PRISC: Programmable Reduced Instruction Set Computers

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Citable link</td>
<td><a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:25620498">http://nrs.harvard.edu/urn-3:HUL.InstRepos:25620498</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA">http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA</a></td>
</tr>
</tbody>
</table>
PRISC: Programmable Reduced Instruction Set Computers

A thesis presented

by

Rahul Razdan

The Division of Applied Sciences

in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the subject of

Computer Science

Harvard University
Cambridge, Massachusetts

May 1994
Copyright
1994 by Rahul Razdan
All rights reserved

This paper is available from the Center for Research in Computing Technology, Division of Applied Sciences, Harvard University as technical report TR-14-94.
Abstract

This thesis introduces Programmable Reduced Instruction Set Computers (PRISC) as a new class of general-purpose computers. PRISC use RISC techniques as a base, but in addition to the conventional RISC instruction resources, PRISC offer hardware programmable resources which can be configured based on the needs of a particular application. This thesis presents the architecture, operating system, and programming language compilation techniques which are needed to successfully build PRISC. Performance results are provided for the simplest form of PRISC -- a RISC microprocessor with a set of programmable functional units consisting of only combinational functions. Results for the SPECint92 benchmark suite indicate that an augmented compiler can provide a performance improvement of 22% over the underlying RISC computer with a hardware area investment less than that needed for a 2 kilobyte SRAM. In addition, active manipulation of the source code leads to significantly higher local performance gains (250%-500%) for general abstract data types such as short-set vectors, hash tables, and finite state machines. Results on end-user applications that utilize these data types indicate a performance gain from 32%-213%.
Preface

During the exploration of these class of computers, the insights of some individuals were invaluable. First and foremost, I would like to thank Mike Smith, my advisor, for his help in focusing my ideas, and for helping me avoid some potholes along the way. I would also like to thank Bill Grundman for his insightful discussions on custom CMOS implementation techniques. Mark Firstenberg and Ed McLellan were very helpful in providing detailed information on the microarchitectures of two recent high performance VAX and ALPHA architecture implementations. Also, I would like to thank Dilip Bhavsar and Lih Weng for their help with polynomial-based hashing methods. In addition, Lee Peterson and the whole GEEP staff at Digital Equipment Corporation were very supportive during this endeavor. On a personal note, I would like to thank my sister (Shefali) and mother (Veena) for their support and encouragement, and Simone for her patience and support during the many long hours it has taken to complete this thesis.
1. Introduction .............................................................. 1
2. Previous Work .......................................................... 6
   2.1 Board-Level Solutions ........................................... 7
   2.2 System-Level Solutions ........................................ 10
3. PRISC-1 Architecture ............................................... 14
   3.1 Programmable Hardware ....................................... 15
   3.2 PFU Microarchitecture ........................................ 17
   3.2 Instruction Set Architecture ................................. 19
   3.3 Software Architecture ...................................... 22
4. PRISC Compilation Techniques .................................... 27
   4.1 Function Width Analysis ...................................... 28
   4.2 Basic Block Optimizations ................................... 32
     4.2.1 BB-Expression ........................................... 33
     4.2.2 BB-Table-Lookup ....................................... 33
   4.3 Control Flow Graph Optimizations .......................... 36
     4.3.1 Boolean-CFG Optimization ............................. 36
     4.3.2 Jump-CFG Optimization ................................ 40
   4.4 Loop Optimizations .......................................... 44
   4.5 Library Routines ............................................ 49
   4.6 Summary ...................................................... 49
5. Hardware Synthesis Techniques .................................... 52
   5.1 PFU-LOGIC Instruction conversion .......................... 53
   5.2 CFG Conversion .............................................. 54
   5.3 Logic Minimization .......................................... 56
   5.4 LUT Optimization ............................................ 58
   5.5 LUT Placement and Routing ................................ 59
   5.5 Conclusions ................................................ 61
6. Performance Modeling and Results ................................ 64
   6.1 Performance Model ........................................... 64
   6.2 Performance Results ......................................... 67
   6.3 Summary ...................................................... 70
7. Software Acceleration Techniques ................................ 73
   7.1 Short-Set Vectors ............................................ 74
     7.1.1 SSV Set Functions ...................................... 74
7.1.2 Packed SSV Set Functions ........................................ 75
7.1.3 Packed SSV Test Functions ...................................... 77
7.2 Hash Tables ............................................................ 78
  7.2.1 Polynomial Division Hashing ................................ 79
  7.2.2 Integer Division Hashing ...................................... 81
7.3 Finite State Machine Evaluation .................................. 83
7.4 Results .................................................................. 85
8. Conclusions ................................................................ 89
9. Future Work ............................................................. 92
Appendix A: PFU Circuit Design ....................................... 95
  A.1 Basic Components .................................................. 96
  A.2 Layered PFUs ........................................................ 97
  A.3 PFU Example Circuit Design .................................. 100
  A.4 Conclusions ......................................................... 103
Appendix B: Multi-cycle PFUs ............................................ 105
References .................................................................. 110
List of Figures

Figure 1-1: Abstraction Example 1
Figure 3-1: PRISC-1 Architecture 14
Figure 3-2: Fuse Technology 15
Figure 3-3: Example PFU Functional View 18
Figure 3-4: PRISC Build Process 23
Figure 4-1: Function Width Analysis 30
Figure 4-2: Control Flow Rules 31
Figure 4-3: Boolean Expression Evaluation 33
Figure 4-4: Ternary NAND Evaluation 34
Figure 4-5: Population Counter 35
Figure 4-6: Boolean-CFG Example 37
Figure 4-7: PFU Boolean-CFG Example 39
Figure 4-8: Jump-CFG Generic Example 40
Figure 4-9: Jump-CFG Example 42
Figure 4-10: Jump-CFG Result 43
Figure 4-11: Loop Unrolling 45
Figure 4-12: PFU-Loop Optimization 46
Figure 4-13: String Length Example 48
Figure 5-1: Instruction Conversion 53
Figure 5-2: CFG Conversion 54
Figure 5-3: Structural Network Example 56
Figure 5-4: Logic Minimization Example 57
Figure 5-5: LUT Optimization 59
Figure 6-1: Performance Modeling 65
Figure 6-2: SPECint92 Results 68
Figure 6-3: CMPPT Example 69
Figure 6-4: CMPPT PFU 70
Figure 7-1: TOUPPER Example 75
Figure 7-2: ESPRESSO Example 76
Figure 7-3: Zero Byte PFU 77
Figure 7-4: LFSR Example 80
Figure 7-5: Integer Hashing  82
Figure 7-6: Inverter FSM  84
Figure 7-7: CAD Result  86
Figure A-1: PFU Interface  92
Figure A-2: LUT and Interconnect Resources  95
Figure A-3: PFU Microarchitecture  97
Figure A-4: Layered PFU Critical Path  98
Figure A-5: Interconnect and LUT Stage  100
Figure A-6: Self-Timed PFU  101
Figure B-1: Normalize Multi-Cycle PFU  105
Figure B-2: Multi-Cycle PFU  106
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>BigNum Multiplier Operations</td>
<td>9</td>
</tr>
<tr>
<td>2-2</td>
<td>Compiled-Code Simulation Results</td>
<td>11</td>
</tr>
<tr>
<td>4-1</td>
<td>Ternary NAND Function</td>
<td>29</td>
</tr>
<tr>
<td>4-2</td>
<td>CFG Resolution Function</td>
<td>32</td>
</tr>
<tr>
<td>4-3</td>
<td>Jump-CFG Generic Example</td>
<td>41</td>
</tr>
<tr>
<td>6-1</td>
<td>SPECint92 Static Optimization Instances</td>
<td>67</td>
</tr>
<tr>
<td>7-1</td>
<td>RISC Data Types</td>
<td>73</td>
</tr>
</tbody>
</table>
1. Introduction

Abstraction is one of the most important concepts in the field of computer science and engineering. A key component of using abstraction as a tool is the interface definition between different levels of abstraction. For example in the domain of hardware design, a boolean gate model is often provided as an interface definition between the boolean abstraction and an underlying technology. Using this interface definition, the problem of building a logic gate from the basic resources provided in a base technology is separated from the problem of building a large complex logic network. The imposition of an interface definition splits a complex problem into two simpler problems. However, the imposition of an interface definition often leads to non-optimal solutions.

![Abstraction Example](image_url)

Figure 1-1: Abstraction Example

Figure 1-1 shows two implementations for a multiplexer function in a Complementary Metal Oxide Semiconductor (CMOS) technology. The logic circuit in Figure 1-1(b) uses strictly the logic gate abstraction, and this abstraction leads to an implementation with
three NAND gates and one inverter. The CMOS circuit in Figure 1-1(c) optimizes past the interface definition, and it requires only two transistors and one inverter. Since a 2-input NAND requires four transistors and an inverter requires two transistors, optimizing past the interface definition reduces the transistor cost from 14 transistors to four transistors. More importantly, the CMOS implementation is also faster.

In the domain of computer design, Amdahl in 1964 defined the interface between computer software and computer hardware as a fixed instruction set. This interface provided a mechanism for the software to control the given hardware resources, but did not allow for dynamic changes in the hardware/software interface. This thesis introduces a new class of computers called Programmable Reduced Instruction Set Computers (PRISC) which optimize across this interface by dynamically creating instructions based on the needs of a given application.

PRISC computers maintain all of the demonstrated advantages of RISC computers, e.g. fixed instruction formats and load/store architecture. However, in addition to offering the conventional set of RISC instructions, PRISC computers offer the ability to create application-specific instructions that are implemented in dynamically programmable functional units (PFUs) and are used to accelerate application performance. The instruction generation process is driven by the particular computations found in a user application, so PRISC computers avoid the semantics gap problems of CISC [1]. Instead, sophisticated compiler algorithms "extract" hardware instructions from software programs to effectively utilize the PFUs.
In general, a PFU cannot compete on an equal footing with a highly customized RISC functional unit. Lewis [3] reports a factor of three performance difference between programmable circuits and mask-programmed circuits. However, despite their inherent hardware disadvantages, PFUs can still accelerate application performance by evaluating boolean functions with low hardware complexity. These functions were not included as instructions in the base RISC instruction set because they did not provide a significant performance gain across a wide variety of applications. Yet, these instructions may be very important for a particular application. Because PFUs can be dynamically programmed, PRISC computers accelerate performance by constructing an instruction for any particular application.

The most general computational model for a PFU is a multi-cycle sequential state machine. Iterative hardware solutions for square-root or transcendental function evaluation are good examples of this class of PFU (See Appendix B). The general model however introduces synchronization complexities between the PFU and the other RISC functional units. For this thesis, we discuss a simpler model that implements a combinational function with two inputs and one output. Furthermore, this combinational function is constrained to fit within the evaluation phase of a RISC pipeline. Given these two restrictions, PFUs can use the same synchronization mechanisms as the other RISC functional units. We shall refer to this first implementation of the PRISC architecture as the PRISC-1 computer.
The next chapter describes the previous work in the field of programmable computing machines. Chapter 3 describes the software and hardware architectures for PRISC computers. Chapter 4 describes the techniques used for hardware extraction in a high-level language compilation environment, and Chapter 5 describes the techniques used for the hardware synthesis of the extracted hardware. Chapter 6 discusses our performance modeling environment and the results obtained from our performance model for the SPECint92 benchmark suite. Chapter 7 discusses the utility of PFU resources for abstract data types such as short-set vectors, hash tables, and finite state machines. Chapter 8 provides conclusions, and Chapter 9 provides future work. Appendix A discusses the circuit design issues in building a PFU. Finally, Appendix B discusses extensions of the combinational PFU model to produce a multi-cycle sequential model.
2. Previous Work

Due to the recent introduction of dynamically programmable hardware devices, research in the use of these devices for general-purpose computing is still in its inceptive stages. We are only aware of the work of Iseli and Sanchez [4] who offer a processor microarchitecture consisting solely of programmable functional units in a VLIW environment. Their computer has no built-in functional units for integer and floating point addition and multiplication. Because of this organization, the clock frequency for their prototype processor currently runs at 5 Mhz. This 5 Mhz clock rate is significantly below the clock frequencies of today's RISC microprocessors (typically over 60 Mhz). Our PRISC techniques are designed to be added directly to today's RISC processors without changing the clock rate. Thus, we can easily augment high performance processors such as the 200 Mhz ALPHA DECchip 21064 [5] with a PFU.

In addition, Iseli and Sanchez do not offer any techniques to compile programs from a general-purpose language such as C to their totally programmable environment, nor do they offer any performance analysis of their computer on accepted benchmarks for general-purpose computer performance. Our study provides techniques for hardware extraction, and it analyzes the performance of PRISC computers on the SPECint92 benchmark suite.

In contrast with the sparse work in programmable hardware for general-purpose applications, there has been a great deal of research on solutions which solve domain
specific problems with programmable hardware. This work can be split into two major categories: board-level solutions and system-level solutions.

2.1 Board-Level Solutions

The board-level work was pioneered by the PAM [6] group in Paris. Their system consists of XILINX [15] programmable boards that are connected to the I/O bus of a general-purpose workstation. Their intention is to partition the computation for a particular problem between the XILINX boards and the workstation processor. The PAM system has shown good results for over 10 applications [11], and the most impressive results were obtained for long integer multiplication [9] and RSA decryption [10]. The SPLASH [7] group from Brown University has mimicked the PAM model, and has been successful in solving problems such as text searching, DNA comparison, and edge detection for graphics applications. The common characteristic of all of these operations is that they exhibit a large amount of computation with limited communication demands. The PAM implementation for long integer multiplication is illustrative of the board-level solutions. The basic operation computed for long integer multiplication is:

\[ C_{out} + P_{0..n+15} \leftarrow S_{0..n+15} + A_{0..15} \times B_{0..n-1} \]

where, \( C_{out} \) is the carry out, \( P, S, A \) and \( B \) are long integers and their subscripts refer to successive 32-bit digits. To build this operation, a 512x32 bit multiplier is built on a PAM board, and seven internal registers: \( B_{<0:31>} \), \( SE_{L} \), \( SP_{<0:31>} \), \( A_{<0:544>} \), \( P_{<0:544>} \), \( PO_{<0:31>} \), and \( PF_{<0:31>} \) are used to support the four I/O operations needed to communicate between the workstation processor and the PAM boards. The \(<n>\) notation
refers to the bit subscript of the registers. The I/O operations and PAM hardware are designed to carefully match the bandwidth of the I/O bus. For the VME bus (50 Mbit/sec), the I/O operations are defined as follows:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WriteA</td>
<td>$B \leftarrow \text{MemData}, SEL \leftarrow a$</td>
</tr>
<tr>
<td>WriteB</td>
<td>$B \leftarrow \text{MemData}, SEL \leftarrow b$</td>
</tr>
<tr>
<td>ReadP</td>
<td>$\text{MemData} \leftarrow \text{PF}$</td>
</tr>
</tbody>
</table>
| WriteS      | $SP \leftarrow \text{MemData}, PF \leftarrow PO$  
  $\text{if } SEL = a \text{ then}$  
  $P \leftarrow P + SP \times 2^{512}, A \leftarrow A + B \times 2^{512}, A \leftarrow A/2^{32}$  
  $\text{if } SEL = b \text{ then}$  
  $P \leftarrow P + SP \times 2^{512} + A \times B, PO \leftarrow P \text{ mod } 2^{32}, P \leftarrow P/2^{32}$ |

$\text{MemData}$ is the 32-bit data coming off the I/O bus. Given these instructions, the workstation performs a big number multiply by the sequence of operations shown in Table 2-1. This table represents an access sequence with time increasing from left to right and top to bottom. Each column represents a particular type of instruction. The column entries indicate the data item associated with the instruction. The left three columns contain the WriteA, WriteB, and WriteS instructions that pass arguments to the PAM board. The rightmost column contains the ReadP instructions that retrieve the results. The boxed section of the table represents the results of one long integer multiplication. First, all the $A$ operands and the first 16 $S$ operands are sent to the PAM board using WriteA and WriteS instructions respectively. Second, the $B$ operands are sent to the board using the WriteB instructions. For every $B$ operand, the following WriteS produces a result that can be moved back to the workstation processor.
### Table 2-1: BigNum Multiplier Operations

<table>
<thead>
<tr>
<th>WriteA</th>
<th>WriteB</th>
<th>WriteS</th>
<th>ReadP</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>---</td>
<td>S₀</td>
<td>(Prev P)</td>
</tr>
<tr>
<td>A₁</td>
<td>---</td>
<td>S₁</td>
<td>(Prev P)</td>
</tr>
<tr>
<td>A₂</td>
<td>---</td>
<td>S₂</td>
<td>(Prev P)</td>
</tr>
<tr>
<td>.</td>
<td>---</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>---</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>A₁₅</td>
<td>---</td>
<td>S₁₅</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>B₀</td>
<td>S₁₆</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>B₁</td>
<td>S₁₇</td>
<td>P₀</td>
</tr>
<tr>
<td>---</td>
<td>.</td>
<td>.</td>
<td>P₁</td>
</tr>
<tr>
<td>---</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>Bₙ₋₁</td>
<td>Sₙ₋₁₋₁₅</td>
<td>Pₙ₋₂</td>
</tr>
<tr>
<td>A₀'</td>
<td>---</td>
<td>S₀'</td>
<td>Pₙ₋₁</td>
</tr>
<tr>
<td>A₁'</td>
<td>---</td>
<td>S₁'</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>---</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>A₁₅'</td>
<td>---</td>
<td>S₁₅'</td>
<td>Pₙ₋₁₅</td>
</tr>
</tbody>
</table>

Using this scheme, the combination of the PAM board and a 16MHz MC68020 workstation processor were able to provide a performance improvement of 25X over a software implementation running on the workstation processor. Unfortunately, these board-level methods incur a high overhead when communicating between the host CPU and the programmable logic. This significant overhead limits the applicability of this approach to a class of algorithms that have a combination of high computational complexity and low communication overhead. To increase the applicability of this approach, the communication protocol and the hardware design on these boards must be optimized using custom hardware design techniques.
This custom design methodology leads to highly inventive algorithms, but does not cater to the general-purpose computing environment. Hand-optimized algorithms cannot easily track changes in I/O protocols and programmable device architectures. In essence, these boards provide a fast hardware prototyping capability, but every algorithm requires a new custom hardware design. In contrast, PFUs in a PRISC computer offer less programmable resources, but the resources reside inside the CPU chip. Communication costs (bandwidth and latency) are minimized, and a larger class of algorithms can benefit from the performance gain offered by a PFU. Also, the hardware design is performed automatically based on high-level language compilation, so the investment in software development can be maintained across several generations of PRISC processors.

2.2 System-Level Solutions

In contrast to board-level solutions, system-level solutions move the whole application to a set of programmable boards. Commercial companies such as Quickturn [8] have shown performance gains of 1000X-10,000X over the fastest general-purpose processors using this method for the logic emulation application. This astounding performance gain can be explained by closely examining the execution of the logic emulation application on general-purpose processors.

The objective of logic emulation is to evaluate boolean networks. In a general-purpose processor, all the nodes for a large network cannot be stored in registers, so the generic scheme for the evaluation of any two-input logic gate in the network is:
LB r1, var1(array_top)
LB r2, var2(array_top)
Logic_op r3, r1, r2
SB r3, var3(array_top)

where the node values are stored in a byte array. Array_top is the top of the node array, and the var fields are constant offsets that specify the input and output nodes for evaluation. More address calculation instructions are needed for extremely large logic networks. The latency associated with the load and store instructions depends on the memory hierarchy for the processor, and the logic_op is a single cycle instruction. Table 2-2 shows the results of a compiled-code simulation of various benchmark networks [31] on a DECstation 7000 system [30]. The 5th column shows the performance of the simulation in terms of cycles per second, and the last column shows effective amount of time spent to evaluate each gate. The cycle time of the CPU is 5ns and the underlying process technology can evaluate a 2-input gate in less than 0.5ns.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Gates</th>
<th>Nodes</th>
<th>ALPHA-INS</th>
<th>Time (c/sec)</th>
<th>ns/gate</th>
</tr>
</thead>
<tbody>
<tr>
<td>S9234</td>
<td>4391</td>
<td>4391</td>
<td>15282</td>
<td>3460</td>
<td>65</td>
</tr>
<tr>
<td>S13207</td>
<td>5319</td>
<td>5561</td>
<td>18421</td>
<td>2827</td>
<td>66</td>
</tr>
<tr>
<td>S15850</td>
<td>14418</td>
<td>14963</td>
<td>50891</td>
<td>1004</td>
<td>69</td>
</tr>
<tr>
<td>S35932</td>
<td>14840</td>
<td>16605</td>
<td>57551</td>
<td>954</td>
<td>70</td>
</tr>
<tr>
<td>S38584</td>
<td>16275</td>
<td>16052</td>
<td>54542</td>
<td>874</td>
<td>70</td>
</tr>
</tbody>
</table>

Thus, there is a 10X performance difference between evaluating a logic instruction and evaluating a logic gate. In a logic network, the communication between the gates is performed using wires. In logic simulation on a processor, the communication between the
gates is performed using the memory hierarchy. The last column in Table 2-2 reflects the combination of these two effects, and also any potential usage of registers to store intermediate results. For the larger networks, the combination of these effects leads to a performance improvement of (70/0.5) 140X. Finally, a logic network evaluates all gates in parallel, but a conventional RISC processor evaluates each gate sequentially. Any particular rank in the logic network can contain 20-100 gates which can be evaluated in parallel, thus leading to a performance gain in the range of 1000X-10,000X.

While this speedup is impressive, it is limited to the domain of logic emulation, and cannot be easily extended to general-purpose computing. Today, general-purpose computing is based on the von Neumann model which separates computation from memory. Extensions of this model support objects such as pointers and stacks. These objects cannot be supported by a system-level programmable solution without building a general-purpose processor with programmable hardware. Of course, the performance of this processor would be poor in comparison with a conventional processor built using custom hardware design techniques.
3. PRISC-1 Architecture

![Figure 3-1: PRISC Datapath](image)

PRISC computers offer a relatively small amount of programmable resources -- typically 10X less than that found in board-level designs and 100X-1000X less than that found in system-level designs. However, as Figure 3-1 shows, PFUs attach directly to the datapath of RISC CPUs just like the other functional units (FU1 and FU2), so the communication cost (bandwidth and latency) between the PFU and the other functional units of the CPU is minimized. Like the other functional units, a PFU has two input operand ports and a single output operand port. In addition, a PFU contains two additional ports which are used for programming purposes. The next two sections discuss the hardware components of a PFU. Section 3.3 discusses the instruction set architecture needed to build and control a PFU, and Section 3.4 discusses the software architecture which will use PFU hardware resources.
3.1 Programmable Hardware

Figure 3-2: Fuse Technology

The programmable fuse is the key enabling technology for building a PFU. The area and electrical characteristics of this fuse are critical to the performance of a PFU. Figure 3-2 shows the three most popular implementations for a programmable fuse: anti-fuse, eprom-fuse, and sram-fuse. Anti-fuses, the oldest fuses, consist of a thin conductor which connects the two terminals (A and B) of the fuse. Since the two terminals are already connected, no action is required to place the fuse in the on-state. To place the fuse in the off-state, a high current which induces electromigration in the conductor is applied between the A and B terminals. The process of electromigration severs the connection between the A and B terminals. Of the three implementations, anti-fuses consume the smallest area and offer the lowest on-state resistance. However, once the conductor is
severed, the fuse cannot be reprogrammed. Since a PFU needs to be reprogrammable, the anti-fuse cannot be used to build PFUs.

Eprom-fuses are special semiconductor devices built by augmenting a standard Complementary Metal Oxide Semiconductor (CMOS) process. A standard CMOS n-channel transistor consists of two n-type semiconductor wells (the source and drain terminals) connected by a p-type semiconductor channel. A metal conductor, (the gate terminal) separated by a thin oxide, overlaps the p-type channel. When a voltage is placed between the gate and source terminals, the p-type channel is converted to a n-type channel, and the source and drain terminals are connected. A eprom-fuse augments this structure with an additional layer of trap oxide and programming metal. The fuse terminals A and B are connected to the source and drain terminals of the standard CMOS n-channel transistor. To program a eprom-fuse, a very high voltage is created between the gate and programming metal terminals. This voltage injects electrons into the oxide layer separating the two metal layers. After programming, the electrons become trapped in the oxide, and provide sufficient voltage between the gate and source terminals to program the p-type channel. The eprom-fuse can be reprogrammed, but requires a special programming mode that uses very high voltage levels. In addition, continuous reprogramming using high voltage may create a reliability problem for the breakdown of the trap oxide layer. Because of the need for a special electrical programming mode and reliability concerns with continuous programming, eprom-fuses are not appropriate for building PFUs.
Like the eeprom-fuse, a sram-fuse uses a standard n-channel transistor for the programming switch, but instead of a trap oxide, a static latch is used to maintain the programming level. Since the fuse requires only n-channel and p-channel devices, sram-fuses do not require a change in the standard CMOS fabrication process. The static latch requires six additional transistors, but allows the fuse to be programmed by just updating the value stored in the latch. Because of its ease of reprogramming, the sram-fuse is used as the basic fuse primitive for building PFUs.

### 3.2 PFU Microarchitecture

Given the sram-fuse, the design of the PFU microarchitecture is an interesting, non-trivial hardware design problem. The timing constraint is to build a functional unit that can fit into the evaluation phase of a RISC CPU pipeline, while the design goal is to maximize the number of "interesting" functions which can be implemented by the PFU. A boolean function is "interesting" if it cannot be evaluated efficiently with conventional RISC instructions and if the evaluation of this function consumes a significant proportion of an application's execution.

Figure 3-3 shows an example functional implementation for a PFU. A PFU has two basic components: an interconnection matrix and logic evaluation units. In the example, the interconnection matrix is implemented by a CMOS n-channel transistor that is controlled by a memory cell. Depending on the value in the memory cell, the two lines are disconnected or connected. A logic evaluation unit consists of a Look Up Table (LUT) which implements a truth table in hardware. A \( n \)-input LUT consists of a multiplexer...
connected to $2^n$ memory cells. All the memory cells are viewed as a large SRAM memory which are loaded by using the \textbf{Paddr} and \textbf{Pdata} ports of the PFU. The combination of the LUT cells and the programmable interconnect allows the PFU to emulate a range of logic functions. Programming a PFU to implement a particular function consists of loading the appropriate values into the interconnection matrix memory cells and the LUT memory cells.

![Figure 3-3: Example PFU Functional View](image)

Because both the interconnection matrix and the logic evaluation units make prodigious use of memory cells, the layout cost of a PFU is dominated by the cost of these memory cells. Appendix A provides a detailed analysis of the circuit design issues for a PFU. In this analysis, we have found that a PFU like the one shown in Figure 3-3, requires 61,056 transistors for a 64-bit word machine, and over 90\% of these transistors are required to implement memory cells. Since the layout costs of PFUs track the layout costs of SRAMs, the PFU shown in Figure 3-3 takes considerably less area than a 2 Kilobyte SRAM which requires at least 100,000 transistors. In terms of real estate, existing processors can easily
accommodate over 16 kilobytes of SRAM cache [5], and many even have translation
lookaside buffers [21] which are larger than a PFU.

3.2 Instruction Set Architecture

To program and operate PFUs, we define the Execute PFU (expfu) instruction. The format of this instruction is presented below in MIPS [24] notation:

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>25</th>
<th>20</th>
<th>15</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>expu</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>LNum</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Expfu is a 32-bit register-to-register instruction which evaluates a boolean function. The inputs to the boolean function are the registers specified by the rs and rt fields, and the output is placed in the register specified by the rd field. The encoding used for the expfu instruction mimics the encoding used for the other RISC register-to-register instructions, so the fast decode features of RISC machines are maintained. In the PRISC-1, this instruction always executes in the evaluation phase of the RISC CPU pipeline and does not retain any sequential state after evaluation. The 11-bit LNum field specifies the logical-PFU number for this instruction.

A logical-PFU number is associated with each "extracted" function in an application. The 11-bit field allows for a capacity of 2048 different PFU programming configurations. The programming information for each logical-PFU is located in the data space of the application. This information is accessed using the logical-PFU number. Associated with
each PFU is an 11-bit register, Pnum, which contains the logical PFU currently programmed into the physical PFU. If the L.Pnum in the instruction matches the value in the Pnum register, the expfu instruction executes normally. However, if there is a mismatch, an exception is raised, and the PFU must be loaded with the correct programming information by an exception handler.

The latency of the expfu exception handler depends on the density of the programming memory, and the hardware resources allocated for PFU programming. In practice, the PFU programming memory is sparsely populated (typically less than 15% of the bits are asserted). A scheme which uses a hardware reset to initialize all the bits to a zero value, and only programs the words which contain one-bits can significantly reduce the overall latency of PFU programming. For example assuming a 20% utilization rate, the programming data of PFU with 2048 memory cells requires only 51 64-bit data and address words.

Given this programming data, there is a tradeoff between hardware cost and programming latency. A low cost solution for programming might use the PAL instruction model from the ALPHA architecture to sequentially load the PFU programming memory using normal load instructions. Assuming five cycles for the loads and one cycle for the PFU memory assignment, the PAL approach can program a PFU in 11*51=561 cycles. A high performance solution might use a pipelined approach in combination with high bandwidth access to memory. Using such a high cost approach, the latency for programming can be brought below 100 cycles.
The **Pnum** register and the programming memory of the PFU add context to the application process, and we must handle this extra state during context switching. In our scheme for context switching, the logical-PFU number zero is reserved to represent an unprogrammed PFU. During a context switch, the hardware clears each **Pnum** register. The next evaluation of the **expfu** instruction triggers an exception that then programs the PFU. Thus, similar to the handling of virtual caches or TLBs, the cost of context switching a PFU is determined by the need to refill the programming memory on the next **expfu** instruction. Since the frequency of context switches is very low (50-100 context switches per second [25]), the incremental cost of faulting in a PFU program after a context switch should have negligible effect on overall performance.

A typical mode of operation would use the **expfu** instruction inside the inner loop of an application to accelerate performance. For example, consider a compute-intensive application such as integer factorization using the Pollard-Raho method [23] which is heavily dependent on the greatest common divisor (gcd) calculation. The gcd calculation between two short integers \(a\) and \(b\) has the following form:

1. \(a = \text{norm}(a)\);
2. \(b = \text{norm}(b)\);
3. if \((a > b)\) \(a = a - b\); else \(b = b - a\);

where **norm** is a normalization of the input variable. The normalization operation is the bottleneck to the whole integer factorization algorithm. A PFU can perform this the normalization for a short integer very quickly. The first time the **expfu** instruction for the
**norm** is encountered, an exception is raised, and the PFU is programmed with the normalization boolean function. Afterwards, the **norm** operation executes in a single cycle.

### 3.3 Software Architecture

To successfully build a PRISC computer, general-purpose applications must effectively use the hardware resources described in the previous chapter. Figure 3-4 shows our compilation process for a PRISC machine. The left side of the figure is similar to the language compilation environment for a high performance RISC machine. The inputs to the compiler are the source files of a high-level programming language such as C and execution profile information from a performance analysis program such as **pixie** [2].

Conventional compilers perform optimizations such as procedure inlining, loop optimizations, and constant propagation, and produce an abstract intermediate description which is consumed by a code generator. In the GNU environment, this description is called Register Transfer Level (RTX). However, unlike conventional RISC compilers, our PRISC compiler contains a step before code generation, called hardware extraction, that identifies the set of RISC instructions which could be implemented in a PFU. We use the profile information to prioritize the results of the hardware extraction step. Chapter 4 will describe these hardware extraction techniques in more detail. Once partitioning between hardware and software is performed, the instructions marked for hardware synthesis are
processed by an automatic compilation process consisting of RTL, logic, and layout synthesis tools.

![Figure 3-4: PRISC Build Process](image)

In the PRISC-1 machine, the PFUs implement only combinational functions, so the RTL synthesis step is unnecessary. If a PFU was allowed to implement sequential logic to model software loops, this step would be necessary to generate the control for the sequential PFU state machines. The logic synthesis process takes as input a combinational function in the form of a set of instructions, and produces as output a netlist of LUTs. During this step, logic minimization algorithms reduce the number of LUTs and interconnect resources which are used by the input function. Finally, the physical synthesis process performs the placement and routing needed to fit the LUT netlist on the resources offered by the PFU.
Chapter 5 provides a more detailed discussion of the algorithms used for logic synthesis, LUT minimization, and LUT place-and-route. Briefly, we have augmented standard algorithms [12,13,14] to the task of PFU synthesis. In general, these algorithms have a worst case performance behavior which cannot be bound by a polynomial function (NP-complete). However, the input functions generated by the hardware extraction algorithms are quite small in comparison with the typical size of problems handled by these algorithms. Thus, in most cases, the existing algorithms, augmented with our simplifications, can quickly synthesize functions for PFUs. A feedback process is built into the hardware extraction and synthesis steps which reduces the requested input function when resource constraints are violated. The output of hardware synthesis is a bit stream that programs the PFU.

The PRISC image builder combines the results of the hardware and software images. The PRISC image builder performs all the usual tasks of a conventional linker, but in addition, data memory is allocated for the bit stream generated by the hardware synthesis process. Finally, a PRISC loader can load and execute the binary image produced by the image builder. Like a conventional loader, the PRISC loader performs a compatibility check between the image and underlying computer for matching PFU resources. If the resources do not match, a binary incompatibility problem exists.

Binary compatibility is a desirable feature to maintain a software investment across a range of computer implementations. We can offer binary compatibility in a PRISC environment at the expense of some performance loss. In our scheme, the PRISC software compiler
retains the original base RISC instructions which were transformed to PFU hardware by the hardware synthesis tools. In the event of binary incompatibility, the PFU programming exception handler executes the base RISC code instead of programming the PFU. This scheme has two places for performance degradation. First, exception processing overhead is introduced for every _expfu_ instruction. Second, the evaluation cost of the boolean function by the base RISC instructions is more expensive than a PFU evaluation. Further, these two forms of performance degradation are introduced at the most critical portions of the application's code. However, we maintain binary compatibility across different PRISC generations, and a recompilation will again achieve maximal performance.
4. PRISC Compilation Techniques

Because dense boolean functions cannot be programmed into a PFU due to resource constraints, the key benefit of a PFU is the ability to evaluate relatively sparse boolean functions. The task of a PRISC compiler is to provide optimizations that utilize PFU resources for the evaluation of sparse boolean functions. The compiler must extract these functions from the two key types of operations in a program -- control and data operations. Fortunately, for control operations, most of the manipulations to the program counter are performed using pc-relative branching, so relatively sparse functions are able to evaluate these operations. Becker, Park, and Farrens [46] have confirmed this assertion by showing that on average 83% of an address trace contains redundant information. Data operations range in their density, but architectures and implementations are designed for the worst case dense functions. So, several sparse data operations can be combined into a PFU and evaluated in a single cycle.

Our PRISC compiler methodically analyzes the control and data operations in a program to extract the functions that should be implemented in a PFU. The first step in this process determines the density of the individual instructions. The density of an instruction is determined by the complexity of the function to be evaluated, and the bit-width of the instruction inputs and output. The complexity is readily available through the instruction type (ADD, AND, ..etc), but since the cost of short operation does not change the software evaluation cost for almost all RISC CPU instructions, the bit-width information is not easily available. For example, a byte ADD and a full word ADD have the same
resource costs in terms of pipeline scheduling for a software compiler, but the two operations have vastly different hardware costs. Thus, we have developed an algorithm that determines the bit-width for all the data instructions in the program. This algorithm is described in Section 4.1.

Using the results of bit-width analysis and the instruction type, every operation which can be easily placed in a PFU is labeled as a PFU-LOGIC instruction. These PFU-LOGIC instructions are used as a basis for three categories of hardware transformational techniques.

1. basic block optimizations;
2. control flow graph optimizations;
3. loop optimizations.

We describe each of these techniques in succeeding sections. The result of these transformations is a new intermediate representation which contains expfu instructions. Since the expfu instruction looks like the other RISC instructions, we can easily fold expfu instructions into any instruction scheduling algorithms.

4.1 Function Width Analysis

We use a ternary algebra, similar to the one used in most logic simulators [29], to find the bit width for every variable in the static code stream. The algebra is based on a partially ordered set \( \{0, 1, X\} \) with \( X < 0 \) and \( X < 1 \). The ordering represents the certainty of a variable state. The third value, \( X \), indicates an unknown state while 0 and 1 represent fully
defined states. All the computational instructions in the RTX have definitions in the boolean space. If we redefine these instructions in the ternary space with functions which are monotonic with respect to the partial ordering, we can use the results to determine the dynamic values of variables with a static analysis.

We can easily achieve the mapping from binary to monotonic ternary functions by offering ternary equivalents to the standard gates. For example, the ternary equivalent of a NAND gate would be:

<table>
<thead>
<tr>
<th>TABLE 4-1: Ternary NAND Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAND</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>X</td>
</tr>
</tbody>
</table>

Notice, the mapping from inputs to outputs is monotonic with respect to the partial ordering of our ternary set, so that given a function

\[ fn : \{0, 1, X\} \rightarrow \{0, 1, X\} \]

and elements \( a, b \in \{0, 1, X\} \) then the following property is true:

\[ a \leq b \Rightarrow fn(a) \leq fn(b) \]

Using the monotonic property, once a variable is declared with a 0 or 1 value in the static analysis, no value assignments in dynamic execution can change its state.

The algorithm for width analysis initializes all variables to the X state for every bit position. Then, a combination of forward and backward traversals are performed to prune the number of bits which are at an X. Forward traversals evaluate every instruction and
check to see if the evaluation changes the output bit vector. For example, for a load byte unsigned instruction, the upper bits are guaranteed to be at a zero value. An event-driven algorithm is used to follow the changes until quiescence.

Backward traversals check to see which bits of a variable are used in subsequent instructions, and proceed backwards over the code to eliminate unnecessary bit calculations. For example, if a variable was stored to memory using a store byte instruction and was not used elsewhere, any instructions which generated the inputs for the store need only generate 8 bits of information. The backward traversal algorithm also uses an event-driven algorithm to propagate changes throughout the application code.

![Diagram](image)

**Figure 4-1: Function Width Analysis**

For this algorithm to work correctly, we must know the forward and backward ternary mappings for all component instructions. In forward traversal, a function evaluation determines the output bits which might be at a X given the state of the input vectors. In backward traversal, an evaluation determines the input combinations which produce an output bit which is at a X state. The example adder in Figure 4-1 illustrates these
mappings. In the forward traversal case, given two bit vectors with only the lowest two bits in the X state, only three bits for the output can be in the X state. In the backward traversal case, given an output with only two bits in the X state, the inputs can be reduced to two bits in the X state.

![Diagram of Forward and Backward Traversal](image)

**Figure 4-2: Control Flow Rules**

The forward and backward mapping for individual instructions are sufficient to propagate the ternary bit vectors within basic blocks. To propagate past basic block boundaries, propagation rules for forward and backward traversal are needed for **splits** and **joins** within the Control Flow Graph (CFG). Figure 4-2 shows examples of the rules for **splits** and **joins**. In the forward traversal, a **split** of a variable vector performs a fanout function, and a **join** requires resolution. In the backward traversal, a **join** performs a fanout function.
function, and a **split** requires resolution. The resolution function for the **join** in the forward direction, and the **split** in the backward direction is shown in the following table:

<table>
<thead>
<tr>
<th>Path1/Path2</th>
<th>0</th>
<th>1</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Because of the data dependent nature of the number of iterations in loop, loops are conservatively handled by terminating all traversals propagating through a backward loop edge. However, a special analysis is performed to find fixed length loops. For these loops, the size of the iterator is reduced to the loop size, and the ramifications of that reduction are propagated using the basic block and control flow algorithms.

Given the bit values for all the variables in the application, we can easily calculate the hardware computational complexity of the individual instructions. After width analysis, all the instructions that can be individually placed in a PFU are marked as PFU-LOGIC instructions. In the integer portion of a typical RISC instruction set, only load/store, wide add, variable length shifts, multiply, and divide remain non-PFU-LOGIC instructions.

### 4.2 Basic Block Optimizations

Once all the instructions have been marked as PFU-LOGIC or non-PFU-LOGIC, we attempt to employ two hardware transformation techniques within the scope of a basic block: BB-Expression and BB-Table-Lookup.
4.2.1 BB-Expression

The first and most obvious basic block optimization is to combine logic expressions that take multiple instructions on RISC machines into a single PRISC instruction. That is, for every basic block carve out the sets of instructions which have been marked as PFU-LOGIC. Figure 4-3 shows a pair of PFU-LOGIC expression examples taken from the ESPRESSO [22] benchmark. The expression on the left contains a set of 32 bit-boolean expressions which manipulates a bit vector representation of boolean product terms, and the expression on the right contains two single bit compares which are combined by a one bit adder. In both examples, we reduce the multiple RISC instructions to a single PFU evaluation.

4.2.2 BB-Table-Lookup

Simple logic recognition is generally successful when an application uses bit vector data structures. However, when the logic expressions become very complex, truth tables are often used to perform the logic evaluations. These truth tables are presented in the form of
constant arrays in the general-purpose programming language, and are converted to a functional representation. A PFU can be used to evaluate the functional representation.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>OUT</th>
<th>Software Eval:</th>
<th>PFU Eval:</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>all r1,A,2</td>
<td>A&lt;1&gt;</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>00</td>
<td>or r1,B,r1</td>
<td>B&lt;0&gt;</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>00</td>
<td>add r1,T,r1</td>
<td>A&lt;1&gt;</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>00</td>
<td>ldb OUT,0(r1)</td>
<td>B&lt;0&gt;</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>01</td>
<td></td>
<td>A&lt;1&gt;</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>10</td>
<td></td>
<td>B&lt;0&gt;</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>10</td>
<td></td>
<td>A&lt;1&gt;</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>10</td>
<td></td>
<td>A&lt;1&gt;</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>00</td>
<td></td>
<td>A&lt;1&gt;</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>10</td>
<td></td>
<td>A&lt;1&gt;</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>01</td>
<td></td>
<td>A&lt;1&gt;</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
<td></td>
<td>A&lt;1&gt;</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>00</td>
<td></td>
<td>A&lt;1&gt;</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>11</td>
<td></td>
<td>A&lt;1&gt;</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>11</td>
<td></td>
<td>A&lt;1&gt;</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td></td>
<td>A&lt;1&gt;</td>
</tr>
</tbody>
</table>

Figure 4-4: Ternary NAND Evaluation

Figure 4-4 shows an example from the logic simulation domain. The \(<n>\) notation in the figure is used to refer to the \(n\)th bit of a word. The figure shows the truth table for the evaluation of a ternary NAND gate where 00 is an illegal state, 01 is a logic zero, 10 is a logic one, and 11 is a logic X. The MIPS instruction set requires at least four instructions to evaluate this or any other two-input ternary gate through table lookup techniques. The number of cycles is slightly bigger in the case of a cache hit, and considerably more in the case of a cache miss for the load byte (ldb) instruction. However, as the figure shows, a single cycle PFU can easily evaluate this ternary gate. In fact, with 4-input LUTs, any ternary 2-input gate can be evaluated with just one LUT per bit.
Figure 4-5 shows the logic expression for an example from the ESPRESSO benchmark where the table was performing a byte-wide population count function. This function determines the Hamming distance between two product terms as a prelude to a boolean consensus operation. The implementation consists of a set of carry-save adder (CSA) cells which add the number of bits in the input word. The CSA representation is shown for clarity, the actual synthesized implementation merges the CSA cells to minimize the number of 4 input LUTs used in the PFU.

We can also use this table lookup replacement technique to replace jump-tables for switch statements. In most cases, the hardware complexity of jump-tables is very small, concentrating in the lowest bits while the addresses for the highest bits remain at the same value. The logic synthesis package minimizes the jump-table offsets, and the generated function can be placed inside a PFU. A final addition is still needed to combine the offsets produced by the PFU with a base address to produce the final jump address. However,
even this addition operation can be removed if we performed the PFU synthesis at link time, after the final addresses for the switch code segments have been resolved.

4.3 Control Flow Graph Optimizations

Conditional branch instructions perform a simple boolean operation -- whether or not to take the branch, and then they change the program counter appropriately. As such, conditional branch instructions are performing a sparse boolean function. Using PFUs, two techniques, boolean-CFG and jump-CFG optimization, are used to eliminate branch instructions and accelerate application performance.

4.3.1 Boolean-CFG Optimization

The boolean-CFG optimization transforms a portion of a control flow graph into a set of boolean equations which can be minimized and programmed into PFU resources. Presently, we use a greedy algorithm to determine which portion of a large CFG should be transformed.

In order to use the boolean-CFG optimization, a subset of a CFG must:

1. have one and only one start basic block;

2. have one and only one exit basic block;

3. contain only PFU-LOGIC instructions in all blocks except the start and exit basic block;

4. not contain any backedges (loops).
The example in Figure 4-6 illustrates the boolean-CFG optimization. The code example implements a mapping between a sparse range of characters and integers. The CFG for the code is shown on the right side of the figure. The conditional basic blocks, $c_n$, contain the instructions which perform the compare and branch, and the assignment basic blocks, $a_n$, contain the instructions for the final assignment and exit jump. The legend below the CFG graph shows an instance of the instructions generated for these two particular blocks.

The process for the conversion of a CFG to a set of boolean equations proceeds in three basic steps: predicate assignment [27], basic block state transformation, and boolean minimization. Predicates are calculated for each basic block by enumerating all the paths to the starting basic block [27]. This enumeration creates a sum-of-products representation where each product is the set of conditions needed to build a single path. In the example, the block predicates are
\[ BP_{c1} = 1 \quad BP_{c2} = p_1 \quad BP_{c3} = p_1p_2 \]
\[ BP_{a1} = \overline{p_1} \quad BP_{a2} = p_1\overline{p_2} \quad BP_{a3} = p_1p_2\overline{p_3} \quad BP_{a4} = p_1p_2p_3 \]

where \( p_1 \leftarrow (c \Rightarrow b') \), \( p_2 \leftarrow (c \Rightarrow s') \), and \( p_3 \leftarrow (c \Rightarrow w') \) are the predicates. Once we have calculated the basic block predicates, the individual basic blocks are transformed to include the effects of the predicate.

Given a basic block predicate, \( BP \), and an assignment of the form \( OUT = A \ op \ B \) where \( op \) is any of the PFU-LOGIC operations, the boolean transformational rule is

\[
OUT_{new} = (A \ op \ B)(BP)^n + OUT_{old}(\overline{BP})^n
\]

where \( OUT_{old} \) is the value of the output variable before assignment,

\( OUT_{new} \) is the value of the output variable immediately after the assignment

\( (\cdot)^n \) function generates a \( n \)-bit vector by replicating a boolean bit.

This transformation rule is critical because it allows the basic block to be evaluated even when the predicate is false. Thus, the evaluation of all the basic blocks in the CFG will produce exactly the same results as the conditional evaluation which followed the control flow. This key point allows the CFG graph to be converted into a hardware netlist which can be evaluated by a PFU in parallel.

In our example, the transformational rule for the assignment in basic block a2 produces:

\[
n_{new} = (00010000)(p_1\overline{p_2})^n + n_{old}(\overline{p_1p_2})^n
\]

As a last step, the CFG netlist with the transformed equations for the PFU-LOGIC instructions is minimized by a boolean logic minimizer. For the example in Figure 4-6, the + operation is a boolean OR not an integer addition.
result after minimization is shown in Figure 4-7. The output bits that are not shown are
tied to a logic zero. If pipeline branch delay slots are ignored for the graph, the CFG takes
4-8 cycles as opposed to the 1 cycle needed for the PFU implementation.

![PFU Boolean-CFG Example](image)

**Figure 4-7: PFU Boolean-CFG Example**

In general, the performance gain or loss offered by a boolean-CFG is determined by the
probability of branching at each conditional branch and the actual hardware complexity of
the operations. If the software has a high probability of taking the shortest path and the off-
path costs are high, the boolean-CFG optimization will yield poor results. In practice, the
profiling information from a performance analysis program such as pixie is used to
compare the equivalent software cost with the PFU hardware cost to determine the
applicability of this optimization for a particular CFG graph. Control flow graphs that
move information or use only low complexity hardware instructions such as constant shift
and logic operations generally benefit from this optimization.
4.3.2 Jump-CFG Optimization

Control flow graphs that use non-PFU-LOGIC instructions such as loads/stores and that do not have a single exit point cannot benefit from the boolean-CFG optimization. However, in many cases, PFUs can still be used to accelerate performance using a technique called jump-CFG optimization. The jump-CFG optimization attempts to convert a control flow graph into a table lookup, i.e. convert a set of if/else statements into a switch statement. PFU resources are used for the complex conditional evaluation combined with the final table lookup. In order to use the Jump-CFG optimization, a subset of the CFG must:

1. have one and only one start basic block;
2. contain only PFU-LOGIC instructions for the conditional expressions;
3. not contain backedges (loops).

![Figure 4-8: Jump-CFG Generic Example](image-url)
An example of the transformation for a jump-CFG optimization is shown in Figure 4-8. In the figure, the $e_n$ instructions are the expressions for the conditional branch instructions, and $b_n$ are the remaining instructions. The intent of jump-CFG optimization is to accelerate performance by centralizing all branching to a single dynamic branch. In Figure 4-8, the evaluation of the conditions is moved up to the start basic block, and all paths to the leaves are enumerated. Table 4-3 shows the 8 possible values for the conditions and the resulting code sequences. As the table and the figure show, only 4 of the 8 combinations ($A_1$, $A_2$, $A_3$, $A_4$) produce unique code streams. We use PFUs to reduce the three condition evaluations to a single evaluation with pointers to the four offset unique code streams. Notice that upon exit, the four code streams can jump to any location and are not required to reconverge like the boolean-CFG optimization. Thus, the Jump-CFG optimization can be applied to a wider class of control flow graphs.

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>Code Sequence</th>
<th>Unique Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>B1, B2, B4</td>
<td>A1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>B1, B2, B4</td>
<td>A1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>B1, B2, B5</td>
<td>A2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>B1, B2, B5</td>
<td>A2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>B1, B3, B6</td>
<td>A3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>B1, B3, B7</td>
<td>A4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>B1, B3, B6</td>
<td>A3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>B1, B3, B7</td>
<td>A4</td>
</tr>
</tbody>
</table>

Figure 4-8 also shows the possible negative ramifications of a jump-CFG optimization. First, there is an increase in the code size which can degrade instruction cache performance. This loss in instruction cache performance should not be overemphasized.
because the code is linearized, so spatial locality is increased. In fact, Mueller [47] has
shown that avoiding unconditional jumps by code replication actually improves cache
performance. Second, this technique forces a premature evaluation of all conditional
expressions in the CFG graph. This premature evaluation can degrade performance in
CFGs where the shortest path is executed with the highest probability. Though this is
tempered by the evaluation of the conditional expressions by a PFU. Just as in the
boolean-CFG optimization, we use branch probability data from a performance analysis
tool to determine the conditions under which we should employ the jump-CFG
optimization.

![Diagram](image)

Figure 4-9: Jump-CFG Example

Figure 4-9 shows an example of jump-CFG optimization from the massive_counts routine
in ESPRESSO benchmark. The c_n basic blocks contain the conditional expressions, and
the a_n basic blocks contain the array increments. The legend on the right of the figure
shows the instructions generated for this code assuming the array address, \texttt{A}, could be allocated into a register. Figure 4-10 shows the resulting structure after jump-CFG optimization. The three conditional checks have been replaced by a single dynamic jump. In this example, the PFU contains the convergence of the conditional evaluations (0x7 and \texttt{val}) combined with a lookup for the address. Unlike the simple conditional evaluation in this example, most cases require more complex bit insert and compare operations.

![Figure 4-10: Jump-CFG Result](image)

Assuming that the addresses for the generated code segments are known (link-time optimization), the conditional evaluation savings alone justify the use of the jump-CFG optimization. Ignoring all branch delay instructions, the original CFG takes six instructions for the best path through the original CFG, but only three instructions in the optimized graph. The most expensive path through the original CFG takes 15 instructions, and 12 instructions in the optimized graph. If branch delays are included, the original graph
encounters three conditional branches as compared with one dynamic jump for the optimized graph. The non-control instructions in each of the component basic blocks are executed in exactly the same manner as the original code. Finally, it should be noted that the premature evaluation of the conditional expressions in the jump-CFG optimization cannot cause exceptions because all the operations are PFU-LOGIC.

4.4 Loop Optimizations

For most software applications, a significant proportion of the overall CPU time is spent in loops. Loop unrolling is a powerful technique used by most modern compilers to accelerate performance by reducing loop overhead [1]. Loop unrolling also offers a greater opportunity for the invocation of the basic block and CFG graph optimizations presented in the previous sections. For example, consider the find-first-bit example in Figure 4-11 that operates on a non-zero number. The left side of the figure shows a "while" loop that counts the number of zeros before a one is reached. The right side shows the pseudo code with the loop unrolled by one iteration.

The width analysis algorithm described in Section 4.1 would declare the loop in the figure as a fixed length loop which can have a maximum of 32 iterations for a 32-bit machine (shift maximum). In this case, we can convert all the variables in the loop to PFU-LOGIC instructions, i.e. the variable $w$ can be limited to 5 bits. If the loop unrolling algorithm fully unrolled this "while" loop, the boolean-CFG optimization can transform this loop into one combinational PFU evaluation.
Given the unrolling shown in the figure, the multiple exit points do not allow the use of the boolean-CFG optimization; however, jump-CFG optimization can be effectively used to accelerate performance. Since this is a loop, only one path through the code is heavily exercised, and the code duplicated for the infrequently exercised paths is not efficiently utilized.

Figure 4-12 offers another transformation for this loop -- the PFU-loop optimization. Like the jump-CFG optimization, the condition expressions are moved to the top of the CFG and evaluated by a PFU. However, unlike the jump-CFG optimization, a jump table is not generated. Instead, a determination is made whether the exit condition is satisfied. If the exit condition is not satisfied, the code through the body of the loop is executed. If the conditions for exit exist, a local exit point is created, and one final iteration through the original code is performed. With the PFU-loop optimization, only one level of code duplication is needed, and the most frequently taken path can still be optimized free of exit
conditionals. In addition, the conditional expression evaluation performed by a PFU is slightly less complex because jump table offsets are no longer needed.

```
w=0;
L: if (((i&0x1)! =0)||( (i>>1)&0x1)! = 0)) l_exit;
i = i>>2;
w = w+2;
go to L;
l_exit:
    if ((i&0x1)==0) exit;
i=i>>1;
w++;
exit:
```

**Figure 4-12 PFU-Loop Optimization**

When loops walk over a bit field and the number of iterations in the loop are large, the previous optimization is very successful, however in most cases, loops walk over memory -- i.e. load/store instructions are part of the conditional expression. We can extend the PFU-loop optimization to include load instructions in the conditional expressions, but including these instructions introduces the following issues:

1. Premature (speculative) execution of load instructions may introduce memory management exceptions;

2. Movement of loads past stores may violate program semantics unless memory is disambiguated;
3. Merged short loads/stores may introduce memory alignment exceptions.

Many modern code scheduling techniques [28] use speculative code motion to maximize performance for superscalar and VLIW machines. Various exception handling schemes have been proposed to support speculative code motion, and PFUs can be used in conjunction with these techniques to increase performance. However, with some knowledge of the initial alignment of array variables and by using conventional exception handling schemes, PFUs can provide considerable performance gain for applications which walk over short memory arrays.

Figure 4-13 shows an example of the PFU-loop optimization for the `strlen` operation. The left side of the figure contains the original MIPS code, and the right side contains the transformed code after loop unrolling and PFU-Loop optimization. All the variables in the figure refer to registers. The amount of loop unrolling is determined by the word size of the machine (four times for a 32-bit, eight times for 64-bit machine). If the initial address for A is guaranteed to be word aligned:

1. 4 `lws` can be merged into a single `lw`;
2. 4 `lw`s can be merged into one addition;
3. `lw` cannot create an alignment exception because the initial address was aligned;
4. `lw` cannot create a speculative memory management exception because the word access will not cross a page boundary.

The key feature of this optimization is the ability of the PFU to perform parallel byte compares to zero. For long strings, the new loop can perform a string compare over
2.5-times faster than the old code for a 32-bit machine and over 5-times faster for a 64-bit machine.

![Figure 4-13: String Length Example](image)

In general, the alignment characteristics of the start of a software array can be determined at compile time. The compiler can force all allocated memory to be aligned so that only calculated addresses are potentially unaligned. If a loop makes multiple array references and the compiler cannot determine whether they are aligned, the PFU-loop optimization cannot be employed, and the old byte-oriented loop must be used. However, if only one array reference is made in a potentially unaligned loop, the compiler can use an additional initial byte loop to align references for the main loop.
4.5 Library Routines

All the optimizations provided in the previous sections use general-purpose compilation techniques to transform software code into hardware constructs. The effectiveness of these techniques is limited by the structural description of the code. Different implementations of the same algorithm can result in vastly different hardware structures. However, if the logical intent of the software function is known, we can use manual hardware design techniques to maximize performance.

The logical intent of runtime library routines is well understood, and we could certainly optimize these routines in a manner that is transparent to user applications. In the C runtime library, character manipulation routines found in `ctype.h` and `string.h` benefit from PFU resources, and even some of the string-to-number conversion routines found in `stdio.h` and `stdlib.h` can use a PFU for fast byte multiplication. In addition to string and character routines, the software implementations of integer division can use a PFU for fast normalization. Of course, if multi-cycle PFUs were allowed, most of the routines in `math.h` could also be offered in hardware form (See Appendix C). Thus, applications which were dependent on a particular mathematical function could use a PFU version of that function to increase performance.

4.6 Summary

This chapter has presented the compilation techniques which are used by a PRISC compiler to effectively utilize PFU resources. Before starting the extraction process, the
PRISC compiler calculates the hardware costs associated with each instruction using a function width analysis algorithm. Using the hardware cost information, several optimizations are employed to reduce the time taken by both control and data operations. For data operations, basic block optimizations such as BB-Expression and BB-Table-Lookup reduce multiple instructions to a single PFU evaluation. Control operations are optimized by using control flow and loop optimizations such as Boolean-CFG, Jump-CFG, and PFU-Loop. These optimizations transform control flow graphs into a form which can be evaluated more optimally using PFU resources. All the optimizations are applied in a methodical manner inside a PRISC compiler.
5. Hardware Synthesis Techniques

After the hardware extraction process, the extracted hardware functions must be synthesized to the available resources on the PFU. The compilation steps generate three forms of input for the hardware synthesis process.

1. Truth Tables;
2. Straight-Line PFU-LOGIC instructions;
3. Control Flow Graph with PFU-LOGIC instructions.

The BB-Table-Lookup and Jump-CFG optimizations generate truth tables. The BB-Expression and PFU-Loop optimizations generate a functional representation consisting of a straight-line series of PFU-LOGIC instructions, and the Boolean-CFG optimization generates a Control Flow Graph (CFG) with PFU-LOGIC instructions. However, the standard synthesis techniques do not manipulate PFU-LOGIC instructions or CFG graphs. Thus, our synthesis process contains two transformation steps, PFU-LOGIC conversion and CFG graph conversion, which transform the input generated from the PRISC compiler to a boolean representation. The overall synthesis process proceeds through the following steps:

1. PFU-LOGIC instruction conversion;
2. CFG conversion;
3. Logic minimization;
4. LUT minimization;
5. LUT placement and routing.

5.1 PFU-LOGIC Instruction conversion

The instruction conversion step transforms the vectored PFU-LOGIC instructions into a scalar boolean representation which can be manipulated by the later synthesis steps. A simple template matching process drives this conversion. The full bit-width boolean representation of every PFU-LOGIC instruction is generated once and stored as a template for future use. When the PRISC compiler extracts a PFU-LOGIC instruction, the instruction conversion step substitutes the scalar boolean template for the vectored instruction. In addition, the function bit-width information associated with the instruction is used to assign constants to the inputs of the boolean network.

![Figure 5-1: Instruction Conversion](image)

Figure 5-1 illustrates this process with an AND instruction where the width analysis algorithms have determined that only the lower two bits are at a X value. The AND instruction template for a 32-bit machine is a logic expression containing 32 AND gates. The non-X constant values are applied to the inputs of the boolean network. This boolean network forms the initial structural network input in the logic minimizer. The details of the logic minimizer will be explained in Section 5.3, but it is important to note that the
performance of the logic minimizer is highly dependent on the initial input logic network. As the figure indicates, the template generation process is straightforward for simple logic instructions, but the selection of template boolean networks for more complex instructions such as ADD dramatically affects the final results. For these instructions, there are many alternatives for the mapping to logic expressions, and it is important to pick a mapping that results in a high performance implementation. In our system, we pick a hierarchical Carry Look Ahead (CLA) scheme based on 8-bit CLA cells.

5.2 CFG Conversion

Figure 5-2: CFG Conversion
After the PFU-LOGIC instructions have been converted to a boolean representation, control flow graphs must be converted to a boolean representation. The input generated by the Boolean-CFG optimization is a CFG where the predicate information has been folded into the basic blocks using the boolean transformational rule presented in Section 4.3.1. Given this information, the CFG conversion process uses the CFG graph to generate the boolean representation for the whole graph. Figure 5-2 shows the input to the CFG conversion process for the example in Figure 4-6.

As the figure shows, none of the conditional basic blocks (c1..c3) manipulate the state variable n, so there is no transformation between the inputs and outputs ($N_{old}$ is equal to $N_{new}$). The assignment basic blocks (a1..a4) modify this variable, so the boolean transformation rule introduces a multiplexer between the input and output. The control for the multiplexer is the block predicate $BP_{an}$ for each assignment basic block. To complete the process of transformation to a boolean netlist, splits and joins in the CFG must have transformations to the boolean network. Splits are simply modeled as fanout operations in the boolean netlist, but joins need a more complex transformation. For a join, at most one of the paths is active, so a join is modeled as a decoded multiplexer:

$$N_{new} = N1_{old}(BP_1)^n + N2_{old}(BP_2)^n \ldots.$$

Figure 5-3 shows the structural network for the example in Figure 5-2.
5.3 Logic Minimization

The logic minimization step accepts two forms of input -- a truth table directly from the compiler or a functional representation from the conversion step. Both the output from the conversion steps and the truth table representation are unoptimized boolean networks. A logic minimizer, MIS [13], is used to reduce the size of initial logic network. The optimization criteria used by MIS is the number of literals in the logic expression.

Before starting the logic minimization procedure, constants are propagated in the logic network. In the case of PFUs, constant propagation significantly reduces the size of the logic network. For example, shifts by constant amounts are reduced to wires. After constant propagation, the MIS logic minimization procedure uses a combination of algebraic and logic reduction techniques to reduce the number of literals in the network.

Algebraic reduction techniques are used because pure boolean minimization techniques are only practical for small networks. MIS uses algebraic techniques to find common subexpressions in a boolean network, and boolean minimization techniques are applied
only for the subexpressions. The common subexpressions are created by forming partial factorizations of each function and generating a list of 'kernels', essential subexpressions from which commonality between two or more functions can always be discovered.

![Figure 5-4: Logic Minimization Example](image)

After algebraic reformulation, boolean minimization algorithms are used to simplify the subexpressions. In general, after the introduction of intermediate variables, many combinations of the inputs to a subexpression are logically impossible. For example, if \( v \) is an intermediate variable and \( f \) is the function which generates \( v \), then \( \text{xor}(f,v) \) are impossible conditions, and are used as don't cares for the minimization of the subexpression. Classical logic minimization procedures such as Quine-McCluskey [45] can be used to find the minimal 2-level network which represents this subexpression. However, exact minimization methods are too expensive, so MIS uses an iterative approximate algorithm, ESPRESSO [32]. Quine-McCluskey generates all the primes for a sum-of-products expression, and exhaustively solves the prime covering problem.
ESPRESSO uses an exploration/search approach where cubes are iteratively expanded to find a minimal representation.

The global algorithm in MIS uses a combination of algebraic subexpression extraction, algebraic subexpression collapse, and boolean simplification steps to create a minimal multi-level network. In practice, this system provides excellent results for logic networks with limited cross-bit expressions. However, if cross-bit expressions such as an add or variable shift are involved, the algebraic optimization step in MIS has difficulty picking the optimal subexpressions. In addition, the compile time for these cross-bit expressions becomes prohibitively expensive. However, a scheme which provides MIS with highly optimal initial implementations of the component functions, and limits the collapse iterations inside MIS gives reasonable synthesis results with outstanding compile time performance. Figure 5-4 (same as Figure 4-7) shows the results of Figure 5-3 after logic minimization by MIS. This function was small enough for a full collapse under logic minimization. Thus, MIS was able to discover that $N_{old}$ is never used to generate the output function for $N_{new}$. All the other output bits are tied to zero.

5.4 LUT Optimization

Once the number of literals has been minimized in the PFU logic expression, the logic network must be decomposed into a network of LUTs. Each LUT implements any function of $n$ variables (4 in our case). Several algorithms [17,18,19] have been proposed to solve this problem. In our work, we use the MIS-PGA mapper [18] from the UC Berkeley. The LUT mapper attempts to solve a covering problem where the whole logic
network must be covered by LUTs. The number of LUTs used and the maximum depth are the optimization criteria. This particular mapper tries to reduce the number of LUTs by taking advantage of reconvergent fanout and the decomposition of symmetric functions. Figure 5-5 shows the results of LUT optimization for our example. The dotted-line box shows that a four input LUT can cover the whole function for the NAND gate. All the NAND gates at the same level are also covered by one LUT each, and three LUTs are required to build the output function for n<3..5>

![Figure 5-5: LUT Optimization](image)

### 5.5 LUT Placement and Routing

Once the LUT network has been generated, it must be placed on the hardware resources of the PFU, and the routing resources needed to connect the LUTs must be programmed. The placement algorithm attempts to minimize interconnect congestion. An optimization based on simulated annealing is used to derive the placement of the LUTs [20]. Simulated
annealing is an optimization methodology based on an analogy to the annealing process in statistical mechanics. In this process, a system is placed in a high energy state, and slowly cooled. The hope is that, similar to molecules in mechanics, the slow cooling will place the system at a minimal energy level. The general simulated annealing algorithm is:

1. Start system in some high energy state.
2. Perturb system slightly (swap 2 atoms).
3. compute $\Delta E$ (energy).
4. if $\Delta E < 0$ then accept this arrangement;
5. else accept this arrangement with $Pr ob(\Delta E, T) = e^{\frac{\Delta E}{kT}}$;
6. if not in equilibrium goto 2;
7. if system not yet frozen, reduce $T$, goto 2;
8. Done.

Simulated annealing is a hill diming algorithm, so it will accept perturbations that are locally non-optimal. This feature of simulated annealing allows it to escape local minima.

To use this algorithm, the move set and cost function used for minimization must be defined.

For the LUT placement problem for a layered PFU, the cost function is a function which measures the congestion created by a placement for the interconnect resources, and the move set is swaps of LUTs in the same layer. In general, simulated annealing is a computationally expensive algorithm. But since a PFU consists of a small number of
elements, our experiments have shown that the computational time taken by the simulated annealing algorithm is small.

The final step is to program the routing resources on the PFU. A maze routing scheme augmented with rip-up and rerouting steps is effective for the relatively small number of nets in a PFU. A maze router connects each wire using a levelized searching technique. A maze router places a grid on the interconnection space, and connects one net at a time using the following algorithm:

1. Start at source.

2. Find all cells reachable at distance one, and label these cells as "active".

3. Find all cells reachable at distance two, these are neighbors of old active cells, and label these as "active" cells.

4. ...

5. Quit when target is found.

As nets are routed, these nets become obstacles for wires routed later in the process. If the target of the net cannot be reached, clashing wires must be removed and rerouted.

5.5 Conclusions

In general, the logic, LUT, placement, and routing algorithms have a worst case performance behavior which cannot be bound by a polynomial function (NP-complete). However, the general-purpose algorithms employed were built to solve large synthesis problems. Our PFU-LOGIC functions are quite small compared with the typical synthesis
problems handled by these algorithms, so in large majority, the existing algorithms can easily synthesize logic for PFUs. In addition, our simplifications to the standard algorithms further reduce the compile time. In all our experiments, the hardware extraction algorithms have taken a larger proportion of computational time than the hardware synthesis algorithms.
6. Performance Modeling and Results

A full analysis of our PRISC architecture ideas requires augmentation of a general-purpose language compiler and the detailed design of the hardware needed for a PRISC-1 computer. However, before investing heavily in these two activities, we have built a performance modeling environment to estimate the potential benefit of a PRISC-1 computer. To avoid biasing our results by choosing applications especially suited for PRISC computers, we have used the SPECint92 benchmark suite as an unbiased judge of our PRISC techniques.

6.1 Performance Model

Figure 6-1 shows our performance modeling environment. In this environment, a MIPS binary file of a user application is instrumented with performance counters by pixie. This augmented binary file is executed, and the basic block count information is stored in a .Counts file. This file can be correlated with the binary using the .Addrs file produced by pixie. Our performance analysis tool, pxsim, uses these three files to estimate the performance gain for a PRISC-1 computer. Pxsim is an extension of the xsim simulator from Stanford University [2].

Like all performance modeling strategies, this particular strategy has some shortcomings. Since pxsim is based on basic block performance data, ordered dependent effects such as caching of the memory hierarchy and swapping for the expfu exception handler are ignored. Not modeling memory system effects causes the performance results to be
pessimistic for the optimizations which reduce usage of memory (BB-Table-Lookup optimization), and optimistic for optimizations which increase usage of memory (Jump-CFG optimization). Although, as mentioned in Chapter 4, the linearization of the instruction stream may actually improve cache performance due to better spatial locality. Exception handling for the expfu instruction is modeled by a one-time 1000 cycle penalty. However, due to the nature of basic block performance modeling, swapping between several PFUs is not modeled. This swapping behavior induces multiple exceptions in a one PFU model. We manually verified that the three highest performing applications (ESPRESSO, EQNTOTT, and COMPRESS) did not invoke this behavior.

![Performance Modeling Diagram]

Because object code is used as input, pxsim does not have full access to the lifetime of variables in main memory. For example, temporary variables are optimized away in a PFU,
but if these variables are allocated to a location in main memory due to register pressure, 
pxsim cannot optimize them away, again leading to pessimistic performance results.

Presently, pxsim does not use any type information that may be available in an object file. This loss of information does not allow for full optimization during the width analysis step. For example, any usage of an enumerated type can substantially reduce the function width of subsequent operations. Also, pxsim does not perform any high-level optimization steps such as procedure inlining. Nor does it perform any substantial code motion to enable the usage of the various hardware transformations. Rather, the existing generated code is used as a static base, and the applicability of the hardware transformational techniques is tested. This lack of active code motion and usage of type information also leads to pessimistic performance results.

Finally, pxsim performs hardware synthesis up to the LUT optimization step, but does not perform LUT placement and routing. Instead, if the maximum depth of the gate level network is less than 6 levels, the PFU function is allowed to replace the software code. This estimate of PFU programmability can lead to optimistic results for 6-level networks which could not fit in a PFU, and pessimistic results for 7 or 8 level networks which could have fit into a PFU. Of all the sources of inaccuracy, the lack of active code motion and the loss of information between the source and object have the most impact on the performance results, so the results of our performance model are pessimistic.
6.2 Performance Results

All our experiments were performed on a DECstation 5000/240 using the MIPS C compiler (V2.10). All the SPEC benchmarks were compiled using the Makefile scripts for the dec_risc machine description. Table 6-1 shows the number of times each of the hardware extraction optimizations was invoked in the SPECint92 benchmarks. The trigger used for invocation is that any individual optimization must increase performance by at least 0.1%. The benchmarks shown are ESPRESSO (EXP), EQNTOTT (EQN), SC, LISP, COMPRESS (CPS), and GCC.

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>EXP</th>
<th>EQN</th>
<th>SC</th>
<th>LISP</th>
<th>CPS</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB_Expression</td>
<td>48</td>
<td>0</td>
<td>12</td>
<td>4</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>BB_Table_Look</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Boolean-CFG</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>Jump-CFG</td>
<td>47</td>
<td>0</td>
<td>35</td>
<td>0</td>
<td>10</td>
<td>103</td>
</tr>
<tr>
<td>PFU-Loop</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Library</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>Total</td>
<td>95</td>
<td>4</td>
<td>47</td>
<td>4</td>
<td>19</td>
<td>147</td>
</tr>
</tbody>
</table>

The BB_Table_Look optimization was never invoked on any of these benchmarks because constant arrays are not declared as constant in application source code. In addition, the MIPS C compiler does not retain the read-only nature of constant information in the object file, so augmenting the source code with constant qualifiers has no effect on the performance results. The library optimization is also invoked infrequently for each of these benchmarks; the C runtime routines do not consume a significant proportion of CPU time.
for most of these benchmarks. Finally, it should be noted that the number of PFUs generated by the extraction algorithms does not stress the 2047 logic-PFU limit for the \texttt{expfu} instruction.

![Figure 6-2: SPECint Results](image)

The histogram in Figure 6-2 shows the performance gain obtained on each of the SPECint92 applications when we include a single PFU resource in the MIPS processor. We calculate the performance gain by dividing the number of instructions taken before PFU transformations by the number taken after PFU transformations. The significant speedup in the EQNTOTT benchmark is caused by the PFU-loop optimization of the \texttt{cmppt} routine which is a bottleneck for this application.

Figure 6-3 shows the MIPS code for this routine, and the compiler generated transformation of the original code to use PFU resources. This routine performs an operation similar to \texttt{strncpy} from the C runtime library, but in the case of \texttt{cmppt} the
basic datatype is a 16-bit integer. In addition, before comparison, both 16-bit integers are compared with a constant, and conditionally assigned a new constant value. The original loop takes 24-28 instructions to complete two iterations as opposed to the 11 instructions needed for the transformed loop. Figure 6-4 shows the functional logic of the PFU needed for the **cmppt** routine (a0 and a1 are the upper and lower halves of variable A).

![Figure 6-3: CMPPT Example](image)

The ESPRESSO and COMPRESS benchmark contained many instances of BB-Expression optimizations. Figure 4-3 has already shown some of these instances for the ESPRESSO example. The SC, LISp and GCC benchmarks primarily used the Jump_CFG benchmark. The relatively poor performance of the LISp benchmark was caused by the short procedure calls in the basic LISp interpreter loop.
In addition to the experiments on the SPECint92 benchmarks, we have tested our PRISC techniques on other applications, and found similar results. For example, a single PFU increases the performance of the BDD [48] boolean manipulation package by 20% on a test that builds BDDs for a suite of random functions. The BDD package heavily uses hash tables, and PFUs provide a fast method of hash function evaluation. Also, the performance of the MPEG [49] video decompression application is increased by 18% with the use of a PFU. The MPEG decompression algorithm manipulates bit-level data, so PFU resources are useful for building specialized bit manipulation instructions.

6.3 Summary

In this chapter, we have shown a single PFU resources provides an average of 22% performance improvement on the SPECint92 benchmarks. These benchmarks were chosen because they provide an unbiased judge of our PRISC techniques. The performance gain on the SPECint92 benchmarks is confirmed with our tests on other applications such as
the BDD and MPEG packages. This performance gain from a single PFU is significant in comparison with other architectural alternatives. For example, consider the addition of more on-chip cache memory. Many of today's RISC CPUs contain at least 8 Kbyte on-chip caches [5]. Increasing the size of this cache to 16 Kbytes only decreases the miss rate by 2.5% [21, P.507]. Under optimistic conditions, a 2.5% miss rate improvement provide a maximum performance gain of 10%, but at a hardware cost which is four times that of a PFU.
7. Software Acceleration Techniques

Conventional RISC computers provide selected functions that operate over the integer, floating point, and boolean data types. Table 7-1 shows the typical population of functional units that are provided for each of these data types. In contrast to conventional functional units, a PFU offers a large set of functions that are defined as follows:

\[ \text{PFU} : B^n \times B^n \rightarrow B^n \]

where \( B = \{0, 1\} \) and \( n \) is the word size of the machine.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Functional Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>adder, multiplier, shifter, divider</td>
</tr>
<tr>
<td>floating point</td>
<td>adder, multiplier, divider</td>
</tr>
<tr>
<td>boolean</td>
<td>and, or, not, nor...</td>
</tr>
</tbody>
</table>

As the results from Chapter 6 have shown, the peep-hole optimizations discussed in Chapter 4 can utilize PFU resources to accelerate the performance of general-purpose programs by an average of 22%. Because of their peep-hole nature, these compiler optimizations are limited by the structure of the input source code. Active manipulation of the source code leads to significantly higher local performance gains (250%-500%) for general abstract data types such as short-set vectors, hash tables, and finite state machines. This chapter discusses the use of PFU resources for each of these data types, and provides results for end-user applications which use these data types.
7.1 Short-Set Vectors

Set vectors are defined as arrays whose elements are constrained to a limited range of values. We shall refer to the set that contains these values as the ssv set. The cardinality of the ssv set determines the minimum number of bits needed to uniquely represent all of its constituent elements. We define short-set vectors as set vectors whose elements can be represented with less than half the number of bits in a machine word. Like other data types, we can define functions over the ssv set that can be used to manipulate the short-set vector abstract data type. These functions are defined as follows:

\[ F_{ssv} : B^m \times B^m \rightarrow B^m \]

where \( m \leq \frac{n}{2} \). Since the bit-width of the inputs of these functions are guaranteed to be less than half the word size of the machine, the hardware complexity of \( F_{ssv} \) functions has been greatly reduced in comparison with the full bit-width operations. Thus, a much greater percentage of \( F_{ssv} \) functions can be implemented in PFUs as compared with the full bit-width functions.

7.1.1 SSV Set Functions

The most commonly used short-set vector is the character string datatype. The ssv set for this data type consists of the ASCII characters, and eight bits are used to encode each element in this set (\( m=8 \)). Many \( F_{ssv} \) operations over the ASCII character set can be fruitfully implemented with PFU resources. For example, consider the \texttt{toupper} function from the C runtime library which converts a lower case letter to an upper case letter. Figure 7-1 shows the MIPS instructions needed to check the bounds and perform a
conversion addition. The <n> notation is used to denote bit positions in a word, and the figure only shows the logic expression for bit five because the other bits are directly connected to their corresponding input bits. Ignoring any pipeline stalls, this function requires five instructions in the base RISC instruction set, but it can be implemented by a PFU (expfu out, I, 0, 1) that executes in a single cycle -- a local performance gain of 500%.

Figure 7-1: TOUPPER Example

7.1.2 Packed SSV Set Functions

In addition to implementing the individual F_{sv} functions, PFUs can be used to perform multiple F_{sv} operations in parallel by packing multiple elements of a short-set vector into a machine word. The ability to pack information into machine words is important for two reasons. First, the packed information is efficiently represented in main memory, so caching strategies are more successful. Second, the ability to view the short-set vector as a word array enables the use of word load/store instructions which can efficiently move
information between main memory and the CPU register set. On conventional RISC computers, many functions on the packed SSV sets require multiple instructions. However, on PRISC, a PFU can perform the functional transformations needed to manipulate the short-set vector abstraction in a single cycle.

This packed vector strategy was used in the 2-level logic minimizer ESPRESSO [32]. In ESPRESSO, a short-set vector is used to represent the product terms in a sum-of-products logic expression. A ternary algebra, requiring two bits to represent, is used to model the three states possible for each boolean variable: variable is present and asserted (01), variable is present and deasserted (10), and variable is absent (11) ($m=2$ and $ssv=\{1,2,3\}$).

**Figure 7-2: ESPRESSO Example**

For example, consider the disjoint cube function which is used to determine if the conditions needed for a boolean consensus operation are satisfied. As the left side of
Figure 7-2 shows, ESPRESSO uses four MIPS instructions to perform this operation in parallel on a RISC computer. However, as the right side of the figure shows, the logic function to implement this operation is quite simple. The even numbered output bits are tied to a logic zero, and the odd numbered output bits are generated using only three logic gates. So, a PFU reduces the four RISC instructions to a single PFU instruction (expfu out, A,B,1) for a local performance gain of 400%.

7.1.3 Packed SSV Test Functions

![Diagram](Figure 7-3: Zero Byte PFU)

The packed vector strategy is successful if there are no sequential dependencies between the elements of the short-set vector. When dependencies occur, the parallel operations cannot be evaluated until all the sequential dependencies are resolved. Strings that use a termination character exhibit this behavior. In the C language, the end of the string is
indicated by a zero, so no operations beyond the zero character are valid. However, PFUs can provide a test function to determine whether the sequential dependency exists.

For example, consider the `strlen` operation from the C runtime library. The PFU needed to catch the end of string for the `strlen` function is shown in Figure 7-3. The input to the PFU is a 32-bit word, `c`, and the logical network determines if any of the bytes in the word are zero. The code transformation for the `strlen` operation was presented in Figure 4-13. With this PFU, a word loop can be written which traverses through the string in multiple byte increments. For long strings, this method leads to a local performance improvement of 250% for 32-bit machines, and 500% for 64-bit machines.

7.2 Hash Tables

The hash table abstract data type is a popular method for organizing data. Hash tables directly reference data in a table by performing transformations on a key. Since the table is randomly accessed, the search operations can be performed in constant time. Transformations are needed because generally the key is not uniformly distributed, so a mapping is needed to translate non-uniform input to an uniform output. These transformations are performed by a hashing function that takes a key, `K`, and generates an address between `0..M` where `M` is the table size.

The hashing function determines the overall performance of hash tables for two reasons. First, the quality of the hashing function determines the number of collisions in a hash table. Collisions require extra processing and can lead to asymptotically worst behavior.
For example, if a chaining strategy is used for collisions and the hash function is especially poor, the asymptotic behavior of searches degrades from $O(1)$ to $O(n)$. Second, the evaluation time of the hashing function is critical because this function is evaluated on every interaction with the hash table. This evaluation time is especially critical for open hashing schemes [34] which evaluate the hashing function even during collision processing. Lum, Yuen, and Dodd [35] report that polynomial division and integer division hashing methods provide the best results for the minimization of collisions. PFU resources can be utilized to accelerate the evaluation time for both of these hashing methods.

### 7.2.1 Polynomial Division Hashing

Polynomial division hashing is commonly used in hardware testing applications to create a hash signature of a large sequence of data [40]. For software hashing purposes, each digit of the key is considered to be a polynomial coefficient, and the address is obtained by dividing the key polynomial $k(x)$ by another polynomial $g(x)$. The coefficients of the remainder polynomial are used as the address into a hash table. The degree of $g(x)$ is determined by the size of the table, $M$, and Linear Feedback Shift Registers (LFSRs) are used to perform the polynomial division. Figure 7-4 shows the hardware and MIPS software implementation for LFSR structures [40,42]. In hardware implementation, LFSRs consist of registers (D flip-flops) in combination with modulo-2 adders (XOR gates). The $c_n$ variables specify the feedback taps from the output and determine the divisor polynomial $g(x)$. If the key is fixed to zero and $g(x)$ is an irreducible polynomial, the LFSR cycles through all $2^n-1$ states of the shift register except the zero state. The
output sequence has a uniform distribution which can be used for pseudo random number generation or hashing. In fact, LFSRs are often used as pseudo random number generators in hardware testing applications.

![LFSR Diagram](image)

**Figure 7-4: LFSR Example**

For hashing, an initial seed is loaded and the key, $K$, is time multiplexed into the LFSR. For example, to create a hash signature for a string, the LFSR is evaluated for every byte of the string, and the state left in the LFSR is used to address the hash table. The uniformity of the hashing function is not very sensitive to the selection of the divisor polynomial $g(x)$. In the Lum, Yuen, and Dodd study, random selection of the $g(x)$ polynomial with the restriction that neither the highest degree coefficient nor the constant term is zero yielded results similar to the integer division based methods.

In software, a conditional branch, combined with logical and shift instructions, are used to implement the LFSR. The variable $H$ contains the state of the LFSR, $K$ contains the key, $C$ contains the $c_n$ variables, and $F$ contains a mask which mimics the feedback in the hardware. With 50% probability for the conditional branch, the software version requires
an average of five instructions to implement the polynomial division hashing. However, the hardware resources needed by a LFSR are trivial, so a PFU can easily implement a LFSR function \textbf{(expfu H,H,K,1)} which evaluates in one cycle -- leading to a local performance gain of 500%.

### 7.2.2 Integer Division Hashing

The hashing function based on integer division takes a prime number $M$ and computes the modulo operation $K \mod M$. This method has very good collision minimization properties [33,35]. However, it can be expensive to evaluate because integer division is typically a multi-cycle iterative hardware algorithm on modern RISC computers, e.g. integer division has a latency of 76 cycles on the R4000 [16]. In addition, several modern architectures do not provide hardware resources for the integer divide instruction [37], so the operation must be emulated in software.

We suggest another scheme that still uses the division method, but the values for $M$ are constrained to $(2^w-1)$. These values for $M$ are quite interesting because they provide a quality of hash function that is almost as good as the purely prime based method [35]. However, unlike the modulo operation with conventional primes, the modulo operation for these numbers can be performed efficiently using PFU resources by using the following recursive transformation:

$$K \mod (2^w - 1) = ((K/2^w)2^w + K \mod 2^w) \mod (2^w - 1)$$

$$= (K/2^w + K \mod 2^w) \mod (2^w - 1)$$
where $2^w \mod (2^w - 1) = 1$. Each iteration of this transformation reduces the key, $K$, by $w$ bits. Given $w$ and $n$, the word size of the machine, the number of iterations needed to reduce a key is $\frac{n}{w} + 1$. After the final iteration, $K$ is reduced to $0 \leq K < 2^w$. In this range,

$$0 \leq K < 2^w - 1 \rightarrow K \mod (2^w - 1) = K \mod 2^w$$

$$K = 2^w - 1 \rightarrow K \mod (2^w - 1) = 0$$

\[ \begin{align*}
&\text{srl } T1,K,w \\
&\text{and } T2,K,M \\
&\text{addu } K,T1,T2 \\
&\text{expfu } K,K,0,1 \\
&\text{bne } K,M,\text{exit} \\
&\text{mov } K,0
\end{align*} \]

\[ \begin{align*}
&\text{bne } K,M,\text{exit} \\
&\text{mov } K,0
\end{align*} \]

Figure 7-5: Integer Hashing

A division by $2^w$ can be performed by a shift operation and produces a $(n-w)$ bit result. A modulo by $2^w$ can be performed by a masking operation and produces a $w$ bit result. Figure 7-5 shows the MIPS instructions needed to implement this hashing scheme. As the figure shows, each evaluation takes 3 instructions, and if pipeline stalls are ignored, the final reduction is performed by a conditional move in two instructions. For values of $w$ greater than seven, the division operation can be reduced from 76 cycles to 17 ($5*3+2$) cycles using RISC instructions. However, the adder needed for the addition can be as small as $\max(w,n-w)$. Thus, all three operations (shift, mask and add) can be evaluated in a PFU that performs the operations in one cycle, and takes 7 ($5*1+2$) cycles for the
smallest hash table -- a local performance gain of 1080% as compared with the hardware divider and 240% as compared with the RISC instruction set implementation.

7.3 Finite State Machine Evaluation

Finite State Machines (FSMs) are one of the bedrock mathematical models in the field of computer science. As such, this model is extensively used in applications such as high-level language lexical analysis [39] to acceleration techniques for high speed logic simulators [36,38]. In the lexical analysis application, FSMs are used as regular expression recognizers. In the logic simulation application, FSMs concisely record the state of the simulator, so that evaluation routines can be optimized based on known contextual information. In both applications, a finite state machine description is generated once, and evaluated in the kernel of the application. Thus, the evaluation time for FSMs must be minimized to maximize end-user application performance.

The data representation of the FSM has an important impact on evaluation performance. In software, the FSM states are generally represented explicitly in tables or linked lists. Because of their random access properties, high-performance applications use table-based methods. In a table-based method, FSM states are stored in an array that contains the mapping between the current state and the input to the next state. The size of this array is the product of the number of states and number of inputs. In this scheme, the cost of a FSM evaluation is a two-dimensional array lookup (address calculation and memory load).
In contrast to the explicit methods, FSMs can also be represented implicitly using a next state function.

$$NS : B^I \times B^S \rightarrow B^S$$

where $I$ is the number of bits needed to represent the inputs into the FSM and $S$ is the number of bits needed to represent the FSM states. If the states are fully encoded, the number of bits needed is $S = \lceil \log_2(FSM\ States) \rceil$. This representation is used to implement FSMs in hardware, but cannot be efficiently evaluated on conventional RISC computers. However on PRISC, the next state function can be loaded into the PFU resources and be evaluated in a single cycle.

For example, Figure 7-6 shows the FSM representation for a 3-state zero-delay inverter in an event-driven logic simulator [41]. The FSM consists of three states and six inputs. States $S_{xx}$, $S_{01}$, and $S_{10}$ represent the stable conditions for the inverter gate. The input consists of the six possible input transitions in a ternary logic system. Evaluation of the
FSM requires concatenation of the current state and input fields, array address calculation, and a byte load instruction. Ignoring cache misses, the table-based evaluation takes a minimum of four cycles. Using a functional representation, the next function for the inverter FSM is:

\[
ns_0 = cs_1 \overline{cs_0} I_2 \overline{I_1} I_0 + \overline{cs_1} cs_0 I_2 \overline{I_1} I_0 + cs_1 \overline{cs_0} I_2 I_1
\]

\[
ns_1 = cs_1 \overline{cs_0} I_4 I_1 I_0 + \overline{cs_1} cs_0 I_2 I_1 + \overline{cs_1} \overline{cs_0} I_2 I_1 I_0
\]

with the current and next state (cs and ns) mapping being \(ERR_{state} = 0, 10_{state} = 1, 01_{state} = 2,\) and \(XX_{state} = 3.\) The mapping for the input, \(I,\) transitions is: \((X \rightarrow 1) = 0, (X \rightarrow 0) = 1, (0 \rightarrow 1) = 2, (0 \rightarrow X) = 3, (1 \rightarrow 0) = 4,\) and \((1 \rightarrow X) = 5.\) In this functional form, the table can be evaluated in one cycle (expfu ns, cs, I, 1) -- a local performance gain of at least 400%.

### 7.4 Results

The utility of any acceleration technique depends upon its impact on end-user applications. The previous sections have shown that PFU resources provide significant local performance gains (250%-500%) for the evaluation of short-set vector functions, hashing functions, and finite state machines. The impact of these local performance gains on an end-user application depends on the importance of the PFU function evaluations in the overall instruction mix. Given the performance model presented in Chapter 6, Figure 7-7 shows our results for four applications from the domain of Computer-Aided Design (CAD) whose kernel computations were changed manually to take advantage of PFU resources.
The first test case, EQNTOTT(EQN), translates a logic equation into a truth table representation [22]. The underlying structure used by the application is a packed ssy set vector (m=16 and ssy={0,1,X}) that represents product terms in the sum-of-products representation for the function. This application spends 95% of the time in the qsort product term comparison routine. This routine compares each element of two set vectors sequentially. As the results from Chapter 6 showed, pxisim was able to use the PFU-loop optimization to achieve a performance gain of 91%. However, the 16 bits are not needed to hold the ssy set for this ssy set vector. Reducing the basic data type to 8 bits increases the overall performance gain to 213%.

![Figure 7-7: CAD Results](image)

The second test case, Sim2ntk (S2K), is part of the CURRIER [43] transistor abstraction tools. This application maintains a large dictionary of node names for a transistor netlist. For large netlists, the performance of this application is dominated by searches on the node
dictionary. Thus, providing PFU resources to manipulate strings leads to an overall performance gain of 186%. The third test case, WLC, is a wirelist comparison program [44] used to verify VLSI layout connectivity. This program uses hashing techniques to color large transistor graphs. As such, the performance of this application is dominated by the hashing calculation, and PFU resources increase overall performance by 52%. Finally, the fourth test case, LSIM, is a logic simulator based on the FSM evaluation techniques [41]. Using FSMs for the evaluation of the basic gates leads to an overall performance gain of 32%. 
8. Conclusions

This thesis has introduced a new style of general-purpose computer, PRISC, which can build application-specific instructions in the context of a RISC environment. PRISC maintains all the demonstrated advantages of RISC, e.g. fixed instruction formats and load/store architecture. However, in addition to offering the conventional RISC instructions, PRISC offers programmable functional units (PFUs) that are used to construct application-specific instructions. With PFUs, PRISC optimizes past the instruction set architecture interface between software and hardware. For this thesis, we have restricted our discussion to a PFU that emulates combinational functions with two inputs and one output. We refer to this implement of PRISC as the PRISC-1 machine.

In addition to introducing the general concept of PRISC, we have:

- completely specified an additional instruction, `expfu`, which is needed to turn a RISC architecture into a PRISC architecture;
- described the microarchitecture changes needed to accommodate a programmable functional unit;
- described compilation techniques that extract hardware from conventional software program; and
- described the hardware synthesis process needed to synthesize the hardware extracted by the compiler.

We have shown that the use of these techniques for the PRISC-1 with a single PFU yields on average performance gain of 22% for the SPECint92 benchmark suite with a hardware investment less than that needed for a 2Kbyte SRAM. The SPECint92 benchmark suite
was chosen as an unbiased judge of our PRISC techniques. The performance gain across all the SPECint92 benchmarks from a single PFU is significant in comparison with other architectural alternatives. For example, utilizing four times the hardware resources of a PFU to increase the primary cache size (from 8Kb to 32Kb) provides less than half the performance gain of a PFU. Beyond compiler-based techniques, we have shown that active source code transformations lead to local performance gains of 250%-500% for the short-set vector, hash table, and finite state machine abstract data types. These local performance gains translate to end-user performance gains of 32%-213% for a set of applications in the domain of computer-aided design.
9. Future Work

This thesis has demonstrated the feasibility of the PRISC style of computing within the PRISC-1 model as an extension to a conventional RISC processor. We would like to extend this analysis to the impact of PRISC techniques on modern processors which exploit instruction level parallelism. PRISC compilation techniques reduce the depth of data dependency graphs, increase the size of basic blocks, and remove branch instructions from the instruction stream -- all side effects that are beneficial for processors that exploit instruction level parallelism. In addition, the availability of multiple PFUs can decrease the penalty associated with PFU programming.

Our work started by developing a simple model for the PFU and developed hardware techniques which utilize this hardware resource. However, the PFU microarchitecture was not optimized for the class of functions that are typically generated by our software extraction techniques. For example, our present results indicate that the lower 8-16 bits are quite important for a PFU implementation. We would like to explore different configurations for a PFU and determine the impact of these designs on overall performance gain. Also, the two-input/one-output PFU model was a severe limitation. In many instances, functions of low hardware complexity could not be programmed into a PFU because of input or output constraints. We would like to explore the performance effects of wide input and output ports in a superscalar setting on the performance of PRISC-1 machines.
Beyond the PRISC-1 model, a new class of multi-cycle PRISC computers offer many challenges in the areas of hardware extraction, PFU microarchitecture design, and ease of fit into a RISC CPU environment. Multi-cycle PFUs offer the ability to execute software loops in hardware. These loops must be extracted from hardware, a faster hardware function must be built, and the remainder of the CPU must be able to communicate with this new PFU. As the PFU-Loop optimization shows, this process is not difficult for loops with only PFU-LOGIC instructions, but loops which contain load/store instructions require special work. However, if successful, substantial performance gains can be achieved by eliminating branch instructions and scheduling past these software loops. Appendix B discusses our initial ideas for multi-cycle PFUs.

Finally, the PRISC techniques presented in this thesis accelerate the computational portion of a von Neumann processor. As long as general-purpose machines separate computation from memory, accelerating the computation will cause the communication to memory to become the performance bottleneck. If computation and memory can be mixed, order of magnitude performance gains are possible. The logic emulation work provides an indication of the possibilities. Unfortunately, most modern programming languages rely heavily on the von Neumann machine model. Thus, a high-level programming language change is required to achieve orders of magnitude gains in performance.
Appendix A: PFU Circuit Design

A Programmable Functional Unit (PFU) is the core resource for PRISC. As Figure A-1 shows, a PFU has four vector inputs (A, B, Paddr, Pdata) and one vector output (OUT).

A PFU must support two distinctive modes of operation -- eval and prog.

![Figure A-1: PFU Interface](image)

The prog mode programs the PFU by using the Paddr and Pdata inputs to define a function:

\[ PFU : A \times B \rightarrow OUT \]

where \( A, B, OUT \in \{0, 1\}^n \) and \( n \) is the word size of the machine. The size of the Paddr and Pdata inputs (\( a \) and \( d \)) is determined by the particular implementation of the PFU. In eval mode, the function constructed during prog mode is evaluated. In a typical mode of operation, a PFU is programmed once and evaluated often. Thus, the latency of the evaluation mode is critical to application performance, but the latency of the programming
mode has a relatively smaller impact on overall application performance. In the PRISC-1, the evaluation of the PFU is constrained to complete within a single cycle.

The design goal for a PFU is to maximize the number of functions offered by the PFU while maintaining the one-cycle timing constraints. The best way to maximize the number of functions is to offer all \( n2^{2n} \) functions possible between A, B and OUT, but clearly, this is impossible for any reasonable values for the word size, \( n \). Thus, the design goal is to maximize the number of "interesting" functions given the timing constraint. A function is "interesting" if it cannot be evaluated efficiently with conventional RISC instructions and the evaluation of the function consumes a significant proportion of an application's execution time. The next section presents the basic building blocks of a PFU, and section A2 discusses how these components are combined to build our first version of a PFU. Section A3 presents the circuit design issues for our initial design, and section A4 offers conclusions.

### A.1 Basic Components

The basic components used to build PFU functions are Look Up Tables (LUTs) in combination with programmable interconnect resources. Figure A-2 shows the functional form of these two resources in a Complementary Metal Oxide Semiconductor (CMOS) technology. The programmable resources consist of a NMOS pass transistor which is open or closed based on the value in the attached memory cell. The LUT resources consist of \( 2^L \) memory cells connected to a \( L \)-input NMOS pass transistor multiplexer. A NMOS transistor is used for both the interconnect and LUT resources because of its low on-value
resistance and minimal area requirements. However, NMOS pass transistors introduce a threshold voltage drop when passing a one-value. This voltage drop introduces a noise margin issue which is handled in the overall design through precharge design techniques [50]. Figure A-2 shows an example of a 3-input LUT. All the memory cells are implemented using fully static SRAM cells to avoid the reliability problems (charge leakage and alpha particles) associated with DRAM cells.

**Figure A-2: LUT and Interconnect Resources**

**A.2 Layered PFUs**

Because PFU must be able to evaluate in a single cycle, all the paths from the inputs to outputs must have predictable delay properties. Thus, we cannot employ the microarchitectures used in popular Field Programmable Gate Array (FPGA) devices [15].
These structures allow for considerable routing freedom, but the interconnect delay cannot be tightly bound. Instead, we have chosen a bipartite structure which alternately uses interconnect and LUT resources.

Figure A-3: PFU Microarchitecture

Figure A-3 shows the basic structure for such a PFU. Since the interconnect and LUT resources are used alternately and the flow of information is unidirectional, the maximum delay can be tightly bound. The number of levels of interconnect and LUT resources is determined by the cycle time of the PRISC machine. We shall refer to this bipartite structure as a layered-PFU.

We can get a sense for the size of a layered PFU by calculating the number of transistors required for its implementation. The number of transistors is given by:

\[ T = LW(M + R2^I) + LWHR(I + 1) + WHR \]

where:
- $L$ is the number of levels of LUT logic;
- $W$ is the word size of the machine;
- $M$ is the number of transistors per LUT;
- $I$ is the number of inputs per LUT;
- $H$ is the number of horizontal interconnect lines per level;
- $R$ is the number of transistors needed for a memory cell.

The first term in the expression calculates the cost of the LUTs, and the remaining two terms calculate the cost of the interconnect resources. The number of transistors needed for a programmable memory cell, $R$, is determined by electrical and reliability considerations -- 6 transistors cells are common for a standard CMOS process. The other parameters are under the full control of the PFU circuit designer. The number of transistors needed to implement a LUT, $M$, is determined by the number of inputs, $I$. As Figure A-2 shows, at least $2^{I+1} - 2$ transistors are needed to build a LUT mux with $I$ inputs. Three-input or four-input LUTs provide the right level of granularity in comparison with hardware costs. Modern high speed RISC processor implementations tolerate at least 12-15 levels of 2-input logic gates within a single cycle. Thus, three levels of interconnect and LUT resources should also fit into a single cycle. Given a 64-bit machine, 4-input LUTs, and typical values for the other parameters ($L=3$, $W=64$, $M=30$, $I=6$, $R=6$), a single PFU requires 61,056 transistors. In comparison, a 2 kilobyte SRAM requires approximately 100,000 transistors. Of course, these numbers do not include transistors added for timing and noise margin reasons, but even if this estimate was off by 100%, PFU resources still would not consume a significant part of the real estate resources found in modern CPUs.
A.3 PFU Example Circuit Design

![Layered PFU Critical Path](image)

Figure A-4: Layered PFU Critical Path

To transform the logical/functional description of a layered PFU in Figure A-3 to a working circuit, we must manage the timing budget and address the noise margin problems associated with the NMOS pass transistors. Figure A-4 shows the critical path through a layered PFU. For each interconnection and LUT pair, the critical path flows through the interconnect resources to the slowest input for the multiplexer. The slowest input for a multiplexer is the input which triggers the discharge of all the intermediate nodal capacitances. Critical to the timing of this path are the capacitances associated with the interconnect resources. This capacitance is determined by the number of transistors connected to each interconnect line. Particularly important to this capacitance is overlap capacitance of the off-programmable devices in the CMOS process.

We address the noise margin problem and minimize the time in the critical path by the use of precharge and cascode design techniques [50]. Figure A-5 shows the basic structure for an interconnection and LUT cell. All the interconnect and multiplexer structures are
duplicated and dual-rail encoding is used to transmit boolean information. This encoding has three valid states: precharge (11), resolved zero (01), and resolved one (10).

![Interconnect Transistor and LUT Stage](image)

**Figure A-5: Interconnect and LUT Stage**

The circuit uses the PRE signal to isolate the inputs and enable the cross-coupled PMOS transistors. When isolated, the PMOS transistors drive the interconnect output lines to a precharge state. This causes all the multiplexer inputs to be off, and the multiplexer outputs also go into the precharge state. When PRE is deasserted and CLK is asserted, the inputs propagate through the interconnect devices, and one rail of the dual-rail network is pulled down. The cascode feedback circuits (PMOS feedback) are activated and accelerate this transition. This causes one of the transistors in the multiplexer tree to turn on, and the output multiplexer cascode is activated as well. This circuit eliminates the noise margin problems with NMOS devices because the precharged circuit only propagates zero values.
through the NMOS devices. In addition, the cascode feedback circuits accelerate the time needed for these state transitions.

Figure A-6: Self-Timed PFU

In a conservative design style, the complement of the CLK signal is used to drive the PRE signal. In this case, half the cycle time is devoted to precharge, thus only half the cycle time is available for evaluation. However, a more aggressive design style would use the dual-rail encoding to detect changes in state, and develop a self-timed scheme which drives the PRE line. In this scheme, almost all the cycle time is available for evaluation, and the benefits of precharging are still applicable. Figure A-6 illustrates this scheme in the context of a two-phase clocking system. As the figure shows, the input data is released on the rising edge of the CLK. At the same time, the last stage of the PFU network is precharged. The outputs for the remaining stages have been precharged in the previous cycle. As the data propagates through the network, the outputs of the I/LUT PAIR circuits move from the precharge stage to one of the resolved stages. Each I/LUT PAIR contains one level of interconnect and one level of LUT resources. A NAND gate senses
the resolution of the output, and invokes a precharge for a I/LUT PAIR two stages back. When the CLK value falls, the output latch is ready to capture the resolved value for the whole PFU function on the next rising edge of the CLK signal. This scheme depends on relative delays for the various races in the circuit to function properly. However, the use of these aggressive custom design techniques is justified because additional levels of I/LAT PAIRs significantly increase the complexity of the functions that a PFU can support.

A.4 Conclusions

The layered PFU is our first attempt at PFU design, and the circuit design techniques used to build the layered PFU design are useful for future PFU designs. However, the microarchitecture of the PFU is likely to change as more information is available concerning the class of "interesting" functions. After the first iteration with the PRISC compilation tools, we have already learned some lessons which would be fruitful for future PFUs. For example, we have learned that the providing programmable resources for the bottom byte of an operation is much more important than the top bytes. Many of the PRISC compilation techniques such as BB-Table-Lookup and Jump-CFG heavily stress the bottom bits of the PFU. In addition, only the Boolean-CFG operation requires a full-word fanout operation, and only from the first bit. Thus, a hierarchical strategy which segments the interconnect resources in combination with a few "long" lines from the first bit could provide a better combination of programmable resources. Similar to instruction usage studies, function frequency studies will determine the allocation of resources for future PFUs.
Appendix B: Multi-cycle PFUs

As an initial step, this thesis has considered only single-cycle combinational PFUs. However, a broader class of functions can benefit from multi-cycle sequential PFUs. In the domain of hardware design, the principle problem introduced by these PFUs is synchronization with the other functional units. The latency of multi-cycle PFUs varies with the application program. Integer dividers and multipliers must also address this problem because the number of cycles needed for an operation is data dependent. These functional units solve the synchronization problem by arbitrating access to the result bus. This scheme can be extended to include multi-cycle PFUs.

![Figure B-1: Multi-Cycle PFU for Normalization](image)

Figure B-1 shows the usefulness of multi-cycle PFUs for the normalization operation. This operation takes at least three instructions in MIPS ISA, but is easily implemented in a multi-cycle PFU. The multi-cycle PFU iterates three times faster than the software loop because the AND instruction and the compare for the BEQ instruction is folded into the shift operation. In addition, since the multi-cycle PFU is invoked with a single instruction and does not manipulate the instruction stream, superscalar processors can easily proceed
past the normalize instruction to evaluate other non-data dependent instructions. A PRISC compiler can transform PFU-LOGIC software loops into multi-cycle PFUs, but a more interesting problem for multi-cycle PFUs is the process by which large combinational functions or complex sequential loops can be mapped onto multi-cycle PFU resources.

![Figure B-2: Multi-cycle PFU](image)

Behavioral synthesis techniques [12] solve this problem by separating a computation into datapath and control sections. The datapath contains data manipulation functions, and the control contains a sequencer that controls the datapath elements. For small scale problems, both functions can be implemented in a PFU, but Figure B-2 suggests another microarchitecture organization.

In this organization, the multi-cycle PFU consists of combinational PFUs and conventional functional units under the control of a microcoded engine. The microcode engine controls all the bypasses between the functional units, and the asynchronous protocol to communicate with the rest of the CPU datapath. A microcode engine is chosen because
microcode provides a convenient method for capturing the control information generated by behavioral synthesis. Both the microcode program and PFU are programmed on an application-specific basis.

This type of multi-cycle PFU organization is especially useful for the iterative mathematical algorithms. For example, the following inductive definition for the squaring function can be used to perform the square root operation.

Base Case:

\[(0.0)^2 = 0, \quad (0.1)^2 = 0.01\]

Inductive Step:

\[(0.q_1q_2\ldots q_{i-1}1)^2 = (0.q_1q_2\ldots q_{i-1})^2 + 2^{-i}.q_1q_2\ldots q_{i-1}01\]

This squaring algorithm requires a bit insert operation (0I), a shift operation (2i), and an addition operation. Since the shift is sequential, the shift and insert can be performed by a combinational PFU, and the addition by a conventional adder. Using the squaring function, the following operation performs the square root operation on a number, \(R\),

\[\text{if } (0.q_1q_2\ldots q_{i-1}1)^2 \leq R \text{ then } q_i = 1 \text{ else } q_i = 0\]

A multi-cycle PFU using the microcode sequencer to drive the iteration and the combinational PFU and adder to perform the squaring operations can easily implement the square root function. Similar algorithms exist for the evaluation of logarithm, division, and transcendental functions.
In summary, multi-cycle PFUs offer the ability to fold iterative software structures into hardware. When the hardware complexity of the software loop is small, the whole loop can be folded into a single PFU. When the body of the software loop requires dense boolean functions such as full word addition, customized functional units can be used in conjunction with PFUs and microcoded control to fold the loop into hardware. In both cases, the multi-cycle PFU offers a higher performance as compared with a RISC instruction based implementation whenever the loop branching overheads are a significant portion of the loop costs. This is typically true for iterative mathematical algorithms such as square-root and division. In addition, folding software loops into hardware frees the remainder of the CPU from the burden of instruction processing, so the CPU can start evaluating instructions beyond the software loop.
References


[30] Bill Grundman, Private Communication


