

---

# **THE MIPS-X RISC MICROPROCESSOR**

edited by

**Paul Chow**

University of Toronto



SPRINGER SCIENCE+BUSINESS MEDIA, LLC

**Library of Congress Cataloging-in-Publication Data**

The MIPS-X RISC microprocessor / edited by Paul Chow.

p. cm. — (Kluwer international series in engineering and computer science. VLSI, computer architecture, and digital signal processing)

1. MIPS-X (Microprocessor) I. Chow, Paul, 1955- . II. Series.  
QA76.8.M524M57 1989  
004.165—dc20

89-19810  
CIP

---

ISBN 978-1-4419-5119-9

ISBN 978-1-4757-6762-9 (eBook)

DOI 10.1007/978-1-4757-6762-9

**Copyright** © 1989 by Springer Science+Business Media New York  
Originally published by Kluwer Academic Publishers in 1989

Second Printing, 1990.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher, Springer Science+Business Media, LLC.

**To Yu-Ling**

# Contents

<b>List of Figures</b>	<b>xi</b>
<b>List of Tables</b>	<b>xv</b>
<b>Contributors</b>	<b>xxvii</b>
<b>Foreword</b> <i>by John Hennessy</i>	<b>xix</b>
<b>Preface</b>	<b>xxi</b>
<b>Acknowledgments</b>	<b>xxiii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Architecture</b>	<b>7</b>
2.1 MIPS-X Overview . . . . .	7
2.1.1 Memory Organization . . . . .	10
2.1.2 Caches . . . . .	11
2.1.3 Pipeline Organization . . . . .	12
2.1.4 The Processor Status Word . . . . .	13
2.1.5 Exception Handling . . . . .	15
2.2 Hardware Design Tradeoffs . . . . .	16
2.2.1 Coprocessor Interface . . . . .	16
2.2.2 Branch Mechanism . . . . .	17
2.2.3 Shift Operations . . . . .	18
2.2.4 Exception Handling . . . . .	18
<b>3 The Compiler System</b>	<b>21</b>
3.1 Overview . . . . .	21
3.2 Interprocedural Analyzer . . . . .	23
3.3 Global Optimizer . . . . .	24

3.4	LISP Front-end . . . . .	25
3.5	Code Generator . . . . .	25
3.6	Reorganizer . . . . .	26
3.6.1	Local Reorganization . . . . .	26
3.6.2	Interblock Reorganization . . . . .	28
3.6.3	Branch Optimization . . . . .	29
3.7	Machine Simulator . . . . .	30
<b>4</b>	<b>A Hardware Overview</b>	<b>33</b>
4.1	The Hardware Resources . . . . .	33
4.2	Clocking . . . . .	34
4.3	General Instruction Flow . . . . .	36
4.4	Control Flow . . . . .	38
4.5	Implementation Issues . . . . .	39
4.5.1	The Naming Convention . . . . .	39
4.5.2	Datapath Design . . . . .	41
<b>5</b>	<b>The Execute Engine</b>	<b>49</b>
5.1	The Register File . . . . .	49
5.1.1	Register Array . . . . .	50
5.1.2	Bypass Registers and Control . . . . .	56
5.1.3	Datapath . . . . .	57
5.2	The Execute Unit . . . . .	60
5.2.1	Interface . . . . .	61
5.2.2	Timing . . . . .	65
5.2.3	PLA Controllers . . . . .	66
5.2.4	Funnel Shifter . . . . .	66
5.2.5	Funnel Shifter Controller . . . . .	69
5.2.6	MD Register . . . . .	71
5.2.7	MD Register Controller . . . . .	73
5.2.8	ALU . . . . .	76
5.2.9	ALU Controller . . . . .	80
5.2.10	PSW . . . . .	83
5.2.11	PSW Controller . . . . .	85
5.2.12	Special Circuitry . . . . .	89
<b>6</b>	<b>Instruction Fetch Hardware</b>	<b>93</b>
6.1	The PC Unit . . . . .	94
6.1.1	PC Unit Interface . . . . .	95
6.1.2	Timing . . . . .	98
6.1.3	Signal Derivations and Origins . . . . .	99

6.1.4	The System/User Bit . . . . .	106
6.1.5	The ALU Drivers . . . . .	106
6.1.6	The Displacement Unit . . . . .	106
6.1.7	The Incrementer . . . . .	108
6.1.8	The PC Chain . . . . .	109
6.1.9	The Cache Miss Finite State Machine . . . . .	111
6.1.10	The Squash Finite State Machine . . . . .	111
6.2	Internal Cache Tags . . . . .	112
6.2.1	Internal Cache Description . . . . .	115
6.2.2	Tags Interface . . . . .	116
6.2.3	Timing . . . . .	119
6.2.4	Tag Store . . . . .	122
6.2.5	Valid Bit Store . . . . .	127
6.2.6	Ring Counter . . . . .	128
6.2.7	Address Driver . . . . .	132
6.2.8	Tag Control . . . . .	133
6.3	The Instruction Cache RAM . . . . .	137
6.3.1	Block Description . . . . .	138
6.3.2	The RAM Cell . . . . .	139
6.3.3	The Cell Arrays . . . . .	143
6.3.4	Sense Amplifiers . . . . .	146
6.3.5	Read/Write Circuits . . . . .	147
6.3.6	Tag Drivers . . . . .	147
6.3.7	Address Decoding . . . . .	148
6.4	Instruction Register . . . . .	151
6.4.1	Overview . . . . .	151
6.4.2	Register Array . . . . .	151
6.4.3	Decoder Array . . . . .	152
6.4.4	Instruction Cache Data Bus Interface . . . . .	156
6.4.5	No Destination Value Logic . . . . .	156
<b>7</b>	<b>The External Interface</b>	<b>161</b>
7.1	The System Builder's View . . . . .	161
7.1.1	Signal Description . . . . .	163
7.1.2	Signal Timings . . . . .	166
7.1.3	MIPS-X Clocks . . . . .	167
7.1.4	Pad Loading . . . . .	167
7.1.5	Memory References . . . . .	167
7.1.6	ICache Misses . . . . .	175
7.1.7	Coprocessors . . . . .	177
7.1.8	Floating-Point Unit . . . . .	184

7.1.9	Exceptions . . . . .	184
7.1.10	Reset . . . . .	188
7.1.11	Instruction Cache Testing . . . . .	188
7.1.12	Datapath Testing . . . . .	189
7.2	The External Interface Hardware . . . . .	190
7.2.1	Outgoing Signals . . . . .	190
7.2.2	Incoming Signals . . . . .	192
7.2.3	Test Pins . . . . .	192
<b>A</b>	<b>Exception Handling</b>	<b>193</b>
A.1	Interrupts . . . . .	193
A.2	Trap On Overflow . . . . .	196
A.3	Trap Instructions . . . . .	196
<b>B</b>	<b>Integer Multiplication and Division</b>	<b>199</b>
B.1	Multiplication and Division Support . . . . .	199
B.2	Multiplication . . . . .	200
B.3	Division . . . . .	203
<b>C</b>	<b>Opcode Map</b>	<b>207</b>
C.1	OP Field Bit Assignments . . . . .	207
C.2	Comp Func Field Bit Assignments . . . . .	208
C.3	Opcode Map of All Instructions . . . . .	210
<b>D</b>	<b>MIPS-X Revision 1 and 2 Pin Numbers</b>	<b>213</b>
D.1	Pin Mapping for Probe Card and Funsim . . . . .	215
D.2	Pin Map for 144 Pin PGA . . . . .	218
<b>E</b>	<b>Revision 1 and Revision 2 Differences</b>	<b>221</b>
	<b>Index</b>	<b>228</b>

# List of Figures

1.1	A chronology of MIPS-X. . . . .	4
2.1	The MIPS-X instruction formats. . . . .	8
2.2	Word numbering in memory. . . . .	11
2.3	Bit and byte numbering in a word. . . . .	11
2.4	The Processor Status Word. . . . .	15
3.1	The MIPS-X compiler. . . . .	22
3.2	Local reorganization algorithm. . . . .	27
3.3	Local reorganization example. . . . .	28
3.4	Interblock reorganization example. . . . .	29
3.5	Delayed branch scheduling. . . . .	30
3.6	Squashed branch scheduling. . . . .	31
3.7	Profile assisted optimization of a usually not-taken branch. . . . .	31
4.1	The MIPS-X die photo. . . . .	35
4.2	The MIPS-X hardware resources. . . . .	36
4.3	Datapath latch cell, and cell combined with flipped version. . . . .	42
4.4	Schematic of datapath latch cell. . . . .	44
4.5	Schematic of Result Bus driver. . . . .	45
4.6	Layout of Result Bus driver. . . . .	45
4.7	Schematic of three-state bus driver. . . . .	46
4.8	Layout of three-state bus driver. . . . .	47
4.9	Layout of 2 <i>pf</i> and 1 <i>pf</i> AND control line drivers. . . . .	48
5.1	Register File block diagram. . . . .	51
5.2	Register cell. . . . .	52
5.3	Register decoder. . . . .	53
5.4	Self-timed write circuitry. . . . .	53
5.5	Register word-line driver. . . . .	55
5.6	Register File bit-line driver. . . . .	55
5.7	Bypass comparators. . . . .	57



5.8	Bypass comparator logic. . . . .	58
5.9	Overall schematic of Register File datapath logic. . . . .	59
5.10	Interface to the Execute Unit. . . . .	65
5.11	PLA schematic. . . . .	67
5.12	Shifter schematic. . . . .	68
5.13	MD register schematic. . . . .	72
5.14	MD register action during an exception. . . . .	73
5.15	MD register action during instruction squashing. . . . .	74
5.16	Input multiplexer for the ALU. . . . .	77
5.17	The ALU $P$ block. . . . .	78
5.18	The ALU $G$ block. . . . .	79
5.19	Generating group $P$ signals. . . . .	79
5.20	Generating an Equal output. . . . .	80
5.21	The ALU carry path. . . . .	81
5.22	The ALU sum output. . . . .	81
5.23	Result Bus control signals generated by the ALU controller. . . . .	84
5.24	Schematic of a PSWcurrent-PSWother bit pair. . . . .	86
5.25	Schematic of the PSW $e$ bit. . . . .	87
5.26	Schematic of the TakeBranch logic. . . . .	90
5.27	Schematic of the overflow logic. . . . .	91
5.28	System/User space circuitry on the Result Bus. . . . .	92
6.1	PC Unit interface. . . . .	96
6.2	PC Displacement Unit. . . . .	107
6.3	PC Incrementer Unit. . . . .	109
6.4	Schematic of the PC Chain. . . . .	110
6.5	State diagram for Cache Miss FSM. . . . .	112
6.6	Schematic for Cache Miss FSM. . . . .	113
6.7	Squash FSM State Diagram. . . . .	114
6.8	Schematic for the Squash FSM. . . . .	114
6.9	Internal Cache Organization. . . . .	116
6.10	Interface to the Tags Unit. . . . .	118
6.11	ICache timing with no internal cache misses. . . . .	119
6.12	ICache timing during internal cache misses. . . . .	120
6.13	Tag Driver cell. . . . .	123
6.14	A Tag cell. . . . .	124
6.15	Layout for 2 adjacent Tag cells. . . . .	125
6.16	Reset Tag cell. . . . .	126
6.17	Tag Done circuit. . . . .	126
6.18	Valid Bit cell. . . . .	128
6.19	Layout for 4 adjacent valid bit cells. . . . .	129

6.20	Ring Counter. . . . .	130
6.21	Flow Through Latch with weak feedback. . . . .	131
6.22	Address Driver. . . . .	132
6.23	Shifting of the PC Bus onto the Address Bus. . . . .	133
6.24	Schematic of one block of the tags, including the tagDone circuitry. . . . .	135
6.25	Generation of the hit/miss signal. . . . .	136
6.26	The RAM timing diagram. . . . .	139
6.27	The RAM block diagram. . . . .	140
6.28	A single path through the RAM. . . . .	141
6.29	The RAM cell schematic. . . . .	141
6.30	The RAM cell layout. . . . .	142
6.31	Four RAM cells and RAM cell boundary. . . . .	144
6.32	Memory array organization and bit select. . . . .	145
6.33	Bit-line precharge. . . . .	146
6.34	The sense amplifiers. . . . .	147
6.35	Sense amplifiers and write drivers. . . . .	148
6.36	Address decoding block diagram. . . . .	149
6.37	Flow-through latch in address decode circuitry. . . . .	149
6.38	Predecode stage in address decode circuitry. . . . .	150
6.39	Final address decode stage and word-line driver. . . . .	150
6.40	Op and Type fields. . . . .	153
6.41	Source, Destination, and Immediate fields. . . . .	154
6.42	Sample decoders. . . . .	155
6.43	Instruction Cache interface. . . . .	158
6.44	ICache interface driver. . . . .	159
6.45	No Destination Value schematic. . . . .	160
7.1	MIPS-X and coprocessors. . . . .	162
7.2	MIPS-X systems on a bus. . . . .	163
7.3	Functional signal diagram for MIPS-X. . . . .	164
7.4	Ecache interface. . . . .	169
7.5	Timing diagram for a load instruction. . . . .	171
7.6	Timing diagram for Ecache miss during consecutive loads. . . . .	172
7.7	Timing diagram for a store instruction. . . . .	173
7.8	Timing diagram for sequence of memory instructions. . . . .	174
7.9	ICache miss timing. . . . .	175
7.10	Timing diagram for an ICache miss. . . . .	178
7.11	Timing diagram for coprocessor instructions. . . . .	181
7.12	Timing diagram of the <i>WBEEnable</i> signal for coprocessors. . . . .	183
7.13	Timing diagram for floating-point operations. . . . .	185

7.14	Interrupt timing. . . . .	186
7.15	Exception timing during page fault. . . . .	187
A.1	Interrupt sequence. . . . .	195
A.2	Trap sequence. . . . .	197
B.1	Signed integer multiplication. . . . .	202
B.2	Signed integer division. . . . .	204

# List of Tables

2.1	MIPS-X instruction set. . . . .	9
4.1	MIPS-X instruction flow. . . . .	37
5.1	Signals connected to the Execute Unit. . . . .	62
6.1	Signals connected to the PC Unit. . . . .	95
6.2	Signals connected to the Tags Unit. . . . .	117
6.3	Signals connected to the Instruction Cache. . . . .	138
6.4	Decoder Array inputs. . . . .	156
6.5	Decoder Array outputs. . . . .	157
7.1	Miss rates in the external cache. . . . .	163
7.2	Signal timing. . . . .	167
7.3	MIPS-X actions for various instruction types when an exception occurs during their ALU, MEM and WB pipe stages. . . . .	188
7.4	Outgoing signals. . . . .	191
7.5	Outgoing signal values for various instructions. . . . .	191
7.6	Incoming signals and derived signals. . . . .	192
B.1	Number of cycles needed to do a multiplication. . . . .	201
B.2	Number of cycles needed to do a divide. . . . .	205

# Contributors

**Dr. John M. Acken**  
Valid Logic Systems  
2820 Orchard Parkway  
San Jose, CA 95134

**Professor Anant Agarwal**  
Massachusetts Institute of Technology  
Department of EECS - Laboratory for  
Computer Science  
545 Technology Square, Rm 418  
Cambridge, MA 02139

**Professor P. Glenn Gulak**  
Department of Electrical Engineering  
University of Toronto  
Toronto, Ontario, M5S 1A4  
Canada

**Professor Mark A. Horowitz**  
Center for Integrated Systems  
Stanford University  
Stanford, CA 94305

**Scott A. McFarling**  
Center for Integrated Systems  
Stanford University  
Stanford, CA 94305

**Steven E. Richardson**  
Center for Integrated Systems  
Stanford University  
Stanford, CA 94305

**Arturo Salz**  
Center for Integrated Systems  
Stanford University  
Stanford, CA 94305

**Richard T. Simoni**  
Center for Integrated Systems  
Stanford University  
Stanford, CA 94305

**Donald C. Stark**  
Center for Integrated Systems  
Stanford University  
Stanford, CA 94305

**Steve W.K. Tjiang**  
Center for Integrated Systems  
Stanford University  
Stanford, CA 94305

# Foreword

The first Stanford MIPS project started as a special graduate course in 1981. That project produced working silicon in 1983 and a prototype for running small programs in early 1984. After that, we declared it a success and decided to move on to the next project—MIPS-X. This book is the final and complete word on MIPS-X.

The initial design of MIPS-X was formulated in 1984 beginning in the Spring. At that time, we were unsure that RISC technology was going to have the industrial impact that we felt it should. We also knew of a number of architectural and implementation flaws in the Stanford MIPS machine. We believed that a new processor could achieve a performance level of over 10 times a VAX 11/780, and that a microprocessor of this performance level would convince academic skeptics of the value of the RISC approach. We were concerned that the flaws in the original RISC design might overshadow the core ideas, or that attempts to industrialize the technology would repeat the mistakes of the first generation designs.

MIPS-X was targeted to eliminate the flaws in the first generation designs and to boost the performance level by over a factor of five.

From this high-level performance goal, we created a list of secondary goals. At the top of the list were simplicity and a high clock rate. We felt that the MIPS design had become overly complex leading to unnecessary bottlenecks on performance and to unnecessary design complexity. Simplifying the instruction set and increasing the instruction count per program put more focus on the need for a high clock rate.

The goal of a high clock rate led us to confront one of the toughest problems facing designers of high performance microprocessors—the differences in on-chip and off-chip bandwidth. How could we build a machine with a simple instruction set and a high clock rate without putting unreasonable demands on the external interface and memory system? The answer was a fast on-chip instruction cache, but limited die area forced us to find a clever design that minimized tag area without sacrificing hit rate. The streamlined instruction set freed a large amount of silicon for the cache,

reinforcing an advantage of the RISC architecture approach. Non-RISC machines are just reaching the level where they have the same amount of cache that MIPS-X had in 1986.

In addition to the aggressive clock rate, the deeper pipeline in MIPS-X forced us to find new ways to deal with branch penalties. Finally, we felt the need to provide for hardware floating-point support, which had been ignored in the first generation RISC machines. We supplied a high performance coprocessor interface for a floating point unit. The on-chip instruction cache created some additional challenges in delivering floating point instructions to the coprocessor.

MIPS-X also tried a few new design approaches not used in MIPS: MIPS-X used a more innovative and aggressive clocking strategy. The 20 MHz clock rate was attained with this approach, and Mark Horowitz's design skills. MIPS-X also had a more complete CAD strategy—especially for simulation. This helped find bugs in the design much earlier. A solid functional decomposition of the design with simulation provided the basis for a structured design where individual segments could be designed in parallel. Paul Chow's efforts at putting together this simulator and coordinating the design were directly responsible for a bug-free design on first silicon.

Probably the thing that surprised us most about MIPS-X is that the project took essentially the same amount of time that the first MIPS project—about three years to silicon and four years to a board-level demo! We thought that we had learned a lot since MIPS, and we had a much more aggressive schedule than the MIPS schedule. Nevertheless, the project took just as long. Of course, the longer design schedule was directly related to the improvement in performance and increase in the size of the design—both the performance and transistor count were more than five times larger. We could never have done a design as complex as MIPS-X with the experience and tools that we used for the first MIPS processor.

We have found that a project of the duration and complexity of MIPS or MIPS-X is at about the limit of what we can handle in a university. Completing this sort of project requires the unwavering dedication of a few people and the individual contributions of many. The output from such a project consists of a wide variety of insights gained by the team members, a collection of standard publications, and a document of this type. Only with a book of this scope can the entire story of the MIPS-X design be told. Our hope is that this document can convey a fraction of what we have learned from the MIPS-X project.

John Hennessy

# Preface

VLSI microprocessors built according to the RISC philosophy have made a significant impact in the commercial marketplace. Much of the work that led to the development of these processors originated from Stanford University and the University of California at Berkeley. This book is a detailed description of the design and implementation of a second generation VLSI RISC microprocessor at Stanford University called MIPS-X.

The MIPS-X project began at Stanford University in the Spring of 1984 with the goal of investigating second generation RISC architectures. Within the project, there were two distinct groups that interacted very closely to define the architecture of MIPS-X. One of the groups was the compiler group that was looking at code-generation and optimization strategies for RISC machines, and the other group was the hardware group that was responsible for specifying and implementing the actual processor. Together, they worked to design the instruction-set architecture of MIPS-X by using data gathered from benchmarks run on instruction-level simulators.

The MIPS-X processor was successfully implemented and tested in 1986. It demonstrated that a processor using simple hardware and sophisticated compilation techniques could provide very high performance. The results of this work are documented in many places, and the references are collected in the Bibliography.

This book concentrates on the hardware implementation of MIPS-X and begins with a description of the MIPS-X architecture and instruction set. This is followed by an overview of the MIPS-X software system and some of the strategies that it uses to generate efficient code. The remainder of the book is an in-depth examination of the implementation of a VLSI RISC microprocessor. We have attempted to document most of the details of the hardware implementation of MIPS-X, including logic equations, logic and circuit diagrams, and some layout considerations. The principle designers have described each of the major sections of MIPS-X: the Instruction Cache, the Instruction Register, the Register File, the Ex-



ecute Unit, the PC Unit, the Tags Unit, and the external interface.

In Chapter 3 Scott McFarling briefly describes the compiler system and provides citations to the literature for more detailed explanations. Steve Richardson contributed the section on the Interprocedural Analyzer, and Steve Tjiang contributed the section on the Code Generator. Chapter 4 gives a general picture of the hardware resources, and mentions some of the implementation issues. Glenn Gulak provided the description of the naming convention that we used for the signals. The Execute Engine, comprising the Register File and the Execute Unit, is described in Chapter 5. Mark Horowitz did the Register File, and the datapath part of the ALU that was used in the Execute Unit. The ALU was simplified for use as the PC Unit incrementer and displacement adders. Richard Simoni did the Execute Unit. The parts of the processor that have to do with instruction fetching are described in Chapter 6. I did the PC Unit, and Jonathan Rose changed the PC PLA to standard cells. Arturo Salz did the Tags Unit, John Acken did the Instruction Cache, and Don Stark did the Instruction Register. Chapter 7 describes the External Interface of MIPS-X. Section 7.1 is essentially a Stanford Computer Systems Laboratory Technical Report called *MIPS-X: The External Interface*. This was done by Anant Agarwal, Arturo Salz, and myself. Appendix A describes the MIPS-X exception handling in detail, and is taken from the *MIPS-X Instruction Set and Programmer's Manual*[8]. Appendix B is also from the Programmer's Manual, and describes the use of the multiplication and division instructions of MIPS-X.

Paul Chow

# Acknowledgments

Many people have worked on the MIPS-X project, and many others need to be thanked for their support and contributions. There are really several groups who must be acknowledged.

I would like to start by personally thanking the exceptional people that made my time at Stanford a most exciting and enjoyable stay: John Hennessy, Mark Horowitz, and the rest of the MIPS-X gang. John made it all possible for me by taking me on at Stanford as a Postdoctoral Fellow, and I have very much enjoyed my time working with him. His accomplishments are many, and I hope I can follow his example in my own research. I will always be impressed by how he kept such a large group working together, being productive, and having fun at the same time. Mark will continue to inspire me because of his expertise in so many diverse areas, but especially in the many aspects of VLSI design. His breadth and depth of understanding were the key elements that made the chip implementation work. The MIPS-X gang is truly a wonderful and interesting collection of people, and without everyone's hard work, the project would not have been such a success. I must add those that are not in the Contributors list of this book: C.Y. Chu, Steven Przybylski, Peter Steenkiste, and Malcolm Wing. I will miss the rafting trips and the tandoori burgers.

Tom Blank, Giovanni De Micheli, Martin Freeman, and their EE392C class built and made the MIPS-X board work. Michael Smith, Drew Wingard, and Jonathan Rose worked on the Revision 2 part.

Besides those that did the work to make MIPS-X a success, there are those that supported us in other ways. The MIPS-X research was supported by the Defense Advanced Research Projects Agency under Contract MDA903-83-C-0335. Paul Chow, Steven Przybylski, and Glenn Gulak were supported in part by the Natural Sciences and Engineering Research Council of Canada. We gratefully acknowledge Jim Gasbarro and Ed McCreight of Xerox for fabrication support. Margaret Rowland helped keep the world organized.

Finally, there are those that helped me put this book together. The

contributors are the most important. Without them this book would not exist. Special mention goes to Rich Simoni, who helped in the crunch of trying to meet the deadline, and for being the man on the scene at Stanford now that I have moved to the University of Toronto. I would like to thank Jennifer Lambe and Susan McClure for their help with figures, formating, and whatever else I needed help with. Many of the figures were done with *idraw*, and I thank John Vlissides for his help and tips.

Helen Davis, Steve Jurvetson, Jim Laudon, Margaret Martonosi, Arul Menezes, Michael Smith, Josep Torrellas, Antonio R. W. Todesco, Wolf-Dietrich Weber, and Eric Williams provided many useful comments.

This manuscript was prepared using the  $\text{\LaTeX}$ [28] document preparation system.

# Chapter 1

## Introduction

MIPS-X is the successor to the MIPS project at Stanford University. Like its predecessor, it is a single chip VLSI processor that uses a simplified instruction set, pipelining and a software code reorganizer to obtain high performance. Key features include single-cycle execution of all instructions, use of an on-chip 512-word instruction cache, coprocessor support, and support for multiprocessor operation. The processor is designed in a 2  $\mu\text{m}$ , 2-level metal CMOS technology, and will have a cycle time of 50 ns.

During the spring of 1984 the original MIPS design team plus about half a dozen new students began investigating architectures for a new high performance microprocessor. The resulting design, MIPS-X, borrows heavily from the original RISC[33, 27] and MIPS designs[21, 35], though most closely resembles the latter for obvious reasons.

Our experience with the MIPS design[35] reinforced some of our initial observations about machine design. One of the goals of the MIPS project was to develop an architecture that would efficiently execute compiled code. To accomplish this goal, we tried to present as much parallelism of the micromachine to the software system as possible, even when this resulted in a complex set of constraints on the assembly level programmer. One way that MIPS achieved parallelism was through the heavy use of pipelining. All the pipeline interlocks were maintained by the software system; there were no hardware interlocks. Not only did this make the hardware simpler, it also gave the compiler a chance to reorganize the code to make use of the cycle normally wasted during an interlock. These design decisions were enormously successful. Both the heavy use of pipelining and the transfer of the interlocks to the software

were important reasons behind the high performance of MIPS.

Not all the original design tradeoffs made for MIPS were, in hindsight, the best ones. In the original MIPS design, two operations could be packed into a single machine instruction. The architecture supported instructions with two ALU operations in them or an ALU operation with a memory operation. This packing of assembly level instructions was done to reduce the required instruction traffic and to make maximal use of the available hardware. It also made the instructions more difficult to decode and the pipeline more difficult to control. As a result, the clock cycle time of the  $3\mu\text{m}$  nMOS version of MIPS was limited by the instruction decode time.

MIPS-X uses much the same overall organization as the original MIPS machine. It is again a heavily pipelined machine, with a deeper pipeline than the original MIPS design, and the resulting pipeline interlocks are handled by the supporting software system. MIPS-X differs from MIPS in a few important areas:

- All MIPS-X instructions implement a single operation and execute in a single clock cycle.
- All instructions are fixed format; every instruction is 32 bits.
- Simple and efficient support for coprocessors is provided.
- Support for its use as the base processor of a shared memory multiprocessor is provided.
- MIPS-X contains a large on-chip instruction cache.
- The processor was fabricated in a  $2\mu\text{m}$ , 2-level metal CMOS technology.

MIPS-X took about two years from the beginning of the project until the design was complete and the layout sent for fabrication. Over half of this time was spent deciding what to implement. These times are very similar to those for the original MIPS microprocessor[22, 34]. Although the implementation methods were different, both projects spent over a year, and more than half their design time, before work on the details of the circuit design and layout began. It is interesting to note that almost no CAD tools[10] were used in this initial phase of the project except for a small amount of exploratory layout to determine cell sizes. Most of

the architectural decisions for MIPS-X were made using data gathered from the original MIPS software system and instruction-level simulator, and good old-fashioned back-of-the-envelope estimates.

Once we had an idea of what we needed to implement, we partitioned the design according to functional units: the Instruction Cache, the Instruction Register, the External Interface, the Register File, the Execute Unit, the PC Unit and the Tags Unit. A designer was assigned to each of the sections and he was responsible for everything from building the functional description to doing the layout (using *magic*[32]) and verification of his section. This meant that each person became the expert for his part of the chip and did not need to know details about the other sections except for the timing of the signals at the boundaries. The details of various global signals were mostly tracked by one person. Partitioning the design in this way also encouraged the use of separate, independent controllers for each section instead of using a more global approach that would have been more complex.

A chronology of major events is shown in Figure 1.1. Once the initial architecture was specified in the form of the architecture manual[8], detailed work on the microarchitecture of the machine began. A functional simulator, called *Funsim*, was written to check that all the correct signals were being generated and that they had the correct timing. *Funsim* was used extensively during the final verification and was the key reason that the pieces of the design could be assembled and the switch-level simulation running programs within a few days. The final verification only took about two weeks[10].

*Funsim* became the definitive description of the machine and was an important part of the verification process. The switch-level simulator, *RSIM*[44], was connected to *Funsim* so that the switch-level circuit could be stimulated and checked by *Funsim*. This was called *comparison-mode* simulation because the switch-level circuit was run in parallel with *Funsim* so that the behaviour of the two circuits could be compared at every clock step. This was first used to verify the pieces of the design by running sequences of test code and small Pascal programs. Once the pieces of the chip worked correctly in this mode, the chip was put together and simulated further to get the last bugs out.

First silicon was received in October of 1986 and, except for a small problem with the instruction cache, some working parts were found. The next step was to build a board for the processor and this was done by a design class in the spring of 1987. By mid summer, we were able to

Mar. 5, 1984	John Hennessy describes the goals for a new processor to follow the MIPS project.
May 7, 1984	The basic pipeline structure is decided. There are still questions about the instruction cache, using register windows or not, branches and branch prediction, character support and LISP support.
Sept. 21, 1984	An all day meeting is held to discuss the work done over the summer. The datapath components are described along with a proposed instruction set and encoding. The chosen instruction cache organization and branch strategies are explained and data is given to support them. A floorplan for the chip is shown.
Oct. 3, 1984	The final format of the instruction set is settled.
Nov. 5, 1984	The first version of the <i>MIPS-X Instruction Set and Programmer's Manual</i> is published.
Nov. 22, 1984	The instruction-level simulator, <i>MilSX</i> , comes up.
Feb. 14, 1985	The small benchmarks run on <i>MilSX</i> .
Mar. 29, 1985	A document with initial ideas for the control signals and circuits in the datapath is given to the design group.
Apr. 23, 1985	Design of the functional simulator begins.
May 19, 1985	A more detailed description of the datapath timing and circuits is done. This includes stick diagrams for some circuits.
July 2, 1985	The functional simulator kernel is completed and people begin writing models of their sections.
Aug. 5, 1985	The functional simulator is put together and begins to run simple programs a few days later.
Oct. 22, 1985	Start to put the layout for the datapath together.
Oct. 25, 1985	The functional simulator executes a simple Pascal program.
Dec. 27, 1985	The pad frame is built around the chip. The die will be 8mm by 8.5mm.
Feb. 6, 1986	The full datapath is extracted.
Mar. 17, 1986	Extraction of the current full design begins to see if anything will break.
March 26, 1986	Simulation of the various parts of the chip begins using RSIM and Funsim.
Apr. 19, 1986	The extracted version of the full chip minus the instruction cache memory cells runs Ackerman. The functional simulator is used to drive the RSIM model.
May 1, 1986	A version of the design is frozen and archived, ready for transfer to MOSIS. Design also sent to Xerox PARC shortly thereafter.
Oct. 3, 1986	Ten MIPS-X die arrive from Xerox PARC. Mosis parts had arrived earlier but they had a mask problem.
Oct. 5, 1986	Assembling of the test setup is done and initial testing is done.
Oct. 17, 1986	One of the dies runs Ackerman to completion with the instruction cache disabled.
July 31, 1987	The MIPS-X board executes Puzzle successfully.

Figure 1.1: A chronology of MIPS-X.

download Pascal programs to the board and run them.

The MIPS-X project was both a large software effort in exploring advanced compilation techniques and the design of the VLSI processor. This book concentrates on the latter, examining the details of the hardware design and describing the circuits used. In the next chapter we begin with an overview of the MIPS-X architecture and instruction set and describe the hardware-software tradeoffs that were made. Chapter 3 follows with an overview of the MIPS-X compiler system and some of the strategies that it uses to generate efficient code. In Chapter 4 we begin our detailed description of the MIPS-X hardware with an examination of the global issues. Chapter 5 explains the execution engine, which includes the Register File and the Execute Unit. The instruction fetch hardware is explained in Chapter 6. This consists of the PC Unit, the Tags Unit, the Instruction Cache and the Instruction Register. The External Interface is explained in Chapter 7. This focuses on the timing of the signals on the pins.



# Chapter 2

## Architecture

The design of VLSI processors is dramatically affected by the constraints of the fabrication technology. The speed/complexity tradeoffs are extremely important and difficult to resolve. Additional chip functionality requires more area, and often slows down every other function of the chip. This slowdown arises from increased control logic complexity and increased capacitance loading on the wires. The decrease in overall speed can negate the desired improvement in machine performance. To prevent this from occurring in MIPS-X, our overall design philosophy was to keep the processor as simple as possible. When new features, like coprocessor support, were added, the goal was to add the minimum required to accomplish the task.

In this chapter, we first show the architecture of MIPS-X by giving an overview of the hardware resources, the instruction set, and the exception handling. We then examine some of the hardware tradeoffs that were made during the design of the processor.

### 2.1 MIPS-X Overview

MIPS-X is a 32-bit microprocessor that uses a simplified instruction set. The instruction set is designed to be very easy to decode and to require only simple hardware for its execution. Table 2.1 lists the MIPS-X instructions and Figure 2.1 shows the format of the instructions. A detailed description of the bit assignments in the instruction opcodes is given in Appendix C. The simplified instruction set has two important benefits: simple, fast circuits can be used and there is more area left to imple-

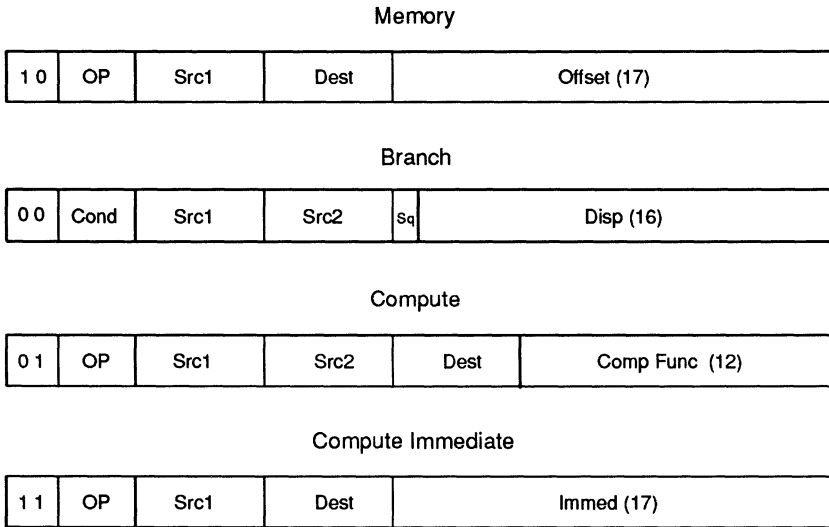


Figure 2.1: The MIPS-X instruction formats.

ment features that can contribute to the performance of the machine. In MIPS-X, most of the area was used to implement an instruction cache.

MIPS-X is a load-store architecture, meaning the only memory operations are explicit loads and stores. It has 32 general purpose registers and provides a 32-bit ALU and a 64-bit to 32-bit funnel shifter for compute operations. Integer multiplication and division are also supported with 1-bit at a time multiply and divide step instructions. Appendix B explains how integer multiplication and division work. Within the program counter section, there is an incrementer that calculates the next sequential address and a displacement adder that computes the branch destination. Providing a separate ALU and displacement adder allows branch instructions to both compute a condition and the target address simultaneously.

There are also several special registers. They are:

- PSW**                    The processor status word.
- PC-4, PC-1**        Locations in the PC Chain used for saving and restoring the state of the PC Chain.
- MD**                    The mul/div register. This is a special register used during multiplication and division.

Instruction	Operands	Operation	Comments
<b>Memory Instructions</b>			
ld	X[Src1],Dest	Dest:=M[X+Src1]	Load
st	X[Src1],Src2	M[X+Src1]:=Src2	Store
ldf	X[Src1],fDest	fDest:=M[X+Src1]	Load floating-point
stf	X[Src1],fSrc2	M[X+Src1]:=fSrc2	Store floating-point
ldt	X[Src1],Dest	Dest:=M[X+Src1]	Load through - bypass cache
stt	X[Src1],Src2	M[X+Src1]:=Src2	Store through - bypass cache
movfrc	CopInstr, Dest	Dest:=CopReg	Move from coprocessor
movtoc	CopInstr, Src2	CopReg:=Src2	Move to coprocessor
aluc	CopInstr	Cop executes CopInstr	Coprocessor ALU
<b>Branch Instructions</b>			
beq, beqsq	Src1, Src2, Label	PC:=Label if Src1 = Src2	Branch if equal
bge, bgesq	Src1, Src2, Label	PC:=Label if Src1 >= Src2	Branch if greater or equal
bhs, bhssq	Src1, Src2, Label	PC:=Label if Src1 >= Src2	Unsigned branch if higher or same
blo, blosq	Src1, Src2, Label	PC:=Label if Src1 < Src2	Unsigned branch if lower
blt, bltsq	Src1, Src2, Label	PC:=Label if Src1 < Src2	Branch if less
bne, bnesq	Src1, Src2, Label	PC:=Label if Src1 <> Src2	Branch if not equal
<b>Compute Instructions</b>			
add	Src1, Src2, Dest	Dest:=Src1+Src2	Integer add
dstep	Src1, Src2, Dest		One step of 1-bit restoring divide algorithm
mstart	Src2, Dest		First step of 1-bit shift and add multiplication
mstep	Src1, Src2, Dest		One step of 1-bit shift and add multiplication
sub	Src1, Src2, Dest	Dest:=Src1-Src2	Integer subtraction
subnc	Src1, Src2, Dest	Dest:=Src1+Src2	Subtract with no carry in
and	Src1, Src2, Dest	Dest:=Src1 bitwise and Src2	Logical and
bic	Src1, Src2, Dest	Dest:=Src1 bitwise and Src2	Bit clear
not	Src1, Dest	Dest:=Src1	Logical not
or	Src1, Src2, Dest	Dest:=Src1 bitwise or Src2	Logical or
xor	Src1, Src2, Dest	Dest:=Src1 bitwise exclusive-or Src2	Logical exclusive-or
mov	Src1, Dest	Dest:=Src1	Really just add Src1, r0, Dest
asr	Src1, Dest, #N	Dest:=Src1 shifted right N positions	Arithmetic shift right
rotlb	Src1, Src2, Dest	Dest:=Src1 byte rotated left	Rotate left by [30:31] bytes
rotlcb	Src1, Src2, Dest	Dest:=Src1 byte rotated left by complement of Src2[30:31] bytes	Rotate left by complement of Src2[30:31] bytes
sh	Src1, Src2, Dest, #N	Dest:=Src2[32-N:31]    Src1[0:32-N-1]	Funnel shift
nop		r0:=r0+r0	No operation. Really just add r0, r0, r0
<b>Compute Immediate Instructions</b>			
addi	Src1, #N, Dest	Dest:=N+Src1	Add immediate 17-bit sign extended
jpc			Jump using value in PC Chain
jpcrs			Jump from PC Chain and restore state
jspci	Src1, #N, Dest	Dest:=PC+1, PC:=Src1+N	Jump indexed, save PC
movfrs	SpecReg, Dest	Dest:=SpecReg	Move from special register
movtos	Src1, SpecReg	SpecReg:=Src1	Move to special register
trap	Vector		Trap to vector

Src2[a : b] means the bit field containing bits a through b in Src2 where the bits are numbered Big-Endian (Bit 0 is the most significant bit).  
The register r0 is hardwired with the constant zero.  
The mov and nop instructions are assembly language equivalents for other instructions.

Table 2.1: MIPS-X instruction set.

The architecture has four types of instructions: memory instructions, conditional branch instructions, compute instructions, and compute immediate instructions. The memory system is byte addressed and is accessed via load and store instructions but the memory system only handles word quantities. All memory operations use the same addressing mode; the contents of a register are added to a 17-bit signed offset to produce a 32-bit address. Memory instructions may communicate with other hardware besides the external cache. We use this fact to implement a fast and simple coprocessor interface.

The coprocessor instructions, *movfrc* (move from coprocessor) and *movtoc* (move to coprocessor), use the memory instruction format. These instructions are used to transfer data between the processor registers and the coprocessor registers. The instruction bits for the coprocessor (*CopInstr*) are put into the *Offset* field, and these bits appear on the address lines of the processor where they can be read by the coprocessor. The format of *CopInstr* is not determined by the processor hardware, but should include bits to identify a coprocessor, a function field, and some source and destination register specifiers.

One coprocessor, usually a floating-point unit, is deemed to be special. The instructions *ldf* (load floating-point) and *stf* (store floating-point) can be used to load and store the registers of one specific coprocessor without needing to use the processor registers as an intermediate step. For these instructions the *Dest* field for *ldf* and the *Src2* field for *stf* instructions appear on the *FPReg* lines of the processor.

All instructions are restartable so MIPS-X will support a dynamic, paged virtual memory system. To help implement such a system, MIPS-X supports both maskable and nonmaskable interrupts. The only internally generated exception is trap on overflow, and this can be disabled by setting a bit in the PSW. For systems requiring more complex interrupt handling, an external interrupt coprocessor can be added. MIPS-X also provides two operating modes, system and user, that execute in separate address spaces to provide the protection needed to implement an operating system. The current mode is stored in the PSW and it can only be changed while executing in system mode.

### 2.1.1 Memory Organization

The memory is composed of 32-bit words and it is a uniform address space starting at 0 and ending at  $2^{32} - 1$ . Each memory location is a

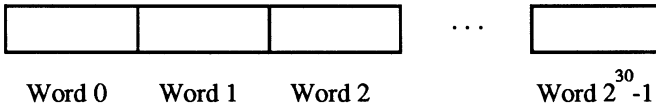


Figure 2.2: Word numbering in memory.

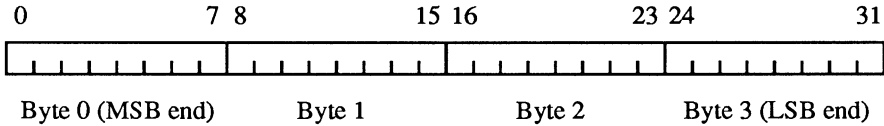


Figure 2.3: Bit and byte numbering in a word.

byte. Load/store addresses are manipulated as 32-bit byte addresses on-chip but only words can be read from memory (ie., only the top 30 bits are sent to the memory system). The numbering of words in memory is shown in Figure 2.2. Bytes (characters) are accessed by sequences of instructions that can do insertion or extraction of characters into or from a word. Instructions that affect the program counter, such as branches and jumps, generate word addresses. This means that the offsets used for calculating load/store addresses are byte offsets, and displacements for branches and jumps are word displacements. The addressing is consistently *Big Endian*[11].

Bytes are numbered starting with the most significant byte at the most significant bit end of the word. The bits in a word are numbered 0 to 31 starting at the most significant bit (MSB) and going to the least significant bit (LSB). Bit and byte numbering are shown in Figure 2.3.

The address space is divided into system and user space. An address with the high order bit (bit 0) set to one (1) will access user space. If the high order bit is zero (0) then a system space address is accessed. Programs executing in system space can access both system and user space. User programs cannot access system space. Any attempt to access system space will result in the address being mapped to user space. Bit 0 of the address will always be forced to 1 (a user space address) in user mode.

### 2.1.2 Caches

Memory bandwidth is an important issue in high performance processor design and MIPS-X is no exception. The processor is capable of

executing one instruction every clock cycle. At a clock frequency of 20 MHz it requires 30 to 40 Mwords/sec of memory bandwidth to satisfy its instruction and data fetch requirements. Getting this amount of data across the available package pins would be a difficult task. The solution used in MIPS-X is to provide a large (2K byte) instruction cache (ICache) on the chip. Only instructions that miss in the on-chip cache need to pass through the package pins. The cache is placed above the datapath, in the area of the chip that is normally used for microcode storage and processor control. External memory fetches, consisting of data references and instruction references that miss in the ICache, are handled by a large (16K to 64K word) external cache (Ecache). The external cache uses a shared bus to communicate with main memory. An added benefit of this two-level cache is that it effectively provides a second port to memory; the processor can fetch an instruction from the ICache at the same time it is accessing off chip data.

Store instructions assume that the Ecache is a write-back cache meaning that two cycles are needed to complete the store. The first cycle is used to probe the cache to make sure the location exists and the second cycle is used to do the actual store. The software must schedule a non-memory instruction after every store instruction. However, ICache misses cannot be predicted so there is one hardware interlock that occurs if an ICache miss occurs after the first cycle of the store instruction. This interlock stalls the processor for one cycle to complete the store before the ICache miss sequence begins.

### 2.1.3 Pipeline Organization

MIPS-X is heavily pipelined so that one instruction can be issued every cycle. Each instruction is divided into five pipeline stages. They are: **IF**, Instruction fetch; **RF**, Instruction decode and register fetch; **ALU**, ALU or shift operation; **MEM**, Wait for data from memory on a load and output data for a store; and **WB**, Write the result into the destination register.

The use of the **ALU** cycle depends on the instruction being executed. For compute or compute immediate instructions, this cycle performs the desired computation. During memory instructions the ALU is used to compute the address of the desired memory location. For branch instructions the ALU is used to compare two registers to determine the branch condition.

Although a compute instruction finishes its computation during the third pipeline cycle (**ALU**), the result is not written back into the Register File until the last pipeline cycle. This delayed write-back is done to make instructions only change machine state during their last pipeline cycle, making exception handling much easier. Internal bypassing is provided so that instructions do not have to wait for previous instructions to write their results in the Register File (during **WB**) before being able to access their results. With bypassing, two ALU instructions can be issued in sequence even if the second one uses the result of the first one. For a load instruction, there is still a delay of one cycle between the load instruction and the availability of its result for the next instruction. The software must take this *load delay* into consideration.

Another pipeline interlock that arises from this pipeline arrangement is a *branch delay* of two. In a branch instruction the result of the comparison is not known until the end of the ALU pipeline stage. By this time, two instructions have been fetched before the result of the branch can take effect. The software tries to schedule useful instructions in the delay slots. To increase the probability of finding useful instructions, one version of the branch instructions makes it possible to statically predict that the branch will be taken. If the branch is not taken, the instructions in the delay slots are *squashed* or cancelled. As with branches, the jump instruction has a delay of two because the ALU is used to compute the jump address.

#### 2.1.4 The Processor Status Word

The Processor Status Word (PSW) holds some of the information pertaining to the current state of the machine. The PSW actually contains two sets of bits that are called *PSWcurrent* and *PSWother*. The current state of the machine is always reflected in *PSWcurrent*. When an exception or trap occurs, the contents of *PSWcurrent* are copied into *PSWother* except for the *e* bit. *PSWother* then contains the processor state from before the exception or trap so that it can be saved. Also, interrupts are disabled, PC shifting is disabled, overflows are masked and the processor is put into system state. The *I* bit is cleared if the exception was an interrupt. A *jump PC and restore state* instruction (*jpers*) causes *PSWother* to be copied into *PSWcurrent*. After the ALU cycle of the *jpers* instruction, the interrupts are enabled and the processor returns with its previous state restored.

The PSW can be both read and written while in system space, but a write to the PSW while in user space has no effect. To change the current state of the machine via the PSW, a *move to special* (*movtos*) instruction must be used to write the bits in *PSWcurrent*. Before restoring the state of the machine, a *move to special* instruction must be used to change the bits in *PSWother*. All the bits are writable except the *e* bit and the *E*-bit shift chain.

The assignment of bits is shown in Figure 2.4. The bits corresponding to *PSWcurrent* are shown in upper case and those in lower case correspond to the bits in *PSWother*. The bits are:

- I, i**     The *I* bit should be checked by the exception handler. It is set to 0 when there is an interrupt request, otherwise it will be set to a 1. This bit never needs to be written but the value will be retained until the next interrupt or exception. The *i* bit contains the previous value of the *I* bit but in general has no meaning since only the *I* bit needs to be looked at when an exception occurs.
- M, m**     Interrupt mask. When set to 1, the processor will not recognize interrupts. Can only be changed by a system process, an interrupt or a trap instruction.
- U, u**     When set to 1, the processor is executing in user state. Can only be changed by a system process, an interrupt or a trap instruction.
- S, s**     Set to 1 when shifting of the PC Chain is enabled.
- e**         Clear when doing an exception or trap return sequence. Used to determine whether state should be saved if another exception occurs during the return sequence. This bit only changes after an exception has occurred so the exception handler must be used to inspect this bit.
- E**         The *E* bits make up a shift chain that is used to determine whether the *e* bit needs to be cleared when an exception occurs. The *E* bits and the *e* bit are visible to the programmer but cannot be written.
- V, v**     The overflow mask bit. Traps on overflows are prevented when this bit is set. If this bit is off, then the processor will trap





all the instructions currently in the pipeline. A detailed description of the exception handling sequence is given in Appendix A.

## 2.2 Hardware Design Tradeoffs

During the design of the MIPS-X instruction set there were a number of interesting tradeoffs made between what needed to be implemented in hardware to achieve maximum performance and what could be done either in the software or with simpler, less efficient hardware. In all cases the decision was made on the basis of performance gain versus hardware required. A less complex solution was used if additional hardware would only improve performance by 5%. We will examine some of the more interesting cases: the coprocessor interface, the branch mechanism, shift operations and exception handling.

### 2.2.1 Coprocessor Interface

Many commercial processors use a coprocessor for complex operations such as floating point but their interfaces are not very efficient. Considerable time was spent trying to find an interface for MIPS-X that would give reasonable performance and still fit within the constraints of VLSI packaging design. The key issue in the MIPS-X design was the presence of the on-chip instruction cache because not all instructions would be visible to the outside world. Some other way would be needed to send instructions to the coprocessors. The simplest solution was to have an extra bus from the chip that would be the coprocessor instruction bus. The problem was that this would necessitate making the instruction cache smaller to fit the extra pads required. It was also argued that if more pads were needed, there would be better uses for them than dedicating them to the coprocessor. Another suggestion was to force coprocessor instructions to be non-cached so that they could always be visible outside of the main processor. However, this meant that all coprocessor instructions incurred the overhead of cache-miss cycles. The final solution made coprocessor instructions a form of memory instruction. The coprocessor instruction was encoded in the offset field of the memory-type instructions and appears on the Address Bus. An extra pin was needed to indicate that this was not a memory access. This solution used available pins without requiring a lot of extra hardware.

The main difficulty was that coprocessor instructions would be limited to 17 bits.

A further refinement of the coprocessor interface was made. It was observed that the coprocessor registers could only access memory by first going through the main processor registers. This was unsatisfactory for floating-point operations so two instructions were added that allowed one coprocessor to load and store its registers directly with memory. This required the addition of four pins to indicate the coprocessor register to be used and meant that a floating-point unit could access memory in the same time as the main processor. Other coprocessors would need an extra cycle to transfer the data between the processor and the coprocessor.

### 2.2.2 Branch Mechanism

The branch mechanism in a pipelined machine has one of the largest impacts on the performance of the machine. Branches disrupt the flow of the pipeline so it is important that their effects are minimized. Condition codes were eliminated very quickly because data[20] showed that most branches required explicit compare operations to set the condition codes. This indicated that very little was saved by having them. Also, condition codes mean extra state that must be saved and is especially difficult to do in a heavily pipelined machine.

The most important aspect of the MIPS-X branch scheme is the two-slot delayed branch. In the MIPS-X pipeline this is the most natural way of implementing a delayed branch scheme because by the time the comparison has been done in the ALU cycle, two instructions have been issued. Having two delay slots was originally thought to be a problem because the experience with MIPS showed that it would be difficult to fill both slots with useful instructions. MIPS had only one delay slot and filled it with an instruction that either did useful work or had no effect if the branch went the wrong way. In MIPS-X we considered several ways of making the branch more efficient. One scheme was to use hardware and build a branch prediction cache. Another idea was to allow only simple conditions that could be evaluated at the end of the RF stage of the pipeline, reducing the branch delay to one. The resulting scheme uses static prediction and a mechanism for *squashing* or cancelling the instructions in the delay slots if the branch is not taken. When used with profiling information, this scheme was shown to be as effective as a hardware branch prediction scheme and the single-slot scheme[30]. It

also required very little hardware mainly because the squashing hardware was already needed to handle exceptions. The branch prediction scheme would have been complicated to implement and it would have taken up much area for the branch prediction cache. The single slot scheme was not used because it would require putting more logic in one of the critical paths of the machine. The critical path was getting data from a load instruction onto the chip and through the bypass registers. The comparison logic would be added to the end of this path.

### 2.2.3 Shift Operations

Implementing hardware to do shift and rotate operations poses a dilemma because these types of instructions are used very infrequently (only 1-5% in MIPS-X traces) yet if no hardware is provided, certain operations, like character handling, become very difficult. We first thought that we could implement a very simple shifter that could do small fixed shifts of 1, 2, 4, and 8 bits but this turned out to be about the same as building a full shifter. This fact, coupled with the need to extract and insert bytes, caused us to re-evaluate building a full shifter. We implemented a funnel shifter that concatenates two 32-bit operands to form a 64-bit double word and then extracts any continuous subfield of 32-bits from that double word. The only variable operation implemented was variable byte rotates because this was needed to make character extraction and insertion work without doing a lot of masking and shifting.

### 2.2.4 Exception Handling

Exception handling was one area where we learned a lot from the MIPS experience. The MIPS processor tried to complete the instructions that were in the pipeline before the point where the exception occurred. This meant that there were a large number of possible states and this increased the complexity of the control. In MIPS-X we tried to minimize the number of possible states in the machine. When an exception occurs in MIPS-X, the pipeline is halted and the PC is immediately set to zero. The shift chain of old PC values is frozen to save the addresses of the instructions that are still in the pipeline. The *PSW<sub>current</sub>* bits are saved into *PSW<sub>other</sub>*, interrupts are turned off and the machine is placed into system mode. At the start of the exception handler, the values in the PC Chain and *PSW<sub>other</sub>* are saved before interrupts and PC Chain shifting is resumed. To restart after an exception the saved PC values

are restored in the PC Chain and three jumps are executed using the contents of the PC Chain. By doing the simplest action for an exception we were able to build very simple controllers for the machine. Very little is lost by not completing instructions in the pipeline because exceptions should occur infrequently.

## Chapter 3

# The Compiler System

A key factor contributing to the performance achieved by RISC microprocessors is the compiler system and this was certainly true for MIPS-X. The MIPS-X project had a software group that worked closely with the hardware designers to define the instruction set of the processor.

This chapter gives an overview of the MIPS-X compiler system. Optimization techniques, code generation, code reorganization, and the instruction-level simulator are described. References to the literature are cited for those interested in learning more details.

### 3.1 Overview

The MIPS-X compiler introduced a number of new optimization techniques and served to drive the design and evaluation of the MIPS-X architecture. Both Pascal and LISP are supported. A high level of both machine independent and machine dependent optimizations are applied. Some phases of the optimization can use profile information from previous executions of the program to guide the optimization. Figure 3.1 shows the various phases of the compiler. They are described below:

- upas**      Pascal Front-end. Generates U-code intermediate form. Based on the Pascal\* compiler[23].
- merger**    Procedure Merger. Selectively inlines procedures to decrease the cost of procedure calls. Can make use of profile information to reduce the number of additional instruction cache misses caused by duplicating code[29].

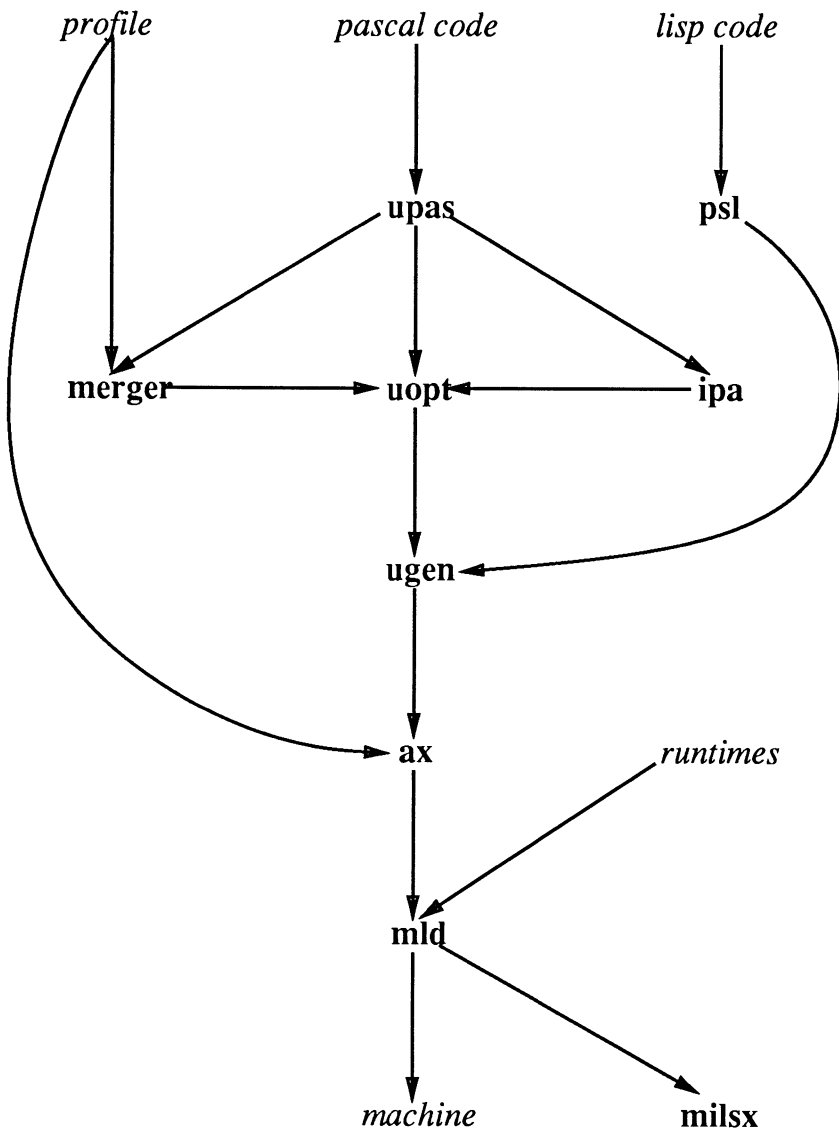


Figure 3.1: The MIPS-X compiler.

- ipa** Interprocedural Analyzer. Constructs data-flow information across procedure boundaries[37].
- uopt** Global optimizer. Performs standard compiler optimizations including register allocation and array indexing optimizations[6, 7].
- psl** Portable Standard LISP compiler. Translates LISP into MIPS-X assembly code[40, 41, 43, 42].
- ugen** Code Generator. Uses Twig[2, 45], a language designed for building code generators.
- ax** Assembler and Reorganizer. Translates assembly language code into legal machine code that satisfies the pipeline constraints of the machine. The code is reordered to reduce pipeline stalls and branch latencies[30].
- mld** Linker. Combines object code from various modules with the runtimes to form an executable file.
- milsx** MIPS-X instruction level simulator. Emulates instructions for performance analysis and compiler debugging.

## 3.2 Interprocedural Analyzer

Richardson and Ganapathi have implemented an interprocedural dataflow analyzer to investigate its impact on code optimization. Theoretical work by Banning and later work by Cooper and Kennedy provided the basis for the analyzer [3, 12].

The interprocedural analyzer scans once through a program's intermediate form (UCODE, as described below) and produces three summary sets:

**MOD(*u*)**, for each procedure *u*, the set of variables global to *u* that may be modified by a call to *u*;

**USE(*u*)**, for each procedure *u*, the set of variables global to *u* that may be in some way accessed by a call to *u*; and

**ALIASES(*x*)**, for each variable *x*, the set of variables that may be aliased to *x*.



These sets are *flow-insensitive* in the sense that the events described (e.g. the modification or use of a given variable) will happen on at least one path of execution through the procedure, but are not guaranteed to happen on every path. Flow-insensitive information is sometimes termed *may* information because it refers to events that may or may not happen when a procedure is invoked. Other work has concentrated on flow-sensitive information, which is more difficult to gather [4].

The summary sets gathered by the interprocedural analyzer might potentially be used for such techniques as procedure integration, linkage tailoring, interprocedural register allocation, or interprocedural constant propagation [36]. The sets have actually been utilized in connection with the MIPS-X global optimizer UOPT. Thus far, however, the results of this experiment have not been encouraging [37].

### 3.3 Global Optimizer

The MIPS-X global optimizer is the same one used for MIPS[6, 7]. The optimizer both reads and writes UCODE, a machine-independent stack-based intermediate language. No changes were needed to UCODE or to the optimizer except for changing some parameters such as the number of registers. The major optimizations undertaken by the optimizer are shown below:

**Common Sub-Expression Elimination** Removes duplicate computations.

**Constant Arithmetic** Computes calculations that can be performed at compile time.

**Copy Propagation** Replaces the use of a copy with the use of the original value, which may make the copy unnecessary.

**Redundant Store Elimination** Removes stores that will be overwritten before the stored value is used.

**Code Motion** Attempts to move calculations out of loops so that they will be executed fewer times.

**Reduction of Operator Strength** Typically this replaces multiplies for array indexing with adds to an array pointer.

**Induction Variable Elimination** Attempts to replace uses of an induction variable with uses of an array pointer, saving the induction variable calculations.

**Register Allocation** Performs global allocation of variables to registers using a variation of Chaitin's graph coloring algorithm[5] developed by Chow[7].

### 3.4 LISP Front-end

The LISP front-end for MIPS-X[40, 41, 43, 42] is a port of the Portable Standard LISP Compiler (PSL)[16, 18, 17]. Steenkiste choose to generate MIPS-X assembly language directly rather than generate UCODE. LISP has some special operations that are currently not well supported by UCODE. In addition, the PSL compiler was more tuned to generating machine code than stack-based code.

The reason for developing a LISP front-end for MIPS-X was to study the impact of LISP on architectural choices and to study new optimizations that might be particularly effective for LISP. Steenkiste showed that the mix of instructions used by LISP programs is very similar to other languages such as Pascal. Thus, MIPS-X has high LISP performance even without any special hardware support. Tags can be handled with normal instructions and the additional savings to be gained from special tag support is small. The compiler also introduced some new ideas for procedure call optimization and interprocedural register allocation. For LISP, certain procedures such as *cons* should always be inlined. The cost of other calls can be reduced with generalized tail recursion removal. If the last operation in a procedure is a call, there is no reason to return to the procedure. Also, Steenkiste showed that most LISP programs spend most of their time executing procedures near the leaf in the call graph. The LISP compiler makes use of this fact when allocating registers across procedure boundaries. Variables used in procedures close to the leaf are allocated first.

### 3.5 Code Generator

The code generator for MIPS-X uses Twig [45] and translates UCODE into MIPS-X assembly code. Twig builds the code generator from a set of pattern-action rules. A pattern is an expression tree and the action

generates code for that tree. Each rule has a cost that is derived from the costs of the machine instructions. Twig uses dynamic programming and tree matching to produce good quality code.

Since Twig generates code by performing pattern matching on trees, UGEN must extract expression trees from its stack-based UCODE input. The pattern matcher takes the trees and breaks them into subtrees that correspond to one or more MIPS-X instructions. The subtrees are chosen so as to minimize the cost of the instruction sequence generated. RISC machines are particularly appropriate for automatic creation of the code generator since there are fewer choices of instructions to perform a particular operation and the cost of each instruction is easy to determine. Therefore there are fewer rules than for CISC machines. However, many Twig patterns were needed to take care of the idiosyncrasies of UCODE.

The code generator assumes an idealized MIPS-X that has no branch, load, and store delays. These delays are taken care of by the assembler through code reorganization.

## 3.6 Reorganizer

In constructing the MIPS-X compiler, the MIPS reorganizer[19, 24] was completely replaced. The MIPS reorganizer generated very good code but was slow and would have been difficult to port to MIPS-X. For MIPS-X we desired good quality code plus a fast and simple algorithm. We also needed a reorganizer that could be used as a test bed for architectural choices and new compilation techniques. The final program also introduced a new technique of reorganizing across basic block boundaries. In the next three subsections we present the methods used for local reorganization, interblock reorganization, and branch scheduling.

### 3.6.1 Local Reorganization

Much of the complexity of the MIPS reorganizer came from attempting to change the order that a resource such as a register is used. Figure 3.2 shows the algorithm used by the MIPS-X reorganizer. The use of a particular resource must be kept in source order. Given a choice, the reorganizer will schedule stores as early as possible. Since stores interfere with all other memory operations, they tend to be difficult to schedule without a stall cycle. By scheduling a store first, if there is another instruction that could have been started before the store that can be

```

procedure LocalReorg(a basic block)
  construct data flow DAG with arcs from instructions to all
    instructions they depend on
  for all instructions i in basic block
    if i is a branch
      label[i] := 2
    else
      label[i] := 0
    end
  end
  end
  for all instructions i in basic block in reverse order
    find maxSon the son of i in the data flow DAG with the
      maximum label
    label[i] := max(label[i],label[maxSon] + InterlockLength(i,maxSon))
  end
  end
  repeat
    Pick instruction i with (in order of priority)
      (1) i is not a branch
      (2) all sons in DAG scheduled and operations completed
      (3) i is a store
      (4) i has maximum label
      (5) i is a load
    Mark i to be emitted next
  until(all instructions in basic block marked)
end

```

Figure 3.2: Local reorganization algorithm.

executed immediately after the store, a cycle will be saved. If there are no stores, the instruction with the longest critical path label is scheduled next. The critical path counts the length of any sequences of instructions depending on an instruction and any interlock delays between instructions such as load delays. If more than one instruction has the same critical path length, loads will be scheduled first. Branches are labeled with a critical path length label of 2. This makes it easy to determine if any instructions can be placed after the branch in delay slots.

As an example, consider the basic block shown in Figure 3.3. The code on the left shows a basic block before reorganization with the stall cycles shown. The basic block would take nine cycles to execute if the

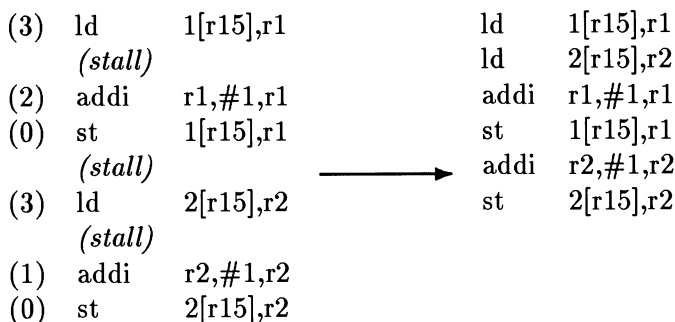


Figure 3.3: Local reorganization example.

original order is maintained. On the right is the same basic block after reorganization. Here only six cycles are needed. The numbers at the far left show the critical path labels that are generated as shown in Figure 3.2. Both loads have the same label and either one could be scheduled first but both are preferred over any other instruction. Next, the add immediate instruction corresponding to the first instruction is scheduled since it is the only instruction that can be started that cycle. After the add, the first store is scheduled since stores have a higher priority. This allows the remaining add immediate to be placed after the store while the memory is busy, even though the add immediate has a critical path label higher than the remaining store.

### 3.6.2 Interblock Reorganization

Ellis and Fisher[14, 13] reorganized code for machines with very wide instructions. To find a large number of operations that could be executed simultaneously, they were forced to analyze more than one basic block at a time. Their algorithm traces through a chain of basic blocks that is thought to be the most likely to be executed sequentially. This trace is reorganized to increase the number of operations performed simultaneously. A similar algorithm could be used to fill pipeline latency stalls on MIPS-X, but the complexity of the trace scheduling does not seem justified for this less ambitious problem.

The MIPS reorganizer[19, 24] analyzed successor basic blocks attempting to fill branch delay slots. An instruction could be moved from

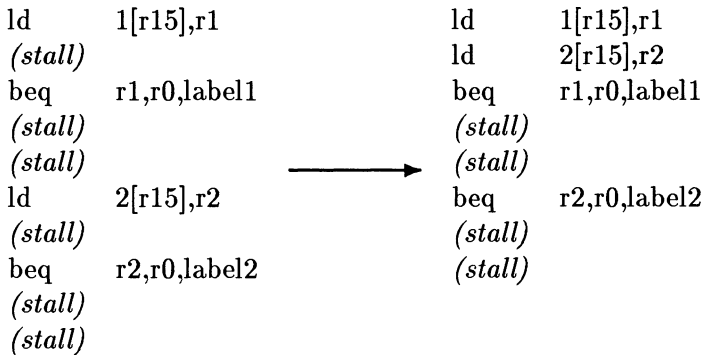


Figure 3.4: Interblock reorganization example.

the sequentially next basic block or from the target of the branch. For this operation to be legal, the reorganizer must insure that the operation moved will not destroy any information that the program still needs. As we will see in the next section, branch slots on MIPS-X are usually filled in a different way. However, any successor instructions that could be put in a branch slot may be used to fill any other pipeline stalls. Care is needed to insure that the moved instruction does not destroy any information needed later in the basic block and that the program semantics are maintained for all possible sequences of basic block executions. As an example, consider the assembly code shown in Figure 3.4. The code on the left is before reorganization. Assuming register 2 contains no information that will be used if the first branch is taken, the second load can be moved up into the first basic block as shown. To make this determination, the reorganizer performs live register analysis. In Figure 3.4, when the first branch is not taken, this optimization saves two cycles. On typical programs, interblock reorganization saves about 3% in execution time.

### 3.6.3 Branch Optimization

Most RISC machines have only a single delay cycle after a branch. MIPS-X has two. As Figure 3.5 shows, if possible, two instructions preceding a branch are moved after the branch. Usually two instructions cannot be found that can be legally moved. Figure 3.6 shows an


<pre> add    r1,r3,r3 add    r2,r3,r3 beq    r4,r0,label1 (stall) (stall) </pre>		<pre> beq    r4,r0,label1 add    r1,r3,r3 add    r2,r3,r3 </pre>
--	---	--

Figure 3.5: Delayed branch scheduling.

alternate method of optimizing the branches. The branch instruction is converted into a squashing branch that will execute the delay instructions only if the branch is taken. Two instructions from the target are copied into the delay slots and the branch displacement is adjusted so that the instruction after the copied instruction is executed after the now filled delay cycles. If the branch is taken, two cycles are saved. If the branch instruction is not executed the instructions in the delay slots are not executed and no harm is done.

Branches are taken more often than not and filling delay slots from the target is better in general than filling instructions from the sequentially next basic block. However, if the reorganizer has profile information available, it can determine which branches are usually taken and which are usually not taken. Usually not-taken branches can be optimized as shown in Figure 3.7. Here the branch is modified so that it branches to the sequentially next basic block in the frequent case. A new branch is inserted so that in the unlikely case, the original target basic block will still be executed next. Even though the introduction of a new branch increases the execution time for the less likely case, this is more than made up by speeding up the more frequent case. With all these optimizations combined, typical branches on MIPS-X require about 1.35 cycles per branch. This is actually faster than most RISC machines with only a single delay slot after branches.

### 3.7 Machine Simulator

*Milssx*, the MIPS-X instruction-level simulator, was developed early in the design of the machine. The simulator served as a testbed for constructing the compiler and evaluating the effectiveness of various optimizations. It has the capability of executing MIPS-X instructions one

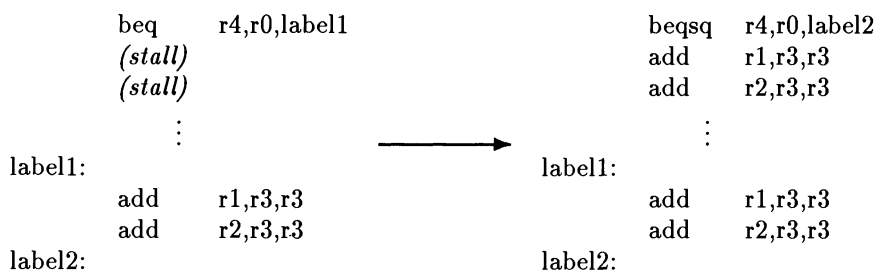


Figure 3.6: Squashed branch scheduling.

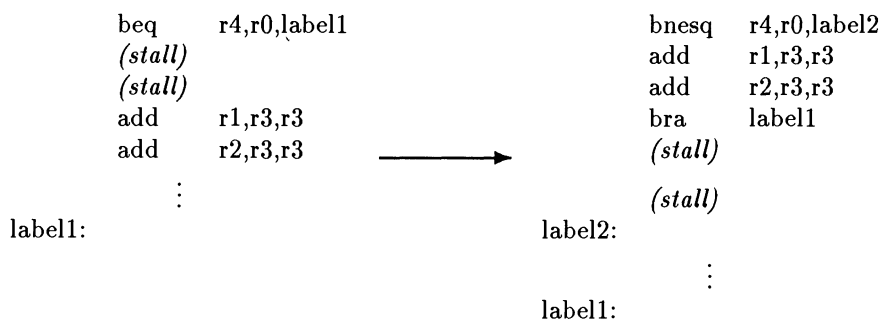


Figure 3.7: Profile assisted optimization of a usually not-taken branch.



at a time so that any program that runs on the simulator, should run on the machine. Another function of the simulator was to collect statistics of which instructions are executed and how fast the machine would be. It can also generate instruction and data traces to aid the design of the overall system.

To perform all these functions for realistic programs, *milsx* needed to be quite fast. This was achieved with a two step process. First, instructions are read in and decoded. The various fields are extracted and converted into a more simple form. For example, a register specifier is represented as a pointer to the address in memory used by the simulator to hold the register contents. Finally, the decoded program is executed. This two-step process results in an instruction emulation rate of 20,000 instructions per second on a VAX-11/780.

# Chapter 4

## A Hardware Overview

Decisions about what should be implemented in hardware and what can be handled in software will have the most impact on the performance of a processor. All processors support the same basic hardware functions since they must all fetch instructions and execute them. What makes one processor different from another is the way that the functions are combined and the way that the hardware is organized to carry out the functions. In a reduced instruction set computer (RISC) architecture, such as MIPS-X, the emphasis has been placed on building hardware that implements the common, simple functions quickly and then letting the software decide how the functions should be combined to carry out more complex operations.

The MIPS-X hardware can be divided into six major sections: the Register File, the Execute Unit, the PC Unit, the Tags Unit, the Instruction Cache (ICache) and the Instruction Register. Figure 4.1 shows a photograph of the MIPS-X die with the sections outlined. In this chapter we will describe the global organization of the hardware resources in MIPS-X and the instruction execution pipeline. We will also explain the global implementation issues.

### 4.1 The Hardware Resources

The dominant feature in Figure 4.1 is the **ICache RAM**. It is 512 words by 32 bits and sits above the datapath. The **Register File** section consists of the array of 32 general-purpose 32-bit registers, the pipeline bypass registers and the registers used for interfacing to external memory.

The **Execute Unit** includes a 64-bit to 32-bit funnel shifter, a 32-bit ALU, the MD registers that support multiplication and division, and the processor status word. In the **PC Unit** there are two 32-bit adders used to increment the PC and to compute the destinations of branches, and a chain of shift registers (the PC Chain) to save the PCs of instructions in execution. These PCs are needed to restart the machine after an exception. The **Tags Unit** contains 32 tags (Tag RAM) and 512 valid bits (Vstore) for the ICache. Between the datapath and the ICache is the **Instruction Register (IReg)**, which contains pipeline staging registers and instruction decode logic. The IReg is also used during ICache misses to write instructions into the ICache RAM array.

Figure 4.2 shows the hardware resources and the major buses that connect them. The *Src1 Bus* and the *Src2 Bus* are used to read data from the Register File. The *Result Bus* carries data words to the Bypass registers to be written to the Register File, and addresses to the Tags Unit where the *Result Bus* is multiplexed with the *PC Bus*. Addresses for memory instructions are carried on the *Result Bus* and addresses during ICache misses come from the *PC Bus*. The *Result Bus* and the *PC Bus* are multiplexed onto the *Address Bus*. The *Immediate Bus* carries immediate values from the IReg for use in the PC Unit and the ALU. The *MemBus* carries instructions from the data pads to the ICache during ICache misses and between the datapath and the data pads during memory instructions.

## 4.2 Clocking

MIPS-X uses a two-phase non-overlapping clocking scheme. The two phases are called *Phi1* or  $\phi_1$ , and *Phi2* or  $\phi_2$ . There are also several derived clocks:

**Phi2\_b** This is the complement of  $\phi_2$ .

**Psi1 or  $\psi_1$**  The  $\psi_1$  clock is  $\phi_1$  qualified by  $\overline{ICacheMiss}$  and  $\overline{Miss}$  and is the clock that is used to cause state changes to occur. The processor is easily stalled by preventing  $\psi_1$  from rising.

**Psi1PC or  $\psi_1PC$**  This clock is only used in the PC Unit. When the processor is stalled with  $\psi_1$ , this clock continues to allow parts of the PC Chain to be clocked during cache misses (See Section 6.1.8).

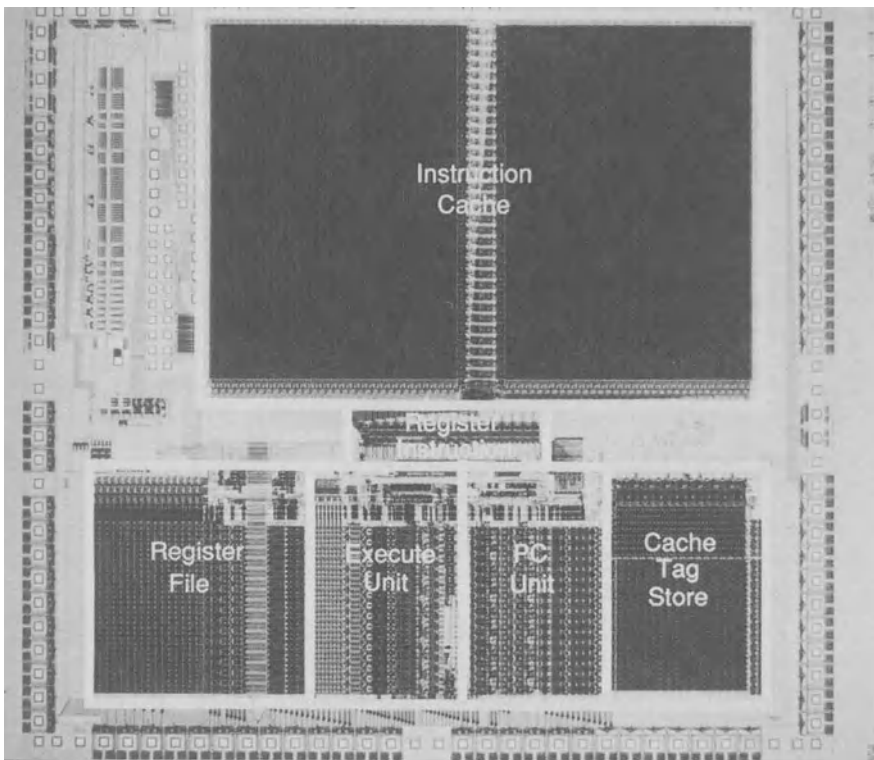


Figure 4.1: The MIPS-X die photo.

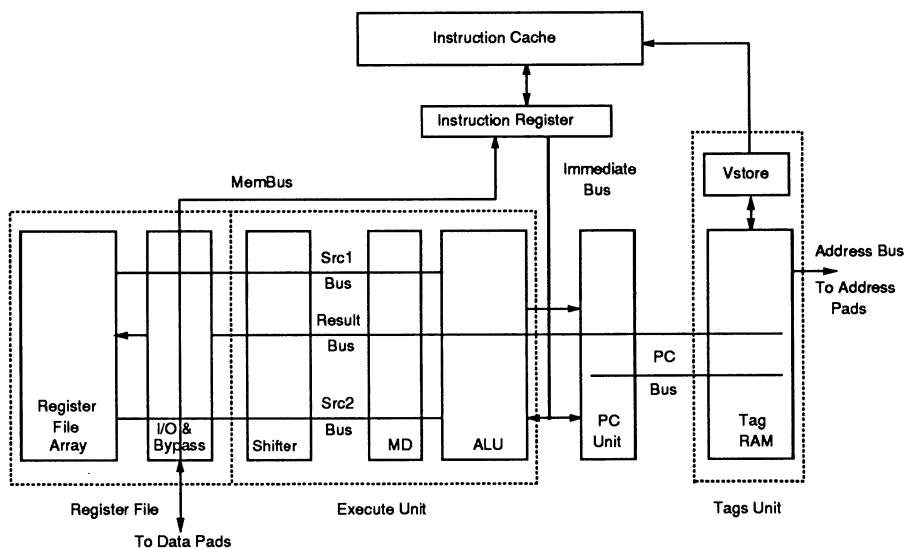


Figure 4.2: The MIPS-X hardware resources.

### 4.3 General Instruction Flow

The interaction of resources in MIPS-X can be understood by examining what the hardware is doing in each phase of the execution of an instruction. The pipeline is only five stages long (10 clock phases) and is shown in Table 4.1. An additional stage,  $IF_{-1}$ , is also shown. This pipe stage occurs before  $IF$ , during the  $WB$  stage of the previous instruction. The use of  $IF_{-1}$  is mainly for historical reasons, and to make the actions within each pipe stage correspond to the name of the pipe stage.

The instruction fetch begins during  $IF_{-1}$  when the PC Bus is driven with the address of the next instruction to be fetched. During  $\phi_1$  of  $IF$ , ICache hit detection is done so that in the next phase the ICache can be accessed or, if there is a miss, the address driven to the address pads. On  $\phi_1$  of  $RF$ , the Register File access begins and the bypass register comparisons are done. The register or bypass values are then driven onto the Src1 and Src2 Buses during  $\phi_2$  so that they can be used in the Execute Unit during the next phase. At the same time in the PC Unit, the current PC and the branch displacement field in the Immediate Bus are being latched for the incrementer and the displacement adder. The PC incrementer, the ALU, and the PC displacement adder all operate

---

<i>IF<sub>-1</sub></i>	$\phi 1$	No action
	$\phi 2$	PC Bus $\leftarrow$ displacement adder, trap vector, incrementer, ALU or value from PC Chain Precharge tag comparators and valid bit store

---

<i>IF</i>	$\phi 1$	Do tag compare Valid bit store access ICache address decoder $\leftarrow$ PC[bits 26 to 31] Detect ICache hit Precharge ICache
	$\phi 2$	Do ICache access Instruction Register $\leftarrow$ ICache

---

<i>RF</i>	$\phi 1$	Do bypass register comparisons to see if bypassing required
	$\phi 2$	Src1 Bus $\leftarrow$ Src1 register or bypass register Src2 Bus $\leftarrow$ Src2 register, bypass register or offset field in memory instruction PC displacement adder latch $\leftarrow$ branch displacement from Immediate Bus Output memory data register $\leftarrow$ Src2 register or bypass source

---

<i>ALU</i>	$\phi 1$	Do ALU operation, shifter operation, PC displacement adder addition Increment PC (calculate next sequential instruction address) Precharge Result Bus
	$\phi 2$	Result Bus $\leftarrow$ ALU Result bypass register $\leftarrow$ Result Bus Memory address pads $\leftarrow$ Result Bus

---

<i>MEM</i>	$\phi 1$	No action
	$\phi 2$	Input memory data register $\leftarrow$ Result register or Memory data pads (load instruction) Memory data pads $\leftarrow$ Output memory data register (store instruction)

---

<i>WB</i>	$\phi 1$	Destination register $\leftarrow$ Input memory data register
	$\phi 2$	No action

---

Table 4.1: MIPS-X instruction flow.

during  $\phi 1$  of the *ALU* stage. The result of the ALU or shifter computation is driven on the Result Bus during  $\phi 2$  of *ALU*, and this is then either used as an address for a memory instruction or staged through the bypass registers to be written to the Register File. Access to the special registers, the PSW, the MD register and the PC registers is done through the ALU when they are being written, and by using the Result Bus when they are being read. For memory instructions, store data are driven to the pads during *MEM*  $\phi 2$  and load data are latched on the chip when *MEM*  $\phi 2$  falls. Register write-back occurs during  $\phi 1$  of *WB*. Instruction timings of all the different types of instructions are given in the Programmer's Manual[8].

## 4.4 Control Flow

The design of MIPS-X put great emphasis on making the control of the machine as simple as possible. Consistent with the philosophy of only implementing instructions in hardware that significantly improve performance, control features were only implemented if they were necessary to get the correct results or if they greatly improved performance. This is particularly true for handling exceptional conditions. Instead of trying to finish instructions that occur logically before an exception, such as was done in MIPS[35], the entire pipeline is frozen when an exception is detected and then restarted after the exception is serviced. To freeze the pipeline, shifting of the PC Chain is stopped and instructions in the pipeline are simply prevented from changing the state of the machine. Most of the hardware can continue to execute as if there were no exception, minimizing the number of places that need to know about the exception.

Control in MIPS-X is primarily done with logic that is local to each of the sections. There are a few global signals that travel through most of the chip:

**Reset** Puts the processor into a known state. This is controlled by an external pin.

**Exception** Precharged signal that connects to the External Interface, the Execute Unit and the PC Unit. It is pulled low by the External Interface when there is an unmaskable interrupt and by the Execute Unit when there is a maskable interrupt or a trap on overflow. The

PC Unit uses it to affect what is driven onto the PC Bus, as well as to generate some signals for the two global control finite state machines.

**Squash** Used during squashing branches and when an exception occurs. It is mainly used to prevent write-backs from occurring, and originates in the Squash finite state machine (Section 6.1.10).

**Cache Miss** Generated during Instruction Cache misses. It is generated in the Cache Miss finite state machine (Section 6.1.9).

The simplicity of the control of MIPS-X is demonstrated by the simplicity of the global controllers, which together use less than 0.2% of the total chip area. The global control is provided by two finite-state machines located above the datapath in the PC Unit. One is used to handle ICache miss sequencing, and the other one is used for instruction squashing. They are discussed in detail in Sections 6.1.9 and 6.1.10.

## 4.5 Implementation Issues

Before the physical layout of the processor could begin there were several global decisions that had to be made. This was very important because the design of the circuits was split among several people. In this section we describe the naming convention that we used for signal names and the basic plan used in designing the datapath.

### 4.5.1 The Naming Convention

To make sure that there would be some consistency in how the signals were labeled, we devised a naming convention very early in the design of the machine. We also had to consider the different tools that we planned to use and their restrictions regarding names. Our naming convention includes the use of a suffix that specifies the polarity, strength, and timing characteristics of the signal. Given the large number of signals in MIPS-X, we found this extremely helpful, especially in avoiding timing errors when using signals generated by different designers.

Names had to be a sequence of letters and digits starting with a letter, and they had to be case insensitive (although it was desirable to use case to make the name easier to read): An example might be *AdderGetPCBus*. The only special character allowed was the underscore (“\_”) character,



and this was used to append information. Global signals were named at the start: the power lines, *Vdd* and *Gnd*; the clock signals, *Phi1*, *Phi2*, *Psi1*; and the buses, such as *MemBus*, *AddressBus* and *PCBus*. Inverted signals had the characters “\_b” appended to the name, and then timing information was appended on the end. This was a two to four character code that started with an optional “p” to indicate a precharged signal. The next two characters were defined as follows:

s1	Stable $\phi 1$
s2	Stable $\phi 2$
v1	Valid $\phi 1$
v2	Valid $\phi 2$
q1	Qualified $\phi 1$
q2	Qualified $\phi 2$
dc	DC signal
w1	Weird $\phi 1$ (i.e., all signal types, other than the above that occur during $\phi 1$ )
w2	Weird $\phi 2$ (i.e., all signal types, other than the above that occur during $\phi 2$ )
ww	wishy-washy (i.e., all signals that do not fit in any of the above)

If a signal could be associated with a pipe stage then a third letter was appended:

i	IF
r	RF
a	ALU
m	MEM
w	WB

A *stable* signal is defined to be constant throughout the phase while a *valid* signal is one that can change during the phase but is guaranteed to be a constant logic value before and during the falling edge of the phase. A *qualified* signal is a signal that is logically ANDed with the clock phase indicated. Some ambiguity can arise. For example, a *v2* signal can also be a *s1* signal. In this case, *s1*, the *strongest* specification is specified.

Control lines that enabled bus drivers or receivers use the forms *objectDrvdestination* and *objectGetsource*. These correspond to an object, such as an adder, either driving (**Drv**) or getting (**Get**) a destination or source signal.

### 4.5.2 Datapath Design

The organization of the MIPS-X datapath can be seen in Figure 4.2. It is comprised of the the Register File, the Execute Unit, the PC Unit, and the Tags Unit. The central control is located in the PC Unit and provides the global signals (described in Section 4.4) that are used by the rest of the smaller controllers sitting above each of the units in the datapath.

The basic floorplan and wiring strategy for the MIPS-X datapath is quite simple. It uses metal 2 running horizontally for the buses, and for the power (*Vdd* and *Gnd*) lines. The control lines run vertically through the datapath in metal 1. The datapath power lines are connected at both ends of the datapath to large vertical power lines that are fed directly from power pins on the pad frame. The *Vdd* and *Gnd* lines are shared between two bit slices of the datapath. This is achieved by flipping every other bit slice. Figure 4.3 shows how this works using a simple latch as an example. The single cell is shown on the left, and this is flipped to share the *Vdd* lines as shown on the right. This combined cell can then be arrayed 16 times, such that the *Gnd* lines are shared, to achieve a 32-bit datapath. The datapath is somewhat unique in that it actually contains room for 33 bits. The thirty-third bit slot is located at the top of the datapath, right below the control line drivers. This bit slice was often left blank but was used on occasion to provide a delay element that matched some component in the datapath. For example, this slot was used in both the Register File and the ALU for a dummy cell used to generate a self-timed clock.

Placed above the datapath are the drivers for the datapath control signals. These drivers again use metal 2 running horizontally, this time to supply global clocks as well as power. The power lines for the control drivers again connect to the wide power buses on the right and left side of the datapath.

One of the first tasks was to generate a set of design rules to use for the layout. The chip was to be fabricated in a 2  $\mu\text{m}$  CMOS technology, but the design rules for this technology were not available when the project started. A worst-case rule file was generated<sup>1</sup> after a number

---

<sup>1</sup>This rule file ended up being a bit conservative, especially for contact surrounds – the diffusion surround was 2  $\mu\text{m}$  as was the metal 2 via surround. Since the diffusion contacts were so large, the minimum transistor size was chosen to be 6  $\mu\text{m}$  wide and 2  $\mu\text{m}$  long.

pcin1atch2.cif scale: 0.075000 (1905K) Size: 83 x 117 microns  
pcin1atch2

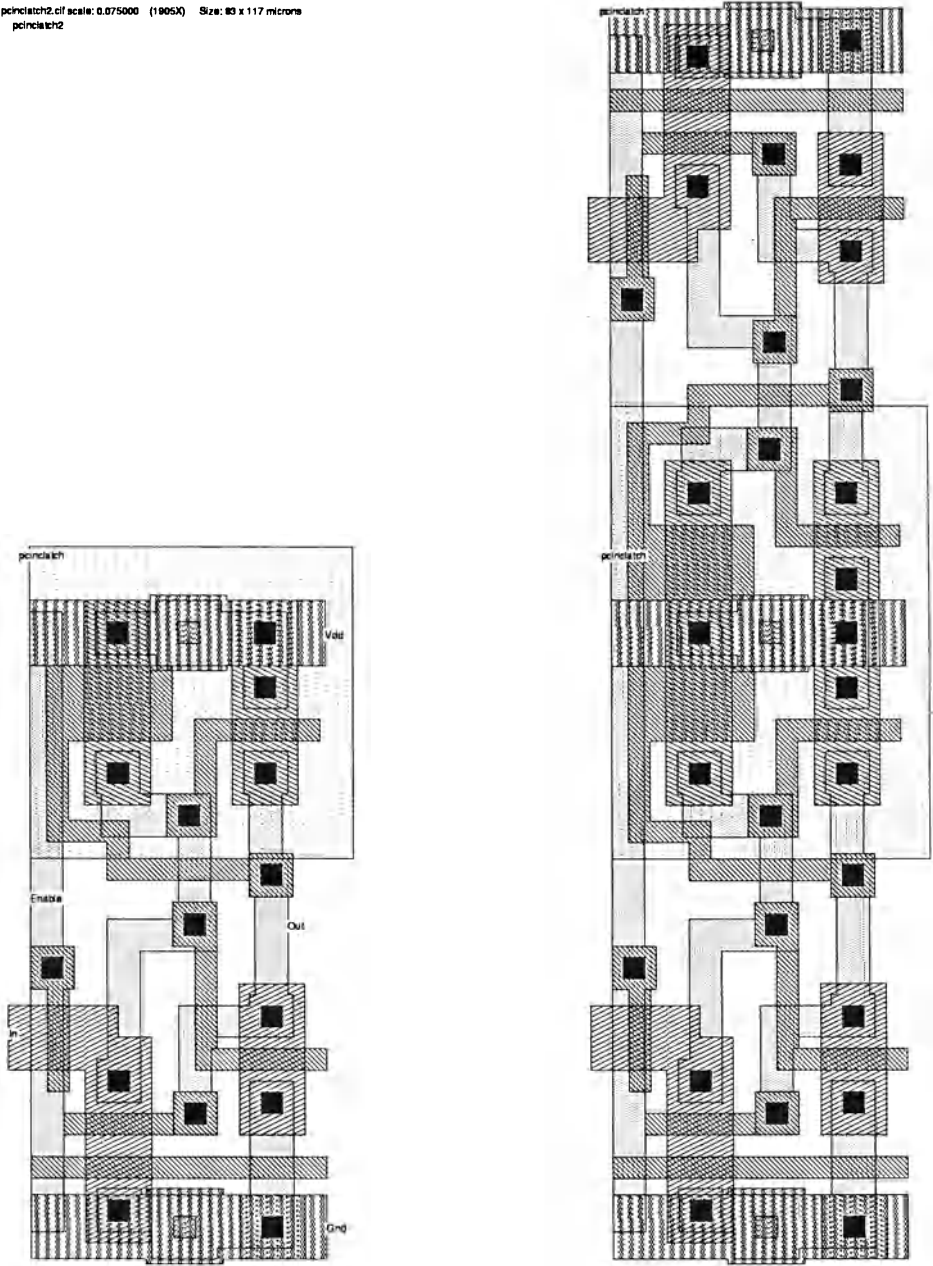


Figure 4.3: Datapath latch cell, and cell combined with flipped version.

of commercial rule files were compared. Once the design-rules were in place we were able to set the pitch of the datapath cells. This was primarily determined by the number of metal 2 tracks needed to carry the necessary buses over the cells. From Figure 4.2 it can be seen that only three bus tracks are needed to carry the major buses, but the pitch was chosen so that four tracks were available to provide room for some local routing in metal 2. Using our design rules the resulting pitch was  $55 \mu\text{m}$ .

At the beginning of the layout, a set of generic cells was provided to the designers. These cells included cells dealing with buses, latch cells and bus drivers, and a set of cells to drive the control lines into the datapath. In addition, since MIPS-X requires three adders (ALU, PC displacement adder, and PC incremter) one basic carry structure was designed and used in all three devices.

Which latch design to use was widely discussed. A static flow-through latch with a single n-channel access transistor was finally selected. Figure 4.4 shows a schematic of the cell, and Figure 4.3 shows the layout of the cell. Using a single n-channel access transistor avoided the need to have complementary versions of the gated clock lines running across the datapath, but did mean that some care was required because of the degraded high-level at the output of the n-channel pass transistor. To prevent the high-level from being degraded further, the feedback pull-down device is made weak by putting a transistor in the pull-down path that is biased only slightly on. This transistor, with its gate connected to a bias voltage called  $V_{bias}$ , acts as a very weak current source. The weak p-channel pull-up transistor serves two functions. It makes the latch static, and also restores the degraded high level to a full high value. To minimize area, all transistors in the latch are minimum size, except the weak p-transistor, which is  $6 \mu\text{m}$  wide and  $7 \mu\text{m}$  long.

There are two types of bus logic used in the datapath. The Result Bus is a precharged bus, and the other buses are driven with three-state drivers. A schematic for the Result Bus driver is shown in Figure 4.5. The layout of the Result Bus driver is shown in Figure 4.6. It has a pull-down path using two  $10 \mu\text{m}$  wide by  $2 \mu\text{m}$  long transistors. The standard three-state bus driver is shown in Figure 4.7. The 6-transistor pre-drive circuit is really the result of merging the NAND and NOR gates that are usually used to drive the output transistors. When the driver is enabled, the gates of the output devices are connected together and driven to the correct value, and when the driver is disabled, the gate of the n-channel



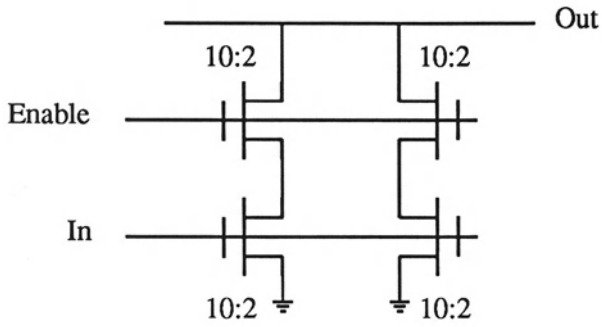


Figure 4.5: Schematic of Result Bus driver.

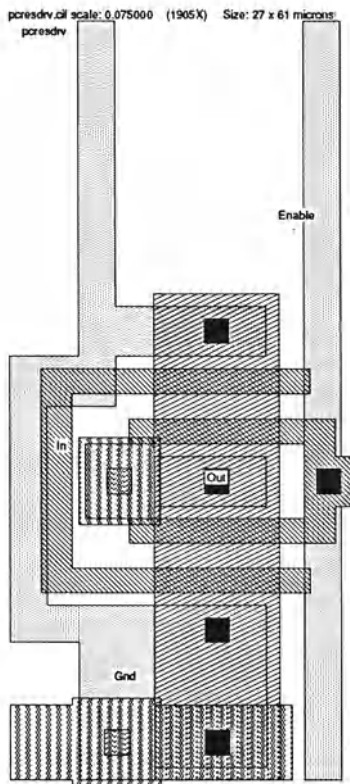


Figure 4.6: Layout of Result Bus driver.

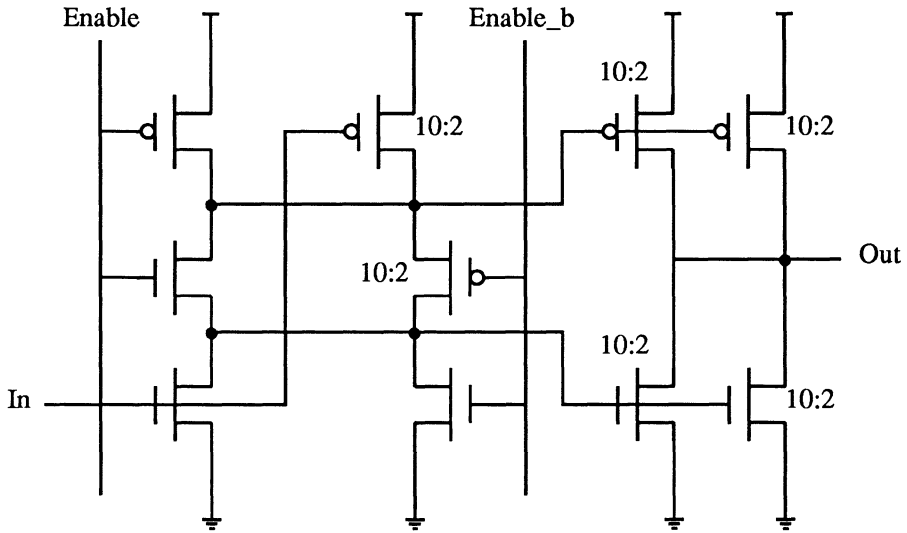


Figure 4.7: Schematic of three-state bus driver.

The only major circuit that is common to several parts of the datapath is the ALU/adder. The full ALU is used in the Execute Unit, while a simplified version is used twice in the PC Unit, once as the incrementer and once as the displacement adder. Section 5.2.8 gives a detailed description of the ALU.

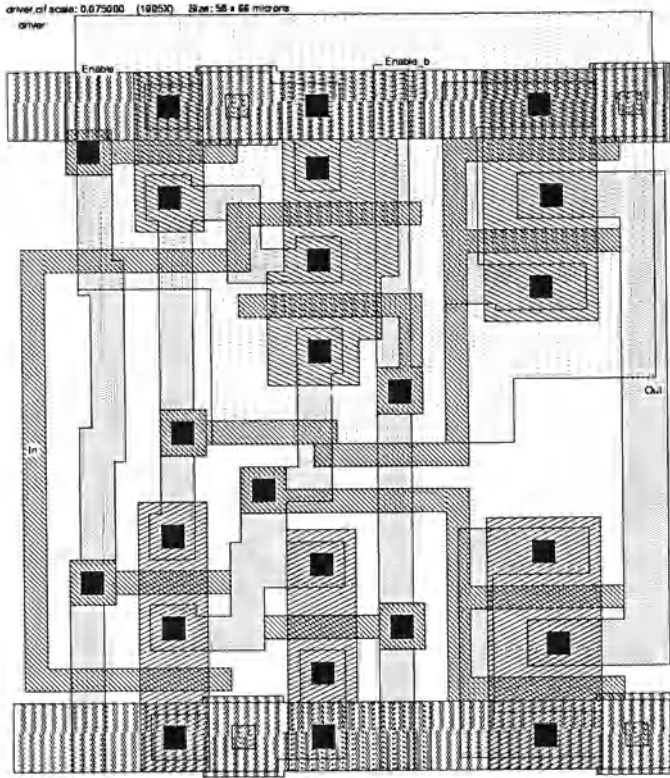


Figure 4.8: Layout of three-state bus driver.



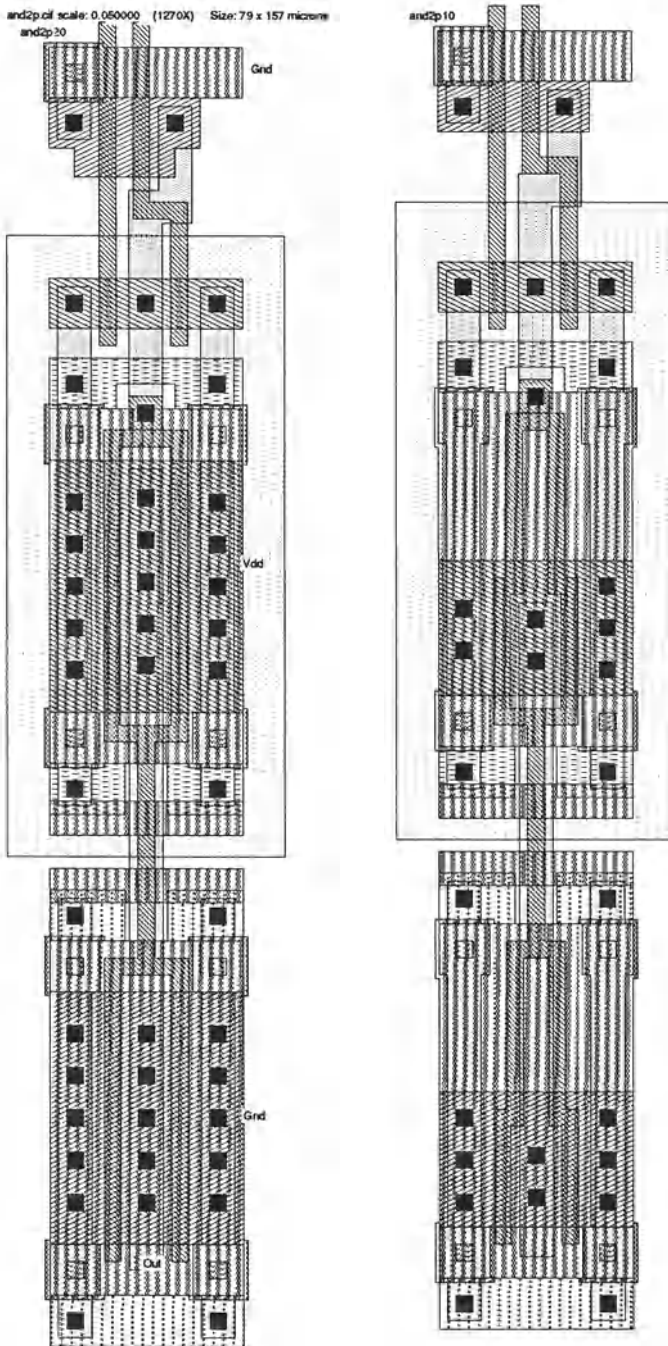


Figure 4.9: Layout of 2 *pf* and 1 *pf* AND control line drivers.

# Chapter 5

## The Execute Engine

The useful work of the machine is done in two major sections: the Register File and the Execute Unit. The Register File section contains the register memory cell array and the bypass network. The Execute Unit includes the shifter, the ALU, and registers to support multiplication and division. Together, these two units comprise less than 15% of the total chip area.

### 5.1 The Register File

The Register File (Regfile) provides the operands needed by the execution unit of MIPS-X. Logically it is a 32 word, triple-ported (dual read, single write) memory that contains a connection to the external memory system to support load and store instructions. For architectural reasons Register 0 stores the constant 0 and cannot be changed. The unit is made more complex because of the basic pipeline that MIPS-X uses: Regfile writes happen only at the end of the pipeline. Thus, the Regfile unit must also provide holding registers for the results that have not yet been written into the Regfile, and provide a way to bypass the Regfile when the following instruction uses these computed-but-not-yet-stored values. The unit is located at the left end of the datapath and it is about 2466  $\mu\text{m}$  high by 2245  $\mu\text{m}$  wide. Only about one half of the width is the actual RAM for the registers. The rest of the space is used to implement the result bypass registers and the interface to the memory bus.

The main functions of the Register File Unit are to:

- Provide a set of general purpose registers.

- Avoid pipeline stalls by allowing results to be used before they are written into the Regfile.
- Provide a path to load and store data from/to the external memory system.

Logically the Regfile can be broken up into eight main blocks as shown in Figure 5.1. The datapath is comprised of the Register Array and its associated read/write circuits, the memory data input register MDRin (which also acts as a bypass register), the memory data output registers MDRout, and the result bypass register, Result. The datapath connects to the Execute Unit through three buses: Src1Bus\_s1a, Src2Bus\_s1a, and ResultBus\_b\_pv2. The two source buses deliver operands to the Execute Unit and the Result Bus provides the results to be written into the Regfile. The Regfile communicates with the external memory system using the MemBus, which connects the memory data registers to the external data bus. RegControl sits above the datapath and receives the operand specifiers for the instruction from the Instruction Register, and the general pipeline control signals used by the rest of the machine. It contains the comparators needed to implement bypassing, and is responsible for controlling the datapath elements.

The following sections will describe the operation of the Regfile in more detail. The next section will describe the design of the actual Register Array, and will be followed by a description of the bypass registers and how they are controlled.

### 5.1.1 Register Array

The initial design of the register cell used dual-differential buses, but this was dropped because the short bit lines made sense amplifiers unnecessary. Instead we used a CMOS version of the 6-transistor RAM cell with split word lines described by Sherburne[39]. Figure 5.2 shows the CMOS cell. This cell can perform either two reads (word lines independently controlled) or one write (both word lines must address the same cell). To provide the three ports that MIPS-X requires, the Register Array is time multiplexed with writes occurring on  $\phi_1$  and reads on  $\phi_2$ . To reduce the access time, three sets of static decoders are placed above the Register Array, one for each access port. The inputs to the decoders are driven on the phase before their output is used so the decode time is not on

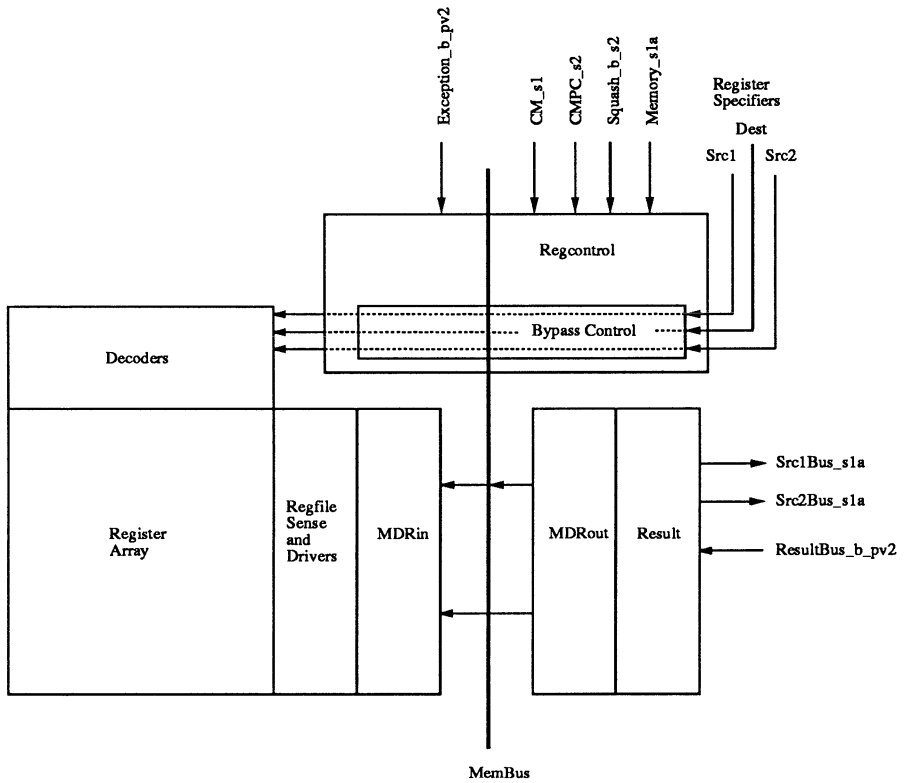


Figure 5.1: Register File block diagram.



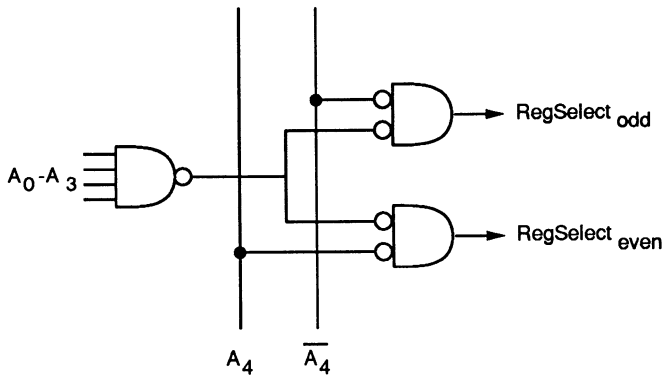


Figure 5.3: Register decoder.

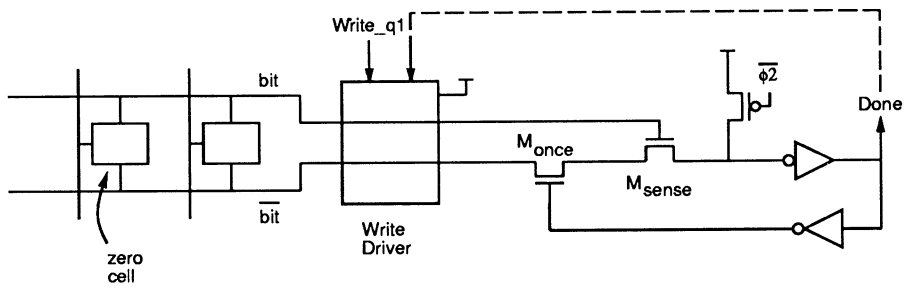


Figure 5.4: Self-timed write circuitry.

node  $\overline{Done}$ , causing  $Done$  to rise, and forcing the write drivers to recover the bit lines for the following read. Transistor  $M_{once}$  is needed to prevent the circuit from oscillating. If it is deleted, then the recovery of the  $\overline{bit}$  line will cause  $\overline{Done}$  to rise, and the write will restart. The write and recovery is quite fast, requiring less than 20 ns to complete.

The word-line drivers and the bit-line drivers into the Register Array also need to perform three different tasks – read, write, and precharge. These functions are generated by using slightly strange logic gates. The word-line driver posed an especially difficult design problem since two of the drivers were needed for each memory cell, and the cells were only  $36\mu\text{m}$  wide. The circuit that was used is shown in Figure 5.5. It is two tristate inverters connected together. When  $\Phi_2$  is high and Write is low the outputs of the Src1 and Src2 Decoders are driven onto the word lines (Src1Sel is driven onto wordline1, Src2Sel is driven onto wordline2). When Write is high and  $\Phi_2$  is low the output of the Destination Decoder (DestSel) is driven onto both word lines. When the write is complete and the circuit goes into recovery, Write becomes low and  $\Phi_2$  remains low. Write falling will cause all the word lines to go low, unless the just written word line will be read on the following cycle. In this case the selected word line remains high, but does not cause a problem since the cell is guaranteed to have the same value that was driven on the bit lines.

The bit-line driver is shown in Figure 5.6. Each bit of memory has two of these drivers, one for  $bit$  and one for  $\overline{bit}$ . This circuit is a slightly modified form of the standard tristate driver that is used many places in MIPS-X. When Write\_b is high and  $\Phi_2\_b$  is low the buffer is in a tristate mode, and the output is floating. When Write\_b is low and  $\Phi_2\_b$  is high the input value is driven onto the bit line. After the write has completed Write\_b will rise, which causes the output to be driven high independent of the input, performing the precharge stage. Notice that this circuit precharges the bit lines any time both Write\_b and  $\Phi_2\_b$  are both high, so some care is needed at the end of the read cycle. The machine must latch the values on the bit lines before  $\Phi_2\_b$  rises and causes the bit lines to be precharged.

The sensing of each bit line was done using a simple CMOS inverter that had a large p:n width ratio. The large p-channel device moved the switching point of the gate closer to  $V_{dd}$ , which slightly sped up the read path. Because the bit lines were fairly short we did not need a more complex sensing scheme.

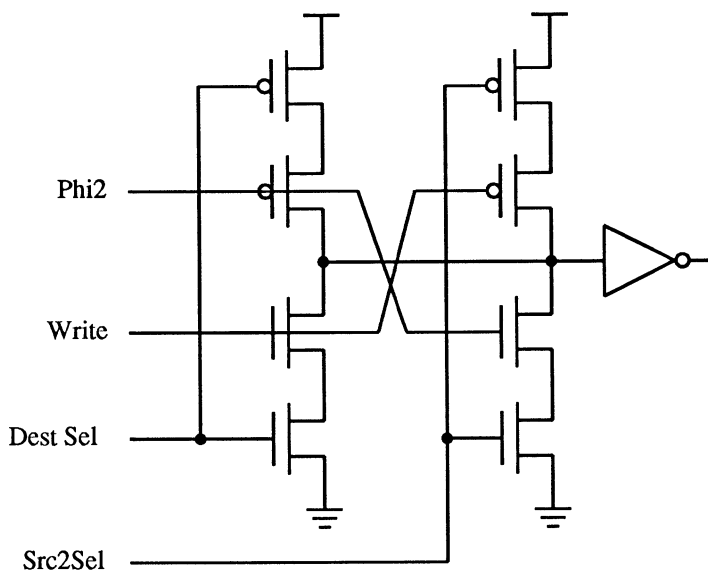


Figure 5.5: Register word-line driver.

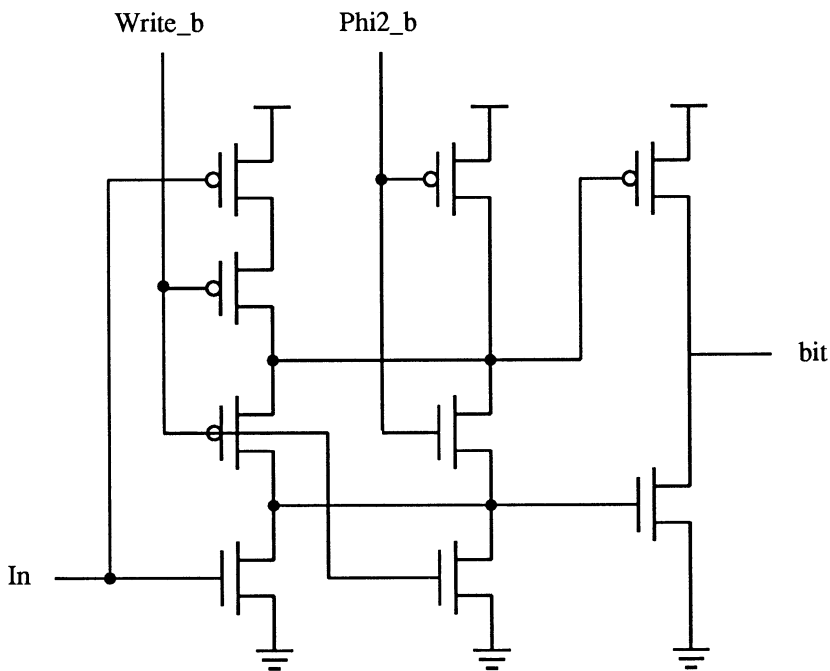


Figure 5.6: Register File bit-line driver.



### 5.1.2 Bypass Registers and Control

ALU results are generated on  $\phi 2$  of ALU, the third machine pipe stage, and are not written into the Register Array until  $\phi 1$  of WB. This means that the Regfile must contain a two-stage shift register to delay the values being written into the Register Array. The destination specifier must also be delayed, since it is driven into the Regfile on  $\phi 2$  of RF and again is not used until  $\phi 1$  of WB. During the time the Regfile is holding the data it is responsible for keeping track of whether the instructions that generated the data were squashed (either from a mispredicted branch or from an exception). If the instruction was squashed it should not affect the user visible state of the machine and thus should not be written into the Register Array. This no-oping of the instruction is performed by appending a single bit to the write destination address. This sixth bit, called NoDest, is set to indicate that this instruction is not to write into the Register Array. This bit is set in the Instruction Register if the instruction does not generate a result (Stores, Branches, etc.), and is set in the Regfile when Squash or Exception are asserted.

To remove some of the constraints caused by delaying the Regfile writes until the last cycle, the Regfile provides bypass paths to use the results of previous instructions before they are written into the Register Array. This bypassing is implemented by using four 6-bit comparators<sup>1</sup> in RegControl as shown in Figure 5.7. During  $\phi 2$  of RF, the comparators check the two current register sources against the destinations of the previous two instructions. In the case of a match, the data in the temporary latch is gated onto the appropriate SrcBus, rather than the value that was fetched from the Register Array. In the case of multiple matches (the last two instructions used the same destination register), the newest value is used.

The bypass comparator logic is shown in Figure 5.8. To avoid requiring both the true and complement forms of the comparison data, both an n-channel and p-channel precharged structure is used for each bit. If the data matches, both precharged nodes will remain unchanged, but if there is a mismatch the output of the n-channel gate will fall and/or the output of the p-channel gate will rise. Inverting the output of the

---

<sup>1</sup>In addition to the 5-bit address, the NoDest bit is compared to make sure that the latch contains valid data before the data is bypassed. This also makes it possible not to bypass data that is written to Register 0. The Instruction Register always sets the NoDest bit when the destination address is 0.

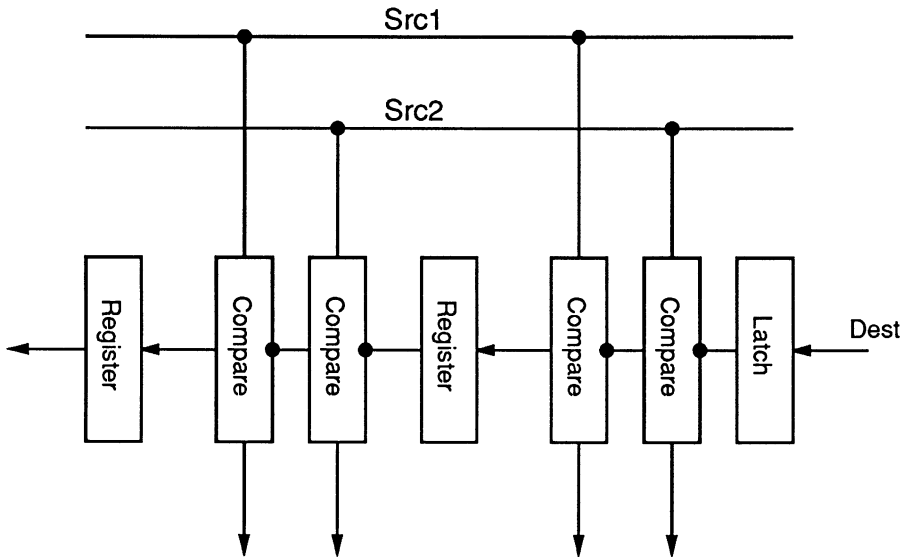


Figure 5.7: Bypass comparators.

n-channel gate and then NORing the outputs together gives the desired match signal. Using this structure allowed the bypass comparators to be quite small. They fit under the wires that contain the register specifiers.

### 5.1.3 Datapath

A schematic of the entire datapath in the Regfile (not including the Register Array itself) is shown in Figure 5.9. Starting with the right most part of the datapath is the Result Register, which latches the value on the ResultBus. To support bypassing, two tristate drivers can be enabled by bypass comparators to drive this value onto either SrcBus when the current instruction uses the result of the immediately preceding instruction.

Moving slightly to the left, the next set of latches make up the Memory Data Output Register (MDRout). These latches delay the data from the Src2Bus, which is latched on  $\phi_2$  of RF<sup>2</sup>, and delayed until the MEM cycle when it is driven out onto the MemBus. This delayed value is the

<sup>2</sup>This is really a bug. The first latch is redundant and should be removed. The bus is stable  $\phi_1$ , and should be latched by a Psi1 latch

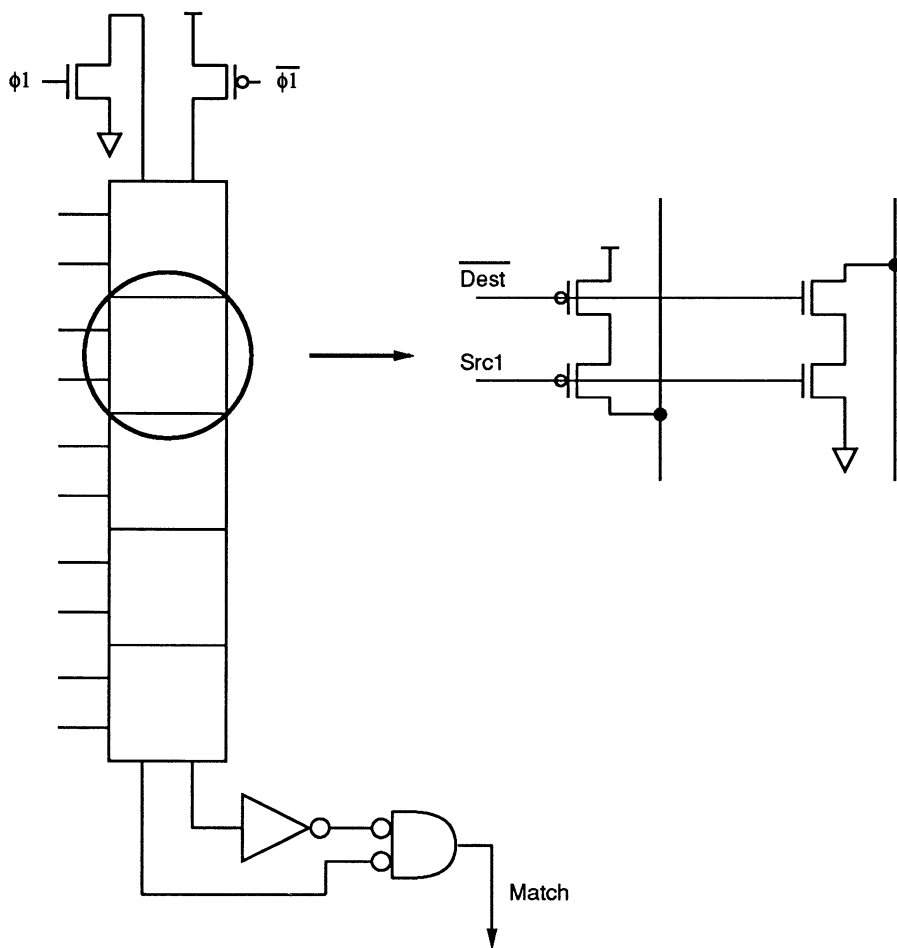
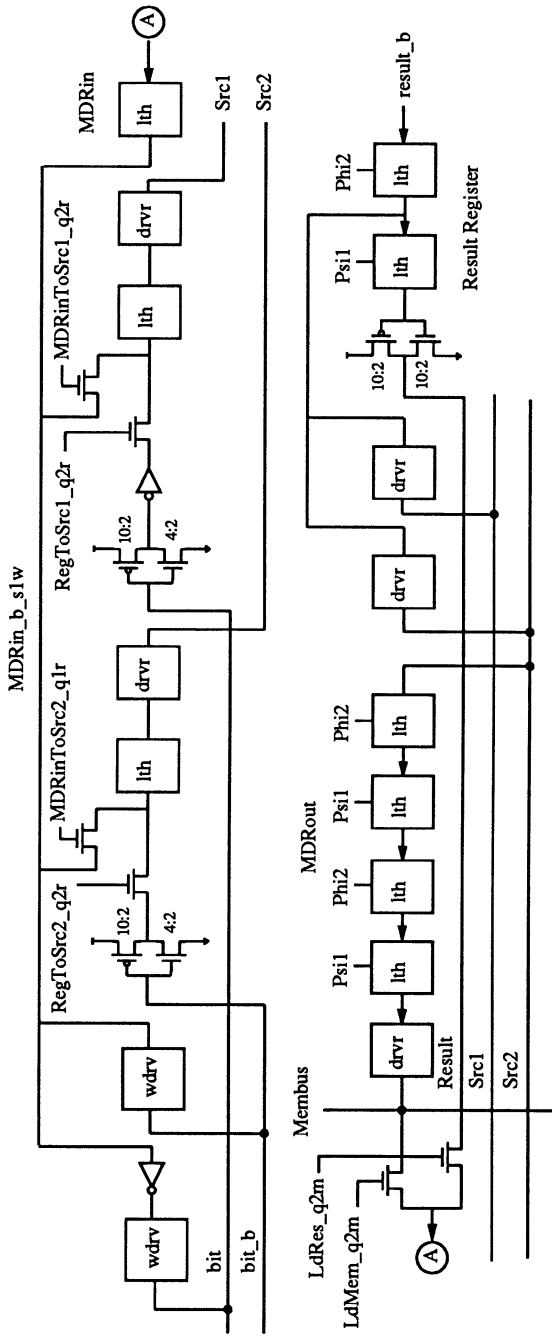


Figure 5.8: Bypass comparator logic.



wdrv is a write driver  
 drv is a noninverting driver  
 lth is an inverting latch

Figure 5.9: Overall schematic of Register File datapath logic

input to a tristate driver that drives the MemBus in the MEM cycle of a store.

The next latch in the datapath is the Memory Data Input Latch (MDRin). It serves two functions. In the case of load instructions MDRin serves as the input latch for the memory data. During  $\phi_2$  of the MEM cycle the value on the MemBus is latched and the output of this latch is used to drive the bit-line write drivers (wdrv) on the next clock phase. For normal ALU instructions LdRes\_q2m is asserted during  $\phi_2$  of MEM, and the latch is used as the second part of the delay chain for ALU results. So for either load or ALU operations the result of the operation is in MDRin by the end of the MEM cycle.

The second set of bypass comparators are used to select whether the data from the Register Array is latched or whether the data from MDRin should be used instead. This simple muxing was used instead of two additional tristate drivers since it occupied much less space, though it does slow down the bypass path for just-loaded data by a couple of nanoseconds.

## 5.2 The Execute Unit

The Execute Unit contains the hardware that is used to do the actual operations on data. The unit is sandwiched between the Register File and the PC Unit in the center of the datapath. It occupies about  $1800 \mu\text{m}$  of the datapath's full  $7440 \mu\text{m}$  length and it is about  $2370 \mu\text{m}$  high. This includes about  $530 \mu\text{m}$  for the height of the controllers that sit above the datapath and the large drivers that drive the control signals through the datapath. The main hardware in the Execute Unit includes:

- A 64-bit to 32-bit funnel shifter.
- The multiply/divide (MD) register to support multiplication and division.
- A 32-bit ALU.
- The processor status word (PSW).
- Circuitry to detect arithmetic overflow and generate branch conditions.

The purpose of this section is to describe the hardware associated with each piece of the unit, including both the datapath circuitry and the accompanying controllers.

### 5.2.1 Interface

The signals connected to the Execute Unit are shown in Table 5.1, and their approximate location in the layout is shown in Figure 5.10.

The functions performed by each signal are:

**Src1Bus\_s1a** Bus on which the Src1 operand is delivered from the Regfile to the Execute Unit.

**Src2Bus\_s1a** Bus on which the Src2 operand is delivered from the Regfile to the Execute Unit.

**ResultBus\_b\_pv2a** Bus that carries an operation's result back to the Regfile. Depending on the instruction, this bus may be driven by either the Shifter or the ALU, as well as by several sources in the PC Unit. This is the Result Bus used by the Regfile and Execute Unit.

**ResultBusPC\_b\_pv2a** Same as ResultBus\_b\_pv2a except in the most significant bit (bit 0). This is the Result Bus used by the PC Unit and the Tags Unit. The difference in the most significant bits is explained in Section 5.2.12.

**ALUResult\_s2a** The 32-bit result of the ALU operation performed during  $\phi_1$ . Depending on the instruction, this may or may not be a useful result.

**ImmedBus\_s1a** The 17-bit immediate bus, used for carrying constants encoded in instructions to the Execute and PC Units. Used as one of the ALU sources for memory and compute immediate instructions.

**Squash\_b\_s2** Active during the ALU cycle of two instructions to be squashed after a branch instruction, and during the two cycles following a cycle in which Exception\_b\_pv2 is active. Used in both the MReg and PSW to prevent squashed instructions from changing state.

signal	bits	timing		direction	connected to Unit
Src1Bus_s1a	32	stable	$\phi 1$	input	Regfile
Src2Bus_s1a	32	stable	$\phi 1$	input	Regfile
ResultBus_b_pv2a	32	valid	$\phi 2$	output	Regfile
ResultBusPC_b_pv2a	32	valid	$\phi 2$	output	PC Unit
ALUResult_s2a	32	stable	$\phi 2$	output	PC Unit
ImmedBus_s1a	17	stable	$\phi 1$	input	IReg
Squash_b_s2	1	stable	$\phi 2$	input	PC Unit
Interrupt_s2	1	stable	$\phi 2$	input	Eface
Exception_b_pv2	1	valid	$\phi 2$	bidirectional	PC Unit
Reset_s2	1	stable	$\phi 2$	input	Eface
Memory_s1a	1	stable	$\phi 1$	input	IReg
Branch_s1a	1	stable	$\phi 1$	input	IReg
Compute_s1a	1	stable	$\phi 1$	input	IReg
ComputeImmed_s1a	1	stable	$\phi 1$	input	IReg
Movtos_s1a	1	stable	$\phi 1$	input	IReg
Movfrs_s1a	1	stable	$\phi 1$	input	IReg
Jpcrs_s1a	1	stable	$\phi 1$	input	IReg
TrapInst_s1a	1	stable	$\phi 1$	input	IReg
MDInst_s1a	1	stable	$\phi 1$	input	IReg
Op0Type1_s1a	1	stable	$\phi 1$	input	IReg
TBWhich_s1a	1	stable	$\phi 1$	input	IReg
TBEqChk_s1a	1	stable	$\phi 1$	input	IReg
TBGrChk_s1a	1	stable	$\phi 1$	input	IReg
TBHighChk_s1a	1	stable	$\phi 1$	input	IReg
ShiftDrvRes_s1a	1	stable	$\phi 1$	input	IReg
ShiftDrvResult_s2a	1	stable	$\phi 2$	input	IReg
TakeBranch_s2a	1	stable	$\phi 2$	output	PC Unit
PSWs_b_s2	1	stable	$\phi 2$	output	PC Unit
PSWu_b_s2	1	stable	$\phi 2$	output	PC Unit
PCM4DrvResBus_q2	1	qualified	$\phi 2$	input	PC Unit
IncDrvResBus_q2	1	qualified	$\phi 2$	input	PC Unit
Psi1	1	clock	$\phi 1$	input	Eface
Phi2_b	1	clock	$\phi 2$	input	Eface
Phi2	1	clock	$\phi 2$	input	Eface
Phi1	1	clock	$\phi 1$	input	Eface

Table 5.1: Signals connected to the Execute Unit.

- Interrupt\_s2** Used to indicate an external interrupt to the Execute Unit, which will activate Exception\_b\_pv2 if interrupts are not masked.
- Exception\_b\_pv2** Active for an external interrupt, external exception, or overflow (ALU or MD Register). Pulled by the Execute Unit in the case of an external interrupt or overflow. Pulled by the Eface in the case of an external exception.
- Reset\_s2** Active during machine reset. Used by the PSW to ensure that the machine comes up after reset with the PSW bits in a reasonable configuration.
- Memory\_s1a** Active during the ALU cycle of all memory instructions.
- Branch\_s1a** Active during the ALU cycle of all branch instructions.
- Compute\_s1a** Active during the ALU cycle of all compute instructions.
- ComputeImmed\_s1a** Active during the ALU cycle of all compute immediate instructions.
- Movtos\_s1a** Active during the ALU cycle of a *movtos* (Move To Special) instruction.
- Movfrs\_s1a** Active during the ALU cycle of a *movfrs* (Move From Special) instruction.
- Jpcrs\_s1a** Active during the ALU cycle of a *jpcrs* (Jump PC and Restore State) instruction.
- TrapInst\_s1a** Active during the ALU cycle of a *trap* (Trap Unconditionally) instruction.
- MDInst\_s1a** Active during the ALU cycle of a *dstep* (Divide Step), *mstep* (Multiply Step), or *mstart* (Multiply Startup) instruction.
- Op0Type1\_s1a** Active during the ALU cycle of any instruction that is encoded with a 1 in bit 0 of the Opcode field and bit 1 of the Type field. These are the instructions in which the ALU always drives the Result Bus.



**TBWhich\_s1a** TakeBranchWhich signal. Only contains a useful value during the ALU cycle of a branch instruction. Indicates the sense of the branch condition encoded in the instruction. Same as bit 0 of the Opcode field in the branch instructions. Used by TakeBranch logic, see Section 5.2.12.

**TBEqChk\_s1a** TakeBranchEqualCheck signal. Only contains a useful value during the ALU cycle of a branch instruction. Active if the instruction indicates the equality condition is to be tested, i.e., the instruction is *beq* or *bne*. Used by TakeBranch logic, see Section 5.2.12.

**TBGrChk\_s1a** TakeBranchGreaterThanCheck signal. Only contains a useful value during the ALU cycle of a branch instruction. Active if the instruction indicates the (signed) greater-than condition is to be tested, i.e., the instruction is *bge* or *blt*. Used by TakeBranch logic, see Section 5.2.12.

**TBHighChk\_s1a** TakeBranchHigherThanCheck signal. Only contains a useful value during the ALU cycle of a branch instruction. Active if the instruction indicates the (unsigned) higher-than condition is to be tested, i.e., the instruction is *bhs* or *blo*. Used by TakeBranch logic, see Section 5.2.12.

**ShiftDrvRes\_s1a** Active during the ALU cycle of those instructions in which the shifter drives the Result Bus, i.e., the *asr*, *rotlcb*, *rotlcb*, and *sh* instructions. Used in the shifter controller to ensure that no more than one output of the decoders is active during instructions that do not use the shifter.

**ShiftDrvResult\_s2a** Same as ShiftDrvRes\_s1a, but delayed by one phase. Used to gate the shifter output onto the Result bus.

**TakeBranch\_s2a** Only contains a useful value during the ALU cycle of a branch instruction. Indicates to the PC Unit whether or not the branch should be taken.

**PSWs\_b\_s2** Current value of the PSW S (PC Shifting) bit.

**PSWu\_b\_s2** Current value of the PSW U (User/System) bit.

**PCM4DrvResBus\_q2** Active when PC-4 in the PC Unit is driving the Result Bus. See Section 5.2.12.

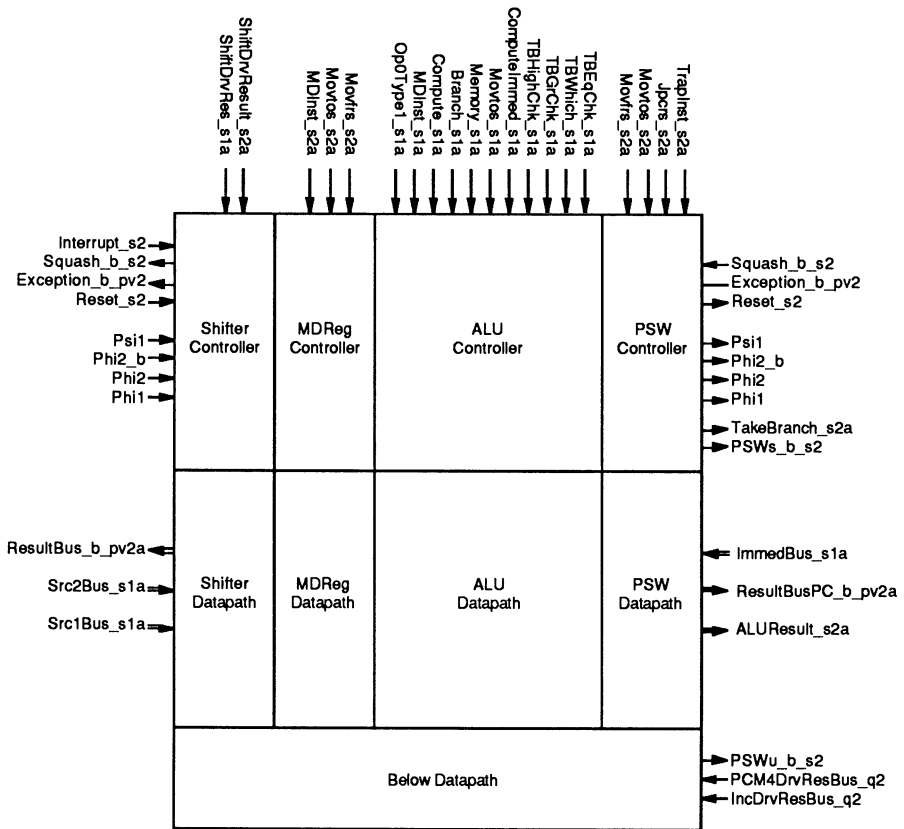


Figure 5.10: Interface to the Execute Unit.

**IncDrvResBus\_q2** Active when the PC incremter in the PC Unit is driving the Result Bus. See Section 5.2.12.

### 5.2.2 Timing

The Execute Unit performs all of its operations for a particular instruction during the ALU cycle of that instruction. With only a few exceptions, all signals that flow into the Execute Unit from other parts of the processor are stable by the end of  $\phi_2$  of the RF cycle, and remain stable throughout  $\phi_1$  of the ALU cycle. All 32-bit data that are generated by the unit are placed on the Result Bus by the end of  $\phi_2$  of the ALU cycle.

### 5.2.3 PLA Controllers

A PLA-style structure is used within the Execute Unit in several places to implement the logic for the control lines that run through the datapath. This structure is described once here, and will simply be referred to as a PLA throughout the remainder of this section. Since it appeared that many of the desired control signals contained few minterms, the decision was made to use a precharged AND-plane with wired-OR outputs. A schematic of this scheme is shown in Figure 5.11. During the precharge phase, each output from the PLA is precharged. In addition, all of the input signals into the PLA must be stable by the end of this phase to prevent charge-sharing problems. During the evaluation phase, each line corresponding to a minterm that evaluates true is discharged. If any of the minterms connected to an output is discharged, then the output is discharged, thus realizing the wired-OR output. Note that all outputs from this style of PLA are qualified with the evaluation clock, meaning that the signal is effectively ANDed with the clock, i.e., the signal is always low when the clock is low. This property was convenient for most of the signals, and often prevented having to run clock signals through the datapath as additional control lines.

### 5.2.4 Funnel Shifter

The funnel shifter extracts any contiguous 32-bit field from the concatenation of two 32-bit inputs. All standard shifts (logical shifts, arithmetic shifts, and rotates) are easily accomplished using this scheme. The shifter occupies  $650 \mu\text{m}$  of the datapath length. The actual shift is done and the result latched during  $\phi_1$ . The result is driven onto the Result Bus during  $\phi_2$ .

A schematic diagram of the shifter is shown in Figure 5.12. It is built in two stages. The first stage uses the three high-order bits of the shift amount to select a 35-bit field out of the 64-bit input. The second stage then uses the two low-order bits of the shift amount (which happen to be fully decoded into four bits in the shift instructions) to select the desired 32-bit field from the 35 bits. This two-stage organization was chosen to reduce the size of the shifter by reducing the number of decoded select lines running through the datapath from 32 (for a single-stage shifter) to 12 (8 for the first stage, 4 for the second). Note that some additional complexity is introduced since the datapath must be 35 bits high in the region of the first stage rather than the standard 32-bit height.

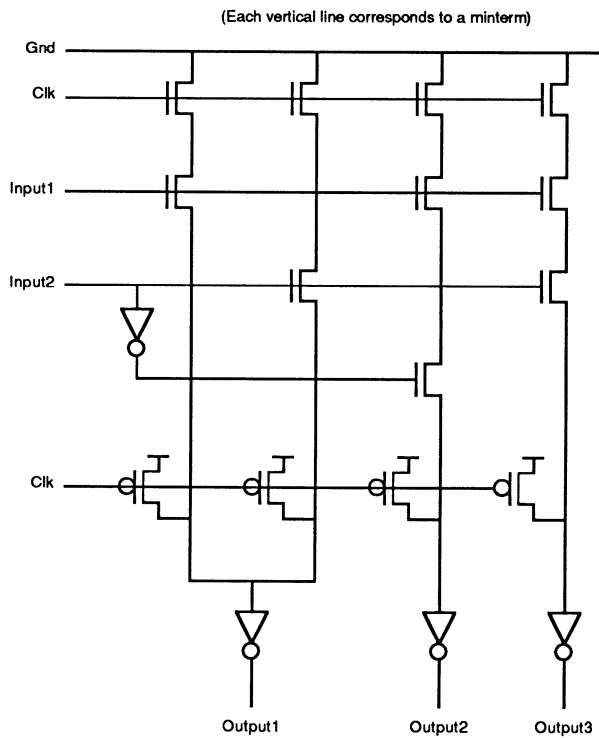


Figure 5.11: PLA schematic.

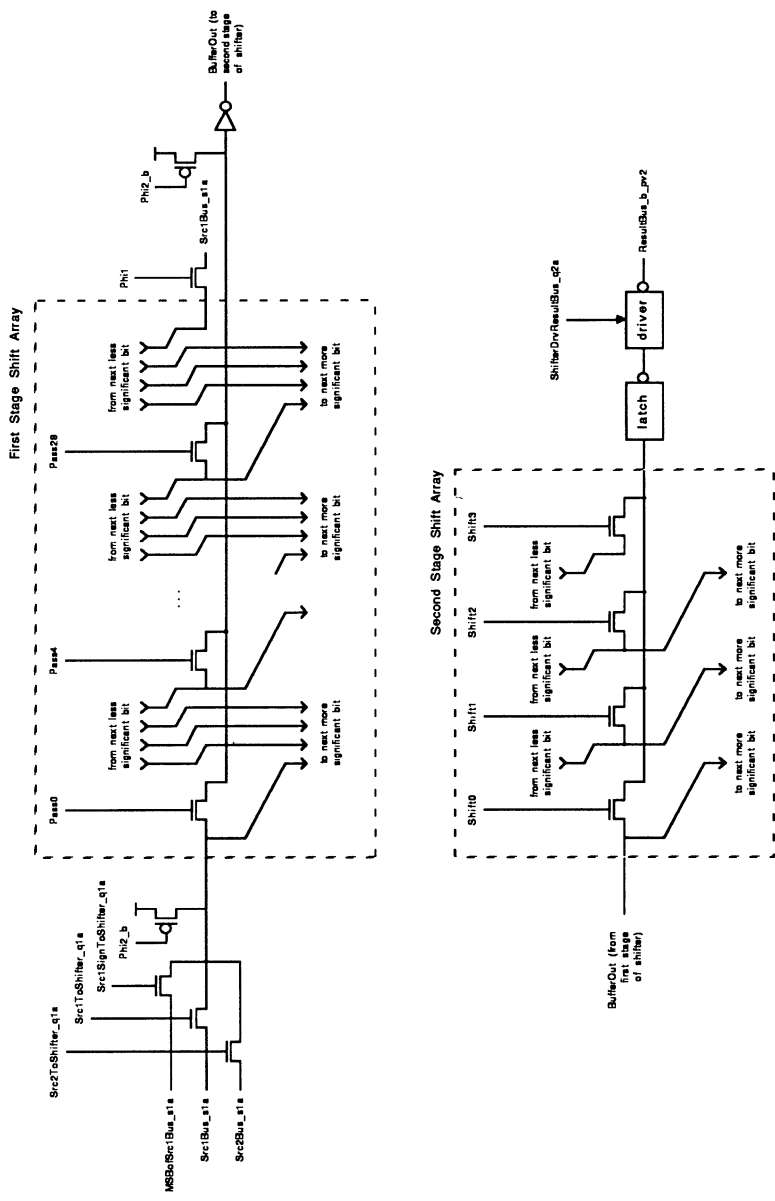


Figure 5.12: Shifter schematic.

A few points about the circuit issues should be mentioned (refer to the schematic, Figure 5.12). There is an input multiplexer that allows the controller to set up the shifter for a funnel shift, rotate, or arithmetic right shift. For the arithmetic shift right, there is a large driver at the bottom of the datapath to drive the most significant bit (i.e., the sign bit) of the Src1 input up through the datapath. This bit, driven into each of the bits in the datapath, can then be used as one of the two 32-bit inputs into the shifter, allowing the arithmetic right shift to be performed. Note that the first stage of the shifter is built as a large discharge network, meaning that all of the internal lines are precharged during  $\phi_2$  and then discharged selectively during  $\phi_1$  to yield the desired result.

### 5.2.5 Funnel Shifter Controller

The funnel shifter controller is quite straightforward. The control lines that must be generated include the input multiplexer lines, the decoded shift amounts for the two shift stages, and the line that indicates that the shifter should drive the Result Bus. The following sections discuss each of the signals generated by the controller.

#### Input Multiplexer Signals

##### Signals<sup>3</sup>:

Src1ToShifter\_q1a = CF24 · ShifterDrvResultBus\_q1a

Src2ToShifter\_q1a = CF22 · ShifterDrvResultBus\_q1a

Src1SignToShifter\_q1a = CF23 · ShifterDrvResultBus\_q1a

One of the 32-bit inputs into the shifter is the Src1 register. The other 32-bit input is selected by the input multiplexer; this input determines the type of shift (funnel shift, rotate, or arithmetic shift) that will be performed. These signals come directly from the Comp Func field of the shifter instructions, and thus may be pulled directly from the Immediate Bus. However, no more than one of these signals should be high at one time, so each one is ANDed with ShifterDrvResultBus\_q1a. This ensures that the signals will only be active for shift instructions. This ANDing is done in the control line drivers just before the lines enter the datapath.

---

<sup>3</sup>CF $nn$  denotes a bit in the CompFunc field corresponding to bit  $nn$  in the instruction word

### First Stage Decoded Shift Amounts

#### Signals:

$$\text{Pass0} = \overline{\text{CF24}} \cdot \overline{\text{CF25}} \cdot \overline{\text{CF26}} \cdot \overline{\text{CF27}} \\ \vee \text{CF24} \cdot \text{CF25} \cdot \overline{\text{Src2}[30]} \cdot \overline{\text{Src2}[31]}$$

$$\vee \text{CF24} \cdot \overline{\text{CF25}} \cdot \overline{\text{Src2}[30]} \cdot \overline{\text{Src2}[31]}$$

$$\text{Pass4} = \overline{\text{CF24}} \cdot \overline{\text{CF25}} \cdot \overline{\text{CF26}} \cdot \overline{\text{CF27}}$$

$$\text{Pass8} = \overline{\text{CF24}} \cdot \overline{\text{CF25}} \cdot \overline{\text{CF26}} \cdot \overline{\text{CF27}}$$

$$\vee \text{CF24} \cdot \text{CF25} \cdot \overline{\text{Src2}[30]} \cdot \overline{\text{Src2}[31]}$$

$$\vee \text{CF24} \cdot \overline{\text{CF25}} \cdot \overline{\text{Src2}[30]} \cdot \overline{\text{Src2}[31]}$$

$$\text{Pass12} = \overline{\text{CF24}} \cdot \overline{\text{CF25}} \cdot \overline{\text{CF26}} \cdot \overline{\text{CF27}}$$

$$\text{Pass16} = \overline{\text{CF24}} \cdot \overline{\text{CF25}} \cdot \overline{\text{CF26}} \cdot \overline{\text{CF27}}$$

$$\vee \text{CF24} \cdot \text{CF25} \cdot \overline{\text{Src2}[30]} \cdot \overline{\text{Src2}[31]}$$

$$\vee \text{CF24} \cdot \overline{\text{CF25}} \cdot \overline{\text{Src2}[30]} \cdot \overline{\text{Src2}[31]}$$

$$\text{Pass20} = \overline{\text{CF24}} \cdot \overline{\text{CF25}} \cdot \overline{\text{CF26}} \cdot \overline{\text{CF27}}$$

$$\text{Pass24} = \overline{\text{CF24}} \cdot \overline{\text{CF25}} \cdot \overline{\text{CF26}} \cdot \overline{\text{CF27}}$$

$$\vee \text{CF24} \cdot \text{CF25} \cdot \overline{\text{Src2}[30]} \cdot \overline{\text{Src2}[31]}$$

$$\vee \text{CF24} \cdot \overline{\text{CF25}} \cdot \overline{\text{Src2}[30]} \cdot \overline{\text{Src2}[31]}$$

$$\text{Pass28} = \overline{\text{CF24}} \cdot \overline{\text{CF25}} \cdot \overline{\text{CF26}} \cdot \overline{\text{CF27}}$$

The high-order three bits (CF25, CF26, and CF27) of the shift amount are decoded into these eight control lines. There is a PLA dedicated to this decoding. The only complication in this is that the MIPS-X architecture specifies byte rotate instructions with the rotate amount specified not in the instruction encoding, but rather in the Src1 register. Therefore, the inputs to the decoder PLA include both the shift amount bits from the Comp Func field (obtained from the Immediate Bus) and the low-order two bits of the Src1 Bus pulled up from the datapath. The PLA has 12 inputs (including both true and complemented signals), eight outputs, and 16 product terms.

### Second Stage Decoded Shift Amounts

#### Signals:

$$\text{Shift0} = \overline{\text{CF24}} \cdot \text{CF31} \vee \text{CF24}$$

$$\text{Shift1} = \overline{\text{CF24}} \cdot \text{CF30} \cdot \text{ShifterDrvResultBus}_s1a$$

$$\text{Shift2} = \overline{\text{CF24}} \cdot \text{CF29} \cdot \text{ShifterDrvResultBus}_s1a$$

$$\text{Shift3} = \overline{\text{CF24}} \cdot \text{CF28} \cdot \text{ShifterDrvResultBus}_s1a$$

The low-order two bits of the shift amount are decoded into these

four control lines. As in the case of the first stage decoding, there is a PLA dedicated to this decoding. In the case of the byte rotate instructions mentioned above, the PLA forces the four lines to the same values they would have if the low-order two bits of the shift amount were 00. This is because the second stage of the shifter is not really needed in the case where the shift amount is a multiple of four, as is the case with byte rotates. The PLA has seven inputs (including both true and complemented inputs), four outputs, and five product terms.

### Driving the Result Bus

#### Signals:

$$\text{ShifterDrvResultBus\_q2a} = \overline{\text{Type0}} \cdot \text{Type1} \cdot \overline{\text{Op0}} \cdot \overline{\text{Op1}} \cdot \text{Op2}$$

This is the control signal that determines whether or not the Result Bus is driven with the shifter's 32-bit output. Note that unlike the other shifter control signals (which are all  $\phi1$  signals), this is a  $\phi2$  signal. This signal is only activated for shift instructions. The shift instructions are detected by decoding the Type and Op fields of the instruction; this takes place in the IReg during  $\phi1$  of the ALU phase. The IReg provides ShifterDrvResultBus\_s1a to the shifter controller to generate the input multiplexer signals (see above). The IReg also latches the signal and provides it to the shifter controller as ShifterDrvResultBus\_s2a. The shifter controller then needs only to perform the clock qualification with  $\phi2$  to generate ShifterDrvResultBus\_q2a.

### 5.2.6 MD Register

The MD (multiply/divide) register is a special register with built-in hardware to facilitate the implementation of the shift-and-add multiply and divide instructions specified by the MIPS-X architecture. The MD register occupies 205  $\mu\text{m}$  of the datapath length.

A schematic of one bit of the MD register is shown in Figure 5.13. Each bit consists of an input multiplexer feeding a string of latches. Selecting different inputs allows the MD register to be written from the Result Bus, shifted left by one bit, or restored to a previous value.

Since the MD register comprises part of the MIPS-X machine state, special care must be taken to ensure that the correct value is stored in the case of an exception or squashed instructions. The easiest way to see how this is handled is to look at a pipeline diagram. A pipeline diagram



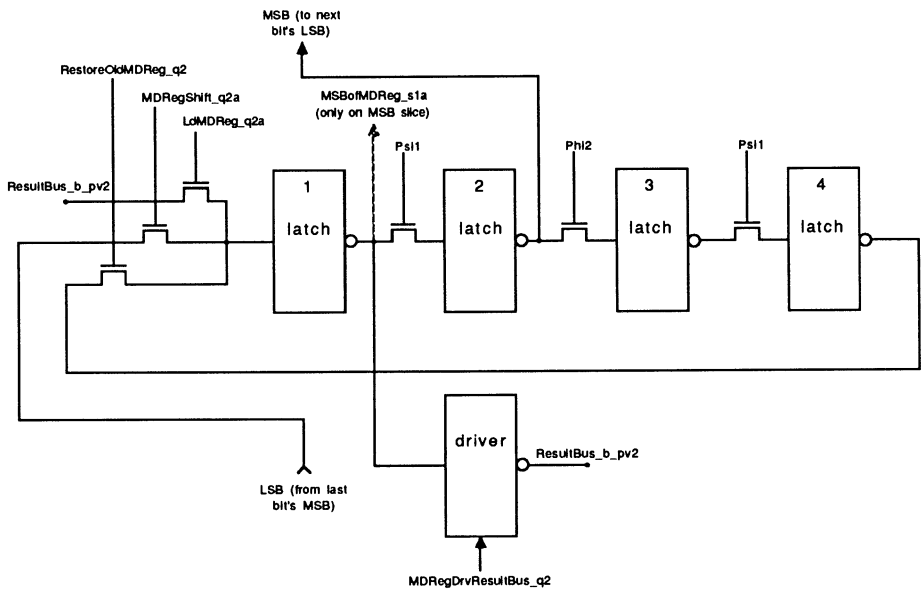
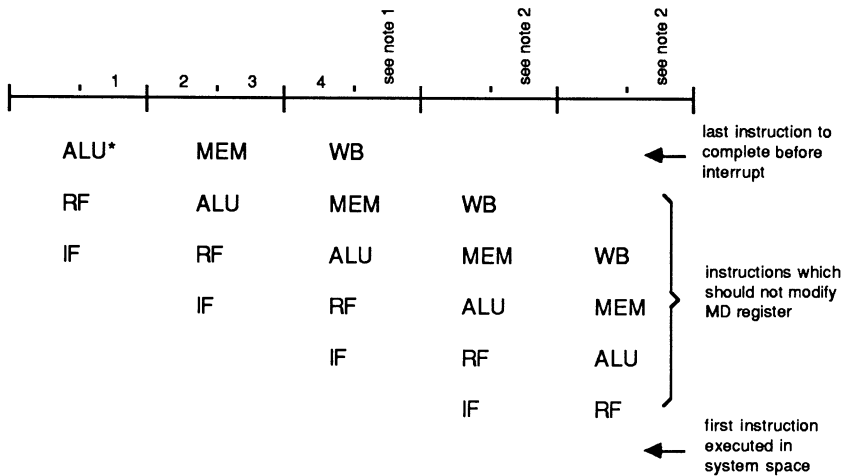


Figure 5.13: MD register schematic.



\* The value stored in the MD register at this point is the value that should be restored since this is the last instruction whose writeback completes before the interrupt.

note 1. Exception\_b\_pv2 falls during this cycle, restoring the old value of the MD register.

note 2. Squash\_b\_s2 falls for these cycles, preventing new values from overwriting the restored value.

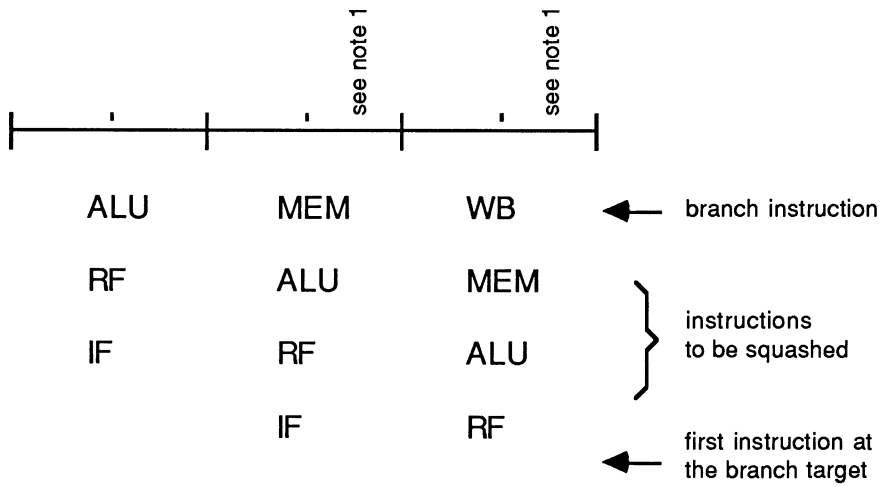
The numbers indicate the latch (see MD register schematic) where the value to restore is held during each phase.

Figure 5.14: MD register action during an exception.

outlining the MD register's operation in the case of an exception is shown in Figure 5.14. Figure 5.15 shows the squashing case.

### 5.2.7 MD Register Controller

The controller for the MD register generates only five signals. These include the input multiplexer lines, the bit to be shifted into the low-order position during instructions that shift the register, and the line that selects whether or not the MD register should drive the Result Bus. Most of the logic used to implement the MD register controller is contained in a single PLA with seven inputs (including true and complemented signals), four outputs, and four product terms. Most of the output signals are



Note 1. Squash\_b\_s2 falls during these cycles, preventing squashed instructions from overwriting the correct value in the MD register.

Figure 5.15: MD register action during instruction squashing.

qualified  $\phi_2$  signals. This causes a slight complication since many of the inputs used in the logic are  $\phi_1$  signals and must therefore be latched prior to entering the PLA. All but one of these inputs come from the IReg and are latched in the IReg. The other  $\phi_1$  input comes from the Immediate Bus; a special latch is included next to the PLA for this signal. The following sections discuss each of the signals generated by the controller.

### Input Multiplexer Signals

#### Signals:

$LdMDReg\_q2a = Movtos\_s2a \cdot CF30 \cdot Squash\_b\_s2 \cdot Exception\_b\_pv2$

$MDRegShift\_q2a = MDInst\_s2a \cdot Squash\_b\_s2 \cdot Exception\_b\_pv2$

$RestoreOldMDReg\_q2 = Exception\_v2$

These signals drive the select inputs of the multiplexer on the front of the MD register latches.  $LdMDReg\_q2a$  loads the MD register with the value on the Result Bus. This line should rise only for a *movtos MD* instruction that is not squashed. In addition, the line is ANDed with  $Exception\_b\_pv2$  in the control line drivers to prevent it from rising in the case of an exception. For multiply/divide instructions (*mstart*, *mstep*, and *dstep*) that are not squashed,  $MDRegShift\_q2a$  rises to shift the MD register left by one bit. This line is also ANDed with  $Exception\_b\_pv2$  in the control line drivers. On an exception  $RestoreOldMDReg\_q2$  rises to restore the value of the MD register that was held immediately following the last instruction to complete its write-back cycle before starting the exception code sequence. Aside from the aforementioned ANDing with  $Exception\_b\_pv2$ , all of the logic used to generate the input multiplexer signals is contained in the MD register controller PLA.

### Shifting in the Low-Order Bit

#### Signals:

$MDRegLSBIn\_s2a = MSBofALU\_b\_s2a$

This is the bit that is shifted into the low-order bit of the MD register when  $MDRegShift\_q2a$  is high. Note that this signal requires no logic. All that is required is a latch to capture the sign bit of the ALU output at the end of  $\phi_1$  for use during  $\phi_2$ . This latch is physically located at the bottom of the datapath, as is a large driver for the line that brings the signal back to the top of the datapath for use in the controllers.

## Driving the Result Bus

### Signals:

MDRegDrvResultBus\_q2a = Movfrs\_s2a · CF30

When this signal is high the value stored in the MD register will be written to the Result Bus. This only occurs when a Movfrs MD is executed. The logic that generates this line is contained in the MD register controller PLA.

## 5.2.8 ALU

As in most microprocessor designs, the ALU is one of the most timing-critical datapath units in MIPS-X. The timing constraints on the ALU were quite severe since the ALU needed to generate an effective address and drive it through to the data pads in a single cycle. Almost half of the cycle was required to get the data from the ALU to the pins, which meant that the ALU really had to complete its operation in one phase. This tight ALU timing did allow the use of a precharged design without the need to generate extra clocks. Since the output needed to be valid by the start of  $\phi_2$ , the ALU used all of  $\phi_2$  for precharge.

The MIPS-X ALU implements all of the logical and arithmetic operations, including supporting multiplication and division, as well as the comparisons for branches and computing memory addresses. It is certainly more than a simple adder. The basic structure is very similar to the standard ALU that is shown in Mead and Conway[31]. Each bit position generates propagate and generate ( $P$  and  $G$ ) signals; these are used in a carry generation block to yield the carries into each bit position. The sum is then generated by the exclusive-or of  $P$  with the carry for that bit. Logicals are done by changing the functions that the  $P$  and  $G$  block perform. By setting  $G$  to 0, the output for each bit is just  $P$ , which can be set to the desired function. To improve performance, a carry bypass scheme, with two levels of bypassing, was used. The standard domino design was not used. The ALU occupies 635  $\mu\text{m}$  of the datapath length.

Data in the ALU flows through several sections. It arrives at the input to the ALU on either of the Src Buses or from the Immediate Bus. A set of multiplexers selects between the Src2 or the Immediate Bus, while another set is used to select between Src1 or Src1 shifted by one bit (used to implement the multiply/divide instructions). A schematic of the input multiplexer is shown in Figure 5.16.



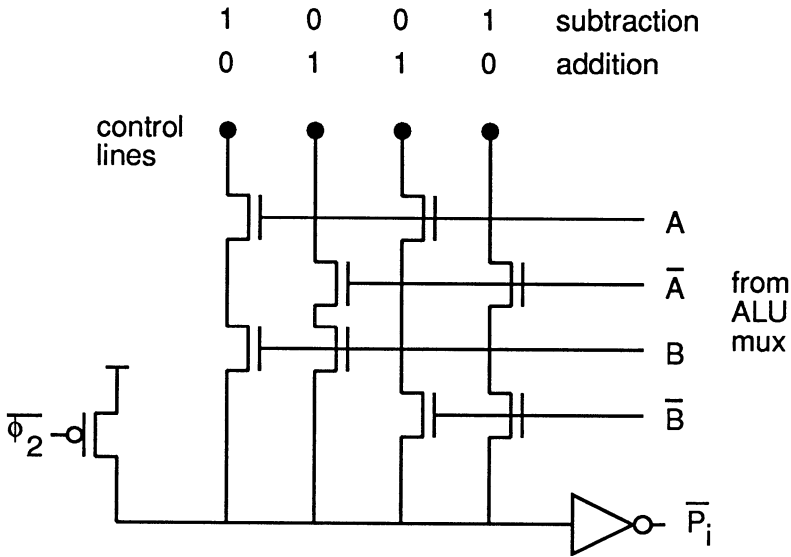


Figure 5.17: The ALU  $P$  block.

The outputs of the multiplexers feed the  $P$  and  $G$  blocks, which are shown in Figures 5.17 and 5.18. The  $P$  block is a general function block that can implement any function of two variables. The function is determined by the setting on the control lines. The control lines are high during precharge ( $\phi_2$ ), but are asserted during  $\phi_1$ . Since  $G$  is needed for only a few functions (add, subtract, and 0) a smaller function block can be used for  $G$ . Notice that the  $P$  block actually generates  $\bar{P}$ . This difference is important. It means that the adder cannot use a normal domino logic carry chain—all  $P$ s start high. But it does mean that the generation of the group  $P$  signals (e.g.,  $P_0 \cdot P_1 \cdot P_2 \cdot P_3$ ) can be done in a domino fashion, using  $\bar{P}$  and a precharged NOR gate. This is shown in Figure 5.19. Generating  $\bar{P}$  also allows an Equal output to be generated quickly for branches from the ALU, since Equal is just  $P_0 \cdot P_1 \cdot P_2 \dots P_{31}$  as shown in Figure 5.20.

The actual carry path is shown in Figure 5.21. As previously mentioned, it has two levels of bypassing, one around each group of four inputs, and the other around groups of eight inputs. Using this scheme, both bypass chains use only pass transistors; there are no inverter delays. Even though the local carry chains are relatively heavily loaded (since they generate the the output carries that go to exclusive-or gates) the

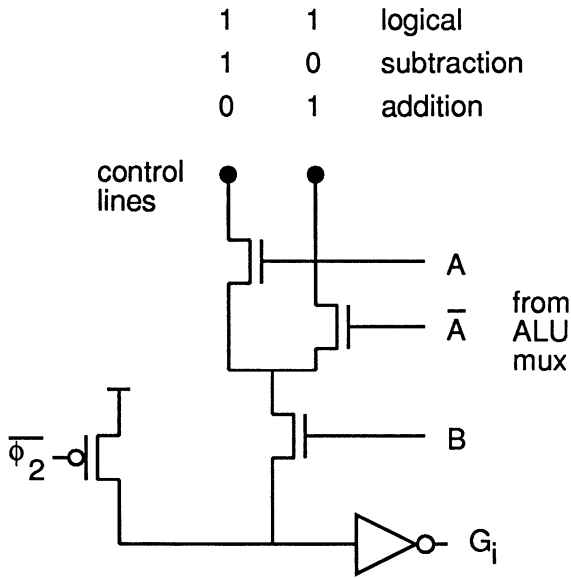


Figure 5.18: The ALU  $G$  block.

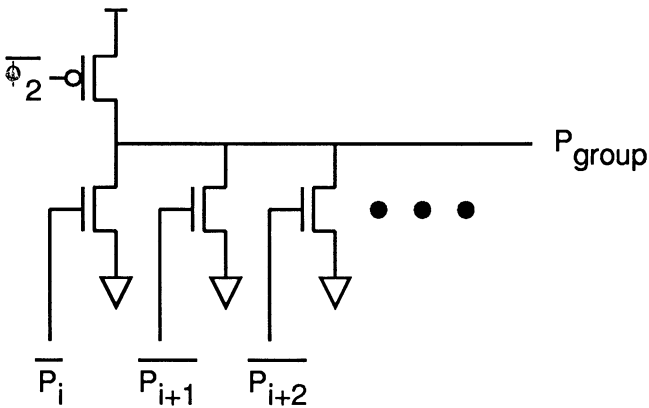


Figure 5.19: Generating group  $P$  signals.



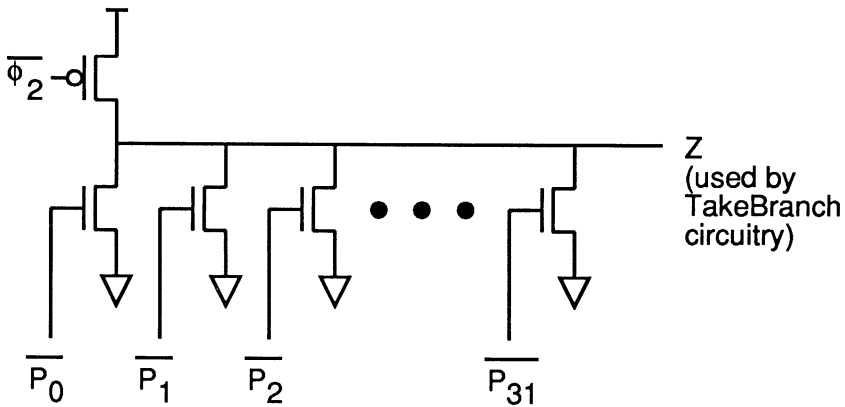


Figure 5.20: Generating an Equal output.

worst-case delay through the carry chain is about 15 ns.

During precharge all the  $P$  signals are asserted, which helps the precharge (it is not necessary to explicitly precharge each node in the carry chain), but it does mean that the circuit must wait until these signals reach their correct value before the carry chain can evaluate. This needed delay is provided by the  $Go$  line.  $Go$  is kept high until the  $P$  signals have stabilized, at which point it is pulled low, causing the carry chain to evaluate.  $Go$  is generated using a dummy  $P$  block to generate a delay that tracks the actual circuit delays. The buffering needed to drive the large load that  $Go$  sees is enough to match the delay in the group generation and provide some extra margin in the design.

The output of the carry chain is buffered and then exclusive-ored with  $P$  to form the sum, as shown in Figure 5.22. This output is then latched on the end of  $\phi_1$  and driven onto the Result Bus on  $\phi_2$ .

### 5.2.9 ALU Controller

Of the Execute Unit controllers, the ALU controller is the most complex. It was found that a PLA implementation of the controller would be too large; instead, a regular structure of gates similar to a Weinberger[46] array was used. In addition, the signals controlling the driving of the Result Bus have some complicated logic involving both  $\phi_1$  and  $\phi_2$  inputs. This logic is implemented with custom layout. The control lines that must be generated for the ALU include the input multiplexer lines, the ALU function lines, and the lines indicating whether the ALU should

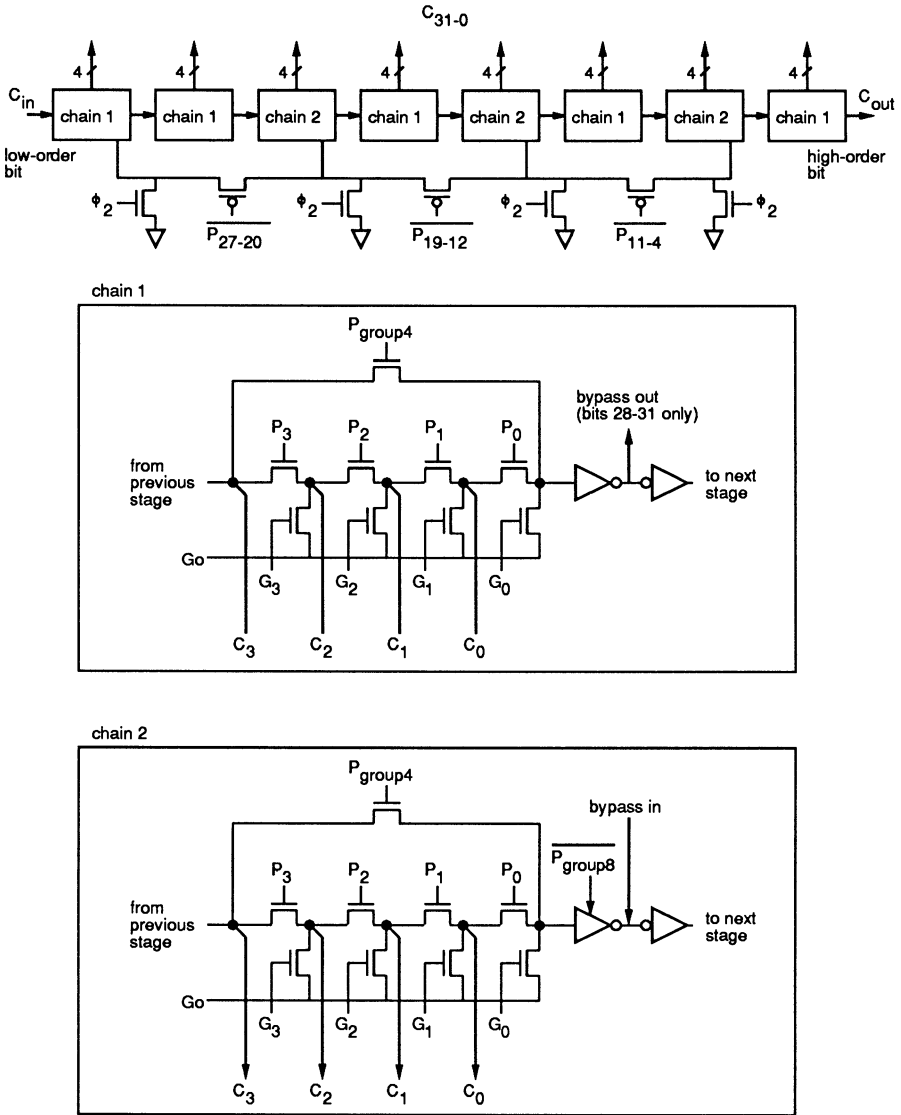


Figure 5.21: The ALU carry path.

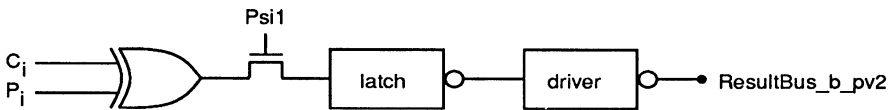


Figure 5.22: The ALU sum output.

drive the Result Bus. The following sections discuss each of the signals generated by the controller.

### Input Multiplexer Signals

#### Signals:

$\text{Src1ToALU}_{s1a} = \text{MDInst}_{b_{s1a}}$

$\text{Src1ShiftedToALU}_{s1a} = \text{MDInst}_{s1a}$

$\text{Src2ToALU}_{s1a} = \text{Branch}_{s1a} \vee \text{Compute}_{s1a}$

$\text{ConstantToALU}_{s1a} = \text{Memory}_{s1a} \vee \text{ComputeImmed}_{s1a}$

$\text{Src1ShiftedLSBIn}_{s1a} = \text{MDInst}_{s1a} \cdot \text{CF23} \cdot \text{MSBofMDReg}_{s1a}$

The input multiplexer selects the 32-bit inputs that will be fed to the ALU. One of the ALU inputs is fed by either the Src1 Bus or the Src1 Bus shifted left by one bit (used in the multiply/divide instructions). The other ALU input is fed by either the Src2 Bus or the Immediate Bus. These inputs are determined primarily by the type of instruction being executed, i.e., Memory, Branch, Compute, or Compute Immediate. The input multiplexer control signals are generated in the Weinberger-style structure.

### ALU Function Lines

#### Signals:

$P0 = \text{CF29} \cdot \text{Compute}_{s1a} \vee \text{Branch}_{s1a}$

$P1 = \text{CF28} \cdot \text{Compute}_{s1a} \vee \text{Memory}_{s1a} \vee \text{ComputeImmed}_{s1a}$

$P2 = \text{CF27} \cdot \text{Compute}_{s1a} \vee \text{Memory}_{s1a} \vee \text{ComputeImmed}_{s1a}$

$P3 = \text{CF26} \cdot \text{Compute}_{s1a} \vee \text{Branch}_{s1a}$

$G0 = \text{CF31} \cdot \text{Compute}_{s1a} \vee \text{Memory}_{s1a} \vee \text{ComputeImmed}_{s1a}$

$G1 = \text{CF30} \cdot \text{Compute}_{s1a} \vee \text{Branch}_{s1a}$

$\text{ALUCarryIn}_{s1a} = \text{CF25} \cdot \text{Compute}_{s1a} \vee \text{Branch}_{s1a}$

These are the lines that actually control the function performed by the ALU on its two 32-bit inputs. The lines consist of four propagate signals, two generate signals, and the carry into the ALU. The basic idea for each of these signals is to force an add to occur for Memory and Compute Immediate instructions, and to force a subtract for Branch instructions. For Compute instructions, each of the lines is extracted from the Comp Func field of the instruction, allowing great flexibility in

forming instructions that perform logical operations. The ALU function lines are all generated in the Weinberger-style structure.

### Driving the Result Bus

#### Signals:

$$\text{Src1DrvResultBus\_q2a} = \text{Movtos} \vee \text{Mult} \cdot \text{MSBofMDReg\_b} \vee \\ \text{Div} \cdot \text{MSBofALU}$$

$$\text{ALUDrvResultBus\_q2a} = \text{Memory} \vee \text{Compute} \cdot \text{Op0} \vee \\ \text{ComputeImmed} \cdot \text{Op0} \vee \text{Mult} \cdot \text{MSBofMDReg} \vee \\ \text{Div} \cdot \text{MSBofALU\_b}$$

where Mult means a multiply instruction (mstart or mstep)  
and Div means a divide instruction (dstep)

These lines determine if the ALU or Src1 Bus is to drive the Result Bus. They are quite complicated to generate, since some of the inputs are  $\phi 1$  signals and others are  $\phi 2$  signals. Basically, all of the  $\phi 1$  signals are logically combined as much as possible during  $\phi 1$  and these intermediate values are then latched and fed through the  $\phi 2$  logic along with the  $\phi 2$  inputs. All of the  $\phi 1$  logic is done in the Weinberger-style structure, and the latches and  $\phi 2$  logic are done in custom layout. Note that in the above equations all timing information and specific references to Comp Func bits have been omitted for clarity. A schematic showing the actual implementation of these signals is shown in Figure 5.23.

#### 5.2.10 PSW

The PSW (Processor Status Word) is a special register containing various status bits for the current state of the machine. The function of each bit is explained in Section 2.1.4. The PSW occupies 170  $\mu\text{m}$  of the datapath length.

A schematic of each PSWcurrent-PSWother bit pair is shown in Figure 5.24. The top two latches make up the PSWcurrent bit, and the bottom two latches make up the PSWother bit. The PSW can write to and read from the Result Bus. Each PSWcurrent bit can be saved in its PSWother bit. The PSWcurrent bit can also be restored from the PSWother bit where it was saved. When an exception is taken,

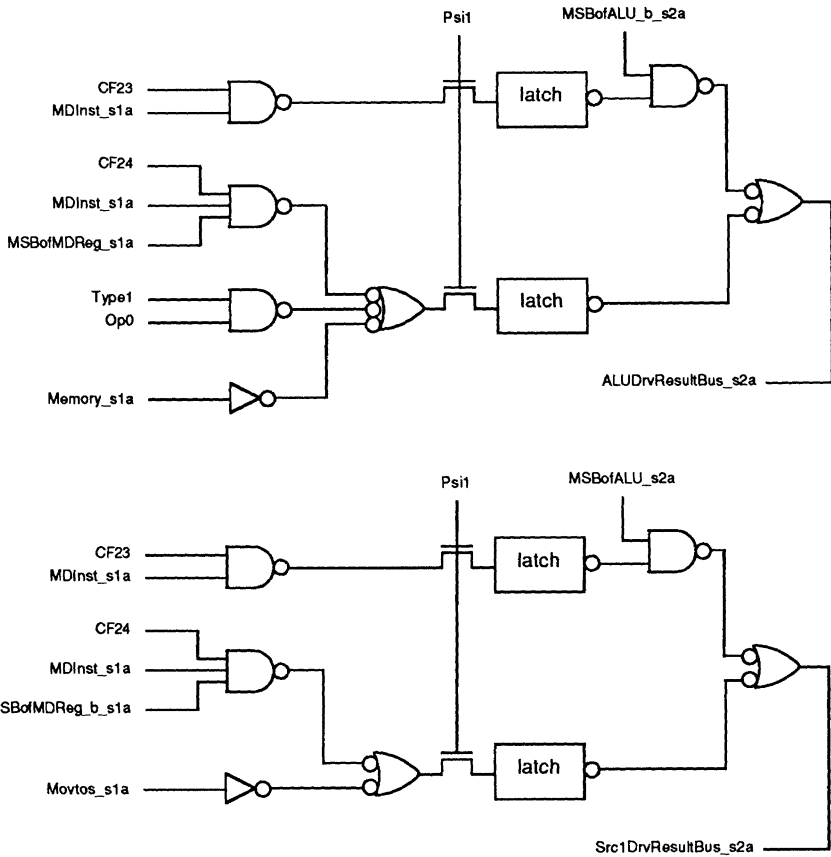


Figure 5.23: Result Bus control signals generated by the ALU controller.

the PSW<sub>current</sub> bits are saved in the PSW<sub>other</sub> bits and most of the PSW<sub>current</sub> bits are forced to a particular value (i.e., either 0 or 1 depending on the bit). This is not true, however, for the *I* and *O* bits. On an exception, they are forced to values that depend on whether or not the exception was caused by an interrupt or overflow.

The PSW *e* bit is totally different from the other bits in the PSW. Its purpose is to indicate to the operating system whether or not the machine state should be saved when an exception is taken. Normally it should be saved; the case in which it should not be saved is if the exception was taken in the middle of an exception return sequence. A schematic of the *e* bit is shown in Figure 5.25. The *e* bit is implemented as a shift chain that has a length tailored to match the length of the time it takes to restore machine state during the exception return sequence. When the return sequence begins, the entire shift chain is cleared; ones are then shifted from the left a phase at a time. When the shift chain is filled with ones, then the full machine state has been restored.

### 5.2.11 PSW Controller

The implementation of the PSW controller is fairly straightforward, since only five signals are required and each of them is a qualified  $\phi_2$  signal. The signals consist of the input multiplexer lines, a line indicating whether or not the *e* bit shift chain should shift, and a line specifying whether or not the PSW should drive the Result Bus.

Most of the logic used to implement the PSW controller is contained in a single PLA with 11 inputs (including true and complemented signals), five outputs, and seven product terms. All of the output signals are qualified  $\phi_2$  signals. Since many of the inputs used in the logic are  $\phi_1$  signals, the signals must be latched prior to entering the PLA. All but one of these inputs come from the IReg and are latched in the IReg. The other  $\phi_1$  input comes from the Immediate Bus; a special latch is included next to the PLA for this signal. The following sections discuss each of the signals generated by the controller.

#### Input Multiplexer Signals

##### Signals:

$$\text{LdPSW}_q2a = \text{Movtos}_s2a \cdot \text{CF}31 \cdot \text{PSWu}_b_s2 \cdot \text{Exception}_b_{pv2}$$

$$\text{ForcePSW}_q2 = \text{Exception}_v2 \vee \text{Reset}_s2 \vee \text{TrapInst}_s2a \cdot \text{Squash}_b_s2$$

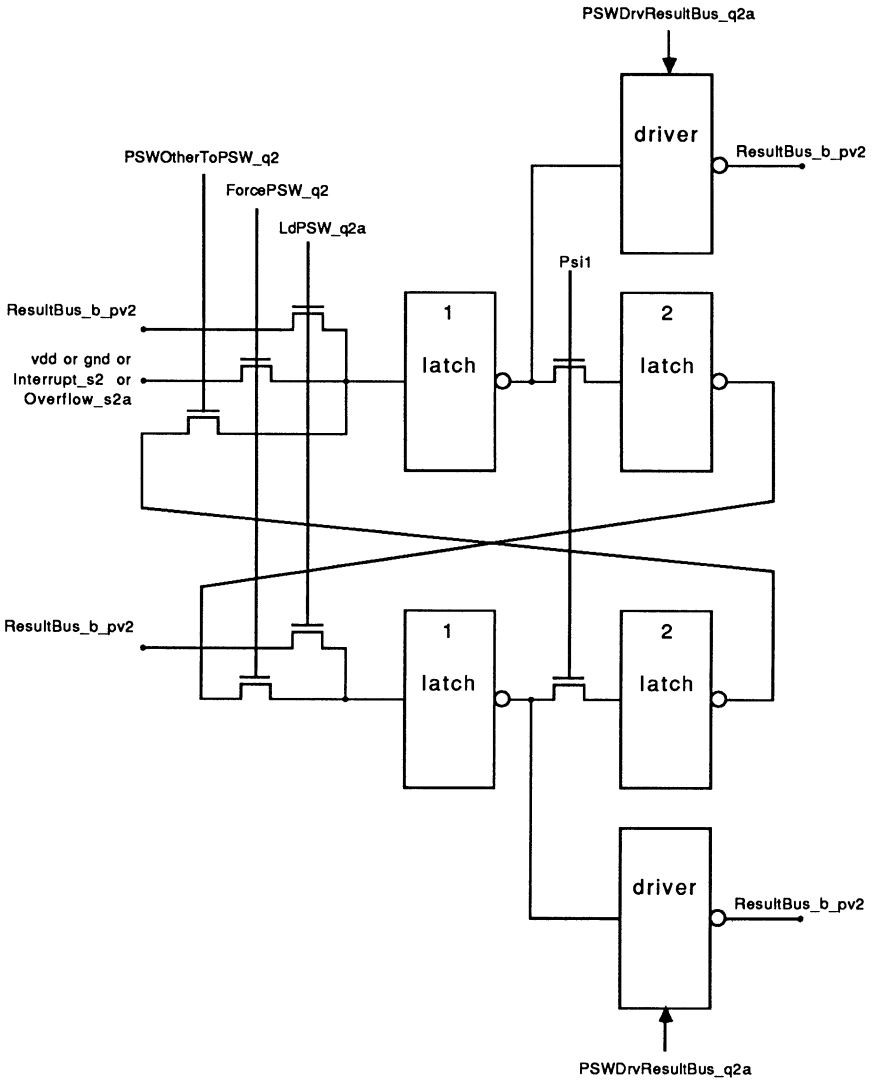


Figure 5.24: Schematic of a PSWcurrent-PSWother bit pair.

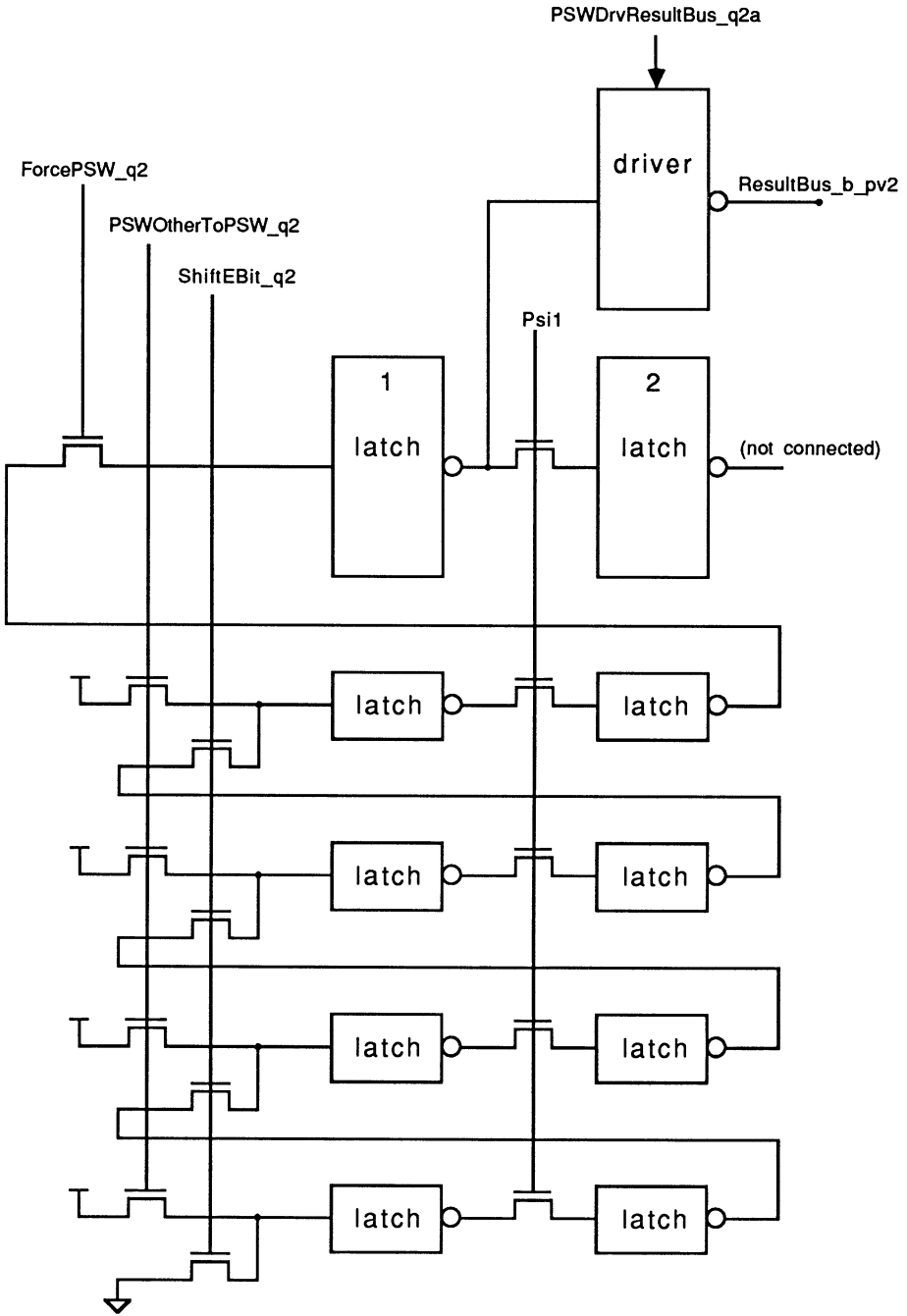


Figure 5.25: Schematic of the PSW e bit.



$$\text{PSWOtherToPSW\_q2} = \text{Jpcrs\_s2a} \cdot \text{PSWu\_b\_s2}$$

These signals drive the select inputs of the multiplexer on the front of the PSW latches. `LdPSW_q2a` loads the PSW with the value on the Result Bus. This line should rise only for a *movtos psw* instruction executed in system space. In addition, the line is ANDed with `Exception_b_pv2` in the control line drivers to prevent it from rising in the case of an exception. On an exception, a reset, or a trap instruction that is not being squashed, `ForcePSW_q2` rises to save each `PSWcurrent` bit in its respective `PSWother` bit and force the `PSWcurrent` bits to predetermined values. `PSWOtherToPSW_q2` restores each `PSWcurrent` bit with the value in its `PSWother` bit. This line is activated as part of the exception return sequence as triggered by a *jpcrs* instruction. Note that a *jpcrs* instruction executed from user space should do nothing. Aside from the ANDing with `Exception_b_pv2` that is done for `LdPSW_q2a`, all of the logic used to generate the input multiplexer signals is contained in the PSW controller PLA.

### E Bit Shift Enable

#### Signals:

$$\text{ShiftEBit\_q2} = \text{PSWs\_s2} \cdot \text{Jpcrs\_b\_s2a}$$

When activated, this signal enables shifting of the *e* bit shift chain; otherwise, shifting is disabled. Note that *e* bit shifting is enabled whenever PC shifting is enabled, except when a *jpcrs* instruction executes (in which case the *e* bit shift chain is reset). The logic for this signal is contained in the PSW controller PLA.

### Driving the Result Bus

#### Signals:

$$\text{PSWDrvResultBus\_q2a} = \text{Movfrs\_s2a} \cdot \text{CF31}$$

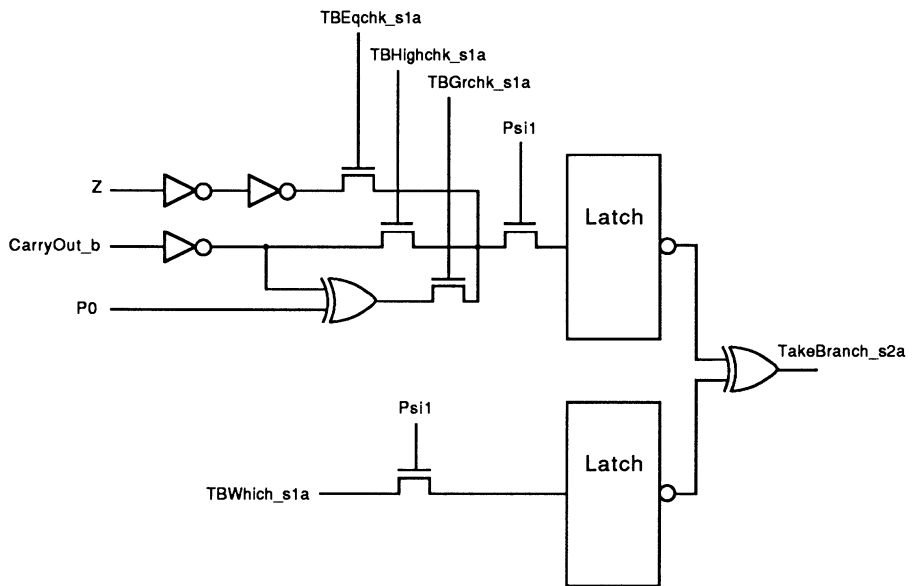
When this signal is high the value of the PSW will be written to the Result Bus. The only case when this occurs is when a *movfrs PSW* is executed. The logic that generates this line is contained in the PSW controller PLA.

### 5.2.12 Special Circuitry

Various pieces of miscellaneous custom logic are included in the Execute Unit. These pieces are located in an otherwise empty “33rd bit” slice at the bottom of the datapath, as well as in the unused bits of the PSW and the unused area where the Immediate Bus enters the datapath. One major function provided by this circuitry is the generation of the Take-Branch signal that indicates for branch instructions whether execution should continue sequentially or at the branch target. Since this decision is based on the output of the ALU, the circuitry is located in the 33rd bit position at the bottom of the datapath under the ALU. A schematic of this is shown in Figure 5.26. The TakeBranch signal is then sent to the PC Unit so it can continue execution at the appropriate address.

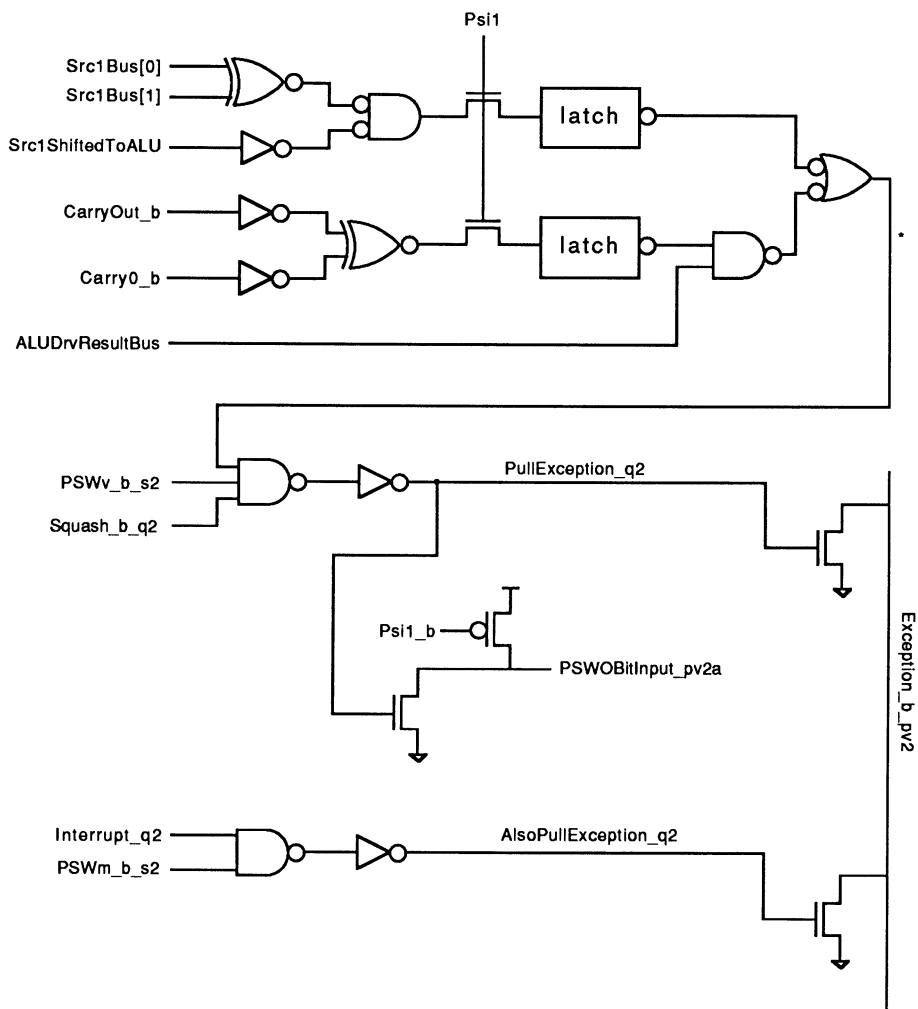
The logic that handles the detection of arithmetic overflow is also implemented in this circuitry. This detection is complicated by the fact that overflow can occur either in the ALU itself or in the shift operation sometimes performed in the ALU input multiplexer. Overflows may be masked by setting the appropriate bit in the PSW. A schematic of the overflow logic is shown in Figure 5.27. In addition to overflow detection, this schematic shows circuitry to activate `Exception_b_pv2` (the global signal used to indicate the occurrence of an exception of any kind) for an external interrupt. The generation of the signal fed to the PSW signalling the occurrence of an exception due to arithmetic overflow is also shown.

Finally, custom logic is provided to force off-chip memory addresses generated by MIPS-X to fall in user space (as opposed to system space) whenever the processor is executing in user mode. Since the lower half of the address space is designated system space and the upper half is designated user space, forcing the high-order bit of an address to a 1 will cause a mapping into user space. This technique is implemented in MIPS-X with a special circuit that operates on bit 0 of the Result Bus between the Execute and PC Units in the datapath. This circuit is shown in Figure 5.28. When MIPS-X is executing in user space, the circuit will force the most significant bit of the Result Bus high on the PC Unit side. (The Tag Unit multiplexes the PC Bus and the Result Bus onto the address pins.) Note that this does not change the value of the Result Bus in the Execute Unit; in this way, the results of compute instructions are not affected. When in system mode, the value of the Result Bus is transferred unaltered to the Result Bus on the PC Unit side. If the PC



$$\begin{aligned}
 \text{TBEqchk} &= \text{Op}[1]_{\text{b}} * \text{Op}[2] \\
 \text{TBHighchk} &= \text{Op}[1] * \text{Op}[2]_{\text{b}} \\
 \text{TBGrchk} &= \text{Op}[1] * \text{Op}[2] \\
 \text{TBWhich} &= \text{Op}[0]
 \end{aligned}$$

Figure 5.26: Schematic of the TakeBranch logic.



$$* = (\text{AddOvfl}) * (\text{ALUDrvResultBus}) + (\text{ShiftOvfl}) * (\text{Src1ShiftedToALU})\_s2a$$

Figure 5.27: Schematic of the overflow logic.

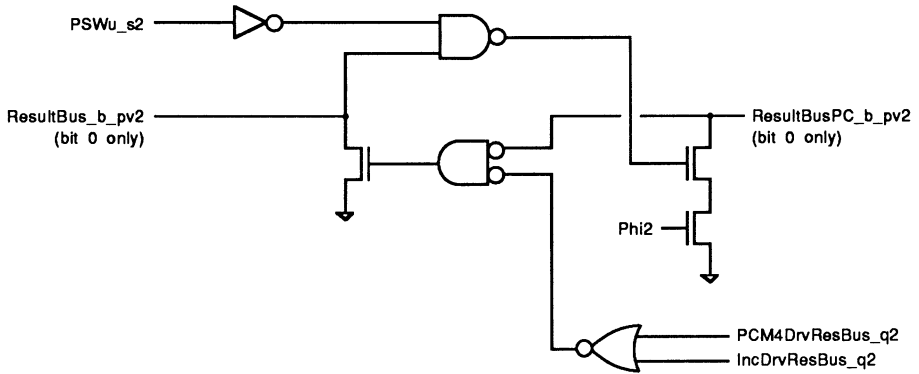


Figure 5.28: System/User space circuitry on the Result Bus.

Unit is driving the Result Bus (e.g., in a *movfrs* instruction) then the Result Bus from the PC Unit is passed unchanged to the Result Bus on the Execute Unit side.

## Chapter 6

# Instruction Fetch Hardware

One of the difficult tasks of keeping a high speed processor busy is to make sure that it has a constant supply of instructions and to make sure that they are supplied quickly. This requires a very high bandwidth for both instructions and data. For MIPS-X, when both an instruction and data are required, the peak bandwidth required is 160 Mbytes/s. Building a memory system to support this would not be simple and transferring this amount of data across the pins of a VLSI chip would have been difficult with the packaging technology available at the time. MIPS-X uses an on-chip instruction cache to reduce the memory bandwidth requirements. Putting an instruction cache on-chip also makes it possible to fetch new instructions every cycle if the instructions are in the cache.

This chapter describes the four units that comprise the instruction fetch hardware. The PC Unit and the Tags Unit are located in the datapath and the ICache RAM and the Instruction Register are located above the datapath. Instruction addresses are generated in the PC Unit and sent to the Tags Unit where they are compared with the tags to determine if the address is in the ICache. At the same time, some of the bits are used to begin accessing the ICache RAM array. If an address misses in the ICache then an ICache miss sequence begins. If a hit is detected, the instruction accessed in the ICache RAM is latched into the Instruction Register. Some initial decoding of the instruction is done in the Instruction Register.

The ICache RAM array takes up about 40% of the total chip area, the PC Unit and the Tags Unit comprise about 12% and the Instruction Register uses a little over 2% of the total chip area.

## 6.1 The PC Unit

The PC Unit handles the addressing of instructions and saving instruction addresses so that the machine can be restarted after an exception. Located between the Execute Unit and the Tag Unit, the PC Unit takes up about 1425  $\mu\text{m}$  of the datapath with some of this length overlapping the Execute Unit and the Tag Unit. The total height is 2918  $\mu\text{m}$  where much of the extra height above the datapath is due to a PLA. The datapath is 32 bits wide but the most significant bit (bit 0) is treated differently from the rest because it is the system/user bit.

The main functions of the PC Unit are:

- Increment the current value on the PC Bus.
- Sign extend and add the displacement field for branches to the current value on the PC Bus.
- Drive the PC Bus during Reset, Exceptions and traps.
- Save the addresses of the instructions currently in the pipeline.
- Sequence the processor during ICache misses.
- Sequence the processor during instruction squashing.
- Drive the ALU result onto the Result Bus.

The main parts of the PC Unit are:

**Incrementer:** Compute the value of the PC Bus incremented by 1.

**Displacement Adder:** Add the branch offset to the PC Bus.

**PC Chain:** Shift chain to save the PC values of instructions in execution.

**Cache Miss FSM:** Finite state machine to handle cache misses.

signal	bits	timing		direction	connected to Unit
CMDelay_s1	1	stable	$\phi 1$	output	Execute,Tags
CM_s1	1	stable	$\phi 1$	output	Regfile
CMPC_s2	1	stable	$\phi 2$	output	Regfile
Squash_b_s2	1	stable	$\phi 2$	output	Execute,Regfile
PCM4DrvResBus_q2	1	qualified	$\phi 2$	input	Execute
IncDrvResBus_q2	1	qualified	$\phi 2$	input	Execute
Exception_b_pv2	1	valid	$\phi 2$	input	Execute
ICacheTest_s1	1	stable	$\phi 1$	input	Eface
Reset_s2	1	stable	$\phi 2$	input	Eface
Psi1PC	1	clock	$\phi 1$	input	Eface
Psi1	1	clock	$\phi 1$	input	Eface
Phi2_b	1	clock	$\phi 2$	input	Eface
Phi2	1	clock	$\phi 2$	input	Eface
Phi1	1	clock	$\phi 1$	input	Eface
TakeBranch_s2a	1	stable	$\phi 2$ ALU	input	Execute
PSWs_b_s2	1	stable	$\phi 2$	input	Execute
Immed[0..16]_s1a	17	stable	$\phi 1$ ALU	input	IReg
PCBus_s1	32	stable	$\phi 1$	output	Execute,Tags
PCBus_b_s1	32	stable	$\phi 1$	output	Tags
ResultBus_b_pv2	32	valid	$\phi 2$	bidirectional	Execute,Tags
ALUResult[1..31]_s2a	31	stable	$\phi 2$ ALU	input	Execute
PSWu_b_s2	1	stable	$\phi 2$	input	Execute
Hit_v1	1	valid	$\phi 1$	input	Tags
IMiss_s2	1	stable	$\phi 2$	output	Tags
TrapInst_s1a	1	stable	$\phi 1$ ALU	input	IReg
TrapInst_s2a	1	stable	$\phi 2$ ALU	input	IReg
Branch_s2a	1	stable	$\phi 2$ ALU	input	IReg
Jpcrs_s2a	1	stable	$\phi 2$ ALU	input	IReg
Jpc_s2a	1	stable	$\phi 2$ ALU	input	IReg
Jspci_s2a	1	stable	$\phi 2$ ALU	input	IReg
Movfrs_s2a	1	stable	$\phi 2$ ALU	input	IReg
Movtos_s2a	1	stable	$\phi 2$ ALU	input	IReg

Table 6.1: Signals connected to the PC Unit.

**Squash FSM:** Finite state machine to handle instruction squashing.

**PC Logic:** This was previously a PLA but it has been converted to standard cell logic. Descriptions here will refer to the PLA.

### 6.1.1 PC Unit Interface

Figure 6.1 shows the logical interface of the PC Unit and the approximate locations of the signals in the layout. Table 6.1 gives the size of the signals, the timing and how they are connected to the neighbouring section.



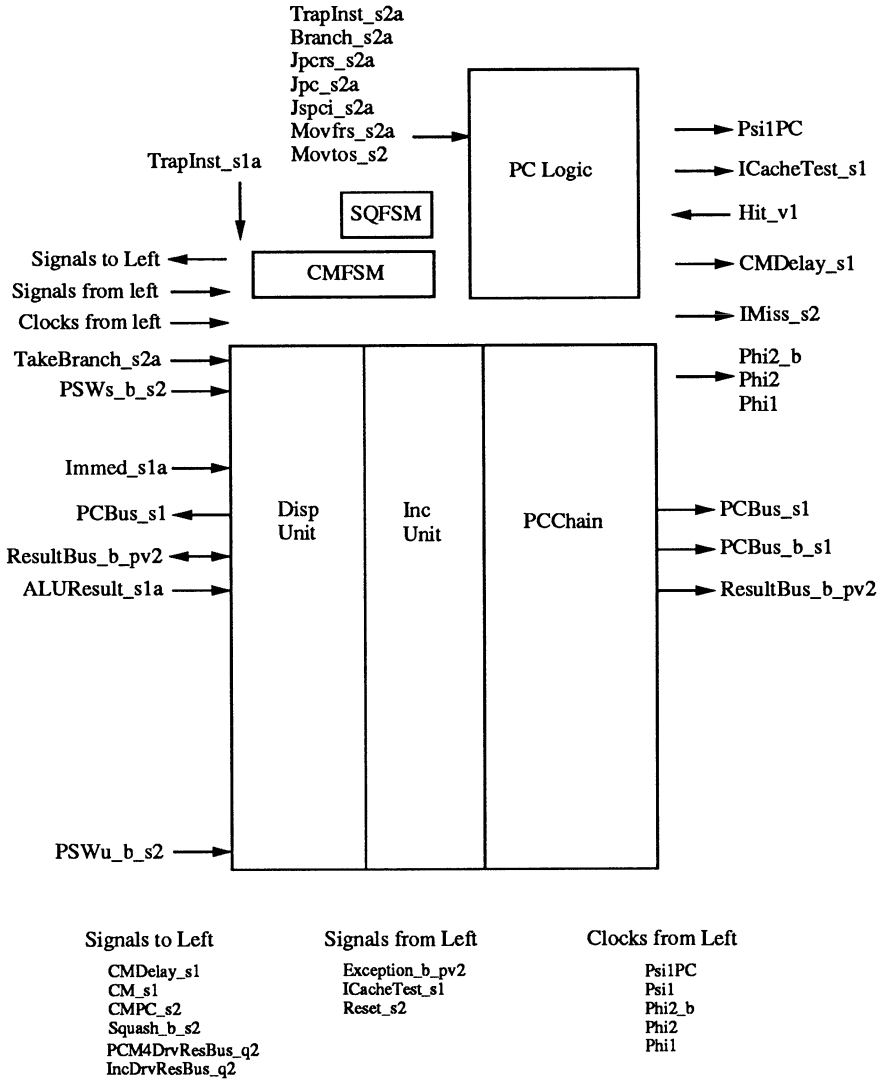


Figure 6.1: PC Unit interface.

The clock signals are described in Section 4.2. The functions performed by the other signals are described below:

**CMDelay\_s1** Cache miss signal from the CMFSM.

**CM\_s1** Cache miss signal from the CMFSM.

**CMPC\_s2** Cache miss signal from the CMFSM.

**Squash\_b\_s2** This signal goes low when an instruction needs to be squashed due to an exception, a trap instruction or a squashing branch.

**Exception\_b\_pv2** A precharged signal that is pulled low when an exception occurs.

**ICacheTest\_s1** Forces the SQFSM to generate the Squash\_b\_s2 signal so that the PC Unit will only increment the value on the PC Bus allowing the cache to be accessed sequentially for testing purposes.

**Reset\_s2** Force the CMFSM into the RUN state and the SQFSM into the first SQUASH state. When the machine is powered on there are two instructions in the pipeline before the first instruction of the reset sequence. These instructions could conceivably be branch instructions that would cause the machine to branch before the reset sequence could begin. Turning on the Squash signal will force the PC to increment until the reset sequence begins.

**TakeBranch\_s2a** This is generated in the Execute Unit and is asserted if a branch condition is true.

**PSWs\_b\_s2** This represents the value of the S bit in the PSW. Shifting of the PC Chain is enabled when the S bit is 1.

**Immed[0..16]\_s1a** This is the immediate value field of the instruction that comes from the Instruction Register.

**PCBus\_s1** This contains the address of the next instruction to be fetched.

**PCBus\_b\_s1** The complement of PCBus\_s1 used by the Tag Unit.

**ResultBus\_b\_pv2** The PC Unit uses this when values in the PC Chain are being saved or to load the PC Chain. Bit 0 of the Result Bus is actually treated differently from the rest of the bits. This is because of the system/user bit. There is some hardware (the *widgit*) that forces this bit to be in the correct address space when memory addresses are generated but does not affect the bit when PC values are being transferred to the Register File. This circuitry is discussed in more detail in Section 5.2.12.

**ALUResult[1..31]\_s2a** This value is used to get the jump address for the *jspci* instruction. The MSB of the jump address is determined by PSWu\_b\_s2.

**PSWu\_b\_s2** This reflects the value in the System/User bit of the PSW. The System/User bit is also used in the MSB of all addresses.

**Hit\_v1** This signal comes from the Tags Unit and indicates whether or not there was a hit or miss in the ICache.

**IMiss\_s2** A signal from the CMFSM that is used in the Tags Unit.

**TrapInst\_s1a** A signal generated by the IReg that indicates the ALU cycle is for a *trap* instruction.

**TrapInst\_s2a** The ALU cycle is a *trap* instruction.

**Branch\_s2a** The ALU cycle is a branch instruction.

**Jpcrs\_s2a** The ALU cycle is a *jpcrs* instruction.

**Jpc\_s2a** The ALU cycle is a *jpc* instruction.

**Jspci\_s2a** The ALU cycle is a *jspci* instruction.

**Movfrs\_s2a** The ALU cycle is a *movfrs* instruction.

**Movtos\_s2a** The ALU cycle is a *movtos* instruction.

### 6.1.2 Timing

New PC values and control signals that affect the drivers on the PC Bus are evaluated or generated during  $\phi_1$  and the PC Bus is driven on  $\phi_2$ . The Incrementer and Displacement Unit adders use the same timing as the ALU in the Execute Unit.

### 6.1.3 Signal Derivations and Origins

Most of the combinational logic is done by the static PLA that is found in the block labeled PCLogic in Figure 6.1. Other logic is done by qualifying signals at the inputs to the drivers for the datapath control lines. There are also small pieces of combinational logic within the finite state machines.

Measurements of the first working chips confirmed that the path through the PC PLA was one of the major critical paths. This PLA has been changed to use standard cells instead. The PLA implementation and the corresponding logic at the control drivers are described first, followed by a description of the standard cell logic.

#### The PC PLA

The PLA was generated initially using *mpla*[38] with a tile set to make a self-timed dynamic PLA. This eventually evolved into a fully static implementation when it was realized that some of the signals could arrive late in  $\phi_2$ . The signals that were of most concern were Exception\_b\_pv2 and TakeBranch\_s2a. The outputs of the PLA are all qualified  $\phi_2$  except for SquashFSMinIn\_v2, which gets latched in the Squash FSM. The use of these logic functions can be seen in Figures 6.2, 6.3, 6.4, 6.6, and 6.8.

Included below is the input for *eqntott*[38] used to generate the PLA.

```

INORDER =      TakeBranch_s2a
                Exception_b_pv2
                Jspci_s2a
                Movtos_s2a
                Movfrs_s2a
                Jpcrs_s2a
                Jpc_s2a
                Branch_s2a
                TrapInst_s2a
                Immed15_s2a
                Immed29_s2a
                Squash2_s2
                Squash_b_s2
                IMiss_s2
                PSWs_s2
                Reset_s2
                CMPC_s2
                ;

```

```

OUTORDER =      SquashFSMinIn_v2
                AluDrvPCBus_q2
                LatchZero_v2
                NonSeqDrvPCBus_q2
                IncDrvPCBus_q2
                IncDrvResBus_q2
                PCM1GetsResBus_q2
                ShiftPC_q2
                ZeroPC_q2
                ShiftNotZeroPC_q2
                PCM4DrvPCBus_q2
                PCM4DrvResBus_q2
                ;
AluDrvPCBus_q2 = (!CMPC_s2)
                & (!Reset_s2)
                & Squash_b_s2
                & Jspci_s2a;
PCM1GetsResBus_q2 = (!CMPC_s2)
                & Movtos_s2a
                & Immed29_s2a;
ShiftPC_q2 = (!CMPC_s2)
                & (PSWs_s2 & !(Movtos_s2a & Immed29_s2a)
                );
ShiftNotZeroPC_q2 = ((PSWs_s2
                | ((Movtos_s2a | Movfrs_s2a) & Immed29_s2a
                )
                ) & (!Squash2_s2)
                )
                | Jpcrs_s2a & (!CMPC_s2)
                & (!Reset_s2)
                & Squash_b_s2;
ZeroPC_q2 = PSWs_s2
                & Squash2_s2;
PCM4DrvPCBus_q2 = (!CMPC_s2)
                & (!Reset_s2)
                & Squash_b_s2
                & (Jpcrs_s2a | Jpc_s2a);
PCM4DrvResBus_q2 = (!IMiss_s2)
                & Movfrs_s2a
                & Immed29_s2a;
LatchZero_v2 = (!Exception_b_pv2)
                & (!Reset_s2);
IncDrvPCBus_q2 = (!CMPC_s2)
                & (!Reset_s2)

```

```

        & (!Squash_b_s2
          | Branch_s2a
            & (!TakeBranch_s2a)
          | (!Jspci_s2a)
            & (!Jpc_s2a)
            & (!Jpcrs_s2a)
            & (!Branch_s2a)
            & (!TrapInst_s2a)
        );
IncDrvResBus_q2 = (!IMiss_s2)
    & Jspci_s2a;
NonSeqDrvPCBus_q2 = Reset_s2
    | (!Exception_b_pv2)
    | ( (!CMPC_s2)
        & (Squash_b_s2)
        & (TrapInst_s2a
            | Branch_s2a
                & TakeBranch_s2a
            )
        );
SquashFSMinIn_v2 = Squash_b_s2
    & (Branch_s2a
        & Immed15_s2a
        & (!TakeBranch_s2a)
        | TrapInst_s2a
    );

```

### Logic at the Control Drivers

This section gives the logic that is done at the inputs of the control line drivers. The left hand sides of the equations are the outputs of the drivers. All outputs from the PLA are qualified with “(pla)” since some of the names are the same. If there is no right hand side then there is no other logic. The order of the drivers starts from the Execute Unit side. PC Bus drivers also have a complemented input that is not shown.

```

AluDrvPCBus_q2 = AluDrvPCBus_q2(pla) & Exception_b_pv2
Phi2
ALU GO
Phi2
LatchTrapVector_q1 = Psi1PC & TrapInst_s1a & !Reset_s2
Reset_s2
LatchZero_v2 = LatchZero_v2(pla)

```

```

LatchDispAdder_q2 = Psi1PC & !TrapInst_s1a & !Reset_s2
NonSeqDrvPCBus_q2 = NonSeqDrvPCBus_q2(pla)
Phi2
ALU GO
Phi2
Psi1
IncDrvPCBus_q2 = IncDrvPCBus_q2(pla) & Exception_b_pv2
IncDrvResBus_q2 = IncDrvResBus_q2(pla) & Exception_b_pv2
Psi1PC
Phi2
Psi1PC
CMPCDrvPCBus_q2 = Phi2 & Exception_b_pv2 & CMPC_s2
PCM1GetsResBus_q2 = PCM1GetsResBus_q2(pla)
ShiftPC_q2 = ShiftPC_q2(pla)
Psi1
ZeroPC_q2 = ZeroPC_q2(pla)
ShiftNotZeroPC_q2 = ShiftNotZeroPC_q2(pla)
Psi1
ZeroPC_q2 = ZeroPC_q2(pla)
ShiftNotZeroPC_q2 = ShiftNotZeroPC_q2(pla)
Psi1
PCM4DrvPCBus_q2 = PCM4DrvPCBus_q2(pla) & Exception_b_pv2
PCM4DrvResBus_q2 = PCM4DrvResBus_q2(pla) & Exception_b_pv2

```

### The Standard Cell Implementation

In this section, the standard cell implementation is described. Explanations are given about how and where various functions are generated.

The original implementation of the PLA used a dynamic PLA structure. This meant that the exception signal could not be used as an input because it had to be a stable signal. The PLA was eventually converted to a static implementation but the exception signal was left at the drivers. The standard cell implementation moved the exception signal into the standard cell logic instead of putting it in the drivers. This is reflected in the following equations. These terms are generated in the standard cell block and then qualified with  $\phi_2$  at the drivers.

```

INORDER =      TakeBranch_s2a
                Exception_b_pv2
                Jspci_s2a
                Movtos_s2a
                Movfrs_s2a
                Jpcrs_s2a

```

```

Jpc_s2a
Branch_s2a
TrapInst_s2a
Immed15_s2a
Immed29_s2a
Squash2_s2
Squash_b_s2
IMiss_s2
PSWs_s2
Reset_s2
CMPC_s2
;
OUTORDER = SquashFMinIn_v2
            AluDrvPCBus_q2
            LatchZero_v2
            NonSeqDrvPCBus_q2
            IncDrvPCBus_q2
            IncDrvResBus_q2
            PCM1GetsResBus_q2
            ShiftPC_q2
            ZeroPC_q2
            ShiftNotZeroPC_q2
            PCM4DrvPCBus_q2
            PCM4DrvResBus_q2
            ;

AluDrvPCBus_q2 = (!CMPC_s2)
                & Exception_b_pv2
                & (!Reset_s2)
                & Squash_b_s2
                & Jspci_s2a;
PCM1GetsResBus_q2 = (!CMPC_s2)
                   & Movtos_s2a
                   & Immed29_s2a;
ShiftPC_q2 = (!CMPC_s2)
             & (PSWs_s2 & !(Movtos_s2a & Immed29_s2a)
              );
ShiftNotZeroPC_q2 = ((PSWs_s2
                    | ((Movtos_s2a | Movfrs_s2a) & Immed29_s2a
                      )
                    ) & (!Squash2_s2)
                   )
                  | Jpcrs_s2a & (!CMPC_s2)

```



```

                                & (!Reset_s2)
                                & Squash_b_s2;
ZeroPC_q2 = PSWs_s2
            & Squash2_s2;
PCM4DrvPCBus_q2 = (!CMPC_s2)
                  & Exception_b_pv2
                  & (!Reset_s2)
                  & Squash_b_s2
                  & (Jpcrs_s2a | Jpc_s2a);
PCM4DrvResBus_q2 = (!IMiss_s2)
                   & Exception_b_pv2
                   & Movfrs_s2a
                   & Immed29_s2a;
IncDrvResBus_q2 = (!IMiss_s2)
                  & Exception_b_pv2
                  & Jspci_s2a;

```

The SquashFSMin\_v2 signal goes to the Squash Finite State Machine, which is located next to the standard cell logic:

```

SquashFSMinIn_v2 = Squash_b_s2
                  & (Branch_s2a
                      & Immed15_s2a
                      & (!TakeBranch_s2a)
                      | TrapInst_s2a
                  );

```

The LatchZero\_v2 signal is done locally at the driver by using a NOR gate and feeding the output into the driver with  $\phi_2$ .

```

LatchZero_v2 = (!Exception_b_pv2)
               & (!Reset_s2);

```

The next signals contain TakeBranch\_s2, which arrives late in  $\phi_2$  at the PC Unit. TakeBranch\_s2 is generated at the bottom of the datapath (near the output of the ALU) by the time  $\phi_1$  falls. It must then be driven all the way up and over to the PC logic. The delay is minimized by using the signal as close to the PC control drivers as possible. To do this, the equations are decomposed into several parts that are combined with TakeBranch\_s2 and  $\phi_2$  at the drivers.

The complete function for IncDrvPCBus\_q2 is:

```

IncDrvPCBus_q2 = (!CMPC_s2)
                & Exception_b_pv2
                & (!Reset_s2)
                & (!Squash_b_s2
                  | Branch_s2a
                    & (!TakeBranch_s2a)
                  | (!Jspci_s2a)
                    & (!Jpc_s2a)
                    & (!Jpcrs_s2a)
                    & (!Branch_s2a)
                    & (!TrapInst_s2a)
                );

```

The complete function is decomposed into the terms T1 through T4, which are computed in the standard cells.

```

T1 = (!CMPC_s2) & Exception_b_pv2 & (!Reset_s2);
T2 = (!Jspci_s2a) & (!Jpc_s2a) & (!Jpcrs_s2a) & (!Branch_s2a)
    & (!TrapInst_s2a);
T3 = T1 & (!Squash_b_s2) | T1 & T2;
T4 = T1 & Branch_s2a;

```

At the driver, T5 is qualified with  $\phi_2$ .

```
T5 = T3 | T4 & (!TakeBranch_s2a);
```

The complete function for NonSeqDrvPCBus\_q2 is:

```

NonSeqDrvPCBus_q2 = Reset_s2
                  | (!Exception_b_pv2)
                  | ( (!CMPC_s2)
                      & (Squash_b_s2)
                      & (TrapInst_s2a
                        | Branch_s2a
                          & TakeBranch_s2a
                      )
                  );

```

The decomposition is:

```

T1 = Reset_s2 | (!Exception_b_pv2);
T2 = (!CMPC_s2) & Squash_b_s2;
T3 = T1 | T2 & TrapInst_s2a;
T4 = T2 & Branch_s2a;

```

At the driver, T5 is qualified with  $\phi_2$ .

```
T5 = T3 | T4 & TakeBranch_s2a;
```

### 6.1.4 The System/User Bit

Bit 0 of addresses from the PC Unit is the System/User bit and is treated differently from the rest of the bits in the PC Unit. When the System/User bit (PSWu\_s2) is set to 1, the machine is executing in user space and when it is set to 0, the machine is in system space. This bit replaces the most significant bit of the Incrementer and Displacement adders so that the address space cannot be affected by the adders. More details of these connections are given when each of the individual sections are discussed.

### 6.1.5 The ALU Drivers

The ALU Drivers are at the front of the PC Unit next to the Execute Unit and are included as part of the Displacement Unit as shown in Figure 6.2. The input of the drivers is ALUResult\_s2a for bits 1 to 31 and PSWu\_b\_s2 for bit 0. PSWu\_b\_s2 goes through an inverter before going into the drivers. The ALU is used to calculate the destination for *jspci* instructions and the result is driven onto the PC Bus with these drivers. The size of the drivers used are 40/2 p-channel transistors and 20/2 n-channel transistors, and are enabled by AluDrvPCBus\_q2.

### 6.1.6 The Displacement Unit

The Displacement Unit has four possible sources for driving the PC Bus. They are Address 0, the Trap Vector, the Reset Vector and the Displacement Adder. These four values are multiplexed into a latch that provides the input for a PC Bus driver.

#### Address 0

When an exception occurs the processor begins execution at Address 0 in system space. Since the latch is inverting, Address 0 is provided by tying the input of the multiplexer to *Vdd*. The System/User bit is not needed in bit 0 because the exception handling routine will always be in system space.

#### The TrapVector

The *trap* instruction contains an 8-bit vector in bits 21 to 28. This is a part of the immediate field so the Immediate Bus (Immed\_s1a) is used

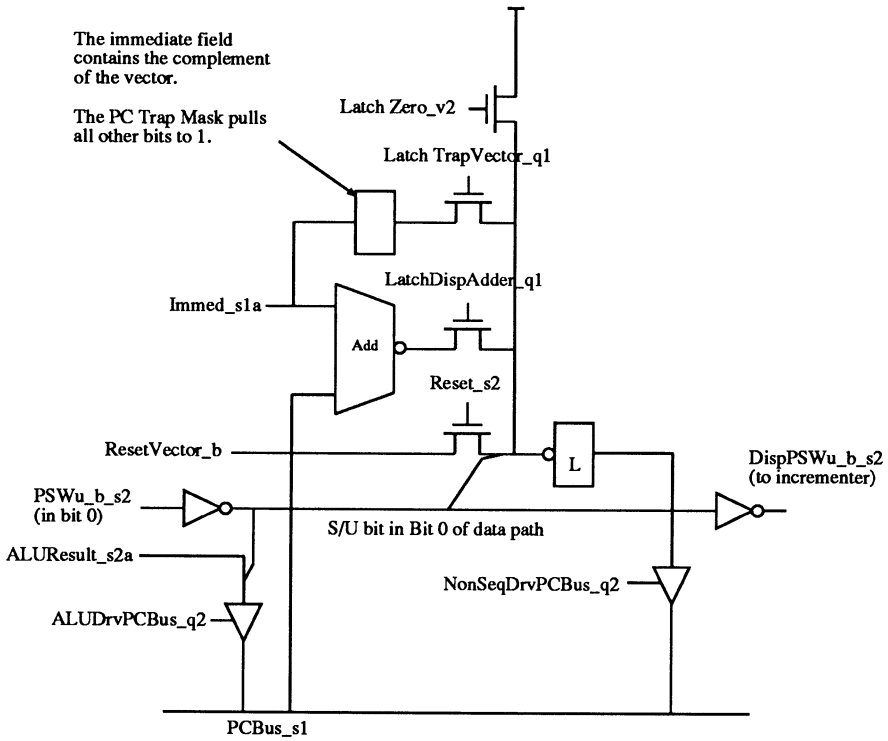


Figure 6.2: PC Displacement Unit.

to get these values to the multiplexer. The reason that the instruction is encoded with the bit complement of the vector is because the latch inverts the bits before they get to the drivers. All bits but bits 21 to 28 are pulled to *Vdd* to mask them to zero. All traps vector to system space.

### The Reset Vector

This sets the location where the processor will begin execution after a reset. The reset vector is specified by programming tabs on the multiplexer input that connect to either of the power supplies. The programming node in the hardware is called `ResetVector_b`. It has been wired to `0x7ffffe0`<sup>1</sup>.

### The Displacement Adder

The Displacement Adder is used to calculate the destination of a branch. The inputs are `PCBus_s1` and a version of the Immediate Bus (`Immed_s1a`) where bits 0 to 15 are sign extended from bit 16. The adder functions on every cycle but the output is only latched if there is a taken branch. The circuit for the adder is a simplified version of the ALU in the Execute Unit. The carry-in line, `Cin_b`, has been tied to *Vdd*.

In the bit 0 position of the adder, the final exclusive-or gate has been removed to make room for an inverter so that the System/User bit will be the correct polarity. The actual output of the adder in this position is not used.

#### 6.1.7 The Incrementer

The Incrementer is always computing the next sequential value of the PC by adding `PCBus_s1` to 0 with the carry-in set (`Cin_b` connected to *Gnd*). The adder circuit is the same cell as the one used for the displacement adder. The output of the incrementer can be driven onto the PC Bus or onto the Result Bus. Figure 6.3 shows a schematic of the Incrementer Unit.

---

<sup>1</sup>This number becomes `0x7ffff80` at the address pins because of the shifting of the PC in the Tags Unit.

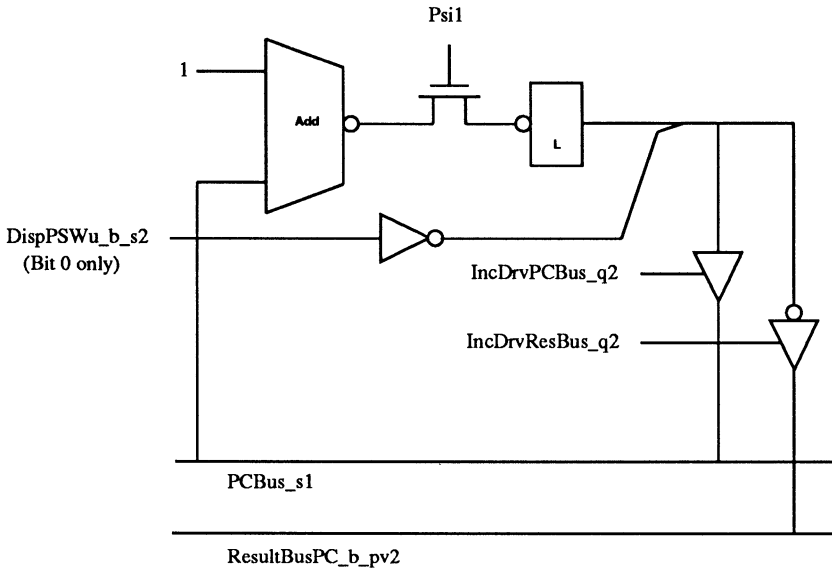


Figure 6.3: PC Incrementer Unit.

### 6.1.8 The PC Chain

The PC Chain is used to save values of the PC for instructions that are currently in execution. The front part of the chain is clocked on  $\psi1PC$  and is required for sequencing values of the PC during cache miss cycles. The remainder of the chain is needed to save the three PC values needed to restart the machine in the event of an exception or trap. Figure 6.4 shows a schematic of the PC Chain.

The  $\psi1PC$  clock is used only in the PC Chain and is generated by qualifying the  $\phi1$  clock with *PadMiss\_v1*. During all cache misses (internal and external) the rest of the machine is frozen because  $\psi1$  does not rise but during internal cache misses, parts of the PC Chain must still be clocked which is the reason for the existence of  $\psi1PC$ .

During an internal cache miss, the instruction that missed, *PCmiss*, and the next instruction to be executed, *PCnext*, are fetched from the external cache. This is done by clocking these addresses through the front part of the chain twice. The first time is to send the addresses off the chip and the second time is to do the write into the tags and cache, and start the “fetch” of the instruction. During internal cache misses, *CMPCDrvPCBus\_q2* is used to enable values from the PC Chain onto

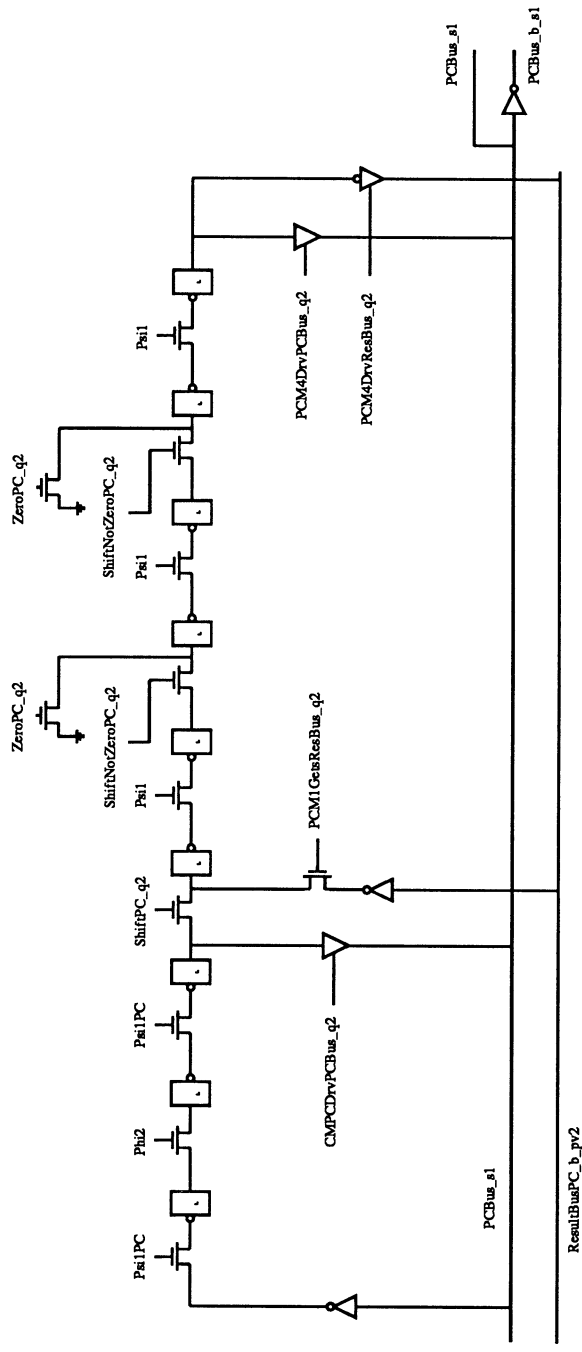


Figure 6.4: Schematic of the PC Chain.

the PC Bus. This sequencing is explained in more detail in Section 6.2.3.

The back end of the PC Chain saves the values needed to restart the machine after an exception or trap. It is clocked on  $\psi_1$  since it is state that needs to be preserved during any cache misses. During  $\phi_2$ , shifting is qualified by the shift bit in the PSW,  $PSW_s_s2$  and whether certain instructions are being executed. See Section 6.1.3 for the exact equations.

When instructions in the branch slots are squashed it is necessary to “no-op” the instructions in the PC Chain so that if an interrupt occurs, the squashed instructions are not restarted. This must be done because the branch instruction may already have fallen off the end of the pipe so that it will not be re-executed. On restart there is no way to determine whether the instructions in the slots were being squashed, so they are changed to “no-ops” by zeroing the PC Chain and making sure there is a *nop* instruction at Address 0.

The back part of the PC Chain can be loaded from the Result Bus during *movtos pcm1* (PC-1) instructions. It can drive the PC Bus during *jpc* or *jpcrs* instructions and the Result Bus during *movfrs pcm4* (PC-4) instructions.

### 6.1.9 The Cache Miss Finite State Machine

The Cache Miss Finite State Machine (CMFSM) is used to sequence the processor through an internal cache miss sequence. The state diagram for the machine is shown in Figure 6.5 and the schematic is shown in Figure 6.6. This machine is constructed out of the same cells as those used for the datapath with a few minor modifications. The input is *Hit\_v1* which comes from the Tags Unit and it is qualified with feedback from the output of the machine to prevent the machine from restarting a miss sequence once it is already in one. In the worst case the machine should do two cycles with *IMiss\_s2* on and two cycles with it off corresponding to the two cache miss cycles and executing the two instructions.

During reset and exceptions the machine is forced into the run state meaning no cache miss signals are on.

### 6.1.10 The Squash Finite State Machine

The Squash Finite State Machine is a very simple 3-state machine implemented as a two-stage shift register. The state diagram is shown in Figure 6.7 and the schematic is shown in Figure 6.8. In the squash state,



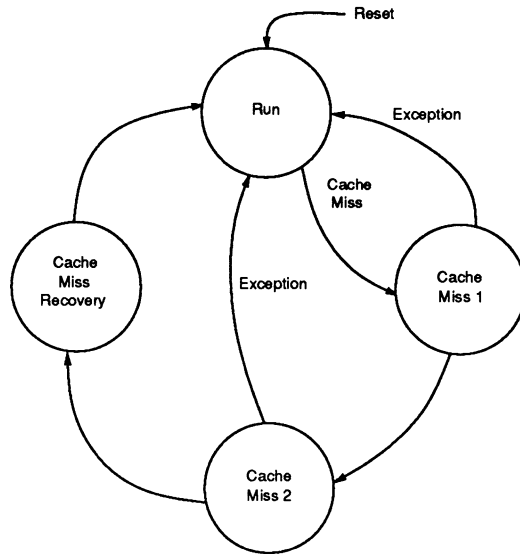


Figure 6.5: State diagram for Cache Miss FSM.

there are two cycles when the PC Unit is forced into increment mode and the instructions are turned into “no-ops”. There is feedback to the input such that it is possible to put any type of branch into the slots of any other branch instruction.

A *Reset* signal will cause the processor to squash the next two instructions. This is necessary because when the machine starts, the instructions already in the pipe will be indeterminate and must be prevented from affecting the reset sequence (for example, they could be branches).

Exceptions also will cause the squash signal to rise for two cycles to cancel the instructions still in the pipe when the exception occurred.

During *ICacheTest.s1* the PC must be forced into increment mode. This is done by simply forcing the squash signal on.

## 6.2 Internal Cache Tags

The Tags Unit provides the basic control signals that allow the internal RAM to be used as an instruction cache for MIPS-X. This unit is located at the end of the datapath on the right-hand side and is about 2344  $\mu\text{m}$  high by 1715  $\mu\text{m}$  wide.

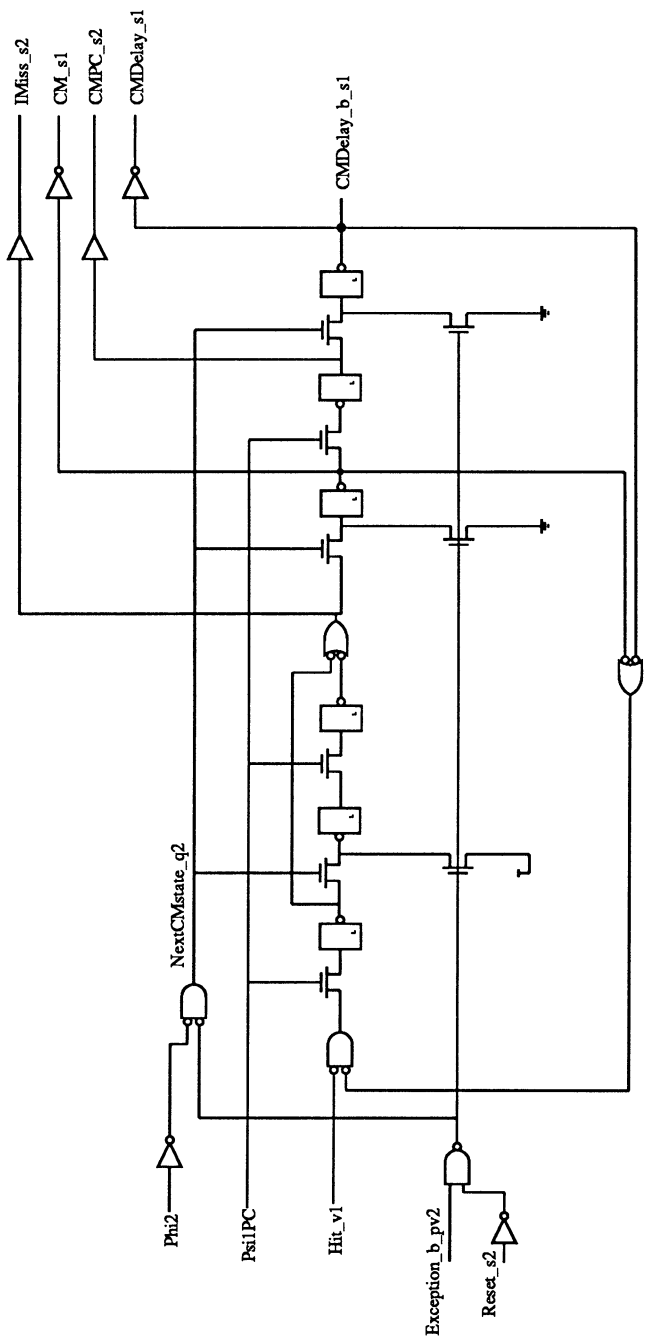


Figure 6.6: Schematic for Cache Miss FSM.

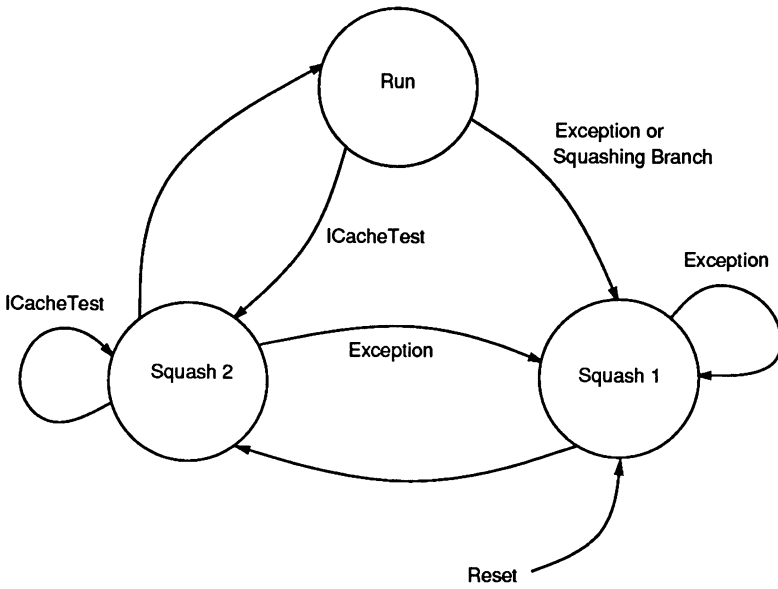


Figure 6.7: Squash FSM State Diagram.

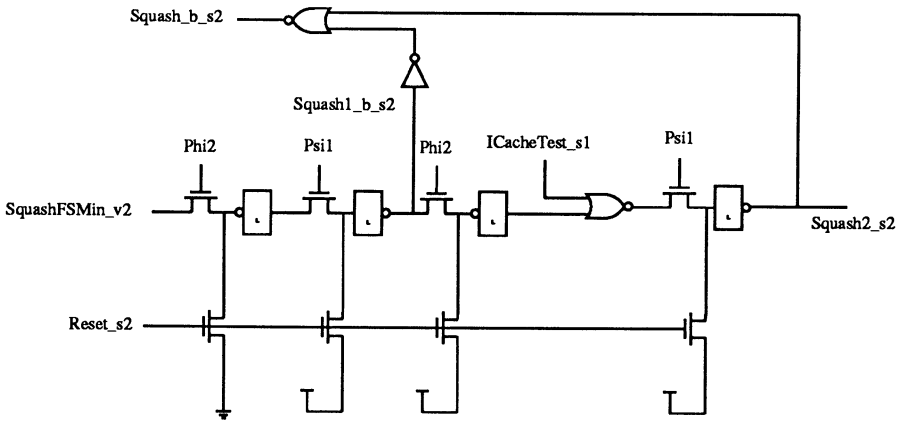


Figure 6.8: Schematic for the Squash FSM.

The main functions performed by the Tags Unit are:

- Detection of a cache *hit/miss*.
- Generation of the block select (tag select) signals.
- Driving the Address Bus with the appropriate value.
- Tag maintenance and replacement.
- Maintenance of the valid word indicators.

The main parts of the Tags Unit are:

<b>Tag Store</b>	Memory where tags are stored and compared.
<b>Valid Bit Store</b>	Memory where the valid word indicators are stored.
<b>Address Driver</b>	Drives the correct value onto the Address Bus.
<b>Ring Counter</b>	Implements the tag replacement algorithm.
<b>Tag Control</b>	Various logic that glues everything together.

The following sections describe the Internal Cache organization, the Tags Unit interface to MIPS-X, its timing, and the hardware associated with each piece of the Tags Unit.

### 6.2.1 Internal Cache Description

Figure 6.9 shows a schematic of the Internal Cache (ICache) organization as well as the usage of the Program Counter (PC) bits.

The ICache is organized into four *sets* and eight *blocks* as shown in Figure 6.9. Bits [26,27] of the PC index into one of the four sets, followed by an 8-way associative tag compare of the PC against the tags in the selected set. If the tag is found in the set we have a *block hit*.

Tags are 24 bits long and consist of bits [0,3..25] of the PC. Note that bits [1,2] are not tagged. Also, each tag contains an extra (*reset*) bit that allows invalidation of all tags upon reset.

The *line size*, the amount of cached data associated with each tag, is sixteen 32-bit words. Bits [28..31] of the PC are used to select one of these words. Since MIPS-X fetches only one word at a time, a cache line can simultaneously have invalid words (not resident in the cache) as well as valid ones. A *valid bit* is associated with each word in the cache to indicate whether or not the word contains valid data. A *cache hit* occurs when there is a *block hit* and the corresponding valid bit is set.

We can now calculate the amount of storage required in the Tags Unit for the Tag Store and Valid Bit Store. Each of the four sets has a

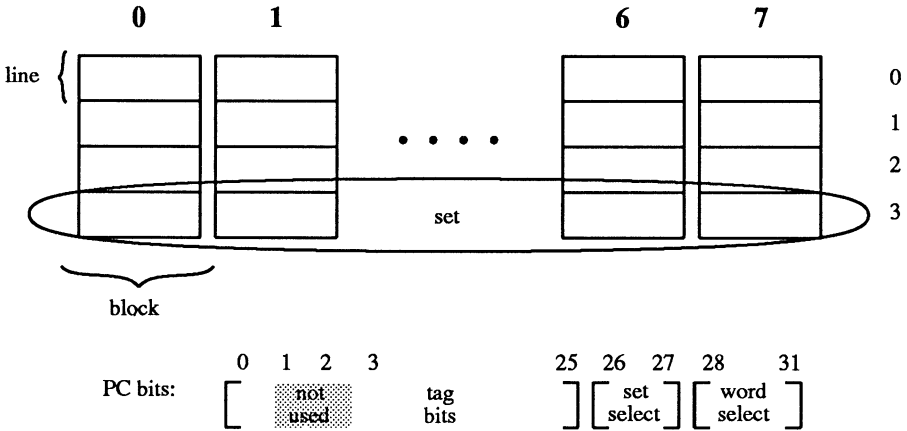


Figure 6.9: Internal Cache Organization.

tag for each of its eight blocks, so there are a total of 32 tags. Since each tag requires 25 bits, we require  $32 \cdot 25 = 800$  bits in the Tag Store. Each tag corresponds to a 16-word line, so 16 valid bits are required for each tag. The Valid Bit Store therefore requires  $32 \cdot 16 = 512$  bits.

The tag replacement algorithm is a variation of FIFO (First In First Out) and is implemented by means of an 8-bit Ring Counter. At any time, the Ring Counter points to the block into which a new tag should be placed (possibly replacing another tag). The Ring Counter is incremented (1) after a tag is replaced due to a *block miss*, or (2) when the Ring-Counter points to the same block in which a *block hit* occurred. Upon reset, the Ring Counter is set to point to block 0.

When a tag is replaced, the valid bits of the whole line are reset (*smashed*). When the instruction that caused the miss is read from memory and written into the ICache, the corresponding valid bit is set.

To be able to perform the tag compare and the valid bit access in one cycle, the valid bits are kept in a separate memory (Valid Bit Store) and not with the instructions, as is usually the case.

### 6.2.2 Tags Interface

The signals connected to the Tags Unit are shown in Table 6.2, and their approximate location in the layout is shown in Figure 6.10.

signal	bits	timing	direction	connected to Unit
Reset_s2	1	stable $\phi 2$	input	Eface
ICacheTest	1	stable $\phi 1$	input	Eface
CacheDisable	1	stable $\phi 1$	input	Eface
hit_v1	1	valid $\phi 1$	output	PC Unit
CMDelay_s1	1	stable $\phi 1$	input	PC Unit
IMiss_s2	1	stable $\phi 2$	input	PC Unit
PCBus_s1	32	stable $\phi 1$	input	PC Unit
PCBus_b_s1	32	stable $\phi 1$	input	PC Unit
ResultBus_b_pv2	32	valid $\phi 2$	input	Execute
AddressBus_v2	32	valid $\phi 2$	output	Eface
TagSelect_s2	8	stable $\phi 2$	output	ICache
phi1	1	clock $\phi 1$	input	Eface
phi2	1	clock $\phi 2$	input	Eface
phi2_b	1	clock $\phi 2$	input	Eface

Table 6.2: Signals connected to the Tags Unit.

The functions performed by each signal are<sup>2</sup>:

<b>Reset_s2</b>	Forces the Ring Counter to point to block 0 and, if ICacheTest is not set, stores a 1 in the <i>reset bit</i> of all tags.
<b>ICacheTest</b>	Sets the cache in test mode whereby all locations of the cache can be sequentially written or read.
<b>CacheDisable</b>	Forces a cache miss to occur on every fetch.
<b>hit_v1</b>	Determines whether a cache hit/miss occurred.
<b>CMDelay_s1</b>	Instructs the tags to set the valid bit of the corresponding PC.
<b>IMiss_s2</b>	Determines what gets driven onto the Address Bus: the PC Bus (when IMiss_s2 is 1) or the Result Bus.
<b>PCBus_s1</b>	The address of the instruction to be fetched.
<b>PCBus_b_s1</b>	The complement of the address of the instruction to be fetched.

---

<sup>2</sup>All of the signals are active high.

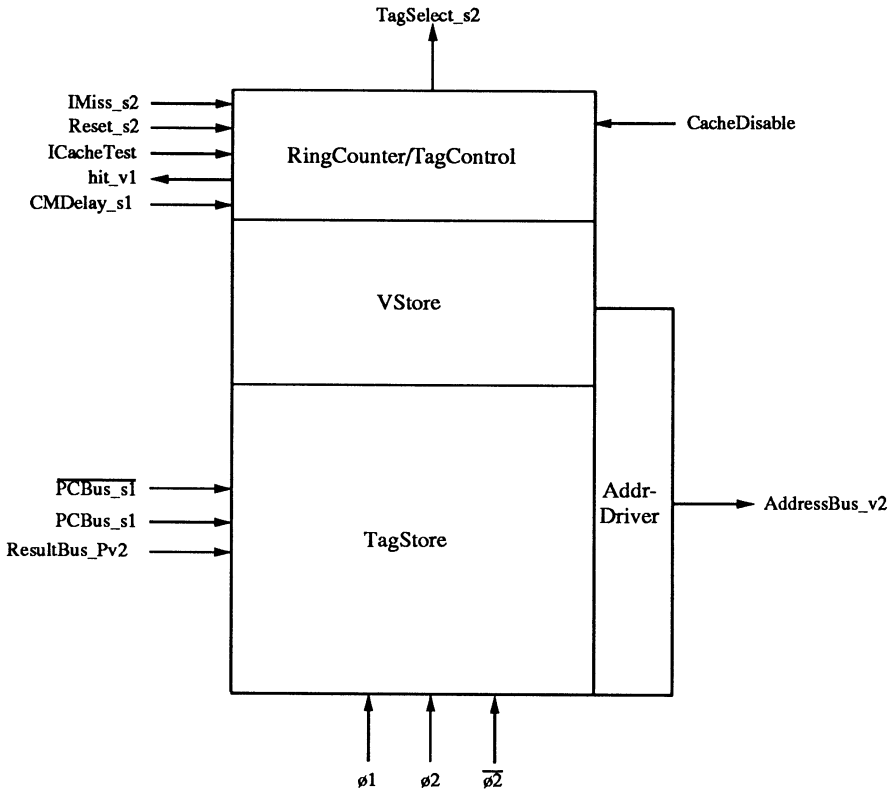


Figure 6.10: Interface to the Tags Unit.

- ResultBus\_b\_pv2** The result of an ALU operation. This signal is only used to drive the Address Bus.
- AddressBus\_v2** The address that goes off chip.
- TagSelect\_s2** Three bits of the address are first fully decoded into the eight lines that comprise this signal before being sent to the ICache RAM. One of the lines will go high, to select the block in which a hit occurs

Since the PC Unit maintains the state of the machine, most operations performed by the Tags Unit are guided by the signals supplied by the PC Unit, such as *IMiss\_s2* and *CMDelay\_s1*.

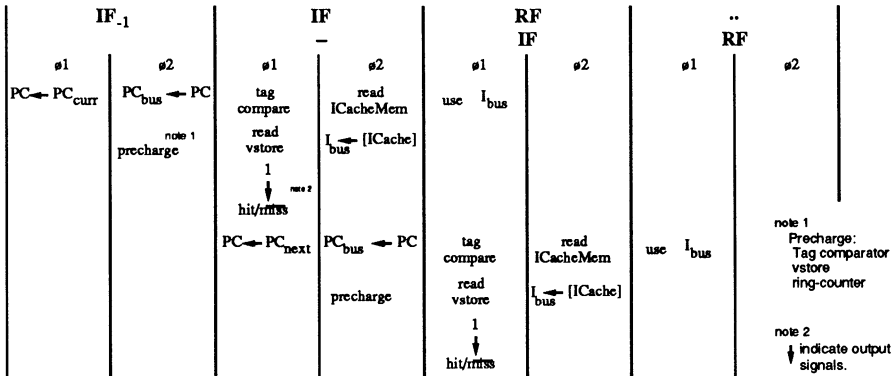


Figure 6.11: ICache timing with no internal cache misses.

### 6.2.3 Timing

Most operations performed by the Tags Unit occur on  $\phi 1$  and most of the input control signals from other units should be stable  $\phi 1$ . All elements within the Tags Unit are precharged on  $\phi 2$  and evaluated on  $\phi 1$ . The timing of two successive ICache accesses with no internal cache misses is shown in Figure 6.11. In the absence of cache misses, the following takes place:

**IF<sub>-1</sub>:** This is the cycle preceding the fetch of an instruction. During  $\phi 1$  the PC is computed, and during  $\phi 2$  it is latched onto the PC Bus.

**IF:** During  $\phi 1$  a tag compare and a valid bit access are performed and a cache miss/hit is determined. If no miss occurs, then the block number in which the hit was detected is latched. During  $\phi 2$  this value is used to read the instruction from the cache onto the Instruction Bus (I bus).

**RF:** The Instruction Bus contains the instruction to be executed. The instruction is decoded and the source registers are accessed.

As long as there are no cache misses the machine will continue to fetch instructions from the cache in this manner.

The timing of a cache miss is shown in Figure 6.12 and proceeds as follows:

**IF<sub>-1</sub>:** The PC is computed during  $\phi 1$  and latched onto the PC Bus during  $\phi 2$ .

**IF:** During  $\phi 1$  a tag compare and a valid bit access are performed,



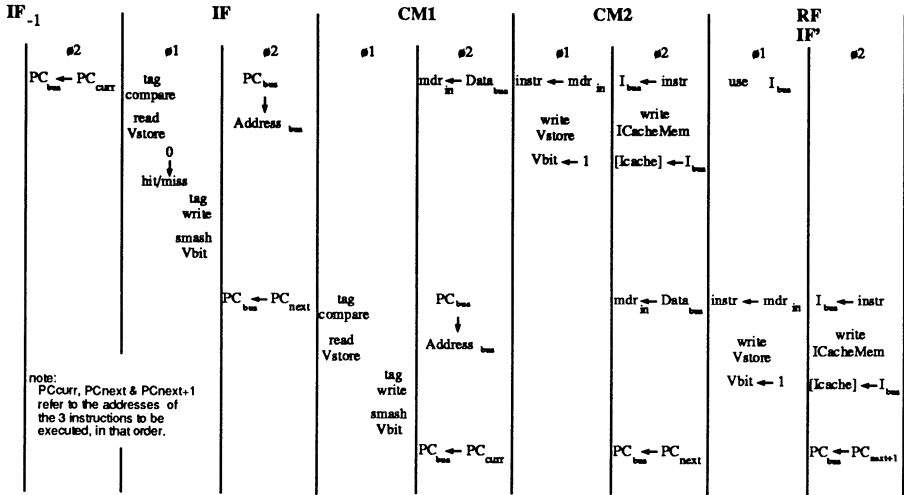


Figure 6.12: ICACHE timing during internal cache misses.

and a *block miss* situation occurs. The new tag value is written into the tag memory and the valid bits of the corresponding line are smashed (set to 0). For all this to happen on the same active clock ( $\phi_1$ ), the tags use a self-timed signal that indicates when the tag compare is done.

During  $\phi_2$ , after detecting a cache miss,  $IMiss\_s2$  will become active causing the previous value of the PC Bus (latched during  $\phi_1$ ) to be driven onto the Address Bus and then onto the Address pins. At this point the machine will go into the cache miss states: CM1 and CM2.

**CM1:** The external memory will provide the instruction at address  $PC_{curr}$  which will be latched into MDR<sub>in</sub> during  $\phi_2$ . At the same time the PC of the instruction that caused the miss ( $PC_{curr}$ ) will again be latched onto the PC Bus.

**CM2:** During  $\phi_1$   $CMDelay\_s1$  will become active, causing the appropriate valid bit to be set. During  $\phi_2$  the instruction that caused the miss will be loaded onto the Instruction Bus and written into the cache.

**RF:** The Instruction Bus now contains the instruction to be executed.

The above sequence applies to a cache miss due to a block miss, in which case a tag needs to be replaced. For the case of a cache miss due to an invalid word (block hit and 0 valid bit), the tags are not written and

the valid bits are not smashed. Nonetheless, since the missing instruction still has to be fetched from external memory, the rest of the timing sequence applies to this kind of cache miss as well.

Note that the machine requires two cycles to fetch an instruction from memory, one to read the external memory, and another to write the instruction into the cache. The machine takes advantage of this: instead of leaving the external bus unused during CM2, MIPS-X fetches the next instruction from memory during this cycle (provided it is not in the cache already).

This is shown in Figure 6.12, where the fetch of the next instruction (corresponding to  $PC_{next}$ ) is seen to miss during CM1. The instruction gets read from memory during CM2, and is finally written into the cache during IF' (the ' implies that this is not a regular fetch since the instruction is not read from the ICache; it was loaded into MDRin at the end of CM2).

In the case of a block hit and an invalid word, the Ring Counter will be incremented if it happens to point to the same block in which the tag was found. This guarantees that during CM2,  $PC_{curr}$  will always generate a block hit and  $PC_{next}$  will never overwrite the tag that corresponds to  $PC_{curr}$ .

It is important to note that a tag compare is performed on every  $\phi 1$ ; this is necessary since the block number in which a tag is stored is needed before the Valid Bit Store or the ICache can be accessed. This is also one of the critical paths in the machine, since in  $\phi 1$  we must do a tag compare to determine a block hit, and then in the same cycle either read out the corresponding valid bit in case of a block hit or write the new tag and smash the valid bits in case of a block miss.

To accomplish this, the Valid Bit Store operates as follows: the word and set select are done at the beginning of the cycle; when the tag compare is done, the appropriate valid bit (the one associated with the block that hit) is multiplexed out. If the tag compare fails (block miss) then no valid bit is selected and a narrow *write pulse* is sufficient to write the tags and smash the valid bits.

Similarly, the tag compare is done on all 32 tags at the same time, and the block select circuitry will multiplex out the eight compare signals within the desired set. An OR of the eight compare signals is then performed to determine a block hit/miss.

### 6.2.4 Tag Store

The Tag Store section is comprised of the following elements:

<b>Tag Driver</b>	Drives the PC Bus onto the tag cells.
<b>Tag Array</b>	Array that stores the tags and does the tag compare.
<b>Done Array</b>	Provides the self-timed <i>tagDone</i> signal.

#### Tag Driver

The function of the Tag Driver is to drive the PC Bus onto the tags and address drivers. It takes the PC Bus as an input and generates a *qualified PC Bus*.

Since the tag compare is accomplished by pulling down on a  $\phi 2$ -precharged *matchLine*, the tag array requires a qualified- $\phi 1$  PC Bus to evaluate the *matchLine*. On the other hand, the Address Driver requires that the PC Bus be stable during  $\phi 1$  to latch it. This is also true for the tag array, since it gets written with a qualified- $\phi 1$  signal. The Tag Driver solves this problem by taking advantage of the  $\phi 1$ - $\phi 2$  clock gap to provide the hold time required by the latches. This is accomplished by driving its outputs during  $\phi 1$ , leaving the outputs undriven during the  $\phi 1$ - $\phi 2$  gap (relying on charge store to retain a '1' value), and then pre-discharging its outputs when  $\phi 2$  becomes active.

The cells of the Tag Driver are designed to match the pitch of the datapath. A schematic of the tag driver is shown in Figure 6.13. The output transistors are  $21 \mu\text{m}$  wide by  $2 \mu\text{m}$  long, all other transistors are  $6 \mu\text{m}$  wide by  $2 \mu\text{m}$  long.

#### Tag Array

The Tag Array consists of the 32 tags. The tags are laid out side by side, with the qualified PC Bus and the Result Bus running horizontally through them, and a *tagWrite* and a *matchLine* running vertically through each tag.

Each tag consists of 24 tag cells and one reset cell. The tag cells are arranged vertically, with bit 25 of the PC at the top, and the reset cell at the bottom.

The tag cells, shown in Figure 6.14, are built around six-transistor fully static CMOS RAM cells with n-channel access pass transistors. The cell is written by raising the *tagWrite* signal and driving complementary bit lines (*PCBus\_q1* and *PCBus\_b\_q1*) to the access transistors. The

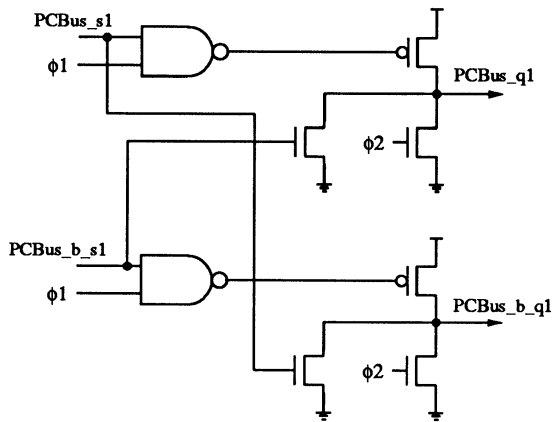


Figure 6.13: Tag Driver cell.

tag comparators are built into the cells and simply pull down on the matchLine when the corresponding PC Bus bit differs from the content of the cell. Note that the cell's contents are never read out, the outcome of the compare (the match line) is the only output. The matchLines are precharged on  $\phi2$  and evaluated on  $\phi1$ . All transistors are minimum size devices ( $6\ \mu\text{m}$  wide by  $2\ \mu\text{m}$  long), and the size of a tag cell is  $45\ \mu\text{m}$  high by  $62\ \mu\text{m}$  wide. The tag cells are designed to flip horizontally and be placed adjacent to each other; the layout for two such cells is shown in Figure 6.15.

The tag reset cells are nearly identical to the tag cells. The only difference is that the ground line of the feedback inverter is connected to the Reset line instead of ground. The reset cells are driven by Tag Driver cells that have their *PCBus* and *PCBus.b* inputs connected to  $V_{dd}$  and ground respectively. When Reset is raised, a 0 is written into all the reset cells; when a tag gets written, a 1 is written into the corresponding reset cell. After a reset none of the tags will match, effectively flushing the tags. Note that care must be taken to ensure that the first instruction executed after a reset is written into the tags. This is because Reset is a stable- $\phi2$  signal, and it may be lowered too late in the cycle (during  $\phi1$ ) to allow the tag to be written. In any event, since the tag write takes place late in  $\phi1$ , after a tag compare, this is not a problem. Figure 6.16 shows the schematic of a reset cell.

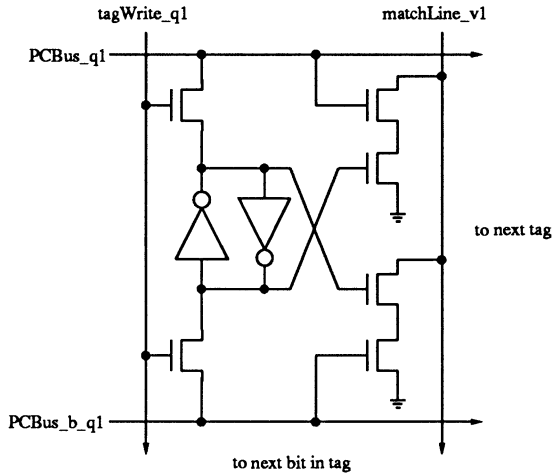


Figure 6.14: A Tag cell.

### Done Array

The Done Array provides a self-timed signal (*tagDone\_v1*) that indicates when the tag compare is finished. The *tagDone* signal is generated by means of a *dummy* matchLine that is guaranteed to fall after all other matchLines.

The Done Array contains 25 cells connected to the dummy match-Line. Each of these cells provides a capacitive load similar to the load present on a match-line due to the compare transistors of a tag cell. The lower 24 cells are only used as capacitors on the dummy match line; the topmost cell in the array (the 25<sup>th</sup>) contains two sets of transistors, one of which will always pull down on the matchLine during  $\phi_1$ .

The dummy matchLine should fall after all other matchLines since (1) it is pulled down by only one transistor (this corresponds to having one bit of the PC differ from a tag entry), and (2) the array is located on the right hand side of the tags where the capacitive load on the qualified PC Bus is greatest.

The dummy matchLine is buffered through a chain of inverter buffers to generate the *tagDone* signal. The buffers are necessary since the *tagDone* signal drives 64 gates whose output must be valid before  $\phi_1$  falls. The size of the buffer transistors are  $6\ \mu\text{m}$  wide by  $2\ \mu\text{m}$  long,  $20\ \mu\text{m}$  wide by  $2\ \mu\text{m}$  long, and  $50\ \mu\text{m}$  wide by  $2\ \mu\text{m}$  long. A schematic of the tag-done circuit is shown in Figure 6.17.



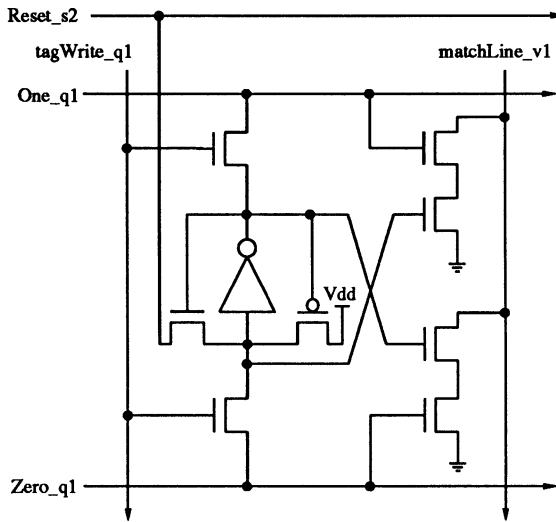


Figure 6.16: Reset Tag cell.

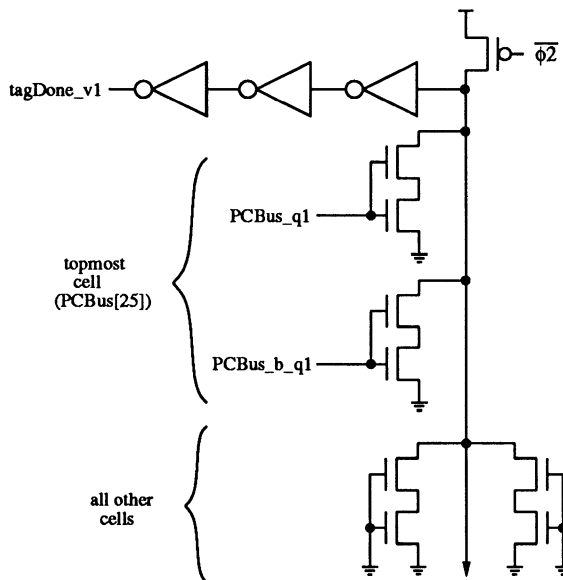


Figure 6.17: Tag Done circuit.

### 6.2.5 Valid Bit Store

The Valid Bit Store is comprised of the following elements:

<b>Valid Bit Word Decoder</b>	Selects word within line.
<b>Valid Bit Array</b>	Array that stores the valid bits.

#### Valid Bit Word Decoder

The Valid Bit Word Decoder takes bits [28..31] of the PC Bus as inputs and generates 16 qualified- $\phi_1$  decoded signals. These are used to select one of the 16 valid bits associated with a tag. Therefore, these 16 decoded signals are used as the *word lines* to select the *bit lines* of the valid bit memory. Since the bit lines are precharged on  $\phi_2$  this selection occurs during  $\phi_1$  when the bit lines are evaluated.

The decoders are laid out in diffusion. The PC Bus bits run vertically in 1<sup>st</sup>-level metal, with programming tabs providing the polysilicon gates at the appropriate places. Each decoder is simply a NAND gate with five inputs (the four *PCBus* (or *PCBus\_b*) bits and  $\phi_1$ ), followed by an inverter-buffer whose transistors are 22  $\mu\text{m}$  wide by 2  $\mu\text{m}$  long. The decoder cells are designed to match the vertical pitch of the valid bit memory cells and they are located on the left side of the Valid Bit Store.

#### Valid Bit Array

The Valid Bit Array consists of the 512 valid bits, organized as 32 lines of 16 bits each. Each line of valid bits is located directly on top of its corresponding tag, with the 16 valid bit cells running vertically.

The *matchLine* (from the tag below), the *tagWrite* signal, and the *bitLine* run vertically through each line in 1<sup>st</sup>-level metal. The word lines (from the Valid Bit Word Decoders) and the power lines run horizontally through the 32 lines in 2<sup>nd</sup>-level metal.

The valid bit cells consist of fully static CMOS RAM cells with one n-channel access transistor, as shown in Figure 6.18. The bit lines are precharged on  $\phi_2$ , and the cell is read out by driving its word line high during  $\phi_1$ . Also, a 0 may be written to each individual cell within the line by pulling down on the bit line; this is used to set a valid bit. Note that a 1 cannot be written into the cell; invalidation of a line (smashing the valid bits) is accomplished by raising the *tagWrite* signal, which serves as the ground line for the feedback inverter. Raising *tagWrite* will write a 1 on all sixteen valid bit cells associated with the tag being replaced. This



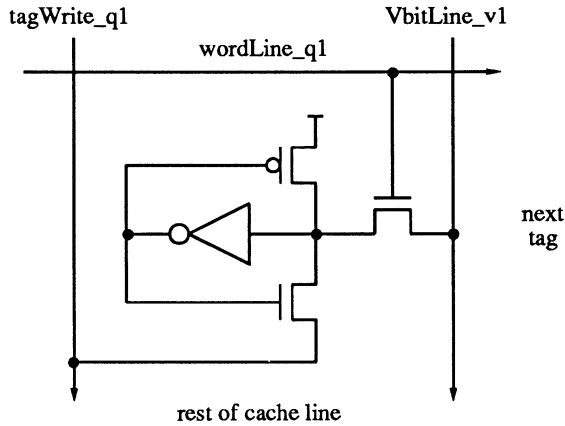


Figure 6.18: Valid Bit cell.

could have also been accomplished by incorporating another n-channel transistor into the cell, but this would have increased the size of the cell beyond the pitch constraints.

All transistors in the valid bit cell are minimal size devices ( $6\ \mu\text{m}$  wide by  $2\ \mu\text{m}$  long), except for the access transistor which is  $13\ \mu\text{m}$  wide by  $2\ \mu\text{m}$  long. The size of the cell is  $35.5\ \mu\text{m}$  high by  $42.5\ \mu\text{m}$  wide. Note that the bit line is discharged through two transistors, the n-channel feedback transistor and the n-channel transistor that drives the *tagWrite\_q1* signal. The cells are designed so they flip horizontally and vertically; four such cells are placed adjacent to each other sharing the  $V_{dd}$  and ground contacts. The layout for four such cells is given in Figure 6.19.

### 6.2.6 Ring Counter

The Ring Counter consists of an 8 bit shift register. Every bit uses 2 static half-latches; one is clocked on  $\phi_1$ , the other is clocked on  $\phi_2$ . At all times only one of the Ring Counter bits is set; the other seven bits are reset. The latches are provided with an asynchronous reset (a pull-down transistor) that will reset the Ring Counter, forcing it to point to block 0. A schematic of the Ring Counter is given in Figure 6.20.

The Ring Counter is shifted to the right (points to the next block) after a block miss or if the Ring Counter points to the same block as the one that hit. The block select lines are compared with the Ring

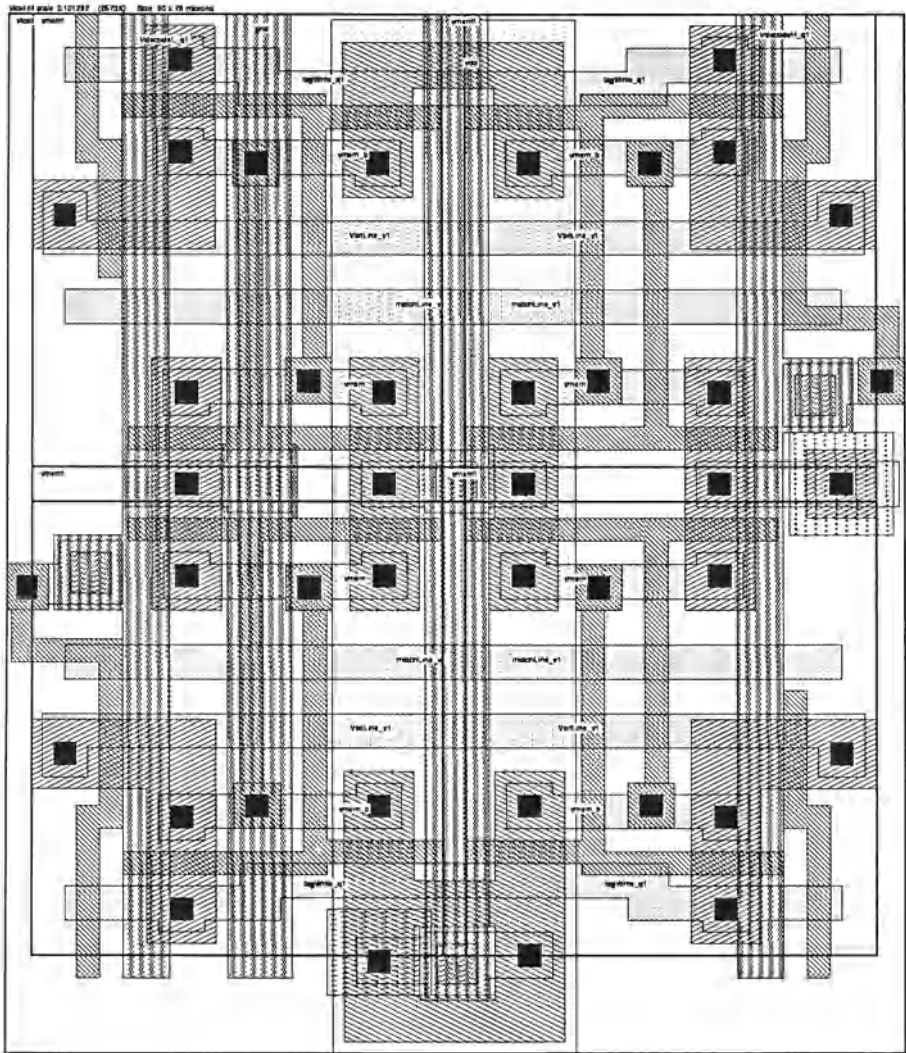


Figure 6.19: Layout for 4 adjacent valid bit cells.

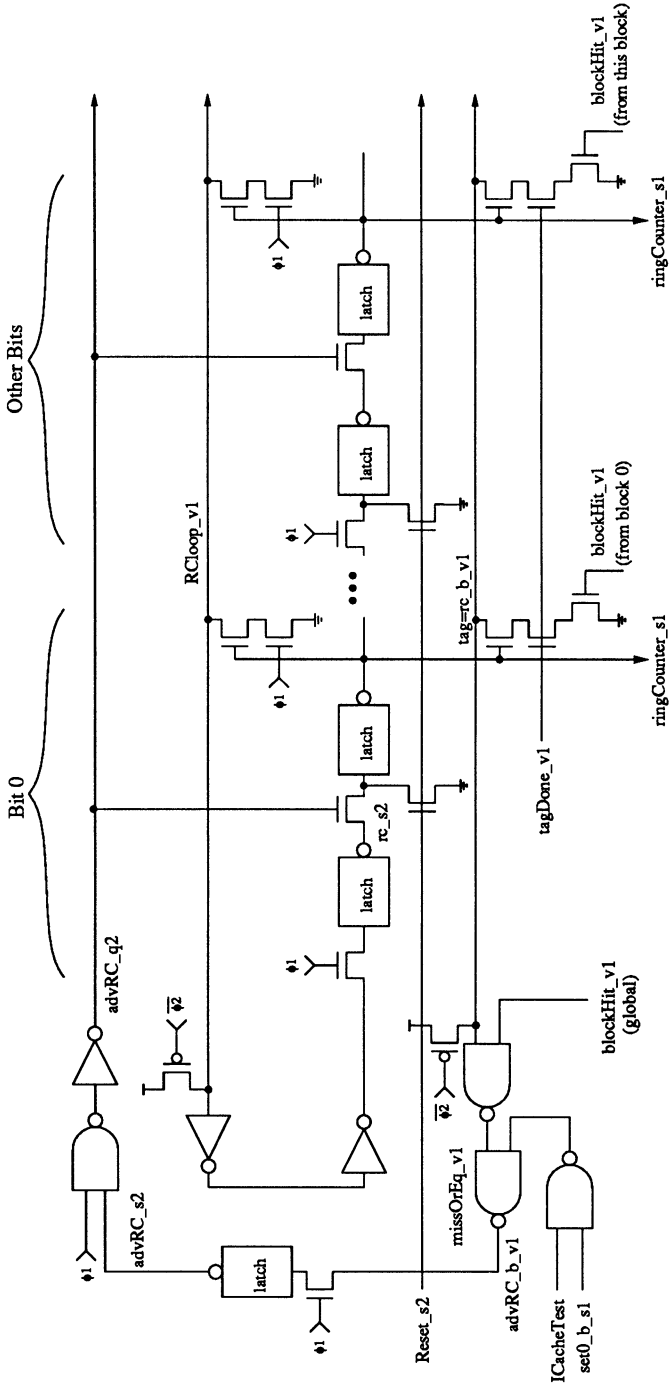


Figure 6.20: Ring Counter.

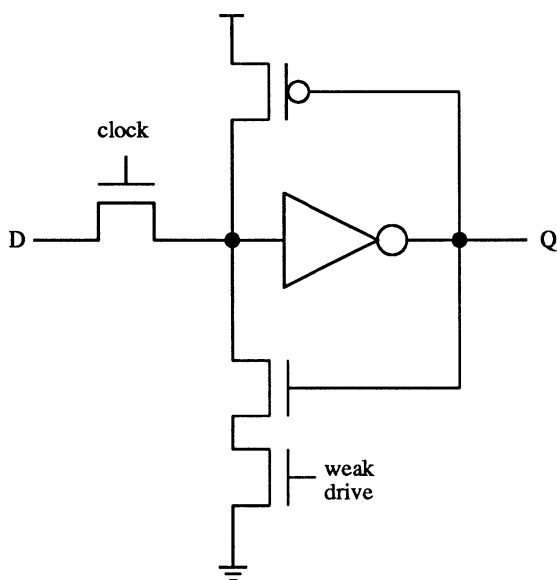


Figure 6.21: Flow Through Latch with weak feedback.

Counter bits by means of a precharged circuit. The result of this compare ( $\text{tag}=\text{rc.b.v1}$ ) and the block miss signal are latched on  $\phi 1$  and on the next  $\phi 2$  cycle will cause the Ring Counter to shift.

If the Ring Counter points to the last block (block 7) and is shifted, a 1 will be shifted into the first position. Note that the last bit is not shifted into the first position; rather, the NOR of all the Ring Counter bits is used instead. Although these two methods are functionally the same, the latter was chosen as a means of recovery in case a glitch causes all the bits to become zero. If this were to happen, a one could never be shifted back into the ring, thereby rendering the ICACHE useless since no more tags would get written.

All latches used in the Ring Counter are static flow-through latches with one n-channel access transistor. To be able to write these latches, the feedback n-channel transistor is made ‘weak’ by means of a *weak drive* signal, and the feedback p-channel transistor is a high resistance device ( $4\ \mu\text{m}$  wide by  $5\ \mu\text{m}$  long). A schematic of this latch is shown in Figure 6.21.

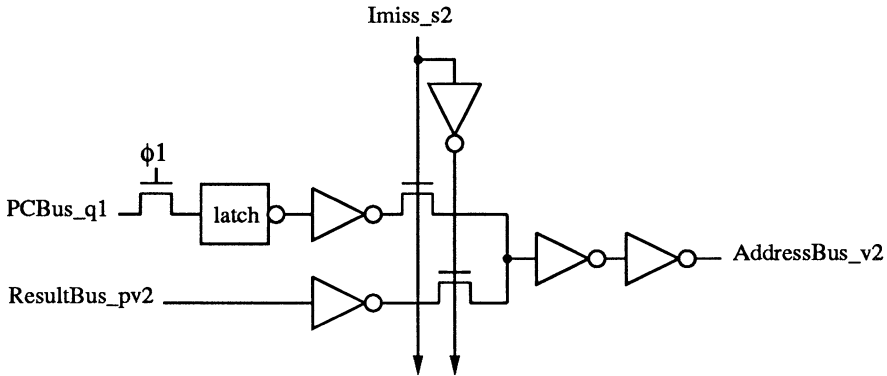


Figure 6.22: Address Driver.

### Testing Circuitry

Special control has been added to the Ring Counter to facilitate testing of the internal cache. When *ICacheTest* is active, the ICache will go into a sequential mode, where the Ring Counter is only incremented when the PC Bus selects set number 0 (determined by bits [26,27]). Since, in sequential mode, the PC Unit increments the PC every cycle, this feature allows writing to all the tags in a block before replacing it. If the normal replacement algorithm is followed, the tags would not be written in sequential order, making it more difficult to determine a cache/tags problem. This feature is easily added by the inclusion of two more NAND gates, as shown in Figure 6.20, where the *set0\_b\_s1* signal comes from the set decoders.

#### 6.2.7 Address Driver

The Address Driver takes *ResultBus\_b-pv2* and *PCBus-q1* as inputs, latches the latter and selects one of them (depending on *IMiss-s2*) to drive the Address Bus. The selection is done by means of an n-channel multiplexer circuit<sup>3</sup> followed by a chain of inverter buffers with transistor sizes  $13.5\ \mu\text{m}$  wide by  $2\ \mu\text{m}$  long and  $61.5\ \mu\text{m}$  wide by  $2\ \mu\text{m}$  long. The Address Driver schematic is shown in Figure 6.22.

<sup>3</sup>Since a large capacitance is driven by the Address Bus, a full transmission gate should be used.

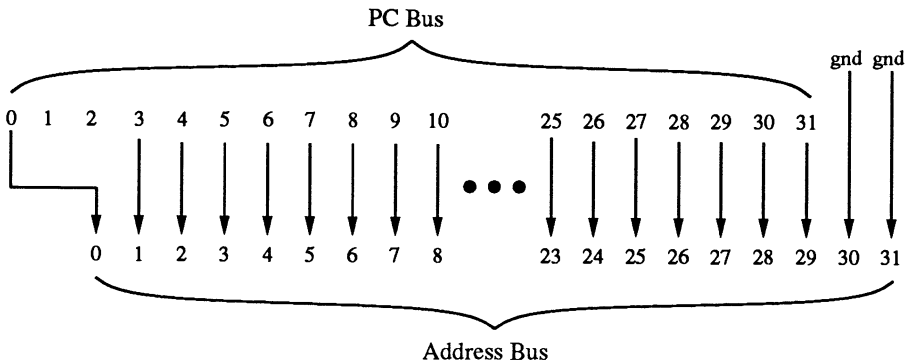


Figure 6.23: Shifting of the PC Bus onto the Address Bus.

### PC Shifting

The PC addresses generated by the PC Unit are word addresses while the data addresses on the Result Bus are byte addresses. These two must be made equivalent when presented to the external memory/cache to maintain a consistent addressing scheme. The convention is to convert all addresses to byte addresses and, since the machine fetches words, the external memory may ignore the two low order bits of the Address Bus. This conversion is accomplished by shifting the PC Bus as shown in Figure 6.23. Note that PC Bus bits 1 and 2 are not passed on to the Address Bus (these bits are not tagged either), and bits 30 and 31 of the Address Bus will always be 0 for instruction addresses. The PC shift is accomplished by the interconnect between the Tag Array and the Address Driver.

### 6.2.8 Tag Control

The tag control contains the logic that “glues” all other parts of the Tags Unit together. The functions performed by this section are:

- Generation of the cache hit/miss signal (*hit\_v1*).
- Set selection.
- Valid bit multiplexing and writing.
- Generation of the tag select signals.
- Generation of the tag write signals.

- Precharging various parts of the circuit.

A complete diagram of the control section for one cache block is shown in Figure 6.24.

### Set Selector

The Set Selector does a 1-out-of-4 decoding of bits 26 and 27 of the PC Bus and drives the decoded signals along the width of the tags section. The decoding is done by four NAND gates, as shown in Figure 6.24, followed by an inverter driver. The driver transistors are  $35.5\ \mu\text{m}$  wide by  $2\ \mu\text{m}$  long.

### Valid Bit Control

The valid bit control multiplexes the appropriate valid bit and, for valid bit writes, it drives the valid bit line low.

Multiplexing of the valid bit is accomplished by means of an n-channel pass transistor that connects each one of the 32 valid bit lines to the I/O bit line ( $Vbit\_b\_v1$ ). This line is precharged on  $\phi2$  and the appropriate pass transistor (the one in the block where a hit was detected) is turned on during  $\phi1$ , when the tag compare is finished. The logic on the gate of each pass transistor is:

$$\text{muxtctor}[j] = \text{tagDone\_v1} \cdot \text{matchLine\_v1}[j] \cdot \text{setSelect}[k]$$

where  $j$  is the number (0..31) of the corresponding matchLine (the one in which a block hit was detected, if any), and  $k$  is the number (0..3) of the addressed set as generated by the set select circuit.

Writing a valid bit is accomplished by multiplexing the valid bit line, as described above, and driving the bit line low. This is simply a large n-channel ( $26.5\ \mu\text{m}$  by  $2\ \mu\text{m}$ ) transistor controlled by the  $Vwrite\_q1$  signal, which is  $CMDelay\_s1$  qualified with  $\phi1$ . This is shown in Figure 6.24.

### Hit/Miss Signal

The circuit that generates the hit/miss signal ( $hit\_v1$ ) is shown in Figure 6.25. Each block generates a *blockHit* signal, which is a  $\phi2$ -precharged line pulled down by one of the four match lines in the block. The set select signals control four n-channel pass transistors that will allow only

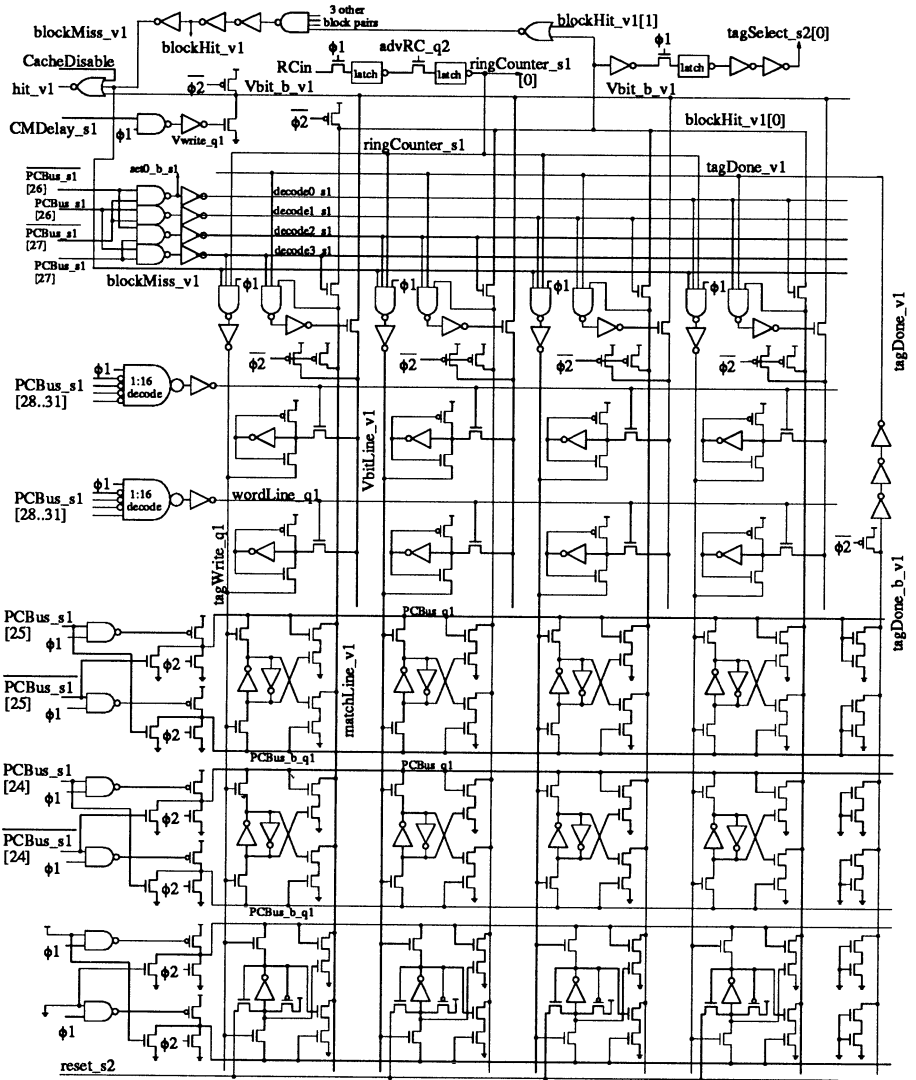


Figure 6.24: Schematic of one block of the tags, including the tagDone circuitry.



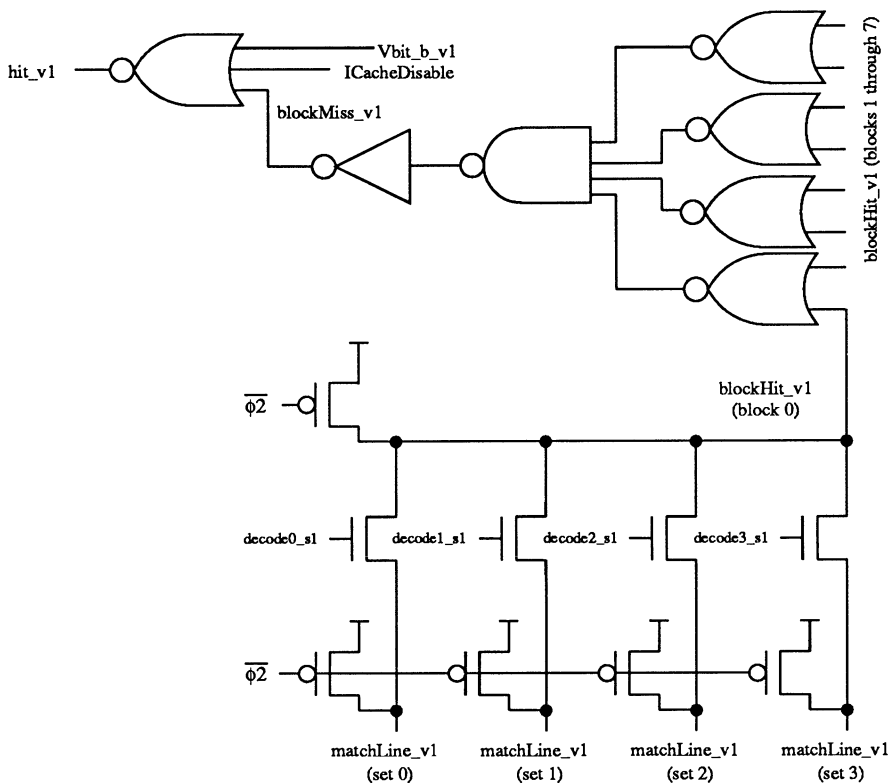


Figure 6.25: Generation of the hit/miss signal.

the match line within the selected set to pull down on the blockHit line. The resulting eight blockHit lines are ORed together to generate the “global” hit/miss signal. The OR is accomplished by means of an OR-AND combination to avoid an exceedingly slow 32-input OR gate. All the precharge and pass transistors are  $6 \mu\text{m}$  wide by  $2 \mu\text{m}$  long.

### Tag Write signals

A *tagWrite\_q1* signal is generated for each of the 32 tags. The tagWrite line is activated when the corresponding tag must be replaced, and is controlled by the following logic:

$$\text{tagWrite}[j] = \text{blockMiss\_v1} \cdot \phi1 \cdot \text{ringCounter\_s1}[j] \cdot \text{setSelect}[k]$$

where  $j$  is the number (0..31) of the corresponding matchLine and  $k$  is the number of the addressed set (decodes0\_s1..decode3\_s1).

The rise of the tagWrite signal is determined by blockMiss\_v1, which will rise sometime during  $\phi_1$  in case of a block miss. The size of the tagWrite driver transistors is 13  $\mu\text{m}$  wide by 2  $\mu\text{m}$  long.

### Tag Select Lines

The Tag Select lines, along with PCBus\_s1[26..31], form the 9-bit cache RAM address. There are eight Tag Select lines, representing a decoded 3-bit address; only one of the eight lines will be set to a 1, and they correspond to the *blockHit\_v1*[0..7] signals generated by each of the blocks. The block in which a hit occurs will be set to 1, all the others will be pulled down during the tag compare. In the case of a block miss, blockHit\_v1[0..7] will be 0. The blockHit lines are buffered by a 6/2 inverter, latched during  $\phi_1$ , and then driven to the ICache RAM by a pair of inverter buffers, as shown on Figure 6.24. The size of the buffers are 14  $\mu\text{m}$  wide by 2  $\mu\text{m}$  long, and 48  $\mu\text{m}$  wide by 2  $\mu\text{m}$  long respectively.

## 6.3 The Instruction Cache RAM

This section describes the static RAM portion of the Instruction Cache (ICache). For a description of the virtual addressing of the ICache and timing relative to the MIPS-X pipeline, see Section 6.2.1.

This RAM has a dual purpose for its existence, first as an internal instruction cache for the MIPS-X microprocessor, and second as a test vehicle for analyzing fault models and failure mechanisms. Some of the unique design decisions in this RAM were driven by the MIPS-X application—but were applied to the design in such a way as to maintain suitability as a test vehicle.

RAMs are typically asynchronous because the clocking scheme of the application is unknown and an asynchronous timing interface is the most general. Since the ICache was designed specifically for MIPS-X, it is a synchronous design using two non-overlapping clock phases. The size of the ICache is 4.2 mm by 5.7 mm and it is organized as 512 words with 32 bits per word.

All of the interface signals for the ICache RAM are shown in Table 6.3. The address lines, a0 through a5, are decoded to select one of the 64 word lines, with each of the word lines enabling eight words of 32

bits each. One of the eight TagSelect lines is used to select a word in the line.

signal	bits	timing	direction	connected to Unit
TagSelect_s2	8	stable $\phi 2$	input	Tags Unit
address	6	valid $\phi 1$	input	Tags Unit
Idata	32	valid $\phi 2$	i/o	IReg
Write	1	stable $\phi 2$	input	Eface
phi1	1	clock $\phi 1$	input	Eface
phi1_b	1	clock $\phi 1$	input	Eface
phi2	1	clock $\phi 2$	input	Eface

Table 6.3: Signals connected to the Instruction Cache.

Several test probe taps have been added to detect, diagnose, and further analyze any faults present in the circuit. For example, because the word lines are metal 1 and polysilicon, 10  $\mu\text{m}$  square metal 2 pads are placed at the top of each word line to allow wafer probing.

The timing diagram for the RAM is shown in Figure 6.26. The application requires the bus to be valid  $\phi 2$ , so the RAM is precharged during  $\phi 1$ . The write signal must be stable  $\phi 2$ , and the address is valid  $\phi 1$ . The RAM operates within the 50 ns clock cycle.

### 6.3.1 Block Description

A block diagram for the RAM is shown in Figure 6.27. The RAM precharges when  $\phi 1$  is high, and some time during  $\phi 1$  the six address lines are expected to become valid and remain valid until they are latched on the leading edge of  $\phi 2$ . The address latches are flow-through latches, allowing the address to percolate through the NAND gate predecoders into the address decoders. A given address raises two word lines, one on each side of the RAM, with 128 cells accessed by each word line. The word lines out of the address decoders are gated with  $\phi 2$ , so that all word lines are low until  $\phi 2$  goes high, allowing the selected word lines to rise. The selected word lines turn on the two access n-channel pass transistors for each RAM cell. The bit select circuitry is driven by eight tag lines from the tag store. The tag select lines must be stable  $\phi 2$  and at most only one tag line may be high. For a read cycle, the bit lines drive the sense amplifier, which in turn pulls down the cache bus that was precharged during  $\phi 1$  by the IReg. For a write cycle, the bus drives

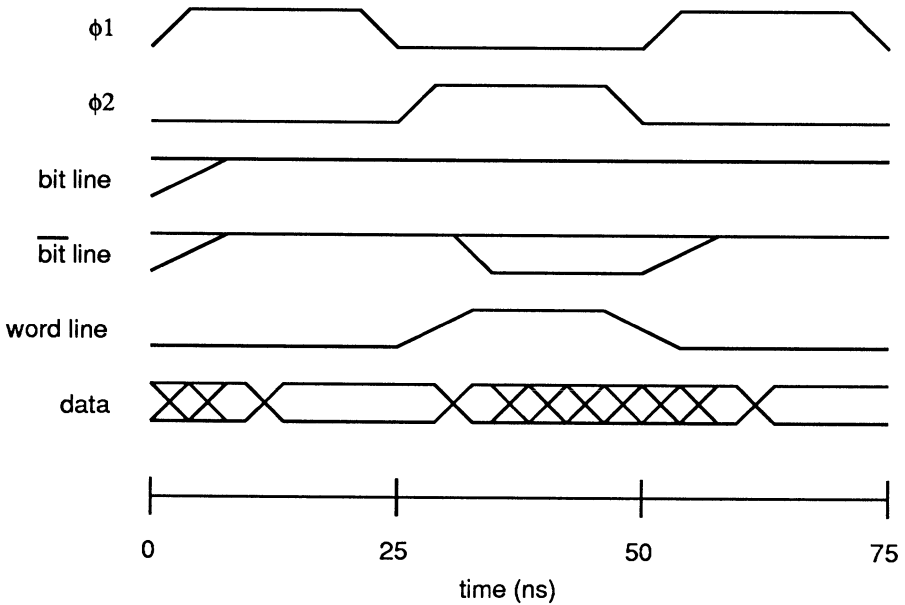


Figure 6.26: The RAM timing diagram.

the pair of inverters that make up the bit-line write drivers. The write drivers discharge the bit or bitbar lines through the bit-line select, which writes the RAM cell.

The circuit schematic for a single path through the RAM is shown in Figure 6.28.

### 6.3.2 The RAM Cell

This design uses a six transistor fully static CMOS RAM cell, that is  $39.75 \mu\text{m}$  by  $29 \mu\text{m}$ . The access transistors are n-channel pass transistors, which are adequate since all reading and writing for the cell is accomplished by pulling one bit line low. Bit lines only need to be pulled low because both bit lines are precharged before the read/write cycle. The circuit schematic is shown in Figure 6.29. The transistor widths are all  $6 \mu\text{m}$ , the lengths of the inverter transistors are  $2 \mu\text{m}$ , and the lengths of the pass transistors are  $2.5 \mu\text{m}$ . The layout of the RAM cell is shown in Figure 6.30.

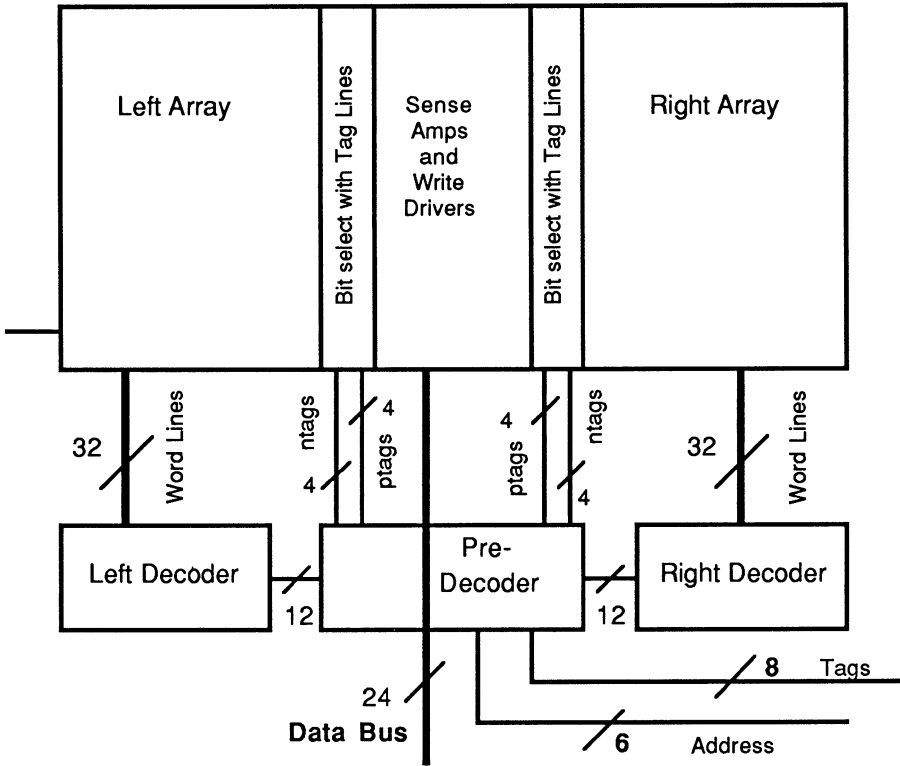


Figure 6.27: The RAM block diagram.

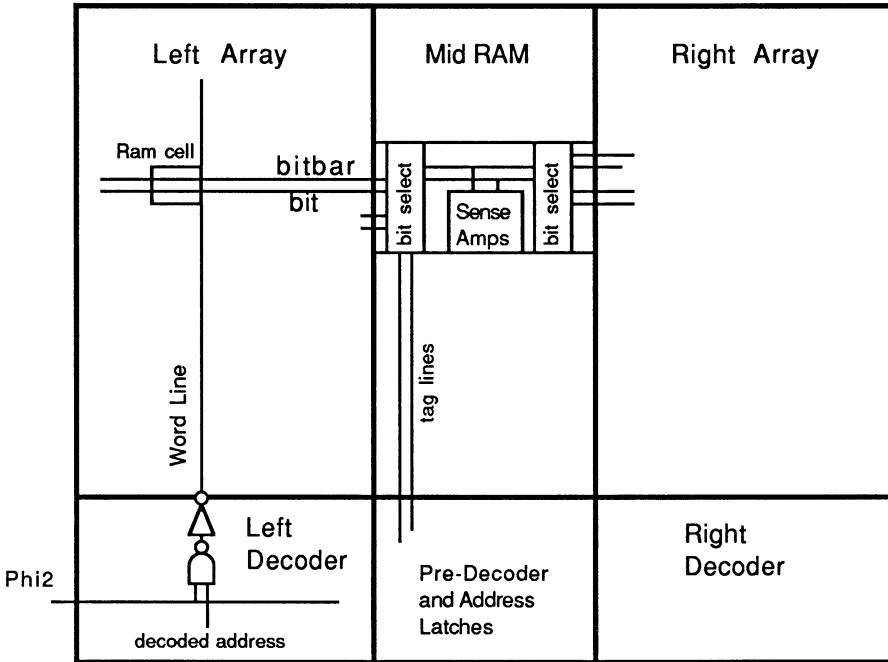


Figure 6.28: A single path through the RAM.

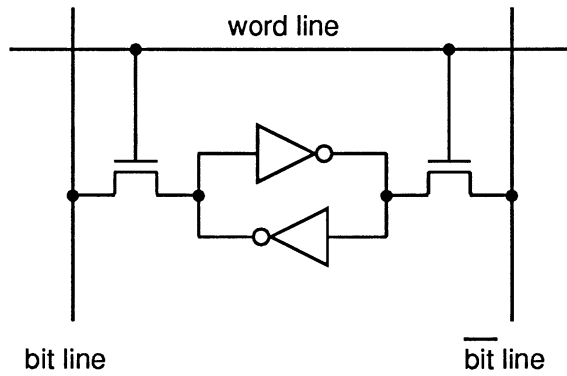


Figure 6.29: The RAM cell schematic.

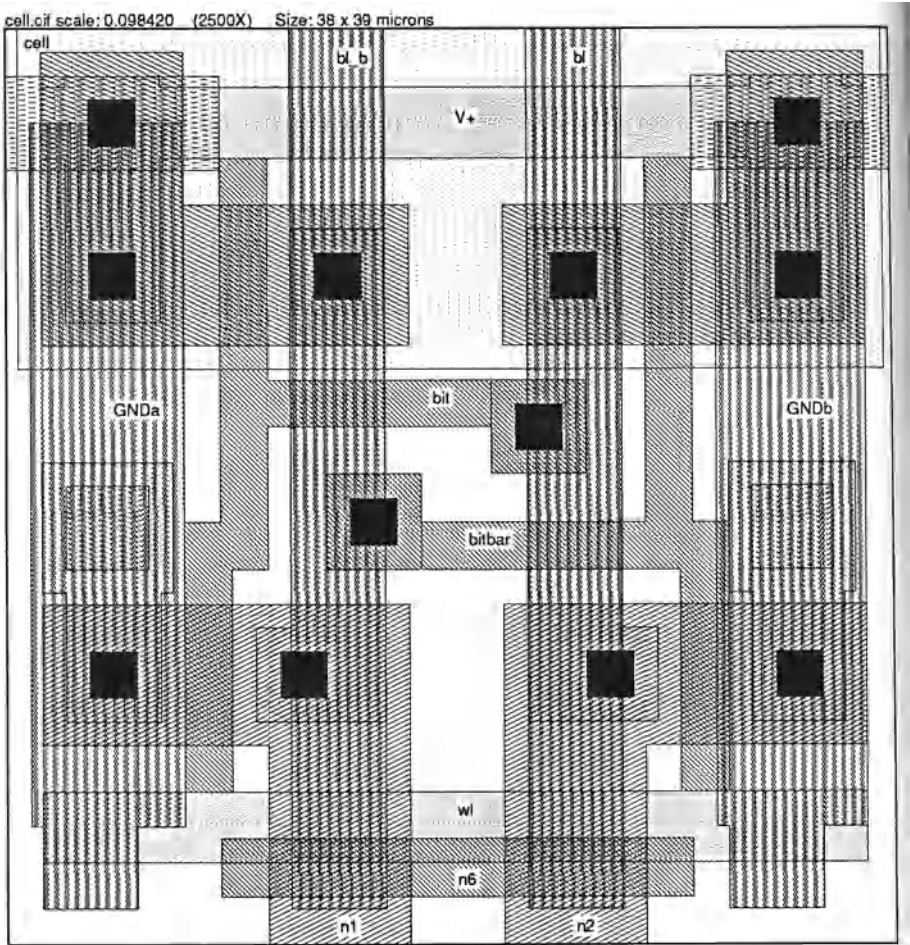


Figure 6.30: The RAM cell layout.

### 6.3.3 The Cell Arrays

RAMs are very regular structures and utilize arrays of repeated cells. The following sections describe some of the arrays and the peripheral requirements to interface the arrays to the rest of the layout.

#### Four Cells

The layout for four adjacent memory cells is shown in Figure 6.31. To insure balanced bit lines, each cell is flipped within an array of four RAM cells. Each word line is in metal 1 and only tapped to polysilicon every two cells. The resistance of the word line is therefore decreased at the expense of increased area. The bit lines and ground lines run parallel to each other in metal 2, while the word lines and  $V_{dd}$  utilize metal 1. A substrate tap exists for every four cells, as well as shared taps to the n-well. RAM cells share  $V_{dd}$  connections with adjacent cells. The 4cell is  $87.5 \mu\text{m}$  by  $66 \mu\text{m}$ .

#### The Half Planes

The RAM is broken into two half planes on either side of the sense amplifiers. The half planes are  $2.55 \text{ mm}$  by  $3.77 \text{ mm}$ . The block diagram for the memory array is shown in Figure 6.32. Figure 6.32a shows the bit select circuit for one bit on one side of the array. This figure shows that p-channel transistors are used to connect the sense amplifiers to the bit lines, and that n-channel transistors are used to connect the write drivers to the bit lines. This figure also shows both ptags and ntags. The ptags are simply inverted tag lines (the sense amplifiers are activated for both read and write) and the ntags are qualified by write and  $\overline{\phi 1}$  so that the write drivers are only connected to the bit lines during the second phase of a write cycle. Figure 6.32b shows the logical organization of the RAM. The eight tag lines select one of eight pieces of the RAM, four on each side. Each piece represents 32 bit line pairs selected by one tag line. Figure 6.32c shows the physical organization of the RAM. The eight pieces are interleaved so that the eight pairs of bit lines are adjacent for each bit.

#### Precharge

The bit lines are precharged from the outside ends by  $10 \mu\text{m}$  wide p-channel transistors. The precharge signal  $\overline{\phi 1}$  is a long polysilicon line



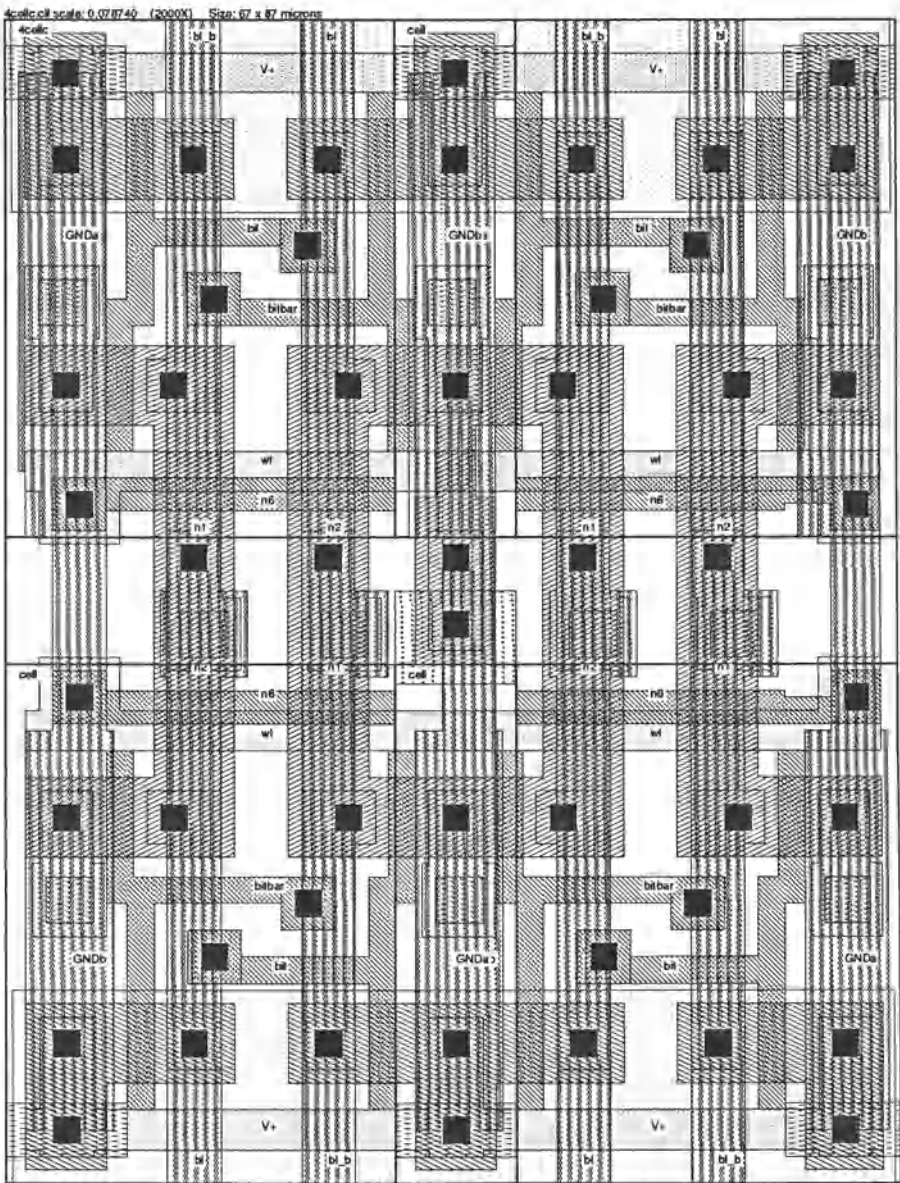


Figure 6.31: Four RAM cells and RAM cell boundary.

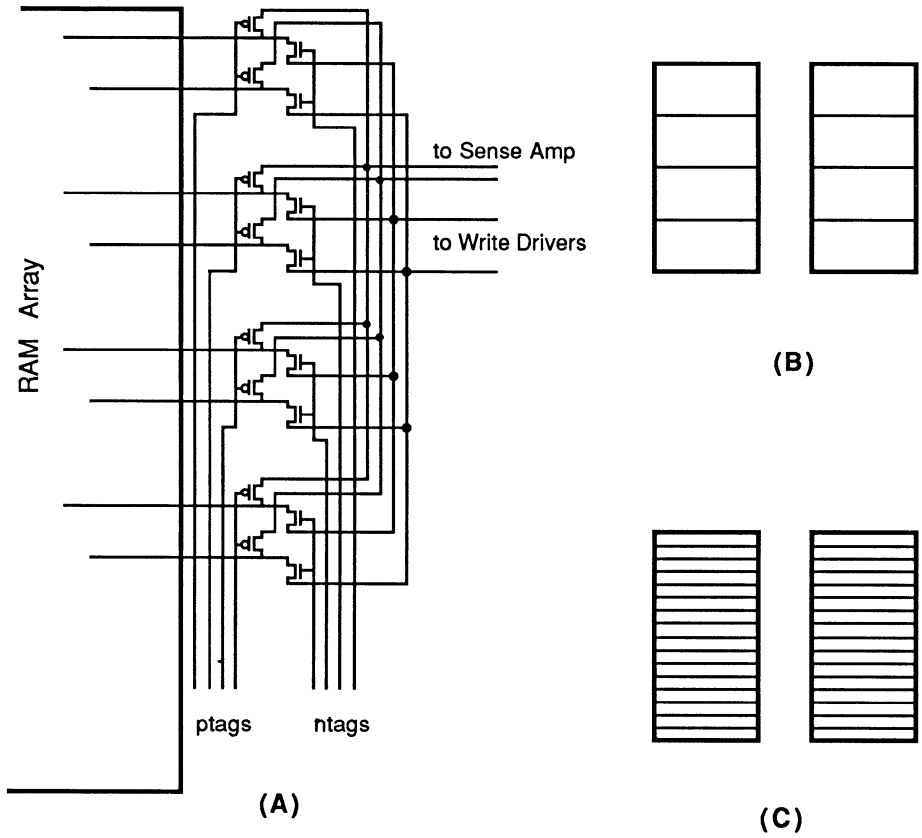


Figure 6.32: Memory array organization and bit select.

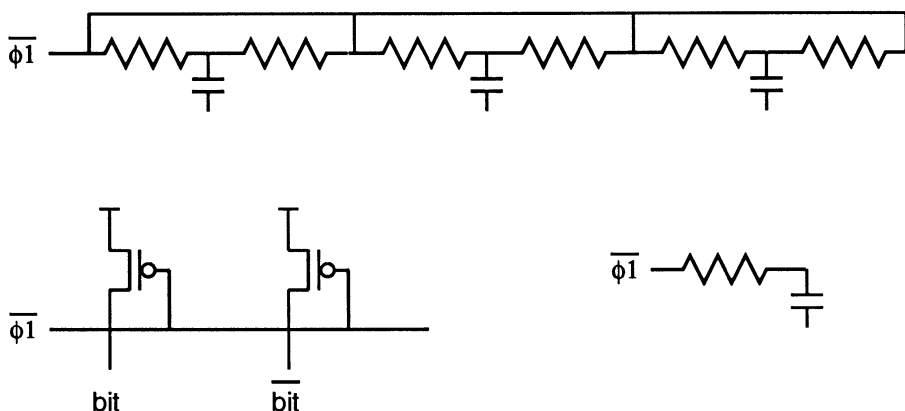


Figure 6.33: Bit-line precharge.

with a metal 1 strap tapping it four times up the side of the array. The purpose of the metal is to speed up the precharge, and the reason for only 4 taps is to spread the precharge out over 15 ns to avoid a huge current spike during precharge. The bit-line precharge approach is shown in Figure 6.33. The lower left hand part of the figure shows that each pair of bit lines is precharged through p-channel transistors enabled by  $\overline{\phi 1}$ . The lower right hand part of the diagram shows the RC model for one bit line. The bit lines would precharge too slowly if  $\overline{\phi 1}$  was distributed in polysilicon. The current spike during precharge would be too great if  $\overline{\phi 1}$  was distributed in metal. Therefore, the top figure shows a metal strap (the top line) tapping a polysilicon conductor (the series of resistors) to spread the precharge current spike, yet still complete the bit-line precharge during  $\overline{\phi 1}$ .

### 6.3.4 Sense Amplifiers

The schematic for the sense amplifiers is shown in Figure 6.34. The sense amplifier inputs are precharged and equalized on  $\phi 1$ , and the internal nodes are pre-discharged. All transistors in the sense amplifier are turned off as the bit lines begin to change. When one input or the other drops by  $V_{th}$ , the p-channel transistor at the top of the sense amp begins to source current to the midpoint of the other side of the sense amp. This turns on the pull-down transistor so that the midpoint of the leg opposite the dropping input is held low. If a zero is being read (i.e.,  $in$  went low),

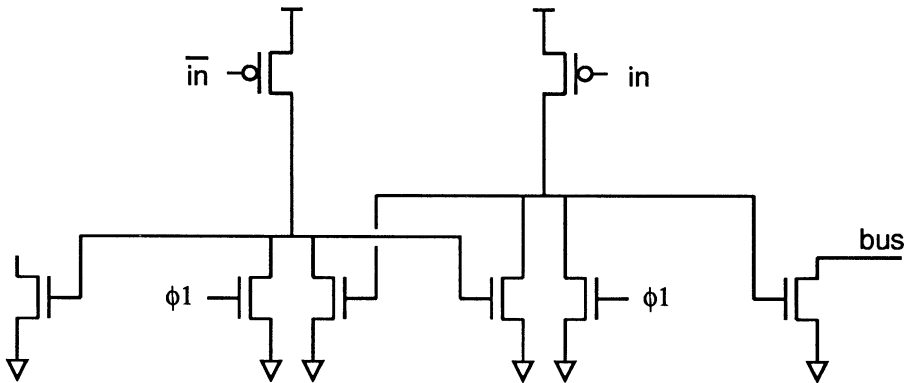


Figure 6.34: The sense amplifiers.

the large n-channel transistor discharges the cache bus. If a one is read (i.e.,  $\overline{in}$  went low), the bus remains high with the other large n-channel transistor providing a balancing load. The delay from the bit line to bit decode to sense amplifier to bus is 7-8 ns. The size of the sense amplifier with write drivers is determined by the width of the 32-bit metal 2 ICache bus that runs over the middle of the RAM, and the height of four RAM cells. The sense amplifier, write drivers, and bit-line multiplexers make up a cell that is 117  $\mu\text{m}$  tall by 501.5  $\mu\text{m}$  wide.

### 6.3.5 Read/Write Circuits

The schematic for the sense amplifiers with associated precharge circuitry and write drivers is shown in Figure 6.35. The write circuitry consists of two inverters, but because each cell is written by pulling one of the bit lines low, the p-channel devices in the inverters do not need enough strength to pull the other line high, only enough to hold the precharged line high and restore any degradation.

### 6.3.6 Tag Drivers

This RAM has two distinct types of address inputs: the six address lines, and the eight fully decoded tag lines. The tag lines are used to select one of eight bit line pairs before they go into the sense amplifier. There are two 8-to-1 multiplexers, one for the eight bit lines and one for the eight bitbar lines. The multiplexers are physically implemented with four bits on each side of the sense amplifier, since four bits are in the Left Array,

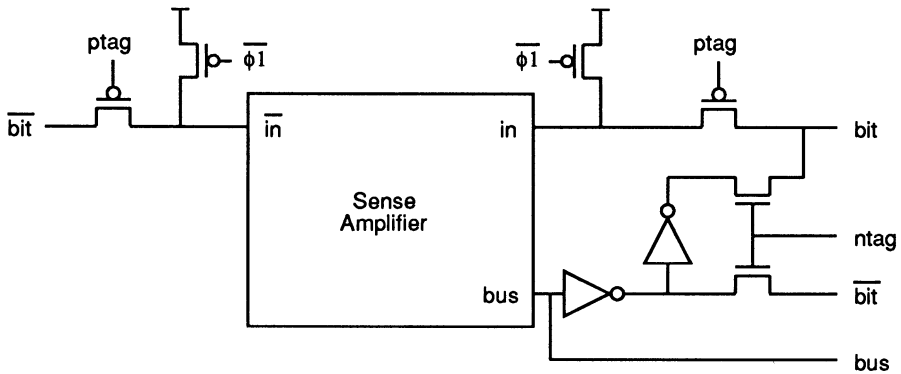


Figure 6.35: Sense amplifiers and write drivers.

and four bits are in the Right Array. Figure 6.32a shows the multiplexers for the four bits in the Left Array. The multiplexers have p-channel pass transistors for sensing the value in a cell during a read cycle and have n-channel pass transistors for driving a bit line low during a write cycle. The tag line inputs are active high, are inverted to drive the p-channel transistors (ptag in Figure 6.35), and qualified with the write signal and  $\overline{\phi 1}$  to drive the n-channel transistors (ntag in Figure 6.35).

### 6.3.7 Address Decoding

The block diagram for address decoding is shown in Figure 6.36. The six address lines enter the flow-through latches, which latch the address on  $\phi 2$  and generate both true and complemented address values for the predecoders. The purpose of the predecoders is to increase the amount of information on each line entering the decoders without increasing the number of signal lines. The twelve predecoded signals are then decoded to raise one of 64 word lines. The schematics for the flow-through latches and predecoders are shown in Figures 6.37 and 6.38. Each predecoded signal is a function of two address lines, resulting in three groups of four predecoded signals. For example, one group of four predecoded signals is  $(a_0 \cdot a_1)$ ,  $(a_0 \cdot \overline{a_1})$ ,  $(\overline{a_0} \cdot a_1)$ , and  $(\overline{a_0} \cdot \overline{a_1})$ . The decoder and word-line driver are shown in Figure 6.39. Only three of the twelve lines leaving the predecoder are selected low for one address; therefore, the decoder needs only three inputs.

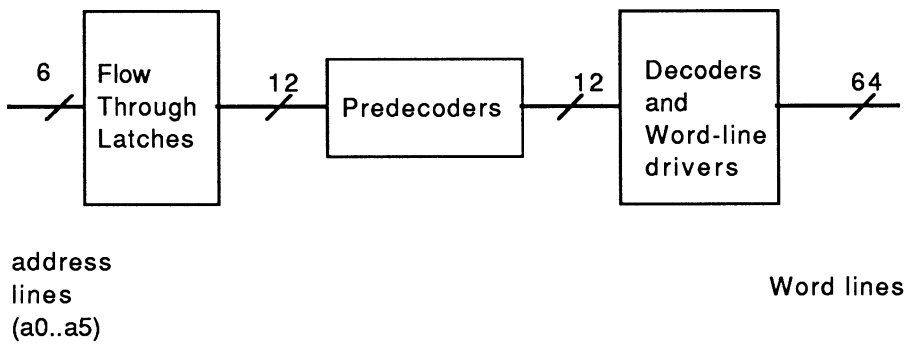


Figure 6.36: Address decoding block diagram.

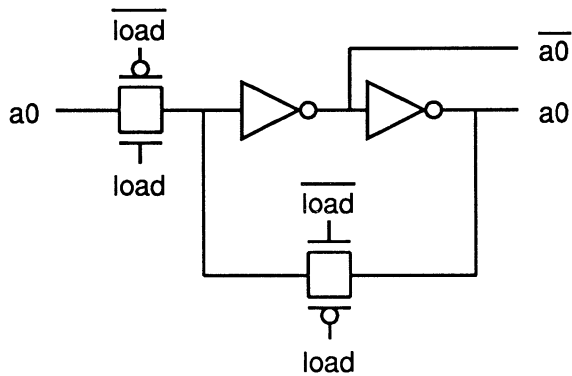


Figure 6.37: Flow-through latch in address decode circuitry.

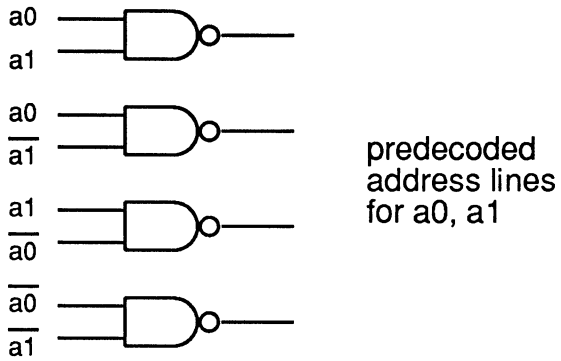


Figure 6.38: Predecode stage in address decode circuitry.

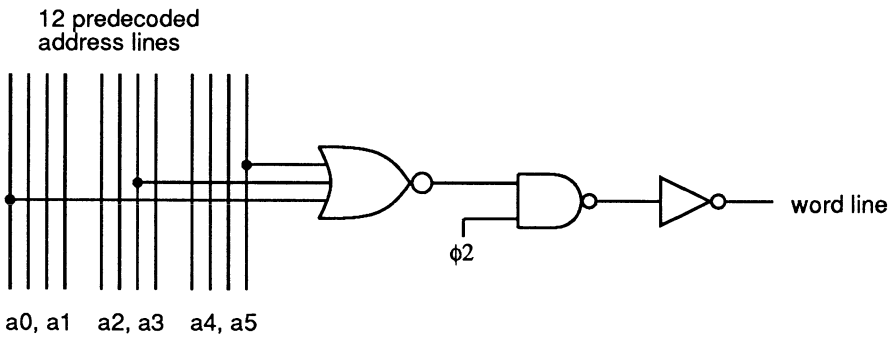


Figure 6.39: Final address decode stage and word-line driver.

## 6.4 Instruction Register

### 6.4.1 Overview

The Instruction Register (IReg) stores the instruction coming out of the ICache, does a preliminary decoding of this instruction, delays, and finally sends parts of the instruction to other parts of the machine. During an ICache miss the IReg writes new instructions into the ICache. The IReg is 32 bits wide and is located near the center of the chip below the ICache and above the Execute Unit. The IReg is 2180  $\mu\text{m}$  wide and 703  $\mu\text{m}$  high.

The main parts of the IReg are:

**Register Array:** Holds parts of the instruction in a shift register until needed.

**Decoder Array:** Does initial decoding of the operand and type fields of the instruction.

**ICache Data Bus Interface :** Handles all communication with the ICache.

**No Destination Value Logic:** Determines when the result of an instruction should not be written back into the Register File.

### 6.4.2 Register Array

The Register Array is implemented as a shift register of either two or three stages, depending on the position of each bit within the instruction. The first two latches hold the instruction during  $\phi_1$  and  $\phi_2$  of Register Fetch (RF), while the third stage (where present) holds certain parts of the instruction during  $\phi_1$  of ALU. The latches are clocked by  $\phi_2$ ,  $\psi_1$ , and  $\phi_2$ , respectively.

There are five sections in the Array: the Type and Op fields (bits 0-1 and 2-4, respectively), the source 1 address (bits 5-9), the source 2 address (bits 10,12,14,16,18), the destination address (bits 11,13,15,17,19), and the immediate field (bits 20-31). These bit positions are identical to the MemBus bit positions except for the source 2 and destination fields, which are interdigitated to facilitate multiplexing.

The Register Array is three stages deep in all sections except the source 1 and source 2 fields; the machine is done with these sections of the instruction at the end of RF. Included in the source 2/destination



section is a multiplexer used to put the contents of the source 2 field on the destination bus for instructions with only one operand.

The Register Array is located in the top center of the IReg, just below the ICache Interface. It is 1768  $\mu\text{m}$  wide and 106  $\mu\text{m}$  high. The cells are laid out on a 55  $\mu\text{m}$  pitch. Clock lines run horizontally, while inputs, outputs, and power run vertically. Schematics are shown in Figures 6.40 and 6.41. The array gets its inputs from the cache interface at the top and sends its outputs to the source 1, source 2, destination, and Immediate buses, and to the Weinberger array.

### 6.4.3 Decoder Array

The decoder array is a Weinberger structure with 11 logical inputs and 29 outputs. All gates are AND structures except for Store.s1a, which is OR-AND. A  $\psi_1$  latch is present on those signals required to be stable on  $\phi_2$ . Some of the outputs are duplicates of one another added for extra drive and ease of wiring. The layout of two typical decoders is shown in Figure 6.42.

The array, located at the bottom of the IReg, is 1380  $\mu\text{m}$  wide and 220  $\mu\text{m}$  high. It is split into two sections by the Immediate Bus. The inputs run horizontally in polysilicon strapped with metal 2. There are 6  $\mu\text{m}$  wide transistors arranged vertically to form the gates. The  $\psi_1$  latch and output inverter are located below the input lines. Power is supplied by the main  $V_{dd}$  and  $G_{nd}$  lines immediately above the array. One of the inputs, ComputeImmed, is formed within the array. The connection between the output inverter and this input can be seen at the bottom of the left part of the structure.

#### Inputs

There are eleven logical inputs to the Decoder Array plus  $\psi_1$  and  $V_{bias}$  running horizontally. Table 6.4 shows the inputs in order from the top.

#### Outputs

Outputs from the Decoder Array are nominally static  $\phi_1$  because their inputs are static  $\phi_1$ . Any signals designated static two (s2) contain an output latch. Table 6.5 lists the outputs in the order they occur in the decoder, from left to right.

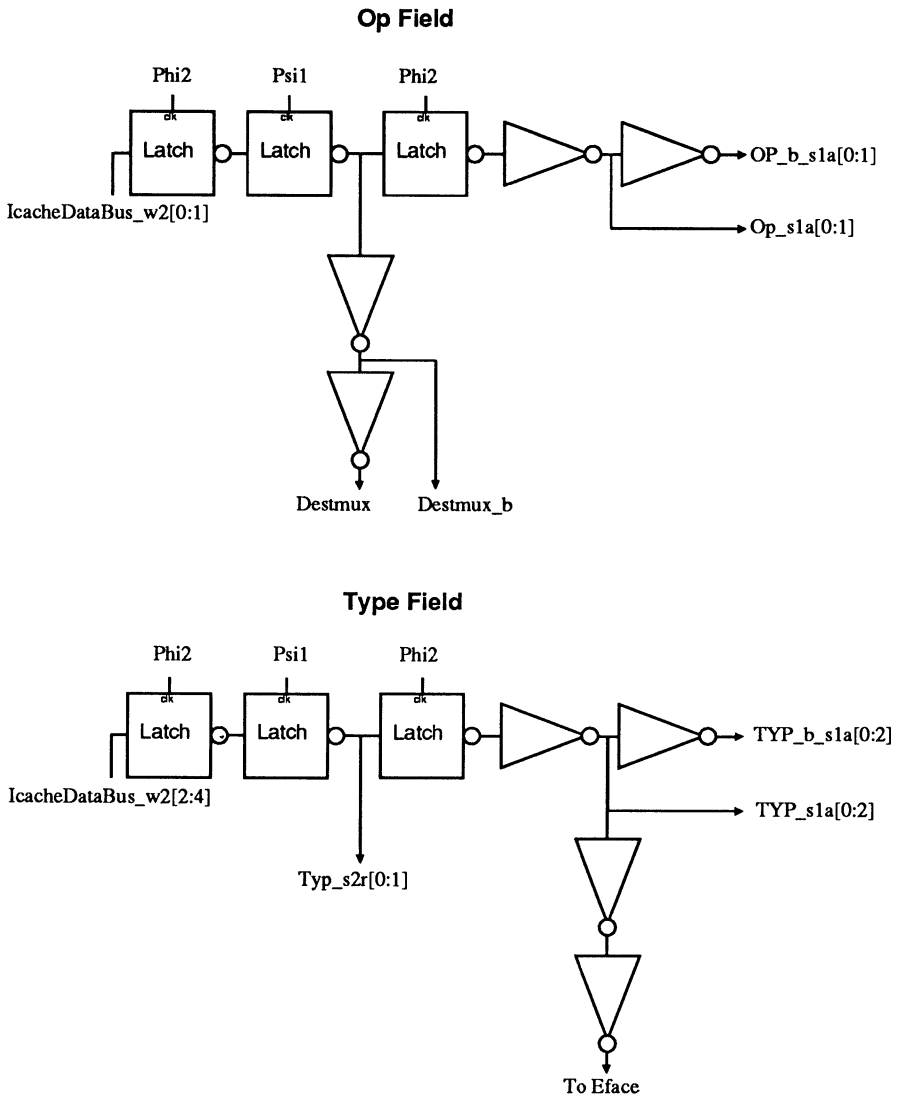


Figure 6.40: Op and Type fields.

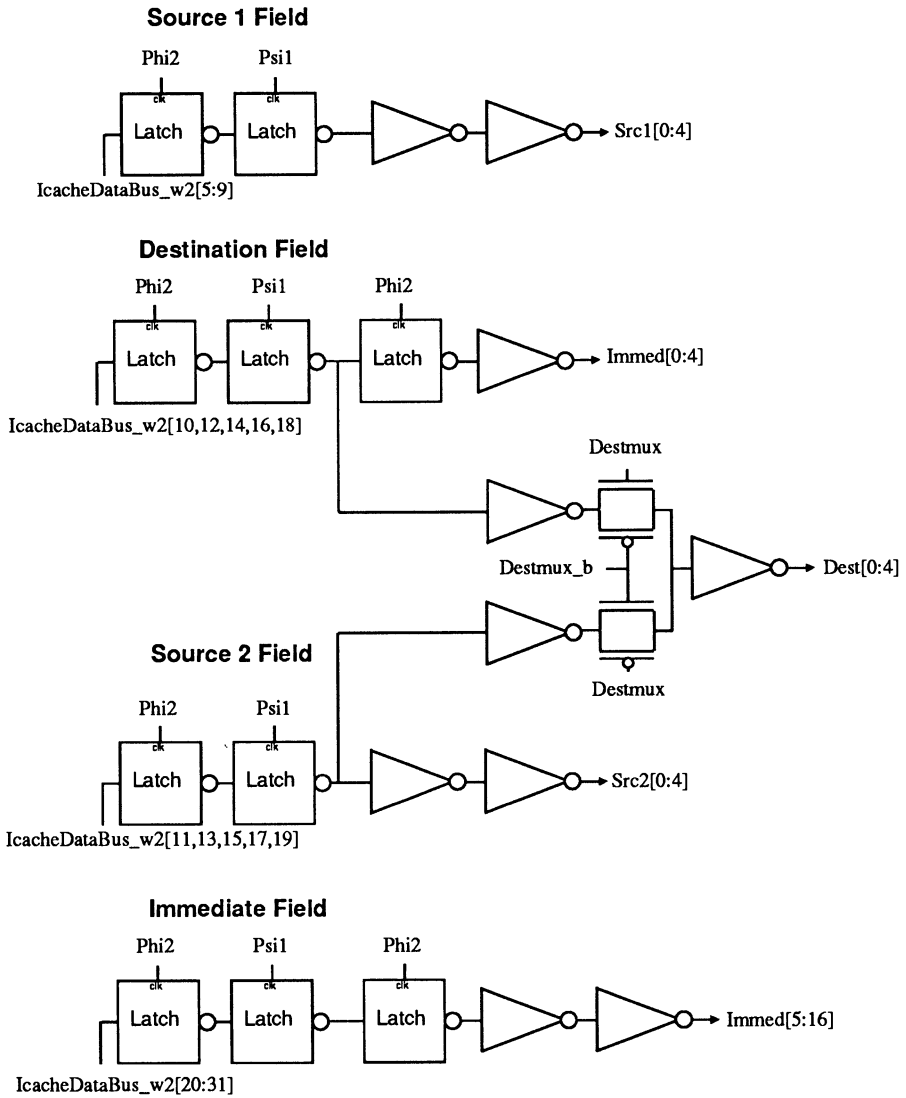


Figure 6.41: Source, Destination, and Immediate fields.



<u>Op.2</u>
Op.2
<u>Op.1</u>
Op.1
<u>Op.0</u>
Op.0
<u>Typ.1</u>
Typ.1
<u>Typ.0</u>
Typ.0
ComputeImmed
Psi1
Vbias

Table 6.4: Decoder Array inputs.

#### 6.4.4 Instruction Cache Data Bus Interface

This interface connects the cache bus (ICacheDataBus\_w2) with the external memory bus (MemBus\_w2). Its function varies with the instruction being performed and the phase of the clock. During cache writes (i.e. after a cache miss), the value from the Memory Bus is driven onto the cache bus and into the register array. During  $\phi_1$  of cache reads, the interface precharges the cache bus. While in test mode, the interface couples the cache and memory buses via pass gates.

The interface is located along the top of the IReg. It is 95  $\mu\text{m}$  high and 118  $\mu\text{m}$  wide with a cell pitch of 55  $\mu\text{m}$ . MemBus\_w2 connects in polysilicon, ICacheDataBus\_w2 in metal 2, and the register array in diffusion. Clocks and control signals run horizontally in metal 1. A  $\phi_2$  qualified version of write used in the cache is produced to the left of the array. The interface schematic is shown in Figure 6.43, and the driver layout in Figure 6.44.

#### 6.4.5 No Destination Value Logic

The No Destination Value Logic detects when an instruction does not produce a value to be written into the Register File. This occurs when the instruction is a branch, a store, a load to the floating-point coprocessor, or the destination field of the instruction is zero. The logic generates the signal NoDestVal, which is sent to the bypass control section of the Register File.

Signal	Logic	Destination
Store_s1a	$((Op2) + (\overline{Op0}))(Op1)(\overline{Typ1})(Typ0)$	Regfile
ShiftDrvResult_s1a	$(Op2)(Typ1)(\overline{Typ0})$	Execute
ShiftDrvResult_s2a	$(Op2)(Typ1)(\overline{Typ0})$	Execute
MDInst_s2a	$(Op2)(\overline{Op1})(\overline{Op0})(Typ1)(\overline{Typ0})$	Execute
Movfrs_s2a	$(Op2)(\overline{Op1})(\overline{Op0})(Cmpl)$	Execute
MDInst3_s1a	$(Op2)(Op1)(Op0)(Typ1)(\overline{Typ0})$	Execute
Typ1Op0_s1a	$(Op0)(Typ1)$	Execute
MDInst_s2a	$(Op2)(\overline{Op1})(\overline{Op0})(Typ1)(\overline{Typ0})$	Execute
Compute_s1a	$(Typ1)(\overline{Typ0})$	Execute
Branch_s1a	$(\overline{Typ1})(Typ0)$	Execute,Regfile
Memory_s1a	$(Typ1)(Typ0)$	Eface,Regfile
Movtos_s1a	$(Op2)(Op1)(\overline{Op0})(Cmpl)$	Execute
ComputeImmed_s1a	$(Typ1)(Typ0)$	Execute
TBHighChk_s1a	$(Op2)(Op1)$	Execute
TBGrChk_s1a	$(Op2)(Op1)$	Execute
TBWhich_s1a	$(Op0)$	Execute
TBEqchk_s1a	$(Op2)(\overline{Op1})$	Execute
Movfrs2_s2a	$(Op2)(Op1)(\overline{Op0})(Cmpl)$	Execute
Movtos_s2a	$(Op2)(Op1)(\overline{Op0})(Cmpl)$	Execute
Jpcrs2_s2a	$(Op2)(Op1)(Op0)(Cmpl)$	Execute
TrapInst_s2a	$(Op2)(Op1)(Op0)(Cmpl)$	Execute
Immed[0..16]_s1a		Execute,PC
TrapInst_s1a	$(\overline{Op2})(Op1)(Op0)(Cmpl)$	PC
TrapInst_s2a	$(Op2)(Op1)(Op0)(Cmpl)$	PC
Branch2_s2	$(\overline{Typ1})(\overline{Typ0})$	PC
Jpcrs_s2a	$(Op2)(Op1)(Op0)(Cmpl)$	PC
Jpc_s2a	$(Op2)(\overline{Op1})(\overline{Op0})(Cmpl)$	PC
Jspci_s2a	$(Op2)(Op1)(\overline{Op0})(Cmpl)$	PC
Movfrs3_s2a	$(Op2)(Op1)(\overline{Op0})(Cmpl)$	PC
Movtos2_s2a	$(Op2)(Op1)(Op0)(Cmpl)$	PC

Table 6.5: Decoder Array outputs.

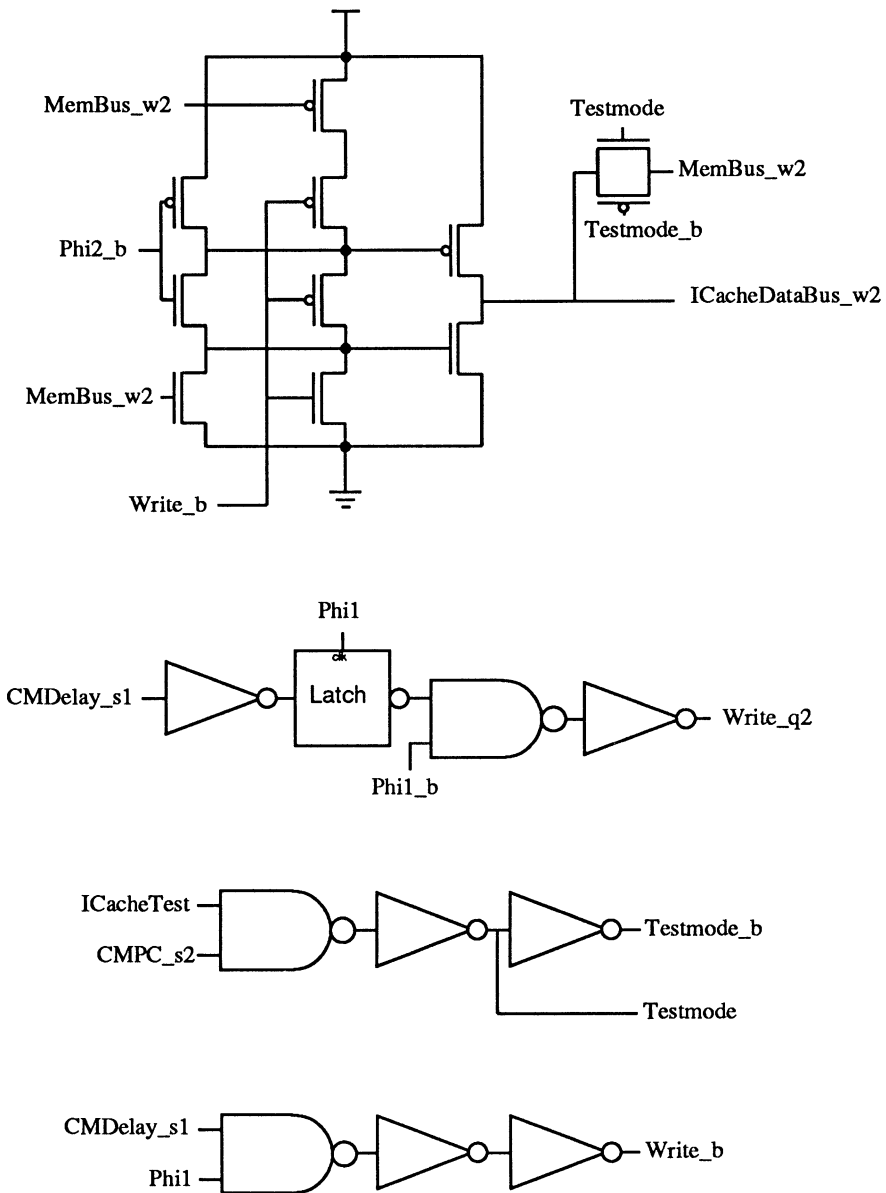


Figure 6.43: Instruction Cache interface.

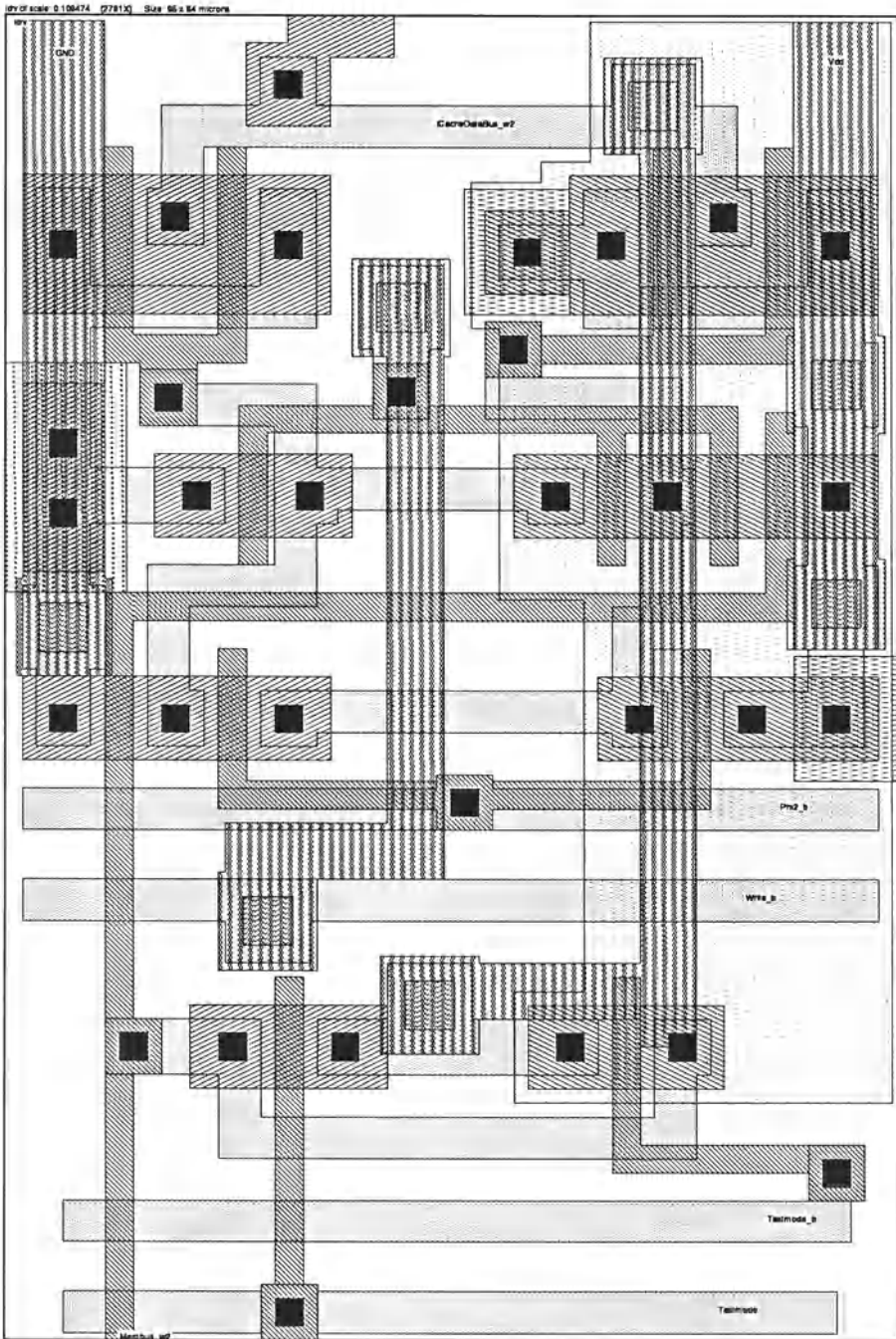


Figure 6.44: ICache interface driver.



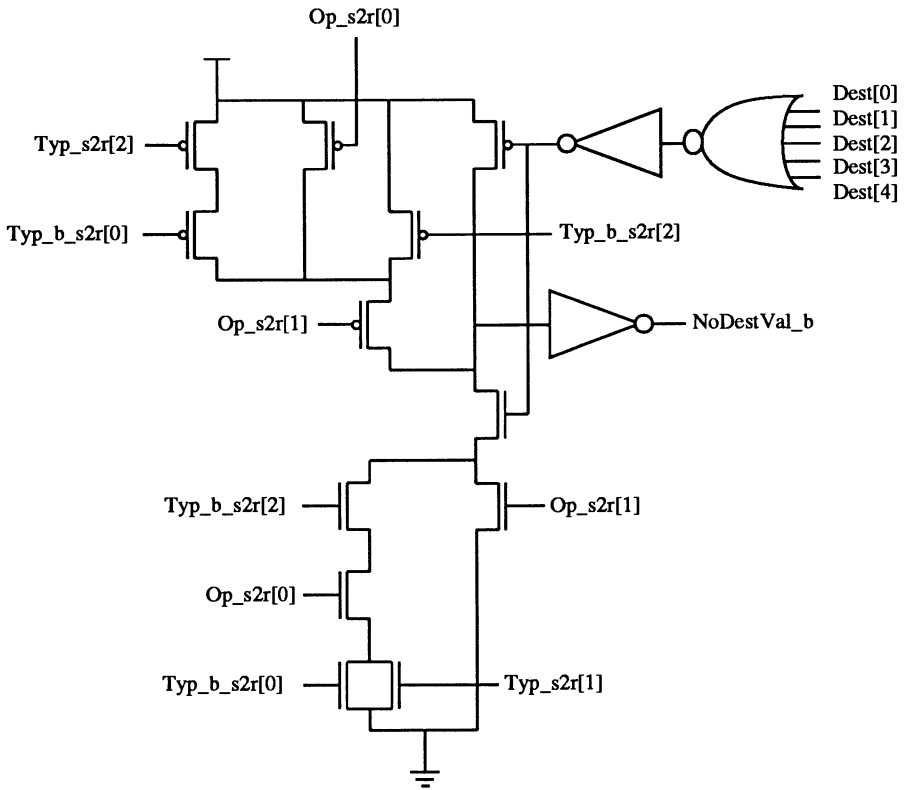


Figure 6.45: No Destination Value schematic.

This section is located along the left edge of the IReg. It is  $450\ \mu\text{m}$  high and  $120\ \mu\text{m}$  wide. The schematic is shown in Figure 6.45.

# Chapter 7

## The External Interface

The connection of the core of the processor to the outside world is done by the External Interface. It is important that this interface minimizes the overhead required for the processor to do a transaction with the outside world. The goal is to allow the implementation of straightforward external circuits to connect the processor to the external cache, coprocessors and main system bus.

This chapter begins by describing the MIPS-X external interface from the point of view of the outside world. The timing of the pins and their functions are given. The behaviour of the processor is described for memory accesses and all of the exceptional conditions. The chapter then examines the circuitry used to implement this interface.

### 7.1 The System Builder's View

This section gives the information that is needed to put the MIPS-X processor into a computing system. To relax the constraints on some of the critical paths in the memory system a late miss detect and a delayed write-back are used. To speed up floating-point load and store operations, a special floating-point coprocessor interface allows direct transfer of data between the floating-point coprocessor and the external cache. All coprocessor instructions are transmitted over the Address Bus instead of using an extra set of pins.

Each MIPS-X processor may have several coprocessors. The number of coprocessors is not limited by the hardware, but by the number of bits in the coprocessor instruction that are used to specify the coproces-

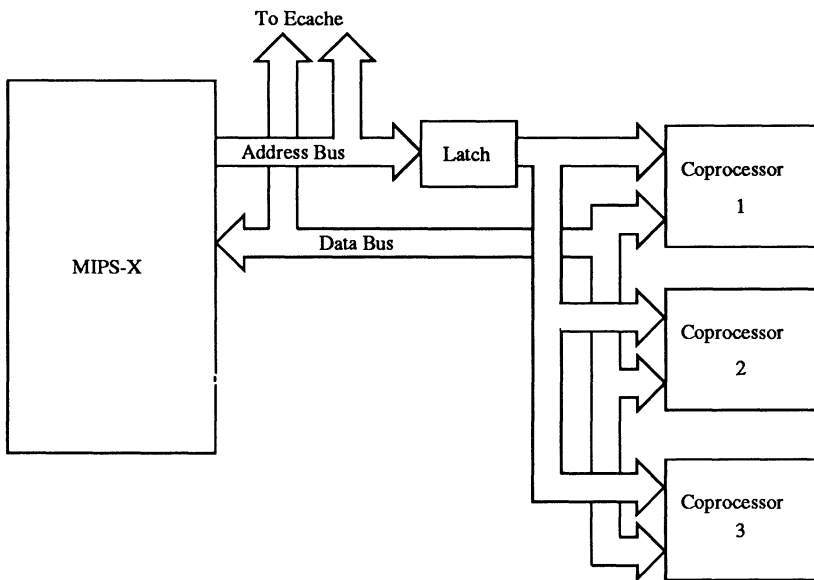


Figure 7.1: MIPS-X and coprocessors.

processor number. A proposed format in the Programmer's Manual[8] shows three bits for up to eight coprocessors. Figure 7.1 shows the MIPS-X and coprocessor interconnection. It includes a MIPS-X main processor and three coprocessors. One coprocessor is special because load and store operations to its registers can take place without going through the MIPS-X registers. This coprocessor is intended to be the floating-point unit.

Accessing the Ecache is one of the critical paths of the system. To reduce the Ecache access time a direct-mapped virtual cache is proposed. A direct-mapped cache does not have the delay of an associative compare and using a virtual cache means that a large cache that does not require time for address translation can be built. Given current static RAM technology, a 16K-word to 64K-word direct-mapped cache is feasible. The proposed cache has a 4-word block size, and the entire block is fetched from main memory on a cache miss. The cache uses a write-back scheme to reduce traffic on the main memory bus. Table 7.1 presents some performance figures for the Ecache using simulations against realistic operating system and multiprogramming workloads.

Cache Size	Miss Rate
16 KWords	2.94%
32 KWords	2.25%
64 KWords	1.66%

Table 7.1: Miss rates in the external cache.

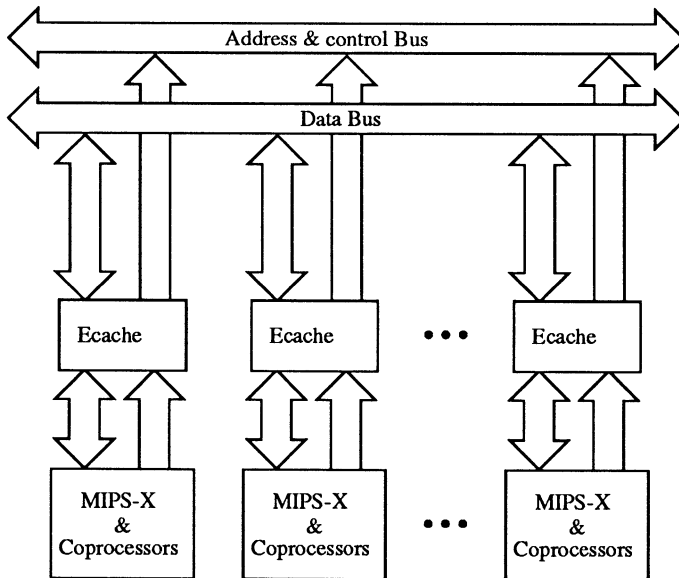


Figure 7.2: MIPS-X systems on a bus.

Figure 7.2 shows three MIPS-X processors, their associated caches and how they could be connected into a shared-memory multiprocessor system with a single bus. The memory requests that are not satisfied by the external cache go to main memory. Important issues such as keeping the caches consistent in the shared-memory environment, I/O, and virtual to physical address translation are not discussed here.

### 7.1.1 Signal Description

This section describes the functions of the pins on MIPS-X. A functional diagram of the pins is shown in Figure 7.3. A high signal represents a logic '1' level or active signal. Logic '0' is represented by a low signal.

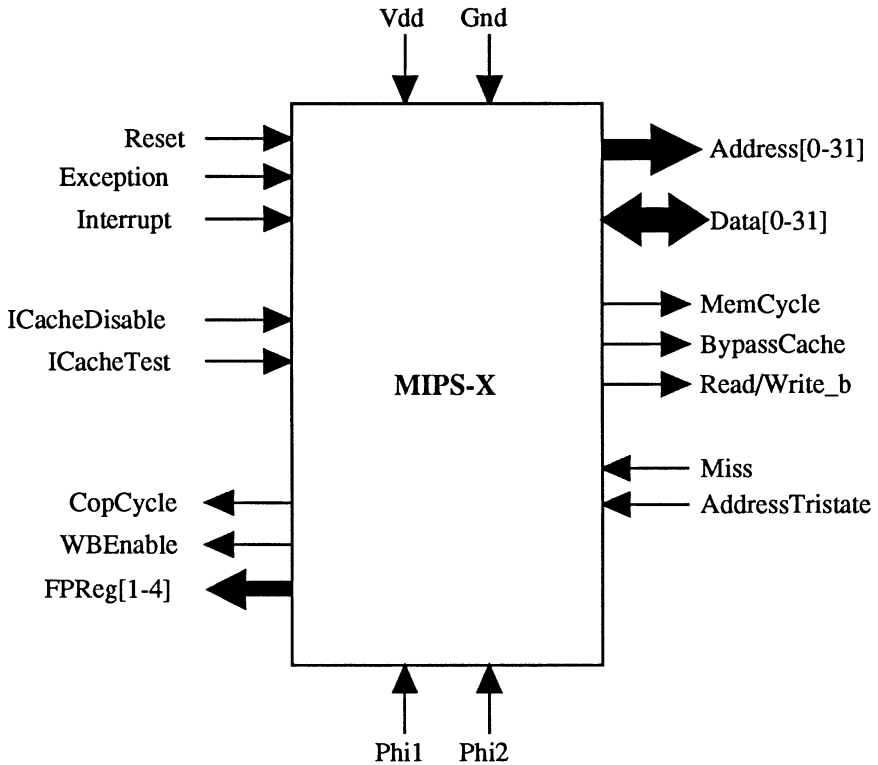


Figure 7.3: Functional signal diagram for MIPS-X.

An active low signal name has ‘\_b’ appended to it. Appendix D gives the pin numbers for the 144 pin PGA for the Revision 1 and Revision 2 parts. Appendix E describes the differences between the Revision 1 part and the Revision 2 part.

### Power and Clock Signals

*Vdd* 5 volt power supply (12 pins).

*Gnd* Ground (12 pins).

$\phi 1$  Input. Phase 1 of a non-overlapping clock.

$\phi 2$  Input. Phase 2 of a non-overlapping clock.

## System Buses

*Address<sub>0</sub>-Address<sub>31</sub>* Three-state outputs. 32-bit Address Bus with byte addressing. The two LSBs of the address (*Address<sub>30</sub>* and *Address<sub>31</sub>*) are ignored by the memory system since the memory system access is word aligned. However, these two bits are needed for coprocessor instructions.

*Data<sub>0</sub>-Data<sub>31</sub>* Input/Output. 32-bit Data bus.

## Memory Control

*MemCycle* Output. Indicates that the current cycle is a valid memory cycle. If this signal is not active, the memory subsystem should ignore the address and data lines during the cycle.

*Read/Write\_b* Output. Defines the direction of a data bus transfer. A high signal means that a data bus read is occurring.

*BypassCache* Output. The memory transaction should not use the external cache. This pin is active during the execution of the *ldt* (load-through) and *stt* (store-through) instructions.

*AddressTristate* Input. Tristate the Address Bus. This signal connects directly to the pads and is independent of the clocks.

*Miss* Input. This signal is used to stall the processor. See Section 7.1.5.

## Exception Control

*Reset* Input. Resets the processor. See Section 7.1.10 for more details.

*Interrupt* Input. Maskable interrupt.

*Exception* Input. Non-maskable interrupt.

## Coprocessor Control

*CopCycle* Output. Indicates that the current MEM cycle is a coprocessor cycle. If *MemCycle* is active then this is an FP load or store, otherwise it is one of the other coprocessor instructions.

**WBEnable** Output. *Write-Back Enable* indicates to the coprocessors that a write-back (or change in coprocessor internal state) of the last coprocessor instruction issued may be performed. See Section 7.1.7 for more details.

**FPReg<sub>1</sub>–FPReg<sub>4</sub>** Outputs. 4-bit floating-point register number. Indicates which FPU register is to be used for data transfers between FPU registers and memory during *ldf* and *stf* instructions. An FPU load or store cycle occurs when both *CopCycle* and *MemCycle* are active.

## Other

**ICacheDisable** Input. Disable the on-chip ICache causing the processor to execute cache-miss cycles.

**ICacheTest** Input. Used to read and write the instruction cache directly. See Section 7.1.11 for more details.

**Vbias** Input. This should normally be left open. There is an internal Vbias generator that connects to all of the latches to make them static. When this pin is grounded, the weak feedback is turned off and the latches become dynamic. This pin can also be driven to set a different bias level than the value generated internally.

### 7.1.2 Signal Timings

Table 7.2 gives the timing of the signals on each of the pins. In Table 7.2, a *stable* signal means that it is at the correct value for the entire phase. A *valid* signal means that the signal may change during the phase but it can be latched at the end of the phase. When an interrupt is acknowledged, further interrupts are masked by the processor. The *Interrupt* pin must be released before the processor unmask interrupts in the PSW or the processor will begin another interrupt sequence. This pin is latched by the processor whenever  $\phi 1$  falls. When *Miss* is high, the *Interrupt* pin must be kept high until after both *Miss* and  $\phi 1$  fall.

The *Exception* pin must be high for a single  $\phi 2$  or it can rise monotonically with  $\phi 2$ . In the processor, it will discharge a precharged line. The signal is not latched internally, otherwise it would take one extra cycle before the exception sequence could begin. *Miss* prevents state changes

<b>Outgoing Signals</b>	
Address <sub>0</sub> - Address <sub>31</sub>	valid $\phi 2$ of ALU, stable $\phi 1$ of MEM
Data <sub>0</sub> - Data <sub>31</sub>	valid $\phi 1$ of WB for <i>st</i> and MEM for <i>montoc</i> , stable $\phi 2$
MemCycle	valid $\phi 2$ of ALU, stable $\phi 1$ of MEM
Read/Write_b	valid $\phi 2$ of ALU, stable $\phi 1$ of MEM
BypassCache	valid $\phi 2$ of ALU, stable $\phi 1$ of MEM
CopCycle	valid $\phi 2$ of ALU, stable $\phi 1$ of MEM
WBEnable	valid $\phi 2$ of MEM, stable $\phi 1$ of WB
FPPreg <sub>0</sub> -FPPreg <sub>3</sub>	valid $\phi 2$ of ALU, stable $\phi 1$ of MEM
<b>Incoming Signals</b>	
Data <sub>0</sub> - Data <sub>31</sub>	valid $\phi 2$ of MEM
AddressTristate	asynchronous
Miss	See Section 7.1.5.
Reset	stable $\phi 2$
Exception	stable $\phi 2$ (can monotonically rise with $\phi 2$ )
Interrupt	stable $\phi 1$
ICacheDisable	stable $\phi 1$
ICacheTest	stable $\phi 1$

Table 7.2: Signal timing.

so *Exception* will only be recognized if it is high for the  $\phi 2$  preceding *Miss* falling.

### 7.1.3 MIPS-X Clocks

MIPS-X uses a 2-phase, non-overlapping clock. These two phases are called  $\phi 1$  and  $\phi 2$ . The Revision 1 parts have been tested up to 16 MHz and the Revision 2 parts should run between 16 and 20 MHz but were never fabricated.

### 7.1.4 Pad Loading

The pads have been designed to drive a 50 *pf* load.

### 7.1.5 Memory References

This section describes the MIPS-X memory interface. It is intended that an external cache (Ecache) be used to give the processor fast access to data and instructions. For multiprocessor systems, the Ecache is also needed to reduce the traffic on the main bus. Details of the Ecache will not be discussed here but it is assumed that some of the signals are generated by the Ecache. It is also assumed that the Ecache is a *write-back* cache.



The organization of the Ecache and its interface is shown in Figure 7.4. The Ecache supplies both instructions and data to the processor and handles them in the same way. The processor does not indicate whether it is doing an instruction or data reference. Instructions that miss in the on-chip ICache are retrieved from the Ecache. The controller for the Ecache is a coprocessor. It supplies the process-id (PID) on an Ecache access.

MIPS-X achieves a fast external cache interface by employing two key features that will be discussed in this section:

- a *late miss* signal (called *Miss*).
- *delayed writes* into the Ecache.

These features provide the Ecache with some extra time for the time critical operations of detecting misses and writing data back into the cache.

For load and store operations, the address is valid  $\phi_2$  of the ALU cycle. The address is latched in an external latch by the falling edge of  $\phi_2$ , and presented to the memory subsystem (the Ecache, and in its absence, memory) during the MEM cycle. This should provide the memory system with enough set-up time for the address lines. The memory control signals (*MemCycle*, *Read/Write\_b*, *Bypass Cache* and *CopCycle*) are sent out at the same time as the address.

### Miss

The *Miss* signal is used to stall the processor, particularly during memory or cache accesses. Internally, the processor does state changes on  $\phi_1$  when the *Miss* signal goes low. When the *Miss* signal is high during  $\phi_1$  the processor is stalled. *Miss* can be lowered up to 10 ns before  $\phi_1$  falls and still allow the processor to successfully complete the  $\phi_1$  clock phase.

### Late Miss

Typically, for correct operation, the processor should save the data from a load only if it has been determined that the data resulted from a hit in the cache. It is known that the critical path in the Ecache is not getting the data to and from the processor pins; it is determining whether the reference hit in the cache. The data from the cache is written into the processor registers during  $\phi_1$  of the WB cycle but to allow more time

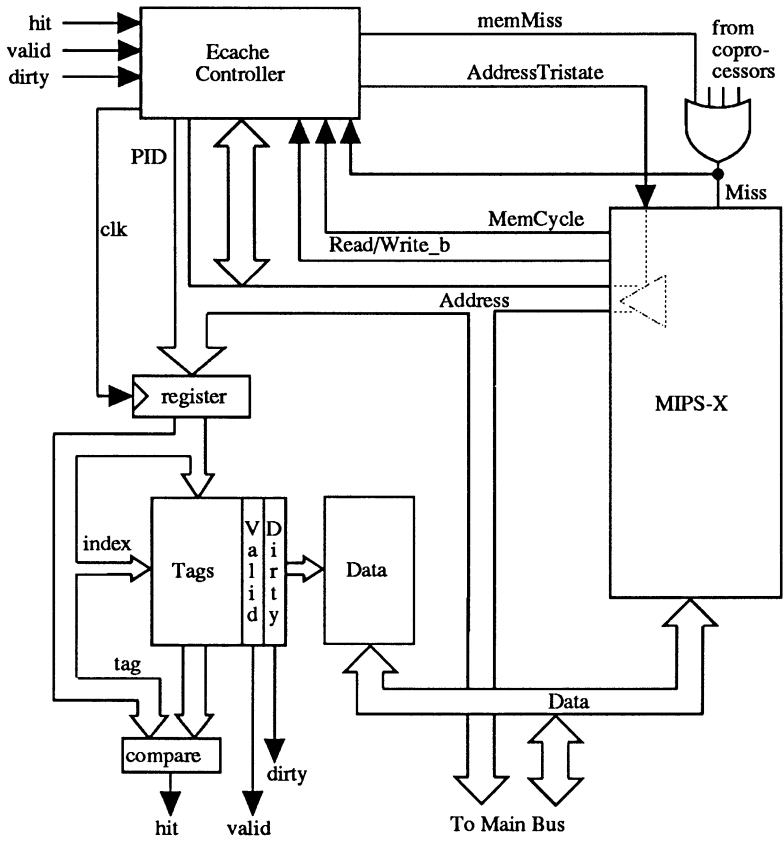


Figure 7.4: Ecache interface.

for the hit detection, MIPS-X allows the *Miss* signal to fall after  $\phi 1$  of WB rises. This means that if *Miss* does not fall, the processor has written invalid data into its register. In this case, the processor continues to re-execute the load cycle while the Ecache fetches the correct data. When the correct data is being presented to the processor, the *Miss* line is released, and the processor continues with the correct data in the register.

To make this scheme work, the compiler must ensure that a load instruction does not use the same register for both the address calculation and the destination. This will insure that all loads are idempotent (can be repeated indefinitely and give the same result).

### Memory Read

Figure 7.5 shows the basic timing for a load instruction. A normal read begins with the processor presenting the new address to the cache by the end of  $\phi 2$  of ALU of the *load* instruction. This address is latched in the external address register, and driven to the memory array. *MemCycle* and *Read/Write\_b* will be high for this cycle. The memory system must present valid data to the processor before the falling edge of MEM  $\phi 2$ . The *Miss* signal should be raised before  $\phi 1$  rising and lowered during  $\phi 1$  if the data is valid. If the memory system is not fast enough to keep up with the processor, then it must stall the processor by keeping the *Miss* signal high.

While the data is latched by the processor, the Ecache controller is performing the tag compare. The *Miss* line is pulled low during the  $\phi 1$  of WB if the reference hits in the cache. If the read generates an Ecache miss, then *Miss* is kept high and the data has to be retrieved from main memory. While the processor is stalled, the Ecache presents the address to main memory which will return the requested data to the Ecache. When the data is finally received from memory, the Ecache lowers *Miss* and the processor continues as usual.

Some complications arise in the Ecache control because of the late miss detection and the Ecache system having a two-stage pipeline. Every cycle, an address is latched externally during  $\phi 2$  falling and supplied to the Ecache. When there is an Ecache miss during the first load of a back-to-back load operation, the address of the second load will displace the address of the load that missed. To handle this event, the Ecache controller must keep the address of the first load in an internal register

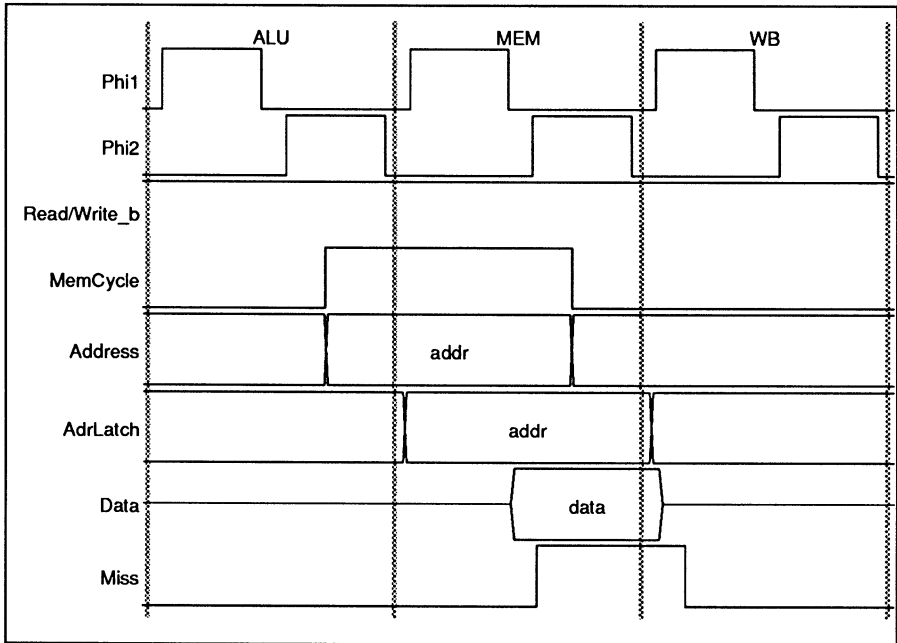


Figure 7.5: Timing diagram for a load instruction.

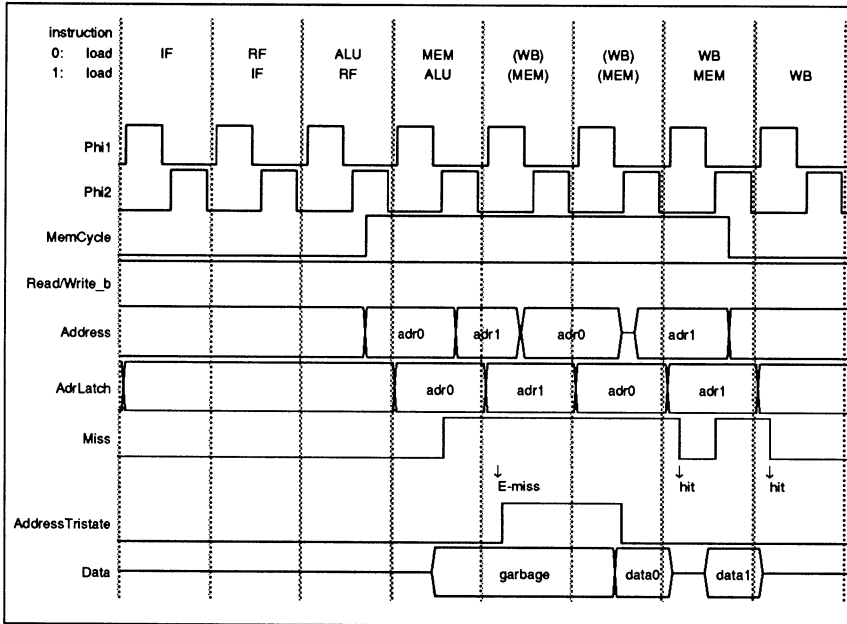


Figure 7.6: Timing diagram for Ecache miss during consecutive loads.

and then reload it in the address register. This is done by first tristating the processor address pins and then driving the address bus with the address that missed. Figure 7.6 shows this sequence.

## Memory Write

To reduce the main memory bus traffic in a multiprocessor configuration MIPS-X assumes that a write-back scheme is used for the Ecache. Under this scheme, writes are only updated in memory when either the Ecache is flushed or a block that needs to be replaced is found dirty. Therefore, before writing data into the Ecache, the *miss* and *dirty* conditions must be determined. If a *miss-and-dirty* condition is encountered, the current cache entry must first be updated in memory and only then can the data be written into the cache. This means that a store instruction needs two memory cycles: one to probe the cache to check if the block exists and one to write the data. The instruction following the store instruction should not use its MEM cycle (the WB of the store instruction) so the

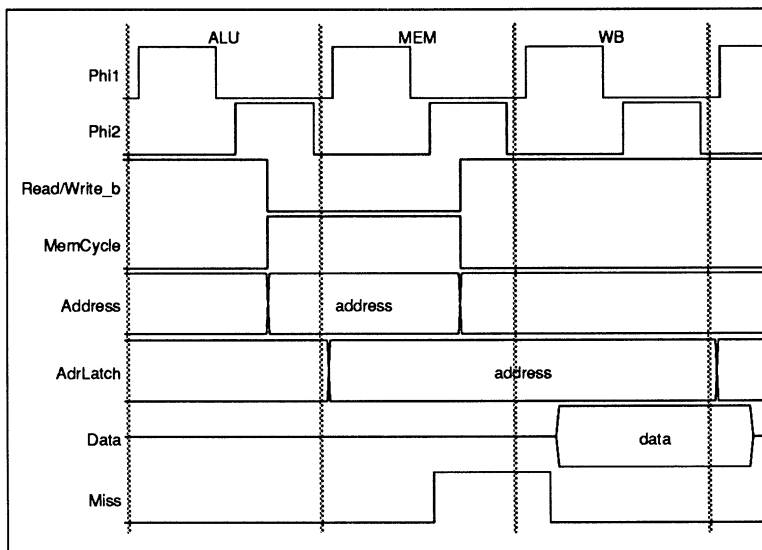


Figure 7.7: Timing diagram for a store instruction.

Ecache can use this cycle to write the data without stalling the processor. The architecture disallows stores followed by another memory or coprocessor instruction but there is no hardware interlock to prevent this. The compiler reorganizer is responsible for scheduling the right instruction here<sup>1</sup>.

Figure 7.7 shows the timing for a store instruction. The write operation begins like a read, except that the *Read/Write\_b* signal is low. Data is driven out of the processor when *Miss* falls during  $\phi_1$  of WB and remains on the bus until  $\phi_1$  rising of the next cycle. The Ecache writes the data during WB (the MEM slot of the next instruction). This scheme is known as the *delayed write*. Note that raising *Miss* before WB will not stall the write; the data will be on the bus for exactly one cycle, independent of *Miss*.

A sequence of memory instructions is shown in Figure 7.8.

<sup>1</sup>There is a hardware interlock that prevents an ICACHE miss from occurring during the second cycle of a store instruction. Section 7.1.6 explains this in more detail.



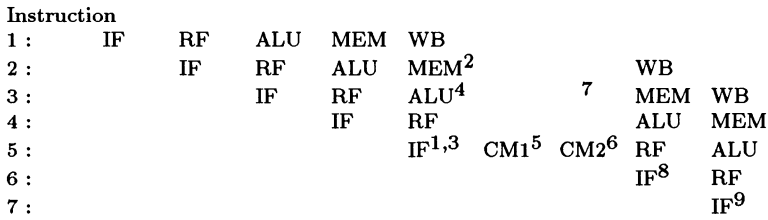


Figure 7.9: ICache miss timing.

## Bus Locking

There is no provision in the MIPS-X processor for implementing synchronization primitives such as a read-modify-write instruction. Two schemes have been proposed to accomplish synchronization. One is to use strictly software synchronization such as Dijkstra's or Dekker's algorithm. The other is to implement coprocessor instructions in the Ecache controller. A possible coprocessor instruction is one that locks the bus on a read and unlocks it on the next store instruction. This is effectively a read-modify-write instruction.

### 7.1.6 ICache Misses

The timing for an ICache miss cycle looks exactly like a load instruction to the Ecache. During an ICache miss, two instructions are fetched during two cache miss cycles. Exceptions during an ICache miss are handled in the same way as those during normal operation. Figure 7.9 describes an ICache miss sequence for each of the pipeline stages. Figure 7.10 shows the timing of the external pins for a typical instruction sequence. In Figure 7.9, assume that instruction 5 misses in the instruction cache. The following enumeration numbers correspond to the superscripts in Figure 7.9.

1. Instruction 5 misses in the ICache. The miss sequence starts from the next cycle.
2. Instruction 2 is closest to completion at this point. An instruction can commit only in its WB phase (stores are an exception) and special care needs to be taken to see that the instruction does not change processor state before the miss is handled.



- If instruction 2 is an ALU operation then the WB is postponed until after the miss sequence.
- If instruction 2 is a load, then the data from memory is latched into the input memory data register, the register that latches input data before it is written into the Register File. If the memory operation corresponding to instruction 2 fails (*Miss* is active) then the sequence of events that follow at the Ecache is the same as an Ecache miss during the first load of a load-load sequence. The timing for this sequence was shown in Figure 7.6. The Ecache controller loads the address register with the load address of instruction 2 (note that this address was displaced from the address register by the address that came from the processor) and completes the load. The *Miss* signal stays high causing the processor to stall throughout the load miss sequence.
- If instruction 2 is a store then the only hardware interlock in the processor takes effect. Recall that store instructions require two cycles to complete and the software ensures that the instruction after the store does not use memory. However, there is no way to predict an ICache miss, which also needs to access memory, so a hardware interlock is implemented that inserts an extra cycle to complete the store before the ICache miss begins. After the ICache miss cycles are completed, instruction 2 will execute a null WB cycle.

If an exception occurs during the ICache miss cycles, the store instruction should be re-executed but the store has actually completed. Before returning from an interrupt, the interrupt handler must convert the first instruction to a *nop* if it is a store instruction. This is done by changing the address for that instruction to Address 0 where there is a *nop*<sup>2</sup>. This action need not be done if it is known that all store instructions are idempotent. Writes to shared memory or device registers may have side effects that render them non-idempotent.

### 3. The address of instruction 5 is sent out.

---

<sup>2</sup>The architecture specifies that a *nop* instruction be found at Address 0 in both system and user space. This allows instructions to be *squashed* on restart by setting their PC address to zero.

4. The address of instruction 5 overrides a possible data address due to instruction 3. Essentially, the internal ICache miss signal causes the PC value to be selected instead of the address for the memory access.
5. Instruction 5 returns from the Ecache. The address of instruction 6 is sent out.
6. Instruction 5 is copied into the ICache. Instruction 6 comes back from the Ecache.
7. The address corresponding to a possible memory-reference by instruction 3 is placed on the Address Bus by MIPS-X and latched into the address latch at the Ecache.
8. Instruction 5 is decoded to begin its RF phase. Instruction 6 is copied into the ICache. It is 'fetched' at the same time.
9. The RF of instruction 6 is started. Instruction 7 is fetched from the ICache (unless another ICache miss occurs).

### 7.1.7 Coprocessors

Coprocessor instructions are cached in the main processor and hence have to be sent to the appropriate coprocessor for execution. Ideally, the coprocessor instructions should be sent over a separate bus, but because this requires a large number of pins a scheme that sends coprocessor instructions over the Address Bus is used. The coprocessor instructions are really memory type instructions with the offset field containing the instruction for the coprocessor. All loads and stores of coprocessor registers must go through the MIPS-X registers except for one special coprocessor which would usually be a floating-point unit.

There are three instructions that deal with the general coprocessor interface. They are move-to-coprocessor (*movtoc*), move-from-coprocessor (*movfrc*) and coprocessor-alu (*aluc*). The first two are used to transfer data between the coprocessor registers and MIPS-X and the last one is to initiate coprocessor operations. The *aluc* instruction is actually a *movfrc* instruction with the destination register set to register 0<sup>3</sup>.

---

<sup>3</sup>The current software system generates *movfrc* instructions but *movtoc* instructions could also be used if the coprocessor ignores the data on the data bus during this cycle. The timing diagrams in this document assume that *movtoc* instructions are used.

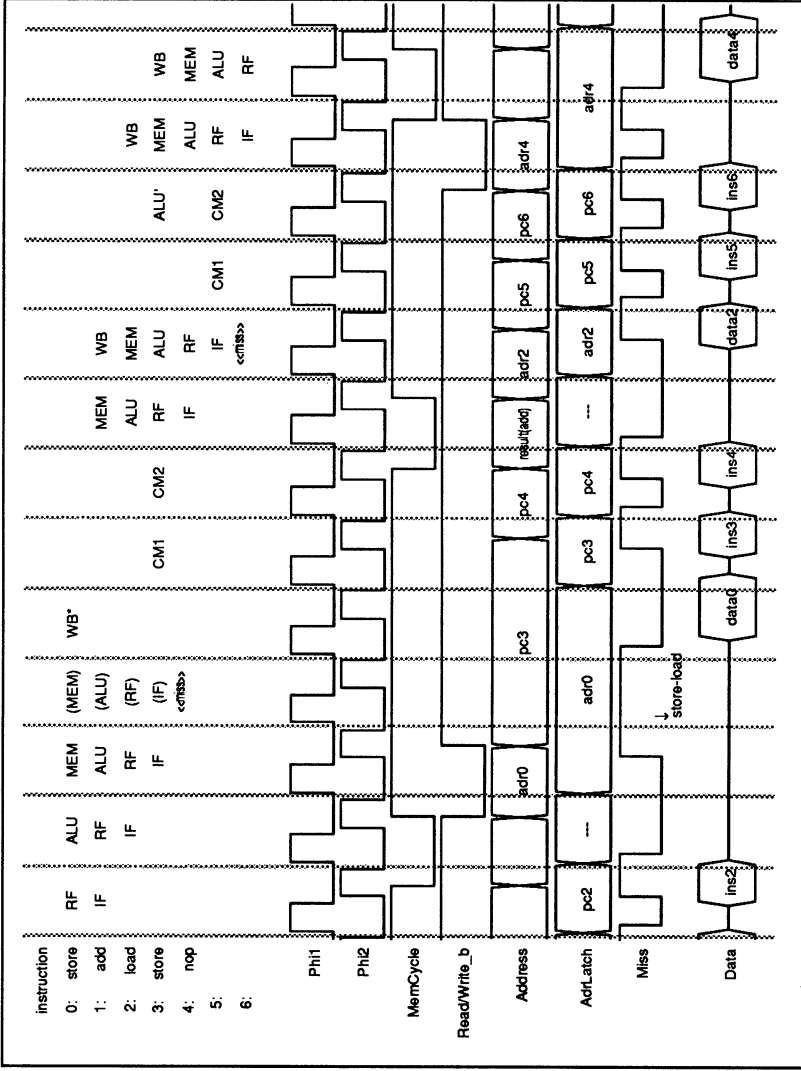


Figure 7.10: Timing diagram for an ICache miss.

The floating-point load and store instructions are described later in Section 7.1.8. Detailed descriptions of these instructions are given in the Programmer's Manual[8] and briefly in Section 2.1.

The timing of the coprocessor instructions is similar to the timing for load and store instructions. During the ALU cycle, the coprocessor instruction is driven on the Address Bus as if it were an address. This is latched by the coprocessor latch on ALU  $\phi 2$  falling. The *CopCycle* signal is asserted at the same time, notifying the coprocessors that the Address Bus contains a coprocessor instruction; the *MemCycle* signal is inactive, so the memory subsystem will ignore the Address Bus during this cycle. The addressed coprocessor will then execute the instruction as follows:

- **movtoc** - The MIPS-X source register is placed on the data bus at MEM  $\phi 1$  rising or when *Miss* falls. If necessary, the coprocessor can stall MIPS-X until it has decoded the instruction by using the *Miss* line. The data stays driven until *Miss* is released during WB  $\phi 1$ . The data is latched by the coprocessor on MEM  $\phi 2$  falling. The coprocessor should only write the destination register if the *WBEEnable* signal is activated by MIPS-X as explained later in Section 7.1.7. Note that this instruction is like a store instruction except that only one memory cycle is needed because a cache probe is not necessary.
- **movfrc** - The coprocessor places the contents of the appropriate register on the data bus before MEM  $\phi 2$  falling. If necessary, MIPS-X can be stalled if this cannot be done in time since the coprocessor only has one cycle to do this<sup>4</sup>. MIPS-X will latch the contents of the data bus on MEM  $\phi 2$  falling.
- **aluc** - The coprocessor decodes the instruction and starts executing the instruction. If *movfrc* instructions are used, then the data on the data bus is read by MIPS-X and dumped into register 0. If

---

<sup>4</sup>This will probably be the case since a 50ns clock cycle will not be enough time for the coprocessor to read the instruction, decode it, decode the register number, index into its register file, and place the contents on the data bus. An alternative solution that would have provided the coprocessor more time to decode the register number would have been to send out the register number in the RF phase on the register number lines provided for the floating-point unit, but this meant a non-trivial change in the MIPS-X processor implementation and was discarded.

*movtoc* instructions are used, then MIPS-X places register 0 on the data bus. This should be ignored by the coprocessor.

Coprocessors should stall the main processor by keeping *Miss* high if they require more time to finish an operation .

### Write-Back Enable

The external interface of MIPS-X provides for restartability of coprocessor instructions. This is necessary if the processor has to operate in a system where interrupts or other exceptions may occur. An instruction should not change processor state until it has reached the WB stage on the main processor. To tell a coprocessor that an instruction can complete, a signal called *WBEnable* is asserted by MIPS-X on the WB cycle of the coprocessor instruction. Before this signal is received, the coprocessor must retain the result of the operation internally until *WBEnable* is activated (if at all). If the coprocessor implements idempotent operations then it may ignore the *WBEnable* signal and commit as soon as it has finished execution.

Figure 7.11 shows the timing for the execution of four consecutive coprocessor instructions: *movtoc*, *movfrc*, *aluc* and *aluc*. Instruction 4 stalls MIPS-X; this is to show what happens when an instruction is sent to a coprocessor that can execute only one instruction at a time, and has not finished executing the previous instruction sent to it. Also, note that when the coprocessor stalls MIPS-X, the Ecache must also stall. Otherwise, it would read the incorrect data from the data bus. This shows that all devices interacting with MIPS-X must look at the *Miss* signal.

If the MEM cycle of a coprocessor instruction coincides with an interrupt or page fault, the *WBEnable* signal will not be generated and the instruction will be re-executed after the interrupt has been serviced. When a coprocessor is waiting for *WBEnable* to complete its current instruction and another coprocessor instruction is issued before *WBEnable*, the current instruction is squashed. The coprocessor should start the new instruction if it was the coprocessor addressed. If a coprocessor sees a *WBEnable* before or concurrent with any other coprocessor instructions issued, then it should complete its current instruction. If an ICache miss occurs during the MEM cycle of a coprocessor instruction, activation of the *WBEnable* signal is delayed until after the CM1 and CM2 cycles because the WB cycle is delayed. If no exception occurs

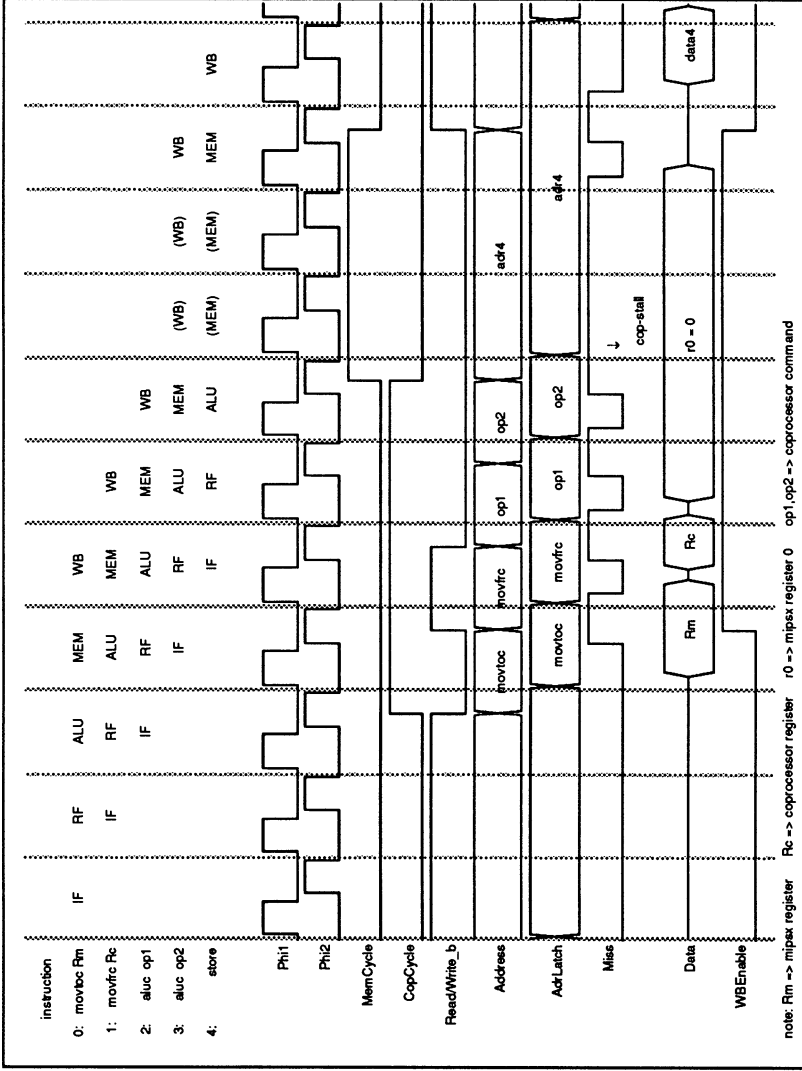


Figure 7.11: Timing diagram for coprocessor instructions.

during these two cycles then *WBEnable* will be activated. When an exception occurs during the CM cycles, *WBEnable* will not be generated and the instruction will be restarted later. Figure 7.12 shows the delaying of the *WBEnable* during an ICache miss followed by a coprocessor instruction that gets squashed due to an interrupt.

There are at least two ways the coprocessor can behave if it is sent back-to-back instructions:

- The coprocessor can be pipelined at least two stages: One for executing and one for performing the write-back. This allows the main processor to send data and instructions to the coprocessors one immediately after another as shown in Figure 7.12. It may be necessary to bypass the coprocessor Result register because the next instruction may need to use it before it has been written back.
- The coprocessor first performs the write-back and then starts executing the new instruction. In this case, the main processor is stalled until the pending write-back is performed and the coprocessor has had time to decode the new instruction.

A better option would be to resolve these interlocks by software, but this may not be possible for all coprocessors, because coprocessor instruction execution times will vary from coprocessor to coprocessor, from system to system, and even from instruction to instruction on the same coprocessor. Nevertheless, it is still worthwhile to try to schedule useful operations between coprocessor instructions when the execution times can be estimated.

The need to look at the *WBEnable* signal before completing the execution of instructions on coprocessors can complicate the coprocessor control. An alternative solution that obviates the *WBEnable* signal is to guarantee that non-idempotent instructions are not restarted on the coprocessors. Then the coprocessor instruction that is in its MEM phase when an exception occurs can be allowed to complete in the coprocessor. Such a feature can be implemented in the software interrupt handler. During the return from interrupt, the handler should check to see if the first instruction to be restarted is a non-idempotent coprocessor instruction and convert it to be the *nop* instruction at Address 0. This is similar to the scheme used to prevent stores from re-executing after an interrupt.

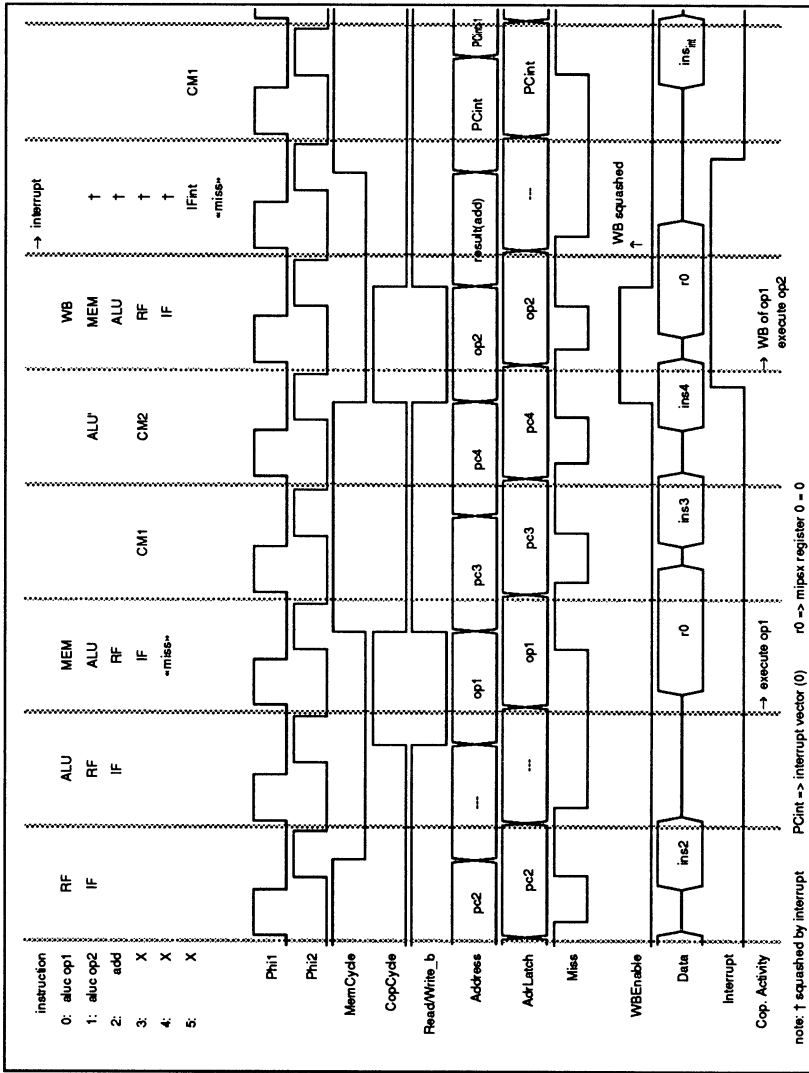


Figure 7.12: Timing diagram of the *WEnable* signal for coprocessors.



### 7.1.8 Floating-Point Unit

The special interface for the floating-point unit tries to optimize floating-point loads and stores. The fact that all loads and stores of coprocessor registers must go through the general purpose registers of the main processor was considered unsatisfactory for the floating-point unit (FPU) since it could be used heavily. The coprocessor interface was extended to allow loads and stores directly between memory and the FPU by including two more instructions for the floating-point coprocessor. Load float (*ldf*) loads a floating-point register from memory and store float (*stf*) stores the contents of a floating-point register into memory.

The two instructions require four pins dedicated to the FPU: *FPCReg1-FPCReg4*. The timing of these two instructions is identical to memory loads and stores. For a *stf*, MIPS-X performs a store cycle by generating the memory address and signaling the FPU to provide the data word. For a *ldf*, MIPS-X performs a load cycle by generating the memory address and signaling the FPU to get the data from the data bus. The FPU source or destination register for a *ldf* or *stf* operation is sent to the FPU by means of the *FPCReg1-FPCReg4* signals, whose timings are identical to the address lines.

Figure 7.13 shows the timing of the various signals required for a stream of floating-point instructions: two *ldf*'s to load the operands followed by an *aluc* (the floating-point operation) to be performed on the two operands and a *movtoc* to move the result into a MIPS-X register, followed by a *stf* to store the result into memory.

It is important to note that when a *ldf* or *stf* is being performed, it may be necessary for either the FPU or the Ecache to wait for one another. The solution adopted is to require each of these to remain in the same state until  $\phi 2$  has fallen; the required synchronization is accomplished by having the FPU and the Ecache stall MIPS-X using the *Miss* signal until they have both finished. The data should then be latched on the falling edge of  $\phi 2$ .

### 7.1.9 Exceptions

Exceptions such as interrupts and page-faults are asynchronous events. The *Interrupt* line is maskable and latched by the processor during  $\phi 1$ . The *Exception* line is not maskable and not latched. Figures 7.14 and 7.15 give the timing for the *Interrupt* and the *Exception* lines respectively. Exceptions and Interrupts can occur while the processor is being stalled

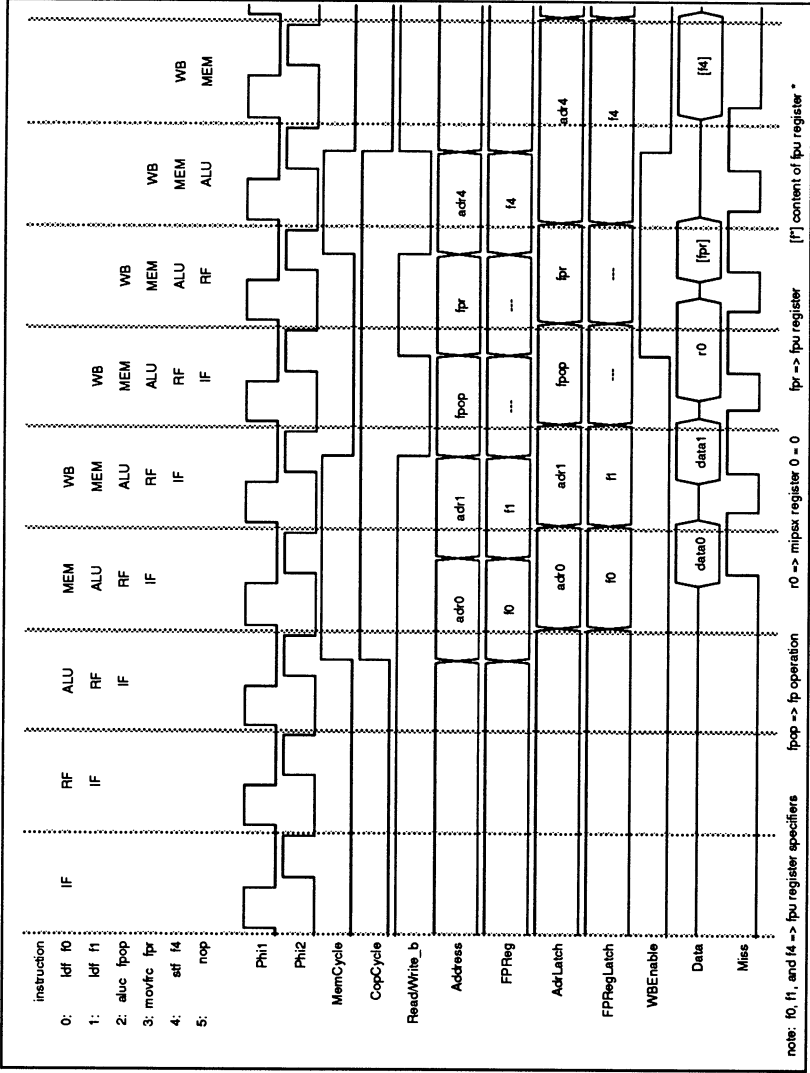


Figure 7.13: Timing diagram for floating-point operations.

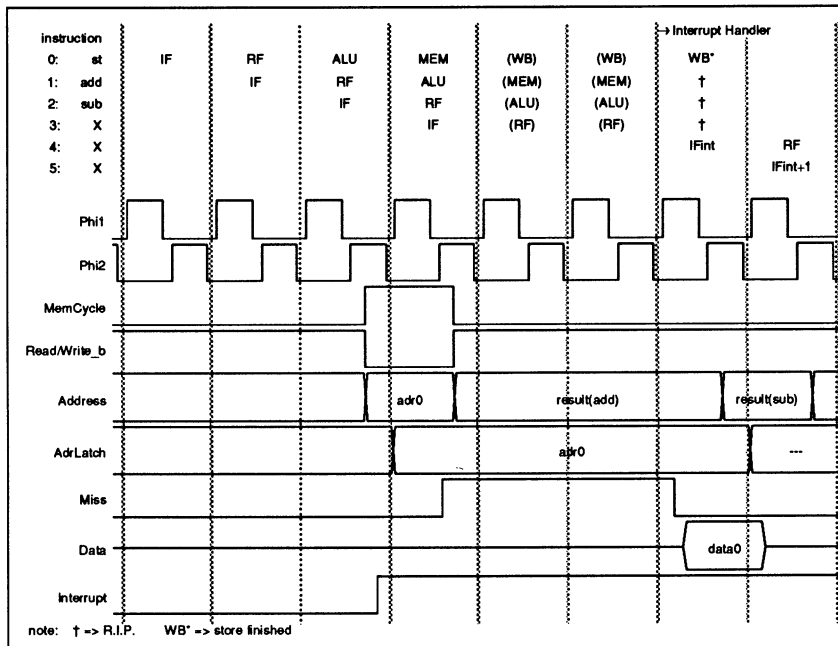


Figure 7.14: Interrupt timing.

by the *Miss* signal. When the *Miss* signal is released, execution will begin at the start of the exception handling routine. Section 7.1.2 explains the timing for these signals in more detail.

When MIPS-X recognizes an interrupt or exception, execution jumps to Address 0 in system space. At this point, the exception routine should save the state of the processor and any coprocessors that are to be used before the interrupt returns. Details of this are explained in Appendix A.

Table 7.3 shows the action taken by MIPS-X for various instructions when an interrupt or exception occurs during the ALU, MEM and WB stages. If an interrupt occurs before the WB stage of a coprocessor instruction, the processor does not activate the *WBEnable* line. The coprocessor remains in a suspended state until it receives the next coprocessor cycle request, signaling a new instruction. In this case it starts off on the new instruction, neglecting to do the write-back of the previous

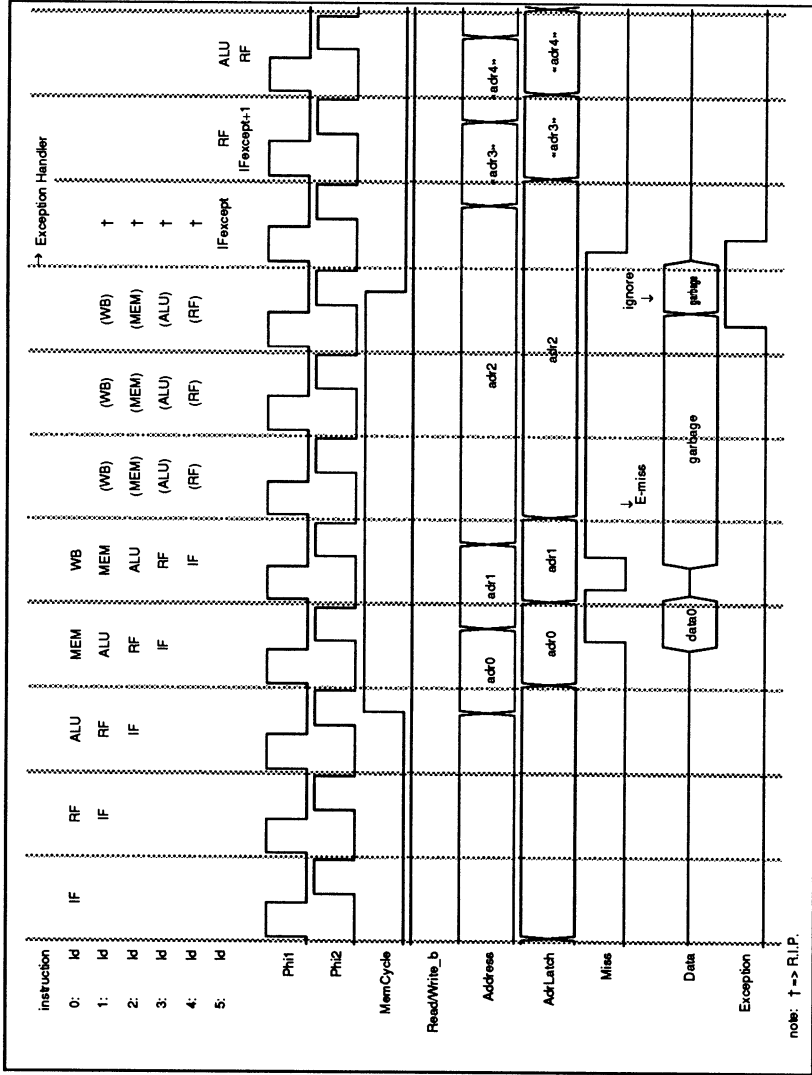


Figure 7.15: Exception timing during page fault.

	ALU	MEM	WB
Compute instruction	Squashed	Squashed	Completes
Load instruction	Squashed (never leaves chip)	Squashed	Completes
Store instruction	Squashed (never leaves chip)	Completes next cycle	Completes
COP instruction	Squashed (never leaves chip)	WBEEnable not generated	Completes if WBEEnable seen

Table 7.3: MIPS-X actions for various instruction types when an exception occurs during their ALU, MEM and WB pipe stages.

one. This effectively squashes the interrupted instruction. Interrupts which arrive after the *WBEEnable* signal has been asserted have no effect on the coprocessor.

### 7.1.10 Reset

To reset the processor, this pin should be held high for at least 4 cycles and should only be released when  $\phi 1$  is active or during the rising edge of  $\phi 1$ . Internally, Reset is expected to be a stable  $\phi 2$  signal.

During reset, the address  $0x7ffff80$ <sup>5</sup> is forced onto the address pins and the processor will begin executing at this location when *Reset* is released. This location should be the start of the reset sequence.

### 7.1.11 Instruction Cache Testing

The instruction cache can be read and written to test whether the bits in the instruction cache are working. The sequence is:

1. Reset the processor by holding the *Reset* pin high for at least 4 cycles to clear the tags. Before releasing *Reset*, set *ICacheTest* high and do 4 more cycles. Then release *Reset* as described in Section 7.1.10.
2. While *ICacheTest* is high the PC Unit is forced to increment starting from the Reset vector. Since the ICache is empty, the processor will now do ICache miss cycles to fill the ICache with the data presented at the data pins. The data should be available at the pins while the *MemCycle* pin is high but can be left driven on the pins

---

<sup>5</sup>This is the byte address that appears on the pins of the chip. Internally, the PC Unit is word addressed and it uses  $0x7ffffe0$  as its starting address. This is reflected in the simulators and the assemblers.

during this part of the test if only one pattern is to be loaded in all locations. With the PC Unit forced to be in increment mode, all locations in the cache will be filled with the data at the data pins after 1024 clock cycles. This number is not 512 cycles (the ICache can hold 512 instructions) because the processor is executing ICache miss cycles. You will see a pattern of two ICache miss cycles and two execute cycles on the Address Bus.

3. During ICacheTest mode the datapath of the processor is still interpreting the data at the data pins as instructions. This means that some care must be taken if the top two bits of the data (corresponding to the *TY* field of an instruction) are *10* meaning that the instruction is a memory instruction. If the data can be interpreted as a store instruction, the data loaded into the ICache will be corrupted because the datapath will be driving the internal data bus. If the data can be interpreted as a load instruction, then the *MemCycle* pin cannot be used to determine when to put the next data pattern on the pins unless you know when it is for an ICache miss and when it is for the load instruction.
4. After filling the ICache, Reset the processor while keeping *ICacheTest* high. This will force the PC Unit back to the Reset vector but it will not invalidate the contents of the tags. When *Reset* is released, the processor will sequence through the 512 ICache locations in 512 cycles. The data on the data pins will be data from the ICache.
5. If none of the patterns loaded into the instruction cache correspond to memory instructions, then the *PadMem* pin should always stay low because all instructions should be in the ICache and no data will be accessed. If *PadMem* does go high it is an indication of data coming from the ICache being interpreted as a memory instruction or a failure in the tags causing an ICache miss cycle.

### 7.1.12 Datapath Testing

A useful feature for testing the datapath is to observe the contents of the Address Bus. Internally, the Address Bus is the output of either the Result Bus or the PC Bus. During ICache miss cycles, the Address Bus will have the contents of the PC Bus. Otherwise, it will show the value

of the Result Bus which usually has the output of the ALU or the shifter, depending on the instruction being executed.

## 7.2 The External Interface Hardware

This section deals with the hardware used to connect the main core of the MIPS-X processor with the pins of the chip. Most of the circuits are located in the space above the Register File and beside the Instruction Cache. Immediately beside the Instruction Cache are some test circuits that were included because there was space. It has no effect on the function of the processor.

The description of the signals is divided into three classes:

- outgoing signals that are driven off the chip,
- incoming signals that come on chip,
- test pins.

Detailed descriptions of the signals are given in Section 7.1.1.

### 7.2.1 Outgoing Signals

The control signals that are driven off the chip are shown in Table 7.4. These signals are issued from a small pipeline that begins at the Instruction Register. The Op/Type field bits are decoded to the required signals and then delayed till they are needed at the pads. Three events can change this process:

1. during an instruction cache miss the signals are forced to perform a Load to service the miss,
2. an Exception or a Reset forces all signals in the pipe to a *nop*,
3. if a Squashing Branch does not go, the two squashed instructions have their signals forced to *nop*.

A *nop* in this instance means that no external operation is performed.

The values driven on the pins for various instructions are shown in Table 7.5. Write-back Enable is the same signal as Cop but delayed by one more cycle. The Floating-Point Register (FPReg) signals are a delayed version of the Src2/Dst field and are tapped off the end of the register decoders. The FPReg signals are ignored during the conditions described above so they are not modified like the other signals.

Name	Signal	Bits	Timing
Read/Write_b	R/W	1	stable $\phi$ 1 MEM
Coprocessor Cycle	Cop	1	stable $\phi$ 1 MEM
Bypass External Cache	Bypass	1	stable $\phi$ 1 MEM
Memory Access	Mem	1	stable $\phi$ 1 MEM
Write-back Enable	WB	1	stable $\phi$ 1 WB
Floating-Point Register	FPReg	4	stable $\phi$ 1 MEM

Table 7.4: Outgoing signals.

Instruction	Mem	R/W	Bypass	Cop
ld	1	1	0	0
st	1	0	0	0
ldf	1	1	0	1
stf	1	0	0	1
movtoc	0	0	0	1
movfrc	0	1	0	1
ldt	1	1	1	0
stt	1	0	1	0
nop	0	1	0	0

Table 7.5: Outgoing signal values for various instructions.



Incoming Signal	Derived Signal	Logic
Phi1	Psi1 Psi1PC	$\overline{Phi1} \cdot \overline{ICacheMiss} \cdot \overline{Miss}$ $Phi1 \cdot ICacheMiss$
Phi2	Phi2-b	$\overline{Phi2}$
Interrupt	Interrupt_s2	
Reset	Reset_s2	
Exception	Exception_b-pv2	

Table 7.6: Incoming signals and derived signals.

### 7.2.2 Incoming Signals

The control signals that come on chip are listed in Table 7.6. These signals are buffered and driven to the rest of the chip. Some are also combined with other logic to derive other signals. These derived signals are also shown in Table 7.6 along with the logic.

### 7.2.3 Test Pins

There are several features in MIPS-X that allow testing of various parts of the chip. The pins *ICacheTest* and *Reset* can be used to test the ICache. This is described in detail in Section 7.1.11. The *ICacheDisable* pin is used to disable the ICache. This allows testing of the datapath without worrying about problems in the ICache and also allows the processor to be used if defects in the ICache are discovered. With the ICache disabled, the processor will always do ICache miss cycles so the pattern observed is two ICache miss cycles followed by two execute cycles.

The Revision 1 MIPS-X part includes a *Test* pin that can be used to enable observation of control lines in the datapath. With the *Test* pin enabled, the four floating-point register pins are converted into input pins and are used to select control lines to be driven onto the MemBus. This allows up to 128 control lines to be examined on the data pads. This feature was never used so the circuits were removed in the Revision 2 MIPS-X part.

# Appendix A

## Exception Handling

An exception is defined as either an event that causes an interrupt or a *trap* instruction that can be thought of as a *software* interrupt. The two sequences cause similar actions in the processor hardware. Because there is a branch delay of two, three PCs from the PC Chain must be saved and restarted on an interrupt. Three PCs are needed in the event that a branch has occurred and fallen off the end of the chain. The two branch slot instructions and the branch destination are saved for restarting. Restarting a trap is slightly different and is explained later. Section 2.1.4 describes what happens in the PSW during interrupts, exceptions, and traps.

### A.1 Interrupts

Interrupts are asynchronous events that the programmer has no control over. Because there are several instructions executing at the same time, it is necessary to save the PCs of all the instructions currently executing so that the machine can be properly restarted after an interrupt. The PCs are held in the *PC Chain*. When an interrupt occurs, the PC Chain is frozen (stops shifting in new values) to allow the interrupt routine to save the PCs of the three instructions that need to be restarted. These are the PCs of the instructions that are in the RF, ALU and MEM cycles of execution. This means that no further exceptions can occur while the PCs are being saved. When the interrupt sequence begins, the interrupts are disabled, *PSW<sub>current</sub>* is copied into *PSW<sub>other</sub>* and the machine begins execution in system state. The contents of *PSW<sub>other</sub>*

should be saved if interrupts are to be enabled before the return from the interrupt. The contents of the *MD* register must also be saved and restored if any multiplication or division is done. If the interrupt routine is very short and interrupts can be left off, it is possible to just leave the PC Chain frozen, otherwise the three PCs must be saved. To save the PCs, use *movfrs* with *pcm4* (PC minus 4, the register containing the PC fetched four instructions ago) as the source. The PC Chain shifts after each read of *pcm4*. The interrupt routine will start execution at location 0.

To return from an interrupt, interrupts must first be disabled to allow the state of the machine to be restored. The PSW must be restored and the PC Chain loaded with the return addresses. The PC Chain is loaded with the addresses of the three instructions to be restarted by writing to *pcm1* (PC minus 1, the register that holds the PC of the previously fetched instruction). The PC Chain shifts after each write to *pcm1*. The instructions are restarted by doing three jumps to the address in *pcm4* and having shifting of the PC Chain enabled.

The first of the three jumps should be a *jpcrs* instruction. It will cause PSW<sub>other</sub> to be copied to PSW<sub>current</sub> with the interrupts turned on and the state returned to user space. The machine state changes after the ALU cycle of the first jump. The effective result is a jump using the contents of *pcm4* as the destination address, and shifting the next address into *pcm4*. The last two instructions of the return jump sequence should be *jpc* instructions, which also use the contents of *pcm4* as the destination address. The *jpc* instructions do not change the state of the machine.

A problem arises because an exception could occur while restarting these instructions. The PC Chain is now in a state that it is not possible to restart the sequence again using the standard sequence of first saving the PC Chain. The start of an exception sequence should first check the *e* bit in the PSW to see whether it is cleared. The *e* bit will be set only when the PC Chain is back in a normal state. If it is clear, then the state of the machine should not be resaved. The state to use for restart should still be available in the process descriptor for the process being restarted when the exception occurred. The sequence for interrupt handling is shown in Figure A.1.

```

.
.
.
lret:  inst  a                ;Instructions a,b and c are restarted
      inst  b
      inst  c
      --- interrupt ---
      inst  d
      inst  e
.
.
.

inthlr: bra to save if e bit set ;Start of interrupt handler
        Do necessary fixes      ;e bit clear so don't save PC Chain
        bra nosave
save:   Save PSWother           ;do save if interrupts to be enabled
        Save MD                 ;if necessary
        movfrs  pcm4,rA         ;save PCs if necessary
        movfrs  pcm4,rB
        movfrs  pcm4,rC
nosave: Enable interrupts      ;if necessary and above saving done
.
.
        Process interrupts
.
.
        Disable interrupts
        Restore MD              ;if necessary
        Restore PSWother       ;if necessary
        movtos  rA,pcm1        ;restore PCs
        movtos  rB,pcm1
        movtos  rC,pcm1
        jpcrs                    ;This changes the PSW as well
        jpc                      ;Doesn't touch PSW
        jpc
        execution begins at label lret

```

Figure A.1: Interrupt sequence.

## A.2 Trap On Overflow

A trap on overflow behaves exactly like an interrupt except that it is generated on-chip instead of externally. This interrupt can be masked by setting the *V* bit in the PSW.

When a trap on overflow occurs, the *O* bit is set in the PSW. The exception handling routine must check this bit to see if an overflow is the cause of the exception.

## A.3 Trap Instructions

Besides the Trap on Overflow, there is only one other type of trap available. It is an unconditional vectored trap to a system space routine in low order memory. After the ALU cycle of the trap instruction the processor goes into system state with the PC Chain frozen. The instruction before the *trap* instruction will complete its WB cycle. The PSW is saved by copying PSW<sub>current</sub> to PSW<sub>other</sub> as described in Section 2.1.4. PSW<sub>current</sub> is loaded as if this were an interrupt.

Before interrupts can be turned on again, some processor state must be saved. The return PCs are currently in the PC Chain. Three PCs must be read from the PC Chain and the third one saved in the process descriptor. It is the instruction that is in the RF cycle. The instruction corresponding to the PC in MEM completes so it need not be restarted. The PC in the ALU cycle should not be restarted because it is the *trap* instruction. PSW<sub>other</sub> must be saved so that the state of the prior process is preserved. If PSW<sub>other</sub> is not saved before interrupts are enabled, then another interrupt will smash the PSW of the process that executed the trap before it can be saved.

All trap instructions have an 8-bit vector number attached to them. This provides 256 legal trap addresses in system space. These addresses are 8 locations apart to provide enough space to store some instructions that will cause a jump to the correct handler. If this is not enough vectors, one of the traps can take a register as an argument to determine the action required.

The return sequence must disable interrupts, restore the contents of PSW<sub>other</sub> and MD if they were saved and then disable PC shifting so that the return address can be shifted into the PC Chain. Two more addresses must be shifted in as well so that the restart will look the same as an interrupt. This can be done by loading the addresses of two

*nop* instructions into the PC Chain ahead of the return address. Three jumps to the addresses in the PC Chain are then executed using *jpcrs* and two *jpcs*. The first jump will copy the contents of PSWother into PSWcurrent and turn on PC shifting. The processor state changes after the ALU cycle of the *jpcrs*. The change of state also enables interrupts and puts the processor in user space.

If an interrupt occurs during the return sequence then the interrupt handler will look at the *e* bit in the PSW to determine whether the state should be saved.

The flow of code for taking a trap and returning is shown in Figure A.2.

```

      .
      .
      .
      trap    vecnum
lret:  .
      .
      .

vecnum: movfrs  pcm4,r0      ;instruction before trap
        movfrs  pcm4,r0      ;trap instruction
        movfrs  pcm4,r31     ;save this one to restart
        Save PSWother       ;if necessary
        Save MD              ;if necessary
        Enable interrupts    ;if necessary and above saving done
      .
      .
      Process requested trap
      .
      .
      Disable interrupts     ;movtos x,pswc where x has M bit set
      Restore MD             ;if necessary
      Restore PSWother       ;if necessary
      movtos    r0,pcm1      ;assume a nop at 0
      movtos    r0,pcm1      ;
      movtos    r31,pcm1     ;instruction after trap
      jpcrs
      jpc
      jpc
      execution begins at label lret

```

Figure A.2: Trap sequence.

## Appendix B

# Integer Multiplication and Division

This appendix describes the multiplication and division support on MIPS-X. The philosophy behind why the current implementation was chosen is described first and then the instructions for doing multiplication and division are described.

### B.1 Multiplication and Division Support

The goal of the multiplication and division support in MIPS-X is to provide a reasonable amount of support with the smallest amount of hardware possible. Speed ups can be obtained by realizing that most integer multiplications are used to obtain a 32-bit result, not a 64-bit result. The result is usually the input to another operation, or it is the address of an array index. In either case, a number larger than 32 bits would not make sense. Since the result is less than 32 bits, one of the operands is most likely to be less than 16 bits or there will be an overflow. In general this means that only about 16 1-bit multiplication or division steps are required to generate the final answer. For very small constants, instructions can be generated inline instead of using a general multiplication or division routine. Therefore, it was felt that there was no great advantage to implement a scheme that could do more than one bit at a time such as Booth multiplication.

The other advantage of only generating a 32-bit result is that it is possible to do multiplication starting at the MSB of the multiplier

meaning that the same hardware can be used for multiplication and division. The required hardware is a single register, the MD register, that can shift left by one bit each cycle, and an additional multiplexer at the source one input of the ALU, that selects the input or two times the input for the source 1 operand.

## B.2 Multiplication

Multiplication is done with the simple 1-bit shift and add algorithm except that the computation is started from the most significant bit instead of the least significant bit of the multiplier. The instruction that implements one step of the algorithm is called *mstep*. For

*mstep* *rSrc1*,*rSrc2*,*rDest*

the operation is:

If the MSB of the MD register is 1

then

$rDest \leftarrow 2 \times rSrc1 + rSrc2$

else

$rDest \leftarrow 2 \times rSrc1$

Shift left MD

For signed multiplication, the first step is different from the rest. If the MSB of the multiplier is 1, the multiplicand should be subtracted from 0. The instruction called *mstart* is provided for this purpose. For

*mstart* *rSrc2*,*rDest*

the operation is:

If the MSB of the MD register is 1

then

$rDest \leftarrow 0 - rSrc2$

else

$rDest \leftarrow 0$

Shift left MD



Number of <i>msteps</i> needed	8	32
Number of cycles with positive multiplier	13	42
Number of cycles with negative multiplier	15	42

Table B.1: Number of cycles needed to do a multiplication.

To show the simplest implementation of a multiplication routine assume that the following registers have been assigned and loaded

*rMer* is the multiplier,  
*rMand* is the multiplicand,  
*rDest* is the result register,  
*rLink* is the jump linkage register.

Then,

```

movtos  rMer,rMD           ;Move the multiplier into MD
nop                                           ;Needed for hardware timing reasons
mstart  rMand,rDest        ;Do the first mstep. Result goes into rDest
mstep   rDest,rMand,rDest ;Repeat 31 times
jspci   rLink,#0,r0       ;Return

```

It is possible to speed up the routine by using the assumption described previously that the numbers will not both be a full 32 bits long. The simplest scheme is to check to see if the multiplier is less than 8 bits long. Some statistics indicate that this occurs frequently.

The routine shown in Figure B.1 implements multiplication with less than 32 *msteps* on average. It will actually do a full 32 *msteps* if it is necessary. In this case it is most likely that overflow will occur and this can be detected if the *V* bit in the PSW is clear so that a trap on overflow will occur. Assume that the registers *rMer*, *rMand* and *rDest* have been assigned and loaded as in the previous example. Two temporary registers, *rTemp1* and *rTemp2* are also required.

The number of cycles required, not including the instructions needed for the call sequence is shown in Table B.1. Compare this with the simple routine using just 32 steps which requires 35 instructions to do the multiplication and a Booth 2-bit algorithm that will need about 19 instructions. It can be observed that if most multiplications require 8 or less *msteps*, then this routine will be faster than just doing 32 *msteps* all the time.

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; MUL
;    fast, unchecked, signed multiply
;        rLink = link
;        rMand = src2
;        rDest = rMer = src1/dest
;        rTemp1 = temp
;        rTemp2 = temp
;
;    Note: This code has been reorganized
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
MUL:
    asr    rMer,rTemp2,#7          ; Test for positive 8-bit number
    bne    rTemp2,r0,lnot8
    sh     r0,rMer,rTemp1,#24     ; assume 8 bit
    movtos rTemp1,md
    mstart rMand,rDest           ; may need nop before this
    mstep  rDest,rMand,rDest
lmul8bit:
    mstep  rDest,rMand,rDest
    mstep  rDest,rMand,rDest
    mstep  rDest,rMand,rDest
    mstep  rDest,rMand,rDest
    jspci  rLink,#0,r0
    mstep  rDest,rMand,rDest
    mstep  rDest,rMand,rDest
lnot8:
    addi   rTemp2,#1,rTemp2
    beqsq  rTemp2,r0,lmul8bit    ; 8 bit negative
    mstart rMand,rDest
    mstep  rDest,rMand,rDest
    movtos rDest,md             ; do full 32 bits
    mstart rMand,rDest          ; may need nop before this
    mstep  rDest,rMand,rDest
    mstep  rDest,rMand,rDest
    mstep  rDest,rMand,rDest
    mstep  rDest,rMand,rDest
    .
    .
    .
    24 msteps
    .
    .
    .
    mstep  rDest,rMand,rDest
    jspci  rLink,#0,r0
    mstep  rDest,rMand,rDest
    mstep  rDest,rMand,rDest

```

Figure B.1: Signed integer multiplication.

## B.3 Division

For division, the same set of hardware is used, except the ALU is controlled differently. The algorithm is a restoring division algorithm. Both of the operands must be positive numbers. Signed division is not supported as it is too hard to do for the hardware required[15].

The dividend is loaded in the *MD* register and the register that will contain the remainder (*rRem*) is initialized to 0. The divisor is loaded into another register called (*rDor*). The result of the division (quotient) will be in *MD*. For

dstep *rRem*,*rDor*,*rRem*

the operation is:

Set *ALUsrc1* input to  $2 \times rRem + MSB(rMD)$

Set *ALUsrc2* input to *rDor*

$ALUoutput \leftarrow ALUsrc1 - ALUsrc2$

If  $MSB(ALUoutput)$  is 1

then

$rRem \leftarrow ALUsrc1$

$rMD \leftarrow 2 \times rMD$

else

$rRem \leftarrow ALUoutput$

$rMD \leftarrow 2 \times rMD + 1$

At the end of 32 *dsteps* the quotient will be in the *MD* register, and the remainder is in *rRem*.

A routine for doing division is shown in Figure B.2. The dividend is passed in *rDend* and the divisor in *rDor*. At the end, the quotient is in *MD* and *rQuot*, and the remainder is in *rRem*. Note that *rDend* and *rRem* can be the same register, and *rDor* and *rQuot* can be the same register. The dividend and divisor are checked to make sure they are positive. This routine does a 32-bit by 32-bit division so no overflow can occur.

The number of cycles needed, not including the calling sequence and assuming the operands are positive, is shown in Table B.2.

## 204 APPENDIX B. INTEGER MULTIPLICATION AND DIVISION

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; DIV
;
;      fast, unchecked, signed divide (should check for zero divide);
;      rLink = link
;      rDend,rRem = src1          (dividend)
;      rDor = rQuot = src2/dest   (divisor/quotient)
;      rTemp1 = temp              (trashed)
;      rTemp2 = temp              (trashed)
;      Note: This code has been reorganized
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DIV:
    mov    rDend,rTemp2          ; dividend > 0 ?
    bge    rDend,r0,lcinit1
    nop
    nop
    sub    r0,rDend,rDend        ; make dividend > 0
lcinit1:
    bgesq  rDor,r0,lcinit2      ; divisor > 0 ?
    addi   r0,#0xff,rTemp1      ; check for 8-bit dividend
    nop
    sub    r0,rTemp2,rTemp2      ; rTemp2 > 0 if positive result
    sub    r0,rDor,rDor          ; make divisor > 0
    addi   r0,#0xff,rTemp1
lcinit2:
    bltsq  rTemp1,rDend,ldivfull ; do 8-bit check
    movtos rDend,md              ; start 32-bit divide
    mov    r0,rRem
    sh    r0,rDend,rDend,#8      ; shift up divisor to do 8 bits
    movtos rDend,md              ; start 8-bit divide
    beq    r0,r0,ldivloop
    mov    r0,rRem
    addi   r0,#8,rTemp1          ; loop counter
ldivfull:
    addi   r0,#32,rTemp1         ; do full 32 dsteps
ldivloop:
    dstep  rRem,rDor,rRem
    dstep  rRem,rDor,rRem
ldivloopr:
    dstep  rRem,rDor,rRem        ; do this instruction 5 times, not shown
    addi   rTemp1,#-8,rTemp1     ; decrement loop counter
    dstep  rRem,rDor,rRem
    bnesq  rTemp1,r0,ldivloopr
    dstep  rRem,rDor,rRem
    dstep  rRem,rDor,rRem
    movfrs md,rQuot              ; get result
    bge    rTemp2,r0,lcinit3     ; check if need to adjust
    ;      sign of result
    nop
    nop
    sub    r0,rQuot,rQuot        ; adjust sign of result
lcinit3:
    jspci  rLink,#0,rLink        ; return
    nop
    nop

```

Figure B.2: Signed integer division.

Number of <i>dsteps</i> needed	8	32
Number of cycles needed	34	60

Table B.2: Number of cycles needed to do a divide.

# Appendix C

## Opcode Map

Figure 2.1 shows the MIPS-X instruction formats. This is a summary of how the bits in the instruction opcodes have been assigned. The first sections will show how the bits in the *OP* and *Comp Func* fields are assigned. Then the opcode map of the complete instruction set will be given.

### C.1 OP Field Bit Assignments

The *OP* bits are bits 2-4 in all instructions. For *memory* type instructions the bits have no particular meaning by themselves. For *branch* type instructions the bits in the *OP* field (also known as the *Cond* field) are assigned as follows:

**Bit 2** Set to 0 if branch on condition true, set to 1 if branch on condition false

**Bits 3-4** Condition upon which the branch decision is made. 00 = unused, 01 = Z, 10 = C, 11 =  $N \oplus V$

For *compute* type instructions the bits are assigned as follows:

**Bit 2** Set to 1 if the ALU *always* drives the Result Bus for the instruction

**Bit 3** Set to 0

**Bit 4** Set to 1 if the shifter *always* drives the Result Bus for the instruction

For *compute immediate* type instructions the bits are assigned as follows:

- Bit 2** Set to 1 if the ALU *always* drives the Result Bus for the instruction
- Bits 3-4** These bits have no particular meaning by themselves

## C.2 Comp Func Field Bit Assignments

The *Comp Func* bits are bits 20 through 31 in the *compute* and *compute immediate* type instructions. The bits are assigned according to whether they are being used by the ALU or the shifter. The bits for the ALU are assigned in the following way:

- Bits 20-22** Unused
- Bit 23** Set to 1 for dstep, 0 otherwise
- Bit 24** Set to 1 for multiply instructions (mstart, mstep), 0 otherwise
- Bit 25** Carry-in to the ALU
- Bits 26-29** Input to the *P* function block.
- |               |   |
|---------------|---|
| <b>Bit 26</b> | $\text{Src1} \cdot \text{Src2}$                       |
| <b>Bit 27</b> | $\text{Src1} \cdot \overline{\text{Src2}}$            |
| <b>Bit 28</b> | $\overline{\text{Src1}} \cdot \text{Src2}$            |
| <b>Bit 29</b> | $\overline{\text{Src1}} \cdot \overline{\text{Src2}}$ |
- Bits 30-31** Input to the *G* function block.
- |               |   |
|---------------|---|
| <b>Bit 30</b> | 0 for ALU add operation, 1 otherwise      |
| <b>Bit 31</b> | 0 for ALU subtract operation, 1 otherwise |

The bits for the shifter are assigned as follows:

- Bits 20-21** Unused
- Bit 22** Set to 1 for funnel shift operation (sh instruction)
- Bit 23** Set to 1 for arithmetic shift operation (asr instruction)
- Bit 24** Set to 1 for byte rotate instructions (rotlb, rotlcb)
- Bit 25** For byte rotate instructions, set to 1 if rotlb, 0 if rotlcb

**Bits 25-31** Shift amount for funnel and arithmetic shift operations (sh and asr instructions). The range is 0 to 31 bits. Although this can be encoded in five bits, the two low-order bits are fully decoded; therefore, the field is seven bits. The two low-order bits are decoded as follows: 0 = bit 31, 1 = bit 30, 2 = bit 29, 3 = bit 28. For example, a shift amount of 30 would become 1110100 in this seven-bit encoding scheme.



### C.3 Opcode Map of All Instructions

#### Memory Instructions

Instruction	TY	OP	Comments
ld	10	000	*
st	10	010	
ldf	10	100	*
stf	10	110	
ldt	10	001	*
stt	10	011	
movfrc	10	101	Src1=0, *
movtoc	10	111	Src1=0
aluc	10	101	Src1=0, Dest=0, *

#### Branch Instructions

Instruction	TY	COND
beq	00	001
bge	00	111
bhs	00	010
blo	00	110
blt	00	011
bne	00	101

#### Compute Instructions

Instruction	TY	OP	Comp Func	Comments
add	01	100	000000011001	
dstep	01	000	000101100110	
mstart	01	000	000011100110	Src1=0
mstep	01	000	000010011001	
sub	01	100	000001100110	
subnc	01	100	000000100110	
and	01	100	000000100011	
bic	01	100	000000001011	Src2=0
not	01	100	000000001111	Src2=0
or	01	100	000000111011	
xor	01	100	000000011011	
mov	01	100	000000011001	Src2=0
asr	01	001	00010bbbddd	Src2=0, bbbddd=rotate amount
rotlb	01	001	000011000000	
rotlcb	01	001	000010000000	
sh	01	001	00100bbbddd	bbbddd=rotate amount
nop	01	100	000000011001	Src1=0, Src2=0, Dest=0

## Compute Immediate Instructions

Instruction	TY	OP	Comp Func	Comments
addi	11	100	Immed	* (Immed is a 17-bit
jspci	11	000	Immed	* signed constant)
jpc	11	101	00000000011	*
jpcrs	11	111	00000000011	
movfrs	11	011	000000000rrr	rrr = special register
movtos	11	010	000000000rrr	rrr = special register
trap	11	110	0vvvvvvvv011	Src1=0, vvvvvvvv=vector
unused	11	001		

## Key to Comments

**star (\*)** This is an instruction that has its *Dest* field in the position where the *Src2* field normally sits. This can also be determined by decoding the MSB of the type field and the middle bit of the OP field.

**bbbddd** This is the *shift amount* for the *asr* and *sh* instructions. It ranges from 1 to 32 bits.

To determine the encoding for the *shift amount*, first subtract the *shift amount* from 32. The result can be encoded as 5 bits. Assume the 5-bit encoding is *bbbef*, where *bbb* is used in the final encoding. The bottom two bits (*ef*) are fully decoded to yield *ddd* in the following way:

ef	ddd
00	0001
01	0010
10	0100
11	1000

For example, to determine the bits required to specify the shift amount for the shift instruction

**asr r4,r3,#5**

first do  $(32 - 5)$  to get 27 and then encode 27 according to the above to get 1101000.

**rrr** This field specifies the special register according to the following table:

Special Register	rrr
psw	001
md	010
pcm4 ( <i>movfrs</i> ), pcm1 ( <i>movtos</i> )	100

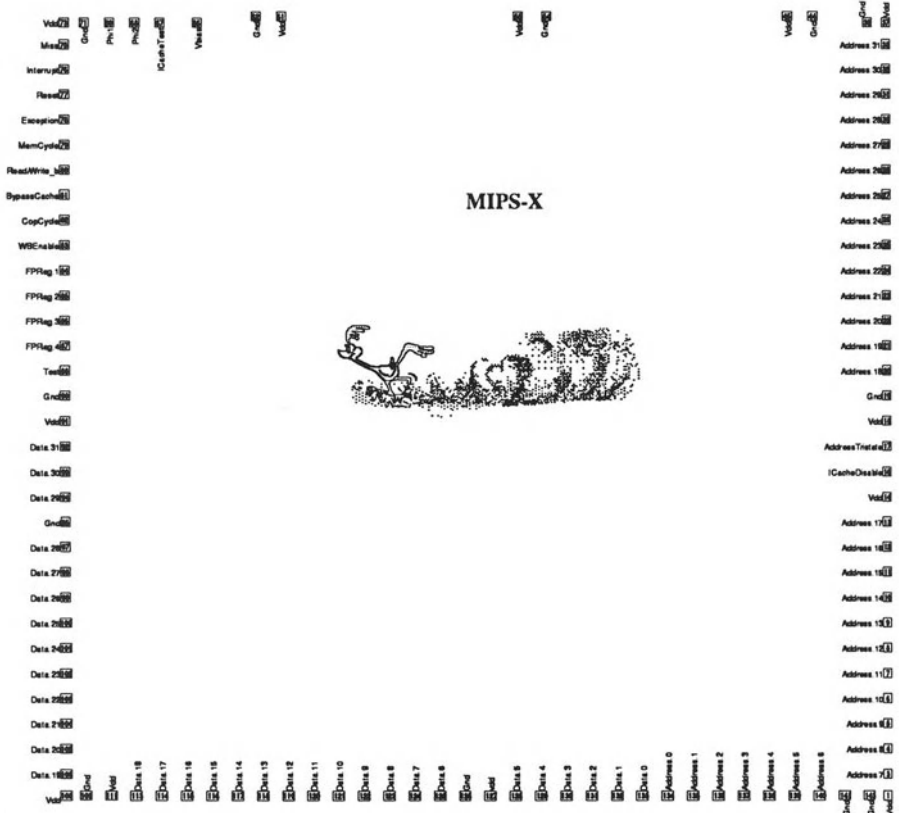
**vvvvvvvv** This is the one's complement of the vector.

## Appendix D

# MIPS-X Revision 1 and 2 Pin Numbers

The pin assignments for both Revision 1 and Revision 2 are the same except that the Revision 2 part does not have a *Test* pin. This pin is not connected (NC) on the Revision 2 part.

Two lists are given. The first one gives the correspondence between the pin numbers and names for the medium tester jumpers, the probe card, the layout and the functional simulator (Funsim). The names given are the names that are used in the layout and may not be exactly the same as the names used elsewhere in this document. The second list maps the pin number of the 144 pin PGA and the signal names to the medium tester jumpers and 64-pin connectors. A map of the pads is shown below. The numbers on the pins are the bonding pad numbers for the 144-pin ceramic PGA from Kyoto Ceramic Co., Ltd. (Kyocera).



## D.1 Pin Mapping for Probe Card and Funsim

This is the list of the pad names and their corresponding pin numbers for the tester, probe card and the layout. For the layout the numbering of the pads starts with Pin 1 in the top left corner of the chip and going counter-clockwise. For each pin, the name used in the layout and the name in the functional simulator are given.

The *ClockTri* pin is used to switch the state of the multiplexer that selects between using the tester to stimulate the clock pins and using an external clock for speed testing. This is only a connection between the tester and the probe card pins. A low value on this pin selects the tester as the clock source.

Tester Jumper #	Probe Card #	Layout Pin #	Pad Name	Funsim Name
120	B 56	1	Vdd - Pad	
56	A 56	2	PadMiss	PadMiss_v1
119	B 55	3	PadInterrupt_v1	PadInterrupt_v1
55	A 55	4	PadReset	PadReset
118	B 54	5	PadException_v2	PadException_v2
54	A 54	6	PadMem	PadMem
117	B 53	7	PadRead	PadReadWrite
53	A 53	8	PadBypass	PadBypass
116	B 52	9	PadCop	PadCop
52	A 52	10	PadWBen	PadWBen
115	B 51	11	PadFPReg.1	PadFPReg
51	A 51	12	PadFPReg.2	"
114	B 50	13	PadFPReg.3	"
50	A 50	14	PadFPReg.4	"
113	B 49	15	PadTest(NC in Rev.2) not in funsim	
49	A 49	16	Gnd - Pad	
112	B 48	17	Vdd - Pad	
48	A 48	18	PadDataBus.31	PadDataBus_s1
111	B 47	19	PadDataBus.30	"
47	A 47	20	PadDataBus.29	"
110	B 46	21	Gnd - Reg Array,Pad	
46	A 46	22	PadDataBus.28	"
109	B 45	23	PadDataBus.27	"
45	A 45	24	PadDataBus.26	"
108	B 44	25	PadDataBus.25	"
44	A 44	26	PadDataBus.24	"
107	B 43	27	PadDataBus.23	"
43	A 43	28	PadDataBus.22	"
106	B 42	29	PadDataBus.21	"

216 APPENDIX D. MIPS-X REVISION 1 AND 2 PIN NUMBERS

Tester Jumper #	Probe Card #	Layout Pin #	Pad Name	Funsim Name
-----	-----	-----	-----	-----
42	A 42	30	PadDataBus.20	"
105	B 41	31	PadDataBus.19	"
41	A 41	32	VddPad	
104	B 40	33	Gnd - Pad	
40	A 40	34	Vdd - Reg Array	
103	B 39	35	PadDataBus.18	"
39	A 39	36	PadDataBus.17	"
102	B 38	37	PadDataBus.16	"
38	A 38	38	PadDataBus.15	"
101	B 37	39	PadDataBus.14	"
37	A 37	40	PadDataBus.13	"
100	B 36	41	PadDataBus.12	"
36	A 36	42	PadDataBus.11	"
99	B 35	43	PadDataBus.10	"
35	A 35	44	PadDataBus.9	"
98	B 34	45	PadDataBus.8	"
34	A 34	46	PadDataBus.7	"
97	B 33	47	PadDataBus.6	"
33	A 33	48	Gnd - Pad	
96	B 32	49	Vdd - Pad	
32	A 32	50	PadDataBus.5	"
95	B 31	51	PadDataBus.4	"
31	A 31	52	PadDataBus.3	"
94	B 30	53	PadDataBus.2	"
30	A 30	54	PadDataBus.1	"
93	B 29	55	PadDataBus.0	"
29	A 29	56	PadAddressBus.0	AddressBus
92	B 28	57	PadAddressBus.1	"
28	A 28	58	PadAddressBus.2	"
91	B 27	59	PadAddressBus.3	"
27	A 27	60	PadAddressBus.4	"
90	B 26	61	PadAddressBus.5	"
26	A 26	62	PadAddressBus.6	"
89	B 25	63	Gnd - Tags,Pad	
25	A 25	64	Gnd - Pad	
88	B 24	65	Vdd - Pad	
24	A 24	66	PadAddressBus.7	"
87	B 23	67	PadAddressBus.8	"
23	A 23	68	PadAddressBus.9	"
86	B 22	69	PadAddressBus.10	"

Tester Jumper #	Probe Card #	Layout Pin #	Pad Name	Funsim Name
-----	-----	-----	-----	-----
22	A 22	70	PadAddressBus.11	"
85	B 21	71	PadAddressBus.12	"
21	A 21	72	PadAddressBus.13	"
84	B 20	73	PadAddressBus.14	"
20	A 20	74	PadAddressBus.15	"
83	B 19	75	PadAddressBus.16	"
19	A 19	76	PadAddressBus.17	"
82	B 18	77	Vdd - Tags	
18	A 18	78	PadICacheDisable	PadCacheDisable
81	B 17	79	PadAddressTristate	PadTriState
17	A 17	80	Vdd - Pad	
80	B 16	81	Gnd - Pad	
16	A 16	82	PadAddressBus.18	AddressBus
79	B 15	83	PadAddressBus.19	"
15	A 15	84	PadAddressBus.20	"
78	B 14	85	PadAddressBus.21	"
14	A 14	86	PadAddressBus.22	"
77	B 13	87	PadAddressBus.23	"
13	A 13	88	PadAddressBus.24	"
76	B 12	89	PadAddressBus.25	"
12	A 12	90	PadAddressBus.26	"
75	B 11	91	PadAddressBus.27	"
11	A 11	92	PadAddressBus.28	"
74	B 10	93	PadAddressBus.29	"
10	A 10	94	PadAddressBus.30	"
9	A 9	95	PadAddressBus.31	"
8	A 8	96	Vdd - Pad	
7	A 7	97	Gnd - Pad	
69	B 5	98	Gnd - ICache	
68	B 4	99	Vdd - ICache	
65	B 1	100	Gnd - ICache Sense Amp	
128	B 64	101	Vdd - ICache	
127	B 63	102	Vdd - ICache	
126	B 62	103	Gnd - ICache	
125	B 61	104	Vbias	
60	A 60	105	PadICacheTest_s1	PadTest
59	A 59	106	PadPhi2	
58	A 58	107	PadPhi1	
57	A 57	108	Gnd - Pad	
3	A 3	-	ClockTri	



## D.2 Pin Map for 144 Pin PGA

The mapping in the MIPS-X column is between the signal names and the pin numbers on the 144 pin PGA. The numbers in the TESTER column are for the 144-pin test adaptor that plugs into the medium tester and gives the tester jumper number and the number on the 64-pin connectors.

MIPS-X -----			TESTER -----		
Bonding Pad No.	Pin No.	Signal Name	Jumper	64-pin conn.	Notes
-----	-----	-----	-----	-----	-----
1	D 3	Vdd	8	A 8	
3	B 1	Address.7	24	A 24	
4	D 2	Address.8	87	B 23	
5	E 3	Address.9	23	A 23	
6	C 1	Address.10	86	B 22	
7	E 2	Address.11	22	A 22	
8	D 1	Address.12	85	B 21	
9	F 3	Address.13	21	A 21	
10	F 2	Address.14	84	B 20	
11	E 1	Address.15	20	A 20	
12	G 2	Address.16	83	B 19	
13	G 3	Address.17	19	A 19	
14	F 1	Vdd	17	A 17	
16	H 2	ICacheDisable	18	A 18	
17	H 1	AddressTristate	81	B 17	
18	H 3	Vdd	40	A 40	
19	J 3	Gnd	7	A 7	
20	J 1	Address.18	16	A 16	
21	K 1	Address.19	79	B 15	
22	J 2	Address.20	15	A 15	
23	K 2	Address.21	78	B 14	
24	K 3	Address.22	14	A 14	
25	L 1	Address.23	77	B 13	
26	L 2	Address.24	13	A 13	
27	M 1	Address.25	76	B 12	
28	N 1	Address.26	12	A 12	
29	M 2	Address.27	75	B 11	
30	L 3	Address.28	11	A 11	
31	N 2	Address.29	74	B 10	
32	P 1	Address.30	10	A 10	

Bonding Pad No.	Pin No.	Signal Name	Jumper	64-pin conn.	Notes
33	M 3	Address.31	9	A 9	
37	N 4	Vdd	41	A 41	
39	Q 2	Gnd	25	A 25	
44	Q 4	Gnd	33	A 33	
45	N 6	Vdd	68	B 4	
54	N 8	Gnd	49	A 49	
55	N 9	Vdd	82	B 18	
61	Q 11	Vdd	88	B 24	
62	P 11	Gnd	57	A 57	
66	N 11	Vbias	125	B 61	
67	P 13	ICacheTest	60	A 60	
68	Q 14	Phi2	59	A 59	(clock line)
69	N 12	Phi1	58	A 58	(clock line)
71	P 14	Gnd	65	B 1	
73	M 13	Vdd	96	B 32	
75	P 15	Miss	56	A 56	
76	M 14	Interrupt	119	B 55	
77	L 13	Reset	55	A 55	
78	N 15	Exception	118	B 54	
79	L 14	MemCycle	54	A 54	
80	M 15	Read/Write_b	117	B 53	
81	K 13	BypassCache	53	A 53	
82	K 14	CopCycle	116	B 52	
83	L 15	WBEnable	52	A 52	
84	J 14	FPPReg.1	115	B 51	
85	J 13	FPPReg.2	51	A 51	
86	K 15	FPPReg.3	114	B 50	
87	J 15	FPPReg.4	50	A 50	
88	H 14	Test	113	B 49	(NC on Rev.2)
90	H 13	Gnd	69	B 5	
91	G 13	Vdd	112	B 48	
92	G 15	Data.31	48	A 48	
93	F 15	Data.30	111	B 47	
94	G 14	Data.29	47	A 47	
95	F 14	Gnd	80	B 16	
97	E 15	Data.28	46	A 46	
98	E 14	Data.27	109	B 45	
99	D 15	Data.26	45	A 45	
100	C 15	Data.25	108	B 44	

## 220 APPENDIX D. MIPS-X REVISION 1 AND 2 PIN NUMBERS

Bonding Pad No.	Pin No.	Signal Name	Jumper	64-pin conn.	Notes
101	D 14	Data.24	44	A 44	
102	E 13	Data.23	107	B 43	
103	C 14	Data.22	43	A 43	
104	B 15	Data.21	106	B 42	
105	D 13	Data.20	42	A 42	
106	C 13	Data.19	105	B 41	
108	A 15	Vdd	120	B 56	
109	C 12	Gnd	89	B 25	
111	A 14	Vdd	127	B 63	
113	C 11	Data.18	103	B 39	
114	A 13	Data.17	39	A 39	
115	B 11	Data.16	102	B 38	
116	A 12	Data.15	38	A 38	
117	C 10	Data.14	101	B 37	
118	B 10	Data.13	37	A 37	
119	A 11	Data.12	100	B 36	
120	B 9	Data.11	36	A 36	
121	C 9	Data.10	99	B 35	
122	A 10	Data.9	35	A 35	
123	A 9	Data.8	98	B 34	
124	B 8	Data.7	34	A 34	
125	A 8	Data.6	97	B 33	
126	C 8	Gnd	104	B 40	
127	C 7	Vdd	128	B 64	
128	A 7	Data.5	32	A 32	
129	A 6	Data.4	95	B 31	
130	B 7	Data.3	31	A 31	
131	B 6	Data.2	94	B 30	
132	C 6	Data.1	30	A 30	
133	A 5	Data.0	93	B 29	
134	B 5	Address.0	29	A 29	
135	A 4	Address.1	92	B 28	
136	A 3	Address.2	28	A 28	
137	B 4	Address.3	91	B 27	
138	C 5	Address.4	27	A 27	
139	B 3	Address.5	90	B 26	
140	A 2	Address.6	26	A 26	
141	C 4	Gnd	110	B 46	
143	B 2	Gnd	126	B 62	

## Appendix E

# Revision 1 and Revision 2 Differences

There are a few differences between the Revision 1 and Revision 2 MIPS-X parts. Most have to do with making the External Interface easier to use. Some are internal changes that do not affect the external interface logic but are mentioned here for archival purposes.

1. On the Revision 1 parts, the instruction fetched from the Ecache during an ICache miss must be valid until  $\phi 1$  falling of the 'WB' cycle of the ICache miss cycle. For example, the first instruction fetched back must be stable until  $\phi 1$  falling of CM2. For Revision 2, the timing is the same as a normal load sequence; the data must be valid until  $\phi 2$  falls during the 'MEM' cycle of an ICache miss cycle. For example, the first instruction fetched back must be valid until CM1  $\phi 2$  falls.
2. The address pins of the Revision 1 parts are set to zero on every  $\phi 1$ , reflecting the precharging of the internal Result Bus. These pins are valid  $\phi 2$ , stable  $\phi 1$  on the Revision 2 parts.
3. The *Miss* pin cannot be used to stall the processor on the Revision 1 parts. The processor must be stalled by stretching  $\phi 2$ . The *Miss* pin stalls the processor correctly on the Revision 2 parts.
4. When an ICache miss occurs between the probe in the Ecache for a store instruction and the actual store, the store must be allowed to complete before the ICache miss sequence can begin. On the Revision 1 parts this interlock must be done by asserting *Miss* on

the processor. For Revision 2 parts, the processor does the interlock and stalls itself until the store completes.

5. Because of the change to the store interlock described in the previous point, the timing of when data appears on the data pins has changed. For Revision 1 parts, data is asserted on the pins during MEM  $\phi 2$ . The Revision 2 parts do different actions for stores and *movtoc* instructions. For store instructions, the data is driven off the chip when *Miss* falls during WB  $\phi 1$  and stays valid until  $\phi 1$  rises. Data for a *movtoc* instruction is driven off chip when *Miss* falls during MEM  $\phi 1$  and stays valid until *Miss* falls during WB  $\phi 1$ . If *Miss* is not high when  $\phi 1$  goes high, then all actions will occur on the rising edge of  $\phi 1$ .
6. There is only one difference in the pinouts of the two versions. The Revision 1 part has a *Test* pin that is used to enable observation of control lines in the datapath. This feature was never used so the pin is not connected in the Revision 2 part.

The following changes are internal changes only and do not affect the External Interface.

1. The testing multiplexers have been removed on the Revision 2 parts. This is why the *Test* pin has gone away.
2. The PLA in the PC Unit has been replaced by standard cell logic. This is to speed up the critical paths for branches and exceptions.
3. The branch path logic has been sped up.
4. The Tag section has been changed to speed up the valid-bit store access.

# Bibliography

- [1] Anant Agarwal, Paul Chow, Mark Horowitz, John Acken, Arturo Salz, and John Hennessy. On-Chip Instruction Caches for High-Performance Processors. In *1987 Stanford Conference on Advanced Research in VLSI*, pages 1–24, Stanford, California, March 1987.
- [2] Alfred V. Aho, Mahadevan Ganapathi, and Steven W.K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *TOPLAS*, to appear.
- [3] J.P. Banning. An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables. In *Proc. 6th POPL Conference*, pages 724–736. ACM, 1979.
- [4] David Callahan. The Program Summary Graph and Flow-sensitive Interprocedural Data Flow Analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, Atlanta, Georgia, June 1988. ACM.
- [5] Gregory Chaitin. Register Allocation and Spilling via Graph Coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, Boston, June 1982. ACM.
- [6] Frederick Chow. *A Portable Machine-Independent Global Optimizer—Design and Measurements*. PhD thesis, Stanford University, December 1983.
- [7] Frederick C. Chow and John L. Hennessy. Register Allocation by Priority-based Coloring. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 222–232, Montreal, Canada, June 1984. ACM.
- [8] Paul Chow. MIPS-X Instruction Set and Programmer's Manual. Technical Report CSL-86-289, Computer Systems Laboratory, Stanford University, May 1986.

- [9] Paul Chow and Mark Horowitz. Architectural Tradeoffs in the Design of MIPS-X. In *The 14th Annual International Symposium on Computer Architecture*, pages 300–308, Pittsburg, Pennsylvania, June 1987. IEEE.
- [10] Paul Chow and Mark Horowitz. The Design and Testing of MIPS-X. In *Fifth MIT Conference on Advanced Research in VLSI*, pages 95–114, MIT, Cambridge, MA, March 1988.
- [11] Danny Cohen. On Holy Wars and a Plea for Peace. *IEEE Computer*, 14(10):48–54, October 1981.
- [12] K. Cooper and K. Kennedy. Efficient Computation of Flow Insensitive Interprocedural Summary Information. In *Proc. '84 Sym. on Compiler Construction*, pages 247–258, Montreal, Canada, June 1984. ACM. Correction appears as Technical Report 87-60, Dept. of Computer Science, Rice University.
- [13] John R. Ellis. *A Compiler for VLIW Architectures*. PhD thesis, Yale University, December 1984.
- [14] J. Fisher, J. Ellis, J. Ruttenberg, and A. Nicolau. Parallel Processing: A Smart Compiler and a Dumb Machine. In *Proc. '84 Sym. on Compiler Construction*, pages 37–47, Montreal, Canada, June 1984. ACM.
- [15] John Gill, Thomas Gross, John Hennessy, Norman Jouppi, Steven Przybylski, and Christopher Rowen. Summary of MIPS Instructions. Technical Note 83-237, Stanford University, November 1983.
- [16] M.L. Griss, E. Benson, and A.C. Hearn. Current Status of a Portable LISP Compiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 276–283, Boston, June 1982. SIGPLAN, ACM.
- [17] M.L. Griss, E. Benson, and G.Q. Maguire. PSL: A Portable LISP System. In *Proceedings of the 1982 Conference on LISP and Functional Programming*, pages 88–97, Pittsburgh, August 1982. ACM.
- [18] M.L. Griss and A.C. Hearn. A Portable LISP Compiler. *Software - Practice and Experience*, 11(6):541–605, June 1981.
- [19] Thomas R. Gross. *Code Optimization of Pipeline Constraints*. PhD thesis, Stanford University, August 1983.
- [20] John Hennessy, Norman Jouppi, Forest Baskett, Thomas Gross, and John Gill. Hardware/Software Tradeoffs for Increased Performance. In *Symposium on Architectural Support for Programming*

*Languages and Operating Systems*, pages 2–11, Palo Alto, March 1982. SIGARCH/SIGPLAN.

- [21] John Hennessy, Norman Jouppi, John Gill, Forest Baskett, Alex Strong, Thomas Gross, Chris Rowen, and Judson Leonard. The MIPS Machine. In *Compton Spring*, pages 2–7. IEEE, 1982.
- [22] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, and Thomas Gross. Design of a High Performance VLSI Processor. In *Proceedings of the Third Caltech Conference on VLSI*, pages 33–54, March 1983.
- [23] John L. Hennessy. Pascal\*: A Pascal Based Systems Programming Language. Technical Report 174, Computer Systems Laboratory, Stanford University, June 1980.
- [24] John L. Hennessy and Thomas R. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM TOPLAS*, 5(3), July 1983.
- [25] Mark Horowitz, Paul Chow, Don Stark, Richard Simoni, Arturo Salz, Steven Przybylski, John Hennessy, Glenn Gulak, Anant Agarwal, and John Acken. MIPS-X: A 20 MIPS Peak, 32-Bit Microprocessor with On-Chip Cache. *IEEE Journal of Solid-State Circuits*, SC-22(5):790–799, October 1987.
- [26] Mark Horowitz, John Hennessy, Paul Chow, Glenn Gulak, John Acken, Anant Agarwal, Chorong-Yeung Chu, Scott McFarling, Steven Przybylski, Steve Richardson, Arturo Salz, Richard Simoni, Don Stark, Peter Steenkiste, Steve Tjiang, and Malcolm Wing. A 32b Microprocessor with On-Chip 2Kbyte Instruction Cache. In *ISSCC Digest of Technical Papers*, pages 30–31, 328, February 1987.
- [27] Manolis G. H. Katevenis. Reduced Instruction Set Computer Architectures for VLSI. Technical Report UCB/CSD 83/141, Computer Science Division (EECS), Univ. of CA at Berkeley, October 1983.
- [28] Leslie Lamport. *L<sup>A</sup>T<sub>E</sub>X User's Guide & Reference Manual*. Addison-Wesley Publishing Company, 1986.
- [29] Scott McFarling. *Profile Based Optimization*. PhD thesis, Stanford University, 1989. To appear.
- [30] Scott McFarling and John Hennessy. Reducing the Cost of Branches. *The 13th Annual International Symposium on Computer Architecture*, pages 396–403, June 1986.
- [31] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley Publishing Company, 1980.



- [32] John K. Ousterhout, Gordon T. Hamachi, Robert N. Mayo, Walter S. Scott, and George S. Taylor. The Magic VLSI Layout System. *IEEE Design & Test of Computers*, 2(1):19–30, February 1985.
- [33] David A. Patterson and Carlo H. Sequin. A VLSI RISC. *IEEE Computer*, 15(9):8–21, September 1982.
- [34] Steven Przybylski. The Design Verification and Testing of MIPS. In *1984 Conference on Advanced Research in VLSI, M.I.T.*, pages 100–109, Boston, MA, January 1984.
- [35] Steven A. Przybylski, Thomas R. Gross, John L. Hennessy, Norman P. Jouppi, and Christopher Rowen. Organization and VLSI Implementation of MIPS. *Journal of VLSI and Computer Systems*, 1(2):170–208, December 1984.
- [36] Stephen Richardson and Mahadevan Ganapathi. Code Optimization Across Procedures. *IEEE Computer*, 22(2):42–50, February 1989.
- [37] Stephen Richardson and Mahadevan Ganapathi. Interprocedural Optimization: Experimental Results. *Software - Practice and Experience*, 19(3), March 1989. To appear in the March, 1989 issue.
- [38] Walter S. Scott, Robert N. Mayo, Gordon Hamachi, and John K. Ousterhout, editors. 1986 VLSI Tools: Still More Works by the Original Artists. Report UCB/CSD 86/272, University of California at Berkeley, Berkeley, CA 94720, December 1985.
- [39] Robert W. Sherburne Jr., Manolis G. H. Katevenis, David A. Patterson, and Carlo H. Sequin. A 32b NMOS Microprocessor with a Large Register File. *IEEE Journal of Solid-State Circuits*, SC-19(5):682–689, October 1984.
- [40] Peter Steenkiste. *LISP on a Reduced-Instruction-Set Processor: Characterization and Optimization*. PhD thesis, Stanford University, March 1987.
- [41] Peter Steenkiste and John L. Hennessy. LISP on a Reduced-Instruction-Set-Processor. In *Proceedings of the 1986 Conference on LISP and Functional Programming*, pages 192–201, Boston, August 1986. ACM.
- [42] Peter Steenkiste and John L. Hennessy. A Simple Interprocedural Register Allocation Algorithm and its Effectiveness for LISP. Submitted for publication, 1987.

- [43] Peter Steenkiste and John L. Hennessy. Tags and Type Checking in LISP: Hardware and Software Approaches. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, October 1987. ACM/IEEE.
- [44] Christopher Terman. Simulation Tools for Digital LSI Design. Technical Report MIT/LCS/TR-304, MIT, September 1983.
- [45] Steven W.K. Tjiang. Twig Reference Manual. Technical Report 120, AT&T Bell Laboratories, 1985.
- [46] Arnold Weinberger. Large Scale Integration of MOS Complex Logic: A Layout Method. *IEEE Journal of Solid-State Circuits*, 2(4):182–190, April 1967.

# Index

- Ackerman, 4
- Address 0, 15, 56, 106, 111, 176, 182, 186
- Address Bus, 16, 36, 40, 115, 117-118, 120, 132-133, 161, 165, 177, 179, 189
- ALU, 8, 12-13, 34, 36, 41, 43, 46, 49, 56, 60-61, 63, 75, 89, 104, 108, 118, 190
  - circuit, 76
  - control for division, 203
  - drivers, 106
  - inputs, 34, 61, 82, 200, 203
  - output, 94, 207
  - overflow, 15, 63, 89
  - signals into, 101, 208
  - usage, 13, 37-38, 106
- branch
  - code reorganization, 23, 26-31
  - computing conditions, 12, 17, 60, 76, 78, 80, 82
  - critical path, 99, 104, 222
  - delay, 13, 15, 26, 28, 112, 193
  - design tradeoffs, 16
  - displacement, 11, 36
  - displacement adder, 34, 94, 108
  - instruction flow, 37
  - instructions, 9-10, 207
  - instructions on reset, 97, 112
  - result, 156
  - signals, 63-64, 78, 82, 89-90
  - signals in PC Unit, 97-99, 102, 104-105
  - squashing, 13, 17, 30-31, 39, 56, 61, 97, 111-112, 190
- cache
  - instruction, *see* instruction cache
  - external, *see* external cache
- CAD, 2-3, 30, 99
- cancelling, *see* instruction squashing
- clocks, 34, 40-41, 43-44, 109, 164, 167, 192
- CMFSM, *see* controllers, cache miss
- control lines, 41, 43-44, 60, 66, 192, 222
  - ALU, 78, 80
  - MD register, 73, 75-76
  - naming, 40
  - PC, 98-99, 101, 104
  - PSW, 85, 88
  - shifter, 69-71
- controllers, 3, 19, 41
  - cache miss, 39, 94, 97-98, 111, 113
  - global, 39, 41
  - squash, 39, 95, 97, 111, 114
- coprocessor, 1-2, 7, 9-10, 16-17, 161, 165, 168, 173, 175, 177, 179-180, 182, 191
  - alu function, 9, 177, 179, 184
  - exception, 186
  - floating-point, 10, 156, 161, 166, 184
- critical path, 99, 121

- branch, *see* branch, critical path
- memory, *see* memory, critical path
- division, 8-9, 15, 34, 49, 60, 63, 71, 75-76, 82-83, 194, 199, 210
- ECache, *see* external cache
- EFace, *see* External Interface
- exception, 7, 10, 13-16, 18, 34, 38-39, 56, 89, 94, 97, 109, 111, 165, 184, 188, 190, 193, 222
  - coprocessor, 180
  - during ICache miss, 175
  - effect on CMFSM, 111
  - effect on SQFSM, 112
  - MD action on, 71, 73
  - PSW action on, *see* PSW, on exception
  - signal, 38, 56, 61, 63, 89, 97, 99, 102, 166-167, 192, 215
  - signal in PC Unit, 99, 101-102, 104-105
  - vector, 106
- Execute Unit, 3, 5, 33-34, 36, 38, 41, 46, 49-50, 60-61, 94-95, 97-98, 101, 108, 117, 151, 157
- external cache, 10-12, 133, 161-163, 167-169, 177, 180, 184, 191, 221
  - bypass, 9, 165
  - miss, 109, 168, 170, 172, 176
  - miss rates, 163
- External Interface, 3, 5, 38, 62-63, 95, 117, 138, 157, 161, 221-222
- finite-state machines, 39
- ICache, *see* instruction cache
- immediate, 97, 99, 102, 104, 151
  - bus, 34, 36, 61, 69-70, 75-76, 82, 85, 89, 106, 108, 152, 154
  - instruction, 9-10, 12, 28, 61, 63, 82, 152, 207, 211
- instruction cache, 1, 3-5, 8, 11, 16, 33-34, 93, 151, 175, 190
  - bus, 156
  - disable, 166
  - in instruction flow, 37
  - miss, 12, 16, 34, 36, 39, 94, 98, 109, 151, 168, 173, 178, 180, 190, 192, 221
  - organization, 115
  - RAM, 33, 93-94, 137
  - reducing misses, 21
  - tags, *see* Tags Unit
  - testing, 95, 97, 112, 132, 166, 188, 192, 217
- Instruction Register, 3, 5, 33-34, 37, 56, 62, 71, 75, 85, 94-95, 97-98, 138, 151, 190
- instruction set, 7-10
  - op codes, 207
- instruction squashing, 39, 56, 88, 94-95, 176, 188
  - branches, *see* branch, squashing
  - coprocessor, 180
  - MD register, 71, 74
- interrupt, *see* exception
- IReg, *see* Instruction Register
- jpdrs, 9, 13, 62-63, 88, 95, 98, 194-195, 197, 211
- jump address, 98
- jump delay, 13
- LISP, 21, 23, 25
- load, *see* memory, read
- load delay, 13, 26-27
- load scheduling, 27
- MD register, 8, 34, 38, 60-61, 63, 71-73, 83, 194-197, 199,

- 212
- MemBus, 34, 36, 40, 50-51, 57, 60, 151, 156, 192
- memory
  - bandwidth, 11, 93
  - caches, *see* instruction cache, external cache
  - coprocessor read, 9-10, 17, 156, 161-162, 166, 177, 179, 184
  - coprocessor write, 9-10, 17, 161-162, 165-166, 177, 179, 184
  - critical path, 18, 161-162, 168
  - data registers, 49-50, 57, 60, 120-121, 176
  - FP read, 165
  - instruction, 9, 34, 37-38, 63, 76, 82, 177, 189, 207, 210
  - interface, 167
  - organization, 10-11
  - protection, 10-11, 14-15, 64, 88-89, 92, 106, 108, 176, 194, 197
  - read, 8-13, 18, 28-29, 38, 49-50, 60, 165-166, 168, 170-172, 174-176, 180, 188-190, 221
  - read signal, 165, 168, 170, 191, 215
  - signal, 63, 82-83, 157
  - synchronization, 175
  - virtual, 10
  - write, 8-10, 12, 26, 38, 49-50, 60, 165-166, 168, 172-176, 179, 182, 188
  - write has no dest, 56, 156
- MIPS, 1-4, 17-18, 24, 26, 28, 38
- movtos, 9, 14, 62-63, 75, 83, 85, 95, 98-99, 102, 111, 157, 195, 197, 201-202, 204, 211-212
- multiplication, 8-9, 15, 34, 49, 60, 63, 71, 75-76, 82-83, 194, 199, 208, 210
- overflow, 10
  - trap, 10, 13-15, 38, 60, 63, 85, 89, 196, 201
- pad loading, 167
- Pascal, 3, 21, 25
- PC Bus, 34, 36-37, 39-40, 89, 189
  - in PC Unit, 94, 97-98, 101, 106, 108, 111
  - in Tags Unit, 117, 119-120, 122-124, 127, 132-134
- PC Chain, 8-9, 14-15, 18, 34, 37-38, 94, 97, 109-110, 193
- PC Unit, 8, 33-34, 36, 38-39, 41, 46, 60-62, 64-65, 89, 93-94, 117-118, 132-133, 157, 188-189, 222
- PC-1, 8, 99, 101-102, 111, 194-195, 197, 212
- PC-4, 8, 62, 64, 95, 99, 101-102, 111, 194-195, 197, 212
- pcm1*, *see* PC-1
- pcm4*, *see* PC-4
- pipeline organization, 12
- PLA
  - Execute Unit, 66-67, 70-71, 73, 75-76, 80, 85, 88
  - PC, 94-95, 99, 101-102, 222
- processor status word, *see* PSW
- PSW, 10, 13-15, 38, 60, 212
  - e bit, 13-14, 85, 87-88, 194, 197
  - implementation, 83, 85-86, 88
  - O bit, 15, 85, 196
  - on exception, 13, 18, 83, 85, 193-194
  - on interrupt, 166
  - on trap, 196
  - overflow, 89, *see also* PSW, O bit
  - reset, 63
  - S bit, 14, 64, 97, 99, 102, 111
  - squashing, 61
  - U bit, 14, 64, 98, 106
  - V bit, 14, 196, 201

- Puzzle, 4
- Regfile, *see* Register File
- Register File, 3, 5, 13, 33-34, 36, 38, 41, 49, 60-62, 95, 98, 151, 156-157, 176, 190
- reset, 88, 94, 97, 111-112, 188, 190
  - Ring Counter, 116
  - signal, 38, 63, 85, 117, 123, 165, 167, 192, 215
  - signal in PC Unit, 97, 99, 101-102, 104-105, 112
  - tags, 115, 122
  - vector, 106, 108, 188
- Result Bus, 34, 36-38, 43, 45, 51, 207-208, 221
  - in Execute Unit, 61, 63-66, 69, 71, 73, 75-76, 80, 82-85, 88-89, 92
  - in PC Unit, 94, 98, 108, 111
  - in Regfile, 50, 190
  - in Tags Unit, 117, 122, 133
- Ring Counter, 115-117, 121, 128, 130-132
- self-timed signals, 41, 52-53, 80, 120, 122, 124
- shifter, 8, 18, 34, 37-38, 49, 60-61, 64, 66, 190, 207-208
- signal naming convention, 39
- SQFSM, *see* controllers, squash
- squash signals, 39, 61, 75, 85
  - in PC Unit, 97, 99, 102, 104-105, 112
  - in Regfile, 56
- squashing, *see* instruction squashing
- Src Buses, 34, 36-37, 50-51, 61-62, 70, 82-83
- standard cell logic, 95, 99, 102, 104, 222
- store, *see* memory, write
- store delay, 12, 26, 172, 176, 221
- store elimination, 24
- store scheduling, 26-28
- store signal, 152
- Tags Unit, 3, 5, 33-34, 41, 61, 93-95, 98, 112, 138
- technology, 41
  - fab, 41
- trap, 9, 13-14, 88, 94, 97, 109, 111, 193, 196-197, 211
  - overflow, *see* overflow, trap
  - signals, 63, 85, 98-99, 101-102, 104-105, 157
  - vector, 37, 106
- Vbias, 43, 152, 166, 217, 219
- Weinberger, 80, 82-83, 152