EXPLOITING CONCURRENCY IN A GENERAL-PURPOSE ONE-INSTRUCTION COMPUTER ARCHITECTURE

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2010

By
Christopher Daniel Emmons
School of Computer Science

Contents

Abstract 14

Declaration 15

Copyright 16

Acknowledgements 17

The author 18

I Introduction and background 19

1 Introduction 20

1.1 Motivation . . . 20

1.2 Research objectives . . . 21

1.3 Research contributions . . . 23

1.4 Thesis structure . . . 24

2 General-purpose computer architecture 25

2.1 General-purpose computer requirements . . . 25

2.2 Challenges in computer architecture . . . 26

2.2.1 Design complexity and productivity gap . . . 26

2.2.2 Power consumption . . . 27

2.2.3 Processor and memory performance gap . . . 28

2.2.4 Cost of communication . . . 29

2.3 Research directions . . . 30

2.3.1 Exploiting parallelism . . . 30

2.3.2 Increasing cache sizes . . . 35

2.3.3 Designing at higher levels of abstraction 35
2.3.4 Communication-centric architectures 36
2.4 Conclusion 37

3 The Fleet architecture 38
3.1 Architecture overview 38
3.2 Instruction set architecture 39
3.2.1 Native hardware data types 39
3.2.2 One instruction 40
3.2.3 Code bags 43
3.2.4 A simple Fleet program 44
3.3 Concurrency 45
3.4 Hardware organization 45
3.4.1 Instruction fetch and dispatch 45
3.4.2 Switch fabric 46
3.4.3 Ships 46
3.5 Early findings 46
3.6 Limitations 47
3.7 Conclusion 47

II The Armada architecture 49

4 The Armada architecture 50
4.1 Differences from Fleet 50
4.2 Architecture enhancements 53
4.2.1 Independent code bags 53
4.2.2 Enhanced local register file 54
4.2.3 Context synchronizers 55
4.2.4 Flow caching 55
4.3 Instruction set architecture 56
4.3.1 Memory model 56
4.3.2 One instruction 56
4.3.3 Choosing the code bag type 58
4.3.4 Fetching code bags 60
4.3.5 Freeing Fleet cores 66

4.3.6 Virtual pipelines . . . 66
4.3.7 Handling state in Ships . . . 68
4.3.8 Context synchronizers . . . 69
4.3.9 Flow caching . . . 70
4.3.10 Hardware reset behavior . . . 72
4.3.11 Event handling . . . 73
4.3.12 Fleet prototype core ISA . . . 73
4.4 Related work . . . 74
4.4.1 Transport-triggered architectures . . . 74
4.4.2 Dataflow machines . . . 74
4.4.3 WaveScalar architecture . . . 75
4.4.4 Independence architectures . . . 75
4.4.5 SCALP and Vortex asynchronous processors . . . 76
4.4.6 TRIPS architecture . . . 76
4.4.7 Fleet at Sun Microsystems and U.C. Berkeley . . . 77
4.5 Conclusion . . . 77

5 The Armada-1 microarchitecture 78
5.1 Overview . . . 78
5.2 Memory subsystem . . . 78
5.3 Fleet cores . . . 79
5.3.1 Simultaneous multithreading . . . 79
5.3.2 Data size . . . 80
5.3.3 Instruction horn . . . 81
5.3.4 Instruction pool and reservation stations . . . 81
5.3.5 Switch fabric . . . 82
5.3.6 Ships . . . 83
5.4 Fetch and dispatch unit . . . 84
5.4.1 Code bag fetch unit . . . 84
5.4.2 Instruction dispatch unit . . . 87
5.5 Context synchronizers . . . 88
5.6 Communication . . . 88
5.7 Conclusion . . . 88

6 Evaluation of Armada-1 90
6.1 ArmadaSim . . . 90

6.1.1 Structural configuration 90
6.1.2 Timing model 91
6.1.3 Statistics gathering 91
6.1.4 Fleet state checker 91
6.1.5 Value-change dump output 92
6.1.6 Verification 92
6.1.7 Design complexity 93
6.1.8 Limitations 94
6.2 Mandelbrot 94
6.2.1 Computation 94
6.2.2 Parallelism 95
6.2.3 Mapping to Armada 98
6.3 Conclusion 101

7 Microarchitecture results and discussion 102
7.1 Performance overview 102
7.2 Exploiting concurrency 106
7.2.1 Instruction-level concurrency 106
7.2.2 Thread-level concurrency 108
7.3 Code density 108
7.4 Code bag organization 109
7.5 Asynchronous system performance 110
7.6 Resource multiplexing 111
7.7 Conclusion 111

III An Armada compiler 113

8 Armada Procedure Call Standard 114
8.1 Overview 114
8.2 Register file 115
8.2.1 Behavior 115
8.2.2 Listing 115
8.3 Stack 116
8.3.1 Allocation policy 116
8.3.2 Choosing the allocation policy 117

8.4 Data placement 118

8.4.1 Argument passing 118

8.4.2 Automatic variables 120

8.4.3 Return values 121

8.5 Procedure calls 121

8.5.1 Serial procedure calls 121

8.5.2 Concurrent procedure calls 122

8.6 Conclusion 126

9 Compiling for Armada 127

9.1 Modern optimizing compilers 127

9.2 LLVM IR 128

9.2.1 Basic blocks 129

9.2.2 Instructions 129

9.3 Armada code generator 132

9.3.1 Data types 132

9.3.2 Pre-processing 133

9.3.3 Code bag formation 135

9.3.4 Ship type and instruction selection 137

9.3.5 Instruction merging 137

9.3.6 Instruction splitting 138

9.3.7 Stray token cleanup 138

9.3.8 Ship allocation 140

9.4 Assembler 141

9.5 Limitations 141

9.6 Conclusion 142

10 ArmadaSim debugger 143

10.1 Trace replay 143

10.2 Message view 143

10.3 Source code view 144

10.4 State tracking view 145

10.5 Limitations 147

10.6 Conclusion 148

11 Testing the Armada compiler **149**
11.1 Objectives 149
11.2 Method and metrics 149
11.2.1 Code structure 150
11.2.2 Code density 150
11.2.3 Instructions fetched 150
11.2.4 Instructions executed 151
11.2.5 Instruction traffic 151
11.2.6 Concurrency 151
11.3 Results 152
11.3.1 Performance 152
11.3.2 Code structure 153
11.3.3 Code density 153
11.3.4 Instructions fetched and instruction traffic 155
11.3.5 Instructions executed 155
11.3.6 Concurrency 156
11.4 Conclusion 157

IV Discussion and conclusion **158**

12 Discussion and conclusion **159**
12.1 Fleet changes 160
12.2 Architecture enhancements 160
12.2.1 Proposed extensions and motivating factors 160
12.2.2 Conclusions 161
12.3 Compiler and tools 162
12.4 Limitations of described work 163
12.4.1 Few benchmarks 163
12.4.2 Performance comparison to modern architectures 163
12.4.3 Compiler maturity 163
12.5 Areas for future work 163
12.5.1 Memory subsystem 164
12.5.2 Higher granularity Ship functionality 164
12.5.3 Armada as an application-specific processor architecture . 164
12.5.4 Flow-control fabric 164

12.5.5 Flow caching 165
12.5.6 Compiler enhancements 165
12.6 Final remarks 165

Bibliography **166**

A Ships in Armada-1 **173**
A.1 Register 173
A.2 IntAdd64 173
A.3 IntDiv64 174
A.4 IntMul64 174
A.5 FPAdd64 174
A.6 FPDiv64 175
A.7 FPMul64 175
A.8 BitShift 175
A.9 BitOp 176
A.10 IntToFP 176
A.11 SExt 177
A.12 Counter 177
A.13 Stride 177
A.14 Cmp 178
A.15 Join 179
A.16 Selector 179
A.17 Toggle 179
A.18 Fetch 180
A.19 Memory 180
A.20 ContextSynchronizer 181
A.21 Stack 181

B Hardware stack allocation support **183**
B.1 Overview 183
B.2 Priming the ring 184
B.3 Stack allocation 185
B.4 Stack release 185
B.5 Testing 185

C Code listings **186**
C.1 Mandelbrot, hand-coded 186
C.2 Mandelbrot, LLVM-generated 189

D LLVM instruction mapping to Armada **196**
D.1 Terminator class 196
D.1.1 `ret` 196
D.1.2 `br` 196
D.1.3 `switch` 197
D.1.4 `invoke` 197
D.1.5 `unwind` 197
D.1.6 `unreachable` 197
D.2 Binary class 197
D.2.1 `add` 197
D.2.2 `sub` 197
D.2.3 `mul` 198
D.2.4 `udiv` 198
D.2.5 `sdiv` 198
D.2.6 `fdiv` 198
D.2.7 `urem` 199
D.2.8 `srem` 199
D.2.9 `frem` 199
D.3 Bitwise binary class 199
D.3.1 `shl` 199
D.3.2 `lshr` 200
D.3.3 `ashr` 200
D.3.4 `and` 200
D.3.5 `or` 200
D.3.6 `xor` 201
D.4 Vector class 201
D.4.1 `extractelement` 201
D.4.2 `insertelement` 201
D.4.3 `shufflevector` 201
D.5 Memory class 201
D.5.1 `malloc` 201
D.5.2 `free` 201

D.5.3 `alloca` 202
D.5.4 `load` 202
D.5.5 `store` 202
D.5.6 `getelementptr` 202
D.6 Conversion class 203
D.6.1 `trunc . . to` 203
D.6.2 `sext. . . to` 203
D.6.3 `zext. . . to` 203
D.6.4 `fptrunc. . . to` 203
D.6.5 `fpext. . . to` 203
D.6.6 `fptoui. . . to` 203
D.6.7 `fptosi. . . to` 203
D.6.8 `uitofp. . . to` 204
D.6.9 `sitofp. . . to` 204
D.6.10 `ptrtoint. . . to` 204
D.6.11 `inttoptr. . . to` 204
D.6.12 `bitcast. . . to` 204
D.7 Miscellaneous class 204
D.7.1 `icmp` 204
D.7.2 `fcmp` 205
D.7.3 `phi` 205
D.7.4 `select` 205
D.7.5 `call` 205
D.7.6 `va_arg` 205
D.7.7 `getresult` 205

List of Tables

3.1 Fleet native data types 39
3.2 Comparison of Fleet and RISC instructions 40
3.3 Move instruction syntax and graphical representation 41

4.1 Code bag comparison 59
4.2 Code bag descriptor syntax and representation 61

7.1 Code size and instruction fetch traffic comparison 109

8.1 Armada-1 register file listing 116

9.1 LLVM instruction classes 130

11.1 Hand-coded versus compiled Mandelbrot 152
11.2 Mandelbrot comparison on various ISA's 155

A.1 Cmp Ship in_op input port values 178

List of Figures

2.1 Design complexity and productivity gap. 27
2.2 Memory and CPU performance gap. 29
2.3 Cache size versus performance in Itanium2 35

3.1 Sum of squares Fleet code snippet. 44
3.2 Fleet architecture organization 45

4.1 Gating fetch of a successor code bag on token cleanup 52
4.2 Move instruction encoding overview. 57
4.3 Code bag descriptor format 60
4.4 Register forwarding example 64
4.5 Painting a horizontal line example 65
4.6 Virtual pipeline example 67
4.7 Color conversion cacheable flow. 72
4.8 Color conversion cleanup code. 73

5.1 Block diagram of a dual-core Armada-1 processor 79
5.2 Instruction horn and pool 81
5.3 Switch fabric 83
5.4 Fetch and dispatch unit 85

6.1 Mandelbrot benchmark inner loops in Java 96
6.2 Mandelbrot inner loop operations 97
6.3 Mandelbrot code bags 98
6.4 Mandelbrot code bag fetch scheme 99
6.5 `InnerLoop` control flow 100

7.1 Overview of Mandelbrot performance on Armada-1 103
7.2 Resource utilization of performance-limiting units 104

7.3 Mandelbrot performance on optimized Armada-1 105
7.4 Concurrently active Ships in Armada-1, Mandelbrot 107
7.5 Effect of code bag instruction arrangement on performance 110

8.1 Argument passing example with native and aggregate types. . . . 119
8.2 Stack layout for a serial procedure call 123
8.3 Stack layout for a concurrent procedure call 125

9.1 Modern optimizing compiler architecture 128
9.2 Control flow graph of a simple LLVM loop 131
9.3 LLVM add and store instructions. 131
9.4 Translation of an LLVM branch to Armada move instructions . . 137

10.1 ArmadaSim Debugger messages view 144
10.2 Debugging multiple threads 145
10.3 Stepping through a program 146
10.4 Tracking Fleet state 147

11.1 Hand-coded and compiled Mandelbrot 154
11.2 ILC comparison between hand-coded and compiled Mandelbrot . 156

B.1 Hardware stack allocation scheme 184

Abstract

The University of Manchester
Christopher Daniel Emmons
Doctor of Philosophy
Exploiting Concurrency in a General-Purpose One-Instruction Computer Architecture
December 2009

Computer performance is suffering greatly from diminishing returns as the increasing cost of implementing complex hardware optimizations and of increasing clock frequency no longer yields the gains in computational ability and power efficiency consumers demand. Notable products including a generation of Intel Pentium 4 processors have been cancelled as a result. This sudden hiccup in an historically predictable performance road map has inspired research and industrial communities to investigate architectures, some rather unorthodox, that complete work more quickly and more efficiently.

One such computer architecture under development, Fleet, exposes fine-grain instruction level concurrency, addresses the growing costs of on-chip communications, and promotes simplicity in the underlying hardware design. This one-instruction computer transports data using simple *move* operations. The globally-asynchronous architecture promotes high modularity allowing specialized configurations of the architecture to be generated quickly with low hardware and software complexity.

The Armada architecture presented in this thesis expands on Fleet by introducing constructs that exploit thread-level concurrency. The proposals herein aim to increase the performance efficiency of Fleet and other communication-centric architectures. Trade-offs between software and hardware complexity and between the static and dynamic division of labor are investigated through the implementation and study of an Armada microarchitecture and an Armada compiler created for this research. This thesis explores the merits and pitfalls of this unique architecture as the basis for general-purpose computers for the future.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor Prof. Steve Furber for providing me with an opportunity to study in Manchester, for being an ever-insightful sounding board, and for imparting expert direction and advice throughout the course of my research. Ivan Sutherland clearly motivated this work with his wacky Fleet architecture. I would like to thank him, Igor Benko, Wesley Clark, and Sun Microsystems for supporting my efforts with an internship in California and with good discourse that helped me find focus and refine my ideas. Thank you family and friends who have provided the best of support and encouragement while being a steady reminder that there is a world outside of the Ph.D. – which can be quite easy to forget. Finally, this work would have certainly been left uncompleted without the surprise reemergence of an old friend. Thank you, Erin, for knowing exactly what I needed when I needed it and for pushing me through the final months.

The author

Christopher Emmons obtained a Bachelor of Science degree in Computer Sciences from the University of Texas at Austin in 2003. While studying at Texas, he worked for Intel Corporation's Cellular and Handheld Group on XScale processors as an Applications Engineer intern. After graduating, he assumed Performance Validation Engineer and, subsequently, System-on-Chip Microarchitect roles at Intel. Leaving Intel in September 2005, Christopher began this Ph.D. work at the University of Manchester. He held two internship positions during the course of his research at ARM Holdings in Austin, Texas, and at Sun Labs in Menlo Park, California. He is currently working for ARM in Austin, Texas, in the Research and Development Group.

Part I

Introduction and background

Chapter 1

Introduction

In 2004, the means for increasing computer processor performance for over a decade reached an inflection point. Facing power inefficiency, difficulties designing increasingly complex hardware, growing costs of on-chip communication, and other barriers, the computer industry shifted focus to new ways of achieving more computing power more efficiently. This thesis presents one such computer architecture that attempts to overcome the contemporary performance, power, and design constraints that are inhibiting the progress of more typical computer processor designs. This architecture, called Armada, aims to create a power-efficient and reconfigurable computer useful for general purpose computing.

1.1 Motivation

For the past two decades, we have enjoyed large and predictable increases in computer performance. This improvement has largely been sustained by increasing utilization of hardware components as the processor operating frequency is increased. Intel unintentionally affirmed this trend is at an end by its highly publicized cancellation of a line of 4+ gigahertz processors in 2004 due to difficulty controlling the heat generated as an unwelcomed byproduct of such fast clock rates.

Designers are also having an increasingly difficult time putting additional transistors on a chip, made available each year by improvements in fabrication technology, to work. The density of transistors on a single chip roughly doubles every two years. This trend has proven very consistent over time and is commonly referred to as Moore's Law[1] [Moo65]. However, a more recent observation indicates that these additional resources are being used less and less efficiently; growth in performance of new processors is roughly the square root of their growth in density on any given chip process technology [Gel01]. The International Technology Roadmap for Semiconductors (ITRS) views this productivity gap as a great challenge for processor development over the coming years [ITR05].

Additionally, logic is becoming less expensive in terms of area and power consumption while communication is increasingly costly. The majority of processors have been designed to reuse once expensive logic as efficiently as possible. This reuse often requires moving large amounts of data through a minimal set of computation units. Although these computational units reside within close proximity to each other, controls spanning across the chip are greatly affected by the increasing wire delays that result from shrinking processor feature sizes [HMH01, AHKB00].

1.2 Research objectives

The overall goal of the research undertaken is to demonstrate a novel computer architecture with high power efficiency that addresses the modern challenges in processor design and manufacture. Computer architectures provide the framework on which processors are built and programmed. Architectures describe the programmer's view of the hardware – the *instruction set architecture* (ISA). They also describe the *microarchitecture* or *hardware organization* of the processor: what high-level blocks the hardware is composed of, what purposes they serve, and how they are interfaced together. Finally, computer architecture also encompasses the low-level implementation of the microarchitecture aptly referred to as the *hardware implementation*. Two or more implementations of the same microarchitecture may differ in characteristics such as size, performance, and clock frequency though they will share the same overall structure. This thesis will primarily focus on the ISA and microarchitectural aspects of computer architecture.

The soundness or goodness of a computer is largely judged on three factors:

[1]Gordon Moore's original observation was that the number of transistors on a die doubled every 24 months, and he projected that the trend would continue for at least the next ten years. That prediction was made nearly 45 years ago.

performance, power consumption, and cost. The computer must minimally provide enough functionality and performance to meet the demands of the target application and, preferably, leave room to spare for increasingly demanding tasks of the future. The performance provided must not come at the expense of unreasonable power consumption. Mobile devices often have the most strict power requirements while consumers of high performance computer servers increasingly consider the cost of ownership of the machine before making their purchases. This cost of ownership includes the energy requirements of the server and the external cooling required to keep it functioning reliably. Finally, the cost of the finished computer must fit within the budget of the target products' requirements and be price-competitive with alternative designs for the computer to be successful.

The Armada architecture presented in this thesis is based on a one-instruction computer architecture called Fleet currently under development by Sun Labs and a handful of students from the University of California, Berkeley. Fleet addresses new challenges in processor design and fabrication and also provides a unique instruction set architecture that frees programmers and the hardware from common sequential programming paradigms. Specifically, it supports the concurrent execution of many instructions and supplies software with a clean method of representing this concurrency. The power of this architecture appears to stem from its unique representation of programs and the implications this has on the partitioning of complexity both between hardware and software and between static compile-time and dynamic run-time responsibilities. One goal of this research is to understand how this new representation affects hardware and software synergy compared to other prominent architecture designs.

The Armada architecture extends Fleet by supporting a coarser granularity of concurrency. Whereas Fleet provides the software representation of instruction level concurrency and capability required to execute many instructions at one time, Armada additionally provides the representation of task or thread level concurrency and capability required to execute many contexts at the same time. The proposed microarchitecture contains multiple Fleet cores on a single chip. Furthermore, each Fleet core is given new capabilities to run multiple tasks at the same time. These hardware enhancements are supported by a small but significant addition to the ISA that enables the spawning and management of these tasks.

To explore the benefits and disadvantages of this representation, a model of an

Armada microarchitecture implementation and an Armada imperative language compiler were created enabling insight into both hardware and software aspects of the architecture. The hope is to use the simple concurrency constructs in the ISA and the modular, decentralized microarchitecture to cheaply exploit concurrency and to create a scalable general purpose architecture for the future.

1.3 Research contributions

The following contributions were produced from this research:

- Method of representing and spawning threads in Fleet-based architectures
- Method of graphically representing Fleet and Armada architecture assembly language programs
- Method of hardware virtualization for one-instruction, asynchronous transport architectures that:
  - enables simultaneous multithreading
  - enables software performance to scale with the increasing number of functional units that may be placed on a chip each year
  - enables increased resource utilization of processing elements in implementations with large numbers of such elements to decrease impact of rising static leakage power costs and to improve throughput
- An Armada microarchitecture: a multicore, globally asynchronous, simultaneous multithreading Fleet-based microarchitecture with thread level concurrency extensions
- Timing approximate simulation model of an Armada microarchitecture
- An assembler for Armada programs
- A trace-replay debugger for debugging highly concurrent Armada assembly language programs
- An Armada code generator for the Low Level Virtual Machine compiler infrastructure supporting high level imperative languages

1.4 Thesis structure

The thesis is divided into four parts. The remainder of part one describes the current challenges in computer architecture introduced here in more detail. Various types of general purpose computer architectures explored in the past and current research directions are discussed. Finally, an overview of the Fleet architecture upon which Armada is largely derived is given.

Part two discusses proposed enhancements to the Fleet architecture that form the Armada architecture. A microarchitecture implementation of Armada is presented, and a low-level simulator for the design is described. A discussion of how a benchmark was mapped to the architecture and some analysis of the microarchitecture's performance running this benchmark is provided.

Part three describes a procedure call standard that defines how software blocks are written for Armada so that they are interoperable with other software. A compiler code generator is described that transforms programs written in high-level imperative languages such as C++ and Java into instructions for the Armada architecture. Code generated for Armada is compared to code generated for other more typical architectures. Additionally, a tool for debugging the highly-concurrent architecture is described.

The final part of the report discusses the results, considers further work, and provides concluding remarks on this research.

Chapter 2

General-purpose computer architecture

This chapter provides an introduction to the area of general purpose computer architecture with a focus on major challenges of today and tomorrow. Some methods and research directions taken to confront these challenges are discussed. Exploiting concurrency is given special focus.

2.1 General-purpose computer requirements

Computer processors have widely varying applications from controlling kitchen appliances and children's toys to handling millions of secure transactions per minute in online store servers. Due to the enormous number of applications processors have, the design space for architectures is also large. This research focuses on the development and analysis of a new architecture to design general purpose computer processors.

General purpose processors may be found in powerful computer servers, home desktop and notebook computers, and in hand-held devices. Typical applications include gaming, media playback and editing, web browsing, data serving, and scientific workload processing. Successful processors will balance price-performance carefully; thus, semiconductor companies produce different flavors of general purpose processors to better meet the needs of the various target segments. For example, although one ISA may be used for all segments, a differentiation in microarchitecture for processors used in high-end desktop PC's and servers may be made from those processors used in mobile, power-constrained devices. Implementation differences such as clock frequency, chip fabrication technology, or cache sizes may be used to target more specific demands of applications and to satisfy a variety of customers with varying budgets while maximizing profits for the processor manufacturer.

General purpose architectures generally have long lifetimes because they serve as the rough blueprints for multiple generations of processors. Pre-compiled programs are expected to retain binary compatibility with these different processors. Therefore, programs should ideally benefit from any performance improvements made between each generation without having to recompile them.

An increasing number of applications in the traditionally general purpose computing design space have common and demanding tasks that they are expected to perform efficiently. For example, high-resolution video capture and compression on a desktop PC requires fast video encoding of a large amount of raw image data. Cryptography algorithms are used to securely transfer data and to enable digital rights management of protected media. The specialization of microprocessors for target applications such as these is enabled by reconfigurable architectures and by the addition of hardware accelerators for common and demanding tasks. As transistor budgets grow, processor manufacturers are able to integrate such reconfigurable logic and specialized accelerators into traditionally general purpose processors.

2.2 Challenges in computer architecture

Optimizing processors for performance, power, and cost requires a carefully balanced architecture which addresses numerous challenges. The following sections describe some of the contemporary challenges architects face.

2.2.1 Design complexity and productivity gap

Improving chip process technologies allow more transistors to be crammed onto processors with each generation. As foretold by Moore's Law, billion transistor chips have arrived; Intel's Itanium 2 processor in production in 2006, codenamed "Montecito," encompasses 1.72 billion transistors. Sun Microsystems' "Niagara" processor, shipping since late 2005, combines eight processing cores to tackle challenging applications. Managing such large numbers of resources effectively has become a formidable challenge.

Design complexity is a metric that refers to the number of devices a chip contains or to the amount of behavior that a design exhibits. As design complexity has steadily risen at a rate of approximately 58% per year, design productivity, the rate at which design engineers and their tools can build new processors, has only risen at a rate of 21% per year as shown in figure 2.1 [ITR05]. Therefore, more than a twofold increase in productivity in the coming years is required to make efficient use of increasing transistor density.

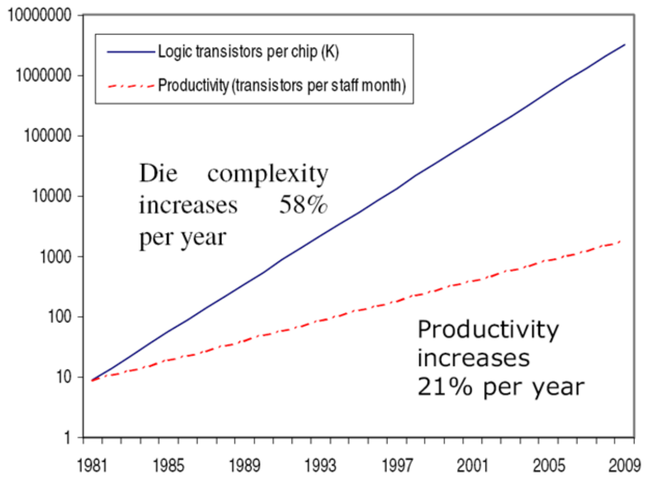


Figure 2.1: Design complexity and productivity gap. (from Sematech)

2.2.2 Power consumption

In recent years, maximizing processor performance has taken a back seat to limiting processor power consumption and increasing processing efficiency. High processor power consumption increases the cost of ownership of computers, limits the run time of mobile devices between charges, and stunts the growth potential of computer processors as the chips reach thermal package limits.

Power consumption in computer processors consists of a static component, called leakage power, and a dynamic component, called switching power (equation 2.1). Static power loss occurs when the circuit is in a steady-state. The switching power component captures the power loss due to the charging and discharging of the circuits as well as short-circuit power.[1] Short-circuit power is

[1]In complementary metal oxide semiconductor technology, or CMOS, short-circuit power

generally a small portion of dynamic power in well-designed circuits and is not discussed here.

$$\begin{aligned} Power_{total} &= Power_{static} + Power_{dynamic} \\ &= (I_{leakage} * V_{dd}) + [\alpha * C * V_{dd}^2 * f_{clk}] \end{aligned} \tag{2.1}$$

Although static leakage power has historically been small compared to dynamic switching power, the situation is changing as feature sizes decrease. The smallest *feature size* of a chip process technology refers to the smallest size of transistors, wires, or gaps between them that can be created onto the chip die with that technology. As these sizes decrease, the capacitance of the system of transistors, C, is lowered. This reduced capacitance decreases the switching time of those transistors, or gate delay, resulting in faster logic performance accommodating faster processor clock frequencies (f_{clk}). The activity factor, $\alpha | 0 \leq \alpha \leq 1$, approximates the average switching activity of the circuit for each clock edge.[2] The supply voltage, V_{dd}, is lowered to reduce interference with the ever-closer neighboring components and to meet thermal requirements. Lowering V_{dd} greatly reduces dynamic power consumption since the dynamic power is proportional to the square of this supply voltage. However, lowering the supply voltage in turn often requires a lowering of the threshold voltage, the voltage level at which transistors switch, to maintain fast clock rates. Lowering the threshold voltage and moving the threshold closer to ground causes a disproportionate increase in the static leakage current, $I_{leakage}$, and thus an increase in static power consumption [KAB+03].

2.2.3 Processor and memory performance gap

The computer processor is only one of the factors in computer system performance. Access to memory for instructions and data is often the bottleneck in today's systems. Computer processor performance has increased at a rate of roughly 55% per year while memory performance has increased at a rate of less than 10% per year as shown in figure 2.2 [HP03].

loss occurs when both n and p transistors are simultaneously enabled during a transition. The power lost is dependent on the slope of the rising and falling transition edges.

[2]The activity factor for a clocked flip-flop, for example, is $\frac{1}{2}$ since it toggles at most once per clock cycle and, thus, toggles once for every two clock edges. Techniques like clock-gating which decrease the number of logic transitions aim to keep this activity factor low.

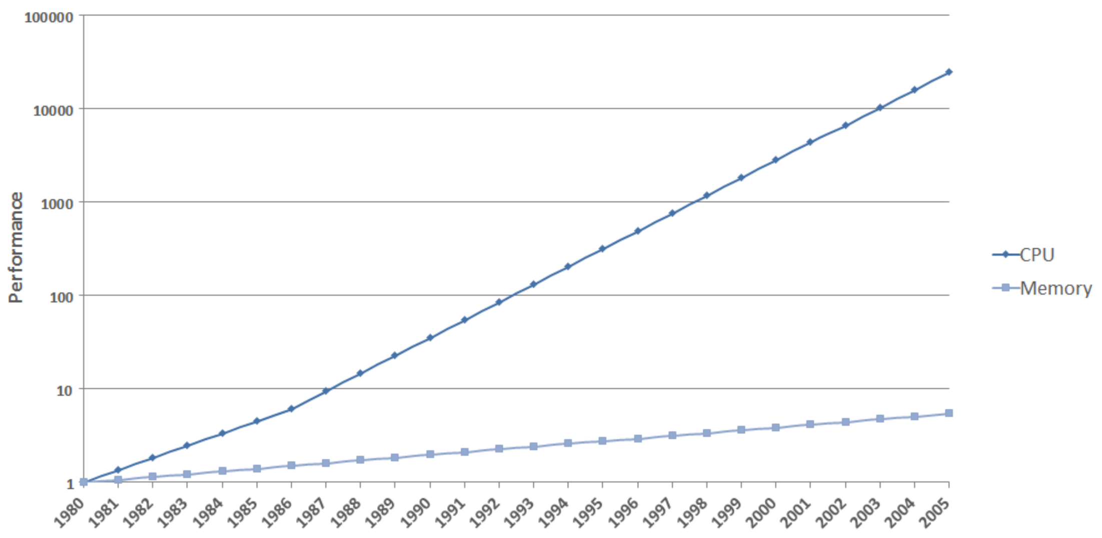


Figure 2.2: Memory and CPU performance gap. (Hennessy and Patterson)

Memory caches are used to store frequently accessed data closer to the processor core for faster lookup to help alleviate this problem. Small, local, simple caches can be accessed with very low latency and the data used very quickly by the processing core. As cache sizes grow, the access latency increases, and the benefits of a larger cache may not compensate for this increased lookup time. For this reason, architectures often have a hierarchy of caches with small, fast, level one (L1) caches close to the processor supported by one or more increasingly slower, larger caches (L2, L3, and so on). Though slower than an L1 cache, these larger caches still provide much faster access to data than off-chip memory with less power consumption. Much research has been conducted on the trade-offs of different cache hierarchy schemes [Jou90, FPT94].

2.2.4 Cost of communication

In early computers, state-holding logic took the form of large and expensive vacuum tubes. Communication was handled by small and relatively cheap wires. Today's trends show that logic in the modern form of transistors is cheap and continually getting cheaper; communication costs, however, are growing in significance.

As mentioned in section 2.2.2, wires and transistors on-chip are becoming smaller. Although smaller feature sizes result in decreases in gate delay, the same is not necessarily true for wire delay. Thinner wires with some material resistivity,

ρ, have increased resistance per unit length, $\frac{R_w}{l}$, due to lower cross-sectional area, A (equation 2.2). Wire delay, τ, is a product of both this resistance and of wire capacitance (equation 2.3). The capacitance is not decreasing proportionally to the resistance; even reducing the length, width, and height of a wire by some common factor will not necessarily change the wire delay significantly since one component of the wire capacitance, fringing capacitance, does not scale with feature size. Therefore, decreasing feature sizes are resulting in an increase in the ratio of wire delay to gate delay and ultimately in communication-bound performance.

$$\frac{R_w}{l} = \frac{\rho}{A} \tag{2.2}$$

$$\tau = R_w C_w \tag{2.3}$$

2.3 Research directions

Computer architects and engineers have responded to the challenges of designing general purpose computer processors in many different ways. Some of the most successful and interesting approaches are described here.

2.3.1 Exploiting parallelism

Parallelism in computer architectures is the ability for multiple actions to occur simultaneously. Parallelism is often categorized into the three types discussed here.

Data level parallelism (DLP)

General purpose architectures often support data-level parallelism (DLP), the parallelism found in the application of identical operations across many different data elements [HGLS86]. Some scientific applications and many multimedia applications exhibit large levels of DLP. Vector processors such as Cray Research's Cray-1 and Cray-2 use single instructions to operate on vectors of data elements at a time. Vector machines continue to be applied to scientific applications with high DLP, and research projects such as VIRAM show promise in applying vector processing to embedded markets [KP02, BG04]. Other common approaches for exploiting DLP are single-instruction multiple-data (SIMD) architecture extensions. SIMD instructions allow programmers to specify a single operation to apply to multiple, distinct data elements. Unlike vector processors, generally only a small number of elements that can fit in a single register at one time are operated on simultaneously. SIMD is fairly ubiquitous and has been supported by many commercial general-purpose processors since the mid-1990's in architecture extensions such as the Apple-IBM-Motorola alliance's AltiVec[3]; Digital's MAX; Intel's MMX, SSE, and WMMX; Sun's VIS; AMD's 3DNOW; MIPS's MDMX; and ARM's NEON.

Instruction level parallelism (ILP)

Many processors today also support instruction-level parallelism (ILP) by taking advantage of resource and data independence between instructions and executing multiple operations at one time. Superscalar processors (or *superscalars*) are capable of fetching and executing more than one instruction simultaneously and thus exploit some level of ILP. Statically-scheduled superscalars may execute several instructions in program order if there are no hazards between the instructions. These superscalars stop executing later instructions when any hazard is first encountered which limits ILP. Another class of architectures, very long instruction word architectures (VLIW), contain multiple operations in a single instruction. The compiler attempts to resolve dependencies and places independent instructions into slots of the instruction word for parallel execution by the hardware. This approach reduces hardware complexity versus superscalars since the independencies are discovered at compile-time and are explicitly specified to the hardware. More comparisons between superscalars and VLIW processors will be given later in this section.

Dynamic scheduling allows processors to find independences that cannot be found at compile time or through static run-time methods. Out-of-order superscalars are capable of examining many instructions at once, determining independences among those instructions, and executing multiple operations simultaneously and potentially out of order. Out-of-order superscalars can look past

[3]Also known as Velocity Engine and VMX depending on which company is referring to the extensions

instructions that cause hazards and execute one or more of the later instructions immediately. However, the hardware resources involved in finding ILP dynamically are large. Structures such as reorder buffers, rename register files, and instruction windows must grow in size and complexity with instruction issue width and the number of supported instructions in flight. Furthermore, dependency tracking logic grows quadratically in the number of instructions [HP03].

The limits of attainable ILP have been explored many times over with researchers drawing different conclusions [Wal91, Wal95, GG98]. The general consensus is that dynamic scheduling and employing speculative techniques of control and data value prediction are necessary to create large opportunities for exploiting ILP, but hardware overhead must be controlled. The latter proves difficult.

Thread level parallelism (TLP)

Some programming languages allow programmers to explicitly define threads of control in a program that may be executed concurrently. Additionally, multiprocessing operating systems may have several programs to execute each having its own set of threads that may also be executed concurrently. The parallelism made available by executing multiple threads at a time is called thread-level parallelism (TLP). Hardware support for TLP typically comes in two forms, multiprocessing and multithreading. Multiprocessing involves two or more processing cores either on a single chip or connected with an external interconnect. The most common type of single chip multiprocessor, or CMP, is called a *symmetric multiprocessor*, or SMP, which contains two or more identical cores connected to shared memory. SMP's are commercially available today on entry-level systems with the introduction of Intel Pentium D and AMD Athlon X2 processors. Commercial systems have widely been available for years that interconnect multiple single core processors to create a multiprocessing system.

Multithreading refers to hardware-supported execution of multiple program threads. Multithreaded processors hold the state of two or more threads at a time. These processors appear as multiple single processor cores to most operating systems though the execution resources in the hardware are actually shared among the threads. Sharing these resources versus duplicating them such as in CMP designs saves die area which helps to contain static, leakage power and chip fabrication costs. Early approaches to multithreading executed a single thread at a time, switching between contexts on long latency operations like memory accesses or at regular, statically-defined intervals. These techniques helped to eliminate wasted cycles but did not improve utilization of idle hardware resources. Simultaneous multithreading (SMT) enables multiple threads to dynamically allocate and use available hardware resources concurrently providing opportunities for higher levels of resource utilization than CMP designs. Under high load, wide instruction issue SMT processors show similar performance to CMP's with several processing cores and smaller supported instruction widths [TEL98]; SMT's with more modest complexity and thus smaller issue widths and numbers of supported threads like those in commercially available processors to date see less improvement. Intel's SMT technology is marketed as Hyper-Threading or HT. SMT die area overhead in Pentium 4 HT processors supporting two threads is 5–6%, and performance gains are from 15–25% for a variety of common desktop applications [KM03]. Despite low die area overhead, design cost for the implementation was considerable due to supporting two logical contexts, changing micro-op prioritization schemes, and validating the permutations of the x86 architecture's operating modes possible among the contexts [KM03].

Finding ILP in programs

The previous sections identified several types of parallelism, and many different approaches have been made to exploit it. Special attention is given to ILP here as the relationship between the hardware and software for making use of it is complex and requires many trade-offs. Architectures can be classified into three types based on hardware and software partitioning to find ILP: sequential, dependence, and independence [RF93].

As discussed, superscalar processors are given programs in the form of streams of instructions that are expected to execute one after another. ILP extraction from the stream must be performed in the hardware without any guarantees from software. However, a compiler may attempt to help hardware exploit ILP by applying any knowledge it has of the hardware implementation to schedule independent instructions next to one another in the code it generates. Such architectures are called sequential architectures. Of the three classes, hardware complexity for exploiting ILP is the greatest, hardware-software coupling is the loosest, and software's responsibility in finding ILP is the lowest.

In another class of architectures called dependence architectures, software communicates dependencies to the hardware directly. To execute instructions simultaneously, the hardware must still find independent operations, but the task is made easier than in sequential architectures due to the additional information the software provides. Data flow architectures are primary examples of dependence architectures. In data flow machines, an instruction typically contains a pointer to its successor instruction, the place to send the result of the current operation; this information expresses the dependence between the instructions. Dependence architectures have an easier task of finding parallelism than do sequential architectures, have loose hardware-software coupling, and rely on software to provide some information for exploiting ILP.

Finally, independence architectures may rely entirely on software to discover exploitable concurrency. VLIW architectures, for example, require software to fill slots in long instruction words with independent operations that may be executed in parallel. Generally, the compiler does most if not all of the scheduling work, and hardware in independence architectures can therefore be relatively simple and still exploit ILP. However, a hardware implementation may choose to search for additional independent operations at run time transparent to software like a superscalar processor does; this would of course result in an increase in hardware complexity. In independence architectures, hardware-software coupling is high as the compiler must often be aware of the hardware microarchitecture in order to determine instruction independences. Software and compilers for independence architectures are the most complex out of the three architecture classes presented here.

Independence architectures are becoming increasingly popular as a means to curb hardware complexity. VLIW independence architectures have been embraced in embedded markets as application-specific processors. TriMedia CPU64 VLIW processors can be found in set-top boxes and other embedded multimedia systems [vESV$^+$99]. Explicitly parallel instruction computing architectures, or EPIC architectures, are a variant of VLIW developed by Hewlett Packard Labs that combine instruction independence discovery in software with traditionally superscalar approaches to dynamic exploitation of ILP [SR00]. The family of Itanium processors use this architecture.

2.3.2 Increasing cache sizes

Today's high-performance processors are employing larger caches to help close the CPU and memory performance gap as shown in figure 2.3 [FH05]. Intel's "Montecito" Itanium 2 processor released in 2006 contains nearly 27MB of total cache memory on-die [MB05]. In addition to performance improving characteristics, caches have been optimized in ways to limit both dynamic and static power requirements thus providing power-efficient use of die area [FKM+02].

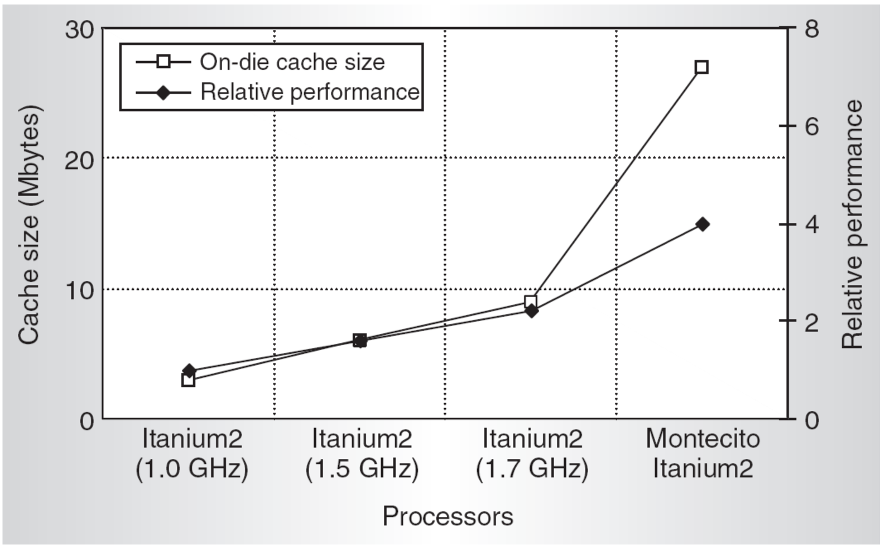


Figure 2.3: Cache size versus performance in Itanium2. (from Flynn)

2.3.3 Designing at higher levels of abstraction

A primary method of attacking the design complexity problem is to design architectures at higher levels of abstraction. Tools such as SystemC allow architects to develop high level models of architecture components and to explore the design space more quickly. SystemC-to-Verilog translators are becoming available to bridge the gap from the object oriented simulation environment to register transfer level (RTL) modeling and realization in hardware.

Another method of designing at a higher level of abstraction is the reuse of hardware and software blocks. SMP's, for example, copy multiple instances of identical cores into processors to exploit thread-level parallelism; the core is designed and validated once and reused many times. Tiled architectures are more extreme examples of this type of hardware reuse. Tiled architectures provide a framework for designing processors by copying a large number of logic and memory blocks in a regular fashion across the die. Raw architectures, for example, are built using multiple tiles containing a MIPS-style pipeline, a pipelined floating-point unit, several caches, and routers for communication with other tiles [TLM$^+$02]. Despite exhibiting lower performance on sequential applications than traditional superscalar processors, some evaluations show a 2–9x improvement with Raw for applications with high ILP and a 10–100x improvement when highly parallel algorithms are carefully optimized for the architecture [TLM$^+$04]. Unlike Raw architectures, Smart Memories have a heterogeneous mix of tiles that may be chosen for targeting specific applications and thus are also a type of reconfigurable architecture [MPJ$^+$00]. The TRIPs architecture is a grid architecture composed of homogeneous but polymorphic resources [SNL$^+$03]. Thus, unlike the Smart Memories approach, TRIPs uses identical processing elements and memories, but the behavior of these components can be dynamically reconfigured to enhance the performance of applications executing at the time by favoring ILP, TLP, or DLP at the software's discretion. These designs all reuse hardware blocks to minimize hardware complexity and development time.

2.3.4 Communication-centric architectures

In addition to tackling the problem of how to make use of growing transistor budgets, tiled architectures are also directly addressing the problem of increasing wire delays as a first order design constraint. This thesis refers to such architectures as communication-centric. The Raw and Smart Memories architectures both limit maximum tile sizes such that intra-tile communications require no more than one clock cycle [TLM$^+$02, MPJ$^+$00]. This requirement helps ensure that the tiled processors will scale with smaller feature sizes.

Some architectures place data movements directly in the hands of the compiler and programmer. MOVE architectures explicitly describe transports among functional units rather than the operations to perform [CM91]. The operations occur as side-effects of the transport. For example, if two data elements get sent to the two input ports of a dyadic multiplication functional unit, the values will be multiplied together. The result can then be moved directly from the multiplier to another functional unit; register files may be completely bypassed in favor of the shortest route to the successor operation. Forsell shows that MOVE architectures do well in reducing hardware complexity but do not compete with the performance of traditional general-purpose architectures [For03].

2.4 Conclusion

This chapter discussed some of the contemporary challenges in computer architecture and also some of the current approaches to address these issues. The next chapter describes the Fleet architecture designed to target these challenges. It is the foundation for the Armada architecture discussed in the remainder of the thesis.

Chapter 3

The Fleet architecture

The previous chapter discussed some of the problems today's computer architectures will face in the foreseeable future. This chapter introduces the Fleet architecture and discusses how it proposes to address those concerns. A description of the instruction set architecture is given followed by an overview of the hardware organization. Finally, some initial observations on working with Fleet and some of the architecture's constraints are discussed.

Note that much of this material may be found in a research memo [Sut05]; however, that document may not be readily available. Thus, this chapter has been included to introduce the Fleet ideas developed by Sutherland, Benko, students from UC Berkeley, and others in the context of this research. Additionally, Fleet has continued to evolve in tandem with this research. The snapshot described here served as the basis of this study and does not reflect the latest updates to the architecture made by the other contributors.

3.1 Architecture overview

Computer architects have reacted to changing design and fabrication technology trends by proposing new architectures to accommodate and also benefit from these developments. Ivan Sutherland has proposed one such architecture called Fleet. Fleet:

- applies the increasing quantity of transistors available each year to integrating more functional units as opposed to complicated control logic
- simplifies hardware design by employing a very modular approach to the

system architecture

- puts expensive communication in the hands of the compiler and programmer where optimizations can be made without complex hardware

The next sections describe Fleet's ISA and organization and present some results from initial testing. Fleet is a work in progress, and some behaviors and specifications have not been concretely established. These loose ends are noted in the chapter as the topics are discussed.

3.2 Instruction set architecture

The Fleet ISA differs from those of ubiquitous architectures in several ways. Here, Fleet hardware data typing support is discussed followed by a description of the one Fleet instruction, the *move* instruction.

3.2.1 Native hardware data types

Fleet natively supports several hardware data types (figure 3.1). Data of any type may have an *out-of-band*, or *OOB*, value. *OOB* values indicate special error or termination conditions such as the end of an array or a memory parity error. Each native type supports at least one *OOB* value, *last*. In order to represent *last*, each data type must therefore have at least one additional bit. If multiple *OOB* values are supported, more bits are needed to represent the type. Different uses for *OOB* will be elaborated on in subsequent sections.

Type	**Description**
token	a dataless event useful for sequencing
boolean	$true$ or $false$
character	unsigned 16 bits
integer	32 bit signed value
long	64 bit signed value
code bag descriptor	a reference to a bag of code
memory pointer	memory address (size currently unspecified)

Table 3.1: Fleet native data types. All types additionally support at least the *OOB* value "*last*" and thus require at least one additional bit to represent this (not shown).

3.2.2 One instruction

Fleet instructions describe transports that the hardware should perform rather than operations. Reduced Instruction Set Computer (RISC) ISA's are some of the most common programming interfaces in today's processors. RISC instructions describe an operation to perform, the operands required for that operation, and where to write the result. Architectures that employ such an ISA are known as *operation triggered architectures* (OTA's). In contrast, Fleet instructions are more closely related to instructions characteristic of *transport triggered architectures* (TTA's). Instructions for TTA's describe a source location of data and one or more destination places to move that data to; the operation performed is a side-effect of the transport. Table 3.2 shows code for a simple dyadic addition operation in both generic RISC and equivalent Fleet instructions.

pseudo-code	**RISC**	**Fleet**

`r0 := r1 + r2`	`add r0, r1, r2`	`r1 -> add.in1`
		`r2 -> add.in2`
		`add.out -> r0`

Table 3.2: Comparison of Fleet and RISC instructions.

The general form of the Fleet move instruction takes one input from a source port and delivers it to one or more destinations[1]. From this template, several embellishments to the instruction have been suggested. One such addition is that of a mini-opcode that is context-specific to each destination. For example, a mini-opcode may instruct an adder unit destination port to negate one datum on arrival perhaps changing the addition operation to a subtraction operation. An assembly language syntax for other proposed variants of the instruction is given in Table 3.3. As it is useful to describe Fleet programs graphically, the table also describes how move instructions are drawn in program diagrams throughout this thesis. These variations are described in the following sections.

One-shot

One-shot instructions transport one item of data from a source location to one or more destinations.

[1]The number of possible destinations per instruction has not been established though Sutherland has recommended three as a starting point.

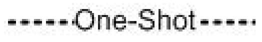
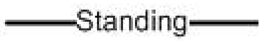
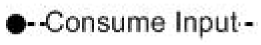
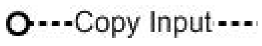
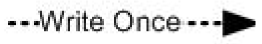
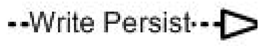
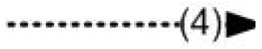
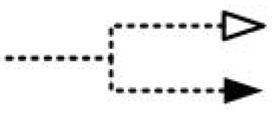
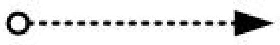
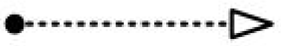
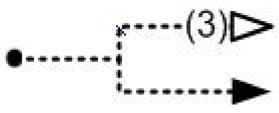
Description	**Text Syntax**	**Graphical Form**
One-shot (arrow shaft is `-`)	`src -> dst`	
Standing (arrow shaft is `=`)	`src => dst`	
Consume data at source (no `*` before arrow)	`src -> dst`	
Copy data at source (`*` before arrow)	`src *-> dst`	
Consume data at destination (no `*` before `dst`)	`src -> dst`	
Reuse data at destination (`*` before `dst`)	`src -> *dst`	
Pass a mini-op to a destination	`src -> dst(4)`	
Multiple destinations (the destination options may vary independently)	`src -> *dst1, dst2`	
Examples		
One-shot preserving data at source and consuming data at destination	`src *-> dst`	
One-shot consuming data at source and reusing data at destination	`src -> *dst`	
Multiple destinations, input consumed, dest1 reuses data and dest1 accepts a mini-op, data consumed at dest2	`src -> *dst1(3), dst2`	

Table 3.3: Move instruction syntax and graphical representation.

Standing

Standing instructions repeatedly move data from a source location to one or more destinations as data at the source becomes available. If multiple destinations are specified and one destination is blocked for a period of time, the other destinations may still receive transfers from the source. The data will queue up for the blocked destination; no data will ever be lost. A standing instruction breaks down when the source data being transferred is *OOB*; the instruction carries that *OOB* value then expires.

Consume input

The data is consumed at its source location by the move instruction and transported to one or more destinations.

Copy input

The data is copied from its source location by the move instruction, and the copy is transported to one or more destinations. The data remains valid at the source, and the next transport from this location will reuse the data.

Write once

The data is consumed at the destination.

Write persist

The data persists at the destination. This form of move is useful for supplying constants to destinations; the source data is communicated only once and is continuously reused.

Fleet move instructions may specify direct transfers between functional units allowing any register file in the system to be bypassed completely. This decentralization of data flow allows Fleet to use simpler, faster register files or perhaps to eliminate the use of a register file altogether.

Programmers can create virtual pipelines between functional units by issuing chains of standing instructions. Standing instructions bind a source to one or more destinations indefinitely. Data is repeatedly forwarded to the destination or destinations as soon as input data at the source is available and space to write this data at the destination is available. Standing instructions are broken when the data being carried is OOB. OOB values may be explicitly introduced by software or be produced as the output of a functional or storage unit based on specific conditions. For example, an arithmetic unit may produce an OOB value if any of its inputs are OOB or if an overflow or underflow condition occurs. A standing instruction that encounters OOB data will deliver this OOB value then expire. OOB status can be propagated in this way, breaking down lengthy virtual pipelines.

3.2.3 Code bags

In most ISA's including the RISC ISA, programmers may assume instructions are executed in the order they are specified in the program. Fleet does not follow this model of inherently sequential processing. Concurrently executable move instructions are placed into units called *code bags*. The programmer or compiler ensures that transports in code bags may be executed in any order and, despite the execution order, will always produce the desired result. Thus, Fleet is a type of independence architecture. Recall from chapter 2 that independence architectures require software to specify which instructions are independent from each other and, therefore, which instructions may be executed concurrently. A particularly capable hardware implementation may issue all instructions in a code bag at one time.

In sequential architectures like RISC architectures, a program counter (PC) keeps track of which instruction should be executed next. Fleet does not have a PC. Instead, a code bag is responsible for fetching its successor bags, if any, by moving *code bag descriptors* defining those successors to a fetch unit. This operation will be described further in a following section.

3.2.4 A simple Fleet program

Listing 3.1 shows a code bag that produces the sum-of-squares for a range of numbers[2]. A stride unit generates the integers 1–100. It is connected to both inputs of a multiplier via a single standing instruction. The multiplier will, therefore, generate the squares $1*1, 2*2, 3*3, \ldots 100*100$. The multiplier's output is connected to an accumulator, also via a standing instruction, that will compute the sum of all inputs received. The stride unit will produce a final $last - OOB$ value after producing the last valid count — 100 in this example. The multiplier will respond to *last* inputs by generating a final *last* value at its output. At this point, the standing instruction connecting the stride unit to the multiplier inputs expires. The *last* output from the multiplier is then carried to the accumulator. This connection will also expire once the *last* value is delivered. Finally, the accumulator is implemented to respond to a *last* input by producing the current accumulated amount as an output, the sum of squares in this example. Note that the instruction moving the accumulator output to the register may have been waiting for the accumulator to generate output for a long period of time. Additionally, the order of the instructions in this code bag, like every code bag, is of no consequence. The programmer should ensure that the output will be the same regardless of any particular hardware implementation's instruction execution order.

```
codebag sumOfSquares {
  1 -> stride.start, stride.step;    // one-shot instruction to
                                     //   multiple destinations
  100 -> stride.stop, stride.next;   // last number to produce
  stride.out => mul.in0, mul.in1;    // send 1, 2, ...100 to mul
  mul.out => acc.in, stride.next;    // standing instruction
  acc.out -> r0;                     // write result when count
} sumOfSquares;                      //   complete(OOB received)
```

Figure 3.1: Fleet code that computes the sum-of-squares for the numbers 1–100. One-time instructions are specified by the single arrow operator, and standing instructions are specified by a double arrow. Sending a token to *stride.next* causes the next number in the sequence to appear at *stride.out*.

[2]If you follow the program and are concerned about left-over state when the program completes, good observation! Cleanup will be discussed in a later section.

3.3 Concurrency

As demonstrated by the ISA, Fleet requires the programmer to explicitly find instruction level concurrency (ILC). Programmers and compilers describe ILC to Fleet hardware by finding independences among instructions and by grouping the independent transports into code bags. Finding independences at compile-time allows hardware to exploit ILC easily at run-time with no processing overhead spent or hardware complexity required to search for candidate instructions.

3.4 Hardware organization

A Fleet processor contains three main types of components. The fetch and dispatch unit fetches code bags from memory and dispatches the instructions to another primary component, the switch fabric. The switch fabric, or interconnect, moves data from sources to destinations based on the instructions it receives. The sources and destinations are the output ports and input ports, respectively, of the third and final type of primary components, Ships. Ships are the functional units and storage units that may generate and receive data, usually performing some action on the inputs. Figure 3.2 shows the high level organization of a Fleet computer.

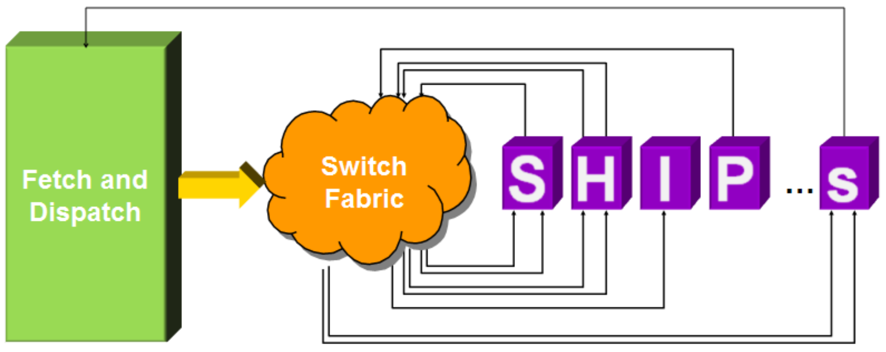


Figure 3.2: Fleet architecture organization.

3.4.1 Instruction fetch and dispatch

The fetch and dispatch unit receives code bag descriptors as an input and produces Fleet instructions loaded from memory as an output. The code bag descriptors are received from a special fetch Ship in the Fleet. The descriptors are then translated into the set of contiguous memory addresses that hold the instructions in the code bag. These instructions are fetched from memory and await dispatching to the switch fabric in an instruction pool.

3.4.2 Switch fabric

The switch fabric receives routing instructions from the fetch and dispatch unit and routes data amongst Ships as directed by those instructions. There are many possible interconnect implementations such as a fully connected crossbar connecting every Ship output port to every Ship input port, a simple bus, or a heterogeneous mix of designs. As reflected in the ISA, the interconnect must hold instructions indefinitely and wait for data to appear at sources. Likewise, it must hold transported data indefinitely until the destination Ship accepts it.

3.4.3 Ships

Ships may be functional units such as adders and multipliers as well as sequencing units or register files. Many Ships receive data at one or more inputs and produce data at one or more outputs. However, not all Ships will contain both input and output ports. An example of a Ship that produces data but does not receive inputs is a random number generator. Similarly, a bit bucket receives inputs but does not generate any outputs. Ships may be implemented in numerous ways; as with the interconnect, how Ships accomplish their tasks is loosely defined. For example, Ships may be pipelined or not and be synchronous or asynchronous. As reflected in the ISA, Ships must only wait for inputs to arrive and wait for outputs to be picked up.

3.5 Early findings

Currently, few quantitative results have been gathered about the Fleet architecture. The Fleet instruction set architecture is unique, and it is taking some time for researchers to grasp the implications of it. A Java simulator of the Fleet architecture has been built by Sun Microsystems and students from U.C. Berkeley that has primarily been used to explore how programs might be written for Fleet. Most programs require specific Ships to be present in the system that exhibit behavior tailored to the task. Some program and hardware pairs include an accumulator of the Fibonacci sequence [Hol05], an eight-element bubble sort [Isa06], an implementation of Euclid's algorithm [Mey06a], and vector-matrix multiplication [Mey06b].

Typical problems encountered were congestion in the switch fabric and difficulties with programming the cleanup of orphan tokens left in the system after algorithm completion [Hol05][Mey06b]. Identification of some sources of congestion and some remedies are presented in [Hol05], but a general solution has not been found.

3.6 Limitations

Although a basic Fleet microprocessor may tackle application-specific tasks well, the architecture is not well-suited for general purpose computing. First, the software's additional responsibilities of mapping transports to hardware ports require programs to know details about the microarchitecture. Specifically, software must be aware of the port addresses for every Ship to make full use of the available resources. Thus, any significant change to the hardware may result in changes required to the software for it to utilize the hardware efficiently. This coupling ultimately leads to either software incompatibility between generations of processors or the inability for legacy code to benefit from new resources that become available in the future. Secondly, perhaps the greatest inhibiting factor to realizing a general purpose Fleet processor is the inability to easily run multiple programs or multiple threads. The current architecture requires programs to share Ships cooperatively. Fleet programs must therefore yield control to other programs in order to multi-program, an archaic programming model. Additionally, the amount of parallelism that can be found statically in a program in the form of ILP has practical limits. TLP offers additional potential for concurrency of operations that Fleet is currently unable to efficiently capitalize on.

3.7 Conclusion

The basic Fleet architecture promotes modularity, flexibility, and concurrency which confront many of the current challenges in computer architecture. Fleet is globally asynchronous, freeing components from implementation restrictions common in synchronous systems. The hardware blocks are easily interchangeable and coupled with other units only through a common asynchronous protocol. Furthermore, the programmer and compiler are given the responsibility of ensuring optimal communication between components; static, compile-time optimization allows thorough analysis to replace complex and inflexible hardware controls and optimizations. Ships may easily be added as more transistors become available on chip. Software may make use of the additional computational units and increase the achievable amount of instruction level parallelism. Finally, the compiler and programmer find the independences among instructions in Fleet; thus, the hardware has no overhead in discovering ILP as it is explicitly provided by software.

However, Fleet does not appear to apply well to general purpose applications. Strong coupling between hardware and software may create binary compatibility problems as more Ships are added to Fleet processors at new address ports. New resources will go unused by legacy software limiting the amount of parallelism exploited by these applications. Additionally, multi-programming and exploiting TLP are not practical in Fleet as all processes would be required to cooperate and share Ships through inflexible compile-time partitioning.

Part II introduces new architectural improvements that attempt to remove the aforementioned limitations. Using Fleet as a great stepping stone toward reaching the goal of a scalable, more power-efficient general computer architecture for the future, the proposed enhancements are integrated with this previous work to form the Armada architecture.

Part II

The Armada architecture

Chapter 4

The Armada architecture

The Fleet architecture provides a unique instruction set architecture which allows software to express independence among instructions in a simple and useful way; hardware can profit from this instruction-level concurrency easily with little complexity. However, Fleet lacks similar methods of extracting and programming for thread-level concurrency.

Armada is a multicore architecture composed of simultaneously-multithreaded Fleet cores. The constructs introduced in Armada attempt to enable exploitation of thread-level concurrency similarly to how Fleet currently takes advantage of instruction-level concurrency. With more concurrency available supported by the many multithreaded Fleet cores on a single chip, the Armada architecture will hopefully demonstrate higher throughput than Fleet with minimal complexity and hardware overhead. To keep the complexity low, a cautious division of labor between software and hardware and between static and dynamic responsibilities is taken.

4.1 Differences from Fleet

Several deviations from the Fleet architecture were made during the early stages of this research. These changes and the rationale for altering the original specification are discussed here.

First, some proposed features of Fleet are orthogonal to its more radical proposals of dismissing the program counter and exposing ILC to hardware through code bags. To focus on these more unexplored frontiers of Fleet, some of these features were removed. Hardware enforcement of data-typing is not supported in

Armada. Additionally, the range of out-of-band values is essentially eliminated; only the *last* token behavior is retained to support virtual pipeline destruction and Ship reset.

Fleet's move instructions contain a count field to support a specific number of repetitions of an instruction before it expires. Armada generalizes this behavior supporting only single and unbounded, standing-instruction types. Supporting a specific repeat count is a specialization left for future study.

The Fleet memo also refers to some flow-control mechanism for facilitating communication between pipelined Ships. Although such a mechanism is probably crucial in achieving high single-thread performance, it is not strictly necessary for a Fleet processor to operate. Armada does not contain such a mechanism. Programming with virtual pipelines is discussed in more detail in the instruction set architecture section of this chapter.

The Fleet proposal refers to a hardware master-clear capability to bring a Fleet processor back to its clean, reset state. As such a system has not been devised, Armada relies on software to manage Fleet state and to detect when Fleet cores are clear of data and instructions. Software can then make the Fleet available for reuse by other programs or threads.

Additionally, the memo describes some Fleet Ships as pure sources and pure sinks. Pure source Ships may be possible depending upon the capabilities of the pipeline communication layer used between such Ships in the processor implementation. As Armada does not contain such a layer, a source Ship must be instructed to generate output by passing it a token. It must therefore expose at least one input trigger port to software, and, consequently, it is not a pure source.

Armada contains only one Ship that may operate as a pure sink, the fetch Ship. The memo describes another pure sink Ship that this research has found unusable. As discussed earlier, Armada requires software to manage token movements in order to prevent deadlock and to detect when a Fleet is free of all state after performing some computation. The memo suggests disposing unneeded stray data tokens into a pure sink bit bucket Ship. Without an output port to deliver a token indicating when stray tokens have reached the bit bucket, software cannot know when these tokens have been removed from Ship output ports and have cleared the switch fabric. Therefore, in Armada, a counter Ship takes the place of the bit bucket in collecting stray tokens. The counter generates a single output after it receives the software-specified number of tokens to dispose of. This output token can then be used to gate the fetch of a successor code bag using some synchronization Ship like a join Ship as shown in figure 4.1. Appendix A details the behavior of these Ships and all others mentioned in this thesis.

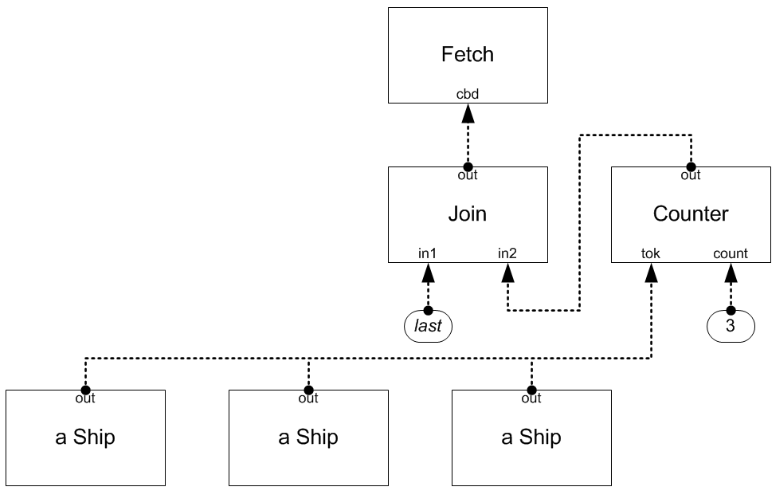


Figure 4.1: Gating fetch of a successor code bag on token cleanup. This example demonstrates a way to ensure stray tokens are cleaned up prior to fetching a successor code bag. The counter Ship produces an output token once all of the stray tokens arrive. The join Ship waits for both inputs $in1$ and $in2$ to arrive. It then consumes $in2$ and forwards $in1$ to the fetch Ship. Software must know which tokens are stray and where those stray tokens are located.

Fleet proposes specialized register Ships that allow software to overwrite their contents. While this behavior is undoubtedly useful as it would decrease the number of stray tokens and eliminate the accompanying data movements required to dispose of them, the implementation and use of such Ships in an asynchronous, highly concurrent environment is unclear. Armada does not provide such capability in its register Ships.

Finally, this research branches off the Fleet architecture as it was defined three years ago. Fleet has continued to grow independently in a different direction through continued development at Sun Microsystems. Analysis of the Fleet architecture as it exists now is generally beyond the scope of this research. However, some aspects of the latest version of Fleet are discussed in terms of related work at the end of this chapter.

4.2 Architecture enhancements

The many, possibly heterogeneous Fleet cores that compose an Armada processor provide the hardware needed to run multiple threads simultaneously. These cores may differ in Ship composition, port mappings, switch fabric design, performance, power consumption, or any number of other characteristics to target specific workloads and to provide flexibility for future enhancements.

To capitalize on these additional hardware resources Armada makes available, four ISA modifications are proposed. The modifications are not orthogonal to each other; several of the constructs work together to exploit thread-level concurrency while balancing complexity. First, the ability to distinguish resource-dependent code bags from resource-independent ones is proposed. This delineation allows software to express which threads are concurrently-executable thus providing hardware with the ability to take advantage of TLC with minimal complexity. The second proposal consists of the mandatory use of a register file for centralizing the largely distributed state in a Fleet and a mechanism to forward this state to other cores as a way to support multithreading. Additionally, Armada introduces context synchronizers that allow many branches of a program's flow to merge before continuing past a barrier point. The final scheme aims to enhance processor performance when executing common code sequences through the use of cacheable software-generated flows based upon the virtual pipelines in Fleet. These flows are systems of virtual pipelines among Ships designed to perform some specific task. Caching flows reduces instruction fetches and average task run-time by reusing these systems and amortizing their setup cost over time. The hardware performs the cache management, autonomously reacting to dynamic run-time conditions, effectively making Armada a self-reconfiguring architecture.

The following sections discuss these proposals in more depth. Subsequently, the instruction set architecture is explored more rigorously, and details on writing software that uses these enhancements are given.

4.2.1 Independent code bags

Code bags in the basic Fleet architecture are all *resource-dependent* bags; the execution of any such code bag is dependent on the Ships, switch fabric, and other hardware resources retaining state from the execution of instructions in predecessor bags. Armada introduces *resource-independent* code bags to express an executable context's independence from any particular set of resources and associated state. These bags have types associated with them that correspond to the different types of the heterogeneous Fleet cores present in an Armada. The hardware can execute independent bags on any unallocated Fleet of a matching type in the system. State like a stack pointer or function arguments may be sent forward from a predecessor bag to an independent code bag successor through the local register file. The Armada hardware will load this data into the register file of the Fleet core allocated to run that successor bag.

4.2.2 Enhanced local register file

The ability to capture and restore system state allows an unbounded number of processes to execute on general-purpose processors.[1] The state is typically captured and restored by an operating system to support multiprogramming and multithreading. Furthermore, state can be replicated to spawn new programs or threads from an active process. In traditional architectures, the state of the system largely resides in the register file and program counter.[2] In contrast, state in Fleet is widely distributed among Ships and the interconnect. Capturing this distributed state is a difficult chore and may require costly hardware support.

As an alternative solution, software can use the local register file to centralize a Fleet core's state at various points in a program. Armada adds a mechanism allowing hardware to package the code bag descriptor of the next independent code bag to fetch along with any arguments that are to be forwarded to that bag. This package can be stored to memory and later recalled for execution.

Note that hardware can only capture thread state at independent code bag fetch boundaries in a program; it cannot capture the state arbitrarily at any moment. Therefore, in the current implementation, a malicious program can spawn many threads that never free the Fleet cores allocated to them and livelock the system. This problem can be solved by the hardware reset mechanism described earlier which is left as future work.

[1]The number of processes that may be executed on modern computers is limited only by the memory available to store the state for each of the processes.

[2]Threads typically only have hardware state in the register file and the program counter. Processes, however, have hardware state that may include page tables and other resources.

4.2.3 Context synchronizers

Armada exposes context synchronizers to software that provide thread barriers in multithreaded program flow; programmers configure synchronizers to wait for multiple threads of a program to complete before continuing on with that program's execution. Hardware support is proposed because many programs will likely require thread synchronization. Additionally, threads in Armada may have quite short lifetimes relative to the time software-only methods would require to support the same barrier behavior.

4.2.4 Flow caching

Fleet already provides one way to speed up the execution of serial code by providing virtual pipelines to connect Ships. Flow caching is an extension of this idea aimed at exploiting a commonly occurring pattern in programs where the same instructions are repeatedly applied to different data. Flows are a specialized subset of independent code bags that setup a network of virtual pipelines once and allow this configuration to be applied to different data elements over time. Flows reduce the number of instruction fetches by amortizing the pipeline network setup cost over future invocations of the same code. Fetching a cacheable flow, like fetching an independent code bag, starts a new thread on a free, compatible Fleet core.

Armada hardware may cache a flow in a Fleet core if that core is not needed by other programs. When software fetches a flow's descriptor, the Armada hardware, at its discretion, would attempt to find a free, cached instance of the flow to run the thread on. If successful, the register file contents are sent to the flow; instruction fetches are largely or completely avoided. If a free instance of the flow is not found, the hardware may either wait for a busy instance of the specified flow to become free, or it may load another copy of the flow onto an empty core as it would load any other independent code bag. If a particular flow can be applied to several data sets at once, Armada may cache multiple instances of the code on different cores at the same time.

The proposed system is self-reconfiguring as the hardware-controlled caching frees the programmer from micromanaging the allocation and eviction of flows. However, software may provide hardware with hints on caching policy and also lock time-sensitive flows like interrupt handlers onto a core. These behaviors and the mechanisms to support them are similar to instruction cache policies and cacheline locking features in traditional architectures.

The following section discusses the software view of how to program with flow caching and the other proposed features of Armada.

4.3 Instruction set architecture

This section describes the Armada ISA in enough detail for programmers to write simple programs using the aforementioned features. General programming concepts are discussed, and a prototype Fleet core is described.

4.3.1 Memory model

This research does not address the instruction and data memory subsystems in any detail. Although Armada may be better matched with a different type of memory model, it currently employs a uniform memory access architecture for simplicity. Memory is byte-addressable, and both instruction and data addresses are 64-bit. Reads and writes of multi-byte data types must be aligned according to their respective sizes in bytes. For example, a four-byte word must begin at an address divisible by four.

4.3.2 One instruction

The *move* instruction is mostly unchanged from that of the basic Fleet architecture (recall section 3.2.2). However, as mentioned earlier in this chapter, the count field is not present in Armada.

Armada defines several opcodes for the different variants of the instruction and fills in the gaps left in the Fleet-overview memo. In particular, a method of handling constants generated at compile-time is given. Figure 4.2 shows a summary of the encodings for the different variations of *move*.

The move variant in figure 4.2(d) is the most complex. While the other moves are a fixed size of 32 bits, this instruction opcode must be directly followed in memory by a constant that may be 4, 8, or 12 bytes long.[3] Fleet allows the hardware to fetch and issue instructions in a code bag in any order; that flexibility

[3]The $TYPE$ field encodes the size of the trailing constant. This field is larger than necessary; it originally described the hardware data type of the constant which has since been defeatured.

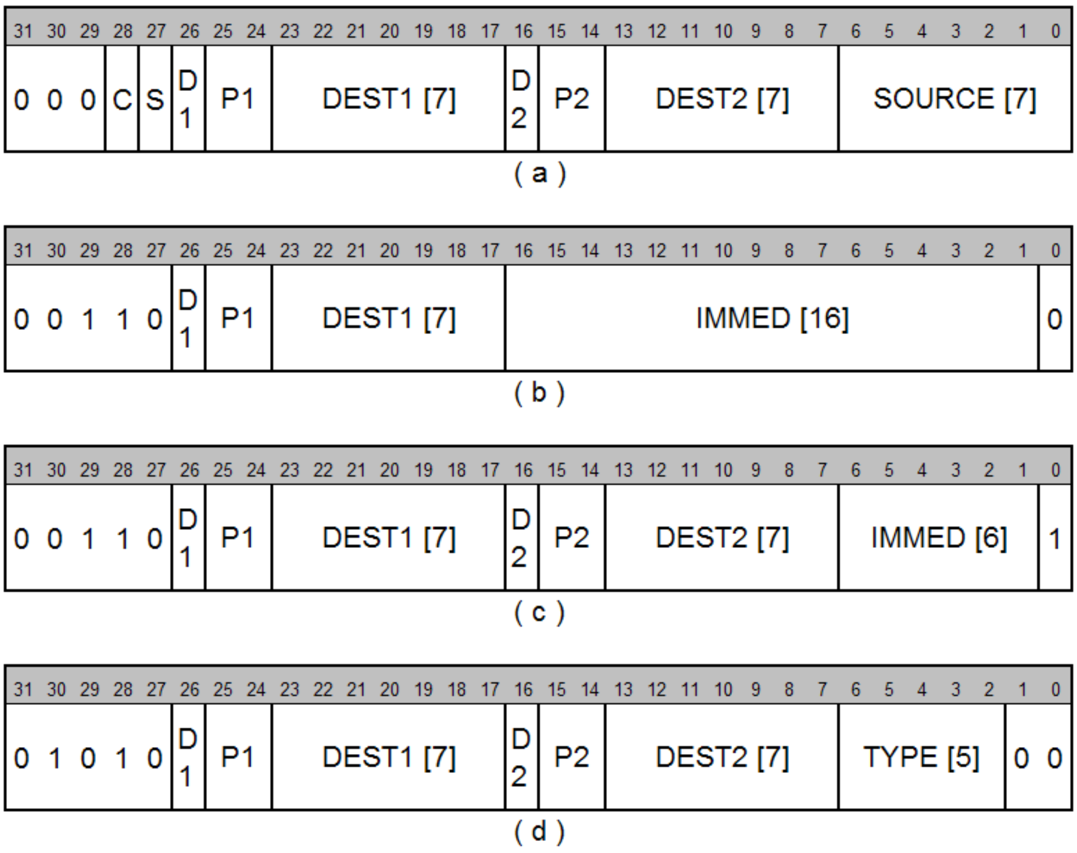

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 C S D1 P1 DEST1 [7] D2 P2 DEST2 [7] SOURCE [7]
(a)
0 0 1 1 0 D1 P1 DEST1 [7] IMMED [16] 0
(b)
0 0 1 1 0 D1 P1 DEST1 [7] D2 P2 DEST2 [7] IMMED [6] 1
(c)
0 1 0 1 0 D1 P1 DEST1 [7] D2 P2 DEST2 [7] TYPE [5] 0 0
(d)

Figure 4.2: The encoding in (a) is the commonly used port-to-port move instruction variant. The C bit indicates whether the source data is consumed or copied. The S bit indicates whether this is a standing instruction that repeatedly moves source data as it becomes available. The D bits indicate whether the moved data should persist at the destination ports, $DESTx$. The P bits pass along destination-specific mini-opcodes. Finally, $SOURCE$ identifies the source port where the data is read from. Encodings (b) and (c) allow small immediate values generated at compile-time to be delivered to Ship input ports. The final encoding (d) moves large constants generated at compile-time that do not fit within the 32-bit instruction opcode. Software places a large constant immediately after the instruction opcode describing it in memory. The $TYPE$ field indicates the constant's length.

allows the hardware to fetch and dispatch instructions that may be in a nearby cache, for example, before those instructions that are stored further away in the system regardless of their location in the bag. Variable-length instructions complicate how the hardware can take advantage of that instruction issue freedom as it must perform a sequential linear scan of the bag to identify the variable-length moves and handle them appropriately.

Armada fetches instructions in a code bag in 32-byte cacheline chunks. The instruction fetch unit decodes instructions within each cacheline and handles variable-length instructions as necessary. Armada does not allow variable-length instructions to cross cacheline boundaries. This restriction allows Armada to issue all instructions within a cacheline that falls in the middle of a code bag without needing the surrounding cachelines. By imposing this cacheline-boundary restriction on long instructions, Armada retains the spirit of Fleet's ability to fetch and execute whichever instructions in a code bag are most readily available though not with the finest level of granularity. However, as microarchitectures commonly move instructions around in groups of cachelines anyway, there is little impact of such a restriction.

4.3.3 Choosing the code bag type

Programs issue instructions by fetching code bags. Armada provides programmers and compilers with three different types of code bags. The characteristics of the different bags make them suitable for different parts of a program as summarized in table 4.1.

Resource-dependent bags

Resource-dependent bags continue a current thread of execution. Small, sequential tasks that rarely execute should be defined in dependent bags to avoid the time and energy costs of allocating a Fleet core for infrequent and short-lived computations.

Resource-independent bags

Resource-independent bags spawn new threads of execution. Software should use independent bags to describe any tasks that can execute concurrently. Armada programmers and compilers should not be shy about spawning threads. The

Type	Spawns thread	Run-time overhead	Software complexity	Uses
dependent	no	lowest	lowest	infrequently-executed, sequential segments of a program
independent	yes	highest	medium	concurrently executable segments of a program; ability to switch core types in a heterogenous Armada to accommodate or optimize for different subroutine characteristics (eg. ILC, TLC, control-flow, etc)
independent flow	yes	medium	highest	concurrently executable segments of a program that are repeated often, eg. inner loops

Table 4.1: Code bag comparison.

number of threads a program may spawn is ultimately only restricted by the amount of physical memory available to store the independent tasks. However, an Armada operating system may choose to impose limits on the number of active threads a program can have similar to how operating systems on traditional architectures impose limits on a program's stack size. These limits are typically never reached by a correctly written program.

Additionally, fetching an independent bag allows a program to choose a different type of Fleet core to execute code on in heterogeneous-multicore Armadas. For example, a program entering a segment rich in floating-point computations may stop the current thread and continue with a new thread on a type of core containing many floating-point units. When that computation is complete, the program may switch back to a Fleet type that has more control-flow Ships to determine what to do next.

Cacheable independent flow bags

Cacheable flow bags are a subset of resource-independent bags that spawn new threads of execution on a software-selected type of Fleet core like independent bags. However, their special property of being reusable with amortized instruction fetch and setup costs makes them especially attractive for programs that execute the same instructions often. Flows typically have higher overhead than independent code bags as they must be specially coded to ready themselves for new data and to self-destruct when hardware commands them to. Thus these bags should only be used when the code sequence they describe is expected to be reused often.

Examples using each code bag type are given in the following sections.

4.3.4 Fetching code bags

Programs refer to code bags using descriptors. Table 4.2 describes the syntax and graphical representation for dependent, independent, and cacheable flow code bag descriptors.

Code bag descriptors

Fleet cores must support independent and dependent code bags. Flow support is optional. The 96-bit descriptors that describe these bags are shown in figure 4.3.

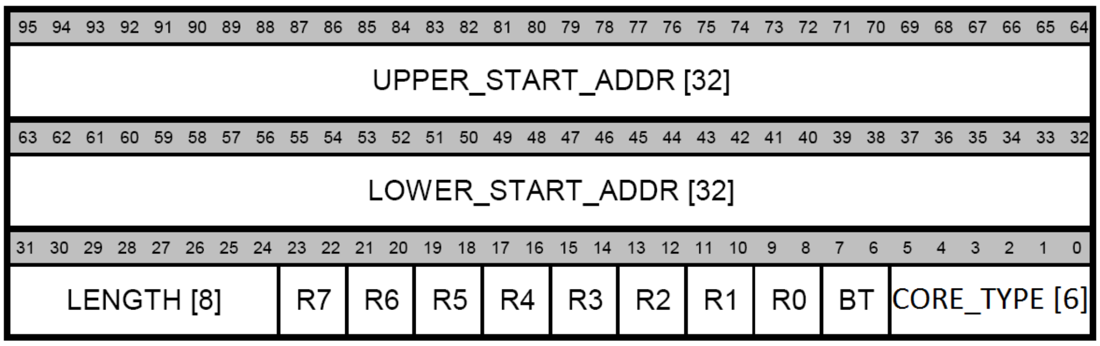
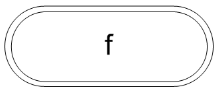


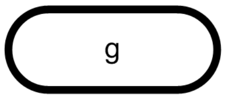
Figure 4.3: Code bag descriptor format.

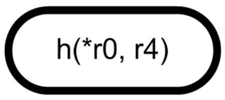
Code bag descriptors describe a code bag's location in memory with a 64-bit physical start address and an 8-bit length field. The hardware uses these fields to load the appropriate instructions from memory.

Certain registers' contents may be forwarded to successor code bags. There is one 2-bit field for each such register that indicates whether the register's contents are ignored, copied, or moved to the successor. When the fetch Ship in a Fleet receives a code bag descriptor, it checks these fields and packages the contents of

Description	Text syntax	Graphical form

Dependent code bag	`dependent codebag f {`
	`  ...`
	`} f;`
		

Independent code bag that does not take any input arguments from the register file	`independent codebag g {`
	`  ...`
	`} g;`
		

Independent code bag that copies the contents of `r0` and moves the contents of `r4` from the fetching Fleet and uses them as inputs	`independent codebag h(*r0, r4) {`
	`  ...`
	`} h;`
		

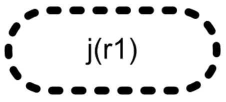
Independent cacheable flow that moves the contents of `r1` from the fetching Fleet and uses it as an input	`independent flow codebag j(r1) {`
	`  ...`
	`} j;`
		

Table 4.2: Code bag descriptor syntax and representation.

any marked registers with the descriptor. This package is sent to the local fetch unit for dispatching.

The *BT* field identifies whether the referenced code bag is dependent, independent, or independent-flow cacheable. Finally, the *CORE_TYPE* field specifies which type of Fleet core in a heterogeneous multicore Armada that the described code bag is designed to execute on. The hardware uses this field to map the code bag onto an appropriate core at run time. If the descriptor describes a dependent bag, the *CORE_TYPE* field is ignored; in this case, the type is implicit as dependent bags must execute on the same core as their predecessor bag.

The fetch Ship

Software fetches a code bag by sending a descriptor for the bag to a special fetch Ship in the Fleet. Every Fleet core is required to contain one of these Ships. The fetch Ship first looks at the descriptor to see if any register values should be forwarded. The descriptor and any forwarded register values are packaged as a code bag fetch request to the Armada hardware.

The fetch Ship communicates the request to a centralized fetch unit, described in the next chapter, which then fetches the code bag. If the bag is dependent, the instructions are sent to the same Fleet that issued the request. If the bag is independent, the hardware finds a free Fleet core of the appropriate type as encoded in the descriptor's *CORE_TYPE* field and sends the instructions and forwarded registers there. Finally, if the bag is an independent cacheable flow, the hardware will attempt to find a free, cached instance of the flow and only forward the register values. If it cannot find a free instance of the flow, it can load the instructions and registers into an appropriately-typed Fleet core as it would when fetching a normal independent code bag.

Register forwarding

In addition to being used as temporary data storage, the register file is also used to forward arguments to successor code bags. By pushing all forwarded state in a Fleet into this one area, the hardware can store context state easily providing it with flexibility on when and where to execute successor bags. The proposed use of the register file and the encapsulation of state have the following requirements and implications:

1. Code bags that fetch independent successor bags must pass any arguments for those successors through the register file. If successor bags do not require arguments, the register file does not have to contain data. If there are more arguments than registers available, some of the arguments may be written to memory and a pointer to that set of data passed using a single register.

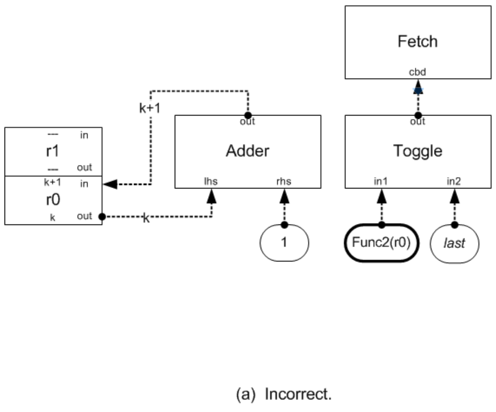
2. Register values can be copied or consumed as they are passed to independent successor bags. For example, a constant that is reused and passed to many code bags may simply be copied while a disposable index variable may be consumed as it is transferred. Unused registers are ignored. The desired forwarding option is selected individually for each register by marking bits in the descriptor of the successor code bag.

3. Independent code bags (including cacheable flow code bags) that require input from their predecessors receive these arguments in their local register file; the hardware will populate the register file with the forwarded arguments provided by the predecessor. The predecessor and independent child code bags they fetch must cooperate on what data is forwarded and how it is forwarded — which registers are used and what data each register holds. The Armada Procedure Call Standard, described in chapter 8, defines a contract that ensures all compliant code will interface correctly.

Registers have two states, empty and filled. A read from an empty register will block until that register is filled with data. A write to a filled register will block until the existing register data is consumed and the register made empty.

When the fetch Ship receives a code bag descriptor that requests register forwarding, the Ship waits for the specified registers to enter the filled state. At any point when all specified registers are filled, it may fetch the next bag. Software must therefore ensure that the registers are filled or will be filled with the intended data prior to fetching the successor. The fetch Ship will read forwarded registers at a single time; if some forwarded registers are filled yet others are still empty, it will wait for the remaining registers to be filled before reading any of the values. This behavior implies that forwarded registers can be read from and written to freely until any time where they might possibly all reach the filled state at once.

Figure 4.4 shows two examples of a code bag attempting to call $Func2(k+1)$. k has been forwarded into `r0` prior to this code bag's execution. Figure 4.4(a) contains a race condition between the readout of k from the register file and the fetch of `Func2`. If k is read and consumed from `r0` prior to the fetch unit receiving the code bag descriptor for `Func2`, the operation is correct. However, it is also possible for the code bag descriptor to reach the fetch unit before k is read from the register resulting in an incorrect call of $Func2(k)$. Figure 4.4(b) corrects this hazard by ensuring that k is read out of the register file and that `r0` is empty prior to releasing the code bag descriptor for `Func2`. The join Ship waits for both inputs to arrive at ports `in1` and `in2` then passes the value from `in1` through to its output. If the Func2 descriptor reaches the fetch unit before $k+1$ is written, the fetch unit will block until `r0` is in the filled state before consuming the argument and passing $Func2(k+1)$ forward for execution. The toggle Ship passes the input from `in1` before passing the data at port `in2`. The fetch unit will accept *last*, software's indication to the hardware that the Fleet is clear and ready for reuse, only after `Func2` has been fetched.



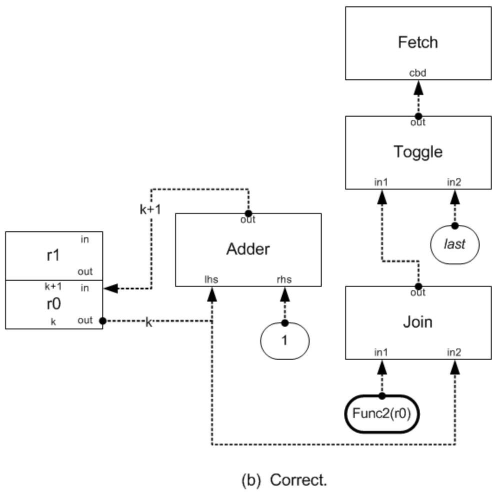


Figure 4.4: Passing a variable to an independent successor bag. (a) incorrectly attempts to call $Func2(k+1)$. Since `r0` initially contains a value and is in the filled state, it is possible for the fetch of `Func2` to occur before k is read out of the register file. This results in an undesired call to $Func2(k)$ and stray state in the Fleet after termination. (b) shows a correct implementation that waits for k to be read out of the register file before fetching `Func2`.

In many cases, some register values may be consumed by a successor bag while other values are copied. Figure 4.5 shows a code bag designed to draw a horizontal line to a frame buffer. The code bag calls the *paint* function for constant row I over columns 0-2047 with the constant color C.

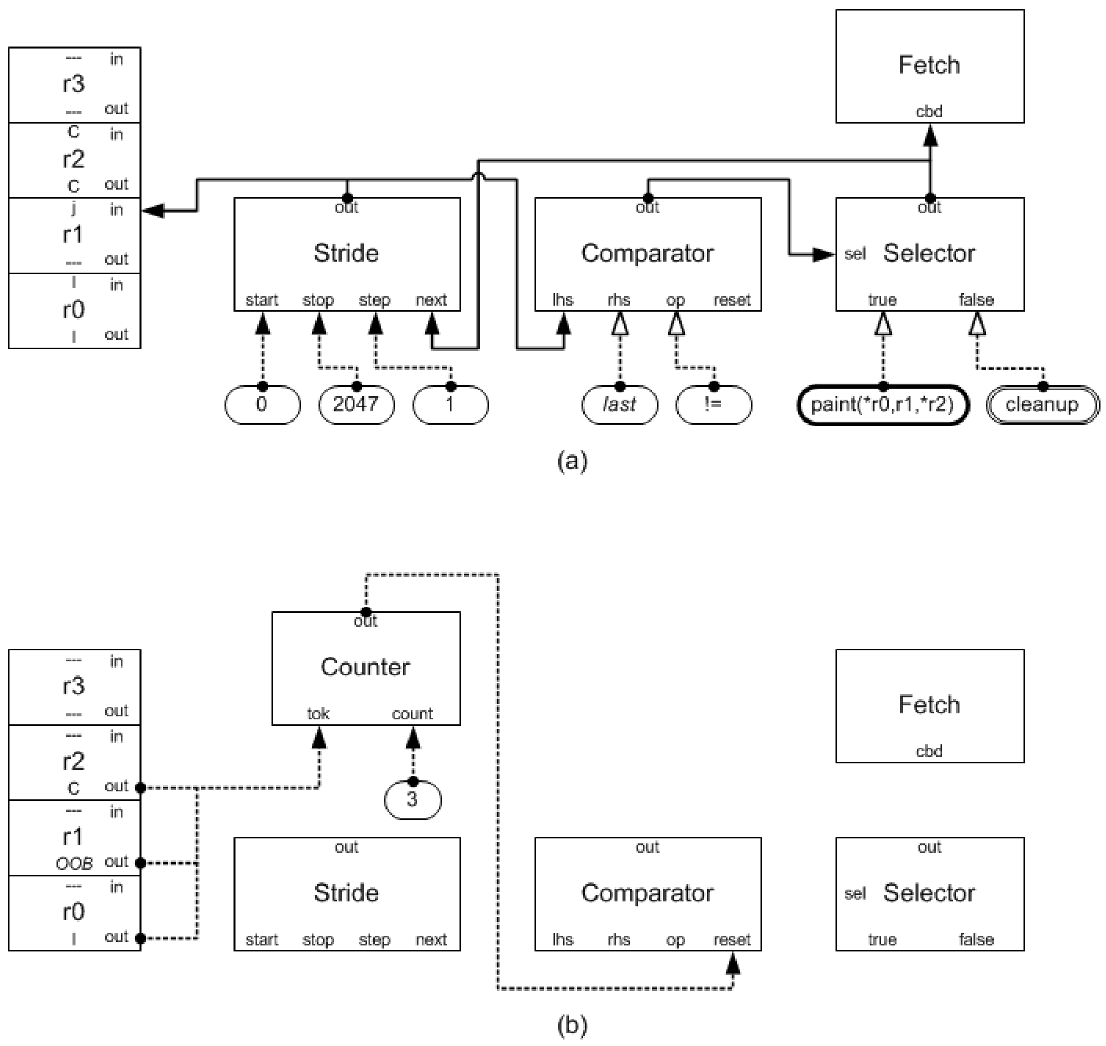


Figure 4.5: Painting a horizontal line. (*a*) shows a code bag designed to paint a horizontal line pixel by pixel by calling the *paint* function. The code bag is passed the row, I, and the color, C, in `r0` and `r2` respectively. The `paint` code bag requires that the arguments *row*, *column*, and *color* occupy registers `r0`, `r1`, and `r2` respectively. (*b*) shows the corresponding cleanup code.

In the line drawing example, the row I and the color C are constants received as arguments to the code bag shown in figure 4.5(a). The stride Ship[4] produces the column numbers to paint which are sent to both `r1` and the comparator. Given a start value s, a stop value t, and a step size i, a stride Ship generates a series of outputs:

$$output_x = s + xi \quad \{\forall x \mid 0 \leq x \leq \tfrac{t}{i}\}$$

The `paint` code bag descriptor is fetched for all columns in the specified range. Note how the descriptor for `paint` makes copies of the constants I and C. These values are sent as arguments to `paint` but left in the line drawer's register file for reuse. The column number in `r1` is consumed by each call to `paint` and subsequently filled by the next column number from the stride Ship. Once all columns are painted, the dependent cleanup bag is fetched. `cleanup`, shown in Figure 4.5(b), empties the registers and subsequently frees the Fleet core for reuse.

4.3.5 Freeing Fleet cores

When a thread terminates, it must free the Fleet core it is running on so it may be reused. Software must ensure that the Ships are free of state[5] and that the switch fabric is clear of tokens. Once the core is back in its reset set, software releases the core by sending a *last* token to the fetch Ship's code bag descriptor input, `cbd`. Once the fetch Ship receives this token, it signals to the Armada hardware that the core is free and may be used to execute other threads.

4.3.6 Virtual pipelines

Virtual pipelines connect multiple Ships together using standing instructions. The number of instruction fetches are reduced, and, ideally, single-thread performance improves by sending tokens through the pipeline quickly without potentially saturating and deadlocking the switch fabric. Virtual pipelines are one of the rare constructs in Fleet and Armada where the sequencing of data is guaranteed; data that enters a pipeline will never be overtaken by data subsequently pushed into that same pipeline.

[4]Originally proposed in [Sut05]

[5]This topic is discussed in section 4.3.7.

As previously mentioned, Armada does not have a dedicated flow control mechanism for virtual pipelines. In this research, a regular standing instruction between some input trigger port on the first Ship in the pipeline and the last Ship output port in the pipeline regulates the flow of data to avoid switch fabric deadlock. For example, figure 4.6 shows how a virtual pipeline can be used to connect a stride Ship that is producing sequential addresses to a memory Ship that will write 0's to clear the contents of those addresses. In Armada, the stride Ship will only generate one output at a time. Receiving any data on the `next` input port of the Ship will trigger the production of another output. In the example, a standing instruction connects the memory Ship's `write_complete` output port to the stride Ship's `next` input port. The stride Ship waits until the write is complete before generating the next address.

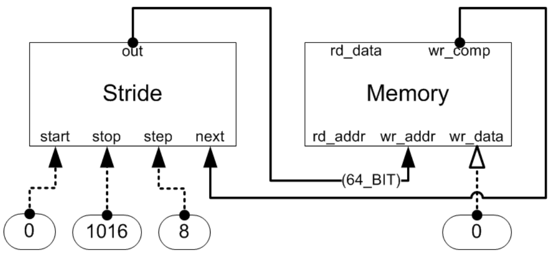


Figure 4.6: Virtual pipeline example. These instructions will clear 1,024 contiguous bytes of memory starting at address 0. `64_BIT` is a mini-opcode encoded in the *move* instruction that expresses the size of the data to write.

Software must ensure that tokens are not generated more quickly than they can be consumed by the slowest Ship in the pipeline. As Fleet does not impose any timing requirements on Ships, software cannot make any assumptions on when or where tokens may possibly pile up in the pipeline. In Armada, the only way to guarantee tokens will not saturate the switch fabric is to ensure that all of the outputs generated directly and indirectly from the initial input token set reach the end of the pipeline prior to inserting any new inputs.

Without a flow-control fabric, this simple pipeline implementation in Armada is not ideal for performance as the latency of the switch fabric is fully exposed unnecessarily. In the previous example, the `write_complete` token takes one trip through the switch fabric before triggering the stride Ship to produce the next address. The next address must also take a trip through the fabric to reach the memory Ship. Thus, the latency from requesting the next address to receiving it is at least twice the latency of the fabric. In the general case, the period between tokens in a virtual pipeline is linearly proportional to the total length of that pipeline; only one data set may flow through it at once.

The tokens flowing in a virtual pipeline in this implementation of Armada are effectively not pipelined at all. Therefore, the primary benefit of virtual pipelines in this first design is the reduction of instruction fetches attributed to the use of standing instructions. A flow-control system can considerably increase the throughput of a virtual pipeline by allowing multiple input data sets to flow through the pipeline at once. As previously mentioned, this system is left as future work.

4.3.7 Handling state in Ships

Some Ships retain state for lengthy periods of time and even indefinitely. This state retention can be very useful as it reduces the amount of data that must be communicated through the switch fabric. Without a hardware reset capability, software must ensure that all state-retentive Ships are in the reset state before releasing the core for reuse. Ships may have internal state like the stride and counter Ships, and they may have persistent inputs like constants that are reused in calculations indefinitely.

Software forces a Ship to clear its state in different ways according to Ship type. Arithmetic Ships accept a persistent input on at most one of their input ports at a time.[6] The persistence is cleared by sending a *last* token to the other input port. The memory Ship behaves in a similar way. For example, a persistent constant may be sent to the write address port to repeatedly address a first-in-first-out buffer (FIFO), or a constant may be sent to the write data port when repeatedly writing a specific pattern to a region of memory as in figure 4.6. The persistence is cleared and the Ship reset by sending *last* to the non-persistent input.

Other Ships may have an input port that does not accept persistent inputs. For example, stride Ships do not accept persistent inputs on `next` input ports, and selector Ships do not accept them on `select` input ports. Sending a *last* token to one of these ports will consume the data on the other input ports.

Finally, some Ships like the comparator Ship provide a special `reset` port for

[6]This constraint is in place for good reason. If both inputs to a multiplier were persistent, for example, the unit would repeatedly generate the same output and flood the switch fabric.

the sole purpose of clearing Ship state.[7] In Armada, it is reasonable to compare two tokens to determine if they are both *last*. Whereas other Ships typically react to *last* tokens by generating a *last* result, the comparator will produce a normal boolean result. The `reset` port provides an escape from standing instructions and clears persistent inputs by consuming all tokens at the input ports.

Upon receipt of a clear signal by any of the three means described, a Ship will wait for all relevant inputs to arrive before clearing those input tokens. For example, a comparator Ship that receives a reset directive will not reset until valid inputs are in place on its `lhs`, `rhs`, and `op` input ports. Likewise, an arithmetic Ship that receives *last* on its `rhs` port will wait indefinitely for an input to appear on its `lhs` port before freeing the Ship. A *last* token sent to a memory Ship `wr_addr` will cause the Ship to wait for a matching input on its `wr_data` port; however, it will *not* wait for an input at its `rd_addr` input as this input port is not relevant to a write operation.

Waiting for the other input tokens before clearing the Ship is required to avoid a race condition between the Ship clear signal and the input tokens arriving at that Ship. If a Ship only clears inputs that have already arrived, another potentially complex mechanism would be required to intercept tokens in-flight in the switch fabric destined for that Ship.

For similar reasons, a Ship that has received a clear signal will only remove one token per relevant input port. Software must track and handle cases where several inputs are queued up at one port.

Once a Ship is clear, it will generate a final *last* token on the relevant output port. Again using the memory Ship as an example, a clear signal sent to a write input port would cause a *last* token to appear at the `wr_comp` output port, but no token would be generated at the `rd_data` output port. This *last* token indicates that the Ship has been successfully reset and will also break down any standing instruction connected that port.

4.3.8 Context synchronizers

Software will often issue many threads and need to ensure those threads are complete before continuing with other tasks. Programmers and compilers interact with Armada's context synchronizers to provide this barrier control.

[7]The comparator is the only Ship that behaves this way in Armada.

A program requests a synchronizer by sending an independent code bag descriptor for the post-barrier successor code and any forwarded register contents it requires to the context synchronizer Ship. This step is similar to fetching an independent code bag. Additionally, the program sends the synchronizer Ship a count indicating how many threads must complete before fetching the successor code. The synchronizer Ship responds with a unique reference for the synchronization request. At this point, the program can fetch independent code bags for all of the threads it wants to spawn passing them the synchronizer reference as an input. This parent thread then frees the core it is running on.

The child threads execute, and, just prior to terminating, each thread sends the synchronizer reference received from its parent to the `decrement` input of the local context synchronizer Ship. When the Armada hardware determines that the last child thread associated with that reference has completed, it sends the post-barrier independent successor code bag and forwarded register contents to a free Fleet core for execution.

4.3.9 Flow caching

Cacheable flows consist of instructions and data cached onto a Fleet core. The flow can be applied to different data elements over time reducing instruction fetches and amortizing the setup cost of any virtual pipelines and persistent Ship inputs.

Flows are generated from specially designed independent code bags that must:

1. Use one or more forwarded input data elements from the register file

2. Indicate readiness for the next set of inputs when the current computation is complete by passing a special code bag descriptor to the fetch Ship

3. Maintain a predictable flow over multiple sets of input arguments delivered through the register file; the state of the Fleet core before operating on a set of valid input data should be equivalent to the state of the core after operating on the data

4. Cleanup and free the allocated core when any input data set is encountered that irreversibly alters the flow through the system such that condition three is not met

5. Cleanup and free the allocated core when all the input arguments are *last*. The hardware uses this method to evict any cacheable flow on demand.

Cacheable flows are fetched like any other independent code bag. The bag descriptor has a bit which identifies it as cacheable. Unlike threads created from independent code bags, flows do not generally reset the underlying resources when they have finished working on a set of data. Instead, they ready themselves for another input data set. Whereas threads will signal hardware once the underlying Fleet core is guaranteed to be back in the reset state, flows signal hardware when they are ready for the next data set. Additionally, hardware may need to evict flows when the Fleet cores they occupy are needed for other purposes. As there is currently not a core reset function implemented in hardware, flows must be carefully defined by software to self-destruct and release the occupied resource upon receiving an indication from the hardware. If the hardware needs to reclaim a Fleet used by a flow, it fills all of the input arguments in the register file of that Fleet with *last* values and waits for the flow to release the core.

Flows must indicate to the hardware when they are ready to accept new inputs. Therefore, it is necessary to introduce a way of differentiating between indicators of when the resources are freed versus when the flow is ready for the next set of arguments. Like other code bags, flows indicate resources are freed by sending *last* to the fetch Ship's `cbd` input port. Flows must additionally indicate when they are ready for another set of inputs by passing a special code bag descriptor, distinguished by a unique BT field (recall figure 4.3), to the `cbd` port. Hardware will not reuse a flow until it has received this ready signal.

Unless hardware signals a flow to destruct, flows ideally set themselves up for the next input data set when they complete their work. However, if an exceptional set of inputs arrive that breaks down the flow making it difficult to reset for the next data, a flow may choose to clean up all resources and release the underlying Fleet core; flows have no obligation to always ready themselves for more data, though that is their most useful behavior.

One example where a cacheable flow may prove useful is in the color conversion from the red, green, and blue color space (RGB) to the luminance and color difference space (YUV) commonly found in many image and video compression algorithms. A conversion operation, defined by equation 4.1, is performed on every pixel of an image.

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.299 & -0.587 & 0.886 \\ 0.701 & -0.587 & -0.114 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{4.1}$$

The flow defined by the code bags in figure 4.7 performs this color space conversion. Transports marked with a minus sign pass a mini-opcode indicating to the destination Ship that the value should be negated prior to performing the arithmetic operation. If any of the RGB color inputs are *last*, the flow will clean itself up and terminate, freeing the Fleet core as in figure 4.8. Note that this implementation assumes either all or none of inputs are *last*.

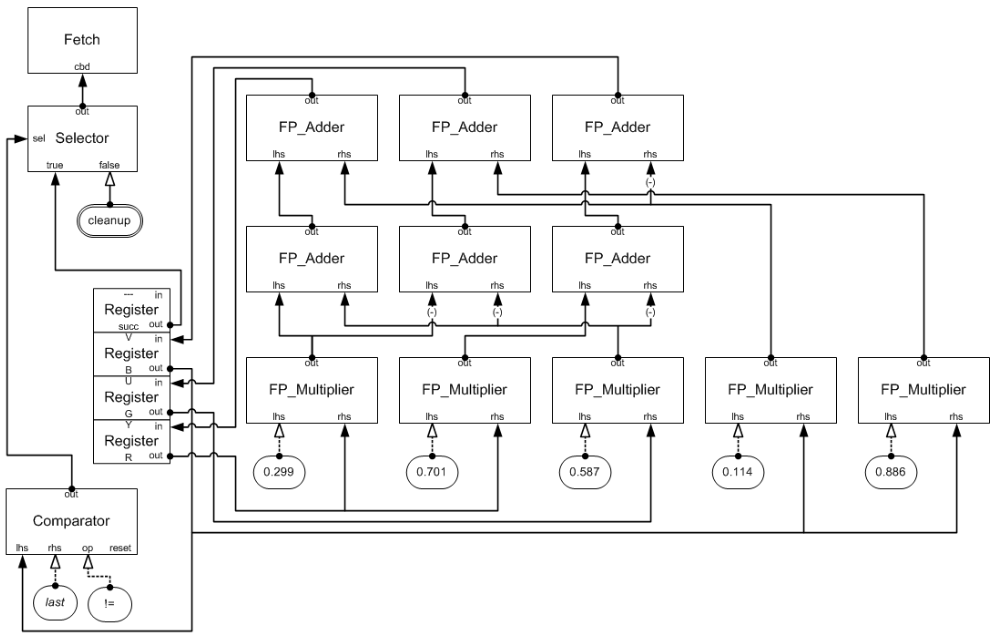


Figure 4.7: Color conversion cacheable flow.

4.3.10 Hardware reset behavior

At hardware reset, Armada will automatically fetch a special code bag from address 0. This initial code bag must only have one move instruction inside of it which software programs to fetch the code bag of reset handler code. In that reset handler bag, software can configure the system and launch programs or an operating system.

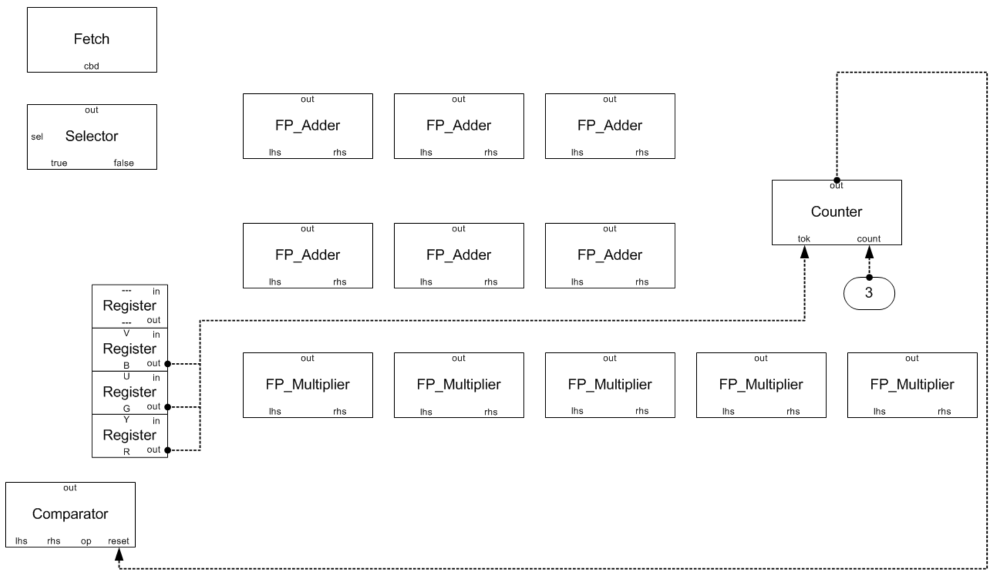


Figure 4.8: Color conversion cleanup code.

4.3.11 Event handling

Traditional architectures have interrupt handlers that react to events caused by I/O devices, on-chip monitors, software, and other sources. Armada supports such events with event handlers. Special registers in an Armada processor are set aside for each handled event. Software programs these registers at reset with independent code bag descriptors for the hardware to fetch in the case that the event occurs.

To prioritize the handling of such events over normal program execution, the OS may assign a Fleet core a special $CORE\_TYPE$ that only event handler code bags use. Thus, normal program code will never occupy those cores increasing the likelihood that the event handling code is assigned to a free Fleet immediately.

4.3.12 Fleet prototype core ISA

Armada is intended to support a heterogeneous mix of Fleet cores. Some cores may have Ship compositions ideal for handling complex program control flow. Others may aim to efficiently run demanding floating-point programs or to perform specialized functions like video encoding.

For this research, only one type of Fleet core was created with the purpose of being able to execute all general-purpose programs. There are many different Ship compositions that would achieve such a goal. This particular configuration was largely influenced by the functional unit requirements of the first hand-coded benchmark written for Armada (described in chapter 6); no effort was made to create the "best" Fleet core for general-purpose computing.

The Ships are fully connected by the switch fabric in this core; every destination port is reachable from every source port.

4.4 Related work

Although Fleet and Armada are very unique in their own right, they do share similarities with some aspects of other types of architectures. This prior work is described in the following sections.

4.4.1 Transport-triggered architectures

Transport-triggered architectures, or TTA's, also have a single *move* instruction [CM91]. However, all known TTA designs maintain use of the program counter and also use synchronous logic. The compiler or low-level programmer must therefore manage the scheduling of instructions such that pipeline and result latencies are accounted for. Additionally, TTA operations are *triggered* when an input is sent to a particular trigger port of a functional unit. Conversely, in Fleet a unit is enabled when all of its activation inputs are present. In TTA's, a fixed number of *move* operations may occur in parallel by packing them into a single instruction word as in VLIW architectures; however, Fleet's code bags are variable length and allow the programmer to express much larger amounts of instruction-level concurrency.

4.4.2 Dataflow machines

Fleet functional units are activated in dataflow fashion enabled by the asynchronous interfaces of the units to the switch fabric. The resulting behavior, though similar to the class of dataflow architectures derived from work by Dennis [DM75], is different in several ways. First, in pure dataflow, instructions are encoded with a type of operation to perform and a destination instruction to forward the result of that operation to. In Fleet, no operation is explicitly specified as that is a consequence of where the data is sent. Additionally, Fleet does not require that the destination address of an operation's result be encoded in an instruction. Thus, unlike pure dataflow, Fleet requires control instructions to pick up data at the output of functional units and to forward that data to its destinations. Fleet also allows side-effects, changes to program state anywhere in a function other than the function's return value, which are not allowed in pure dataflow. Therefore, in addition to being easily targeted by a functional language compiler typically associated with dataflow machines, Fleet is more easily targeted by compilers for ubiquitous imperative languages such as C, C++, Java, and Fortran which often contain such side-effects.

4.4.3 WaveScalar architecture

The WaveScalar architecture from the University of Washington has similar aims to Fleet and Armada. It attempts to confront growing design complexity and communication delays. Like Armada, it is strongly dataflow oriented, abandons the program counter, and has traditional memory semantics [SMSO03]. Unlike Armada, WaveScalar is synchronous. WaveScalar originally focused on exploiting data-level parallelism. In WaveScalar, data are tagged uniquely for each iteration of a loop allowing the same instructions to operate on different iterations of a loop at once. This behavior is very similar to dynamic, tagged-token dataflow machines of past [GKW85]. In Armada, instructions are also tagged allowing the same functional units to execute entirely different instructions on data from different threads. Armada may benefit from dynamic dataflow-like DLP extensions in the future. Proposed enhancements to WaveScalar, specifically the SpMT WaveCache, aim to also exploit thread-level parallelism including speculative threads [PWD+09]. This behavior is supported by a transactional memory and hardware threading extensions.

4.4.4 Independence architectures

Unlike programs for dataflow architectures which encode the *dependences* between instructions, programs for independence architectures indicate which instructions are *independent* from each other [RF93]. Fleet is in this class of independence architectures as Fleet compilers place sets of independent instructions that may execute concurrently into code bags. Some other members of the class of independence architectures include VLIW processors and the Horizon processor.

VLIW processors typically have fixed-length instruction words that contain several smaller, concurrently-executable instruction slots within it. Each instruction slot in a long instruction word corresponds to a particular pipeline and set of hardware resources. If a set of hardware resources cannot be used at a particular time, the corresponding slot is filled with a no-op instruction. Fleet code bags do not require this space-consuming waste. Horizon programs encode a lookahead field within memory-access instructions indicating how many subsequent instructions may be executed before the current instruction must complete [TS88]. Unlike the potentially very large code bags in Fleet, Horizon programs may only specify up to seven independent instructions as concurrently executable.

4.4.5 SCALP and Vortex asynchronous processors

Both Endecott's SCALP and Fulcrum Microsystems' Vortex processor share many similarities with Fleet [End95][Lin07]. The functional units in both processors have result and operand ports interfaced with an interconnect that may also bypass any register file and communicate directly with other functional units. The units share similar asynchronous connections with the interconnect and, therefore, also exhibit dataflow behavior like Fleet. However, the hardware in both designs is responsible for ensuring program-order execution of instructions like in traditional superscalar processors.

The Vortex units may also store state internally as in Fleet. Vortex has two classes of instructions of which one is equivalent to the Fleet *move* instruction including a vector or count field though lacking a form of standing instruction. The other class of instructions provide operations for functional units to perform. In Fleet, either the destination ports and destination opcodes dictate the behavior of the functional unit or an operation may be sent to a functional unit's port directly by encoding the function to perform as data within the standard *move* instruction. Also unlike Fleet, Vortex has a sophisticated branch unit that handles high level control-flow constructs which are handled with primitive Ships in Fleet.

4.4.6 TRIPS architecture

The Fleet and Armada architectures are similar to EDGE instruction set architectures that perform explicit dataflow graph execution [BG04]. The TRIPS architecture is one such implementation of an EDGE architecture developed at the University of Texas at Austin. The synchronous TRIPS architecture schedules hyperblocks of VLIW instructions that execute in dataflow-determined fashion [SNL$^+$03]. A program counter is used to jump between the hyperblocks. The dataflow-oriented hyperblocks behave similarly to Fleet's code bags, and instructions within a block are executed as soon as the required inputs are available. TRIPS can execute multiple hyperblocks simultaneously. In some cases, the concurrently-executing hyperblocks are speculative continuations of the active thread. TRIPS can also be tuned to support more general thread-level parallelism and execute hyperblocks from multiple threads concurrently. This behavior is similar to the threading extensions to Fleet proposed in this work. The TRIPS implementation is more restrictive than Fleet and ARMADA by requiring fixed sized hyperblocks, a fixed number of loads and stores per block, a fixed number of register file accesses per block, and a fixed number of outputs per block. Fleet and Armada have no such restrictions which makes code bags more dense.

4.4.7 Fleet at Sun Microsystems and U.C. Berkeley

Fleet continues to develop at both Sun Microsystems and U.C. Berkeley. As mentioned previously, this research took a snapshot of the Fleet architecture in 2006 and branched from the core group. The other group of researchers are investigating ways of programming individual Ships with small kernels of instructions. These instructions can be dynamically loaded and unloaded from Ships changing the behavior for different parts of programs. That research includes study into programming these systems with a new mid-level language above the level of move instructions.

4.5 Conclusion

This chapter described several enhancements to the Fleet architecture that enable further exploitation of concurrency. The Armada-1 microarchitecture discussed in the next chapter is the first design based on the Armada architecture. It contains hardware that exploits the thread-level concurrency described by Armada programs. Due to time constraints, further implementation and study of flow caching was not rigorously pursued and is left as future work.

Chapter 5

The Armada-1 microarchitecture

This chapter describes one possible hardware organization of the Armada architecture. This first Armada microarchitecture focuses on functional support for the thread-level concurrency extensions described in the previous chapter; cacheable flows are not supported. Processor performance was not a key consideration as exploring the programmability and use of the novel architecture took precedence.

5.1 Overview

Armada-1, shown in figure 5.1, is the first microarchitecture based on Armada. It is designed to exploit thread-level concurrency with minimal complexity. Armada-1 supports configurations of one, two, and four Fleet cores. These cores each have distinct switch fabrics and Ships, and they share a common fetch unit. As mentioned previously, only one type of Fleet core was created for the research; although the Armada architecture generally supports the integration of different types of cores, Armada-1 is a homogeneous chip multiprocessor.

5.2 Memory subsystem

The data and instruction memories and accompanying infrastructure in Armada-1 are relatively primitive and are not specified in a high level of detail. As described in the previous chapter, Armada-1 has a uniform access memory architecture with byte-addressing and 64-bit physical addresses. There is no virtual memory system. The instruction memory is only a cache that must be preloaded with programs; there are no other components in the instruction memory hierarchy

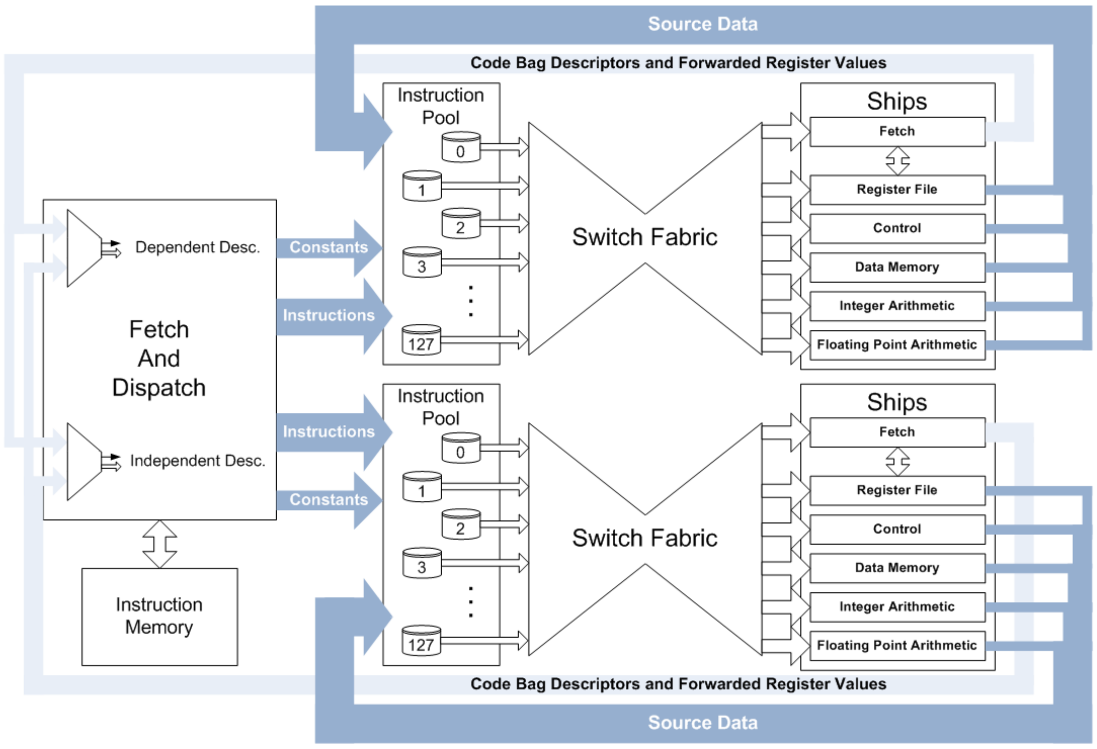


Figure 5.1: Block diagram of a dual-core Armada-1 processor.

for it to connect to. The data memory is also a flat structure that must be preloaded with any program data.

5.3 Fleet cores

Armada-1 integrates several simultaneously-multithreading Fleet cores together within a single chip. The cores receive instructions from the fetch and dispatch unit (FDU) through an instruction horn. Those instructions merge with data in the instruction pool. Once paired together, instructions carry data through a funnel-and-horn switch fabric to the Ship destinations. The Ships forward outputs to the instruction pool where that data merges with instructions, and the process continues.

5.3.1 Simultaneous multithreading

The Fleet cores in Armada-1 multiplex the use of resources across multiple threads. Fleet cores are designed to integrate large numbers of Ships, and there are times that instructions from a single thread will not use all of those resources at the same time. Inactive resources may still significantly contribute to the processor's leakage power consumption. Allowing multiple threads to simultaneously use available resources in the system has two very useful implications. First, the leakage-power cost is amortized across those different threads. Secondly, the hardware is able to support even larger numbers of threads without the addition of more Ships and an increase to die area. Supporting numerous contexts is particularly attractive as the Armada ISA encourages software to spawn many threads.

Data and instructions from different threads may be in the instruction pool or in-flight in different parts of the switch fabric concurrently. Likewise, different Ships may operate on data from different threads concurrently. Due to these characteristics, Armada-1's Fleet cores are considered to be simultaneously multithreading.

The cores in Armada-1 implement a tagging scheme, or coloring scheme, that allows the hardware to distinguish between instructions and data from different threads. The switch fabric preserves this coloring, reading colored data from the instruction pool and delivering colored data to Ship input ports. Ships will only operate on data at their input ports that have identical colors, and any output they produce will be marked with that same color. State-retentive Ships, such as the stride Ship discussed previously, must be capable of maintaining unique state for the full number of contexts supported by the microarchitecture. Instructions in the instruction pool, also colored, will only merge with Ship output data of the same color.

Each core effectively manages multiple virtual channels of instructions and data occupying the same physical wires and processing elements. One such virtual channel or color layer on a physical Fleet core is referred to as a virtual Fleet. Thus, an Armada-1 processor built with four physical cores each supporting 32 virtual channels has 128 virtual Fleets on which to execute threads. This scheme is analogous to commercial SMT cores that present themselves as multicore processors to operating systems.

5.3.2 Data size

In the original Fleet design, multiple values of out-of-band were possible. Although eventually *OOB* was limited to one value, *last*, Armada-1 has remnants of the original specification. Thus, the native data size is 96 bits to accommodate typical 64-bit integer and double-precision floating point data plus 32 bits to represent different possibilities of *OOB*. Code bag descriptors are also 96-bit and are handled as primitive data objects.

5.3.3 Instruction horn

The instruction horn feeds colored instructions from the shared FDU to the instruction pools within the individual cores (figure 5.2). Steering elements route instructions through the horn to reservation stations within the pool. Each steering element looks at one bit of the source address and steers the instruction toward the target station based on that bit's value.

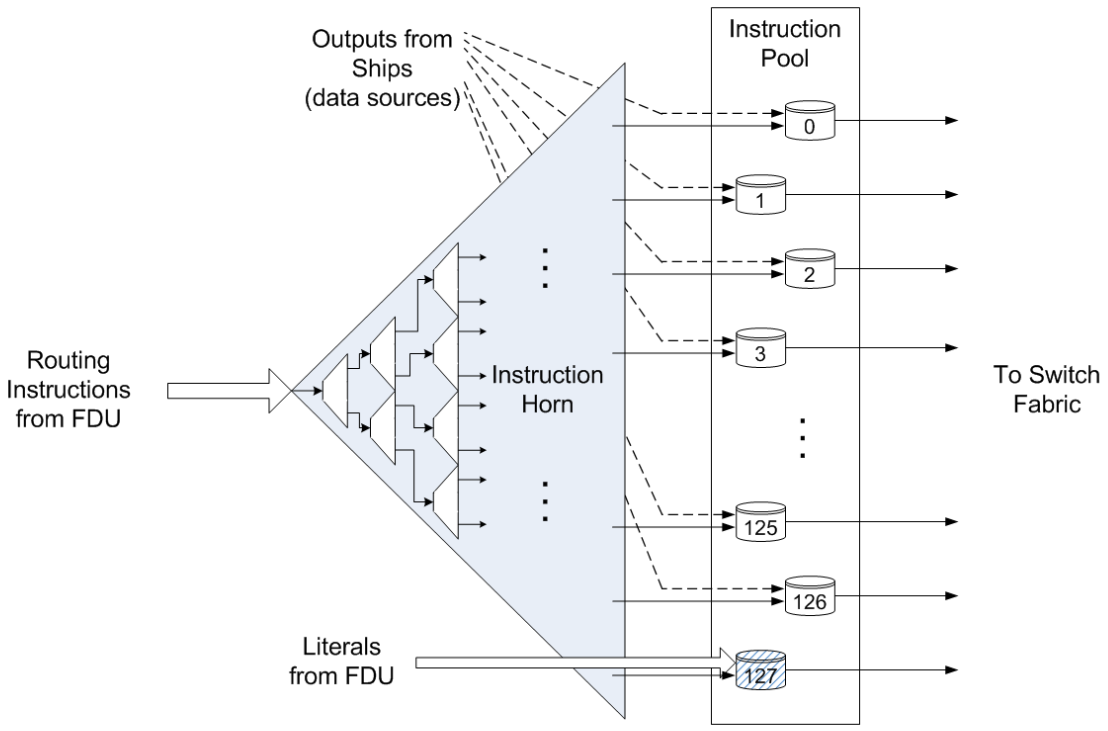


Figure 5.2: Instruction horn and pool.

5.3.4 Instruction pool and reservation stations

The instruction pool is an abstract place where executable instructions wait before entering the switch fabric. In the microarchitecture, this instruction pool consists of many reservation stations that merge instructions and data together into switch fabric payloads.

There is one reservation station for each Ship output port in a Fleet. The stations wait for port-to-port move instructions to arrive from the instruction horn and for data to arrive from the Ship output ports. Once each of these different inputs of matching color arrive, the station combines the destination addresses and mini-opcodes in the instruction with the data into a switch fabric payload.

After forming this payload, the reservation station either preserves or destroys Ship data and the instruction depending on the characteristics of the C and S bits in that move instruction (recall the move instruction opcode in figure 4.2(a)). If the instruction has the consume-source bit set, the station will erase the Ship data. Otherwise, it will transfer that same data when the next instruction with the same color arrives. If the instruction has the standing bit set and the data it merges with is not *last*, a copy of the instruction will remain at the station to merge with the next data of matching color produced by the associated Ship. Otherwise, the instruction is discarded after one use.

The Fleet cores handle constants generated at compile-time and the three move instruction variants that carry these values in a special way. Each core dedicates one reservation station to handling these constants. In the prototype cores of Armada-1, the address of this reservation station is 127 as shown in figure 5.2. Unlike the other stations, the constant reservation station's data input link is not connected to a Ship output port. Instead, this link is connected directly to the FDU. The FDU strips the constant from constant move instructions and sends it to the reserved reservation station via this connection. The FDU also converts constant move instructions into port-to-port move instructions; it replaces the constant value with the address of the constant reservation station. These converted move instructions traverse the instruction horn as normal. Thus, constants and their associated move instructions take different pathways to the reservation station. The constant and the instruction for multiple constant move instructions in a single code bag arrive in the correct order because each pathway guarantees in-order delivery of the payloads.

5.3.5 Switch fabric

The switch fabric and supporting structures move data from reservation stations in the instruction pool to Ship input ports. The switch fabrics in the Fleet cores of Armada-1 are directly derived from the experimental funnel-and-horn design of FLEETzero, a Fleet test chip developed by Sun Microsystems (figure 5.3) [CLJ+01]. The Armada-1 funnel uses arbitrated “demand merge” elements as described in the FLEETzero paper. These merge elements are arbiters that forward the first payload to arrive at either of its two input ports. The payload eventually arrives at the trunk of the switch fabric which passes it forward into the switch horn. The switch horn routes the payloads to their one or two destination Ships.

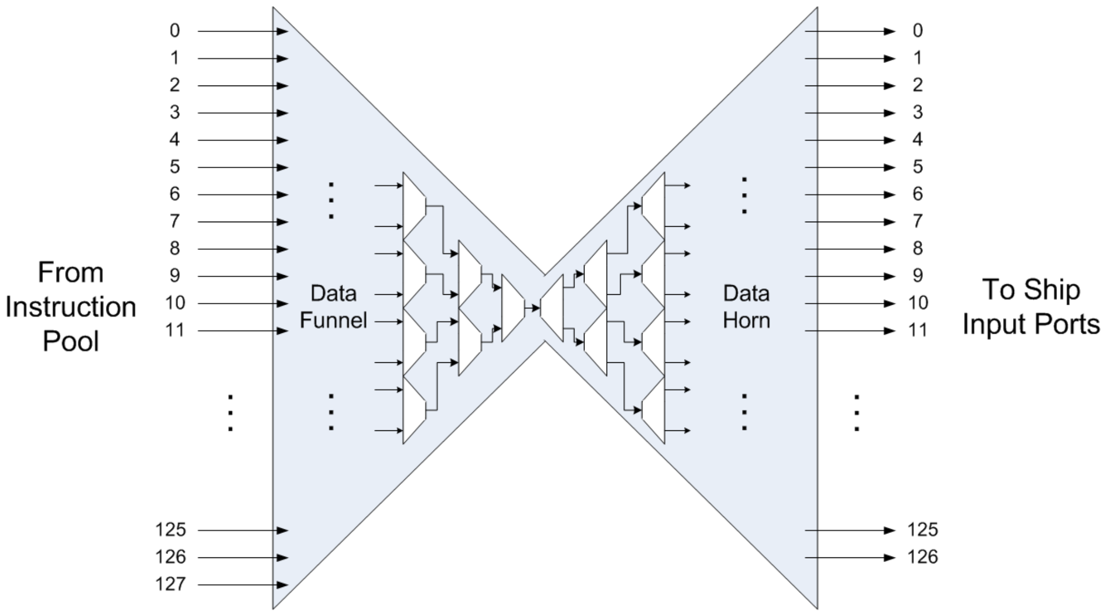


Figure 5.3: Switch fabric

5.3.6 Ships

Ships, the functional units in Fleet architectures, accept input data from the switch fabric and may forward Ship-specific output data to the instruction pool reservation stations. As previously described, a Ship may have one or many inputs and zero or many outputs. A Ship may also hold state established by previous inputs. Ships in Armada-1 may receive inputs of different colors in any order. The Ship is responsible for handling sets of data of each color independently from data of other colors. A Ship that carries state must therefore be capable of holding independent state for every possible color. Ships tag any output data they generate with the same color as the input data that invoked it.

Armada-1 consists of several classes of Ships. Control Ships allow programmers to express sequencing and changes in control flow. Bitwise-logic Ships perform bitwise operations on inputs. Integer and floating-point arithmetic Ships enable fundamental numerical operations. Memory Ships provide a means to communicate with external system memories. The fetch class consists of only one type of Ship that enables program branching and facilitates virtualization. The register class contains only type of register Ship used to temporarily store data and to hold forwarded code bag arguments. Finally, context-synchronization Ships provide a thread-merging capability. The full array of Armada-1 Ships and the port address mapping are described in appendix A.

5.4 Fetch and dispatch unit

The Armada-1 shared fetch and dispatch unit receives code bag descriptors from the Fleet cores and ultimately delivers colored instructions and constants to those cores. Internally, the FDU consists of two units, the code bag fetch unit (CBFU) and the instruction dispatch unit (IDU), as shown in figure 5.4. The FDU receives two sets of inputs from each of the different Fleet cores. Dependent code bag descriptors and independent code bag descriptors together with forwarded register values each go through different channels into the CBFU. The inputs from the different cores go through arbiters so that the CBFU handles only one request at a time per channel.

5.4.1 Code bag fetch unit

The CBFU has several responsibilities including:

- allocating virtual Fleets for independent code bags to execute on
- freeing a virtual Fleet when software signals that the resources are cleared
- providing the instruction memory with addresses of instructions to fetch
- forwarding register values to their destination virtual Fleet
- delivering constants to their destination virtual Fleet

The CBFU contains a Fleet allocation table that keeps track of the virtual Fleets that are in use. When an independent code bag descriptor arrives, the

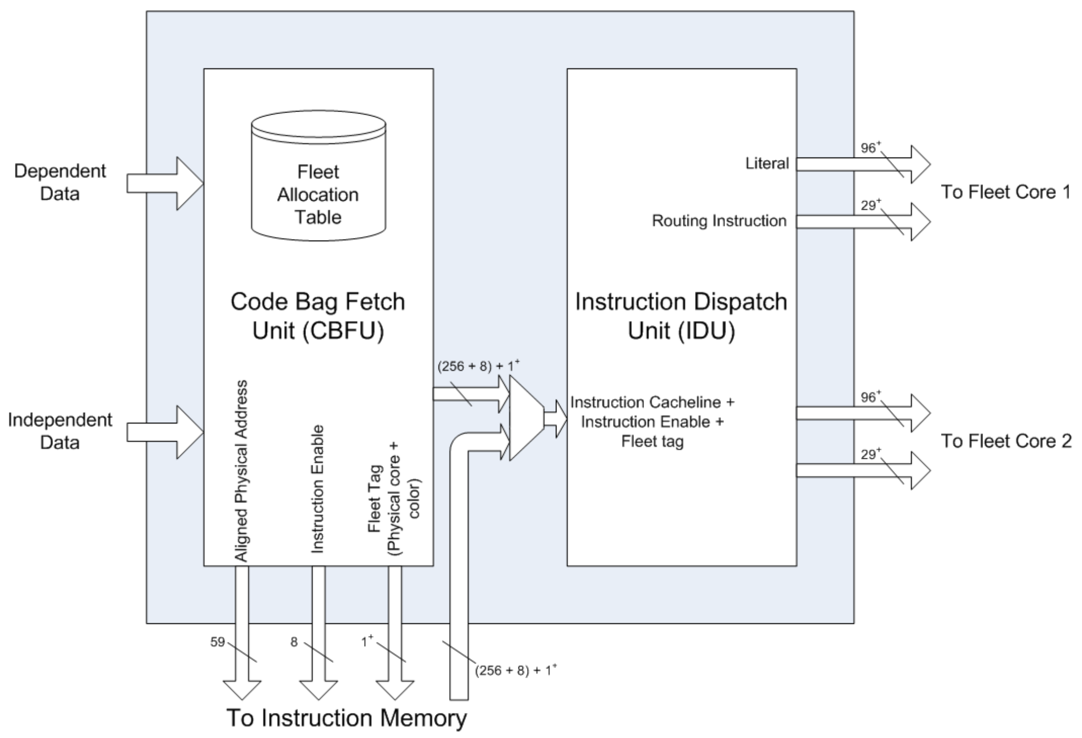


Figure 5.4: Fetch and dispatch unit.

CBFU attempts to find a free virtual Fleet to allocate for the bag. If found, that Fleet is marked busy, and the bag is fetched and routed to that Fleet. If all virtual Fleets are in use, the data channel delivering independent code bag descriptors stalls until resources are freed up. Because software may request thousands of threads at a time under normal conditions, the independent channel queue is backed by a buffer in main memory. Overflowing requests are written to main memory and subsequently fed into the CBFU by a direct-memory-access engine when the queue size decreases.[1] If the operating system sees that a program is fetching more independent bags than it or the system memory supports, the operating system may throw an exception similar to the stack overflow exception thrown in many modern general purpose computing systems.

Non-*last* dependent descriptors do not require an allocation table reference. Because a dependent code bag runs on the same virtual Fleet already allocated to the predecessor bag that fetched it, the dependent code bag descriptor queue is not subject to the blocking condition that the independent descriptor queue

[1]The independent code bag descriptor and forwarded register queue is not implemented this way in the simulator described in the next chapter. For simplicity, the queue's size in the model is unbounded; thus, no DMA-engine is needed.

may experience; resource-dependent code bag fetches can always be fulfilled.

The CBFU will free a virtual Fleet upon receipt of a *last* dependent descriptor originating from that Fleet. In this case, the CBFU must reference the Fleet allocation table. The Fleet allocation table handles requests from this queue separately from those made by the independent code bag queue; if an independent request to the allocation table blocks because there are no free Fleets, the table will continue to handle dependent code bag requests. These dependent requests will eventually free virtual Fleets thus allowing the CBFU to continue fulfilling allocation requests from the independent code bag queue.

When fetching a code bag, the CBFU extracts the start address and length of the code bag from the descriptor. The CBFU passes one or more 32-byte-aligned physical addresses and 8-bit instruction enables to the instruction memory indicating which instructions to fetch. The memory then delivers a 32-byte cacheline of instructions and the same 8-bit instruction enable data to the IDU. Additionally, the CBFU will pass the tag of the destination virtual Fleet where the instructions should be executed on to the memory. The tag contains bits identifying which physical core of resources the instructions are destined for and what color to tag those instructions with. By passing the tag and instruction enables to the instruction memory, the CBFU can fire and forget the requests. This decoupling gives the memory freedom to deliver instructions in any order. Cached instructions may be fetched and delivered immediately, and requests for instructions that must be fetched from farther away in the memory hierarchy can be initiated and queued for delivery later; the memory has all the information it needs to complete requests efficiently and out of order without help from the CBFU.[2]

The CBFU must also forward register values to virtual Fleets. The destination of values for an independent code bag is resolved when a virtual Fleet is allocated for the bag. In Armada-1, the CBFU forwards register values by constructing constant move instructions with the register value as the constant and the target register as the destination. The instructions are packed into a cacheline and delivered to the IDU with appropriate instruction enables and target Fleet tag. This interface is identical to the one that the instruction memory shares with the IDU. The constant move instructions hold the 96 bits of the register value in

[2]Recall that this implementation has only a single-level memory hierarchy. Therefore, this decoupling will not affect performance in Armada-1.

addition to 32 bits of routing information. Thus, only two of these instructions may be delivered in a single 256-bit instruction cacheline. If all eight registers require forwarding, for example, four cachelines are created and delivered to the IDU. Handling forwarded values by interacting with the IDU in the same way as regular instructions helps to limit the special-purpose hardware needed to deliver this data.

5.4.2 Instruction dispatch unit

The IDU receives instructions from both the instruction memory and the CBFU, and it outputs move instructions and constants to the specified virtual Fleets. The IDU receives a 32-byte instruction cacheline, an 8-bit instruction enable, and a target Fleet tag as one input data set. An arbiter merges the requests from the CBFU and the instruction memory into one stream; the delivery order does not impact correct behavior.

Upon receiving an input data set, the IDU extracts instructions based on the instruction enable bits, sequentially reading from the lower bytes of the cacheline and working up to the higher order bytes. If the instruction is a move-immediate type, the IDU extracts the immediate value, sign-extends it to a 64-bit integer, and then finally zero-extends the length to 96 bits. If the instruction is a move-constant type, the IDU interprets a number of the following bytes in the cacheline as data and not as instructions. The number of bytes needed to represent the constant is encoded within the instruction. Armada-1 does not allow move-constant instructions and the constant data they reference to extend across cacheline boundaries; the assembler must organize instructions in code bags so as not to violate this requirement.

Once the instructions and constants have been extracted, the IDU appends coloring information from the target Fleet tag to them. The instructions and constants are then sent to different queues based on the target physical Fleet core in the tag. As previously described in section 5.3.4, the IDU forwards constants directly to constant reservation stations within the target Fleet's instruction pool and converts constant-move instructions into port-to-port move instructions.

5.5 Context synchronizers

Context synchronizers allow a program's many independent threads of execution to merge before the program executes additional code. Each synchronizer Ship in the different Fleet cores is connected to a central context synchronization unit (CSU). That unit interfaces with main memory to store and update synchronization objects requested by the programs. These objects consist of an independent code bag descriptor, up to eight forwarded register values, and a 64-bit count. When software requests a synchronization object, the local Ship forwards the request to the CSU. The CSU allocates a block of memory for the object and returns a reference to it; in this design, the reference is the physical address of the object in memory. Local context synchronizer Ships similarly forward decrement requests to the CSU. The CSU looks up the reference and decrements the counter. When the counter reaches zero, the CSU releases the barrier by forwarding the independent code bag descriptor and any register value arguments specified in the synchronization object to the FDU for dispatching. The CSU then frees the memory allocated for the synchronization object.

5.6 Communication

Armada-1 components communicate asynchronously using a single-tracked, bundled data protocol. Each bit of data is carried by a single wire, high indicating logic 1 and low indicating logic 0. A single wire signals when the bundle of data wires is driven to the correct value. The sending unit pulls this line high once the data wires are driven to their correct values to send a request, and the receiving unit pulls the signal low once it has read the data lines to send an acknowledge. This signal has a delay matched to the worst-case delay of the data lines so that a request signal does not reach the receiver before the data lines reflect the desired value. Most bundles of data wires in Armada-1 are 96 bits wide to handle the transfer of data for all native data types at once.

5.7 Conclusion

Although several design trade-offs were made to complete this first design of an Armada processor within a constrained time limit, the end product is capable of functionally testing the independent and dependent code bag programming model, a key contribution of this work for exploiting thread-level concurrency in this one-instruction architecture. One of the methods introduced to enable concurrent computations through instruction and data coloring and through resource multiplexing will hopefully keep die area and leakage power to a minimum while achieving proportionally larger performance gains. The next chapter discusses how the Armada-1 microarchitecture was evaluated using the Mandelbrot benchmark.

Chapter 6

Evaluation of Armada-1

To evaluate the Armada architecture and, more specifically, the Armada-1 microarchitecture described in the previous chapter, a low-level simulator of the design was constructed. The Mandelbrot benchmark was developed using Armada features and evaluated on the simulator. This chapter discusses the simulator and benchmark implementations.

6.1 ArmadaSim

ArmadaSim is a timing-approximate, SystemC model of the Armada-1 microarchitecture. SystemC is an open-source modeling environment that provides C++ constructs for building standard modular hardware components[Ope06]. SystemC also provides a reference simulation engine for running the models. Many developers in the SystemC community work to enhance the models and to develop standards that maximize interoperability and reuse. Due to these efforts, supplemental tools are also becoming more readily available for visualization and power estimation. In addition to growing developer-community support and Institute of Electrical and Electronics Engineers (IEEE) standards approval, strong industry support is also a testament to the general acceptance of SystemC as a mature, robust modeling tool.

6.1.1 Structural configuration

The simulator may be centrally configured in an architecture definition file. The number of Fleet cores to instantiate and the number of virtual Fleets to support per physical core are defined there. The file contains definitions of the widths of commonly used data structures and buses. It also describes the types of Fleet cores to integrate into an Armada. After modifications have been made, the sources are recompiled to generate a new executable model.

6.1.2 Timing model

ArmadaSim simulates time required for inter-component communication and computation using a centrally-configurable timing model. Timing files can be interchanged to model different implementation choices such as fabrication process technology. The configuration points for simulation events vary in level of detail; some timing values approximate setup and hold times of local wires while others approximate more abstract behavior such as fetching a cache line from memory or performing an asynchronous multiplication operation. A minimum and maximum time value is specified for each configuration point. At run time, the simulator randomly chooses and applies a delay from this range at each occurrence of a timed event. For example, the simulator may choose different delays for the result latency of an asynchronous multiplier each time it is used. Such a timing model seems appropriate for asynchronous processor implementations in which completion times for tasks are often data-dependent. This dynamic timing model enables more robust verification and evaluation of Armada-1 than a static timing model could. However, the timing model applies variation randomly and does not consider real-world reasons for asynchronous timing variation.

6.1.3 Statistics gathering

ArmadaSim hardware blocks inherit from statistics-gathering interfaces defined in the source code which enable a variety of measurements. The current implementation provides an interface which enables the measurement of resource utilization over time. Other statistics interfaces may be easily added for future work. In addition to fine-grained measurement, total wall-clock time and simulation time are also reported.

6.1.4 Fleet state checker

Armada requires software to free all virtual Fleet state before releasing that Fleet resource back to the hardware. Due to large amounts of concurrently active operations, cleanup of the cores is not straightforward and is prone to errors without a proper methodology. As a debug tool, a state checker was built into the simulator. Whenever a program attempts to free a virtual Fleet, the checker verifies that each Ship is free of any state for the appropriate color. If stray state is discovered, the simulator terminates and indicates the error. The simulator provides a listing of the Ships that have residual state as well as the code bag descriptors responsible for delivering the stray tokens. A current limitation of the checker is that it does not catch stray instructions and data that are in-flight. This enhancement is left as future work.

6.1.5 Value-change dump output

SystemC natively provides useful facilities for generating value-change dump (VCD) files that can be imported into many waveform viewers. ArmadaSim supports this feature enabling users to view signals connecting both very primitive blocks and large composite structures. The signals are arranged in a hierarchy based on the block structure hierarchy of the architecture to ease signal selection for viewing. Studying waveforms is extremely useful for debugging Armada programs as well as the simulator.

6.1.6 Verification

Verification of the simulator was performed at many levels of the design from independent testing of primitive blocks to holistic verification of the complete system. At one extreme, each independent block was tested with mostly random inputs to input ports, and the generated output was verified. Blocks that retain state were tested with verification methods that demanded such state was reflected in the outputs over time. When coloring extensions were added to the architecture, inputs of random colors were driven into components at random intervals, and the outputs from the components were verified for each color. Each block was tested in this fashion as it was developed. Nearly all of the design bugs were found and remedied at this stage.

At the next level of verification, two or more of these primitive blocks were put together and tested as a larger component. The inputs provided to these blocks were more directed to likely possibilities, but randomness was introduced when possible. The instruction horn is one example of a block tested at this mid-level verification stage. Many arbiter blocks are connected together to form this much larger block. The horn was tested holistically by inputting random, valid payloads and then verifying that the correct payload reached the appropriate destination output. The few bugs that were found in this stage were mostly errors with how the units were interfaced together.

At the highest level of verification, the primary and mid-level blocks were put together to form a functioning Armada system. The most common bugs found at this stage were differences between the assembler used to generate executable binaries of the tests and how the implementation of the microarchitecture interpreted those opcodes. As Armada and Fleet were very fluid, changes were made to opcode formats regularly that were not always reflected in both the simulator and the assembler causing them to be out of sync. Additionally, some previously mentioned shortcuts were taken in order to complete the design more quickly; ways of handling these abstractions also had to be kept in sync between the pieces of software, and some discrepancies were discovered and resolved during this high-level validation stage.

6.1.7 Design complexity

A register-transfer level construction of the architecture was not developed, and the man-hours involved in that process are therefore unknown. However, some observations on the complexity involved in the implementation and verification of the Armada-1 simulator are worth consideration given the severity of the design productivity gap previously discussed in chapter 2. The complexity of the simulator development likely reflects the complexity involved in implementing an Armada-1 processor at the Verilog or VHDL level.

The modularity inherited from the Fleet architecture has proven to be extremely useful in decoupling the implementation of units from one another. As described in the previous section, the majority of bugs were found by testing units that were developed in complete isolation from one another. Most units, especially Ships, were simple to design and easy to test and validate. Additionally, Ship designs in particular took advantage of large amounts of reuse. The input and output port communication logic was implemented in a parent object that child Ship objects simply inherited from. Such reuse proved valuable as it kept the number of bugs in the colored-communications logic low and kept the means to fix the occasional error centralized. Finally, the total time required to implement the working Armada microarchitecture model in SystemC by the researcher was only three months including a SystemC learning curve. Although these observations may not quantitatively reveal the true complexity involved in realizing Armada microarchitectures in hardware,[1] they are strong hints that such complexity will likely be very low for the design and debug phases of the process. The decoupled behavior allows optimizations to individual units to be performed as deemed necessary without incurring the costs of affecting other units.

6.1.8 Limitations

ArmadaSim is fairly robust today having a decent feature set, good performance, and some focused validation. As previously mentioned, the timing model is useful for functional validation purposes but cannot be used to accurately reflect real-world performance with good detail. The simulator does not currently support the flow caching enhancement first described in chapter 4.

6.2 Mandelbrot

In order to test the ideas behind Armada, the execution of suitable workloads on an Armada system must be analyzed. Because this research is not looking into memory performance, ideal workloads for analysis favor large amounts of computation over memory accesses. The Mandelbrot benchmark is one such workload. The algorithm contains ample amounts of ILP, TLP, and DLP that capable hardware may take advantage of. Additionally, the control flow is non-trivial making coding of the algorithm for Armada an interesting challenge. Finally, this benchmark is accepted by commercial markets as one gauge of processor performance; it is used in the SiSoft Sandra benchmarking suite as a multimedia benchmark.

6.2.1 Computation

The Mandelbrot set is a fractal, an object or quantity that exhibits self-similarity at all scales[Wei06]. The set is computed from the equation:

$$z_{n+1} = z_n^2 + C \tag{6.1}$$

[1]Indeed, asynchronous tools currently lag behind the capabilities and robustness of synchronous ones.

with $z_0 = C$ for points C on the complex plane. Points that do not diverge toward infinity when entered into the equation are members of the Mandelbrot set; those that do tend toward infinity are not in the set. An approximation of whether C tends toward infinity may be made by testing whether z_n is greater than some limit bound, $LIMIT$, for $0 \leq n < ITER\_MAX$. Therefore, if z_{ITER_MAX-1} does not exceed $LIMIT$, the point is assumed to be a member of the Mandelbrot set.

Representations of the Mandelbrot set make interesting fractal drawings. An image resolution is chosen with dimensions $window.width \times window.height$. Each pixel is mapped to a point on a *viewport* of the complex plane. The *viewport* is characterized by its top-left corner, width, and height. In most drawings, points in the Mandelbrot set are colored black. Points not in the set are colored differently based on the final value of n, the number of iterations taken to reach $LIMIT$. Thus, points that tend toward infinity at different rates than others are painted different colors.

The benchmark creates this graphical representation of the Mandelbrot set. A window of some variable resolution is specified. The typical view port of the complex plane chosen for the benchmark is the range with top left corner at $(-2.5, 1.5)$, width of 4.0, and height of 3.0. Many points in this area are in the Mandelbrot set; therefore, the benchmark spends the majority of its time executing the innermost loop of the benchmark. Figure 6.1 shows a Java code snippet of the core of the Mandelbrot algorithm.

6.2.2 Parallelism

The Mandelbrot benchmark described has ample amounts of ILP, TLP, and DLP that may be taken advantage of by a highly concurrent architecture like Armada. Some ILP is present in the innermost loop of the algorithm where the large majority of operations are performed. The innermost loop body consists of nine fundamental arithmetic operations as depicted in the data flow diagram in figure 6.2. Independent operations can occur simultaneously if their inputs and enough hardware resources are available to handle the multiple operations. The diagram conveys a sense of time through varying lengths of operation boxes and the vertical space between them. A multiply operation is assumed to take twice the amount of time as an addition or subtraction operation; communication of each piece of data takes about half the time of an addition or subtraction

```
1  for (int i=0; i<window.height; ++i) {
2      double c_i;
3
4      c_i = ((double)i / (double)window.height) *
5             viewport.getHeight() + viewport.getMinY();
```

```
6      for (int j=0; j<window.width; ++j) {
7          double c_r, z_r, z_i, z_r_next;
8          int iter;
9
10         c_r = ((double)j / (double)window.width) *
11                viewport.getWidth() + viewport.getMinX();
```

```
12         z_r = z_i = 0.0;
13         for (iter = 0; iter < ITER_MAX; ++iter) {
14             z_r_next = z_r*z_r - z_i*z_i + c_r;
15             z_i = 2.0*z_r*z_i + c_i;
16             z_r = z_r_next;
17             if (z_r*z_r + z_i*z_i > LIMIT) break;
18         }
19         color_pixel(i, j, iter, ITER_MAX);
20     }
21 }
```

Figure 6.1: Mandelbrot benchmark inner loops in Java. In the translation to the Armada assembly language, lines 1–5 are mapped to the `Main` code bag, 6–11 to `OuterLoop`, and 12–19 to `InnerLoop` (as shown in figure 6.3).

operation. Though the exact timings may vary in an asynchronous system like Fleet, the diagram shows which operations are likely to occur simultaneously if sufficient resources are available.

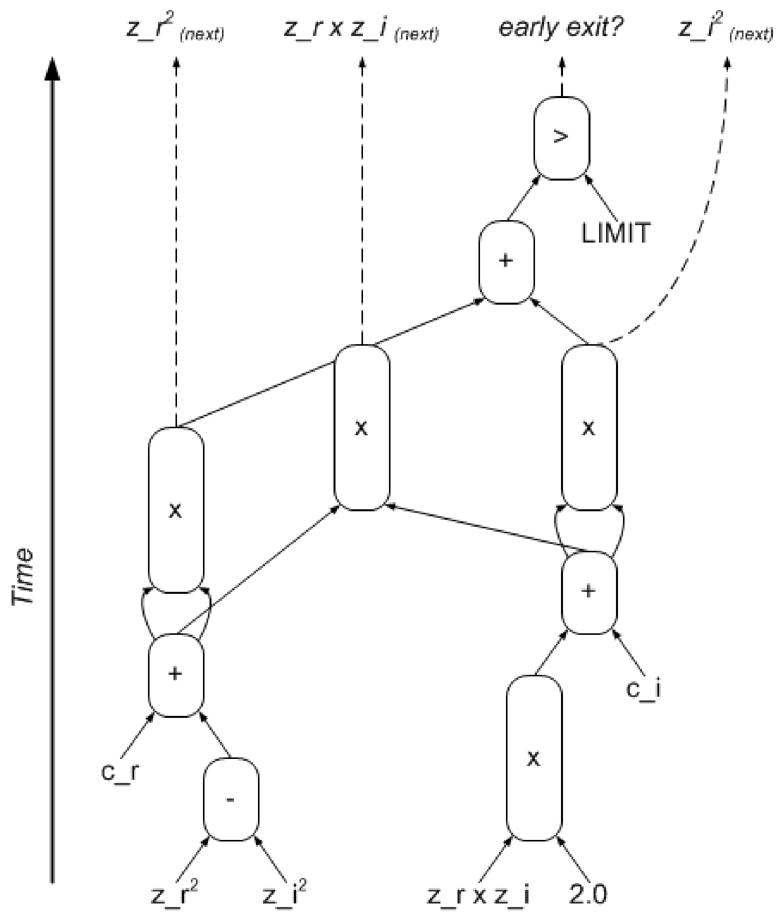


Figure 6.2: Mandelbrot inner loop operations. Many operations are likely to overlap in the computation-intensive inner loop of the benchmark. This ILP can be exploited by an architecture to reduce the total computation time of the loop body.

The algorithm also contains abundant amounts of TLP and DLP. The same set of operations are performed for every data element or pixel, a characteristic of DLP. Single-threaded architectures may exploit this parallelism by applying SIMD operations to multiple pixels at a time. Alternatively, architectures that efficiently support numerous threads may map each pixel to an independent thread of execution to take advantage of the TLP in the algorithm. Hybrid approaches of using the parallelism are also possible. For example, each row can be treated as a thread. Each thread may then use SIMD instructions to operate on a set of pixels within that row at a single time.

6.2.3 Mapping to Armada

In this section, the process of mapping the Mandelbrot benchmark to the Armada instruction set architecture is discussed. The most challenging aspect of the algorithm to map is the control flow. Using Armada's independent code bags to exploit thread-level concurrency is also of primary interest to the research and given special attention here. The Armada-assembly code for the benchmark is provided in appendix C.

Implementation structure

The Mandelbrot benchmark for Armada was hand-coded in the Armada assembly language. The final design is a sensible construction, but there was no rigorous optimization stage. Therefore, further improvements are possible. The core of the program is held in nine code bags, three independent bags and six dependent bags. Figure 6.3 shows the relationship between these code bags. Independent code bags have a dotted pattern on them, and dependent code bags are plain. The dashed arrows in the figure describe the fetch of independent code bags, and the solid arrows indicate the fetch of dependent code bags. The shaded regions indicate shared state; an independent code bag and all dependent bags that are fetched by it or any of its descendants share the same state. In Armada, all instructions from such a set of code bags that share state will execute on the same virtual Fleet.

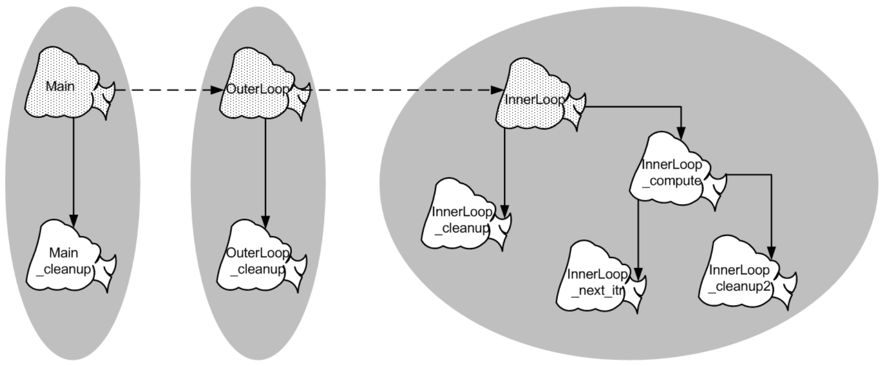


Figure 6.3: Mandelbrot code bags. Each shaded region identifies code that shares state and is executed on one virtual Fleet.

`Main` is the entry point to the program. `Main` fetches `window.height` instances of the independent code bag `OuterLoop` before finally fetching dependent code bag `Main_cleanup` to clean up and release the virtual Fleet it occupies. Each instance of `OuterLoop` fetches `window.width` instances of the independent code bag `InnerLoop` before cleaning up by fetching dependent code bag `OuterLoop_cleanup`. Thus, `Main` and `OuterLoop` work together to fetch `window.height` × `window.width` instances of `InnerLoop` — one instance for each pixel of the window. Each of these instances have independent state and may therefore execute on many virtual Fleets concurrently. Figure 6.4 illustrates this technique of releasing the `InnerLoop` code bags.

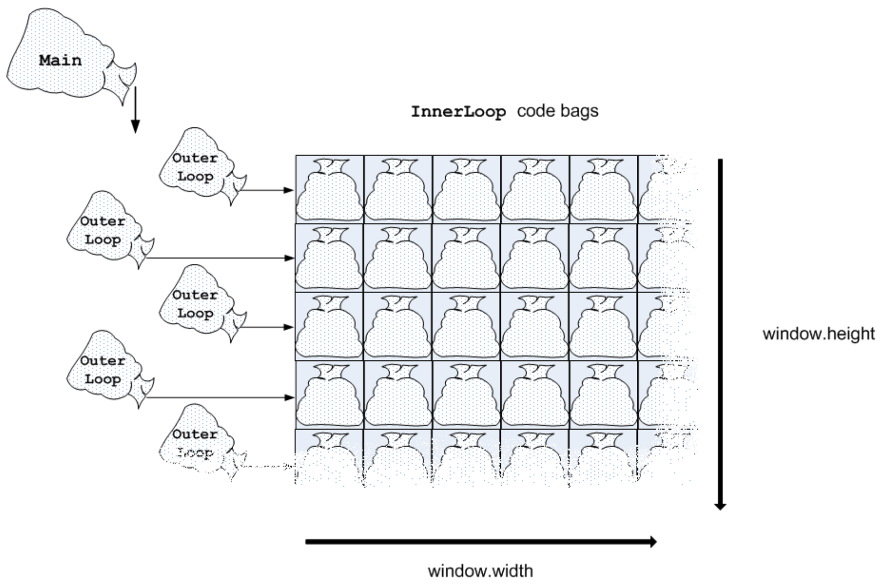


Figure 6.4: Mandelbrot code bag fetch scheme. `Main` fetches one `OuterLoop` bag for every row of the window. These bags may execute concurrently and will each fetch one `InnerLoop` code bag for every column in their row. All *window.height* × *window.width* number of `InnerLoop` bags may execute concurrently to color in each pixel in the window.

A typical count-controlled `for` loop with no early exits is implemented using a stride Ship as the counter, a compare Ship to perform the loop conditional test, and a selector Ship to choose between two sets of actions to perform depending on the result of the comparison. One set of actions would be to iterate the counter and repeat the loop body, and the other set would be to exit the loop and perhaps continue with additional behavior. This technique is applied in the simple example shown in figure 4.5(a). The `Main` and `OuterLoop` code bags work in this way.

Each `InnerLoop` bag must know the coordinates of the pixel it is operating on so that it can write to the correct address in the frame buffer. Therefore, `Main` and `OuterLoop` must minimally forward this information to the `InnerLoop` bags using Armada's register forwarding scheme. `Main` passes both the row number, i, and the imaginary component of the point on the complex plane to test, c_i, to the `OuterLoop` bags it fetches. Because c_i is the same for every column in any single row, the calculation need only be performed once per row. `OuterLoop` leaves these values untouched, forwarding them to each `InnerLoop` bag it fetches. `OuterLoop` additionally forwards the column number, j, and the real component of the point to test, c_r, to the `InnerLoop` bags.

Cleanup

`InnerLoop` applies the Mandelbrot set equation (equation 6.1) to the point on the complex plane forwarded to it in a tight loop. This `for` loop is more complex than the simple `for` loops handled by `Main` and `OuterLoop`; it must additionally manage an early exit condition. The count-controlled part of the loop terminates when $n = ITER\_MAX$. However, the loop will terminate early if the value produced by the Mandelbrot equation exceeds $LIMIT$. Exiting due to loop counter expiration leaves the Fleet in a state that is not equivalent to the state it is left in if this early exit condition causes termination. Because of the different residual states, two different code bags of cleanup instructions, one for each case, are required to free the occupied Fleet (figure 6.5).

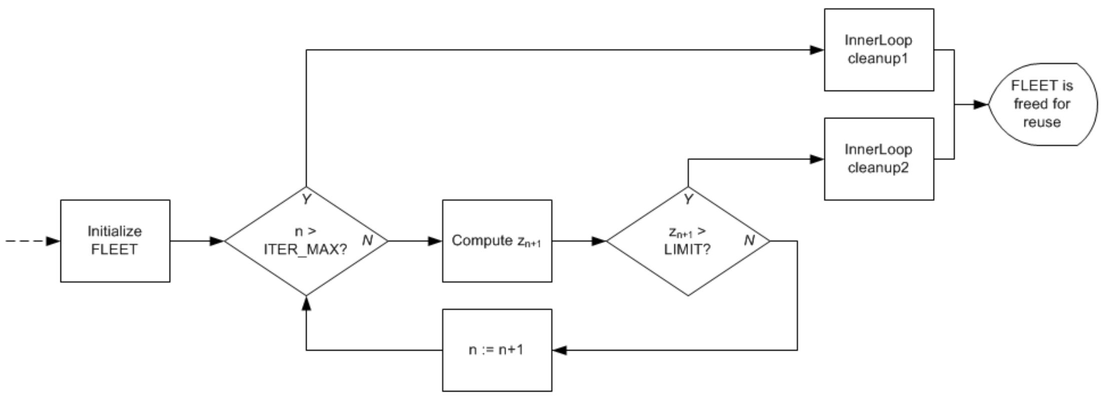


Figure 6.5: `InnerLoop` control flow.

Cleaning up a Fleet is not a trivial task, but the application of some useful techniques can simplify the problem. The process used to track and cleanup residual state consisted of maintaining state tables for each virtual Fleet. First, one state table is built. Starting with the independent code bag, each input and output port of every Ship is score-boarded as instructions reference them. A couple of cases require special consideration. Data may queue up behind write-persist data sent previously to the same destination. Additionally, standing instructions may forward data automatically for some time which must also be accounted for in the table. In the case of conditional branching behavior caused by the conditional fetch of different dependent code bags, the state table is duplicated so that one table is maintained for each branch; trees of branches result in trees of state tables. When a branch terminates, the state table identifies all residual state in the system that must be cleaned up. When a branch loops back to an already visited section of code, the difference between the current table and the table as it was when first entering that section of code is computed. This residual state must be cleaned up before looping back and re-executing that code. These tables were drawn by hand for the coding of this program, but, as described in part III of this thesis, such a process can be automated.

6.3 Conclusion

This chapter described the ArmadaSim simulator of the Armada-1 microarchitecture. The Mandelbrot benchmark was ported to the architecture and makes use of the thread-level concurrency and context synchronization enhancements described in chapter 4. The benchmark's large amounts of obvious parallelism and its low dependence on memory behavior made it an ideal test of Armada-1 given the current progress and level of detail in the design. The next chapter discusses the results from simulation runs of the benchmark on ArmadaSim.

Chapter 7

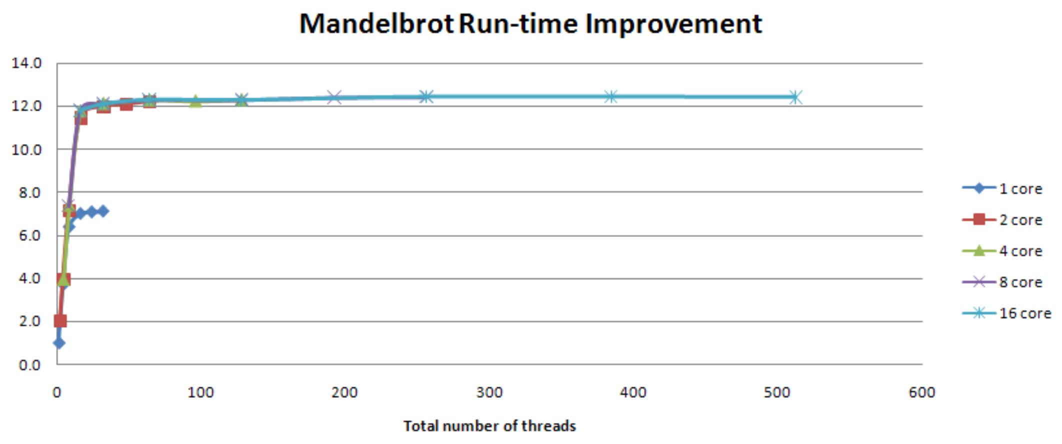
Microarchitecture results and discussion

The Mandelbrot benchmark described in the previous chapter was executed on different configurations of the Armada microarchitecture model. This chapter presents these results and discusses the ability of the microarchitecture to exploit instruction-level and thread-level concurrency. Static and dynamic code size is evaluated in comparison to existing commercial architectures. Finally, the chapter touches on the asynchrony aspect of Armada performance and performance modeling.

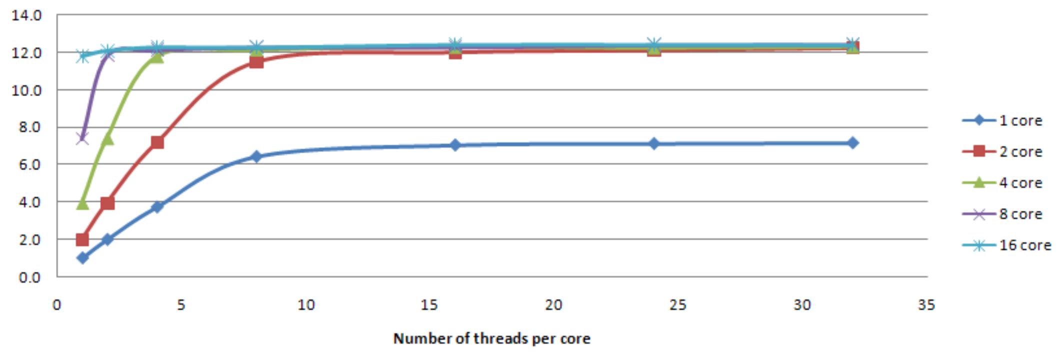
7.1 Performance overview

The Armada extensions supporting thread-level concurrency allow the Mandelbrot benchmark run time to scale with respect to the number of hardware threads. The compiled benchmark does not require modification to utilize additional Fleet cores present in an Armada. Figure 7.1 shows the speed up achieved by various configurations of the Armada-1 microarchitecture over the base case of a single-core, single-threaded implementation.

As shown in figure 7.1(a), Armada-1 achieves a greater than 12x performance improvement over a single, non-SMT Fleet core. A single-core, SMT configuration, denoted by the diamonds in the figure, is limited to less than an 8x performance improvement despite the number of simultaneously-executable threads supported. This data indicates the presence of performance bottlenecks within the core. Multiple-core configurations hit performance bottlenecks when



(a)



(b)

Figure 7.1: Overview of Mandelbrot performance on Armada-1. (a) shows performance improvement relative to the total number of threads supported by all cores in the microarchitecture. (b) shows performance improvement relative to the number of threads supported by each individual core.

16 threads are supported regardless of whether the multithreading is achieved through core replication or simultaneous multithreading on a single core (figure 7.1(b)). This data indicates a bottleneck in the units that are shared among all cores. Figure 7.2 examines the bottlenecks in detail for different configurations of a dual-core Armada-1 implementation.

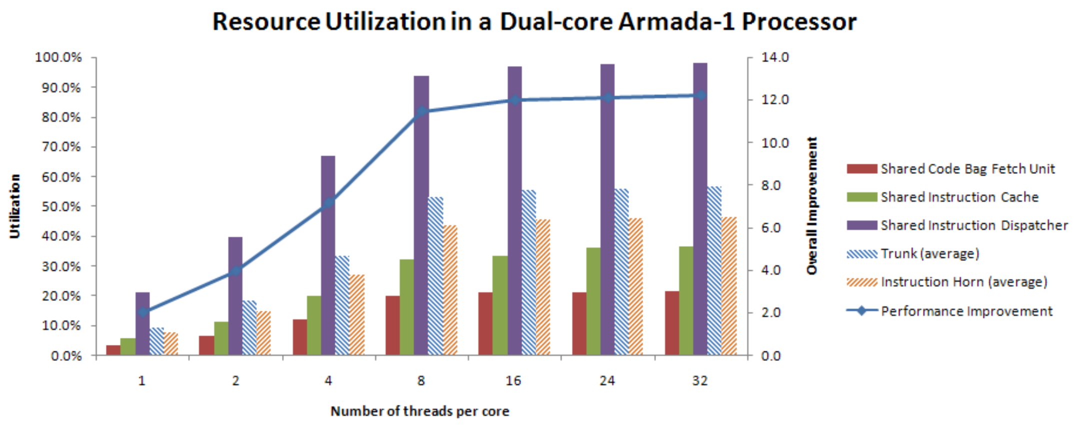
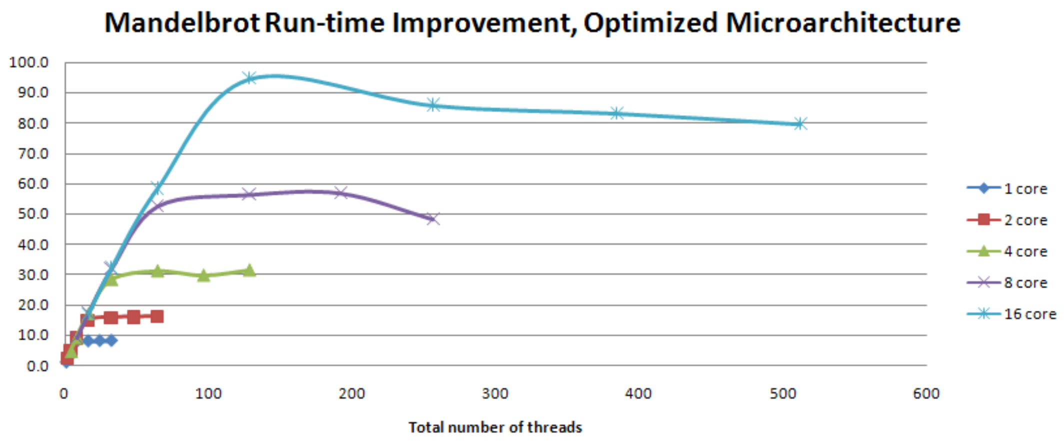


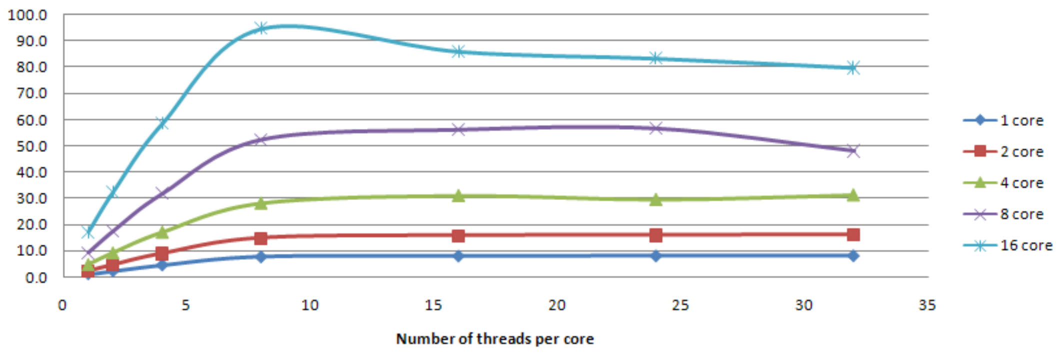
Figure 7.2: Resource utilization of performance-limiting units in a dual-core Armada-1 implementation. The trunk and horn utilization data shown is an average of the utilization measured on each core.

Using the utilization data collected from the simulations, the key core-specific bottleneck was identified as the trunk element of the switch fabric. The shared components that limit performance are the fetch and dispatch unit, instruction cache, and instruction dispatcher. It is possible to improve the throughput of these structures by modifying their implementation. Although these modifications are beyond the scope of this work, figure 7.3 shows the performance improvement possible if each of these components is effectively sped up by a factor of ten. The results presented in the remainder of this chapter come from simulating the benchmark on this hypothetical, optimized implementation.

With 16, 8-way simultaneously-multithreaded cores, the Armada-1 processor delivers a nearly 100-fold increase in performance over a single Fleet core. Supporting more than eight threads per core does not result in further performance improvement (figure 7.3(b)). In this optimized implementation, the primary bottlenecks are within the cores and no longer in the shared units. For the Mandelbrot benchmark, the floating point addition and multiply Ships are highly utilized and are the limiting factors of the performance.



(a)



(b)

Figure 7.3: Overview of Mandelbrot performance on a hypothetical, optimized implementation of Armada-1. This data reflects a 10x speedup in the fetch and dispatch unit, instruction cache, instruction dispatcher, and switch-fabric trunk element. (a) shows performance improvement relative to the total number of threads supported by all cores in the microarchitecture. (b) shows performance improvement relative to the number of threads supported by a single core.

7.2 Exploiting concurrency

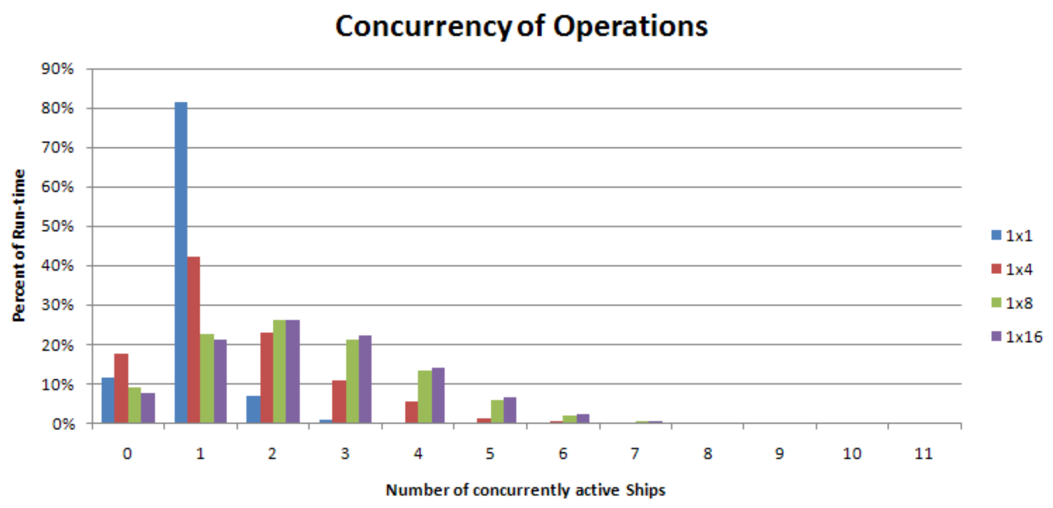
In Armada, instruction-level concurrency is discovered and represented statically by software. Code bags contain instructions that can be fetched and executed in any quantity and order. The amount of work that software requests the fetch and dispatch unit to do concurrently for a single thread is proportional to the number of instructions in the bag being fetched. However, the *execution* of move instructions is additionally dictated by data-flow ordering. Empirically, it is possible to determine how many operations are being performed simultaneously by counting how many Ships are active at any given time. Figure 7.4 shows the number of concurrently active Ships over the entire benchmark on single-core and 16-core Armada-1 configurations.

The single-core histogram shows that for non-SMT configurations, only one Ship is active at any time for the vast majority of the benchmark run time. The SMT behavior added to Fleet in Armada provides more work to perform concurrently ultimately providing an improvement to the benchmark run time. The 16-core histogram shows that the number of concurrently active Ships actually decreases when moving from an 8-way SMT to a 16-way SMT configuration. This pullback in the number of concurrent operations correlates with the decrease in overall performance between the configurations observed in figure 7.3. Possible reasons for this performance degradation are discussed in section 7.5.

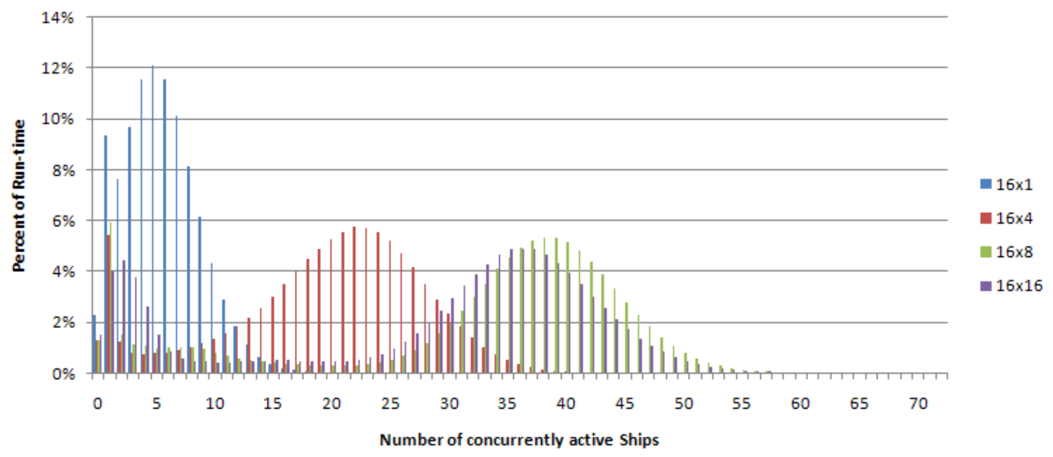
7.2.1 Instruction-level concurrency

The amount of instruction-level concurrency in the Mandelbrot benchmark that Armada-1 can capitalize upon is reflected in the amount of concurrent operations that take place in the single-core, single-threaded configuration shown in figure 7.4(a). This data accounts for the result latencies of the operations as well as data-flow dependencies. Over 80% of the time, only one operation is active at once, showing the necessity for fast single-threaded performance in the absence of exploiting other forms of concurrency.

Armada-1 is strongly limited in single-thread performance because of insufficient pipelining. As discussed in section 4.1, the Armada-1 switch fabric does not have a flow-control mechanism for pipelining standing instructions. The Ships are also not pipelined. Additionally, Armada-1 does not support branch prediction or data speculation, both of which can increase ILC. However, these deficiencies



(a)



(b)

Figure 7.4: Concurrently active Ships in Armada-1 running Mandelbrot. These histograms show the number of concurrently active Ships over the duration of the benchmark. (a) describes one physical core with one, four, eight, and sixteen-way SMT configurations. (b) describes a sixteen physical core configuration with each core having one, four, eight, and sixteen-way SMT configurations.

can be partially overcome by exploiting thread-level concurrency.

7.2.2 Thread-level concurrency

Like ILC, thread-level concurrency is also discovered and represented statically by software. The number of independent code bags that can be fetched concurrently determines the maximum degree of TLC the hardware may achieve. For the majority of tests performed, the resolution was limited to 32x24 pixels to control simulation time making about 770 threads available for concurrent execution. As seen in figure 7.4(b), many-core configurations like the 16-core ones shown enable large amounts of operations to occur concurrently. By having more cores available, core-local bottlenecks are mitigated. By increasing the multithreading capacity of each core, oft-idle units are more highly utilized thus lowering benchmark run-time with minimal hardware overhead. Workloads that do not use many threads will not benefit from Armada's TLC extensions.

7.3 Code density

Instruction set density, or code density, is a significant factor in system performance and cost. Dense instruction streams require less system bandwidth than sparse ones often resulting in better performance and lower power consumption. Instruction caches can be utilized more efficiently thereby reducing the frequency of costly off-chip memory accesses.

Armada generally requires more instructions than a traditional RISC architecture to represent the same computation. In Fleet architectures, the communication involved in computations is explicitly described by software whereas it is implicit in RISC and CISC architectures. Table 7.1 compares the code size and amount of instruction fetch traffic for three different architectures running the Mandelbrot benchmark.

The ideal number of physical Ships for a general-purpose Fleet architecture is related to the amount of concurrent operations possible in a program. In Armada-1, 7-bit addresses for both input and output ports were chosen somewhat arbitrarily to fill a 32-bit instruction. The composition of Ships in the cores was largely influenced by the requirements of the Mandelbrot benchmark. Future implementations should take into account the concurrency available in typical programs when selecting processing resources. An analysis of concurrently active

Architecture	Code size (bytes)	Num exec insts	Inst traffic (bytes)
MIPS	180	424,860	1,699,440
x86-64	200	582,467	2,228,524
Armada-1	756	761,072	3,401,640

Table 7.1: Code size and instruction fetch traffic comparison. MIPS code compiled with gcc v4.1.2 -O2 optimization. x86 code compiled with gcc v4.3.2 -O2 optimization. Armada version is hand-coded.

units, similar to but more detailed than the analysis described in section 7.2, would be useful in determining the quantities of different types of resources to put into a Fleet core. Additionally, the ideal number of bits needed for addresses could be optimized with the same procedure. Optimizing address size will reduce instruction size and instruction-related traffic.

7.4 Code bag organization

Fleet proposes that concurrently-executable instructions all be placed into a code bag. The order the hardware chooses to fetch and issue the instructions is out of software control. While this representation is useful in allowing hardware to fetch some non-contiguous instructions as they are available, software-provided hints on execution order can help to improve performance. Armada-1, for example, attempts to fetch instructions with lower physical addresses before fetching those with higher physical addresses from a code bag. If software was aware of this preference, it could place move instructions in the bag based on the level that the corresponding arcs appear in the data-flow graph. High-priority arcs at the lower, leaf levels of the graph would be positioned at the lowest addresses in a code bag, and arcs at the highest level of the graph would be positioned at the highest addresses. The difference in execution time of the Mandelbrot benchmark when move instructions are positioned in these two opposing arrangements for Armada-1 16-core configurations varied between 0.1% and 6.4% as shown in figure 7.5. However, somewhat unexpectedly, the instruction orderings that were considered more ideal did not always yield a performance improvement. For the 16-way SMT case, for example, the arrangement considered non-ideal actually performed 6.4% better than the other arrangement, the largest variance measured. The next section explores possible reasons for such behavior.

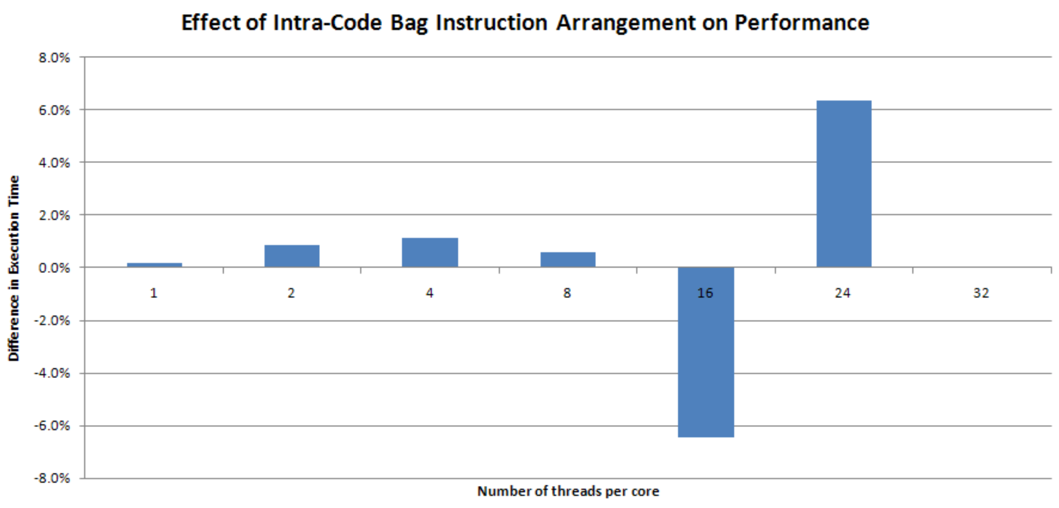


Figure 7.5: Effect of intra-code bag instruction arrangement on Mandelbrot benchmark performance. The improvement of the first-needed prioritized instruction arrangement over the last-needed, inverse-priority arrangement is shown.

7.5 Asynchronous system performance

For the Mandelbrot benchmark, a Fleet core's performance saturates and may actually decrease when more than eight contexts are active at once as shown in figure 7.3b. The Fleet funnel-and-horn switch fabric exhibits high occupancy within the funnel and relatively low occupancy in the horn. The speed of the trunk element limits the rate at which data can enter the horn. Merge elements within the funnel rarely have multiple inputs to propagate in the single-threaded case as shown by the low concurrency in figure 7.4. As more threads are allowed to occupy a single core, the funnel occupancy rises. When more than one input is present at a merge element, there is an additional delay incurred in the propagation of the second datum. ArmadaSim simulates a small delay required for the merge element to respond to an acknowledge from the successor element indicating that the forwarded data has been received. This reverse latency is hidden from the pipeline through that merge element if no other data is waiting to pass through it. Conversely, if another datum is at the merge element's input port, it must wait for that delay before the merge element processes it and moves it forward.

When more than 200 threads are active at once on any multicore, multithreaded configuration tested, performance begins to degrade further. This degradation, visible in the 8- and 16-core configurations of figure 7.3a, is caused by saturation of the shared units in the processor model. The fetch and dispatch unit and the instruction memory become overloaded reducing their overall throughput. Similar behavior has been observed and reported on in asynchronous pipeline and ring buffer experiments where throughput is limited by the availability of holes in the pipeline[MJC$^+$99, Wil94].

7.6 Resource multiplexing

Armada-1 explored just one approach of increasing resource utilization that only reaps benefits when many threads execute concurrently. An alternative way of both increasing utilization of Ships and reducing the size of the switch fabric is to allow some port addresses in move instructions to map to the same physical Ship input ports. Some of the bits in the address would route data to the appropriate location, and the remaining bits could disambiguate the contexts to which the data applies. For example, if 5-bit addresses are used, the lower three address bits could be used to route data to the correct physical Ship, and the remaining two bits could be used to identify the context to which that data applies. This scheme may decrease the number of Ships needed in a core and increase Ship utilization among instructions within an individual thread. It is very similar to the tagging or coloring scheme used to distinguish between thread contexts in Armada-1.

Additionally, the costs of implementing SMT using the virtual channels in Armada-1 may outweigh the simpler alternative of increasing the number of physical cores and allowing only one context per Fleet. The appropriate trade-off is largely dependent on the targeted process technology for the processor. The exploration of this design space is beyond the scope of this work.

7.7 Conclusion

This chapter demonstrated that the Armada architecture is effective at capitalizing on the thread-level concurrency available in a benchmark like Mandelbrot. Nearly a 100-fold increase in performance is realized over a single-core Fleet by replicating cores, employing simultaneous-multithreading using virtual channels, and by allowing software to spawn threads easily with Armada's addition of resource-independent code bags.

While hand-coding a benchmark for an unconventional processor like Armada is possible, compiling programs directly from a high-level language automatically is often much more difficult. The next part of this thesis shows that ubiquitous, imperative-language compilers can successfully target the Armada architecture. This capability provides further evidence that the Armada architecture may be used as the foundation for a general-purpose computer processor.

Part III

An Armada compiler

Chapter 8

Armada Procedure Call Standard

Compilers transform high level languages into machine code for the target architecture. The compiler as well as low-level programmers that write the native machine code must adhere to a procedure call standard to ensure that the code they generate will cooperate with other code libraries and modules that their code may reference. Although such standards are relatively straightforward though tedious, the Armada Procedure Call Standard (APCS) introduced here ensures both serially- and concurrently-called code using Armada's novel resource-dependent and resource-independent code bags can work interchangeably. Readers familiar with procedure call standards for traditional architectures may wish to skip to section 8.5 which discusses Armada-specific behavior.

8.1 Overview

There are four stages in a procedure call [BD95]. The first stage is the caller prologue where the caller prepares to invoke the subroutine. The second stage is the callee entry where the subroutine takes control over which instructions are executed. In the third stage, the callee exit, the subroutine relinquishes control back to the caller. Finally, the caller recovers from the procedure in the caller-epilogue stage.

As concurrency is an essential part of the Armada architecture, there are two procedure call methods in the APCS. The serial method is commonly used to call functions in parts of the program dominated by control flow code where actions are serialized. There is also a concurrent method of calling functions that may spawn one or more threads of execution to exploit thread-level concurrency available in programs. The APCS therefore describes the caller and callee function responsibilities for both the serial and concurrent cases. The following sections discuss these responsibilities. First, the behavior of the key resources involved in procedure calls, the register file and the stack, are described. Then the standard for how data is passed between caller and callee is described. Finally, the serial and concurrent methods of calling procedures are discussed.

8.2 Register file

Unlike traditional architectures, the purpose of the Armada register file is not solely to hold often-used values close to the processing elements; the file's primary purpose is to transfer arguments to execution contexts of independent code bags. A caller function will place arguments to the callee in the registers and fetch an independent code bag. That fetch operation will consume the values in the register file, package them with the independent code bag descriptor, and place the resulting independent context fetch transaction into a queue. The new context will eventually be allocated to a Fleet tile, the arguments loaded into the register file of that Fleet, and the bag of instructions executed.

8.2.1 Behavior

Armada registers are either $full$ of data or *empty*. The fetch unit will block[1] if an independent code bag descriptor cannot be fetched because one or more argument registers are *empty*. A bit-field in the code bag descriptor specifies the registers that must be full prior to executing the successor code bag. The register file may not be used at all when fetching dependent code bags that will execute on the same physical tile because data on any Ship output port may be accessed directly.

8.2.2 Listing

The Armada-1 architecture provides eight, 96-bit, general-purpose registers that have a role in calling procedures. These registers were designed to hold complete

[1]The fetch unit will appear to block from the software's perspective. The hardware may prefetch the instructions within the code bag onto a Fleet tile or perform other setup behaviors in anticipation of the register data arriving.

code bag descriptors as well as 64-bit data with a variety of OOB values. Register use depends on how the procedure call is invoked by the caller. $A0 - A4$ may hold arguments to pass to procedures. SSP may hold a software-allocated successor stack pointer to be used by a callee. LSP holds the local stack pointer for the currently active routine. LR holds the independent code bag descriptor of the code to execute after the callee completes. A full register listing is shown in table 8.1. The registers' involvement in the procedure calling convention is described in detail in subsequent sections.

Register name	**Alias**	**Description**
R0	A0	argument passing
R1	A1	argument passing
R2	A2	argument passing
R3	A3	argument passing
R4	A4	argument passing
R5	SSP	successor stack pointer
R6	LSP	local stack pointer
R7	LR	link register

Table 8.1: Armada-1 register file listing.

8.3 Stack

The APCS requires stacks to be full descending. In a full descending stack, the stack grows to lower addresses (*descending*), and the stack pointer points to the last element on the stack (the pointed-to memory is $full$). Thus, when a new item is pushed onto the stack, the stack pointer must first be decremented to allocate space for that item. The stack must maintain 64-bit alignment across procedure calls. Empty buffering space should be added if needed to maintain this alignment. Argument and return value passing are discussed in section 8.4.

8.3.1 Allocation policy

For the concurrent procedure call cases, a single stack cannot easily be shared among many threads. Therefore, the APCS accommodates dynamic stack allocation for callee threads. Many hardware and software mechanisms in the literature may be adapted to support this behavior [EO88][VH99][BMBW00]. A subroutine's stack may be allocated by software or by hardware. The two methods have different advantages and disadvantages as discussed in the following sections. The caller function decides which method to use on a per-call basis. The compiler or programmer can choose the best method to use for each call and thus use both models within a single library or program.

Software allocation

Software may allocate the successor stack directly and pass this pointer to the callee. In this model, the software must supply the successor stack pointer in the SSP register when fetching an independent code bag descriptor. Software determines how the stack memory is allocated. For example, the stack space may be allocated on the heap by a thread library or may simply be a copy of the parent stack pointer's current location.

Hardware allocation

Alternatively, the program can have the hardware create a stack dynamically. To decrease the amount of time required to allocate these stacks, hardware assists in the stack allocation process by maintaining a stack pool of pre-allocated and recycled stacks. The hardware, with operating system assistance[2], will provide a stack pointer automatically for the callee by populating the SSP register.

8.3.2 Choosing the allocation policy

The stack policy may vary for each subroutine call. In some cases, a shared, linear stack like those typically used in traditional architectures may not be adequate as several callees may be capable of running concurrently and may need to have their own independent memory. The Armada architecture's hardware support for stack allocation provides fast allocation and deallocation of stack memory for the concurrent execution of subroutines. However, if a caller must execute a single procedure call serially, the preferred method is to avoid stack allocation and release penalties by using the shared linear stack method and allowing the single

[2]Although the hardware still requires operating system assistance, an intelligent design will amortize the high software intervention cost by managing stack allocation and release in hardware when possible perhaps by using stack caching and recycling mechanisms. Armada-1 implements such a system.

successor subroutine to grow down from the end of the parent's stack. Amdahl's law tells us that the sequential segments of a program are the limiting factors to speedup in computer systems; so it is worthwhile to optimize the serial cases as much as possible.

8.4 Data placement

The caller and callee exchange data in a similar way for both serial and concurrent call methods. This section describes how the caller transfers arguments to the callee and how the callee returns values to the caller.

8.4.1 Argument passing

The caller prologue is responsible for passing input arguments to the callee. Registers are the preferred means of transport, and the stack is used when the size of the register file argument passing registers is inadequate.

Argument registers

The 96-bit registers $A0 - A4$ are argument passing registers. The argument passing registers contain the input arguments for the callee. 96 bits are enough to contain whole code bag descriptors as well as all other primitive Armada machine types with OOB values. The arguments passed via the argument passing registers are loosely packed such that each primitive type allocates an entire register regardless of its size. For example, five 16-bit characters will occupy $A0 - A4$ even though they could be packed into a single register. This restriction allows data to be passed from caller to callee without incurring packing and unpacking overhead. Arguments are passed in ascending order using the argument passing registers. Argument one is passed to the callee via $A0$, argument two is passed via $A1$, and so forth.

If there are more arguments than argument passing registers, the arguments overflow onto the callee stack. For example, if a callee takes nine integers as arguments, $A0 - A4$ would contain arguments one through five, and arguments six through nine would be pushed onto the callee stack. Data placement on the stack is discussed in a later section.

Aggregate types

Non-native aggregate types such as data structures passed to the callee with pass-by-value semantics may not fit in a single register. Such structures are passed over multiple registers if the entire unpacked structure can fit within the available registers. For example, if a callee has two parameters, an integer and a data structure comprised of two doubles, the integer would be passed in $A0$, the first double within the structure in $A1$, and the second double within the structure in $A2$.

Structures that are too large to fit *in their entirety* in the available argument passing registers are pushed onto the callee stack. For example, if a callee takes two arguments, an integer and a data structure comprised of five doubles, the integer would be passed in $A0$, the data structure would be pushed onto the callee stack, and $A1 - A4$ would be empty.

When there is a mix of non-native aggregate types and native types passed as arguments to a callee, arguments are positioned in the following way. First, the arguments are placed in the argument passing registers if possible regardless of whether they are native types or aggregate types. As described earlier, an aggregate type may not fit in its entirety in the available argument passing registers. If this is the case, it is skipped over for the remainder of the first pass. The remaining arguments are visited and placed into the argument passing registers if possible. Once the argument passing registers are full or only arguments of aggregate types that do not fit in the available argument passing registers remain, the unplaced arguments are placed in reverse sequence on the callee stack. This scenario is demonstrated in figure 8.1.

```
struct elephant {
    char var_array[1000];
};

struct mouse {
    int x, y, z;
};

void proc(int a, struct elephant b, double c, struct mouse d,
          int e, int f);
```

Figure 8.1: Argument passing example with native and aggregate types.

In this example, the argument for parameter a is placed in register $A0$. The aggregate-typed argument for parameter b is skipped over for now as it will not fit in the four remaining argument passing registers. c's argument is placed in register $A1$. Finally, the aggregate-typed argument for parameter d is placed in registers $A2-A4$ ($d.x$, $d.y$, and $d.z$ respectively) since it does fit in its entirety into the available argument passing registers. The remaining arguments are skipped over as there are no more argument passing registers available. Then, the unplaced arguments are revisited in reverse sequence beginning with the argument to parameter f and pushed onto the callee stack.

Overflowing to the stack

The callee stack provides additional storage for arguments that do not fit in the argument passing registers. Arguments are pushed onto the stack in reverse order. For the example in figure 8.1, the argument for f is pushed first followed by the arguments for e then b. Aggregate types for parameters such as b have each of their members pushed on the stack in reverse order as well. If $b.var\_array[999]$ is at address 3999 on the empty descending stack (byte-addressed, base 10), then $b.var\_array[0]$ is at address 3000. As a result, an object of an aggregate type residing in the heap, statically positioned in memory, or residing on the stack is accessed in the same way given a pointer to the base of the object.

8.4.2 Automatic variables

Automatic variables, or automatics, are variables that are created and destroyed as program flow enters and exits scoped portions of a user program. For example, variables used solely by a single function and defined within that function are automatic variables. Generally, automatics do not occupy registers but instead reside directly at Ship output ports as they are created. However, if an occupied Ship must be used for another purpose or the parent function otherwise needs to bank an automatic variable, it may store the variable onto the end of the stack as in traditional architectures. This is useful in many cases including in performing recursive function calls.

8.4.3 Return values

Callee return values are passed back to the caller on the stack. Although in the serial case it is possible to return small values back via one or more registers, this method does not work for the concurrent case where many callees may need to return values to a caller at once. Having the return values returned on the stack guarantees that the necessary storage for those values is available. Placing the values on the stack for both serial and concurrent cases guarantees code uniformity in accessing those values; it allows a single implementation of a function to be called serially or concurrently. If the APCS allowed different methods of returning values for serial and concurrent cases, functions that could potentially be called both serially and concurrently would have to be implemented two different ways in order to adhere to the standard. This replication would increase code size, software maintenance cost, and invite confusion for the programmer.

8.5 Procedure calls

The following sections discuss the caller and callee responsibilities for serial and concurrent methods of calling procedures. Particular attention must be paid to the handling of the two stack pointers. The local stack pointer (LSP) points to the last element on the stack for the actively running function. The successor stack pointer (SSP) points to the last element on the stack for the successor subroutine being called. When fetching an independent code bag, the Armada hardware will move the SSP into the LSP position for the called function.

8.5.1 Serial procedure calls

In a serial procedure call, the caller function jumps to a single callee. Upon termination, the callee returns control back to the caller. The process in Armada is very similar to the process in traditional architectures. The following sections describe caller and callee requirements for the four stages of the function call process.

Caller prologue

In preparation to call a function, the caller must make room for the callee's return value on the stack, put arguments into the appropriate registers or stack locations, and place a code bag descriptor for the callee to fetch once it has completed. Figure 8.2(a) shows the layout of the stack after the prologue has set it up for the function call. The caller passes the new location of the stack frame to the callee via the SSP register.

Callee entry

When fetching an independent code bag descriptor, the Armada hardware repositions the SSP value from the caller into the LSP position of the callee effectively setting up the local stack frame for the subroutine. The callee may allocate additional space on the stack for local, automatic variables as shown in figure 8.2(b). The callee can reference the arguments passed in by the caller from the register file or the stack as appropriate.

Callee exit

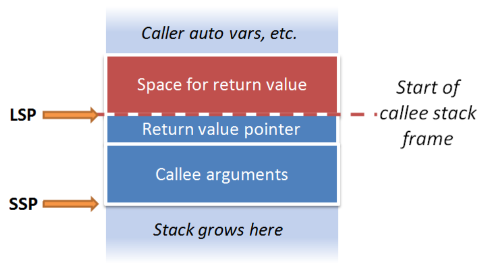
Upon exiting, the callee uses the return value pointer provided by the caller to store its result, if any, to memory. The successor stack pointer is incremented to remove the callee's arguments, automatic variables, and return value pointer from the stack as in figure 8.2(c). Finally, the callee fetches the independent code bag descriptor provided by the caller in the link register (LR) and terminates.

Caller epilogue

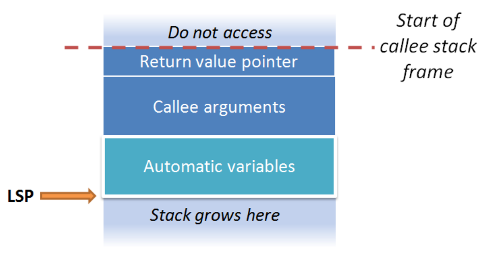
The callee fetches the caller epilogue code bag prior to terminating. This code bag may now access the callee return value. As with the caller prologue to callee entry phase, the hardware moves the SSP setup by the callee into the LSP position when returning to the caller as showing in figure 8.2(d).

8.5.2 Concurrent procedure calls

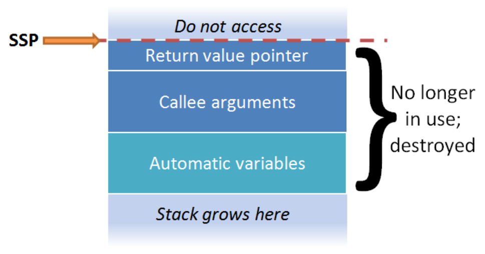
Spawning threads is designed to be easy with the Armada architecture. The APCS describes how procedures can be called to execute concurrently. Program flow can then later be resumed after a barrier point once the threads have completed.



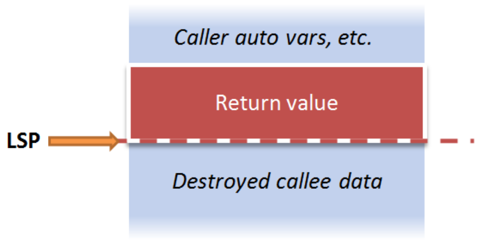
(a) Caller prologue.



(b) Callee entry.



(c) Callee exit.



(d) Caller epilogue.

Figure 8.2: Stack layout for a serial procedure call. The caller allocates space for the return value, if any, and passes a pointer to that memory and any arguments to the callee (a). The SSP in the caller is transferred to the LSP register in the callee, and the callee allocates any additional space it needs for automatic variables onto the stack (b). Upon callee exit, the LSP is discarded, and the SSP is positioned to effectively remove all local stack memory used by the callee (c). Finally, upon return to the caller, the SSP is again transferred to the LSP register by the hardware, and the original caller has the same LSP as before the call was made (d). The LSP points to the return value, if any, left by the callee.

Caller prologue

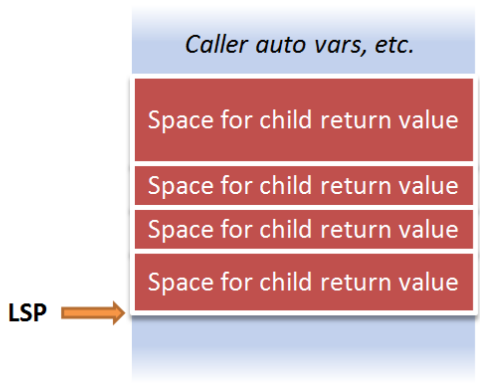
The caller may spawn one or more threads to execute concurrently while it continues running itself. The spawned threads may have one or more barriers that synchronize execution before continuing on with the program flow.

The caller function allocates separate stacks for the threads it creates. Unique stacks allow callee threads to execute freely without interfering with one another's local variables. The stacks may be allocated from the heap by the caller directly. Additionally, hardware may provide accelerated stack-allocation support as in Armada-1 (see appendix B). The caller allocates space for the return values of each callee either on its own stack or on the heap. Only the pointers to these memory locations are passed to the callees. Arguments to each callee are passed in through registers when available and the callees' individual stacks as needed. The stack layout for the concurrent caller prologue with return value storage allocated on its stack is shown in figure 8.3.

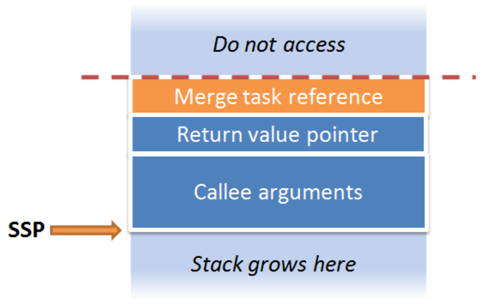
The caller must also setup any barrier code needed to synchronize execution among the threads it calls. The caller allocates barrier objects from the hardware as needed. Software supplies the number of threads to wait for, the independent code bag descriptor to fetch when those threads have completed, and arguments for that post-barrier code. The hardware then returns a reference to the barrier object. The caller places this barrier reference onto the stacks of callee threads. The caller passes a code bag descriptor to veneer code to the callees to handle the barrier object. That process is described in the caller epilogue section.

Callee entry and exit

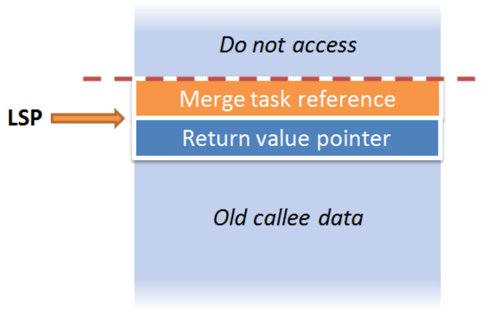
The callee entry and exit are identical to the serial case described previously. This symmetry enables functions to easily be called serially or concurrently as needed. Although this symmetry aids writing reentrant functions, it cannot alone guarantee thread safety. The functions may, for example, have unchecked access to global variables which can cause those routines to fail if run concurrently. It is the compiler or programmers' responsibility to ensure such accesses are avoided or guarded.


Caller auto vars, etc.
Space for child return value
Space for child return value
Space for child return value
Space for child return value
LSP

(a) Caller prologue.



(b) A callee stack setup by the caller.



(c) Barrier veneer code entry.

Figure 8.3: Stack layout for a concurrent procedure call. The caller allocates space for the return values, if any. In this example, the caller allocates space for those values on its own stack (a). The caller then allocates stacks for each callee. If the callee threads must all complete before control is returned to the caller, the caller will allocate a merge-task resource from the hardware and push a reference to it onto each callee stack. The caller will then replace the *LR* value with a code bag descriptor for veneer code that handles the task merging. The caller also passes return-value pointers unique to each callee and any arguments to the callees (b). The callees behave identically to the serial procedure case. If a merge-task is used, callees will fetch veneer code that decrements the merge-task reference counter when they each complete (c). The hardware then detects when all callees have terminated, and fetches a successor code bag to execute.

Caller epilogue

The veneer code set up by the caller decrements the barrier object count using the barrier reference provided on the stack. The stack may now be freed or may be freed after the barrier. Freeing the stack now, when possible, is the preferred method so as to hide this latency from program execution as much as possible. This veneer code bag then terminates. Program flow is continued when the barrier object reaches a zero count indicating that the spawned threads have completed. The hardware automatically fetches the post-barrier code to execute as set up by the caller prologue.

8.6 Conclusion

The APCS describes the rules that software must follow when writing or generating assembly code to guarantee interoperability among code. Serial and concurrent methods of calling functions described by the APCS make use of the Armada architecture's independent code bags and thread level concurrency support. The next chapter describes a compiler that can generate APCS-compliant code from a high-level imperative programming language.

Chapter 9

Compiling for Armada

New and radical computer architectures are easy to envision. Programming these architectures is an entirely different matter. Part of this research aims to prove that the Armada ISA can be targeted by a modern imperative-language compiler. This research does not concede or conclude that imperative languages are the best choice for exploiting concurrency. These languages are ubiquitous, though, and it must be possible for an imperative language compiler to target the Armada ISA for Armada to be a successful general-purpose computer architecture in the foreseeable future.

This chapter describes the implementation of a compiler back-end for the Armada architecture. Due to time constraints on the research, the back-end is not production quality and primarily targets only the basic Fleet instruction set; in particular, thread-level extensions proposed for Armada are not supported. However, the back-end does show that compilers for common imperative languages can manage Armada's fundamental programming interface and approach to exploiting ILP.

9.1 Modern optimizing compilers

Compilers take high-level languages that are designed for humans and translate them into primitive instructions that a machine can understand. Modern optimizing compilers like the GNU Compiler Collection (GCC) and the Low Level Virtual Machine compiler infrastructure (LLVM) typically have three parts as shown in figure 9.1. The first part, the front-end, takes a high-level programming language as input. This language might be C, Java, or Fortran, for example.

The front-end translates the input program into a common intermediate representation (IR). Because this IR is independent of the high-level language used, subsequent steps in the compilation process can operate directly on the IR without having to know anything about the high level language the program was originally written in. The optimizer, or the middle-end as it is often intriguingly called, performs optimizations on the IR. Common optimizations involve removing dead code, constant and value propagation, and subexpression elimination among many others. Finally, the back-end of the compiler is responsible for translating the optimized IR into the target machine's language. For this reason, a back-end for a specific target architecture is often referred to as a code generator for that architecture. An LLVM Armada code generator is the focus of this chapter.

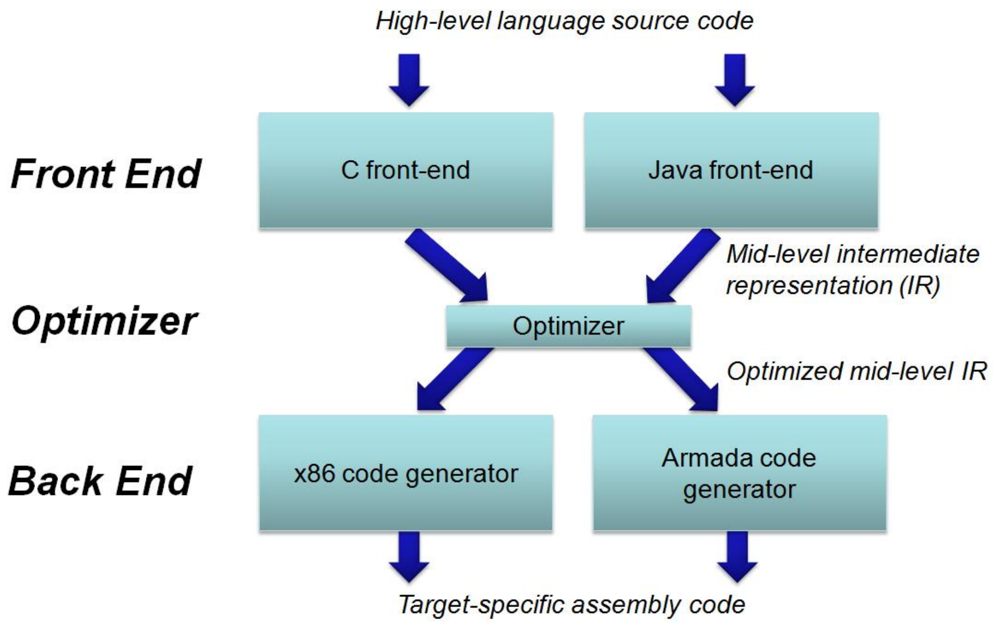


Figure 9.1: Modern optimizing compiler architecture. High-level language source code written in languages such as C, Java, or Fortran are translated into a common intermediate representation, or IR. This allows a common optimizer stage to optimize programs written in any of the supported high-level languages. Finally, a code generator converts the optimized IR to assembly code for the target architecture.

9.2 LLVM IR

Although *virtual machine* is in its name, the LLVM compiler infrastructure can be used as a static compiler. A static configuration of LLVM version 2.3 was used to create the Armada code generator. LLVM was chosen as the basis for the Armada code generator because of its large community support and excellent documentation. GCC was also considered, but its complexity and inconsistent documentation made it less amenable to the proof-of-concept compiler needed for this research. Other research compilers investigated like Trimaran and SUIF have generally lost support and are no longer actively updated.

The Armada code generator described here receives LLVM IR for input and produces Armada code as output. Although the full specification of the IR is beyond the scope of this work, the parts of the IR that are most relevant to the Armada code generator are described in the following sections.

9.2.1 Basic blocks

The IR fundamentally consists of basic blocks of sequenced LLVM instructions. A basic block has only one entry point and only one exit point. This definition implies that branches cannot jump into the middle of a basic block. Furthermore, there can only be one branch instruction in a basic block, and this branch instruction must be the last instruction in the block. By definition, if any instruction in a basic block is executed, then all of the instructions in the basic block must be executed. The all-or-none instruction execution semantics of basic blocks are similar to those of Fleet and Armada code bags. This symmetry is exploited by the back-end as discussed in the following sections.

A basic block may have one or more *predecessor* blocks that jump to the start of it. Additionally, a basic block may have one or more *successor* blocks that its last instruction can conditionally jump to.

9.2.2 Instructions

There are seven classes of LLVM instructions as shown in table 9.1. Most instructions in the LLVM IR are in static single assignment (SSA) form. In SSA form, variables are defined exactly once. Where a high level language may allow a variable to be redefined multiple times throughout a program, the LLVM IR creates new, unique versions of that variable as needed. LLVM supports an infinite number of typed virtual registers to hold these values. These virtual registers are mapped to real machine registers in a subsequent stage of the compilation process. Figure 9.2 shows the control flow graph (CFG) for a program in LLVM

IR format that prints the numbers 0 to 99. Each box in the CFG corresponds to a basic block. Note how the second assignment of the variable *i* creates a new variable, *indvar.next*, rather than overwriting the original variable.

Class	**Description**	**Examples**
terminator	Every basic block ends with a single terminator instruction. Terminator instructions change control flow and typically have no value.	br, ret
binary	Binary operations execute an operation over two inputs. The inputs must be of the same type.	add, sub, mul, udiv
bitwise binary	Bitwise operations manipulate the bits of an input.	shl, ashr, xor
vector	Vector operations support insertion, removal, and permutation of vector data.	extractelement, insertelement, shufflevector
memory	The LLVM IR contains instructions to access and address memory. Memory locations are not in SSA form in the IR. LLVM provides instructions to allocate memory on a heap or stack, to free memory, to load and store data, and to address data within aggregate data structures.	malloc, free, alloca, load, store
conversion	A slew of LLVM conversion instructions allow values to be cast from one type to another.	trunc, zext, sext, fptoui, ptrtoint, bitcast
miscellaneous	In addition to the *phi* instructions, the IR has other miscellaneous instructions to compare values and call functions among other behaviors.	icmp, fcmp, call, va_arg

Table 9.1: LLVM instruction classes.

Most LLVM instructions produce a value as a result. For example, figure 9.3 shows two typical LLVM instructions. The *add* instruction produces a result, *%tmp*. A few instructions like *store*, however, do not produce a result.

LLVM implements load-store architecture semantics; data are transferred between registers and memory explicitly through load and store instructions. Unlike the majority of instructions in LLVM, memory instructions are not in SSA form.

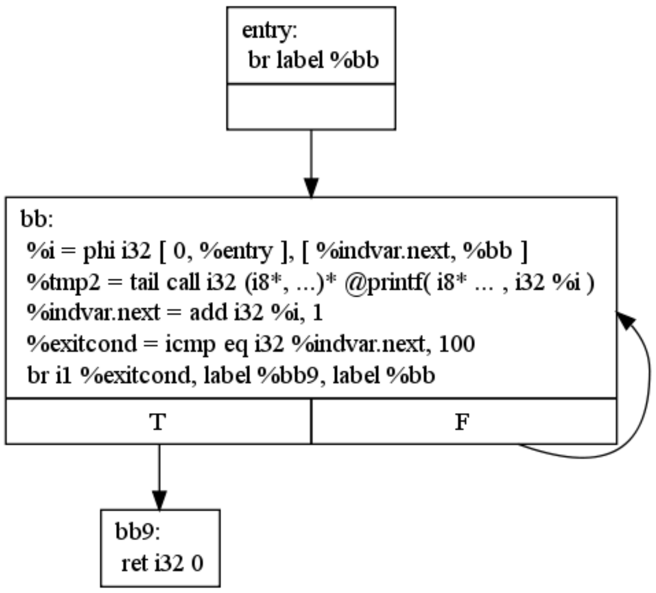

entry:
br label %bb
bb:
%i = phi i32 [0, %entry], [%indvar.next, %bb]
%tmp2 = tail call i32 (i8*, ...)* @printf(i8* ... , i32 %i)
%indvar.next = add i32 %i, 1
%exitcond = icmp eq i32 %indvar.next, 100
br i1 %exitcond, label %bb9, label %bb
T
F
bb9:
ret i32 0

Figure 9.2: Control flow graph of a simple LLVM loop.

Because pointers to memory for these instructions can take on a very large number of possible values, there is not a reasonable, compact way to represent these possibilities in SSA form.

Additionally, LLVM enforces strong typing on its IR. In figure 9.3, the *add* instruction is operating on two 32-bit integers. The *store* instruction then stores the result to a memory location referred to by the 32-bit integer pointer *%ptr*. Cast instructions enable explicit conversions between data types. The LLVM primitive types are void, integer, boolean, and floating point.

```
        %tmp = add i32 4, %var
        store i32 %tmp, i32* %ptr
```

Figure 9.3: LLVM add and store instructions. LLVM instructions are strongly typed and are generally in static single assignment (SSA) form.

Basic blocks can have multiple predecessors. Some variables may have different values depending on the block that directly preceded the execution of the current block. For example, referring back to figure 9.2, i will take on the constant value 0 for the first iteration of the loop when the preceding, invoking block is *entry*. On subsequent iterations of the loop, the preceding block is *bb* and thus $i = indvar.next$. This flow-dependent assignment, characteristic of programs in SSA form, is captured by the LLVM *phi* instruction. The *phi* instruction in basic block *bb* conditionally assigns a value to a variable depending on the predecessor block that invoked it. The *phi* instruction is a member of the *miscellaneous* class of LLVM instructions.

9.3 Armada code generator

The Armada code generator creates Armada assembly code from the LLVM IR. This assembly code is run through an assembler to generate Armada binaries. These binaries can then be executed on Armada architecture models such as ArmadaSim described in chapter 6.

9.3.1 Data types

As described in a previous section, LLVM IR instructions are strongly typed. The LLVM IR supports void, boolean, 8- to 64-bit signed and unsigned integers, and single- and double-precision floating point primitive types. Additionally, the IR supports four derived types – pointers, functions, structures, and arrays.

Void

The `void` type is a trivial mapping to Armada; the type is data-less and 0 bytes in size.

Boolean

The `boolean` type is mapped to an unsigned 64-bit integer. The least significant bit holds the boolean value, and the meaning of the rest of the bits is undefined.

Integers

LLVM provides a full range of 8- to 64-bit signed and unsigned integers. Some examples of uncommon integer sizes supported by LLVM are 21 and 37 bits. Armada supports 8-, 16-, 32-, and 64-bit signed and unsigned integers. LLVM integer types in-between supported sizes are promoted to the closest-sized Armada integer type that can fully represent the data.

Floating point

Armada supports 64-bit, double-precision floating point values. It does not have single-precision support; all LLVM single-precision floating point types are promoted to the double-precision type.

Derived types

As Armada-1 supports 64-bit addressing, all pointers are treated as 64-bit unsigned integers. Elements of arrays and structures are accessed through their base pointers with offsets calculated by the size of the primitive types they contain. Finally, functions are translated to 64-bit pointers to their address in the program binary. The LLVM function type includes the argument types and the return type. The back-end uses this information to determine how arguments are passed to functions and how the return value is returned from the callee. The Armada Procedure Call Standard described in the previous chapter dictates how the back-end should move this data between caller and callee.

9.3.2 Pre-processing

Prior to forming code bags and translating the IR into Armada assembly, several pre-processing steps transform the IR at a high-level to make subsequent processes easier.

LR insertion pass

The LR insertion pass introduces the link register into the IR. If the function makes subroutine calls, space is allocated on the stack for the LR so it may be recalled later. This pass also adds code at the end of the terminal basic block of a function to forward the contents of the LR to the fetch unit.

APCS conformance pass

This pass makes a function and all of its callers conform to the Armada Procedure Call Standard in terms of handling return values. It does three things.

First, it transforms the signature of all non-void-return-type functions into void-return-type functions that take one additional argument — a pointer to the old return type. This satisfies the APCS requirement that return values are always returned indirectly through memory as opposed to a register transfer. The return value pointer precedes all other arguments. For example, this prototype:

```
int fac(int f);
```

is converted to:

```
void fac(int *_ret_result_storage, int f);
```

Second, for every non-void-return-type function modified, all callers of that function are adapted accordingly. The callers create space on their stack to hold the return data. They also pass a pointer to this space as an additional argument to each call of the modified function. For the example above, a caller would create storage space for fac's return value. Then, every call to fac in this function would be changed to provide the return storage space pointer as the first argument to the callee.

Finally, the pass changes all LLVM `return` instructions within the transformed function into a two instruction sequence. The first instruction stores the return value into the return value space provided by the caller. The second instruction returns `void`.

Procedure call block-splitting pass

The LLVM IR embeds procedure calls within the middle of basic blocks. This approach works fine for architectures with a program counter. The instruction to execute upon return is simply the instruction after the procedure call. As alluded to earlier, Armada will eventually transform the basic blocks into code bags. Armada therefore treats a procedure call as the termination of the block and any subsequent code as another basic block. This allows the callee to return by fetching the code bag descriptor for the remaining code in the caller. This pass splits blocks containing procedure calls into two or more basic blocks at every procedure call.

Entry point veneer pass

When callees return to callers in Armada, they fetch an independent code bag jumping into the function just past the procedure call. After the block-splitting pass, this return point is now a separate basic block in the IR. This pass traverses each new entry point into the function created by the splitting pass and adds the necessary code to reload values stored on the stack prior to the function call. It also reads the callee's return value out of memory into an LLVM IR variable.

Stack pointer insertion pass

LLVM allocates spaces on the stack for variables with the `alloca` instruction. Instructions using those variables refer to the `alloca` instruction's value. This is an abstract reference that must be made concrete. This pass converts such references to loads offset from the local stack pointer.

9.3.3 Code bag formation

In the Fleet and Armada architectures, instructions are fetched and executed in variably-sized groups of code bags. If any one instruction in a bag is fetched and executed, all instructions in the bag will be fetched and executed. LLVM basic blocks behave similarly; once a program jumps into a block of instructions, all instructions in that block are guaranteed to execute. The code generator exploits this symmetry and directly maps basic blocks to code bags.

However, there is a fundamental difference between basic blocks and code bags in how instructions are ordered. In LLVM blocks, instructions are assumed to execute in sequence. These semantics differ from the unordered nature of code bags. Because the majority of LLVM instructions are in SSA format, this difference in sequencing is not a major problem. In SSA, variables can only be assigned once per block. Therefore, there is minimal ambiguity due to read-after-write or write-after-read hazards (RAW and WAR respectively) within a block[1]. If an Armada processor attempts to execute an instruction before its inputs have been assigned values, the instruction will stall until those inputs are ready. Thus, the execution of instructions originating from LLVM SSA instructions are governed by data-flow ordering.

The code generator cannot completely overlook certain sequencing aspects of LLVM IR semantics. For example, memory access instructions are problematic. Consider multiple *load* instructions reading from a hardware first-in first-out

[1]However, two pointers could alias to a single address making RAW and WAR hazards a concern. Programmers and the compiler should work together to eliminate or minimize such cases. Failure to do so will cause some instructions to execute serially that could otherwise run concurrently. LLVM and other modern compilers attempt to determine whether two pointers may alias each other to optimize code most efficiently.

(FIFO) register. The meaning of the values read from the FIFO likely depend on the order in which they were read. A similar scenario involving *store* operations can easily be envisioned. The code generator must therefore preserve the execution sequence of at least some memory access instructions. In this first implementation of the compiler, all memory accesses are forced to occur in the order given by the LLVM IR. This ordering is implemented by gating memory accesses on the completion of previous accesses using tokens. The memory Ship generates a completion token whenever a store operation completes. Recall that in Armada-1 all memory options are instantaneous. The memory Ship has been implemented to generate completion tokens based upon the mini-opcode sent to it. The programmer may choose to:

1. not generate a completion token
2. generate a completion token upon store completion
3. generate a split-completion token once a store operation has been issued to the memory subsystem in-order but has not necessarily completed

Although the current compiler back-end always uses the conservative option two, the other options are available to optimize memory subsystem performance. Option three is particularly useful as it can be used to guarantee the ordering of memory operations without waiting for those operations to complete. Therefore, multiple memory operations may be outstanding to help hide memory subsystem latency in a real system. Once the hardware issues a split-completion token, it guarantees that any other subsequently-issued requests to memory will occur after that previous operation has completed. Support for split-completion tokens in the compiler back-end is left as future work.

Additionally, although RAW and WAR hazards do not generally occur within a single code bag, these hazards can present in cases where instructions from multiple bags are in-flight at once. In program loops, for example, the code generator must guarantee that operations from older iterations complete before corresponding operations from the current iteration. The back-end accomplishes this by ensuring that all instructions in the current bag have completed before fetching the next code bag. This very conservative implementation guarantees correctness but unnecessarily limits ILC present among multiple code bags. Optimization in this area is left as future work.

9.3.4 Ship type and instruction selection

The back-end decides which Ship or Ships to use for each LLVM IR instruction and generates Armada *move* instructions to and from those Ships. This compiler pass is currently only capable of generating code for a single IR instruction at a time; it is unable to map multiple IR instructions together into an optimal Armada operation. For example, the multiple IR instructions involved in a *for* loop cannot currently make use of Armada's stride Ship.

The selection pass allocates virtual Ships for the Armada instructions it generates. These virtual Ships are mapped to physical resources in the Fleet in a later pass. Virtual Ships are distinguished from physical ones in code listings by the presence of a tilde in front of the virtual Ship name. The full IR instruction-to-Ship mapping is specified in appendix D.

Once the appropriate Ship type is selected, the compiler must generate Armada move instructions that move data to and from the Ship appropriately. For example, the branch instruction shown in figure 9.4 is translated into four move instructions making use of a selector and fetch Ship.

At this stage in the compilation process, all move instructions generated are move-once, copy-source to a single destination (denoted by `*->`). The token cleanup stage described later will convert copy-source to move-source operations when the source data does not need to be preserved. Recall that standing instructions are not supported in this first implementation of the compiler back-end.

```
br i1 %cond, %iftrue, %iffalse
```

```
%cond *-> ~selector_3.in_sel;
(iftrue_bag) *-> ~selector_3.in_true;
(iffalse_bag) *-> ~selector_3.in_false;
~selector_3.out *-> fetch.in_next_cbd;
```

Figure 9.4: Translation of LLVM branch to Armada move instructions.

9.3.5 Instruction merging

The initial instruction selection pass generates many single-destination instructions. The instruction-merging compiler pass then combines multiple move instructions that have the same source address or source value into a single Armada instruction with multiple destinations. The pass thus reduces the number of instructions in code bags decreasing static code size and promoting increased instruction fetch efficiency.

9.3.6 Instruction splitting

The instruction merging pass can potentially merge a large number of instructions together into a single instruction with a large number of destination addresses. In an actual implementation of Armada, the instruction opcode size limits the actual number of destinations allowed per instruction. For example, Armada-1 allows only two destinations per 32-bit instruction. The instruction splitting pass looks for instructions that have more destination addresses than the number supported by the underlying Armada implementation and splits those instructions into multiple *move* instructions. This pass is decoupled from the instruction merging pass because the number of supported destinations and, therefore, the splitting process, are implementation-dependent.

9.3.7 Stray token cleanup

Unlike traditional architectures, the presence or absence of data plays a key role in the programming and execution of Armada programs. Whereas registers may be overwritten freely in a traditional synchronous machine, for example, asynchronous Fleet architectures require an unused or expired token be disposed of to clear the path for subsequent tokens traveling along the same route or using the same resource. Failure to do so will result in imminent deadlock and errors in the program. In Armada, software is responsible for handling these stray tokens. The assembly-level programmer or compiler must keep track of tokens, determine when they expire and become stray tokens, and, finally, dispose of the stray tokens.

Source of stray tokens

Stray tokens are expired values that no longer have any use in a Fleet program. Stray tokens are generated in multiple ways. A common way is for a temporary variable to go out of scope. For example, once an operation on a loop iterator satisfies the loop termination condition, the iterator may no longer be of any use as in example 9.2. The program must then dispose of the token. Another example is when values passed in through registers to a function call are never referenced because of a particular control flow path taken within the function. If software performs any speculation, mispredictions transform speculated values into stray tokens.

Tracking stray tokens

The compiler keeps track of every value in the program from the moment they are created. This process is similar to liveness analysis for values in compilers for traditional architectures. When values are passed in to a function through registers, those tokens are added to a token *live* list. Similarly, automatic variables created by the function are also added to the *live* list. In this way, the compiler behaves in a traditional manner.

Fleet architecture compilers, however, have the additional responsibility of understanding data transformations that occur at Ships. For example, two values that are sent to an adder will be consumed and generate a new result value. The compiler must replace the two operands from the *live* list with the result token. This process is complicated by persistent tokens, such as constants, that reside at a Ship's input port until Ship-specific events occur that clear the token. If one of the inputs to an adder is persistent, for example, that token will only be cleared when the other operand is *last*. Operand persistence and other Ship state, such as the internal count in a Counter Ship, must be cleared when such state goes out of scope. If this clearance does not happen as a side-effect of some calculation in the program, the compiler must generate code to explicitly clear a Ship's state.

Removing stray tokens

In the original Fleet proposal, a bit-bucket Ship was proposed to accept stray tokens [Sut05]. It was a true sink that generated no output. This research determined that a pure sink was insufficient in a highly concurrent, asynchronous architecture like Fleet. In a system where software manages the tokens flowing through the system, it is essential to determine when tokens have been successfully eliminated. The problem with the bit-bucket Ship is that it does not have a way to indicate when stray tokens have arrived in the bucket. As Fleet imposes no strict timing constraints on Ships or the switch fabric, it is important to know when those tokens have been eliminated to avoid deadlock and miscalculations caused by data arriving out of the intended order. Armada thus uses a counter to collect the stray tokens. Software provides an input to the counter indicating how many tokens to wait for. The stray tokens are moved to the counter's token input port. When the specified number of tokens have arrived, the counter generates a single output token that can be used to gate the fetch of another code bag or free the underlying Fleet tile for example. New code bags with token-cleanup code are introduced into programs as needed. Typically, these code bags are added near forks in control flow where tokens often become extraneous depending on the path taken by the program. Token cleanup proves to be a costly problem as described in chapter 11.

9.3.8 Ship allocation

Thus far, the instructions created have been almost exclusively using virtual Ships created as-needed by the compiler. This limitless supply of virtual Ships must be mapped to a finite set of real Ships that are in the targeted Fleet tile type. This is accomplished by the Ship allocation pass of the compiler.

The compiler is supplied with a Ship composition map for each type of Fleet tile in the system. In this study, the targeted Armada-1 has only one core type. The mapping describes the addresses for all software-accessible input and output ports of the Ships. Each function is compiled individually and thus not entangled with any other function. The Ship allocation pass passes over each function separately visiting each code bag. The virtual Ships in each instruction are replaced by physical Ships as available. If two instructions move data to virtual destination Ships that cannot possibly be live at the same time, the same physical Ship may be used in both of the moves.

Like register allocators for traditional architectures, the Ship allocation pass uses graph-coloring techniques to assign the virtual Ships to physical resources. Ships are analagous to registers in register-allocation schemes. An interesting problem, however, is that there are many more Ships in a Fleet than registers in most architectures. The initial Ship allocator took many minutes to execute forcing a search for a more optimal solution.

As the LLVM IR is in SSA format[2], the interference graphs for each function are chordal [Hac05][HGG06][BDR07]. A chord is an edge in the graph connecting two non-adjacent nodes. A cycle of nodes in a chordal graph has at most

[2]`store` instructions generated by the LLVM IR may not be in SSA form, but after Armada inserts store completion tokens, they effectively are.

three nodes. The significance of this characteristic is that it reduces the complexity of the Ship assignment problem when Ships do not need to be spilled. While the problem is NP-complete for non-chordal graphs, the SSA-elimination algorithm for chordal ones is optimal in polynomial time [Yi05][BDJS06]. The list-coloring implementation used in the Armada back-end is an adaptation of a method proposed for register and functional unit allocation in VLIW processors [ZW03].

If there are not enough physical Ships to accommodate all of the instructions, the compiler would ideally introduce serialization to eliminate the overlapping liveness of the constrained resources. This may be accomplished by gating the execution of one move instruction on completion of others that use the same resource. The compiler may split a code bag in two to separate the colliding instructions into different groups. The output token of the constrained resource is then used to gate the fetch of the next code bag that also needs that resource. Note that this spilling process is not yet implemented in the code generator and left as future work.

After this Ship selection stage, the compilation process is complete. Armada assembly code is generated that is compliant with the APCS and can be assembled into machine code.

9.4 Assembler

The Armada assembler takes the assembly code text and builds program binaries that can execute on Armada hardware or hardware simulators like Armada-1. The assembler was written before the LLVM back-end and before the idea of creating a high-level language compiler to target Armada was envisioned. Unlike the compiler, it does assemble all valid Armada assembly code including independent code bags, standing move instructions, and other features not supported by the back-end. The operation is straight-forward and will not be described in detail.

9.5 Limitations

A fully operational and robust modern optimizing compiler requires a large amount of effort to create and validate. Although LLVM was leveraged to aid the process, a fully functional compiler was out of the scope of this research due to time constraints. The current compiler generates Armada assembly code that may possibly need hand-coded modification in two ways. First, although the compiler detects conditions where there are more virtual Ships referenced than there are physical Ships to map them to, it cannot correct the code and spill Ships when needed. Additionally, the compiler cannot always split instructions with too many virtual Ship destinations into multiple, two-destination instructions as required by Armada-1.

In addition to these functional limitations, numerous optimization-related limitations were described in the chapter. The compiler back-end is generally very conservative to ensure correct operation. However, this approach limits available concurrency that the architecture could exploit.

9.6 Conclusion

The Armada compiler back-end generates assembly code that can be assembled with minimal manual intervention into binaries for the Armada-1 architecture model. The next chapter discusses a debugger designed to aid assembly-level programmers as well as the manual steps required to complete the compilation process using the limited compiler back-end described in this chapter. The quality of the code output from the compiler is then compared to the hand-coded implementation of the Mandelbrot benchmark in chapter 11.

Chapter 10

ArmadaSim debugger

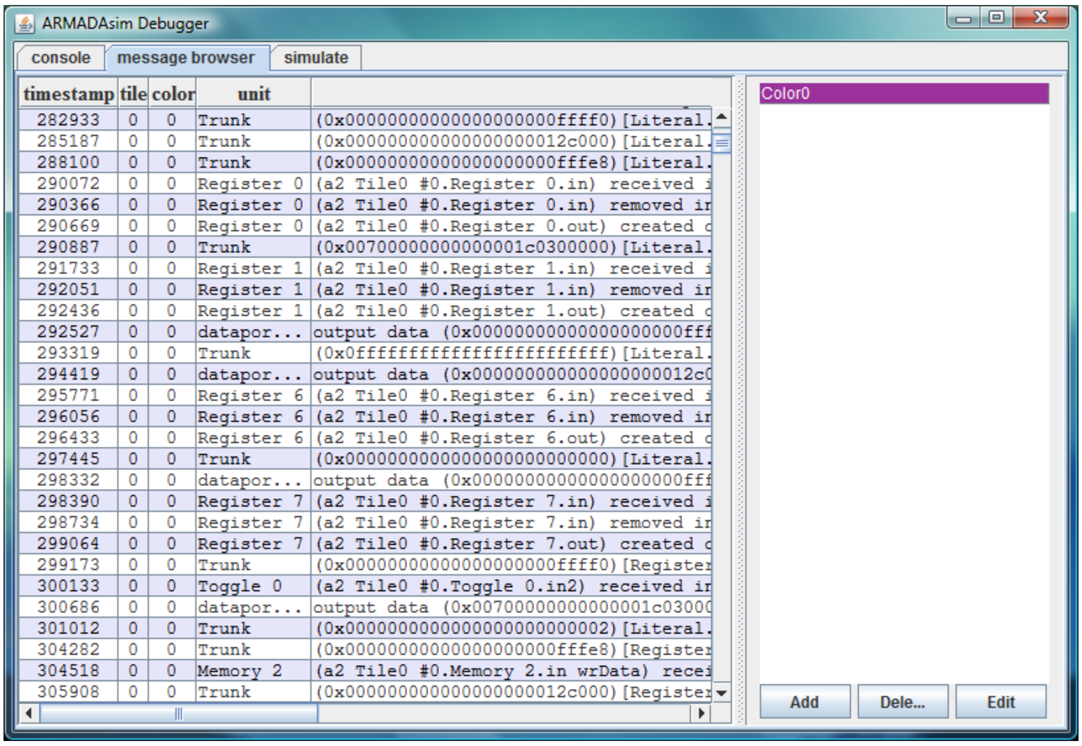
Concurrent execution adds additional complexity to the debugging process. Not only are many instructions from a single thread elgible for execution at one time; many threads may be executing concurrently as well. The ArmadaSim Debugger helps the low-level programmer visualize the state of the system and the possibilities of which instructions may imminently execute.

10.1 Trace replay

The ArmadaSim Debugger replays traces captured from a previously executed simulation allowing programmers to step through the code and see how the processor state changed. The ArmadaSim simulator may be set up to capture traces from all parts of the processor. It must minimally be configured to capture messages from the Fetch unit and from the trunk element of the switch fabric to generate a trace usable by the debugger; other messages are ignored. The Fetch Ship messages tell the simulator which Fleets in the Armada are in use at any time. The trunk element provides a timestamp on when instructions pass through the switch fabric. Recall that every instruction must pass through the trunk element of the horn-and-funnel switch fabric in a Fleet in Armada-1.

10.2 Message view

The ArmadaSim Debugger relies on the simulation trace or message file to generate helpful debug information. Figure 10.1 shows the message view of the debugger. The left pane shows simulator messages along with their corresponding timestamp. The Fleet tile where the message originated is given by the physical *tile* and *color* fields. As the simulator may be configured to dump many messages, it is useful to filter out irrelevant messages in the debugger. The right pane allows the programmer to create custom filters to only show messages that meet given criteria. In figure 10.1, a *Color0* filter is specified that only shows messages originating from color layer 0 of the Armada.

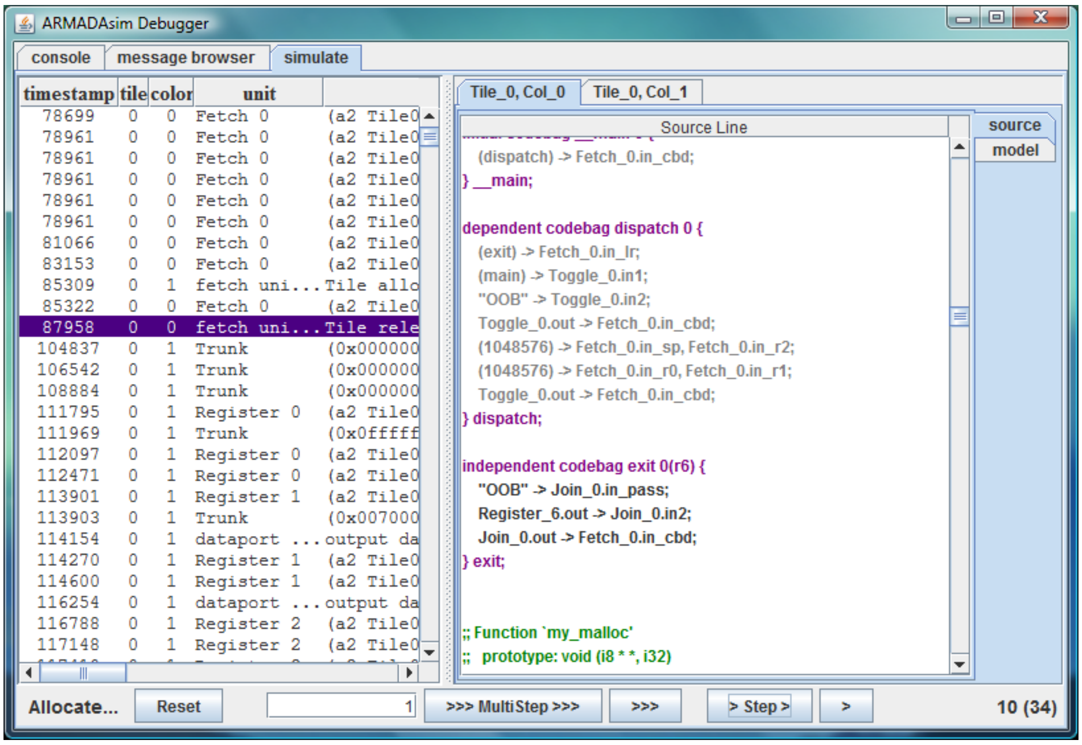


timestamp	tile	color	unit	
282933	0	0	Trunk	(0x0000000000000000000ffff0) [Literal.
285187	0	0	Trunk	(0x00000000000000000012c000) [Literal.
288100	0	0	Trunk	(0x0000000000000000000fffe8) [Literal.
290072	0	0	Register 0	(a2 Tile0 #0.Register 0.in) received i
290366	0	0	Register 0	(a2 Tile0 #0.Register 0.in) removed ir
290669	0	0	Register 0	(a2 Tile0 #0.Register 0.out) created c
290887	0	0	Trunk	(0x0070000000000001c0300000) [Literal.
291733	0	0	Register 1	(a2 Tile0 #0.Register 1.in) received i
292051	0	0	Register 1	(a2 Tile0 #0.Register 1.in) removed ir
292436	0	0	Register 1	(a2 Tile0 #0.Register 1.out) created c
292527	0	0	datapor...	output data (0x000000000000000000000fff
293319	0	0	Trunk	(0x0fffffffffffffffffffffff) [Literal.
294419	0	0	datapor...	output data (0x00000000000000000000012c0
295771	0	0	Register 6	(a2 Tile0 #0.Register 6.in) received i
296056	0	0	Register 6	(a2 Tile0 #0.Register 6.in) removed ir
296433	0	0	Register 6	(a2 Tile0 #0.Register 6.out) created c
297445	0	0	Trunk	(0x000000000000000000000000) [Literal.
298332	0	0	datapor...	output data (0x000000000000000000000fff
298390	0	0	Register 7	(a2 Tile0 #0.Register 7.in) received i
298734	0	0	Register 7	(a2 Tile0 #0.Register 7.in) removed ir
299064	0	0	Register 7	(a2 Tile0 #0.Register 7.out) created c
299173	0	0	Trunk	(0x0000000000000000000ffff0) [Register
300133	0	0	Toggle 0	(a2 Tile0 #0.Toggle 0.in2) received ir
300686	0	0	datapor...	output data (0x0070000000000001c03000
301012	0	0	Trunk	(0x000000000000000000000002) [Literal.
304282	0	0	Trunk	(0x0000000000000000000fffe8) [Register
304518	0	0	Memory 2	(a2 Tile0 #0.Memory 2.in wrData) recei
305908	0	0	Trunk	(0x00000000000000000012c000) [Register

Figure 10.1: ArmadaSim Debugger messages view. The ArmadaSim Debugger messages view displays all messages in the trace file generated by ArmadaSim. Custom filters like the *Color0* filter shown can be used to suppress the display of chosen messages from the trace file.

10.3 Source code view

The primary use for the debugger is to track the progress of a previously-executed program. The user can load up the assembly code for the program the trace was generated from and step through the code. The debugger displays the trace file messages in the left pane and syntax-highlighted source code in the right pane as shown in figure 10.2. When multiple Fleets are active simultaneously, new source code tabs are generated in the debugger for each virtual Fleet. The programmer may switch tabs to see what each Fleet is currently doing. When the user steps through to the next instruction, the debugger can switch tabs automatically to the Fleet that generated the message. The user may also choose to only debug one Fleet at a time and have the debugger skip to the next message originating from the Fleet under inspection.



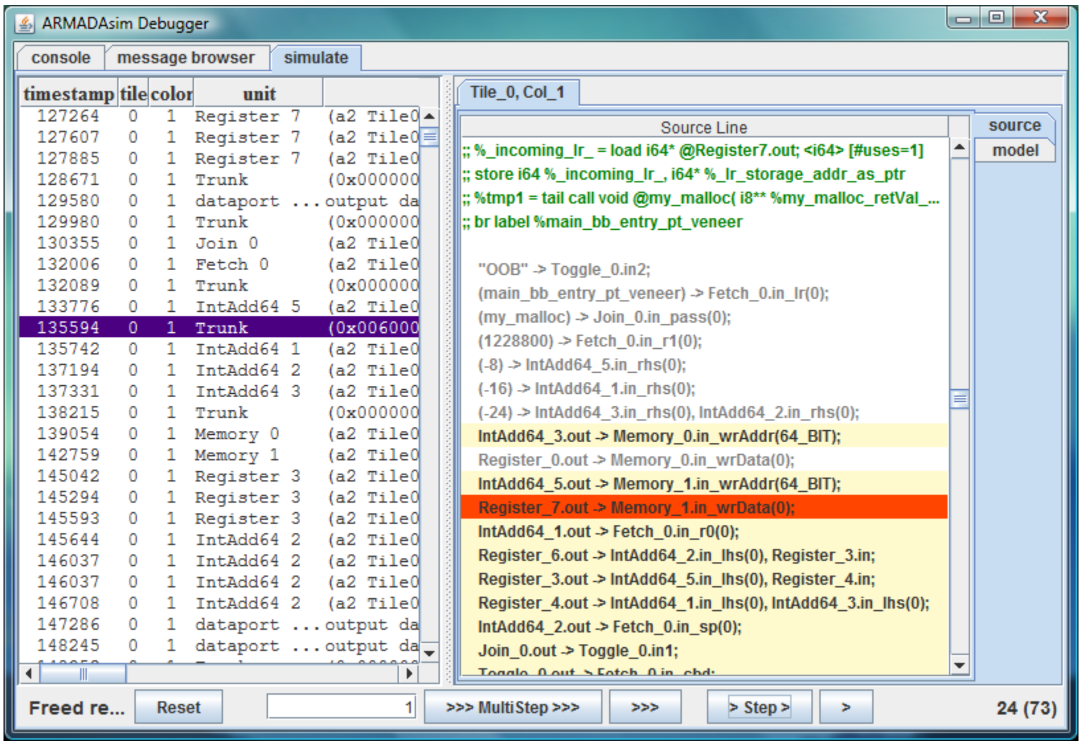
```
timestamp tile color unit
  78699    0    0   Fetch 0      (a2 Tile0
  78961    0    0   Fetch 0      (a2 Tile0
  78961    0    0   Fetch 0      (a2 Tile0
  78961    0    0   Fetch 0      (a2 Tile0
  78961    0    0   Fetch 0      (a2 Tile0
  78961    0    0   Fetch 0      (a2 Tile0
  81066    0    0   Fetch 0      (a2 Tile0
  83153    0    0   Fetch 0      (a2 Tile0
  85309    0    1   fetch uni...Tile allo
  85322    0    0   Fetch 0      (a2 Tile0
  87958    0    0   fetch uni...Tile rele
 104837    0    1   Trunk        (0x000000
 106542    0    1   Trunk        (0x000000
 108884    0    1   Trunk        (0x000000
 111795    0    1   Register 0   (a2 Tile0
 111969    0    1   Trunk        (0x0fffff
 112097    0    1   Register 0   (a2 Tile0
 112471    0    1   Register 0   (a2 Tile0
 113901    0    1   Register 1   (a2 Tile0
 113903    0    1   Trunk        (0x007000
 114154    0    1   dataport ...output da
 114270    0    1   Register 1   (a2 Tile0
 114600    0    1   Register 1   (a2 Tile0
 116254    0    1   dataport ...output da
 116788    0    1   Register 2   (a2 Tile0
 117148    0    1   Register 2   (a2 Tile0
```

```
Source Line
   (dispatch) -> Fetch_0.in_cbd;
} __main;

dependent codebag dispatch 0 {
   (exit) -> Fetch_0.in_lr;
   (main) -> Toggle_0.in1;
   "OOB" -> Toggle_0.in2;
   Toggle_0.out -> Fetch_0.in_cbd;
   (1048576) -> Fetch_0.in_sp, Fetch_0.in_r2;
   (1048576) -> Fetch_0.in_r0, Fetch_0.in_r1;
   Toggle_0.out -> Fetch_0.in_cbd;
} dispatch;

independent codebag exit 0(r6) {
   "OOB" -> Join_0.in_pass;
   Register_6.out -> Join_0.in2;
   Join_0.out -> Fetch_0.in_cbd;
} exit;


;; Function `my_malloc'
;;   prototype: void (i8 * *, i32)
```

Figure 10.2: Debugging multiple threads. The ArmadaSim Debugger keeps track of all active Fleet contexts. The programmer can flip through the tabs assigned to each context to inspect the state of each active Fleet.

10.4 State tracking view

A difficult problem encountered when debugging highly-concurrent programs is keeping track of which instructions are currently eligible for execution. The debugger highlights all instructions that may be currently in-flight based on which code bags have been fetched as in figure 10.3. Expired instructions are grayed out. The programmer steps through programs by stepping through the instructions as they travel through the trunk of the switch fabrics. The programmer has two levels of stepping granularity. The user can step through instructions only or can follow every state change such as when data arrives at a Ship.



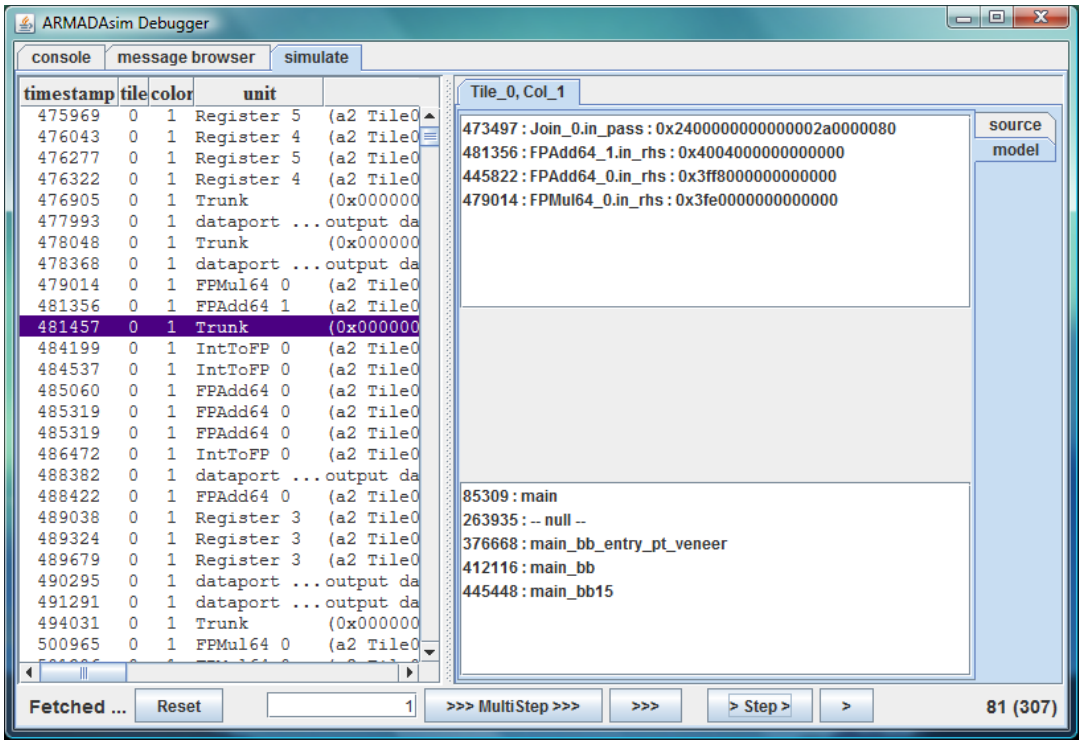
timestamp	tile	color	unit	
127264	0	1	Register 7	(a2 Tile0
127607	0	1	Register 7	(a2 Tile0
127885	0	1	Register 7	(a2 Tile0
128671	0	1	Trunk	(0x000000
129580	0	1	dataport ...	output da
129980	0	1	Trunk	(0x000000
130355	0	1	Join 0	(a2 Tile0
132006	0	1	Fetch 0	(a2 Tile0
132089	0	1	Trunk	(0x000000
133776	0	1	IntAdd64 5	(a2 Tile0
135594	0	1	Trunk	(0x006000
135742	0	1	IntAdd64 1	(a2 Tile0
137194	0	1	IntAdd64 2	(a2 Tile0
137331	0	1	IntAdd64 3	(a2 Tile0
138215	0	1	Trunk	(0x000000
139054	0	1	Memory 0	(a2 Tile0
142759	0	1	Memory 1	(a2 Tile0
145042	0	1	Register 3	(a2 Tile0
145294	0	1	Register 3	(a2 Tile0
145593	0	1	Register 3	(a2 Tile0
145644	0	1	IntAdd64 2	(a2 Tile0
146037	0	1	IntAdd64 2	(a2 Tile0
146037	0	1	IntAdd64 2	(a2 Tile0
146708	0	1	IntAdd64 2	(a2 Tile0
147286	0	1	dataport ...	output da
148245	0	1	dataport ...	output da

Source Line

```
;; %_incoming_lr_ = load i64* @Register7.out; <i64> [#uses=1]
;; store i64 %_incoming_lr_, i64* %_lr_storage_addr_as_ptr
;; %tmp1 = tail call void @my_malloc( i8** %my_malloc_retVal_...
;; br label %main_bb_entry_pt_veneer

  "OOB" -> Toggle_0.in2;
  (main_bb_entry_pt_veneer) -> Fetch_0.in_lr(0);
  (my_malloc) -> Join_0.in_pass(0);
  (1228800) -> Fetch_0.in_r1(0);
  (-8) -> IntAdd64_5.in_rhs(0);
  (-16) -> IntAdd64_1.in_rhs(0);
  (-24) -> IntAdd64_3.in_rhs(0), IntAdd64_2.in_rhs(0);
  IntAdd64_3.out -> Memory_0.in_wrAddr(64_BIT);
  Register_0.out -> Memory_0.in_wrData(0);
  IntAdd64_5.out -> Memory_1.in_wrAddr(64_BIT);
  Register_7.out -> Memory_1.in_wrData(0);
  IntAdd64_1.out -> Fetch_0.in_r0(0);
  Register_6.out -> IntAdd64_2.in_lhs(0), Register_3.in;
  Register_3.out -> IntAdd64_5.in_lhs(0), Register_4.in;
  Register_4.out -> IntAdd64_1.in_lhs(0), IntAdd64_3.in_lhs(0);
  IntAdd64_2.out -> Fetch_0.in_sp(0);
  Join_0.out -> Toggle_0.in1;
```

Figure 10.3: Stepping through a program. Users can step through the source code of a program with the debugger. The debugger highlights which instructions have executed and expired, which instructions are eligible for execution, and the current instruction passing through the switch fabric.

Another difficult problem in thinking about Armada programs is keeping track of which Ships have data at their input or output ports. A programmer may forget to supply a Ship with data it needs to compute an output causing the program to halt indefinitely. The debugger provides a model view of each active Fleet tile that allows the programmer to see data waiting at Ship input and output ports (figure 10.4). Visualizing this state allows the programmer to understand why a program has halted, why a Ship produced the output it did, or what stray state must be cleaned up before freeing a Fleet tile for reuse. This view also shows the code bag fetch history for the active Fleet. As code from previously fetched bags may still be in the system, this listing helps programmers keep track of which instructions may have lingering effects.



```
timestamp tile color unit
475969    0    1    Register 5    (a2 Tile0
476043    0    1    Register 4    (a2 Tile0
476277    0    1    Register 5    (a2 Tile0
476322    0    1    Register 4    (a2 Tile0
476905    0    1    Trunk         (0x000000
477993    0    1    dataport ...output da
478048    0    1    Trunk         (0x000000
478368    0    1    dataport ...output da
479014    0    1    FPMul64 0     (a2 Tile0
481356    0    1    FPAdd64 1     (a2 Tile0
481457    0    1    Trunk         (0x000000
484199    0    1    IntToFP 0     (a2 Tile0
484537    0    1    IntToFP 0     (a2 Tile0
485060    0    1    FPAdd64 0     (a2 Tile0
485319    0    1    FPAdd64 0     (a2 Tile0
485319    0    1    FPAdd64 0     (a2 Tile0
486472    0    1    IntToFP 0     (a2 Tile0
488382    0    1    dataport ...output da
488422    0    1    FPAdd64 0     (a2 Tile0
489038    0    1    Register 3    (a2 Tile0
489324    0    1    Register 3    (a2 Tile0
489679    0    1    Register 3    (a2 Tile0
490295    0    1    dataport ...output da
491291    0    1    dataport ...output da
494031    0    1    Trunk         (0x000000
500965    0    1    FPMul64 0     (a2 Tile0
```

```
473497 : Join_0.in_pass : 0x2400000000000002a0000080
481356 : FPAdd64_1.in_rhs : 0x4004000000000000
445822 : FPAdd64_0.in_rhs : 0x3ff8000000000000
479014 : FPMul64_0.in_rhs : 0x3fe0000000000000
```

```
85309 : main
263935 : -- null --
376668 : main_bb_entry_pt_veneer
412116 : main_bb
445448 : main_bb15
```

Figure 10.4: Tracking Fleet state. The debugger updates models for each active Fleet tile as the user steps through the program. Ship input port data, Ship output port data, and code bag fetch history for the Fleet under inspection are shown in the right pane.

10.5 Limitations

This debugger was primarily designed to aid in the compiler development process, and, thus, shares the same limitations as the compiler. In particular, the debugger does not track persistent inputs at Ship input ports, and it does not handle standing instructions.

10.6 Conclusion

The ArmadaSim Debugger helps programmers keep track of the large state space that an asynchronous, highly-concurrent architecture like Armada may have. By replaying traces generated by ArmadaSim, programmers can track changes to every Fleet in the Armada at once. The programmer can monitor very low-level state such as Ship input and output port events or step through many instructions at a time. Despite its limitations, the debugger handles all code generated by the Armada compiler making it a useful aid to the compiler development and verification process.

Chapter 11

Testing the Armada compiler

The Armada compiler demonstrates that it is feasible to translate a program written in a high-level imperative language into Armada assembly code. Although the thread-level extensions to the architecture are not explored by this proof-of-concept compiler, unique aspects of Fleet architectures such as the code bags of move instructions and the absence of a program counter are successfully targeted. This chapter examines the code generated by the compiler relative to what is possible with hand-coded optimizations as well as to code generated for other ISA's by traditional modern compilers.

11.1 Objectives

Although this compiler is largely exploratory, it is useful to have some idea of how the Armada compiler compares to more traditional compilers in terms of code density and dynamic code size. The compiler-generated version of the Mandelbrot benchmark is also compared to the hand-coded version to determine how much optimization the compiler is leaving on the table. Direct comparisons of benchmark run-times between traditional and Armada ISA's are avoided as ArmadaSim has not been correlated to real-world silicon performance.

11.2 Method and metrics

The Armada code generator output is compared to the output of x86-64 and MIPS code generators for the same version of the LLVM compiler infrastructure.

Thus, the same high-level optimizations are applied. However, the MIPS and x86-64 code generators are much more mature than the experimental Armada code generator and contain low-level optimizations not present in the Armada generator. The code density, number of instructions fetched, number of instructions committed, and instruction traffic are measured for all the target architectures and compared.

11.2.1 Code structure

The hand-coded implementation of the Mandelbrot benchmark attempts to make the best use of all of Armada's relevant features. The limited exploratory compiler generates code that is quite different from the envisioned low-level program. The code structure of the two different implementations are compared to better understand the challenges of compiling a high-level imperative language program for Armada.

11.2.2 Code density

Code density is the measure of the number of bits of instructions required to implement a particular algorithm. High code density promotes efficient use of the instruction cache, lower memory requirements for constrained embedded systems, and less instruction traffic at run time. These benefits can result in lower power consumption, higher performance, and lower costs for systems. This metric is measured by compiling the code with similar compiler options across the studied platforms and measuring the resulting executable file size. Library calls are avoided and replaced with dummy calls as needed to eliminate external effects on the executable file's organization and on code density.

11.2.3 Instructions fetched

The number of instructions fetched from memory are measured for each test platform. In Armada, standing instructions are unique in that they may be fetched once but effectively executed many times over. Although some processors for the other platforms may have mechanisms of re-executing previously fetched and decoded instructions (eg. a trace cache), these types of optimizations are not considered. Similarly, instructions that are fetched but never executed by speculative machines are not counted. Thus, for all test platforms except the

Armada with standing instruction support, the number of instructions fetched is the same as the number of instructions executed.

11.2.4 Instructions executed

Instructions executed by a processor must be fetched from memory, decoded, and issued. These steps all require time and energy. They also utilize some of the available system bandwidth that other threads could otherwise use. The GNU debugger (GDB) is used on the various target platforms to count the number of native instructions executed before the program completes. Standing instructions used in the hand-coded version of the Mandelbrot benchmark are counted as executed each time they move data through the switch fabric. It is therefore possible for the number of instructions executed or committed by Armada to exceed the number of instructions fetched.

11.2.5 Instruction traffic

Instruction traffic is analogous to code density measured at run time. Algorithms packed densely into instructions consume less bus bandwidth at run time. For many systems, dense code packing results in less toggling of pins on the chip package, a typically costly action in terms of power. In fixed-instruction-length architectures, instruction traffic from the instruction cache can be calculated by multiplying the instructions committed by the instruction size. Computing instruction traffic for variable-length instruction set architectures such as the x86 ISA is more difficult. An extension developed for GDB is used to count the number of bits in every instruction executed by the target architectures. For the case of standing instructions in Armada, instruction traffic is measured at the L1 cache only.

11.2.6 Concurrency

The amount of functional-unit concurrency achieved with the hand-coded version of Mandelbrot is compared with the amount achieved by the code generated by the compiler. The utilization of the Ships is compared between both versions of Mandelbrot running a single-core, single-threaded Armada configuration in ArmadaSim. Thread-level concurrency is not measured as the Armada code generator does not currently support independent code bags and hardware multithreading.

11.3 Results

Given the low level of maturity of the compiler, the implementation of Mandelbrot that it generates performs somewhat respectably relative to the optimized, hand-coded implementation. Table 11.1 compares the two implementations[1]. Some of this data has been previously presented in chapter 7.

	Hand-coded 1 Fleet	**Hand-coded 32 Fleets (2x16SMT)**	**Compiled 1 Fleet**
code size (bytes)	756		1176
number of code bags	9		15
relative performance	1	19.76	0.59

Table 11.1: Hand-coded versus compiled Mandelbrot.

11.3.1 Performance

The compiled implementation performs about half as fast as the hand-optimized version. Although slower, the performance is quite reasonable considering the limitations of this first code generator. It is not uncommon for hand-coded versions of algorithms for traditional architectures to exhibit similar improvement over compiler-generated versions. Supporting persistent inputs and standing instructions will narrow this gap. Merging the basic blocks into larger code bags will also eliminate overhead of managing data that goes in and out of scope. Finally, compiler support for the Armada thread-level extensions will provide the greatest performance benefit for parallelizable workloads and benchmarks like Mandelbrot; the hand-coded, multithreaded implementation achieves a 20x improvement on a 32-Fleet configuration. Such speed-ups are not possible without Armada's thread-level concurrency support.

[1]The compiled version used for this analysis differs slightly from the listing given in appendix C. The version used for this analysis does not contain calls to `malloc` and `free` to enable a fair comparison with the hand-coded version.

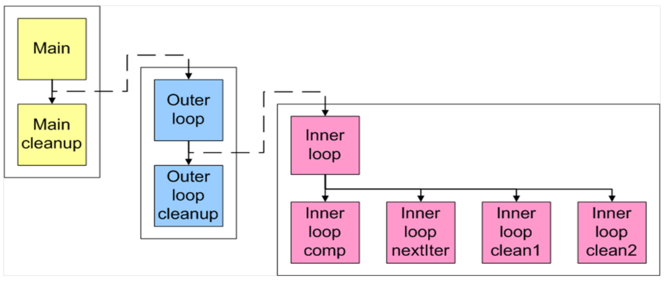
11.3.2 Code structure

The organization of the hand-coded and compiled Mandelbrot programs differ greatly. While the hand-coded version was aggressively optimized for the architecture, the compiler-generated format is generic and does not reflect the organization chosen to take advantage of thread-level extensions. Figure 11.1 shows the control flow diagrams for both implementations. Cleanup code bags, highlighted in figure 11.1(c), are inserted into the LLVM code by the Armada code generator. These code bags contain move instructions that pull stray, out-of-scope values from Fleets and dispose of them. As fetching code bags largely serializes the execution of instructions, these cleanup bags are detrimental to performance especially within the critical loop. The hand-coded implementation attempts to group cleanup code within existing bags thus avoiding that penalty. Such an optimization is possible for the compiler but is currently not in place.

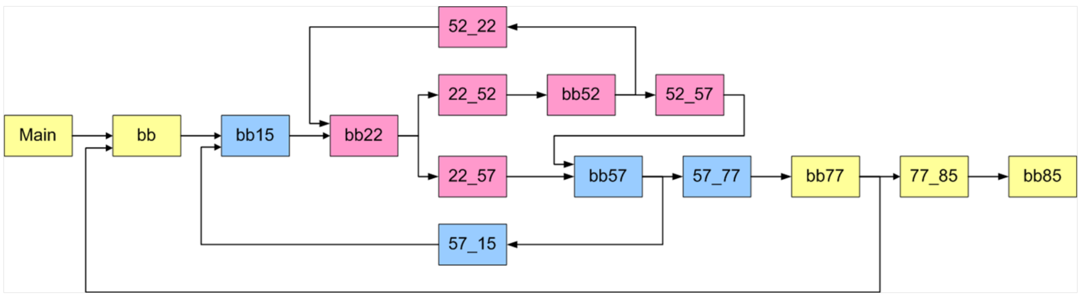
11.3.3 Code density

The compiled code is 56% larger than the hand-coded version. Primary reasons for this difference in code density are the lack of persistent Ship inputs and the larger number of code bags in the compiled version. Without persistent inputs, some values must be recommunicated with additional move instructions in other code bags. There are also more code bags in the compiled version because the code generator currently does not attempt to merge basic blocks into single code bags. The hand-coded implementation, however, attempts to merge as many instructions into a single bag as possible.

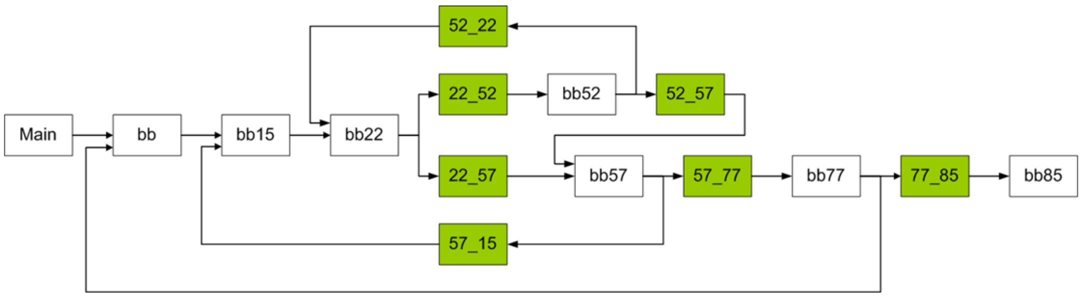
Compared to other architectures, the Armada code size is quite large at about four times the size of the same benchmark implemented for the MIPS RISC and x86-64 CISC architectures. As described in section 7.3, explicitly describing every movement of data in a Fleet requires more bits than only defining the operation as traditional architectures do. Additionally, Armada places data constants directly into instructions. In the other architectures examined, many constants are placed in separate memory regions (such as a literal pool) and read into the register file by load instructions. The size of the literal pool is not accounted for in table 11.2.



(a) Hand-coded implementation.



(b) Compiler implementation.



(c) Cleanup code bags in the compiled implementation.

Figure 11.1: Hand-coded and compiled Mandelbrot.

11.3.4 Instructions fetched and instruction traffic

Despite the larger code size, the number of instructions fetched at run-time by the hand-coded benchmark is fairly comparable to the number fetched on traditional architectures. The compiled version, however, fetches 2.7 and 3.7 times as many instructions as CISC and RISC architectures do respectively. Unsurprisingly, the amount of data actually transferred while fetching these instructions is also considerably higher. The compiled Armada implementation transfers about 2.8 times more instruction data than the hand-coded implementation. The compiler's inability to use standing instructions and persistent inputs contribute to this shortcoming. Additionally, poorer code organization and more orphan tokens to clean up than the hand-optimized version both contribute to instruction-fetch bloat.

	Hand-coded	**Compiled**	**MIPS**	**x86-64**
code size (bytes)	756	1176	180	200
instructions fetched	622,861	1,576,952	424,860	582,467
instructions executed	761,072			
instruction traffic (bytes)	3,401,640	9,418,396	1,699,440	2,228,524

Table 11.2: Mandelbrot comparison on various ISA's.

11.3.5 Instructions executed

Aside from the hand-coded Mandelbrot implementation, the number of instructions fetched is identical to the number of instructions executed for the various architectures (also shown in table 11.2). The hand-coded implementation on Armada uses standing instructions that are fetched only once and repeat execution indefinitely. The results indicate that standing instructions reduced the instruction fetch count by nearly 20% when compared to the number of instructions executed. While this result does provide strong evidence that standing instructions reduce instruction fetches and fetch traffic, the number of fetches for Armada architectures is still higher than the number of fetches required by more traditional designs.

11.3.6 Concurrency

Thread-level concurrency is not currently supported by the Armada code generator. However, the compiler does formulate code bags which enables programs to achieve some degree of functional-unit concurrency. Figure 11.2 shows that the compiled code leaves the Ships idle nearly 20% more of the time than the hand-coded version does. Although neither implementation results in high levels of operation concurrency for a single thread, the compiled version is not unreasonably off-pace from the optimized code. This result suggests that more must be done fundamentally at the architecture and microarchitecture levels to enable higher levels of operation concurrency. As mentioned previously, one method of increasing single-thread ILC and Ship utilization is to implement a separate flow-control fabric on top of the existing switch fabric. The additional communication infrastructure can reduce communication latencies between Ships and more efficiently pipeline operations than is currently possible with Armada-1.

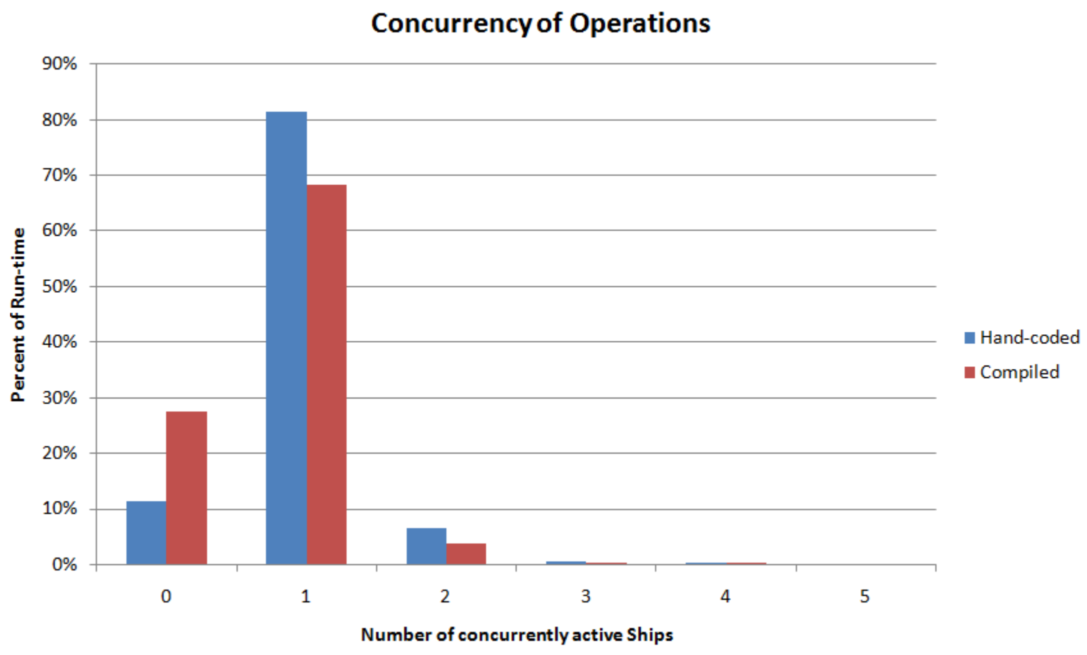


Figure 11.2: ILC comparison between hand-coded and compiled Mandelbrot. The benchmark implementations were executed on a single core, non-SMT configuration of ArmadaSim.

11.4 Conclusion

The compiled version of the benchmark has decent performance compared to the optimized code considering the compiler limitations and early stage of development. However, producing code around 41% slower performance than the optimized version on a single-core, non-SMT configuration, the compiler has room for improvement. More intelligent grouping of basic blocks into code bags, persistent input support, standing instruction support, and a dedicated flow-control fabric are the major keys to enhancing single-threaded performance. However, the biggest gains to be seen in programs and benchmarks with lots of thread-level concurrency available can be harnessed using Armada's TLC extensions. Limitations on research time prohibited further compiler enhancement leaving these improvements as future work.

Part IV

Discussion and conclusion

Chapter 12

Discussion and conclusion

The aim of this research was to enhance the highly-concurrent, communication-centric Fleet architecture and demonstrate that such an unconventional processor could be the basis for a general-purpose computer. Thread-level concurrency extensions were introduced, and a timing-approximate simulator of the architecture was created. The work undertaken generated a primitive but reasonably functional set of tools for the architecture including an imperative-language compiler, an assembler, and a debugger. The developed tool chain generated a reasonable implementation of the Mandelbrot benchmark. This work has thus provided strong evidence that the Armada architecture can indeed serve as general-purpose computer architecture.

However, the performance of the architecture appears to trail traditional CISC and RISC machines in terms of instruction fetch and execution efficiency. With lower code density, more instructions required to perform identical work, and higher instruction fetch traffic than those architectures, Armada puts high pressure on its shared instruction fetch and dispatch unit and instruction cache. However, the thread-level extensions incorporated into Armada, though not currently exploited by the compiler, do allow the performance of workloads with thread-level concurrency to scale greatly with the addition of more Fleets without any changes to program binaries.

The following sections discuss the merits of Fleet, the extensions introduced with Armada, and the compiler designed to target the architecture. Finally, limitations of the work performed and areas for future work are discussed.

12.1 Fleet changes

After some study, a decision was made to remove support for the numerous out-of-bounds (ie. *OOB*) possibilities envisioned by the pioneering Fleet work. The cost of checking for a number of possible conditions dictated by dynamically changing conditions would require a certain level of serialization of the code. As *OOB* values would be considered common, checks would be similarly frequent. Recovering from error cases would also be difficult especially in the absence of a hardware-reset for a Fleet core. Had this decision been made earlier in the research, the 96-bit data paths would have been replaced by smaller 65-bit paths throughout the design. Using 65-bits would accommodate common 64-bit data types coupled with the one *OOB* value that remains in the architecture, *last*. 96-bit code bag descriptors would not be treated as a first-order data type but instead be accessed indirectly through a 64-bit address.

Hardware enforcement of data types was seen as a distraction to the study of Fleet and Armada in this exploratory, proof-of-concept research. It was not considered in this study.

12.2 Architecture enhancements

A number of architecture enhancements were added to the Fleet concept to form Armada. These extensions are discussed here.

12.2.1 Proposed extensions and motivating factors

Armada introduces independent code bags to the ISA to provide programmers and compilers with a low-level method of spawning threads. The motivation was to make threading a natural and unavoidable part of programming. In the process, it was hoped that software written and compiled once could take advantage of additional cores added to an Armada processor over time as permitted by increasing transistor budgets.

The hardware supports the execution of multiple threads by two methods. Core replication produces multiple instances of physical resources like Ships and the switch fabric. Additionally, resource-multiplexing adds a layer of virtual channels on one physical Fleet and adds tags to instructions and data to uniquely identify separate thread contexts operating on top of the same hardware. Resource-multiplexing attempted to increase Ship utilization. Higher utilization would imply good use of the die area and reduce the impact of static, leakage power that accompanies enabled but seldom-used resources.

The context synchronizers were introduced to support barrier behavior and thread merging. Though not discussed in detail in this work, the units were tested and behaved as expected. The stack resource ring was implemented to lower the cost of spawning threads that require their own stacks.

12.2.2 Conclusions

These extensions perform admirably when executing the hand-coded Mandelbrot benchmark. The code can execute on a single Fleet yet exhibit increases in performance with the addition of up to 128 virtual Fleets. The best performance was achieved with a 16x8-way SMT configuration. Further improvement was hampered by saturation of the shared fetch, dispatch, and instruction cache units. Unfortunately, the compiler development did not progress enough to make use of these extensions. The compiled version of the benchmark does spawn threads, but the execution of those threads is serial; there is no TLC.

Ship utilization was increased successfully using the virtual channels. Although in principle this increased utilization would decrease the relevance of the static power consumed by the units, other techniques like implementing multiple power domains would also alleviate the problem. The cost of implementing virtual channels for simple operations like additions will likely be higher than simply duplicating adders for each channel. Virtual channels sharing large, complex Ships may be worthwhile, however. More study using additional benchmarks and more precise simulations will help answer this question.

The performance of the stack allocation ring was not thoroughly evaluated. Although it functioned as anticipated, it is not possible to fully understand the performance without understanding the costs associated with stack allocation by the operating system. When designing the unit, it was assumed that a traditional memory management unit would map physical memory onto a process' virtual memory map. The stack allocation units would have to perform such behavior as well, but this level of detail was not implemented.

12.3 Compiler and tools

The compiler, assembler, and debugger are essential requirements for general-purpose computing. Those tools were created to target Armada.

The compiled code's performance was 60% of the hand-coded, optimized version for single-core performance. This result is quite promising considering the limited optimizations built in to the tool. Fleet with a shared-memory subsystem accommodates imperative-language programming common to general-purpose computing. Static single assignment form, increasingly common in optimizing compilers, maps easily to move instructions. Given their single entry point and single exit point semantics, basic blocks also map well to Fleet and its code bags. However, for the Mandelbrot benchmark, there are so many data dependencies within each block that instruction-level concurrency and functional-unit concurrency are quite low. This result largely diminishes the value of code bags in exploiting ILC as envisioned. Although compiler enhancements may help, the optimized, hand-coded version of the benchmark was also not very successful at keeping many Ships utilized concurrently. Architecture changes such as the flow-control fabric suggested are required to increase this level of concurrency.

The debugger along with a simple waveform viewer proved to be useful tools for understanding what was happening in the highly-concurrent Armada architecture. However, these tools only test one possible sequence of events at a time. Although the simulator had randomness built-in to help test software more thoroughly, this feature is still inadequate to test all the possible cases. As instructions and data may arrive in vastly different arrangements, formal methods for ensuring correct behavior would be essential for validating the correctness of software. Software today has generally very limited concurrency, and there are few related bugs that have serious impact. For example, researchers have discovered concurrency-related bugs in operating systems for traditional architectures that have been hidden in the code for years. Though these bugs could deadlock the system, the failing conditions occur so rarely that these bugs are difficult to discover and fix. Fleet and Armada have such few restrictions on instruction ordering, data movement, and timing in general that any bug in software would be much more likely to appear.

12.4 Limitations of described work

The work performed covered a broad range of material from hardware simulation to software compilation. The uniqueness of the proposed architecture greatly limited reuse of existing infrastructure and tools. Thus, much of the research period was used to develop tools, models, and standards to evaluate Armada. Due to time constraints, there are several notable limitations that are described here.

12.4.1 Few benchmarks

One of the most prominent deficiencies in the work is the lack of benchmark results for Armada. One goal of the compiler was to make benchmark implementation easy so more benchmarks could be ported to the architecture. Time constraints ultimately prevented more benchmarks from being implemented and tested.

12.4.2 Performance comparison to modern architectures

ArmadaSim is a timing-approximate model. Although some time was spent researching and incorporating the measured performance of asynchronous implementations of functional units to Ships, the timing model was not detailed enough to fairly compare Armada performance against real silicon. An accurate performance model is important to this research as a drastic change to a new computer architecture will only occur in the general-purpose computing domain if the performance improvement is great.

12.4.3 Compiler maturity

The compiler is not complete and requires some manual intervention in the Ship-allocation phase. Optimizations for Armada have not been investigated. The work focused on proving that a traditional compiler could target Armada. This objective leaves enhancements as future work.

12.5 Areas for future work

In addition to the incomplete activities described in the limitations section, there are some other areas of broader study that are left open to future research.

12.5.1 Memory subsystem

The memory-processor performance gap is a notable problem in computer architecture. Fleet and Armada may be unique vehicles for exploring interesting memory architectures.

12.5.2 Higher granularity Ship functionality

Armada exposes communication at an extremely fine level of granularity. Exposing communications at this level may not be very useful as moving data between such small bits of logic sitting very near each other is not extremely costly. One area of study is to incorporate more behavior into Ships. For example, a Ship may be a hardware accelerator block for video encoding as opposed to a simple adder.

12.5.3 Armada as an application-specific processor architecture

Following from coordinating communication of data at a higher level of granularity is the possibility of generating application-specific designs quickly from a library of accelerators, cores, or other logic blocks. Because the switch fabric has so few constraints on the blocks attached to it, integrating new logic is easy. Additionally, software accesses the logic uniformly with moves to and from numbered addresses. Therefore, the ISA remains relatively unchanged with the exception of port assignments for the logic. That factor is not particularly problematic for ASP's where software is purpose-built to perform a specific task using a specific set of hardware resources.

12.5.4 Flow-control fabric

Armada-1 does not exhibit high single-threaded concurrency of operations. A flow-control fabric would enable Ships to communicate data more quickly between each other especially in the case where standing instructions are used.

12.5.5 Flow caching

Flow caching was described in chapter 4 as a means to increase performance by caching commonly-executed code onto Fleet cores. By caching these flows, instruction fetches are largely avoided resulting in decreased instruction-side pressure and latencies. Flow caching may prove to be a viable way of making reconfigurable computing easily accessible to programmers.

12.5.6 Compiler enhancements

Many optimization and enhancement opportunities remain for the LLVM code generator implemented for this research. Standing instruction support and persistent input support are key lacking features. Additionally, a much larger outstanding problem is how to have an imperative language compiler generate threaded code.

12.6 Final remarks

Armada builds on Fleet in design and in spirit. Using Fleet cores as fundamental building blocks, Armada promotes exposing expensive communication and makes concurrency a requirement rather than an optimization. Current progress into the Armada research has demonstrated that the architecture is highly amenable to ubiquitous, imperative-language programming and may possibly serve as a general-purpose computer architecture for the future. More investigation is required, however, to prove if its use in this domain is practical.

Bibliography

[AHKB00] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA '00: Proceedings of the 27th international symposium on Computer Architecture*, pages 248–259, 2000.

[BD95] Mark W. Bailey and Jack W. Davidson. A formal model and specification language for procedure calling conventions. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 298–310, 1995.

[BDJS06] Philip Brisk, Foad Dabiri, Roozbeh Jafari, and Majid Sarrafzadeh. Optimal register sharing for high-level synthesis of SSA form programs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(5):772 – 779, 2006.

[BDR07] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of spill everywhere under SSA form. *SIGPLAN Notices*, 42(7):103–112, 2007.

[BG04] D. Burger and JR Goodman. Billion-transistor architectures: there and back again. *Computer*, 37(3):22–28, 2004.

[BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, 2000.

[CLJ$^+$01] William S. Coates, Jon K. Lexau, Ian W. Jones, Scott M. Fairbanks, and Ivan E. Sutherland. FLEETzero: An asynchronous switching experiment. In *ASYNC '01: Proceedings of the 7th international symposium on Asynchronous Circuits and Systems*, pages 173 – 182, 2001.

[CM91] Henk Corporaal and Hans (J.M.) Mulder. MOVE: a framework for high-performance processor design. In *Supercomputing '91: Proceedings of the 1991 ACM / IEEE conference on Supercomputing*, pages 692–701, 1991.

[DM75] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *ISCA '75: Proceedings of the 2nd annual symposium on Computer Architecture*, pages 126–132, 1975.

[End95] P. B. Endecott. *SCALP: A Superscalar Asynchronous Low-Power Processor*. PhD thesis, University of Manchester, Manchester, UK, 1995.

[EO88] Carla Schlatter Ellis and Thomas J. Olson. Algorithms for parallel memory allocation. *Int. J. Parallel Program.*, 17(4):303–345, 1988.

[FH05] M.J. Flynn and P. Hung. Microprocessor design issues: thoughts on the road ahead. *IEEE Micro*, 25, 2005.

[FKM$^+$02] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer Architecture*, pages 148–157, 2002.

[For03] Martti Forsell. Analysis of transport triggered architectures in general purpose computing. In *Proceedings of the 21st IEEE Norchip Conference*, pages 183–186, 2003.

[FPT94] Matthew Farrens, Andrew R. Pleszkun, and Gary Tyson. A study of single-chip processor / cache organizations for large numbers of transistors. *SIGARCH Computer Architecture News*, 22(2):338–347, 1994.

[Gel01] P.P. Gelsinger. Microprocessors for the new millennium: Challenges, opportunities, and new frontiers. In *ISSCC '01: International Solid-State Circuits Conference Digest of Technical Papers*, pages 22–25, 2001.

[GG98] J. Gonzalez and A. Gonzalez. Limits of instruction level parallelism with data speculation. *Proceedings of the VECPAR Conference*, pages 585–598, 1998.

[GKW85] J. R Gurd, C. C Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.

[Hac05] Sebastian Hack. Interference graphs of programs in SSA-form. Technical report, University Karlsruhe, June 2005.

[HGG06] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *Compiler Construction 2006*, volume 3923 of *Lecture Notes In Computer Science*, pages 247–262, March 2006.

[HGLS86] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[HMH01] R. Ho, K.W. Mai, and M.A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[Hol05] Mike Holenderski. Accumulating the Fibonacci sequence using FLEET. Technical memo UCMH#2005-mh02, 2005.

[HP03] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

[Isa06] Nemanja Isailovic. Eight-element bubble sort in FLEET. Technical memo, 2006.

[ITR05] ITRS. *International Technology Roadmap for Semiconductors (ITRS) 2005 Edition: Design*. Japan Electronics and Information Technology Industries Association, 2005.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Computer Architecture News*, 18(3a):364–373, 1990.

[KAB$^+$03] Nam Sung Kim, Todd Austin, David Blaauw, Trevor Mudge, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003.

[KM03] D. Koufaty and D.T. Marr. Hyperthreading technology in the net-burst microarchitecture. *IEEE Micro*, 23(2):56–65, 2003.

[KP02] Christoforos Kozyrakis and David Patterson. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. In *MICRO 35: Proceedings of the 35th annual ACM / IEEE international symposium on Microarchitecture*, pages 283–293, 2002.

[Lin07] Andrew Lines. The Vortex: A superscalar asynchronous processor. In *ASYNC '07: Proceedings of the 13th IEEE international symposium on Asynchronous Circuits and Systems*, pages 39–48, 2007.

[MB05] C. McNairy and R. Bhatia. Montecito: a dual-core, dual-thread Itanium processor. *IEEE Micro*, 25(2):10–20, 2005.

[Mey06a] Trevor Meyerowitz. Euclid's algorithm in FLEET take 2. Technical memo, 2006.

[Mey06b] Trevor Meyerowitz. Matrix-vector multiplication in FLEET. Technical memo, 2006.

[MJC$^+$99] C.E. Molnar, I.W. Jones, W.S. Coates, J.K. Lexau, S.M. Fairbanks, and I.E Sutherland. Two FIFO ring performance experiments. In *Proceedings of the IEEE*, volume 87, pages 297–307, 1999.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[MPJ$^+$00] K. Mai, T. Paaske, N. Jayasena, R. Ho, WJ Dally, and M. Horowitz. Smart Memories: a modular reconfigurable architecture. *Computer Architecture, 2000. Proceedings of the 27th international symposium on*, pages 161–171, 2000.

[Ope06] Open SystemC Initiative. Homepage: *SystemC: Welcome.* http://www.systemc.org/, September 2006.

[PWD$^+$09] Songwen Pei, Baifeng Wu, Min Du, Gang Chen, Leandro A. J. Marzulo, and Felipe M. G. Franca. Spmt wavecache: Exploiting thread-level parallelism in wavescalar. In *CSIE '09: Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering*, pages 530–535, Washington, DC, USA, 2009. IEEE Computer Society.

[RF93] B.R. Rau and J.A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7(1):9–50, 1993.

[SMSO03] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *MICRO 36: Proceedings of the 36th annual IEEE / ACM international symposium on Microarchitecture*, pages 291 – 302, 2003.

[SNL$^+$03] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S.W. Keckler, and C.R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *ACM SIGARCH Computer Architecture News*, 31(2):422, 2003.

[SR00] M.S. Schlansker and B.R. Rau. EPIC: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, 2000.

[Sut05] I. E. Sutherland. FLEET – A One-Instruction Computer. Memo to Berkeley students, December 2005.

[TEL98] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 533–544, New York, NY, USA, 1998. ACM Press.

[TLM$^+$02] M.B. Taylor, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, et al. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[TLM$^+$04] MB Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, et al. Evaluation of the Raw microprocessor: an exposed-wire-delay architecture for ILP and streams. *Computer Architecture, 2004. Proceedings. 31st annual international symposium on*, pages 2–13, 2004.

[TS88] M.R. Thistle and B.J. Smith. A processor architecture for horizon. *Supercomputing '88: Proceedings*, 1:35–41, 14-18 Nov 1988.

[vESV$^+$99] J.T.J. van Eijndhoven, F.W. Sijstermans, K.A. Vissers, E.J.D. Pol, M.I.A. Tromp, P. Struik, R.H.J. Bloks, P. van der Wolf, A.D. Pimentel, and H.P.E. Vranken. TriMedia CPU64 architecture. In *Computer Design, 1999. (ICCD '99) International Conference on*, pages 586–592, Austin, TX, USA, 1999.

[VH99] Voon-Yee Vee and Wen-Jing Hsu. A scalable and efficient storage allocator on shared memory multiprocessors. In *ISPAN '99: Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks*, page 230, Washington, DC, USA, 1999. IEEE Computer Society.

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, 1991.

[Wal95] David W. Wall. *Limits of instruction-level parallelism*, pages 432–444. IEEE Computer Society Press, 1995.

[Wei06] Eric W. Weisstein. Homepage: *Fractal.* http://mathworld.wolfram.com/Fractal.html, August 2006.

[Wil94] Ted E. Williams. Performance of iterative computation in self-timed rings. *Journal of VLSI Signal Processing*, 7(1-2):17–31, 1994.

[Yi05] Kwangkeun Yi, editor. *Register Allocation via Coloring of Chordal Graphs*, volume 3780 of *Lecture Notes in Computer Science.* Springer, 2005.

[ZW03] Thomas Zeitlhofer and Bernhard Wess. List-coloring of interval graphs with application to register assignment for heterogeneous register-set architectures. *Signal Processing*, 83(7):1411–1425, 2003.

Appendix A

Ships in Armada-1

A.1 Register

Description

Holds a data value.

Input ports

in: write value to register; must not be persistent

Output ports

out: read value from register

Mini-opcodes

none

A.2 IntAdd64

Description

Adds two signed 64-bit integers.

Input ports

in_lhs: left-hand-side of operation; only one of in_lhs and in_rhs may be persistent
in_rhs: right-hand-side of operation; only one of in_lhs and in_rhs may be persistent

Output ports

out: result

Mini-opcodes

in_lhs and in_rhs: change the sign of either input (multiply by -1) by moving with miniop=1

A.3 IntDiv64

Description

Divides two signed 64-bit integers.

Input ports

in_lhs: left-hand-side of operation; only one of in_lhs and in_rhs may be persistent
in_rhs: right-hand-side of operation; only one of in_lhs and in_rhs may be persistent

Output ports

outQ: quotient
outR: remainder

Mini-opcodes

in_lhs and in_rhs: change the sign of either input (multiply by -1) by moving with miniop=1

A.4 IntMul64

Description

Multiply two signed 64-bit integers.

Input ports

in_lhs: left-hand-side of operation; only one of in_lhs and in_rhs may be persistent
in_rhs: right-hand-side of operation; only one of in_lhs and in_rhs may be persistent

Output ports

out: result

Mini-opcodes

in_lhs and in_rhs: change the sign of either input (multiply by -1) by moving with miniop=1

A.5 FPAdd64

Description

Adds two 64-bit double-precision floating point numbers.

Input ports

in_lhs: left-hand-side of operation; only one of in_lhs and in_rhs may be persistent
in_rhs: right-hand-side of operation; only one of in_lhs and in_rhs may be persistent

Output ports

out: result

Mini-opcodes

in_lhs and in_rhs: change the sign of either input (multiply by -1) by moving with miniop=1

A.6 FPDiv64

Description

Divides two 64-bit double-precision floating point numbers.

Input ports

in_lhs: left-hand-side of operation; only one of in_lhs and in_rhs may be persistent
in_rhs: right-hand-side of operation; only one of in_lhs and in_rhs may be persistent

Output ports

out: result

Mini-opcodes

in_lhs and in_rhs: change the sign of either input (multiply by -1) by moving with miniop=1

A.7 FPMul64

Description

Multiplies two 64-bit double-precision floating point numbers.

Input ports

in_lhs: left-hand-side of operation; only one of in_lhs and in_rhs may be persistent
in_rhs: right-hand-side of operation; only one of in_lhs and in_rhs may be persistent

Output ports

out: result

Mini-opcodes

in_lhs and in_rhs: change the sign of either input (multiply by -1) by moving with miniop=1

A.8 BitShift

Description

Bit shifts 32- or 64-bit integers.

Input ports

in_lhs: base value; only one of in_lhs and in_rhs may be persistent
in_rhs: amount to shift by; only one of in_lhs and in_rhs may be persistent

Output ports

out: result

Mini-opcodes

in_lhs: 0: 32_BIT
    1: 64_BIT
in_rhs:
    0: shift left (LSL)
    2: logical shift right (LSR)
    3: logical shift left (LSL)

A.9 BitOp

Description

Perform bitwise operations.

Input ports

in1: val1
in2: val2

Output ports

out: result

Mini-opcodes

in_lhs: none
in_rhs:
    0: AND
    1: OR
    2: XOR

A.10 IntToFP

Description

Converts a 64-bit integer to a 64-bit double-precision floating point number.

Input ports

in: integer to convert; may not be persistent

Output ports

out: floating point result

Mini-opcodes

in: change the sign (multiply by -1) by moving with miniop=1

A.11 SExt

Description

Sign-extends a 32-bit integer to 64-bits

Input ports

in: integer to sign-extend; may not be persistent

Output ports

out: result

Mini-opcodes

none

A.12 Counter

Description

Counts a number of tokens generating an output token, *last*, once all tokens have arrived.

Input ports

in_tok: port where tokens are counted; may not be persistent
in_cnt: number of tokens to wait for; may not be persistent

Output ports

out: generates *last* token when all inputs have arrived

Mini-opcodes

none

A.13 Stride

Description

Generates a series of numbers and a *last* token when complete

Input ports

in_start: first value to generate
in_step: increments output value by this amount for each subsequent value generated
in_stop: maximum bound; if output value will exceed this bound for positive step or be less than this bound for negative step, the *last* token is generated and state cleared
in_next: any token sent to this port other than *last* will cause Ship to generate the next output; a *last* token received at this port will reset Ship state

Output ports

out: output tokens

Mini-opcodes

none

A.14 Cmp

Description

Compares two tokens generating a boolean evaluation output.

Input ports

in_lhs: left-hand-side value to compare; one of lhs or rhs may be peristent
in_rhs: right-hand-side value to compare; one of lhs or rhs may be peristent
in_op: type of comparison to perform; may be persistent
in_reset: any token received at this port will cause Ship to reset

Op	Mnemonic	Description
0	`INT_EQ`	integer, ==
1	`INT_NE`	integer, ! =
2	`INT_UGT`	unsigned integer, >
3	`INT_UGE`	unsigned integer, >=
4	`INT_ULT`	unsigned integer, <
5	`INT_ULE`	unsigned integer, <=
6	`INT_SGT`	signed integer, >
7	`INT_SGE`	signed integer, >=
8	`INT_SLT`	signed integer, <
9	`INT_SLE`	signed integer, <=
16	`FP_EQ`	double-precision floating-point, ==
17	`FP_NE`	double-precision floating-point, ! =
18	`FP_GT`	double-precision floating-point, >
19	`FP_GE`	double-precision floating-point, >=
20	`FP_LT`	double-precision floating-point, <
21	`FP_LE`	double-precision floating-point, <=

Table A.1: Cmp Ship in_op input port values.

Output ports

out: boolean result of comparison

Mini-opcodes

none

A.15 Join

Description

There are two variants of the join Ship used in testing, a 2-way and 4-way join. Ship takes 2 and 4 tokens, respectively, as inputs. When all inputs arrive, input 1 is passed forward to the output port.

Input ports

in_pass: value that is passed forward to the output port once all other inputs have arrived; may be persistent
in_2: input to wait for; may not be persistent
in_3: input to wait for; may not be persistent
in_4: input to wait for; may not be persistent

Output ports

out: delivers token from in_pass once all other inputs have also arrived

Mini-opcodes

none

A.16 Selector

Description

Passes one of two inputs forward to the output port based on boolean input.

Input ports

in_true: input to pass forward if boolean input is true; may be persistent
in_false: input to pass forward if boolean input is false; may be persistent
in_sel: boolean value that dictates input to forward; may not be persistent; *last* token delivered to this input resets the Ship

Output ports

out: selected input forwarded here

Mini-opcodes

none

A.17 Toggle

Description

Delivers two tokens in a predefined order despite the order in which those tokens arrive at the Ship. It may deliver the first token to the output port even if the second input has not yet arrived

Input ports

in_1: first data value to forward; may not be persistent
in_2: second data value to forward; may not be persistent

Output ports

out: delivers first data value followed by second data value

Mini-opcodes

none

A.18 Fetch

Description

Fetches code bags and forwards values to new threads.

Input ports

in_cbd: code bag descriptor of bag to fetch; may not be persistent; *last* delivered to this port signals hardware that the Fleet core is ready for reuse
in_r0: forwarding register; may be persistent
in_r1: forwarding register; may be persistent
in_r2: forwarding register; may be persistent
in_r3: forwarding register; may be persistent
in_r4: forwarding register; may be persistent
in_r5: forwarding register; may be persistent
in_sp: forwarding register; may be persistent
in_lr: forwarding register; may be persistent

Output ports

none

Mini-opcodes

none

A.19 Memory

Description

Reads and writes to shared system memory.

Input ports

in_rdAddr: address to read from; may not be persistent
in_wrAddr: address to write to; one of in_wrAddr and in_wrData may be persistent
in_wrData: data to write; one of in_wrAddr and in_wrData may be persistent

Output ports

out_rdData: data read from memory
out_wrComp: write completion indicator token

Mini-opcodes

Opcodes representing size of data to read or write from memory are read by the in_rdAddr and in_wrAddr ports.

0: 8-bit
1: 16-bit
2: 32-bit
3: 64-bit

A.20 ContextSynchronizer

Description

Provides thread barrier operation. Waits for threads to complete. Once all done, fetches an independent code bag to continue program flow.

Input ports

in_icbd: independent code bag descriptor to fetch once barrier threads complete; may not be persistent
in_cnt: number of threads to wait for
in_r0—in_r7: registers to forward to code bag fetched after barrier
in_decrement: threads barrier is waiting for send the synchronizer reference to decrement here when they complete; when all barrier threads complete, hardware fetches the post-barrier code bag

Output ports

out_reference: returns a reference to a synchronizer object; should be forwarded to child threads which pass this reference to the in_decrement port when they complete
out_decrement_conf: generated to confirm synchronizer reference was accepted by the in_decrement port

Mini-opcodes

none

A.21 Stack

Description

Supplies software with pointers to stacks that can be assigned to new threads; software also returns stacks no longer needed to this Ship

Input ports

in_stack_req: any token sent to this port requests a stack memory pointer
in_stack: stack pointer previously allocated by this Ship that is no longer needed by software

Output ports

out_stack: response from in_stack_reg; outputs pointer to stack memory for software to use

Mini-opcodes

none

Appendix B

Hardware stack allocation support

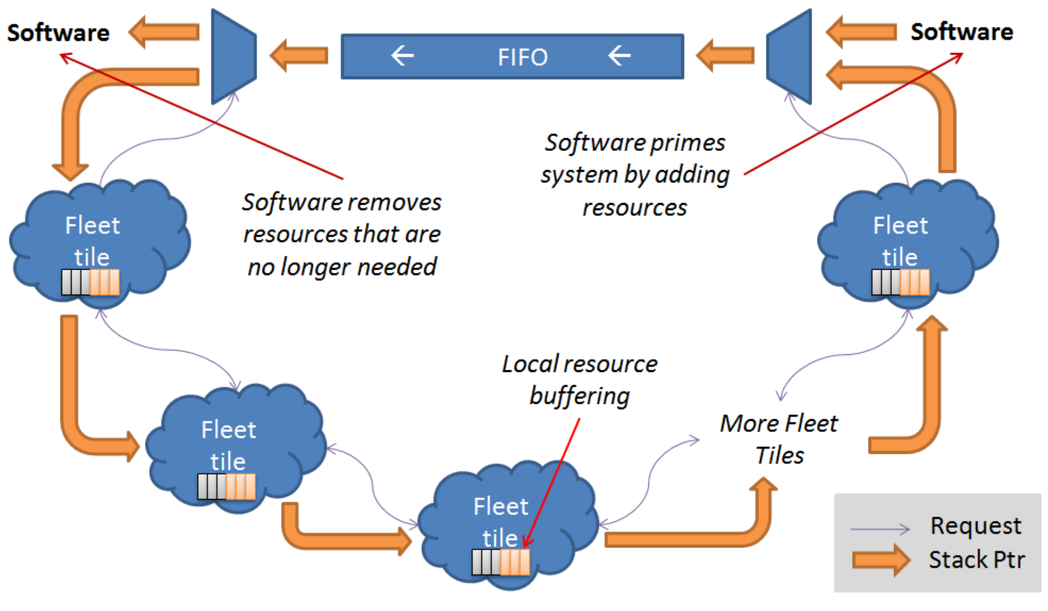
Armada-1 has hardware support for low-latency stack memory allocation when spawning threads. Caller code can issue a stack allocation request to a special stack Ship in the Fleet. This ship will return a pointer to memory that has been pre-allocated for this purpose. Ideally, the memory would be mapped into the running process' page table. Armada-1 does not have a memory management unit (MMU); so, this behavior is not implemented.

Although this support was implemented in Armada-1, very little research was done regarding the efficiency of the mechanism due to time constraints. Without an MMU, the merits of the mechanism are difficult to quantify. It is also not used by the benchmarks tested. Therefore, the details of the implementation have been relegated to this appendix.

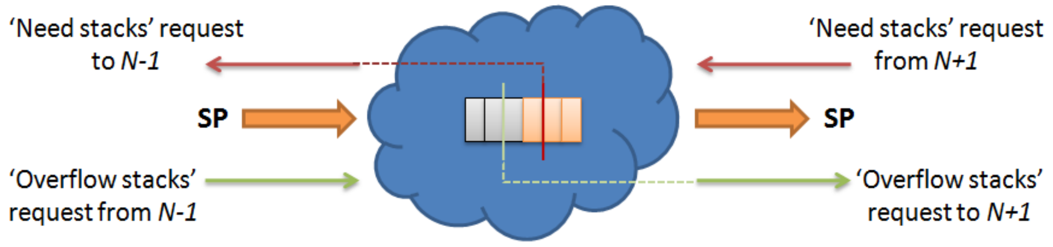
B.1 Overview

The idea behind pre-allocating stacks is to decrease the latency of spawning many concurrent threads that require their own stack. As Armada was designed to allow software to spawn many threads quickly with low-overhead, hardware support seemed appealing. A ship in each Fleet core buffers some amount of memory pages reserved for these stacks. When a Fleet requests a stack, the allocation is local avoiding latencies associated with any centralized resource scheme.

In this implementation, the ships in the Fleet cores are connected to two of their neighbors as in figure B.1. A ship can request stacks from one neighboring core. It can push stacks to the other neighbor.



(a)


'Need stacks' request to N-1
SP
'Overflow stacks' request from N-1
'Need stacks' request from N+1
SP
'Overflow stacks' request to N+1

(b)

Figure B.1: Hardware stack allocation scheme. Each Fleet in the Armada has a local stack Ship that buffers stack resources (a). They are connected to one another in a ring formation. One segment of the ring interfaces with the operating system to allow stacks to be inserted and removed from the ring. Each stack Ship can request stacks from one neighbor and push stacks to the other (b).

B.2 Priming the ring

Initially, all cores are starved for stacks and will send requests for memory to their neighbors. These requests ultimately trigger an event (ie. an interrupt) demanding operating system attention. The operating system allocates memory for stacks and pushes them into a hardware FIFO. That FIFO feeds into one core in the ring of Fleets. Stacks are propagated through the ring until all stack Ship buffers meet their minimum watermark level.

B.3 Stack allocation

When a fetch Ship in a core fetches an independent code bag descriptor and the hardware stack allocation method is specified by the programmer, a pointer to one of the locally-buffered stacks is returned. If that causes the local stack buffer to fall below its low watermark, it will request another stack from its neighbor.

B.4 Stack release

When stacks are released via the local stack Ship, the stack memory is returned to the local pool of available stack resources. If many stacks are freed at once, the local buffers may become full. When those buffers reach a high watermark, they begin pushing stacks to their neighbors. Ultimately, the Fleet that terminates the ring and interfaces with the software-accessible FIFO may push those stacks out of the ring. This again initiates an event that is handled by the operating system.

B.5 Testing

The stack ring was functionally tested with a directed test. This test ensured the system primed itself by requesting stacks from the operating system. This program in turn also functionally tested the Armada event-handling system. Once the system was primed, test code spawned many threads that requested hardware stack allocation. The ring did not have enough buffers from its initial priming to fulfill all these requests, and additional events provoked the software to top-up the ring by pushing in more stacks. Finally, these threads terminated and released their stacks in near unison. This caused the ring to fill up with stacks and cause more events to push the stacks out of the ring. The event handler responded by reading out of the FIFO effectively ’freeing’ those stacks.

Appendix C

Code listings

C.1 Mandelbrot, hand-coded

```
// input aliases for Tile 0
alias Register_0.in 0;
alias Register_1.in 1;
alias Register_2.in 2;
alias Register_3.in 3;
alias Register_4.in 4;
alias Register_5.in 5;
alias Register_6.in 6;
alias Register_7.in 7;

alias Fetch_0.in_cbd 8;
alias Fetch_0.in_reg0 9;
alias Fetch_0.in_reg1 10;
alias Fetch_0.in_reg2 11;
alias Fetch_0.in_reg3 12;
alias Fetch_0.in_reg4 13;
alias Fetch_0.in_reg5 14;
alias Fetch_0.in_reg6 15;
alias Fetch_0.in_reg7 16;

alias Cmp_0.in_lhs 17;
alias Cmp_0.in_rhs 18;
alias Cmp_0.in_op 19;
alias Cmp_0.in_rst 20;
alias Cmp_1.in_lhs 21;
alias Cmp_1.in_rhs 22;
alias Cmp_1.in_op 23;
alias Cmp_1.in_rst 24;

alias FourJoin_0.in1 25;
alias FourJoin_0.in2 26;
alias FourJoin_0.in3 27;
alias FourJoin_0.in4 28;
alias FourJoin_1.in1 29;
alias FourJoin_1.in2 30;
alias FourJoin_1.in3 31;
alias FourJoin_1.in4 32;

alias Memory_0.in_rdAddr 33;
alias Memory_0.in_wrAddr 34;
alias Memory_0.in_wrData 35;

alias Selector_0.in_sel 36;
alias Selector_0.in_true 37;
alias Selector_0.in_false 38;
alias Selector_1.in_sel 39;
alias Selector_1.in_true 40;
alias Selector_1.in_false 41;

alias Stride_0.in_start 42;
alias Stride_0.in_step 43;
alias Stride_0.in_stop 44;
alias Stride_0.in_next 45;
alias Stride_1.in_start 46;
alias Stride_1.in_step 47;
alias Stride_1.in_stop 48;
alias Stride_1.in_next 49;

alias Toggle_0.in1 50;
alias Toggle_0.in2 51;
alias Toggle_1.in1 52;
alias Toggle_1.in2 53;

alias IntToFP_0.in 54;
alias IntToFP_1.in 55;
alias IntToFP_2.in 56;
alias IntToFP_3.in 57;

alias FPMul64_3.in1 58;
alias FPMul64_3.in2 59;

alias IntAdd64_0.in1 64;
alias IntAdd64_0.in2 65;
alias IntAdd64_1.in1 66;
alias IntAdd64_1.in2 67;
alias IntAdd64_2.in1 68;
alias IntAdd64_2.in2 69;
alias IntAdd64_3.in1 70;
alias IntAdd64_3.in2 71;
alias IntAdd64_4.in1 72;
alias IntAdd64_4.in2 73;
alias IntAdd64_5.in1 74;
alias IntAdd64_5.in2 75;
alias IntAdd64_6.in1 76;
alias IntAdd64_6.in2 77;
alias IntAdd64_7.in1 78;
alias IntAdd64_7.in2 79;
alias IntAdd64_8.in1 80;
alias IntAdd64_8.in2 81;
alias IntAdd64_9.in1 82;
alias IntAdd64_9.in2 83;
alias IntAdd64_10.in1 84;
alias IntAdd64_10.in2 85;
alias IntAdd64_11.in1 86;
alias IntAdd64_11.in2 87;

alias IntDiv64_0.in1 88;
alias IntDiv64_0.in2 89;
alias IntDiv64_1.in1 90;
alias IntDiv64_1.in2 91;
alias IntDiv64_2.in1 92;
alias IntDiv64_2.in2 93;
alias IntDiv64_3.in1 94;
alias IntDiv64_3.in2 95;

alias IntMul64_0.in1 96;
alias IntMul64_0.in2 97;
alias IntMul64_1.in1 98;
alias IntMul64_1.in2 99;
alias IntMul64_2.in1 100;
alias IntMul64_2.in2 101;
alias IntMul64_3.in1 102;
alias IntMul64_3.in2 103;

alias FPAdd64_0.in1 104;
alias FPAdd64_0.in2 105;
alias FPAdd64_1.in1 106;
alias FPAdd64_1.in2 107;
alias FPAdd64_2.in1 108;
alias FPAdd64_2.in2 109;
alias FPAdd64_3.in1 110;
alias FPAdd64_3.in2 111;

alias FPDiv64_0.in1 112;
alias FPDiv64_0.in2 113;
alias FPDiv64_1.in1 114;
alias FPDiv64_1.in2 115;
alias FPDiv64_2.in1 116;
alias FPDiv64_2.in2 117;
alias FPDiv64_3.in1 118;
alias FPDiv64_3.in2 119;

alias FPMul64_0.in1 120;
alias FPMul64_0.in2 121;
alias FPMul64_1.in1 122;
alias FPMul64_1.in2 123;
alias FPMul64_2.in1 124;
alias FPMul64_2.in2 125;
//
alias BitBucket.in 127;


// output aliases for Tile 0
alias Register_0.out 0;
alias Register_1.out 1;
alias Register_2.out 2;
alias Register_3.out 3;
alias Register_4.out 4;
alias Register_5.out 5;
alias Register_6.out 6;
alias Register_7.out 7;

alias Literal.out 8;

alias Cmp_0.out 17;
alias Cmp_1.out 21;

alias FourJoin_0.out 25;
alias FourJoin_1.out 29;

alias Memory_0.out_rdData 33;
alias Memory_0.out_wrComp 34;

alias Selector_0.out 36;
alias Selector_1.out 39;

alias Stride_0.out 42;
alias Stride_1.out 46;

alias Toggle_0.out 50;
alias Toggle_1.out 52;

alias IntToFP_0.out 54;
alias IntToFP_1.out 55;
alias IntToFP_2.out 56;
alias IntToFP_3.out 57;

alias FPMul64_3.out 58;

alias IntAdd64_0.out 64;
alias IntAdd64_1.out 66;
alias IntAdd64_2.out 68;
alias IntAdd64_3.out 70;
alias IntAdd64_4.out 72;
alias IntAdd64_5.out 74;
alias IntAdd64_6.out 76;
alias IntAdd64_7.out 78;
alias IntAdd64_8.out 80;
alias IntAdd64_9.out 82;
alias IntAdd64_10.out 84;
alias IntAdd64_11.out 86;

alias IntDiv64_0.outQ 88;
alias IntDiv64_0.outR 89;
alias IntDiv64_1.outQ 90;
alias IntDiv64_1.outR 91;
alias IntDiv64_2.outQ 92;
alias IntDiv64_2.outR 93;
alias IntDiv64_3.outQ 94;
alias IntDiv64_3.outR 95;

alias IntMul64_0.out 96;
alias IntMul64_1.out 98;
alias IntMul64_2.out 100;
alias IntMul64_3.out 102;

alias FPAdd64_0.out 104;
alias FPAdd64_1.out 106;
alias FPAdd64_2.out 108;
alias FPAdd64_3.out 110;

alias FPDiv64_0.out 112;
alias FPDiv64_1.out 114;
alias FPDiv64_2.out 116;
alias FPDiv64_3.out 118;

alias FPMul64_0.out 120;
alias FPMul64_1.out 122;
alias FPMul64_2.out 124;



//// MANDELBROT ////
initial codebag reset 0{
(dispatch) -> Fetch_0.in_cbd;
} reset;

// Put the various programs to run here
dependent codebag dispatch 0 {
(Main) -> Toggle_0.in1;
"OOB" -> Toggle_0.in2;
Toggle_0.out => Fetch_0.in_cbd;
} dispatch;

```

```
independent codebag Main 0{
// viewport.width
(4.0D) -> FPDiv64_0.in1;
// viewport.height
(3.0D) -> FPDiv64_1.in1;
// viewport.getMinY
(-1.5D) -> *FPAdd64_0.in2;

"OOB" -> *Cmp_0.in_rhs;
(OuterLoop) -> *Selector_0.in_true;
(MainCleanup) -> *Selector_0.in_false;


// window.width
(320) -> IntToFP_0.in, Fetch_0.in_reg4;
IntToFP_0.out -> FPDiv64_0.in2;
// window.height
(240) -> IntToFP_1.in, Stride_0.in_stop;

IntToFP_1.out -> FPDiv64_1.in2;

FPDiv64_0.out -> Fetch_0.in_reg3;
FPDiv64_1.out -> *FPMul64_0.in2;

(0) -> Stride_0.in_start;
(1) -> Stride_0.in_step, Stride_0.in_next;
Stride_0.out => IntToFP_2.in, Register_0.in;
IntToFP_2.out => FPMul64_0.in1;
Register_0.out => Cmp_0.in_lhs, Fetch_0.in_reg1;

FPMul64_0.out => FPAdd64_0.in1;
FPAdd64_0.out => Fetch_0.in_reg2, Stride_0.in_next;

(1) -> *Cmp_0.in_op;
Cmp_0.out => Selector_0.in_sel;
Selector_0.out => Fetch_0.in_cbd;
} Main;

dependent codebag MainCleanup 0 (r1, r2, r3, r4) {
Register_4.out -> Stride_0.in_start, Stride_0.in_stop;
Stride_0.out -> FourJoin_0.in1;
Register_1.out -> FourJoin_0.in2;
Register_2.out -> FourJoin_0.in3;
Register_3.out -> FourJoin_0.in4, Stride_0.in_step;
FourJoin_0.out -> Cmp_0.in_rst;
} MainCleanup;


// window.width, i, c_i, widthRatio
independent codebag OuterLoop 0 (r1, r2, *r3, *r4) {
"OOB" -> *Cmp_0.in_rhs;

// viewport.getMinX
(-2.5D) -> *FPAdd64_0.in2;

(InnerLoop) -> *Selector_0.in_true;
(OuterLoopCleanup) -> *Selector_0.in_false;


Register_4.out -> Stride_0.in_stop, Fetch_0.in_reg4;
(0) -> Stride_0.in_start;
(1) -> Stride_0.in_step, Stride_0.in_next;
Stride_0.out => IntToFP_0.in, Register_0.in;
Register_0.out => Cmp_0.in_lhs, Fetch_0.in_reg1;
IntToFP_0.out => FPMul64_0.in1;

(1) -> *Cmp_0.in_op;
Cmp_0.out => Selector_0.in_sel;
Selector_0.out => Fetch_0.in_cbd;

Register_1.out -> Fetch_0.in_reg0;
Register_2.out -> Fetch_0.in_reg2;
Register_3.out -> *FPMul64_0.in2;
FPMul64_0.out => FPAdd64_0.in1, Stride_0.in_next;
FPAdd64_0.out => Fetch_0.in_reg3;
} OuterLoop;

dependent codebag OuterLoopCleanup 0 (r0, r1, r2, r3, r4){
Register_0.out -> Stride_0.in_start;
Register_1.out -> Stride_0.in_stop;
Register_2.out -> Stride_0.in_step;
Register_3.out -> FourJoin_0.in1;
Register_4.out -> FourJoin_0.in2;
Stride_0.out -> FourJoin_0.in3, FourJoin_0.in4;
FourJoin_0.out -> Cmp_0.in_rst;
} OuterLoopCleanup;

// i, j, c_i, c_r, window.width
independent codebag InnerLoop 0 (*r0, r1, *r2, r3, *r4) {
// init z_r, z_i
(0.0D) -> Register_5.in, Register_6.in;

"OOB" -> *Cmp_0.in_lhs;
(InnerLoop_compute) -> *Selector_0.in_true;
(InnerLoop_cleanup) -> *Selector_0.in_false;

(100000.0D) -> *Cmp_1.in_rhs;
(InnerLoop_cleanup2) -> *Selector_1.in_true;
(InnerLoop_next_itr) -> *Selector_1.in_false;

(1)  -> *Cmp_0.in_op;
(18) -> *Cmp_1.in_op;

Register_3.out -> *FPAdd64_1.in2, FourJoin_0.in2;
Register_2.out -> *FPAdd64_2.in2, FourJoin_0.in3;

(-1) -> FourJoin_0.in1, FourJoin_0.in4;
FourJoin_0.out -> Stride_0.in_start, Memory_0.in_wrData;
(1) -> Stride_0.in_step, Stride_0.in_next;
(256) -> Stride_0.in_stop;
Stride_0.out => Cmp_0.in_rhs, Register_7.in;
Cmp_0.out => Selector_0.in_sel;
Selector_0.out => Fetch_0.in_cbd;

Cmp_1.out => Selector_1.in_sel;
} InnerLoop;

dependent codebag InnerLoop_compute 0 {
Register_5.out -> FPMul64_0.in1, Register_2.in;
Register_2.out -> FPMul64_0.in2, FPMul64_2.in1;

Register_6.out -> FPMul64_1.in1, Register_3.in;
Register_3.out -> FPMul64_1.in2, FPMul64_2.in2;

FPMul64_0.out -> FPAdd64_0.in1, FPAdd64_3.in1;
FPMul64_1.out -> FPAdd64_0.in2(1), FPAdd64_3.in2;
FPAdd64_0.out -> FPAdd64_1.in1;
FPAdd64_1.out -> Register_5.in, FourJoin_0.in3;

FPMul64_2.out -> FPMul64_3.in1;
(2.0D) -> FPMul64_3.in2;
FPMul64_3.out -> FPAdd64_2.in1;
FPAdd64_2.out -> Register_6.in, FourJoin_0.in4;

FPAdd64_3.out -> Cmp_1.in_lhs;
Selector_1.out -> FourJoin_0.in1, FourJoin_0.in2;
FourJoin_0.out -> Fetch_0.in_cbd;
} InnerLoop_compute;

dependent codebag InnerLoop_next_itr 0 {
"OOB" -> Memory_0.in_wrAddr(1);
Memory_0.out_wrComp -> BitBucket.in;
Register_7.out -> Memory_0.in_wrData, Stride_0.in_next;
} InnerLoop_next_itr;
```

```

// this bag gets loaded if iter == iter_max
//   causing inner loop to terminate execution
// i, j, X, X, window.width,
dependent codebag InnerLoop_cleanup 0 {
"OOB" -> FPAdd64_1.in1;
FPAdd64_1.out -> FPAdd64_2.in1;
Register_7.out -> BitBucket.in;
Register_4.out -> IntMul64_0.in1;
Register_0.out -> IntMul64_0.in2;
IntMul64_0.out -> IntMul64_2.in2;
IntMul64_2.out -> IntAdd64_0.in1;
(16) -> IntMul64_1.in1, IntMul64_2.in1;
Register_1.out -> IntMul64_1.in2;
IntMul64_1.out -> IntAdd64_0.in2;
IntAdd64_0.out -> Memory_0.in_wrAddr(1);
Register_5.out -> FourJoin_0.in1;
Register_6.out -> FourJoin_0.in2;
FPAdd64_2.out -> FourJoin_0.in3;
Memory_0.out_wrComp -> FourJoin_0.in4;

FourJoin_0.out -> Cmp_1.in_rst;
Selector_1.out -> Cmp_0.in_rst;
} InnerLoop_cleanup;

// this bag gets loaded if z_r^2 + z_i^2 > bound
//   causing inner loop to terminate execution
dependent codebag InnerLoop_cleanup2 0 {
"OOB" -> Memory_0.in_wrAddr(4), FourJoin_0.in4;
Memory_0.out_wrComp -> FourJoin_0.in1, FourJoin_0.in2;
Register_7.out -> Memory_0.in_wrData, FourJoin_0.in3;
FourJoin_0.out -> Stride_0.in_next;
} InnerLoop_cleanup2;
```

C.2 Mandelbrot, LLVM-generated

Note that some small hand-coded changes were made in the following code to address the limitations of the compiler back-end. The mandatory aliases as shown in the previous listing are omitted here for brevity. The corresponding LLVM IR is shown at the top of each code bag. The comments have been generated automatically by the compiler and are not intended to draw specific attention to the reader.

```
;; initial codebag - this is hardcoded and assumes your program has a
;;  'main' function
initial codebag __main 0 {
  (dispatch) -> Fetch_0.in_cbd;
} __main;

dependent codebag dispatch 0 {
  (exit) -> Fetch_0.in_lr;
  (main) -> Toggle_0.in1;
  "OOB" -> Toggle_0.in2;
  Toggle_0.out -> Fetch_0.in_cbd;
  (1048576) -> Fetch_0.in_sp, Fetch_0.in_r2;
  (1048576) -> Fetch_0.in_r0, Fetch_0.in_r1;
  Toggle_0.out -> Fetch_0.in_cbd;
} dispatch;

independent codebag exit 0(r6) {
  "OOB" -> Join_0.in_pass;
  Register_6.out -> Join_0.in2;
  Join_0.out -> Fetch_0.in_cbd;
} exit;


;; Function 'my_malloc'
;;   prototype: void (i8 * *, i32)
;;   predecessors:  N/A
;;   successors:    N/A
;; ======================================
;; StackView (Function 'my_malloc')
;; ---------------------------------------
;; nothing
;; ======================================
independent codebag my_malloc 0 (r0, r1, r6, r7) {
  ;;  %_incoming_sp_ = load i64* @Register6.out ; <i64> [#uses=1]
  ;;  %_lr_ = load i64* @Register7.out ; <i64> [#uses=1]
  ;;  store i64 %_lr_, i64* @Fetch.in_cbd
  ;;  store i8* null, i8** %_ret_val_storage_ptr_
  ;;  %_outgoing_sp_ = bitcast i64 %_incoming_sp_ to i64 ; <i64> [#uses=1]
  ;;  store i64 %_outgoing_sp_, i64* @Fetch.in7
  ;;  ret void

  "OOB" -> Toggle_0.in2;
  Register_7.out -> Join_0.in_pass(0);                  // cleanup:  fetch next bag when cleanup complete
  Register_0.out -> Memory_2.in_wrAddr(64_BIT);         // store:  ... -> [_ret_val_storage_ptr_]
  (0) -> Memory_2.in_wrData(0);                         // store:  <<>> -> ...
  Register_6.out -> Fetch_0.in_sp(0);                   // store:  fetch_sp
  (2) -> Counter_0.in_cnt(0);                           // cleanup:  num stray tokens to wait for
  Counter_0.out -> Join_0.in2(0);                       // cleanup:  trigger bag fetch when all stray tokens arrive
  Memory_2.out_wrComp -> Counter_0.in_tok(0);           // cleanup:  remove this stray token
  Register_1.out -> Counter_0.in_tok(0);                // cleanup:  remove this stray token
  Join_0.out -> Toggle_0.in1;
  Toggle_0.out -> Fetch_0.in_cbd(0);                    // cleanup:  wait until done before fetching next bag
  Toggle_0.out -> Fetch_0.in_cbd(0);
} my_malloc;


;; Function ‘my_free’
;;   prototype: void (i8 *)
;;   predecessors:  N/A
;;   successors:    N/A
;; ======================================
;; StackView (Function ‘my_free’)
;; --------------------------------------
;; nothing
;; ======================================
independent codebag my_free 0 (r0, r6, r7) {
  ;;  %_incoming_sp_ = load i64* @Register6.out ; <i64> [#uses=1]
  ;;  %_lr_ = load i64* @Register7.out ; <i64> [#uses=1]
  ;;  store i64 %_lr_, i64* @Fetch.in_cbd
  ;;  %_outgoing_sp_ = bitcast i64 %_incoming_sp_ to i64 ; <i64> [#uses=1]
  ;;  store i64 %_outgoing_sp_, i64* @Fetch.in7
  ;;  ret void

  "OOB" -> Toggle_0.in2;
  Register_7.out -> Join_0.in_pass(0); // cleanup:  fetch next bag when cleanup complete
  Register_6.out -> Fetch_0.in_sp(0);  // store:  fetch_sp
  Join_0.out -> Toggle_0.in1;          // cleanup:  wait until done before fetching next bag
  Register_0.out -> Join_0.in2(0);     // cleanup:  remove this stray token
  Toggle_0.out -> Fetch_0.in_cbd(0);
  Toggle_0.out -> Fetch_0.in_cbd(0);
} my_free;


;; Function ‘main’
;;   prototype: void (i32 *, i32, i8 * *)
;;   predecessors:  N/A
;;   successors:    main_bb_entry_pt_veneer
;; ======================================
;; StackView (Function ‘main’)
;; --------------------------------------
;; Offset: -8, Name: _lr_storage, Type: i64 *
;; Offset: -16, Name: my_malloc_retVal, Type: i8 * *
;; Offset: -24, Name: stacked__ret_val_storage_ptr_, Type: i32 * *
;; ======================================
independent codebag main 0 (r0, r1, r2, r6, r7) {
  ;;  %_lr_storage_addr = add i64 %_incoming_sp_, -8 ; <i64> [#uses=1]
  ;;  %_lr_storage_addr_as_ptr = inttoptr i64 %_lr_storage_addr to i64* ; <i64*> [#uses=1]
  ;;  %my_malloc_retVal_addr = add i64 %_incoming_sp_, -16 ; <i64> [#uses=1]
  ;;  %my_malloc_retVal_addr_as_ptr = inttoptr i64 %my_malloc_retVal_addr to i8** ; <i8**> [#uses=1]
  ;;  %stacked__ret_val_storage_ptr__addr = add i64 %_incoming_sp_, -24 ; <i64> [#uses=1]
  ;;  %stacked__ret_val_storage_ptr__addr_as_ptr = inttoptr i64 %stacked__ret_val_storage_ptr__addr to i32**
  ;;          <i32**> [#uses=1]
  ;;  %_incoming_sp_ = load i64* @Register6.out ; <i64> [#uses=3]
  ;;  store i32* %_ret_val_storage_ptr_, i32** %stacked__ret_val_storage_ptr__addr_as_ptr
  ;;  %_incoming_lr_ = load i64* @Register7.out ; <i64> [#uses=1]
  ;;  store i64 %_incoming_lr_, i64* %_lr_storage_addr_as_ptr
  ;;  %tmp1 = tail call void @my_malloc( i8** %my_malloc_retVal_addr_as_ptr, i32 1228800 )
  ;;  br label %main_bb_entry_pt_veneer

  "OOB" -> Toggle_0.in2;
  (main_bb_entry_pt_veneer) -> Fetch_0.in_lr(0);
  (my_malloc) -> Join_0.in_pass(0);              // cleanup:  fetch next bag when cleanup complete
  (1228800) -> Fetch_0.in_r1(0);
  (-8) -> IntAdd64_5.in_rhs(0);
  (-16) -> IntAdd64_1.in_rhs(0);
  (-24) -> IntAdd64_3.in_rhs(0), IntAdd64_2.in_rhs(0);
  IntAdd64_3.out -> Memory_0.in_wrAddr(64_BIT);   // store:  ... -> [stacked__ret_val_storage_ptr__addr_as_ptr]
  Register_0.out -> Memory_0.in_wrData(0);        // store:  <<_ret_val_storage_ptr_>> -> ...
  IntAdd64_5.out -> Memory_1.in_wrAddr(64_BIT);   // store:  ... -> [_lr_storage_addr_as_ptr]
  Register_7.out -> Memory_1.in_wrData(0);        // store:  <<_incoming_lr_>> -> ...
  IntAdd64_1.out -> Fetch_0.in_r0(0);
  Register_6.out -> IntAdd64_2.in_lhs(0), Register_3.in;
  Register_3.out -> IntAdd64_5.in_lhs(0), Register_4.in;
  Register_4.out -> IntAdd64_1.in_lhs(0), IntAdd64_3.in_lhs(0);
  IntAdd64_2.out -> Fetch_0.in_sp(0);
  Join_0.out -> Toggle_0.in1;
  Toggle_0.out -> Fetch_0.in_cbd;
  Toggle_0.out -> Fetch_0.in_cbd;
  (4) -> Counter_0.in_cnt(0);          // cleanup:  num stray tokens to wait for
  Counter_0.out -> Join_0.in2(0);      // cleanup:  trigger bag fetch when all stray tokens arrive
  Memory_0.out_wrComp -> Counter_0.in_tok(0);    // cleanup:  remove this stray token
  Memory_1.out_wrComp -> Counter_0.in_tok(0);    // cleanup:  remove this stray token
  Register_1.out -> Counter_0.in_tok(0);         // cleanup:  remove this stray token
  Register_2.out -> Counter_0.in_tok(0);         // cleanup:  remove this stray token
} main;


;;   predecessors:  main
;;   successors:    main_bb
independent codebag main_bb_entry_pt_veneer 0 (r6) {
  ;;  %_incoming_sp_1 = load i64* @Register6.out ; <i64> [#uses=1]
  ;;  %_callee_adjusted_sp_ = add i64 %_incoming_sp_1, 24 ; <i64> [#uses=0]
  ;;  br label %main_bb

  (main_bb) -> Fetch_0.in_cbd(0);
  Register_6.out -> IntAdd64_1.in_lhs(0);
  (24) -> IntAdd64_1.in_rhs(0);
  (0) -> Register_0.in(0);                              // phi:  setup value for successor bb ‘main_bb’
} main_bb_entry_pt_veneer;

;;   predecessors:  main_bb_entry_pt_veneer   main_bb77
;;   successors:    main_bb15
dependent codebag main_bb 0 {
  ;;  %i.0.reg2mem.0 = phi i32 [ 0, %main_bb_entry_pt_veneer ], [ %indvar.next109, %main_bb77 ] ; <i32> [#uses=3]
  ;;  %tmp910 = sitofp i32 %i.0.reg2mem.0 to double ; <double> [#uses=1]
  ;;  %tmp13 = mul double %tmp910, 6.250000e-03 ; <double> [#uses=1]
  ;;  %tmp14 = sub double %tmp13, 1.500000e+00 ; <double> [#uses=1]
  ;;  %tmp59 = mul i32 %i.0.reg2mem.0, 640 ; <i32> [#uses=1]
  ;;  br label %main_bb15

  (main_bb15) -> Fetch_0.in_cbd(0);
  (0.5000D) -> FPMul64_0.in_rhs(0);
  (1.50000D) -> FPAdd64_0.in_rhs(1);
  Register_0.out *-> IntToFP_0.in(0), IntMul64_0.in_lhs(0);   // sitofp:  (float)i.0.reg2mem.0
  IntToFP_0.out -> FPMul64_0.in_lhs(0);
  FPMul64_0.out -> FPAdd64_0.in_lhs(0);
  (8) -> IntMul64_0.in_rhs(0);
  (0) -> Register_1.in(0);                              // phi:  setup value for successor bb ‘main_bb15’
} main_bb;

;;   predecessors:  main_bb   main_bb57__to__main_bb15__cleanup
;;   successors:    main_bb22
dependent codebag main_bb15 0 {
  ;;  %j.0.reg2mem.0 = phi i32 [ 0, %main_bb ], [ %indvar.next108, %main_bb52__to__main_bb57__cleanup ] ; <i32> [#uses=3]
  ;;  %tmp1617 = sitofp i32 %j.0.reg2mem.0 to double ; <double> [#uses=1]
  ;;  %tmp19 = mul double %tmp1617, 6.250000e-03 ; <double> [#uses=1]
  ;;  %tmp20 = sub double %tmp19, 2.500000e+00 ; <double> [#uses=1]
  ;;  br label %main_bb22

  (main_bb22) -> Join_0.in_pass;
  (0.00000D) -> Register_4.in(0), Register_5.in(0);      // phi:  setup value for successor bb ‘main_bb22’
  (0.5000D) -> FPMul64_0.in_rhs(0);
  (2.50000D) -> FPAdd64_1.in_rhs(1);
  Join_0.out -> Fetch_0.in_cbd(0);
  Register_1.out *-> IntToFP_0.in(0);                   // sitofp:  (float)j.0.reg2mem.0
  IntToFP_0.out -> FPMul64_0.in_lhs(0);
  FPMul64_0.out -> FPAdd64_1.in_lhs(0), Join_0.in2;
  (0) -> Register_3.in(0);                              // phi:  setup value for successor bb ‘main_bb22’
} main_bb15;

;;   predecessors:  main_bb15   main_bb52__to__main_bb22__cleanup
;;   successors:    main_bb22__to__main_bb57__cleanup   main_bb22__to__main_bb52__cleanup
dependent codebag main_bb22 0 {
  ;;  %iter.0.reg2mem.0 = phi i32 [ 0, %main_bb15 ], [ %tmp51, %main_bb52 ] ; <i32> [#uses=2]
  ;;  %z_i.0.reg2mem.0 = phi double [ 0.000000e+00, %main_bb15 ], [ %tmp38, %main_bb52 ] ; <double> [#uses=3]
  ;;  %z_r.0.reg2mem.0 = phi double [ 0.000000e+00, %main_bb15 ], [ %tmp31, %main_bb52 ] ; <double> [#uses=3]
  ;;  %tmp25 = mul double %z_r.0.reg2mem.0, %z_r.0.reg2mem.0 ; <double> [#uses=1]
  ;;  %tmp28 = mul double %z_i.0.reg2mem.0, %z_i.0.reg2mem.0 ; <double> [#uses=1]
  ;;  %tmp29 = sub double %tmp25, %tmp28 ; <double> [#uses=1]
  ;;  %tmp31 = add double %tmp29, %tmp20 ; <double> [#uses=3]
  ;;  %tmp33 = mul double %z_r.0.reg2mem.0, 2.000000e+00 ; <double> [#uses=1]
  ;;  %tmp36 = mul double %tmp33, %z_i.0.reg2mem.0 ; <double> [#uses=1]
  ;;  %tmp38 = add double %tmp36, %tmp14 ; <double> [#uses=3]
  ;;  %tmp42 = mul double %tmp31, %tmp31 ; <double> [#uses=1]
  ;;  %tmp45 = mul double %tmp38, %tmp38 ; <double> [#uses=1]
  ;;  %tmp46 = add double %tmp42, %tmp45 ; <double> [#uses=1]
  ;;  %tmp47 = fcmp ogt double %tmp46, 1.000000e+05 ; <i1> [#uses=1]
  ;;  br i1 %tmp47, label %main_bb22__to__main_bb57__cleanup, label %main_bb22__to__main_bb52__cleanup

  (main_bb22__to__main_bb57__cleanup) -> Selector_0.in_true(0);
  (main_bb22__to__main_bb52__cleanup) -> Selector_0.in_false(0);
  (2.00000D) -> FPMul64_2.in_rhs(0);
  (100000.0D) -> Cmp_0.in_rhs(0);
  Register_5.out *-> FPMul64_0.in_lhs(0), FPMul64_0.in_rhs(0);
  Register_5.out *-> FPMul64_2.in_lhs(0);
  Register_4.out *-> FPMul64_1.in_lhs(0), FPMul64_1.in_rhs(0);
  Register_4.out *-> FPMul64_3.in_rhs(0);
  FPMul64_0.out -> FPAdd64_4.in_lhs(0);
  FPMul64_1.out -> FPAdd64_4.in_rhs(1);
  FPAdd64_4.out -> FPAdd64_2.in_lhs(0);
  FPAdd64_1.out *-> FPAdd64_2.in_rhs(0);
  FPMul64_2.out -> FPMul64_3.in_lhs(0);
  FPMul64_3.out -> FPAdd64_3.in_lhs(0);
  FPAdd64_0.out *-> FPAdd64_3.in_rhs(0);
  FPAdd64_2.out *-> FPMul64_4.in_lhs(0), FPMul64_4.in_rhs(0);
  FPAdd64_3.out *-> FPMul64_5.in_lhs(0), FPMul64_5.in_rhs(0);
  FPMul64_4.out -> FPAdd64_5.in_lhs(0);
  FPMul64_5.out -> FPAdd64_5.in_rhs(0);
  FPAdd64_5.out -> Cmp_0.in_lhs(0);
  (FP_GT) -> Cmp_0.in_op(0);
  Cmp_0.out -> Selector_0.in_sel(0);
  Selector_0.out -> Fetch_0.in_cbd(0);
  Register_3.out *-> Register_2.in(0);                   // phi:  setup value for successor bb ‘main_bb57’
} main_bb22;

;;   predecessors:  main_bb22
;;   successors:    main_bb57
dependent codebag main_bb22__to__main_bb57__cleanup 0 {
  ;;  br label %main_bb57

  (main_bb57) -> Join_0.in_pass(0);                      // cleanup:  fetch next bag when cleanup complete
  Join_0.out -> Fetch_0.in_cbd(0);                       // cleanup:  wait until done before fetching next bag
  (6) -> Counter_0.in_cnt(0);                            // cleanup:  num stray tokens to wait for
  Counter_0.out -> Join_0.in2(0);                        // cleanup:  trigger bag fetch when all stray tokens arrive
  FPAdd64_1.out -> Counter_0.in_tok(0);                  // cleanup:  remove this stray token
  Register_3.out -> Counter_0.in_tok(0);                 // cleanup:  remove this stray token
  Register_4.out -> Counter_0.in_tok(0);                 // cleanup:  remove this stray token
  Register_5.out -> Counter_0.in_tok(0);                 // cleanup:  remove this stray token
  FPAdd64_2.out -> Counter_0.in_tok(0);                  // cleanup:  remove this stray token
  FPAdd64_3.out -> Counter_0.in_tok(0);                  // cleanup:  remove this stray token
} main_bb22__to__main_bb57__cleanup;
```

```

;;   predecessors:  main_bb22__to__main_bb57__cleanup   main_bb52__to__main_bb57__cleanup
;;   successors:    main_bb57__to__main_bb77__cleanup   main_bb57__to__main_bb15__cleanup
dependent codebag main_bb57 0 {
  ;;  %iter.0.reg2mem.1 = phi i32 [ %iter.0.reg2mem.0, %main_bb52__to__main_bb22__cleanup ],
  ;;          [ %tmp51, %main_bb52 ] ; <i32> [#uses=1]
  ;;  %tmp61 = add i32 %j.0.reg2mem.0, %tmp59 ; <i32> [#uses=1]
  ;;  %tmp6164 = sext i32 %tmp61 to i64 ; <i64> [#uses=1]
  ;;  %tmp65 = getelementptr i32* null, i64 %tmp6164 ; <i32*> [#uses=1]
  ;;  store i32 %iter.0.reg2mem.1, i32* %tmp65, align 4
  ;;  %indvar.next108 = add i32 %j.0.reg2mem.0, 1 ; <i32> [#uses=2]
  ;;  %exitcond = icmp eq i32 %indvar.next108, 640 ; <i1> [#uses=1]
  ;;  br i1 %exitcond, label %main_bb57__to__main_bb77__cleanup, label %main_bb57__to__main_bb15__cleanup

  (main_bb57__to__main_bb77__cleanup) -> Selector_0.in_true(0);
  (main_bb57__to__main_bb15__cleanup) -> Selector_0.in_false(0);
  Register_1.out -> IntAdd64_0.in_lhs(0), IntAdd64_3.in_lhs(0);
  IntMul64_0.out *-> IntAdd64_0.in_rhs(0);
  IntAdd64_0.out -> SExt_0.in(0);                       // sext:  (int64_t)tmp61
  (0) -> IntAdd64_2.in_lhs(0), Cmp_0.in_op(0);          // getelemptr
  (2) -> IntMul64_1.in_lhs(0);                          // getelemptr
  IntMul64_1.out -> IntAdd64_2.in_rhs(0);               // getelemptr
  SExt_0.out -> IntMul64_1.in_rhs(0);                   // getelemptr
  IntAdd64_2.out -> Memory_0.in_wrAddr(16_BIT);         // store:  ... -> [tmp65]
  Register_2.out -> Memory_0.in_wrData(0);              // store:  <<iter.0.reg2mem.1>> -> ...
  (1) -> IntAdd64_3.in_rhs(0);
  IntAdd64_3.out -> Cmp_0.in_lhs(0), Register_1.in(0);
  (8) -> Cmp_0.in_rhs(0);
  Cmp_0.out -> Selector_0.in_sel(0);
  Selector_0.out -> Fetch_0.in_cbd(0);
} main_bb57;

;;   predecessors:  main_bb57
;;   successors:    main_bb77
dependent codebag main_bb57__to__main_bb77__cleanup 0 {
  ;;  br label %main_bb77

  (main_bb77) -> Join_0.in_pass(0);                     // cleanup:  fetch next bag when cleanup complete
  Join_0.out -> Fetch_0.in_cbd(0);                      // cleanup:  wait until done before fetching next bag
  (4) -> Counter_0.in_cnt(0);                           // cleanup:  num stray tokens to wait for
  Counter_0.out -> Join_0.in2(0);                       // cleanup:  trigger bag fetch when all stray tokens arrive
  FPAdd64_0.out -> Counter_0.in_tok(0);                 // cleanup:  remove this stray token
  IntMul64_0.out -> Counter_0.in_tok(0);                // cleanup:  remove this stray token
  Memory_0.out_wrComp -> Counter_0.in_tok(0);           // cleanup:  remove this stray token
  Register_1.out -> Counter_0.in_tok(0);                // cleanup:  remove this stray token
} main_bb57__to__main_bb77__cleanup;

;;   predecessors:  main_bb57__to__main_bb77__cleanup
;;   successors:    main_bb77__to__main_bb85__cleanup   main_bb
dependent codebag main_bb77 0 {
  ;;  %indvar.next109 = add i32 %i.0.reg2mem.0, 1 ; <i32> [#uses=2]
  ;;  %exitcond110 = icmp eq i32 %indvar.next109, 480 ; <i1> [#uses=1]
  ;;  br i1 %exitcond110, label %main_bb77__to__main_bb85__cleanup, label %main_bb

  (main_bb77__to__main_bb85__cleanup) -> Selector_0.in_true(0);
  (main_bb) -> Selector_0.in_false(0);
  Register_0.out -> IntAdd64_0.in_lhs(0);
  (1) -> IntAdd64_0.in_rhs(0);
  IntAdd64_0.out -> Cmp_0.in_lhs(0), Register_0.in(0);
  (6) -> Cmp_0.in_rhs(0);
  (INT_EQ) -> Cmp_0.in_op(0);
  Cmp_0.out -> Selector_0.in_sel(0);
  Selector_0.out -> Fetch_0.in_cbd(0);
} main_bb77;

;;   predecessors:  main_bb77
;;   successors:    main_bb85
dependent codebag main_bb77__to__main_bb85__cleanup 0 {
  ;;  br label %main_bb85

  (main_bb85) -> Join_0.in_pass(0);                     // cleanup:  fetch next bag when cleanup complete
  Join_0.out -> Fetch_0.in_cbd(0);                      // cleanup:  wait until done before fetching next bag
  Register_0.out -> Join_0.in2(0);                       // cleanup:  remove this stray token
} main_bb77__to__main_bb85__cleanup;

;;   predecessors:  main_bb77__to__main_bb85__cleanup
;;   successors:    main_bb85_cr
dependent codebag main_bb85 0 {
  ;;  tail call void @my_free( i8* null ) nounwind
  ;;  br label %main_bb85_cr

  (main_bb85_cr) -> Fetch_0.in_lr(0);
  (my_free) -> Fetch_0.in_cbd(0);
  (0) -> Fetch_0.in_r0(0);
  IntAdd64_1.out -> IntAdd64_0.in_lhs(0);
  (-24) -> IntAdd64_0.in_rhs(0);
  IntAdd64_0.out -> Fetch_0.in_sp(0);
} main_bb85;

;;   predecessors:  main_bb57
;;   successors:    main_bb15
dependent codebag main_bb57__to__main_bb15__cleanup 0 {
  ;;  br label %main_bb15

  (main_bb15) -> Join_0.in_pass(0);                      // cleanup:  fetch next bag when cleanup complete
  Join_0.out -> Fetch_0.in_cbd(0);                       // cleanup:  wait until done before fetching next bag
  Memory_0.out_wrComp -> Join_0.in2(0);                  // cleanup:  remove this stray token
} main_bb57__to__main_bb15__cleanup;

;;   predecessors:  main_bb22
;;   successors:    main_bb52
dependent codebag main_bb22__to__main_bb52__cleanup 0 {
  ;;  br label %main_bb52

  (main_bb52) -> Join_0.in_pass(0);                      // cleanup:  fetch next bag when cleanup complete
  Join_0.out -> Fetch_0.in_cbd(0);                       // cleanup:  wait until done before fetching next bag
  (3) -> Counter_0.in_cnt;
  Counter_0.out -> Join_0.in2(0);                        // cleanup:  remove this stray token
  Register_2.out -> Counter_0.in_tok(0);                 // cleanup:  remove this stray token
  Register_4.out -> Counter_0.in_tok(0);                 // cleanup:  remove this stray token
  Register_5.out -> Counter_0.in_tok(0);                 // cleanup:  remove this stray token
} main_bb22__to__main_bb52__cleanup;

;;   predecessors:  main_bb22__to__main_bb52__cleanup
;;   successors:    main_bb52__to__main_bb22__cleanup   main_bb52__to__main_bb57__cleanup
dependent codebag main_bb52 0 {
  ;;  %tmp51 = add i32 %iter.0.reg2mem.0, 1 ; <i32> [#uses=3]
  ;;  %tmp54 = icmp slt i32 %tmp51, 1000 ; <i1> [#uses=1]
  ;;  br i1 %tmp54, label %main_bb52__to__main_bb22__cleanup, label %main_bb52__to__main_bb57__cleanup

  (main_bb52__to__main_bb22__cleanup) -> Selector_0.in_true(0);
  (main_bb52__to__main_bb57__cleanup) -> Selector_0.in_false(0);
  Register_3.out -> IntAdd64_0.in_lhs(0);
  (1) -> IntAdd64_0.in_rhs(0);
  IntAdd64_0.out *-> Cmp_0.in_lhs(0), Register_3.in(0);
  IntAdd64_0.out *-> Register_2.in(0), Join_0.in2;
  (100) -> Cmp_0.in_rhs(0);
  (INT_SLT) -> Cmp_0.in_op(0);
  Cmp_0.out -> Selector_0.in_sel(0);
  Selector_0.out -> Join_0.in_pass;
  Join_0.out -> Fetch_0.in_cbd(0);
  FPAdd64_3.out -> Register_4.in(0);                     // phi:  setup value for successor bb ‘main_bb22’
  FPAdd64_2.out -> Register_5.in(0);                     // phi:  setup value for successor bb ‘main_bb22’
} main_bb52;

;;   predecessors:  main_bb52
;;   successors:    main_bb22
dependent codebag main_bb52__to__main_bb22__cleanup 0 {
  ;;  br label %main_bb22

  (main_bb22) -> Join_0.in_pass(0);                      // cleanup:  fetch next bag when cleanup complete
  Join_1.out -> Fetch_0.in_cbd(0);                       // cleanup:  wait until done before fetching next bag
  Register_2.out -> Join_0.in2(0);                       // cleanup:  remove this stray token
  IntAdd64_0.out -> Join_1.in2;
  Join_0.out -> Join_1.in_pass;
} main_bb52__to__main_bb22__cleanup;

;;   predecessors:  main_bb52
;;   successors:    main_bb57
dependent codebag main_bb52__to__main_bb57__cleanup 0 {
  ;;  br label %main_bb57

  (main_bb57) -> Join_0.in_pass(0);                     // cleanup:  fetch next bag when cleanup complete
  Join_0.out -> Fetch_0.in_cbd(0);                      // cleanup:  wait until done before fetching next bag
  (5) -> Counter_0.in_cnt(0);                           // cleanup:  num stray tokens to wait for
  Counter_0.out -> Join_0.in2(0);                       // cleanup:  trigger bag fetch when all stray tokens arrive
  FPAdd64_1.out -> Counter_0.in_tok(0);                 // cleanup:  remove this stray token
  Register_3.out -> Counter_0.in_tok(0);                // cleanup:  remove this stray token
  Register_4.out -> Counter_0.in_tok(0);                // cleanup:  remove this stray token
  Register_5.out -> Counter_0.in_tok(0);                // cleanup:  remove this stray token
  IntAdd64_0.out -> Counter_0.in_tok(0);
} main_bb52__to__main_bb57__cleanup;


;;   predecessors:  main_bb85
;;   successors:    N/A
independent codebag main_bb85_cr 0 (r6) {
  ;;  %_lr_storage_addr5 = add i64 %_callee_adjusted_sp_3, -8 ; <i64> [#uses=1]
  ;;  %_lr_storage_addr5_as_ptr = inttoptr i64 %_lr_storage_addr5 to i64* ; <i64*> [#uses=1]
  ;;  %stacked__ret_val_storage_ptr__addr4 = add i64 %_callee_adjusted_sp_3, -24 ; <i64> [#uses=1]
  ;;  %stacked__ret_val_storage_ptr__addr4_as_ptr = inttoptr i64 %stacked__ret_val_storage_ptr__addr4 to i32**
  ;;          <i32**> [#uses=1]
  ;;  %_incoming_sp_2 = load i64* @Register6.out ; <i64> [#uses=1]
  ;;  %_callee_adjusted_sp_3 = add i64 %_incoming_sp_2, 24 ; <i64> [#uses=3]
  ;;  %_ret_val_storage_ptr__val = load i32** %stacked__ret_val_storage_ptr__addr4_as_ptr ; <i32*> [#uses=1]
  ;;  %_lr_ = load i64* %_lr_storage_addr5_as_ptr ; <i64> [#uses=1]
  ;;  store i64 %_lr_, i64* @Fetch.in_cbd
  ;;  store i32 0, i32* %_ret_val_storage_ptr__val
  ;;  %_outgoing_sp_ = bitcast i64 %_callee_adjusted_sp_3 to i64 ; <i64> [#uses=1]
  ;;  store i64 %_outgoing_sp_, i64* @Fetch.in7
  ;;  ret void

  "OOB" -> Toggle_0.in2;
  (-8) -> IntAdd64_0.in_rhs(0);
  (-24) -> IntAdd64_1.in_rhs(0);
  Register_6.out -> IntAdd64_2.in_lhs(0);
  (24) -> IntAdd64_2.in_rhs(0);
  IntAdd64_1.out -> Memory_0.in_rdAddr(64_BIT);         // load:  [stacked__ret_val_storage_ptr__addr4_as_ptr]
  IntAdd64_0.out -> Memory_1.in_rdAddr(64_BIT);         // load:  [_lr_storage_addr5_as_ptr]
  Memory_1.out_rdData -> Join_0.in_pass(0);             // cleanup:  fetch next bag when cleanup complete
  Memory_0.out_rdData -> Memory_2.in_wrAddr(32_BIT);    // store:  ... -> [_ret_val_storage_ptr__val]
  (0) -> Memory_2.in_wrData(0);                         // store:  <<>> -> ...
  IntAdd64_2.out -> Fetch_0.in_sp(0), Register_0.in;
  Register_0.out -> IntAdd64_0.in_lhs(0), IntAdd64_1.in_lhs(0);     // store:  fetch_sp
  Join_0.out -> Toggle_0.in1;
  Toggle_0.out -> Fetch_0.in_cbd;
  Toggle_0.out -> Fetch_0.in_cbd;
  Memory_2.out_wrComp -> Join_0.in2(0);                 // cleanup:  remove this stray token
} main_bb85_cr;
```

Appendix D

LLVM instruction mapping to Armada

This appendix gives a general description of how the LLVM intermediate representation instructions map to Armada Ships.

D.1 Terminator class

D.1.1 `ret`

Other passes transform the functions such that they all return `void`.

LLVM

```
ret void
```

Armada

```
lr -> Fetch.in_cbd
```

D.1.2 `br`

LLVM

```
br i1 <cond>, label <iftrue>, label <iffalse>
```

Armada

```
<cond> -> Selector.in_sel
<iftrue> -> Selector.in_true
<iffalse> -> Selector.in_false
Selector.out -> Fetch.in_cbd
```

LLVM

```
br label <dest>
```

Armada

```
<dest> -> Fetch.in_cbd
```

D.1.3 `switch`

Unimplemented.

D.1.4 `invoke`

Unimplemented.

D.1.5 `unwind`

Unimplemented.

D.1.6 `unreachable`

Unimplemented.

D.2 Binary class

D.2.1 `add`

LLVM

```
<result> = add <ty> <var1>, <var2> ; yields {ty}:result
```

Armada

if `<ty>` is an `int`:

```
<var1> -> IntAdd64.in_lhs
<var2> -> IntAdd64.in_rhs
; result on IntAdd64.out
```

else if `<ty>` is a `float`:

```
<var1> -> FPAdd64.in_lhs
<var2> -> FPAdd64.in_rhs
; result on FPAdd64.out
```

D.2.2 `sub`

LLVM

Armada

if `<ty>` is an `int`:

```
<var1> -> IntAdd64.in_lhs
<var2> -> IntAdd64.in_rhs(NEGATE)
; result on IntAdd64.out
```

else if `<ty>` is a `float`:

```
<var1> -> FPAdd64.in_lhs
```

```
<var2> -> FPAdd64.in_rhs(NEGATE)
; result on FPAdd64.out
```

D.2.3 `mul`

LLVM

Armada

if `<ty>` is an `int`:

```
<var1> -> IntMul64.in_lhs
<var2> -> IntMul64.in_rhs
; result on IntMul64.out
```

else if `<ty>` is a `float`:

```
<var1> -> FPMul64.in_lhs
<var2> -> FPMul64.in_rhs
; result on FPMul64.out
```

D.2.4 `udiv`

LLVM

```
<result> = udiv <ty> <var1>, <var2> ; yields {ty}:result
```

Armada

```
<var1> -> IntDiv64.in_lhs
<var2> -> IntDiv64.in_rhs
; must dispose of remainder token if not used on IntDiv64.outR ; result on IntDiv64.outQ
```

D.2.5 `sdiv`

LLVM

```
<result> = sdiv <ty> <var1>, <var2> ; yields {ty}:result
```

Armada

```
<var1> -> IntDiv64.in_lhs
<var2> -> IntDiv64.in_rhs
; must dispose of remainder token if not used on IntDiv64.outR ; result on IntDiv64.outQ
```

D.2.6 `fdiv`

LLVM

```
<result> = fdiv <ty> <var1>, <var2> ; yields {ty}:result
```

Armada

```
<var1> -> FPDiv64.in_lhs
<var2> -> FPDiv64.in_rhs
; result on FPDiv64.out
```

D.2.7 `urem`

LLVM

```
<result> = urem <ty> <var1>, <var2> ; yields {ty}:result
```

Armada

```
<var1> -> IntDiv64.in_lhs
<var2> -> IntDiv64.in_rhs
; must dispose of quotient token if not used on IntDiv64.outQ ; result on IntDiv64.outR
```

D.2.8 `srem`

LLVM

```
<result> = srem <ty> <var1>, <var2> ; yields {ty}:result
```

Armada

```
<var1> -> IntDiv64.in_lhs
<var2> -> IntDiv64.in_rhs
; must dispose of quotient token if not used on IntDiv64.outQ ; result on IntDiv64.outR
```

D.2.9 `frem`

Unimplemented.

D.3 Bitwise binary class

D.3.1 `shl`

LLVM

```
<result> = shl <ty> <var1>, <var2> ; yields {ty}:result
```

Armada

```
<var1> -> BitShift.in_lhs(32_BIT or 64_BIT)
<var2> -> BitShift.in_rhs(LSL)
; result on BitShift.out
```

D.3.2 `lshr`

LLVM

```
<result> = lshr <ty> <var1>, <var2> ; yields {ty}:result
```

Armada

```
<var1> -> BitShift.in_lhs(32_BIT or 64_BIT)
<var2> -> BitShift.in_rhs(LSR)
; result on BitShift.out
```

D.3.3 `ashr`

LLVM

```
<result> = ashr <ty> <var1>, <var2> ; yields {ty}:result
```

Armada

```
<var1> -> BitShift.in_lhs(32_BIT or 64_BIT)
<var2> -> BitShift.in_rhs(ASR)
; result on BitShift.out
```

D.3.4 `and`

LLVM

```
<result> = and <ty> <var1>, <var2> ; yields {ty}:result
```

Armada

```
<var1> -> BitOp.in_lhs
<var2> -> BitOp.in_rhs(AND)
; result on BitOp.out
```

D.3.5 `or`

LLVM

```
<result> = or <ty> <var1>, <var2> ; yields {ty}:result
```

Armada

```
<var1> -> BitOp.in_lhs
<var2> -> BitOp.in_rhs(OR)
; result on BitOp.out
```

D.3.6 `xor`

LLVM

```
<result> = xor <ty> <var1>, <var2> ; yields {ty}:result
```

Armada

```
<var1> -> BitOp.in_lhs
<var2> -> BitOp.in_rhs(XOR)
; result on BitOp.out
```

D.4 Vector class

D.4.1 `extractelement`

Unimplemented.

D.4.2 `insertelement`

Unimplemented.

D.4.3 `shufflevector`

Unimplemented.

D.5 Memory class

D.5.1 `malloc`

LLVM

```
<result> = malloc <type>[, i32 <NumElements>][, align <alignment>]
```

Armada

Treated as regular function call.

D.5.2 `free`

LLVM

```
free <type> <value> ; yields {void}
```

Armada

Treated as regular function call.

D.5.3 alloca

LLVM

```
<result> = alloca <type>[, i32 <NumElements>][, align <alignment>] ; yields {type*}:result
```

Armada

Handled by stack value placement compiler pass.

D.5.4 load

LLVM

```
<result> = load <ty>* <pointer>[, align <alignment>]
<result> = volatile load <ty>* <pointer>[, align <alignment>]
```

Armada

All loads are considered volatile in this implementation.

```
<pointer> -> Memory.in_rdAddr(sizeof(ty))
; result on Memory.out_rdData
```

D.5.5 store

LLVM

```
store <ty> <value>, <ty>* <pointer>[, align <alignment>] ; yields {void}
volatile store <ty> <value>, <ty>* <pointer>[, align <alignment>] ; yields {void}
```

Armada

All stores are treated as volatile in this implementation.

```
<pointer> -> Memory.in_wrAddr(sizeof(ty))
<value> -> Memory.in_wrData
; completion token on Memory.out_wrComp
```

D.5.6 getelementptr

LLVM

```
<result> = getelementptr <ty>* <ptrval>{, <ty> <idx>}*
```

Armada

```
<ptrval> -> IntAdd64.in_lhs
sizeof<ty> -> IntMul64.in_lhs
<idx> -> IntMul64.in_rhs
IntMul64.out -> IntAdd64.in_rhs
; result on IntAdd64.out
```

D.6 Conversion class

D.6.1 `trunc . . to`

LLVM

```
<result> = trunc <ty> <value> to <ty2> ; yields ty2
```

Armada

```
<value> -> BitOp.in_lhs
<2^{sizeof(<ty2>)} - 1> -> BitOp.in_rhs(AND)
; result on BitOp.out
```

D.6.2 `sext. . . to`

LLVM

```
<result> = sext <ty> <value> to <ty2> ; yields ty2
```

Armada

Current implementation always extends to 64 bits.

```
<value> -> SExt.in(8_BIT or 16_BIT or 32_BIT)
; result on SExt.out
```

D.6.3 `zext. . . to`

No-op in Armada.

D.6.4 `fptrunc. . . to`

As Armada only supports double-precision floating point, calls to fptrunc are considered no-ops since lower-precision floating point numbers are not supported.

D.6.5 `fpext. . . to`

As Armada only supports double-precision floating point, calls to fpext are considered no-ops since all floating point values are already double precision.

D.6.6 `fptoui. . . to`

Unimplemented.

D.6.7 `fptosi. . . to`

Unimplemented.

D.6.8 `uitofp. . . to`

LLVM

```
<result> = uitofp <ty> <value> to <ty2> ; yields ty2
```

Armada

```
<value> -> IntToFP.in
; output on IntToFP.out
```

D.6.9 `sitofp. . . to`

LLVM

```
<result> = sitofp <ty> <value> to <ty2> ; yields ty2
```

Armada

```
<value> -> IntToFP.in
; output on IntToFP.out
```

D.6.10 `ptrtoint. . . to`

No-op in Armada.

D.6.11 `inttoptr. . . to`

No-op in Armada.

D.6.12 `bitcast. . . to`

No-op in Armada.

D.7 Miscellaneous class

D.7.1 `icmp`

LLVM

```
<result> = icmp <cond> <ty> <var1>, <var2> ; yields {i1}:result
```

Armada

```
<var1> -> Cmp.in_lhs
<var2> -> Cmp.in_rhs
<one of INT CMP ops (see Cmp Ship)> -> Cmp.in_op
; result on Cmp.out
```

D.7.2 `fcmp`

LLVM

```
<result> = icmp <cond> <ty> <var1>, <var2> ; yields {i1}:result
```

Armada

```
<var1> -> Cmp.in_lhs
<var2> -> Cmp.in_rhs
<one of FP CMP ops (see Cmp Ship)> -> Cmp.in_op
; result on Cmp.out
```

D.7.3 `phi`

Handled in the Armada passes.

D.7.4 `select`

LLVM

```
<result> = select i1 <cond>, <ty> <val1>, <ty> <val2> ; yield
```

Armada

```
<val1> -> Selector.in_true
<val2> -> Selector.in_false
<cond> -> Selector.in_sel
; result on out Selector.out
```

D.7.5 `call`

Handled in the Armada passes.

D.7.6 `va_arg`

Unimplemented.

D.7.7 `getresult`

Unneeded for call's as they never return values in Armada. Unimplemented for invoke's because invoke's are unimplemented in this version of the compiler.