HASTE: Hybrid Architectures with a Single, Transformable Executable

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Benjamin A. Levine

Carnegie Mellon University
Pittsburgh, Pennsylvania
May, 2005

Abstract

Hybrid architectures, which are composed of a conventional processor closely coupled with reconfigurable logic, seem to combine the advantages of both types of hardware. They present some practical difficulties however. The interface between the processor and the reconfigurable logic is crucial to performance and is often difficult to implement well. Partitioning the application between the processor and logic is a difficult task, typically complicated by entirely different programming models, heterogeneous interfaces to external resources, and incompatible representations of applications. A separate executable must be produced and maintained for each type of hardware. An architecture called HASTE (Hybrid Architecture with a Single Transformable Executable) solves many of these difficulties. HASTE allows a single executable to represent an entire application, including portions that run on a reconfigurable fabric and portions that run on a sequential processor. This executable can execute in its entirety on the processor, but for best performance portions of the application that are mapped onto the fabric at run-time. Extensive experiments show that this concept is feasible for a range of different benchmarks, and that HASTE architectures can provide performance several times better than commercial FPGAs, while being easier to program and providing all of the advantages of having a single executable.

Acknowledgements

I would first like to thank my advisor, Professor Herman Schmit. This work could not have been completed without his support and guidance. I must also gratefully acknowledge the Semiconductor Research Corporation and IBM Corporation, who supported the author during this research with an SRC/IBM Graduate Research Fellowship. Finally, and most importantly, I would like to thank my wife Ruth, without whom I would never have made it as far I have. Her love, support, and patience have kept me going through many tough times and long nights of work.

Contents

1 Introduction **1**

2 Background and Related Work **4**

2.1 Silicon Complexity 4

2.2 System Complexity 6

2.3 Possible Solutions 7

2.3.1 Reuse 7

2.3.2 Reconfigurable Hardware for Hardware Platforms 8

2.4 Goals 11

2.5 Related Work 12

3 HASTE Systems and Applications **14**

3.1 Haste Overview 14

3.2 Application Model 17

3.2.1 Operations 17

3.2.2 Sequential model 18

3.2.3 Kernel Model 20

3.2.4 Complete Application Model 23

3.3 System Model 23

3.3.1 Component Completeness 26

3.4 Observations 27

4 Instruction Set Architectures **28**

4.1 Generic HASTE Assembly Language (GHAL) 28

4.1.1 Loop Delimiters 29

4.1.2 Streaming Memory Accessors 33

4.1.3 Select Instructions 34

4.2 ISA Requirements 35

4.2.1 Operations and Placement 35

4.2.2 Data Flow and Routing 38

4.3 Queue ISA (QISA) 38

4.4 Register ISA (RISA) 45

4.5 Relative Register ISA (RRISA) 49

4.6 Observations 51

5 RCF Architecture **52**

5.1 Global RCF Model 54

5.1.1 Global Parameters 54

5.2 Register File 55

5.3 Processing Elements 57

5.4 Interconnect 59

5.5 RCF Classes 60

5.5.1 Static Register Fabric 60

5.5.2 Asymmetric Pass Register Fabric 63

5.5.3 Symmetric Pass Register Fabric 63

5.6 Observations 65

6 CTE and SPU Operation **67**

6.1 Queue ISA SPU and CTE Operation 69

6.1.1 QISA SPU Operation for Example 69

6.1.2 QISA CTE Operation 72

6.1.3 QISA CTE Operation for Example 76

6.1.3.1 Table and Configuration Description 77

6.1.3.2 Processing of Example Instructions 78
6.2 Register ISA SPU and CTE Operation 82
6.2.1 RISA SPU Operation for Example 82
6.2.2 RISA CTE Operation 84
6.2.3 RISA CTE Operation for Example 88
6.2.3.1 Table and Configuration Description 91
6.2.3.2 Processing of Example Instructions 92
6.3 Relative Register ISA SPU and CTE Operation 97
6.3.1 RRISA SPU Operation for Example 97
6.3.2 RRISA CTE Operation 100
6.3.3 RRISA CTE Operation for Example 102
6.3.3.1 Table and Configuration Description 104
6.3.3.2 Processing of Example Instructions 107
6.4 Conclusions 108

7 Tool Flow and Simulation Environment **109**
7.1 Application Mapping 112
7.1.1 Kernel Annotation 112
7.1.2 Compilation 114
7.1.3 Conversion to GHAL 117
7.1.4 Conversion to DAG 119
7.1.5 Mapping to Specific HASTE Implementations 121
7.1.6 Assembly 122
7.2 Simulation and Validation Tool Flow 122
7.3 Hardware Implementation Tool Flow 131
7.3.1 ASIC and FPGA Implementations 131
7.3.2 HASTE Hardware Implementation 135
7.4 Summary 135

8 Comparison of HASTE ISAs **137**
8.1 ISA Metrics 137

8.1.1 Code Length . . . 138
8.1.2 Code Size . . . 139
8.1.3 Hardware Utilization . . . 140
8.1.4 Hardware Latency . . . 141
8.2 ISA-Specific Mapping Procedures . . . 141
8.2.1 Queue ISA . . . 142
8.2.1.1 Levelization, Planarization, and Compression . . . 143
8.2.1.2 Finding Node Locations . . . 144
8.2.1.3 Writing Out Assembly Code . . . 146
8.2.2 Register and Relative-Register ISAs . . . 148
8.2.2.1 Parameter Checking . . . 148
8.2.2.2 Finding Minimum Latency . . . 149
8.2.2.3 Finding Lower Bounds on Width for Given Latency . . . 149
8.2.2.4 Check Connectivity . . . 150
8.2.2.5 Register File Size Checking . . . 150
8.2.2.6 Writing Out Assembly Code . . . 151
8.3 Mapping Experiments and Results . . . 152
8.3.1 Queue ISA . . . 152
8.3.2 Register ISA and Relative Register ISA . . . 155
8.4 ISA Performance . . . 157
8.4.1 Code Length . . . 157
8.4.2 Code Size . . . 160
8.4.3 Hardware Utilization . . . 162
8.4.4 Latency . . . 162
8.5 Summary . . . 163

9 HASTE Kernel, Application, and Architecture Functionality 165

9.1 Benchmark Kernels . . . 166
9.1.1 Validation Results . . . 166
9.2 Large Application Implementation . . . 167
9.2.1 ATR Application Description . . . 167

9.3 Observations 172

10 Hardware Modeling and Synthesis **173**

10.1 ALU Design 174

10.2 Static Register Fabric 176

10.3 Asymmetric Pass Register Fabric 181

10.4 Symmetric Pass Register Fabric 181

10.5 HASTE Components 184

10.6 Observations 185

11 Area and Performance of Kernel Implementations **187**

11.1 ASIC Implementations 187

11.1.1 Procedure 189

11.1.2 Results 190

11.2 FPGA Implementation 193

11.2.1 Procedure 194

11.3 HASTE Implementations 197

11.4 Comparisons 200

11.4.1 Speed 200

11.4.2 Area 202

11.4.3 NTUA 202

11.4.4 Best Fabric 205

11.5 Observations 208

12 Conclusions and Future Work **209**

12.1 Conclusions 209

12.2 Future Work 210

12.2.1 Depth and Width Virtualization 210

12.2.2 Feedback 211

12.2.3 Narrow Tiles 211

A Glossary **212**

B HASTE ISA Reference **214**

C Benchmark Kernels **239**

D Example Testbench **258**

E DAG for Simple Example Kernel in GML Format **260**

F ATR Sourcecode **266**

Bibliography **274**

List of Tables

3.1 HASTE Application and Kernel Assumptions 24

4.1 New Instructions in GHAL 30

5.1 RCF Model Parameters 55
5.2 RCF Classes 60
5.3 Static Register Fabric Configuration Fields 62
5.4 Asymmetric Pass Register Fabric Configuration Fields 63
5.5 Symmetric Pass Register Fabric Configuration Fields 65

6.1 CTE Inputs from SPU and CTE Outputs to RCF for QISA 73
6.2 QISA CTE Example 79
6.3 CTE Inputs from SPU and CTE Outputs to RCF for RISA 87
6.4 RISA CTE Example - Issue Unit 93
6.5 RISA CTE Example - Config Stations 94
6.6 CTE Inputs from SPU and CTE Outputs to RCF for RISA 101
6.7 RRISA CTE Example 105

7.1 Comparison of Execution Statistics for Original (PISA) and GHAL Assembly Code 127

8.1 Queue ISA Mappings 153
8.2 Queue ISA Mapping Results 156
8.3 Register ISA Mapping Results 158
8.4 Relative Register ISA Mapping Results 159
8.5 Composite Results for HASTE ISAs 160

9.1 Kernel Benchmarks 167
9.2 Validation Runs 169

10.1 ALU Synthesis Results 176
10.2 Typical HASTE Component Areas, Min delay 185

11.1 ASIC Synthesis Run Types 189
11.2 Benchmark Synthesis Results - ASIC 191
11.3 FPGA Implementation Results 196
11.4 Comparison of Best Implementations for Each Benchmark 198
11.4 continued 199
11.4 continued 200

List of Figures

3.1 Basic HASTE Architecture and Operation 15
3.2 Application Model 19

4.1 Equivalent PISA, Instrumented PISA, and GHAL programs 31
4.2 Original C code for simple example program 32
4.3 Application Mapping in Compiler 36
4.4 Placement of Operations by CTE 37
4.5 Stack and Queue Machine Operation 39
4.6 Instruction Ordering for Stack and Queue Machines 40
4.7 Examples showing the level and planar properties of graphs 42
4.8 Queue ISA Instruction Variants 43
4.9 Select Instruction Variants 44
4.10 HASTE ISA Instruction Formats 46
4.11 RISA Register Addressing 47
4.12 Relative Register ISA Addressing. 50

5.1 RCF Model 53
5.2 Pass Register File 56
5.3 Processing Element 57
5.4 Interconnect Model 59
5.5 Static Register Fabric 61
5.6 Asymmetric Pass Register Fabric 62
5.7 Symmetric Pass Register Fabric 64

6.1 CTE and System Model 68
6.2 Example Loop Body 70
6.3 DFGs for CTE Examples 70
6.4 Queue Contents for QISA SPU Implementation 71
6.5 Fabric Configuration Key 73
6.6 Queue ISA CTE 74
6.7 QISA Fabric Configuration for Example 78
6.8 Register Contents for RISA SPU Implementation 83
6.9 RISA Fabric Configuration for Example 86
6.10 Register ISA CTE 88
6.11 Register Contents for RRISA SPU Implementation 100
6.12 Relative Register ISA CTE 102
6.13 RRISA Fabric Configuration for Example 106

7.1 Overall HASTE Tool Flow 110
7.2 Application Mapping Tool Flow 113
7.3 Unannotated C Code 115
7.4 Kernel C code with annotations 116
7.5 Assembly for Example Sequential Code 118
7.6 Assembly for Example Kernel 119
7.7 GHAL Assembly Code 120
7.8 DAG Generated from GHAL Code 121
7.9 Validation and Simulation Tool Flow 123
7.10 Steps in Validation Process 124
7.11 Results From Original Assembly 126
7.12 Application VHDL Generated from DAG 129
7.13 Testbench Structure 130
7.14 Hardware Implementation Tool Flow 132
7.15 Structural Model of Simple Example 134

8.1 Hardware Utilization 141

8.2 Examples showing the level and planar properties of graphs 142
8.3 Effect of level changes on Queue DAGs 145
8.4 Effect of limited read connectivity 147
8.5 Use of NOOPs to Space Instructions 147
8.6 Code Length 161
8.7 Code Size 161
8.8 Hardware Utilization 162
8.9 Hardware Latency 163

9.1 Steps in Validation Process 168
9.2 ATR Application Flow 171

10.1 Static Register Fabric Synthesis: No Multiplier 177
10.2 Static Register Fabric Synthesis: 32-Bit Multiplier 178
10.3 Static Register Fabric Synthesis: 64-Bit Multiplier 179
10.4 Asymmetric Pass Register Fabric Synthesis 182
10.5 Symmetric Pass Register Fabric Synthesis 183

11.1 Hardware Implementation Tool Flow 188
11.2 Frequency for Fastest Benchmark Implementations, by Technology. 201
11.3 Area for Smallest Benchmark Implementations, by Technology. 203
11.4 NTUA for Fastest Benchmark Implementations, by Technology. 204
11.5 NTUA Loss for Common Fabric Compared to Best Fabric 206
11.6 NTUA for Best Single Fabric Benchmark Implementations, by Technology. . . . 207

List of Algorithms

1 Queue ISA CTE Algorithm 75

2 RISA ISA Issue Unit Algorithm 89

3 RISA ISA Configuration Station Algorithm 90

4 RRISA CTE Algorithm 103

Chapter 1

Introduction

The last two decades have seen a staggering increase in the power and complexity of digital systems of all types. The manufacturing processes for silicon integrated circuits continue to improve, allowing designers to include ever-greater numbers of transistors, running at ever-higher frequencies, on every integrated circuit. This would seem to promise the opportunity to continue building digital systems of continually increasing sophistication, with dramatically better functionality and performance at each new design cycle, as has been the case throughout the history of the industry. These kinds of improvements are necessary to meet projected demands for lower cost, more energy efficient, and more powerful devices possessing substantial computational capabilities for diverse application workloads. For instance, in the near future cellular phones may be expected to work with multiple global telecommunications standards, support complex graphical user interfaces, allow real-time videoconferencing, encrypt data for secure transmission, and use sophisticated error-correcting codes for operation in environments with very low signal to noise ratios. Moreover, they will need to do all this with very little power and cost only a few tens of dollars to produce. Historic trends regarding improvements in digital system performance would seem to support the feasibility of these sorts of expectations. Unfortunately, it is becoming apparent that numerous trends are making this vision increasingly problematic.

Most of the cost reductions and performance gains seen in electronics in recent years have been enabled by the use of integrated circuits designed for a specific application. These application-specific integrated circuits (ASICs) are custom hardware designs that are optimized to meet the

performance and other requirements of the device in which they will be used. ASICs have also allowed for the replacement of many separate components with a single component, reducing cost and size. However, rapid increases in IC design and manufacturing costs are limiting the applications for which ASICs makes economic sense. Since ASICs are too expensive in many cases and standard processors cannot offer the performance and power efficiency needed for many applications, a newer approach, namely programmable ICs in the form of systems-on-chip (SoCs) with heterogeneous programmable IP blocks have become an increasingly important part of the semiconductor industry. Use of these types of ICs leads to a number of new problems, including the performance losses and reduction in power efficiency incurred when using programmable hardware as opposed to custom hardware, and the difficulties encountered when implementing applications on systems containing components with multiple programming models.

This thesis will explore one particular technology which may help address some of these problems. The HASTE (Hybrid Architectures with a Single, Transformable Executable) concept is a variant of a class of architectures which combine traditional sequential processors with spatially programmed computational fabrics, an architectural class often called a *hybrid architecture*. What makes HASTE unique is the use of a single executable to describe both portions of the application; both the portion that runs on the sequential processor as well as the portion that runs on the spatial fabric. To be more exact, portions of the executable can be automatically transformed at run time from sequential code into a spatial configuration. Hence the name, Hybrid Architectures with a Single, Transformable Executable. In particular, this thesis will investigate the costs and benefits associated with this property of transformability. The final results are quite encouraging. The results to be presented will show that HASTE is a viable and realistic technology that can be implemented with fairly conventional hardware and software and which is competitive with standard approaches.

This thesis can be considered to be divided into two sections. The first section discusses the HASTE concept itself, including system components, system operation, and application representation. It begins with a discussion of the motivation for HASTE, related work, and other background information in Chapter 2. Chapter 3 provides an introduction to HASTE and discusses some key concepts and theoretical constructs. The various instruction set architectures that can be used with HASTE are discussed in Chapter 4. Details of the reconfigurable fabrics

used in HASTE are covered in Chapter 5. The conversion of sequential executables to spatial configurations is explained in Chapter 6.

The second section discusses the experimental tools and techniques used to explore the HASTE concept as well the results obtained from these experiments. First, the tools used to create HASTE executables and to simulate HASTE operation are shown in Chapter 7. Next, the hardware models and techniques used to evaluate the performance of HASTE, as well as the results of performance evaluations of the HASTE fabrics, are presented in Chapter 10. Experiments showing correct operation of HASTE applications are shown in the next chapter, Chapter 9. Experiments comparing the different HASTE ISAs are shown in Chapter 8. The experiments in Chapter 11 compare the area and performance of HASTE implementations of various applications to implementations using semi-custom ASIC technology and implementations using modern FPGAs. Conclusions and future work are in Chapter 12.

Chapter 2

Background and Related Work

While the speed and transistor density of semiconductors continue to increase, it is becoming increasingly difficult to take full advantage of these advances. A range of trends is making it economically infeasible to use state of the art semiconductors in many applications. While these trends are diverse in nature, they can be divided into two main groups: those due to the increased complexity of manufacturing silicon semiconductors in the deep sub-micron (DSM) realm, along with the difficulties inherent in using DSM devices; and those due to the increased complexity of products designed for highly competitive and rapidly changing marketplaces. These two groups of trends will be referred to as silicon complexity and system complexity, respectively. The two terms are used in this thesis in a manner consistent to their usage in the International Roadmap for Semiconductor Technology [1].

2.1 Silicon Complexity

As their sizes approach nanometer scale, the basic behaviors and characteristics of semiconductor devices and interconnect change rapidly, and they will change even more rapidly in the near future as process technology continues past the 90 nanometer technology node. In particular, the basic switching activity of CMOS field-effect transistors (FETs) is scaling poorly, in the sense that the simple models of this behavior that have been used in the past are increasingly insufficient as the behavior becomes increasingly complex. In addition there are fundamental changes in terms of interconnect delay, power, and reliability [2]. Relative interconnect delay relates to the fact

that interconnect delay is increasing more rapidly than gate delay. The result is that the time it takes signals to propagate from gate to gate is proportionately a larger portion of the time available in one clock cycle. Hence interconnect delay is increasingly the limiting factor on clock speed and performance. This also means that global structures, which must be accessed in a single clock cycle from all portions of the die, are increasingly untenable [3]. Even where global structures are feasible, the interconnect capacitance greatly increases the power cost [4]. This requires designers to look at new types of designs that limit globally synchronous interconnect and requires better modeling of interconnect to achieve reasonable performance during synthesis.

DSM devices are much more demanding to manufacture, and require designers to consider manufacturability as part of the design process. However, this can require expensive and time-consuming simulations to measure the optical effects of mask interference patterns and other manufacturing details. Further, process variations across a wafer and even across a single die are harder to control in DSM processes and require more carefully designed circuits. Designs that provide redundant resources and fault tolerance designs may also be required to meet reliability and manufacturability requirements [5].

Since current microprocessor designs rely heavily on global structures, they are particularly susceptible to these problems. The critical path of a typical processor involves many large components, such as branch predictors, ALUs, and dispatch units, making meeting timing constraints particularly difficult [6]. Aggressive pipelining has been used to overcome this to a large degree, but this requires increasingly complex logic to handle pipeline stalls, correct for branch misprediction, and handle various other hazards, which then makes the overall design larger, in turn limiting the gains provided by pipelining.

DSM devices also have characteristics that diverge quite notably from standard FET models. They require much more careful circuit and system design to ensure proper operation [7]. These factors requiring more careful circuit design provide another indication that design cost problems will get worse rapidly, not only due to the inherent complexity of larger systems, but also due to more stringent requirements for circuit designs. It seems clear that some new methodologies for designing digital systems and ICs will be necessary to take full advantage of DSM technology.

2.2 System Complexity

As the number of transistors on a single chip increases, the complexity of the designs needed to utilize those transistors is also increasing. This increase in complexity is driving rapid increases in the cost of designing, verifying, and completing projects. These complexity-related costs, when coupled with a fast-paced and highly competitive market, place an enormous price on failed designs. So not only are the projects getting more complex, but failure has become so expensive that even more testing and verification is required to ensure success, driving costs even higher [1]. Current design flows for digital systems of all types are not scaling well, requiring enormous engineering efforts to produce very complex designs such as microprocessor and DSP designs. The organizational and administrative costs required for very large projects requiring hundreds of engineers are substantial, and seem to be growing much faster than linearly [8]. The costs of producing a new chip are increasing dramatically, for both these design reasons and for manufacturing reasons as well. For instance, mask sets for a single complex chip such as a microprocessor or DSP are expected to exceed one million dollars in the near future. This and other increases in non-recurring engineering (NRE) costs are making it even more important that chips work correctly the first time. Increasing design costs make custom design prohibitively expensive, yet denser designs, which require more full-custom layout, are needed to maintain profitability [5].

Trends in the consumer market seem to point to more computing devices being produced in large quantities for communications and entertainment products of various types. These devices will require significant processing power to handle tasks such as wireless protocols, encryption, streaming video, and compression. They will typically also be portable, putting severe constraints on available power. Most of these devices must sell for much less than the large desktop computers that have previously been the only products requiring comparable amounts of processing power, and the application workloads will differ greatly from those needed for today's typical desktop computer.

Given these difficulties, it seems that some new directions must be explored to provide the performance needed by future digital systems. It will no longer be sufficient to simply ride the wave of increasing processor performance and decreasing processor price as a means to enable new digital devices. Scaling todays superscalar processors to a billion transistors seems a daunting

task, and while it may be achieved, the end result will be inappropriate and unusable for many applications. More power efficient, flexible, easier to design and manufacture, and yet still very powerful architectures must be developed if digital technology is going to continue to show the incredible performance gains and price drops of recent years, which consumers and marketplaces expect will continue indefinitely.

2.3 Possible Solutions

2.3.1 Reuse

Given that system complexity and costs are rising so quickly, reuse at all levels of a design seems to be one of the only ways to build practical, affordable systems. Design reuse in this case ranges from simply designing chips that can be used across a range of applications and reused for future applications, to designing chips using IP blocks that can be reused across a number of chips, to using structures and layouts that can be reused many places across a single die.

Using one design for many applications seems difficult given the increasingly stringent requirements for power and cost. However, many researchers are considering the idea of a hardware platform [9], a programmable chip design that can be used for a set of related applications and which has the flexibility to be used for future applications as well. While these hardware platforms cannot match the performance efficiency of an ASIC designed for a specific application, the cost savings afforded by amortizing the chip design across the much higher volumes allowed by utilization in multiple applications can allow for a larger chip design with nearly the performance of a smaller ASIC at a much lower cost.

The general concept of a hardware platform is a system-on-chip (SoC) composed of multiple digital cores, and possibly analog cores, connected by a predefined communication scheme. The cores would be IP blocks that could be shared across multiple hardware platform designs and the communications infrastructure for sending data between cores could be predesigned and reused as well. Differentiation between different hardware platforms would be accomplished by including different combinations of cores, varying amounts of memory, and using different device packaging. Differentiation between applications would be accomplished by programming the cores and possibly the interconnect. The amount of programmability would vary from core to

core, but most, if not all cores would have some amount of flexibility. General-purpose processor cores could of course run different programs, but even fixed-purpose cores such as data converters could have programmable parameters. One important aspect of research into hardware platforms is determining what kinds of cores should be included. Including hardware optimized for specific applications seems to contradict the goal of producing flexible hardware, but more general purpose hardware may not provide the necessary performance and efficiency.

2.3.2 Reconfigurable Hardware for Hardware Platforms

One of the ways that ASICs achieve high performance is by the creation of custom datapaths tailored to the specific application. In order to duplicate this in a hardware platform, some sort of reconfigurable hardware is needed. Reconfigurable hardware in its most general sense is simply a set of functional units, storage, and interconnect, any or all of which can be configured for a specific application. A commercial field-programmable gate array (FPGA) is a common kind of reconfigurable hardware. A typical FPGA has a set of look-up tables (LUTs), which can be programmed to implement any Boolean function, registers which can be programmably included or excluded from the data path, and typically a rich set of programmable switchboxes and multiplexers. FPGAs thus have programmable functional units, storage, and interconnect. FPGAs are very flexible and can achieve very high performance for many applications. FPGA-like hardware has been considered for inclusion in hardware platforms. FPGAs have some drawbacks however, and are not the best or only choice for reconfigurable hardware. Programming FPGAs can be difficult, especially when computing tasks must be partitioned across heterogeneous processing resources. Effective partitioning requires accurate estimation of performance, but this requires time-consuming place and route for accuracy with FPGA fabrics. Further, the fine granularity of FPGAs can lead to less efficient implementation of applications with fixed data widths as compared to reconfigurable architectures with coarser granularity. Another problem with FPGAs is the long configuration times required by these devices. This is due in large part to another one of their disadvantages, the large size of the files needed to configure them. Some modern FPGAs have configuration files of tens of millions of bits [10]. Hardware that could provide most of the flexibility of FPGAs while avoiding these disadvantages would make hardware platforms more feasible.

One of the main promises of reconfigurable computing has been to provide the performance of an ASIC with the flexibility of a general purpose CPU, and that has been fulfilled to some degree. Reconfigurable architectures can implement custom datapaths and other logic with performance close to that of a comparable ASIC, while still having the ability to be reprogrammed for new tasks as needed. Reconfigurability can enable a single architecture to potentially be used for many different products and applications, and allows existing products to be given new features and functionality, while still providing high performance. This enables high-level reuse. In addition, reconfigurable architectures are typically well-suited to implementation by layouts composed of repeating design units, which enables a potentially large amount of low-level reuse, as does their suitability for inclusion as IP blocks in systems on a chip.

Like an ASIC, reconfigurable hardware achieves much of its performance by exploiting parallelism. Rather than separating operators in time, as in the sequential execution of instructions in a conventional processor, operators are separated in space, allowing many operations to occur simultaneously. In order for this to provide a performance advantage, an application must have a significant degree of available parallelism; applications that do not will not execute efficiently on reconfigurable hardware. Reconfigurable architectures are typically inefficient for applications with lots of branching and with irregular data access patterns, and many support little or no control flow, or support only certain kinds of memory access. This problem has typically been handled by either implementing the necessary functionality on an application specific basis, thus giving away most of the flexibility needed for widespread use, or by coupling some sort of general purpose processor with a reconfigurable fabric and partitioning applications so that sections of code operate on the hardware best suited for them.

While the latter is certainly a valid approach, it raises a number of issues that have been a severe hindrance to wide acceptance of this type of hybrid hardware. Performing the partitioning has been a much more difficult task than was initially anticipated. Partitioning is complicated by the fact that the CPU and the reconfigurable hardware typically have entirely different programming models, separate interfaces to external resources, and very different representations of applications. After partitioning and compilation/synthesis, an application consists of two or more separate executables, and these executables must be very specific as to the particular type of processor, reconfigurable fabric, and interface between them. For example, an application

running on a conventional workstation with an FPGA-based coprocessor card will require a standard executable for the main processor, as well as a configuration bit stream for each FPGA on the card. Updating an application requires modifications to all executables, which presents both developmental and logistical problems. This precludes any immediate performance increase from newer hardware, since at the very least generation of two new executables is required, and more typically a significant amount of redesign is required.

The interface between the processor and the reconfigurable logic is crucial to performance and difficult to implement well. Design and testing of interface code and logic can consume as much or more time as design and testing of application code and logic. Since interfaces are largely specific to particular instances of hardware, interface design and testing is usually not portable to new architectures and must be repeated for even minor hardware changes. This interface code and hardware also restricts the amount of code that can be efficiently run on the reconfigurable fabric. Only when the performance gained is greater than performance lost to interface overhead can use of the reconfigurable fabric be justified. This means that sections of code that do not run for very long, even if they have lots of available parallelism, cannot be sped up by running on the reconfigurable fabric.

Another problem that has hindered the development of reconfigurable hardware of all kinds has been the difficulty of programming. Bit streams for FPGAs are typically generated by creating designs in VHDL or Verilog and then using a series of tools to synthesize the design, map it to the specific technology, place and route the design to specific FPGA, and finally generate the bit stream itself. This is a much more complex process than compiling a program for high-level language and it can take much longer, up to many hours for particularly large and dense designs. Much effort has been spent to develop easier ways to program existing architectures such as FPGAs and to design architectures that are inherently easier to program; but it is still the case that reconfigurable systems take more skill and effort to program than conventional software. These requirements greatly reduce the pool of designers available to successfully implement applications using reconfigurable computing, and form a significant barrier to its use.

2.4 Goals

Reconfigurable hardware allows the flexibility, and opportunities for design and architecture reuse, that will be needed for future digital systems, particularly those implementing the kinds of data-intensive and/or data-streaming applications that are often implemented with ASICs today. As has been seen, however, it has other drawbacks that limit its usefulness and practicality for commercially viable products. In particular, the burdens of heterogeneous programming models and executables, the overhead imposed by interface requirements between heterogeneous hardware, difficulties imposed by tool flows, and the lack of portability and performance scaling are particular problems. If these problems could be solved, it would substantially increase the usability of reconfigurable architectures, which in turn could help solve many of the challenges facing digital designers in the coming years.

An ideal reconfigurable architecture would have a complete and unified computational model, such that an entire application could be programmed as a single entity, with no need for interface programming or design. It would allow for both temporal and spatial distribution of execution, matched to the characteristics of different portions of an application, but without requiring the designer to attend to the details of coordinating different modes of execution. In order to maximize performance, there would be little or no overhead required to use either mode of execution, so that all of an application that could benefit from spatial execution could do so. Such an architecture would have programming semantics and tool flow closer to that of conventional software than hardware design, so that it could be used by a much larger group of potential designers. Executables would be portable and run on different instances of the architecture without recompilation, allowing legacy applications for old versions of the architecture to run on newer versions of the architecture with improved performance. This ideal architecture would lend itself to implementations exhibiting a high degree of low-level reuse, and the architecture would itself be usable in a wide range of products and applications, enabling high-level reuse. This ideal architecture may not feasible in its entirety, but it does seem possible to achieve many, if not most, of its characteristics using HASTE. HASTE is targeted towards embedded systems that need to implement highly parallel multimedia and streaming applications with high performance as well as efficiency. This project has the following goals for the HASTE architecture:

- Allow the use of a single executable for both the sequential and spatial portions of an

application.

- Allow application specification from a high-level language, not a hardware description language.
- Create a configuration for the spatial portion of an application at run time, so applications can be sent to hardware of different sizes based on run-time performance requirements.
- Enable high-level reuse by providing a high-performance architecture which is programmable and usable for a wide range of applications.
- Enable low-level reuse by composing the architecture primarily from identical, repeated tiles.

2.5 Related Work

Research on reconfigurable computing architectures has been an active area for 15 years or more, although arguably the roots of this area go back to the early 1960s [11]. Throughout most of this period, the predominant use of reconfigurable computing involved a hybrid architecture of some sort, meaning that the reconfigurable hardware was used in a system that also included a conventional processor, often called a host computer. Compton and Hauck identify four different classes of hybrid architectures in their survey of reconfigurable computing [12]. They classify architectures by how closely coupled the reconfigurable hardware is with the conventional processor(s), ranging from systems in which there is little or no regular communications with the host to systems in which the reconfigurable hardware takes the form of a functional unit in the host processor itself. HASTE is an instance of the second-most closely coupled style of architecture, in that it operates as an independent coprocessor that can run without being controlled by the host, once the host initializes the hardware and provides the data to to be processed and/or the location of that data. The third-most closely coupled style generally communicates with the host only using external I/O and in general appears more like a separate processor.

Many reconfigurable architectures have been designed that operate essentially as coprocessors, as HASTE does, although unlike HASTE they all require that the reconfigurable hardware be programmed using something other than the sequential executable run on the host. Examples of

these architectures include such projects as Garp [13], Napa [14], OneChip [15], REMARC [16], and MorphoSys [17]. Other architectures such as PipeRench [18] and RaPiD [19] are not quite as closely coupled to a host processor, but would most likely be used in a manner more similar to that of a coprocessor, as with the systems listed above, as opposed to the loosely-coupled separate processor model.

All of these systems have shown substantial performance improvements over conventional sequential systems and the advent of commercial products show that these kinds of coprocessors are feasible and practical solutions to some of the problems facing more conventional architectures. However, all of these architectures suffer from the necessity to develop a conventional executable for the sequential processor and a very different configuration for the coprocessor. As has been discussed, this presents problems that HASTE is designed to circumvent by dynamically translating from one form to the other. Dynamic translation from one form of executable to another has been investigated by other projects, albeit in quite different contexts. The DAISY project [20] at IBM investigated the dynamic conversion of PowerPC executables to VLIW code. The Transmeta Cruesoe processor similarly translates x86 machine code into VLIW code at runtime [21]. While these are converting the code to target a different architecture, they are still targeting essentially sequential hardware, with inherent limits to parallelism, especially as compared to highly parallel spatial fabrics.

Many other research projects ([22] [23] [24] [25], among others) use some form of hardware-based dynamic translation or optimization to change a running executable on a conventional superscalar processor for better performance. While some significant performance gains have been realized, the hardware cost of these techniques is high and the performance still lags behind that observed with the reconfigurable coprocessors previously discussed. These techniques are again limited by the parallelism of the underlying processors. These approaches makes sense for general-purpose computing, but they are not well-suited to the high levels of parallelism and restrictions on power and area that the sorts of embedded multimedia applications that HASTE is targeted to tend to exhibit. Numerous projects, including [26], [27], and [28], have dealt with the memory access bottlenecks by implementing specialized hardware, similar in purpose to the MAU.

Chapter 3

HASTE Systems and Applications

Chapter 2 discussed some of the challenges facing the designers of digital systems due to increasing silicon complexity and increasing system complexity. Reconfigurable computational fabrics were shown to have many desirable characteristics but to present considerable drawbacks as well. Hybrid systems were introduced as a partial solution to some of these problems, but the need for two separate executables was shown to present additional difficulties. The HASTE concept solves the problems introduced by separate executables and provides mechanisms for close coupling of processors and reconfigurable fabrics by requiring only a single executable. This chapter will discuss the operation and basic characteristics of HASTE systems and applications and will define some of the terminology that will be used in the remainder of this thesis.

3.1 Haste Overview

HASTE is targeted towards embedded systems and SoCs that must efficiently implement highly parallel applications that operate primarily on streaming data. This category includes many applications of interest, especially those in the signal, image, and video processing domains, as well as encryption, channel coding, and other communication applications. These are the types of applications whose computationally intensive portions are typically implemented as custom datapaths in SoCs or ASICs. HASTE provides performance characteristics similar to custom datapaths for these applications, but since it is programmable it can be used with hardware platforms and programmable SoCs. While these applications seem to be the best fit for HASTE, this does not rule out the applicability of HASTE for more conventional processor workloads. That will remain the work of future research, however.

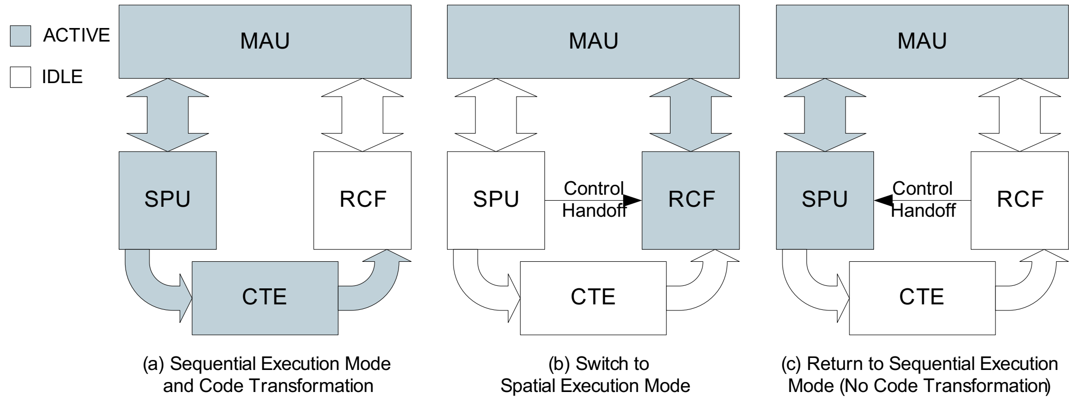


Figure 3.1: Basic HASTE Architecture and Operation

HASTE is an architectural concept, rather than a single architecture. This thesis will outline the design space and general characteristics of HASTE systems, and will then explore some specific implementations. It is important to note that there is no single HASTE architecture, only specific HASTE implementations, just as there is no single RISC architecture, only specific RISC implementations such as the MIPS 4KEc [29] or ARM 720T [30].

Figure 3.1 illustrates the operation of the basic HASTE architecture. All HASTE architectures have these same components: a sequential processing unit (SPU), a reconfigurable computational fabric (RCF), a code transformation engine (CTE), and a memory access unit (MAU). The RCF is an array of reconfigurable logic, reconfigurable interconnects, and storage, arranged to form a pipelined datapath; it will be discussed in detail in Chapter 5. The SPU is a Von Neumann processor which operates sequentially on a series of instructions which can implements control flow as well as data processing. It may be very similar to a conventional RISC processor, or a more specialized processor architecture may be used. With some minor changes to the HASTE concept as presented here, a modern superscalar processor could be used for the SPU; however, that will not be explored in this thesis. The CTE converts portions of the sequential instruction stream for the SPU into a configurations for the RCF. The MAU provides access to memory and memory-mapped I/O for both the SPU and the RCF, using streaming memory access instructions. The SPU may also access main memory directly, bypassing the MAU. The CTE, SPU, and MAU are covered in Chapter 6.

A HASTE application is composed of sections of sequential code, interspersed with computa-

tionally intensive kernels, each kernel being comprised of a single loop body which iterates a fixed number of times and has a single entry point and a single exit point. In the next section this application model will be considered in more detail. The basic operation of HASTE proceeds as follows: the SPU executes the code outside of the kernel or kernels, then executes one iteration of the loop body comprising a kernel, while the CTE creates a configuration for the RCF, as shown in Figure 3.1(a). Once this is complete, control is handed off to the RCF as shown in Figure 3.1(b). On subsequent loop iterations, the RCF uses this configuration to execute the loop body as a pipelined datapath, effectively executing the entire loop body in parallel. While memory access in the SPU can use either the MAU and streaming memory access instructions or conventional memory access instructions (loads and stores), the RCF must use the MAU and streaming memory access instructions exclusively. Once the kernel is complete, control passes back to the SPU, as shown in Figure 3.1(c), which executes sequential code until encountering the next kernel or the end of the program.

In a typical hybrid architecture, if it was desired that a kernel potentially run on both portions of the architecture, the kernel code would need to be compiled into an executable to run on the processor and then separately synthesized into a configuration for the reconfigurable logic, giving two separate executables. While the configuration could be embedded into the executable to create a so-called 'fat binary', the fact remains that there are two completely separate executables and there is no guarantee that they actually perform the same function. HASTE, by contrast, requires only a single executable that can run on both the processor and the reconfigurable logic, and in a properly designed HASTE system, the functionality of the kernel will be the same regardless of which type of hardware it is run on.

To make the HASTE concept work an appropriate application representation must be chosen. It must have a valid sequential semantic and yet also contain the information necessary for the CTE to create a spatial configuration. This application representation is an important part of the HASTE concept and this thesis will investigate and compare several different application representations. The idea of an application representation in this context encapsulates several things, including the form and encoding of the HASTE executable, which must be associated with the specific SPU architecture needed to process the it; the format of kernel configurations in the RCF; and the CTE algorithms used to convert the instruction stream into RCF configurations.

The remainder of this chapter will present models of the various components that make up both the application representation and the HASTE system itself.

3.2 Application Model

Before application representations are discussed, it is first necessary to be able to describe the applications themselves. For this purpose, a common application model (and corresponding C++ class; see Chapter 7) has been developed for HASTE. A HASTE application is composed of segments of sequential code and one or more computationally intensive *kernels*, sections of parallelizable code. Models for these two types of code will initially be examined separately and then unified later in this section. A number of assumptions about applications and kernels will be introduced in this section, some of which are inherent to HASTE and some of which are adopted for other reasons. Both types of assumptions will be identified when first introduced and then summarized at the end of the section.

3.2.1 Operations

An application is considered to be formed of a number of operations chosen from a set O of possible operations. This set is divided into three subsets: computation operations, data movement operations, and control flow operations. The set of operations of each type will be called O_c, O_m, and O_f, respectively, such that $O_c \cup O_m \cup O_f = O$. Each operation consumes some number N_i of input operands and produces some number N_o of outputs, with either or both N_i and N_o allowed to be equal to zero in certain cases. Each operation is represented in a control dataflow graph (CDFG) as a node, with data edges corresponding to operands and control edges corresponding to control flow. Note that the same operand can be carried on multiple data edges leaving an operation, so a node may have a number of outgoing data edges greater than N_o. If an operation produces a single output value ($N_o = 1$), than all outgoing data edges carry the same output value If an operation produces two or more distinct output values ($N_o > 1$), then each outgoing data edge must be designated as to which output value it carries. Inputs to nodes represent source operands, and only one value per input is allowed. So nodes may only have as many incoming data edges as their value of N_i.

Control flow operations do not produce data, so they have $N_o = 0$. A conditional branch may evaluate data, so $N_i \geq 0$ for control operations. Computation operations are required to have N_i and N_o both greater than 0; they take in one or more operands, perform some calculation on them and output one or more results. Data movement operations do not change operands, but instead move them in some architecturally dependent way, typically between registers, or between registers and memory. Operands moved from memory are not represented by input edges and are not counted in N_i; similarly, operands moved to memory are not represented by output edges and are not counted in N_o, so it is possible to have a data movement node with no edges in the case of a memory-to-memory transfer. Since memory-to-memory data transfers are not included in the current application model, however, all data transfer nodes have at least one input edge or one output edge, and may have both. A data transfer node with $N_i = 0$ and $N_0 > 0$ is called a ***data source*** and a data transfer node with $N_i > 0$ and $N_0 = 0$ is called a ***data sink***. Figure 3.2(a) shows a graph with 5 different operation nodes. Nodes 1 and 2 are data sources and node 5 is a data sink.

Note that this discussion of data transfer nodes indicates that a load/store architecture is being assumed in this thesis; the HASTE concept as currently developed assumes a load-store, aka register-register architecture, although it could potentially be expanded to other architecture types such as memory-register or memory–memory. It will also be assumed that every operation has the same latency, specifically a single clock cycle. This is a reasonable assumption for a RISC-type SPU architecture, except for load and store operations and possibly multiplication and other complex operations. The HASTE model could encompass multi-cycle operation latencies, but for this thesis, only single-cycle operation latencies will be investigated.

3.2.2 Sequential model

The sequential code model consists of a sequence of operations that can execute on the SPU, each drawn from a set of possible sequential operations O_s, which may contain all three kinds of operations (computation, data movement, and control flow), but which may not necessarily contain all of the operation in O, such that $O_s \subseteq O$. The operations in sequential code of each type will be called O_{sc}, O_{sm}, and O_{sf}, with $O_{sc} \subseteq O_c$, $O_{sm} \subseteq O_m$, and $O_{sf} \subseteq O_f$. Having operations that execute only in the RCF, i.e., operations that are in O but not in O_s, is counter to

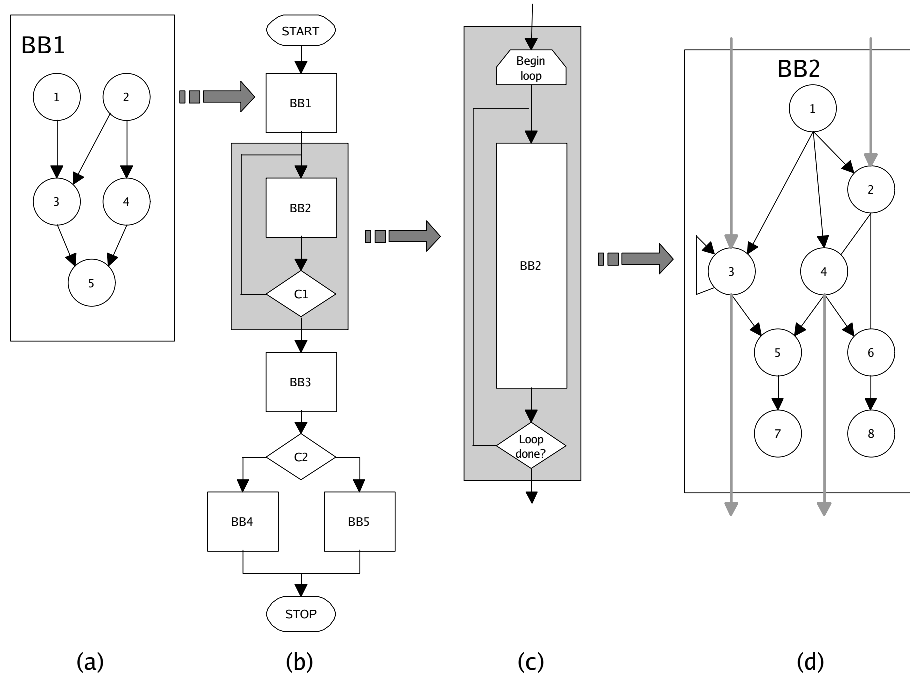


Figure 3.2: Application Model

the goals of HASTE as presented here, since that would force code containing those operations to run only on the RCF. Therefore, all operations are compatible with the SPU, so $O_s = O$ for the purposes of this thesis. Any sequence of n operations can be represented as s_n, with $s_n = (o_1, \ldots, o_n);\ o_i \in O,\ \forall i$. This sequence refers to the order of operations in the executable, which is just a particular representation of the application. As long as the data and control dependencies implied by the executable are respected, the actual order of execution in the SPU may be different than the order given in the executable.

The sequential code can be expressed as a conventional CDFG, with control flow operations (operations in O_{sf}) determining the execution of basic blocks. In Figure 3.2(b), for instance, a control operation, C2, is seen that determines whether basic block 3 (BB3) is followed by BB4 or BB5. The basic blocks contain only operations from O_{sm} and O_{sc}. Each basic block in the sequential code is expressed as a dataflow graph (DFG); a DFG is similar to a CDFG, but has no control flow. Each basic block has a particular sequential ordering applied to it. Note that the DFG form of the sequential code is explicitly represented only in the compiler; once

a sequential ordering is imposed, the DFG structure is only implicit in the sequence. For any given basic block DFG, potentially many different sequences of operations can be created. The only restriction on the sequence is that a valid topological order is followed; i.e., no operation can placed in a sequence before any of the operations on which it depends. Figure 3.2(a) shows a basic block, BB1, made up of 5 operations. Since this is a basic block from the sequential portion of an application, there are many different valid sequences for this basic block. For example, $(o_1, o_3, o_2, o_4, o_5)$, $(o_2, o_4, o_1, o_3, o_5)$ and $(o_1, o_2, o_4, o_3, o_5)$ are all valid sequences for this basic block. Note that each instruction sequence corresponds to just one DFG, the original DFG shown for BB1, so mapping from the set of DFGs to the set of instructions sequences is a one-to-many mapping.

3.2.3 Kernel Model

Kernels must execute on both the SPU and the RCF, so they must have a valid sequential order and otherwise meet all of the requirements discussed in Section 3.2.2. Kernel code has additional requirements not imposed on sequential code, however, since it must run on the RCF, not just the SPU. As in [31], the basic unit of code that can be executed in reconfigurable hardware is defined as being comprised of a single loop body and a single back edge spanning the entire code segment. However, HASTE requires that the loop body be a single basic block, instead of a hyperblock. While it seems feasible to execute alternate paths on the RCF, as is done on the Garp architecture described in [13], this concept will not be included in this discussion of HASTE. The kernel model does in effect allow control flow through the conversion of control flow into data flow. This is done by executing all paths in parallel and using select statements to choose the value to be used by successive operations. Kernel code cannot contain any actual control flow operations other than loop begin and loop end operations. While this may seem like a limited model, it is more than sufficient to support kernels from a wide range of applications. A kernel is thus a basic block, delimited by loop end and loop begin operations. These loop delimiters execute the basic block a number of times as the body of a loop. The number of times that the loop body executes must be fixed at the beginning of loop. This may be a runtime value, however; the iteration count does not have to be known at compile-time. Figure 3.2 (c) shows a kernel, composed of a basic block BB2 and loop delimiters. Not all basic blocks meeting

these requirements are kernels; only those that are compatible with operation on the RCF and have been designated as such are considered kernels.

As with all other basic blocks, an ordering must be placed on the operations in the kernel basic block so that can they be executed on the SPU as a sequential series of operations, and this ordering must be topologically valid. However, those basic blocks that are designated as kernels have further constraints placed on them. The algorithms used by the current CTE model, as will be discussed in Chapter 6, place restrictions on the ordering of the kernel operations. A further consideration for kernels is the source of operands and the destination of results. The HASTE concept requires a streaming model of memory access for the RCF and most operands will come from sequential memory locations (or addresses varying by a fixed stride) and most results will be stored in the same kind of memory locations. More complex memory addressing schemes could be used, as long as memory locations are known well in advance of their being needed, but are not implemented at present. Each data source node in the kernel represents a stream or vector of data that will be loaded from memory by the MAU and each data sink node represents a stream or vector of results that will be stored in memory by the MAU. Equivalently, MAU streams may instead be generated by or sent to external hardware using a standard FIFO like interface located at a specific memory address. Anywhere reference is made to accessing memory through the MAU, an external hardware interface may be in use instead, using this kind of memory-mapped I/O, unless otherwise specified.

While in theory HASTE could support various kinds of feedback in kernels, only a very limited form is currently supported. Allowing general feedback is problematic in a pipelined fabric and may require techniques such as c-slow retiming [32] in order to maintain performance. Allowing feedback also makes the task of the CTE much more complicated, since the source of a particular operand may be anywhere in the fabric. Feedback through the MAU is also never allowed; in other words, no address used as a destination for a data sink node can be subsequently read by a data source node. This is due to the difficulties inherent in providing consistent behavior in both the the RCF and SPU in the presence of unknown and arbitrary latencies in the MAU. Note that the SPU can perform arbitrary memory loads and stores outside of kernels using traditional memory instructions that bypass the MAU. While these restrictions on feedback may seem very limiting, in practice most applications of interest can be implemented with only the feedback

allowed by HASTE, and in fact most vector processors do not even allow the feedback that HASTE does.

There are some cases where a fixed value needs to be used in each loop iteration and that value is not known at compile-time; a constant value that is a parameter of the loop, for instance. This is represented as an edge that enters the basic block, with its source external to the block. All such edges are considered to represent static operands; that is, operands that are only written once by code outside the basic block, before the first iteration. These are also be called live-in values, after the common convention in compiler design. In other cases it may be required that a given value to be output only once after all iterations of the loop are complete; a sum of all the results of each iteration would be an example. These kinds of operands appear as edges that exit the basic block. These represent values that are read by code outside the hyperblock only after the last iteration and will be referred to as live-out values.

Figure 3.2(d) shows a basic block intended to run on the RCF. Node 1 is a data source and represents a stream of data read from memory. Nodes 7 and 8 are data sinks and represent streams of data stored in memory. Node 1 produces a new value each loop iteration and nodes 7 and 8 each consume a value each loop iteration. Node 2 has an input that has its source external to the basic block and is thus a live-in value. Node 3 and node 4 both have out edges that have targets external to the basic block and they represent live-out values. A node that produces a live-out value does not behave differently in regard to other nodes in the basic block than nodes that do not produce live-out values. For instance, node 5 has the output of node 4 as one of its source operands; it get the value produced by node 4 each iteration. The only value that gets passed outside the basic block, however, is the the last value produced by node 4. Other than live-in and live-out values, all data passed into and out of a kernel must go through the MAU.

In general, no feedback is allowed in kernel code; the DFGs for the kernel DFG must be acyclic. However, node 3 shows the limited feedback that is allowed: the output of a node can be used as one of its own inputs. This is particularly useful for counters and accumulators. These must have an initial value; in this example, this is represented by the in-edge for node 3 that has its source external to the basic block. This is a live-in value, but because it is the same input to node 3 that is also written by the output of the node, it does not stay constant like a typical live-in value. So if node 3 is a counter, for instance, the live-in value to node 3 represents an

initial value to the counter, and the live-out value represents the final count of the node.

Kernel code can be executed either in the SPU or in the RCF; thus all operations executable in the kernel, by definition members of the set O_k, must be compatible with both. Since every operation that can execute in the RCF can execute in the SPU, but not the converse, $O_k \supset O_s = O$, $O_{km} \supseteq O_{sm} = O_m$, and $O_{kc} \supseteq O_{sc} = O_c$. Since there is no control flow in the kernel, $O_{kf} = \emptyset$. The absence of control flow in kernels means that the sequential execution of kernels in the SPU is always the same, which is of course not true of SPU code in general.

3.2.4 Complete Application Model

As has been discussed, applications for HASTE consist of a sequence of operations, each corresponding to an instruction that can be executed on the SPU. This sequence can be divided into basic blocks of computation and data movement operations, with control operations determining the order of basic block execution. A complete HASTE application is shown in Figure 3.2(b). This application has five basic blocks, BB1-BB5, and two control operations C1 and C2. Basic blocks are represented as DFGs, with a sequential order applied to each DFG. The basic blocks may contain only operations from O_c and O_m, while control flow operations are by definition from O_f.

One of these basic blocks, BB2, has been designated as a kernel and therefore may only contain operations in the set O_k. This kernel may execute on either the SPU or the RCF, and in either case it must produce the same results and have the same side effects on the SPU state. In order to execute on the RCF, the kernel operation sequence must be converted into a configuration by CTE. If it cannot be so converted, than it executes on the SPU only. A number of assumptions and restrictions are imposed on the application model by the HASTE concept and others are adopted for simplicity in this early research on HASTE. Table 3.1 lists these assumptions and the reasons for each.

3.3 System Model

Given this application model, a method is needed to describe the system that the application executes on in terms of this application model. On the system hardware, each operation is

Table 3.1: HASTE Application and Kernel Assumptions

Assumption	Reason
All operations execute in a single clock cycle.	Simplifies HASTE modeling and simulation.
All operations can execute on SPU.	RCF only operations complicate HASTE for limited gain.
Kernel loop body is a basic block.	Simplifies RCF design.
Kernel basic block DFGs are acyclic.	General feedback difficult to implement in pipelined architecture.
No feedback through MAU.	Possible in general, but very dependent on MAU operation and latency
Iteration count must be fixed before first loop iteration.	Simplifies MAU and RCF operation.
No memory-stored operands allowed for computation operations in kernel	General memory access difficult to implement in pipelined fabric.

represented by a single machine instruction. The SPU can execute an entire application as a series of instructions executed in the order determined by the control flow instructions. Using a simple SPU model, it might be assumed that any sequence s_n of n instructions, $s_n = (o_1, \ldots, o_n); o_i \in O_S,\ \forall i$, can be executed on the SPU. To be precise, if S is the set of all possible sequences of instructions s from O_S, and S_{SPU} is the set of all sequences that can be executed on the SPU, then $S_{SPU} = S$. For other SPU models, this may not be the case, so $S_{SPU} \subset S$. An example of this might be the restrictions on sequencing required by the branch delay slot found in MIPS processors [33].

Any SPU program can be uniquely described as such a sequence of instructions. Note that this sequence does not designate the order in which the instructions are executed, which is determined by control flow operations that may be dependent on run-time data, but only the order of instructions composing the program. Upon running the program, a particular sequence of instructions will be executed; this executed sequence will be designated $ex(s)$. Note that $|ex(s)|$ is not necessarily equal to $|s|$, since control flow instructions, if present, can cause instructions in s to be skipped, resulting in a shorter sequence. For a sequence of length n, s_n, there is a set of p possible execution sequences $EX(s) = \{ex(s)_1, \ldots, ex(s)_p\}$ whose lengths in general are not equal to n. Since a kernel is a sequence of instructions that can be executed on the SPU, kernels can also thus be represented as a sequence of instructions $k = (o_1, \ldots, o_n); o_i \in O_k,\ \forall i$, with

$k \in S_{SPU}$. Since there is no control flow allowed in kernels, $p = |EX(k)| = 1$ and $|ex(k)| = |k|$.

The RCF does not execute sequentially, so a program running on the RCF cannot be described by a sequence. Instead, it is necessary to specify a ***configuration*** for the RCF. A configuration for the RCF requires the specification of where in the fabric each operation is located, where each operand is produced, and where each operand is consumed. This can be accomplished by specifying a dataflow graph, G, with each vertex in the DFG representing an operation in the kernel and each edge designating where an operand is produced and consumed; and a mapping m, composed of two one-to-one functions m_v and m_e, which map vertices and edges, respectively, to specific locations in the fabric. Note that the DFG is implicit in the instruction sequence but is not explicitly represented. It is the same DFG used for the basic block in the application model, as described previously, and thus there is one correct DFG that represents a particular kernel. There may potentially be many different valid mapping functions, however.

The mapping functions must meet several criteria. Every operation (vertex) in the DFG must be placed at a location in the fabric such that all of the operations it is dependent upon are located where they can be accessed. In addition, it must be possible to implement each edge given the available interconnect resources, and there must be enough of these resources so that all of the edges in the graph can be implemented. As will be shown, there are many different kinds of possible RCFs, and it is not necessarily true that a valid mapping can be found for a given DFG on a given fabric. Given a valid DFG and mapping, the fabric configuration is generated by specifying an operation for each processing element in the fabric and specifying the necessary values to control the fabric interconnect in such a way as to implement the DFG and mapping.

The goal of the CTE is thus to take a sequence of instruction representing a kernel, and from them produce the DFG and mapping functions so that the fabric can be configured. Given a sequence of instructions representing a kernel, k, the CTE should produce a configuration $c(k)$, or equivalently, a DFG G and a mapping, m, if one is possible for the fabric. If C_{RCF} is the set of configurations that can be implemented on the fabric and S_k is the set of all kernel sequences, than CTE performs the function $f : S_k \rightarrow C_{RCF}$. However, in general, a CTE may not be able to perform all such transformations. Only some sequences may be understood by the CTE and a valid configuration may not be possible to generate for every sequence. So S_{CTE} is let to represent all kernel sequences that a given CTE can both understand and generate a valid

configuration from. Then C_{CTE} represents the set of all configurations that the CTE can produce from elements of S_{CTE}, so $f_{CTE}: S_{CTE} \rightarrow C_{CTE}$. Note that a CTE will always produce the same configuration for a given sequence. Ideally, $S_{CTE} = S_k$ and $C_{CTE} = C_{RCF}$; but this is not usually the case. If a kernel sequence is not in S_{CTE} then that sequence can only be executed on the SPU. A different CTE might have a larger set of transformable sequences so that same sequence might be transformable on this more powerful CTE.

So the system model describes a HASTE system as having an SPU which can execute some set of sequences of instructions S_{SPU} with an individual sequence s composed of instructions from O_s. The actual executed sequence is $ex(s)$ and this executed sequence may vary in length from the original sequence. Some portions of these sequences are designated as kernels; a kernel sequence k is composed of instructions from O_k. A kernel sequence always executes in the same order, so $ex(k) = k$. If $k \in S_{CTE}$, than then CTE can transform it into an equivalent configuration $c(k)$.

3.3.1 Component Completeness

There are several different system characteristics that can now be described using this terminology. The first is SPU completeness. An SPU is complete if it can execute any sequence of instructions from O_S; this implies that $S_{SPU} = S$. An SPU is incomplete if it cannot execute some sequences of instructions (without generating an exception or other error condition), implying $S_{SPU} \subset S$. The queue ISA SPU that will be discussed in Chapter 4 is incomplete, because only sequences of instructions with certain properties can run on the SPU. Other sequences do not have a defined behavior or will have undesirable or unexpected behaviors. Most conventional SPU architectures are at least partially complete, although this of course does not mean that all sequences performs any sort of useful computation.

Given a sequence of SPU instructions, it can be designated as a kernel if it meets certain criteria, namely that it contains only instructions in O_k and that it meets the control flow requirements mentioned earlier. This kernel code can be guaranteed to run on the SPU if the SPU is complete or if the sequence is in S_{SPU}. (If neither is true and the code doesn't run on the SPU, than there is no reason to have any further interest in it!). In order to run on the RCF, however, there are two additional requirements. The first is that at least one configuration c exists that provides a proper mapping from the edges and vertices of a DFG G that is isomorphic

to the original kernel DFG, to the routing and computational resources of the RCF. If not, the kernel is not physically implementable on the specific RCF, at least as it was expressed in sequential code for the SPU. Changing the original DFG implementing the kernel in the compiler could avoid this problem in some cases, particularly if the exact RCF to be used was known in advance, but that is not always the case. If any kernel sequence that can be run on the SPU can be implemented on the RCF, than the RCF is called complete, assuming that the fabric is always sufficiently large. For a complete RCF, then for every sequence in S_k, there is at least one corresponding configuration in C_{RCF} that performs the same computation.

However, there is another criteria for running a kernel on the RCF, and that is that the CTE can correctly implement the kernel on the fabric, given the SPU sequence. It is possible for a RCF to be complete, but for the CTE to still be unable to map kernels that could be implemented on the RCF. If, however, the CTE can map any valid kernel sequence that can be run on the RCF to the RCF, than the CTE is called complete and it can be said that $S_{CTE} = S_k$ and $C_{CTE} = C_{RCF}$. Note that it is possible to have an incomplete RCF and still have a complete CTE, if all sequences that do have mappings to configurations on the RCF are transformable by the CTE.

3.4 Observations

A general description of HASTE applications and kernels and the operation of the SPU, CTE, MAU, and RCF has now been presented. However, these have all been described in very general and mostly abstract terms. In the next chapter, Chapter 4, the instructions sets that will be used to represent operation sequences will be discussed using more specific terms. The chapter after that, Chapter 5, will describe the HASTE hardware more thoroughly and in particular will describe the RCF in detail, which will provide the background necessary for understanding what RCF configurations consist of. Once the instructions sets and fabrics have been explained, it will be possible to describe how the CTE works in detail; this will be covered in Chapter 6.

Chapter 4

Instruction Set Architectures

The previous chapter described the application and system models for HASTE, but it must still be explained how executables that run on a sequential processor can be transformed into configurations for a spatial fabric. There are two main components that make this possible. One is the CTE, which will be covered in Chapter 6, and the other is the set of instructions that kernels are expressed in, which is the subject of this chapter. The kernels need to be expressed in a manner that will ensure that the kernels will run and give the same results on both the SPU and the CTE. So the executable for each kernel must have a valid sequential meaning and also must contain enough information to allow for the construction of a valid configuration. A configuration means information about the placement of operations and the connections between operators needed to produce the same results as the sequential semantic; in effect, placement and routing of the fabric. There are several possible ISA types that meet these requirements and they will be presented in this chapter. Experiments comparing them are reported in Chapter 8.

4.1 Generic HASTE Assembly Language (GHAL)

All HASTE applications are initially expressed in the Generic HASTE Assembly Language, or GHAL. This is a conventional register-based assembly language, based on the Portable Instruction Set Architecture developed for use with the SimpleScalar simulator [34]; PISA in turn is based on the MIPS ISA [33]. GHAL serves as an intermediate format between the original application source code and the ISA-specific executables that actually run on HASTE architectures. Unlike

most intermediate formats, however, it can be executed directly using a simulator. Simulations of GHAL are performed using a modified version of SimpleScalar; this was possible since GHAL is an extension of PISA. The main differences between GHAL and PISA are threefold; the addition of special instructions to designate the beginning and end of kernel loops and control loop execution; the addition of streaming memory instructions; and the introduction of a select instruction that allows for the implementation of some control flow by converting it into data flow using if-conversion [35]. In addition, GHAL has 32 more general purpose registers than PISA does. This reduces register spills in the compiler and simplifies the process of creating GHAL code. There are a few other minor difference between the GHAL and PISA, relating mainly to some instruction names and operand ordering. These can be seen by comparing the GHAL reference in Appendix B to the PISA instruction reference in [34]. The three main differences between GHAL and PISA are covered in this section, however, and a list of the most important new instructions in GHAL and not in PISA is shown as Table 4.1. A simple PISA assembly program and its GHAL equivalent are shown in Figures 4.1(a) and 4.1(c), respectively, and these will be used as examples throughout this section. These programs both implement the kernel portion of the simple C program shown in Figure 7.3. Note that this particular example is also covered in more detail in Chapter 7.

4.1.1 Loop Delimiters

Loop delimiters have two important functions in HASTE: they are used to identify kernel code and they can also control iteration of kernel loop bodies. The identification function is used for profiling execution and can also be used to tell the SPU that a kernel is starting and that it should start passing instructions to the CTE for transformation into a configuration. The control function implements the actual iteration of the loop body if it executes on the SPU and it enforces the requirement that the number of loop iteration be known before the first loop iteration. There are two kinds of loop delimiters; dummy loop delimiters and active loop delimiters. The dummy loop delimiters DLPBGN, DLPEND, and DLPDONE do not actually execute; they are only used for profiling kernels and are not counted in any simulation statistics. Technically, they are not even part of GHAL, but are used for profiling PISA versions of kernels. In Figure 4.1(b), the dummy loop delimiters are present on lines 6, 24, and 26. The DLPBGN instruction on

Table 4.1: New Instructions in GHAL

Instruction	Syntax	Description
DLPBGN	DLPBGN	Dummy loop delimiter, used for profiling only. Placed before beginning of loop body.
DLPEND	DLPEND	Dummy loop delimiter, used for profiling only. Placed as last instruction in loop body.
DLPDONE	DLPDONE	Dummy loop delimiter, used for profiling only. Placed as first instruction after end of loop body.
LPBGN	LPBGN *rd, rs*	Loop delimiter for beginning of loop. Register *rd* is used to hold loop iterator and iteration count is taken from value in *rs*.
LPBGNI	LPBGN *rd, uimm*	Loop delimiter for beginning of loop. Register *rd* is used to hold loop iterator and iteration count is taken from unsigned immediate *uimm*.
LPEND	LPEND *rd, target*	Loop delimiter for end of loop. Register *rd* holds loop iterator and target is label at beginning of loop (after LPBGN). Value in *rd* is decremented and if > 0, execution branches to *target*.
SETB	SETB *p, addr*	Set base address for port *p* to *addr*.
SETS	SETS *p, uimm*	Set stride for port *p* to unsigned immediate *uimm*.
RECV	RECV *rd, p*	Receive word from memory port *p* into register *rd*.
RECVB	RECVB *rd, p*	Receive byte from memory port *p* into register *rd* and sign-extend.
RECVBU	RECVBU *rd, p*	Receive byte from memory port *p* into register *rd*.
RECVH	RECVH *rd, p*	Receive half-word from memory port *p* into register *rd* and sign-extend.
RECVHU	RECVHU *rd, p*	Receive half-word from memory port *p* into register *rd*.
SEND	SEND *p, rs*	Send word in register *rs* to memory port *p*.
SENDB	SENDB *p, rs*	Send low byte in register *rs* to memory port *p*.
SENDH	SENDH *p, rs*	Send low half-word in register *rs* to memory port *p*.
SEL	SEL *rd, rs, rt, ru*	If *rs* is zero, copy value in register *rt* to register *rd*. Otherwise, copy value in register *ru* into register *rd*.

```
1) simple_kernel:
2)    .frame    $sp,0,$31
3)    .mask     0x00000000,0
4)    .fmask    0x00000000,0
5)    addu      $7,$5,20000
6) $L43:
7)    lbu       $2,0($4)
8)    lbu       $3,1($4)
9)    lbu       $6,2($4)
10)   srl       $2,$2,2
11)   addu      $3,$3,$2
12)   sltu      $2,$6,$3
13)   beq       $2,$0,$L44
14)   move      $2,$6
15)   j         $L45
16) $L44:
17)   addu      $2,$3,4
18) $L45:
19)   sh        $2,0($5)
20)   addu      $5,$5,2
21)   addu      $4,$4,3
22)   slt       $2,$5,$7
23)   bne       $2,$0,$L43
24)   j         $31
25)   .end simple_kernel
```

(a) Original PISA Code

```
1) simple_kernel:
2)    .frame    $sp,0,$31
3)    .mask     0x00000000,0
4)    .fmask    0x00000000,0
5)    addu      $7,$5,20000
6)    dlpbgn
7) $L43:
8)    lbu       $2,0($4)
9)    lbu       $3,1($4)
10)   lbu       $6,2($4)
11)   srl       $2,$2,2
12)   addu      $3,$3,$2
13)   sltu      $2,$6,$3
14)   beq       $2,$0,$L44
15)   move      $2,$6
16)   j         $L45
17) $L44:
18)   addu      $2,$3,4
19) $L45:
20)   sh        $2,0($5)
21)   addu      $5,$5,2
22)   addu      $4,$4,3
23)   slt       $2,$5,$7
24)   dlpend
25)   bne       $2,$0,$L43
26)   dlpdone
27)   j         $31
28)   .end simple_kernel
```

(b) Instrumented PISA Code

```
1) simple_kernel:
2)    .frame    $sp,0,$31
3)    .mask     0x00000000,0
4)    .fmask    0x00000000,0
5)    setb      P1,0($4)
6)    sets      P1,3
7)    setb      P2,1($4)
8)    sets      P2,3
9)    setb      P3,2($4)
10)   sets      P3,3
11)   setb      P4,0($5)
12)   sets      P4,2
13)   lpbgni    $40,10000
14) $L35:
15)   recvbu    $2,P1
16)   recvbu    $3,P2
17)   recvbu    $6,P3
18)   srli      $2,$2,0x2
19)   addu      $3,$3,$2
20)   addiu     $48,$3,4
21)   sltu      $44,$6,$3
22)   sel       $2,$44,$6,$48
23)   sendh     P4,$2
24)   lpend     $40,$L35
25)   j         $31
26)   .end simple_kernel
```

(c) Equivalent GHAL Code

Figure 4.1: Equivalent PISA, Instrumented PISA, and GHAL programs

line 6 identifies the beginning of the loop body and is placed just before the first instruction in the loop body. It only executes one time, and the next instruction after DLPBGN is the first instruction in the first iteration of the loop. The DLPEND instruction on line 24 identifies the last instruction in the loop body except for the loop-closing branch statement on line 25. DLPEND executes once for each loop iteration. The DLPDONE instruction on line 26 identifies the end of all loop iterations and is placed after the loop-closing branch. It executes only once, after all loop iterations are complete. These three instructions are all needed for accurate kernel profiling. The profiling process is covered in Chapter 9.

The active loop delimiters LPBGN and LPEND are actually executed and perform the branching for kernels when executing on the SPU and otherwise delimit the code that is contained in a kernel. The LPBGN instruction specifies a register to hold the loop counter and a count of the number of times to iterate; this may be supplied as either a constant (in which case the immediate instruction version LPBGNI is used) or the value may be in a register (in which case LPBGN is used). In Figure 4.1(b), there is a LPBGNI instruction on line 13. It designates reg-

```
/* * simple.c * */
#define NUM_DATA 10000
#include <stdlib.h>
void simple_kernel (unsigned char* in_data, unsigned short* out_data);
int main(void) {
  unsigned char *p_in;
  unsigned short *p_out;
  int i;
  p_in = malloc(3*NUM_DATA*sizeof(unsigned char));
  p_out = malloc(NUM_DATA*sizeof(unsigned short));
  for (i = 0; i < (NUM_DATA*3); i++) {
    p_in[i] = rand();
  }
  simple_kernel(p_in,p_out);
return 0;
}
void simple_kernel (unsigned char* in_data, unsigned short* out_data) {
  int i;
  unsigned char a,b,c;
  unsigned short x,y,z;
  asm("dlpbgn");
  for (i=0;i<NUM_DATA;i++) {
    a = in_data[i*3];
    b = in_data[(i*3)+1];
    c = in_data[(i*3)+2];
    x = a >> 2;
    y = b + x;
    if (y > c)
      z = c;
    else
      z = y + 4;
    out_data[i] = z;
    asm("dlpend");
  }
  asm("dlpdone");
}
```

Figure 4.2: Original C code for simple example program

ister \$40 as the loop counter and sets an immediate loop iteration count of 10,000. The LPEND instruction specifies the loop count register and a branch target. The instruction decrements the counter register and checks to see if it is greater than zero. If it is, execution continues at the branch target address. If not, iteration is complete and execution continues to the next instruction in memory. The LPEND instruction can be seen on line 24 of this example. It identifies the same register as the loop counter as was used for LPBGN, register \$40, and specifies the branch target address using the label \$L35 placed just before the first instruction in the loop body. These two instructions replace the instructions on lines 5, 21, 23, and 25 of the original code in Figure 4.1(a). Although nothing actually prevents it, modifying the loop counter register during execution of the loop body can cause unexpected results and complicates operation of the MAU. A general purpose register was used here to allow for more flexible loop iteration in future versions of GHAL and HASTE; a special interlocked loop counter could have been used had it been deemed necessary to enforce the restriction on modifying the loop counter.

4.1.2 Streaming Memory Accessors

To match the streaming memory model mentioned earlier, GHAL has a set of streaming memory access instructions. These *send* and *receive* instructions correspond to *store* and *load* operations of a conventional ISA. Rather than referring to memory addresses, however, they refer to memory access ports. The current GHAL design assumes hardware with 32 memory ports; ports 0-15 are receive ports and ports 16-31 are send ports. Prior to issuing a *send* or *receive* instruction, and prior to kernel initiation, the port must be set up by supplying a base address using the SETB command and a stride using the SETS command or their variants. Only some of the variants on SETS and SETB are shown in Table 4.1, but all are covered in Appendix B. The first *receive* instruction using a port gets the memory value at the base address; subsequent receives of the same port get the values in memory at subsequent addresses that are obtained by adding the stride value to the base address at each access. *Send* instructions work similarly, with the first send to a port sending data to the base address, and subsequent ones sending data to subsequent addresses. A stride of zero may be used for access to memory mapped I/O. In Figure 4.1(b), three receive ports and one send port are in use. The receive ports, ports 0,1, and 2, are set up in lines 5 through 10, and the receive port, port 16, is set up in lines 11 and 12. In this example, port 0 is

set with a base address of 0($4), meaning a zero offset from the address in register $4, port 1 has a base address of 1($4), meaning a one-byte offset from the address in register $4, and port 2 has a base address of 2($4), meaning a two-byte offset from the address in register $4. Each of these three receive ports has a stride of 3 bytes (set in lines 6, 8, and10). Just as for the *load* and *store* instructions in PISA, there are *send* and *receive* instructions for words, half-words, and bytes; the receive instructions for half-words and bytes have both signed and unsigned versions which control whether sign-extension is performed. All of these memory accessors are listed in Table 4.1. In this example, it can be seen that all three receives, in lines 15, 16, and 17, are receives of unsigned bytes. So the access patterns is that in the first iteration three consecutive bytes are read from addresses 0($4), 1($4), and 2($4), in the next iteration three consecutive bytes are read from addresses (0+3)($4) = 3($4), (1+3)($4) = 4($4), and (2+3)($4) = 5($4), and so on. Looking at lines 8, 9, 10, and 22 in Figure 4.1(a), it can be seen that the GHAL code is in fact reading the same data from memory as the PISA version. The GHAL code is simpler, because address incrementing is done automatically, so no instructions equivalent to those on lines 21 and 22 of the original PISA code are needed in the GHAL code.

4.1.3 Select Instructions

Since no control flow is allowed in the kernels, *select* instructions are introduced into the ISA; see `sel` in Table 4.1 for the syntax. The instruction has three source operands; a single selector and two possible select values. Depending on whether the selector is zero or non-zero, either the first or second select value is passed to the destination operand. In the PISA code in Figure 4.1(a), it can be seen that a different value ends up in register $2 depending on whether the branch in line 14 is taken or not. If it is taken, then $2 gets the sum of registers $3 and $4 in line 18; if the branch is not taken, it gets the value in register $6 in line 15. This corresponds to the *if..else* construct in the original C code. In the equivalent GHAL code registers $3 and $4 are always summed on line 20. Then either this sum or the value in register $6 is placed in register $2 by the select statement on line 22. Note that the number of instructions executed in the PISA code varies depending on the branch taken, but the number of instructions executed in the GHAL code is always the same.

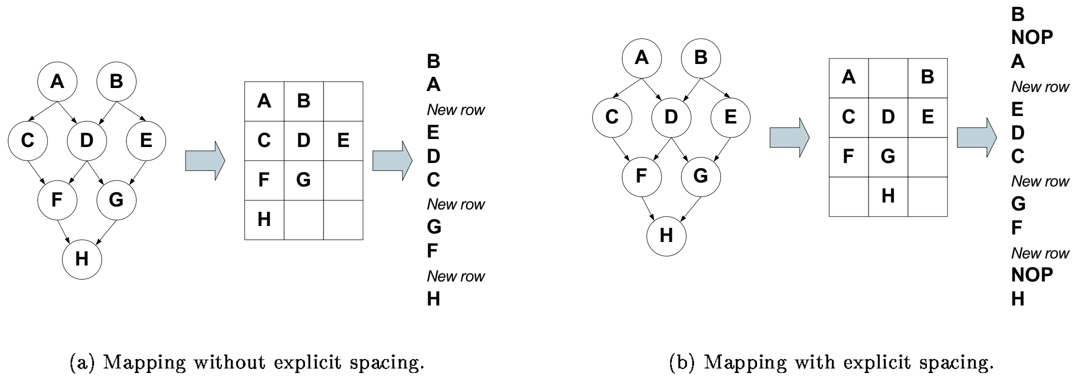
4.2 ISA Requirements

While GHAL has several features that improve its suitability for HASTE, it is still not easily transformable, at least not without further consideration of the transformation process. It is necessary to be able to take the sequential code for the kernel, k, and from it determine the mapping function m, and at least indirectly the kernel DFG, G. Equivalently, the kernel application needs to be placed and routed on the RCF. To construct G, it is necessary to determine the nodes and edges of the graph. To determine the mapping function, it is necessary to know where on the fabric each of these nodes and edges goes. The ISA needs to implicitly or explicitly provide enough information to the CTE so that it can produce G and m. As discussed in the previous chapter, m is composed of two sub-functions, m_v and m_e which map vertices(nodes) and edges, respectively, from G to the fabric. In order to discuss what the requirements are for the ISA, it is necessary to discuss G, m_v, and m_e in more detail.

4.2.1 Operations and Placement

In the current HASTE implementation, the nodes in G correspond on a one-to-one basis to the instructions in the kernel sequence, with the exception of the loop delimiters, which are not considered part of the kernel. So a sequence of n instructions (not including delimiters) corresponds to a DFG with n nodes, and vice versa. The functionality of each operation in the fabric thus corresponds to the functionality of each corresponding instruction in the sequence. So determining the number of nodes and the operation performed by each is simple. An instruction sequence like that shown in Figure 4.1(b) with three `recvbu` instructions, and one each of `srli`, `addu`, `addiu`, `sltu`, `sel`, and `sendh` instructions, will have a corresponding DFG with three `recvbu` nodes and one each of `srli`, `addu`, `addiu`, `sltu`, `sel`, and `sendh` nodes. Note that the loop delimiting instructions and label on lines 13,14, and 24; the memory access setup instructions on lines 5-12; and the subroutine-related code on lines 1-4, 25, and 26 are not part of the kernel DFG.

In addition to the number and type of nodes, the CTE needs to determine where in the fabric to place each node. While in theory the CTE could determine the placement entirely by itself from an arbitrary sequence of instructions, this would require a much more complex CTE than would be practical. The HASTE concept assumes that most of the work of determining the



(a) Mapping without explicit spacing.

(b) Mapping with explicit spacing.

Figure 4.3: Application Mapping in Compiler

configuration will be done at compile time, so it is only necessary to give enough information about the desired configuration for the CTE to recreate the configuration. So for a given DFG, the compiler determines a mapping for that DFG to a specific ISA and fabric. In Figure 4.3(a) can be seen a mapping for a DFG to a simple fabric with four rows (or pipeline stages) and three columns. Since it is desirable that the CTE be able to complete its job as quickly as possible, ideally within a single loop iteration, it would be preferable to be able to determine the placement for each node as soon as it is seen by the CTE. Given the pipelined nature of the RCF, it makes sense to order the instructions in the sequential code such that all instructions for a given pipeline stage are in the sequence before any instruction for any subsequent pipeline stages. This also ensures that the topological ordering requirement for sequential code is met. This still leaves the question of where in each stripe to place each operation. Again keeping with the the premise that the CTE's job should be made as simple as possible, the instruction sequence is simply created such that each instruction appears in the order that it is placed in the fabric. The instruction sequence in Figure 4.3(a) was created by simply copying the operations from right to left, starting with the first pipeline stage and continuing on to successive pipeline stages. Figure 4.4 shows how the DFG is recreated by the CTE by reading the instruction sequence and "pushing" each instruction onto the pipeline stage (Note that this process is covered in much more detail in Chapter 5). Some indication of the need for a new row is needed; the exact form this takes differs between ISAs. In any case, the new row signal causes the current row to move up into the fabric, and the CTE can start placing the next row. Determining the nodes in G

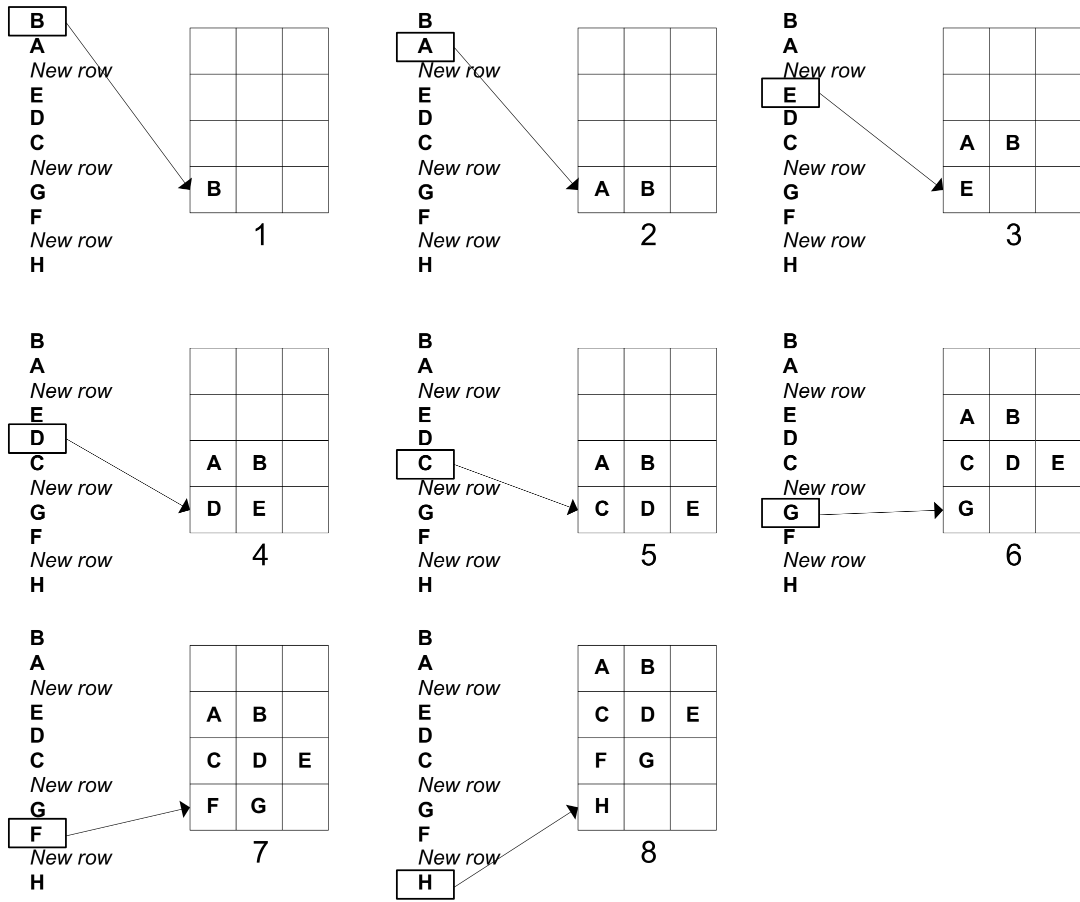


Figure 4.4: Placement of Operations by CTE

and where each node goes in the fabric (m_v) is thus quite straightforward. The only complexity arises when it is necessary to place operations in the fabric such that are spaces between them, as shown in Figure 4.3(b). This might need to be done because of routing restrictions in the fabric, for instance. In this case, NOOP instructions are inserted into the instruction sequence to properly space the operations on each row. This causes some inefficiency in the sequential code, since these NOOPs must be fetched and decoded, but this is necessary in order to keep CTE operation simple. These same placement techniques are used for all ISAs; the only difference is in how the "new row" signal is generated, and this will be discussed for each ISA in turn.

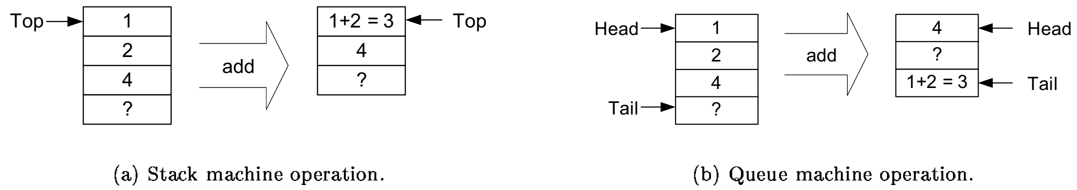
4.2.2 Data Flow and Routing

The more difficult part of determining the structure of G is finding the edges, which represent data dependences between instructions, or equivalently, represent data flow between operations. The exact manner that these dependences are determined is different for each specific ISA. Once these are known, the CTE must determine how to configure the interconnect to implement the correct data flow. This is in effect implementing routing for the RCF and is covered in detail in Chapter 5. Performing a general routing task where connected operations could be anywhere in the fabric would be quite difficult to do at run time with RCF. However, the pipelined nature of the RCF greatly simplifies the routing problem. For any given operation, it is only necessary to route operands from a register in the current stripe, and the result of each operation needs only to be routed to a register in the next stripe. So once the input operands for an operation are known, as long as the locations of those operands are known, the required routing can be determined. Since all of the input operands for any given instruction must have been generated or stored in the previous stripe, and since instructions are sequenced in stripe order, all input operands must already have been seen by the CTE for any given operation needing those operands. So the main information the CTE needs is what input operand(s) each operation needs. This varies from ISA to ISA and will be covered in the section for each ISA.

4.3 Queue ISA (QISA)

The easiest kind of ISA for the CTE to convert is a queue ISA, which is an ISA targeting a queue machine. A queue machine is analogous to the more familiar stack machine, such as the Java Virtual Machine [36], except that it uses an operand queue rather than an operand stack. Figure 4.5(a) shows an operation using an operand stack. In a stack machine operands are always taken off of the top of the stack and the result is placed back on the top of the stack. In this example, the operation is an add, so the top two values, 1 and 2, are taken from the stack and added. The result, 3, is placed back on the top of the stack.

An operand queue does not have a top, but instead has a head(front) and tail(back). Operands are always taken from the head of the queue and results are always placed on the tail of the queue. In the example shown in Figure 4.5(b), the same operation performed in the stack machine



(a) Stack machine operation.

(b) Queue machine operation.

Figure 4.5: Stack and Queue Machine Operation

example is performed using a queue machine. The two operands, 1 and 2, are taken from the head of the queue and added. The result, 3, is placed on the tail of the queue.

The queue machine is used for HASTE because the inherent ordering of operations better matches the requirements for the types of instruction sequences needed to ensure that a simple CTE can be used. Because of the use of a fixed data storage structure and rules that determine where operands are taken from for each operation, both queue and stack machines place strict limits on the instruction sequences that are valid for a given DFG. Figure 4.6 shows a simple DFG and the instruction sequences generated from it for a stack machine and for a queue machine. Note that in the stack machine schedule, operations that are on the same level of the DFG are separated by operations on a different level. The arrows connect blocks of operations that are on the same level of the DFG. The queue machine schedule has all of the operations that are on the same level of the DFG in sequence. While there are different possible stack and queue machine schedules, it is shown in [37] that any correct sequencing of the example DFG for a stack machine will separate operations that are on the same level and should be adjacent for correct CTE operation. Further, a correct sequencing for a queue machine will always keep operations together that should be on the same level. So queue machines require instruction sequences that are an exact match for the type of instruction sequences that are desired for simple CTE operation.

Given some arbitrary stream of instructions, it may be possible to reorder them so that they can run on a queue processor, but this is not true of most instruction sequences. Fortunately there is a correspondence between certain properties of a DFG and the correct ordering of instructions for a queue machine. Any DFG that is both *level* and *planar* will produce an instruction sequence that will run correctly on a queue machine [37]. A level graph is one that can be drawn with the nodes in rows such that every edge goes from a node in one row to a node in the next row

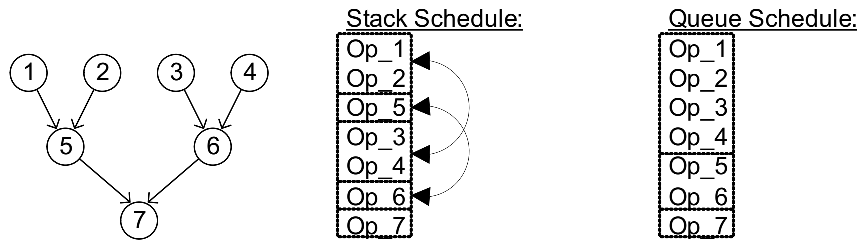


Figure 4.6: Instruction Ordering for Stack and Queue Machines

and no edge skips a row. An example of a non-level graph and a level graph are shown in Figure 4.7(a) and Figure 4.7(b), respectively. A planar graph is one that can be drawn without any edges crossing. An example of a graph that is both level and planar is shown in Figure 4.7(c).

By adding new nodes to a DFG, a level-planar graph can be produced from any arbitrary DFG, so queue machines can be used to implement any instruction sequence, if certain queue manipulation instructions are allowed. These instructions are SWAP, which replaces edges that cross, and PASS, which allows edges to cross levels. In the queue machine, a PASS instruction removes an operand from the head of the queue and places it on the tail of the queue, and a SWAP instruction removes two operands from the head of the queue, reverses their order and places them on the tail of the queue. This process of making an arbitrary DFG both level and planar is illustrated in Figure 4.7. The original non-level, non-planar graph is shown in Figure 4.7(a). The graph is first made level by add PASS nodes to each edge that crosses more than one level; the level version is shown in Figure 4.7(b). Next the graph is made planar by inserting SWAP nodes everywhere that two edges cross. Finally, the graph is made level again by inserting PASS nodes for all edges spanning the level where a SWAP node was added. The final graph is shown in Figure 4.7(c).

The queue ISA has some drawbacks that seem to make it a rather poor choice for use in HASTE. Use of the queue ISA requires that the SPU be a queue machine, which requires construction of a novel type of processor. Further, all application code would have to be compiled to run on the queue-based ISA. Many PASS and SWAP operations have to be added to queue DFGs to make them level planar, so the corresponding code is longer than it would be otherwise. The level planarity property makes it impossible to directly indicate feedback in a queue ISA DFG as can be done in a generic DFG, as was shown in Figure 3.2(d). Currently single operator feedback

is only possible in QISA by using a special instruction type which includes feedback implicitly. The only such operation currently implemented is ZACC, the accumulate from 0 operation, as described in Appendix B. Finally, the queue ISA allows a fanout of at most 2 from any node; in other words, any result may only be read at most 2 times. The PASS2 instruction takes an operand from the head of the queue and places two copies of it on the tail of the queue. This allows a result to be read more than twice, but this requires the addition of yet more instructions.

The queue ISA does have several advantages, however. Because the operands used by each operation are implied by the ordering of operations in the sequence, no explicit operation designation, like specifying register numbers in a conventional ISA, is needed. This means that each instruction need only contain a field to designate what operation is to be performed and can thus be much smaller than instructions for other types of ISAs. The job of the CTE is simplified because it doesn't have to keep track of explicit operand designations; it can determine implicitly where to get the operands for each operation. All result operands are written to the same column, so the CTE always knows where to find them. In addition, the CTE can determine when a new row is needed just by observing the number of operands available in a given row. A queue ISA allows the use of a very simple CTE, as will be discussed in Chapter 6.

The queue ISA has all of the instructions in GHAL, except that unlike GHAL, no register operands or other operands designation is required. The only operands needed are immediate values for instructions that require them. In addition to the standard GHAL operations, the SWAP and PASS instructions, as previously described, are also included. Another queue specific instruction is DROP, which simply consumes a value off of the queue. This is typically only needed for instructions that produce two different values, only one of which is needed. The queue ISA requires some variants on the standard GHAL instructions as well. Normally, a queue instruction places one copy of its result on the queue, which means that operand can only be used once. Equivalently, each node in the DFG would only have a single out edge. This is impractical for most applications, so instructions are allowed to have variants that produce two identical results. To designate this a '2' is simply appended to the instruction. Thus an ADD instruction adds two operands and places the sum on the queue; an ADD2 instruction performs the same addition but places two copies of the sum on the queue. This is illustrated in Figure 4.8(a). The ISA reference in Appendix B shows which instructions can have their output doubled in this way.

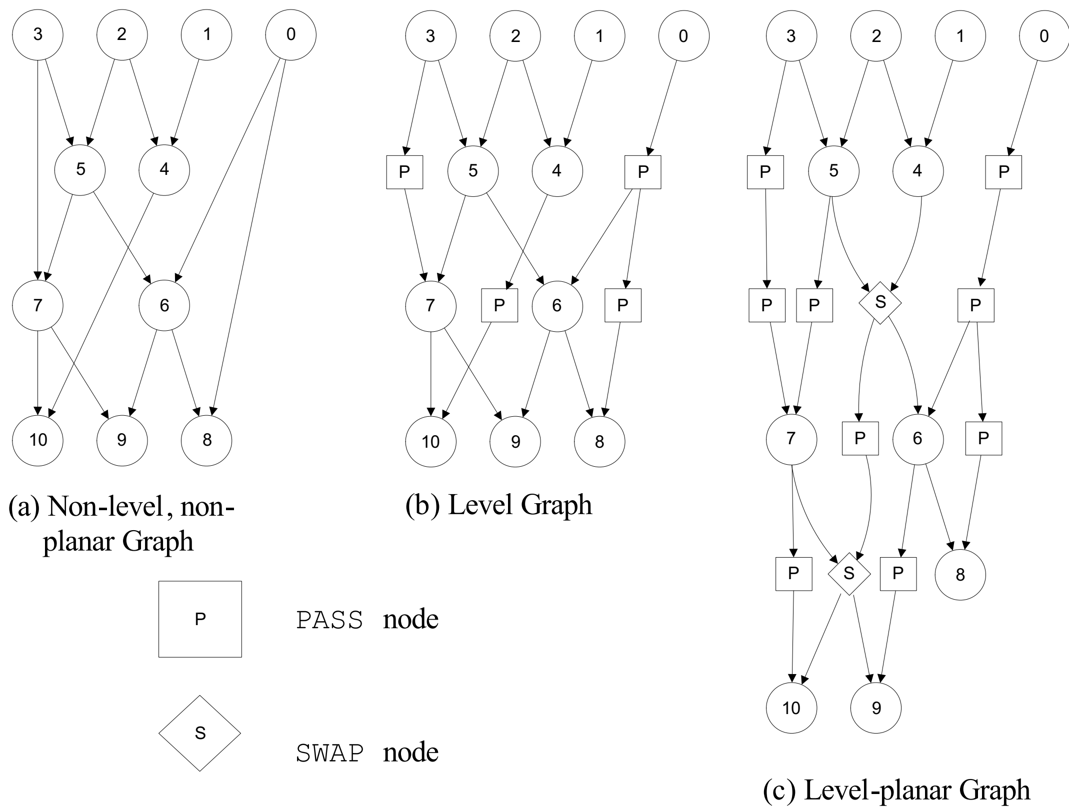
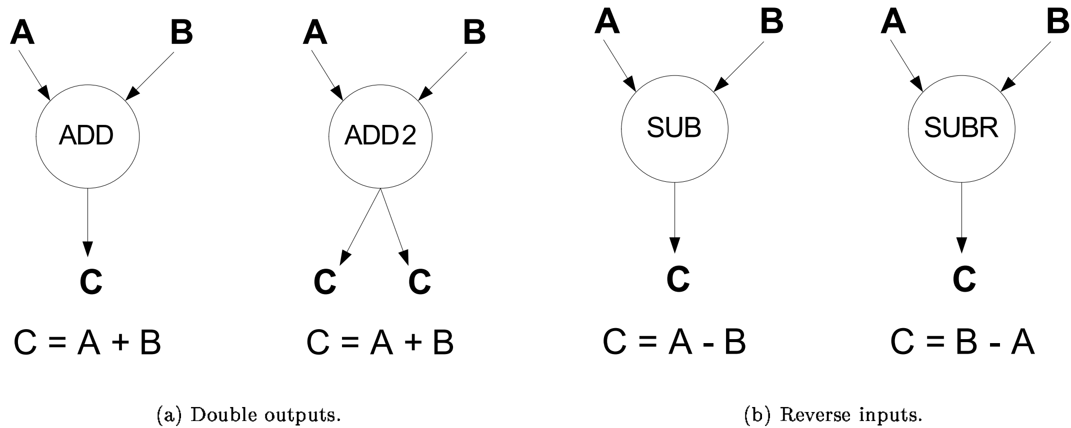


Figure 4.7: Examples showing the level and planar properties of graphs



(a) Double outputs. (b) Reverse inputs.

Figure 4.8: Queue ISA Instruction Variants

Other problems with the queue ISA come from the limitations placed on the DFG due to the level planarity requirement. One way to simplify meeting this requirement is the allow the ability to reverse the order that operands are used in. If this were not possible, than the DFG would have to restructured to place the operands in the correct order. This is designated by adding an 'R' to the instruction and is is illustrated in Figure 4.8(b). Note that this is only required and implemented for operations with non-commutative operands, like subtraction; hence there is no ADDR (reversed input addition) instruction. An instruction can have both reversed inputs and doubled outputs. The ISA reference in Appendix B shows which instructions can have their inputs reversed. The select instruction has three inputs, which means that there are six possible input orderings. This is implemented by having three different versions of the select instruction in the queue ISA, each of which can be reversed. All of the possible select operations are shown shown in Figure 4.9.

The final instruction variation is needed only for instructions which have two different outputs; for example, multiplying two 32-bit numbers produces two 32-bit results, one for the high bits and one for the low bits of the result. If the output order is reversed, or exchanged, then an X is added to the instruction. In the SimpleScalar simulation of the Queue ISA, these instructions variants ('2','R', and 'X') are implemented as annotations to the original GHAL instructions; however, in actuality, there would be different instructions with different opcodes for each instructions and variant.

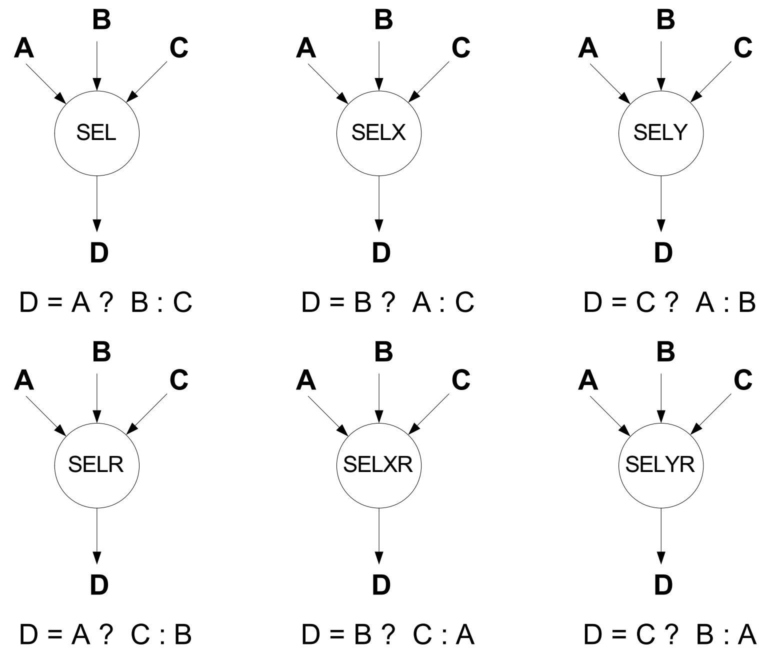


Figure 4.9: Select Instruction Variants

The Queue ISA has relatively simple instruction formats, as shown in Figure 4.10(a). As with all the instruction formats in HASTE, it is assumed that instructions must be some multiple of 8 bits in length. Most queue ISA instructions require one byte to identify the specific instruction. Since the identity of operands is determined implicitly due to the nature of the queue architecture, no fields are needed to specify what the inputs to an instruction are or where the results should go. R-type instructions require no other information than an 8-bit instruction field. Queue ISA instructions with small immediate values, e.g, port number or shift amount are specified in the S-type instruction format. S-type instructions have one byte to identify the instruction and 5 bits to represent the port number or shift amount, with 3 unused bits needed to reach a width that is a multiple of 8 bits. QISA instructions with numeric immediate values use the I-type instruction format, with one byte for the instruction and 16 bits for the immediate value. It would have been possible to allow larger immediates, perhaps with the addition of new versions of immediate instructions that specify the immediate width, but this was not done due to limited utility (few immediates in the kernel need more than the standard 16 bits) and to maintain compatibility with PISA, RISA, and RRISA, all of which use a 16-bit immediate width. The

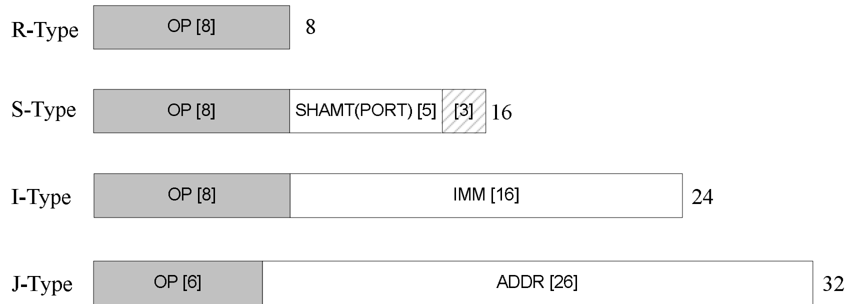
final QISA instruction format is the J-type, used for jump instructions. To maintain compatibility with the PISA compiler, a 26-bit target address is used. To avoid requiring an extra byte for J-format instructions, the J format instructions can be specified using only the upper 6 bits of the instruction field. This 6-bit field plus the 26-bit address field requires only 32 bits, rather than the 34 bits needed if the full 8-bit instruction field was used. The 34 bits instruction would need to be stored in 5 bytes, rather than just 4.

4.4 Register ISA (RISA)

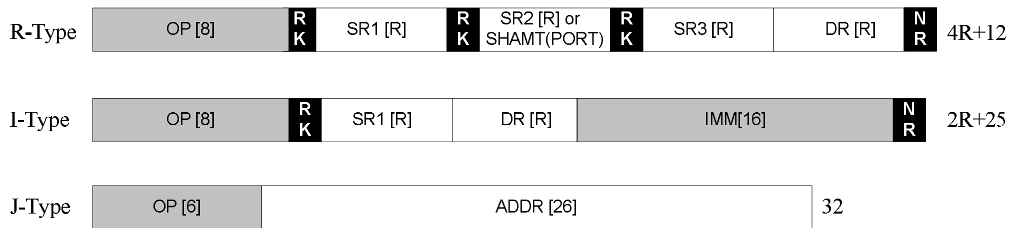
Given some of the difficulties inherent in using the queue ISA, using a more traditional register ISA would seem to be a desirable alternative. Using a register ISA makes the construction of the SPU easy, since a conventional or nearly conventional processor architecture can be used. However, extracting parallelism and constructing a spatial configuration from code produced by a standard compiler is an extremely difficult, if not impossible, task. It is necessary to produce executables in this register ISA that allow for the creation of a configuration that can be implemented on the fabric that they will be run on. Thus the compiler must know the base parameters of the fabric and the algorithms used by the CTE, and modify the DFG accordingly. It does this by producing a spatial configuration as part of the compilation pass. Then register assignments are made and the sequential code can be written out. This will be covered in more detail in Chapter 7.

Another problem is that the task of the CTE is more complicated than for the queue ISA. Since operands are identified only by register numbers, the CTE must keep track of the location of these values and route the fabric accordingly. There is no correspondence between the architected register file number used in the SPU and the register location in the fabric. To completely specify the location of a register in the fabric, both the column of the register file and the number of the register file entry must be specified. Figure 4.11 shows a sequence of three instructions, the first of which is a NOOP placed there just for spacing. Given an SPU with 8 registers, registers would be accessed as shown in the middle portion of the figure. First, for the `SUB` instruction registers 1 and 5 are read. The result is placed in the register file in register 4. Next, registers 6 and 3 are read for the ADD instruction, with its result placed in register 7. Both the source and destination of operands is given explicitly for the SPU. For the RCF, however, these register numbers have

(a) Queue ISA



(b) Register File ISA



(c) Relative Register ISA

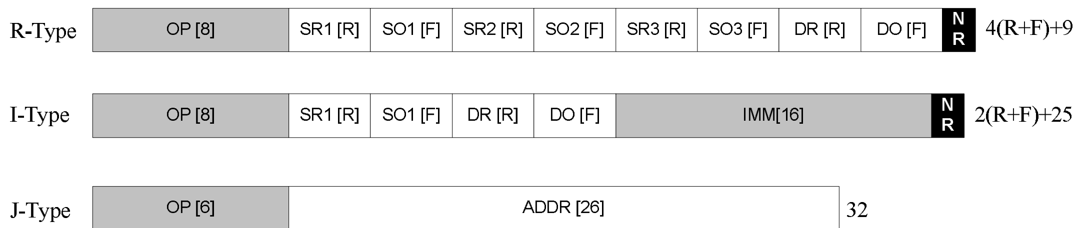


Figure 4.10: HASTE ISA Instruction Formats

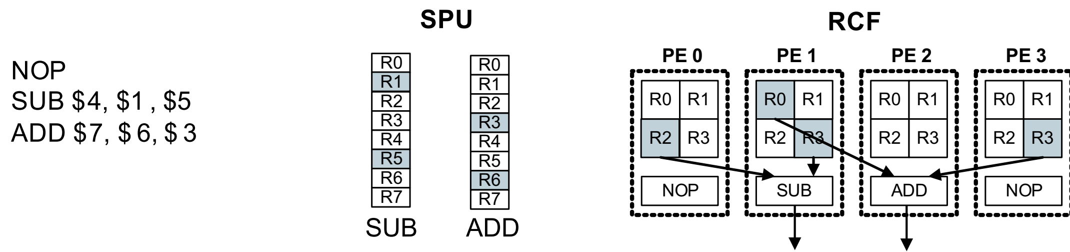


Figure 4.11: RISA Register Addressing

no relation to their locations in the fabric. The CTE uses the register designations to keep track of the dependences, but must assign and track fabric registers itself. In order to allow the CTE to do this, result operands are always placed in the same column they are generated in. The first open register in the register file in that column is used to store the result. The CTE keeps track of the column and register number in the RCF that the SPU register is associated with for use when operations need that operand. In order to free up registers when they are no longer needed, the last use of a particular value must be marked as such. The HASTE tools take care of this determination and indicates it using an exclamation mark after the register number of a source operand. A bit in the corresponding instruction field is set if this is the last use of that value. This bit is labeled RK in 4.10(b). Note that there will typically be more registers in a single row of the fabric than there are registers in the SPU. CTE operation for the register ISA is covered in Chapter 6.

Another modification is necessary to allow use of a fabric with limited interconnect. In the fabric, if an operation needs operands that are in columns too far apart to be read in any single PE, than one or both operands must be moved. A move instruction can be implemented to read operands from one column and write to another. The CTE interprets a move instruction as a routing directive that reads a value from a register in one column and writes it to a register in a different column. The move instruction would be treated as a NOOP by the sequential processor, since the value being moved would still be referred to by the same register number and be in the same location in the SPU's register file. Rather than introduce a new instruction, the PISA pseudo operation `MOVE` is promoted to be a true instruction. A move from one register to the same register is interpreted by the CTE as move from the column the operand is in to the column the MOVE instruction is in. Other than these minor changes, and the mapping and

transformations performed by the tools, RISA is no different than GHAL.

The Register ISA formats are shown in Figure 4.10(b). There are three instruction formats, R-type, I-type, and J-type. R-type and I-type instruction formats have an additional single bit field labeled NR. This bit indicates whether the current instruction is the last one on the current row. Note that the J-type instruction does not have this field, since jump instructions are never part of kernels and rows have no meaning in sequential code. The NR bit is ignored in sequential code for instructions using the R-type and I-type formats. The remaining fields are very conventional; the OP field designates the specific instruction. As in QISA, this is an 8-bit field for all instructions except those that use the J-format, in which case it is a 6-bit field. This is done only for compatibility with QISA. In both the I- and R-type instructions, the RS1, RS2, and RS3 fields are source register designators; note the SEL instructions is the only one that actually uses all three source registers. The RD field is the destination register. The one exception to this is the full-width multiply instruction, which uses the RS1 and RS2 fields to designate the source values and the RD and RS3 fields to designate the result registers for the low and high bits of the result, respectively. Note that the widths of all of the register fields are labeled R, and the format widths are given as functions of R. R in this case designates the number of bits needed to select a specific register. As such, it depends on the number of register in the SPU, since that is what these register designators refer to (the CTE does the transformation to actual RCF registers). So $R = \text{ceil}[\log_2(\#of\ SPU\ registers)]$. The width of the largest format for any given value of R is used as the width for all formats. For example, with 32 registers, R $=$ 5, so the R-type instructions have width $4{\cdot}5{+}\ 12 = 32$ bits, I-type instructions have width $2{\cdot}5{+}\ 25 =$ 35 bits, which is rounded up to 40 bits, and J-type instructions always have a width of 32 bits, so all instructions are 40 bits wide. 40 bits are sufficient for up to 128 SPU registers. It is important to remember that the number of registers in each RCF row is not limited by the number of SPU registers, nor does the number of SPU registers determine how many different live register values can be in the fabric at any row. Any sequential kernel that can be implemented within the limits of available SPU registers can be implemented on any RCF fabric with sufficient registers.

4.5 Relative Register ISA (RRISA)

Determining the routing for the PEs is difficult for a register file ISA because the only available information about operands is the SPU register file number. Another problem is that the size of the application is limited by the size of the architected register file. Rather than require the CTE to determine which register file and register entry to read and write operands to and from, it would be nice if this information was present in the executable. An efficient way to do this is shown in Figure 4.12. In Figure 4.12(a) are two example instructions. Each register is addressed by a register number and an offset. The meaning of this in the RCF is straightforward. The register number refers to a register in the register file of each PE in the fabric. The offset refers to the column location of the register file in the fabric, relative to the current column. So in this example, the SUB instructions first operand is located in register 2 at offset -1, meaning the column one to the left of where the instruction is being implemented, and the second operand is in register 3 at offset 0, meaning the current PE. Figure 4.12(b) shows the locations of operands for the example instructions. Note that RRISA allows results to be placed in a column different than the one that they were calculated in, while QISA and RISA require that results are always placed in the same column as the one they were calculated in. As in QISA, feedback cannot be expressed directly with RRISA, since results can only be placed in a register file in the next row. RRISA supports the ZACC instructions as does QISA.

Since register addressing refers specifically to the RCF in RRISA, an equivalent method for addressing the correct registers in the SPU is needed. One way to implement this ISA would require the SPU to use a register file with a sliding window. The column location of each instruction would need to be tracked in the SPU and the register file window indexed accordingly, as shown in Figure 4.12(c). The size of the window is dependent on the read and write spans and the size of each register file in the fabric. Since the movement of the register file window is predictable, register values that aren't needed can be cached, allowing for a very large register file and thus allowing for very wide fabrics. Some kernels will require modification of their DFG to allow for operands to be read from locations outside the register window; this can be done by using MOVE instructions, as discussed for the register ISA. These move instructions may be needed for all portions of the application, not just the kernels. New row indicators are also needed just as for the register file ISA and these will also be needed for the whole application,

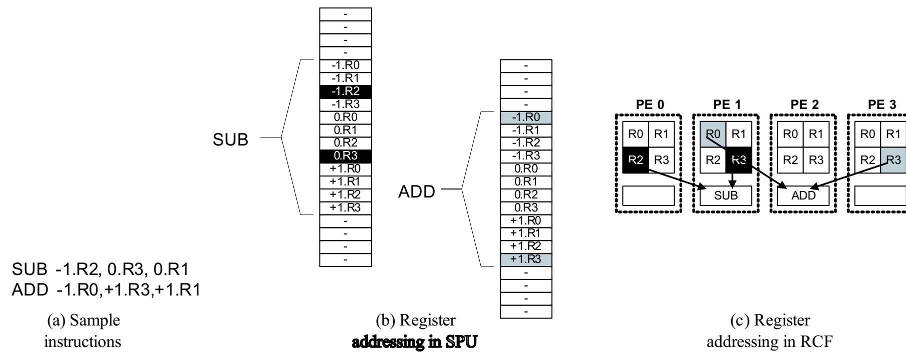


Figure 4.12: Relative Register ISA Addressing.

since it is necessary to properly index the register window. Except for the register file, a mostly standard RISC CPU can be used for this ISA. Alternatively, translation could be performed from relative register numbers to fixed conventional register numbers using a technique analogous to conventional register renaming. However, a sliding window register file model will be assumed for the RRISA SPU, more details of which will be covered in Chapter 6.

The Relative Register ISA instruction formats are shown in Figure 4.10(c). These formats appear quite different than those used for RISA. Since, as discussed, relative register addresses consist of a register number, corresponding to a register number in an RCF register file, and an offset, corresponding to a column offset relative to the instructions location, the operands are thus designated by a two number designator R,O, with R the register number and O the offset. In Figure 4.10(c), the source operand pairs are thus SR1 and SO1; SR2 and SO2; and SR3 and SO3. The destination register is designated by DR and DO. The other difference between the R2ISA and RISA formats is that a register kill bit is not needed for proper CTE operation. If an RCF location holds a value that is no longer needed, it can be overwritten by another value explicitly. The width of these fields is determined by the range of possible offsets, with $F =$ ceil$[\log_2(\max(RC, WC))]$, where RC is the number of columns that can be read from the current column, and WC is the number of columns that can be written from the current column; and the number of registers in each PE register file, with $R =$ ceil$[\log_2(\#of\ RCF\ registers\ per\ PE)]$. So as with RISA, instruction width depends the specific fabric architecture targeted. Since only fabrics with read connectivity $RC =$ write connectivityWC are being considered in the current HASTE implementation, the max(RC,WC) term in the formula for F can be replaced with just

RC. For example, with a fabric with $RC = 3$ (and thus $WC = 3$), and number of registers NR = 4 has F = 2 and R = 2, so each register designator takes 4 bits total. The width of the R-type instructions would then be 25 bits and I-type 33 bits. As with the other ISAs rounding is done to the nearest byte, giving a 40-bit instruction word. A widely connected fabric with large register files, for instance with $RC = 17$ and NR = 10, would have F = 5 and R = 4, requiring 9 bits per register designator. This means 45 bits are needed for R-type instructions and thus the instruction word would be 48 bits wide.

4.6 Observations

In this chapter the requirements of HASTE ISAs and three different ISA types have been shown, each with their own advantages and disadvantages. All three provide enough information for the CTE to recreate a valid fabric configuration for the kernel. The details of this process are covered in Chapter 6. Before covering this material, however, specific details of the RCF must be introduced so that the exact task performed by the CTE can be determined. The next chapter will cover the RCF model and the parameters that define specific fabrics. While there are many possible RCF fabrics, three styles of RCF will be primarily considered, each of which corresponds to one of the ISAs introduced in this chapter.

Chapter 5

RCF Architecture

As discussed in Chapter 3, the HASTE architecture consists of three main computational components, the sequential processing unit (SPU), the code transformation engine (CTE), and the reconfigurable computing fabric (RCF), plus the memory access unit (MAU). While these components will exist in any HASTE system and will perform the same functions, the actual characteristics of each component are not fixed and may vary substantially from implementation to implementation. The characteristics of each of the components are dependent largely on the ISA used and the other components in the system; all four components and the ISA must be designed as a unit in order to create a functioning system. In order to make comparisons between different HASTE implementations, parameterizable models of the components are needed. These models need to be broad enough to allow for design space exploration across a large range of possible system implementations, but not so broad as to make comparison difficult.

In this section a parameterized model of the RCF is presented. In addition to the parameters needed to describe the characteristics of all of the parts that make up the RCF, the signals needed to configure each RCF fabric have been determined, so that the configuration signals that CTE must generate are known. In addition to the parameterized architectural models described in this chapter, physical models for estimating area and performance are discussed in Chapter 10. Models of the CTE and the SPU are covered in Chapter 6.

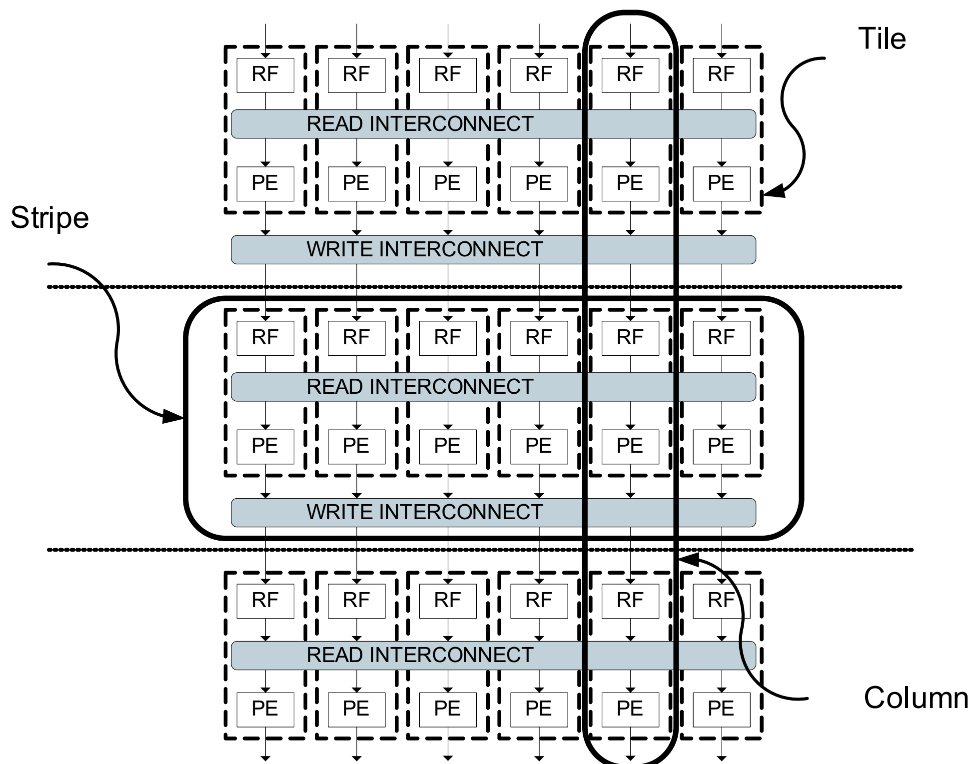


Figure 5.1: RCF Model

5.1 Global RCF Model

The RCF model assumes that the fabric consists of coarse grained processing elements, register files, and programmable interconnect, arranged to form a pipelined datapath, as shown in Figure 5.1. Each pipeline stage, or stripe, is composed of several tiles and two programmable interconnect networks. Each tile contains a programmable processing element (PE), and a register file (RF). One interconnect network allows the PEs in one stripe to read values from the RFs in the same stripe. The other interconnect network allows the PEs in one stripe to write values into the RFs in the next stripe. All of these components have their own set of parameters, and these parameters, in addition to a set of global parameters, describe an individual RCF instantiation. Each of these parameter sets will be discussed in this chapter. All of the RCF parameters are shown in Table 5.1.

Customized datapaths for each specific application kernel are implemented by programming the PEs, RFs, and interconnect. A single wide configuration word specifies all of the necessary programming for each tile, including the PE, the RF, and those portions of both the read and write interconnect networks associated with that tile. A set of configuration words, one for each processing element in a stripe, provides all of the necessary information to configure that stripe. Each configuration word is composed of several fields, with each field corresponding to a particular hardware structure. The hardware model specifies the specific fields that make up the configuration word for each processing element. The number and types of fields required will vary between different RCF classes.

5.1.1 Global Parameters

There are several global parameters that either apply to every component of the RCF or which apply to the structure of the RCF as a whole. The first and most important is the datapath width, B. This represents the width in bits of all data in the RCF. This means that all data interconnect, data storage, and data processing in the RCF is designed to work with data words of Bbits wide. It is possible to design HASTE implementations such that operations on data wider than B can be implemented automatically by the CTE; in others, all data and operations wider than B are handled by decomposition into narrower widths in the DFG before the kernel assembly code is produced. Most of this thesis assumes that all data and operations in the DFG

Table 5.1: RCF Model Parameters

Symbol	Name	Description
		Global Parameters
B	Datapath width	The width in bits of all data and datapaths in fabric.
F_w	Fabric width	The width of the fabric in tiles/columns.
F_d	Fabric depth	The depth of the fabric in tiles/stripes.
		Register File Parameters
M_W	# of write muxes	Number of write muxes per register file.
M_R	# of read muxes	Number of read muxes per register file.
I_{MW}	Write mux inputs	Number of inputs to each write mux.
I_{MR}	Read mux inputs	Number of inputs to each read mux.
r_s	# of static registers	Number of static registers in each register file.
r_p	# of pass registers	Number of static registers in each register file.
r	# of registers per file	Total number of registers in each register file.
		Processing Element Parameters
P	# of input operands	Maximum number of input operands for any instruction.
T	# of outputs	Maximum number of outputs produced by any instruction.
M_O	# of operand muxes	Number of operand muxes in each processing element.
I_{MO}	Operand mux inputs	Number of inputs to each operand mux.
		Interconnect Parameters
R_C	Read connectivity	Number of different tiles that the current tile can read from.
W_C	Write connectivity	Number of different tiles that the current tile can write to.

are B bits wide or narrower; however, the composition of wider operations than B is mentioned in Chapter 12. The second global parameter is F_w, the width of the fabric in columns, where a column is simply a horizontal position that tiles can occupy, or equivalently, the set all of tiles in a fabric at the same horizontal position, as can be seen in Figure 5.1. For purposes of the thesis it is assumed that F_w is always wide enough to hold the maximum width of any kernel to be implemented on it. The final global parameter is F_d, the depth of the fabric in pipeline stages. As with the width, it is assumed that the physical depth of the RCF is sufficient for any kernel to be implemented. Some justification for these two assumptions is given in Chapter 12.

5.2 Register File

Each register file in the RCF is identical to every other RCF register file. Each register file consists of a number of registers, each of which is B bits wide. These registers can be of two types, either static registers or pass registers. A static register is simply a conventional register that can only be read using the register file's read ports and which can only be written to using

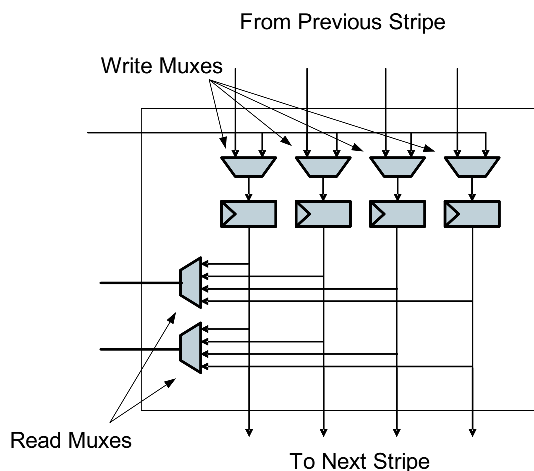


Figure 5.2: Pass Register File

the register file's write ports. A pass register is one in which all register values which are not overwritten in the current stripe are passed to the corresponding registers in the register file in the same column in the next stripe. A pass register file is shown in Figure 5.2. Since all values in the pass register file are accessed each clock cycle, as they are transferred to the next register file in the pipeline every clock cycle, it is not necessary to consider read ports as would be done for a conventional register file. Instead, it is only necessary to have sufficient read multiplexers and other interconnect resources to read as many register values as needed by the read interconnect value, as will be covered in the next section. Any of the pass register values can be read by processing elements in the current stripe, by selecting the value to be read using one of the read multiplexers. Similarly, it can be assumed that it is always possible to write as many new values into the pass registers as might be needed according to the write connectivity, since all values are being written anyways. By default, each pass register is overwritten by the corresponding register from the previous stripe. A multiplexer at the input of each register allows the selection of a different value to be written into the register; these values come from functional units in the previous stripe. There are M_W write multiplexers and M_R read registers in each register file. Each write multiplexer has I_{MW} inputs, one for each PE that can write to it, plus one for the value passed from the previous stripe, and each read multiplexer has I_{MR} inputs, one for each register.

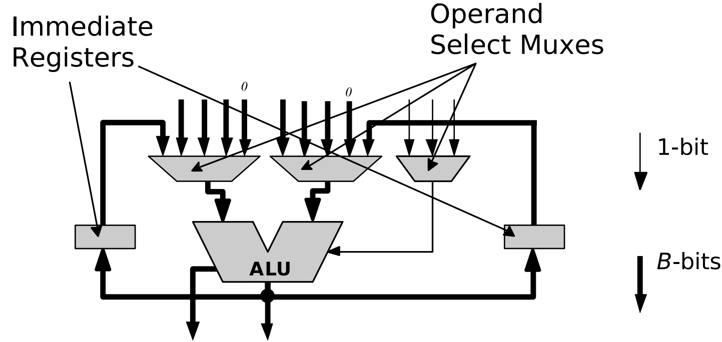


Figure 5.3: Processing Element

The number of static registers, r_s, is another parameter for the register file, as is the number of pass registers, r_p. The total number of registers of both types is r, where $r = r_s + r_p$. While mixed register files, register files with both $r_s > 0$ and $r_p > 0$, can be useful in some circumstances and for some ISAs, the remainder of this thesis will only discuss those with only one or the other type of register, as mixed register files are not needed to implement any of the different ISAs being studied. Further, it will be shown that register files with static registers will only require a single register, while those with pass registers will need at least two. Therefore, it is only necessary to specify r, with $r = 1 \Rightarrow (r_s = 1, r_p = 0)$ and $r > 1 \Rightarrow (r_s = 0, r_p > 1)$. In order to configure the pass register file, the select values for each of the read and write muxes must be set, the exact number of which depends on the specific interconnect parameters in use. A static register file with a single register, the only kind that will be considered, may have a single write mux that needs to be configured, but no read muxes; this will be shown more clearly in Section 5.5.

5.3 Processing Elements

The main processing element parameters are B, the processor element bitwidth, which is the same as the global bitwidth parameter; the number of input operands, P; and the number of outputs, T. For the purposes of this thesis only PEs with $P = 3$ will be considered. The two primary inputs are both B-bits wide; the third input is only used for the select operation, which

has a boolean input for the third input. Thus only a single-bit input is needed for the third input. This single bit input is connected to the LSB of the B-bit wide values used elsewhere in the connect. In most cases it is only necessary to consider PEs with T=1, although it is necessary to consider PEs with T=2 in two instances. The first is if there is a full-width multiplier in the ALU, in which case two outputs are needed, one for the B low bits of the result and one for the B high bits of the result. The second is for the queue ISA that was introduced in the chapter, which requires an instruction to swap the order of two inputs and thus requires two outputs.

The processing element is shown in Figure 5.3. There are three operand muxes, one for each input, so the number of operand muxes $M_O = P = 3$. These select where each input operand is being read from, either a register file, an immediate value, or a constant zero. The number of inputs to the operand multiplexers, I_{MO}, depends on the number of register files each PE can read from, designated R_C. The PE can read at most two values from any column other than the current one, and three from the current column. The read mux in the register file determines the specific register being read from the register file selected by the operand mux. The third operand mux has fewer inputs because a select with an immediate select value or constant zero value isn't necessary.

All PEs in a given RCF are identical and all can execute any of the operations that are specified as being legal for the RCF. In addition to the typical arithmetic operations, the ALU also contains logic for getting data from and sending data to the MAU. Three different ALU types are considered in this thesis, with the only difference being the kind of multiplier, if any. The three choices are no multiplier, a full-width multiplier, or a half-width multiplier. As mentioned, the full-width multiplier multiplies two B-bit values and produces a $2 \cdot B$-bit result, which is expressed as a B-bit low result and a B-bit high result. The half-width multiplier multiplies two $B/2$-bit values and produces a B-bit result. The inputs to the multiplier are taken from the low $B/2$bits of each B-bit input value. To configure the PE, it is necessary to specify the select values for each input mux, specify an operation for the ALU to perform, and specify a value for the immediate register(s) if needed. The immediate registers are full width, because an immediate value may also be a live-in value instead of a true compile time immediate.

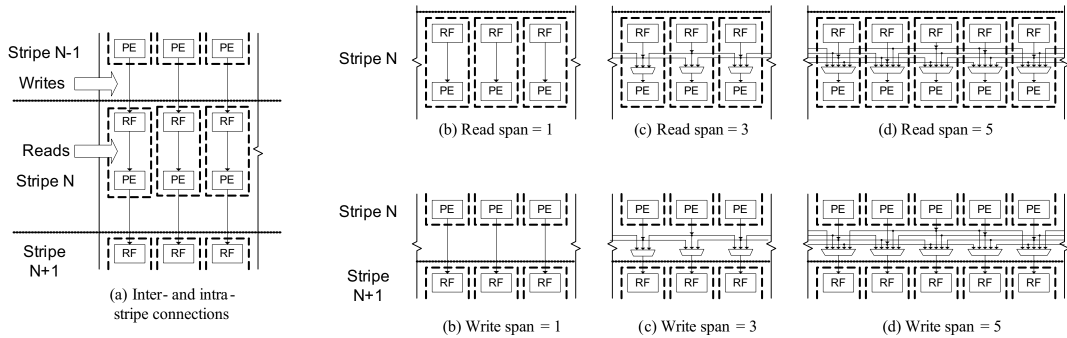


Figure 5.4: Interconnect Model

5.4 Interconnect

A somewhat simplified interconnect model is shown in Figure 5.4 and as was shown in Figure 5.1. There are two sets of interconnect resources, as shown in sub-figure (a). One set connects the register files in a stripe with the processing elements in the same stripe. These resources perform reads of values in the register files. As was shown in the previous section, these read resources may consist of a set multiplexers in both the register file, if $r > 1$, which select the the specific register to be read by each PE input, and a set of multiplexers in the PE, which select which register file each input reads from. Each register file can be read by R_C PEs, where R_C is referred to as the read span. R_C is always odd and the PEs that are connected are always symmetrical, so $R_C = 3$ means the current PE and the PEs to the immediate right and left can read the register file; $R_C = 5$ means that two PEs on either side and the current PE can all read the register file, and so on. In Figure 5.4 can be seen examples of read spans equal to 1,3, and 5 in (b), (c), and (d), respectively. Each PE can read three different values from the register file in its own column, but only only two from other register files, one of which must be a single bit value.

The second set of interconnect resources connect the processing elements in a stripe with the register files in the next stripe. These wires perform writes of results produced by the PE into register files. The architecture has an associated write span, W_C, similar to the read span, which determines how many different register files each functional unit can write to. In Figure 5.4, can be seen examples of write spans equal to 1, 3, and 5 in (e), (f), and (g), respectively. If the write span is larger than one, a write mux is required in the register file for each register, as was shown

Table 5.2: RCF Classes

	r_s	r_p	T	W_C	R_C	M_W	M_R	M_O	I_{MW}	I_{MR}	I_{MO}
Static Register Fabric	1	0	2	1	RC	0	0	3	n/a	n/a	2·RC+4
Asymmetric Pass Register Fabric	0	NR	1 {or 2}	1	RC	NR	RC	3	2{3}	NR	2·RC+5
Symmetric Pass Register Fabric	0	NR	1 {or 2}	RC	RC	NR	RC	3	RC+1 {2·RC+1}	NR	2·RC+5

Note: Values in curly brackets are for **T** = 2 case for pass register fabrics.

in Figure 5.2. Note that only two types of interconnects will be considered; interconnects with $W_C = 1$ and $R_C > 1$ and interconnects with $W_C = R_C > 1$. These two classes encompass the requirements of the ISAs that will be explored and limiting the interconnect possibilities in this way reduces the search space to a manageable extent.

5.5 RCF Classes

There are three main classes of RCF that will be examined in this thesis. The parameters of each are summarized in Table 5.2. While the overall RCF model can model a wider range of fabrics, a large part of the design space and almost all of that portion of the design space that is relevant to the ISAs being investigated, can be covered using just these three RCF classes. For each class, only two parameters, NR and RC, are needed where NR is the number of registers and RC is the connectivity. From these two parameters all of the other parameters can be derived. Examples of each class are shown in Figures 5.5, 5.6, and 5.7. For each RCF class, there are a set of configuration fields that will define the functionality of the RCF and which must be set by the CTE. These will be described as each RCF class is covered below.

5.5.1 Static Register Fabric

A portion of a static register fabric with a connectivity of 3 is shown in Figure 5.5. This fabric would be described as an NR1_RC3 fabric, which is enough information to determine all of the necessary parameters from Table 5.2. There is no real write interconnect, since $W_C = 1$ and

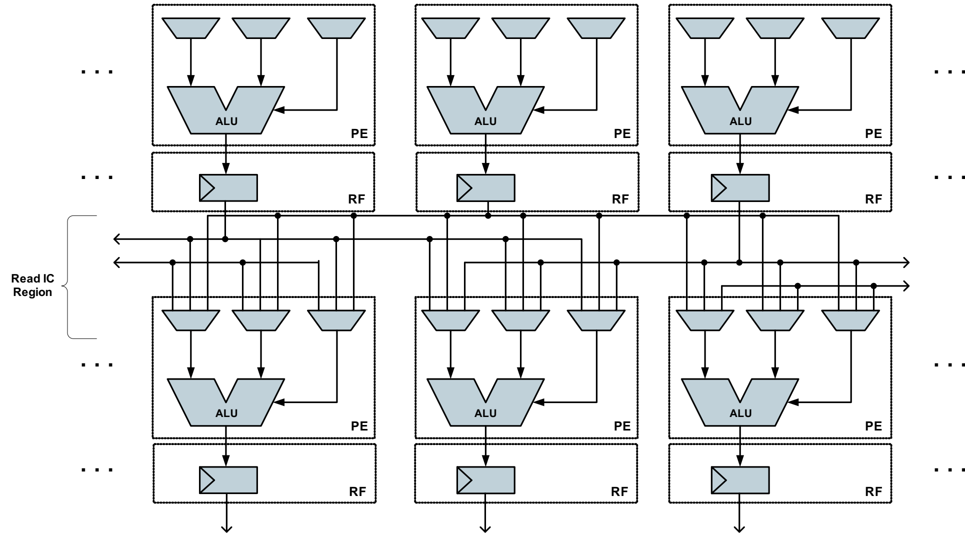


Figure 5.5: Static Register Fabric

there are no pass registers. For clarity, only one PE output is shown, in actuality there are always two for this class of fabric, and the immediate registers are not shown. The static register fabric is designed for use with QISA. No pass registers are needed for QISA, since the graph must be level, and PASS operations are implemented in tiles, so every variable passed from one level to another lower level has storage explicitly designated as that in the tiles consumed by PASS operations. Assuming only half-width multipliers, only a single register is then needed per tile, assuming the SWAP operation uses the feedback/immediate registers in the ALU as shown in Figure 5.3. Since the queue ISA has no way to specify where results should be stored, they are stored in the same column they are produced in, which is equivalent to saying that $W_C = 1$. The only configuration fields needed for this fabric are a select for each operand mux, to select which register it is reading from, a function select field to tell the ALU what to do, and possibly one or two immediate values. These fields need to be set for each tile (PE/RF pair) The configuration fields needed for each tile are are shown in Table 5.3, with the field size given in brackets (note that the symbol $\lceil x \rceil$ indicates $ceiling(x)$).

Table 5.3: Static Register Fabric Configuration Fields

FUNC[6]	Function select for ALU.
AIMM1[32]	Immediate value 1 for ALU.
AIMM2[32]	Immediate value 2 for ALU.
OS1SEL[$\lceil log_2(2 \cdot RC + 4) \rceil$]	Operand select for ALU input 1 - determines which register is being read.
OS2SEL[$\lceil log_2(2 \cdot RC + 4) \rceil$]	Operand select for ALU input 2 - determines which register is being read.
OS3SEL[$\lceil log_2(2 \cdot RC + 2) \rceil$]	Operand select for ALU input 3 - determines which register is being read.

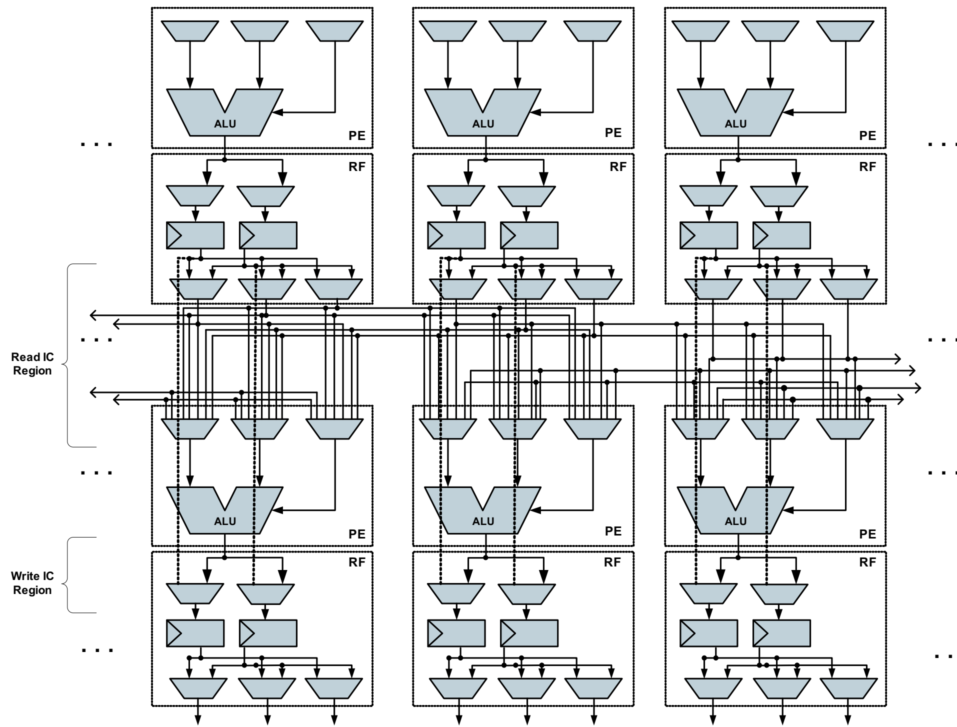


Figure 5.6: Asymmetric Pass Register Fabric

Table 5.4: Asymmetric Pass Register Fabric Configuration Fields

Field	Description
FUNC[6]	Function select for ALU.
AIMM1[32]	Immediate value 1 for ALU.
AIMM2[32]	Immediate value 2 for ALU.
OS1SEL[$\lceil log_2(2 \cdot RC + 5) \rceil$]	Operand select for ALU input 1.
OS2SEL[$\lceil log_2(2 \cdot RC + 5) \rceil$]	Operand select for ALU input 2.
OS3SEL[$\lceil log_2(2 \cdot RC + 3) \rceil$]	Operand select for ALU input 3.
WSELn[1]	WSELn selects whether the nth register is written by an ALU result or by the previous passed value.
RSELn[$\lceil log_2(R) \rceil$]	RSELn selects which register is being read for the nth register file being read from.

5.5.2 Asymmetric Pass Register Fabric

A portion of an asymmetric pass register fabric with a read connectivity of three, a write connectivity of one, and two registers per tile is shown in Figure 5.6. This is called an asymmetric fabric since $W_C \neq R_C$. This particular fabric would be designated as an NR2_WC1_RC3 fabric. Even though $W_C = 1$, write interconnect is still needed, since there are pass registers, and a mux is needed for each register to determine whether the previous register value is passed in, or if the register is written with the result of the previous ALU in the same column. Again, for clarity, the ALU is shown with a single output. A double output is only necessary if a full-width multiplier is used. The asymmetric pass register fabric is designed for use with RISA. Since RISA DFGs do not need to be level, pass registers are needed to make operands available to more than just the one stripe immediately after the one in which they are produced. Since RISA has no way to specify where results go, they always go to the register file in the same column and in the next row, so the write connect value $W_C = 1$ is sufficient. The same function and operand select fields are needed as were needed for the static register file, plus a new set of fields to determine whether or not a register is being written by the ALU in its column. In addition, for each ALU which could be reading from the file a field is needed to determine the specific register being read. The configuration fields for each tile are are shown in Table 5.4.

5.5.3 Symmetric Pass Register Fabric

A portion of a symmetric pass register fabric with a connectivity of three and two registers per tile is shown in Figure 5.6. This is called a symmetric fabric, since $W_C = R_C$. This particular fabric would be designated as an NR2_WC3_RC3 fabric. It has a more complicated write interconnect,

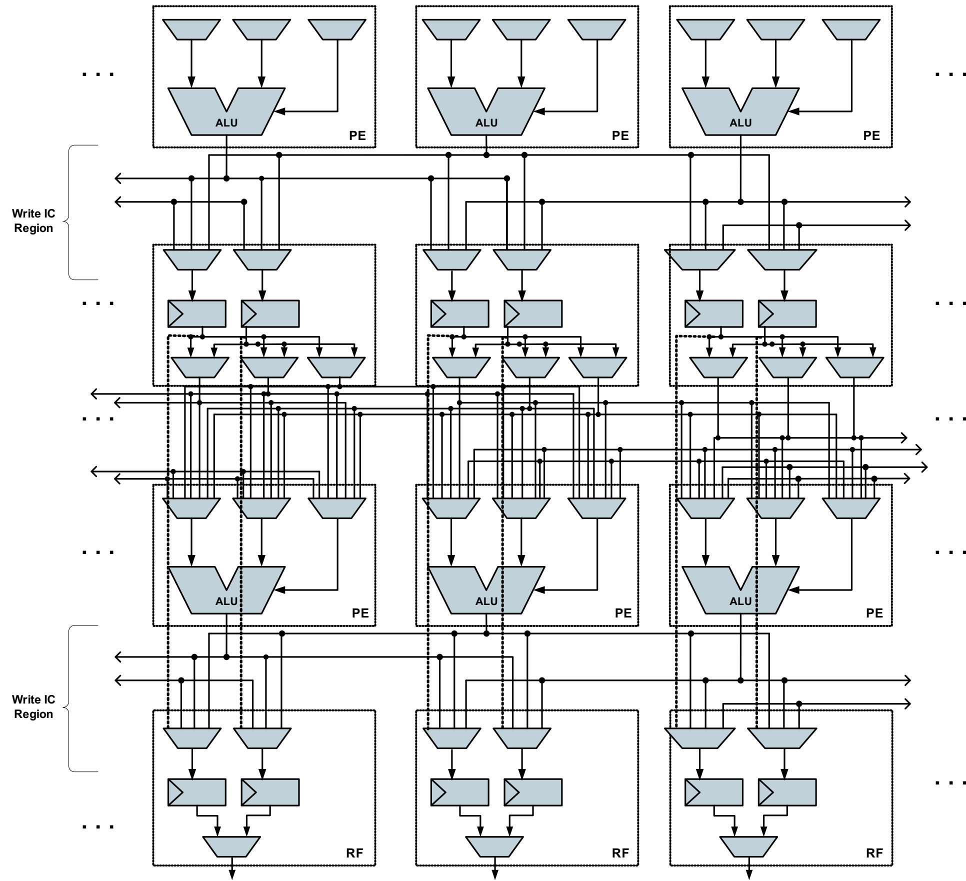


Figure 5.7: Symmetric Pass Register Fabric

Table 5.5: Symmetric Pass Register Fabric Configuration Fields

FUNC[6]	Function select for ALU.
AIMM1[32]	Immediate value 1 for ALU.
AIMM2[32]	Immediate value 2 for ALU.
OS1SEL[$\lceil log_2(2 \cdot RC + 5)\rceil$]	Operand select for ALU input 1.
OS2SEL[$\lceil log_2(2 \cdot RC + 5)\rceil$]	Operand select for ALU input 2.
OS3SEL[$\lceil log_2(2 \cdot RC + 3)\rceil$]	Operand select for ALU input 3.
WSELn[$\lceil log_2(RC + 2)\rceil$]	WSELn selects whether the nth register is written by an ALU result or by the previous passed value.
RSELn[$\lceil log_2(NR)\urcorner$]	RSELn selects which register is being read for the nth register file being read from.

since ALU results can be written into register files in columns other than the one it was produced in. This fabric class was designed for use with RRISA. Since RRISA can specify where to write results, having $W_C > 1$ is needed to make it possible to write to those other columns. The register specification format for RRISA makes the symmetric read and write connectivity desirable so that the same field size can be used for registers being read and registers being written. Like RISA, RRISA doesn't require level graphs, so it needs to have pass registers. No new fields are needed, there are just more inputs to choose from for the write muxes. Instead of just choosing between the default passed register value or the ALU result (or one of the ALU results if $T = 2$) in the same column, ALU results from other column may also need to be selected. The configuration field list is the same as for the asymmetric pass register fabric, with just a change in widths. The width required depends on the value of T, which is 1, unless a full-width multiplier is being used. The configuration fields for the fabric are shown in Table 5.5.

5.6 Observations

A general fabric model was introduced in this chapter. From this model, three parameterizable fabric classes were derived, one for each of the ISAs covered previously. Note that there are two main types of register files; static register files and pass register files. The fabric best suited for QISA is the single register static register fabric. The fabrics best suited for RISA and RRISA are both pass register fabrics; they differ only in that the RISA fabric requires results to be written to the same column they are computed in and RRISA allows results to be written to different columns, since this is possible in the ISA as well. The asymmetric pass register fabric best suited

for RISA and the symmetric pass register fabric best suited for RRISA vary only in their write interconnect.

All of the fabrics are parameterized. The most important ones to explore are the interconnect distance and the register file size. The register file size is fixed at one for QISA. Having a wider interconnect should allow for better hardware utilization, but may run more slowly and have larger areas, due to the more complicated interconnect. These trade-offs will need to be examined to determine what parameters work best and may vary for different applications.The general operation of the fabric and the configuration fields necessary for each class were also covered. With this information, plus the information covering the ISAs themselves in the previous chapter, operation of the CTE can now be discussed in the next chapter.

Chapter 6

CTE and SPU Operation

The code transformation engine (CTE) and the sequential processing unit (SPU) are closely integrated and will be covered together in this chapter, although the primary focus will be on the more novel component, the CTE. As has been explained in previous chapters, the SPU processes sequential code. When a kernel is encountered, the CTE accepts a stream of sequential instructions from the SPU and produces a set of configuration words for the reconfigurable fabric. It must do this stripe-by-stripe, in the order that the instructions are presented, and it must do so in a reasonable amount of time, ideally within a single iteration of the loop. Since it is required that the architecture be scalable to very large fabrics, CTE designs which require arbitrary access to the entire fabric are not allowable. The CTE must also be feasible to implement in a reasonable amount of hardware. CTE designs meeting these requirements have been developed and will be presented in this section. Due to the differences between ISAs, each ISA requires its own specific CTE design. As each ISA requires its own RCF class, CTE designs are specific to particular RCF classes as well as specific ISAs. An example CTE for each ISA discussed in Chapter 4 will be shown in this chapter, along with a specific RCF implementation suitable for that ISA. However, the CTE algorithms presented here will work for any RCF of the same class, even if they have different parameters than shown in the example.

The general CTE hardware design shown in the lower portion of Figure 6.1 is used for all CTE types; the details of the hardware inside the blocks labeled Issue Unit and Config Station is what actually varies for the different CTEs. The CTE is shown along with the other components of the HASTE system, in order to show how the CTE interacts with the other components. The CTE itself consists of two main portions: an issue unit, which receives the sequential instruction

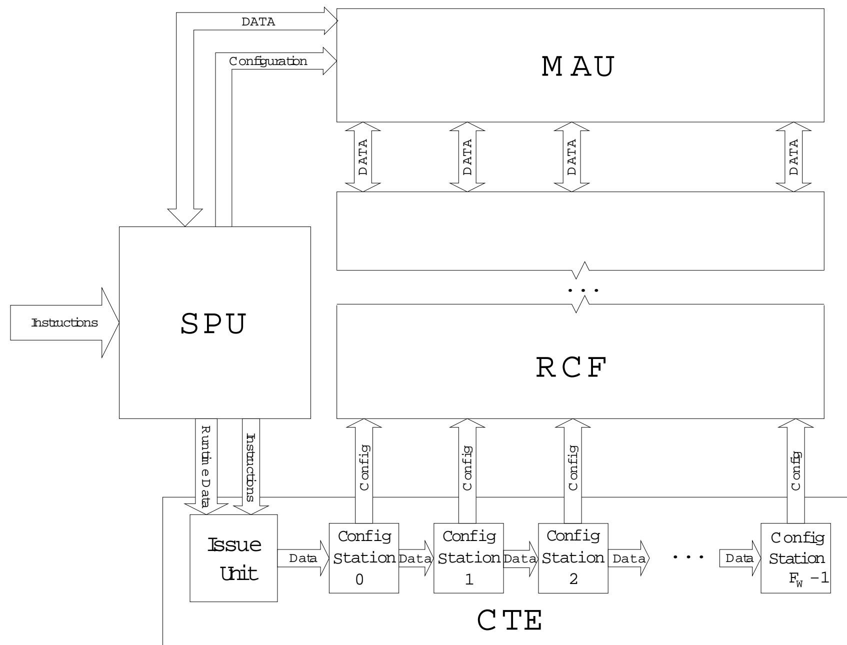


Figure 6.1: CTE and System Model

stream and runtime data from the SPU; and a chain of configuration stations, one for each column of the fabric, where the final configuration word for each tile is determined. The configurations for the tiles are stored in the fabric itself.

While SPU and CTE operation is different for different ISAs, some aspects of their operation are common to all ISAs. The SPU fetches and executes all instructions in the application sequentially. It sends MAU configuration information (instructions setting the base and stride for one or more memory ports) to the MAU when encountered. The MAU can be used with the SPU for non-kernel code, so MAU configuration instructions do not necessarily indicate the presence of a kernel. All kernels use the MAU, however, so kernel execution can be considered to begin with the configuration of the MAU by the SPU. Once the SPU encounters a loop begin instruction, it is then certain that a kernel is being executed and the SPU will then start sending the fetched instructions to the CTE. In addition to the instructions, the SPU also sends the current values of one or more of the input operands for each instruction to the CTE, in case these values are live-in values. The SPU continues to execute instructions as usual. The CTE reads an instruction, along with any input operand values, each clock cycle into the issue unit. A lookup table

is maintained in the issue unit to convert the operation field or fields of the instruction into a suitable value for the FUNC (ALU function) field of the configuration word. The lookup table also contains other information about the instruction needed by the CTE algorithm, such as the number of operands. The issue unit does other processing, which will be covered in detail for each ISA. The issue unit passes the decoded instruction and other data into the first configuration station. Every clock cycle instructions advance from one configuration station to the next. In the configuration station, the final configuration for each tile is determined. When a row is complete, the configuration for the row is passed into the fabric. There are significant differences in operation for the different ISAs, of course. The following sections will provide details of SPU and CTE operation for each ISA type and will show the execution of an example running in the SPU and being processed by the CTE. The configuration produced by the CTE for each ISA will be shown as well.

All of the examples will use the same loop body, which implements the pseudo-code shown in Figure 6.2(a). The original DFG for this loop body is shown in Figure 6.3(a); in this figure, the letters shown below each node producing a value correspond to the values produced by each line of pseudo-code. A version of this loop body DFG mapped to a specific fabric for a QISA implementation is shown in Figure 6.3(b), and a version mapped to a fabric for RISA and RRISA implementations shown in 6.3(c). Note that the loop delimiters are omitted for clarity. Note the NOOP instructions in Figure 6.3(b); these are inserted to allow for routing to a QISA fabric (equivalently, an NR1_RC3 fabric, as described in the previous chapter). The insertion of NOOPs to allow for routing to a specific fabric is discussed in more detail in Chapter 8.

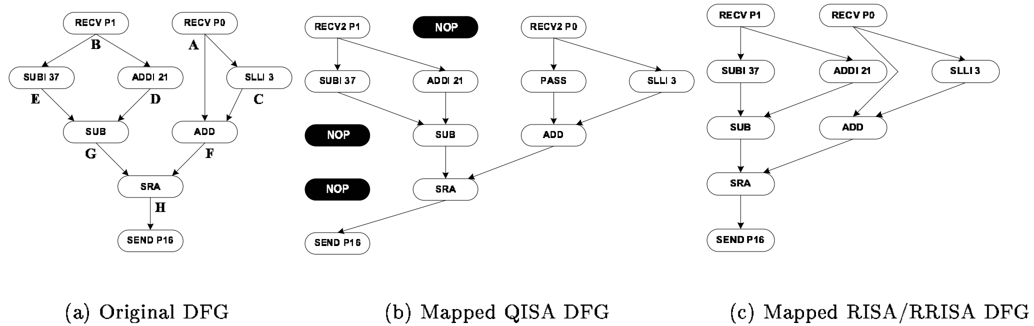
6.1 Queue ISA SPU and CTE Operation

6.1.1 QISA SPU Operation for Example

The QISA code for the example is shown in Figure 6.2(b) and the corresponding DFG is shown in Figure 6.3(b). This code sequence implements the loop body for the pseudo-code shown in Figure 6.2(a). For this example, it is assumed that the queue is empty when the first loop iteration starts. If the queue is not empty when a loop body starts, the value on the queue (only one entry is allowed) is stored in a loop constant register in the SPU and inserted into the queue at the beginning of each successive loop iteration. In the example pseudo-code shown in

(a) Pseudo-code	(b) QISA	(c) RISA	(d) RRISA
A = X[i]	RECV2 P0	RECV $1, P0	RECV :0, P0
B = Y[i]	NOOP	RECV# $2, P1	RECV# :0, P1
C = A << 3	RECV2 P1	SLLI $3, $1, 3	SLLI -1:1, -1:0, 3
D = B + 21	SLLI 3	ADDI $4, $2, 21	ADDI -1:1, -1:0, 21
E = B - 37	PASS	SUBI# $2, $2!, 37	SUBI# :0, :0, 37
F = C + A	ADDI 21	ADD $1, $3!, $1!	ADD :0, :1, :0
G = D - E	SUBI 37	SUB# $2, $4!, $2!	SUB# :0, :1 :0
H = F >> G	ADD	SRA# $1, $1!, $2!	SRA# :0, +1:0, :0
Z[i] = H	SUB	SEND# P16, $1!	SEND# P16, :0
	NOOP		
	SRA		
	NOOP		
	SEND P16		

Figure 6.2: Example Loop Body



(a) Original DFG (b) Mapped QISA DFG (c) Mapped RISA/RRISA DFG

Figure 6.3: DFGs for CTE Examples

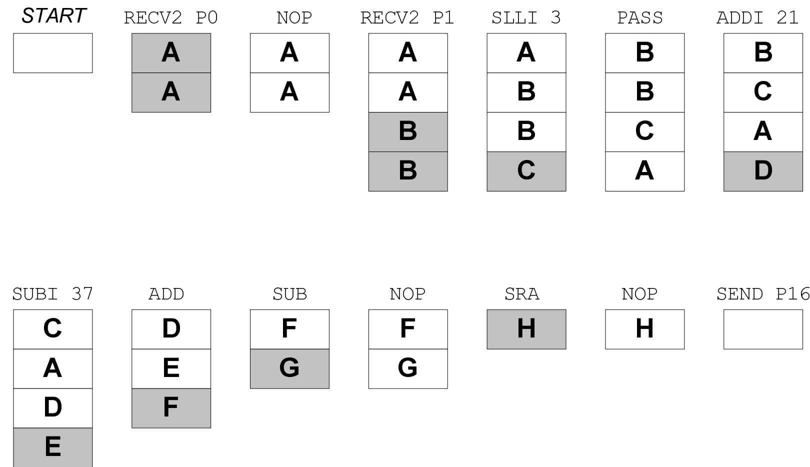


Figure 6.4: Queue Contents for QISA SPU Implementation

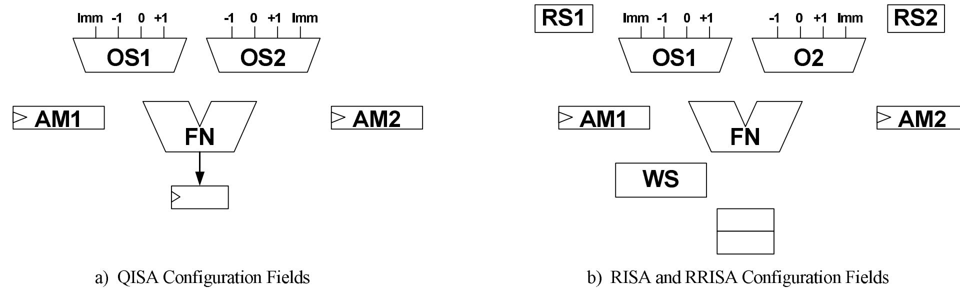
Figure 6.2(a), the (implicit) loop variable is i, and X[i] and Y[i] represent input streams, while Z[i] represents an output stream. MAU initialization for these streams is assumed to have been already completed.

In Figure 6.4 the contents of the queue are shown before and after the execution of each instruction. Each letter corresponds to a variable in Figure 6.2(a) and is shown in its position in the queue, with the head of the queue at the top. Queue items shown with a gray background correspond to items that were produced by the preceding instruction. Starting in the upper left corner, this figure shows the loop body starting with an empty queue. The first instruction, RECV2 P0, causes the SPU to receive a value, A, from MAU port P0, and place two copies of A on the queue. The next instruction is a NOOP and thus no changes are made to the queue. The next instruction, RECV2 P1, causes the SPU to receive a value, B, from MAU port P1, and place two copies of B on the queue; note that they appear on the tail, or bottom of the queue. The next instruction, SLLI 3, takes a single value, A, from the head of the queue, performs a logical shift left by three places, and then places the new value, C, on the queue. The PASS instruction takes the value at the head of the queue, A, and places it at the tail of the queue; note that no new value is produced. The next instruction, ADDI 21, takes a single value, B, from the head of the queue and adds 21, placing the resultant sum on the queue as new value D. Similarly, the next instruction, SUBI 37 takes the value B (the second of the two identical B values placed on the queue) and subtracts 37, giving the new value E. The next instruction

is a two-operand instruction, ADD, which sums the two values at the head of the queue, C and A , and places the sum on the queue as value F. The next instruction is also a two-operand instruction, but it unlike ADD, SUB is non-commutative. In QISA, SUB is defined so that the the second value taken from the queue, E, is subtracted from the first value taken from the queue, D, thus implementing the expression G = D - E defined in the pseudo-code. Note that this is not evident in the DFGs shown in Figure 6.3; however, the order of operations is maintained in the data structures used in the HASTE tools and is clearly defined in the GHAL definitions in Appendix B. The next instruction is a NOOP, so the queue is not affected. The next instruction is SRA, which implements the expressions H = F > > G, followed by another NOOP. The final instruction is a SEND P16, which sends the single remaining value on the queue, H, to MAU port 16, leaving an empty queue. This is the same queue state as was found at the beginning of the loop body, and the next and all succeeding loop iterations can proceed exactly as the first iteration did, if the loop body is running solely on the SPU. This loop body can run on the RCF, however, so in normal operation, while the SPU runs the first loop iteration, the CTE is creating the configuration, as will be described in the next section, and succeeding iterations will run on the RCF.

6.1.2 QISA CTE Operation

The static register fabric as used with the queue ISA and as shown in Figure 5.5 has only a few fields that need to be set for each tile. These are the ALU function select field FUNC, two immediate value fields for those instructions needing them, AIMM1 and AIMM2, and the three operand select values OS1SEL, OS2SEL, and OS3SEL. The inputs to the CTE are the current instruction and the current first queue value seen by the SPU for that operation, Q1. The value of Q1 is needed for a possible live-in value, of which only one is currently supported for QISA, and which can only occur for the first operation on the first row; Q1 is ignored thereafter. The instruction itself contains the instruction opcode field OP, as well as possibly a immediate value IMM, representing a true 16-bit immediate or a 5-bit shift or port immediate value for those instructions requiring them (no instruction can have more than one immediate value in QISA, so only one field is needed). The input and output fields are shown in Table 6.1. For each output field, an abbreviated version is shown in parentheses; the shorter version will be used in some



a) QISA Configuration Fields

b) RISA and RRISA Configuration Fields

Figure 6.5: Fabric Configuration Key

Table 6.1: CTE Inputs from SPU and CTE Outputs to RCF for QISA

Input Field	Description	Output Field, per PE	Description
`OP`	Instruction Opcode	`FUNC (FN)`	Function select for ALU.
`IMM`	Instruction immediate, shift amount or port number	`AIMM1 (AM1)`	Immediate value 1 for ALU.
`Q1`	First queue value	`AIMM2 (AM2)`	Immediate value 2 for ALU.
		`OS1SEL (OS1)`	Operand select for ALU input 1.
		`OS2SEL (OS2)`	Operand select for ALU input 2.
		`OS3SEL (OS3)`	Operand select for ALU input 3.

figures and tables for brevity. The output fields are replicated for each tile across the width of the fabric; they are shifted into the fabric so as to configure each tile. Figure 6.5(a) shows graphically how each output field is used in each tile; the entire configuration for the example is in Figure 6.7. Note that these figures are simplified somewhat in that only two operand select multiplexers are shown, since three-input instructions do not occur in the example.

The hardware comprising the CTE is shown in Figure 6.6. The issue unit consists mainly of two special FIFOs for storing column locations, an instruction look-up table, a multiplexer, and control logic in the form of an issue controller. The FIFOs are special in that up to three values can be removed from or added to them in a single cycle and in that the value of all elements stored in the FIFO can be incremented simultaneously. The configuration stations have some fairly simple logic in a configuration controller and registers to hold configuration data as it is shifted across the width of the fabric. Algorithm 1 shows pseudo-code for the queue ISA CTE algorithm. As indicated, portions of the code represent the functionality of the issue unit and

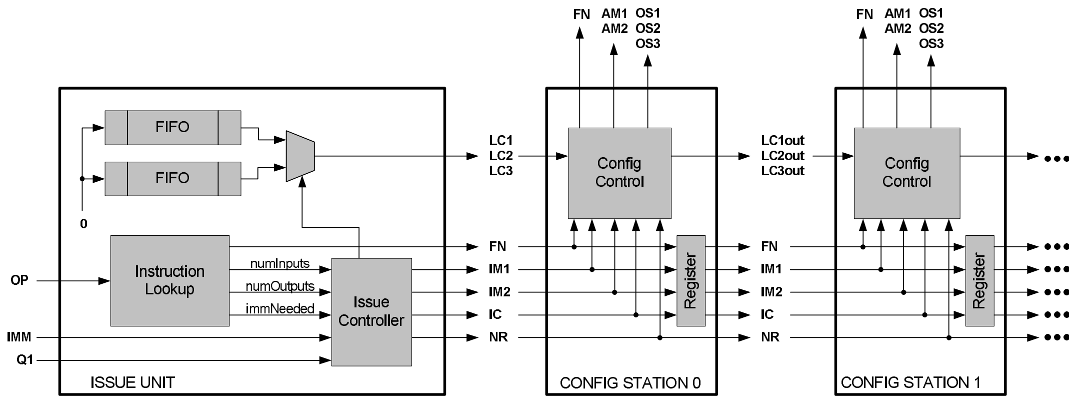


Figure 6.6: Queue ISA CTE

portions the functionality of the configuration stations.

As each instruction is received by the CTE, the lookup table in the issue unit returns the number of inputs and outputs for the current OP, as well as the value for the FN field of the configuration word. The lookup also determines if there is an immediate value that needs to be passed from IMM to either the IM1 field or IM2 field. If this is the first operation in the first row there may be a live-in value, in which case the Q1 field is passed to the IM1 field. The IC signal tells the configuration stations which immediates to use for each instruction. The issue unit needs to maintain two lists of column numbers to track the locations of operands. These are stored in the FIFOs. One is used to track the column location of operands which will be used in the current row and is referred to as the *this* FIFO, and the other is used to maintain the column location of operands which were produced in the current row and will be used in the next row, and is referred to as the *next* FIFO. Each output of the current instruction is represented by an entry in the *next* FIFO. Since each instruction starts in column 0, at the left of the fabric, a 0 is inserted into the *next* FIFO for each output of the current instruction. As each instruction is shifted onto the configuration stations, the location of operands for the next row in the *next* FIFO are incremented, to reflect that the position of each instruction is also increasing by one column. The FIFOs have a control input, not shown, which increments all of the FIFO values and thus enables this updating of column locations. The location of each input for the current instruction is retrieved from the *this* FIFO. These locations are shifted onto the configuration stations as location values numbers LC1, LC2, and LC3. The configuration

Algorithm 1 Queue ISA CTE Algorithm

```
Issue Unit:
Inputs     : instruction = [OP, IMM, Q1]
Outputs  : LOC1, LOC2, LOC3, FUNC, ICTRL, IMM1, IMM2, NEW_ROW

liveInPossible = true;
clear next;
clear this;
for each instruction in kernel
  ICTRL = 0;
  [FUNC, numInputs, numOutputs, immNeeded] <= lookup(instruction.OP);
  if (immNeeded == 1) then
    IMM1 <= instruction.IMM
    ICTRL <= ICTRL + 1;
  endif
  if (immNeeded == 2) then
    IMM2 <= instruction.IMM                                         a
    ICTRL <= ICTRL + 2;
  endif
  if (numInputs > 0 and this is empty) then
    if (liveInPossible) then
      IMM1 <= instruction.Q1;
      ICTRL <= ICTRL + 1;
      numInputs = numInputs - 1;
    else
      this = next;
      clear next;
      signal NEW_ROW;                                               b
    endif
  endif
  for all elements in next {increment element};
  if (numOutputs ≥ 1) {enqueue 0 in next};
  if (numOutputs == 2) {enqueue 0 in next}
  liveInPossible = false
  if (numInputs > 0) { LOC1 <= dequeue value from this }
  if (numInputs > 1) { LOC2 <= dequeue value from this }            c
  if (numInputs > 2) { LOC3 <= dequeue value from this }
 }
}

Configuration Station:
Inputs: FUNC, IMM1, IMM2, LOC1, LOC2, LOC3
Outputs: FUNC, AIMM1, AIMM2, OS1SEL, OS2SEL, OS3SEL, LOC1, LOC2, LOC3

OS1SEL <= (ICTRL[0] ? Imm : LOC1);
OS2SEL <= (ICTRL[1] ? Imm : LOC2);
OS3SEL <= LOC3;                                                     d
LOC1 <= LOC1 - 1;
LOC2 <= LOC2 - 1;
LOC3 <= LOC3 - 1;
```

station logic converts these absolute column numbers into relative settings for the OS1, OS2, and OS3 fields that control the operand select multiplexers, by decrementing them as they pass from station to station. The decremented values produced by the configuration station in column 0 are shown in Figure 6.6 as LC1out, LC2out, LC3out. These are the values for LC1, LC2, and LC3 supplied to the next station.

If there are no more operands in *this* FIFO, the CTE starts a new row, and all configuration words in the configuration stations are shifted down into the fabric. The NR (New Row) signal is asserted by the issue unit to control the starting of a new row. At the start of a new row the algorithm requires that the contents of the *next* FIFO should be copied into the *this* FIFO, since the values produced in the previous row will be used as inputs in the new row. In actuality, the usage of each of the two identical FIFOs is switched using the multiplexer, so it is not necessary to actually copy the values from one FIFO to the other.

6.1.3 QISA CTE Operation for Example

The process of converting this code into a configuration is shown in Table 6.2. This process is the implementation of Algorithm 1 using the hardware shown in Figure 6.6 and will be explained in detail. Table 6.2 shows the value of important signals throughout the conversion process. Each row shows the values for one stage of the conversion process, each stage corresponding to a shaded box in Algorithm 1. Stage **a** involves reading the lookup table and the setting of immediate values and the immediate control signal. Stage **b** involves the preparation of the *this* and *next* FIFOs by either incrementing the *next* FIFO to account for the addition of a new node in the current row, or the shifting of values from the *this* FIFO to the *next* FIFO. Stage **c** involves updating the FIFOs by adding and removing operand locations. Stage **d** involves the generation of the final configurations in the configuration stations and shifting the finished configuration into the fabric after the completion of each row. Note that stages **a-c** take place in the issue unit, while stage **d** takes place concurrently in each configuration station.

6.1.3.1 Table and Configuration Description

The leftmost column of each row of Table 6.2 identifies the stage and clock cycle represented by that row. The next three columns represent the instruction being processed. The "Inst" column shows the instruction as sent to the SPU. The next two columns, "OP" and "IMM" show the instruction broken down into opcode and immediate fields, as are sent from the SPU to the CTE. The value for Q1 should be shown in this section, but since the example does not have a runtime constant, it was not needed and the corresponding column was removed for brevity. The next twelve columns show the signals and values in the issue unit itself. The first four show the values returned by the instruction lookup based on the input opcode; note that *numInputs* is shown as nI, *numOutputs* is shown as nO, and *immNeeded* is shown as iN. The next two show the contents of the FIFOs, with input values inserted at the right end of the list and output values removed from the left end of the list. The next two columns show the operand location values as read from the *this* FIFO. Note that only LC1 and LC2 are shown, since there are no three operand instructions in the example and thus LC3 is not needed. The next four values show the outputs of the Issue Controller, including both immediate values (IM1 and IM2), the immediate control signal (IC) and the new row signal (NR). Finally, the next four sets of five columns each show the outputs of each configuration station. Since the example is four columns wide, four configuration stations are shown. For each station, the values for the ALU function select (FN), the ALU immediates (AM1 and AM2), and operand selects (OS1 and OS2) are shown. The OS3 field is not shown since there are no three-operand instructions in the example. The character 'x' is used to designate entries as "don't cares" and the '-' character is used for the FIFOs only, to indicate that they are empty.

The fabric configuration produced in the example is shown in Figure 6.7. This figure shows each ALU used by the example, with the ALU function (FN) shown inside the ALU symbol. ALUs in the fabric that are not used are not shown. Each ALU input multiplexer (operand select multiplexer) that is used has its operand select value (OS1 or OS2) and the active connection shown; unused interconnect is not shown. The two immediate registers are also shown for each used ALU, with the contents (AIMM1 or AIMM2) shown inside the register if it is used. Figure 6.5 serves as a key to the configuration, showing the names of the configuration fields in their location. Finally, the output produced by each ALU is shown in its output register for comparison

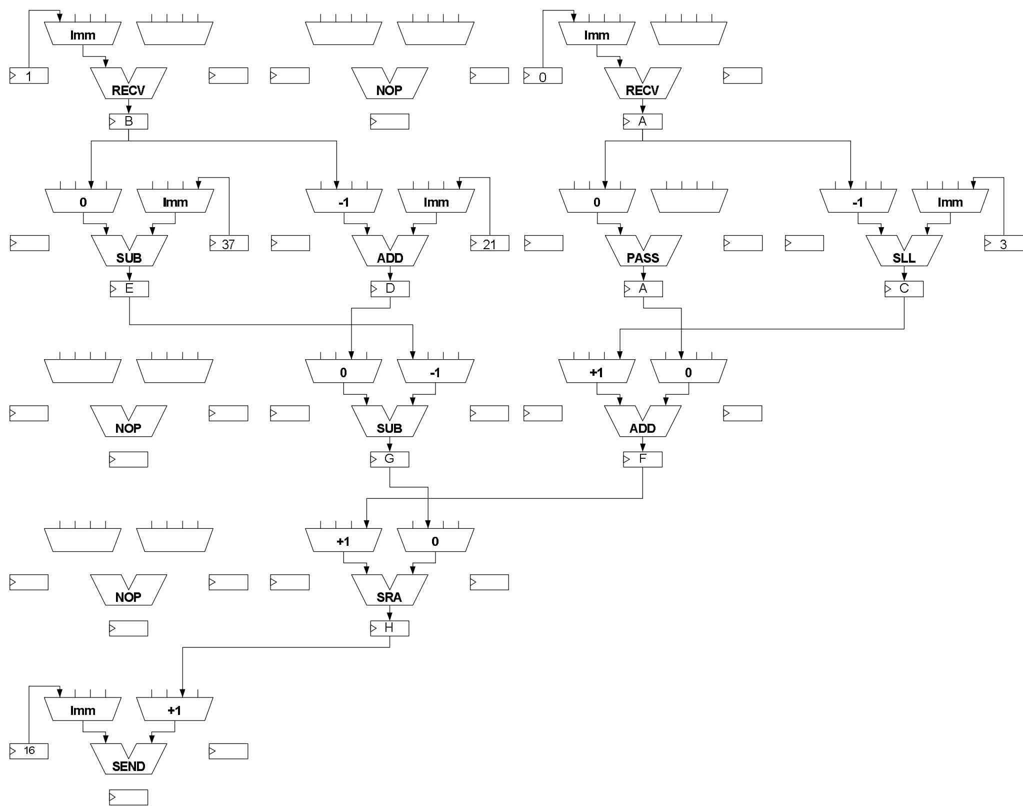


Figure 6.7: QISA Fabric Configuration for Example

to Figure 6.3(a).

6.1.3.2 Processing of Example Instructions

The first instruction is RECV2 P0, which in the SPU receives a value from Port 0 of the MAU and places two copies on the queue. The opcode is "RECV2" and the immediate (port number) is 0. In stage **a**, the instruction lookup returns the ALU function "RECV", and calls for no inputs (nI = 0), two outputs (nO = 2), and an immediate as the first operand (iN = 1). Note that the RECV ALU function does not include the '2' suffix; this is only needed in the SPU and CTE and does not matter in the RCF, since the configuration is the same for a PE regardless of how many times its result is read. The controller outputs the corresponding immediate value and control signal (IM1 = 0 and IC = 1). In the next stage, stage **b**, nothing happens since a new row is not needed and there are no entries in the *next* FIFO to increment. In stage **c**, the *next* FIFO is updated by adding 0's for the two outputs of the instruction; the 0's correspond to the location

Table 6.2: QISA CTE Example

		Issue Unit														Configuration Stations																			
		CTE Inputs		Inst. Lookup				FIFOs				Controller				Config Station 0					Config Station 1					Config Station 2					Config Station 3				
	Inst	OP	IMM	FUNC	nI	nO	iN	this	next	LOC1	LOC2	IMM1	IMM2	ICTRL	NR	FN	AM1	AM2	OS1	OS2	FN	AM1	AM2	OS1	OS2	FN	AM1	AM2	OS1	OS2	FN	AM1	AM2	OS1	OS2
1a	RECV2 P0	RECV2	0	RECV	0	2	1	-	-	x	x	0	x	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
1b	"	"	"	"	"	"	"	-	-	x	x	0	x	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
1c	"	"	"	"	"	"	"	-	0-0	x	x	0	x	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
1d	"	"	"	"	"	"	"	-	0-0	x	x	0	x	1	0	RECV	0	x	Imm	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
2a	NOP	NOP	x	NOP	0	0	0	-	0-0	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
2b	"	"	"	"	"	"	"	-	1-1	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
2c	"	"	"	"	"	"	"	-	1-1	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
2d	"	"	"	"	"	"	"	-	1-1	x	x	x	x	0	0	NOP	x	x	x	x	RECV	0	x	Imm	x	x	x	x	x	x	x	x	x	x	x
3a	RECV2 P1	RECV2	1	RECV	0	2	1	-	1-1	x	x	1	x	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
3b	"	"	"	"	"	"	"	-	2-2	x	x	1	x	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
3c	"	"	"	"	"	"	"	-	2-2-0-0	x	x	1	x	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
3d	"	"	"	"	"	"	"	-	2-2-0-0	x	x	1	x	1	0	RECV	1	x	Imm	x	NOP	x	x	x	x	RECV	0	x	Imm	x	x	x	x	x	x
4a	SLLI 3	SLLI	3	SLL	1	1	2	-	2-2-0-0	x	x	x	3	2	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
4b	"	"	"	"	"	"	"	2-2-0-0	-	x	x	x	3	2	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
4c	"	"	"	"	"	"	"	2-0-0	0	2	x	x	3	2	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
4d	"	"	"	"	"	"	"	2-0-0	0	2	x	x	3	2	0	SLL	x	3	+2	Imm	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
5a	PASS	PASS	x	PASS	1	1	0	2-0-0	0	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
5b	"	"	"	"	"	"	"	2-0-0	1	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
5c	"	"	"	"	"	"	"	0-0	1-0	2	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
5d	"	"	"	"	"	"	"	0-0	1-0	2	x	x	x	0	0	PASS	x	x	+2	x	SLL	x	3	+1	Imm	x	x	x	x	x	x	x	x	x	x
6a	ADDI 21	ADDI	21	ADD	1	1	2	0-0	1-0	x	x	x	21	2	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
6b	"	"	"	"	"	"	"	0-0	2-1	x	x	x	21	2	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
6c	"	"	"	"	"	"	"	0	2-1-0	0	x	x	21	2	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
6d	"	"	"	"	"	"	"	0	2-1-0	0	x	x	21	2	0	ADD	x	21	0	Imm	PASS	x	x	+1	x	SLL	x	3	0	Imm	x	x	x	x	x
7a	SUBI 37	SUBI	37	SUB	1	1	2	0	2-1-0	x	x	x	37	2	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
7b	"	"	"	"	"	"	"	0	3-2-1	x	x	x	37	2	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
7c	"	"	"	"	"	"	"	-	3-2-1-0	0	x	x	37	2	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
7d	"	"	"	"	"	"	"	-	3-2-1-0	0	x	x	37	2	0	SUB	x	37	0	Imm	ADD	x	21	-1	Imm	PASS	x	x	0	x	SLL	x	3	-1	Imm
8a	ADD	ADD	x	ADD	2	1	0	-	3-2-1-0	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
8b	"	"	"	"	"	"	"	3-2-1-0	-	x	x	x	x	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
8c	"	"	"	"	"	"	"	1-0	0	3	2	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
8d	"	"	"	"	"	"	"	1-0	0	3	2	x	x	0	0	ADD	x	x	+3	+2	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
9a	SUB	SUB	x	SUB	2	1	0	1-0	0	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
9b	"	"	"	"	"	"	"	1-0	1	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
9c	"	"	"	"	"	"	"	-	1-0	1	0	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
9d	"	"	"	"	"	"	"	-	1-0	1	0	x	x	0	0	SUB	x	x	+1	0	ADD	x	x	+2	+1	x	x	x	x	x	x	x	x	x	x
10a	NOP	NOP	x	NOP	0	0	0	-	1-0	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
10b	"	"	"	"	"	"	"	-	2-1	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
10c	"	"	"	"	"	"	"	-	2-1	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
10d	"	"	"	"	"	"	"	-	2-1	x	x	x	x	0	0	NOP	x	x	x	x	SUB	x	x	0	-1	ADD	x	x	+1	0	x	x	x	x	x
11a	SRA	SRA	x	SRA	2	1	0	-	2-1	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
11b	"	"	"	"	"	"		2-1	-	x	x	x	x	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
11c	"	"	"	"	"	"		-	0	2	1	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
11d	"	"	"	"	"	"		-	0	2	1	x	x	0	0	SRA	x	x	+2	+1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
12a	NOP	NOP	x	NOP	0	0	0	-	0	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
12b	"	"	"	"	"	"	"	-	1	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
12c	"	"	"	"	"	"	"	-	1	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
12d	"	"	"	"	"	"	"	-	1	x	x	x	x	0	0	NOP	x	x	x	x	SRA	x	x	+1	0	x	x	x	x	x	x	x	x	x	x
13a	SEND P16	SEND	16	SEND	1	0	1	-	1	x	x	16	x	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
13b	"	"	"	"	"	"	"	1	-	x	x	16	x	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
13c	"	"	"	"	"	"	"	-	-	x	1	16	x	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
13d	"	"	"	"	"	"	"	-	-	x	1	16	x	1	0	SEND	16	x	Imm	+1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

of the first node in a stripe, since instructions are issued from the left and the leftmost column is column 0, as shown in Figure 6.1. Finally, in stage **d**, the OS1 value for the first configuration station is determined to be "Imm", meaning that the first immediate should be read, due to the value of IC. This value, plus the FN code and the immediate value, provide the configuration needed for the tile that will implement this instruction.

The second instruction is NOOP. In stage **a**, the lookup returns 0 inputs, 0 outputs, and no immediates, as well as the NOOP function code. In stage **b**, the values in the *next* FIFO are incremented. Since a new instruction, the current NOOP instruction, has been issued on this row, the first instruction moves to column 1, so the incremented values represent the location of the instruction that produces the first two operands produced in the current row. Nothing is done for this instruction in stage **c**, since it has no inputs or outputs. In stage **d**, it can be seen that the first instructions is now in column one and the NOOP instruction is in column 0, with no fields other than FN needed for the NOOP.

The third instruction is RECV2 P1, identical to the first instruction except that the port number (immediate value) is 1. Stage a is the same as for the first instruction. In stage **b**, the next FIFO is not empty as it was for the first instruction; it contains the values "1-1". These values are incremented to "2-2", since with the introduction of the new instruction on the current row, the first instruction moves to column 2. Then in stage **c**, two 0's are added to the FIFO, representing the outputs of the current instruction. In stage d, the final configurations for each of the columns in the current row are shown.

The fourth instruction is SLLI 3, the first instruction in this example with an input operand. The lookup results are one input (nI = 1), one output (nO = 1), immediate as second operand (iN =2), and FN = "SLL". Note that the second operand takes the immediate for this instruction (if it was necessary for immediate to be the first operand, the SLLIR instruction could have been used) and that the function code does not indicate an immediate; instead the IC signal indicates the immediate. In stage **a**, the immediate value and control signals are set (IM2 = 3 and IC = 2). In stage **b**, an input is needed, but the *this* FIFO is empty. Therefore the NR signal is asserted and the row shown in the gray box on line 3d is shifted into the fabric. The *next* and *this* FIFO mux is also switched, so that the location of operands generated by the previous row can now be used to determine the location of the operands to be consumed in the current row.

In stage **c**, the location of the input operand is taken from the *this* FIFO; it is column 2. This is passed to the configuration stations as LOC1. In addition, a 0 will be placed on the *next* FIFO to represent the operand produced by the current instruction. Note in stage **d**, the configuration has function code "SLL"; the use of an immediate is indicated by the second operand select value, OS2 = Imm. Also, the first location is shown as +2. Currently, this instruction is in column 0, and it needs to get its input from column 2. Therefore, the operand select multiplexer is set to +2, meaning to select a value from two columns to the right. Thus the configurations in the fabric rely on relative, not absolute locations.

The fifth instruction is PASS, which returns one input (nI = 1), one output (nO = 1), no immediate (iN = 0), and FN = "PASS" from the lookup. Stage **a** produces an indication of no immediate (IC = 0). Stage **b** increments the *next* FIFO, since the SLLI instruction now moves to column 1, and then stage **c** gets the location of the input operand as column 2 and places another 0 on the *next* FIFO. In stage **d**, the configurations are generated. Note that the operand select value for the SLL instruction has been decremented from +2 to +1. The input operand for the instruction is still in column 2, but the instruction is now in column 1, so the offset is now only +1, or one column to the right. The sixth and seventh instructions are similar to the SLLI instruction; all have one input operand, one immediate operand as the second input, and one output value. The final configuration for this row is shown in row 7d. Note that the final operand select value for the SLLI instruction is now -1, since the instruction ends up in column 3 and the input operand is in column 2, so it must select from one column to the left.

The next row starts with a NOOP inserted for spacing; this fabric has $R_C = 3$, so this NOOP is needed so that the next instruction, an ADD, is in a column where it can read values from both column 3 and column 2. This next instruction is the first instruction with two non-immediate input operands in the example. It is processed just as before, except that two values are taken from the *this* FIFO in stage **c**. The order is important in general, although not in this case, since addition is commutative. It is important for the next instruction, which is a SUB instruction. The first operand for the subtraction is the first one from the *this* FIFO and the second operand is the second one. This does the correct subtraction as required by the original pseudo-code. If the inputs were not in the correct order, a SUBR instruction could be used instead. The place and route tools account for the ordering of operands where needed and convert instructions to

the reversed input versions if necessary. The completed third row is shown in row 10d.

The fourth and fifth rows proceed just as the previous rows did. The completed row configurations are shown in the table on rows 12d and 13d. At the end of the kernel, the *this* and *next* FIFOs are both empty, which is the expected result. After the loop end instruction is encountered, the final row is shifted into the fabric and the fabric is completely configured. Execution then switches from the SPU to the RCF. It can be seen that the final RCF configuration as shown in Figure 6.7 corresponds to the gray boxes in Table 6.2 and that the configuration implements the DFG shown in Figure 6.3(b).

6.2 Register ISA SPU and CTE Operation

6.2.1 RISA SPU Operation for Example

The RISA code for the example is shown in Figure 6.2(c) and the DFG shown in Figure 6.3(c) corresponds to this code sequence. This code sequence implements the loop body for the pseudo-code shown in Figure 6.2(a). For this example, there are no live-in variables, as can be seen from the pseudo-code. It will be further assumed that there are no live-out register values and thus in this application all input and output to and from the loop body is carried by streams. Most streaming applications have this same characteristic; the HASTE concept does not require this, however. In the example pseudo-code in Figure 6.2(a), all data passed into the loop body is carried by input streams X[i] and Y[i] and all data passed out of the loop body is carried by output stream Z[i], all using with loop variable i . As f the QISA example, MAU initialization is assumed to have been already completed.

In Figure 6.8 the contents of the SPU registers after the execution of each instruction are shown. Only the four registers actually used by the loop body example are shown and it is assumed that there is nothing significant in any of the registers at the start of the loop body. Each letter corresponds to a variable in Figure 6.2(a) and is shown in the register where it is stored. Registers shown with a shaded background correspond to values that were produced by the preceding instruction. Registers surrounded by a thick outline show register locations that were read by the preceding instruction, although a new value may have also been placed in that location (and will have a gray background if so). Starting in the upper left corner, this figure

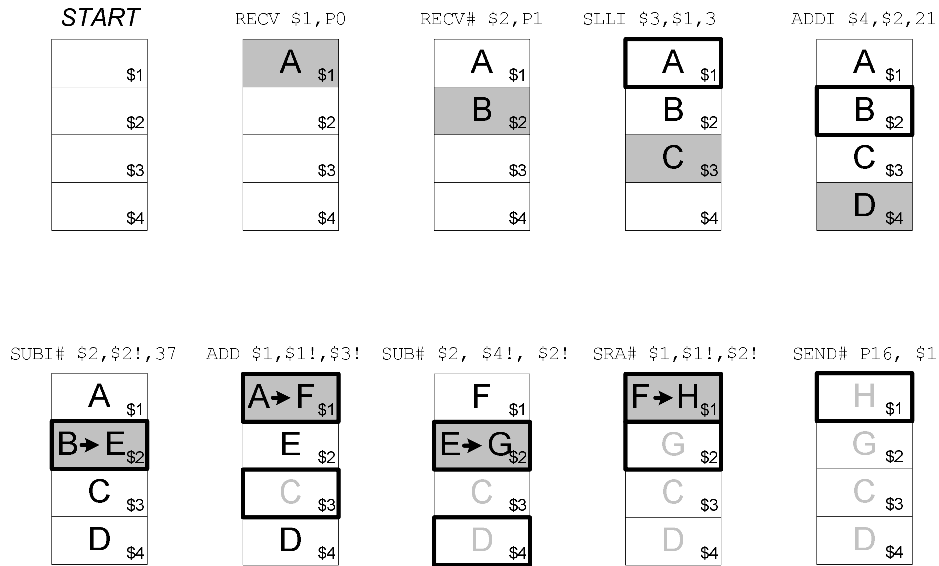


Figure 6.8: Register Contents for RISA SPU Implementation

shows the loop body starting with an empty register file. The first instruction, RECV $1,P0 causes the SPU to receive a value, A, from MAU port P0, and place it in register $1. Since a register can be read from multiple times, unlike a queue value, it is only necessary to store one copy of A, as opposed to the two copies that were stored in the QISA example. No NOOPs are required in the RISA version of the example, again unlike the QISA version, so the next instruction is RECV# $2,P1 which causes the SPU to receive a value, B, from MAU port P1, and places it in register $2. The '#' character indicates that this is the last instruction on a row. Unlike QISA, the CTE cannot determine the end of the row itself, so the last instruction in the row is annotated by the compiler. This is ignored by the SPU, but is necessary for the CTE.

Next is the first arithmetic instruction, SLLI $3,$1,3, which reads the value A from register $1, performs a logical shift left by three places, and then places the new value, C, in register $3. The next instruction, ADDI $4,$2,21, reads value B from register $2 and adds 21, placing the resultant sum, value D, in register $4. Similarly, the next instruction, SUBI# $2, $2!,37 reads value B from register $2 again, and subtracts 37, giving the new value E, which it places in register $2. Note that this instruction is the last instruction to use value B, which is stored in register $2. The compiler realizes this and therefore allocates the same register for value E. The

'!' after the second $2 in the in the instructions also indicates that this is in fact the last read of the value stored in source register $2 (The first $2 is the destination register; it just so happens that this is the same as the source being used for the last time, but this is not always the case). The SPU can ignore the '!' and just use the register allocation designated by the compiler, but the CTE will use this information so that it can efficiently allocate registers in the fabric, as will be shown in the next section. The '#' after SUBI indicates that is the last instruction on the current row, as was the case for the second RECV instruction.

The next instruction is a two-operand instruction, ADD $1,$1!,$3!, which sums the two values C and A, and places the sum (value F) in register $1. Both A and C are read for the last time from registers $1 and $3, as indicated; register $3 is not shown as being empty, however, because '!' notation is ignored in the SPU and thus values stay in registers unless they are overwritten by another value. Values that aren't used again are shown in gray text in the register file. Note that A was written on the first row, used previously on the second row, and is used here on the third row of the DFG. The next instruction, SUB# $2,$4!,$2! is also a two-operand instruction, subtracting E in register $2 from D in register $4, and placing the result in register $2. The next instruction is SRA# $1,$1!,$2!, implements the expressions H = F > > G, and is otherwise the same as the preceding instruction. The final instruction is a SEND# P16 $1!, which sends the value in register $1, H, to MAU port 16. This register is read for the last time by this instruction. Now the next and all succeeding loop iterations can proceed exactly as the first iteration did, if the loop body is running solely on the SPU. This loop body can run on the RCF, however, so in normal operation, while the SPU runs the first loop iteration, the CTE is creating the configuration, as will be described in the next section, and succeeding iterations will run on the RCF, just as was seen in the QISA example.

6.2.2 RISA CTE Operation

The asymmetric pass register fabric is used with the register ISA and is shown in Figure 5.6. It has all of the same fields as the static register fabric: the ALU function select field FUNC, two immediate value fields AM1 and AM2, and three operand select values OS1, OS2, and OS3. In addition, it has two new sets of fields: RSn and WSn. The first are the read select fields, which determines which register is being read for each possible read. At first glance it might appear

there would need to be n such fields, with n equal to the product of the read connectivity of the fabric and the number of input operands. For the simple fabric used in this example, there is a read connectivity of three and only two inputs are used, so one might expect there to be six RS fields, RS1 through RS6; more complicated fabrics would have even more RS fields. However, the read register fields are not stored with the register file being read but instead are actually stored in the tile that is reading from the register file; since not all possible reads are performed, only enough fields for the reads actually performed are needed. Since there can be at most three input operands per tile, it is only necessary to store at most three RS values per tile, so $n = 3$; in the example only two are needed. These are passed to the appropriate register file using the operand select values. Since register file sizes are small, each field is only a few bits, so this does not require a large number of configuration bits. These fields were not needed for the static register fabric, since there was only one register to be read for each operand select multiplexer input location. The second set of fields are the write select fields, with n in this case equal to number of registers in a register file. These are single bits that determine whether a new value should be written to the register from the ALU or if register should just pass in the value from the previous corresponding pass register. For the example fabric that will be used, with two registers per register file, there will be two single-bit write select fields, WS1 and WS2; these will be concatenated to form a single two-bit field, WS, in succeeding figures. These fields were not needed for the static register fabric, since the single register in each tile was always written by the corresponding ALU. In Figure 6.5(b), the locations of the PE configuration fields for the example fabric shown. The complete configuration is shown in Figure 6.9.

The inputs to the CTE are the current instruction and the current values of the first two source registers, IR1, IR2. These current register values are needed for live-in values, which can occur anywhere in a fabric, and cannot be changed after loop execution has begun. Thus they represent run-time constants; note that the third source for instructions using three sources cannot be a live-in value, as enforced by the compiler. The instruction itself contains the instruction opcode field OP; a destination register field, DR; source register fields SR1, SR2, and SR3; source register kill fields RK1, RK2, and RK3 (these correspond to the '!' characters seen previously); a new row field NR (this corresponds to the '#' character which indicates a new row); and an immediate field IMM. The input and output fields are shown in Table 6.3, with both full and abbreviated

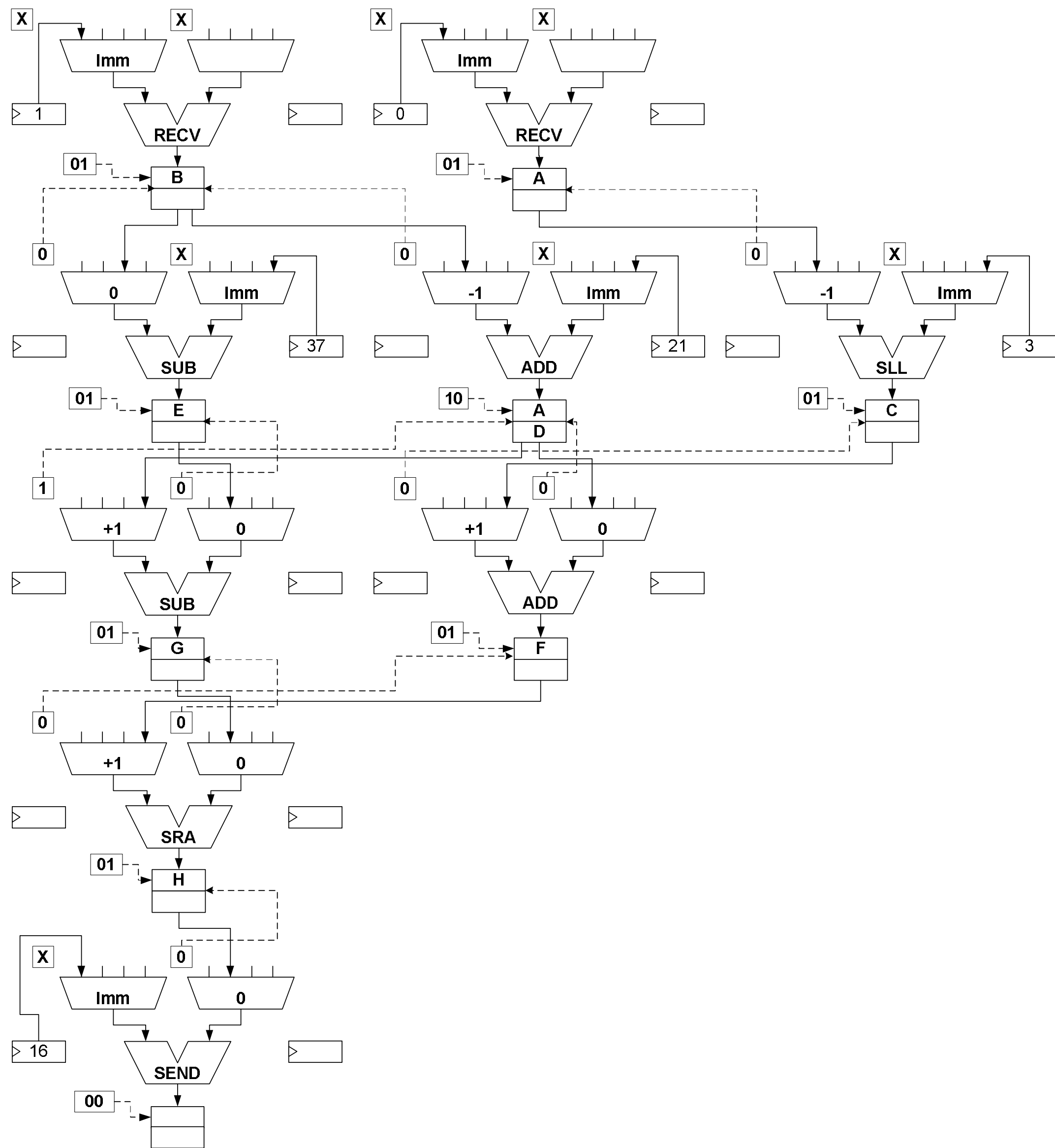


Figure 6.9: RISA Fabric Configuration for Example

Table 6.3: CTE Inputs from SPU and CTE Outputs to RCF for RISA

Input Field	Description	Output Field	Description
OP	Instruction Opcode	FUNC (FN)	Function select for ALU.
DR	Destination register.	AIMM1 (AM1)	Immediate value 1 for ALU.
SR1, SR2, SR3	Source registers.	AIMM2 (AM2)	Immediate value 2 for ALU.
RK1, RK2, RK3	Register kill fields.	OS1SEL (OS1)	Operand select for ALU input 1.
NR	New row field.	OS2SEL (OS2)	Operand select for ALU input 2.
IMM	Instruction immediate, shift amount or port number	OS3SEL (OS3)	Operand select for ALU input 3.
LR1, LR2	Live-in register values.	RSELn (RSn)	Register read selects.
		WSELn (WS)	Register write selects.

forms shown. For the register specification fields, (SR1, SR2, SR3, and DR) a -1 value indicates that a register is not specified; this usually represents the largest possible value for that field, which cannot be a valid register by convention. The SPU decoder fills in this value as needed, depending on the instruction opcode and instruction format.

The hardware comprising the CTE is shown in Figure 6.10. It is similar to the QISA CTE. Instead of FIFOs to store operand locations, there is a register table and there are some different input and output fields. Otherwise both the issue unit and the configuration stations are similar to their QISA counterparts. Algorithms 2 and 3 show pseudo-code for the register ISA CTE algorithm. Algorithm 2 represents the functionality of the issue unit and Algorithm 3 represents the functionality of the configuration stations.

As with the QISA CTE algorithm, the first step of the RISA CTE algorithm is to look up the OP field in the instruction lookup. This provides information about the usage of immediates for each instruction and allows the configuration of the related fields. The register table contains information that allows for translation between SPU registers and fabric registers. As each new instruction on a row is processed, the register table must be updated to reflect the new location of those instructions on that same row, as new instructions are issued from the left and old ones on the same row are pushed farther to the right. The register table is then used to find the location of source registers for the current instruction; since they come from the previous row, their locations are already fixed. If a source register is not in the table, it is assumed to be a live-in value and that value is placed in the fabric as a constant. If the current instruction produces an output, this is then noted by an entry in the register table. The configuration stations use the

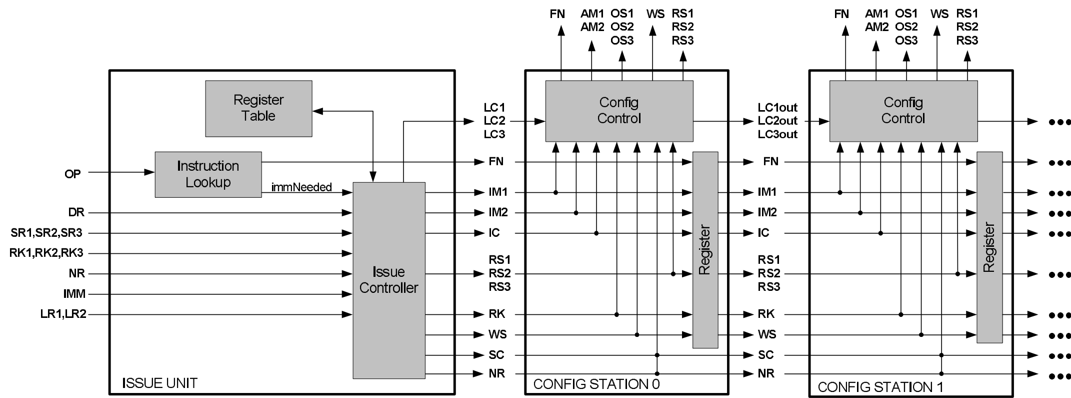


Figure 6.10: Register ISA CTE

information from the issue unit to determine everything except for write locations. The values for register reads are also examined so that the register tables can be updated to reflect register locations that can be cleared due to register kills. Finally, at the end of each row, the final locations of SPU register in the fabric can be determined and stored in the register table. The usage of registers in each column is stored in the configuration station register tables, mirroring that in the issue unit. In addition, if the current row has tiles that read from registers in previous rows that are to the right of the rightmost register in the current row, the current register reads are shifted into the fabric so that the register tables in the configuration stations can be updated. This entire process will be covered in more detail in the next section.

6.2.3 RISA CTE Operation for Example

The process of converting this code into a configuration is shown in Tables 6.4 and 6.5. This process is the implementation of Algorithms 2 and 3 using the hardware shown in Figure 6.10 and will be explained in detail in this section. As was the case for the QISA example, the tables shows the value of important signals throughout the conversion process. Rather than one table, the number of signals in the RISA CTE required two tables, one for the issue unit and one for the config stations. Each row in both tables shows the values for one stage of the conversion process, with each stage corresponding to a shaded box in Algorithm 2 and/or Algorithm 3. Stage **a** involves reading the lookup table and the setting of immediate values and the immediate control signal. Stage **b** involves the translating SPU register numbers into fabric locations. Stage

Algorithm 2 RISA ISA Issue Unit Algorithm

```
Issue Unit:
Inputs    : instruction = [OP,  SR1, SR2, SR3, RK1, RK2, RK3, RD, IMM, NR, LR1, LR2]
Outputs  : LC1, LC2, LC3, FUNC,  IM1, IM2, IC, RS1, RS2, RS3, RKx, WE, NR

clear IRT
read_right = 0;
for each instruction in kernel                                              a
        NR = 0
        read_right-
        IC = 0;
        [FN, immNeeded] <= lookup(instruction.OP)
        if (immNeeded ==  1) then
                IM1 <= instruction.IMM
                IC <= IC + 1
        else if (immNeeded ==  2) then
                IM2 <= instruction.IMM
                IC <= IC + 2
        endif
        if (!instruction.NR) increment live write column locations in IRT

        for n = 1..3                                                        b
                 if (instruction.SRn != -1) then
                        if (instruction.SRn not in register table)
                                IC <= IC+ n
                                IMn <= instruction.LRn
                        else if (instruction.RKn)
                                IRT.clear(instruction.SRn)
                        endif
                        LCn = IRT.col(instruction.SRn)
                        if (LCn > read_right) tmp_read_right = LCn;
                        RSn = IRT.reg(instruction.SRn)
                endif
        end

if (instruction.DR != -1) then                                              c
        WE <= 0
        new live register at column 0 in IRT = instruction.DR
endif

if (instruction.NR)                                                         d
        place each live value in IRT in first open register in its column
        this_col = 0;
        while (read_right > 0)
                SC = 1
                shift reads right
                read_right--
        end
        SC = 0
        NR = 1
end if
```

Algorithm 3 RISA ISA Configuration Station Algorithm

```
Configuration Control:
Inputs: LC1, LC2, LC3, FN, IM1, IM2, IC, RS1, RS2, RS3, RK, WS, NR, SC
Outputs: config[FN, AM1, AM2, OS1, OS2, OS3, WS, RS1, RS2, RS3], LC1out, LC2out, LC3out

clear RT;
while (!done)
        if (!SC)
                config.FN     <= FN
                config.AM1    <= IM1
                config.AM2    <= IM2
                config.OS1    <= (IC[0] ? Imm : LC1)
                config.OS2    <= (IC[1] ? Imm : LC2)
                config.OS3    <= LC3                                   c
                config.RS1    <= RS1
                config.RS2    <= RS2
                config.RS3    <= RS3
                LC1out        <= LC1 - 1;
                LC2out        <= LC2 - 1;
                LC3out        <= LC3 – 1;
        endif

        if (RSn == 0 && RKn) clear RT[n]
        if (NR && WE)
                RO = first open register in RT
                RT[ RO] = 1;                                           d
                config.WS <= RO (one-hot encoded)
        endif
end
```

c tracks new register locations in the issue unit and begins forming the config word in the config stations. Stage **d** takes place at the end of each row and involves fixing the locations of values produced in the current row, as well as producing the final config words for each column. Note that stages **a** and **b** take place in the issue unit only, while stages **c** and **d** takes place in both the configuration station and the issue unit. Stages **a-c** take place for each instruction, while stage **d** takes place only at the end of a row.

6.2.3.1 Table and Configuration Description

For Table 6.4 (as well as for Table 6.5), the leftmost column of each row identifies the stage and clock cycle represented by that row. The "Inst" column shows the instruction as sent to SPU. The next eight columns show all of the fields that can be sent from the SPU to the CTE. Not all of these fields will be used for any single instruction, as can be seen from the ISA formats as shown in Figure 4.10. The "OP" column shows the instruction opcode. "SR1", "RK1", "SR2", and "RK2" columns give the register and register kill fields for the first and second source registers. Note that since there are no three operand instructions in the example, a third source register is never needed and thus the "SR3" or "RK3" fields are not shown. There is also single destination register field "DR"; if the design were using a 64-bit multiplies, as discussed in Chapter 4, there would be second destination register field here as well. There is an immediate field, "IMM", which is used both for 16-bit immediates for I-Type instructions, as well for 5-bit shift amounts and port numbers. Finally, the "NR" column holds the new row bit. The live-in values "LR1", LR2", and "LR3" would be shown in this section, but the example does not have any live-in values, so the corresponding columns were removed for brevity.

The next eighteen columns show the signals and values in the issue unit itself. The first two show the values returned by the instruction lookup based on the input opcode; "FN" is the function to be performed by the ALU for that instruction and "iN" signifies if an immediate is needed, and if so, whether it should be in the first or second operand location (immediates are not supported for the third operand location). The next six show the contents of the issue register table (IRT), with an entry for each register in each column of the fabric. A separate entry should be included for the "live" register values, but instead these values are shown in the table using a bold italic font, in order to save space. The next ten columns show the outputs of the Issue

Controller, including both immediate values, "IM1" and "IM2"; the immediate control signal "IC"; two operand location signals,"LC1" and "LC2", (the third operand location is not needed for this example); register select signals "RS1" and "RS2"; two register kill signals concatenated as "RKx"; the write enable signal "WE"; and the new row signal "NR".

In Table 6.5 there are three sets of twelve columns, each set of which shows the outputs and internal signals of a configuration station. Since the example is three columns wide when mapped to RISA, only three issue stations are shown. For each station, the output values ALU function select "FN", ALU immediates "AM1" and "AM2", operand selects "OS1" and "OS2", register selects "RS1" and "RS2", and concatenated write selects "WS" are shown. As always, signals referring to the third input operand are not shown. In addition, the internal values are shown for the concatenated register kills signals "RKx", the write enable "WE", and the internal register table values "RT", one for each of the registers in the column.

The fabric configuration produced in the example is shown in Figure 6.7. This figure shows each ALU used by the example, with the ALU function (FN) shown inside the ALU symbol. ALUs in the fabric that are not used are not shown. Each ALU input multiplexer (operand select multiplexer) that is used has its operand select value (OS1 or OS2) and the active connection shown; unused interconnect is not shown. The two immediate registers are also shown for each used ALU, with the contents (AIMM1 or AIMM2) shown inside the register if it is used. The rectangle to the left of the register files shows the write enable signals, while the small squares above each operand select multiplexer show the register signal for the register being read. The values stored in each register are shown although the pass register wires are not shown.

6.2.3.2 Processing of Example Instructions

As previously discussed, the processing of the example kernel is shown in Tables 3 and 2, and the final configuration is shown in Figure 6.9. At the beginning of kernel processing, the issue register table and the configuration station register tables are cleared. The first instruction is RECV \$1,P0, which in the SPU receives a value from Port 0 of the MAU and places it in register \$1. It has an opcode of RECV, DR = 1, and an immediate value of 0. In stage **a**, the instruction

Table 6.4: RISA CTE Example - Issue Unit

		CTE Inputs								Issue Unit																			
										Inst. Lookup		Issue Register Table																	
												Col 0		Col 1		Col 2													
	Inst	OP	SR1	RK1	SR2	RK2	DR	IMM	NR	FUNC	iN	0	1	0	1	0	1	RR	IM1	IM2	IC	LC1	LC2	RS1	RS2	RKx	WE	SC	NR
1a	RECV $1,PO	RECV	x	x	x	x	1	0	0	RECV	1	-	-	-	-	-	-	-1	0	x	1	x	x	x	x	x	x	0	0
1b	"	"	"	"	"	"	"	"	"	"	"	-	-	-	-	-	-	"	"	x	"	x	x	x	x	00	x	"	"
1c	"	"	"	"	"	"	"	"	"	"	"	***1***	-	-	-	-	-	"	"	x	"	x	x	x	x	"	1	"	"
1d	"	"	"	"	"	"	"	"	"	"	"	***1***	-	-	-	-	-	"	"	x	"	x	x	x	x	"	"	"	"
2a	RECV# $2,P1	RECV	x	x	x	x	2	1	1	RECV	1	-	-	***1***	-	-	-	-2	1	x	1	x	x	x	x	x	x	"	0
2a	"	"	"	"	"	"	"	"	"	"	"	-	-	***1***	-	-	-	"	"	x	"	x	x	x	x	00	x	"	"
2c	"	"	"	"	"	"	"	"	"	"	"	***2***	-	***1***	-	-	-	"	"	x	"	x	x	x	x	"	1	"	"
2d	"	"	"	"	"	"	"	"	"	"	"	2	-	1	-	-	-	0	"	x	"	x	x	x	x	"	"	"	1
3a	SLLI $3,$1,3	SLLI	1	0	x	x	3	3	0	SLL	2	2	-	1	-	-	-	-1	x	3	2	x	x	x	x	x	x	"	0
3b	"	"	"	"	"	"	"	"	"	"	"	2	-	1	-	-	-	1	x	"	"	1	x	0	x	00	x	"	"
3c	"	"	"	"	"	"	"	"	"	"	"	2	***3***	1	-	-	-	"	x	"	"	"	x	"	x	"	1	"	"
3d	"	"	"	"	"	"	"	"	"	"	"	2	***3***	1	-	-	-	"	x	"	"	"	x	"	x	"	"	"	"
4a	ADDI $4,$2,21	ADDI	2	0	x	x	4	21	0	ADD	2	2	-	1	***3***	-	-	0	x	21	2	x	x	x	x	x	x	"	0
4b	"	"	"	"	"	"	"	"	"	"	"	2	-	1	***3***	-	-	"	x	"	"	0	x	0	x	00	x	"	"
4c	"	"	"	"	"	"	"	"	"	"	"	2	***4***	1	***3***	-	-	"	x	"	"	"	x	"	x	"	1	"	"
4d	"	"	"	"	"	"	"	"	"	"	"	2	***4***	1	***3***	-	-	"	x	"	"	"	x	"	x	"	"	"	"
5a	SUBI# $2,$2!,37	SUBI	2	1	x	x	2	37	1	SUB	2	2	-	1	***4***	***3***	-	-1	x	37	2	x	x	x	x	x	x	"	0
5b	"	"	"	"	"	"	"	"	"	"	"	-	-	1	***4***	***3***	-	"	x	"	"	0	x	0	x	01	x	"	"
5c	"	"	"	"	"	"	"	"	"	"	"	***2***	-	1	***4***	***3***	-	"	x	"	"	"	x	"	x	"	1	"	"
5d	"	"	"	"	"	"	"	"	"	"	"	2	-	1	4	3	-	"	x	"	"	"	x	"	x	"	"	"	1
6a	ADD $1,$1!,$3!	ADD	1	1	3	1	1	x	0	ADD	0	2	-	1	4	3	-	-1	x	x	0	x	x	x	x	x	x	"	0
6b	"	"	"	"	"	"	"	"	"	"	"	2	-	-	4	-	-	2	x	x	"	1	2	0	0	11	x	"	"
6c	"	"	"	"	"	"	"	"	"	"	"	2	***1***	-	4	-	-	"	x	x	"	"	"	"	"	"	1	"	"
6d	"	"	"	"	"	"	"	"	"	"	"	2	***1***	-	4	-	-	"	x	x	"	"	"	"	"	"	"	"	"
7a	SUB# $2,$4!,$2!	SUB	4	1	2	1	2	x	1	SUB	0	2	-	***1***	4	-	-	1	x	x	0	x	x	x	x	x	x	"	0
7b	"	"	"	"	"	"	"	"	"	"	"	-	-	1	-	-	-	"	x	x	"	1	0	1	0	11	x	"	"
7c	"	"	"	"	"	"	"	"	"	"	"	***2***	-	***1***	-	-	-	"	x	x	"	"	"	"	"	"	1	"	"
7d	"	"	"	"	"	"	"	"	"	"	"	***2***	-	***1***	-	-	-	1	x	x	"	"	"	"	"	"	"	"	1
8d	"	"	"	"	"	"	"	"	"	"	"	2	-	1	-	-	-	0	x	x	"	"	"	"	"	"	"	1	"
9a	SRA# $1,$1!,$2!	SRA	1	1	2	1	1	x	1	SRA	0	2	-	1	-	-	-	-1	x	x	0	x	x	x	x	x	x	0	0
9b	"	"	"	"	"	"	"	"	"	"	"	-	-	-	-	-	-	1	x	x	"	1	0	0	0	11	x	"	"
9c	"	"	"	"	"	"	"	"	"	"	"	***1***	-	-	-	-	-	"	x	x	"	"	"	"	"	"	1	"	"
9d	"	"	"	"	"	"	"	"	"	"	"	***1***	-	-	-	-	-	1	x	x	"	"	"	"	"	"	"	"	1
10d	"	"	"	"	"	"	"	"	"	"	"	1	-	-	-	-	-	0	x	x	"	"	"	"	"	"	"	1	1
11a	SEND# P16,$1!	SEND	1	1	x	x	x	16	1	SEND	1	1	-	-	-	-	-	-1	16	x	1	x	x	x	x	x	x	0	0
11b	"	"	"	"	"	"	"	"	"	"	"	-	-	-	-	-	-	"	"	x	"	x	0	x	0	01	x	"	"
11c	"	"	"	"	"	"	"	"	"	"	"	-	-	-	-	-	-	"	"	x	"	x	"	x	"	"	0	"	"
11d	"	"	"	"	"	"	"	"	"	"	"	-	-	-	-	-	-	"	"	x	"	x	"	x	"	"	"	"	1

Table 6.5: RISA CTE Example - Config Stations

Configuration Stations — Config Station 0

	FN	AM1	AM2	OS1	OS2	RS1	RS2	WS	RT 0	RT 1	RKx	WE
1a	x	x	x	x	x	x	x	xx	0	0	xx	x
1b	x	x	x	x	x	x	x	xx	0	0	xx	x
1c	RECV	0	x	Imm	x	x	x	xx	0	0	00	1
1d	"	"	x	"	x	x	x	xx	0	0	"	"
2a	x	x	x	x	x	x	x	xx	0	0	xx	x
2a	x	x	x	x	x	x	x	xx	0	0	xx	x
2c	RECV	1	x	Imm	x	x	x	xx	0	0	00	1
2d	RECV	1	x	Imm	x	x	x	01	1	0	"	"
3a	x	x	x	x	x	x	x	x	1	0	xx	x
3b	x	x	x	x	x	x	x	x	1	0	xx	x
3c	SLL	x	3	+1	Imm	0	x	x	1	0	00	1
3d	"	x	"	"	"	"	x	x	1	0	"	"
4a	x	x	x	x	x	x	x	xx	1	0	xx	x
4b	x	x	x	x	x	x	x	xx	1	0	xx	x
4c	ADD	x	21	0	Imm	0	x	xx	1	0	00	1
4d	"	x	"	"	"	"	x	xx	1	0	"	"
5a	x	x	x	x	x	x	x	xx	1	0	xx	x
5b	x	x	x	x	x	x	x	xx	1	0	xx	x
5c	SUB	x	37	0	Imm	0	x	xx	1	0	01	1
5d	SUB	x	37	0	Imm	0	x	01	1	0	01	1
6a	x	x	x	x	x	x	x	xx	1	0	xx	x
6b	x	x	x	x	x	x	x	xx	1	0	xx	x
6c	ADD	x	x	1	2	0	0	xx	1	0	11	1
6d	ADD	x	x	1	2	0	0	xx	1	0	11	1
7a	x	x	x	x	x	x	x	xx	1	0	xx	x
7b	x	x	x	x	x	x	x	xx	1	0	xx	x
7c	SUB	x	x	0	1	0	0	xx	1	0	11	1
7d	SUB	x	x	0	1	0	0	xx	1	0	11	1
8d	SUB	x	x	0	1	0	0	01	1	0	11	1
9a	x	x	x	x	x	x	x	xx	0	0	xx	x
9b	x	x	x	x	x	x	x	xx	0	0	xx	x
9c	SRA	x	x	1	0	0	0	xx	0	0	11	1
9d	SRA	x	x	1	0	0	0	xx	0	0	11	1
10d	SRA	x	x	1	0	0	0	01	1	0	11	1
11a	x	x	x	x	x	x	x	xx	1	0	xx	x
11b	x	x	x	x	x	x	x	xx	1	0	xx	x
11c	SEND	16	x	Imm	0	x	0	xx	1	0	01	0
11d	SEND	x	x	Imm	0	x	0	00	0	0	01	0

Configuration Stations — Config Station 1

	FN	AM1	AM2	OS1	OS2	RS1	RS2	WS	RT 0	RT 1	RK	WE
1a	x	x	x	x	x	x	x	xx	0	0	xx	x
1b	x	x	x	x	x	x	x	xx	0	0	xx	x
1c	x	x	x	x	x	x	x	xx	0	0	xx	x
1d	x	x	x	x	x	x	x	xx	0	0	xx	x
2a	x	x	x	x	x	x	x	xx	0	0	xx	x
2a	x	x	x	x	x	x	x	xx	0	0	xx	x
2c	RECV	0	x	Imm	x	x	x	xx	0	0	00	1
2d	RECV	0	x	Imm	x	x	x	01	1	0	"	"
3a	x	x	x	x	x	x	x	xx	1	0	xx	x
3b	x	x	x	x	x	x	x	xx	1	0	xx	x
3c	x	x	x	x	x	x	x	xx	1	0	xx	x
3d	x	x	x	x	x	x	x	xx	1	0	xx	x
4a	x	x	x	x	x	x	x	xx	1	0	xx	x
4b	x	x	x	x	x	x	x	xx	1	0	xx	x
4c	SLL	x	3	0	Imm	0	x	xx	1	0	00	1
4d	"	x	"	0	Imm	0	x	xx	1	0	00	1
5a	x	x	x	x	x	x	x	xx	1	0	xx	x
5b	x	x	x	x	x	x	x	xx	1	0	xx	x
5c	ADD	x	21	-1	Imm	0	x	xx	1	0	00	1
5d	ADD	x	21	-1	Imm	0	x	10	1	1	00	1
6a	x	x	x	x	x	x	x	xx	1	1	xx	x
6b	x	x	x	x	x	x	x	xx	1	1	xx	x
6c	x	x	x	x	x	x	x	xx	1	1	xx	x
6d	x	x	x	x	x	x	x	xx	1	1	xx	x
7a	x	x	x	x	x	x	x	xx	1	1	xx	x
7b	x	x	x	x	x	x	x	xx	1	1	xx	x
7c	ADD	x	x	0	1	0	0	xx	1	0	11	1
7d	ADD	x	x	0	1	0	0	xx	1	0	11	1
8d	ADD	x	x	0	1	0	0	01	1	0	11	1
9a	x	x	x	x	x	x	x	xx	0	0	xx	x
9b	x	x	x	x	x	x	x	xx	0	0	xx	x
9c	x	x	x	x	x	x	x	xx	0	0	xx	x
9d	x	x	x	x	x	x	x	xx	1	0	xx	x
10d	x	x	x	x	x	x	x	xx	0	0	xx	x
11a	x	x	x	x	x	x	x	xx	0	0	xx	x
11b	x	x	x	x	x	x	x	xx	0	0	xx	x
11c	x	x	x	x	x	x	x	xx	0	0	xx	x
11d	x	x	x	x	x	x	x	xx	0	0	xx	x

Configuration Stations — Config Station 2

	FN	AM1	AM2	OS1	OS2	RS1	RS2	WS	RT 0	RT 1	RK	WE
1a	x	x	x	x	x	x	x	xx	0	0	xx	x
1b	x	x	x	x	x	x	x	xx	0	0	xx	x
1c	x	x	x	x	x	x	x	xx	0	0	xx	x
1d	x	x	x	x	x	x	x	xx	0	0	xx	x
2a	x	x	x	x	x	x	x	xx	0	0	xx	x
2a	x	x	x	x	x	x	x	xx	0	0	xx	x
2c	x	x	x	x	x	x	x	xx	0	0	xx	x
2d	x	x	x	x	x	x	x	xx	0	0	xx	x
3a	x	x	x	x	x	x	x	xx	0	0	xx	x
3b	x	x	x	x	x	x	x	xx	0	0	xx	x
3c	x	x	x	x	x	x	x	xx	0	0	xx	x
3d	x	x	x	x	x	x	x	xx	0	0	xx	x
4a	x	x	x	x	x	x	x	xx	0	0	xx	x
4b	x	x	x	x	x	x	x	xx	0	0	xx	x
4c	x	x	x	x	x	x	x	xx	0	0	xx	x
4d	x	x	x	x	x	x	x	xx	0	0	xx	x
5a	x	x	x	x	x	x	x	xx	0	0	xx	x
5b	x	x	x	x	x	x	x	xx	0	0	xx	x
5c	SLL	x	3	-1	Imm	0	x	xx	0	0	00	1
5d	SLL	x	3	-1	Imm	0	x	01	1	0	00	1
6a	x	x	x	x	x	x	x	xx	1	0	xx	x
6b	x	x	x	x	x	x	x	xx	1	0	xx	x
6c	x	x	x	x	x	x	x	xx	1	0	xx	x
6d	x	x	x	x	x	x	x	xx	1	0	xx	x
7a	x	x	x	x	x	x	x	xx	1	0	xx	x
7b	x	x	x	x	x	x	x	xx	1	0	xx	x
7c	x	x	x	x	x	x	x	xx	1	0	xx	x
7d	x	x	x	x	x	x	x	xx	1	0	xx	x
8d	x	x	x	x	x	x	x	xx	0	0	xx	x
9a	x	x	x	x	x	x	x	xx	0	0	xx	x
9b	x	x	x	x	x	x	x	xx	0	0	xx	x
9c	x	x	x	x	x	x	x	xx	0	0	xx	x
9d	x	x	x	x	x	x	x	xx	0	0	xx	x
10d	x	x	x	x	x	x	x	xx	0	0	xx	x
11a	x	x	x	x	x	x	x	xx	0	0	xx	x
11b	x	x	x	x	x	x	x	xx	0	0	xx	x
11c	x	x	x	x	x	x	x	xx	0	0	xx	x
11d	x	x	x	x	x	x	x	xx	0	0	xx	x

lookup returns the ALU function “RECV” and an immediate as the second operand (iN = 2). The controller outputs the corresponding immediate value and control signal (IM2 = 0 and IC = 2). The register table is empty, so incrementing the column locations does nothing. In the next stage, stage **b**, nothing happens since there are no source registers to be processed. In stage **c**, since there is a destination register designated, the WE flag is set and the register number (1) is inserted in the register table. Since it cannot be determined where the register value will be stored in the fabric yet, this is still a “live” register value. For compactness, the register value is shown in one of the register locations in the table; in actuality, it would be held in a separate entry in the table for ’live’ register values. In the configuration station, the configuration values relating to source operands are set; for this example, this means OS1 is set to ”Imm” and AM2 is set to value “0”. Since the new row signal is not set, nothing happens in stage **d**.

The second instruction is RECV# $2,P1, is identical to the first instruction, except that the port number (immediate value) is 1 and in that the new row bit is set, meaning that this instruction is the last one on this row. Note that the previous instruction has been shifted to configuration station 1 and configuration station 0 is ready for this instruction. In stage **a**, as for the first instruction, the instruction lookup returns the ALU function “RECV” and an immediate as the second operand (iN = 2). The controller outputs the corresponding immediate value and control signal (IM2 = 1 and IC = 2). The live register in the register table, register 1, is moved to the next column location, column 1. In stage **b**, again nothing happens since there are no source registers to be processed. In stage **c**, since there is a destination register, the WE flag is set and the register number (2) is inserted in the register table as a live register in column 0. In the configuration station, OS1 is set to ”Imm” and AM2 is set to value “1”. Finally, in stage **d**, the configuration can be finished for the row. All live registers in the issue station register table are now placed into the first open register in the column in which the live register is located. In the example, the live register 1 is in column 1; the first register open in column 1 is register 0, so register 1 is stored in the register table at column 1, register 0. A normal font is used to indicate that this is now a valid register table entry, not a live register. Similarly, register 2 is stored in column 0, register 0. In the configuration stations, wherever WE is set an entry is stored in the first open register. Note that these entries only indicate that the register is in use, but not which SPU register is stored in that location. In both column 0 and column 1 WE is set, so the

first open register in each is set to 1; register 0 is open for both. The write select field is then set to correspond to the first open register found previously, using a one-hot encoding. Since in both columns register 0 is being written to, both WS fields are set to "01", which indicates that register 0 will be written to. At this point, the entire configuration for the first row is complete and is shown in the gray boxes on line 2d in Table 6.7.

The next instruction is SLLI \$3, \$1, \$3, the first instruction in this example with a source register. This instruction in the SPU takes the value in register \$1, shifts it left 3 places and places the result in register \$3. It has an opcode of SLLI, SR1=1, RK1=0, DR = 3, and an immediate value of 3. In stage **a,** the lookup results are FN = "SLL" and an immediate as the second operand (iN =2). The controller outputs the corresponding immediate value and control signal (IM2 = 3 and IC = 2). There are no live registers, so nothing changes in the RT. In stage **b,** the only source register, register \$1, is found in the register table at LC1= 1 and RC1 = 0. Since RK1 is not set, nothing is changed in the register table. In stage **c**, since there is a destination register, the WE flag is set and the register number (3) is inserted in the register table as a live register in column 0. In the configuration station, OS1 is set to LC1 to +1, OS2 is set to "Imm" and AM2 is set to value "3". Since NR is not set, nothing happens in stage **d**.

The next instruction is ADDI \$4, \$2, 21. As before, the previous instruction has shifted to the next configuration station. The shifted value for OS1 has been decremented to 0. This instruction in the SPU takes the value in register \$4, adds 21, and places the result in register \$3. It has an opcode of ADDI, SR1=2, RK1=0, DR = 4, and an immediate value of 21. In stage **a,** the lookup results are FN = "ADD" and an immediate as the second operand (iN =2). The controller outputs the corresponding immediate value and control signal (IM2= 21 and IC = 2). The location for live register 3 is shifted to column 1. In stage **b,** the only source register, register \$2, is found in the register table at LC1= 0 and RS1 = 0. Since RK1 is not set, nothing is changed in the register table. In stage **c**, since there is a destination register, the WE flag is set and the register number (4) is inserted in the register table as a live register in column 0. In the configuration station, OS1 is set to LC1=0, OS2 is set to "Imm", RS1 is 0 and AM2 is set to value "21". Since NR is not set, nothing happens in stage **d**.

The last instruction on this row is SUBI# \$2, \$2!, 37. This instruction in the SPU takes the value in register \$2, subtracts 37, and places the result back in register \$2. It has an opcode of

SUBI, SR1=2, RK1=1, DR = 2, and an immediate value of 37. In stage **a,** the lookup results are FN = "SUB" and an immediate as the second operand (iN =2). The controller outputs the corresponding immediate value and control signal (IM2= 37 and IC = 2). The location for live register 3 is shifted to column 2 and the location for live register 4 is shifted to column 1. In stage **b,** the only source register, register $2, is found in the register table at LC1= 0 and RS1 = 0. Since RK1 is set, register 2 is cleared from the register table. Note that in the RCF, the value stored in column 0, register 0 could be still be read by other tiles in the same row. However, in order to maintain correspondence between the SPU and RCF, register assignment in the HASTE tool flow will not allow access to killed registers in the same row. Since register $2 is killed, it cannot be read until there is a new row. In can be written to however, as in this example. In stage **c** the WE flag is set and the register number (2) is inserted back in the register table as a live register in column 0. It is live, so it cannot be found in the table until the locations are fixed by a new row. In configuration station 0, OS1 is set to LC1=0, OS2 is set to "Imm", RS1 is set to 0, and AM2 is set to value "21". Since NR is set, in stage **d** register locations are fixed in the RT; register $2 is in column 1, register 0, which happens to be its previous location in this example. Register $4 is in column 1; the first open register 1, so it is placed there. Register $3 is in column 2 and is placed in register 0. Register $1 remains in its previous location, since it has not been killed. In the configuration stations, any station with WE enabled sets the first open register location and selects that register for writing in WS, as discussed previously.

6.3 Relative Register ISA SPU and CTE Operation

6.3.1 RRISA SPU Operation for Example

The RRISA code for the example is shown in Figure 6.2(d) and the corresponding DFG is shown in Figure 6.3(c). This code sequence implements the loop body for the pseudo-code shown in Figure 6.2(a). As with the previous examples, all input and output to the loop body is carried by streams, with the loop variable i, and X[i] and Y[i] representing input streams and Z[i] representing an output stream. As in previous examples, MAU initialization is assumed to have been already completed.

Even though there are no columns *per se* in the RRISA SPU, registers are designated using

relative column notation. As discussed in Chapter 4, RRISA uses register designations that correspond to the RCF and uses special hardware in the SPU, in the form of a special register file and an instruction numbering queue, to use these register designations during SPU operation. This is in contrast to RISA, which uses a more conventional register designation that corresponds directly to the SPU and which must be translated into RCF register locations in the CTE. During non-kernel RRISA SPU operation, more conventional register designations are used as well; only during kernel execution is the register designation shown here used. In non-kernel instructions, a column offset of 0 is always used and the column location is assumed to be zero for all instructions. The sliding register file used in the SPU has 64 registers in column 0, regardless of the number actually present in the RCF fabric. This allows the SPU to function just as the RISA SPU for non-kernel code, with a conventional 64-entry register file. When running kernel code, other column locations are used in the sliding register file, and these columns have only as many registers as are actually present in each column of the RCF.

One complexity of the RRISA SPU arises from the fact that it is preferable to issue instructions into the CTE such that they can be configured and pushed onto the fabric in order, as was done for QISA and RISA. This means that the final column an instruction will be in is not known until the last instruction in a row is seen. This is not a problem for the RISA and QISA SPUs, since is it not necessary to know the column an instruction will be in in the CTE in order to correctly execute it in the SPU. It is necessary to know the column for correct execution in the SPU for RRISA, however, since register locations are relative to absolute column locations. Therefore, new row markers are located in the SPU instruction fetch queue and column locations are assigned to kernel instructions based on the number of instructions in the row. The first instruction found in a row is then known to be at column n-1, where n is the number of instructions in that row, the next instruction is at n-2, and so on, down to column 0.

In Figure 6.11 the contents of the SPU registers after the execution of each instruction are shown. The column location, found as described above, is shown in parentheses next to each instruction. Since mapping is being done to a fabric with three columns and two registers per column, six registers are shown for the corresponding SPU register file (the extra registers associated with column 0 for use with non-kernel code are not shown). Only four of these registers are actually used by the loop body example (or corresponding fabric configuration) and

it is assumed that there is nothing significant in any of the registers at the start of the loop body. Each letter corresponds to a variable in Figure 6.2(a) and is shown in the register where it is stored. Registers shown with a gray background correspond to values that were produced by the preceding instruction. Registers surrounded by a thick outline show register locations that were read by the preceding instruction, although a new value may have also been placed in that location (and will have a gray background if so). Starting in the upper left corner, this figure shows the loop body starting with an empty register file. The first instruction, RECV :0,P0 is at column 1, and causes the SPU to receive a value, A, from MAU port P0, and place it in register 0 in the same column, column 1. The next instruction, RECV# :0,P1, is at column 0, and causes the SPU to receive a value, B, from MAU port P1, and places it in register 0 of the same column, column 0. The '#' character indicates that this is the last instruction on a row. The column counter is then set to the number of instruction in the current row minus 1; 2, in this case.

Next is the first arithmetic instruction, SLLI -1:1,-1:0,3, which reads the value A from register 0 one column to the right; i.e., register 0 of column 1, performs a logical shift left by three places, and then places the new value, C, in register 1 of column 1. The next instruction, ADDI -1:1,-1:0,21, reads value B from register 0 in column 0 and adds 21, placing the resultant sum, value D, in register 1 of column 0. Similarly, the next instruction, SUBI# :0, :0,37 reads value B from register 0 of column 0 again and subtracts 37, giving the new value E which it places in the same register. Note that this instruction is the last instruction to use value B, which was stored in register 0 of column 0. The compiler realizes this and therefore allocates the same register for value E. Note that a '!' is not needed to indicate this as was done for RISA, since the register allocations for the CTE are made explicitly. The '#' after SUBI indicates that is the last instruction on the current row, as was the case for the second RECV instruction.

The first instruction in the row is in column 1 and is a two-operand instruction, ADD :0,:1,:0, which sums the two values C and A, and places the sum (value F) in register 0 of column 1. Note that values that aren't used again, such as C, are shown in gray text in the register file. Note that A was written on the first row, used previously on the second row, and is used here on the third row of the DFG and then overwritten by value F. The next instruction, SUB# :0,:1,:0 is also a two-operand instruction, subtracting E in register 0 of column 0 from D in register 1 of column 0, and placing the result in register 0 of column 0. The next instruction is SRA# :0,+1:0,:0,

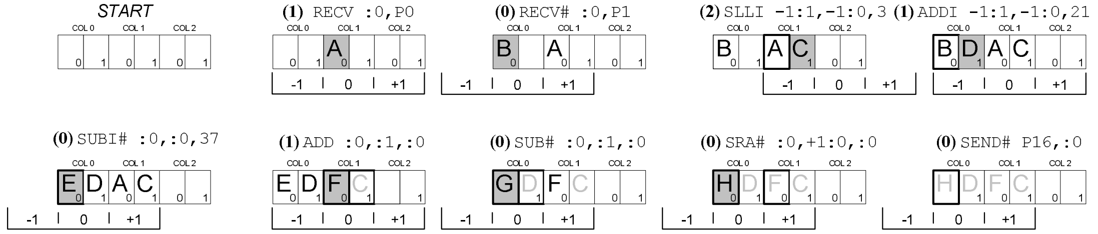


Figure 6.11: Register Contents for RRISA SPU Implementation

which implements the expressions $H = F >> G$, and is otherwise the same as the preceding instruction. The final instruction is a SEND# P16, :0, which sends the value in register 0 of column 0, H, to MAU port 16. This register is read for the last time by this instruction. Now the next and all succeeding loop iterations can proceed exactly as the first iteration did, if the loop body is running solely on the SPU. This loop body can run on the RCF, however, so in normal operation, while the SPU runs the first loop iteration, the CTE is creating the configuration, as will be described in the next section, and succeeding iterations will run on the RCF, just as was seen in the QISA and RISA examples.

6.3.2 RRISA CTE Operation

The symmetric pass register fabric is used with the relative register ISA and is shown in Figure 5.7. It has all of the same fields as the symmetric pass register fabric: the ALU function select field FUNC, two immediate value fields AM1 and AM2, and three operand select values OS1, OS2, and OS3, the read select fields, RSn, and the write select field WS. The read select fields work just as they did for the RISA fabric as previously discussed. The write select fields are now more complicated in that each tile may write to registers in other columns. As with the read select fields, values are stored in the columns containing the ALUs doing the writing and not the columns containing the registers being written. Since each ALU can only write to one location, only one such write select field is needed for each column. The write control hardware of the asymmetric pass register fabric is thus necessarily rather different than the simpler hardware used in the symmetric pass register fabric. If a register is not written to, it receives the value in the corresponding register in the previous row, just as for any other pass register fabric. In Figure 6.5(b), the locations of the PE configuration fields for the example fabric shown. The

Table 6.6: CTE Inputs from SPU and CTE Outputs to RCF for RISA

Input Field	Description	Output Field	Description
OP	Instruction Opcode	FUNC (FN)	Function select for ALU.
DR	Destination register.	AIMM1 (AM1)	Immediate value 1 for ALU.
DO	Destination offset	AIMM2 (AM2)	Immediate value 2 for ALU.
S1R, S2R, S3R	Source registers.	OS1SEL (OS1)	Operand select for ALU input 1.
S1O, S2O, S3O	Source offsets.	OS2SEL (OS2)	Operand select for ALU input 2.
NR	New row field.	OS3SEL (OS3)	Operand select for ALU input 3.
IMM	Instruction immediate, shift amount or port number	RSELn (RSn)	Register read selects.
LR1, LR2	Live-in register values.	WSEL (WS)	Register write select.

complete configuration is shown in Figure 6.13.

As with RISA, the inputs to the CTE are the current instruction and the current values of the first two source registers, LR1 and LR2. The instruction itself contains the instruction opcode field OP; a destination register and offset, DR and DO; source registers S1R, S2R, and S3R, and source offsets S1O, S2O, and S3O; a new row field NR (this corresponds to the '#' character which indicates a new row); and an immediate field IMM. No register kill fields are needed, since register usage is assigned explicitly in both the CTE and SPU. In RRISA code, registers are specified in an *offset*:*register* format; if the offset is 0, it is not shown, for brevity. The input and output fields are shown in Table 6.6, with both full and abbreviated forms shown. For the register specification fields, (S1R, S2R, S3R, and DR) a -1 value indicates that a register is not specified; this usually represents the largest possible value for that field, which cannot be a valid register by convention. The SPU decoder fills in this value as needed, depending on the instruction opcode and instruction format.

The hardware comprising the CTE is shown in Figure 6.12. It is simpler than either the QISA and RISA CTEs. Since operand locations are specified explicitly, there is no hardware needed for tracking operand locations, such as the operand queues in the QISA CTE and the register table in the RISA CTE (there is a set of register flags in the Immediate Control logic that tracks whether a register has been written to in the kernel). The only hardware required in the entire CTE is for the handling of immediate values. The instruction lookup gives the base FN code for the OP and determines if an immediate is needed. If so, the immediate control supplies the immediate to the correct immediate field and sets the immediate control (IC) signal appropriately. The

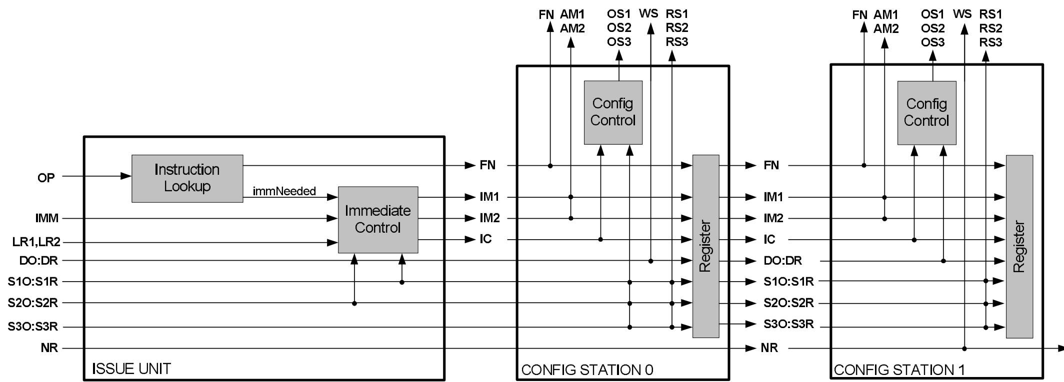


Figure 6.12: Relative Register ISA CTE

immediate control logic also checks if a register location is used for the first time; if so, the live-in value is supplied as an immediate. Logic in the config stations sets the operand select signals according to the immediate control value and source offset field. Other than those few fields, all other configuration information comes directly from instruction itself. Algorithm 4 shows pseudo-code for the register ISA CTE algorithm. As indicated, portions of the code represent the functionality of the issue unit and portions the functionality of the configuration stations.

6.3.3 RRISA CTE Operation for Example

The process of converting this code into a configuration is shown in Table 6.7. This process is the implementation of Algorithm 4 using the hardware shown in Figure 6.12 and will be explained in detail in this section. As was the case for the RISA and QISA examples, Table 6.7 shows the value of important signals throughout the conversion process. Each row shows the values for one stage of the conversion process, each stage corresponding to a shaded box in Algorithm 1. Unlike the previous examples, the algorithm is simple enough to require only two stages, instead of the four requires by QISA and RISA. Stage **a** involves reading the lookup table and the setting of immediate values and the immediate control signal in the issue station. Stage **b** involves the generation of the final configurations in the configuration stations. Note that stage **a** take place in the issue unit and stage **b** takes place subsequently in each configuration station.

Algorithm 4 RRISA CTE Algorithm

```
Issue Unit:
Inputs    : instruction = [OP,  IMM, S1O:S1R, S2O:S2R, S3O:S3R,DO:DR, NR], LR1, LR2
Outputs  : FN, IM1, IM2, IC, S1O:S1R, S2O:S2R, S3O:S3R,DO:DR, NR

clear RegisterUsageFlags
for each instruction in kernel
        IC = 0;
        [FN, immNeeded] <= lookup(instruction.OP);
        if (immNeeded ==  1) then
                IM1 <= instruction.IMM
                IC <= IC + 1;
        else if (immNeeded ==  2) then
                IM2 <= instruction.IMM
                IC <= IC + 2;                                          a
        endif
        for n = 1..2
                if ((SnR != -1) AND (SnO+thisCol:SnR not in register table))
                                IC <= IC + n
                                IMn <= IRn
                endif
        end
end

Configuration Station:
Inputs: FN, IM1, IM2, IC, S1O:S1R, S2O:S2R, S3O:S3R,DO:DR, NR
Outputs: FN, AM1, AM2, OS1, OS2, OS3, RS1, RS2, RS3, WS

clear RT;
until done                                                             b
        OS1SEL <= (ICTRL[0] ? Imm : S1O);
        OS2SEL <= (ICTRL[1] ? Imm : S2O);
        OS3SEL <= S3O;
end
```

6.3.3.1 Table and Configuration Description

The leftmost column of each row identifies the stage and clock cycle represented by that row. The "Inst" column shows the instruction as sent to SPU. The next nine columns show all of the fields that can be sent from the SPU to the CTE. Not all of these fields will be used for any single instruction, as can be seen from the ISA formats as shown in Figure 4.10. The "OP" column shows the instruction opcode. The "S1R", "S1O", "S2R", and "S2O" columns give the register and offset fields for the first and second source registers. Note that since there are no three operand instructions in the example, a third source register is never needed and thus the "S3R" or "S3O" fields are not shown. There are also register and offsets fields "DR" and "DO" for the single destination register field "DR". There is an immediate field, "IMM", which is used both for 16-bit immediates for I-Type instructions, as well for 5-bit shift amounts and port numbers. Finally, the "NR" column holds the new row bit. The live-in values "LR1" and "LR2" would be shown in this section, but the example does not have any live-in values, so the corresponding columns were removed for brevity.

The next five columns show the signals and values in the issue unit itself (fields which are unchanged in the issue unit are not shown). The first two show the values returned by the instruction lookup based on the input opcode; "FN" is the function to be performed by the ALU for that instruction and "iN" signifies if an immediate is needed, and if so, whether it should be in the first or second operand location (immediates are not supported for the third operand location). The next three columns show the outputs of the Immediate Control, including both immediate values, "IM1" and "IM2", and the immediate control signal "IC".

The next three sets of eight columns each show the outputs and internal signals of each configuration station. Since the example is three columns wide when mapped to RRISA, only three issue stations are shown. For each station, the output values ALU function select "FN", ALU immediates "AM1" and "AM2", operand selects "OS1" and "OS2", register selects "RS1" and "RS2", and write select "WS" are shown. As always, signals referring to the third input operand are not shown. The fabric configuration produced in the example is shown in Figure 6.13. This figure uses the same notation as did the RISA example; refer to Figure 6.5(b) for details.

Table 6.7: RRISA CTE Example

	Inst	CTE Inputs									Issue Unit: Inst. Lookup		Issue Unit: Immediate Control		
		OP	SR1	SO1	SR2	SO2	DR	DO	IMM	NR	FUNC	IN	IM1	IM2	IC
1a	RECV :0, P0	RECV	x	x	x	x	0	0	0	0	RECV	1	0	x	1
1b	"	"	"	"	"	"	"	"	"	"	"	"	0	x	1
2a	RECV# :0, P1	RECV	x	x	x	x	0	0	1	1	RECV	1	1	x	1
1d	"	"	"	"	"	"	"	"	"	"	"	"	1	x	1
3a	SLLI -1:1,-1:0,3	SLLI	0	-1	x	x	1	-1	3	0	SLL	2	x	3	2
3b	"	"	"	"	"	"	"	"	"	"	"	"	x	3	2
4a	ADDI -1:1,-1:0,21	ADDI	0	-1	x	x	1	-1	21	0	ADD	2	x	21	2
4b	"	"	"	"	"	"	"	"	"	"	"	"	x	21	2
5a	SUBI# :0,:0,37	SUBI	0	0	x	x	0	0	37	1	SUB	2	x	37	2
5b	"	"	"	"	"	"	"	"	"	"	"	"	x	37	2
6a	ADD :0,:1,:0	ADD	1	0	0	0	0	0	x	0	ADD	0	x	x	0
6b	"	"	"	"	"	"	"	"	"	"	"	"	x	x	0
7a	SUB# :0,:1,:0	SUBI	1	0	0	0	0	0	x	1	SUB	0	x	x	0
7b	"	"	"	"	"	"	"	"	"	"	"	"	x	x	0
8a	SRA# :0,+1:0,:0	SRA	0	1	0	0	0	0	x	1	SRA	0	x	x	0
8b	"	"	"	"	"	"	"	"	"	"	"	"	x	x	0
9a	SEND# P16,:0	SEND	0	0	x	x	x	x	16	1	SEND	16	16	x	1
9b	"	"	"	"	"	"	"	"	"	"	"	"	16	x	1

	Configuration Stations: Config Station 0								Config Station 1								Config Station 2							
	FUNC	AM1	AM2	OS1	OS2	RS1	RS2	WS	FN	AM1	AM2	OS1	OS2	RS1	RS2	WS	FN	AM1	AM2	OS1	OS2	RS1	RS2	WS
1a	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x
1b	RECV	0	x	Imm	x	x	x	0:0	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x
2a	x	x	x	x	x	x	x	x:x	RECV	0	x	Imm	x	x	x	0:0	x	x	x	x	x	x	x	x:x
1d	RECV	1	x	Imm	x	x	x	0:0	RECV	0	x	Imm	x	x	x	0:0	x	x	x	x	x	x	x	x:x
3a	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x
3b	SLL	x	3	-1	Imm	0	x	-1:1	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x
4a	x	x	x	x	x	x	x	x:x	SLL	x	3	-1	Imm	0	x	-1:1	x	x	x	x	x	x	x	x:x
4b	ADD	x	21	-1	Imm	0	x	-1:1	SLL	x	3	-1	Imm	0	x	-1:1	x	x	x	x	x	x	x	x:x
5a	x	x	x	x	x	x	x	x:x	ADD	x	21	-1	Imm	0	x	-1:1	SLL	x	3	-1	Imm	0	x	-1:1
5b	SUB	x	37	0	Imm	0	x	0:0	ADD	x	21	-1	Imm	0	x	-1:1	SLL	x	3	-1	Imm	0	x	-1:1
6a	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x
6b	ADD	x	x	0	0	1	0	0:0	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x
7a	x	x	x	x	x	x	x	x:x	ADD	x	x	0	0	1	0	x:x	x	x	x	x	x	x	x	x:x
7b	SUB	x	x	0	0	1	0	0:0	ADD	x	x	0	0	1	0	0:0	x	x	x	x	x	x	x	x:x
8a	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x
8b	SRA	x	x	+1	0	0	0	0:0	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x
9a	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x
9b	SEND	16	x	Imm	0	x	0	x:x	x	x	x	x	x	x	x	x:x	x	x	x	x	x	x	x	x:x

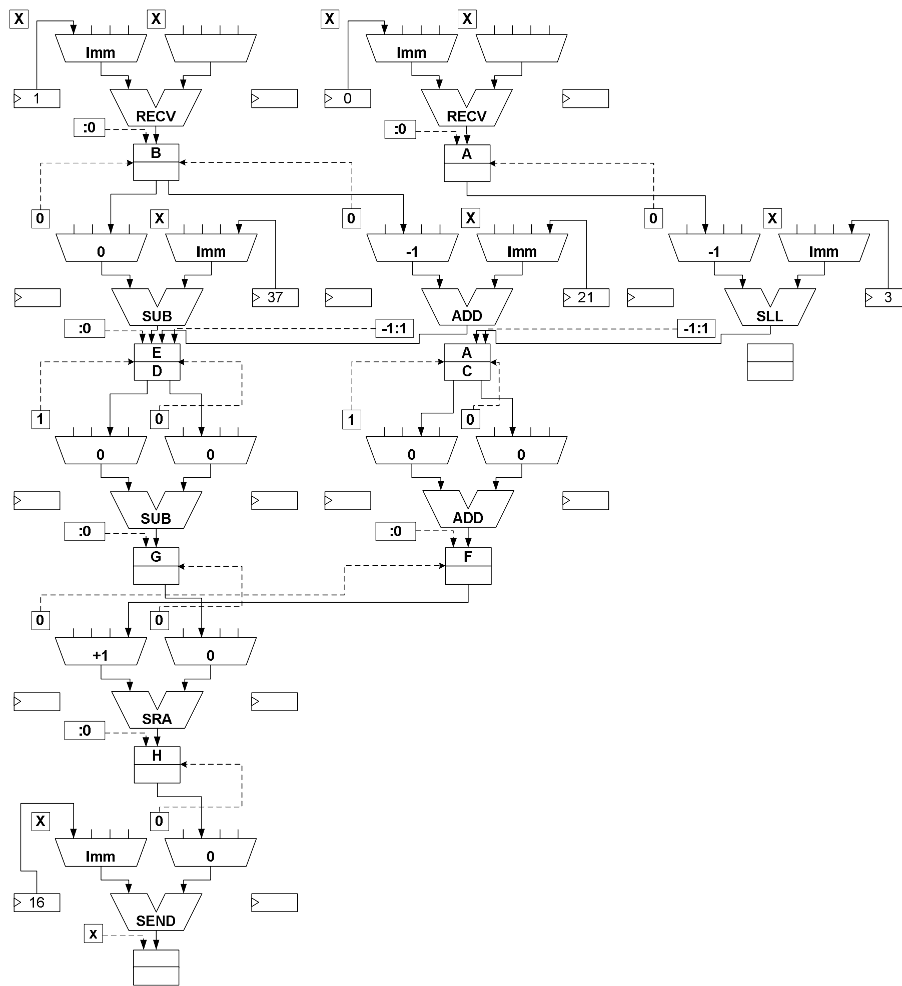


Figure 6.13: RRISA Fabric Configuration for Example

6.3.3.2 Processing of Example Instructions

As previously discussed, the processing of the example kernel is shown in Table 6.7 and the final configuration is shown in Figure 6.13. The first instruction is RECV :0, P0, which in the SPU receives a value from Port 0 of the MAU and places it in register 0 with no offset. In stage **a**, the instruction lookup returns the ALU function "RECV" and an immediate as the first operand (iN = 1). The controller outputs the corresponding immediate value and control signal (IM1 = 0 and IC = 1). In stage b, in the configuration station, the configuration values relating to source operands are set; for this example, this means OS1 is set to "Imm". The second instruction, RECV# :0,P1, is identical to the first instruction, except that the port number (immediate value) is 1 and in that the new row bit is set, meaning that this instruction is the last one on this row. Note that the previous instruction has been shifted to configuration station 1 and configuration station 0 is ready for this instruction. In stage **a**, as for the first instruction, the instruction lookup returns the ALU function "RECV" and an immediate as the first operand (iN = 1). The controller outputs the corresponding immediate value and control signal (IM1 = 1 and IC = 1). In stage b, the configuration, the OS1 value is set to "Imm" and configuration is finished for the row, as shown in the gray boxes on line 2b in Table 6.7.

The next instruction is SLLI -1:1, -1:0, 3 the first instruction in this example with a source register. This instruction in the SPU takes the value in register 0, at offset -1, shifts it left 3 places and places the result in register 1 at offset -1. In stage **a,** the lookup results are FN = "SLL" and an immediate as the second operand (iN =2). The controller outputs the corresponding immediate value and control signal (IM2 = 3 and IC = 2). In the configuration station in stage b, OS1 is set to S1O = -1, OS2 is set to "Imm" and AM2 is set to value "3". The next instruction is ADDI -1:1, -1:0, 21. As before, the previous instruction has shifted to the next configuration station. This instruction in the SPU takes the value in register 0 at offset -1, adds 21, and places the result in register 1 at offset -1. In stage **a,** the lookup results are FN = "ADD" and an immediate as the second operand (iN =2). The controller outputs the corresponding immediate value and control signal (IM2= 21 and IC = 2). In the configuration station, OS1 is set to S1O = -1, OS2 is set to "Imm", and AM2 is set to value "21". The last instruction on this row is SUBI# :0, :0, 37. This instruction in the SPU takes the value in register 0 of the current column, subtracts 37, and places the result back in register 0 of the current column. In stage

a, the lookup results are FN = "SUB" and an immediate as the second operand (iN =2). The controller outputs the corresponding immediate value and control signal (IM2= 37 and IC = 2). In configuration station 0, OS1 is set to S1O = 0, OS2 is set to "Imm", and AM2 is set to value "21". This instruction completes the current row, as shown in row 5b of Table 6.7.

6.4 Conclusions

In this chapter CTE designs for all three HASTE ISA were shown and their operation illustrated by examples. All three CTE designs meet the requirements outlined at the beginning of the chapter: they only require access to the first row of the fabric; they are scalable to fabrics of arbitrary size; they take roughly a single kernel iteration to produce a configuration (slightly more in the case of RRISA); and they require relatively simple hardware. While the QISA and RISA CTE require some unusual hardware, neither has unreasonable hardware requirements. The RRISA CTE is quite simple, with the most complicated component being the instruction lookup table. Functional models of all of these CTE designs have been tested for a range of RCF fabrics, showing that the relatively simple algorithms presented in this chapter do in fact produce correct kernel configurations. More discussion of this testing is presented in Chapter 9.

Chapter 7

Tool Flow and Simulation Environment

In order to evaluate different HASTE ISAs and show that the HASTE concept works, tools were needed to simulate the operation of HASTE systems. In addition tools were needed to map applications to different HASTE fabrics, and to evaluate and estimate the hardware implementations of kernels on HASTE, as well as competing technologies. In some cases, existing tools could be used, although usually some modifications needed to be made to these tools for use with HASTE. In other cases, custom tools had to be developed. Some of these tools work on entire applications, while others only work on application kernels. In a few cases, steps in the flow must be done manually.

The entire tool flow is shown in Figure 7.1. The broad arrows in the figure represent tools (or in some cases manual procedures), the rounded rectangles represent simulators, the "card stack" figures represent executables, and the rectangles with a wavy edge represent code or data of some type. The key in the lower left of the figure gives more information about the flow, including identification of which tools were created specifically for HASTE, which were modified versions of existing tools, and which were existing tools used as-is. The tool flow can be broken down into three sub-flows; application mapping, simulation and validation, and hardware implementation. Rather than discuss the overall tool flow as a unit, sections will detail each of the sub-flows and will cover all of the tools in the context of the sub-flows. Where there is overlap between the sub-flows, specific tools will be discussed in the most relevant section.

A number of sometimes conflicting requirements had to be considered when creating the tool flow for this project. The tools needed to be as flexible as possible, so that different ISAs

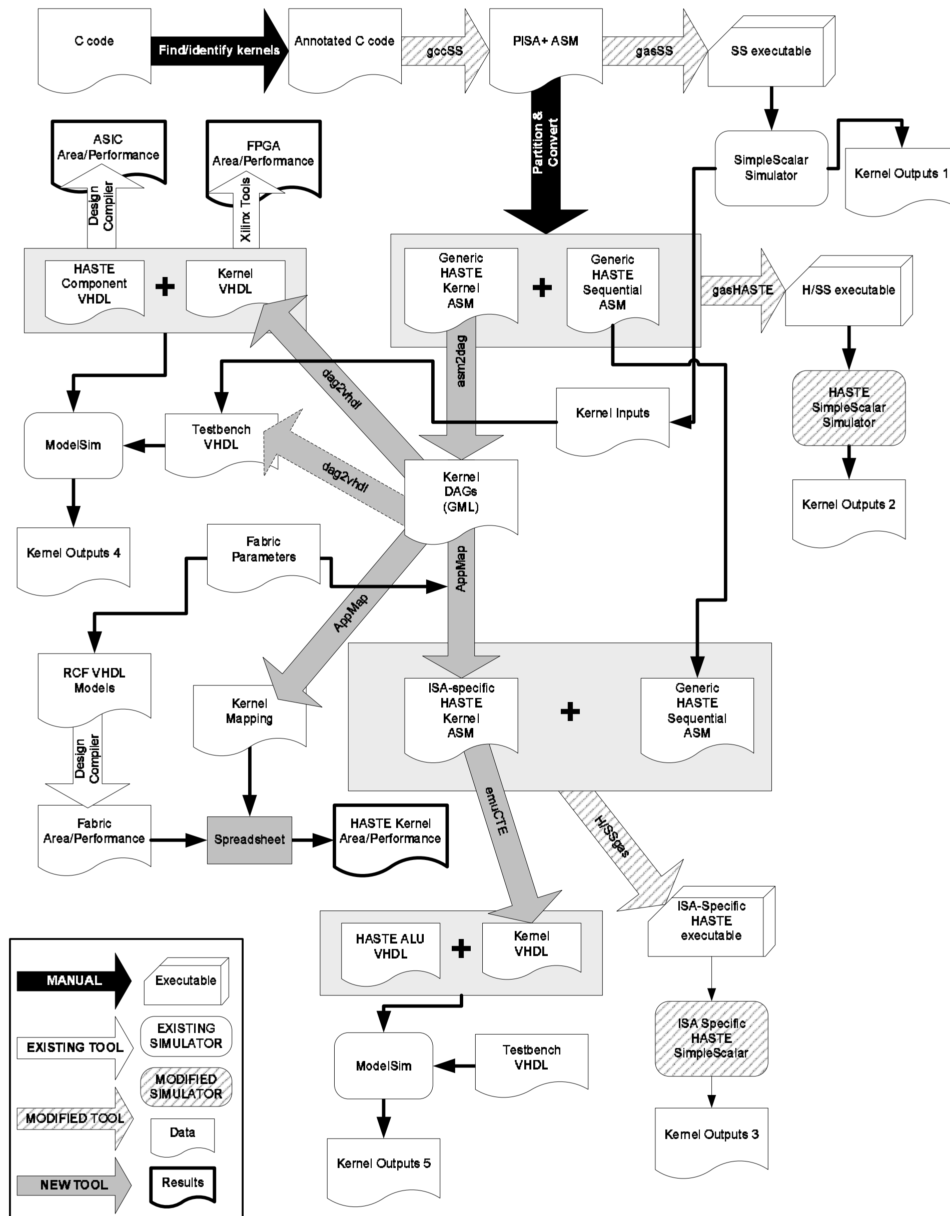


Figure 7.1: Overall HASTE Tool Flow

and fabrics could be studied and compared. The use of test applications and kernels from various sources was necessary to allow for comparison with other architectures and to ensure that reasonable and relevant benchmarks were available. The mapping of applications to hardware needed to be good enough to allow for valid comparisons between different implementations, but also fast enough to allow for performing many experiments. The fast mapping of kernels to many different architectures was deemed more important than being able to investigate a very large number of kernels, so a semi-manual compilation flow was deemed acceptable. Comparison of sequential code was not deemed important, as this was largely a traditional compiler problem and not necessary for exploration of the HASTE concept. Due to the limited resources available for tool development, existing tools were to be leveraged wherever possible.

The flow shown in Figure 7.1 succeeds in meeting the requirements of the project. Kernels can be captured directly from C source code, written in HASTE assembly code, or captured from directed acyclic graphs (DAGs) produced by other tools such as the DIL compiler [38] for PipeRench (DAGs, as a form of dataflow graph, are used in the HASTE tool flow). Code can be simulated at various points in the flow to verify kernel inputs and outputs. Code for any ISA can be represented as a generic HASTE assembly program and kernels can be represented as generic HASTE assembly or by an equivalent DAG. The generic DAG can then be converted for specific ISAs and mapped to a specific fabric. The generic DFG can also be converted to VHDL for synthesis for FPGA or ASIC implementations.

All of the tool development and modification of existing tools was done by the author, with three exceptions. Some portions of the AppMap code for queue ISAs were originally written by Benjamin Ylvisaker; these portions were subsequently entirely rewritten to match a newer set C++ classes and APIs, but a few of the original algorithms were retained. The dag2vhdl code was inspired by similar code originally written by Brian Van Essen for creating Verilog versions of kernels in a specific queue ISA; substantial revision would have required to match the generic HASTE assembly language, so the software was rewritten from scratch and additionally made to generate VHDL rather than Verilog. Finally, the Graph Template Library (GTL) version 1.2 [39], which was written by researchers at the University of Passau, Germany, was used throughout AppMap to represent graphs and graph algorithms.

7.1 Application Mapping

The first sub-flow to be covered is application mapping, which is shown in Figure 7.2. Note that this figure is just a portion of the overall flow which was seen in Figure 7.1. Application mapping is the name chosen to describe the process of implementing an application and/or application kernels on a specific HASTE architecture. Although part of the process resembles compilation and even uses a standard C compiler, much of the process is more closely related to traditional computer-aided design (CAD), such as that used for FPGAs. Rather than refer to it as compilation, or placement and routing, the entire process will be referred to as application mapping. Design capture of kernels can be done at either the level of annotated C code, generic HASTE assembly code, or DFGs, as indicated in the figure. Sequential code can be entered as the first two forms, excluding the DFG form; despite the conceptual representation of sequential code as CDFGs, there is no actual graph-based representation of sequential code implemented in the current tool flow. Note that design capture at the level of unannotated C code or PISA assembly code is not considered, since these two forms are followed by manual steps in the flow, and thus design capture is more efficiently done at the level of their succeeding forms. An example will be presented in the next section that starts with unannotated C code and proceeds through the entire application mapping flow. The application mapping flow is used at least in part by the other two flows and for all of the experiments in the remainder of this thesis. Results comparing application mapping to the different ISAs and different fabrics are presented in Chapter 8.

7.1.1 Kernel Annotation

C code from existing benchmarks can be used in the tool flow. Kernels in this C code need to be identified manually, either by inspection, profiling of the code, or some combination of both. The C code may need to be modified slightly to make kernels more suitable for HASTE; if this is the case, care must be taken to ensure that the modified code works the same as the original code. Once kernels are identified, they must be annotated by including assembly language directives so that they can be identified in the assembly language version. The process of finding and identifying kernels could be done in an automated manner; modern ILP compilers such as IMPACT [40] have been used for this purpose in other projects. However, the amount of time needed to learn such a compiler, modify it for use with HASTE, and evaluate methods

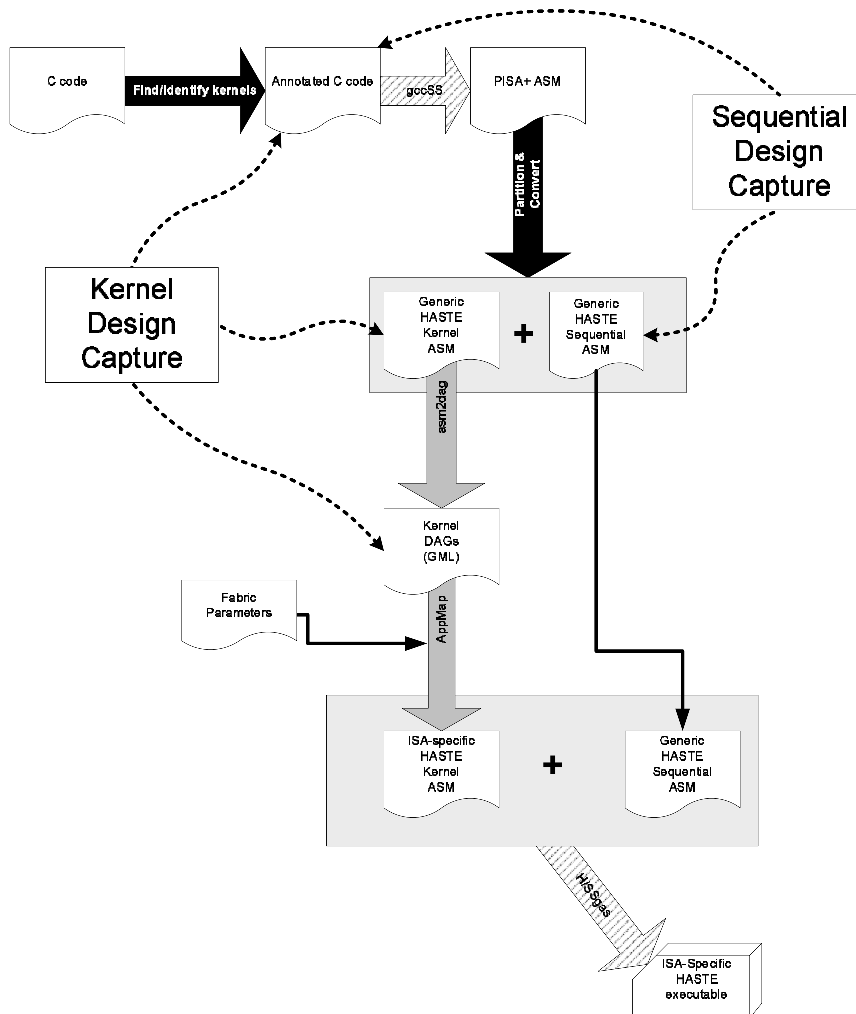


Figure 7.2: Application Mapping Tool Flow

for identifying kernels was deemed to be too time-consuming for this project. Indeed, similar efforts have been the subject of entire PhD theses on their own. Given that large numbers of benchmarks were not needed for this thesis, it was decided that a manual method for kernel identification and kernel annotation was sufficient.

A simple example of an unannotated program written in standard C is shown in Figure 7.3; it includes both a kernel (simple_kernel) and a main procedure that basically just acts as a wrapper for the kernel and provides some random input data. Both the input and output data streams are stored in a memory buffer; in a real application, these streams might be sent to or received from another procedure, received from a sensor, written to a storage device, or sent to another device, but in any case, the same application mapping methods could be used, as long as the memory access requirements of GHAL were met (i.e., fixed base address and stride, with stride = 0 for memory-mapped I/O).

The first step of the process is to annotate the original C code so that the kernel could be identified. Annotation is implemented using ASM statements, which insert assembly language commands directly into the compiled code. The annotations can be seen in the listing in Figure 7.4; they are shown in a large, bold font. The annotations insert the dummy loop delimiters, as discussed in Chapter 4, into the code. They are used to identify the kernel, with DLPBGN used to identify the beginning of the kernel (placed just before the first loop iteration), DLPEND used to identify the end of the loop body, and DLPDONE used to identify the end of kernel.

7.1.2 Compilation

Once the C code has been annotated to identify the kernels, it is compiled using a modified version of the gcc compiler, version 2.7.2.3, for the PISA instruction set. PISA (Portable Instruction Set Architecture) is a MIPS-like ISA developed for use with the SimpleScalar simulator [34]. Since SimpleScalar was a good choice for a simulator for this project and since PISA is a very generic ISA, PISA was chosen as the basis for the HASTE assembly language. The modified version of gcc for SimpleScalar will be referred to as gccSS. Appendix B documents the HASTE assembly language and details the differences between HASTE assembly and PISA.

A couple of minor modifications were made to the gccSS compiler. First, the number of integer registers was increased from 32 to 64, with all of the new registers being considered as

```
/* * simple.c * */
#define NUM_DATA 10000
#include <stdlib.h>
void simple_kernel (unsigned char* in_data, unsigned short* out_data);
int main(void) {
   unsigned char *p_in;
   unsigned short *p_out;
   int i;
   p_in = malloc(3*NUM_DATA*sizeof(unsigned char));
   p_out = malloc(NUM_DATA*sizeof(unsigned short));
   for (i = 0; i < (NUM_DATA*3); i++) {
      p_in[i] = rand();
   }
   simple_kernel(p_in,p_out);
   return 0;
}
void simple_kernel (unsigned char* in_data, unsigned short* out_data) {
   int i;
   unsigned char a,b,c;
   unsigned short x,y,z;
   for (i=0;i<NUM_DATA;i++) {
      a = in_data[i*3];
      b = in_data[(i*3)+1];
      c = in_data[(i*3)+2];
      x = a >> 2;
      y = b + x;
      if (y > c)
         z = c;
      else
         z = y + 4;
      out_data[i] = z;
    }
}
```

Figure 7.3: Unannotated C Code

```
void simple_kernel (unsigned char* in_data, unsigned short* out_data) {
   int i;
   unsigned char a,b,c;
   unsigned short x,y,z;
   asm("dlpbgn");
   for (i=0;i<NUM_DATA;i++) {
      a = in_data[i*3];
      b = in_data[(i*3)+1];
      c = in_data[(i*3)+2];
      x = a >> 2;
      y = b + x;
      if (y > c)
         z = c;
      else
         z = y + 4;
      out_data[i] = z;
      asm("dlpend");
   }
asm("dlpdone");
}
```

Figure 7.4: Kernel C code with annotations

temporary registers. Since the PISA instruction word is 64 bits long and has 8-bit register fields, this did not require any major changes to the compiler. Increasing the number of registers beyond 64 caused numerous problems apparently related to subtle aspects of the machine definition and it was not possible to fix these problems in a reasonable amount of time. The additional registers greatly reduced the chance that the variables needed within a loop body would not fit within the available registers and have to be spilled to memory. Since PISA, like MIPS, has only 8 temporary registers that do not have to be saved by a procedure, the new registers increased by a factor of 5 the number of available temporary registers. The names of several assembly language instructions were modified but no further modifications were made to gcc.

Annotated C code was compiled using gccSS with the following options:

-O3: This tells gcc to use numerous optimizations, in particular, this forces it to use more registers, including the newly added ones.

-fomit-frame-pointer: This lets gcc omit the frame pointer for code which doesn't need it, freeing up an additional register.

-verbose-asm: This produces more verbose (and thus readable) assembly code.

-S: This tells gcc to stop after compilation and emit assembly code.

Any code that can be compiled for SimpleScalar can be compiled with the modified SSgcc. All normal C run-time libraries are available. The sequential portion of the example code after compilation is shown in Figure 7.5 and the kernel portion of the example code after compilation is shown in Figure 7.6. The only modification that is sometimes needed is the manual moving of the DLPBGN and DLPEND commands to the actual loop boundaries to allow for accurate profiling of the kernels. Note that the positions of the loop delimiters DLPBGN and DLPEND are not in exactly the correct positions, but they are close enough that they can easily be moved to where they need to be; the arrows in the figure show their correct locations. Again, a tool could have been written to handle this detail, but it was not deemed worth the effort.

7.1.3 Conversion to GHAL

The PISA assembly language next has to be converted to the generic HASTE assembly language (GHAL). There were three primary changes that had to made to the kernel code; those relating to control flow for loops, those relating to memory access, and those relating to general control flow. Since no control instructions are allowed in the HASTE kernel model, looping in kernels must be implemented using only the loop delimiter instructions, LPBGN and LPEND, or their variants LPBNGI and LPENDR, as discussed in Chapter 4. These must be inserted manually, replacing the original code that implemented the looping. If the dummy loop delimiters, DLPBGN and DLPEND, are present for kernel profiling, they should be removed as well. Ordinary *load* and *store* instructions are not allowed in the HASTE kernel. Instead, the indexed memory *send* and *receive* instructions must be used. In addition, the proper memory port setup instructions must be inserted. Again, there is no automated way to do this at the present time. The specific memory *send* and *receive* instructions and the setup instructions are covered previously and will not be reviewed here. Finally, all control flow in the kernels must be eliminated. This is done using a form of partially predicated execution. In effect, control paths are executed in parallel. When an assignment is made to a value that depends on a particular control path being taken, then a SELECT instruction is used to determine which value is assigned, based on the original conditional; this instruction is covered in more detail in Chapter 4 and in Appendix B. Once again, this part of the conversion process is also done manually, although there are tools and

```
main:
        .frame $sp,40,$31 # vars= 0, regs= 5/0, args= 16, extra= 0
        .mask 0x800f0000,-8
        .fmask 0x00000000,0
        subu $sp,$sp,40
        sw $31,32($sp)
        sw $19,28($sp)
        sw $18,24($sp)
        sw $17,20($sp)
        sw $16,16($sp)
        jal __main
        li $4,0x00007530 # 30000
        jal malloc
        li $4,0x00004e20 # 20000
        move $17,$2
        jal malloc
        move $19,$2
        move $16,$17
        addu $18,$17,30000
$L29:
        jal rand
        sb $2,0($16)
        addu $16,$16,1
        slt $2,$16,$18
        bne $2,$0,$L29
        move $4,$17
        move $5,$19
        jal simple_kernel
        move $2,$0
        lw $31,32($sp)
        lw $19,28($sp)
        lw $18,24($sp)
        lw $17,20($sp)
        lw $16,16($sp)
        addu $sp,$sp,40
        j $31
        .end main
```

Figure 7.5: Assembly for Example Sequential Code

```
simple_kernel:
     .frame    $sp,0,$31
     .mask     0x00000000,0
     .fmask    0x00000000,0
     dlpbgn
     addu $7,$5,20000
$L43:
     lbu  $2,0($4)
     lbu  $3,1($4)
     lbu  $6,2($4)
     srli $2,$2,2
     addu $3,$3,$2
     sltu $2,$6,$3
     beq  $2,$0,$L44
     move $2,$6
     j    $L45
$L44:
     addu $2,$3,4
$L45:
     sh   $2,0($5)
     dlpend
     addu $5,$5,2
     addu $4,$4,3
     slt  $2,$5,$7
     bne  $2,$0,$L43
     dlpdone
     j    $31
     .end simple_kernel
```

Figure 7.6: Assembly for Example Kernel

techniques that could have been adopted to perform this process. Figure 7.7 shows the kernel code after it has been modified to use GHAL. Note that the pointers to the input and output buffers, in registers $4 and $5 in the original code are now used as the base addresses in the SETB instructions. Once converted into GHAL, the kernel code is now expressed in a form in which all memory operations are expressed in the form used by the HASTE architecture and all control flow has been converted into either loop constructs or dataflow operations. GHAL is executable in a modified version of SimpleScalar for producing data vectors for simulation and for obtaining information about HASTE execution. In addition, GHAL provides a base form from which a dataflow graph for the kernel can be constructed and then transformed into any ISA specific form.

7.1.4 Conversion to DAG

The next step in the application mapping process is conversion of the kernel (only) into a DAG. This is necessary because the next step in the flow, mapping to ISA-specific code for a particular

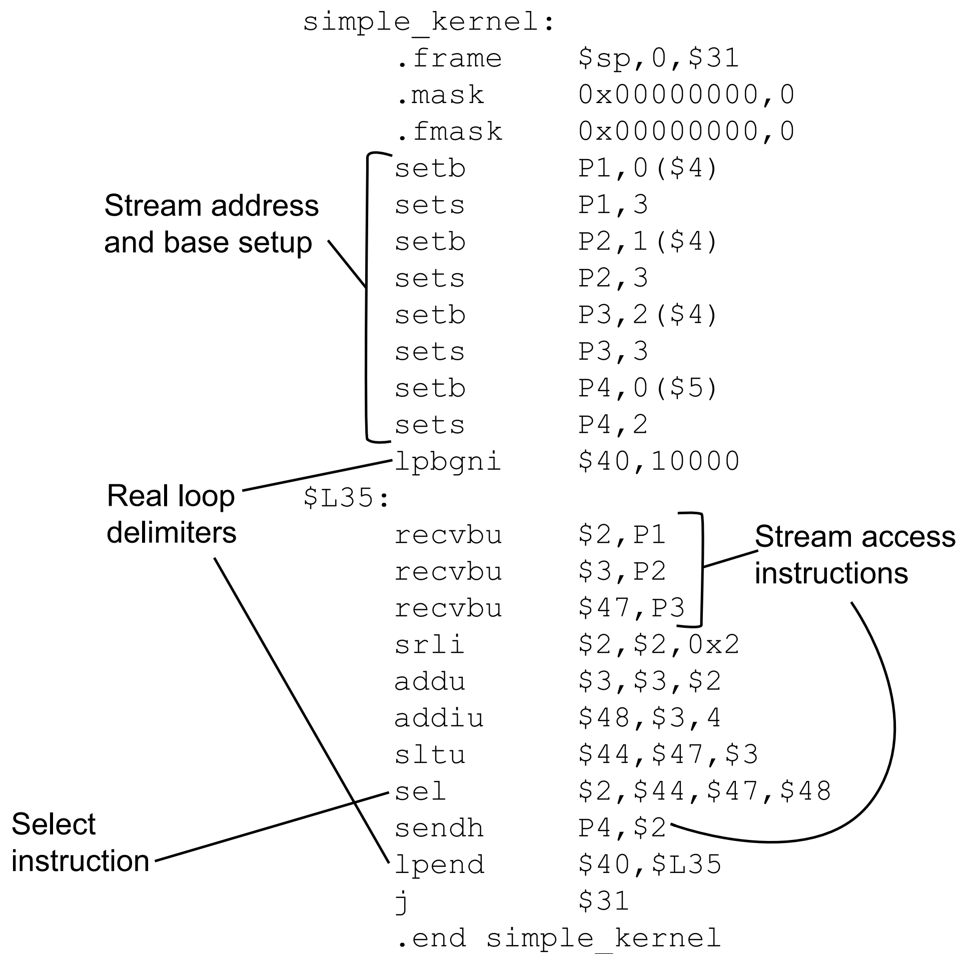


Figure 7.7: GHAL Assembly Code

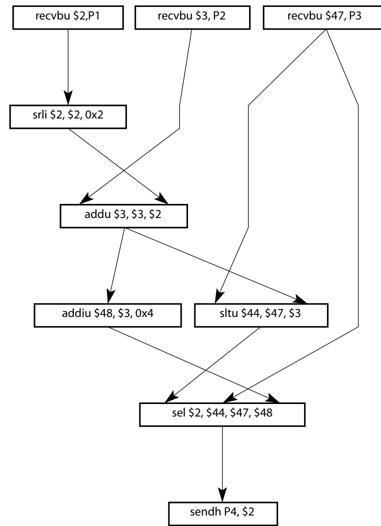


Figure 7.8: DAG Generated from GHAL Code

fabric requires a DAG as input. In addition, kernels can be captured in DAG form from other tool flows. A tool called ASM2DAG was written specifically for this task in C++. This tool reads in a single kernel in GHAL and outputs a textual representation of the corresponding directed, acyclic dataflow graph, which can be read by the mapping tools used in the next step. The DAG file is written in GML, the Graph Modelling Language, which is part of the Graph Template Library [39]. The DAG file for the example kernel shown in Figure 7.7 is listed in Appendix E. A graphical representation of the DAG is shown in Figure 7.8.

7.1.5 Mapping to Specific HASTE Implementations

Once in DAG form, kernels can be transformed into a form appropriate for the specific ISA and fabric that they will be run on. Such transformed kernels can than be run on either the RCF or the SPU of the target architecture. The AppMap program was used to map code into a form suitable for a specific ISA and a specific fabric. This was done by in effect placing and routing the DAG onto the fabric, and then writing the application out in an order that the CTE could sucessfully transform into an equivalent fabric configuration. The details of this process vary greatly between ISAs and these details are covered in the individual ISA subsections of Section 10.2.

In a production HASTE system, all of the code, both kernel and sequential code, would be transformed into ISA-specific format. However, for the purposes of this research, not all sequential code for applications was converted into ISA-specific code. The ISA-specific HASTE simulators could all run GHAL as well as their specific ISA, so for those experiments where the sequential code was not important, often the sequential code was left in GHAL form. Various simple tools written in Perl, and not shown in the tool flow, were used for creating ISA-specific sequential code (the queue ISA also used part of the AppMap code for sequential code as well as kernel code). Once converted, the kernel code and sequential code were recombined into a single file.

7.1.6 Assembly

The final step in the application mapping process was the creation of the final executable. This was done using a modified version of the SimpleScalar variant of the GNU assembler(gas). This modified assembler, H/SSgas can handle all three of the HASTE ISAs, as well as GHAL and PISA, by specifying different command-line switches. The executable format is basically the same as the standard PISA, using the specific ISA formats discussed in Chapter 4.

7.2 Simulation and Validation Tool Flow

The second sub-flow is the simulation and validation flow, shown in Figure 7.9. In order to show the feasibility of the HASTE concept, it is important to show that kernels implemented in various HASTE ISAs and on different HASTE fabrics give identical results to the original application kernels when provided with the same input data. If the implementations created by the CTE produce different results on different fabrics, or if any of the fabric implementations do not produce the results intended by the programmer, as represented by the original compiled kernel code, than HASTE architectures will not be usable in any practical situations. Results from using this flow are covered in Chapter 9.

The steps involved in the validation process include most of those covered in the application mapping sub-flow; the steps of the validation process are listed in Figure 7.10. Each number in the list of steps corresponds to a numbered box in Figure 7.9. The same simple example

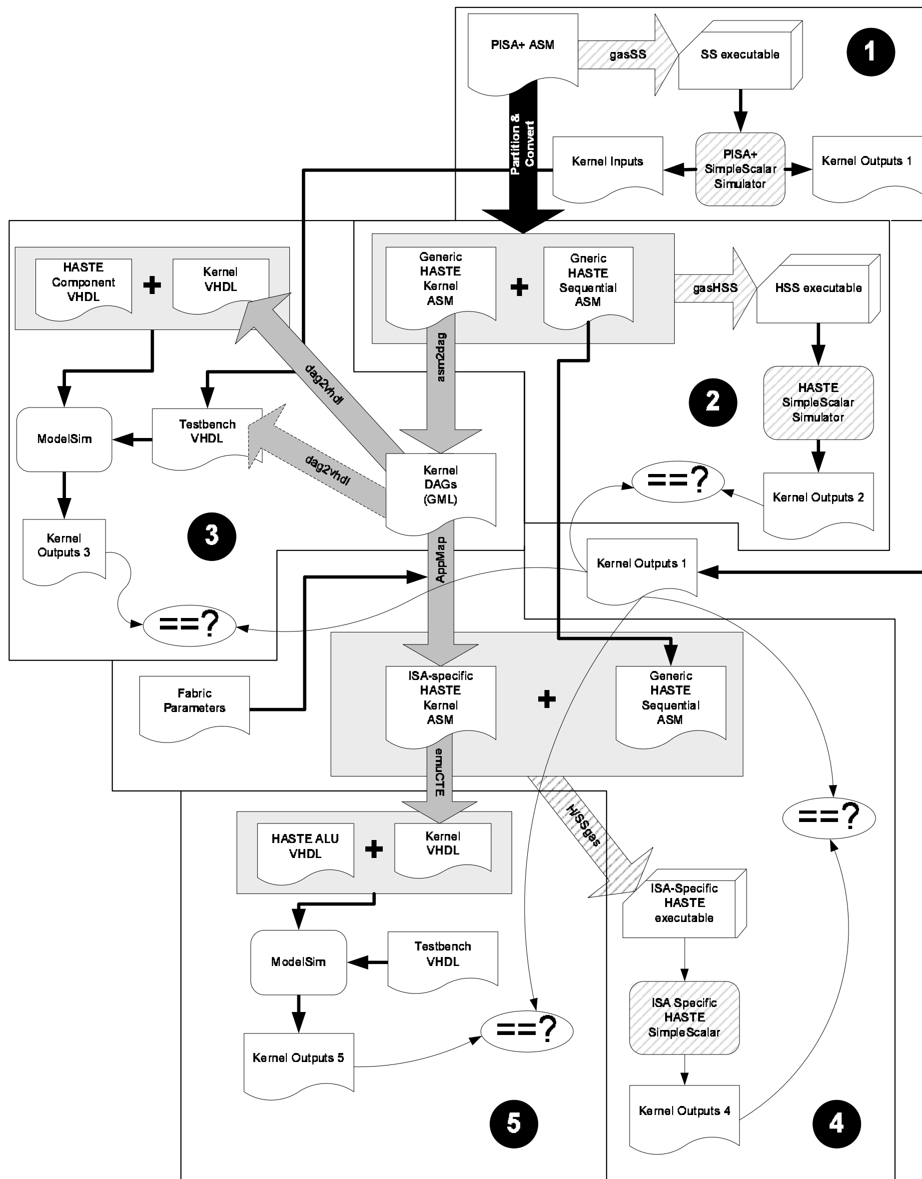


Figure 7.9: Validation and Simulation Tool Flow

1. Original Code:
    (a) Annotation of kernel.
    (b) Simulation and trace generation.
2. GHAL:
    (a) Conversion to GHAL.
    (b) Simulation and trace generation.
    (c) Comparison of traces to original.
3. Kernel DAG:
    (a) GHAL listing converted to DAG.
    (b) DAG converted to HDL.
    (c) HDL simulation using original inputs.
    (d) Comparison of output traces to original.
4. Mapped Version:
    (a) DAG mapped to specific ISA and fabric.
    (b) Mapped assembly generated.
    (c) Mapped assembly simulated and traces generated.
    (d) Comparison of traces to original.
5. Implemented DAG
    (a) Mapped assembly converted to fabric-specific DAG using CTE algorithm.
    (b) Fabric-specific DAG converted to HDL.
    (c) HDL simulation using original inputs.
    (d) Comparison of output traces to original.

Figure 7.10: Steps in Validation Process

program used in the previous section will be used to demonstrate each step of the procedure. The baseline for comparison is the input and output traces generated by the original code. The traces are then compared to those generated by the code at each phase of the HASTE mapping process. While it is not absolutely necessary to check results at each step of the process, doing so increases the likelihood that any discrepancies between implementations will be found. Note that although no simulator for an entire HASTE system has been created, simulation was used at each step to verify operation of all of the HASTE components. Specifically, SPU operation for each ISA was simulated using modified SimpleScalar code in Step 4c; CTE operation was simulated using C code in Step 5a, with specific CTE algorithms used to generate the fabric-specific DAGs from the mapped, ISA-specific assembly; and fabric operation was simulated in Step 5c using the HDL generated from the fabric-specific DAGs. These simulations are architecture-level, not microarchitecture-level. Since the goal of this portion of this thesis was to evaluate the overall HASTE architecture and not a specific hardware implementation of it, this level of simulation is appropriate and justified. Some hardware implementation details are covered in Chapter 11, which evaluates the performance and area costs of HASTE architectures. The remainder of this section will discuss each step of the validation process.

Step 1: The validation process starts with the original C source code. The same example covered previously will be used; the C listing for this example is Figure 7.3 and the original assembly listing is Figure 7.6. The executable produced from the original assembly code is the baseline executable; the loop delimiters are not actually executed, and all of the executed instructions are PISA instructions. This baseline executable is then run on a slightly modified SimpleScalar simulator. Most of the modifications involve identifying kernels and recording kernel statistics. A reproducible set of input data was provided by creating pseudo-random data of a type and range of values appropriate for the kernel. The simulator collects statistics about the number of instructions, number of kernel instructions and so on; the output for this simple example is shown in Figure 7.11(a). Note that the loop delimiters are not counted in the statistics. As can be seen, for 10,000 loop iterations, 136,170 cycles were spent in the kernel. This is reasonable, given that there are 13 instructions in the loop body if the first branch is taken and 14 instructions otherwise. Given the artificial nature of this application, the percentage of instructions in the kernel (11%) is not particularly relevant; it could be made much higher by

```
sim: ** starting functional simulation **
Completed kernel 1:  10000 iterations, 136170 instructions, 30000 loads, 10000 stores

sim: ** simulation statistics **
sim_num_insn                 1165726 # total number of instructions executed
sim_total_kernel_insn         136170 # total number of kernel instructions executed
sim_num_kernels                    1 # number of different kernels executed
sim_total_iterations           10000 # total number of kernel loop iterations executed
sim_total_kload_insn           30000 # Total number of kernel load instructions executed
sim_total_kstore_insn          10000 # Total number of kernel store instructions executed
sim_kernel_percent           11.6811 # percent of program insts in kernel(s)
sim_num_refs                  497708 # total number of loads and stores executed
sim_elapsed_time                   1 # total simulation time in seconds
sim_inst_rate           1165726.0000 # simulation speed (in insts/sec)
ld_text_base              0x00400000 # program text (code) segment base
ld_text_size                   25792 # program text (code) size in bytes
ld_data_base              0x10000000 # program initialized data segment base
ld_data_size                    4096 # program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base             0x7fffc000 # program stack segment base (highest address in stack)
ld_stack_size                  16384 # program initial stack size
ld_prog_entry             0x00400140 # program entry point (initial PC)
ld_environ_base           0x7fff8000 # program environment base address address
ld_target_big_endian               0 # target executable endian-ness, non-zero if big endian
mem.page_count                    27 # total number of pages allocated
mem.page_mem                    108k # total size of memory pages allocated
mem.ptab_misses                   27 # total first level page table misses
mem.ptab_accesses            5858882 # total page table accesses
mem.ptab_miss_rate            0.0000 # first level page table miss rate
```

(a) Simulation Statistics

```
0x000000f5 0x00000024 0x00000045
0x0000007a 0x00000071 0x00000045
0x00000056 0x00000041 0x00000007
0x00000028 0x000000ad 0x00000071
0x0000001b 0x00000016 0x00000075
0x00000059 0x00000067 0x000000f8
0x000000ee 0x0000007c 0x00000094
0x0000007d 0x00000088 0x00000050
0x0000003b 0x00000019 0x000000f3
0x0000004f 0x0000002c 0x000000e9
0x0000008c 0x00000021 0x0000000d
0x000000d2 0x0000009c 0x0000007f
0x00000017 0x000000f2 0x000000c0
0x0000001e 0x0000001a 0x0000006d
0x0000008f 0x00000035 0x00000083
0x00000004 0x0000008f 0x000000eb
0x000000fc 0x0000007d 0x00000067
0x00000090 0x000000fa 0x000000f0
0x000000e0 0x00000035 0x00000009
0x000000d4 0x00000085 0x00000035
0x000000bd 0x00000011 0x00000057
0x000000cb 0x000000e3 0x000000f3
0x0000004a 0x000000fa 0x000000e5
0x0000000a 0x00000019 0x00000000
0x00000078 0x000000a8 0x00000035
0x000000fb 0x000000ad 0x000000c4
0x000000e6 0x000000a9 0x00000041
0x0000004e 0x0000003a 0x0000003c
0x0000003e 0x0000001a 0x00000071
```

(b) I/O Trace

Figure 7.11: Results From Original Assembly

running the application for more iterations. In addition to gathering statistics, the simulator records traces of all memory access to and from the kernel. A portion of the input trace is shown in Figure 7.11(b). The values in the input trace are produced by the main portion of the executable using a pseudo-random number generator. All versions of the code should provide the same input values to the kernel; this original trace of input values will be used to verify this fact. In addition, this input trace will be used to provide inputs to kernel implementations that are simulated without the rest of the application being implemented, primarily in the case of the VHDL implementations that will be discussed in steps 3 and 5.

Step 2: Next, a GHAL version of the application is created, as covered in Section 7.1.3. Once the kernel has been expressed in GHAL it can be assembled and run on a GHAL version of the SimpleScalar simulator. Run statistics, input traces, and output traces are gathered during simulation. The input traces gathered from this run are compared to the input traces gathered from the first run to make sure the same data was supplied to the kernel. The traces are compared using the Linux utility 'diff'. If the order of memory accesses has changed between runs, due perhaps to a different order of equivalent memory access instructions, a simple Perl script is used to reorder the data in the trace files. Finally, the output traces gathered from this run are compared to those from the first run. If all output values are identical in both versions then the kernel implementation is considered to be correct.

Table 7.1: Comparison of Execution Statistics for Original (PISA) and GHAL Assembly Code

simulator statistic	statistic description	Original Code	GHAL Code
sim_num_insn	Total # of instructions executed	1165762	1129564
sim_total_kernel_insn	Total # of kernel instructions executed	136170	100000
sim_total_iterations	Total # of kernel loop iterations executed	10000	10000
sim_total_kload_insn	Total # of kernel load instructions executed	30000	30000
sim_total_kstore_insn	Total # of kernel store instructions executed	10000	10000
sim_kernel_percent	Percent of program insts in kernel(s)	11.6811%	8.8530%

In addition, the run statistics are compared to see how the sequential GHAL code compares to the original assembly code. Table 7.1 shows the statistics gathered from both runs of the example, the first being the original run as shown in Figure 7.11(a) and the second being the results from the GHAL version of the code, as shown in Figure 7.7. Note that the number of instructions is smaller for the GHAL version than for the original version, and that it is an exact multiple of the number of loop iterations. The smaller number of instructions is due to the fact that the send and receive instructions are self-incrementing, and thus the last two add instructions in the original code are not needed. Using a select instruction rather a branch eliminates the need for a jump instruction. The fact that the number of kernel instructions is an exact multiple of the number of iterations is due to the fact that both parts of the if..then statement implemented with a branch and jump in the original code are executed in every iteration when using a select instruction instead. In the original version, only the part needed based on the input data is executed; since the two paths have different numbers of instructions, the number of instructions executed varies from iteration to iteration. Since all instructions are executed regardless of runtime data, the same number of instructions are executed for each iteration of the GHAL version. Further exploration of these differences in cost for GHAL code and other runtime statistics is discussed in Chapter 8; the run statistics for a range of different benchmarks are covered in that chapter as well.

Step 3: For all HASTE ISAs, the next step is to convert the GHAL code into a generic DAG. The asm2dag program is used to automatically perform this conversion. This DAG form can be used to generate assembly for any of the three basic HASTE ISAs. In addition, this generic DAG form is used to generate VHDL for the ASIC and FPGA versions of the applications. The DAG generated for this example is shown in Figure 7.8. This figure is generated automatically using GML format written directly by asm2dag. The Graphlet tool is used to create a Postscript

version of the graph as reproduced herein.

Since this DAG is an abstract, intermediate form, there is no simulator that can run the DAG form directly. However, it is desirable to be able to ensure that this DAG form is constructed correctly and that it produces the desired results. Therefore, this next step entails the production of VHDL from the DAG and then simulation of this VHDL on a commercial HDL simulator, ModelTech's ModelSim. The dag2vhdl tool is used to perform the conversion. This tool was written specifically for this project in C++. It is a fairly straightforward program that uses structural VHDL to specify and connect pre-written, parameterized, VHDL modules that correspond to each node type that could be found in the DAG (and thus to each possible kernel-legal GHAL instruction). It also produces a testbench file for the application. The application VHDL generated for this simple example is shown in Figure 7.12 . The testbench for the example is rather large and is included as Appendix D.

A figure showing the structure of the VHDL simulation is shown in Figure 7.13. The testbench reads in the original input trace data from a file and applies this data as stimuli for the kernel inputs. The testbench applies inputs for a number of cycles equal to the depth of the pipelined application (this depth is determined automatically by dag2vhdl) then starts reading the output of the application. The application output is compared to the output trace data and an error is generated if any discrepancy is found. A report file is generated showing the results of the simulation. All parts of this step can be performed automatically; a Makefile calls asm2dag to generate the DAG from the disassembled executable generated in Step 2, then calls dag2vhdl to create the VHDL application and testbench, and finally calls ModelSim from the command line to simulate the design.

Step 4: The next part of the validation process requires the choice of a specific ISA and specific architectural parameters, and use of the AppMap tool to map the application to that specific ISA and architecture. Given each specific ISA, certain fabric parameters must also be chosen. For the queue ISA, the only relevant parameter is the read connectivity; the width is determined by the DAG width. For the register ISA, the fabric width, read connectivity, and register file size can all be changed. For the relative register ISA, the write connectivity can be changed, as well as all of the parameters listed for the register ISA. Rather than validate all possible mapping of applications to fabrics, a procedure was developed to find and implement

```
library IEEE,work;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_arith.all;
use work.hastecomps.all;
entity simple is
port ( var_recvbu_0 : in std_logic_vector(7 downto 0);
       var_recvbu_1 : in std_logic_vector(7 downto 0);
       var_recvbu_2 : in std_logic_vector(7 downto 0);
       clk          : in std_logic;
       sendh_0      : out std_logic_vector(7 downto 0) );
end simple;
architecture STRUCT of simple is
     signal var_addiu_0 : std_logic_vector(8 downto 0);
     signal var_addu_0 : std_logic_vector(8 downto 0);
     signal var_sel_0 : std_logic_vector(7 downto 0);
     signal var_sltu_0 : std_logic_vector(0 downto 0);
     signal var_srli_0 : std_logic_vector(7 downto 0);
     signal pass_var_recvbu_2 : std_logic_vector(7 downto 0);
     signal pass_var_recvbu_1 : std_logic_vector(7 downto 0);
     signal pass_var_recvbu_2_0 : std_logic_vector(7 downto 0);
     signal pass_var_recvbu_2_1 : std_logic_vector(7 downto 0);
begin -- STRUCT
node_srli_0 : srli_op
   generic map ( width_inA => 8, width_outA => 8, imm => 16#2#)
   port map (var_recvbu_0, clk, var_srli_0);
node_addu_0 : addu_op
   generic map ( width_inA => 8, width_inB => 8, width_outA => 9)
   port map (pass_var_recvbu_1, var_srli_0, clk, var_addu_0);
node_addiu_0 : addiu_op
   generic map ( width_inA => 9, width_outA => 9, imm => 4)
   port map (var_addu_0, clk, var_addiu_0);
node_sltu_0 : sltu_op
   generic map ( width_inA => 8, width_inB => 9, width_outA => 1)
   port map (pass_var_recvbu_2_1, var_addu_0, clk, var_sltu_0);
node_sel_0 : sel_op
   generic map ( width_inA => 1, width_inB => 8, width_inC => 9, width_outA => 8)
   port map (var_sltu_0, pass_var_recvbu_2, var_addiu_0, clk, sendh_0);
node_pass_var_recvbu_2 : pass_op
   generic map ( width_inA => 8, width_outA => 8)
   port map (pass_var_recvbu_2_1, clk , pass_var_recvbu_2);
node_pass_var_recvbu_1 : pass_op
   generic map ( width_inA => 8, width_outA => 8)
   port map (var_recvbu_1, clk , pass_var_recvbu_1);
node_pass_var_recvbu_2_0 : pass_op
   generic map ( width_inA => 8, width_outA => 8)
   port map (var_recvbu_2, clk , pass_var_recvbu_2_0);
node_pass_var_recvbu_2_1 : pass_op
   generic map ( width_inA => 8, width_outA => 8)
   port map (pass_var_recvbu_2_0, clk , pass_var_recvbu_2_1);
end STRUCT;
```

Figure 7.12: Application VHDL Generated from DAG

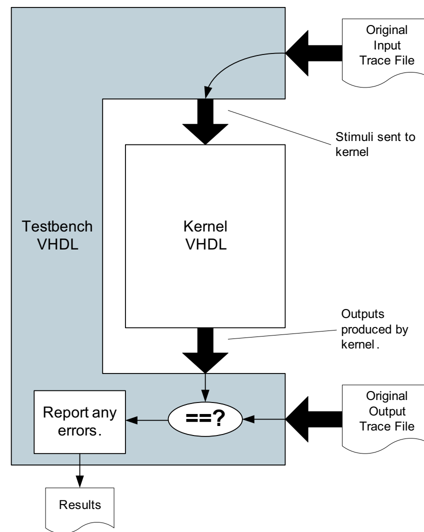


Figure 7.13: Testbench Structure

the most efficient architectures for a given architecture. This procedure will be discussed in more detail in Chapter 8; it will suffice here to say that a subset of possible architectures are picked for each ISA type and each application is mapped to each architecture in the relevant set, using the mapping algorithms covered in that same chapter. The assembly code generated for each mapping of each kernel is assembled into an ISA- and fabric-specific executable and simulated as before, with the output traces compared to the original traces. As with all steps of the validation flow, scripts were developed to automate this process.

Step 5: The previous step validated operation of the ISA-specific HASTE code as it would be run on the SPU. This was enabled by the use of a heavily modified version of the SimpleScalar simulator. However, no such cycle-accurate simulator exists for the CTE. This could have been done using an HDL, but since the specifics of each CTE vary widely depending on the ISA, fabric width, machine language encodings, the decision was made to use a higher level emulation of CTE operation. Using existing C++ code from asm2dag and AppMap, as well as new code representing various CTE algorithms, a new tool was created to generate application mappings from the ISA- and fabric-specific assembly code discussed in Step 4. This program, emuCTE, takes a sequence of HASTE assembly instructions and a set of architectural parameters as inputs, and generates an application mapping and a mapped application DAG. The application mapping

is represented internally as a mapping of application graph operations to hardware graph nodes representing specific functional units, and a mapping of register reads and writes to specific edges in the hardware graph. This mapping is then automatically compared to the input fabric parameters to verify that it is a valid mapping for the specific fabric. If not, then the CTE algorithm did not work properly. If it is a valid mapping, then a graphical representation of the mapping is generated and a new DAG representing the mapped application is generated. This new DAG is then converted to VHDL form, using a VHDL model of the RCF fabric instead of the operator models used in Step 3. The same test bench and vectors that were used in Step 3 are used and the VHDL is verified as before.

7.3 Hardware Implementation Tool Flow

The third and final sub-flow is the Hardware Implementation flow. This flow is used to generate information concerning the performance and area requirements of kernels implemented on various HASTE fabrics, as well as standard cell ASIC implementations, and implementations using a commercial FPGA. Since only kernels are compared, the DAG form, as generated in the application mapping sub-flow, is used as the entry point to this sub-slow. This sub-flow is shown in Figure 7.14. The results obtained with this sub-flow are covered in Chapter 11.

7.3.1 ASIC and FPGA Implementations

For both the ASIC and the FPGA implementations, the dag2vhdl tool described previously, was used to convert generic HASTE DAGs into a structural VHDL netlist. The netlist describes the kernel design as a series of multi-bit components and interconnection between components. This netlist references a set of synthesizable, RTL-level components designed specifically for HASTE in VHDL. Each of these components corresponds to a single GHAL instruction. Two component libraries were created, one with a pipeline register after each operation and one without pipeline registers.

The use of the component libraries ensured that all computational logic in the design is implemented as parameterized components, each of which corresponds to a specific GHAL instruction. All of the instructions in Appendix B designated as kernel-legal have a VHDL implementation.

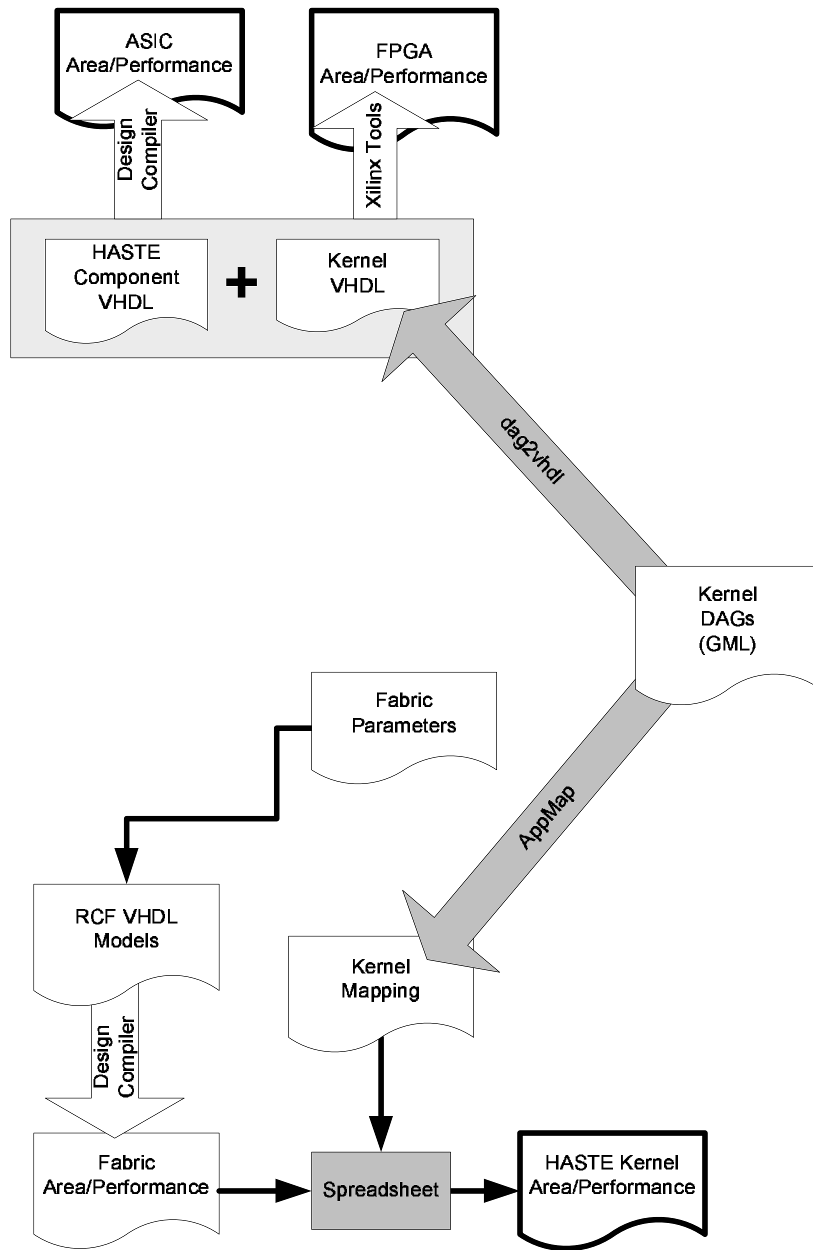


Figure 7.14: Hardware Implementation Tool Flow

All of these components are parameterized by input and output data widths, and by an immediate value parameter for those instructions needing them. The data widths for each input and output were determined by examining the range of data observed during profiling. Thus the ASIC implementation (and the FPGA implementation) can potentially be much more efficient than the HASTE implementation, since all of the operations in the HASTE implementation are fixed at 32 bits in width.

Figure 7.15 shows a schematic of the structural VHDL for the example used in this chapter. It uses 6 different components, corresponding to the five different operations in the kernel, not including the *send* and *receive* operations, and a pass component. The pass component is simply a pipeline register. For pipelined implementations, the dag2vhdl tool must make the design level by inserting pass operations, just as is done for queue ISA applications. This is necessary to ensure that all data passes through the same number of pipeline registers and thus all data in the pipeline is synchronized. In pass register fabrics, this insertion of registers is done inherently by the use of the pass registers themselves, but it must be done explicitly here. In addition, registers must be inserted for those *receive* operations that are not on the first level. In a HASTE fabric, this is handled by the structure of the memory interface for *send* and *receive* instructions. For the ASIC and FPGA implementations an external MAU is assumed, so it is not necessary to implement *send* and *receive* components. The inputs to the kernel are pipelined, however, to ensure that they are synchronized. The outputs of multiple output kernels are not similarly pipelined. Both the decision to pipeline the inputs and not pipeline the outputs are somewhat arbitrary and were done primarily to simplify testbench construction. The impact of these decisions on area and performance are negligible; the addition of few registers to a fully pipelined design is not a significant change.

For the ASIC implementation, the kernel VHDL and the components libraries were all synthesized using Synopsys Design Compiler, with DC-Ultra optimization enabled. The synthesis tool targeted a proprietary commercial standard cell library for a production 90 nanometer process. Given that the designs are relatively small and simple by contemporary standards, placement and routing was not performed, as the estimated area and timing from the synthesis tool were accurate enough for purposes of this research.

For the FPGA implementation, the Xilinx XC3S200-5FT256 FPGA was used as a target

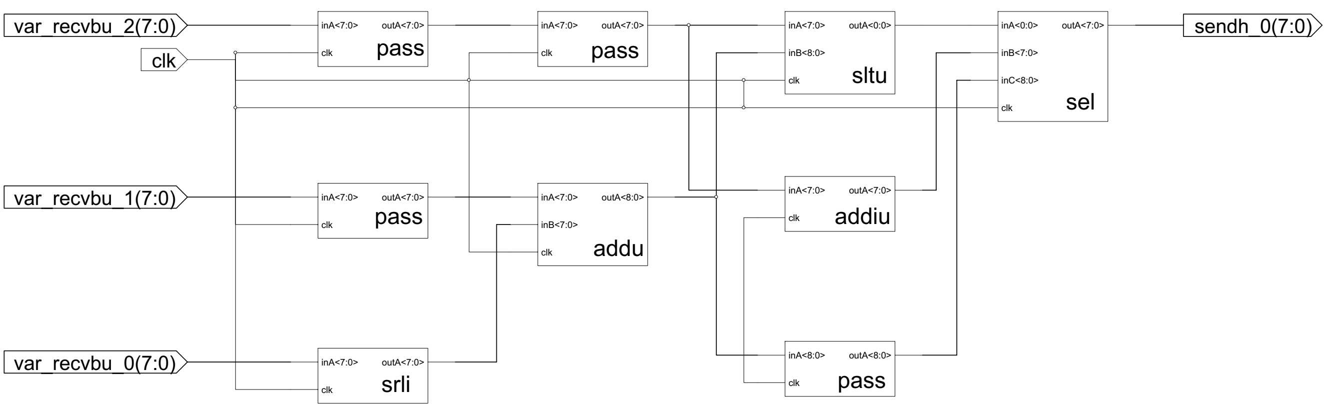


Figure 7.15: Structural Model of Simple Example

device (reasons for this are covered in Chapter 11). The Xilinx ISE Alliance design tools were used, in particular the Xilinx XST synthesis tool. XST was used to synthesize the application kernel design and the Xilinx supplied tools were used to place and route the design for the target device and produce information on area and performance.

7.3.2 HASTE Hardware Implementation

Determining the characteristics of HASTE hardware implementations is straightforward, given the synthesis results for the fabric that will described in Chapter10 and the mapping results for kernels obtained from the application mapping sub-flow. For each mapping, a spread sheet model containing data gathered from the synthesis of various RCF tile models was used to determine the tile needed, based on the fabric parameters used in the mapping. Next the fabric size was computed from the width and depth required from the mapping; the number of tiles is simply the product of the width and depth, since a rectangular fabric is requires. From the synthesis results the tile area is known and thus the total fabric area is the product of the tile area and the number of tiles. The fabric performance is determined by the tile performance, since the entire fabric is pipelined and all interconnect is local to a small region of the fabric. This process will be covered in more detail in Chapter 11.

7.4 Summary

The overall tool flow and three different sub-flows were discussed in this chapter. Each tool sub-flow contained some existing tools and some tools that were designed specifically for this project. Some of the existing tools had to be modified for use with HASTE, some quite substantially. The first sub-flow, the application mapping sub-flow, permits mapping from C code to executables for the various HASTE ISAs. The second sub-flow, the simulation and validation sub-flow, allows for the simulation of applications at different phases of the application mapping process, and the validation of the results produced. This sub-flow allows for extensive testing of the tool flows and different application implementations. The final sub-flow is the hardware implementation sub-flow. This allows for implementing the applications kernels as FPGA designs or as ASICs and comparison of the physical characteristics of these implementations with various HASTE fabrics.

Use of the various tool flows will allow for verification that HASTE and the HASTE tools work correctly, comparison of the different ISAs and different fabric parameters, and comparison of HASTE to competing technologies.

Chapter 8

Comparison of HASTE ISAs

As has been shown, there are three basic ISA types that can be used with HASTE architectures. This chapter will examine how effective and efficient these different ISAs are at representing applications. A set of common benchmark kernels will be implemented for a range of different fabric types and the results compared in terms of how efficient the ISA is in representing the spatial implementation of the kernel and what penalties, if any, are incurred for the sequential implementation by the use of each ISA.

8.1 ISA Metrics

There are four metrics that will be used to evaluate the different ISAs. The first is code length, the total number of sequential instructions in the executable. Since none of the HASTE ISAs have control flow in the kernels, the number of instructions executed is always the same for every iteration of the kernel, unlike the original assembly code which may have differing numbers of executed instructions depending on control flow. Code length is important since it controls the time required to execute the kernel on the sequential processor and also the time required to convert the application from the sequential form into a configuration for the RCF. The number of instructions for each kernel is initially the same, but as was shown in Chapter 4, the different ISAs usually require the addition of new instructions, not in the original kernel, that will increase the number of instructions. These added instructions include the MOVE operation for the register and relative-register ISAs and the PASS and SWAP operations for the queue ISA. The code length is the number of instructions in the original GHAL kernels plus the number of added

instructions of all types.

The second metric is the code size, which refers to the number of bytes needed to represent the application. This depends on two factors; the size of each instruction and the number of instructions (code length). Since the same basic instructions are used, the instruction size is mostly dependent on the bits needed to encode operand sources and result destinations.

The third metric, hardware utilization, measures the percentage of the fabric doing actual computation. The width of the fabric times the number of stripes determines the total number of tiles needed for the application. Only those tiles holding operations that are part of the original kernel are considered to be doing useful work, so the hardware utilization is defined as the number of instructions in the original kernel divided by the number of tiles needed. The physical characteristics of the hardware required will be discussed in detail in Chapter 11; this chapter will only consider the number of tiles required.

The fourth metric is the number of pipeline stages needed for the application, which determines the latency of the fabric. It is dependent on how close to the optimal latency the ISA characteristics allow applications to be mapped. The remainder of this section will explore each of these metrics in detail.

8.1.1 Code Length

Code length is affected by several factors. The first is of course the length of the original GHAL code. As was shown by example in the previous chapter, the GHAL code may be shorter or longer than the original assembly code. For instance, if the original code had many address calculations that are handled automatically by the stream accessors in HASTE, then the resulting GHAL code will be shorter. If the original code had significant control flow, then the GHAL code may be longer, since all of the branching is replaced by predicate calculations and select statements. All ISAs start with the same GHAL code, however, so when comparing HASTE ISAs, the length of the GHAL version of the kernel is not a factor that effects the comparison. Code length is affected by the instructions needed to control the routing of data and the placement of instructions, and the number and types of these instructions varies from ISA to ISA.

The Queue ISA generally requires many more additional instructions to implement placing and especially routing than either of the other ISAs. First, the Queue ISA instructions have

limited fanout in DAG form; each instructions can place at most two copies of its result on the queue, meaning that if that result is needed by more than two other instructions, PASS2 instructions are needed to duplicate the results. Second, additional PASS and SWAP operators are usually needed to make the DAG level-planar. Finally, many NOOP instructions may be needed to properly space the instructions in the fabric to meet the interconnect constraints.

Unlike the Queue ISA, register ISAs (RISA and RRISA) do not have any fanout limitations, since a result can be read as many times as needed until it is killed (RISA) or overwritten (RRISA). Furthermore, they do not have to have level-planar DAGs, so PASS and SWAP operators are not needed. The only additional instructions that may be needed are MOVE operators, for those cases when a result is needed in a column that is not within the read connection width of the result value's location, and rarely some NOOPs are needed to locate other operators in the fabric. This never occurs for a completely connected fabric and occurs more frequently as the connection width is reduced. The only difference between RISA and RRISA that affects code length is the fact that in general, fewer MOVE instructions are needed for RRISA implementations on fabrics with limited read interconnect, since RRISA can displace the column location of results on writes to place the values closer to where they will be consumed, whereas RISA implementations always write results to the same column that they are produced in.

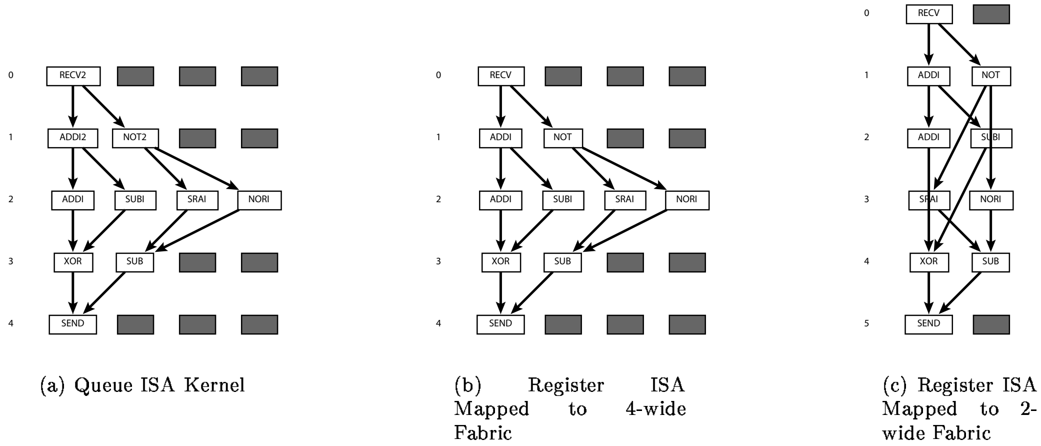
8.1.2 Code Size

For code size, it is assumed that all instructions must have widths that are multiples of one byte. QISA has variable instruction width, depending on the instruction type, while RISA and RRISA have fixed instructions widths for a given fabric target, but may have different instructions widths for versions targeted for different fabrics. Fetching instructions of variable width, or of widths different than that of the memory width, present a few problems not seen with fixed instructions widths, but solutions to these problems have been found and successfully implemented in various CISC architectures such as the Intel x86 CPUs [41]. Note that the instruction formats for each ISA were implemented by mapping the fields onto the 64-bit instruction format used in the SimpleScalar simulator. This is an implementation detail and doesn't affect the code sizes as they would be found in an actual HASTE implementation. The total code size depends on both the size of each instruction, which depends on the instruction formats, and the number

of instructions, which is the same as the code length discussed above. Note that the J-type instruction format is not actually a factor in these ISA comparisons, since jumps are not kernel legal and only kernel code is compared in these calculations.

8.1.3 Hardware Utilization

Hardware utilization is a more abstract metric than the ones discussed previously, since it is not an inherent property of a particular executable, at least for the register ISAs. Hardware utilization requires that an application be mapped to a specific fabric, and the results are dependent on not only the executable, but the properties of the fabric and the mapping algorithm as well. However, using a reasonably efficient mapping procedure and mapping to all fabrics that are reasonable for a given kernel should provide a fair comparison. Higher hardware utilization is beneficial, because then less hardware is needed to implement a given kernel; equivalently, it means that for a fixed hardware size, larger kernels can be implemented. The Queue ISA is at a large disadvantage in terms of hardware utilization because operations cannot move between levels, at least not without adding additional operations. Thus a queue kernel with one wide level will have a very low hardware utilization. Register and relative register ISAs allow operations to be moved from level to level, so they can be mapped to fabrics narrower than their widest level and thus get better hardware efficiency. Figure 8.1(a) shows a simple queue DAG with 10 operations. Since one level has a width of four, it must be mapped to a four-wide fabric. The depth is fixed as well, at 5 levels, so a 4 x 5 five fabric must be used, with only ten tiles occupied, for a hardware utilization of 50%. Figure 8.1(b) shows the same kernel as a relative register kernel mapped to a fabric with the same dimensions. Since the queue version has the same number of operations, the hardware utilization is the same. However, with a relative register ISA as shown in this example, as well as a register ISA, it is possible to map to fabrics of different widths. In Figure 8.1(c), the kernel is mapped to a two-wide fabric. A six-deep fabric is needed, but this is offset by the fact that 10 tiles out of 12 are occupied, for a hardware utilization of 83%, much better than the queue ISA version.



(a) Queue ISA Kernel

(b) Register ISA Mapped to 4-wide Fabric

(c) Register ISA Mapped to 2-wide Fabric

Figure 8.1: Hardware Utilization

8.1.4 Hardware Latency

As with hardware utilization, hardware latency is affected by the mapping process for RISA and RRISA, and is fixed by the kernel for the queue ISA. For all ISAs, the minimum latency is determined by the kernel; there must be at least as many stripes as there are levels in the DAG. Since the applications are pipelined by the fact that there is at least one register in each tile in which results are stored, there must be at least as many pipeline stages (or equivalently, stripes) as there are operations in the longest path from any input to any output. This minimum latency may not be the best from a practical standpoint, however, as the example in Figure 8.1 shows. A minimum latency mapping may have a very low hardware utilization. However, mapping to narrower widths not only increases latency, which may be a consideration in some cases, but may also increase the storage requirements in each stripe, requiring larger register files or perhaps more MOVE operations to locate values in open registers.

8.2 ISA-Specific Mapping Procedures

The application mapping process was described in detail in Chapter 7, with the exception of the mapping of DAGs to specific fabrics and ISAs. Because that mapping process is different for the different ISAs, and because those differences are pertinent to any comparison of the ISAs, the

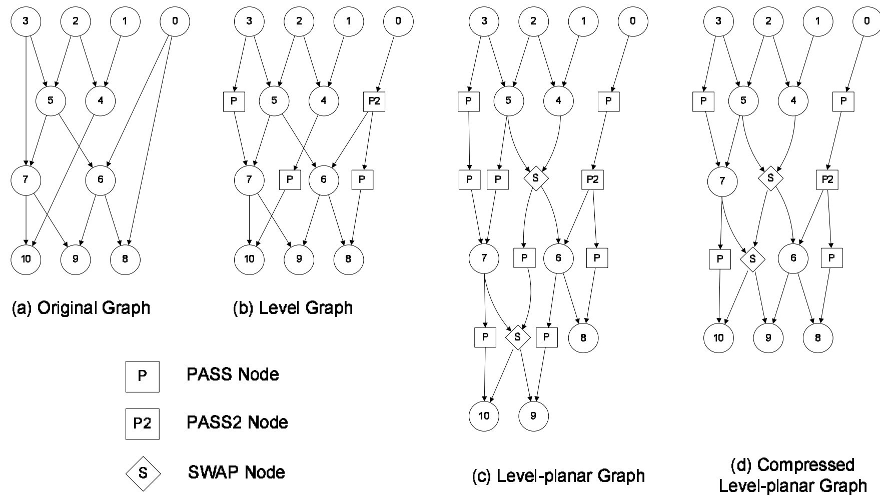


Figure 8.2: Examples showing the level and planar properties of graphs

details of that portion of the mapping process were postponed until this section. The mapping process and its effect on the ISA metrics previously discussed will be covered first for the Queue ISA and then for the Register and Relative Register ISAs. Since the process is similar for RISA and RRISA, they will be covered in a single section; differences between the two ISAs will be highlighted where necessary.

8.2.1 Queue ISA

The Queue ISA mapping procedure has three steps; in the first step a level, planar, and compressed DAG is created from the original kernel DAG, in the second step operations are assigned to specific locations in the fabric, and in the final step the ISA-specific assembly language is written out. AppMap is used to perform all three steps. The width, depth, and read connectivity of the fabric can be given to AppMap on the command line, or AppMap can determine the smallest size fabric and read connectivity that it can map a given kernel to. If the fabric size is too small or the read connectivity too limited, AppMap may not be able to find a valid mapping and another mapping will have to be attempted with a larger fabric and/or greater read connectivity.

8.2.1.1 Levelization, Planarization, and Compression

The first step in mapping a DAG to the Queue ISA is to make the DAG level-planar; this is a prerequisite for running on a queue SPU, as was discussed in Chapter 4. The AppMap program performs this process as part of the application mapping sub-flow discussed in the previous chapter. The conversion of any DAG to a level-planar DAG is broken into three phases: a levelization phase, a planarization phase, and finally a compression phase. This conversion process will be illustrated using the example shown in Figure 8.2. In order to make the graph level-planar, three operators are needed: the PASS and PASS2 operators for levelization, and the SWAP operator for planarization.

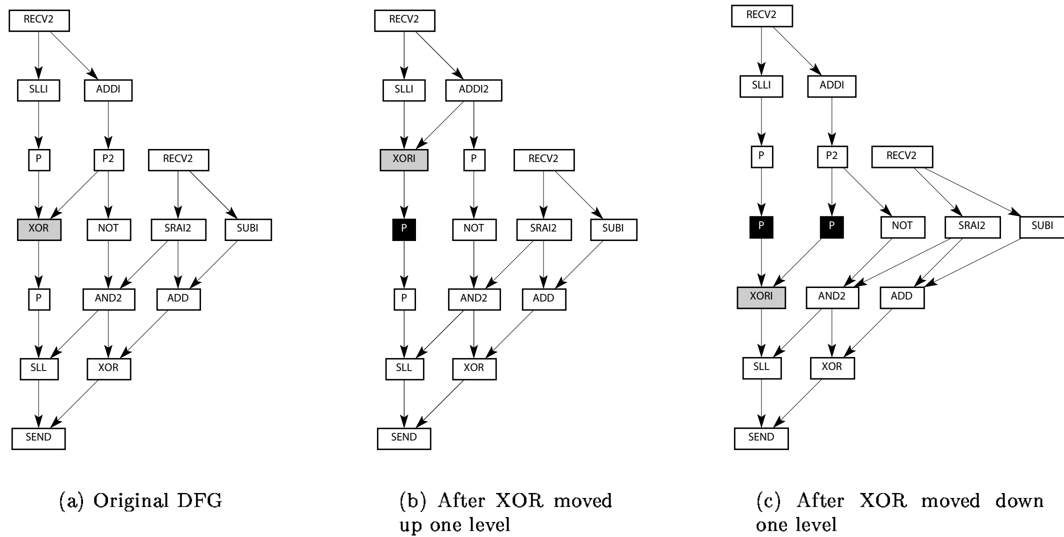
Levelization is straightforward. A topological sort of the original DAG is performed, as shown in Figure 8.2(a), so that all nodes that have a maximum path of length n from the source node or nodes in the graph that are on the nth level. In the figure, the numbered circles are graph nodes that represent GHAL operations. In a topologically sorted graph, there can be arcs that span one or more levels. For example, there are arcs from operation 0, 3, and 4 span more than one level. The PASS and PASS2 operations, shown as squares in Figure 8.2, are used to make the graph level by breaking those long arcs. Figure 8.2(b) shows the original graph after it has been made level by the addition of three PASS operations and one PASS2 operation. Note that operation 0 has two out edges. In the original graph this would be supported by having the operation be one with a '2' suffix, that places two copies of its result on the queue. If this was not changed, two PASS operations would have to be added on the second level of the graph, one for each copy of the result. Instead, AppMap changes operation 0 to a version that only places one copy on the queue and uses a single PASS2 to duplicate the result. During the levelization phase, AppMap will always make sure that result duplication is done as low in the DAG as possible in order to minimize the number of nodes added.

This level graph in Figure 8.2(b) is not planar; there are two places where edges cross. The next step in the process is thus planarization. AppMap first tries to find an ordering of nodes on each level that produces a graph that is planar, using the algorithm described in [42]. If such an ordering can not be found, AppMap uses some simple heuristics to find an ordering of the nodes on each level that minimizes the number of crossings. Then each crossings is replaced by a SWAP operator, shown as a diamond with an S in the figure. The SWAP operator can

make the graph planar, but in doing so it destroys the level property. Therefore, the process of creating a level-planar graph requires another levelization phase after SWAP operations are used to create planarity. Figure 8.2(c) shows a level-planar graph generated from Figure 8.2(b) using two SWAP operations, four PASS operations, and a single PASS2 operation. After planarizing the level graph, the final phase is compression. The levelization and planarization processes can result in the addition of unnecessary PASS nodes. AppMap fixes this problem in the compression phase by looking for cut sets of the graph composed entirely of PASS nodes and removing any such sets it finds. The removal of these nodes results in a compressed level-planar graph shown in Figure 8.2(d).

8.2.1.2 Finding Node Locations

The queue ISA does not allow much flexibility in the mapping process. Once a level planar DAG has been obtained, AppMap can make few changes to the graph. The level planar DAG form fixes the width of the fabric; it must be as wide as the widest level in the DAG. Operations on one level can not be moved to another level to reduce the width of the level, since this movement will require the addition of at least one new PASS operation on that level, keeping the width of the level the same as before in the best case, and wider than before in many cases. This is demonstrated in Figures 8.3(a,b,c). Figure 8.3(a) shows a section of a queue DAG with one wide level; in this figure, P indicates a PASS operation and P2 indicates a PASS2 operation. One might think that moving the XOR operation up or down could reduce the width of the wide level, but this is not the case. Figure 8.3(b) shows the result obtained by moving the XOR operation up one level and Figure 8.3(c) shows the result of moving the XOR operation down one level. In all cases, the width of the level for which improvement was attempted stays the same or gets worse, and other levels may be made wider as well. The depth of the fabric is also determined by the DAG structure for the queue ISA; i.e., there need to be as many fabric rows as there are levels. This depth can not be varied, since no operations will be moved from one level to another, as it has just been shown there is no benefit to doing so. If the size of the level-planar DAG is larger than that of the fabric specified for AppMap, it will return an error message stating it was unable to map the kernel to the fabric requested. If no fabric size was specified, AppMap will assume a fabric of exactly the same size as the level-planar DAG.



(a) Original DFG

(b) After XOR moved up one level

(c) After XOR moved down one level

Figure 8.3: Effect of level changes on Queue DAGs

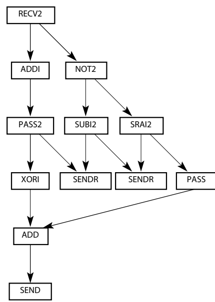
Given a level planar DAG, the only other mapping procedure that needs to be undertaken is to determine the column location for each operation. The level for each node can not be changed, and the order of the nodes on each level cannot be changed, but their location in specific columns is somewhat flexible. Given a certain read connectivity, it is necessary to position the nodes on each level such that the maximum column offset for each read is less than or equal to the read connectivity value. Note that write connectivity is not a consideration, since ALUs in static register fabrics always write to the static register in the same column as the ALU. In some cases it may not be possible to map a DAG to fabric with limited connectivity. Figure 8.4(a) shows a simple DAG. In Figure 8.4(b), a mapping is attempted for a read connectivity of 3. The darker lines going into the ADD operation show the reads possible for RC = 3, and the lighter arrows show the reads required by the application. It is clear that mapping is not possible for this example, regardless of the position of the ADD operation, since it is not possible to cover all of the required read arrows with the possible reads. A read connectivity of at least 5 would be needed to map this kernel (without reordering the nodes). A minimal required connectivity can be found for each kernel can be determined using a simple greedy algorithm, which moves the operations on each level as close to the center of the DAG as possible and then finds the largest offset. This does not provide a provable minimum read connectivity, but it has determined

the minimum read connectivity for all examples found to date. If this minimal required read connectivity is greater than the desired read connectivity supplied to AppMap, AppMap will produce an error message stating that it was unable to map the kernel to the requested fabric. If no read connectivity was supplied, AppMap will assume a fabric with read connectivity equal to the minimal connectivity found.

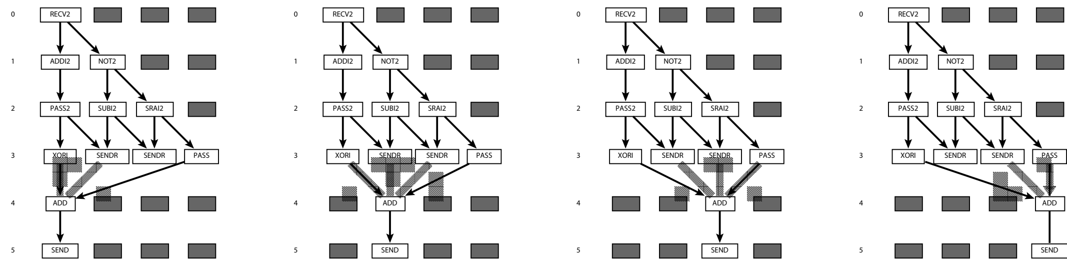
Achieving a mapping for some kernels can be done without increasing the connectivity, by adding NOOP operations in order to place nodes in columns that allow all the required reads. AppMap places the nodes on each level, using another simple greedy algorithm to insert as few NOOPs as possible to locate nodes such that all reads are possible with the given read connectivity. Figure 8.5(a) shows a DAG mapped to a fabric with RC = 3 and Figure 8.5(b) shows the same DAG mapped to a fabric with RC = 5. The first mapping requires several NOOP operations, while the second does not. Thus the code for the implementation for the fabric with RC = 3 will be longer than that for the fabric with RC = 5. However, the wider read connectivity requires a more complicated fabric interconnect, and thus larger and slower hardware. These costs will be examined in Chapter 11, however; in this chapter only the ISA efficiency metrics discussed in Section 8.1 are being considered, so one would say that the mapping for the fabric with RC = 5 is the best of the two. Mapping to a fabric with a higher value of RC would provide no benefit for this kernel.

8.2.1.3 Writing Out Assembly Code

In the final step, the mapped DAG is scanned by AppMap to find the kernel assembly code sequence. This is done by simply traversing the graph level by level and recording the operations found for each node in order. Assuming operations are issued into the fabric in order from the left, as shown in Chapter 6, the levels must be scanned from right to left in order to produce a valid instruction sequence. As long as the graph is level-planar, and the required read connectivity is less than or equal to that of the fabric, an instruction sequence produced in this manner will be transformable by the CTE into a valid fabric configuration matching the level-planar DAG. An example showing a corresponding assembly language sequence and DAG was shown in Figures 6.2(b) and 6.3(b) and will not be repeated here.

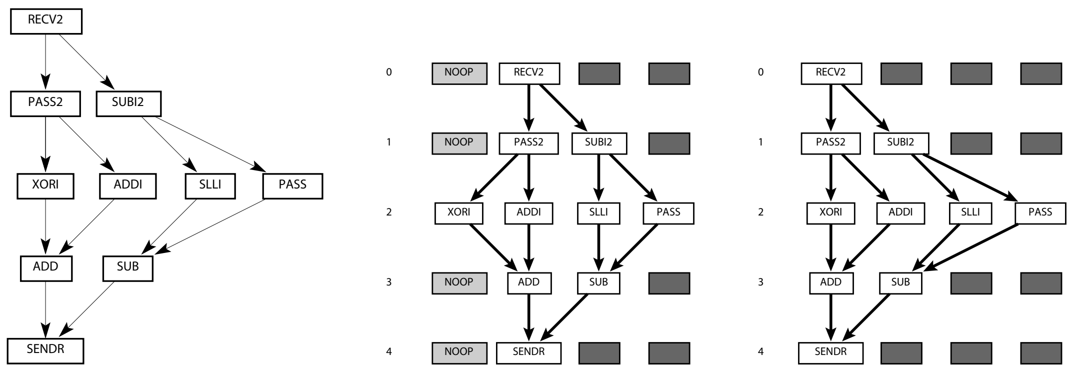


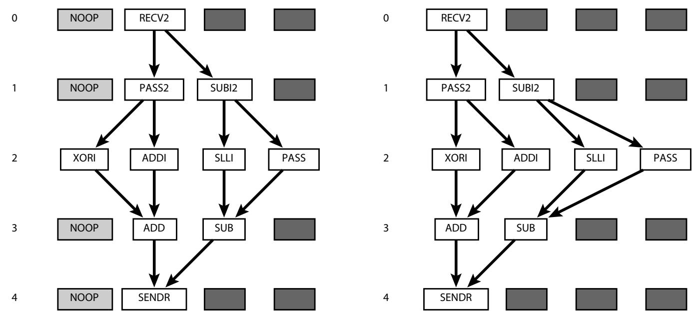
(a) Orginal DAG



(b) Mapping with RC=3; no legal position.

Figure 8.4: Effect of limited read connectivity





(a) Original DFG (b) Mapping with RC = 3 (c) Mapping with RC = 3

Figure 8.5: Use of NOOPs to Space Instructions

8.2.2 Register and Relative-Register ISAs

The mapping process for the register and relative register ISA are very similar and will thus be discussed in one section. Mapping to the register ISAs is more complicated than mapping to the queue ISA because there are more variables to be considered, and all of these variables are inter-related in complicated ways. The primary reason for this is that there are fewer limitations on the structure of DAGs for these ISAs. Unlike queue ISA DAGs, DAGs for register ISAs do not need to be either level or planar. This fact has numerous repercussions. The first comes from the ability to move nodes from one level to another without requiring the addition of new nodes. This means that there are many more possible graph levellings to consider when mapping. This movement of modes does have an effect on hardware implementation, in that moving nodes from level to level can change the number of pass registers required. The second comes from the ability to easily reorder nodes on any given level, since planarity does not need to be maintained. So for any given DAG, one can consider many different rows in which to place each node, as well as many locations for each node within each row. The mapping problem, in fact becomes the general resource-constrained scheduling problem, which is known to be intractable [43]. In addition to allowing many different mappings for a given kernel on a given fabric, there are many different fabrics to which any given application can be mapped. In addition to variations in width, depth, and read connectivity, pass register fabrics can vary in terms of their register file size. Note that write connectivity is not a parameter that needs to be considered, because it is either 1 for the asymmetric fabrics used with the registers ISA, or equal to the read connectivity for the symmetric fabrics used with the relative register ISAs. The actual mapping process is is fairly complicated. In brief, AppMap performs a greedy mapping to a minimally sized fabric. If unsuccessful, it repeats the process until successful, or until a user specified value is exceeded.

8.2.2.1 Parameter Checking

Users can specify the fabric width, depth, read connectivity, and/or register file size as parameters on the command line for AppMap, or it can map to a range of different fabrics. If a desired width is given, AppMap will find a minimum latency(depth) for the given kernel and supplied width. If a depth was also supplied, and it is less than the minimum depth found, AppMap will return an error. Similarly, if a desired depth is given, AppMap will find a minimum width for the given

kernel and supplied depth. If a width was also supplied, and it is less than the minimum width found, AppMap will again return an error. If both width and depth were supplied, AppMap performs both checks. If neither is supplied, but a desired read connectivity is, AppMap sets the width to the largest value that will guarantee full read connectivity. For RC = 3, that is a width of two, for RC = 5 that is a width of three and so on. The depth is then set to the minimum depth for that width. If neither width or depth is given, but a desired register size is, then the width is set to maximum needed width for that DAG. This is found by finding the minimum latency for the kernel and finding the width of the widest level for that latency. If both a desired read connectivity and register file size are given, then the width and depth are set as if only the read connectivity had been given. If no parameters are given, then the minimum latency and corresponding width are used.

8.2.2.2 Finding Minimum Latency

Finding the minimum latency for a kernel, as required above, is very simple. It is only necessary to assign each operation node to a level such that all of its parents are on the immediately previous level. First, nodes with no parents, or source nodes, are assigned to the first level, level 0. Next all of the child nodes of the source nodes that have no parents on other levels are assigned to the next level, level 1. This process, *ASAP Scheduling* [43], is repeated until all nodes have been assigned to a level. The name ASAP Scheduling is derived from the fact that each operations is scheduled as soon as possible. The highest-numbered level (plus one, since level numbers start at 0) gives the minimum latency. For example, the DAG in Figure 8.5(a) happens to be ASAP-scheduled. If the RECV node is on level 0, than the SEND node is on level 4 and 4 is the highest level number, so the latency is $4 + 1 = 5$ clock cycles.

8.2.2.3 Finding Lower Bounds on Width for Given Latency

A classic method from multiprocessor scheduling is useful here. Hu's algorithm [44] applies to scheduling identical processors with unit delay and this method, which will not be reviewed here, can find the minimum latency for a task graph given a fixed number of processors, or the minimum number of processors to execute a task graph in a given latency. For this application, the number of processors is equivalent to the width of the fabric and the latency is the same as

the depth of the fabric. Hu's methods give exact solutions for task graphs that are trees, but the DAGs in this work are not necessarily trees., However, the solutions do provide lower bounds on minimum latency and/or resource requirements. (See [43] for proof). So given a latency from the ASAP schedule or elsewhere, it is possible to find a lower bound on the fabric width using Hu's methods. This gives some idea of the fabric width necessary to achieve this latency; certainly is is not possible achieve the minimum latency with a narrower fabric. Similarly, given a fabric width, a lower bound on latency can be found. Although the DAGs are not perfect trees, and thus the computed minimums may not be exactly the lower bound, they are usually very close.

8.2.2.4 Check Connectivity

At this point a width and depth for the fabric has been found. All of the nodes in the DAG are next assigned to levels to produce a mapping that fits the desired width and depth. This is done in a greedy manner that simply places nodes on a level until it is full or until there are no more nodes to place on that level. Next the maximum read and write offsets are found. If the largest offset is larger than 3, or larger than the requested connectivity, than a heuristic method is used to rearrange the nodes on each level involved in the large read and/or write. This usually succeeds, but if not AppMap then tries moving first the source, and then the destination of the offending read and/or write up or down one level. If this fails and a read connectivity was specified then AppMap will return an error. If a read connectivity was not specified, than AppMap will relax the offset check to 5, and repeat the process, relaxing the connectivity as many times as needed until a valid mapping is found. While doing mappings during this phase, AppMap assumes unlimited register file size.

8.2.2.5 Register File Size Checking

At this point, a kernel mapping that meets the desired width, depth, and read connectivity has been found. The final parameter to be checked is register file size. AppMap first checks the mapping to see what the highest register usage is anywhere in the mapping. If a register size was specified and the highest register usage is no larger, AppMap does nothing. If the highest register usage is larger than that specified, or if no register size was specified, AppMap will try to optimize the register usage. AppMap first totals all register storage in each row and finds the

highest total of all rows. This value is then divided by the fabric width, with the result rounded up to the next integer value to determine the minimum possible register size. If this value is greater than the specified register file size, AppMap will repeat the mapping process with a wider fabric, until it succeeds in mapping or until a specified fabric width is reached, at which point it quits with an error. If the minimum register size is less than the specified value, or if none was specified, AppMap will still try to reorder reads and/or writes (for RRISA only) so that no register size is larger than the minimum, or larger than the specified value if one was given. If it is unable to meet a register size specification, it will repeat the process with a wider fabric, until it succeeds or until it exceeds the specified width.

8.2.2.6 Writing Out Assembly Code

The process of writing out assembly code is very similar to that described for the queue ISA. The main difference is that AppMap must also assign register numbers and include them in the assembly language. This is very different for RISA and RRISA. Register assignment for RRISA is quite straightforward, in that the register assignments are all relative, and thus the same for both the RCF fabric and the SPU. AppMap simply scans the mapped kernel, row by row, records the relative location of all register reads and writes, and then writes out the instructions and register locations in order. Any valid RRISA mapping written out correctly in this manner can be converted by a RRISA CTE to create a configuration for a compatible RCF. An example showing a corresponding RRISA assembly language sequence and DAG was shown in Figures 6.2(d) and 6.3(c). Register assignment for RISA is more complicated because the register assignments in the fabric have nothing to do with register assignments in the SPU. AppMap begins by performing conventional register allocation using graph coloring [45]. If it cannot perform a valid allocation of registers it will return an error. With the addition of the extra registers in the SPU model, this has not occurred for any of the benchmarks tested. AppMap then traverses the mapped kernel to find the last usage of each assigned register value. Finally it scans the mapped kernel, and writes out the instructions and allocated register locations in order. As with RRISA, any valid RISA mapping written out correctly in this manner can be converted by a RISA CTE to create a configuration for a compatible RCF. An example showing a corresponding RISA assembly language sequence and DAG was shown in Figures 6.2(c) and 6.3(c).

8.3 Mapping Experiments and Results

8.3.1 Queue ISA

A mapping was performed for each of 12 benchmark kernel DAGs for each read connectivity equal to or above the minimum connectivity (more information about these benchmarks can be found in Chapter 9). For each read connectivity a mapping was produced and statistics gathered by the AppMap tool. Fabric sizes were not specified; instead the minimal sizes found by AppMap were used. The results for the Queue ISA mappings produced from AppMap are shown in Table 8.1. The table shows latency and hardware utilization, as well the types of nodes added at each stage of the mapping process. Table 8.2 summarizes all of the level-planar mappings, showing all four metrics. Some benchmarks, such as fir8cpx, show a large penalty to code size and length if mapped to a fabric with low connectivity, while others, such as idea, can be mapped to any fabric, regardless of the connectivity, with no change in code size or length. In these cases only the numbers for the level-planar case are shown, since there will be no change regardless of the connectivity of the fabric they are mapped to. Changing the connectivity does not effect the latency, because as has been shown, more limited connectivity requires only the addition of NOOPs. Most of the queue implementations show a significant reduction in the hardware utilization due to process of making the application level and planar, and the additional PASS, SWAP, and NOOP operations add significantly to the code size. Four of the benchmarks, img-hp, img-erode, img-thresh, and img-out were inherently planar or level-planar and thus did not require the addition of SWAP nodes and few if any PASS nodes. These were all small benchmarks, however; in general larger kernels need many additional nodes to be made level-planar.

Table 8.1: Queue ISA Mappings

Benchmark	Version	Op & IO Nodes	Pass Nodes	Swap Nodes	NOOP Nodes	Total Nodes	Depth	HW Util %
l2alaw	Original	49	0	0	0	49	23	71%
l2alaw	Level	49	87	0	0	136	23	30%
l2alaw	Lvl.-Planar	49	104	6	0	159	34	16%
l2alaw	RC3	49	104	6	42	201	34	16%
dct1	Original	180	0	0	0	180	20	90%
dct1	Level	180	163	0	0	343	20	21%
dct1	Lvl.-Planar	180	356	94	0	630	39	13%
dct1	RC3	180	356	94	182	812	39	13%
dct1	RC5	180	356	94	78	708	39	13%
dct1	RC7	180	356	94	38	668	39	13%
dct1	RC9	180	356	94	1	631	39	13%
fir8cpx	Original	270	0	0	0	270	26	87%
fir8cpx	Level	270	284	0	0	554	26	27%
fir8cpx	Lvl.-Planar	270	2141	319	0	2730	118	5%
fir8cpx	RC3	270	2141	319	591	3321	118	5%
fir8cpx	RC5	270	2141	319	280	3010	118	5%
fir8cpx	RC7	270	2141	319	121	2851	118	5%
fir8cpx	RC9	270	2141	319	47	2777	118	5%
fir8cpx	RC11	270	2141	319	9	2739	118	5%
rgb2ycc	Original	81	0	0	0	81	16	42%
rgb2ycc	Level	81	52	0	0	133	16	36%
rgb2ycc	Lvl.-Planar	81	211	43	0	335	27	13%
rgb2ycc	RC3	81	211	43	82	417	27	13%
rgb2ycc	RC5	81	211	43	36	371	27	13%

Benchmark	Version	Op & IO Nodes	Pass Nodes	Swap Nodes	NOOP Nodes	Total Nodes	Depth	HW Util %
rgb2ycc	RC7	81	211	43	12	347	27	13%
rgb2ycc	RC9	81	211	43	2	337	27	13%
dec_cor	Original	33	0	0	0	33	14	29%
dec_cor	Level	33	12	0	0	45	14	29%
dec_cor	Lvl.-Planar	33	12	0	0	45	16	26%
dec_cor	RC3	33	12	0	20	65	16	26%
dec_cor	RC5	33	12	0	4	49	16	26%
dec_cor	RC7	33	12	0	1	46	16	26%
img_prew	Original	30	0	0	0	30	11	37%
img_prew	Level	30	11	0	0	41	11	37%
img_prew	Lvl.-Planar	30	31	11	0	72	17	25%
img_prew	RC3	30	31	11	7	79	17	25%
img_prew	RC5	30	31	11	3	75	17	25%
img_med	Original	106	0	0	0	106	29	73%
img_med	Level	106	189	0	0	295	29	14%
img_med	Lvl.-Planar	106	719	145	0	970	64	7%
img_med	RC3	106	719	145	102	1072	64	7%
img_med	RC5	106	719	145	53	1023	64	7%
img_med	RC7	106	719	145	12	982	64	7%
img_med	RC9	106	719	145	2	972	64	7%
img_hp	Original	33	0	0	0	33	20	55%
img_hp	Level	33	7	0	0	40	21	52%
img_hp	Lvl.-Planar	33	7	0	0	40	21	52%
img_thresh	Original	31	0	0	0	31	17	91%
img_thresh	Level	31	0	0	0	31	17	91%
img_thresh	Lvl.-Planar	31	0	0	0	31	17	91%

Benchmark	Version	Op & IO Nodes	Pass Nodes	Swap Nodes	NOOP Nodes	Total Nodes	Depth	HW Util %
`img_erode`	Original	16	0	0	0	16	9	89%
`img_erode`	Level	16	0	0	0	16	9	89%
`img_erode`	Lvl.-Planar	16	0	0	0	16	9	89%
`img_out`	Original	17	0	0	0	17	7	61%
`img_out`	Level	17	0	0	0	17	7	61%
`img_out`	Lvl.-Planar	17	0	0	0	17	7	61%
`img_out`	RC3	17	0	0	1	18	8	53%
`idea`	Original	530	0	0	0	530	241	73%
`idea`	Level	530	205	0	0	735	241	73%
`idea`	Lvl.-Planar	530	561	74	0	1165	498	21%

8.3.2 Register ISA and Relative Register ISA

Since there are many different fabrics that any RISA or RRISA kernel can be mapped to, deciding what fabric to map to for comparison to QISA is difficult. In order to solve this problem a methodology was developed to explore a wide range of register ISA mappings in order to systematically find mappings that give near optimal results. This methodology is implemented using the AppMap tool and a set of Perl scripts. This methodology is outlined below:

1. Find minimum latency, l_{min}, of DAG from ASAP schedule.
2. For that latency, find lower bound on width $w_{max} = w_{Hu}(l_{min})$, using Hu's algorithm.
3. Increment latency and compute new lower bound on width as above.
4. Repeat step 3 until latency, l_2, is found such that $w_{Hu}(l_2) = 2$.
5. There is now a range of widths from w_{max} to 2 that can be mapped to, and an associated lower bound on latency for each.

Table 8.2: Queue ISA Mapping Results

Benchmark	RC	Width	Code Length	Code Size (Bytes)	HW Util %	Latency
l2alaw	RC3	9	201	265	16%	34
l2alaw	RC5+	9	159	223	16%	34
dct1	RC3	37	812	900	13%	39
dct1	RC5	37	708	798	13%	39
dct1	RC7	37	668	756	13%	39
dct1	RC9	37	631	719	13%	39
dct1	RC11+	37	630	718	13%	39
fir8cpx	RC3	20	3321	3561	5%	118
fir8cpx	RC5	20	3010	3250	5%	118
fir8cpx	RC7	20	2851	3091	5%	118
fir8cpx	RC9	20	2777	3017	5%	118
fir8cpx	RC11	20	2739	2979	5%	118
fir8cpx	RC13+	20	2730	2970	5%	118
rgb2ycc	RC3	16	417	501	13%	27
rgb2ycc	RC5	16	371	455	13%	27
rgb2ycc	RC7	16	347	431	13%	27
rgb2ycc	RC9	16	337	421	13%	27
rgb2ycc	RC11+	16	335	419	13%	27
dec_cor	RC3	8	45	67	26%	16
dec_cor	RC5	8	46	68	26%	16
dec_cor	RC7	8	49	71	26%	16
dec_cor	RC9+	8	65	87	26%	16
img_prew	RC3	7	72	98	25%	17
img_prew	RC5	7	75	101	25%	17
img_prew	RC7+	7	79	105	25%	17
img_med	RC3	24	1072	1186	7%	64
img_med	RC5	24	1023	1137	7%	64
img_med	RC7	24	982	1096	7%	64
img_med	RC9	24	972	1086	7%	64
img_med	RC11	24	970	1084	7%	64
img_hp	RC3+	3	40	66	52%	21
img_thresh	RC3+	2	31	52	91%	17
img_erode	RC3+	2	16	25	89%	9
img_out	RC3	4	17	35	53%	8
img_out	RC5+	4	18	35	53%	8
idea	RC3+	5	1165	1678	21%	498

6. For each width, perform minimal latency and minimal storage mapping, as previously described, to a completely connected fabric with unlimited storage.

7. For each width there can now be found a minimal required storage sr_{min} and an achieved latency $>= l_{min}$. sr_{min} can only be reduced at the expense of increasing the latency. A minimum register file size R_{min} = ceil(sr_{min}/W) can now be computed.

8. Map to each width and every connectivity with register file size R_{min}. If any given mapping fails, increment register file size until successful.

This methodology was used for all of the benchmarks that were used in the QISA mapping experiments for RISA and RRISA. Since comparing the ISAs is the primary goal, only the data that is directly comparable to the queue results in Table 8.2 are shown in Table 8.3 for RISA and in Table 8.4 for RRISA.

8.4 ISA Performance

Composite results for each ISA type are shown in Table 8.5. For each connectivity value, the best results for each ISA were chosen and the metrics were totaled for each ISA type. The results for each metric are shown graphically in Figures 8.6, 8.7, 8.8, and 8.9. Each metric will be discussed and the ISAs compared in the remainder of this chapter.

8.4.1 Code Length

In Figure 8.6 the effect of connectivity on the code length is shown for each ISA. The value shown is the sum of the code length for each of the 12 kernels in the benchmarks set. First, it can be seen that connectivity has little effect on the code length for register ISAs. There is somewhat more of an effect for the queue ISA, as it has been shown that many NOOPs may have to be added to allow for mapping to the fabrics with more limited connectivity. There is an additional 15% more instructions needed for QISA mappings to the least connected fabrics as compared to the most connected fabrics. Perhaps most noticeable, however, is that QISA mappings create code that is on average around four times as long as the register ISAs. This has a direct effect

Table 8.3: Register ISA Mapping Results

Benchmark	RC	Width	Code Length	Code Size (Bytes)	HW Util %	Latency
l2alaw	RC3+	2	49	245	91%	27
dct1	RC3	3	180	900	100%	60
dct1	RC5	4	180	900	100%	45
dct1	RC7	5	180	900	100%	36
dct1	RC9	6	180	900	100%	30
dct1	RC11+	7	180	900	95%	27
fir8cpx	RC3	3	274	1370	95%	95
fir8cpx	RC5	4	270	1350	96%	70
fir8cpx	RC7	5	270	1350	95%	57
fir8cpx	RC9	6	270	1350	94%	48
fir8cpx	RC11	6	270	1350	94%	48
fir8cpx	RC13+	7	270	1350	89%	43
rgb2ycc	RC3	2	81	405	99%	41
rgb2ycc	RC5	3	81	405	96%	28
rgb2ycc	RC7	4	81	405	92%	22
rgb2ycc	RC9	5	81	405	85%	19
rgb2ycc	RC11+	6	81	405	79%	17
dec_cor	RC3	2	33	165	61%	27
dec_cor	RC5	3	22	165	48%	23
img_prew	RC3	2	30	150	88%	17
img_prew	RC5	3	30	150	71%	14
img_prew	RC7+	4	30	150	63%	12
img_med	RC3	2	106	530	96%	55
img_med	RC5	3	106	530	88%	40
img_med	RC7	4	106	530	83%	33
img_med	RC9	5	106	530	71%	30
img_hp	RC3+	2	33	165	79%	21
img_thresh	RC3+	2	31	155	91%	17
img_erode	RC3+	2	16	80	89%	9
img_out	RC3	2	17	85	85%	10
img_out	RC5+	3	17	85	71%	8
idea	RC3+	2	530	2650	100%	265

Table 8.4: Relative Register ISA Mapping Results

Benchmark	RC	Width	Code Length	Code Size (Bytes)	HW Util %	Latency
l2alaw	RC3+	2	49	245	91%	27
dct1	RC3	3	180	900	100%	60
dct1	RC5	6	180	900	100%	30
dct1	RC7	7	180	900	95%	27
dct1	RC9+	9	180	900	91%	22
fir8cpx	RC3	3	270	1350	100%	91
fir8cpx	RC5	5	270	1350	95%	57
fir8cpx	RC7	7	270	1350	90%	43
fir8cpx	RC9+	9	270	1350	86%	35
rgb2ycc	RC3	3	81	405	96%	28
rgb2ycc	RC5	5	81	405	85%	28
dec_cor	RC3	3	33	132	48%	23
img_prew	RC3	3	30	150	71%	14
img_prew	RC5	5	30	150	63%	12
img_prew	RC7+	7	30	150	39%	11
img_med	RC3	3	106	530	88%	55
img_med	RC5	5	106	530	71%	30
img_hp	RC3+	3	33	165	79%	21
img_thresh	RC3+	2	31	155	91%	17
img_erode	RC3+	2	16	80	89%	9
img_out	RC3	3	17	85	71%	8
idea	RC3+	2	530	2650	100%	265

Table 8.5: Composite Results for HASTE ISAs

RC	ISA	Code Length	Code Size	HW Util%	Latency
3	QISA	7209	8434	7.3	868
3	RISA	1380	6900	95.4	644
3	RRISA	1376	6847	93.8	618
5	QISA	6662	7888	7.4	868
5	RISA	1365	6880	93.8	567
5	RRISA	1376	6847	90.1	527
7	QISA	6405	7629	7.4	868
7	RISA	1365	6880	92.5	530
7	RRISA	1376	6847	87.0	509
9	QISA	6290	7514	7.5	868
9	RISA	1365	6880	90.6	509
9	RRISA	1376	6847	85.8	496
11	QISA	6247	7471	7.5	868
11	RISA	1365	6880	89.6	504
11	RRISA	1376	6847	85.8	496
13	QISA	6238	7462	7.5	868
13	RISA	1365	6880	88.7	499
13	RRISA	1376	6847	85.8	496

on SPU performance and the amount of time taken for the CTE to produce a configuration for the RCF.

8.4.2 Code Size

Figure 8.7 shows the effect of connectivity on code size. Although it has been shown that the code length is much higher for QISA, QISA also has potentially much smaller instructions, as small as 1 byte, as compared to the 4 to 6 byte instructions sizes for the register ISAs. Despite that advantage, it is clear that the QISA code is larger, as well as longer, than the register ISA code. The average instruction size for the Queue ISA is over 1.2 bytes per instruction and in combination with the much higher code length results in larger overall code size. As was shown for code length, the code size varies little for the register ISAs. Because the hardware utilization is high for the register ISAs, there is little call for NOOPs to space nodes on each level and thus the variation in code size and length is due to the addition of a very few MOVE instructions found in the best mappings. The instruction size varied little for the range of values found in the best register mappings; it was 5 bytes in all but one benchmark.

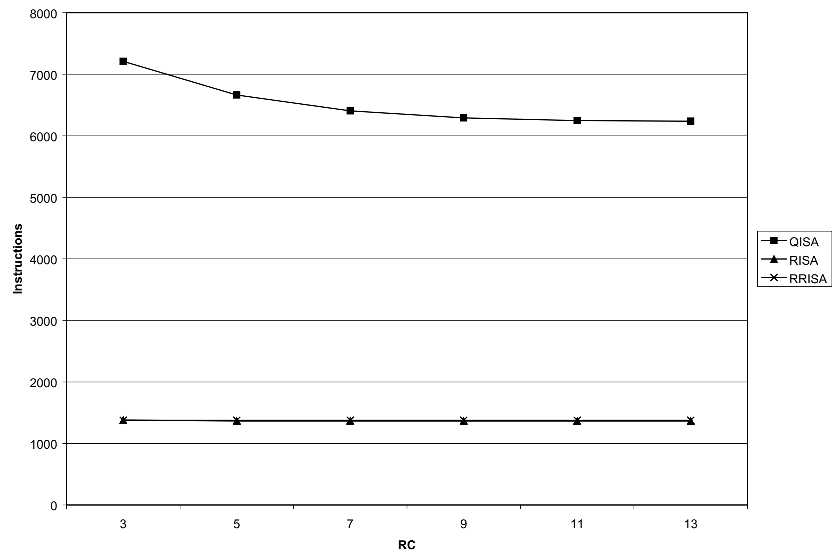


Figure 8.6: Code Length

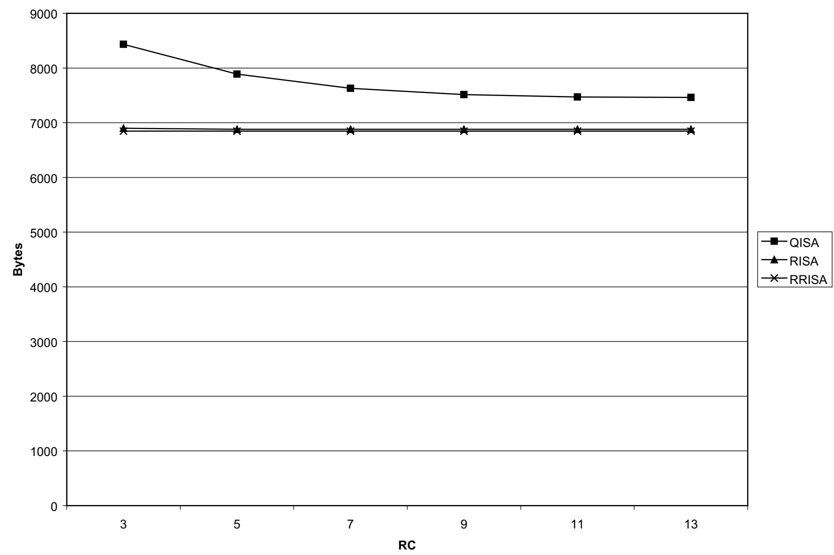


Figure 8.7: Code Size

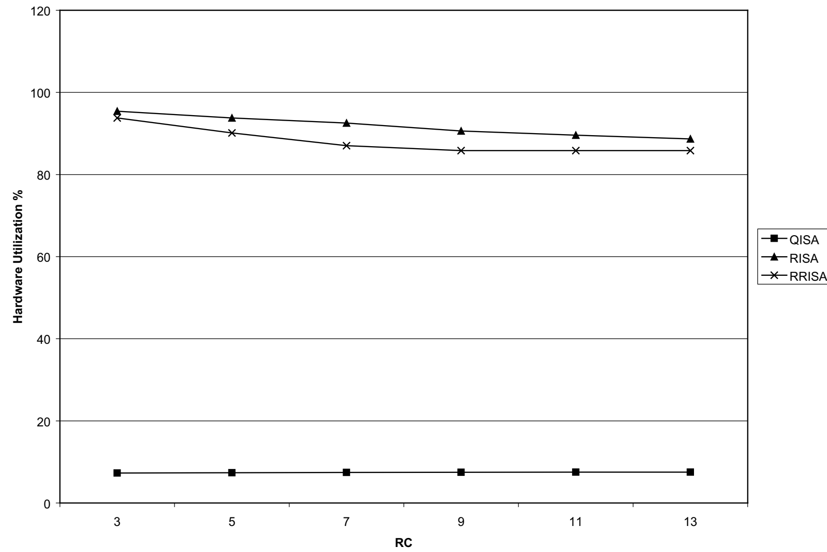


Figure 8.8: Hardware Utilization

8.4.3 Hardware Utilization

Hardware utilization is shown in Figure 8.8. As one would expect, connectivity has no effect on the QISA hardware utilization, since connectivity doesn't change the fabric size or the number of useful instructions. Connectivity has a small effect on the register ISAs, with narrower connectivity generally giving somewhat better hardware utilization. The fabric widths for the best mappings for narrower connectivities tended to be narrower, so this effect on hardware utilization is not unexpected. The best RRISA mappings had somewhat better hardware utilization than the RISA mappings. The more flexible interconnect of the RRISA fabric makes this an expected result.

8.4.4 Latency

Hardware latency is shown in Figure 8.9. As was the case for hardware utilization, connectivity has no effect for this metric for the queue ISA. Connectivity does not effect the number of levels in the queue DAG, and the latency is not changed during mapping for the QISA. Connectivity does effect latency for the register ISAs, however. Since the best mappings for narrower connectivities tended to result in narrower fabrics, this means that fabrics must be deeper for the narrower

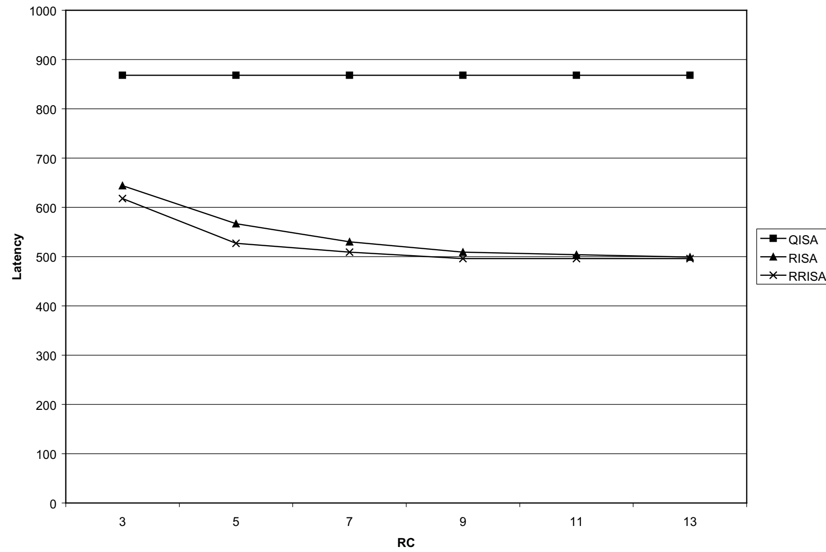


Figure 8.9: Hardware Latency

connectivities, given an essentially fixed number of nodes and high degree of hardware utilization. Therefore, there is clearly a trade off between hardware utilization and latency for register ISAs. Increasing connectivity reduces the latency but also reduces the hardware utilization as seen in Figure 8.8. Reducing connectivity has the opposite effect for both. Finally, one can see that the relative register ISA gives slightly better results than the register ISA in terms of latency. Again, this can be attributed to the more flexible interconnect of the relative register ISA.

8.5 Summary

The composite results for the three ISAs, as well as the individual benchmark results show clearly that the queue ISA is substantially worse than either of the register ISAs for all of the metrics examined. Coupled with the unconventional nature of the queue SPU and the necessity to convert all code in an application to queue form, not just the kernels, this seems to cast doubt on the queue ISA as a good choice for HASTE. However, the queue ISA requires simpler hardware for the CTE and RCF and this may give it a performance advantage compared to the other ISAs. The RRISA and RISA implementations are very close in all of the metrics, with a slight edge

going to RRISA due to the slightly better hardware utilization and latency results. The area and performance required for kernels implemented on HASTE fabrics is investigated in Chapter 11 and should further define the differences between the ISAs.

Chapter 9

HASTE Kernel, Application, and Architecture Functionality

An important goal of this project was to show that the HASTE concept was feasible and that it could be used to implement useful application kernels. To this end, 12 benchmark kernels were simulated and validated using the basic methodology outlined in Figure 7.10 and their results verified, as will be discussed in this chapter. In addition, a complete test application was implemented to show the utility of HASTE for a realistic application with multiple kernels that would be difficult to implement with other hybrid architectures. This application is an automatic target recognition (ATR) application that examines infrared images to find areas of interest that may contain vehicles. This application requires the use of a variety of different image processing techniques and the exact use of these depend on the characteristics of each individual image.

This chapter will show experimentally that HASTE kernel implementations produce identical results when mapped to the three primary HASTE ISAs and when mapped to different HASTE fabrics. While a true formal verification of HASTE implementations is likely not possible and is in any case outside the scope of this thesis, extensive experimental results should provide sufficient validation of the architecture. This section will refer to the validation of kernels as meaning that different implementations of the kernel have been shown to produce the same identical and correct results as all other implementations of the kernel.

9.1 Benchmark Kernels

A set of 12 kernels from various signal processing and image processing applications were used as benchmarks. The kernels and some of their characteristics are listed in Table 9.1. Each kernel was part of an application written in C. The C source code for all of the kernels, as well as GHAL and DFG versions, are provided in Appendix C. The kernels are briefly described below:

`l2alaw`: Converts linear PCM audio samples to compressed samples. From the G.721 benchmark in the MediaBench benchmark suite [46].

`dct1`: One dimensional discrete cosine transform. From the JPEG benchmark in MediaBench [46].

`fir8cpx`: Eight-tap, complex valued finite impulse response filter. From Texas Instruments DSP library [47].

`rgb2ycc`: RGB to YCC color space conversion. From the MPEG2 benchmark in MediaBench [46].

`dec_cor`: Image decimation and correlation filtering. By author.

`img_prew`: Prewitt edge detection. By author.

`img_med`: 3 x 3 image median filter. By author.

`img_hp`: 3 x 3 high-pass correlation filter. By author.

`img_thresh`: Multi-pixel image thresholding. By author.

`img_erode`: Morphological erosion of binary images. By author.

`img_outline`: Highlights portions of images. By author.

`idea`: IDEA encryption. From PipeRench benchmark suite [48].

9.1.1 Validation Results

All of the kernels in Figure 9.1 were simulated and validated as described previously in Section 7.2. The steps in the validation procedure are reprinted below as Figure 9.1 for reference. Validation was done throughout the development process and numerous bugs in the tool flow were identified by this validation process. Once the development process was complete, each kernel was validated at each step in the validation procedure, including validation of versions mapped to each ISA type and to a range of fabrics as appropriate. The various validation runs are listed in Table 9.2. Note that the different mappings that were validated are not listed here; however, all of the

Table 9.1: Kernel Benchmarks

Kernel Name	Application Domain	GHAL Instructions	Min Latency	Max Width
l2alaw	DSP	49	23	3
dct1	DSP	180	20	10
fir8cpx	DSP	270	26	12
rgb2ycc	Video Processing	81	16	6
dec_cor	Image Processing	33	14	6
img_prew	Image Processing	30	11	7
img_med	Image Processing	106	29	5
img_hp	Image Processing	33	20	3
img_thresh	Image Processing	31	17	2
img_erode	Image Processing	16	9	2
img_out	Image Processing	17	7	4
idea	Cryptography	530	241	3

mappings that were used in the ISA experiments in Chapter 8 were validated. For all steps and kernels, at least 10,000 iterations were simulated. In each case, for every kernel and mapping, exact duplicate outputs were generated at every step of the validation process. This shows that the compilation flow works correctly, that the CTE algorithms function properly for all ISAs and a range of application fabrics, and that the ALU fabric models correctly implements the kernels.

9.2 Large Application Implementation

In addition to the application kernels, a large application with multiple kernels was implemented and tested. Rather than compare kernel inputs and outputs, the inputs and outputs of each version of the program were compared in steps 1, 2, and 4. The parameters for the kernels being used and the order in which they are used varies according to the characteristics of the input data. This would be difficult to implement with conventional hybrid architectures, due to the overhead inherent in loading and switching configurations in most other types of reconfigurable fabrics.

9.2.1 ATR Application Description

The ATR application is a simple target recognition algorithm that scans 8-bit FLIR (Forward-Looking Infra Red) imagery for heat signatures that indicate the possible presence of a vehicular target. FLIR imagery was taken from the Fort Carson RSTA data archive [49], a standard

1. Original Code:
   (a) Annotation of kernel.
   (b) Simulation and trace generation.
2. GHAL:
   (a) Conversion to GHAL.
   (b) Simulation and trace generation.
   (c) Comparison of traces to original.
3. Kernel DAG:
   (a) GHAL listing converted to DAG.
   (b) DAG converted to HDL.
   (c) HDL simulation using original inputs.
   (d) Comparison of output traces to original.
4. Mapped Version:
   (a) DAG mapped to specific ISA and fabric.
   (b) Mapped assembly generated.
   (c) Mapped assembly simulated and traces generated.
   (d) Comparison of traces to original.
5. Implemented DAG
   (a) Mapped assembly converted to fabric-specific DAG using CTE algorithm.
   (b) Fabric-specific DAG converted to HDL.
   (c) HDL simulation using original inputs.
   (d) Comparison of output traces to original.

Figure 9.1: Steps in Validation Process

Table 9.2: Validation Runs

Benchmarks	Step 1	Step 2	Step 3	Step 4 Queue	Step 4 Register	Step 4 Rel. Reg.	Step 5
l2alaw	√	√	√	√	√	√	√
dct1	√	√	√	√	√	√	√
fir8cpx	√	√	√	√	√	√	√
rgb2ycc	√	√	√	√	√	√	√
dec_cor	√	√	√	√	√	√	√
idea	√	√	√	√	√	√	√
img_prew	√	√	√	√	√	√	√
img_med	√	√	√	√	√	√	√
img_hp	√	√	√	√	√	√	√
img_thresh	√	√	√	√	√	√	√
img_erode	√	√	√	√	√	√	√
img_outline	√	√	√	√	√	√	√
√= all correct							

dataset often used in computer vision research. The algorithm used for this example are fairly rudimentary, but the image processing kernels used are representative of those used in many contemporary ATR and computer vision applications. The basic flow of the application is shown in Figure 9.2. The source code for this program is in Appendix F. The gray rounded rectangles correspond to kernels that can be implemented using the current HASTE infrastructure. The other block correspond to code that is sequential only. The example image is used to show the effect of the different kernels in the applications. For testing and verification a set of 20 256 x 256, 8-bit FLIR images in PGM format was used an input set. The program produced 20 8-bit images of the same size with identified targets segmented out. Rather than use kernel output traces to validate the operation of the overall program, the output images produced by the program in validation steps 2 and 4 were compared to those produced in step 1. Since the entire program cannot be simulated in steps 3 and 5, and each individual kernel was verified previously, those steps were not implemented for this application.

The application begins with a median filter to remove noise from the image. The median filter replaces the value in each pixel with the median value of the pixel and the eight pixels surrounding it. The median filter acts similar to a low-pass filter, with the added advantage that extremely high or low values for a pixel will not skew the filtered values for adjacent pixels. The next kernel applies a sharpening, or high-pass filter, which accentuates areas with high contrast,

which occurs when a hot object such as a vehicle exhaust is in close proximity to a cooler region. This filter is performed by convolving the image with a 3 x 3 filter mask. While these high contrast regions may occur randomly, high contrast regions caused by target vehicles are usually characterized by straight edges, so an edge detection algorithm is applied in the next kernel. Prewitt edge detection convolves the image with two 3 x 3 masks and sums the absolute value of the results for each mask to create the output pixel value.

The next step is to threshold the image to find all pixels with sufficiently high values to be of interest. If a pixel value is above the threshold i is set to a value of 1; otherwise it is set to a value of 0. Since it hard to determine a correct threshold value *a priori*, a relatively low value is used initially. The thresholding is done in a kernel with a variable threshold. This is implemented by simply storing the threshold value in a register or on the queue before calling the kernel. It is recognized as a live-in value by the CTE and implemented as a constant in RCF configuration. After thresholding, sequential code analyzes the image to find the number of 'on' (equal to one) pixels and the number of isolated 'on' pixels. After this step, the algorithm either goes to the next step or calculates a new threshold and repeats the thresholding operation. An image thresholded with a good value should have some 'on' pixels, but few isolated 'on' pixels, as shown in the example. An erosion operation is then applied to the binary image. Erosion is a morphological operation that in effect 'thins' regions of on pixels, so that only large clusters of 'on' pixels survive. The locations of these clusters are assumed to be the locations of targets. In order to show these regions, the final kernel "outlines" them by setting all pixels in the original image not near a target location to black, leaving those pixels near the target region the same as in the original image. For efficiency, slightly different versions of this final kernel are present for images having different numbers of target regions. Since the cost of including multiple versions of simple kernels is relatively low, as each is represented as at most a few dozen lines of assembly code, it is feasible to do so. Including many different FPGA configurations would be much more expensive.

The results of simulation and validation for the large application showed that that the exact same results were obtained at Steps 1, 2, and 4 of the validation process. Both kernel code and sequential code was simulated at each step, showing that large applications can run on (simulated) HASTE hardware.

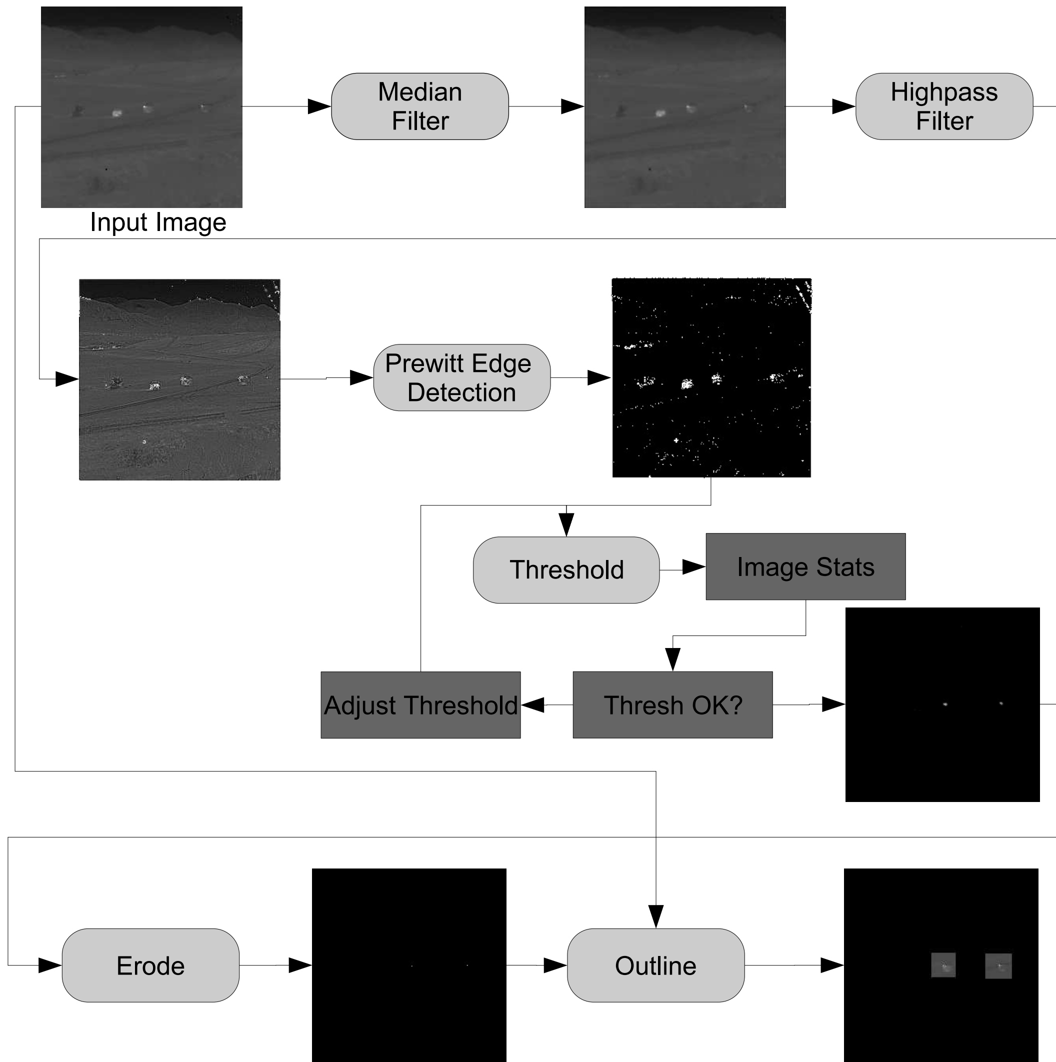


Figure 9.2: ATR Application Flow

9.3 Observations

The validation methodology covered here showed that it was possible to produce versions of each kernel at each step of the HASTE tool flow that produced correct results. Perhaps most importantly, this validates the ability of the CTE algorithms to produce configurations that perform the desired functions correctly. In addition, the correct operation of the entire HASTE tool flow has been demonstrated. Finally, the ability to implement an application with multiple kernels shows the ability of HASTE to be used for more realistic applications than the single kernel benchmarks first presented. Now that the correct operation of HASTE has been demonstrated, the next two chapters will compare HASTE implementations to other hardware implementations of the kernels.

Chapter 10

Hardware Modeling and Synthesis

In order to assess the usefulness of the HASTE concept, investigation of the physical characteristics of HASTE architectures was needed. Given that the HASTE concept describes a family of architectures, not a single architecture, it was clear that in order to properly investigate HASTE, the implementation characteristics of a wide range of these architectures would need to be considered, rather than focusing on a single architectural instance. While in practice a HASTE architecture would be likely implemented as a full-custom design, the experiments in this thesis use standard-cell implementations. This allowed for the exploration of a large design space and enabled the gathering of information concerning the effect of various parameters of the architecture on die area and performance. An actual HASTE architecture could have much better performance and much smaller die area; one study shows that carefully designed custom hardware implementations can be twice as fast and half as large as standard-cell implementations [50]. However, the relative performance of different implementations should be similar, regardless of the implementation methodology.

All of the HASTE RCF implementations studied follow the fabric model introduced in Chapter 5. In particular, this chapter will investigate the three primary fabric styles in that chapter, and will investigate a range of values for the parameters relevant to each. Physical implementations of the CTE or SPU will not be investigated in any great detail, however. Creating a synthesizable SPU model would be quite time-consuming, particularly in the case of the queue ISA SPU, and little would be learned. In general, it is unlikely that any useful comparisons could be

obtained between different versions of SPUs or between SPUs and competing implementations. As to comparing different SPU models, there are only a few different parameters to explore, and these are not particularly significant. The most basic assumptions about the SPU in terms of the specifics of the ISA, word-size, and so on, would be static, so there would be little to be gained from a comparative study. As to the second point, a useful comparison to other CPU implementations would be difficult to obtain, since real CPUs are implemented with extensive custom design, and any advantages or disadvantages of the SPU design compared to conventional CPUs would be overshadowed by difference in implementation. For context, however, a MIPS core was synthesized for the same standard-cell library used elsewhere in this thesis and those results will be presented here. As for the CTE, the CTE functionality must match all of the fabric parameters, and design of a synthesizable, completely parameterizable model of it has proven difficult. Given that the CTE algorithms are relatively simple, and that the CTE is relatively small, the physical characteristics of the CTE were not explored in any depth. A few representative CTE instances were synthesized, but no large-scale investigation was done of the design space. In the future, a research project investigating HASTE architectures might well focus on specific microarchitectural details and optimizations of the CTE and SPU, but that is beyond the scope of the present research. Finally, the data available shows the the size of the CTE and SPU are relatively small and fast compared to all but the very smallest RCF, so focusing primarily on the characteristics of the RCF seems justified.

This chapter will investigate the area and delay of a wide range of fabrics, by synthesizing parameterizable models to a commercial 90-nanometer standard cell library. All three styles of RCF fabric will be investigated, as well as three multiplier options for each. The results of mapping applications to the different RCF fabrics will not be covered here, but the results in this chapter will be used for such evaluations in Chapter 11.

10.1 ALU Design

All of the HASTE ISAs use the same basic RCF ALU design, one which can implement all of the kernel-legal HASTE operations. A parameterizable VHDL model of the HASTE ALU was designed, with three variations: no multiplier, 16-bit multiplier, and 32-bit multiplier. Where possible, the Synopsys Designware IP blocks were used for arithmetic functions; these are specially

optimized designs that work with Synopsys Design Compiler to produce particularly small and/or fast designs, depending on the constraints imposed by the user. Synopsys Design compiler can pick from many different logical structures for specific arithmetic functions in order to meet user constraints. For instance, Designware adders can be implemented with any of the following structures: ripple-carry, carry-look-ahead, fast carry-look-ahead, Brent-Jung, conditional-sum, ripple-carry-select, or parallel-prefix.

While multipliers are expensive in terms of area and delay, they are needed by many applications relevant to HASTE. The PISA ISA specifies a 32-bit multiply with a 64-bit result. This 64-bit result is written to special registers, HI and LO. These special registers can cause problems for HASTE implementations since it may be difficult for the CTE to locate results in the fabric when these special registers are used. GHAL solves this problem by allowing 4-operand instructions, which allow the designation of specific general-purpose registers for both portions of the 64-bit result. The 32-bit multiplier is expensive in terms of both area and delay, and including one in each tile could make HASTE fabrics impractical. One option would be include a multiplier in only some tiles in each stripe. This presents some significant problems in that it makes CTE design more complex and complicates the mapping process. It is still a viable option, but this kind of heterogeneous fabric will not be discussed here. Another solution is to use a smaller multiplier. A 16-bit multiplier, producing a 32-bit result, is roughly a fourth the size of a 32-bit multiplier. 32-bit multiplies can be composed from these smaller multiplies if needed. PISA does have a pseudo-instruction, MUL, which produces a single 32-bit result that can be stored in any register. It is not actually a 16-bit multiply, however; it assumes a full 32-bit multiply and then takes the low 32 bits and moves them from the LO register to the designated register. The GHAL ISA eliminates the HI and LO registers and promotes MUL from a pseudo-instruction to a full instruction performing a true 16-bit multiply. The 32-bit MULT instruction was kept as part of the GHAL ISA. Three versions of the ALU were evaluated, one with no multiplier, one with a 16-bit multiplier, and one with a 32-bit multiplier. Regardless of which ALU design was used, GHAL itself was unchanged; only which instructions were kernel-legal changed.

To investigate the cost of including the different multipliers in the fabric, VHDL versions of the three different ALU variations were synthesized to the same standard cell library used for the entire fabric. A minimum delay version, a minimum area version, and three intermediate

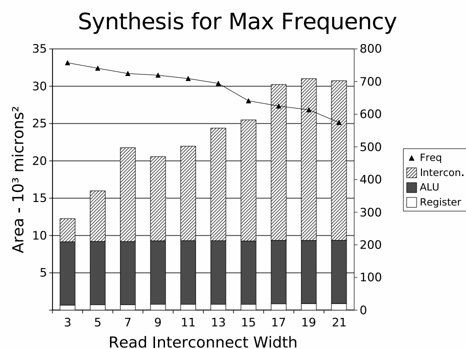
Table 10.1: ALU Synthesis Results

ALU Type	Clock Period	Frequency (MHz)	Total Area (μ^2)
no_mul	0.60 ns	1667	14,235
"	1.99 ns	503	5,695
"	3.00 ns	333	5,054
"	3.89 ns	257	4,826
"	4.12 ns	243	4,718
mul	1.30 ns	769	33,801
"	1.93 ns	518	14,218
"	2.97 ns	337	13,664
"	3.54 ns	282	11,503
"	3.65 ns	274	11,367
mul64	1.40 ns	714	58,147
"	1.92 ns	521	36,100
"	2.97 ns	337	30,431
"	3.57 ns	280	29,787
"	3.76 ns	266	27,001

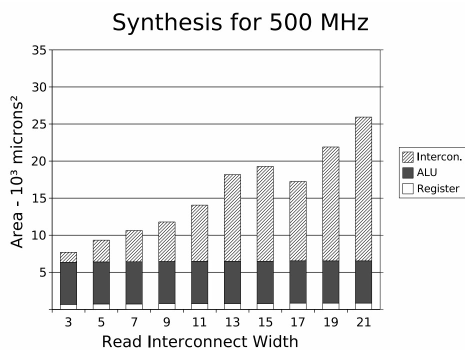
versions were synthesized for each type of ALU, which are heretofore referred to as no_mul, mul, and mul64 (for the ALU with no multiplier, ALU with multiplier with 32 bit result, and ALU with multiplier with 64 bit result). The results are include in Table 10.1. As expected, including a multiplier is expensive, particularly the full width multiplier. The extra area costs are worst for the fastest ALUs, with the mul version being twice as large as the no_mul version, and the mul64 version being twice again as large as the mul version.

10.2 Static Register Fabric

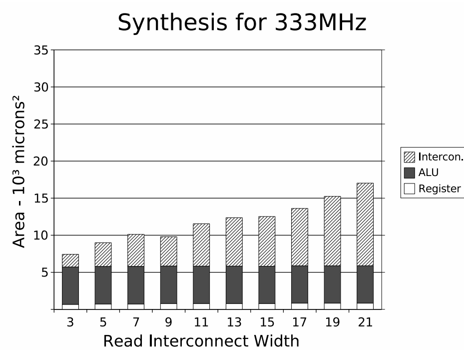
The queue ISA requires that all operands that are produced on a level i and consumed on a level j, with $j > i + 1$, must be passed from level to level using explicit PASS operations. Therefore, pass registers are not needed to map a queue ISA if those pass operations are implemented in functional units. Without having some way to direct the result produced by a PE to a specific column other than the one that the PE is located in, the result must be stored in a register located in the same column. Allowing results to be stored in a different column could allow for more efficient use of the fabric, but it is not clear how this could be accomplished with a queue ISA. The relative register ISA does allow for this, but it requires an explicit operand



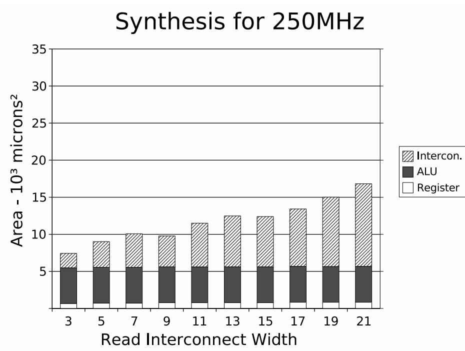
(a)



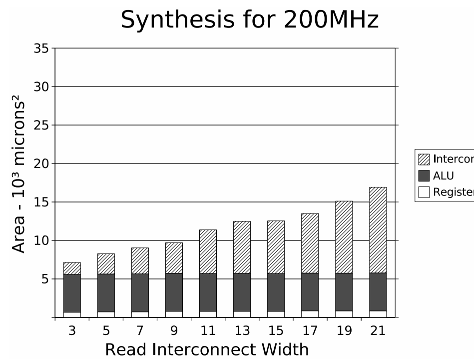
(b)



(c)

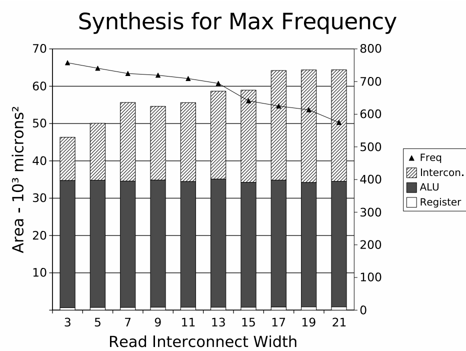


(d)

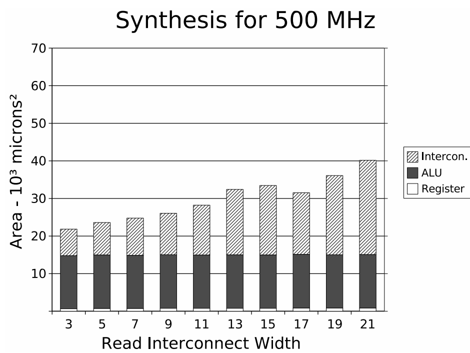


(e)

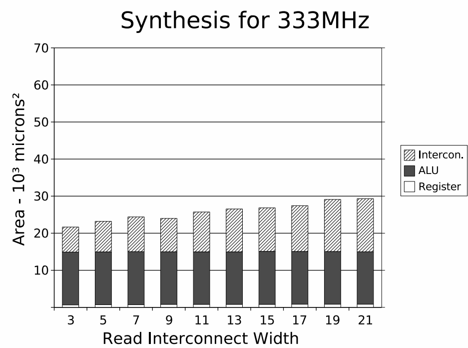
Figure 10.1: Static Register Fabric Synthesis: No Multiplier



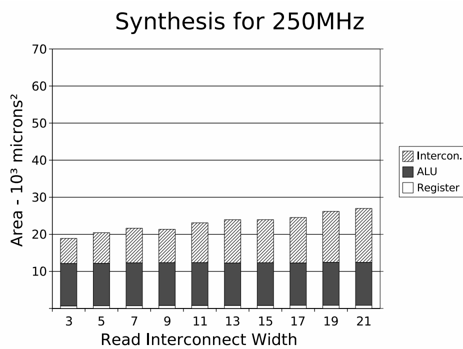
(a)



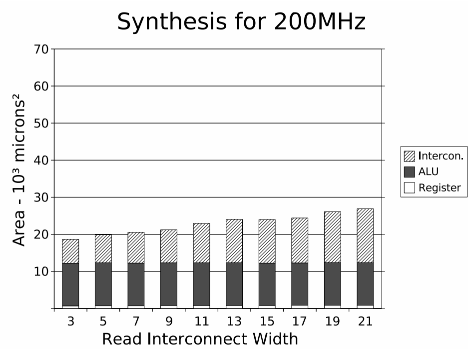
(b)



(c)

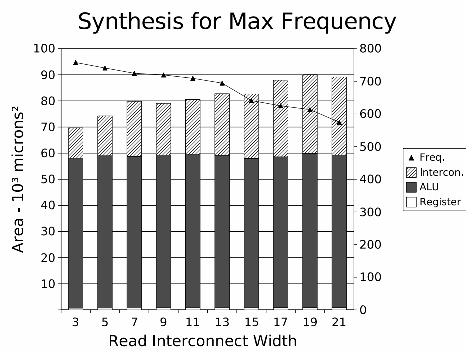


(d)

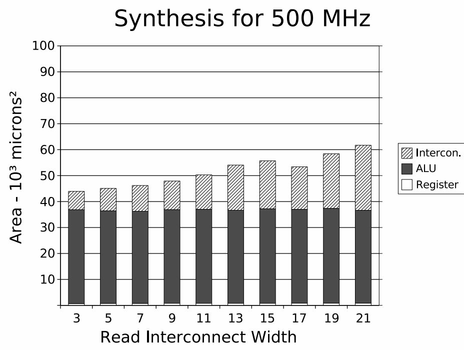


(e)

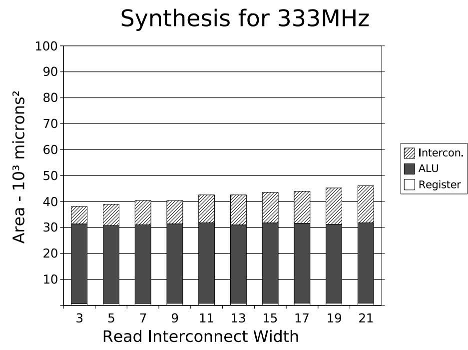
Figure 10.2: Static Register Fabric Synthesis: 32-Bit Multiplier



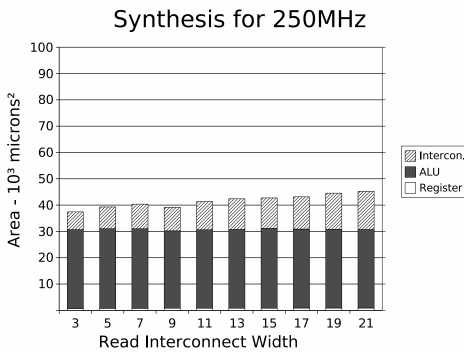
(a)



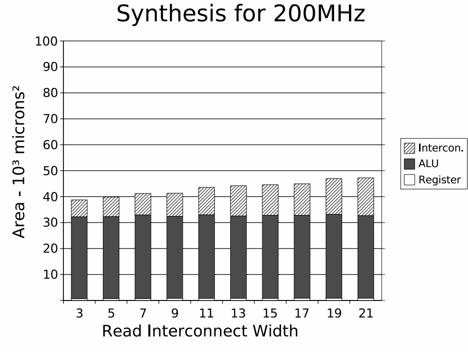
(b)



(c)



(d)



(e)

Figure 10.3: Static Register Fabric Synthesis: 64-Bit Multiplier

designation, which specifies the column offset. It is possible that a multiple-pass CTE algorithm could allow for the automatic determination of column offset for PE results in a queue ISA, but given the relatively limited offsets found in queue ISAs implementations, due to the leveled-planarity requirement for the DAG, it is not clear that the advantage gained would be worth the large increase in complexity required in the CTE. So the static register fabric described previously, with the number of registers equal to one and the write connectivity equal to one, is the fabric type best suited to the Queue ISA.

The only tile architectural parameter one needs to be concerned with for static register fabric implementations for the queue ISA is thus the read connectivity, RC, which must be ≥ 3. Of course, the global parameters, width, depth, and word width can vary, but this is true of all HASTE architectures and need not be considered here (A 32-bit fabric was assumed throughout this chapter). The other factors that need to be considered when synthesizing tiles for queue ISAs are the target clock period and the ALU type. As discussed previously, there are three different ALU types, one with no multiplier, one with a 32 bit multiplier and one with a 64-bit multiplier. Using a parameterizable VHDL tile model for this static register fabric, synthesis runs for RC values from 3 to 21 for all three ALU multiplier options, and for a range of clock frequencies, were performed. Using a detailed floorplan model, these results were adjusted as needed to account for wiring delays and for wiring-density induced area requirements.

The static register tile synthesis results are summarized graphically in Figures 10.1, 10.2, and10.3, which show the synthesis results for static register fabric tiles with no multiplier, a 32-bit multiplier, and a 64-bit multiplier, respectively. Each subfigure shows the area of the synthesized tile area on the y-axis, broken down into area consumed by registers, ALU, and interconnect. These areas are shown for different read connections and the different subfigures show the results for various clock frequencies. The area scale is the same for all subfigures of each figure. The first subfigure, (a), in each case shows the results obtained when synthesized for maximum speed; the clock frequency reached is shown as a separate data series on the chart. The speed is fixed for the remaining subfigures at 500 MHz, 333 MHz, 250 MHz, and 200 MHz, corresponding to clock periods of 2 ns, 3 ns, 4 ns, and 5 ns. The most obvious result is that the tiles with the 64-bit multiplier are larger than the tiles with 32-bit multipliers, which are in turn larger than the tiles with no multipliers. Another obvious results is that the tiles with higher RC

values require more interconnect area, with interconnect area for the RC = 21 case ranging from 200% to 1400% of the RC = 3 case. Total tile areas range from just over 7,000 μ^2 for the slowest, least-connected tiles with no multipliers, to over 90,000 μ^2 for fast, highly connected tiles with 64-bit multipliers.

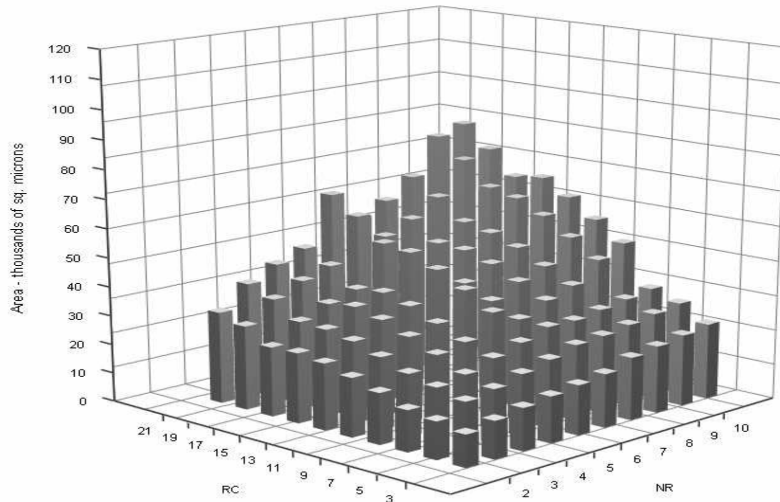
10.3 Asymmetric Pass Register Fabric

For both of the pass register fabrics, both the number of registers, NR, and the read connectivity, RC, need to be considered. From these two values, all of the parameters can be derived for these fabrics, as was discussed in 5.5. The variation in the number of registers in the pass register file is the most important difference as compared to the static register fabric. Read connectivity varies as it did with the static register case. Since there is a very large design space, the pass register architectures were divided into two separate components. A parameterized VHDL model was created for each of these components, one containing the pass register file and related interconnect, and the other containing the ALU and remaining interconnect.

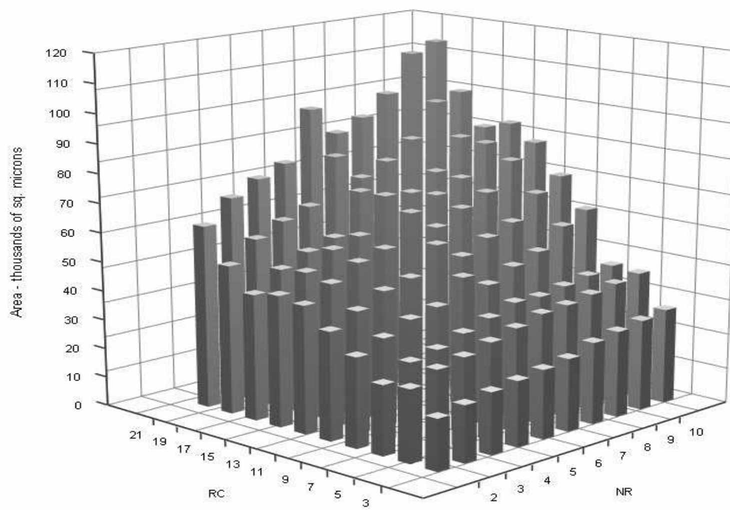
Using these two components, asymmetric pass register tiles were synthesized for all possible combinations of values with $3 \leq NR \leq 10$ and $3 \leq RC \leq 21$. These were synthesized for minimum delay, minimum area, and for delay combinations targeting overall tile delays of 2 ns, 3 ns, 4 ns, and 5 ns, using all three ALU possibilities. Nearly three thousand different parameter combinations were synthesized. Figures 10.4 (a) and (b) show an overview of the important aspects of this data. Shown are the areas for the minimum area and minimum delay runs for each value of NR and RC. The values shown are for tiles with no multipliers; there is little difference in the data for the other multiplier cases, except that the areas are shifted a fairly constant amount, which is explained by the area consumed by the multiplier. The magnitude of these areas are consistent with those shown in Table 10.1.

10.4 Symmetric Pass Register Fabric

The synthesis runs for the symmetric pass register fabrics were very similar to those done for the asymmetric pass register fabrics. Again, tiles were synthesized for all possible combinations of values with $3 \leq NR \leq 10$ and $3 \leq RC \leq 21$, for all three multiplier options, and for minimum area,

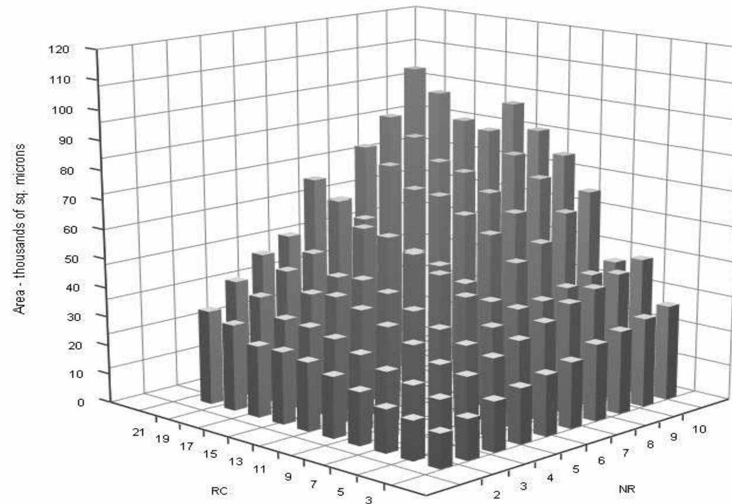


(a) Minimum Area

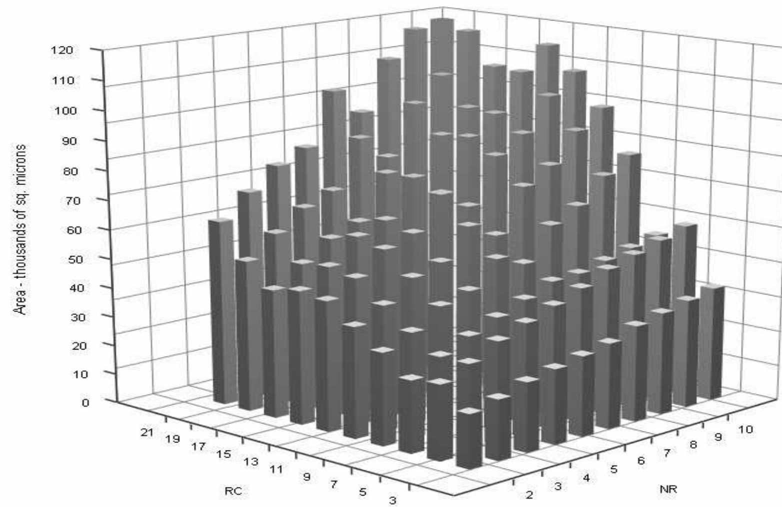


(b) Minimum Delay

Figure 10.4: Asymmetric Pass Register Fabric Synthesis



(a) Minimum Area



(b) Minimum Delay

Figure 10.5: Symmetric Pass Register Fabric Synthesis

minimum delay, and four intermediate delay values. Figures 10.5 (a) and (b) show an overview of this data. As for the asymmetric case, the minimum area and minimum delay runs for each value of NR and RC are shown. The values shown are for tiles with no multipliers; the same effect of the other multiplier options was observed for these fabrics as was observed for the asymmetric pass register fabrics.

10.5 HASTE Components

Table 10.2 shows areas and delays for a range of HASTE components, all optimized for best performance. The SPU shown is not a true HASTE SPU, but rather a simple MIPS compatible design found on an open source hardware website [51] and synthesized for the same standard cell library used for the fabrics. It is very similar to a RISA SPU, however. The CTE values are simple implementations written in VHDL and are representative of a "typical" CTE. One can see that the static register tiles can be smaller and faster than the pass register tiles, but the differences are not great if the tiles being compared have the same values for RC and NR is not large. Since the tile area and performance does vary so much depending on the parameters, it is hard to draw conclusions without seeing the results obtained when mapping kernels to fabrics with specific parameters. This will be done in Chapter 11. It is interesting to note that the CTE and SPU are quite small compared to the areas for fabrics of any but the most trivial sizes. Even a 4 by 5 fabric of the smallest tile shown is larger than the sum of the areas of the SPU and either of the CTEs. This supports the decision to concentrate on the physical characteristics of the fabrics and not the CTE and SPU.

Given the extra complications involved in using the full-width multipliers, since they require two separate outputs, as well as the fact that few if any of the benchmarks considered for this project required a full-width multiply, these synthesis results confirmed a decision to primarily use fabrics with half-width multipliers or no multipliers. The gains from using full-width multipliers are small and the costs are high. While using a half-width multiplier is uncommon in conventional CPU architectures, it is a trade-off that is used in many DSP architectures. 32-bit DSP architectures such as the Texas Instruments TMS320C28x [52] and the Analog Devices TigerSHARC [53] provide only 16-bit multiply-accumulate units. These can be used to perform wider multipliers, which can also be done in HASTE using standard GHAL instructions. Many

Component	Description	Area, μ^2	Freq., MHz
SPU	Simple pipelined MIPS core with full-width multiplier	137,196	650
CTE	10-Column QISA CTE	67,012	742
CTE	10 Column Register ISA CTE	81,010	731
RCF Tile	Static Register Fabric, RC=3, no_mul	12,224	758
RCF Tile	Static Register Fabric, RC=21, no_mul	30,774	675
RCF Tile	Static Register Fabric, RC=3, mul	46,586	758
RCF Tile	Static Register Fabric, RC=21, mul	63,948	673
RCF Tile	Asymmetric Pass Register Fabric, NR=2, RC=3, no_mul	17,453	662
RCF Tile	Asymmetric Pass Register Fabric, NR=2, RC=21, no_mul	62,523	658
RCF Tile	Asymmetric Pass Register Fabric, NR=10, RC=3, no_mul	30,954	633
RCF Tile	Asymmetric Pass Register Fabric, NR=10, RC=21, no_mul	109,547	629
RCF Tile	Asymmetric Pass Register Fabric, NR=2, RC=3, mul	51,749	660
RCF Tile	Asymmetric Pass Register Fabric, NR=2, RC=21, mul	98,878	655
RCF Tile	Asymmetric Pass Register Fabric, NR=10, RC=3, mul	64,601	630
RCF Tile	Asymmetric Pass Register Fabric, NR=10, RC=21, mul	150,220	620
RCF Tile	Symmetric Pass Register Fabric, NR=2, RC=3, no_mul	18,134	633
RCF Tile	Symmetric Pass Register Fabric, NR=2, RC=21, no_mul	63,203	629
RCF Tile	Symmetric Pass Register Fabric, NR=10, RC=3, no_mul	36,396	599
RCF Tile	Symmetric Pass Register Fabric, NR=10, RC=21, no_mul	128,736	585
RCF Tile	Symmetric Pass Register Fabric, NR=2, RC=3, mul	53,004	620
RCF Tile	Symmetric Pass Register Fabric, NR=2, RC=21, mul	99,558	620
RCF Tile	Symmetric Pass Register Fabric, NR=10, RC=3, mul	69,833	615
RCF Tile	Symmetric Pass Register Fabric, NR=10, RC=21, mul	167,946	580

Table 10.2: Typical HASTE Component Areas, Min delay

of the benchmarks do not need multipliers at all, since any multiplications in these kernels are by small constant values and are efficiently implemented using shifts and adds.

10.6 Observations

As one would expect, fabrics with wider interconnect (larger values of RC) are somewhat slower than those with more narrow interconnects. However, the differences are not as large as might be thought. For the maximum frequency static register fabrics, the widest interconnect fabrics, with RC = 21, were at most 24% slower than the narrowest interconnect fabrics with RC = 3. The effect of interconnect width on speed were even less for the other fabric styles, less than 8% for the asymmetric fabric and less than 10% for the symmetric fabric. The effect on area was more dramatic, especially for the minimum area implementations, where more optimization was possible to take advantage of the less complicated interconnect in fabrics with lower connectivities. This can be seen in Figure 10.1(b), where the area varies by a factor of nearly 4 across the range

of RC values from 3 to 21. This is harder to see in Figure10.3, where the size of the 64-bit multiplier, which is independent of the interconnect width, dominates the total area. A similar range in areas can be seen as the connectivity is changed for the pass register fabrics in Figures 10.4 and 10.5. The interconnect value has a larger effect on area than the number of registers, although fabric areas can vary by a factor of nearly 2.5 as the register file size ranges from 2 to 10. The full meaning of these hardware results will become clear once actual kernels are mapped to different fabrics and the various tradeoffs inherent in mapping to different fabrics can be seen. This will be explored in the next chapter.

Chapter 11

Area and Performance of Kernel Implementations

The final set of experiments in this thesis compares implementations of kernels in HASTE fabrics to implementations of those same kernels in a commercial FPGA and in a commercial 90 nm standard-cell ASIC technology. The experiments were constructed so as to be as conservative as possible; i.e. any assumptions made tended to make the ASIC and FPGA implementations look smaller and/or faster and tended to make the HASTE implementation look larger and/or slower than might likely be observed in practice. This chapter will detail the procedures used to determine area and performance of an FPGA implementation, an ASIC implementation, and implementation in each of the three HASTE ISAs on a range of fabrics, for each of the benchmark kernels. Each of the first three sections will cover the procedures used and then review the results for one the three implementation technologies. The final section of this chapter will compare and analyze all three technologies. The overall design tool flow for kernel implementation was introduced in Chapter 7 and Figure 11.1 is reproduced from that chapter for reference.

11.1 ASIC Implementations

An alternative to implementing the benchmark kernels in programmable logic would be to use a standard semi-custom design flow and target the design to a standard-cell library. This is the industry standard approach for designs which need ASIC performance but which cannot justify the time and cost of full-custom design. It was expected that the ASIC implementations would be significantly smaller and faster than either the HASTE or FPGA implementation, but it

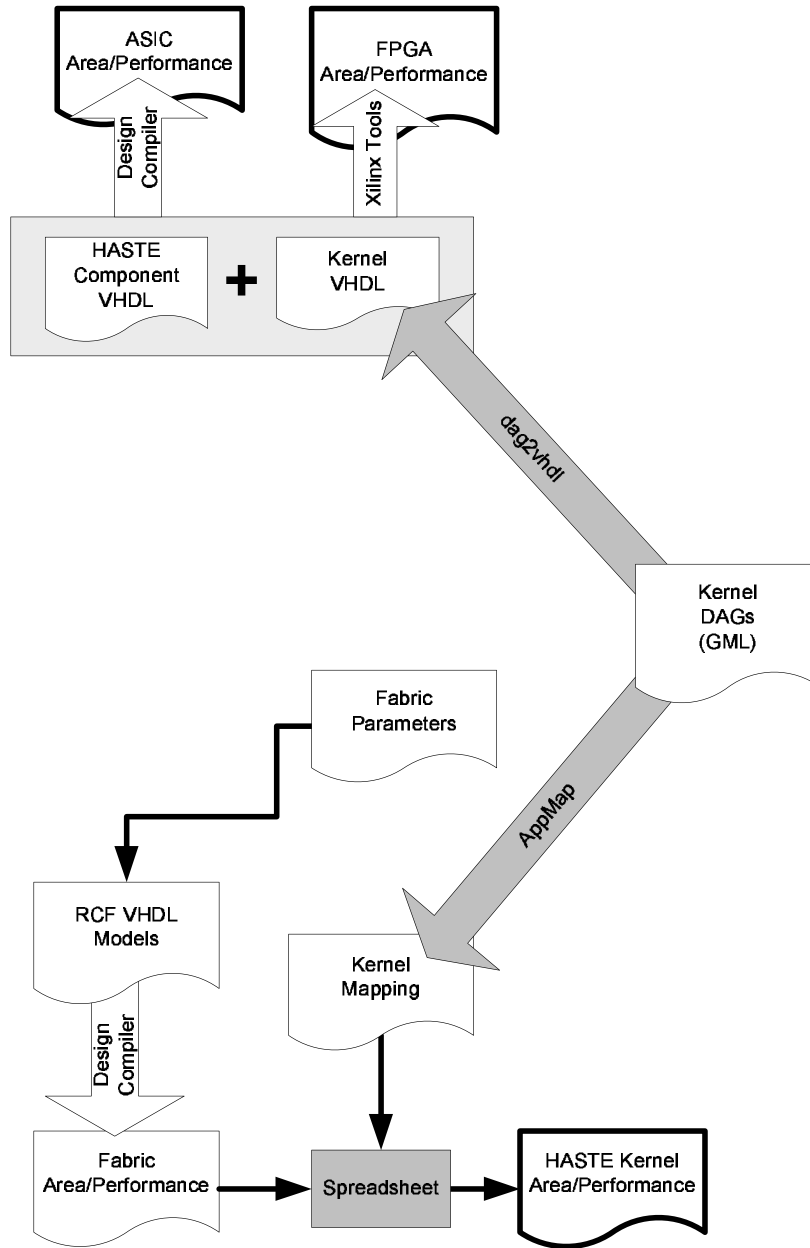


Figure 11.1: Hardware Implementation Tool Flow

was not clear initially just how much slower and larger the programmable logic implementations can come to the ASIC approach. Therefore, all of the benchmark kernels were implemented in a commercial standard-cell library for a 90-nanometer CMOS process. The remainder of this section will detail the implementation process and the results obtained.

As with all of the implementations, it was assumed that memory access is handled by memory access hardware on the same die as the application kernel hardware. This makes sense in that the target environment for HASTE is a heterogeneous SOC, with integrated memory and I/O, as well as other computing resources which may serve as sources or targets of I/O streams. Therefore, the ASIC implementations do not consider I/O cells or pins, only logic and register cells. The kernels are assumed to be rectangular regions of an SOC die, with fast, global interconnect to the remainder of the chip. Since the kernels are relatively small, a single synchronous clock can be used for the kernel logic and interface to the rest of the chip. This does not preclude the use of other clock domains on the die or the use of asynchronous interconnect between the kernel and other parts of the systems, but these details are outside the scope of this research.

11.1.1 Procedure

Many different synthesis runs were performed for each design. First, a minimal area design (minAPipe) using the pipelined library was produced by setting no timing constraints and setting an area constraint of 0. Next, a minimal clock period design (MinClk) was by reducing the clock period until timing constraints could no longer be met, using no area constraints. Next, minimal area designs for pipelined designs with clock periods of 1ns, 2 ns, 3 ns, 4ns, and 5ns were produced by setting minimum clock period constraints and using no area constraints; the

Table 11.1: ASIC Synthesis Run Types

Designation	Area Constraint	Delay Constraint
MinAPipe	0	None
MinClk	None	Min, $\approx$ 1 ns
MinA2	None	2 ns
MinA3	None	3 ns
MinA4	None	4 ns
MinA5	None	5 ns
MinA_NP	0	None

synthesis tool first optimizes for delay and then for area. Manually setting maximum areas and recompiling did not seem to give better results than those obtained automatically by the synthesis tool. This same range of clock periods was used for the HASTE implementations as well and represent a reasonable design space to explore trade-offs of speed and area. A minimal area design was produced using the non-pipelined library; the area of this design was compared to the combinational area of the minimal area pipelined design to determine if the pipelining caused significant loss of efficiency in the design by not allowing for logic to be combined and optimized across pipeline registers. The different runs are listed in Table 11.1.

11.1.2 Results

The ASIC implementations results for each of the benchmarks are shown in Table 11.2. The table shows the results for each run for each benchmark. For each run, the minimum clock period and associated frequency are shown. Next, the area devoted to combinatorial logic is shown, followed by the area devoted to registers, and then the total area. No clock period and therefore no clock frequency is shown for the minimal area runs; this wasn't needed or relevant for the non-pipelined versions, and the results obtained for the minimal area pipelined versions were incomplete due to the lack of timing constraints and thus no relevant results were reported by the synthesis tool.

The results are fairly predictable; the minimum delay version is typically the largest, and there is a generally clear trade-off between area and delay among the different runs. It is interesting to note that the register area is nearly the same for all runs for each benchmark, as would be expected, since the kernel design determines the pipelining and all performance differences are determined solely by the combinational logic. Also, the logic area for the unpipelined run is very near the logic area for the minimum area pipelined version. This indicates that their was little or no logic optimization that could have been done had the logic not been divided with pipeline registers. Some of the smaller benchmarks saw little or no difference between runs. Due to their small size, there were few possible implementations for synthesis tools to target, resulting in the small amount of variation.

Table 11.2: Benchmark Synthesis Results - ASIC

Benchmark	Run	Clock Period	Freq (MHz)	Logic Area	Register Area	Total Area
l2alaw	MinClk	0.6ns	1667	3313	13963	17276
l2alaw	MinA1	1.0 ns	1000	2357	13983	16340
l2alaw	MinA2	2.0 ns	500	2212	13983	16195
l2alaw	MinA3	3.0ns	333	1996	13983	15979
l2alaw	MinA4	4.0ns	250	1846	13983	15829
l2alaw	MinA5	5.0ns	200	1837	13983	15820
l2alaw	MinAPipe	N/A	N/A	2223	13593	15816
l2alaw	MinA_NP	N/A	N/A	2531	0	2531
dct1	MinClk	0.8ns	1250	134844	114926	249770
dct1	MinA1	1.0 ns	1000	107477	114926	222403
dct1	MinA2	2.0 ns	503	97695	114927	212622
dct1	MinA3	3.0ns	333	67568	114927	182495
dct1	MinA4	4.0ns	250	49882	114927	164809
dct1	MinA5	5.0ns	222	43977	114926	158904
dct1	MinAPipe	N/A	N/A	40154	114926	249770
dct1	MinA_NP	N/A	N/A	40178	0	40178
fir8cpx	MinClk	0.8ns	1250	217411	165009	382420
fir8cpx	MinA1	1.0 ns	1000	212778	16035	377813
fir8cpx	MinA2	2.0 ns	500	158479	164970	323449
fir8cpx	MinA3	3.0ns	333	98078	164970	263048
fir8cpx	MinA4	4.0ns	250	73847	164970	238817
fir8cpx	MinA5	5.0ns	225	68279	164970	233249
fir8cpx	MinAPipe	N/A	N/A	65021	166478	231499
fir8cpx	MinA_NP	N/A	N/A	74235	0	74235
rgb2ycc	MinClk	0.6ns	1250	42083	32433	74516
rgb2ycc	MinA1	1.0 ns	1010	31588	32435	64023
rgb2ycc	MinA2	2.0 ns	500	24109	32440	56549
rgb2ycc	MinA3	3.0ns	333	15274	32433	47707
rgb2ycc	MinA4	4.0ns	251	13712	32433	46145
rgb2ycc	MinA5	5.0ns	225	13475	32433	45908
rgb2ycc	MinAPipe	N/A	N/A	13398	32433	45831
rgb2ycc	MinA_NP	N/A	N/A	13550	0	13550
dec_cor	MinClk	0.7ns	1430	11190	7770	18960
dec_cor	MinA1	1.0 ns	1000	7855	7622	15617
dec_cor	MinA2	2.0ns	500	5314	7762	13076
dec_cor	MinA3	3.0ns	333	3927	7762	11689
dec_cor	MinA4	4.0ns	250	3685	7762	11447
dec_cor	MinA5	5.0ns	200	3645	7762	11407
dec_cor	MinAPipe	N/A	N/A	3644	7762	11406
dec_cor	MinA_NP	N/A	N/A	3658	0	3658

Benchmark	Run	Clock Period	Freq (MHz)	Logic Area	Register Area	Total Area
img_prew	MinClk	0.7ns	1429	9381	5680	15061
img_prew	MinA1	1.0 ns	1010	5660	5630	11290
img_prew	MinA2	2.0 ns	508	3176	5626	8802
img_prew	MinA3	3.0ns	382	3136	5626	8762
img_prew	MinA4	4.0ns	277	3164	5626	8790
img_prew	MinA5	5.0ns	277	3164	5626	8790
img_prew	MinAPipe	N/A	N/A	3285	5451	8736
img_prew	MinA_NP	N/A	N/A	3164	0	3164
img_med	MinClk	0.6ns	1695	2589	34105	36694
img_med	MinA1	1.0 ns	1000	3570	33453	37023
img_med	MinA2	2.0 ns	500	2106	33453	35559
img_med	MinA3	3.0ns	431	2106	33453	35559
img_med	MinA4	4.0ns	431	2106	33453	35559
img_med	MinA5	5.0ns	431	2523	33435	35976
img_med	MinAPipe	N/A	N/A	4218	31977	36195
img_med	MinA_NP	N/A	N/A	2110	0	2110
img_hp	MinClk	0.7ns	1429	11598	12484	24082
img_hp	MinA1	1.0 ns	1000	8279	12471	20750
img_hp	MinA2	2.0 ns	510	6506	12450	17085
img_hp	MinA3	3.0ns	333	4625	12460	17085
img_hp	MinA4	4.0ns	251	3758	12459	16217
img_hp	MinA5	5.0ns	223	3472	12460	15932
img_hp	MinAPipe	N/A	N/A	3611	12228	15839
img_hp	MinA_NP	N/A	N/A	3407	0	3407
img_thresh	MinClk	0.46ns	2174	1840	1120	2960
img_thresh	MinA1	1.0 ns	2174	1840	1120	2960
img_thresh	MinA2	2.0 ns	2174	1840	1120	2960
img_thresh	MinA3	3.0ns	2174	1840	1120	2960
img_thresh	MinA4	4.0ns	2174	1840	1120	2960
img_thresh	MinA5	5.0ns	2174	3040	1120	4160
img_thresh	MinAPipe	N/A	N/A	1840	1120	2960
img_thresh	MinA_NP	N/A	N/A	1840	0	1840
img_erode	MinClk	0.46ns	2174	340	3923	4263
img_erode	MinA1	1.0 ns	2174	340	3923	4263
img_erode	MinA2	2.0 ns	2174	340	3923	4263
img_erode	MinA3	3.0ns	2174	340	3923	4263
img_erode	MinA4	4.0ns	2174	340	3923	4263
img_erode	MinA5	5.0ns	2174	340	3923	4263
img_erode	MinAPipe	N/A	2174	340	3923	4263
img_erode	MinA_NP	N/A	N/A	340	0	340
img_out	MinClk	0.6ns	1694	559	2614	3173
img_out	MinA1	1.0 ns	1000	470	2614	3084
img_out	MinA2	2.0 ns	538	470	2614	3084
img_out	MinA3	3.0ns	518	470	2614	3084
img_out	MinA4	4.0ns	512	470	2614	3084

Benchmark	Run	Clock Period	Freq (MHz)	Logic Area	Register Area	Total Area
img_out	MinA5	5.0ns	541	1081	2614	3695
img_out	MinAPipe	N/A	N/A	1061	2571	3631
img_out	MinA_NP	N/A	N/A	470	0	470
idea	MinClk	0.9ns	1111	268986	239818	508804
idea	MinA1	1.0 ns	1000	217451	239818	457269
idea	MinA2	2.0 ns	500	199649	239818	439467
idea	MinA3	3.0ns	333	142227	239820	382047
idea	MinA4	4.0ns	250	107595	239818	347413
idea	MinA5	5.0ns	208	91174	239818	330992
idea	MinAPipe	N/A	N/A	82581	239819	322400
idea	MinA_NP	N/A	N/A	84461	0	84461

Overall, these are fast and relatively small implementations of the kernels. The designs are dominated by register area and one could argue that they are over-pipelined, but these are still small and fast designs, particularly for a design flow that is fully automated from assembly language to silicon. Recent research shows that a full custom or semi-custom implementations of datapath design can produced that are as much a smaller than fully-automated standard cell implementations and commensurately faster [50]. However, since fully automated design flows are being used, with minimal manual effort needed to implement the design, the performance and area advantage of custom designs is not particularly relevant; any of the implementations could be greatly improved with extensive designer attention. As discussed in Chapter 1, though, it is increasingly rare that this level of design effort makes economic sense for anything other than very high volume designs.

11.2 FPGA Implementation

Another method that could be used to implement the benchmark kernels would be to use an FPGA. FPGA are becoming increasingly popular for computationally intensive kernels, since they have become relatively standard, readily available, programmable parts that provide performance nearer that of an ASIC than can be obtained from other programmable solutions such as general purpose CPUs or DSPs. FPGA fabrics have mostly been implemented in specific FPGA devices as a single packaged die with I/O buffers, and increasingly, other specialized hardware such as

clock generators and embedded memories. More recently, FPGA fabrics have been made available as hard IP that can be used as part of an SOC. The usage being considered is this latter case, since this fits with the intended application domain for HASTE.

FPGA manufacturers make public few details about the actual design of their chips or IP blocks. Even basic information such as the number of transistors and die size for a particular device are difficult to obtain. After much searching, a publicly available summary of a report on a particular Xilinx FPGA was found. This report summary [54] was prepared by a company, Chipworks, that publishes information they obtain by reverse-engineering commercial IC devices by use of electron microphotography and other techniques. While access to the full report was prohibitively expensive, the freely available summary did detail the die area of the Xilinx Spartan 3 XC3S200 device, specifically 20.3 mm. Since the number of FPGA resources a particular application uses can be determined from the FPGA synthesis tools, knowing the total die area allows estimation of the die area used by the application. Since this device is fabricated in a 90-nanometer process, according to information provided by the manufacturer, it is then possible to fairly compare die area of implementations in the FPGA with die area of implementations in the semi-custom ASIC flow and the HASTE implementations.

Given the limited amount of information available about the FPGA, however, there is some uncertainty in the area estimations for the FPGA implementations. In particular, not all of the die area is used by logic that can be used for implementing applications. Significant portions of the die area are consumed by I/O drivers and other special purpose logic. An examination of die photographs for other FPGAs lead to the somewhat arbitrary assumption that 80% of the die are is devoted to the user-programmable logic; this estimate is likely too low, so the area required for the FPGA implementations will look better (better meaning smaller in this context) than is likely the case in actuality.

11.2.1 Procedure

The same kernel VHDL that was generated by `dag2vhdl` and the same component library that was used for the ASIC implementation was used for the FPGA implementation. The Xilinx ISE Alliance design tools were used, in particular the Xilinx XST synthesis tool. XST was used to synthesize the kernel design and the Xilinx supplied tools were used to place and route the

design for the Xilinx XC3S200-5FT256, the exact same device referenced in the Chipworks report summary.

The application was synthesized at the highest effort with all relevant optimizations enabled and maximum speed as the primary objective. After the first synthesis, a clock period constraint was set and repeatedly lowered until the timing constraint could not be met. After the application was placed and routed, the timing analyzer tool was used to verify that the critical path for the design did not include I/O delays, since the assumption is that the FPGA fabric is being used as part of an SOC. Then the design was synthesized again with minimal area as the objective. Configuration file sizes were determined by scaling the whole chip configuration file size by the proportion of chip area used by the design. Table 11.3 shows the results obtained for the FPGA implementations of the benchmarks. In some cases the design used more resources than are actually available on the device. The area was still determined using the same method as was used for the smaller designs in these cases, by just scaling the area linearly with the number of slices in the design. As one would expect, the FPGA implementations are much larger and much slower than the ASIC implementations. Note also that the configuration file sizes are much larger than the kernel code sizes shown in Chapter 8, the very largest of which was less than 4 KB.

Table 11.3: FPGA Implementation Results

Benchmark	Run	Clock Freq (MHz)	Slices	% of Total	Die Area	Config File Size (KBytes)
l2alaw	Max F.	251	236	12%	2.00 mm^2	10.1
l2alaw	Min A.	198	227	12%	1.92 mm^2	9.7
l2alaw	NP	48	73	4%	0.62 mm^2	3.1
dct1	Max F.	243	2441	127%	20.65 mm^2	104
dct1	Min A.	83	2133	111%	18.04 mm^2	90.9
dct1	NP	33	1178	61%	9.96 mm^2	50.2
fir8cpx	Max F.	226	3913	204%	33.1 mm^2	166.8
fir8cpx	Min A.	57	3661	191%	30.97 mm^2	156.1
fir8cpx	NP	29	1901	99%	16.08 mm^2	81.0
rgb2ycc	Max F.	232	1104	58%	9.34 mm^2	47.1
rgb2ycc	Min A.	136	959	50%	8.11 mm^2	40.9
rgb2ycc	NP	41	525	27%	4.44 mm^2	22.4
dec_cor	Max F.	239	212	11%	1.79 mm^2	9.0
dec_cor	Min A.	136	184	10%	1.56 mm^2	7.9
dec_cor	NP	38	140	7%	1.18 mm^2	5.9
img_prew	Max F.	243	247	13%	2.09mm^2	10.5
img_prew	Min A.	140	192	10%	1.62 mm^2	8.2
img_prew	NP	41	158	8%	1.34 mm^2	6.8
img_med	Max F.	233	671	25%	5.68 mm^2	28.6
img_med	Min A.	134	580	30%	4.91 mm^2	24.7
img_med	NP	33	329	8%	2.78 mm^2	14.0
img_hp	Max F.	242	241	13%	2.04 mm^2	10.3
img_hp	Min A.	141	189	8%	1.60 mm^2	8.1
img_hp	NP	40	158	8%	1.34 mm^2	6.8
img_thresh	Max F.	251	171	9%	0.45 mm^2	2.3
img_thresh	Min A.	142	150	8%	0.32 mm^2	1.6
img_thresh	NP	36	112	6%	0.1 mm^2	0.5
img_erode	Max F.	263	84	4%	0.71 mm^2	3.6
img_erode	Min A.	140	72	4%	0.61 mm^2	3.1
img_erode	NP	36	61	3%	0.52 mm^2	2.6
img_out	Max F.	253	110	6%	0.93 mm^2	4.7
img_out	Min A.	139	97	5%	0.82 mm^2	4.1
img_out	NP	34	48	3%	0.41 mm^2	2.1
idea	Max F.	209	8448	440%	71.46 mm^2	360.1
idea	Min A.	54	7503	391%	63.46 mm^2	319.8
idea	NP	31	4812	251%	40.7 mm^2	205.1

11.3 HASTE Implementations

Given that many different mappings were done for each benchmark, and each mapping could be implemented in a fabric with tiles of a different speed grade, a large amount of data was produced. As discussed in the previous chapter, the half-width multiplier ALU was used for all fabrics. The most interesting data is that shown in Table 11.4. For each benchmark, the "best" fabric for each ISA was found and compared with the "best" ASIC and FPGA implementations. Total die area and clock speed are shown for all implementations. In addition, the fabric dimensions and parameters, as well as the tile area, are shown for each of the HASTE implementations. In order to find the "best" implementation, the Throughput per Unit Area (TUA) for each implementation was computed and normalized to the implementation with the lowest value in the same table. Since all of the implementations compute the same number of results every clock cycle, this metric can simply be computed as $TUA = \frac{clock\ freq.}{total\ area}$. As all of the values are normalized to the slowest implementation, the Normalized TUA (NTUA) values are unitless. The higher the NTUA value, the more results that can be computed by that implementation for that benchmark per unit area of silicon die area, so highest NTUA corresponds to "best" in these rsults. The absolute throughput per unit area was computed for every implementation, and the best for each was found for each benchmark. Without exception, the best in this regard was the same implementation found to be the fastest.

The results do show clearly that the limited interconnect fabrics (RC = 3) have the best characteristics in terms of both area and speed. The area and delay costs of longer interconnect clearly outweigh the advantages in this process technology, and as interconnect delays grow larger relative to area in future technologies, the advantages of this local interconnect style should only increase.

Table 11.4: Comparison of Best Implementations for Each Benchmark

l2alaw	Fabric Depth	Fabric Width	Fabric Type: NR/WC/RC	Tile Area, $\mu m^2 \times 10^3$	Total Area, $\mu m^2 \times 10^3$	Clock Freq., MHz	Normalized Throughput / Unit Area
ASIC	-	-	-	-	17.3	1667	1241
FPGA	-	-	-	-	2010.0	251	1.61
QISA	34	9	1 / 1 / 3	31.9	9748	758	**1.0**
RISA	27	2	3 / 1 / 3	22.1	1193	642	4.61
RRISA	27	2	3 / 3 / 3	23.1	1248	619	4.32

dct1	Fabric Depth	Fabric Width	Fabric Type: NR/WC/RC	Tile Area, $\mu m^2 \times 10^3$	Total Area, $\mu m^2 \times 10^3$	Clock Freq., MHz	Normalized Throughput / Unit Area
ASIC	-	-	-	-	249.8	1250	425.3
FPGA	-	-	-	-	20650	243	**1.0**
QISA	26	37	1 / 1 / 3	27.3	26225	758	2.46
RISA	60	3	13 /1 / 3	55.2	9943	628	5.37
RRISA	35	6	7 / 3 / 3	31.9	6706	618	5.82

fir8cpx	Fabric Depth	Fabric Width	Fabric Type: NR/WC/RC	Tile Area, $\mu m^2 \times 10^3$	Total Area, $\mu m^2 \times 10^3$	Clock Freq., MHz	Normalized Throughput / Unit Area
ASIC	-	-	-	-	382.4	1250	478.7
FPGA	-	-	-	-	33100.0	226	**1.0**
QISA	118	20	1 / 1 / 3	27.3	64336	758	1.73
RISA	95	3	12 /1 / 3	36.5	10404	630	6.81
RRISA	48	6	7 / 3 / 3	31.9	9197	618	7.31

rgb2ycc	Fabric Depth	Fabric Width	Fabric Type: NR/WC/RC	Tile Area, $\mu m^2 \times 10^3$	Total Area, $\mu m^2 \times 10^3$	Clock Freq., MHz	Normalized Throughput / Unit Area
ASIC	-	-	-	-	74.5	1250	675.4
FPGA	-	-	-	-	9340.0	232	**1.0**
QISA	26	16	1 / 1 / 3	27.2	11341	758	2.69
RISA	29	3	6 /1 / 3	26.3	2288	637	7.89
RRISA	18	6	2 / 3 / 3	20.9	2260	620	7.23

dec_cor	Fabric Depth	Fabric Width	Fabric Type: NR/WC/RC	Tile Area, $\mu m^2 \times 10^3$	Total Area, $\mu m^2 \times 10^3$	Clock Freq., MHz	Normalized Throughput / Unit Area
ASIC	-	-	-	-	18.9	1250	491.7
FPGA	-	-	-	-	1790.0	240	**1.0**
QISA	8	16	1 / 1 / 3	27.3	3490	758	1.62
RISA	27	2	2 / 1 / 3	20.2	1093	660	2.91
RRISA	27	2	2 / 3 / 3	20.9	1130	620	2.68

Table 11.4: continued

prew	Fabric Depth	Fabric Width	Fabric Type: NR/WC/RC	Tile Area, $\mu m^2 \times 10^3$	Total Area, $\mu m^2 \times 10^3$	Clock Freq., MHz	Normalized Throughput / Unit Area
ASIC	-	-	-	-	15.1	1429	815.8
FPGA	-	-	-	-	2090	243	**1.0**
QISA	17	7	1 / 1 / 3	27.3	3244	758	2.01
RISA	17	2	4 / 1 / 3	23.7	807	640	4.65
RRISA	17	2	4 / 3 / 3	26.3	893	619	4.20

img_med	Fabric Depth	Fabric Width	Fabric Type: NR/WC/RC	Tile Area, $\mu m^2 \times 10^3$	Total Area, $\mu m^2 \times 10^3$	Clock Freq., MHz	Normalized Throughput / Unit Area
ASIC	-	-	-	-	36.7	1695	2552
FPGA	-	-	-	-	5680	233	2.27
QISA	64	24	1 / 1 / 3	27.3	41873	758	**1.0**
RISA	55	2	12 / 1 / 3	36.5	4016	630	6.65
RRISA	33	4	8 / 3 / 3	35.5	4687	616	5.54

img_hp	Fabric Depth	Fabric Width	Fabric Type: NR/WC/RC	Tile Area, $\mu m^2 \times 10^3$	Total Area, $\mu m^2 \times 10^3$	Clock Freq., MHz	Normalized Throughput / Unit Area
ASIC	-	-	-	-	24.1	1429	502.1
FPGA	-	-	-	-	2040	241	**1.0**
QISA	21	3	1 / 1 / 3	27.3	1717	758	3.74
RISA	21	2	2 / 1 / 3	20.2	850	660	4.25
RRISA	21	2	2 / 3 / 3	20.9	879	620	3.91

thresh	Fabric Depth	Fabric Width	Fabric Type: NR/WC/RC	Tile Area, $\mu m^2 \times 10^3$	Total Area, $\mu m^2 \times 10^3$	Clock Freq., MHz	Normalized Throughput / Unit Area
ASIC	-	-	-	-	3.2	1429	800.9
FPGA	-	-	-	-	450	251	**1.0**
QISA	17	2	1 / 1 / 3	27.3	927	758	1.47
RISA	17	2	2 / 1 / 3	20.2	688	660	1.11
RRISA	17	2	2 / 3 / 3	20.9	620	620	1.02

erode	Fabric Depth	Fabric Width	Fabric Type: NR/WC/RC	Tile Area, $\mu m^2 \times 10^3$	Total Area, $\mu m^2 \times 10^3$	Clock Freq., MHz	Normalized Throughput / Unit Area
ASIC	-	-	-	-	4.3	2173	1376.7
FPGA	-	-	-	-	710	263	**1.0**
QISA	9	2	1 / 1 / 3	27.3	491	758	4.17
RISA	9	2	2 / 1 / 3	20.2	364	660	3.16
RRISA	9	2	2 / 3 / 3	20.9	376	620	2.91

Table 11.4: continued

img_out	Fabric Depth	Fabric Width	Fabric Type: NR/WC/RC	Tile Area, $\mu m^2 \times 10^3$	Total Area, $\mu m^2 \times 10^3$	Clock Freq., MHz	Normalized Throughput / Unit Area
ASIC	-	-	-	-	3.2	1694	1963.5
FPGA	-	-	-	-	930	253	**1.0**
QISA	7	4	1 / 1 / 3	27.3	763	758	3.65
RISA	10	2	2 / 1 / 3	20.2	405	660	3.88
RRISA	10	2	2 / 3 / 3	20.9	418	620	3.56
idea	Fabric Depth	Fabric Width	Fabric Type: NR/WC/RC	Tile Area, $\mu m^2 \times 10^3$	Total Area, $\mu m^2 \times 10^3$	Clock Freq., MHz	Normalized Throughput / Unit Area
ASIC	-	-	-	-	508.8	1111	746.7
FPGA	-	-	-	-	71460	209	**1.0**
QISA	498	5	1 / 1 / 3	27.3	67880	758	3.82
RISA	265	2	12 / 1 / 3	36.5	19347	630	8.55
RRISA	265	2	12 / 3 / 3	71.8	38063	611	5.49

11.4 Comparisons

11.4.1 Speed

Figure 11.2 shows the clock frequency for the fastest implementations for each benchmark by technology. The ASIC implementation is always the fastest and the FPGA implementation is always the slowest. Of the HASTE implementations, the QISA implementation is always the fastest, followed by the RISA, and RRISA implementations. This shows, not surprisingly, that the simplest HASTE fabric, the static register fabric used for the QISA implementation, is the fastest, followed by the next simplest, the asymmetric pass register fabric used for the RISA implementations. The slowest fabric is also the most complicated, the symmetric pass register fabrics used for the RRISA implementations. The QISA implementation is consistently more than three times faster than the FPGA implementations and the RRISA and RISA implementations are consistently more than 2.5 times faster than the FPGA implementations. The ASIC implementation frequencies vary widely, but are on average about twice as fast as the QISA implementations.

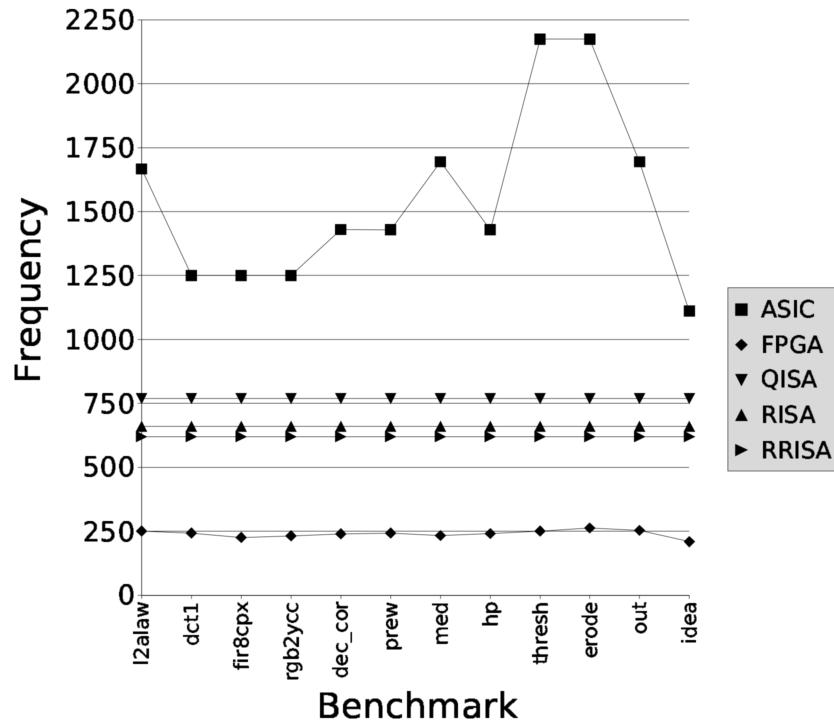


Figure 11.2: Frequency for Fastest Benchmark Implementations, by Technology.

11.4.2 Area

Figure 11.3 shows the area of the smallest implementations for each benchmark by technology. Note that these results are shown on a log scale due to wide range of values. As one would expect, the ASIC implementations are consistently 2 orders of magnitude smaller than the other implementations. The FPGA implementations are typically the largest implementations, although the QISA implementations are several times larger in two instances and both the RISA and RRISA are slightly larger for one benchmark. The QISA implementations are larger than either register ISA, except for the applications that were already inherently level-planar. It should be noted that the benchmark for which the HASTE ISA implementations are all larger than the FPGA implementation is img-thresh, which is one of the narrowest benchmarks in terms of bitwidth, in that it only deals with 8-bit or smaller values, so the 32-bit HASTE datapath is particularly wasteful of resources

11.4.3 NTUA

Figure 11.4 shows the normalized throughput per unit area for each benchmark, by technology. Note that the FPGA implementations are alway the worst by this metric, so all of the other implementations are normalized to the FPGA implementation, which are given a value of one. Again, the results are shown on a log scale. The ASIC implementations are around three orders of magnitude better than the FPGA implementations by this metric. The RRISA and RISA implementations range from 2 to 13 times better than the FPGA versions, and the QISA implementations ranges from 1 to 9 times better. The only QISA implementations that were better than either register ISA in terms of NTUA were the four small benchmarks mentioned in Chapter 8 that were inherently level-planar and therefore did not require many additional instructions for the queue implementations. The harmonic means of the NTUAs is 707.9 for the ASIC implementations, 3.42 for the QISA implementations, 5.48 for the RISA implementations, and 5.24 times better for the RRISA implementations (harmonic means are used here because rates are being compared).

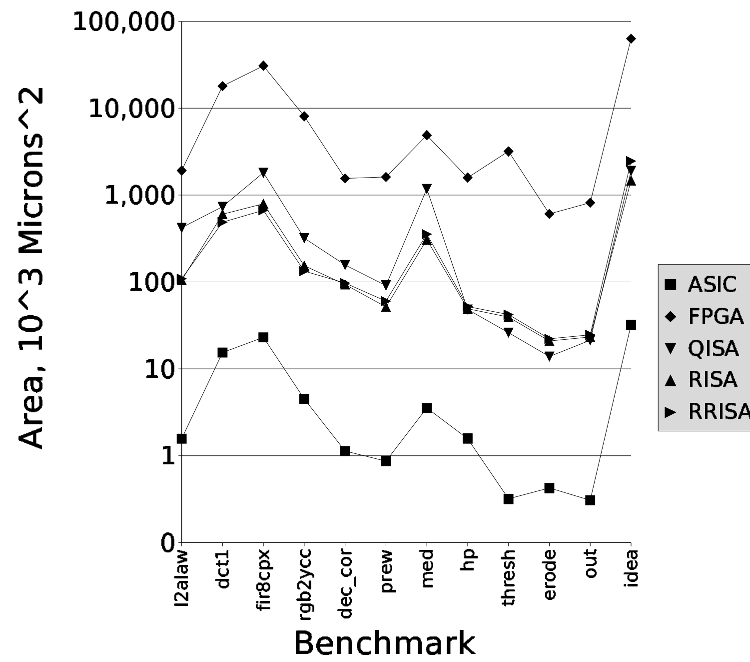


Figure 11.3: Area for Smallest Benchmark Implementations, by Technology.

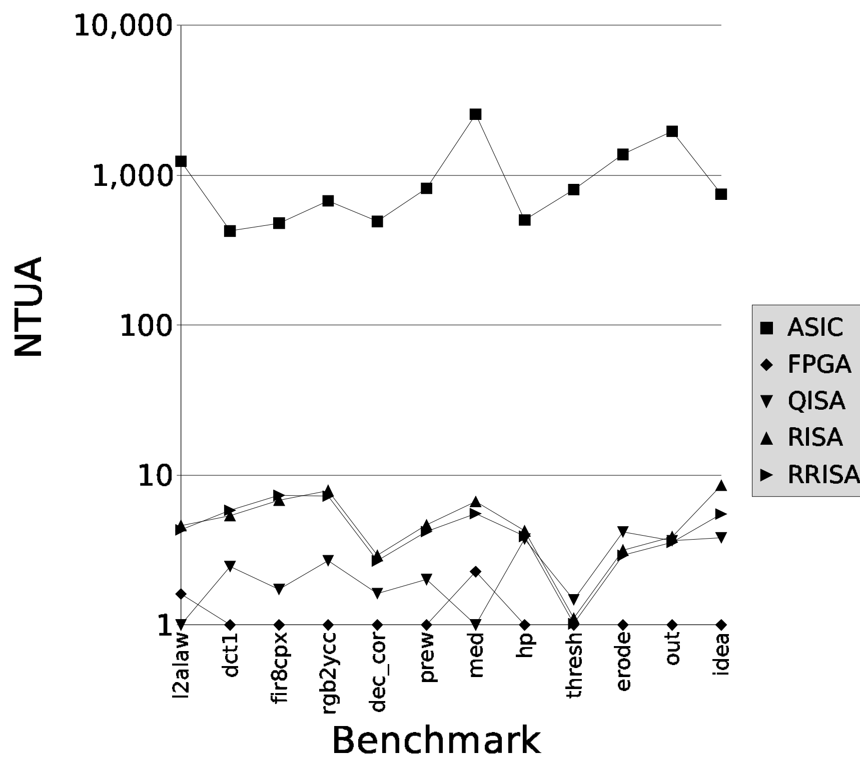


Figure 11.4: NTUA for Fastest Benchmark Implementations, by Technology.

11.4.4 Best Fabric

The previous section looked at the best implementation for each benchmark in each technology, which required using many different HASTE fabrics, since some fabrics were better for some benchmarks and other fabrics better for different benchmarks. A real HASTE system would have a single fabric type that would be used for all benchmarks. In order to determine this, all fabrics were evaluated to find the single best fabric for all the benchmarks. The harmonic mean of TUA for all benchmarks was used as a metric and the single best fabric for each ISA was found and are listed below:

ISA	Best Fabric
QISA	NR1_WC1_RC3
RISA	NR4_WC1_RC3
RRISA	NR4_WC3_RC3

It is interesting to note that all of the best fabrics have the minimum connectivity possible, with RC = 3. Also, a relatively small register file size of 4 is the best for both RISA and RRISA. The best overall QISA fabric also happened to be the best fabric for each benchmark individually. This was not true for RISA or RRISA; the best fabric was not the best for any individual benchmark (note that a NR=4 and RC=4 fabric was best for img_prew benchmark for both RISA and RRISA, but it not the fastest version of this fabric). Of course, using a single fabric for all benchmarks will have some cost in terms of performance for the benchmarks. Figure 11.5 shows the percentage of the best NTUA seen using the best fabric for the benchmark that was achieved for each benchmark using the overall best fabric. Note that QISA is not included, since the percentage would be 100% in each case. RISA showed a somewhat bigger cost than RRISA for choosing a single fabric, with a penalty of greater than 50% seen for three benchmarks, versus only 1 for RRISA. For both ISAs, the penalty was less than 20% for a majority of the benchmarks. Figure 11.6 is a revision of Figure 11.4, using the single best fabric overall for the RISA and RRISA implementations. There is little difference between the two figures, although the advantage of RISA and RRISA as compared to QISA and FPGA are reduced.

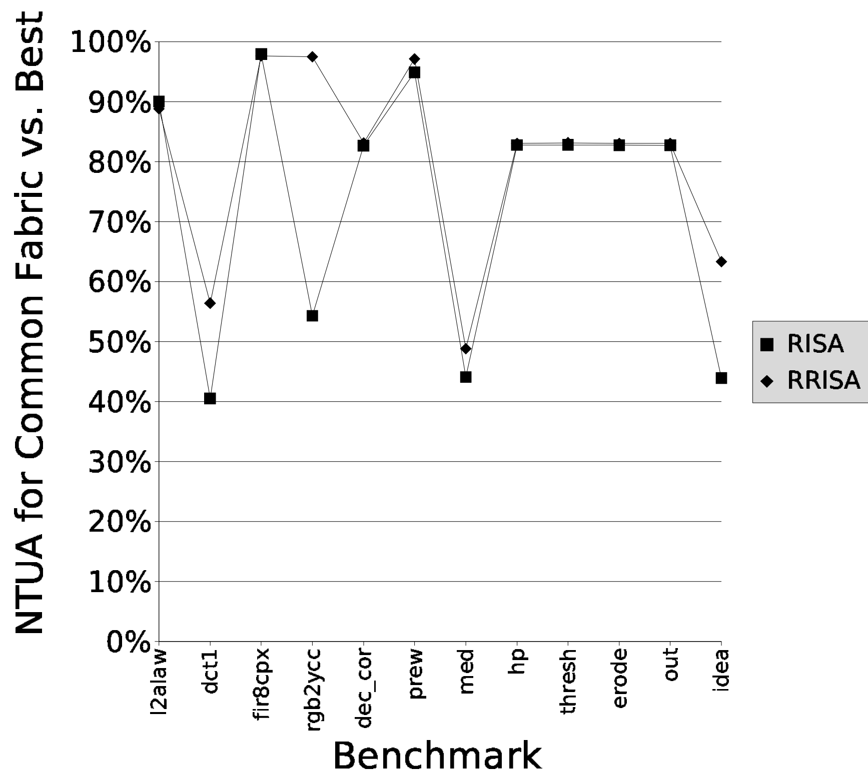


Figure 11.5: NTUA Loss for Common Fabric Compared to Best Fabric

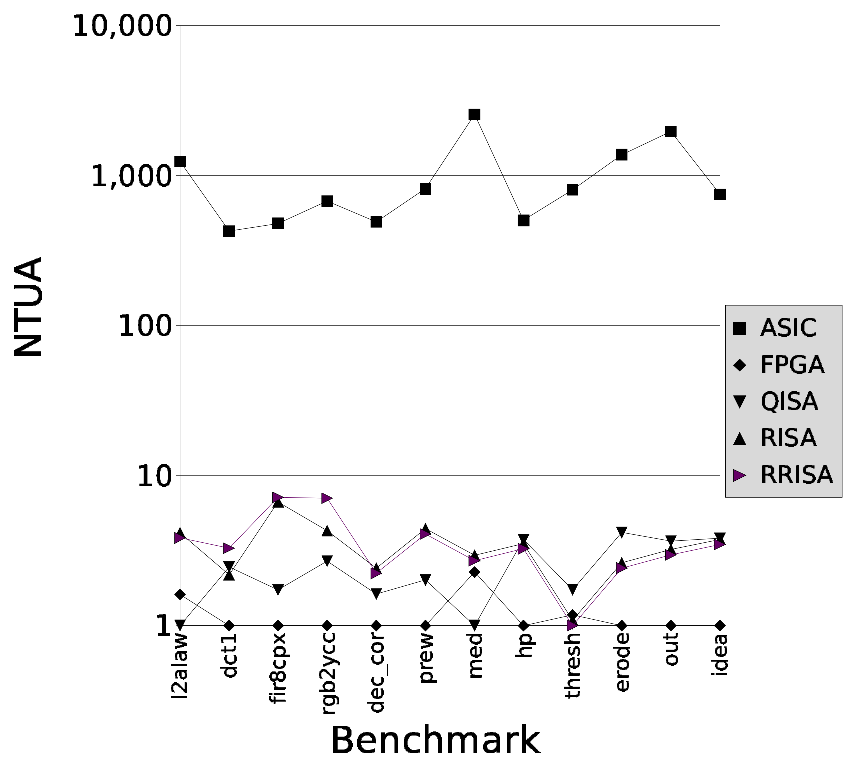


Figure 11.6: NTUA for Best Single Fabric Benchmark Implementations, by Technology.

11.5 Observations

The results in this chapter show that HASTE fabrics that are much better than FPGA fabrics in terms of speed and NTUA, as well as area for all but two benchmarks. The kernel assembly files sizes for HASTE are also much smaller than the configuration files for the kernel implementations on FPGAs, as seen by comparing Table 11.3 with Tables 8.2, 8.3, and 8.4. The performance of a real HASTE system would likely be even better than the results shown here, since it would be implemented as much more efficient custom silicon, as opposed to the standard cells used here. As expected, HASTE implementations were larger and slower than ASIC implementations. They were typically two to three times slower than ASIC implementations and as much as a hundred times larger. Once again, custom design of a HASTE fabric would greatly improve area and performance. While this is also true of ASIC implementations, such custom design has to be done for every different kernel, whereas once a custom HASTE fabric design is implemented, the performance gains will apply to every kernel. This is the kind of design reuse that was discussed in Chapter 1.

The differences between the three HASTE ISAs and RCF types were fairly small as compared to the differences between HASTE and FPGAs or ASICs. Mappings of QISA kernels to static register fabrics were slower than RISA or RRISA mappings to pass register fabrics. They were also larger, with the exception of some very small kernels. RISA and RRISA mappings were very similar, with a slight edge for the RISA mappings, due apparently to the fact that the asymmetric pass register fabric is simpler than the symmetric pass register fabric.

Chapter 12

Conclusions and Future Work

12.1 Conclusions

In the HASTE ISA comparisons in Chapter 8, it was shown that QISA fared poorly in all of the ISA metrics. RISA and RRISA were very close to each other in all metrics, with a slight advantage for the RRISA in terms of the hardware latency and the hardware utilization. In the area and performance evaluations in Chapter 11, the HASTE ISAs compared well to the FPGA implementations, with clock speeds averaging more than twice as fast, and and as much half the size. Given that the comparison is between a very highly optimized, full custom FPGA design and an automated standard cell implementation of the HASTE fabrics, this shows a significant advantage for HASTE, which would be even greater if the HASTE fabrics had the extensive custom design and optimization of the FPGA. The register ISAs were slower than the queue ISA, but were also smaller and had better normalized throughput per unit area values. The RRISA and RISA implementations were very similar, although the RRISA implementations were slightly slower and larger on average.

If absolute clock speed for the CTE is the most important consideration, than the QISA implementations on the static register fabric may be the best. However, the code size and code length disadvantages of the QISA kernels, as well as the overall inefficiencies of having to make the entire application queue-legal and the need to use a special queue-based SPU, seem to clearly outweigh the small performance advantages. The only QISA implementations that were better

than the register-based ISAs were the four very small benchmarks that were inherently level-planar. Either the RRISA or RISA implementations provide nearly as good speed, smaller area, and generally better throughput per unit area, when compared to QISA, as well as the ability to use more conventional code for the non-kernel portions of the applications, and a conventional (or near-conventional for RRISA) SPU. The clear superiority of fabrics with limited, highly local interconnects is another important result. As discussed previously, this advantage should only increase in future process technologies.

Perhaps the most important result is the demonstration that not only do HASTE architectures work correctly, but they result in spatial fabrics that are much better than FPGA fabrics in terms of both speed and area. In addition, HASTE architectures are much more easily programmable and require a single executable. The kernel assembly files sizes for HASTE are also much smaller than the configuration files for the kernel implementations on FPGAs, as seen by comparing Table 11.3 with Figure 8.7. The performance of a real HASTE system would likely be much better than the results shown here, since it would be implemented as much more efficient custom silicon, as opposed to the standard cells used here.

12.2 Future Work

While this thesis outlined the basic requirements and functionality of HASTE systems, there were several assumptions made about the fabrics that limit their suitability for real systems. In addition there are many additional architectural concepts that can be applied to the HASTE concept to improve its performance and efficiency, and to expand the range of kernels that can be implemented on HASTE architectures.

12.2.1 Depth and Width Virtualization

In this thesis, it was assumed that the size of the fabric was exactly large enough to fit each particular kennel. While this simplified experiments and analysis, it is not a valid assumption for a real programmable fabric. This problem can be handled using the concept of hardware virtualization. The PipeRench architecture demonstrated that depth virtualization of a pipelined fabric can be done practically and automatically in hardware [18]. Since all of the best RISA

and RRISA fabrics were narrow, it would be possible to just use a relatively narrow fabric with depth virtualization to allow for implementation of kernels of all sizes. This would preclude the use of the queue ISA, since it is not possible to change the width of queue DFGs, another advantage of the register-based ISAs. Virtualization of pipeline width is more complicated than depth virtualization, but there are several possible techniques that should be explored. It may also be possible to take a fixed size fabric and rearrange the interconnect to create pipelines of different widths. For instance, a ten tile by ten tile fabric might be reconfigured as a five by twenty fabric, a four by twenty-five fabric, or a two by fifty fabric. Some sort of virtualization, either of depth, width, or both, is probably needed for any practical HASTE system.

12.2.2 Feedback

None of the application kernels exhibited inter-iteration feedback. Allowing this would allow for the implementation of a wider range of applications. Feedback presents problems for a pipelined fabric, however, and may require c-slow re-timing of the fabric or other technique that may severely impact performance. Feedback may also be implemented through the memory interface for feedback across many iterations.

12.2.3 Narrow Tiles

HASTE as shown here uses 32-bit tiles. This effects the area efficiency for applications that use narrow operand bit widths. Using narrow tiles that operate on 16 or 8-bit data, for instance, and that can be combined to operate on wider values would be one way to improve area efficiency. This techniques has been implemented in several architectures, including PipeRench. The combined executable property of HASTE makes this more complicated and may require the development of new techniques.

Appendix A

Glossary

Column	A set of tiles on different levels at the same postion within each stripe.
CTE	Code Transformation Engine
DFG	Dataflow graph.
F_w	Fabric Width
F_d	Fabric Depth
GHAL	Generic HASTE Assebly Language. Form of architecture/ISA independent assembly code used in the HASTE tool flow.
HASTE	Hybrid Architectures with a Single, Transformable Executable
n_i	Number of inputs for an operation.
n_o	Number of outputs for an operation.
O_c	Set of computation operations
O_m	Set of data movement operations
O_f	Set of control flow operations
O_{sc}	Set of computation operations that can run on the SPU.
O_{sm}	Set of data movement operations that can run on the SPU.

O_{sf} Set of control flow operationst that can run on the SPU.

O_k Set of all operations that can run on the RCF.

O_{kc} Set of computation operations that can run on the RCF.

O_{km} Set of data movement operations that can run on the RCF.

QISA Queue ISA

RCF Reconfigurable Computational Fabric

RISA Register ISA

Row A set of tiles at the same level in the fabric. Equivalent to a pipeline stage. Also referred to as a stripe.

RRISA Relative Register ISA

SPU Sequential Processing Element

Stripe See *Row*

Tile Repeating element composing the RCF. Each tile consists of a register file and an ALU.

Appendix B

HASTE ISA Reference

ADD	0x40	Add	2

Queue Formats: ADD, ADD2

Register Format: ADD rout1, rin1, rin2

Description: Add word in1 to word in2. Result is out1.

Function: `out1 ← in1 + in2`

Queue
```
POP   in1, in2
PUSH  (in1 + in2)
```

Register
```
GPR[rout1] ← (GPR[in1] + GPR[in2])
```

ADDI	0x41	Add Immediate	2, I

Queue Formats: ADDI, ADDI2

Register Format: ADDI rout1, rin1, imm(16)

Description: Add word in1 to immediate value imm. Result is out1.

Function: `out1 ← in1 + imm`

Queue
```
POP   in1
PUSH  (in1 + imm)
```

Register
```
GPR[rout1] ← (GPR[rin1] + imm)
```

AND	0x4E	And	2,K
Queue Formats:	AND, AND2		
Register Format:	AND rout1, rin1, rin2		
Description:	Bitwise Boolean AND of words *in1* and *in2*. Place result in *out1*.		
Function:	`out1 ← in1 AND in2`		
Queue	`POP   in1, in2` `PUSH  (in1 AND in2)`		
Register	`R[rout1] ← (R[in1] AND R[in2])`		

ANDI	0x4F	And Immediate	2,I,K
Queue Formats:	ANDI, ANDI2		
Register Format:	ANDI rout1, rin1, imm		
Description:	Bitwise Boolean AND of word *in1* and immediate *imm*. Place result in *out1*.		
Function:	`out1 ← in1 AND imm`		
Queue	`POP   in1` `PUSH  (in1 AND imm)`		
Register	`R[rout1] ← (R[rin1] AND imm)`		

DROP	0xF6	Drop	Q,K
Queue Formats:	DROP		
Register Format:	N/A		
Description:	Discard word at front of queue.		
Function:	`˘ ← in1`		
Queue	`POP   in1`		

LIS	0xEC	Load Immediate Signed	2,I,K

Queue Formats: LIS, LIS2

Register Format: LIS imm

Description: Place sign-extended immediate *imm* in *out1*.

Function: `out1 ← imm`

Queue `PUSH sign_extended(imm)`

Register $(R[rout1])|_{LO} \leftarrow imm$
$(R[rout1])|_{HI} \leftarrow$ sign bit of imm

LIU	0xEB	Load Immediate Unsigned	2,I,K

Queue Formats: LIU, LIU2

Register Format: LIU imm

Description: Place zero-extended unsigned immediate *imm* in *out1*.

Function: `out1 ← imm`

Queue `PUSH zero_extended(imm)`

Register $(R[rout1])|_{LO} \leftarrow imm$
$(R[rout1])|_{HI} \leftarrow 0000000000000000$

LPBGN	0xDA	Loop Begin

Queue Formats: LPBGN

Register Format: LPBGN rout1, rin1

Description: Begin iterating loop number of times equal to *in1*. For register ISAs, store loop variable in *out1,* for queue ISA use architected loop register.

Function: `loop reg ← in1`

Queue `POP in1 (store in loop register)`

Register `R[rout1] ← GPR[rin1]`

LPBGNI	0xDB	Loop Begin Immediate	I

Queue Formats: LPBGNI

Register Format: LPBGNI rout1, imm

Description: Begin iterating loop number of times equal to *imm.* For register ISAs, store loop variable in *out1,* for queue ISA use architected loop register.

Function: `loop reg ← imm`

Queue `(store imm in loop register)`

Register `R[rout1] ← imm`

LPEND	0xDC	Loop End

Queue Formats: LPEND addr

Register Format: LPEND rin1, addr

Description: Decrement loop variable. Branch to *addr* if loop variable > 0. For register ISAs, loop variable is in *in1,* for queue ISA it is in architected loop register.

Function: `loop reg--; branch to addr if > 0`

Queue `loop reg--; branch to addr if > 0`

Register `R[rin1]--; branch to addr if > 0`

LUI	0xA2	Load Upper Immediate	I,K

Queue Formats: N/A

Register Format: LUI imm

Description: Place 16-bit immediate *imm* into high bits of *out1,* zeroes into lower bits.

Function: `out1 ← imm`

Queue `PUSH zero_padded(imm)`

Register $(R[rout1])|_{LO} \leftarrow 0000000000000000$
$(R[rout1])|_{HI} \leftarrow imm$

MOVE	0xE9	Move	K

Queue Formats: N/A

Register Format: MOVE rout1, rin1

Description: Place value *in1* into *out1*.

Function: `out1 ← in1`

Register `R[rout1] ← (R[rin1])`

MUL	0xF2	Signed Partial Multiply	2,K

Queue Formats: MUL, MUL2

Register Format: MUL rout1, rin1, rin2

Description: Multiply two signed words, *in1* and *in2*. Place low word of result in *out1*.

Function: `out1 ← in1 x in2`

Queue
```
POP   in1, in2
PUSH (in1 x in2)|LO
```

Register `R[rout1] ← (R[rin1] x R[rin2])|LO`

MULI	0xE1	Signed Partial Multiply by Immediate	2,I,K

Queue Formats: MULI, MULI2

Register Format: MULI rout1, rin1, imm

Description: Multiply signed word *in1* by signed immediate *imm*. Place low word of result in *out1*.

Function: `out1 ← in1 x imm`

Queue
```
POP   in1
PUSH (in1 x imm)|LO
```

Register `R[rout1] ← (R[rin1] x imm)|LO`

MULIU 0xE3 Unsigned Partial Multiply by Immediate 2,I,K

Queue Formats: MULIU, MULIU2

Register Format: MULIU rout1, rin1, imm

Description: Multiply unsigned word *in1* by unsigned immediate *imm*. Place low word of result in *out1*.

Function: `out1 ← in1 x imm`

Queue
```
POP   in1
PUSH (in1 x imm)|LO
```

Register
```
R[rout1] ← (R[rin1] x imm)|LO
```

MULT 0x46 Signed Multiply K,X

Queue Formats: MULT, MULTX

Register Format: MULT rout1, rout2, rin1, rin2

Description: Multiply two unsigned words, *in1* and *in2*. Place low word of result in *out1* and high word of result in *out2*.

Function: `(out1, out2) ← in1 x in2`

Queue
```
POP   in1, in2
PUSH (in1 x in2)|LO, (in1 x in2)|HI
```

Register
```
R[rout1] ← (R[rin1] x R[rin2])|LO
R[rout2] ← (R[rin1] x R[rin2])|HI
```

MULTI 0xE6 Signed Multiply by Immediate I,X,K

Queue Formats: MULTI, MULTIX

Register Format: MULTI rout1, rout2, rin1, imm

Description: Multiply signed word *in1* by signed immediate *imm*. Place low word of result in *out1* and high word of result in *out2*.

Function: `(out1, out2) ← in1 x imm`

Queue
```
POP   in1
PUSH (in1 x imm)|LO,(in1 x imm)|HI
```

Register
```
R[rout1]     ← (R[rin1] x imm)|LO
R[rout1 + 1] ← (R[rin1] x imm)|HI
```

MULTIU	0XE7	Unsigned Multiply by Immediate	I,X,K

Queue Formats: MULTIU, MULTIUX

Register Format: MULTIU rout1, rout2, rin1, imm

Description: Multiply unsigned word *in1* by unsigned immediate *imm*. Place low word of result in *out1* and high word in *out2*.

Function: `(out1, out2) ← in1 x imm`

Queue
```
POP   in1
PUSH (in1 · imm)|LO,(in1 x imm)|HI
```

Register
```
R[rout1]     ← (R[rin1] x imm)|LO
R[rout1 + 1] ← (R[rin1] x imm)|HI
```

MULTU	0xE5	Unsigned Multiply	K

Queue Formats: MULTU, MULTUX

Register Format: MULTU rout1, rout2, rin1, rin2

Description: Multiply two unsigned words, *in1* and *in2*. Place low word of result in *out1* and high word of result in *out2*.

Function: `(out1, out2) ← in1 x in2`

Queue
```
POP   in1, in2
PUSH (in1 x in2)|LO,(in1 x in2)|HI
```

Register
```
R[rout1]     ← (R[rin1] x R[rin2])|LO
R[rout1 + 1] ← (R[rin1] x R[rin2])|HI
```

MULU	0xE2	Unsigned Partial Multiply	2,K

Queue Formats: MULU, MULU2

Register Format: MULU rout1, rout2, rin1, rin2

Description: Multiply two unsigned words, *in1* and *in2*. Place low word of result in *out1*.

Function: `out1 ← in1 x in2`

Queue
```
POP   in1, in2
PUSH (in1 x in2)|LO
```

Register
```
R[rout1] ← (R[rin1] x R[rin2])|LO
```

NOP	0x00	No Operation	K

Queue Formats: NOP

Register Format: NOP

Description: No operation.

Function: None.

NOR	0x54	Nor	2,K

Queue Formats: NOR, NOR2

Register Format: NOR rout1, rin1, rin2

Description: Bitwise Boolean NOR of words *in1* and *in2*. Place result in *out1*.

Function: `out1 ← in1 NOR in2`

Queue
```
POP   in1, in2
PUSH  in1 NOR in2
```

Register `R[rout1] ← (R[in1] NOR R[in2])`

NORI	0xEA	Nor Immediate	2,I,K

Queue Formats: NORI, NORI2

Register Format: NORI rout1, rin1, imm

Description: Bitwise Boolean NOR of words *in1* and immediate *imm*.
Place result in *out1*.

Function: `out1 ← in1 NOR imm`

Queue
```
POP   in1
PUSH  in1 NOR imm
```

Register `R[rout1] ← (R[rin1] NOR imm)`

NOT	0x5F	Not	2,K

Queue Formats: NOT, NOT2

Register Format: NOT rout1, rin1

Description: Bitwise Boolean NOT of word *in*. Place result in *out1*.

Function: `out1 ← NOT in1`

Queue

```
POP   in1
PUSH  NOT in1
```

Register `GPR[rout1] ← (NOT GPR[in1])`

OR	0x50	Or	2,K

Queue Formats: OR, OR2

Register Format: OR rout1, rin1, rin2

Description: Bitwise Boolean OR of words *in1* and *in2*. Place result in *out1*.

Function: `out1 ← in1 OR in2`

Queue

```
POP   in1, in2
PUSH  in1 OR in2
```

Register `R[rout1] ← (R[in1] OR R[in2])`

ORI	0x51	Or Immediate	2,I,K

Queue Formats: ORI, ORI2

Register Format: ORI rout1, rin1, imm

Description: Bitwise Boolean OR of words *in1* and immediate *imm*. Place result in *out1*.

Function: `out1 ← in1 OR imm`

Queue

```
POP   in1
PUSH  in1 OR imm
```

Register `R[rout1] ← (R[rin1] OR imm)`

PASS	0xDE	Pass	Q,2,K

Queue Formats: PASS, PASS2

Register Format: N/A

Description: Queue only. Remove word at front of queue, place at end of queue.

Function: `out1 ← in1`

Queue
```
POP   in1
PUSH  in1
```

RECV	0xB0	Receive Data	2,K

Queue Formats: RECV, RECV2

Register Format: RECV rout1, rin1

Description: Receive word from memory port given by *in1*. Place word in *out1*. Address in memory port incremented after read.

Function: `out1 ← Receive_word(Port = in1)`

Queue
```
POP   in1
PUSH  Receive_word(Port = in1)
```

Register `R[rout1] ← Receive_word(Port = R[rin1])`

RECVB	0x60	Receive Data Byte	2,K

Queue Formats: RECVB, RECVB2

Register Format: RECVB rout1, rin1

Description: Receive signed byte from memory port given by *in1*. Place sign-extended byte in *out1*. Address in memory port incremented after read.

Function: `out1 ← Receive_byte(Port = in1)`

Queue
```
POP   in1
PUSH  Receive_byte(Port = in1)
```

Register `R[rout1] ← Receive_byte(Port = R[rin1])`

RECVBP	0x61	Receive Data Byte Port Imm.	2,I,K

Queue Formats: RECVBP, RECVBP2

Register Format: RECVBP rout1, port

Description: Receive signed byte from memory port given by *port*. Place sign-extended byte in *out1*. Address in memory port incremented after read.

Function: `out1 ← Receive_byte(Port = port)`

Queue `PUSH  Receive_byte(Port = port)`

Register `R[rout1] ← Receive_byte(Port= port)`

RECVBPU	0xF1	Receive Data Byte Port Imm. Unsigned	2,I,K

Queue Formats: RECBPU, RECBPU2

Register Format: RECVBPU rout1, port

Description: Receive unsigned byte from memory port given by *port*.
Place sign-extended byte in *out1*. Address in memory port incremented after read.

Function: `out1 ← Receive_byte(Port = port)`

Queue `PUSH  Receive_byte(Port = port)`

Register `R[rout1] ← Receive_byte(Port = port)`

RECVBU	0xF0	Receive Data Byte Unsigned	2,K

Queue Formats: RECVBU, RECVBU2

Register Format: RECVBU rout1, rin1

Description: Receive unsigned byte from memory port given by *in1*. Place byte in *out1*. Address in memory port incremented after read.

Function: `out1 ← Receive(Port = in1)`

Queue
```
POP   in1
PUSH  Receive_byte(Port = in1)
```

Register `R[rout1] ← Receive_byte(Port = R[rin1])`

RECVH	0x64	Receive Data Halfword	2,K

Queue Formats: RECVH, RECVH2

Register Format: RECVH rout1, rin1

Description: Receive halfword from memory port given by *in1*. Place half-word in *out1*. Address in memory port incremented after read.

Function:
```
out1 ← Receive_halfword(Port = in1)
```

Queue
```
POP   in1
PUSH  Receive_halfword(Port = in1)
```

Register
```
R[rout1] ← Receive_halfword(Port = R[rin1])
```

RECVHP	0x65	Receive Data Halfword Port Imm.	2,I,K

Queue Formats: RECVHP, RECVHP2

Register Format: RECVHP rout1, port

Description: Receive halfword from memory port given by *port*. Place halfword in *out1*. Address in memory port incremented after read.

Function:
```
out1 ← Receive_halfword(Port = port)
```

Queue
```
PUSH  Receive_halfword(Port = port)
```

Register
```
R[rout1] ← Receive_halfword(Port = port)
```

RECVHPU	0xF5	Receive Data Halfword Uns. Port Imm.	2,I,K

Queue Formats: RECVHPU, RECVHPU2

Register Format: RECVHPU rout1, port

Description: Receive unsigned halfword from memory port given by *port*. Place halfword in *out1*. Address in memory port incremented after read.

Function:
```
out1 ← Receive_halfword(Port = port)
```

Queue
```
PUSH  Receive_halfword(Port = port)
```

Register
```
R[rout1] ← Receive_halfword(Port = port)
```

RECVHU	0xF4	Receive Data Halfword Unsigned	2,K

Queue Formats: RECVHU, RECVHU2

Register Format: RECVHU rout1, rin1

Description: Receive unsigned halfword from memory port given by *in1*. Place halfword in *out1*. Address in memory port incremented after read.

Function: `out1 ← Receive_halfword(Port = in1)`

Queue
```
POP   in1
PUSH  Receive_halfword(Port = in1)
```

Register `R[rout1] ← Receive_halfword(Port = R[rin1])`

RECVP	0xB1	Receive Data Port Imm.	2,I,K

Queue Formats: RECVP, RECVP2

Register Format: RECVP rout1, port

Description: Receive word from memory port given by *port*. Place word in *out1*. Address in memory port incremented after read.

Function: `out1 ← Receive(Port = port)`

Queue `PUSH  Receive(Port = port)`

Register `GPR[rout1] ← Receive(Port = port)`

SEL	0xE8	Select	2,K,R

Queue Formats: SEL, SEL2, SELR, SEL2R

Register Format: SEL rout1, rin1, rin2 (*rin3* = *rin2* + *1*)

Description: If LSB of *in1* equals 1, place *in2* in *out1*. Otherwise, place *in3*.

Function:
Forward: `out1 ← (in1 ? in2 : in3)`
Reverse: `out1 ← (in1 ? in3 : in2)`

Queue
```
POP   in1, in2, in3
PUSH (in1 ? in2 : in3) [R: (in1 ? in2 : in3)]
```

Register `GPR[rout1] ← (GPR[rin1] ? GPR[rin2] : GPR[rin3])`

SELX 0xFA Select X (Queue variant) Q,2,K,R

Queue Formats: SELX, SELX2, SELXR, SELX2R

Register Format: N/A

Description: If LSB of *in2* equals 1, place *in1* in *out1*. Otherwise, place *in3* in *out1*.

Function:

```
Forward:  out1 ← (in2 ? in1 : in3)
Reverse:  out1 ← (in2 ? in3 : in1)
```

Queue

```
POP   in1, in2, in3
PUSH (in2 ? in1 : in3) [R: (in2 ? in3 : in1)]
```

SELY 0xFB Select Y (Queue variant) Q,2,K,R

Queue Formats: SELY, SELY2, SELYR, SELY2R

Register Format: N/A

Description: If LSB of *in3* equals 1, place *in1* in *out1*. Otherwise, place *in2* in *out1*.

Function:

```
Forward:  out1 ← (in3 ? in1 : in2)
Reverse:  out1 ← (in3 ? in2 : in1)
```

Queue

```
POP   in1, in2, in3
PUSH (in3 ? in1 : in2) [R: (in3 ? in2 : in1)]
```

SEND 0xB2 Send Data R,K

Queue Formats: SEND, SENDR

Register Format: SEND rin1, rin2

Description: Send word *in1* to memory port given by *in2*.

Function: `Send(in1, Port = in2)`

Queue

```
POP   in1, in2
Send(in1, Port = in2)
```

Register

```
Send(R[rin1], Port = R[rin2])
```

SENDB	0x68	Send Data Byte	R,K

Queue Formats: SENDB, SENDBR

Register Format: SENDB rin1, rin2

Description: Send byte *in1* to memory port given by *in2*.

Function:
```
Send_byte(in1, Port = in2)
```

Queue
```
POP   in1, in2
Send_byte(in1, Port = in2)
```

Register
```
Send_byte(R[rin1], Port = R[rin2])
```

SENDBP	0x69	Send Data Byte Port Immediate	I,K

Queue Formats: SENDBP

Register Format: SENDBP rin1, port

Description: Send byte *in1* to memory port given by *port*.

Function:
```
Send_byte(in1, Port = port)
```

Queue
```
POP   in1
Send_byte(in1, Port = port)
```

Register
```
Send_byte(R[rin1], Port = port)
```

SENDH	0x6A	Send Data Halfword	R,K

Queue Formats: SENDH, SENDHR

Register Format: SENDH rin1, rin2

Description: Send halfword *in1* to memory port given by *in2*.

Function:
```
Send_halfword(in1, Port = in2)
```

Queue
```
POP   in1, in2
Send_halfword(in1, Port = in2)
```

Register
```
Send_halfword(R[rin1], Port = R[rin2])
```

SENDHP	0x6B	Send Data Halfword Port Immediate	I,K

Queue Formats: SENDHP

Register Format: SENDHP rin1, port

Description: Send halfword *in1* to memory port given by *port*.

Function:
```
Send_halfword(in1, Port = port)
```

Queue
```
POP   in1
Send_halfword(in1, Port = port)
```

Register
```
Send_halfword(R[rin1], Por t= port)
```

SENDP	0xB3	Send Data Port Immediate	I,K

Queue Formats: SENDP

Register Format: SENDP rin1, port

Description: Send word *in1* to memory port given by *port*.

Function:
```
Send(in1, Port = port)
```

Queue
```
POP   in1
Send(in1, Port = port)
```

Register
```
Send(R[rin1], Port = port)
```

SEQ	0xED	Set If Equal	2,K

Queue Formats: SEQ, SEQ2

Register Format: SEQ rout1, rin1, rin2

Description: Send *out1* to one if *in1* equals *in2*, zero otherwise.

Function:
```
out1 = (in1 == in2)
```

Queue
```
POP   in1, in2
PUSH  (in1 == in2)
```

Register
```
R[rout1] = (R[rin1] == R[rin2])
```

SEQI	0xEF	Set If Equal Immediate	I,2,K

Queue Formats: SEQI, SEQI2

Register Format: SEQI rout1, rin1, imm

Description: Send *out1* to one if *in1* equals *imm*, zero otherwise.

Function:
```
out1 = (in1 == imm)
```

Queue
```
POP   in1
PUSH  (in1 == imm)
```

Register
```
R[rout1] = (R[rin1] == imm )
```

SETB	0xB5	Set Port Base Address	R,K

Queue Formats: SETB, SETBR

Register Format: SETB rin1, address

Description: Send base address for port given by *in1* to *address*.

Function:
```
Set_base_address(address, Port = in1)
```

Queue
```
POP   in1
Set_base_address(address, Port = in1)
```

Register
```
Set_base_address(address, Port = R[rin1])
```

SETBP	0xB7	Set Port Immediate Base Address	I,K

Queue Formats: SETBP

Register Format: SETB port, address

Description: Send base address for *port* to *address*.

Function:
```
Set_base_address(address, Port = port)
```

Queue
```
Set_base_address(address, Port = port)
```

Register
```
Set_base_address(address, Port = port)
```

SETS	0xB8	Set Port Stride	R,K

Queue Formats: SETS, SETSR

Register Format: SETS rin1, rin2

Description: Send stride for port given by *in1* to stride in *rin2*.

Function: `Set_stride(stride = in2, Port = in1)`

Queue
```
POP   in1, in2
Set_stride(stride = in2, Port = in1)
```

Register
```
Set_stride(stride = R[in2], Port = R[in1])
```

SETSI	0xB9	Set Port Stride Immediate	I,K

Queue Formats: SETSI

Register Format: SETSI rin1, stride

Description: Send stride for port given by *in1* to *stride*.

Function: `Set_stride(stride = stride, Port = in1)`

Queue
```
POP   in1
Set_stride(stride = stride, Port = in1)
```

Register
```
Set_stride(stride = stride, Port = R[in1])
```

SETSP	0xBA	Set Port Immediate Stride	I,K

Queue Formats: SETSP

Register Format: SETB port, rin1

Description: Send stride for port to value in *in1*.

Function: `Set_stride(stride = in1, Port = port)`

Queue
```
POP   in1
Set_stride(stride = in1, Port = port)
```

Register
```
Set_stride(stride = R[in1], Port = port)
```

SETSPI	0xBB	Set Port Immediate Stride Immediate	I,K

Queue Formats: SETSPI

Register Format: SETSPI port, stride

Description: Send stride for *port* to *stride*.

Function: `Set_stride(stride = stride, Port = port)`

Queue `Set_stride(stride = stride, Port = port)`

Register `Set_stride(stride = stride, Port = port)`

SLL	0x56	Shift Left Logical	2,K R

Queue Formats: SLL, SLL2, SLLR, SLL2R

Register Format: SLL rout1, rin1, rin2

Description: Shift word *in1* left number of places indicated by *in2*. Place result in *out1*.

Function: `out1 ← in1 << in2`

Queue
```
POP   in1, in2
PUSH (in1 << in2)
```

Register `R[rout1] ← (R[rin1] << R[rin2])`

SLLI	0x55	Shift Left Logical Immediate	2,I,K

Queue Formats: SLLI, SLLI2

Register Format: SLLI rout1, rin1, shamt

Description: Shift word *in1* left number of places indicated by immediate *shamt*. Place result in *out1*.

Function: `out1 ← in1 << shamt`

Queue
```
POP   in1
PUSH (in1 << shamt)
```

Register `R[rout1] ← (R[rin1] << shamt)`

SLT	0x5B	Set Less Than	2,K,R

Queue Formats: SLT, SLT2, SLTR, SLT2R

Register Format: SLT rout1, rin1, rin2

Description: Set *out1* to 1 if *in1* is less than *in2* , otherwise set *out1* to 0.

Function: `out1 ← (in1 < i2 ? 1 : 0 )`

Queue

```
POP   in1, in2
PUSH (in1 < in2 ? 1 : 0)
```

Register `R[rout1] ← (R[rin1] < R[rin2] ? 1 : 0)`

SLTI	0x5C	Set Less Than Immediate	2,I,K

Queue Formats: SLTI, SLTI2

Register Format: SLTI rout1, rin1, imm

Description: Set *out1* to 1 if *in1* is less than immediate *imm* , otherwise set *out1* to 0.

Function: `out1 ← (in1 < imm ? 1 : 0 )`

Queue

```
POP   in1, imm
PUSH (in1 < imm ? 1 : 0)
```

Register `R[rout1] ← (R[rin1] < imm ? 1 : 0)`

SLTIU	0x5E	Set Less Than Immediate Unsigned	2,I,K

Queue Formats: SLTIU, SLTIU2

Register Format: SLTIU rout1, rin1, imm

Description: Set *out1* to 1 if *in1* is less than immediate *imm* , otherwise set *out1* to 0.

Function: `out1 ← (in1 < imm ? 1 : 0 )`

Queue

```
POP   in1, imm
PUSH (in1 < imm ? 1 : 0)
```

Register `R[rout1] ← (R[rin1] < imm ? 1 : 0)`

SLTU 0x5D Set Less Than Unsigned 2,K,R

Queue Formats: SLTU, SLTU2, SLTUR, SLTU2R

Register Format: SLTU rout1, rin1, rin2

Description: Set *out1* to 1 if *in1* is less than *in2* , otherwise set *out1* to 0.

Function: `out1 ← (in1 < i2 ? 1 : 0 )`

Queue
```
POP   in1, in2
PUSH (in1 < i2 ? 1 : 0)
```

Register `R[rout1] ← (R[rin1] < R[in2] ? 1 : 0)`

SNE 0xEE Set If Not Equal 2,K

Queue Formats: SNE, SNE

Register Format: SNE rout1, rin1, rin2

Description: Set *out1* to 1 if *in1* is not equal to *in2* , otherwise set *out1* to 0.

Function: `out1 ← (in1 !=i2 ? 1 : 0 )`

Queue
```
POP   in1, in2
PUSH (in1 != i2 ? 1 : 0)
```

Register `R[rout1] ← (R[rin1] != R[in2] ? 1 : 0)`

SNEI 0xE0 Set If Not Equal Immediate I,K,R

Queue Formats: SNEI

Register Format: SNEI rout1, rin1, imm

Description: Set *out1* to 1 if *in1* is not equal to *imm* , otherwise set *out1* to 0.

Function: `out1 ← (in1 !=i2 ? 1 : 0 )`

Queue
```
POP   in1
PUSH (in1 != imm  ? 1 : 0)
```

`Register  R[rout1] ← (R[rin1] != imm ? 1 : 0)`

SRA	0x5A	Shift Right Arithmetic	2,K,R

Queue Formats: SRA, SRA2, SRAR, SRA2R

Register Format: SRA rout1, rin1, rin2

Description: Shift word *in1* right number of places indicated by *in2*. Shift in 1 or 0 according to MSB to preserve sign. Place result in *out1*.

Function: `out1 ← in1 >> in2`

Queue
```
POP   in1, in2
PUSH (in1 >> in2)
```

Register `R[rout1] ← (R[rin1] >> R[rin2])`

SRAI	0x59	Shift Right Arithmetic Immediate	2,I,K

Queue Formats: SRAI, SRAI2

Register Format: SRAI rout1, rin1, shamt

Description: Shift word *in1* right number of places indicated by *shamt*. Shift in 1 or 0 according to MSB to preserve sign. Place result in *out1*.

Function: `out1 ← in1 >> shamt`

Queue
```
POP   in1
PUSH (in1 >> shamt)
```

Register `R[rout1] ← (R[rin1] >> shamt)`

SRL	0x58	Shift Right Logical	2,K,R

Queue Formats: SRL, SRL2, SRLR, SRL2R

Register Format: SRL rout1, rin1, rin2

Description: Shift word *in1* right number of places indicated by *in2*. Place result in *out1*.

Function: `out1 ← in1 >> in2`

Queue
```
POP   in1, in2
PUSH (in1 >> in2)
```

Register `R[rout1] ← (R[rin1] >> R[rin2])`

SRLI	0x57	Shift Right Logical Immediate	2,I,K

Queue Formats: SRLI, SRLI2

Register Format: SRLI rout1, rin1, shamt

Description: Shift word *shamt* right number of places indicated by immediate *imm*. Place result in *out1*.

Function: `out1 ← in1 >> shamt`

Queue

```
POP   in1
PUSH (in1 >> shamt)
```

Register `R[rout1] ← (R[rin1] >> shamt)`

SUB	0x44	Subtract	2,K,R

Queue Formats: SUB, SUB2, SUBR, SUB2R

Register Format: SUB rout1, rin1, rin2

Description: Subtract word *in2* from word *in1*. Place result in *out1*.

Function: `out1 ← in1 - in2`

Queue:

```
POP   in1, in2
PUSH (in1 - in2)
```

Register: `R[rout1] ← (R[rin1] - R[rin2])`

SUBI	0xF7	Subtract Immediate	2,I,K

Queue Formats: SUBI, SUBI2

Register Format: SUBI rout1, rin1, imm

Description: Subtract immediate value *imm* from word *in1*. Place result in *out1*.

Function: `out1 ← in1 - imm`

Queue:

```
POP   in1
PUSH (in1 - imm)
```

Register: `R[rout1] ← (R[rin1] - imm)`

SUBIU	0xFC	Subtract Immediate Unsigned	2,I,K

Queue Formats: SUBIU, SUBIU2

Register Format: SUBIU rout1, rin1, imm

Description: Subtract immediate value *imm* from word *in1*. Place result in *out1*.

Function: `out1 ← in1 - imm`

Queue:

```
POP   in1
PUSH (in1 - imm)
```

Register: `R[rout1] ← (R[rin1] - imm)`

SUBU	0x45	Subtract Unsigned	2,K,R

Queue Formats: SUBU, SUBU2, SUBUR, SUBU2R

Register Format: SUBU rout1, rin1, rin2

Description: Subtract word *in2* from word *in1*. Place result in *out1*.

Function: Forward: `out1 ← in1 - in2`

Queue:

```
POP   in1, in2
PUSH (in1 - in2)
```

Register: `R[rout1] ← (R[rin1] - R[in2])`

SWAP	0xFD	Swap	Q,K

Queue Formats: SWAP

Register Format: N/A

Description: Queue only. Swap two words at front of queue, place at end of queue.

Function: `(out1, out2) ← (in2, in1)`

Queue

```
POP   in1, in2
PUSH  in2, in1
```

XOR	0x52	Exclusive Or	2,K

Queue Formats: XOR, XOR2

Register Format: XOR rout1, rin1, rin2

Description: Bitwise Boolean XOR of words *in1* and *in2*. Place result in *out1*.

Function: `out1 ← in1 XOR in2`

Queue

```
POP   in1, in2
PUSH  in1 XOR in2
```

Register `R[rout1] ← (R[in1] XOR R[in2])`

XORI	0x53	Exclusive Or Immediate	2,I,K

Queue Formats: XORI, XORI2

Register Format: XORI rout1, rin1, imm

Description: Bitwise Boolean XOR of words *in1* and immediate *imm*. Place result in *out1*.

Function: `out1 ← in1 XOR imm`

Queue

```
POP   in1
PUSH  in1 XOR imm
```

Register
`R[rout1] ← (R[rin1] XOR imm)`

ZACC	0x53	Accumulate From Zero	2,K

Queue Formats: ZACC, ZACC2

Register Format: ZACC rout1, rin1

Description: Starting with zero, accumulate values from *rin1*. Place result in *out1*.

Function: `out1 ← in1 + acc (acc = acc + in1)`

Queue `PUSH  acc + in1 (acc = acc + in1)`

Register `R[rin1] ← (R[rin1] + acc)`

Appendix C

Benchmark Kernels

Note: Source code for kernels used in the ATR application are in Appendix F.

DCT1

```
/*jfdctint.
*
* Copyright (C) 1991-1994, Thomas G. Lane.
* This file is part of the Independent JPEG Group's software.
* For conditions of distribution and use, see the accompanying README file.
*
* This file contains a slow-but-accurate integer implementation of the
* forward DCT (Discrete Cosine Transform).
*
* A 2-D DCT can be done by 1-D DCT on each row followed by 1-D DCT
* on each column.  Direct algorithms are also available, but they are
* much more complex and seem not to be any faster when reduced to code.
*
* This implementation is based on an algorithm described in
* C. Loeffler, A. Ligtenberg and G. Moschytz, "Practical Fast 1-D DCT
* Algorithms with 11 Multiplications", Proc.  Int'l.  Conf.  on Acoustics,
* Speech, and Signal Processing 1989 (ICASSP '89), pp.  988-991.
* The primary algorithm described there uses 11 multiplies and 29 adds.
* We use their alternate method with 12 multiplies and 32 adds.
* The advantage of this method is that no data path contains more than one
* multiplication; this allows a very simple and accurate implementation in
* scaled fixed-point arithmetic, with a minimal number of shifts.
*/
#include <stdio.h>
```

```
#include <stdlib.h>
// Original file was jfdctint.c
// and was Copyright (C) 1991-1994, Thomas G. Lane.
// Modified 19May2003 by Ben Levine
// Removed references to external header files and fixed options.
// 20May2003 - Ben Levine
// Added simple main() and sample input and output:
// Operation verified in MATLAB
//
typedef short INT16;
typedef long INT32;
#define ONE ((INT32) 1)
#define NUM_DATA 10000
#define DCTSIZE 8
#define BITS_IN_JSAMPLE == 8
#define CONST_BITS 13
#define PASS1_BITS 2
#define DESCALE(x,n) RIGHT_SHIFT((x) + (ONE << ((n)-1)), n)
#define FIX_0_298631336 ((INT32) 2446)
#define FIX_0_390180644 ((INT32) 3196)
#define FIX_0_541196100 ((INT32) 4433)
#define FIX_0_765366865 ((INT32) 6270)
#define FIX_0_899976223 ((INT32) 7373)
#define FIX_1_175875602 ((INT32) 9633)
#define FIX_1_501321110 ((INT32) 12299)
#define FIX_1_847759065 ((INT32) 15137)
#define FIX_1_961570560 ((INT32) 16069)
#define FIX_2_053119869 ((INT32) 16819)
#define FIX_2_562915447 ((INT32) 20995)
#define FIX_3_072711026 ((INT32) 25172)
/* Multiply an INT32 variable by an INT32 constant to yield an INT32 result.
* For 8-bit samples with the recommended scaling, all the variable
* and constant values involved are no more than 16 bits wide, so a
* 16x16->32 bit multiply can be used instead of a full 32x32 multiply.
* For 12-bit samples, a full 32-bit multiplication will be needed.
*/
#define MULTIPLY(var,const) (((INT16) (var)) * ((INT16) (const)))
#define RIGHT_SHIFT(x,shft) ((x) >> (shft))
typedef int DCTELEM;
void fdct1 (DCTELEM *data);
int main (void)
{
int i;
int *p_data;
p_data = malloc(DCTSIZE*NUM_DATA*sizeof(DCTELEM));
for (i = 0; i < (DCTSIZE * NUM_DATA); i++) {
p_data[i] = (int)((unsigned char)rand());
}
fdct1(p_data);
return 0;
```

```
}
// 2 KERNELS
void fdct1 (DCTELEM* data)
{
DCTELEM in0, in1, in2, in3, in4, in5, in6, in7;
DCTELEM out0, out1, out2, out3, out4, out5, out6, out7;
DCTELEM tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
DCTELEM tmp10, tmp11, tmp12, tmp13, tmp14;
DCTELEM z1, z2, z3, z4, z5, z6;
DCTELEM *dataptr;
int ctr;
dataptr = data;
asm("dlpbgn");
for (ctr = NUM_DATA; ctr >= 0; ctr--) {
in0 = dataptr[0];
in1 = dataptr[1];
in2 = dataptr[2];
in3 = dataptr[3];
in4 = dataptr[4];
in5 = dataptr[5];
in6 = dataptr[6];
in7 = dataptr[7];
tmp0 = in0 + in7;
tmp7 = in0 - in7;
tmp1 = in1 + in6;
tmp6 = in1 - in6;
tmp2 = in2 + in5;
tmp5 = in2 - in5;
tmp3 = in3 + in4;
tmp4 = in3 - in4;
tmp10 = tmp0 + tmp3;
tmp13 = tmp0 - tmp3;
tmp11 = tmp1 + tmp2;
tmp12 = tmp1 - tmp2;
out0 = (DCTELEM) ((tmp10 + tmp11) << PASS1_BITS);
out4 = (DCTELEM) ((tmp10 - tmp11) << PASS1_BITS);
tmp14 = tmp12 + tmp13;
z1 = MULTIPLY(tmp14, FIX_0_541196100);
out2 = (DCTELEM) DESCALE(z1 + MULTIPLY(tmp13, FIX_0_765366865),
CONST_BITS-PASS1_BITS);
out6 = (DCTELEM) DESCALE(z1 + MULTIPLY(tmp12, - FIX_1_847759065),
CONST_BITS-PASS1_BITS);
/* Odd part per figure 8 --- note paper omits factor of sqrt(2).
* cK represents cos(K*pi/16).
* i0..i3 in the paper are tmp4..tmp7 here.
*/
z1 = tmp4 + tmp7;
z2 = tmp5 + tmp6;
z3 = tmp4 + tmp6;
z4 = tmp5 + tmp7;
```

```
z6 = tmp4 + tmp5 + tmp6 + tmp7;
z5 = MULTIPLY(z6, FIX_1_175875602); /* sqrt(2) * c3 */
tmp4 = MULTIPLY(tmp4, FIX_0_298631336); /* sqrt(2) * (-c1+c3+c5-c7) */
tmp5 = MULTIPLY(tmp5, FIX_2_053119869); /* sqrt(2) * ( c1+c3-c5+c7) */
tmp6 = MULTIPLY(tmp6, FIX_3_072711026); /* sqrt(2) * ( c1+c3+c5-c7) */
tmp7 = MULTIPLY(tmp7, FIX_1_501321110); /* sqrt(2) * ( c1+c3-c5-c7) */
z1 = MULTIPLY(z1, - FIX_0_765366865); /* sqrt(2) * (c7-c3) */
z2 = MULTIPLY(z2, - FIX_2_562915447); /* sqrt(2) * (-c1-c3) */
z3 = MULTIPLY(z3, - FIX_1_961570560); /* sqrt(2) * (-c3-c5) */
z4 = MULTIPLY(z4, - FIX_0_390180644); /* sqrt(2) * (c5-c3) */
z3 += z5;
z4 += z5;
out7 = (DCTELEM) DESCALE(tmp4 + z1 + z3, CONST_BITS-PASS1_BITS);
out5 = (DCTELEM) DESCALE(tmp5 + z2 + z4, CONST_BITS-PASS1_BITS);
out3 = (DCTELEM) DESCALE(tmp6 + z2 + z3, CONST_BITS-PASS1_BITS);
out1 = (DCTELEM) DESCALE(tmp7 + z1 + z4, CONST_BITS-PASS1_BITS);
dataptr[0] = out0;
dataptr[1] = out1;
dataptr[2] = out2;
dataptr[3] = out3;
dataptr[4] = out4;
dataptr[5] = out5;
dataptr[6] = out6;
dataptr[7] = out7;
asm("dlpend");
dataptr += DCTSIZE; /* advance pointer to next row */
}
asm("dlpdone");
}
```

L2ALAW

```
/*
* This source code is a product of Sun Microsystems, Inc.  and is provided
* for unrestricted use.  Users may copy or modify this source code without
* charge.
*
* SUN SOURCE CODE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING
* THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR
* PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.
*
* Sun source code is provided with no support and without any obligation on
* the part of Sun Microsystems, Inc.  to assist in its use, correction,
* modification or enhancement.
*
* SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT TO THE
* INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS SOFTWARE
* OR ANY PART THEREOF.
*
* In no event will Sun Microsystems, Inc.  be liable for any lost revenue
* or profits or other special, indirect and consequential damages, even if
* Sun has been advised of the possibility of such damages.
*
* Sun Microsystems, Inc.
* 2550 Garcia Avenue
* Mountain View, California 94043
*/
// Original file g711.c from MediaBench/g721
// 21May2003 - Ben Levine
// converted search function for seg_end to
// if statements
// Added simple main() and sample input and output:
/* INPUT: */
/* -8415 */
/* -4899 */
/* 6203 */
/* 4308 */
/* 14191 */
/* 741 */
/* 18113 */
/* -699 */
/* 20585 */
/* 13148 */
/* 31634 */
/* 20095 */
/* 13340 */
/* -986 */
/* -25257 */
/* 10803 */
```

```
/* -8823 */
/* -23590 */
/* 4375 */
/* 21168 */
/* OUTPUT: */
/* 53 */
/* 6 */
/* 141 */
/* 133 */
/* 190 */
/* 242 */
/* 164 */
/* 112 */
/* 161 */
/* 188 */
/* 171 */
/* 166 */
/* 191 */
/* 123 */
/* 45 */
/* 176 */
/* 52 */
/* 34 */
/* 132 */
/* 161 */
#define QUANT_MASK (0xf) /* Quantization field mask.  */
#define SEG_SHIFT (4) /* Left shift for segment number.  */
#define DATA_SIZE 100
#include <stdlib.h>
void linear2alaw(int *pcm, unsigned int *alaw);
int main (void)
{
int i;
int *pcm_data;
unsigned int *alaw_data;
pcm_data = malloc(DATA_SIZE*sizeof(int));
alaw_data = malloc(DATA_SIZE*sizeof(unsigned int));
for (i = 0; i < DATA_SIZE; i++)
{
pcm_data[i] = (rand() >> 15) - 32768;
}
linear2alaw(pcm_data,alaw_data);
return 0;
}
void linear2alaw(int *pcm, unsigned int *alaw)
{
int mask;
int seg;
unsigned char aval;
int pcm_val;
```

```
int i;
for (i = 0; i < DATA_SIZE; i++)
{
asm("dlpbgn");
pcm_val = pcm[i];
if (pcm_val >= 0)
{
mask = 0xD5; /* sign (7th) bit = 1 */
}
else
{
mask = 0x55; /* sign bit = 0 */
pcm_val = -pcm_val - 8;
}
if (pcm_val <= 0xFF)
seg = 0;
else if (pcm_val <= 0x1FF)
seg = 1;
else if (pcm_val <= 0x3FF)
seg = 2;
else if (pcm_val <= 0x7FF)
seg = 3;
else if (pcm_val <= 0xFFF)
seg = 4;
else if (pcm_val <= 0x1FFF)
seg = 5;
else if (pcm_val <= 0x3FFF)
seg = 6;
else if (pcm_val <= 0x7FFF)
seg = 7;
else
seg = 8;
/* Combine the sign, segment, and quantization bits.  */
if (seg >= 8) /* out of range, return maximum value.  */
aval = 0x7F ^ mask;
else {
aval = seg << SEG_SHIFT;
if (seg < 2)
aval |= (pcm_val >> 4) & QUANT_MASK;
else
aval |= (pcm_val >> (seg + 3)) & QUANT_MASK;
aval = aval ^ mask;
}
alaw[i] = (unsigned int)aval;
asm("dlpend");
}
asm("dlpdone");
}
```

FIR8CPX

```
#include <stdlib.h>
typedef unsigned char UINT8;
typedef unsigned short UINT16;
typedef unsigned int UINT32;
#define NUM_DATA 10000
#define SEED 12345
#define C1R 0x0148
#define C2R 0xc109
#define C3R 0x9121
#define C4R 0x4ffc
#define C5R 0x70b2
#define C6R 0xaa11
#define C7R 0x9101
#define C8R 0x5712
#define C1I 0x9612
#define C2I 0x8b19
#define C3I 0x0fa1
#define C4I 0x9900
#define C5I 0xbc32
#define C6I 0x1287
#define C7I 0x4512
#define C8I 0x7612
void fir8cpx (UINT16* in_data, UINT32* out_data);
int main(void)
{
UINT16 *p_in;
UINT32 *p_out;
int i;
p_in = malloc(2*NUM_DATA*sizeof(UINT16));
p_out = malloc(2*NUM_DATA*sizeof(UINT32));
srand(SEED);
for (i = 0; i < (NUM_DATA*2); i++)
{
p_in[i] = rand() >> 15;
}
fir8cpx(p_in,p_out);
return 0;
}
// 1 KERNEL
void fir8cpx (UINT16* in_data, UINT32* out_data)
{
UINT16 valA_R, valB_R, valC_R, valD_R, valE_R, valF_R, valG_R, valH_R;
UINT16 valA_I, valB_I, valC_I, valD_I, valE_I, valF_I, valG_I, valH_I;
UINT32 sum;
UINT32 outR, outI;
int i;
asm("dlpbgn");
```

```
for (i=0;i<(NUM_DATA-8);i=i+2) {
valA_R = in_data[i];
valA_I = in_data[i+1];
valB_R = in_data[i+2];
valB_I = in_data[i+3];
valC_R = in_data[i+4];
valC_I = in_data[i+5];
valD_R = in_data[i+6];
valD_I = in_data[i+7];
valE_R = in_data[i+8];
valE_I = in_data[i+9];
valF_R = in_data[i+10];
valF_I = in_data[i+11];
valG_R = in_data[i+12];
valG_I = in_data[i+13];
valH_R = in_data[i+14];
valH_I = in_data[i+15];
outR = 0;
outR += valA_R * C1R;
outR -= valA_I * C1I;
outR += valB_R * C2R;
outR -= valB_I * C2I;
outR += valC_R * C3R;
outR -= valC_I * C3I;
outR += valD_R * C4R;
outR -= valD_I * C4I;
outR += valE_R * C5R;
outR -= valE_I * C5I;
outR += valF_R * C6R;
outR -= valF_I * C6I;
outR += valG_R * C7R;
outR -= valG_I * C7I;
outR += valH_R * C8R;
outR -= valH_I * C8I;
out_data[i] = outR;
outI = 0;
outI += valA_R * C1I;
outI -= valA_I * C1R;
outI += valB_R * C2I;
outI -= valB_I * C2R;
outI += valC_R * C3I;
outI -= valC_I * C3R;
outI += valD_R * C4I;
outI -= valD_I * C4R;
outI += valE_R * C5I;
outI -= valE_I * C5R;
outI += valF_R * C6I;
outI -= valF_I * C6R;
outI += valG_R * C7I;
outI -= valG_I * C7R;
```

```
outI += valH_R * C8I;
outI -= valH_I * C8R;
out_data[i+1] = outI;
asm("dlpend");
}
asm("dlpdone");
}
```

RGC2YCC

```
//
// Original file was jccolor.c
// and was Copyright (C) 1991-1994, Thomas G. Lane.
// Modified 19May2003 by Ben Levine
// Removed references to external header files and fixed options.
// Only using function rgb_ycc_convert, greatly modified.
// Modified so as not to require look up table.
// Also changed to use array pointers for input and
// output and not the data structures used in the JPEG code.
// Inputs are 8 bit RGB values, RGB for 1st pixel, followed by RGB for 2nd
// and so on.  Similarly for 8-bit YCC output.
// 20May2003 - Ben Levine
// Added simple main() and sample input and output:
// Operation verified in MATLAB (use actual integer values;
// rounding errors large compared to FP)
//
#include <stdlib.h>
typedef long INT32;
typedef unsigned char UINT8;
#define SCALEBITS 16 /* speediest right-shift on some machines */
#define ONE_HALF ((INT32) 1 << (SCALEBITS-1))
#define MAXJSAMPLE 255
#define R_Y 19595
#define G_Y 38470
#define B_Y 7471
#define R_CB 11000
#define G_CB 21709
#define B_CB 32768
#define R_CR 32768
#define G_CR 27439
#define B_CR 5329
#define N_PIXELS 10000
#define SEED 12345
void rgb_ycc_convert (UINT8* rgb_image, UINT8* ycc_image);
int main(void)
{
UINT8 *p_rgb;
UINT8 *p_ycc;
int i;
p_rgb = malloc(3*N_PIXELS*sizeof(UINT8));
p_ycc = malloc(3*N_PIXELS*sizeof(UINT8));
srand(SEED);
for (i = 0; i < (N_PIXELS*3); i++)
{
p_rgb[i] = rand();
}
rgb_ycc_convert(p_rgb,p_ycc);
```

```
return 0;
}
// 1 KERNEL
void rgb_ycc_convert (UINT8* rgb_image, UINT8* ycc_image)
{
INT32 r, g, b;
INT32 y,cb,cr;
int i;
asm("dlpbgn");
for (i=0;i<N_PIXELS;i++) {
r = rgb_image[3*i];
g = rgb_image[(3*i)+1];
b = rgb_image[(3*i)+2];
y = (R_Y * r) + (G_Y * g) + (B_Y * b) + ONE_HALF;
cb = - (R_CB * r) - (G_CB * g) + (B_CB * b) + (ONE_HALF * (MAXJSAMPLE+1));
cr = (R_CR * r) - (G_CR * g) - (B_CR * b) + (ONE_HALF * (MAXJSAMPLE+1));
ycc_image[3*i] = (y >> SCALEBITS);
ycc_image[(3*i)+1] = (cb >> SCALEBITS);
ycc_image[(3*i)+2] = (cr >> SCALEBITS);
asm("dlpend");
}
asm("dlpdone");
}
```

DEC_COR

```
#include <stdlib.h>
typedef unsigned char UINT8;
typedef unsigned short UINT16;
typedef unsigned int UINT32;
#define N_PIXELS 10000
#define SEED 12345
void dec_cor (UINT8* in_image, UINT16* out_image);
int main(void)
{
UINT8 *p_in;
UINT16 *p_out;
int i;
p_in = malloc(9*N_PIXELS*sizeof(UINT8));
p_out = malloc(N_PIXELS*sizeof(UINT16));
srand(SEED);
for (i = 0; i < (N_PIXELS*9); i++)
{
p_in[i] = rand() >> 23;
}
dec_cor(p_in,p_out);
return 0;
}
// 1 KERNEL
void dec_cor (UINT8* in_image, UINT16* out_image)
{
UINT8 pelA, pelB, pelC, pelD, pelE, pelF, pelG, pelH, pelI;
UINT32 sum, step1, step2, term1, term2, term3, term4, term5, term6;
UINT16 outpel;
int i;
asm("dlpbgn");
for (i=0;i<N_PIXELS;i++) {
pelA = in_image[9*i];
pelB = in_image[(9*i)+1];
pelC = in_image[(9*i)+2];
pelD = in_image[(9*i)+3];
pelE = in_image[(9*i)+4];
pelF = in_image[(9*i)+5];
pelG = in_image[(9*i)+6];
pelH = in_image[(9*i)+7];
pelI = in_image[(9*i)+8];
sum = ((pelA + pelB) + (pelC + pelD)) + (pelE + pelE) + ((pelF + pelG) + (pelH
+ pelI + 1));
// Divide sum by 10
step1 = sum << 14;
term1 = step1 >> 4;
term2 = step1 >> 5;
term3 = step1 >> 8;
```

```
term4 = step1 >> 9;
term5 = step1 >> 12;
term6 = step1 >> 13;
step2 = term1 + term2 + term3 + term4 + term5 + term6;
outpel = step2 >> 14;
out_image[i] = outpel;
asm("dlpend");
}
asm("dlpdone");
}
```

IDEA

```
#include <stdio.h>
#include <stdlib.h>
#define DATASETSIZE 20000
#define IDEAKEYSIZE 16
#define IDEABLOCKSIZE 8
#define IDEAROUNDS 8
#define IDEAKEYLEN (6*IDEAROUNDS+4) //52
#define INSIZE (IDEAKEYSIZE+DATASETSIZE)
#define OUTSIZE 124 //(2*IDEAKEYLEN+2*DATASETSIZE)
#define M(x) ((x) & 0xffff)
#define SEED 12345
typedef unsigned int word32; //values are 0-4294967295
word32 ek[IDEAKEYLEN] =
{
0xf100,
0x1020,
0x3040,
0x5060,
0x7080,
0x90a0,
0xb0c0,
0x0020,
0x4060,
0x80a0,
0xc0e1,
0x0121,
0x4161,
0x81a1,
0xc1e2,
0xc101,
0x4181,
0xc202,
0x4282
};
#define MUL(x,y) (x = M(x-1), \
t16 = M((y)-1), \
t32 = x*t16 + x + t16 + 1, \
x = M(t32), \
t16 = t32>>16, \
x = (x-t16) + (x<t16))
static void ideaCipher(word32 *in, word32 *out);
int main(void)
{
int i;
word32 *p_in;
word32 *p_out;
p_in = malloc(DATASETSIZE*sizeof(word32));
```

```
p_out = malloc(DATASETSIZE*sizeof(word32));
srand(SEED);
for (i = 0; i < DATASETSIZE; i++)
{
p_in[i] = rand();
}
ideaCipher(p_in,p_out);
return 0;
}
static void
ideaCipher(word32 *in, word32 *out)
{
int i, j;
word32 *key;
word32 x1, x2, x3, x4, s2, s3;
//MUL temporaries
word32 t16;
word32 t32;
asm("dlpbgn");
for(j=0; j<(DATASETSIZE/2); j++)
{
key = ek;
x1 = x2 = *in++;
x3 = x4 = *in++;
x1 = ((x1>>8) & 0x0000ff00) | (x1>>24);
x2 = ((x2<<8) & 0x0000ff00) | ((x2>>8) & 0x000000ff);
x3 = ((x3>>8) & 0x0000ff00) | (x3>>24);
x4 = ((x4<<8) & 0x0000ff00) | ((x4>>8) & 0x000000ff);
MUL(x1, 0x0102);
x2 += 0x0304;
x3 += 0x0506;
MUL(x4, 0x0708);
s3 = x3;
x3 ^= x1;
MUL(x3, 0x090a);
s2 = x2;
x2 ^= x4;
x2 += x3;
MUL(x2, 0x0b0c);
x3 += x2;
x1 ^= x2;
x4 ^= x3;
x2 ^= s3;
x3 ^= s2;
MUL(x1, 0x0d0e);
x2 += 0x0f10;
x3 += 0x080a;
MUL(x4, 0x0c0e);
s3 = x3;
x3 ^= x1;
```

```
MUL(x3, 0x1012);
s2 = x2;
x2 ^= x4;
x2 += x3;
MUL(x2, 0x1416);
x3 += x2;
x1 ^= x2;
x4 ^= x3;
x2 ^= s3;
x3 ^= s2;
MUL(x1, 0x181a);
x2 += 0x1c1e;
x3 += 0x2002;
MUL(x4, 0x0406);
s3 = x3;
x3 ^= x1;
MUL(x3, 0x1c20);
s2 = x2;
x2 ^= x4;
x2 += x3;
MUL(x2, 0x2428);
x3 += x2;
x1 ^= x2;
x4 ^= x3;
x2 ^= s3;
x3 ^= s2;
MUL(x1, 0x2c30);
x2 += 0x3438;
x3 += 0x3c40;
MUL(x4, 0x0408);
s3 = x3;
x3 ^= x1;
MUL(x3, 0x0c10);
s2 = x2;
x2 ^= x4;
x2 += x3;
MUL(x2, 0x1418);
x3 += x2;
x1 ^= x2;
x4 ^= x3;
x2 ^= s3;
x3 ^= s2;
MUL(x1, 0x5058);
x2 += 0x6068;
x3 += 0x7078;
MUL(x4, 0x8008);
s3 = x3;
x3 ^= x1;
MUL(x3, 0x1018);
s2 = x2;
```

```
x2 ^= x4;
x2 += x3;
MUL(x2, 0x2028);
x3 += x2;
x1 ^= x2;
x4 ^= x3;
x2 ^= s3;
x3 ^= s2;
MUL(x1, 0x3038);
x2 += 0x4048;
x3 += 0xd0e0;
MUL(x4, 0xf100);
s3 = x3;
x3 ^= x1;
MUL(x3, 0x1020);
s2 = x2;
x2 ^= x4;
x2 += x3;
MUL(x2, 0x3040);
x3 += x2;
x1 ^= x2;
x4 ^= x3;
x2 ^= s3;
x3 ^= s2;
MUL(x1, 0x5060);
x2 += 0x7080;
x3 += 0x90a0;
MUL(x4, 0xb0c0);
s3 = x3;
x3 ^= x1;
MUL(x3, 0x0020);
s2 = x2;
x2 ^= x4;
x2 += x3;
MUL(x2, 0x4060);
x3 += x2;
x1 ^= x2;
x4 ^= x3;
x2 ^= s3;
x3 ^= s2;
MUL(x1, 0x80a0);
x2 += 0xc0e1;
x3 += 0x0121;
MUL(x4, 0x4161);
s3 = x3;
x3 ^= x1;
MUL(x3, 0x81a1);
s2 = x2;
x2 ^= x4;
x2 += x3;
```

```
   MUL(x2, 0xc1e2);
   x3 += x2;
   x1 ^= x2;
   x4 ^= x3;
   x2 ^= s3;
   x3 ^= s2;
   MUL(x1, 0xc101);
   x3 += 0x4181;
   x2 += 0xc202;
   MUL(x4, 0x4282);
   *out++ = ((x1<<24) & 0xff000000) | ((x1<<8) & 0x00ff0000) | ((x3<<8) & 0x0000ff00)
| ((x3>>8) & 0x000000ff);
   *out++ = ((x2<<24) & 0xff000000) | ((x2<<8) & 0x00ff0000) | ((x4<<8) & 0x0000ff00)
| ((x4>>8) & 0x000000ff);
   asm("dlpend");
   }
   asm("dlpdone");
   }
```

Appendix D

Example Testbench

```
library IEEE,std;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;
use work.io_utils.all;
ENTITY simple_test_top IS END simple_test_top;
ARCHITECTURE behavior OF simple_test_top IS
   file in_vectors  : text open read_mode is "simple.in.2.trace";
   file out_vectors : text open read_mode is "simple.out.2.trace";
   file output      : text open write_mode is "STD_OUTPUT";
COMPONENT simple
   port ( var_recvbu_0 : in std_logic_vector(7 downto 0);
          var_recvbu_1 : in std_logic_vector(7 downto 0);
          var_recvbu_2 : in std_logic_vector(7 downto 0);
          clk          : in std_logic;
          sendh_0      : out std_logic_vector(7 downto 0) );
END COMPONENT;
   SIGNAL var_recvbu_0 : std_logic_vector(7 downto 0);
   SIGNAL var_recvbu_1 : std_logic_vector(7 downto 0);
   SIGNAL var_recvbu_2 : std_logic_vector(7 downto 0);
   SIGNAL clk : std_logic := '1';
   SIGNAL sendh_0 : std_logic_vector(7 downto 0);
   constant ClockPeriod : time := 20 ns;
BEGIN
   clk <= not clk after ClockPeriod/2;
   uut: simple PORT MAP(
      var_recvbu_0 => var_recvbu_0,
      var_recvbu_1 => var_recvbu_1,
      var_recvbu_2 => var_recvbu_2,
      clk => clk,
      sendh_0 => sendh_0 );
   tb : PROCESS
```

```
    variable InVectorLine : line;
    variable OutVectorLine : line;
    variable BufLine : line;
    variable recvbu_0_var : integer;
    variable recvbu_1_var : integer;
    variable recvbu_2_var : integer;
    variable sendh_0_good : integer;
    variable pipeDelay : integer := 4;
    variable cycleCount : integer := 0;
    variable errorFound : integer := 0;
    BEGIN
       var_recvbu_0 <= (others=>'0');
       var_recvbu_1 <= (others=>'0');
       var_recvbu_2 <= (others=>'0');
       while not endfile(in_vectors) loop
          readline(in_vectors,InVectorLine);
          if (cycleCount > pipeDelay) then
             readline(out_vectors, OutVectorLine);
             read_based(OutVectorLine, sendh_0_good);
          end if;
          if InVectorLine(1) = '#' then next; end if;
          read_based(InVectorLine, recvbu_0_var );
          read_based(InVectorLine, recvbu_1_var );
          read_based(InVectorLine, recvbu_2_var );
          var_recvbu_0 <= std_logic_vector(to_unsigned(recvbu_0_var,8));
          var_recvbu_1 <= std_logic_vector(to_unsigned(recvbu_1_var,8));
          var_recvbu_2 <= std_logic_vector(to_unsigned(recvbu_2_var,8));
          if (cycleCount > pipeDelay) then
             if (std_logic_vector(to_unsigned(sendh_0_good,8)) /= sendh_0) then
                write(BufLine, string'("Error found for output sendh_0. Should be "));
                write(BufLine, sendh_0_good); write(BufLine, string'(", value computed is "));
                write(BufLine, to_integer(unsigned( sendh_0)));
                writeline(output, BufLine);
                errorFound := errorFound + 1;
            end if;
         end if;
         cycleCount := cycleCount + 1;
         wait for ClockPeriod;
      end loop;
      if (errorFound = 0) then
         write(BufLine, string'("No errors found"));
         writeline(output, BufLine);
      else
         write(BufLine, string'("ERRORS! Found "));
         write(BufLine, errorFound);
         write(BufLine, string'(" errors."));
         writeline(output, BufLine);
      end if;
      wait;
   END PROCESS;
END;
```

Appendix E

DAG for Simple Example Kernel in GML Format

```
graph [
    directed 1
    node [
        id 0
        nodeLabel "recvbu_0"
        nodeType "INPUT"
        nodeInputType "0_0"
        nodeOutputType "1_1"
        nodeWidth 9
        nodeOp "recvbu"
        nodeDest1Operand "$2"
        nodeSrc1Operand "P1"
        ports [
            port [
                name "NULL"
                x 0.0
                y 0.0
            ]
            port [
                name "out1"
                x 0.0
                y 1.0
            ]
        ]
        label "recvbu"
    ]
    node [
        id 1
        nodeLabel "recvbu_1"
        nodeType "INPUT"
        nodeInputType "0_0"
        nodeOutputType "1_1"
        nodeWidth 9
        nodeOp "recvbu"
        nodeDest1Operand "$2"
        nodeSrc1Operand "P2"
        ports [
            port [
                name "NULL"
                x 0.0
                y 0.0
```

```
        ]
        port [
            name "out1"
            x 0.0
            y 1.0
        ]
    ]
    label "recvbu"
]
node [
    id 2
    nodeLabel "recv_2"
    nodeType "INPUT"
    nodeInputType "0_0"
    nodeOutputType "1_1"
    nodeWidth 9
    nodeOp "recv"
    nodeDest1Operand "$47"
    nodeSrc1Operand "P3"
    ports [
        port [
            name "NULL"
            x 0.0
            y 0.0
        ]
        port [
            name "out1"
            x 0.0
            y 1.0
        ]
    ]
    label "recvbu"
]
node [
    id 3
    nodeLabel "srli_0"
    nodeType "OP"
    nodeInputType "1_1"
    nodeOutputType "1_1"
    nodeWidth 32
    nodeOp "srli"
    nodeDest1Operand "$2"
    nodeSrc1Operand "$2"
    nodeSrc2Operand "0x2"
    ports [
        port [
            name "NULL"
            x 0.0
            y 0.0
            ]
        port [
            name "in1"
            x 0.0
            y -1.0
        ]
        port [
            name "out1"
            x 0.0
            y 1.0
            ]
    ]
    label "slli"
]
```

```
node [
    id 4
    nodeLabel "addu_0"
    nodeType "OP"
    nodeInputType "2_2"
    nodeOutputType "1_1"
    nodeWidth 32
    nodeOp "addu"
    nodeDest1Operand "$3"
    nodeSrc1Operand "$2"
    nodeSrc2Operand "$2"
    ports [
        port [
            name "NULL"
            x 0.0
            y 0.0
        ]
        port [
            name "in1a"
            x -0.66
            y -1.0
        ]
        port [
            name "in1b"
            x 0.66
            y -1.0
        ]
        port [
            name "out1"
            x 0.0
            y 1.0
        ]
    ]
    label "addu"
]
node [
    id 5
    nodeLabel "addiu_0"
    nodeType "OP"
    nodeInputType "1_1"
    nodeOutputType "1_1"
    nodeWidth 32
    nodeOp "addiu"
    nodeDest1Operand "$48"
    nodeSrc1Operand "$3"
    nodeSrc2Operand "0x4"
    ports [
        port [
            name "NULL"
            x 0.0
            y 0.0
        ]
        port [
            name "in1"
            x 0.0
            y -1.0
        ]
        port [
            name "out1"
            x 0.0
            y 1.0
        ]
    ]
```

```
    label "addiu"
]
node [
    id 6
    nodeLabel "sltu_4"
    nodeType "OP"
    nodeInputType "2_1_1"
    nodeOutputType "1_1"
    nodeWidth 32
    nodeOp "subu"
    nodeDest1Operand "$44"
    nodeSrc1Operand "$47"
    nodeSrc2Operand "$3"
    ports [
        port [
            name "NULL"
            x 0.0
            y 0.0
        ]
        port [
            name "in1"
            x -0.66
            y -1.0
        ]
        port [
            name "in2"
            x 0.66
            y -1.0
        ]
        port [
            name "out1"
            x 0.0
            y 1.0
        ]
    ]
    label "sel"
]
node [
id 7
    nodeLabel "sel_0"
    nodeType "OP"
    nodeInputType "3_1_1_1"
    nodeOutputType "1_1"
    nodeWidth 32
    nodeOp "sel"
    nodeDest1Operand "$2"
    nodeSrc1Operand "$44"
    nodeSrc2Operand "$47"
    nodeSrc3Operand "$48"
    ports [
        port [
            name "NULL"
            x 0.0
            y 0.0
        ]
        port [
            name "in1"
            x -0.5
            y -1.0
        ]
        port [
            name "in2"
            x 0.0
```

```
            y -1.0
        ]
        port [
            name "in3"
            x 0.5
            y -1.0
        ]
        port [
            name "out1"
            x 0.0
            y 1.0
        ]
    ]
    label "sel"
]
node [
    id 8
    nodeLabel "send_0"
    nodeType "OUTPUT"
    nodeInputType "1_1"
    nodeOutputType "0_0"
    nodeWidth 32
    nodeOp "send"
    nodeDest1Operand "P4"
    nodeSrc1Operand "$2"
    ports [
        port [
            name "NULL"
            x 0.0
            y 0.0
        ]
        port [
            name "in1"
            x 0.0
            y -1.0
        ]
    ]
    label "send"
]
edge [
    source 0
    target 3
    sourcePort "out1"
    targetPort "in1"
]
 edge [
    source 1
    target 4
    sourcePort "out1"
    targetPort "in1a"
]
edge [
    source 2
    target 7
    sourcePort "out1"
    targetPort "in1b"
]
edge [
    source 2
    target 6
    sourcePort "out1"
    targetPort "in1a"
]
```

```
    edge [
        source 3
        target 4
        sourcePort "out1"
        targetPort "in1b"
    ]
    edge [
        source 4
        target 5
        sourcePort "out1"
        targetPort "in1"
    ]
    edge [
        source 4
        target 6
        sourcePort "out1"
        targetPort "in1b"
    ]
    edge [
        source 5
        target 7
        sourcePort "out1"
        targetPort "in1c"
    ]
    edge [
        source 6
        target 7
        sourcePort "out1"
        targetPort "in1a"
    ]
    edge [
        source 7
        target 8
        sourcePort "out1"
        targetPort "in1"
    ]
    node_style [
        name "default_node_style"
        style [
            graphics [
                fill "white"
                outline "black"
                width 1.0
            ]
        ]
    ]
]
```

Appendix F

ATR Sourcecode

```
#include <stdlib.h>
#include <stdio.h>
#define PIX_SORT(a,b,c) { if ((a)>(b)) PIX_SWAP((a),(b), (c)); }
#define PIX_SWAP(a,b,c) { (c) = (a);(a)=(b);(b)=(c); }
#define FRAME 10
typedef unsigned char UINT8;
typedef unsigned int UINT32;
typedef int INT32;
int num_rows;
int num_cols;
int thresh (UINT8* in_image, UINT8* out_image, int thresh_val);
void img_med (UINT8* in_image, UINT8* out_image);
void img_hp(UINT8* in_image, UINT8* out_image);
void img_prew(UINT8* in_image, UINT8* out_image);
void img_erode1(UINT8* in_image, UINT8* out_image);
void img_erode2(UINT8* in_image, UINT8* out_image);
void img_out(UINT8* in_image,UINT8* orig_img, UINT8* out_image);
int main(void)
{
  char buffer[50];
  FILE *input;
  FILE *output;
  int max_val;
  int num_pixels;
  UINT8 *in_img;
  UINT8 *out_img;
  UINT8 *tmp1_img;
  UINT8 *tmp2_img;
  char in[20];
  char out[20];
  char num[4];
  int r,c;
  int t_val;
  int num_k = 0;
  int num_on, min_on, max_on;
```

```
int done;
UINT8 pixel;
int i;
min_on = 500;
max_on = 5000;
for (i = 1; i <21 ; i++) {
  strcpy(in,"flir/ATR");
  if (i < 10) {
    num[0] = 48 + i;
    num[1] = 0;
  } else if (i <20) {
    num[0] = 49;
    num[1] = 48 + (i - 10);
    num[2] =0;
  } else {
    num[0] = 50;
    num[1] = 48;
    num[2] =0;
  }
  strcat(in,num);
  strcat(in,".pgm");
  strcpy(out,"out/out");
  strcat(out,num);
  strcat(out,".pgm");
  printf("IN = %s OUT = %s\n",in,out);
  input = fopen(in, "r");
  output = fopen(out, "w");
  fgets(buffer,50,input);
  if (strcmp(buffer,"P5\n") != 0) {
    printf("File %s is not a PGM file!\n",in);
   return(1);
  }
  fscanf(input,"%d %d",&num_cols,&num_rows);
  fscanf(input,"%d",&max_val);
  if (max_val > 255) {
    printf("Max val = %d : Too large!\n", max_val);
    return(1);
  }
  fprintf(output, "%s",buffer);
  fprintf(output, "%d %d\n",num_cols,num_rows);
  fprintf(output, "%d\n",max_val);
  num_pixels = num_rows * num_cols;
  if (i == 1) {
    in_img = malloc(num_pixels);
    out_img = malloc(num_pixels);
    tmp1_img = malloc(num_pixels);
    tmp2_img = malloc(num_pixels);
  }
  fread(in_img,1,num_pixels,input);
  img_med(in_img,tmp1_img);
  img_hp(tmp1_img,tmp2_img);
  img_prew(tmp2_img,tmp1_img);
  num_k += 3;
  done = 0;
  t_val = 128;
```

```
    while (done == 0) {
      num_k += 1;
      num_on = thresh(tmp1_img,tmp2_img,t_val);
      if (num_on > max_on) {
        t_val += 5;
      } else if (num_on < min_on) {
        t_val -= 5;
      } else {
        done = 1;
      }
    }
    img_erode1(tmp2_img,tmp1_img);
    img_erode2(tmp1_img,tmp2_img);
    img_out(tmp2_img,in_img,out_img);
    num_k += 3;
    fwrite(out_img,1,num_pixels,output);
    fclose(input);
    fclose(output);
  }
  return(0);
}
int thresh (UINT8* in_image, UINT8* out_image,int thresh_val)
{
  UINT8 pelA;
  UINT8 outA;
  int i;
  int tmp;
  int num_pix;
  int num_on = 0;

  num_pix = num_rows * num_cols;
  for (i=0;i<num_pix;i++) {
    pelA = in_image[i];
    if (pelA > thresh_val) {
    outA = 255;
    num_on++;
  } else {
    outA = 0;
  }
   out_image[i] = outA;
  }
  return num_on;
}
void img_med (UINT8* in_image, UINT8* out_image)
{
  UINT8 pelA, pelB, pelC, pelD, pelE, pelF, pelG, pelH, pelI;
  UINT8 tmp;
  UINT8 *pA, *pB, *pC, *pD, *pE, *pF, *pG, *pH, *pI;
  UINT8 outpel;
  UINT8* p0;
  int i;
  int num_pix;
  pA = &in_image[0];
  pB = &in_image[1];
  pC = &in_image[2];
```

```
  pD = &in_image[num_cols];
  pE = &in_image[num_cols+1];
  pF = &in_image[num_cols+2];
  pG = &in_image[2*num_cols];
  pH = &in_image[(2*num_cols)+1];
  pI = &in_image[(2*num_cols)+2];
  pO = &out_image[num_cols+1];
  num_pix = (num_rows-2) * (num_cols-2);
  for (i=0; i < num_pix; i++) {
    pelA = pA[i];
    pelB = pB[i];
    pelC = pC[i];
    pelD = pD[i];
    pelE = pE[i];
    pelF = pF[i];
    pelG = pG[i];
    pelH = pH[i];
    pelI = pI[i];
    PIX_SORT(pelB, pelC, tmp) ;
    PIX_SORT(pelE, pelF, tmp) ;
    PIX_SORT(pelH, pelI, tmp) ;
    PIX_SORT(pelA, pelB, tmp) ;
    PIX_SORT(pelD, pelE, tmp) ;
    PIX_SORT(pelG, pelH, tmp) ;
    PIX_SORT(pelB, pelC, tmp) ;
    PIX_SORT(pelE, pelF, tmp) ;
    PIX_SORT(pelH, pelI, tmp) ;
    PIX_SORT(pelA, pelD, tmp) ;
    PIX_SORT(pelF, pelI, tmp) ;
    PIX_SORT(pelE, pelH, tmp) ;
    PIX_SORT(pelD, pelG, tmp) ;
    PIX_SORT(pelB, pelE, tmp) ;
    PIX_SORT(pelC, pelF, tmp) ;
    PIX_SORT(pelE, pelH, tmp) ;
    PIX_SORT(pelE, pelC, tmp) ;
    PIX_SORT(pelG, pelE, tmp) ;
    PIX_SORT(pelE, pelC, tmp) ;
    outpel = pelE;
    pO[i] = outpel;
  }
}
void img_hp(UINT8* in_image, UINT8* out_image)
{
  UINT8 pelA, pelB, pelC, pelD, pelE, pelF, pelG, pelH, pelI;
  UINT8 *pA, *pB, *pC, *pD, *pE, *pF, *pG, *pH, *pI;
  INT32 sum, step1, step2,step3, term1, term2, term3, term4;
  UINT8 outpel;
  UINT8* pO;
  int i;
  int num_pix = (num_rows-2) * (num_cols-2);
  pA = &in_image[0];
  pB = &in_image[1];
  pC = &in_image[2];
  pD = &in_image[num_cols];
  pE = &in_image[num_cols+1];
```

```
  pF = &in_image[num_cols+2];
  pG = &in_image[2*num_cols];
  pH = &in_image[(2*num_cols)+1];
  pI = &in_image[(2*num_cols)+2];
  pO = &out_image[num_cols+1];
  for (i=0;i<num_pix;i++) {
    pelA = pA[i];
    pelB = pB[i];
    pelC = pC[i];
    pelD = pD[i];
    pelE = pE[i];
    pelF = pF[i];
    pelG = pG[i];
    pelH = pH[i];
    pelI = pI[i];
    sum = (pelE << 3) + (pelA + pelB + pelC + pelD + pelF + pelG + pelH + pelI);
    // Divide sum by 10
    step3 = sum >> 4;
    if (step3 > 255) step3 = 255;
    if (step3 < 0) step3 = 0;
    outpel = step3;
    pO[i] = outpel;
  }
}
void img_prew(UINT8* in_image, UINT8* out_image)
{
  UINT8 pelA, pelB, pelC, pelD, pelF, pelG, pelH, pelI;
  UINT8 *pA, *pB, *pC, *pD, *pF, *pG, *pH, *pI;
  INT32 sumH, sumV, sum;
  UINT8 outpel;
  UINT8* pO;
  int i;
  int num_pix;
  pA = &in_image[0];
  pB = &in_image[1];
  pC = &in_image[2];
  pD = &in_image[num_cols];
  pF = &in_image[num_cols+2];
  pG = &in_image[2*num_cols];
  pH = &in_image[(2*num_cols)+1];
  pI = &in_image[(2*num_cols)+2];
  pO = &out_image[num_cols+1];
  num_pix = (num_rows-2) * (num_cols-2);
  for (i=0;i<num_pix;i++) {
    pelA = pA[i];
    pelB = pB[i];
    pelC = pC[i];
    pelD = pD[i];
    pelF = pF[i];
    pelG = pG[i];
    pelH = pH[i];
    pelI = pI[i];
    sumH = (pelA + pelD + pelG) - (pelC + pelF + pelI);
    sumV = (pelA + pelB + pelC) - (pelG + pelH + pelI);
    if (sumH < 0) sumH = -sumH;
```

```
    if (sumV < 0) sumV = -sumV;
    sum = sumH + sumV;
    if (sum > 255) sum = 255;
    if (sum < 0) sum =0;
    outpel = sum;
    pO[i] = outpel;
  }
}
int num_on_pixels(UINT8* in_image)
{
  int i;
  int num_on = 0;
  int num_pix = (num_rows-2) * (num_cols-2);
  UINT8 *pel;
  pel = &in_image[num_cols+1];
  for (i = 0 ; i < num_pix ; i++ ) {
    if (pel[i] == 255) {
      num_on++;
    }
  }
  return num_on;
}
void img_erode1(UINT8* in_image, UINT8* out_image)
{
  UINT8 pel1, pel2, pel3, pel4, pel5, pel6, pel7, pel8, pel9;
  UINT8 *pA, *pB, *pC, *pD, *pE, *pF, *pG, *pH, *pI;
  UINT8 outpel;
  UINT8* pO;
  int i;
  int num_pix;
  pA = &in_image[0];
  pB = &in_image[1];
  pC = &in_image[2];
  pD = &in_image[num_cols];
  pE = &in_image[num_cols+1];
  pF = &in_image[num_cols+2];
  pG = &in_image[2*num_cols];
  pH = &in_image[(2*num_cols)+1];
  pI = &in_image[(2*num_cols)+2];
  pO = &out_image[num_cols+1];
  num_pix = (num_rows-2) * (num_cols-2);
  for (i=0;i<num_pix;i++) {
    pel1 = pA[i];
    pel2 = pB[i];
    pel3 = pC[i];
    pel4 = pD[i];
    pel5 = pF[i];
    pel6 = pG[i];
    pel7 = pH[i];
    pel8 = pI[i];
    pel9 = pI[i];
    outpel = pel1 & pel2 & pel3 & pel4 & pel5 & pel6 & pel7 & pel8 & pel9;
    pO[i] = outpel;
  }
}
```

```
void img_erode2(UINT8* in_image, UINT8* out_image)
{
  UINT8 pel1, pel2, pel3, pel4, pel5, pel6, pel7, pel8, pel9;
  UINT8 *pA, *pB, *pC, *pD, *pE, *pF, *pG, *pH, *pI;
  UINT8 outpel;
  UINT8* pO;
  int i;
  int num_pix;
  pA = &in_image[0];
  pB = &in_image[1];
  pC = &in_image[2];
  pD = &in_image[num_cols];
  pE = &in_image[num_cols+1];
  pF = &in_image[num_cols+2];
  pG = &in_image[2*num_cols];
  pH = &in_image[(2*num_cols)+1];
  pI = &in_image[(2*num_cols)+2];
  pO = &out_image[num_cols+1];
  num_pix = (num_rows-2) * (num_cols-2);
  for (i=0;i<num_pix;i++) {
    pel1 = pA[i];
    pel2 = pB[i];
    pel3 = pD[i];
    pel4 = pE[i];
    pel5 = pE[i];
    pel6 = pE[i];
    pel7 = pE[i];
    pel8 = pE[i];
    pel9 = pE[i];
    outpel = pel1 & pel2 & pel3 & pel4 & pel5 & pel6 & pel7 & pel8 & pel9;
    pO[i] = outpel;
  }
}
void img_out(UINT8* in_image, UINT8* orig_img, UINT8* out_image)
{
  int i;
  UINT8 pel, check;
  int row, col;
  UINT8 outVal;
  UINT8* pA;
  UINT8* pB;
  int row1 = -1000;
  int col1 = -1000;
  int row2 = -1000;
  int col2 = -1000;
  int num_pix;
  int diffrow1, diffcol1;
  int diffrow2, diffcol2;
  int have_mark1 = 0;
  int have_mark2 = 0;
  int in1, in2, inframe;
  pA = &orig_img[0];
  pB = &in_image[(FRAME*num_cols)+FRAME];
  num_pix = (num_rows * num_cols);
  row = 0;
```

```
  col = 0;
  for (i = 0 ; i < num_pix; i++) {
    diffrow1 = row - row1;
    diffrow2 = row - row2;
    diffcol1 = col - col1;
    diffcol2 = col - col2;
    if (diffrow1 < 0) diffrow1 = -diffrow1;
    if (diffrow2 < 0) diffrow2 = -diffrow2;
    if (diffcol1 < 0) diffcol1 = -diffcol1;
    if (diffcol2 < 0) diffcol2 = -diffcol2;
    in1 = ((diffrow1 < FRAME) && (diffcol1 < FRAME));
    in2 = ((diffrow2 < FRAME) && (diffcol2 < FRAME));
    inframe = in1 | in2;
    pel = pA[i];
    if (i < num_pix) {
      check = pB[i];
    } else {
      check = 0;
    }
    if (!inframe) {
      if (check == 255) {
        if (have_mark1 == 1) {
          have_mark2 = 1 ;
          row2 = row + FRAME;
          col2 = col + FRAME;
        }
        else {
          have_mark1 = 1;
          row1 = row + FRAME;
          col1 = col + FRAME;
        }
      }
    }
    if (inframe) {
      outVal = pel;
    } else {
      outVal = 0;
    }
    out_image[i] = outVal;
    col = col + 1;
    if (col == num_cols) {
      col = 0;
      row = row + 1;
    }
  }
}
```

Bibliography

[1] International Sematech, "International technology roadmap for semiconductors." http://public.itrs.net/, December 2004.

[2] D. Sylvester and H. Kaul, "Future performance challenges in nanometer design," in *Proceedings of the 38th conference on Design automation*, pp. 3–8, ACM Press, 2001.

[3] D. Sylvester and K. Keutzer, "Impact of small process geometries on microarchitectures in systems on a chip," in *Proceedings of the IEEE*, vol. 89, pp. 467–489, 2001.

[4] R. Puri, L. Stok, J. Cohn, D. Kung, D. Pan, D. Sylvester, A. Srivastava, and S. Kulkarni, "Pushing asic performance in a power envelope," in *Proceedings of the 40th Conference on Design Automation*, pp. 788–793, ACM Press, 2003.

[5] W. Maly, "Ic design in high-cost nanometer-technologies era," in *Proceedings of the 38th conference on Design automation*, pp. 9–14, ACM Press, 2001.

[6] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus ipc: the end of the road for conventional microarchitectures," pp. 248–259, 2000.

[7] S. Borkar, "Design challenges of technology scaling," *IEEE Micro*, vol. 19, no. 4, pp. 23–29, 1999.

[8] J. Hennessy, "The future of systems research," *Computer*, vol. 32, no. 8, pp. 27–33, 1999.

[9] K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns and platform-based design," in *IEEE Transactions of Computer-Aided Design of Integrated Circuits and Systems*, pp. 1523–1543, December 2000.

[10] Xilinx, Inc., *Xilinx Virtex-II Platform FPGAs: Complete Databook*. http://direct.xilinx.com/bvdocs/publications/ds031.pdf.

[11] G. Estrin, B. Bussell, R. Turn, and J. Bibb, "Parallel processing in a restructurable computer system," *IEEE Transactions on Electronic Computers*, pp. 747–755, 1963.

[12] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, 2002.

[13] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *IEEE Symposium on FPGAs for Custom Computing Machines* (K. L. Pocek and J. Arnold, eds.), (Los Alamitos, CA), pp. 12–21, IEEE Computer Society Press, 1997.

[14] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold, and M. Gokhale, "The NAPA adaptive processing architecture," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)* (K. L. Pocek and

J. M. Arnold, eds.), pp. 28–37, IEEE Computer Society, IEEE Computer Society Press, April 1998.

[15] R. D. Wittig and P. Chow, "OneChip: An FPGA processor with reconfigurable logic," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (J. Arnold and K. L. Pocek, eds.), (Napa, CA), pp. 126–135, Apr. 1996.

[16] T. Miyamori and K. Olukotun, "A quantitative analysis of reconfigurable coprocessors for multimedia applications," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines* (K. Pocek and J. Arnold, eds.), (Napa, CA), pp. 2–11, IEEE Computer Society, IEEE, April 1998.

[17] H. Singh, G. Lu, E. Filho, R. Maestre, M.-H. Lee, F. Kurdahi, and N. Bagherzadeh, "Morphosys: case study of a reconfigurable computing system targeting multimedia applications," in *Proceedings of the 37th conference on Design automation*, pp. 573–578, ACM Press, 2000.

[18] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor, "Piperench: A virtualized programmable datapath in 0.18 micron technolog," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 2002.

[19] C. Ebeling, D. C. Cronquist, P. Franklin, and S. Berg, "Mapping applications to the RaPiD configurable architecture," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (J. M. Arnold and K. L. Pocek, eds.), (Napa, CA), pp. 106–115, Apr. 1997.

[20] K. Ebcioglu and E. R. Altman, "Daisy: dynamic compilation for 100architectural compatibility," in *Proceedings of the 24th annual international symposium on Computer architecture*, pp. 26–37, ACM Press, 1997.

[21] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Proceedings of the international symposium on Code generation and optimization*, pp. 15–24, IEEE Computer Society, 2003.

[22] Y. Chou and J. P. Shen, "Instruction path coprocessors," in *Proceedings of the 27th annual international symposium on Computer architecture*, pp. 270–281, ACM Press, 2000.

[23] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta, "Performance characterization of a hardware mechanism for dynamic optimization," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pp. 16–27, IEEE Computer Society, 2001.

[24] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the fill unit to work: dynamic optimizations for trace cache microprocessors," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pp. 173–181, IEEE Computer Society Press, 1998.

[25] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W. mei W. Hmu, "A hardware mechanism for dynamic extraction and relayout of program hot spots," in *Proceedings of the 27th annual international symposium on Computer architecture*, pp. 59–70, ACM Press, 2000.

[26] J. B. C. A. D. Binu K. Mathew, Sally A. McKee, "Algorithmic foundations for a parallel vector access memory system," in *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pp. 156 – 165, 2000.

[27] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a smarter memory controller," in *Proceedings of the Fifth International Symposium on High Performance Computer Architecture*, pp. 70–79, 1999.

[28] C. Kozyrakis, "A media-enhanced vector architecture for embedded memory systems," Tech. Rep. CSD-99-1059, University of California at Berkeley, 1999.

[29] MIPS Technologies, http://www.mips.com, *MIPS32 4KEc Processor Core Datasheet*, November 2002.

[30] ARM Holdings PLC, http://www.arm.com, *ARM720T (Rev 3) Technical Reference Manual*, September 2000.

[31] T. J. Callahan and J. Wawrzynek, "Instruction-level parallelism for reconfigurable computing," in *Field-Programmable Logic: From FPGAs to Computing Paradigm* (R. W. Hartenstein and A. Keevallik, eds.), pp. 248–257, Springer-Verlag, Berlin, / 1998.

[32] F. R. C. Leiserson and J. Saxe, "Optimizing synchronous circuitry by retiming," in *The Proceedings of the 3rd Caltech Conference on VLSI*, pp. 87–116, 1983.

[33] G. Kane and J. Heinrich, *MIPS RISC Architecture.* Prentice Hall, 1992.

[34] D. C. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," Tech. Rep. CS-TR-1997-1342, 1997.

[35] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 177–189, ACM Press, 1983.

[36] T. Lindholm and F. Yellin, *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[37] L. S. Heath, S. V. Pemmaraju, and A. N. Trenk, "Stack and queue layouts of directed acyclic graphs: Part I," *SIAM Journal on Computing*, vol. 28, no. 4, pp. 1510–1539, 1999.

[38] M. Budiu and S. C. Goldstein, "Fast compilation for pipelined reconfigurable fabrics," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (S. Kaptanoglu and S. Trimberger, eds.), (Monterey, CA), pp. 195–205, ACM Press, 1999.

[39] "Gtl: The graph template library." http://infosun.fmi.uni-passau.de/GTL/.

[40] K. Khouri, G. Lakshminarayana, and N. Jha, "Impact: A highlevel synthesis system for low power control-flow intensive circuits," in *Proc. Design Automation & Test in Europe Conf.*, pp. 848–854, 1998.

[41] Intel, *Intel386DX Microprocessor 32-Bit CHMOS Microprocessor Datasheet*, December 1995.

[42] S. L. M. Junger and P. Mutzel, "Level planarity testing in linear time," tech. rep., Zentrum fur Angewandte Informatik Koln, Lehrstuhl Junger, 1999.

[43] G. De Micheli, *Synthesis and Optimization of Digital Circuits.* New York, NY: McGraw-Hill, 1994.

[44] T. C. Hu, "Parallel sequencing and assembly line problems," *Operations Research*, vol. 9, no. 6, pp. 841–848., 1961.

[45] G. Chaitin, "Register allocation and spilling via graph coloring," in *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, pp. 98–105, 1982.

[46] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *International Symposium on Microarchitecture*, pp. 330–335, 1997.

[47] "Texas instruments dsp developers resources." http://dspvillage.ti.com.

[48] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "Piperench: a co/processor for streaming multimedia acceleration," in *Proceedings of the 26th annual international symposium on Computer architecture*, pp. 28–39, IEEE Computer Society, 1999.

[49] T. Y. J. Ross Beveridge, Durga P. Panda, "Fort carson rsta data collection final report," tech. rep., Colorado State Technical Report., November 1993.

[50] W. J. Dally and A. Chang, "The role of custom design in asic chips," in *Design Automation Conference*, pp. 643–647, 2000.

[51] "Open cores." http://www.opencores.org.

[52] *Texas Instruments TMS320C28x DSP Users Manual.* http://www.ti.com.

[53] A. Devices, *Analog Devices ADSP-TS201S Users Manual.*

[54] "Chipworks analysis reports - xilinx xc3s200ft256afq spartan 3 structural analysis." http://www.chipworks.com/reports/flyers/Xilinx/.