater or Equal, $W = (F1 \geq F2)$.
fle	...,F1,F2	...,W	Comparison: Floating Point Lesser or Equal, $W = (F1 \leq F2)$.
feqsign	...,F1,F2	...,W	Comparison: Floating Point Equal Sign.
fneqsign	...,F1,F2	...,W	Comparison: Floating Point Inequal Sign.
f2i	...,F	...,I	Convert Floating Point Number F to Integer I.
f2d	...,F	...,F1,F2	Convert Floating Point Number F to Double Floating Point Number $F1.F2$.

dadd	..., FA1,FA2, FB1,FB2	..., FC1,FC2	Floating Point (double word) Addition, $FC1.FC2 = FA1.FA2 + FB1.FB2$, where $FC1.FC2$ represent a double word floating point number.
dsub	..., FA1,FA2, FB1,FB2	..., FC1,FC2	Floating Point (double word) Subtraction, $FC1.FC2 = FA1.FA2 - FB1.FB2$.
ddiv	...,FA1,FA2, FB1,FB2	..., FC1,FC2	Floating Point (double word) Division, $FC1.FC2 = FA1.FA2 \div FB1.FB2$.
dmul	..., FA1,FA2, FB1,FB2	..., FC1,FC2	Floating Point (double word) Multiplication, $FC1.FC2 = FA1.FA2 \times FB1.FB2$.
deq	..., FA1,FA2, FB1,FB2	...,W	Comparison: Double Floating Point Equal, $W = (FA1.FA2 == FB1.FB2)$. True is represented as 1, false 0.
dneq	..., FA1,FA2, FB1,FB2	...,W	Comparison: Double Floating Point Inequal, $W = (FA1.FA2 \neq FB1.FB2)$.
dgt	..., FA1,FA2, FB1,FB2	...,W	Comparison: Double Floating Point Greater, $W = (FA1.FA2 > FB1.FB2)$.
dlt	..., FA1,FA2, FB1,FB2	...,W	Comparison: Double Floating Point Lesser, $W = (FA1.FA2 < FB1.FB2)$.
dge	..., FA1,FA2, FB1,FB2	...,W	Comparison: Double Floating Point Greater or Equal, $W = (FA1.FA2 \geq FB1.FB2)$.

dle	..., FA1,FA2, FB1,FB2	...,W	Comparison: Double Floating Point Lesser or Equal, $W = (FA1.FA2 \leq FB1.FB2)$.
deqsign	..., FA1,FA2, FB1,FB2	...,W	Comparison: Double Floating Point Equal Sign.
dneqsign	..., FA1,FA2, FB1,FB2	...,W	Comparison: Double Floating Point Inequal Sign.
d2i	...,F1,F2	...,I	Convert Double Floating Point Number *F1.F2* to Integer *I*.
d2f	...,F1,F2	...,F	Convert Double Floating Point Number *F1.F2* to Floating Point Number *F*.

A.5 Branching Instructions

SAFA Mnemonic	Stack Before	Stack After	Brief Decription
goto *BOffset*	...	...	Branch to Program Counter (PC) + *BOffset*.
wgoto *WOffset*	...	...	Branch to Program Counter (PC) + *WOffset*.
enter	...,Addr	...	Procedure Entry. The following conditions must be met: - PFP points to FR of Host of Callee. - CFP points to FR of Callee's stack frame. - Callee address on stack as *Addr*. To facilitate the procedure calling, an Assembler Macro **penter** is provided. Refer to Section A.7.

return	...,Addr	...	Return from procedure. The following conditions must be met: - PFP points to FR of Host of Caller. - CFP points to FR of Caller's caller. - Caller return address on stack as *Addr*. To setup the above conditions, the Assembler Macro **exit** can be used. Refer to Section A.7.
iftrue *BOffset*	...,W	...	Branch to Program Counter (PC) + *BOffset* if *W* is true (one).
iffalse *BOffset*	...,W	...	Branch to PC + *BOffset* if *W* is false (zero).
hw_iftrue *HWOffset*	...,W	...	Branch to Program Counter (PC) + *HWOffset* if *W* is true (one).
hw_iffalse *HWOffset*	...,W	...	Branch to Program Counter (PC) + *HWOffset* if *W* is false (zero).
ifgt *BOffset*	...,W	...	Branch to Program Counter (PC) + *BOffset* if integer *W* is greater than zero.
di_ifgt *BOffset*	...,W1,W2	...	Branch to Program Counter (PC) + *BOffset* if double integer *W1.W2* is greater than zero.

iflt *BOffset*	...,W	...	Branch to Program Counter (PC) + *BOffset* if integer *W* is lesser than zero.
di_iflt *BOffset*	...,W1,W2	...	Branch to Program Counter (PC) + *BOffset* if double integer *W1.W2* is lesser than zero.
ifge *BOffset*	...,W	...	Branch to Program Counter (PC) + *BOffset* if integer *W* is greater than or equal zero.
di_ifge *BOffset*	...,W1,W2	...	Branch to Program Counter (PC) + *BOffset* if double integer *W1.W2* is greater than or equal zero.
ifle *BOffset*	...,W	...	Branch to Program Counter (PC) + *BOffset* if integer *W* is lesser than or equal zero.
di_ifle *BOffset*	...,W1,W2	...	Branch to Program Counter (PC) + *BOffset* if double integer *W1.W2* is lesser than or equal zero.
ifeq *BOffset*	...,W	...	Branch to Program Counter (PC) + *BOffset* if integer *W* is equal to zero.
di_ifeq *BOffset*	...,W1,W2	...	Branch to Program Counter (PC) + *BOffset* if double integer *W1.W2* is equal to zero.
ifne *BOffset*	...,W	...	Branch to Program Counter (PC) + *BOffset* if integer *W* is not equal to zero.

di_ifne *BOffset*	...,W1,W2	...	Branch to Program Counter (PC) + *BOffset* if double integer *W1.W2* is not equal to zero.
flag	...	...	A special flag for debugging. Refer to Section B.2 details.
halt	...	...	Halt the execution.

A.6 Stack Manipulation Instructions

SAFA Mnemonic	Stack Before	Stack After	Brief Decription
shiftl N	...,W1	...,W	Bitwise Shirting Left. $W = W1 << N$, with N ranging from 1 to 32
shiftr N	...,W1	...,W	Bitwise Shifting Right. $W = W1 >> N$, with N ranging from 1 to 32
pop$X$	...,W1, ..., WX	...	Erase X number of words from stack, with X ranging from 1 to 6.
swap	...,W1,W2	...,W2,W1	Reverse the two topmost words.
dwswap	..., WA1,WA2, WB1,WB2	..., WB1,WB2, WA1,WA2	Reverse the two topmost double words.
dwbtcycle3	..., WA1,WA2, WB1,WB2, WC1,WC2	...WB1,WB2, WC1,WC2, WA1,WA2	Cycle the three topmost double words from bottom to top.
dwtbcycle3	..., WA1,WA2, WB1,WB2, WC1,WC2	...WA1,WA2, WC1,WC2, WB1,WB2	Cycle the three topmost double words from top to bottom.
btcycle3	...,W1, W2, W3	..., W2, W3, W1	Cycle the three topmost words from bottom to top.
tbcycle3	...,W1, W2, W3	..., W3, W1, W2	Cycle the three topmost words from top to bottom.
btcycle4	...,W1, W2, W3, W4	..., W2, W3, W4, W1	Cycle the four topmost words from bottom to top.

tbcycle4	...,W1, W2, W3, W4	..., W4, W1, W2, W3	Cycle the four topmost words from top to bottom.
dup	...,W	...,W,W	Replicate the topmost word.
dup*X*	...,W	..., W, W, ..., W	Replicate the topmost word *X* times, where *X* range from 2 to 4.
dwdup	..., W1, W2	..., W1,W2, W1,W2	Replicate the topmost double word.
dwdup*X*	..., W1, W2	..., W1,W2, ..., W1,W2	Replicate the topmost double word *X* times, where *X* range from 2 to 4.
hwdup	..., HW	..., HW.HW	Duplicate the halfword *HW* within one word *W*, such that *W* is composed of two identical *HW*.
bdup	..., B	..., B.B.B.B	Quadruplicate the byte *B* within one word *W*, such that *W* is composed of four identical *B*.
bsplit	..., W	..., B1, B2, B3, B4	Split the topmost word *W* into four bytes, where *B1* is the most significant byte and *B4* the least.
hwsplit	..., W	..., HW1, HW2	Split the topmost word *W* into two halfwords, where *HW1* is the more significant half.
bcntbit	..., B	..., N	Counts the number of '1' bit in byte *B*.
hwcntbit	..., HW	..., N	Counts the number of '1' bit in halfword *HW*.

wcntbit	. . . , W	. . . , N	Counts the number of '1' bit in word *W*.
dwcntbit	. . . , W1, W2	. . .	Counts the number of '1' bit in double word *W1.W2*.

A.7 SAFA Assembler Introduction

A SAFA assembly program, following the procedural paradigms, consists of a number of independent code modules (procedures). Each procedure contains two major components:

1. **Data Components**: A number of parameters, which serves as "input" to the procedure. A few local variables to be used as temporary storages.

2. **Code Components**: A sequence of assembly code to perform the required computation.

The syntax for various components will be covered in the subsequent sections.

A.7.1 Syntax for Procedure

The syntax of a procedure is summarized in Figure A.1.

During execution, the *data components* of a procedure is encapsulated in a *stack frame*. This special stack frame is named *Own Stack Frame* to distinguish it from others. Information pertaining to OSF is stored in a special register *Own Frame Register (OFR)*. An example of stack frame is given in Figure A.2.

When a procedure is entered, the Current Frame Pointer (CFP) is automatically set to OFR. The CFP can then be used to manipulate the local variables and access the parameters. Because of this, a programmer must be careful when using frame register instructions (Section A.1) that alter the CFP automatically. The CFP should be reset to OFR by the instruction `setown` under such cases. Before a procedure is exited, the return value should be place on top of the stack. The CFP will be reset to the caller's OFR after procedure activation.

Other than the syntax above, there are some other semantic restrictions

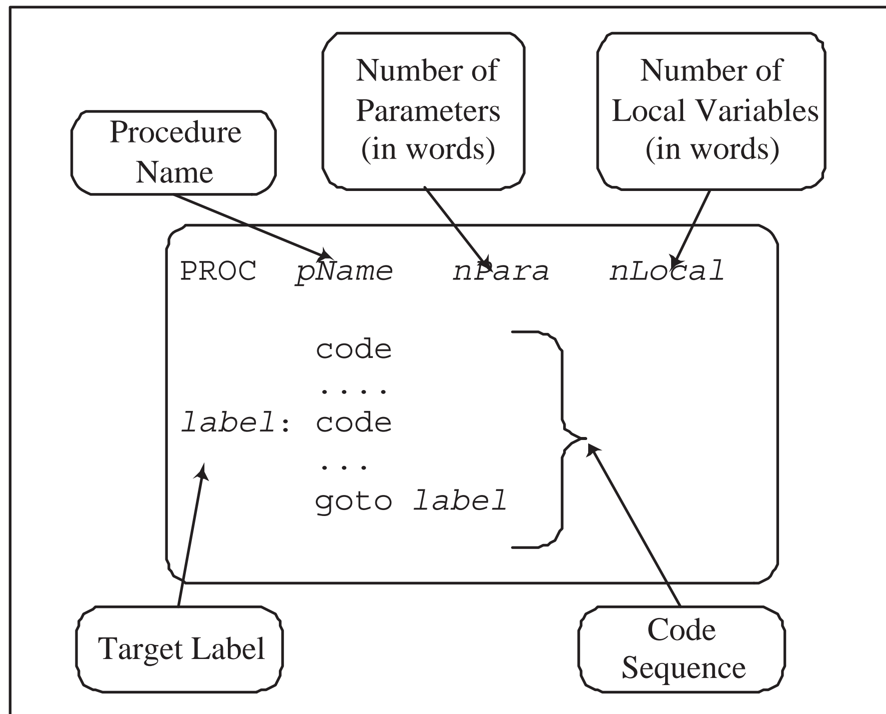


Figure A.1: Syntax for a Procedure in SAFA Assembly Code.

on the procedure definition:

1. There must be a procedure with the name "main" to serve as a starting point of the program execution.

2. The definition of a procedure must precedes its activation. Currently, mutually recursive procedures are not supported by the assembler.

A.7.2 Syntax for Data Values

In quite a number of the SAFA assembly code, the programmer is required to supply a data value as operands, offset etc. For example the instruction cfb_wload (refer to Section A.1) requires a *BOffset*. A data value can be specified using one of the following formats:

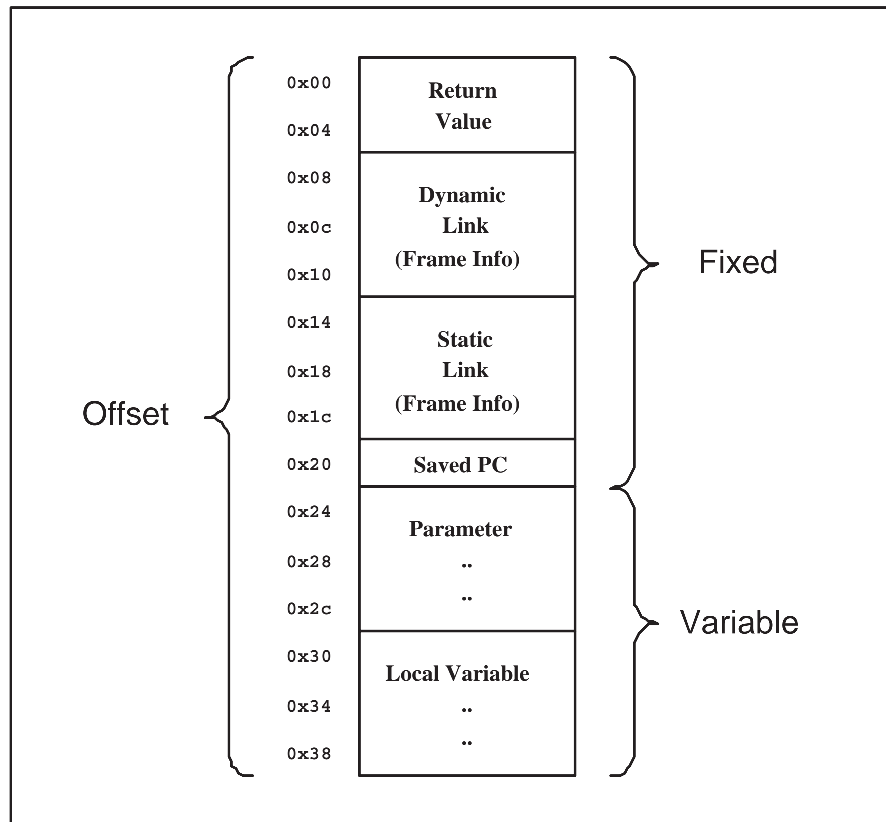


Figure A.2: Layout of a Procedure Stack Frame

Format	Example	Description
Decimal	1, 4096, -12345 etc	A decimal value can be simply used without additional specifier
Hexadecimal	x24, x1f, x1abc etc	Use prefix 'x' to specify hexadecimal values.
Floating Point	<f3.14>, <f1.23e3>, <f-1.72> etc	Use prefix 'f' and put the value in enclosing caret signs '< ' and '>'. Scientific notation of the form *mantissa*e*exponent* is also supported.
Double Floating Point	<d3.1415927>, <d3.45e10>, <d-1.2> etc	Use prefix 'd' and put the value in enclosing caret signs '<' and '>'. Scientific notation of the form *mantissa*e*exponent* is also supported.

As an example, to load the first parameter residing at offset 24_{16} of a procedure onto a stack, we can use

```
cfb_wload x24
```

Or converting 24_{16} to 36_{10} and use

```
cfb_wload 30
```

To load the value of π onto stack, we can use the following code

```
iwload <f3.1415>
```

Unlike *JVM* byte-code, a data value has *no* type information after it is on stack. Hence, the same iwload instruction can be use to load an integer, for example *12345* onto stack:

```
iwload 12345
```

A.7.3 Built in Assembly Macros

For the SAFA simulator, we decided to have *no* microcoded instruction at all. Each instruction would just perform a very simple task, which conceivably can be completed under one time tick. For example, the procedure activation "enter" instruction (refer to Section A.5) just jump to the destined procedure address and change the few special frame registers (the *Own*, *Host*, *Caller*) accordingly. The task of setting up the callee stack frame, which includes setting up dynamic link, static link etc is assumed to be the programmer's task. Although this improves the accuracy of the simulator, this add considerably burden to the programmer. To ease the programming task, a number of commonly used code sequences are packaged into macros. These macros are only valid at the assembly level, which will be expanded automatically during assembly process. The following table summarize the available macros in SAFA:

SAFA Macro	Stack Before	Stack After	Brief Decription
PENTER *FR*,*ProcName*	..., Param1, ..., ParamN	...	Setup the callee frame and then activate the target procedure *ProcName*. The frame register to hold the new stack frame during construction is specified by *FR*, ranging from 0 to 7. Note that the original content of the frame register *FR* will be destroyed. The parameters for the target procedure must be loaded onto the stack starting from left to right i.e. last parameter should be the topmost on the stack.

SAFA	Stack		Brief
Macro	Before	After	Decription
EXIT *FR1*,*FR2*	..., Result	..., Result	Exit the current procedure. The required information for the return instruction (refer to Section A.5) will be setup automatically, using frame register *FR1* and *FR2* as temporary storage. The original content for these two frame registers will be destroyed.
SAVEFRM *FR*,*BOffset*	...	...	Save the information of frame register *FR* onto the own stack frame at offset *BOffset*. It is the responsibility of the callee procedure to preserve the content of the all the frame registers upon returning. So, before using a frame register, a procedure is required to save the information for later restoration.
RESTOREFRM *FR*,*BOffset*	...	...	Restore the frame register information stored at offset *BOffset* to frame register *FR*. Usually used before a procedure return to restore the content of any used frame registers.

A.7.4 Sample Translation

Given a simple C-like high level language program as follows:

```
int Multiply(int x, int y)
{
    int result;

    result = x * y;
    return result;
}

void main()
{
    int a;

    a = Multiply(3,5);

}
```

The corresponding SAFA assembly code is given below:

```
PROC Multiply 2 1
    cfb_wload x24
    cfb_wload x28
    imul
    cfb_wstore x2c
    EXIT 1,2

PROC main 0 1
    ibload 3
    ibload 5
    PENTER 3,Multiply
    cfb_wstore x24
```

A.7.5 Using the assembler

The SAFA assembler *safaAs* is written using *yacc* and *lex*, both from the software suite *Software Generation Utilities for Solaris-ELF 4.0* and compiled by *gcc* (v2.95.2 for Solaris).

A two passes assembly process is used:

1. Invoke by *safaAs_1Pass* ¡ *SAFA Assembly Program*.
2. Invoke by *safaAs output_file* ¡ *SAFA Assembly Program*.

A simple script *assembler* is provided to perform the two passes. The syntax is as follows:

```
assembler SAFA_Assembly_Program Output_File
```

As a convention, the assembly program uses the extension ".d", where as the SAFA binary uses the extension ".safa".

Appendix B

SAFA Simulator

In Section 5.1, we mentioned that a software simulator of the SAFA architecture is built for experimentation. A brief introduction is given in this appendix, for the various versions of the simulator. The main difference for the versions are basically in the presentation i.e. user interface with the same underlying execution core. In Section B.1, the simplest bare-bone version is introduced. The more user-friendly and information-rich version is presented in Section B.2.

The software tools used in building these versions are listed below:

Platform *Sunfire* 4800 server with 8 CPUs.

Compiler *gcc* version 2.95.2 19991024 (release).

GUI *Tcl/Tk* version 8.0.

B.1 Simulator in Plain Text

This version of simulator is suitable for a quick execution of SAFA binary. The binary is loaded and executed until the `halt` instruction is encountered, or a maximum number of time ticks specified by user elapsed. Only a statistic file, an optional

memory dump and an optional final machine state information are available at the termination of the simulation. Since it is not possible to get any other information during execution, this version is best for gathering benchmark result. If debugging or more detailed in-execution information are required, the GUI version in the next section would be more suitable.

The command and arguments to invoke this simulator is given below:

```
safaHalt input.safa [-d flags] [-tick tickLimit] [-c configFile]
     [-memdump] [-trace] [-f statFile]
```

The usage of the options and arguments is summarized in the following table:

Options	Arguments	Description
-d		Enable debugging messages for the Simulator. Depending on the arguments, one or more units in the SAFA simulator will output debug message *every* time tick. Causes extreme slow down of the simulation and generate huge amount of information on screen.
	f	Enable debug messages for fetch unit.
	d	Enable debug messages decode unit
	i	Enable debug messages issue unit
	e	Enable debug messages execution units
	m	Enable debug messages memory
	s	Enable debug messages cpu
	o	Enable debug messages instructions
	r	Enable debug messages frame register
	+	Turn on all debug messages
-tick	tickLimit	The *tickLimit* determines the maximum number of time ticks for the simulation. The default value is 30000.

-c	configFile	Configures the various units in SAFA simulator according to the information in the file *configFile*. The format is given in Section B.1.1. The default configuration file used is "*safa.cfg*" in the working directory.
-memDump	-	Output the memory state at the end of simulation. Format is given in Section B.1.3.
-trace	-	Output the CPU state at the end of simulation. Format is given in Section B.1.3.
-f	statFile	Produce the statistic in the file "*statFile*" instead of the default file name "*stat.dat*" in the working directory.

B.1.1 Configuration File

As mentioned in Section 5.4, various hardware parameters are available for tweaking to setup different execution models. To facilitate frequent modification, these parameters are grouped and stored in a configuration file. A sample is given in Figure B.1. As can be seen, each set of parameters is preceded by a descriptive line, which should make the file quite self-explanatory.

B.1.2 Statistic File

This is arguably the most important file for a simulation. It is generated automatically after each simulation, containing statistic information for all the units in SAFA simulator. For all the data reported, they are accompanied by informative messages, which allow easy interpretation and understanding. A sample statistic file is given in Figure B.2

```
Fetch Size (multiples of four)
8
Multiple Decode/Issue Rate
1
Issue Q Threshold (# of instructions)
1
Instruction Queue Size
24
Reorder Buffer Size: VTag Size,Driz In,Out Threshold
16
32
4
12
Integer Exe Unit: nUnits, resrv entries + pipeline stages
1
4
2
Float Exe Unit: nUnits, resrv entries + pipeline stages
1
4
4
Load/store unit: resrv entries + ports + store delay
8
4
1
Cache: Hit Delay, Miss Delay
1
4
Instruction Cache: On/Off, #blocks, blockSize, #set.
0
8
4
4
Data Cache: On/Off, #blocks, blockSize, #set.
0
8
4
4
Frm Inst Unit: max exe, entries
1
8
Memory Size (in bytes, multiples of four)
8192
Branch Predictor, on/off, maxlevel
0
3
```

Figure B.1: Sample Configuration File

```
Statistic for Executing sample_source/linpack10_CONV.safa.
Configuration Config/I4_BP.cfg

******** SAFA Configuration *****

Aggressive Mode: OFF
Memory Size 8192 bytes
Instruction Queue Size = 24 bytes
Decode/Issue Rate = 4 instruction(s)
Issue/Decode Queue Threshold = 4 instruction(s)
Reorder Buffer with
RTAG = 16, VTAG = 32, Drizzle [In = 4|Out = 12]
Frame Instruction Queue Size = 8 instruction(s)
Branch Predictor : 1, Max Level : 3

******* Statistic for SAFA *****

Stall Statistics:
Issued Q > Threshold : 7862
Decoded Q > Threshold : 585
Branching : 4214 times
Switch to new address: 1829 times

******* Statistic for Fetch Unit *****
Number of Ticks: 29934
Stalled: 16158 tick(s)
Bytes Fetched: 108137 byte(s)

******* Statistic for Fetch Unit End *****

******* Statistic for Instruction Cache  *****
Number of Ticks: 29934
Total Access: 0
Hit: 0
Hit percentage = NaN
******* Statistic for Instruction Cache End *****

******* Statistic for Decode Unit *****
Number of Ticks: 29934
Stalled: 11513 tick(s)
No Decode: 17774 tick(s)
Empty Queue: 2312 tick(s)
Partial Decoding: 280
Total Inst Decoded: 36263
Branch Encountered: 2625 inst(s)
Cannot Speculate: 6835 inst(s)
******* Statistic for Decode Unit End ****
```

Figure B.2: Sample Statistic File (Part1)

```
******* Statistic for Branch Predictor *****
Predicted : Total 1405 Times
True : 0 Time(s)
False : 1405 Time(s)
Correct Predictions : 1098
Percentage : 78.149%
******* Statistic for Branch Predictor End *****

******* Statistic for Issue Unit *****
Strict Issue: Disabled
Number of Ticks: 29934
Stalled: 7862 tick(s)
Virtual Tag Usage: Total = 234756, Avg = 7
Real Tag Usage: Total = 44989, Avg = 1
Drizzle: Out = 0 Failed = 0, In = 0 Failed = 0
Inst Issued: 31205 inst(s)
Int Inst: 8409 inst(s)
Float Inst: 1567 inst(s)
Mem Int: 3166 inst(s)
Freg Int: 15438 inst(s)
Branch Int: 2625 inst(s)
Immediate Inst: 5029 inst(s)
Total Ticks Not Issuing Any Insts: 19117 tick(s)
Not Issued Due to RoB Overflow: 0 inst(s)
Not Issued Due to RoB Underflow: 0 inst(s)
Not Issued Due to VTag: 0 inst(s)
Not Issued Due to No Decode: 11255 tick(s)
******* Statistic for Issue Unit End *****

******* Statistic for Frame Inst Unit *****
Number of Ticks: 29934
Inst Executed: 15438 inst(s)
Mem Inst: 13201 inst(s)
Queue Full for : 151 tick(s)
******* Statistic for Frame Inst Unit End *****

******* Statistic for Int 1 Exe Unit *****
Number of Ticks: 29934
Inst Executed: 7885 inst(s)
Pipe Full: 2610 tick(s)
RS Full: 5087 tick(s)
Idle: 9114 tick(s)
******* Statistic for Int 1 Exe Unit End *****
```

Figure B.3: Sample Statistic File(Part2)

```
******* Statistic for Flt 1 Exe Unit *****
Number of Ticks: 29934
Inst Executed: 1450 inst(s)
Pipe Full: 0 tick(s)
RS Full: 0 tick(s)
Idle: 17359 tick(s)
******* Statistic for Flt 1 Exe Unit End *****

******* Statistic for Brn Exe Unit *****
Number of Ticks: 29934
Inst Executed: 2622 inst(s)
Pipe Full: 2622 tick(s)
RS Full: 354 tick(s)
Idle: 17506 tick(s)
******* Statistic for Brn Exe Unit End *****

******* Statistic for Load/Store Unit *****
Number of Ticks: 29934
Load Inst: 13071 inst(s)
Store Inst: 3655 inst(s)
Port Full: 139 tick(s)
RS Full: 0 tick(s)
Idle: 8333 tick(s)
******* Statistic for Load/Store Unit End *****

******* Statistic for Data Cache  *****
Number of Ticks: 29934
Total Access: 12183
Hit: 12183
Hit percentage = 1.00
******* Statistic for Data Cache End *****

******* Statistic for SAFA End *****
```

Figure B.4: Sample Statistic File (Part 3)

```
....
<0x0000000c> | b1 04 3c 3a
<0x00000010> | 2b 2c 24 44
<0x00000014> | 04 fd 42 00
<0x00000018> | 46 00 01 ff
<0x0000001c> | c4 46 00 15
<0x00000020> | 00 08 03 44
<0x00000024> | 04 00 34 2d
<0x00000028> | 3c 2d 38 2d
....
```

Figure B.5: Sample Memory Dump File (Partial)

B.1.3 Memory Dump and CPU State

As the plain-text version simulator does not give any information during execution, it is sometime needed to check the final machine state for confirmation that the execution has been performed correctly[1]. If the correctness of the execution needs to be confirmed, then a per time tick execution state must be inspected. For such cases, the GUI version of the simulator would be more suitable.

A complete machine state is formed by the main memory state (all memory values) and the CPU state (all components state). The memory state is stored in a file, which is conventionally named as *memory dump* file, with the following format:

```
<Memory Address> | byte_0 byte_1 byte_2 byte_3
```

A partial memory dump can be found in Figure B.5 which listed a part of the memory space.

For the CPU state (or conventionally known as CPU *trace*), the following information is included:

PC and NextPC The current program counter and the program counter for the

[1]Note that although highly unlikely, a machine state can still be correct even when the execution is *wrong*

```
****** TimeTick = 1510 Starts ******
PC = <0x0000073a> NextPC = <0x0000073a>

*Issue Unit Information Starts *
RoB Usage2 | Free 18 | OTS 2
Operands Tag Stack
-=-=-=-=-=-=-=-=-=
RT 1 => RoBEntry [1e|54|1|W]
RT 0 => RoBEntry [09|4a|1|W]
*Issue Unit Information Ends   *

Issue Output:
Resrv.Entry No.1 : <0x50> for Integer Unit

*Frame Instruction Unit Information Starts*
-------------------------------------------
Frame Registers :
No.0 FR[B<00000000>|L<00000>|I<00000>|S<000>|It<000>] READY
            .............. Removed....................
No.7 FR[B<00000000>|L<00000>|I<00000>|S<000>|It<000>] READY

Global: FR[B<000007e0>|L<00008>|I<00000>|S<004>|It<000>] READY
Host: FR[B<000007e0>|L<00008>|I<00000>|S<004>|It<000>] READY
Caller: FR[B<00000800>|L<00286>|I<00000>|S<004>|It<000>] READY
Own: FR[B<00000c78>|L<00016>|I<00008>|S<004>|It<000>] READY

CFR[8] PFP[11]
*Frame Instruction Unit Information Ends  *
****** TimeTick = 1510 End ******
```

Figure B.6: Sample CPU Trace File (Abridged)

next time tick.

Issue Unit Information The information for operands tag stack, as well as issued instructions are listed.

Frame Register Information The detailed information for each of the frame registers are reported.

A abridged example of the CPU trace is given in Figure B.6.

B.2 Simulator with GUI

The plain text SAFA simulator discussed previously is best suited to simply finished a SAFA program execution. If the behavior of the SAFA architecture is to be studied more deeply, then the GUI version would be much more handy. The GUI SAFA simulator is built specifically to give full control to the user. Some of the main features for this simulator are:

- A per time tick view of the machine state.
- Fuller information for all components.
- Inspection and conversion of values stored in memory.
- Turning on/off debug messages at run time.

The command to invoke this simulator, which is very close to the plain text version is given as follows:

```
safa_tk input.safa [-d flags] [-tick tickLimit] [-c configFile]
     [-memdump] [-trace] [-f statFile]
```

The various options and arguments are interpreted in the same way as in Section B.1. Figure B.7 shows a screen snapshot of this simulator in work. The panels for the individual components will be described in details next.

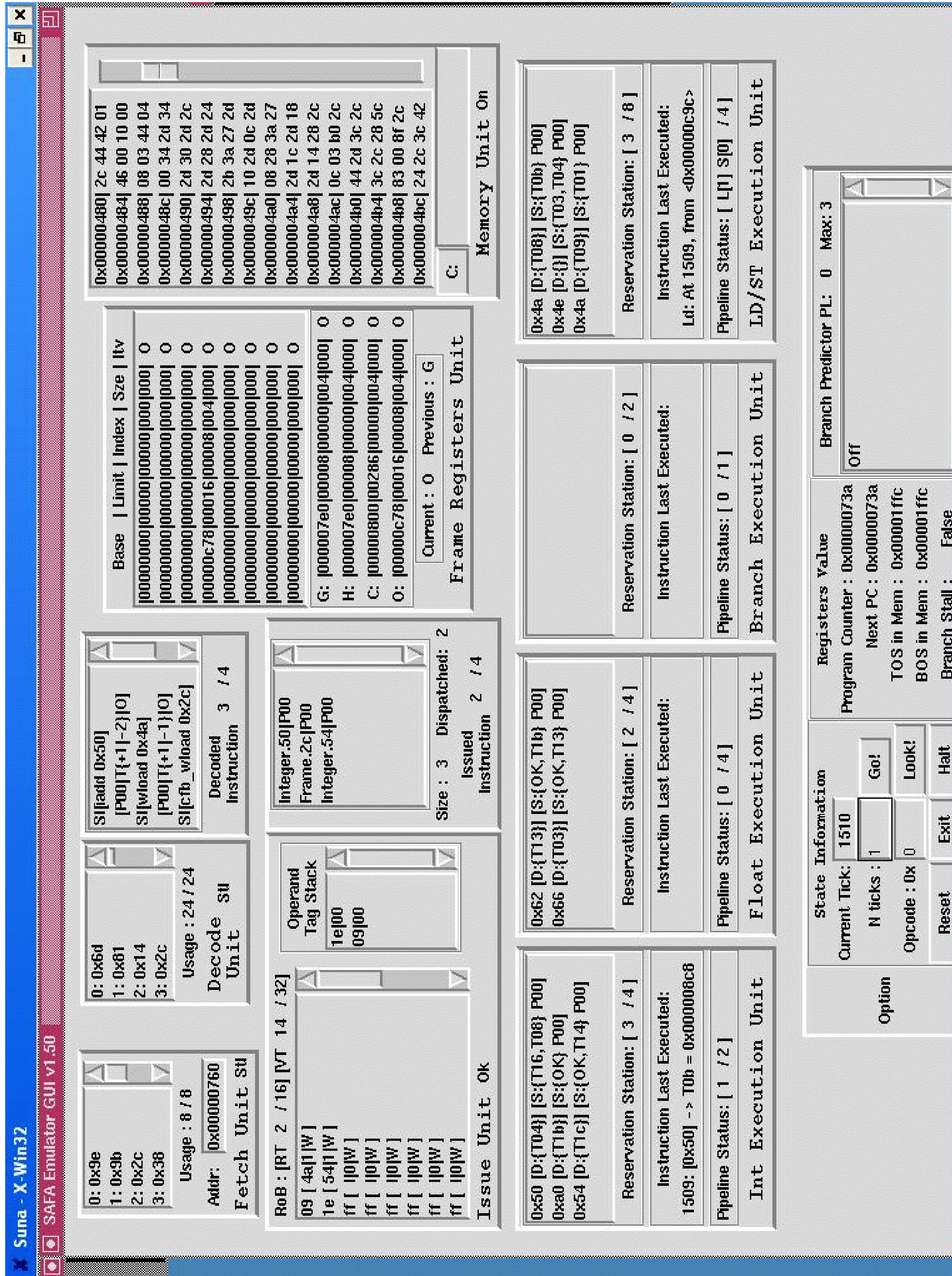


Figure B.7: SAFA Simulator GUI v1.5 Screen Shot

B.2.1 Main Control Panel

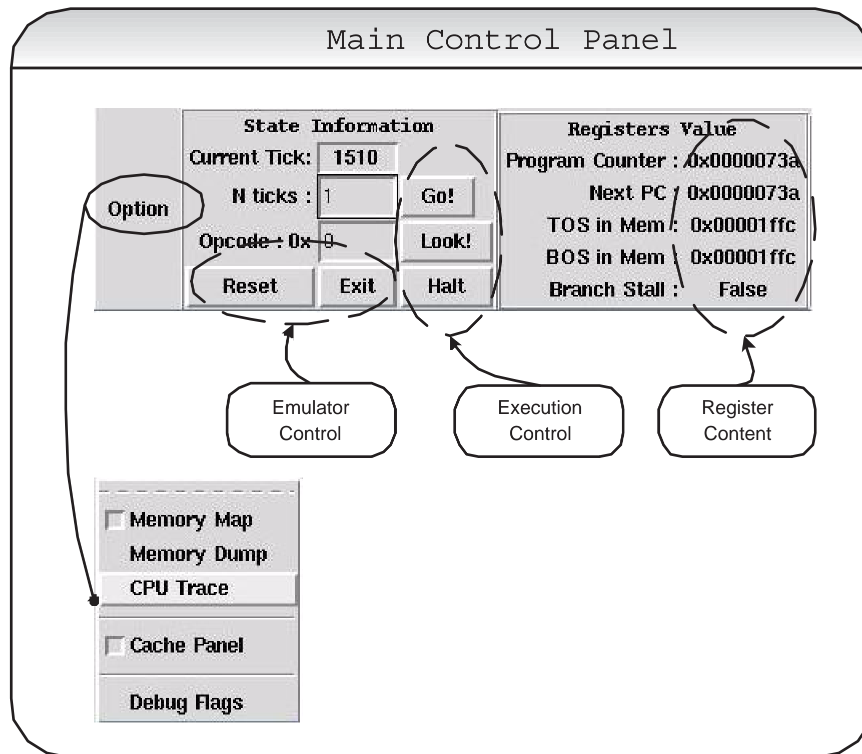


Figure B.8: Main Control Panel GUI

The main control panel shown (Figure B.8) contains system-side information and serves as the control center of the simulation. The information provided are basically the value of the few register in the simulator, summarized in the following tables.

Register	Description
Program Counter	The address of the current instruction, in hexadecimal.
NextPC	The address of the next instruction, in hexadecimal.
TOS in Mem	When RoB overflows, part of the stack will be ship off to main memory and **TOS** is used to keep track of the **T**op **O**f **S**tack.
BOS	Likewise, **BOS** keep track of the **B**ottom **O**f **S**tack in memory.
Branch Stall	Not exactly a register, it is actually just a signal whether the system is stalled (fetching and decoding stopped) because of branch instruction.

To conduct the simulation, the following commands are provided:

Button	Input	Description
Go!	*Nticks*	The simulator executes *Nticks* cycles.
Look!	*Opcode*	The simulator executes until the *Opcode* is found (decoded). If the assembly code is embedded with `flag` instruction, this can serves as a debugging mechanism.
Halt	-	The simulator executes until the `halt` instruction is decoded.
Exit	-	Shutdown the simulator.
Reset	-	Reset the simulator.

Other less frequently used options are grouped under a pop-up menu (click on the "Option" button). The options are:

Option	Description
Memory Map	Switching on/off of the display for memory unit.
Memory Dump	Dump the current memory content to the file “MemDump.dat”.
CPU Trace	Capture the current CPU state in the file “Trace.dat”.
Cache Panel	Show the instruction and data cache panels.
Debug Flags	Set/Un-set the Simulator debug messages. These messages are shown on the standard output.

B.2.2 Components Window

Fetch Unit

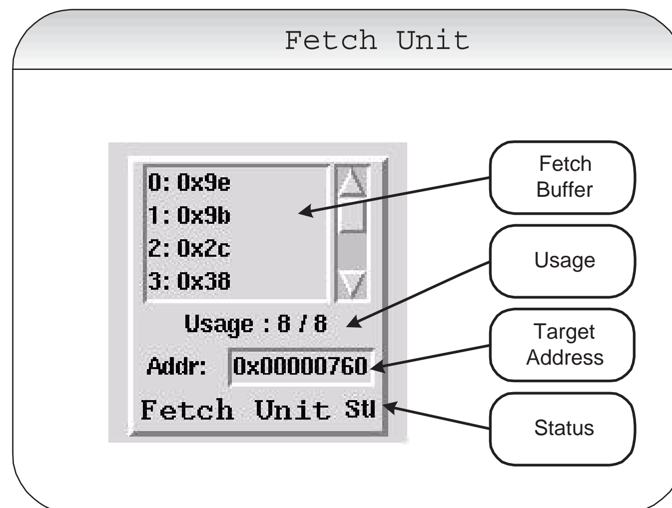


Figure B.9: Fetch Unit GUI

Field	Format	Description
Fetch Buffer	*Index*:0x*Value*	Buffer for fetched code. *Index* shows the relative position of each byte *Value*.
Target Address	0x:*Addr*	Next fetching target. Note that this may be far advanced compared to the PC because of the fetch ahead.
Usage	*Cur*/*Max*	Current number *Cur* of maximum *Max* temporary storage used.
Status	*Ok* or *Stl*	Whether the unit is working *Ok* or stalled *Stl*.

Decode Unit

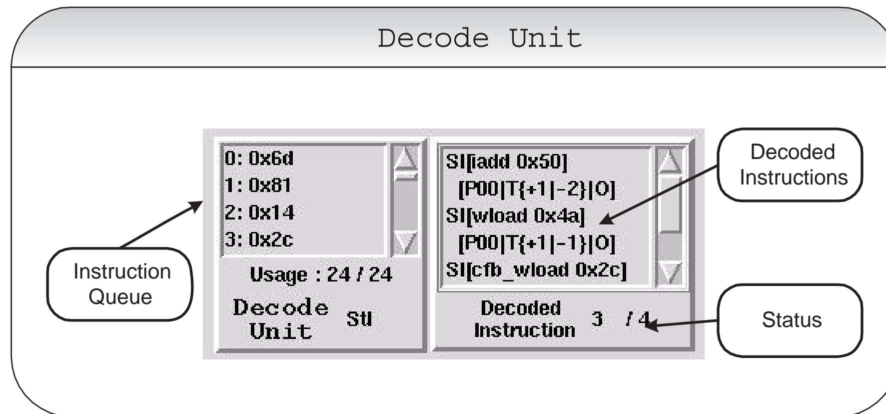


Figure B.10: Decode Unit GUI

Field	Format	Description
Instruction Queue	*Index*:0x*Value*	Buffer for the byte codes. *Index* shows the relative position of each byte *Value*.
Status	*Ok* or *Stl*	Whether the unit is working (*Ok*) or stalled (*Stl*).

Field	Format	Description
Decoded Instructions	$SP[Mne \quad OP]$ $[PL\|\mathrm{T}\{Push\}\|Pop\}\|Imp]$	***SP*** Whether the instruction can be speculated (**SI**) or not (**I**). **SI** Instruction that can be speculated. **I** Non-speculative Instruction. ***Mne*** SAFA Assembly Mnemonic. ***OP*** Opcode. ***PL*** Prediction Level. ***Push*** Number of tags pushed. ***Pop*** Number of tags poped. ***Imp*** Implementation status: **O** = implemented, **X** = not implemented.

Issue Unit

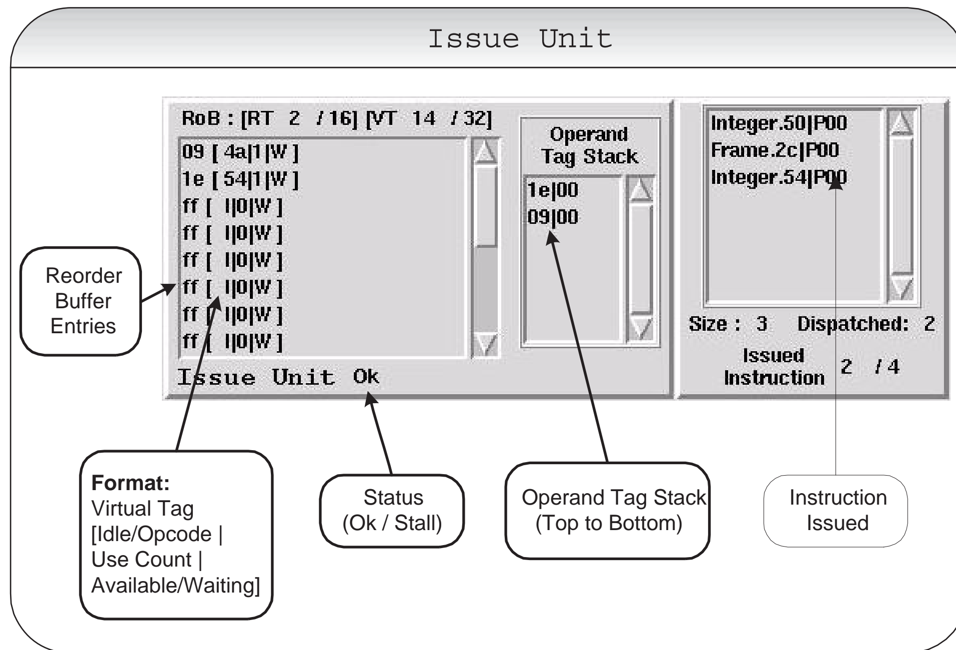


Figure B.11: Issue Unit GUI

Field	Format	Description
RoB Entry	$T[OP \mid UC \mid ST\ ]$	***T*** Virtual Tag Number assigned. ***OP*** Opcode of the producer instruction. ***UC*** Use count of the entry. ***ST*** Status: **W** = waiting, **A** = available.

Field	Format	Description
Operand Tag	$Tag\|PL$	***Tag*** Virtual Tag Number. ***PL*** Associated Prediction Level.
Issued Instrcution	$EU.OP\|PL$	***EU*** Execution Unit for this instruction. ***OP*** Opcode of this instruction. ***PL*** Associated Prediction Level.

Frame Register Unit

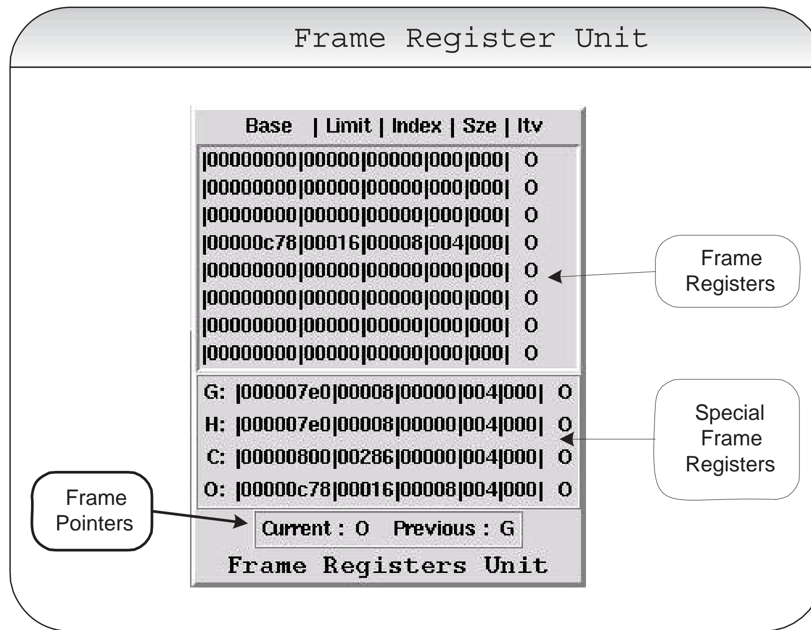


Figure B.12: Frame Register Unit GUI

Field	Format	Description
General Frame Register Information	$\|BA\|Lmt\|Idx\|\ Sze\|Itv\|$	***BA*** Base Address, in hexadecimal. ***Lmt*** Upperbound of number of items, in decimal. ***Idx*** Index of current item, in decimal. ***Sze*** Size (number of bytes) of one item, in decimal. ***Itv*** Interval to skip while moving index (number of items), in decimal.

Special Frame Register	Same As General Frame Register	Same As Above
Frame Pointer	*FrNum*	The CFP and PFP, can be either general frame registers (from 0 to 7) or special registers (**G**lobal, **H**ost, **C**aller, **O**wn).

Branch Predictor Unit

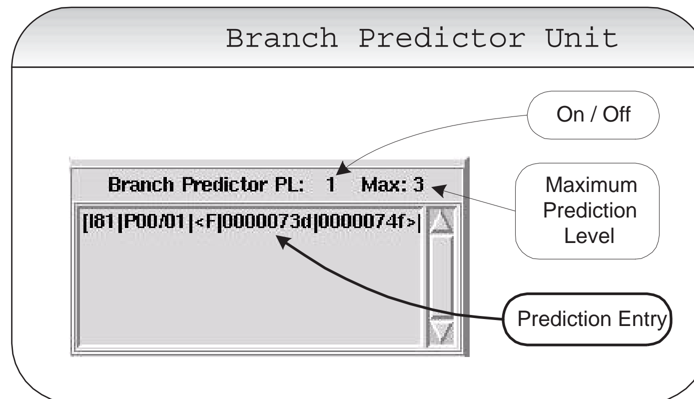


Figure B.13: Branch Predictor Unit GUI

Field	Format	Description
On/Off	*0* or *1*	Whether the predictor is on (**1** or off (**0**)
Maximum	N	N is the maximum allowed prediction levels.
Prediction Entry	$[OP\|PLO/PLN]$ $<TF\|\text{TAddr}\|\text{FAddr}>$	***OP*** Opcode of the branch instruction. ***PLO*** Old Prediction Level, i.e. before speculating on this new branch. ***PLN*** New Prediction Level after speculating on this branch. ***TF*** Predicted outcome: **T**rue or **F**alse. ***TAddr*** PC value if the outcome is true. ***FAddr*** PC value if the outcome is false.

Execution Unit

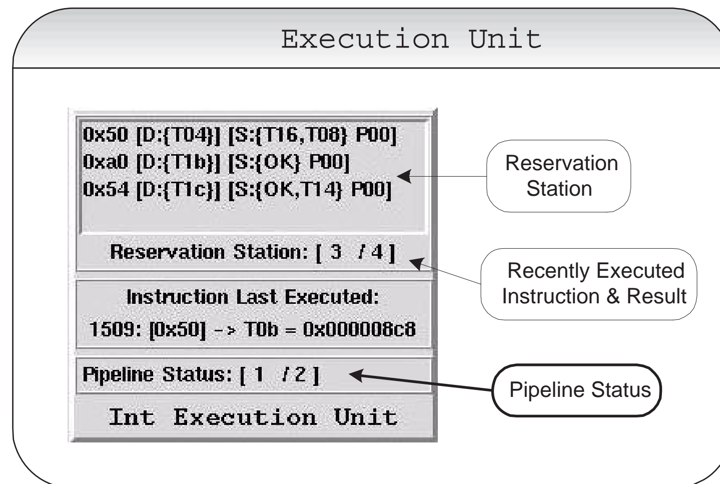


Figure B.14: Execution Unit GUI

Field	Format	Description
Reservation Station Entry	OP [D: {Opr}] [S:{Opr} PL]	***OP*** Opcode of the instruction. ***D*** Destination (output) of the instruction. ***S*** Source (input) of the instruction. ***Opr*** Operand Status: **OK** = value acquired, **T***Num* = tag number of unavailable value. ***PL*** Associated Prediction Level.

Memory Unit

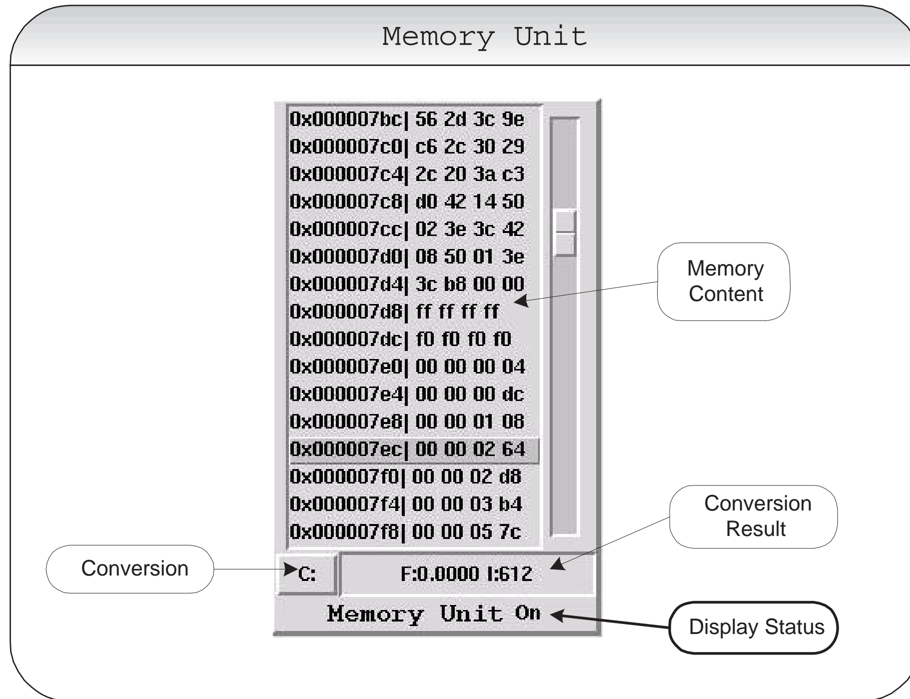


Figure B.15: Memory Unit GUI

Field	Format	Description
Memory Value	*Addr*| *B1 B2 B3 B4*	***Addr*** Address of the one memory word, in hexadecimal. **B*N*** Individual byte, in hexadecimal. A memory word can be selected via mouse click.

Conversion	-	Allow conversion to integer and floating point of the selected memory word.
Conversion Result	F:*FRes* I:*IRes*	***FRes*** Selected memory word in floating point. ***IRes*** Selected memory word in integer.

Appendix C

SAFA Benchmark Programs

C.1 Sieve of Erathosthense

```
PROC Sieve 2 3
    ibload 0
    cfb_wstore x2c
forLoop: cfb_wload x2c
    cfb_wload x28
    ige
    iftrue forLoopEnd
    cfb_wload x24
    cfb_wload x2c
    iadd
    ibload 1
    bstore
    cfb_wload x2c
    inc
    cfb_wstore x2c
    goto forLoop
forLoopEnd: ibload 2
    cfb_wstore x30
whileLoop: cfb_wload x30
    cfb_wload x28
    ige
    iftrue end
    cfb_wload x30
    ibload 2
    imul
    cfb_wstore x34
innerFor: cfb_wload x34
    cfb_wload x28
    ige
    iftrue innerForEnd
    cfb_wload x24
    cfb_wload x34
    iadd
    ibload 0
    bstore
    cfb_wload x34
    cfb_wload x30
    iadd
    cfb_wstore x34
    goto innerFor
innerForEnd: cfb_wload x30
    inc
    cfb_wstore x30
innerWhile: cfb_wload x30
    cfb_wload x28
    ige
    iftrue whileLoop
    cfb_wload x24
    cfb_wload x30
    iadd
    bload
    ifeq innerWhileEnd
```

```
    goto whileLoop
innerWhileEnd:
    cfb_wload x30
    inc
    cfb_wstore x30
    goto innerWhile
end: exit 1,2
```

```
PROC main 0 2
    ibload 100
    cfb_wstore x24
    cfb_wload x24
    ibload 1
    newarray
    pop2
    cfb_wstore x28
    cfb_wload x28
    cfb_wload x24
    penter 3,Sieve
    halt
```

C.2 Bubble Sort

```
PROC LCG 5 2
    ibload 1
    cfb_wstore x3c
    ibload 0
    cfb_wstore x38
loop: cfb_wload x38
    cfb_wload x28
    ige
    iftrue end
    cfb_wload x2c
    cfb_wload x3c
    imul
    cfb_wload x30
    iadd
    cfb_wload x34
    idiv
    cfb_wstore x3c
    pop
    cfb_wload x24
    cfb_wload x38
    ibload 4
    imul
    iadd
    cfb_wload x3c
    wstore
    cfb_wload x38
    inc
    cfb_wstore x38
    goto loop
end: exit 1,2
```

```
PROC BubbleSort 2 3
doLoop: ibload 0
    cfb_wstore x2c
    ibload 0
    cfb_wstore x30
forLoop: cfb_wload x30
    cfb_wload x28
    dec
    ige
    iftrue forEnd
    cfb_wload x24
    cfb_wload x30
    ibload 4
    imul
    iadd
    wload
    cfb_wload x24
    cfb_wload x30
    inc
    ibload 4
    imul
    iadd
    wload
    ile
    iftrue forUpdate
    cfb_wload x24
    cfb_wload x30
    ibload 4
    imul
    iadd
    wload
    cfb_wstore x34
    cfb_wload x24
    cfb_wload x30
    ibload 4
    imul
    iadd
    cfb_wload x24
    cfb_wload x30
    inc
    ibload 4
    imul
    iadd
    wload
    wstore
    cfb_wload x24
    cfb_wload x30
    inc
```

```
    ibload 4
    imul
    iadd
    cfb_wload x34
    wstore
    ibload 1
    cfb_wstore x2c
forUpdate: cfb_wload x30
    inc
    cfb_wstore x30
    goto forLoop
forEnd: cfb_wload x2c
    ifne doLoop
end: exit 1,2
```

```
PROC main 0 2
    ibload 50
    cfb_wstore x24
    cfb_wload x24
    ibload 4
    newarray
    pop2
    cfb_wstore x28
    cfb_wload x28
    cfb_wload x24
    ihwload 1277
    ibload 0
    iwload 131012
    penter 3,LCG
    cfb_wload x28
    cfb_wload x24
    penter 3,BubbleSort
    halt
```

C.3 Bubble Sort: Frame Register Version

```
PROC LCG 5 2
    ....
    Same As LCG in Bubble Sort
    ....
    ....
    ....
```

```
PROC BubbleSort 4 6
    SAVEFRM 4 x40
    ibload x24
    loadnextfrm
    cfset4
    cfinfostore
    cfsetown
doLoop:
    ibload 0
    cfb_wstore x34
    ibload 0
    cfset4
    idxstore
forLoop:
    idxlimitcmp
    inc
    ifge forEnd
    frload4
    cfincidx
    frload4
    ile
    iftrue forUpdate
    frload4
    cfdecidx
    frload4
    swap
    frstore4
    cfincidx
    frstore4
    cfsetown
    ibload 1
    cfb_wstore x34
forUpdate:
    cfset4
    goto forLoop
forEnd:
    cfsetown
    cfb_wload x34
    ifne doLoop
end:
    RESTOREFRM 4 x40
    exit 1,2
```

```
PROC main 0 1
    ibload 10
    cfb_wstore x24
    cfb_wload x24
    ibload 4
    newarray
    cfset4
    cfinfostore
    cfinfoload
    cfsetown
    cfb_wload x24
    ihwload 1277
    ibload 0
    iwload 131012
    penter 3,LCG
    cfset4
    cfinfoload
    cfsetown
    cfb_wload x24
    penter 3,BubbleSort
```

```
halt
```

C.4 Fibonacci Series

```
PROC fib 1 0
    cfb_wload x24
    ifne not0
    ibload 0
    exit 1,2
not0:
    cfb_wload x24
    ibload 1
    ineq
    iftrue not1
    ibload 1
    exit 1,2
not1:
    cfb_wload x24
    ibload 1
    isub
    penter 3,fib
    cfb_wload x24
    ibload 2
    isub
    penter 3,fib
    iadd
    exit 1,2
```

```
PROC main 0 3
   ibload 10
   penter 3,fib
   halt
```

C.5 Quick Sort

```
PROC LCG 5 2                                ....
    ....                                    ....
    Same As LCG in Bubble Sort              ....
```

```
PROC QuickSort 3 4
    cfb_wload x28
    cfb_wstore x30
    cfb_wload x2c
    cfb_wstore x34
    cfb_wload x24
    cfb_wload x28
    cfb_wload x2c
    iadd
    ibload 2
    idiv
    pop
    ibload 4
    imul
    iadd
    wload
    cfb_wstore x3c
outer:
    cfb_wload x30
    cfb_wload x34
    ige
    iftrue outerEnd
iLoop:
    cfb_wload x24
    cfb_wload x30
    ibload 4
    imul
    iadd
    wload
    cfb_wload x3c
    ige
    iftrue jLoop
    cfb_wload x30
    inc
    cfb_wstore x30
    goto iLoop
jLoop:
    cfb_wload x24
    cfb_wload x34
    ibload 4
    imul
    iadd
    wload
    cfb_wload x3c
    ile
    iftrue switch
    cfb_wload x34
    dec
    cfb_wstore x34
    goto jLoop
switch:
    cfb_wload x30
    cfb_wload x34
    igt
    iftrue outer
    cfb_wload x24
    cfb_wload x30
    ibload 4
    imul
    iadd
    wload
    cfb_wstore x38
    cfb_wload x24
    cfb_wload x30
    ibload 4
    imul
    iadd
    cfb_wload x24
    cfb_wload x34
    ibload 4
    imul
    iadd
```

```
    wload
    wstore
    cfb_wload x24
    cfb_wload x34
    ibload 4
    imul
    iadd
    cfb_wload x38
    wstore
    cfb_wload x30
    inc
    cfb_wstore x30
    cfb_wload x34
    dec
    cfb_wstore x34
    goto outer
outerEnd:
    cfb_wload x28
    cfb_wload x34
    ige
    iftrue secondHalf
    cfb_wload x24
    cfb_wload x28
    cfb_wload x34
    penter 3,QuickSort
secondHalf:
    cfb_wload x30
    cfb_wload x2c
    ige
    iftrue end
    cfb_wload x24
    cfb_wload x30
    cfb_wload x2c
    penter 3,QuickSort
end: exit 1,2
```

```
PROC main 0 2
    ibload 50
    cfb_wstore x24
    cfb_wload x24
    ibload 4
    newarray
    pop2
    cfb_wstore x28
    cfb_wload x28
    cfb_wload x24
    ihwload 1277
    ibload 0
    iwload 131012
    penter 3,LCG
    cfb_wload x28
    ibload 0
    cfb_wload x24
    dec
    penter 3,QuickSort
    halt
```

C.6 Student Array: Conventional Array Access

```
PROC main 0 7
    ibload 1
    cfb_wstore x28
    ibload 100
    cfb_wstore x2c
    ibload 200
    ibload 4
    newarray
    pop2
    cfb_wstore x34
    ibload 0
    cfb_wstore x24
init:
    cfb_wload x24
    cfb_wload x2c
    ige
    iftrue initEnd
    cfb_wload x24
    ibload 8
    imul
    cfb_wload x34
    iadd
    cfb_wload x24
    wstore
    ihwload 1277
    cfb_wload x28
    imul
    ibload 101
    idiv
    cfb_wstore x28
    pop
    cfb_wload x24
    ibload 8
    imul
    ibload 4
    iadd
    cfb_wload x34
    iadd
    cfb_wload x28
    wstore
    cfb_wload x24
    inc
    cfb_wstore x24
    goto init
initEnd:
    ibload 0
    cfb_wstore x30
    ibload 0
    cfb_wstore x24
sum:
    cfb_wload x24
    cfb_wload x2c
    ige
    iftrue sumEnd
    cfb_wload x30
    cfb_wload x24
    ibload 8
    imul
    ibload 4
    iadd
    cfb_wload x34
    iadd
    wload
    iadd
    cfb_wstore x30
    cfb_wload x24
    inc
    cfb_wstore x24
    goto sum
sumEnd:
    halt
```

C.7 Student Array: Frame Register and Index

```
PROC main 0 7
    ibload 1
    cfb_wstore x28
    ibload 100
    cfb_wstore x2c
    ibload 200
    ibload 4
    newarray
    cfset4
    cfinfostore
    ibload 0
    cfsetown
    cfb_wstore x24
init:
    cfb_wload x24
    cfb_wload x2c
    ige
    iftrue initEnd
    cfb_wload x24
    frstore4
    cfset4
    cfincidx
    ihwload 1277
    cfsetown
    cfb_wload x28
    imul
    ibload 101
    idiv
    cfb_wstore x28
    pop
    cfb_wload x28
    cfset4
    frstore4
    cfincidx
    cfsetown
    cfb_wload x24
    inc
    cfb_wstore x24
    goto init
initEnd:
    ibload 0
    cfb_wstore x30
    ibload 0
    cfb_wstore x24
    cfset4
    ibload 1
    idxstore
    ibload 1
    itvstore
    cfsetown
sum:
    cfb_wload x24
    cfb_wload x2c
    ige
    iftrue sumEnd
    cfb_wload x30
    frload4
    cfincidx
    iadd
    cfsetown
    cfb_wstore x30
    cfb_wload x24
    inc
    cfb_wstore x24
    goto sum
sumEnd:
    halt
```

C.8 Student Array: Frame Register and Offset

```
PROC main 0 10
    ibload 1
    cfb_wstore x28
    ibload 100
    cfb_wstore x2c
    ibload 200
    ibload 4
    newarray
    cfset4
    cfinfostore
    ibload 0
    cfsetown
    cfb_wstore x24
init:
    cfb_wload x24
    cfb_wload x2c
    ige
    iftrue initEnd
    cfb_wload x24
    frstore4
    cfset4
    cfincidx
    ihwload 1277
    cfsetown
    cfb_wload x28
    imul
    ibload 101
    idiv
    cfb_wstore x28
    pop
    cfb_wload x28
    cfset4
    frstore4
    cfincidx
    cfsetown
    cfb_wload x24
    inc
    cfb_wstore x24
    goto init
initEnd:
    ibload 0
    cfb_wstore x30
    ibload 0
    cfb_wstore x24
    cfset4
    cfinfoload
    swap
    pop
    iwload x20001
    swap
    cfset5
    cfinfostore
    cfsetown
sum:
    cfb_wload x24
    cfb_wload x2c
    ige
    iftrue sumEnd
    cfb_wload x30
    frload5
    iadd
    b_addbase 8
    cfsetown
    cfb_wstore x30
    cfb_wload x24
    inc
    cfb_wstore x24
    goto sum
sumEnd:
    halt
```

C.9 Student List: Conventional Linked List Traversal

```
PROC main 0 6
    ibload 1
    cfb_wstore x28
    ibload 100
    cfb_wstore x2c
    ibload 0
    cfb_wstore x24
    ibload 0
    cfb_wstore x34
init:
    cfb_wload x24
    cfb_wload x2c
    ige
    iftrue initEnd
    ibload 3
    ibload 4
    newarray
    pop2
    cfb_wstore x38
    cfb_wload x38
    cfb_wload x24
    wstore
    cfb_wload x28
    ihwload 1277
    imul
    ibload 101
    idiv
    cfb_wstore x28
    pop
    cfb_wload x38
    ibload 4
    iadd
    cfb_wload x28
    wstore
    cfb_wload x38
    ibload 8
    iadd
    cfb_wload x34
    wstore
    cfb_wload x38
    cfb_wstore x34
    cfb_wload x24
    inc
    cfb_wstore x24
    goto init
initEnd:
    ibload 0
    cfb_wstore x30
    cfb_wload x34
    cfb_wstore x38
sum:
    cfb_wload x38
    ifeq sumEnd
    cfb_wload x38
    ibload 4
    iadd
    wload
    cfb_wload x30
    iadd
    cfb_wstore x30
    cfb_wload x38
    ibload 8
    iadd
    wload
    cfb_wstore x38
    goto sum
sumEnd:
    halt
```

C.10 Student List: Frame Register and Index

```
PROC main 0 10
    ibload 1
    cfb_wstore x28
    ibload 100
    cfb_wstore x2c
    ibload 0
    cfb_wstore x24
    ibload 0
    cfset4
    baseloadidx
    cfsetown
init:
    cfb_wload x24
    cfb_wload x2c
    ige
    iftrue initEnd
    ibload 3
    ibload 4
    newarray
    cfset5
    cfinfostore
    cfsetown
    cfb_wload x24
    frstore5
    cfb_wload x28
    ihwload 1277
    imul
    ibload 101
    idiv
    cfb_wstore x28
    pop
    cfb_wload x28
    cfset5
    cfincidx
    frstore5
    cfset4
    cfinfoload
    pop2
    cfset5
    cfincidx
    frstore5
    cfinfoload
    cfset4
    cfinfostore
    cfsetown
    cfb_wload x24
    inc
    cfb_wstore x24
    goto init
initEnd:
    ibload 0
    cfb_wstore x30
    cfset4
    cfinfoload
    cfset5
    cfinfostore
sum:
    cfset5
    cfinfoload
    pop2
    ifeq sumEnd
    ibload 1
    idxstore
    frload5
    cfsetown
    cfb_wload x30
    iadd
    cfb_wstore x30
    cfset5
    cfincidx
    frload5
    baseloadidx
    goto sum
sumEnd:
    halt
```

C.11 Student List: Frame Register and Offset

```
PROC main 0 10
    ibload 1
    cfb_wstore x28
    ibload 100
    cfb_wstore x2c
    ibload 0
    cfb_wstore x24
    ibload 0
    cfset4
    baseloadidx
    cfsetown
init:
    cfb_wload x24
    cfb_wload x2c
    ige
    iftrue initEnd
    ibload 3
    ibload 4
    newarray
    cfset5
    cfinfostore
    cfsetown
    cfb_wload x24
    frstore5
    cfb_wload x28
    ihwload 1277
    imul
    ibload 101
    idiv
    cfb_wstore x28
    pop
    cfb_wload x28
    cfset5
    cfb_wstore x4
    cfset4
    cfinfoload
    pop2
    cfset5
    cfb_wstore x8
    cfinfoload
    cfset4
    cfinfostore
    cfsetown
    cfb_wload x24
    inc
    cfb_wstore x24
    goto init
initEnd:
    ibload 0
    cfb_wstore x30
    cfset4
    cfinfoload
    cfset5
    cfinfostore
sum:
    cfset5
    cfinfoload
    pop2
    ifeq sumEnd
    cfb_wload x4
    cfsetown
    cfb_wload x30
    iadd
    cfb_wstore x30
    cfset5
    cfb_wload x8
    baseloadidx
    goto sum
sumEnd:
    halt
```

C.12 Linpack Benchmark

```
PROC abs 1 0
    cfb_wload x24
    dup
    iwload <f0>
    flt
    iffalse return
    iwload <f-1>
    fmul
return: exit 1,2
```

```
PROC idamax 4 5
    ibload 0
    cfb_wstore x44
    cfb_wload x24
    dec
    ifge ge1
    iwload -1
    cfb_wstore x44
    wgoto end
ge1: cfb_wload x24
    dec
    ifne nne1
    ibload 0
    cfb_wstore x44
    wgoto end
nne1: cfb_wload x30
    ibload 1
    ieq
    hw_iftrue inc1
    cfb_wload x28
    cfb_wload x2c
    ibload 4
    imul
    iadd
    wload
    penter 3,abs
    cfb_wstore x34
    cfb_wload x30
    inc
    cfb_wstore x40
    ibload 1
    cfb_wstore x3c
l1: cfb_wload x3c
    cfb_wload x24
    ige
    hw_iftrue end
    cfb_wload x24
    cfb_wload x40
    cfb_wload x2c
    iadd
    ibload 4
    imul
    iadd
    wload
    penter 3,abs
    cfb_wstore x38
    cfb_wload x38
    cfb_wload x34
    fle
    iftrue small
    cfb_wload x3c
    cfb_wstore x44
    cfb_wload x38
    cfb_wstore x34
small:    cfb_wload x40
    cfb_wload x30
    iadd
    cfb_wstore x40
    cfb_wload x3c
    inc
    cfb_wstore x3c
    goto l1
inc1: ibload 0
    cfb_wstore x44
    cfb_wload x28
    cfb_wload x2c
    ibload 4
    imul
```

```
    iadd
    wload
    penter 3,abs
    cfb_wstore x34
    ibload 1
    cfb_wstore x3c
l2: cfb_wload x3c
    cfb_wload x24
    ige
    iftrue end
    cfb_wload x28
    cfb_wload x3c
    cfb_wload x2c
    iadd
    ibload 4
    imul
    iadd
    wload
    penter 3,abs
    cfb_wstore x38
    cfb_wload x38
    cfb_wload x34
    fle
    iftrue small2
    cfb_wload x3c
    cfb_wstore x44
    cfb_wload x38
    cfb_wstore x34
small2: cfb_wload x3c
    inc
    cfb_wstore x3c
    goto l2
end: cfb_wload x44
    exit 1,2
```

```
PROC dscal 5 2
    cfb_wload x24
    ifle end
    cfb_wload x34
    dec
    ifeq inc1
    cfb_wload x24
    cfb_wload x34
    imul
    cfb_wstore x3c
    ibload 0
    cfb_wstore x38
l1: cfb_wload x38
    cfb_wload x3c
    ige
    iftrue end
    cfb_wload x2c
    cfb_wload x38
    cfb_wload x30
    iadd
    ibload 4
    imul
    iadd
    dup
    wload
    cfb_wload x28
    fmul
    wstore
    cfb_wload x38
    cfb_wload x34
    iadd
    cfb_wstore x38
    goto l1
inc1: ibload 0
    cfb_wstore x38
l2: cfb_wload x38
    cfb_wload x24
    ige
    iftrue end
    cfb_wload x2c
    cfb_wload x38
    cfb_wload x30
    iadd
    ibload 4
    imul
    iadd
    dup
    wload
    cfb_wload x28
    fmul
```

```
    wstore
    cfb_wload x38
    inc
    cfb_wstore x38
    goto l2
end: exit 1,2
```

```
PROC daxpy 8 3
    cfb_wload x24
    ibload 0
    ile
    hw_iftrue end
    cfb_wload x28
    iwload <f0>
    feq
    hw_iftrue end
    cfb_wload x34
    dec
    ifne not1
    cfb_wload x40
    dec
    ifeq both1
not1: ibload 0
    cfb_wstore x48
    ibload 0
    cfb_wstore x4c
    cfb_wload x34
    ifge cy
    cfb_wload x24
    ineg
    inc
    cfb_wload x34
    imul
    cfb_wstore x48
cy: cfb_wload x40
    ifge initl1
    cfb_wload x24
    ineg
    inc
    cfb_wload x40
    imul
    cfb_wstore x4c
initl1: ibload 0
    cfb_wstore x44
l1: cfb_wload x44
    cfb_wload x24
    ige
    iftrue midend
    cfb_wload x38
    cfb_wload x4c
    cfb_wload x3c
    iadd
    ibload 4
    imul
    iadd
    dup
    wload
    cfb_wload x28
    cfb_wload x2c
    cfb_wload x48
    cfb_wload x30
    iadd
    ibload 4
    imul
    iadd
    wload
    fmul
    fadd
    wstore
    cfb_wstore x48
    cfb_wload x34
    iadd
    cfb_wstore x48
    cfb_wload x4c
    cfb_wload x40
    iadd
    cfb_wstore x4c
    cfb_wload x44
    inc
    cfb_wstore x44
    goto l1
midend: exit 1,2
both1: ibload 0
    cfb_wstore x44
l2: cfb_wload x44
```

```
    cfb_wload x24
    ige
    iftrue end
    cfb_wload x38
    cfb_wload x44
    cfb_wload x3c
    iadd
    ibload 4
    imul
    iadd
    dup
    wload
    cfb_wload x28
    cfb_wload x2c
    cfb_wload x44
    cfb_wload x30
    iadd
    ibload 4
    imul
    iadd
    wload
    fmul
    fadd
    wstore
    cfb_wload x44
    inc
    cfb_wstore x44
    goto l2
end: exit 1,2
```

```
PROC dgefa 3 9
    ibload 0
    cfb_wstore x50
    cfb_wload x28
    dec
    cfb_wstore x4c
    cfb_wload x4c
    ibload 0
    ilt
    hw_iftrue end
    ibload 0
    cfb_wstore x40
l1: cfb_wload x40
    cfb_wload x4c
    ige
    hw_iftrue end
    cfb_wload x24
    cfb_wload x40
    ibload 4
    imul
    iadd
    wload
    cfb_wstore x30
    cfb_wload x40
    inc
    cfb_wstore x44
    cfb_wload x28
    cfb_wload x40
    isub
    cfb_wload x30
    cfb_wload x40
    ibload 1
    penter 3,idamax
    cfb_wload x40
    iadd
    cfb_wstore x48
    cfb_wload x2c
    cfb_wload x40
    ibload 4
    imul
    iadd
    cfb_wload x48
    wstore
    cfb_wload x30
    cfb_wload x48
    ibload 4
    imul
    iadd
    wload
    iwload <f0>
    feq
    hw_iftrue loopUpdate
    cfb_wload x48
    cfb_wload x40
```

```
    ieq
    iftrue noSwitch
    cfb_wload x30
    cfb_wload x48
    ibload 4
    imul
    iadd
    wload
    cfb_wstore x38
    cfb_wload x30
    cfb_wload x48
    ibload 4
    imul
    iadd
    cfb_wload x30
    cfb_wload x40
    ibload 4
    imul
    iadd
    wload
    wstore
    cfb_wload x30
    cfb_wload x40
    ibload 4
    imul
    iadd
    cfb_wload x38
    wstore
noSwitch: iwload <f-1.0>
    cfb_wload x30
    cfb_wload x40
    ibload 4
    imul
    iadd
    wload
    fdiv
    cfb_wstore x38
    cfb_wload x28
    cfb_wload x44
    isub
    cfb_wload x38
    cfb_wload x30
    cfb_wload x44
    ibload 1
    penter 3,dscal
    cfb_wload x44
    cfb_wstore x3c
inner: cfb_wload x3c
    cfb_wload x28
    ige
    hw_iftrue loopEnd
    cfb_wload x24
    cfb_wload x3c
    ibload 4
    imul
    iadd
    wload
    cfb_wstore x34
    cfb_wload x34
    cfb_wload x48
    ibload 4
    imul
    iadd
    wload
    cfb_wstore x38
    cfb_wload x48
    cfb_wload x40
    ieq
    iftrue noColSwitch
    cfb_wload x34
    cfb_wload x48
    ibload 4
    imul
    iadd
    cfb_wload x34
    cfb_wload x40
    ibload 4
    imul
    iadd
    wload
    wstore
    cfb_wload x34
    cfb_wload x40
    ibload 4
    imul
    iadd
    cfb_wload x38
    wstore
noColSwitch: cfb_wload x28
    cfb_wload x44
    isub
    cfb_wload x38
    cfb_wload x30
    cfb_wload x44
```

```
    ibload 1
    cfb_wload x34
    cfb_wload x44
    ibload 1
    penter 3,daxpy
    cfb_wload x3c
    inc
    cfb_wstore x3c
    wgoto inner
loopUpdate: cfb_wload x40
    cfb_wstore x50
loopEnd: cfb_wload x40
    inc
    cfb_wstore x40
    wgoto l1
end: cfb_wload x2c
    cfb_wload x28
    dec
    ibload 4
    imul
    iadd
    cfb_wload x28
    dec
    wstore
    exit 1,2
```

```
PROC dgesl 4 6
    cfb_wload x28
    dec
    cfb_wstore x44
    cfb_wload x44
    ibload 1
    ilt
    hw_iftrue secondPart
    ibload 0
    cfb_wstore x38
l1: cfb_wload x38
    cfb_wload x44
    ige
    hw_iftrue secondPart
    cfb_wload x2c
    cfb_wload x38
    ibload 4
    imul
    iadd
    wload
    cfb_wstore x40
    cfb_wload x30
    cfb_wload x40
    ibload 4
    imul
    iadd
    wload
    cfb_wstore x34
    cfb_wload x40
    cfb_wload x38
    ieq
    iftrue noSwitch
    cfb_wload x30
    cfb_wload x40
    ibload 4
    imul
    iadd
    cfb_wload x30
    cfb_wload x38
    ibload 4
    imul
    iadd
    wload
    wstore
    cfb_wload x30
    cfb_wload x38
    ibload 4
    imul
    iadd
    cfb_wload x34
    wstore
noSwitch: cfb_wload x38
    inc
    cfb_wstore x48
    cfb_wload x28
    cfb_wload x48
    isub
    cfb_wload x34
```

```
    cfb_wload x24
    cfb_wload x38
    ibload 4
    imul
    iadd
    wload
    cfb_wload x48
    ibload 1
    cfb_wload x30
    cfb_wload x48
    ibload 1
    penter 3,daxpy
    cfb_wload x38
    inc
    cfb_wstore x38
    wgoto l1
secondPart: ibload 0
    cfb_wstore x3c
l2: cfb_wload x3c
    cfb_wload x28
    ige
    hw_iftrue end
    cfb_wload x28
    cfb_wload x3c
    inc
    isub
    cfb_wstore x38
    cfb_wload x30
    cfb_wload x38
    ibload 4
    imul
    iadd
    dup
    wload
    cfb_wload x24
    cfb_wload x38
    ibload 4
    imul
    iadd
    wload
    cfb_wload x38
    ibload 4
    imul
    iadd
    wload
    fdiv
    wstore
    cfb_wload x30
    cfb_wload x38
    ibload 4
    imul
    iadd
    wload
    iwload <f-1>
    fmul
    cfb_wstore x34
    cfb_wload x38
    cfb_wload x34
    cfb_wload x24
    cfb_wload x38
    ibload 4
    imul
    iadd
    wload
    ibload 0
    ibload 1
    cfb_wload x30
    ibload 0
    ibload 1
    penter 3,daxpy
    cfb_wload x3c
    inc
    cfb_wstore x3c
    wgoto l2
end: exit 1,2
```

```
PROC matgen 3 4
    ihwload 1325
    cfb_wstore x34
    iwload <f0>
    cfb_wstore x30
    ibload 0
    cfb_wstore x38
nl: cfb_wload x38
```

```
    cfb_wload x28
    ige
    iftrue loopb
    ibload 0
    cfb_wstore x3c
innerCheck:    cfb_wload x3c
    cfb_wload x28
    ige
    iftrue innerEnd
    ihwload 3125
    cfb_wload x34
    imul
    iwload 65536
    idiv
    cfb_wstore x34
    pop
    cfb_wload x24
    cfb_wload x3c
    ibload 4
    imul
    iadd
    wload
    cfb_wload x38
    ibload 4
    imul
    iadd
    cfb_wload x34
    i2f
    iwload <f32768.0>
    fsub
    iwload <f16384.0>
    fdiv
    wstore
    cfb_wload x24
    cfb_wload x3c
    ibload 4
    imul
    iadd
    wload
    cfb_wload x38
    ibload 4
    imul
    iadd
    wload
    cfb_wload x30
    fle
    iftrue iU1
    cfb_wload x24
    cfb_wload x3c
    ibload 4
    imul
    iadd
    wload
    cfb_wload x38
    ibload 4
    imul
    iadd
    wload
    goto iU2
iU1: cfb_wload x30
iU2: cfb_wstore x30
    cfb_wload x3c
    inc
    cfb_wstore x3c
    goto innerCheck
innerEnd: cfb_wload x38
    inc
    cfb_wstore x38
    goto nl
loopb: ibload 0
    cfb_wstore x38
lbstart:    cfb_wload x38
    cfb_wload x28
    ige
    iftrue loopc
    cfb_wload x2c
    cfb_wload x38
    ibload 4
    imul
    iadd
    iwload <f0>
    wstore
    cfb_wload x38
    inc
    cfb_wstore x38
    goto lbstart
loopc: ibload 0
    cfb_wstore x3c
lcOuter: cfb_wload x3c
    cfb_wload x28
    ige
    iftrue end
    ibload 0
    cfb_wstore x38
```

```
lcInner: cfb_wload x38
    cfb_wload x28
    ige
    iftrue lcInnerEnd
    cfb_wload x2c
    cfb_wload x38
    ibload 4
    imul
    iadd
    dup
    wload
    cfb_wload x24
    cfb_wload x3c
    ibload 4
    imul
    iadd
    wload
    cfb_wload x38
    ibload 4
    imul
    iadd
    wload
    fadd
    wstore
    cfb_wload x38
    inc
    cfb_wstore x38
    goto lcInner
lcInnerEnd: cfb_wload x3c
    inc
    cfb_wstore x3c
    goto lcOuter
end: cfb_wload x30
    exit 1,2
```

```
PROC main 0 7
    ibload 5
    cfb_wstore x24
    cfb_wload x24
    ibload 4
    newarray
    pop2
    cfb_wstore x2c
    cfb_wload x24
    ibload 4
    newarray
    pop2
    cfb_wstore x30
    cfb_wload x24
    ibload 4
    newarray
    pop2
    cfb_wstore x28
    ibload 0
    cfb_wstore x34
make: cfb_wload x34
    cfb_wload x24
    ige
    iftrue makeEnd
    cfb_wload x28
    cfb_wload x34
    ibload 4
    imul
    iadd
    cfb_wload x24
    ibload 4
    newarray
    pop2
    wstore
    cfb_wload x34
    inc
    cfb_wstore x34
    goto make
makeEnd: cfb_wload x28
    cfb_wload x24
    cfb_wload x2c
    penter 3,matgen
    cfb_wstore x38
    cfb_wload x28
    cfb_wload x24
    cfb_wload x30
    penter 3,dgefa
    cfb_wload x28
    cfb_wload x24
    cfb_wload x30
```

```
cfb_wload x2c
penter 3,dgesl
halt
```