Design of a High Performance VLSI Processor

Article · January 1983
DOI: 10.1007/978-3-642-95432-0_3

5 authors, including:

John L. Hennessy
Stanford University
290 PUBLICATIONS **31,932** CITATIONS
SEE PROFILE

Norman P. Jouppi
Google Inc.
213 PUBLICATIONS **16,115** CITATIONS
SEE PROFILE

Design of a High Performance VLSI Processor

John L. Hennessy, Norman P. Jouppi, Steven Przybylski, Christopher Rowen and Thomas Gross
Computer Systems Laboratory
Stanford University
Stanford, California 94305

Abstract

Current VLSI fabrication technology makes it possible to design a 32-bit CPU on a single chip. However, to achieve high performance from that processor, the architecture and implementation must be carefully designed and tuned. The MIPS processor incorporates some new architectural ideas into a single-chip, nMOS implementation. Processor performance is obtained by the careful integration of the software (e.g., compilers), the architecture, and the hardware implementation. This integrated view also simplifies the design, making it practical to implement the processor at a university.

1 Introduction

MIPS [10] is a new 32-bit processor designed to execute compiled code for general purpose applications. MIPS is both a streamlined (or reduced) instruction set architecture and an implementation of that architecture as an nMOS chip. The MIPS implemetation has several goals. First, we wanted to determine whether a competitive 32-bit processor could be built using the concepts in the MIPS architecture and Mead-Conway style design. A number of benchmarks have been run on the 68000 (at 8MHz) and our MIPS simulator (assuming a clock period of 4MHz). These benchmarks show an average speed-up of 460% for C programs and 550% for Pascal programs using the same compiler technology for both processors. If we can build a 4MHz part, we will have succeeded in demonstrating the architecture and implementation.

Other experimental goals were just as important. We wanted to simplify the VLSI implementation as much as possible: at the heart of our strategy was to replace hardware with simple software wherever there was no significant performance advantage to the hardware implementation; specific examples of the types of tradeoffs we made are discussed in [11]. Last, we wanted to experience the problems of designing a large chip, to further focus our architectural and design aid research activities.

In this paper, we give an overview of the architectural design and VLSI implementation of MIPS. We will attempt to show how a set of new ideas in architecture was implemented in a VLSI processor, to yield a high performance CPU. We will discuss the limitations we encountered in our efforts to increase performance, the design principles that simplified our efforts and made a fairly ambitious design possible, and the problems we have yet to overcome before we embark upon another project of similar scope.

2 Architectural Design

One overall goal of the MIPS architecture design was to explore the extent to which instruction set design can be made into a scientific process. This can be done by evaluating the design choices on the basis of two data points: quantative data about the usefulness of an instruction and careful estimates of the hardware cost. For MIPS, a C compiler, derived from the Portable C compiler, and a simulator formed a measurement system that provides firm, realistic data. This provided a fundamental workbench to test out new instructions. The proposer of the new instruction had to get the compiler to generate the instruction, collect data for a number of benchmarks, and examine what instructions, if any, were displaced by the new instruction. These measures are then combined to give a reasonably accurate estimate of the potential of the new instruction. Unfortunately, determining the hardware cost of an instruction that does not fit into the model of other instructions in the architecture is at best an ad hoc process.

MIPS is a reduced or streamlined instruction set machine, like the Berkeley RISC processor [13] and the IBM 801 [15]. The fundamental philosophy of such architectures is to concentrate on the most frequently–used simple instructions and to build more complex instructions from a customized series of simpler instructions. Because the simpler instructions, e.g., simple loads and stores, arithmetic operations, etc., are used with much higher frequency than more complex instructions, eliminating the complex instructions will allow the architecture to optimize its performance for the simple instructions. Such a philosophy becomes even more applicable in a VLSI implementation, because power and area constraints force the designer to choose among fundamental alternatives.

We also chose to design a load-store architecture, i.e., a machine in which only the load and store operations access memory and all ALU instructions are register-register format. This structure is very compatible with a streamlined instruction set; it also simplifies the implementation of page faults, traps, and interrupts, which can be particularly difficult to implement in a pipelined machine. Lastly, given reasonable register allocation algorithms [1], using a register-based machine is a good match for VLSI technology. Since the cost of off-chip communication is high, getting data onto the chip and operating on it in the register set is a cost–effective strategy. An as alternative to the software register allocator, the RISC project provides the hardware for a register stack [13].

We observed, like the 801 project, that a simplified instruction set can expose all internal machine cycles and states at the instruction set level. In contrast, a more complex machine hides many internal cycles within a single instruction. With modern compiler technology, exposing all machine operations and making them subject to optimization provides obvious benefits. The simpler instructions also have consistent running times so that the code generator can more easily choose between candidate implementations of a function.

In MIPS, we decided to attempt to expose all the internal implementation details affecting performance. Primary among these implementation issues is the pipeline structure. In choosing to expose this structure, we also decided that the synchronization function performed by pipeline interlocks could be moved into the software thus substantially simplifying of the hardware and losing little or no performance [9]. Therefore, the MIPS hardware architecture was defined without pipeline interlocks and with delayed branches (see 4.2).

Performance in executing a single sequential instruction stream comes from the ability to execute portions of that stream in parallel. This parallelism can be obtained from both pipelining and using of multiple function units. The internal micromachine of a pipelined CPU is inherently parallel. One goal that we imposed was to exploit internal microengine parallelism, to the degree that it makes sense, by projecting it into the instruction set. VLSI technology reinforces this goal; on-chip parallelism is cheap compared to off-chip, sequential communication. Thus, the MIPS instruction set is an attempt to carefully blend the

instruction set requirements with the capabilities of a high-performance, pipelined, VLSI microengine.

Many of our goals required that implementation-dependent features, such as the pipeline length and the parallel activity within instructions, be visible in the instruction set. This can lead to significant complexities in the user's view of the machine and in the construction of code generators. It also makes it difficult to alter the instruction set design specification during the actual hardware design, which we wanted to be free to do. Our implementation dependent optimization includes three primary code improvements:

1. reordering instructions to avoid pipeline dependencies,
2. reordering instructions so that the instruction following a branch can always be executed, and
3. packing together separate, simple instructions (called instruction pieces) into a single parallel instruction word.

These optimizations, similar to those performed by a microcode compactor [3, 6], can be performed with a high degree of success in the final phase of assembling instructions. Thus, we decided to define the instruction set at two different levels. The first level, called the user-level or assembly language instruction set, defines instructions that are unpacked and have no pipeline dependencies or branch delays. The assembly language instruction set is comprehensible and easy to generate code for. The machine-level instruction set is the low level instruction set actually run by the processor. This instruction set is generated only by a single program, the *reorganizer*. The reorganizer does all implementation-dependent optimization, and isolates the user level instruction set from these implementation details.

A related goal of the actual implementation process was to simplify and regularize the hardware wherever possible, subject to reasonable performance constraints. The unavoidable irregularities in the processor design often became software responsibilities. This shifting of responsibility included not only management of the pipeline but also:

- saving and restoring the pipe contents during a fault or interrupt,
- restoring operands after arithmetic overflows,
- specifying all program counter related operations (e.g. saving the PC on procedure call and relative jumps) so as to simplify the actual hardware implementation rather than conform to a predefined software model.

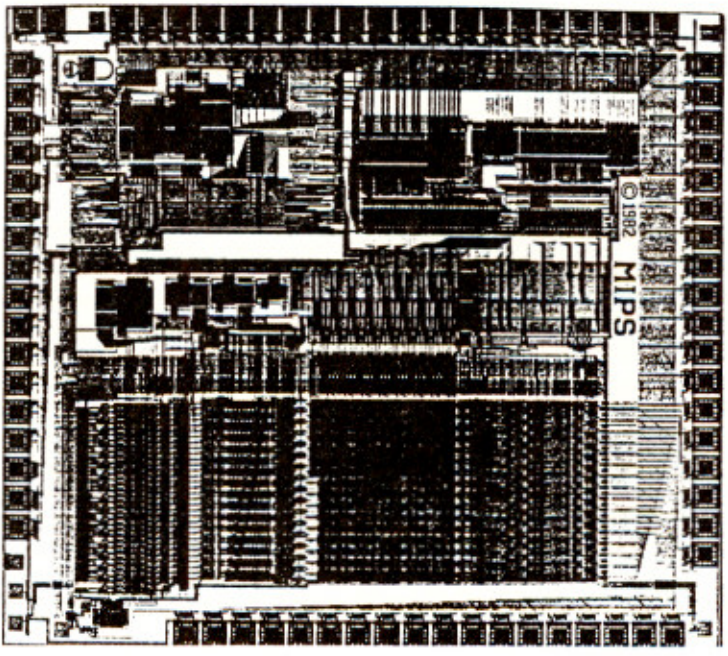
This division of the architectural definition into two parts and the freedom of giving the software responsibility for irregularities in the organization increased performance significantly. This strategy also simplified the processor design and thereby made the implementation of the MIPS chip possible with limited manpower and tools.

3 VLSI Implementation

The MIPS chip is a monolithic VLSI implementation of the MIPS processor architecture. The chip is implemented in standard, one-level metal nMOS using Mead-Conway design rules with buried contacts. The total dimensions are 3750λ by 4220λ. With λ=2μm, we expect to run with a clock period of 250ns (4 MHz clock); this will give an execution rate of two million instructions per second.

The MIPS chip consists of four logical blocks. The 32-bit data path contains two bidirectional buses linking a fast ALU, a barrel shifter, a file of 16 registers, a complex program counter, and an address masking unit. The data path is controlled by an encoded

control bus that distributes register transfer commands to the data path. The Instruction Decode Unit latches and interprets all the components of the instruction in parallel and drives the appropriate control information onto the control bus at the appropriate time. The Master Pipeline Control communicates with the outside world, responds to all internal and external exceptional conditions, and controls the basic sequencing of the pipeline. Figure 1 shows the major parts of the chip.



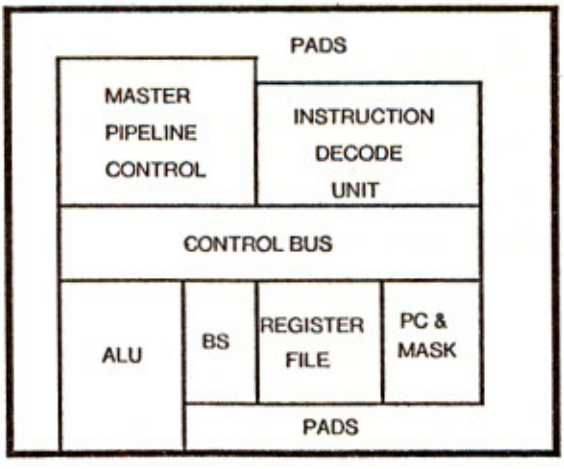


Figure 1: MIPS layout and floorplan

Maximum performance of the MIPS machine requires a high–bandwidth memory interface. MIPS memory is word-addressable and all memory references are 32 bits wide. Separate address and data pins permit partial overlapping of memory references, and also eliminate delays often incurred in multiplexing the address and data pins. To execute at full speed, the instruction access time is bounded by 250ns. A simple read-only instruction cache is possible, since the architecture forbids writes into the instruction stream. Data memory may be slower, with a maximum access time of 450ns.

The bulk of the implementation was done by three designers over the course of about 18 months. These designers were also involved in other significant efforts on design aids and architectural refinements. They were assisted occassionally by other logic and circuit designers. The total time for logic and circuit design and layout for the MIPS processor and the test chips has been approximately 2.3 man-years; the chronology and effort expenditures are given in Section 5.1.

3.1 Timing Methodology

The timing methodology for MIPS uses a two-phase, non-overlapping clock without precharging. Both the data bus and the control bus are potentially active during each clock phase of every cycle. The control bus is actively driven on both cycles by the Instruction Decode Unit. Each data bus writer incorporates bus pulldowns,but the bus itself includes the depletion pullups. Although precharging is normally a powerful tool, it does not mesh well with high bus utilization and overlapped execution of the MIPS pipeline. A simple non-overlapping precharge scheme would require a doubling of the clock rate and possibly an increase in clock skew overhead. A more intelligent precharge scheme with overlap between the precharge and the following bus transfer would be more difficult to design. Either precharge technique is susceptible to erroneous bus discharge. The MIPS method offers good performance and greater tolerance to timing errors, for the cycles can be extended to overcome bus discharges at the beginning of any transfer.

Similarly, the functional units, e.g. the ALU, are not precharged. Instead, the ALU amortizes its longer operation time as compared to the bus transfer time, by working across clock phases. During phase one, the ALU sources are driven from the register file to the ALU and the ALU begins operation; at the end of phase one, the ALU inputs are detached from the bus. During phase two, the ALU completes its cycle and drives the result out on the bus. Its effective execution period runs from the middle of phase one to the beginning of phase two, including the non-overlap period between the clock phases.

Each instruction passes through five pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Operand Decode (OD), Operand Store/Execution (SX), and Operand Fetch (OF). The pipeline is fully synchronous and contains three active instructions; thus, only two sets of stages can be active simultaneously: IF with OD and OF, or ID with SX. In the absence of unexpected external events, e.g., cache misses and page faults, the processor will simply toggle between two states. Each stage requires one full clock cycle, so each instruction completes in five cycles (really three pairs of stages). In Figure 2, the activities occurring for a sequence of three instructions are shown; we assume that all instructions are base-offset loads combined with ALU instructions.

3.2 Data path

The relatively complex and long data path requires metal data buses to achieve any reasonable level of performance. Polysilicon is relegated to the control lines crossing the width of the data path. Register transfers thus become dominated by two effects. First, high

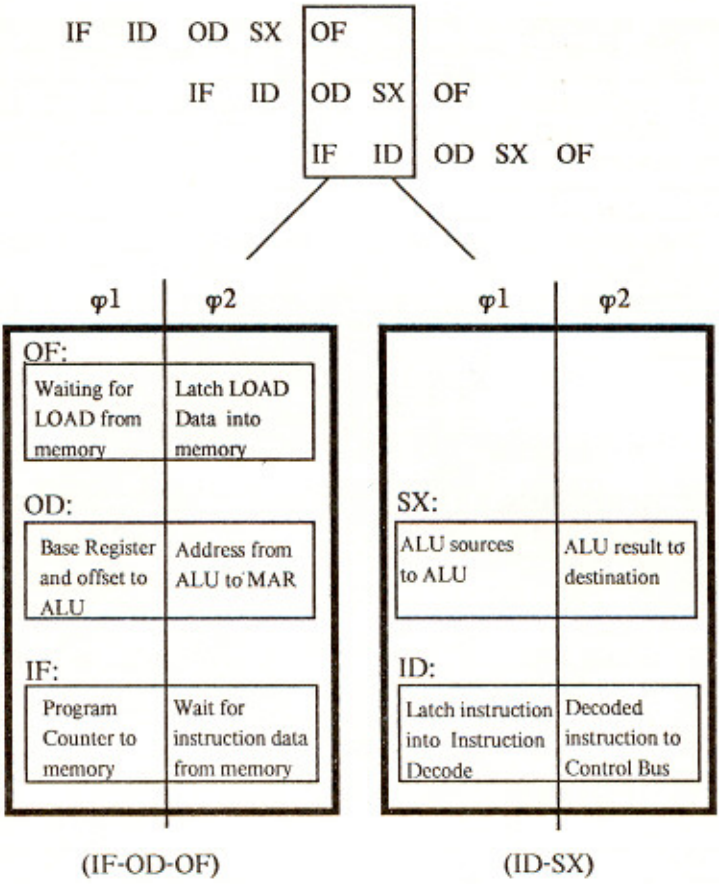


Figure 2: Activity in pipeline stages

polysilicon sheet resistance makes the control signal delay almost completely diffusion-limited; a narrow cell pitch is critical. We have achieved a pitch of 33λ with few performance compromises. Second, without precharging, the transfer of a selected value can be limited by the rise time of the bus. This potential delay is combatted by placing the driving pullups on the bus, rather than in the source data path cell. With the data and source select pulldowns in the cell, the bus then forms a distributed AND-OR-INVERT gate. When the bus becomes deselected between cycles, it rises to an intermediate or high level.

We recognized early in the design that the ALU lay in the critical path for many operations. We decided to devote considerable design time, area and power to maximizing the performance of this unit. The arithmetic and logical function blocks are separated to minimize loading on the adder. We implemented a carry-lookahead tree, with propagate and generate signals computed in pairs. This allows a total ALU add time of under 100ns. The ALU is also responsible for the detection of fourteen different conditions needed for conditional branch and set operations.

Much of the data path design was dictated by the need to optimize the add time. Since the ALU needed a fast carry lookahead and we decided to use a narrow pitch, the layout forced the ALU to be at the end of the data path so that the buses did not need to pass through the PG-tree in metal. This forced the barrel shifter into the middle of the data path.

Adequate support for multiplication and division is an important MIPS objective. However, full multiply and divide instructions are not consistent with our objective of single–cycle execution on all instructions, and the limited silicon area available. We implemented a modified Booth's algorithm for multiplication and division by way of a pair of special registers integrated into the ALU (H and L). These permit multiplication at a rate of two bits per ALU operation (four bits per full instruction) and division at a rate of one bit per ALU operation. A two-bit Booth multiply step requires both a shift operation and an add operation. To complete these within the same time slot as a standard register-register add requires that the H/L registers be closely integrated with the ALU and that a special nonbus communication path be used. The ALU has been extended to 34 bits to support this two–bit shift and add operation.

The barrel shifter is used for the full complement of rotates and logical and arithmetic shifts, plus character insertion and extraction. This variety of functions is controlled by an input multiplexer, which selects the data to go to each word of a two-word combined rotator. The shift amount determines which 32–bit section from the 64-bit concatenation of the two buses goes to the output. The combined rotator is implemented as a pair of cascaded shifters – the first by the *shift amount* div 4, the second by the *shift amount* mod 4. We chose this implementation in the face of severe cell–pitch constraints and the need to drive one bidirectional bus through the shifter. The barrel shifter lies between the registers and the ALU. During ALU operations, two operands must get past the shifter on phase one with the result returning on phase two. This was achieved in a pitch of 33 λ by including one normal bidirectional path (A Bus), and using an internal one-directional bus through the shifter for the other path to the ALU.

Our principal goals for the register array were small size and fast register transfer speed for all operations. Any two cells may be read onto either bus on either phase, or two cells may be written from either bus on phase two. The bus methodology described earlier allows the register pullups to be of very modest strength, so that the power consumption is kept to a minimum. The register array contains both the sixteen general purpose registers and that portion of the Surprise Register that holds the trap code.

The variety of program counter operations and the requirement for instruction restart made the program counter a major challenge. 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 (for interrupt service), the branch value, or values from either of the data buses. Simultaneously, the old value must be shifted into the buffer of old values. When an exception occurs, these values are saved by the service routine and restored on return. The complexity of the program counter structure and its central role in the processor put it directly in the critical path. Because of these speed requirements and full utilization of the ALU by user instructions, a separate PC-incrementer with a simple carry-lookahead scheme was used. The pitch constraint of 33λ was extremely severe in this unit, especially where the program counter and address masking functions overlap.

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. Thus, the first level of memory mapping (segmentation) is provided by the chip; the external memory map need only supply a one–level page map to implement a full segmented and paged system. The use of VLSI made the on-chip segmentation relatively straightforward and decreased the off-chip hardware requirement with no significant performance impact.

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 substituted with a process identifier from another special register. These two special registers are accessible only to processes running in supervisor state. The masking unit also detects attempts to access outside the legal segment and raises an exception to the master

pipeline control. Though the virtual address is a full 32-bits, the package constraint only permits 24 address pins. With word addressing, however, this gives an address space of 64 megabytes.

3.3 Control Bus

The control bus encodes both register sources and destination for each type of operation. It includes a set of true/complement pairs for the ALU sources or memory reference sources, and for the ALU destination or memory load destination. The control bus also includes a collection of special signals including the pipestage, the branch condition results, and various exceptional conditions.

Each data path control driver taps this bus with one or more NOR decoders and latches the decoded value into a driver in the phase immediately preceding its active use in the data path. These signals are driven via dynamic bootstrap drivers by the appropriate clock. The bootstrap drivers put a considerable load on the clocks, but great care has been taken to minimize skew by conservative routing of clocks in metal. In a few cases, bootstrap drivers cannot be used because the control signal may need to be active on both clock phases, e.g., in register reads. These drivers are implemented as large static superbuffers. We have recently discovered that both the dynamic and static drivers may be heftier than necessary because the propagation delay time through the polysilicon control lines appears to be limited solely by diffusion delay.

3.4 Instruction Decode Unit

The Instruction Decode Unit (IDU) latches each instruction and translates the instruction pieces into appropriate control signals for the data path. The instruction word is latched during phase one of the Instruction Decode cycle, and the outputs from the IDU are latched on phase two and distributed to the control bus. The design and encoding of the instruction set is carefully tailored to allow a natural decomposition of the instruction word. Each type of instruction field may appear in one and only one position within the word. Thus, decoding may be done in parallel by two independent PLA's, providing a faster total decode time.

The Load-Store-Branch PLA decodes those fields of the instruction that may contain opcode fields for memory operations or program counter operations. It generates encoded register transfer sources and destinations for the appropriate execution cycles of this instruction. Those signals may be placed immediately on the control bus for the Operand Decode cycle or delayed up to one and a half clock periods until they are needed in following cycles.

The second PLA decodes ALU instruction pieces for the Operand Decode (OD) cycle and for the Operand Store/Execute (SX) cycle. Rather than decoding both pieces simultaneously and delaying the SX output for a cycle, the ALU PLA decodes these two pieces in sequence. The instruction fields for the second ALU piece are held for a cycle and then applied to the same PLA. These latter results appear just before the SX cycle, instead of before the OD cycle. This technique doubles the effective throughput of the PLA.

Both PLA's also contain a compact mechanism for the definition of register fields. Some instruction pieces have implied operands, e.g., multiply step uses the special H-L register, but most instructions require explicit general-purpose registers. Substitution of explicit fields for implied ones is controlled by an extra set of OR lines for each register field in the ALU and Load-Store-Branch PLAs. If the explicit register specifier is indicated, a simple multiplexer on the output of the PLA selects these extra OR lines, rather than the implied operand lines. This field substitution mechanism significantly reduces the size of the IDU.

A small instruction class PLA, operating in parallel with the decode PLA's, selects between the outputs of the two decode PLA's. This selection is based only on the few bits which determine one of four possible instruction piece combinations:

1. Load (long offset)
2. Load (short offset) + ALU2
3. ALU3 + ALU2
4. All conditional instructions

This approach allows the complete instruction decode to occur in one cycle, except for the SX field, which is not needed until one cycle later.

The outputs of the PLA's are linked by a Level Sensitive Scan Design shifter. This LSSD chain allows the function of the PLA's and the data path to be confirmed independently. The heavily utilization of all processor resources mandates this kind of fine control of individual clock cycles for testing purposes. The inputs to the control bus processor can be read or written by the LSSD chain. To test the entire data path, only the bus connections need be working, since complete control of the data path can be obtained via the LSSD chain. This limited use of LSSD represents a good compromise. Little area is lost in the LSSD shift register, compared to a scheme that shifts through all the storage in the processor. However, this limited use of LSSD seems to have most of the advantages of a more complete scheme.

3.5 Master Pipeline Control

The difficult design of the Master Pipeline Control (MPC) reflects the complex control problem of interfacing a pipelined processor to an unpredictable environment. Under normal circumstances, the MPC must control the pipeline execution with cache misses and memory wait states that cause the pipeline to be suspended. It must also deal with a host of potential "surprises" that may occur, including arithmetic overflow, privilege violation, internal masking error, illegal instruction, page fault, interrupt, bus error and reset. The processor must not only respond, but also save the processor state accurately for instruction restart. Finally, the MPC is responsible for generating and sensing all control signals to the pins.

The core of the MPC is the fundamental control engine of the processor: a 16-state finite state machine implemented in a PLA with 46 min-terms. Its inputs include indicators of cache hit, memory ready, and encoded exception information. Its outputs indicate the current machine state in two forms: a standard encoding of the 16 states, and special decoded versions of the same information. These decoded versions control the disabling of register writes and the modification of the program counter on exceptions; these latter signals are needed to control the execution of the next pipestage.

This PLA is surrounded by the surprise state register, and a special PLA for sensing and encoding all the different exception cases, four blocks of random logic. The first block forms the link to the IDU. It creates qualified clocks for instruction decoding and masks illegal instruction indicators coming from the IDU. The second block senses masking errors and sends special signals to the program counter for interrupt service. The third block detects ALU overflow conditions and comparison results. The last logic block forms the interface to the chip's control pins; this interface block qualifies the signals and governs the timing. Our LSSD chain extends through the MPC to pick up all the critical PLA outputs.

The Master Pipeline Control turned out to be unexpectedly difficult to implement. It is both slower and larger than originally planned. The complexity of its logical design has also left little time to fully optimize its physical implementation.

3.6 The Test Chips

During Autumn 1981, we decided to submit the pieces of the MIPS processor as test chips. The six major pieces of the design for which we submitted test chips are the ALU, the barrel shifter, the PC unit, the register file, the IDU, and the MPC. We created a test frame consisting of the control bus and a data path bus interface to allow the frame to fit in a 64-pin package. The test frame was general enough to accommodate all the test pieces. The major aims of the test chips were:

1. To get an early check on the individual pieces of the chip before assembling a complete processor. We were (and still are) afraid of getting back a large chip that is fabricated correctly but does not work and cannot be diagnosed.
2. To obtain some early performance measurements. Our goal was calibration of performance estimates and identification of unforseen critical paths.

In addition, we believed that the test chips could be layed out and debugged in parallel with other design activities.

In reality, what we learned from the test chips was very different. We did uncover several bugs in our individual designs, but we were never able to do performance testing. In addition, we discovered the following startling facts:

1. Yield could be a potentially serious problem, at least for the MOSIS runs. For our first fabrication of the register file, we received no working chips from a batch of ten.
2. Complete integration and simulation of parts takes longer than expected.
3. We had to address testing fairly early on. The ICTEST system [16] provided an unified simulation and testing environment that simplified our task and encouraged us to do more complete simulation.
4. Our static checks were not sufficiently comprehensive and hence some designs had ratio errors. We have worked on fixing the static checker so that it does not overlook any potential problems.
5. Many of our tools broke on the test chips, which was a blessing in disguise.
6. The power consumption of the test chips exceeded our estimate by 50%. We have revised the power budget for the full chip accordingly.

Thus, while the test chips took longer than we expected and did not produce the results we desired, they helped us face up to some of the most demanding parts of the project early on. The success of this approach is discussed in more detail in Section 5.2.

3.7 The Package Constraint

We have chosen an eighty-four pin chip carrier for the MIPS chip. This package imposes two constraints on the 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 the full range of faults, interrupts, and status information. The pins may be summarized as follows:

Function	# Pins	Function	# Pins
Address	24	Data I/O	32
Vdd	3	Gnd	3
Substrate	1	Clocks	2
LSSD	4	System Status In	7
MIPS Status Out	8	Total	84

Since the eighty-four pin chip carrier also imposes a power limit of about two watts, we have had to allocate our power budget carefully. Using a (supposedly) conservative 0.1 mA per inverse square of pullup, we arrived at a total of 311 mA for the power budget in Figure 3.

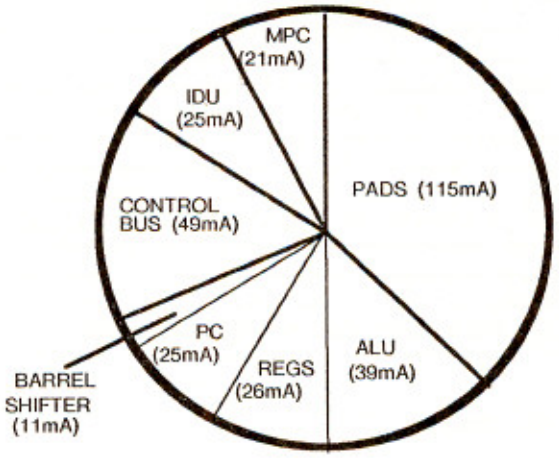


Figure 3: MIPS power budget

Unfortunately, this supposedly conservative estimate is not consistent with our test chip experience. Two different designs, the Program Counter and the Register File test chips from different fab runs (M25KE1,M25HH1,M25KD1), consistently dissipated 1.5 times the expected power. This suggests the pullup current is typically closer to 0.15 mA per inverse square. In this case, we would expect operating current of 470 mA and power of 2.3 W. All these figures represent typical values under nominal temperature and voltage conditions. MIPS appears to use all the power budget available.

4 Limits to Performance

Performance constraints ideally come from the implementation technology and the software environment. Practically, manpower and design tool availability are also important in limiting the performance obtainable in a VLSI design. The first part of this section describes two limitations imposed by the available design tools. The rest of this section describes the interplay of architectural and implementation technology constraints in MIPS. This interaction is demonstrated by examining three performance-enhancing changes that we considered in the the design of the chip. All three of these changes could have been adopted

into the architecture with alterations only in the reorganizer and machine-level architectural specification.

4.1 Limitations of Available Design Tools

One limitation of our design environment was the difficulty of identifying critical paths in the processor. We knew that the propagation delay through the ALU should be on the critical path, but we didn't know during layout whether it would be, or what the other delays along this path would be. Delays within the data path were easy to estimate via circuit simulations of bit slices. The control structure, however, is irregular in comparison with the data path and cannot be simulated in slices. Circuit simulation of the whole control structure was impractical due to its size, and partitioning was impactical due to its complexity. During layout of the control sections, we relied mainly on our limited intuition for transistor sizing. This guessing led to queasiness in the designers and the development of TV. TV is a timing analyzer begun in the summer of 1982; it is described in [12].

Another example of limitations imposed by our design tools can be seen in the final layout of the MIPS processor. The global routing in the control area takes up significantly more space then we initially estimated. A major fraction of this extra area comes from an initially poor placement of blocks and the use of a (sometimes) space-inefficient but simple-to-implement routing scheme. Given the existence of a reasonable automatic router, we could have more carefully optimized the placement and allowed the router to do the bulk of the work; iterating the placement would also be possible because redoing the placement involves little designer effort compared to redoing the routing. Alternatively, we could spend several weeks of designer time hand-optimizing the placement and routing. For the present, we have decided to live with a slightly larger and slightly slower chip.

4.2 Reduced Branch Delay

In a pipelined processor, the instruction fetch unit fetches the next sequential instruction before the previous instruction has completed execution. If the instruction in execution is a branch and if the branch is taken, then the fetched instruction is not the next instruction to be executed. Furthermore, if the pipe is deeper than two instructions, the instruction following the branch can have already begun execution. In most pipelined architectures, this effect is handled by the hardware: if the branch succeeds, any instructions sequentially following the branch are backed-out (if needed) and thrown away; the instruction fetch unit then gets the instructions from the branch destination and execution continues. Because of the high frequency of branches and their aggravating effects on pipelines, various schemes such as branch prediction are used to mitigate the effect.

A simple alternative to complicating the instruction pipeline or implementing a prediction scheme is to have the instructions that are already fetched executed regardless of the effect of the branch. This scheme is called *delayed branching*; a delayed branch of length n means that the n sequential instructions after the branch are always executed. The compiler is given the task of making those instructions safe to execute, and, whenever possible, using the instructions to shorten the execution time of the program [7]. The data shown later in the section shows that this problem can be successfully solved when the branch delay is short.

In the original design of the architecture, MIPS had a uniform branch delay of two for all branches. The ALU computation currently performed in SX was instead performed in a pipestage following OF called EX. This made it easy to synthesize instructions of the form "add memory to register" by combining a load/store piece and an ALU piece. However, it increases the branch delay for all conditional and unconditional relative and absolute

branches by one (i.e., from one to two). These types of branches typically account for 18.9% of all instructions executed (σ=7.1%). Table 1 shows that for a uniform branch delay of two, the average number of branch slots filled would go down to 66.7% (of two instructions) from 90% (of one instruction). A branch delay of length two would increase execution time of programs by an average of 14.3% (0.86 slots empty/branch, 18.9% branches).

Program Name	% Filled Branch Slots for branch delay 1	2	3
Fibonacci	90.0	65.0	56.6
Hanoi	85.7	50.0	38.0
Puzzle I	95.0	79.9	72.2
Puzzle II	75.9	56.5	48.9
Puzzle III	97.0	76.2	67.8
Queens	96.5	72.4	59.7
Average	90.0	66.7	57.2

Table 1: Utilization of branch delay lengths 1,2, and 3

The costs of moving to a branch delay of one were nontrivial. First, it placed much tighter requirements on page–fault detection. In both cases, the memory address leaves the chip at the end of the SX pipestage. In the original design, detection of a page fault is not required until one and a half pipestages later, when the destination of a load instruction is written. In the reduced branch delay design, a page fault must be detected before the ALU result is written only half a pipestage later. Faster virtual memory mapping hardware was investigated and found to be implementable with current technology. Also, two instruction pieces synthesizing a memory–to–register instruction could no longer be packed into the same instruction, but now needed to be reordered into separate instructions. This sometimes increases code size if other instructions cannot be packed with the first piece and the second pieces, but it is not significant in practice. The suggestion for moving up the EX cycle to allow branch delay of one was originally suggested by Jud Leonard in the early fall of 1981 and adopted shortly thereafter.

4.3 Bypassing

The original architectural specification left undefined whether the result of the ALU computation in the SX pipestage could be stored to main memory in the same pipestage by using the same register for the ALU destination and the store source. Spice simulations showed the time required to write the ALU computation through a register to the memory interface would add an extra delay of about 40ns; this delay occurs because the write-through might need to recharge a bus discharged by the original register value. This extra time would be required in every pipestage, whether there was a write-through or not. Thus, allowing write-through would increase the length of each processor cycle by approximately 16%. Bypassing was proposed as an alternative; to bypass, the result of the ALU would be written to both the A and B bus of the data path, and the write-through of the register to the B bus on its way to the storage interface would be disabled. Calculations showed this could be done for an increase in the complexity of the control and a lengthening of each pipestage by about 20ns, or a penalty of about 8% of every cycle.

This left us with three design alternatives: write-through without bypassing, bypassing with disabling of write-through, and redefinition of write-through as illegal in the architecture

with enforcement provided by the pipeline reorganizer. If no other pieces were available for packing, this last alternative could decrease code density, and thus increase execution time, by forcing two pieces previously packable into one instruction into two separate instructions. We define the *packing rate* to be the number of instruction pieces per instruction word. To choose among these alternatives, the pipeline reorganizer and MIPS simulator were used to take data on the usefulness of bypassing. The results of these measurements appear in Table 2. Based on this data, we conclude that the improvement in execution speed obtained by adding bypassing to the MIPS architecture is small. We compared the costs of including either write-through or bypassing, and chose to prohibit storing of a register into memory and modifying it by an ALU operation in the same cycle.

Program	Improvement with Bypassing			
Name	Packing rate	Density	Size	Time
Fibonacci	3.5 %	4.6 %	4.5 %	4.5 %
Puzzle I	23.2 %	16.4 %	12.9 %	5.6 %
Puzzle II	4.2 %	0.7 %	0.6 %	0.3 %
Puzzle III	3.0 %	0.8 %	0.8 %	0.5 %
Queens	6.8 %	5.4 %	5.0 %	5.2 %
Avg.	8.1 %	5.6 %	4.8 %	3.2 %

Table 2: Evaluation of bypassing

4.4 Increased Pipelining

Figure 4 shows the basic timing within one pipestage. Although the bus is heavily used, additional bus bandwidth is available. First, little use of the bus is made during an ALU computation, as well as during the set-up time for the next clock phase. During most of this time, the bus pullups are "precharging" the bus in preparation for the next bus transfer. This precharging could be more effectively done by use of a precharging circuit triggered by the rising edge of each clock phase. This would free up the buses for approximately 50% of each pipestage. This bus time could be utilized by overlapping bus transfers with ALU usage. If the 70ns for next state determination of the Master Pipeline Control were unchanged, this could reduce the cycle time per pipestage from 250ns to 170ns. This overlapping would increase the execution speed of the processor by about 50%. Unfortunately, this overlapping would place severe demands on the rest of the processor. If the same instruction cache and instruction decoding hardware were used, another pipestage would need to be added between IF and ID, to provide the same time for instruction fetch and decode. This would increase branch delays by one, or two for most branches and by three for indirect branches (used on procedure return). This would take away 15 to 20 percent of the added performance (according to calculations similar to those made in Section 4.2). A similar reduction in time available for data references would best be handled by the addition of a data cache.

Other problems with this approach arise in its VLSI implementation. The Master Pipeline Control hardware would be significantly increased in complexity. This hardware is currently a critical path in the machine, so increasing its complexity and requiring it to operate 50% faster could prove extremely difficult. Much of the delays in the control section are caused by large capacitive loads of global control signals, so it might be possible to attain a speed up by increasing the size of control line drivers, although this could cause problems with the overall power budget. Also, this organization would require a three bus design, because the previous ALU or Barrel Shifter result might have to be substituted for either

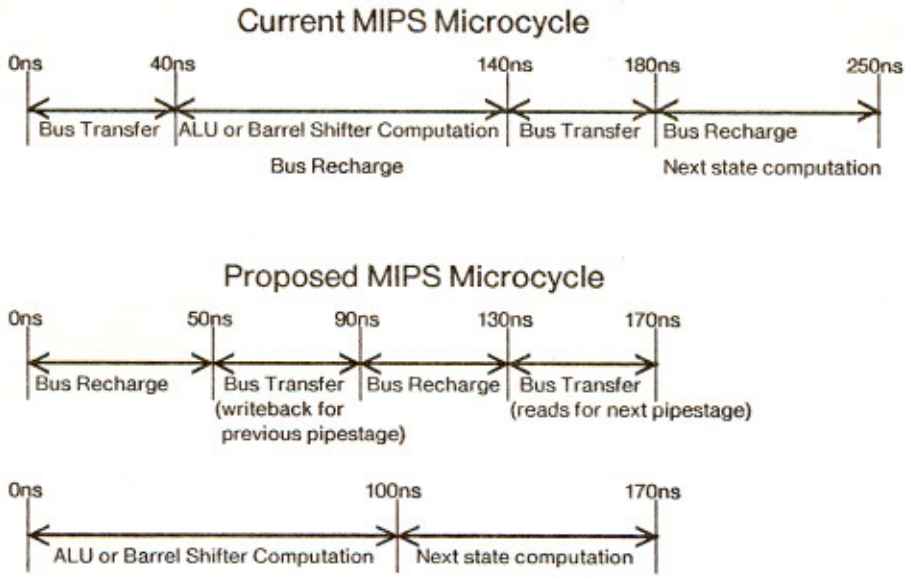


Figure 4: Increased microengine pipelining

operand. Not allowing computations to be available in the next pipestage would be unacceptable due to severe degradation of performance and code density, especially for heavily arithmetic computations, e.g. full-word multiplies and divides. Using a three bus data path would increase pitch by approximately 21%, the area of the chip by about 7%, and would also increase bus transfer times by about 10ns due to increased diffusion delay in lengthened polysilicon control lines crossing the data path. This is a penalty of 13% if we assume a transfer time of 40ns and 25ns for precharging.

Without better cache technology and lower polysilicon resistance, this improved design immediately loses over half of its potential performance increase. The impact on the Master Pipeline Control is hard to estimate; increased demands on it could remove any remaining performance advantage, as well as further increase the size of the chip, causing additional delays due to critical propagation times in the master control lines. The increased complexity of this design would also require a larger design team and more design time due to its substantially more complex logical structure. Lastly, this idea was only proposed after much of the grueling layout for the two bus design had been completed and we had long passed the point of making such a major design change.

5 Lessons, Successes, and Failures

In this section we discuss some of our experiences in designing a large and architecturally novel intergated circuit within a university environment. While some of our experiences may be peculiar to the strange manner in which large projects are attempted at universities, we believe that many of our experiences will be common to any large, complex, and evolving IC design project.

5.1 Chronology

The MIPS project began in earnest two years ago. The following timetable shows some of the milestones of the development process:

Winter 1981 — First ideas for a streamlined processor, discussions of a VLSI implementation. Initial thoughts about a pipelined, compiler-supported architecture.

Spring 1981 — Development of the basic foundation of the architecture by the VLSI Processor Design class. Proposal of an initial instruction set.

Summer 1981 — Stabilization of the instruction set and the data path resources. Development of the framework of the implementation: a two bus structure with each bus potentially carrying data twice per pipe stage. Development of a code generator for the portable C compiler. Start of initial versions of the Code Reorganizer and a Software Simulator.

Autumn 1981 — Four members of the group started to work on implementations of the pieces of the data path. The number of pipestages changed from six to five to reduce the branch delay from two instructions to one. The final few instructions were examined and benchmarked. The pitch of the data path was fixed at 33λ per bit, and we decided to only use buried contacts. We investigated code reorganization algorithms. An ISP description of the processor was completed in December 1981.

Winter 1982 — The first of the test chips was sent for fabrication. It included the register file and its control bus decoders. Serious work began on specifying the control portion of the processor. Bypassing of ALU results was rejected. The instruction set stopped changing.

Spring 1982 — Completion of the program counter test chip. Design of the instruction decode unit. First circulation of the Master Pipeline Control document. A U-Code to MIPS assembly language code generator was started to provide an optimizing Pascal compiler for MIPS. The Reorganizer was rewritten to include all desired optimizations.

Summer 1982 — Submission of the instruction decode test chip and the barrel shifter test chip; implementation of the Master Pipeline Control (MPC) unit.

Autumn 1982 — Submission of the MPC test chip, completion of the ALU test chip. Assembly of the 6 test chips into a processor began. The Timing Verifier was developed and the Pascal compiler was producing high quality MIPS assembly language. By Christmas 1982, chip assembly was completed and simulation of the entire processor began.

Figure 5 shows how the available manpower was divided over the various tasks during the entire design process by showing the tenths of person-years for the first and second years of the project as well as the total. The total amount of effort that has been expended is about

6.1 person–years. The figures show that the project manpower grew from the first to second years as the implementation became the dominant concern. What does *not* show in these figures is that a total of 15 people have at one time or another been working on the main design.

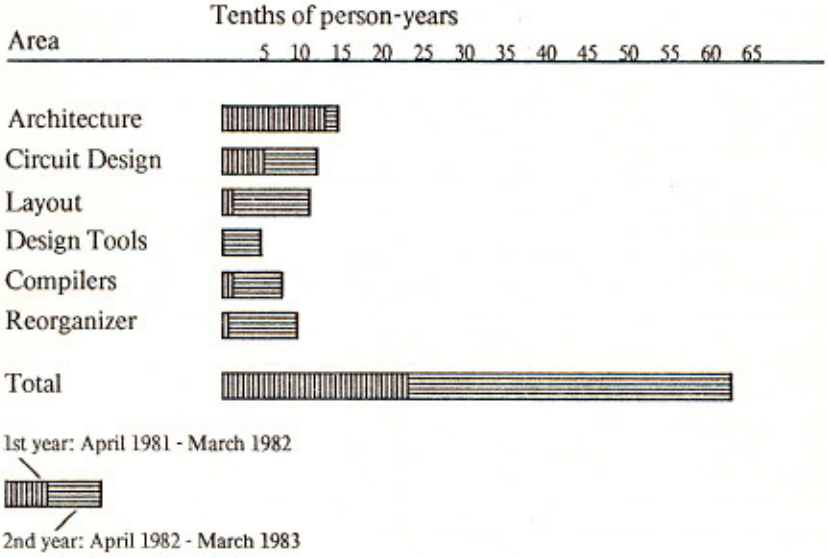


Figure 5: Effort distribution in the MIPS design

5.2 Successes

We strongly believe that instruction set selection on the basis of hard data from compiled code was an important and successful component of our methodology. The fact that we had a C compiler targetted to a MIPS simulator very early in the design allowed us to make design decisions based on data that was directly relevant to the architecture. Many of the last instructions and features considered were borderline in terms of benefits versus hardware cost. Being able to quantify the benefits made the decisions much more straightforward and provided the impetus to resist constant additions to the architecture.

The specification of an assembly language interface that is distinct from the machine language was also a great advantage. Not only were our early compiler efforts isolated to a certain extent from the turmoil of a changing architecture, but the hardware designers were freer to change things with the knowledge that most software changes would be limited to the reorganizer. Another advantage is that different versions or implementations of the architecture need not behave exactly alike. Only the reorganizer need be changed from one implementation to the next. This approach saved us considerable work when we realized that the PC-saving instructions, used for procedure call, could not save the value of the PC that we had initially intended because the PC was incremented at that point in the pipe. Because this instruction was encapsulated by a macro-level call instruction, the required change to the architecture and software was trivial.

One goal throughout the MIPS design was to simplify the hardware implementation

whenever possible. This meant keeping the VLSI implementation regular at the cost of sometimes complicating the software. Also, the streamlined instruction set philosophy and our goal of simplicity in the implementation drove us to eliminate all state not visible from the instruction set. Eliminating internal state meant that complex state savings sequences and information flows to save internal state were not needed. Despite our best efforts, one nonorthogonal feature crept into the design. To perform Booth multiplication two bits at a time, one extra bit, which extends HL to 65 bits, is required. We call this bit the D-bit, for obvious reasons. This single bit of state caused the addition of several very irregular one-bit data paths as well as additional critical timing paths and many headaches in the layout and documentation. The D-bit serves to remind us how valuable simplicity and regularity are in VLSI implementations.

The experience of the RISC chip [14] and the Geometry Engine [2] shows that if performance is not carefully and comprehensively considered during the design, then the results are bound to be disappointing. Our attack on this problem was two-fold. First, we did extensive Spice simulations of a few obvious potential bottlenecks early on in the design cycle. This was to verify the attainable performance range. Second, and more significantly, we undertook to build a timing analyzer, TV (Timing Verifier), that would locate the critical paths in the completed circuit [12]. Several large capacitive loads driven by weak pullups, which might have easily gone unnoticed, have been uncovered by TV.

The test chips were a definite success. Despite disappointing yields and an inability to do thorough performance testing, the effort to place a pad frame around each section was quite worthwhile. Completing the test chip forced the designer of each section to fully simulate his section of the processor before it was incorporated into the final design. Furthermore, this simulation and debugging of each section could be done in parallel. If the pieces had been assembled without thorough separate simulation, this debugging would have been much more sequential. As in any large system, bugs in one part of the circuit might have caused misleading behavior in another, making the debugging still harder. If the test chips had not been assembled, the processor might have been ready for simulation perhaps four to six weeks earlier. In our estimation, this time is shorter than the length of the added debugging effort that would have been required if the pieces were assembled directly. Additionally, we uncovered several bugs in our design aids early. Had the design aids broken on the entire chip, we would have had a more difficult time tracking down these bugs and delayed our final design still further.

Another good decision was to use a layout language called SILT [4] as our chip assembly tool. Each of the six basic blocks of the processor was laid out with the graphical layout editor ICARUS [5]. After layout, blocks were placed and interconnected with SILT. This approach saved considerable effort in the assembly of the test chips because they were all quite similar and shared a lot of SILT code; without this approach, the test chips would have been prohibitively expensive to construct. Graphical editors are very convenient when a designer is laying out the leaf cells of a design; these cells are typically very random and tightly packed. However, modifying a large design can be very painful with a graphical editor, as are more global tasks such as block placement and routing. If a wire has been forgotten, a great many rectangles might have to be manually changed to separate two major functional blocks by 7λ.

In contrast, a procedural layout tool like SILT is very awkward for designing leaf cells because of an extended edit-compile-plot cycle, and because it is often very difficult to visualize what a long sequence of *place box* commands produces. SILT is very good at laying out geometry that can be algorithmically expressed. An example of this phase of layout is the requirement to "place the barrel shifter above the ALU". If the ALU should happen to change height and the SILT description was written with some care, the barrel shifter and all its connections should move appropriately. SILT also has a simple, built-in river router that provides a useful tool for connecting arrays of locations, such as buses, groups of control lines and even single wires in some cases.

SLIM [8] also proved to be a great labor saving tool in the design of the ten PLA's on the

chip. It allowed each PLA to be abstracted from the level of Boolean equations to a higher functional description that could be written, read, and maintained with significantly less effort.

5.3 Failures

In any large project with many decisions, some of them will be wrong. In our case, many of the wrong decisions really represent a lack of decision. Specifically, a number of things continued to change after the layout had begun. Among the worst of these was the pitch of the data path. Due to changes in the support of buried and butting contacts, we decided late in Fall 1981 to banish butting contacts and use a data path pitch of 33λ. This forced a significant amount of layout effort to be repeated. While talented designers may be very effective in doing the layout of a tight data path cell, having such designers redo a layout is a waste of talent. The squeeze from 35λ to 33λ done during the redesign with buried contacts made the layout very difficult; the small performance improvements gained were probably not worth the effort expended in the design with such a narrow pitch.

The instruction set also was in a state of flux until quite late in the design. Minor changes were being made well into 1982, as the pieces of the data path were being designed. The best example is the *count leading zeros* instruction which was added late in Fall 1981, dropped around May 1982, then temporarily added again for about a week in June. It is useful to point out that this instruction was added and dropped based solely on its ability to be implemented by the hardware. Despite that fact that most of the late changes resulted in definite improvements (the change from 6 to 5 pipe stages is a good example), they inevitably caused repetition of effort.

Another area in which a lack of forethought, or rather lack of parallel thought, caused problems was in the external interface. Because all the designers were fairly busy, the details of the Master Pipeline Control had to be worked out after the architecture was firmly entrenched. That meant solving all the hard problems relating to privileges, page faults, and exceptions within a fixed framework. The result is a few minor quirks in the exception handling system that *may* have been avoidable if there were more flexibility in the structure of the mechanism. A prime example is the difficulty in returning from an interrupt with the lock-step, fixed length, pipeline. The problem arises because one needs to restart the three instructions that were in the pipe at the time of the exception using three successive indirect jumps to those three instructions. However, complications arise because the last indirect jump has to do its data fetch from system space after the first target instruction is being fetched from user space. If redesigning the existing program counter block had been an option at the time this problem was realized, a conceptually simpler mechanism might have been found.

As in many projects without rigorous enforcement of documentation standards, the current state of the design rested only in the heads of those most intimately involved with the project. This meant that our communication problems were larger than they should have been. In several instances, the same problem had to be thought out a second and third time because the answers were not written down the first time. The timing of the control buses was derived on at least three separate occasions before the Master Pipeline Control document was written.

The documentation problem was amplified by our manpower situation. A lot of the people that were working on the project were only peripherally involved or only involved for a quarter or two. They spent a great deal of time learning the basics of the processor. Furthermore, the core project team spent valuable time teaching new contributors the basic ins and outs of the design.

Worse still was a misunderstanding that arose on several occasions. Someone on the edge of the project would undertake to do a small piece of the processor without fully

understanding the interfaces to the surrounding blocks or the timing requirements. Unable to find any hard documentation or some person who knew all the answers, the designer would, knowingly or otherwise, make assumptions, some of which inevitably turned out to be false. When the erroneous assumption was finally caught at the supposed completion of the task, significant redesign was often needed.

5.4 Lessons

Many of the lessons that we think we have learned are the same ones learned by VLSI designers, and engineers in general, time and time again. One problem that can be particularly serious in VLSI is synchronizing the design of various subsystems; one aspect of the design should not get way ahead the others. If manpower and time permit, it appears best to have all the portions enter circuit design and subsequent layout together. A more coordinated design might have saved much aggravation in the design of the control portions of the chip.

Many engineering projects are sparsely documented; this is especially troublesome when the work force is likely to change and the design is evolutionary. Documentation is a difficult and unexciting task, but it is seldom a waste of time.

The effort required to design a VLSI part makes it important to decide on as many things as possible before the drawing of transistors begins in earnest. Particularly when a number of people will be working on interacting parts, all aspects of these interactions, be they physical, electrical or temporal, should be at least considered, if not set in concrete, before either party proceeds. This is not to say that small sections should not be laid out on a trial basis during earlier stages of the design to get size, power and speed estimates, but rather that everybody should be aware that these initial efforts are likely to be made obsolete by future developments. If the high-level design can also be partially flexible to implementation-driven changes, the design will be dramtically eased.

The design tools being used on an ambitious design *will* break. Of course, the tools always break just before a deadline and just after the maintainer has gone on a long backpacking trip into the wilderness. We found some design aids that broke because the design was too big, but more often our design tools could not handle the less constrained design methodology that we used. This often produced unexpected results in tools believed to be very stable. In other cases, we found that our design could not be recognized as "standard" by the tools and was rejected outright (our bus-pullups caused this difficulty).

There comes a time in the design process when one can no longer afford to go back and change things. Changing things early is far less expensive than changing them later. A great deal of engineering time and aggravation can be saved by fixing the freeze date on the project, continually moving that date can cause constant redesign.

MIPS also taught us some things about designing processors in VLSI. First, high performance and Mead-Conway design techniques are not mutually exclusive for large designs. Given the right tools one can make large chips go as fast as similar small chips. However, to break into the 100ns clock period range requires circuit and analysis techniques that are beyond most Mead-Conway designs and designers. Self-timed precharging, dynamic logic, sense amps, and similar approaches are needed to make the most of the technology.

Adopting a streamlined instruction set has allowed us to experiment with pipelining as a media to attain high performance. Without the simplifications of a streamlined instruction set and the two-part definition of the architecture, an implementation would have been out the question. Both the practical size and the manageable complexity of our integrated circuit stem directly from these two ideas.

VLSI is a technology that is much more bottom-up driven than other implementation media. The advantages and disadvantages of alternative implementations are difficult to

assess without actually carrying out the designs. Low-level layout problems encountered late in the design cycle sometimes force changes in the specification. To whatever extent possible, the noncritical design specifications should be alterable as the design progresses and unforseen problems demand change to the specification.

If at all possible, every aspect of the processor should be analyzed from the context of hardware costs and software benefits. Hardware resources are far from free, particularly within the context of a single–chip implementation. The costs in manpower, area, power and complexity must be compared with the performance improvement resulting from the feature.

Conclusions

Our biggest success was completing the design of a microprocessor that combines high performance with the correct handling of systems issues and external interrupts. We were able to realize this project within the bounds of our environment because we started with few preconceived notions about processor design and then considered design decisions without bias. Instruction set design can be turned into a scientific process where experimental evidence backs design decisions. This approach allowed us to increase preformance by implementing some unique hardware-software tradeoffs: functions with low utility but a high cost of realization in hardware are relegated to software where the problems are more efficiently handled.

As this paper goes to press, we are completing the final stages of simulation of the entire MIPS chip and expect to submit the processor for fabrication shortly, provided our tools and energy hold up for the next few weeks.

Acknowledgments

Many people have contributed to the MIPS project; especially worthy of note are John Gill, Forest Baskett, and Jud Leonard.

The MIPS project has been supported by the Defense Advanced Research Projects Agency under contract # MDA903-79-C-0680.

References

1. Chaitin, G.J., Auslander,M.A., Chandra,A.K., Cocke,J., Hopkins,M.E., Markstein,P.W. Register Allocation by Coloring. Research Report 8395, IBM Watson Research Center, 1981.

2. Clark, J. The Geometry Engine: A VLSI Geometry System for Graphics. Proc. SIGGRAPH '82, ACM, 1982.

3. Davidson, S., Landskov, D., Shriver, B.D., and Mallett, P.W. "Some Experiments in Local Microcode Compaction for Horizontal Machines." *IEEE Trans. on Computers C-30*, 7 (July 1981), 460 - 477.

4. Davis, T. and Clark, J. SILT: A VLSI Design Language. Technical Report 226, Computer Systems Laboratory, Stanford University, October, 1982.

5. Fairbairn, D. and Rowson, J. An Interactive Layout System. In *Introduction to VLSI Systems*, Mead, C. and Conway, L., Eds., Addison-Wesley, 1980, ch. 4.4, pp. 109-115.

6. Fisher,J.A. "Trace Scheduling: A Technique for Global Microcode Compaction." *IEEE Trans. on Computers C-30*, 7 (July 1981), 478-490.

7. Gross, T.R. and Hennessy, J.L. Optimizing Delayed Branches. Proceedings of Micro-15, IEEE, October, 1982, pp. 114-120.

8. Hennessy, J.L. A Language for Microcode Description and Simulation in VLSI. Proc. of the Second CalTech Conference on VLSI, California Institute of Technology, January, 1981, pp. 253-268.

9. Hennessy, J.L. and Gross, T.R. "Postpass Code Optimization of Pipeline Constraints." *ACM Trans. on Programming Languages and Systems* (1983). Accepted for publication.

10. Hennessy, J.L., Jouppi, N., Baskett, F., and Gill,J. MIPS: A VLSI Processor Architecture. Proc. CMU Conference on VLSI Systems and Computations, October, 1981, pp. 337-346.

11. 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.

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

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

14. Patterson, D.A. and Sequin, C.H. "A VLSI RISC." *Computer 15*, 9 (September 1982), 8-22.

15. 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.

16. Watson, I., Newkirk, J., Mathews, R. and Boyle, D. ICTEST: A Unified System for Functional Testing and Simulation of Digital ICs. Proc. Cherry Hill International Test Conference, Philadelphia, 1982.