The Microarchitecture of FPGA-Based Soft Processors

by

Peter Yiannacouras

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2005 by Peter Yiannacouras

Abstract

The Microarchitecture of FPGA-Based Soft Processors

Peter Yiannacouras

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2005

As more embedded systems are built using FPGA platforms, there is an increasing need to support processors in FPGAs. One option is the *soft processor*, a processor implemented in the reconfigurable logic of the FPGA. Commercial soft processors have been widely deployed, and hence we are motivated to understand their microarchitecture. We must re-evaluate microarchitecture in the soft processor context because an FPGA platform is significantly different than an ASIC platform. This dissertation presents an infrastructure for rapidly generating RTL models of soft processors, as well as a methodology for measuring their area, performance, and power. Using the automatically-generated soft processors we explore many interesting microarchitectural axes in the trade-off space. We also compare our designs to Altera's Nios II commercial soft processors and find that our automatically generated designs span the design space while remaining very competitive.

Contents

List of Tables **vii**

List of Figures **viii**

1 Introduction **1**

1.1 Research Goals 3

1.2 Organization 4

2 Background **5**

2.1 Basic Processor Architecture 5

2.2 The MIPS-I ISA 6

2.3 Application-Specific Instruction-set Processors (ASIPs) 9

2.4 FPGA Architecture 10

2.5 Industrial Soft Processors 12

2.6 Architectural Exploration Environments 14

2.6.1 Parameterized Cores 15

2.6.2 ADL-based Architecture Exploration Environments 16

2.7 Closely Related Work 17

2.8 Summary 20

3 The SPREE System **21**

3.1 Input: The Architecture Description 23

3.1.1 Describing the Instruction Set Architecture 23

3.1.2 Describing the Datapath . . . 26
3.2 The SPREE Component Library . . . 27
3.2.1 Selecting and Interchanging Components . . . 27
3.2.2 Creating and Describing Datapath Components . . . 28
3.3 Generating a Soft Processor . . . 31
3.3.1 Datapath Verification . . . 31
3.3.2 Datapath Instantiation . . . 33
3.3.3 Control Generation . . . 34
3.4 Practical Issues in Component Abstraction . . . 39
3.4.1 Combinational Loops . . . 40
3.4.2 False Paths . . . 40
3.4.3 Multi-cycle Paths . . . 41
3.5 Summary . . . 42

4 Experimental Framework **43**
4.1 Processor Verification . . . 43
4.2 FPGA Device and CAD Tools . . . 44
4.2.1 Determination of Confidence Across Seeds . . . 45
4.3 Metrics for Measuring Soft Processors . . . 47
4.4 Comparing with Altera Nios II Variations . . . 48
4.5 Benchmark Applications . . . 49
4.5.1 MiBench Benchmarks . . . 50
4.5.2 XiRisc Benchmarks . . . 51
4.5.3 RATES Benchmarks . . . 52
4.5.4 Freescale Benchmark . . . 52
4.6 ISA Reduction . . . 53
4.7 Summary . . . 54

5 Exploring Soft Processor Microarchitecture **55**
5.1 Comparison with Nios II Variations . . . 56

5.2 The Impact of Hardware vs Software Multiplication 58
5.3 The Impact of Shifter Implementation 62
5.4 The Impact of Pipelining 65
5.4.1 Branch Delay Slots in the 7-stage Pipeline 72
5.4.2 The Impact of Inter-Stage Forwarding Lines 75
5.5 Register Insertion for Clock Speed Enhancement 77
5.6 Architectures that Minimize Area 78
5.6.1 Fully Serialized ALU 79
5.6.2 Shared Shifting and Memory Alignment 81
5.7 Instruction Set Subsetting 81
5.8 Optimizations 84
5.8.1 Dual Word-Size Data Memory 85
5.8.2 Arithmetic Unit Result Splitting 86
5.8.3 Previous Stage Decode 87
5.8.4 Instruction-Independent Enable Signals 89
5.9 Application Specificity of Architectural Conclusions 90
5.10 CAD Setting Independence 93
5.11 Device Independence - Stratix vs Stratix II 95

6 Conclusions **97**
6.1 Contributions 98
6.2 Future Work 98

A SPREE System Details **100**
A.1 ISA Descriptions 100
A.2 Datapath Descriptions 107
A.2.1 2-stage Pipeline with Serial Shifter 108
A.2.2 3-stage Pipeline with Multiplier-based Shifter 109
A.2.3 5-stage Pipeline with LUT-based Shifter and Forwarding 112
A.3 The Library Entry 116

B Exploration Result Details **131**

B.1 CAD Settings 131

B.2 Data from exploration 133

B.2.1 Shifter Implementation, Multiply Support, and Pipeline Depth 133

B.2.2 Forwarding 137

B.2.3 Minimizing Area Processors 138

B.2.4 Subsetted Processors 139

B.2.5 Nios II 140

Bibliography **141**

List of Tables

2.1 MIPS-I instructions supported . . . 9
2.2 Comparison of Nios II Variants . . . 13
2.3 Comparison of ADL-based Architecture Exploration Environments . . . 17
3.1 GENOPs used in SPREE . . . 24
3.2 Components in the SPREE Component Library . . . 28
3.3 Control generation for pipelined and unpipelined datapths contrasted. . . . 34
4.1 Relative Areas of Stratix Blocks to LEs. . . . 48
4.2 Benchmark applications evaluated. . . . 49
B.1 Data for hardware multiply support over different shifters and pipelines. . . . 133
B.2 Data for hardware multiply support over different shifters and pipelines (cont'd). 134
B.3 Data for software multiply support over different shifters and pipelines. . . . 135
B.4 Data for software multiply support over different shifters and pipelines (cont'd). . 136
B.5 Measurements of pipelines with forwarding. . . . 137
B.6 Measurements of processors which minimize area . . . 138
B.7 ISA subsetting data on processors with full hardware multiply support. . . . 139
B.8 Area and performance of Nios II. . . . 140

List of Figures

2.1 The MIPS instruction format [39] 7

2.2 The Stratix Architecture 11

2.3 Nios II vs Nios Design Space 12

2.4 Nios II ISA Instruction Word Format 14

3.1 Overview of the SPREE system. 22

3.2 An overview of the SPREE RTL generator. 23

3.3 The MIPS SUB instruction shown as a dependence graph of GENOPs. 25

3.4 The MIPS SUB instruction described using C++ code. 26

3.5 A datapath description shown as an interconnection of components. 26

3.6 The Component Interface. 29

3.7 Library entry format. 30

3.8 Sample component description for a simplified ALU. 30

3.9 Different decode logic implementations used. 36

3.10 The pipeline stage model used by SPREE. 38

3.11 A combinational loop formed between two components. 40

3.12 A false path. 40

3.13 A multi-cycle path. 41

4.1 CAD flow overview. 44

4.2 Verification of Normal Distribution of Clock Frequency Measurements 46

4.3 Accuracy versus number of seeds for three processors. 47

5.1 Comparison of our generated designs vs the three Altera Nios II variations. . . . 56
5.2 Wall-clock-time vs area of processors with hardware multiplication support. . . . 60
5.3 Cycle count speedup of full hardware support for multiplication. 60
5.4 Energy/instruction for hardware vs software multiplication support. 61
5.5 A barrel shifter implemented using a multiplier 62
5.6 Average wall-clock-time vs area for different pipeline depths. 63
5.7 Energy per instruction across different pipelines. 64
5.8 Processor pipeline organizations studied. 65
5.9 Area across different pipeline depths. 67
5.10 Performance across different pipeline depths. 68
5.11 Wall-clock-time versus area across different pipeline depths. 70
5.12 Alternative 4-stage pipeline 70
5.13 Energy per instruction for the different pipeline depths. 71
5.14 Energy per cycle for the different pipeline depths. 72
5.15 Branch delay slot instruction separation. 73
5.16 Average wall-clock-time versus area space for all pipelines. 73
5.17 Clock frequency speedup after ignoring multiple delay slots. 75
5.18 Average wall-clock-time vs area for different forwarding lines. 76
5.19 Energy per instruction for three pipelines with forwarding. 77
5.20 The impact of RISE on a processor across the benchmark set 78
5.21 Average wall-clock-time vs area space including outliers 80
5.22 ISA usage across benchmark set 82
5.23 Area effect of subsetting on three architectures 82
5.24 Clock Speed effect of subsetting on three architectures 83
5.25 Impact of 8-bit store port on area and performance of different 2-stage pipelines. 85
5.26 Impact of result splitting on area and performance. 86
5.27 Impact of previous stage decode on performance 87
5.28 Impact of instruction-independent enable signals on area and performance 89
5.29 Performance of all processors on each benchmark. 90

5.30 Performance per unit area of all processors on each benchmark. 91

5.31 Effect of three different optimization focusses on area measurement. 93

5.32 Effect of three different optimization focusses on clock frequency. 94

5.33 Average wall-clock-time vs area for different pipeline depths on Stratix II. 95

A.1 Library entry format. 116

A.2 Sample component description for a simplified ALU. 116

Chapter 1

Introduction

With the rapidly rising cost and time-to-market of designing state-of-the-art ASICs, an increasing number of embedded systems are being built using Field-Programmable Gate Array (FPGA) platforms. Such systems often contain one or more embedded microprocessors which must also migrate to the FPGA platform to avoid the increased cost and latency of a multi-chip design. FPGA vendors have addressed this issue with two solutions: (i) incorporating one or more *hard processors* directly on the FPGA chip and surrounding it with FPGA fabric (eg., Xilinx's Virtex II Pro [61] and Altera's Excalibur [1]), and (ii) implementing *soft processors* which use the FPGA fabric itself (eg., Xilinx's MicroBlaze [60] and Altera's Nios [2]).

While the internal hard processors can be fast, small, and relatively cheap, they have several drawbacks. First, the number of hard processors included in the FPGA device may not match the number required by the application, leading to either too few or wasted hard processors. Second, the performance requirements of each processor in the application may not match those provided by the available FPGA-based hard processors. Third, due to the fixed location of each FPGA-based hard processor, it can be difficult to route between the processors and the custom logic. Finally, inclusion of one or more hard processors specializes the FPGA chip, impacting the resulting yield and narrowing the customer base for that product.

While a soft processor cannot easily match the performance/area/power consumption of a hard processor, soft processors do have several compelling advantages. Using a generic FPGA chip, a designer can implement the exact number of soft processors required by the application, and the FPGA CAD (computer-aided design) tools will automatically place them within the design to ease routing. Since it is implemented in configurable logic, a soft processor can be tuned by varying its implementation and complexity to match the exact requirements of an application. Finally, since no special hardware is required in the FPGA to support such processors, the FPGA remains a generic part with a large customer base. While these benefits have resulted in wide deployment of soft processors in FPGA-based embedded systems [41], the architecture of soft processors has yet to be studied in depth.

The microarchitecture of hard processors has been studied by many researchers and vendors for decades. However, the trade-offs for FPGA-based soft processors are significantly different than those implemented in full or semi-custom VLSI design flows [37, 38]: for example, on-chip memories are often faster than the clock speed of a soft processor's pipeline, and hard multipliers are area-efficient and fast compared to other functions implemented in configurable logic [5]. Because of this, the body of knowledge created by the decades of research into hard processors is not immediately transferable to soft processors, motivating us to revisit the microarchitectural design space in an FPGA context.

Furthermore, processor microarchitecture has traditionally been studied using high-level functional simulators that estimate area and performance due to the difficulty in varying designs at the logic layout level. In contrast, FPGA CAD tools allow us to quickly and accurately measure the exact speed, area, and power of the final placed and routed design for any soft processor. Hence we have the compelling opportunity to develop a complete and accurate understanding of soft processor microarchitecture.

With this knowledge, future tools could automatically navigate the soft processor design space and make intelligent application-specific architectural trade-offs based on

a full understanding of soft processor microarchitecture. One can envision a tool which can take an application and a set of design constraints as inputs, and automatically generate a customized soft processor which will satisfy those design constraints. The application would also be compiled specifically for execution on this customized soft processor. A crude version of this software system may be a design space iteration over all architectures, however, with the knowledge generated by this research, the tool can make some a priori decisions and at least reduce the design space.

1.1 Research Goals

The focus of this research is to develop an understanding of the soft processor microarchitectural design space. To this end, we set the following four goals:

1. To build a system for automatically-generating soft processors based on a simple but powerful input description.
2. To develop a methodology for comparing soft processor architectures.
3. To populate and analyze the soft processor design space and draw architectural conclusions.
4. To validate our results through comparison to an industrial soft processor family.

To satisfy the first goal, a system called *Soft Processor Rapid Exploration Environment (SPREE)* has been developed, which automatically generates an RTL (Register Transfer Level) description of a soft processor from text-based ISA (Instruction Set Architecture) and datapath descriptions. SPREE allows one to build a datapath at a high level, abstracting away from details such as component interfaces, and functional unit latencies. The user can assemble a datapath which can functionally support the given ISA, and SPREE will then automatically convert the description to an RTL form and generate the necessary control logic. FPGA CAD tools are then used to accurately measure area, clock frequency, and power of the resulting RTL designs, and RTL simulation

is used to verify correctness and measure the cycle counts of several embedded benchmark applications on these designs. The second research goal addresses issues such as accounting for area in FPGAs, choosing benchmarks, and selecting appropriate metrics for measurement. The third goal requires extensive use of the SPREE system to generate interesting processor designs and evaluate different microarchitectural axes. The fourth goal is required to achieve confidence in our architectural conclusions. By comparing the processors generated by SPREE to a family of industrial soft processors, the Altera Nios II variations, one can evaluate whether the conclusions drawn by SPREE suffer from prohibitive overheads. Through these measurements on several automatically generated processor architectures, we expect to gain a confident, complete, and accurate understanding of the soft processor architectural design space.

1.2 Organization

This dissertation is organized as follows. Chapter 2 gives a background in processor microarchitecture, and FPGA architecture, as well as summarizes the relevant research fields and research projects. Chapter 3 describes our SPREE system, including its interface and various problems left for manual intervention. Chapter 4 describes the experimental framework built around SPREE in order to facilitate our architectural exploration. Chapter 5 compares our automatically generated designs against an industrial soft processor for validation and then proceeds with an architectural exploration of various architectural parameters. Chapter 6 concludes by summarizing the dissertation, naming its contributions, and listing future extensions of this work.

Chapter 2

Background

In this chapter we discuss relevant background required to understand this work, and also summarize the research field of processor architectural exploration. The organization is as follows: Section 2.1 discusses computer architecture fundamentals and terminology; Section 2.2 describes the MIPS-I ISA (Instruction Set Architecture) which is our selected base ISA for our exploration; Section 2.3 discusses the field of Application Specific Instruction-set Processors; Section 2.4 gives an overview of FPGA architecture and the specific device employed in this work; Section 2.5 discusses soft processors and their presence in industry; Section 2.6 summarizes existing architectural exploration approaches; finally Section 2.7 describes research which is more directly related to our work.

2.1 Basic Processor Architecture

The field of processor architecture has been explored extensively for more than 40 years. Microprocessors have evolved from simple three to five stage pipelined cores to out-of-order speculative machines with intricate caching, and branch prediction schemes. In our work we do not consider such advanced topics in computer architecture, since the soft processor market currently targets only embedded applications which have much simpler architectures. Thus, we limit our exploration space to in-order issue pipelines with no caching and no branch prediction.

We now summarize basic processor architecture terminology, assuming the reader is largely familiar with basic computer architecture principles. If not, the necessary information on pipelines and computer architecture is available in Hennessy and Patterson [20]. A processor pipeline divides the execution of an instruction into steps and executes the different steps of several instructions simultaneously. Pipelining allows one to increase the clock frequency of the processor by shrinking the size of the steps, but pipelining also gives rise to **data hazards**. Data hazards occur when an instruction performs an operand fetch step before a preceding instruction has completed its operand write step to the same operand. The effect of this, if not guarded, would result in the read step reading an out-of-date operand value. This is referred to as a **read-after-write (RAW)** hazard, and the number of simultaneous steps that can potentially cause a hazard is the **hazard window**. To preserve program correctness, the architecture can delay the read step until the write step has completed (known as **interlocking** or **stalling**), or send the value to be written to the read step (known as **forwarding**). Both methods require **hazard detection logic** to signal the delay or the forwarding.

Pipelining also causes **branch penalties**, the penalty associated with incorrectly executing steps of instructions that should never have been fetched (which were fetched because the step which decides whether a branch is taken occurs later in the pipeline). A **branch delay slot instruction** refers to the instruction in the pipeline directly after a branch. The idea is that the branch delay slot instruction can be executed regardless of whether the branch is taken, allowing the processor to perform useful work as it computes the result of the branch condition.

2.2 The MIPS-I ISA

The MIPS (Microprocessors without Interlocking Pipe Stages) ISA was developed by John Hennessy et al. in 1981[24]. The goal of MIPS was to exploit the fact that large portions of the chip were being unused as traditional architectures with shallow pipelines

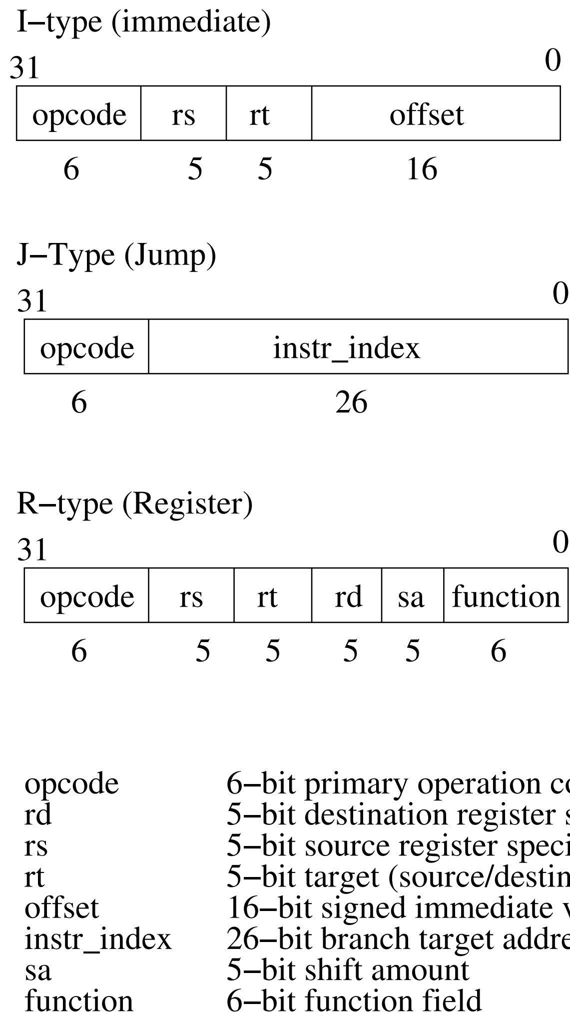


Figure 2.1: The MIPS instruction format [39]

stalled to execute complex instructions. The premise was that the stalling was hindering the pipeline from achieving faster performance. The MIPS ISA was designed ensuring all instructions can be executed in a single cycle thereby eliminating the stalls; inherently long instructions such as multiply and divide are handled specially in hardware as detailed below. Since then MIPS has become a popular instruction set widely supported by compilers and instruction simulators.

MIPS is a load-store RISC (Reduced Instruction Set Computer) instruction set with three operands, and its instruction format is shown in Figure 2.1. The operands can specify two locations in the register file for reading (`rs` and `rt`) and one for writing (usually `rd`, but can also be `rt`). There have been many revisions of MIPS since its inception named MIPS-I, MIPS-II, MIPS-III, MIPS-IV, MIPS-32/64—the newer revisions were augmented with more floating point support, trapping, and virtual memory instructions. The following properties of the MIPS ISA should be understood before proceeding.

Branch Delay Slots: MIPS has one branch delay slot instruction accompanying every branch or call instruction whether the branch is conditional or unconditional. Branch delay slots help decrease the branch penalty (the number of instructions incorrectly fetched before the processor could determine whether the branch was taken or not). This is done by imposing the rule that the instruction after the branch is always executed whether the branch is taken or not.

HI/LO registers: The MIPS multiply instruction produces a 64-bit result, which is stored in two special 32-bit registers called HI and LO storing the upper and lower halves of the 64-bit result respectively. These special registers are accessed via special instructions which can copy the values to the register file.

Nop instruction: MIPS does not contain an explicit null operation, or `nop` instruction. However, the instruction whose opcode is zero corresponds to a shift-left-by-zero instruction, which is effectively a null operation as it does not modify the processor state.

For this research MIPS-I was selected as our base ISA for two reasons. First, its wide support in instruction simulators and compilers avoids the time investment in having

Table 2.1: MIPS-I instructions supported

Type	Instruction
Branch	j, jal, jr, jalr, beq, bne, blez, bgtz, bltz, bgez
Memory	lb, lh, lw, lbu, lhu, sb, sh, sw
ALU	sll, srl, sra, sllv, srlv, srav, mfhi, mflo, mult, multu, addi, addiu, slti, sltiu, andi, ori, xori, lui, add, addu, sub, subu, and, or, xor, nor, slt, sltu

to create one. Second, its simplicity makes it an attractive starting point. We require a simple but robust ISA with clean instruction decoding to facilitate simpler control generation. We believe MIPS in general meets this requirement, and is much better than alternatives such as x86. MIPS-I is selected over the recent MIPS revisions since this work does not explore floating-point, virtual memory, nor exceptions.

For further simplicity, a reduced version of the MIPS-I ISA is used in our exploration. Table 2.1 lists the instruction supported in our exploration. We have removed all floating point instructions, unaligned loads/stores (`lwl`, `lwr`, `swl`, `swr`), writes to the HI/LO registers (`mtlo`, `mthi`), division (`div`, `divu`), and complex branches (`bgezal`, `bltzal`)—the motivation for removing these instructions is discussed in Section 4.6.

2.3 Application-Specific Instruction-set Processors (ASIPs)

Although this work is generalized across a benchmark set, the future of this research is to enable intelligent application-specific architectural decisions. The majority of architectural research has been targetted toward general purpose computing—processors designed to run several different applications. In another branch of computing, the embedded processor domain, processors are usually designed to run only one application. This property allows one to consider specializing the architecture to run that specific application well (at the expense of running other applications poorly), making it an *Application Specific Instruction-set Processor (ASIP)*. The large costs of designing and manufacturing an ASIP in the traditional ASIC flow may make this option unattractive

for designers whose application may change. With FPGAs, it is simple to replace one processor with another, as the device need only be reprogrammed. Thus, FPGAs provide the compelling advantage of allowing soft processors to become more aggressively customized to its application.

In recent years, there has been a growing interest in the architecture of ASIPs [28] and techniques for automatic customization such as compiler generation, and custom instructions [13]. Most notably, Tensilica [49] has been providing their Xtensa configurable processor commercially since 1999. The Xtensa processor can be automatically tuned by adding functional units, increasing parallelism using VLIW, and adding custom instructions (including vector operations and fused instructions). Much of this can be done automatically using AutoTIE [16], an infrastructure that automatically detects data structures in an application and creates register files for holding the data, and functional units for operating on them. These and many other techniques exist for creating application-specific processors. With the knowledge generated in this dissertation, one can more accurately apply such techniques based on a complete and accurate understanding of the design space.

2.4 FPGA Architecture

The architecture of FPGAs has become increasingly complex and no longer consists of a simple array of lookup tables (LUTs) and flip flops connected by programmable routing. FPGAs now include on-chip RAM blocks, and multipliers. FPGA devices and their architectures vary across device families and across vendors. In this work we focus on Altera's Stratix [5, 32] family and hence we discuss its architecture in more detail.

The Stratix FPGA is illustrated in Figure 2.2 and is comprised of a sea of logic elements (LEs) grouped in blocks of ten referred to as a logic array block (LAB). Each LE contains a 4-input lookup table and a flip flop. Stratix also contains fast multipliers, known as DSP blocks, which can perform 32x32 bit multiplies at 150 MHz, which is as fast

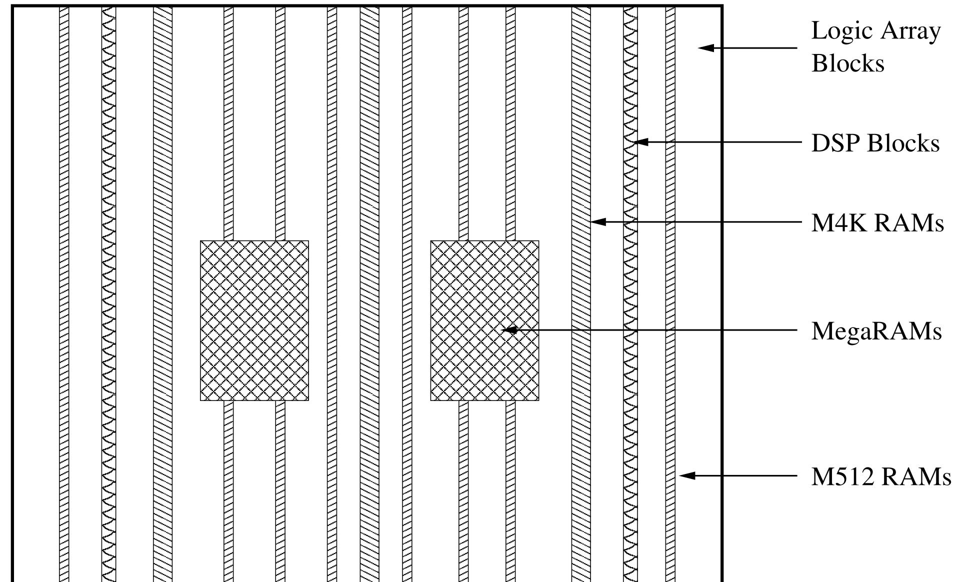


Figure 2.2: The Stratix Architecture

or faster than most soft processor designs—Altera's industrial Nios II soft processor has variations which run between 135MHz and 150 MHz. Finally, Stratix has three different sizes of block RAMs: M512 (512 bits), M4K (4096 bits), and Mega-RAM (65536 bytes). The speeds of each RAM are 320 MHz, 290 MHz, and 270 MHz respectively, making them also quite fast compared to logic. All RAMs are synchronous, meaning that they have registered inputs, and are dual ported allowing them to read/write to any two locations simultaneously. The only exception is the M512 which supports reading on one port and writing on the second port but can never do two reads or two writes simultaneously. Moreover, the ports have individually configurable data widths. More details on the architecture of Stratix can be found in the Stratix Device Handbook [5]. From the statistics above, it is clear that FPGAs are a much different platform than traditional hard processors, as highlighted by the difference in the relative speeds of logic versus multipliers and memory.

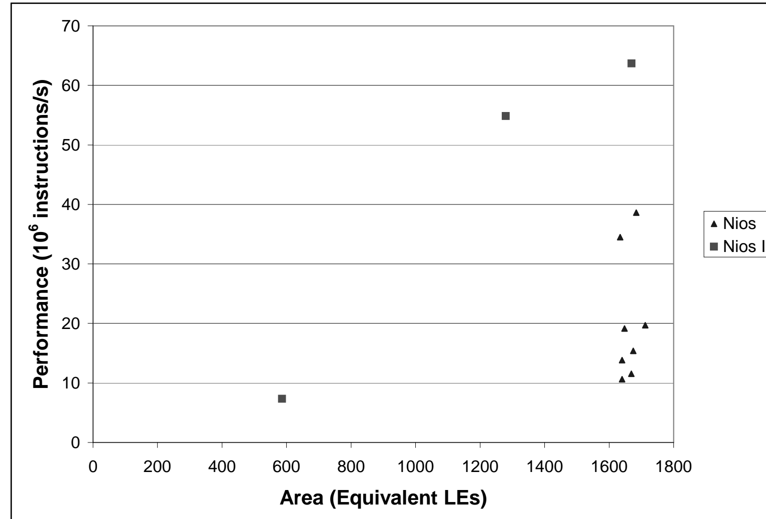


Figure 2.3: Nios II vs Nios Design Space

2.5 Industrial Soft Processors

In September 2000, Altera released the first commercial soft processor the Nios [2]. Shortly after this, Xilinx released their soft processor, the Microblaze [60]. Since then the popularity of soft processors has grown immensely and now 16% of programmable logic designs contain an embedded soft processor with 50% of those designs using Nios, and 40% using Microblaze [41], the remainder mostly comprised of freely available cores [42] or a custom-made processor [59]. Both industrial soft processors have undergone several revisions and Altera has recently released its second generation soft processor the Nios II. The Nios II architecture varies greatly from the original Nios; intelligent architectural decisions enabled Altera to create three Nios II cores which dominate all Nios versions. We have measured the area and benchmarked each of the Nios II and Nios cores and plotted the performance-area space shown in Figure 2.3—which agrees with figures released by Altera [3]. The figure shows that the three Nios II cores dominate the Nios cores in both size and performance. We have selected Nios II as the industrial core to validate our exploration against, and a description of its architecture follows.

Nios II has three mostly-unparameterized architectural variations: `Nios II/e`, a very small unpipelined 6-CPI processor with a serial shifter and software multiplication sup-

Table 2.2: Comparison of Nios II Variants

		Nios II/e	Nios II/s	Nios II/f
Performance	DMIPS/MHz	0.16	0.75	1.17
	Max DMIPS	28	120	200
	Clock (MHz)	150	135	135
Area	(LEs)	600	1300	1800
Pipeline		unpiped	5	6
Branch	Prediction	-	static	dynamic
ALU	Multiplier	-	3-cycle	1-cycle
	Divider	-	-	optional
	Shifter	serial	3-cycle	1-cycle

port; `Nios II/s`, a 5-stage pipeline with a multiplier-based barrel shifter, hardware multiplication, and an instruction cache; and `Nios II/f`, a large 6-stage pipeline with dynamic branch prediction, instruction and data caches, and an optional hardware divider. The three variations are contrasted in more detail in Table 2.2, which compares the three variations in terms of performance, area, and architecture. Performance is measured using the Dhrystone benchmark and is reported in DMIPS (Dhrystone Millions of Instructions per Second), and DMIPS/MHz, a clock frequency independent measurement of the same. Clearly Nios II/f outperforms the other two, however it is also largest in area as reported by the number of Stratix LEs used to implement each processor. In terms of pipelining, The Nios II/e is unpipelined, thereby requiring no branch prediction. The Nios II/s is a 5-stage pipeline with static branch prediction, while Nios II/f is a 6-stage pipeline with dynamic branch prediction. In both cases, the prediction scheme is not known. The Nios II/s uses the on-chip multipliers for performing both multiplication and shifting operations, but for both operations, the pipeline is stalled for 3 cycles. The Nios II/f also uses the on-chip multipliers for both multiplication and shifting and completes both in a single cycle. In addition, Nios II/f has the option of implementing a hardware divider unit which requires 4-66 cycles to compute its result.

All three Nios II variants use the same ISA known as the Nios II instruction set. It is very similar to the MIPS-I ISA discussed in Section 2.2; even its instruction format shown in Figure 2.4 is nearly identical to that of the MIPS-I shown in Figure 2.1, although the

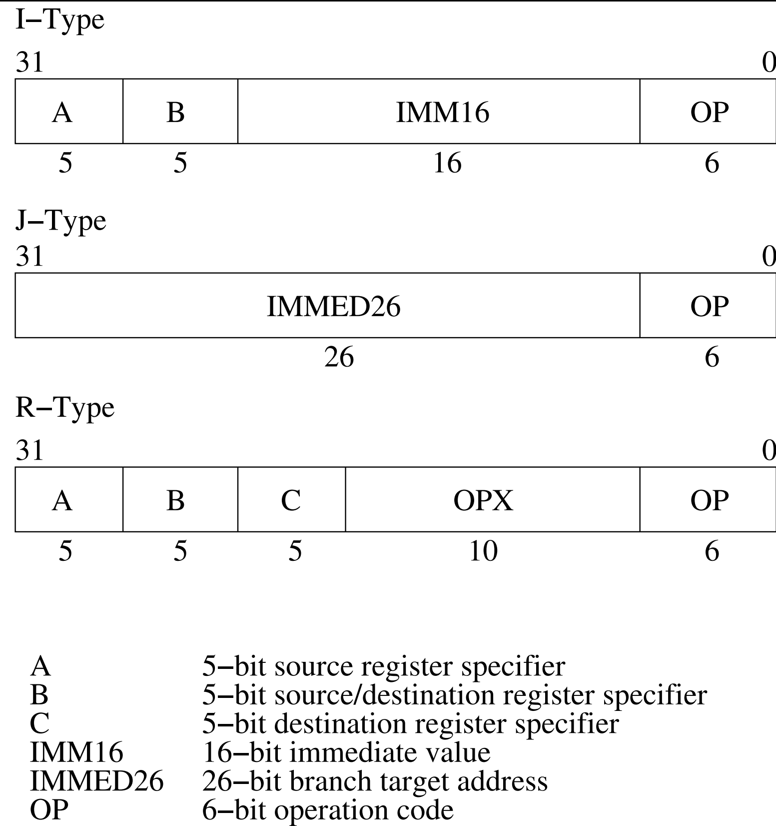


Figure 2.4: Nios II ISA Instruction Word Format

location of the fields in the instruction word have been rearranged. However it does have three significant differences: (i) Nios II does not have branch delay slot instructions; (ii) Nios II does not have HI/LO registers for multiply, instead the Nios II multiply writes its 32-bit result to the register file–separate instructions are used for calculating the upper and lower 32-bit results of a multiply; (iii) its `nop` instruction is implemented as an add instruction instead of a shift left instruction as in MIPS.

2.6 Architectural Exploration Environments

While industry architects have optimized commercial soft processors, as seen by Metzgen [37, 38], to the best of our knowledge a microarchitectural exploration of soft processors has never been conducted in the depth presented in this work. However, the architectural exploration of traditional hard processors has become a popular research

topic in recent years, and that research has produced numerous exploration environments that are primarily driven by one of two exploration methods: *parameterized cores* or *architecture description language* (ADLs). The two methods are described below.

2.6.1 Parameterized Cores

A parameterized processor core is a processor designed at the RTL level (Register Transfer Level) allowing for certain aspects of the architecture to be varied. The RTL design is expressed using an HDL (Hardware Description Language) such as Verilog or VHDL (Verilog Hardware Description Language) which contains parameters that can be tuned to alter the processor architecture in a manner intended by the original designer. The functionality of the processor is guaranteed by the original designer for many combination of parameter values. The advantage of parameterized cores is that the design is at the RTL level allowing for accurate measurements of speed, area, and power impact. However, few existing parameterized cores target FPGAs specifically, and all of them narrowly constrain the potential design space because of the hard-coded parameterization. Changing the ISA, timing, or control logic requires large-scale modification to the source code of the processor, making these parameterized cores unsuitable for a rapid and broad design space exploration.

We now discuss some examples of parameterized cores: the Opencores [42] website is dedicated to providing a collection of freely available IP cores to the general public. They have many processors available, some of them which target FPGAs, and most having tunable parameters for meeting design constraints which can be explored. However this exploration space is too narrow for our needs. LEON [15] is a VHDL description of a SPARC processor which has been used by Padmanabhann [43] to conduct semi-automatic system-level explorations (caches, TLBs, etc). The research we are conducting does not include memory hierarchy and other system-level issues and the narrow microarchitectural design space afforded by the LEON core (register window size, and optional multiply/divide/multiply-accumulate units) is inadequate for the breadth of ex-

ploration we intend to conduct. The XiRisc [33] is similarly a parameterized core written in VHDL. This DLX-like [20] processor allowed for various parameterizations including 2-way VLIW, 16/32-bit datapaths, and optional shifter, multiplier, divider, and multiply-accumulate units. While these results are interesting, the core does not easily allow for further interesting exploration. Fagin [12] performed a small architectural exploration specifically for FPGA-based processors by creating an unpipelined MIPS core processor and manually adding pipelining, and then forwarding. Again this exploration space was very narrow and only the area effect on Actel FPGAs was considered. Gschwind [18] produced a MIPS-based VHDL core for exploration, but no exploration results were extracted from it.

2.6.2 ADL-based Architecture Exploration Environments

An ADL (Architecture Description Language) is a language which completely specifies the architecture of the processor. A multitude of ADL-based architecture exploration environments have been proposed—a good summary of these is provided by Gries [17] and by Tomiyama [50]. The focus of these ADLs is to drive the creation of custom compilers, instruction set simulators, cycle accurate simulators, and tools for estimating area and power. Unfortunately these ADLs are often verbose and overly general, caused primarily by the need to simultaneously maintain instruction semantics (for instruction set simulation and compiler generation) and instruction behaviour (for cycle accurate simulation and RTL generation). Since we are only interested in the latter, our architecture specification can be simplified considerably. Furthermore, few ADLs provide a path to synthesis through RTL generation, and for those that do [27, 47, 56] the resulting RTL is often a very high-level description (for example, in SystemC), and therefore depends heavily on synthesis tools to optimize the design. For example, Mishra [27] provided RTL generation for the EXPRESSION ADL and discovered that the automatically generated RTL incurred 20% more area, 52% more power, and 28% slower clock frequency than an equivalent processor which was coded behaviourally for the purpose of cycle accurate

Table 2.3: Comparison of ADL-based Architecture Exploration Environments

Name	Source	Design focus[1]	Path to hardware	Tool generation
ASIP-Meister	[23, 29]	micro	yes (generated)	sim, comp
Chess/Check	[19]	micro	yes (generated)	sim, comp
CoCentric	[48]	system	yes (SystemC)	no
Expression	[27, 40]	micro	yes (generated)	sim, comp
LisaTek	[9, 46, 47]	micro	SystemC or control only	sim, asm
Mescal	[45, 55, 56]	system+micro	yes (generated)	sim, asm
PICO	[26]	system+ micro	yes (generated)	comp, sim

simulation. It seems reasonable to assume that had this comparison been against a design coded for efficient synthesis, the results would likely be significantly worse. Table 2.3 summarizes some ADL-based architectural exploration environments which provide a path to hardware. In an FPGA, using different hardware resources results in large trade-offs—hence the soft processor designer needs direct control of these decisions. For this reason, the RTL generation provided by these ADL-based environments are inadequate for our purposes. In addition, availability and learning curve were also considered when rejecting the option of using ADL-based environments. For example, the LISA language is a commercial product and publications report requiring one month to learn the language and design a single processor [47].

2.7 Closely Related Work

In this section, more closely related research is discussed and contrasted with the work in this dissertation. The following works are similar in theme, but for reasons detailed below remain inadequate for our purposes.

Mishra and Kejariwal augmented the EXPRESSION ADL to include RTL generation enabling synthesis-driven architectural exploration [27, 40]. The quality of these results were described above and were significantly worse than a model which was already a poor implementation of an industrial processor core—it was meant for simulation not

[1]Micro refers to the microarchitecture of a processor, namely the machine state, register file, and execution units, while system refers to system-level issues such as memory hierarchy, I/O, and operating system support.

synthesis. A small exploration was then conducted [40] for an FFT benchmark where the number of functional units were increased from 1 to 4, the number of stages in the multiplier unit were increased from 1 to 4, and `sin/cos` instructions were added to the instruction set. The exploration was not complete as it did not consider the entire processor (measurements were only made for the execution stage). Moreover, this exploration was performed for traditional hard processors, without any focus on FPGA-based processors.

The UNUM [11] system automatically generates microprocessor implementations where users can seamlessly swap components without explicit changes to the control logic. In this philosophy UNUM is identical to our system. The output of the UNUM system is a processor implemented in *Bluespec* [8], a behavioural synthesis language which can be translated to RTL. The drawback to this approach is that there is overhead to using the behavioural synthesis language which also abstracts away implementation details that are essential for efficient FPGA synthesis. This system is still being developed and has yet to be used in an architectural study.

The PEAS-III/ASIPMeister project [29] focuses on ISA design and hardware software co-design, and proposes a system which generates a synthesizable RTL description of a processor from a clock-based micro-operation description of each instruction. Unfortunately the design space is very limited as PEAS-III does not support hardware interlocking (the compiler must insert null operations) and does not allow multi-cycle functional units. Moreover, a small structural change to the architecture requires changes to the description of many instructions to produce the correct control logic. It is interesting to contrast the RTL generation of the PEAS-III system to ours: PEAS-III infers the datapath from the micro-operation instruction descriptions; in our system we infer the micro-operations of each instruction from the datapath, allowing the user to carefully design the datapath. This choice reflects our desire for efficient synthesis since we believe careful design of the datapath is crucial for efficiency. PEAS-III was used [23] to conduct a synthesis-driven exploration which explored changing the multiply/divide unit to

sequential (34-cycles), and then adding a MAC (multiply-accumulate) instruction. The results were compared for their area and clock frequency as reported by the synthesis tool.

Plavec[44] designed an open-source RTL description of a processor which implemented the Altera Nios ISA. During the design process, architecture and implementation decisions were made incrementally based on bottlenecks found along the way. FPGA nuances lead naturally to a 3-stage pipeline starting point, which was then successfully increased to a 4-stage pipeline to relieve a heavily dominating critical path. Further exploration of pipeline depth was hindered by the arduousness associated with control modification. Estimates were performed by simply adding registers to the datapath to quantify upper bounds on the frequency gain, and back of the envelope calculations were used to estimate cycle count increase. The open-source RTL model was used to perform simpler self-contained architectural modifications such as register window size, and register file size.

Finally, there has recently been a surge of interest in using FPGAs as a platform for performing processor and system-level architectural studies [22]. However, the goal of such work is to overcome the long simulation times associated with cycle-accurate simulation of large and complex processors. Often the role of the FPGA is to emulate cycle-accurate details or accelerate computation, whereas the role of FPGAs in our work is to serve as the final platform for the processor. Researchers in this field of FPGA-based system emulation often focus on an architectural novelty (for example transactional parallel systems [30], caching [34], vector-thread processors [25]) and build FPGA-based emulators to explore the space. None of this work focusses on area, clock frequency, or power; a functional FPGA model is all that is desired to extract cycle-to-cycle behaviour of the systems.

2.8 Summary

This chapter has defined basic computer architecture terminology as well as summarized the MIPS-I ISA and the subset of it used in our exploration. Since our research is motivated by the desire to meet stringent design constraints through application specific customizations, the field of architecting ASIPs has been briefly summarized. Since we focus on soft processors, we describe the architecture of FPGAs and our target FPGA device, as well as the industrial Nios II soft processor core we benchmark against. We have conducted a comparison of processor architecture exploration environments and have highlighted the motivations for designing a custom exploration environment. Finally, the chapter has surveyed and contrasted research which is closely related to that in this dissertation.

Chapter 3

The SPREE System

In this chapter, the Soft Processor Rapid Exploration Environment (SPREE) is described. The purpose of SPREE [62] is to facilitate the microarchitectural exploration of soft processors. Figure 3.1 depicts the role of SPREE in this research infrastructure, and provides an overview of its functionality: from an architecture description, the RTL generator emits synthesizable RTL which is used to measure area, performance, and power.

RTL generation is employed to gain very accurate measurements of a given soft processor design: FPGA CAD tools can be used to extract accurate area, clock frequency, and power measurements from an RTL description of a processor, and RTL simulators can be used to execute benchmark applications on the processor and measure exact cycle counts. We will use SPREE to rapidly generate synthesizable RTL descriptions for a wide variety of soft processor architectures, enabling us to thoroughly explore and understand the soft processor design space.

Some reduction in the breadth of our soft processor exploration was necessary to reduce the development time of SPREE. We consider simple, in-order issue processors that use only on-chip memory as main memory and hence have no cache. The memory on the FPGA is faster than a typical processor implementation eliminating the need for exploring caches. Moreover, the largest FPGA devices have more than one megabyte

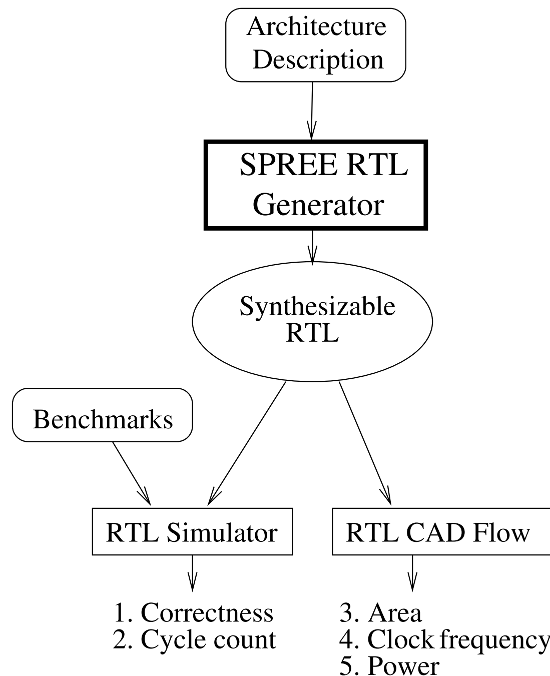


Figure 3.1: Overview of the SPREE system.

of on chip memory which is adequate for many applications (in the future we plan to broaden our application base to those requiring off-chip RAM and caches). We also do not yet include support for branch prediction, exceptions, or operating systems. Finally, in this research we do not modify the ISA or the compiler, with the exception of evaluating software vs hardware support for multiplication (due to the large impact of this aspect on cycle time and area).

The complete SPREE system is composed of the SPREE RTL Generator, and the SPREE Component Library. Figure 3.2 depicts a block-level diagram of the SPREE RTL Generator, which takes as input a description of the target ISA and the desired datapath, verifies that the datapath supports the ISA, instantiates the datapath, and then generates the corresponding control logic. The output is a complete and synthesizable RTL description (in Verilog) of a soft processor. The SPREE Component Library is a collection of hand-implemented components used to build a datapath. As shown in Figure 3.2, the library also interfaces with the RTL Generator. The subsequent sections will describe in more detail the architecture description input, the Component Library,

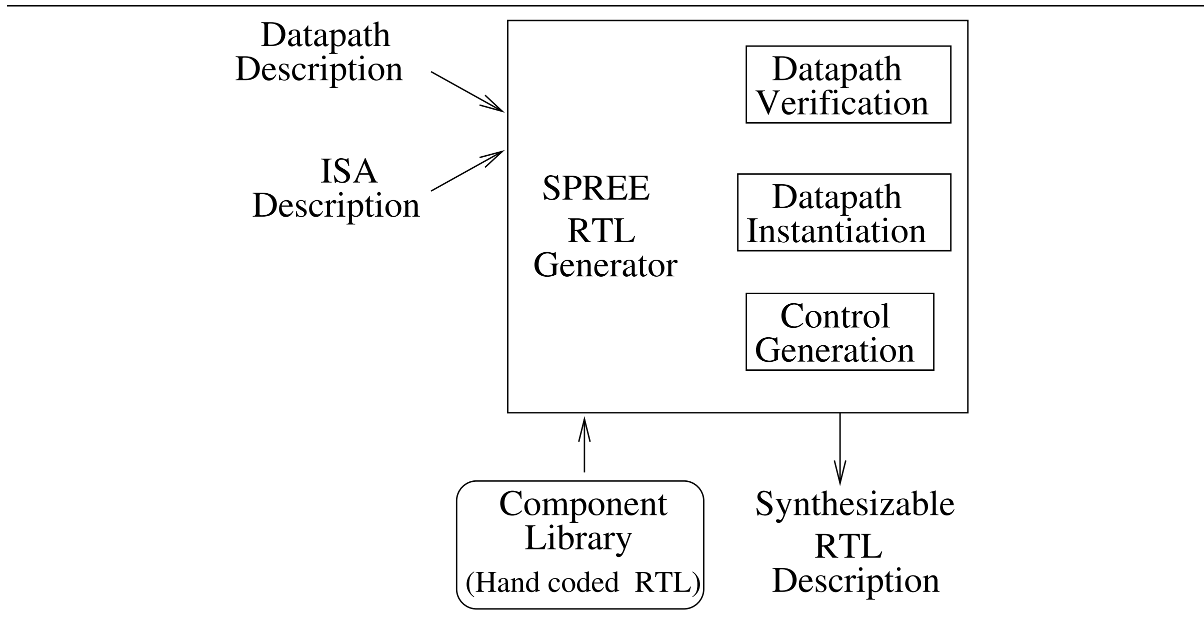


Figure 3.2: An overview of the SPREE RTL generator.

and the operations of the RTL Generator.

3.1 Input: The Architecture Description

The input to the SPREE system is the description of the desired processor, composed of textual descriptions of the target ISA and the processor datapath which implements that ISA. The functionality of each instruction in the ISA is described in a language which is also used to describe the functionality of components in the Component Library. The datapath is then described as an interconnection of these components. The following describes each of these in more detail.

3.1.1 Describing the Instruction Set Architecture

The instruction-set of the processor is described using a set of *generic operations (GENOPs)* which form a common language for describing the behaviour of a component and the semantics of an instruction. A GENOP is a small unit of functionality performed inside a typical microprocessor: examples of GENOPs include subtraction (`SUB`), program counter write (`PCWRITE`), load byte from memory (`LOADBYTE`), and register read (`REGREAD`). Each

Table 3.1: GENOPs used in SPREE

GENOP Name	Semantics	Description
NOP	o0=i0	Passes the input through unchanged
IFETCH	o0=opcode,o1=rs,o2=rt,o3=rd,...	Outputs complete instruction opcode
PCREAD	o0=program counter	Outputs the current PC value
REGREAD	o0=register[i0]	Reads register i0 from register file
HIREAD	o0=HI	Reads the MIPS HI register
LOREAD	o0=LO	Reads the MIPS LO register
REGWRITE	Write i0 into i1	Performs write to register file
PCWRITE	Write PC if i1 true	Branch on condition i1
PCWRITEUNCOND	Write PC	Jump to target
HIWRITE	Write HI register	Writes to the MIPS HI register
LOWRITE	Write LO register	Writes to the MIPS LO register
CONST	ox=x	Outputs the index on every port
SIGNEXT16	o0=sign extended i1	Sign extends i1 to 32-bits
BRANCHRESOLVE	o0=eq,o1=ne,o2=lez,o4=gtz,o5=gez	Computes branch flags
MERGE26LO	o0=hi 4-bits of i0,lo 26-bits of i1	Computes jump targets
ADD	o0=i0+i1	Performs addition
SUB	o0=i0-i1	Performs subtraction
SLT	o0=(i0<i1)	Compares i0 and i1
AND	o0=i0&i1	Bitwise and of i0 and i1
OR	o0=i0|i1	Bitwise or of i0 and i1
XOR	o0=i0^i1	Bitwise xor of i0 and i1
NOR	o0=~(i0|i1)	Bitwise nor of i0 and i1
STOREWORD	Store i0 in address i1	Performs 32-bit stores
STOREHALF	Store i0 in address i1	Performs 16-bit stores
STOREBYTE	Store i0 in address i1	Performs 8-bit stores
LOADWORD	o0=data in address i1	Performs 32-bit loads
LOADHALF	o0=data in address i1	Performs 16-bit loads
LOADBYTE	o0=data in address i1	Performs 8-bit loads
SHIFTLEFT	o0=i0<<i1	Performs left shift
SHIFTRIGHTLOGIC	o0=i0>>i1	Shifts right filling with zeros
SHIFTRIGHTARITH	o0=i0>>i1	Shifts right filling with hi bit of i0
MULT	o0,o1 = i0 * i1	Outputs upper and lower halves of product
DIV	o0= i0 div i1, o1= i0 mod i1	Returns divisor and remainder

GENOP has a predetermined interface using indexed input and output ports. For example, the SUB GENOP performs the subtraction function `o0=i0-i1` by taking input port 0 (`i0`), subtracting input port 1 (`i1`) from it and returning the result on output port 0 (`o0`). Thus, the SUB GENOP uses the two input ports 0 and 1, and the output 0 to perform the subtraction. The complete set of GENOPs used in SPREE is shown in Table 3.1 using the same notation.

Each instruction in the processor description is described in terms of a data dependence graph of GENOPs. An example of such a graph is shown in Figure 3.3 for the MIPS `subtract-signed` instruction. In the graph, the nodes are GENOPs and the edges rep-

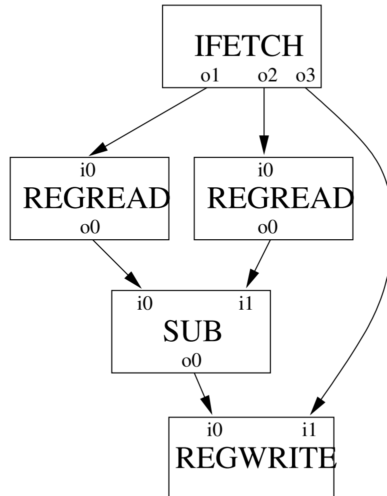


Figure 3.3: The MIPS SUB instruction (`sub rd,rs,rt`) shown as a dependence graph of GENOPs.

resent a flow of data from one GENOP to another. We impose the rule that *no GENOP can execute until all of its inputs are ready.* For a given instruction this graph shows the mandatory sequence of GENOP execution, although the datapath will determine the exact timing. From the example in Figure 3.3, the `IFETCH` GENOP has no inputs, and outputs the different fields of the instruction word. From the instruction word, the two source operand identifiers are passed to the register file (`REGREAD`) to read the source operand values. These values, are sent to the `SUB` GENOP and the difference, as well as the destination register from the instruction word, are connected to the register file for writing (`REGWRITE`).

The graph is described using C++ code as shown in Figure 3.4. New `GenOp` objects are created and links are made between the indexed ports of each GENOP using the `add_link` function. In the example, the instruction fetch (`IFETCH`), the subtraction (`SUB`) and the two register read GENOPs (`REGREAD`) are first dynamically allocated. Connections are made between ports 1 and 2 of `IFETCH` to port 0 of both `REGREADs` to read the `rs` and `rt` operands respectively. Their values are connected to the ports of the subtraction operation, and the result on port 0 is connected to the writeback operation `REGWRITE` on port 0. Finally, the index of the destination register from port 3 of `IFETCH` is connected

```
GenOp * ifetch=new GenOp(GENOP_IFETCH);
GenOp * op=new GenOp(GENOP_SUB);
GenOp * rs = new GenOp(GENOP_RFREAD);
GenOp * rt = new GenOp(GENOP_RFREAD);
add_link(ifetch,1,rs,0);
add_link(rs,0,op,0);
add_link(ifetch,2,rt,0);
add_link(rt,0,op,1);
GenOp * wb = new GenOp(GENOP_RFWRITE);
add_link(op,0,wb,0);
add_link(ifetch,3,wb,1);
```

Figure 3.4: The MIPS SUB instruction described using C++ code.

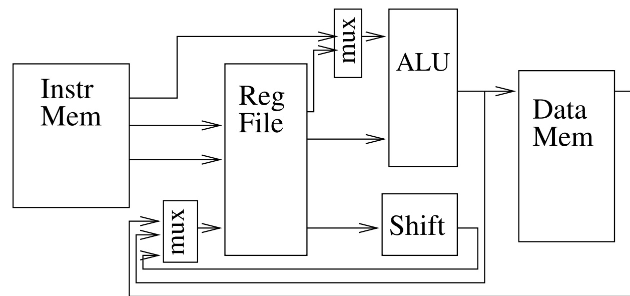


Figure 3.5: A datapath description shown as an interconnection of components.

to port 1 of `REGWRITE`. The rest of the supported MIPS-I ISA is described similarly.

3.1.2 Describing the Datapath

The datapath is described by listing the set of components and the interconnection between their physical ports. An example of a datapath is shown in Figure 3.5. In this simplified datapath, the instruction memory feeds the register file with the addresses of the two source operands to read. The register file feeds the ALU and shifting unit whose results are written back to the register file. Otherwise, the result of the ALU drives the effective address of the data memory location to read from or write to. If data memory is read, the loaded value is also written to the register file. The multiplexers shown in the datapath are optional, if not specified they will be automatically inferred when a port has multiple drivers as discussed in Section 3.3.2.

A processor architect can create any datapath that supports the specified ISA. The datapath must also include certain control components when necessary, for example:

pipeline registers, hazard detection units, and forwarding lines are available in the component library and must be used in an appropriate combination to ensure correct functionality of the processor. We hope to further automate the insertion of these components in the future.

The decision to use a structural architectural description in SPREE reflects our goal of efficient implementation. Structural descriptions provide users with the ability to manage the placement of all components including registers, and multiplexers in the datapath. This management is crucial for balancing the logic delay between registers to achieve fast clock speeds. By analyzing the critical path reported by the CAD tool, users can identify the components which limit the clock frequency and take one of three actions: (i) reducing the internal logic delay of a component, for example, making a unit complete in two cycles instead of one; (ii) moving some of the logic (such as such as multiplexers and sign-extenders) from the high delay path into neighbouring pipeline stages to reduce the amount of logic in the high delay path; (iii) adding non-pipelined registers in the high delay path causing a pipeline stall. The latter two of these actions depend critically on this ability to manually arrange the pipeline stages, referred to as *retiming*, which is difficult for modern synthesis tools because of the complexity in the logic for controlling the pipeline registers. Without a good ability to optimize delay we risk making incorrect conclusions based on poor implementations. For example, one might conclude that the addition of a component does not impact clock frequency because the impact is hidden by the overhead in a poorly designed pipeline. For this reason, architectural exploration in academia has traditionally neglected clock frequency considerations.

3.2 The SPREE Component Library

3.2.1 Selecting and Interchanging Components

The SPREE Component Library, which is used to build the datapath described above, stores the RTL code and interface descriptions of every available processor component.

Table 3.2: Components in the SPREE Component Library

Component Name	Description
addersub	An arithmetic unit used to perform add, sub, and slt instructions.
addersub_1	Same but has a 1-cycle latency.
branchresolve	Compares two register operands and computes the branch conditions.
const	Outputs a constant value.
data_mem	1-cycle latency data memory including necessary alignment logic.
data_mem_reg	Same but 2-cycle latency (a register before the alignment logic.
delay	A delay register that is always enabled.
forwarding_line	A forwarding line for avoiding data hazards in a pipeline.
hazard_detector	A data hazard detector which stalls the pipeline.
hi_reg	The MIPS HI register.
ifetch_unpiped	Unpipelined instruction fetch unit including program counter.
ifetch_pipe	Pipelined version of the same.
lo_reg	The MIPS LO register.
logic_unit	The logic unit, capable of executing and, or, xor, and nor.
lui	Shifts left by 16 to execute the MIPS load-upper-immediate instruction.
merge26lo	Concatenates the 4 upper bits and 26 lower bits of two oerands.
mul	performs 32-bit multiplication producing a 64-bit result.
mul_1, mul_2	1-cycle and 2-cycle latency multipliers.
mul_shift	performs multiplication and shifting (left, right logical, right arithmetic).
nop	The null component, behaves as a wire.
pcadder	A 30-bit adder used for computing branch targets.
pipereg	A pipelined register.
pipedelayreg	A non-pipelined register which stalls the pipeline when used.
reg_file	The register file which can perform two reads and one write simultaneously.
serialalu	A fully serialized ALU capable of performing arithmetic, logic, and shift operations.
shifter_LUT	A LUT-based barrel shifter.
shifter_LUT_1	A 1-cycle pipelined version of the same.
shifter_serial	An variable-latency serial shifter requiring as many cycles as the amount being shifted.
shifter_serial_datamem	A combined serial shifter and data memory unit where the shifter is used for memory alignment.
signext16	Sign extends the 16-bit input forming a 32-bit signed result.
zeroer	A wire which either passes the input when enabled or the value 0 when not.

Examples of these include register files, shifters, and ALUs. To evaluate different options for a given part of the datapath, a user can easily interchange components and regenerate the control logic. Therefore, whether a component is pipelined, combinational, or variable in latency, the automatically-generated control logic adapts to accommodate it. A list of the components in the SPREE Component Library is given in Table 3.2.

3.2.2 Creating and Describing Datapath Components

The Component Library described above can also be expanded by a user to include custom components. In this section we describe how these components are described

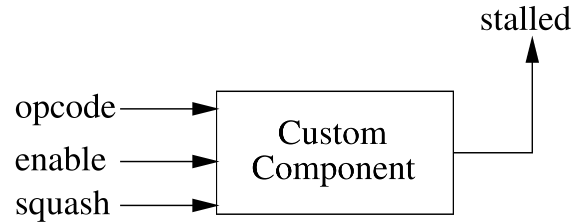


Figure 3.6: The Component Interface.

and imported into the Component Library. To create a datapath component, a user must perform three tasks: (i) provide the RTL description of the new component; (ii) ensure it adheres to SPREE's component interface rules; and (iii) describe its interface and functionality in a library entry. Each of these three steps will be considered in detail below.

The RTL Description The user must write the RTL description for the new component in Verilog. Since the goal is to produce an efficiently-synthesizable processor, users should consider the resources available on the target FPGA device and implement the component as efficiently as possible. The Verilog is placed in a text file where the top-level module is that specified by the component name.

The Component Interface The control generation in SPREE can support components with a wide variety of timing interfaces including zero cycle latency (or purely combinational), pipelined, multi-cycle unpipelined, and variable cycle latency. SPREE components are constrained to using the following *style* of control interface by requiring the presence of the following four control signals (when appropriate): *opcode*, *enable*, *squash*, and *stalled* signals as shown in Figure 3.6. The opcode signal is used to tell the component what operation to perform and is mandatory when a component can perform more than one operation. For example, this signal will instruct the arithmetic unit to perform an add or subtract. The enable signal indicates when the component should perform its operation and is thus used for scheduling. The squash signal is used only for pipeline registers whose contents must be destroyed when the stage is squashed. The

```
Module <base component name>_<version name> {
  File <Verilog source file>
  Parameter <param name> <parameter width>
  …
  Input <port name> <port width>
  …
  Output <port name> <port width>
  …
  Opcode <port name> <port width> [en <enable port name>] [squash <squash port name>] {
    <GENOP> <opcode value> <latency> <port mapping>
    …
  }
  …
  [clk]
  [resetn]
}
```

Figure 3.7: Library entry format.

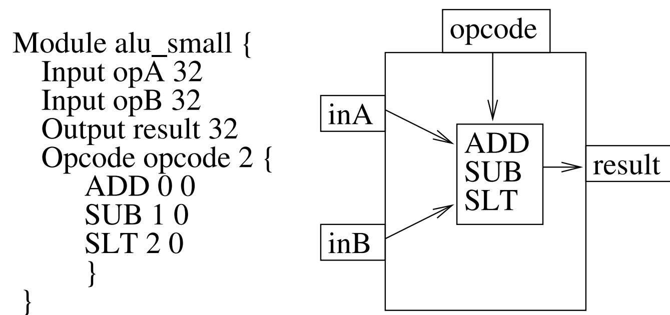


Figure 3.8: Sample component description for a simplified ALU. The ALU supports the GENOPs `ADD`, `SUB`, and `SLT`.

squash interface is exposed to the user to allow users to embed their own pipeline registers within a component. Finally the stalled signal is generated by variable-latency components to indicate that its operation has yet to complete. These four signal types are generic enough to accommodate a wide variety of components and must be used where appropriate when creating a custom component. The control logic generated by SPREE will drive the opcode, enable and squash signals and monitor the stalled lines; the user need only support and declare these signals. The names of each physical port in the component which connect to these signals can have arbitrary names, except for the stalled signal which must have the port name "stalled".

The Library Entry The interface and functionality of a component is communicated to SPREE through a text library entry. Figure 3.7 gives the complete format for a library entry which is detailed in Appendix A.3. Figure 3.8 shows a simplified library entry for a small ALU. The RTL interface to the component is described by the `Module` line, which defines the name of the Verilog module, and by the names and bit-widths of the RTL input and output ports which follow. The functionality of the component is described in the `Opcode` section which defines an opcode port (`opcode`). The fields inside the `Opcode` section describe the functionality of the component. Each line begins with the name of the supported operation and is proceeded by two integers: (i) the opcode port value that selects that operation, and (ii) the latency in cycles for the operation to complete (variable cycle latency is denoted with a negative latency). For example, the `ADD` function of the simple ALU specified in Figure 3.8 is selected by opcode 0 and has zero extra cycles of latency. The `Opcode` section can support an arbitrary number of GENOPs, which allows for versatile functional units, and a component can have an arbitrary number of opcode ports, which allows for parallelism within a component.

3.3 Generating a Soft Processor

From the above inputs (ISA description, datapath description, and Component Library), SPREE generates a complete Verilog RTL model of the desired processor. As shown in Figure 3.2 and described below, SPREE generates the processor in three phases: (i) datapath verification, (ii) datapath instantiation, and (iii) control generation.

3.3.1 Datapath Verification

A consistency problem arises as there are two separate inputs that describe the processor datapath and ISA: it is possible to assemble a datapath incapable of executing the described ISA. To prevent the generation of non-functional processors, SPREE must verify that the datapath indeed supports the ISA by ensuring the flow of data through the

datapath is analogous to the flow of data imposed by the instruction descriptions. In the ISA description, each instruction has an associated graph of GENOPs describing its functionality as discussed in Section 3.1.1. The datapath is described as an interconnection of components, but the components are described in terms of GENOPs as seen in Section 3.2.2. Therefore the datapath is also a graph of GENOPs. To verify that the datapath supports the ISA, SPREE must confirm that each of the instruction graphs are subgraphs of the datapath graph.

The algorithm for detecting subgraphs is a simultaneous traversal of both the instruction graph and the datapath graph beginning from the `IFETCH` node. Connections in the instruction graph are confirmed present in the datapath one at a time. A connection is confirmed if it connects between identical ports of the same GENOP, and all downstream connections are confirmed. For example, using the subtract instruction graph in Figure 3.3, we wish to confirm the link from `IFETCH` port o1 to `REGREAD` port i0. From the datapath graph, all `REGREAD` i0 ports fed by IFETCH port o1 are found. Of the possible candidates, one is chosen arbitrarily and is assumed to confirm that in the instruction graph. The algorithm then recursively confirms all output links of the `REGREAD` GENOP in the instruction graph using the same method. This occurs for all downstream links until the `REGWRITE` is confirmed which terminates the confirmation as it has no outputs. At this point, if all downstream instruction graph links are confirmed in the datapath, then the link is confirmed, if not, the algorithm will choose a different candidate. If none of the candidates confirm the instruction link, SPREE exits and reports an error citing the unconfirmed instruction link.

There are two factors which complicate this verification process: (i) the presence of NOP GENOPs in the datapath, and (ii) the indexed ports associated with GENOPs. NOPs appear in the datapath as multiplexers, and registers. Both of these components are guaranteed to function correctly by the control generation, therefore, they do not alter the functionality of the datapath. The ISA description does not contain NOPs, therefore, when the verification takes place, NOPs in the datapath are ignored. The indexed ports

cause the graph traversal to be slightly different than in traditional graphs. It is not enough to know that one node connects to another: rather, SPREE must verify how the nodes connect, specifically, with which ports. This is complicated further by the commutativity of the inputs of some GENOPs.

The datapath verification confirms that the components are connected in a manner which allows for correct instruction execution. However, timing considerations (such as data and control hazards in a pipeline) which depend on the architecture are not verified here. For example, forwarding lines and data hazard detection must be manually inserted to prevent pipelined processors from operating on out-of-date register contents. It is up to the user to engineer a datapath while being mindful of these considerations. If not, the simulation verification step described in Chapter 4 will catch the error.

3.3.2 Datapath Instantiation

From the input datapath description, we must generate an equivalent Verilog description. This task is relatively straight-forward since the connections between each component are known from the datapath description. However, to simplify the input, SPREE allows physical ports to be driven from multiple sources and then automatically inserts the logic to multiplex between the sources, and generates the corresponding select logic during the control generation phase.

Automatic multiplexer insertion simplifies the input, but must be carefully controlled directly by the user to prevent excessive overhead since multiplexers are often large units when implemented in FPGAs. SPREE will automatically create a new multiplexer for each instance of a port with multiple drivers. This may create some duplicate multiplexing, so it may be advantageous to share multiplexing logic to save area, however, it may also decrease clock frequency since the multiplexer may be placed further from some components. To share multiplexing logic, the user can direct a number of signals into a special *nop* component which acts as a wire and use the output of the wire to feed components which share the inputs. SPREE allows the designer to manage this tradeoff

Table 3.3: Control generation for pipelined and unpipelined datapths contrasted.

	Unpipelined	Pipelined
Decode logic	centralized	distributed
FSM generation	automatic	manual (use stall logic)
Stall logic	N/A	automatic
Squashes	none	automatic

without having to worry about multiplexer select signals.

3.3.2.1 Removal of Unnecessary Hardware

SPREE is equipped with the ability to remove unused hardware which aids in minimizing a processor design. While performing the verification described in Section 3.3.1, connections that are not used by the ISA can be identified by noting which edges in the datapath graph do not appear in any of the subgraphs of the instructions. These connections are marked and later removed by the RTL generator. Then, any components without connections are removed. This capability enables another feature of our research: ISA subsetting. Users can disable instructions not used by their application, and the generator will eliminate any wiring or hardware that is not required by any instruction. ISA subsetting will be further discussed in Section 5.7.

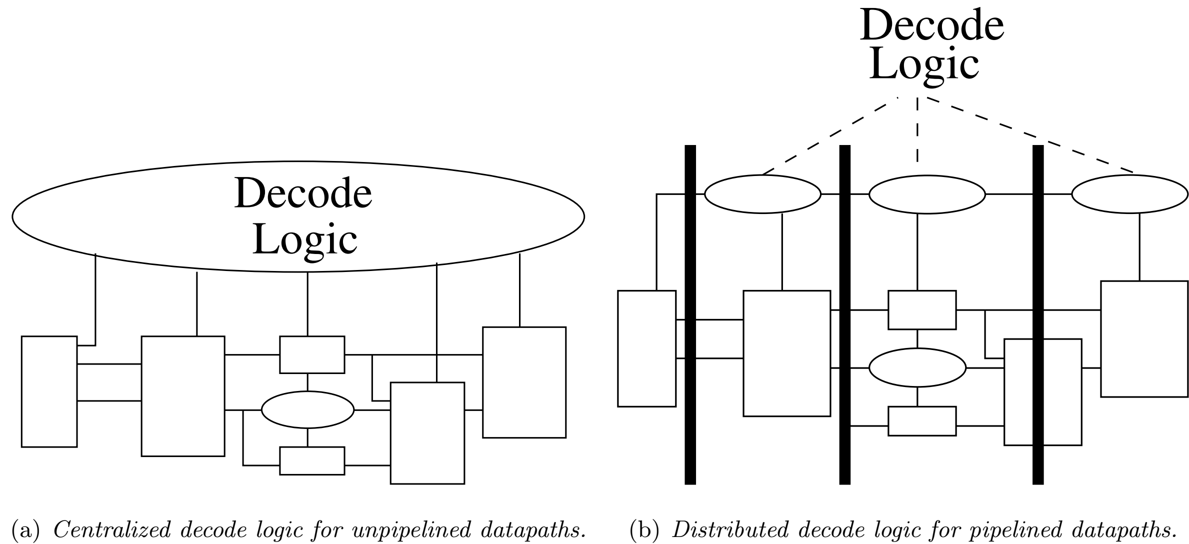
3.3.3 Control Generation

Once the datapath has been described and verified, SPREE automatically performs the laborious task of generating the logic to control the datapath's operation to correctly implement the ISA. From the datapath and ISA descriptions, SPREE is used to generate the decode logic, finite-state-machine (FSM), stall logic, and squashing logic required by the datapath. The generation of these is different for pipelined and unpipelined datapaths. Table 3.3 summarizes the differences in the control generation steps. The control generation for unpipelined datapaths generates the decode logic, and the FSM, while the control generation for pipelined datapaths generates the decode logic, stalling logic, and squashing logic. The details for both are described below.

3.3.3.1 Unpipelined Control Generation

The control logic for unpipelined datapaths must generate the opcode values which tell the components what operations to perform, and activate the enable signals when it is time to perform the specified operation. Also, the control logic needs to respond to stalled components and appropriately wait for their completion. These are done in two steps: (i) decode logic generation which calculates the opcode values, and (ii) FSM generation which activates the enable signals and responds to stalled components. Both of these steps are described below.

Decode Logic Generation The decode logic is responsible for computing the operation of each component from the instruction word and broadcasting it to the opcodes of the components. From the datapath and ISA description we know which operation is performed by each component for a given instruction, hence the opcode value is simply calculated as a boolean function from the instruction word. A single monolithic decode logic block is created which computes all opcodes for all components from the instruction word immediately after it has been fetched. The resulting opcode values are broadcast to all components as seen in Figure 3.9(a). Long routing paths are required between the centralized decode logic and the scattered components potentially causing decreased maximum clock frequency. An option exists within SPREE to detect components which will not be used in the first cycle of execution and put registers on the opcode values feeding those components in order to reduce the effect of these long connections. The impact of this option is discussed in Section 5.8.3. The advantage of the centralized decoding approach is reduced area as registers for propagating the instruction word or the opcode values are not required. Since an unpipelined datapath will likely be used for designs requiring minimal area, the routing problem is ignored and we continue to employ the generation of centralized decode logic for unpipelined datapaths.



(a) *Centralized decode logic for unpipelined datapaths.* (b) *Distributed decode logic for pipelined datapaths.*

Figure 3.9: Different decode logic implementations used.

FSM Generation A finite-state-machine (FSM) is required to drive the enable signals of the components and to respond to stall requests. SPREE generates this finite-state-machine from the datapath and ISA inputs. The unpipelined control generation performs automatic finite-state-machine creation by interpreting the latency of components as the number of wait states until completion, the procedure is as follows: Using the ISA description, the datapath is traversed with each instruction starting from the instruction fetch and determining which enable signals must be activated and when. A component enable is activated if the instruction uses the component, and all its inputs are ready. The first condition is calculated based on the instruction code, while the second condition requires analysis of the flow of data through the datapath and proper accounting of timing and stall signals. As the datapath is traversed with each instruction, the enable signal for each component is activated in the cycle of its latest arriving data input. From this traversal, the number of cycles required for each instruction is deduced, not including wait states for variable-latency components (for variable-latency components the state machine must wait for the stall signal to signal completion). The required number of states in the finite-state-machine is then the maximum number of cycles required for the longest instruction plus one extra state used as a dedicated wait state for all variable-

latency components. The activation of enables for each component is distributed among these states to form a simple minimal-state state-machine.

3.3.3.2 Pipelined Control Generation

The control logic for pipelined datapaths must also generate opcode values, activate the enable signals, and respond to stalled components. However these are done much differently than was done for the unpipelined datapaths. In addition, the pipelined control generation needs to squash instructions which were fetched but must not complete. All of these are done in three steps: (i) decode logic generation which calculates the opcode values, (ii) stalling logic generation which intervenes in the activation of the enable signals and responds to stalled components, and (iii) squashing logic generation which eliminates instructions in the pipeline.

Decode Logic Generation In pipelined architectures, SPREE distributes the decode logic to each stage by propagating the instruction word and inserting necessary decode logic locally to each stage, as illustrated in Figure 3.9(b). This alleviates the long routing paths of the centralized decode logic but increases area since more registers are inserted and the smaller decode logic is less amenable to logic optimization. The user can optionally locate the decode logic in the previous stage, which can have the effect of shortening a control-dominated critical path (further discussion and evaluation of the effect of this feature occurs in Section 5.8.3).

Stalling Logic Generation The pipelined control generation creates inter-stage stalling logic used to enforce the rule that when a component stalls in a pipeline, all components in the same or earlier stages as that component must also stall. This propagation of stalls is referred to as the *stall distribution network* and is automatically generated in SPREE. A stage is enabled (or not stalled) when none of its components are stalled and the stage following is not stalled. This condition is implemented using simple combinational logic and is used to propagate the stalls to all appropriate components. The determination

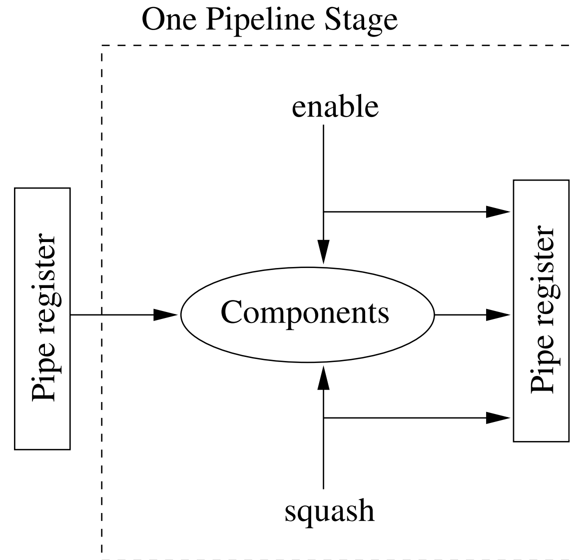


Figure 3.10: The pipeline stage model used by SPREE.

of which components to propagate the stall to is a much more difficult and requires the definition of a pipeline stage. Our pipeline stage model is shown in Figure 3.10, and includes all logic and the terminating registers of a stage and has one stage-wide enable signal and one stage-wide squash signal. The partitioning of the datapath components into the different pipeline stages is performed as follows: from the ISA and datapath descriptions, the datapath is traversed with each instruction and each components is placed in the stage when its inputs arrive (all inputs must arrive from the same stage in pipelined datapaths—this is enforced by SPREE, if violated SPREE exits citing the offending connection). When a pipelined unit is found (such as a pipeline register or the register file), the stage boundary is placed immediately following the component.

In addition to the inter-stage stalling logic generated above, each stage in the pipeline requires its own FSM to handle any multi-cycle unpipelined components within that stage. When a component within a stage requires more than one clock cycle to complete, a finite-state-machine is required to indicate its completion. For simplicity reasons, and due to its lack of usefulness, these FSMs are not automatically generated. Stalling in a pipeline is very much discouraged as indicated by the success of the MIPS architecture which aimed to eliminate the interlocking of pipeline stages from the processor. To

include a multi-cycle unpipelined component in a SPREE-generated processor, the user must implement the finite-state-machine within the RTL of the component and use the variable-latency interface to indicate when the component is stalled. Another option is to use the pipeline delay register `pipedelayreg` from the Component Library (see Table 3.2) to insert a non-pipelined cycle delay within a stage.

Squashing Logic Generation Squashing refers to the elimination of an instruction from the pipeline by changing it into a null instruction. Since an instruction occupies only one stage of the pipeline, to squash an instruction one must squash the stage it resides in. There are two conditions which can each cause a pipeline stage to be squashed. One condition occurs when the pipeline stalls, in which case the latest stalling stage forwards null operations to the next stage until it is not stalled. This logic is implemented simply by activating the squash for a stage if that stage is not stalled and the previous one is. The second condition occurs when a branch is mis-speculated and stages in the pipeline are executing instructions which should never have been fetched. The recovery from mis-speculated branches is fully automated; SPREE will automatically squash all instructions behind the branch omitting the branch delay slot instruction.

3.4 Practical Issues in Component Abstraction

SPREE's processor generation is simplified by using functional component abstraction, meaning only the functionality and interface need be known for each component. This abstraction reduces the complexity of SPREE, although some practical issues arise from naively connecting components without understanding their internal structure. There are three such issues, discussed below, which can affect both functionality and performance of a processor.

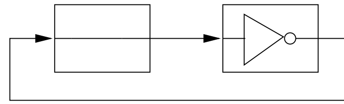


Figure 3.11: A combinational loop formed between two components.

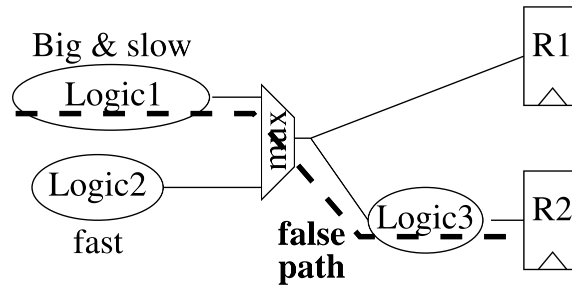


Figure 3.12: A false path.

3.4.1 Combinational Loops

A *combinational loop* is a feedback path purely through combinational logic. An example is shown in Figure 3.11 which illustrates a feedback path between two components forming a ring oscillator. If registers exist within the path, then this is a valid connection. If not, the components may oscillate radically as in the ring oscillator example shown. It is possible to create such situations using SPREE since the abstraction allows careless use of the components. Luckily this rarely occurs since data naturally flows forward through a processor pipeline, but bizarre datapaths with feedback is allowed and should be handled without neglecting combinational loops. Users must be aware of component internal structure and prevent combinational loops, and must always parse through the output of the CAD software which will issue warnings when a combinational loop is detected.

3.4.2 False Paths

Modern timing analysis tools produce conservative maximum operating frequencies if the functionality of the given circuit is not fully understood. During timing analysis, the longest possible combinational logic and routing path between any two registers dictates

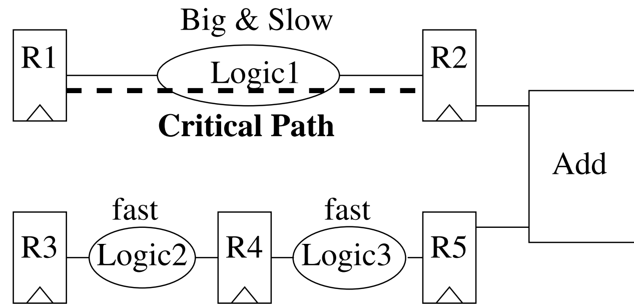


Figure 3.13: A multi-cycle path.

the maximum operating frequency. However, sometimes this path is never actually used, in which case, it is referred to as a *false path* [58]. In the example in Figure 3.12, the critical path is through `Logic1`, the multiplexer, `Logic3` and to R2. However, if R2 is enabled only when the the faster `Logic2` is used, then the path from `Logic1` to R2 will never actually be used. Since CAD tools are unable to detect such a situation, the timing analysis is conservative leading to a slower reported clock frequency. SPREE's component implementation abstraction can make it difficult to detect such cases—false paths can be detected by analyzing the critical path and reasoning about whether the reported path will ever be used. An example of a false path was observed in our work, in which an arithmetic unit whose result was broadcast both to the register file and data memory created a false path which limited the processor clock frequency. Details of this are discussed later in Section 5.8.2. In general, users can largely prevent this phenomenon during component design: if a component produces separate outputs, they should be each given separate output ports instead of being multiplexed into a single output port.

3.4.3 Multi-cycle Paths

Timing analysis tools can be conservative about the timing requirements of a given path leading to slower maximum operating frequencies. In determining the longest logic path between two registers, the timing analysis may not be aware that some logic paths need not be completed in a single cycle. For example, Figure 3.13 shows an example of a big and slow block of logic (`Logic1`) which is given multiple cycles to execute by the

controlling state machine. The timing analyzer, being unaware of this fact, may report a conservative maximum operating frequency under the assumption that such a logic block need be completed in a single cycle. Such situations can arise in the building of a datapath, in which case it is up to the user to ensure the critical path is not affected. For example, in an unpipelined processor, the destination register index can be provided early by the instruction word, while the data to be written to that destination register may come from data memory or some multi-cycle execution unit. The user should monitor the output report files from the CAD software and reason as to whether the critical path is truly one which must complete in a single cycle.

3.5 Summary

This chapter has described the Soft Processor Rapid Exploration Environment which facilitates our exploration of soft processor architecture through the employment of RTL generation from textual descriptions of a soft processor. The textual description is composed of an ISA and datapath description, each detailed in this chapter. The chapter also outlined the use of the Component Library, and the process for generating the complete RTL description, including the control logic. Finally, possible pitfalls in using the system were described. The next chapter describes the remainder of the infrastructure used in our exploration.

Chapter 4

Experimental Framework

Having described the design and implementation of the SPREE software system for generating soft processors in the previous chapter, this chapter will now describe the framework for measuring and comparing the soft processors it produces. We present a method for verifying the correctness of our soft processors, methods for employing FPGA CAD tools, a methodology for measuring and comparing soft processors (including a commercial soft processor), and the benchmark applications that are used to do so.

4.1 Processor Verification

The SPREE system verifies that the datapath is capable of executing the target ISA as described in Section 3.3.1. However, the generated control logic and the complete system functionality must also be verified. To do so, we have implemented trace-based verification by using a cycle-accurate industrial RTL simulator (Modelsim [36]) that generates a trace of all writes to the register file and memory as it executes an application. This trace is compared to one generated by MINT [53] (a MIPS instruction set simulator) and it is ensured that the traces match. All processors presented in this work have been verified to be functionally correct through this process.

Creating the traces from automatically generated RTL descriptions is somewhat cumbersome. First, the applications must be instrumented so that the end of the execution

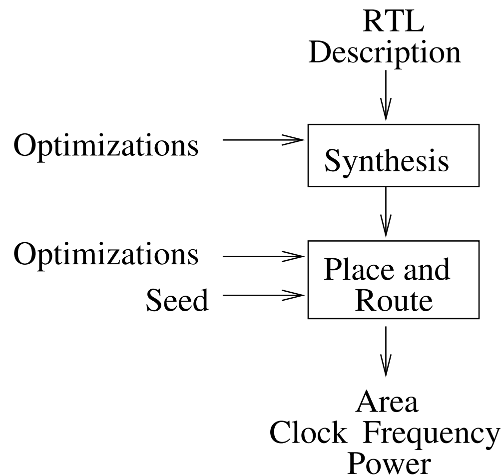


Figure 4.1: CAD flow overview. Optimizations and seed values add noise to the system affecting the final area, clock frequency, and power measurements, and hence must be carefully managed.

is detected by the RTL simulator. Special startup code initializes the stack pointer and jumps to the program `main` function. Upon returning from `main`, the program returns to special code that indicates the termination of the benchmark by writing the value `0xDEADDEAD` to data memory which is then caught by the simulator. Second, since the processors are automatically generated, the places to "tap in" to the architecture to generate the traces change with each processor. SPREE was augmented with the ability to automatically generate test benches which correctly tap in to the generated processor. Finally, to facilitate debugging of erroneous control logic, SPREE automatically generates debug outputs for observing the pipeline state including, the instruction word in each stage and the stage-wide enable and squash signals.

4.2 FPGA Device and CAD Tools

While SPREE itself operates independently of the target FPGA architecture, a particular FPGA device was selected for performing our FPGA-based exploration. The Component Library targets Altera Stratix [32] FPGAs. Quartus II v4.2 [4] CAD software is used for the synthesis, technology mapping, placement and routing of all designs to a Stratix EP1S40F780C5 device (a middle-sized device in the family, with the fastest speed grade)

using default optimization settings. From the generated report files, one can extract area, clock frequency, and power measurements from the placed-and-routed result.

It is important to understand that one must proceed carefully when using CAD tools to compare soft processors. Normally when an HDL design fails design constraints (as reported by the CAD software), there are three alternatives that avoid altering the design: (i) restructure the HDL code to encourage more efficient synthesis, (ii) use different optimization settings of the CAD tools, and (iii) perform seed sweeping—a technique which selects the best result among randomly-chosen starting placements. These three alternatives are design-independent techniques for coaxing a design into meeting specifications, and their existence illustrates the non-determinism inherent in combinatorial optimization applied in a practical context.

We have taken the following measures to counteract variation caused by the non-determinism inherent in CAD algorithms: (i) we have manually coded our component designs structurally to avoid the creation of inefficient logic from behavioural synthesis; (ii) we have experimented with optimization settings and ensured that our conclusions do not depend on them as seen in Section 5.10, and (iii) for the area and clock frequency of each soft processor design we determine the arithmetic mean across 10 seeds (different initial placements before placement and routing) so that we are 95% confident that our final reported value is within 2% of the true mean. The analysis of this confidence interval is detailed in the subsequent section.

4.2.1 Determination of Confidence Across Seeds

The randomness in the place-and-route phase of compilation is captured in an integer parameter known as a *seed*. Implementations are identical for the same seed value but vary non-deterministically across different seeds. Thus the measurements of area, power, and clock frequency may also vary non-deterministically. Power measurements are seen to exhibit only 0.13% variation across 10 seeds on three processors (2, 3, and 5-stage pipelines each with different shifter implementations), and area measurements vary ran-

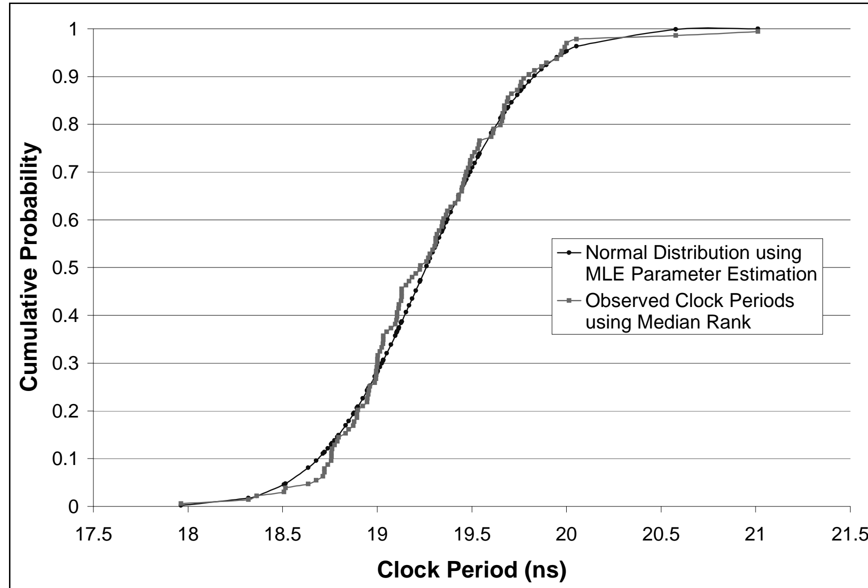


Figure 4.2: Verification of Normal Distribution of Clock Frequency Measurements

domly only when certain unsuccessful optimization techniques were applied and so this variation is ignored—the CAD software was directed to trade area for better speed and on average improved clock frequency by 2.4% at the expense of 47% increased area. As a result, power and area are considered to be independent of seed variations. The maximum operating clock frequency varied significantly across seeds, hence requiring analysis and quantification of confidence.

The maximum clock frequency measurements varied randomly according to a normal Gaussian distribution. This was verified through experimentation by sweeping through 122 seeds for Altera's Nios [2], which is enough seeds to produce a relatively smooth curve as shown in Figure 4.2. The figure verifies the normality of the noise by plotting the observed cumulative distribution function (CDF) of the data using the median rank method versus a CDF of a normal distribution whose parameters were estimated using maximum likelihood estimation (MLE). The result is shown in Figure 4.2 which indicates that a normal distribution is a good estimation of the distribution of the clock frequency.

With the distribution known, one can determine the number of seeds needed so that the arithmetic mean of the measurements is within some ϵ percent of the true mean (that obtained by taking the arithmetic mean across an infinite number of seeds) with

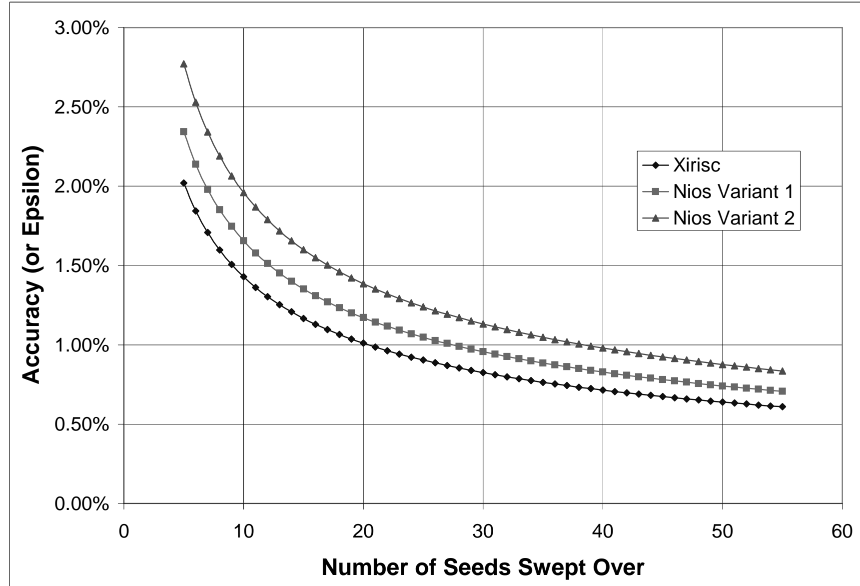


Figure 4.3: Accuracy versus number of seeds for three processors.

some specified degree of confidence. We performed this analysis on three processors: XiRisc [51], and two variations of Altera Nios. The graph shown in Figure 4.3 shows the value of ϵ as a function of the number of seeds for three different soft processors and with 95% confidence. The conclusion drawn from this work is that we will sweep over 10 seeds for each compilation in order to be 95% confident that we are accurate to 2% of the true mean.

4.3 Metrics for Measuring Soft Processors

To measure area, performance, and power of the soft processors, an appropriate set of specific metrics is required. For an FPGA, one typically measures area by counting the number of resources used. In the Stratix FPGA, the main resource is the *Logic Element* (LE), where each LE is composed of a 4-input *lookup table* (LUT) and a flip flop. Other resources, such as the hardware multiplier block and memory blocks can be converted into an equivalent number of LEs based on the relative areas of each in silicon. The relative area of these blocks was provided by Altera [10] and are listed in Table 4.1, allowing us to report area in terms of *equivalent LEs*. Note that the relative areas include the

Table 4.1: Relative Areas of Stratix Blocks to LEs.

Stratix Block	Relative Area
LE	1
DSP (9x9 bit)	CENSORED
M512 RAM	CENSORED
M4K RAM	CENSORED
Mega-RAM	CENSORED

programmable routing associated with each block.

To measure performance, we have chosen to report the wall-clock-time for execution of a collection of benchmark applications, since reporting clock frequency or instructions-per-cycle (IPC) alone can be misleading. To be precise, we multiply the minimum clock period (determined by the Quartus timing analyzer after routing) with the arithmetic mean of the cycles-per-instruction (CPI) across all benchmarks, and multiply that by the average number of instructions executed across all benchmarks. Calculating wall-clock-time in this way prevents a long-running benchmark from biasing our results.

The Quartus Power Play tool is used to produce a power measurement based on the switching activities of post-placed-and-routed nodes determined by simulating benchmark applications on a post-placed-and-routed netlist of a processor in Modelsim. We subtract out static power as it is solely a function of area, and we also subtract the power of the I/O pins since this power is significant and is more dependent on how the processor interfaces to off-chip resources than its microarchitecture. For each benchmark, we measure the dynamic energy per instruction and report the arithmetic mean of these across the benchmark set. The remainder of this document focusses only on dynamic energy even when “dynamic” is not specified.

4.4 Comparing with Altera Nios II Variations

Several measures were taken to ensure that comparison against the commercial Nios II soft processor is as fair as possible. Each of the Nios processors were generated with memory systems identical to those of our designs: two 64KB blocks of RAM for separate instruction and data memory. Caches are not included in our measurements, though some

Table 4.2: Benchmark applications evaluated.

Source	Benchmark	Reduced Input	Decreased Iterations	Removed I/O	Contains Multiply	Dyn. Instr. Counts
MiBench [52]	BITCNTS	x	x	x		26175
	CRC32	x		x		109414
	QSORT	x		x	x	42754
	SHA	x		x		34394
	STRINGSEARCH	x		x		88937
	FFT	x		x	x	242339
	DIJKSTRA	x		x	x	214408
	PATRICIA	x		x		84028
XiRisc [51]	BUBBLE_SORT			x		1824
	CRC			x		14353
	DES			x		1516
	FFT			x	x	1901
	FIR			x	x	822
	QUANT			x	x	2342
	IQUANT			x	x	1896
	TURBO			x		195914
	VLC			x		17860
Freescale [14]	DHRY		x	x	x	47564
RATES [54]	GOL	x	x	x		129750
	DCT	x	x	x	x	269953

logic required to support the caches will inevitably count towards the Nios II areas. The Nios II instruction set is very similar to the MIPS-I ISA with some minor modifications (for example, it has no branch delay slots)—hence Nios II and our generated processors are very similar in terms of ISA. Nios II supports exceptions and OS instructions, which are so far not supported in SPREE generated processors meaning that SPREE saves on area and complexity.[1] Finally, like Nios II, we also use GCC for compiling benchmark applications, though we did not modify any machine specific parameters nor alter the instruction scheduling. Despite these differences, we believe that comparisons between Nios II and our generated processors are relatively fair.

4.5 Benchmark Applications

We measure the performance of our soft processors using 20 embedded benchmark applications from four sources, which are summarized in Table 4.2. Some benchmark ap-

[1]Nios II architect Kerry Veenstra suggests that exception logic accounts for approximately 100 LEs worth of logic [57]

plications operate solely on integers, others on floating point values (although for now we use only software floating point emulation), some are compute intensive, others are control intensive. Table 4.2 also indicates any changes we have made to the application to support measurement, including reducing the size of the input to fit in on-chip memory, decreasing the number of iterations executed in the main loop, and removing file and other I/O since we do not yet support an operating system. More in-depth descriptions of each benchmark and its modifications follows; source code can be found on the SPREE webpage [62].

4.5.1 MiBench Benchmarks

The benchmarks below were extracted from the freely available MiBench benchmark suite developed at the University of Michigan. In addition to removing I/O, the input data sets were reduced in order to reduce RTL simulation time. Franjo Plavec [44] at the University of Toronto performed much of this task as he ported the benchmarks onto Altera's Nios [2] processor.

1. BITCNTS: From the Automotive section of MiBench, the application uses four different algorithms to count the number of bits in a 32-bit word and iterates over them. The benchmark initially generated random values to count the bits of, however, this was rather inconvenient for debugging purposes. Instead, the application was modified to iterate over an array of fixed integers.

2. CRC32: From the Telecomm section, the application computes the 32-bit CRC as in the frame check sequence in ADCCP (Advanced Data Communications Control Procedure). The input string was reduced in size.

3. QSORT: A quick sort algorithm which uses the standard C library `qsort` function to sort an array of strings. The number of strings to sort were reduced. This benchmark was also from the Automotive section.

4. sha: From the Security section, performs an NIST (National Institute of Standards and Technology) Secure Hash Algorithm on an input string that was reduced in size.

5. stringsearch: From the Office section, the application searches for a number of substrings in a number of longer strings. The number of strings to search through were reduced.

6. FFT_MI: From the Telecomm section, the benchmark performs forward and reverse fast fourier transforms on floating point values. The benchmark was stripped of command line parameters, and random number generation to facilitate debugging. Only one wave is transformed without performing the inverse transform, and its size is reduced.

7. dijkstra: From the Network section, this application performs dijkstra's algorithm. The number of nodes was reduced and I/O was removed.

8. patricia: From the Network section, the benchmark uses the Patricia trie library for performing longest prefix matching. The number of inputs and number of iterations were reduced. I/O was also removed.

4.5.2 XiRisc Benchmarks

The following benchmarks were developed as part of a suite used to measure the performance of their XiRisc processor [51]. These benchmarks required no (or very little) modification since the applications target a similar simulation environment for embedded systems.

9. bubble_sort: A simple N^2 bubble sort algorithm performed on a small array of integers. The array was increased from 10 to 20 elements.

10. crc: Computes the checksum value for a given string using an automatically generated CRC Table from Rocksoft.

11. DES: Applies a des encryption algorithm to an input text of 8 characters. The input string was modified since it was only 4 characters long.

12. FFT: Performs a fixed point fft on 16 samples.

13. FIR: An FIR filter with 8 taps performed on 10 data points.

14. QUANT: Quantization operation. This is used in jpeg and mpeg compression after the fdct step to reduce the information size of the samples.

15. IQUANT: the inverse operation of quant, used in jpeg and mpeg.

16. TURBO: An implementation of a Turbo decoder.

17. VLC: Variable length compression algorithm used after quant in jpeg and mpeg compression schemes.

4.5.3 RATES Benchmarks

The first two applications were taken from Lesley Shannon at the University of Toronto [54]. For these benchmarks, I/O and dynamic allocation was eliminated.

18. GOL: John Conway's Game of Life, a simulation of cells interacting with each other. The simulation size was reduced by decreasing the number of cells.

19. DCT: Performs a 2D-Direct Cosine Transform. Note this benchmark is a floating point application, though the floating point operations are implemented as software routines. The number of iterations was reduced to 1 and the size of the input block was reduced by half.

4.5.4 Freescale Benchmark

The last benchmark is the Dhrystone benchmark since it is often reported by soft processor vendors, including Altera.

20. DHRY: Executes the Dhrystone 2.1 benchmark. I/O calls were stripped and the number of iterations reduced.

4.6 ISA Reduction

Within the subset of MIPS I supported, there are certain instructions which are not (or seldomly) used and unnecessarily add overhead to the processor. We have removed these instructions from the supported ISA to remove this overhead. The first such instruction is the divide instruction. Our benchmark profiling has revealed that over all benchmarks in the suite 0.027% of dynamic instructions are divides. We have also noticed that the hardware divider is a very large and slow unit (a single-cycle divider is 1500 LEs and has a maximum operating clock frequency of 14 MHz). For these reasons, the divide instruction has been eliminated from the ISA and replaced with a software routine - the Altera Nios II makes the same design decision but offers an optional hardware divide in the fast variant.

MIPS has two instructions, `BGEZAL` and `BLTZAL`, which execute branch-on-condition-and-link operations. These instructions serve as conditional calls to subroutines. With this instruction, the architecture is forced to have a separate adder for the branch computation. By eliminating these instructions, we open the door to using the ALU to perform the branch address computation. For this reason, and since the instructions are not used in any benchmark, we have removed it.

In trying to accommodate the inherently long cycle latency of multiply/divide instructions, MIPS uses two dedicated registers, HI and LO, for storing the results of this unit. Special instructions are used to write these results to the register file, but there are also instructions for writing to these registers, `MTLO` and `MTHI`. This could allow the compiler to use these registers, however, the `MTLO` and `MTHI` instructions do not appear in any of our benchmarks. Furthermore, with our assumption of using on-chip memory, there is no performance gain in using these instructions versus a normal load and store.

Thus we have removed the support for this instruction saving 80 LEs and providing an average of 4.9% clock frequency improvement.

4.7 Summary

In this chapter, the remainder of our research infrastructure was described. The trace-based verification method used to guarantee the functionality of the automatically generated soft processors was described. A mid-sized Altera Stratix was chosen as the target device and the Quartus II 4.2 software was selected for synthesis, technology mapping, and place-and-route onto the device. Metrics were chosen to evaluate processors on their size, performance, and energy requirements, these metrics are: **e**quivalent LEs for area, **w**all-clock-time for performance, and **e**nergy/instruction for power. The details in generating the Nios II variations were revealed, and the benchmark suite was described in detail. Finally the further reduction of the MIPS-I ISA used in this research was described.

Chapter 5

Exploring Soft Processor Microarchitecture

In this chapter, we perform an investigation into the microarchitectural trade-offs of soft-processors using the SPREE exploration environment. We first validate our infrastructure by showing that the generated designs are comparable to the highly-optimized Nios II commercial soft processor variations. We then investigate in detail the following aspects of soft processor microarchitecture: hardware vs software multiplication—whether the architecture should contain hardware support for performing multiply instructions; shifter implementations—how one should implement the shifter, since shifting logic can be expensive in FPGA fabrics; pipelining—we look at pipeline organization, measure different pipeline depths, and experiment with inter-stage forwarding logic. SPREE is also used to explore more radical architectures, for example, a fully serialized ALU. An investigation is then performed into generating ASIPs by performing architectural ISA subsetting per benchmark, which is followed by several optimizations which were discovered during our experiments. Finally, a quantitative analysis of the following three factors on our architectural conclusions is performed: their application specificity, their fidelity across CAD settings, and their fidelity across FPGA devices.

5.1 Comparison with Nios II Variations

Each Nios II variation (the economy, standard, and fast as described in Section 2.5) was synthesized, placed, and routed by the CAD tools and was benchmarked using the complete benchmark set. Similarly, the complete set of generated processors (MIPS-I based processors with different pipelines, shifters and multiplication support), were measured. The area in equivalent LEs and average wall-clock-time was extracted from all processors, each pair forming the x and y coordinates respectively of the design point in a scatter plot corresponding to that processor.

Figure 5.1 shows the scatter plot, where our generated designs are compared to the three Nios II variations. The three points in the space for Nios II lie furthest left for `Nios II/e` (smallest area, lowest performance), furthest right for `Nios II/f` (largest area, highest performance), and in between for `Nios II/s`. The figure shows that our generated designs span the design space, and that one of our generated designs is even smaller and faster than the `Nios II/s`—hence we examine that processor in greater detail.

The processor of interest is an 80MHz 3-stage pipelined processor, which is 9% smaller and 11% faster in wall-clock-time than the `Nios II/s`, suggesting that the extra area used to deepen the `Nios II/s` pipeline succeeded in increasing the frequency, but brought overall wall-clock-time down. The generated processor has full inter-stage forwarding support which prevent data hazard delays, and suffers no branching penalty. The pipeline stalls only on load instructions (which must await the value being fetched from data memory) and on shift and multiply instructions (which complete in two cycles instead of one, since both are large functional units). In contrast, the Nios II/s is a five stage pipeline with more frequent hazards and larger branch penalties (recall Nios II does not have branch delay slots), and the multiplication and shift operations stall the pipeline for 3 cycles compared to our two. The cycles-per-instruction (CPI) of this processor is 1.36 whereas the CPIs of `Nios II/s` and `Nios II/f` are 2.36 and 1.97 respectively. However, this large gap in CPI is countered by a large gap in clock frequency: `Nios II/s`

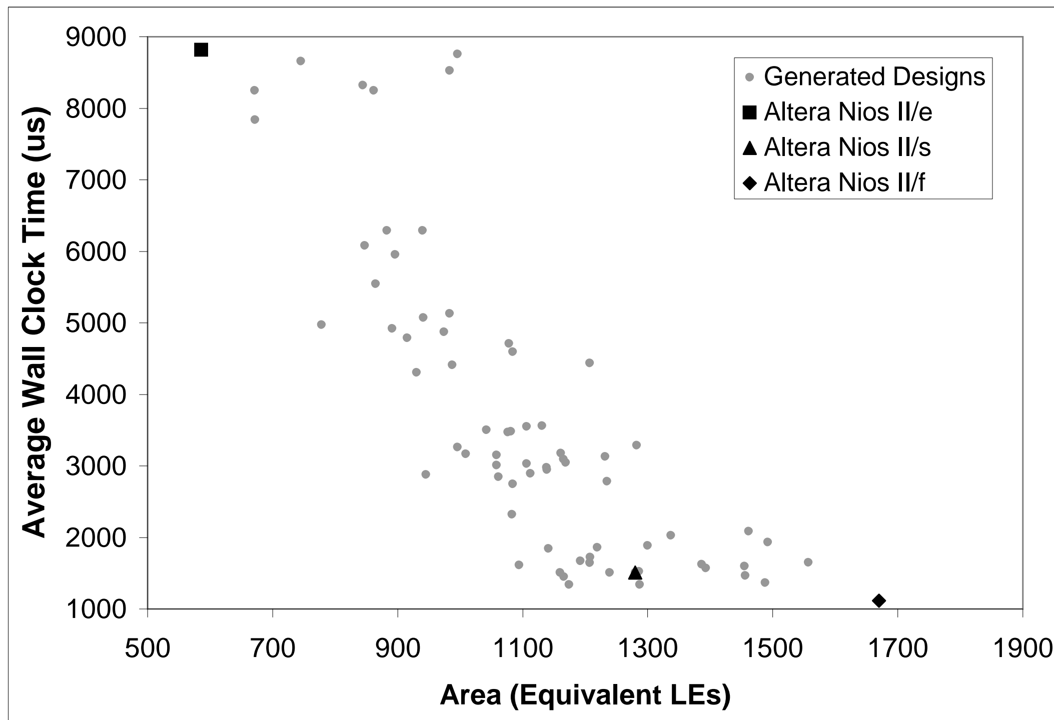


Figure 5.1: Comparison of our generated designs vs the three Altera Nios II variations.

and `Nios II/f` achieve clock speeds of 120 MHz and 135 MHz respectively, while the generated processor has a clock frequency of only 80MHz. These results demonstrate the importance of evaluating wall-clock-time over clock frequency or CPI alone, and show that faster frequency is not always better. A similar conclusion was drawn for the original Nios by Plavec [44], who matched the wall-clock-time of the original Nios [2] by targetting lower cycle counts instead of higher frequencies.

Our smallest generated processor is within 15% of the area of `Nios II/e`, but is also 11% faster (in wall-clock-time). The area difference of 85 LEs can be attributed to overhead in the generated designs compared to the hand-optimized `Nios II/e`. Specifically, these overheads are caused by two factors: (i) the lack of optimization in the generated control logic since SPREE supports only 50 instructions but decodes enough bits to hold 158; and (ii) the lack of optimizations performed across components—a component can be simplified if information is known about its inputs, but SPREE components were designed for the general case. With respect to performance, Altera reports that `Nios II/e`

typically requires 6 cycles per instruction, while our smallest processor typically requires 2-3 cycles per instruction. Although our design has less than half the CPI of the `Nios II/e`, our design also has half the clock frequency (82MHz for our design, 159 MHz for Nios II/e), reducing the CPI benefit to an 11% net win in wall-clock-time for our design. Note that comparing our generated processors against Nios II/e is more fair than against the other variations for two reasons: (i) it also does not have support for caches (ii) it has much simpler exception handling being an unpipelined processor.

Bearing in mind the differences between Nios II and our processors, it is not our goal to draw architectural conclusions from a comparison against Nios II. Rather, we see that the generator can indeed populate the design space while remaining relatively competitive with commercial, hand-optimized soft processors.

5.2 The Impact of Hardware vs Software Multiplication

Whether multiplication is supported in hardware or software can greatly affect the area, performance, and power of a soft processor. For this reason, the `Nios II/e` has no hardware support while the other two Nios II variations have full hardware support. There are many variations of hardware multiplication support which trade off area for cycle time. For example, the original Altera Nios supported a hardware instruction which performed a partial multiplication which can be used in a software routine to perform a full 32-bit multiply much faster than the typical software routine which uses a sequence of shift and add instructions. In this work, we do not consider such hybrid implementations, we focus only on either full or no hardware multiplication support.

Implementing full hardware multiplication support on newer FPGAs is simplified by the recent addition of dedicated multipliers in the FPGA fabric. We conducted a preliminary investigation into implementing multiplication in the dedicated multipliers versus in the programmable fabric using LUTs. Our experiments showed that a 2-stage pipelined processor with the multiplier implemented using lookup tables required 125% more area,

25% more energy per cycle, and reduced clock frequency by 10x over the same processor with multiplication implemented in the dedicated multipliers. This shows that implementing such a multiplier should be accomplished using the dedicated multipliers. Thus, our exploration of multiplication support is binary: a purely software approach with no hardware support (small and slow), and a purely hardware approach with minimal cycle latency (big and fast) where the software support used is the default C multiplication routine provided with GCC and is the same used for Nios II. We implement both versions of multiplication support on a variety of architectures, and compare the results in order to quantify and bound the tradeoff space.

Figure 5.2 illustrates the trade-off between area and wall-clock-time for multiplication support. In the figure we plot the Nios II variations, as well as the collection of our generated designs each with either full hardware support for multiplication or software-only multiplication. In terms of area, removing the multiplication saves 230 equivalent LEs, or approximately one fifth the area of the processor. However, in some of the designs, the multiplier is also used to perform shift operations as recommended by Metzgen [38], hence the multiplier itself is not actually removed even though it is no longer used for multiplies. For such designs the control logic, multiplexing, and the MIPS `HI` and `LO` registers used for storing the multiplication result are all removed, resulting in an area savings of approximately 80 equivalent LEs. In both cases the area savings is substantial, and depending on the desired application may be well worth any reduction in performance.

Figure 5.3 shows the impact of hardware support for multiplication on the number of cycles to execute each benchmark, but only for those benchmarks that use multiplication (see Table 4.2). We see that some applications are sped up minimally while others benefit up to 8x from a hardware multiplier, proving that multiplication support is certainly an application-specific design decision. Software-only support for multiplication roughly doubles the total number of cycles required to execute the entire benchmark suite compared to hardware support. This increase translates directly into a wall-clock-time slowdown of a factor of two, since the clock frequency remains unimproved by the removal

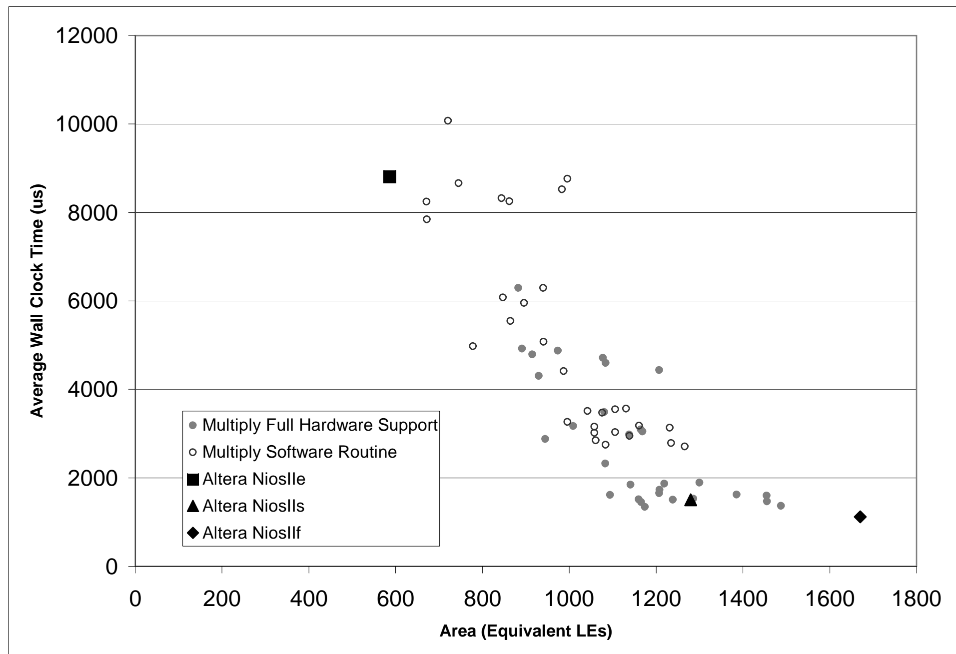


Figure 5.2: Average wall-clock-time vs area of processors with and without hardware multiplication support.

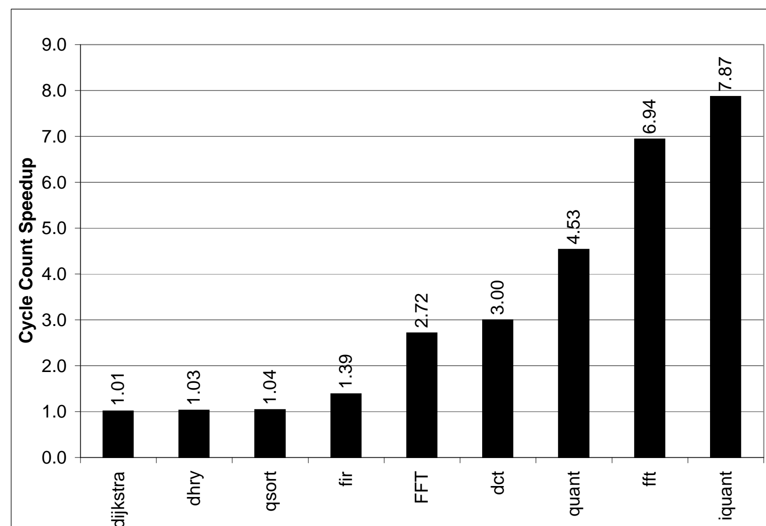


Figure 5.3: Cycle count speedup of full hardware support for multiplication, for only the benchmarks that use multiply instructions.

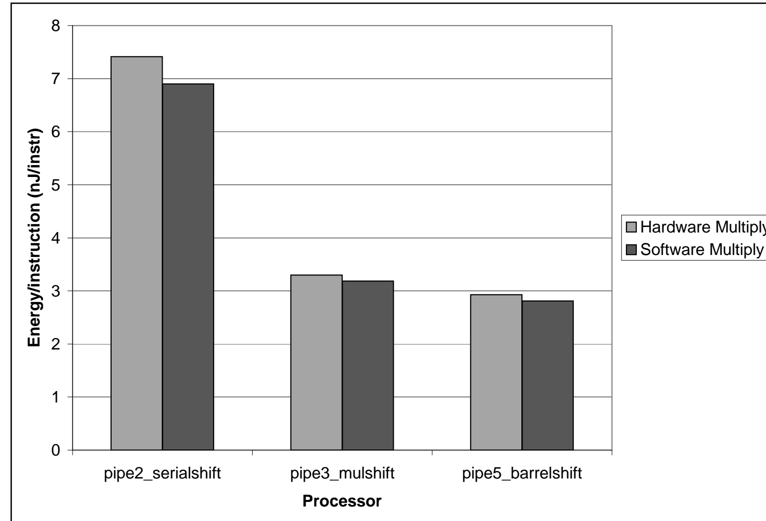


Figure 5.4: Energy/instruction for hardware vs software multiplication support.

of the multiplication hardware: during the design of each architecture, care was taken to ensure the critical path was not heavily dominated by any one path in the architecture. Hence, either the multiplication was not the critical path in the design, or if it was, there was another path equal to it in delay which prevented any noticeable increase in frequency.

The impact of multiplication support on energy can be seen in Figure 5.4 where three processors were used to see the difference between supporting multiplication in hardware or software. The figure shows that the energy per instruction is reduced for the software multiplication when averaged across all benchmarks. In fact, even benchmarks which do not contain multiply instructions exhibit reduced energy per instruction, proving that the multiply hardware is wasting energy even when it is not used (no circuitry exists to prevent it from switching when not used). On average, the software multiplication saves between 4% and 7% nJ/instruction, however, it also must execute more instructions—the single hardware multiply instruction is replaced by a subroutine. It follows that multiply-intensive applications will consume more energy if software multiplication is used, whereas applications with little or no multiplies will save on the energy wasted in supporting hardware multiply.

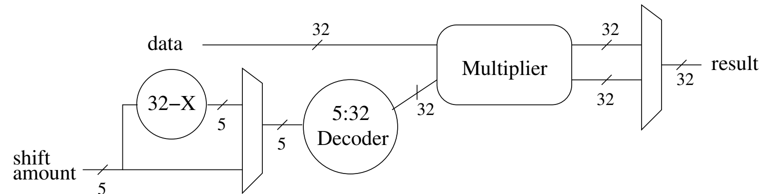


Figure 5.5: A barrel shifter implemented using a multiplier

5.3 The Impact of Shifter Implementation

Shifters can be implemented very efficiently in an ASIC design, however this is not true for FPGAs due to the relatively high cost of multiplexing logic [38]. We study three different shifter implementations: a *serial shifter*, implemented by using flip-flops as a shift register and requiring one cycle per bit shifted; a *LUT-based barrel shifter* implemented in LUTs as a tree of multiplexers; and a *multiplier-based barrel shifter* implemented using hard multipliers as shown in Figure 5.5. The 5:32 decoder has the effect of exponentiating the shift amount, allowing the left shifted result to be computed as $2^{shift\ amount} * (data)$ on the low 32-bits of the product, and the right shifted result to computed as $2^{32-shift\ amount} * (data)$ on the high 32-bits of the product. The multiplexer on the output of the multiplier selects between the high and low 32-bit words of the the 64-bit product for right and left shift respectively. More information on this implementation of shifting is available in Metzgen [38].

We study the effects of using each of these shifter types in four different architectures with different pipeline depths. Note that both barrel shifters, multiplier-based and LUT-based, are pipelined to match the clock speed supported by the rest of the pipeline. For example, the barrel shifters complete in two stages in the 5-stage pipeline but in one stage in the 3-stage and 4-stage pipelines. Figure 5.6 gives the wall-clock-time versus area tradeoff space for the different shifter implementations in the four architectures. In each series we have in order from left-to-right the 3 shifter implementations: Serial, Multiplier-based, LUT-based. The shape of the series is consistent across the four pipelines, and the L-type shape is indicative of the efficiency in using multiplier based shifters as discussed

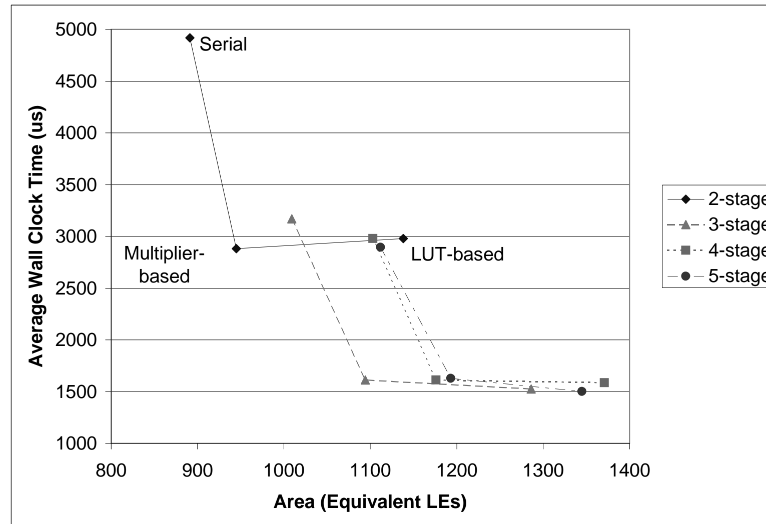


Figure 5.6: Average wall-clock-time vs area for different pipeline depths.

below.

The figure shows that the serial shifter is the smallest (furthest left) while the LUT-based barrel shifter is largest, on average 250 LEs larger than the serial shifter. In contrast, the multiplier-based shifter is only 64 LEs larger than the serial shifter: the multiplier is being shared for both shift and multiplication instructions, and the modest area increase is caused by the additional logic required to support shift operations in the multiplier (the 5:32 decoder and the multiplexer from Figure 5.5).

The impact of each shifter type on wall-clock-time is also seen in Figure 5.6. On average, the performance of both the LUT-based and multiplier-based shifters are the same, because in all architectures the cycle counts are identical. The differences in wall-clock-time are caused only by slight variations in the clock frequency for different architectures. Thus, the multiplier-based shifter is superior to the LUT-based shifter since it is smaller yet yields the same performance. There is a definite trade-off between the multiplier-based shifter and serial shifter: the multiplier-based shifter is larger as discussed before—however, it yields an average speedup of 1.8x over the serial shifter.

Ideally, customization should be performed not only to the microarchitecture, but to the ISA as well. The MIPS ISA was designed to eliminate interlocking by ensuring all operations can complete in a single cycle. However, when implemented on an FPGA

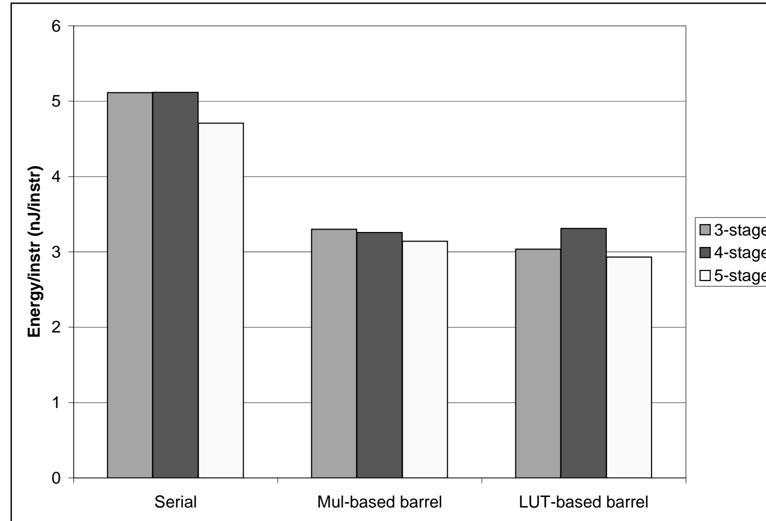


Figure 5.7: Energy per instruction across different pipeline depths and different shifter implementations

platform, shift operations require more than one cycle, hence requiring interlocking (interlocking was used in all four of the pipelines when the shifter was either the multiplier or LUT-based barrel shifter). Since shift and multiply instructions both require more than one cycle, one should accommodate either both or neither in the ISA. Currently, MIPS accommodates the multiply with dedicated destination registers, but not shift operations since it assumes they can be performed in a single cycle. To correct this discrepancy, one might provide dedicated result registers for shift operations; however, a better solution is that used by Altera in the Nios II ISA: remove the dedicated multiply result registers and reduce the output of the unit to 32-bits (the multiplier can emit either the 32 high or low bits of the product). With this modification, shifting can be supported for free in the multiplier, resulting in a single shift/multiply unit which is the only component which interlocks the pipeline.

In Figure 5.7 we show the energy per instruction for each of the shifter types with three different pipelines. Both the LUT-based and multiplier-based barrel shifters consume the same amount of energy, even though the LUT-based shifter is significantly larger in area. This is due to the increased switching activity in the multiplier and its tighter integration with the datapath (MIPS multiply instructions are written to dedicated registers while

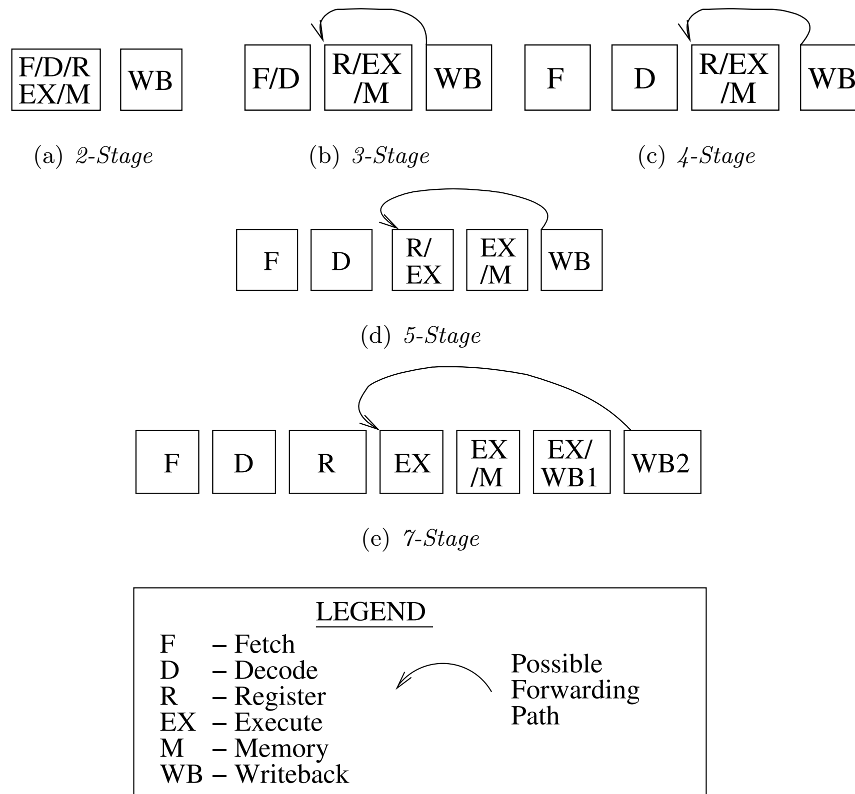


Figure 5.8: Processor pipeline organizations studied.

the shift result must be written directly to the register file). The processors with serial shifters consume more energy per instruction than those with barrel shifters because of the switching activity in the pipeline while the serial shifter is stalled. The shifter consumes significant energy as counters and comparators are toggled for every cycle of the shift, in addition to the shift register itself. Further energy overhead is caused by the SPREE Component Library which is yet to utilize power-aware features (even when not used, many functional units remain active). As the pipeline stalls for many cycles, these overheads accumulate and surpass that of a barrel shifter which would complete without stalling.

5.4 The Impact of Pipelining

We now use SPREE to study the impact of pipelining in soft processor architectures by generating processors with pipeline depths between two and seven stages, the organiza-

tions of which are shown in Figure 5.8. A 1-stage pipeline (or purely unpipelined processor) is not considered since it provides no benefit over the 2-stage pipeline: the instruction fetch stage and writeback stage can be pipelined for free, increasing the throughput of the system and decreasing the size of the control logic by a small margin. The free pipelining arises from both the instruction memory and register file being implemented in synchronous RAMs which require registered inputs. The 6-stage pipeline is also not considered since the 5-stage pipeline had competing critical paths in the writeback stage and decode stage requiring both stages to be split to achieve significant clock frequency gain. For every pipeline, data hazards are prevented through interlocking, branches are statically predicted to be not-taken, and mis-speculated instructions are squashed. For each pipeline depth, we use a multiplier-based shifter and full hardware multiply support. The results are similar with different shifter units and software multiply support.

Figure 5.9 shows that as expected, area increases with the number of pipeline stages due to the addition of pipeline registers and data hazard detection logic. The 5-stage pipeline suffers a considerably smaller area increase over the 4-stage pipeline. The reason for this is in the 5-stage pipeline, some interlocking logic was removed: For shorter pipelines, memory operations stall until they have completed, while in the 5-stage pipeline memory operations are contained within their own pipeline stage, eliminating the need for the corresponding stalling logic. The removal of this logic counteracts the increase in area from pipeline registers and hazard detection logic.

Figure 5.10(a) shows the maximum clock frequency of the different pipelines and illustrates that deepening pipelines indeed improves clock frequency. In an ideal setting, growing from N to N+1 stages should yield an (N+1)/N clock frequency speedup. Realistically, routing, setup time, and hold time overheads prevent such large gains. Moreover, in FPGAs, designs are synthesized into an interconnection of large logic blocks causing more coarse-grained control over register placement than is available at the transistor level. This effect also imposes additional overhead. Thus, the clock frequency gains shown in the figure are reduced. The critical path of the 2-stage pipeline was in the

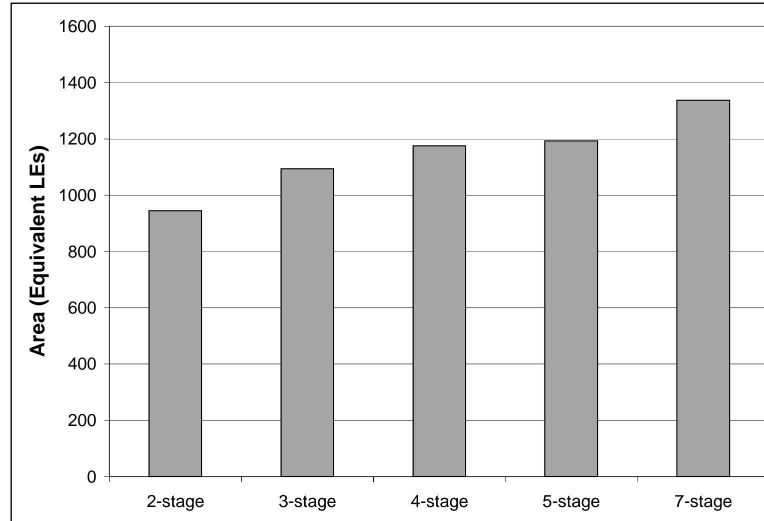
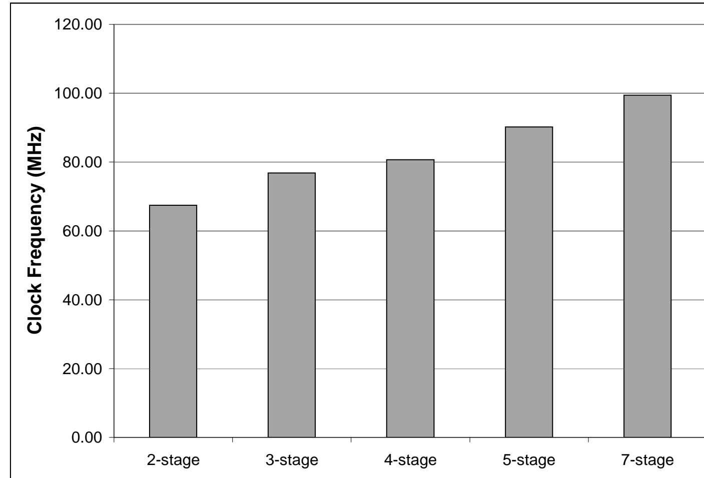


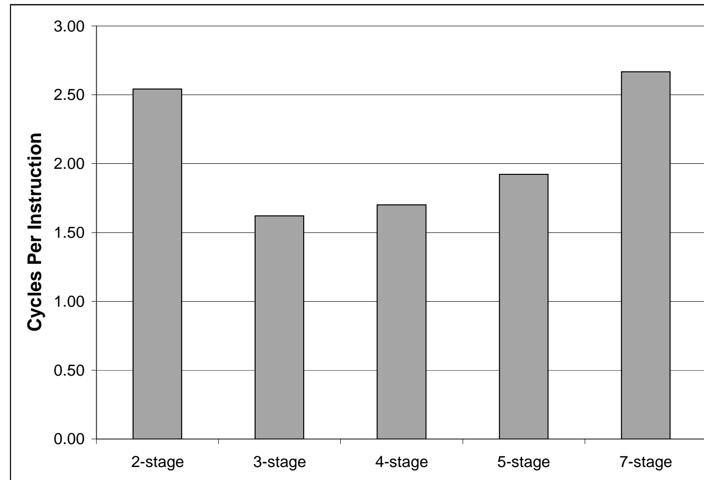
Figure 5.9: Area across different pipeline depths.

F/D/EX/M stage from the output of the register file through the arithmetic unit and memory alignment logic to the data memory. In the 3-stage pipeline this stage is pipelined and the hazard detection logic in the F/D stage forms the critical path. The 4-stage pipelines splits the fetch and decode into separate stages causing the critical path to appear in the EX/M stage as the data loaded from data memory is aligned and passed through the writeback multiplexer. In the 5-stage pipeline the execute stage is separated from the memory and the critical path becomes the branch resolution logic. Finally, in the 7-stage pipeline the branch resolution completes in the EX/M stage and the critical path moves to logic which prevents the squashing of delay slot instructions as described in the subsequent section.

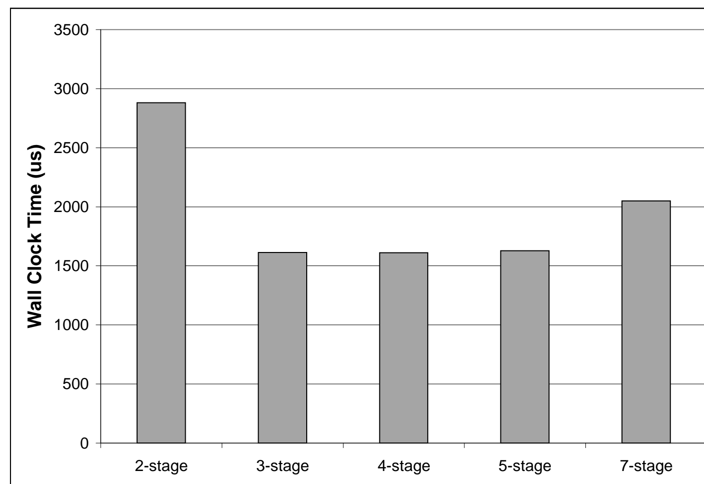
With respect to wall-clock-time, we see that deepening the pipeline improves performance over that of the 2-stage pipeline as seen in Figure 5.10(c). The 2-stage pipeline has neither branch penalties nor data hazards, but suffers from frequent stalls which are mostly due to an FPGA nuance: reading operands from the register file stalls the pipeline because the register file is implemented using the synchronous RAMs in the FPGA, which inherently incur a one cycle delay. This frequent stalling of the 2-stage pipeline is apparent in Figure 5.10(b) which shows that the cycles-per-instruction for the 2-stage pipeline is significantly worse than the other pipelines. There is a large perfor-



(a) *Clock frequency across different pipeline depths.*



(b) *Cycles-per-instruction across different pipeline depths averaged across benchmark suite.*



(c) *Wall-clock-time across different pipeline depths.*

Figure 5.10: Performance across different pipeline depths.

mance gain for increasing the pipeline depth from 2 to 3 stages since we pipeline this cycle delay. In the 3-stage pipeline the operand fetch is executed in parallel with the write back, which will cause stalls only on *read-after-write* (RAW) hazards instead of on the fetch of every operand. Combined with the increase in clock frequency shown in Figure 5.10(a), this decrease in stalls leads to the 1.7x wall-clock-time speedup for the 3-stage pipeline over 2-stages. We conclude that the FPGA nuance of using synchronous RAMs causes a major performance disadvantage to 2-stage pipelines.[1]

While Figure 5.10(a) shows that frequencies improve for the 4, 5, and 7 stage pipelines over the 3-stage, their cycle counts increase due to increased branch penalties and data hazards, as seen in Figure 5.10(b). The net effect on wall-clock-time, shown in Figure 5.10(c), shows that the overall performance of the 3, 4, and 5 stage pipelines remains relatively constant while the 7-stage pipeline suffers a performance loss (discussed in the next section). These results indicate that pipelining trades slower cycles-per-instruction for faster clock frequency evenly, such that the product of the two (wall clock time) remains mostly unaffected.

Figure 5.11 shows the wall-clock-time versus area for the different pipeline depths. Though the 3-stage pipeline seems the most attractive, it has the least opportunity for future performance improvements: for example, the cycle count increase suffered by the deeper pipelines can potentially be reduced by devoting additional area to branch prediction or more aggressive forwarding. Frequency improvements may also be possible with more careful placement of pipeline registers. We conclude that the 3-stage pipeline provides a good balance between area and performance while the 2-stage pipeline suffers in performance for a modest area savings and the deeper pipelines will suffer in area for modest performance gains.

Tradeoffs not only exist in the number of pipeline stages, but also in the placement of these stages. While deciding the stage boundaries for our 3-stage pipeline was obvious

[1]The register file may be implemented in the flip flops in each LE instead, however this would require 1024 LEs for the register file as well as associated multiplexing/enable logic.

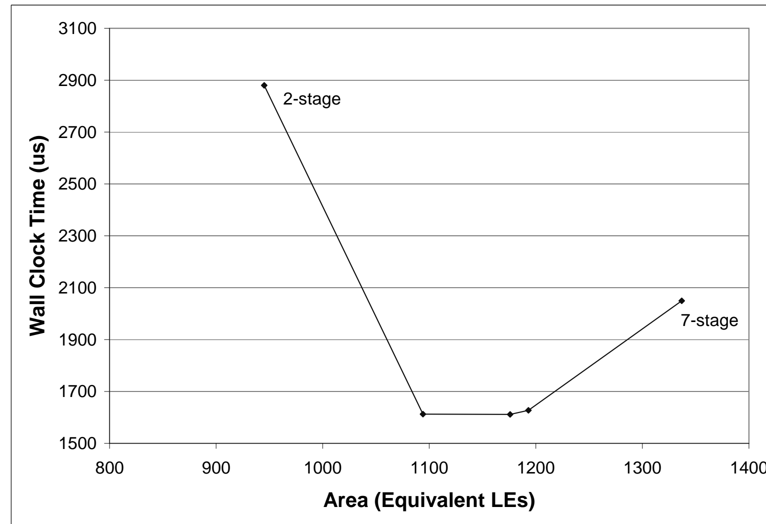


Figure 5.11: Wall-clock-time versus area across different pipeline depths.

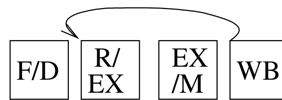


Figure 5.12: Alternative 4-stage pipeline

and intuitive, deciding how to add a fourth stage pipeline was not. One can add a decode stage as shown in Figure 5.8(c), or further divide the execution stage as in Figure 5.12. We implemented both pipelines for all three shifters and observed that although the pipeline in Figure 5.8(c) is larger by 5%, its performance is 16% better. Hence there is an area-performance trade-off, proving that such trade-offs exist not only in pipeline depth, but also in pipeline organization.

The energy per instruction of the pipelines can be seen in Figure 5.13 and the energy per cycle is shown in Figure 5.14. The energy consumed per instruction of the three, four, and five stage pipelines remain relatively consistent with a slight decrease as the pipeline depth increases in spite of the extra area gained, while the energy per cycle of the three, four, and five stage pipelines decreases with pipeline depth indicating that there is less switching activity per cycle as the deeper pipelines spend more time stalling. The fact that the energy per instruction decreases shows that the amount of wasted switching per instruction is decreased by a larger margin than the wasted switching associated

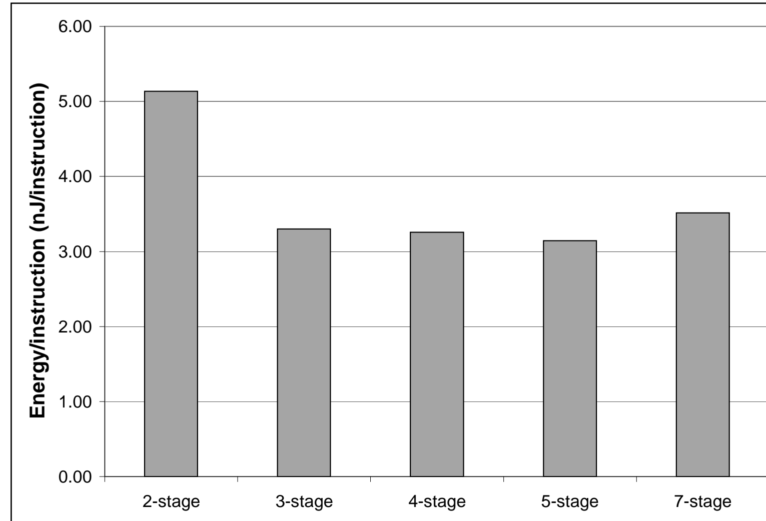


Figure 5.13: Energy per instruction for the different pipeline depths.

with stalling the pipeline and inserting null operations. We attribute this to decreased *glitching*[2] in the logic as more pipeline registers are added.

The 2-stage pipeline suffers from significant energy per instruction consumption over the 3-stage pipeline as seen in Figure 5.13. Earlier, it was shown that the 2-stage pipeline suffers from a performance disadvantage due to the presence of synchronous RAMs in the FPGA, namely, it must stall and wait for register file accesses. Curiously, Figure 5.14 shows that the energy consumed per cycle is the same for both, thus the extra cycles required for the register file accesses translates directly to increased energy per instruction. We attribute this to extra glitching in the 2-stage pipeline. There is no logic in the writeback stage, all components reside in the `F/D/EX/M` stage of the 2-stage pipeline, so any glitching, especially at the outputs of the instruction memory effects all logic downstream including control logic.

The 7-stage pipeline consumes more energy per instruction than the three, four, or five stage pipelines. This is caused by the increase in squashing as the branch penalty is increased and more instructions are fetched and then squashed. This wasted energy accumulates and surpasses any energy savings from decreased glitching.

[2]Glitching refers to the spurious toggling of gate outputs often due to differing arrival times of the gate inputs.

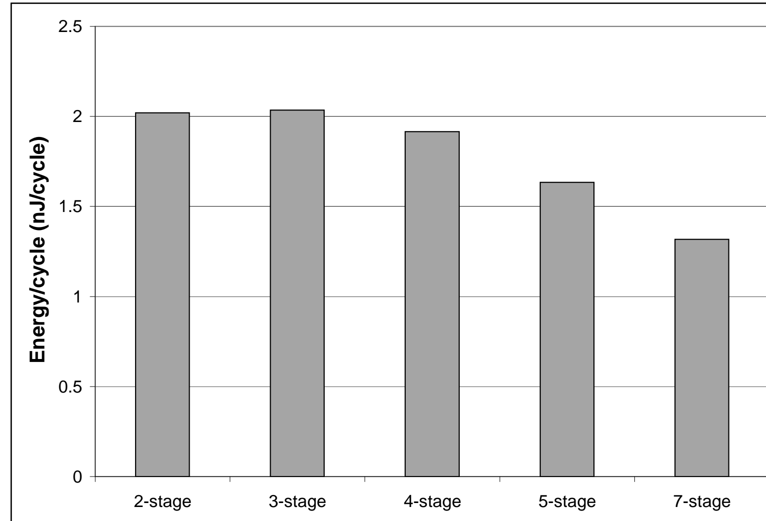


Figure 5.14: Energy per cycle for the different pipeline depths.

5.4.1 Branch Delay Slots in the 7-stage Pipeline

In the 7-stage pipeline shown in Figure 5.8(e), delay slots are very problematic as the pipeline is now big enough to contain two unresolved branches and their corresponding branch delay slot instructions. SPREE was designed to allow users to extend the pipeline arbitrarily without having to worry about branch mis-speculation. SPREE can automatically squash mis-speculated instructions, while protecting the delay slot instruction, but the hardware to support this became a performance bottleneck.

Branch delay slots are complicated by two factors: (i) A branch delay slot instruction can be in any stage prior to the stage the branch is resolved, and (ii) there can be more than one branch delay slot instruction in the pipeline at a time. The first factor is shown in Figure 5.15, where the branch (`BEQ`) is resolved in the EX stage but the branch delay slot instruction (`ADD`) stalled and was separated from the branch (ie. the branch continues through the pipeline as the branch delay slot instruction is stalled in an earlier stage). If the branch is taken, only the F stage should be squashed. It follows that a branch delay slot instruction can be in any stage prior to the branch resolution which complicates the logic used to protect branch delay slot instructions. The second factor is also apparent in Figure 5.15. If the `ADD` had not stalled, the branch delay slot instruction for the jump `JR`

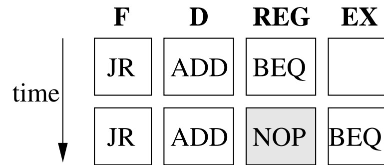


Figure 5.15: Branch delay slot instruction separation.

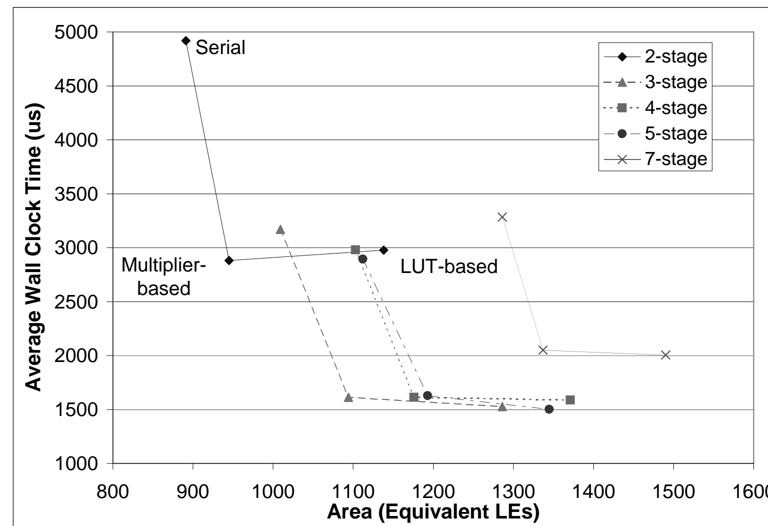


Figure 5.16: Average wall-clock-time versus area space for all pipelines.

would have been fetched and thus there would be two unresolved delay slot instructions in the pipeline. Both of these factors complicate the identification of branch delay slot instructions in the pipeline.

To facilitate the automatic handling of mis-speculated branches, we have addressed both of the two aforementioned factors. The branch delay separation problem is solved by propagating a flag through the pipeline which indicates whether the instruction in that stage was fetched after a branch. This flag grants immunity to that stage from squashing hence protecting branch delay slot instructions. To address the multiple delay slot problem, the fetch unit is stalled if a delay slot instruction exists in the pipeline and the last instruction fetched was a branch. This prevents multiple delay slot instructions from being in the pipeline. With both factors handled, users can extend the pipeline and place the branch resolution in any stage without worrying about branch mis-speculation.

Figure 5.16 shows the 7-stage pipeline in the performance-area design space alongside

the smaller pipelines. Due to increased hazard detection logic and pipeline registers, the area of the 7-stage pipeline is significantly larger. Curiously, the 7-stage pipelines are approximately 150 LEs larger than the 5-stage pipelines, while the 5-stage pipelines were only 90 LEs larger than the 3-stage pipelines. This may seem unintuitive, however, the hazard window doubled from 2 to 4 in going to 7-stages, and the branch penalty also doubled from 1 to 2 cycles. Both require more hardware for detecting data hazards and managing branch mis-speculation. In addition, the extra logic required to handle the multiple delay slot problem discussed above also increases area.

With respect to wall clock performance, we notice that the 7-stage pipeline suffered a large speed degradation, compared to the 4 and 5-stage pipelines which were able to match the performance of the fast 3-stage pipeline. One reason for this performance degradation is the increased cycle count due to the larger hazard window and branch penalty. With more hazards occurring and larger branch penalties, the pipeline spends more time stalling instead of executing useful instructions. Additional hardware such as aggressive forwarding and branch prediction would be required to counteract this effect, and is a topic of future work.

The other reason for the performance degradation is the less than satisfactory clock frequency improvement. The clock frequencies of all the pipelines are shown in Figure 5.10(a). The clock frequency gain from 3 to 5-stages is much larger than from 5 to 7-stages: the new logic inserted to prevent the multiple delay slot problem had formed the critical path in the design and was responsible for limiting the frequency gain. This motivated an investigation into the exact cost in speed and area of this logic. SPREE was modified to regenerate the 7-stage pipelines without the logic to handle the multiple delay slot problem. These processors are no longer correctly functional, but they give insight into the impact of the multiple delay slot problem.

Figure 5.17 shows the clock frequency speedup for each of the three 7-stage pipeline variations when the logic for the multiple delay slot problem is removed compared to when it is present. The figure obviates the frequency limitation imposed by the new

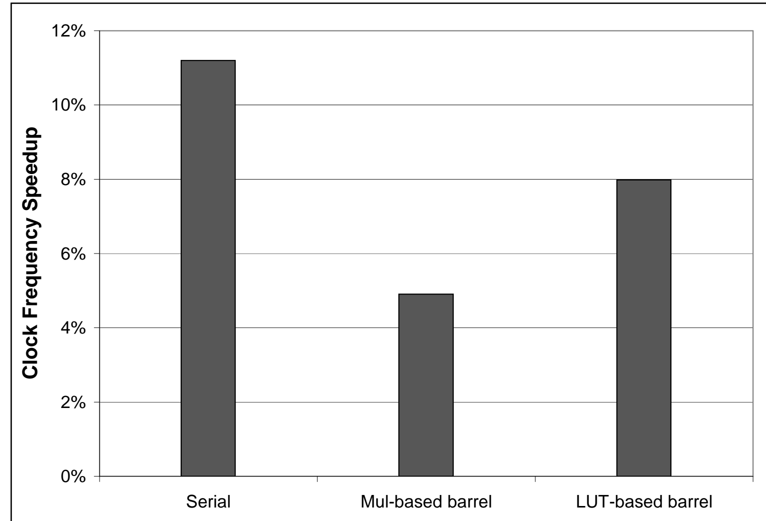


Figure 5.17: Clock frequency speedup after ignoring multiple delay slots.

extra logic. In the case of the serial shifter, the clock frequency is 11% better without the extra logic, the multiplier-based barrel shifter is 5% better as the multiplier also limits the clock frequency, and the LUT-based barrel shifter is 8% faster. In terms of area, the extra logic requires only 12 LEs, however they are in the stalling logic, which is a critical part of the design. This statistic invites further investigation into the actual cost of branch delay slots. They may be clearly beneficial for small pipelines, however modern ISAs [21] do not use branch delay slots for three reasons: (i) branch penalties in modern architectures are so big that saving one or two cycles using branch delay slots is not a practical use of chip area; (ii) branch predictors are so good they can predict branches with close to perfect accuracy [35]; and (iii) modern architectures speculate on all instructions so there is no need to have branch delay slot instructions which only allow for speculation after branches. We can use our infrastructure to accurately quantify when delay slots are good or bad, but we leave this for future work.

5.4.2 The Impact of Inter-Stage Forwarding Lines

An important optimization of pipelined architectures is to include forwarding lines between stages to reduce stalls due to RAW hazards. We use SPREE to evaluate the benefits of adding forwarding lines to our pipelined designs. In all pipelines studied in

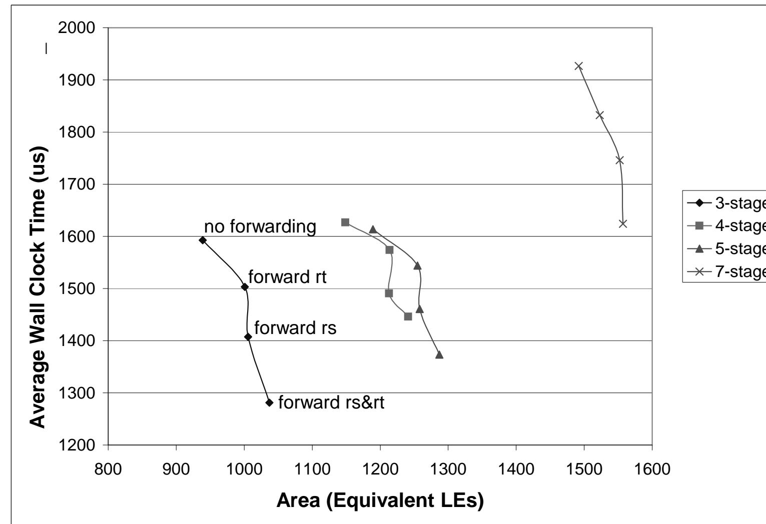


Figure 5.18: Average wall-clock-time vs area for different forwarding lines.

this paper, there is only one pair of stages where forwarding is useful: from the write-back stage (WB) to the first execute stage (EX) (see Figure 5.8). Since the MIPS ISA can have two source operands (referred to as `rs` and `rt`) per instruction, there are four possible forwarding configurations for each of the pipelines: no forwarding, forwarding to operand `rs`, forwarding to operand `rt`, and forwarding to both `rs` and `rt`.

Figure 5.18 shows the effects of each forwarding configuration on wall-clock-time and area (note that points in the same series differ only in their amount of forwarding). As more forwarding is added, the processor moves right (more area) and down (faster wall-clock-time). While there is clearly an area penalty for including forwarding, it is consistently 65 LEs for any one forwarding line, and 100 LEs for two across the three different pipeline depths. In all cases the performance improvement is substantial, with more than 20% speedup for supporting both forwarding lines. An interesting observation is that there is clearly more wall-clock-time savings from one forwarding line than the other: forwarding operand `rs` results in a 12% speedup compared to only 5% for operand rt, while the area costs for each are the same. Also, the inclusion of forwarding did not decrease clock frequency significantly.

The impact of forwarding lines on energy is shown in Figure 5.19. Energy is decreased by 15% (compared to no forwarding) when forwarding is present for both the `rs` and

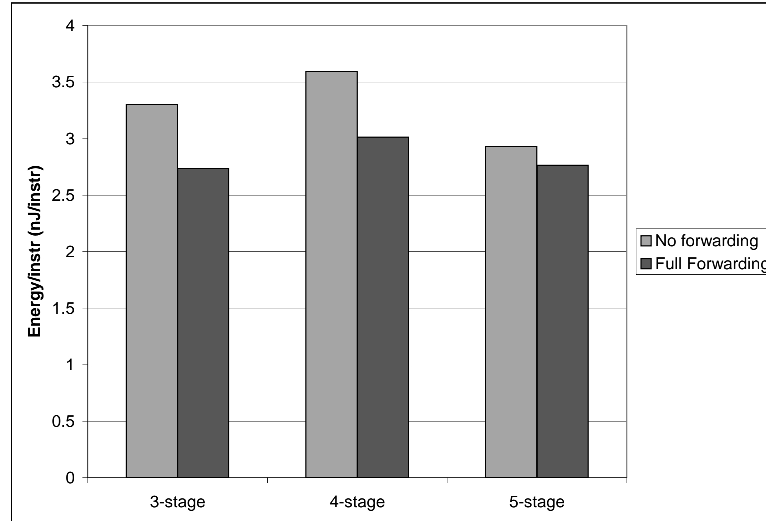


Figure 5.19: Energy per instruction for three pipelines each with no forwarding, and full forwarding (both `rs` and `rt`).

`rt` operands. This indicates that the energy consumption of the forwarding lines and associated control logic is considerably less than the energy consumed in the pipeline when instructions are stalled (without forwarding lines).

5.5 Register Insertion for Clock Speed Enhancement

Adding pipeline registers increases frequency but decreases CPI as more data hazards and branch penalties occur. Registers can be used in a more direct way for trading clock frequency and CPI: non-pipelined registers can be inserted within a pipeline stage which prevent it from executing in a single cycle, but allow it to run at a higher clock frequency. We have coined this technique RISE (Register Insertion for clock Speed Enhancement). An example of RISE is having a multi-cycle unpipelined execution unit, such as a 2-cycle unpipelined multiply. Whenever used, the multiplier must stall the pipeline for 2-cycles as it computes the result. The alternative is to use a single-cycle multiplier, but this may limit the clock frequency of the whole processor. In this way, RISE is used to trade maximum clock frequency for CPI.

The effect of RISE is examined in two cases. First we consider the three stage pipeline

with multiplier-based barrel shifter. When the multiplier-based shifter is implemented as a single-cycle execution unit, it forms the critical path of the processor and limits the clock frequency to 48.7 MHz while the average CPI is 1.48. If the multiplier-based shifter is implemented as a two-cycle unpipelined execution unit, the frequency increases to 76.9 MHz but the CPI also increases to 1.62. The clock frequency is improved by 58% while the CPI worsens by 9.4%. In this case, no benchmark benefits from the one-cycle unpipelined implementation and thus RISE is an intuitive architectural design choice where the two-cycle implementation is a clear win.

We now consider the 5-stage pipeline with 2-cycle multiplier-based barrel shifter. This processor has critical path through the shifter which limits the clock speed to 82.0 MHz while achieving 1.80 average CPI. RISE is used to make the multiplier-based shifter a 3-cycle unpipelined execution unit which results in a clock frequency of 90.2 MHz and 1.92 average CPI. The 10% clock frequency improvement is countered by an average CPI increase of 6.7%. Figure 5.20 shows the instruction throughput in MIPS of both processors and indicates that benchmarks can favour either one. Specifically, BUBBLE_SORT achieves 10% increased performance when using the 3-cycle multiplier-based shifter while CRC achieves 6% increased performance with the 2-cycle implementation. It follows that RISE can be used both for making application specific tradeoffs between clock frequency and CPI, and also for making intuitive architectural design decisions as in the example previous.

5.6 Architectures that Minimize Area

SPREE has also been used to explore more radical architectural modifications. Since SPREE can automatically accommodate different interfaces and can allow for different component implementations, one can experiment with more creative architectures. Two such experiments are described below; both have the common goal of reducing the area of the processor.

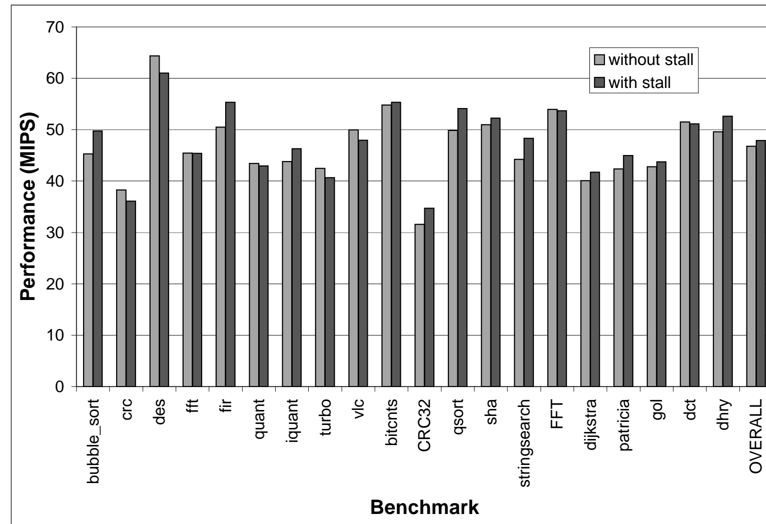


Figure 5.20: The impact of RISE on a processor across the benchmark set

5.6.1 Fully Serialized ALU

Often the ALU and other execution units account for a significant fraction of the area. This, of course, depends on the implementation of these execution units, so one interesting experiment is to compare how small the execution units can be made. In this experiment, all of the execution units (excluding the multiplication) are serialized and shared, creating a single serial ALU which can perform shifting (left, right-logical, right-arithmetic), logic functions (and, or, nor, xor) and arithmetic functions (add, sub, set-on-less-than). The logic and arithmetic functions will now require 32 cycles to complete, while the shifting depends on the amount being shifted. This serial ALU is implanted in two different datapaths, which are then generated by SPREE, and benchmarked. The two datapaths employ the same 2-stage pipeline (no hazards or branch penalties) and differ only in the amount of extra cycle latency in each (the number of non-pipelined registers in the datapath). One of the datapaths has no extra cycle latency and has a typical CPI of 35 cycles, the other has a significant amount of extra cycle latency providing a typical CPI of 40 cycles, but achieves faster clock speeds.

Figure 5.21 shows the entire design space as generated by SPREE, including the two designs with the serial ALU. We see that the wall-clock-times for these two designs are

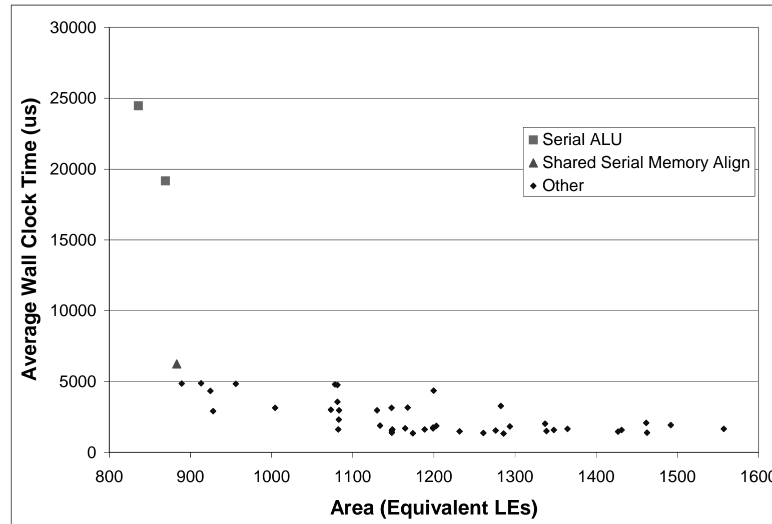


Figure 5.21: Average wall-clock-time vs area space including more serialized processors

significantly worse than any other design, approximately 5x and 3.92x slower than the next slowest non-serialized generated design, for the 35 and 40 CPI datapaths respectively. The 40 CPI datapath performs 27% faster in overall wall-clock-time since the increased CPI lead to a much faster clock speed: 119 MHz versus 88 MHz for the 35 CPI datapath. Adding cycle latency here for increased clock speed is clearly beneficial since the base CPI value is so high, and the logic delay is so imbalanced (the high-speed serial ALU is placed in a datapath which cannot sustain such high clock speeds).

The areas of the two designs are reduced but not by a significant margin. Compared to an equivalent datapath which uses a serial shifter, but performs logic and arithmetic operations in parallel, the serial ALU saved 53 LEs, approximately 6% area savings, while incurring 5x performance degradation. We had expected close to 64 LEs from the serialization of the logic and arithmetic units which each require 32 LEs, however even greater gains were expected in reducing the multiplexer which selects between the results of the arithmetic, logic, and shifting units. Unfortunately, the extra control logic required to perform the operations in serial, in combination with the increased multiplexing at the inputs of the serial ALU, diminished the expected area savings.

5.6.2 Shared Shifting and Memory Alignment

Another interesting architectural idea is to use the shifting logic to perform memory alignment operations. Memory alignment must be performed when reading/writing 8 or 16-bit values from the 32-bit memory ports. For example, when a load is performed, all 32-bits at that location are read, then logic must zero out unwanted portions of the word and potentially shift the desired value to the appropriate bit position. As mentioned before, this shifting logic is generally expensive in FPGAs so it may be advantageous to use the shifting unit to perform this task. This modification was also performed to a 2-stage pipeline with no extra cycle latencies.

Figure 5.21 shows this point in the performance-area design space. Similar to the serial ALU, this design is also smaller and slower than any other design. Compared to an architecture that is nearly equivalent (i.e. without sharing the shifter for memory alignment), this new design performs 28% slower and saves only 9 LEs. This is an insignificant area savings, indicating that the shifting logic saved was not appreciably larger than the extra logic required to coordinate the sharing and integrate with the shifter unit.

5.7 Instruction Set Subsetting

SPREE has been augmented with the capability to reduce the submitted datapath so that only connections and components which are actually used by the ISA are implemented in the generated soft processor. This capability enables another interesting ability: if one reduces (*subsets*) the ISA, the processor is automatically reduced as appropriate. Thus we can explore the effect of customizing a processor for running a specific application by analyzing the application and removing support for all instructions which it does not use from the processor.

Applications rarely fully utilize the complete ISA. To verify this claim, all benchmarks were analyzed and the number of unique instructions in each application counted.

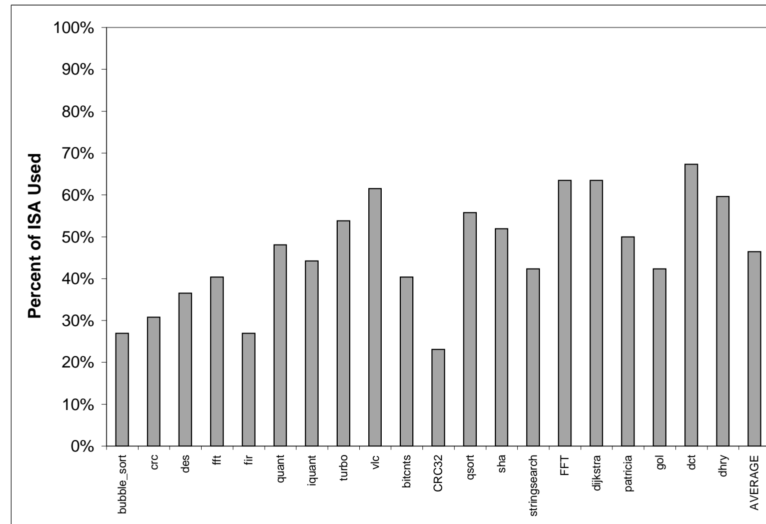


Figure 5.22: ISA usage across benchmark set

The results of this analysis are shown in Figure 5.22 which illustrates that most of the benchmarks used in this thesis rarely use more than half of the supported instructions.[3] Some benchmarks, in particular BUBBLE_SORT, FIR, and CRC32 use only about one quarter of the ISA. This fact motivates an investigation into the impact of eliminating the architectural support for parts of the ISA which are not used. Such an elimination is practical if (i) it is known that the soft processor will only run one application; or (ii) the soft processors can be reconfigured, since the designer can regenerate the soft processor as the application changes.

To evaluate the effect of subsetting, three previously generated architectures were subsetted for each of the 20 benchmarks: (i) A 2-stage pipeline with LUT-based barrel shifting; (ii) The 3-stage pipeline with multiplier-based barrel shifting; (iii) a 5-stage pipeline with LUT-based barrel shifting. Since the execution of each benchmark is unaffected, clock frequency is used to measure performance gain.

The relative area of each subsetted processor with respect to its non-subsetted version is shown in Figure 5.23. It is apparent that the three benchmarks which utilized 25% of the ISA (BUBBLE_SORT, FIR, and CRC32) achieved the most significant area savings.

[3]The reader is reminded that the current supported instruction set is a subset of MIPS-I and consists of 50 instructions. In comparison, the MIPS-IV ISA contains 128 instructions excluding floating point.

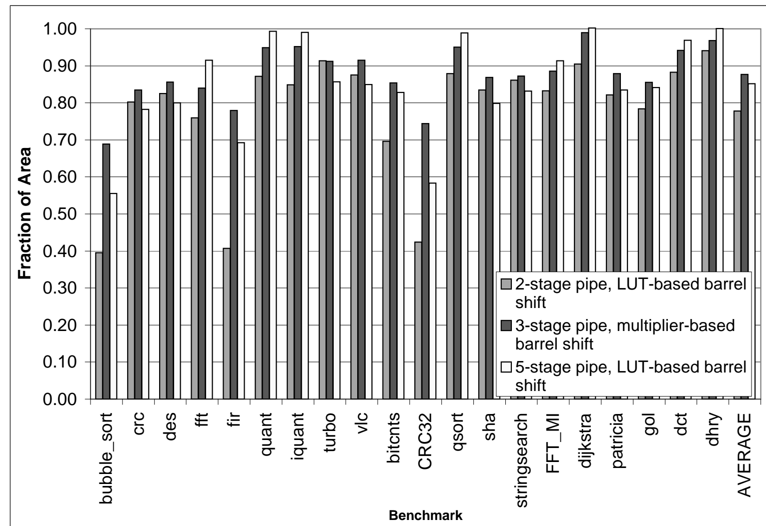


Figure 5.23: Area effect of subsetting on three architectures

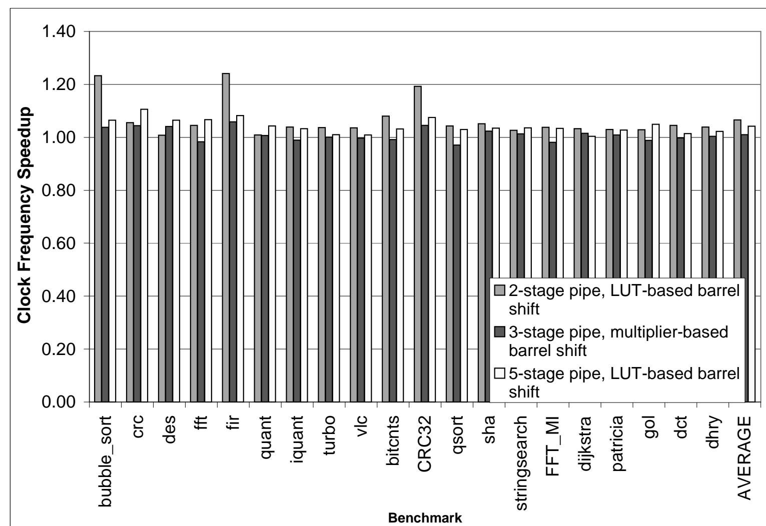


Figure 5.24: Clock Speed effect of subsetting on three architectures

In the 2-stage architecture, a 60% area savings is achieved by these three benchmarks while most other benchmarks saved 10-25%. Closer inspection of these three benchmarks revealed that they are the only benchmarks which do not contain shift operations. The large area savings comes from elimination of the shifter unit since, as mentioned before, shifters are large functional units in FPGAs. The savings is more pronounced in the 2 and 5-stage pipeline where the shifter is LUT-based and hence larger as seen in Section 5.3.

There is one problem with eliminating the shifter unit completely. Recall that in the MIPS ISA, there is no explicit `nop` instruction, `nops` are encoded as a shift left by zero instruction. To facilitate the complete removal of the shifter, one must remove all `nop` instructions, or, re-encode the `nop` instruction as another instruction without state side-effects. In this work the `nop` was re-encoded as an `add` zero (similar to Nios II) to allow for complete removal of the shifter. All benchmarks use the arithmetic unit, therefore an `add` does not hinder any subsetting.

Figure 5.24 shows the clock frequency speedup of the subsetted architectures. In general we see modest speedups, 7% and 4% on average for the 2 and 5-stage pipelines respectively. The 3-stage pipeline, being one of the most well-balanced in terms of logic delay, did not achieve significant speedups. In other words, when logic was removed from a path, there is often another path to maintain the previous critical path length; the odds of reducing all paths is relatively small. Again there is notable performance improvement in the 2-stage pipeline for the three benchmarks, BUBBLE_SORT, FIR, and CRC32. This is because the LUT-based shifter was severely limiting the clock frequency of that architecture, removing it allowed for more than 20% frequency improvement.

5.8 Optimizations

During the development of SPREE, many optimizations were discovered. These optimizations are caused by FPGA-specific considerations, CAD timing analysis limitations,

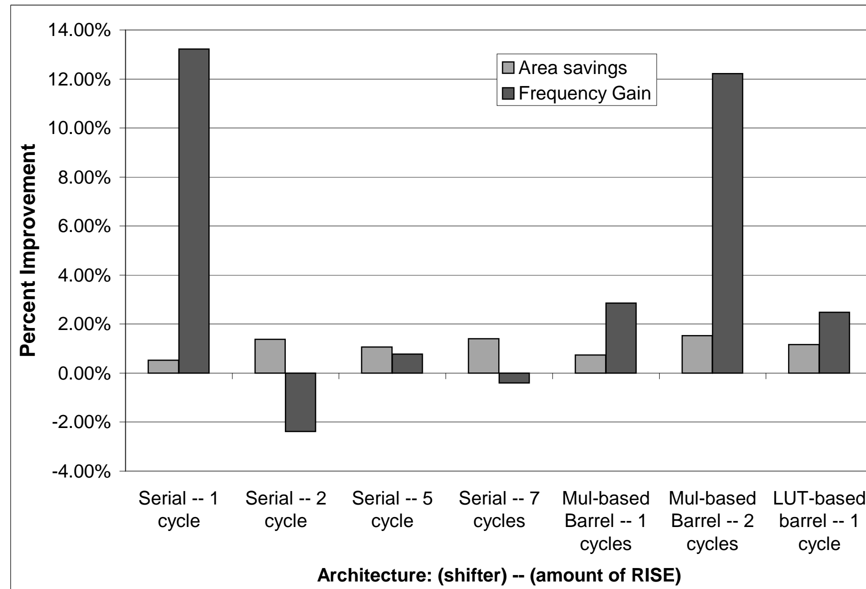


Figure 5.25: Impact of 8-bit store port on area and performance of different 2-stage pipelines.

or limitations in SPREE. We discuss the following four optimization below: dual word-size data memory which is an example of an FPGA-specific optimization utilizing the robustness of the on-chip RAM blocks; arithmetic unit result splitting which often prevents a false path from becoming critical; previous stage decode which prevents control paths from becoming critical; and instruction-independent enable signals which give the user the ability to tell SPREE to be less conservative with a component enable signal.

5.8.1 Dual Word-Size Data Memory

The memory alignment of byte and halfword loads and stores to data memory required by the MIPS ISA costs a significant amount of area and can often form part of the critical path of an architecture. However, this cost can be reduced by capitalizing on the capabilities of modern FPGAs. The RAMs on Stratix are all dual-ported and can individually have their aspect ratio modified. Therefore, we can make one port a typical 32-bit wide port, and the other an 8-bit wide port for reading/writing bytes. Doing so would reduce the memory alignment logic since we only need to accommodate halfwords, which we do with the 32-bit port.

We implemented this technique for the data memory of a number of architectures: all

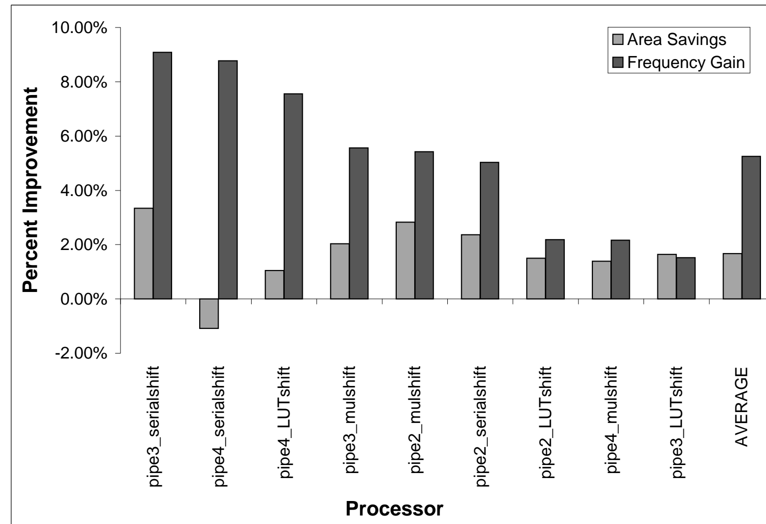


Figure 5.26: Impact of result splitting on area and performance.

2-stage pipelines with the three different shifters and different amounts of RISE. RISE is quantified according to the number of cycles typically required for the execution stage. For example, a RISE of 5 cycles means that most instructions require the execution stage to stall for 5 cycles. We realized that for loads the results are actually the same or worse since we must now multiplex between the two ports of the RAM. We then implemented the optimization for stores only and gathered the results shown in Figure 5.25. The area savings is apparent in all architectures and amounts to 10-12 LEs. With respect to frequency we see two architectures gain significantly in clock speed. These two architectures both had critical paths through the store memory alignment unit. Another architecture suffered more than 2% performance degradation, which is outside our allowable noise margin. We can attribute this penalty partly to noise, and partly to the extra routing required to access the newly used port on the RAM. Because of this, one must be careful when using this optimization since it is not always beneficial.

5.8.2 Arithmetic Unit Result Splitting

As discussed in Section 3.4.2, false paths can cause the timing analysis tool to report a conservative value for maximum operating clock frequency. This phenomenon was observed in the arithmetic unit, which performs addition, subtraction, and the set-on-

less-than operations. The set-on-less-than instruction compares the two source operands and return 1 or 0 depending on whether the first operand is less than the second. To perform this operation, a subtraction is executed and the carry-out of the most significant bit is fed to the least significant bit of the 32-bit result. The memory alignment logic then requires the two least significant bits of the result. However, the timing analysis tool, being conservative, assumes that the worst case path is that of a set-on-less-than instruction which passes through all 32-bits of the arithmetic unit. Clearly the set-on-less-than path and the memory alignment path will never be used by the same instruction as MIPS is a load-store ISA. To overcome this inaccuracy, the arithmetic unit was rewritten to have two outputs, one of which goes only to the memory alignment logic (which does not contain the set-on-less-than result), and the other which goes to the writeback stage (which contains all results of the arithmetic result).

We applied this transformation to several processors and achieved the results shown in Figure 5.26. Several architectures achieve 4% to 9% clock frequency speedup, others less than 2% which falls in the noise margin. We also see an area savings likely caused by the combining of multiplexing of the two arithmetic results with the results of all other functional units. Overall this optimization gives good area and speed improvements.

5.8.3 Previous Stage Decode

In the pipeline, the decoding of opcodes is distributed to each stage and is computed in the same stage they are used. This may extend the critical path of an architecture since the decode logic can be significant. Most notably, the arithmetic unit will suffer from this problem whenever in the critical path because the carry-in to the adder is a control signal (it is low for an add instruction and high for a subtract). As a solution, one can pre-compute the opcode values in the previous stage and store their values in pipe registers, which may improve the critical path. This option was implemented as a parameter to SPREE and we compared architectures with it turned on and off. The architectures used were a 3-stage, 4-stage, and 7-stage pipeline, where in each pipeline

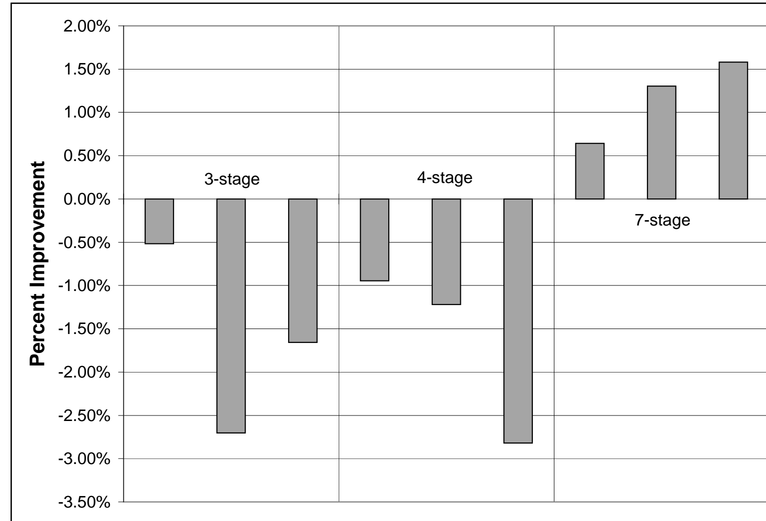


Figure 5.27: Impact of previous stage decode on performance

the shifter was varied.

Figure 5.27 shows the clock frequency improvement for previous stage decoding across the selected benchmarks. In the 3 and 4-stage pipelines, a noticeable speed degradation occurs – though this is close to our 2% place and route error margin, we regain confidence in that all six points show the same trend. This is caused by the merging of decode logic in the first stage of the pipeline, causing it to grow much bigger. For example, the 3-stage pipeline has only two stages of decode logic since the third stage is writeback during which nothing need be controlled. When using previous stage decode for this pipeline, all the decode logic becomes merged into stage 1, where some of the opcodes are registered for stage 2, while others are not because they are used in stage 1—there is no stage before stage 1 to pre-compute the opcodes in. This merging of decode logic is performed automatically by SPREE which currently does not support maintaing the two decode logic blocks separately. This growth in the decode logic not only grows in area, but in routing resources. The congested routing often makes any opcode-affected critical path even worse. In the 7-stage pipeline however, the decode logic is distributed so thinly to each of the stages, that combining decode logic in stages 1 and 2 has no negative impact on the frequency and the improvements expected are seen. With the speed improvement so small, and the approximately 25 LE cost for using previous stage decode, this option

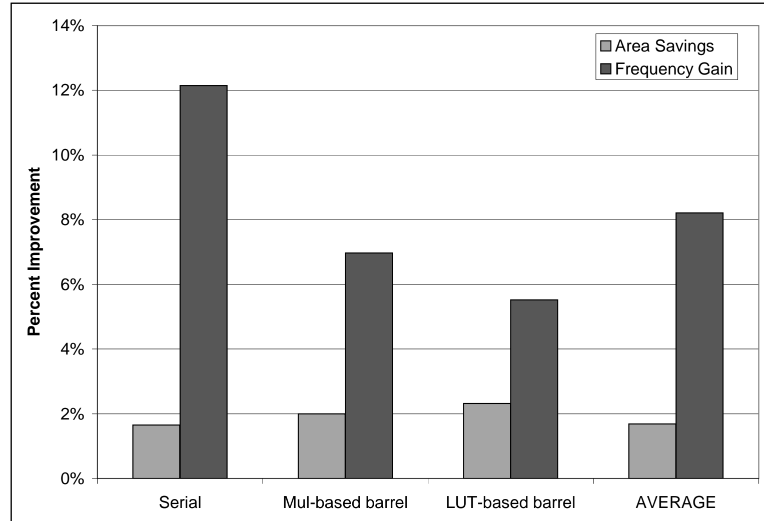


Figure 5.28: Impact of instruction-independent enable signals on area and performance

is off by default.

5.8.4 Instruction-Independent Enable Signals

As mentioned previously SPREE assumes a component interface which uses enable signals to schedule operations. These enable signals are activated by SPREE when two conditions are met: (i) the stage the component resides in is activated; (ii) and when the instruction being executed requires the component activated. However, certain modules can be activated for any instruction without affecting the functionality of the processor, hence not requiring the second condition. For example, reading an operand from the register file may occur for any instruction, whether the instruction requires an operand or not. Eliminating this condition would reduce the logic necessary for the enable signal, and since the hazard detection and stalling logic was seen as a critical path, this reduced logic may even increase clock frequency. The option to eliminate this condition by introducing instruction-independent enable signals was added to SPREE allowing the exploration of this optimization. Instruction-independent enable signals were used on the register file for the 3-stage pipeline whose critical path was the hazard detection and stalling in the operand fetch stage. We varied the shifter implementation and measured the effect of this optimization on each variant.

Figure 5.28 shows the impact of this optimization on the area and performance of each variant. Since cycle-to-cycle behaviour is unchanged, clock frequency is used to report overall performance improvement. The area savings is approximately 20 LEs which translates to almost 2% area reduction. The performance improvement is significant, varying between 12% and 5% depending on the architecture and its critical path. These results show that this optimization reduces logic and can significantly increase the clock frequency without cost. There may be negative impact on energy consumption from this optimization, but we have yet to study its effect and plan to once we consider more power-aware architectures.

5.9 Application Specificity of Architectural Conclusions

So far, the performance of each processor has been measured by averaging over a large set of benchmarks as described in Chapter 4, giving a "universally" fast/slow attribute to each processor. Since the long-term goal of this research is to make application-specific design decisions, one must consider how much the design space varies if only considering one application at a time. Specifically, it would be interesting to know what the penalty is for selecting a universally good processor versus making application-specific decisions. This problem is analyzed by comparing the performance of all processors on a per-application level.

Figure 5.29 shows the performance of all processors on each benchmark in millions of instruction per second (MIPS). The processors include all pipeline stages, shifter implementations, and multiplication support. The bold line indicates the performance of the fastest overall processor (the 3-stage pipeline with LUT-based barrel shifting) which is calculated by taking the arithmetic mean of all performance measurements across the benchmark set. The figure shows that the fastest overall processor is also often the fastest processor for each benchmark since there are very few dots that are higher than the bold line. Some exceptions are for the STRINGSEARCH, CRC32, and TURBO benchmarks for

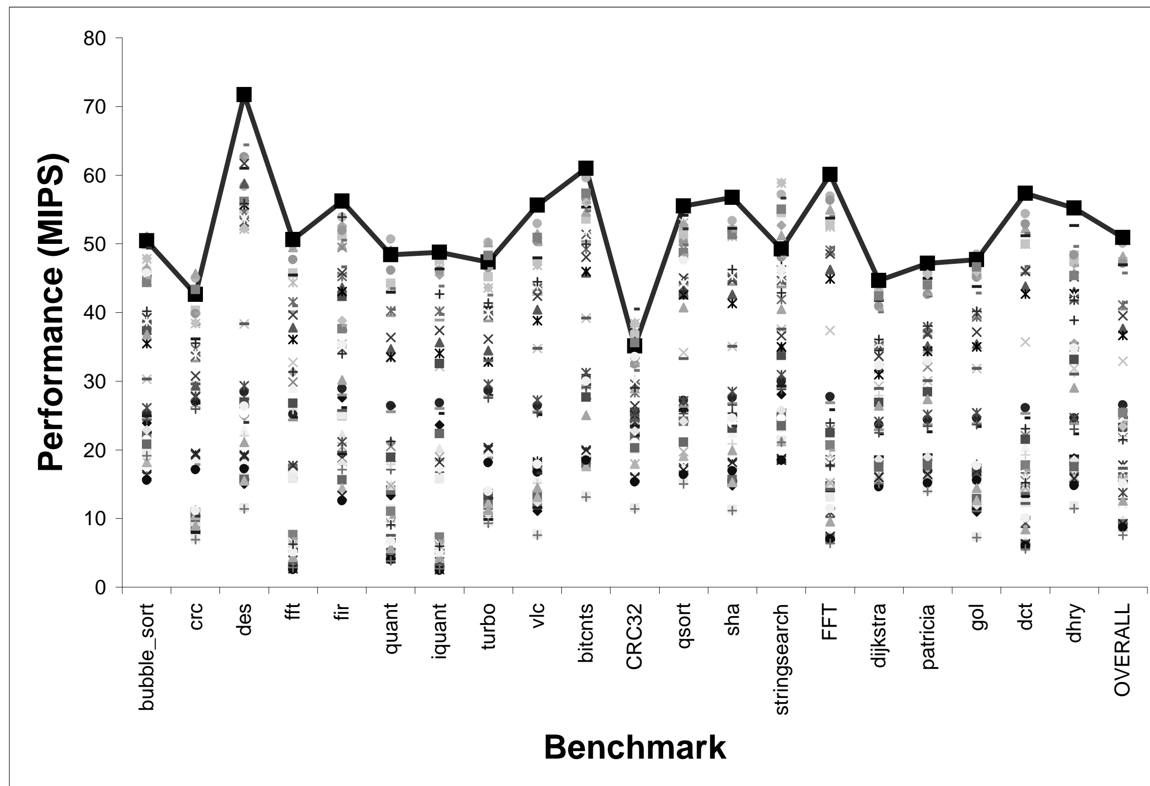


Figure 5.29: Performance of all processors on each benchmark—in bold is the overall fastest processor.

which there are processors that can execute 16.4%, 13.3%, and 5.7% faster respectively. Overall there is little room for application specific consideration with respect to performance since the architectural axes used in this work often trade area for speed. For example, the benefit of using a serial shifter or software multiply is in reducing area at the expense of performance. Therefore, if only performance is considered, there is no motivation for using either of these two options. This motivates a simultaneous consideration of area and speed.

Instead of considering performance alone, we now consider performance per unit area. The performance per unit area was measured in MIPS/LE for each benchmark executed on each processor (the same set of processors as above) and graphed in Figure 5.30. The bold line is the 3-stage pipeline with multiplier-based shifter which has the best performance per unit area out of all the processors (determined by comparing the arithmetic mean of performance per unit area across all benchmarks for each processor). The figure

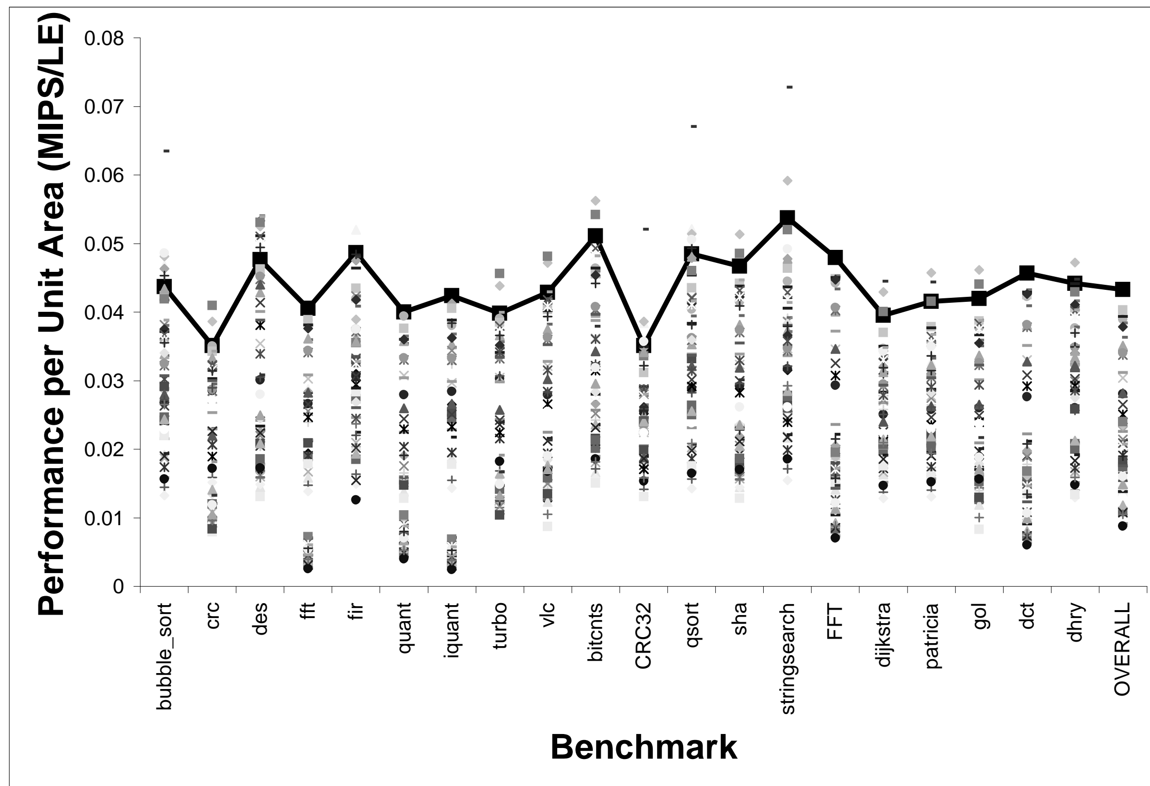


Figure 5.30: Performance per unit area of all processors on each benchmark—in bold is the best processor overall benchmarks.

shows much more promising results as the best overall processor is not the best processor for many benchmarks, in fact, only 6 of the 20 benchmarks achieve their highest performance per unit area using the best overall processor. The benchmarks STRINGSEARCH, QSORT, CRC32, and BUBBLE_SORT can achieve approximately 30% increased performance per unit area using processors other than the best overall processor. On average across the entire benchmark set, a benchmark can achieve 11.4% improved performance per unit area over the best overall processor. This indicates a strong motivation for making application-specific architectural decisions. We expect this number to grow with more interesting architectural axes but leave this for future work.

5.10 CAD Setting Independence

As discussed in Chapter 4, the optimization settings in the Quartus CAD software can potentially alter the design space terrain. Our hypothesis is that the results presented in this chapter hold across different CAD optimization settings. Some variation is expected, however, we believe that the conclusions drawn from this research will hold because the structure of the processors are very similar: all processors contain instruction fetch units, branch resolution, memory alignment, and other common components. Moreover, all components, including these common components are coded structurally to avoid potentially superfluous synthesis of logic functions from behavioural descriptions. As a result, the synthesis tool is more directed and becomes less sensitive to optimizations.

We have evaluated the effect of CAD settings on the two to five stage pipelines each with one of the three shifter implementations. Quartus was used to synthesize and place-and-route each design using three sets of optimization focusses: (i) the `default` focus which considers both area and speed; (ii) an `area` focus which sets all settings to minimize area and neglect speed (aggressive register packing, mux-restructuring on, gate and register duplication off); (iii) a `speed` focus which maximizes operating clock frequency by increasing area (register packing off, mux-restructuring off, gate and register duplication on). Details on the exact optimization settings used for each focus can be found in Appendix B.

Figure 5.31 and Figure 5.32 respectively show the area and clock frequency of each processor using the three different optimization focusses. The area of the `default` focus and `area` focus are co-linear proving that they track each other exactly, which is also true of the clock frequency measurements. This proves that in the processors there is little room for the `area` focussed compile to trade speed for area. A similar conclusion can be drawn about the `speed` focus, which increases area significantly but still tracks the area of the `default` focus and `area` focus. The clock frequency measurements also track each other without appreciable increase. We thus conclude that the speed optimizations attempted to increase the clock frequency using duplication of critical path logic but

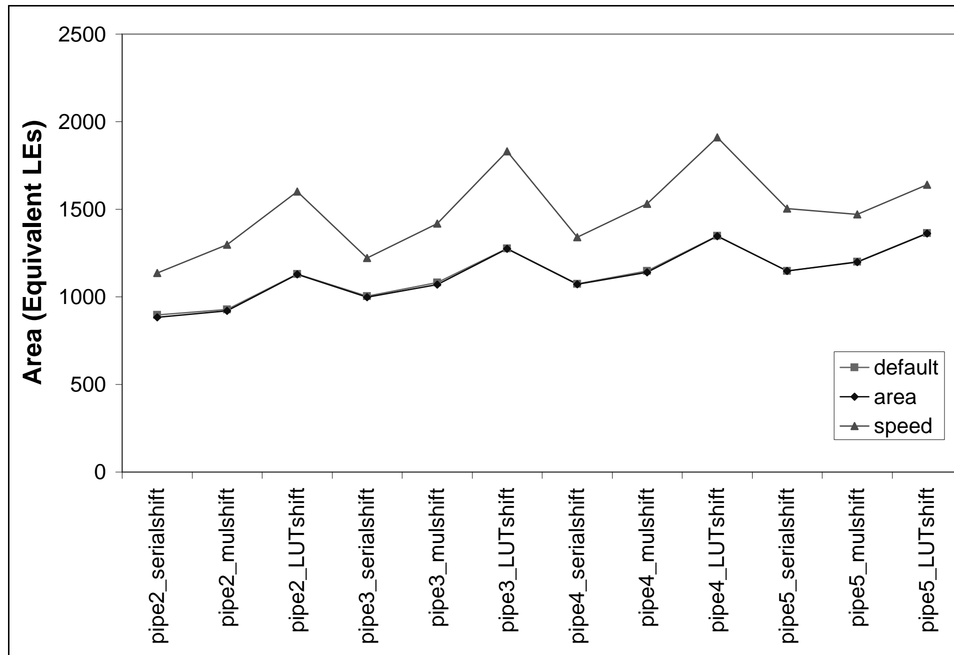


Figure 5.31: Effect of three different optimization focusses on area measurement. The `default` and `area` are colinear.

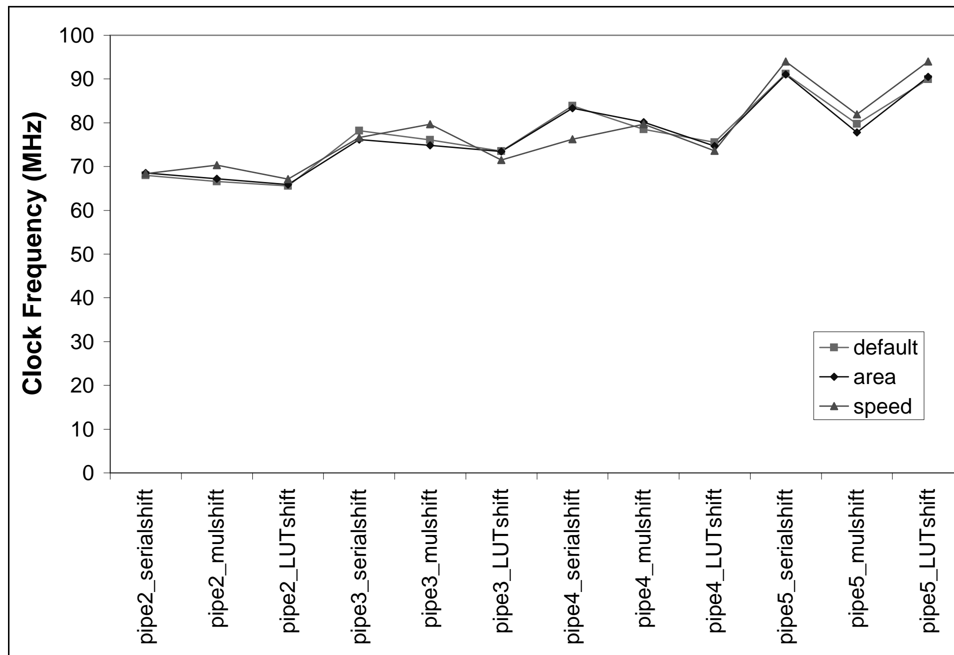


Figure 5.32: Effect of three different optimization focusses on clock frequency measurement.

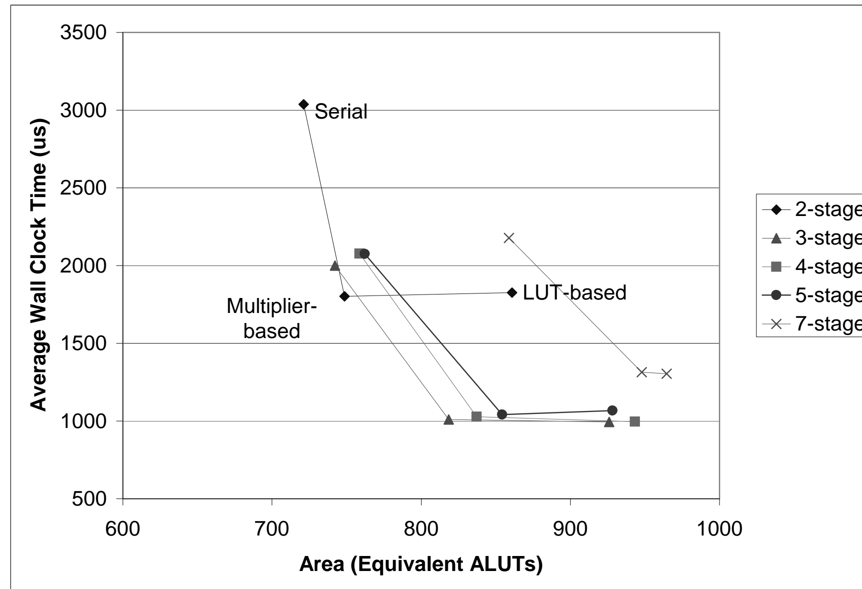


Figure 5.33: Average wall-clock-time vs area for different pipeline depths when implemented on Stratix II.

was unsuccessful. Since the different focusses track each other we maintain that the architectural conclusions do not vary significantly with different optimization settings. Moreover, since the optimizations were unsuccessful, we conclude that the processors are too small, simple, and efficiently coded to be sensitive to CAD optimizations.

5.11 Device Independence - Stratix vs Stratix II

The difference between ASIC and FPGA platforms is large enough that we are motivated to revisit the microarchitectural design space in an FPGA context. However, FPGA devices differ among themselves: across device families and vendors the resources and routing architecture on each FPGA vary greatly. We have focused on a single FPGA device, the Altera Stratix, to enable efficient synthesis through device-specific optimizations. Our hypothesis, is that in spite of differences in FPGA architecture, the conclusions drawn about soft processor architecture will be transferable between many FPGA families. In the future, we plan to investigate this across a range of different FPGA families. For now, we have migrated to Stratix II and observed that there is some noise in the architectural conclusions, but most of the conclusions still hold.

The fifteen pipelined processors (five different pipeline depths each with three different shifter implementations), were synthesized to Stratix II where measurements were made using the same experimental procedure with the exception of area: Stratix II has a different fundamental logic block so area is measured in terms of equivalent ALUTs instead of LEs. The wall-clock-time versus area plot in Figure 5.33 for the Stratix II[7, 31] is nearly identical to Figure 5.6 for the Stratix, except the LUT-based shifter is smaller in area as expected [6]. Further exploration into the device dependence of the soft processor architectural conclusions drawn here is needed. Due to the difficulty in migrating the Stratix-optimized SPREE Component Library to other devices, this task is left as future work.

Chapter 6

Conclusions

As FPGA-based soft processors are adapted more widely in embedded processing, we are motivated to understand the architectural trade-offs to maximize their efficiency. We have presented SPREE, an infrastructure for rapidly generating soft processors, and have analyzed the performance, area, and power of a broad space of interesting designs. We presented a rigorous method for comparing soft processors. We compared our generated processors to Altera's Nios II family of commercial soft processors and discovered a generated design which came within 15% of the smallest Nios II variation while outperforming it by 11%, while other generated processors both outperformed and were smaller than the standard Nios II variation.

Our initial exploration included varying hardware multiplication support, shifter implementations, pipeline depths and organization, and support for inter-stage forwarding. We found that a multiplier-based shifter is often the best, and that pipelining increases area but does not always increase performance. We observed that for a given pipeline depth, there still exist performance/area trade-offs for different placements of the pipeline stages. We also quantified the effect of inter-stage forwarding, and observed that one operand benefits significantly more from forwarding than the other. These observations have guided us towards interesting future research directions, including the addition of a closely-coupled compiler which would enable further avenues for optimization and cus-

tomization.

We have explored other more novel ideas such as instruction set subsetting where we witnessed dramatic area reductions, as well as other architectures which achieved area reduction through serialization and resource sharing. We also studied the effect of adding non-pipelined cycle latency to a pipeline, which we refer to as RISE, and found that it can provide an application specific tradeoff as well. Finally, we evaluated the fidelity of our architectural conclusions across different applications, CAD settings, and FPGA devices.

6.1 Contributions

This dissertation makes the following contributions:

1. an infrastructure for generating efficient RTL descriptions from text-based architectural descriptions;
2. a methodology for comparing and measuring soft processor architectures;
3. accurate area, clock frequency, and energy measurements for a wide variety of soft processors;
4. a comparison of Altera's Nios II soft processor variations against our automatically generated processors;
5. architectural features, component implementations, and potential compiler optimizations which are specific to FPGA-based soft processors.

6.2 Future Work

In the future, we plan to further validate the soft processor architectural conclusions drawn from SPREE by testing the fidelity of the conclusions across more FPGA devices and by implementing exception support to put SPREE generated soft processors

in the same class as other commercial embedded processors. We will further investigate power/energy and more power-aware architectural features. We also plan to broaden our architectural exploration space by including dynamic branch predictors, caches, more aggressive forwarding, VLIW datapaths, and other more advanced architectural features. In addition, we will explore compiler optimizations, ISA changes, custom instructions and hardware/software tradeoffs using SPREE. Finally, we will research and adopt different exploration methods since our exhaustive exploration strategy is not applicable to a more broad architectural space.

Appendix A

SPREE System Details

A.1 ISA Descriptions

In SPREE, the subset of the MIPS-I ISA supported is hardcoded as a graph of GENOPS. A text-based frontend is postponed as future work since we have fixed the ISA for this study. The descriptions of all supported instructions are shown below.

```
case MIPSOP_J:
{
    GenOp * op = new GenOp(GENOP_MERGE26LO);
    path->add_link(new GenOp(GENOP_PCREAD),0,op,0);
    path->add_link(ifetch,IF_INSTR_INDEX,op,1);
    path->add_link(op,0,new GenOp(GENOP_PCWRITEUNCOND),0);
    break;
}
case MIPSOP_JAL:
{
    GenOp * op = new GenOp(GENOP_MERGE26LO);
    path->add_link(new GenOp(GENOP_PCREAD),0,op);
    path->add_link(ifetch,IF_INSTR_INDEX,op,1);
    path->add_link(op,0,new GenOp(GENOP_PCWRITEUNCOND));

    GenOp * add = new GenOp(GENOP_ADDU);
    path->add_link(new GenOp(GENOP_PCREAD),0,add);
    GenOp * wb = new GenOp(GENOP_RFWRITE);
    path->add_link(add,0,wb);
    path->add_link(new GenOp(GENOP_CONST),31,wb,1);
    break;
}
case MIPSOP_BEQ:
case MIPSOP_BNE:
{
    GenOp * op = new GenOp(GENOP_ADDU);
    path->add_link(new GenOp(GENOP_PCREAD),0,op,0);
    GenOp * sext = new GenOp(GENOP_SIGNEXT16);
```

```
        path->add_link(ifetch,IF_OFFSET,sext);
        path->add_link(sext,0,op,1);

        GenOp * br_res = new GenOp(GENOP_BRANCHRESOLVE);
        GenOp * rs = new GenOp(GENOP_RFREAD);
        GenOp * rt = new GenOp(GENOP_RFREAD);
        path->add_link(ifetch,IF_RS,rs);
        path->add_link(rs,0,br_res);
        path->add_link(ifetch,IF_RT,rt);
        path->add_link(rt,0,br_res,1);

        GenOp * pc = new GenOp(GENOP_PCWRITE);
        path->add_link(op,0,pc);
        switch(mipsop)
        {
            case MIPSOP_BEQ: path->add_link(br_res,BR_EQ,pc,1); break;
            case MIPSOP_BNE: path->add_link(br_res,BR_NE,pc,1); break;
            default: break;    // silence g++ warnings
        }
        break;
    }
    case MIPSOP_BLEZ:
    case MIPSOP_BGTZ:
    {
        GenOp * op = new GenOp(GENOP_ADDU);
        path->add_link(new GenOp(GENOP_PCREAD),0,op,0);
        GenOp * sext = new GenOp(GENOP_SIGNEXT16);
        path->add_link(ifetch,IF_OFFSET,sext);
        path->add_link(sext,0,op,1);

        GenOp * br_res = new GenOp(GENOP_BRANCHRESOLVE);
        GenOp * rs = new GenOp(GENOP_RFREAD);
        path->add_link(ifetch,IF_RS,rs);
        path->add_link(rs,0,br_res);

        GenOp * pc = new GenOp(GENOP_PCWRITE);
        path->add_link(op,0,pc);
        switch(mipsop)
        {
            case MIPSOP_BLEZ: path->add_link(br_res,BR_LEZ,pc,1); break;
            case MIPSOP_BGTZ: path->add_link(br_res,BR_GTZ,pc,1); break;
            default: break;    // silence g++ warnings
        }
        break;
    }
    case MIPSOP_ADDI:
    case MIPSOP_ADDIU:
    case MIPSOP_SLTI:
    case MIPSOP_SLTIU:
    {
        GenOp * op;
        switch((MipsOp_t)mipsop)
        {
            case MIPSOP_ADDI: op=new GenOp(GENOP_ADD); break;
            case MIPSOP_ADDIU: op=new GenOp(GENOP_ADDU); break;
            case MIPSOP_SLTI: op=new GenOp(GENOP_SLT); break;
            case MIPSOP_SLTIU: op=new GenOp(GENOP_SLTU); break;
            default: break;    // silence g++ warnings
        }
```

```
        GenOp * rs = new GenOp(GENOP_RFREAD);
        GenOp * sext = new GenOp(GENOP_SIGNEXT16);
        path->add_link(ifetch,IF_RS,rs);
        path->add_link(rs,0,op);
        path->add_link(ifetch,IF_OFFSET,sext);
        path->add_link(sext,0,op,1);
        GenOp * wb = new GenOp(GENOP_RFWRITE);
        path->add_link(op,0,wb);
        path->add_link(ifetch,IF_RT,wb,1);
        break;
    }
    case MIPSOP_ANDI:
    case MIPSOP_ORI:
    case MIPSOP_XORI:
    {
        GenOp * op;
        switch((MipsOp_t)mipsop)
        {
            case MIPSOP_ANDI: op=new GenOp(GENOP_AND); break;
            case MIPSOP_ORI: op=new GenOp(GENOP_OR); break;
            case MIPSOP_XORI: op=new GenOp(GENOP_XOR); break;
            default: break;    // silence g++ warnings
        }
        GenOp * rs = new GenOp(GENOP_RFREAD);
        path->add_link(ifetch,IF_RS,rs);
        path->add_link(rs,0,op);
        path->add_link(ifetch,IF_OFFSET,op,1);
        GenOp * wb = new GenOp(GENOP_RFWRITE);
        path->add_link(op,0,wb);
        path->add_link(ifetch,IF_RT,wb,1);
        break;
    }
    case MIPSOP_LUI:
    {
        GenOp * op = new GenOp(GENOP_SHIFTLEFT);
        path->add_link(new GenOp(GENOP_CONST),16,op,1);
        path->add_link(ifetch,IF_OFFSET,op,0);
        GenOp * wb = new GenOp(GENOP_RFWRITE);
        path->add_link(op,0,wb);
        path->add_link(ifetch,IF_RT,wb,1);
        break;
    }
    case MIPSOP_LB:
    case MIPSOP_LH:
    case MIPSOP_LW:
    case MIPSOP_LBU:
    case MIPSOP_LHU:
    {
        GenOp * rs = new GenOp(GENOP_RFREAD);
        path->add_link(ifetch,IF_RS,rs);
        GenOp * sext = new GenOp(GENOP_SIGNEXT16);
        path->add_link(ifetch,IF_OFFSET,sext);
        GenOp * effaddr=new GenOp(GENOP_ADD);
        path->add_link(rs,0,effaddr);
        path->add_link(sext,0,effaddr,1);

        GenOp * mem;
        switch(mipsop){
            case MIPSOP_LB: mem=new GenOp(GENOP_LOADBYTE); break;
```

```
            case MIPSOP_LH: mem=new GenOp(GENOP_LOADHALF); break;
            case MIPSOP_LW: mem=new GenOp(GENOP_LOADWORD); break;
            case MIPSOP_LBU: mem=new GenOp(GENOP_LOADBYTEU); break;
            case MIPSOP_LHU: mem=new GenOp(GENOP_LOADHALFU); break;
            default: mem=NULL;
        }
        path->add_link(effaddr,0,mem);
        GenOp * wb = new GenOp(GENOP_RFWRITE);
        path->add_link(mem,0,wb);
        path->add_link(ifetch,IF_RT,wb,1);
        break;
    }
    case MIPSOP_SB:
    case MIPSOP_SH:
    case MIPSOP_SW:
    {
        GenOp * rs = new GenOp(GENOP_RFREAD);
        path->add_link(ifetch,IF_RS,rs);
        GenOp * sext = new GenOp(GENOP_SIGNEXT16);
        path->add_link(ifetch,IF_OFFSET,sext);
        GenOp * effaddr=new GenOp(GENOP_ADD);
        path->add_link(rs,0,effaddr);
        path->add_link(sext,0,effaddr,1);

        GenOp * mem;
        switch(mipsop){
            case MIPSOP_SB: mem=new GenOp(GENOP_STOREBYTE); break;
            case MIPSOP_SH: mem=new GenOp(GENOP_STOREHALF); break;
            case MIPSOP_SW: mem=new GenOp(GENOP_STOREWORD); break;
            default: mem=NULL;
        }
        path->add_link(effaddr,0,mem,1);
        GenOp * rt = new GenOp(GENOP_RFREAD);
        path->add_link(ifetch,IF_RT,rt);
        path->add_link(rt,0,mem);
        break;
    }

    case MIPSOP_SLL:
    case MIPSOP_SLLV:
    {
        GenOp * op = new GenOp(GENOP_SHIFTLEFT);
        if (mipsop==MIPSOP_SLLV)
        {
            GenOp * rs = new GenOp(GENOP_RFREAD);
            path->add_link(ifetch,IF_RS,rs);
            path->add_link(rs,0,op,1);
        }
        else
            path->add_link(ifetch,IF_SA,op,1);
        GenOp * rt = new GenOp(GENOP_RFREAD);
        path->add_link(ifetch,IF_RT,rt);
        path->add_link(rt,0,op,0);
        GenOp * wb = new GenOp(GENOP_RFWRITE);
        path->add_link(op,0,wb);
        path->add_link(ifetch,IF_RD,wb,1);
        break;
    }
    case MIPSOP_SRL:
```

```
case MIPSOP_SRA:
case MIPSOP_SRLV:
case MIPSOP_SRAV:
{
    GenOp * op;
    switch(mipsop)
    {
        case MIPSOP_SRL:
        case MIPSOP_SRLV:
            op = new GenOp(GENOP_SHIFTRIGHTLOGIC); break;
        case MIPSOP_SRA:
        case MIPSOP_SRAV:
            op = new GenOp(GENOP_SHIFTRIGHTARITH); break;
        default: op=NULL;
    }
    if (mipsop==MIPSOP_SRLV || mipsop==MIPSOP_SRAV)
    {
        GenOp * rs = new GenOp(GENOP_RFREAD);
        path->add_link(ifetch,IF_RS,rs);
        path->add_link(rs,0,op,1);
    }
    else
        path->add_link(ifetch,IF_SA,op,1);
    GenOp * rt = new GenOp(GENOP_RFREAD);
    path->add_link(ifetch,IF_RT,rt);
    path->add_link(rt,0,op,0);
    GenOp * wb = new GenOp(GENOP_RFWRITE);
    path->add_link(op,0,wb);
    path->add_link(ifetch,IF_RD,wb,1);
    break;
}
case MIPSOP_JR:
{
    GenOp * rs = new GenOp(GENOP_RFREAD);
    path->add_link(ifetch,IF_RS,rs);
    path->add_link(rs,0,new GenOp(GENOP_PCWRITEUNCOND),0);
    break;
}
case MIPSOP_JALR:
{
    GenOp * rs = new GenOp(GENOP_RFREAD);
    path->add_link(ifetch,IF_RS,rs);
    path->add_link(rs,0,new GenOp(GENOP_PCWRITEUNCOND),0);

    GenOp * add = new GenOp(GENOP_ADDU);
    path->add_link(new GenOp(GENOP_PCREAD),0,add);
    GenOp * wb = new GenOp(GENOP_RFWRITE);
    path->add_link(add,0,wb);
    path->add_link(new GenOp(GENOP_CONST),31,wb,1);
    break;
}
case MIPSOP_MFHI:
{
    GenOp * wb = new GenOp(GENOP_RFWRITE);
    path->add_link(new GenOp(GENOP_HIREAD),0,wb);
    path->add_link(ifetch,IF_RD,wb,1);
    break;
}
case MIPSOP_MTHI:
```

```
{
    GenOp * rs = new GenOp(GENOP_RFREAD);
    path->add_link(ifetch,IF_RS,rs);
    path->add_link(rs,0,new GenOp(GENOP_HIWRITE));
    break;
}
case MIPSOP_MFLO:
{
    GenOp * wb = new GenOp(GENOP_RFWRITE);
    path->add_link(new GenOp(GENOP_LOREAD),0,wb);
    path->add_link(ifetch,IF_RD,wb,1);
    break;
}
case MIPSOP_MTLO:
{
    GenOp * rs = new GenOp(GENOP_RFREAD);
    path->add_link(ifetch,IF_RS,rs);
    path->add_link(rs,0,new GenOp(GENOP_LOWRITE));
    break;
}
case MIPSOP_MULT:
case MIPSOP_MULTU:
{
    GenOp * op;
    switch((MipsOp_t)mipsop)
    {
        case MIPSOP_MULT: op=new GenOp(GENOP_MULT); break;
        case MIPSOP_MULTU: op=new GenOp(GENOP_MULTU); break;
        default: break;    // silence g++ warnings
    }
    GenOp * rs = new GenOp(GENOP_RFREAD);
    GenOp * rt = new GenOp(GENOP_RFREAD);
    path->add_link(ifetch,IF_RS,rs);
    path->add_link(ifetch,IF_RT,rt);
    path->add_link(rs,0,op);
    path->add_link(rt,0,op,1);
    path->add_link(op,1,new GenOp(GENOP_HIWRITE));
    path->add_link(op,0,new GenOp(GENOP_LOWRITE));
    break;
}
case MIPSOP_DIV:
case MIPSOP_DIVU:
{
    GenOp * op;
    switch((MipsOp_t)mipsop)
    {
        case MIPSOP_DIV: op=new GenOp(GENOP_DIV); break;
        case MIPSOP_DIVU: op=new GenOp(GENOP_DIVU); break;
        default: break;    // silence g++ warnings
    }
    GenOp * rs = new GenOp(GENOP_RFREAD);
    GenOp * rt = new GenOp(GENOP_RFREAD);
    path->add_link(ifetch,IF_RS,rs);
    path->add_link(ifetch,IF_RT,rt);
    path->add_link(rs,0,op);
    path->add_link(rt,0,op,1);
    path->add_link(op,1,new GenOp(GENOP_HIWRITE));
    path->add_link(op,0,new GenOp(GENOP_LOWRITE));
    break;
```

```
}
case MIPSOP_ADD:
case MIPSOP_ADDU:
case MIPSOP_SUB:
case MIPSOP_SUBU:
case MIPSOP_AND:
case MIPSOP_OR:
case MIPSOP_XOR:
case MIPSOP_NOR:
case MIPSOP_SLT:
case MIPSOP_SLTU:
{
    GenOp * op;
    switch((MipsOp_t)mipsop)
    {
        case MIPSOP_ADD: op=new GenOp(GENOP_ADD); break;
        case MIPSOP_ADDU: op=new GenOp(GENOP_ADDU); break;
        case MIPSOP_SUB: op=new GenOp(GENOP_SUB); break;
        case MIPSOP_SUBU: op=new GenOp(GENOP_SUBU); break;
        case MIPSOP_SLT: op=new GenOp(GENOP_SLT); break;
        case MIPSOP_SLTU: op=new GenOp(GENOP_SLTU); break;
        case MIPSOP_AND: op=new GenOp(GENOP_AND); break;
        case MIPSOP_OR: op=new GenOp(GENOP_OR); break;
        case MIPSOP_XOR: op=new GenOp(GENOP_XOR); break;
        case MIPSOP_NOR: op=new GenOp(GENOP_NOR); break;
        default: break;    // silence g++ warnings
    }
    GenOp * rs = new GenOp(GENOP_RFREAD);
    GenOp * rt = new GenOp(GENOP_RFREAD);
    path->add_link(ifetch,IF_RS,rs);
    path->add_link(rs,0,op);
    path->add_link(ifetch,IF_RT,rt);
    path->add_link(rt,0,op,1);
    GenOp * wb = new GenOp(GENOP_RFWRITE);
    path->add_link(op,0,wb);
    path->add_link(ifetch,IF_RD,wb,1);
    break;
}
case MIPSOP_BLTZ:
case MIPSOP_BGEZ:
{
    GenOp * op = new GenOp(GENOP_ADDU);
    path->add_link(new GenOp(GENOP_PCREAD),0,op,0);
    GenOp * sext = new GenOp(GENOP_SIGNEXT16);
    path->add_link(ifetch,IF_OFFSET,sext);
    path->add_link(sext,0,op,1);

    GenOp * br_res = new GenOp(GENOP_BRANCHRESOLVE);
    GenOp * rs = new GenOp(GENOP_RFREAD);
    path->add_link(ifetch,IF_RS,rs);
    path->add_link(rs,0,br_res);

    GenOp * pc = new GenOp(GENOP_PCWRITE);
    path->add_link(op,0,pc);
    switch(mipsop)
    {
        case MIPSOP_BLTZ: path->add_link(br_res,BR_LTZ,pc,1); break;
        case MIPSOP_BGEZ: path->add_link(br_res,BR_GEZ,pc,1); break;
        default: break;    // silence g++ warnings
```

```
        }
        break;
    }

    case MIPSOP_BLTZAL:
    case MIPSOP_BGEZAL:
    {
        GenOp * op = new GenOp(GENOP_ADDU);
        GenOp * pcread=new GenOp(GENOP_PCREAD);
        path->add_link(pcread,0,op,0);
        GenOp * sext = new GenOp(GENOP_SIGNEXT16);
        path->add_link(ifetch,IF_OFFSET,sext);
        path->add_link(sext,0,op,1);

        GenOp * add8 = new GenOp(GENOP_ADDU);
        path->add_link(pcread,0,add8,0);
        path->add_link(new GenOp(GENOP_CONST),4,add8,0);

        GenOp * br_res = new GenOp(GENOP_BRANCHRESOLVE);
        GenOp * rs = new GenOp(GENOP_RFREAD);
        path->add_link(ifetch,IF_RS,rs);
        path->add_link(rs,0,br_res);

        int br_port;
        switch(mipsop)
        {
            case MIPSOP_BLTZ: br_port=(int)BR_LTZ; break;
            case MIPSOP_BGEZ: br_port=(int)BR_GEZ; break;
            default: break;    // silence g++ warnings
        }

        GenOp * pc = new GenOp(GENOP_PCWRITE);
        path->add_link(op,0,pc);
        path->add_link(br_res,br_port,pc,1);

        GenOp * wb = new GenOp(GENOP_RFWRITE);
        path->add_link(add8,0,wb);
        path->add_link(new GenOp(GENOP_CONST),31,wb,1);
        path->add_link(br_res,br_port,wb,2);
        break;
    }
```

A.2 Datapath Descriptions

Some examples of datapath descriptions are shown in this section. Below are the complete structural descriptions of three processors: a 2-stage pipeline with serial shifter, a 3-stage pipeline with multiplier-based shifter, and a 5-stage pipeline with LUT-based barrel shifter and forwarding. A simple text-based frontend can be made for these descriptions.

A.2.1 2-stage Pipeline with Serial Shifter

```
/********************** Component List **********************/

RTLComponent *addersub=new RTLComponent("addersub","slt");
RTLComponent *logic_unit=new RTLComponent("logic_unit");
RTLComponent *shifterbarrel=new RTLComponent("shifter","perbit");
RTLComponent *mul=new RTLComponent("mul");
RTLComponent *hi_reg=new RTLComponent("hi_reg");
RTLComponent *lo_reg=new RTLComponent("lo_reg");
RTLComponent *reg_file=new RTLComponent("reg_file");
RTLComponent *ifetch=new RTLComponent("ifetch");
RTLComponent *pcadder=new RTLComponent("pcadder");
RTLComponent *signext=new RTLComponent("signext16");
RTLComponent *merge26lo=new RTLComponent("merge26lo");
RTLComponent *data_mem=new RTLComponent("data_mem","");
RTLComponent *branchresolve=new RTLComponent("branchresolve");

RTLComponent *nop_opB=new RTLComponent("nop");

RTLParameters const8_params;
const8_params["VAL"]="4";
RTLComponent *const8=new RTLComponent("const",const8_params);

RTLParameters const16_params;
const16_params["VAL"]="16";
RTLComponent *const16=new RTLComponent("const",const16_params);

RTLParameters const31_params;
const31_params["VAL"]="31";
RTLComponent *const31=new RTLComponent("const",const31_params);


/********************** Datapath Wiring **********************/

RTLProc proc("system");

proc.addConnection(ifetch,"rs",reg_file,"a_reg");
proc.addConnection(ifetch,"rt",reg_file,"b_reg");

        // Conditional Branch path
proc.addConnection(ifetch,"offset",signext,"in");
proc.addConnection(signext,"out",pcadder,"offset");
proc.addConnection(ifetch,"pc_out",pcadder,"pc");
proc.addConnection(pcadder,"result",ifetch,"load_data");
proc.addConnection(branchresolve,"eq",ifetch,"load");
proc.addConnection(branchresolve,"ne",ifetch,"load");
proc.addConnection(branchresolve,"lez",ifetch,"load");
proc.addConnection(branchresolve,"ltz",ifetch,"load");
proc.addConnection(branchresolve,"gez",ifetch,"load");
proc.addConnection(branchresolve,"gtz",ifetch,"load");

        // J and JAL path
proc.addConnection(ifetch,"pc_out",merge26lo,"in1");
proc.addConnection(ifetch,"instr_index",merge26lo,"in2");
proc.addConnection(merge26lo,"out",ifetch,"load_data");
proc.addConnection(ifetch,"pc_out",addersub,"opA");
```

```
        // JR and JALR path
proc.addConnection(reg_file,"a_readdataout",ifetch,"load_data");

        // Other IR constant fanouts
proc.addConnection(signext,"out",nop_opB,"d");
proc.addConnection(ifetch,"offset",nop_opB,"d");
proc.addConnection(ifetch,"sa",shifterbarrel,"sa");

        // RS fanout
proc.addConnection(reg_file,"a_readdataout",addersub,"opA");
proc.addConnection(reg_file,"a_readdataout",logic_unit,"opA");
proc.addConnection(reg_file,"a_readdataout",shifterbarrel,"sa");
proc.addConnection(reg_file,"a_readdataout",mul,"opA");
proc.addConnection(reg_file,"a_readdataout",branchresolve,"rs");

        // RT fanout
proc.addConnection(reg_file,"b_readdataout",nop_opB,"d");
proc.addConnection(reg_file,"b_readdataout",mul,"opB");
proc.addConnection(reg_file,"b_readdataout",data_mem,"d_writedata");
proc.addConnection(reg_file,"b_readdataout",branchresolve,"rt");

proc.addConnection(nop_opB,"q",addersub,"opB");
proc.addConnection(nop_opB,"q",logic_unit,"opB");
proc.addConnection(nop_opB,"q",shifterbarrel,"opB");

        // MUL
proc.addConnection(mul,"hi",hi_reg,"d");
proc.addConnection(mul,"lo",lo_reg,"d");

        // Data memory
proc.addConnection(addersub,"result",data_mem,"d_address");

        // Adder
proc.addControlConnection(const8,"out",nop_opB,"d");

        // Shifter
proc.addConnection(const16,"out",shifterbarrel,"sa");

        // Writeback
proc.addConnection(addersub,"result",reg_file,"c_writedatain");
proc.addConnection(addersub,"result_slt",reg_file,"c_writedatain");
proc.addConnection(logic_unit,"result",reg_file,"c_writedatain");
proc.addConnection(shifterbarrel,"result",reg_file,"c_writedatain");
proc.addConnection(data_mem,"d_loadresult",reg_file,"c_writedatain");
proc.addConnection(hi_reg,"q",reg_file,"c_writedatain");
proc.addConnection(lo_reg,"q",reg_file,"c_writedatain");

        // Writeback destination
proc.addConnection(ifetch,"rt",reg_file,"c_reg");
proc.addConnection(ifetch,"rd",reg_file,"c_reg");
proc.addConnection(const31,"out",reg_file,"c_reg");


/********************** Controller **********************/
CtrlUnPipelined control(&fdp,&proc);
```

A.2.2 3-stage Pipeline with Multiplier-based Shifter

```
/******
 * Pipepelined Processor
 *
 * F - R/E - W
 ******/

/********************** Component List **********************/

RTLComponent *addersub=new RTLComponent("addersub","slt");
RTLComponent *logic_unit=new RTLComponent("logic_unit");
RTLComponent *mul=new RTLComponent("mul","shift_stall");

RTLComponent *ifetch=new RTLComponent("ifetch","pipe");
RTLComponent *data_mem=new RTLComponent("data_mem","stall");
RTLComponent *reg_file=new RTLComponent("reg_file","pipe");

RTLComponent *pcadder=new RTLComponent("pcadder");
RTLComponent *signext=new RTLComponent("signext16");
RTLComponent *merge26lo=new RTLComponent("merge26lo");
RTLComponent *branchresolve=new RTLComponent("branchresolve");
RTLComponent *hi_reg=new RTLComponent("hi_reg");
RTLComponent *lo_reg=new RTLComponent("lo_reg");

RTLParameters const8_params;
const8_params["VAL"]="0";
RTLComponent *const8=new RTLComponent("const",const8_params);

RTLParameters const16_params;
const16_params["VAL"]="16";
RTLComponent *const16=new RTLComponent("const",const16_params);

RTLParameters const31_params;
const31_params["VAL"]="31";
RTLComponent *const31=new RTLComponent("const",const31_params);

RTLProc proc("system");

/********************** Registers **********************/

RTLParameters width5_params;
width5_params["WIDTH"]="5";
RTLParameters width26_params;
width26_params["WIDTH"]="26";

RTLComponent *offset_reg1=new RTLComponent("pipereg");
RTLComponent *instr_index_reg1=new RTLComponent("pipereg",width26_params);
RTLComponent *sa_reg1=new RTLComponent("pipereg",width5_params);
RTLComponent *dst_reg1=new RTLComponent("pipereg",width5_params);
RTLComponent *pc_reg1=new RTLComponent("pipereg");

RTLComponent *pc_fakereg1=new RTLComponent("fakedelay");

/********************** Other Components **********************/

RTLComponent *nop_opB=new RTLComponent("nop");
RTLComponent *rs_zeroer=new RTLComponent("zeroer",width5_params);
RTLComponent *rt_zeroer=new RTLComponent("zeroer",width5_params);
RTLComponent *rd_zeroer=new RTLComponent("zeroer",width5_params);
```

```
/********************** Stage Requests **********************/
proc.putInStage(const31,"op",1);
proc.putInStage(const16,"op",2);
proc.putInStage(const8,"op",2);
proc.putInStage(hi_reg,"op",2);
proc.putInStage(lo_reg,"op",2);
proc.putInStage(ifetch,"pcreadop",1);
proc.putInStage(addersub,"op",2);

/********************** Datapath Wiring **********************/

proc.addConnection(ifetch,"rs",rs_zeroer,"d");
proc.addConnection(ifetch,"rt",rt_zeroer,"d");
proc.addConnection(rs_zeroer,"q",reg_file,"a_reg");
proc.addConnection(rt_zeroer,"q",reg_file,"b_reg");

        // Stage 1 pipe registers
proc.addConnection(ifetch,"offset",offset_reg1,"d");
proc.addConnection(signext,"out",offset_reg1,"d");
proc.addConnection(ifetch,"instr_index",instr_index_reg1,"d");
proc.addConnection(ifetch,"sa",sa_reg1,"d");
proc.addConnection(ifetch,"pc_out",pc_reg1,"d");

proc.addConnection(ifetch,"rt",rd_zeroer,"d");
proc.addConnection(ifetch,"rd",rd_zeroer,"d");
proc.addConnection(const31,"out",rd_zeroer,"d");
proc.addConnection(rd_zeroer,"q",dst_reg1,"d");

        // Conditional Branch path
proc.addConnection(ifetch,"offset",signext,"in");
proc.addConnection(offset_reg1,"q",pcadder,"offset");
proc.addConnection(pc_reg1,"q",pcadder,"pc");
proc.addConnection(pcadder,"result",ifetch,"load_data");

proc.addConnection(branchresolve,"eq",ifetch,"load");
proc.addConnection(branchresolve,"ne",ifetch,"load");
proc.addConnection(branchresolve,"lez",ifetch,"load");
proc.addConnection(branchresolve,"ltz",ifetch,"load");
proc.addConnection(branchresolve,"gez",ifetch,"load");
proc.addConnection(branchresolve,"gtz",ifetch,"load");

        // J and JAL path
proc.addConnection(pc_reg1,"q",merge26lo,"in1");
proc.addConnection(instr_index_reg1,"q",merge26lo,"in2");
proc.addConnection(merge26lo,"out",ifetch,"load_data");

proc.addConnection(ifetch,"pc_out",pc_fakereg1,"d");
proc.addConnection(pc_fakereg1,"q",addersub,"opA");

        // JR and JALR path
proc.addConnection(reg_file,"a_readdataout",ifetch,"load_data");

        // Other IR constant fanouts
proc.addConnection(offset_reg1,"q",nop_opB,"d");

        // RS fanout
proc.addConnection(reg_file,"a_readdataout",addersub,"opA");
proc.addConnection(reg_file,"a_readdataout",logic_unit,"opA");
proc.addConnection(reg_file,"a_readdataout",mul,"sa");
```

```
proc.addConnection(reg_file,"a_readdataout",mul,"opA");
proc.addConnection(reg_file,"a_readdataout",branchresolve,"rs");

        // RT fanout
proc.addConnection(reg_file,"b_readdataout",nop_opB,"d");
proc.addConnection(reg_file,"b_readdataout",mul,"opB");
proc.addConnection(reg_file,"b_readdataout",data_mem,"d_writedata");
proc.addConnection(reg_file,"b_readdataout",branchresolve,"rt");
//proc.addConnection(reg_file,"b_readdataout",data_mem,"d_writedata");

proc.addConnection(nop_opB,"q",addersub,"opB");
proc.addConnection(nop_opB,"q",logic_unit,"opB");
proc.addConnection(nop_opB,"q",mul,"opA");

        // MUL
proc.addConnection(mul,"hi",hi_reg,"d");
proc.addConnection(mul,"lo",lo_reg,"d");
proc.addControlConnection(dst_reg1,"q",mul,"dst");

        // Data memory
proc.addConnection(addersub,"result",data_mem,"d_address");

        // Adder
proc.addControlConnection(const8,"out",nop_opB,"d");

        // Shifter
proc.addConnection(const16,"out",mul,"sa");
proc.addConnection(sa_reg1,"q",mul,"sa");

        // Writeback
proc.addConnection(addersub,"result",reg_file,"c_writedatain");
proc.addConnection(addersub,"result_slt",reg_file,"c_writedatain");
proc.addConnection(logic_unit,"result",reg_file,"c_writedatain");
proc.addConnection(mul,"shift_result",reg_file,"c_writedatain");
proc.addConnection(data_mem,"d_loadresult",reg_file,"c_writedatain");
proc.addConnection(hi_reg,"q",reg_file,"c_writedatain");
proc.addConnection(lo_reg,"q",reg_file,"c_writedatain");

        // Writeback destination
proc.addConnection(dst_reg1,"q",reg_file,"c_reg");


/********************** Controller **********************/
PipelineOptions options;
options.register_opcodes=false;
CtrlPipelined control(&fdp,&proc,options);

/********************** Hazard detection **********************/
HazardDetector *rs_haz=control.newHazardDetector(rs_zeroer,"q",dst_reg1,"q");
HazardDetector *rt_haz=control.newHazardDetector(rt_zeroer,"q",dst_reg1,"q");
control.stallOnHazard(rs_haz,1);
control.stallOnHazard(rt_haz,1);
```

A.2.3 5-stage Pipeline with LUT-based Shifter and Forwarding

```
/******
 * Pipepelined Processor
```

```
*
* F − R/E − W
******/

    /********************** Component List **********************/

    RTLComponent *addersub=new RTLComponent("addersub","slt_1");
    RTLComponent *logic_unit=new RTLComponent("logic_unit");
    RTLComponent *shifter=new RTLComponent("shifter","barrel_pipe1");
    RTLComponent *mul=new RTLComponent("mul","1");

    RTLComponent *ifetch=new RTLComponent("ifetch","pipe");
    RTLComponent *data_mem=new RTLComponent("data_mem","stall");
    RTLComponent *reg_file=new RTLComponent("reg_file","pipe");

    RTLComponent *pcadder=new RTLComponent("pcadder");
    RTLComponent *signext=new RTLComponent("signext16");
    RTLComponent *merge26lo=new RTLComponent("merge26lo");
    RTLComponent *branchresolve=new RTLComponent("branchresolve");
    RTLComponent *hi_reg=new RTLComponent("hi_reg");
    RTLComponent *lo_reg=new RTLComponent("lo_reg");

    RTLParameters const8_params;
    const8_params["VAL"]="0";
    RTLComponent *const8=new RTLComponent("const",const8_params);

    RTLParameters const16_params;
    const16_params["VAL"]="16";
    RTLComponent *const16=new RTLComponent("const",const16_params);

    RTLParameters const31_params;
    const31_params["VAL"]="31";
    RTLComponent *const31=new RTLComponent("const",const31_params);

    RTLProc proc("system");

    /********************** Registers **********************/

    RTLParameters width6_params;
    width6_params["WIDTH"]="6";
    RTLParameters width5_params;
    width5_params["WIDTH"]="5";
    RTLParameters width26_params;
    width26_params["WIDTH"]="26";
    RTLParameters width16_params;
    width16_params["WIDTH"]="16";

    RTLComponent *rt_reg0=new RTLComponent("pipereg",width5_params);
    RTLComponent *rd_reg0=new RTLComponent("pipereg",width5_params);
    RTLComponent *rs_reg0=new RTLComponent("pipereg",width5_params);
    RTLComponent *offset_reg0=new RTLComponent("pipereg",width16_params);
    RTLComponent *sa_reg0=new RTLComponent("pipereg",width5_params);
    RTLComponent *instr_index_reg0=new RTLComponent("pipereg",width26_params);
    RTLComponent *pc_reg0=new RTLComponent("pipereg");


    RTLComponent *offset_reg1=new RTLComponent("pipereg");
    RTLComponent *instr_index_reg1=new RTLComponent("pipereg",width26_params);
    RTLComponent *sa_reg1=new RTLComponent("pipereg",width5_params);
```

```
RTLComponent *pc_reg1=new RTLComponent("pipereg");

RTLComponent *logic_reg2=new RTLComponent("pipereg");
RTLComponent *storedata_reg2=new RTLComponent("pipereg");

RTLComponent *dst_reg1=new RTLComponent("pipereg",width5_params);
RTLComponent *dst_reg2=new RTLComponent("pipereg",width5_params);

RTLComponent *result_reg3=new RTLComponent("pipereg");

RTLComponent *pc_fakereg1=new RTLComponent("fakedelay");

/********************** Other Components **********************/

RTLComponent *nop_rs=new RTLComponent("nop");
RTLComponent *nop_rt=new RTLComponent("nop");
RTLComponent *nop_result=new RTLComponent("nop");

RTLComponent *nop_opB=new RTLComponent("nop");
RTLComponent *rs_zeroer=new RTLComponent("zeroer",width5_params);
RTLComponent *rt_zeroer=new RTLComponent("zeroer",width5_params);
RTLComponent *rd_zeroer=new RTLComponent("zeroer",width5_params);

/********************** Stage Requests **********************/
proc.putInStage(const31,"op",2);
proc.putInStage(const16,"op",3);
proc.putInStage(const8,"op",3);
proc.putInStage(hi_reg,"op",4);
proc.putInStage(lo_reg,"op",4);
proc.putInStage(ifetch,"pcreadop",1);

/********************** Datapath Wiring **********************/

        //Stage 0 pipe registers
proc.addConnection(ifetch,"rt",rt_reg0,"d");
proc.addConnection(ifetch,"rs",rs_reg0,"d");
proc.addConnection(ifetch,"rd",rd_reg0,"d");
proc.addConnection(ifetch,"sa",sa_reg0,"d");
proc.addConnection(ifetch,"offset",offset_reg0,"d");
proc.addConnection(ifetch,"instr_index",instr_index_reg0,"d");
proc.addConnection(ifetch,"pc_out",pc_reg0,"d");

proc.addConnection(rs_reg0,"q",rs_zeroer,"d");
proc.addConnection(rt_reg0,"q",rt_zeroer,"d");
proc.addConnection(rs_zeroer,"q",reg_file,"a_reg");
proc.addConnection(rt_zeroer,"q",reg_file,"b_reg");

        // Stage 1 pipe registers
proc.addConnection(offset_reg0,"q",offset_reg1,"d");
proc.addConnection(signext,"out",offset_reg1,"d");
proc.addConnection(instr_index_reg0,"q",instr_index_reg1,"d");
proc.addConnection(sa_reg0,"q",sa_reg1,"d");
proc.addConnection(pc_reg0,"q",pc_reg1,"d");

proc.addConnection(rt_reg0,"q",rd_zeroer,"d");
proc.addConnection(rd_reg0,"q",rd_zeroer,"d");
proc.addConnection(const31,"out",rd_zeroer,"d");
proc.addConnection(rd_zeroer,"q",dst_reg1,"d");
```

```
        // Conditional Branch path
proc.addConnection(offset_reg0,"q",signext,"in");
proc.addConnection(offset_reg1,"q",pcadder,"offset");
proc.addConnection(pc_reg1,"q",pcadder,"pc");
proc.addConnection(pcadder,"result",ifetch,"load_data");

proc.addConnection(branchresolve,"eq",ifetch,"load");
proc.addConnection(branchresolve,"ne",ifetch,"load");
proc.addConnection(branchresolve,"lez",ifetch,"load");
proc.addConnection(branchresolve,"ltz",ifetch,"load");
proc.addConnection(branchresolve,"gez",ifetch,"load");
proc.addConnection(branchresolve,"gtz",ifetch,"load");

        // J and JAL path
proc.addConnection(pc_reg1,"q",merge26lo,"in1");
proc.addConnection(instr_index_reg1,"q",merge26lo,"in2");
proc.addConnection(merge26lo,"out",ifetch,"load_data");

proc.addConnection(pc_reg0,"q",pc_fakereg1,"d");
proc.addConnection(pc_fakereg1,"q",addersub,"opA");

        // JR and JALR path
proc.addConnection(nop_rs,"q",ifetch,"load_data");

        // Other IR constant fanouts
proc.addConnection(offset_reg1,"q",nop_opB,"d");

        // RS fanout
proc.addConnection(reg_file,"a_readdataout",nop_rs,"d");

proc.addConnection(nop_rs,"q",addersub,"opA");
proc.addConnection(nop_rs,"q",logic_unit,"opA");
proc.addConnection(nop_rs,"q",shifter,"sa");
proc.addConnection(nop_rs,"q",mul,"opA");
proc.addConnection(nop_rs,"q",branchresolve,"rs");

        // RT fanout
proc.addConnection(reg_file,"b_readdataout",nop_rt,"d");

proc.addConnection(nop_rt,"q",nop_opB,"d");
proc.addConnection(nop_rt,"q",mul,"opB");
proc.addConnection(nop_rt,"q",storedata_reg2,"d");
proc.addConnection(nop_rt,"q",branchresolve,"rt");
//proc.addConnection(nop_rt,"q",data_mem,"d_writedata");

proc.addConnection(nop_opB,"q",addersub,"opB");
proc.addConnection(nop_opB,"q",logic_unit,"opB");
proc.addConnection(nop_opB,"q",shifter,"opB");

        // MUL
proc.addConnection(mul,"hi",hi_reg,"d");
proc.addConnection(mul,"lo",lo_reg,"d");

        // Data memory
proc.addConnection(addersub,"result",data_mem,"d_address");
proc.addConnection(storedata_reg2,"q",data_mem,"d_writedata");

        // Adder
proc.addControlConnection(const8,"out",nop_opB,"d");
```

```
        // Logic Unit
    proc.addConnection(logic_unit,"result",logic_reg2,"d");

        // Shifter
    proc.addConnection(const16,"out",shifter,"sa");
    proc.addConnection(sa_reg1,"q",shifter,"sa");

        // Writeback
    proc.addConnection(addersub,"result",nop_result,"d");
    proc.addConnection(addersub,"result_slt",nop_result,"d");
    proc.addConnection(logic_reg2,"q",nop_result,"d");
    proc.addConnection(shifter,"result",nop_result,"d");
    proc.addConnection(data_mem,"d_loadresult",nop_result,"d");
    proc.addConnection(hi_reg,"q",nop_result,"d");
    proc.addConnection(lo_reg,"q",nop_result,"d");

    proc.addConnection(nop_result,"q",reg_file,"c_writedatain");

        // Writeback destination
    proc.addConnection(dst_reg1,"q",dst_reg2,"d");
    proc.addConnection(dst_reg2,"q",reg_file,"c_reg");


  /********************** Controller **********************/
  PipelineOptions options;
  options.register_opcodes=false;
  CtrlPipelined control(&fdp,&proc,options);

  proc.addControlConnection(nop_result,"q",result_reg3,"d");
  control.insertPipeReg(result_reg3,2);

  /********************** Hazard detection **********************/
  HazardDetector *rs_haz1=control.newHazardDetector(rs_zeroer,"q",dst_reg1,"q");
  HazardDetector *rt_haz1=control.newHazardDetector(rt_zeroer,"q",dst_reg1,"q");
  control.stallOnHazard(rs_haz1,2);
  control.stallOnHazard(rt_haz1,2);

  HazardDetector *rs_haz2=control.newHazardDetector(rs_zeroer,"q",dst_reg2,"q");
  HazardDetector *rt_haz2=control.newHazardDetector(rt_zeroer,"q",dst_reg2,"q");
  control.bypassOnHazard(rs_haz2,3,result_reg3,"q",nop_rs,"d");
  control.bypassOnHazard(rt_haz2,3,result_reg3,"q",nop_rt,"d");
```

A.3 The Library Entry

The library entry is a textual description of a component's functionality and interface formatted as in Figure A.1; an example of a component description is seen in Figure A.2 for a simplified arithmetic unit. The `Module` keyword starts a new description of a component and is followed by the module's name. The name is separated by the first underscore into a base component name and a version name. The top-level name of the Verilog module must take the name of the base component. In the automatically gen-

```
Module <base component name>_<version name> {
  File <Verilog source file>
  Parameter <param name> <parameter width>
  ...
  Input <port name> <port width>
  ...
  Output <port name> <port width>
  ...
  Opcode <port name> <port width> [en <enable port name>] [squash <squash port name>] {
    <GENOP> <opcode value> <latency> <port mapping>
    ...
  }
  ...
  [clk]
  [resetn]
}
```

Figure A.1: Library entry format.

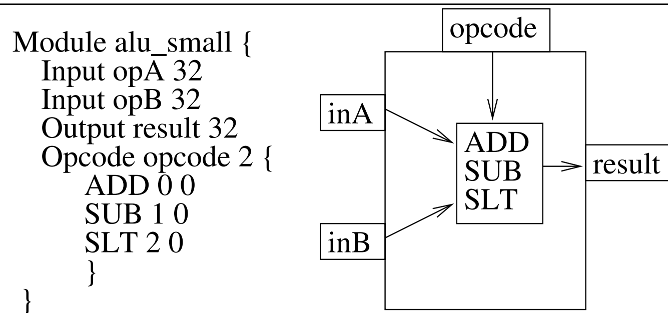


Figure A.2: Sample component description for a simplified ALU. The ALU supports the GENOPs `ADD`, `SUB`, and `SLT`.

erated RTL description, wire names are named after the base component name to avoid lengthy wire names. Within the curly braces, the `File` statement indicates the name of the source file where the RTL can be found (the library entry and RTL implementation must be in the same directory).

The interface is described beginning with the component parameters and their default values using the `Parameters` keyword. In Verilog, modules are paremeterizable so one can design one module and instantiate different variations of it by modifying the parameters passed to each instance. The SPREE infrastructure also supports this parameterization allowing users to modify component parameters as they are selected from the library. The `Parameter` statements identify the names of the parameters and their default values respectively following the `Parameter` keyword.

The input and output ports of the module are defined using the `Input`/`Output` keyword followed by the name of the port and its bit width. The bit width can be an integer or a parameter name. Only ports used to move data are declared here; ports used for control signals, clock, and reset ports are declared separately as outlined below.

The `Opcode` statement is used to define control signals, and functionality. All library entries must contain an `Opcode` statement even if there is no physical opcode port—in this case the opcode port width is set to zero creating a *virtual opcode port*. In the `Opcode` statement, the user defines the names of the opcode, enable, and squash ports if present; the stall port is inferred if the latency of any operation indicates it is variable-latency. In the example in Figure A.2, the ALU has an opcode port named ”opcode” which is 2-bits wide. Within the curly braces of the `Opcode` statement is the list of the GENOPs supported by this component. Only one GENOP from the list can be executed at a time, which allows for resource sharing. In the ALU example, the component can be used for either addition (ADD), subtraction (SUB), and set-on-less-than (SLT) operations.

Each GENOP in the list is accompanied by three fields: (i) the opcode value which indicates the value on the opcode port required to perform the operation; (ii) the latency in cycles where 0 indicates combinational, and negative indicates variable-latency; and (iii) (not shown in the example) the port mapping from the indexed GENOP ports to the component physical ports. The format for this port mapping is {GENOP port}:{component port}, where GENOP port is the index preceded with either i or o for input or output, and the component port is indicated by name. The ALU example should have the following port mapping beside each GENOP after the latency: `i0:opA i1:opB o0:result`.

SPREE supports an arbitrary amount of parallelism by allowing a component to have more than one `Opcode` declaration. For example, a register file is a single component capable of performing two operand reads and one write simultaneously. In the library entry for such a component, there would be three opcode ports, two with an RFREAD GENOP and one with an RFWRITE. Each of these parallel operations would have unique control interfaces specified by their respective `Opcode` statement.

Finally, the last two optional flags in the `Module` section are `clk` and `resetn`. These flags indicate the presence of a clock signal named "clk" and an active-low global reset signal named "resetn" in the component. SPREE will connect the processor clock and global reset to these ports. Components which, for example, are purely combinational do not require a clock or reset signal.

All the components listed in Table 3.2 have been described using library entries as described above. The following is a listing of all the library entries for all components used in this exploration.

```
Module ifetch {
  File ifetch.v
        Input load_data 32
        Input load 1
        Output pc_out 32
        Output next_pc 30
        Output opcode 6
        Output rs 5
        Output rt 5
        Output rd 5
        Output sa 5
        Output offset 16
        Output instr_index 26
        Output func 6
        Output instr 32
        Opcode pcreadop 0 {
                PCREAD 0 0 o0:pc_out o1:next_pc
        }
        Opcode ifetchop 0 {
                IFETCH 0 0 o0:rs o1:rt o2:rd o3:sa o4:offset o5:instr_index o6:opcode o7:func o
        }
        Opcode op 1 en en {
                PCWRITE 0 0 i0:load_data i1:load
                PCWRITEUNCOND 1 0 i0:load_data
        }
        clk
        resetn
  boot_instr
}

Module ifetch_pipe {
  File ifetch_pipe.v
        Input load_data 32
        Input load 1
        Output next_pc 30
        Output sa 5
        Output offset 16
        Output instr_index 26
        Output rd 5
        Output rt 5
        Output rs 5
```

```
        Output func 6
        Output opcode 6
        Output instr 32
        Output pc_out 32
        Opcode pcreadop 0 {
                PCREAD 0 0 o0:pc_out o1:next_pc
        }
        Opcode ifetchop 0 en_instruction_independent en squashn squashn {
                IFETCH 0 0 o0:rs o1:rt o2:rd o3:sa o4:offset o5:instr_index o6:opcode o7:func c
        }
        Opcode op 1 en we {
                PCWRITE 0 0 i0:load_data i1:load
                PCWRITEUNCOND 1 0 i0:load_data
        }
        clk
        resetn
  boot_instr
}

Module reg_file {
  File reg_file.v
        Input a_reg 5
        Output a_readdataout 32
        Input b_reg 5
        Output b_readdataout 32
        Input c_reg 5
        Input c_writedatain 32
        Opcode a_op 0 {
                RFREAD 0 1 i0:a_reg o0:a_readdataout
        }
        Opcode b_op 0 {
                RFREAD 0 1 i0:b_reg o0:b_readdataout
        }
        Opcode c_op 0 en c_we {
                RFWRITE 0 0 i0:c_writedatain i1:c_reg
        }
        clk
        resetn
}

Module reg_file_pipe {
  File reg_file_pipe.v
        Input a_reg 5
        Output a_readdataout 32
        Input b_reg 5
        Output b_readdataout 32
        Input c_reg 5
        Input c_writedatain 32
        Opcode a_op 0 en_instruction_independent a_en {
                RFREAD 0 1 i0:a_reg o0:a_readdataout
        }
        Opcode b_op 0 en_instruction_independent b_en {
                RFREAD 0 1 i0:b_reg o0:b_readdataout
        }
        Opcode c_op 0 en c_we {
                RFWRITE 0 0 i0:c_writedatain i1:c_reg
        }
        clk
        resetn
```

```
}

Module pcadder {
  File pcadder.v
        Input pc 32
        Input offset 32
        Output result 32
        Opcode op 0 {
                ADDU 0 0 i0:pc i1:offset o0:result
        }
}

Module pcadder_merge26lo {
  File pcadder_merge26lo.v
        Input pc 32
        Input offset 32
        Input instr_index 26
        Output result 32
        Opcode op 1 {
                MERGE26LO 0 0 i0:pc i1:instr_index o0:result
                ADDU      1 0 i0:pc i1:offset o0:result
        }
}

Module addersub {
  File addersub.v
        Parameter WIDTH 32
        Input opA 32
        Input opB 32
        Output result 32
        Opcode op 3 {
                ADD  3 0 i0:opA i1:opB o0:result
                ADDU 1 0 i0:opA i1:opB o0:result
                SUB  0 0 i0:opA i1:opB o0:result
                SUBU 2 0 i0:opA i1:opB o0:result
                SLT  6 0 i0:opA i1:opB o0:result
                SLTU 4 0 i0:opA i1:opB o0:result
        }
}

Module addersub_1 {
  File addersub_1.v
        Parameter WIDTH 32
        Input opA 32
        Input opB 32
        Output result 32
        Opcode op 3 {
                ADD  3 1 i0:opA i1:opB o0:result
                ADDU 1 1 i0:opA i1:opB o0:result
                SUB  0 1 i0:opA i1:opB o0:result
                SUBU 2 1 i0:opA i1:opB o0:result
                SLT  6 1 i0:opA i1:opB o0:result
                SLTU 4 1 i0:opA i1:opB o0:result
        }
        clk
        resetn
}

Module addersub_slt {
```

```
  File addersub_slt.v
        Parameter WIDTH 32
        Input opA 32
        Input opB 32
        Output result 32
        Output result_slt 1
        Opcode op 3 {
                ADD  3 0 i0:opA i1:opB o0:result
                ADDU 1 0 i0:opA i1:opB o0:result
                SUB  0 0 i0:opA i1:opB o0:result
                SUBU 2 0 i0:opA i1:opB o0:result
                SLT  6 0 i0:opA i1:opB o0:result_slt
                SLTU 4 0 i0:opA i1:opB o0:result_slt
        }
}

Module addersub_slt_1 {
  File addersub_slt_1.v
        Parameter WIDTH 32
        Input opA 32
        Input opB 32
        Output result 32
        Output result_slt 1
        Opcode op 3 {
                ADD  3 1 i0:opA i1:opB o0:result
                ADDU 1 1 i0:opA i1:opB o0:result
                SUB  0 1 i0:opA i1:opB o0:result
                SUBU 2 1 i0:opA i1:opB o0:result
                SLT  6 1 i0:opA i1:opB o0:result_slt
                SLTU 4 1 i0:opA i1:opB o0:result_slt
        }
        clk
        resetn
}

Module serialalu {
  File serialalu.v
        Parameter WIDTH 32
        Input opA 32
        Input opB 32
        Output result 32
        Output shift_count 6
        Opcode op 4 en start {
                ADD  6 -1 i0:opA i1:opB o0:result
                ADDU 4 -1 i0:opA i1:opB o0:result
                SUB  14 -1 i0:opA i1:opB o0:result
                SUBU 12 -1 i0:opA i1:opB o0:result
                SLT  15 -1 i0:opA i1:opB o0:result
                SLTU 13 -1 i0:opA i1:opB o0:result
                AND  0 -1 i0:opA i1:opB o0:result
                OR   1 -1 i0:opA i1:opB o0:result
                XOR  2 -1 i0:opA i1:opB o0:result
                NOR  3 -1 i0:opA i1:opB o0:result
                SHIFTLEFT        9 -1 i0:opB i1:opA o0:result
                SHIFTRIGHTLOGIC  8 -1 i0:opB i1:opA o0:result
                SHIFTRIGHTARITH  10 -1 i0:opB i1:opA o0:result
        }
        clk
        resetn
```

```
}

Module shifter_barrel {
  File shifter_barrel.v
        Parameter WIDTH 32
        Input opB 32
        Input sa 5
        Output result 32
        Opcode op 2 {
                SHIFTLEFT         0 0 i0:opB i1:sa o0:result
                SHIFTRIGHTLOGIC   1 0 i0:opB i1:sa o0:result
                SHIFTRIGHTARITH   3 0 i0:opB i1:sa o0:result
        }
}

Module shifter_barrel1 {
  File shifter_barrel1.v
        Parameter WIDTH 32
        Input opB 32
        Input sa 5
        Output result 32
        Opcode op 2 {
                SHIFTLEFT         0 1 i0:opB i1:sa o0:result
                SHIFTRIGHTLOGIC   1 1 i0:opB i1:sa o0:result
                SHIFTRIGHTARITH   3 1 i0:opB i1:sa o0:result
        }
  clk
  resetn
}

Module shifter_barrel_pipe1 {
  File shifter_barrel_pipe1.v
        Parameter WIDTH 32
        Input opB 32
        Input sa 5
        Output result 32
        Opcode op 2 {
                SHIFTLEFT         0 1 i0:opB i1:sa o0:result
                SHIFTRIGHTLOGIC   1 1 i0:opB i1:sa o0:result
                SHIFTRIGHTARITH   3 1 i0:opB i1:sa o0:result
        }
  clk
  resetn
}

Module shifter_perbit_pipe {
  File shifter_perbit_pipe.v
        Parameter WIDTH 32
        Input opB 32
        Input sa 5
        Input dst 5
        Output result 32
        Opcode op 2 en start {
                SHIFTLEFT         0 −2 i0:opB i1:sa o0:result
                SHIFTRIGHTLOGIC   1 −2 i0:opB i1:sa o0:result
                SHIFTRIGHTARITH   3 −2 i0:opB i1:sa o0:result
        }
        clk
        resetn
```

```
}

Module shifter_perbit {
  File shifter_perbit.v
        Parameter WIDTH 32
        Input opB 32
        Input sa 5
        Output result 32
        Opcode op 2 en start {
               SHIFTLEFT          0 −1 i0:opB i1:sa o0:result
               SHIFTRIGHTLOGIC    1 −1 i0:opB i1:sa o0:result
               SHIFTRIGHTARITH    3 −1 i0:opB i1:sa o0:result
        }
        clk
        resetn
}

Module logic_unit {
  File logic_unit.v
        Parameter WIDTH 32
        Input opA 32
        Input opB 32
        Output result 32
        Opcode op 2 {
               AND  0 0 i0:opA i1:opB o0:result
               OR   1 0 i0:opA i1:opB o0:result
               XOR  2 0 i0:opA i1:opB o0:result
               NOR  3 0 i0:opA i1:opB o0:result
        }
}

Module mul_3 {
  File mul_3.v
        Parameter WIDTH 32
        Input opA 32
        Input opB 32
        Output hi 32
        Output lo 32
        Opcode op 1 {
               MULT  1 3 i0:opA i1:opB o0:lo o1:hi
               MULTU 0 3 i0:opA i1:opB o0:lo o1:hi
        }
  clk
  resetn
}

Module mul_1 {
  File mul_1.v
        Parameter WIDTH 32
        Input opA 32
        Input opB 32
        Output hi 32
        Output lo 32
        Opcode op 1 {
               MULT  1 1 i0:opA i1:opB o0:lo o1:hi
               MULTU 0 1 i0:opA i1:opB o0:lo o1:hi
        }
  clk
  resetn
```

```
}

Module mul {
  File mul.v
        Parameter WIDTH 32
        Input opA 32
        Input opB 32
        Output hi 32
        Output lo 32
        Opcode op 1 {
                MULT  1 0 i0:opA i1:opB o0:lo o1:hi
                MULTU 0 0 i0:opA i1:opB o0:lo o1:hi
        }
}

Module mul_shift {
  File mul_shift.v
        Parameter WIDTH 32
        Input opA 32
        Input opB 32
        Input sa 5
        Output hi 32
        Output lo 32
        Output shift_result 32
        Opcode op 3 {
                MULT             6 0 i0:opA i1:opB o0:lo o1:hi
                MULTU            4 0 i0:opA i1:opB o0:lo o1:hi
                SHIFTLEFT        0 0 i0:opA i1:sa o0:shift_result
                SHIFTRIGHTLOGIC  1 0 i0:opA i1:sa o0:shift_result
                SHIFTRIGHTARITH  3 0 i0:opA i1:sa o0:shift_result
        }
}

Module mul_shift_stall {
  File mul_shift_stall.v
        Parameter WIDTH 32
        Input opA 32
        Input opB 32
        Input dst 5
        Input sa 5
        Output hi 32
        Output lo 32
        Output shift_result 32
        Opcode op 3 en start {
                MULT             6 -2 i0:opA i1:opB o0:lo o1:hi
                MULTU            4 -2 i0:opA i1:opB o0:lo o1:hi
                SHIFTLEFT        0 -2 i0:opA i1:sa o0:shift_result
                SHIFTRIGHTLOGIC  1 -2 i0:opA i1:sa o0:shift_result
                SHIFTRIGHTARITH  3 -2 i0:opA i1:sa o0:shift_result
        }
        clk
        resetn
}

Module mul_shift1 {
  File mul_shift1.v
        Parameter WIDTH 32
        Input opA 32
        Input opB 32
```

```
        Input sa 5
        Output hi 32
        Output lo 32
        Output shift_result 32
        Opcode op 3 {
                MULT              6 1 i0:opA i1:opB o0:lo o1:hi
                MULTU             4 1 i0:opA i1:opB o0:lo o1:hi
                SHIFTLEFT         0 1 i0:opA i1:sa o0:shift_result
                SHIFTRIGHTLOGIC   1 1 i0:opA i1:sa o0:shift_result
                SHIFTRIGHTARITH   3 1 i0:opA i1:sa o0:shift_result
        }
  clk
  resetn
}

Module mul_shift_pipe1 {
  File mul_shift_pipe1.v
        Parameter WIDTH 32
        Input opA 32
        Input opB 32
        Input sa 5
        Output hi 32
        Output lo 32
        Output shift_result 32
        Opcode op 3 {
                MULT              6 1 i0:opA i1:opB o0:lo o1:hi
                MULTU             4 1 i0:opA i1:opB o0:lo o1:hi
                SHIFTLEFT         0 1 i0:opA i1:sa o0:shift_result
                SHIFTRIGHTLOGIC   1 1 i0:opA i1:sa o0:shift_result
                SHIFTRIGHTARITH   3 1 i0:opA i1:sa o0:shift_result
        }
  clk
  resetn
}

Module signext16 {
  File signext16.v
        Input in 16
        Output out 32
        Opcode op 0 {
                SIGNEXT16 0 0 i0:in o0:out
        }
}
Module merge26lo {
  File merge26lo.v
        Input in1 32
        Input in2 26
        Output out 32
        Opcode op 0 {
                MERGE26LO 0 0 i0:in1 i1:in2 o0:out
        }
}
Module branchresolve {
  File branchresolve.v
        Parameter WIDTH 32
        Input rs 32
        Input rt 32
        Output eq 1
        Output ne 1
```

```
        Output ltz 1
        Output lez 1
        Output gtz 1
        Output gez 1
        Output eqz 1
        Opcode op 0 en en {
                BRANCHRESOLVE 0 0 i0:rs i1:rt o0:eq o1:ne o2:ltz o3:lez o4:gtz o5:gez o6:eqz
        }
}

Module hi_reg_1 {
  File hi_reg.v
        Parameter WIDTH 32
        Input d 32
        Output q 32
        Opcode op 0 {
                HIREAD 0 1 o0:q
        }
        Opcode op2 0 en en {
                HIWRITE 0 0 i0:d
        }
    clk
    resetn
}

Module lo_reg_1 {
  File lo_reg.v
        Parameter WIDTH 32
        Input d 32
        Output q 32
        Opcode op 0 {
                LOREAD 0 1 o0:q
        }
        Opcode op2 0 en en {
                LOWRITE 0 0 i0:d
        }
    clk
    resetn
}

Module hi_reg {
  File hi_reg.v
        Parameter WIDTH 32
        Input d 32
        Output q 32
        Opcode op 0 {
                HIREAD 0 0 o0:q
        }
        Opcode op2 0 en en {
                HIWRITE 0 0 i0:d
        }
    clk
    resetn
}

Module lo_reg {
  File lo_reg.v
        Parameter WIDTH 32
        Input d 32
```

```
        Output q 32
        Opcode op 0 {
                LOREAD 0 0 o0:q
        }
        Opcode op2 0 en en {
                LOWRITE 0 0 i0:d
        }
    clk
    resetn
}

Module register {
  File components.v
        Parameter WIDTH 32
        Input d WIDTH
        Output q WIDTH
        Opcode op 0 en en {
                NOP 0 1 i0:d o0:q
        }
        clk
        resetn
}

Module pipereg {
  File components.v
        Parameter WIDTH 32
        Input d WIDTH
        Output q WIDTH
        Opcode op 0 en en squashn squashn {
                NOP 0 1 i0:d o0:q
        }
        clk
        resetn
}

Module pipedelayreg {
  File components.v
        Parameter WIDTH 32
        Input d WIDTH
        Input dst 5
        Output q WIDTH
        Opcode op 0 en en squashn squashn {
                NOP 0 -2 i0:d o0:q
        }
        clk
        resetn
}

Module delay {
  File delay.v
        Parameter WIDTH 32
        Input d WIDTH
        Output q WIDTH
        Opcode op 0 {
                NOP 0 1 i0:d o0:q
        }
    clk
    resetn
}
```

```
Module fakedelay {
  File components.v
        Parameter WIDTH 32
        Input d WIDTH
        Output q WIDTH
        Opcode op 0 {
                NOP 0 1 i0:d o0:q
        }
  clk
}

Module zeroer {
  File components.v
        Parameter WIDTH 32
        Input d WIDTH
        Output q WIDTH
        Opcode en 1 {
                NOP 1 0 i0:d o0:q
        }
}

Module nop {
  File components.v
        Parameter WIDTH 32
        Input d WIDTH
        Output q WIDTH
        Opcode op 0 {
                NOP 0 0 i0:d o0:q
        }
}

Module const {
  File components.v
        Parameter VAL 16
        Parameter WIDTH 32
        Output out 32
        Opcode op 0 {
                CONST 0 0 o0:out
        }
}

Module branch_detector {
  File components.v
        Input opcode 6
        Input func 6
        Output is_branch 1
        Opcode op 0 {
                NOP 0 0 i0:opcode o0:is_branch
        }
}

Module data_mem {
  File data_mem.v
        Input d_writedata 32
        Input d_address 32
        Output d_loadresult 32
        Opcode op 4 en en {
                LOADBYTE 7 1 i0:d_address o0:d_loadresult
```

```
                LOADHALF 5 1 i0:d_address o0:d_loadresult
                LOADWORD 0 1 i0:d_address o0:d_loadresult
                LOADBYTEU 3 1 i0:d_address o0:d_loadresult
                LOADHALFU 1 1 i0:d_address o0:d_loadresult
                STOREBYTE 11 1 i1:d_address i0:d_writedata
                STOREHALF 9 1 i1:d_address i0:d_writedata
                STOREWORD 8 1 i1:d_address i0:d_writedata
        }
        clk
        resetn
  boot_data
}

Module data_mem_dp {
  File data_mem_dp.v
        Input d_writedata 32
        Input d_address 32
        Output d_loadresult 32
        Opcode op 4 en en {
                LOADBYTE 7 1 i0:d_address o0:d_loadresult
                LOADHALF 5 1 i0:d_address o0:d_loadresult
                LOADWORD 0 1 i0:d_address o0:d_loadresult
                LOADBYTEU 3 1 i0:d_address o0:d_loadresult
                LOADHALFU 1 1 i0:d_address o0:d_loadresult
                STOREBYTE 11 1 i1:d_address i0:d_writedata
                STOREHALF 9 1 i1:d_address i0:d_writedata
                STOREWORD 8 1 i1:d_address i0:d_writedata
        }
        clk
        resetn
}

Module shifter_serial_datamem {
  File shifter_serial_datamem.v
        Input d_writedata 32
        Input d_address 32
        Input sa 5
        Output d_loadresult 32
        Output shift_result 32
        Opcode op 4 en start {
                LOADBYTE 7 -1 i0:d_address o0:d_loadresult
                LOADHALF 5 -1 i0:d_address o0:d_loadresult
                LOADWORD 0 -1 i0:d_address o0:d_loadresult
                LOADBYTEU 3 -1 i0:d_address o0:d_loadresult
                LOADHALFU 1 -1 i0:d_address o0:d_loadresult
                STOREBYTE 11 -1 i1:d_address i0:d_writedata
                STOREHALF 9 -1 i1:d_address i0:d_writedata
                STOREWORD 8 -1 i1:d_address i0:d_writedata
                SHIFTLEFT         12 -1 i0:d_writedata i1:sa o0:shift_result
                SHIFTRIGHTLOGIC   14 -1 i0:d_writedata i1:sa o0:shift_result
                SHIFTRIGHTARITH   15 -1 i0:d_writedata i1:sa o0:shift_result
        }
        clk
        resetn
}
```

Appendix B

Exploration Result Details

B.1 CAD Settings

Listing B.1: Base Settings common for all compiles

```
set_global_assignment -name FAMILY Stratix
set_global_assignment -name DEVICE EP1S40F780C5
set_global_assignment -name FITTER_EFFORT "STANDARD_FIT"

set_global_assignment -name AUTO_ROM_RECOGNITION OFF
set_global_assignment -name AUTO_RAM_RECOGNITION OFF
```

Listing B.2: Area-focussed Optimization Settings

```
# Analysis & Synthesis Assignments
# ================================
set_global_assignment -name STRATIX_OPTIMIZATION_TECHNIQUE AREA
set_global_assignment -name MUX_RESTRUCTURE ON
set_global_assignment -name STATE_MACHINE_PROCESSING AUTO
set_global_assignment -name ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP OFF
set_global_assignment -name ADV_NETLIST_OPT_SYNTH_GATE_RETIME OFF

# Fitter Assignments
# ==================
set_global_assignment -name AUTO_PACKED_REGISTERS_STRATIX "MINIMIZE_AREA"
set_global_assignment -name PHYSICAL_SYNTHESIS_COMBO_LOGIC OFF
set_global_assignment -name PHYSICAL_SYNTHESIS_REGISTER_DUPLICATION OFF
set_global_assignment -name PHYSICAL_SYNTHESIS_REGISTER_RETIMING OFF
```

Listing B.3: Speed-focussed Optimization Settings

```
# Analysis & Synthesis Assignments
# ================================
```

```
set_global_assignment -name STRATIX_OPTIMIZATION_TECHNIQUE SPEED
set_global_assignment -name MUX_RESTRUCTURE OFF
set_global_assignment -name STATE_MACHINE_PROCESSING AUTO
set_global_assignment -name ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP ON
set_global_assignment -name ADV_NETLIST_OPT_SYNTH_GATE_RETIME ON

# Fitter Assignments
# ==================
set_global_assignment -name AUTO_PACKED_REGISTERS_STRATIX OFF
set_global_assignment -name PHYSICAL_SYNTHESIS_COMBO_LOGIC ON
set_global_assignment -name PHYSICAL_SYNTHESIS_REGISTER_DUPLICATION ON
set_global_assignment -name PHYSICAL_SYNTHESIS_REGISTER_RETIMING ON
```

B.2 Data from exploration

B.2.1 Shifter Implementation, Multiply Support, and Pipeline Depth

Table B.1: Data for hardware multiply support over different shifters and pipelines.

Pipe depth		2-stage			3-stage			4-stage		
Shifter		Serial	Multiplier	LUT-based	Serial	Multiplier	LUT-based	Serial	Multiplier	LUT-based
CAD flow default	Area (LEs)	889	928	1130	1005	1083	1277	1075	1149	1349
	Speed (MHz)	67.46	67.28	64.39	79.08	76.35	72.50	84.26	79.92	74.74
	Energy (nJ/cycle)	1.71	2.02	1.94	1.59	2.04	2.05	1.55	1.91	2.12
	Energy (nJ/instr)	7.41	5.14	4.93	5.11	3.30	3.04	5.12	3.26	3.31
	Wall clock time (us)	4916	2888	3017	3106	1622	1557	2987	1627	1593
CAD flow area	Area (LEs)	884	920	1129	998	1070	1275	1071	1141	1346
	Speed (MHz)	68.51	67.25	65.85	76.15	74.88	73.42	83.35	80.10	74.66
CAD flow speed	Area (LEs)	1137	1298	1602	1222	1418	1831	1341	1532	1910
	Speed (MHz)	68.38	70.34	67.10	76.61	79.68	71.47	76.26	79.65	73.59
CPI	bubble_sort	2.82	2.67	2.67	1.64	1.61	1.60	1.65	1.62	1.62
	crc	8.71	2.50	2.50	7.86	2.00	1.64	7.86	2.00	1.64
	des	4.49	2.37	2.37	3.38	1.47	1.27	3.39	1.48	1.28
	fft	3.91	2.68	2.68	2.67	1.73	1.48	2.70	1.76	1.52
	fir	2.44	2.33	2.33	1.48	1.44	1.37	1.61	1.58	1.51
	quant	5.06	2.56	2.56	3.96	1.75	1.46	4.03	1.82	1.53
	iquant	2.85	2.51	2.51	1.82	1.66	1.51	1.86	1.69	1.55
	turbo	7.15	2.36	2.36	6.25	1.76	1.47	6.27	1.78	1.49
	vlc	6.07	2.56	2.56	4.89	1.64	1.40	4.95	1.70	1.46
	bitcnts	3.74	2.23	2.23	2.75	1.37	1.24	2.88	1.51	1.37
	CRC32	3.00	2.80	2.80	2.00	2.00	2.00	2.20	2.20	2.20
	qsort	2.60	2.48	2.48	1.47	1.45	1.42	1.60	1.57	1.55
	sha	4.55	2.44	2.44	3.46	1.50	1.39	3.51	1.55	1.43
	stringsearch	2.40	2.25	2.25	1.43	1.31	1.29	1.60	1.48	1.47
	FFT_MI	4.85	2.43	2.43	3.68	1.46	1.30	3.74	1.53	1.37
	dijkstra	3.66	2.85	2.85	2.32	1.78	1.66	2.39	1.85	1.73
	patricia	3.65	2.77	2.77	2.35	1.69	1.62	2.47	1.81	1.74
	gol	6.16	2.74	2.74	4.81	1.67	1.53	4.91	1.77	1.63
	dct	5.10	2.58	2.58	3.80	1.54	1.36	3.88	1.61	1.44
	dhry	3.59	2.74	2.74	2.27	1.59	1.53	2.38	1.70	1.64
	AVERAGE	4.34	2.54	2.54	3.21	1.62	1.48	3.29	1.70	1.56

Table B.2: Data for hardware multiply support over different shifters and pipelines (cont'd).

Pipe depth		5-stage			7-stage		
Shifter		Serial	Multiplier	LUT-based	Serial	Multiplier	LUT-based
CAD flow default	Area (LEs)	1082	1190	1340	1283	1338	1493
	Speed (MHz)	92.63	91.01	91.54	103.76	98.19	105.80
	Energy (nJ/cycle)	1.34	1.64	1.63	1.10	1.32	1.14
	Energy (nJ/instr)	4.71	3.15	2.93	4.81	3.51	3.04
	Wall clock time (us)	2903	1614	1497	3228	2076	1927
CAD flow area	Area (LEs)	1081	1187	1336	1282	1320	1489
	Speed (MHz)	93.12	91.36	89.57	102.17	94.92	105.86
CAD flow speed	Area (LEs)	1312	1463	1845	1509	1563	1773
	Speed (MHz)	96.04	93.32	91.84	102.07	103.25	107.31
CPI	bubble_sort	1.85	1.81	1.81	2.73	2.69	2.69
	crc	8.28	2.50	2.14	9.50	3.43	3.43
	des	3.39	1.48	1.27	3.81	1.71	1.71
	fft	2.97	1.99	1.81	3.81	2.66	2.66
	fir	1.73	1.63	1.62	2.41	2.30	2.30
	quant	4.38	2.10	1.89	5.39	2.90	2.90
	iquant	2.18	1.95	1.87	3.13	2.82	2.82
	turbo	6.70	2.22	1.93	7.67	2.92	2.92
	vlc	5.07	1.88	1.64	5.88	2.49	2.49
	bitcnts	3.00	1.63	1.50	3.69	2.20	2.20
	CRC32	2.60	2.60	2.60	4.00	4.00	4.00
	qsort	1.70	1.67	1.65	2.39	2.34	2.34
	sha	3.67	1.73	1.61	4.41	2.36	2.36
	stringsearch	1.99	1.87	1.85	3.02	2.88	2.88
	FFT_MI	3.89	1.68	1.52	4.53	2.17	2.17
	dijkstra	2.70	2.16	2.05	3.79	3.14	3.14
	patricia	2.66	2.01	1.94	3.58	2.87	2.87
	gol	5.20	2.06	1.92	6.12	2.84	2.84
	dct	4.03	1.76	1.59	4.73	2.30	2.30
	dhry	2.39	1.71	1.65	3.07	2.34	2.34
	AVERAGE	3.52	1.92	1.79	4.38	2.67	2.67

Table B.3: Data for software multiply support over different shifters and pipelines.

Pipe depth		2-stage			3-stage			4-stage		
Shifter		Serial	Multiplier	LUT-based	Serial	Multiplier	LUT-based	Serial	Multiplier	LUT-based
CAD flow default	Area (LEs)	666	862	893	772	978	1046	835	1051	1116
	Speed (MHz)	68.89	67.02	64.01	78.31	75.88	71.24	90.52	78.34	72.08
CPI	bubble_sort	2.82	2.67	2.67	1.64	1.61	1.60	1.65	1.62	1.62
	crc	8.71	2.50	2.50	7.86	2.00	1.64	7.86	2.00	1.64
	des	4.49	2.37	2.37	3.38	1.47	1.27	3.39	1.48	1.28
	fft	23.23	17.75	17.75	14.79	11.60	9.35	15.96	12.77	10.52
	fir	3.63	3.29	3.29	2.17	1.98	1.89	2.40	2.22	2.13
	quant	16.84	11.56	11.56	11.69	8.02	6.46	12.49	8.82	7.27
	iquant	23.67	18.69	18.69	14.70	12.17	9.79	15.90	13.37	10.99
	turbo	7.15	2.36	2.36	6.25	1.76	1.47	6.27	1.78	1.49
	vlc	6.07	2.56	2.56	4.89	1.64	1.40	4.95	1.70	1.46
	bitcnts	3.74	2.23	2.23	2.75	1.37	1.24	2.88	1.51	1.37
	CRC32	3.00	2.80	2.80	2.00	2.00	2.00	2.20	2.20	2.20
	qsort	2.73	2.57	2.57	1.55	1.50	1.46	1.70	1.65	1.61
	sha	4.55	2.44	2.44	3.46	1.50	1.39	3.51	1.55	1.43
	stringsearch	2.40	2.25	2.25	1.43	1.31	1.29	1.60	1.48	1.47
	FFT_MI	10.07	6.38	6.38	7.09	4.11	3.44	7.44	4.46	3.79
	dijkstra	3.71	2.89	2.89	2.34	1.80	1.68	2.42	1.88	1.76
	patricia	3.65	2.77	2.77	2.35	1.69	1.62	2.47	1.81	1.74
	gol	6.16	2.74	2.74	4.81	1.67	1.53	4.91	1.77	1.63
	dct	11.52	7.44	7.44	8.01	4.80	4.00	8.44	5.22	4.43
	dhry	3.68	2.82	2.82	2.33	1.64	1.57	2.45	1.75	1.69
	AVERAGE	7.59	5.05	5.05	5.27	3.28	2.80	5.54	3.55	3.08

Table B.4: Data for software multiply support over different shifters and pipelines (cont'd).

Pipe depth		5-stage			7-stage		
Shifter		Serial	Multiplier	LUT-based	Serial	Multiplier	LUT-based
CAD flow default	Area (LEs)	846	1098	1111	1055	1243	1261
	Speed (MHz)	91.92	91.75	92.45	104.21	101.21	106.42
CPI	bubble_sort	1.85	1.81	1.81	2.73	2.69	2.69
	crc	8.28	2.50	2.14	9.50	3.43	3.43
	des	3.39	1.48	1.27	3.82	1.71	1.71
	fft	18.10	14.92	12.67	24.27	18.85	18.85
	fir	2.53	2.34	2.25	3.41	3.14	3.14
	quant	14.01	10.39	8.83	18.24	13.11	13.11
	iquant	18.42	15.89	13.51	25.04	20.13	20.13
	turbo	6.70	2.22	1.93	7.67	2.92	2.92
	vlc	5.09	1.88	1.64	5.88	2.49	2.49
	bitcnts	3.02	1.63	1.50	3.69	2.20	2.20
	CRC32	2.60	2.60	2.60	4.00	4.00	4.00
	qsort	1.82	1.77	1.73	2.58	2.49	2.49
	sha	3.67	1.73	1.61	4.41	2.36	2.36
	stringsearch	1.99	1.87	1.85	3.02	2.88	2.88
	FFT_MI	8.05	5.08	4.41	10.04	6.41	6.41
	dijkstra	2.73	2.19	2.07	3.83	3.17	3.17
	patricia	2.66	2.01	1.94	3.58	2.87	2.87
	gol	5.21	2.06	1.92	6.12	2.84	2.84
	dct	9.15	5.93	5.14	11.50	7.50	7.50
	dhry	2.47	1.78	1.71	3.17	2.41	2.41
	AVERAGE	6.09	4.10	3.63	7.82	5.38	5.38

B.2.2 Forwarding

Table B.5: Measurements of pipelines with forwarding.

		pipe3_mulshift				pipe4_mulshift				pipe5_barrelshift				pipe7_barrelshift			
Forwarding		none	rs	rt	rs&rt	none	rs	rt	rs&rt	none	rs	rt	rs&rt	none	rs	rt	rs&rt
Area	(LEs)	1083	1149	1149	1175	1135	1200	1200	1233	1340	1427	1427	1466	1493	1555	1522	1558
Speed	(MHz)	76.35	76.70	77.49	77.99	74.14	73.40	75.48	73.93	91.54	87.51	88.02	88.81	105.80	104.94	102.95	106.63
Energy	(nJ/cycle)	2.04	2.13	1.94	2.01	1.94	2.06	1.87	2.04	1.63	1.72	1.64	1.77	1.14	1.37	1.38	1.41
	(nJ/instr)	3.30	3.05	2.99	2.73	3.61	3.30	3.26	3.01	2.93	2.89	2.98	2.76	3.04	3.29	3.50	3.19
CPI	bubble_sort	1.61	1.33	1.61	1.33	2.01	1.61	2.01	1.60	1.81	1.62	2.02	1.62	2.69	2.24	2.69	2.24
	crc	2.00	1.71	1.79	1.50	2.28	1.93	2.00	1.64	2.14	1.93	2.00	1.64	3.43	3.07	3.14	2.79
	des	1.47	1.42	1.44	1.39	1.44	1.30	1.41	1.27	1.27	1.31	1.42	1.28	1.71	1.53	1.68	1.49
	fft	1.73	1.63	1.56	1.46	1.88	1.73	1.66	1.48	1.81	1.76	1.69	1.52	2.66	2.50	2.42	2.24
	fir	1.44	1.33	1.38	1.27	1.65	1.50	1.52	1.37	1.62	1.64	1.62	1.51	2.30	2.16	2.16	2.02
	quant	1.75	1.67	1.55	1.46	1.90	1.75	1.61	1.46	1.89	1.82	1.68	1.53	2.90	2.75	2.58	2.43
	iquant	1.66	1.41	1.59	1.35	1.94	1.65	1.83	1.51	1.87	1.69	1.87	1.55	2.82	2.51	2.72	2.37
	turbo	1.76	1.52	1.58	1.33	1.94	1.68	1.74	1.47	1.93	1.70	1.76	1.49	2.92	2.65	2.71	2.44
	vlc	1.64	1.53	1.54	1.43	1.77	1.64	1.55	1.40	1.64	1.68	1.61	1.46	2.49	2.32	2.26	2.08
	bitcnts	1.37	1.29	1.34	1.25	1.49	1.30	1.45	1.24	1.50	1.43	1.57	1.37	2.20	1.96	2.15	1.91
	CRC32	2.00	1.40	2.00	1.40	2.60	2.00	2.60	2.00	2.60	2.20	2.80	2.20	4.00	3.40	4.00	3.40
	qsort	1.45	1.28	1.43	1.27	1.71	1.47	1.66	1.42	1.65	1.60	1.77	1.55	2.34	2.10	2.30	2.05
	sha	1.50	1.34	1.46	1.30	1.71	1.49	1.62	1.39	1.61	1.54	1.67	1.43	2.36	2.14	2.25	2.02
	stringsearch	1.31	1.07	1.30	1.06	1.72	1.31	1.71	1.29	1.85	1.48	1.88	1.47	2.88	2.47	2.87	2.45
	FFT_MI	1.46	1.35	1.42	1.31	1.56	1.39	1.48	1.30	1.52	1.45	1.54	1.37	2.17	1.98	2.08	1.87
	dijkstra	1.78	1.52	1.71	1.45	2.16	1.80	2.02	1.66	2.05	1.88	2.09	1.73	3.14	2.77	3.00	2.63
	patricia	1.69	1.49	1.62	1.42	2.03	1.73	1.92	1.62	1.94	1.86	2.03	1.74	2.87	2.57	2.76	2.45
	gol	1.67	1.50	1.56	1.38	1.92	1.68	1.78	1.53	1.92	1.77	1.87	1.63	2.84	2.59	2.69	2.44
	dct	1.54	1.41	1.51	1.38	1.65	1.47	1.54	1.36	1.59	1.55	1.61	1.44	2.30	2.11	2.17	1.98
	dhry	1.59	1.48	1.54	1.42	1.78	1.61	1.70	1.53	1.65	1.71	1.81	1.64	2.34	2.15	2.25	2.04
	AVERAGE	1.62	1.43	1.55	1.36	1.86	1.60	1.74	1.48	1.79	1.68	1.82	1.56	2.67	2.40	2.54	2.27
Wall clock	(us)	1622	1429	1524	1331	1915	1667	1763	1527	1497	1467	1576	1340	1927	1746	1888	1624

B.2.3 Minimizing Area Processors

Table B.6: Measurements of processors which minimize area

		serialalu	serialalu_fastclock	serialshift_memalign_share
Area	(LEs)	836	869.4	883
Speed	(MHz)	88.444	119.129	69.108
CPI	bubble_sort	29.68	31.53	3.57
	crc	29.14	30.57	9.86
	des	31.95	33.47	6.01
	fft	25.56	27.19	4.40
	fir	25.89	27.46	2.77
	quant	22.18	23.69	6.11
	iquant	26.10	27.67	4.41
	turbo	26.24	27.34	7.33
	vlc	27.80	29.39	9.04
	bitcnts	29.05	30.34	5.22
	CRC32	24.60	26.40	6.00
	qsort	28.28	29.90	4.47
	sha	32.51	34.07	6.12
	stringsearch	29.98	31.27	3.24
	FFT_MI	29.55	30.93	5.29
	dijkstra	28.98	30.76	4.48
	patricia	29.60	31.38	4.89
	gol	29.62	31.14	9.47
	dct	29.42	30.95	5.65
	dhry	30.08	31.98	5.88
	AVERAGE	28.31	29.87	5.71
Wall clock time	(us)	24458	19159	6314

B.2.4 Subsetted Processors

Table B.7: ISA subsetting data on processors with full hardware multiply support.

	pipe2_barrelshift		pipe3_mulshift_stall		pipe5_barrelshift	
	Area (LEs)	Speed (MHz)	Area (LEs)	Speed (MHz)	Area (LEs)	Speed (MHz)
ORIGINAL	1151	55.89	1082	76.08	1339	91.47
bubble_sort	455	68.88	746	78.98	743	97.42
crc	923	58.96	903	79.41	1047	101.15
des	950	56.31	926	79.21	1071	97.43
fft	874	58.42	909	74.84	1225	97.58
fir	468	69.36	844	80.50	927	99.03
quant	1003	56.37	1027	76.60	1330	95.41
iquant	977	58.06	1030	75.29	1326	94.43
turbo	1052	57.97	987	76.17	1147	92.35
vlc	1007	57.90	990	75.84	1137	92.29
bitcnts	801	60.36	924	75.45	1108	94.39
CRC32	488	66.67	805	79.49	781	98.35
qsort	1012	58.27	1029	73.83	1324	94.20
sha	961	58.78	940	77.87	1069	94.64
stringsearch	991	57.34	944	77.07	1114	94.70
FFT_MI	958	57.99	958	74.67	1224	94.59
dijkstra	1041	57.74	1071	77.26	1349	91.80
patricia	946	57.55	951	76.78	1118	94.00
gol	902	57.47	926	75.18	1127	95.97
dct	1016	58.40	1019	75.87	1297	92.74
dhry	1083	58.08	1048	76.35	1346	93.51
AVERAGE	895	59.54	949	76.83	1140	95.30

B.2.5 Nios II

Table B.8: Area and performance of Nios II.

		Nios II/e	Nios II/s	Nios II/f
Area	(LEs)	586	1279.9	1670
Speed	(MHz)	159.26	119.73	134.87
CPI	bubble_sort	7.60	2.37	2.19
	crc	12.78	2.21	1.80
	des	8.90	2.39	2.33
	fft	59.52	1.92	1.58
	fir	10.78	1.59	2.13
	quant	54.38	2.54	2.21
	iquant	62.81	1.73	1.93
	turbo	11.40	2.64	2.17
	vlc	10.85	2.49	1.67
	bitcnts	8.18	2.14	1.59
	CRC32	4.60	1.15	1.31
	qsort	8.98	3.11	2.42
	sha	7.33	1.69	1.21
	stringsearch	6.52	1.51	1.68
	FFT	25.11	3.64	2.61
	dijkstra	8.24	1.99	1.73
	patricia	7.37	3.11	2.30
	gol	16.75	2.72	1.94
	dct	25.74	3.35	2.40
	dhry	9.68	2.93	2.26
	AVERAGE	18.38	2.36	1.97
Wall clock time	(us)	8816	1507	1118

Bibliography

[1] "Altera Excalibur Devices," http://www.altera.com/products/devices/arm/arm-index.html, Altera.

[2] "Nios," http://www.altera.com/products/ip/processors/nios/nio-index.html, Altera.

[3] "Nios II Processor Implementation in Cyclone II FPGAs," http://www.altera.com/products/devices/cyclone2/features/nios2/cy2-cyclone2_nios2.html, Altera.

[4] "Quartus II," http://www.altera.com/products/software/products/quartus2/qts-index.html, Altera.

[5] "Stratix Device Handbook," http://www.altera.com/literature/lit-stx.jsp, Altera.

[6] "Stratix II - Design Building Block Performance," http://www.altera.com/products/devices/stratix2/features/architecture/st2-dbb_perf.html, Altera.

[7] "Stratix II Device Handbook," http://www.altera.com/literature/lit-stx2.jsp, Altera.

[8] "Bluespec," http://www.bluespec.com, Bluespec.

[9] G. Braun, A. Nohl, W. Sheng, J. Ceng, M. Hohenauer, H. Scharwachter, R. Leupers, and H. Meyr, "A novel approach for flexible and consistent ADL-driven ASIP de-

sign," in *DAC '04: Proceedings of the 41st annual conference on Design automation.* New York, NY, USA: ACM Press, 2004, pp. 717–722.

[10] R. Cliff, "Altera Corporation," Private Communication, 2005.

[11] N. Dave and M. Pellauer, "UNUM: A General Microprocessor Framework Using Guarded Atomic Actions," in *Workshop on Architecture Research using FPGA Platforms in the 11th International Symposium on High-Performance Computer Architecture.* IEEE Computer Society, 2005.

[12] B. Fagin and J. Erickson, "DartMIPS: A Case Study in Quantitative Analysis of Processor Design Tradeoffs Using FPGAs," in *Proceedings of the 1993 International Workshop on Field Programmable Logic and Applications*, Oxford, England, September 1993.

[13] J. A. Fisher, "Customized instruction-sets for embedded processors," in *Proceedings of the 36th ACM/IEEE conference on Design automation conference.* ACM Press, 1999, pp. 253–257.

[14] "Dhrystone 2.1," http://www.freescale.com, Freescale.

[15] "LEON SPARC," http://www.gaisler.com, Gaisler Research.

[16] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in *Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems.* ACM Press, 2003, pp. 137–147.

[17] M. Gries, "Methods for Evaluating and Covering the Design Space during Early Design Development," Electronics Research Lab, University of California at Berkeley, Tech. Rep. UCB/ERL M03/32, August 2003.

[18] M. Gschwind and D. Maurer, "An Extendible MIPS-I Processor in VHDL for Hardware/Software Co-Design," in *Proc. of the European Design Automation Conference EURO-DAC '96 with EURO-VHDL '96*, GI. Los Alamitos, CA:

IEEE Computer Society Press, September 1996, pp. 548–553. [Online]. Available: citeseer.ist.psu.edu/gschwind96extendible.html

[19] M. R. Hartoog, J. A. Rowson, P. D. Reddy, S. Desai, D. D. Dunlop, E. A. Harcourt, and N. Khullar, "Generation of software tools from processor descriptions for hardware/software codesign," in *DAC '97: Proceedings of the 34th annual conference on Design automation.* New York, NY, USA: ACM Press, 1997, pp. 303–306.

[20] J. L. Hennessy and D. A. Patterson, *Computer Architecture; A Quantitative Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[21] "Itanium Instruction Set," http://www.intel.com/design/itanium/manuals/iiasdmanual.htm, Intel.

[22] *Workshop on Architecture Research using FPGA Platforms.* San Francisco, California: International Symposium on High-Performance Computer Architecture, 2005.

[23] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai, "PEAS-III: An ASIP Design Environment," in *Proceedings of International Conference on Computer Design.* Austin, TX: IEEE Computer Society Press, September 2000.

[24] e. a. John Hennessy, "The MIPS Machine," in *IEEE COMPCON*, 1982, pp. 2–7.

[25] J. Kasper, R. Krashinksy, C. Batten, and K. Asanovic, "A Parameterizable FPGA Prototype of a Vector-Thread Processor," in *Workshop on Architecture Research using FPGA Platforms in the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[26] V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman, "PICO: Automatically Designing Custom Computers," *Computer*, vol. 35, no. 9, pp. 39–47, 2002.

[27] A. Kejariwal, P. Mishra, J. Astrom, and N. Dutt, "HDLGen: Architecture Description Language driven HDL Generation for Pipelined Processors," University of California, Irvine, Tech. Rep. CECS Technical Report 03-04, 2003.

[28] K. Keutzer, S. Malik, and A. R. Newton, "From ASIC to ASIP: The Next Design Discontinuity," in *ICCD*, 2002.

[29] A. Kitajima, M. Itoh, J. Sato, A. Shiomi, Y. Takeuchi, and M. Imai, "Effectiveness of the ASIP design system PEAS-III in design of pipelined processors," in *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation.* New York, NY, USA: ACM Press, 2001, pp. 649–654.

[30] C. Kozyrakis and K. Olukotun, "ATLAS: A Scalable Emulator for Transactional Parallel Systems," in *Workshop on Architecture Research using FPGA Platforms in the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[31] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault, D. Cashman, D. Galloway, M. Hutton, C. Lane, A. Lee, P. Leventis, S. Marquardt, C. McClintock, K. Padalia, B. Pedersen, G. Powell, B. Ratchev, S. Reddy, J. Schleicher, K. Stevens, R. Yuan, R. Cliff, and J. Rose, "The stratix ii logic and routing architecture," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays.* New York, NY, USA: ACM Press, 2005, pp. 14–20.

[32] D. M. Lewis, V. Betz, D. Jefferson, A. Lee, C. Lane, P. Leventis, S. Marquardt, C. McClintock, B. Pedersen, G. Powell, S. Reddy, C. Wysocki, R. Cliff, and J. Rose, "The stratix$^{\text{tm}}$ routing and logic architecture." in *FPGA*, 2003, pp. 12–20.

[33] A. Lodi, M. Toma, and F. Campi, "A pipelined configurable gate array for embedded processors," in *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays.* New York, NY, USA: ACM Press, 2003, pp. 21–30.

[34] S.-L. Lu, E. Nurvitadhi, J. Hong, and S. Larsen, "Memory Subsystem Performance Evaluation with FPGA based Emulators," in *Workshop on Architecture Research using FPGA Platforms in the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[35] S. McFarling, "Combining Branch Predictors," Digital Equipment Corporation - Western Research Laboratory, Tech. Rep. WRL Technical Note TN-36, 1993.

[36] "Modelsim," http://www.model.com, Mentor Graphics.

[37] P. Metzgen, "A high performance 32-bit ALU for programmable logic," in *Proceeding of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*. ACM Press, 2004, pp. 61–70.

[38] ——, "Optimizing a High-Performance 32-bit Processor for Programmable Logic," in *International Symposium on System-on-Chip*, 2004.

[39] "MIPS," http://www.mips.com, MIPS Technologies.

[40] P. Mishra, A. Kejariwal, and N. Dutt, "Synthesis-driven Exploration of Pipelined Embedded Processors." in *VLSI Design*, 2004, pp. 921–926.

[41] K. Morris, "Embedded Dilemma," http://www.fpgajournal.com/articles/embedded.htm, November 2003.

[42] "Opencores.org," http://www.opencores.org/, Opencores.

[43] S. Padmanabhan, J. Lockwood, R. Cytron, R. Chamberlain, and J. Fritts, "Semi-automatic Microarchitecture Configuration of Soft-Core Systems," in *Workshop on Architecture Research using FPGA Platforms in the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[44] F. Plavec, "Soft-Core Processor Design," Master's thesis, University of Toronto, 2004.

[45] W. Qin, S. Rajagopalan, M. Vachharajani, H. Wang, X. Zhu, D. August, K. Keutzer, S. Malik, and L.-S. Peh, "Design Tools for Application Specific Embedded Processors," in *EMSOFT '02*, October 2002.

[46] O. Schliebusch, A. Chattopadhyay, and M. Steinert, "RTL Processor Synthesis for Architecture Exploration and Implementation," in *DATE 2004: Conference on Design, Automation & Test in Europe*, 2004. [Online]. Available: citeseer.ist.psu.edu/schliebusch04rtl.html

[47] O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr, "Architecture Implementation Using the Machine Description Language LISA," in *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*. IEEE Computer Society, 2002, p. 239.

[48] "CoCentric," http://www.synopsys.com/products/cocentric_studio/cocentric_studio.html, Synopsys.

[49] "Xtensa," http://www.tensilica.com, Tensilica.

[50] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau, "Architecture Description Languages for Systems-on-Chip Design," in *The Sixth Asia Pacific Conference on Chip Design Language*, 1999.

[51] "XiRisc," http://www.micro.deis.unibo.it/~campi/XiRisc/, University of Bologna.

[52] "MiBench," http://www.eecs.umich.edu/mibench/, University of Michigan.

[53] "MINT simulation software," http://www.cs.rochester.edu/u/veenstra/, University of Rochester.

[54] "RATES - A Reconfigurable Architecture TEsting Suite," http://www.eecg.utoronto.ca/~lesley/benchmarks/rates/, University of Toronto.

[55] M. Vachharajani, "Microarchitecture Modeling for Design-Space Exploration," Ph.D. dissertation, Princeton University, 2004.

[56] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August, "Microarchitectural Exploration with Liberty," in *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.

[57] K. Veenstra, "Altera Corporation," Private Communication, 2005.

[58] N. H. Weste and D. Harris, *CMOS VLSI Design A Circuits and Systems Perspective*. Boston, MA, USA: Pearson Education Inc., 2005.

[59] S. Wilson, "Roll Your Own Micro," *Chip Design Tools, Technologies, and Methodologies*, February 2004.

[60] "MicroBlaze," http://www.xilinx.com/microblaze, Xilinx.

[61] "Xilinx Virtex II Pro," http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Virtex-II+Pro+FPGAs, Xilinx.

[62] P. Yiannacouras, "SPREE," http://www.eecg.utoronto.ca/~yiannac/SPREE/.