12-1-1985

Architectural Approaches For Gallium Arsenide Exploitation In High-Speed Computer Design

David Allen Fura
Purdue University

Architectural Approaches For Gallium Arsenide Exploitation In High-Speed Computer Design

D. Fura

TR-EE 85-17
December 1985

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

This research was supported by RCA Advanced Technology Laboratories, Morrestown, New Jersey, in years 1984 and 1985.

ARCHITECTURAL APPROACHES FOR GALLIUM ARSENIDE EXPLOITATION IN HIGH-SPEED COMPUTER DESIGN

A Thesis

Submitted to the Faculty

of

Purdue University

by

David Allen Fura

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical Engineering

December 1985

dedicated
to my mother

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my major professor, Professor Veljko Milutinović, for his guidance and encouragement throughout the course of this research.

I am equally indebted to RCA Corporation for their generous support of this research, and especially wish to thank Walt Helbig, Bill Heagerty, and Wayne Moyers for their technical contributions to this research.

I also offer my sincere thanks to Professor Jose Fortes and Professor James Cooper for serving on my committee and for their thoughtful review of this thesis.

I am likewise greatly appreciative of the outstanding support provided by Tim Gilbert, Sharon Katz, and especially Mickey Krebs in the preparation of this thesis.

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES x

ABSTRACT xv

CHAPTER I - INTRODUCTION 1

CHAPTER II - GaAs TECHNOLOGY 3

2.1 GaAs Device Families 3
2.2 GaAs Logic Families 4
2.3 GaAs-Silicon Comparison 9

CHAPTER III - DESIGN CONSIDERATIONS FOR GaAs COMPUTER SYSTEMS 16

3.1 The Effect of GaAs Characteristics on Computer Design Strategy 16
3.2 Hardware Design Issues 18
3.2.1 Processor Configuration 19
3.2.2 Single-Chip GaAs Processor Designs 19
3.2.3 Instruction Pipeline Design 21
3.2.4 Register File Design 21
3.2.4.1 Register Cell Design 25
3.2.4.2 Register File Partitioning 28
3.2.5 Execution Unit Design 31
3.2.5.1 Adder Design 32
3.2.5.2 Multiplier/Divider Design 33
3.2.6 Instruction Format Design 35
3.2.6.1 Frequency-based Instructions 36
3.2.6.2 Context-based Instructions 38
3.2.7 Memory System Design 39

Page

3.2.7.1 The Role of Virtual Memory 40
3.2.7.2 Memory Hierarchy 40
3.2.7.3 Run-time Control of Hierarchical Memory Systems 41
3.2.7.4 Compile-time Control of Hierarchical Memory Systems. 44
3.2.7.5 Pipelined Memory Systems 44
3.3 Compiler Design Issues 45
3.3.1 Compiler Optimizations in Control 48
3.3.1.1 Sequencing Hazard Interlocks 48
3.3.1.2 Timing Hazard Interlocks 52
3.3.2 Compiler Optimizations in Memory 53
3.3.2.1 Register File Compiler Optimizations 56
3.3.2.1.1 Reusability 56
3.3.2.1.2 Prefetching 58
3.3.2.2 Cache / Main Memory Compiler Optimizations 58
3.3.2.2.1 Reusability 61
3.3.2.2.2 Prefetching 62

CHAPTER IV - PIPELINE AND INSTRUCTION FORMAT EXPERIMENTS 63

4.1 Evaluation Methodology 63
4.1.1 Workload Model 64
4.1.2 Architecture Model 65
4.1.3 Workload to Architecture Translation 66
4.2 Pipeline Experiment 66
4.2.1 Rationale 67
4.2.2 Candidate Pipeline Descriptions 67
4.2.3 Evaluation Criterion 68
4.2.4 Causes of Non-ideal Performance 68
4.2.5 Modeling Memory and Compiler Effects 70
4.2.5.1 Memory Parameters 70
4.2.5.2 Compiler Parameters 72
4.2.6 Modeling the Workload Effects 72
4.2.6.1 Workload Parameter Definitions 72
4.2.6.2 Workload Parameter Values 74
4.2.7 Analytical Pipeline Performance Model 74
4.2.7.1 Normal Silicon (3,3) 76
4.2.7.2 Normal Silicon (3,6) 76
4.2.7.3 Normal Silicon (6,3) 76
4.2.7.4 Packed (3,3) 76
4.2.7.5 Packed (3,6) 82
4.2.7.6 Packed (6,3) 82

Page

4.2.7.7 Pipelined Memory (3,3) 82
4.2.7.8 Pipelined Memory (3,6) 82
4.2.7.9 Pipelined Memory (6,3) 82
4.2.8 Experimental Results 83
4.2.9 Discussion 96
4.2.9.1 Candidate Pipeline Comparison 96
4.2.9.2 Memory Configuration Comparison 96
4.2.9.3 Compiler and Memory Parameter Comparison 97
4.2.9.4 Workload Comparison 97
4.2.9.5 Summary 98
4.3 Instruction Format Experiment 99
4.3.1 Rationale 99
4.3.2 Candidate Instruction Format Descriptions 99
4.3.3 Evaluation Criteria 100
4.3.4 Evaluation Theory and Implementation 102
4.3.5 Static Instruction Count Subexperiment 103
4.3.5.1 Procedure 103
4.3.5.2 Results 103
4.3.6 Dynamic Instruction Count Subexperiment 105
4.3.6.1 Procedure 105
4.3.6.2 Results 105
4.3.7 Execution Time Subexperiment 105
4.3.7.1 Workload Model - Cache Model Discussion 108
4.3.7.2 Cache Simulator Description 109
4.3.7.3 MIPS Simulator Modifications 109
4.3.7.4 Procedure and Results 110
4.3.7.4.1 Procedure 110
4.3.7.4.2 Results 110
4.3.7.5 Discussion 111
4.3.7.5.1 The Effect of Instruction Format on Instruction Counts 111
4.3.7.5.2 The Effect of Cache Size on Execution Time 115
4.3.7.5.3 The Effect of Fewer Register Fields 115
4.3.7.5.4 The Effect of Smaller Immediate Field Lengths 116
4.3.7.5.5 The Effect of Variable Immediate Field Sizes 116
4.3.7.5.6 The Use of Compact Formats for Instruction Packing 117
4.3.7.5.7 Summary 118

CHAPTER V - SUMMARY AND RECOMMENDATIONS 120

Page

5.1 Summary 120
5.2 Recommendations 120

LIST OF REFERENCES 121

APPENDICES

Appendix A: Analytical Pipeline Performance Model Derivation 128
Appendix B: Determination of Candidate Instruction Format Costs 138

LIST OF TABLES

Table Page

2.1 Performance Characteristics of GaAs Designs....12

2.2 Performance Comparison of GaAs and Silicon [BasNu84]. 13

4.1 Workload Characteristics Relevant for Pipeline Study....75

4.2 Instruction Fields for Candidate Instruction Formats....101

4.3 Breakdown of Dynamic Costs for the Candidate Formats....107

Appendix
Table

B.1 Costs for Candidate Format 28(3). 142

B.2 Costs for Candidate Format 28(3210)....143

B.3 Costs for Candidate Format 24(32). 144

B.4 Costs for Candidate Format 24(3210)....145

B.5 Costs for Candidate Format 24(2). 146

B.6 Costs for Candidate Format 24(210). 147

B.7 Costs for Candidate Format 20(32). 148

B.8 Costs for Candidate Format 20(3210)....149

Table Page

B.9 Costs for Candidate Format 16(21). 150

B.10 Costs for Candidate Format 16(210).....151

LIST OF FIGURES

Figure Page

2.1 BFL D-MESFET Inverter [EdWeZ79].....6

2.2 SDFL D-MESFET Inverter [EdWeZ79].....7

2.3 DCFL Inverter [EdWeZ79].....8

2.4 ECL/CML HBT Logic Gate [AsMiA84].....10

2.5 STL-like HBT Inverter [YuMcS84].....11

3.1 Example Silicon Instruction Pipeline [FurMi85].....22

3.2 Example Silicon Pipeline Implemented in GaAs [FurMi85].....22

3.3 Example GaAs Instruction Pipeline with a Pipelined Memory [FurMi85].....23

3.4 Example GaAs Instruction Pipeline with Instruction Packing [FurMi85].....24

3.5 Example GaAs Instruction Pipeline with Long-latency Datapath Elements [FurMi85].....24

3.6 Microcode Pipeline of the HP-FOCUS Processor [BeDoF81].....26

3.7 Register Cell Design of the HP-FOCUS Processor [BeDoF81].....27

3.8 Register Cell Design of the Berkeley RISC-II Processor [Sherb84].....29

3.9 Register Cell Design Employing a Single Read Bus [MiFuH86].....30

Figure Page

3.10 Example Pipelined Memory System. 46

3.11 Example Program Sequence on a Silicon Processor Before Branch Fill..50

3.12 Example Program Sequence on a Silicon Processor After Branch Fill. ..50

3.13 Example Program Sequence on a GaAs Processor Before Branch Fill. ..51

3.14 Example Program Sequence on a GaAs Processor After Branch Fill. 51

3.15 Example Program Sequence Showing a Destination-Source Conflict.....54

3.16 Example Program Sequence Showing Default Compiler Action for Destination-Source Conflict.....54

3.17 Example Program Sequence Showing a Successful Reorganization of Destination-Source Conflict. 55

3.18 Example Program Sequence Showing Poor Register Allocation. 57

3.19 Example Program Sequence Showing Good Register Allocation. 57

3.20 Example Program Sequence on a Silicon Processor Before Load Fillin..59

3.21 Example Program Sequence on a Silicon Processor After Load Fillin....59

3.22 Example Program Sequence on a GaAs Processor Before Load Fillin....60

3.23 Example Program Sequence on a GaAs Processor After Load Fillin. 60

4.1 Normal Silicon (3,3) Pipeline.....77

4.2 Normal Silicon (3,6) Pipeline.....77

4.3 Normal Silicon (6,3) Pipeline.....78

4.4 Packed (3,3) Pipeline.....78

4.5 Packed (3,6) Pipeline.....79

4.6 Packed (6,3) Pipeline.....79

Figure Page

4.7 Pipelined Memory (3,3) Pipeline....80

4.8 Pipelined Memory (3,6) Pipeline....80

4.9 Pipelined Memory (6,3) Pipeline....81

4.10 Pipeline Performance vs. "pih" in (3,3) Configuration for All Benchmarks....84

4.11 Pipeline Performance vs. "pih" in (3,6) Configuration for All Benchmarks....84

4.12 Pipeline Performance vs. "pih" in (6,3) Configuration for All Benchmarks....85

4.13 Pipeline Performance vs. "pdh" in (3,3) Configuration for All Benchmarks....85

4.14 Pipeline Performance vs. "pdh" in (3,6) Configuration for All Benchmarks....86

4.15 Pipeline Performance vs. "pdh" in (6,3) Configuration for All Benchmarks....86

4.16 Pipeline Performance vs. "pbf" in (3,3) Configuration for All Benchmarks....87

4.17 Pipeline Performance vs. "pbf" in (3,6) Configuration for All Benchmarks....87

4.18 Pipeline Performance vs. "pbf" in (6,3) Configuration for All Benchmarks....88

4.19 Pipeline Performance vs. "plf" in (3,3) Configuration for All Benchmarks....88

4.20 Pipeline Performance vs. "plf" in (3,6) Configuration for All Benchmarks....89

4.21 Pipeline Performance vs. "plf" in (6,3) Configuration for All Benchmarks....89

Figure Page

4.22 Pipeline Performance vs. "pbf" in (3,3) Configuration for Arithmetic-heavy Benchmark. 90

4.23 Pipeline Performance vs. "pbf" in (3,6) Configuration for Arithmetic-heavy Benchmark. 90

4.24 Pipeline Performance vs. "pbf" in (6,3) Configuration for Arithmetic-heavy Benchmark. 91

4.25 Pipeline Performance vs. "plf" in (3,3) Configuration for Arithmetic-heavy Benchmark. 91

4.26 Pipeline Performance vs. "plf" in (3,6) Configuration for Arithmetic-heavy Benchmark. 92

4.27 Pipeline Performance vs. "plf" in (6,3) Configuration for Arithmetic-heavy Benchmark. 92

4.28 Pipeline Performance vs. "pbf" in (3,3) Configuration for Branch-heavy Benchmark. 93

4.29 Pipeline Performance vs. "pbf" in (3,6) Configuration for Branch-heavy Benchmark. 93

4.30 Pipeline Performance vs. "pbf" in (6,3) Configuration for Branch-heavy Benchmark. 94

4.31 Pipeline Performance vs. "plf" in (3,3) Configuration for Load-heavy Benchmark. 94

4.32 Pipeline Performance vs. "plf" in (3,6) Configuration for Load-heavy Benchmark. 95

4.33 Pipeline Performance vs. "plf" in (6,3) Configuration for Load-heavy Benchmark. 95

4.34 Instruction Format Static Instruction Counts. 104

4.35 Instruction Format Dynamic Instruction Counts. 106

4.36 Instruction Format 28(3) Execution Time vs. Cache Size. 112

Figure Page

4.37 Execution Time for each Instruction Format with 64-byte Cache. 113

4.38 Execution Time for each Instruction Format with 512-byte Cache.....114

ABSTRACT

Fura, David A., M.S.E.E., Purdue University. December 1985. Architectural Approaches for Gallium Arsenide Exploitation in High-Speed Computer Design. Major Professor: Veljko M. Milutinovic.

Continued advances in the capability of Gallium Arsenide (GaAs) technology have finally drawn serious interest from computer system designers. The recent demonstration of very large scale integration (VLSI) laboratory designs incorporating very fast GaAs logic gates herald a significant role for GaAs technology in high-speed computer design. In this thesis we investigate design approaches to best exploit this promising technology in high-performance computer systems.

We find significant differences between GaAs and Silicon technologies which are of relevance for computer design. The advantage that GaAs enjoys over Silicon in faster transistor switching speed is countered by a lower transistor count capability for GaAs integrated circuits. In addition, inter-chip signal propagation speeds in GaAs systems do not experience the same speedup exhibited by GaAs transistors; thus, GaAs designs are penalized more severely by inter-chip communication.

The relatively low density of GaAs chips and the high cost of communication between them are significant obstacles to the full exploitation of the fast transistors of GaAs technology. A fast GaAs processor may be excessively underutilized unless special consideration is given to its information (instructions and data) requirements. Desirable GaAs system design approaches

encourage low hardware resource requirements, and either minimize the processor's need for off-chip information, maximize the rate of off-chip information transfer, or overlap off-chip information transfer with useful computation. We show the impact that these considerations have on the design of the instruction format, arithmetic unit, memory system, and compiler for a GaAs computer system.

Through a simulation study utilizing a set of widely-used benchmark programs, we investigate several candidate instruction pipelines and candidate instruction formats in a GaAs environment. We demonstrate the clear performance advantage of an instruction pipeline based upon a pipelined memory system over a typical Silicon-like pipeline. We also show the performance advantage of packed instruction formats over typical Silicon instruction formats, and present a packed format which performs better than the experimental packed Stanford MIPS format.

CHAPTER I
INTRODUCTION

Digital integrated circuits employing Gallium Arsenide (GaAs) technology have been regularly presented since the mid 1970s. The faster switching speed of GaAs and its higher resistance to adverse environmental conditions have created sporadic bursts of enthusiasm throughout the last ten years. However, not until recent significant advances in GaAs material quality and fabrication technology has GaAs begun to attract serious interest from computer system designers.

A coherent computer system design strategy requires a thorough understanding of the underlying implementation technology. The advances made in Silicon technology have drastically improved the capability of Silicon-based computer systems. Apart from the performance improvements due strictly to improved technology, technology-driven architectural advances have also played an important role. The effect that the characteristics of Silicon very large scale integration (VLSI) have had on computer design strategies is observable in the recent enthusiasm for designs such as dataflow computers [Denni80], systolic arrays [Kung82], and reduced instruction set computers (RISCs) [Patte85].

As GaAs technology is only expected to begin achieving integration levels approaching 10,000 gates by the late 1980s [Leopo85], it is not surprising that design strategies appropriate for GaAs should differ from those encountered in VLSI Silicon. In fact, Silicon-GaAs differences are more pronounced than as indicated by level of integration alone. Two additional key differences are the already mentioned higher speed of GaAs gates, as well as a corresponding higher penalty for inter-chip communication for GaAs chips. Clearly, it is vital to understand the characteristics of GaAs which influence computer system design before attempting to build GaAs-based computer systems, and it should not be assumed that Silicon-based techniques are desirable for GaAs implementations.

The purpose of this thesis is to explore the use of GaAs technology in computer system design. We are interested in *architectural* approaches for fully

exploiting the fast transistors of low-density GaAs chips by minimizing the deleterious effects of a slow off-chip environment. System *packaging* approaches for improving this slow inter-chip communication are also important issues but are not within the scope of this work.

We adopt a three step approach to GaAs computer system design. We first examine GaAs technology and determine its characteristics which are relevant for computer design. We then study the suitability of popular Silicon designs for satisfying the requirements of GaAs, and explore the use of little-used or novel approaches as well. Finally, we determine through experimentation whether the approaches which seem appropriate for GaAs implementation are indeed the most desirable.

Because the implementation technology plays such a critical role in the design of computer systems, Chapter II provides a description of GaAs technology relevant for a clear understanding of the architectural design tradeoffs required by GaAs. Chapter III discusses the architectural design issues which are affected by the characteristics of GaAs technology, and suggests possible design approaches. Chapter IV presents the discussion and results of two experiments which were undertaken in order to examine some approaches presented in Chapter III. We finish in Chapter V by summarizing our results and recommending a direction for future work in this area.

CHAPTER II
GaAs TECHNOLOGY

To provide a sound basis for subsequent GaAs computer system design discussions, this chapter presents an overview of digital GaAs technology. We discuss the relative merits of candidate device families and their logic gate implementations, and we also present some of the advanced GaAs digital designs which have appeared in the last few years. In order to permit a rational GaAs-Silicon comparison, we select the GaAs technology which appears to have the first shot at VLSI levels of integration in production quantities. In comparing this GaAs technology with Silicon NMOS we illuminate several differing characteristics. These GaAs-Silicon differences will then provide much of the motivation for the discussion of the next chapter.

2.1 GaAs Device Families

Just as Silicon technology has undergone major change, GaAs technology has seen rapid advancements in its relatively short history. The first published digital GaAs circuits were introduced in the mid 1970s. The earliest devices to be utilized in these circuits were depletion-mode metal-semiconductor field effect transistors (D-MESFETs). Some widely used devices which followed include enhancement-mode MESFETs (E-MESFETs), modulation-doped FETs (MODFETs), and heterojunction bipolar transistors (HBTs).

D-MESFETs were the first devices to be used in digital circuit designs, and the ease with which they are fabricated is one of their primary advantages over other device technologies. Some additional strengths include high insensitivity to fanout and large noise margins for D-MESFET logic gates [YaHiA83]. Unfortunately, D-MESFET logic designs must utilize complex circuits resulting in large power and area requirements [EdWeL83]. They require two power supplies and voltage level shifting logic to allow logic gates to be cascaded.

E-MESFET circuits require but a single power supply and no voltage level shifting logic, thus requiring less power and area than D-MESFET designs

[EdWeL83]. For this reason E-MESFETs are considered more appropriate for VLSI implementations. E-MESFET logic circuits can also be faster than D-MESFET circuits; however, they are more sensitive to fanout loading and perform poorly in high load environments [EdWeL83]. E-MESFETs also require higher material quality and more complex processing to achieve the high threshold voltage uniformity necessary for working devices [MaOhH84] [EdWeL83].

MODFETs, also commonly known as high electron mobility transistors (HEMTs), achieve much faster switching speeds than either D-MESFETs or E-MESFETs; consequently, they have generated much interest for high-speed computer design. MODFETs utilize a layer of AlGaAs material to supply electrons into an undoped GaAs channel. Because the room temperature mobility of electrons in undoped GaAs is almost twice as high as in n-channel GaAs MESFETs, MODFETs are able to more quickly change their output state with low power consumption [SolMo84]. At liquid nitrogen temperature (77 $^{\circ}$ K), electron mobilities in MODFETs are improved even further - approximately six times higher than at room temperature [SolMo84]. Ionized impurity scattering in n-channel GaAs MESFETs deny this higher mobility to MESFET devices [SolMo84]. The major disadvantages of MODFETs are a more complex processing requirement than MESFETs and the need for a very high-quality AlGaAs layer [SolMo84].

HBTs do not suffer from the threshold voltage problems that plague the FETs which we just described, and this is an important advantage for VLSI implementations [Eden82]. In addition to their built-in threshold voltage control, HBTs are very fast and have higher output drive capability than FETs, resulting in lower sensitivity to fanout and loading [AsMiA83]. HBTs may also be employed in circuit designs with differential inputs and outputs which result in decreased switching noise [AsMiA83]. The disadvantages of HBTs are higher processing complexity than FETs, as well as relatively high power and chip area requirements [AsMiA83].

2.2 GaAs Logic Families

Several logic families exist which utilize the above devices. Some widely-used families which utilize FETs include buffered FET logic (BFL), Schottky diode FET logic (SDFL), and direct coupled FET logic (DCFL). Other logic families utilizing bipolar transistors include emitter-coupled logic (ECL) and Schottky transistor logic (STL).

Early BFL logic circuits utilizing D-MESFETs exhibited fast switching speeds at high power levels. One reported design had gate delays of 34 ps at 41.0 mW per gate [NuPeB82]. Efforts to reduce the power consumption of BFL gates resulted in low power BFL (LPBFL) designs with a relatively small penalty in switching speed. Advanced LPBFL designs include a 32-bit adder containing 420 gates with gate delays of 230 ps and a power of 2.8 mW per gate [YaHiA83]. To our knowledge, the highest level of integration achieved with a BFL design is an LPBFL 12x12-bit multiplier which, including its non-BFL input and output buffers, incorporated 1083 gates [FuTaI84]. The LPBFL gates had switching speeds of 170 ps and a power dissipation of 1.7 mW. Figure 2.1 is an example of a BFL inverter which demonstrates the complexity inherent in D-MESFET-based circuits. However, LPBFL versions require only one or two diodes for voltage level shifting instead of the three shown.

SDFL logic circuits also use D-MESFETs and have a relatively complex logic circuit implementation as shown in Figure 2.2. However, because of their lower power and area requirements, they have achieved higher integration levels than BFL designs [EdWeZ79]. In fact, the first reported GaAs LSI (> 1000 transistors) design utilized SDFL gates [LeKaW82]. This 1008-gate 8x8-bit multiplier had gate delays of 150 ps and a power dissipation of 0.6-2.0 mW per gate. The highest reported level of integration achieved with SDFL logic gates appears to be a combination gate array/SRAM chip [VuRoN84]. This design incorporated 432 programmable cells, 32 interface I/O buffer cells, and four 4x4-bit SRAMs for a total of approximately 8000 devices. The average gate propagation delay and power dissipation were 150-300 ps and 1.5 mW, respectively.

DCFL logic circuits utilizing E-MESFET drivers and D-MESFET loads (DCFL E/D-MESFETs) have achieved by far the highest level of integration of any GaAs technology. The use of the simple circuit configuration of Figure 2.3 gives DCFL designs a decided advantage in power dissipation and area requirements, both extremely important for VLSI implementations. Several significant DCFL E/D-MESFET designs have been reported. A 2000-gate gate array exhibiting gate delays of 215 ps and power requirements of 0.5 mW per gate was reported [ToUcK85]. A 3168-gate 16x16-bit multiplier with gate delays of 150 ps and a power dissipation of 0.3 mW per gate has been presented [NaSuS83]. However, the highest reported level of integration achieved to date is a 16K-bit SRAM containing 102,300 devices [IsInI84]. The access time was 4.1 ns and the total chip power consumption was 2.5 W. Ring oscillator measurements showed gate delays of 115 ps at 0.1 mW per gate.

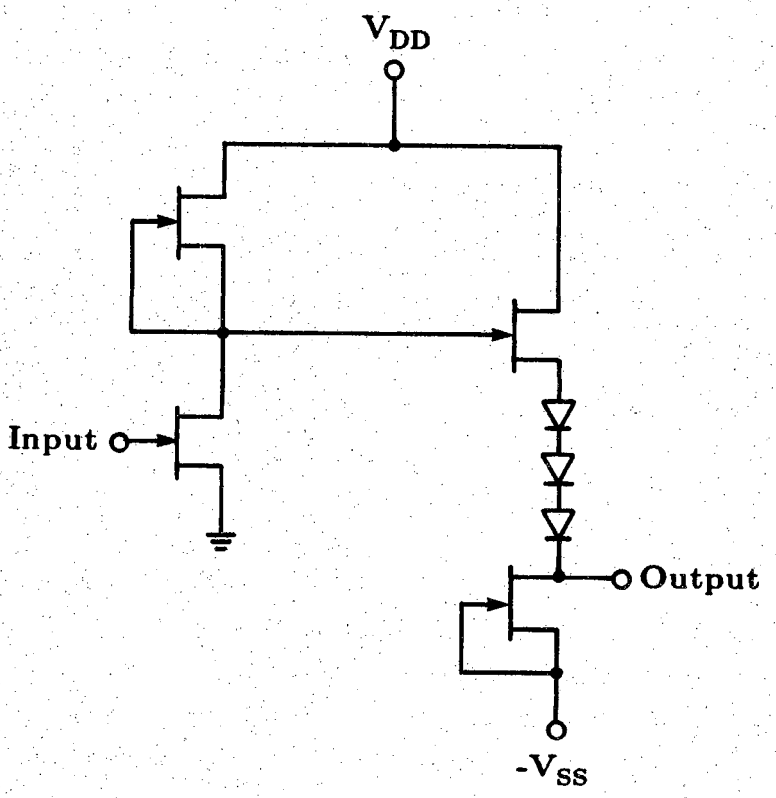


Figure 2.1 BFL D-MESFET Inverter [EdWeZ79].

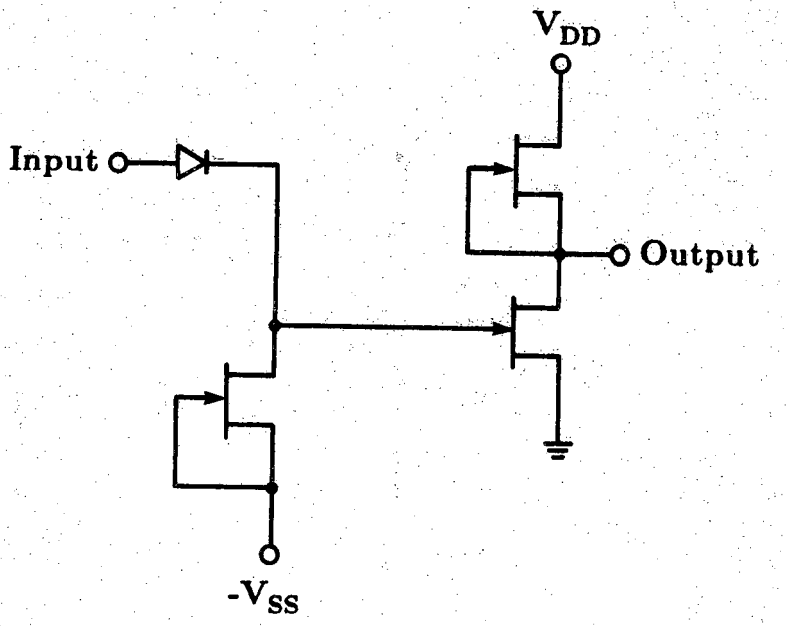


Figure 2.2 SDFL D-MESFET Inverter [EdWeZ79].

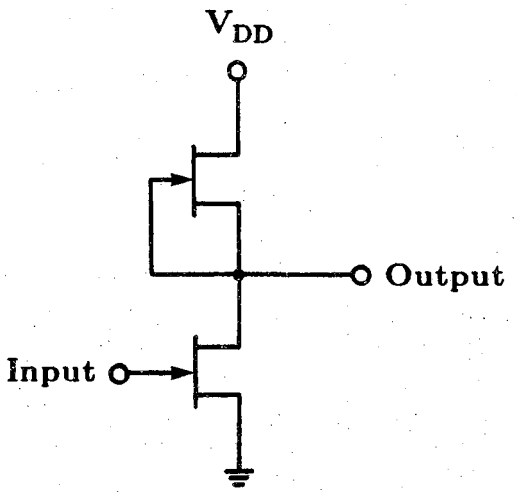


Figure 2.3 DCFL Inverter [EdWeZ79].

The DCFL logic circuit design of Figure 2.3 is also utilized for MODFET devices. Because of the early state of MODFET development, this promising technology has not achieved the levels of integration experienced by E/D-MESFETs. The highest level of integration achieved thus far is a 4K-bit SRAM [KuMiS84]. This chip had an access time of 4.4 ns and power dissipation of 860 mW at room temperature. At 77 °K, access times and power requirements of 2.0 ns and 1.6 W, respectively, were achieved. However, MODFET designs have the distinction of holding the fastest gate propagation times. The current record at room temperature is 11.6 ps, while at 77 °K it is only 8.5 ps [Rose85].

HBTs have only recently been used in digital logic circuits and have achieved the lowest level of integration of the devices we've discussed. Ring oscillator measurements were performed using common mode logic (CML) ECL gates [AsMiA84]. Propagation delays of 60 ps were achieved with a gate power dissipation of 3.0 mW. An example ECL/CML logic gate is shown in Figure 2.4. Thus far, the highest reported level of integration for HBTs is a 1K-gate gate array utilizing an STL-like logic implementation [YuMcS84]. Using a circuit design represented by the inverter of Figure 2.5, a propagation delay and power dissipation of 400 ps and 1.0 mW, respectively, were achieved.

Table 2.1 summarizes the current relative performance levels of these five logic circuit families. The DCFL E/D-MESFET family shows the highest capability in these published designs. Because DCFL E/D-MESFETs were first to achieve VLSI (>10,000 transistors) densities in laboratory environments, they were among the first to be seriously considered for processor implementation. In fact, the description of an 8-bit GaAs processor using E/D-MESFET technology has already been published [HeScZ85].

2.3 GaAs-Silicon Comparison

Because of their early attainment of VLSI densities in laboratory environments, DCFL E/D-MESFETs will be used to represent GaAs technology throughout the rest of this thesis. The performance characteristics of Silicon are based primarily on NMOS, which has the same logic gate circuit design as DCFL designs.

Table 2.2 shows performance characteristics of both DCFL E/D-MESFET GaAs and NMOS Silicon [BasNu84] which are relevant for computer system design. From this table, three fundamental differences between GaAs and Silicon are evident.

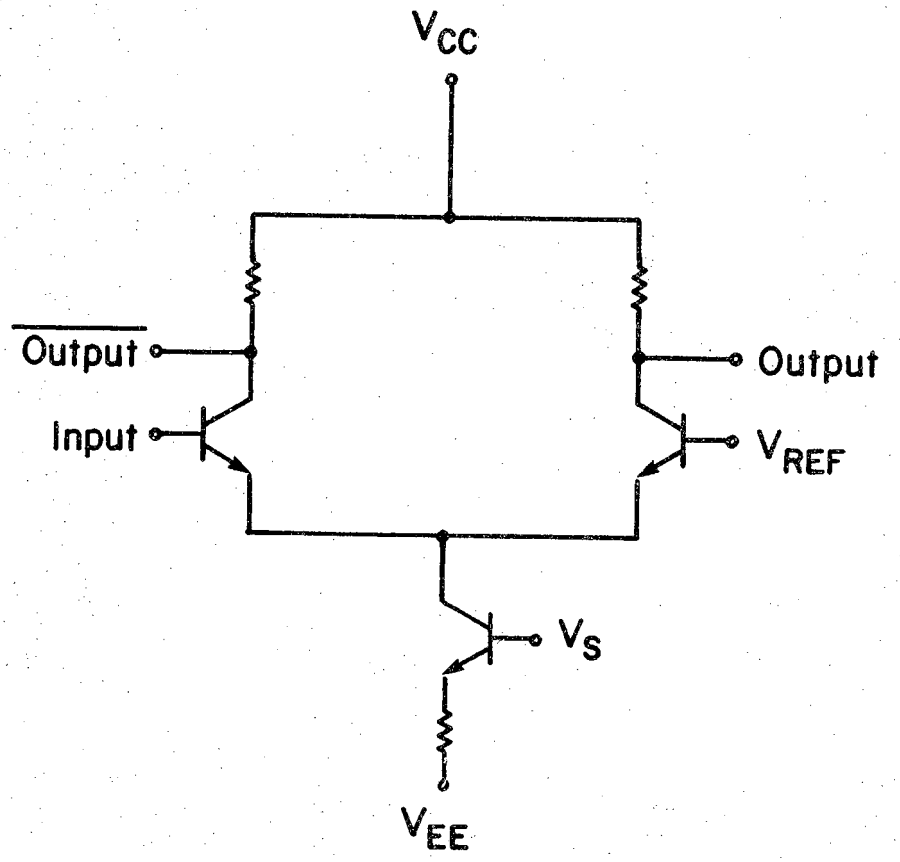


Figure 2.4 ECL/CML HBT Logic Gate [AsMiA84].

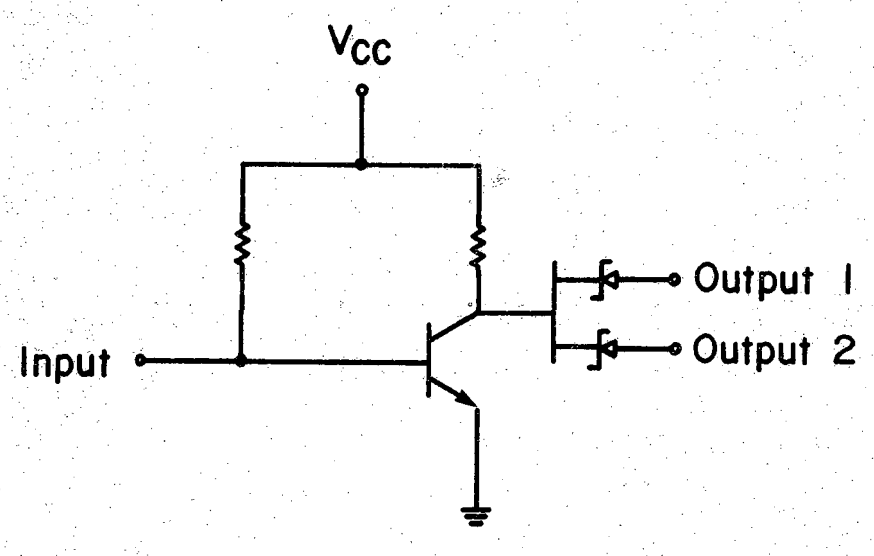


Figure 2.5 STL-like HBT Inverter [YuMcS84].

Table 2.1 Performance Characteristics of GaAs Designs.

Unit	Speed (ns)	Total Power (W)	Device Count (K)	Reference
ARITHMETIC				
32-bit adder (BFL D-MESFET)	2.9 total	1.2	2.5	[YaHiA83]
16x16-bit multiplier (DCFL E/D-MESFET)	10.5 total	1.0	10.0	[NaSuS83]
CONTROL				
gate array/SRAM (SDFL D-MESFET)	.15/gate	3.0	~8.0	[VuRoN84]
1K-gate gate array (STL HBT)	.40/gate	1.0	~6.0	[YuMcS84]
2K-gate gate array (DCFL E/D-MESFET)	.08/gate	0.4	8.2	[ToUcK85]
MEMORY				
4K-bit SRAM (DCFL MODFET)	2.0 total at 77 ° K	1.6	26.9	[KuMiS84]
16K-bit SRAM (DCFL E/D-MESFET)	4.1 total	2.5	102.3	[IsInI84]

Table 2.2 Performance Comparison of GaAs and Silicon [BasNu84].

	GaAs (DCFL E/D-MESFET)	Silicon (NMOS)
COMPLEXITY		
transistor count/chip	20 - 30 K	200 - 300 K
chip area	yield & power dependent	yield & power dependent
SPEED		
gate delay	50 - 150 ps	1 - 3 ns
on-chip memory access	0.5 - 2.0 ns	10 - 20 ns
off-chip/on-package memory access	4 - 10 ns	40 - 80 ns
off-chip/off-package memory access	20 - 80 ns	100 - 200 ns
IC DESIGN		
transistors/gate	1 + fanin	1 + fanin
transistors/memory cell		
static	6	6
dynamic	1	1
fanin (typical transistor size)	2 - 3	5
fanout (typical transistor size)	3 - 4	5
gate delay increase for each additional fanout (relative to gate delay with fanout = 1)	25 - 40%	25 - 40%

(1) GaAs logic gates switch considerably faster than Silicon logic gates. This is the most significant advantage that GaAs enjoys over Silicon in the context of this thesis. The principal reason for this GaAs advantage is the higher mobility of GaAs electrons. The electron mobility of 4000-5000 cm^2/Vs in n-channel MESFETs is approximately six times higher than the electron mobility of 800 cm^2/Vs for Silicon [NuPeB82]. A secondary GaAs speed advantage is the ability of GaAs to be fabricated as a semi-insulating material, which reduces parasitic capacitances [NuPeB82].

(2) The transistor count capability of GaAs is much lower than that of Silicon. The limitations of GaAs are due to problems with both power dissipation and yield.

Power dissipation is a concern of chip designers regardless of the technology. If total chip power consumption significantly exceeds two watts, then the associated heat may cause reliability problems. Special packaging techniques must then be used to remove heat more quickly from the chip. Although GaAs transistors require less power than Silicon transistors at similar switching speeds, fast GaAs transistors require more power than slow MOS transistors. GaAs designs requiring 0.2 milliwatts per gate will be limited to approximately 30,000 transistors if a two watt maximum power dissipation limit is imposed. An example two watt Silicon design, the Motorola MC68020, uses nearly 200,000 transistors [MaMoM84].

Because yield is *inversely* proportional to chip area, while transistor count is *directly* proportional to chip area, transistor count may be traded off for higher yields and, hence, lower costs. GaAs material is also currently of lower quality, i.e., it has higher defect densities, than Silicon [Walle84]. Therefore, GaAs wafers experience poorer yields than Silicon wafers with similar areas. This problem is compounded by the fact that the GaAs material is more expensive than Silicon, since Gallium is a rare material, and as a compound material, GaAs requires additional processing to create it and to verify its composition [Namor84]. In order to satisfy cost constraints, some GaAs designs may experience severe area limitations, and, hence, be limited to lower transistor counts.

(3) As indicated by the on-chip and off-chip memory access times, the speed advantage that GaAs enjoys over Silicon for the on-chip environment is not matched by an equal off-chip speedup. Inter-chip signal propagation speed is not significantly different for GaAs and Silicon chips since it is primarily dependent upon packaging considerations rather than integrated

circuit technology. Inter-chip signals are first limited to the speed of light; however, the media dielectric constant and capacitive loading on the signal lines can reduce signal propagation speeds to one third the speed of light or lower [MiSiF86]. Because of this, the penalty for inter-chip communication is higher, in terms of gate delays, for a GaAs design than it is for Silicon designs.

CHAPTER III
DESIGN CONSIDERATIONS FOR GaAs COMPUTER SYSTEMS

In Chapter II we presented an overview of GaAs device and logic families. The characteristics of GaAs were then compared with those of Silicon in order to illuminate the relevant differences between the two technologies.

In this chapter most of our discussion is based at least indirectly on the results of Chapter II. We first extend these results by defining more clearly those GaAs characteristics that influence computer design, and describe in general terms the appropriate strategies for dealing with some problems posed by GaAs technology. We then discuss the design approaches which appear to be suitable for computer system hardware and the compiler for a GaAs processor, concentrating on those approaches which are more valuable in GaAs system designs than in Silicon designs.

The discussions throughout the rest of this thesis are oriented to GaAs processor systems which execute compiled high level language (HLL) programs. No specific application area is targeted, as these discussions are intended for GaAs processor system design in general.

3.1 The Effect of GaAs Characteristics on Computer Design Strategy

The design of a GaAs computer system is intimately dependent upon the GaAs characteristics presented in the previous chapter; therefore, these characteristics deserve closer scrutiny.

As previously stated, GaAs transistors are significantly faster than Silicon transistors. The purpose of this thesis then is to determine the best approaches to maximize the exploitation of this GaAs advantage.

Unfortunately, GaAs chips generally have fewer transistors than Silicon chips. This obviously has an enormous impact on computer design. Minimizing chip count is desirable for performance, reliability, and cost reasons. Designs which minimize hardware complexity reduce chip count and are therefore very desirable.

A significant problem by itself, low transistor count severely compounds the problem caused by large inter-chip propagation delays. Together, these two problems may potentially limit the exploitation of the great strength of GaAs technology - its fast transistors.

A GaAs processor is able to execute instructions faster than a Silicon processor only if it has a corresponding increase in its supply of instructions and data. A fast GaAs processor should not be forced to spend its time waiting for information from its external environment. Three methods of resolving this information problem are to reduce the processor's need for off-chip information, increase the effective information transfer rate, or overlap the information transfers with processor execution.

Obviously, if the entire system could be built within one GaAs chip, the need to access off-chip information would be minimized. However, because GaAs chips are expected to contain fewer transistors than Silicon chips, the need for off-chip information will be even greater. Silicon processors are able to alleviate this problem by incorporating large amounts of on-chip memory in the form of a register file, cache, or microprogram store. Silicon's abundant transistors may also be used in the design of complex arithmetic units which, while performing complex functions, utilize each data element longer than simple arithmetic units do in performing simple operations. Because of the lower transistor count of GaAs chips, many of these Silicon solutions will not be available to a GaAs processor.

Increasing the effective rate of information transfer can be accomplished in two ways. The *information content* of each transfer can be increased or the *rate* of transfer can be increased.

Increasing the information content of transfers can be accomplished either by transmitting more bits per transfer or by eliminating redundancy within the transferred information. Upper limits on the number of bits per transfer are imposed by pin limitations of integrated circuits. However, Silicon supercomputers using SSI/MSI components, such as the Cray-1, are able to utilize this technique for data transfers [Russe78], but this technique is limited to operations on very regular data structures such as arrays. This thesis is not limited to the applications with well-structured data which are necessary for maximum performance on Silicon vector supercomputers. Redundancy removal, on the otherhand, generally requires an encoding and decoding capability. A compiler can effectively provide the encoding function on instructions; however, the decoding function must be performed by hardware resources within the processor. Many Silicon processors incorporate large on-

chip microprograms in order to provide instruction decoding. GaAs processors will likely not have the transistors available to provide such a thorough decoding capability.

A second technique for increasing the information transfer rate is to increase the effective rate of transfer. Silicon computer systems rely increasingly on cache memories, and multiple level memory hierarchies in general, to provide effective processor-memory transfer rates near the rate required by the processor. These traditional Silicon solutions may not be adequate for GaAs processors; however, because inter-chip signal propagation delays will take larger percentages of GaAs instruction cycle times.

Overlapping information transfers with processor execution is the final technique that we consider for reducing the GaAs processor information problem. Parallel execution and information transfer implies that information transfers are initiated before the processor has a need for this information. For a completely autonomous information transfer mechanism, separate datapaths and memory are required. This is more easily affordable in Silicon implementations than in GaAs.

Clearly, the low transistor count of GaAs chips and the large penalty for communication between them are real obstacles to the successful exploitation of the fast gates of GaAs technology. Silicon computer systems are designed within an implementation environment that has matched increased on-chip switching speeds with enormous levels of integration; therefore, the computer design techniques used in Silicon are not entirely compatible with the requirements imposed by GaAs technology. GaAs computer systems require approaches in both hardware and compiler design which differ from those traditionally used in Silicon computer design.

3.2 Hardware Design Issues

Given the general GaAs-driven design considerations of the previous section, we are now in a position to discuss design approaches for the hardware of GaAs processor systems.

We begin our hardware discussion by describing design approaches within the processor before moving to the off-chip memory environment. We first discuss our choice of processor configuration, followed by a presentation of suitable design approaches for the instruction pipeline, register file, execution unit, and instruction format. Our memory design discussions include virtual memory, memory hierarchies, both run-time and compile-time memory

management, and pipelined memory systems.

3.2.1 Processor Configuration

A number of different processor configurations are available as candidates for a GaAs implementation. Two representative Silicon processor designs are the Cray-1 [Russe78] and the Motorola MC68020 [MaMoM84].

The Cray-1 is a supercomputer implemented in Silicon emitter-coupled logic (ECL), and optimized to perform floating point operations on regular data structures such as arrays. Although the use of ECL allowed the Cray-1 to achieve a low 12.5 ns cycle time, a large number of these SSI and MSI parts were required to implement the processor. Because of the severe performance penalty for inter-chip communication in the GaAs environment, multiple-chip configurations such as these are not especially desirable. In fact, it has been reported that even if gate delays could be reduced to zero, the performance of a well-known supercomputer would only be increased by about 20 percent due to the dominance of off-chip delays [Gilbe84]. In contrast with the lower transistor count capability of GaAs compared to Silicon NMOS, GaAs has a higher transistor count potential than Silicon ECL. Therefore, the use of higher-density GaAs parts would improve the performance of vector processors such as the Cray; however, we are not concerned with such special-purpose environments in this thesis.

Processors such as the Motorola MC68020 take the opposite approach of the Cray, as they are implemented on a single chip. With this approach, the datapath (execution unit, register file, etc.) is on-chip, and the datapath execution time is not influenced by inter-chip signal propagation delays. This configuration has obvious advantages in a GaAs implementation environment. In fact, a single-chip processor configuration will achieve a higher relative performance increase through the use of GaAs technology than either Silicon mainframes or Silicon supercomputers [Gheew84]. It is because of this large potential increase in performance, in addition to a broader application market, that this thesis is oriented to the study of computer systems utilizing single-chip VLSI GaAs processors.

3.2.2 Single-Chip GaAs Processor Designs

The decoding and control logic (microcode) of the Motorola 68000 requires 68% of that chip's area [Katev83]. Although some may argue that this is

acceptable for a Silicon processor, it is clearly not tolerable for transistor-scarce GaAs processors. In contrast to the 68000, the Berkeley RISC-II processor uses only 10% of its area for decoding and control [Katev83]. The characteristics of processors such as the Berkeley RISC-II are worthy of further study for possible incorporation into GaAs processors.

The Berkeley RISC-II [Katev83] is an example of a reduced instruction set computer (RISC). Other well known RISC processors include the IBM 801 [Radin83] and the Stanford MIPS [HeJoG82]. RISCs are designed utilizing a philosophy which espouses the fast execution of the most frequently used instructions of an application environment, while avoiding the negative aspects of complexity associated with a Silicon implementation. One of the Berkeley RISC-II designers presented his view of instruction set design. "First, the most necessary and frequent operations (instructions) in programs were identified. Then, the data-path and timing required for their execution was identified. And last, other *frequent* operations (instructions), which *could also fit* into that data-path and timing, were included into the instruction set" [Katev83]. The result of this strategy is a processor with low decoding and control requirements, and consequently, low transistor count requirements. The 32-bit Berkeley RISC-II processor required only 41K transistors, while the 32-bit Stanford MIPS required but 25K transistors. These numbers are in contrast to the Motorola 68020, a complex instruction set computer (CISC), using approximately 190K transistors [MaMoM84]. In fact, the low transistor count requirements of the Stanford MIPS led to its being selected by the U.S. government as the architecture for its first 32-bit GaAs processor program [Barne85].

There are several processor features which result from the RISC design philosophy, and which will likely be inherited by GaAs processors as well. RISC processors generally implement only a few simple instructions, execute every instruction in one cycle, use a register-to-register execution model, and access off-chip data through explicit data load or data store instructions [Patte85]. RISC processors also rely more heavily on the capabilities of optimizing compilers. In fact, functions are implemented in hardware only if they cannot be performed at compile time [Radin83]. This constant evaluation of hardware-software tradeoffs, implicit in the RISC philosophy, which leads to both reduced hardware resource requirements, as well as demonstrated superior performance [PatPi82][HeJoG82], makes the RISC design philosophy very appropriate for GaAs processor implementations.

3.2.3 Instruction Pipeline Design

Instruction pipelining is frequently used in Silicon processors to increase execution speed. An example instruction pipeline for a Silicon processor might resemble Figure 3.1. In this example the instruction fetch time is equal to the instruction execution (datapath) time. This pipelined implementation results in approximately twice the execution speed of a non-pipelined implementation. This speedup is due to the overlapping of instruction fetching with execution, which allows the instruction memory system to completely satisfy the processor's instruction requirements.

A GaAs processor will require instructions at a faster rate than a Silicon processor, and it is very likely that conventional Silicon-like instruction pipelines will not satisfy a GaAs processor's instruction requirements for two reasons. First, the ratio of off-chip memory access delay to both on-chip memory access delay and arithmetic logic unit (ALU) delay is much higher for a GaAs processor than for a Silicon processor. Second, the lower transistor count of GaAs chips precludes the use of an on-chip cache for memory access speedup [MiFuH86]. In fact, if a GaAs processor utilizes a typical Silicon-like datapath (i.e. ALU, shifter, register file) design with an on-package instruction cache, the ratio of instruction fetch delay to datapath delay will be approximately three [Heage85]. For an off-package cache, the ratio may easily reach six. The Silicon instruction pipeline of Figure 3.1 does not fare very well under these conditions, as observed in Figure 3.2.

An instruction pipeline should not have a processor datapath underutilization built into it. Ideally, the effective instruction fetch time exactly matches the instruction execution time. The techniques for achieving good pipeline design in a GaAs processor are part of the discussions of the next few sections. Pipelines which result from careful GaAs computer system design approach those shown in Figures 3.3, 3.4, and 3.5. Figure 3.3 is the result of increasing the effective *rate* of instruction fetches. Figure 3.4 is the result of increasing the *content* of instruction fetches. Figure 3.5 is the result of decreasing the *required rate* of instruction fetches.

3.2.4 Register File Design

As already indicated, the RISC design philosophy typically results in processors which use a register-to-register execution model. In addition to its contribution to complexity reduction, the register-to-register execution model has other desirable features for a GaAs implementation.

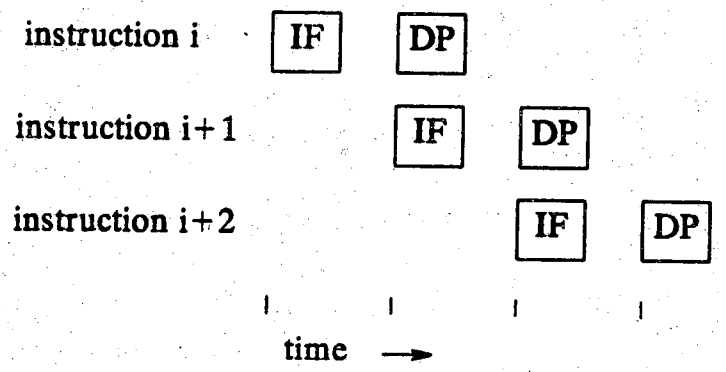


Figure 3.1 Example Silicon Instruction Pipeline [FurMi85].

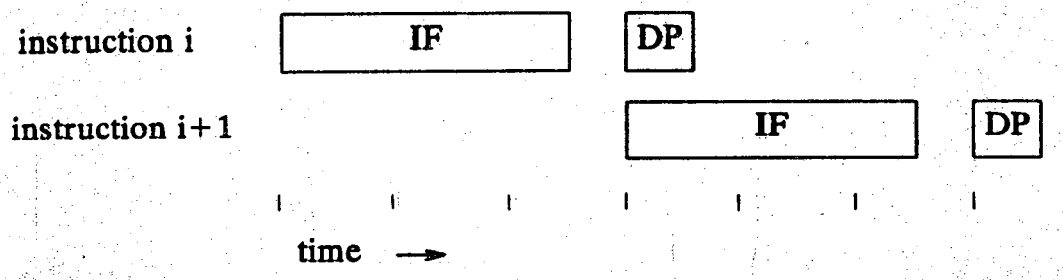


Figure 3.2 Example Silicon Pipeline Implemented in GaAs [FurMi85].

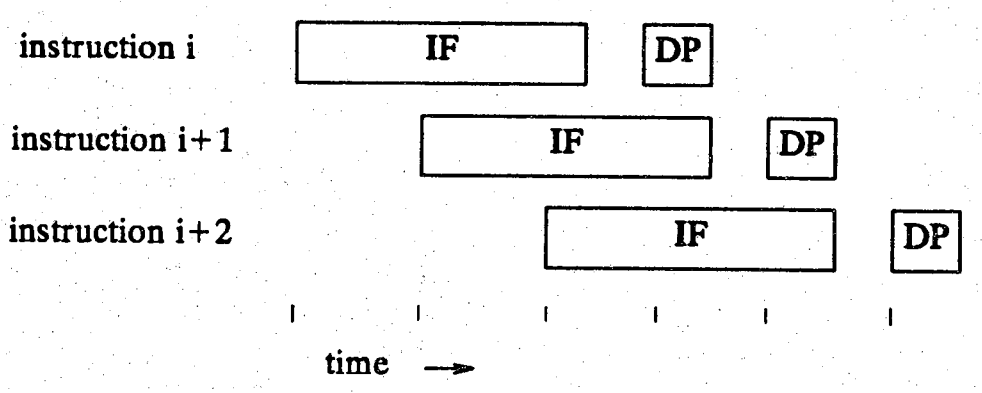


Figure 3.3 Example GaAs Instruction Pipeline with a Pipelined Memory [FurMi85].

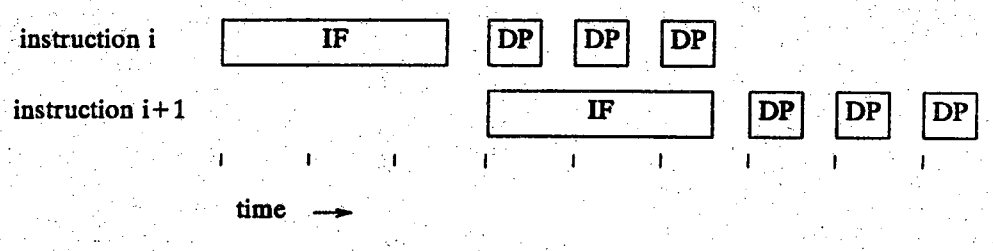


Figure 3.4 Example GaAs Instruction Pipeline with Instruction Packing [FurMi85].

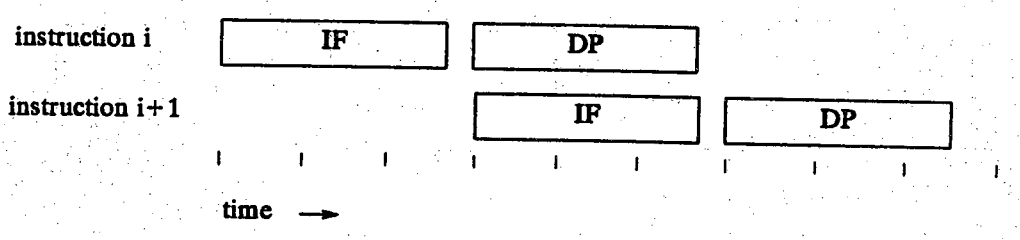


Figure 3.5 Example GaAs Instruction Pipeline with Long-latency Datapath Elements [FurMi85].

First, registers are fast on-chip memory. The access time of a register is much shorter than that of off-chip memory, and this difference is more pronounced for a GaAs processor. As stated earlier, maximizing on-chip memory is of great importance for a GaAs processor.

Register files are generally more effective than caches at the small capacities available in a GaAs processor. For example, an instruction cache containing 16 32-bit words can be expected to achieve a hit ratio of only 70%, while an equivalent data cache would hit only 55% of the time [Smith85]. Since register file data placement is performed by the compiler instead of a run-time caching mechanism, register file accesses never miss. In addition, register files aren't burdened by the hardware overhead which is required by caches.

A register address is directly specified within the instruction. Therefore, an address calculation is not required and no virtual to physical translation need be performed.

The short length of register addresses leads to compact code which can be expected to increase the hit ratios of program memory accesses at the higher levels of the memory hierarchy.

The importance of register files thus established, this section presents the design issues involved with register memory cells and register file partitioning.

3.2.4.1 Register Cell Design

In Silicon processors, performance depends heavily on the speed of datapath elements such as the register file. For this reason, register cell designs emphasizing access speed and multiple read and/or write ports are common. As an example, the microcode pipeline of the HP-FOCUS processor [BeDoF81] is shown in Figure 3.6. Because of the fast access time of its on-chip microcode memory, fast register file access was also needed. Figure 3.7 shows the register cell design used in the HP-FOCUS, which allows two simultaneous data reads or writes and supports the 55 ns cycle time.

In GaAs processors, fewer transistors will be available for register file implementation; therefore, simple register cells are very advantageous. Even if simple register cells reduce the datapath speed, performance may not be negatively impacted. One approach to reducing a high ratio of instruction fetch delay to datapath delay is to increase the datapath delay. This may seem very undesirable; however, reducing the effective instruction fetch delay, which is intuitively the best approach, introduces new problems which are

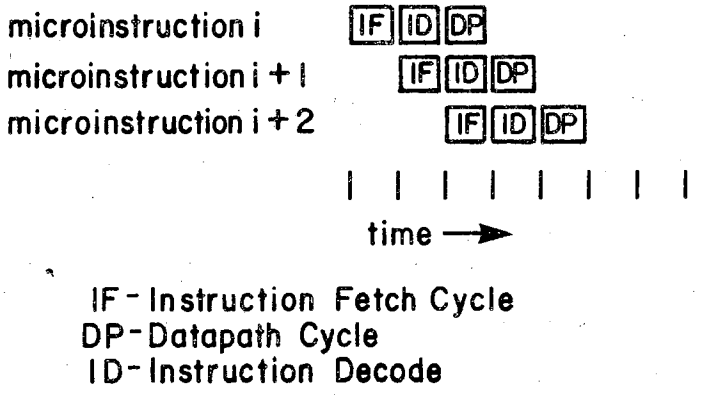


Figure 3.6 Microcode Pipeline of the HP-FOCUS Processor [BeDoF81].

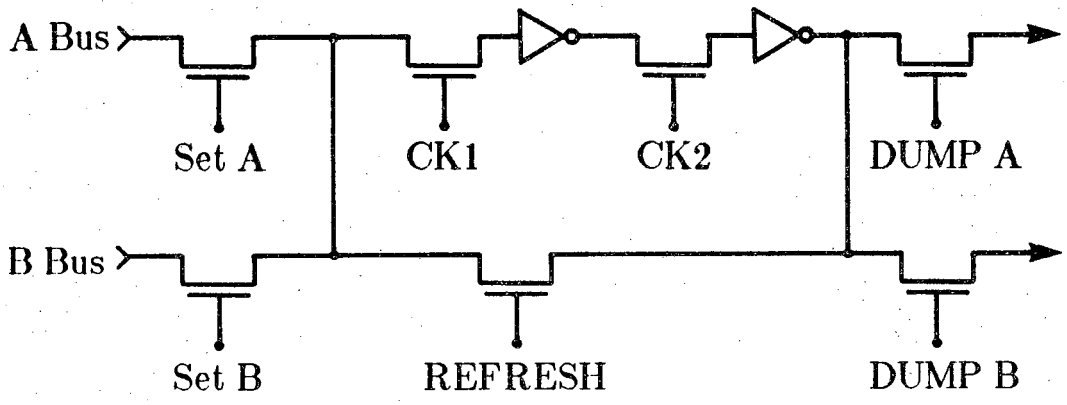


Figure 3.7 Register Cell Design of the HP-FOCUS Processor [BeDoF81].

discussed later. Using slow register cells at least has the advantage of a low resource requirement. The selection of an appropriate register cell design must be considered in the context of the entire system. There is certainly a greater disparity in access time between off-chip memory and a slow register cell than between a fast and a slow register cell. It is conceivable that a larger number of slow registers may provide better system performance than a smaller number of fast registers.

Simple register cells which make good candidates for GaAs processors are shown in Figures 3.8 [Sherb84] and 3.9 [MiFuH86]. Figure 3.8 shows the register cell design of the Berkeley RISC-II. Its transistor and area requirements are much lower than those of the HP-FOCUS. Although this register cell allows parallel reads, its read time is slower than the cell of the HP-FOCUS [Sherb84]. Figure 3.9 shows a register cell with a single read bus. This cell uses less area than the other cells, but requires sequential reading and writing.

3.2.4.2 Register File Partitioning

Register files generally succeed at reducing the processor's need to access off-chip data during the execution of HLL procedures. However, at procedure boundaries (calls, returns) the register file values must be stored to memory and new values loaded in. This massive off-chip communication is bad enough in Silicon implementations, but is even more damaging to a GaAs processor.

To alleviate this procedure boundary problem, multiple window register file schemes have been introduced by Silicon designers, and used in processors such as the Berkeley RISC-II [Katev83]. In a multiple window register file, each procedure is allocated one window for its data. Whenever a procedure call or return is encountered, instead of emptying and refilling the register file, a new window is allocated, perhaps by simply changing a pointer value as in the Berkeley RISC-II. The only time that emptying and refilling is required is on an "overflow" or "underflow." An overflow occurs when a procedure call is encountered and no unused windows exist. An underflow occurs when a procedure return is encountered and the values of the returned-to procedure were saved to memory because of a previous overflow. The eight window scheme of the Berkeley RISC-II was responsible for an approximate 50 percent reduction in the number of data loads and stores [Patte85]; consequently, this technique shows potential applicability for a GaAs processor implementation.

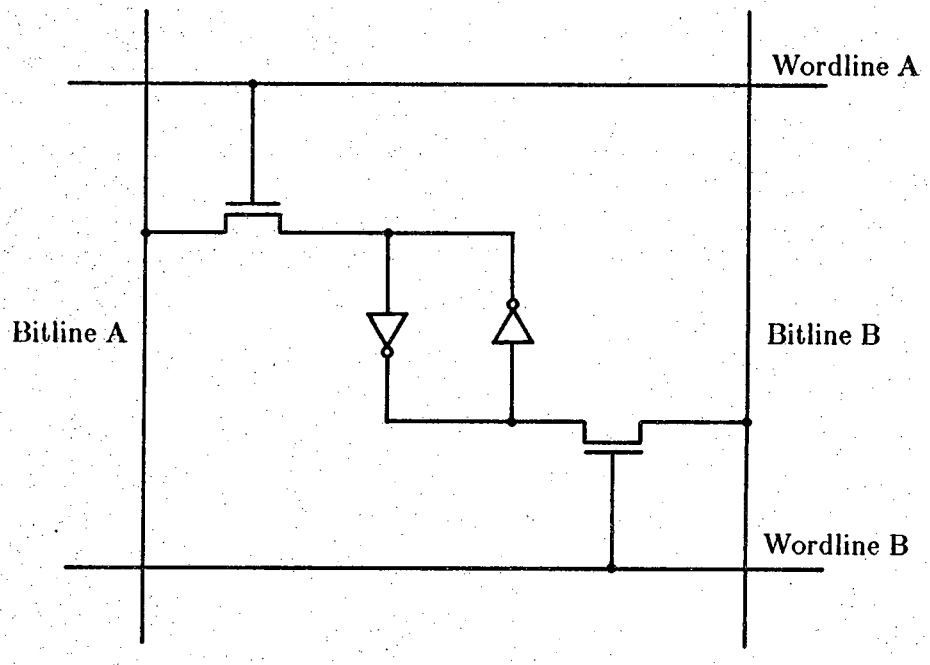


Figure 3.8 Register Cell Design of the Berkeley RISC-II Processor [Sherb84].

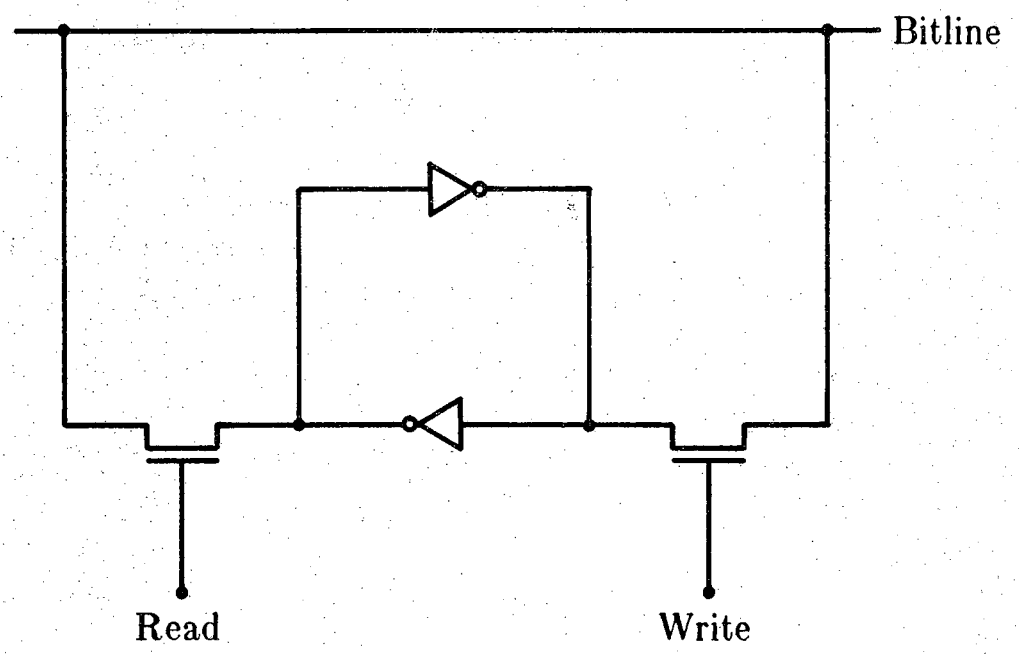


Figure 3.9 Register Cell Design Employing a Single Read Bus.

A major problem which prevents the implementation of the Berkeley RISC-II register window scheme in GaAs is the large number of registers required for its implementation - 138. In fact, the Berkeley implementation would consume nearly all the transistors available to a 30K transistor GaAs processor. For this reason two variations of the Berkeley method are potential candidates for a GaAs implementation. They are multiple window schemes with a) variable-size windows and b) background loading and storing.

Multiple window register files with variable-size windows have been discussed in the context of Silicon implementations [Katev83]. The real advantage of this approach is that more windows can be formed from fewer registers, as compared to a fixed-size approach. The reason for this is that most procedures use very few local variables and formal parameters in well structured programs [Tanen78]. A fixed-size window scheme will encounter very poor register file utilization, and this is quite undesirable in a GaAs implementation where off-chip delays are large. When only enough registers are allocated to minimally satisfy each procedure's needs, additional registers are made available to implement more windows and reduce overflows and underflows. The drawbacks to this approach are additional hardware requirements and added delay for register address calculation, and additional overhead for procedure calls/returns and overflow/underflow handling. A compromise approach is to provide multiple windows but limit the number of sizes which may be used, and choose sizes to reduce complexity [Furht85].

Multiple window register files with fixed-size windows and background mode loading and storing have also been discussed [Katev83]. The advantage of this approach is that intelligent preloading and prestoring may reduce the frequency of overflows and underflows. The primary drawbacks to this approach are the additional processor-memory bandwidth required, and the need for an independent input-output controller capability.

3.2.5 Execution Unit Design

The effect that the low transistor count and large off-chip delays have on register cell design for a GaAs processor is felt in the execution unit design as well. Once again, scarce hardware resources should not be used to create or exacerbate a mismatch between execution unit information needs and off-chip memory system capabilities. The execution unit design for a GaAs processor should instead be part of a system-level effort to achieve a match between these two.

If the effective off-chip memory access time cannot be cheaply reduced to match the datapath delay, then another method of matching the off-chip memory and datapath delays may be appropriate. A useful approach for GaAs processor execution unit design is to approach the ideal "instruction fetch delay - datapath delay" equality from the direction indicated in Figure 3.5. This approach is summarized as "reducing the execution unit's need for off-chip data in some useful way." Two methods for accomplishing this are to 1) Remove resources from the execution unit in order to slow down the execution of primitive operations, and reallocate the resources elsewhere, perhaps to the register file. 2) Maintain or add resources to the execution unit only to support complex operations which require large amounts of time. These two approaches will be in evidence throughout this section.

3.2.5.1 Adder Design

Silicon processors typically require high-speed adders to maximize their performance. Again, this is because of the relatively fast access times of Silicon memories in comparison to datapath times, especially when the memory is on-chip. For example, the HP-FOCUS utilized a full carry lookahead adder to satisfy its 55 ns cycle time in a Silicon implementation.

The adder designs available for a GaAs implementation range from the high-speed, high-resource-requirement full-carry-lookahead adder to the low-speed, low-resource-requirement ripple-carry adder. Others which have speeds and resource requirements between these two extremes include conditional-sum and carry-select adders [Hwang79].

As in register cell design, simple designs are advantageous for GaAs adders. In addition to transistor count differences, the above adder designs exhibit differences in regularity, which introduce chip area differences as well.

VLSI implementation environments introduce complexity into adder performance evaluation [Sherb84]. In a Silicon SSI/MSI TTL (transistor-transistor logic) implementation, adder speed is determined by the number of gate delays required to obtain the final result. In a VLSI implementation, designers have potential opportunities for performance enhancement, such as in varying transistor sizes to improve speed in critical paths. Additional variables are also introduced, such as large signal propagation delays because of long wire lengths, large fanins, and large fanouts.

In a Silicon VLSI environment, it has been shown that the regular layouts and low fanin/fanout requirements of adders such as the ripple-carry and

carry-select, reduce the performance advantage of the traditional carry-lookahead approaches, which are very irregular [Sherb84].

From both a performance and implementation cost standpoint, ripple-carry and carry-select adders are more suitable than traditional carry-lookahead adders for implementation into a GaAs processor.

3.2.5.2 Multiplier/Divider Design

The frequency of use of multiplication and division operations varies from application to application. In a distribution of instructions from a computer aided design (CAD) application environment, multiplies were only 3 percent of all instructions executed [McDan82]. However, the high frequency of multiplies in signal processing applications prompted the designers of the Texas Instruments TMS320 [MaCaM82] to include a 200 ns on-chip hardware multiplier. In this section we present multiplication and division techniques which are advantageous for GaAs processor implementations. Designs for both high frequency and low frequency usage will be given.

Silicon processors which are targeted to general purpose applications typically utilize the datapath adder to perform multiplication and division. Silicon CISCs implement multiplication and division with microcode routines, while Silicon RISCs use special multiply-step or divide-step instructions which are stored within the program. In special purpose application environments where multiply and/or divide operations are more frequent, Silicon processors incorporate additional hardware. Example hardware multipliers include an implementation of the modified Booth algorithm in the TMS320 and an array multiplier in the NEC IPP [NuKuM84]. An example division technique is the "division by repeated multiplication" method used in the IBM 360/91 [AnEaG67] which requires a fast multiplier.

Multiplication and division operations require relatively large amounts of time; therefore, if justified by frequency of use, additional hardware support is desirable for GaAs processors as well. In addition to faster speed, an advantage in using a hardware implementation of a multiplication/division algorithm is that less off-chip information (fewer instructions) is required than in traditional software approaches. However, candidate approaches must satisfy the limited transistor count typical of GaAs.

The standard add-and-shift multiplication technique and subtract-and-shift division technique require the fewest hardware resources and are quite desirable from this standpoint, especially in general purpose environments.

Minor hardware additions, such as those incorporated in the Stanford MIPS [HeJoG82], improve these two techniques; achieving multiplication in n/2 steps and division in n steps, where n is the word length.

Silicon CISC implementations of these multiplication and division techniques have an advantage over Silicon RISC implementations in one respect. A Silicon CISC must only fetch one instruction from off-chip memory in order to execute either a multiply or divide, while a RISC must fetch several instructions. A microcoded CISC, therefore, does a better job of reducing off-chip communication needs, and in principle, this is very desirable for GaAs processors. This does not imply that microcoded CISCs are appropriate for GaAs; however, achieving the higher information content of CISC-like instructions is desirable. Modifying RISC principles to allow a single instruction to represent a sequence of add-shift operations (or subtract-shift operations) is worthy of consideration. This idea is presented in greater detail in the next section.

The hardware multipliers used on the TMS320 and NEC IPP require too many hardware resources to be incorporated into a GaAs processor. If the need for fast multiplication is very strong, two hardware approaches may be used.

A serial multiplier with moderate hardware resource requirements may be constructed. A serial multiplier presented in [Ghana83] requires 64 cycles to perform a 32-bit by 32-bit multiplication. Since each cycle period need only be long enough to allow signal propagation through a flip-flop and minimal logic, a faster clock may be used to achieve a serial multiplication time much lower than that achieved by the datapath adder. This solution makes very good use of the architectural strength of GaAs - its fast gates. This type of iterative approach has also been cited by an early GaAs architecture researcher [Gilbe84] as being desirable for GaAs. An off-datapath serial multiplier also allows concurrent multiplication and datapath execution.

An alternative approach is to incorporate a hardware multiplier into a coprocessor. The multiplier design in such an implementation is not as constrained as in the processor; therefore, designs with greater hardware requirements are more appropriate here. Of course, this technique may be extended to allow two or more such coprocessors if the application environment demands it. This approach also allows the concurrent operation of the coprocessor and the processor's datapath.

3.2.6 Instruction Format Design

The use of the RISC design philosophy in GaAs processor design might appear to reduce instruction format design to a trivial problem. This is of course not so. A legitimate concern for GaAs processor design is the effect of the instruction format on the timely transfer of instructions to the processor. The instruction bandwidth requirement of a processor is strongly dependent on the instruction format. Although the basing of design decisions on instruction bandwith alone is not to be encouraged, this architectural metric acquires added significance in the GaAs environment, but should be used within the context of the system design.

Compact programs require a lower memory-processor bandwidth and are more beneficial in GaAs processor systems than in Silicon processor systems for at least two reasons.

First, the technology used to implement the memory at the highest levels of the memory hierarchy will likely be GaAs. Since GaAs memory chips will likely remain less dense than Silicon memory chips, GaAs caches, etc., will be relatively small. It has been shown that memory size is the single most important factor in cache hit ratios [SmiGo83], and that hit ratios increase rapidly at small cache sizes before leveling off at high cache capacities [Smith85]. Since a decrease in program size is equivalent to an increase in memory size, compact programs are indeed very desirable in a GaAs processor system.

Second, because of the extremely fast instruction cycle times possible in a GaAs processor, a memory access miss in a GaAs processor system will likely entail a longer delay, in terms of instruction cycles, than a memory access miss in a Silicon processor system. It is, therefore, more important to minimize these misses in a GaAs processor system.

The major disadvantage of the RISC design philosophy in a GaAs implementation is the generally low information content of RISC instructions. Of course, it is the very simplicity of RISC instructions which lead to their low decoding logic requirements. Therefore, any attempt to reduce program size through increased encoding of instructions must be done so as to minimize its impact on decoding complexity.

We discuss two methods for increasing program compactness which can have little impact on a GaAs processor's decoding requirements. The first approach relies on the high dynamic frequency of short immediate fields and few operand addresses; while the second approach makes use of the repetitive nature of some complex operations such as multiply and divide.

3.2.6.1 Frequency-based Instructions

Techniques based on Huffman codes [Huffm52] are frequently considered in instruction set design. Huffman coding is a technique for assigning the most frequent instructions the smallest encodings. A pure Huffman implementation would require sequential decoding and much hardware, and is not a serious candidate for a GaAs processor. However, the concept of providing small encodings to frequent occurrences is very applicable for GaAs processor instruction sets.

Compact instruction formats may be designed to incorporate small immediate fields and few address fields. The resulting reduction in program size is not due to an explicit encoding function operating on these fields, but results from the high dynamic frequency of small immediate data values and both one-address and two-address instructions in real programs. In one study of XPL programs [AleWo75], it was shown that 61 percent of the branch distances required eight bits or less, while 81 percent could be represented with 12 bits. They also found that 47 percent of the numeric constants could be represented by only four bits, and that 87 percent required eight bits or less. It is also estimated that 87 percent of all assignment statements require only two operand addresses [Myers82].

Compact instruction formats which result from using short immediate fields and few operand addresses have three beneficial aspects for a GaAs implementation.

First, as just mentioned, the proper design of immediate and operand fields can be expected to reduce total program size and provide the benefits for a GaAs implementation described above.

Second, this approach takes advantage of the dynamic characteristics of program behavior. Small immediate values and few operand addresses are not the output of an encoding algorithm, but occur naturally and frequently in real programs; therefore, there is no need for a significant decoding function within the processor to "undo" any additional encoding.

Third, compact operations may be packed into a single instruction. Because pin limitations of a GaAs processor will limit the size of instruction fetch transfers, multiple operation fetching, as shown in Figure 3.4, is only possible for short operations. A well-designed packed instruction format can improve the performance of a processor in a long-latency off-chip environment.

Many Silicon instruction sets display varying levels of program compactness, and some even employ operation packing. The longest immediate

data values for many Silicon processors require 32 bits of information. In order to include immediate fields in single-word instructions to represent 32-bit values, extremely long instructions would be required.

Even relatively sparse instruction formats, such as the Berkeley RISC-II, take advantage of the low frequency of use of such long immediate values by only supporting short immediate values within a single instruction. The use of very large immediate values requires two RISC-II instructions.

The Stanford MIPS limits its maximum immediate field size to 24 bits to allow all immediate values to be used within single instructions. However, the MIPS instruction set takes advantage of small immediate data values by packing a second operation into instructions which require short immediate fields. This operation packing also makes use of the high frequency of one- and two-address operations because the instruction fields for the packed operations are only large enough for two-address operations.

The ability of the Stanford MIPS to execute two operations per instruction fetch is limited by the occurrence of long immediate values and also by the ability of its compiler to find suitable useful (non-NOOP) packing candidates. A more complex instruction format also results from the MIPS style of packing. This results in a 10 percent increase in the MIPS instruction cycle time [Patte85], in addition to increased decoding hardware requirements.

The Transputer [Whitb85] relies very heavily on the high frequency of small immediate values as it only provides four bits of immediate field in every 8-bit instruction. Larger immediate values must be built from multiple instructions four bits at a time. The small size of its instruction format allows the Transputer to pack four such instructions (from now on we call these *operations*) into a single packed instruction.

The Transputer is better able to meet its maximum rate of four operation executions per instruction fetch because of a somewhat different definition of "operation." The Transputer uses even more primitive operations than the MIPS or RISC-II. The use of a large immediate data value is implemented by a sequence of "build immediate field" operations. However, these operations have a 50 percent overhead as only four of the eight bits contain actual data. However, the rigid field boundaries of the Transputer's instruction format can be expected to result in a simpler decoding function than required by the MIPS instruction format.

Compact instruction formats must be designed with a good understanding of the instruction requirements of the intended application environment. However, the exploitation of the high frequency of usage of small immediate

values and few operand addresses may provide significant benefits for a GaAs processor implementation.

3.2.6.2 Context-based Instructions

Huffman codes are based on the frequency of usage of data items without considering the environment surrounding the data items. Because instruction executions are not independent of each other, additional compaction opportunities are available.

A compaction technique which uses context information, in addition to the instruction frequency information used by Huffman-based techniques, is conditional coding [Hehne76]. In this technique the encoding of the next instruction to be executed is dependent upon the probability of its occurrence, in the context of the execution of the current instruction. Therefore, if there are n instructions in the instruction set, then each instruction has n different encodings - one associated with each of the n possible preceding instructions. A strict implementation of this scheme is not practical for a GaAs processor; however, the concept of using context information to reduce program size is applicable.

A less rigorous, but simpler, technique is to replace frequent instruction sequences with a single instruction [Hehne76]. In fact, this technique is typically used on microcoded CISCs. A program consisting of CISC macroinstructions is generally more compact than a program containing RISC instructions; because each macroinstruction corresponds to a sequence of microinstructions, while each RISC instruction corresponds to a single microinstruction-like instruction. It is possible that this CISC mechanism can be utilized to good advantage in some GaAs processor environments.

If justified by frequency of use, complex instructions such as multiply, divide, and perhaps even multiply-accumulate, may be added to the instruction set of a GaAs processor. Even if a transistor-scarce GaAs processor cannot support the hardware to directly execute these operations, and must instead use the main datapath, there are advantages to using complex instructions.

For example, the implementation of multiply on a RISC can be performed in a number of ways.

A linear sequence of multiply-step instructions can be used for each multiply in the program. For a processor such as the Stanford MIPS, this may require nearly 20 instructions per multiply. If multiplies are 10 percent of all instructions executed, then this technique nearly triples the program size.

Alternatively, a loop containing as few as one multiply-step instruction can be used. However, introducing loops into programs is not generally desirable both because of the time wasted on looping overhead, and because of the large number of instructions required to perform branch fillin if the GaAs processor is highly pipelined.

A third technique is to include a linear sequence of multiply-step instructions into a system procedure which is callable from anywhere within the program. This technique, therefore, requires that a procedure call and return be executed for each multiply instruction. Beyond the normal overhead associated with procedure calls, this method degrades the execution locality, possibly decreasing memory hit ratios.

The implementation of a single multiply instruction entails none of the above disadvantages; however, complexity is introduced into the pipeline control mechanism. Single-cycle instruction execution is a feature of "true" RISCs because it leads to simple pipeline control. When a CISC-like multiply instruction is encountered, the processor will likely spend a long time executing it. Therefore, the instruction pipeline must be halted. If the instruction memory is also pipelined, then a time delay will probably exist before the entire memory pipeline can be halted, and buffering may be needed. Clearly, the benefits of CISC-like instructions must be weighed against this implementation complexity, and the decision to use CISC-like instructions should be considered in the context of the entire system.

3.2.7 Memory System Design

Memory system design is an extremely important issue in a computer system containing a GaAs processor. The capabilities of a fast GaAs processor cannot be fully exploited unless the memory system is able to satisfy the processor's increased information needs. The low transistor count of GaAs memory chips and the long inter-chip delays, with respect to the cycle time of a GaAs processor, both limit a memory system's ability to provide the capacity and transfer bandwidth required by a GaAs processor.

3.2.7.1 The Role of Virtual Memory

The application environment also has a large influence on the design of a memory system for a GaAs processor system, particularly with regard to the issue of virtual memory. A virtual memory system is one which provides a mapping from logical addresses to physical addresses [Denni70]. Logical addresses are produced by the compiler, while physical addresses are used for accessing physical memory.

There are several advantages attributable to virtual memory systems. Since virtual memory systems normally contain a large capacity backing store such as a magnetic disk, both the programmer and compiler are able to generate code without regard to the actual size of main memory. The burden of memory allocation is transferred from the programmer to the operating system; and multi-tasking and protection mechanisms are easier to incorporate. Virtual memory is used extensively in computer systems in universities and industry.

The disadvantage of virtual memory is that the logical to physical address translation is necessary. Virtual memory systems, therefore, have longer memory access latencies than non-virtual memory systems. In applications which require the speed of a GaAs processor, the performance loss due to a virtual memory implementation is quite expensive. Many Silicon supercomputers do not use virtual memory; they instead rely on large main memories. Applications such as these, as well as many embedded applications with special-purpose programs and relatively small memory requirements, will not require virtual memory in GaAs processor systems.

3.2.7.2 Memory Hierarchy

Large memory systems generally utilize several different types of memory components to achieve a favorable cost-performance balance. The fastest memory parts are also generally of the lowest capacity, as well as being the most expensive; while slower, higher capacity memory is generally the cheapest. Memory system designers try to provide memory speeds approaching the speed of the fastest technology, while achieving a cost per bit approaching the cost of the cheapest technology. This is achieved through the exploitation of the locality of references which exist within most computer programs.

Two types of program locality exist [Denni72]. Temporal locality means that once information is used it will likely be used again within a short time span. Programming constructs which cause this include loops, recursive

procedures, and activation record accesses. Spatial locality means that when a unit of information is used, its neighboring information will likely be used within a short time span. Programming constructs which cause this include sequential program execution, activation record accesses, and structured data accesses.

These two localities of reference are responsible for the success of hierarchical memory systems. By keeping the information which is within the current referencing locality in the fastest memory, fast information access will be available to the processor a large percentage of the time. This clearly would not be the case if accesses were uniformly distributed throughout the entire memory system.

The memory technology best able to meet the speed requirements of a GaAs processor is GaAs. Therefore, GaAs memories will likely be used to implement the highest level of the memory hierarchy. The lower levels of a memory hierarchy usually incorporate cheaper, larger, and, hence, slower memories. As the capacities increase at each successive level, fast, less-dense technologies (i.e. GaAs) lose their speed advantage over slow, dense technologies (i.e. Silicon). This is because off-chip signal propagation delays are larger for low density chips, which require large amounts of board area. This is especially true when low-density memory chips cause board area capacities to be exceeded, requiring additional inter-board communication.

3.2.7.3 Run-time Control of Hierarchical Memory Systems

Most Silicon hierarchical memory systems make extensive use of run-time information control mechanisms. For example, caches, which are often used as the fastest element of the memory hierarchy, use hardware to decide at run-time what information is to be located within the cache. Similar run-time approaches are commonly used for main memory as well.

Two techniques are frequently used in Silicon systems to decide what information should be moved into a higher level of the memory hierarchy. The simplest method is to move the information into the higher level when it is needed by the processor and not already at the higher level. The processor is required to wait until the requested information is moved and this may result in a considerable delay. Another common method, known as prefetching [Smith78], relies on a form of spatial locality known as sequential locality. Sequential locality is caused by the sequential execution of most programs and the sequential access of structured data. In prefetching techniques, information

which is located at addresses slightly higher than the currently accessed addresses is moved into a higher level of the memory hierarchy. This run-time-initiated information movement is potentially very advantageous in a GaAs processor system.

In addition to deciding what information should be moved into a higher level of the memory hierarchy, run-time mechanisms must decide where the information is to be located within the higher level.

Existing Silicon memory systems vary in the amount of power they give to the run-time hardware in deciding where in the higher level to locate the moved information. Three placement policies are commonly used in Silicon caches: fully associative, set associative, and direct mapped [Smith82]. In a fully associative memory level, the run-time hardware is free to locate the information in any location. In a set associative memory level, the run-time hardware is constrained to locate the information within a subset of the memory level called the "set." In a direct mapped memory level, the run-time hardware has only one valid location in which to locate the information.

In Silicon caches, the fully associative technique generally achieves the highest hit ratios among these three methods, followed by the set associative and direct mapped techniques [Smith82]. However, at small cache sizes, the difference in hit ratios between the three techniques becomes insignificant [SmiGo83].

In the fully associative and set associative techniques, the run-time hardware decides where in the higher level to locate the moved information. The movement of information into the higher level implies an equal-volume movement of information out of the higher level. Therefore, the location in the higher level is chosen so as to hopefully displace information which is no longer needed by the processor. Of the information candidates for displacement, the information which will not be needed by the processor for the longest period of time cannot be predicted by a run-time mechanism.

Three common replacement algorithms are the least recently used (LRU), first in first out (FIFO), and random (RAND) methods [Smith82].

The LRU technique exploits temporal locality by replacing the information which saw last recent use. A strict implementation of this technique requires excessive overhead if a fully associative policy or set associative policy with large set size is used. A strict implementation is, therefore, generally restricted to small set sizes; however, LRU approximations may be used for larger sets.

The FIFO method does a poorer job of temporal locality exploitation but also requires less overhead for its implementation.

The RAND method makes no attempt to exploit temporal locality but is easily implemented.

If the information which is being displaced was modified while in the higher hierarchical level, then this information must be stored back into memory at the lower levels. There are two common techniques used to achieve this storing.

The copy-back method [Smith82] waits until the information is displaced before storing to the lower levels. In order to avoid storing information which has not been modified, a "dirty bit" is commonly associated with a block of information. If this bit is set, then the information has seen recent modification while in the higher level and should, therefore, be written to the lower level. If the dirty bit is not set, then no storage to the lower levels is necessary.

In the write-through technique [Smith82], the information is written to the lower levels as soon as the processor modifies it. If the processor must wait for its data stores to complete before continuing execution, then this technique will introduce long delays into program execution. However, if buffering is used, such as in pipelined memory systems discussed later in this chapter, then no additional delays are introduced. The write-through method also does not require the overhead associated with the copy-back approach.

As already mentioned, memories consisting of GaAs chips will generally be smaller than Silicon versions. Because of the minimal run-time overhead associated with the direct mapped implementation just discussed, as well as its relatively good performance at small memory capacities, this approach is desirable for the memory at the highest level of the memory hierarchy.

At the lower memory levels, the set associative and fully associative techniques become advantageous because of the larger capacity of the lower levels. The LRU, FIFO and RAND replacement policies are all viable methods at these lower memory levels as well.

If pipelining is employed in the memory system design, then the write-through method is preferable to the copy-back approach for the cache and main memory levels of the memory hierarchy. Write-through eliminates both the need for long-latency copy-back operations and reduces coherency problems by ensuring that the lower memory levels are continuously updated.

3.2.7.4 Compile-time Control of Hierarchical Memory Systems

The compiler has three advantages over run-time hardware in the implementation of memory hierarchy information control mechanisms. First, it has a larger base of knowledge from which to make decisions since it has a view of the entire program. Second, it presumably has more time to implement a more optimal strategy. Finally, compiler algorithms don't require additional hardware.

Although the temporal and spatial localities of reference allow run-time information control mechanisms to work well, there are times when the required locality is missing. LRU-based run-time mechanisms, which exploit the referencing localities, are responsible for bringing information into the higher levels of the memory hierarchy, and for deciding which information at the higher levels should be replaced. Two instances where LRU-based mechanisms fail are the following.

(1) Information is accessed for the first time after a long period of non-use.

(2) Information is accessed once but will not be accessed again for a long period of time.

The compiler has the potential to detect this nonlocalized behavior and the power to help the run-time mechanism to perform more effectively. Since the delays caused by nonlocalized accessing patterns are more costly, in terms of instruction cycles, for a GaAs processor, the effort expended on compiler design and the increased time for compilation are offset by greater gains in performance. A more thorough discussion of this issue is presented in Section 3.3.2.2.

3.2.7.5 Pipelined Memory Systems

Pipelining is a common technique for speeding the execution of long-latency operations. Pipelining is frequently used within the processor to overlap instruction fetching, decoding, and execution, etc. It is also used for implementing complex arithmetic operations as in the IBM 360/91 [AnEaG67].

Because of the longer relative delays associated with memory accesses in a GaAs processor system, memory pipelining is a very attractive approach. In fact, memory pipelining has already been used in Silicon systems, on the Amdahl 470V/6 [Smith78]. Memory pipelining is even more feasible in a GaAs processor system because the long access delay of off-chip memory is not necessarily due to slow memories, but instead due to long inter-chip delays.

These delays are easily pipelined. An example of a pipelined memory system is shown in Figure 3.10. This pipeline consists of three stages. In the first stage the address (and data if write) is propagated from the processor to a latch physically near the memory. In the second stage the memory is accessed, and, for a memory read, the data is stored into another latch physically near the memory. The third stage is used, for a memory read, to propagate the data to the processor.

In a three stage pipeline such as this, three memory accesses may be concurrently serviced. Assuming a GaAs processor system in which the ratio of instruction fetch delay to datapath delay equals three, this memory system will produce a pipeline such as in Figure 3.3. Clearly, pipelined memory systems decrease the "effective" memory access delay, even though the total memory access latency is unchanged. Pipelined memory systems are extremely valuable in GaAs processor systems, because they are so successful at increasing the information transfer bandwidth between processor and memory. However, as discussed in the next chapter, the increased pipeline depth resulting from pipelined memory implementations introduces performance problems associated with program branches. Overall, though, pipelined memory systems should have a positive effect on the performance of GaAs processor systems.

3.3 Compiler Design Issues

The high penalty for inter-chip communication and low levels of integration of GaAs chips combine to increase the importance of the compiler in GaAs processor systems. Without the support of a powerful compiler technology, GaAs processor systems will struggle to fully exploit the speed advantage of GaAs technology, except possibly for selected special-purpose applications.

As discussed earlier, GaAs processor system design utilizes the concepts central to the RISC design philosophy. It is not surprising then, that the increased reliance on compiler solutions, utilized by Silicon RISC designers, is transferred to GaAs processor systems as well. In fact, the characteristics of GaAs dictate that an even increased reliance on compiler solutions be utilized.

Silicon RISC designers have demonstrated the superior performance of RISC computers over Silicon CISCs [PatPi82][HeJoG82]. Much of the credit for the improved performance of RISCs is given to simplified instruction sets which allow the rapid execution of the most frequently used instructions. However, the increased role of compiler technology also plays a large part in the success of RISC processors.

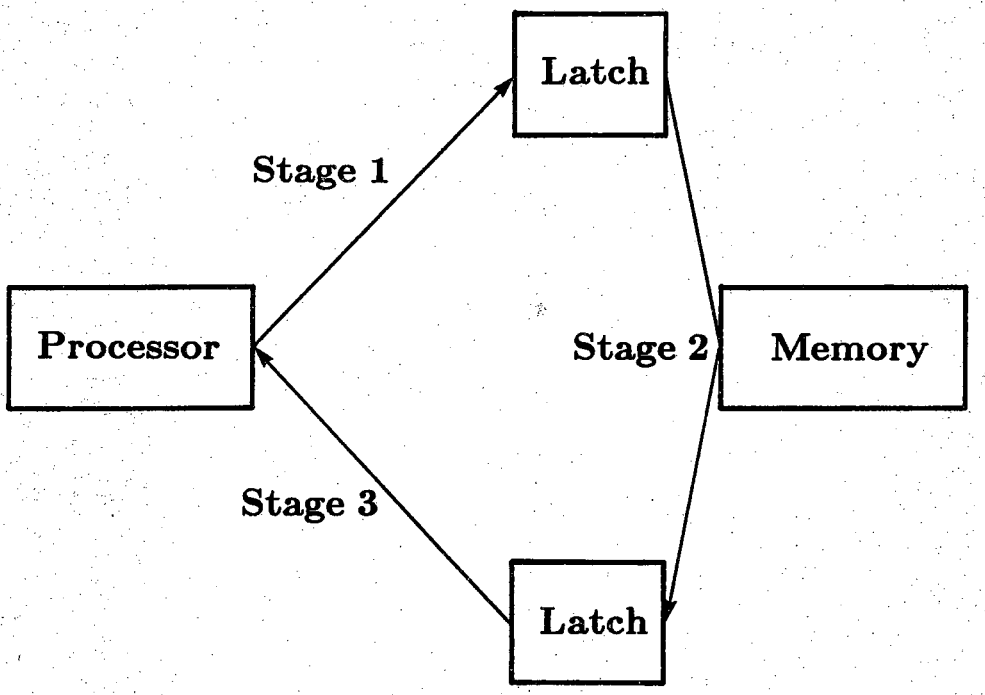


Figure 3.10 Example Pipelined Memory System.

In order to minimize the instruction cycle time, RISC designers attempt to eliminate hardware complexity. One technique used to achieve this is the transfer of functionality from hardware to the compiler. There are several examples of this in Silicon RISCs. Interlock hardware for sequencing hazards [Gross83] was eliminated on the IBM 801 [Radin83], Berkeley RISC-II [Katev83], and Stanford MIPS [HeJoG82]. This introduced a compiler optimization called "branch delay fillin," a technique commonly used by microcode programmers. Interlock hardware for timing hazards [Gross83] was also eliminated on the Stanford MIPS in order to reduce hardware complexity, with timing hazard detection and avoidance instead performed by the MIPS compiler. Because transistor count limitations will be greater for GaAs processors than for Silicon processors, the transfer of even more functionality to the compiler may be desirable for GaAs processors.

RISC instructions are comparable to the microinstructions of a CISC processor. However, a CISC compiler only has access to predefined (by the processor architect) microinstruction sequences in the form of macroinstructions. A RISC compiler, on the otherhand, has access to microinstruction-like RISC instructions and, therefore, has a much greater quantity of instructions to use for both hardware-independent and hardware-dependent optimizations. An increased number of hardware-independent optimizations such as code motions and common subexpression eliminations, therefore, present themselves to a RISC compiler [Radin83]. A hardware-dependent optimization called "load delay fillin," which is not available on most CISCs, presents itself to RISC compilers as well [Radin83]. In a memory-to-memory or memory-to-register CISC instruction, a data memory read must precede the operation execution. If the read requires a large amount of time, then so will the complete execution of the entire instruction. Because RISCs generally use register-to-register and explicit data load instructions, the compiler can schedule the data load instruction in advance, and then "fill in" the data load latency with other useful instructions. Because GaAs processors can be expected to have longer data load latencies (in terms of instruction cycles), the burden on the compiler to find candidate instructions for the fillin gap is increased.

A third area where Silicon RISCs place increased reliance on compiler technology is the result of their decreased instruction cycle times. Because Silicon RISCs have such short cycle times, they are more negatively affected by off-chip delays than Silicon CISCs. As a result, the compiler for the IBM 801

incorporated a sophisticated register coloring scheme in order to reduce that processor's need for off-chip information [AusHo82]. Also, the IBM 801 instruction set included instructions to allow the compiler to override the hardware caching mechanism in some instances when the compiler detected a better strategy for reducing the cache miss ratio [Radin83]. The Stanford MIPS incorporated an instruction packing scheme, and required the compiler to perform the packing [HeJoG82]. This approach allows two operations to be executed during the time required for one fetch, and also reduced the program size, both very beneficial characteristics for a processor in an environment with high penalties for off-chip communication. Because GaAs processors will have even shorter instruction cycle times than Silicon RISCs, GaAs processors will benefit even more from these types of compiler optimizations.

3.3.1 Compiler Optimizations in Control

The two techniques for reducing hardware complexity, which were just briefly listed, are examples of the migration of control hardware into the compiler. These are both described more fully here.

3.3.1.1 Sequencing Hazard Interlocks

Sequencing hazards are caused by branch instructions on a pipelined processor. The problem arises because before the execution of a branch instruction is complete, and hence, before the decision to jump can be established, successive instructions have already been fetched. In general, the execution of these successive instructions may lead to incorrect results. Silicon CISCs and Silicon RISCs usually handle this problem in two different ways.

Silicon CISCs usually employ hardware which halts the execution of the instructions immediately following the branch instruction in the event that the branch is to be taken. This results in a delay in execution until the pipeline can be refilled with the instructions at the destination of the branch. Some CISCs rely on the compiler or run-time algorithms to predict the outcome of the branch condition and fetch the instructions down the appropriate path. However, a wrong guess again requires the emptying of the pipeline. An even more ambitious, and hardware-consuming, CISC solution involves fetching and decoding instructions down both paths. This can then be expanded to three paths, etc.

Silicon RISCs use a technique called "delayed branching" to solve the sequencing hazard problem. Most RISCs always execute the instruction following the branch instruction, thus, there is no need for special hardware to halt instruction execution. However, in order to ensure correct program execution, only a subset of all possible instructions are eligible for placement in the "fillin slots" after branch instructions. If eligible instructions cannot be found, then the compiler must insert NOOPs into the fillin slots.

Since branches are typically 25 percent of all instructions in compiled HLL programs [Katev83], their negative effect on performance can be costly. The delayed branching method for sequencing hazard resolution, in addition to promising simpler hardware, offers potentially higher performance as well. Whenever the RISC compiler is able to successfully move a useful instruction into the fillin slot, the delayed branching method exhibits no branching overhead. However, when the RISC compiler is not able to fill the slot, an instruction cycle is lost whether the branch is taken or not. The CISC approach, with sequencing hazard resolution hardware, loses an instruction cycle whenever branches are taken, but loses nothing when sequential operation is maintained.

The performance of the delayed branching scheme, then depends solely on the RISC compiler. The Stanford MIPS compiler was able to fill approximately 90 percent of the branch fillin slots [HeJoP83], so an instruction cycle was lost on only 10 percent of the branch instruction executions. The CISC approach, with its dependency on dynamic branching probability, doesn't do so well. Since approximately 75 percent of all branch instructions change the program flow [Smith81], an instruction cycle is lost on 75 percent of all branch executions.

However, branch instructions are potentially more costly for the delayed branching scheme in GaAs processor systems than in Silicon systems. As discussed earlier, pipelined memory systems are very advantageous in GaAs processor systems because of their ability to decrease effective memory access delays. However, as evident in Figure 3.3, pipelined memory systems increase the total pipeline length and, consequently, increase the number of fillin slots following branches. For the pipeline in Figure 3.3, the branch delay contains three slots instead of one. Figures 3.11-3.14 show an example program sequence to demonstrate the branch delay fillin optimization on both a Silicon and a GaAs processor. In Figure 3.11, an unoptimized program sequence is shown for a Silicon processor with a branch delay of one. In Figure 3.12, the sequence is shown after the successful fillin of the single fillin slot. Figure 3.13

```
add   a,10          'a takes a plus 10'
add   b,a           'b takes b plus a'
add   c,1           'c takes c plus 1'
bgt   c,0           'if c greater than 0 jump'
NOOP
```

Figure 3.11 Example Program Sequence on a Silicon Processor Before Branch Fill.

```
add   a,10          'a takes a plus 10'
---
add   c,1           'c takes c plus 1'
bgt   c,0           'if c greater than 0 jump'
add   b,a           'b takes b plus a'
```

Figure 3.12 Example Program Sequence on a Silicon Processor After Branch Fill.

```
add   a,10          'a takes a plus 10'
add   b,a           'b takes b plus a'
add   c,1           'c takes c plus 1'
bgt   c,0           'if c greater than 0 jump'
NOOP
NOOP
NOOP
```

Figure 3.13 Example Program Sequence on a GaAs Processor Before Branch Fill.

```
---
---
add   c,1           'c takes c plus 1'
bgt   c,0           'if c greater than 0 jump'
add   a,10          'a takes a plus 10'
add   b,a           'b takes b plus a'
NOOP
```

Figure 3.14 Example Program Sequence on a GaAs Processor After Branch Fill.

shows the same program sequence in unoptimized form with a branch delay of three. Figure 3.14 shows the sequence after optimization in which two of the slots were successfully filled. The third instruction cannot be moved because its completion is required before the execution of the branch instruction. Therefore, the compiler must search outside this code sequence in order to find a third fillin candidate. This example demonstrates both the instruction interdependencies which limit instruction reorganization and the need for more sophisticated branch fillin algorithms for GaAs processors. A significant increase in compiler capability is required in order to successfully fill the larger number of slots. Although the Stanford MIPS compiler was able to fill one branch fillin slot 90 percent of the time, its fill success on the second and third slots was 43 percent and 39 percent, respectively [HeJoP83]. Advances in compiler technology resulting in high fillin probabilities for larger branch delays will result in much higher performance for GaAs processor systems.

3.3.1.2 Timing Hazard Interlocks

Timing hazards generally arise in pipelined processors when multiple pipeline stages have potential access to datapath resources at the same time. Three types of timing hazards have been identified [Gross83]. These are called destination-source conflicts, source-destination conflicts, and destination-destination conflicts.

An example of a destination-source conflict is when a pipestage attempts to read from a hardware resource (i.e. register) before a previous pipestage has finished writing to the resource. Destination-source conflicts occur naturally in register-to-register architectures whenever an instruction writes to a register which is a source register for the succeeding instruction.

An example of a source-destination conflict is when a pipestage attempts to write to a hardware resource at the same time that a previous pipestage is reading from the resource. Another source-destination conflict occurs when a pipestage writes to a hardware resource before a pipestage of a preceding instruction is to read it. The second type of source-destination conflict occurs when a pipestage reads a resource which is previously written by a pipestage in a succeeding instruction. If an exception of some kind occurs, the succeeding instruction may not be executed before the pipestage of the preceding instruction reads the resource.

An example of a destination-destination conflict is when two pipestages attempt to write to a resource concurrently. This may result when the value of

a data load is to be written to a register at the same time that the result of an ALU operation is to be written to the same register.

CISC processors typically utilize hardware to prevent incorrect execution due to timing hazards. An example technique is the "scoreboard" used in the CDC 6600 [Thort64]. Some RISC processors such as the Berkeley RISC-II use a hardware technique called "internal forwarding" to resolve destination-source conflicts.

The Stanford MIPS, on the otherhand, relies entirely on software to resolve timing hazards. The MIPS compiler is tasked with reorganizing instructions so that all conflicts are removed. If the compiler cannot find a suitable candidate instruction to prevent a conflict, it must insert a NOOP. In Figure 3.15 is shown an example code sequence with a destination-source conflict, since register a is both the destination of the first instruction and the source of the second instruction. In Figure 3.16 is the default action of a compiler to resolve the conflict, and Figure 3.17 shows a successful reorganization to eliminate the NOOP. It has been reported that the MIPS instruction cycle would have been lengthened by 10 percent if hardware interlocks were used [Patte85].

Because of its reduction in hardware requirements, software interlocking may be desirable for a GaAs processor. However, this approach again places a great burden on the compiler in order to minimize the number of NOOPs inserted into the program.

3.3.2 Compiler Optimizations in Memory

The low transistor count of GaAs memory chips and the high performance penalty of inter-chip communication severely hinder the memory system in its attempt to maintain an adequate supply of instructions and data for a GaAs processor. Fortunately, the compiler for such a processor has the potential to greatly increase the efficiency of the hardware resources which implement the memory system.

A compiler can provide memory system support in two ways. First, it can increase the reusability of information, i.e., it can increase the length of time that useful information is kept in the higher levels of the memory hierarchy. Second, it can overlap the transfer of information into the higher hierarchical levels with the execution of useful instructions through information prefetching. These two techniques are each discussed here for two types of memory.

```
add   a,10          'a takes a plus 10'
add   b,a           'b takes b plus a'
add   c,d           'c takes c plus d'
```

Figure 3.15 Example Program Sequence Showing a Destination-Source Conflict.

```
add   a,10          'a takes a plus 10'
NOOP
add   b,a           'b takes b plus a'
add   c,d           'c takes c plus d'
```

Figure 3.16 Example Program Sequence Showing Default Compiler Action for Destination-Source Conflict.
Branch Fill.

```
add   a,10        'a takes a plus 10'
add   c,d         'c takes c plus d'
add   b,a         'b takes b plus a'
```

Figure 3.17 Example Program Sequence Showing a Successful Reorganization of Destination-Source Conflict.

(1) Memory which is normally controlled by the compiler, i.e., the register file.

(2) Memory which is normally controlled by run-time mechanisms, i.e., the cache and main memory.

3.3.2.1 Register File Compiler Optimizations

Register file usage is normally directly controlled by the compiler, and, in fact, registers have absorbed considerable abuse for the very fact that they require this compiler control. Because of the long history of register usage, compiler designers have developed fairly advanced techniques for utilizing registers efficiently. The RISC designers were instrumental in advancing compiler technology in this area in order to exploit their faster execution cycle times. GaAs processors, because of even faster instruction execution, will experience a corresponding benefit from improved register file compiler optimizations.

3.3.2.1.1 Reusability

In a GaAs processor which executes register-to-register instructions, and only accesses off-chip data via explicit data load and data store instructions, data loads and stores are extremely costly. The primary reason for this is the much longer access delay for off-chip memory. A secondary cost is the increase in total program size caused by the presence of data loads and stores. Larger program sizes can be expected to decrease hit ratios at all levels of the instruction memory system.

In typical compiled HLL programs on RISC machines, data loads and stores are approximately 30 percent of all executed instructions [Patte85]. In the Silicon environment, larger register files may be incorporated in order to keep more useful data on-chip. As indicated earlier, the designers of the Berkeley RISC-II used 138 registers, divided into eight windows, to reduce their frequency of data loads and stores to approximately 15 percent [Patte85]. Clearly, this solution is not applicable for a transistor-scarce GaAs processor. The multiple window schemes for GaAs disussed earlier, although potentially good approaches, do add hardware complexity to a GaAs processor.

The designers of the IBM 801 relied very heavily on the capabilities of compiler technology. Their PL.8 compiler incorporated a highly sophisticated register allocation scheme to reduce the frequency of loads and stores [AusHo82].

Figures 3.18 and 3.19 show the improvement that can be gained from an intelligent compiler. Figure 3.18 shows an example unoptimized code sequence. Figure 3.19 shows the same sequence, but produced by a compiler with a good register allocation scheme. Clearly, a compiler-based approach to reduce the frequency of data loads and stores has an inherent advantage over hardware approaches in the GaAs environment if it can achieve adequate results.

3.3.2.1.2 Prefetching

As indicated previously, RISC processor designs allow a compiler optimization not allowed on typical CISCs. This optimization, "data load fillin," reduces the negative effect of data load latencies on performance. Because GaAs processors can be expected to have longer off-chip data load latencies than Silicon processors, this optimization can have a greater positive impact on performance for a GaAs processor.

In a GaAs processor system, the data memory may execute data loads and stores in parallel with processor execution. In principle, for all data memory accesses, the processor need only initiate the access, and also receive the data value resulting from a load. If the compiler is able to schedule enough useful instructions after the data load initiation, the data load latency is effectively eliminated.

The compiler then is tasked with scheduling each data load instruction so that the data load result is in the processor before or at the time the processor requires it. This optimization is similar to the branch fillin problem described earlier in that only a subset of the possible instructions can be used to perform the load fillin.

This optimization is shown in Figures 3.20-3.23 for a Silicon pipeline represented by Figure 3.1 and a GaAs pipeline represented by Figure 3.3. In Figure 3.20 is shown an unoptimized program sequence for the Silicon pipeline, while Figure 3.21 shows the optimized version. Figure 3.22 shows the same unoptimized program sequence for the GaAs pipeline, and Figure 3.23 shows its optimized form. As in the case of branch filling, data load filling success is limited by data dependencies, and more sophisticated optimization strategies are required to approach 100 percent fillin on a GaAs processor. Because data loads are so frequent, the implementation of improved compiler technologies can improve GaAs processor system performance significantly.

```
store   a,TEMP          'store a into TEMP'
add     b,c             'b takes b plus c'
load    a,TEMP          'load from TEMP to a'
add     b,d             'b takes b plus d'
add     d,a             'd takes d plus a'
```

Figure 3.18 Example Program Sequence Showing Poor Register Allocation.

```
---
add     b,c             'b takes b plus c'
---
add     b,d             'b takes b plus d'
add     d,a             'd takes d plus a'
```

Figure 3.19 Example Program Sequence Showing Good Register Allocation.

```
add   a,b          'a takes a plus b'
add   a,1          'a takes a plus 1'
add   c,b          'c takes c plus b'
load  d,A[c]       'load A[c] into d'
NOOP
add   e,d          'e takes e plus d'
```

Figure 3.20 Example Program Sequence on a Silicon Processor Before Load Fillin.

```
add   a,b          'a takes a plus b'
---
add   c,b          'c takes c plus b'
load  d,A[c]       'load A[c] into d'
add   a,1          'a takes a plus 1'
add   e,d          'e takes e plus d'
```

Figure 3.21 Example Program Sequence on a Silicon Processor After Load Fillin.

```
add   a,b              'a takes a plus b'
add   a,1              'a takes a plus 1'
add   c,b              'c takes c plus b'
load  d,A[c]           'load A[c] into d'
NOOP
NOOP
NOOP
add   e,d              'e takes e plus d'
```

Figure 3.22 Example Program Sequence on a GaAs Processor Before Load Fillin.

```
---
---
add   c,b              'c takes c plus b'
load  d,A[c]           'load A[c] into d'
add   a,b              'a takes a plus b'
add   a,1              'a takes a plus 1'
NOOP
add   e,d              'e takes e plus d'
```

Figure 3.23 Example Program Sequence on a GaAs Processor After Load Fillin.

3.3.2.2 Cache / Main Memory Compiler Optimizations

The contents of the LRU-based memory levels of the memory hierarchy (i.e. cache, main memory, etc.), are normally determined by run-time hardware. However, as already indicated, in many instances in which the temporal and spatial localities of reference, upon which LRU mechanisms are based, fails, the compiler is able to provide assistance. Because a similar delay due to memory misses results in greater wasted instruction cycles for GaAs processors than for Silicon processors, compiler optimizations to improve hit ratios gain added importance in the GaAs environment.

3.3.2.2.1 Reusability

This section describes two ways in which the compiler is able to increase the useful time of cache blocks, main memory pages, etc. The first technique is to increase the temporal and spatial localities of reference, and the second technique involves providing support which is used at run-time to reduce the negative consequences of poor locality.

Increasing referencing locality is more effective for large memory units such as main memory pages or segments, but also may be used with cache blocks as well. What is desired is an increased correlation between high temporal locality and high spatial locality for particular information units. In other words, the information which is used within nearby time periods should also be stored in nearby memory locations. If the compiler (and linker) knows the page size, etc., then information which exhibits high temporal locality, as determined by the compiler, can be allocated memory locations within the same page, etc. Even without page size information, more spatial locality by itself will decrease miss ratios. A study was performed on a compiler algorithm to increase the spatial locality of data having large temporal locality [AbKuL81]. In this study, miss ratios for the unmodified programs were as much as 20 times higher than the miss ratios for the modified programs. Clearly, this type of compiler optimization will have an enormous impact on the performance of a GaAs processor system.

As mentioned earlier, one instance where temporal locality is not present is the use of some particular information followed by a large period of non-use. Bringing this type of information into a higher level of the memory hierarchy will decrease that level's hit ratio, because it results in more useful information being replaced. If the compiler detects that an information access will displace information of higher future usefulness, then it may override the run-time

information control mechanism. One possible method for accomplishing this is the use of special data load and store instructions which inhibit the memory system's run-time mechanism. This technique is used on the IBM 801 [Radin83]. A special data store instruction is provided which signals the cache to not perform block replacement. As with many compiler optimizations designed for Silicon RISCs, this technique is potentially even more profitable for a GaAs processor.

3.3.2.2.2 Prefetching

As mentioned earlier, temporal locality is not present in instances where information is accessed for the first time after a long period of non-use. These instances occur when the present referencing locality is exited and a new locality established. These inter-locality gaps are disastrous for LRU-based run-time mechanisms. It is possible for the compiler to detect these inter-locality gaps and to assist the run-time mechanism in preparing for them. This technique is more useful for the cache than for the main memory in systems with magnetic disks. Since disk accesses normally require milliseconds, page faults are usually handled by switching to another waiting task. Unless the prefetching scheme can detect the prefetch candidate milliseconds before need, nothing is gained by beginning the disk transfer early.

When the compiler detects a prefetching candidate, it requires a method to initiate the prefetch. One technique is to use special memory prefetch instructions, which are executed by the processor as special memory instructions. These special instructions may be handled by the memory hardware in much the same way as any memory access. When such an instruction is executed, the processor calculates the memory address and does nothing more. Many of the NOOPs normally executed by the processor because the compiler was unable to successfully perform branch or load fillin may be replaced by these special memory instructions. Once again, this compiler support can increase the performance of a GaAs processor significantly.

CHAPTER IV
PIPELINE AND INSTRUCTION FORMAT EXPERIMENTS

In Chapter II we presented an overview of GaAs technology. We studied the GaAs device families and logic families commonly used in digital designs. We selected the most capable and mature GaAs technology, the DCFL E/D-MESFET family, compared it with Silicon NMOS, and found a number of significant differences. We then enumerated those characteristics of the GaAs DCFL E/D-MESFET family which we believe will significantly influence the design of computer systems using this technology.

In Chapter III we presented approaches to computer system design that we consider to be appropriate for GaAs technology. We first described the manner in which the characteristics of GaAs influence computer system design. We then discussed design approaches for the hardware and compiler of GaAs computer systems.

In this chapter we describe two experiments which extend some of the work presented in Chapter III. We first discuss our evaluation methodology, including our evaluation tools. We then present our first experiment which compares candidate GaAs instruction pipelines, and follow that with an experiment to compare candidate GaAs instruction formats.

4.1 Evaluation Methodology

We utilize simulation as our primary evaluation technique. We choose this approach because of its advantages over other evaluation methodologies, and because the appropriate tools are readily available to us. Analytical models are also sometimes used, but these are generally not as representative as simulations. We don't like hardware prototyping for many reasons including cost, lack of flexibility for design modifications, and, of course, the fact that no GaAs systems of the type we wish to model have ever been built before.

Our primary evaluation criterion is the time required to execute compiled HLL programs. In order to obtain HLL program execution times, a simulation

system requires three principal components. First, an application environment in the form of HLL programs is necessary. Second, a simulation program which implements the architecture description is needed. Finally, a method of translating the HLL programs to the architecture description is required, and this translation should be optimized to exploit any execution speedup opportunities presented by the architecture.

4.1.1 Workload Model

Because computer system performance depends heavily on the characteristics of the programs it executes, the selection of an appropriate application environment is of considerable importance. We can observe the effect of different application environments in the design of commercial Silicon systems such as the Cray-1 [Russe78] for highly arithmetic environments with regular data structures, the TMS320 [MaCaM82] for signal processing applications, and the MC68020 [MaMoM84] for general purpose applications.

Our application environment consists of a broad mixture of programs written in the high level language PASCAL. These programs vary considerably in their use of iteration, recursion, arithmetic, and data structures. Considered collectively, they represent a general purpose programming environment; while the characteristics of selected individual programs may be used to enhance the responsiveness of execution time to particular architectural variations.

The ten PASCAL programs which represent our workload model were obtained from Stanford University through RCA Corporation. Many of these programs are widely used for benchmarking purposes and appear frequently in the literature.

(1) Ack - a highly recursive program to compute Ackermann's function.

(2) Bubble - a program to perform a bubble sort of 500 integers.

(3) Fib - a highly recursive program to compute a Fibonacci number.

(4) Intmm - a computation-heavy program to multiply two 40x40-element integer arrays.

(5) Perm - a highly recursive program to calculate all permutations of the numbers 1 through 7.

(6) Puzzle - an iteration-heavy, computation-heavy program to solve a 3 dimensional cube packing problem.

(7) Queen - a program to solve the Eight Queens problem.

(8) Quick - a program to perform a quick sort of 5000 integers.

(9) Sieve - a program which calculates the number of primes between 0 and 8190.

(10) Towers - a highly recursive program to solve the Towers of Hanoi problem with 18 discs.

4.1.2 Architecture Model

Our architecture model is a simulation program written in the high level language C for the Stanford MIPS processor. The MIPS simulator was made available to us by RCA. It performs simulation at the instruction level; therefore, it requires MIPS instructions for its input. Its output is the program execution time, in terms of the number of instructions executed.

The MIPS architecture is very appropriate for this study because its transistor count is compatible with GaAs E/D-MESFET capabilities of the near future. It is helpful to revisit several MIPS characteristics in order to provide a better understanding of the following two experiments.

First, MIPS employs delayed branching with a branch delay of one*. Again, this means that the first instruction after every branch operation is always executed, and the compiler is responsible for finding instructions for the fillin slots such that correct program execution is maintained. If the compiler cannot find a useful instruction for a fillin slot, it must then insert a NOOP into the slot.

Second, the MIPS compiler must perform an analogous function for the first instruction after data load operations.

Finally, the MIPS processor employs instruction packing. A MIPS instruction may contain two operations which are executed sequentially in the time necessary to perform one instruction fetch. Not all operation combinations may be packed, however, and instructions may therefore contain either one or two operations.

---

* The Stanford MIPS literature states that the "indirect branch instruction" (see Appendix B) has a branch delay of two. However, we consider this instruction to be a pair of *operations* - a data load followed by a branch. We are carefully using *instruction* to refer to those atomic entities which are fetched into the processor, and we use *operation* whenever an instruction contains multiple executable pieces. This distinction is important when discussing packed instructions containing multiple operations, in addition to the issue of indirect branch instructions.

In addition to the Stanford software, we will use a cache simulator which was designed and implemented at Purdue University. The cache simulator is implemented as a procedure which is callable from within the MIPS simulator. It receives addresses and possibly data and returns the number of instruction cycles required for access, while updating the data and tag information. Modifications to the cache size, block size, prefetch strategy, cache hit delay, and cache miss delay allow for a flexible cache implementation.

4.1.3 Workload to Architecture Translation

There must be some way to translate the PASCAL benchmark programs into a form acceptable to the architecture simulator. The software package that we use for this translation was written by Stanford and again provided to us by RCA. The package consists of a PASCAL compiler, optimizer, code generator, assembler and reorganizer, linker, and loader.

The compiler transforms PASCAL programs into an intermediate language, which then receives hardware-independent optimizations before being converted into MIPS-like instructions. The assembler and reorganizer perform the branch delay fillin, load delay fillin, instruction packing, etc., and convert the MIPS assembler instructions into machine code. The linker combines this code with the run-time library containing multiplication routines, input-output routines, etc., and the loader logically stores the linked program into memory locations between 0 and 31,999. This loaded program is then written to a file where it is kept until required by the architecture simulation program.

4.2 Pipeline Experiment

We now describe an experiment undertaken in order to evaluate three instruction pipelines in a GaAs processor environment. We begin by explaining our motivation for performing this experiment, and then we present our choice of pipelines for examination. We describe our evaluation criterion and present some background to illustrate why optimal pipeline performance is usually not achieved. We then discuss our methods for quantifying these causes of suboptimal performance, and we incorporate them into our analytical pipeline performance model description. We present our experimental results and finish with a discussion of these results.

4.2.1 Rationale

Section 3.2.3 presented the problems associated with the use of Silicon pipelines in a GaAs processor. Again, the disadvantage of such an approach is that the datapath of a GaAs processor will have a severe underutilization permanently built into it. Three approaches to alleviate this problem were advocated, and they are represented by Figures 3.3 - 3.5. Figure 3.3 shows an instruction pipeline which results from the use of a pipelined instruction memory; Figure 3.4 shows an instruction pipeline which incorporates instruction packing; while Figure 3.5 shows an instruction pipeline resulting from the use of slow datapath elements. We believe that it is very desirable to determine the relative performance capabilities of these candidate instruction pipelines in a GaAs processor environment.

4.2.2 Candidate Pipeline Descriptions

Because of the nature of our simulation tools, we find it relatively straightforward to study two of the above three candidate instruction pipelines.

Through modifications to the MIPS simulator and analytical techniques described later, it is relatively easy to model the performance of the instruction pipeline with a pipelined instruction memory. This instruction pipeline will be called the "pipelined memory" pipeline from now on.

The second instruction pipeline that we model is the pipeline with instruction packing. We use the packed MIPS format to represent this pipeline, which we will call the "packed" pipeline.

The pipeline with slower datapath elements is considerably harder to model because a large number of variables are involved. This pipeline advocates the removal of hardware resources from datapath elements used to perform simple operations, and the reallocation of them elsewhere. The freed hardware resources may be used to implement an on-chip serial multiplier, larger number of registers, etc. However, there are several difficulties in the analysis of this pipeline. First, datapath elements require redesign and relayout in order to determine both the additional transistors and area available for other units. The two hardware candidates which we've identified to receive these freed resources are a serial multiplier or larger number of registers. The serial multiplier requires considerable effort for its design and incorporation into the processor. This is in addition to the effort needed to model its presence in such a processor. An increase in register file size cannot be effectively modeled because the MIPS compiler which we are using does not produce correct code

when register values must be spilled [Linn85]. A set of benchmarks necessary to test the performance benefit of the additional registers must contain procedures with many variables; however, these benchmarks cannot be compiled by our compiler. Because of these difficulties, we will not study the instruction pipeline with slow datapath elements.

We do include a third pipeline in this study however. In order to discover the performance advantage, if any, of the two candidate GaAs pipelines over a Silicon pipeline, we choose an example Silicon pipeline for evaluation. The pipeline that we use is for an unpacked version of the MIPS instruction set. We call this pipeline the "normal Silicon" pipeline.

4.2.3 Evaluation Criterion

For this study we choose as our evaluation criterion the number of useful (i.e. non-NOOP) operations executed per datapath cycle. We are not concerned with speeding execution by minimizing the datapath time, nor are we allowed to arbitrarily lengthen the datapath time in order to inflate our performance metric. As we will soon observe, the important constraint which we will obey is the ratio of instruction fetch delay to datapath delay.

We define an "ideal" instruction pipeline to be one which yields a useful operation execution on every datapath cycle. In order to achieve this ideal operation execution rate, the pipeline must exploit the parallelism typically present in modern single-processor computer systems. Example parallel resources include the instruction memory, instruction fetch logic, instruction decode logic, instruction execution logic (datapath), and data memory.

4.2.4 Causes of Non-ideal Performance

In reality one-operation-per-cycle execution is not achieved. Memory system deficiencies and disruptive programming constructs both reduce the performance of a computer system. We can observe three types of disruptions which degrade the performance of our pipelines. In all three cases, the magnitude of performance degradation is highly dependent upon the capabilities of both the memory system and compiler.

(1) Branches. The form of parallelism exploitation utilized by pipelining is severely undermined by program branches, as described in Section 3.3.1.1. We find it helpful to present again the manner in which program branches degrade the performance of pipelined processors which utilize delayed

branching. In the simple example Silicon pipeline of Figure 3.1, we observe the concurrent usage of the instruction fetch logic / instruction memory and the instruction execution logic. Note, however, that while the execution logic executes an instruction, instruction i for example, the instruction fetch logic is fetching instruction i+1. This is the essence of the pipelined method of parallelism exploitation. However, if the execution of instruction i causes a program branch, the work (fetch of i+1) performed by the instruction fetch logic is for naught unless the compiler can guarantee that instruction i+1 is useful. In the GaAs pipelined memory pipeline of Figure 3.3, the compiler must ensure that three useful instructions follow branch operations in order to prevent a variation from ideal operation execution.

(2) Instruction Fetches. An instruction memory which exhibits a longer fetch delay than the instruction execution logic (datapath) delay must have parallelism introduced into it, and this parallelism must be fully exploited or else performance degradation will result. An example pipeline for an instruction memory exhibiting no parallelism and a long fetch delay is shown in Figure 3.2. This pipeline is unacceptable if high performance is a design goal. Pipelines resulting from parallel instruction memory systems are shown in Figures 3.3 and 3.4. The pipeline of Figure 3.3 results from a pipelined instruction memory consisting of three stages. These "parallel" stages require compiler assistance to be fully exploited in the presence of branch operations as discussed above. The pipeline of Figure 3.4 results from an instruction memory which provides three operations in parallel. In general, the restricted number of pins on the processor chip will limit the size of instruction transfers, while the large size (number of bits) required by some operations will limit the number of operations which may be concurrently fetched. Again, compiler assistance may be needed to maximize the packing of operations into instructions. One final cause of non-ideal operation execution is the variation in fetch times between the levels of the memory hierarchy. Generally, the parallelism just discussed is only introduced into the highest level of the memory hierarchy. If the instruction requested by the processor is not in the highest memory level, then a delay is introduced. Therefore, the "hit ratio" of the highest level of the instruction memory hierarchy is also important.

(3) Data Loads. A data memory which has a longer access delay than the datapath delay must depend on parallelism exploitation in order to avoid degrading the processor's performance. Typical memory hardware

methods for introducing parallelism include interleaved memory and pipelined memory. Interleaved memories allow multiple words to be concurrently accessed much as multiple operations are accessed in packed instruction fetches. However, because of the lack of redundancy in most data, the pin count limitations of a single-chip processor reduce the applicability of this approach. Pipelined memories are useful and easily exploitable for data stores; and also useful for sequences of data loads. In general, though, the compiler is most capable of exploiting the parallel datapath and data memory units by successfully performing load fillin. Again, the effect of cache misses may also cause performance degradation.

4.2.5 Modeling Memory and Compiler Effects

As demonstrated in the last section, both the off-chip memory system and the compiler have large influences on the performance of our GaAs pipelines. Because there is a lack of experimental data for the memory system parameters which we require, we will use a range of values which we consider to be representative of future memory system designs. Also, since we presently know of no compilers which have been designed to exploit the optimization opportunities presented by our candidate pipelines, we use a range of values for our compiler-based parameters.

4.2.5.1 Memory Parameters

As mentioned in Section 3.2.3, a reasonable number for the ratio of instruction fetch delay to datapath delay is three when an off-chip/on-package instruction memory is used. We will, therefore, use this ratio for all on-package accesses throughout the rest of this experiment. For off-chip/off-package accesses we will use six as the ratio of memory access delay to datapath delay. Also, we will assume that the capacity of this off-package memory is infinite, therefore limiting this analysis to a two-level memory hierarchy.

We will use three memory configurations of the four which are possible with our two-level memory hierarchy and separate instruction and data memory. The first configuration, which we will call the "(3,3)" configuration, consists of a two-level instruction memory and a two-level data memory. The ratio of the memory access delay to datapath delay for the fastest level of the hierarchy in both cases is then three. The second configuration, denoted

"(3,6)," contains a two-level instruction memory and a single-level data memory. The ratio for the data memory is always six. The third configuration, the "(6,3)" configuration, contains a single-level instruction memory and two-level data memory. The ratio for the instruction memory is always six in this configuration.

We require six memory parameters, four of which are directly derivable from the pipeline and memory configuration. However, since we will require all six parameters in our analytical performance model, we will define all six here.

(1) "nih" - Number of effective datapath cycles for an instruction cache hit. This parameter is one or three if an on-package instruction cache is used; it is one or six otherwise. Note that the "effective" access delay of a pipelined memory is one.

(2) "nim" - Number of effective datapath cycles for an instruction cache miss. This parameter is four or six if an on-package instruction cache is used; it is six otherwise. Note that an instruction cache miss always results in a delay of three cycles, unless a single-level instruction memory is used.

(3) "pih" - Probability of instruction cache hit. This parameter is not directly derivable from the pipeline and memory configuration. We instead will use a range of values for this parameter. For the default value we observe the empirical cache hit ratios presented in [Smith85] for small cache sizes. Based on these results we select a value of 0.8 as our default value.

(4) "ndh" - Number of datapath cycles for a data cache read hit. This parameter is three if an on-package data cache is used; it is six otherwise.

(5) "ndm" - Number of datapath cycles for a data cache read miss. This parameter is always six.

(6) "pdh" - Probability of data cache read hit. As with pih, this parameter is not derivable from our pipeline and memory configurations, so we will use a range of values. From the results in [Smith85] we select a default value of 0.8 for this parameter as well.

There is an assumption implied here for data memory accesses. The parameters ndh, ndm, and pdh are valid for data memory *reads* only. We assume that the data memory is pipelined such that data memory writes only cost the processor one datapath cycle. A similar assumption for data memory reads would add considerable complexity into this analysis.

4.2.5.2 Compiler Parameters

There are two compiler parameters required for our analysis.

(1) "pbf" - Probability of branch fill. This is the probability that the branch fillin slots immediately following branch operations contain useful instructions. As in the case of both pih and pdh above, we will use a range of values for this parameter, as there is no empirical data, to our knowledge, available for a compiler targeted to an architecture with our large branch delays. We will use 0.6 as our default value because this is the value obtained by the MIPS compiler for a branch delay of three [HeJoP83]. However, because the MIPS compiler was actually targeted to a machine with a branch delay of one, we don't believe their motivation for successfully filling three slots was especially strong; thus, perhaps our default value is low for a branch delay of three. However, since we also utilize branch delays of six in this experiment, we consider our default to be a suitable compromise.

Note that the value for pbf is the probability that *all* of the possible slots are filled. Thus, for a branch delay of three and pbf of 0.6 there are an average of 1.8 useful instructions and 1.2 NOOPs following each branch operation.

(2) "plf" - Probability of load fill. This is the probability that the load fillin slots immediately following load operations contain useful instructions. We will use a range of values for this parameter, and since even less empirical data is available for this parameter than for pbf, we will use the same default value - 0.6.

4.2.6 Modeling the Workload Effects

We now discuss the characteristics of our benchmarks which we require in our analytical performance model. We first define the workload parameters and then we then present them in Table 4.1.

4.2.6.1 Workload Parameter Definitions

Section 4.2.4 discussed three causes of non-ideal instruction pipeline performance. Again, these are derived from branches, instruction fetches, and data loads. In order to determine their total negative effect on program execution time, we need to know the number of branches, instruction fetches,

and data loads present in our benchmark programs. We will use three abbreviations for these three parameters.

(1) "ni" - Number of instructions in the benchmark program under study, or the average number of instructions if the workload contains multiple benchmark programs.

(2) "nl" - Number of load operations in the benchmark program under study, or the average number of load operations if the workload contains multiple benchmark programs.

(3) "nb" - Number of branch operations in the benchmark program under study, or the average number of branch operations if the workload contains multiple benchmark programs.

In addition to these three parameters, the analytical performance model described in the next section requires three additional parameters which describe the effect of the Stanford MIPS compiler on the benchmark programs.

Because two of our candidate instruction pipelines do not use packed instructions, and because our analysis is based upon the MIPS instruction format, which does use packed instructions, we must know the number of packed instructions in the benchmark programs. Again we shall abbreviate this parameter.

(4) "np" - Number of packed instructions in the benchmark program under study, or the average number of packed instructions if the workload contains multiple benchmark programs. Appendix B describes the Stanford MIPS instruction set, and the instructions which are eligible to be packed are evident there. However, just because a MIPS instruction contains space for two operations does not imply that every instance of this instruction is packed. The ALU operation piece of any MIPS may contain a NOOP, which is actually a "MOV Rx,Rx" operation. Also, we consider both the conditional branch instruction and the conditional trap instruction to be packed since two MIPS operations are clearly required for both of these instructions.

In order to model the effect of a compiler for a GaAs processor, the analytical model of the next section must know some additional effects of the MIPS compiler on the benchmark programs.

(5) "pbf0" - Probability of branch fill that the Stanford MIPS compiler achieved on the given benchmark programs. Determining this value is complicated by the instruction packing of the MIPS instruction set. The approach we've taken is to ignore packing in determining pbf0. If a

branch instruction is not packed and could be, this does not affect pbf0. Also, if a fillin slot after a branch contains at least one operation, then that slot is considered to be filled, otherwise it is unfilled.

(6) "plf0" - Probability of load fill that the Stanford MIPS compiler achieved on the given benchmark programs. The above statements concerning packing apply here as well.

4.2.6.2 Workload Parameter Values

Through minor modifications to the Stanford MIPS simulation program we can obtain the above six workload parameter values. The simulator modifications involve the insertion of appropriate counters into the simulation program. Because these modifications are minor in contrast to the large size of the simulation program, we don't list them. Table 4.1 contains the results which we obtained for the ten benchmark programs. The benchmark average at the bottom of the table weights the contribution of each benchmark program equally.

4.2.7 Analytical Pipeline Performance Model

We use our pipeline performance model to evaluate each of our candidate pipelines. As discussed in Section 4.2.3 the ideal pipeline execution rate is one useful operation per datapath cycle. Equivalently, this ideal rate is one datapath cycle per useful operation.

Our performance equations calculate the total number of datapath cycles required to execute the benchmark programs. These values will then be used to help create the plots shown in the next section.

Altogether, we have nine candidate GaAs pipelines. They are the normal Silicon pipeline in the (3,3), (3,6), and (6,3) memory configurations; the packed pipeline in the (3,3), (3,6), and (6,3) memory configurations; and the pipelined memory pipeline in the (3,3), (3,6), and (6,3) memory configurations.

The normal Silicon pipelines are shown in Figures 4.1 - 4.3 and their performance equations are given in Equations 4.1 - 4.3. The packed pipelines are shown in Figures 4.4 - 4.6 and their performance equations are given in Equations 4.4 - 4.6. Finally, the pipelined memory pipelines are shown in

Table 4.1 Workload Characteristics Relevant for Pipeline Study.

Benchmark	ni*	nb*	nl*	np*	pbf0	plf0
ack	1000	216	270	189	0.62	0.90
bubble	1000	217	211	350	0.51	1.00
fib	1000	212	303	212	0.71	1.00
intmm	1000	63	134	637	1.00	1.00
perm	1000	143	304	230	0.75	0.88
puzzle	1000	316	263	489	0.95	0.92
queen	1000	151	299	252	0.63	0.65
quick	1000	208	156	633	0.56	0.97
sieve	1000	249	50	449	1.00	1.00
towers	1000	129	362	203	0.44	0.74
average	1000	190	235	364	0.72	0.91

* Measured per 1000 MIPS instructions.

Figure 4.7 - 4.9 and their performance equations are given in Equations 4.7 - 4.9.

The derivation of the performance equations is rather bulky; therefore we include it in Appendix A instead of here.

4.2.7.1 Normal Silicon (3,3)

$$\begin{aligned}\text{Execution time} &= ni * (6 - 3 * pih) \\ &+ nl * (6 - 3 * pih) * (plf0 - plf) \\ &+ nb * (6 - 3 * pih) * (pbf0 - pbf) \\ &+ np * (6 - 3 * pih)\end{aligned} \tag{4.1}$$

4.2.7.2 Normal Silicon (3,6)

$$\begin{aligned}\text{Execution time} &= ni * (6 - 3 * pih) \\ &+ nl * (6 - 3 * pih) * (plf0 - plf) \\ &+ nb * (6 - 3 * pih) * (pbf0 - pbf) \\ &+ np * (6 - 3 * pih)\end{aligned} \tag{4.2}$$

4.2.7.3 Normal Silicon (6,3)

$$\begin{aligned}\text{Execution time} &= ni * 6 \\ &+ nl * 6 * (plf0 - 1) \\ &+ nb * 6 * (pbf0 - pbf) \\ &+ np * 6\end{aligned} \tag{4.3}$$

4.2.7.4 Packed (3,3)

$$\begin{aligned}\text{Execution time} &= ni * (6 - 3 * pih) \\ &+ nl * (6 - 3 * pih) * (plf0 - plf) \\ &+ nb * (6 - 3 * pih) * (pbf0 - pbf)\end{aligned} \tag{4.4}$$

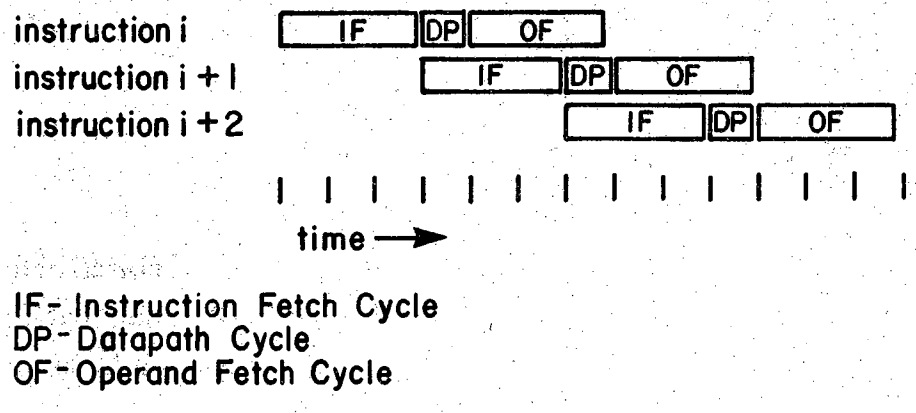


Figure 4.1 Normal Silicon (3,3) Pipeline.

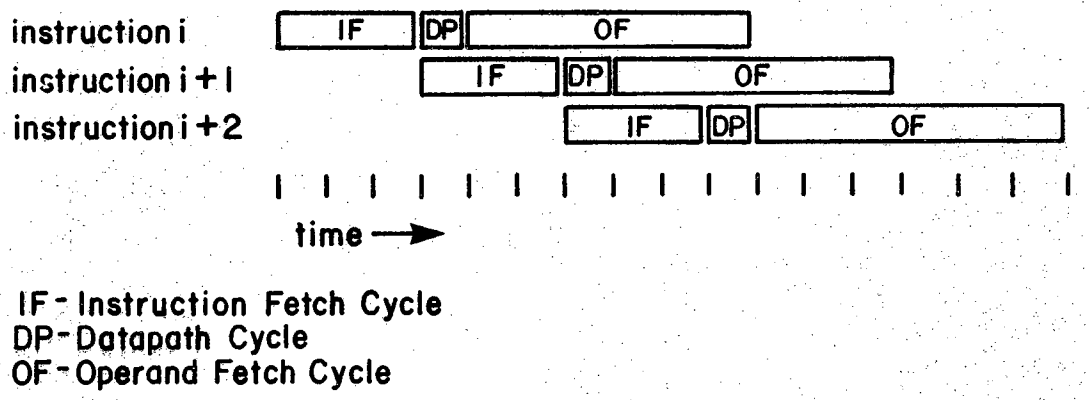


Figure 4.2 Normal Silicon (3,6) Pipeline.

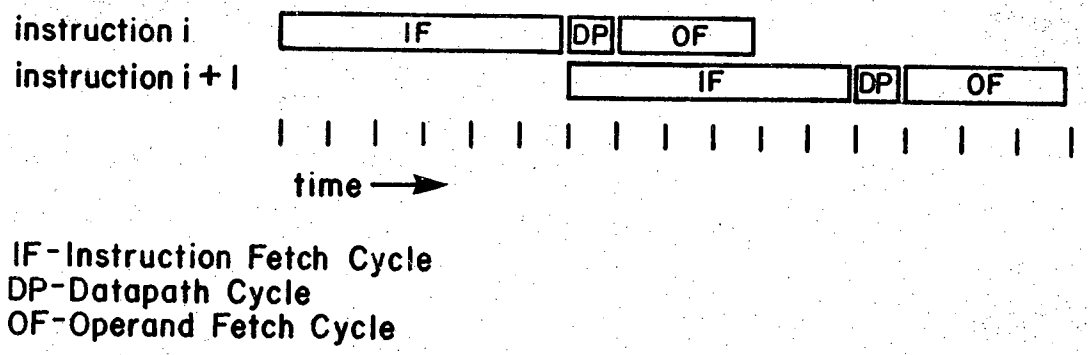


Figure 4.3 Normal Silicon (6,3) Pipeline.

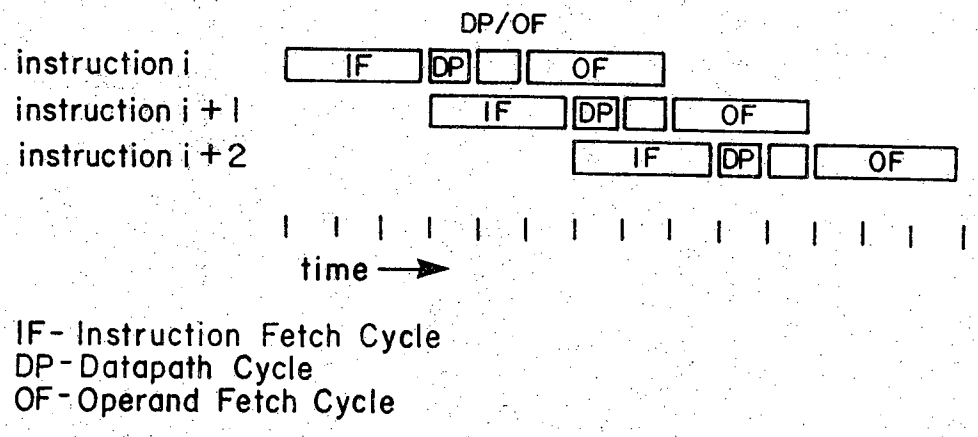


Figure 4.4 Packed (3,3) Pipeline.

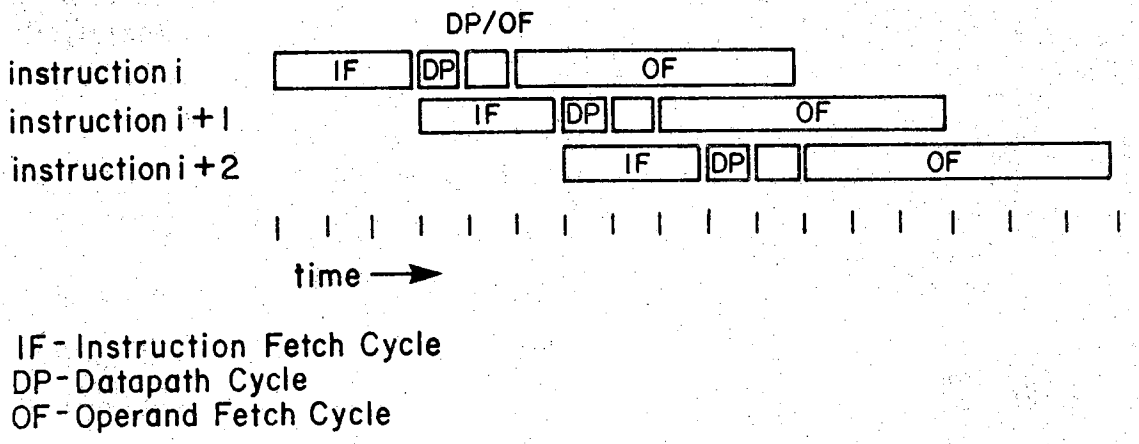


Figure 4.5 Packed (3,6) Pipeline.

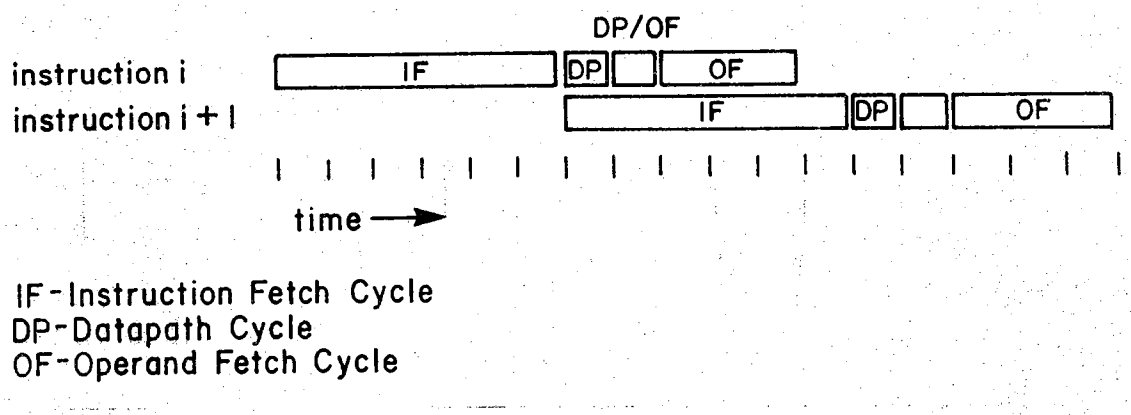


Figure 4.6 Packed (6,3) Pipeline.

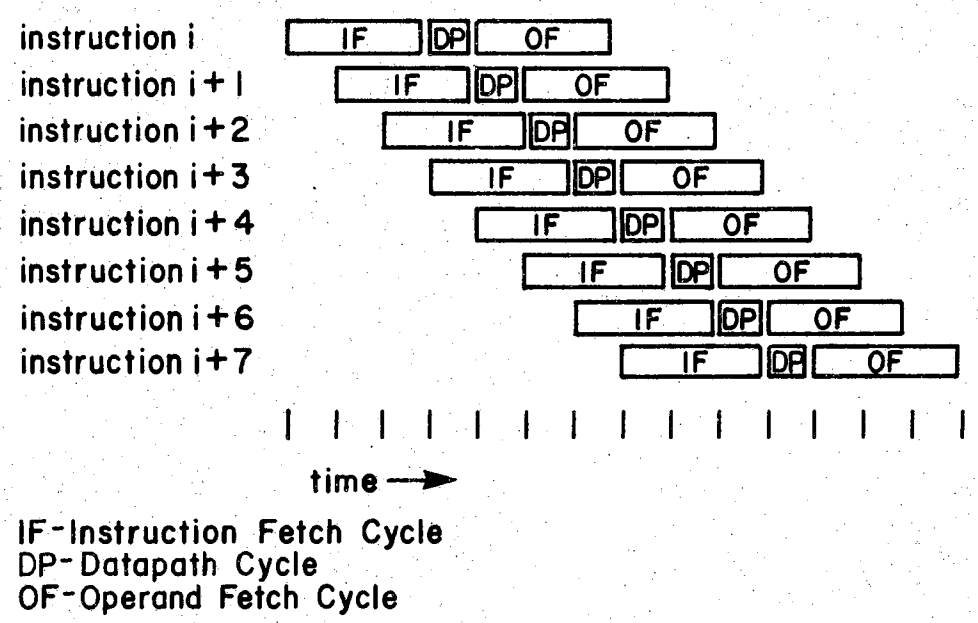


Figure 4.7 Pipelined Memory (3,3) Pipeline.

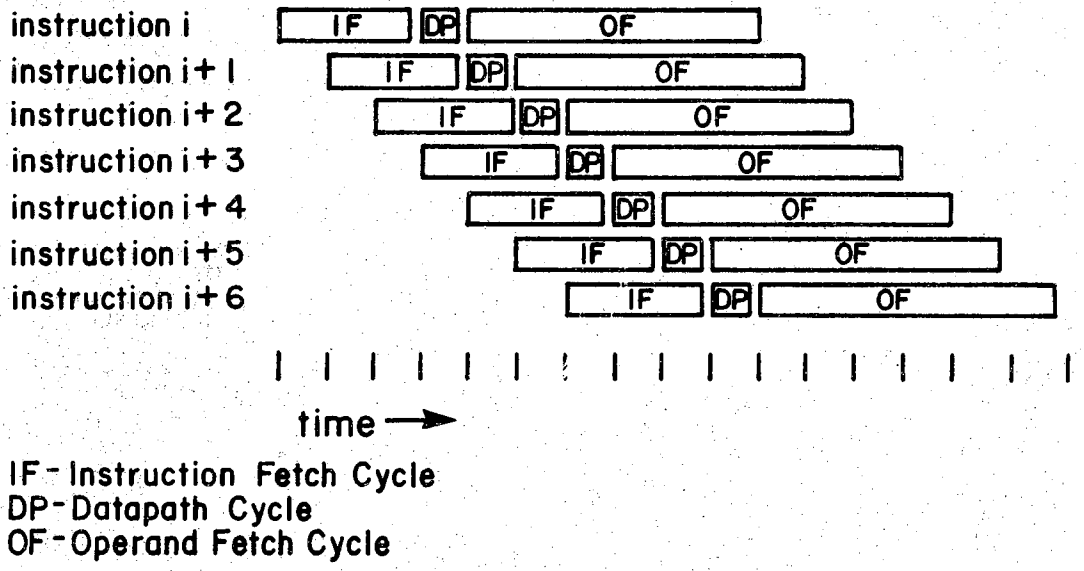


Figure 4.8 Pipelined Memory (3,6) Pipeline.

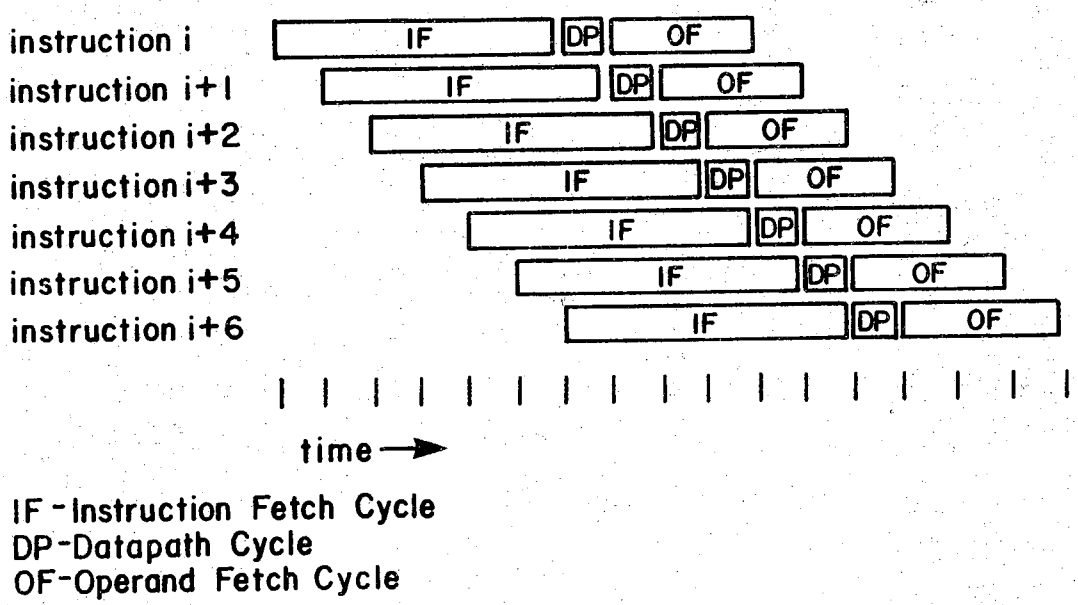


Figure 4.9 Pipelined Memory (6,3) Pipeline.

4.2.7.5 Packed (3,6)

$$\begin{aligned}\text{Execution time} &= ni * (6 - 3 * pih) \\ &+ nl * (6 - 3 * pih) * (plf0 - plf) \\ &+ nb * (6 - 3 * pih) * (pbf0 - pbf)\end{aligned} \tag{4.5}$$

4.2.7.6 Packed (6,3)

$$\begin{aligned}\text{Execution time} &= ni * 6 \\ &+ nl * 6 * (plf0 - 1) \\ &+ nb * 6 * (pbf0 - pbf)\end{aligned} \tag{4.6}$$

4.2.7.7 Pipelined Memory (3,3)

$$\begin{aligned}\text{Execution time} &= ni * (4 - 3 * pih) \\ &+ nl * [3 * (1 - pdh) \\ &\qquad + (4 - 3 * pih) * (2 + plf0 - 3 * plf)] \\ &+ nb * (4 - 3 * pih) * (2 + pbf0 - 3 * pbf) \\ &+ np * (4 - 3 * pih)\end{aligned} \tag{4.7}$$

4.2.7.8 Pipelined Memory (3,6)

$$\begin{aligned}\text{Execution time} &= ni * (4 - 3 * pih) \\ &+ nl * (4 - 3 * pih) * (5 + plf0 - 6 * plf) \\ &+ nb * (4 - 3 * pih) * (2 + pbf0 - 3 * pbf) \\ &+ np * (4 - 3 * pih)\end{aligned} \tag{4.8}$$

4.2.7.9 Pipelined Memory (6,3)

$$\begin{aligned}\text{Execution time} &= ni * 1 \\ &+ nl * (5 + plf0 - 3 * plf - 3 * pdh) \\ &+ nb * (5 + pbf0 - 6 * pbf) \\ &+ np * 1\end{aligned} \tag{4.9}$$

4.2.8 Experimental Results

It is advantageous to study the candidate pipelines in four workload environments. First, we evaluate each pipeline in the workload environment represented by the entire set of benchmark programs. Then we evaluate each pipeline in arithmetic-heavy, branch-heavy, and load-heavy environments represented by the benchmark programs intmm, puzzle, and towers, respectively.

As indicated in the last section, the ideal pipeline performance standard is one datapath cycle per useful instruction. We instead plot on the vertical axis the number of datapath cycles per 1000 *packed MIPS instructions*. Although not every MIPS instruction contains exactly one useful operation, we do have an easily obtainable basis for pipeline comparison. On the horizontal axis we have a range of values for either a compiler parameter - pbf or plf, or a memory parameter - pih or pdh. Thus, we are in a position to determine the effect of both compiler and memory system capability on the performance of our candidate pipelines.

In the workload consisting of the ten benchmarks, we show the pipeline performances as a function of each of our four parameters: pih, pdh, pbf, and plf. These results are shown in Figures 4.10 - 4.21.

The arithmetic-heavy benchmark, intmm, is interesting because of its low number of branches and loads, and because of the high packing rate achieved by the MIPS compiler. For this benchmark we show the effect that pbf and plf have on pipeline performance. These results are shown in Figures 4.22 - 4.27.

The branch-heavy benchmark, puzzle allows us to observe the performance of our pipelines in an environment with a large amount of iteration. For this benchmark we amplify the effect that branches have on pipeline performance by showing the performance as a function of pbf in Figures 4.28 - 4.30.

The load-heavy benchmark, towers, allows us to observe the great number of data loads associated with procedure call/return overhead in a highly-recursive environment. We show the effect of plf on performance in Figures 4.31 - 4.33.

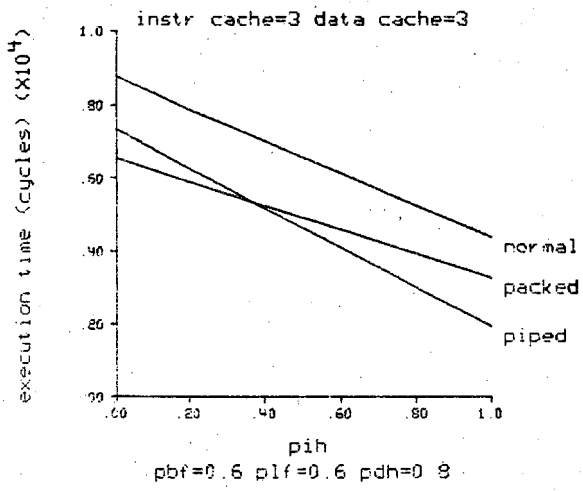


Figure 4.10 Pipeline Performance vs. "pih" in (3,3) Configuration for All Benchmarks.

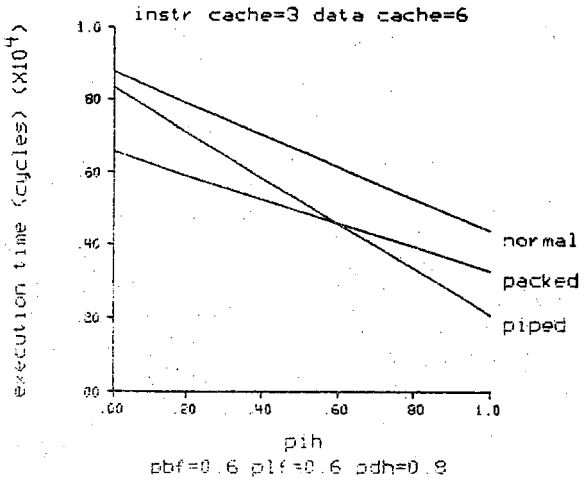


Figure 4.11 Pipeline Performance vs. "pih"in (3,6) Configuration for All Benchmarks.

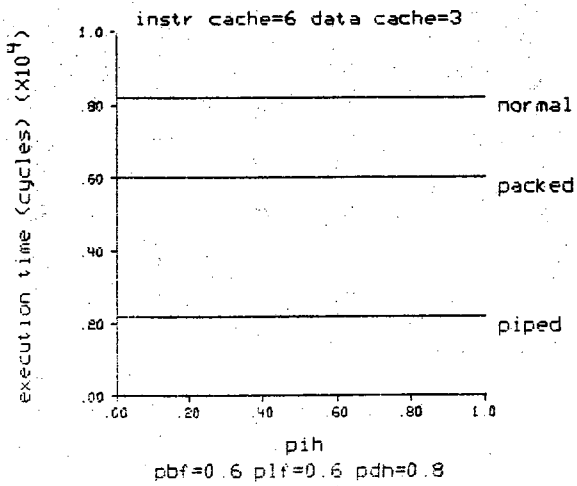


Figure 4.12 Pipeline Performance vs. "pih" in (6,3) Configuration for All Benchmarks.

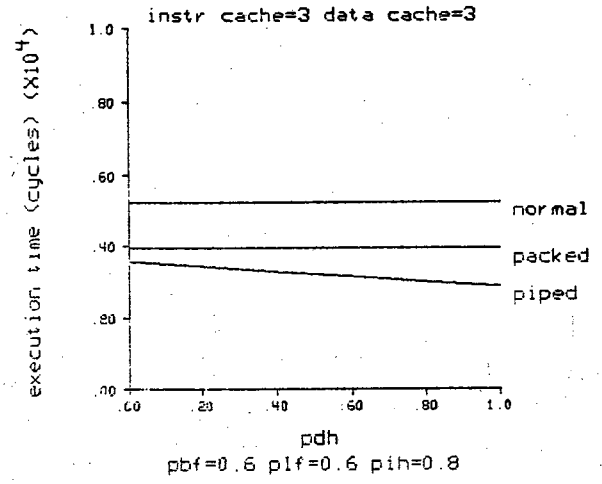


Figure 4.13 Pipeline Performance vs. "pdh"in (3,3) Configuration for All Benchmarks.

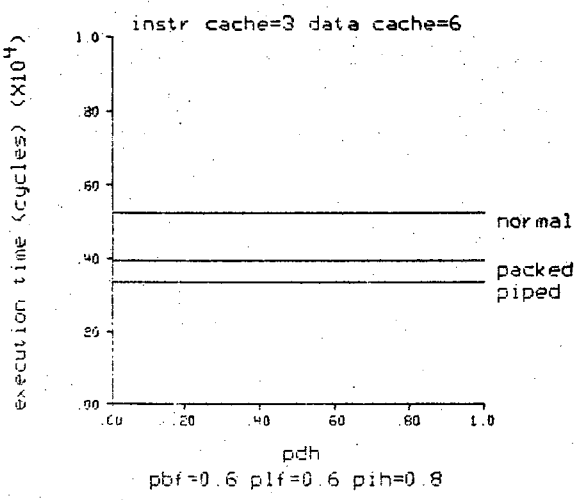


Figure 4.14 Pipeline Performance vs. "pdh" in (3,6) Configuration for All Benchmarks.

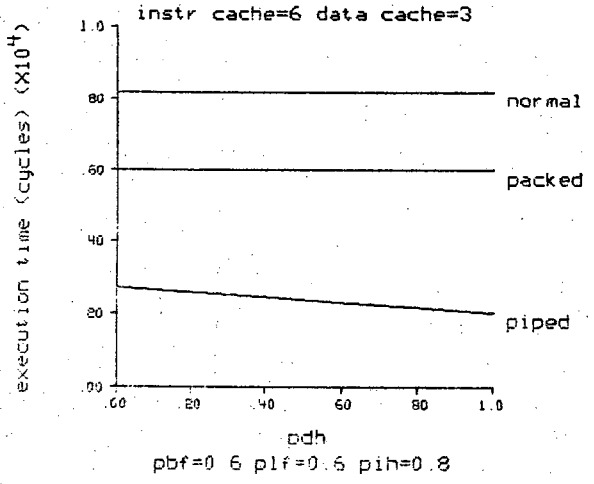


Figure 4.15 Pipeline Performance vs. "pdh"in (6,3) Configuration for All Benchmarks.

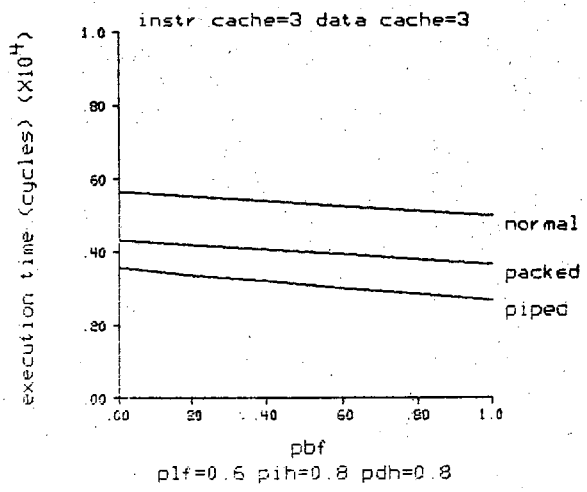


Figure 4.16 Pipeline Performance vs. "pbf" in (3,3) Configuration for All Benchmarks.

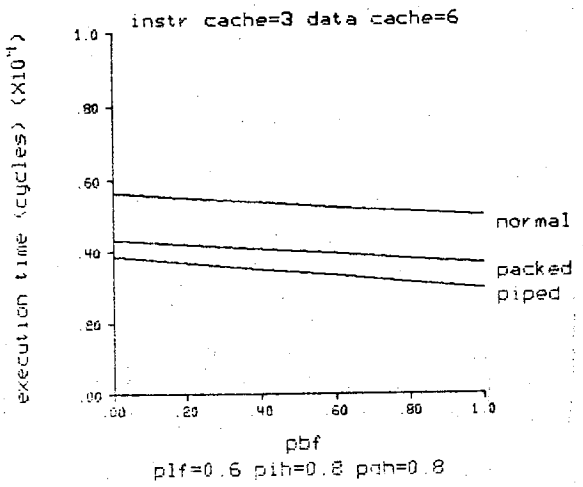


Figure 4.17 Pipeline Performance vs. "pbf"in (3,6) Configuration for All Benchmarks.

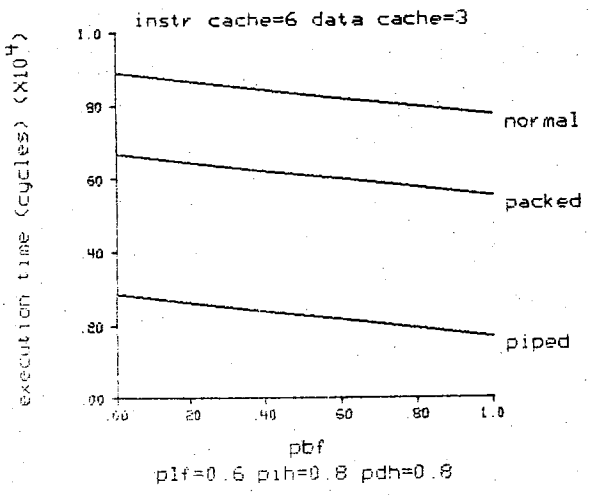


Figure 4.18 Pipeline Performance vs. "pbf" in (6,3) Configuration for All Benchmarks.

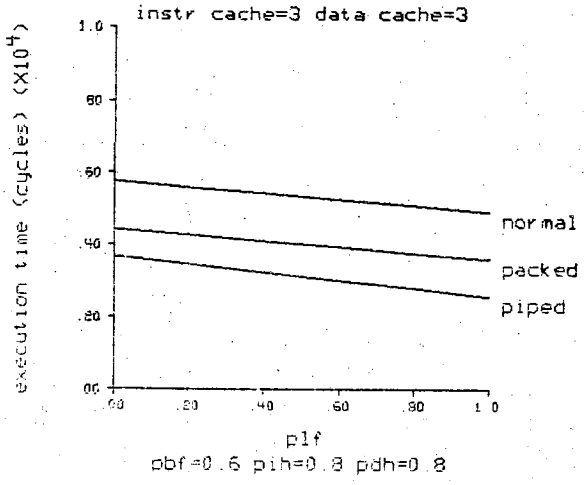


Figure 4.19 Pipeline Performance vs. "plf"in (3,3) Configuration for All Benchmarks.

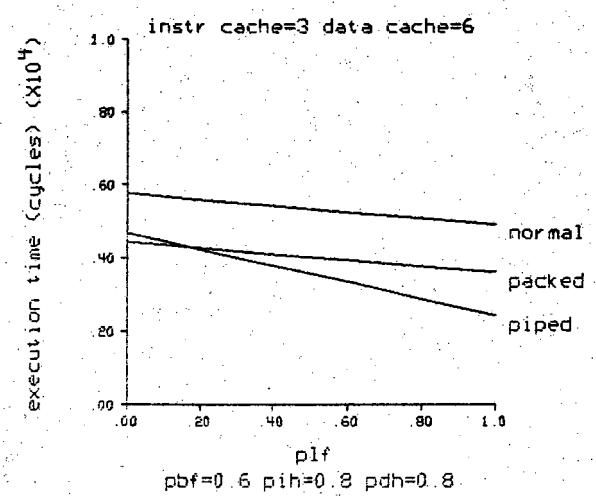


Figure 4.20 Pipeline Performance vs. "plf" in (3,6) Configuration for All Benchmarks.

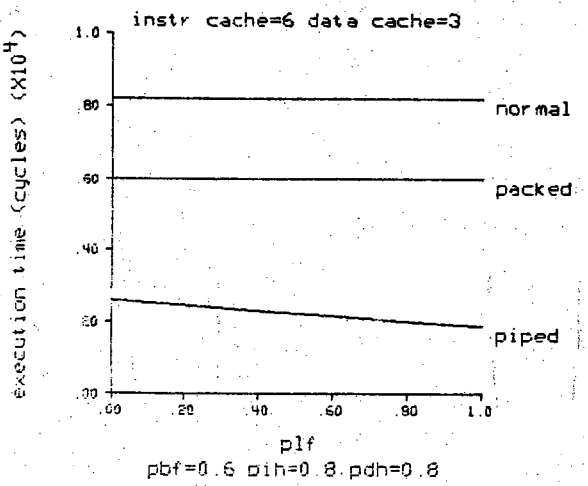


Figure 4.21 Pipeline Performance vs. "plf"in (6,3) Configuration for All Benchmarks.

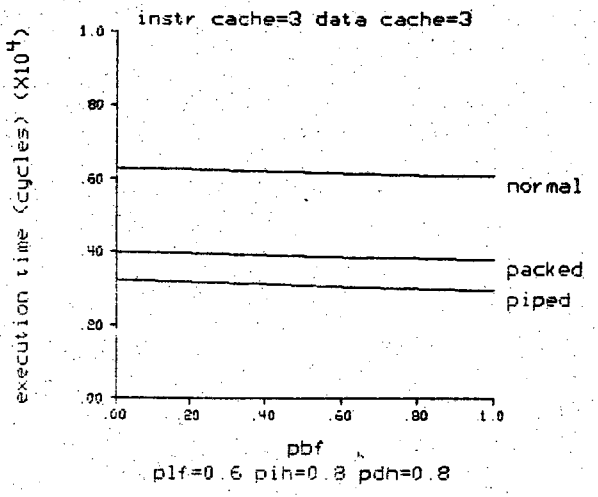


Figure 4.22 Pipeline Performance vs. "pbf" in (3,3) Configuration for Arithmetic-heavy Benchmark.

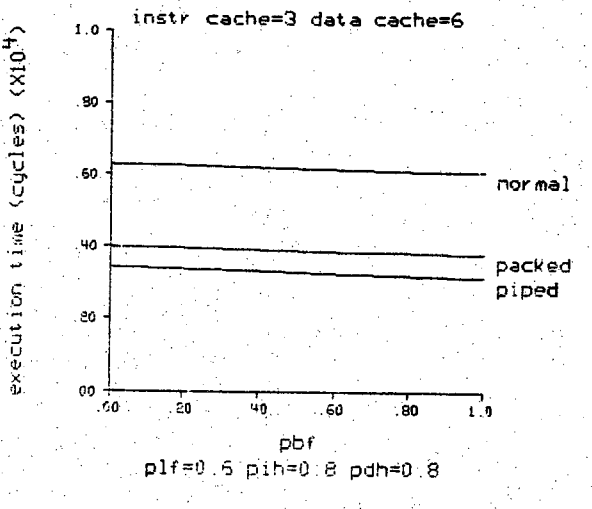


Figure 4.23 Pipeline Performance vs. "pbf"in (3,6) Configuration for Arithmetic-heavy Benchmark.

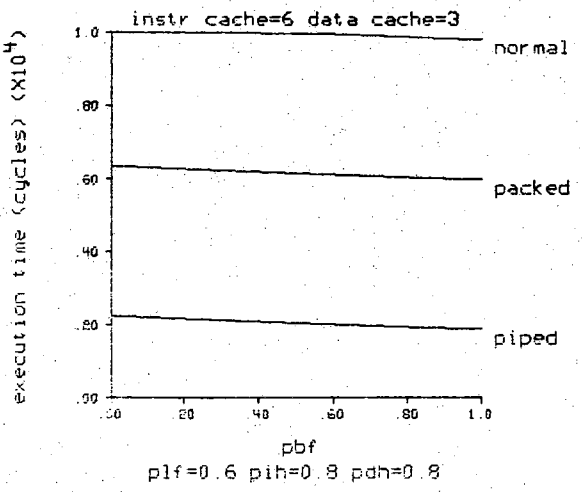


Figure 4.24 Pipeline Performance vs. "pbf" in (6,3) Configuration for Arithmetic-heavy Benchmark.

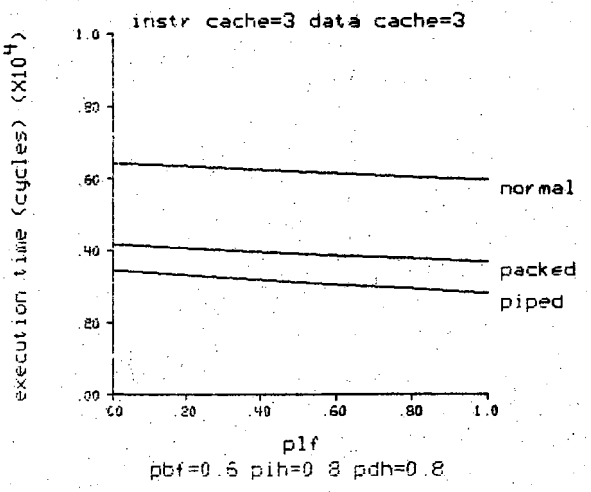


Figure 4.25 Pipeline Performance vs. "plf"in (3,3) Configuration for Arithmetic-heavy Benchmark.

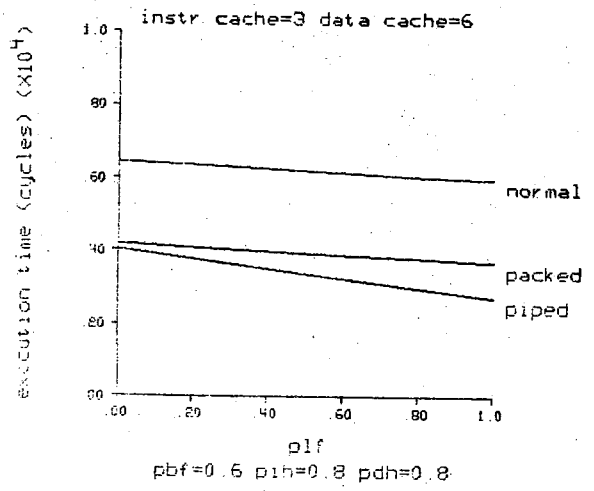


Figure 4.26 Pipeline Performance vs. "plf" in (3,6) Configuration for Arithmetic-heavy Benchmark.

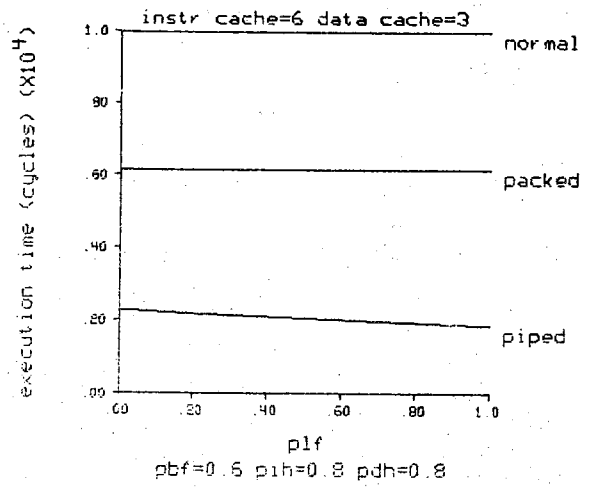


Figure 4.27 Pipeline Performance vs. "plf"in (6,3) Configuration for Arithmetic-heavy Benchmark.

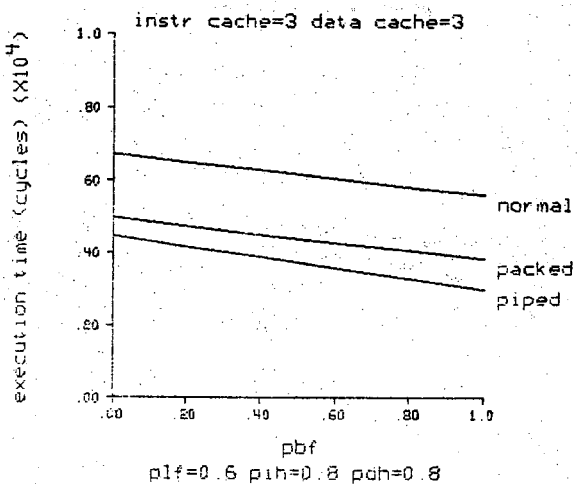


Figure 4.28 Pipeline Performance vs. "pbf" in (3,3) Configuration for Branch-heavy Benchmark.

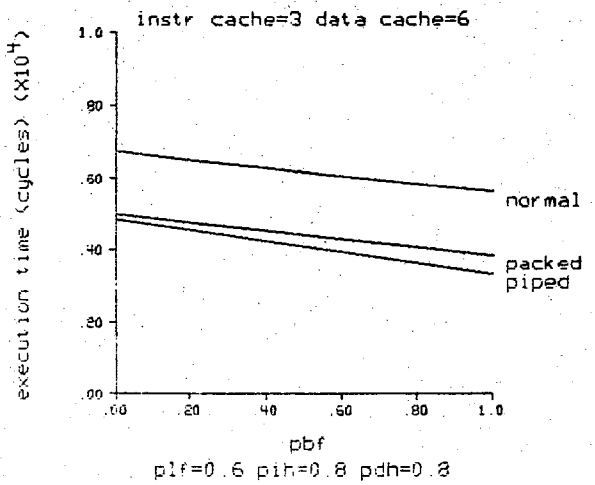


Figure 4.29 Pipeline Performance vs. "pbf"in (3,6) Configuration for Branch-heavy Benchmark.

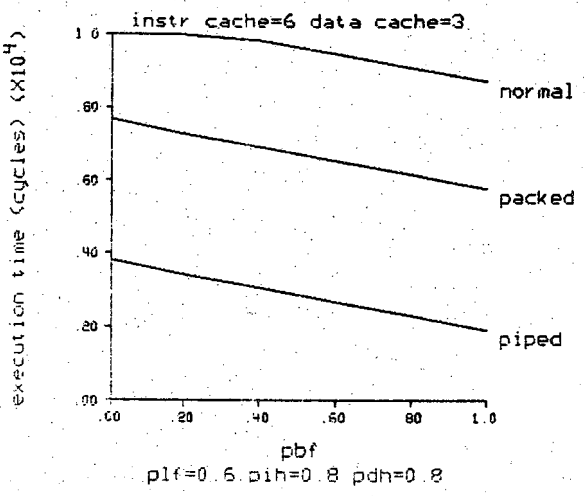


Figure 4.30 Pipeline Performance vs. "pbf" in (6,3) Configuration for Branch-heavy Benchmark.

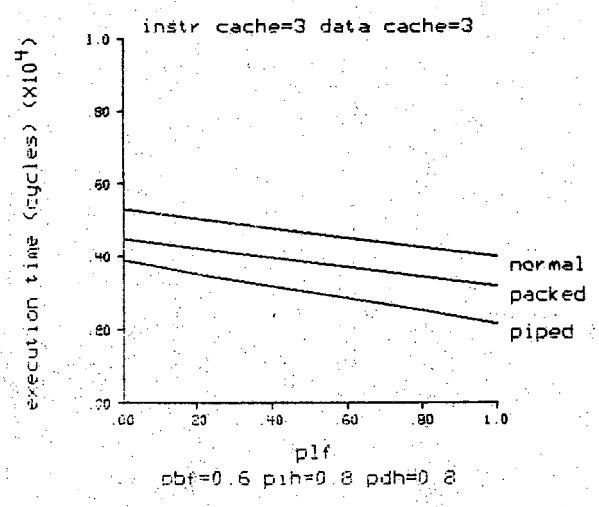


Figure 4.31 Pipeline Performance vs. "plf"in (3,3) Configuration for Load-heavy Benchmark.

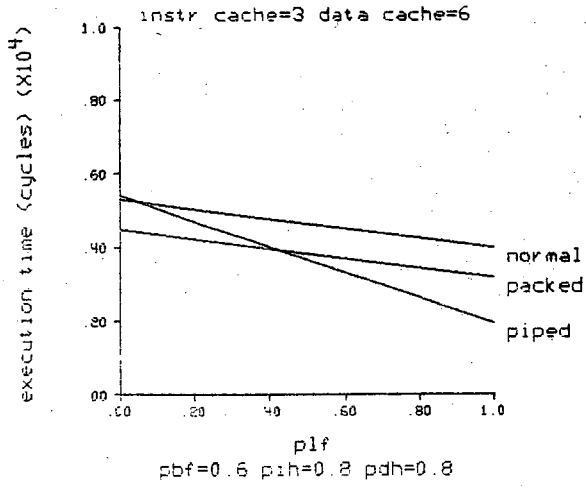


Figure 4.32 Pipeline Performance vs. "plf" in (3,6) Configuration for Load-heavy Benchmark.

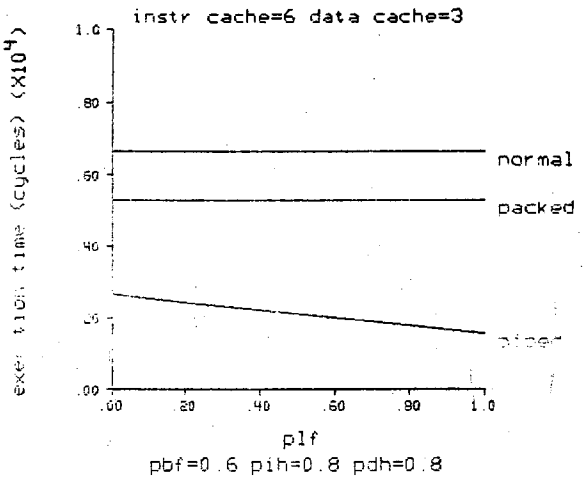


Figure 4.33 Pipeline Performance vs. "plf"in (6,3) Configuration for Load-heavy Benchmark.

4.2.9 Discussion

4.2.9.1 Candidate Pipeline Comparison

Clearly, the pipelined memory pipeline is generally superior to both the normal Silicon and packed pipelines. This is not surprising, because the pipelined memory pipeline is the only pipeline of the three potentially able to execute one useful operation per datapath cycle. The packed pipeline can sometimes execute two useful operations per three datapath cycles, while the normal Silicon pipeline is limited to one useful operation per three datapath cycles. In only a very few pathological cases is the Silicon pipeline performance equal to the performance of either of the other two pipelines. Clearly, a Silicon-like pipeline performs very poorly in our model of a GaAs processor environment.

It is apparent that the pipelined memory pipeline is the most sensitive to variations in all four parameters. This results from the "leanness" of the pipelined memory pipeline. Although it has the highest potential performance, it also experiences the most degradation from unfilled branch fillin slots, unfilled load fillin slots, and cache misses.

An interesting result is the lower sensitivity to pih demonstrated by the packed pipeline. This results because fewer instructions must be fetched when instruction packing is used; therefore, the number of instruction cache misses decreases as well. In fact, the packed pipeline outperforms the pipelined memory pipeline at low values of pih.

4.2.9.2 Memory Configuration Comparison

In most cases, the (3,3) memory configuration provides the highest performance, followed by the (3,6) configuration, and finally the (6,3) configuration. This is intuitively pleasing because we should expect faster off-chip memories to provide better performance. The (3,6) configuration generally outperforms the (6,3) configuration because, as seen in Table 4.1, instructions are fetched approximately four times as often as data.

A significant deviation from the general discussion just concluded is the much superior performance of the (6,3) memory configuration when the pipelined memory pipeline is used. In fact, this combination provides the best performance to be observed in this experiment. The reasons for this are the memory pipelining which reduces the "effective" instruction delay to one

datapath cycle, and the (6,3) configuration, which eliminates the penalty of instruction cache misses (the off-package memory is assumed to have infinite capacity).

4.2.9.3 Compiler and Memory Parameter Comparison

Of all the parameters pih, pdh, pbf, and plf, the one which most heavily influences pipeline performance is pih. This is very reasonable because instruction fetches are more frequent than either data loads or branches.

In the (3,6) memory configuration, the parameter pdh has no effect on performance. This is because in this configuration, the data memory hierarchy has only one level of infinite capacity. Therefore, data cache misses are not defined.

In the (6,3) memory configuration, the parameter pih has no effect on performance because of the same reasoning just presented.

For both the normal Silicon and packed pipelines, pdh has no effect on performance in any memory configuration. These two pipelines have such long effective instruction fetch times that they can absorb long data load times associated with data cache misses. In fact, the delays caused by instruction fetches negate any performance improvement that a data cache might provide.

For both the normal Silicon and packed pipelines in the (6,3) memory configuration, plf has no effect on performance. The effective instruction fetch delay for these two pipelines is so long that any semblance of instruction pipelining is lost.

4.2.9.4 Workload Comparison

One of the most visible differences between the workloads is the radical difference in execution time between them. This is caused by our use of the number of datapath cycles per *packed MIPS instruction.* Because of differences in the packing rates of MIPS instructions from workload to workload, there is a large difference in number of datapath cycles required by the pipelined memory and normal Silicon pipelines, both of which use unpacked instruction formats. A higher packing rate for MIPS instructions implies more unpacked instructions must be executed. From Table 4.1, the MIPS packing rate is 37 percent for the ten-benchmark workload, 64 percent for the arithmetic-heavy workload, 49 percent for the branch-heavy workload, and 20 percent for the load-heavy workload. Therefore, we expect the execution time in "number of

datapath cycles per MIPS instruction" to be highest in the arithmetic-heavy workload for both the pipelined memory and normal Silicon pipelines.

In the arithmetic-heavy workload environment, we observe almost no performance dependency on either pbf or plf. This is due to the extremely low number of branches (6 percent) and loads (13 percent) in this benchmark.

For the branch-heavy workload environment we do observe a higher performance dependency on pbf as we expect. This is due to the high frequency of branches (32 percent) in this benchmark.

For the load-heavy workload environment we see an increased performance dependency on plf as also expected. This is again due to the high percentage of loads (36 percent) in this benchmark.

4.2.9.5 Summary

The most significant result of this experiment is the superb performance of the pipelined memory pipeline in all environments, even though its performance was hurt by the branch-heavy and load-heavy benchmarks. In the branch-heavy environment, even at a pbf of zero, the pipelined memory pipeline still performs best. The load-heavy benchmark is more punishing when the (3,6) memory configuration is used, but the pipelined memory pipeline is still best at values of plf above 0.4. In the arithmetic-heavy benchmark, the low frequency of branches and data loads allows the pipelined memory pipeline to excel even at very low values of pbf and plf.

Another important result concerns the tradeoff between the (3,6) and (6,3) memory configurations in implementations which cannot have both a fast instruction cache and a fast data cache. Allocating the slower memory to the instructions can be beneficial from a performance standpoint as long as the memory is pipelined or, perhaps, if multiple consecutive instructions are fetched. Because data memory accesses are generally not as regular as instruction memory accesses, the opportunity for exploiting pipelined and interleaved data memory is limited. There are exceptions, of course, such as at procedure boundaries, and for applications with large amounts of regular data structures such as arrays. In general though, the same high degree of spatial locality which yields high instruction cache hit ratios can be exploited by pipelined or parallel access methods as well. If the less-regular data can be managed sufficiently well to provide good hit ratios at the faster, smaller data memory, then improved performance should result. This is the technique adopted by the designers of the Transputer, as they fetch four instructions into

their single-chip processor concurrently, while advocating that the large on-chip memory be used for data storage [Whitb85].

One final comment concerns the issue of instruction packing. As demonstrated by every pipeline performance graph which measured the effect of pih, the packed pipeline reduces the negative effect of low instruction cache hit ratios in multiple-level instruction memories. If a small instruction cache is to be used in a GaAs processor system, then instruction packing certainly deserves strong consideration.

4.3 Instruction Format Experiment

We now describe an experiment undertaken in order to evaluate ten instruction formats in a GaAs processor environment. We begin by explaining our motivation for performing the experiment, and then we present our choice of formats for examination. We describe our evaluation criteria and discuss the general theory and implementation of our instruction format evaluation methodology. We then discuss the procedure and results of our experiment, presented as three sub-experiments. We finish with a discussion of our results.

4.3.1 Rationale

In the Silicon environment, the RISC philosophy has demonstrated the greater importance of good pipeline design over instruction compactness. However, as pointed out in Section 3.2.6, instruction formats which result in small program sizes can be more advantageous in GaAs implementations than in Silicon implementations. Again, the reasons for this include the smaller capacity of GaAs memory chips, as well as the higher ratio of off-chip to on-chip delays for GaAs. We believe that it is desirable to determine what effect compact instruction formats can have in a GaAs processor environment.

4.3.2 Candidate Instruction Format Descriptions

Section 3.2.6 indicated that considerable redundancy usually exists in the immediate field and address fields of Silicon instruction formats. The constants located in immediate fields are usually small, and the higher order bits in such cases convey no essential information to the processor. Most computations don't require three addresses; therefore, three-address instruction formats often have address fields with no information content. We find redundancy

elimination in these two particular areas to be deserving of further exploration in the context of a GaAs environment.

We choose to study ten instruction format candidates for incorporation into a GaAs processor. In addition to these, we will also include the MIPS format. The instruction fields for the ten candidate formats are shown in Table 4.2, while the relevant instruction fields for the MIPS format are described in Appendix B.

Each of our candidate instruction formats has its structure encoded into its name. For example, the format "28(3210)" contains 28 bits and may use either 3, 2, 1, or 0 registers. In general, when a lower number of registers are used, the immediate field may be increased accordingly. Since each format requires eight bits for its opcode, and register specifications require four bits, the immediate field lengths for the 28(3210) format are 8, 12, 16, and 20 bits, corresponding to 3, 2, 1, and 0 registers, respectively.

The candidate instruction formats are chosen on their ability to be directly compared, so that a variety of sub-experiments are possible.

Formats 28(3) and 24(2) allow a direct tradeoff study between the higher number of register fields of 28(3) versus the reduced total number of bits in 24(2).

Formats 28(3) and 24(32) allow a direct tradeoff study between the higher immediate field length of 28(3) versus the reduced total number of bits in 24(32).

Comparing the performance of the formats in each of the pairs: 28(3) versus 28(3210), 24(2) versus 24(210), 24(32) versus 24(3210), 20(32) versus 20(3210), and 16(21) versus 16(210), we can examine the result of shifting instruction bits from register fields to the immediate field, when a full set of register addresses is not required.

With formats 16(21) and 16(210), two instructions can be fetched concurrently using the same processor-memory bandwidth as a single 32-bit instruction. Therefore, packed versions of formats 16(21) and 16(210) can be compared to the packed MIPS format.

4.3.3 Evaluation Criteria

In this study of compact instruction formats there are three types of information which are of interest to us, and which will form our basis for comparison.

Table 4.2 Instruction Fields for Candidate Instruction Formats.

Format	Opcode (# bits)	Reg1 (# bits)	Reg2 (# bits)	Reg3 (# bits)	Imm1 (# bits)	Imm2 (# bits)	Total (# bits)
28(3)	8	4	4	4	8		28
28(3210)	8	4	4	4	8		28
	8	4	4		12		28
	8	4			16		28
	8				20		28
24(32)	8	4	4	4	4		24
	8	4	4		4	4	24
24(3210)	8	4	4	4	4		24
	8	4	4		4	4	24
	8	4	4		8		24
	8	4			12		24
	8				16		24
24(2)	8	4	4		8		24
24(210)	8	4	4		8		24
	8	4			12		24
	8				16		24
20(32)	8	4	4	4			20
	8	4	4		4		20
20(3210)	8	4	4	4			20
	8	4	4		4		20
	8	4			8		20
	8				12		20
16(21)	8	4	4				16
	8	4			4		16
16(210)	8	4	4				16
	8	4			4		16
	8				8		16

First, we would like to know the magnitude of the effect of short instruction formats on the overall program size.

Second, we would also like to see how the use of short instructions affects the number of instructions which must be executed.

Finally, we are most interested in determining the effect of short instructions on execution time.

4.3.4 Evaluation Theory and Implementation

We are again using the simulation tools which were described in Section 4.1. Our approach in this experiment is to utilize all the simulation tools in their normal way, except for the MIPS simulation program. We instead modify this program to account for instruction set differences between the MIPS format and our candidate formats. In this way we obtain a performance measure for each of our candidate instruction formats.

We have identified three areas where our candidate instruction formats differ from the MIPS instruction format, and these provide the basis for our simulation program changes.

First, the MIPS instruction format is packed; therefore, some MIPS *instructions* contain two *operations*. Since none of the candidate formats use packing, a packed MIPS instruction requires two candidate instructions.

Second, some of the candidate formats only have two register address fields, while the MIPS operations within MIPS instructions can have two or three register addresses. Therefore, in some cases a single MIPS operation will require two candidate instructions.

Third, the candidate formats all have shorter immediate fields than the MIPS format. Because of this, in some cases two candidate instructions may be required in order to execute a single MIPS operation.

Because of these differences between the MIPS and candidate formats, there will be differences in their static instruction counts and dynamic instruction counts. The static instruction count is the number of instructions which are contained within a program; it is a measure of program *size*. The dynamic count is the number of instructions which are executed when a program is run; it is a measure of program *execution time*.

Because we know the instruction field sizes for the MIPS format and for all the candidate formats, it is relatively straightforward to determine, for each of the candidate formats, which MIPS instructions require multiple candidate

instructions. We show in Appendix B the ten candidate instruction formats and the number of instructions which they require in order to execute MIPS instructions of given characteristics. Actually, Appendix B shows the "additional" number of candidate instructions which are needed.

A straightforward modification to the MIPS simulator is necessary in order to implement the mapping from each "MIPS instruction" to the "number of candidate instructions" shown in Appendix B. We do not change the normal execution of the MIPS program with our additions; we merely add code to implement our data gathering. Again, our changes to the simulator in this experiment are minor in comparison to the size of the simulation program, and the large total program size prohibits us from listing these changes here.

4.3.5 Static Instruction Count Subexperiment

4.3.5.1 Procedure

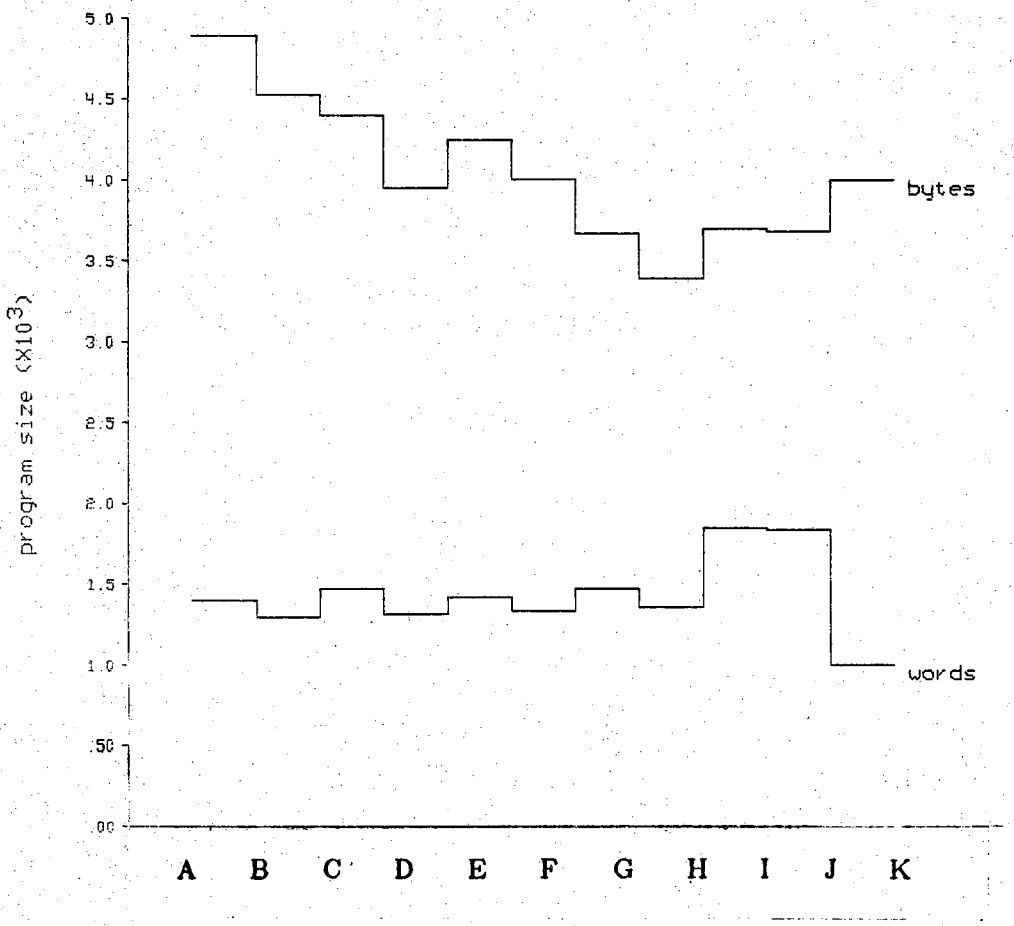
Obtaining the static instruction count is relatively straightforward. We add code to the section of the MIPS simulator which loads the MIPS instructions into memory. For each of the candidate formats, we examine each MIPS instruction as it is loaded and perform the analysis of Appendix B. After the loading is complete, the total number of candidate instructions is then output to a file.

4.3.5.2 Results

We show the results of this subexperiment in Figure 4.34. We show the program size in terms of instruction words and bytes. In this figure, the candidate instruction counts are normalized to 1000 MIPS instructions. These results are for the entire set* of benchmark programs described in Section 4.1, and each benchmark receives equal weight.

---

* The benchmark "towers" is not included in this experiment because of its excessive computational requirements, as it is currently written.



A: format 28(3)
B: format 28(3210)
C: format 24(32)
D: format 24(3210)
E: format 24(2)
F: format 24(210)
G: format 20(32)
H: format 20(3210)
I: format 16(21)
J: format 16(210)
K: MIPS format

Figure 4.34 Instruction Format Static Instruction Counts.

4.3.6 Dynamic Instruction Count Subexperiment

4.3.6.1 Procedure

We can obtain the dynamic instruction count in much the same manner as we obtain the static instruction count. However, instead of adding code to examine MIPS instructions as they are *loaded* by the simulator, we add code to examine them as they are *executed*. The modified simulation program then outputs the total number of candidate instruction executions into a file.

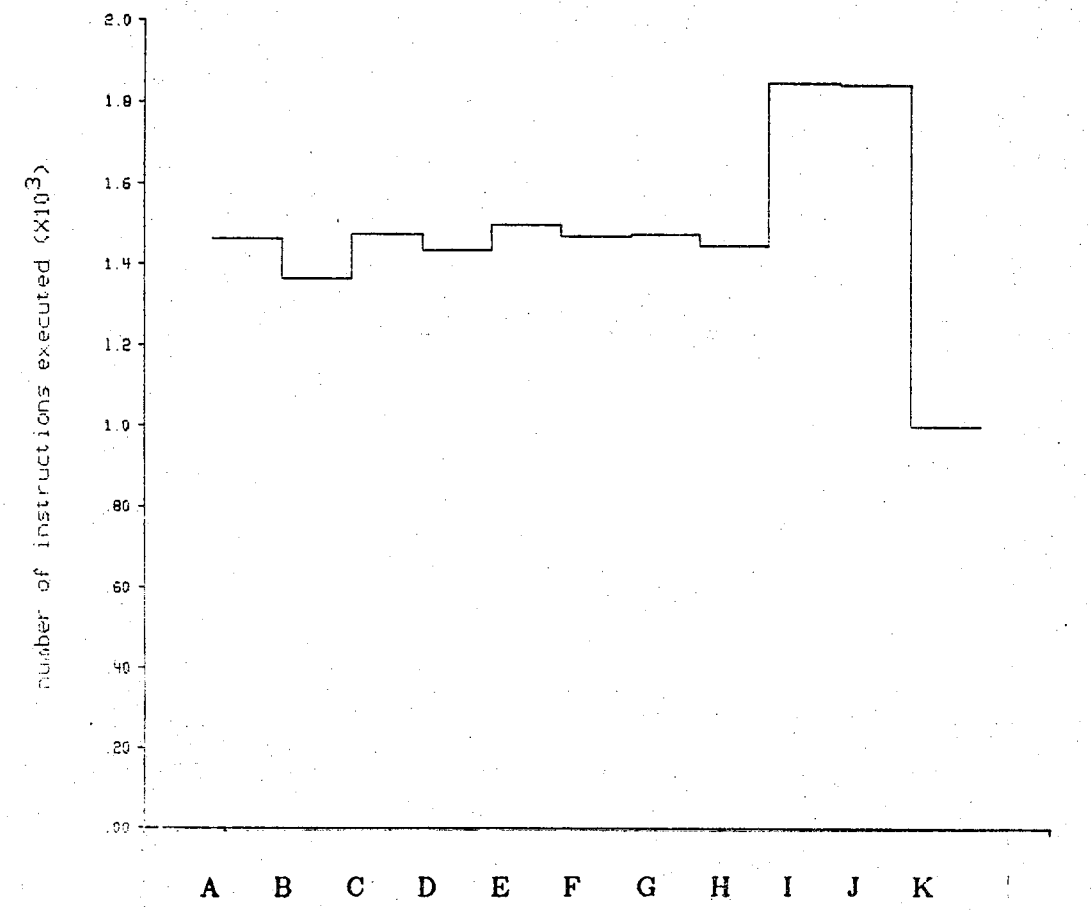
4.3.6.2 Results

These results are shown in Figure 4.35. Again, the dynamic instruction count for each candidate format is normalized to 1000 MIPS instruction executions. These results are for the same set of benchmarks that were used in the previous subexperiment, where each benchmark again receives equal weight.

In Table 4.3 we show the breakdown of costs which cause the candidate pipelines to have higher dynamic instruction counts. The values in this table are the percentage increase in dynamic execution count, measured as a percentage of one MIPS instruction, caused by the associated cost. Most of the candidate formats show immediate costs of approximately ten percent; only the 16-bit formats are significantly affected by address costs; and all the formats show a relatively high packing cost.

4.3.7 Execution Time Subexperiment

Determining the execution time for each of our candidate instruction formats is more difficult than calculating the above static and dynamic instruction counts. Once again, the benefits derived from compact instructions are attributable to higher cache hit rates, main memory hit rates, etc. Therefore, we must incorporate into our architectural model a cache simulator; however, the small size of our benchmark programs prohibits them from being used as a suitable workload model for any cache of reasonable size.



A: format 28(3)
B: format 28(3210)
C: format 24(32)
D: format 24(3210)
E: format 24(2)
F: format 24(210)
G: format 20(32)
H: format 20(3210)
I: format 16(21)
J: format 16(210)
K: MIPS format

Figure 4.35 Instruction Format Dynamic Instruction Counts.

Table 4.3 Breakdown of Dynamic Costs for the Candidate Formats.

Format	Immediate Cost*	Address Cost*	Packing Cost*
28(3)	10	0	37
28(3210)	0	0	37
24(32)	11	0	37
24(3210)	7	0	37
24(2)	10	3	37
24(210)	7	3	37
20(32)	11	0	37
20(3210)	8	0	37
16(21)	11	39	37
16(210)	11	39	37

* Measured as a percentage of one MIPS instruction.

4.3.7.1 Workload Model - Cache Model Discussion

To solve the mismatch between our workload model and the desired size of our cache model, we can either modify our workload model or modify our cache model. Our first choice is to modify our workload model; however, we are thwarted by the inability of our MIPS compiler, which is not of production quality [Linn85], to produce reliable code for large benchmarks. Therefore, we must modify our cache model in order to match it with our workload model.

We first determine the size of both an application workload that we wish to model and a cache that we wish to model. For our application workload we choose a size of 256K bytes. For our cache size we select a range of values: 4K, 8K, 16K, and 32K bytes. We then observe that our benchmark programs typically require approximately 4K bytes of storage for the instructions that are actually used; many of the system procedures which are loaded are not actually used. We use this ratio of "real world" program size to "simulation world" program size of 64 to 1, and apply it to our "real world" cache sizes to determine our "simulation world" cache sizes of 64, 128, 256, and 512 bytes.

However, there is an undetermined degree of inaccuracy in this approach, as there are at least two problems with it.

First, a single instruction in our "simulation world" corresponds to 64 instructions in the "real world." However, the only way the execution of a single instruction in our "simulation world" cache can accurately model the execution of 64 instructions in a "real world" cache, is when the 64 "real world" instructions are stored in sequential memory locations. Since sequential execution leads to high hit ratios, and since the sequential execution of 64 instructions is extremely rare, we can expect our "simulation world" cache to experience higher hit ratios than a "real world" cache.

Fortunately, the second cause of inaccuracy tends to balance the effect of the first problem. The locality of reference which leads to large cache hit ratios is largely the result of HLL programming constructs such as loops and recursive procedure calls. It's observed that the sizes of these loops and recursive procedures are the same whether they're in "real world" programs or our "simulation world" programs. Because our "simulation world" cache is only 1/64th as large as the "real world" cache it's trying to model, the "simulation world" cache will be less successful at capturing entire loops and recursive procedures than the "real world" cache. Therefore, we can also expect our "simulation world" cache to experience poorer hit ratios than a "real world" cache.

4.3.7.2 Cache Simulator Description

Our cache simulation program was designed at Purdue. It is a simple yet flexible simulation tool. Among the cache parameters which may be changed are the cache size, the block size, the access time for a hit, the access time for a miss, and the cache configuration: instruction only, data only, or combined.

The cache simulator implements a direct mapped placement policy. As discussed in Section 3.2.7.3, direct mapping is desirable for GaAs caches because of its low overhead and relatively good performance at small cache sizes.

The cache simulator uses a write-through policy, and a data write resulting in a cache miss causes replacement to occur; however, there is no time penalty associated with data cache write misses. As discussed in Section 3.2.7.5, pipelined memory systems are advantageous in a GaAs processor system. A write-through policy and data cache write misses may cause no additional delay in a pipelined memory system.

We choose a block size of two words for our cache simulator. An early implementation of the simulation program did not allow a block size of one to be used, as we would have liked.

4.3.7.3 MIPS Simulator Modifications

We interface the MIPS instruction simulator to the cache simulator by replacing the existing memory access operations with calls to the cache simulation procedure. Data memory accesses are implemented in a straightforward manner; however, because a single MIPS instruction may correspond to multiple candidate instructions, we require a mechanism to allow us to perform multiple calls to the cache simulator for a single MIPS instruction.

Our solution is to use an array of records. Each element (record) of the array corresponds to a single MIPS instruction, and the array is indexed by the MIPS instruction address. Each record contains a count and an address. The count represents the number of candidate instructions required by this MIPS instruction. The address represents the candidate instruction's address and is the beginning address of the set of candidate instructions representing this MIPS instruction. The candidate instruction address at MIPS instruction i equals the sum of the counts of the previous i-1 MIPS instructions.

The above array is initialized when the MIPS program is loaded, again using the analysis of Appendix B. During simulation, when a MIPS instruction fetch is executed, the address from the appropriate record of the array is used to exercise the cache simulator. If the count from the record is greater than one, then the address is repeatedly incremented and sent to the cache simulator until "count" accesses have been simulated. A separate variable is used in order to maintain the number of instruction cycles lost due to both instruction and data memory accesses. Upon completion of the simulation, this variable is written to a file. Also, the MIPS instruction fetch operation, which was replaced by the call to the cache simulator, is implemented inside the cache simulation procedure independent of the just described activity.

4.3.7.4 Procedure and Results

4.3.7.4.1 Procedure

We observed in the pipeline experiment the effect that instruction pipeline differences can have on performance. Therefore, for a fair instruction format comparison, we must select an instruction pipeline to be used by all the candidate formats. We find it most useful for this experiment to use the pipeline indicated by Figure 3.2. Therefore, one instruction is fetched for every three datapath cycles. For this subexperiment then, a cache hit results in a delay of three cycles and a miss results in a six cycle delay.

There are two parts to this subexperiment. First, we run the MIPS simulator and cache simulator to obtain the execution time for each instruction format as a function of cache size. Then we use this information in order to obtain the execution time for each cache size as a function of instruction format.

4.3.7.4.2 Results

In Figure 4.36 we show the results of the first subexperiment. This figure plots execution time versus cache size for three cache configurations: instruction cache only, data cache only, and combined instruction and data cache. In this figure, the execution time has been normalized to the number of instruction cycles necessary to execute 1000 MIPS instructions in its Silicon environment and with no cache miss penalties.

We only present one plot because the results for the other candidate formats are nearly identical to this one.

In Figures 4.37 and 4.38 we show the second set of plots. These figures show execution time versus instruction format for a particular cache size (64 and 512 bytes) and cache configuration (instruction-only and data-only). Again, the execution times have been normalized to the number of instruction cycles necessary to execute 1000 MIPS instructions in its Silicon environment and with no cache miss penalties.

4.3.7.5 Discussion

4.3.7.5.1 The Effect of Instruction Format on Instruction Counts

From Figure 4.34, the number of words required to implement the benchmark programs is seen to increase as the format lengths are decreased, as expected. All of the formats have static instruction counts within 50 percent of the instruction count of the MIPS format except for the two 16-bit formats: 16(21) and 16(210). These 16-bit formats have instruction counts almost 100 percent greater than the MIPS instruction count. Clearly, the transition from a 20-bit format to a 16-bit format can be damaging.

The number of bytes required to implement the benchmark programs, or equivalently, the program size, generally shows a decline as the format length is decreased, until the 16-bit formats are reached. The 20-bit formats generate the most compact programs, but the 16-bit formats achieve code sizes smaller than the MIPS format as well.

From Figure 4.35, the dynamic instruction count is seen to follow the same trend as the static instruction count. The 16-bit formats are again severely penalized. Tables 4.2 and 4.3 can help explain this phenomena. From Table 4.2 we observe that the 8-bit opcode requirement only leaves eight bits for addressing, enough for two addresses. The extremely large address cost for these two formats, indicated in Table 4.3, allows us to conclude that two addresses are very frequently not enough. From Table 4.3 we see that three addresses are required 28 percent of the time (39/137). This is much higher than we anticipated, in light of the 13 percent figure for 3-address assignment statements presented in [Myers82].

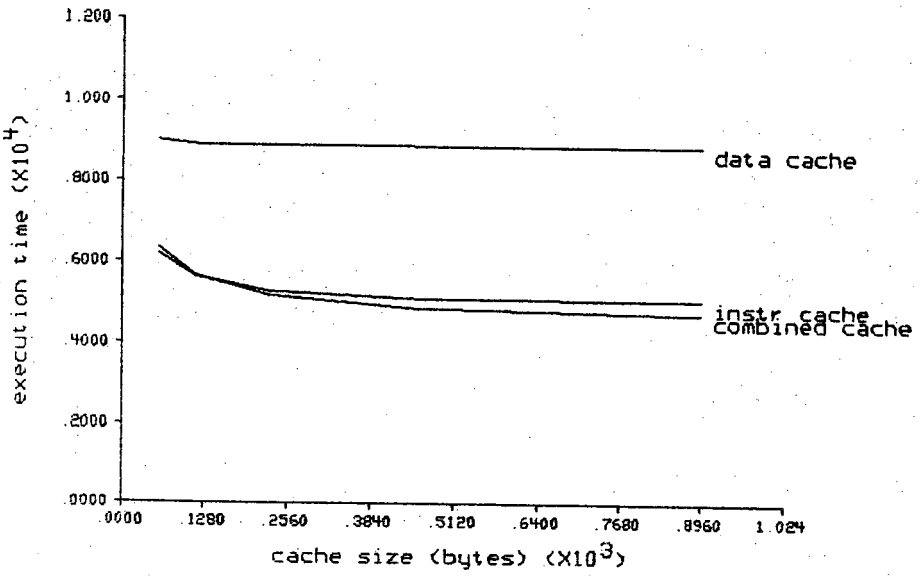
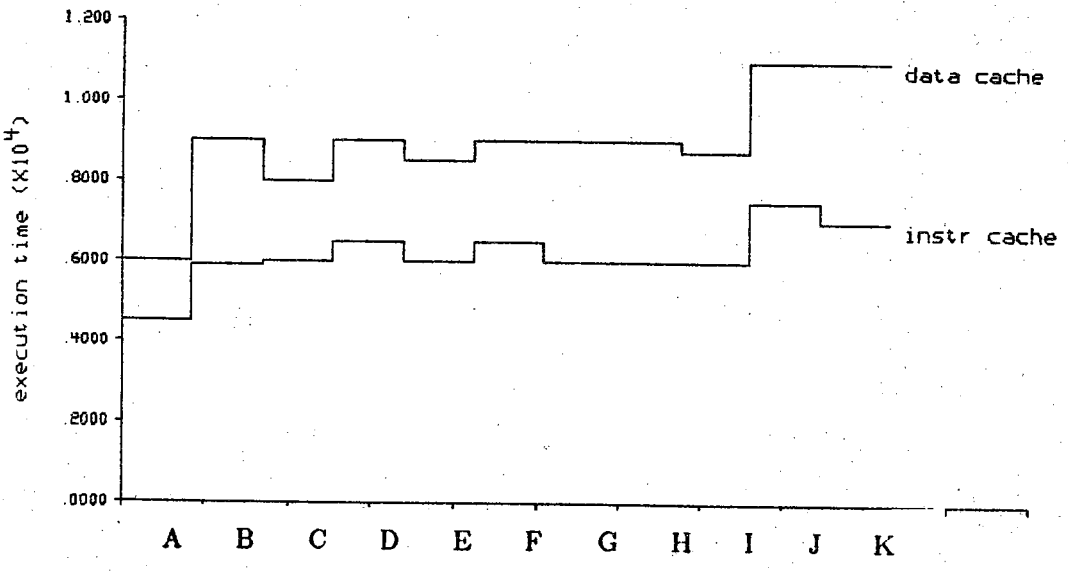
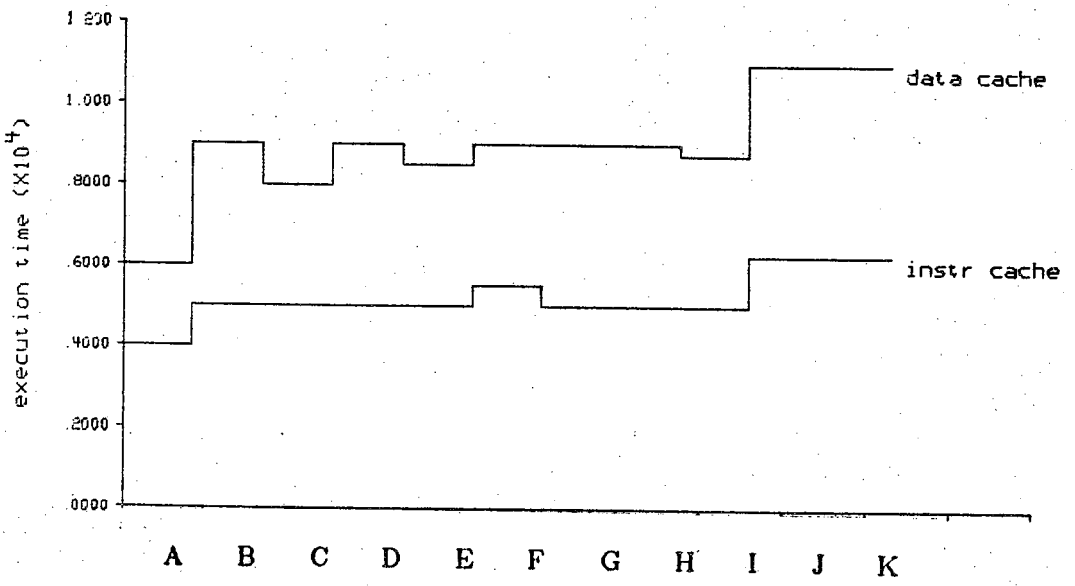


Figure 4.36 Instruction Format 28(3) Execution Time vs. Cache Size.



A: MIPS format
B: format 28(3)
C: format 28(3210)
D: format 24(32)
E: format 24(3210)
F: format 24(2)
G: format 24(210)
H: format 20(32)
I: format 20(3210)
J: format 16(21)
K: format 16(210)

Figure 4.37 Execution Time for each Instruction Format with 64-byte Cache.



A: MIPS format
B: format 28(3)
C: format 28(3210)
D: format 24(32)
E: format 24(3210)
F: format 24(2)
G: format 24(210)
H: format 20(32)
I: format 20(3210)
J: format 16(21)
K: format 16(210)

Figure 4.38 Execution Time for each Instruction Format with 512-byte Cache.

4.3.7.5.2 The Effect of Cache Size on Execution Time

From Figures 4.36 and 4.37 we can observe some general trends. For these formats, execution time generally decreases as cache size is increased. We obviously expect this; however, the small memory capacity at which the execution time decrease levels off clearly demonstrates the smallness of our benchmarks.

We also observe that a data cache alone does a much poorer job than either an instruction cache alone or a combined cache. The reason for this is the much higher frequency of instruction fetches than data loads in an environment which penalizes a non-cache access three cycles. We see a small advantage for the combined cache at increasing cache sizes but no significant difference between the performance of the instruction cache and the combined cache.

4.3.7.5.3 The Effect of Fewer Register Fields

Looking back at Table 4.2 we observe that the only difference between instructions formats 28(3) and 24(2) is the lower total number of bits in format 24(2) due to the removal of one register field. We now discuss the relative performance of these two instruction formats.

Figure 4.34 shows that the total program size for format 24(2) is approximately ten percent lower than the program size for format 28(3). From Figure 4.35, we see that the number of format 24(2) instructions executed is only a few percent higher than the number of format 28(3) instructions executed.

The execution times for these two instruction formats shown in Figures 4.36 and 4.37 indicate a generally slight inferior performance by the 24(2) instruction format. In memory configurations consisting of an instruction cache, the 24(2) format execution time is generally a few percent higher (approaching five percent) than the 28(3) execution time. In the data-cache-only configurations where we expect the relative format performances to match their relative number of instruction executions, the performance of both formats is nearly the same.

This experiment then indicates a slight degradation in performance in reducing instruction format size by eliminating one register field.

4.3.7.5.4 The Effect of Smaller Immediate Field Lengths

Again looking at Table 4.2, we observe that the only difference between instruction formats 28(3) and 24(32) is the lower total number of bits in format 24(32) due to its smaller immediate field. We now discuss the relative performance of these two instruction formats.

Figure 4.34 shows that the program size for format 24(32) is nearly ten percent lower than the program size for format 28(3). Again, Figure 4.35 shows the relative number of instructions executed, where it is seen that format 24(32) requires slightly more instruction executions than format 28(3).

The execution times shown in Figures 4.36 and 4.37 show little difference in performance between these two instruction formats, although the slight differences that do appear favor format 28(3). In general, the relative performance of these two instruction formats follows their relative number of instructions executed.

In this experiment then, we observe that reducing instruction format size by reducing immediate field size has very little impact on performance.

4.3.7.5.5 The Effect of Variable Immediate Field Sizes

From Table 4.2, the difference between instruction formats 28(3), 24(32), 24(2), 20(32), 16(21) and 28(3210), 24(3210), 24(210), 20(3210), 16(210), respectively, is the greater flexibility in immediate field size allowed in the second set of formats. The formats in the first set maintain a rigid immediate field, while formats in the second set increase their immediate field size to consume all the bits which are not needed by register addresses. We now discuss the relative performance of these two methods of implementing immediate fields.

From Figure 4.34 we observe that the program sizes of the variable-immediate-field formats are smaller, with the differences ranging from approximately ten percent in formats 28(3210) and 24(3210) to almost no change in format 16(210). The same trend is evident in the number of instruction executions shown in Figure 4.35. We can explain the lower improvement of the shorter instruction formats by viewing again Table 4.2, where we observe that the shorter formats just don't have enough bits to substantially lengthen their immediate fields.

The execution times shown in Figures 4.36 and 4.37 consistently show that the formats with variable length immediate fields perform better than the

formats with fixed length immediate fields by a small margin.

In this experiment we have observed that varying immediate field lengths to use the instruction bits not needed by register addresses does indeed lead to smaller program size, fewer instruction executions, and lower execution time.

4.3.7.5.6 The Use of Compact Formats for Instruction Packing

Instruction formats 16(21) and 16(210) are both only one half as long as a single MIPS instruction; therefore, two such instructions may be fetched in parallel and require no more processor-memory bandwidth than required by the MIPS format. We find it interesting to examine the performance of such a form of instruction packing.

From Figure 4.34 we observe that the program size of both 16-bit formats is nearly ten percent lower than the MIPS format. However, Figure 4.35 shows that the number of 16-bit instructions executed is nearly 90 percent higher.

If two instructions of the 16(21) format are concatenated to form a single 32-bit packed instruction, then this compact, packed instruction format would actually require five percent fewer instruction executions than the MIPS format. This is also true for the 16(210) format. It is apparent that this type of packed format is more successful at eliminating redundancy than is the MIPS format.

From Table 4.2 we see that much of the performance degradation in the 16(21) and 16(210) formats is due to addressing cost. Again, this cost results from the 2-address limit imposed by the short instruction length and 8-bit opcode. If 4-bit opcodes were to be used for some frequent operations then the number of instruction executions could be significantly reduced at the cost of additional decoding and control logic. Alternatively, 18-bit or 20-bit operations may instead be packed to provide much better performance with reasonable instruction sizes. If two 20(3210)-format instructions were packed, Figure 4.34 indicates that approximately 30 percent fewer of these 40-bit instructions would be required than the 32-bit MIPS instructions.

It's quite apparent that the successful elimination of redundancy from instructions makes instruction packing increasingly attractive. In an environment with long effective off-chip memory latency, the use of compact formats for instruction packing is an attractive approach.

4.3.7.5.7 Summary

In general, this experiment has shown that reducing redundancy in immediate fields and address fields of instructions leads to programs which are more compact; programs which don't necessarily execute faster, and may, in fact, run slower; and instruction formats which may be concatenated to form packed formats which can be expected to perform better than the packed MIPS format, using our definition of performance - useful operations per datapath cycle.

The above conclusions require qualification, of course. All of our system implementation assumptions directly affect our results. These include our particular choice of pipeline, our choice of a two-level memory hierarchy, our memory access to datapath delay ratio choices, etc. In addition to these, the characteristics of our simulation system also affect our results.

The availability of the Stanford software, which was provided to us by RCA, enabled us to conduct this experiment. However, this simulation system has limitations which influenced the results we just presented.

One problem which is somewhat significant is the use of the MIPS compiler on non-MIPS architectures. Any architecture designed to execute high level languages will only perform to its potential if the HLL compiler knows the characteristics of the architecture. One example illustrates this problem. The MIPS system routines (for multiply, etc.), consisting of both instructions and data, are stored in lower memory, and the MIPS compiler makes liberal use of the long immediate field available in MIPS load and store instructions to directly access data at these lower addresses. The frequent use of these long immediates penalizes our candidate instruction formats which have short immediate fields. However, a compiler for our candidate formats would know the immediate field limitations and utilize other addressing modes, such as "base + displacement." Therefore, the immediate cost for the shorter formats is likely overstated. However, the immediate cost values shown in Table 4.3 are not excessively large, so we suspect that only a minor variation from reality exists here.

A more serious problem, and one which introduces an error of unknown magnitude, is the use of small benchmarks as the workload for our cache simulator. Since our small benchmarks do not represent a workload that can drive a realistically-sized cache simulator, we were forced to rely on a technique for circumventing this problem, described in Section 4.3.7.1, which is an unproven one. Since any execution time improvement attributed to compact instruction formats is due to increased memory hit ratios, our unproven cache

simulation technique directly affects our execution time results.

Therefore, we can more comfortably state that our redundancy reduction has produced compact programs, and provided the basis for a packed instruction format which is expected to perform better than the MIPS format (by our definition of performance). Whether or not these compact instruction formats lead to faster or slower execution time than less compact formats is still undetermined, although it appears that there is no significant difference.

CHAPTER V
SUMMARY AND RECOMMENDATIONS

5.1 Summary

We have presented a computer system design methodology which we believe is advantageous for GaAs technology. We advocate a three-stage approach: (1) Study GaAs technology and packaging technology in order to determine their characteristics relevant for computer design. (2) Clearly define the effect that GaAs has on computer design, determine appropriate general design strategies, and suggest promising design solutions. (3) Perform experiments to establish the validity of earlier assumptions, and to provide an empirical foundation for further research.

5.2 Recommendations

Advances in both GaAs circuit technology and packaging technology will require a constant reevaluation of their impact on computer design. New design approaches may deserve consideration when new developments occur in GaAs and packaging technology to significantly alter their capabilities.

The most reliable techniques for evaluating the performance of GaAs computer systems are those based upon empirical data. A rational computer system design strategy demands that GaAs computer design decisions be based upon empirical data obtained either from simulation, or more preferably from actual GaAs implementations.

LIST OF REFERENCES

[AbKuL81] Abu-Sufah, W., Kuck, D.J., Lawrie, D.H., "On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations," *IEEE Transactions on Computers*, Vol. c-30, No. 5, May 1981, pp. 341-356.

[AleWo75] Alexander, W.G., Wortman, D.B., "Static and Dynamic Characteristics of XPL Programs," *IEEE Computer*, Vol. 8, Nol. 11, November 1975, pp. 41-46.

[AnEaG67] Anderson, S.F., Earle, J.G., Goldschmidt, R.E., Powers, D.M., "The IBM System/360 Model 91: Floating-Point Execution Unit", *IBM Journal of Research and Development*, Vol. 11, No. 1, January 1967, pp. 34-53.

[AsMiA83] Asbeck, P.M., Miller, D.L., Anderson, F.J., Eisen, F.H., "Emitter-Coupled Logic Circuits Implemented with Heterojunction Bipolar Transistors," *Proceedings of the GaAs IC Symposium*, Phoenix, Arizona, October 1983, pp. 170-173.

[AsMiA84] Asbeck, P.M., Miller, D.L., Anderson, R.J., Deming, R.N., Chen, R.T., Liechti, C.A., Eisen, F.H., "Application of Heterojunction Bipolar Transistors to High Speed, Small-Scale Digital Integrated Circuits," *Proceedings of the GaAs IC Symposium*, Boston, Massachusetts, October 1984, pp. 133-136.

[AusHo82] Auslander, M., Hopkins, M., "An Overview of the PL.8 Compiler," *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, Boston, Massachusetts, June 1982, pp. 22-31.

[Barne85] Barney, C., "DARPA Eyes 100-mips GaAs Chip for Star Wars," *ElectronicsWeek* , Vol. 58, No. 20, May 20, 1985, pp. 22-23.

[BasNu84] Bass, S., Neudeck, G., "VLSI Transistor Count and Basic Delays," *Internal Report*, Purdue University, 1984.

[BeDoF81] Beyers, J.W., Dohse, L.J., Fucetola, J.P., Kochis, R.L., Taylor, G.L., Zeller, E.R., "A 32-Bit VLSI CPU Chip," *IEEE Journal of Solid-State Circuits*, Vol. sc-16, No. 10, October 1981, pp. 537-541.

[Denni70] Denning, P.J., "Virtual Memory," *ACM Computing Surveys*, Vol. 2, No. 3, September 1970, pp. 62-97.

[Denni72] Denning, P.J., "On Modeling Program Behavior," *Proceedings of the Spring Joint Computer Conference*, 1972, pp. 937-944.

[Denni80] Dennis, J.B., "Data Flow Supercomputers," *IEEE Computer*, Vol. 13, No. 11, November 1980, pp. 48-56.

[Eden82] Eden, R.C., "Comparison of GaAs Device Approaches for Ultrahigh-Speed VLSI," *Proceedings of the IEEE*, Vol. 70, No. 1, January 1982, pp. 5-12.

[EdWeL83] Eden, R.C., Welch, B.M., Lee, F.S., "Implications and Projections of Gallium Arsenide Technology in High Speed Computing," *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, Port Chester, New York, October-November 1983, pp. 30-33.

[EdWeZ79] Eden, R.C., Welch, B.M., Zucca, R., Long, S.I., "The Prospects for Ultrahigh-Speed VLSI GaAs Digital Logic," *IEEE Journal of Solid- State Circuits*, Vol. sc-14, No. 2, April 1979, pp. 221-239.

[Furht85] Furht, B., "RISC Architectures with Multiple Overlapping Windows," *Proceedings of Midcon/85*, Chicago, Illinois, September 1985, pp. 23/2.1-23/2.10.

[FurMi85] Fura, D.A., Milutinović, V.M., "Computer Architecture Design in GaAs," *Proceedings of Midcon/85*, Chicago, Illinois, September 1985, pp. 24/3.1-24/3.7.

[FuTaI84] Furutsuka, T., Takahashi, K., Ishikawa, S., Yano, S., Higashisaka, A., "A GaAs 12 x 12 Bit Expandable Parallel Multiplier LSI Using Sidewall-Assisted Closely-Spaced Electrode Technology," *Proceedings of the International Electron Devices Meeting*, San Francisco, California, December 1984, pp. 344-347.

[Ghana83] Gnanasekaran, R., "On a Bit-Serial Input and Bit-Serial Output Multiplier," *IEEE Transactions on Computers*, Vol. c-32, No. 9, September 1983, pp. 878-880.

[Gheew84] Gheewala, T.R., "System Level Comparison of High Speed Technologies," *Proceedings of the IEEE 1984 International Conference on Computer Design*, Port Chester, New York, October 1984, pp. 245-250.

[GiGrH83] Gill, J., Gross, T., Hennessy, J., Jouppi, N., Przybylski, S., Rowen, C., "Summary of MIPS Instructions," *Technical Note No. 83-237*, Stanford University, November 1983.

[Gilbe84] Gilbert, B.K., "Design and Performance Trade Offs in the Use of Si VLSI and Gallium Arsenide in High Clockrate Signal Processing," *Proceedings of the IEEE 1984 International Conference on Computer Design*, Port Chester, New York, October 1984, pp. 260-266.

[Gross83] Gross, T., "Code Optimization of Pipeline Constraints," *Technical Report No. 83-255*, Stanford University, December 1983.

[Heage85] Heagerty, W., GaAs Seminar presented at Purdue University, January 1985.

[Hehne76] Hehner, E.C.R., "Computer Design to Minimize Memory Requirements," *IEEE Computer*, Vol. 9, No. 8, August 1976, pp. 65-70.

[HeScZ85] Helbig, W.A., Schellack, R.H., Zieger, R.M., "The Design and Construction of a GaAs Technology Demonstration Microprocessor," *Proceedings of Midcon/85*, Chicago, Illinois, September 1985, pp. 23/1.1-23/1.6.

[HeJoG82] Hennessy, J., Jouppi, N., Gill, J., Baskett, F., Strong, A., Gross, T., Rowen, C., Leonard, J., "The MIPS Machine," *Digest of Papers, Spring COMPCON 82*, San Francisco, California, February 1982, pp. 2-7.

[HeJoP83] Hennessy, J., Jouppi, N., Przybylski, S., Rowen, C., Gross, T., "Design of a High Performance VLSI Processor," *Technical Report No. 236*, Stanford University, February 1983.

[Huffm52] Huffman, D.A., "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the I.R.E.*, Vol. 40, No.9, September 1952, pp. 1098-1101.

[Hwang79] Hwang, K., *Computer Arithmetic: Principles, Architecture, and Design*, John Wiley & Sons, 1979.

[IsInI84] Ishii, Y., Ino, M., Idda, M., Hirayama, M., Ohmori, M., "Processing Technologies for GaAs Memory LSIs," *Proceedings of the GaAs IC Symposium*, Boston, Massachusetts, October 1984, pp. 121-124.

[Katev83] Katevenis, M.G.H., "Reduced Instruction Set Computer Architectures for VLSI," *Report No. UCB/CSD 83/141*, University of California at Berkeley, October 1983.

[KuMiS84] Kuroda, S., Mimura, T., Suzuki, M., Kobayashi, N., Nishiuchi, K., Shibatomi, A., Abe, M., "New Device Structure for 4Kb HEMT SRAM," *Proceedings of the GaAs IC Symposium*, Boston, Massachusetts, October 1984, pp. 125-128.

[Kung82] Kung, H.T., "Why Systolic Architectures?," *IEEE Computer*, Vol. 15, No. 1, January 1982, pp. 37-46.

[LeKaW82] Lee, F.S., Kaelin, G.R., Welch, B.M., Zucca, R., Shen, E., Asbeck, P., Lee, C.P., Kirkpatrick, C.G., Long, S.I., Eden, R.C., "A high-speed LSI GaAs 8x8-bit parallel multiplier," *IEEE Journal of Solid-State Circuits*, Vol. SC-17, No. 4, August 1982, pp. 638-647.

[Linn85] Linn, J., University of Southwestern Louisiana, Private Conversation, 1985.

[MaMoM84] MacGregor, D., Mothersole, D., Moyer, B., "The Motorola MC68020," *IEEE Micro*, Vol. 17, No. 8, August 1984, pp. 101-118.

[MaOhH84] Matsuoka, Y., Ohwada, K., Hirayama, M., "Uniformity Evaluation of MESFET's for GaAs LSI Fabrication," *IEEE Transactions on Electron Devices*, Vol. ed-31, No. 8, August 1984, pp. 1062-1067.

[McCaM82] McDonough, K., Caudel, E., Magar, S., Leigh, A., "Microcomputer with 32-bit arithmetic does high-precision number crunching," *Electronics*, Vol. 55, No. 8, February 24, 1982, pp. 105-110.

[McDan82] McDaniel, G., "An Analysis of a Mesa Instruction Set Using Dynamic Instruction Frequencies," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, March 1982, pp. 167-176.

[MiFuH86] Milutinović, V., Fura, D., Helbig, W., "An Introduction to GaAs Microprocessor Architecture for VLSI," *IEEE Computer*, Vol. 19, No. 3, March 1986, to appear.

[MiSiF86] Milutinović, V., Silbey, A., Fura, D., Bettinger, M., Keirn, K., Helbig, W., Heagerty, W., Zieger, R., "Design Issues in GaAs Computer Systems," *Proceedings of the 1986 Hawaii International Conference on System Science*, Honolulu, Hawaii, February 1986, to appear.

[Myers82] Myers, G.J., *Advances in Computer Architecture - Second Edition*, John Wiley & Sons, 1982.

[Namor84] Namordi, M.R., GaAs Seminar presented at Purdue University, October 1984.

[NaSuS83] Nakayama, Y., Suyama, K., Shimizu, H., Yokoyama, N., Ohnishi, H., Shibatomi, A., Ishikawa, H., "A GaAs 16 x 16 bit Parallel Multiplier," *IEEE Journal of Solid-State Circuits*, Vol. sc-18, No. 5, October 1983, pp. 599-603.

[NuKuM84] Nukiyama, T., Kusano, T., Matsumoto, K., Kurokawa, H., Hoshi, T., Goto, H., Temma, T., "A VLSI Image Pipeline Processor," *Proceedings of the 1984 IEEE International Solid-State Circuits Conference*, San Francisco, California, February 1984, pp. 208-209.

[NuPeB82] Nuzillat, G., Perea, E.H., Bert, G., Damay-Kavala, F., Gloanec, M., Peltier, M., Ngu, T.P., Arnodo, C., "GaAs MESFET IC's for Gigabit Logic Applications," *IEEE Journal of Solid-State Circuits*, Vol. sc-17, No. 3, June 1982, pp. 569-584

[PatPi82] Patterson, D.A., Piepho, R.S., "Assessing RISCs in High-Level Language Support," *IEEE MICRO*, Vol. 15, No. 11, November 1982, pp. 9-19.

[Patte85] Patterson, D.A., "Reduced Instruction Set Computers," *Communications of the ACM*, Vol. 28, No. 1, January 1985, pp. 8-21.

[Radin83] Radin, G., "The 801 Minicomputer," *IBM Journal of Research and Development*, Vol. 27, No. 3, May 1983, pp. 237-245.

[Rose85] Rose, C.D., "Speed Record Claimed for GaAs Transistor," *ElectronicsWeek*, Vol. 58, No. 19, May 13, 1985, pp. 19-20.

[Russe78] Russell, R.M., "The CRAY-1 Computer System," *Communications of the ACM*, Vol. 21, No. 1, January 1978, pp. 63-72.

[Sherb84] Sherburne, R.W. Jr., "Processor Design Tradeoffs in VLSI," *Report No. UCB/CSD 84/173*, University of California at Berkeley, April 1984.

[SmiGo83] Smith, J.E., Goodman, J.R., "A Study of Instruction Cache Organizations and Replacement Policies," *Proceedings of the Tenth Annual Symposium on Computer Architecture*, Stockholm, Sweden, June 1983, pp. 132-137.

[Smith78] Smith, A.J., "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer*, Vol. 11, No. 12, December 1978, pp. 7-21.

[Smith81] Smith, J.E., "A Study of Branch Prediction Strategies," *Proceedings of the Eighth Symposium on Computer Architecture*, May 1981, pp. 135-148.

[Smith82] Smith, A.J., "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, September 1982, pp. 473-530.

[Smith85] Smith, A.J., "Cache Evaluation and the Impact of Workload Choice," *Proceedings of the 12th Annual Symposium on Computer Architecture*, Boston, Massachusetts, June 1985, pp. 64-73.

[SolMo84] Solomon, P.M., Morkoc, H., "Modulation-Doped GaAs/AlGaAs Heterojunction Field-Effect Transistors (MODFET's), Ultrahigh-Speed Devices for Supercomputers," *IEEE Transactions on Electron Devices*, Vol. ed-31, No. 8, August 1984, pp. 1015-1027.

[Tanen78] Tanenbaum, A.S., "Implications of Structured Programming for Machine Architecture," *Communications of the ACM*, Vol. 21, No. 3, March 1978, pp. 237-246.

[ToUcK85] Toyoda, N., Uchitomi, N., Kitaura, Y., Mochizuki, M., Kanazawa, K., Terada, T., Ikawa, Y., Hojo, A., "A 42ps 2K-Gate GaAs Gate Array," *Proceedings of the 1985 IEEE International Solid-State Circuits Conference*, February 1985, pp. 206-207.

[VanLi74] Van Tuyl, R.L., Liechti, C.A., "High-Speed Integrated Logic with GaAs MESFET's," *IEEE Journal of Solid-State Circuits*, Vol. sc-9, No. 5, October 1974, pp. 269-276.

[VuRoN84] Vu, T.T., Roberts, P.C.T., Nelson, R.D., Lee, G.M., Hanzal, B.R., Lee, K.W., Zafar, N., Lamb, D.R., Helix, M.J., Jamison, S.A., Hanka, S.A., Brown, J.C. Jr., Shur, M.S., "A Gallium Arsenide SDFL Gate Array with On-Chip RAM," *IEEE Journal of Solid-State Circuits*, Vol. sc-19, No. 1, February 1984, pp. 10-22.

[Walle84] Waller, L., "GaAs ICs bid for commercial success," *Electronics*, Vol. 57, No. 12, June 14, 1984, pp. 101-102.

[Whitb85] Whitby-Stevens, C., "The Transputer," *Proceedings of the 12th Annual International Symposium on Computer Architecture*, Boston, Massachusetts, June 1985, pp. 292-300.

[YaHiA83] Yamamoto, R., Higashisaka, A., Asai, S., Tsuji, T., Takayama, Y., Yano, S., "Design and Fabrication of Depletion GaAs LSI High-Speed 32-Bit Adder," *IEEE Journal of Solid-State Circuits*, Vol. sc-18, No. 5, October 1983, pp. 592-599.

[YuMcS84] Yuan, H., McLevige, W.V., Shih, H.D., Hearn, A.S., "GaAs Heterojunction Bipolar 1K Gate Array," *Proceedings of the 1984 IEEE International Solid-State Circuits Conference*, San Francisco, California, February 1984, pp. 42-43.

Appendix A: Analytical Pipeline Performance Model Derivation

We can derive an equation to determine the execution time for each of our candidate pipelines for a given set of benchmarks. Again, we have selected three instruction pipelines and three memory configurations for this study, yielding nine candidate pipeline configurations. These are the normal Silicon pipeline in the (3,3), (3,6), and (6,3) memory configurations, the packed pipeline in the (3,3), (3,6), and (6,3) memory configurations, and the pipelined memory pipeline in the (3,3), (3,6), and (6,3) memory configurations.

Before we begin the derivation of the pipeline performance equations, it is helpful to better understand the need for removing some MIPS compiler effects from our benchmark programs.

A.1 Undoing the Effects of the Stanford MIPS Compiler

In our upcoming analysis we vary the parameters pbf and plf to determine their effect on performance. Whenever our value for pbf (plf) is not one, we are in effect simulating the effect of NOOPs following branch (load) instructions. However, the MIPS compiler also performed branch and load fillin, and when unsuccessful, inserted NOOPs behind branch and load instructions. Clearly, before we can accurately simulate the effects of particular values of pbf and plf, we must determine the number of NOOPs already introduced by the MIPS compiler, and remove this effect from our calculations.

There are two approaches for removing the compiler's branch fillin and load fillin effects. We now discuss these techniques in the context of branches, but this discussion is equally applicable to data loads.

The first approach that we can take is to effectively remove all the NOOPs introduced by the MIPS compiler from the benchmark programs by subtracting a value from our calculated execution time. This value which we subtract will be proportional to "nb * (1 - pbf0)," which is the total number of NOOPs due to unfilled branch fillin slots. After eliminating these NOOPs, we can then add the effect of our simulated NOOPs to the calculated execution time. The value that we add is proportional to "nb * (1 - pbf)."

The second way to approach this problem is to first unfill all the branch delay fillin slots by adding a value to our calculated execution time. This action can be thought of as inserting a NOOP between a branch operation and its filled slot, for all the successfully filled slots. The execution time to be added is proportional to "nb * pbf0," which is the total number of successfully filled branch fillin slots. This approach is attractive because for our GaAs processor, we will have branch delays greater than one. This creation of unfilled slots is easily extended to any branch delay. For example, if we want to create two more slots (for a total branch delay of three), we then add an additional time to our calculated execution time proportional to "nb * 2." After unfilling all the fillin slots, we can then perform our simulated fillin on these slots by subtracting from the calculated time a value proportional to "nb * bd * pbf," where bd is the branch delay of our candidate GaAs pipeline. It is this second approach which we use in our analysis to follow.

A.2 Execution Time Calculations

Once again, the parameters which are obtained from the benchmark programs:

ni: Total number of instructions.

nl: Total number of loads.

nb: Total number of branches.

np: Total number of packed instructions.

pbf0: Probability of branch fill achieved by the MIPS compiler.

plf0: Probability of load fill achieved by the MIPS compiler.

The parameters derivable from the memory configuration:

nih: Number of cycles required for an instruction cache hit.

nim: Number of cycles required for an instruction cache miss.

ndh: Number of cycles required for a data cache hit.

ndm: Number of cycles required for a data cache miss.

The parameters which are to be varied in the pipeline experiment.

pih: Probability of instruction cache hit.

pdh: Probability of data cache hit.

pbf: Probability of branch fill.

plf: Probability of load fill.

A.2.1 Assumptions

(1) In cases where there are only one or two operations to be executed and three or more datapath cycles available to execute them, the operations can be moved around among the slots. This will improve performance, for example when a prior data load doesn't finish before the first cycle of an instruction but does finish before the second, etc. An operation requiring the loaded value can be executed in the second slot instead of the first.

(2) For load fills we only perform load fillin on the assumption of a data cache hit. This means that on a data cache miss the processor must halt, and the load latency introduced by the cache miss must be absorbed.

A.2.2 Normal Silicon (3,3)

nih = 3, nim = 6, ndh = 3, ndm = 6

A.2.2.1 Instruction fetches:

delay = (number of instr. fetches)
* [(delay for cache hit) * (prob. of cache hit)
\+ (delay for cache miss) * (prob. of cache miss)]
= ni * [nih * pih + nim * (1 - pih)]
= **ni * (6 - 3 * pih)** (A.1)

A.2.2.2 Data loads:

For every load, we must first unfill the fillin slot, then fill it back up again using our simulation plf. The cost of an unfilled slot is the fetch of one NOOP.

delay = (number of data loads)
* [(cost of instr. fetch) * (MIPS unfill)
\- (cost of instr. fetch) * (GaAs processor fill)]
= nl * [(6 - 3 * pih) * plf0 - (6 - 3 * pih) * plf]
= **nl * (6 - 3 * pih) * (plf0 - plf)** (A.2)

A.2.2.3 Branches:

For every branch, we must first unfill the fillin slot, then fill it back up again using our simulation pbf. The cost of an unfilled slot is the fetch of one NOOP.

delay = (number of branches)
* [(cost of instr. fetch) * (MIPS unfill)
\- (cost of instr. fetch) * (GaAs processor fill)]
= nb * [(6 - 3 * pih) * pbf0 - (6 - 3 * pih) * pbf]
= **nb * (6 - 3 * pih) * (pbf0 - pbf)** (A.3)

A.2.2.4 Packed instructions:

For every packed MIPS instruction we must unpack it. The cost of a packed MIPS instruction is one instruction fetch. Note that the second operation (the packed piece) of a MIPS instruction is never a data load or branch - see Appendix B.

delay = (number of packs) * (cost of instr. fetch)
= **np * (6 - 3 * pih)** (A.4)

A.2.3 Normal Silicon (3,6)

nih = 3, nim = 6, ndh = 6, ndm = 6

A.2.3.1 Instruction fetches: [see (A.1)]

delay = **ni * (6 - 3 * pih)**

A.2.3.2 Data loads: [see (A.2)]

delay = **nl * (6 - 3 * pih) * (plf0 - plf)**

A.2.3.3 Branches: [see (A.3)]

delay = **nb * (6 - 3 * pih) * (pbf0 - pbf)**

A.2.3.4 Packed instructions: [see (A.4)]

delay = **np * (6 - 3 * pih)**

A.2.4 Normal Silicon (6,3)

nih = 6, nim = 6, ndh = 3, ndm = 6

A.2.4.1 Instruction fetches:

We always require nih cycles.

delay = ni * nih
= **ni * 6** (A.5)

A.2.4.2 Data loads:

We need to first unfill the fillin slot, and then fill it back up with probability one. Because of the long instruction fetch time the load delay is 0. The cost of an unfilled slot is one instruction fetch.

delay = (number of data loads)
* [(cost of instr. fetch) * (MIPS unfill)
- (cost of instr. fetch) * (GaAs processor fill)]
= nl * (6 * plf0 - 6 * 1)
= **nl * 6 * (plf0 - 1)** (A.6)

A.2.4.3 Branches:

delay = (number of branches)
* [(cost of instr. fetch) * (MIPS unfill)
- (cost of instr. fetch) * (GaAs processor fill)]
= nb * (6 * pbf0 - 6 * pbf)
= **nb * 6 * (pbf0 - pbf)** (A.7)

A.2.4.4 Packed instructions:

delay = (number of packs) * (cost of instr. fetch)
= **np * 6** (A.8)

A.2.5 Packed (3,3)

nih = 3, nim = 6, ndh = 3, ndm = 6

A.2.5.1 Instruction fetches: [see (A.1)]

delay = **ni * (6 - 3 * pih)**

A.2.5.2 Data loads: [see (A.2)]

delay = **nl * (6 - 3 * pih) * (plf0 - plf)**

A.2.5.3 Branches: [see (A.3)]

delay = **nb * (6 - 3 * pih) * (pbf0 - pbf)**

A.2.6 Packed (3,6)

nih = 3, nim = 6, ndh = 6, ndm = 6

A.2.6.1 Instruction fetches: [see (A.1)]

delay = **ni * (6 - 3 * pih)**

A.2.6.2 Data loads: [see (A.2)]

delay = **nl * (6 - 3 * pih) * (plf0 - plf)**

A.2.6.3 Branches: [see (A.3)]

delay = **nb * (6 - 3 * pih) * (pbf0 - pbf)**

A.2.7 Packed (6,3)

nih = 6, nim = 6, ndh = 3, ndm = 6

A.2.7.1 Instruction fetches: [see (A.5)]

delay = **ni * 6**

A.2.7.2 Data loads: [see (A.6)]

delay = **nl * 6 * (plf0 - 1)**

A.2.7.3 Branches: [see (A.7)]

delay = **nb * 6 * (pbf0 - pbf)**

A.2.8 Pipelined Memory (3,3)

nih = 1, nim = 4, ndh = 3, ndm = 6

A.2.8.1 Instruction fetches:

delay = (number of instr. fetches)
* [(delay for cache hit) * (prob. of cache hit)
+ (delay for cache miss) * (prob. of cache miss)]
= ni * [nih * pih + nim * (1 - pih)]
= **ni * (4 - 3 * pih)** **(A.9)**

A.2.8.2 Data loads:

There are two causes of unpleasantness here: load fillins and data cache misses, but their effects are independent. For a data cache miss we must always add an additional three cycles. For the fillin problem we must first undo the MIPS fillin (which is for one slot only), then create two new unfilled slots in order to achieve the desired load delay of three. Finally, we must perform the simulation fillin on all three slots. The cost of an unfilled slot is one instruction fetch.

delay = (number of data loads)
* [(data cache miss delay) * (prob. of data cache miss)
+ (cost of instr. fetch) * (MIPS unfill)
+ (cost of instr. fetch) * (two new unfilled slots)
- (cost of instr. fetch) * (GaAs processor fill)]
= nl
* [3 * (1 - pdh)
+ (4 - 3 * pih) * plf0
+ (4 - 3 * pih) * 2
- (4 - 3 * pih) * 3 * plf]
= **nl * [3 * (1 - pdh)**
+ (4 - 3 * pih) * (2 + plf0 - 3 * plf)] (A.10)

A.2.8.3 Branches:

We need to first undo the MIPS fillin, then create two new unfilled slots in order to achieve the desired branch delay of three. Finally, we perform the simulation fillin on all three slots. The cost of an unfilled slot is one instruction fetch.

delay = (number of branches)
* [(cost of instr. fetch) * (MIPS unfill)
+ (cost of instr. fetch) * (two new unfilled slots)
- (cost of instr. fetch) * (GaAs processor fill)]
= nb
* [(4 - 3 * pih) * pbf0
+ (4 - 3 * pih) * 2
- (4 - 3 * pih) * 3 * pbf]
= **nb * (4 - 3 * pih) * (2 + pbf0 - 3 * pbf)** (A.11)

A.2.8.4 Packed instructions:

delay = (number of packs) * (cost of instr. fetch)
= **np * (4 - 3 * pih)** (A.12)

A.2.9 Pipelined Memory (3,6)

nih = 1, nim = 4, ndh = 6, ndm = 6

A.2.9.1 Instruction fetches: [see (A.9)]

delay = **ni * (4 - 3 * pih)**

A.2.9.2 Data loads:

We need to first undo the MIPS fillin and create five new unfilled slots. Then we perform simulation fillin on all six slots. The cost of an unfilled slot is one instruction fetch.

delay = (number of loads)
* [(cost of instr. fetch) * (SU-MIPS unfill)
+ (cost of instr. fetch) * (five new unfilled slots)
- (cost of instr. fetch) * (GaAs processor fillin)]
= nl
* [(4 - 3 * pih) * plf0
+ (4 - 3 * pih) * 5
- (4 - 3 * pih) * 6 * plf]
= **nl * (4 - 3 * pih) * (5 + plf0 - 6 * plf)** (A.13)

A.2.9.3 Branches: [see (A.11)]

delay = **nb * (4 - 3 * pih) * (2 + pbf0 - 3 * pbf)**

A.2.9.4 Packed instructions: [see (A.12)]

delay = **np * (4 - 3 * pih)**

A.2.10 Pipelined Memory (6,3)

nih = 1, nim = 1, ndh = 3, ndm = 6

A.2.10.1 Instruction fetches:

We always require one cycle.

delay = **ni * 1** (A.14)

A.2.10.2 Data loads:

delay = (number of data loads)
* [(data cache miss delay) * (prob. of data cache miss)
+ (cost of instr. fetch) * (MIPS unfill)
+ (cost of instr. fetch) * (two new unfilled slots)
- (cost of instr. fetch) * (GaAs processor fill)]
= nl
* [3 * (1 - pdh)
+ 1 * plf0
+ 1 * 2
- 1 * 3 * plf]
= **nl * (5 + plf0 - 3 * plf - 3 * pdh)**

A.2.10.3 Branches:

We need to first undo the MIPS fillin and create five new unfilled slots. Then we perform simulation fillin on all six slots. The cost of and unfilled slot is one instruction fetch.

delay = (number of branches)
* [(cost of instr. fetch) * (MIPS unfill)
+ (cost of instr. fetch) * (five new unfilled slots)
- (cost of instr. fetch) * (GaAs processor fillin)]
= nb
* (1 * pbf0
+ 1 * 5
- 1 * 6 * pbf)
= **nb * (5 + pbf0 - 6 * pbf)**

A.2.10.4 Packed instructions:

delay = (number of packs) * (cost of instr. fetch)
= **np * 1**

Appendix B: Determination of Candidate Instruction Format Costs

In this appendix we present the costs associated with each candidate instruction format, and describe our methodology for determining these costs.

Once again, the three costs which a candidate format may experience are:

(1) Packing cost. Some MIPS instructions are successfully packed and hence, contain two operations. None of the candidate formats can execute two operations from a single instruction fetch.

(2) Address cost. MIPS instructions contain operations with as many as three address specifications. Some of the candidate formats have only two register address fields, while some have only two address fields, one of which may be an immediate field.

(3) Immediate cost. MIPS instructions have immediate fields containing as many as 24 bits. None of the candidate formats have immediate fields that long, and most immediate fields are much shorter.

B.1 Cost Determination

Packing costs are determined simply by examining MIPS instructions which are packable, and determining if the packed operation is a NOOP or not. The cost penalty is one instruction if the packed operation is not a NOOP. Because only 12 bits of a MIPS instruction are allocated to the packed operation, it must be a 2-address ALU operation. For our candidate formats, the packed operation cannot yield an address or immediate cost.

Address costs are determined by examining the address (register or immediate) needs of MIPS instructions and determining if the candidate format has enough of the right kind of address fields. The cost penalty is one instruction if the candidate format lacks the necessary fields.

Immediate costs are determined by examining the immediate values of MIPS instructions and determining if the candidate format has enough bits to represent them. The cost penalty is one instruction if the candidate format lacks the necessary immediate field length. We assume that the entire second

word may be used for immediate if necessary. For some candidate formats, this yields a maximum immediate field size of 20 bits while some MIPS instructions contain 24-bit immediate fields. However, the MIPS 24-bit immediate fields are mainly used for addressing, and since the MIPS instructions and data are all loaded into memory addresses below 32,000, we should not see immediate fields of over 20 bits for addressing in this study. The only other MIPS instruction which uses over 20 bits of immediate field is the "load immediate" instruction; but we found this instruction to represent well below one percent of all instruction executions, so we can safely ignore it. Therefore, a maximum penalty of one is sufficient.

B.2 Description of the MIPS Instruction Set

The MIPS instruction set contains 28 instructions which we now briefly describe. We define the abbreviations that are used in Section B.3 and list the characteristics of the relevant fields of the MIPS instructions. This information is taken directly from a Stanford Technical Note [GiGrH83].

(AA) ALU3 + ALU2. Contains two registers, one register/4-bit immediate field, and one packed operation.

(BC) Branch Conditionally. Contains one register, one register/4-bit immediate field, one 4-bit condition field, and one 12-bit immediate field.

(BU) Branch Unconditionally. Contains one 24-bit immediate field.

(JB) Jump Based. Contains one register and one 20-bit immediate field.

(JBI) Jump Based Indirect. Contains one register and one 20-bit immediate field.

(JBIA) Jump Based Indirect + ALU2. Contains one register, one 8-bit immediate field, and one packed operation.

(JD) Jump Direct. Contains one 24-bit immediate field.

(JI) Jump Indirect. Contains one 24-bit immediate field.

(JISS) Jump Indirect and Setup Su-register. Contains one 24-bit immediate field.

(LB) Load Based. Contains one register and one 20-bit immediate field.

(LBA) Load Based + ALU2. Contains one register, one 8-bit immediate field, and one packed operation.

(LBIA) Load Base-Indexed + ALU2. Contains three registers and one packed operation.

(LBSA) Load Base-Shifted + ALU2. Contains two registers, one 4-bit immediate field, and one packed operation.

(LD) Load Direct. Contains one register and one 24-bit immediate field.

(LI) Load Immediate. Contains one register and one 24-bit immediate field.

(LSD) LoadS Direct. Contains one 24-bit immediate field.

(SPC) SavePC. Contains one 24-bit immediate field.

(SC) Set Conditionally. Contains one register, one register/4-bit immediate field, and one 4-bit condition field.

(SB) Store Based. Contains two registers and one 20-bit immediate field.

(SBA) Store Based + ALU2. Contains two registers, one 8-bit immediate field, and one packed operation.

(SBIA) Store Base-Indexed + ALU2. Contains three registers and one packed operation.

(SBSA) Store Base-Shifted + ALU2. Contains two registers, one 4-bit immediate field, and one packed operation.

(SD) Store Direct. Contains one register and one 24-bit immediate field.

(SPCB) StorePC Based. Contains one register and one 20-bit immediate field.

(SPCBA) StorePC Based + ALU2. Contains one register, one 8-bit immediate field, and one packed operation.

(SPCD) StorePC Direct. Contains one 24-bit immediate field.

(SSUD) StoreSU Direct. Contains one 24-bit immediate field.

(TC) Trap Conditionally. Contains one register, one register/4-bit immediate field, one 4-bit condition field, and one 11-bit code field.

B.3 Presentation of Costs

In Table 4.2 we listed the fields for the candidate instruction formats. Based upon this table and Sections B.1 and B.2 above, we can derive the costs for each candidate format. We list these costs in Tables B.1 - B.10. We use several shorthand notations:

(1) We say "+1 over x" to indicate a cost of one instruction if a MIPS immediate value requiring more than x bits is encountered.

(2) We say "+1 if packed" to indicate a cost of one instruction if a packed MIPS instruction is encountered.

(3) We say "+1 if xreg" to indicate a cost of one instruction if a MIPS instruction requires x or more *distinct* registers.

(4) We say "+1 if xaddr" to indicate a cost of one instruction if a MIPS instruction requires x or more *distinct* addresses (register or immediate operands).

(5) We say "+1" to indicate a cost of one instruction always.

Table B.1 Costs for Candidate Format 28(3).

MIPS Instr.	Cost	MIPS Instr.	Cost
AA	+1 if packed	LI	+1 over 8
BC	+1	LSD	+1 over 8
	+1 over 8	SPC	+1 over 8
BU	+1 over 8	SC	+0
JB	+1 over 8	SB	+1 over 8
JBI	+1 over 8	SBA	+1 if packed
JBIA	+1 if packed	SBIA	+1 if packed
JD	+1 over 8	SBSA	+1 if packed
JI	+1 over 8	SD	+1 over 8
JISS	+1 over 8	SPCB	+1 over 8
LB	+1 over 8	SPCBA	+1 if packed
LBA	+1 if packed	SPCD	+1 over 8
LBIA	+1 if packed	SSUD	+1 over 8
LBSA	+1 if packed	TC	+1
LD	+1 over 8		+1 over 8

Table B.2 Costs for Candidate Format 28(3210).

MIPS Instr.	Cost	MIPS Instr.	Cost
AA	+1 if packed	LI	+1 over 16
BC	+1	LSD	+1 over 20
BU	+1 over 20	SPC	+1 over 20
JB	+1 over 16	SC	+0
JBI	+1 over 16	SB	+1 over 12
JBIA	+1 if packed	SBA	+1 if packed
JD	+1 over 20	SBIA	+1 if packed
JI	+1 over 20	SBSA	+1 if packed
JISS	+1 over 20	SD	+1 over 16
LB	+1 over 12	SPCB	+1 over 16
LBA	+1 if packed	SPCBA	+1 if packed
LBIA	+1 if packed	SPCD	+1 over 20
LBSA	+1 if packed	SSUD	+1 over 20
LD	+1 over 16	TC	+1

Table B.3 Costs for Candidate Format 24(32).

MIPS Instr.	Cost	MIPS Instr.	Cost
AA	+1 if packed	LI	+1 over 4
BC	+1	LSD	+1 over 4
	+1 over 4	SPC	+1 over 4
BU	+1 over 4	SC	+0
JB	+1 over 4	SB	+1 over 4
JBI	+1 over 4	SBA	+1 over 4
JBIA	+1 if over 4		+1 if packed
	+1 if packed	SBIA	+1 if packed
JD	+1 over 4	SBSA	+1 if packed
JI	+1 over 4	SD	+1 over 4
JISS	+1 over 4	SPCB	+1 over 4
LB	+1 over 4	SPCBA	+1 over 4
LBA	+1 over 4		+1 if packed
	+1 if packed	SPCD	+1 over 4
LBIA	+1 if packed	SSUD	+1 over 4
LBSA	+1 if packed	TC	+1
LD	+1 over 4		+1 over 4

Table B.4 Costs for Candidate Format 24(3210).

MIPS Instr.	Cost	MIPS Instr.	Cost
AA	+1 if packed	LI	+1 over 12
BC	+1	LSD	+1 over 16
BU	+1 over 16	SPC	+1 over 16
JB	+1 over 12	SC	+0
JBI	+1 over 12	SB	+1 over 8
JBIA	+1 if packed	SBA	+1 if packed
JD	+1 over 16	SBIA	+1 if packed
JI	+1 over 16	SBSA	+1 if packed
JISS	+1 over 16	SD	+1 over 12
LB	+1 over 8	SPCB	+1 over 12
LBA	+1 if packed	SPCBA	+1 if packed
LBIA	+1 if packed	SPCD	+1 over 16
LBSA	+1 if packed	SSUD	+1 over 16
LD	+1 over 12	TC	+1

Table B.5 Costs for Candidate Format 24(2).

MIPS Instr.	Cost	MIPS Instr.	Cost
AA	+1 if 3reg	LI	+1 over 8
	+1 if packed	LSD	+1 over 8
BC	+1	SPC	+1 over 8
	+1 over 8	SC	+0
BU	+1 over 8	SB	+1 over 8
JB	+1 over 8	SBA	+1 if packed
JBI	+1 over 8	SBIA	+1 if 3reg
JBIA	+1 if packed		+1 if packed
JD	+1 over 8	SBSA	+1 if packed
JI	+1 over 8	SD	+1 over 8
JISS	+1 over 8	SPCB	+1 over 8
LB	+1 over 8	SPCBA	+1 if packed
LBA	+1 if packed	SPCD	+1 over 8
LBIA	+1 if 3reg	SSUD	+1 over 8
	+1 if packed	TC	+1
LBSA	+1 if packed		+1 over 8
LD	+1 over 8		

Table B.6 Costs for Candidate Format 24(210).

MIPS Instr.	Cost	MIPS Instr.	Cost
AA	+1 if 3reg	LI	+1 over 12
	+1 if packed	LSD	+1 over 16
BC	+1	SPC	+1 over 16
BU	+1 over 16	SC	+0
JB	+1 over 12	SB	+1 over 8
JBI	+1 over 12	SBA	+1 if packed
JBIA	+1 if packed	SBIA	+1 if 3reg
JD	+1 over 16		+1 if packed
JI	+1 over 16	SBSA	+1 if packed
JISS	+1 over 16	SD	+1 over 12
LB	+1 over 8	SPCB	+1 over 12
LBA	+1 if packed	SPCBA	+1 if packed
LBIA	+1 if 3reg	SPCD	+1 over 16
	+1 if packed	SSUD	+1 over 16
LBSA	+1 if packed	TC	+1
LD	+1 over 12		

Table B.7 Costs for Candidate Format 20(32).

MIPS Instr.	Cost	MIPS Instr.	Cost
AA	+1 if packed	LI	+1 over 4
BC	+1	LSD	+1 over 4
	+1 over 4	SPC	+1 over 4
BU	+1 over 4	SC	+0
JB	+1 over 4	SB	+1 over 4
JBI	+1 over 4	SBA	+1 over 4
JBIA	+1 over 4		+1 if packed
	+1 if packed	SBIA	+1 if packed
JD	+1 over 4	SBSA	+1 if packed
JI	+1 over 4	SD	+1 over 4
JISS	+1 over 4	SPCB	+1 over 4
LB	+1 over 4	SPCBA	+1 over 4
LBA	+1 over 4		+1 if packed
	+1 if packed	SPCD	+1 over 4
LBIA	+1 if packed	SSUD	+1 over 4
LBSA	+1 if packed	TC	+1
LD	+1 over 4		+1 over 4

Table B.8 Costs for Candidate Format 20(3210).

MIPS Instr.	Cost	MIPS Instr.	Cost
AA	+1 if packed	LI	+1 over 8
BC	+1	LSD	+1 over 12
BU	+1 over 12	SPC	+1 over 12
JB	+1 over 8	SC	+0
JBI	+1 over 8	SB	+1 over 4
JBIA	+1 if packed	SBA	+1 if (2reg
JD	+1 over 12		and over 4)
JI	+1 over 12		+1 if packed
JISS	+1 over 12	SBIA	+1 if packed
LB	+1 over 4	SBSA	+1 if packed
LBA	+1 if (2reg	SD	+1 over 8
	and over 4)	SPCB	+1 over 8
	+1 if packed	SPCBA	+1 if packed
LBIA	+1 if packed	SPCD	+1 over 12
LBSA	+1 if packed	SSUD	+1 over 12
LD	+1 over 8	TC	+1

Table B.9 Costs for Candidate Format 16(21).

MIPS Instr.	Cost	MIPS Instr.	Cost
AA	+1 if 3addr	LI	+1 over 4
	+1 if packed	LSD	+1 over 4
BC	+1	SPC	+1 over 4
	+1 if 2addr	SC	+1 if 3addr
	+1 over 4	SB	+1 if (3addr
BU	+1 over 4		or over 4)
JB	+1 over 4	SBA	+1 if (3addr
JBI	+1 over 4		or over 4)
JBIA	+1 over 4		+1 if packed
	+1 if packed	SBIA	+1 if 3addr
JD	+1 over 4		+1 if packed
JI	+1 over 4	SBSA	+1 if 3addr
JISS	+1 over 4		+1 if packed
LB	+1 if (3addr	SD	+1 over 4
	or over 4)	SPCB	+1 over 4
LBA	+1 if (3addr	SPCBA	+1 over 4
	or over 4)		+1 if packed
	+1 if packed	SPCD	+1 over 4
LBIA	+1 if 3addr	SSUD	+1 over 4
	+1 if packed	TC	+1
LBSA	+1 if 3addr		+1 if 2addr
	+1 if packed		+1 over 4
LD	+1 over 4		

Table B.10 Costs for Candidate Format 16(210).

MIPS Instr.	Cost	MIPS Instr.	Cost
AA	+1 if 3addr	LI	+1 over 4
	+1 if packed	LSD	+1 over 8
BC	+1	SPC	+1 over 8
	+1 if 2addr	SC	+1
	+1 over 8	SB	+1 if (3addr
BU	+1 over 8		or over 4)
JB	+1 over 4	SBA	+1 if (3addr
JBI	+1 over 4		or over 4)
JBIA	+1 over 4		+1 if packed
	+1 if packed	SBIA	+1 if 3addr
JD	+1 over 8		+1 if packed
JI	+1 over 8	SBSA	+1 if 3addr
JISS	+1 over 8		+1 if packed
LB	+1 if (3addr	SD	+1 over 4
	or over 4)	SPCB	+1 over 4
LBA	+1 if (3addr	SPCBA	+1 over 4
	or over 4)		+1 if packed
	+1 if packed	SPCD	+1 over 8
LBIA	+1 if 3addr	SSUD	+1 over 8
	+1 if packed	TC	+1
LBSA	+1 if 3addr		+1 if 2addr
	+1 if packed		+1 if over 8
LD	+1 over 4		