Processor Design Tradeoffs in VLSI

By

Robert Warren Sherburne, Jr.

B.S. (Worcester Polytechnic Institute) 1978
M.S. (University of California) 1981

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Engineering

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: Carlo H. Séquin 4/13/1984
Chairman Date
Carlo H. Séquin 4/16/1984
David A. Hodges 4/16/84

.....

PROCESSOR DESIGN TRADEOFFS IN VLSI

Robert Warren Sherburne, Jr.

ABSTRACT

As the density of circuit integration is increased, management of complexity becomes a critical issue in chip design. Hundreds of man-years of design time are required for complex processors which are presently available on a few chips. This high cost of manpower and other resources is not acceptable. In order to address this problem, the Reduced Instruction Set Computer (RISC) architecture relies on a small set of simple instructions which execute in a regular manner. This allows a powerful processor to be implemented on a single chip at a cost of only a few man years. A critical factor behind the success of the RISC II microprocessor is the careful optimization which was performed during its design. Allocation of the limited chip area and power resources must be carefully performed to ensure that all processor instructions operate at the fastest possible speed. A fast implementation alone, however, is not sufficient; the designer must also consider overall performance for typical applications in order to ensure best results. Areas of processor design which are analyzed in this work include System Pipelining, Local Memory Tradeoffs, Datapath Timing, and ALU Design Tradeoffs. Pipelining improves performance by increasing the utilization of the datapath resources. This gain is diminished, however, by data and instruction dependencies which require extra cycles of delay during instruction execution. Also, the larger register file bitcells which are needed in order to support concurrency in the datapath incur greater delays and reduce system bandwidth from the expected value. Increased local memory (or register file)

capacity significantly reduces data I/O traffic by keeping needed data frequently in registers on the chip. Too much local memory, though, can actually reduce system throughput by increasing the datapath cycle time. Various ALU organizations are available to the designer; here several approaches are investigated as to their suitability for VLSI. Carry delay as well as power, area, and regularity issues are examined for ripple, carry-select, and parallel adder designs. First, a traditional, fixed-gate delay analysis of carry computation is performed over a range of adder sizes. Next, delays are measured for NMOS implementations utilizing dynamic logic and bootstrapping techniques. The results differ widely: the fixed-delay model shows the parallel design to be superior for adders of 16 bits and up, while the NMOS analysis showed it to be outperformed by the carry-select design through 128 bits. Such a result underscores the need to reevaluate design strategies which were traditionally chosen for TTL-based implementations. Single-chip VLSI implementations impose a whole new set of constraints. It is hoped that this work will bring out the significance of evaluating the design tradeoffs over the whole spectrum ranging from the selection of a processor architecture down to the choice of the carry circuitry in the ALU.

In this research I was supported for three years by a General Electric doctoral fellowship. The RISC project was supported in part by ARPA Order No. 3803 and monitored by NESC #N00039-78-G-0013-0004.

Table of Contents

CHAPTER 1:

INTRODUCTION

In the world of integrated circuits a revolution is taking place. Silicon chips, which only a decade ago contained several transistors or logic gates, now accommodate up to hundreds of thousands of transistors. Several 32-bit Central Processing Unit (CPU) implementations, as well as 64 and 256 Kilobit dynamic Random Access Memories (RAMs), have been produced on a single chip. Higher levels of integration offer systems which are not only smaller, cheaper, and less costly to operate than their predecessors: they offer higher performance as well. By shrinking the circuitry so that it can reside on less than a square centimeter of silicon area, wire delays are reduced dramatically. As a result, the user obtains higher performance at lower cost.

The rising complexity confronting the designer is a serious concern as device capacity on a chip increases. The design of a typical, 32-bit microprocessor requires 30 or more man-years. If this were performed by a single person, the fabrication technology will have changed so much that the original assumptions made regarding chip constraints would be grossly invalid. In order to shorten the elapsed design time, chips are partitioned into modules, each of which is constructed by a design team. Each team optimizes its module while conform-

ing to the specifications assigned by the project leader or manager. This divide-and-conquer approach is the traditional methodology in industry, where early product release yields a high return on the initial investment.

A disadvantage of the divide-and-conquer strategy is that no design team is familiar with the chip as a whole. This makes global optimization difficult, if not impossible. The overall organization of the system and its microarchitecture sets a fundamental limit on performance. A poor choice of microarchitecture will render a design doomed to a short life, if not outright failure, in the marketplace. The microarchitect's responsibility is to address this issue. He must be familiar with the architecture as well as the constraints of the fabrication technology. Since the chip constraints are constantly changing, design decisions must be periodically reevaluated.

Traditional CPU's such as in the IBM 360/370, DEC VAX-11/780, and so forth consist of several circuit boards, filled with standard Small and Medium-Scale Integration (SSI and MSI) packages. Performance is limited by the logic delays and wiring delays associated with the many inter-chip communication paths. Bipolar chips are normally used in such designs because they offer high transconductance. This means that a greater output current drive is available, which is important for reducing wire delays. Increasing the speed of signal propagation between chips requires more power per chip. Expensive cooling systems must then be added in order to control chip temperature.

In contrast, more recent 32-bit CPU designs have been implemented by Very Large-Scale Integration (VLSI) on a single chip. As a larger fraction of the system is placed on a chip, interchip wiring delays are reduced. Interchip delays are replaced with smaller, on-chip wire delays, increasing system performance. A ceiling for maximum transistor count is set by the limited chip area and power dissipation of the technology. Because Metal-Oxide Semiconductor

(MOS) transistors are smaller and consume less power than their bipolar counterparts, they are more attractive for VLSI. The poor transconductance of the MOS transistor increases off-chip signal delay, but this is offset by the reduced number of such delays inherent in higher levels of integration. MOS is presently the most popular fabrication technology for VLSI systems.

A VLSI processor is expensive to develop. Instead of relying on available SSI/MSI parts, the designer must perform circuit design, layout, and simulation at the device level. Optimization of one module on the chip affects the area, power, and timing available for the other modules. Wiring is costly in terms of chip resources: a 32-bit bus occupies a large amount of area in the planar layout. The number of Input/Output (I/O) pads is limited by chip periphery. This complicates testing by restricting access to internal state. Redesign is costly because new masks and wafers are required, delaying the product several months. As a consequence, the board-level design is more attractive for implementing complicated processors.

The high costs associated with a single-chip design are mainly due to complexity in design and testing. These penalties of single-chip implementation, can be alleviated by simplifying the CPU design. By reducing complexity and internal state of the machine, it will be simpler to design and test, and will be more likely to be free of design errors on the first try.

Several NMOS, single-chip VLSI microprocessors have been designed with this idea in mind. The RISC I [1], RISC II [2], and MIPS [3] implementations utilize low-level instructions, each of which requires a single machine cycle to execute. This regular execution timing simplifies the application of pipelining for high performance while remaining conceptually simple. Less instruction decoding is necessary, allowing small, fast Programmed Logic Arrays (PLAs) or simple decoders to be used.

This contrasts with commercial single-chip microprocessors, which dedicate over half the die area to microcode Read-Only Memory (ROM) for instruction decoding. In the Reduced Instruction Set Computer (RISC) design, area freed up by the reduced control circuitry is used for an expanded register file. Depending on the application environment, other functions may be incorporated on chip instead with the freed-up area to improve system performance.

In a single-chip design, datapath speed is a limiting factor in system performance. The datapath consists of the functional modules which manipulate data and provide for its temporary storage. Additionally, as its name implies, it includes the paths over which data may flow between these modules. Datapath cycle time is determined by delays in the main datapath modules and the communication overhead incurred to and from these modules during the machine cycle.

In order to minimize the datapath cycle time, the microarchitect determines what datapath modules are necessary and how they are orchestrated during the cycle. First, a functionally partitioned, two-dimensional representation of data flow is composed. Timing schemes are formulated which maximize concurrency of data operations and data flow. Next, the modules are optimally placed on the chip for minimum communication path delay and area. Several iterations may be required in order to stay within the limits of the chip resources.

This thesis studies fundamental design tradeoffs of importance to the microarchitect of a VLSI processor. All of these tradeoffs are interrelated; the microarchitect must decide which tradeoffs to make for best performance under specified conditions. These conditions include the fabrication technology and amount of chip resources available, as well as the type of environment for which the system is designed for.

Pipelining at the system level is investigated in Chapter 2. It presents the timing of the basic operations to be performed using increasing levels of concurrency. It is when overall datapath timing and resource allocation are determined that optimal topology of information flow must be considered.

Within the datapath itself, the local memory (register file) and ALU delays limit the speed which may be attained. A larger local memory effectively reduces data traffic arising from procedure calls and returns. Datapath bandwidth, however, is reduced due to the increased register cycle time. A conflict then exists between the desire for maximum datapath bandwidth, and the need to reduce data I/O overhead.

Since the local memory may occupy a significant portion of the chip, it is important to ensure that it makes effective use of available resources. Programming environments which include many nested procedures benefit significantly from a multiple-bank local memory scheme. On the other hand, those with few procedures may suffer from the increased register cycle time. In addition to the programming environment, the register-bank swapping strategy, overflow interrupt overhead, and data I/O bandwidth affect optimal memory size. These tradeoffs are investigated in Chapter 3.

Datapath bandwidth may be improved by pipelining the read and write operations of the register file. Different bit cells are required for different levels of pipelining. In general, the number of wordlines and bitlines in the cell must increase with higher concurrency. Register cell design issues relating to datapath timing are investigated in Chapter 4.

ALU delay can also be improved with increased parallelism. Adder delay analysis has traditionally been performed using the notion of fixed gate delay. This is appropriate for TTL implementations where performance is dominated by on-chip buffer delay. Such an approach is not suitable for VLSI, where

performance becomes limited by wiring and transistor parasitics. In Chapter 5, relative performance of ripple, partial lookahead, conditional carry, and parallel adders is analyzed and compared for both the constant gate delay model and an NMOS model which takes into account device parasitics and permits evaluation of alternative circuit design strategies for increased performance.

Interactions between these areas of design tradeoffs are investigated in Chapter 6. Designing for limited chip area and power resources is also discussed. Conclusions drawn from these areas of analysis are summarized in Chapter 7.

References

[1] D.A. Patterson, C.H. Séquin: "RISC I: A Reduced Instruction VLSI Computer," Proceedings of the 8th Symposium on Computer Architecture, ACM SIGARCH CAN, pp. 443-457, May 1981.

[2] M.G.H. Katevenis, R.W. Sherburne, D.A. Patterson and C.H. Séquin: "The RISC II Micro-Architecture," Proceedings of the IFIP TC10/WG10.5 International Conference on Very Large Scale Integration (VLSI '83), Trondheim, Norway, pp. 349-359, August 1983.

[3] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross: "Design of a High Performance VLSI Processor," Proceedings of the Third Caltech Conference on VLSI, Computer Science Press, pp. 33-54, March 1983.

CHAPTER 2:

SYSTEM PIPELINING

The goal of pipelining is to make more effective use of system resources, and in so doing, increase performance. Greater levels of pipelining lead to increased concurrency. The execution of one instruction (or microinstruction) may overlap initiation and completion of several others. This will be illustrated for datapath timing in a later chapter. In this chapter we will take into account the interaction with external (off-chip) memory.

Performance Improvement by Pipelining

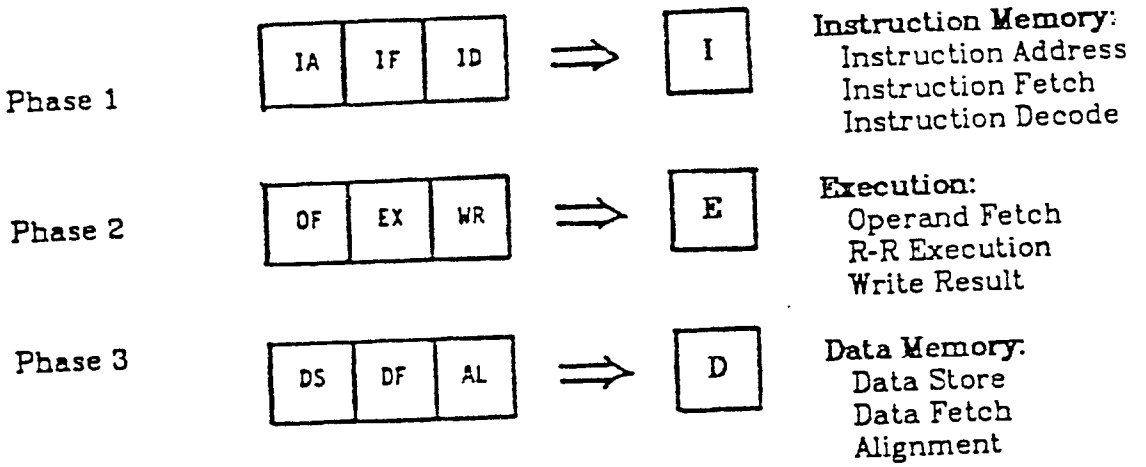


Figure 1: The Three Phases in Instruction Processing

There are several sequential steps involved in the execution of a single memory-to-memory instruction. The first phase of this cycle consists of the instruction fetch and decode. The next phase encompasses the register-to-register operation within the CPU, where data modifications are performed. For data I/O (Input/Output) instructions, this cycle consists of address calculations for LOADs, STOREs, or JUMPs. Finally, there is a third phase for LOAD and STORE instructions during which they access data memory. Support for sub-word data (e.g. bytes or half-words) may be included here. These three phases, each sub-divided into three subphases, utilize different resource groups (Figure 1).

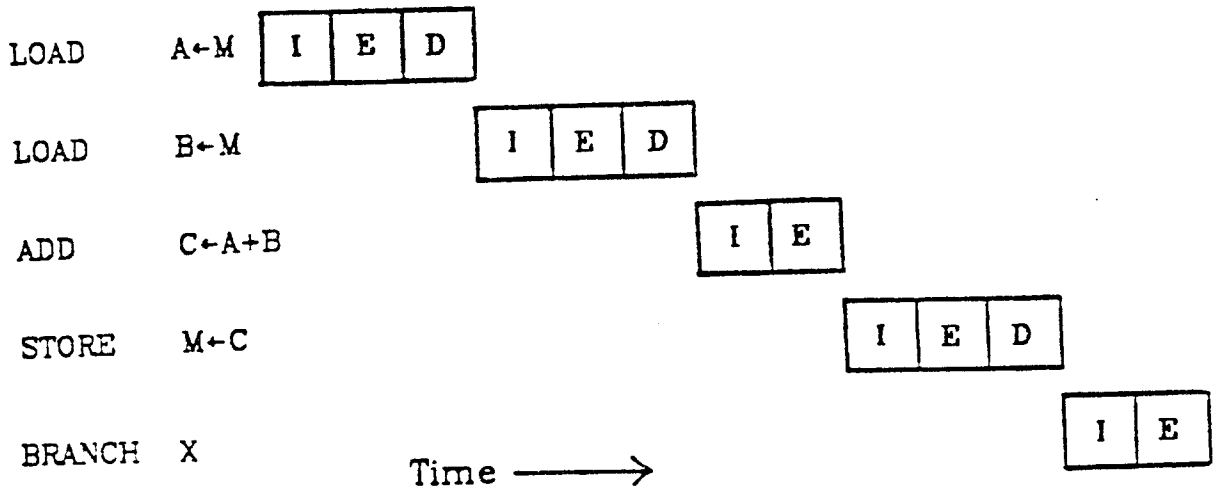


Figure 2: Timing of Sequential Execution

Timing for simple, serial execution is illustrated in Figure 2. Shown are five instructions, three of which access data memory. This approach makes poor usage of the resources on chip. For example, the ALU is used during less than half of the phases (one third for data I/O instructions). The I/O bus is not used at all during the execution phase.

Even very simple pipelining can increase performance substantially. A 2-way pipelined scheme is shown in Figure 3, in which instruction fetching overlaps execution of the previous instruction. This yields up to twice the bandwidth

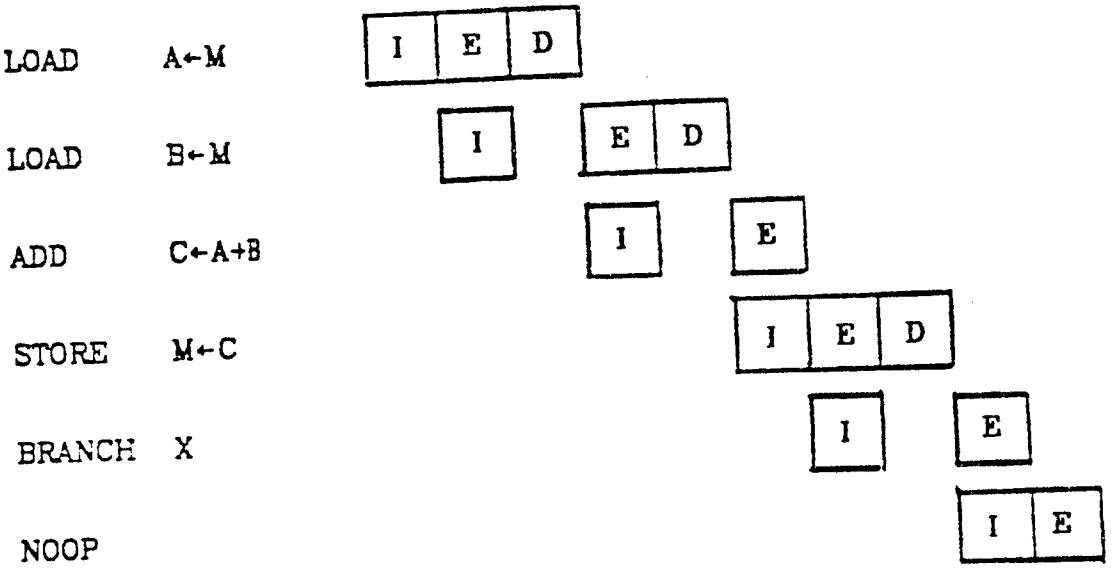


Figure 3: Two-Way Pipelined Timing Showing Branch Delay

of the serial scheme. It is assumed that only a single I/O operation is permitted in each phase, thus a wait-state is inserted while data I/O takes place. Hence, performance is I/O limited -- a single memory access occurs in each phase. This is the timing scheme of the RISC I and RISC II microprocessors [1].

In the event of a program branch, the branch address calculation is not completed until the following instruction has been fetched. This results in a delay of one phase before the target address is ready. In order to accommodate this delay, a NOOP (NO OPeration) instruction is inserted after the branch instruction. This creates overhead for all program branches. This overhead may be reduced by redefining the branch instruction in such a way that it is supposed to take effect only after the subsequent instruction. The code can then be reordered so that the instruction after the branch performs useful work prior to the branch occurrence [2]. Otherwise, a wait-state (in the form of a NOOP) is required. Code reorganization for this "delayed branch" optimization will be discussed in more detail later in this chapter.

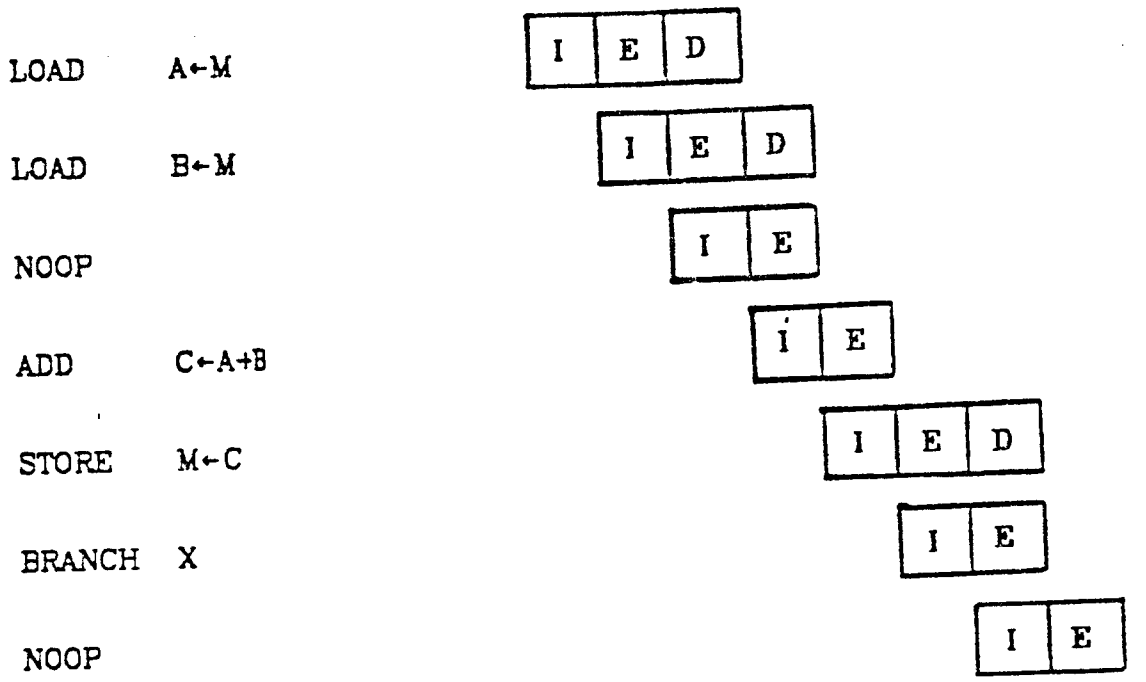


Figure 4: Three-Way Pipelined Timing Showing Data Delay

The pipelining can be further improved by permitting two memory I/O operations per phase, as shown in Figure 4. Multiple I/O operations per phase may be attained either by multiplexing a single port or by replicating ports. In this pipelining scheme a new instruction is initiated each phase, and as many as three instructions may overlap. Unoptimized branch delay remains at one phase, as in the previous scheme. However, the overlapped data memory access may now require wait states following LOAD instructions. As indicated in the figure, the data fetch is not completed prior to the execution of the following instruction. If this following instruction utilizes the fetched data as one of its operands, it must wait one phase. Code reorganization for data dependency optimization is similar to that for delayed branches (to be discussed). At best, this approach is up to three times faster than that of the sequential approach.

A four-way pipelined timing scheme is detailed in Figure 5. The execution phase is divided into two sub-phases: E_1 (register file read, with internal

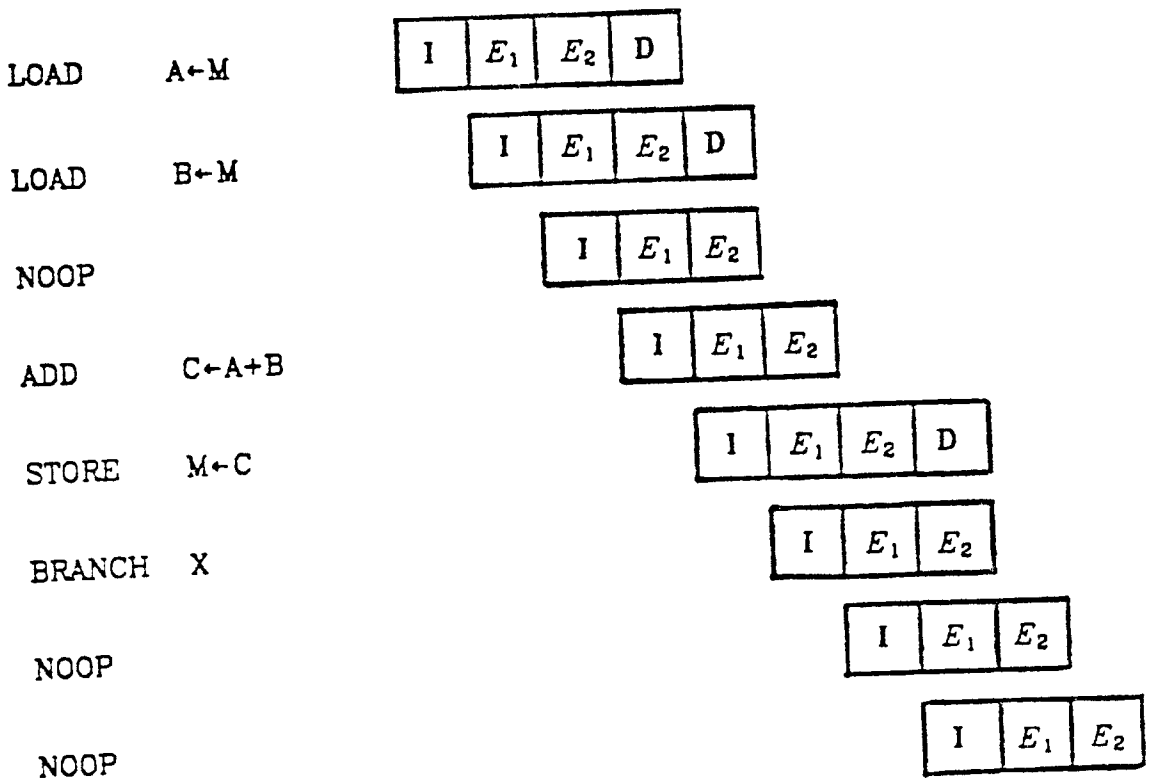


Figure 5: Four-Way Pipelined Timing

forwarding and overlapped write) and E_2 (ALU or shifter operation). The ALU and register file are used in every phase; this allows maximal resource usage. Overhead due to unoptimized data dependencies remains a single phase if the ALU result is directly forwarded to the concurrent read phase of the following instruction. Unoptimized branch overhead, however, is now two phases per branch. Two memory accesses per phase must be supported. This requires two I/O busses, as in the TMS 320 [3] and the MIPS [4] microprocessors. Additionally, the memory I/O cycle is now shortened to match half the execution time, so faster memory is necessary in order to realize the full gain of this added level of pipelining.

The processor-limited execution time for a given program using the various pipelining schemes discussed may be given as:

I. Sequential $2N + D$

II. 2-Way Pipelined $N + D + (J)$

III. 3-Way Pipelined $N + (D) + (J)$

IV. 4-Way Pipelined $\frac{1}{\alpha}[\, N + (D) + (\, 2J)\,]$

where N represents the number of coded instructions, with D data accesses and J jumps. Time is normalized with respect to a single register cycle. Optimizable overhead (branching, data dependencies) is shown in parentheses. Ideally, the factor α is 2 for the 4-way scheme, assuming that the factor α represents the available memory cycle speedup available over the previous schemes. Maximum performance improvement occurs for $\alpha=2$.

A comparison of ideal performance (fully optimizable) for these approaches is shown in Figure 6. Results are normalized with respect to the non-pipelined (sequential) case; the four-way scheme assumes $\alpha=2$. Bandwidth reduction caused by added data I/O cycles is shown in the shaded areas on the graph; the full shaded penalty occurs for the case of all instructions performing data LOADs. A fraction of this area would then pertain to the actual data I/O cost for a particular program.

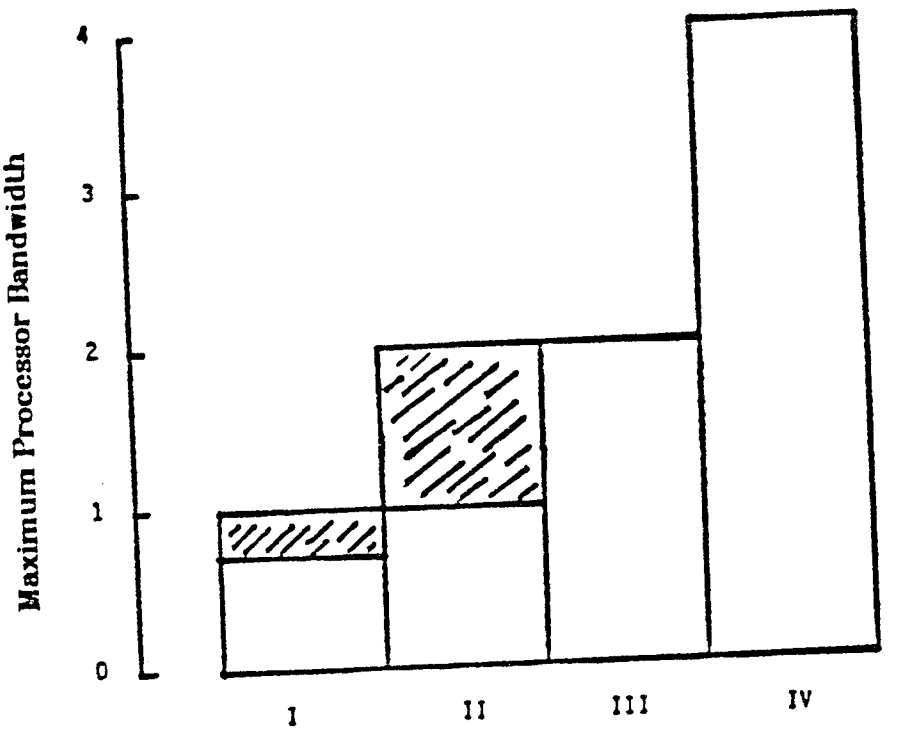


Figure 6: Performance Comparison of Pipelined Schemes (for α=2)

(shaded area indicates maximum possible data I/O overhead)

Optimization of Pipeline Dependencies

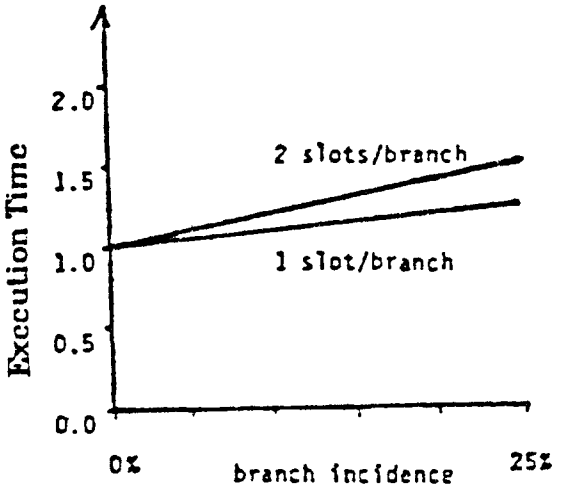
Ideal performance is proportional to the number of pipelining levels employed in the system. However, overhead due to data and instruction dependency reduces the efficiency of pipelining. This overhead is most significant for highly-pipelined machines. Code reorganization at the register-to-register instruction level may reduce this penalty inherent in pipelined implementations. The effective use of on-chip local memory also reduces data dependencies, by reducing data I/O traffic. Design tradeoffs concerning local memory will be investigated in a later chapter.

Code is optimized by reordering so that the required data or instruction is available when needed. The optimized jump or load (in the event of a branch

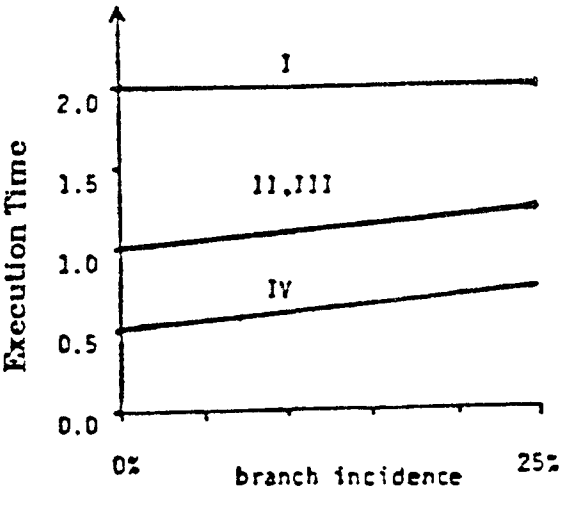
slot or data dependency optimization, respectively) is performed *earlier* than is needed in a sequentially executing program. Useful work may be done in the interim if the optimizing compiler can find an instruction to put into the empty slot. This is the goal of the *delayed jump scheme* [2]. This reordering is not easily done for conditional jumps for which branch prediction techniques must be utilized. Data from the VAX-11/780 indicate that conditional branches constitute 7% to 17% of the dynamic instruction count [5]. This penalty may be considered acceptable unless the number of pipelining levels is high. Stack machines are less flexible in terms of code reorganization; for this reason only register-based machines will be considered.

Conditional and unconditional branches, whether absolute or relative, constitute less than 25% of typical programs. Subsequent to optimization, unfilled branch slots for the MIPS processor vary from 3% to 24% (for a single slot per branch), and 21% to 50% (for two slots per branch) [6]. Results of this optimization vary depending on the programming environment and compiler technology. The IBM 801 compiler "...is able, generally, to convert about 60% of the branches in a program into the execute form."[7]. Figure 7(a) illustrates the effect of unoptimized branches on overall performance. In (b) this is compared among the four timing schemes. Estimated results of optimization are presented in (c), assuming the worst case MIPS unfilled branch slot incidence mentioned above (24% and 50%). As expected, the overhead of unfilled branch slots increases with pipelining. This overhead need not be directly proportional to the number of branches; the graph indicates an upper bound for convenience of discussion.

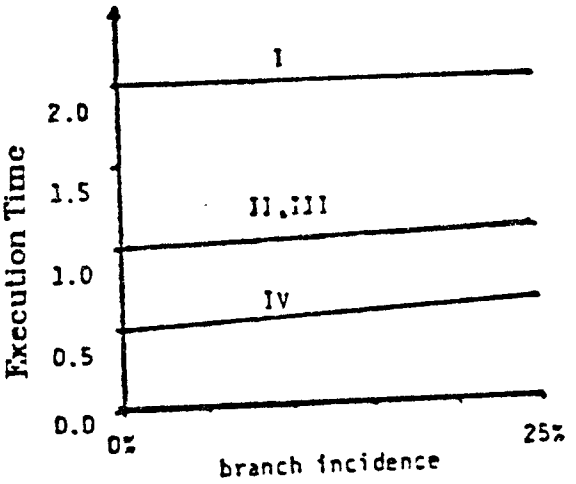
Dependencies arising from LOADs and branches are similar, with the exception that the former refers to data memory, and the latter refers to instruction memory. Such dependencies are inherently reduced with a register-based architecture. This is because the frequency of LOADs is reduced by depending



(a) Effect of slot overhead



(b) Comparison among pipeline schemes



(c) Comparison after optimization

Figure 7: Dependency Overhead and Optimization

mainly on local register storage of operands. It is expected that optimized data dependency overhead will be less than that for branches, since the dynamic LOAD count is typically less than 15%, which is observed for Quicksort [2]. Performance overhead of data dependencies may be determined with the aid of Figure 7.

Execution time overhead due to dependencies is also accompanied by a

corresponding increase in code size due to the NOOP instructions. Dependency optimization significantly reduces this overhead by replacing NOOPs with useful instructions. Elimination of remaining NOOPs may be accomplished by encoding wait states in the instructions responsible for the dependencies.

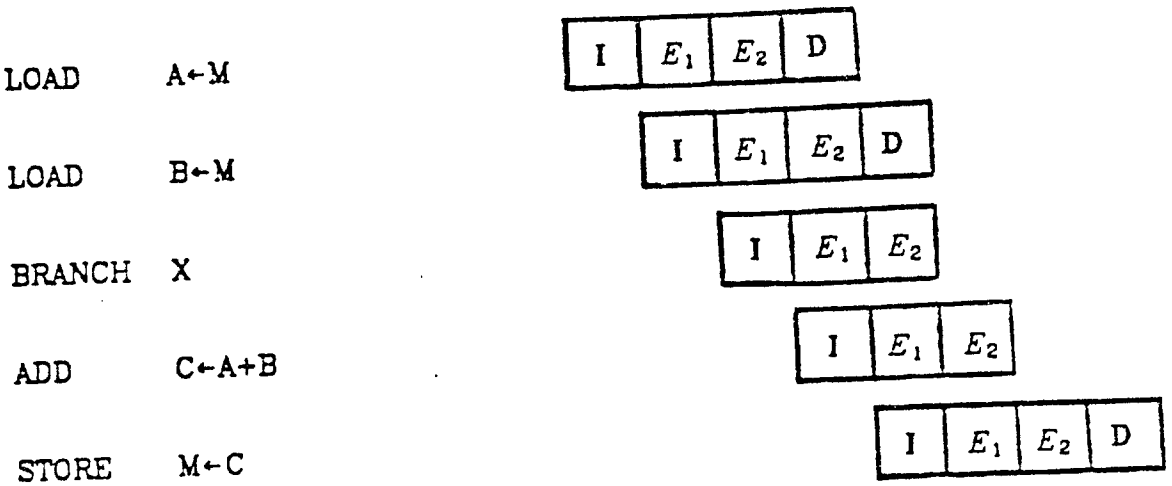


Figure 8: Reorganized Code for Four-Way Pipeline

After code reorganization for dependency optimization, the four-way pipelined instruction sequence of Figure 5 appears as shown in Figure 8. This example includes a performance improvement of 60% as well as a 37% reduction in code size.

This code optimization at the level of the machine cycle is important for highly pipelined machines. Complex instructions cannot be reordered at this level, and as a result perform poorly. Discussion of this issue for the MULTIPLY instruction is given in [8].

Pipelining Datapath Modules

Pipelining may also be applied at the submodule level within the datapath. For example, the ALU and register file may each exploit several levels of concurrency. Dependencies have been investigated as one side-effect which limits

performance of a pipelined system. Additional costs of pipelining are the additional storage elements and associated clocks needed to hold intermediate results. A high degree of pipelining entails more circuitry for these functions. A consequence may be extra delay in the throughput of all instructions. This delay results from the propagation time through the added storage elements and data skewing, as well as the required clock setup time for latching the intermediate results. This overhead can be significant in a highly-pipelined module, such as the parallel adder example in [9]. Datapath pipelining will be considered in more detail in Chapter 4. Typically, pipelining in a datapath module is employed only to the degree that it helps to alleviate a severe bottleneck in system performance. The degree of attainable pipelining is then determined by the slowest link in the system which cannot be improved; often this is the I/O cycle. The following chapter will address I/O-limited performance issues.

References

[1] D.T. Fitzpatrick, J.K. Foderaro, M.G.H. Katevenis, H.A. Landman, D.A. Patterson, J.B. Peek, Z. Peshkess, C.H. Séquin, R.W. Sherburne, K.S. VanDyke: "VLSI Implementations of a Reduced Instruction Set Computer," VLSI Systems and Computations, Carnegie-Mellon University Conference, Computer Science Press, pp. 327-336, October 1981. Also in: "A RISCy Approach to VLSI," VLSI Design, vol. II, no. 4, pp. 14-20, 4th qtr. 1981, and Computer Architecture News (ACM SIGARCH), vol. 10, no. 1, pp. 28-32, March 1982.

[2] D.A. Patterson, C.H. Séquin: "RISC I: A Reduced Instruction VLSI Computer," Proceedings of the 8th Symposium on Computer Architecture, ACM SIGARCH CAN, pp. 443-457, May 1981.

[3] S. Magar, E. Caudel, A. Leigh: "A Microcomputer with Digital Signal Processing Capability," International Solid-State Circuits Digest of Technical Papers, pp. 32-33, February 1982.

[4] J. Hennessy, N. Jouppi, F. Baskett, J. Gill: "MIPS: A VLSI Processor Architecture," VLSI Systems and Computations, Carnegie-Mellon University Conference, Computer Science Press, October 1981.

[5] D.W. Clark and H.M. Levy: "Measurement and Analysis of Instruction Use in the VAX-11/780," Proceedings of the 9th Symposium on Computer Architecture, ACM SIGARCH CAN, pp. 9-17, March 1982.

[6] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross: "Design of a High Performance VLSI Processor," Proceedings of the Third Caltech Conference on VLSI, Computer Science Press, pp. 33-54, March 1983.

[7] G. Radin: "The 801 Minicomputer," Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, ACM SIGARCH CAN, pp. 39-47, March 1982.

[8] M. E. Hopkins, "Compiling High Level Function on Low Level Machines," Proceedings of the International Conference on Computer Design, ICCD '83, pp. 617-619, Oct.-Nov. 1983.

[9] R.P. Brent and H.T. Kung, "A Regular Layout for Parallel Adders," IEEE Transactions on Computers, vol. c-31, no. 3, pp. 260-264, March 1982.

CHAPTER 3:

LOCAL MEMORY TRADEOFFS

A fundamental limitation to processor performance is set by the ratio of the amount of memory traffic and the available I/O bandwidth. The bandwidth limit for a given technology is set by area and power constraints. Only a limited number of I/O pads with their associated driver circuits can be placed on the chip periphery. Wire bonding technology has not followed in the footsteps of the shrinking transistor; pad size has remained constant over the years. Power dissipation of I/O drivers is determined by a delay-power product, because the off-chip loading is primarily capacitive. Multiplexing the pads for several I/O transactions per cycle requires a faster settling time, and hence greater power dissipation.

Memory traffic consists of two classes of information: instructions and data. Several options are available for reducing either component. At a high level, the set of machine instructions may be designed to include powerful constructs which are equivalent to many simple instructions. This has been done traditionally for large mainframe computers. There is much debate, however, with regard to its use in VLSI implementation. Advocates of the simpler Reduced Instruction Set Computer (RISC) maintain that, within the constraints of single-chip implementation, a complex instruction set is a poor use of limited chip resources [1]. Microprocessors to date have devoted the majority of their die

area to instruction microcode ROM. RISC implementations utilize this area to provide more local memory. Use of local memory keeps much of the needed data local to the processor and allows data traffic to be reduced. A register-based machine, for example, can store frequently-used operands in a fast, multiple-port register file.

Register allocation is performed by the compiler and requires no hardware overhead. It is performed independently for each subroutine, thus procedure calls require separate register banks or blocks of local memory. Register contents may have to be swapped out of local memory in order to make room for the next procedure. The I/O overhead entailed is costly and may actually increase data traffic over that required by off-chip operand storage. In order to overcome this performance degradation, the RISC I microprocessor organizes its local memory as multiple register banks, with each bank supporting a different procedure level [1]. In addition, adjacent banks overlap partially in order to facilitate parameter passing among subroutines. This approach drastically reduces data I/O traffic.

A multiprogramming or multitasking environment puts even more stringent demands on the performance of local memory. During each context switch a new register bank must be made available for the next executing program. In the case of a single register bank, its contents must be saved during every context switch. Multiple banks reduce this overhead by allowing context information from several programs to reside on chip. Since multitasking may be interrupt driven, arbitrary switching to any other process must be allowed. This contrasts with procedure level changes, where a stack organization will suffice. Microprocessors implementing context switching support include the Fujitsu FSSP (4 banks) [2] and the Siemens SAB 80199 (8 banks) [3].

All implementations with multiple register banks for procedure or context

switching support must accommodate register overflows. When the number of banks is exceeded, some swapping to external memory is required in order to make room on chip. When the capacity of a given bank is exceeded, external memory storage must be used, to store additional operands.

The alternative to the approach discussed above is a pure memory-to-memory architecture. With this scheme all data are stored in external memory, and no saving or restoring of registers is necessary upon a procedure call or context switch. On the other hand, all operand manipulations require data load and store operations to be performed. The above mentioned data I/O bottleneck, and the resulting greater latency compared to that of the register file architecture, may reduce performance. An example of a memory-to-memory microprocessor is the TMS 9995 [4].

The relative merit of the memory-to-memory approach versus a register-based machine depends on the programming environment and memory performance. Additional data traffic of the memory architecture must be compared to that incurred in a register machine when the procedure nesting depth or number of processes exceed the available number of register banks, as well as that occurring when capacity of each bank is exceeded.

Clearly, an infinitely large local memory is desirable since it can reduce off-chip data traffic to zero. Thus, the architects would like to have as much on-chip memory as possible. However, if the local memory is too large, it will also be slower, and system performance will be degraded. This chapter analyzes these tradeoffs.

Local Memory in RISC II

The RISC II is the second in a series of 32-bit, NMOS microprocessors developed at U.C. Berkeley [5]. The RISC instruction set consists solely of single

register-to-register operations [6]. This simple and regular implementation reduces control complexity, chip area, and design time, and it simplifies implementation of pipelined execution [7]. The simple RISC instruction set is an easier target for highly optimizing compilers than is a complex instruction set [8]. Proper optimization can also reduce dependency overhead inherent in pipelined implementations [9], thus making more effective use of the available datapath bandwidth.

A drawback of such an instruction set is that it requires higher memory bandwidth for fetching these instructions. Because the instructions are simpler, it often requires several of them to synthesize a complex instruction. This increases overall code size. On the other hand, the RISC microarchitecture includes support for subroutine call and return, one of the most time-consuming operations in typical high-level language programs for machines which keep variables in registers [6]. The number of register saves and restores is reduced by employing a local memory organized as multiple register banks. A new bank is allocated whenever a procedure is called. The banks represent stack levels, so that register save or restore need be performed only during stack overflows or underflows.

RISC II includes eight register banks, or windows, one of which is reserved for interrupt processing. At any one time there are ten registers local to the present procedure level. Additionally, there are six "high" and six "low" registers which are shared by adjacent procedure levels; these are used primarily for passing parameters and results between procedures. Each window swap (for save or restore) involves sixteen registers: the ten locals and one set of overlaps. Ten global registers are accessible from any procedure level, thus a total of 32 registers are addressable from any procedure.

Table I and Figure 1 show the relative execution time and performance of two C programs, "Tower of Hanoi" and "Puzzle", versus the number of windows on the chip. These results are based on earlier studies of procedure behavior and register file management overhead for RISC [10,11]. Both benchmarks nest to a depth of twenty. However, "Tower" has a very high rate of procedure calls and returns (19%) and thus makes intensive use of the multiple windows. Performance improves steadily through the use of, say, seven windows. "Puzzle", on the other hand, performs well with only one or two windows; it has only 0.7% calls and returns.

WINDOWS	1	2	3	5	7	9	∞
TOWER 19% Dynamic Call & Return	7.08	3.02	2.52	1.38	1.10	1.02	1.00
PUZZLE 0.7% Dynamic Call & Return	1.17	1.02	1.00	1.00	1.00	1.00	1.00

TABLE I: Normalized RISC II Execution Time

(relative to case of infinite windows)

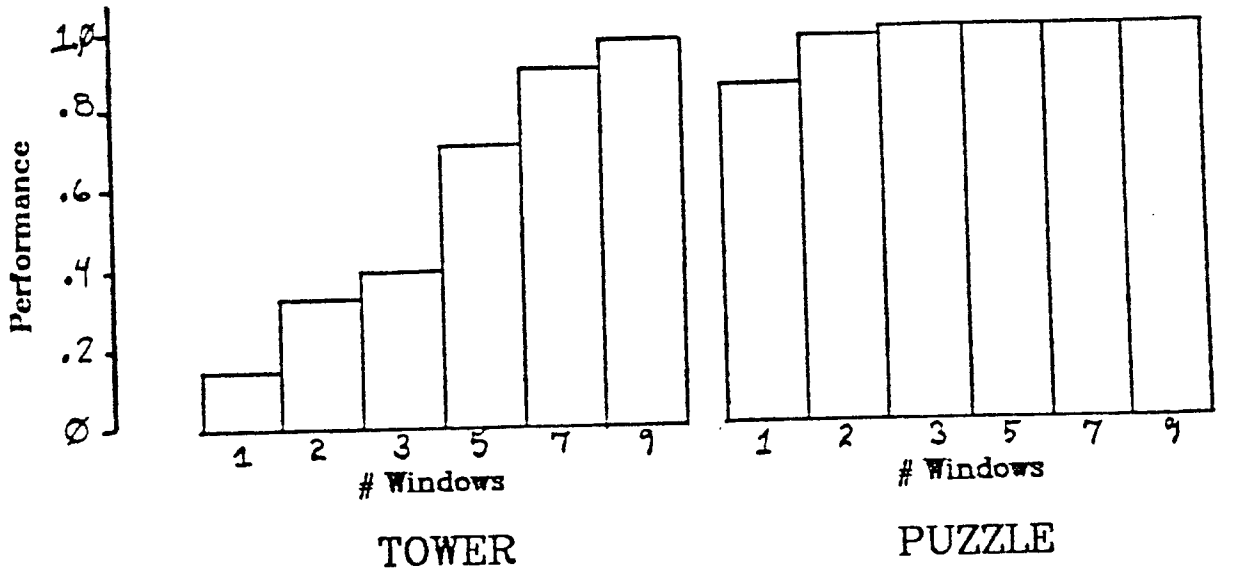


Figure 1: Normalized Performance of RISC II

Typical programs have a procedure call or return every twenty instructions, so the benchmarks shown here represent extremes [12,13]. In consideration of limited chip resources, a careful analysis of the program environment is desirable. If few procedures are used, a smaller local memory allows resources to be utilized for performance improvement in other areas.

Cost of Fixed-Size Window Swaps

The cost of register window overflow is determined by two factors: the overhead of servicing the interrupt caused by the overflow, and the cost of the actual data transfer between the register file and external memory. The RISC II microprocessor incurs a penalty of about thirty instructions for the window overflow/underflow interrupt routine. The single I/O bus implementation supports one memory access per cycle, which means that each load or store for the window swap takes two cycles. With sixteen registers per window, a total of 32 cycles are required. Although each window swap is costly, overflow/underflow occurs infrequently if there is a sufficient number of windows on chip. However, reducing local memory size below a program-dependent limit degrades performance significantly due to this high cost of swapping. With fewer windows, better swap interrupt and I/O support is crucial.

Since each swap utilizes the same protocol (sixteen adjacent registers swapped to/from the current window) better data I/O support can be provided. For example, a single instruction may provide all the necessary information for multiple register copying. Then it is not necessary to fetch individual Load or Store instructions for each register transfer. Furthermore, only a starting address needs to be passed to the memory controller in order to initiate the 16-word move. Two data words may then be passed on the bus each machine cycle. Compared to the present scheme, with one data word every two cycles interleaved with instruction fetching, throughput is increased by a factor of four.

At compile time, the dynamic procedure nesting profile is not known. Therefore the compiler cannot anticipate window overflows in a multiple-window implementation. For this reason, overflows must be detected on chip. It is the cost of handling this interrupt which accounts for thirty instructions in RISC II. For a processor with a single window, the compiler can anticipate the swaps and this overhead can be reduced. Every executed call or return requires a save or restore operation, respectively.

Table II presents RISC II execution time as a function of data I/O bandwidth and local memory size. The cost of each swap includes the thirty cycles for interrupt overhead, as well as the sixteen data word transfers. Since swap overhead for "Puzzle" is small, only "Tower" is considered here.

WINDOWS	1	2	3	5	7	9	∞
One-Half Data I/O Per Cycle	7.08	4.91	3.95	1.74	1.19	1.05	1.00
Single Data I/O Per Cycle	4.04	3.90	3.19	1.55	1.14	1.03	1.00
Dual Data I/O Per Cycle	2.52	3.39	2.81	1.46	1.12	1.03	1.00
Swapping Interrupt Overhead	0.00	1.89	1.43	0.36	0.09	0.02	0.00

TABLE II: Execution Time for "Tower" with Varying Swap Bandwidth
(includes interrupt overhead for multiple window cases)

Performance penalty due to interrupt overhead is illustrated by the shaded area in Figure 2. As before, seven or more windows are desirable for high performance, regardless of the level of data I/O support. This is because the interrupt overhead is so high. An exception is the single window case with dual data I/O per cycle, which is seen to provide better performance than register files

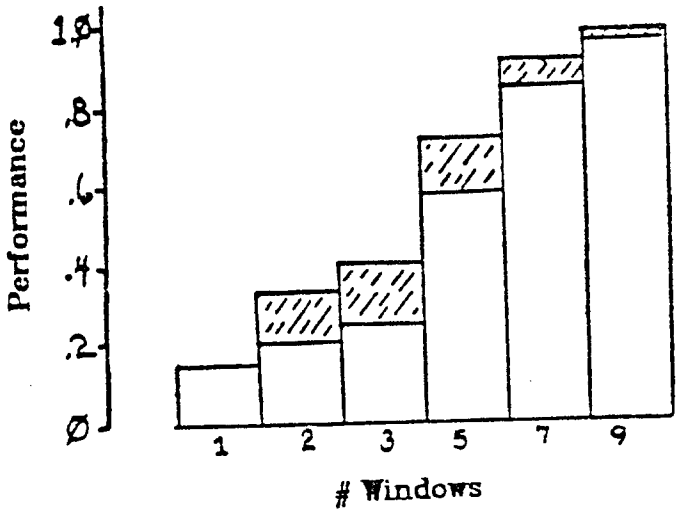


Figure 2: Performance with Overhead of RISC II Swap Interrupt

("Tower" Benchmark, half data I/O per cycle)

with two or three windows. With a large local memory, efficient interrupt support is not so important since few swaps occur. With few windows, though, it can be crucial. The remainder of this chapter will not consider the interrupt overhead; it is assumed that the area freed by reducing local memory size may be dedicated toward better interrupt support, and that overall swapping cost is dominated by the register traffic.

Improved Swapping Strategies

Thus far, we have only considered fixed-size window swaps for which all registers are transferred to/from memory. This scheme is attractive due to its simplicity and ease in providing higher swap bandwidth. However, such a scheme swaps all registers in the window, whether they were used or not. A study of several C programs has determined that on the average only four registers are used per procedure in RISC [10]. Therefore, the fixed-swap scheme performs four times the number of necessary save and restore data transfers.

Register Usage Record

In order to keep track of the registers actually used, a "dirty bit" may be employed. During each register write, a bit is set to indicate a register that needs to be saved when the window gets swapped. Swaps thus vary in length, depending on the number of bits set. The increased hardware complexity necessary to support such an approach, however, is undesirable.

An alternative is to utilize a single-word register usage mask for each window. Each bit which is set in the mask indicates usage of a specific register. A stack of such masks must be maintained on-chip for resident windows. During a window overflow, the appropriate mask is stored with the window contents. Additional logic is required in order to encode and decode the mask and provide for a mask stack.

A single window implementation does not require this hardware. The compiler can insert code before each call in order to save the registers used in the current window. Restoring registers after a return can be done on demand by the compiler. Since not all registers may need to be restored, further reduction in I/O is anticipated.

Save overhead may also be eliminated by performing a data memory write in parallel with all register file writes. This "store-through" scheme requires a dual-bus microarchitecture which can fetch an instruction and perform a data access in each cycle, such as the TMS 320 [14] or MIPS [15] microprocessors. Overall, swap overhead may then be reduced by more than a factor of eight.

Variable Window Size

Although we have described alternative strategies for local memory management, we have not yet addressed effective use of the on-chip memory

area. The above schemes reduce data traffic and off-chip register save space by a factor of four, but on-chip memory still attains only 25% utilization. If the register file windows can vary in size, such a waste of resources can be avoided.

For a variable-size window scheme, "bank" and "window" no longer need to be synonymous. The register file may be divided into fixed-size banks for regular and efficient swapping. Several procedures may reside within a single bank of, say, 16 or 32 registers; they may also span bank boundaries. This scheme requires additional hardware, in the form of pointers for each procedure domain, and an adder to calculate the physical addresses of the registers. Further details of variable-size window schemes may be found in [1].

WINDOWS	1	2	3	5	7	9	∞
Full-Bank Register Swaps	4.04	2.01	1.76	1.19	1.05	1.01	1.00
Partial Register Swaps	1.76	1.25	1.19	1.05	1.01	1.00	1.00
Partial Swaps with Store-Through	1.38	1.13	1.10	1.03	1.00	1.00	1.00
Variable Size with Full-Bank Swaps	2.01	1.21	1.08	-	-	-	1.00

TABLE III: Execution Time for "Tower" with various Swap Schemes

(one data I/O per cycle assumed for all cases)

Performance comparison of these schemes is presented in Table III. All cases assume single data I/O per cycle and four registers per procedure. Interrupt overhead, which occurs for the multiple window and variable size schemes, is not included here. The variable window scheme is assumed to utilize two equal-size banks, with total register count being the number of windows indicated in the table times sixteen. One of these banks is swapped during an overflow or underflow. Total number of registers is sixteen times the number of

windows indicated in Table III. Significant performance improvement is observed for implementations with few windows using these alternative swap schemes. By using more efficient swapping strategies, high performance may be attained with less chip area dedicated to local memory.

Register File Delay

Up to this point, only I/O limited performance has been discussed. From the designer's point of view, attention should also be focused on datapath bandwidth. Especially for RISCs, where each execution cycle consists of a uniform register-to-register operation, the datapath cycle time determines maximum system performance. The machine cycle of the RISC II consists of a dual-port register read, followed by an ALU or shift operation; these latter operations overlap the register write of the previous instruction and bitline precharge of the next instruction. The machine cycle is then limited directly by the register file read-write-precharge cycle time.

The register file cycle time increases with local memory size and depends on several design parameters. Read delay consists of two components: wordline assertion, and bitline discharge (Figure 3). The wordline, or addressing, delay is proportional to the gate capacitance loading of the access transistor. This delay then is linearly proportional to word size. In technologies where the wordline itself is the dominant resistance in addressing delay, such as with a polysilicon wordline, this delay follows the square of word length. Periodic buffering of the address (as for the dynamic ripple carry ALU) is necessary to reduce this delay to a linear function of word size. Bitline discharge delay increases with the product of the resistance of the wordline access transistor and the bitline loading capacitance. This delay then increases proportionally with memory word capacity.

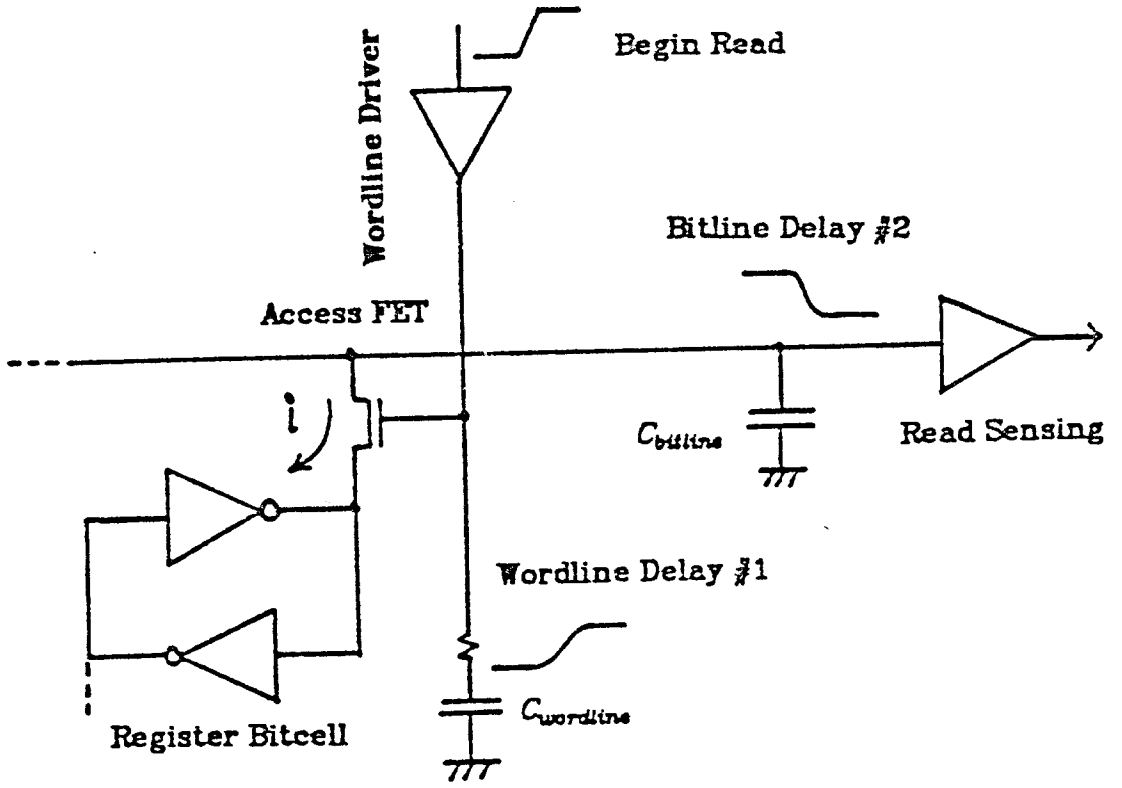


Figure 3: Register File Read Delay

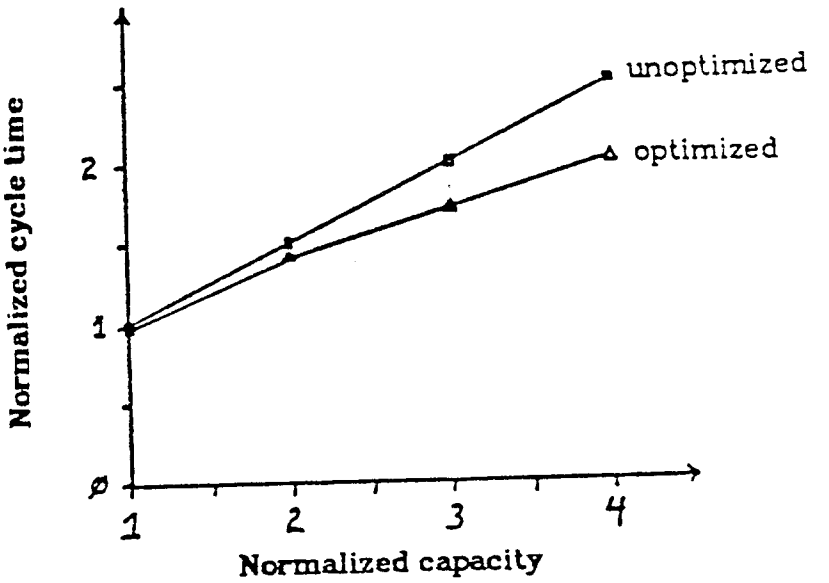


Figure 4: Cycle Delay versus Local Memory Capacity

Total register delay is therefore expected to increase proportionally with local memory size. However, some optimization is allowed by the wordline

access transistor. Reducing its width allows faster addressing at the cost of increased bitline discharge delay. Optimal performance is attained when both delays are equal [16]; this implies an access transistor size (and hence register cycle delay) which is proportional to the square root of the memory capacity. The write and precharge delays are then affected similarly. As a result, data-path cycle time increases with the square root of word length or word capacity of the register file (Figure 4). This effect must be taken into account to obtain a more realistic performance estimate for various local memory sizes.

WINDOWS		1	2	3	5	7	9
Normalized RISC II Register Cycle Time (ignoring swaps)		1.00	1.22	1.41	1.73	2.00	2.24
Full-Bank	$\frac{1}{2}$	7.08	3.68	3.55	2.39	2.20	2.28
Register Swaps with Varying Data I/O per cycle	1	4.04	2.45	2.48	2.06	2.10	2.26
	2	2.52	1.84	1.95	1.90	2.05	2.24
Partial Register Swaps		1.76	1.53	1.68	1.82	2.02	2.25
Partial Swap with Store-Through		1.38	1.38	1.55	1.78	2.01	2.24
Variable Size with Half Bank Swaps		2.01	1.48	1.52	-	-	-

TABLE IV: Datapath Bandwidth Limited Execution Time

(smaller local memory is faster; using "Tower" benchmark)

Table IV includes the effect of variable register cycle delay. The partial register swap schemes using a single window yield the best performance, only rivaled by the variable size scheme with its additional hardware complexity. A single window, fixed-swap implementation with dual data I/O per cycle approaches the performance of the seven window RISC II with half data I/O per cycle. Execution time for "Puzzle", with little swap overhead, follows the

register cycle time dependence with memory size; it executes nearly twice as fast with one window as it does with seven. Inclusion of register file delay then has a major impact on the relative merits of the swapping schemes. Of course, the smaller local memory implementations have higher datapath bandwidth, so a similarly faster external memory is required in order to realize this performance. Local memory size is traded off against memory bandwidth requirements.

Chip design tradeoffs in VLSI must be made using both architectural and circuit design considerations. One measure of the cost of local memory, increased delay, yields two minimum execution time solutions: I/O limited, and datapath bandwidth limited. Because of the limited number of pads that can be placed on a chip, memory I/O is a severe bottleneck in system performance. For this reason, a large local memory was chosen for RISC II. Presently, memory speed is increasing, making datapath bandwidth a more critical limit to system performance. In the future, increased chip resources will make possible a greater local memory hierarchy [17]; I/O bandwidth may then be replaced by datapath bandwidth as the primary factor limiting system performance.

References

[1] M.G.H. Katevenis: "Reduced Instruction Set Computer Architectures for VLSI," Doctoral Dissertation, Computer Science Division, University of California, Berkeley, 1983.

[2] N. Inui, H. Kikuchi, T. Sakai: "16-bit C-MOS Processor Packs in Hardware for Business Computers," Electronics, pp. 182-186, June 16, 1981.

[3] J. Gosch: "Microprocessor does Multitasking in Real Time," Electronics, pp. 71-71, Nov. 3, 1982.

[4] J. Schabowski: "Tough Control Tasks Take 16 Bits," Electronics, pp. 91-94, Dec. 18, 1980.

[5] M.G.H. Katevenis, R.W. Sherburne, D.A. Patterson, C.H. Séquin: "The RISC II Micro-Architecture," Proceedings of the IFIP TC10/WG10.5 International

Conference on Very Large Scale Integration (VLSI '83), Trondheim, Norway, pp. 349-359, August 1983.

[6] D.A. Patterson, C.H. Séquin: "A VLSI RISC," IEEE Computer, vol. 15, no. 9, pp. 8-21, September 1982.

[7] J. D. Wright: "Relation of Microcode to Future Machine Design," COMPCON Digest of Papers, pp. 104-106, March 1983.

[8] G. Radin: "The 801 Minicomputer," Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, ACM SIGARCH CAN, pp. 39-47, March 1982.

[9] T. Gross: "Code Optimization Techniques for Pipelined Architectures," COMPCON Digest of Papers, pp. 278-285, March 1983.

[10] D. Halbert, P. Kessler: "Windows of Overlapping Register Frames," CS 292R Final Class Report, Computer Science Division, University of California, Berkeley, Spring 1980.

[11] Y. Tamir, C.H. Séquin: "Strategies for Managing the Register File in RISC," IEEE Transactions on Computers, vol. c-32, no. 11, November 1983.

[12] D.R. Ditzel, H.R. McLellan: "Register Allocation for Free: The C Machine Stack Cache," Proceedings, Symposium on Architectural Support for Programming Languages and Operating Systems, Palo Alto, pp. 48-56, March 1982. (ACM: SIGARCH CAN vol. 10, no. 2, SIGPLAN Notices vol. 17, no. 4)

[13] D.W. Clark, H.M. Levy: "Measurement and Analysis of Instruction Use in the VAX-11/780," Proceedings, Symposium on Architectural Support for Programming Languages and Operating Systems, Palo Alto, pp. 9-17, March 1982.

[14] S. Magar, E. Caudel, A. Leigh: "A Microcomputer with Digital Signal Processing Capability," Proceedings of the International Solid-State Circuits Conference, pp. 32-33, February 1982.

[15] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross: "Design of a High Performance VLSI Processor," Proceedings of the Third Caltech Conference on VLSI, Computer Science Press, pp. 33-54, March 1983.

[16] A.M. Mohsen, C.A. Mead: "Delay-Time Optimization for Driving and Sensing of Signals on High-Capacitance Paths of VLSI Systems," IEEE Journal of Solid-State Circuits, vol. sc-14, no. 2, pp. 231-239, April 1979.

[17] D.A. Patterson, C.H. Séquin: "Design Considerations for Single-Chip Computers of the Future," IEEE Journal of Solid-State Circuits, vol. sc-15, no. 1, pp. 44-52, February 1980,

CHAPTER 4:

DATAPATH TIMING

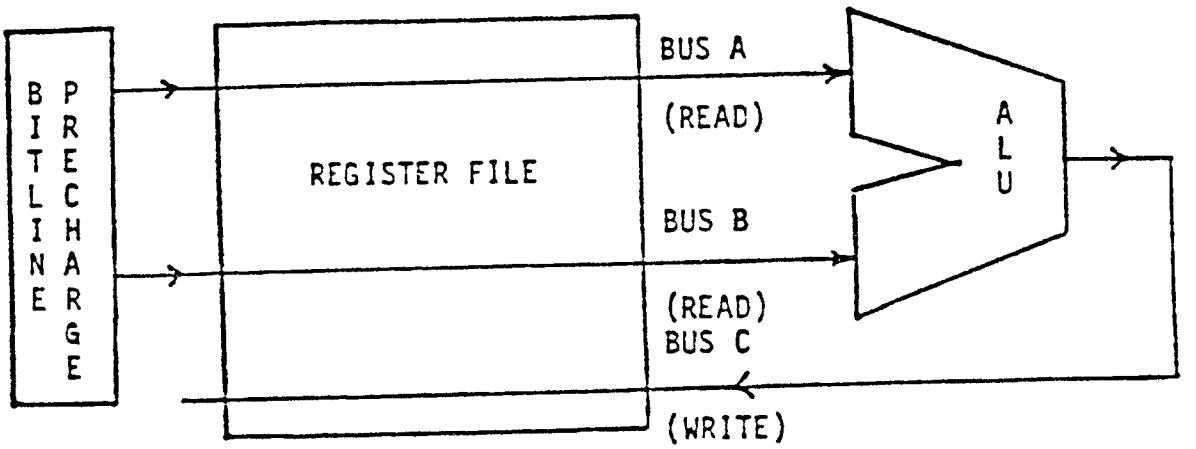
A critical issue for register-to-register machines is the register file organization and timing. A variety of bitline and wordline configurations yields a wide range of datapath bandwidth. The microarchitect must determine what silicon resources are required for each approach and study the tradeoffs between the various timing schemes. The purpose here is to review a variety of possible datapath timing schemes in order to develop an intuitive understanding of the tradeoffs involved.

Various datapath designs are compared here in terms of the speed at which register-to-register (R-R) operations can be executed. This determines the performance limit of simple, register-oriented machines, such as the RISC I [1], MIPS [2], and the 801 [3]. Regular instruction cycle timing yields simple, regular implementation of the control circuitry. Regularity of instruction execution permits pipelining without requiring complicated hardware pipeline interlocks.

The benefits of instruction pipelining may also be exploited by a microcoded machine. Pipelining favors the use of regular, register-to-register microoperations, rather than the use of irregular, but fast, microcoded implementation of the relatively few dynamically executed, compiled complex instructions [4].

When viewed from the datapath, the actual instruction coding is not visible. Therefore no assumptions regarding the machine's instruction set are made in this chapter. Instruction coding is a higher level issue, for which tradeoffs can be assessed once a programming environment has been chosen.

For simplicity, overhead of I/O will be ignored. Performance limits due to off-chip communication were considered in the previous chapter. Program counter logic will not be considered explicitly; it may be encompassed for our purposes within the domain of the addressable register file. Therefore, ideal system speed, to be discussed in this chapter, is determined directly by the datapath bandwidth.



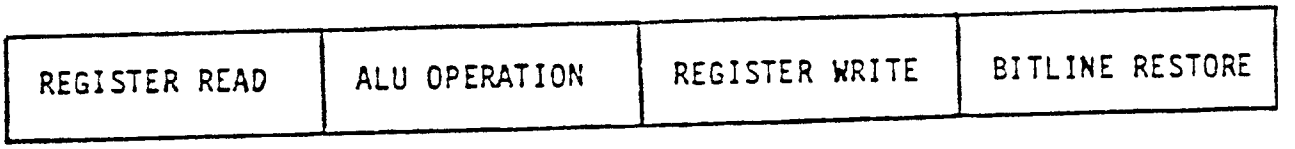


Figure 1: Register-to-Register Timing Example

The fundamental datapath operations to be performed during each register-to-register cycle are shown in Figure 1. They include reading two operands from the register file, performing an ALU or shifter operation, and writing the result back into the register file. In NMOS implementations, the read

bitlines must be restored to a logic "1" prior to reading. This is necessary if the read bitline is dynamically precharged, and/or the bitline is used for both reading and writing through the same register cell port. This is to ensure the read value is valid, and that the read operation does not accidentally *write* into the cell.

A critical, limiting factor in determining allowable concurrency is the register file organization: three of the four basic operations concern it. Various bitline (bus) and wordline (addressing) organizations will be considered in discussing timing schemes which exploit greater levels of concurrency than the sequential example above.

Shared Bitline Register Files

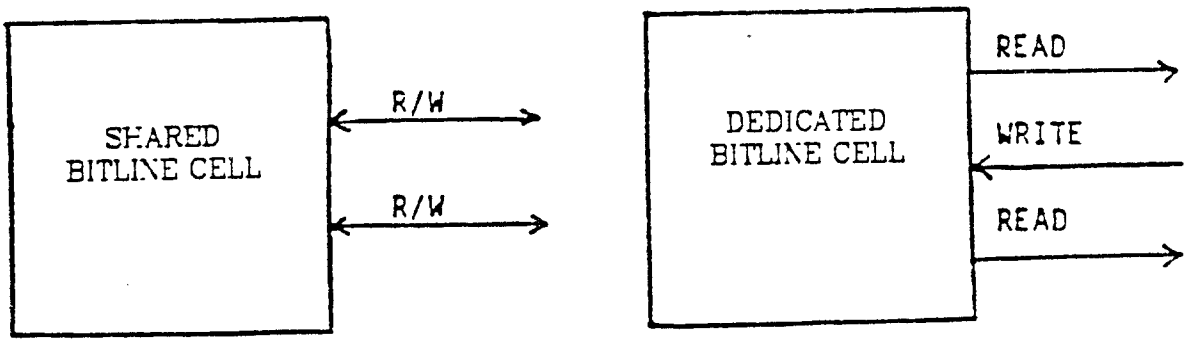


Figure 2: Shared and Dedicated Bitline 2-Port Cells

Of fundamental importance in determining allowable concurrency within a register file read-write cycle is the bitline arrangement. A *shared bitline* organization utilizes the same bitlines for both reading and writing. A *dedicated bitline* approach utilizes separate bitlines for reading and writing, allowing some overlap of these operations (Figure 2). Advantages of the shared bitline design include its economy of area, due to fewer bitlines and fewer transistors. This cell may also be faster, since its smaller size reduces loading on the wordlines. This helps make up for the reduced level of concurrency attainable with fewer

bitlines. Overall system timing for the shared bitline approach is identical to that in Figure 1.

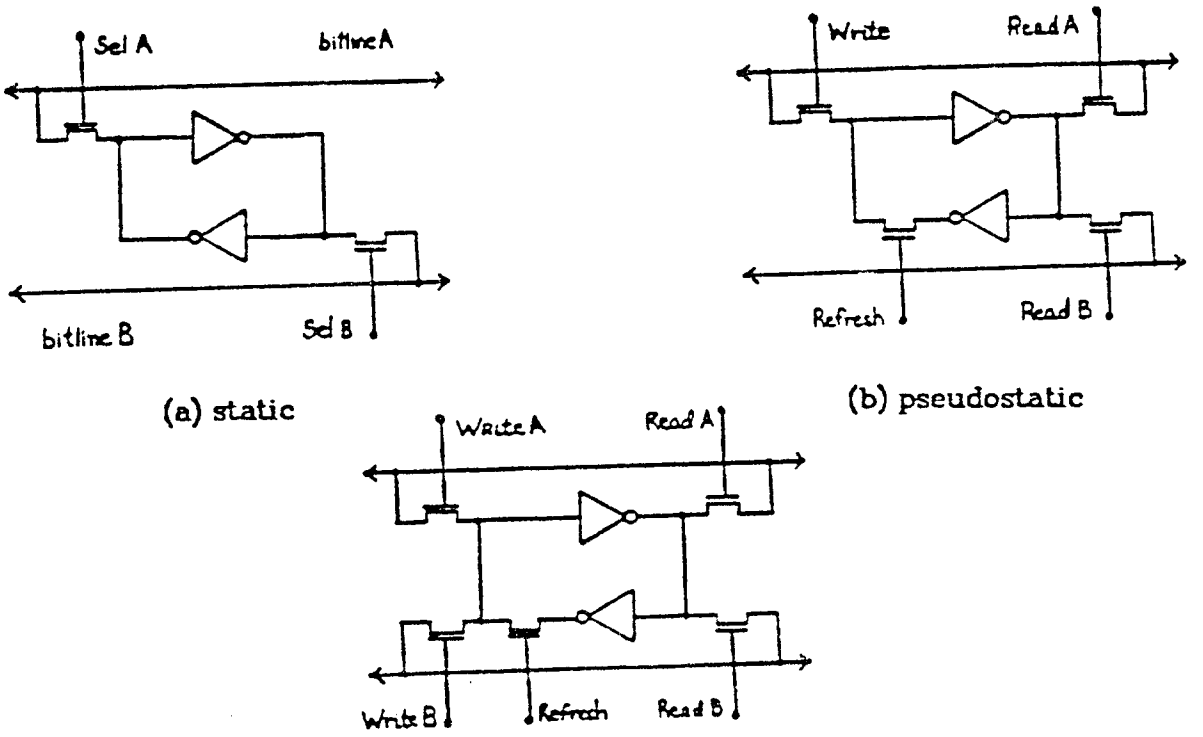


Figure 3: Shared Bitline Register Cells

Another design choice exists for the shared bitline cell, leading to two different structures. One may use shared (Figure 3(a)) or dedicated (Figure 3(b)) wordlines for the read and write operations. In the first case there is one set of wordlines, to be used for both reading and writing to common ports; writing is usually performed via complementary signals driving two ports. This is a derivation of typical commercial Random Access Memory (RAM) designs. The shared wordline approach leads to a fast and compact cell. Such a static design has been implemented in the RISC II, which requires a large, dual-port register file [5]; its symmetry was crucial in attaining a compact layout.

Care must be taken to ensure that reading onto a precharged bus will not

change the state of the shared wordline cell. As shown in Figure 3(a), each bit cell inverter may be considered to be transformable into a 2-input NOR gate by the addition of the wordline transistor. If the bus is initially discharged to ground, wordline access will behave as a NOR input and set the cell state (write). If the bus is precharged prior to accessing, reading will cause either no change (if both sides of the wordline FET are high), or bus discharge will occur (cell node grounded). The wordline transistor in this latter case acts as a voltage divider. A narrow transistor will have a greater voltage drop across it during discharging, allowing the cell state to be maintained. A wide transistor will reduce the voltage drop, allowing the high logic level of the bus to change the cell state. This *read disturb* problem then constrains the size of the wordline transistors, limiting speed of bitline discharge during reading.

Dedicated wordlines form separate ports for reading and writing. A pseudostatic design, as shown in Figure 3(b), allows temporary breaking of the feedback loop (by the refresh FET). This allows single-port writing (with no wordline FET voltage drop) and eliminates read disturb by breaking the feedback loop. The wordline transistor size is not constrained as before. In the case of a dual-port pseudostatic cell with two pairs of wordlines, two locations in the register file may be written simultaneously (Figure 3(c)). This scheme was chosen for the Caltech OM-2 [6], as well as the HP FOCUS chip [7]. In order to maintain data integrity, the refresh transistor must be clocked periodically. This is usually done every cycle, during the bitline restore phase.

A disadvantage of this design is its asymmetry, due to the refresh transistor and the single inverter drive for reading. The first inverter is not used for reading because its gate can only be charged to VDD-VT by the enhancement pass transistors. Overall cycle time then must include the worst case delay of discharging both bitlines.

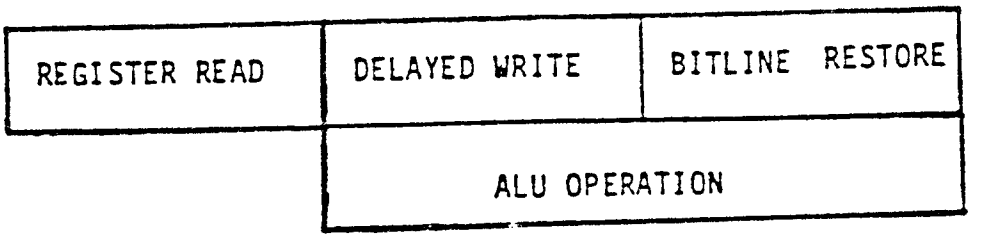


Figure 4: Timing of Delayed Write Scheme

In order to reduce the datapath cycle from four phases to three, the RISC II increased the level of pipelining and incorporated a *delayed write* scheme [8]. In effect, writing is delayed to overlap the ALU computation of the following instruction. This added level of pipelining is helpful as it allows greater time for interrupts to be detected without destroying the contents of the register file. Also, if the ALU delay is significant, it may overlap *both* the register write and bitline restore phases (Figure 4).

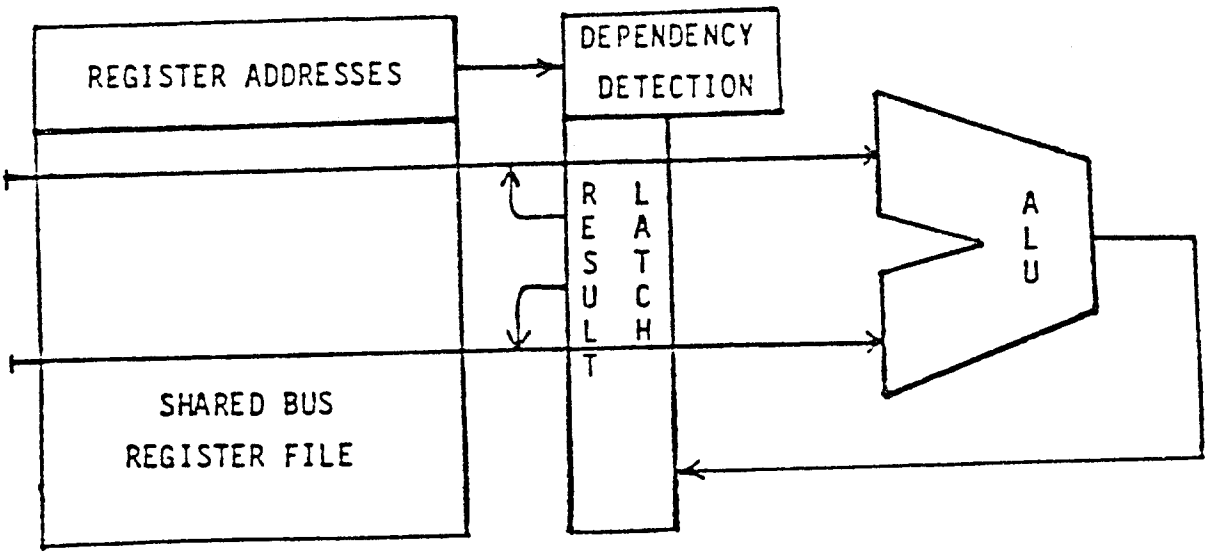
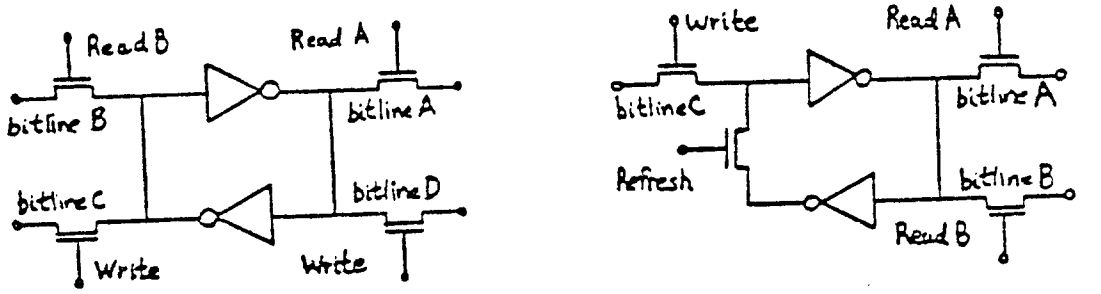


Figure 5: Delayed Write with Internal Forwarding

Some performance degradation might result from this scheme due to *data dependencies*. The result of a computation is not available in the register file for the read phase of the following instruction. This is a consequence of this pipelined implementation with the delayed write. The problem was solved in the case of RISC II by detecting data dependencies and forwarding the data through a temporary register to the ALU or shifter. This *internal forwarding*, or *chaining* technique allows the data in this register to *override* the result of the register file read (Figure 5). This technique is transparent to the programmer or compiler writer. Such an approach is routinely used to increase performance of highly-pipelined computers, such as the CRAY I.

Dedicated Bitline Register Files



(a) Static Dedicated Bitline Cell

(b) Pseudostatic Dedicated Bitline Cell

Figure 6: Dedicated Bitline Cells

As previously mentioned, the dedicated bitline design utilizes separate bitlines for reading and writing. Implicitly, this requires separate wordlines as well to guarantee independence of read and write operations (see Figure 6). This structure supports a higher level of concurrency and therefore may be desirable for a high-speed datapath. Restoring of the bitline may overlap the writing of the

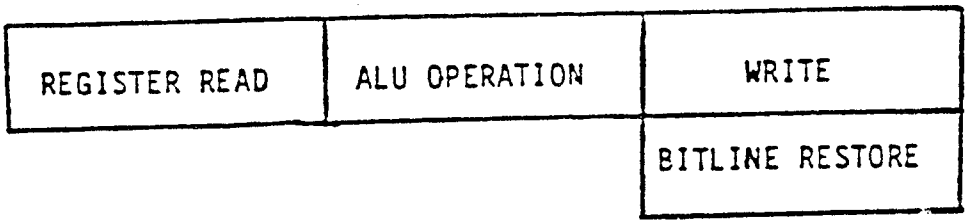


Figure 7: Timing of Dedicated Bitline Datapath

cell since it uses separate control and data lines. This makes possible the three-phase timing of Figure 7. Such an approach has been used in the RISC I [8] and the Matsushita MN1613 [9]. This scheme, however, will be slower than the previous approach (shared bitline with delayed writing) if the ALU delay is greater than that of the bitline restore. This, in conjunction with the cell area difference, makes the three-phase dedicated bitline scheme discussed here undesirable for large register files.

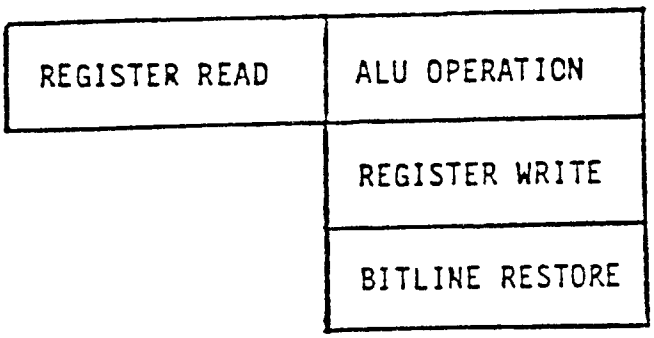


Figure 8: Timing of Dedicated Bitline with Delayed Write

In order to increase the concurrency, the delayed-write scheme may be used. Timing of the ALU operation, register write, and the bitline restore all overlap (Figure 8). Internal forwarding logic is necessary to eliminate data dependency problems as before. Forwarding is performed in parallel with the register write.

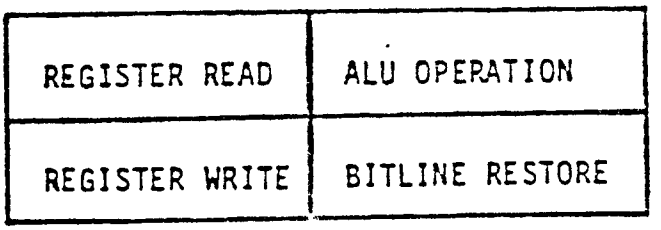


Figure 9: Overlapped Read/Write Scheme Timing

Alternatively, the read and write operations may be overlapped, as shown in Figure 9. Dependency detection logic is again required, and internal forwarding is performed in parallel with the register write.

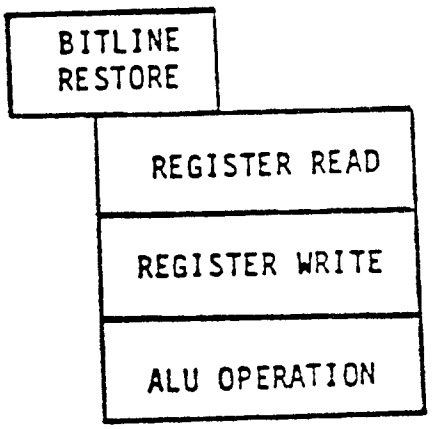


Figure 10: Delayed Write with Overlapped Read Timing

For even higher performance, two sets of data dependency logic are required. The first forwards the result of an ALU or shift operation to the data read register of the following instruction. The second forwards this data, as it is being written into the register file, to the read register for the instruction after that. This allows the greatest concurrency, as shown in Figure 10.

In order to combine the register read and bitline restore in a single phase, the restore must be initiated early enough during the read phase so that it overlaps the addressing delay. At the time the read wordlines are driven to a logic

"1", the bitlines must be precharged above the bit cell logic threshold in order to eliminate writing into the cell. Precharge may continue, overlapping wordline delay so that adequate noise margins are maintained. Alternatively, current sensing may be used, in which case the bitline voltage remains constant. This technique has been utilized in MOS ROMs [10].

System throughput for this single-phase timing scheme may be quadruple that of the original four-phase sequential example at the beginning of this chapter. This performance increase is achieved by maximizing module usage in each phase, in tune with effective chip resource utilization.

The treatment of datapath timing and register file organization has been very simplistic in this chapter. Since the bit cells must be designed uniquely for each timing scheme, their area will vary. This has a varying impact on chip resources and on cycle delay time. A more detailed analysis is thus required for the selection of an optimal register cell and timing scheme. This will be discussed in more detail in Chapter 6.

References

[1] D.A. Patterson, C.H. Séquin: "RISC I: A Reduced Instruction VLSI Computer," Proceedings of the 8th Symposium on Computer Architecture, ACM SIGARCH CAN, pp. 443-457, May 1981.

[2] J. Hennessy, N. Jouppi, F. Baskett, J. Gill: "MIPS: A VLSI Processor Architecture," VLSI Systems and Computations, Carnegie-Mellon University Conference, Computer Science Press, October 1981.

[3] G. Radin: "The 801 Minicomputer," Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, ACM SIGARCH CAN, pp. 39-47, March 1982.

[4] J. D. Wright: "Relation of Microcode to Future Machine Design," COMPCON Digest of Papers, pp. 104-106, March 1983.

[5] R.W. Sherburne, M.G.H. Katevenis, D.A. Patterson, C.H. Séquin: "Datapath Design for RISC," Proceedings of the Conference on Advanced Research in

VLSI, Massachusetts Institute of Technology, pp. 53-62, January 1982.

[6] C.A. Mead, L.A. Conway: *Introduction to VLSI Systems*, Addison Wesley Publishing Co., 1980.

[7] J. Beyers, L. Dohse, J. Fucetola, R. Kochis, C. Lob, G. Taylor, E. Zeller: "A 32-Bit VLSI CPU Chip," IEEE Journal of Solid-State Circuits, vol. sc-16, no. 5, pp. 537-541, October 1981.

[8] D.T. Fitzpatrick, J.K. Foderaro, M.G.H. Katevenis, H.A. Landman, D.A. Patterson, J.B. Peek, Z. Peshkess, C.H. Séquin, R.W. Sherburne, K.S. VanDyke: "VLSI Implementations of a Reduced Instruction Set Computer," VLSI Systems and Computations, Carnegie-Mellon University Conference, Computer Science Press, pp. 327-336, October 1981. Also in: "A RISCy Approach to VLSI," VLSI Design, vol. II, no. 4, pp. 14-20, 4th qtr. 1981, and Computer Architecture News (ACM SIGARCH), vol. 10, no. 1, pp. 28-32, March 1982.

[9] H. Kadota, S. Ozawa, K. Kawakami: "A New Register File Structure for the High-Speed Microprocessor," IEEE Journal of Solid State Circuits, vol. sc-17, no.5, pp. 892-897, October 1982.

[10] J. Wong, P. Siu, M. Ebel: "A 45ns Fully-Static 16K MOS ROM," Proceedings of the International Solid-State Circuits Conference, pp. 150-151, February 1981.

CHAPTER 5:

ALU DESIGN TRADEOFFS

Traditionally, evaluation of different adder schemes has been carried out with the assumption of a fixed gate delay. Such a straightforward comparison is permitted by low levels of integration, using SSI parts. These parts are designed to accommodate a wide range of capacitance loading due to off-chip wiring. As a result, delay exhibits little dependence on the loading capacitance typically encountered [1]. The designer calculates circuit delay by simply determining the number of gates in the critical path.

The custom nature of VLSI, on the other hand, gives the designer more freedom to optimize performance. Dynamic logic and bootstrapping techniques can be used to increase performance. Under this variety of approaches, gate delays can no longer be considered constant. Comparison of adder performance based on the fixed delay model is inadequate for VLSI implementation.

This chapter will begin with a review of adder design strategies. Initially a gate-level view will be used in order to simplify understanding. It is also directly applicable to fixed gate delay analysis. This will be followed by a discussion of design in NMOS using dynamic logic and bootstrapping. Finally, different carry schemes will be evaluated for both the fixed delay and NMOS implementations.

Adder Design at the Gate Level

An example of a single-bit cell of a full adder is shown in Figure 1. Three delays exist: input translation, carry calculation, and sum generation. The

translation and sum delays are constant; they each consist of a single gate delay. The carry delay, however, is cumulative since its calculation is dependent on the result from the previous bit cell. The carry output of the most significant bit is thus dependent on all the previous stages. Overall carry delay will vary with the method used for its calculation, as well as with the number of bits N. For this reason we will focus on the circuitry that calculates the carry.

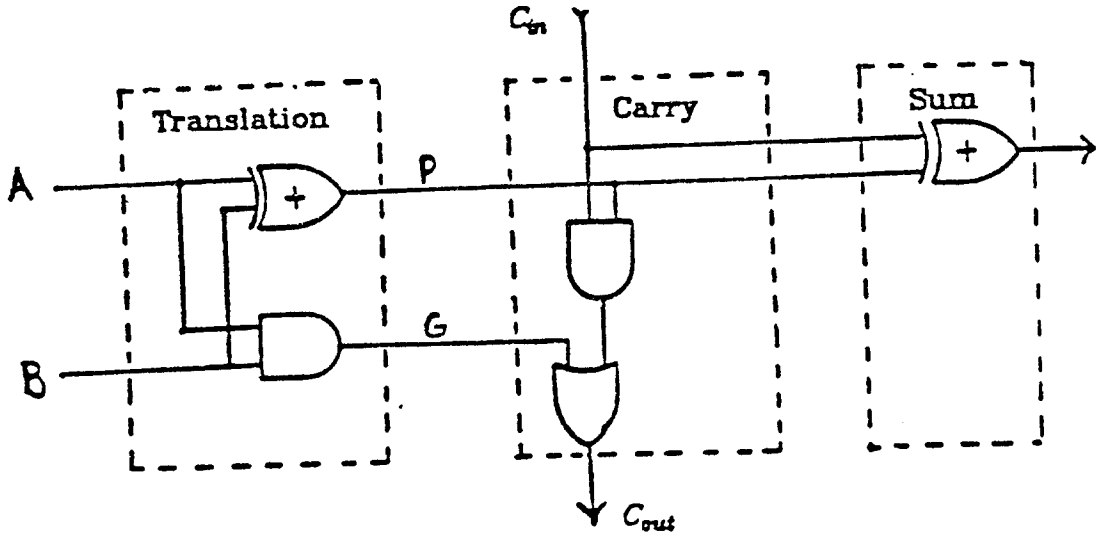


Figure 1: Full Adder Cell

Ripple Carry

The simplest adder scheme utilizes a ripple carry as shown in Figures 1 and 2. For an N-bit adder, the carry propagates, or ripples, across N stages. Each stage consists of 2 gate delays, so the total carry delay for an N-bit adder is 2N. Advantages of this design include minimal gate count, as well as regularity and short wire length for implementation in VLSI.

The ripple carry approach is used for small word sizes or in applications where speed is not critical. An 8-bit ripple adder was chosen for the Intel 8080 8-bit microprocessor because the regular, compact layout had less parasitics and was actually faster than a lookahead approach [2]. The Motorola 68000 used

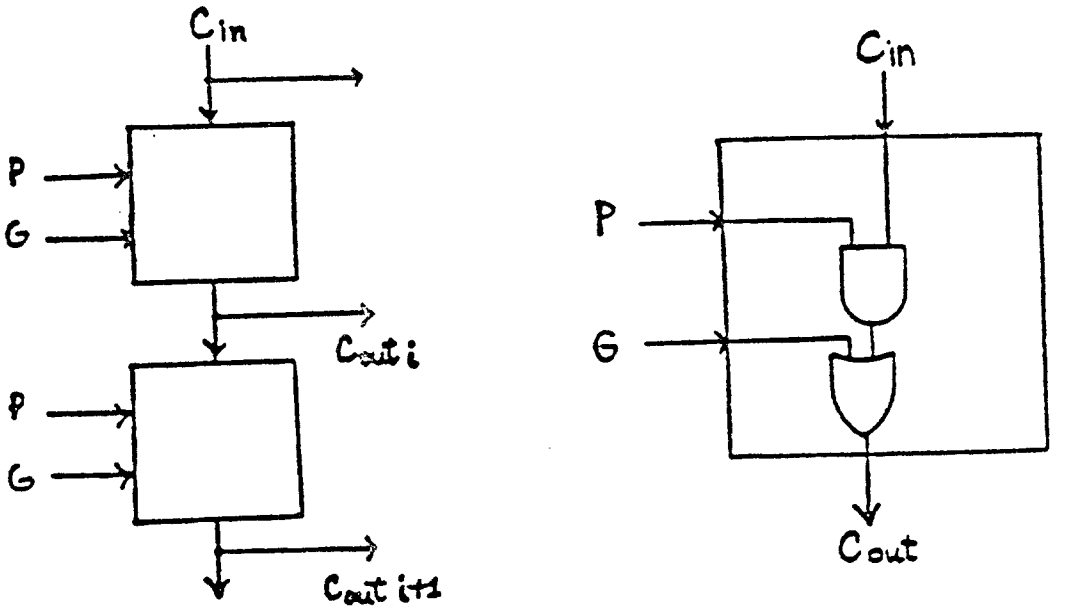


Figure 2: Ripple Adder Scheme (2 bits shown)

a 16-bit ripple adder because it was found to be faster than a lookahead adder with the same amount of power dissipation [3]. Ripple adders were also used for the 16-bit Caltech OM-2 [4] and the 32-bit RISC II [5] implementations, mainly because of their small chip area and short layout time.

Methods of reducing ripple carry delay are presented in [2]. Using an increased fan-in of 4, the delay can be reduced to an average of 4 gate levels for each 3 bit group by propagating multiple intermediate carry terms between each stage. However, many more gates and wires are required, and the overall structure is much less regular than that of Figure 2. For these reasons, such an approach will not be investigated further.

Carry-Select

A carry-select (or conditional carry) adder is shown in Figure 3. The carry output of the first M-bit ripple adder is used to select the proper output of the next pair of ripple adders, each with complementary carry inputs. All ripple adders operate at the same time, so overall delay consists of an M-bit ripple add

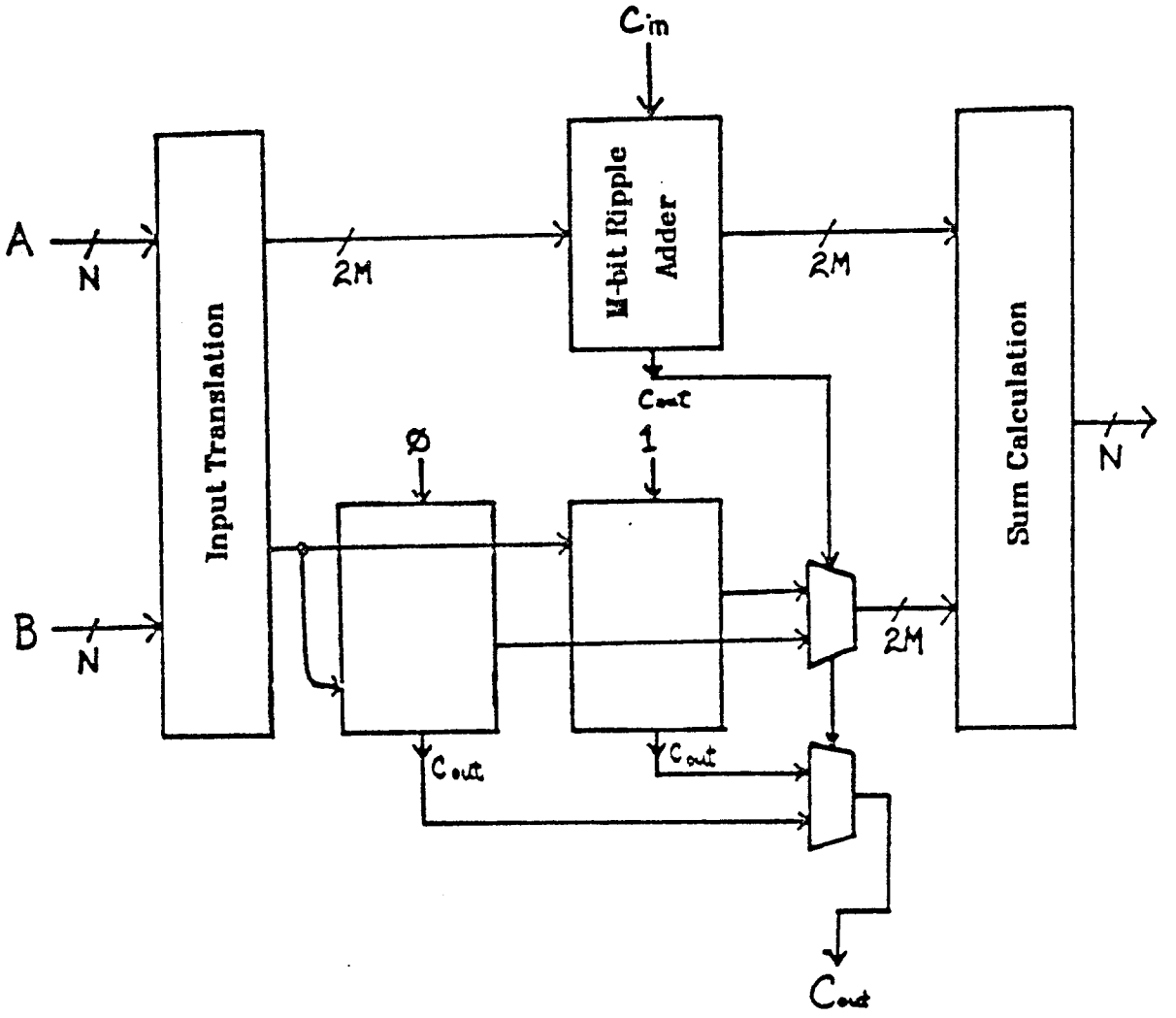


Figure 3: Carry-Select Adder

followed by a cascade of multiplexors. Carry delay goes as $(M + \frac{N}{M})$ so there exists an optimal M yielding a lower bound in time. This choice of M for highest performance is $\sqrt{N}$, assuming bit delay is equal to multiplex delay.

The carry select adder is fully modular. Layout may be done with a few basic cells. No irregular wiring is required among the modules. This is important in reducing the design time, layout area, and the probability of design errors in the random wiring. Gate count (and therefore power and area) for carry calculation is nearly twice that of the ripple carry approach.

Carry Lookahead

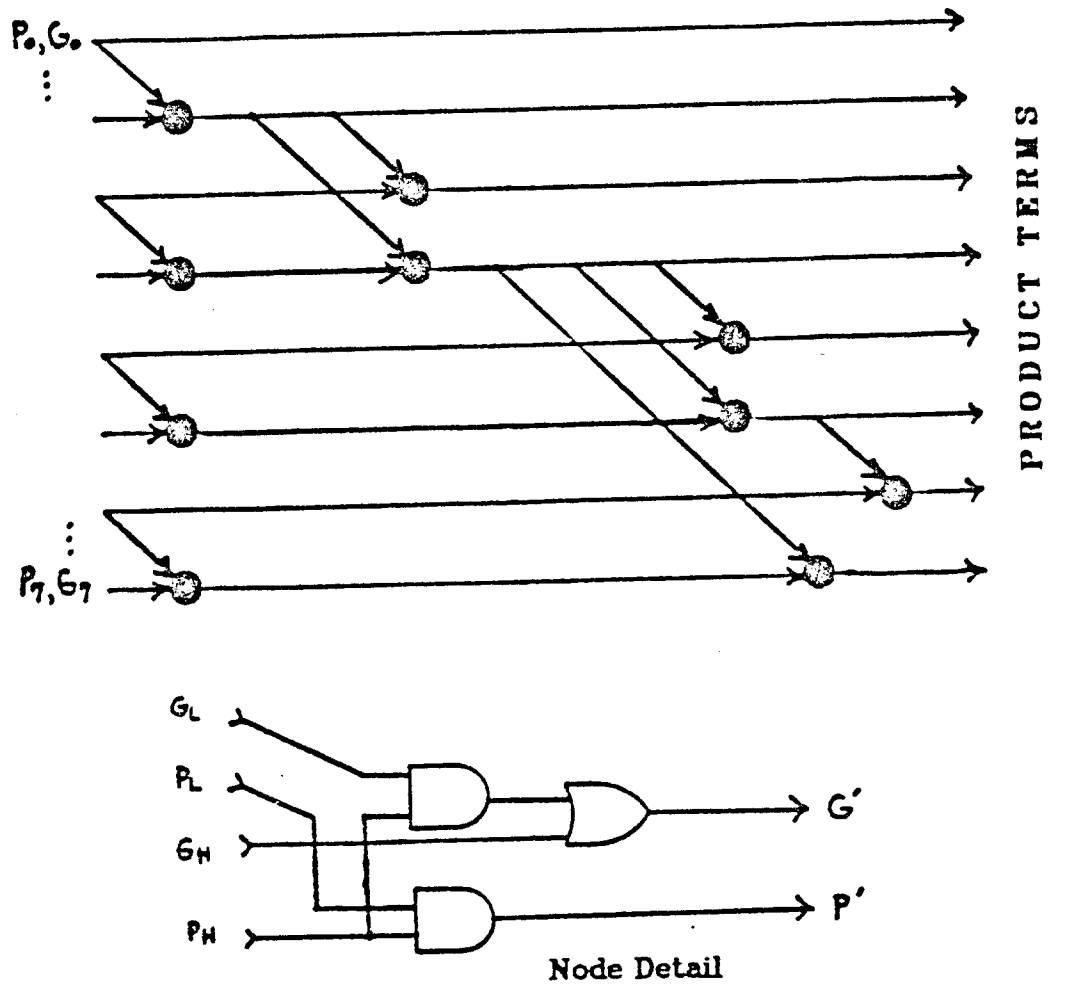


Figure 4: Parallel Adder (8 bits shown)

A full-lookahead (or parallel) adder performs calculation of all P and G product terms. Figure 4 details the organization of an 8-bit parallel adder as well as the design of the individual circuit modules. The overall delay for an N-bit parallel adder goes as $\log_f N$ assuming a gate fan-in of f. This $\log_f$ behavior is important in reducing carry delay for large adders. Parallel adders have been implemented in the HP Focus [6], MIPS [7] and Xerox Dragon [8] 32-bit microprocessors.

Such an approach requires nearly four times the gate count of the ripple

scheme for carry calculation. The associated increase in power consumption and irregular wiring makes this design much more costly for VLSI implementation than the previous ones. A frequently used compromise is to do a *partial* carry lookahead, trading off gate and power requirements against carry delay. In this approach, lookahead is performed in M-bit groups, the results of which are input to M-bit ripple adders as in the carry-select scheme. Results of this partial lookahead are partial products which are input to ripple carry adders. The Bellmac-32 [9] and the RISC I [10] 32-bit implementations, as well as a prototype design by Siemens [11] utilize partial carry lookahead adders with M=4. Another example is the partial carry lookahead adder using MSI parts shown in TI's TTL Data Book [1]. A regular layout for lookahead computation is discussed in [12].

Adder Designs in NMOS

So far our model assumed fixed delay and power per gate. In NMOS the high transistor impedance and the variety of static and dynamic circuit implementations reduces the validity of such an analysis. Dynamic logic requires no static power consumption. Operation is performed in two-phases: first the output nodes are dynamically precharged, then they are selectively discharged. This selective discharge of precharged nodes without static pullups requires no ratioing of the transistors as for static logic. This allows transistors driving critical paths to be freely increased in size, to attain desired speed.

The equivalent gate model for a precharged ripple carry chain is shown in Figure 5(a). This is similar to the ripple logic in Figure 2. The NMOS dynamic ripple circuitry is given in Figure 5(b). During φ_1 the output logic levels are precharged. At this time the carry input is not allowed to discharge the chain. On φ_2 the carry input is enabled to propagate along the ripple stages by selective discharge. The G terms are also enabled to discharge the carry line.

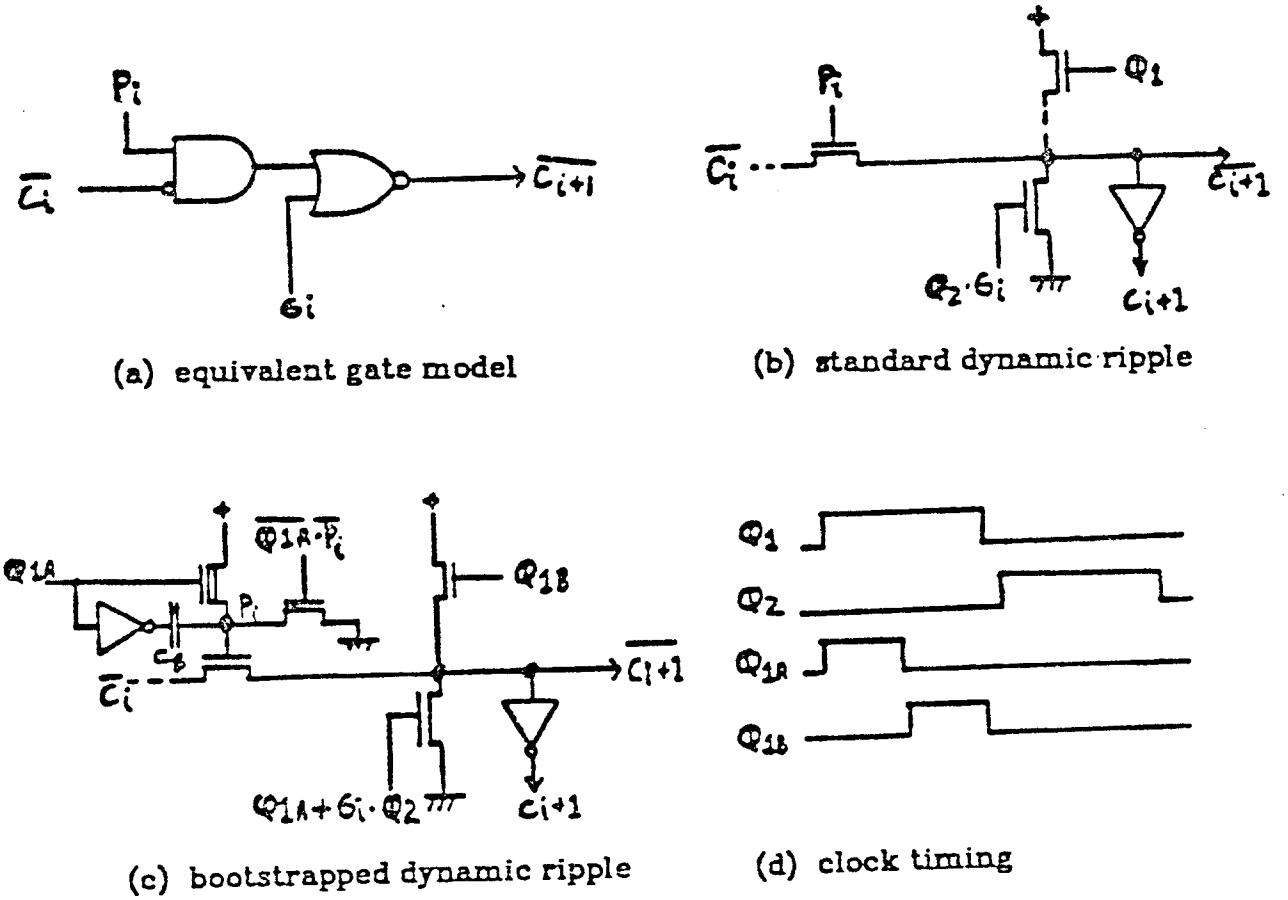


Figure 5: Dynamic Ripple Carry Circuitry

Sensing of the carry bit is performed by a static inverter; its output drives the sum logic.

A modified ripple stage is shown in Figure 5(c). It utilizes bootstrap capacitor C_B to increase conductance of the pass transistor P_i. The first clock phase φ_1 is divided into two sub-phases φ_{1a} and φ_{1b} as shown in 5(d). During the first sub-phase, the entire carry line is discharged to ground. At this time the bootstrap capacitor C_B is precharged. During φ_{1b} the bootstrapping takes place, raising the pass transistor gate to 7 or 8 volts. This results in charge being coupled onto the carry line. Since it was initially discharged, this coupling does not cause it to overshoot the supply voltage. Otherwise, increased delay would result in order to discharge the carry line to a logic "0". By the end of

$\varphi 1b$ the carry line is precharged; operation then proceeds with φ_2 as for the previous example. Use of this bootstrapping technique allows for higher performance of a long ripple chain.

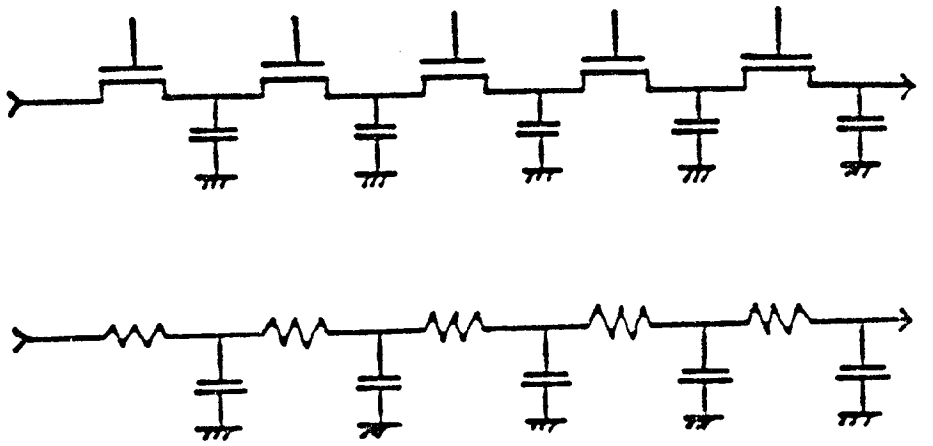


Figure 6: Transmission Line Equivalent of Carry Chain

A long dynamic carry chain will have many pass transistors in series. As a result, carry propagation across N bits will be quite slow. Behavior of such a carry chain is equivalent to that of a transmission line (Figure 6). Assuming that the pass transistors are of minimum channel length and large enough to dominate carry line parasitics, the transmission line delay becomes independent of transistor width. This is because any change in channel conductance (proportional to width) is accompanied by a proportional change in gate capacitance. The overall RC product remains constant for a particular technology.

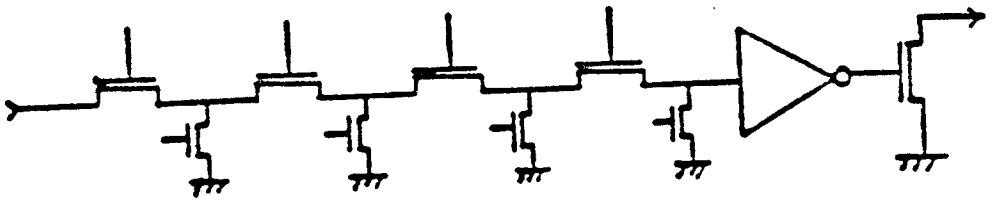


Figure 7: Carry Chain Buffering (precharge devices not shown)

One way of overcoming this long carry chain delay is to periodically buffer the carry line (Figure 7). This is equivalent to the use of repeaters on lossy transmission lines. Overall delay for long chains then becomes a linear function of carry chain length, rather than a square function as would be the case without the buffers.

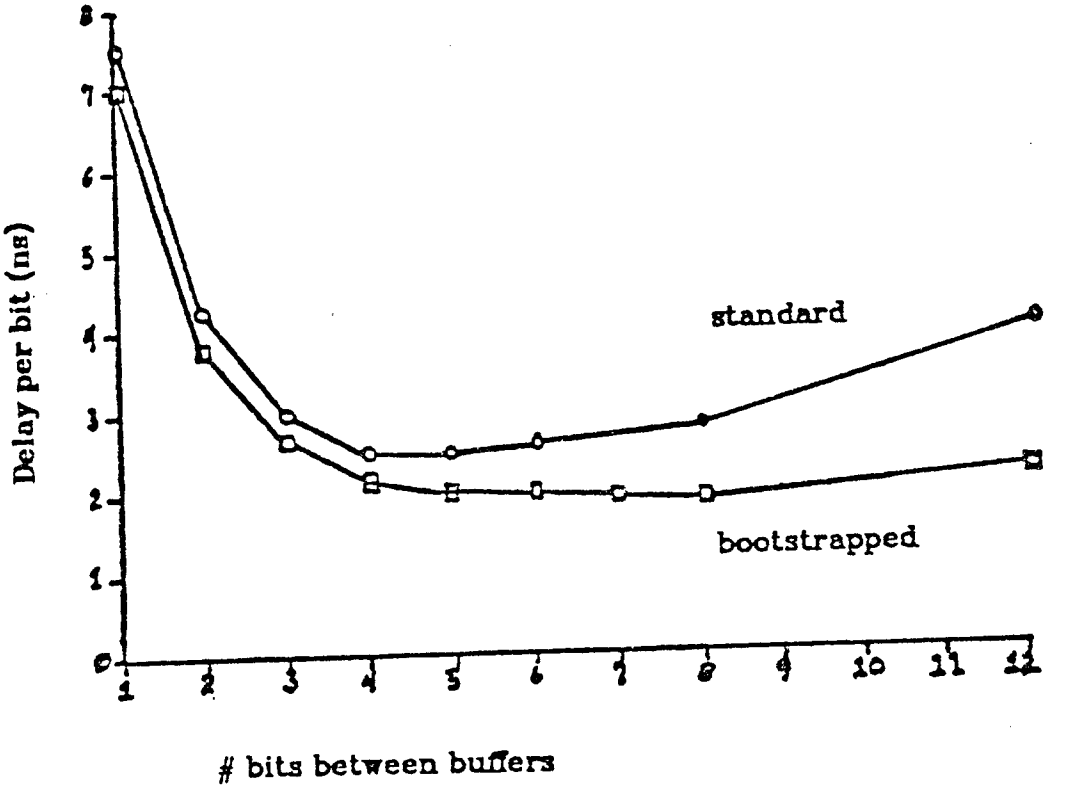


Figure 8: Buffered Carry Chain Optimization

Results of an analysis to determine optimal length of chain sections between buffers are shown in Figure 8. Data are based on SPICE simulation results using the device parameters in Table I. The ratio of parasitic to gate capacitance was 1:4. For the standard ripple carry design, four bits are optimal. This value was implemented in the Caltech OM-2 and the RISC I. The bootstrapped approach yields higher performance through eight bits, at which point only half the number of buffers are required.

λ	2.0	μm	Capacitances:		
Transistors:			metal	0.14	fF/λ^2
V_{ETO}	0.9	V	diffusion bulk	0.3	fF/λ^2
V_{DTO}	-3.2	V	diffusion side-wall	0.3	fF/λ
V_{DD}	5.0	V	poly over field	0.2	fF/λ^2
V_{BB}	-2.0	V	gate	1.6	fF/λ^2
γ	0.75	$V^{1/2}$	gate-src overlap	0.5	fF/λ
k'	20.7	$\mu A/V^2$	Resistances:		
μ_o	600	$cm^2/V \cdot sec$	polysilicon	50	$\Omega/\blacksquare$
min. electr. channel L	4.0	μm	diffusion	10	$\Omega/\blacksquare$

TABLE I: NMOS Device Parameters (worst-case speed)

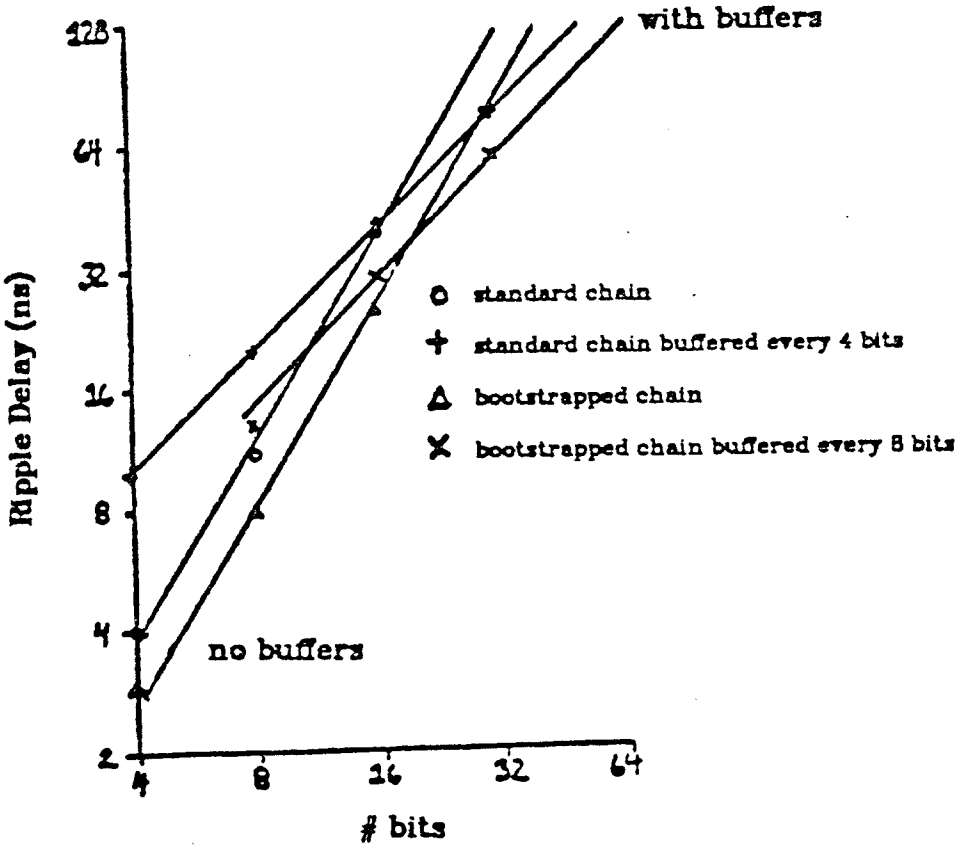


Figure 9: Comparison of Ripple Delays

A comparison of ripple carry delays with and without buffering is presented in Figure 9. Because of the added delay incurred by static buffers, the unbuffered designs are fastest for small adders. In the graph, a 16-bit adder is

shown to be fastest with no buffering at all. Larger chains, though, may benefit significantly from buffering; it allows delay dependence on adder size to be reduced from N^2 to N as seen by the reduction in slope on the graph. The effect of increased parasitics (versus the 1:4 ratio mentioned above) is to make the overall delay increase much more quickly with chain length, so that buffering is more attractive for smaller adders. Further analysis is necessary for optimization in such a case.

The performance of the carry-select approach using ripple adders can also be evaluated using the ripple data. Each multiplex operation requires a single buffered stage and precharge logic, for which delays may be obtained from Figure 8.

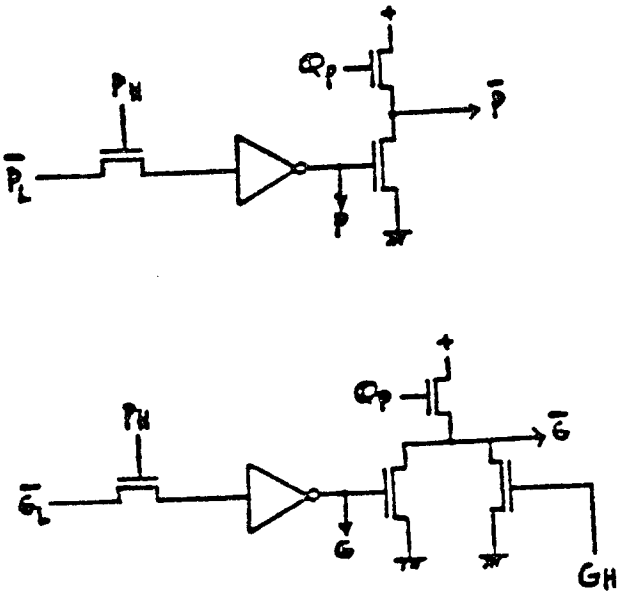


Figure 10: Parallel Adder Logic Stages in NMOS

The parallel adder is implemented in alternating precharged and static logic stages (Figure 10). The logic functions shown are identical to those in Figure 4. Operation is again two phase: first, precharging of the dynamic gates, then a selective discharge, driven by the input terms. Because both polarities of

the intermediate P and G product terms are required, fully dynamic logic chains are not appropriate. Delay of a series of parallel adder stages is similar to that of ripple carry stages which are buffered every bit, as both consist of a static and a dynamic gate. The result of the P and G product term calculation for all bits must then be processed in a final stage to include the carry input. Such parallel adder logic has been implemented in the MIPS and Xerox Dragon 32-bit microprocessors.

Evaluation of Carry Schemes

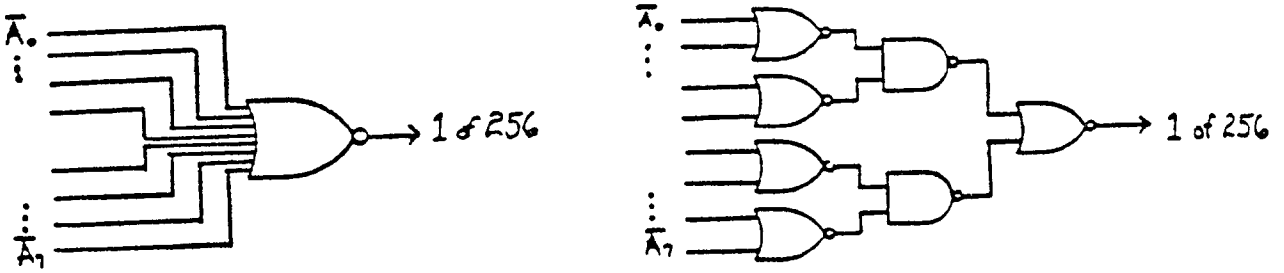
Some adder strategies for representative microprocessors are summarized in Table II. It is not clear which design is best for a given datapath size, although small adders all are of the ripple type. First a comparison using the fixed gate delay model is performed, as a starting point. This is compared to results of NMOS implementation using dynamic logic and bootstrapping techniques where applicable. Although speed will be the main focus of analysis, implications on chip area and power will also be discussed.

INTEL 8080	8 Bit Ripple
MOTOROLA 68000	16 Bit Ripple
CALTECH OM-2	16 Bit Ripple
UCB RISC II	32 Bit Ripple
BELLMAC-32	32 Bit Partial Lookahead
UCB RISC I	32 Bit Partial Lookahead
HP FOCUS	32 Bit Parallel
STANFORD MIPS	32 Bit Parallel
XEROX DRAGON	32 Bit Parallel

TABLE II: Microprocessor ALU's

The fixed gate delay model has been applied to large adders in order to evaluate performance asymptotically. This is not appropriate for comparing approaches for microprocessors with adders of only 32 bits or less. This is too small for an asymptotic analysis because the constant components of delay cannot be ignored. The data presented will consider absolute delays for typical ALU sizes.

Earlier discussion of the parallel adder considered arbitrary gate fan-in. In TTL, there is little penalty for increased fan-in: most of the delay is attributed to the output driver. In VLSI, however, increased fan-in has its cost. For short paths, the gate delay is highly dependent on transistor parasitics at the output node. Increasing the number of inputs to a NOR gate adds more drain diffusion and overlap parasitics to the output node. Because fixed device ratios must be maintained for adequate noise margins in static logic, these intrinsic device parasitics cannot be eliminated from consideration.



(a) 8-input NOR gate **(b) equivalent using 2-input gates**

Figure 11: Realizations of 8-Input NOR Function

Delays of an 8-input, static NOR gate were compared with an equivalent realization composed of 2-input gates (Figure 11). The results, in Table III, are based on the device parameters given in Table I. Delay was measured as the time required for the output to reach 3V and 2V for logic "1" and "0", respec-

Static NOR Fan-In (K=4)	No Wire Delay		With Wire Delay	
	t_{dLH}	t_{dHL}	t_{dLH}	t_{dHL}
8 (Figure 11(a))	15ns	3ns	21ns	4ns
2 (Figure 11(b))	18ns	12ns	23ns	16ns

TABLE III: Gate Fan-In Comparison

tively. Delay to logic "0" (t_{dHL}) is significant for the 2-input version. However, the interesting delay is that to logic "1" (t_{dLH}): this is the limiting delay. Neglecting wire loading, the smaller fan-in incurs 20% delay penalty.

With wire delay, penalty for the 2-input constraint reduces to less than 10%. Wire loading was calculated based on a 42 λ spacing between NOR inputs; this is the datapath pitch between each bit slice (as determined by the register file) of the RISC II microprocessor [13]. A minimum-width metal line was assumed to connect the drains of the multiple NOR input transistors.

Since these resulting circuit delays are so similar, we will restrict ourselves to circuits using a fan-in of 2 in the subsequent comparison of various carry implementations. This simplifies the analysis by reducing the number of variables to be considered. Coincidently, the parallel adders in the MIPS and the Xerox Dragon micrprocessors both use gates with a fan-in of 2 for the carry computation.

Table IV summarizes the delay and gate count for the fixed delay model carry schemes. Figure 12(a) depicts adder delay as a function of N, while 12(b) gives gate count as a first approximation of area and power requirements. Performance for the more complicated schemes is seen to improve considerably

Carry Scheme	Gate Delays for Carry	# Gates for Carry
Full Ripple	2N	2N
Carry-Select	$4\sqrt{N}$ - 2	4N - 2
Parallel	$4\log_2 N$ - 2	8N - $3\log_2 N$ - 8

TABLE IV: Carry Computation: Fixed Gate Delay Model

(fan-in = 2, neglecting wiring delays)

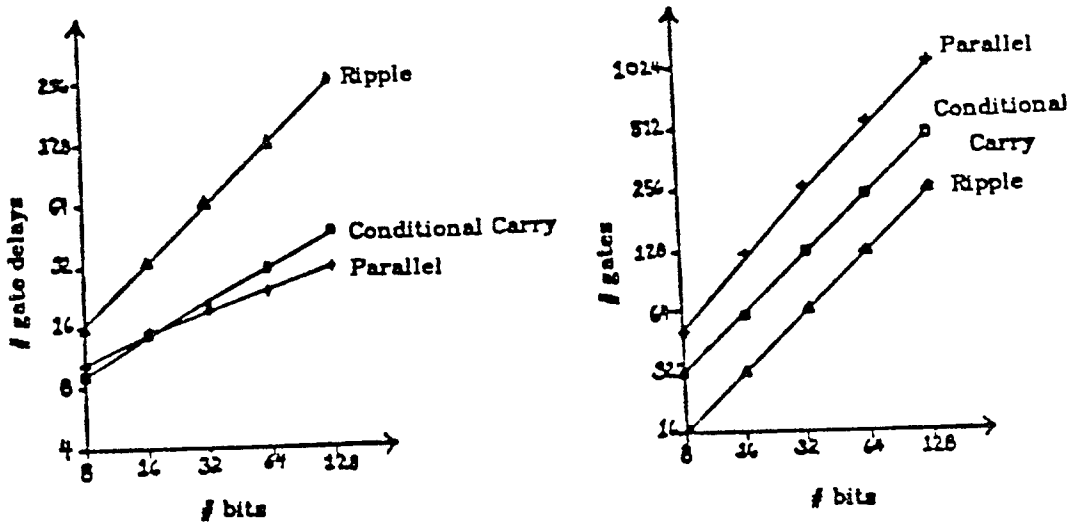


Figure 12: Delay and Gate Count Comparison

over that for the ripple scheme, though at the cost of additional area and power requirements. The carry-select design is fastest from 8 to 16 bits; it requires nearly twice the number of gates as the ripple version. The parallel design is

fastest for 32 bits and beyond, though at a gate and power cost approaching four times that of the ripple carry scheme.

Adder Scheme	Optimal Carry Delay	Inverter Count (Power)
Full Ripple	$\frac{N}{M}\,\tau_{chain}$	$\frac{N}{M}$
Carry-Select	$2\sqrt{\frac{N}{M}\,\tau_{buffer}\tau_{chain}} - \tau_{buffer}$	$\approx 2\frac{N}{M}$
Parallel	$(2\log_2 N - 1)\,\tau_{buffer}$	$5N - 2\log_2 N - 5$

TABLE V: MOS Implementation Comparison

(where ripple chain length $M = \sqrt{\frac{\tau_{buffer}}{\tau_{bit}}}$)

Results of the NMOS delay analysis using optimal size buffered ripple carry chains are summarized in Table V. These results represent optimal solutions; actual values will differ slightly for the carry-select adder in order to accommodate the granularity of optimal chain length. Data for typical adder sizes are given in Figure 13. Results are based upon the optimally buffered, bootstrapped carry chain length of 8 bits with 15ns of delay, and a single buffer stage delay of 7ns. Using these parameters, the ripple adder is best for 8 bits and also attractive for 16 bits, due to its reduced area and power requirements. The carry-select is fastest through 128 bits and requires much fewer buffers than the parallel design. In fact, for a large adder the parallel approach requires nearly 20 times as many buffers as the carry-select scheme. This high number of buffers can significantly impact power resources on the chip. Even in a CMOS implementation using no static power, the additional buffers are costly in terms of silicon area. These results differ markedly from those obtained using the

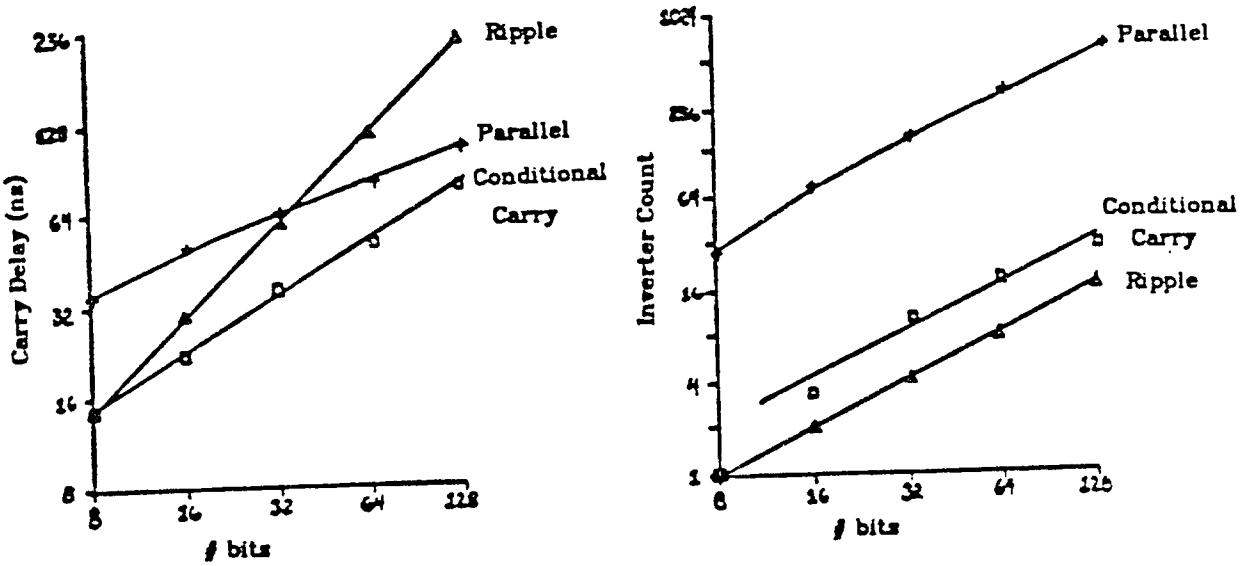


Figure 13: NMOS Carry Logic Comparison

fixed gate delay model.

A more accurate comparison must include the effect of wiring delays. The ripple adder has the shortest paths and would be least affected by such delays. The parallel adder, in contrast, has wire lengths which increase with N; the longest connections must span half the width of the ALU. Inclusion of such delays could only lessen the gains of increased parallelism. This further reduces the applicability of the fixed gate delay analysis and makes the parallel adder unattractive for VLSI implementations.

References

[1] Engineering Staff of Texas Instruments, Inc., *The TTL Data Book for Design Engineers*, Application Data, SN54/74182.

[2] H.C. Lai and S. Muroga: "Minimum Parallel Binary Adders with NOR (NAND) Gates," IEEE Transactions on Computers, vol. c-28, no. 9, pp. 648-659, September 1979.

[3] Les Crudele, Motorola, private communication, January 1981.

[4] C.A. Mead and L.A. Conway: *Introduction to VLSI Systems*, Addison Wesley, (Menlo Park, 1980).

[5] M.G.H. Katevenis, R.W. Sherburne, D.A. Patterson and C.H. Séquin: "The RISC II Micro-Architecture," Proceedings of the IFIP TC10/WG10.5 International Conference on Very Large Scale Integration (VLSI '83), Trondheim, Norway, pp. 349-359, August 1983.

[6] J. Beyers, L. Dohse, J. Fucetola, R. Kochis, C. Lob, G. Taylor, E. Zeller: "A 32-Bit VLSI CPU Chip," IEEE Journal of Solid-State Circuits, vol. sc-16, no. 5, pp. 537-542, October 1981.

[7] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross: "Design of a High Performance VLSI Processor," Proceedings of the Third Caltech Conference on VLSI, Computer Science Press, pp. 33-54, March 1983.

[8] C. Thacker, "The Dragon Project," Lecture given at the Computer Systems Seminar, University of California, Berkeley, October 21, 1982.

[9] B. Murphy, L. Thomas, A. MacRae: "Twin Tubs, Domino Logic, CAD Speed Up 32-Bit Processor," Electronics, vol. 54, no. 20, pp. 106-111, October 6, 1981.

[10] D. Fitzpatrick, J. Foderaro, M. Katevenis, H. Landman, D. Patterson, J. Peek, Z. Peshkess, C. Séquin, R. Sherburne, K. VanDyke: "VLSI Implementations of a Reduced Instruction Set Computer," VLSI Systems and Computations, Carnegie-Mellon University Conference, Computer Science Press, pp. 327-336, October 1981. Also in: "A RISCy Approach to VLSI," VLSI Design, vol. II, no. 4, pp. 14-20, 4th qtr. 1981, and Computer Architecture News (ACM SIGARCH), vol. 10, no. 1, pp. 28-32, March 1982.

[11] M. Pomper, W. Beifuss, K. Horninger, W. Kaschte, "A 32-Bit Execution Unit in an Advanced NMOS Technology," IEEE Journal of Solid-State Circuits, vol. sc-17, no. 3, pp. 533-538, June 1982.

[12] R.P. Brent and H.T. Kung, "A Regular Layout for Parallel Adders," IEEE Transactions on Computers, vol. c-31, no.3, pp. 260-264, March 1982.

[13] R.W. Sherburne, M.G.H Katevenis, D.A. Patterson, C.H. Séquin, "Datapath Design for RISC," Proceedings of the Conference on Advanced Research in VLSI, Massachusetts Institute of Technology, pp. 53-62, January 1982.

CHAPTER 6:

PROCESSOR PERFORMANCE

Previous chapters have discussed individual areas of design tradeoffs, one by one. In reality, there is much more interaction among these areas than has been suggested so far. In order to perform the analysis necessary to account for this interaction, an understanding of the entire processor design and its associated tradeoffs is required. Present 32-bit microprocessors include up to several hundred thousand transistors. Familiarization with every aspect of the design then can be a monumental task.

However, it is not necessary to consider processor behavior all the way down to the circuit level. Use of higher levels of abstraction allow some tradeoffs to be evaluated independently of circuit details or of the fabrication technology. This yields good results without overwhelming the designer and without burdening him with unnecessary detail. In some other cases, however, the strengths and weaknesses of a particular implementation technology will have an impact even at the architectural level. For example, the cost of implementing a particular function on the chip may vary so much among different processing technologies that it may become prohibitive in some instances.

Limited chip area and power resources make processor design optimization

a real challenge. A large local memory will reduce the amount of data I/O required during execution at the cost of chip resources otherwise available for other functions. Increasing the datapath wordsize has a similar effect. Optimal local memory capacity, discussed in Chapter 3, may be too costly to implement. Other strategies for performance improvement, such as increased swap support or processor pipelining, must then be considered.

Increased pipelining boosts processor throughput, as discussed in Chapters 2 and 4. A side effect of pipelining in the register file is higher memory area cost, due to the increase in the number of wordlines and bitlines necessary to support the concurrent operations. As a result, less local memory may be implemented in a given amount of chip area. This reduces the gain offered by pipelining in the first place. For fixed local memory capacity, the highly pipelined implementation will have slower register operations. Increased pipelining can even degrade system performance.

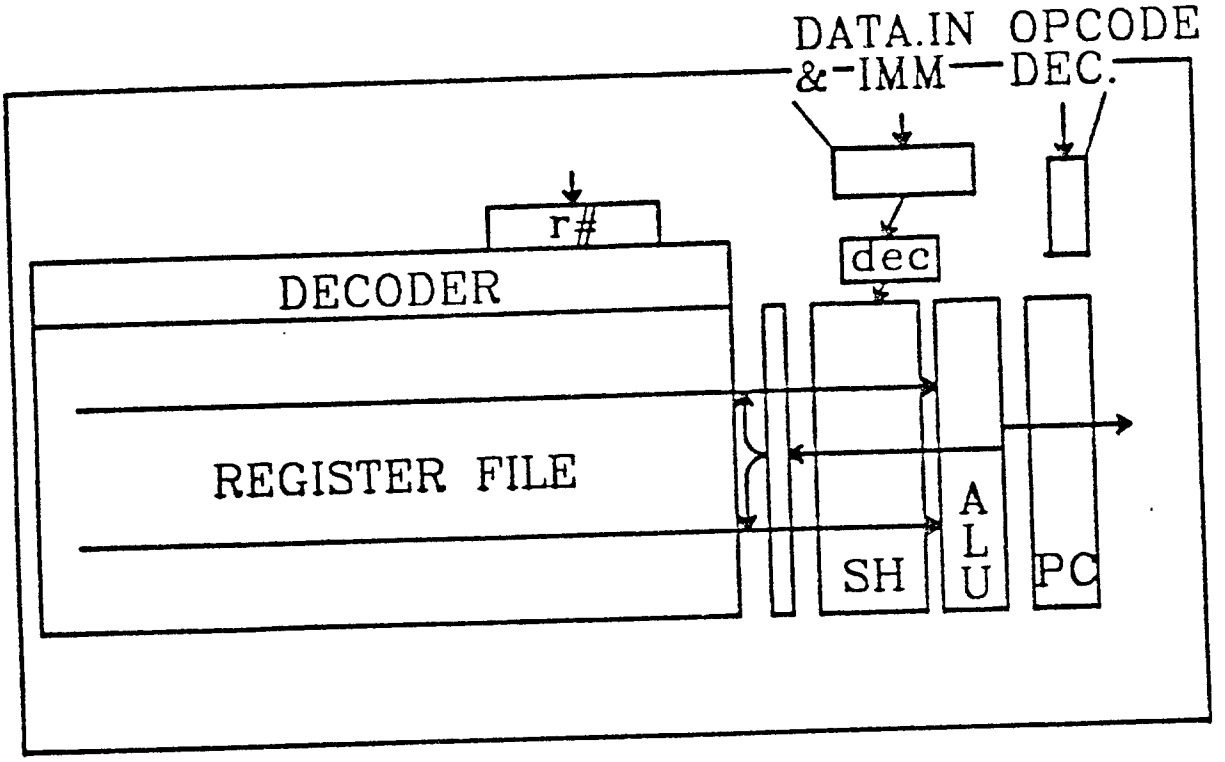


Figure 1: Chip Area Allocation in RISC II

Figure 1 shows area allocation for the RISC II microprocessor [1]. The register file occupies the majority of the datapath area; it also consumes half of the overall chip power. In contrast, the ALU occupies little area. Based on the findings of Chapter 5, it is assumed that a sufficiently fast ALU can be realized to match the register file speed. Because the register file is a limiting factor in system performance and datapath bandwidth (Chapters 3 and 4), it remains the focal point of discussion in this chapter.

System Timing

In Chapter 4, several datapath timing schemes were evaluated. The number of sub-phases in each datapath cycle were reduced by going from a shared bit-line to a dedicated bitline configuration. The delayed write and overlapped read/write schemes further reduce the cycle time. A four-fold speedup was predicted, as shown in Table I.

Datapath Timing Scheme (Chapter 4)	Bitline Configuration	Datapath Sub-Phases
Sequential	Shared Dedicated	4φ 3φ
Delayed Write or Overlapped Read/Write	Shared Dedicated	3φ 2φ
Delayed Write with Overlapped Read/Write	Dedicated	1φ

Table I: Datapath Timing Schemes

Overall system timing including pipelining is summarized in Table II. For all except the four-way pipeline, the datapath timing schemes vary the number of subphases, and thus system performance, by a factor of two. The overall range

of system bandwidth now doubles for an eight-fold variation. Performance of the sequential and two-way schemes is further affected by the frequency of data loads and stores; each incurs 50% or 100% overhead per cycle, respectively. For an instruction mix including 25% data loads and stores, system bandwidth can then vary ten-fold among the possible timing schemes.

Pipelining Scheme (Chapter 2)	Phases Per Instruction (Chapter 2)	Datapath Sub-Phases (Chapter 4)	Overall Cycle Time (sub-phases)
Sequential	2	4φ 3φ 2φ	8φ 6φ 4φ
Two-Way	1	4φ 3φ 2φ	4φ 3φ 2φ
Three-Way	1	4φ 3φ 2φ	4φ 3φ 2φ
Four-Way	1	1φ	1φ

Table II: System Cycle Time

Local Memory Capacity

In Chapter 3, discussion focused on optimal local memory size. Chip area was assumed sufficient to allow its implementation. In the real world, this may not be a valid assumption. The limited chip area must be shared among many functions. System architecture and microarchitecture both play important parts in determining how much room is available for local memory. A non-ideal local memory capacity impairs performance; some improvement may be achieved with increased swap support or pipelining.

Pipelining Scheme	Bitline Configuration	Number of Bitlines	Number of Wordlines	Area Factor
Sequential	Shared Dedicated	2 3	2 4	4 12
Two-Way	Shared Dedicated	2 3	2 4	4 12
Three-Way	Shared Shared Dedicated	2 4 4	4 2 4	8 8 16
Four-Way	Dedicated	4	4	16

Table III: Bit Cell Area Variation

Increased datapath pipelining generally requires a larger bit cell (Chapter 4). An effect of pipelining, then, is a reduction in the amount of local memory which can be realized in fixed chip area. Table III presents the number of bitlines and wordlines required for various levels of pipelining. A simple estimate of relative cell size may be obtained by the product of the number of bitlines (horizontal) and wordlines (vertical) passing through the cell. In the case of a pseudostatic bitcell design, the refresh line is added to the wordline count. This model yields a four-fold variation in bit cell size. The degree of pipelining then significantly affects the amount of local memory attainable on chip.

The RISC I, with a pseudostatic, dedicated bitline cell incurs an area factor of 12. The RISC II utilizes a static, shared bitline and wordline cell; its area factor is only 4. This factor of three difference in bitcell size predicted by our simple area model closely matches actual silicon implementation (2,733.5 versus 924 λ^2 per bit).

Local memory capacity is also limited by allowable power dissipation. Power for a static register file is determined by the number of registers. For each static register, one inverter of the pair maintains a constant current to ground. There is little or no additional power required for increased pipelining. If the optimal local memory size cannot be achieved due to power limitations, then pipelining may need to be increased for higher performance.

Static bit cell power consumption (in NMOS) may be reduced by lengthening the depletion load transistors. This increases cell area. The long depletion loads in the RISC II register file lengthened the cell sufficiently to admit four bitlines without additional area penalty; however, only two were used by the two-way pipelined datapath using dedicated bitlines [2].

Power dissipation may be reduced without an area penalty, using high-resistance polysilicon loads or dynamic storage. These strategies can increase register cycle time (due to longer write and restore delays), as well as the susceptibility to soft errors induced by alpha particles.

Complementary-MOS (CMOS) is attractive due to its extremely low static power dissipation, which is determined only by leakage currents. In the past, CMOS was used primarily in specialized, low-power applications, such as in digital watches and other battery-operated products. The additional area required for "wells" or "tubs" needed to accommodate the complementary devices made CMOS too expensive to compete with the (then simpler) NMOS process. The resulting emphasis placed on NMOS technology further widened the gap in performance between these two technologies. At present, however, NMOS chips have reached their limit in allowable power dissipation. A great deal of attention is now being focused on CMOS process development; it is emerging as the primary candidate for exploiting higher device densities.

Datapath Bandwidth

Datapath bandwidth has been discussed in terms of phases, assuming fixed phase length. Processor cycle times using this assumption were presented in Table II. In Chapter 3, register cycle time was considered to grow with the square-root of local memory capacity. Since the register delay makes up most of the cycle time, processor bandwidth decreases with enlarged memory capacity. System performance was reevaluated to include this effect.

Depending on the bitline configuration and level of pipelining employed, bit cell area was shown to vary (Table III). A larger cell requires longer bitlines and/or wordlines. This leads to increased delay, which follows the square root of cell area in a manner similar to that discussed in Chapter 3.

Pipelining Scheme	Datapath Sub-Phases	Bitline Configuration	Delay Factor	Relative Cycle Time
Sequential	4φ 3φ 2φ	Shared Shared Dedicated	2 2 $\sqrt{12}$	16 12 13.8
Two-Way	4φ 3φ 2φ	Shared Shared Dedicated	2 2 $\sqrt{12}$	8 6 6.9
Three-Way	4φ 3φ 2φ	Shared Shared Dedicated	$\sqrt{8}$ $\sqrt{8}$ 4	11.28 8.46 8
Four-Way	1φ	Dedicated	4	4

Table IV: Processor Cycle Times

The relative cycle time for various pipelining and datapath timing schemes is given in Table IV. Results are based on the number of sub-phases per cycle

(Table II) times the square root of the bit cell area factor (Table III). Whereas Table II predicted an eight-fold range in datapath bandwidth, the new results shows only half of this is actually achieved. (These results assume a constant memory capacity; where several bitcell entries yield the same number of datapath sub-phases, the smaller cell was chosen).

Both the RISC I and RISC II implementations utilized two levels of system pipelining. The RISC I, with its large bitcell and 3φ datapath timing scheme, has a relative cycle time of 10.4 using our delay model. The smaller bitcell of RISC II allowed a relative cycle time of only 8.0, despite a 4φ datapath cycle. Comparison of actual datapath bandwidth for the fabricated chips was not possible, due to design errors in the control logic of RISC I which did not allow the full datapath bandwidth to be attained.

Data Wordsize

A four-bit microprocessor may suffice for a microwave oven controller: little memory is addressed, operations are simple, and high speed is not required. A large microprocessor will do the job more than adequately, but it will not be cost-effective. Other applications such as number crunching of massive amounts of data pertaining to seismic exploration or weather observation require much more processing power. These scientific calculations using single and double precision floating point data require 64 and 128 bit wordsizes in conjunction with high processor bandwidth. Despite this wide range in wordsize, these applications all have one thing in common: specialization. Processor wordsize is determined unambiguously by the application.

In contrast, a time-shared, High-Level Language (HLL) programming environment supports a wide variety of uses. Data wordsize distribution includes 8-bit ASCII characters through 128-bit extended precision floating point numbers. In such an application, the choice of processor wordsize introduces

some interesting tradeoffs.

A wide datapath can execute in a single cycle operations which would otherwise require several cycles. However, the wide datapath requires proportionally more area and power resources. For a design where the datapath dominates chip area, such as the RISC II, this cost is significant. The wider datapath is also slower. Local memory cycle time (Chapter 3) as well as ALU delay (Chapter 5: conditional carry scheme) both increase with the square root of wordsize. Depending on the application, then, a wider datapath may or may not offer improved performance.

Today, typical, time-shared HLL systems utilize 32-bit processors. This allows up to 4 Gigabytes of memory to be addressed, which is sufficient for most applications. Complex arithmetic operations, such as multiplication, may be best performed by a co-processor on the system bus. Such a co-processor may even handle larger word sizes than the CPU itself.

Assuming the CPU is required to work with words of 32 bits or less, it is interesting to observe the effect of reducing the datapath width to 16 bits. Bandwidth increases by 41% due to the smaller ALU and local memory size. Overall performance then will be improved if less than 29% of the instructions require double cycles for 32-bit data.

These multiple cycles include 32-bit data loads and stores as well as ALU and shift operations. Additionally, each program branch (jump, call, return) may modify the upper half of the 32-bit address. This also requires an additional cycle. Extra cycles due to normal program counter incrementing may be neglected, since they are very infrequent.

An additional incentive for reducing wordsize is that more functionality can be added using the resources made available. Increased local memory capacity, better swap support, and more specialized ALU functions (such as 2-bit multiply)

can be added to further increase performance.

In practice, the "dead time" between clock phases as well as the clock delays themselves reduce the 16-bit speed improvement. Also, more registers will be utilized in the 16-bit processor, since each 32-bit datum requires two registers. Although this may not justify increased window size since only five or less registers are typically used per procedure [3], swapping overhead will increase for the partial swap scheme. A more detailed examination of data lengths for the particular application is necessary in order to evaluate the impact of reduced wordsize.

Designing for Limited Chip Resources

As we have seen, the local memory occupies the largest part of the datapath area on the chip and it is the most critical component determining processor bandwidth. In designing for limited area, realizable local memory capacity is reduced with pipelining. Local memory capacity may also be limited by maximum die size in one dimension, which sets a limit on the overall length of the datapath. In Figure 1, the RISC II local memory size was restricted in the number of registers by the maximum mask pattern size and package cavity. For that design, the critical chip cost due to pipelining is attributed to the number of wordlines.

Table V compares area and length costs per unit bandwidth for fixed capacity local memory. These costs are given in terms of area- and length-delay products in order to account for processor bandwidth variation; they are obtained by multiplying the cycle time (Table IV) by the area or length factor (Table III). The highest performance return for a given amount of area or length is seen to occur for the two-way pipelined, delayed-write scheme with shared bitlines. This is similar to the approach used in the RISC II microprocessor [4].

Pipelining Scheme	Datapath Sub-Phases	Bitline Configuration	Area-Delay Product	Length-Delay Product
Sequential	4φ 3φ 2φ	Shared Shared Dedicated	64 48 166	32 24 55.4
Two-Way	4φ 3φ 2φ	Shared Shared Dedicated	32 24 83	16 12 27.7
Three-Way	4φ 3φ 2φ	Shared Shared Dedicated	90 68 128	23 17 32
Four-Way	1φ	Dedicated	64	16

Table V: Relative Chip Area and Length Costs per Unit Bandwidth
(costs given as Area- and Length-Delay products to reflect performance)

Figure 2 presents the "Tower of Hanoi" benchmark data for the fixed-swap scheme from Chapter 3. Performance is compared among the pipelining schemes of Table V, using the best implementation for each level of pipelining. Relative performance is plotted as a function of chip area, in units of the area factor times the number of local memory windows. Since one window in RISC is reserved for global data, entries begin at two windows.

In accordance with the area-delay product in Table V, the 2-way pipelined implementation (#2 in the figure) yields the best performance with little chip area. It even outperforms the three-way version (#3) over the entire range. The four-way version (#4) is not better until three or four times the area is available; maximum performance improvement over the two-way implementation is about 50%.

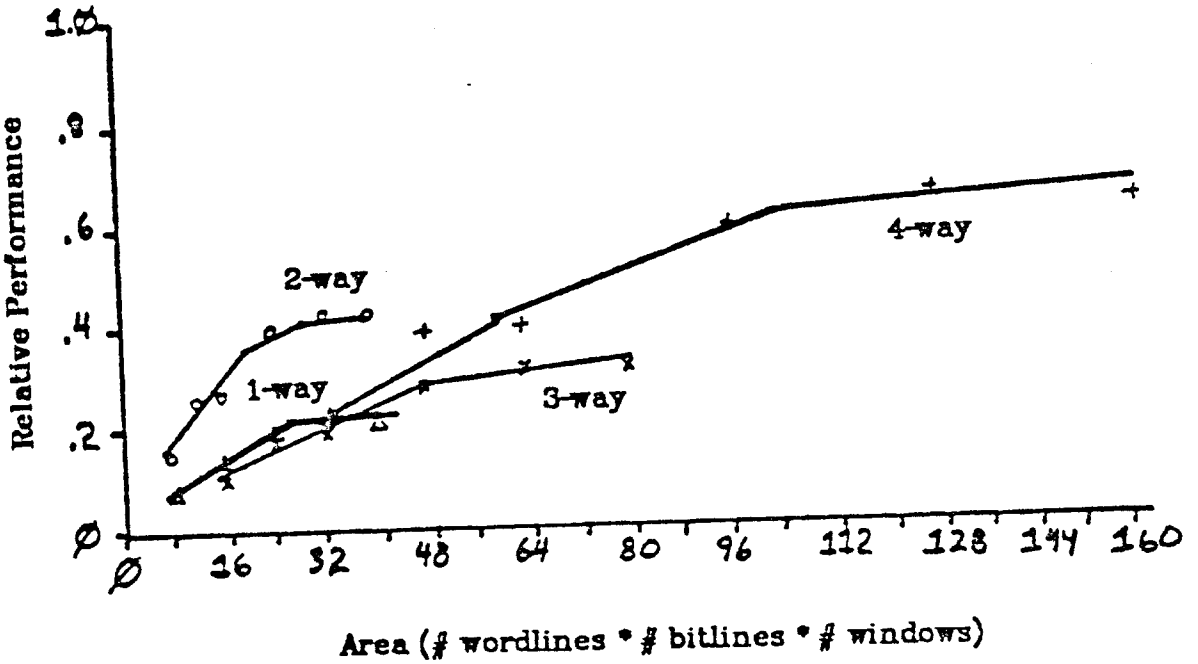


Figure 2: Pipelined System Performance Versus Local Memory Area

(RISC II executing "Tower of Hanoi," fixed swaps, 2 cycles per register)

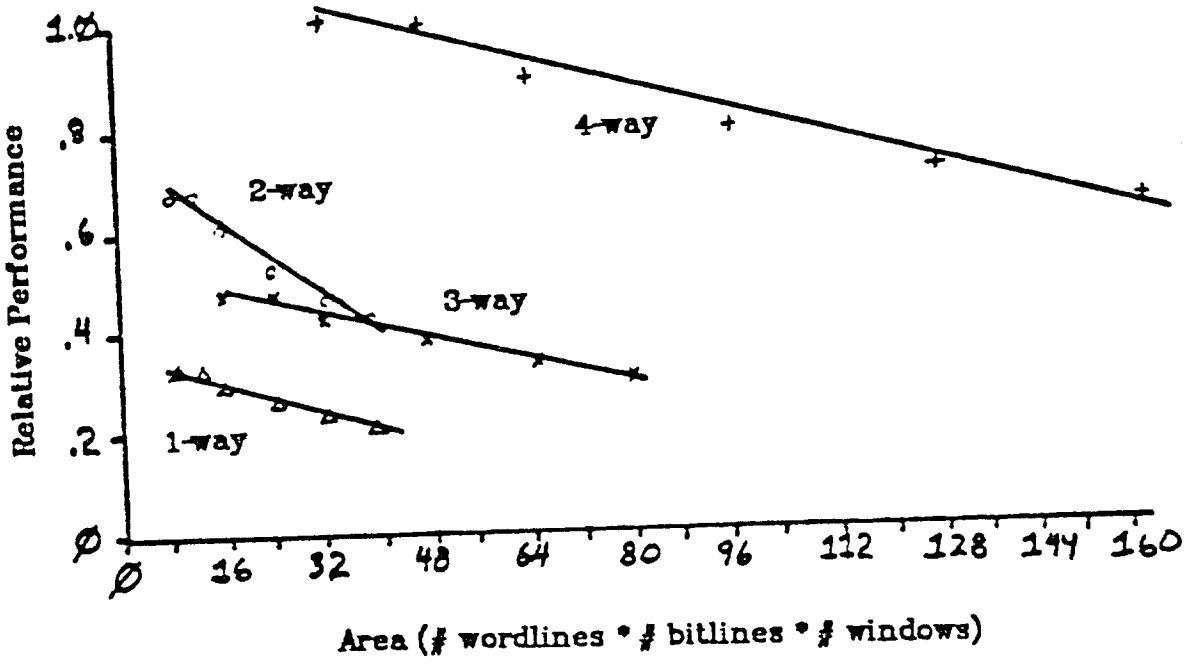


Figure 3: Pipelined System Performance Versus Memory Area

(RISC II executing "Tower of Hanoi," partial swaps)

Figure 3 presents data using the more efficient partial swap method of local memory management, which was seen to perform the best in Chapter 3. With such a reduction in swap overhead, performance now degrades noticeably as the local memory capacity is increased, due to the increase in register cycle time. Overall performance improves by 50% versus the fixed swap scheme, while requiring only two windows (less than a third of the capacity required for the fixed swap scheme) for peak performance. As before, the four-way version yields 50% better throughput than the two-way, at a cost of three times the register area. The two-way pipeline remains superior to the three-way pipeline.

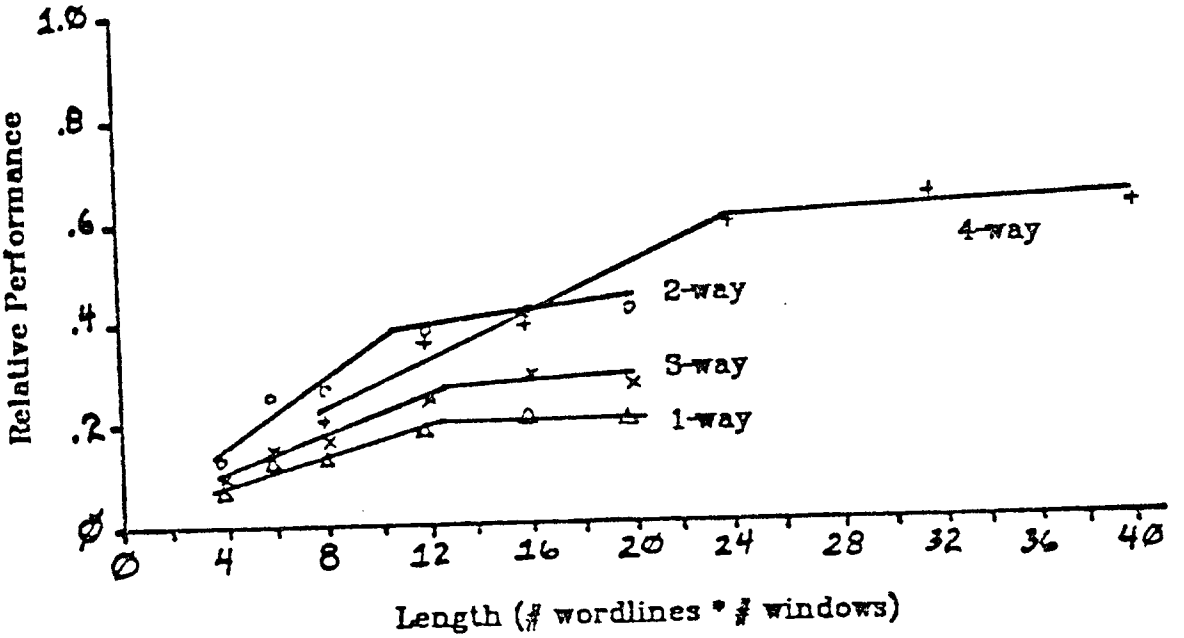


Figure 4: Pipelined System Performance Versus Memory Length
(RISC II executing "Tower of Hanoi," fixed swaps, two cycles per register)

Figures 4 and 5 present similar results, this time in terms of the chip *length* constraint. Performance is given versus the number of bit cell wordlines times the number of windows. Relative performance of the pipelined schemes is similar to that in Figures 2 and 3. However, the variation in length cost is not as dramatic as that for area; only a factor of two in length separates the optimal

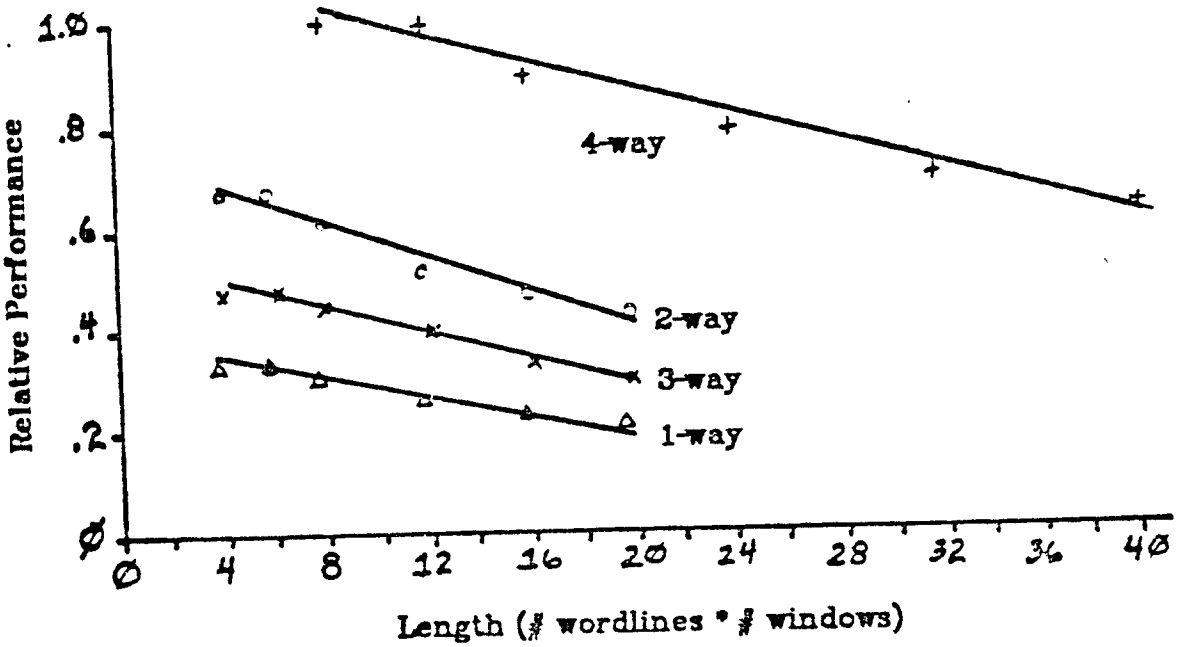


Figure 5: Pipelined System Performance Versus Memory Length

(RISC II executing "Tower of Hanoi," partial swaps)

two-way and four-way implementations.

In order to compare performance in view of limited power resources, relative cycle times given in Table IV are used. Again, the two-way and four-way schemes are the best performers.

Performance measured using benchmarks with few procedure calls and returns will be similar to that for the partial swap scheme, as it significantly reduces the swap cost. In this case, optimal local memory size consists of only a couple windows.

Overall, the four-way pipeline gives the best performance, as expected. However, limited chip size may not allow this performance to be attained. In this case, the two-way pipeline may offer the best results. These results will vary with the amount of data I/O cycles and data dependencies encountered by the system.

As we have seen, processor design optimization in VLSI is a complex task, which must account for the limited resources available on a chip. The microarchitect must not only be familiar with the limitations of the integrated circuit technology available; he must also have some knowledge of the demands of the programming environment for which the processor is designed. This is truly a great challenge for the microarchitect.

References

[1] M.G.H. Katevenis, R.W. Sherburne, D.A. Patterson and C.H. Séquin: "The RISC II Micro-Architecture", Proceedings of the IFIP TC10/WG10.5 International Conference on Very Large Scale Integration (VLSI '83), Trondheim, Norway, pp. 349-359, August 1983.

[2] R.W. Sherburne, M.G.H. Katevenis, D.A. Patterson, C.H. Séquin: "Datapath Design for RISC," Proceedings of the Conference on Advanced Research in VLSI, Massachusetts Institute of Technology, pp. 53-62, January 1982.

[3] D. Halbert, P. Kessler: "Windows of Overlapping Register Frames," CS 292R Final Class Report, Computer Science Division, University of California, Berkeley, Spring 1980.

[4] D.T. Fitzpatrick, J.K. Foderaro, M.G.H. Katevenis, H.A. Landman, D.A. Patterson, J.B. Peek, Z. Peshkess, C.H. Séquin, R.W. Sherburne, K.S. VanDyke: "VLSI Implementations of a Reduced Instruction Set Computer," VLSI Systems and Computations, Carnegie-Mellon University Conference, Computer Science Press, pp. 327-336, October 1981. Also in: "A RISCy Approach to VLSI," VLSI Design, vol. II, no. 4, pp. 14-20, 4th qtr. 1981, and Computer Architecture News (ACM SIGARCH), vol. 10, no. 1, pp. 28-32, March 1982.

CHAPTER 7:

CONCLUSIONS

There are many tradeoffs to be considered in the design of a microprocessor. Often, these tradeoffs are interrelated and thus increase the difficulty of the task of the chip designer. In order to simplify understanding of these issues, this work has first present individual design areas in which tradeoffs can be made. Each of these design areas has been discussed individually in order to clarify the range of choices and their associated costs. Later, overall chip design was viewed with reference to all of these design tradeoffs combined in different ways.

In Chapter 1 the special constraints of VLSI single-chip processors were introduced. The high cost of custom design favors a simple and regular implementation. The RISC architecture addresses these issues by simplifying the instruction set and thereby reducing the control logic on the chip. This not only frees up valuable chip area, but also reduces design time significantly. The notion of limited chip resources (area, pins, power) sets the context for the rest of the paper. Attention is focused on the datapath itself, since it dominates chip area in RISC implementations, and its performance limits overall system speed.

System pipelining was investigated in Chapter 2. With careful design of the

datapath, pipelining may produce significant performance gains. As the degree to which pipelining is exploited is increased, however, data and jump dependencies make it more difficult to attain further speedup. As a result, a careful study of program behavior is necessary in order to accurately assess the value of various levels of pipelining. At some point, limited chip resources are better utilized to speed up other critical paths in the system rather than to support added pipelining.

Local memory tradeoffs were discussed in Chapter 3. A fundamental limit to performance exists due to the memory I/O traffic alone. Data memory traffic can be significantly reduced through the use of an on-chip local memory organized in multiple banks. Careful study is necessary in order to determine the ideal size of this local memory. A large local memory reduces datapath bandwidth and consumes resources available for other functions; too small a local memory will result in a processor that is restricted by data I/O. In some cases, however, more sophisticated hardware support for local memory management can compensate for this performance loss. Local memory design was a critical factor in optimizing the performance of the RISC microprocessors.

Datapath timing for register-based machines was examined in Chapter 4. Several schemes were presented in order to reduce the number of required clock phases in each datapath cycle. The corresponding increase in concurrency requires different register bitcell designs. In some cases, additional circuitry is needed in order to eliminate data dependencies within the datapath itself.

Design tradeoffs for the ALU were discussed in Chapter 5. Several adder schemes were compared: ripple, carry-select, and parallel. An initial analysis was performed based on the assumption of fixed gate delay, which is applicable to TTL-based implementations. Next, results of NMOS circuit simulations were

utilized to obtain a more realistic comparison of these schemes for VLSI implementation. Because dynamic logic and bootstrap techniques are available in NMOS technology, these results differ significantly from those obtained with the fixed gate delay model. The NMOS ripple carry performed best at 8 bits, while the carry-select was optimal through 128 bits. The parallel adder was determined to be undesirable for VLSI implementation because of its large area and power consumption. This contrasts with the TTL-based results, where the parallel adder is most attractive.

In Chapter 6, all of the previous design areas were considered together in order to evaluate overall processor performance under the constraint of limited chip resources. Higher levels of pipelining were found to be of diminishing return; the bit cells needed to support increased concurrency reduce the bandwidth of the datapath. The two-way pipelined system with a register file using shared bitlines and a delayed write scheme was found to make the most efficient use of limited local memory area. Such a design was utilized in the RISC II microprocessor.

Each system application requires its own analysis for optimization of performance. Additionally, design decisions must be continually reassessed as the available chip resources and constraints change with improvements in technology. It is hoped that the ideas brought out in this paper will address the nature of critical processor design tradeoffs and will prove useful to other designers faced with the task of fitting a high-performance processor onto a single VLSI chip.