ORGANIZATION AND VLSI IMPLEMENTATION OF MIPS

**Steven A. Przybylski, Thomas R. Gross, John L. Hennessy,
Norman Jouppi, Christopher Rowen**

Technical Report: CSL-TR-84-259

APRIL 1984

The MIPS project has been supported by the Defense Advanced Research Projects Agency under contracts # MDA-903-79-C-0680 and # MDA 903-83-C-0335. Steven Przybylski has a scholarship from the Natural Sciences and Engineering Research Council of Canada and Christopher Rowen is supported by an IBM Fellowship.

Organization and VLSI Implementation of MIPS

Steven A. Przybylski, Thomas R. Gross, John L. Hennessy, Norman P. Jouppi, Christopher Rowen

Computer Systems Laboratory
Stanford University

Abstract

MIPS is an 32-bit, high performance processor architecture implemented as an nMOS VLSI chip. The processor uses a low level, streamlined instruction set coupled with a fast pipeline to achieve an instruction rate of two million instructions per second. Close interaction between the processor design and compilers for the machine yields efficient execution of programs on the chip. Simplifying the instruction set and the requirements placed on the hardware by the architecture, facilitates both processor control and interrupt handling in the pipeline. High speed MOS circuit design techniques and a sophisticated timing methodology enable the processor to achieve a 250nS clock cycle.

The MIPS project has been supported by the Defense Advanced Research Projects Agency under contracts # MDA 303-79-C-0680 and # MDA 903-83-C-0335. Steven Przybylski has a scholarship from the Natural Sciences and Engineering Research Council of Canada and Christopher Rowen is supported by an IBM Fellowship.

Table of Contents

List of Figures

List of Tables

1 Introduction

A computer architecture is measured by its effectiveness as a host for applications and by the performance levels obtainable by implementations of the architecture. The suitability of an architecture as a host is determined by two factors: its effectiveness in supporting high level languages, and the base it provides for system level functions. The efficiency of an architecture from an implementation viewpoint must be based both on the cost and performance of implementations.

MIPS is a 32-bit processor design implemented in VLSI that attempts to seek the best compromise between performance and functionality. The importance of these factors in the overall effectiveness of the architecture varies, but the role of program host probably remains most important. However, the instruction set designer must carefully consider both the usefulness of the instruction set for encoding programs and the performance of implementations of that instruction set.

Since most programs are written in high level languages, the role of the architecture as a host for programs depends on its ability to express the code generated by compilers for the high level languages of interest. The effectiveness is a function of the compiler technology and, to a lesser extent, the programming language. Compilers tend to translate languages to similar types of code sequences. However, some unique language features may be significant enough to deserve treatment at the architectural level. Examples of such of features might include: support for tags in a language with tagged objects, support for floating point arithmetic, and support for parallel constructs. In considering such features, the architect must evaluate both the cost of implementing hardware support and the use of the feature relative to more commonly used features.

Program optimization has become a standard part of many compilers built today. Thus, the architecture should be designed as a target for compiler translation and optimization. One of the key implications of this observation is that the architecture should expose as much computation as possible. Unless the implications of a particular machine instruction are visible the compiler cannot make a reasonable choice between two alternatives. Likewise, hidden computations cannot be optimized away. This view of the optimizing compiler argues for a simplified instruction set that maximizes the visibility of all operations needed to execute the program.

A large instruction set architecture will require microcode to implement the instruction set. In VLSI, silicon area limitations often force the use of microcode for all but the smallest and simplest instruction sets. In a processor that is microcoded, an additional level of translation, i.e., from the machine code to microinstructions, is done by the hardware. By allowing the compiler to implement this level of translation, the cost of the translation is taken once at compile-time rather than repetitively every time a machine

instruction is exccutcd. This view of the optimizing compiler as generating microcode for a simplified instruction set is explained in depth in [10]. In addition to eliminating a level of translation, the compiler "customizes" the generated code to fit the application [11]. This customizing can be thought of as a realizable approach to dynamically microcoding the architecture. MIPS attempts to exploit this view by "compiling down" to a low level instruction set.

The architecture and its strength as a compiler target determine much of the performance at the architectural level. However, to make the hardware usable an operating system must be created on the hardware. The operating system requires certain architectural capabilities to achieve full functional performance with a reasonable efficiency. If the necessary features are missing, the operating system will be forced to forego some of its user-level functions. This architectural support is absolutely necessary and will often require underlying hardware support to achieve acceptable performance. Among the features considered necessary in the construction ofmodern operating systems are:

- Privileged and user modes with protection of specialized machine instructions in user mode.
- Support for external interrupts and in ternal traps.
- Memory mapping support including support for demand paging, and provision for memory protection.
- Support for synchronization primitives if conventional instructions cannot be used, as in a multiprocessor.

Some architectures provide additional instructions for supporting the operating system. These instructions are included for two primary reasons. First, to establish a standard interface for some function that may be hardware dependent. Second, to enhance the performance of the operating system by supporting some special operation in the architecture. Standardizing an interface in the architectural specification can be more definitive, but it can carry performance penalties when compared for example, to a standard at the assembly language level, which is implemented by macros, or to an interface that is implemented by standard libraries. Putting the interface into the architecture 'commits the hardware designers to supporting it, but it does not inherently enforce or solidify the interface.

Enhancing operating system performance via the architecture can be beneficial. However, such enhancements must be compared to alternative improvements that will increase general performance. Even when significant time is spent in the operating system, the bulk of the time is spent executing general code rather than special functions, which might be supported in the architecture. Thus, the architect must carefully weigh the architectural feature to determine how it affects other components of the instruction set (overhead costs, etc.), as well as the opportunity cost related to the components of the instruction set that could have

been included instead. Often the cost of the feature when combined with its low merit (due primarily to infrequent usage and secondly to lack of a major performance improvement) forms a strong argument against its presence in the architecture.

The organization of the hardware for an architecture can dramatically affect quantitative measures of architectural performance. Since the architecture imposes implementation requirements on the hardware, performance measurements made on the architecture that are implementation independent, may not yield realistic measures of the performance of an actual implementation of the architecture. This is especially true when the implementation is in VLSI. In VLSI, the interaction of the architecture and its implementation is more pronounced. This is due to a variety of causes, chief among them being:

- The limited speed of the technology encourages the use of parallel implementations. That is, many slower hardware components are used rather than a smaller number of fast components. This basic design methodology has been used by a wide range of designers on projects as varied as systolic arrays [14] to the MicroVAX I datapath chip [16].

- The cost of complexity in the architecture. This is true in any implementation medium, but is exacerbated in VLSI, where complexity becomes more difficult to accommodate. A corollary of this rule is that no architectural feature comes for free.

- Communication is more expensive than computation. Architectures that require significant amounts of global interaction will suffer in implementation.

- The chip boundaries impose hard limits on data bandwidth and a substantial penalty when compared to on-chip transmission times.

The architecture affects the performance of the hardware primarily at the organizational level, where it imposes certain requirements. Smaller effects occur at the implementation level where the technology and its properties become relevant. The technology acts strongly as a weighting factor favoring some organizational approaches and penalizing others.

The key goal in implementation is obviously to provide the fastest hardware possible; this translates into two rules:

1. Minimize the overall clock speed of the system. This implies both reducing the overhead on instructions as well as organizing the hardware to minimize the delays in each clock cycle.

2. Subject to the first guideline, give preference in performance to the most heavily used parts of the architecture.

This second rule may dictate sacrificing performance in some components of the architecture in return for increased performance of the more heavily used parts.

The observation that these types of tradeoffs are needed, together with the fact that larger architectures generate additional overhead, have led to the reduced instruction set approach [22, 20]. Such architectures are streamlined to eliminate instructions that occur with low frequency in favor of building such instructions out of sequences of simpler instructions. The overhead per instruction can be significantly reduced and the implementor does not have to discriminate among the instructions in the architecture. In fact, most simplified instruction set machines use single cycle execution of instructions to eliminate complex tradeoffs both by the hardware implementor and the compiler writer. The simple instruction set permits a high clock speed for the instruction execution, and the one-cycle nature of the instructions simplifies the control of the machine.

The MIPS architecture has been designed to maximize its effectiveness as an high-level language host, to provide sufficient systems-level support, and to allow high performance implementation in VLSI.. MIPS uses a simplified, low-level instruction set. The compiler takes advantage of the instruction set by generating code requiring little additional interpretation at execution time. Because all the effects of an instruction sequence are exposed to the compiler, the compiler can optimize the code to a very low level of detail. The architecture matches its implementation well, and many of the more complex tasks required from the architecture are supported by a level of software corresponding to microcode in other machines. The architecture supports the operating system functions that are required, and provides some extra functionality (limited memory mapping) on-chip.

2 Architecture of MIPS

Within the constraints of the technology the goal for MIPS was to provide suitable systems support and maximize performance for compiled code. The instruction set is simple, orthogonal, and suitable for efficient execution of compiled code. This simplicity allows all instructions to execute in the same amount of time, and all to be one word in length.

MIPS is a load/store machine; this isolates memory access instructions, as well as facilitating the fixed instruction length and execution time. A small set of addressing modes is supported for both load and store instructions. The memory is word-addressed. This streamlines the memory interface both on and off chip and increases performance, since the bulk of the references are to word length items. Byte addressing is supported by a group of instructions that have capability similar to the byte pointer facility on a DEC-20. The comparison of byte and word addressing together with the details of the byte addressing support are discussed in [9].

There are sixteen general purpose registers; the instruction formats treat the general purpose registers uniformly. All ALU instructions are register-register and are available in two and three operand formats; one

of the source registers may be replaced by a small constant. Support for integer multiplication and division consists of special multiply and divide instructions (see Section 3.6.2) that are expanded into sequences of real machine instructions, No on-chip hardware is provided for floating point arithmetic.

The architecture has no condition codes. Instead, there is a compare-and-branch operation. Abandoning condition codes in favor of a compareand-branch instruction has benefits for both the compiler and the processor implementation [9]. In general, it simplifies pipelining and branch handling in the implementation and eliminates the need to attempt optimization of the condition code setting.

2.1 The visible instruction set

The compiler and operating system would prefer to see a simple, well-structured instruction set. However, this conflicts with the goal of exposing all operations, and allowing the internal processor organization to closely match the architecture. To overcome these two conflicting requirements, the MIPS instruction set architecture is defined at two levels. The first level is the level that is visible to the compiler or assembly language programmer. It presents the MIPS machine as a simple, streamlined processor. Table 1 summarizes the MIPS definition at this level. Each of these assembly-level instructions is translated to machine level instructions; this translation process also includes a number of machine-dependent optimizations. The machine level instructions that are executed by the hardware and a brief description of the optimizing translation appear in Section 3.1.1.

2.2 Systems issues

The ability of the processor to deal with interactions between itself and its support environment (peripherals, memory, etc.) can greatly affect the overall system performance, as well as limit the scalability of the architecture across different performing levels. The designer must consider the systems level aspects of the architecture in a complete and consistent fashion that will accommodate a wide range of implementations.

The MIPS architecture aims to support a variety of high performance workstation environments. At the low end we anticipate dedicated control engines with no virtual memory support and few, if any, peripherals. A high end application might be a sophisticated multiprocessing workstation with several megabytes of memory, local disks, and network and graphics capabilities. A great many of the characteristics of the resulting architecture and subsequent organization and implementation are based on the requirements of the high end of this spectrum. The key theme is to make the systems-level architecture match the high performance instruction set architecture. A consistent effort is made to minimize the complexity of the interfaces so the architecture is useful in simpler systems. Wherever possible, flexibility for the non-CPU components is retained. Besides lessening the implementation task and processor support requirements, simplifying the external interfaces leads to higher performance.

Operation	Operands	Comments	
Arithmetic and logical operations			
Add	srcl, src2, dst	dst: = src2 + srcl	Integer addition
And	srcl, src2, dst	dst: = src2 & srcl	Logical and
Ic	srcl, src2, dst	dst: = byte srcl of dst is replaced by src2	Insert byte
Or	srcl, src2, dst	dst: = src2 | srcl	Logical or
Rlc	srcl, src2, src3, dst	dst: = src2||src3 rotated by srcl positions	Rotate combined
Rol	srcl , src2, dst	dst: = src2 rotated by srcl positions	Rotate
Sll	srcl, src2, dst	dst: = src2 shifted left by srcl positions	Shift left logical
Sra	srcl, src2, dst	dst: = src2 shifted right by srcl positions	Shift right arithmetic
Srl	srcl, src2, dst	dst: = src2 shifted right by srcl positions	Shift right logical
Sub	src 1, src2, dst	dst: = src2 - srcl	Integer subtraction
Subr	srcl, src2, dst	dst: = srcl - src2	Reverse integer subtraction
Xc	srcl, src2, dst	dst: = byte srcl of src2	Extract byte
Xor	src 1, src2, dst	dst: = src2 ⊕ srcl	Logical xor
Transport operations			
Ld	A[src], dst	dst: = M[A + src]	Load based
Ld	[srcl + src2], dst	dst: = M[srcl + src2]	Load based-indexed
Ld	[srcl>>src2], dst	dst: = M[srcl shifted by src2]	Load based-shifted
Ld	A, dst	dst: = M[A]	Load direct
Ld	I, dst	dst: = I	Load immediate
Mov	src, dst	dst: = src	Move (byte or register)
St	srcl, A[src]	M[A + src]: = srcl	Store based
St	srcl, [src2 + src3]	M[src2 + src3]: = srcl	Store based-indexed
St	srcl, [src2>>src3]	M[src2 shifted by src3]: = srcl	Store based-shifted
St	src, A	M[A]: = src	Store direct
Control transfer operations			
Bra	dst	PC: = dst + PC	Unconditional relative jump
Bra	Cond, srcl, src2, dst	PC: = dst + PC if Cond(srcl,src2)	Conditional jump
Jmp	dst	PC: = dst	Unconditional jump direct
Jmp	A[src]	PC:= A + src	Unconditional jump based
Jmp	@A[src]	PC: = M[A + src]	Unconditional jump indirect
Trap	Cond, srcl, src2	PC: = 0 if Cond(srcl, src2)	Trap instruction
Other operatiotrs			
SavePC	A	M[A]: = PC-,	Save multi-stage PC after trap or in terrupt
set	Cond, src, dst	dst: = -1 if Cond(src,dst) dst: = 0 if not Cond(src,dst)	Set conditional

Table 1: MIPS assembly instructions

The primary implications of supporting a sophisticated multiprocessing environment are that virtual memory and demand paging are necessary. Some sort of privilege support and a non-trivial memory hierarchy are also needed.

2.2.1 Exceptions in a pipelined machine

One of our primary goals in MIPS is to attain higher performance levels through the use of pipelining. Though this is typically an organizational or implcmcntation issue, it strongly affects the way in which exceptional conditions arc handled in the architecture. The interactions bctwccn these diffcrcn t conceptual levels are much stronger in VLSI than in TTL-like implcmcntation media.

One of the lessons of some of the more heavily pipclined complex architectures [I, 15] is that the handling of exceptions can become very complicated and irregular. One of the major problems is that some instructions change visible or hidden state before that instruction can bc guaranteed to complete without interruption. Pipelining complicates this because one instruction may change the machine state, and at some later time an instruction earlier in the pipeline (and hence earlier in a control-flow sense) may cause an exception. This means that the machine may have to undo changes it has made in the state, so that the instruction that faulted can be correctly restarted; this approach is often used to implement the auto- increment/decrement addressing modes on architectures such as the VAX [16]. Alternatively, the current state can bc saved as of the instant of the fault without retracting any changes. This introduces an added complexity in that it necessitates restarting the machine in the middle of an instruction, as in the Motorola 68010 [18].

Further problems arc introduced by instructions with very long execution times. To maintain a reasonable maximum interrupt latency, a very long instruction will need to bc interrupted and restarted. Many architectures with such instructions USC the general register set for intcrmcdiatc computations; this helps mininiize the amount of special support needed for interrupting long instructions. Multiple memory rcfcrcnccs per instruction arc a root cause of many of thcsc problems. When combined with demand paging support, architectures with multiple memory refcrcnccs per instruction will constantly bc faced with the problem of partial completion of an instruction.

The rcduccd-instruction-set philosophy counters many of these problems. First, all the instructions are simple, and thus short. Instructions that alter state before previous instructions finish are not a natural part of the architectural style. Load/store architectures do not have the problems associated with multiple memory rcfcrcnccs or the maximum in tcrrup t latency. Eliminating a few complex addressing modes, such as auto increment, removes the majority of the remaining cases that require the processor to be interruytcd after the executing instruction has changed the visible processor state.

2.3 Support for virtual memory

The primary requirement of the memory system was support for virtual memory management and demand paging. Other desirable attributes are a large, uniform address space for each process, and support for multiprocessing. One mechanism for facilitating multiprocessing is the incorporation of a process identification number into the virtual memory address. This helps achieve fast context switches by allowing the cache and memory address translation units to avoid the cold start penalties. These penaltics appear in systems that require caches and Translation Lookaside Buffers (TLBs) to be flushed because processes share the same virtual address space. The classical solution of fixed sized segmented address schemes (at least with small segments) inevitably runs into problems when a large application is run. The penalty for using segmented addressing is often quite high, both on the implementation and on compiled programs that **must** maintain and utilize the segmented addresses.

2.3.1 Memory Mapping

The primary motivation and constraint in the design of the memory management mechanism that was included in the MIPS architecture was the desire to retain flexibility in the implementation of the processos and any future systems. The realities of the initial implementation technology (a 4μm channel length nMOS) meant that it was not feasible to include all of the virtual to physical translation on the same chip as the processor.

Consequently, a novel memory segmentation scheme was added to the architecture. Each process has a process address space of 2^{32} words. The first step of the translation is to remove the top n bits of the address and replace them by an n bit *process identifier* (PID). Figure 1 shows this to be the virtual address generated by the CPU. Thus, the accessible portion of the process address space is split into the low 2^{31-n} words and the high 2^{31-n} words. An attempt to access any of the non-visible words will cause an exception. The operating system can then remap the process identifier in such a way as to give the faulting process a smaller PID number and thus a larger visible portion of its process address space. This will only happen if the program used more heap or stack space then it was initially allocated.

The constraining factor in this scheme is that the total size of all the visible process address spaces **must** be less than the size of the implementation's virtual address space. This restricts the number of processes that can actively use the memory map; should this number of processes become very large, the operating system will need to periodically reuse a PID. Whenever a process with a shared PID is made active, a process and cache sweep will be needed. This should not happen frequently, since the number of small processes **that** can be created is very large.

An additional level of translation maps the processor's virtual address to a physical address. A simple

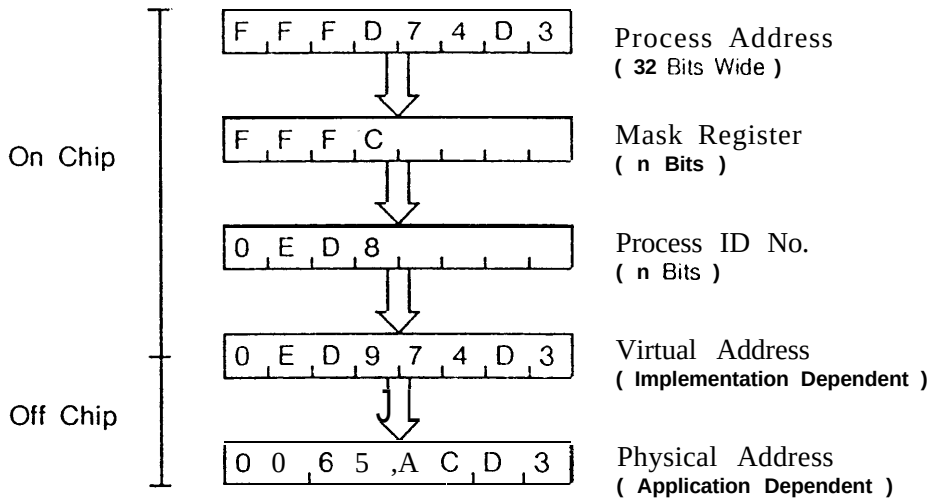


Figure 1: Address translation

system might USC a small direct map or even no external memory mapping, while a more sophisticated workstation might include a TLB and disk-resident page tables. The important point is that this mechanism allows portability across different implementations of the processor. The only effect of changing the size of the virtual address space is to change the total amount of the visible process address space. Figure 2 shows two processes running on an implementation in which the virtual address space is smaller than the per process address space. The first process currently has 64K words visible while the second has 8M words accessible. Since the process addresses are 8 bits wider then the virtual addresses, *n*, the number of masked bits must be greater than 8, and the high order 8 bits of the process identifiers are insignificant.

2.3.2 The processor/memory interface

Since one of the goals of the design was to retain as much flexibility in the design of the memory hierarchy as possible, the processor architecture specified only a portion of the aspects of the processor/memory interface. As the primary intended implementation was a single integrated circuit, a major concern was a limitation in the speed and number of busses necessary to match the performance of the processor.

MIPS differentiates between instruction and data memory accesses. This helps supply adequate memory bandwidth by simplifying the task of creating an instruction cache. Since instruction caches are read only, and the instruction locality is quite good for well structured programs, a small, simple cache can yield good hit ratios [23]. With current RAM access times, it is possible to run the current MIPS implementation with a cache only for instruction access and doing data access directly to main memory. Depending on the speed of the mapping hardware, this will incur a performance penalty from 0-10%.

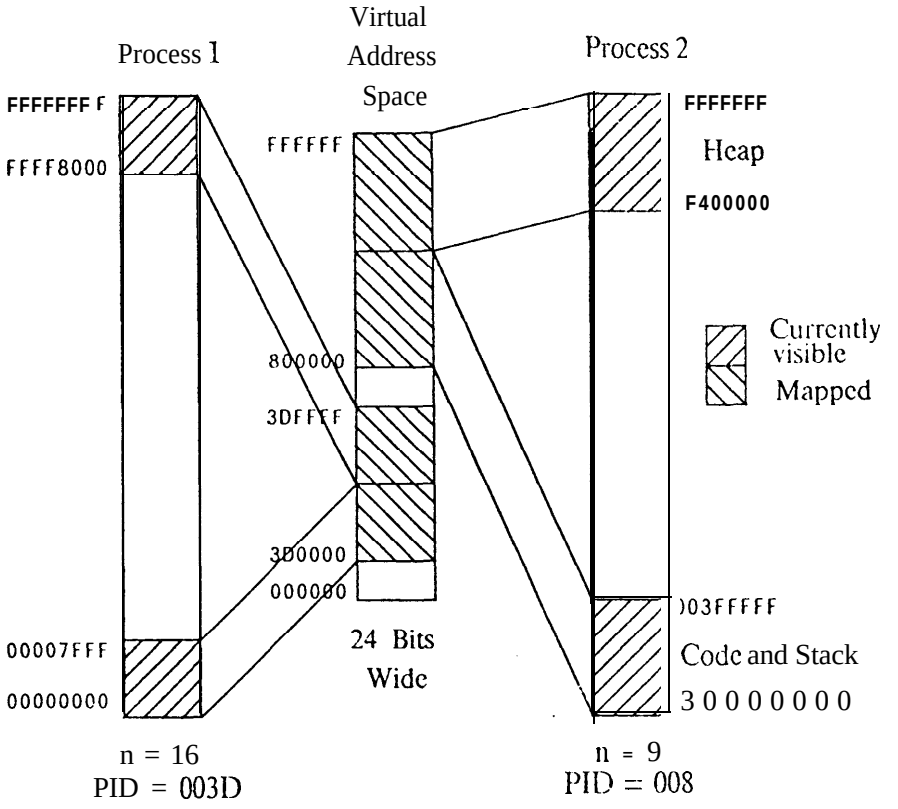


Figure 2: Address space mapping for two processes

2.4 Support for operating systems

A key concept in simplified instruction sets is transfering complexity from the hardware to the software when the expected benefits of a hardware implementation do not warrant their cost. However, there are situations in which some minimum level of hardware support is necessary to make the resulting system manageable. One such situation is that of privilege. The MIPS architecture provides for two levels of privilege: supervisor and user. The set of privileged instructions consists only of those that change either the process status register or any of the on-chip mapping registers. To allow the rest of the system to enforce protection, the current privilege level is included as part of the virtual address.

A software trap instruction with an ll-bit trap code field is used for system calls. This trap code field, along with exceptions on illegal instructions allows extensibility of the instruction set and variation among implementations.

The shift of the burden of complexity from the hardware to software yielded an unusual method for handling the infrequent case of trap on overflow. It is typically quite hard, without slowing down the

machine, to detect that an overflow has occurred early enough to prevent the bad result from overwriting a destination register. However, given one operand, the operation performed (for example, addition, subtraction, and single steps of multiplication or division) and 32 bits of the result, one can invert the operation to retrieve the second operand. Thus, the decision on how to proceed can be made by the programming language run-time system. It can either replace the bad result with some known value, or fix an operand and restart, or halt the running process. Although this undoing of an operation may cost a few dozen instructions, it is done so infrequently that system performance increases by putting this function in software rather than lengthening the clock cycle to prevent the overwriting in the hardware. While the recovery from infrequent overflow events can be done effectively in software, the detection of overflow after every integer operation would be far too costly a burden for the software Thus. the architecture supports user-maskable overflow detection and trapping on integer operations. This two-part approach typifies the MIPS philosophy: use a hardware implementation only where the performance gained, justifies the hardware cost.

3 Organization and implementation

In the MIPS processor, a close integration exists between the organization of the implementation and the architecture. This section discusses four related areas in the processor organization and its implementation. The first part presents the general processor organization and pipeline structure, including details of the machine-level instruction set and the mapping from the assembly level instructions. The organization and implementation of the control portions of the processor that run the pipeline. handle exceptions, and decode instructions are explained in the second portion of this section. The third area details the key features of the datapath implementation. Finally, we give some interesting design metrics for the VLSI implementation, elaborate on performance bottlenecks, and discuss techniques we have used to increase the speed of the chip.

3.1 Instruction set interpretation

The machine-level instruction set of MIPS is closely tied to the pipeline structure. The five pipestages and their functions are summarized in Table 2. All pipestages take the same time to execute, and each instruction makes a single pass through the stages in the order shown in Table 2. A new instruction is fetched on every other pipestage; that is, the machine cycle is exactly two pipestages in length.

The main machine resources that are used during the execution of an instruction are the instruction memory, the ALU (or the barrel shifter), and the data memory. Figure 3 illustrates the concurrent execution of pipestages and the allocation of the major resources to the individual pipestages. The allocation of resources to pipestages is static. The instruction memory is always 100% busy; the usage of the ALU and the data memory depends on the instruction mix executed. Many combinations of active instructions will result in 100% utilization of the machine resources.

Stage	Mnemonic	Task(s)
Instruct ion Fetch	IF	Send out the PC, increment it
Instruction Decode	ID	Decode instruction
Operand Decode	OD	Compute effective address and send to memory if load or store Compute new program counter if branch Use ALU for register-register operation otherwise
Operand Store/ Execution	SX	Write operand if store Use ALU for comparison if compare-and-branch Use ALU for register-register operation otherwise
Operand Fetch	OF	Read operand if load

Table 2: Major pipestages and their functions

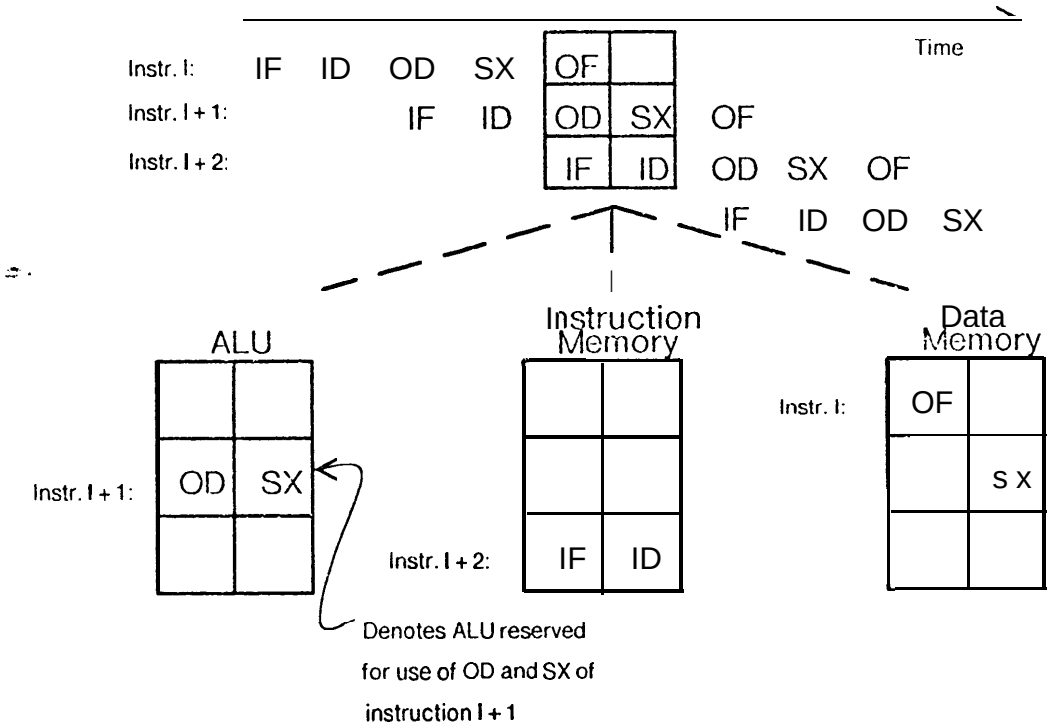


Figure 3: Static resource allocation of the MIPS pipeline

The execution of instructions in the pipeline can be easily understood by considering one example from each of the three major classes of instructions shown in Table 1:

1. an add instruction representing the arithmetic and logical instructions,

2. a compare-and-branch instruction representing the control transfer instructions,

3. a load instruction representing the transport instructions

The add instruction needs only the ALU and can be completely executed during the SX cycle of the pipeline.

Since MIPS is a load/store architecture, the arithmetic and logical instructions do not require use of the ALU during OD, nor do they use the data memory. A compare-and-branch instruction has to execute two basic operations: test whether the specified condition holds between the two arguments (registers), and modify the program counter if the condition is met. Both operations require the ALU: the condition evaluation is an ALU operation, as is the addition of the relative offset to the program counter to obtain the branch destination address. Thus, the compare-and-branch needs to make use of the ALU during both the OD and SX cycles; it does not utilize the data memory. The execution of a load instruction requires the use of the ALU only once to compute the effective address of the item that is to be retrieved from memory, leaving the ALU idle during SX. All memory reference instruction accesses the data memory during the OF and SX stages.

The static allocation of resources to pipestages and the synchronous pipeline with fixed length instructions would typically mean that the add instruction and the load instruction would underutilize the machine resources. However, the machine-level instruction set of MIPS allows multiple assembly language instructions to be packed into a machine instruction when the pipeline resource allocation and the available instruction encoding space both permit this packing. The structure of the add instruction, which does not use the data memory or the ALU during OD, and the structure of the load instruction, which does not use the ALU during SX, means that these two pieces can be executed in one machine language instruction.

Figure 4 shows the actual resource utilization for a sample instruction sequence. Assume that instruction I retrieves a nonzero value into register RO. That is, the branch is not taken. Instruction $I + 2$ of the sequence is an example of *a packed* instruction. Two independent assembly-level instructions are bound together into one machine-level instruction.

The second level of the MIPS processor architecture is closely linked to the machine organization; it allows two operator instructions (e.g., instruction I + 2 in Figure 4) and includes pipeline scheduling constraints. The actual hardware executes the machine-level instruction set; translation between assembly language (the architectural level) and the hardware instructions (organizational level) is done by the reorganizer [7]. The pipeline scheduling constraints arise in two forms: the absence of hardware interlocks on register access and the delayed branches.

3.1.1 Instruction reordering

One of the unique features of MIPS (and the source of its name, "Microprocessor without Interlocked Pipe Stages") is that the access for registers is not interlocked. Thus, the compiler is responsible for generating instructions that will correctly execute given the structure of the MIPS pipeline and the register accessing performed by the instructions. Figure 2 shows that a data word read from memory by instruction I arrives

instruction. If the branch instruction rcfcrcnccs memory, as in an indirect jump, two succeeding instructions are fetched and cxccutcd. This concept of a delayed branch is common in microprogramming and has also been used in the RISC [21] and IBM 801 architectures [22]. The compilers and code generators are unaware of this behavior of branch instructions; the reorganizer finds a suitable instruction to be the successor of a branch. The algorithms employed arc described in detail in [6].

3.2 Exceptions

The MIPS organization has a number of features that simplify the implementation of cxccptions. First, the pipeline can bc designed to guarantee that instructions do not alter any state out of order. As discussed in detail in Section 3.4, the memory subsystem must indicate whether or not a refcrcncc will complete by the end of SX. Thus, each memory refcrcncc is committed before any subsequent instruction does any potential harm. The synchronous nature of the pipeline also means that the list of combinations of possible cxccptions and pipcstagcs is quite short (see Figure 5). Furthermore, if an instruction encounters a fault in a particular pipestagc, the state of the other instructions in the CPU is precisely known.

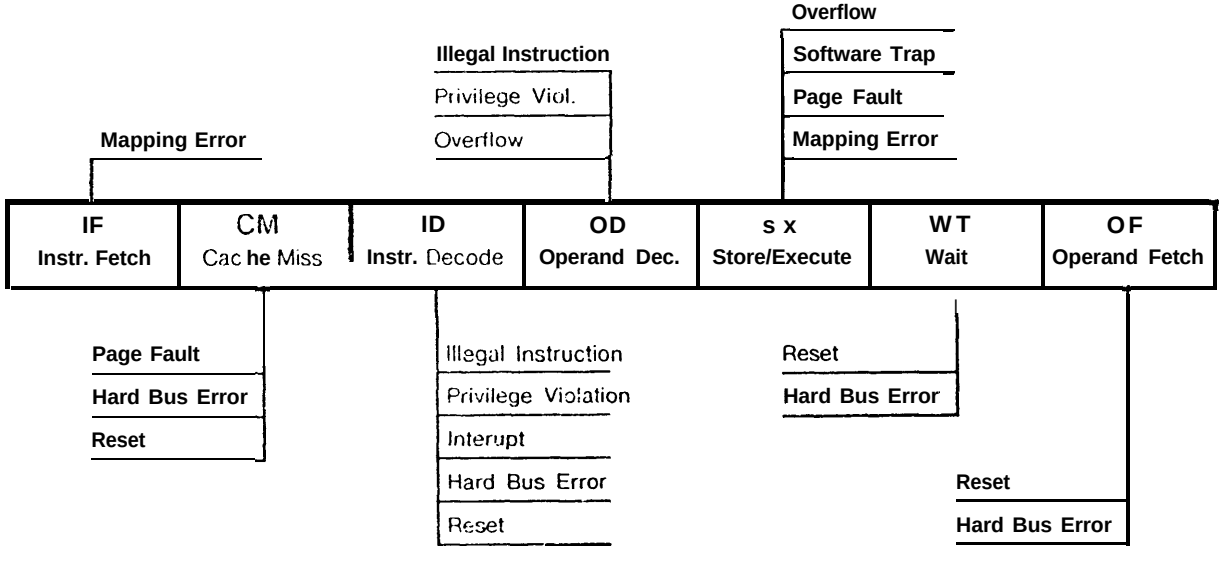


Figure 5: All possible exceptions

Although it appears that handling the exceptions should be trivial, the reality is complicated by the fact that several exceptions can happen at once. The system can choose to handle all the exceptions simultaneously, or one at a time. Handling the cxccptions one at a time is easier to implement and is sufficient provided that no irrcproducible infonnation is lost. 'l'hc general strategy is to report the exception that is associated with the

Instructions in the pipeline

```
I:    Ld 1[R14], R0          ;; No-op
I+1:  Bra Equal #0, R0, _next
I+2:  St R15, 3[R14]         ;; Add #1, R0
```

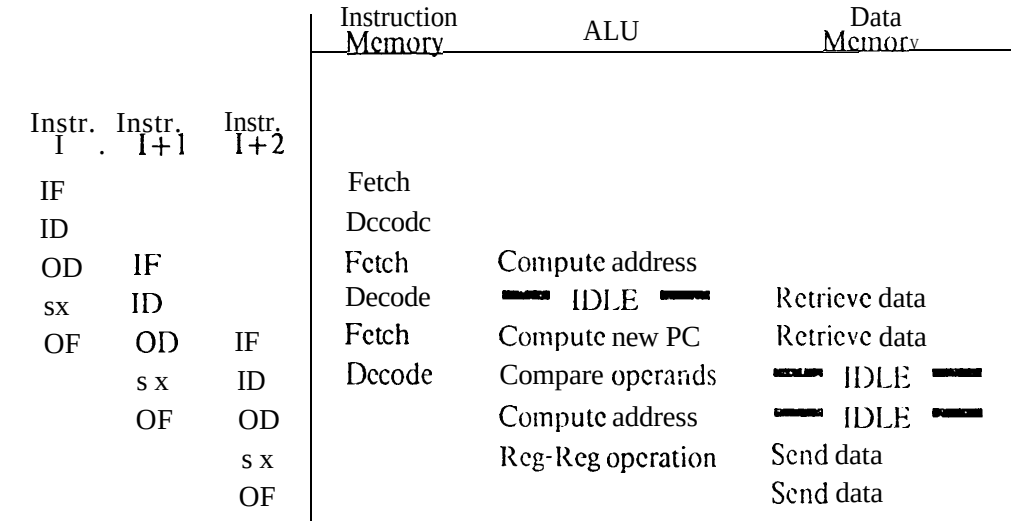


Figure 4: Dynamic resource utilization of the MIPS pipeline

during the OF cycle of that instruction. This cycle is executed concurrently with the OD cycle of the next instruction (Figure 3). Thus, instruction $I + 1$ cannot read or write the register that is the target of the load during OD cycle of instruction $I + 1$. The reorganizer reorders the instructions for each basic block to satisfy these constraints; this reorganization establishes at compile time the schedule of instruction execution. Scheduling instructions in software has two benefits: it enhances performance by eliminating instances of pipeline interlocking, and it simplifies the pipeline control hardware allowing a shorter time per pipestage [6].

During this instruction scheduling process, the reorganizer also attempts to pack assembly language instructions into machine-level instructions. When the reorganizer finds that it has two assembly instructions that can be scheduled next, it chooses the instruction that can be packed together with the previous instruction. This packing process means that the final code density is usually greater than one assembly language instruction per machine instruction.

3.1.2 Delayed branches

Figure 4 also illustrates another property of the MIPS hardware architecture that is invisible at the assembly-language level. A compare-and-branch instruction modifies the program counter at the end of the SX cycle. At this time, the next instruction has already been fetched and decoded. Rather than complicating the processor control to flush the pipe if the branch is taken, we define the semantics of the branch instruction to include a delay in the execution of the branch. Each branch instruction always executes one succeeding

earliest instruction. If several simultaneous exceptions happen to the same instruction, then the most serious exception is reported. For instance, if one instruction incurs an overflow in SX while its successor is faulting with an illegal instruction in ID, then the overflow will be the exception reported. If this instruction stream is resumed, then the illegal instruction exception will occur again.

If an instruction is subject to an exception early on in the pipeline, then the instructions that are ahead of it should be completed. Otherwise, the instruction stream will not be directly restartable (unless those instructions have not affected the state of the processor). This "running out" of the pipeline is not particularly difficult; however, complications arise when a second or third exception happens during the flushing. Consider the case when the last instruction of a resident page generates an arithmetic overflow in SX, and in the previous pipestage the succeeding instruction (on the next page) causes a page fault on instruction fetch. As the pipeline is being run out, the overflow will occur. The exception to be reported must be changed from page fault to overflow so that it indicates the failed execution of the earlier instruction in the pipeline.

Further confusion erupts with respect to the split instruction and data memory streams. If in the previous example, the first instruction generates a data reference fault rather than an overflow, a different problem arises. The instruction cache will report a miss in the attempt to fetch the second instruction. While the processor idles in cache miss states, the instruction cache tries to fetch the word from main memory causing a page fault. During the consequent emptying of the pipeline, a second main memory reference will be attempted for the data references of the first instruction. This reference will also fail with a page fault, and the system must report the page fault of the first instruction's data reference.

3.3 Operating System support in a hardwired machine

Architects of microcoded machines often use the microcode to add support for operating systems [5]. A great deal of the exception stack maintenance, TLB and page table maintenance, and general exception handling and dispatch is done by the processor's microcode. This is not possible in MIPS for the simple reason that it is not a microcoded machine.

However, there is a need for these functions to be always accessible with a minimum of delay. The solution in the MIPS implementation is to have a body of code permanently resident at location zero of physical memory. Whenever an exception occurs, the processor jumps to this location, and while executing the jump turns off address translation and disables interrupts. This is the target regardless of the exception type. At this location code exists to save a small amount of the more transient processor state, and to do the primary dispatch based on the details of the event. The corresponding return-from-exception sequence permanently resides in memory.

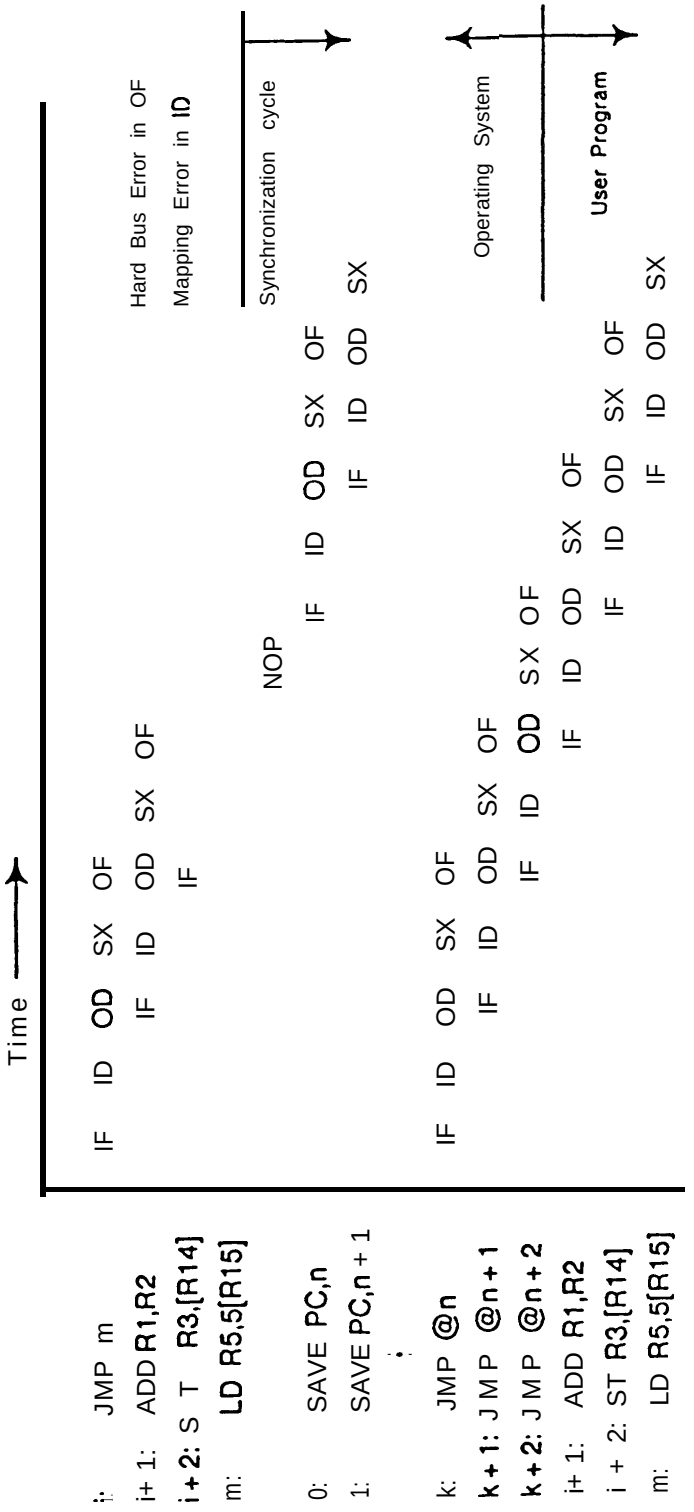


Figure 6: Exception sequences

raised during the second stage of either cycle, it indicates a bus error or other serious fault. In this case, the pipeline is also flushed, though perhaps in not a directly restartable way.

One of the basic premises of the MIPS architecture is that exposing the microengine in the architecture eliminates a level of interpretation and thus obtains more efficient utilization of the hardware. One could then reasonably expect these non-microcoded routines to execute nearly as fast as the equivalent microcoded routines of other machines. The non-microcoded machine's clock cycle is limited by cache access time and hit rate, while the microcoded machine's clock cycle depends on the the micromemory access time. The non-microcoded execution rate, coupled with the flexibility of user specified interrupt dispatching and state saving, allows tailoring of this lowest level of system software to any particular application.

As described in Section 3.1.2, the synchronous pipeline and fixed instruction execution time combine to imply a branch delay. The maximum branch delay of two implies that three return addresses must be saved on an exception to ensure proper restarting of the instruction stream in the worst case. Figure 6 shows a situation in which the instruction following a branch incurs an exception. The branch cannot be repeated since it successfully completed: it may have other effects. Thus the addresses of the instructions to be returned to are $i+1, i+2$, and m.

To save these three return addresses, the program counter includes a shift register structure that saves the address of the last three instructions fetched. After an exception and its subsequent jump to physical location. zero, these three values are accessible via a special instruction: SavePC. Returning from an exception requires three successive indirect jumps to the addresses saved. In the process of doing these jumps, the privilege and interrupt levels must be restored to reflect the change from system state back to user state.

3.4 The memory interface

Section 2.3.2 described the architectural aspects of the interface between main memory and the CPU. The split instruction and data streams provide a basic mechanism for making more effective use of main memory bandwidth. These two streams are interleaved to even out the bandwidth requirements across the chip's boundaries. Over the course of two pipestages, two addresses are sent out and two words are returned. As there is a single mapping unit that operates on both instruction and data addresses. the natural decomposition is to have separate data and address busses. Combining them onto a single bus would increase the complexity ofthe internal and external bus multiplexing circuitry as well as doubling the required bandwidth of the bus.

Figure 7 shows the timing of the transfers on the address and data bus and their relation to each other. Although the access time of the instruction cache is less than that of the data memory, both access types span two Cycles. The external unit must signal, within the first cycle, if it needs more time to complete the request via the *hit* or *ready* lines. These cause the insertion of cache miss or wait states respectively. Also within the first cycle of either type of reference, the page fault line can be raised to indicate that the reference cannot be satisfied. In this case, the pipeline is flushed in a manner that allows it to be restarted. If the *error* signal is

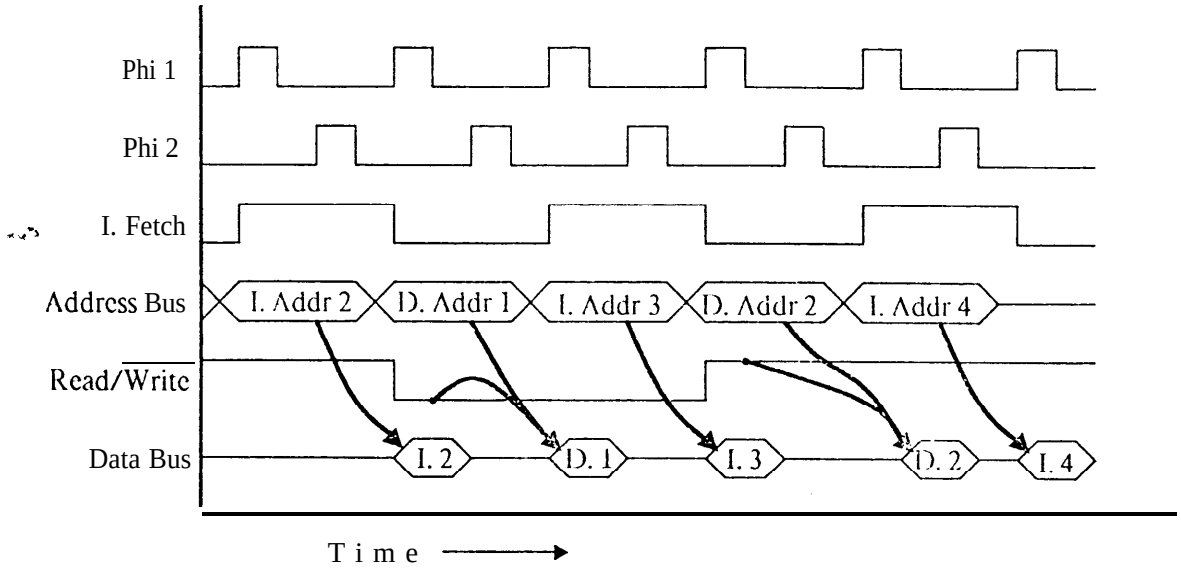


Figure 7: Bus timing diagram

Another consequence of the synchronous pipelined and fixed resource allocation is that one can determine at instruction decode time whether any particular instruction is going to use its data memory reference. The free memory cycles of instructions that do not use this particular resource would normally go unused. However, with MIPS the rest of the system is warned of these upcoming free cycles, so that memory can be kept busy doing cache prefetches, write-backs, I/O or other DMA activity. Thus the system can make the most of the peak memory bandwidth provided. This feature is significant: when running optimized code, over fifty percent of the data memory cycles are typically unused by the processor, due to the high percentage of register-register operations.

3.5 The decomposition of control and decode

Most microcoded machines and hardwired CPUs use a collection of logic and microcode that simultaneously decodes the instructions, interacts with other components of the system, and deals with the exception handling. In MIPS however, the instruction set and pipeline framework allow for the very clean decomposition of these functions into two distinct functional units: the Instruction Decode Unit (IDU) and the Master Pipeline Control (MPC). The main feature of the machine that allows this decomposition is the uniform nature of the instructions. Because all the instructions are essentially interchangeable with respect to their size, execution time, and possible use of resources, the specifics of the currently executing instructions are largely irrelevant to the handling of exceptions. The communication between the MPC and IDU is limited to status signals and information regarding memory use by the executing instruction, used by the MPC to drive the memory interface. This clean decomposition allows separate implementation and optimization of the two control components of the processor.

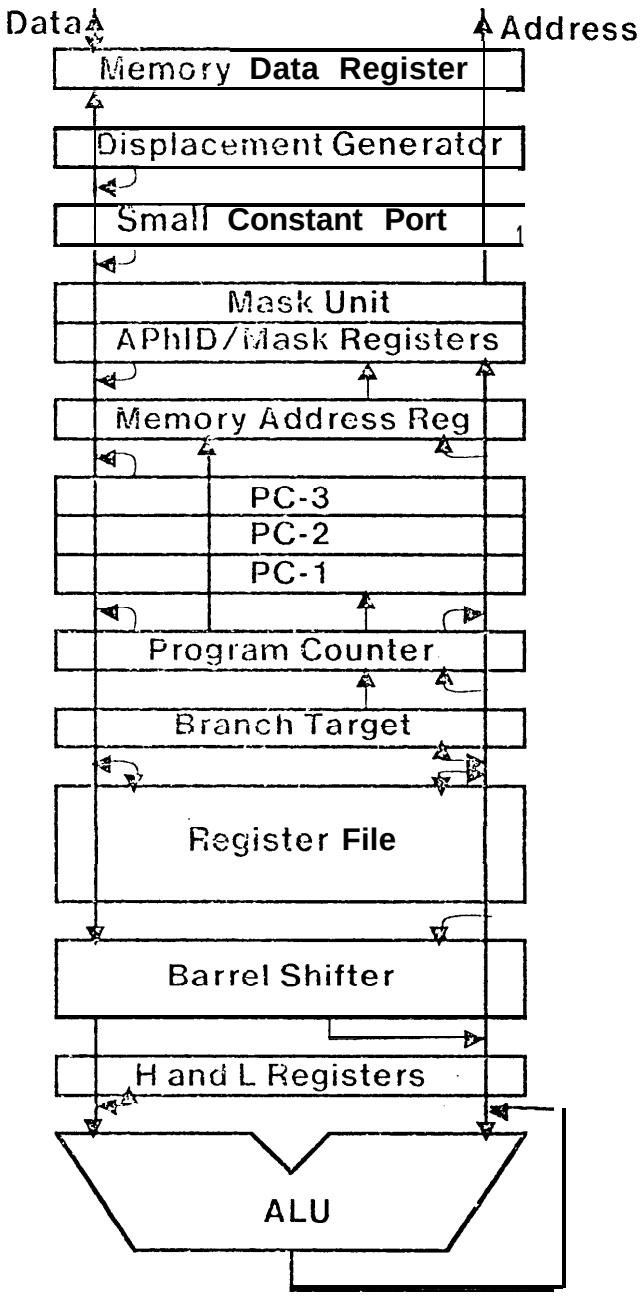


Figure 8: Datapath block diagram

Busses in MOS circuits provide a special challenge for the circuit designer because

Bus Delay = k X (Capacitance X Voltage Swing)iDrivcr Size.

For busses with many drivers, the bus loading is dominated by the drivers themselves, so

Bus Capacitance = k X Driver Size X Number of Drivers.

As a result,

Bus Delay = k X Number of Drivers X Voltage Swing

The delay could be improved by either reducing the number of drivers (smaller register file) or by reducing the voltage swing. The voltage swing may be reduced by using sense amplifiers, as is typically done in RAM designs. An alternative version of MIPS has been fabricated using a clamped bus structure that is similar to a sense amplifier with a limited swing bus. This alternative bus design reduces the effective voltage swing by about a factor of four, decreasing the bus delay to around 10nS and eliminating the precharge clock phases.

3.6.2 Data path components

Arithmetic operations in the critical path require careful logic and circuit design. The arithmetic and logical function blocks are separated to ensure minimal loading on the adder. The adder has a full carry-lookahead tree, with propagate signals and generate signals produced for each pair of bits, yielding a total ALU delay of 80ns. Adequate support for integer multiplication and division are an important MIPS objective. Special "H" (High) and "L" (Low) registers integrated into the ALU permit modified Booth's algorithm multiplication at a rate of two bits per ALU operation (four bits per instruction with packing) and non-restoring division at a rate of one bit per ALU operation (two bits per instruction with packing). The ALU has been extended to 34 bits to accommodate overflow detection for this two-bit-shift-and-add operation.

The barrel shifter is used for arithmetic and logical shifts, rotates and character insertion and extraction. This variety of functions is controlled by an input multiplexer that selects the data for each word of a two-word combined rotator. The shift amount determines which 32 bit section from this 64 bit combined quantity goes to the output. The combined rotator is implemented as a pair of cascaded shifters: the first shifts by the shift amount divided by 4, the second by the shift amount modulo 4. Severe pitch constraints dictated this special shifter organization. The barrel shifter lies between the registers and the ALU. During ALU operations, both operands must be transferred through the shifter on phase one, and the result must return on phase two. The tight 33λ pitch prohibits two bidirectional busses. One operand and the result travel on one bidirectional bus. The other source operand bypasses the barrel shifter in a special null operation.

The high resource utilization of the MIPS pipeline places severe demands on the register file. Any cell may be read onto either bus on either phase, and may be written from either bus on phase two as shown in Figure 9. Register refresh occurs whenever the cell is not written. This permits faster register write timing but complicates register decoding and expands the area needed for control line drivers. The register array contains both the sixteen general purpose registers and that portion of the process status register that holds the trap code.

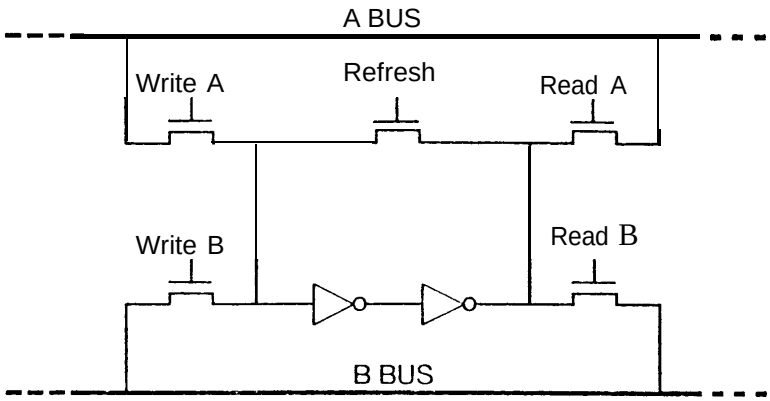


Figure 9: Register cell design

The variety of program counter operations and the requirement for instruction restart complicate the logical design and layout of the program counter block. The program counter must hold the current value, three previous values for pipeline restart and one possible future value for branching. On every cycle, one of six possible sources must be selected for the new value: increment, self-refresh, zero (to start interrupt response), the branch value, or values of either of the data busses. Simultaneously, the old value must be shifted into the FIFO buffer containing old values. The PC-incrementer could easily become the major performance bottleneck, but a simple carry-lookahead incrementer overcomes this problem.

The address masking primitives are integrated into the program counter structure along with the Memory Address Register. This masking allows a machine address to be converted to a process virtual address. The size of the process virtual address space is defined by a bit mask in a special register. When masking is enabled for normal operations, the high order bits of the machine address are replaced by a Process Identifier from another special register. These two special registers are accessible only to processes running in supervisor state. The Address Masking Unit also detects attempts to access outside the legal segment and raises an exception to the master pipeline control.

3.6.3 Control bus design

The control bus encodes the register transfer source and destination for each type of operation, as well as a code for the operation itself. It includes a set of true/complement pairs for the two ALU sources, the ALU destination and the memory transfer source/destination. It also includes a collection of special signals including pipestage, branch condition results, and various exceptional conditions.

Each datapath control driver taps this bus with one or more NOR decoders and latches the decoded signals into the driver in the phase immediately preceding its active use in the datapath. These signals are driven via

bootstrap drivers by the appropriate clock as shown in Figure 10. The bootstrap drivers put a considerable load on the clocks, but much care has been taken to minimize skew by conservative routing in metal. In a few cases bootstrap drivers cannot be used because the control signal may need to be active on both clock phases, as in register reads. These drivers are implemented as large static superbuffers.

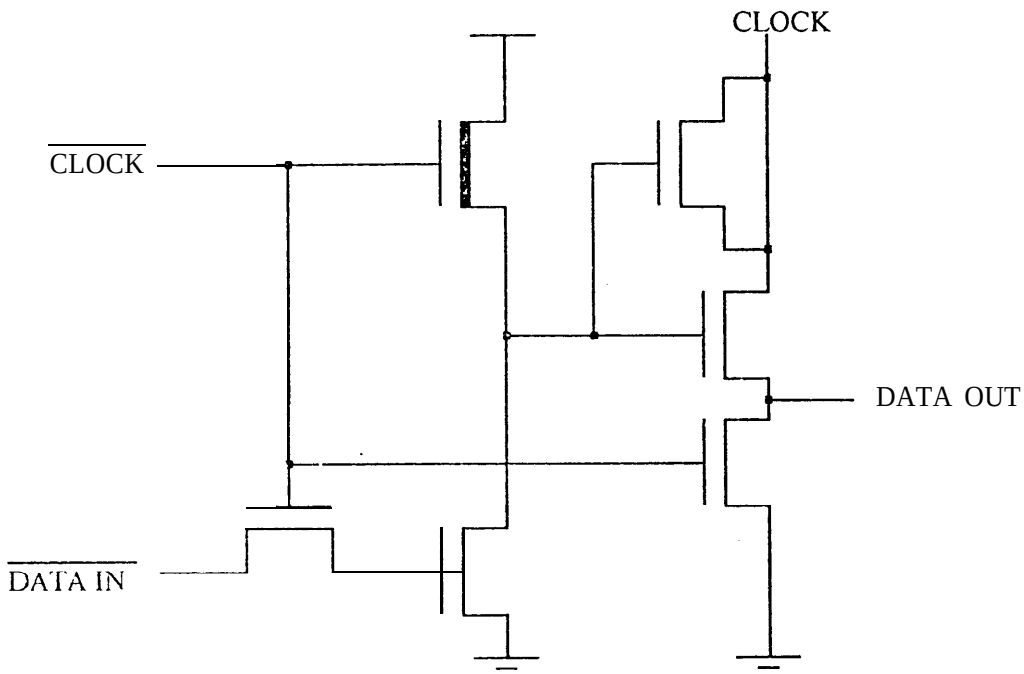
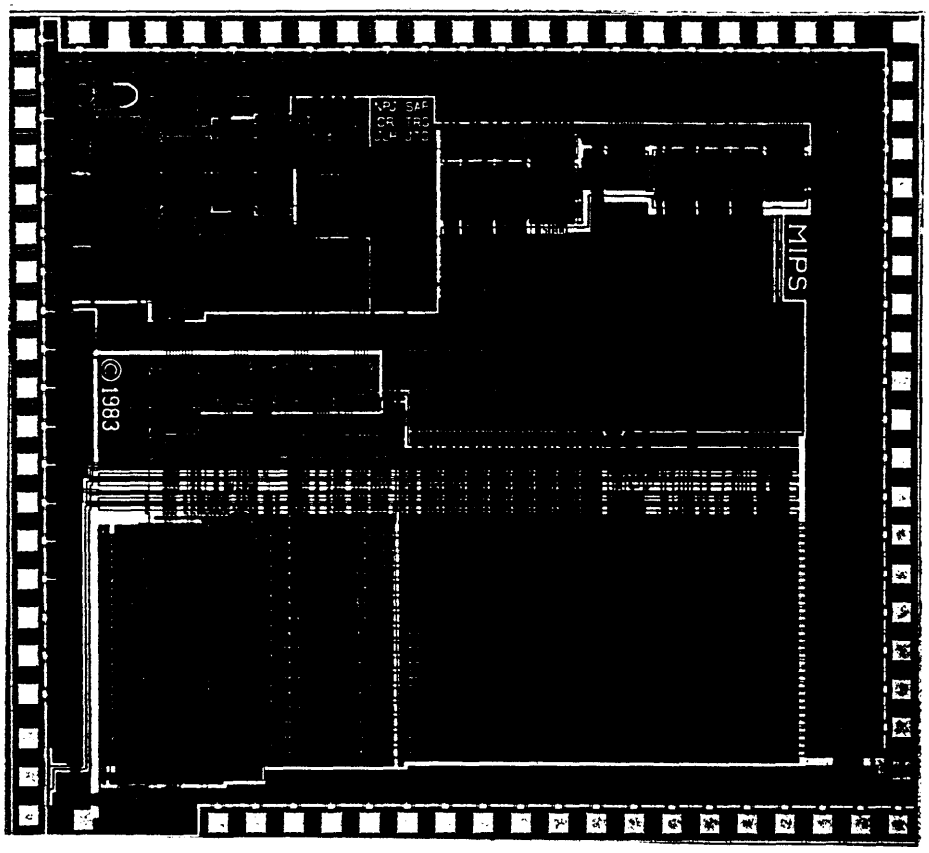


Figure 10: Dynamic bootstrap driver design

3.6.4 The package constraint

MIPS is housed in an eighty-four pin chip carrier. This package imposes two constraints on our VLSI implementation: a careful allocation of pins in the external interface and a power budget of roughly two Watts. Separate address and data pins are crucial to achieving a high memory bandwidth. A full assortment of control and status pins are necessary to support our realistic view of the computing environment as a world of faults and exceptions. The breakdown of pad usage is as follows:

Address	24 pins
Data	32 pins
Status out	8 pins
Status in	7 pins



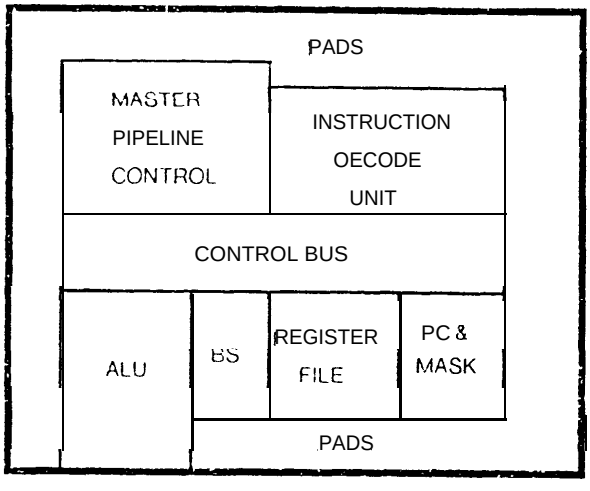


Figure 12: M IPS photomicrograph nnd floorplan

Regularity is the number of instantiated rectangles per drawn rectangle, which we computed using cifstat [4]. The register file has the highest regularity because it is the only part of the chip replicated in two dimensions except for the program counter stack. The control bus with decoders and the MPC have the lowest regularity for two reasons. First, they are the most random parts of the chip. But more importantly, they and the periphery were laid out by SIL, a silicon assembly language. Although at the SIL level much structure exists, the structure is largely lost in compiling to CIF.

Clocks, power, substrate	9 pins
LSSD	4 pins
Total	84 pins

The eighty-four pin chip carrier also imposes a power limit of about two watts. The fabricated processor draws less than 350mA (1.75W). Using a conservative estimate of power for each square of pullup, the device power requirements may be decomposed as shown in Figure 11.

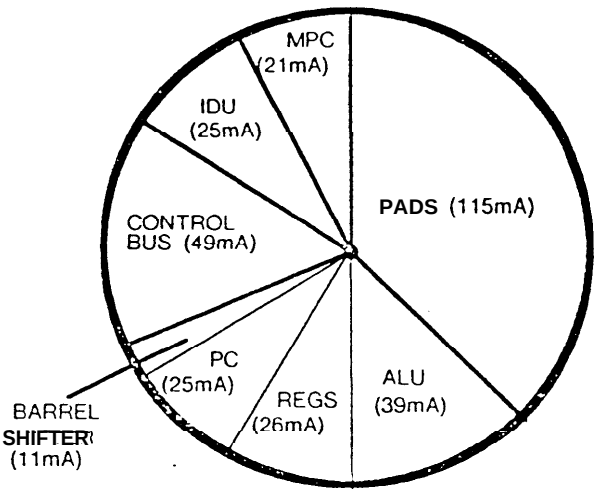


Figure 11: MIPS current distribution

Figure 12 shows the floor-plan and a photomicrograph of the MIPS chip. The chip is implemented in standard, one-level metal nMOS using Mead-Conway design rules with buried contacts. The total dimensions are 3750λ by 4220h.

3.7 Design metrics and timing characteristics

Table 3 presents several design metrics for MIPS. The datapath and periphery of the chip each occupy about 40% of the total chip are. The remaining 20% is devoted to control. Considering the functionality of the control structure (pipclining, exception handling, and instruction decoding), 20% of the chip area is a small price. The transistor density (i.e., ratio of percent of total transistors to percent of total area) is greater than two to one for all units in the data path except the bar-r-cl shifter and the data path control. Thcsc two units arc limited by communication, not by transistor sites. The MPC has the lowest transistor to area ratio of any part except for the pads. The control section as a whole (including the decoders for the data path) have an average transistor density of about one.

The role of the Master Pipeline Control is to sequence the machine, interface with the outside world, and to deal with all the exceptions. For the most part it is entirely unaware of the characteristics and current operations of the data path. It consists primarily of a sixteen state finite-state machine that incorporates the responses to the nine basic exceptions as well as the normal sequential operation of the machine. This state machine deals with cache misses, extended memory cycles, and DMA cycles; it also arbitrates among multiple faults and ensures that flushing of the pipeline is done properly.

Once the instruction decode unit is pruned of these subsidiary tasks, the IDU can concern itself solely with mapping instructions into control for the data path. In MIPS, the instruction set is encoded in such a way that the decoding of instructions is primarily just multiplexing the appropriate nibbles of the instructions onto the control word fields. This multiplexing is essentially a parallel operation with a small number of gate delays from the input register to the data path control drivers. The careful selection of the location of the various fields within the different instruction classes enables the multiplexing hardware to be shared. This further reduces the amount of logic required to perform the decoding.

3.6 Datapath implementation

The MIPS datapath consists of five principal functional blocks: the arithmetic logic unit, the barrel shifter, the register file, the program counter and the address mask unit. These resources are interconnected through a pair of thirty-two bit global buses and numerous local paths, as shown in Figure 8. The datapath communicates with the remainder of the MIPS system through the address and data ports and the datapath control bus.

3.6.1 Data bus design

The machine cycle time is closely linked to the time needed to move data from one resource to another in the datapath. This delay consists of the control propagation on the polysilicon control wires and the data propagation on the metal busses. The control delay is reduced by strict adherence to a 33X pitch in the datapath and by the use of bootstrapped control drivers. This combination holds the control delay to 10ns, which is limited primarily by the parasitic RC time constant.

Two different bus structures were examined. The first was a precharged bus that preset to the high state by a separate precharge clock during the period between each principal clock phase. To prevent spurious bus discharge, data must be valid as soon as the precharge clock falls. This precharge clock may overlap slightly with the beginning of the following clock phase. This clock overlap introduces momentary contention over the bus, but obviates a clock skew delay between the precharge and principal clock. The precharged bus method introduces a delay of about 40ns, in addition to the precharge time.

Given two circuits having the same number of levels of logic and similar power dissipation, the faster one will be the one with the smaller area. Parts of the design forced to have larger area due to communication topologies can have this disadvantage removed by increasing power consumption. Thus, we might expect to see power consumption per area to be roughly constant throughout the chip. Table 3 shows that power dissipation varies by a factor of about three around the chip. However, the most extreme figures for power per area are mainly determined by number of levels of logic. The ALU has many levels of logic, and hence was designed to dissipate a larger share of power to overcome this. On the other hand, the operation of the barrel shifter involves only a small number of levels of logic, so its power dissipation is low. The power ratio (or microamperes consumed per transistor) confirms that power is used to overcome multilevel gate delays and communication delays; this ratio is highest in the control bus (which does chip wide communication), the ALU carry generation logic, and the pads (which do inter-chip communication).

Part	% Area	% Trans	% Power	μA/Trans	Regularity	Power / Area	Trans / Area
Data Path:	**39.2**	**73.5**	**49.2**	8.6	**14.6**	1.3	**1.9**
ALU	6.3	13.7	12.3	11.5	20.7	2.0	**2.2**
Shifter	5.8	8.6	3.5	**5.2**	18.3	**0.6**	**1.5**
Registers	8.3	21.3	a.2	4.8	76.8	1.0	**2.7**
PC	6.1	14.7	7.9	6.9	25.0	1.3	**2.4**
Disp&Buf	1.7	3.6	1.9	6.2	12.7	1.1	2.1
Decode	11.0	11.1	15.5	17.8	4.6	1.4	1.0
Control:	**18.2**	**21.7**	**14.5**	8.5	**11.2**	0.8	1.2
MPC	9.9	9.2	6.6	9.2	4.9	0.7	0.9
IDU	8.3	12.6	7.9	8.0	20.1	1.0	1.5
Periphery:	**42.6**	**4.8**	**36.3**	**98.0**	9.2	0.9	**0.1**
Total	14.8Ml	24661	1.6W	**12.8**	**12.0**	1	1

Table 3: MIPS design metrics

Figures 13 and 14 are plots of the five most critical paths on clock phases one and two. These plots are similarly oriented to the floorplan and photograph of the chip (Figure 12). Each jog in the plot denotes a level of logic: ends of paths are capped with a bar signifying a latch. Timing results for conservative, worst case 3μm SPICE parameters indicate a clock cycle time of 250ns. giving a machine cycle of 500 nS. Typical SPICE parameters yield clock cycle times on the order of 166ns. In subsequent discussions, all timing results will be for the worst case parameters.

Noticeably absent from the critical paths is the entire data path. Without the reduced voltage swing bus, the data path busses become part of the critical path. The improved bus design yields a total speedup of

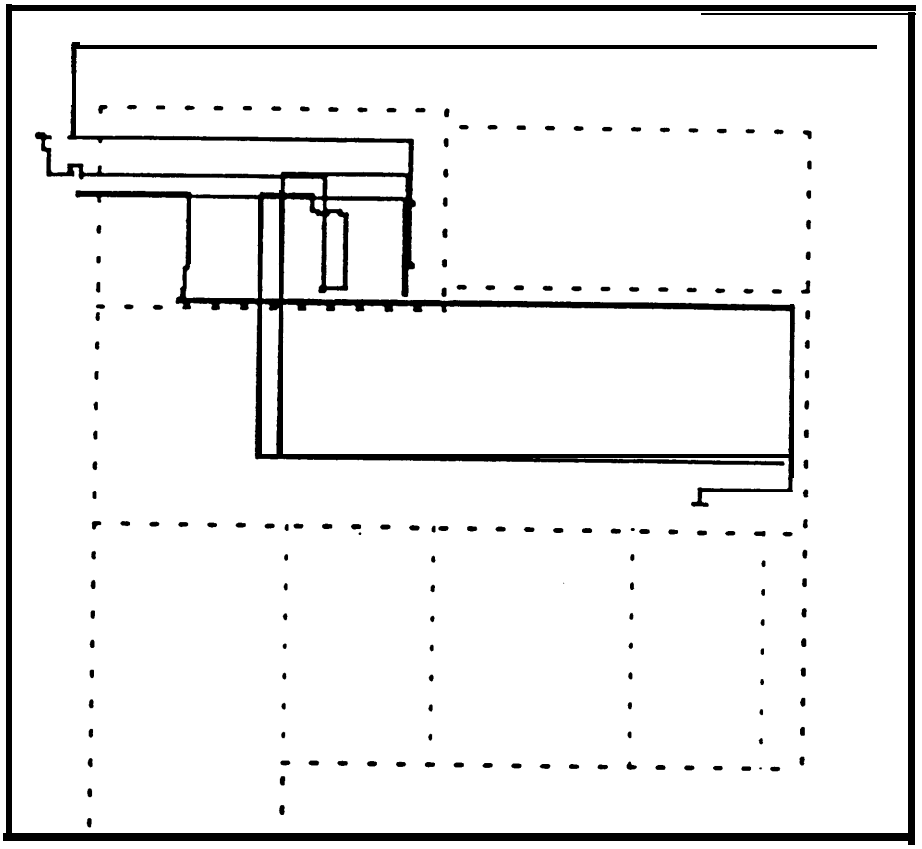


Figure 13: Critical paths on phase **one**

about 150ns, which removes the busses from the critical timing path. The regular structure of the data path also made it easier to quantify its delays and minimize them during design.

The bulk of the critical paths heavily involve the control component of the chip. Most of the delays occur in the MPC and the control decoders. There are several reasons for this:

1. Properly handling ten different exceptional conditions in a truly pipelined machine (i.e., a pipeline with more than instruction prefctch) is costly.

2. The complexity of the control prevented easy understanding of delays and potential critical paths at design time [8].

3. Scvcral logic structures used (e.g., PLAs and large fan-in NOR gates in the control bus) have inherently slow performance.

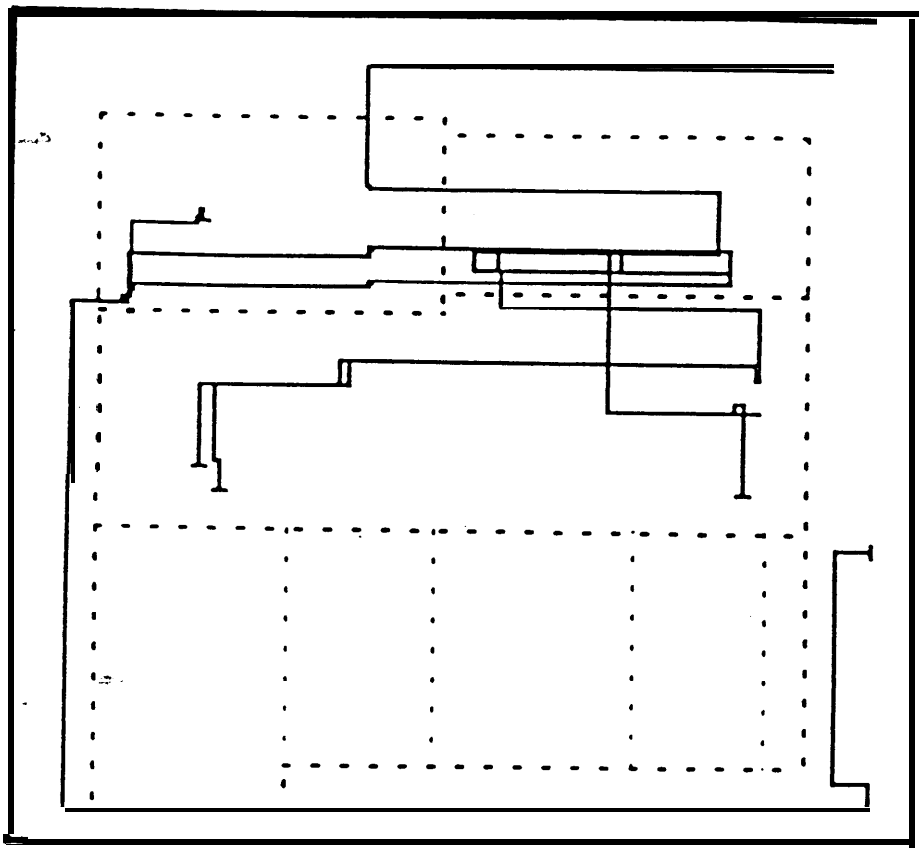


Figure 14: Critical paths on phase two

Not much can be done to alleviate the first cause. It is a price one must pay for the increased performance derived from pipelining and building a processor that handles the conditions arising in the system's environment. The second cause could bc eliminated to a large extent by timing analysis on a schematics level before layout. The last cause is difficult but not impossible to avoid. Our control synthesis tool, SLIM, IS very useful but currently only gcncratcs PLAs, which for some structures can bc inefficient. Finally, large fan-in NOR gates are inherent in many functions, and only better circuit design techniques can help reduce their delay. Although power dissipation per area is the lowest in the MPC, it is difficult to power up structures such as PLAs, since most of the delay comes from pull-down capacitance on the large fan-in NOR gates.

3.8 Timing methodology

Some of the advanced circuit structures used in MIPS represent a deviation from standard Mead-Conway design styles [17]. Another important deviation is the use of a timing-dependent clocking methodology instead of a timing-independent approach. Like the high performance circuit structures, this methodology can yield higher clock speeds.

Figure 15 is a typical circuit from MIPS which relies on verification through a timing-dependent clocking methodology, When phase one falls, the data is latched on the storage node and the pull-up raises node N. If the pull-down is disabled long enough before the latch is closed for node N to rise to a logic 1 (e.g. due to clock skew), the circuit will malfunction. A timing dependent verifier was built into TV [13, 12] to ensure correct design of such circuits.

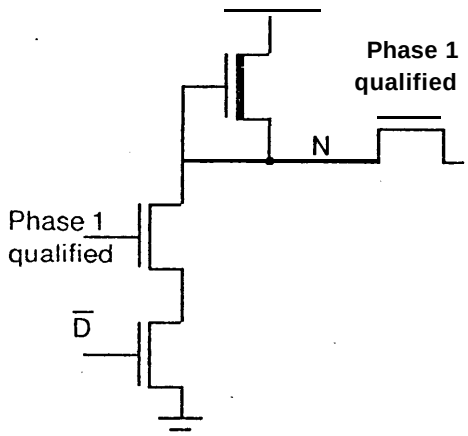


Figure 15: Timing-dependent clocking

Although not specifically prohibited in Mead-Conway designs, other stricter two-phase timing disciplines [19] allow signals to pass through only one pass transistor qualified by a clock during each clock phase. In MIPS, most paths pass through several transistors gated by the active clock phase. Without the ability to pass through several clock-qualified pass transistors during each clock phase, our design would become much more complicated and significantly slower. Since we did not design with this methodology, accurate estimates are not available, but we suspect that such a methodology could negatively impact performance by as much as a factor of two.

A significant difference between nMOS and bipolar circuits that can be exploited is the type of latches used In bipolar circuits and MOS circuits using master-slave, edge-triggered flip-flops, signals must be stable at input to the latch at a single *point* in time. At this point they are transferred to the output of the flip-flop. In MOS technologies using simple pass transistor latches, the signal may propagate through the latch during the *interval* when the clock is high. If a critical path only drives non-critical paths, and is driven only by

non-critical paths, time may be "borrowed" from adjacent phases to reduce the effective time of the critical path.

Figure 16 illustrates this principle. The thin lines represent paths consisting of an arbitrary amount of logic; the thick vertical lines represent latches. The time to the left of each latch is the arrival time of the signal after the rise of the other clock. Note that one path travels through two latches on phase 2. Without borrowing, the cycle time is simply:

$$T_{cycle} = \text{longest phase 1 time} + \text{longest phase 2 time}$$
$$= 150\ nS + 120\ nS$$
$$= 270\ nS$$

However, with borrowing the cycle time is actually 220ns, phase 1 being 120 nS and phase 2 being 100 nS. This is the longest time required for two paths active on opposite clock phases in series. Borrowing may only occur while a clock is high; for chips that require clocks of small duty cycle, borrowing may be limited by the high time of the clocks.

In MIPS, where one clock or another is high for about 90% of the cycle, possibilities for borrowing abound Restrictions on borrowing are imposed by the possibility of destroying charge or bootstrap effects. For example, borrowing is not possible for the signals that select a write destination in the register file, or for the pull-downs discharging a precharged bus. These signals must be stable at the rise of a clock phase or else data will be lost. Similarly, data must be present at bootstrap drivers at the rise of the clock phase or else the bootstrap effect may be lost. Since the current MIPS implementation does not use precharging, and bootstrap drivers are only used to gate storage nodes, significant benefits can be attained from borrowing. TV incorporates a borrowing algorithm in its analysis and shows that borrowing saves about 35% of the current clock cycle.

4 Architectural Performance Measurement

The MIPS processor derives its high performance from a combination of fast hardware and an efficient match between the processor architecture and software. The MIPS architecture exploits advances in software technology more successfully than most other processors. The small set of simple instructions aids efficient code generation, and the orthogonal register file permits global register allocation to reduce memory traffic. The exposed pipeline allows the software code reorganizer to achieve high hardware resource utilization.

The Pascal and Fortran compilers are built on top of the UCode system, which contains the standard phases of parsing, intermediate code generation, global optimization, register allocation and code generation. The optimizations, performed by UOPT [3], include common subexpression elimination, code motion, induction variable elimination, constant folding, dead code elimination, and redundant store removal. Register

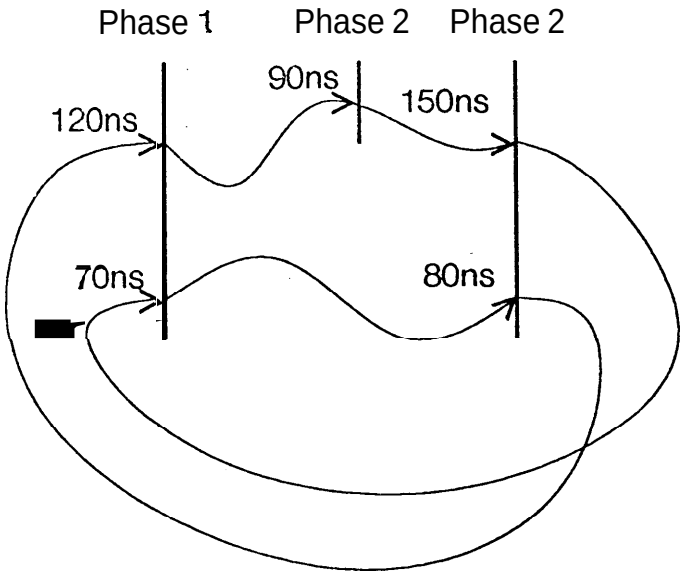


Figure 16: Borrowing time from less critical phases

allocation is done by a priority-based, graph coloring algorithm [2] that moves most local scalar variables into the register file and requires minimal spill code. The code generator produces a stream of simple operations, independent of all resource interaction, branch delays and instruction combinations. The code reorganizer performs three crucial functions. It assembles and packs one or two independent operations into each 32-bit instruction word. It moves appropriate instructions into the words following each delayed branch, and it reorders instructions within each basic block to eliminate resource conflicts [6]. Compared to simple no-op insertion to eliminate resource interaction, the reorder operation cuts execution time by 5%. Reordering combined with instruction packing saves 10%. The complete machine level optimization done by the reorganizer (reordering, packing and branch optimization) saves a total of 30% of the execution time. In addition. shifting the burden for these features to the software yields a simplification of the pipeline hardware and allows it to run faster. We estimate that a 20% improvement in the clock cycle has been achieved due to these simplifications.

4.1 Software Simulation

The MIPS software system includes compilers for Pascal, C, and FORTRAN. MIPS has been extensively characterized using Pascal benchmark programs. These benchmarks include a variety of short compute-bound programs exercising all aspects of the MIPS instruction set. The majority involve non-numeric manipulation of arrays, lists and scalars. Many are recursive and contain numerous procedure calls. The Pascal benchmarks include:

- Puzzle: Solves 3D cube packing problem

- **Queen**: Solves eight queens chess problem
- **Perm**: Computes all permutations of one through seven
- **Towers**: Solves Towers of Hanoi problem for 14 discs
- **Intmm**: Multiplies two 40x40 matrices
- Bubble: Bubble sort of 500 integers
- Quick: QuickSort of5000 integers
- Tree: Tree insertion sort of 5000 integers

All benchmarks were assembled with the Stanford UCode Pascal compiler with global optimization and checking turned off. Because the compilers use the same front-end and code optimizer, the compiler technology does not bias the performance data. All MIPS benchmarks were processed through the code reorganizer/assembler and then passed to the instruction set simulator to gather performance data. This simulator detects any erroneous instruction interaction and reports detailed statistics on instruction profiles. It makes no allowance for events outside the CPU chip, such as cache misses or page faults; both of these effects should be small because of the size of the benchmarks.

The results of the Pascal benchmarks are shown in Table 4 for MIPS (4MHz), the Motorola 68000 (8MHz) and the Digital DEC20/60.

	MIPS	68000	DEC20/60
Puzzle	2.40	6.1	2.6
Queen	.44	1.9	.5
Perm	.56	3.0	.6
Towers	.64	2.9	.7
Intmm	.80	5.0	.5
Bubble	.58	3.7	.7
Quick	.41	2.6	.4
Tree	1.01	9.9	1.0
Avg. (relative to MIPS):	1.00	5.1	1.0

Table 4: Pascal benchmark performance (in seconds)

It is interesting to examine where the factor of six in improvement over the Motorola 68000 comes from. Our examination of the data has led to the estimates shown in Table 5.

Concept	Benefits	Imorovement
Streamlined instruction set	lower overhead/instruction single cycle decode	1.5
MIPS Organization	Longer pipeline Higher memory bandwidth	1.5
Low level instruction set	Machine level optimization Streamlined pipeline	2.0
Good compiler architecture match	Better utilization of hardware resources	1.2 - 1.5
Total	High performance	5.2 - 6.6

Table 5: Breakdown of performance improvement

5 Conclusions

The MIPS design attempts to eliminate the artificial barriers between the processor architecture its organization, and its implcmcntation. This synergistic approach results in hardware simplifications and in increased performance. We have tried to consistently evaluate the needs for a particular architectural capability and to dctcrminc where and how it is best implcmcnted. For cxamplc, by including the pipeline organization of the processor as part of the architccturc, we gain two important benefits:

1. The compiler is able to see and optimize effects that arc usually hidden but that can have substantial performance impact.

2. Because this function can bc performed predominantly in software without incurring any significant penalty, the hardware can bc made simpler and hence faster.

Reducing the instruction set to the point where it can be exocutcd without microcode removes a level of interpretation and thus speeds up execution. Additional benefits come from the ability to decode instructions in a single cycle and to keep the pipeline full most of the time.

Although the MIPS performance improvements come largely from its architccturc and processor organization, a well designed implementation is necessary to achieve a high performance nMOS chip. Without careful circuit design and a tight timing strategy we could lose a factor of three to four in pcrformancc. The VLSI implcmcntation medium stresses the need for simplicity in designs and has a fundamental impact on the metrics used to compare alternative organizations. Because the processor design balances implcmcntation cost (in dollars and performancc) against functionality. the architect must clearly understand the interaction between the architecture, the organization, and the implcmcntation.

Acknowledgements

WC are indebitcd to John Shott and the Stanford Integrated Circuits Laboratory for fabricating MIPS and assisting with testing efforts. The DARPA MOSIS service has provided fabrication runs throughout the project. Paul Losleben of DARPA has been an enthusiastic supporter of the project. John Gill, Forest Baskett, and Jud Leonard have also made valuable contributions.

References

1. Anderson, D.W., Sparacio, F.J., and Tomasulo, R.M. "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling." *IBM Journal of Research and Development 11*, 1 (January 1967), 8-24.

2. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W. Register Allocation by Coloring. Research Report 8375, IBM Watson Research Center, 1981.

3. Chow, F. *A Portable, Machine-independent Global Optimizer – Design and Measurments.* Ph.D. Th., Stanford University, 1984.

4. Fitzpatrick, D. *Cifstat.* University of California, Berkeley, Berkeley, California, 1783.

5. *DEC VAX11 Architecture Handbook.* Digital Equipment Corp.. Maynard, MA., 1779.

6. Gross, T. *Code Optimization of Pipeline Constraints.* Ph.D. Th., Stanford University, September 1983.

7. Hennessy. J.L. and Gross, T.R. "Postpass Code Optimization of Pipeline Constraints." *ACM Trans. on Programming Languages and Systems 5, 3* (July 1983), 422 -448.

8. Hennessy, J., Jouppi, N.. Przybylski, S., Rowen, C. and Gross, T. Design of a High Performance VLSI Processor. Proc. of the Third Caltech Conf. on VLSI, Calif Institute of Technology, Pasadena, Ca., March, 1983, pp. 33-54.

9. Hennessy, J.L., Jouppi, N., Baskett, F., Gross, T.R., and Gill, J. Hardware/Software Tradeoffs for Increased Performance. Proc. SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems, ACM, Palo Alto, March, 1982, pp. 2 - 11.

10. Hopkins, M. A Perspective on Microcode. Proceedings of COMPCON Spring 83, IEEE, San Francisco, March, 1983, pp. 108-110.

11. Hopkins, M. Compiling High-Level Functions on Low-Level Machines. Proc. Int. Conf. on Computer Design, IEEE, Port Chester, N.Y.,, October, 1983.

12. Jouppi, N. TV: An nMOS Timing Analyzer. Proceedings 3rd CalTech Conference on VLSI, California Institute of Technology, March, 1983, pp. 71-85.

13. Jouppi, N. Timing Analysis for NMOS VLSI. Proc. 20th Design Automation Conference, IEEE, June, 1983, pp. 411-418.

14. Kung, H. T. and Leiserson. C. E. Algorithms for VLSI Processor Arrays. In Mead, C. A. and Conway, L., *Introduction to VLSI Systems*, Addison-Wesley, 1978.

15. Lampson, B. and Pier, K. A Processor for a High-Performance Personal Computer. Proc. Seventh Annual Symposium on Computer Architecture, IEEE/ACM, La Baule, France, May, 1980, pp. 146-160.

16. Louie, G., Ho, T.. and Cheng, E. "The MicroVAX I Data-path Chip." *VLSI Design 4, 8* (Dec. 1983), 14 - 21.

17. Mead, C. and Conway, L,.. *Introduction to VLSI Systems.* Addison-Wesley, Menlo Park, Ca., 1980.

18. *MC68000 Users Manual.* 2nd edition, Motorola Inc., 1780.

19. Noice, D., Mathews, R., Newkirk, J. A Clocking Discipline for Two-Phased Digital Systems. Proc. of ICCC, New York, September, 1982, pp. 108-111.

20. Patterson, D.A. and Ditzel, D.R. "The Case for the Reduced Instruction Set Computer." *Computer Architecture News 8, 6* (October 1980), 25 - 33.

21. Patterson, D.A. and Sequin C.W. RISC-I: A Reduced Instruction Set VLSI Computer. Proc. of the Eighth Annual Symposium on Computer Architecture, Minneapolis, Minn., May, 1981, pp. 443 - 457.

22. Radin, G. The 801 Minicomputer. Proc. SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems, ACM, Palo Alto, March, 1982, pp. 39 - 47.

23. Smith, J., and Goodman, J. A Study of Instruction Cache Organizations and Replacement Policies. Proc. of the Tenth Annual Symposium on Computer Architecture, ACM, Stockholm, June, 1983, pp. 132 - 137.