UNIVERSITY of CALIFORNIA
SANTA CRUZ

KESTREL: DESIGN AND IMPLEMENTATION OF A MASSIVELY PARALLEL COPROCESSOR

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

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

David M. Dahle

September 1999

The thesis of David M. Dahle is approved:

Richard Hughey, Chair

Kevin Karplus

Anujan Varma

Dean of Graduate Studies

Contents

List of Figures vii

List of Tables ix

Acknowledgements xii

1 Introduction 1
- 1.1 Architecture Design . . . 2
- 1.2 Design Constraints . . . 4
- 1.3 Human Resources . . . 6
- 1.4 Document Overview and Intended Audience . . . 6

2 Kestrel Processor Array 9
- 2.1 Kestrel Processing Element Overview . . . 10
  - 2.1.1 Architecture . . . 10
  - 2.1.2 Instruction Encoding . . . 12
- 2.2 Timing . . . 13
  - 2.2.1 Two-Phase Clocking . . . 13
  - 2.2.2 Signal Naming Conventions . . . 15
  - 2.2.3 Event Timing . . . 15
- 2.3 Design Constraints . . . 15
  - 2.3.1 PE Floorplan . . . 16
  - 2.3.2 PE Core Floorplan . . . 17
  - 2.3.3 Power Consumption . . . 18
- 2.4 Register Bank and Operand Buses . . . 20
  - 2.4.1 Register Bank . . . 20
  - 2.4.2 Operand Buses . . . 21
  - 2.4.3 Problems and Improvements . . . 24
- 2.5 ALU . . . 25
  - 2.5.1 Architecture . . . 25
  - 2.5.2 Circuit Design . . . 27
  - 2.5.3 Problems and Improvements . . . 28
- 2.6 Comparator . . . 30
  - 2.6.1 Architecture . . . 30

2.6.2 Multiprecision Comparisons . . . 30
2.6.3 Circuit Design . . . 33
2.6.4 Problems and Improvements . . . 34
2.7 Flag and Result Selectors . . . 35
2.7.1 Architecture . . . 35
2.7.2 Implementation Issues . . . 38
2.7.3 Problems and Improvements . . . 38
2.8 Bit Shifter and Mask Flag Generator . . . 39
2.8.1 Architecture . . . 39
2.8.2 Nested Conditional Processing . . . 41
2.8.3 Implementation Issues . . . 43
2.8.4 Problems and Improvements . . . 44
2.9 Wired-OR and Mesh Interconnections . . . 44
2.9.1 Wired-OR . . . 44
2.9.2 Bit-Serial Mesh . . . 45
2.9.3 Problems and Improvements . . . 46
2.10 Multiplier . . . 49
2.10.1 Architecture . . . 49
2.10.2 Booth Multiplication . . . 51
2.10.3 Modified-Booth Multiplication . . . 53
2.10.4 Implementation Details . . . 55
2.10.5 Layout Design . . . 57
2.10.6 Problems and Improvements . . . 57
2.11 Static RAM . . . 58
2.11.1 Architecture . . . 59
2.11.2 Layout Design . . . 59
2.11.3 Problems and Improvements . . . 61
2.12 Chip Fabrications . . . 62
2.12.1 Multiplier Test Chip . . . 62
2.12.2 Memory Test Chip . . . 62
2.12.3 2-PE Prototype . . . 63
2.12.4 64-PE Chip . . . 65

3 Kestrel System **68**
3.1 Board Overview . . . 69
3.2 PCI Interface . . . 72
3.2.1 Address Space Configuration . . . 72
3.2.2 Interrupts . . . 75
3.2.3 Initialization . . . 76
3.2.4 Problems and Improvements . . . 77
3.3 Clock Generator . . . 77
3.3.1 Design . . . 77
3.3.2 Problems and Improvements . . . 78
3.4 Array Controller . . . 80
3.4.1 Design . . . 80
3.4.2 Instruction Pipelining . . . 87

3.4.3 Problems and Improvements . . . 87
3.5 Command Register . . . 89
3.5.1 Operating Modes . . . 90
3.5.2 Kestrel-Array Configuration . . . 93
3.5.3 Command-Register Reset . . . 93
3.5.4 Problems and Improvements . . . 94
3.6 Status Register . . . 95
3.6.1 Most-Significant Byte . . . 95
3.6.2 Least-Significant Byte . . . 95
3.6.3 Problems and Improvements . . . 97
3.7 Input/Output Data Queues . . . 98
3.7.1 Command and Status Register Fields . . . 98
3.7.2 Programmable Flags . . . 99
3.7.3 Reading from the Queues . . . 100
3.7.4 Input-Queue Handling . . . 101
3.7.5 Output Queue Handling . . . 102
3.7.6 Flushing the Queues . . . 102
3.7.7 Problems and Improvements . . . 103
3.8 Host Interface . . . 104
3.8.1 Low-Level Interface . . . 104
3.8.2 High-Level Interface . . . 107
3.8.3 Problems and Improvements . . . 108
3.9 User Interface . . . 109
3.9.1 Runtime Environment . . . 110
3.9.2 Application Programmer's Interface . . . 114
3.9.3 Programs and Improvements . . . 116
3.10 Summary: Running a Kestrel Program . . . 117
3.10.1 Board Initialization . . . 117
3.10.2 Interrupt Handler . . . 118
3.10.3 Program Termination . . . 119

4 Evaluation and Future Work **120**
4.1 Applications . . . 120
4.1.1 Edit Distance . . . 121
4.1.2 Smith and Waterman . . . 123
4.1.3 Viterbi HMM Scoring . . . 124
4.1.4 Sum-of-all-paths HMM Scoring . . . 126
4.1.5 Performance . . . 128
4.2 Architectural Comparison . . . 129
4.2.1 CNAPS . . . 129
4.2.2 MasPar . . . 130
4.2.3 Pentium MMX . . . 130
4.2.4 MGAP . . . 131
4.2.5 RaPiD . . . 131
4.3 Problems and Improvements Summary . . . 132
4.3.1 Layout Design . . . 132

4.3.2 Board and Controller Design 133
4.3.3 System Software Improvements 134
4.4 Future Work 135

Bibliography **137**

A Programming the PLX 9050's EEPROM **140**

B Host High-Level Command Format **143**

C Verilog Simulation **145**

D Simulating the 64-PE Kestrel Chip **147**

E Testing the 64-PE Kestrel Chip **151**

List of Figures

1.1 High-level diagram of the Kestrel. 4

2.1 Diagram of the Kestrel array. 9
2.2 Diagram of the PE architecture. 12
2.3 Two-phase clocking diagrams. 14
2.4 Processing element floorplan. 17
2.5 PE bus and a global-logic routing directions and layers. 17
2.6 Floorplan of the PE core. 19
2.7 Register-bank cell design. 20
2.8 Register bank interface to operand and result buses. 22
2.9 The PE ALU architecture. 26
2.10 ALU circuit that converts the function codes to kill and generate signals. . 29
2.11 ALU circuit design. 29
2.12 Comparator architecture. 31
2.13 Circuit design for the comparator's subtractor. 34
2.14 Circuit design for the comparator's full adder. 34
2.15 Flag and result selectors. 36
2.16 The bit shifter architecture. 40
2.17 Mask-flag generator architecture. 40
2.18 Wired-OR and mesh circuits. 45
2.19 Bit-serial mesh communication patterns. 47
2.20 Multiplier interface to the Kestrel PE. 50
2.21 Examples of normal, Booth recoded, and modified-Booth recoded multiplication. 52
2.22 Organization of the modified-Booth multiplier. 56
2.23 Booth-multiplier partial-product computation circuitry. 56
2.24 Booth-multiplier recoder circuitry. 57
2.25 Multiplier floorplan. 58
2.26 SRAM interface to Kestrel PE, including address calculation. 60
2.27 Floorplan of the SRAM. 61
2.28 Photographs of the multiplier and memory test chips. 63
2.29 Photograph of the 2-PE prototype and 64-PE chips. 64
2.30 Floorplan of the 64 PE Kestrel chip. 67

2.31 Row of eight Kestrel PEs with global control logic. 67

3.1 Diagram of the Kestrel system. 69
3.2 Photograph of the Kestrel board. 70
3.3 Diagram of the Kestrel board architecture. 71
3.4 Clock-generator circuit. 78
3.5 Relationship between clock-generator delayed signals. 79
3.6 Array-controller's program-counter circuits. 82
3.7 Array-controller registers. 84
3.8 Data movement between controller and array. 85
3.9 Kestrel-array instruction immediate selector. 86
3.10 Array-controller execution pipeline showing only data movement. 88
3.11 Correct wiring for the FPGA's PROM containing the controller's configuration. 89
3.12 The Kestrel-PE array and reconfiguration controls for dynamic array resizing. 94
3.13 Command-line format for input and output files. 112
3.14 A simple example program demonstrating the application programmer's interface to the Kestrel host server. 115

4.1 Example of a dynamic-programming matrix. 122
4.2 An example hidden Markov model (HMM). 125

A.1 Current PLX 9050 EEPOM configuration from the file `eeprom3.mon`. . . . 142

List of Tables

1.1 Contributors to the Kestrel system design and implementation. 7

2.1 The 52-bit Kestrel PE instruction fields. 13
2.2 Kestrel-array execution pipeline. 16
2.3 Register bank and operand bus driver external data and control signals. . . 22
2.4 Instruction encoding for selecting the operandB bus source. 23
2.5 Control and data signals for the ALU. 26
2.6 Function encoding of ALU operations, by Don Speck [5]. 27
2.7 Comparator control and data signals. 32
2.8 Encodings for multiprecision comparison operations. 32
2.9 Flag and result selector control and data signals. 36
2.10 The flag choices and their corresponding instruction encodings `fb_v2[4:0]`. 37
2.11 Result selection and instruction encoding. 37
2.12 Bit shifter control and data signals. 41
2.13 Mask flag generator control and data signals. 42
2.14 Signal encodings for the bit shifter and mask flag generator. 43
2.15 Wired-OR and Mesh control and data signals. 46
2.16 Decoded values for mesh control signals. 48
2.17 Multiplier control and data signals. 50
2.18 Kestrel multiplier's booth recoding. 53
2.19 SRAM and address generator control and data signals. 60
2.20 SRAM addressing mode selection based on the `rm_v2[1:0]` instruction field. 61

3.1 Kestrel local-bus address translation based on the top three bits of the 28-bit address space. 73
3.2 Address space allocation for board components and their PLX 9050 register values. 74
3.3 Local Address Space Bus Region Descriptor Register. 74
3.4 Local Address Space Bus Address Remap Register. 75
3.5 Local Address Space Range Register (LASXRR). 76
3.6 Interrupt Control/Status Register. 76
3.7 Instruction fields used by the array controller and board to control a Kestrel program. 81

3.8 Non-instruction-bit signals used in the program and loop-counter stacks from Figure 3.6. 83
3.9 Selection of the new PC based on the instruction fields and branch conditions. 83
3.10 Data signals used by the scratch and wired-OR bit-shifter register in Figure 3.7. 84
3.11 Non-instruction-bit signals used for data movement from Figure 3.8. 86
3.12 Command-register bit fields that control controller operations, the Kestrel-array configuration, and two bits to workaround problems with the data-queue interface. 91
3.13 Summary of controller operating modes set from the command register. . . 91
3.14 Top byte of the status register. 96
3.15 Bottom byte of status register when not in diagnostic mode. 96
3.16 Lower byte of status register in `DIAGNOSTIC` mode, controlled by the `D_OUT[2:0]` instruction field. 97
3.17 Fields related to queue operations from the command and status registers. . 99
3.18 Queue configuration registers for almost-empty and almost-full thresholds. . 100
3.19 Low-level interface between the Kestrel host machine or simulator and the runtime environment. 106
3.20 Status signals sent by the host to the runtime environment. 106
3.21 Host high-level command for Kestrel-program execution. 108
3.22 Error conditions returned by the `KCMD_GET_ERROR_CODE` message. 108
3.23 General runtime-environment command-line options. 110
3.24 Application programmer's interface to the Kestrel host server. 115

4.1 Wall time in seconds to search 10MByte of the Swissprot database on the 20 MHz Kestrel. 128
4.2 Performance in MOPS of selected SIMD chips. 129

D.1 Contents of directories in `/projects/kestrel/design/` containing Kestrel chip layout files. 148
D.2 Contents of directory `/projects/kestrel/design/Kestrel64/SIMULATE` containing files and scripts to simulate the Kestrel 64-PE chip design. 149

E.1 Instruction bits sharing the same tester channel. 153

Abstract

Kestrel: Design and Implementation of a Massively Parallel Coprocessor

by

David M. Dahle

Kestrel is a massively parallel coprocessor system designed to accelerate biological sequence analysis while providing general-purpose computing capabilities. The system, implemented on a PCI card, consists of 512 SIMD, 8-bit processing elements (PEs) connected in a linear array that communicate through shared-register banks. The PE array is implemented as eight 64-PE, 1.4 million transistor chips implemented in the HP 0.5 μm technology. An array controller provides global instruction sequencing and manages data movement between the array and host machine, and the host machine's software controls board operations and provides system access to users at remote workstations.

Many people contributed to the design and implementation of the Kestrel system, so this document focuses on the author's contributions while including enough information to provide relatively complete hardware documentation. The author's contributions include the physical layout design of most of the processing element, layout simulations, Kestrel board and array controller debugging, the runtime environment's program-execution core, and integration of all the various hardware and software components to form a working system.

Acknowledgements

I would like to thank Richard Hughey, Kevin Karplus, Anujan Varma, Leslie Grate, and Eric Rice for providing valuable feedback on this document. I would also like to thank all the members of the Kestrel team who I worked with over the years, including Jeff Hirschberg, Don Speck, Hansjörg Keller, Doug Williams, Osama Salem, Justin Meyers, Jennifer Leech, Leslie Grate, Liz Avila, Rachel Karchin, Richard Hughey and Kevin Karplus. Also, thanks to Dragon Systems Inc. for producing wonderful voice recognition software that spared my hands and wrists.

Chapter 1

Introduction

Kestrel is a massively parallel, linear-array coprocessor designed to accelerate sequence analysis algorithms from computational biology [5]. Sequence analysis involves comparing sequences representing protein, RNA, or DNA strands, and describing how related the sequences are based on some criteria. Sequence analysis uses dynamic-programming algorithms that efficiently map to Kestrel's linear-array, Single Instruction, Multiple Data (SIMD) [7] architecture. Kestrel allows the use of computationally expensive sequence analysis algorithms due to its accelerated execution times, while the simple architecture and design provide a low-cost system.

The sequence analysis algorithms used in computational biology are based on dynamic-programming algorithms such as Smith and Waterman [19] and hidden Markov models (HMMs) [17]. These algorithms construct an m by n matrix of costs, where m and n are the length of a query sequence and the comparison sequence, respectively. Each element in the matrix is calculated by adding terms for the transition from the adjacent elements and choosing the minimum or maximum of the resulting sums, depending on whether the

numbers are costs or scores, respectively.

Kestrel's linear-array architecture solves dynamic-programming problems efficiently, because the calculation for each element depends only on the results from the two preceding diagonals in the matrix. All elements along a diagonal can be calculated simultaneously, and the values from the preceding calculations are always available from either the current or adjacent processor. Additionally, the Kestrel Processing Element (PE) has hardware to perform addition and minimization in one instruction, improving the performance of the dynamic-programming calculations. The structure of the linear PE network matches the data flow of the sequences, where the query sequence is fixed in the PEs, and the database sequences shift through the chain, naturally appearing in the correct PE at the correct time.

However, the goal of the Kestrel design was never just to develop a dynamic-programming engine, but to produce a machine with general-purpose computing capabilities. Consequently, the Kestrel design has been a constant battle between adding more features and keeping the design simple. While more features would improve the performance of a class of applications, they increase the design complexity and cost.

1.1 Architecture Design

The Kestrel system is a linear chain of 8-bit SIMD PEs, where PEs connect through shared register banks [18], as shown in Figure 1.1. PEs communicate through the 32-byte, 3-port register banks, and data I/O from the end register banks is handled by an array controller. The array controller moves data between the host and the end register banks, handles instruction flow, broadcasts instructions to the array, and manages the I/O queues. Each PE includes a multifunction ALU, comparator and minimizer, condition stack,

multiplier, and 256-byte local memory. All arithmetic components have extra hardware to handle both signed and multiprecision calculations. Many of the components can be used in parallel, so a single instruction could add two numbers, compare the result with a third and select the minimum or maximum to store in a register, read a value from local memory, and perform an operation on the condition stack. Multiply instructions can be done in parallel with SRAM and condition stack operations, but not with ALU or comparator/minimizer operations.

The linear-array and SIMD structures provide the most efficient PE organization for sequence analysis algorithms while keeping the design simple. In sequence analysis algorithms, every dynamic-programming element requires the same calculation, so all PEs perform the same operation. The SIMD model also produces the simplest hardware, since instruction decoding can be performed globally, rather than in the PEs.

To overcome limitations of the basic SIMD model, several features have been added to Kestrel that provide support for Multiple Instruction, Multiple Data (MIMD) operations. While every PE receives the same instruction broadcast, they can disable themselves based on local data. Thus, MIMD operations can be simulated by disabling some PEs, while others perform the broadcast instructions. The array controller can also change the global instruction stream by executing a jump based on conditions computed in the array.

The Kestrel design provides the basic functionality required for sequence analysis dynamic-programming algorithms, while supporting additional operations in hardware for more general-purpose applications. The ALU/minimizer supports the recurrence used by dynamic-programming algorithms, and the a local memory supports the traceback phase of dynamic-programming algorithms. The local memory supports indexed addressing based

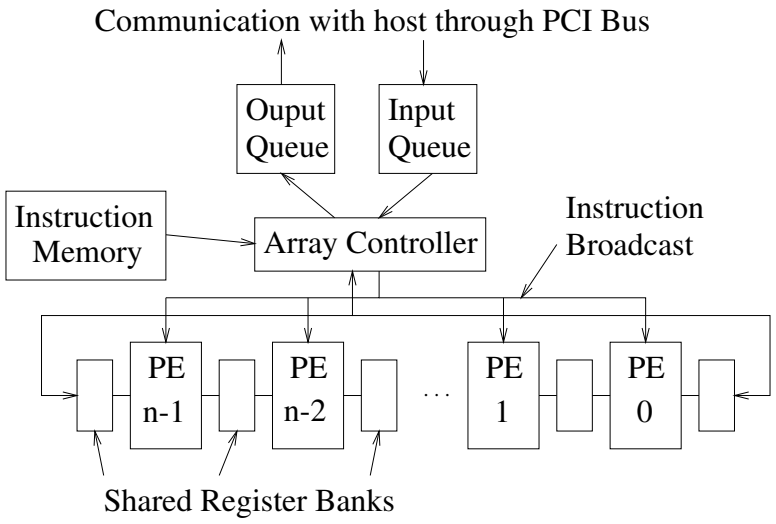

Figure 1.1: High-level diagram of the Kestrel hardware system. Shared registers banks connect adjacent processors in a linear chain. A controller broadcasts instructions to the processors, and handles transfers of data between the end register banks and the host system, buffered through data queues.

on local data, supporting table lookups based on the character stored in a PE. Features not explicitly needed by sequence analysis include a multiplier and nested conditional stack. The multiplier supports signal-processing applications, and the condition stack allows PEs to be disabled based on complex conditionals, and supports efficient packing of bit vectors. The bit vectors can then be stored in the local memory for later use, as done in sequence alignment algorithms. Most components also support multiprecision operations that provide support for both sequence analysis algorithms and other applications.

1.2 Design Constraints

Matching the I/O requirements of the Kestrel array to the disk system is crucial to Kestrel's performance. Designing a system with high bandwidth would produce a fast system, but would greatly increase the cost and complexity. On the other hand, a system

with lower bandwidth capability than can be provided by inexpensive disks would hurt overall system performance. Thus, the PE design attempts to match the number of instructions required to perform basic operations to the bandwidth of typical disk systems. For example, Smith and Waterman sequence alignment requires approximately 20 instructions per database character, producing a bandwidth requirement of approximately 1 MB/s with a clock speed of 20 MHz, matching the data transfer rate from a typical workstation disk.

Since the broadcast required by the SIMD architecture will eventually place an upper limit on the system's maximum clock speed, it is advantageous to perform as many operations in a single instruction as possible. Given this constraint, the Kestrel PE has a relatively orthogonal instruction encoding, allowing independent operation of several PE components during each instruction. This allows a virtually unlimited number of possible instructions, as operations between different components can be combined in a flexible way.

While several operations can be performed in a single instruction, the number of instruction bits has to be restricted to allow the PEs to fit in a relatively small 84-pin package. Consequently, some operations between components may seem to interact in bizarre ways to those unfamiliar with the architecture. Also, the PE design attempts to minimize the required circuitry at the expense of programmability. For example, the ALU is flexible and provides many different operations, but the operand buses are nonorthogonal. Two buses must always come from registers while the third can come from a variety of sources within the PE. As a result of these architectural nuances, efficient Kestrel programming requires detailed knowledge of the Kestrel architecture.

1.3 Human Resources

Many people have contributed to the design and implementation of the Kestrel system over the years, as summarized in Figure 1.1. Jeff Hirschberg performed early architectural simulations, wrote the first assembler and MasPar simulator, generated test vectors to verify the chips both in simulation and on the tester, designed the Kestrel board, and wrote the Verilog model used as the specification for much of the layout design. Don Speck contributed to virtually every aspect of the architecture, circuit, and layout designs, and his extensive design experience was crucial to the project's success. Hansjörg Keller designed the board's array controller and implemented several applications, and Doug Williams designed the static RAM. The author's contributions included the physical layout design of most of the PE and multiplier design, layout simulations, Kestrel board and array controller debugging, the runtime environment's program-execution core, and integration of all the various hardware and software components to form a working system. Justin Meyers and Jennifer Leech designed and implemented the runtime environment's user interface, and Osama Salem implemented most of the host network server used by the runtime environment.

1.4 Document Overview and Intended Audience

The primary audience for this document is a future hardware designer responsible for the next revision of the Kestrel system, but it should also prove useful to application developers. While this document does not discuss Kestrel assembly language, the low-level design discussion should help those already familiar with the architecture understand

Team Member	Major Contributions
Jeff Hirschberg	Verilog Architecture Models, Simulators and Assemblers, Board Design, Test-Vector Generation and Chip Testing
Don Speck	Architecture and VLSI Design Consultant
David Dahle	VLSI Design and Simulations, Chip Fabrications, Board and Controller Debugging, Applications, Runtime Environment/Server Execution Core
Doug Williams	SRAM Design and Layout
Hansjörg Keller	Array Controller Design, Applications
Osama Salem	Kestrel Host Network Server
Justin Meyers	Runtime Environment, Simulator, Applications
Jennifer Leech	Runtime Environment, Simulator

Table 1.1: Contributors to the Kestrel system design and implementation.

the nuances; in particular, Section 3.4 provides much-needed documentation of the array controller.

This document focuses on the author's contributions to the Kestrel project, while providing enough additional information to form relatively complete hardware documentation. For example, the author designed both the architecture and layout for the multiplier used in the Kestrel PE, so a detailed discussion of Booth multiplication is included. On the other hand, Doug Williams and Don Speck designed the static RAM, and author's contribution was to connect it to the PE and global logic, so only a discussion of the SRAM's interface is included.

The subsequent chapters are organized as follows: Chapter 2 presents a detailed discussion of the Kestrel PE, starting with an overview of the architecture and detailing the design and implementation of each component. It also discusses the chip fabrications, test results, and design errors found. Chapter 3 presents the overall Kestrel system, discussing the Kestrel board and support software that executes Kestrel programs. As Jeff Hirschberg

designed the Kestrel board, Chapter 3 presents a design overview, how to use the hardware interface to execute a Kestrel program, and issues relating to the software support system. Along the way, the "Problems and Improvements" sections discuss problems with the current implementation and presents suggestions for future improvements. Chapter 4 concludes by presenting the basic sequence analysis applications and their performance compared to a typical workstation, comparisons to other contemporary SIMD machines, and a summary of changes and improvements that should be made to the next hardware and software revisions.

Chapter 2

Kestrel Processor Array

The Kestrel processor array consists of 512 processing elements (PEs) and 513 register banks. As shown in Figure 2.1, each PE shares a left and right register bank with the two neighboring PEs. All interprocessor communication occurs through shared register banks, with data written to an adjacent register bank available for reading by the neighboring PE. The array controller, discussed in the next chapter, controls data movement in and out of the array ends. As a result of the SIMD instruction stream, a sequence of instructions that reads from the left and writes to the right register bank causes data to stream through the array.

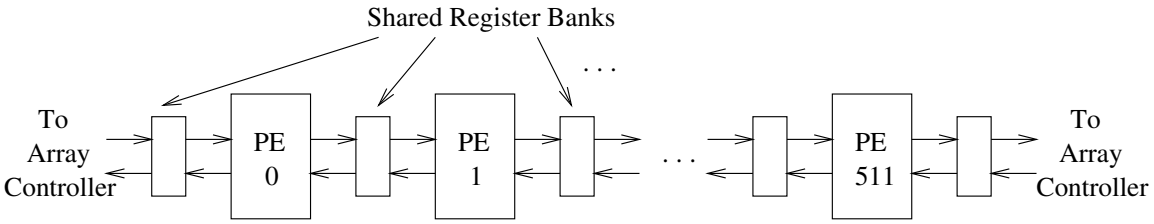

Figure 2.1: Diagram of the Kestrel processing element (PE) array with shared register banks.

Jeff Hirschberg was responsible for developing the architecture models for the Kestrel PE array, and developed test patterns to verify the correctness of both the layout and fabricated chips. Appendix C describes how to use the Verilog model to generate test vectors for layout simulation and chip testing. The author was responsible for all physical layout design, based primarily on the detailed Verilog model, with two exceptions: Doug Williams designed and implemented the SRAM, and the author designed the multiplier architecture. Additionally, VLSI expert Don Speck was an ever-present force, making critical contributions to most aspects of both architecture and VLSI designs.

This chapter presents an overview of the PE design along with important design constraints, followed by a detailed description of each design component. These descriptions focus on the author's contributions, but often include additional information for completeness. Section 2.1 presents an overview of the PE architecture and instruction encoding. The subsequent sections describe each component in detail: register bank (Section 2.4), ALU (Section 2.5), comparator (Section 2.6), flags selector (Section 2.7), bit shifter (Section 2.8), wired-OR and bit-serial mesh (Section 2.9), multiplier (Section 2.10), and SRAM (Section 2.11). Finally, Section 2.12 describes the 64-PE chip, as well as the three test chips used to verify individual components of the 64-PE chip.

2.1 Kestrel Processing Element Overview

2.1.1 Architecture

Each 8-bit Kestrel PE includes several functional units that can work independently, allowing a wide spectrum of operations in a single instruction. The functional units include an ALU, comparator, bit shifter, mask flag generator, multiplier, and a 256-byte

local memory. Additionally, each processor can read and write to two adjacent shared register banks. Three operand buses and one result bus connect the functional units, as shown in the high-level diagram in Figure 2.2.

The operand buses each have a specific function, and consequently have different capabilities. An instruction selects values for operandA (opA) and operandC (opC) independently from the left or right shared register bank, and operandB (opB) from several possible sources, including the multiplier, bit shifter, memory data register (MDR), instruction immediate, and operandC. OperandA and operandB are the most general purpose as the primary ALU and multiplier inputs, and operandC is either multiplier or comparator operand. The asymmetry of the operand buses complicates application and compiler development, but greatly simplifies the hardware implementation.

During one instruction, an ALU operation can be performed on operandA and operandB, the ALU result compared with operandC, and the final result selected based on the comparison or other condition. Alternatively, operandA and operandB can be multiplied, optionally adding in the most significant byte of the previous product and/or operandC, efficiently supporting multi-precision multiplications. An adjacent shared register bank stores the result if a local mask flag enables the PE, and the local memory optionally saves the result using a locally or globally computed address. Local memory writes execute immediately, but reads must first load a value into a memory data register (MDR) and consequently must be scheduled at least one instruction before being needed as an operand. Flag logic processes flags from various sources and produces a flag bit that can be used with the bit shifter and mask flag generator to perform nested conditional operations. Alternatively, the flag bits can be packed into bit vectors using the bit shifter, and stored in local

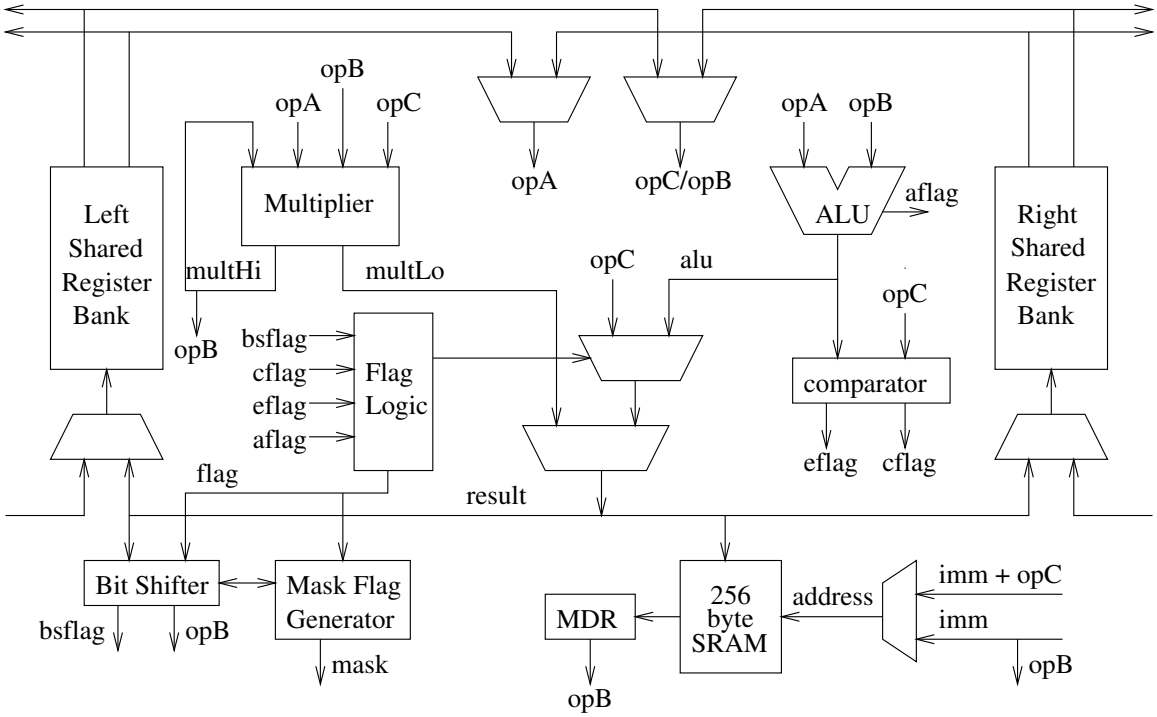

Figure 2.2: Diagram of the PE architecture with shared register banks.

memory for later processing, as occurs during all sequence alignment algorithms.

2.1.2 Instruction Encoding

The Kestrel PE instruction fields, shown in Table 2.1, represent a balance between independence of PE functional units, minimizing global logic decoding, and keeping the number of instruction bits small enough for an 84-pin package. The ALU/multiplier, bit shifter, flag and result selectors, and local memory can perform independent operations simultaneously. However, all compete for the same operands. For example, if an instruction chooses operandB as the immediate and performs a local memory operation, the local memory computes its address from the same immediate. The instruction fields are usually independent, affecting only one functional unit, though there are some exceptions. For

Instruction Field	Purpose
nop_v2	Array Controller No Operation
wr_v2	SRAM Write Enable
rd_v2	SRAM Read Enable
finv_v2	Flag Invert
lc_v2	ALU Latch Carry-Out
mp_v2	ALU Select Carry-In
ci_v2	ALU Carry-In (if selected)
force_v2	Unconditional Instruction Execution
rm_v2[1:0]	Result Selector
opB_v2[2:0]	OperandB Source Selection
bit_v2[3:0]	Bit Shifter Function
fb_v2[5:0]	Flag Selector
func_v2[4:0]	ALU/Multiplier Function
dest_v2[5:0]	Destination Register
opA_v2[5:0]	OperandA Register
opC_v2[5:0]	OperandC Register
imm_v2[7:0]	Immediate

Table 2.1: The 52-bit Kestrel PE instruction fields, plus the Array Controller's no operation signal nop_v2.

example, the result select field rm_v2[1:0] selects both the result and the local memory addressing mode, with a combination of ci_v2 and lc_v2 overriding rm_v2[1:0] and forcing selection of the multiplier output.

2.2 Timing

2.2.1 Two-Phase Clocking

The Kestrel system uses a nonoverlapping two-phase clocking discipline, greatly simplifying memory element design, since level-sensitive transparent latches, rather than edge-triggered latches, are most commonly used in CMOS design. The two alternating clock phases, called ϕ_1 (phi1) and ϕ_2 (phi2) never overlap, separated by the gap encompassing all clock skew, as shown in Figure 2.3 (a). Each clock phase can directly control the master

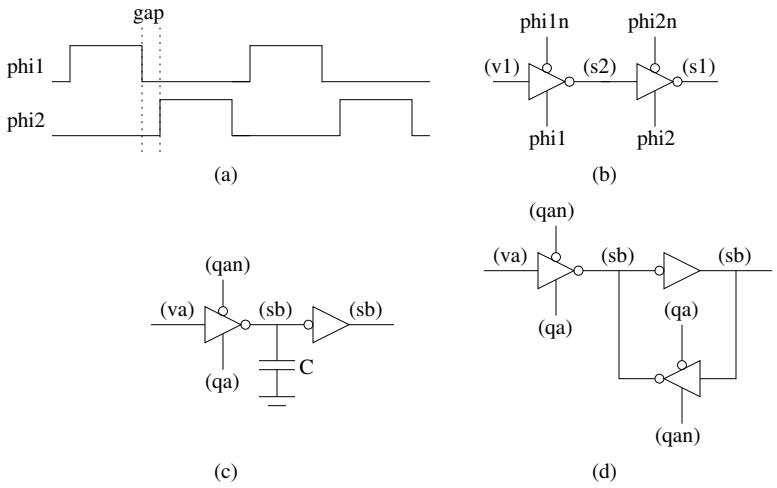

Figure 2.3: (a) Structure of nonoverlapping two-phase clocks; (b) master and slave transparent latches of a dynamic register; (c) dynamic latch design; and (d) static latch design.

and slave transparent latches of a dynamic register, as shown in Figure 2.3 (b).

Knowing when signals are valid is critical to correct circuit design [13]. A signal satisfying a dynamic latch's setup and hold times clocked on ϕ_1 is called `V1`, and a signal valid during all of ϕ_1 is called `S1`, and similarly for signals valid during ϕ_2. A signal valid at the end of ϕ_1, during a gap, and during all of ϕ_2 is called `V1S2`. Additionally, information signals can be combined with clocks to produce timing signals. For example, an `S1` signal ANDed with ϕ_1 produces a `Q1` timing signal, and a `S2` signal ANDed with ϕ_2 produces a `Q2` signal. An `HL1` signal that precharges during ϕ_2 and evaluates during ϕ_1 indicates only one possible high-to-low transition per clock cycle.

Figure 2.3 (c) shows the design of a dynamic latch. Clocked by a `QA` timing signal, a `VA` input becomes a `VASB` output, or more conservatively `SB`. The latch storage is the capacitance of the internal wire, so the output will not stay valid indefinitely if not refreshed. Figure 2.3 (d) shows the a static latch design, which holds the stored value indefinitely, even if the clock is turned off.

2.2.2 Signal Naming Conventions

All signals in the Kestrel design contain timing information as part of their name. Ideally, the names indicate the minimum timing requirements for correct operation. However, many signals indicate less restrictive timing, and are often highly over-optimistic. For example, many global control signals are labeled `V2S1`, when they are more likely to be `V1` when operated near the maximum clock frequency. When tables in subsequent sections list control signals with timing information, they also indicate correct minimum timing requirement.

2.2.3 Event Timing

PE instruction execution pipelines over three clock cycles, summarized in Table 2.2. Instruction broadcast occurs during the first cycle, with instructions latched at the end of phase 2. Since the instruction latches are transparent, instructions begin global decoding and broadcast during phase 2. Most instruction execution occurs during the beginning of cycle 2, including register reads, ALU/comparator, and multiplier low byte calculations, producing an 8-bit result and a flag. During phase 2, the PE writes to a register, performs a bit shifter and mask flag generator operation, completes the multiplier high-byte calculation, and a read or write to the SRAM. Finally, the third cycle performs the wired-OR and communicates the result to the array controller.

2.3 Design Constraints

Kestrel's SIMD architecture requires an instruction broadcast every cycle. Consequently, the primary layout design constraint was effective routing of instruction bits while

Clock Cycle	Clock Phase	Events
1	ϕ_2	Latch instruction globally
2	ϕ_1	Register read, ALU, comparator, Multiplier low byte, Mesh communication, SRAM address calculation
2	ϕ_2	Register write, Bit Shifter, Mask Flag generation, Multiplier high byte, SRAM read/writes
3	ϕ_1	Wired-OR evaluation on each chip
3	ϕ_2	Wired-OR communication to array controller

Table 2.2: Three-cycle kestrel-array execution pipeline, with a cycle for instruction broadcast, instruction execution, and wired-OR evaluation and communication.

minimizing the delay and power consumption. Toward this goal, the design strategy focused on decoding instructions globally, producing control signals that simplified PE circuit designs and decreased the overall transistor count, in turn reducing overall switching and power consumption.

2.3.1 PE Floorplan

Figure 2.4 shows the Kestrel-PE floorplan. The register and bus control logic handles reads and writes to the register bank and the placement of register values onto the operand buses. The PE core contains all components except the multiplier and SRAM, and abuts the bus drivers to minimize wire length. The multiplier and SRAM blocks consume about 20 and 50 percent of the PE area, respectively, and place the largest constraints on the PE's shape.

Routing global signals and local operand buses placed constraints on metal layer usage, with the overall routing scheme shown in Figure 2.5. Each PE must route global

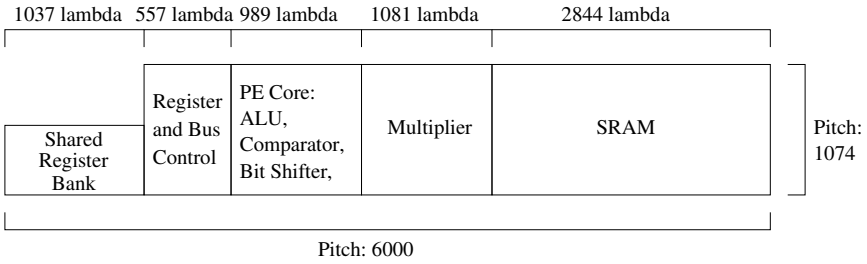

Figure 2.4: Processing element floorplan with blocks for the register bank and control logic, multiplier, SRAM, and PE core containing the remaining components.

logic vertically, communicate with two register banks and route operand buses horizontally to the multiplier and SRAM. Global control signals for the bus control and ALU blocks run in metal2, since there are over 80 wires and circuitry must fit underneath. Global control signals for the multiplier and SRAM run in metal1, since there are few and the operandB and result buses route over the multiplier to the SRAM in metal2.

2.3.2 PE Core Floorplan

Figure 2.6 shows the floorplan of the PE core, excluding the multiplier and SRAM, and including the mesh and wired-OR connections, discussed in Section 2.9. The multi-

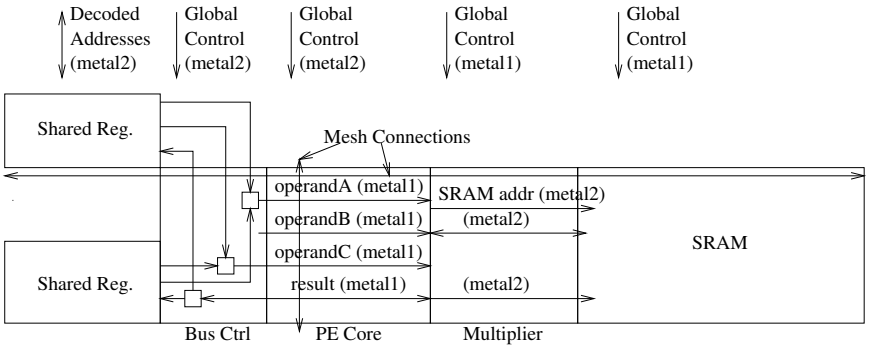

Figure 2.5: PE bus and a global-logic routing directions and layers.

plier and SRAM have distinct functions and operate in isolation, whereas the remaining PE components interlock to share local data and global control signals. The major components organize into bit-slices, with operand buses routed in metal1, and global control signals routed in metal2. The five flag MUXs for the ALU, comparator, bit shifter, and mesh align vertically to share the eight flag MUX control signals `fSel_v2s1[3:0]` and `fSel_v2s1n[3:0]`. The second stage of the flag selector abuts the result driver control logic to reduce flag-based result selection delay.

2.3.3 Power Consumption

The PE design required careful attention to power consumption, as PE replication amplifies design choices. Consequently, excess circuit switching in the PE could noticeably increase overall power requirements. Global logic performs all instruction decoding, transmitting only low-level control signals. Consequently, the number of transistors in each PE is reduced at the expense of more global control signals.

For example, the 4-to-1 flag MUXs, implemented using four clocked inverters, take eight global control signals, two for each clocked inverter, instead of two for the entire MUX. This choice increases the number of global control signals, but decreases the overall power consumption by reducing the number of transistors across the entire chip. Another example is the sending of both polarities of a global signal when PE circuitry uses both polarities in every bit-slice.

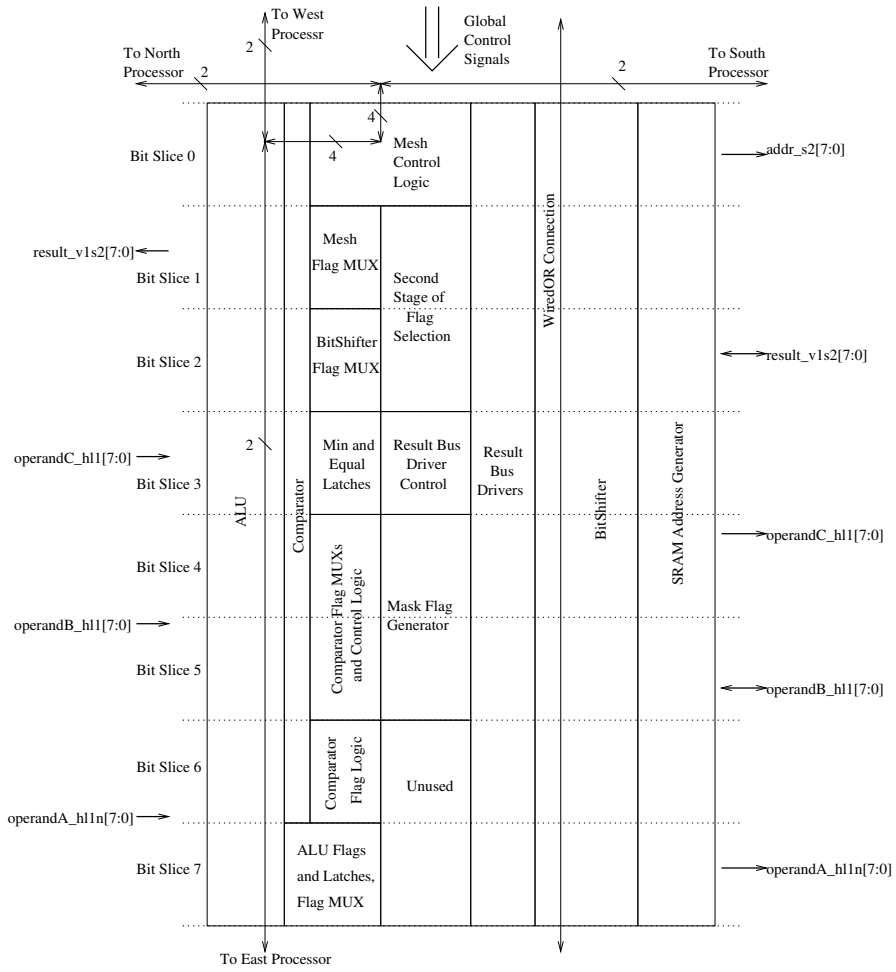

Figure 2.6: Floorplan of the PE core containing the ALU, comparator, bit shifter and mask flag generator, flag and result selector, and the SRAM address generator. Local data signals run horizontally in metal 1, and global control signals run vertically in metal 2 .

2.4 Register Bank and Operand Buses

The 32-byte register bank has two read ports and one write port, supplying two operands and storing one result in a single instruction. A register bank with an additional read port would provide more flexible instructions, at the expense of longer access times and larger area. With thoughtful instruction scheduling, a third operand can often be loaded into the SRAM's MDR, providing general three-operand instructions with no access time penalty.

2.4.1 Register Bank

Figure 2.7 shows the circuit design of the 3-ported register bank cell, with layout designed by Don Speck. The register bank consists of an array of 32 by 8 cells, leading to a height of half the PE height. A 16 by 8 by 2 array of cells would give a height matching the PE height, and would reduce read and write line capacitance, but would require 2-to-1 MUX to complete the address decoding locally in every PE. The former design simplified PE design without significant increase in access time, and allowed global address decoding

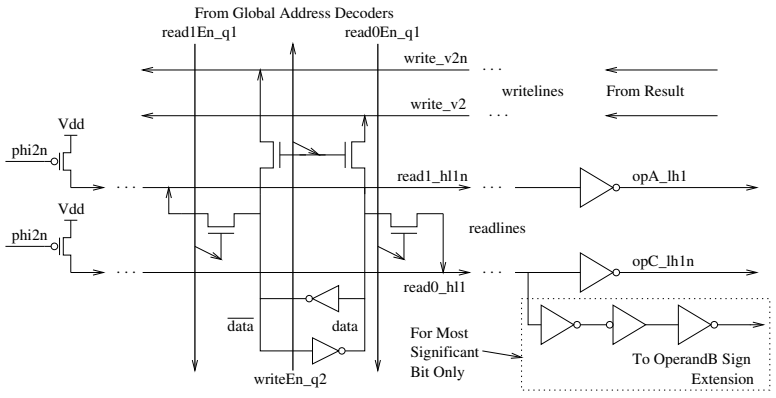

Figure 2.7: Circuit design for the 3-ported register bank cell. Layout design by Don Speck.

to be shared between interleaved rows of register banks.

Matching the overall PE timing scheme, reads occur during phase 1, and writes and readline precharging occur during phase 2. Global address decoders select the appropriate column of cells for each of the three ports, qualified with the appropriate clock signals to prevent erroneous reads and writes during phase 2. During a read, an NMOS transistor pulls the readline low if the corresponding data signal is low, using the inverters' large NMOS transistors to reduce the fall time. The two read lines have opposite polarities, since they connect to opposite data polarities. During a write, both polarities of the data are driven into the cell through the writelines, with the writeline driven low overpowering the inverters' weak PMOS transistor, flipping the stored bit.

2.4.2 Operand Buses

Figure 2.8 shows the register-bank interface to the operand and result buses. The three operand buses precharge during phase 2, and bus drivers pull them low after the register read in phase 1, if necessary. Since domino logic is noninverting, the active-low read port drives an active-low operand bus. OperandA is the natural choice for the active-low bus, since an active-low operandC implies an active-low operandB, as operandB can duplicate operandC. The result bus cannot use precharging, as combinational logic in the ALU or multiplier drive the bus, possibly transitioning several times before settling on the final output.

Table 2.3 shows all control and data signals used by the register bank and operand bus drivers. Global control signals select whether operands come from the left or right register banks. The direction bit of the operandA and operandC instruction fields multiplexes

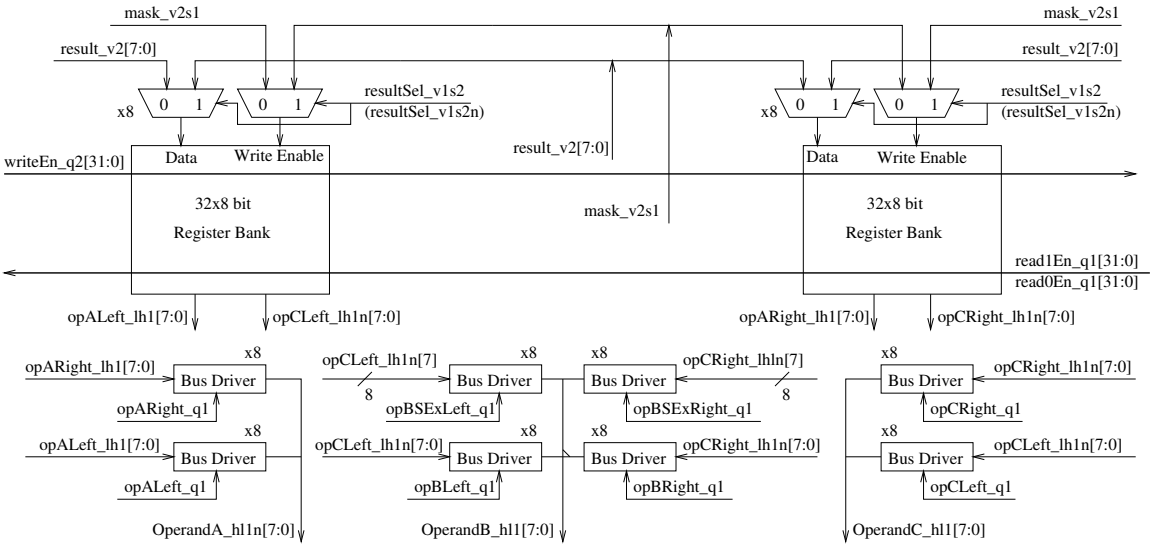

Figure 2.8: Register bank interface to operand and result buses.

Signal	Source	Type	Timing	Purpose
maskLeft_v1s2 maskRight_v1s2	Local	Input	S2	Mask flag from left or right to disable register write
resultLeft_v2[7:0] resultRight_v2[7:0]	Local	Input	V2	Result buses from left and right PE
resultSel_v1s2 resultSel_v1s2n	Global	Input	S2	Select result from left or right PE for register write
operandA_hl1n[7:0]	Local	Output	HL1	OperandA bus
operandB_hl1[7:0]	Local	Output	HL1	OperandB bus
operandC_hl1[7:0]	Local	Output	HL1	OperandC bus
opALeft_q1 opARight_q1	Global	Input	Q1	Select operandA from left or right register bank
opCLeft_q1 opCRight_q1	Global	Input	Q1	Select operandC from left or right register bank
opBLeft_q1 opBRight_q1	Global	Input	Q1	Optionally drive operandC onto operandB
opBSExLeft_q1 opBSExRight_q1	Global	Input	Q1	Optionally drive sign extended operandC onto operandB
read0En_q1[31:0] read1En_q1[31:0]	Global	Input	Q1	Decoded read addresses for operandA and operandC
writeEn_q2[31:0]	Global	Input	Q1	Decoded write address

Table 2.3: Register bank and operand bus driver external data and control signals.

between left and right register banks, qualified by phase 1 to prevent drive fights during phase 2 precharging. The operandB select instruction field, shown in Table 2.4, along with the direction bit for operandC, controls whether operandC, or the sign extension of operandC, drives operandB. If neither is selected, then operandB will be driven elsewhere in the PE. The register bank multiplexes results from left and right PEs based on the direction field of the destination register, and the corresponding mask flag disables writes if deasserted.

The bus drivers shown in Figure 2.8 are pulldowns consisting of two series nMOS transistors, one for control and one for data. Since register reads are time critical, transistor sizes were calculated based on the total bus load. To estimate the appropriate driver size, the total output capacitance on the driver, converted to an equivalent input transistor size and using the guidelines for stage ratios of gates, determined the driver size [14].

The critical path for register reads includes driving the sign extension of operandC onto the operandB bus. Since the most significant bit of register output drives all eight bits of operandB, it has eight times the load of other register output bits, requiring two extra stage-ratioed inverters to quickly drive the output load, as shown in Figure 2.7.

opB_v2[2:0]	Source
000	operandC
001	sign extension of operandC
010	SRAM MDR register
011	sign extension of SRAM MDR
100	Multiplier high byte register
101	sign extension of multiplier high byte
110	Bit Shifter
111	Instruction Immediate

Table 2.4: Instruction encoding for selecting the operandB bus source.

The result bus, driven by either the multiplier or the result multiplexer in the ALU, is also speed critical. For PEs at a chip or PE row boundary, the result must be written to a cross-chip register bank by the end of phase 2. In these cases, buffers isolate the local result bus from the globally transmitted signal, since the local PE result drivers cannot drive a global chip signal. The late arrival of the result bus from a cross-chip PE implies the writelines, shown in Figure 2.7, cannot precharge as the input may make several transitions before stabilizing. In contrast, the SRAM uses precharging for writes, requiring a valid result bus during all phase 2. The PE mask flag must be valid at the register bank during all phase 2 to prevent unwanted writes, so cross-chip PE mask flag transmission occurs during phase 1.

2.4.3 Problems and Improvements

Debugging of the Kestrel board revealed a result-bus glitch that occurred even when the Kestrel system idled. Latches preserve the result bus generated during phase 1 until the end of phase 2, so operand-bus precharging does not change the result. However, at the beginning of the next clock cycle, the result bus driver immediately drives a value based on the precharged operands onto the result bus until the register read completes and the ALU or multiplier calculation repeats. The delay is the register read time plus the propagation delay through the ALU or multiplier, and causes a glitch on the output pads, even when executing a nop instruction. Standby power consumption of the Kestrel system could be reduced by disabling the Kestrel chip output pads when the controller asserts a nop instruction.

The inverters at the end of each readline in Figure 2.7 should have a small nMOS

transistor and a large pMOS transistor, since the low-to-high transition is time critical, while the high-to-low transition only occurs during phase 2 precharging. While the nMOS transistors are only six lambda wide for the operandA and operandB inverters, they should be even smaller, and I completely missed the corresponding optimization to the operandC sign extension inverter chain, resulting in a large nMOS transistor at the output. A future design revision should shrink the pMOS and nMOS transistors for the second and third inverters, respectively, reducing the delay along what is clearly the critical path.

2.5 ALU

The 8-bit Kestrel ALU provides 32 arithmetic and logical operations, and supports multiprecision operations on both signed and unsigned numbers.

2.5.1 Architecture

Figure 2.9 shows the ALU architecture, designed by Jeff Hirschberg, with Table 2.5 listing each signal and its purpose. The carry-out of an ALU operation `aCout_v1` can be stored in a static latch, controlled by the `lc_v2s1` instruction bit, and fed into the carry-in of a subsequent operation. Alternatively, the carry-in can be taken directly from the instruction bit `ci_v2s1`. The Flag MUX selects between four possible signals, including the carry `aCout_v1`, the true sign `ats_v1`, the carry latch `cLatch_v2s1`, and zero. The `aFlag_v1n` signal routes to the flag selector that chooses a flag from each of the functional units, as discussed in Section 2.7. The true sign bit `ats_v1` is the sign bit produced on operands sign extended to 9-bits and the calculation performed on the extended operands, yielding the correct sign even when overflow occurs [10].

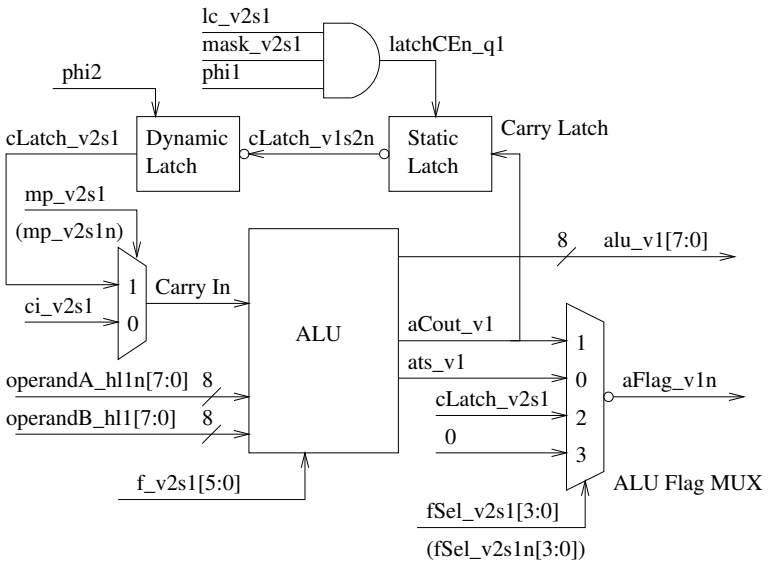

Figure 2.9: The ALU architecture, by Jeff Hirschberg, includes support for multiprecision operations and signed and unsigned numbers.

Signal	Source	Type	Timing	Purpose
mask_v2s1	Local	Input	S1	Local PE Enable
operandA_hl1n[7:0]	Local	Input	HL1	ALU operands
operandB_hl1[7:0]	Local	Input	HL1	
aFlag_v1n	Local	Output	V1	Output flag to flag selector
alu_v1[7:0]	Local	Output	V1	ALU result
cLatch_v2s1	Local	Internal	V2S1	Contents of carry latch
aCout_v1	Local	Internal	V1	ALU carry
ats_v1	Local	Internal	V1	ALU true sign
lc_v2s1	Global	Input	V1	Carry latch enable
mp_v2s1	Global	Input	V1	Select ALU carry-in
mp_v2s1n	Global	Input	V1	source
ci_v2s1	Global	Input	V1	Optional instruction carry-in
func_v2s1[5:0]	Global	Input	V1	ALU operation selector
fSel_v2s1[3:0]	Global	Input	V1	Flag MUX controls
fSel_v2s1n[3:0]	Global	Input	V1	

Table 2.5: Control and data signals for the ALU.

2.5.2 Circuit Design

The bit-slice ALU consists of a carry chain and logic for result bit calculation[1]. The carry chain uses generate (G) and kill (K) signals, each a 4-bit lookup table indexed by the bits of the operand. The result bit for each slice is the exclusive OR of the carry input and the propagate ($\mathsf{P} = \overline{\mathsf{G}} \wedge \overline{\mathsf{K}}$) signal.

The need to keep the pin count and instruction size low meant a full unencoded generate and kill lookup table of b-bits cost too many instruction bits. However, blindly omitting operations would not work, as operands are not symmetric, with operandA always a register and operandB coming from several sources. Asymmetric operations would tremendously handicap programmers and compiler writers if, for example, the ALU supported $A - B$ but not $B - A$. Figure 2.6 shows the resulting 5-bit encoding scheme, saving 3-bits from the unencoded version.

Figure 2.10 shows the circuit used to convert function codes to kill and generate signals. The design is primarily two 4-to-1 MUXs selecting between the function code bits based on the operand values. The MUXs use clocked inverters instead of transmis-

[1] Don Speck designed an optimized circuit for the ALU, and a function encoding that reduces the instruction bit count without losing important operations, so this discussion is included for completeness [5].

F Code	$\mathsf{F_4F_3} = 00$		$\mathsf{F_4F_3} = 01$		$\mathsf{F_4F_3} = 10$		$\mathsf{F_4F_3} = 11$	
$\mathsf{F_2F_1F_0}$	$\mathsf{C_{in}} = 0$	$\mathsf{C_{in}} = 1$	$\mathsf{C_{in}} = 0$	$\mathsf{C_{in}} = 1$	$\mathsf{C_{in}} = 0$	$\mathsf{C_{in}} = 1$	$\mathsf{C_{in}} = 0$	$\mathsf{C_{in}} = 1$
	Increment Functions		Increment Functions		Addition Functions		Decrement Functions	
000	-1	zero	$\overline{\mathsf{A} \wedge \mathsf{B}}$	$-(\mathsf{A} \wedge \mathsf{B})$	-1	zero	-1	zero
001	$\mathsf{A} \vee \mathsf{B}$	$(\mathsf{A} \vee \mathsf{B}) + 1$	$\mathsf{A} \oplus \mathsf{B}$	$(\mathsf{A} \oplus \mathsf{B}) + 1$	$\mathsf{A} + \mathsf{B}$	$\mathsf{A} + \mathsf{B} + 1$	$(\mathsf{A} \wedge \mathsf{B}) - 1$	$\mathsf{A} \wedge \mathsf{B}$
010	$\mathsf{A} \vee \overline{\mathsf{B}}$	$(\mathsf{A} \vee \overline{\mathsf{B}}) + 1$	$\overline{\mathsf{B}}$	$-\mathsf{B}$	$\mathsf{A} - \mathsf{B} - 1$	$\mathsf{A} - \mathsf{B}$	$(\mathsf{A} \wedge \overline{\mathsf{B}}) - 1$	$\mathsf{A} \wedge \overline{\mathsf{B}}$
011	A	$\mathsf{A} + 1$	$\mathsf{A} \wedge \overline{\mathsf{B}}$	$(\mathsf{A} \wedge \overline{\mathsf{B}}) + 1$	$\mathsf{A} + \mathsf{A}$	$\mathsf{A} + \mathsf{A} + 1$	$\mathsf{A} - 1$	A
100	$\overline{\mathsf{A}} \vee \mathsf{B}$	$(\overline{\mathsf{A}} \vee \mathsf{B}) + 1$	$\overline{\mathsf{A}}$	$-\mathsf{A}$	$\mathsf{B} - \mathsf{A} - 1$	$\mathsf{B} - \mathsf{A}$	$(\overline{\mathsf{A}} \wedge \mathsf{B}) - 1$	$\overline{\mathsf{A}} \wedge \mathsf{B}$
101	B	$\mathsf{B} + 1$	$\overline{\mathsf{A}} \wedge \mathsf{B}$	$(\overline{\mathsf{A}} \wedge \mathsf{B}) + 1$	$\mathsf{B} + \mathsf{B}$	$\mathsf{B} + \mathsf{B} + 1$	$\mathsf{B} - 1$	B
110	$\overline{\mathsf{A} \oplus \mathsf{B}}$	$-(\mathsf{A} \oplus \mathsf{B})$	$\overline{\mathsf{A} \vee \mathsf{B}}$	$-(\mathsf{A} \vee \mathsf{B})$			$(\mathsf{A} \oplus \mathsf{B}) - 1$	$\mathsf{A} \oplus \mathsf{B}$
111	$\mathsf{A} \wedge \mathsf{B}$	$(\mathsf{A} \wedge \mathsf{B}) + 1$	zero	1			$(\mathsf{A} \vee \mathsf{B}) - 1$	$\mathsf{A} \vee \mathsf{B}$

Table 2.6: Function encoding of ALU operations, by Don Speck [5].

sion gates to prevent placing an overwhelming load on the global function code signals `func_v2s1[5:0]`, and to reduce the transistor count compared to a combinational-gate implementation.

Figure 2.11 shows Don Speck's ALU circuit design. The carry chain has been optimized by factoring out those signals not on the critical path. The remainder of each bit-slice computes the exclusive OR of the carry-in and propagate signals. A NAND gate computes the carry-out of the entire operation, and the exclusive OR of the carry-out and the most significant bit's propagate computes the true sign.

Two-thirds of the ALU's bit-slice area consists of the kill and generate signal MUXs. For transistors on the critical path, nMOS transistor of 6 lambda, twice minimium size, are common. The complementary pMOS transistor are sized only slightly larger as the total ouput load is small, and the increased output capacitor reduces the effect of the increased pullup strength of double-sized pMOS transistors. Consequently, the ALU has asymmetric rise and fall times.

2.5.3 Problems and Improvements

Increasing the ALU's width to 16 bits would improve the performance of any program performing multiprecision arithmetic, including the sequence-analysis algorithms. However, the size of the register bank and operand buses would also need increased, so this would require a major PE redesign.

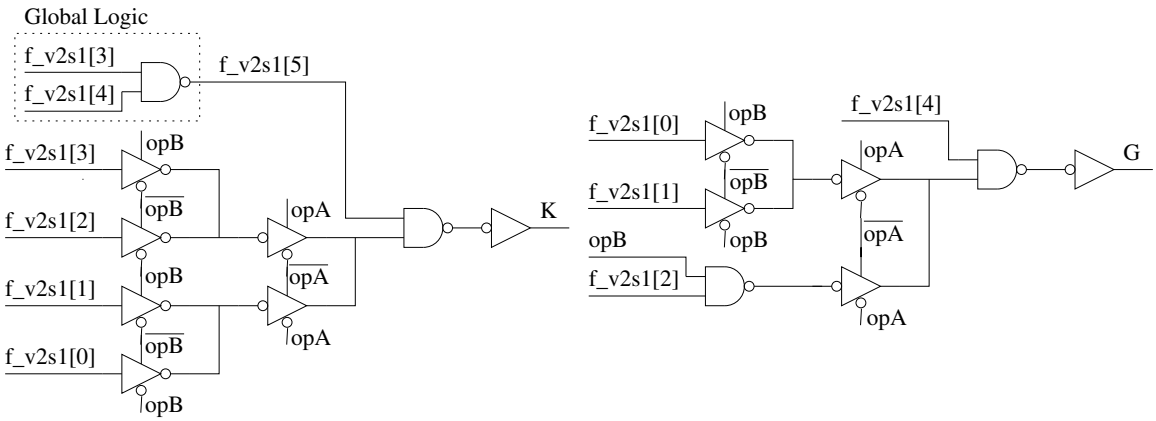

Figure 2.10: ALU circuit that converts the function codes to generate and kill signals, by Don Speck.

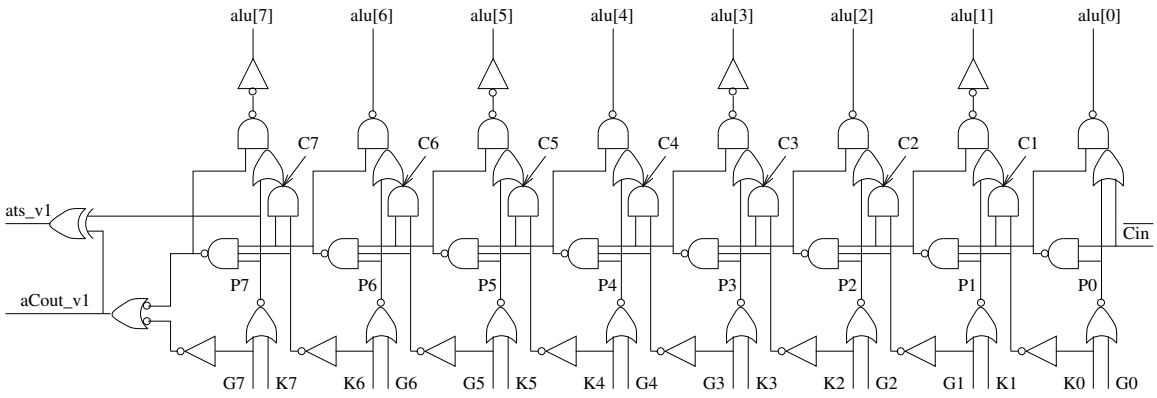

Figure 2.11: ALU circuit design, by Don Speck [5].

2.6 Comparator

The comparator subtracts the ALU result from operandC, supporting several comparison types by generating unsigned, signed, and modulo-256 result flags. The desired flag can then be used to select either the ALU result or operandC as the instruction result, an operation forming the core of many dynamic programming algorithms used in sequence comparison. The comparator also supports efficient multiprecision comparison and selection using a top-down approach, locking the result selection as soon as it finds two nonequal bytes.

2.6.1 Architecture

Figure 2.12 shows the comparator architecture, and Table 2.7 gives descriptions of the important signals. The subtractor block produces a carry `cCout_v1`, the true sign `cts_v1`, the most significant bit of the subtraction result `msb_v1`, used for modulo-256 comparisons, and an equal flag `eq_v1`. The carry flag MUX selects between the comparison flags and the minLatch, and the equal flag MUX selects between the equal flag and the eqLatch. The flags can then the used with the result selector to choose the overall instruction result based on the comparison, and/or save the flags into latches for use in a subsequent step of a multiprecision comparison.

2.6.2 Multiprecision Comparisons

The comparator supports top-down multiprecision comparison using the reset and nonreset comparator enable signals, `nrComp_v2s1` and `rComp_v2s1`, respectively. When an instruction performs a reset-enabled comparison, the equal and carry latches always save

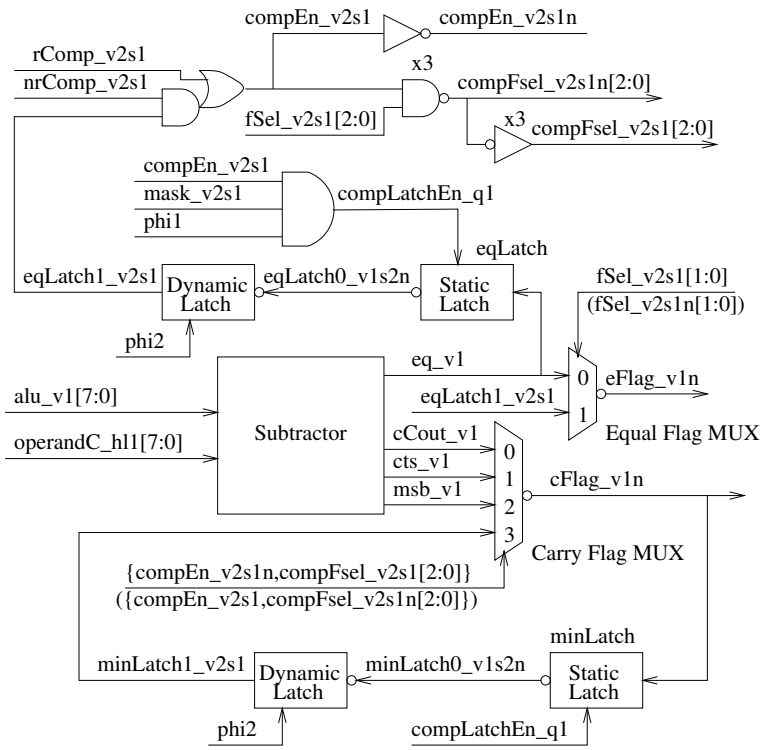

Figure 2.12: The comparator architecture, by Jeff Hirschberg, supports unsigned, signed, modulo-256, and top-down multiprecision comparisons.

the new flags. If an instruction asserts only the nonreset enable, then the latches only save the new flags if no nonequal bytes have been seen. If the instruction deasserts both enables, then no flag latching occurs. Table 2.8 summarizes the relationship between the enables and comparator operations.

As an example of a multiprecision minimize, consider selecting the minimum of two multiprecision numbers. The top-down multiprecision comparison begins with a reset-enabled comparison, latching the equal and comparison flags, with subsequent comparisons nonreset enabled. The comparison instructions use the carry `cCout_v1` or true sign `cts_v1` to control the result selector for the most significant byte, depending on whether the operands are unsigned or signed, respectively. All subsequent comparisons use the carry, since the

Signal	Source	Type	Timing	Purpose
mask_v2s1	Local	Input	S1	Local PE enable
alu_v1[7:0]	Local	Input	V1	Result from ALU
operandC_hl1[7:0]	Local	Input	hl1	operandC bus
cFlag_v1n	Local	Output	V1	Flag from carry Flag MUX
eFlag_v1n	Local	Output	V1	Flag from equal Flag MUX
eqLatch1_v2s1	Local	Internal	S1	Contents of eqLatch
minLatch1_v2s1	Local	Internal	S1	Contents of minLatch
compLatchEn_q1	Local	Internal	Q1	Latch enable
compEn_v2s1	Local	Internal	S1	Comparator enable
compEn_v2s1n	Local	Internal	S1	
compFsel_v2s1[2:0]	Local	Internal	V1	Flag MUX selectors qualified
compFsel_v2s1n[2:0]	Local	Internal	V1	by the comparator enable
nrComp_v2s1	Global	Input	V1	Controls for single and
rComp_v2s1	Global	Input	V1	multiprecision comparisons
fSel_v2s1n[1:0]	Global	Input	V1	Flag MUX controls
fSel_v2s1[2:0]	Global	Input	V1	

Table 2.7: Comparator control and data signals, including global controls and their locally qualified versions, as well as local data.

nrComp_v2s1	rComp_v2s1	compEn_v2s1	Action
0	0	0	Comparator always disabled
X	1	1	Comparator always enabled
1	0	eqLatch1_v2s1	Comparator enabled if no nonequal bytes have been seen

Table 2.8: Global reset and nonreset comparator enable signals (nrComp_v2s1 and rComp_v2s1, respectively) and their affect on the local comparator enable (compEn_v2s1).

sign bit only appears in the most significant byte.

While the comparator sees equal bytes, it remains enabled. The result selector's choice for the result is irrelevant since both bytes are identical. When the comparator encounters the first nonequal bytes, the eqLatch and the minLatch store the equal and carry flags a final time, and the comparator turns off on the remaining comparisons, as the comparator knows which operand is smaller. For the remaining comparisons, the comparator forces the minLatch as the result selector flag by qualifying the carry flag MUX select signals `fSel_v2s1[2:0]` with the comparator enable `compEn_v2s1`. Thus, once the comparator becomes disabled, the carry flag MUX always selects the minLatch as its output, forcing selection of the smaller number.

2.6.3 Circuit Design

Figure 2.13 shows the comparator's subtractor circuit and flag logic. The equal chain determines equality, and the carry chain, along with the kill and propagate for the most significant bit, control a full adder, shown in Figure 2.14, that calculates the carry-out `cCout_v1` and the most significant bit of the subtraction `msb_v1`. Fortunately, $\overline{G7} = 0$ and $K7$=1 is not possible statically, and the layout design tries to match the delays between $\overline{G7}$ and $K7$ to prevent excess dynamic power consumption in the full adder. The exclusive OR of `cCout_v1`, `alu_v1[7]` and `opC_hl1n[7]` computes the true sign `cts_v1`.

The optimized carry and equal chains evaluate in parallel one bit-slice behind the ALU, only slightly increasing the delay for an ALU/minimize or maximize operation. As shown in the floorplan in Figure 2.6, the comparator's bit-sliced segment is small, with most area used for multiprecision and flag-selection logic. Transistors along the carry and equal

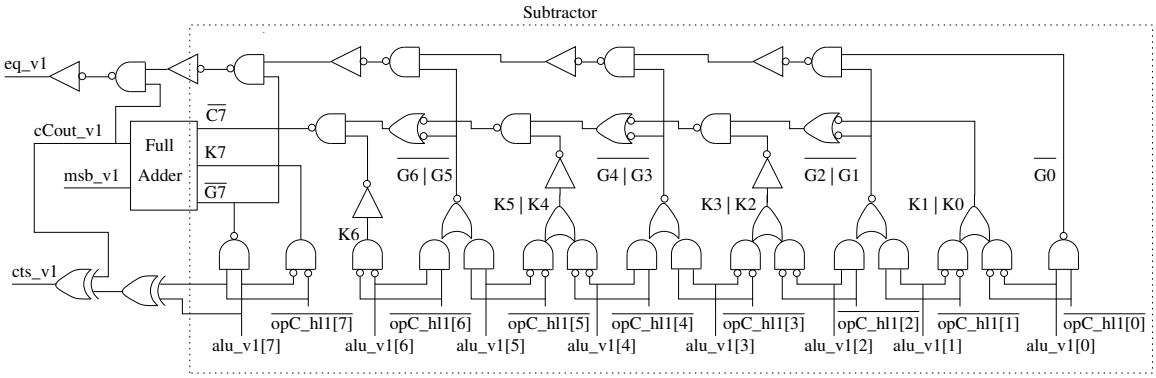

Figure 2.13: Circuit design for the comparator's subtractor, by Don Speck [5].

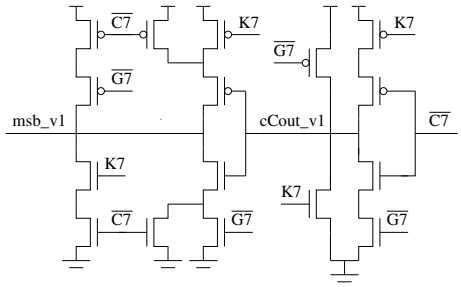

Figure 2.14: Circuit design for the comparator's full adder from Figure 2.13, by Don Speck.

chains have similar sizes to those in the ALU.

2.6.4 Problems and Improvements

The polarities of the true sign `cts_v1` and carry `cCout_v1` flags complicate comparison of signed multiprecision numbers. The true sign flag has the opposite polarity from the carry flag, since the true sign calculation exclusive ORs `cCout_v1`, `alu_v1[7]`, and `opC_hl1n[7]` instead of `opC_hl1[7]`. Consequently, when performing a signed multiprecision comparison, the operands must be swapped after the first comparison to invert the polarity of the carry flag and make it consistent with the true sign polarity used for the

most significant byte. The problem is easily fixed by replacing the `opC_hln[7]` exclusive OR input with `opC_hl1[7]`, as both are available in the layout. This change should be made even though it would require changes to existing programs.

2.7 Flag and Result Selectors

The flag selector chooses a flag bit from one of the functional units as part of the instruction result. The flag bit can be used to select between the ALU and operandC as the result, to process a nested conditional or can be packed into a bit vector for future use. Consequently, efficient conditional and bit packing operations require fast result flag generation.

2.7.1 Architecture

The flag selector's second stage chooses the flag from one of five flag MUXs, as shown in Figure 2.15 (a). The resulting flag is exclusive ORed with `finv_bar_v2s1` for optional inversion, then sent to the result selector, shown in Figure 2.15 (b). Table 2.9 lists the control and data signals for both the flag and result selectors. Table 2.10 lists the 32 flag choices and the corresponding instruction encoding `fb_v2[4:0]`. The `fb_v2[4:0]` instruction field codes not only the flag choice, but also comparator and mesh functionality. Selecting the reset or nonreset comparator flags sets the `rComp_v2s1` or `nrComp_v2s1` signals, respectively, and choosing a flag not from the comparator MUX disables the comparator. Section 2.9 describes the functionality of the bit-serial mesh network.

The instruction field `rm_v2[1:0]` chooses the result selection method, as shown in Table 2.11. The result selector can unconditionally choose either operandC or the ALU

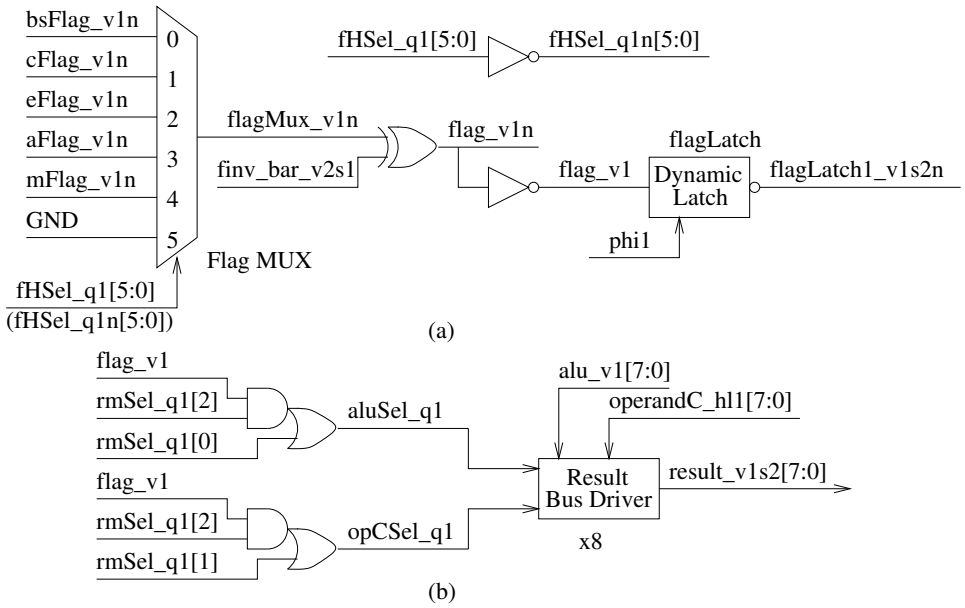

Figure 2.15: (a) Flag selector second stage and (b) result selector for ALU and operandC, designed by Jeff Hirschberg.

Signal	Source	Type	Timing	Purpose
bsFlag_v1	Local	Input	V1	Bit Shifter Flag MUX output
cFlag_v1	Local	Input	V1	Comparator Carry Flag MUX output
eFlag_v1	Local	Input	V1	Comparator Equal Flag MUX output
aFlag_v1	Local	Input	V1	ALU Flag MUX output
mFlag_v1	Local	Input	V1	Mesh Flag MUX Output
alu_v1n[7:0]	Local	Input	V1	ALU result
operandC_hl1[7:0]	Local	Input	HL1	OperandC bus
flag_v1n	Local	Output	V1S2	Final Flag
flagLatch1_v1s2n	Local	Output	V1S2	Latched Flag
result_v1s2[7:0]	Local	Output	V1S2	Result bus
rmSel_q1[2:0]	Global	Input	Q1	Result Selector control
fHSel_q1[5:0]	Global	Input	Q1	Second Stage Flag MUX control
fHSel_q1n[5:0]	Local	Internal	Q1	
finv_bar_v2s1	Global	Input	V1	Flag invert control

Table 2.9: Flag and result selector control and data signals.

fb_v2[4:0]	Selected Flag	fb_v2[4:0]	Selected Flag
Comparator MUX		Equal MUX	
00000	cCout_v1 (reset)	10000	eq_v1
00001	cts_v1 (reset)	10001	eqLatch1_v2s1
00010	msb_v1 (reset)	10010	Unused
00011	minLatch1_v2s1	10011	Unused
00100	cCout_v1 (nonreset)	Mesh MUX	
00101	cts_v1 (nonreset)	10100	+1 (get from west)
00110	msb_v1 (nonreset)	10101	-1 (get from east)
00111	minLatch1_v2s1	10110	+8 (get from north)
Bit Shifter MUX		10111	-8 (get from south)
01000	nor_v2s1	11000	+2 (get from west)
01001	bsLatch0_v2s1[0]	11001	-2 (get from east)
01010	bsLatch0_v2s1[7]	11010	+16 (get from north)
01011	wor_hl1	11011	-16 (get from south)
ALU MUX		11100	+4 (get from west)
01100	ats_v1	11101	-4 (get from east)
01101	aCout_v1	11110	+32 (get from north)
01110	cLatch_v2s1	11111	-32 (get from south)
01111	0		

Table 2.10: The flag choices and their corresponding instruction encodings fb_v2[4:0].

result, or choose conditionally based on the flag. An instruction indicates a multiply with ci_v2 = 1 and lc_v2 = 1, in inwhich case the result always comes from the multiplier. The rm_v2[1:0] instruction field also encodes the SRAM address computation method, as discussed in Section 2.11.

		Result Chosen	
rm_v2[1:0]	rmSel_q1[2:0]	flag = 0	flag = 1
00	001	alu_v1[7:0]	
01	001	alu_v1[7:0]	
10	010	operandC_hl1[7:0]	
11	100	operandC_hl1[7:0]	alu_v1[7:0]

Table 2.11: Result selection and instruction encoding rm_v2[1:0], overridden by the multiplier result when ci_v2 = 1 and lc_v2 = 1.

2.7.2 Implementation Issues

The two-stage flag selector avoids needing to routing 20 potential flag signals to a large multiplexer, keeping signal wires short and easing wiring congestion, already a problem due to the large number of global control signals. In total, the flag multiplexer uses 14 control signals, with six (`fHSel_q1[5:0]`) selecting the functional unit, and eight (`fSel_v2s1[3:0]` and `fSel_v2s1n[3:0]`) selecting the flag within a functional unit. The second stage selector's global control signals include only active-high versions, with local inverters producing the active low versions. While increasing the overall transistor count, the inverters are near minimum size, since they are not on the critical path. The local inverters also reduced the PE core height by 56 lambda, filling otherwise unused space forced by compacted global control signals in metal2.

2.7.3 Problems and Improvements

The `fHSel_q1[5:0]` control signals for the flag selector's second stage were originally qualified phase 1 to make the flag selector's output valid during phase 2. However, the flagLatch, shown in Figure 2.15 (a), also latches the flag at the end of phase 1, producing a signal valid during phase 2. The redundant flagLatch should be removed as the `Q1` flag selects already produces a `V1S2` flag.

The two unused flag-bus encodings should select the mask flags from the adjacent PEs, allowing efficient movement of a sentinel through the array. The mask flags are ANDed with the `force_v2` instruction bit, so mask flags selection would only makes sense in conditional code. Selecting the mask flags to the flag bus would require only minor layout modifications, as both flags are already available in each PE, and there is space available

in the PE core for an additional flag selector MUX.

2.8 Bit Shifter and Mask Flag Generator

The bit shifter servers as both a logical shifter and, in conjunction with the mask flag generator, a nested condition stack. It performs single-bit left and right shifts on an 8-bit register, with bits moving to and from the register through the flag. Packing flag bits is an important operation for dynamic programming algorithms that select between several choices, save the resulting decision, and later perform postprocessing to recover the sequence of decisions that yield an optimal solution.

When used in conjunction with the mask flag generator, the bit shifter can efficiently process nested conditionals, allowing an individual PE to perform operations conditionally based on local data. Local conditionals add great flexibility over the basic SIMD model where every PE always executes every instruction.

2.8.1 Architecture

Figure 2.16 shows the bit shifter architecture, essentially a group of static latches connected through MUXs. The MUX select control serves as the function code, controlling data movement through the latches. The most significant bit has additional inputs used during conditional processing. Table 2.12 lists the important bit shifter control and data signals and gives a brief description of each. As an example, `bsSel_v1s2[2:0]` set to 001 and 000 performs a left and right shift, respectively, with the current flag shifted in.

Figure 2.17 shows the architecture of the mask-flag generator that controls the local PE enable, and Table 2.13 gives a short description of the noteworthy signals. The

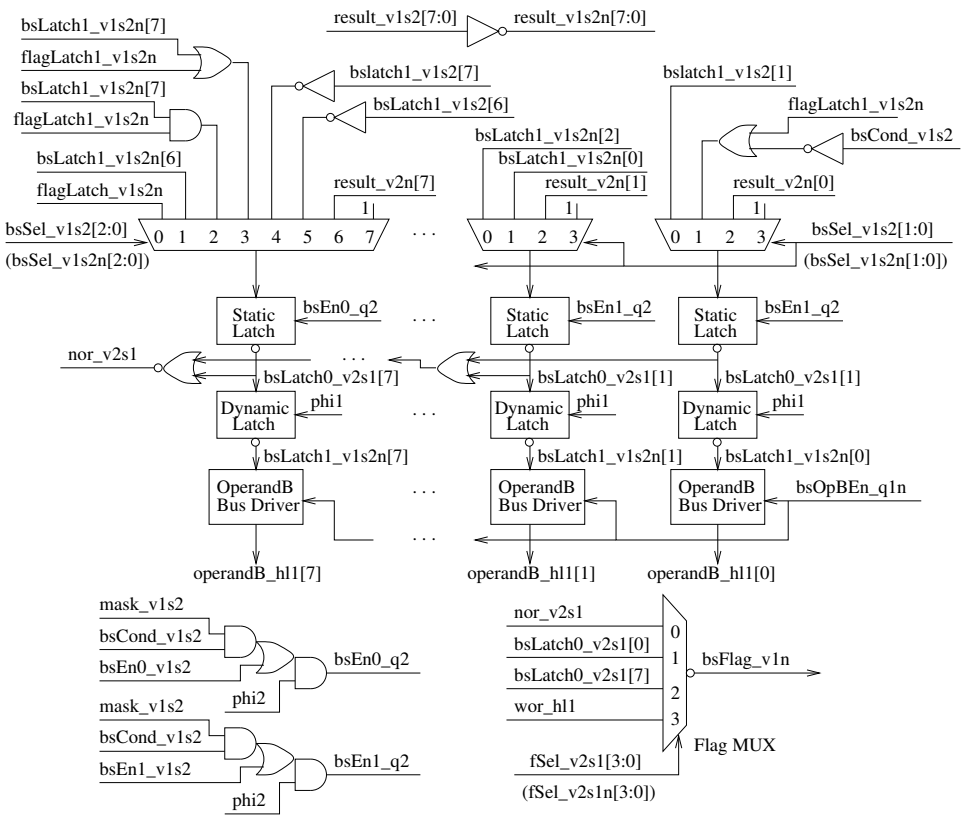

Figure 2.16: Bit shifter architecture, essentially a group of static latches connected through MUXs. The MUX select signals provide the function code, controlling data movement through the latches. Designed by Jeff Hirschberg.

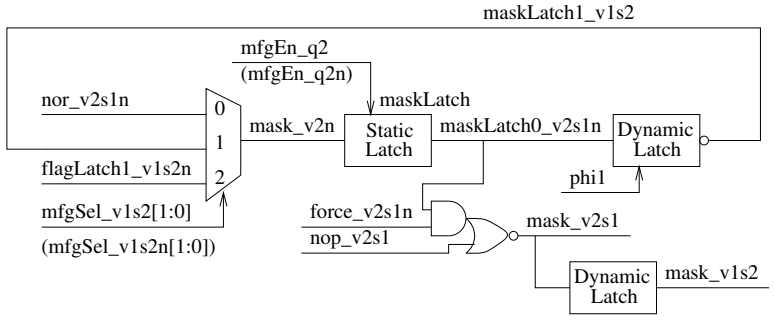

Figure 2.17: Mask flag generator architecture that allows efficient processing of nested conditionals when used with the bit shifter. Designed by Jeff Hirschberg.

Signal	Source	Type	Timing	Purpose
mask_v1s2	Local	Input	S2	Local PE enable
flagLatch1_v1s2n	Local	Input	V2	Instruction result flag
result_v1s2[7:0]	Local	Input	V2	Instruction result bus
bsFlag_v1	Local	Output	V1	Flag MUX output
nor_v2s1n	Local	Output	V2S1	NOR of all bit shifter
operandB_hl1[7:0]	Local	Output	HL1	operandB bus
bsOpBEn_q1n	Global	Input	Q1	Write BS contents to operandB
bsEn0_v1s2	Global	Input	S2	Latch enable for MSB
bsEn1_v1s2	Global	Input	S2	Latch enable for all non-MSB bits
bsCond_v1s2	Global	Input	S2	Conditional operation
bsSel_v1s2[2:0]	Global	Input	V2	Bit Shifter MUX selector
bsSel_v1s2n[2:0]	Global	Input	V2	
fSel_v2s1[3:0]	Global	Input	V1	Flag MUX selector
fSel_v2s1n[3:0]	Global	Input	V1	

Table 2.12: Explanation of bit shifter control and data signals from Figure 2.16.

local PE enable, called the mask flag, can be selected from the NOR of the bit shifter, the flag, or the inverse of the previous mask. The mask is also qualified by two global signals, the force and nop signals, `nop_v2s1` and `force_v2s1`, respectively. The force instruction bit indicates whether instructions should be executed conditionally or unconditionally. If force is asserted, then instructions always execute, but if deasserted instructions only execute in PEs with the local enable asserted. Broadcast from the Kestrel array controller, the nop signal disables all PEs, even overriding the force bit. The array controller uses the nop to pause instruction execution to handle interrupt conditions that require host processing, such as an empty or full data queue.

2.8.2 Nested Conditional Processing

When operating together, the bit shifter and mask-flag generator efficiently handle conditional instruction execution. This is especially important on a SIMD machine where

Signal	Source	Type	Timing	Purpose
nor_v2s1n	Local	Input	V2	NOR of bit shifter
flagLatch_v1s2n	Local	Input	V2	Instruction result flag
maskLatch0_v2s1n	Local	Output	V2S1	PE mask flag
mask_v2s1	Local	Output	V1	
mask_v1s2	Local	Output	V1S2	
force_v2s1n	Global	Input	V1	Force mask asserted
nop_v2s1	Global	Input	V1	Deassert mask (overrides force)
mfgEn_q2	Global	Input	Q2	Enable change to mask flag
mfgEn_q2n	Global	Input	Q2	
mfgSel_v1s2[1:0]	Global	Input	V2	Mask flag selection control
mfgSel_v1s2n[1:0]	Global	Input	V2	

Table 2.13: Explanation of mask flag generator control and data signals from Figure 2.17.

every clause of a conditional must be broadcast to every PE. Conditionals work by setting the mask to the NOR of the 8 bits of the bit shifter register. Conditionals are true when zero, so when all bits are zero, the bit shifter sets the mask to one, enabling the PE. If a conditional evaluation pushes a one onto the bit shifter, then the mask becomes zero, disabling the PE. When an operation shifts a one off the bit shifter, the PE will enable if all eight resulting bits are zero.

Table 2.14 lists the bit shifter and mask-flag operations, along with the assembly language mnemonic and values for the important bit shifter control signals in Figure 2.16. The `bsSel_v1s2[2:0]` signal controls the MUXs for the bit shifter latch inputs, and the `bsCond_v1s2`, `bsEn0_v1s2`, and `bsEn1_v1s2` control the latch enables. `bsEn0_v1s2` and `bsEn1_v1s2` control unconditional enables for the most significant bit and bits zero through six, respectively. The `bsCond_v1s2` control signal enables all eight latches if the PE asserts its mask. Operations that only affect the most significant bit have `bsEn0_v1s2` and `bsEn1_v1s2` set to one and zero, respectively. Refer to the Kestrel assembly language pro-

bit_v2[3:0]	Mask Flag	Bit Shifter Operation				
		Mnemonic	bsSel	bsCond	bsEn0	bsEn1
0000	unchanged		XXX	0	0	0
0001	! maskLatch1_v1s2	bsnotmask	XXX	0	0	0
0010	flag_v1	bsflagmask	XXX	0	0	0
0011	unchanged	bscondleft	101	1	0	0
0100	nor_v2s1	bsor	010	0	1	0
0101	nor_v2s1	bsand	011	0	1	0
0110	nor_v2s1	bsnot	100	0	1	0
0111	nor_v2s1	bsset	000	0	1	0
1000	unchanged	bsclear	111	0	1	1
1001	nor_v2s1	bsclearm	111	0	1	1
1010	nor_v2s1	bspopnot	001	0	1	1
1011	nor_v2s1	bspop	101	0	1	1
1100	nor_v2s1	bslatch	110	0	1	1
1101	unchanged	bscondlatch	110	1	0	0
1110	nor_v2s1	bspush	000	0	1	1
1111	unchanged	bscondright	000	1	0	0

Table 2.14: Signal encodings for the bit shifter and mask flag generator.

grammer's manual for more details about these operations [11].

2.8.3 Implementation Issues

The bit-shifter design minimizes power consumption rather than optimizes speed, since the bit shifter operates during phase 2, when the critical path is the cross-chip register write. Consequently, all NMOS transistor are 4 lambda, PMOS transistors 6 lambda wide, with the MUXs implemented using transmission gates. Figure 2.6 shows the location of the bit shifter relative to the other functional units. The bit shifter's flag MUX shares the flag selector control lines `fSel_v2s1[3:0]` and `fSel_v2s1n[3:0]`, and aligns with the other flag MUXs, as shown in Figure 2.6. The mask flag fills otherwise unused space, caused by global control routing, adjacent the result selector driver control.

2.8.4 Problems and Improvements

The bit shifter operates most efficiently as a condition stack, and does not perform arbitrary shifts. Augmenting the PE with a barrel shifter would improve the performance of algorithms requiring variable multi-bit shifts, such as when aligning mantissas during floating-point addition.

2.9 Wired-OR and Mesh Interconnections

The wired-OR and Mesh provide global interconnections between PEs on the same 64-PE Kestrel chip. The wired-OR provides the only conditional instruction sequencing based on conditions inside the PEs. This differs from the local conditionals handled by the bit shifter, where a PE locally decides if an instruction should be performed while the overall instruction stream remains unchanged. With the wired-OR, the PE array signals the instruction sequencer for a branch, potentially causing the broadcast of different instructions. The Mesh provides bit-serial communications between between PEs powers of two apart, providing efficient log-reduction operations.

2.9.1 Wired-OR

Figure 2.18 (a) shows the wired-OR connection for one PE, with Table 2.15 giving brief signal descriptions. A precharged wire `wor_hl1n` implements the wired-OR as a wired-NOR between PEs on the same chip, with precharging during phase 2 and evaluation during phase 1. If an active PE (with `maskLatch0_v2s1n` = 0) has a one in the most significant bit of its bit shifter, then it pulls the wired-NOR signal low. The `force_v2` instruction bit that enables all PEs, regardless of the local mask flag, does not affect whether a PE participates

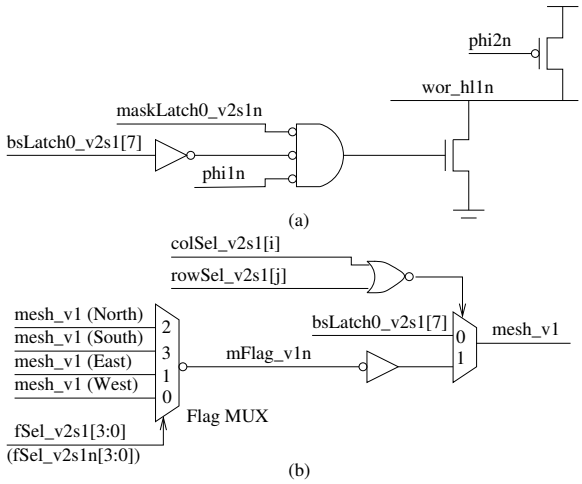

Figure 2.18: (a) Wired-OR and (b) Mesh connection for each PE.

in the wired-OR.

A pair of cross-coupled pulldowns isolates each row of 8 PEs from the other rows. While this decreases noise immunity, it is essential to prevent a single PE from having to discharge a wire that runs to every PE across the entire chip. A dynamic latch creates a `V1S2` version of the wired-OR signal for off-chip communication with the instruction sequencer during phase 2. The instruction sequencer then ORs the wired-OR signal from each chip to produce the wired-OR for the entire array.

2.9.2 Bit-Serial Mesh

The mesh allows communications between the bit shifter of PEs powers of two apart on the same chip. Each 64-PE chip organizes into eight rows of eight PEs, with each PE connecting to the neighboring +1 and -1 PEs in the same row and the +8 and -8 PEs in the adjacent rows. A PE at the beginning or end of a row connects to the last PE of the previous row or the first PE of the next row, respectively. Figure 2.18 (b) shows each PEs mesh connection, allowing communications with the four neighboring processors. The

Signal	Source	Type	Timing	Purpose
Wired-OR Control and Data Signals				
maskLatch0_v2s1n	Local	Input	S1	Local PE Enable
bsLatch0_v2s1[7]	Local	Input	S1	MSB of Bit Shifter
wor_hl1n	Global	Output	HL1	Global Wired-OR signal
Mesh Control and Data Signals				
rowSel_v2s1[i]	Global	Input	V1	Mesh Row Select
colSel_v2s1[j]	Global	Input	V1	Mesh Column Select
mesh_v1	Global	Output	V1	Mesh Connection to adjacent PEs
mesh_v1 (North)	Global	Input	V1	Mesh Connections from the
mesh_v1 (South)	Global	Input	V1	four adjacent PEs
mesh_v1 (East)	Global	Input	V1	
mesh_v1 (West)	Global	Input	V1	
fSel_v2s1[3:0]	Global	Input	V1	Flag MUX selector
fSel_v2s1n[3:0]	Global	Input	V1	

Table 2.15: Wired-OR and Mesh control and data signals for the design in Figure 2.18.

PE can either transmit the most significant bit of its bit shifter, or the bit received from a neighboring processor.

Figure 2.19 depicts the communication patterns, with symmetric data movement in the other directions. The PE at column i and row j transmits its bit shifter if `colSel_v2s1[i]` or `rowSel_v2s1[j]` is a one, and copies its output from an adjacent PE if both are zero. Table 2.16 shows the different values for the row, column, and flag selection bits for each of the 16 mesh communication patterns. PEs from Figure 2.19 with arrows passing through them have both row and column selects of zero, whereas PEs sending data have either row or column selects of one.

2.9.3 Problems and Improvements

Using the bit shifter as a condition stack is incompatible with wired-OR operations. The wired-OR is asserted if any active PE has a one in its bit shifter's most significant bit,

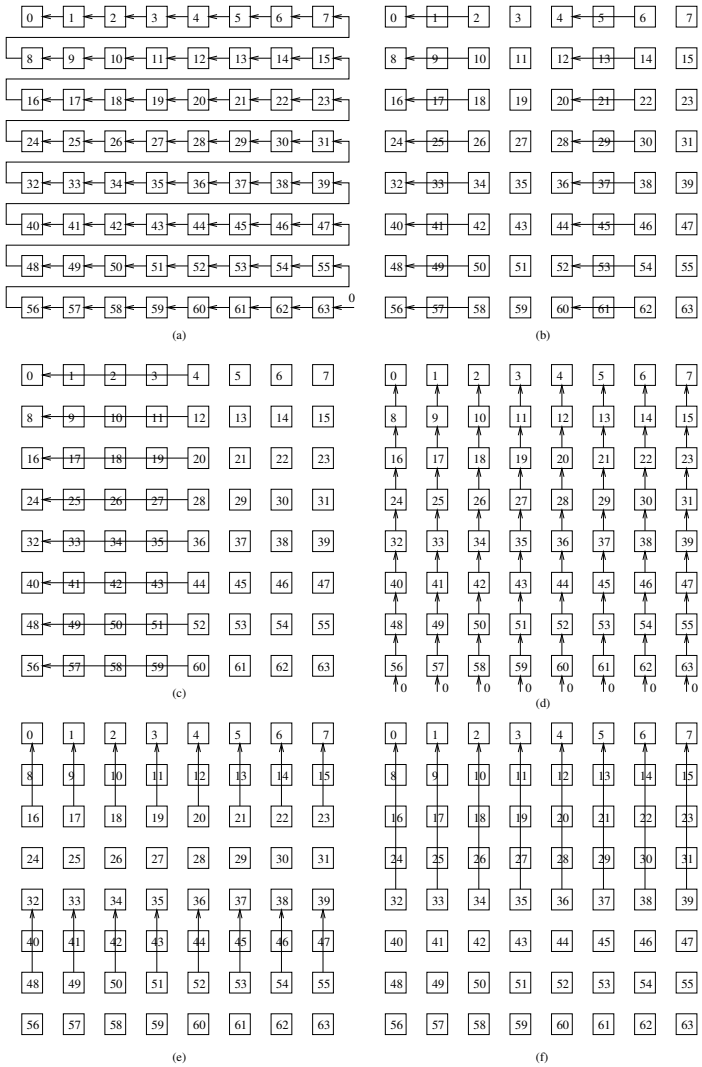

Figure 2.19: Mesh patterns for (a-c) east-to-west and (d-f) south-to-north communications used for log reduction; the west-to-east and north-and to-south commuications patterns are symmetric. (a) All processors copy data from the next processor in the same row, but log-reduction operations use only every other connection. (b) Every other column copies data from second next processor in the same row with only alternating pairs participating. (c) The first processor in each column receives data from the fourth. (d-f) The south-to-north patterns are symmetric to the east-to-west patterns.

colSel_v2s1[7:0]	rowSel_v2s1[7:0]	fSel_v2s1[3:0]	fb_v2[4:0]	Description
00000000	11111111	0001	10100	+1 (get from west)
00000000	11111111	0010	10101	-1 (get from east)
11111111	00000000	0100	10110	+8 (get from north)
11111111	00000000	1000	10111	-8 (get from south)
00000000	00010001	0001	11000	+2 (get from west)
00000000	01000100	0010	11001	-2 (get from east)
00010001	00000000	0100	11010	+16 (get from north)
01000100	00000000	1000	11011	-16 (get from south)
00000000	00000001	0001	11100	+4 (get from west)
00000000	00010000	0010	11101	-4 (get from east)
00000001	00000000	0100	11110	+32 (get from north)
00010000	00000000	1000	11111	-32 (get from south)

Table 2.16: After global logic decodes the `fb_v2[4:0]` instruction field to produce the row and column selects, the PE at column i and row j transmits its bit shifter if `colSel_v2s1[i]` or `rowSel_v2s1[j]` is a one, and copies its output from an adjacent PE if both are zero as shown in Figure 2.19.

whereas normal nested-conditional operations disable the PE when a program pushes a one onto the bit shifter. To overcome this, programs must conditionally shift the ORed bit onto the bit shifter, leaving the mask unchanged. Changing the polarity of the ORed signal would eliminate this complication, but would require changes to existing programs.

2.10 Multiplier

The 8-bit multiplier adds considerable general purpose computing power to the Kestrel architecture. The multiplier hardware supports both signed and unsigned operands, treating the 8-bit operands as 9-bit 2's complement numbers. It also supports multiprecision multiplication through the addition of two unsigned 8-bit operands to the product from a register and from the most-significant byte of the previous multiply. Modified-Booth recoding speeds product calculation by consuming two multiplier bits per partial product, producing the least-significant byte as the multiply instruction result, with the most-significant byte saved for use by a subsequent instruction.

2.10.1 Architecture

The multiplier computes the product of operandA and operandB, optionally adding in operandC and the most-significant byte of the previous product, as shown in Figure 2.20, with Table 2.17 giving brief signal descriptions. The product's least-significant byte drives the result bus at the end of phase 1, and the multHi latch saves the most-significant byte at the end of phase 2. A subsequent instruction can use the multHi latch or its sign extension as operandB, or add it to a later product. The multiplier and multiplicand, operandA and operandB, respectively, can independently be signed or unsigned, but the two add-in operands are always unsigned. Since a multiply takes a full clock cycle, dynamic latches hold operands stable during phase 2 during operand buses recharging, and the multHi latch becomes transparent.

To support both signed and unsigned multipliers and multiplicands using Booth's algorithm, which automatically treats operands as signed, the multiplier converts each

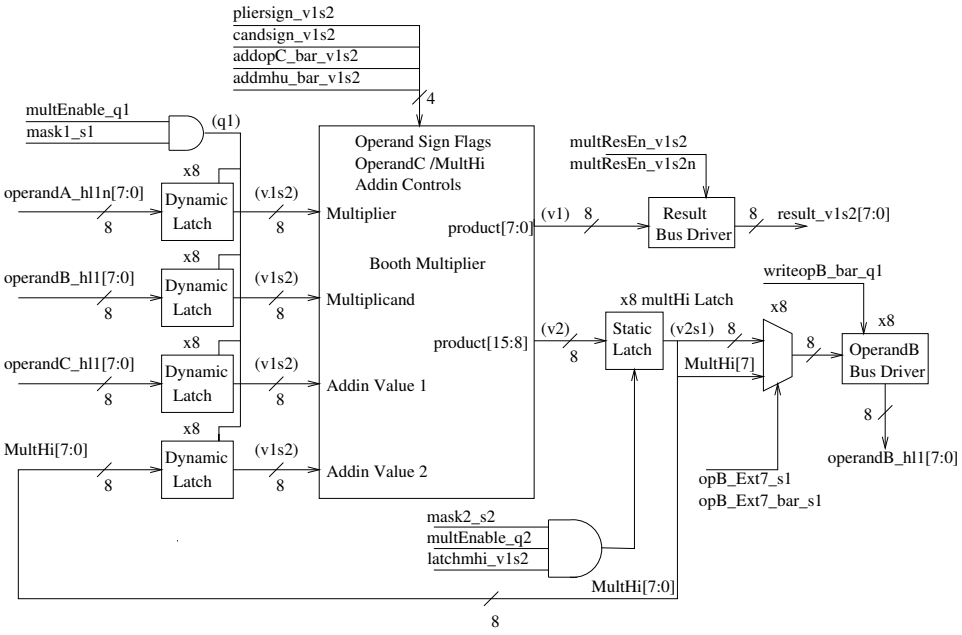

Figure 2.20: Multiplier interface to the Kestrel PE.

Multiplier Control and Data Signals				
Signal	Source	Type	Timing	Purpose
multEnable_q1	Local	Input	Q1	Phase 1 multiply enable
multEnable_q2	Local	Input	Q2	Phase 2 multiply enable
mask1_s1	Local	Input	S1	Phase 1 PE enable
mask2_s2	Local	Input	S2	Phase 2 PE enable
operandA_hl1n[7:0]	Local	Input	HL1	Multiplier
operandC_hl1[7:0]	Local	Input	HL1	Add-in operand
operandB_hl1[7:0]	Local	In/Out	HL1	multiplicand/multHi Out
result_v1s2[7:0]	Local	Output	V1S2	Result Bus for MultLo
latchmhi_s2	Global	Input	S2	MultHi latch enable
addopC_bar_v1s2	Global	Input	V1S2	Add operandC to product
candsign_v1s2	Global	Input	V1S2	Sign of multiplicand
pliersign_v1s2	Global	Input	V1S2	Sign of multiplier
addmhi_bar_v1s2	Global	Input	V1S2	Add multHi latch to product
writeopB_bar_q1	Global	Input	Q1	Save top byte of product
opB_Ext7_s1	Global	Input	S1	Drive multHi latch
opB_Ext7_bar_s1	Global	Input	S1	onto operandB bus
multResEn_v1s2	Global	Input	V1S2	Drive MultLo onto
multResEn_v1s2n	Global	Input	V1S2	Result Bus

Table 2.17: Multiplier control and data signals.

operand to a 9-bit 2's complement number. If an instruction selects an unsigned operand, then the multiplier sets the most-significant bit to zero, otherwise, it duplicates the sign bit.

2.10.2 Booth Multiplication

Booth's multiplication algorithm accelerates signed number multiplication by reducing the number of partial products. It examines several bits of the multiplier and computes a partial product based on some multiple of the multiplicand. However, instead of multiplying the subset of multiplier bits by the multiplicand to produce a partial product, Booth's algorithm relates the bits to sequences of ones and zeros using the identity

$$\sum_{n=i}^{j} 2^n = 2^{j+1} - 2^i \ .$$

Thus, Booth's algorithm recodes multiplier bits into terms representing the beginning and end of sequences of ones. For the Kestrel multiplier's 2-bit Booth recoding, partial products are multiplicand factors of -2, -1, 0, 1, or 2, whereas a nonredundant radix-4 encoding and can produce factors of 0, 1, 2, or 3. Booth recoding results in faster designs because the -1 and -2 terms are easily done with negations and shifts, whereas the factor of 3 requires both a shift and an add.

As an example, consider the straightforward multiplication of two 9-bit unsigned numbers, as shown in Figure 2.21 (a), where, for each multiplier bit, a partial product term consisting of the multiplicand times the corresponding multiplier bit, shifted the appropriate number of places, adds into the final product. Next, consider a Booth recoded multiply that consumes two multiplier bits per partial product, as shown in Figure 2.21 (b). Since a 2-bit

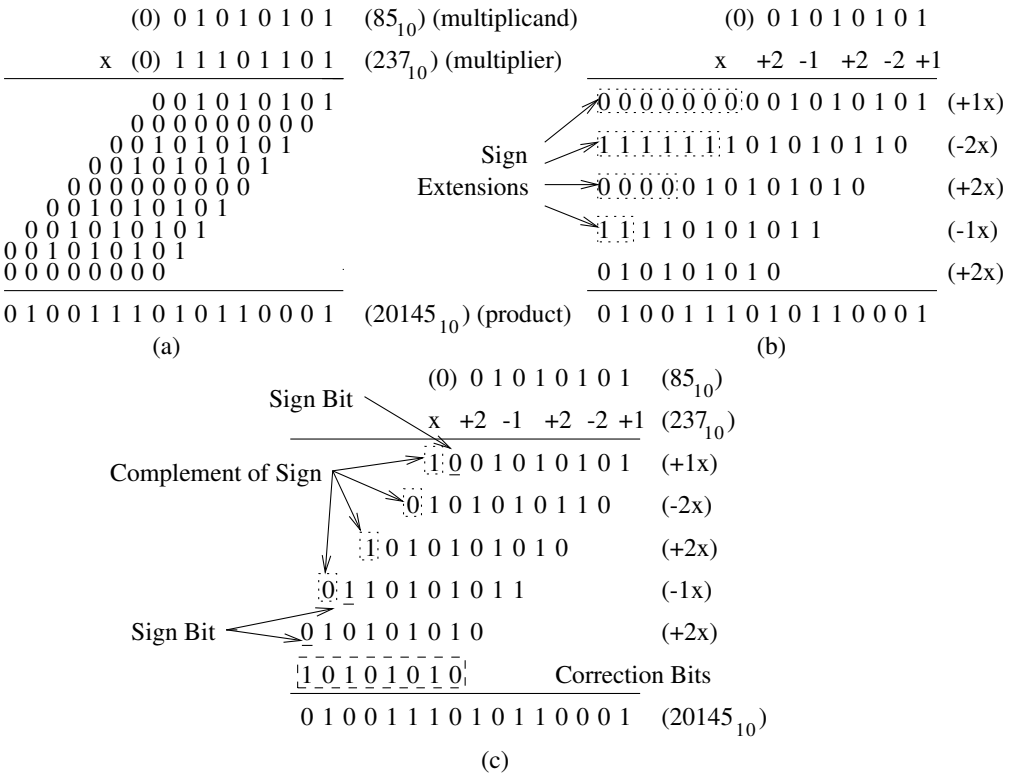

Figure 2.21: (a) Straightforward multiplication of two 9-bit unsigned numbers, (b) multiplication using 2-bit Booth recoding with sign extension bits, and (c) multiplication using modified-Booth multiplication, eliminating the sign extension bits.

encoding of a 9-bit number leaves one unpaired bit, the first partial product consumes only one multiplier bit M_0, and can be only zero or one times the multiplicand. The first bit pair M_2M_1 behaves differently from subsequent pairs because $M_0 = 1$ does not represent the beginning of a sequence. Table 2.18 (a) shows the recoding for the first bit pair M_2M_1, where $M_2 = 1$ indicates the beginning of a sequence of ones, resulting in a negative partial product. Subsequent bit pairs $M_{i+1}M_i$ look for both the beginning and end of sequences, as shown in Table 2.18 (b), using the preceding bit M_{i-1} to check for an ongoing sequence of ones. Determination of the multiplicand factor interprets $M_{i+1}M_i$ as a 2-bit 2's complement number plus M_{i-1} as a carry-in.

Multiplier		
M_2	M_1	Partial Product
0	0	$0 * \text{multiplicand}$
0	1	$+1 * \text{multiplicand}$
1	0	$-2 * \text{multiplicand}$
1	1	$-1 * \text{multiplicand}$

(a)

Multiplier			
M_{i+1}	M_i	M_{i-1}	Partial Product
0	0	0	$0 * 2^i * \text{multiplicand}$
0	0	1	$+1 * 2^i * \text{multiplicand}$
0	1	0	$+1 * 2^i * \text{multiplicand}$
0	1	1	$+2 * 2^i * \text{multiplicand}$
1	0	0	$-2 * 2^i * \text{multiplicand}$
1	0	1	$-1 * 2^i * \text{multiplicand}$
1	1	0	$-1 * 2^i * \text{multiplicand}$
1	1	1	$0 * 2^i * \text{multiplicand}$

(b)

Table 2.18: (a) Recoding for the multiplier's first bit pair M_2M_1, and (b) the recoding for bit pairs M_4M_3, M_6M_5, and M_8M_7.

2.10.3 Modified-Booth Multiplication

While the multiplication in Figure 2.21 (b) reduces the number of partial products, the sign extensions unnecessarily increases the number of adders. Modified-Booth's algorithm eliminates the sign extension, requiring only ten adders per partial product for 9-bit operands. Consider an i-bit number sign extended to n places:

$$A = a_{i-1} \cdots a_{i-1} a_{i-1} a_{i-2} \cdots a_1 a_0 \ .$$

Next, make the ith bit the complement of the sign bit a_{i-1}, and set all the bits above the ith equal to zero, giving

$$A' = 0 \cdots 0 \overline{a}_{i-1} a_{i-1} \cdots a_1 a_0 \ .$$

For the case when $a_{i-1} = 1$

$$A' + \sum_{k=i}^{n-1} 2^k = A \ ,$$

$$A' + 2^n - 2^i = A \ ,$$

and performing the addition modulo 2^n

$$A' - 2^i = A \ .$$

For the case when $a_{i-1} = 0$

$$A' - 2^i = A \ ,$$

showing that for both cases that prepending the sign bit's complement to the partial product just before the sign bit and subtracting a constant correction eliminates the need for sign extension. To compute the constant correction for the Booth recoding used for the multiplication in Figure 2.21 (b), 2^i must be subtracted from the result at each inverted sign bit position i:

$$P' - 2^{14} - 2^{12} - 2^{10} - 2^9 = P \ ,$$

$$P' - (0101011000000000)_2 = P \ ,$$

$$P' + (1010101000000000)_2 = P \ ,$$

$$P' + 2^{15} + 2^{13} + 2^{11} + 2^9 = P \ .$$

Figure 2.21 (c) shows the multiplication of two numbers using modified-Booth recoding with the added correction term.

2.10.4 Implementation Details

Figure 2.22 shows the overall organization of Kestrel's modified-Booth multiplier. Each row includes ten adders, with the S- and 8 cells simplified because of fewer inputs and hardwired correction bits. The first row consists of full adders to add the partial product, multiply accumulate, and operandC. A carry-propagate adder completes partial product addition, producing the lower eight bits by the end of phase 1, and the upper eight bits by the end of phase 2.

Each adder row from Figure 2.22 calculates a partial product based on recoded signals from bits of the multiplier, with Figure 2.23 showing the recoding circuitry. The first row computes 0 or 1 times the multiplicand, so a NAND suffices. The second row produces only a subset of multiplicand factors, and its partial product circuitry uses multiplier bits M_1 and M_2 directly. The remaining rows can produce all multiplicand factors, and use special recoding and partial-product selection circuits shown in Figure 2.24 and Figure 2.23 (c), respectively. For each row that can produce a negative partial product, the recoder's negate signal connects to the carry-in of the first adder to get the +1 in the 2's complement negation.

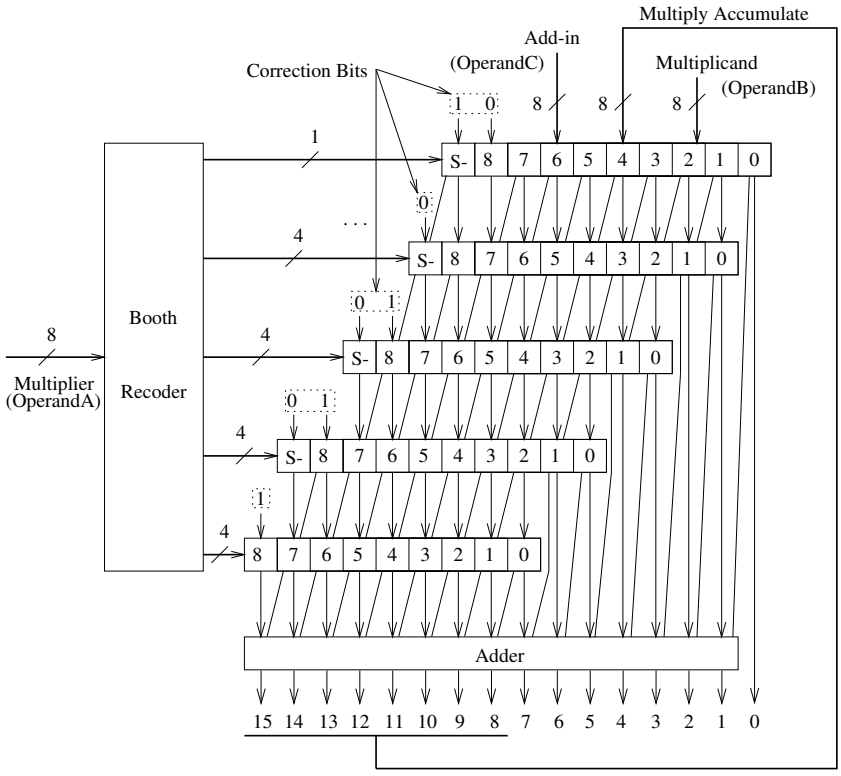

Figure 2.22: Organization of the modified-Booth multiplier.

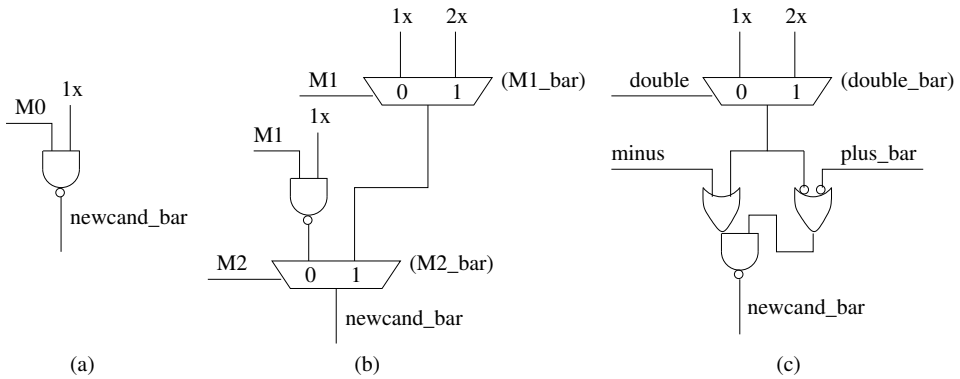

Figure 2.23: Partial-product computation circuitry for (a) the first row, (b) the second, and (c) the third, fourth, and fifth rows. By Don Speck.

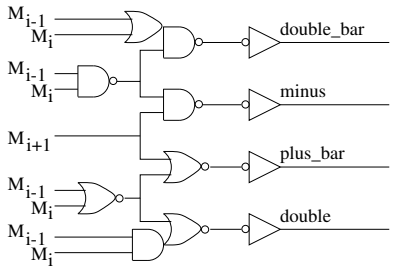

Figure 2.24: Bit-pair recoder circuitry for the third, fourth, and fifth rows. By Don Speck.

2.10.5 Layout Design

Figure 2.25 shows the multiplier floorplan. As indicated in the PE floorplan in Figure 2.5, the operandB and result buses, along with the SRAM address, route across the multiplier to the SRAM in metal2. Each row contains adders and partial product calculation circuitry, separated by metal1 routing of recoded multiplier signals, and operandA and result buses route to circuitry at the top and bottom, respectively. Dynamic latches hold the three operand buses and multHi stable during phase 2, and broadcast the multiplicand (operandB) to each row for partial product calculations. The design attempts to balance speed and power consumption by optimizing the least-significant byte calculation, driven onto the result bus by the end of phase 1, and slowing the most-significant byte calculation, latched locally at the end of phase 2.

2.10.6 Problems and Improvements

To reduce power consumption, the multiplier does not latch new inputs if an instruction does not perform a multiply. This reduces circuit switching inside the multiplier, reducing the overall power consumption. However, the ALU instruction field `func_v2[4:0]`

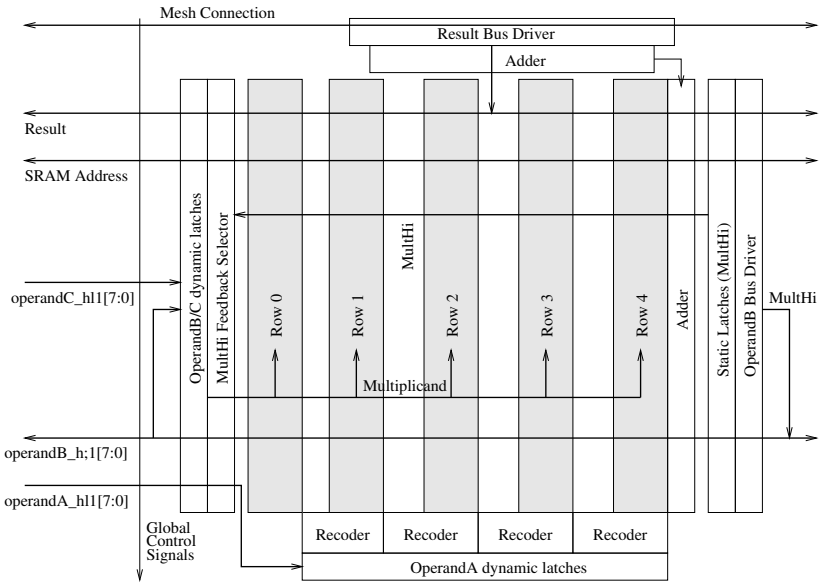

Figure 2.25: Multiplier floorplan, positioned in the PE as shown in Figure 2.4.

controls whether the multiplier treats operands as signed or unsigned and whether operandC or multHi are added into the product, but are not latched depending on whether an instruction performs a multiply. Consequently, the multiplier's product can change even if a multiply is not performed. Adding latches to the global logic to broadcast these control signals only when performing a multiply will reduce the multiplier's standby power consumption.

2.11 Static RAM

Each PE has a 256-byte local static RAM, accessible using either global or local addressing. The memory has a single read-write port with reads storing data into memory data registers (MDRs). Consequently, a read must occur at least one instruction before the instruction that uses the data places the MDR on the operandB bus. If an instruction reads

and writes simultaneously, the value to write goes into memory and the MDR.

2.11.1 Architecture

Figure 2.26 shows the SRAM interface to the Kestrel PE, including address calculation, and Table 2.19 gives brief signal descriptions. The SRAM writes data from the result bus, or reads data into the MDR register, during phase 2. The SRAM memory address can be either the instruction immediate `imm_v2[7:0]` or operandC plus the instruction immediate. Table 2.20 shows the address mode encoding in the result selection field `rm_v2[1:0]`. Compared to Table 2.11, operandC cannot be used by both the result selector and the SRAM address generator. The control for placing the instruction immediate on the operandB bus is also included with the address calculation circuitry since no other unit uses the instruction immediate.

2.11.2 Layout Design

The address generation circuitry is located in the PE core, just below the bit shifter as shown in Figure 2.6. The immediate and operandC adder circuit is a simplified version of the ALU design, hardwired for addition, and the resulting address runs across the multiplier to the SRAM. Figure 2.27 shows the overall floorplan of the SRAM unit, designed entirely by Doug Williams and Don Speck [21]. The primary Vdd and GND connections for the PE run through the SRAM cell array, with metal3 wires running across the multiplier to the PE core.

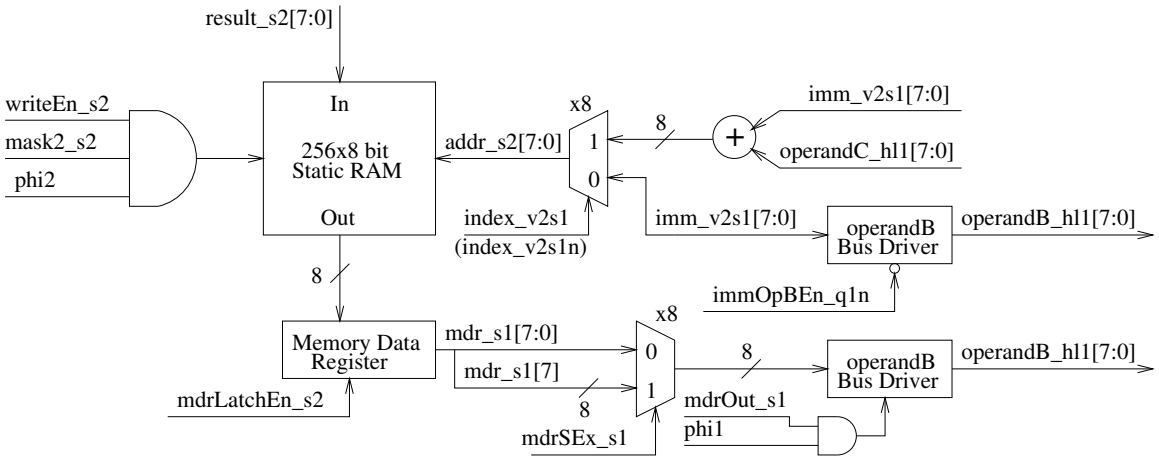

Figure 2.26: SRAM interface to Kestrel PE, including address calculation.

SRAM Control and Data Signals				
Signal	Source	Type	Timing	Purpose
mask2_s2	Local	Input	S2	Local PE enable
result_s2[7:0]	Local	Input	S2	Result bus, input data
addr_s2[7:0]	Local	Input	S2	Address
operandB_hl1[7:0]	Local	Output	HL1	OperandB bus, MDR output
mdrLatchEn_s2	Global	Input	S2	Overwrite MDR with new data
writeEn_s2	Global	Input	S2	Enable memory write
mdrOut_s1	Global	Input	S1	MDR onto operandB bus
mdrSEx_s1	Global	Input	S1	MDR sign extension to operandB
Address Generator Control and Data Signals				
operandC_hl1[7:0]	Local	Input	HL1	optional PE-local index
addr_v1s2[7:0]	Local	Output	V1S2	Address result, goes to SRAM
operandB_hl1[7:0]	Local	Output	HL1	OperandB bus, immediate output
index_v2s1	Global	Input	V1	Select between absolute or
index_v2s1n	Global	Input	V1	indexed addressing
immOpBEn_q1n	Global	Input	Q1	Write immediate to operandB
imm_v2s1[7:0]	Global	Input	V1	Instruction immediate value

Table 2.19: SRAM and address generator control and data signals.

rm_v2[1:0]	Addressing Mode
00	operandC_hl1[7:0] + imm_v2[7:0]
01	imm_v2[7:0]
10	imm_v2[7:0]
11	imm_v2[7:0]

Table 2.20: SRAM addressing mode selection based on the rm_v2[1:0] instruction field.

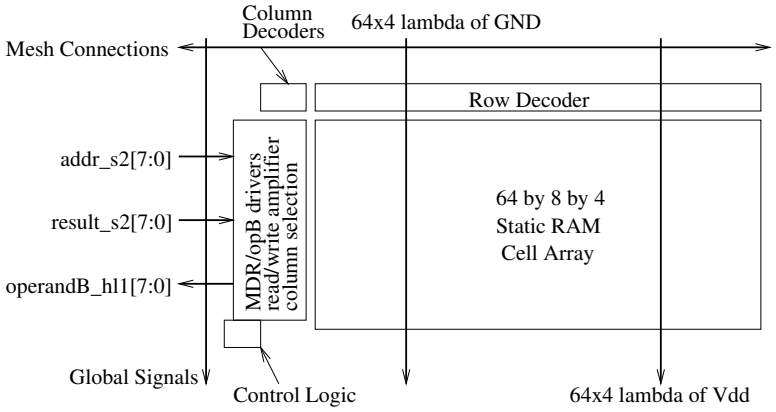

Figure 2.27: Floorplan of the SRAM, with design and layout by Doug Williams and Don Speck.

2.11.3 Problems and Improvements

Increasing the the size of the PE's local memory will always improve performance of many algorithms. Using DRAM instead of SRAM could potentially quadruple the local-memory size without substantially increasing the physical area. However, a 256-byte local memory matches the 8-bit PE data width, and DRAM would drastically complicate the design and reduce the likelihood that inexperienced designers could produce a working system on the first attempt. Increasing the local memory size without changing to DRAM would increase the footprint of each PE, reducing the maximum number of PEs possible in a single system.

2.12 Chip Fabrications

Before fabricating the 64-PE Kestrel design, several test chips allowed independent verification of the multiplier, SRAM, and register banks. Additionally, a 2-PE test chip allowed testing of the PE design and interprocessor communication. While these chips provided important design feedback and revealed several bugs, they primarily built confidence that the designers and tools could successfully create a 1.4 million-transistors design in a submicron technology. All designs were created in full-custom VLSI using the Berkeley Magic tool suite, simulated using `irsim`, and fabricated using the Mosis VLSI Fabrication Service.

2.12.1 Multiplier Test Chip

The multiplier test chip, fabricated in the Orbit 2.0 μm process using a TinyChip standard pad frame as shown in Figure 2.28 (a), allowed isolated testing of the multiplier. Of the four chips fabricated, all worked as expected, except for one problem not found in simulation. The dynamic latches for the multHi that controlled the addition of MultHi into the product of a subsequent multiply had clock and data lines reversed on the nMOS part. The resulting charge sharing problem caused the addition of erroneous values into subsequent multiplications. This proved a valuable lesson as the initial layout of the ALU and comparator dynamic latches contained the same error.

2.12.2 Memory Test Chip

The memory test chip, as shown in Figure 2.28 (b), measured 2.49 mm by 2.49 mm with 18 996 transistors, including both the SRAM and register bank. Of the 25 chips

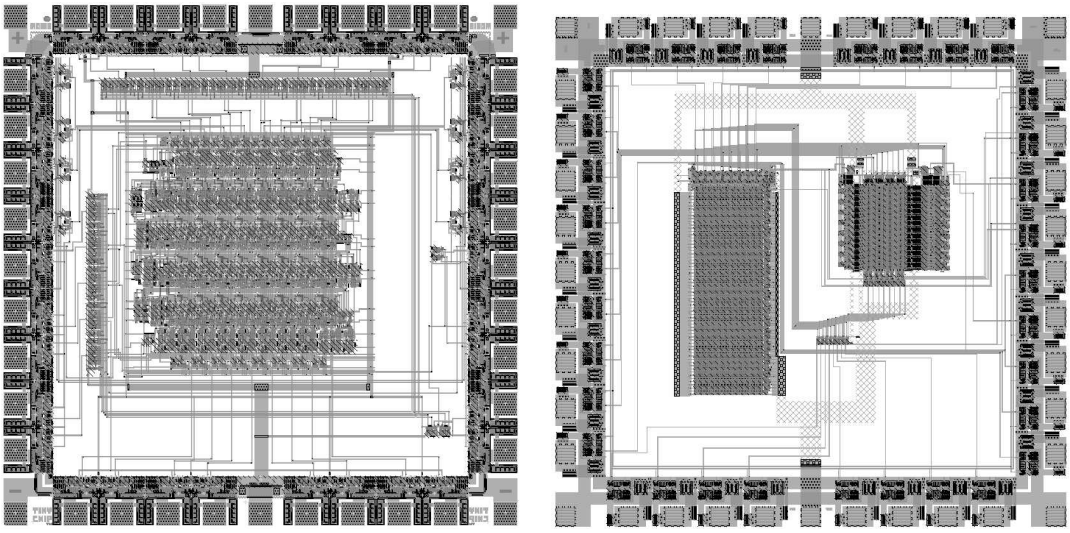

Figure 2.28: (a) Multiplier test chip, fabricated in the Orbit 2.0 micron process as a TinyChip, and (b) the memory test chip, fabricated in the HP 0.5 micron process.

fabricated in the HP 0.8 μm process, one had a defect in two adjacent bits of the SRAM, and another had a defect that caused unpredictable behavior in the register bank. The SRAM could perform a read or write in approximately 7.5 ns, and the register bank could perform a read or write in 4 ns or less. The arrangement of on-chip control lines limited register-bank speed testing to the tester's minimum pulse width of 4 ns.

2.12.3 2-PE Prototype

The Kestrel PE prototype chip, shown in Figure 2.29 (a), has 51 966 transistors in 4.29 mm by 4.05 mm, fabricated in the HP 0.8 μm process. The design consists of two PEs, three shared register banks, and instruction latches and global decoding logic. This chip allowed testing PE functionality and data movement through the shared register banks, while providing invaluable experience with the global layout issues that became increasingly complex in the 64-PE design. While more PEs would have fit on the prototype, the increased complexity would not have provided more testing information.

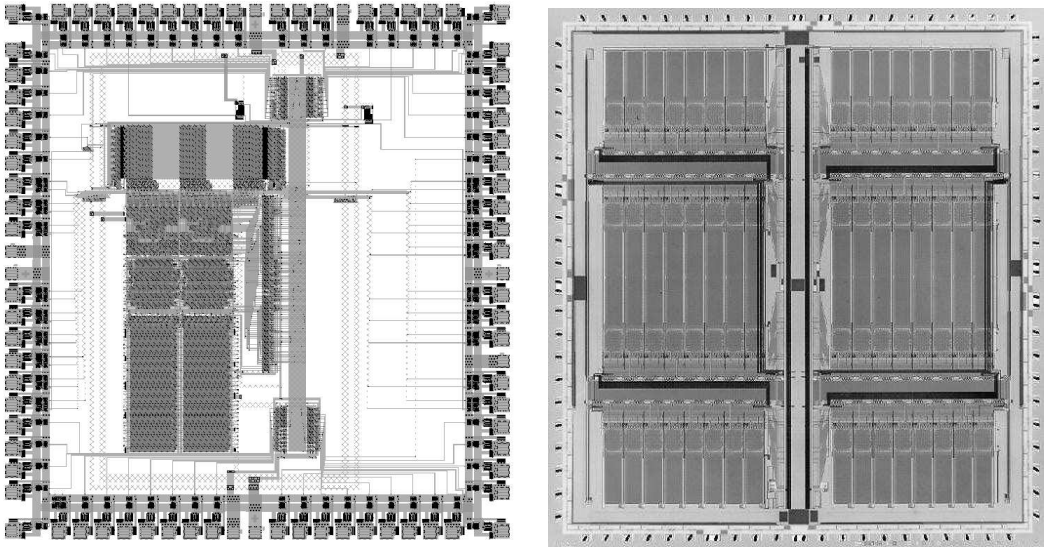

Figure 2.29: (a) 2-PE prototype chip, fabricated in the HP 0.8 micron process, and (b) the final 64-PE chip, fabricated in the HP 0.5 micron process.

The PE prototype was fully functional apart from a minor control logic error where the global logic for the bit shifter's operandB bus driver (`bsOpBEn_q1n`) used a bit from the `bit_v2[3:0]` instruction field instead of the correct bit from `opB_v2[2:0]`. The design simulations before fabrication did not detect the problem since they never read the bit shifter contents onto the operandB bus, instead probing the bit shifter register directly. Of the twenty-five chips fabricated, four failed due to fabrication defects. The chips ran with a 35 ns cycle time, and there was little variation in timing performance, apart from two chips that ran approximately 10% faster.

The pin-out for the prototype is almost identical to the 64-PE chip, aside from timing changes on the mask and wired-OR signals. The mask flag transmitted between PEs is `V1S2` in the prototypes, and sent one phase earlier as `V2S1` on the 64-PE chip, allowing part of phase 2 and all of the following phase 1 for cross-chip communications. This critical change guarantees the register bank produces the correct `S2` signal to prevent unwanted

register writes without slowing the clock. Also, a dynamic latch changes the off-chip wired-OR from a `V1` signal on the prototype to a `V2` signal on the 64-PE chip, providing phase 1 for on-chip evaluation and phase 2 for communication to the instruction sequencer.

2.12.4 64-PE Chip

The 64-PE chip, shown in Figure 2.29 (b), consists of eight rows of eight PEs, with 1.4 million transistors and dimensions of 7.2 mm by 8.3 mm in the HP 0.5 micron process. Testing indicated a maximum possible clock cycle time of 22 ns, with power consumption within the one-watt range that can be dissipated from the `PGA84` package without heatsinks or fans when operated at approximately 25 MHz. Five of the twenty-five chips had fabrication defects ranging from one or two bad SRAM locations to complete incoherence.

The only design error found fortunately does not affect normal operations. The tristate pin to disable all output signals, instead of turning them off, turns all the outputs on. This has the unfortunate affect of shorting adjacent chips in the array, and should never be used on the Kestrel board or on the tester. Appendix D describes how to create a netlist from layout and simulate the design, and Appendix E describes how to test the 64-PE chips using the IMS tester.

Floorplan

The bit-serial mesh interconnection, included at the last minute, greatly increased the complexity of the global layout design, shown in Figure 2.30. As discussed in Section 2.9, the mesh network superimposed on the array connects PEs to their +1, -1, +8, and -8 neighbors in the linear chain. Consequently, the linear PE array does not "snake", with

row's data flowing in the opposite direction as the adjacent rows, but rather always flows in the same direction. This allows easy +8 and -8 connections to the PE in the same position in the adjacent rows. As shown by the `result/mask` lines in Figure 2.30, two sets of unidirectional result buses and mask flags run the length of the rows to connect row ends together.

Instruction bits route from the pads to the instruction latches, where they get broadcast down the chip's center. Each 8-PE row has its own copy of the instruction decoding circuitry, as shown in Figure 2.31, that produces control signals for the corresponding row, with PEs routing the signals horizontally across the row. Address decoders for operandA and operandC appear on the right and the destination address decoders appear on the left. The register banks located above and located to the right correspond to logically left and right shared registers, respectively.

Rows labeled "Reversed" and "Normal" in Figure 2.30 refer to the row's orientation. Since the register bank is only half the PE's width, every other row gets rotated 180 degrees so the register banks interleave. Unfortunately, this moves the global mesh connections that run down the PE's side, forcing connections to run across the PE horizontally. To solve this, "Reversed" rows have PEs flipped horizontally about their vertical centerline so after a 180 degree rotation, their mesh wires align with the adjacent rows. Due to the flipping, "Reversed" PEs also connect to their register banks differently, and so have different register interface wiring.

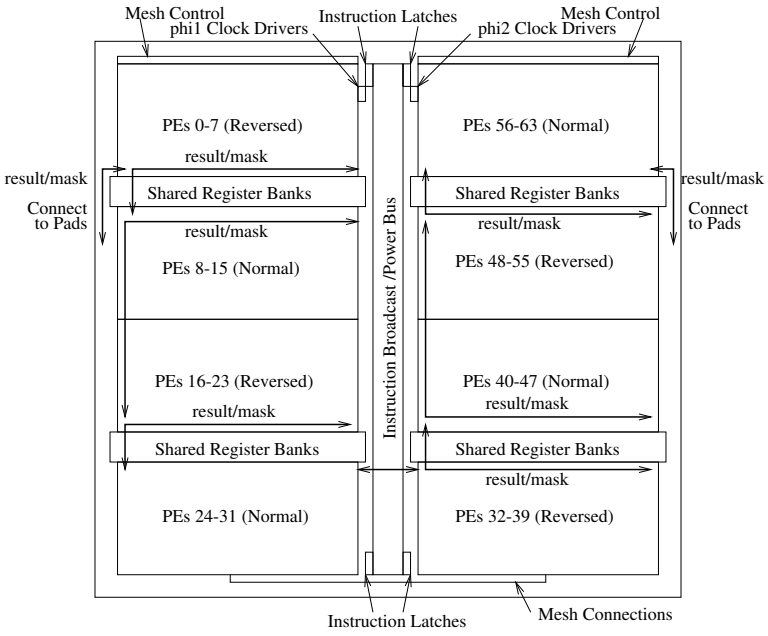

Figure 2.30: Floorplan of the 64 PE Kestrel chip including eight rows of eight PEs, instruction laches, instruction broadcast, clock drivers, and mesh control logic.

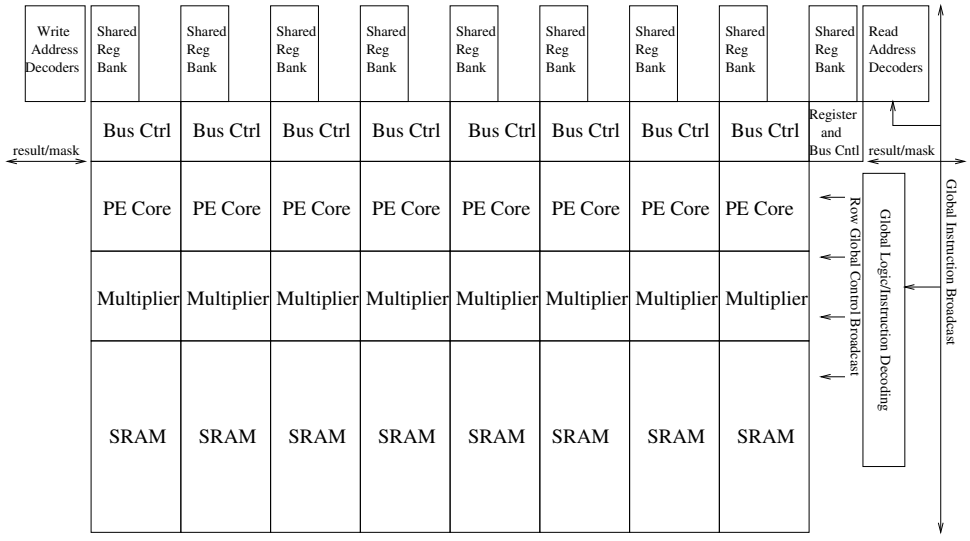

Figure 2.31: Row of eight Kestrel PEs with global control logic.

Chapter 3

Kestrel System

The Kestrel system consists of host machines providing network access for 512-PE Kestrel boards, and UNIX machines providing a user interface. Depicted in Figure 3.1, the user provides a Kestrel program and input and output streams to the UNIX runtime environment, and the runtime environment connects to the Kestrel host through a network connection and provides the program and data streams. The host then configures the Kestrel board and executes the user's program, supplying input data and storing the program's output data. The system supports both multiple users and multiple Kestrel boards, where the runtime environment can search a list of hosts for an unused board. The hardware component is a Peripheral Component Interface (PCI) board that interfaces the Kestrel array with the host machine, and the software component is the host server that operates the board and provides the network interface, and the UNIX runtime environment that provides the user interface.

Most design work for these components was done by other members of the Kestrel team, so this chapter does not discuss any one component in detail. Instead, it discusses the

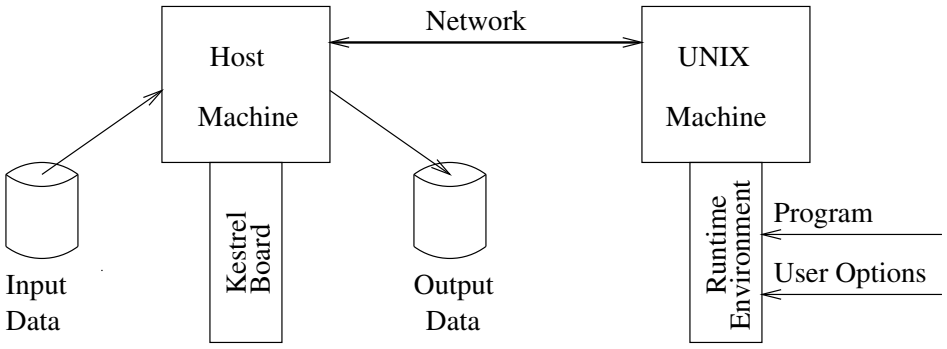

Figure 3.1: Diagram of the Kestrel system.

integration of hardware and software components to form a working system, focusing on how the system runs a Kestrel program, and presenting low-level details only when necessary to explain design problems. Section 3.1 presents an overview of the board architecture, and subsequent sections describe individual components and their operations. Section 3.2 describes the PCI interface and Section 3.3 the two-phase clock generator. Section 3.4 presents the array-controller architecture as well as the instruction fields and their purposes. Sections 3.5 and 3.6 describe how to operate the array controller using the command and status registers, respectively, and Section 3.7 describes queue data-movement operations. Sections 3.8 and 3.9 presents the interfaces provided by the host and runtime environment, respectively. Finally, Section 3.10 summarizes by presenting a walk-through of the execution of a Kestrel program.

3.1 Board Overview

The Kestrel board, shown in Figure 3.2, interfaces the Kestrel array to the host system and provides high-level SIMD instruction sequencing. Figure 3.3 shows a digram of

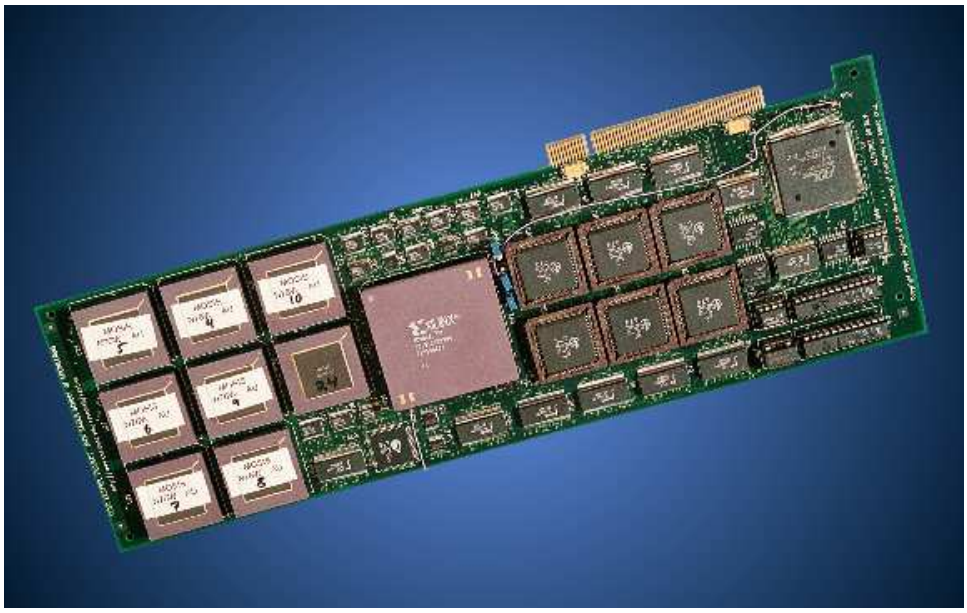

Figure 3.2: Photograph of the Kestrel board.

the board architecture, including the Kestrel array, array controller, and additional components used by the host to execute programs. The PLX 9050 bus bridge interfaces the Kestrel system to the PCI bus, providing a simple 32-bit local bus that connects board components to the PCI bus. The clock generator and custom logic produce the two-phase nonoverlapping clock used by the array and controller. The board supports a dynamically resizable array of 512 PEs, and the array controller broadcasts SIMD instructions, provides global program-flow control, manages data flow between the array and input and output data streams, and performs administrative tasks such as loading program memory and low-level program debugging. The command register manages controller operations and the array configuration, and the status register signals program-error conditions, data-queue states, and controller diagnostic information.

Input and output queues buffer data transfers between the Kestrel array and the host system. The controller, through the PLX 9050, generates interrupts when the queues

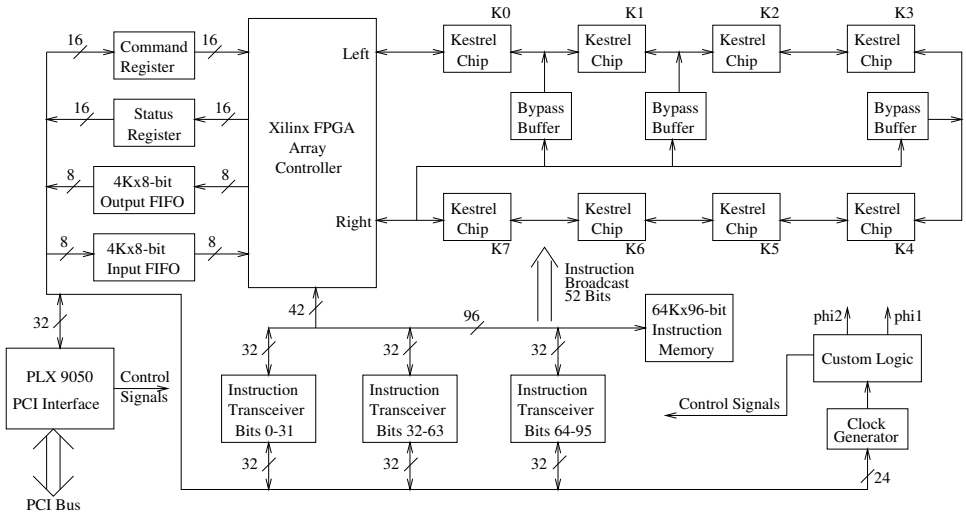

Figure 3.3: Diagram of the Kestrel board architecture, designed by Jeff Hirschberg.

require attention by the host, due to an empty input queue or a full output queue. Other program events cause interrupts, including program errors or breakpoints that indicate the end of program execution. Once the host loads and starts a program running, any further interaction occurs through an interrupt handler.

Jeff Hirschberg designed the Kestrel board, and Hansjörg Keller designed the FPGA array controller. Unfortunately, both left the project before board testing began without writing any significant documentation, so board testing and debugging involved reverse engineering the design to isolate problems and develop workarounds. The board and controller schematics provided a reasonable source for low-level design details, but failed to provide an architectural overview or the commands required for Kestrel-program execution. Consequently, most work involved fixing problems and developing host driver software, and making the board, host, and runtime environment work together to successfully execute a Kestrel program.

3.2 PCI Interface

The PLX 9050-1 bus bridge interfaces the Kestrel board to the PCI bus. The 9050 provides a wealth of features, most of which are unused by Kestrel. Primarily, the 9050 handles all PCI bus signals and provides a simple on-board local bus for data and control movement between the Kestrel array and the host system.

3.2.1 Address Space Configuration

The 9050 maps PCI-bus addresses to the local-bus address space, where on-board control logic decodes the local addresses to individual component enables. The address translation uses the top three bits of the 28-bit address space, as shown in Table 3.1. Components have different characteristics, including address bus data width and data hold-time requirements, with the configuration for each component programmed into the PLX before data transfer begins.

The 9050 has four programmable address spaces, each with a Bus Region Descriptor and Address Remap register describing the local-bus properties. Table 3.2 shows the distribution of components over the four address spaces, and the values for the two 9050 configuration registers for each component, the meanings of which are described below. Since there are more components than address spaces, components not used together during the same operation map to the same address space, preventing contention and minimizing the number of times the configuration registers must be written. The clock only needs to be programmed once during board initialization after power up, and the input queue only once before running a Kestrel program. Loading a program uses the three instruction transceivers and the control/status registers, and managing program execution requires only the queues

Local Address Bits 27-25	Width	Type	Part
000	8 bits	Read Write	Output Data Queue Input Data Queue
001	8 bits	Write	Program Input Queue Signal Thresholds
010	32 bits	Read/Write	Instruction Transceiver Bits 0-31
011	32 bits	Read/Write	Instruction Transceiver Bits 32-63
100	32 bits	Read/Write	Instruction Transceiver Bits 64-95
101	16 bits	Read Write	Array Controller Status Register Array Controller Register
110	24 bits	Write	Clock Generator Delay Lines
111	Unused		

Table 3.1: Kestrel local-bus address translation based on the top three bits of the 28-bit address space.

and control/status registers.

Table 3.3 summarizes the fields of the Bus Region Descriptor register as well as the values identical between all components. The default setting disables data prefetching, sets the byte mode to little-endian, and configures wait states for the local bus. The wait states generally have no effect since the chip-enable circuitry uses only a subset of the bus-control signals. The bus-width field varies for each component based on values from Table 3.1. Programming the clock requires a longer write-cycle hold time due to data setup and hold time requirements.

The Bus Address Remap register indicates how to translate PCI addresses to local addresses. Table 3.4 shows the remap register's fields, with the defaults enabling the address space, and the top three bits of the address remap field corresponding to the local addresses listed in Table 3.1. The Range register fields, shown in Table 3.5, specifies which address bits PLX translates using the Bus Address Remap register. A one bit in the address decoding field indicates the corresponding bit in the PCI address should be replaced with the the bit

Component	Bus Address Remap	Bus Region Descriptor
Address Space 0		
Control/Status Register	0x0A000001	0x00400020
Address Space 1		
Input/Output Queues	0x00000001	0x00000020
Program Input Queue	0x02000001	0x00000020
Program Clock	0x0C000001	0xC0800020
Instruction Transceivers 0-31	0x04000001	0x00800020
Address Space 2		
Instruction Transceivers 32-63	0x06000001	0x00800020
Address Space 3		
Instruction Transceivers 64-95	0x06000001	0x00800020

Table 3.2: Address space allocation for board components and their PLX 9050 register values.

Bits	Value	Description
0	0	Burst Mode Enable
1	0	Ready Input Enable
2	0	Bterm Input Enable
3-4	0	Prefetch Count
5	1	Prefetch Count Enable
6-10	0	Number of Read Address-to-Data wait states
11-12	0	Number of Read Data-to-Data wait states
13-14	0	Number of Read/Write Data-to-Address wait states
15-19	0	Number of Write Address-to-Data wait states
20-21	0	Number of Write Data-to-Data wait states
22-23	Depends	Bus Width: 0: 8-bit 1: 16-bit 2: 32-bit
24	0	Byte ordering (0=little endian)
25	0	Big Endian Byte Lane Mode
26-27	0	Read Store Delay
28-29	0	Write Strobe Delay
30-31	Depends	Write Cycle Hold

Table 3.3: Local Address Space Bus Region Descriptor Register. Refer to the PLX 9050 datasheet for full details [20].

from the Bus Address Remap register. A zero indicates no translation, with the number of zeros determining the size of the address space.

3.2.2 Interrupts

The 9050 also generates PCI interrupts based on the array controller's interrupt signal. The Interrupt Control/Status register, summarized in Table 3.6, manages whether the controller's active-low interrupt signal actually generates a PCI interrupt. Kestrel uses only interrupt 1, so the second interrupt must always be disabled. To enable interrupt generation, both the PCI Interrupt Enable and the Local Interrupt 1 Enable must be one. Since PCI interrupts are level-sensitive and not edge-sensitive, the first operation of the interrupt handler must disable the interrupt generation by clearing the interrupt control register, otherwise, the interrupt handler will be called repeatedly until the system crashes. Once the handler completes, it re-enables the interrupts through the Interrupt Control/Status register.

Bits	Default	Description
0	1	Address Space Enable
1	0	Unused
2-3	0	Unused for memory-mapped I/O
4-27	Depends	Remap PCI Address to Local Address
28-31	0	Unused

Table 3.4: Local Address Space Bus Address Remap Register. Refer to the PLX 9050 datasheet for full details [20].

Bits	Default	Description
0	0	Memory Space Indicator Type (0=memory or 1=I/O mapped)
1-2	0	Where to locate memory in PCI address space (0=anywhere)
3	0	Allow prefetchable memory
4-27	0xfff000	Address Decoding Range (mask)
28-31	0	Unused

Table 3.5: Local Address Space Range Register (LASXRR). Refer to the PLX 9050 datasheet for full details [20].

Bits	Default	Description
0	1	Local Interrupt 1 Enable
1	0	Local Interrupt 1 Polarity (0=active low)
2	X	Local Interrupt 1 State (read only)
3	0	Local Interrupt 2 Enable
4	0	Local Interrupt 2 Polarity (0=active low)
5	X	Local Interrupt 2 State (read only)
6	1	PCI Interrupt Enable
7	0	Software Interrupt
8-31	0	Unused

Table 3.6: Interrupt Control/Status Register. Refer to the PLX 9050 datasheet for full details [20].

3.2.3 Initialization

An EEPROM connected to the 9050 initializes all registers to the correct initial value, and is programmed as described in Appendix A. Consequently, configuration registers usually only to be need changed when switching between components assigned to the same address space or when handling interrupts. However, one important exception is the local-bus reset (bit 30) of the 9050's `CNTRL` register. Setting this bit causes a reset of the queues, and the command register and FPGA as discussed in Section 3.5.3. The host should perform this reset between each Kestrel program.

3.2.4 Problems and Improvements

The use of the three most-significant local-address bits 27-25 for address translation complicates address configuration. Similar components with the same Bus Region Descriptor cannot be placed in the same address space simultaneously since the Microsoft Windows-based host system limits the size of contiguous PCI address ranges to 64 MBytes, whereas an address space addressing bits 25-27 would require 256 MBytes of address space. This problem can be easily corrected by using the least-significant address bits.

3.3 Clock Generator

The Kestrel PE design uses a two-phase, nonoverlapping clock as discussed in Section 2.2.1. The 64-PE chips do not generate the clocks from a single source, but require both as inputs. Moving the clock generation to the board simplified the chip design, and increased the likelihood of creating a working system on the first attempt. A flexible software-programmable clock generator could have allowed optimization of the phase and gap times to acheive maximum speed, but board design problems prevented this strategy.

3.3.1 Design

The on-board clock generator takes a single clock input and generates a two-phase clock at twice the frequency. That is, for each cycle of the input clock, the system generate two phase 1 and phase 2 pulses. The clock generator design, shown in Figure 3.4, exclusive ORs three delayed version of the input clocks to produce the two-phase clocks, with delays programmable through the board's PCI interface.

An exclusive OR of the original clock and the delayed version `Delay0` produces

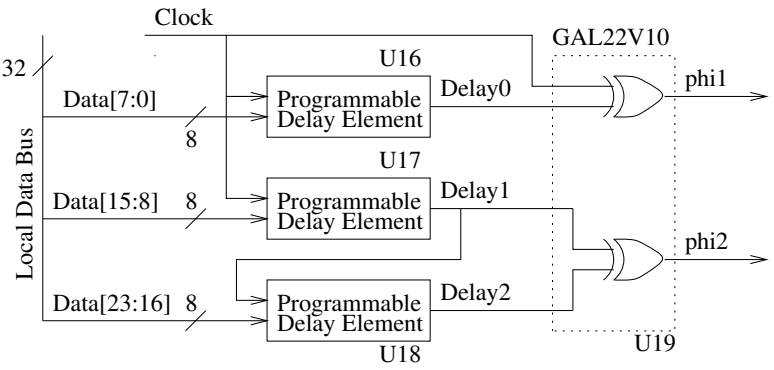

Figure 3.4: Kestrel's software-programmable, two-phase clock generator, designed by Jeff Hirschberg.

the phase 1 clock. A second version `Delay1`, with a delay time including the lengths of phase 1 and the gap, determines the phase 2 start time. The third version `Delay2`, based off `Delay1`, determines the length of phase 2, with phase 2 generated from the exclusive OR of `Delay1` and `Delay2`. Figure 3.5 shows the relationships between the delayed clocks and the resulting phase 1 and phase 2 clock signals.

The programmable delay elements take an 8-bit number and a clock and produce a clock delayed by 10 ns plus 0.15 ns times the 8-bit number. The current Kestrel system uses a 10 MHz clock, running the PE array at 20 MHz, and using programmable delays for `Delay0`, `Delay1`, and `Delay2` of 53, 101, and 53 delay units, respectively. These delays produce phi1, gap, phi2, and gap lengths approximately of 18 ns, 5 ns, 18 ns, and 8 ns, respectively.

3.3.2 Problems and Improvements

The clocks were a major source of problems during Kestrel board debugging. The board is dense, and due to poor clock buffer placement, many clock signals must travel the

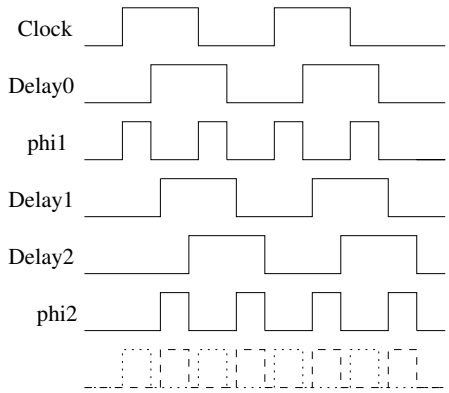

Figure 3.5: Relationships between the clock-generator's delayed clocks `Delay0`, `Delay1`, and `Delay2` and the outputs phi1 and phi2.

entire board length. Consequently, signal quality is low, and when operated at the target clock frequency of 33 MHz, the system becomes unstable and crashes during periods of heavy usage. Future board revisions should reduce the total number of chips, allowing better component placement and routing of critical signals. Moving glue-logic functions currently performed by discrete components to a denser FPGA array controller might improve board signal quality.

The original board design used the PCI-bus 33 MHz clock for the local bus and the on-board clock generator for the Kestrel array and controller. The on-board clock frequency could be reduced by changing the crystal, and the current system uses a 10 MHz crystal. To reduce the local-bus clock frequency, the PLX 9050's `BCLKO` output pin (`U1.63`) had to be disconnected from the board so a lower-frequency clock could be connected. After connecting the clock crystal's output `PHI_SOURCE` (`U21.5`) to the clock driver input `U39.43`, the local bus ran at 10 MHz, eliminating the intermittent crashes while reducing the data transfer speed.

3.4 Array Controller

The array controller manages all aspects of Kestrel program execution, including control flow, data movement, and error-condition handling. Control-flow operations include branches, subroutines, and loops, and data movement includes moving data between the queues and the ends of the array. The controller also generates interrupts based on program conditions such as program errors or breakpoints, and empty or full data queues requiring host intervention. Finally, a diagnostic mode provides access to internal controller values for run-time debugging, and a single-step mode for interactive program execution.

Array controller operations interlock with the command and status registers, input and output queues, and the instruction memory or transceivers, forming one cohesive functional unit. The controller manages each of the components based on the operating mode specified in the control register and instruction bits in the transceivers or the addressed instruction memory location.

3.4.1 Design

The controller provides basic control flow and data movement for Kestrel programs, using 40 instruction bits as shown in Table 3.7. The controller's instruction fields are highly horizontal, almost always corresponding directly to single operations. Fields group into four basic types: control flow, data register, data movement, and diagnostic/immediate fields.

The control-flow fields operate both a program and loop-counter stack for nested subroutines and loops, as well as program counter (PC) generation and branch logic. Data-register fields control when and from where internal registers latch new data, and the data-movement fields control data flow between the Kestrel array and the input/output queues.

Bits	Field	Description
	Control-Flow Fields	
0-1	I_PC_SEL[1:0]	PC Select, Overridden by Branches
2	I_PUSH_PC	Pop Address of PC Stack
3	I_POP_PC	Push Immediate onto PC Stack
4	I_BR_0	Jump on Counter Stack Decrement and Pop
5	I_BR_W_OR	Jump on Wired-OR
6	I_PUSH_CNT	Push Value onto Counter Stack
7	I_DEC_CNT	Decrement Top of Stack and Pop on underflow
8	I_CNT_LOAD_0	Load Scratch Register into low or
9	I_CNT_LOAD_1	high bytes of Top of Counter Stack
	Data-Register Fields	
10	I_CBS_LOAD	Parallel Wired-OR Load from each Chip
11	I_CBS_SLEFT	Shift 1-bit Wired-OR into Shift Register
12-13	I_SCR_SEL[1:0]	Scratch Register Source Select
14	I_SCR_STORE	Replace Scratch Register Contents
	Data-Movement Fields	
15	I_IN_DATA_SEL_0	Select Kestrel Array Input Data
16	I_SPARE0	Unused
17	I_DATA_READ	Write Data To Kestrel Array
18	I_DATA_WRITE	Write array data to output queue
19	I_FIFO_OUT	Force data write to output queue
	Diagnostic/Immediate Fields	
20	I_BREAK	Program Breakpoint
21-36	I_IMM[15:0]	Array controller immediate
37	D_pop_cnt	Force loop-counter stack pop
38-40	D_OUT[2:0]	Diagnostic mode status register select
41	BOARD_IMM_MUX	Select Kestrel immediate source
42-43	I_SPARE1[1:0]	Unused
44-95	Kestrel-array instruction bits	

Table 3.7: Instruction fields used by the array controller and board to control a Kestrel program.

Finally, the diagnostic and immediate fields provide program debugging support and instruction immediate control.

Control Flow

The PC stack shown in Figure 3.6 (a) supports up to 15 nested subroutine calls, and the PC selector generates the address of the next instruction, with Table 3.8 describing the non-instruction-bit signals. The PC selector performs all program flow control using the `I_PC_SEL[1:0]`, `I_BR_0`, and `I_BR_W_OR` signals. The `I_PC_SEL[1:0]` field chooses the source for the next PC, and the two branch flags modify that choice depending on whether the instruction performs a branch. Table 3.9 illustrates the relationship between the instruction fields and the resulting operations. When performing a branch, the immediate field `I_IMM[15:0]` provides the target address.

The loop-counter stack shown in Figure 3.6 (b) supports up to 15 nested loops. The number of loop iterations can come from the immediate `I_IMM[15:0]` or from the scratch register `SCR_OUT[7:0]`. For each loop iteration, control logic decrements the top of stack and branches to the beginning of the loop, with the destination address supplied by

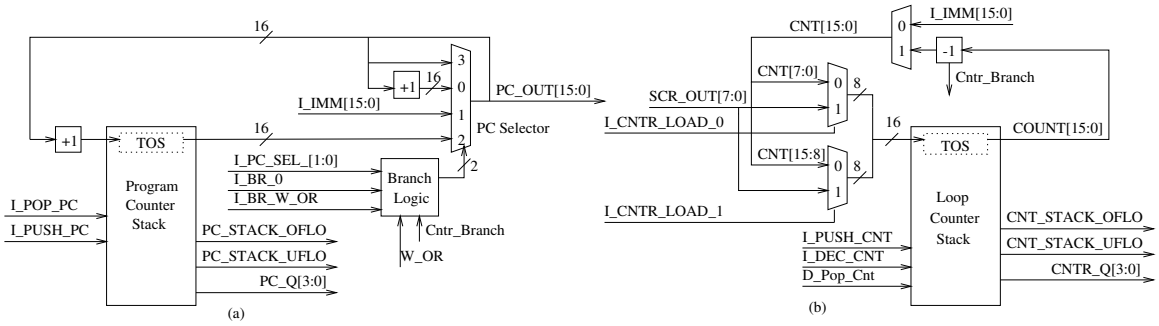

Figure 3.6: (a) Program-counter (PC) stack and selector with branch control logic, and (b) loop-counter stack and loop-counter selectors.

Signal	Type	Timing	Description
PC_STACK_OFLO	Output	V1S2	Program-counter stack overflow
PC_STACK_UFLO	Output	V1S2	Program-counter stack underflow
CNT_STACK_OFLO	Output	V1S2	Loop-counter stack overflow
CNT_STACK_UFLO	Output	V1S2	Loop-counter stack underflow
PC_OUT[15:0]	Output	V1S2	Current program counter
SCR_OUT[7:0]	Internal	V2S1	Scratch-register contents
W_OR	Internal	V1S2	Kestrel-array wired-OR
Cntr_Branch	Internal	V1S2	Top of counter stack decrement underflowed

Table 3.8: Non-instruction-bit signals used in the program and loop-counter stacks from Figure 3.6.

I_PC_SEL[1:0]	I_BR_W_OR	I_BR_0	W_OR	Cntr_branch	Operation
00	0	0	X	X	goto next instruction
00	1	0	0/1	X	0=don't branch/1=branch
00	0	1	X	0/1	0=don't branch/1=branch
01	X	X	X	X	always branch
10	0	0	X	X	return from subroutine
11	X	X	X	X	idle; don't change PC

Table 3.9: Selection of the new PC based on the instruction fields and branch conditions.

the immediate I_IMM[15:0]. When the counter underflows, control logic pops the top of stack and continues to the next sequential instruction.

Data Registers

The scratch register, shown in Figure 3.7 (a), is the central point for data movement between the array and queues, with Table 3.10 giving brief descriptions of the data signals. The scratch register's input can come from the array, the input queue, or the wired-OR bit-shifter register. Currently, data written from the array to the output queue must pass through the scratch register, and so the scratch register cannot be used for long-term storage.

The wired-OR bit-shifter performs the final stage of the Kestrel-array wired-OR

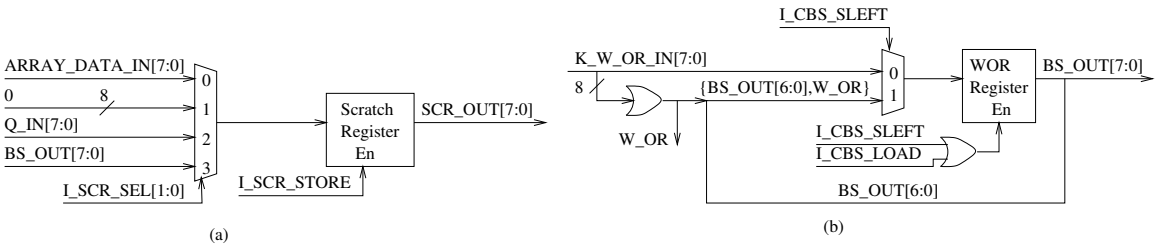

Figure 3.7: (a) Scratch register and source selection, and (b) the wired-OR bit-shifter register.

operation between the most significant bit of active PEs' bit shifters. The register, shown in Figure 3.7 (b), takes the wired-OR signals from the eight Kestrel chips `K_W_OR_IN[7:0]` and shifts the OR into the least-significant bit or performs a parallel load of all eight bits. The shift operation allows packing flag bits together to reduce the output size, and the parallel load could be used for debugging to quickly isolate defective chips using the wired-OR flags.

Data Movement

As shown in Figure 3.8 (a), with Table 3.11 briefly describing the data signals, the controller can send and receive data from both ends of the Kestrel array. Data written to the array can come from either the scratch register or directly from the input queue, and data read from the array always goes into the scratch register. The scratch register can

Signal	Type	Timing	Description
`K_W_OR_IN[7:0]`	Input	`V2`	Wired-OR from each Kestrel chip
`Q_IN[7:0]`	Input	`V1S2`	Next input queue data
`SCR_OUT[7:0]`	Internal	`V2S1`	Scratch register contents
`BS_OUT[7:0`	Internal	`V1S2`	Wired-OR bit-shifter register
`ARRAY_DATA_IN[7:0]`	Internal	`V2`	Data input from Kestrel array
`W_OR`	Internal	`V1S2`	Kestrel-array wired-OR

Table 3.10: Data signals used by the scratch and wired-OR bit-shifter register in Figure 3.7.

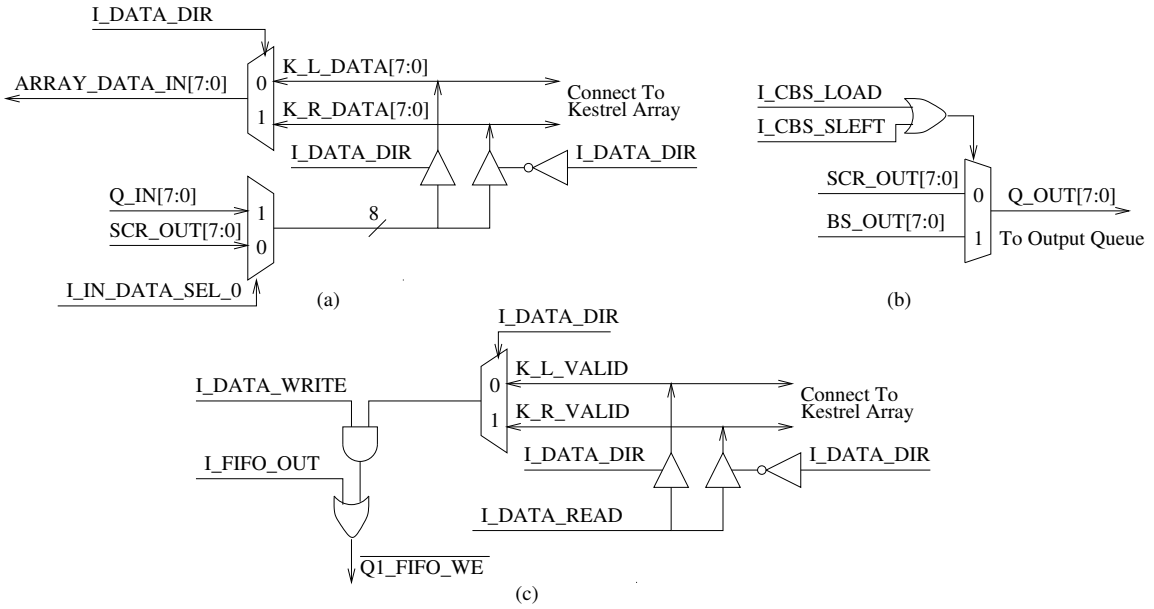

Figure 3.8: Movement of (a) data and (c) the data valid flag between the Kestrel array and the array controller, and (b) the output-queue sources.

then be written to the output queue as shown in Figure 3.8 (b). Data can be copied from one end of the array to the other through the scratch register, allowing data to be cycled through the array without going through the host.

The data valid signal, as shown in Figure 3.8 (c), indicates if the array actually produced data, as the end PE could be disabled through a conditional operation. The controller only writes data from the array to the queues when the array signals the data is valid, but in the current implementation the scratch-register write always occurs. Data can be copied from one end of the array to the another by storing array output data in the scratch register, then writing the scratch register to the other end of the array.

Figure 3.8 (b) shows that data written to the output queue during an instruction that performs a wired-OR bit-shifter operation writes the `BS_OUT[7:0]` register instead of the scratch register. Otherwise, this would require two operations as the wired-OR

Signal	Type	Timing	Description
K_L_DATA[7:0]	In/Out	V2	Data connection to left Kestrel PE
K_R_DATA[7:0]	In/Out	V2	Data connection to right Kestrel PE
K_L_VALID	In/Out	V1	Mask-flag connection to left Kestrel PE
K_R_VALID	In/Out	V1	Mask-flag connection to right Kestrel PE
SCR_OUT[7:0]	Internal	V2S1	Scratch-register contents
BS_OUT[7:0	Internal	V1S2	Wired-OR bit-shifter register
ARRAY_DATA_IN[7:0]	Internal	V2S1	Data input from Kestrel array
Q_IN[7:0]	Input	V1S2	Next input queue data
Q_OUT[7:0]	Output	V1S2	Output queue data
$\overline{\text{Q1\_FIFO\_WE}}$	Output	V1S2	Output queue data write enable

Table 3.11: Non-instruction-bit signals used for data movement from Figure 3.8.

calculation occurs after the scratch register write in the instruction pipeline, as discussed below.

Diagnostic and Immediate Control

The Kestrel-array immediate can be selected between the scratch register and the immediate field imm_v2[7:0]_in, as shown in Figure 3.9. The selection occurs on the board using discrete components, with the result broadcast to all Kestrel PEs. The diagnostic-instruction fields control the status register configuration in diagnostic mode, as discussed in Section 3.6.

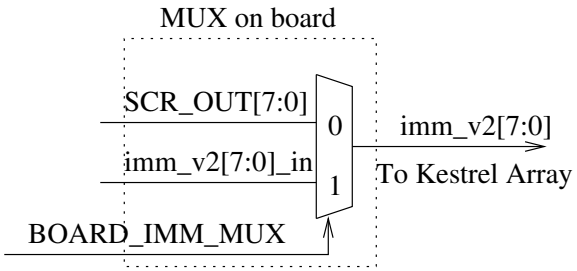

Figure 3.9: Kestrel-array instruction immediate selector.

3.4.2 Instruction Pipelining

The controller uses a four-stage pipeline, shown in Figure 3.10 that interlocks with the Kestrel-array event organization, described in Section 2.2.3. Most controller operations occur during the first cycle, including program- and loop-counter operations, instruction broadcast to the Kestrel array, and input-queue reads. During the second cycle, the Kestrel array execution occurs, with the controller managing data movement between the array and queues. The scratch-register write and wired-OR operation occurs during the third cycle, with data writes to the output queue occurring during the last clock cycle.

Programs using the wired-OR branch or writing the scratch register to the array require nop instructions to stall the pipeline, as the controller does not automatically detect data hazards. The latter stall could eliminated by modifying the controller design to forward the scratch register to the array I/O unit. However, the wired-OR delay cannot be removed because the controller does not capture the wired-OR signals from the array until late during the third cycle. Consequently, an array bit-shifter operation followed by a wired-OR branch always requires two intervening nop instructions.

3.4.3 Problems and Improvements

The current array-controller design is primitive, performing only a minimal set of operations. Possible future enhancements include a more flexible scratch register, and possibly a small data stack and ALU for performing simple operations. A loop counter with flexible counting arithmetic could be utilized by the compiler, for example, to perform arbitrary loops without using the limited PE resources.

Neither the controller nor the broadcast path has pipeline registers, so when an

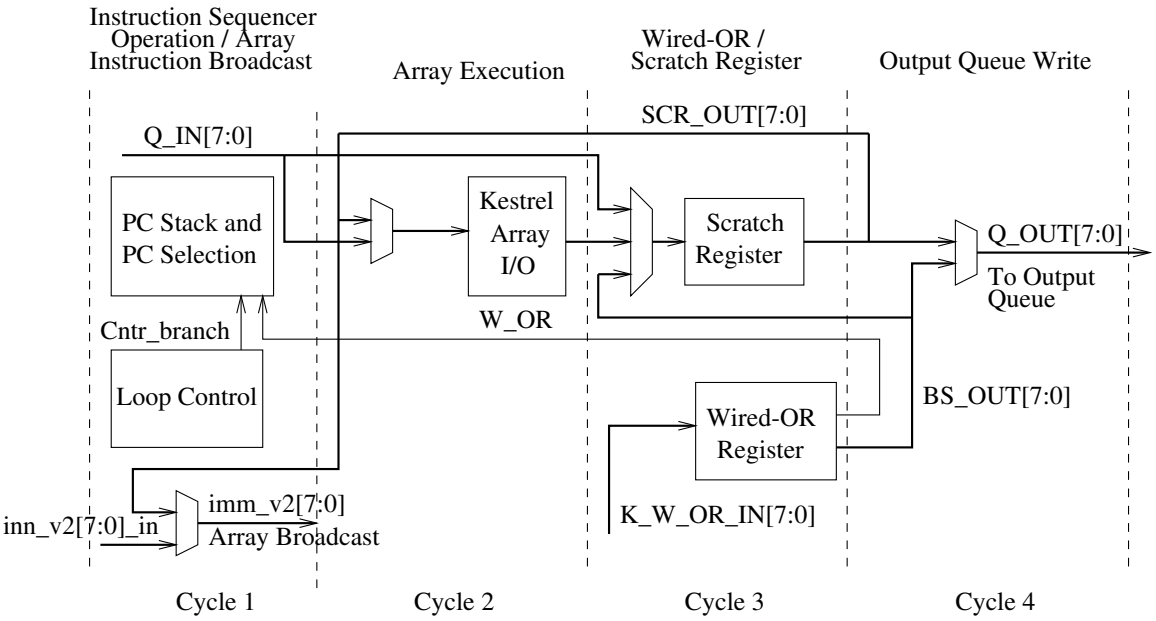

Figure 3.10: Array-controller execution pipeline showing only data movement.

external condition forces a stall, instructions already in the pipeline still complete. As discussed in Section 3.7, this becomes problematic when the output queue fills, the controller halts, but instructions still in the pipeline want to write to the output queue. To avoid losing data, a workaround in the current implementation saves the data until the space in the output queue becomes available. Adding pipeline registers would prevent instructions already in the pipeline from completing.

Almost all controller operations occur during the first cycle, including the instruction memory read. Splitting the first cycle in two, one for fetch and one for execution, could increase the maximum clock speed, with an extra delay element between the instruction memory and the Kestrel array to keep the array and controller synchronized. This may also fix a timing problem with the return-from-subroutine instruction that apparently takes too long to read the return address off the stack.

The FPGA configuration for the controller can be programmed through a special serial cable controlled by the Xilinx Foundation software or from a PROM. In the current board design, the header pin for serial data connects to the wrong FPGA pin, and should connect to `U42.U3` instead of `U42.C1`. Also, the download cable, used when no PROM is present in the board, uses the FPGA's $\overline{\texttt{INIT}}$ signal, which has no header, and is most easily connected to the instruction-memory socket at pin `U8.25`. Three of the PROM connections to the FPGA are wrong, and must be patched. Pins 3 and 4, `_RESET_OE` and `_CE`, respectively, must be bent so as not to touch the socket, and connected to the FPGA's `DONE` signal at header 2 (HD2). Pin 1, the `DATA` signal, must connect to the FPGA's serial download pin `Q_OUT[6]` (`U42.U3`). Figure 3.11 shows the correct wiring for the FPGA's PROM.

3.5 Command Register

The command register, summarized in Table 3.12, defines the controller's operating mode and the Kestrel-array configuration. It provides bits that provide workarounds to problems with the array controller's interface to the input and output queues, as discussed

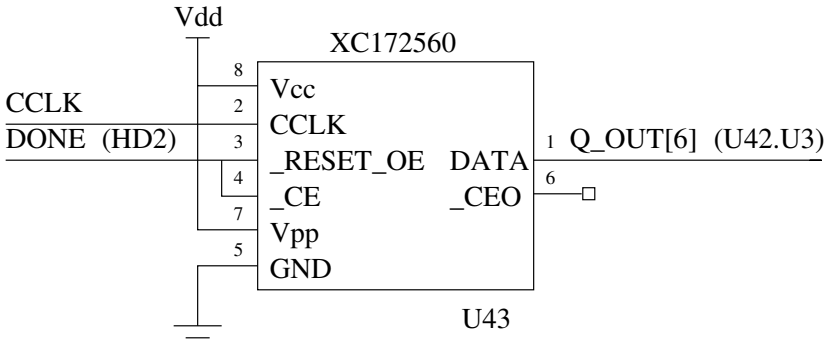

Figure 3.11: Correct wiring for the FPGA's PROM containing the controller's configuration.

in Section 3.7, and the reset bit for the FPGA array controller.

3.5.1 Operating Modes

The command-register bits `RUN`, `SINGLE`, and `DIAG` set the controller operating mode, as shown in Table 3.13. The mode settings allow the host to perform operations other than program execution, including reading and writing to the instruction memory, programming the output queue, and debugging programs.

- `STOP` mode idles the Kestrel array and controller and resets a halt condition. The array controller halts on an event that requires attention from the host before execution can continue, and will only continue when the host clears the condition and restarts the program by switching to `STOP` mode and back to `RUN` mode. When switching between operating modes, the host should always transition through `STOP` mode to prevent anomalous behavior.

- In `SINGLE_STEP` mode, the array controller executes a single instruction if there are no interrupt conditions asserted. In the current implementation, execution occurs when the mode changes to `SINGLE_STEP` mode. The host should always start the controller in `STOP` mode and then switch to `SINGLE_STEP` mode, otherwise, the wrong instruction may be executed.

- `DIAGNOSTIC` mode also allows single-instruction execution, but the controller executes the instruction in the transceivers, not from the memory at the current PC. Unlike `SINGLE_STEP` mode, flipping the `LOAD_TOGGLE` bit causes the single-instruction execution. The host uses `DIAGNOSTIC` mode to perform administrative tasks, such as

Bit	Name	Description
0	LOAD_TOGGLE	Perform a single operation. Mode dependent.
1 2 3	RUN SINGLE DIAG	Array controller operating mode. See Table 3.13 for definitions.
4 5 6 7	$\overline{\text{TRI\_K0}}$ $\overline{\text{TRI\_K1}}$ $\overline{\text{TRI\_K2\_K3}}$ $\overline{\text{TRI\_K4\_K7}}$	Kestrel chip output enables; Due to a flaw in the chip design, these must always be one to prevent drive fights.
8 9 10	$\overline{\text{TRI\_K0\_K1\_BUF}}$ $\overline{\text{TRI\_K1\_K2\_BUF}}$ $\overline{\text{TRI\_K3\_K4\_BUF}}$	Kestrel array bypass buffer output enables; must be one unless the proper Kestrel chip sockets are empty.
11	QIN_LOAD	Read element from input queue.
12	RBQ_OUT	Flush controller output buffer.
13 14	COMMAND_3 COMMAND_4	Unused
15	$\overline{\text{FPGA\_RESET}}$	controller software reset; does not affect FPGA configuration.

Table 3.12: Command-register bit fields that control controller operations, the Kestrel-array configuration, and two bits to workaround problems with the data-queue interface.

Mode Bits	Mode	Description
000	STOP	Array controller idle
001	RUN	Normal program execution.
010	SINGLE_STEP	Execute a single instruction.
011	DIAGNOSTIC	Read internal state through status register, execute single instruction from instruction transceivers when LOAD_TOGGLE changes.
100	LOAD_PROGRAM	Copy instruction from transceivers to program memory at current PC when LOAD_TOGGLE changes. PC incremented automatically.
101	Unused (currently same as STOP)	
110	READ_PROGRAM	Copy instruction from instruction memory at current PC to transceivers when LOAD_TOGGLE changes. PC incremented automatically.
111	PROGRAM_FIFO	Same as DIAGNOSTIC except data written to output queue programs $\overline{\text{Q\_OUT\_NEAR\_FULL}}$ threshold.

Table 3.13: Summary of controller operating modes set from the command register.

draining the input and output queues after program termination. `DIAGNOSTIC` mode also changes the status register, exposing the contents of internal register as selected by the `D_OUT[2:0]` instruction field, as discussed in Section 3.6.

- The `LOAD_PROGRAM` mode allows the host to load a program into instruction memory through the transceivers. To load a program, the host first sets the PC to the desired address by executing a jump instruction in `DIAGNOSTIC` mode. The array controller copies the instruction in the transceivers to memory when the `LOAD_TOGGLE` bit changes. The controller automatically updates the PC before each load using preincrement, so the initial PC should be set to the address before the first instruction.

- `READ_PROGRAM` mode copies instructions from memory to the transceivers, which allows instruction-memory testing. `READ_PROGRAM` using the same PC semantics as `LOAD_PROGRAM` mode.

- `PROGRAM_FIFO` mode is identical to `DIAGNOSTIC` mode, except that data written to the output queue programs the almost-full threshold. To set the threshold, the host loads configuration bytes into the input queue and executes a move instruction that copies data from the input queue to the output queue through the scratch register. Refer to Section 3.7 for details on the data format.

For all operations using the `LOAD_TOGGLE` bit, the `LOAD_DONE` bit from the status register toggles when the operation completes. While this always occurs after a fixed number of cycles, it is useful when the host needs to run several instructions in a tight loop, eliminating the need for a fixed delay between operations. This is especially important if the speed difference between the host and array controller changes, where a hard-coded

delay could become too short.

3.5.2 Kestrel-Array Configuration

Bits 4 through 10 of the command register from Table 3.12 allow software reconfiguration of the Kestrel-PE array, as shown in Figure 3.12. Three bypass buffers allow the array to be shortened to 64, 128, or 256 instead of 512 PEs, with the caveat that controller data flow becomes unidirectional. To shorten the array, the host disables the outputs of the unused Kestrel chips with command-register bits 4-7, and enables the bypass buffer between the used and unused chip with command-register bits 8-10.

Unfortunately, a bug in the Kestrel-chip design prevents the output enables from functioning properly. Instead of turning all outputs off, the output enable turns all outputs on. Consequently, the command-register bits 4-7 must always be one to prevent drive fights between adjacent array chips. The bypass buffers can still configure a board with fewer than eight Kestrel chips, but jumpering across the empty sockets eliminates the unidirectional data limitation.

3.5.3 Command-Register Reset

When a reset occurs, after setting the local-bus reset bit of the PLX 9050's `CNTRL` as described in Section 3.2.3, the command register latches data from the local bus. While the 9050 does not drive the local bus during reset, resistors—hidden in the bottom left-hand corner of last page of the board schematics—pull the lower 16 bits to the correct values to initialize the command register. The initialization sets the operating mode to `STOP`, sets the chip-enables bits 4-10 and bits 11-14 to one (luckily causing no drive fights between the

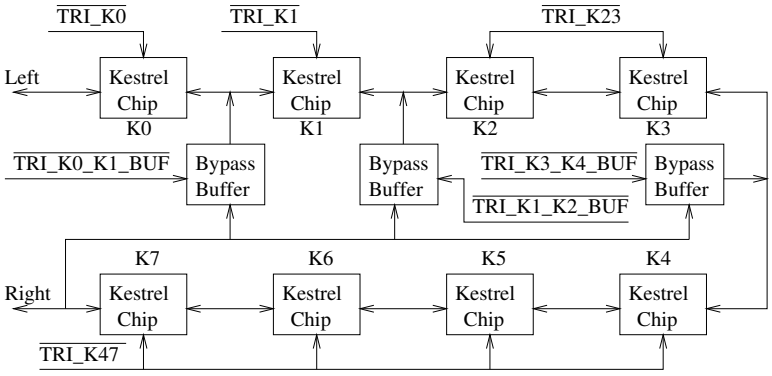

Figure 3.12: The Kestrel-PE array and reconfiguration controls for dynamic array resizing.

flawed Kestrel chips), and the $\overline{\texttt{FPGA\_RESET}}$ bit to zero, causing a reset of the FPGA array controller data, but not of the configuration. The $\overline{\texttt{FPGA\_RESET}}$ bit must be set to one before the array controller can be used.

3.5.4 Problems and Improvements

In the original board design, the local bus ran off the 33 MHz PCI-bus clock, and the array controller ran off the board's clock generator. Consequently, command-register writes occurred on the local-bus clock, so the array controller's operating mode was an asynchronous input. However, the 33 MHz clock was too fast for the design to handle, and the current board's local-bus uses the same 10 MHz clock used by the two-phase clock generator that runs the array controller, so command-register inputs are no longer asynchronous.

3.6 Status Register

The 16-bit status register primarily indicates the cause of interrupts, but can also provide diagnostic information when the array controller operates in `DIAGNOSTIC` mode. The most-significant byte provides queue status bits, and the least-significant byte indicates program-error conditions. When in `DIAGNOSTIC` mode, the lower byte changes to an array-controller internal register selected by the `D_OUT[2:0]` instruction field.

3.6.1 Most-Significant Byte

The status register's most-significant byte provides data-queue and FPGA status, as shown in Table 3.14. Six of the eight bits indicate data-queue conditions, and if any are set, the array controller triggers an interrupt; Section 3.7 presents a detailed discussion of queue flags and operations. The two remaining bits, `LOAD_TOGGLE` and `DONE`, indicate the state of the FPGA. `LOAD_DONE` signals the completion of an operation involving the command register `LOAD_TOGGLE` bit, and the `DONE` flag indicates if the FPGA was properly configured by an PROM or serial cable. A zero in the `DONE` bit indicates an unconfigured FPGA and an unusable board.

3.6.2 Least-Significant Byte

When not in `DIAGNOSTIC` mode, the status register's least-significant byte signals program-error conditions. Table 3.15 lists the five error conditions potentially triggered by a Kestrel program. Overflowing or underflowing the loop- or program-counter stacks or encountering a breakpoint causes program execution to halt, and the setting of the appropriate bit. The host uses breakpoints during interactive program debugging and to

Bit	Name	Description
8	$\overline{\texttt{Q\_OUT\_FULL}}$	Output-queue full
9	$\overline{\texttt{Q\_OUT\_NEAR\_FULL}}$	Output-queue almost full
10	$\overline{\texttt{Q\_IN\_EMPTY}}$	Input-queue empty; superseded by VALID_NEXT_DATA
11	$\overline{\texttt{Q\_IN\_NEAR\_EMPTY}}$	Input-queue almost empty
12	LOAD_DONE	Toggled to indicate completion of an operation. Mode dependent
13	DONE	FPGA has a valid configuration; Kestrel system unusable if not set.
14	VALID_NEXT_DATA	array controller's input queue empty
15	STOP_ON_QFULL	array controller stalled on full output queue

Table 3.14: Top byte of the status register.

detect program completion.

To detect an end of program, the Kestrel assembler adds a breakpoint instruction to the last instruction of every program. The host then compares the address of the breakpoint instruction with the end-of-program address. To determine the breakpoint instruction's address, the host switches the array controller to DIAGNOSTIC mode, and reads the current PC from the status register. Table 3.16 shows the values supplied from the status register's lower byte while in DIAGNOSTIC mode, controlled by the D_OUT[2:0] instruction field.

Bit	Name	Description
0	CNT_STACK_OFLO	Loop-counter-stack overflow
1	CNT_STACK_UFLO	Loop-counter-stack underflow
2	PC_STACK_OFLO	PC-stack overflow
3	PC_STACK_UFLO	PC-stack underflow
4	BREAK_INT	Instruction breakpoint reached
5-7	Currently unused and always zero	

Table 3.15: Bottom byte of status register when not in diagnostic mode.

D_OUT[2:0]	Value	Description
000	BS_OUT[7:0]	Wired-OR bit-shifter contents
001	SCR_OIT[7:0]	Scratch-register contents
010	CNTR_Q[3:0]	Number of elements on loop-counter stack
011	PC_Q[3:0]	Number of elements on program-counter stack
100	COUNT[7:0]	Contents of loop-counter top of stack
101	COUNT[15:8]	
110	PC_OUT[7:0]	Current program counter
111	PC_OUT[15:8]	

Table 3.16: Lower byte of status register in DIAGNOSTIC mode, controlled by the D_OUT[2:0] instruction field.

3.6.3 Problems and Improvements

The selector for the status register's lower byte in DIAGNOSTIC mode comes from three instruction bits, D_OUT[2:0]. While this causes no problems for the already wasteful and under-used 96-bit instruction format, a future board revision should use a more compact 64-bit instruction encoding. In this case, the D_OUT[2:0] field should be moved to the command register, using two of the unused bits, with third made available after redesigning the array-controller's queue interface.

Connecting the output queue's empty flag to the status register would simplify queue handling after program-execution halts. The change would allow the host to empty the output queue when not full without reprogramming the almost-full threshold to act as the empty flag. Section 3.7.6 describes the current procedure for determining when all output data has been read after a program completes.

3.7 Input/Output Data Queues

The data queues provide a 4-KByte buffer between the Kestrel array and the PCI bus, allowing block data transfers between the host and array. The queues are the primary interrupt source during normal program execution, signaling when the Kestrel array exhausts the input queue or fills the output queue. In these cases the host moves data between the queues and user-defined data files. Other interrupt sources usually result in program termination unless running the interactive debugger.

Unfortunately, the original array-controller design overlooked several nuances of queue operations, necessitating workarounds to prevent the addition or removal of bytes from the data streams. Consequently, queue operations often require extra data movement and the use of additional command and status bits.

3.7.1 Command and Status Register Fields

The bits pertaining to the queues in the command and status registers, summarized in Table 3.17, provide the current state of the queues as well as workarounds for problems in the array controller. The $\overline{\texttt{Q\_OUT\_FULL}}$ and $\overline{\texttt{Q\_IN\_EMPTY}}$ flags indicate a full output queue and an empty input queue, and the $\overline{\texttt{Q\_OUT\_NEAR\_FULL}}$ and $\overline{\texttt{Q\_IN\_NEAR\_EMPTY}}$ flags indicate an almost-full output queue or an almost-empty input queue based on software-programmable thresholds. The array controller generates interrupts for the almost empty and full flags, but does not halt execution until the output queue fills or the input queue empties. The controller stops when the input queue empties, even if the Kestrel program requires no more data, so the host must pad the program's input data with a extra byte to allow the program to continue after exhausting the input stream.

Bit	Name	Description
	Status Register	
8	$\overline{\texttt{Q\_OUT\_FULL}}$	Output-queue full.
9	$\overline{\texttt{Q\_OUT\_NEAR\_FULL}}$	Output-queue almost full.
10	$\overline{\texttt{Q\_IN\_EMPTY}}$	Input-queue empty; superseded by VALID_NEXT_DATA
11	$\overline{\texttt{Q\_IN\_NEAR\_EMPTY}}$	Input-queue almost empty.
14	VALID_NEXT_DATA	Array controller's input queue empty.
15	STOP_ON_QFULL	Array controller halted on full output queue.
	Command Register	
11	QIN_LOAD	Read element from input queue.
12	RBQ_OUT	Flush array-controller output buffer.

Table 3.17: Fields related to queue operations from the command and status registers.

The array controller's VALID_NEXT_DATA flag supersedes the $\overline{\texttt{Q\_IN\_EMPTY}}$ flag, and when used in conjunction with the QIN_LOAD command bit, provides a workaround to an off-by-one problem reading data from the input queue, as discussed below. The host uses the STOP_ON_QFULL flag to check if the output queue filled while it read data during processing of an $\overline{\texttt{Q\_OUT\_NEAR\_FULL}}$ condition. Finally, the RBQ_OUT command bit prevents data loss when the array controller halts for a full output queue. Both problems are discussed in more detail below.

3.7.2 Programmable Flags

The $\overline{\texttt{Q\_OUT\_NEAR\_FULL}}$ and $\overline{\texttt{Q\_IN\_NEAR\_EMPTY}}$ flags signal that the host should service the queues soon to prevent a controller halt. The threshold for these flags can be programmed by writing to the queues in a special mode that puts the data in configuration registers. Table 3.18 shows the format of the configuration registers for both the input and output queues. Each queue has a set of independent configuration registers, and au-

tomatically advances to the next location after each write. The input queue asserts the $\overline{\texttt{Q\_IN\_NEAR\_EMPTY}}$ flag when the number of elements in the queue falls below the almost-empty threshold, and the output queue asserts the $\overline{\texttt{Q\_OUT\_NEAR\_FULL}}$ when the number of empty slots in the output queue falls below the almost-full threshold.

Writes to the `PROGRAM_FIFO` address space through the PCI interface go directly into the input queue's configuration register. Writes to the output queue's configuration registers must be done using the array-controller's `PROGRAM_FIFO` mode. The host loads the configuration bytes into the input queue and executes move instructions that copy data from the input queue to the output queue through the scratch register. Since `PROGRAM_FIFO` mode operates similarly to `DIAGNOSTIC` mode, the host places the move instruction in the instruction transceivers and toggles the `LOAD_TOGGLE` bit four times.

3.7.3 Reading from the Queues

The value at the queues' output is not the next element to read but the previous element, so when data goes into an empty queue, the first byte does not appear at the output until after the first read. The original controller design assumed that the input queue's output was the next byte, so the first input read produces the wrong data. To work around this problem, the host must force a read after filling an empty input queue. To

Byte	Value
0	Least-significant byte almost-empty threshold
1	Most-significant byte almost-empty threshold
2	Least-significant byte almost-full threshold
3	Most-significant byte almost-full threshold

Table 3.18: Queue configuration registers for almost-empty and almost-full thresholds. See the Cypress CY7C48X1 datasheet for more details [3].

speed the process and eliminate the need to execute a read instruction in diagnostic mode, a zero-to-one transition on the QIN_LOAD forces an input-queue read. The same problem occurs when reading the output queue, so when emptying the queues, the host reads an extra byte and discards the first.

3.7.4 Input-Queue Handling

When the input queue becomes empty and asserts the $\overline{\texttt{Q\_IN\_EMPTY}}$ flag, the byte still on the queue's output has yet to be used by the controller. Consequently, the controller does not need to halt until the Kestrel program performs another input-queue read. The controller produces the VALID_NEXT_DATA that supersedes the $\overline{\texttt{Q\_IN\_EMPTY}}$ flag, indicating when the controller actually exhausts all input data.

An input-queue almost-empty interrupt signals the host to add more data to the input queue, but Kestrel program execution continues. While the host adds data to the input queue, the Kestrel program can simultaneously read data. If the program reads data faster than the host adds data, the controller will halt when the queue empties. When the host continues to add data, the input queue clears the $\overline{\texttt{Q\_IN\_EMPTY}}$ flag in the status register, no longer indicating that the controller stopped. The host only knows when the controller halted program execution when this flag is asserted, so the host will not know to perform a restart. Fortunately, the VALID_NEXT_DATA flag is not updated until the controller restarts execution, so the host uses the QIN_LOAD bit to advance the first byte into the controller and restarts the program.

3.7.5 Output Queue Handling

Host handling of an almost-full output-queue condition has complications similar to the input queue. While the host drains the output queue, the Kestrel program can continue adding data, and if it adds data faster than the host can remove it, the controller halts. However, the host keeps removing data, clearing the $\overline{\texttt{Q\_OUT\_FULL}}$ flag. When the host reads the status register, no flags indicate that program execution stopped, and a deadlock occurs. To overcome the problem, the controller asserts the `STOP_ON_QFULL` signal when it halts for a full output queue, and keeps the signal asserted until the host switches the controller to `STOP` mode.

As discussed in Section 3.4, the controller has no pipeline registers, so if an output-queue full forces a halt, instructions already in the pipeline still complete. If the instructions write data to the output queue, the bytes will be lost as the queue ignores writes when full. Thus, the controller must save any output bytes internally. After handling a full-queue condition, the host must extract the data using a zero-to-one transition on the `RBQ_OUT` command-register bit, causing the controller to write between zero and three bytes to the output queue.

3.7.6 Flushing the Queues

When a program completes, it leaves an unknown amount of data in the output queue that the host machine must retrieve and append to the output stream. This situation differs from output-queue handling at other times, since the queue-full flag gives the exact number of output bytes to read. Unfortunately, the output-queue empty flag does not connect to the status register, so as the host reads output-queue data, it cannot tell when

the queue empties. Instead, the host must reprogram the almost-full threshold to act as an empty flag.

As discussed in Section 3.7.2, programming the output queue requires copying data from the input queue in PROGRAM_FIFO mode, so the draining the output queue also requires draining the input queue. The remaining input data could be discarded, since the program already completed, but leftover data often indicates a bug in the Kestrel program, and the number of unused input bytes should be reported to the user.

To flush both queues, the host first switches the controller to DIAGNOSTIC mode and repeatedly executes an instruction that copies input data to the output queue. Once the input queue empties, the host records the number of bytes n transfered, switches the controller to PROGRAM_FIFO mode, loads the four data bytes to the input queue, and writes them to the output queue's configuration register. With the almost-full flag acting as an empty flag, the host reads the remaining output-queue data, the last n bytes of which are input data. The host must watch the $\overline{\text{Q\_OUT\_FULL}}$ signal so if the combined data in the input and output queue does not fit in the output queue, the host must empty the output queue while moving data from the input to the output queue.

3.7.7 Problems and Improvements

Handling of the almost-full output-queue condition does not work properly under all conditions, and the current system ignores the flag. The problem may be that the controller triggers saving of data when the output queue fills at a slightly different time than the controller decides to halt. If the full signal becomes deasserted between the two events, the output stream becomes corrupted as the three saved bytes never make it to the

output queue.

When a Kestrel program uses almost all of its input data so that the input queue constantly asserts the almost-empty flag, the controller always generates an interrupt to request more input data. However, the host has no more data to supply to the board, so the interrupt handler does nothing. If the Kestrel program performs a long calculation after using nearly all of its input data, the host will waste significant amounts of time handling useless interrupts. To fix this problem, the host should reprogram the input-queue's almost-empty flag to behave similarly to the empty flag after feeding all input data to the board.

3.8 Host Interface

The host machine controlling the Kestrel board runs a network server that accepts remote connections from the runtime environment. The server presents an interface that allows the runtime environment to access the board's hardware registers, useful for low-level operations needed during program debugging operations. The server also provides a high-level interface for efficient program execution. In the high-level interface, the runtime environment gives a program, an input stream, a destination for the output stream, and the host manages all details of program execution. When the program completes, the server sends a message to the client reporting status information.

3.8.1 Low-Level Interface

The host's low-level interface, summarized in Table 3.19, provides direct access to board hardware registers. The serial simulator also provides the low-level interface, simulating the functionality of the entire board. The interface design attempts to mimic

the communications that would occur over the PCI bus, and prototyping the interface in the simulator provided a way to develop and test the runtime environment before the board was operable.

The runtime environment sends each command as a nine-character ASCII string, with eight hexadecimal digits and a NULL terminator. The hexadecimal digits represent the address of the component and a bit indicating a read or write operation. Setting the most-significant bit of the 32-bit number indicates a write, and clearing the bit indicates a read. When performing a write operation, the address string is followed by another string containing the data, and when performing the read, the host transmits a 9-byte string with the result. For example, a write of the data `0x12345678` to the least-significant word of the instruction transceivers would send `80000004` and `12345678`, each with a null terminator.

The three data-transfer commands provide block data movement between the host and the runtime environment. The commands transfer data in binary chunks, foregoing the inefficiency of a hexadecimal ASCII encoding. The `READ_QOUT` and `WRITE_QIN` read and write data to the output and input queues, respectively. The commands should always set the write bit, with the data string indicating the block-transfer size. The `FLUSH_QUEUES` command flushes and returns the data in both the input and output queues, as described in Section 3.7.6. The write flag should always be set, with the data representing a boolean that signals if an extra output-queue read should be performed to synchronize the queue read, addressing the issue discussed in Section 3.7.3.

To execute a program using the low-level interface, the runtime environment performs all initialization, such as loading program memory, uses the command register to begin program execution, and sends the `ENABLE_INTERRUPTS` command to enable inter-

Constant	Number	Description
Direct-Access Commands		
STATUS_REG_ADDRESS	0	Read status register.
QIN_ADDRESS	1	Write byte to input queue.
QOUT_ADDRESS	2	Read byte from output queue.
CMD_REG_ADDRESS	3	Write command register.
T1_ADDRESS	4	Read/Write instruction-transceivers bits 0-31.
T2_ADDRESS	5	Read/Write instruction-transceivers bits 32-63.
T3_ADDRESS	6	Read/Write instruction-transceivers bits 64-95.
PROG_FIFO_ADDRESS	7	Program input-queue configuration.
PROG_DELAY_LINES	8	Program clock generator.
ENABLE_INTERRUPTS	9	Enable PLX 9050 interrupt generation.
Block Data-Transfer Commands		
FLUSH_QUEUES	10	Retrieve contents of input and output queues.
READ_QOUT	11	Read a block of data from output queue.
WRITE_QIN	12	Write a block of data to input queue.

Table 3.19: Low-level interface between the Kestrel host machine or simulator and the runtime environment.

rupt generation, which writes to the PLX 9050's interrupt control register, as discussed in Section 3.2.2. The Kestrel program runs until an interrupt occurs and the host's interrupt handler sends an `IRQ` message, as defined in Table 3.20, to the runtime environment for processing. Since the host does not handle the interrupts locally, the network communication required to handle each interrupt makes program execution extremely slow.

Signal	Definition
IRQ	"FFFFFFFF"
SYNC_SIGNAL	"EEEEEEEE"
TERMINATE_SIGNAL	"DDDDDDDD"
UPDATE_SIGNAL	"CCCCCCCC"
ERROR_SIGNAL	"AAAAAAAA"

Table 3.20: Status signals sent by the host to the runtime environment. The low-level interface uses only the `IRQ` message, and the high-level interface generates the remainder.

3.8.2 High-Level Interface

The host's high-level interface frees the runtime environment from all low-level details, and eliminates all network overhead during program execution by handling interrupts on the host. The runtime environment provides a program, an input data stream, a destination for output data, and starts the program. Once execution begins, the host performs all interrupt handling and data management, sending the runtime environment a message with status information when the program finishes.

Table 3.21 lists the high-level commands provided by the host server, with Appendix B giving the data format details. To run a program, the runtime environment supplies object code with `KCMD_SPECIFY_PROGRAM`, input data or file name with `KCMD_SPECIFY_QIN`, a location for output data with `KCMD_SPECIFY_QOUT`, optionally programs the queue almost-full and almost-empty thresholds with `KCMD_PROG_QOUT` and `KCMD_PROG_QIN`, respectively, and starts the program with the `KCMD_RUN_PROGRAM` command. The runtime environment sends a `KCMD_SPECIFY_QIN` command for each input file. The host can send an `ERROR_SIGNAL` signal at anytime if an error occurs; the runtime environment uses the `KCMD_GET_ERROR_CODE` command to determine the cause, with possible errors listed in Table 3.22.

While the Kestrel program runs, the host handles all interrupts, only sending an `UPDATE_SIGNAL` message periodically based on data movement between the host and Kestrel board. The runtime environment prints a dot whenever it receives this signal, helping the user detect if their program fails to run properly. When the program completes, the host sends a `TERMINATE_SIGNAL` followed by information about the program, including the status register contents, amount of input data used, and elapsed time. If the user omitted the

Constant	Number	Description
KCMD_SPECIFY_PROGRAM	13	Load Kestrel program into instruction memory.
KCMD_SPECIFY_QIN	14	Specify source for input data.
KCMD_SPECIFY_QOUT	15	Specify destination for output data.
KCMD_RETRIEVE_QOUT	16	Download program output if not sent to a file.
KCMD_PROG_QIN	17	Program input-queue almost-empty threshold.
KCMD_PROG_QOUT	18	Program output-queue almost-full threshold.
KCMD_RUN_PROGRAM	19	Begin program execution.
KCMD_GET_ERROR_CODE	20	Learn the cause of an ERROR_SIGNAL.
KCMD_TERMINATE_PROGRAM	21	Reset server for another program.

Table 3.21: Host high-level command for Kestrel-program execution.

Constant	Number	Description
	0	No error occurred
NOMEM	1	Host out of memory
UNKNOWN_DATA_SOURCE	2	Bad data source/destination type
INPUT_EXHAUSTED	3	Kestrel program exhausted input data
OPEN_FILE_FAILED	4	Could not open file
FILE_IO_ERROR	5	Error reading/writing to file

Table 3.22: Error conditions returned by the KCMD_GET_ERROR_CODE message.

-oremote option, discussed in the next section, the runtime environment must retrieve the output data from the host using the KCMD_RETRIEVE_QOUT command and save the data to a file. Finally, the runtime environment gives the KCMD_TERMINATE_PROGRAM to reset the host for another program.

3.8.3 Problems and Improvements

The interface design between the runtime environment and host is adequate, and should be expandable in the future. The largest improvements will come from minimizing synchronized network traffic where the runtime environment sends a request and waits

for a response. The first improvement would come from including debugging support in the host server that would eliminate the need for the low-level interface, vastly improving debugger performance and allowing debugging of all programs regardless of their size. Debugger support would include single-step, continue, and retrieve-array-state commands that are currently implemented in the runtime environment. Also, the `PROG_DELAY_LINES` command should be removed from the low-level interface, as the clock generator should only be programmed directly by the host.

The simulator supports only the basic low-level commands and only the `READ_QOUT` block-transfer command. Supporting the high-level commands in the simulator would make the host server and simulator have identical interfaces, but should only be done once the high-level interface supports program debugging—otherwise code will be duplicated in the runtime environment and simulator, complicating program maintenance. When the high-level interface supports debugging, the low-level interface will no longer be needed, and the corresponding control code could be removed from the runtime environment.

3.9 User Interface

The user interface for the Kestrel systems takes an assembled Kestrel program and an input data stream, runs the program, and stores the program's output data in a user-specified output file. The user is responsible for producing the input file, ensuring the input data ordering matches the program data reads. The output file contains the raw output stream, and usually requires a postprocessing step to extract or format the data into its final form. Kestrel program development usually requires writing three programs: the Kestrel program, an preprocessor to create or format the input file stream, and a postprocessor to

format or utilize the results.

3.9.1 Runtime Environment

The runtime environment provides a uniform frontend for Kestrel program execution on either the hardware or in a simulator. The basic command line takes the assembled Kestrel program, a set of input files, and an output file:

```
kestrel <options> {-s|-b} <program> <input(s)> <output> [# PEs]
```

where `-s` invokes the simulator and `-b` runs the program on the Kestrel hardware, `<program>` is the Kestrel object file, `<input(s)>` is a set of one or more input files, `<output>` is the output file, and `[# PEs]`, used only with `-s`, is the number of PEs to simulate.

Table 3.23 lists the general command-line options for the runtime environment. The `-debug` option starts the debugger that allows interactive program execution using single stepping, breakpoints, and the downloading of internal Kestrel array values for viewing by the user. As discussed in Section 3.8, the current system has two levels of interaction between the runtime environment and the host, consisting of a low-level interface to the kestrel board's register, and a high-level interface for efficient program execution by the host. As the high-level interface does not support program debugging operations, the debugger uses the low-level commands to access board registers. Program execution in the

Option	Description
`-debug`	Invoke in debugger mode
`-slow`	Force use of low-level host interface
`-showirq`	Display message for each interrupt received
`-noloadi`	No program memory loading before execution (useful for hardware testing only)

Table 3.23: General runtime-environment command-line options.

debugger is slow as the debugger must also handle interrupt processing over the network connection between the host and UNIX machines.

The remaining three options primarily served as debugging aids while the Kestrel system was debugged, and are now not very useful. The `-slow` option forces the runtime environment to always use the slower low-level interface, and the `-showirq` prints a message every time the runtime environment handles an interrupt while using the low-level interface. The `-noloadi` prevents the runtime environment from writing the program to the board's instruction memory, and was briefly useful for debugging a hardware problem.

Simulator or Board

The simulator, invoked with the `-s` option, performs a serialized simulation, computing results one processor at a time. Since simulating a full 512-PE array is slow, the simulator is most useful when simulating a small number of PEs, as is often useful during the early stages of program development. The simulator uses the low-level interface used by the debugger, so interactive debugging is slow when used with the simulator, as all interrupt handling must be done over the network. Using the Kestrel hardware with the `-b` option, the runtime environment finds an unused Kestrel board and launches the program. If all systems are in use, the runtime environment quits, forcing the user to retry.

Program Object File

The `program` field specifies the object file generated by the Kestrel assembler [11]. The object file is an ASCII file that contains Kestrel instructions stored as 24-bit hexadecimal numbers, one per line. The object file can also contain debugging information, relating each instruction to a line in the source file.

Input/Output Files

The `<input(s)>` and `<output>` fields specify the input files and output file, respectively. There are several options available for formatting both input and output files. The input stream can be spread over multiple files, and files can be local to either the UNIX or Kestrel host machine. The data can be encoded as either raw binary or ASCII with one data byte per line.

Figure 3.13 shows the input and output file specification format. Multiple input files are supported, with the overall input stream defined as the concatenation of all the files. An input stream spread over multiple files is useful for sequence analysis applications whose input starts with a short query sequence or model, followed by a large database of sequences. Spreading the input over multiple files eliminates the extra step of merging the files before execution.

The `-iremote` and `-oremote` options indicate that the corresponding file is located on the host machine, and is useful for storing a large database such as those used during sequence analysis. Using remote files eliminates network communication overhead, and reduces host memory requirements for large files. If a file is not specified as remote, the runtime environment transmits the input file over the network before program execution

```
<input(s)> = <iopts> -in file1 <iopts> -in file2 ... <iopts> -in fileN
<iopts> = [[-iremote]|[-iformat {binary|hex|octal|decimal}]]
<output> = <oopts> file
<oopts> = [[-oremote]|[-oformat {binary|hex|octal|decimal}]]
```

Figure 3.13: Command-line format for input and output files. The `-iremote/oremote` and `-iformat/oformat` options are mutually exclusive, as remote files accessed directly by the server are always binary. The options are not sticky, applying to each file separately.

begins, and the host stores the program in memory. Consequently, the host must have enough memory and swap space available to store the data, and the network communication time does not overlap with Kestrel program execution. Remote files should be used for input or output files larger than a few megabytes, as data is read from the file as needed and much of the I/O time overlaps Kestrel program execution.

For example, if a file is located on host machine, then that file would be specified as:

```
-iremote -in C:\\data\\swissprot.dat
```

accessing the file `C:\data\swissprot.dat`, with the double back quotes to escape shell processing. The host machines also have access to the network filesystem, and can access files directly over the network. For example, the file specification:

```
-iremote -in \\\\fs\\kestrel\\database\\swissprot.dat
```

accesses the UNIX file `/projects/kestrel/database/swissprot.dat`, again with the extra back quotes used to escape shell processing[1]. However, the user account used to start the server on the host machine must have the appropriate access permissions to access the file.

The `-iformat` and `-oformat` options allow different formats for input and output data. The data can be either raw binary or an ASCII file with one data element per line, formatted as hexadecimal, decimal, or octal. The ASCII formats are incompatible with the `-iremote` or `-oremote` options, as the host machine always interprets files as binary. Performing ASCII conversion on the host makes little sense, as remote files are most useful for large data files where ASCII encoding would only make files larger.

[1] This example assumes `csh`, and quoting requirements may be different for different shells.

3.9.2 Application Programmer's Interface

The user can also access the Kestrel host server from within other applications using a set of library functions, summarized in Table 3.24[2] . The library functions handle all details of communicating with the host server's network interface, as described in Section 3.8.2, allowing the user to run batch jobs without the overhead of invoking the runtime environment.

Figure 3.14 shows an example program that uses the application programmer's interface. First, the program connects to the Kestrel server with the `InitializeSocket(int machine)` function, where `machine` is `MACHINE_ANY`, `MACHINE_MERLIN`, or `MACHINE_MARSH`, indicating which host server to connect to. The function returns a file descriptor that must be passed to as the first argument to all subsequent library calls. Next, the `TransmitProgram(int fd, char *file_name)` function reads a Kestrel executable and transmits it to the server. The `TransmitInputData(int fd, char *file_name, int remote, char *buffer, int size)` function configures the data input stream. If `file_name` is nonzero and `remote` = 0, then the function reads binary data from the specified file and transmits it to the Kestrel server. If `file_name` is nonzero and `remote` = 1, then the function transmits only the file name, emulating the behavior of the runtime environment's `-iremote` option. If `file_name` is zero, then the last two arguments specify a buffer for which to take the data. This function can be called repeatedly for input streams spread over multiple files.

The `ConfigureOutputData(int fd, char *file_name)` function indicates the destination for the output data. If `file_name` is nonzero, then the host server writes di-

[2]The directory `/projects/kestrel/kservlib` contains the source code for these functions and example programs of their use.

Function	Purpose
InitializeSocket	Open connection to Kestrel server
TransmitProgram	Load Kestrel program
TransmitInputData	Provide input data or file name
ConfigureOutputData	Provide destination for output data
RunProgram	Run Kestrel program
GetServerQoutBuffer	Retrieve output data if no remote file
ResetServer	Reset Kestrel server, prepare for new program
CloseSocket	Close connection to Kestrel server

Table 3.24: Application programmer's interface to the Kestrel host server. These functions handle all details of communicating with the host server's network interface.

```
#include <stdio.h>
#include <stdlib.h>

#include "kservlib.h"

int main(int argc, char **argv)
{
  int indata_size, outdata_size, status, socket;

  if (!(socket = InitializeSocket(MACHINE_ANY))) {
    return (99);
  }
  TransmitProgram(socket, argv[1]);
  TransmitInputData(socket, argv[2], 0, NULL, 0);
  ConfigureOutputData(socket, NULL);
  RunProgram(socket, &indata_size, &outdata_size, &status);
  GetServerQoutBuffer(socket, argv[3], NULL);
  ResetServer(socket);
  CloseSocket(socket);

  return 0;
}
```

Figure 3.14: A simple example program that demonstrates the use of the application programmer's interface to the Kestrel host server. The program takes three command line arguments: the Kestrel object file, input data file, and output data file.

rectly to the output file, emulating the behavior of the runtime environment's `-oremote` option. If `file_name` is zero, then the host collects the output data in a buffer which the program retrieves with the `GetServerQoutBuffer(int fd, char *file_name, char *buffer)` function after the program completes. If `file_name` is nonzero, then output data is written to the indicated file. If `file_name` is zero, than the data is written to a buffer pointed to by `buffer` that must be large enough to hold all the output data.

The `RunProgram(int fd, int *used_in, int *out_data, int *status)` function runs the Kestrel program. After the program completes, the return values in `user_in`, `out_data`, and `status` indicate the number of input bytes used, the number of output bytes, and the contents of the status register upon completion. The `ResetServer(int fd)` function resets the server, and the `CloseSocket(int fd)` closes the network connection, allowing other clients to connect to the server. After reseting the server, another Kestrel program can be run without closing and reopening the network connection.

3.9.3 Programs and Improvements

While the current system is adequate for simple program execution, future projects that use Kestrel as one step in a long process will require a more sophisticated interface. The addition of a job queue would allows users or programs to start a job regardless of whether any machine is available. Also, generating all the input data or saving all output data in a file may become prohibitive for large problems with data sizes in the gigabyte range. The ability to generate the input or process the output as the Kestrel program runs could greatly simplify integration of Kestrel as a step in other computational processing.

The user interface for the debugger is primitive and extremely difficult to use. The

underlying machinery for running programs and downloading internal array state information during debugging is sound, but an improved interface could make it more usable. Also, since the debugger does not support remote files stored on the host, using the debugger with large input data streams is impossible. Interrupt handling over the network requires too much time, since each 4 KBytes of program input or output requires several synchronized interactions between the host and runtime environment.

The current serial simulator does not correctly simulate the current implementation of the array controller. The simulator does not model the pipeline delays of the data movement within the controller, so programs that explicitly use the scratch register will not behave identically when run on the board and simulator.

3.10 Summary: Running a Kestrel Program

This section summarizes the normal execution of a Kestrel program by the runtime environment using the host's high-level interface. It presents actions performed by the user, runtime environment, and host, linking together details provided by this chapter.

3.10.1 Board Initialization

The host must initialize the PCI-bus interface and the clock generator before running a Kestrel program, as described in Sections 3.2 and 3.3, respectively. The clock initialization should be performed as soon after starting the machine as possible, as the default settings cause overlapping clock phases that result in excessive power consumption in the Kestrel array.

Starting a Program

After the user starts a Kestrel program with a runtime environment, specifying the object file, input stream, and output stream, and connects to an available Kestrel host, the runtime environment performs a sequence of high-level commands to prepare the Kestrel board for program execution. When the host server opens a new connection, it enables the array controller by setting the command register's `FPGA_RESET` bit.

The runtime environment downloads the object file to the host with the `KCMD_SPECIFY_PROGRAM` command, and the host loads the program into program memory and initializes the program counter, as described in Section 3.5.1. The `KCMD_SPECIFY_QIN` and `KCMD_SPECIFY_QOUT` commands initialize the input and output streams, respectively, verifying that each input file exists and determining the total number of input bytes. The runtime environment currently does not reprogram the almost-queue flags, using the default thresholds of 1024 for both the almost-empty and almost-full flags. The runtime environment starts the program by issuing the `KCMD_RUN_PROGRAM` command, causing the host to put the controller into `RUN` mode and enable the PLX 9050's interrupt generation.

3.10.2 Interrupt Handler

Once Kestrel program execution begins, the host interacts with the Kestrel board solely through the interrupt handler, while the runtime environment waits for messages from the host. Each time the board triggers an interrupt, the handler reads the status register to determine the cause. The two primary causes for interrupts are for queue handling and program termination. For queue handling, the host moves data between the Kestrel board and the user's input and output files, and if the array controller halted because it

ran out of data, then the host performs a restart. Section 3.7 presents a detailed description of queue operations. Error conditions that can cause program termination include underflowing or overflowing the loop- and program-counter stacks, or reaching a program breakpoint. During normal program execution, a breakpoint signifies the end of program when the instruction is the program's last. When the program finishes, the host transmits the `TERMINATE_SIGNAL` to the runtime environment, along with status information.

3.10.3 Program Termination

After the runtime environment receives the `TERMINATE_SIGNAL`, it displays status information to user about why the program terminated. If the output file was not remote, created directly by the host, the runtime environment performs a `KCMD_RETRIEVE_QOUT` to copy the output data from the host and save it to a file. The runtime environment gives a `KCMD_TERMINATE_PROGRAM` to reset to server for the next program.

Chapter 4

Evaluation and Future Work

This chapter concludes by presenting an evaluation of the current system, a comparison to contemporary SIMD machines, and directions for future work. Section 4.1 presents the performance of the target biological sequence analysis applications. Section 4.2 presents a survey of contemporary SIMD machines, and Section 4.3 summarizes the changes that should be made to the next revision of the Kestrel system. Finally, Section 4.4 gives directions for future work.

4.1 Applications

This section presents existing sequence analysis applications for the Kestrel system, including edit distance, Smith and Waterman, and HMM scoring. While edit distance is not used for biological sequence analysis, it is the simplest and contains all the features of the more complicated dynamic-programming algorithms. These algorithms score a single sequence or model versus a large database, and produce a score rating the similarity between the query sequence or model and every sequence in the database. The current Kestrel

system performs sequence scoring 10 to 20 times faster than a fast serial implementation, but does not perform as well as a workstation on sequence alignment, because the dynamic-programming traceback phase does not parallelize well.

4.1.1 Edit Distance

The edit distance between two character strings is the minimum number of character insertions and deletions required to transform one string into another. The results of the edit distance calculation can be used to generate the actual sequence of operations by running the algorithm backwards along the minimum cost path determined during the forward pass. The edit distance between two strings a and b can be found by dynamic programming using the recurrence:

$$c_{i,j} = \min \begin{cases} c_{i-1,j-1} + \text{dist}(a_i, b_j) & \text{match} \\ c_{i-1,j} + \text{dist}(a_i, \phi) & \text{insert} \\ c_{i,j-1} + \text{dist}(\phi, b_j) & \text{delete,} \end{cases} \tag{4.1}$$

where $\text{dist}(a_i, b_j)$ is the cost of matching a_i to b_j, $\text{dist}(a_i, \phi)$ is the gap cost of not matching a_i to any character in b, and $\text{dist}(\phi, b_j)$ is the gap cost of not matching b_j to any character in a. Edit distance is calculated by setting $\text{dist}(a_j, \phi) = \text{dist}(\phi, b_j) = 1$, and $\text{dist}(a_i, b_j) = 0$ if $a_i = b_j$ or 2 otherwise.

The dynamic-programming solution to edit distance constructs an m by n matrix, where m and n are the lengths of a and b, respectively [2]. The elements in the matrix are computed using Equation 4.1 based on the elements to the left, top, and left-top of the current element as shown in Figure 4.1 (a). The minimum of the three choices is saved in each

element along with information indicating the choice resulting in the minimum. When the matrix is filled, element $c_{m,n}$ contains the minimum edit distance cost. Figure 4.1 (b) shows an example dynamic-programming matrix for the edit distance between the sequences A and B, which contain ϕABCD and ϕACBFCE, respectively. Each element in the matrix in Figure 4.1 (b) contains both the minimum cost and an arrow indicating which adjacent element the algorithm used to calculate the cost. The ϕ character at the start of each sequence ensures the first row and column always choose the insert or delete case, respectively.

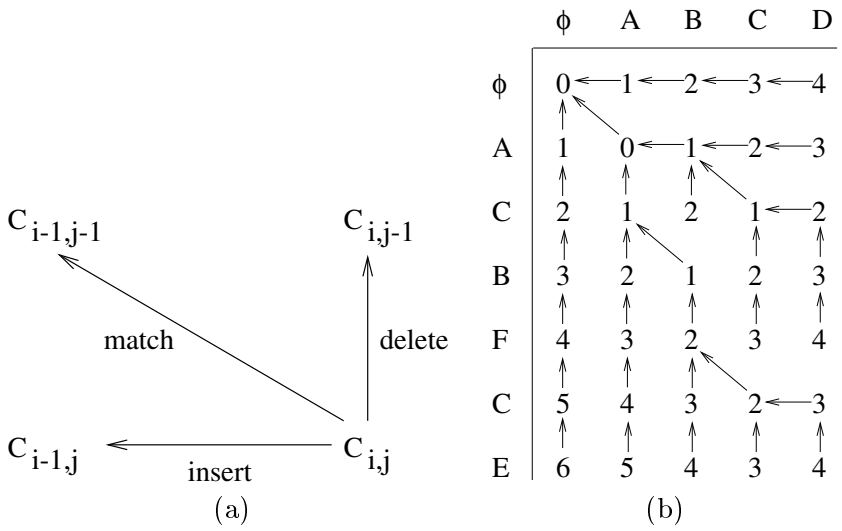

Figure 4.1: (a) Illustration of the data dependencies for the calculation of an element in the edit distance dynamic-programming matrix. (b) Dynamic-programming matrix for the edit distance between the strings ϕABCD and ϕACBFCE. Each element in the matrix contains both a cost and an arrow indicating which adjacent element the algorithm used to calculate the cost.

Edit distance determines the sequences of operations that transforms one sequence into another by performing traceback on the dynamic-programming matrix. Starting at element $c_{m,n}$ of the dynamic-programming matrix, traceback follows the sequence of arrows that resulted in the minimum edit distance cost. Though only a few of these elements are

used during traceback, all the choices must be saved in the matrix, because only when the last element in the matrix is computed does the algorithm know what parts of the matrix will contribute to the minimum-cost operation sequence.

4.1.2 Smith and Waterman

The Smith and Waterman [19] dynamic-programming algorithm with affine gap costs, implemented by Leslie Grate, Justin Meyers, and Rachel Karchin, is conceptually the same as edit distance except the recurrence relation is more complex to support affine gap costs and local alignment:

$$
\begin{aligned}
M(i,j) &= \min \begin{cases} 0, \\ M(i-1,j-1) + s(x_i, y_j), \\ I(i-1,j-1) + s(x_i, y_j), \\ D(i-1,j-1) + s(x_i, y_j); \end{cases} \\
I(i,j) &= \min \begin{cases} M(i-1,j) - d, \\ I(i-1,j) - e; \end{cases} \\
D(i,j) &= \min \begin{cases} M(i,j-1) - d, \\ D(i,j-1) - e; \end{cases}
\end{aligned}
\tag{4.2}
$$

where $-d-(g-1)e$ is the affine gap cost, d is the gap opening cost, e is the gap extension cost, g is the gap length, and $s(x_i, y_j)$ is the substitution score of character x_i from first sequence with character y_j from the second sequence. Each element in the dynamic-programming

matrix holds three values, representing the matching states $M(i,j)$, inserting states $I(i,j)$, and deleting states $D(i,j)$. The delete state $D(i,j)$ and insert state $I(i,j)$ equations reflect the cost of opening a gap d when moving from a match to insert or delete state, and the cost of extending a gap e when continuing to insert or delete characters. The match state $M(i,j)$ reflects that a match always has the same cost whether it follows another match or closes a gap (there is no gap close penalty). The zero term in $M(i,j)$ is the term for local alignment, allowing the score to be reset when it drops below zero.

4.1.3 Viterbi HMM Scoring

The preceding dynamic-programming algorithms can be generalized using HMMs of the form shown in Figure 4.2, allowing position dependent-transition and character-emission costs. The states of the HMM are grouped into *fat states* as shown in Figure 4.2, with each fat state corresponding to a column in the dynamic-programming matrix. The transitions between fat states, as well as the insert node, is called a `fat edge`. The rectangular boxes are match states that match a single position from the model with a character, and diamonds are insert states that model the insertion of random characters between two alignment positions. Both match and insert states have a table of emission probabilities that represent the probabilities or costs of emitting a character given a particular character in the sequence. The circular states represent delete states that model gaps in the alignment, and arrows between states represent transitions, each with a corresponding probability.

An optimal path analogous to Equations 4.1 and 4.2 can written for the HMM in Figure 4.2. The resulting recurrence relations, called the Viterbi equations, compute the

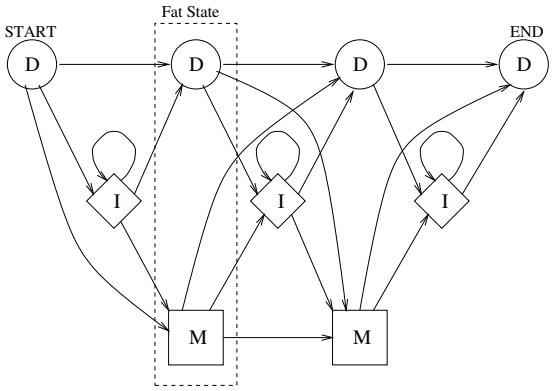

Figure 4.2: An example hidden Markov model (HMM) where circles represent delete states, diamonds represent insert states, and squares represent match states. The match and insert states have a table of emission costs based on the characters from the sequences used to construct the model. Each state has three transitions to subsequent match, insert, and delete states, each with an associated probability.

elements of a dynamic-programming matrix based on an HMM:

$$V_j^M(i) = -\log e_{M_j}(x_i) + \min \begin{cases} V_{j-1}^M(i-1) - \log a_{M_{j-1}M_j}, \\ V_{j-1}^I(i-1) - \log a_{I_{j-1}M_j}, \\ V_{j-1}^D(i-1) - \log a_{D_{j-1}M_j}; \end{cases}$$

$$V_j^I(i) = -\log e_{I_j}(x_i) + \min \begin{cases} V_j^M(i-1) - \log a_{M_jI_j}, \\ V_j^I(i-1) - \log a_{I_jI_j}, \\ V_j^D(i-1) - \log a_{D_jI_j}; \end{cases} \tag{4.3}$$

$$V_j^D(i) = \min \begin{cases} V_{j-1}^M(i) - \log a_{M_{j-1}D_j}, \\ V_{j-1}^I(i) - \log a_{I_{j-1}D_j}, \\ V_{j-1}^D(i) - \log a_{D_{j-1}D_j}. \end{cases}$$

where $e_{M_j}(x_i)$ and $e_{I_j}(x_i)$ are the emission probabilities from match or insert state j given the character x_i from the query sequence, and a_{XY} are the transition probabilities between two states connected in the model. The probability of a path through the model is the product of the probabilities of the emission and transition costs along the path. To prevent numerical underflow from multiplying a large number of small, positive numbers, the negative logarithms of the probabilities are taken and the resulting numbers added.

4.1.4 Sum-of-all-paths HMM Scoring

The Viterbi algorithm's primary drawback is that it finds only one path, ignoring those with identical or slightly smaller probabilities. To include other probable paths in the final score, the score can be computed as a sum of the probabilities of all possible paths in the alignment, to give an overall sense of how well the sequence is represented by the HMM, not just the score of the best alignment. This algorithm, called sum-of-all-paths, or the Forward algorithm, replaces the mins from Equation 4.3 with a sum of probabilities:

$$\begin{aligned}
F_j^M(i) &= \log e_{M_j}(x_i) + \log[a_{M_{j-1}M_j} \exp(F_{j-1}^M(i-1)) \\
&\qquad + a_{I_{j-1}M_j} \exp(F_{j-1}^I(i-1)) + a_{D_{j-1}M_j} \exp(F_{j-1}^D(i-1))]; \\
\\
F_j^I(i) &= \log e_{I_j}(x_i) + \log[a_{M_jI_j} \exp(F_{j-1}^M(i-1)) \\
&\qquad + a_{I_jI_j} \exp(F_{j-1}^I(i-1)) + a_{D_{j-1}I_j} \exp(F_{j-1}^D(i-1))]; \\
\\
F_j^D(i) &= \log[a_{M_{j-1}D_j} \exp(F_{j-1}^M(i-1)) + a_{I_{j-1}D_j} \exp(F_{j-1}^I(i-1)) \\
&\qquad + a_{D_{j-1}D_j} \exp(F_{j-1}^D(i-1))].
\end{aligned} \tag{4.4}$$

The most important observation about Equation 4.4 is that to compute the sum of probabilities and keep the scores as logarithms to prevent numerical problems, the scores must be transformed back to probabilities using exponentiation, added, then converted back to scores by taking the logarithm. This calculation is slow, but can be sped by manipulating the equations and using table lookups.

Implementing the sum-of-all-paths algorithm on Kestrel presents serious problems because the table lookup algorithm used in the serial implementation uses too much memory for Kestrel's small processing elements, and the exponentiation and logarithm calculations would drastically reduce performance. One possible solution is to keep the scores as probabilities, adding and multiplying directly. Converting Equation 4.4 from log-probabilities to probabilities gives:

$$
\begin{aligned}
K_j^M(i) &= e_{M_j}(x_i) \times [a_{M_{j-1}M_j} \times K_{j-1}^M(i-1) \\
&\qquad + a_{I_{j-1}M_j} \times K_{j-1}^I(i-1) + a_{D_{j-1}M_j} \times K_{j-1}^D(i-1)]; \\
K_j^I(i) &= e_{I_j}(x_i) \times [a_{M_jI_j} \times K_{j-1}^M(i-1) \\
&\qquad + a_{I_jI_j} \times K_{j-1}^I(i-1) + a_{D_{j-1}I_j} \times K_{j-1}^D(i-1)]; \\
K_j^D(i) &= a_{M_{j-1}D_j} \times K_{j-1}^M(i-1) + a_{I_{j-1}D_j} \times K_{j-1}^I(i-1) \\
&\qquad + a_{D_{j-1}D_j} \times K_{j-1}^D(i-1).
\end{aligned}
\tag{4.5}
$$

Equation 4.5 can be used, along with a floating-point format with sufficiently large exponents, to implement sum-of-all-paths HMM scoring on Kestrel.

4.1.5 Performance

Kestrel can calculate a modulo-256 edit-distance score at a rate of 3.875 instruction per character [4], and can produce the sequence of matches, inserts, and deletes of a sequence up to 2.7 million characters in length using three-level checkpoints to reduce the memory requirements of the traceback matrix [8]. The edit-distance alignment performance against a large sequence is poor as the dynamic-programming traceback does not parallelize and the fragments of the sequence must be fed to the array many times to recompute portions of the dynamic-programming matrix from a checkpoint.

Table 4.1 lists the performance of the biological sequence-scoring algorithms currently implemented on Kestrel. The Smith and Waterman and Viterbi HMM scoring are at least twenty times faster than a 433 MHz DEC Alpha. The all-paths HMM scoring algorithm is only ten times faster than the Alpha. However, the current implementation uses an inefficient floating-point format, and recent work by Kevin Karplus shows that substantial improvements can be made by keeping the scores in log-probability form and using a numerical approximation of $\log(e^a + e^b)$.

	Kestrel	433 MHz DEC alpha		
Query Size	≤ 512	32	128	512
SW	12	20	73	282
HMM (Viterbi)	41	80	232	862
HMM (all paths)	236	155	541	2120

Table 4.1: Wall time in seconds to search 10MByte of the Swissprot database on the 20 MHz Kestrel.

4.2 Architectural Comparison

There is a great diversity among SIMD processing chips, and this section attempts to compare several typical architectures, both recent and planned. For simplicity, comparisons are based on performance per chip; it could be argued that performance per gate is more appropriate because this considers pure logic rather than access to the latest technology.

As with Kestrel, each of the machines described below has its target applications areas and advantages and disadvantages. This comparison does not highlight Kestrel's independent functional units, conditional evaluation, or programming model. Table 4.2 shows estimated performance of several chips on integer addition, multiplication, and MAC (multiply accumulate) operations of various sizes in millions of operations per second (MOPS). The peak I/O bandwidth data relates to transfer between local buffer memory or data cache.

Operation	Data Bytes		Kestrel	MasPar-2	MGAP-2	MMX	CNAPS	RaPiD
	Operands	Result						
Addition	1	2	2112	100	1000	1600	960	4800
Addition	2	2	1056	100	505	800	960	4800
Addition	4	4	528	133	253	400	(480)	2400
Multiplication	1	2	2112	40	150	600	960	1600
Multiplication	2	4	352	22		400	480?	1600
MAC	1	2	1056	29	130	400	960	1600
MAC	2	4	302	18		200	480?	1600
Peak I/O MB/S			66	20?	800?	1600?	60?	>4000?

Table 4.2: Performance in MOPS of selected SIMD chips.

4.2.1 CNAPS

CNAPS is a linear SIMD array of 16-bit processing elements. Each PE has an 8×16-bit multiplier, a 16-bit ALU and a 32-bit adder, a 4 Kbyte local memory, and a

32-element 16-bit single-port register bank. CNAPS is clearly targeting MAC operations, and is the SIMD machine most similar to Kestrel. The large local memory includes support for sparse data representations. The latest chips contain 32 PEs and run at 30 MHz. An earlier 64-PE chip with 16 spares had 14 million transistors; the 32-PE chip would have around 6 million transistors, not including spares. I/O between the PEs and the controller uses shared 8-bit buses. Inter-PE communication uses a 2-bit bus, and instructions are 31 bits. Parallelism of execution for the different components allows 1 billion 16-bit MAC operations per second [9].

4.2.2 MasPar

The MasPar MP-2 is a mesh-based SIMD supercomputer-class machine with a global router. Processing elements feature 40 32-bit registers, a 32-bit ALU with functional units to aid floating-point calculation, and a 64 Kbyte off-chip, locally-addressed memory. Each chip of just under one million transistors has 32 processing elements and two router interfaces. The ALU and memory interface can operate concurrently to partially alleviate the bandwidth problem of off-chip communication. The chips have a 12.5 MHz clock, with simple operations requiring three clock cycles [1].

4.2.3 Pentium MMX

The Intel Pentium's MMX unit provides small-scale SIMD processing of 8-, 16- or 32-bit subfields of its 64 bit data. The Pentium architecture is cached and features two execution pipelines which under optimal conditions enable execution of 2 MMX instructions per cycle. The MMX unit has just 8 64-bit registers, and MMX instructions have

only two operands. To aid image processing applications, MMX includes instructions for saturate arithmetic, clamping results to the maximum or minimum possible value to avoid overflow and underflow. The table displays peak rates for the 200 MHz Pentium MMX; real performance for applications that do not fit in the 8 MMX registers or have data hazards between instructions is likely to be considerably less [12].

4.2.4 MGAP

The MGAP-2 is a very fine grained architecture. Each chip has 1536 1-bit PEs connected in an octagonal mesh. Each PE has a 16-bit dual-port local storage and two 3-input, 1-output function multiplexers for calculation. Based on published layout descriptions, the chips may have on the order of one half to two thirds of a million transistors. The configuration is stored in a local register but memory access is globally controlled. The system runs at 50 MHz. Performance is 6.8 kDCT/s (thousand discrete cosine transforms per second) (8×8 blocks of 8-bit pixels) for the MGAP-2 chip [15, 16].

4.2.5 RaPiD

The RaPiD system is the most hardware-oriented among these examples: it is a "coarse grained FPGA," with 16 cells per chip arranged in a linear array. Each cell consists of three 16-bit ALUs, one multiplier, six datapath registers and three 32 by 16-bit memory blocks and some glue logic. The interconnection between these functional units, as well as between the cells, uses ten 16-bit segmented buses. A separate control path allows dynamic scheduling of the pipeline operations and some data path flow control. Transistor estimates are not yet available. The reference's estimate of fitting 16 cells on a chip running at 100

MHz will allow close to 1.6 million DCTs/sec (8×8 blocks of 8-bit pixels) [6].

4.3 Problems and Improvements Summary

This section summarizes the most important improvements that should be made to the Kestrel system, focusing on bugs and design flaws that degrade system performance and ignoring architectural changes. Most changes concern the board and software system design.

4.3.1 Layout Design

- Invert the polarity of the bit shifter's most significant bit before using with the wired-OR, making the wired-OR compatible with nested-conditional operations. This change requires the addition of an inverter from the bit shifter's layout, but can be easily done as there is unused space in the design. The changes will also need to be made to the Verilog model, simulator, and existing applications that use the wired-OR.

- Invert the polarity of the comparator's true sign `cts_v1` by using `opC_hl1[7]` signal instead of `opC_hl1n[7]`, and is easily done as both polarities are available in the layout. This change also requires modifications to the Verilog model, simulator, and existing applications that perform signed comparisons.

- Correct the logic that controls the Kestrel chip output enable pin. The control logic is located in the pad frame near the left and right data pads. The Verilog model should also be modified to use the output enable to turn off the pads when writing data into the array during testing.

- Allow the mask flag from the adjacent PEs to be selected on the flag bus. This change requires minimal changes the existing layout, but requires changes to the Verilog model, simulator, and assembler.

- Move the two-phase clock generator to the Kestrel chips so the board uses only a single clock.

4.3.2 Board and Controller Design

- Redesign the local address decoding to use the three least-significant bits instead of the three most-significant bits. This will allow components with the same Address Descriptor registers to simultaneously use the same address space.

- Redesign the clock distribution system to achieve a 33 MHz clock rate. This will include redesigning the clock generator, altering component placement to make the clock driver adjacent to the clock generator, and reduce the overall number of components to improve routing and signal quality.

- Pipeline the Kestrel array instruction broadcast so that instruction memory reads and broadcast to the array occur in two cycles rather than one. This also requires an additional pipeline stage in the array controller, and will create a one-cycle branch delay.

- Change to a 64-bit instruction format from a current 96-bit format, reducing the number of instruction-memory chips from six to four. The change would require encoding the controller's instruction bits, and would increase the number of instructions required to perform many controller operations, but would not be a limitation for most

programs that have a small controller-to-array operation ratio.

- Correct wiring problems between the FPGA and configuration PROM, and include a header for the $\overline{\texttt{INIT}}$ signal use by the serial download cable. No FPGA pins used for configuration should also be used for data signals as this greatly complicates debugging.

- Connect the output queue's empty flag to the status register. The host can then use the empty flag to detect when all data has been read from the output queue.

- Add pipeline registers between stages in the array controller so when the controller halts for a full output queue, the instructions already in the pipeline do not complete.

- The Verilog mode, discussed in Appendix C, does not model the current controller or board design, and does not support the current assembler's output.

4.3.3 System Software Improvements

- Add debugging operations to host server's high-level interface, remove the low-level interface, simulator to support the high-level interface, and remove all global operations from the runtime environment.

- Fix reliability problems with the interface between the runtime environment and the host server. This would also include adding a job queue so the user can still in queue a program for execution even when all machines are already busy, as will be especially important to support the Web interface.

- Improve usability of the runtime environment's debugger.

- Fix bugs in the current simulator, especially pertaining to the simulation of the array controller.
- The serial simulator does not model the pipeline delays of the data movement within the array controller, so programs that explicitly use the scratch register will not behave identically when run on the board and simulator.

4.4 Future Work

Future changes and enhancements to the Kestrel architecture will be driven by application development. Implementing a variety of algorithms outside the sequence-analysis domain is currently revealing the need for new architectural features. These new features should be analyzed terms of feasibility and how many applications they can enhance. Hopefully, new features could be added to make the architecture more general-purpose while preserving its simplicity. While there are many improvements and fixes that would make the existing system faster, application development will provide most direction for future architecture research.

Additions to the existing Kestrel architecture should be done with great care. Design changes that improve the performance of many algorithms are much more likely to be worth the engineering effort than changes that only improve the performance of one or two applications. Since there is a working system, any changes in the hardware will require changes in much of the software support structure. For example, changes to the Kestrel PE will require changes to be instruction encoding that will propagate up to the software level, requiring changes in the simulator, assembler, and runtime environment.

To facilitate application development, enhancements to the system software should

make the system robust and user friendly so new algorithms can be implemented and evaluated quickly. Also, the assembly language programming environment creates a steep learning curve for those unfamiliar with the architecture. Both a high-level language and development environment would make Kestrel development more accessible, and generate more feedback on improving the architecture. Ultimately, the goal of the Kestrel project is to accelerate a wide range of data-parallelizable algorithms using a relatively simple and inexpensive system. The ability of users to use Kestrel to solve problems faster than a traditional approaches will be the ultimate indicator of the project's success.

Bibliography

[1] Donald W. Blevins et al. BLITZEN: A highly integrated massively parallel machine. *J. Parallel and Distributed Computing*, 8(2):150–160, February 1990.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[3] Cypress Semiconductor Corporation. *Cypress CY7C48X1 FIFO Datasheet*. January 15, 1997.

[4] David Dahle. *Edit Distance on Kestrel with level three checkpoints*. CS243 Class Project, Spring 1998.

[5] David M. Dahle, Jeffrey D. Hirschberg, Kevin Karplus, Hansjörg Keller, Eric Rice, Don Speck, Douglas H. Williams, and Richard Hughey. Kestrel: Design of an 8-bit SIMD parallel processor. *17th Conference on Advanced Research in VLSI*, pages 145–162, 1997.

[6] Carl Ebeling, Darren C. Conquist, and Paul Franklin. RaPiD — reconfigurable pipelined datapath. In *6th Int. Wkshp. Field-Programmable Logic and Applications*, pages 126–135, New York, 1996. Springer-Verlag.

[7] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[8] J. Alicia Grice, Richard Hughey, and Don Speck. Reduced space sequence alignment. *CABIOS*, 13(1):45–53, 1997.

[9] Dan W. Hammerstrom and Daniel P. Lulich. Image processing using one-dimensional processor arrays. *Proc. IEEE*, 84(7):1005–1018, 1996.

[10] Jeffery D. Hirschberg and Justin N. Meyer. *The Kestrel Simulator Programmer's Reference Manual, Revision 3.* UC Santa Cruz, February 26, 1998.

[11] Richard Hughey. *KASM Assembly Manual.* UC Santa Cruz Technical Report UCSC-CRL-98-11, September 1998.

[12] Intel Corporation, http://developer.intel.com/drg/mmx/manuals/prm/prm.htm. *MMX Technology Developer's Programmer's Reference Manual*, 1997.

[13] Kevin Karplus. *A simplified view of two–phase clocking.* CE 222 Fall 1995 Class Handout, Fall 1995.

[14] Kevin Karplus and T.V. Verghese. Sizing cMOS gates along a critical path—a tutorial. Technical Report UCSC-CRL-87-30, UC Santa Cruz, January 1987.

[15] Thomas P. Kelliher, Eric S. Gayles, Robert M. Owens, and Mary Jane Irwin. The MGAP-2: An advanced, massively parallel VLSI signal processor. In *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, volume 5, pages 3219–22. IEEE, May 1995.

[16] Heung-Nam Kim, Manjut Borah, Robert Michael Owens, and Mary Jane Irwin. 2-D discrete cosine transforms on a fine grain array processor. In *Proc. VLSI Signal Processing VII*, pages 356–367. IEEE, October 1994.

[17] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler. Hidden Markov models in computational biology: Applications to protein model ing. *Journal of Molecular Biology*, 235:1501–1531, February 1994.

[18] Daniel P. Lopresti and Richard Hughey. The B-SYS programmable systolic array. Technical Report CS-89-32, Dept. Computer Science, Brown University, Providence, RI, June 1989.

[19] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[20] PLX Technology. *PCI 9050-1 Datasheet*. April 17, 1997.

[21] Doug Williams. *Prograss on the Design of the Kestrel SRAM*. UC Santa Cruz, June 1996.

Appendix A

Programming the PLX 9050's EEPROM

The PLX 9050 interfaces the Kestrel board to the host system through the PCI bus, as discussed in Section 3.2. The 9050 has an EEPROM that sets the initial values of the configuration registers. The EEPROM requires no external programmer, unlike the Xilinx FPGA PROM, since the 9050 can directly program the EEPROM through software. The following presents the step-by-step procedure for programming the 9050's EEPROM:

1. Insert a NM93CS46 chip in the 8-pin DIP socket U46 (adjacent the PLX 9050) on the kestrel board.

2. Insert the board into the machine "merlin" and turn on the power.

3. At the first boot prompt, select "Microsoft Windows", and at the second boot prompt, select "Clean Boot."

4. Next, change directories into `plx/monitor` using then `cd` command.

5. Invoke the PLX monitor program `plxmon`

6. Before the EEPROM can be programmed, bit 7 in PLX registers `0x10` and `0x14` must be cleared to zero using the `pcr` command. The bits from the two registers can be cleared with the commands:

```
pcr 10 ffafaf00
pcr 14 0000e801
```

   where 10 and 14 are the register numbers, and `ffafaf00` and `0000e801` are the new register values, respectively. The reigster contents can be viewed at any time using the commands:

```
        pcr 10
output> 0x010: 0xffafaf00
        pcr 14
output> 0x014: 0x0000e801
```

   where the program output the text after `output>`.

7. Now that the two bits have been cleared, the EEPROM can be programmed from the file `eeprom3.mon` using the read command: `read eeprom3.mon`, where Figure A.1 lists the current contents of the `eeprom3.mon` file. The file contains a series of `eep` commands that cause the PLX to write to the EEPOM.

```
eep 000 905010b5 echo PCIIDR; Device ID, Vendor ID
eep 004 06800001 echo PCICCR; Class Code
eep 008 00000000 echo Subsystem ID, Subsystem Vendor ID
eep 00c 00000100 echo Maximum Latency, Minimum Grant, Int Pin, Int Routing

eep 010 0fff0000 echo LAS0RR; Local Address Space 0 Range
eep 014 0fff0000 echo LAS1RR; Local Address Space 1 Range
eep 018 0fff0000 echo LAS2RR; Local Address Space 2 Range
eep 01c 0fff0000 echo LAS3RR; Local Address Space 3 Range
eep 020 00000000 echo EROMRR; Expansion ROM Range

eep 024 00000001 echo LAS0BA; Local Address Space 0 Base Address (Re-Map)
eep 028 04000001 echo LAS1BA; Local Address Space 1 Base Address (Re-Map)
eep 02c 08000001 echo LAS2BA; Local Address Space 2 Base Address (Re-Map)
eep 030 0c000001 echo LAS3BA; Local Address Space 3 Base Address (Re-Map)
eep 034 00000000 echo EROMBA; Expansion ROM Base Address (Re-Map)

eep 038 00800000 echo LAS0BRD; Local Address Space 0 Bus Region Descriptors
eep 03c 00800000 echo LAS1BRD; Local Address Space 1 Bus Region Descriptors
eep 040 00800000 echo LAS2BRD; Local Address Space 2 Bus Region Descriptors
eep 044 00800000 echo LAS3BRD; Local Address Space 3 Bus Region Descriptors
eep 048 00800000 echo EROMBRD; Expansion ROM Bus Region Descriptors

eep 04c 00000000 echo CS0BASE; Chip Select 0 Base
eep 050 00000000 echo CS1BASE; Chip Select 1 Base
eep 054 00000000 echo CS2BASE; Chip Select 2 Base
eep 058 00000000 echo CS3BASE; Chip Select 3 Base
eep 05c 00000000 echo INTCSR; Interrupt Control/Status
eep 060 087e4000 echo CNTRL; User I/O, EEPROM, Init Control
```

Figure A.1: Current PLX 9050 EEPOM configuration from the file `eeprom3.mon`.

Appendix B

Host High-Level Command Format

The host provides a high-level interface to the Kestrel board, as described in Section 3.8.2, that frees the runtime environment from direct interaction with the Kestrel hardware. The commands, listed in Table 3.21, allow the runtime environment to load a program, manipulate data, run the program, and store the results in the users output files. The following table details the exact data format for each command.

Action	Data Type	Description
Load Program Memory		
Send	ASCII 9-byte	KCMD_SPECIFY_PROGRAM | WRITE=1<<31
Send	ASCII 9-byte	0
Receive	ASCII 9-byte	Wait for SYNC_SIGNAL
Send	32-bit binary	Number of instructions
Send	32-bit binary	Address of first instruction
Send	32-bit binary	Number of end-of-program instruction
Send	ASCII hex 25-byte	24-byte ASCII hex instruction with a terminator, one for each instruction
Specify Input Data Stream Source		
Send	ASCII 9-byte	KCMD_SPECIFY_QIN | WRITE
Send	ASCII 9-byte	0
Receive	ASCII 9-byte	Wait for SYNC_SIGNAL
Send	32-bit binary	Data source type DATA_SOURCE_FILE=1 or DATA_SOURCE_TRANSMIT=2
Send	32-bit binary	Length in bytes

Action	Data Type	Description
Send	text/binary	File name string or data stream
Specify Output Data Destination		
Send	ASCII 9-byte	KCMD_SPECIFY_QOUT | WRITE
Send	ASCII 9-byte	0
Receive	ASCII 9-byte	Wait for `SYNC_SIGNAL`
Send	32-bit binary	Data destination type DATA_SOURCE_FILE=1 or DATA_SOURCE_TRANSMIT=2
Send	32-bit binary	Length of data
Send	text/binary	File name string if type DATA_SOURCE_FILE, omitted if type DATA_SOURCE_TRANSMIT
Retrieve Ouput Data		
Send	ASCII 9-byte	KCMD_RETRIEVE_QOUT | WRITE
Send	ASCII 9-byte	0
Receive	ASCII 9-byte	Wait for `SYNC_SIGNAL`
Receive	32-bit binary	Size of data
Receive	binary	Data
Program Output Queue Almost Full Threshold		
Send	ASCII 9-byte	KCMD_PROGY_QOUT | WRITE
Send	ASCII 9-byte	Threshold (0-4096)
Program Input Queue Almost Empty Threshold		
Send	ASCII 9-byte	KCMD_PROGY_QIN — WRITE
Send	ASCII 9-byte	Threshold (0-4096)
Run Program		
Send	ASCII 9-byte	KCMD_PROGY_QOUT | WRITE
Send	ASCII 9-byte	threshold (0-4096)
Receive	ASCII 9-byte	ERROR_SIGNAL, UPDATE_SIGNAL, or TERMINATE_SIGNAL
Receive	32-bit binary	Status register contents
Receive	32-bit binary	Current program counter
Receive	32-bit binary	Total input stream length
Receive	32-bit binary	Amoun t of input data used
Receive	32-bit binary	Number of output bytes
Receive	32-bit binary	Execution time in milliseconds, including I/O
Get Error Condition		
Send	ASCII 9-byte	KCMD_GET_ERROR_CODE | READ=0
Receive	ASCII 9-byte	Error condition
Terminate Program Session		
Send	ASCII 9-byte	KCMD_TERMINATE_PROGRAM | WRITE
Send	ASCII 9-byte	0

Appendix C

Verilog Simulation

This appendix presents how to use the Verilog model of the 64-PE Kestrel chip, written by Jeff Hirschberg, with Mentor Graphics Verilog. The Verilog simulates the 64-PE Kestrel chip and generates test patterns for use with both the IMS tester, as discussed in Appendix E, and with the layout simulator, as discussed in Appendix D. The Verilog uses object code generated by the old 64-bit assembler, and does not model the current array-controller design. Jeff Hirschberg created all the test vectors used to verify the Kestrel design.

The Verilog source is stored in the Kestrel project's CVS repository, accessible with the command:

```
cvs -d /projects/kestrel/cvs checkout verilog
```

creating a `verilog` directory with many subdirectories with self-explanatory names. The `Kestrel` directory contains the top-level cell, and the file `KestrelTestArray64.v` contains the main loop that runs the simulation. The variables `irsim` and `ims`, set on lines 90 and 93, respectively, control whether the simulation outputs ims tests or `irsim` test vectors to

simulate the layout.

To run the Verilog simulation using Mentor Graphics, add `/projects/cadtools/mentor` to the executable search path, and set the environment variables:

```
setenv MGC_HOME /projects/cadtools/mentor
setenv LM_LICENSE_FILE 1717@services
```

Next, execute the commands `qhlib work` and `qvlcom */*.v` to configure the working directory and compile all files. To run the simulation, give the command

```
qhsim -c KestrelTestArray64,
```

where the program automatically uses the object code in the file `program` and the input in `stageIn`, formatted as ASCII hexadecimal with one byte per line. The simulation produces IMS test vectors in the file `ims` if `ims` is set to one, and `irsim` test vectors in `KestrelTest.iraim` if the `irsim` varialble is set to one.

Object files can be created with the 64-bit assembler `kasm64` from the `/projects/kestrel/bin/` directory. The directory `/projects/kestrel/arch/assembler/testing` contains the test programs used to verify the layout design and fabricated chips on the tester. These programs used a special version of the assembler, located in that same directory called `binkasm`, that allows direct specification of instruction operand fields. Programs in `/projects/kestrel/arch/testing` algorithmically generated the tests for the ALU, comparator, and bit shifter.

Appendix D

Simulating the 64-PE Kestrel Chip

This appendix presents a description of how to simulator the 64-PE Kestrel chip, including extraction of the netlist from the layout design and the conversion of the Verilog output files into `irsim` command files. Due to the design's large size, careful attention was required when extracting the netlist as any small inefficiency could greatly increase time required for the the edit-compile-simulate process. The Verilog model does not directly create the `irsim` command script so the details of the simulation did not have to be programmed into the Verilog model.

The first step to simulate the kestrel design is converting the layout to a netlist. The directory `/projects/kestrel/design/` contains the physical design and other data files. The design files for the 64-PE Kestrel chips are spread across several subdirectories listed in Table D.1. The top-level cell `kestrel_chip` is located in the `Kestrel64` directory, and the directory `/projects/kestrel/design/Kestrel64/SIMULATE` contains all files needed to extract and simulate the netlist.

Table D.2 list the files used when compiling and simulating the 64-PE netlist. Many

Directory	Description
`sram/magic`	256 byte local SRAM
`multiplier/magic`	8-bit booth multiplier
`alu/magic`	PE core
`registers/magic`	Register bank
`global/magic`	Global logic
`PE/magic`	PE interconnect
`Kestrel64/magic`	Global interconnect, top level

Table D.1: Contents of directories in `/projects/kestrel/design/` containing Kestrel chip layout files.

of the files are located in other directories, and the `make_link` script automatically creates symbolic links. The script also creates symbolic links all Magic files and the directories listed in Table D.1, so the extraction process does not span multiple directories. After creating symbolic links to layout files, the netlist can be generated with the command `make kestrel_chip.sim` that runs Magic and ext2sim to create the `kestrel_chip.sim` and `kestrel_chip.al` files. The process takes approximately one-half hour, and requires a machine with at least 200 megabytes of memory to avoid heavy use of swap space. All files are create on the local `/tmp` directory to avoid the performance penalty of network filesystem access. Next, the script runs a program that removes unnecessary aliases from the `kestrel_chip.al`, and invokes the `irsim` simulator. Once the simulator loads the `kestrel_chip.sim` and `kestrel_chip.al` files, the user should create binary version of the netlist using the `wnet` command. The new file `kestrel_chip.inet` is half the size of the original and loads several times faster.

The system uses a modified version of Magic that supports the command `:extract rename`, added by Richard Hughey. This command causes Magic to replace cell names with numbers to reduce the lengths of nodes names. While executing the `:extract all`

Name	Type	Description
`make_link`	File	Script to create links to .mag files
`make_test`	File	Script to create irsim command file
`6pe.irsimcmd.m4`	File	M4 header file used to convert the simulation macros generated by the verilog model into `irsim` scripts.
`irconv`	Link	Perl script to convert nodes names for irsim scripts.
`Makefile`	Link	Makefile for simulating the 64-PE chip.
`scmos-sub.tech26`	Link	Technology file for the 0.5 micron process.
`tests`	Link	Directory containing `irsim` test files.

Table D.2: Contents of directory `/projects/kestrel/design/Kestrel64/SIMULATE` containing files and scripts to simulate the Kestrel 64-PE chip design.

command, Magic produces the `cell_to_number.pm` file that describes the mapping between cell names and numbers, formated as a Perl package for inclusion by the `irconv` program. Since the user cannot know how Magic will rename cells, the simulator command script must refer to nodes with translated names. To convert the node names, the `irconv` script substitutes the numbers assigned by Magic to cell names into the simulator command script. Within the command script, each node name appears within a `NODE()` statement so it can be identified by the conversion script.

The `tests` directory, shown in Table D.2, contains compressed files with test vectors for simulating the 64-PE Kestrel chip. The files are formatted as one M4 macro for each kestrel instruction. Each macro supplies instruction bits, data, and verification values for every important signal in every PE. The `make_test` script converts the M4 macros into a sequence of `irsim` commands by concatenating a test vector file with the `64pe.irsimcmd.m4` file, piping the results through the `M4` program, and piping the `M4` output through the `irconv` program to change the node names to match those in the layout. The resulting file, always named `test.irsimcmd`, can then be used by `irsim` to simulate the design. The `make`

`simulate` command will start the simulator with

the appropriate options, running the `test.irsimcmd` test script.

Appendix E

Testing the 64-PE Kestrel Chip

This appendix describes how to test the Kestrel chips using the IMS tester with test files generated from the Verilog model discussed in Appendix C. The directory `/projects/-kestrel/design/run_ims` contains a program to automatically run a series of tests, verifying functionality and finding the maximum clock speed that each chip will operate correctly with each test.

The program `runtest` in `/projects/kestrel/design/run_test` provides a fully automated mechanism for testing a Kestrel chip. To use this program, the IMS tester environment must be configured with the commands:

```
setenv IMS_ROOT /projects/daizu/ims
setenv IMS_LM jelly.cse.ucsc.edu
setenv IMS_SERVER jelly
if (! $?LD_LIBRARY_PATH) then
  setenv LD_LIBRARY_PATH $IMS_ROOT/lib
else
  setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH\:$IMS_ROOT/lib
endif
```

along with adding `$IMS_ROOT/bin` to the executable search path. The directory `tests` contains IMS tests for various components in the Kestrel PE. The file `AllImsTests` lists all

the tests to run, and indicates dependencies between tests; for example, the wired-OR must function correctly for the sram test to work, as the sram test uses the wired-OR to signal if an error ocurred. The file `Kestrel_Settings.ims` contains the IMS configuration file, defining timing the signalling information for the tester. To fully test a Kestrel chip, run on the tester, insert a 64-PE Kestrel chip in the test fixture, and run the `runtest chipXX.results` program, where the file `chipXX.results` will contain the final clock settings and current measurements or a list of errors that occurred.

Interpreting the test results is complicated by the design of the test fixture and the output-enable bug in the Kestrel chip. The IMS tester has fewer than 84 channels, so they must be shared. It takes two test vectors to load an instruction, one to load latches on the text fixture, and another to actually execute the instruction. Table E.1 lists the instruction bits, with those on the same line sharing the same tester channels. Also, the left and right data and mask signals use the same channels, so if an instruction reads and writes from the ends of the array, a third vector is required. The output-enable bug affects testing results when data is written into array, as the other end cannot be turning off, causing a drive fight.

Unlatched	Latched
nop	dest[5]
force	dest[4]
func[4:0]	(imm[7], imm[3], imm[5], imm[4], imm[6])
ci	bit[2]
mp	
lc	bit[3]
finv	dest[3]
fb[4:0]	(dest[2], dest[1], dest[0], rd, wd)
opB[2:0]	(imm[2], imm[1], imm[0])
opA[5:0]	
opC[5:0]	
rm[1:0]	bit[1:0]

Table E.1: Instruction bits mapped to the same channel as the tester has fewer than channels than the chip pins. The left and right data and mask pins also map to the same channel.