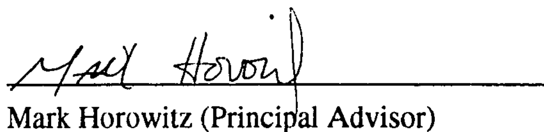
STRiP: A SELF-TIMED RISC PROCESSOR

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Mark Edward Dean
June 1992

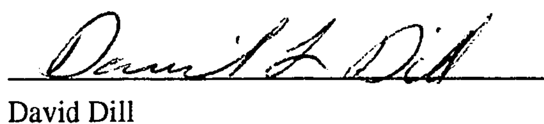
© Copyright by Mark Edward Dean 1992
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



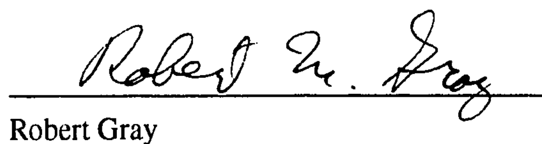
Mark Horowitz (Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



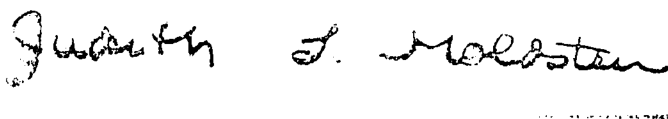
David Dill

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Robert Gray

Approved for the University Committee on Graduate Studies:



STRiP: A Self-Timed RISC Processor

ABSTRACT

Most modern microprocessors and subsystems use a global synchronizing clock to sequence through their operations. Global circuit synchronization simplifies the design and interfacing of the digital logic structures while minimizing the pipeline sequencing overhead. But the worst-case design constraints, based on environment, process, and a single critical logic path, limit a synchronous system's ability to take full advantage of the available silicon performance. Synchronous operation and communication also restrict efficient data transfer between devices of differing processing rates or access methods. Although synchronous logic structures dominate the digital system industry, alternatives must be considered which extract more of the available silicon performance, and provide simple and efficient processing-rate-independent interfaces. Self-timed or asynchronous digital systems and interfaces offer an attractive alternative to synchronous logic structures by eliminating synchronous operating constraints. These systems provide adaptive operation, efficient and flexible interfaces, reduced power consumption, and a wider environmental operating range. Unfortunately, traditional self-timed designs have not been extensively used. Their biggest problems are the complex logic structures and overheads caused by completion detection and handshaking circuits.

STRiP is a self-timed RISC processor architecture that provides the logic simplicity and sequencing efficiency of synchronous structures, along with adaptive operation, efficient asynchronous interfaces, and wide operating range of asynchronous structures. The key concept in STRiP is a self-timed pipeline-sequencing method called dynamic clocking. Dynamic clocking sequences the pipelined functional units in lock-step, but adjusts each sequencing period to match the present environment, process, and pipelined operations. This lock-step operation allows traditional synchronous logic structures to be used. Each cycle's period is determined ahead of time, eliminating self-timed sequencing overheads. To support a fully asynchronous interface, the dynamic-clock generator stops the clock until external dependencies are resolved. Results show that a dynamically clocked RISC processor will operate twice as fast as a equivalent synchronous processor, both built with the same silicon technology.

Acknowledgements

Many people provided support during my studies at Stanford University. I am indebted to my advisors, Mark Horowitz and David Dill, for many important ideas and guidance which led to the development of STRiP and its self-timed sequencing structure. Their support and encouragement gave me the confidence to continue my research even when early ideas did not prove viable. My many conversations with Steve Przybylski were invaluable in helping me understand the trade-offs and constraints in high-performance memory system design and led to the development of predictive prefetching. I would also like to thank Martin Hellman and Teresa Meng for their backing throughout my stay at Stanford.

I owe a special debt of gratitude to IBM, especially James Canavino and Paul Mugge, which have supported my research, provided the resources to complete my Ph.D. work, and have given me the rare opportunity to gain insight into the workings of processor and system technologies. Their confidence in my work made my return to an academic environment pleasant and rewarding.

I am especially grateful to my parents, James and Barbara Dean, my grandparents, Eugene and Ophelia Peck, and my great-grandmother, Leola Bassett. Their experiences, hard work, inspiration, and love motivated my goals and made my career accomplishments possible. Above all though, I am eternally grateful to my best friend and wife, Paula Bacon, who has so dramatically changed the course of my life. She has patiently supported my many busy evenings and weekends while I completed my academic work. I hope I can provide her with the love and happiness she has unselfishly given me.

Table of Contents

Chapter 1: Introduction **1**

1.1 Processor Design 1
1.2 STRiP Principles 3
1.3 Thesis Organization 4

Chapter 2: High Performance Processor Design **5**

2.1 Elements of Performance 5
2.2 Sequencing Methods 8
2.2.1 Synchronous Systems 9
2.2.2 Asynchronous Systems 15
2.2.3 Self-Timed Synchronous Systems 21
2.3 Memory System Design 21
2.3.1 Caching 22
2.3.2 Fetch and Prefetch Strategies 24
2.3.3 Predictive Prefetching 25
2.4 Summary 27

Chapter 3: Dynamic Clocking **29**

3.1 Basic Structure 29
3.2 Constraints and Tradeoffs 32
3.2.1 Tracking Cells 32
3.2.2 C-element 34
3.2.3 Pulse Generator 34
3.2.4 Clock Control and Distribution 35
3.2.5 Functional Unit Design 37
3.2.6 Clock Buffers 38
3.3 A Self-Timed MIPS-X 38
3.3.1 Tracking Cell Selection 39
3.3.2 Synchronous vs Self-Timed 43

Chapter 4: Improving Memory System Performance....47

4.1 Modern Memory System Design....47
4.2 Prefetching....50
4.2.1 Predictive Prefetching....52
4.2.2 Predictive Prefetching Protocol....53
4.2.3 Analysis Assumptions....57
4.2.4 Analysis Technique....60
4.2.5 Simulation Results....62
4.3 Support Hardware....71
4.3.1 Zero-Level Caches....71
4.3.2 First-Level Caches....75
4.3.3 Access Time Evaluation....79
4.4 Summary....80

Chapter 5: STRiP Implementation....82

5.1 Basic Structure....82
5.1.1 Critical Functional Units....83
5.1.2 Datapath and Floor Plan....88
5.2 Tracking Cell Designs....91
5.2.1 Adder Tracking Cell....92
5.2.2 Branch Tracking Cell....94
5.2.3 First-Level Cache Tracking Cell....96
5.2.4 The External Interface Tracking Cell....98
5.3 The KNOB....101
5.3.1 Implementation Alternatives....101
5.3.2 KNOB Design and Analysis....105
5.4 The C-element....108
5.4.1 C-element Constraints....108
5.4.2 C-element Designs....110
5.4.3 Performance Analysis....112
5.5 The External Interface....114
5.5.1 Communication Protocol....115
5.6 Exception Handling....118
5.6.1 External Interrupts....119
5.6.2 Internal Exceptions....120
5.7 Performance Analysis....120
5.8 Summary....124

Chapter 6: Conclusions....126

Appendix A: Memory System Terminology **129**

Appendix B: Signal Naming Convention **133**

Appendix C: SPICE Parameters **135**

Appendix D: STRiP's Instruction Set **138**

D.1 Instruction Set Architecture 138
D.2 Processor Status Word 142

Appendix E: C-element Logic Diagrams **144**

Appendix F: External Interface Signal Descriptions **148**

Bibliography **156**

List of Tables

Table 2.1: Operating Condition Categories for Military and Commercial CMOS devices. 12
Table 3.1: List of operations requiring an add operation 42
Table 3.2: Frequency-of-use and processing latencies for the MIPS-X processor critical logic paths. 45
Table 4.1: PIA storage based on reference types and their ordering......55
Table 4.2: Logic elements in zero-level cache reference path......80
Table 4.3: Functional Unit gate-delays for zero-level cache reference path. 80
Table 5.1: CMOS adder performance comparison......85
Table 5.2: STRiP's critical path latencies and their frequency-of-use. 93
Table 5.3: C-element State-Table......109
Table 5.4: C-element propagation delay from A to Q with a fanout of 16. 113
Table 5.5: C-element propagation delay from ABCD switched simultaneously to Q with a fanout of 16. Technology = 2.0um CMOS (MOSIS)......113
Table 5.6: C-element propagation delay from D to Q with a fanout of 16. 114
Table 5.7: Performance parameters measured during SPICE simulation of dynamic clocking structure......122
Table 5.8: Estimated performance improvement, over MIPS-X, attributed to dynamic clocking and the use of high speed functional units. 122
Table D.1: STRiP (MIPS-X) Memory and Branch instructions......139
Table D.2: STRiP (MIPS-X) Compute instructions......140
Table F.1: Encoding of least significant address bits to support word and quad-word transfers......149
Table F.2: Processor signal encoding during external data transfers......151
Table F.3: Signal encoding for response to internal cache snoop operation......153

List of Illustrations

Figure 2.1: Timing template for a general five-stage RISC pipeline. 6
Figure 2.2: Block diagram of general RISC pipeline. 10
Figure 2.3: (a) Single-phase and (b) multi-phase clock waveforms and pipelines. 11
Figure 2.4: Propagation delay as a function of temperature, voltage, and process....13
Figure 2.5: Example data stream showing complexities of dual-rail encoding schemes....18
Figure 2.6: Self-timed dual-rail full-adders using (a) PLA-like and (b) DCVSL structures....19
Figure 2.7: Traditional two-input dual-rail completion detector. 19
Figure 2.8: A traditional asynchronous pipeline structure....20
Figure 2.9: Memory Hierarchy with Zero-Level Caches 26
Figure 3.1: General dynamic clocking structure. 31
Figure 3.2: Flow-chart describing the sequence of operations required to support dynamic clock generation....33
Figure 3.3: Pulse Generator block diagram. 35
Figure 3.4: Pulse Generator transistor level diagram. 36
Figure 3.5: Local qualified clock buffers: (a) single-phase and (b) two-phase non-overlapping clock generation. 37
Figure 3.6: Block diagram of MIPS-X with dynamic clocking and internal data cache....40
Figure 4.1: Memory Hierarchy with Zero-Level Caches 49
Figure 4.2: First-Level cache tag structure to support predictive prefetching....56
Figure 4.3: Zero-Level cache tag structure to support predictive prefetching....57
Figure 4.4: Baseline Design 58
Figure 4.5: Block size versus miss penalty and miss rate [37]. 59
Figure 4.6: (a) Miss rate and (b) average access time versus block size [37]....59
Figure 4.7: Unique references as a function of time: R2000 traces [78]....61
Figure 4.8: Zero-level instruction cache miss ratios. 63
Figure 4.9: Instruction reference average access times. 64
Figure 4.10: Factor-for-Unutilized-Prefetches (FUP) for zero-level instruction caches....65
Figure 4.11: Zero-level data cache miss ratios....67
Figure 4.12: Zero-level data cache average access time....68
Figure 4.13: Factor-for-Unutilized-Prefetches (FUP) for zero-level data caches....69
Figure 4.14: Memory system relative CPI using zero-level caches. 70
Figure 4.15: Zero-Level Instruction Cache block diagram....72
Figure 4.16: Zero-level cache tag and valid-bit CAM cell....73

Figure 4.17: Zero-level cache tag tracking cell. 74
Figure 4.18: Zero-level cache data RAM cell. 74
Figure 4.19: Zero-level cache bit-line precharge tracking cell. 75
Figure 4.20: First-level cache block diagram. 76
Figure 4.21: First-level cache prefetch address RAM array. 77
Figure 4.22: Timing diagram for prefetch address tag RAM. 78
Figure 5.1: STRiP Block Diagram 84
Figure 5.2: (a) Register File floorplan and (b) Register Array memory cell. 87
Figure 5.3: State diagram for internal cache miss FSM. 88
Figure 5.4: STRiP's datapath diagram. 89
Figure 5.5: STRiP floor plan. 90
Figure 5.6: Example critical logic path and equivalent tracking cell circuit. 91
Figure 5.7: Gate level schematic of adder tracking cell. 94
Figure 5.8: Branch tracking cell gate-level diagram. 95
Figure 5.9: First-Level Cache floor plan. 97
Figure 5.10: First-level Cache tracking cell gate-level diagram. 99
Figure 5.11: External interface tracking cell gate-level diagram. 99
Figure 5.12: Timing diagram showing external tracking cell signals during read cycle. 100
Figure 5.13: Adjustable delay elements using (a) variable supply voltage and (b) selectable inverter chains. 104
Figure 5.14: Adjustable delay inverter using current-starved delay element design style 104
Figure 5.15: Adjustable delay logic gate using variable capacitive loading on each logic gate 104
Figure 5.16: Delay versus delay element control voltage assuming worst-case conditions and a 0.8um CMOS process [51]. 105
Figure 5.17: KNOB varitability on (a) tracking cell cycle times and (b) pulse generator output pulse 107
Figure 5.18: 4-input pseudo-dynamic C-element logic diagram. 111
Figure 5.19: Processor complex block diagram. 115
Figure 5.20: Timing diagram showing generic, external read and write cycle signalling. 116
Figure 5.21: Dynamic clocking structure used during SPICE simulation. 121
Figure 5.22: SPICE plot of clock generator output and select feedback delay. 124
Figure D.1: The STRiP (MIPS-X) instruction formats. 139
Figure D.2: The Processor Status Word (PSW). 143
Figure E.1: 4-input C-element implementation using cross-coupled NORs. 144
Figure E.2: 4-input C-element implementations using majority function circuits. 145
Figure E.3: 4-input C-element using pseudo-NMOS logic structure. 145
Figure E.4: 4-input dynamic C-element. 146
Figure E.5: 4-input dynamic C-element tree using 2-input dynamic C-elements. 146
Figure E.6: 4-input dynamic C-element with charge-sharing reduction circuitry. 147

Chapter 1

Introduction

The ever-increasing processing power required to handle today's commercial applications dictates the design of computer systems that fully exploit the capabilities offered by contemporary VLSI technologies. Many architectures and logic structures exist which help to minimize the execution times of these applications. But even with the most advance architectural methods, there is significantly more performance available within existing silicon technologies. System and processor performance is lost because of the worst-case design constraints required for synchronous operation. We will show that self-timed sequencing of a processor's pipeline, along with an enhanced memory system and asynchronous external interface, significantly improve the operational efficiency of the processor complex.

1.1 Processor Design

Optimizing a processor's execution rate has become a key component in maximizing a computer system's performance. A processor's performance is often measured as the time required to execute a set of instructions. The following formula for CPU time gives the key elements controlling processor performance:

CPU time = Clock cycle time * CPI * Instruction Count

The three elements determining processor performance are: the period of each processor clock cycle, the number of cycles-per-instruction (CPI), and the number of instructions required to process the targeted information. Modern architectures attempt to reduce the cycle time of the processor, minimize its CPI, and limit the number of instructions required to run an application.

The most significant advancement of recent years which is used by most modern processor architectures is *pipelining*. Pipelining divides each instruction into independent operations, allowing parallel execution with other instructions. By permitting parallel execution of multiple instructions, pipelining utilizes an application's available instruction-level parallelism, minimizing the total cycles required to execute that application. Superscalar and superpipelined architectures further utilize instruction-level parallelism, providing the best compromise between cycle time and CPI. Studies have determined that superscalar and superpipelined architectures provide approximately the same level of performance; 30-40% more than a basic scalar architecture [50, 54, 96]. But independent of their architecture, all modern processors are constrained by the same sequencing paradigm; synchronous operation.

Synchronous processor design and operation are based on a global synchronizing clock. External data transfers are based on the same clock. Synchronous operation simplifies the processor's design by eliminating the need for hazard-free logic structures, which greatly reduces the size and complexity of the combinational logic. Unfortunately, synchronous operation also restricts the processor to a worst-case operating frequency in order to guarantee operation under all possible conditions. Since the clock frequency for a synchronous processor is constant, its execution rate is independent of the operating conditions. But the actual speed of the processor varies with the environmental conditions (voltage and temperature), silicon process, and pipeline operations. For reliable operation, the clock frequency must be slower than the operating rate required under worst-case conditions. This results in a reduced utilization of the technologies performance when the processor is operating under typical conditions. Therefore, synchronous operation provides ease of implementation while restricting the ability of the processor to fully utilize the raw performance of the silicon technologies.

In an attempt to eliminate synchronous operating constraints, numerous asynchronous or self-timed logic structures and design styles have been studied and implemented [18, 21, 22, 23, 24, 25, 28, 45, 46, 59, 63, 68, 70, 72, 74, 87, 99, 104, 108]. A few researchers have applied these techniques to processor design [23, 45, 58a, 63, 68]. Self-timed processor design is a discipline of digital design in which the sequencing control of

the processor's pipeline is distributed over the elements which compose that pipeline. In contrast, a synchronous processor operates based on a central global clock; all pipeline elements processing in lock-step. Self-timed digital systems provide several advantages: operation which tracks the processor's environment (voltage and temperature) and process, external asynchronous interfaces that are adaptive and efficient, and a wider and more reliable operating range. In theory, a self-timed processor provides the system with all its available silicon performance and allows efficient data transfer independent of the individual operating rates of the communicating devices.

Realizing these theoretical advantages is not necessarily easy, however. Since they require variable encodings to support completion detection, self-timed systems typically require complex logic structures. Also, the completion detection and communication overhead required during sequencing reduce the efficiency of the pipeline. Because of these constraints, no self-timed processor implementation has been commercialized. But, our research shows that by utilizing the operational characteristics of a processor pipeline structure, self-timed pipeline operation can be greatly simplified. By using a synchronous structure with a self-adjusting clock, the complexity and overhead of asynchronous structures are avoided while providing adaptive operation, a wide operating range, and self-timed external interfaces.

1.2 STRiP Principles

STRiP is a self-timed RISC processor implementation. Their are two main elements which made the STRiP implementation possible. The first is the use of a self-timed synchronous-pipeline sequencing structure called *dynamic clocking*. A dynamically clocked processor uses a synchronous pipeline structure which is sequenced by an self-adjusting global clock. The dynamic-clock generator is self-contained on the processor chip allowing it to track changes in temperature, voltage, and process. It also receives feedback information from the pipeline permitting it to adjust the clocks period to the requirements of the pending pipeline operations. To support a fully asynchronous external interface the dynamic-clock generator stops the clock until external dependencies are resolved. Dynamic clocking eliminates the complexity and overheads common to traditional self-timed structures. It also provides adaptive pipeline sequencing and an efficient external interface which interconnects devices independent of their individual operating rates. Our research shows that a dynamically clocked processor can

extract twice as much performance from existing silicon technologies than is possible through traditional synchronous approaches.

To optimize the performance of a self-timed pipeline, the memory system access time must be removed from the critical logic path. The most common solution to this problem is to pipeline the first-level caches. This decreases the effective access time of the caches but increases the pipeline depth, resulting in increased branch and load penalties. A better approach is to add small, low-latency caches, called *zero-level caches*, between the CPU and the first-level caches. To reduce the miss rate normally associated with small caches (less than 256 bytes), an adaptive prefetching method called *predictive prefetching* was developed. Predictive prefetching uses a history of references to predict future data and instruction references. The combination of zero-level caching and predictive prefetching reduced the memory system latency by half, removing the memory system from the critical logic paths of the processor.

1.3 Thesis Organization

To understand the performance bottlenecks in current processors, one must first understand the basics of high performance processor design. Chapter 2 reviews the structural elements which determine performance levels in modern processors. It also describes two elements of pipeline operation which were the basis of our research at Stanford: pipeline sequencing and memory system design. The next two chapters give detailed descriptions of specific logic structures developed to improve pipeline sequencing and memory system performance. Chapter 5 provides detailed designs of basic structures required for a efficient STRiP implementation and gives processor performance measurements which are then compared with performance parameters of an equivalent synchronous design. Finally, the thesis concludes with possible enhancements to the proposed self-timed organization.

Chapter 2

High Performance Processor Design

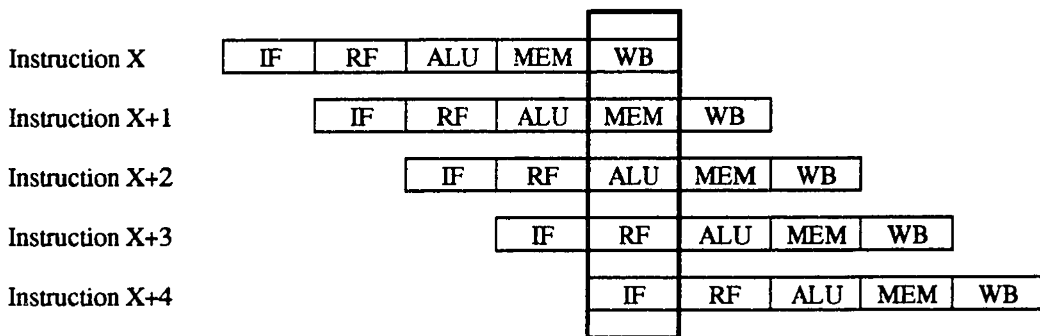
The demand for high performance computer systems has spawned thousands of research papers, technical books, architectures, and actual implementations in the field of high-performance processor design. This chapter provides the background for understanding the architectural and structural improvements in processor design proposed in Chapter 3 and 4. The chapter will focus on two dominant aspects of processor design: *pipeline sequencing* and *memory system design*. Both areas significantly affect and often limit the performance of contemporary processors. The chapter presents design and processing constraints for both domains of processor design, allowing the reader to understand the proposed logic structures and how they improve performance within existing silicon technologies.

2.1 Elements of Performance

Processor performance is measured in many different ways. Basically, the computer that performs the same amount of work in the least time is the fastest. We will use the processing time of the *central-processing unit* (CPU) as the processor performance measure (excluding waiting for I/O responses). As discussed in Chapter 1 we express processor performance by the following formula:

$$\text{CPU time} = \text{Clock cycle time} * \text{CPI} * \text{Instruction Count}$$

This formula illustrates how processor performance is dependent upon three major elements. One element, clock cycle time, is often divided into two components: the number of gates in the critical logic path and the gate delay of the silicon technology. The methods involved in improving the characteristics of each performance component are interdependent, making it difficult to isolate and improve one characteristic without degrading the other characteristics. Instruction count is constrained by the instruction set architecture (CISC, RISC, VLIW) and the compiler technology. The processor's CPI is controlled by the processor's hardware organization, but is also affected by the instruction set architecture. The silicon technology, as well as hardware organization, determines the cycle time, or sequencing rate, of the processor. By increasing functionality of instructions, CISC and VLIW architectures decrease the instruction count, but increase the CPI and cycle time. RISC architectures reduce the CPI, and typically the cycle time, but increase the instruction count by simplifying the instruction set. Advances in VLSI technology have allowed significant decreases in cycle times, providing speedups regardless of the processor architecture. However, the hardware organization used affects the utilization of the available silicon performance.



Operations:
- IF - Instruction Fetch
- RF - Register Fetch and Instruction Decode
- ALU - ALU Operation
- MEM - Data Memory Access
- WB - Register Write-back

Figure 2.1: Timing template for a general five-stage RISC pipeline.

Most modern processors use a hardware organization called *pipelining* [37, 58] which increases performance by overlapping the execution steps of different instructions. To pipeline instructions the various steps of instruction execution are performed by independent functional units or *pipe stages*. To execute an instruction, the processor passes it from one pipe stage to the next until all of the required operations have been performed. The pipe stages are separated by registers or latches. At any given moment there are several instructions in progress, each occupying a different pipe stage. Figure 2.1 illustrates how each instruction overlaps the operations of other instructions in a typical RISC pipeline. Each line segment represents one pipeline cycle. Pipelining reduces the average number of cycles required to execute an instruction without appreciably increasing the processor cycle time. Compared to a non-pipelined design, pipelining results in a significant improvement in processor performance.

Other hardware organizations attempt to increase performance by further exploiting instruction-level parallelism. The MIPS R4000 [57] uses *superpipelining* to segment the pipe stages into smaller and faster units, increasing the number of pipe stages and reducing the processor's cycle time. But superpipelining increases the cycle penalties caused by branch and load hazards, resulting in a higher CPI.

Superscalar processors like the RS/6000 [33, 34, 35, 38] and i860 [44] attempt to exploit fine-grain instruction parallelism by utilizing multiple, independently-pipelined functional units, allowing multiple instructions to be issued per cycle. Even though a superscalar architecture provides a significant potential for high processing rates, there are key problems in the exploitation of low-level parallelism by a superscalar organization: detecting data and control dependencies, resolving these dependencies, and scheduling the order of instruction execution. Static and dynamic code scheduling algorithms are used to optimize the available parallelism and resolve these dependencies. Although superscalar processors reduce the effective CPI, the complexity of instruction scheduling and register bypassing, and the increase in signal loading increase the processor's cycle time.

Jouppi [54], Johnson [50], and Smith [96] showed that there is a limited amount of instruction-level parallelism within existing applications. These studies showed a CPI improvement of 30%-40% for a dual-issue superscalar processor with in-order issue and out-of-order completion. Jouppi also showed that superpipelining and superscalar techniques are roughly equivalent ways of exploiting instruction-level parallelism. Smith's research suggests that if instruction scheduling is restricted to a single basic block, most simple pipeline machines exploit a significant amount of the instruction-level

parallelism, even without parallel instruction issue or higher degrees of pipelining. These results suggest that other hardware organizations must be developed to increase the processor performance without significantly increasing its complexity.

Our research has concentrated on improving processor performance by reducing its cycle time without effecting the instruction count or CPI. One approach to reducing the cycle time is to increase the pipeline sequencing efficiency. Section 2.2 describes the constraints of existing processor sequencing methods and how they limit the performance extracted from the silicon technologies. Another way of reducing the cycle time of the processor is to improve the performance of the constraining functional elements. In many computer systems the memory subsystem performance dramatically affects, and in many cases limits, the performance of advanced microprocessors. Section 2.3 provides a brief study of existing memory systems and how they affect the processor performance. Each section proposes a method of improving processor performance without significantly changing the pipeline organization of the processor. Chapter 3 and 4 then provide a detailed understanding of the proposed structural enhancements.

2.2 Sequencing Methods

Instruction execution through a pipeline of functional units requires the processor and system design to adhere to a strict pipeline sequencing method. A sequencing method, or discipline, is characterized by the way it connects sequence and time. This section describes two pipeline sequencing methods: *synchronous* and *asynchronous* or *self-timed.* Synchronous systems are the best known and most widely used, mainly because of their design simplicity. These systems use a single signal (or set of signals) called clocks to relate sequence to time. But synchronous systems pose some serious limitations, which are made even worse as silicon technologies scale down and chips become larger. Self-timed systems provide a viable alternative to the traditional sequencing method by connecting sequence and time in the interior of each logic element rather than through a global clock. Connected properly, self-timed elements provide correct sequential operation which is insensitive to element and wire delays. But the design complexities and the sequencing overheads of self-timed systems have limited their use. This section describes the design constraints of synchronous and self-timed systems and proposes an alternative sequencing structure which combines the advantages of both.

2.2.1 Synchronous Systems

Today's processors, both public domain and proprietary, are dominated by a synchronous design philosophy. In the synchronous design discipline the processor depends on a global synchronizing signal or *clock* to connect the properties of sequence and time [84]. The *transitions* of the clock serve to define the instants at which processor state changes may occur. The *period* of the clock serves as a time reference for the latency and wire delay of the connected clocked elements. Difficulties relative to the designing, optimizing, modifying, and reliably operating synchronous processors arise from binding the sequencing and timing constraints of the processor's functional elements.

Most processors are constructed using a set of functional units (ALU, decoder, register file, etc.) connected via a pipeline structure. Figure 2.2 gives a general block diagram of a synchronous RISC processor pipeline structure. All functional-unit operations are initiated by the clock transitions and their latencies or access times must be less than one or multiple clock periods. To provide a finer time resolution, the clock can be split into phases. Typically the phases are non-overlapping to remove the need of supporting *two-sided timing constraints* [31, 37, 64]. Two-sided timing constraints refer to the minimum and maximum delay requirements imposed on the functional units by a single-phase clocking structure. A multi-phase, non-overlapping discipline removes all minimum delay requirements. Figure 2.3 shows how clocks are connected in a single-phase and multi-phase pipeline and the combinational logic delay constraints. But the use of multi-phase, non-overlapping clock structures becomes more difficult as clock frequencies and chip size increase because of the relative increase in *clock skew* [46].

The pipeline's functional units are combinational logic blocks. They are separated by one or two storage elements, depending on their processing latency and the synchronization point required between adjacent functional units. Sietz [84] gives three important consequences of a synchronous pipeline structure on the functional units' logical design: it assures deterministic behavior if the physical aspects of the design are also correct, it relieves the designer of any requirement that the combinational logic be free of transients (static or dynamic hazards) on it outputs, and it ensures that the storage- or history-dependence of the system resides entirely within the clocked storage elements, simplifying the design process, maintenance, and testing of the system. Synchronous-processor pipeline designs have been extensively researched and evaluated, and numerous references exist which detail their implementation [1, 5, 26, 31, 36, 42, 43, 44, 47, 52, 55, 56, 67, 71, 76, 77, 85, 98, 100, 102, 106].

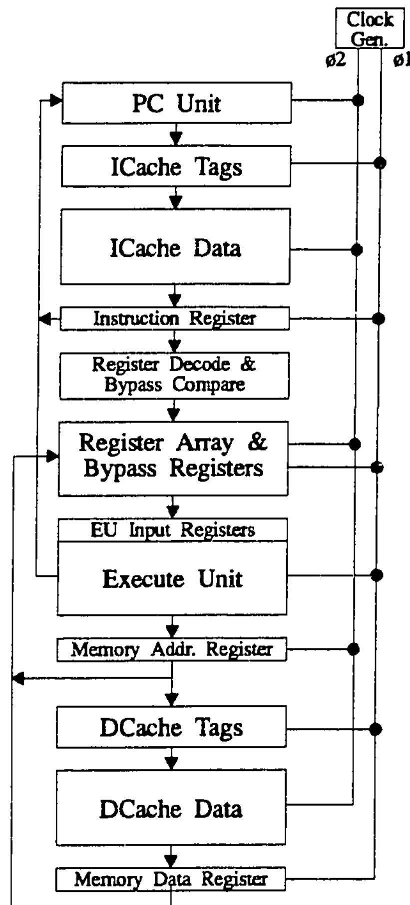


Figure 2.2: Block diagram of general RISC pipeline.

The synchronous design characteristics listed in the previous paragraph are the main reasons synchronous digital systems dominate the digital design industry. Combinational logic used in a synchronous data path or control unit (i.e. finite-state machine) need not be free of logic hazards, simplifying its implementation. Tools have been developed to aid the design of logic for synchronous machines. Also, the period of the synchronous system's global clock can be statically altered to match the processing requirements of the functional units. If a unit is slower than expected, the system will still function at a slower clock rate. The global clock also guarantees a constant operating rate. The computation time of a synchronous system for a given operation will always be the same and will always yield the same result. These attributes have allowed synchronous

processor design to be the simplest, most area efficient, and most popular processor implementation structure.

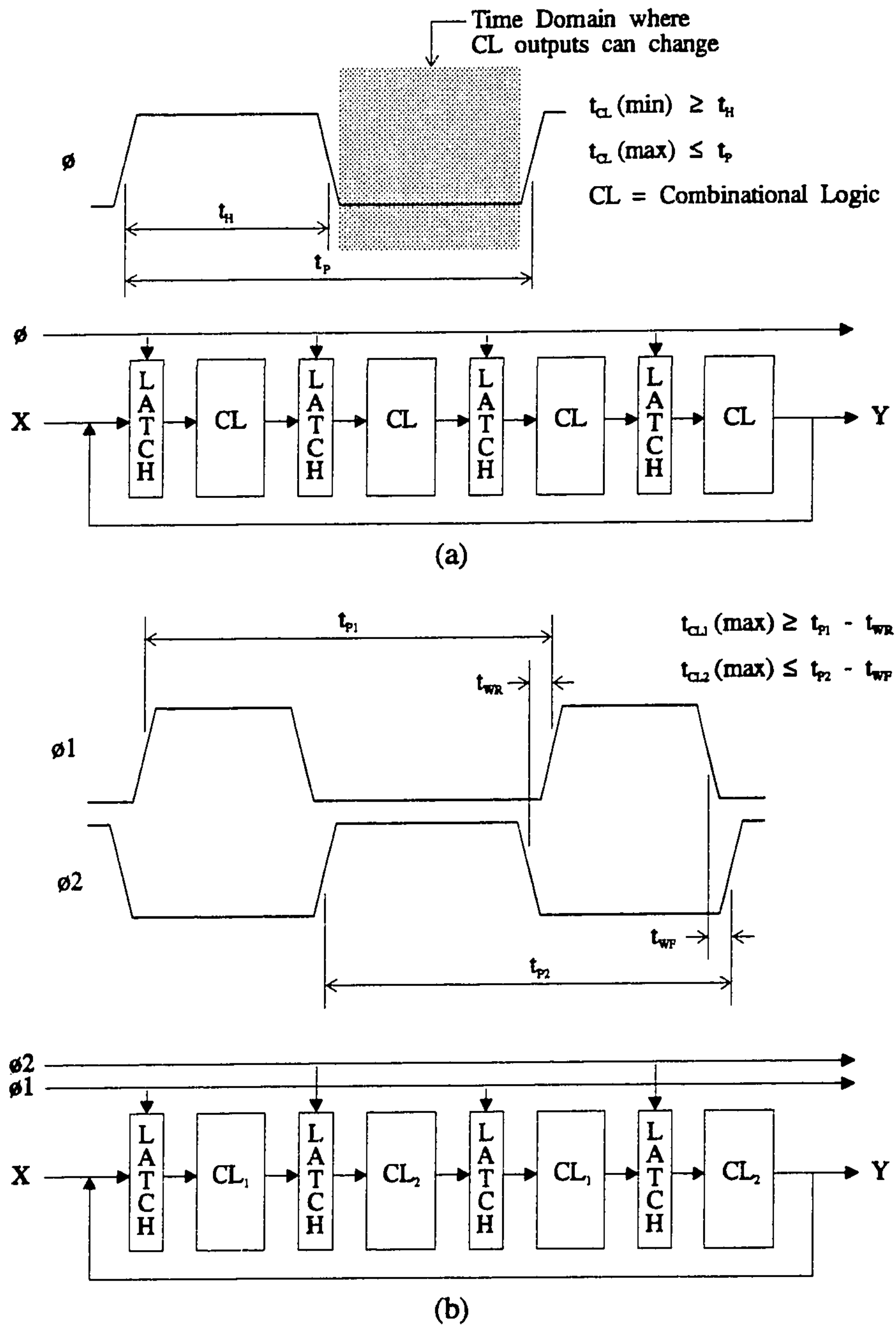


Figure 2.3: (a) Single-phase and (b) multi-phase clock waveforms and pipelines.

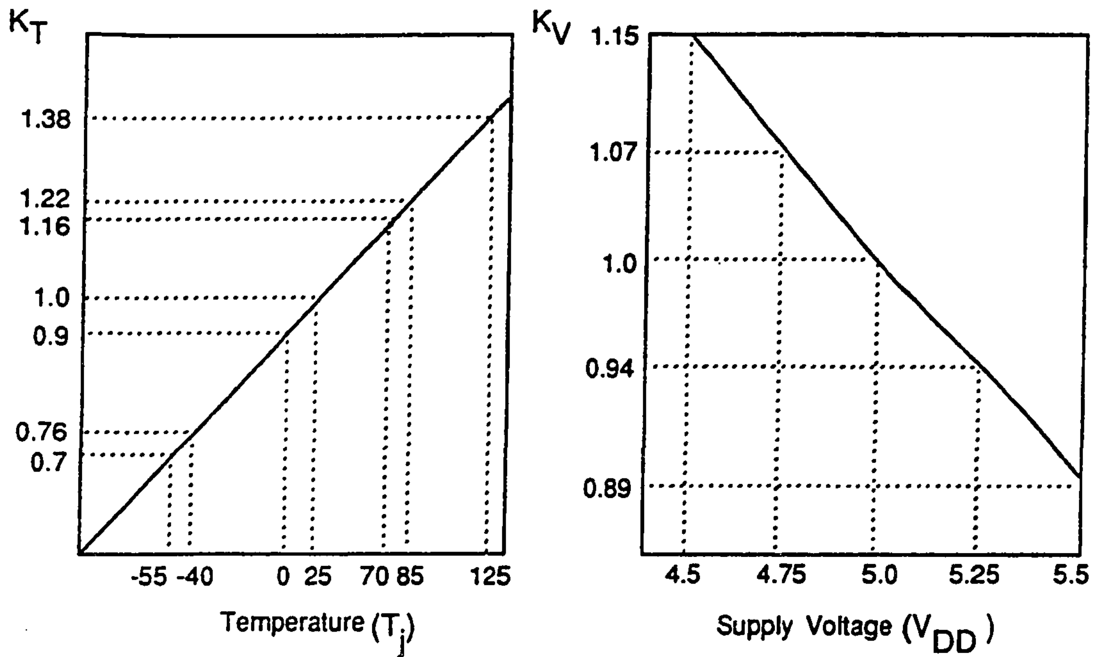
DESIGN CONSTRAINTS

Although synchronous processor designs have proven to be effective, testable, and usually scalable, they have yet to take full advantage of the base technology's available performance. Present CMOS process technologies allow commercial microprocessors and systems to operate at external clock rates of up to 66MHz (Intel i860 and HP PA). That same CMOS process can produce significantly more performance if synchronous design constraints were not required. Since the clock period is *static*, the operating frequency of the processor must be set assuming worst-case operating conditions. The clocking structure is unable to adapt to the system's operating environment. Thus, the main performance limiting constraint of a synchronous processor design is that the processor must operate assuming worst-case environmental conditions (temperature and supply voltage) and worst-case process. Most CMOS component data books give signal latencies and functional operating frequencies for best case, nominal, and worst-case operating conditions. Table 2.1 gives the conditions which define these three categories. Figure 2.4 illustrates the dependence of VLSI circuit speeds on process and environmental parameters.

Environmental Parameters	**Environmental Conditions**			**Units**
	Best Case	Nominal	Worst-Case	
Commercial:				
Supply Voltage, V_{CC}	5.25	5.00	4.75	Volts
Operating free-air Temperature, T_A	0	25	70	°C
Military:				
Supply Voltage, V_{CC}	5.50	5.00	4.50	Volts
Operating free-air Temperature, T_A	-55	25	125	°C

Table 2.1: Operating Condition Categories for Military and Commercial CMOS devices.

Manufacturers attempt to take advantage of process variations by sorting their VLSI components into groups, or *bin-splits*, which defines the components operating frequency under worst-case conditions. These bin-splits result from frequency deterministic test



$$t_{wc} = K_P * K_T * K_V * t_{nom} = K_{wc} * t_{nom}$$

t_{wc} = worst-case processing latency or the synchronous pipeline clock period

t_{nom} = nominal processing latency or the optimal average sequencing time

K_T = worst-case temperature degradation factor

K_V = worst-case supply voltage degradation factor

K_P = worst-case process degradation factor = 1.5

K_{wc} = 1.862 for commercial voltage and temperature ranges

Figure 2.4: Propagation delay as a function of temperature, voltage, and process (LSI 0.7um CMOS process, April 1990).

performed on each VLSI component during the manufacturing process. For example, Intel sorts its i860 processors, and support components, into three frequency groups; 50MHz, 40MHz, and 33MHz. The percent performance increase between adjacent frequency groups is 25% and 21%, respectively. Since the frequency test are discrete

and start at the highest discrete operating frequency and work down, the actual worst-case operating frequency of each component is unknown. A component failing the 50MHz test, but passing the 40MHz test, may actual operated reliably at 45MHz, a loss in potential operating performance of 12.5%. Most system designers are aware of this performance loss but are unable to safely take advantage of any available component performance above the specified limit.

Another design constraint of synchronous processor implementation is how the critical logic paths among the pipelined functional units limit the sequencing efficiency of the structure. The sequencing of a pipeline structure is typically limited by one critical logic path. In many cases that critical logic path's frequency-of-use is low. For example, in the MIPS-X processor developed at Stanford University [17, 42], the most critical pipeline logic path was the Compare-and-Branch operation. But this operation was only required approximately 15% of the time [37, 75]. The next most critical logic path, the Add operation, required approximately six fewer gate delays or 12.5% less logic delay than the most critical pipeline path. This results in a 10% lost in performance (0.85 x .125) over a optimally clocked or asynchronous structure. Therefore, the synchronous processor is typically operating at a rate slower than that required for most logical operations.

While the wire delay for the interconnection of local devices is still insignificant with respect to circuit speeds, a wire that traverses an entire integrated circuit (IC) can have appreciable delay. This fact decreases the efficiency of a global synchronizing signal or clock since it must be distributed over the entire chip area. The global-clock signal wire delays cause clock skews which must be compensated for by reducing the functional unit latencies or increasing the time between phases of a multi-phase clocking scheme. Since critical path functional units are optimized to begin with, the clock period must be lengthened to support any clock skew in the global clock signal.

One way to avoid clock skew and infrequently used critical path dependencies is to divide the processor into processing units, each independently sequenced by a local clock. This results in a locally synchronous, globally asynchronous processor structure [16]. The problem with this approach is the difficulty of synchronizing the data transfer between two processing units. Since the synchronization overhead can be significant, care must be taken in partitioning the functional unit into processing units. The synchronization process must also avoid prolonged metastable conditions which would cause unreliable processor operation. Because of the synchronization overhead and potential for metastable conditions, most designers partition the processing units so that

their functional delays are some multiple of a global clock period. This approach, while not optimum, removes the problem of synchronization.

A significant problem with synchronous structures at the system level is the requirement that elements constructed using a variety of technologies and architectures must operate at a common clock rate or at an integer division of that clock rate. This implies, for example, that a RISC processor constructed using a 0.8um CMOS process must operate synchronously with a FPU constructed using a 1.2um BiCMOS process and a DMA controller constructed using a 1.6um CMOS process. The optimal operating frequency of each of these components may well be different. The problem is compounded by the fact that many synchronous system elements are inherently asynchronous. Examples of these elements include caches built using discrete SRAMs, system memory built using DRAMs, and channel interface units or switches which connect to mostly asynchronous buses (FutureBus, VME, Micro Channel, EISA, ISA, etc.). This mix of technologies, structures, and interfaces causes a designer to trade-off efficient individual element operation for synchronous operation among all elements. In most cases, without synchronous operation among connected devices, the communication overhead between devices is unacceptable and metastable conditions are unavoidable.

The synchronous digital system constraints discussed limit a designer's ability to extract the available performance potential of most silicon technologies. The next section discusses alternatives to a synchronous system structure that can extract more performance from present silicon technologies.

2.2.2 Asynchronous Systems

In the literature, the terms synchronous, asynchronous, and self-timed can refer to different things in different contexts. As defined in the previous sections, a synchronous system or processor transfers information between communicating logic elements in lock-step with a global clock signal. An *asynchronous system* or *processor* is one where the information transfer between communicating logic elements is NOT performed in synchrony with a global clock signal, but is performed at times determined by the latencies of the communicating logic elements. Asynchronous systems are constructed with self-timed logic elements. Therefore, an asynchronous system or processor will also be referred to as "self-timed".

The temporal control of a self-timed system is delegated to the communicating logic elements used to construct the system. Time and sequence are related inside these logic

elements, such that signal transitions at their terminals must occur in predefined order. Signal transitions at a self-timed element's input initiate processing, while output signal events indicate completion of processing. The time required to perform a computation is determined by the delay between initiation and completion, and by interconnection delays. To provide proper operation, communicating self-timed logic elements are connected through a closed control-signal path, usually containing a *request* and *acknowledge* signal. A self-timed element cannot request or accept new input data until the connected logic elements or storage devices acknowledge receiving the result of its previous computation. This communication protocol allows each self-timed element's operational characteristics to be isolated from the characteristics of the other system elements. An asynchronous interface can remove many timing constraints from the system level design.

ADVANTAGES

An asynchronous digital system can avoid many of the constraints associated with a synchronous digital system. The first advantage of an asynchronous digital system is its ability to adapt its operation rate to the silicon process parameters (resulting from the VLSI manufacturing process) and environmental operating conditions (temperature and supply voltage). Therefore, an asynchronous processor built from a CMOS technology will execute faster at 5.5 volts and 0°C than at 4.5 volts and 70°C. In general, this allows a self-timed design to take full advantage of the available silicon performance. Also this adaptability allows self-timed circuits to operate over a wider environmental range. The reliable operating range of an asynchronous system is limited only by the silicon's physical resiliency to damage, and not a range dictated by a global clock. These factors allow a single asynchronous system to support both commercial and military environments while providing the best possible performance for any given operating environment.

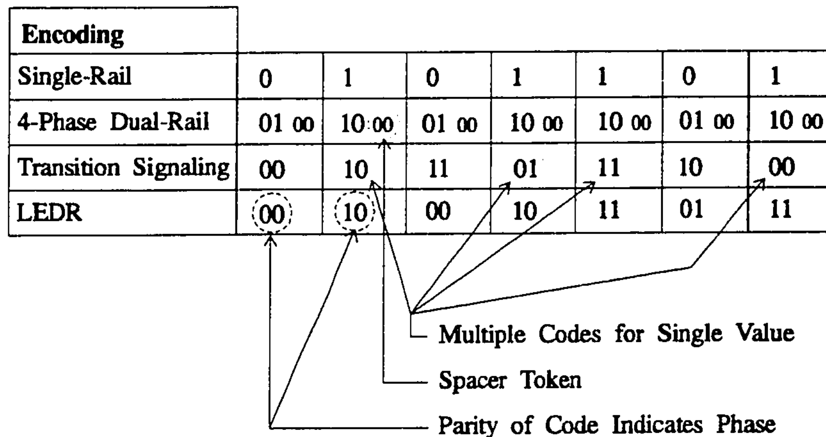
If all the elements in a system or processing complex are self-timed, the system designer will be able to interface logic elements through an asynchronous communication protocol. This communication protocol allows each self-timed element's operational characteristics to be isolated from the characteristics of the other system elements. As a result, processor complex chip-sets will not be rendered incompatible or inefficient whenever the processors silicon technology improves. The asynchronous logic systems can provide system designers with: correct operation without understanding the

synchronization constraints of each independent system element, faster development cycles since each logic element need not be redesigned for efficient system operation, a longer usable life for subsystem logic elements, and a design structure which provides the maximum achievable performance for the technology and operating conditions provided. Also, performance is maximized without any lost in system reliability and robustness.

DESIGN CONSTRAINTS

With the obvious advantages of self-timed system structures, present self-timed circuit designs have some negative attributes which have hampered their use in general-purpose computer systems. The key component in a computer system is the CPU. Without an asynchronous CPU, or processor, an asynchronous system is impossible to implement efficiently. Only a few asynchronous general-purpose processors have been proposed [23, 58a, 63]. Most general-purpose processors contain complex data paths and a multi-token pipeline structure. A multi-token pipeline is a pipeline structure where each functional unit is operating on a different data variable, or token, during any given cycle. With the new generation of microprocessors using 64-bit data paths (i.e. MIPS R4000, Intel i860, IBM RS/6000, HP PA-RISC) encoding of logic signals to support self-timed operation presents some serious problems.

Dual-rail variable encoding is the most widely used style of self-timed circuit design [6, 11, 18, 21, 24, 28, 46, 59, 62, 69, 87, 104, 108]. Figure 2.5 gives example data streams for three dual-rail encoding schemes. These variable encodings increase the implementation complexity of the connected functional units. To illustrate this, Figure 2.6 shows two dual-rail implementations of a full-adder element [69, 84]. Using dual-rail encoding allows the sequencing-control logic to detect completion of each functional unit operation. A traditional two-input dual-rail completion detector is shown in Figure 2.7. The gate labeled "C" is a Muller C-element. The C-element's output becomes 1 when all the inputs are 1, becomes 0 when all the inputs are 0, and remains at the previous output state when the inputs are not all 0 or 1. The use of a C-element tree is required to detect completion for a larger number of variables. C-elements are discussed in more detail in a later chapter. Figures 2.5, 2.6, and 2.7 show the basic configurations and complexities of traditional self-timed logic structures.



Encoding							
Single-Rail	0	1	0	1	1	0	1
4-Phase Dual-Rail	01 00	10 00	01 00	10 00	10 00	01 00	10 00
Transition Signaling	00	10	11	01	11	10	00
LEDR	00	10	00	10	11	01	11

Figure 2.5: Example data stream showing complexities of dual-rail encoding schemes.

Another problem with building a self-timed, multi-token pipeline structure is the time required to generate a completion detection signal and the communication overhead required to support data transfer between adjoining pipelined elements. Figure 2.8 illustrates how self-timed elements are typically connected in a asynchronous pipeline structure. The asynchronous sequencing overheads result from the completion detector (CDs) and C-element (Cs) delays. These devices communicate the operating condition of each logic block to its successor and predecessor, adding to the sequencing period of that logic block. Williams [108] has shown how this overhead can affect the pipeline's throughput and provides a detailed discussion of the performance levels of several asynchronous pipeline structures. He also proposed methods for minimizing both completion detection and communication overheads. By using a single data-variable bit to detect completion (rather than using a full completion tree for all the bits), Williams significantly reduced the completion detection overhead for the carry-save-adders used in his divider circuit. He eliminated the communication overhead between function elements by pipelining the communication operations among the internal elements. This was possible since the divider operated on one token at a time. Because of the functional unit granularity required, these methods are less efficient for multi-token, pipelines used in general-purpose processors.

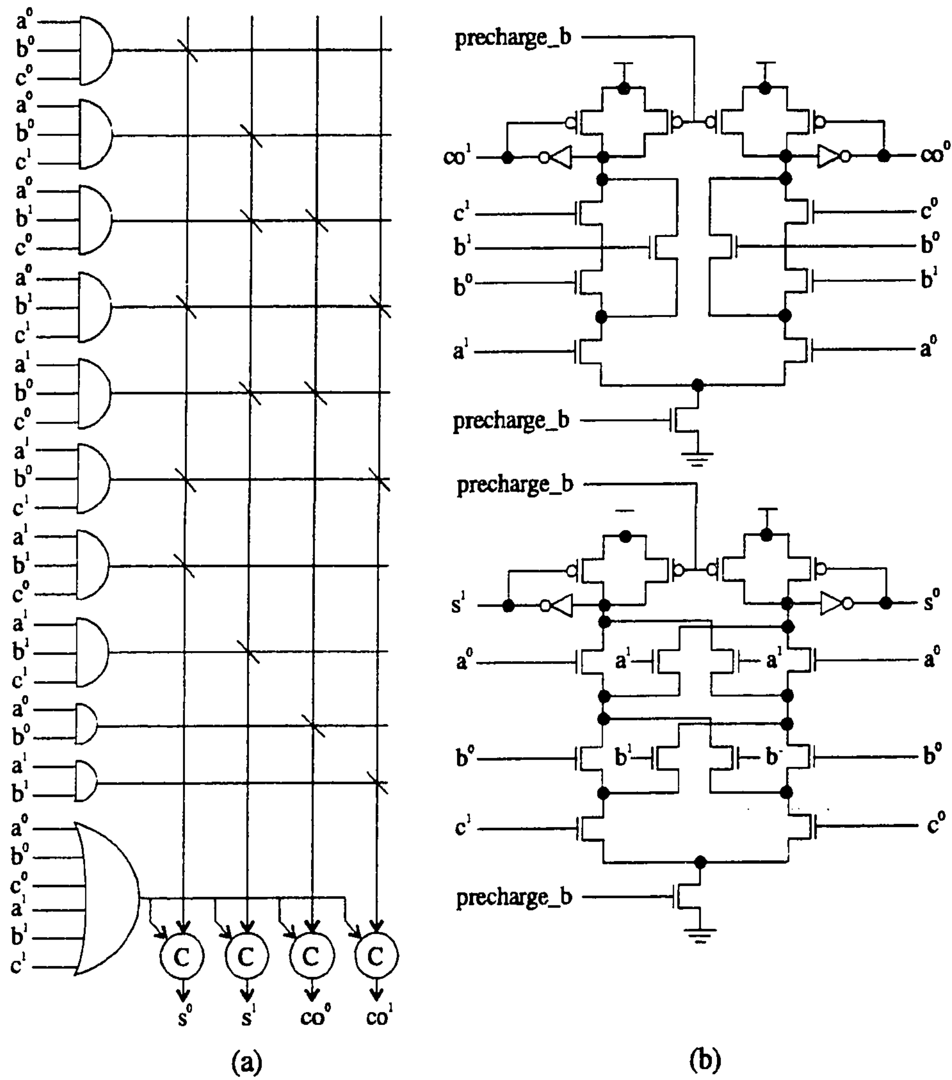


Figure 2.6: Self-timed dual-rail full-adders using (a) PLA-like and (b) DCVSL structures.

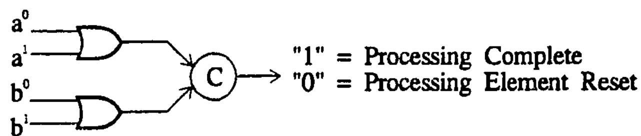
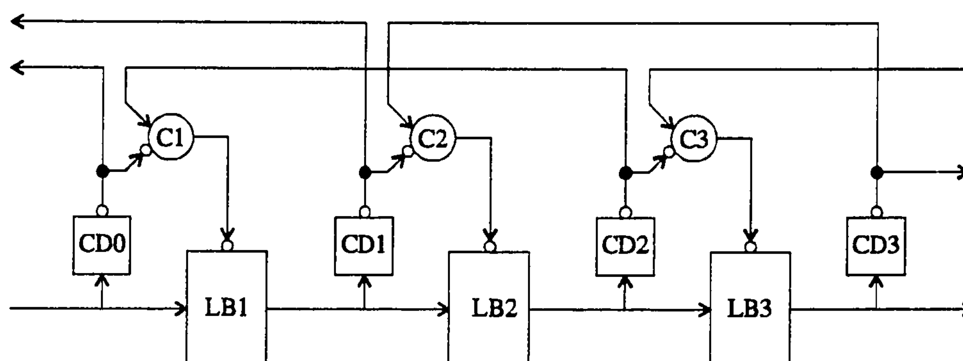


Figure 2.7: Traditional two-input dual-rail completion detector.



LB = Precharged dual-rail combinational logic blocks
CD = Completion Detectors
C = C-elements

Figure 2.8: A traditional asynchronous pipeline structure.

Another way to minimize the effects of completion detection and communication is to construct large processing elements so that the delay of completion detection and communication is small compared to the processing elements' delays [69]. But in a general-purpose, VLSI processor the functional units may only require 20 series gate delays (as is required by the functional units described in Chapter 5), while the completion detector and communication signalling requires three gate delays, minimum (one C-element and one NOR when using a single data-bit pair for completion detection). For this example, the completion detection circuit would increase the functional unit latency by roughly 15%.

Considering that development tools for self-timed circuit design are still being researched, it's difficult for a system designer to implement a efficient and competitive self-timed processor or logic system, but not impossible. DSPs [46, 69], control logic, multi-processor switch networks, and even a 16-bit RISC processor [63] have been implemented using self-timed logic structures. But before self-timed logic becomes widely accepted, its implementation complexity and size must be reduced and its performance level must be proven better than an equivalent synchronous design.

2.2.3 Self-Timed Synchronous Systems

Even with the potential for adaptive operation and ease of interfacing, self-timed processor designs have yet to be commercialized. The increased complexity and sequencing overhead of self-timed logic structures have limited their use in processor and system designs. The ideal pipeline sequencing method would have the implementation size, complexity, and design ease of a synchronous pipeline with the adaptive operation (to environmental conditions and process parameters), wide operating range, and interfacing efficiency common to self-timed structures. Chapter 3 describes a pipeline sequencing method with these characteristics called *dynamic clocking*.

Dynamic clocking is a pipeline sequencing method which is best described as a *self-timed, synchronous structure*. All pipelined functional units sequence in lock-step via a global sequencing signal. The period of this signal adapts on a cycle-by-cycle basis to the environmental conditions, process parameters, and pending pipeline operations. The pipeline sequencing signal, or "clock", also stops and waits for operations with indeterminate delays to complete (external memory and I/O transfers). This design style supports a fully asynchronous external interface. The main goal of dynamic clocking is to provide processor and system designers with a sequencing and interface method which uses synchronous design tools and logic elements while providing the interface efficiency and performance available through self-timing.

2.3 Memory System Design

Memory system performance dramatically effects, and in many cases limits the performance of advanced microprocessors. The ability to reduce the average access time of the memory system depends on many factors: the processor's architecture, the program's behavior, the caches' sizes and organizations, and the fetch and prefetch strategies. A balance must be struck between the caches' access times (the general rule is smaller caches have lower latencies), the caches' miss rates, and the penalty paid for a cache miss. Ideally, the first-level cache access time is less than the other pipelined functional unit latencies and the miss rate is low. The main goal in developing a more efficient memory system is to minimize the average time required to access a memory word. This section first describes the hardware and software techniques presently being researched, developed, and used to optimize memory system performance in advanced

microprocessors. Finally, a modification to the traditional memory hierarchy is proposed which significantly improves memory system performance and processor efficiency.

2.3.1 Caching

Caches are used in all forms of computer systems: embedded controllers, laptops, personal computers, workstations, mainframes, and supercomputers. Caching is the simplest way of maximizing a system's performance. Because of their importance to efficient system operation, caches have been extensively studied over the years. A. J. Smith [93] lists over 400 books and papers published since 1968. Since Smith's bibliography, many other documents have been written analyzing cache structure and operation. Studies have focused on cache size [2, 92], associativity [39, 41, 78, 97], block size [79, 94], and fetch size and fetch strategies [7, 15, 32, 40, 53, 60, 81, 90, 91, 95]. Przybylski [78] gives the most complete study and analysis of each of the major memory hierarchy design variables and their inter-dependencies. Since present integrated circuit technology allows a CPU and a small first-level cache to reside on a single chip, studies on how to maximize the performance and efficiency of small caches are numerous. Most of the small cache studies have deal only with hit rates [2, 4, 30, 41] and have not considered implementation complexity, access time, and interface efficiency.

Reducing the average access time for instruction and data requests is the main goal of any memory system design. Excluding the register file, the cache closest to the CPU, or first-level cache, has the shortest access time and highest bandwidth of all the levels in the memory hierarchy. Since most program behavior exhibit the principles of *spatial locality, temporal locality,* and *instruction sequentiality,* most CPU accesses are contained in the first-level cache(s). An efficient memory system design has an average memory access time approximately equal to the access time of the first-level cache.

There are several ways to reduce the effective access time of data and instruction references to the memory system. One way is to increase the memory system's throughput by pipelining the first-level caches [61, 75]. This technique is usually required when the first-level caches have an access time greater than the operational latency of the other functional units in the pipeline. The MIPS R4000 is an example of a processor that employs first-level cache pipelining. This pipeline cache caused two dependency hazards in the machine, one for data and one for instructions. For data, dependency checking logic stalls the machine to ensure a data load operation completes

before the data is required by subsequent instructions. For instructions, the pipeline cache increases the number of cycles required to restart the pipeline after a miss predicted branch operation occurs. Therefore, the problems with this technique is that it increases the number of stages in the pipeline (resulting in an increase in the branch and load penalties) and forces the memory unit to be more complex.

Another technique to reduce the average access time of data references is to provide queues, either explicitly architected or transparent to the user, that allow the machine to continue executing instructions while waiting for the memory request to be serviced. Since the memory system speed is not integrated into the processors architecture, more hardware is required to resolve data conflicts. The IBM 801 [10] provides a single element transparent queue that allows instructions after a load, which do not use the requested data, to continue to operate after the load reference is initiated. Queues have also been used in the instruction fetch stream. Intel's X86 family of processors use instruction queues to fetch ahead of the instruction request and disconnect the memory system performance from the processors sequencing rate.

Another method of improving the performance of instruction references is the use of *target instruction buffers*. Rau [81] first suggested the use of target instruction buffers for CISC type instruction streams (IBM 360) and Hill [40] examines their use in other architectures. A target instruction buffer stores the targets of previous non-sequentialities in the instruction stream. These targets include the target word after a branch and *d* succeeding words in an attempt to eliminate the penalty of a non-sequentiality in the instruction stream. When a branch is taken, the instructions are taken out of the target instruction buffer and the instruction-fetch control logic references instructions sequential to the ones in the buffer. Buffering of a couple of the most recent targets produced a significant improvement in memory system performance. Rau and Rossman [82] later studied the effect these buffers have on the IBM 370, CDC 6600, and the Manchester University MU5 system. They found that the use of prefetch buffers and target instruction buffers reduced instruction fetch delays by 50%. The AMD29000 [5] used a target instruction buffer in early processor implementations. The main problem with this fetch structure is the high amount of continuous memory bandwidth required in the next level of the memory hierarchy.

To improve the performance of data references Jouppi [53] suggest the use of *miss caching* and *victim caching* to improve the first-level cache's average miss penalty. Miss caching places a small fully-associative cache between the first-level cache and the next level in the memory hierarchy. The miss-cache stores the missed cache blocks in a LRU

fashion. Misses in the cache that hit in the miss-cache have only a one-cycle miss penalty. Victim caching is an improvement to miss caching where the cache block replaced by the new cache entry is stored in the small fully-associative cache. Both methods increase the associative of a directed mapped cache and are most effective in the data stream. Jouppi also suggested the use of ***stream buffers*** in the refill path of a direct-mapped instruction cache to store prefetched cache blocks starting at a cache miss address. This is similar to the *fetch-on-fault* prefetch strategy described in Chapter 4, but removes the requirement of dual-porting the cache. None of these techniques reduces the access time of the first-level cache, but they do minimize the first-level cache miss penalty. Therefore, these caching techniques provide no reduction in the pipeline's cycle time, one of the main goals of our research.

2.3.2 Fetch and Prefetch Strategies

One criteria used in all memory system designs is the minimization of the first-level cache miss ratio. The miss ratio of a first-level cache is often limited by its size, which is typically controlled by the available technology. To further reduce its miss ratio aggressive fetch and prefetch strategies are required. There have been a number of studies of fetch and prefetch strategies [7, 15, 30, 32, 53, 60, 66, 81, 90, 91]. We define *fetch strategy* as the interface method and protocol used to access and transfer information into a cache memory and processor pipeline during a cache miss operation. *Prefetch strategy* is defined as the interface method and protocol used to access and transfer information into a cache memory or data buffer before it is requested by the processor. Prefetching is typically based on previous processor reference patterns. The history of reference patterns may simply be the last reference, used in sequential prefetching, or the collection of all previous references, used in predictive prefetching.

Specifying a fetch strategy involves defining a control algorithm for a significant number of operational parameters. The basic fetch parameters include: which word in a cache block is returned first, when is the processor allowed to continue execution, what replacement algorithm is used, and how many subblocks or blocks are retrieved during a fetch. The most common fetch strategy, called *non-blocking-requested-word-first* (*nbrwf*), fetches based on a cache miss, returns the requested word first, allows the processor to continue execution after the first word is retrieved, and fetches a total of one cache block (fetch size = block size). Modifications to this strategy include: *blocking-sequential-word-first* (*bswf*) a very simple policy where the processor execution is stalled

until the entire cache block is fetched or ***terminating-non-blocking-requested-word-first*** (*tnbrwf*) where if another miss occurs while a fetch is in progress, terminate the fetch and service the new processor request. Przybylski [78] found that the *nbrwf* fetch strategy provided the best performance, though only 4% better than *bswf*.

A prefetch strategy contains the same basic fetch parameters as a fetch strategy plus additional parameters to control when the prefetch occurs, when the prefetch operation has priority over a processor request, and how deep in the memory hierarchy the prefetch is propagated. An efficient prefetch strategy can reduce the miss rate of a cache or prefetch buffer, resulting in a reduction of the memory-system average-access time. But, often prefetching of instructions and data is not practical because of the lack of transfer bandwidth between the cache and higher levels in the memory system. Also, while most prefetch algorithms are effective at predicting future instruction references, they have a difficult time predicting future data references. This has prompted research and development of other methods of reducing the memory systems average access time. The next section briefly describes an alternate method of using small low-latency caches along with an aggressive prefetching algorithm to significantly reduce the average access time of the memory system.

2.3.3 Predictive Prefetching

To minimize the access time of the first level in the memory heirarchy, we propose adding very small fully-addressable prefetch buffers between the CPU and the first-level cache(s). Their location in the memory hierarchy and their operational characteristics lead us to call these prefetch-buffers *zero-level caches* or *L0 caches*. Figure 2.9 shows the location of the zero-level caches in a traditional memory hierarchy. Zero-level caches are placed in both data and instruction reference paths. Because of their small size (less than 256 bytes) zero-level caches have less than half the access delay of typical internal first-level caches (usually 8KB in size). The zero-level cache size is restricted to minimize the access time and to keep the implementation complexity of these fully-associative caches manageable. Therefore, a zero-level cache is approximately 3% the size and has less than half the latency of a typical internal first-level cache.

Unfortunately, previous implementations of small caches have tended to be ineffective. The high miss ratios associated with small caches more than offset their fast access times. To improve the miss ratios of the zero-level caches several fetch and prefetch strategies were studied and simulated, and a new prefetch algorithm was created.

Predictive prefetching is a hardware controlled prefetching method which uses a history of reference patterns to predict the future reference patterns of the CPU. Predictive prefetching is similar in concept to branch prediction using a branch-target-buffer [50, 65, 88], except it is designed to support the general reference stream for both instructions and data. The main goal of predictive prefetching is to provide a simple and effective means of prefetching processor references from the first-level caches to the zero-level caches. With the proper interface size, block size, and cache size, predictive prefetching allows the zero-level cache to have a hit-ratio similar to the first-level cache, but with a access time no more than half that of a typical first-level cache. Chapter 4 gives details of zero-level caching and predictive prefetching and their hardware implementation structures.

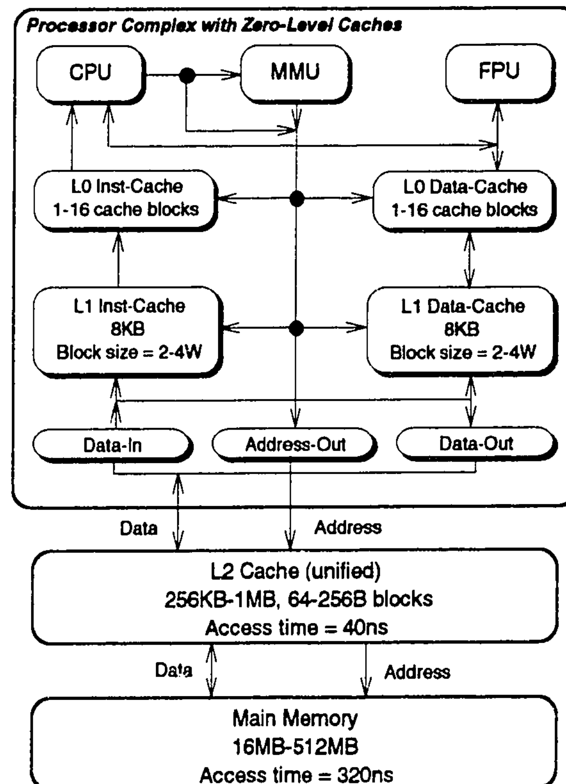


Figure 2.9: Memory Hierarchy with Zero-Level Caches

2.4 Summary

This chapter provided an overview of traditional pipeline sequencing methods. Synchronous pipeline operation dominates modern processor designs but limits potential processor performance because of worst-case design constraints. Traditional asynchronous pipeline structures avoid worst-case design constraints by providing adaptive operation and asynchronous interfaces. But these asynchronous structures are constrained by pipeline sequencing overheads caused by completion signalling. We proposed an alternative pipeline sequencing method called dynamic clocking. Dynamic clocking is a self-timed synchronous sequencing method which provides the adaptive-operating and efficient-interfacing characteristics common to asynchronous logic structures while offering the implementation ease, sequencing efficiency, and small physical design of synchronous design structures. Dynamic clocking gives the designer more flexibility in determining the pipeline's sequencing dependencies and allows the extraction of all the raw performance available in a silicon technology.

This chapter also described the attributes and constraints of modern memory system designs. It suggested the use of fully addressable prefetch buffers between the CPU and first-level caches to reduce the average memory system access time of a traditional memory system. To help minimize the miss ratio of these small caches (less than 256 bytes), a prefetching algorithm is proposed. Predictive prefetching uses the history of references to predict future instruction and data references. The main goal in modifying the memory hierarchy is to reduce the average access time of the memory system to a latency level less than the other critical pipeline stages. By using zero-level caches with predictive prefetching and dynamic clocking, a processor's performance is improved: average cycle time is reduced without increasing CPI or instruction count.

PLEASE NOTE:

Page(s) missing in number only; text follows.
Filmed as received.

UMI

Chapter 3

Dynamic Clocking

Dynamic clocking is a pipeline sequencing method for processors and controllers which provides a mechanism for designing a self-timed, synchronous pipeline structure. Dynamic clocking provides processor and system designers with a pipeline sequencing and interface structure which uses synchronous design methods and logic elements, provides environment and process dependent performance, and yields an efficient asynchronous interface scheme.

This chapter provides a detailed description of a dynamic clocking structure for a typical RISC processor. The basic dynamic clocking sequencing structure is given in Section 3.1. Section 3.2 details the design constraints and implementation tradeoffs involved when using dynamic clocking. Section 3.3 describes a dynamically-clocked RISC processor which is based on the MIPS-X processor. An optimized RISC processor implementation, which takes better advantage of self-timed operation, is described in Chapter 5.

3.1 Basic Structure

There are two main goals targeted by the development of dynamic clocking. The first is to improve processor performance by recovering the available silicon performance lost because of synchronous design constraints. The second goal is to provide an efficient and simple means of interconnecting chips and subsystems, independent of their individual optimal operating rates. To achieve these two goals, a simple means of asynchronously sequencing a processor pipeline is required. When compared to a

synchronous design, this asynchronous sequencing method must not increase the logic required to build the pipeline structure or decrease the typical performance of the functional units. Therefore, we must understand the constraints limiting the throughput of a synchronous processor and eliminate these constraints without significantly changing the traditional logic structures used.

The throughput of a processor is determined by the functional units with the worst-case operational latency. In general, synchronous designs using a static-frequency clock are limited by the latency of the slowest operation among all the pipelined functional units. This limitation holds true independent of the frequency-of-use of that operation. A fully asynchronous pipeline, implemented using self-timed logic elements, is only limited by the slowest operation for a given cycle. The slowest operation will vary on a cycle-by-cycle basis. Because of the typical RISC pipeline structure and functional unit design, the asynchronously-sequenced functional units which finish before the slowest operation must wait until that operation has completed before advancing to the next data token. This implies that as long as a pipeline can be sequenced based on the slowest operation for each cycle, a fully asynchronous structure is not required to achieve maximum performance.

Dynamic clocking uses a self-timed, synchronous pipeline sequencing structure, avoiding the complexities of a fully asynchronous structure. It is a synchronous structure clocked by an environmentally adaptable, operationally variable, stoppable clock. A dynamically-clocked pipeline will sequence the functional units in lock-step based on the environmental conditions, process parameters, and the critical-path operation pending for each cycle. The *dynamic-clock generator* is on chip and receives operational information from the pipeline functional units and provides a clock period long enough to support the pending operations. The dynamic clock generator does not require an external oscillator or crystal, so the use of a phase-lock-loops are not required. On operations where latencies are indeterminate (data transfers between the processor and independently sequenced devices) the clock will stop and wait until a completion signal is received from the active processing unit. Finally, since the dynamic-clock generator is entirely on the processor chip, its operation tracks the environmental conditions and process parameters present for the logic elements in the pipeline.

Santoro [83] developed and tested a similar clocking structure for a interactive-array multiplier used in a high-performance floating-point multiplier. He proved that a functional unit's processing latency can accurately be tracked, and a local-clock generator can be constructed around this tracking element. In the multiplier application, only one

operation was tracked, a carry-save-add (CSA). Therefore, the clock was not required to vary based on a variety of operations. The multiplier's clock generator did adjust for changes in the silicon process and environment.

Santoro found that a tracking cell, constructed using the components of the CSA critical-logic path, provided more accurate tracking of the CSA latency than did a tracking cell constructed from a inverter chain. Also, the ability to start and stop the clock for each multiply operation allowed automatic and efficient synchronization to all input operands. This type of clock operation eliminates metastability concerns when interfacing two non-synchronized devices.

Like Santoro's method, the dynamic clock generator uses tracking cells for delay matching. However, there were additional complications because of the need to support a variety of operations and dynamically adapt the clock period to those operations on each cycle. Figure 3.1 is a general block diagram of a dynamic clock generator. The tracking cells are chosen and designed dependent on the critical processing paths in the pipeline. The expected utilization of each critical-logic path and its relative processing latency are two main factors in deciding which paths will be tracked.

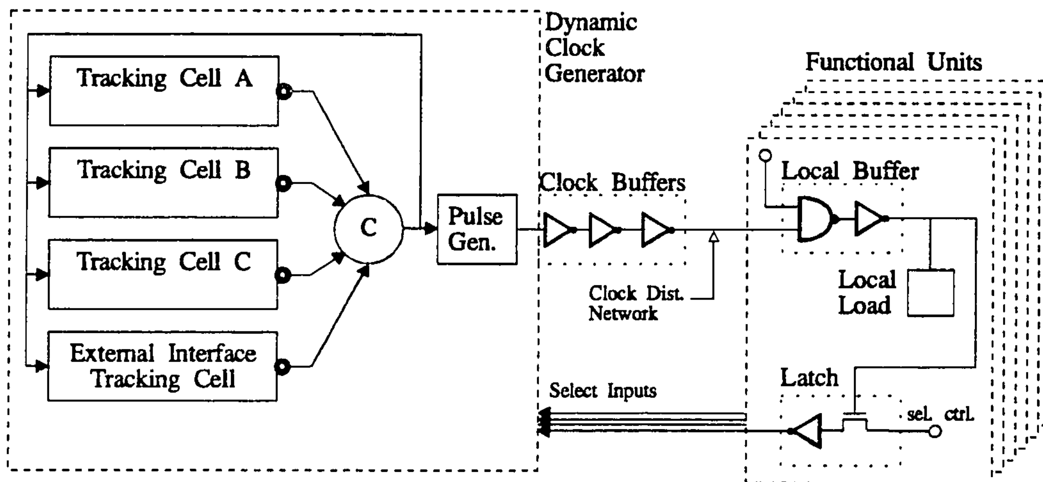


Figure 3.1: General dynamic clocking structure.

The independent tracking of each critical-logic path supports a continuously running dynamic clock generator, thus avoiding sequencing overhead caused by stopping the clock, waiting for a completion signal, and starting the clock before each cycle

(experienced in Santoro's application). To support this dynamic clocking scheme, information from the instruction decoder, the caches, and the bus interface unit is used to determine the operations required during the next pipeline cycle. The tracking cells allow the clock generator to establish the clock period required for the next operation during the present cycle. Therefore, the dynamic clock generator establishes the required clock period ahead of when it is actually used to sequence the pipeline (in continuous time units). Creating the clock ahead of time eliminates the communication overhead associated with traditional completion detection schemes. The flow chart shown in Figure 3.2 illustrates the sequence of operations required by a dynamically clocked digital system.

The clock stops during data transfer cycles to devices outside the clocked pipeline structure. The clock generator must wait on a completion signal from the transferring device before processing can continue. The latency required to restart the clock is composed of chip interface delays, generator delays, clock buffering delays, and the clock distribution wires. In many cases the clock startup time is small compared to the processing time of the transfer. Also, the clock-startup time can be overlapped with part of the data-transfer time, reducing its overhead. The operations requiring stopping of the clock would include: second-level cache accesses, memory system accesses, floating-point unit transfers, I/O device transfers, and shutdown operations (to conserve power).

3.2 Constraints and Tradeoffs

The following subsections describe the constraints and critical elements which controlled the dynamic clock generator implementation and the construction of the pipelined elements.

3.2.1 Tracking Cells

The tracking cells must accurately track the delay of the target operations and match the variations caused by changes in temperature, voltage, and process. This can best be accomplished by duplicating the series of gates, wires, and signal loads that exists in the critical-logic path being tracked. The tracking cells must also be symmetric in their output transition delays, providing a consistent clock period from cycle to cycle. Since the minimum delay of the tracking cell must be greater-than the maximum delay of the respective critical logic path, asymmetric operation reduces the efficiency of the dynamic

clock generation. The goal is to have the cycle to cycle clock period variation for a single critical-logic path be as small as possible (for constant environmental conditions and process).

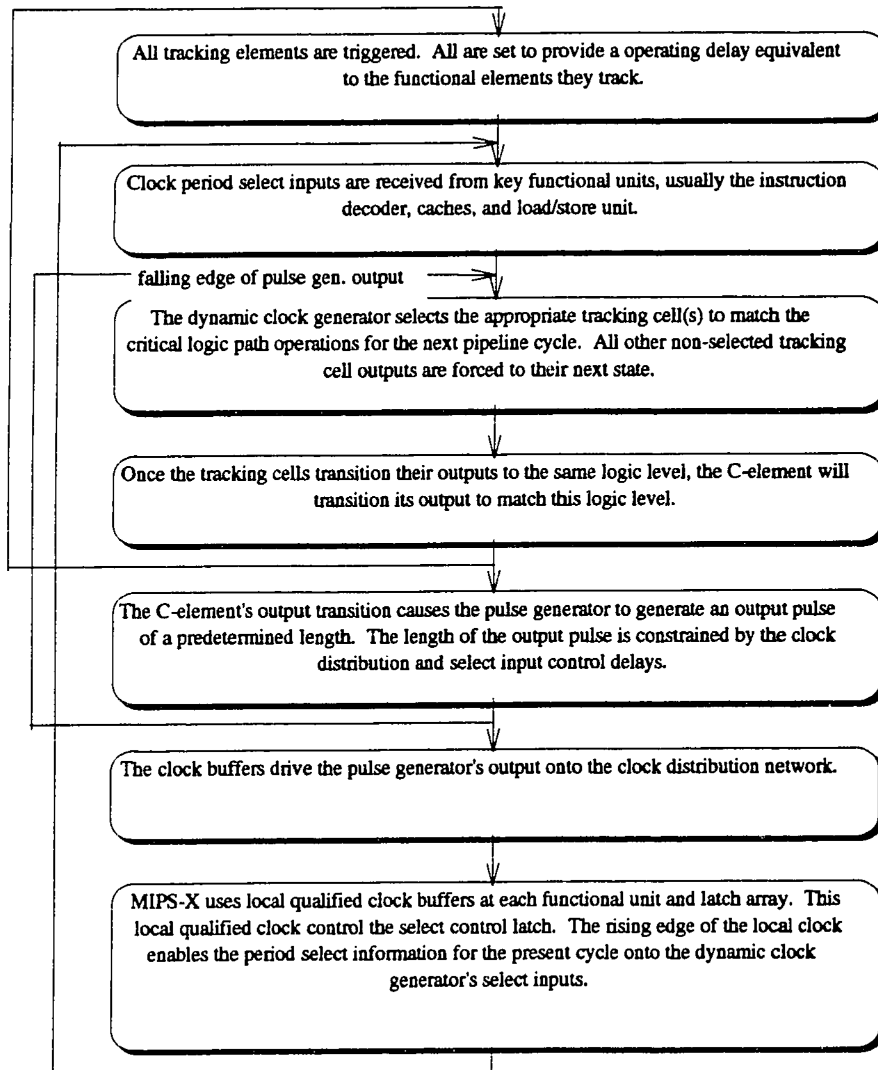


Figure 3.2: Flow-chart describing the sequence of operations required to support dynamic clock generation.

Tracking cells are designed to match the critical-logic paths which dominate the pipeline sequencing period. The designer must identify these dominant operations based on their latency and frequency-of-use. Decoding of tracked operations must be available at least during the cycle before the operations occur. This allows generation of select signals for enabling the tracking cells in the dynamic clock generator. Section 3.3 provides an example of the tracking cell selection process for a dynamically clocked RISC processor based on the MIPS-X implementation.

A tracking cell must also be designed so that its non-selected output transition occurs before the output transition of the minimum delay tracking cell. Generally, the output transition of a tracking cell is advanced by forcing the internal gates to switch in parallel, instead of serially. Multiplexers are used to accomplish this.

3.2.2 C-element

The gate in Figure 3.1 labeled "C" is a Muller C-element [72]. It is in the clock generation logic path. The main purpose of the C-element is to detect the transition of all tracking cells and indicate to the pulse generator when all the tracking cells have timed-out. The C-element operating characteristics allow several tracking cells to be activated during a single clock cycle. The tracking cell with the longest delay will control the C-element's switch point and thus the cycle time for that pipeline period.

The C-element delay must be minimized to reduce its overhead during clock generation. The tracking cells are designed to hide most of the C-element delay without sacrificing tracking efficiency. The amount of overhead caused by the C-element should be less than 5%. The C-element must be non-inverting, have symmetric output transition delay, and drive 16 standard loads. Since no existing design satisfied all these criteria, we designed a new one which is described in Chapter 5.

3.2.3 Pulse Generator

The pulse width generated by the pulse generator must be greater than or equal to the worst-case delay between the output transition of the pulse generator and the valid transition of the inputs that select the period of the next clock. The series of elements composing this delay includes: (a) the global clock buffers, (b) the clock distribution network, (c) the local clock buffers, (d) the select control latch, and (e) the select input wire delay. Another minor constraint on the pulse generator is the latching delay times

of the pipeline storage elements (MIPS-X uses static latches while the STRiP implementation assumed dynamic latches).

The pulse generator is basically a delay element connected to XOR/XNOR gates to create a output pulse on each C-element output transition. Figure 3.3 is a block diagram of the pulse generator. The pulse generator generates true and complement output pulses, allowing true and complement clock signals to be generated. The output pulse width, which becomes the ø1 period, is set by the propagation time of the delay element. The XOR/XNOR gates must be designed for symmetric and identical propagation delays. This reduces their effects on the pulse width time set by the delay element. The pulse generator adds to the skew between the clock generation point (the C-element output) and the functional units, but has little effect on the dynamic clock period set by the tracking cells.

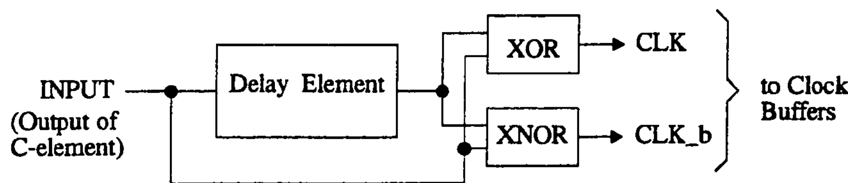


Figure 3.3: Pulse Generator block diagram.

Figure 3.4 is a transistor-level schematic of the pulse generator. The delay element consists of an inverter chain whose delay satisfies the select feedback constraint. Fifteen inverters were used in the inverter chain, yielding a ø1 period equal to approximately half the minimum cycle time (equivalent to half the MIPS-X first-level cache latency of 30 gate delays). The XOR/XNOR gates were designed to provide symmetric operation, full CMOS output levels (enough drive to support a fanout of 16) and minimum C-element loading. The structure shown in Figure 3.4 provided the best compromise between symmetry, drive and loading.

3.2.4 Clock Control and Distribution

Because of the dynamic clocking structure, a two-phase overlapping clock scheme is used for the pipelined-functional units. Any series connected ø1 and ø2 latches must be

physically located close enough to each other to allow a non-overlapping local qualified clock buffer to be used, Figure 3.5(b). Even when the ø1 and ø2 latches are separated by a logic element, care must be taken to insure proper operation (due to clock skews in the clock distribution network). The use of locally-qualified clock buffers minimizes the clock skew between functional units.

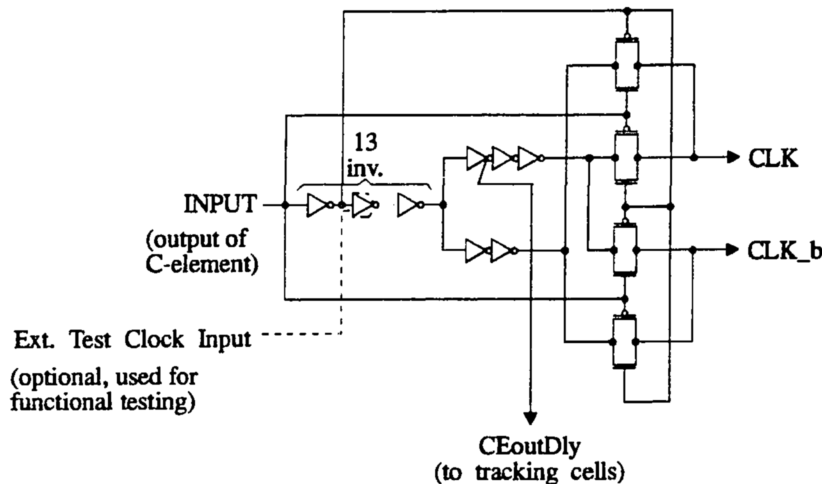


Figure 3.4: Pulse Generator transistor level diagram.

The period of the dynamic clock generator clock must be controllable from an external source to allow sufficient control for manufacturing test. Also, during normal operation the clock periods generated by the tracking cells and the ø1 period generated by the pulse generator must be externally and separately adjustable. This feature is referred to as the *KNOB*. This external adjustability allows an incorrectly tracked operational delay to be compensated for by varying the tracking cell or pulse generator delays. The KNOB adds enough flexibility to repair a non-functional sequencing structure. Chapter 5 defines and evaluates implementation options for the KNOB.

If dynamic latches are used in the pipeline structure, the clocks must not be stopped for more than the minimum dynamic storage time of the latches. The clock is suspended during dependent transfers to devices operating asynchronously of the processor pipeline (floating-point unit and external bus interface). This implies that a time-out timer be incorporated to terminate the stopped-clock condition before latched data is lost. The use

of static latches eliminates this concern but can decrease the performance of the processor.

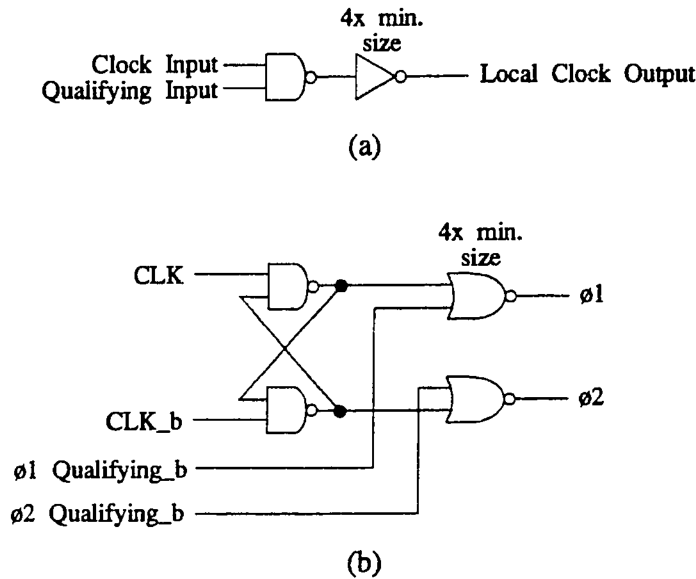


Figure 3.5: Local qualified clock buffers: (a) single-phase and (b) two-phase non-overlapping clock generation.

3.2.5 Functional Unit Design

Functional-unit operation must be independent of the dynamic-clock duty cycle. The clock period will track pipeline variations caused by environmental, process, and operational changes. But the clock duty cycle does not directly track variations in functional unit operation. The select feedback delay is the main constraint setting the clock high time (ø1). The clock low time (ø2) varies with the selected tracking cells. Therefore, functional units cannot depend on the length of the clock phases.

To satisfy this constraint, most precharged logic operations must be self-timed to remove the signal-precharge constraints from the clock phase periods. Where possible, static logic elements are used in the functional units. This design approach also simplifies tracking cell implementation.

3.2.6 Clock Buffers

The clock buffers are designed to minimize the clock skew between the pulse-generator output and the inputs of the functional units. They also are required to drive the clock distribution network. The clock distribution network we used was derived from the MIPS-X implementation. This network consisted of four clock signals with 20pf of load/clock distributed over 10mm of signal wire. The use of locally-qualified clock buffers helped to reduced the total load seen by the global clock signals. To create the clock buffers, a series of three inverters is used, each four times the previous inverter's size. The sizes were 6x, 24x, and 96x the minimum inverter size used in the processor design. The factor-of-four increase per logic stage has proven to be a good rule of thumb for building static CMOS buffers [107].

If these ground rules and constraints are followed, a dynamically-clocked processor will have most of the advantages of self-timed operations, and the implementation simplicity of a synchronous design. The next section describes the configuration and operation of the critical elements in the dynamic-clocking structure for a MIPS-X compatible architecture.

3.3 A Self-Timed MIPS-X

The MIPS-X processor architecture provides a good platform for determining the feasibility of using dynamic clocking on a RISC processor. The function unit designs are unmodified except for the self-timing of some precharge operations. The addition of a internal first-level data cache, of the size and complexity of the first-level instruction cache, provides an implementation example closely matched to integration levels achieved by other modern RISC processors. IRSIM and SPICE simulations are utilized to establish the functional unit latencies and control overhead. Since much study and analysis has been performed on RISC processors similar to MIPS-X, information is readily available to help understand the instruction mix, projected cache hit rates, and functional-unit performance and utilization. All of these elements are important in the development of an efficient dynamically-clocked processor.

Figure 3.6 is a block diagram of the MIPS-X pipelined functional units and their main interface signals. The dynamic clock generator replaces the normal synchronous

clock generator. The dynamic clock generator differs from the typical synchronous clock generator in the following ways:

(1) The dynamic clock generator does not require an external clock, crystal, or oscillator input.
(2) The dynamic clock generator does not require phase-lock-loop (PLL) circuitry.
(3) The dynamic clock generator uses feedback information from the pipeline to determine the period of the next clock cycle.
(4) The dynamic-clock cycle time automatically adapts to variations in process, temperature, and supply voltage.

The next section illustrates the tracking cell selection process for the MIPS-X architecture and implementation.

3.3.1 Tracking Cell Selection

Since the period of each pipeline cycle is dependent on the pending operations in the pipeline, and not on a single critical logic path, each functional unit must be optimized independently. During the design of each functional unit, a tracking cell is designed to match the operating structure of that unit. This tracking cell will provide the information required by the dynamic clock generator to set the cycle time for that functional unit's operation. Some functional units do not require tracking elements because their optimized execution rate, or access time, is always less than other frequently used pipeline operations. The performance and frequency-of-use of the functional units determine which operations are tracked, thus determining the processors operating rate. The dynamic clock generator tracking cells were chosen based on the analysis of the propagation delays of the MIPS-X functional units and previous studies of instruction usage [31, 37, 50, 75].

MIPS-X uses a five stage pipeline: (1) instruction fetch [IF], (2) register fetch/instruction decode [RF], (3) execute [ALU], (4) memory access [MEM], and (5) register write-back [WB]. An instruction fetch operation occurs on every pipeline cycle and happens too early in the pipeline to support a selectable tracking cell. It is important that the first-level instruction-cache access either be faster than the other frequently used

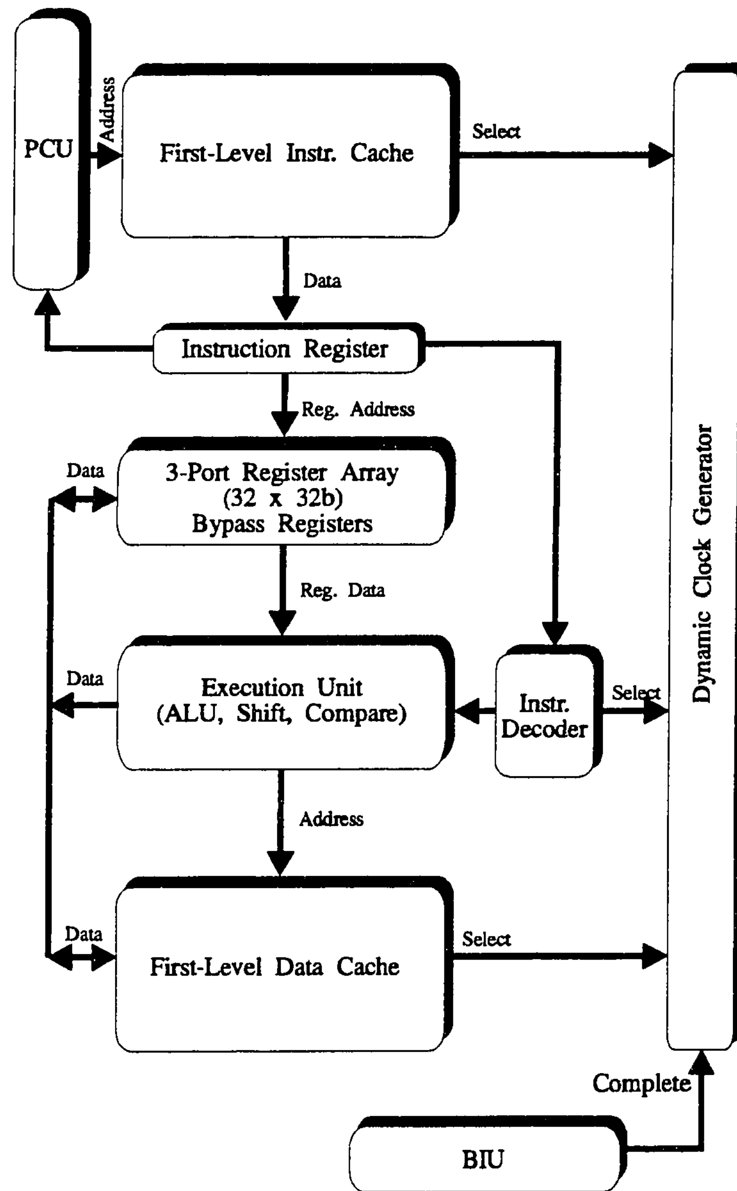


Figure 3.6: Block diagram of MIPS-X with dynamic clocking and internal data cache.

functional operations or be tracked on every pipeline cycle. The first-level cache access time (after self-timing the bit-line precharge and tag access) is less than several frequently used operations. But, a first-level cache tracking cell is required to optimize

the sequencing efficiency of the self-timed processor. The latency of the first-level cache sets the minimum cycle time of the self-timed, MIPS-X pipeline. Tracking Cell A is assumed to match the latency of the first-level caches and is called the ***minimum delay tracking cell.*** The minimum-delay tracking cell does not require a select input since it has the shortest delay of all the tracking cells.

The access time of this cache configuration, including tag-compare, bit-line precharge, address drives/latches and data latches, is approximately 30 gate delays. A gate delay is equivalent to an inverter delay with a fanout of four. Both first-level caches are assumed to be 2KBytes and 8-way set associative to minimize their miss rates (identical to MIPS-X). A copy-back first-level data cache organization with write buffers is assumed, allowing store operations to occur in the minimum cycle time. If a instruction fetch misses in the first-level cache, a select signal is driven to the dynamic clock generator to indicate that an external memory cycle is required. The pipeline is stalled and the clock stopped during the next cycle until the memory reference has completed.

Register reads, instruction decode, and bypass register selection occurs during the next stage in the pipeline sequencing. Like instruction fetches, instruction decoding and operand fetches are required on every processor cycle. The register fetch, instruction decode, and bypass register select operation were faster than a first-level cache access. The time required to latch and drive the source data to the other functional units is included in the operational delays of the tracked operations and is not considered as part of the register select/fetch delay. Therefore these operations execute within the cycle time set by the other pipeline functions.

The operation chosen to set the second most critical cycle time of the dynamic clock generator was the add operation (specifically, an ALU addition driving through bypass registers). Some form of addition is required on approximately every processor cycle. Table 3.1 lists the operations requiring an addition and their expected frequency-of-use [31, 37]. Some of these additions, i.e. PC increment, can be implemented with a faster execution time than the other add operations. But the other PC address calculations (branch address, jump address, and trap address) are multiplexed with the PC increment, increasing their total latencies. A full 32-bit addition, excluding PC increment, is required approximately 65% of the time.

Three factors lead to the choice of the add operation to set the second-fastest cycle time: the frequency-of-use, the add delay relative to the instruction fetch delay (approximately four gate delays longer), and the use of identical adder logic in both the

PCU and ALU. The ALU add path was longer-than, or equal-to the other add operations. Tracking Cell B is designed to match the critical delay path of an ALU add operation. The ALU adder tracking cell delay equals approximately 34 gate delays. All add operations except for PC increment cause the adder tracking cell to be selected. The select input for the tracking cell is generated by the instruction decode circuitry during the RF pipe stage (all selected adds occur during the ALU pipe stage).

Pipelined Functions Requiring an Add Operation	**Frequency-of-use (%)**
PC increment	100
Branch address calculation (conditional and unconditional)	15
Memory reference address calculation (load/store)	30
Arithmetic instructions	20
Total (excluding PC increment)	65

Table 3.1: List of operations requiring an add operation

The next slowest critical logic path is the compare-and-branch operation, which occurs during the pipeline's execute stage. This operation is required for each conditional branch instruction. The compare-and-branch operation uses the ALU adder to accomplish the compare (equal-to, greater-than, or less-than) and based on the results, selects the appropriate PC address for the next instruction fetch cycle. The total compare-and-branch delay is approximately six gate delays longer than the add operational delay. A branch instruction is approximately 15% of all executed instructions.

Finally, a tracking cell is required to interface the Bus Interface Unit (BIU) to the dynamic clock generator. This cell is utilized on each external data transfer which stalls the pipeline. External cycles are required during I/O transfers, other non-cachable data transfers (video memory), and on first-level cache misses. During an external data transfer the dynamic clock stops during ø2 and remains in ø2 until a transfer complete

signal is received by the Bus Interface Unit. We estimate (from detailed implementation given in Chapter 5) that the clock startup time will increase the second-level cache access time by less than 5%, assuming the processor and external cache are implemented in the same technology. Some of this additional delay can be hidden by driving the completion signal early relative to the data signals. At best, only half of this delay could be safely masked by an early completion signal. But this may be unnecessary, since the frequency of an external cycle is approximately 6% of all operations (caused by instruction and data cache misses).

Chapter 5 gives detailed implementations of the tracking cells used in STRiP's dynamic clock generator. The next section compares the performance of synchronous MIPS-X implementation to a dynamically clocked implementation using the tracking cells defined in this section.

3.3.2 Synchronous vs Self-Timed

The comparative performance of the synchronous and self-timed MIPS-X implementations was evaluated using a variety of simulation and analytical data. IRSIM, a switch level simulator, and SPICE, a non-linear circuit simulator, were used to determine the processing latencies of each critical logic path. Previous results from studies determining instruction mix and cache performance for RISC based systems were combined to determine the frequency-of-use of each functional unit. We assumed the design of each logical element is identical in both synchronous and self-timed implementations, except for specific cases where the sequencing method effects the design style (i.e. self-timed precharged buses versus phase precharge buses). The memory systems are identical in size and complexity. The primary goal is to determine the effectiveness of self-timing on an otherwise synchronous processor implementation.

Since the instruction set architecture for the synchronous and self-timed MIPS-X implementation are equivalent, the conflict penalties incurred by the two machines are considered to be equal. These penalties are caused by cache misses, load delays, miss predicted branches, and other resource conflicts. To simplify our calculations, the second-level cache is assumed to have a 100% hit ratio. This assumption has little effect on the final comparison because of the projected second-level cache size (512KB - 2MB). Both machines have identical cycles-per-instruction (CPI) measures, resulting from an equal number of conflict penalties per instruction.

The main difference between the two machines is the period required, and used, to sequence each operation through the pipeline. The synchronous system sequences the pipeline at a constant rate, determined by the worst-case environment, silicon process, and critical logic path. The self-timed system will attempt to adjust each cycle's period to match the prevailing environmental conditions, silicon process, and pipeline operations. To calculate a relative performance difference, independent of technology, gate delays are used to represent the processing latencies of the critical-logic paths. It is assumed that all logic functions are implemented in the same technology. The second-level cache access time is normalized to the same gate delay measure used for the processor. The analysis assumes a nominal operating environment (25°C and 5V), a nominal process, and the ability to sort the synchronous processor to within 25% of the available process performance (LSI Logic assumes a 50% process variance from nominal to worst-case).

Table 3.2 list the frequency-of-use and processing latencies for the critical logic paths of the MIPS-X processor. The processing latency used for the second-level cache access assumes an external cache controller with external SRAMs and parity checking. An average cycle time per instruction for both synchronous and self-timed implementations is calculated with these numbers. The number of stall cycles caused by branch and load penalties is assumed to be equivalent in both synchronous and self-timed systems, affecting their performance similarly.

The average synchronous cycle time per instruction is calculated as follows:

Average Sync. Cycle Time
= worst-case logic path latency
+ second-level cache [access frequency * access time]
= 40 + (0.078 * (3 * 40))
= 49.4 gate delays

The second-level cache access time required three processor cycles since synchronous system transfers require a discrete number of cycles. If the second-level cache access could be accomplished in two cycles, the average synchronous cycle time would equal 46.2 gate delays, a 6.5% performance increase.

Critical Operation	**Frequency-of-Use**	**Processing Latency** (Gate Delays)
Instruction Fetch (first-level cache access)	100%	30
Data Fetch (Load/Store, first-level cache access)	30%	30
Add/Sub (arithmetic and Load/Store, Execute Unit operation)	45%	34
Compare-and-Branch (branch instruction, PCU operation)	15%	40
Combined internal cache miss ratios (instruction and data)	6%	--
Second-level cache references (combined zero- and first-level cache miss ratios * % of references per instruction)	130% * 6% = 7.8%	100

Table 3.2: Frequency-of-use and processing latencies for the MIPS-X processor critical logic paths.

The average self-timed cycle time per instruction is calculated as follows:

Average Self-Timed Cycle Time
= branch [access frequency * processing latency]
+ add [access frequency * processing latency]
+ first-level cache [access frequency * access time]
+ second-level cache [access frequency * access time]
= $[(0.15)(40) + (0.45)(34) + (1 - 0.15 - 0.45)(30)](1 - 0.078)$
+ $(0.078)(100)$
= 37.1 gate delays

Therefore, assuming that both synchronous and self-timed MIPS-X systems are operated independent of worst-case process, temperature, and voltage, the self-timed system will operate 25% faster than the synchronous system. This performance advantage can be combined with the performance improvement provided by the self-

timed design when operating at nominal temperature, voltage, and process. The degradation factors for temperature, voltage, and process (provided in Chapter 2) can be used to calculate the total performance difference between synchronous and self-timed systems operating under nominal conditions:

$$t_{sync} = K_T * K_V * K_P * K_S * t_{self\text{-}timed}$$

where

t_{sync} = the average synchronous pipeline clock period
$t_{self\text{-}timed}$ = the average self-timed pipeline clock period
K_T = temperature degradation factor
K_V = voltage degradation factor
K_P = process degradation factor
K_S = sequencing degradation factor

Therefore the total performance advantage of self-timed pipeline sequencing under nominal operating conditions is:

$$\frac{t_{sync}}{t_{self\text{-}timed}} - 1 = (K_T * K_V * K_P * K_S) - 1$$
$$= (1.22 * 1.15 * 1.25 * 1.25) - 1$$
$$= 1.19 \text{ or } 119\%$$

Assuming perfect tracking of critical logic paths and no overhead in the dynamic clock generation, a dynamically clocked MIPS-X processor will theoretically operate more than twice as fast as an equivalent synchronous processor under nominal operating conditions. In fact, this hold even if the dynamic clock generator overhead is as much as 10% (caused by the C-element and inaccurate tracking cells). Self-timing through dynamic clocking provides a viable alternative to traditional synchronous operation by extracting most of the available silicon performance from a given implementation. The elimination of the external interface clock also allows efficient scalability as technologies change.

Chapter 4

Improving Memory System Performance

In modern processors the single most important factor limiting performance is usually the memory system. To optimize the performance of the processor, the memory system must have low latency and a high bandwidth. Cost-per-bit is also a factor in determining the practical memory system size and performance. This chapter describes an internal memory system and aggressive prefetching technique which doubles the performance of a traditional memory system structure without significantly increasing its cost or size. First, trends and constraints of modern memory system design are discussed. Second, an adaptive prefetching algorithm is described, called predictive prefetching, which uses a history of references to predict future references. Next, small low-latency caches (< 256 bytes) using predictive prefetching are analyzed and compared with a traditional memory hierarchy. Finally, the memory system structure used to support predictive prefetching is described.

4.1 Modern Memory System Design

Obviously, reducing the average memory system access time requires access times reductions to one or more levels in the memory hierarchy. The memory subsystem closest to the processor has the largest effect on the average access time and is usually constrained to work in a integer number of processor cycles. This first level in the memory hierarchy is usually a first-level cache. Today's processors often contain internal first-level caches whose sizes are determined by the available silicon area. Cache studies show that the speed-size-miss ratio tradeoff for very small caches (less

than 1K bytes) causes inefficient operation because of high miss-rates and total miss-penalties. Therefore, processor and system designers make the first-level caches as large as possible and use the fastest available RAMs. Modern microprocessors contain internal first-level caches of at least 16K bytes in total size or external first-level caches as large as 2M bytes.

Most processors are sequenced at a rate equivalent to the access time of the first-level cache. This simplifies the cache access and reduces the complexity of information control within the pipeline. The disadvantage of large first-level caches is that their access times are greater than the optimum processing rates of the other pipelined functional units. Therefore, one of the most common methods of decreasing a memory system's average access time involves pipelining the first-level caches.

Pipelining the first-level caches can double their average throughput and cut their effective access time in half. But cache pipelining adds complexity to the cache structures and increases the pipeline depth of the processor. Increasing the pipeline depth will increase the processing penalty for a miss-predicted branch instruction and data conflicts following a load operation. If branch and load delay slots are scheduled by the compiler, increasing the pipeline depth can increase the number of delay slots which must be filled with useful instructions. Therefore, care must be taken when using multi-cycle or pipelined caches so that potential operational penalties do not offset the gains in throughput.

A good example of a processor which uses pipelined first-level caches is the MIPS R4000 microprocessor. The R4000 is binary code compatible to the MIPS R3000 microprocessor. The R4000 uses an eight-stage pipeline (3-cycle first-level caches) while the R3000 uses a five-stage pipeline. Because the R4000 must be compatible with the R3000 binary code, its branch penalty is three cycles and the number of load delay slots is two. The R4000 branch and load CPI penalty is 0.42 cycles/instruction while the R3000 branch and load CPI penalty is 0.11 cycles/instruction [55, 57]. Assuming identical size caches, identical branch prediction strategies, and identical pipeline sequencing rates, the R3000 five stage pipeline can provide more performance than the deeper R4000 pipeline (Note: the higher CPI penalty of the R4000 is offset by a 50% reduction in cycle time). An ideal memory system would support the average access time provided by pipelined caches without an increase in pipeline depth and CPI.

As described in Chapter 2, there are studies that suggest methods of reducing the effective memory system access time through the addition of queues, FIFOs, buffers, and special memory structures. An alternate method for decreasing a memory system's

average access time, without increasing the number of pipeline stages, involves adding very small (less than 256 bytes), fully-addressable prefetch buffers to the memory hierarchy, between the CPU and the first-level cache(s). Their location in the memory hierarchy and their operational characteristics lead us to call these prefetch-buffers "zero-level caches" or "L0 caches". Figure 4.1 shows the location of the zero-level caches in the memory hierarchy. We placed a zero-level cache in both data and instruction reference paths to minimize resource conflicts. The zero-level caches are restricted in size to minimize their access times (less than half that of the first-level caches) and to keep the fully-associative configuration manageable. Therefore, a zero-level cache is roughly 4% the size and has less than half the latency of a 8K byte, direct-mapped first-level cache.

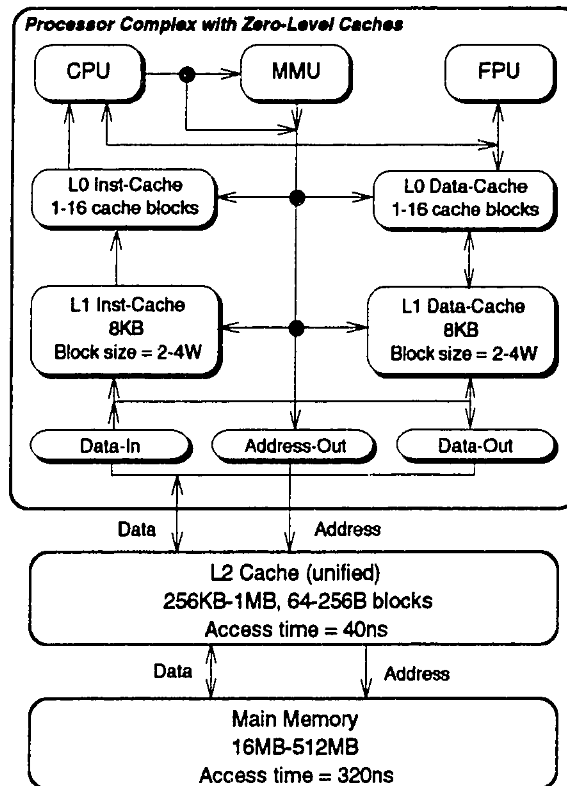


Figure 4.1: Memory Hierarchy with Zero-Level Caches

As mentioned earlier, one problem with small caches is that their high miss ratios more than offset their fast access times. Aggressive fetch and prefetch strategies can

significantly reduce the miss ratios of small caches in the memory hierarchy. Most prefetch algorithms are more effective at predicting future instruction references than future data references. We have developed a hardware-driven prefetching algorithm and a memory system organization which makes the use of zero-level instruction and data caches efficient and practical. The following sections will describe how this prefetching algorithm works, the hardware required, and the resulting performance relative to traditional memory system structures.

4.2 Prefetching

When developing a prefetching algorithm a designer must determine the importance of several key parameters and then set their optimum values. Some of these parameters include: condition determining the object to be prefetched, condition determining when the object is prefetched, size of the prefetched object, the prefetch depth into the memory system, the prefetch priority as compared to a processor request, and the bandwidth required to support the prefetch algorithm. We are interested in an adaptive prefetching algorithm which provides efficient prefetching of both data and instructions. The algorithm must be hardware-driven, providing efficient operation independent of the application software. The most common hardware-driven prefetch strategies are *sequential-prefetch-always*, *fetch-on-fault*, and *tagged-sequential prefetching*. The following defines each of these strategies:

(1) Sequential-Prefetch-Always -- A fetch to the next level in the memory hierarchy is initiated as a result of the processor reference to the cache, regardless of whether or not a cache miss occurred. This prefetch strategy requires a significant amount of memory system bandwidth and can fill the cache with worthless data. To accommodate the bandwidth requirements many designs set the interface size between the cache and the next level in the memory hierarchy to greater than a word.

(2) Fetch-On-Fault -- A fetch to the next level in the memory hierarchy is initiated on a cache miss and retrieves more than one cache block of data (fetch size > block size). The additional blocks or subblocks of fetched data are sequentially addressed from the cache miss block address. This is probably the most common prefetch strategy and is often assumed to be a part of the fetch strategy instead of

prefetching. Most implementations of this prefetch strategy allow the processor to continue execution before the prefetching is completed (like nbwrf described in Chapter 2).

(3) Tagged-Sequential Prefetching -- A fetch to the next level in the memory hierarchy is initiated on a processor request to a cache block which has been fetched but never accessed by the processor. A prefetch also occurs on a cache miss since this is considered the first access to that cache block. Therefore, tagged-sequential-prefetch is a modification to the fetch-on-fault prefetching strategy, allowing prefetches to occur on initial block accesses as well as on cache misses. The address of the prefetched cache block is the next sequential block address after the block that was accessed. This is the most efficient of the three prefetch strategies but requires a more complex control structure. A zero-level cache implementation using this strategy must be dual-ported to allow simultaneous read and write of the cache's tag and data RAMs.

Of all the methods previously studied, tagged-sequential prefetching provides a reasonable reduction in instruction-cache miss rates while allowing for a manageable and efficient implementation structure. But none of the hardware-driven prefetch strategies surveyed significantly reduce the miss rates for small data caches. The fetch and prefetch strategies studied still require a memory system structure with large caches closest to the CPU. The resulting access times are not optimum for the target processing rate.

Extensive studies have been performed to understand the access patterns for data and instruction references relative to workload and processor type [8, 29, 66, 90]. By understanding the typical access patterns of a processor we created an adaptive prefetching structure, called predictive prefetching, which significantly increases the accuracy of the prefetch. An improved prefetch accuracy minimizes the amount of unused prefetched data, increases the caches hit rate, and reduces the required memory system bandwidth. Predictive prefetching uses a history of processor reference patterns to predict future references. The reference history is stored in the first-level cache tags and updated after every processor reference. The address used during each first-level cache reference is stored in the tags of the previous reference, providing prefetch addressing when each reference is reused. Predicted references are prefetched from the first-level cache (designed for high bandwidth) to the zero-level cache (designed for low

latency). The use of zero-level caches and predictive prefetching provides a memory system with average access times less than half that of a traditional memory hierarchy.

4.2.1 Predictive Prefetching

Predictive prefetching is similar in concept to branch prediction using a branch-target-buffer. A branch-target-buffer stores a history of branch addresses to provide a prediction method for future branch addresses. Hardware branch prediction accuracies have been studied and are well understood [65, 88]. If the hit-rate for correct branch-address prediction is high enough, the average branch penalty is reduced. Johnson [50] suggested the use of the first-level cache tags as a storage medium for the branch-target-buffer. By using the first-level cache tags to store branch prediction information, Johnson was able to significantly increase the amount of branch prediction data, increasing the branch prediction accuracy with a small increase to the total cache size.

We propose an extension to the branch-target-buffer structure used by Johnson to help in the prefetching of both data and instruction references. The processor reference patterns are recorded such that for each referenced address, the temporally next referenced address is stored. The stored reference patterns are addressed so that prefetching information is provided and used before the processor can reference subsequent addresses. The predictive prefetching algorithm studied in our research records the instruction and data reference patterns separately. Therefore, past instruction reference patterns are used to predict future instruction references and past data reference patterns are used to predict future data references.

Our predictive prefetching algorithm, like Johnson's branch-target-buffer, uses the first-level cache tags to store the reference history of the system. Each time a reference is reused, the history of the reference which followed the present reference is used to predict the next processor reference. The amount of active reference history is limited by the number of first-level cache blocks and their lifetime in the cache. This factor thus bounds the effectiveness of the predictive prefetching algorithm. We will show that typical on-chip first-level caches (≥ 8K byte) provide enough tag entries (thus reference history) to support accurate data and instruction prefetching.

Predictive prefetching differs from Johnson's method by using the stored reference history for prefetching all references, not just branch address prediction. This decouples the prefetch mechanism from the instruction decoder. But, if desired, the stored instruction reference history can also support branch prediction, as Johnson proposed.

Another difference is that predictive prefetching occurs for both instruction and data references. Each cache block stores the address (index address) of the temporally next addressed cache block. These addresses are then used by the prefetch unit to prefetch data from the first-level cache to the zero-level cache. Each prefetched zero-level cache block is tagged so that the next prefetch will not occur until the prefetched data block is referenced by the processor. This method of prefetching will be referred to as *tagged-predictive prefetching*. If a prefetch address is not stored for any given cache block, the next sequential address is used for prefetching. Since a sequential address is a full 32-bit address, not a cache-block address, it can be used to prefetch past the first-level cache. This increases the memory system efficiency during cold-starts.

The main goal of predictive prefetching is to provide a simple and effective means of prefetching processor references from the first-level caches to the zero-level caches. With the proper fetch size, block size, interface size, and cache size, predictive prefetching allows a zero-level cache to have a hit ratio similar to a first-level cache but with half the access time. Also, the use of full addresses on sequentially-predicted prefetch addresses enhances the entire memory system effectiveness. The following section gives details of our predictive-prefetching algorithm and its hardware implementation.

4.2.2 Predictive Prefetching Protocol

As mentioned in the previous section, our predictive prefetching algorithm uses the first-level cache tags to store the reference history of the processor. This information is then used to prefetch both data and instruction addresses into the zero-level caches. The reference history is stored as first-level cache index addresses, minimizing the number of address bits per cache block. A 8K-byte first-level cache with four-words-per-block has an index address of nine bits. Therefore, each first-level cache block contains a tag entry for the *prefetch index address* (PIA), which corresponds to the cache reference following the reference of that block, and a *valid prefetch address* (VPA) bit.

The reference history of the processor is updated on each access to the first-level cache. Each first-level cache access is monitored to determine whether it was caused by a prefetch request or a zero-level cache miss. This allows the exact reference pattern of the processor to be saved. Because the prefetch addresses are stored and accessed during the same cycle, the memory cells used to store the reference history must be dual-ported. Because of this, the size and control of the reference history storage is different than the

other first-level cache tag storage. If a prefetch address has not been stored for a given cache block, the next sequential address is used as the prefetch address.

As a first-level cache block is fetched or prefetched, the prefetch address stored with that block is also transferred to the zero-level cache. The zero-level cache contains tag storage for the prefetch information. As with the first-level cache prefetch storage, the zero-level cache must be dual-ported to support simultaneous fetch and update cycles. This dual-porting slightly increases the complexity of the zero-level cache.

Because the zero-level cache is approximately twice as fast as the first-level cache, the processor is cycling at approximately twice the access rate of the first-level cache. Since a prefetch request occurs in parallel with normal processor cycling, each prefetch operation requires two cycles. If a miss occurs in the zero-level cache during the first cycle of a prefetch operation, that operation is aborted (unless the miss address matches the prefetch address). Otherwise, the prefetch can complete without interruption. This operating mode minimizes the stall time required to support a zero-level cache miss. It also reduces the number of prefetched addresses which are unreferenced.

The following describes the key characteristics of predictive prefetching and the memory system required to support it:

1. A first-level cache reference is caused by a zero-level cache miss or a zero-level cache prefetch request.
2. Each zero-level cache miss is immediately followed by a zero-level cache prefetch request.
3. The index address of each first-level cache reference is stored in the PIA entry for one of the two previous first-level cache references. The storage location of each reference address is based on whether it and the previous first-level cache reference was caused by a prefetch request or a zero-level cache miss. Each first-level cache reference (index address) is stored in the tag of the previous reference except when a prefetch reference is followed by a reference caused by a zero-level cache miss. In this case the miss reference (index address) overwrites the entry stored for the previous prefetch reference. This provides an accurate history of the actual processor reference pattern, avoiding reference history storage of unused prefetched references. The following defines the storage procedure for each combination of references:

   a. If the present first-level cache reference is caused by a zero-level cache miss and the previous first-level cache reference was caused by a zero-level cache miss,

than the index address of the present reference is stored in the prefetch tag of the previous reference.

b. If the present first-level cache reference is caused by a zero-level cache miss and the previous first-level cache reference was caused by a zero-level cache prefetch request, then the index address of the present reference is stored in the prefetch tag of the reference before the previous reference (two references back from the present reference).

c. If the present first-level cache reference is caused by a zero-level cache prefetch request, then the index address of the present reference is stored in the prefetch tag of the previous reference, independent of the cause of the previous reference.

Table 4.1 outlines the prefetch index address storage patterns based on the order of reference types. Therefore, each first-level cache tag address must be temporarily latched (based on the type of reference) for use during the next two first-level cache references. The type of reference (prefetch or miss) must be communicated to the first-level cache controller. Dual-porting of the PIA RAM cells helped to simplify the logical implementation of this protocol. Finally, the VPA bit is set valid on each PIA update.

Present L1 Cache Ref. (n)	Previous L1 Cache Ref. (n-1)	Storage Location of Present Index Address
L0 Cache miss	L0 cache miss	PIA of reference (n-1)
L0 cache miss	L0 cache prefetch request	PIA of reference (n-2)
L0 cache prefetch request	L0 cache miss	PIA of reference (n-1)
L0 cache prefetch request	L0 cache prefetch request	PIA of reference (n-1)

Table 4.1: PIA storage based on reference types and their ordering.

4. A prefetch address tag is invalidated (VPA is set invalid) after a first-level cache miss. A first-level cache miss only occurs during full 32-bit address references (zero-level cache misses and sequential-address prefetching). The PIA is not updated and VPA is left invalid until the next first-level cache reference. This allows full-address

sequential prefetching without storing full addresses in the prefetch tags. Full address sequential prefetching supports prefetching past the first-level cache during cold-start conditions.

5. A prefetch request is terminated if a zero-level cache miss occurs before the prefetch request has completed. If the zero-level cache miss address equals the overlapping prefetch request address, the prefetch request is allowed to complete and the data is forwarded to the target register (instruction or data). A prefetch address comparator is required to support this operation.
6. The zero-level cache must also contain PIA and VPA bits for each cache block entry. This information is transferred at the same time data is transferred from the first-level cache. These entries along with a prefetch indicator (PI) bit facilitate tagged prefetching.
7. If a previously unreferenced zero-level cache block is accessed and the its PIA is valid, the PIA is used as the prefetch address for the first-level cache. For an invalid PIA, the PC is incremented by one cache block and used as the prefetch address. A previously referenced zero-level cache block will not initiate a prefetch request on subsequent processor references to that block.
8. All cache entries are invalid during power-on reset or a cache flush operation.

Figures 4.2 and 4.3 show the zero-level and first-level cache tag entries assuming a 8K-byte, direct mapped, first-level cache and a fully-associative zero-level cache, both with 4-word blocks.

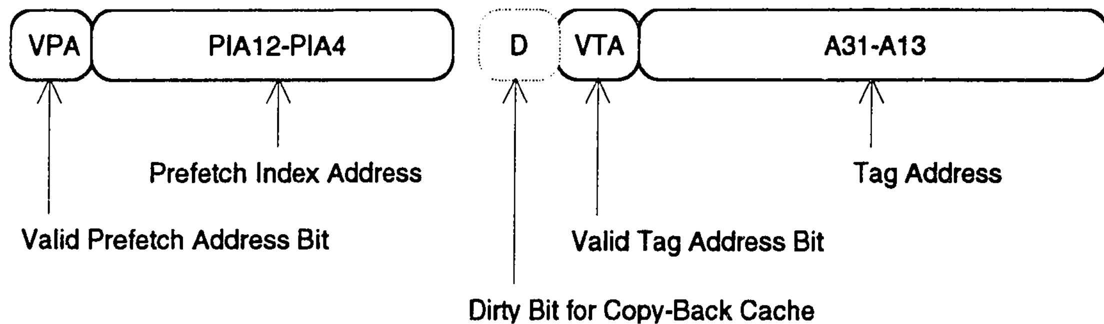


Figure 4.2: First-Level cache tag structure to support predictive prefetching.

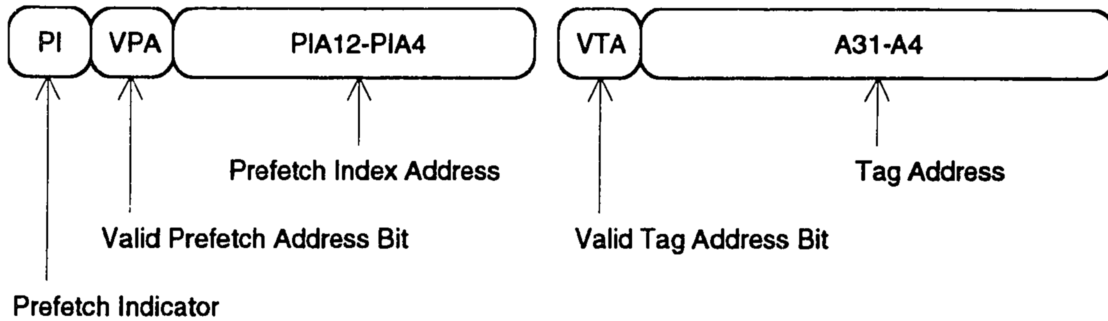


Figure 4.3: Zero-Level cache tag structure to support predictive prefetching

The tagged-predictive-prefetching algorithm is meant to be fully hardware controlled. Assuming an 8K-byte first-level cache and 4-words per block, the increase in first-level cache storage required to save the reference history is approximately 7%. We must determine the effectiveness of tagged-predictive-prefetching and zero-level caches compared to other common memory system implementations. This analysis is performed in the following section.

4.2.3 Analysis Assumptions

The baseline system parameters were chosen to approximate the technology and physical structure for the present generation microprocessor systems. The baseline design is influenced by the baseline configuration used by Jouppi [53] in his study of small fully-associative caches and prefetch buffers. The target technologies are a 0.8um CMOS process [49] and a 0.8um BiCMOS process [49]. These technologies can yield sequencing rates of 100-200MHz. This implies a average memory system access time of less than 10nsec.

The CPU, floating point unit, memory management unit, and first-level instruction and data caches are assumed to reside on the same chip. This level of integration is common in modern microprocessors (Intel i860, Intel i486, Motorola 68040, MIPS R4000, etc.). Figure 4.4 shows the baseline configuration for which all results are compared. Figure 4.1 shows the proposed memory system configuration with zero-level caches. Split first-level instruction and data caches are direct-mapped, yielding the fastest effective access time [40]. The data cache is a copy-back, write-allocate design, while the instruction cache assumes writes are prohibited to instruction storage. The size

of each cache is 8K-bytes. The MIPS R4000 contained similar caches built in similar CMOS technologies. Their resulting access times of 20nsec are insufficient to support the target sequencing rates. The R4000 pipelines internal caches, providing a higher sequencing rate but resulting in a higher CPI rating. Our goal is to avoid increasing the CPI rating while supporting the target sequencing rate.

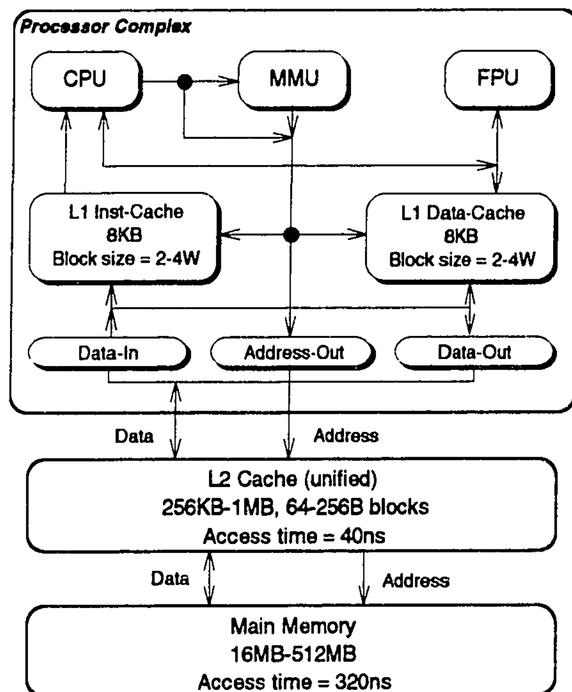


Figure 4.4: Baseline Design

The internal cache block and fetch sizes were assumed to be equal. There are several aspects of the targeted system which dictated the block and fetch size: Hennessy's and Przybylski's studies [37, 79] showing very small miss-rate differences between caches with block sizes of four to sixteen words, the external data interface size, the data transfer size between zero-level caches and the first-level caches, and the maximum size and the maximum number of entries allowed in the zero-level caches. Figure 4.5 shows the abstract tradeoff of block size versus miss rate, memory-access time and memory-transfer time. Figure 4.6 shows the specific numbers generated from Hennessy's study.

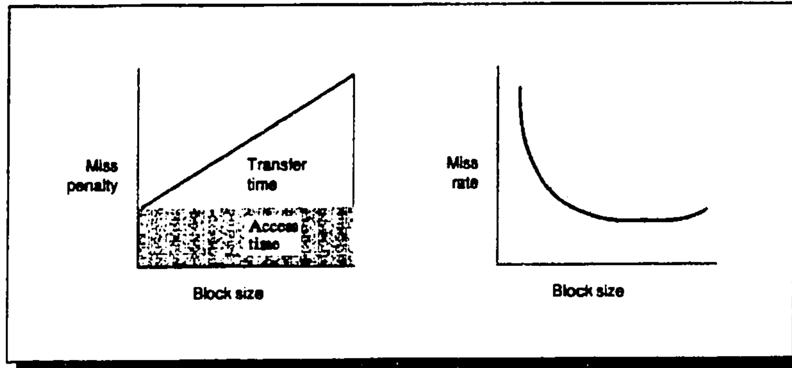
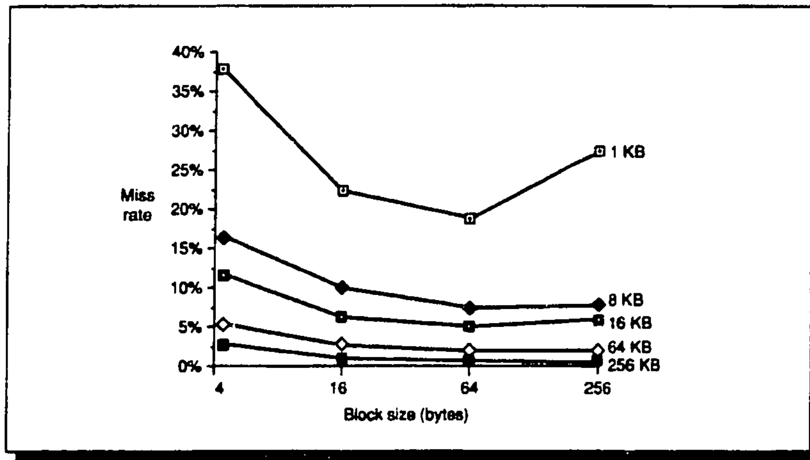
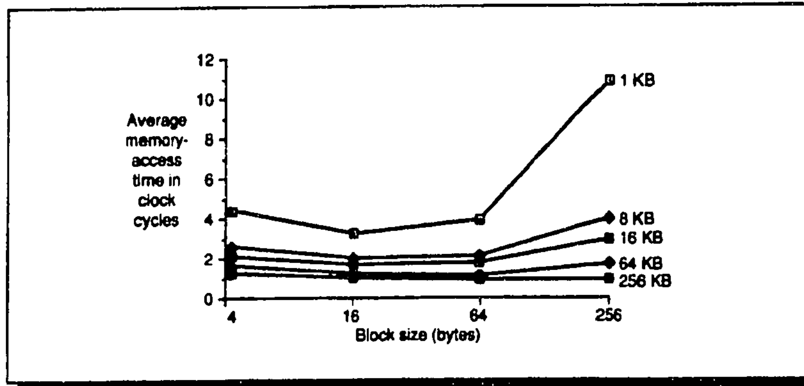


Figure 4.5: Block size versus miss penalty and miss rate [37].



(a)



(b)

Figure 4.6: (a) Miss rate and (b) average access time versus block size [37].

Large block sizes can reduce *cold start* or *compulsory misses*, increase *collision* or *conflict misses*, and increase the cache miss penalty. The third type of cache misses, *capacity misses*, are unaffected by block size. Considering all these factors, a block size range of two to four words seems to be optimal.

The second-level cache is an unified, direct mapped cache. Its size is in the range of 256K to 1024K bytes with a block size of 16 to 64 words. Because of the size of the second-level cache in comparison to the number of unique addresses accessed by the targeted applications, relatively little performance are lost because of second-level cache misses. The second-level cache uses a copy-back, write-allocate write replacement protocol and is addressed by the processor via real addresses. This makes cache coherency possible through the use of a *snoopy* interface [8, 19, 37, 92].

4.2.4 Analysis Technique

The primary tools for the study of memory systems are analytical models and simulations. With the target memory system established (resulting from the targeted average access time and silicon area restrictions of the target technologies) a trace driven simulator was written in C to investigate the effectiveness of the memory system structure. The design space was restricted because of practical implementation factors. The traces used during our evaluation were derived from user reference address traces of six programs running on a R2000 processor. These traces were four RISC machine traces used by Przybylski [78] in his study of cache and memory hierarchy design. They were obtained by using a program, called PIXIE, to add code to each basic block to emit the effective address of each load or store and the start of each basic block.

The applications used to generate the traces were chosen as representative of the reference stream activity on personal engineering workstations. The programs consisted of a edit session (*emacs*), a C compile (*ccom*), a text search (*grep* and *egrep*), a text formatting session (*troff*), a MOS switch-level simulator (*irsim*), and a program that analyzes address traces (*analyzer*). Seven uniprocess traces were generated by capturing one million references following the start point for each program. The RISC traces were synthesized by multiplexing between the selected uniprocess traces so that the appropriate context switch intervals were created. Each trace file was organized to provide approximately 600,000 references for a warm-start initialization of the caches and 1 million references for the actual statistical analysis. Figure 4.7 shows the unique reference activity for the resulting four RISC traces.

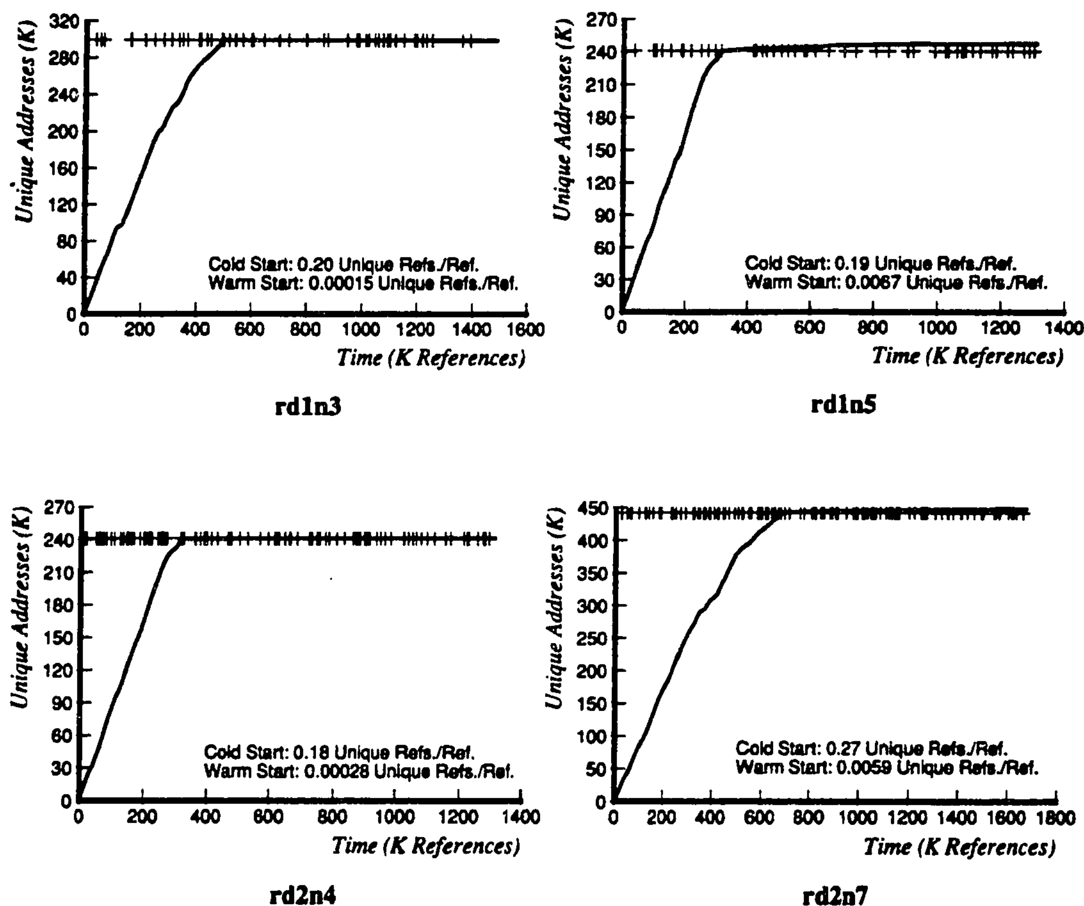


Figure 4.7: Unique references as a function of time: R2000 traces [78].

The number of unique addresses accessed by the resulting context-switched trace files is relatively small but adequate for our analysis of small internal caches. While longer traces [14] of larger programs exhibit significantly different second-level cache miss rates, the use of longer traces add very little to the analysis of internal memory system.

The zero-level cache size was varied from one block to 16 blocks and was not allowed to exceed 256 bytes. The zero- and first-level cache block, fetch, and interface sizes were assumed to be equal. A bypass path allows data to be transferred from the first-level caches to both the CPU and zero-level caches during miss cycles. The external interface is assumed to be two words (64-bits), matching other modern processor interface capacities. The block size was varied between two and four words (an eight word block size was tried with the maximum zero-level cache size to determine its

effectiveness). Tagged-sequential prefetching, tagged-predictive prefetching, a zero-level cache operated without prefetching, and a memory system without a zero-level cache were simulated. The following section describes the results from the trace simulation.

4.2.5 Simulation Results

First we investigated the effectiveness of a zero-level instruction cache with and without prefetching. Figure 4.8 compares the miss ratios of a zero-level instruction cache with storage capacities of 1 to 16 blocks and blocks sizes of two and four words. Tagged-sequential prefetching, tagged-predictive prefetching, and a zero-level cache without prefetching were simulated. Independent of the block size, both prefetching algorithms yield a near minimum miss rate with a zero-level instruction cache size of three blocks. This result is attributed to the average *basic-block* size of the trace files and the room allotted the prefetcher to store prefetched instructions without destroying useful references. Smith [96] has found that the average basic-block size for most RISC applications is seven instructions, with a mean size of four instructions.

The miss ratios for predictive prefetching were three to four times smaller then the miss ratios for sequential prefetching. The miss ratio for tagged-predictive prefetching with a zero-level instruction cache size of three blocks was 2.5%. In all cases the miss ratio for no prefetching proved to be inadequate, never falling below 25%. A eight-word cache block was tried, resulting in no improvement over the four-word cache block. A four-word cache block always provide a better miss ratio than a two-word cache block and increased the first-level cache transfer bandwidth. The higher bandwidth provided by the four-word block and interface size allowed the prefetcher to stay ahead of the processor request.

The average instruction access time for the memory hierarchy, with and without the zero-level instruction cache, is shown in Figure 4.9. This chart shows the memory system access times normalized to the first-level cache access time, which is one cycle. With tagged-predictive prefetching and a zero-level instruction cache size of three blocks, the average access time is half that of a traditional memory hierarchy without a zero-level cache. There is approximately a 10% difference in average access times when comparing the two prefetching strategies. This illustrates that after a certain point, significant reductions in the miss ratio yield only marginal reductions in average access times.

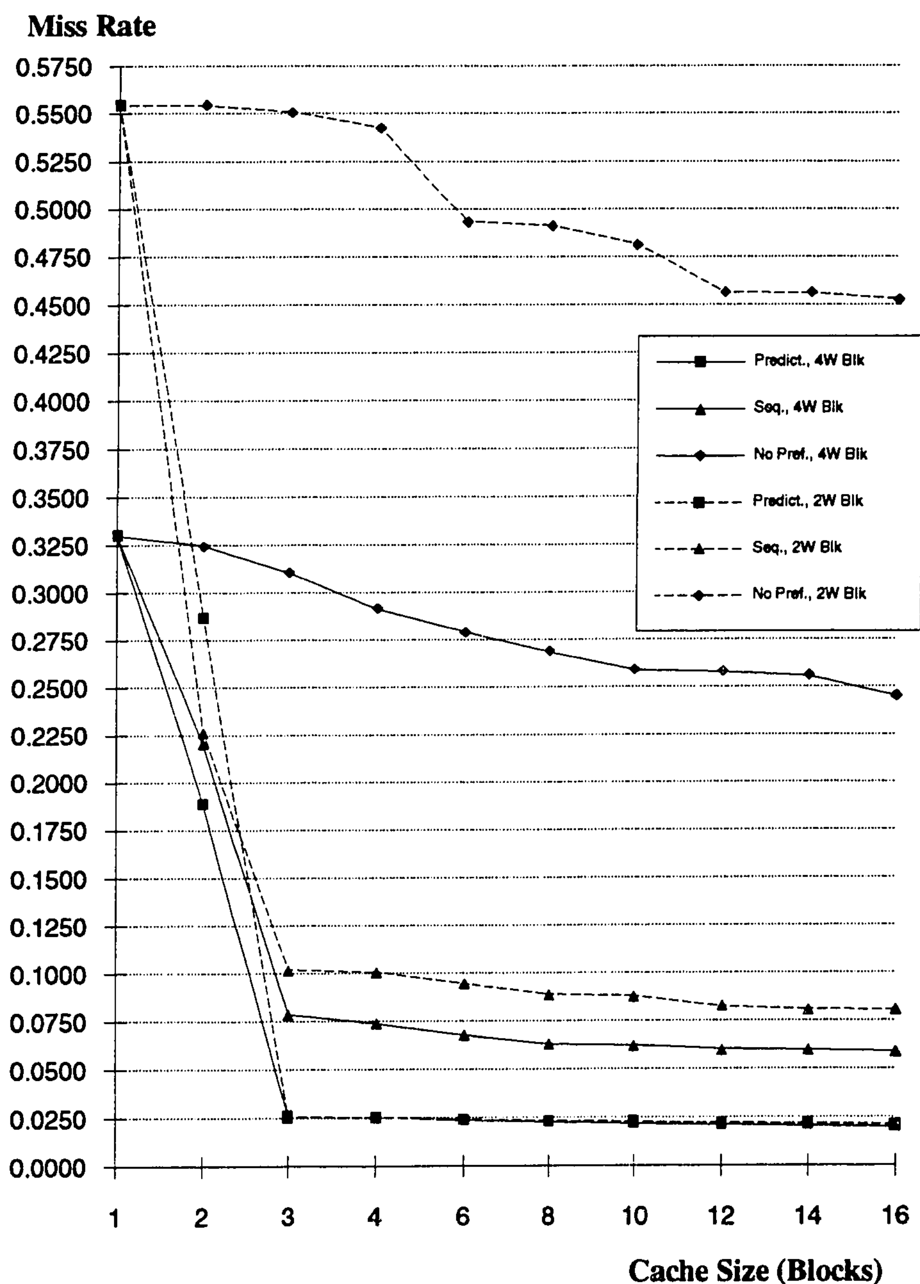


Figure 4.8: Zero-level instruction cache miss ratios.

A predictive prefetching strategy for instruction references will cost more in silicon area than a sequential prefetching design because of the required reference-history storage. But, because the instruction reference history is stored in the prefetch address tag RAM, this information can also be used as a branch-target-buffer for branch

prediction. A comparator is required to guarantee that the predicted address matches the actual requested branch address. The use of the instruction reference history for instruction prefetching and branch prediction minimizes the total hardware cost when branch prediction is required.

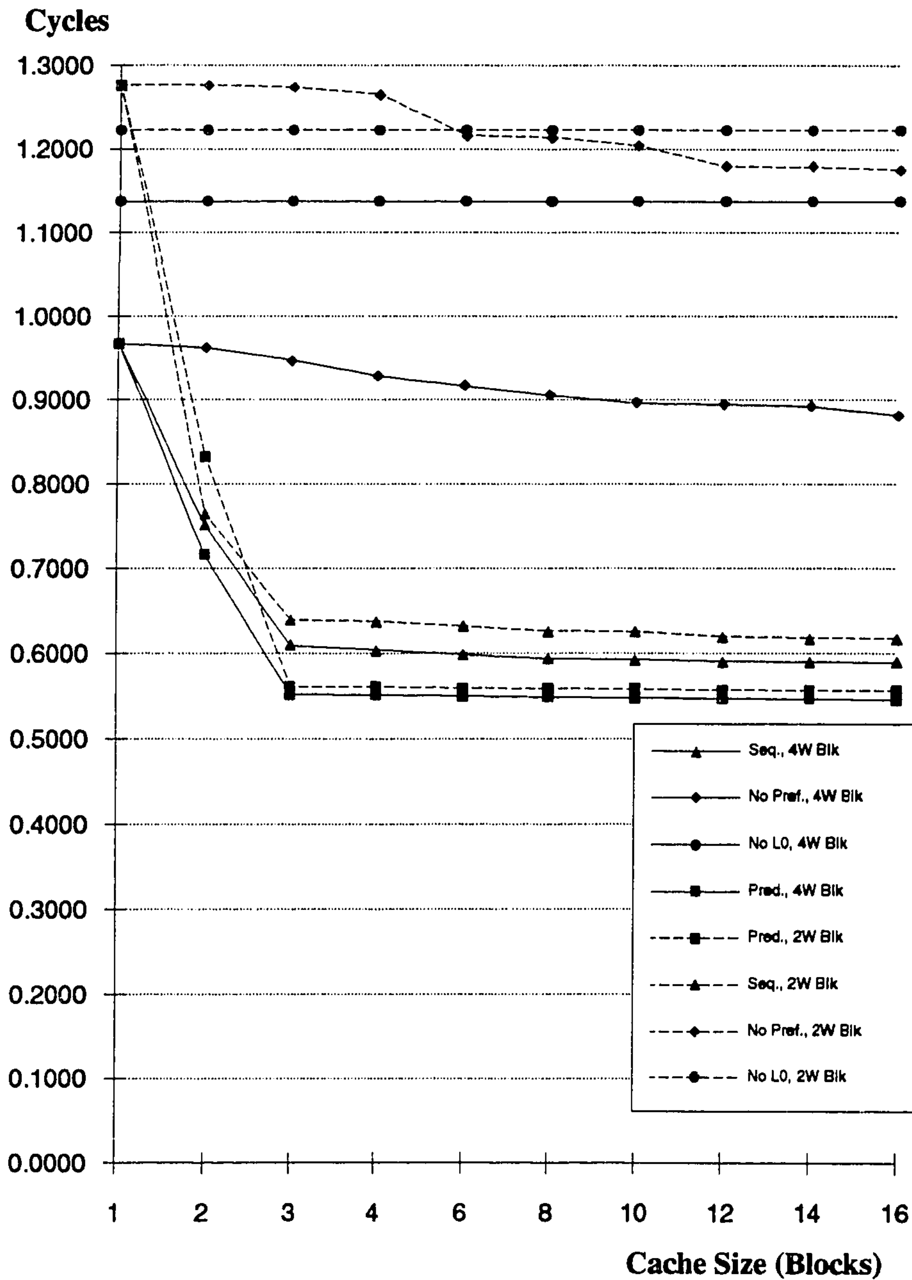


Figure 4.9: Instruction reference average access times.

Another aspect of using a zero-level instruction cache and tagged-predictive prefetching is the *factor-for-unutilized-prefetches* or FUP. This prefetch parameter refers to the percentage of prefetched cache blocks never accessed by the CPU. The higher the FUP, the less efficient the prefetch algorithm. This unit of measure gives a good indication of the amount of first-level cache bandwidth wasted by the prefetch algorithm. Figure 4.10 shows the FUP for the zero-level instruction cache for both tagged-sequential prefetching and tagged-predictive prefetching. The graph shows that less than 8% of the blocks prefetched using predictive prefetching were never accessed by the CPU (with a zero-level cache size greater than three blocks). Sequential prefetching never provided a FUP of less than 22%.

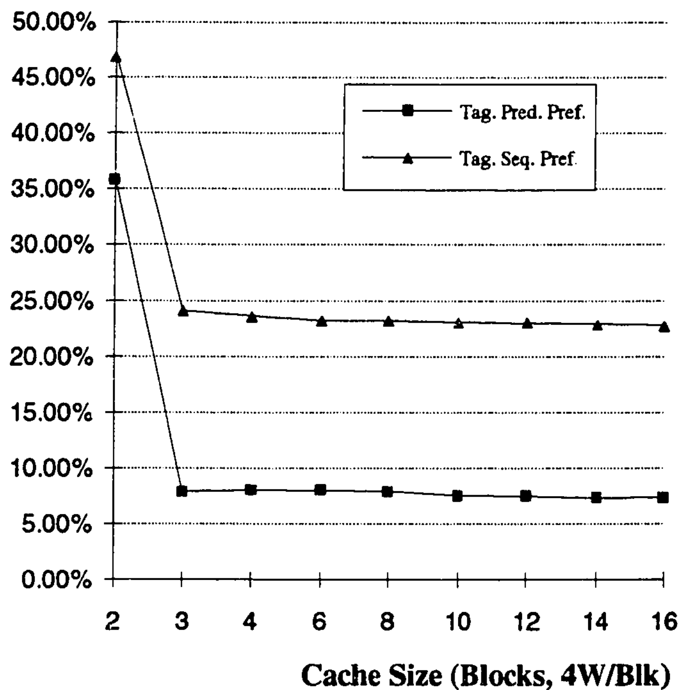


Figure 4.10: Factor-for-Unutilized-Prefetches (FUP) for zero-level instruction caches.

Therefore, tagged-predictive prefetching provides an efficient hardware driven prefetch method for reducing the miss rate of zero-level instruction caches. Our measurements show that approximately 70% of the available first-level cache bandwidth was required to support the use of a zero-level cache with tagged-predictive prefetching. The resulting average-access time for instruction references was 1.1 times the zero-level

caches access time and half the access time of a traditional (non-pipelined) memory system.

For the data reference stream, predictive prefetching proved to be more effective than the other fetch protocols. Figure 4.11 compares the miss ratios for the zero-level data cache structures studied. A miss ratio of 12.75% was observed for a 16 block, zero-level data cache using four-word blocks and tagged-predictive prefetching. This was half the miss ratio of the same size cache using tagged-sequential prefetching. The block size had more of an effect on the miss ratio than was observed for the instruction references. The miss ratios for the zero-level cache with a two-word block were up to 60% greater than when using a four-word block. A eight-word block was simulated with a zero-level cache size of 256 bytes (maximum allowable size) and tagged-predictive prefetching. The miss ratio when using a eight-word block was 13.1%, slightly greater than for a four-word block.

Figure 4.12 illustrates how effective a zero-level data cache with predictive prefetching is in reducing the average access times of the data fetches. Tagged-predictive prefetching requires approximately half the average access time of the traditional memory system. Figure 4.13 shows the FUP for the data memory hierarchy when using tagged-sequential prefetching, tagged-predictive prefetching, and tagged-predictive prefetching only on read references. The data prefetch efficiency when using predictive prefetching for data read and write cycles proved to be the best, although significantly less than that achieved for the instruction stream. But the predictive prefetch efficiency was sufficient to support the data reference frequency of the simulated traces. The memory performance for data references using the zero-level data cache and tagged-predictive prefetching is twice that of the memory system without these enhancements.

Figure 4.14 combines the results of the instruction and data simulations to yield a total memory system CPI. This data was generated assuming that both the instruction and data zero-level caches were the same size and used the same fetch algorithm. The results show that with zero-level caches added to the memory hierarchy using tagged-predictive prefetching, the performance of the memory system can be doubled. Actual implementations of zero-level caching would probably use different sizes for the instruction and data caches. A four-block zero-level instruction cache and a 16-block zero-level data cache, both having a block size of four words, provides an efficient implementation with minimum hardware cost.

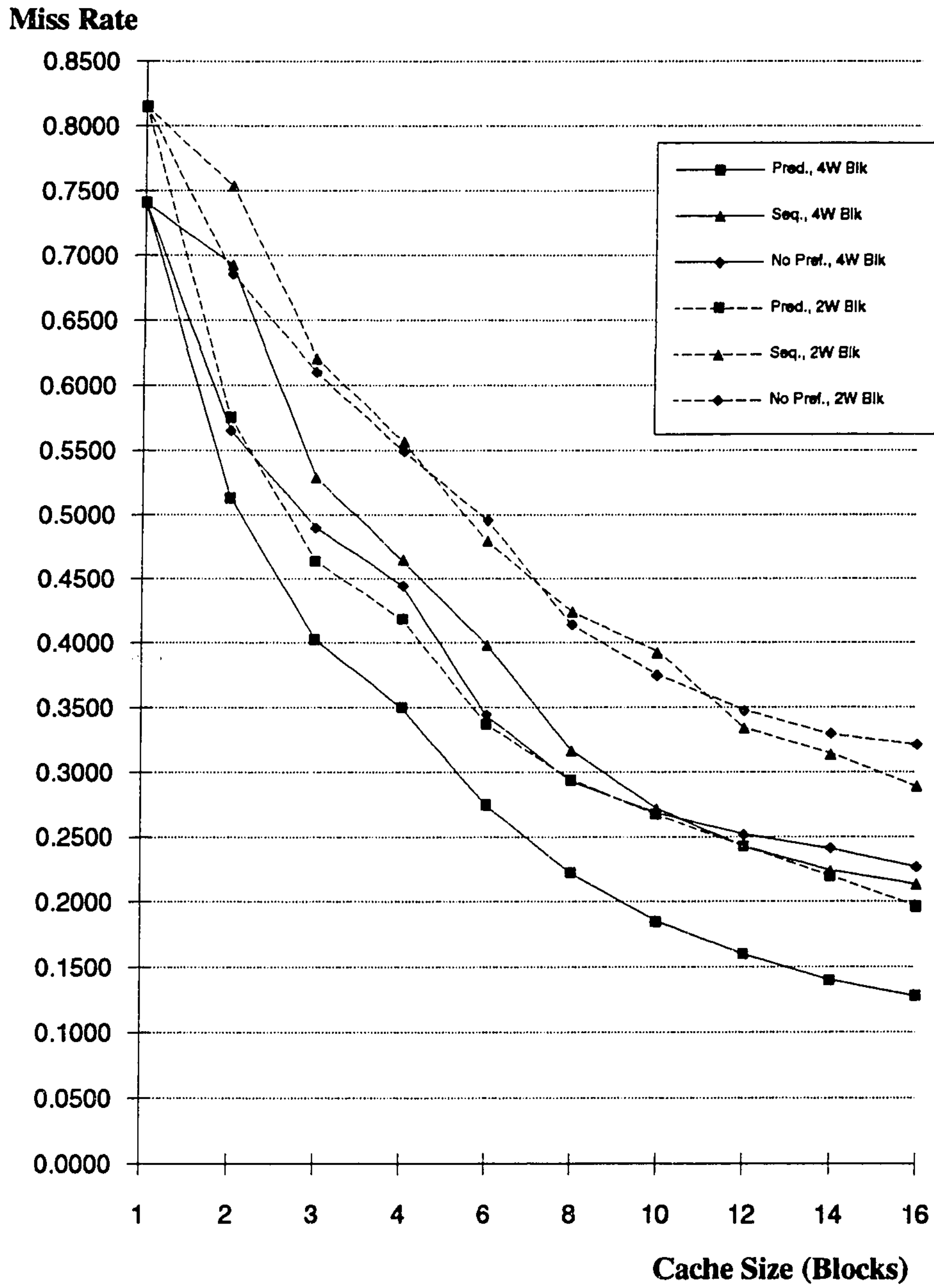


Figure 4.11: Zero-level data cache miss ratios.

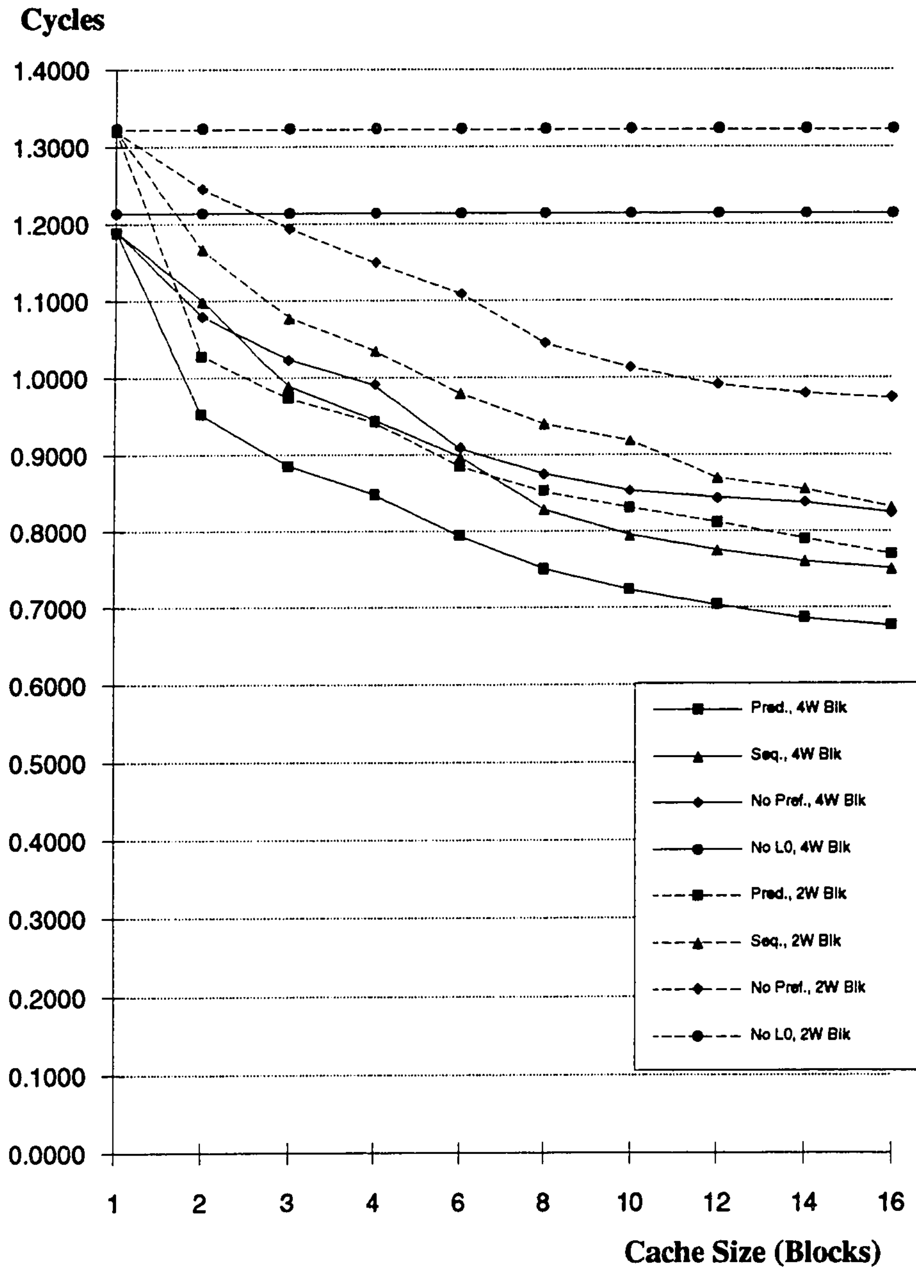


Figure 4.12: Zero-level data cache average access time.

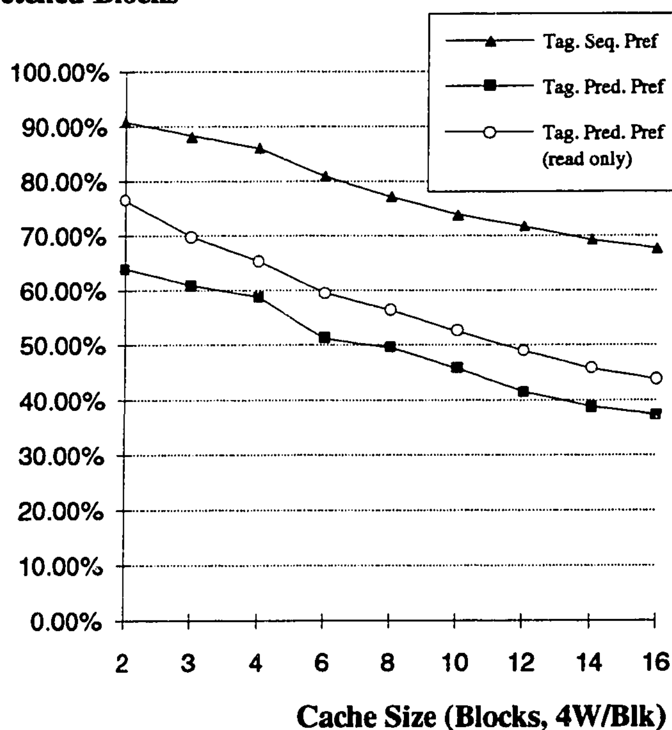


Figure 4.13: Factor-for-Unutilized-Prefetches (FUP) for zero-level data caches.

We can compare these results with a similar memory hierarchy using pipelined first-level caches, which also reduces the cycle time of the memory system. The MIPS R4000 uses this type of cache structure. When compared to a non-pipelined memory system, both zero-level caching and pipelined caches provide approximately twice the memory system CPI performance (0.6 versus 1.2). But pipeline caches increase the number of pipe stages, increasing the number of load and branch delay slots. The CPI load and branch penalty for the R4000 is 0.42, while the CPI load and branch penalty for the R3000 is 0.11. Therefore, a traditional five-stage RISC processor using zero-level caching and predictive prefetching will outperform a eight-stage RISC processor using pipelined first-level caches by approximately 20% (assuming equal cycle times and a memory system CPI of 1.2 for both configurations).

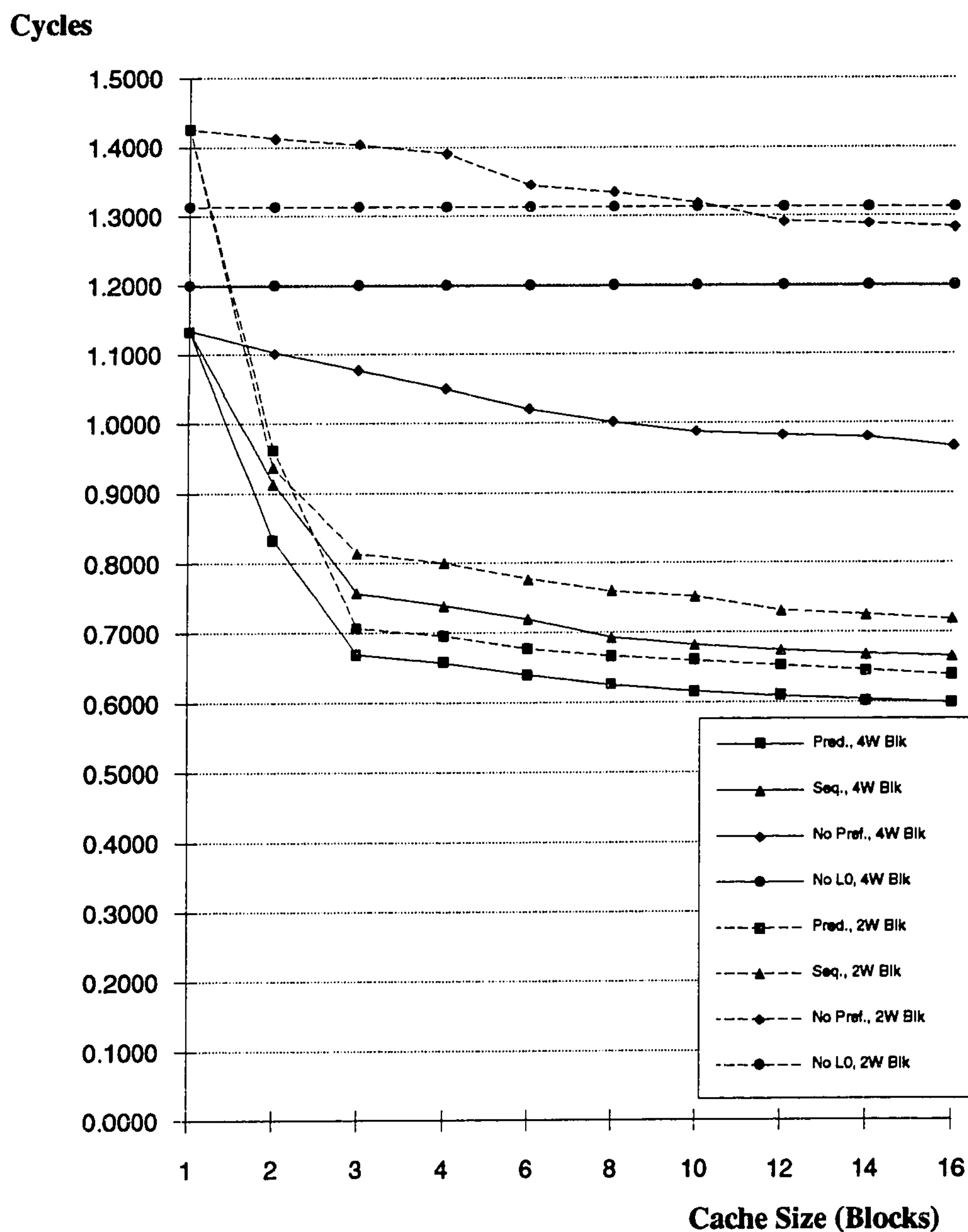


Figure 4.14: Memory system relative CPI using zero-level caches.

4.3 Support Hardware

Prefetching places special requirements on the structure and operation of the memory system. This section describes the hardware structures used in the internal memory system to support tagged-predictive prefetching. First, the zero-level cache structures and their operation are discussed. Next, a description of the first-level cache structure used in our implementation is given. Finally, an evaluation of the internal cache access times is provided. The signal naming convention used throughout this section is given in Appendix B.

4.3.1 Zero-Level Caches

As previously described, zero-level caches are small fully-associative caches placed between the CPU and the first-level caches. Their design was driven by the fetch and prefetch requirements of the instruction stream of the processor. Zero-level caches are unique in that they simultaneously support a processor read reference and a prefetch update operation. Since instruction fetches occur 100% of the time, dual-porting is used to support the bandwidth required by these parallel operations. Both data and tag RAMs of the zero-level cache must be dual-ported. Therefore, zero-level caches are designed to operate like a fully-addressable prefetch buffer and support tagged-predictive prefetching.

Zero-level caches are fully-associative, minimizing the number of cache entries required for a given miss rate. By limiting their size to a maximum of 16 cache blocks, their complexity and access time are minimized. The replacement algorithm also varies with cache size to reduce complexity. A LRU algorithm is used with cache sizes less than or equal to four blocks and a random replacement algorithm is used for larger caches. It is also important to structure the interfaces between the zero-level and first-level caches to support the high bandwidth required by the prefetch algorithm and the improved execution rate of the CPU. Therefore, the zero-level and first-level cache block, fetch, and data interface sizes are equal, optimizing their block transfer rates. Finally, to simplify cache coherency and optimize write performance, the zero-level data cache uses a write-through policy with write-buffers.

Figure 4.15 gives a block diagram of the zero-level instruction cache (the data cache has a similar configuration with the Program Counter Unit, Instruction Prefetch Unit, and Instruction Register replaced by the Memory Address Register, Data Prefetch Unit, and

Memory Data Register, respectively). The zero-level caches are made up of two units, the Tag Unit and the Data Unit. The Tag Unit provides cache hit/miss indication signals, tag prefetch control signals, and select signals for the Data Unit. A content-addressable memory (CAM) is used for the tag storage and a tag-access tracking cell is used to provide self-timed control. Figure 4.16 shows how the tag storage cells are constructed and Figure 4.17 shows the tag tracking cell.

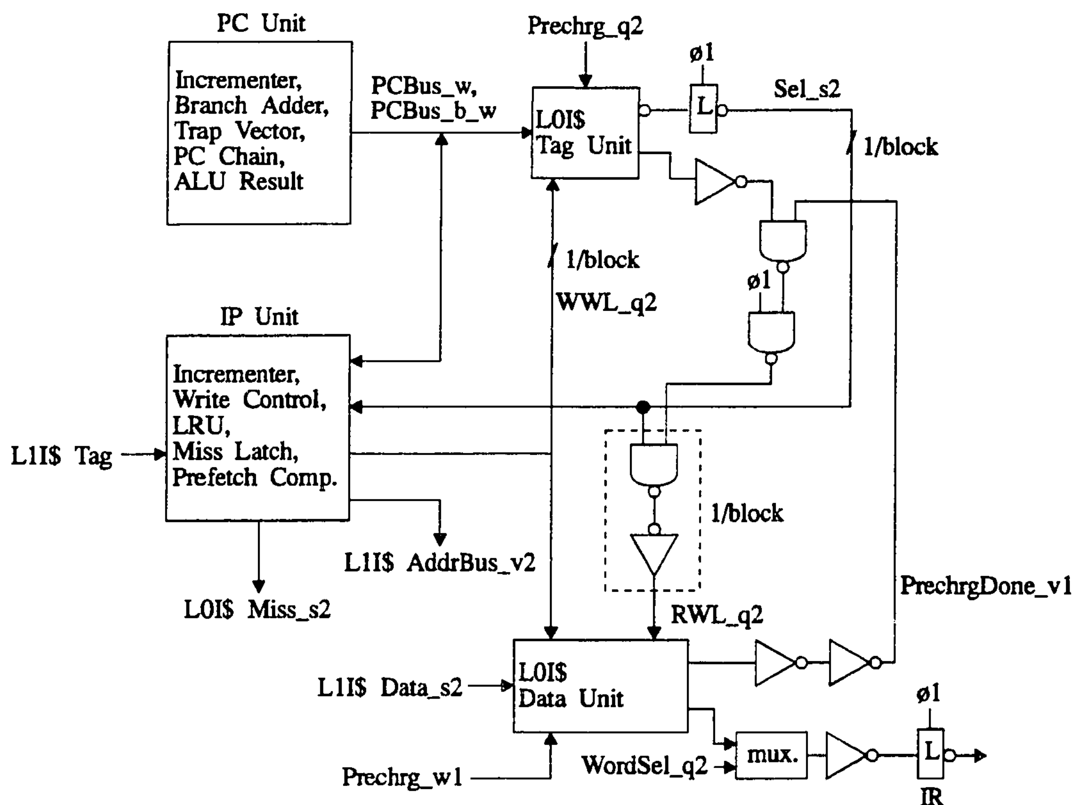


Figure 4.15: Zero-Level Instruction Cache block diagram.

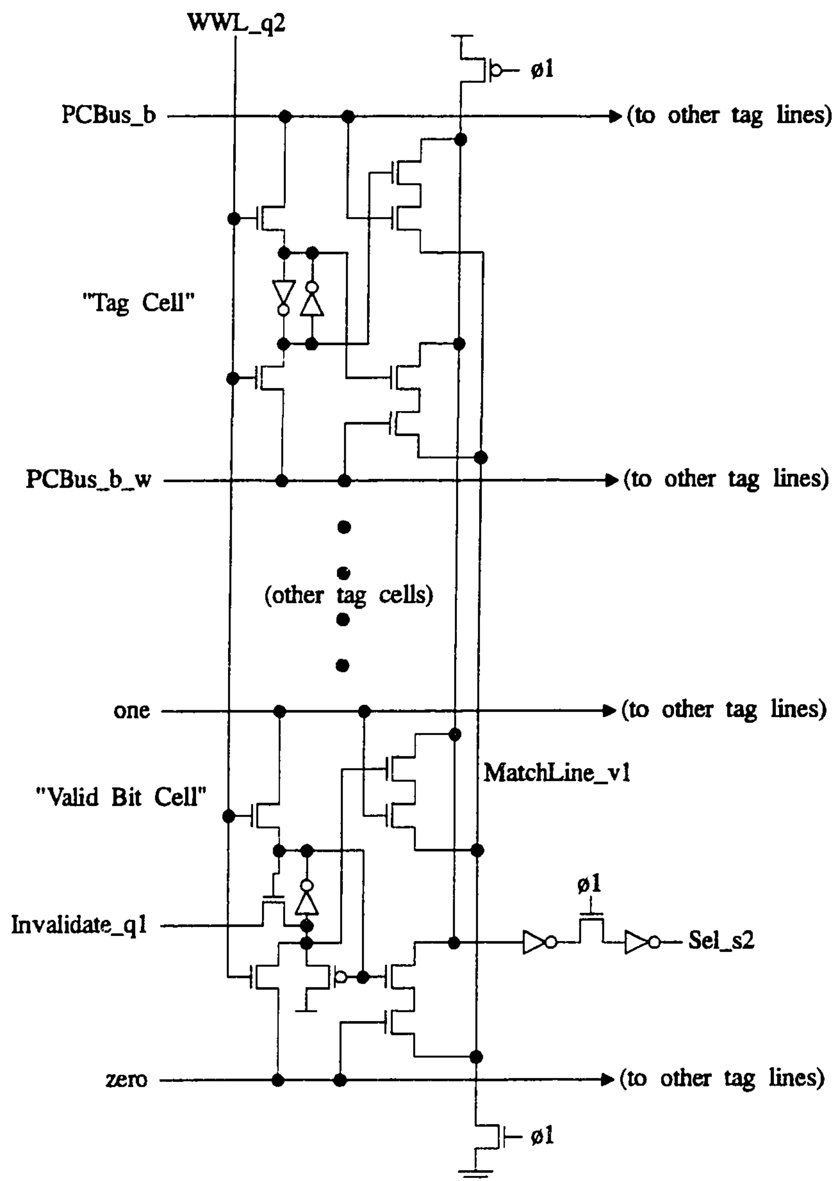


Figure 4.16: Zero-level cache tag and valid-bit CAM cell.

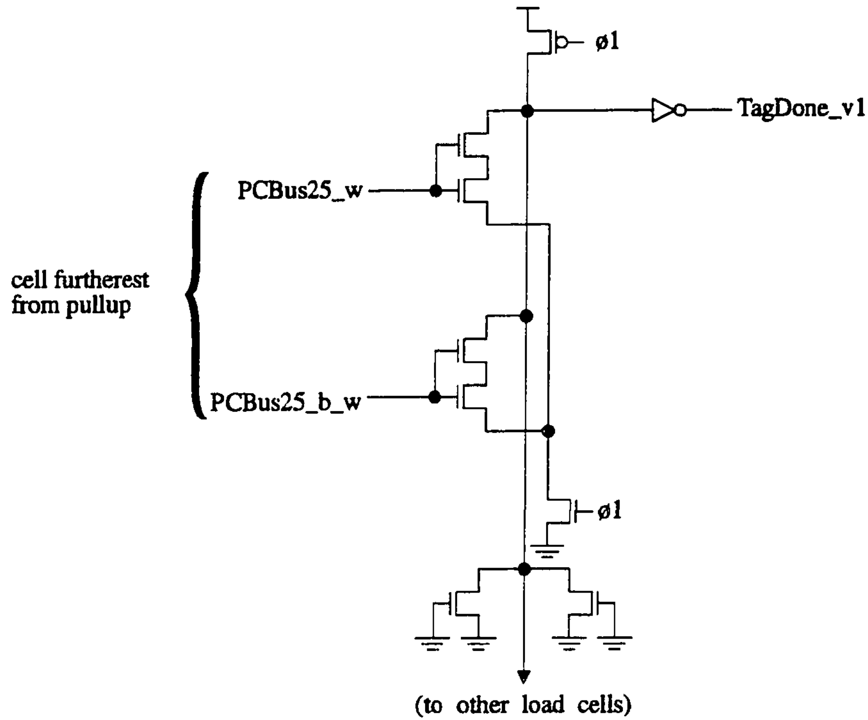


Figure 4.17: Zero-level cache tag tracking cell.

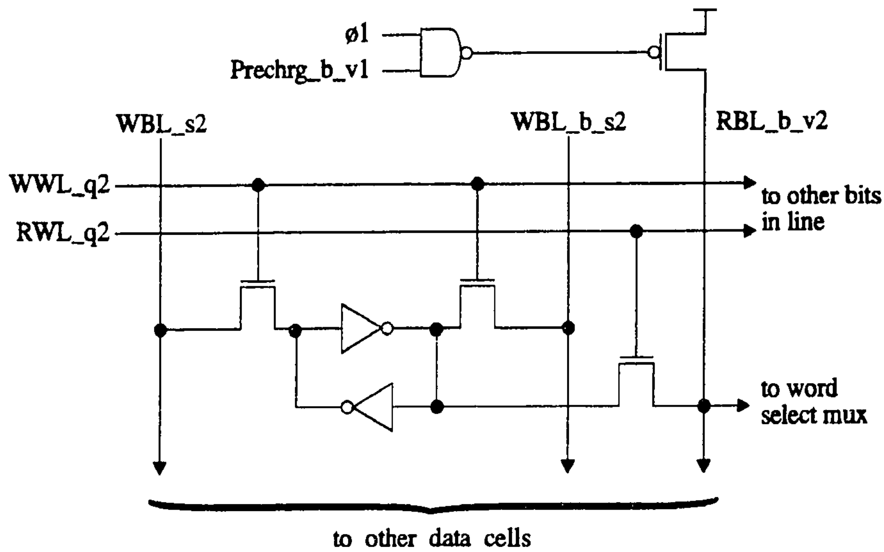


Figure 4.18: Zero-level cache data RAM cell.

The Data Unit provides cache data storage, tagged prefetched address storage, and select control signals. To support simultaneous read/write operations, the Data Unit storage cells are dual-ported RAMs. If the prefetch request matches the CPU reference request, the data is also bypassed to the target CPU register. Figure 4.18 gives the basic Data Unit RAM cell structure. A bitline-precharge tracking cell is also required, providing self-timed signalling for read-bitline precharging. This optimizes cache read performance and isolates the cache's operation from the clock duty cycle. Figure 4.19 shows the structure of the bitline-precharge tracking cell.

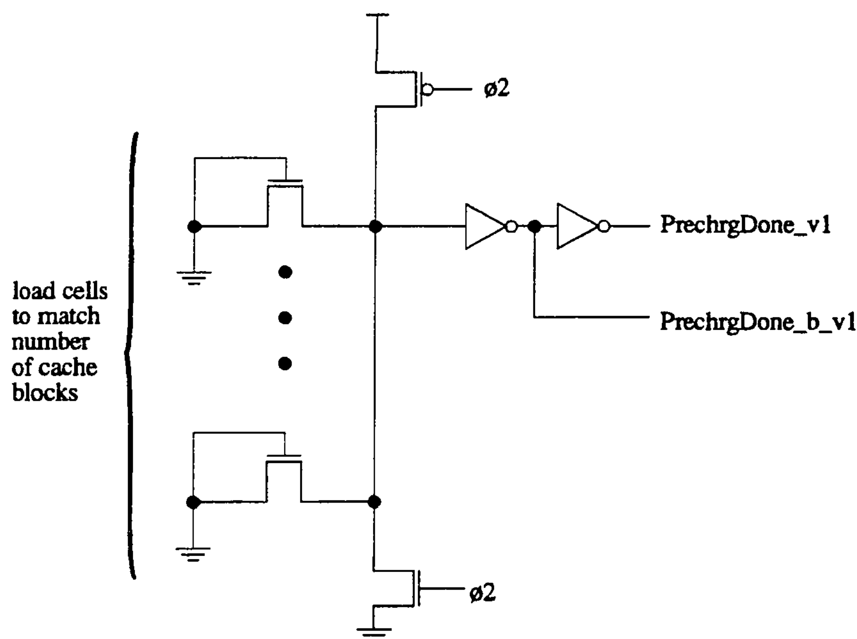


Figure 4.19: Zero-level cache bit-line precharge tracking cell.

4.3.2 First-Level Caches

To provide the instruction and data bandwidth required by a RISC processor operating at greater than 100MHz, internal first-level instruction and data caches are employed. Their on-chip location decreases their access time (no chip boundaries) and allows for a wide data path (interface size equals block size). Both factors are important to support the prefetching algorithm required for effective operation of the zero-level caches. To

minimize the first-level cache miss rate they are built as large as the CMOS technology will allow (assumed to be 8K bytes for instruction and 8K bytes for data). The first-level caches are also assumed to be direct-mapped, providing the simplest and fastest cache array. Figure 4.20 gives a block diagram of the first-level cache array.

The first-level cache address is driven from the Prefetch Unit. The Tag and Data RAMs are constructed using single-port, six-transistor fully static RAM cells. During a read operation the bitlines are precharged at the beginning of the cycle by a self-timed precharge-control circuit similar to the one used by the zero-level caches. Reads to the first-level cache can be caused by: a zero-level cache miss, a prefetch request from the Prefetch Unit, or a snoop operation to a dirty cache block (data cache only). First-level cache write request occur during a first-level cache miss operation or a write requests from the processor. The first-level instruction cache requires no write policy, since during normal operation, writes cannot occur to cachable instruction memory. The first-level data cache uses a Copyback-Write-Allocated (CBWA) write policy to provide maximum efficiency for data operations.

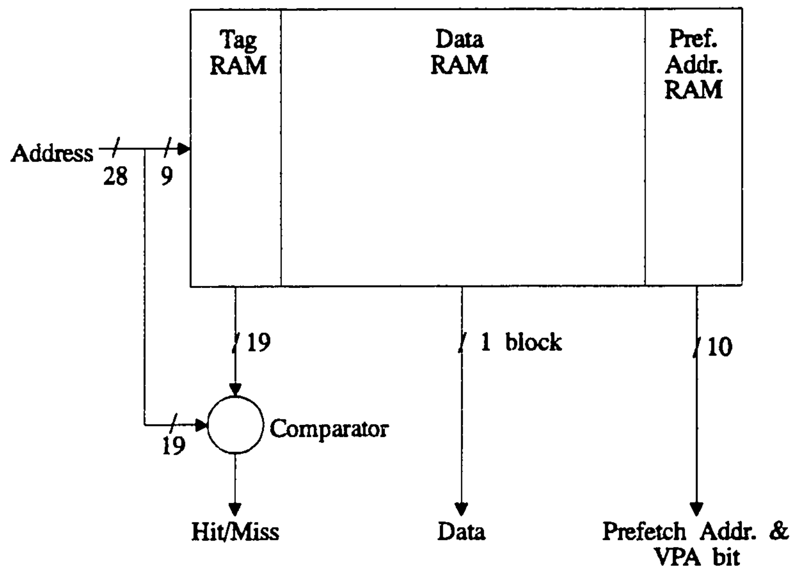


Figure 4.20: First-level cache block diagram.

Figure 4.21 is a block diagram of the first-level cache Prefetch Address RAM array. This RAM array stores the reference history of the processor, providing a means of predicting future references. It is dual-ported to allow the updating of a prefetch address while another is being read. This dual-porting is required since the block being referenced is never the block requiring a PIA update and prefetching utilizes 70% of the first-level cache bandwidth (instruction stream). The remaining 30% would not provide adequate bandwidth for the updates. Single-rail sensing (ratioed inverter) is used on each of the differential bitlines (similar to the structure used in the register file array), simplifying the sense amp and providing true and complement signals to the zero-level cache. Because the other cache tag bits have an additional comparator in their path, the speed of dual-rail sensing was not required. The read bitlines are precharged by the same control signal used in the data array.

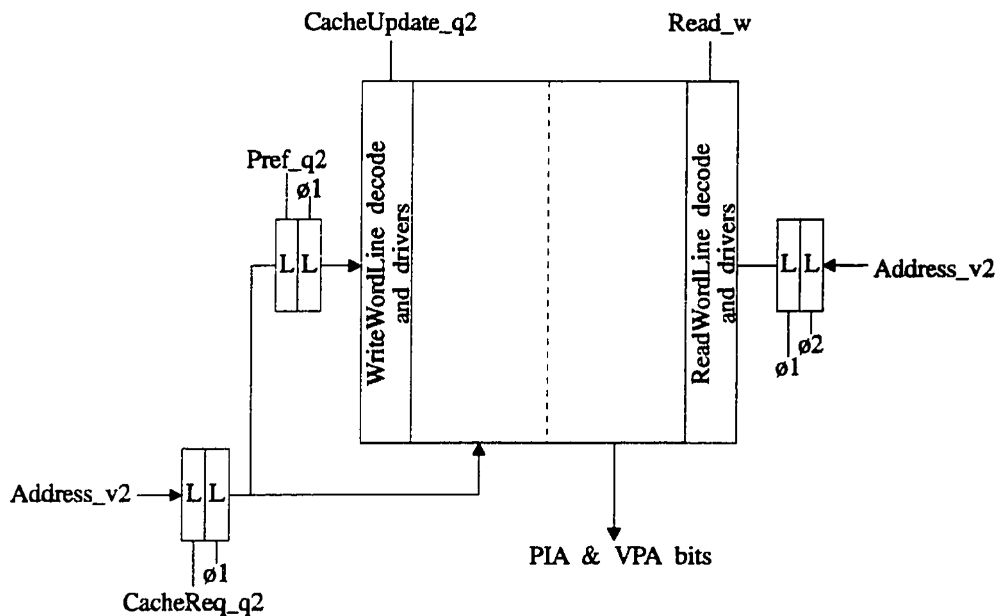


Figure 4.21: First-level cache prefetch address RAM array.

Figure 4.22 is an example timing diagram showing the control signal operation for the prefetch address RAM, illustrating the update protocol. Note that the index address for a first-level cache prefetch access is stored based on the cause of the next access. The present first-level cache prefetch address is stored only if the next first-level cache

access is another prefetch access. Therefore, a prefetch access is stored in the cache block selected by the previous cache access only if the next access is a prefetch access. If the next first-level cache access is caused by a zero-level cache miss, then the index address of that access is stored, instead of the previous prefetch address. This prefetch-index address storing protocol and the use of tagged prefetching allows the CPU's access patterns to be stored precisely. This method of storing the CPU's reference history also allows the information to be used for branch prediction.

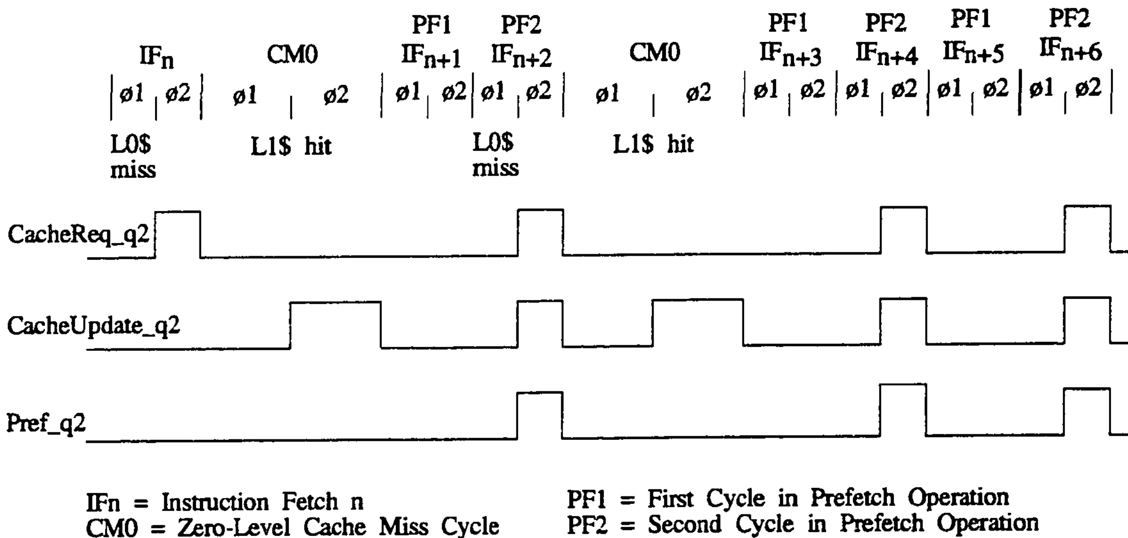


Figure 4.22: Timing diagram for prefetch address tag RAM.

The instruction and data streams are handled independently. The data stream is unique in that it contains both read and write references. It was found that handling data read and write references identically minimized the zero-level cache miss rate, increased the prefetch efficiency, and minimized the average data memory access time. The additional tag bits required to support tagged-predictive prefetching increased the number of bits in the instruction and data first-level cache by 7% (over a simple direct-mapped 8K-byte cache with a block size of four words). The analysis in Section 4.2 shows that the performance advantages provided by predictive prefetching and the fact that the same reference history can be used for branch prediction justifies the 7% increase in cache storage.

4.3.3 Access Time Evaluation

To determine if a zero-level cache containing 16 blocks (4W/Blk) is significantly faster than the target first-level cache (direct-mapped, 8K bytes), a general-purpose circuit simulation program called SPICE was used to analyze the zero-level caches access time. SPICE allowed the zero-level cache to be model as depicted in Figure 4.15. The zero-level cache SPICE model included; (1) the address latches and tri-state drivers in the PC-Unit, (2) the zero-level cache tag unit, data unit, and word-line control logic, and (3) the Instruction Register. The loading of all signals was set to match actual circuit conditions. The CMOS process models used were a 2.0um model based on the MOSIS CMOS process, and a 0.8um model provided by Mark Johnson [49]. Appendix C gives the parameters of all technology models used during our analysis.

All circuit delays will be measured in *gate-delays.* The gate delay of a silicon process is the delay of an inverter with a fanout of four. This unit of measure allows the process to be nomalized out of the circuit delay specification. A gate-delay in the 2um MOSIS process is 0.9ns and in the 0.8um process is 0.45ns under nominal operating conditions. We estimate that a direct mapped 8K byte internal cache subsystem has an access time of 35-40 gate delays, based on the performance of caches in processors built by IBM, MIPS, and Intel.

Table 4.2 provides a list of serial logic elements which make up the critical logic path used to model the zero-level cache reference delay. Table 4.3 list each functional unit in the zero-level cache reference path and the gate delays measured from the SPICE simulation. All simulations were performed assuming nominal process and environmental conditions (5V and 25°C). The logic path delay (measured in gate delays) is measured from the rising edge of the PC Unit control signal to the falling edge of instruction register output. The total delay for a zero-level cache access was approximately 14.5 gate delays. This is less than half the number of gate delays required for a first-level cache access. It is also less than the number of gate delays required for an add operation (18 gate delays) and a compare-and-branch operation (24 gate delays). In the 0.8um CMOS process the zero-level cache access time was 7.5ns.

Functional Units	Internal Logic Elements in Reference Path
PC Unit	Inverter Latch Tri-state driver (including four additional tri-state driver loads)
Tag Unit (CAM)	CAM cell (address signal loaded by 16 CAM cells) CAM bit-line (bit-line loaded by 28 CAM cells with only one cell active) TagDone sense-amp (ratioed inverter)
Control logic for self-timing and data selection	NAND NAND qualified clock word-line driver (NAND + 4x inverter)
Data Unit (RAM)	RAM cell (word-line loaded by 128 RAM cells) RAM bit-line (bit-line loaded by 16 RAM cells with only one cell active)
4-1 MUX	minimum size pass transistor (with three pass transistor loads) + ratioed inverter
Instruction Register	minimum size pass transistor + inverter

Table 4.2: Logic elements in zero-level cache reference path

Functional Units	Logic delay (gate delays)
PC Unit	3
Tag Unit (CAM)	4
Word-Line Control	4
Data Unit (RAM)	2
Instruction Register	1.5
Total	14.5

Table 4.3: Functional Unit gate-delays for zero-level cache reference path.

4.4 Summary

This study shows that the performance of a traditional RISC processor memory system is improved by 100% through the use of small fully-associative caches, called zero-level caches, and a simple hardware controlled prefetching algorithm, called tagged-predictive prefetching. Zero-level caches contain less than 16 cache blocks, yielding access times half that of an internal 8K byte, direct-mapped, first-level cache. To

minimize the miss ratio of these small caches, an aggressive and efficient prefetch algorithm was required. Through the use of tagged-predictive prefetching, the miss ratio of small data and instruction caches can significantly be reduced. For a instruction cache size of 48 bytes (three cache blocks) a miss ratio of 2.5% was achieved. A data cache size of 256 bytes (16 cache blocks) yielded a miss ratio of 12.75% when predictive prefetching was used. Zero-level caches and predictive prefetching halved the memory system cycle time without pipelining the first-level caches and without incurring the increased branch and load penalties that accompany increased pipeline depth. By not pipelining the memory system, the overhead caused by increased branch and load penalties is avoided. A RISC processor using zero-level caching and predictive prefetching is 20% faster than a RISC processor using pipelined first-level caches (all other factors being equal).

The predictive prefetching algorithm studied in our research recorded the instruction and data reference patterns separately. Therefore, past instruction reference patterns are used to predict future instruction references and past data reference patterns are used to predict future data references patterns. Prefetching of data addresses based on instruction references, and the recording structure to accomplish this, was not studied but may provide an alternative to the algorithm described in this paper.

Chapter 5

STRiP Implementation

To determine the feasibility of dynamic clocking we applied the design techniques and structures described in the previous chapters to a RISC architecture. The result was a self-timed RISC processor called STRiP. STRiP's architecture is based on, and binary code compatible with, the Stanford MIPS-X processor. The addition of self-timing to the MIPS-X architecture allows the pipeline to sequence at a rate defined by the pending functional unit operations, the resulting silicon process, and the present environmental conditions. The design changes improved the performance of the processor by 2-3 times.

This chapter details the architectural implementation of STRiP. First an overview of the critical functional units is provided, followed by implementation details of the dynamic clock generator elements. Next we give a brief description of an external interface and exception handling protocol. Finally, we analyze the performance of STRiP and compare the results with other modern processor designs.

5.1 Basic Structure

The STRiP instruction set architecture is identical to the MIPS-X architecture and is pipelined so that one instruction can be issued every cycle. Appendix D gives more details on instruction encoding and execution characteristics. Each instruction is designed to execute through a five-stage pipeline with the following execution pattern:

Instruction Fetch	IF
Register Fetch and Instruction Decode	RF
ALU Operation (add, sub, shift, compare, logical)	ALU
Data Memory Access (load/store operations)	MEM
Register Write-Back	WB

Each pipelined processing stage is divided into two phases, ø1 and ø2, supported by a two-phase, overlapped, dynamically-clocked sequencing method. Because the functional units are self-timed and designed to operate independent of the clock duty cycle, the clock periods are the main timing constraint. Clock phases can be used to start, stop, or extend processing but their length should not be used for timing within the functional units.

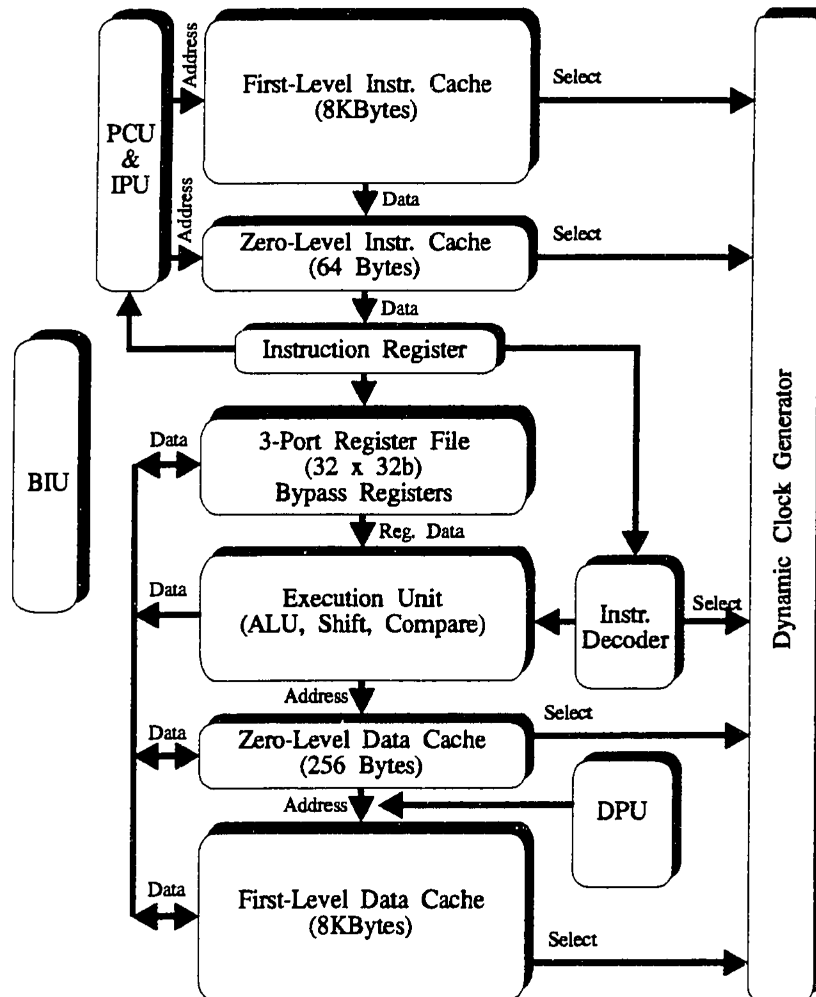
There are several steps required in the development of a dynamically-clocked processor. First, the operation of each functional unit must be optimized, independent of the other functional units. This independent optimization is required since each can be the critical logic path on any given cycle. Next, the critical logic paths and their frequency-of-use are evaluated. From this information the dominant pipeline operations are determined. Operations are selected for tracking within the dynamic clock generator if they can be determined in the cycles before their occurrence. Finally, the dynamic clock generator tracking cells are constructed and the logic generating the select inputs designed.

The hardware implementation of a processor can be divided into seven major sections: the clock generator, the internal memory system (including caches and prefetching logic), the PC Unit, the Register File, the Execution Unit, the BIU (Bus Interface Unit) and the Instruction Register/Decoder. Figure 5.1 is a block diagram of STRiP's functional unit organization. A brief understanding of its key functional units and floorplan is given in the following sections.

5.1.1 Critical Functional Units

The functional unit designs used in STRiP were based on MIPS-X implementations. Only minor changes were required to most of the MIPS-X functional unit designs. The unit requiring the most change was the Execute Unit. The Execute Unit includes a 32-bit ALU, a 64-bit to 32-bit funnel shifter, MD registers that support multiplication and division, and the processor status word (PSW). As in most processor designs, the ALU is

one of the most timing-critical datapath units. The MIPS-X ALU implements all of the logical and arithmetic operations, as well as providing branch comparisons and load/store address calculations. The adder accounts for a significant amount of the ALU's logic delay. Because MIPS-X used precharged logic for its adder implementation (Manchester carry-chain), it was inefficient for dynamic clock operation. A high performance static design was required.



BIU	= Bus Interface Unit		DPU	= Data Prefetch Unit	
IPU	= Instruction Prefetch Unit		PCU	= Program Counter Unit	

Figure 5.1: STRiP Block Diagram

There have been several studies relative to algorithms, implementations, and performance for computer addition [80, 103, 105]. The adder designs considered included carry-look-ahead, conditional sum, carry-select, multiple-output Domino logic, Ling, and parallel. Table 5.1 gives a relative performance comparison of the adder designs considered. A parallel adder [103], modified for efficient operation in a dynamically clocked pipelined, provided the best performance and reasonable complexity for the targeted CMOS technologies. The parallel adder also required minor modifications to provide the other ALU functions, logical and compare operations. The same adder designed for the ALU is used in the Prefetch Units and PC Unit. This approach will insure the tracking of all units which require addition.

Adders	**# of Serial Transistors from c_{in} to S_{32}**	**# of Complex Gate Delays from c_{in} to S_{32}**
Carry-Look-Ahead	21	6
Conditional-Sum	16	5
Carry-Select	18	5
Multiple-Output Domino Logic	18	5
CMOS Ling	14	4
Parallel (Mod. Todesco)	14	7

Table 5.1: CMOS adder performance comparison.

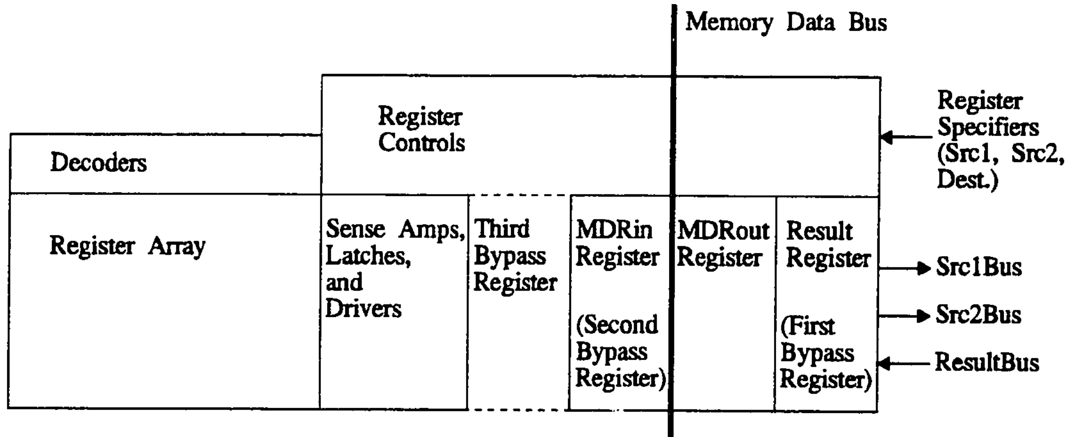
One other element in the Execute Unit required minor modifications. The shifter used in the MIPS-X implementation was a two-stage funnel shifter. This shifter structure proved to be efficient and only required changes to the transistor sizes for optimum operation.

Another critical functional unit, used by every instruction, is the Register File. The Register File provides the operands to the Execution Unit and memory system. It also contains bypass or holding registers which provide computed-but-not-yet-stored values to the instruction stream. The Register File is basically a 32 word, triple-ported (two read ports, one write port) static memory. If the register file is slower then the other critical logic functions, it will dominate the performance potential of the processor.

The main difference between the Register File used in STRiP and the one used in MIPS-X is the ability to simultaneously read and write the register file, instead of time-multiplexing the operations. To accomplish this, the register file required independent read and write bit-lines and word-lines, a self-timed read-bit-line precharge circuit, a self-timed write circuit, and a third bypass register (actually the third bypass register was only required in the BiCMOS implementation since the read-through-register operation was not fast enough). The parallel read/write structure allows reads to start during the beginning of the cycle, instead of during ø2 as in MIPS-X. Performing the register read and write operations in parallel reduced the total Register File cycle time. Figure 5.2 is a floorplan diagram of STRiP's Register File and the memory cell used in the Register Array.

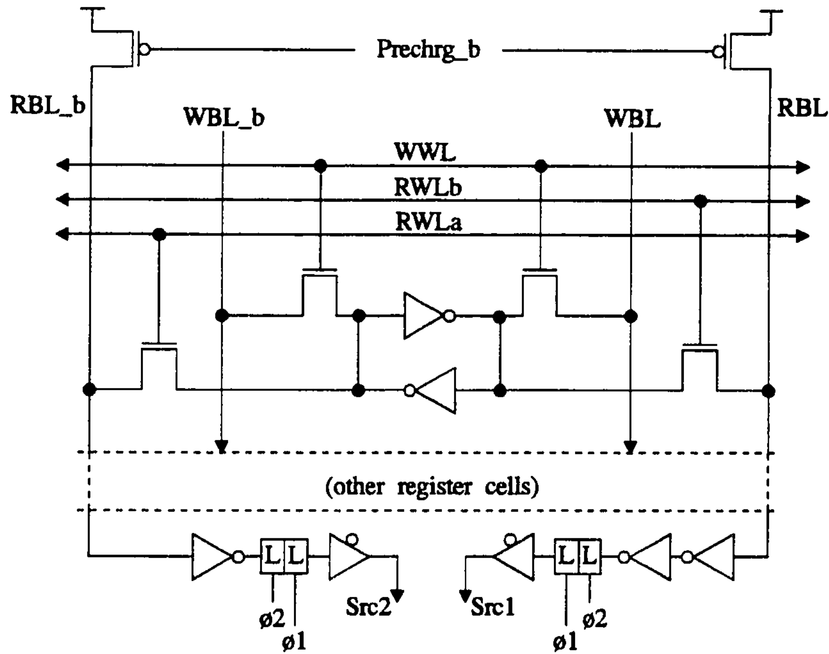
As mentioned in early chapters, the memory system often limits the processor sequencing rate. Since instruction fetches are required during every processor cycle and data fetches 30% of the time, the average memory access time is critical. For this reason STRiP uses the internal memory system described in Chapter 4. Zero-level caching with predictive prefetching will remove the memory system from the critical logic paths. To support this memory subsystem, a finite-state-machine (FSM) handling zero- and first-level cache miss and prefetch cycling is required. The state diagram for the internal memory FSM is shown in Figure 5.3. Both the instruction and data paths require a FSM with these states.

When in the RUN state, the zero-level Tag Unit signals the state machine upon a cache miss or prefetch request. This signalling occurs at the end of ø1. In the RUN/Prefetch state(s) the pipeline sequences normally with a prefetch operation occurring in parallel. The RUN/Prefetch state(s) are entered after a prefetch request, a prefetch hit-compare, or a first-level instruction cache hit. The prefetch operation always requires two cycles. This is due to the speed of the first-level instruction cache versus the minimum processor cycle time. If a zero-level instruction cache miss occurs during the first Prefetch/RUN state and the missed address is different than the prefetch address, that prefetch operation is terminated. This eliminates unnecessary stall time due to the overlapping of prefetch and fetch operations. If a zero-level cache miss occurs during the second Prefetch/RUN state and the missed address matches the prefetch address, the prefetched data is bypassed to the target register.



NOTE: Third bypass register required only when read-through-register operation is not fast enough (BiCMOS STRiP implementation).

(a)



(b)

Figure 5.2: (a) Register File floorplan and (b) Register Array memory cell.

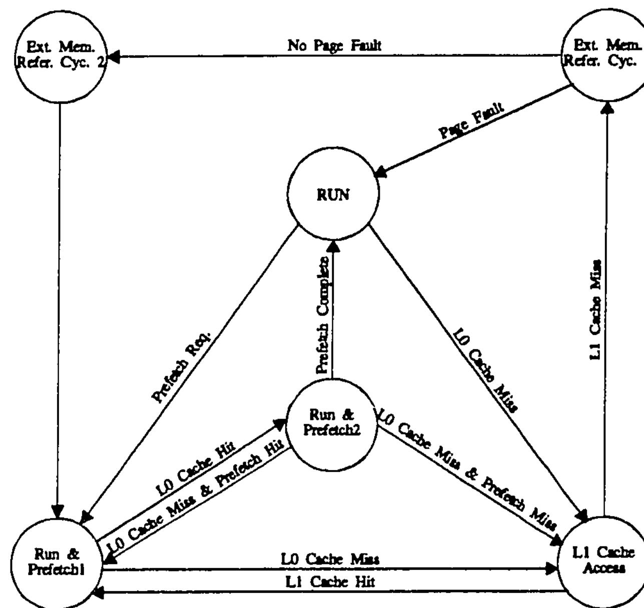


Figure 5.3: State diagram for internal cache miss FSM.

A first-level cache access state is entered on a zero-level cache miss or prefetch miss-compare. A first-level cache access in this state is one cycle, since the dynamic-clock generator adjusts the cycle time to match the cache access time. If a first-level cache miss occurs, the state machine sequences through an external memory request cycle. Note that the machine requires two cycles to fetch information from external memory, similar to MIPS-X. The first external miss cycle reads the cache block from the external memory, while the second cycle writes the block into the zero-level and first-level instruction caches. The machine takes advantage of this by starting a prefetch cycle during the second cycle of the first-level cache miss.

5.1.2 Datapath and Floor Plan

Proper floor planning and datapath design is critical in the efficient operation of the processor. Figure 5.4 shows the hardware resources contained in STRiP's pipeline, the phase on which each latch is controlled, and the major buses that connect the pipelined elements. The main differences between the STRiP datapath structure and MIPS-X is the use of internal data caches, a tri-state result bus (instead of precharged), and the Execute Unit configuration. The prefetch units and first-level caches are not considered part of the main pipeline.

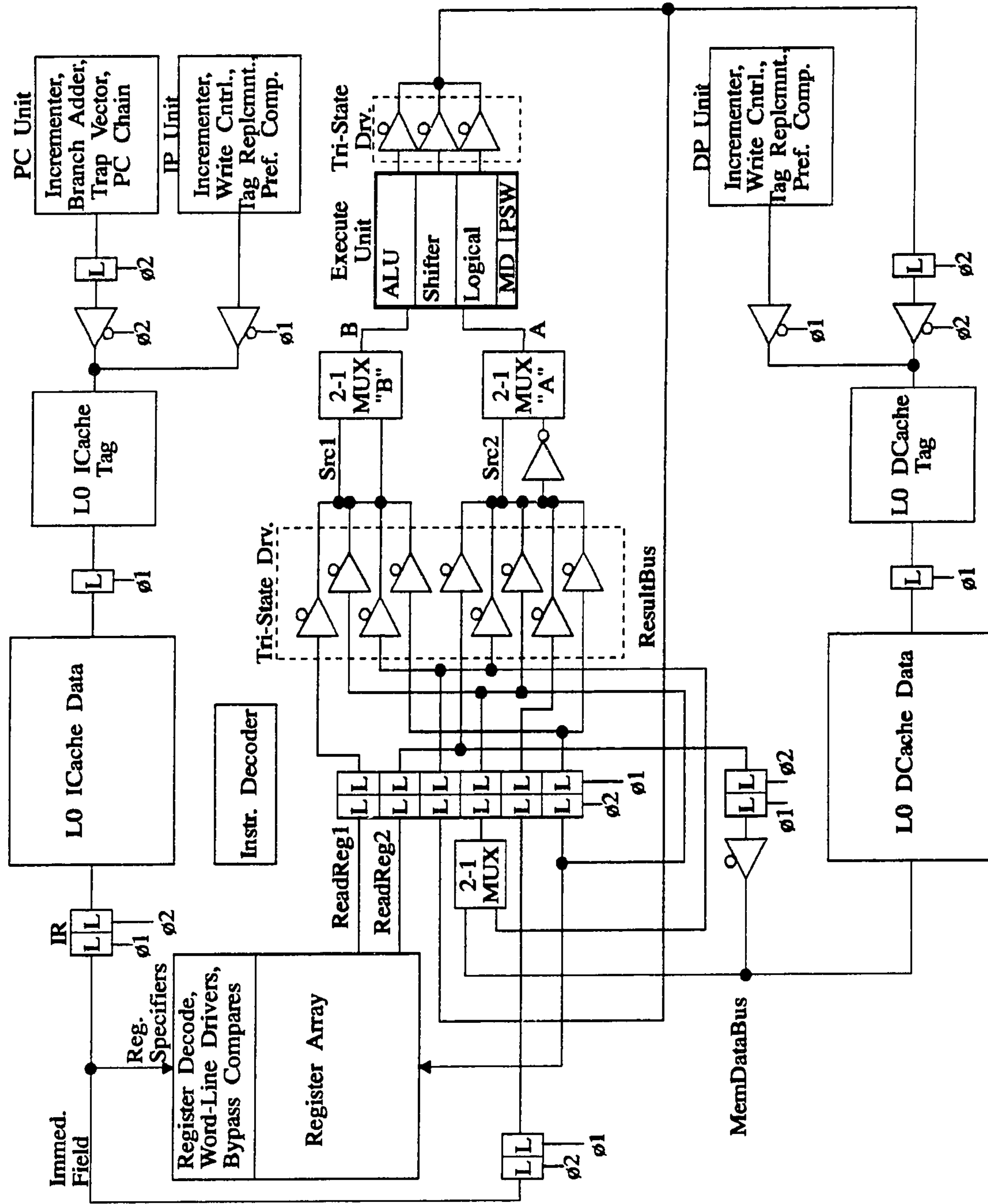


Figure 5.4: STRiP's datapath diagram.

The proposed floor plan, Figure 5.5, gives a good indication of the major bus locations relative to the connected functional units. The *SRC1* and *SRC2* buses are used to read data from the register array, bypass registers, and immediate field and provide that data to the Execute Unit. The *Result Bus* carries data to the bypass registers, the PC Unit (for indexed jump operations), and memory address register (for load/store operations). The *PC Bus* contains instruction fetch addresses generated by the PC Unit. The *Immediate Bus* carries immediate values from IR for use in the PC Unit and the Execute Unit. The *Mem Data Bus* and *Instr Data Bus* carry data from the first-level caches to the zero-level caches during a zero-level cache miss or prefetch cycle.

All of the functions and structures discussed thus far in the chapter provide a general framework of the hardware organization used in STRiP. The next sections describe functional elements which are critical for self-timed operation via dynamic clocking.

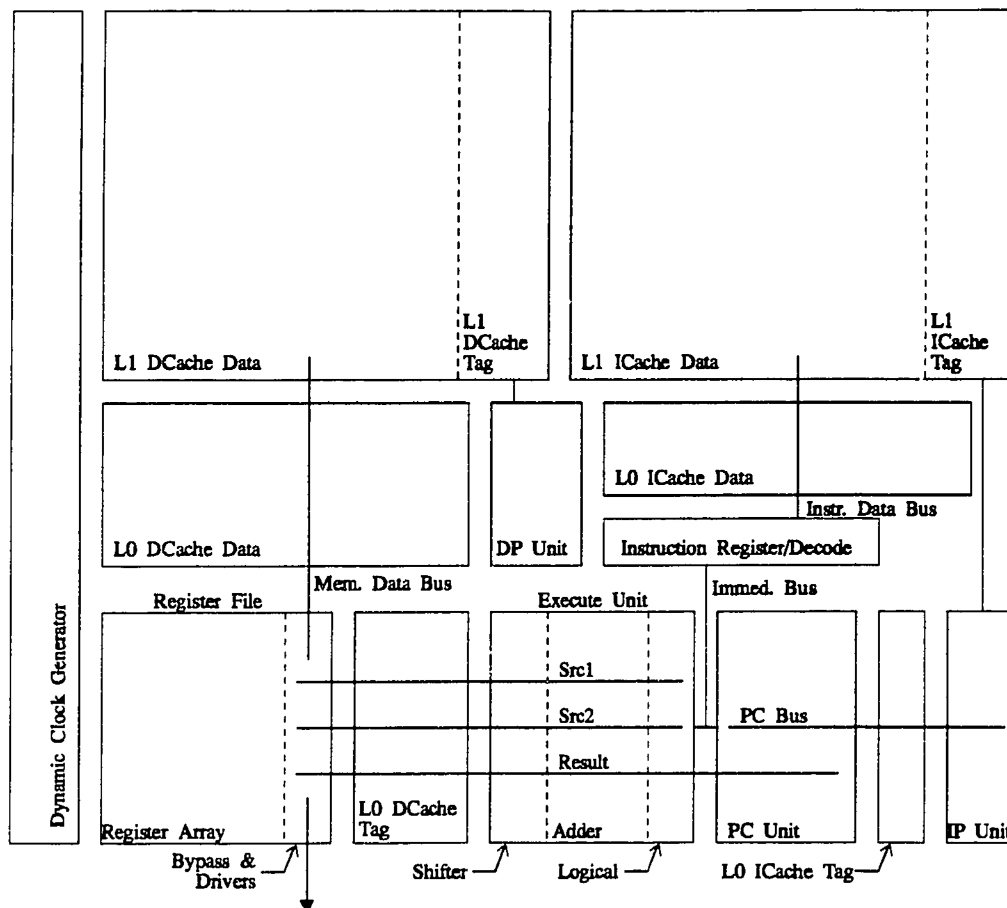


Figure 5.5: STRiP floor plan.

5.2 Tracking Cell Designs

The key elements in the dynamic clock generator are the tracking cells. Each tracking cell must be designed to exactly match the operational delay of the targeted critical logic path. To achieve accurate tracking and optimum performance, the tracking cells incorporate the same type and size of series gates, signal wires, and loads seen in the critical logic paths. Santoro [83] in his self-timed multiplier research found this approach to building tracking elements provided accurate tracking under all environmental conditions. Signal loading is duplicated by using active and passive devices sized to match the total gate load per signal. The length and type of material used to create each critical path signal wire is also used by the tracking cell. Because each tracking cell must be inverting, some tracking cells may contain one less series gate than the actual critical logic path. The C-element delay compensates for the lost gate delay.

To support symmetric output transitions, the organization of the critical path gates may be altered in the tracking cell. Most gates in the STRiP implementation are ratioed to optimize performance and symmetric operation, eliminating the need to rearranged gates for symmetric tracking cell operation. But, in some complex gate designs it is difficult to provide identical transition delays. A simple example of this situation is given in Figure 5.6. In the processor critical logic path the XOR gates alternate in the series gate positions, causing them to switch to the same logic state. If the XORs are asymmetric, this structure reduces the symmetry of this series connection of gates.

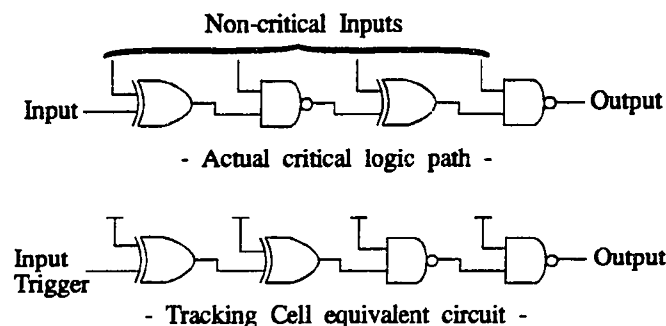


Figure 5.6: Example critical logic path and equivalent tracking cell circuit.

The tracking cell equivalent circuit places the XOR gates back-to-back in the chain of gates. But, the worst-case path may be the XORs switching to the same logic level, resulting in a slower circuit than the XORs switching to opposite levels. Therefore, this rearrangement of gates can eliminate tracking of the worst-case path. If this is the case, addition loading is added to simulate the worst-case delay and still provide symmetric operation. Also, worse-case inputs are used while minimum delay inputs are tied off. This guarantees that for each clock cycle controlled by this tracking cell, the number and type of gate transitions will be identical and worst-case.

Before the tracking cells are designed, the critical operations to be tracked must be chosen. Table 5.2 gives the latency and frequency-of-use for the critical logic paths. The latencies are normalized to gate delays (one gate delay equals an inverter delay with a fanout of four). An instruction fetch occurs on every machine cycle, but the zero-level cache latency is less than the next most frequent operation, addition. An addition is required 65% of the time (100% if PC increment is included). Therefore, the Execute Unit's add operation was chosen as the minimum delay critical logic path in the pipeline. The next longest pipeline operation is a compare-and-branch. It is approximately six gate delays longer than the add operation delay and occurs 15% of the time. A first-level cache access caused by a zero-level cache miss occurs infrequently (approximately 6% of all references) but has delay of roughly twice that of the add operation. Therefore, first-level cache access must be tracked during pipeline stalls caused by zero-level cache misses. The slowest operation is an external transfer. Its delay is indeterminate and ranges from a second-level cache access to an I/O transfer for a device on a system bus. A special tracking cell is designed to monitor these request to external devices, stop the clock when they occur, and restart the clock upon their completion. Therefore, the dynamic-clock generator contains four tracking cells. The following sections give detailed implementations for each of the tracking cells.

5.2.1 Adder Tracking Cell

The adder tracking cell was designed to track the add/subtract/compare operation in the ALU. The ALU add operation occurs during add/subtract, jump, and load/store instructions. Add operations also occur in other functional units for branch address calculation, prefetch address calculation and PC incrementing. The ALU critical logic path proved to be slightly longer than the other logical units performing addition. Figure 5.7 shows the gates and internal loads used to match the ALU add critical logic path.

The complex gates match those gates used in the carry-look-ahead section of the parallel adder design chosen for the critical functional units. The figure also illustrates how the logic was connected in a series. Care was always taken to connect the gate input providing the worse case delay to the series connections in the tracking cell. All other gate inputs are strapped to Vcc or GND.

Critical Operation	**Freq.-of-Use** (per instruction)	**Latency** (gate delays)
Register Access	100%	13
L0 ICache	100%	15
L0 DCache	30%	15
Adder*	65%	18
Compare & Branch	15%	24
L1 ICache	2.5%	45
L1 DCache	4.0%	45
External Cycles	1.0%	100

Table 5.2: STRiP's critical path latencies and their frequency-of-use.

The adder tracking cell consist of eleven series connected gates. These gates, plus the C-element, have a logic latency of 17 gate delays. The critical path of the add operation actually contains an additional lightly loaded result bus latch. The latch was not included in the tracking cell since it would cause the tracking cell to be non-inverting. The C-element delay compensates for the latch delay. To accurately track the source and result bus loading, the wire between the tri-state drivers and the input to the next series gate must be the same length, width, and material type as the actual source and result buses. To improve tracking accuracy, active gates and passive transistors are used to simulate similar loads in the ALU critical logic path.

The adder tracking cell is active on every cycle. Its logic delay sets the minimum sequencing period of the pipeline, thus not requiring a select input. Its output connects to the **A** input of the C-element. The **A** input, by design, provides the least logic delay. This optimizes the clock generator to support the minimum sequencing period, which has the highest frequency-of-use. The *CarryOut* signal is used as an input to the branch tracking cell. This is possible since the compare operation for the branch instruction uses the ALU adder. Therefore, the compare-and-branch critical logic path is identical to the

ALU adder path, up to the *CarryOut* signal. Sharing of logic functions reduces the dynamic-clock generator logic.

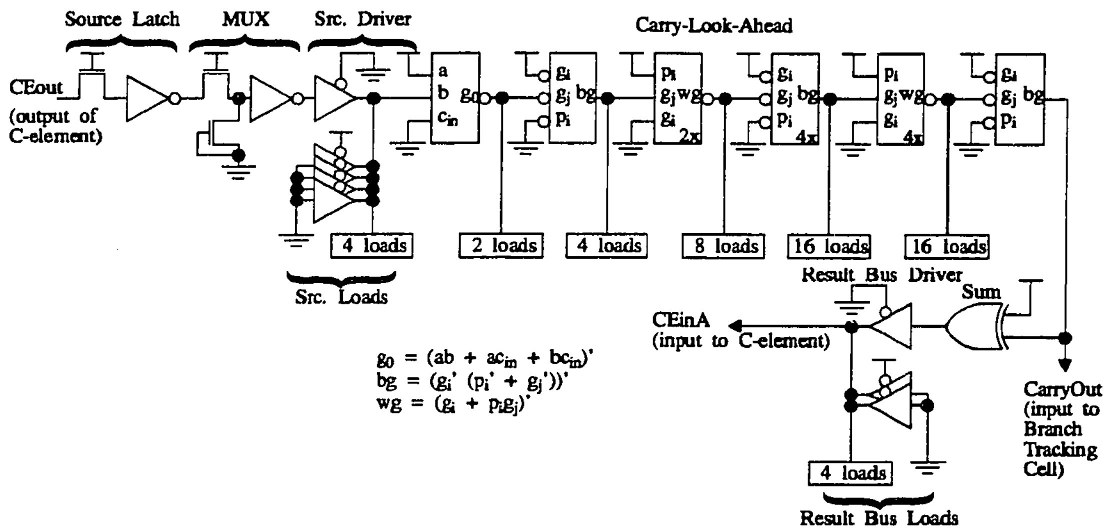


Figure 5.7: Gate level schematic of adder tracking cell.

5.2.2 Branch Tracking Cell

The branch tracking cell tracks the next slowest of the processor pipelined critical logic paths. It also represents the next most frequent operation executed by the processor. The total number of gate delays required in a compare-and-branch operation is approximately 22 (the tracking cell + C-element). The frequency-of-use of a conditional branch operation is 15% [37, 75]. Besides having the adder in its critical logic path, a branch operation also includes additional logic to determine greater-than, less-than, or equal-to. Additionally, the branch logic path contains address selection multiplexers and drivers. Even though the branch-on-equal operation has a faster compare delay (it does not require the full adder delay), it was included in the set of branch operations. To improve performance in the actual design, the branch-on-zero could be eliminate from the decoder selecting the branch tracking cell.

Figure 5.8 shows the gate level implementation of the branch tracking cell. Two 2-to-1 multiplexers are used to select between the full delay of the tracking cell and the set

trigger, which forces the tracking cell to switch early. The tracking cell delay is less than the minimum cycle time when a compare-and-branch operation is not selected. When the tracking cell is triggered for minimum delay, it is set up to respond immediately to the next input change for the next cycle. The selection multiplexers also closely match the pass transistors used in the actual branch critical logic path, adding little additional delay to the optimum cycle time. The XOR gates were rearranged (as compared to the actual logic path) to provide symmetric operation of the tracking cell.

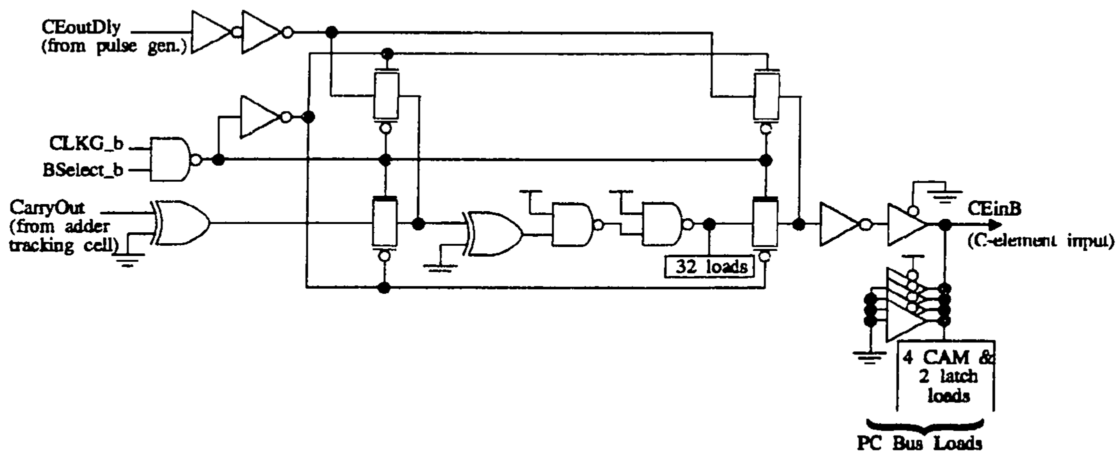


Figure 5.8: Branch tracking cell gate-level diagram.

Three main design constraints control the configuration of the branch tracking cell as well as any other tracking cell designed for selectability. First, the logic delay of the circuit between the 2-to-1 multiplexers must be less than the adder tracking cell delay, minus the pulse generator delay. This is required to guarantee that the tracking cell is set to the final state before the next cycle begins (the C-element changes state). This constraint also holds true for the tracking cell circuit delay between the output 2-to-1 multiplexer and the input to the C-element.

The second constraint applies to the input signal of the 2-to-1 multiplexers. The multiplexers select the full tracking cell delay during ø1. This gives the tracking cell select inputs time to become valid (stable-ø2 signals). If the tracking cell is selected, the full tracking cell delay selection is maintained through ø2. If the tracking cell is not selected, the tracking cell output is switched early. The multiplexer data input, which

advances the tracking cell when not selected, must be driven by a stable-ø2 signal. *CEoutDly* is the multiplexer data input which satisfies this constraint. *CEoutDly* is a signal generated near the end of the pulse generator inverter chain and is basically the C-element output delayed by several gate delays. This constraint guarantees monotonic operation of the tracking cell output.

The last constraint relates to the select input to the tracking cell. The select input must be a stable-ø2 signal, relative to the clock generator local output signals. This constraint also guarantees monotonic operation of the tracking cell output. To satisfy this constraint, the pulse width generated by the pulse generator, which results in the ø1 time of the pipeline, must be equal to, or greater than, the total delay of the global clock buffers, the clock distribution network, the local clock buffers, the select control latch, and the select input wire delay. These logic delays account for the clock skew between the clock generator and the functional unit output signals.

5.2.3 First-Level Cache Tracking Cell

The first-level cache tracking cell is designed to provide an accurate indication of the access time of the first-level instruction and data caches. This cell is selected during the pipeline stall cycle after each zero-level cache miss. The first-level cache tracking cell is not selected during prefetch request. In this case two pipeline sequencing periods (two minimum delay cycles) are required for each prefetch operation. From the analysis of the memory hierarchy (Chapter 4) we find that the first-level cache tracking cell is selected on approximately 6% of the processor cycles. By using a tracking cell for first-level cache accesses, the dynamic clock generator can provide the proper sequencing period without a completion detect signal from the cache. Eliminating the completion detect handshaking for first-level cache cycles eliminates the clock startup time occurring after a completion signal is received. This savings is significant. Our results indicate that the startup time is approximately 20% of the cache access time.

The first-level cache tracking cell is the most complicated of the tracking cells. The tracking cell models the delay of the cache RAM shown in Figure 5.9. A STRiP first-level cache is 8KB in size with an interface data path size of 16 bytes (128 bits). The cache is divided into 128 word-lines of 512 bits each. The word-lines are split to limit the number of bit-line loads to 64. The design also assumed each word-line uses local word-line drivers to distribute the word-line loads. A 4-to-1 multiplexer selects the addressed 128 bits from the 512 bit access group. Each multiplexer drives the first-stage

of a two-stage static sense amp. Static sense amps are used to maximize the performance of the cache and to simplify the tracking cell implementation. The worst-case access path includes the compare logic of the tag bits. Cache write operations are not tracked since they always occur in parallel with other pipeline operations or during external interface cycles. The gate level implementation of the first-level cache tracking cell is shown in Figure 5.10.

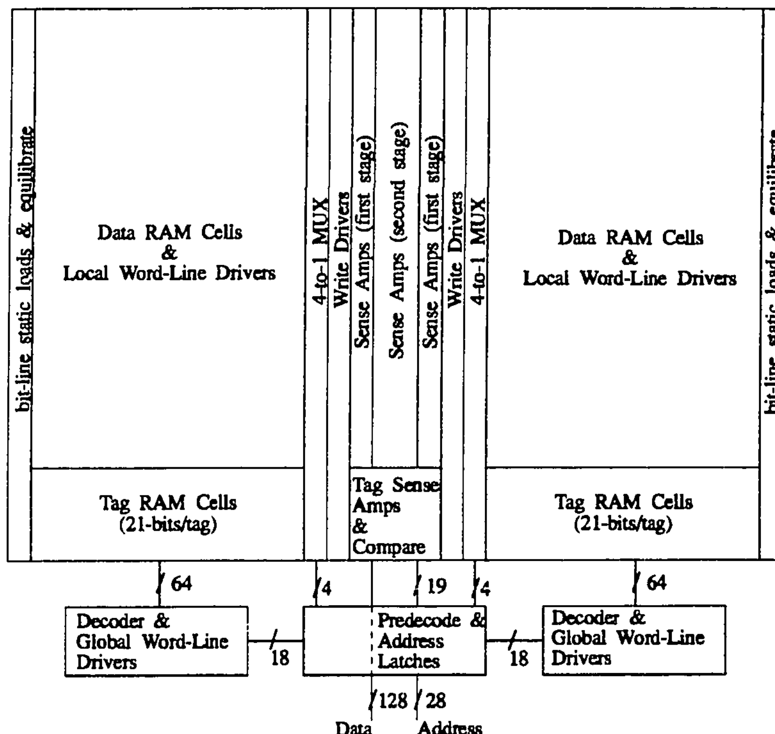


Figure 5.9: First-Level Cache floor plan.

The tracking cell contains all of the elements used in the access path of the first-level cache. Because of the unique operating characteristics and constraints of the tracking cell, the tracking cell elements are not connect in the same serial order as in the actual critical logic path. This does not reduce the tracking cell accuracy, but does reduce its complexity. The following is a list of serially connected elements which makeup the critical logic path for a first-level cache access:

- address latch and buffers
- predecoder
- decoder and global word-line driver
- local word-line driver and word-line loads
- bit-line select transistors and bit-line loads
- 4-to-1 multiplexer
- first and second stage sense amps
- address bit compare
- address word compare
- hit/miss latch

To provide accurate tracking, the first-level cache tracking cell includes the equivalent loads present on the predecoder, global word-line, local word-line, and bit-line signals. Figure 5.10 shows these load as active gates or passive transistors. Since capacitance dominates the loading characteristics, appropriately sized transistors connected as capacitors are used instead of full active gates. Also, delays caused by large gates driving large loads can be accurately duplicated by using small gates driving small loads. These approaches require detailed analysis and understanding of the capacitive loads and drivers associated with each connection, but can provide a smaller physical layout. The safest, but most area intensive, approach is to use exactly the same gates and loads used in the cache. Only when this approach is too area intensive should ratioed-down gates and loads be substituted.

5.2.4 The External Interface Tracking Cell

The external interface tracking cell is selected for each external data transfer cycle, except for resource-independent stores and write-back operations. It allows the clock generator to stop or suspend operation until the external device signals data-transfer completion. The completion signal is also internally buffered to match the data signal delays. These delays are caused by pad buffers, bus drivers, bus loads, and device latching. Because the clock period is not known in advance, the clock generator startup time adds overhead to the optimum cycle time. The clock startup time associated with an external operation includes the tracking cell delay and the C-element delay, totaling seven gate delays. Assuming a 0.8um CMOS process and a minimum external cycle of 40ns (second-level cache access), the clock startup time will add less than 10% to the

optimum cycle time. Some of the clock startup time can be hidden by overlapping the clock startup delay with some internal data signalling delays. Figure 5.11 is a gate level schematic of the external interface tracking cell.

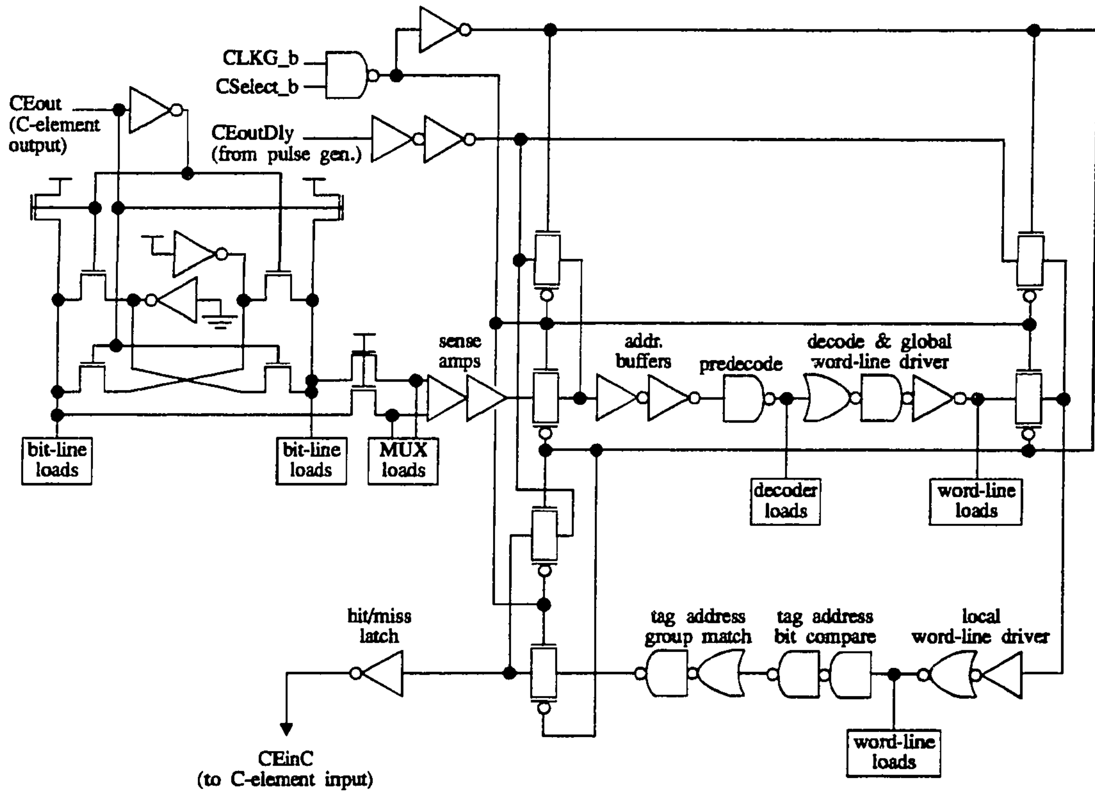


Figure 5.10: First-level Cache tracking cell gate-level diagram.

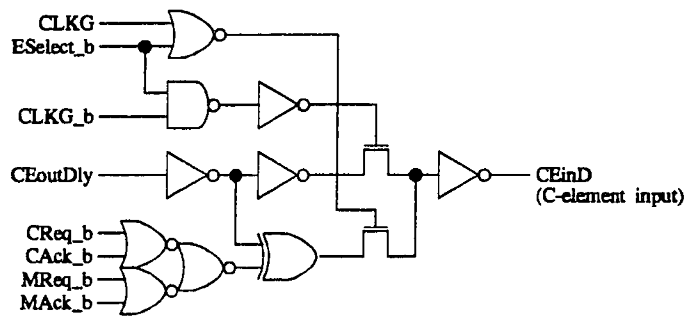


Figure 5.11: External interface tracking cell gate-level diagram.

The external communication protocol can also add overhead to the data transfer between asynchronously operated devices. This overhead is reduced by overlapping the external handshaking with the clock startup, allowing the processor to continue processing before the external handshaking has completed. Figure 5.12 is a timing diagram of a external read operation. *ACK_b* switching low (active) indicates that the data is valid on the external bus, allowing the processor to continue operation as soon as the data can be latched. The *REQ_b* inactive to *ACK_b* inactive delay need not be part of the internal cycle, improving overall system performance. The internal communication control logic must also guarantee that the external bus signals and *REQ_b* not be driven until the *ACK_b* signal is inactive. With proper protocols, the asynchronous external interface can optimize the systems performance and interconnection efficiency.

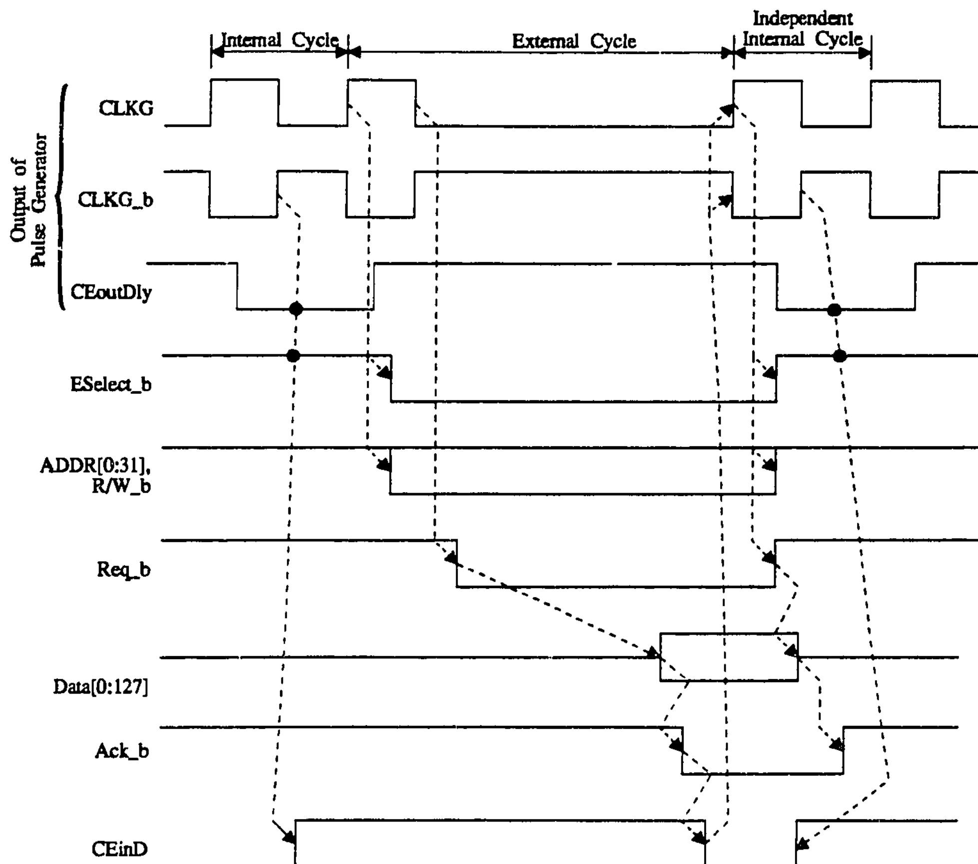


Figure 5.12: Timing diagram showing external tracking cell signals during read cycle.

5.3 The KNOB

All synchronous processors have a maximum clock frequency which provides reliable operation under all environmental conditions defined for the system. This maximum clock frequency depends heavily on the accuracy of the design implementation and process parameters for a given manufacturing run. If an error is made during logic synthesis or the process parameters change, increasing the critical logic delay, the clock frequency is reduced to provide reliable operation. This ability to set the clock frequency based on the resulting implementation and process parameters is important, allowing increased manufacturing yields and usage of otherwise broken devices.

The ability to compensate for implementation errors or miscalculations is important in a dynamically clocked processor design. If the dynamic clock generator design is correct, the processor will automatically adapt to variations in process, temperature, and voltage. But the reliable operation of a dynamically clocked structure depends on the designer's ability to target critical logic paths and build tracking cells which accurately match those logic delays. If the designer has one too few gates in the tracking cell or pulse generator inverter chain, the resulting processor chip would be rendered non-functional. To compensate for possible design errors, a method was developed to adjust the tracking cell and pulse generator delays via external signals. The method and implementation used to provide this flexibility is called a KNOB.

5.3.1 Implementation Alternatives

The main purpose of tracking cell and pulse generator KNOBs is to provide adjustability to the clock characteristics, which are controlled by the clock generator delay elements. The addition of a KNOB transforms these static delay elements into adjustable delay elements. The KNOB adjustments are set during system-board manufacturing and are not meant for dynamic adjustment during normal processor operation. Its main purpose is to allow an otherwise non-functioning device to operate properly, but with performance reduced from the target level.

Four major candidates for KNOB implementations were identified: controlling the supply voltage of selected gates in the logic path, using multiplexers to select between several logic delay paths, using voltage-controlled current-starved logic gates [48], and

adding voltage controlled load capacitors to internal delay element signals [12]. Each will be briefly analyzed in this section.

The KNOB used in STRiP's dynamic clock generator must meet the following constraints:

(1) The KNOB circuitry, when disabled, should not affect the delay element timing.
(2) The KNOB should require a limited number of input control signals (less than four for all generator delay elements).
(3) The KNOB should not add a significant amount of logic to the dynamic clock generator circuit.
(4) The KNOB should operate under worse case conditions and not reduce the effective operating range of the dynamic clock generator.

The first approach considered was to provide a separate and adjustable supply to selected gates within the tracking cells and pulse generator. To support external CMOS signal levels, the delay element input and output gates are sourced by the nominal supply voltage. This isolates the reduced voltage signals from the other parts of the clock generator. This structure is diagrammed in Figure 5.13a. The input control voltage is *VCTRL*. As this voltage level is decreased from 5V, the circuit delay increases. Conversely, a delay elements logic delay decreases as *VCTRL* is driven above 5V. But *VCTRL* variability is limited to guarantee reliable signalling between the reduced voltage gates and the normal supply voltage gates. Also, noise margins are reduced when connecting gates with different input and output signal swings. Other problems with this KNOB approach include tight control voltage tolerances, and added complexity to the physical layout of the delay element.

An alternative method is adding multiplexers in the logic path of the delay elements, allowing selection between several logic delay paths. Figure 5.13b illustrates how an adjustable delay element might be implemented using this approach. The delay control signals are decoded and used to select one leg of the multiplexer. Each leg is connected to an increasing number of serial connected logic gates. This structure allows the propagation delay of a delay element to vary within preselected, discrete increments. The disadvantages of the approach is its limited variability, increased number of control signals, and the required amount of implementation logic.

Using current-starved inverters to build adjustable delay elements was suggested by Jeong et al. [48] in their designs of PLL-based clock generators. Current-starved logic gates can also be used to adjust the tracking cells and pulse generator propagation delays. This approach uses a control signal to "current starve" a logic gate via a series-connected device. A representative example of this approach is used in Jeong's CMOS clock generator and is shown in Figure 5.14. *VCTRL* modulates the "ON" resistance of transistor *T1* and *T2*. These variable resistances control the current for charging and discharging the output capacitance. Large values of *VCTRL* allow a large current to flow, minimizing resistance and delay.

Figure 5.16 [51] shows delay time versus control voltage for a 12 stage inverter chain operated at worst-case conditions and using a 0.8um CMOS process. The main problem with this KNOB structure is the increased minimum logic delay caused by the current-mirror circuitry. From our analysis, the minimum propagation delay of a current-starved inverter is twice that of a normal CMOS inverter. This characteristic would make building precise tracking cells difficult. Other problems include susceptibility to crosstalk, noise injection, dc current drawn by the current-mirrors, and noise presented on the control signal *VCTRL*.

The final approach considered involves using shunt transistors connected between large load capacitors at the output of each internal gate. The resistance of the shunt transistors is controlled by its gate voltage. Figure 5.15 shows an example of this technique, used by Bazes [12] in an NMOS DRAM controller chip. The shunt transistors allow *VCTRL* to control the effective loads seen by the internal gates. The larger *VCTRL*, the larger the load, resulting in an increased logic delay.

Again, Figure 5.16 [51] shows the delay time versus control voltage for a 12 stage inverter chain using variable capacitive loads at each inverter output. From the curve, *VCTRL* can vary between the supply voltage rails without disabling the delay element and provides a variation of approximately 3.3ns/V between *VCTRL* = 2V - 5V. Therefore, using voltage controlled load capacitors provides better noise rejection than using current-starved gates. Also this approach uses no dc power and can easily provide a 2:1 variation to the propagation delay of each gate. Finally, the implementation size of the variable loads should have little effect on the tracking cell and pulse generator layouts.

Therefore, the use of variable-capacitive loads within the tracking cells and pulse generator was chosen as the best implementation technique for the clock generator

KNOBs. The next section describes how the design and performance of the tracking cells and pulse generator are effected when using variable-capacitance KNOBs.

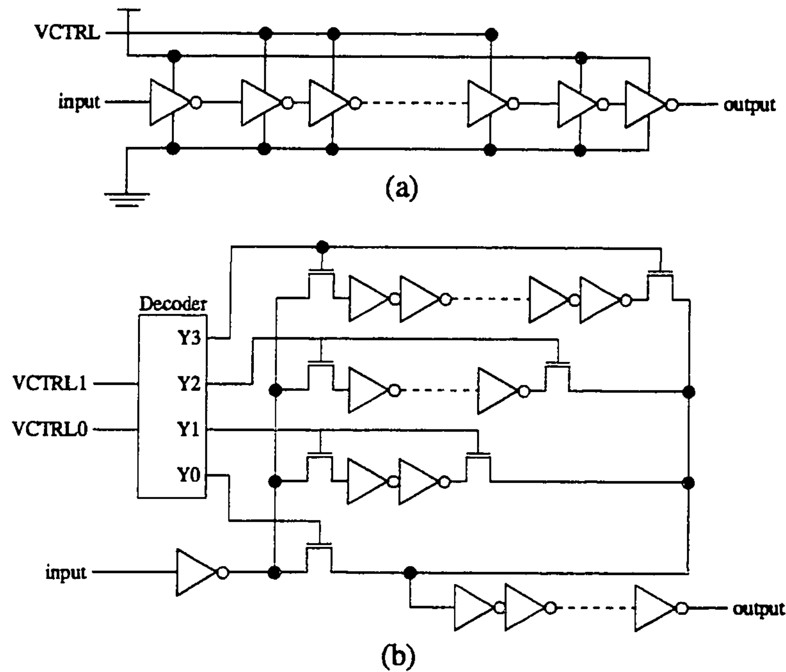


Figure 5.13: Adjustable delay elements using (a) variable supply voltage and (b) selectable inverter chains.

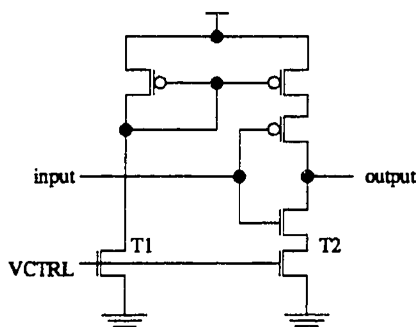


Figure 5.14: Adjustable delay inverter using current-starved delay element design style.

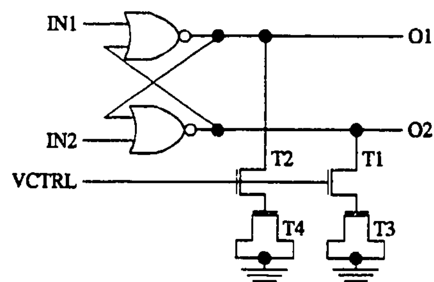


Figure 5.15: Adjustable delay logic gate using variable capacitive loading on each logic gate.

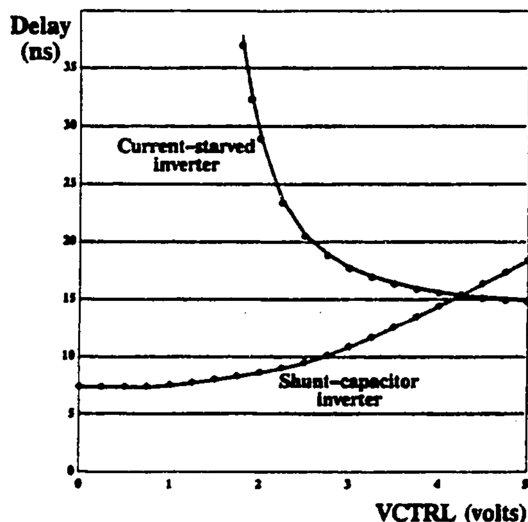


Figure 5.16: Delay versus delay element control voltage assuming worst-case conditions and a 0.8um CMOS process [51].

5.3.2 KNOB Design and Analysis

The addition of voltage controlled variable loads to the clock generator delay elements converts these elements into *voltage controlled delay elements* or VCDEs. The elements are designed such that their minimum delay will match the targeted functional operations. If an error is made during the propagation delay analysis, the propagation delay can be increased at the module or board level by increasing the KNOB control voltage. The effects of the added circuitry on delay element performance as well as the KNOB variability is analyzed in this section.

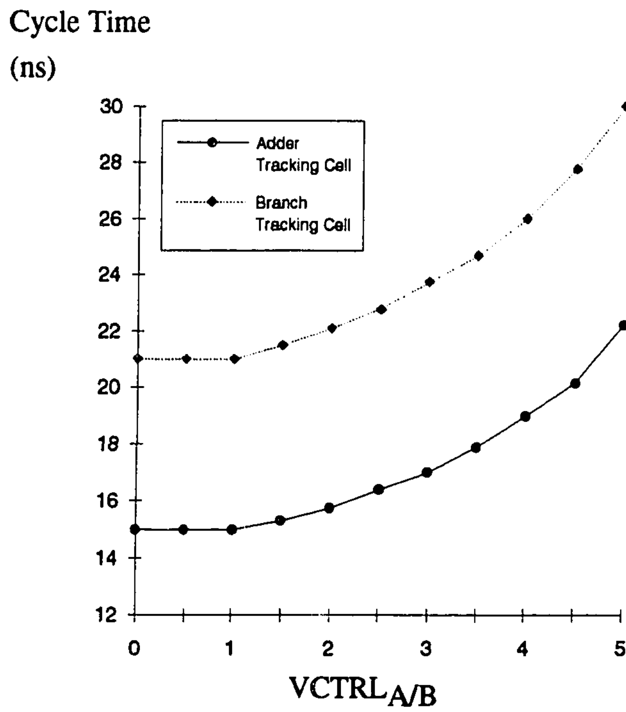
Two external signals are required: $VCTRL_{A/B}$ controls the adder, branch, and first-level cache tracking cell delays, and $VCTRL_{PG}$ controls the pulse generator delay. With the KNOB control signals set at 0V, the clock generator elements will operate at their optimum speed. The stunt transistors are of minimum size, minimizing the additional load seen by the logic elements when the KNOB is turned "OFF". The transistor size of the load capacitors was 60/2 for the 2.0um CMOS process and 20/0.8um for the 0.8um

CMOS process. The load capacitors must be sized so that the maximum load does not adversely effect the internal signal quality. A KNOB was not required for the external interface tracking cell since its period is externally controlled.

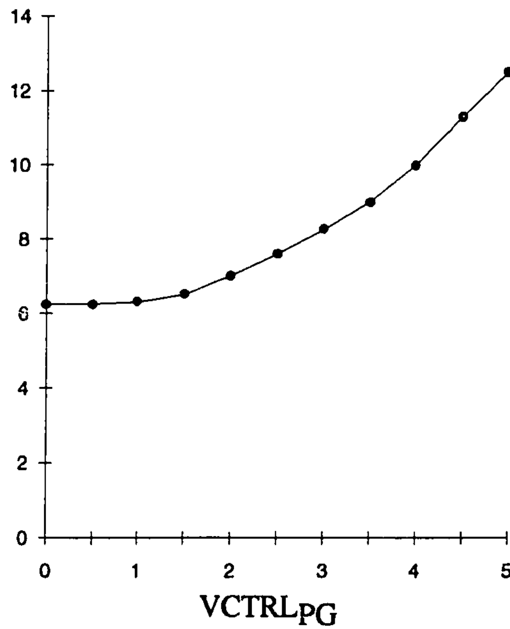
A variable capacitor is placed at each internal node of the tracking cells. This additional load has little effect on nodes with large drivers and large loads. But the majority of the internal tracking cell nodes are driven by minimum size gates with a fanout less than four. The tracking cell variability provide by the KNOB can be seen in Figure 5.17(a). This graph shows how the clock period in 2.0um CMOS varies based on the KNOB control-input signal. The KNOB provides a variability of approximately 50% over the minimum tracking cell delay. Both tracking cells varied at the same rate, maintaining a delay differential of approximately 30%.

More variability is possible if each load capacitor was sized relative to the driver capability of each node. The capability to increase the tracking cell delay by 50% is adequate for most designs. If a design is more than 50% over target, its performance is most likely inadequate to support the target system environment.

Figure 5.17(b) shows the pulse generator output variation versus the KNOB control signal. The graph has basically the same shape as the plots shown for the tracking cell variations. Since the series of inverters used to build the pulse generator delay element are lightly loaded, we were able to create a pulse width variability of 100% over the minimum period. This provided enough variability to compensate for any miscalculation in the select feedback path or minimum latchable clock pulse width. Note that the maximum ø1 period is always less than the minimum clock cycle time. The minimum ø2 period is also maintained at a reasonable amount, approximately 3ns.



(a)



(b)

Figure 5.17: KNOB varitability on (a) tracking cell cycle times and (b) pulse generator output pulse.

5.4 The C-element

A common component found in most asynchronous handshake and completion detection circuits is the Muller C-element. Unlike other asynchronous structures using dual-rail encoding, dynamic clocking uses a single C-element in the control of the processor pipeline. This C-element is located in the dynamic clock generator and combines the tracking cell outputs. Its primary function is to signal the pulse generator when all tracking elements have timed-out. The C-element allows the selected tracking cells with the slowest propagation delay to set the cycle period for the next pipeline operation. Because the dynamic clock generator accuracy and performance depend heavily on the C-element design, a detailed study was required. This section presents the design of several types of C-elements and gives a performance comparison between them. The C-element designs are structured to meet the requirements of the dynamic clock generator circuit and may not be applicable for other logic structures.

5.4.1 C-element Constraints

The C-element combines the outputs of the tracking cells and generates an output signal based on the last input transition. The C-element is a storage device whose output changes state when all its inputs reach the same logic value. Its output remains in this state until all the inputs change to the opposite logic value. The state-table given in Table 5.3 describes the operation of the C-element. The primary C-element input constraint restricts each input signal to a single transition (monotonic operation) for each output transition.

The dynamic clock generator structure and logic implementation place special requirements on the C-element design. If the tracking cells are designed to exactly match the propagation delays of selected pipelined operations, the C-element delay adds overhead to the optimum sequencing period of the pipeline. By optimizing the C-element performance, this overhead can be minimized. In most designs the C-element delay will be used to provide a level of guard-banding for the dynamic clock generator design. Circuit design guard-banding is used to guarantee proper operation under conditions un-foreseen during the development process. Independent of how the C-element propagation delay is perceived or compensated for, its operating characteristics and performance must match the requirements of the dynamic clock generator.

INPUTS A B C D	Previous Output State	Resulting Output State
0 0 0 0	x	0
0 x x x	0	0
x 0 x x	0	0
x x 0 x	0	0
x x x 0	0	0
1 1 1 1	x	1
1 x x x	1	1
x 1 x x	1	1
x x 1 x	1	1
x x x 1	1	1

Table 5.3: C-element State-Table

The following is a list of design requirements and constraints used to develop the C-element for the dynamic clock generator:

(1) **Performance** - The nominal propagation delay of the C-element must be as small as possible. The smaller the delay, the less the need for delay compensation in the tracking cells.

(2) **Number of Inputs** - Since the pipeline structure requires four operations to be tracked, the C-element must be able to receive four tracking- cell outputs. The C-element also requires a reset input for initialization. The reset design assumes that the C-element inputs will be set to the same logic level before reset is released. The dynamic clock generator design guarantees this constraint.

(3) **Drive Capability** - The C-element must be able to drive 16 standard loads (a standard load is equivalent to the fan-in of a minimum size inverter). The rule of thumb used throughout the design is a logic gate can have a maximum fan-out of four. This implies that the C-element output drive transistors must be four times the size used in a standard size gate.

(4) **Fan-in per Input** - To minimize the effects of output loading on the total tracking cell delay, the C-element fan-in per input is limited to four.

(5) **Symmetric Output Transitions** - The propagation delay for rising and falling output transitions must be equal. (This constraint applies for each input, separately; the propagation delays among different inputs need not be the same.) Symmetry is important property for minimizing the clock period variation between "like" pipeline operations. Asymmetry reduces the sequencing accuracy of the dynamic clock generator, reducing the processors optimum performance.

(6) **Reliable Operation** - The C-element must be able to operate reliably under worst-case conditions for supply voltage, temperature, process, and all noise sources. Since C-elements can be designed with dynamic nodes, charge-sharing among series stacked transistors can be a problem. Charge-sharing within the C-element can significantly reduce its noise margin and reduce reliable operation.

(7) **Dynamic Storage** - STRiP's logic implementation and analysis assumes the use of dynamic latches. This implies a minimum rate for pipeline sequencing. Because of this structure, the C-element can also use internal dynamic storage between state changes.

(8) **Power Consumption** - Static power consumption should be zero.

The following sections will discuss several C-element designs, their attributes, and the performance measured through SPICE simulations. The process parameters used during the SPICE simulations were from a MOSIS 2.0um CMOS process. All transistor level schematics give device sizes in λ. For a 2.0um process, $\lambda = 1u$.

5.4.2 C-element Designs

There are many ways to design a C-element. We analyze seven different C-element designs, two of which were developed in the course of our research. The designs studied include: cross-coupled NOR gates, a majority function gate, a pseudo-NMOS design [46, 64], a 4-input dynamic C-element, a 4-input dynamic C-element with charge-sharing reduction logic (developed during our research), a tree of 2-input dynamic C-elements, and a new design called a *pseudo-dynamic* C-element. The circuit diagrams for these designs are given in Appendix E. The cross-coupled NOR and majority function designs are fully-static logic structures. The dynamic C-element designs have internal dynamic nodes with outputs which are always actively driven. Charge-sharing reduction logic

was added to the 4-input dynamic design to improve its operating characteristics, creating a new structure. The pseudo-dynamic C-element was developed specifically to satisfy the requirements of the dynamic clock generate.

The pseudo-dynamic C-element design takes advantage of the monotonic input operating constraint and provides a C-element function by only using NMOS transistor stacks. Figure 5.18 gives an diagram of this C-element. Using only NMOS stacks, the performance of the gate is optimized. This optimization results from the NMOS switching characteristics and the reduced capacitive load on major internal nodes. Charge-sharing reduction circuitry was also incorporated in the pseudo-dynamic C-element gate to maximize reliability. The number of devices used in the implementation is relative large, but did not affect the overall element performance. Analysis shows that the pseudo-dynamic C-element provides the best operating characteristics of all the C-element designs studied.

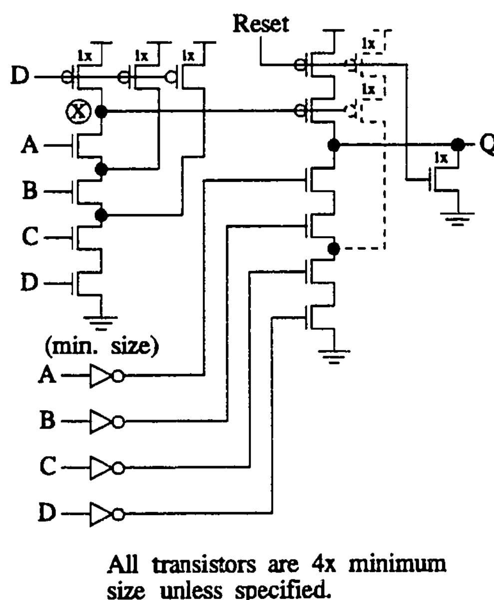


Figure 5.18: 4-input pseudo-dynamic C-element logic diagram.

5.4.3 Performance Analysis

The main goal of this C-element comparison is to determine which design provides the best operating characteristics and highest performance. Each C-element is analyzed using the SPICE simulator and 2.0um MOSIS CMOS process models. All measurements were taken using nominal supply voltage levels and temperature, 5V and 25°C. Operation was checked at worst-case conditions, 4V and 125°C, to guarantee correct behavior but not to check propagation delay. The inputs were connected such that if a series transistor stack exist in the C-element, input **A** drives the transistors connected to the deepest internal node and input **D** drives the transistors connected to the supply nodes.

To study the propagation delay of each C-element design, three input transition patterns were used. These input patterns included: (a) inputs **BCD** change state first (simultaneously) and then input **A** changes, (b) inputs **ABC** change state first (simultaneously) and then input **D** changes, and (c) all inputs change simultaneously. Input pattern (a) provides a understanding of the propagation delay from the minimum delay input. Pattern (b) will give the propagation delay for the worst-case input and also yield the worst-case charge sharing effects. The last input pattern yields the worst-case propagation delay of the C-element. Another input pattern, inputs **ABD** changing state first and then input **C** changing, was also tested to better understand the extent of the charge-sharing effects. Tables 5.3, 5.4 and 5.5 give the results of the performance analysis.

The pseudo-dynamic design provides the fastest and most symmetric operating characteristics of all the designs we considered. Its propagation delay from the minimum delay input, **A**, was well under the target criteria with an output transition asymmetry of 15%. The symmetry was further improved by reducing the size of the **A** input inverter which drives the output transistor stack. This modification increased the t_{PHL} delay to match the t_{PLH} delay. Simulation also showed how the charge-sharing reduction circuit *bootstraps* the output by 0.25V when inputs **BCD** change state. The bootstrapping does not effect the reliable operation of the C-element. When all inputs are switched simultaneously, the pseudo-dynamic design also provided the best operating characteristics. The pseudo-dynamic C-element was typically 25% faster than the other implementation techniques for all input patterns.

Charge-sharing was most evident when input **D** was switched after inputs **ABC** had changed state. The 4-input dynamic C-element design without the charge-sharing

circuitry failed with this input sequence. Charge sharing changed the internal node voltage by more than 2.5V. This was enough to cause the output to switch prematurely. By adding the charge-sharing reduction circuitry to the dynamic design, the negative charge-sharing affects were significantly reduced.

	C-element Output Delay from input A	
C-element Design	t_{PLH} (ns)	t_{PHL} (ns)
pseudo-dynamic	1.47	1.27
dynamic	2.05	1.56
dynamic w/charge-sharing reduction	2.07	1.59
dynamic tree	2.25	2.05
pseud0-NMOS	2.42	1.99
cross-coupled NOR	3.21	2.45
majority function	3.70	3.05

Table 5.4: C-element propagation delay from **A** to **Q** with a fanout of 16. Technology = 2.0um CMOS (MOSIS)

	C-element Output Delay from input ABCD switched simultaneously	
C-element Design	t_{PLH} (ns)	t_{PHL} (ns)
pseudo-dynamic	2.07	2.08
dynamic	2.12	1.89
dynamic w/charge-sharing reduction	2.40	2.45
dynamic tree	2.60	2.45
pseud0-NMOS	2.65	2.45
cross-coupled NOR	3.83	3.05
majority function	3.84	3.57

Table 5.5: C-element propagation delay from **ABCD** switched simultaneously to **Q** with a fanout of 16. Technology = 2.0um CMOS (MOSIS)

	C-element Output Delay from input D	
C-element Design	t_{PLH} (ns)	t_{PHL} (ns)
pseudo-dynamic	2.07	2.08
dynamic	failed due to charge sharing	
dynamic w/charge-sharing reduction	2.46	2.73
dynamic tree	2.75	2.70
pseud0-NMOS	2.71	2.02
cross-coupled NOR	3.53	2.95
majority function	3.76	3.67

Table 5.6: C-element propagation delay from **D** to **Q** with a fanout of 16.

5.5 The External Interface

The connection of STRiP's CPU to the outside world is done by the Bus Interface Unit (BIU). It is important that this interface minimizes the overhead required for external data transfers. All external data transfers between the second-level caches, system memory, coprocessors, and main system bus use a fully asynchronous handshaking protocol. This type of interface allows devices and subsystems of different operating speeds to communicate easily and efficiently across a common bus. Many times the processing and data transfer rate of the processor is different than the external memory device's data access time and transfer rate. A synchronous interface must insert extra delay (cycles) to synchronize the communication between these two dissimilar devices. By using dynamic clocking, the internal processor clock stops during the external transfer until the external device signals completion. This capability eliminates the synchronization overhead and potential metastable conditions common in synchronous interfaces.

The external interface is designed to support an optional second-level copy-back cache. For cache coherency, signals are provided to allow snoop operations to the internal caches. The second-level cache is assumed fully inclusive of the internal first-level cache. Therefore, the external cache will always be snooped first, reducing the snoop bandwidth to the internal caches. The basic organization of a single-processor subsystem is shown in Figure 5.19.

This section gives a brief description of STRiP's external communication protocol. The interface assumes an external floating-point coprocessor, similar to MIPS-X. The number of interface pins was selected based on presently available package technologies

and was not limited to the same packaging chooses available to the MIPS-X designers. Detailed signal descriptions are given in Appendix F.

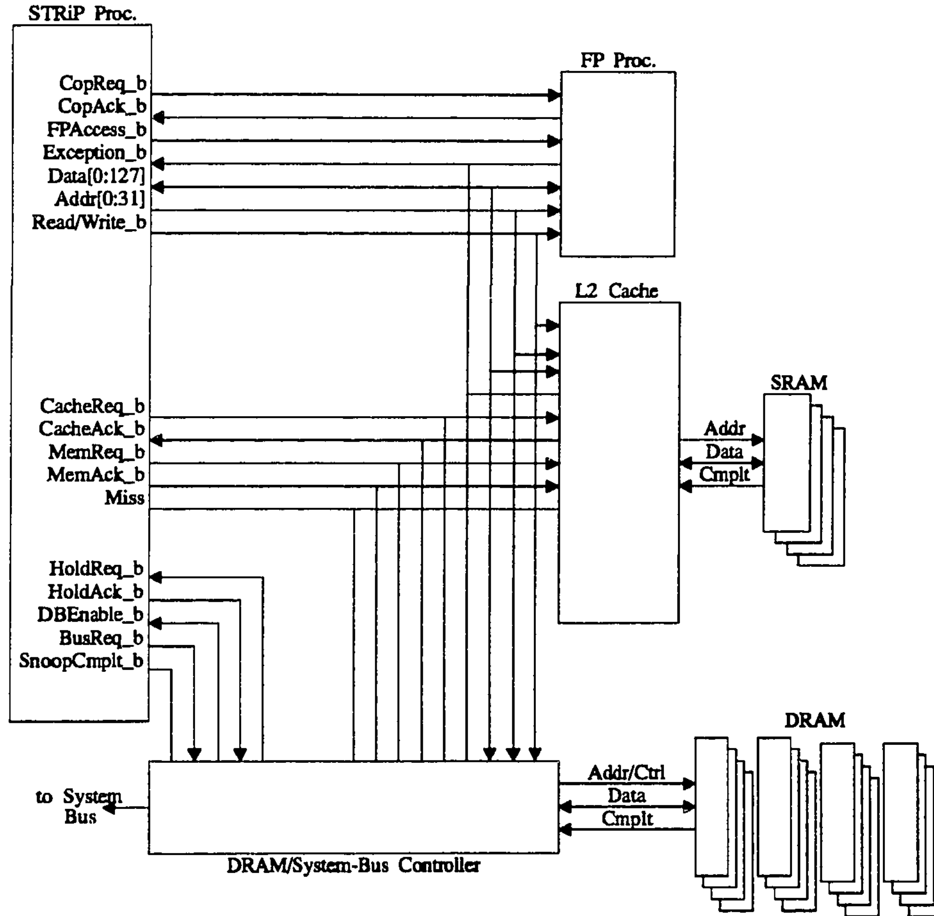


Figure 5.19: Processor complex block diagram.

5.5.1 Communication Protocol

The majority of the external data transfers are to/from the external memory system. The external memory system can contain caches (second-level) built with SRAMs and main system memory, typically built with DRAMs. External memory transfers are caused by an internal first-level cache miss, a sequential prefetch, or a first-level data cache copy-back cycle. All external read transfers caused by an internal miss occur while the

pipeline is stalled. Prefetch and copy-back cycles can occur in parallel with normal pipeline sequencing. We will describe the generic read and write protocol for STRiP's asynchronous interface since the signalling is the same independent of the request type (cache or main memory).

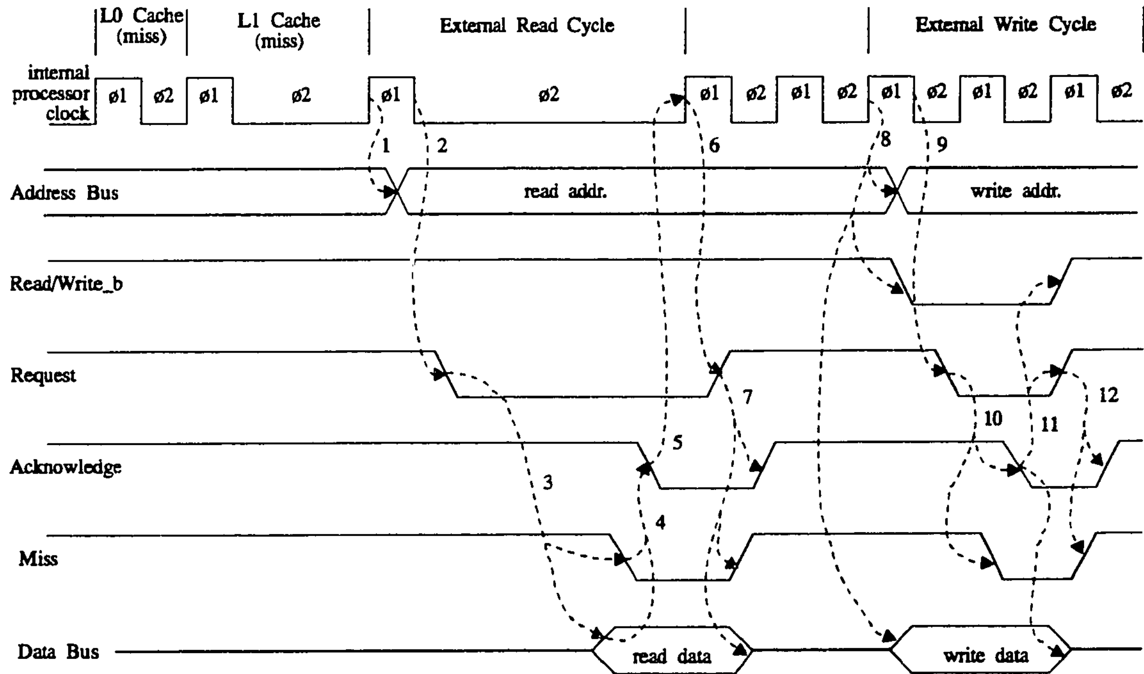


Figure 5.20: Timing diagram showing generic, external read and write cycle signalling.

Figure 5.20 illustrates the external signal timing supporting an external read (caused by a internal cache miss), followed by an external write (caused by a copy-back request). The following is a list of relative signal transitions (labeled in Figure 5.20) to support these transfers:

1. The BIU drives the miss address on to the **Address Bus** at the beginning of ø1. **Read/Write_b** is also driven high to indicate a read cycle.
2. The BIU drives **Request** active at the beginning of ø2. **Address Bus** and **Read/Write_b** setup to **Request** active is 0ns (min.).

3. **Request** causes an external memory system access. The requested memory system device drives the accessed location on to the **Data Bus**. If the accessed location is available, **Miss** is also driven inactive.
4. Once valid data is driven to the **Data Bus**, the external device drives **Acknowledge** active. The **Data Bus** and **Miss** setup to **Acknowledge** active is 0ns (min.).
5. The **Data Bus** is driven on-chip and latched into the internal caches and target register on the rising edge of the internal clock. **Miss** is also sampled to determine if the data is valid. The internal pipeline continues operation, independent of the BIU's external cycling.
6. The end of the external read cycle causes the BIU to drive **Request** inactive.
7. The external memory system releases the **Data Bus**, drives **Miss** high and **Acknowledge** inactive, indicating that the external bus has been released. **Data Bus** released and **Miss** setup to **Acknowledge** inactive is 0ns (min.).
8. Once the BIU recognizes that the external data bus has been released, the write-back address and data are driven to the bus, along with **Read/Write_b** driven low.
9. At the beginning of the next ø2 clock phase, **Request** is driven active, starting a write cycle. **Address Bus**, **Data Bus**, and **Read/Write_b** valid to **Request** active is 0ns (min.).
10. When the external memory system successfully completes the write access, it drives **Miss** low and **Acknowledge** active. **Miss** setup to **Acknowledge** active is 0ns (min.).
11. The BIU drives **Read/Write_b** high, tri-states the **Data Bus**, and drives **Request** inactive. **Read/Write_b** high and **Data Bus** tri-state setup to **Request** inactive is 0ns (min.)
12. The external device completes the protocol by driving **Miss** and **Acknowledge** high. **Miss** high to **Acknowledge** inactive is 0ns (min.).

Although the full protocol is based on a 4-phase asynchronous handshaking protocol, the read latency seen by the processor is similar to a 2-phase protocol. Therefore, the transfer latency is primarily dependent on the external-devices access time since most of the communication overhead has been eliminated. The BIU uses fundamental mode logic structures to handle the asynchronous signalling between parallel processor requests and external device handshaking. This is required since the end of the read-cycle

handshaking and the entire write cycle occur in parallel with the internal-pipeline sequencing of subsequent instructions.

Note that only signal setup times need to be specified, relative to **Request** and **Acknowledge**. All setup times are 0ns (min) and must be guaranteed by the device driving the control signals. Hold times are automatically guaranteed by the asynchronous protocol. The protocol is designed to function properly independent of the connected devices' processing rates. Since the interconnection of asynchronous devices is correct-by-design, very little timing analysis is required by the system designer.

To guarantee reliable operation, the **Request** and **Acknowledge** wire delays and loads must be greater than or equal to the other interface signals (**Address Bus**, **Data Bus**, etc.). This guarantees the setup times as seen by the receiving device. This requires an increase awareness by the system designer of the circuit board characteristics. Synchronous interfaces can easily hide miss-matches in control signal loading, but asynchronous interfaces depend heavily on the control of these circuit board delays. Traditional synchronization logic is used to handle input signals which occur asynchronous to the pipeline sequencing (external interrupts or bus requests).

Dynamic clocking supports a fully asynchronous interface without the synchronization overheads caused by mismatches in device operating rates. The ability to stop the clock during external stall cycles provides a efficient means of communications without metastable conditions. Asynchronous interrupt inputs and bus-request signals are handled as they are today in synchronous processors, but these signals are not in the critical processing path of the processor. Therefore, STRiP's external interface provides an means of interconnecting devices independent of their processing rates, without imposing synchronization overheads or 4-phase latencies.

5.6 Exception Handling

To support precise exceptions, STRiP (like MIPS-X) delays all processor state changes until the final pipeline stage (**WB**). Data/instruction transfers to external processors (coprocessors and floating-point processors) are also delayed until the final pipeline cycle. All instructions executed by the pipeline are issued and retired in order. Therefore, when an exception occurs, all instructions in the pipeline are restartable. Because of this feature, STRiP does not allow any instructions in the pipeline to complete after an exception is received. This is true independent of which instruction caused the exception.

All floating-point operations will also be issued and retired in order of their occurrence. But since the external FP processor operates asynchronously, relative to the scalar pipeline, FP operations are retired out-of-order, relative to the scalar instruction stream. Software interlocks can be used to guarantee ordering where required. These interlocks are not recommended since they restrict parallel processing of the scalar and FP operations. To handle FP exceptions, the FP processor must store the issued chain of encoded operations which have not been retired (not instruction addresses like the PC Chain). The FP processor must halt its pipeline sequencing once an FP exception is detected and must remain idle until the scalar processor can service the exception. Therefore, the FP exceptions are not precise with respect to the scalar instruction stream, but are precise with respect to the issue order of FP operations.

In response to an exception, the pipeline is halted and the PC is immediately set to zero (exception vector address). The PC Chain is frozen so that the addresses of the instructions in the **RF**, **ALU**, and **MEM** pipe stages can be saved. The *PSWcurrent* bits are saved into *PSWother*, interrupts are turned off, and the machine is placed into system mode. The exception handler saves the PC Chain and *PSWother* values before enabling interrupts and PC Chain shifting. After the exception is resolved, the PC values are restored in the PC Chain and three jumps are executed using the contents of the PC Chain (**jpcrs** and two **jpc** instructions). Since exceptions occur infrequently, very little is lost by not completing instructions that were in the pipeline before the exception point.

5.6.1 External Interrupts

Exceptions are also caused by external data transfer errors or internal computation faults. External exceptions such as interrupts and page-faults are asynchronous events. An interrupt input provides a maskable processor interrupt input for I/O devices requiring service by the processor. It is normally driven independent of any external processor transfer cycle. The same techniques used to synchronize asynchronous inputs to a synchronous processor clock are used in STRiP. The interrupt input is synchronized with STRiP's dynamic clock. Since a synchronous processor cannot respond to an interrupt input during a stall condition or external transfer, the fact that the dynamic clock stops during external cycles does not reduce the response time to an interrupt.

The non-maskable interrupt signal supports error detection based on an external memory system transfer. Errors which can cause this exception include page-faults (detected by the TLB), and parity checks and ECC errors (detected by the memory

system controllers). The non-maskable interrupt has a setup time of 0ns (same as data) to the acknowledge signal driven by the communicating device. This allows the processor to begin the exception handling routine during the next cycle and avoid any permanent changes to the machine state. Therefore, the non-maskable interrupt does not require synchronization since, by definition, it is synchronous with the end of an external transfer. Data is discarded on prefetch cycles which cause an non-maskable interrupt and the interrupt ignored. Memory faults caused by alternate bus masters (i.e. DMA) are handled through a non-maskable asynchronous interrupt input.

5.6.2 Internal Exceptions

Internal exceptions are caused by overflow operations (*trap-on-overflow*) and **trap** instructions. They occur synchronous to the pipeline sequencing. This allows them to be accepted and serviced before the processor state is permanently changed. The trap-on-overflow interrupt can be masked by setting a bit in the processor status word register. When a trap-on-overflow occurs, another 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.

Trap instructions operate in the same manner as those provided in the MIPS-X implementation. The trap instructions are unconditional software interrupts which vector the processor to a system space routine in low system memory. These are similar to the software interrupts provided in the X86 processors from Intel. The trap instructions have an 8-bit vector number which provides 256 possible trap addresses. These vectors, when shifted by three bits, are address of the vector routines in low system memory. By separating the vector addresses by eight instructions, a short routine is used to provide a jump to the correct trap handler. Since handling of traps was not the main concern of our research, we reference the reader to Chow's book [Chow89] for more details on trap instructions and trap handlers.

5.7 Performance Analysis

To analysis the functionality and performance of STRiP's dynamic clocking structure, a SPICE model was generated to match the physical layout characteristics of the clock distribution network. The MIPS-X clock distribution network parameters were used as a model for analyzing the feasibility of a dynamically clocked RISC pipeline. Figure 5.21 is a block diagram of the SPICE simulation model used during our analysis. All clock

network loads and wire lengths were based on the layout results of the MIPS-X pipeline. Signal quality, tracking cell selectability and performance, cycle-to-cycle symmetry and stability, clock skew, select feedback delay, and performance under worst-case conditions were a few key parameters analyzed during the study. The select feedback signals were sequenced to provide a mix of cycle lengths, testing cycle-to-cycle interactions. The majority of the analysis used a MOSIS 2.0um CMOS process model, but 0.8um CMOS and BiCMOS process models were used to understand the scalability and potential performance of a dynamic clocking structure.

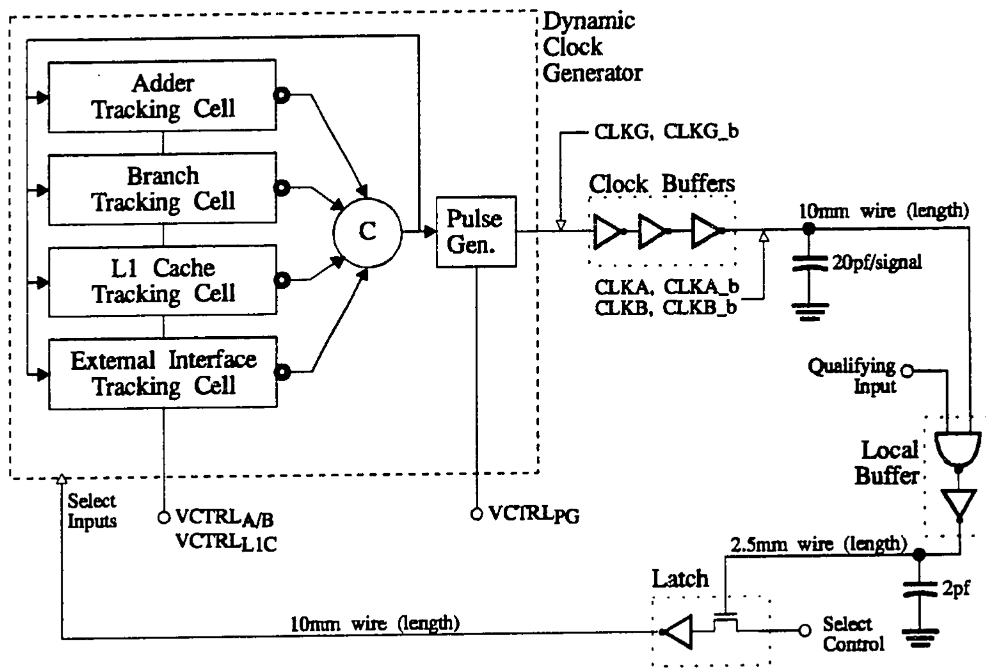


Figure 5.21: Dynamic clocking structure used during SPICE simulation.

Throughout the simulation analysis, no adverse effects caused by the dynamic clocking structure were observed. Table 5.7 is a summary of the key parameters measured during the simulation. The worst-case conditions of Vcc = 4.0V and Temp. = 125°C were chosen to provide an understanding of the tracking characteristics and functionality of dynamic clocking under extreme operating conditions.

	2.0um CMOS		0.8um CMOS		0.8um BiCMOS
Measured Parameters	typical	worse case*	typical	worse case*	typical
Gate delay, inverter with fanout = 4, (ns)	0.9	1.75	0.45	0.80	0.35
Minimum cycle time (ns)	15.0	27.0	7.5	14.5	5.5
Branch cycle time (ns)	21.0	35.5	10.5	19.0	8.0
L1 Cache cycle time (ns)	40.0	72.0	18.5	32.5	15.0
ø1 period (ns)	7.0	12.6	3.5	6.3	3.0
Clock Gen. to selects (ns)	6.5	11.8	3.0	5.4	3.0
Clock startup time (ns)	7.0	12.6	3.5	6.5	2.5
C-element delay, min., (ns)	1.47		0.68		0.53
High level output voltage	5	4	5	4	4.25
Low level output voltage	0	0	0	0	0.75
Effective clock rate (MHz)	63	35	120	65	175

*Worse case conditions -- Vcc = 4.0V, Temperature = 125°C, Process = nominal

Table 5.7: Performance parameters measured during SPICE simulation of dynamic clocking structure.

PERCENT	CONTRIBUTING ATTRIBUTE
75%	Not requiring worse-case operating frequencies under nominal operating conditions.
25%	Matching the required pipeline period to the pending operations.
50% (peak)	Using an asynchronous external interface.
125%	Using the high-speed static adder, 3-port register file, and fully-addressable prefetch buffer.

Table 5.8: Estimated performance improvement, over MIPS-X, attributed to dynamic clocking and the use of high speed functional units.

The dynamic clock generator drives the clock distribution network the same way the synchronous, static-frequency clock generator used in MIPS-X drove the identical network. The main difference in these two methods is the way the clock is generated. The drivers used to drive the global and local clock signals are structurally identical (except in the BiCMOS case). Therefore, the global- and local-clock signal quality and the response characteristics were much the same as those found in the MIPS-X

implementation. The use of dynamic clocking did not adversely affect the drive quality of the clock signals.

The main advantage of dynamic clocking is the ability to allow the processor operating rate to be as fast as the silicon can support. This capability allows a dynamically clocked processor to operation approximately 100% faster than an equivalent synchronous processor implementation. STRiP's typical operating rate is three times faster than that achieved by MIPS-X using the same 2.0um CMOS process. Table 5.8 list the elements in the STRiP design which provide performance benefits over the MIPS-X implementation and the amount of performance each contributed.

The design of a dynamically clocked processor requires more detailed analysis than for an equivalent synchronous processor design. There are three main design constraints which must be addressed during the development process. The first is to guarantee that the propagation delay of the tracking cells match the processing delays of the targeted functional operations. If the tracking cells are built as described in this chapter, their accuracy should not be a serious problem. Tracking cells are also commonly used in self-timed PLAs, SRAMs, DRAMs, and floating point units. Therefore, their construction and reliable operation have been demonstrated in commercially available chips.

The second difficulty in dynamic clock design is the determination of which critical logic paths are most frequently used and dominate the data processing patterns in the processor pipeline. This determination drives the decision of which logical operations to tracked. There are commercially available simulation and diagnostic tools which can assist in this analysis. Not all RISC processors are like MIPS-X; some would probably require a different set of tracking cells in a dynamically clocked implementation.

The last major constraint in the dynamic clock generator design is the response time of the functional units which control the select inputs. This response time determines the width of the ø1 period. Our goal was to provide a response time which was less than half the minimum cycle time of the pipeline. Since the response time depends on the clock distribution network, care must be taken to fully evaluate all parameters which affect the select signals. Since MIPS-X had already been built, these parameters were relatively easy to generate. Figure 5.22 shows the relationship between the output of the clock generator and the select input created by the simulation model. This plot is representative of the select signal response for all technologies simulated. Note that the select setup time to the rising edge of the clock is approximately 0.5ns. This proved to be enough to provide reliable clock generator operation under all possible conditions.

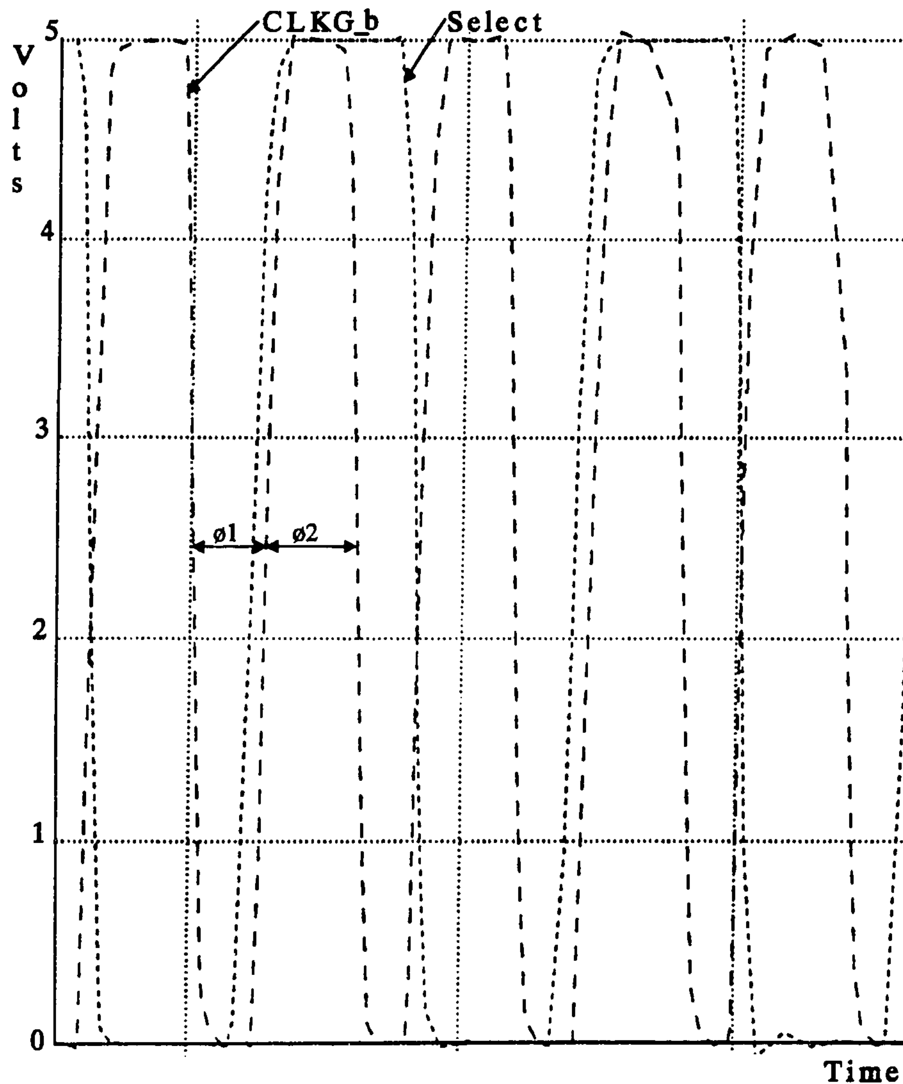


Figure 5.22: SPICE plot of clock generator output and select feedback delay.

5.8 Summary

This chapter provided an understanding of the effects dynamic clocking has on the operation and design of pipeline structures. The efficiency of a self-timed RISC processor, STRiP, is compared with it synchronous equivalent, MIPS-X. From our analysis, dynamic clocking utilizes the available silicon performance without reducing the system reliability or increasing the processors complexity. Dynamic clocking also

allows the use of a fully asynchronous external interface, providing efficient data transfer independent of the processing rates of the communicating devices. A self-timed processor design using dynamic clocking requires a more detailed design analysis than does an equivalent synchronous implementation. But by allowing the processor's cycle time to vary with process, voltage, temperature, and pipeline operations, 100% more performance is achievable under typical operating conditions.

Chapter 6

Conclusions

We have shown that a self-timed synchronous sequencing method, called dynamic clocking, can more than double the performance of modern RISC processors by extracting silicon performance which is not accessible through conventional sequencing methods. While synchronous operation of the internal pipeline and external interface simplifies a processor's implementation and sequencing, worst-case design constraints restricts a processor from taking full advantage of the available silicon performance. Self-timed systems provide an attractive alternative, but suffer from complex logic structures and sequencing overheads. Dynamic clocking combines the simple and efficient sequencing structures of synchronous design with the adaptive operating and interface attributes of self-timed systems.

Our research first investigated the structures used in modern processor design to understand their advantages and constraints (Chapter 2). Because synchronous structures are sequenced based on worst-case operating conditions, they are restricted to approximately 55% of the silicon performance under nominal operating conditions. Asynchronous logic structures adapt with changes in operating conditions and provide a efficient device interface, independent of the communication rates of the devices. But dual-rail encoding of logic variables to provide completion detection increases the complexity of an asynchronous implementation. This complexity, along with sequencing overheads caused by completion signalling, significantly reduce the performance and increase the design time of traditional asynchronous structures.

The study of contemporary processor pipeline characteristics showed that lock-step operation of a processor's pipeline is efficient if the cycle-by-cycle sequencing is based

on the present environmental conditions, process parameters, and pipeline operations. But synchronous communications between external devices limits the interface efficiency of devices with different operating rates or access methods. A fully asynchronous external interface would allow the interconnection of devices independent of their processing rate. This led to the formulation of the dynamic clocking concept and the definition of a self-timed RISC processor architecture (STRiP).

Dynamic clocking, as described in Chapter 3, is best defined as a self-timed synchronous pipeline sequencing method. It allows adaptive lock-step sequencing of the processor's pipeline and a fully asynchronous external interface. The logic structures used in a dynamically clocked pipeline are identical to synchronous logic structures, thus avoiding the complexities of traditional self-timed structures. The dynamic clock generator uses tracking cells, with select control, to create the required clock cycle for each pipeline operation. The ability to instantaneously start and stop the clock supports the fully asynchronous external interface without synchronization overheads or metastable conditions. Our study shows that applying dynamic clocking to a typical RISC processor (MIPS-X) doubles its performance under typical operating conditions.

To further optimize the processor's performance, Chapter 4 described an enhanced memory system hierarchy. Most modern processor designs are limited by the average access time of their memory systems. To eliminate this constraint, small (less than 256 bytes) low-latency caches were added to the memory hierarchy. An aggressive and adaptive prefetching algorithm was created to minimize the small cache's miss rates. Zero-level caching with predictive prefetching cuts the memory system's average access time by more than 50%, removing the memory system from the pipeline's critical logic paths. The addition of zero-level caches also avoided increasing the processor's CPI rating, common when using pipelined caches.

Finally, a RISC processor optimized for self-timed sequencing via dynamic clocking is described. STRiP (self-timed RISC processor) uses traditional functional unit designs along with zero-level caches and predictive prefetching. Its implementation was simulated in several silicon technologies. In a 0.8um CMOS process STRiP has an average sequencing rate of 120MHz. When compared to an equivalent synchronous processor built in the same technology, STRiP provides twice the performance in a typical operating environment. The fastest technology tested was a 0.8um BiCMOS process, which yielded a average sequencing rate of 175MHz.

Additional research and development work is required to verify and optimize the operation of a dynamically-clocked processor. Building an actual working processor

would reveal the feasibility and constraints of dynamic clocking. Also, applying dynamic clocking to other processor architectures would provide a better understanding of its general usability. We have briefly considered the effectiveness of dynamic clocking on a Superscalar or Superpipelined architecture. We believe both of these architectures can benefit from the use of dynamic clocking, but more detailed studies are required. Also, further studies are needed to understand the interaction of multiple on-chip dynamically-clocked devices. For example, the interaction of a processor with a floating-point unit, both operating asynchronously to each other and both using dynamic clocking. As for further research in predictive prefetching, we did not evaluate the possibility of storing and using a history of data reference strides to predict future data references. Used in combination with the data reference history, stride prediction could significantly reduce the average access time for data references.

Dynamic clocking appears to be a feasible and efficient method of sequencing a processor pipeline. By providing self-timed adaptability and simple design constraints, dynamic clocking promises double the performance of a traditional processor architecture without significantly increasing its complexity.

Appendix A

Memory System Terminology

One difficulty with the existing literature on caches is inconsistent and conflicting terminology. This thesis employs the terminology S. A. Przybylski adopted in his study of cache and memory hierarchy design. The following is a list of terms and their definition:

Internal and External:

Internal will refer to a logic structure which resides on the processor chip. External refers to a logic structure which is implemented off the processor chip or whose interface must cross the processor chip's interface pins.

Word:

A word is defined to be 32-bits or 4-bytes.

Cache:

A cache is a small memory subsystem that at any one time can hold the contents of a fraction of the overall memory of the machine. The main purpose of a cache is to provide a fast local storage of the most recently and most frequently accessed data and instruction references. Their are several good references which detail the fundamentals and theory behind cache design and usage (see Chapter 2).

Zero-Level Cache:

A fully-addressable prefetch buffer which is place between the CPU and the first-level cache. A zero-level cache is very small (less than 64 words), fully associative, and would normally utilize a prefetch strategy to minimize its miss

rate. The STRiP architecture contains a zero-level instruction cache and a zero-level data cache.

First-Level Cache:

Also called the primary cache. In most system structures it would be the first storage layer in the memory system hierarchy. Most advanced processor chips contain internal first-level caches which range in size from 1K words to 4K words. In the STRiP architecture the first-level caches, instruction and data, are the second layer in the memory hierarchy.

Second-Level Cache:

In a multi-level cache hierarchy, the second-level cache lies beyond the first-level cache. It can range in size form 16K words to 512K words. Second-level caches are typically implemented using discrete SRAMs. These caches are also known as secondary caches.

Main Memory:

Main memory is the memory subsystem placed beyond any cache in the system. Often considered the last level in the memory hierarchy (if the magnetic media storage is not counted as a level). The main memory in most microprocessor systems in made up of DRAMs and ranges in size form 256K words to 256M words.

Block:

Also commonly called a line, a block is the minimum unit of information that is directly addressable in the cache. Each block has an address tag. The *block size* can range from one word to 32 words. A block may be divided into sub-blocks if the cache *fetch size* is smaller than the block size.

Tag:

A cache tag is a unit of storage which defines the main memory address of a stored unit of data. The tag indicates which portion or block of main memory is currently occupying a cache block.

Fetch and Prefetch Size:

The amount of memory that is fetched or prefetched from the next level in the memory hierarchy. A *data fetch* occurs only when requested by the CPU while a *data prefetch* is generated independent of the CPU's operation and attempts to fetch data ahead of a CPU's request for it. Fetch size is also known as the transfer size and sub-block size.

Fetch and Prefetch Strategy:

The algorithm for deciding when a fetch of some data from the next level in the memory hierarchy is going to be initiated, which address is to be fetched, and which word or group of words is to be returned first.

Replacement Strategy:

The algorithm for choosing which block will receive a newly fetched block of data. The most common replacement strategies are *Random* and *Least Recently Used* (LRU).

Set-Associativity:

A *set* is a collection of two or more blocks which are addressed and checked in parallel. If there are *n* blocks in a set, the cache is called *n-way set-associative.* If there is one block per set (each block has only one place it can appear in the cache) the cache is called *direct-mapped.* The cache is *fully associative* if the cache is made up of one set (a block can be placed anywhere in the cache).

Hit or Miss:

A *hit* occurs when the requested memory address is found in a cache. A *miss* occurs when the requested memory address is not found in the cache.

Write Strategy:

The write strategy, or policy, is all the details of how writes are handled in a cache. There are two basic write-hit options: (1) a *write-through* policy where the information is written to both the cache block and the next level in the memory hierarchy and (2) a *copy-back* or *write-back* policy where the information is written only to the cache block. There are also two options on a write-miss: (1) a *write-allocate* policy loads the block from the next level in the memory hierarchy and than performs the write-hit policy and (2) a *no-write-allocate* policy does not modify the cache on a write miss and only writes the data to the next level in the memory hierarchy. The write-hit and -miss strategies combine to form the cache's write policy. The most common write policies are copy-back write-allocate (CBWA) and write-through no-write-allocate (WTNWA).

Spatial Locality:

Given a memory location is referenced, there is a high probability that neighboring locations will be referenced within the program's lifetime.

Temporal Locality:

Given a memory location is referenced, there is a high probability that it will referenced again within the program's lifetime.

Instruction Sequentiality:

Given a instruction reference from memory location n, there is a high probability that the next instruction reference will be to memory location $n+1$. This is a subset of the spatial locality property but contributes significantly to the effectiveness of the prefetch strategies described later in this chapter.

Appendix B

Signal Naming Convention

The naming convention used for STRiP's signal names was identical to that used by MIPS-X designers. The main purpose of defining a naming convention early in the design process is to guarantee some consistency in signal labels. Signal labels should also convey information on the functional significants and active sense of a signal.

A signal name is make-up of three parts; the signals functional name, its active sense, and the signal type. The format of the signal name is *SignalFunction_Sense_Type*. The signals functional name provides information on the control/data conducted by the signal wire (*TakeBranch, ReadWordLine, ResultBus, ...*). If a signal is active "high", the "*_Sense*" part of the signal name is omitted. An active "low" signal or a signal which has been complemented uses a "*_b*" in the "*_Sense*" section of the signal name. An example of this would be *ResultBus_b*. The "*_Type*" section of the signal name is defined as follows:

_s1	Stable ø1
_s2	Stable ø2
_v1	Valid ø1
_v2	Valid ø2
_q1	Qualified ø1
_q2	Qualified ø2
_dc	DC signal
_w1	Weird ø1 (all signals types occurring during ø1 which cannot be define by one of the above ø1 signal types)

_w2 Weird ø2 (all signals types occurring during ø2 which cannot be define by one of the above ø1 signal types)

_w Weird (all signals types occurring asynchronously to the phases of the clock)

A *stable* signal is defined to be constant throughout the specified phase. A *valid* signal can change state during the specified phase but is guaranteed to be constant before and during the falling edge of that phase. A *qualified* signal is created by logically ANDing a signal with a clock, which is active high on the specified phase. If a signal can be associated with a pipeline stage, a letter is appended to the end of the signal type:

i **IF** pipeline stage

r **RF** pipeline stage

a **ALU** pipeline stage

m **MEM** pipeline stage

w **WB** pipeline stage

An example of some signal names are: *TakeBranch_b_s2a, ResultBus_b_v2a,* or *IncDrvResBus_q2.*

Appendix C

SPICE Parameters

The following SPICE model cards were used during STRiP simulations. They were chosen as representative of generally available silicon processes used for manufacturing commercial microprocessors. Typical environmental conditions used during simulation were Vcc = 5.0V and Temp. = 25°C. Worse case environmental conditions used during simulation were Vcc = 4.0V and Temp. = 125°C.

SPICE model cards for NMOS and PMOS transistors representing a 2.0um MOSIS CMOS process (typical). Data courtesy of Mark Horowitz, Stanford University.

```
*
*
.MODEL NT NMOS (LEVEL=2
+LAMBDA=1.991479e-2   LD=0.115U          TOX=423E-10
+NSUB=1.0125225E+16   VTO=0.822163       KP=4.893760E-5    GAMMA=0.47
+UEXP=5.324966E-3     PHI=0.6            UO=599.496        CRIT=12714.2
+DELTA=3.39718E-5     VMAX=65466.1       XJ=0.55U          RSH=0
+NFS=5.666758E+11     NEFF=1.0010E-2     NSS=0.0           TPG=1.00
+CGSO=0.9388E-10      CGDO=0.9388E-10    CJ=1.4563E-4
+MJ=0.6               CJSW=6.617E-10     MJSW=0.31)
*
.MODEL PT PMOS (LEVEL=2
+LAMBDA=4.921086E-2   LD=0.18U           TOX=423E-10
```

```
+NSUB=1.421645E+15   VTO=-0.776658     KP=1.916950E-5   GAMMA=0.52
+UEXP=0.142293       PHI=0.6           UO=234.831       UCRIT=20967
+DELTA=1.0E-6        VMAX=34600.2      XJ=0.50U         RSH=0
+NFS=4.744781E+11    NEFF=1.0010E-2    NSS=0.0          TPG=-1.00
+CGSO=1.469E-10      CGDO=1.469E-10    CJ=2.4E-4
+MJ=0.5              CJSW=3.62E-10     MJSW=0.29)
*
*
```

SPICE model cards for NMOS and PMOS transistors representing a 0.8um CMOS or BiCMOS process (typical). Data courtesy of Mark Johnson and Norm Jouppi during EE371, "Advanced VLSI Design", at Stanford University, Spring quarter, 1990.

```
*
*
.MODEL NT NMOS (LEVEL=3
+  VTO=0.77      TOX=1.65E-8     UO=570          GAMMA=0.80
+  VMAX=2.7E5    THETA=0.404     ETA=0.04        KAPPA=1.2
+  PHI=0.90      NSUB=8.8E16     NFS=4E11        XJ=0.2U
+  CJ=6.24E-4    MJ=0.389        CJSW=3.10E-10   MJSW=0.26
+  PB=0.80       CGSO=2.1E-10    CGDO=2.1E-10    DELTA=0.0
+  LD=0.0001U    RSH=0.50)
*
.MODEL PT PMOS( LEVEL=3
+  VTO=-0.87     TOX=1.65E-8     UO=145          GAMMA=0.73
+  VMAX=0.00     THETA=0.223     ETA=0.028       KAPPA=0.04
+  PHI=0.90      NSUB=9E16       NFS=4E11        XJ=0.2U
+  CJ=6.5E-4     MJ=0.42         CJSW=4.0E-10    MJSW=0.31
+  PB=0.80       CGSO=2.7E-10    CGDO=2.7E-10    DELTA=0.0
+  LD=0.0001U    RSH=0.50)
*
*
```

SPICE model cards for bipolar transistors representing a 0.8um BiCMOS process (typical). Data courtesy of Mark Johnson and Norm Jouppi during EE371, "Advanced VLSI Design," at Stanford University, Spring quarter, 1990.

```
*
*
.MODEL  b1.6 NPN
+is=1e-18   ikf=4ma    bf=100   br=20      Vaf=25   Var=3V
+Re=50      Rc=400     Rb=800
+tf=15ps    tr=400ps
+Cje=8fF    Vje=0.9    mje=0.3
+Cjc=8fF    Vjc=0.5    mjc=0.2
+Cjs=30fF   Vjs=0.5    mjs=0.3
+nf=1.0     Eg=1.20    nc=1.5   xcjc=0.5   fc=0.5
*
*
```

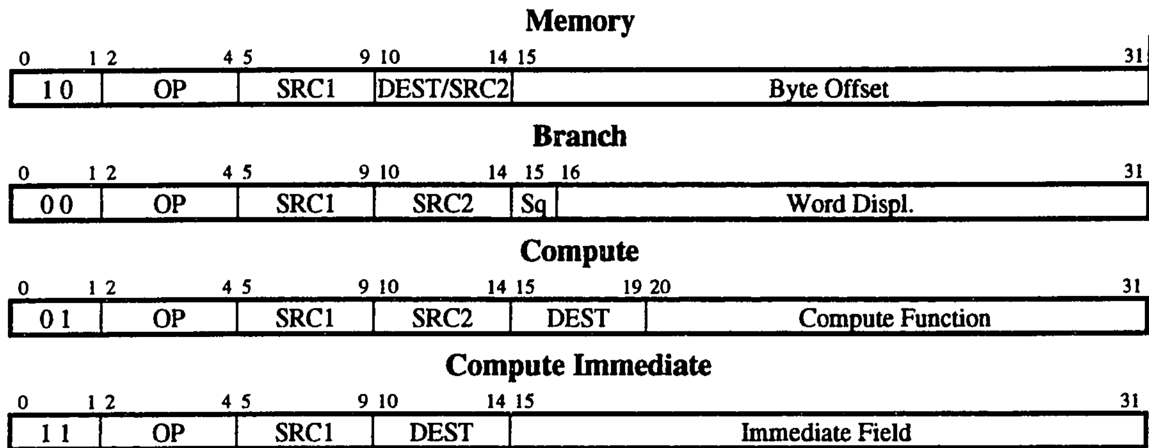
Appendix D

STRiP's Instruction Set

D.1 Instruction Set Architecture

Since STRiP is based on the MIPS-X architecture, it has a 32-bit data flow between datapath units and uses a simplified instruction set (relative to other commercial RISC processors). The instructions are grouped into four instruction formats. Figure D.1 shows the bit-fields used in each instruction format. There are 10 memory instructions (including coprocessor/FP processor transfer instructions), 12 branch instructions, 17 compute instructions, and seven compute immediate instructions. Table D.1 and D.2 lists the STRiP (MIPS-X) instructions. The use of simple, small, and therefore fast hardware structures optimize the pipeline's performance while providing more area for other features which maximize the performance of the processor (caches).

STRiP, like MIPS-X, uses a traditional five-stage pipeline. Results generated during the **ALU** processing stage are not written to the Register File until the final pipeline cycle (**WB**), simplifying exception handling. Bypassing is used to provide result data to instructions occurring before **WB**. The results of a load instruction cannot be bypassed to the **ALU** stage of the next sequential instruction, a *load delay* of one cycle is required between a load instruction and the next instruction using that load data. The software must fill the *load-delay slot* (the next sequential instruction after a load instruction) with an instruction which is independent of the load data.



SRC1, SRC2 = source specifiers for Register File read-ports. Byte Offset = load/store offset from SRC1
DEST = destination specifier for Register File write-port. Word Displ. = branch displacement from PC
OP = operation specifier within instruction group. Compute Func. = encoded computation
Sq = squashing specifier for branch prediction Immed. Field = immediate data field

Figure D.1: The STRiP (MIPS-X) instruction formats.

Instruction	Operands	Operation	Comments
Memory Instructions (including coprocessor/FP processor transfers)			
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 caches
stt	X[SRC1],SRC2	M[X+SRC1]:=SRC2	Store through - bypass caches
movfrc	CopInstr,DEST	DEST:=CopReg	Move from coprocessor
movtoc	CopInstr,SRC2	CopReg:=SRC2	Move to coprocessor
aluc	CopInstr	Cop execute CopInstr	Send/Exe. coprocessor instr.
aluf	FPInstr	FP Unit execute FPInstr	Send/Exe. floating-point instr.
Branch Instructions			
beq,beqsq	SRC1,SRC2,Displ.	PC:=PC+Displ. if SRC1 = SRC2	Branch if equal
bge,bgesq	SRC1,SRC2,Displ.	PC:=PC+Displ. if SRC1 ≥ SRC2	Branch if greater or equal
bhs,bhssq	SRC1,SRC2,Displ.	PC:=PC+Displ. if SRC1 ≥ SRC2	Unsigned branch if higher or same
blo,blosq	SRC1,SRC2,Displ.	PC:=PC+Displ. if SRC1 < SRC2	Unsigned branch if lower
blt,bltsq	SRC1,SRC2,Displ.	PC:=PC+Displ. if SRC1 < SRC2	Branch if less
bne,bnesq	SRC1,SRC2,Displ.	PC:=PC+Displ. if SRC1 ≠ SRC2	Branch if not equal

Table D.1: STRiP (MIPS-X) Memory and Branch instructions.

Instruction	Operands	Operation	Comments
Compute Instructions			
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_b	Subtract with no carry in
and	SRC1,SRC2,DEST	DEST:=SRC1 $\wedge$ SRC2	Logical AND
bic	SRC1,SRC2,DEST	DEST:=SRC1_b $\wedge$ SRC2	Bit clear
not	SRC1,,DEST	DEST:=SRC1_b	Logical INVERT
or	SRC1,SRC2,DEST	DEST:=SRC1 $\vee$ SRC2	Logical OR
xor	SRC1,SRC2,DEST	DEST:=SRC1 $\oplus$ 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 SRC2[30:31] bytes
rotlcb	SRC1,SRC2,DEST	DEST:=SRC1 byte rotated left	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"
Compute Immediate Instructions			
addi	SRC1,#N,DEST	DEST:=N + SRC1	Add immediate 17-bit sign extended
jpc			Jump using value in PC Chain. Used to return from exception handler.
jpcrs			Jump from PC Chain and restore state. Used to return from exception handler.
jspci	SRC1,#N,DEST	DEST:=PC+1, PC:=SRC1 + N	Jump indexed, save PC
movfrs	SpecReg,DEST	DEST:=SpecReg	Move from special register (PSW, PC-4, PC-1, MD)
movtos	SRC1,SpecReg	SpecReg:=SRC1	Move to special register (PSW, PC-4,. PC-1, MD)
trap	Vector		Trap to vector (256 vector addresses, eight words per vertor, located in low memory)

Table D.2: STRiP (MIPS-X) Compute instructions.

The results of a branch-instruction comparison is not known until the end of the **ALU** pipeline cycle. Before the resulting branch operation is known (taken or not-taken), two more instructions have been fetched and are in their **RF** and **ALU** pipeline stages. These instructions reside in the *branch-delay slots* for that branch instruction. To increase the probability of finding useful instructions to fill STRiP's two branch-delay slots,

squashing versions of the branch instructions are provided (beqsq, bgesq, bhssq, blosq, bltsq, and bnesq). These branch instructions make 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. Since the ALU is used to compute jump addresses, the jump instruction also has two delay slots.

There are several unique instructions/operations which need further explaination. Data moves to external coprocessors, or floating-point processors, occur during the **WB** pipeline cycle (unlike other memory store operations which occur during the **MEM** pipeline cycle). The instructions included in this group are *movtoc, aluc, ldf,* and *aluf.* Holding-off the data transfer until the write-back cycle allows the **MEM** stage of the *ldf* instruction to be used to access the internal memory system. Data moves from external processors (*movfrc* and *stf*) use the **ALU** pipeline stage to execute the external transfer. This allowed the **MEM** and **WB** pipeline stages to be used for the data store and register write-back operations. Since the external processors execute asynchronously to STRiP's pipeline sequencing, performing external processor transfers in this manner guarantees precise exception handling for the scalar pipeline.

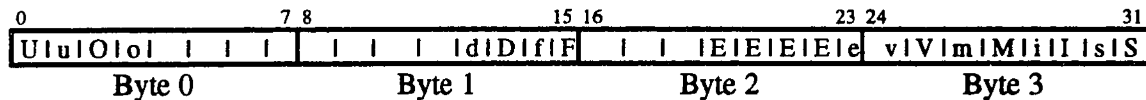
Multiplication is done with the simple 1-bit shift and add algorithm except that the computation is started from the most significant bit instead of the least significant bit of the multiplier. The instruction that implements one step of the algorithm is called *mstep.* For signed multiplication, the first step is different from the rest. If the most-significant-bit of the multiplier is 1, the multiplicand should be subtracted from 0. The instruction called *mstart* is provided for this purpose. Division is done, 1-bit at a time, by using a restoring division algorithm. Both operands must be positive numbers. The *dstep* instruction implements one step of the algorithm. For more details on multiplication and division in the MIPS-X architecture see Chow's book on the MIPS-X processor [Chow89].

STRiP's memory space is identical to MIPS-X. The addressing is consistently *Big Endian* [Cohen81]. The memory space is composed of 32-bit words. Load/store addresses are manipulated as 32-bit byte addresses, but only words can be read from memory. Only the most significant 30 bits are sent to the internal memory system. Since byte data can not be directly accessed, an instruction sequence is required to insert or extract bytes from a word. Instructions that affect the program counter (branches and jumps) generate word addresses. Offsets used to calculate load/store addresses are byte offsets, while displacements for branches and jumps are word displacements.

D.2 Processor Status Word

The PSW consist of two set of bits, *PSWcurrent* and *PSWother*. When an exception or trap occurs, the current machine state (*PSWcurrent*) is copied to *PSWother* so that it can be saved. Interrupts, PC shifting, and overflow exceptions are disabled and the processor is placed in system state. A *jpcrs* instruction (jump PC and restore state) causes *PSWother* to be copied back to *PSWcurrent*, restoring the machine state to its pre-exception values.

The PSW bit assignments match those used in MIPS-X and is shown in Figure D.2. The upper case bits correspond to the *PSWcurrent* data bits and the lower case bits correspond to the *PSWother* data bits.



The PSW bits are defined as follows:

I, i This bit is set to 0 when there is an interrupt request, otherwise it is a 1.

M, m Interrupt mask bit. When set to a 1 the processor interrupt is masked.

U, u Processor state bit. When 1 the processor is in user state, when 0 the processor is in system state. It can only be changed by a system process, interrupt or trap instruction.

S, s When set to 1, shifting of the PC Chain is enabled.

e Cleared when doing an exception or trap return sequence. Used to determine whether state should be saved if another exception occurs before the completion of the return sequence (three jump instructions).

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.

V, v The overflow mask bit. When set, trap on overflows is masked.

O, o This bit gets set on an overflow exception and is cleared on all other exceptions.

F, f The internal instruction cache enable bit. When set, the internal instruction caches are disabled. This bit can only be changed by a system process.

D, d The internal data cache enable bit. When set, the internal data caches are disabled. This bit can only be changed by a system process.

Figure D.2: The Processor Status Word (PSW).

Appendix E

C-element Logic Diagrams

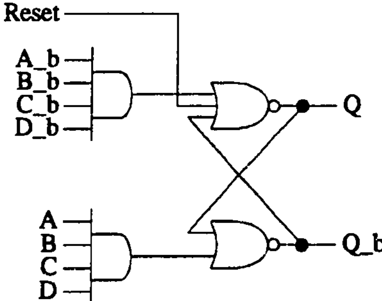


Figure E.1: 4-input C-element implementation using cross-coupled NORs.

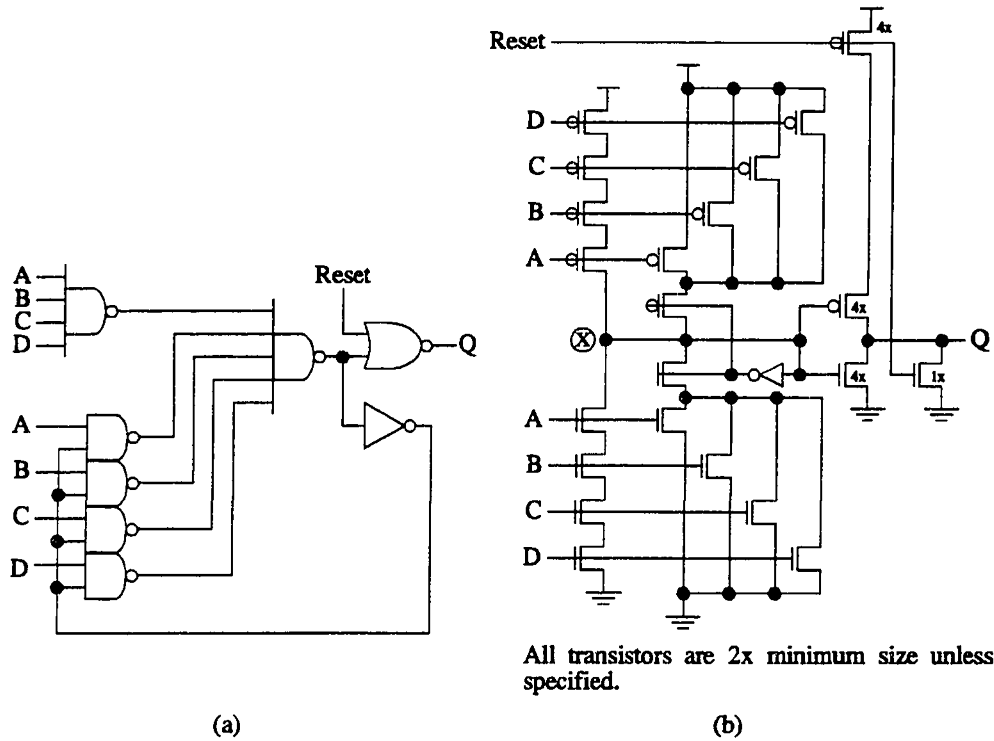


Figure E.2: 4-input C-element implementations using majority function circuits.

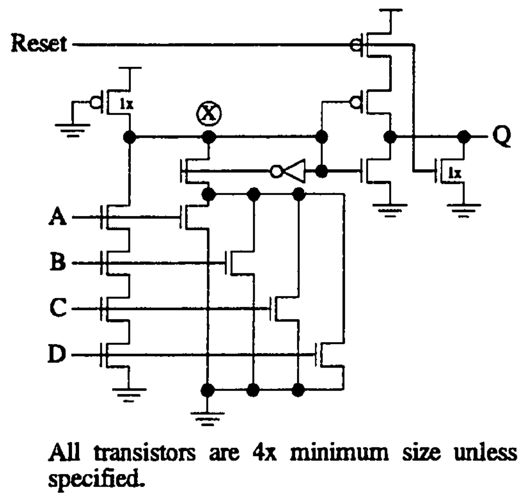


Figure E.3: 4-input C-element using pseudo-NMOS logic structure.

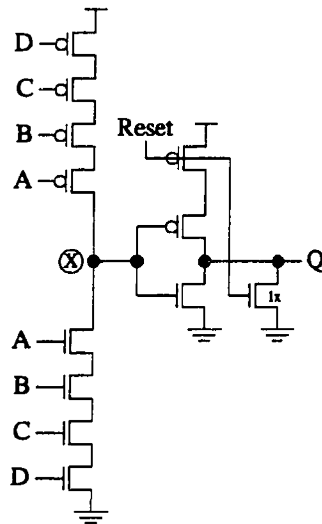


Figure E.4: 4-input dynamic C-element.

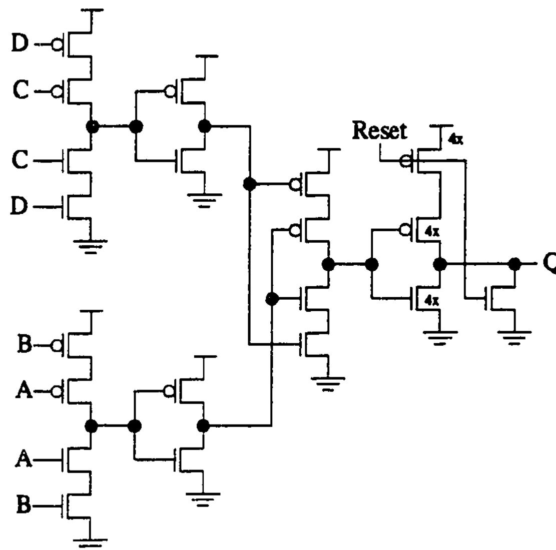
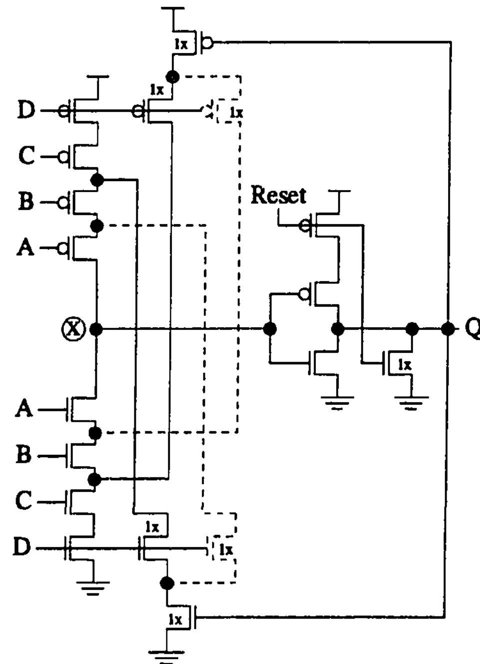


Figure E.5: 4-input dynamic C-element tree using 2-input dynamic C-elements.



All transistors are 4x minimum size unless specified.

Figure E.6: 4-input dynamic C-element with charge-sharing reduction circuitry.

Appendix F

External Interface Signal Descriptions

Cost, size, and availability were key factors in the determination of the number of interface signals used. The target module configuration was a *Quad Flat Pack* (plastic or metal) containing 240 pins [LSI Logic]. The QFP modules are less expensive and make more efficient use of board space than the *Pin Grid Array* (PGA) modules commonly used for microprocessors. The use of the plastic (PQFP) or metal (MQUAD) QFP will depend on the projected power dissipation and the amount of cooling required to optimize performance. It was determined that 25% of the pins were needed for power and ground. The remaining pins could be used for communication, testing, or configuration signals. The following is a a brief functional description of STRiP's module pins:

Power and Ground:

Vcc (27 pins) - 3.3V or 5.0V nominal (depending on the technology)

GND (27 pins)- Ground

Processor Address and Data Buses:

Address[0-31] - The processor's address bus is made up of 32 tri-state output signals. *HoldAck_b* low will tri-state the address bus to allow other bus masters to drive the bus (supports snooping and local shared memory MP). STRiP transfers information word or quad-word (cache line) aligned. Since the internal data cache is copy-back, all writes to external system memory are a cache-line wide (128-bits). The least significant address bits, *Address[28-31]*, are encoded to

indicate the targeted memory address and transfer size. Table F.1 gives the encoding scheme used. The address bus is also used to transfer coprocessor selects for **movtoc**, **movfrc**, and **aluc** instructions, and floating-point register selects during **ldf**, **stf**, and **aluf** instructions. $Address_0$, along with *Read/Write_b* and *FPAccess_b*, are used to distinguish between the different coprocessor transfers. Table F.2 gives the encoding for each coprocessor and processor transfer.

Address[28-31]	**Requested transfer size**	**Active Data Bus Signals**
1110	word	*Data[96-127]*
1101	word	*Data[64-95]*
1011	word	*Data[32-63]*
0111	word	*Data[0-31]*
0000	quad-word	*Data[0-127]*

Table F.1: Encoding of least significant address bits to support word and quad-word transfers

Data[0-127] - STRiP's data bus supports cache line transfers, coprocessor instruction/data transfers, and the capability of two double-precision floating-point data transfers in a single cycle (not defined in the present instruction set). The availability of a 128-bit data bus provides a significant amount of interface performance, while still supporting a low cost package. *Data[0-127]* are outputs during write transfers from the processor and inputs during processor read cycles. The data bus is tri-stated when ever *HoldAck_b* is active and *DBEnable_b* is inactive. STRiP also uses the data bus, when *DBEnable_b* is active during an external processor cycle, to provide dirty-line data during memory transfer cycles from other system processors (*HoldReq_b, HoldAck_b, MemReq_b, MemAck_b, Miss, Read/Write_b,* and *DBEnable_b* are low, indicating a write-snoop-hit of an internal dirty-cache-line).

Memory Control:

MemReq_b - *MemReq_b* is an I/O signal which allows the processor to transfer data to/from memory devices and keep internal caches coherent. The processor drives

MemReq_b when the BIU has a pending load/store operation to non-cachable system-memory space or cachable-memory space which was not present in the second-level cache. As an input, when *HoldAck_b* is active, the processor monitors *MemReq_b* along with *Read/Write_b* to determine the type of system memory transfer generated by another system processor. This allows STRiP to snoop its internal cache to maintain memory coherency.

MemAck_b - *MemAck_b* is an input signal that indicates when the externally addressed device has driven valid data on the data bus (read) or has accepted data from the data bus (write). Once *MemAck_b* is received, the processor's pipeline can be released (if stalled because of data dependencies).

Read/Write_b - *Read/Write_b* is an output when the processor controls the bus (*HoldAck_b* inactive) and an input during external processor cycles (*HoldAck_b* active). This signal indicates the memory transfer type and sets the data bus direction. *Read/Write_b* also designates the direction of a coprocessor transfer (see Table 5.3).

CacheReq_b - When the processor is transferring data to/from cachable memory space, it will drive *CacheReq_b* to request access to any external cache subsystems (normally a second-level cache). If the external cache indicates a hit (*CacheAck_b* active and *Miss* inactive) the processor continues processing. An external cache-read-miss (*CacheAck_b* and *Miss* active) will cause the BIU to generate an memory request to transfer the data from memory. With the internal data cache in copy-back mode, all external transfers to cachable memory space will be 128-bits wide. Therefore, a second-level-cache write miss cycle can immediately write the new line data into the cache (displaced dirty lines are written back on subsequent cycles). *CacheReq_b*, along with *CacheAck_b* and *Miss*, allows easy support of an optional second-level cache.

MemReq_b	*CacheReq_b*	*CopReq_b*	*FPAccess_b*	*Read/Write_b*	*Address0*	*Transfer Type*
↓	1	1	1	0	X	Non-Cachable Memory Write
↓	1	1	1	1	X	Non-Cachable Memory Read
1	↓	1	1	0	X	Cachable Memory Write
1	↓	1	1	1	X	Cachable Memory Read
↓	0	1	1	1	X	*Cachable Memory Read/Write, L2 Cache Miss
1	1	↓	1	0	1	movtoc instr.
1	1	↓	1	0	0	aluc instr.
1	1	↓	1	1	1	movfrc instr.
1	1	↓	1	1	0	(unused code)
1	1	↓	0	0	1	ldf instr.
1	1	↓	0	0	0	**aluf instr.
1	1	↓	0	1	1	stf instr.
1	1	↓	0	1	0	(unused code)

* Assumes a copy-back L2 cache. L2 cache will not indicate a miss on a 128-bit write cycle.

** New floating-point transfer instruction. Not provided in MIPS-X.

Table F.2: Processor signal encoding during external data transfers

CacheAck_b - CacheAck_b is an input and indicates when the external second-level cache has driven the data bus with valid data (read) or has latched the data from the data bus (write).

Miss - *Miss* is an input, driven by the second-level-cache, and indicates the results of the second-level-cache access. If Miss is active when *CacheAck_b* is driven active, the second-level cache does not contain the data-address requested. On a read-miss STRiP's BIU will generate a *MemReq_b* to request the data from system memory. On a write-miss, the second-level cache must remove the data from the data-bus before driving *CacheAck_b* active. The processor will ignore *Miss* during write-miss cycles, since no further action is required. With the first-level-caches in copy-back mode, a write to cachable memory space occurs only when a dirty-line is being written-back.

If *Miss* is high when Reset transitions from active to inactive (high-to-low transition), this indicates to the BIU that a second-level-cache does not exist. With no second-level-cache, the BIU will never drive *CacheReq_b* and will drive *MemReq_b* for all memory system transfers (cachable or non-cachable). If *Miss* is high when *MemAck_b* transitions low (indicating data available/accepted), the memory address is non-cachable and will not be stored in the internal or second-level caches.

SnoopCmplt_b- The processor uses *SnoopCmplt_b* to indicate the completion of an internal snoop cycle. A snoop operation occurs when an external processor owns the local bus (*HoldReq_b* and *HoldAck_b* active) and performs a memory transfer (*MemReq_b* active). The processor will snoop the contents of its internal caches, invalidate entries on a write-hit, and provide data on a hit to a dirty cache-line (DBEnable_b active). Table F.3 gives the signal encoding for each snoop response.

Exception Control:

Reset - *Reset* is an input and resets the processor to a known initial state. The internal caches are disabled and invalidated and the reset vector is placed on the address bus (*Address[0-31]* = 0x7fffff80). Since the caches are disabled, the processor will issue an external memory-read request to this address after *Reset* goes inactive.

Interrupt - *Interrupt* is a maskable interrupt input signal. When an interrupt is taken, it is necessary to save the PCs of all the instructions currently executing, allowing restarting after the interrupt is serviced. Because there is a branch delay of two, three PCs (instructions in RF, ALU and MEM stages) are saved in the PC Chain.

Exception_b - *Exception_b* provides a non-maskable interrupt function to support external events which need immediate attention (memory errors or bus time-out). *Exception_b* has priority over *Interrupt* and an internal trap instructions.

HoldReq_b, HoldAck_b, MemReq_b are active (low)			
Read/Write_b (input)	*Miss* (output)	*SnoopCmplt_b* (output)	response description
0	1	↓	write-snoop-miss or write-snoop-hit to clean cache line
0	0	↓	* write-snoop-hit to dirty cache line
1	1	↓	read-snoop-miss or read-snoop-hit to clean cache line
1	0	↓	* read-snoop-hit to dirty cache line

* External processor must drive *DBEnable_b* low to receive dirty-line data from processor.

Table F.3: Signal encoding for response to internal cache snoop operation

Coprocessor Control:

CopReq_b - *CopReq_b* is an output signal which when active, indicates a transfer request to/from a connected coprocessor or floating-point processor. During a coprocessor cycle, the address bus contains the coprocessor selects or floating-point register selects, while the least significant data bus signals (*Data[64-127]*) provide the medium for instruction/data transfer. If FPAccess_b is active when CopReq_b transitions to an active state (high-to-low transition), the transfer is between the processor and floating-point processor. *Read/Write_b* and $Address_0$ indicate the type of coprocessor transfer (see Table 5.3).

CopAck_b - *CopAck_b* is an input signal, driven by the attached coprocessors, which indicates the completion of a coprocessor data transfer. During a write to a coprocessor (**ldf**, **aluf**, **movtoc**, or **aluc** instructions), the coprocessor drives CopAck_b active to indicate data has been latch from the data bus. During coprocessor read operations (**stf** or **movfrc**) the coprocessor drives the same signal, indicating valid data on the data bus.

FPAccess_b - This signal replaced the MIPS-X coprocessor signals *FPReq[1-4]* and the use of *MemCycle* during coprocessor transfers. It was decided that all floating-point data transfer would occur through the internal first-level cache. Therefore, the floating-point instructions **ldf** and **stf** transfer data between the

first-level data cache and floating-point processor. The floating-point register address is driven on the least significant address bus bits (*Address[27-31]* to support 32 floating point registers). FPAccess_b is also driven active during the **aluf** instruction, with the data bus containing the floating-point instruction. Another change made to STRiP's coprocessor interface (versus MIPS-X) is that the transfer-too coprocessor cycles (**ldf**, **aluf**, **movtoc**, and **aluc**) are not sent to the BIU until the WB pipeline cycle. This eliminated the need for the *WBEnable* signal to guarantee precise interrupts. The BIU also allows the processor to continue processing, in parallel with the coprocessor transfer. transfer-from coprocessor transfers (**stf** and **movfrc**) occur during the ALU pipeline cycle and stall the pipeline until the transfer is completed.

Bus Control/Access:

HoldReq_b - *HoldReq_b* in an input signal, driven active by other system processors when they wish to access devices directly connected to STRiP's external interface. This signal forces STRiP's BIU to tri-state the address and data buses, and *Read/Write_b* signal when the BIU does not require the external bus. *HoldAck_b* is driven active once the BIU has released the external interface. HoldReq_b not only allows external processors to access processor complex devices, but it provides a means to snoop the internal first-level caches and external second-level cache.

HoldAck_b - Once the BIU receives a HoldReq_b active signal, HoldAck_b is driven active once the external interface signals (*Address[0-31], Data[0-127],* and *Read/Write_b*) are tri-stated.

DBEnable_b - The Data Bus Enable signal (*DBEnable_b*) signals the processor to drive the external data bus during a external processor cycle to a local bus device (*HoldReq_b* and *HoldAck_b* active). If a snoop cycle hits a dirty cache-line, STRiP may be asked to provide that dirty data. When *DBEnable_b* is driven active, the dirty cache-line is driven on to the external data bus.

BusReq_b - *BusReq_b* is driven by the BIU during external hold cycles (*HoldReq_b* and *HoldAck_b* active) to indicated that the processor request control of the external interface. This signal provides a means of processor sharing on the local bus based on the immediate needs of each connected processor. *BusReq_b* can be used to support a "fairness" protocol, providing equal access to all requesting processors.

Test Signals:

TestClk - *TestClk* is an input signal which can be used to sequence STRiP's pipeline when the dynamic clock generator is disabled (*DynClkReset_b* is active).

ScanClk - This clock input is used to scan test data to and from STRiP's internal functional units when *Reset* is active.

DynClkReset - This input signals allows external test hardware to disable the dynamic clock generator, providing a means to sequence STRiP's pipeline via an external test clock (*TestClk*).

ScanDataIn - *ScanDataIn* is an input signal which enables test information to be scanned into internal functional units (via *ScanClk*). *ScanDataIn* must be driven synchronously with respect to the scan clock.

ScanDataOut - *ScanDataOut* allows test results from scanned input data to be evaluated by external hardware. *ScanDataOut* is driven synchronously by the processor relative to *ScanClk*.

KNOB Control:

TCKNOB - This input is the control voltage for the tracking cell KNOB. *TCKNOB = 0V* allows the tracking cells to operate at their designed optimum delay. As *TCKNOB* increases in voltage, the tracking cells increase in delay. *TCKNOB = 5V* increases the tracking cell delays to 1.5x their optimum delays.

PGKNOB - This input is the control voltage for the pulse generator KNOB. PG*KNOB = 0V* allows the pulse generator to generate ø1 periods at their designed optimum width. As *PGKNOB* increases in voltage, the pulse generator output increases the width of ø1. *TCKNOB = 5V* increases the ø1 period to twice its designed optimum width.

Bibliography

1. Agarwal, A., *Analysis of Cache Performance for Operating Systems and Multiprogramming,* Ph.D. Thesis, Stanford University, May, 1987.

2. Agarwal, A., Chow, P., Horowitz, M., Acken, J., Salz, A., Hennessy, J., "On-chip Instruction Caches for High Performance Processors," *Proceedings of the Conference on Advanced Research in VLSI,* Stanford, CA, March, 1987, pp. 1-24.

3. Alpert, D., "Performance Tradeoffs for Microprocessor Cache Memories," Computer Systems Laboratory Technical Report CSL-TR-83-239, Stanford University, Stanford, CA, December, 1983.

4. Alpert, D., Flynn, M., "Performance Tradeoffs for Microprocessor Cache Memories," *IEEE Micro,* August, 1988, pp.44-55.

5. *Am29000 User's Manual,* Advanced Micro Devices Inc., Sunnyvale, CA, 1987.

6. Anantharaman, T.S., "A Delay Insensitive Regular Expression," *IEEE VLSI Technical Bulletin,* September 1986.

7. Baer, J.-L., Chen, T.-F., "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," *ACM,* 1991, pp. 176-186.

8. Baer, J.-L., Wang, W.-H., "On the Inclusion Properties for Multi-Level Cache Hierarchies," *Proceedings of the 15th Annual Symposium on Computer Architecture,* IEEE Computer Society Press, June, 1988, pp. 73-80.

9. Baer, J.-L., Wang, W.-H., "Architectural Choices for Multi-Level Cache Hierarchies," Technical Report TR-87-01-04, Department of Computer Science, University of Washington, January, 1987.

10. Bakoglu, H.B., Grohoski, G.F., Montoye, R.K., "The IBM RISC System/6000 Processor: Hardware Overview," *IBM Journal of Research and Development,* vol. 34, no. 1, January, 1990, pp.12-22.

11. Bassett, P., *A High-Speed Asynchronous Communication Technique for MOS VLSI Systems,* Massachusetts Institute of Technology, December, 1985.

12. Bazes, M., "A Novel Precision MOS Synchronous Delay Line," *IEEE Journal on Solid State Circuits*, vol. SC-20, no. 6, December, 1985, pp. 1265-1271.

13. Bennett, B.T., Pomerene, J.H., Puzak, T.R., Rechtschaffen, R.N., "Prefetching in a Multi-Level Hierarchy," *IBM Technical Disclosure Bulletin* 25(1):88-89, June, 1982.

14. Borg, A., Kessler, R.E., Wall, D.W., "Generation and Analysis of Very Long Address Traces," *Proceedings of the 17th Annual Symposium on Computer Architecture*, IEEE Computer Society Press, May, 1990, pp. 270-279.

15. Brent, G.A., *Using Program Structure to Achieve Prefetching for Cache Memories*, Ph.D. Thesis, University of Illinois, January, 1987.

16. Chapiro, D.M., "Globally-Asynchronous Locally-Synchronous Systems," Report No. STAN-CS-84-1026, Department of Computer Science, Stanford University, Stanford, CA, October, 1984.

17. Chow, P., *The MIPS-X RISC Microprocessor*, Kluwer Academic Publishers, 1989.

18. Chu, T.-A., "On the Models for Designing VLSI Asynchronous Digital Systems," *Integration*, Elsevier Science Publishers B.V., North-Holland, 1986, pp. 99-113

19. Cohen, E.I., King, G.M., Brady, J.T., "Storage Hierarchies," *IBM Systems Journal*, 28(1):62-76, 1989.

20. Colwell, R., Nix, R., O'Donnell, J., Papworth, D., Rodman, P., "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Transactions on Computers*, August, 1988, pp. 967-979.

21. David, I., Ginosar, R., Yoeli, M., "An efficient Implementation of Boolean Functions as Self-Timed Circuits," Technical Report No. 678, Dept. of Electrical Engineering, Technion and Israel Institute of Technology, September, 1988.

22. David, I., Ginosar, R., Yoeli, M., "Implementing Sequential Machines as Self-Timed Circuits," Technical Report No. 692, Dept. of Electrical Engineering, Technion and Israel Institute of Technology, November, 1988.

23. David, I., Ginosar, R., Yoeli, M., "Self-Timed Implementation of a Reduced Instruction Set Computer," Technical Report No. 732, Dept. of Electrical Engineering, Technion and Israel Institute of Technology, October, 1989.

24. Dean, M.E., Williams, T.E., Dill, D.L., "Efficient Self-Timing with Level-Encoded 2-Phase Dual-Rail (LEDR)," *MIT Conference on Advanced Research in VLSI*, March 1991.

25. Dean, M.E., Dill, D.L., Horowitz, M.A., "Self-Timed Logic Using Current-Sensing Completion Detection (CSCD)," *IEEE Conference on Computer Design*, October, 1991, pp. 187-191.

26. Dobberpuhl, D.W., "A 200MHz 64b Dual-Issue CMOS Microprocessor," *IEEE International Solid-State Circuit Conference*, San Fransico, CA, February, 1992.

27. Dobberpuhl D.W., Glasser L.A., *The Design and Analysis of VLSI Circuits*, Addison-Wesley, 1985.

28. Ebergen, J.C., "A Formal Approach to Designing Delay-insensitive Circuits," *Computer Science Notes*, Eindhoven University of Technology, October, 1988.

29. Eickemeyer, R.J., Patel, J.H., "Performance Evaluation of On-Chip Register and Cache Organizations," *Proceedings of the 15th Annual Symposium on Computer Architecture*, IEEE Computer Society Press, June, 1988, pp. 64-72.

30. Farrens, M.K., Pleszkun, A.R., "Improving Performance of Small On-Chip Instruction Caches," *Proceedings of the 16th Annual Symposium on Computer Architecture*, IEEE Computer Society Press, May, 1989, pp. 234-241.

31. Flynn, M.J., *Studies in Processor Design*, preliminary draft, Stanford University, Stanford, CA, 1990.

32. Gindele, B.S., "Buffer Block Prefetching Method," *IBM Technical Disclosure Bulletin*, July, 1977, pp.696-697.

33. Grohoski, G.F., "Machine Organization of the IBM RISC System/6000 Processor," *IBM Journal: Research Development*, vol. 34, no. 1, January, 1990, pp. 37-58.

34. Grove, R.D., Oehler, R., "IBM Second Generation RISC Processor Architecture," *Digest of Papers, COMPCON 90*, February, 1990, pp.166-172.

35. Grove, R.D., Oehler, R., "IBM RISC System/6000 Processor Architecture," *Microprocessors and Microsystems*, July/August, 1990, pp.357-366.

36. Hatamian, M., "Understanding Clock Skew in Synchronous Systems," *Concurrent Computations*, S.C. Schwartz, Plenum, 1988.

37. Hennessy, J.L., Patterson, D.A., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo, CA, 1990.

38. Hester, P.D., "RISC System/6000 Hardware Background and Philosophies," *IBM RISC Systerm/6000 Technology*, SA 23-2619, IBM Corp., 1990, pp. 2-7.

39. Hill, M.D., "A Case for Direct-Mapped Caches," *IEEE Computer*, vol. 21, no. 12, December, 1988, pp. 25-40.

40. Hill, M.D., *Aspects of Cache Memory and Instruction Buffer Performance*, Ph.D. Thesis, University of California, Berkeley, 1987.

41. Hill, M.D., Smith, A.J., "Evaluating Associativity in CPU Caches," *IEEE Transactions on Computers*, December, 1989, pp.1612-1630.

42. Horowitz, M., Chow, P., Stark, D., Simoni, R., Salz, A., Przybylski, S., Hennessy, J., Gulak, G., Agarwal, A., Acken, J., "MIPS-X: A 20 MIPS Peak, 32-bit Microprocessor with On-Chip Cache," *Journal of Solid State Circuits*, October, 1987, pp.790-799.

43. *i486 Microprocessor*, order no. 240440-001, Intel Corporation, April, 1989.

44. *i860 64-bit Microprocessor*, Intel Corp., order number 240296-002, April, 1989.

45. Jacobs, G.M., Brodersen, R., "Self-Timed Integrated Circuits for Digital Signal Processing Applications," *Proceedings of Third Workshop on VLSI Signal Processing*, Monterey, California, September 1988.

46. Jacobs, G.M., "Self-Timed Integrated Circuits for Digital Signal Processing," Memorandum No. UCB/ERL M89/128, University of California, Berkeley, CA, November 1989.

47. Jeong, D.-K., "Clocking and Synchronization Circuits in Multiprocessor Systems," Report no. UCB/CSD 89/505, University of California, Berkeley, April, 1989.

48. Jeong, D-K., Borriello, G., Hodges, D., Katz, R., "Design of PLL-based clock generation circuits," *IEEE Journal of Solid-State Circuits*, vol. SC-22, no. 2, April, 1987, pp. 255-261.

49. Johnson, M., Jouppi, N., notes from EE371, Advanced VLSI Design, Stanford University, Stanford, CA, Spring Quarter, 1990.

50. Johnson, W.M., *Super-Scalar Processor Design*, Ph.D. Thesis, Stanford University, Stanford, CA, June, 1989.

51. Johnson, M.G., Hudson, E.L., "A Variable Delay Line PLL for CPU-Coprocessor Synchronization," *IEEE Journal of Solid-State Circuits*, vol. 23, no. 5, October, 1988, pp. 1218-1223.

52. Jouppi, N.P., "Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU," *Proceedings of the 16th Annual Symposium on Computer Architecture*, IEEE Computer Society Press, May, 1989, pp. 281-289.

53. Jouppi, N.P., "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proceedings of the 17th Annual Symposium on Computer Architecture*, IEEE Computer Society Press, May, 1990, pp. 270-279.

54. Jouppi, N.P., Wall, D.W., "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," *ACM*, 1989, pp. 272-286.

55. Kane, G., *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1987.

56. Katevenis, M.G.H., *Reduced Instruction Set Computer Architecture for VLSI*, MIT Press, Cambridge, MA, 1985.

57. Killian, E., "MIPS R4000 Technical Overview," *Hot Chips III*, Stanford University, Stanford, CA, August, 1991.

58 Kogge, P.M., *The Architecture of Pipelined Computers*, McGraw-Hill, New York, NY, 1981.

58a. Komori, S., Takata, H., Tamura, T., Asai, F., Ohno, T., Tomisawa, O., Yamasaki, T., Shima, K., Asada, K., Terada, H., "An Elastic Pipeline Mechanism by Self-Timed Circuits," *IEEE Journal of Solid-State Circuits*, vol. 23, no. 1, February, 1988, pp. 111-117.

59. Lau, C.H., "SELF, A Self-Timed Systems Design Technique," *Electronics Letters*, vol. 23, no. 6, March, 1987, pp. 269-270.

60. Lee, R.L., Yew, P.-C., Lawrie, D.H., "Data Prefetching in Shared Memory Multiprocessors," CSRD Report 639, Center for Supercomputing Research and Development, University of Illinois, Urbana-Champaign, January, 1987.

61. Liou, K., "Design of Pipelined Memory Systems for Decoupled Architectures," Technical Report #617, Computer Sciences Department, University of Wisconsin, Madison, October, 1985.

62. Martin, A.J., "On the Existence of Delay-Insensitive Circuits," *MIT Conference on Advanced Research in VLSI*, March 1989.

63. Martin, A.J., Burns, S.M., Lee, T.K., Borkovic, D., Hazewindus, P.J., "The Design of an Asynchronous Microporcessor," Caltech-CS-TR-89-02, California Institute of Technology, Pasadena, CA, 1989 and Proceedings of the Caltech Conference on VLSI, March, 1989.

64. Mead, C., Conway, L., *Introduction to VLSI Systems*, Addison-Wesley, 1980.

65. McFarling, S., Hennessy, J., "Reducing the Cost of Branches," *Proceedings of the 13th Annual Symposium on Computer Architecture,* IEEE Computer Society Press, June, 1986, pp. 396-403.

66. McNiven, G.D., Davidson, E.S., "Analysis of Memory Referencing Behavior for Design of Local Memories," *Proceedings of the 15th Annual Symposium on Computer Architecture,* IEEE Computer Society Press, June, 1988, pp. 56-63.

67. Melear, C., "The Design of the 88000 RISC Family," *IEEE Micro,* vol. 9, no. 2, April, 1989, pp. 26-38.

68. Meng, T.H., *Asynchronous Design for Programmable Digital Signal Processors,* Ph.D. Thesis, UC Berkeley, 1988.

69. Meng, T.H., *Synchronoization Design for Digital Systems,* Kluwer Academic Publishers, 1991.

70. Molnar, C.E., Fan, T.P., Rosenberger, F.U., "Synthesis of Delay-Insensitive Modules," Journal of Distributed Computing, vol. 1, 1986, pp. 226-234.

71. Murakami, K., Irie, N., Kuga, M., Tomita, S., "SIMP (Single Instruction stream/Multiple Instruction Pipeline): A Novel High-Speed Single-Processor Architecture," *ACM,* 1989, pp. 78-85.

72. Muller, D.E., "Asynchronous Logics and Applications to Information Processing," *Proceedings of Symposium on Applications Switching Theory Space Technology,* 1963, pp. 289-297..

73. Nigh P., Wojciech M., "A self-testing ALU using Built-in Current Sensing," *IEEE Custom Integrated Circuits Conference Proceedings,* June 1989.

74. Nowick, S.M., Dill, D.L., "Synthesis of Asynchronous State Machines Using a Local Clock," *IEEE International Conference on Computer Design,* October, 1991, pp. 192-197.

75. Olukotun, K., Mudge, T., Brown, R., "Performance Optimization of Pipelined Primary Caches," preliminary draft of conference paper, 1991.

76. Patterson, D.A., Ditzel, D.R., "The Case for the RISC," *Computer Architecture News,* vol. 8, no. 6, October, 1980, pp. 25-33.

77. Patterson, D.A., "A RISCy Approach to Computer Design," *COMPCON 1982,* San Francisco, CA, 1982, pp. 8-14.

78. Przybylski, S.A., *Cache and Memory Hierarchy Design,* Morgan Kaufmann Publishers, San Mateo, CA, 1990.

79. Przybylski, S.A., "The Performance Impact of Block Size and Fetch Strategies," *Proceedings of the 17th Annual Symposium on Computer Architecture*, IEEE Computer Society Press, May, 1990, pp. 160-169.

80. Quach, N.T., Flynn, M.J., "High-Speed Addition in CMOS," Technical Report CSL-TR-90-415, Computer Systems Laboratory, Stanford University, Stanford, CA, February, 1990.

81. Rau, B. Ramakrishna, "Sequential Prefetch Strategies for Instructions and Data," Technical Report No. 131, Digital Systems Laboratory, Stanford University, January, 1977.

82. Rau, B.R., Rossman, G., "The Effect of Instruction Fetch Strategies Upon the Performance of Pipelined Instruction Units," *4th Annual International Symposium on Computer Architecture*, June, 1977, pp. 80-89.

83. Santoro, M., Horowitz, M, "A Pipelined Iterative Array Multiplier," *IEEE International Solid-State Circuits Conference*, February, 1988.

84. Seitz C., "System Timing," Chapter 7 in *Introduction to VLSI Systems*, eds. Mead C. & Conway L., Addison-Wesley, 1980.

85. Sherburne, R.W., "Processor Design Tradeoffs in VLSI," Report no. UCB/CSD 84/173, University of California, Berkeley, CA, April, 1984.

86. Short, R.T., Levy, H.M., "A Simulation Study of Two-Level Caches," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, June, 1988, pp.81-89.

87. Singh N.P., "A Design Methodology for Self-Timed Systems," M.Sc. Thesis, MIT Laboratory for Computer Science Technical Report TR-258, MIT, Cambridge, Mass., February 1981.

88. Smith, A.J., Lee, J., "Branch Prediction Strategies and Branch Target Buffer Design," *Computer* 17:1, January, 1984, pp.6-22.

89. Smith, A.J., "Cache Memories," *ACM Computing Surveys* (September 1982), 473-530.

90. Smith, A.J., "Sequentiality and Prefetching in Database Systems," *ACM Transactions on Database Systems*, September, 1978, pp.223-247.

91. Smith, A.J., "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer* 11, 12 (December 1978), 7-21.

92. Smith, A.J., "Problems, Directions and Issues in Memory Hierarchies," *Proceedings of the 18th Annual Hawaii Conference on System Sciences,* 1985, pp.468-476.

93. Smith, A.J., "Bibliography and Readings on CPU Cache Memories and Related Topics," *Computer Architecture News,* 14(1):22-42, January, 1986.

94. Smith, A.J., "Line (Block) Size Choice for CPU Cache Memories," *IEEE Transaction on Computers,* September, 1987, pp. 1063-1075.

95. Smith, J.E., Goodman, J.R., "Instruction Cache Replacement Policies and Organization," *IEEE Transactions on Computers,* March, 1985, pp.234-241.

96. Smith, M.D., Johnson, M., Horowitz, M.A., "Limits on Multiple Instruction Issue," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,* April, 1989, pp. 290-302.

97. So, K., Rechtschaffen, R.N., "Cache Operations by MRU Change," *IEEE Transactions on Computers,* June, 1988, pp. 700-709.

98. Stone, H.S., *High Performance Computer Architecture,* Addison-Wesley, Reading, MA, 1990, Second Edition.

99. Sutherland I., "Micropipelines," *Communications of the ACM,* vol. 32 no. 6, July 1989.

100. Tabak, D., *Advanced Microprocessors,* McGraw-Hill, New York, NY, 1990.

101. Tanksalvala, D., Hewlett-Packard Company, "A 90MHz CMOS RISC CPU Designed for Sustained Performance," *IEEE International Solid-State Circuits Conference,* February, 1990, pp. 52-53.

102. *The Sparc Architecture Manual,* Version 8, Part No. 800-1399-12, Sun Microsystems Inc., Mountain View, CA, December, 1990.

103. Todesco, A.R.W., Horowitz, M.A., "A 32-bit BiCMOS Adder Implementation," preliminary draft, Stanford University, Stanford, CA, January, 1989.

104. Unger, S.H., *Asynchronous Sequential Switching Circuits,* Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

105. Waser S., Flynn M., *Topics in Arithmetic for Digital Systems Designers,* Preliminary Second Edition, February, 1990.

106. Weiss, R., "RISC Processors: The New Wave in Computer Systems," *Computer Design,* May, 1987, pp. 53-73.

107. Weste N., Eshraghian K., *Principles of CMOS VLSI Design, A Systems Perspective*, Addison-Wesley, 1985.

108. Williams T.E., "Latency and Throughput Tradeoffs in Self-Timed Asynchronous Pipelines and Rings," Computer Systems Laboratory Technical Report CSL-90-431, Stanford University, Stanford, CA., May 1990.