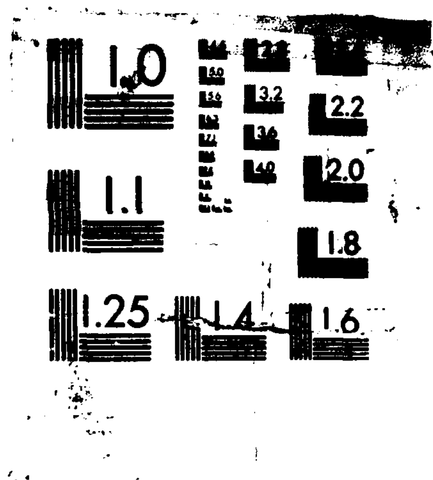
AD-A182 299 ARCHITECTURAL TRADEOFFS IN THE DESIGN OF MIPS-X(U) 1/1
STANFORD UNIV CA COMPUTER SYSTEMS LAB P CHOW ET AL.
1987 MDA903-83-C-0335

UNCLASSIFIED F/G 12/6 NL



MICROCOPY RESOLUTION TEST CHART

To appear in The 14th Annual International Symposium on Computer Architecture, June 3-6, 1987, Pittsburgh, PA.

AD-A182 299

DTIC FILE COPY

DTIC ELECTE JUN 2 9 1987 S A D

Architectural Tradeoffs in the Design of MIPS-X

Paul Chow and Mark Horowitz

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

The design of a RISC processor requires a careful analysis of the tradeoffs that can be made between hardware complexity and software. As new generations of processors are built to take advantage of more advanced technologies, new and different tradeoffs must be considered. We examine the design of a second generation VLSI RISC processor, MIPS-X.

MIPS-X is the successor to the MIPS project at Stanford University and like MIPS, it is a single-chip 32-bit VLSI processor that uses a simplified instruction set, pipelining and a software code reorganizer. However, in the quest for higher performance, MIPS-X uses a deeper pipeline, a much simpler instruction set and achieves the goal of single cycle execution using a 2-phase, 20 MHz clock. This has necessitated the inclusion of an on-chip instruction cache and careful consideration of the control of the machine. Many tradeoffs were made during the design of MIPS-X and this paper examines several key areas. They are: the organization of the on-chip instruction cache, the coprocessor interface, branches and the resulting branch delay, and exception handling. For each issue we present the most promising alternatives considered for MIPS-X and the approach finally selected. Working parts have been received and this gives us a firm basis upon which to evaluate the success of our design.

Introduction

The first generation reduced instruction set processors (IBM 801[1], RISC[2,3] and MIPS[4,5]) have shown the importance of making the correct tradeoffs across the boundary that separates hardware complexity and software functionality. Hardware should only be used to support features that clearly improve performance. As implementation technology improves, new features can be considered and new tradeoffs must be made.

The goal of the MIPS-X project was to combine a new technology, a 2μm, 2-level metal CMOS process, with the knowledge and experience gained from the first generation RISC machines, to build a single processor with a peak rate of 20 MIPS and then to use 6-10 of these processors as the nodes in a shared memory multiprocessor. The resulting machine would be about two orders of magnitude more powerful than a VAX 11/780 minicomputer.

We describe here the design of the single processor, MIPS-X. The overriding principle was to keep the design as simple as possible. The original MIPS team was heavily involved in the initial architectural discussions, and they helped steer MIPS-X away from the kinds of trouble that they faced with MIPS. The major areas of concern were control related, of which the most important were considered to be instruction decode and exception handling. Both were not considered early enough in the MIPS design and created difficult implementation problems in the final chip.

The design of the instruction format was straightforward since we religiously adhered to a maxim given in the first working document on MIPS-X. It stated, "The goal of any instruction format should be:

1. Simple decode,
2. simple decode, and
3. simple decode.

Any attempts at improved code density at the expense of CPU performance should be ridiculed at every opportunity." Needless to say, all instruction sets considered for MIPS-X were fixed format 32-bit words and the amount of decoding was minimal. The effects of having this simple instruction format is discussed in the conclusions.

Not all areas were as stable as the instruction decode. Before presenting the major tradeoffs we made in the MIPS-X design, the next section describes the basic architecture of the processor and the following section gives an overview of the hardware and organization of the machine. This is followed by several sections, each discussing a major design issue in MIPS-X, the solution used and the rational for that decision.

MIPS-X Architecture

The goal of the MIPS-X project was to design a microprocessor with an order of magnitude more performance than the original MIPS processor. MIPS-X borrows heavily from the original MIPS design; it is again a heavily pipelined machine, and the resulting pipeline interlocks are handled by the supporting software system. MIPS-X differs from MIPS in that it aims for single-cycle execution using a much faster clock (20 MHz), a deeper pipeline and better implementation technology.

The high instruction rate means that memory bandwidth is an important consideration. At the projected clock frequency

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This document has been approved for public release and sale; its distribution is unlimited.

87 5 2

of 20 MHz it is very difficult to satisfy instruction and data fetch requirements across the available package pins. To alleviate this problem, MIPS-X has a 2K-byte, on-chip instruction cache (Icache). Only instructions that miss in the Icache pass through the package pins. The Icache is placed above the datapath, in the area of the chip that is normally used for microcode storage and processor control. Data references and instruction references that miss in the Icache are handled by a large 64K word external cache (Ecache). The Ecache uses a shared bus to communicate with main memory. An added benefit of this two-level cache is that it provides a second port to memory; the processor can fetch an instruction from the Icache at the same time it is accessing off-chip data.

A deep pipeline is used to allow the machine to start a new instruction every cycle. Each instruction is divided into five pipeline stages. They are described in Figure 1. All control is hardwired.

IF	Instruction fetch.
RF	Instruction decode and register fetch.
ALU	ALU or shift operation
MEM	Wait for data from memory on a load and output data for a store.
WB	Write the result into the destination register.

Figure 1: MIPS-X Pipestages

The machine uses a load-store architecture; the only memory operations are explicit loads and stores. The use of the ALU cycle depends on the instruction being executed. For compute instructions, this cycle performs the desired computation, for memory instructions it is used to compute the address of the desired memory location and for branch instructions, it is used to compute the condition. All memory operations use the same addressing mode; the contents of a register are added to a 17-bit signed offset to produce a 32-bit address. There are 32 general purpose registers in the datapath with a 32-bit ALU and a funnel shifter for compute operations.

Although a compute instruction finishes its computation during the third pipeline cycle (ALU), the result is not written back into the register file until the last pipeline cycle. This delayed writeback is done to make instructions only change machine state during their last pipeline cycle, making exception handling much easier. Bypassing is used to reduce the number of pipeline interlocks.

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

A Hardware Overview

The major components of MIPS-X are the instruction cache data array, the instruction register and the datapath. The datapath is composed of the register file, the execution unit, PC unit and the tag store for the instruction cache. The organization of these parts is shown in Figure 2.

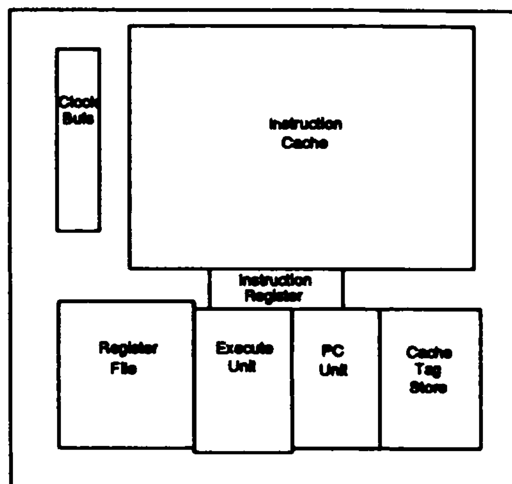


Figure 2: MIPS-X Floorplan

The instruction cache is organized as an 8-way set-associative cache, with 4 sets (rows) and 16 words in each block (line). A sub-block replacement scheme is used so there are 512 valid bits, one per word, as well as the 32 tags. These are located in the datapath to decrease the time needed to detect an instruction cache miss.

The instruction register latches the output from the instruction cache and predecodes some fields of each instruction. It also controls the flow of data during cache misses so that instructions can be written into the cache. During a cache miss, the instruction is latched in the instruction register from the data bus while it is going to the cache memory array. This latch provides a very useful testing feature by allowing the processor to run with the cache disabled.

The register file contains 31 general purpose registers and a hardwired constant zero register. It is useful to have a read-only register as a place to write unwanted data. The constant zero was chosen because it is used as a source value for many instructions such as loading immediate values by doing an add immediate to Register 0. Registers to handle two levels of bypassing and the memory data registers are also in this section.

Shifting and ALU operations are done in the execute unit. It contains a 64-bit to 32-bit funnel shifter and a 32-bit ALU. There is also a special register, called the MD register, that is used during multiplication and division instructions.

The program counter, or PC unit, contains a displacement adder for branches, an incrementer and a chain of shift registers to save the PC values of the instructions currently in execution. Having both the displacement adder and the incrementer means that as soon as the branch condition is determined the PC bus can be driven with the correct value. The PC values in the shift chain are needed to restart the machine after an exception.

In a small area above each section of the datapath is local instruction decoding and control for that section. The overall control of the machine is handled by two finite state machines located in the PC unit. One of them is used to handle Icache misses and the other one does instruction *squashing* during exceptions and branches. Squashing an instruction converts it into a *no-op* instruction.

Critical Paths

To run the processor at or above 20 MHz meant that much attention had to be paid to possible critical paths. In each cycle, we tried to minimize the number of series operations as much as possible. Whenever feasible, a signal was given a full phase to be decoded and driven from one section to another.

There were a few paths that we felt were most likely to be critical paths and we spent a lot of time concentrating on them. The most important of these involved external data fetches. In the specification for the pipeline, addresses would be computed during $\phi 1$ of the ALU cycle and driven to the address pads during $\phi 2$. The Ecache would be accessed during the MEM cycle. Even assuming that the address could be driven off the chip by the end of ALU, completing a fetch in 50 ns would be tight because of the address buffer delay, memory access time and setup time for the fetched data. Getting the result of the tag compare back in a cycle seemed impossible since this would also involve delay through some comparators. To ease the constraint on getting the tag compare back, we decided to use a *late-miss* signal. This meant that the cache would inform the processor at the beginning of the WB cycle whether the cache access during MEM was successful. If there was a miss, then the processor would effectively go back and re-execute $\phi 2$ of MEM to try the access again. This loop would continue until the cache got the data and signaled a hit. Throughout the design we had to be careful not to unnecessarily add delay to the memory-fetch path.

Other paths that we tried to optimize included the path from branch condition generation to driving the PC Bus, instruction cache hit detection, squeezing the ALU time into 1 phase to get the address out by the end of the cycle and doing register reads and writes in one cycle. The latter two were strictly circuit design issues and are not discussed any further here.

The Instruction Cache

Advances in processor architecture and VLSI technology have increased faster than the improvements in packaging technology. This has meant that high-performance VLSI processors have become memory bandwidth limited. For example, if we assume that one instruction is fetched every cycle while, on average, data is only fetched every third cycle, then MIPS-X will have an average bandwidth of 26 MWords/s and a peak bandwidth of 40 MWords/s. Clearly, on-chip memory would help to alleviate this bottleneck. For MIPS-X, we built an on-chip 512-word instruction cache and the tradeoffs made in its design are described in detail elsewhere[6]. We will only discuss the salient features here.

The instruction cache was the first part of the chip to be designed. We first fixed a die size that we felt had enough area to implement the functionality we desired yet small enough that we could expect a reasonable yield of working parts. The datapath and control would take about half of the area inside the padframe so the cache was allocated the remaining area fixing its area and aspect ratio. The other main constraint on the cache was that the cycle time had to be less than the 50ns clock cycle. Given these constraints we investigated many different floorplans and organizations, trying to minimize the average cost of an instruction fetch. This cost is a function of the cache hit rate, the miss penalty, and the cache access time.

We found that the performance of the cache was more sensitive to the the miss service time than the miss ratio. This meant that the implementation details of the cache were more important than the cache organization because the implementation affected how quickly we could determine whether an address hit in the cache. With our pipelining, this meant the difference between stalling the machine for 2 or 3 cycles on a cache miss. By placing the tag and valid-bit stores in the datapath close to the PC unit a 2-cycle miss could be realized. This lengthened the datapath by the number of cache tags and meant that we could not have smaller block sizes because more tags would make the datapath too long. However, the benefits of having fewer cache miss cycles far outweighed the slightly lower miss rates achievable by having smaller blocks.

Initial simulations of this organization yielded disappointing results. Using a set of medium size programs we achieved miss rates that averaged over 20%. We felt that real programs would have worse miss rates, pushing the cost of an instruction fetch close to 1.5 cycles. We found a way to reduce the number of cache miss cycles to 1 by writing the missed instruction into the Icache as soon as it got back onto the chip, but since accessing external data was already one of the critical paths we did not want to risk extending the cycle time to complete the write. Instead we realized that the 2 cache miss cycles could be used to fetch back 2 instructions, the one that missed and the next one to be executed. Doing this double fetch did not affect the critical path and, in fact, was easier to do than fetching back only one instruction because it minimized the disruption of the pipeline. Fetching back 2 words almost halves the miss ratio, driving down the cost of an instruction fetch to that of a single-cycle miss. The key realization here was that there was extra cache bandwidth available and that we could use it to fetch back the next instruction, significantly improving the cache miss ratio without impacting the cycle time of the machine. Fetching back more words would not be advantageous because the bandwidth of the cache is fully used.

Trace driven simulations show that with our set of large Pascal and Lisp benchmarks, the cache has an average miss rate of 12% resulting in an average instruction executing in 1.24 cycles.

The Coprocessor Interface

The coprocessor interface was considered from the very beginning of the design. It also led to some of the most interesting discussions within the MIPS-X design team. We spent considerable time trying to find an efficient interface that would give reasonable performance and still fit within the constraints of VLSI packaging and design. This problem was exacerbated by the presence of the on-chip instruction cache, since now all instructions would not be visible to the outside world.

The proposal for the first instruction set had a single bit in every instruction to specify whether the instruction was for the CPU or a coprocessor. For instructions with the coprocessor bit set, MIPS-X would perform all the addressing calculations, but would not affect any of its stored data. That is, all coprocessor memory instructions still used the processor to generate the addresses and the required control signals, while the coprocessor either acted as a source or sink of the data. To make the coprocessor instructions visible outside of the processor, a dedicated bus was required to transfer the instruction off the processor chip. This scheme had 2 disadvantages: all interprocessor communication had to go through memory, and a coprocessor bus was required. A minor concern was that half the opcode space was devoted to the coprocessor; there had to be a more efficient encoding.

The next instruction format divided the opcode space into three instruction types: memory operations, branches and compute operations. The memory and compute instructions had a 3-bit field to specify the coprocessor number, branches were only done on the main processor. If Coprocessor 0 was specified then the instruction was for the main processor, otherwise the instruction was for one of the 7 available coprocessors. To branch on a coprocessor condition, the coprocessor would first be told to assert a single input to the main processor and a *branch on coprocessor true* or *branch on coprocessor false* would be executed to test the status of that input. Several coprocessors could be connected by wire-oring their outputs. This scheme still had the problem that data transfers between processors must be done through memory.

It was then proposed that all coprocessor instructions must be non-cached, removing the need for a coprocessor bus. The issue of pins and pin bandwidth was heavily debated within the MIPS-X design team. Pins on the processor were in short supply and devoting approximately 20 of them to the coprocessor interface seemed excessive. The question was not just whether there were enough pins available. Without the coprocessor bus, MIPS-X would need only about 90 signal pins, a relatively small number by today's standards. Rather the argument focused on what would be the best use of these pins if we had them. It was not at all clear that using them for the coprocessor interface was the most effective use of the pins. To prevent coprocessor instructions from being cached, a bit in the instruction cache would be set when an instruction being loaded was detected to be a coprocessor instruction. If the bit was set during an instruction fetch that missed, the coprocessor would get the instruction off the memory bus as the main processor read the instruction from memory during the cache miss cycle.

The obvious disadvantage of this approach was that all coprocessor operations incurred an overhead from the internal cache miss. Our initial benchmarks indicated that this would not cause a significant performance loss, but when we generated traces from some floating point intensive code we realized a significant percentage of the instructions were floating point instructions. This caused a re-examination of the decision to not cache coprocessor instructions, and led to the coprocessor scheme that was finally chosen.

The opcode encoding of the machine was changed again, this time making coprocessor operations a form of memory operation or more accurately, memory instructions became a type of coprocessor instruction. Coprocessor instructions work in this scheme by using the address lines to transmit the coprocessor instruction. A memory instruction takes a 17-bit offset constant and adds it to the contents of a register to compute the memory address. If the memory system ignores the cycle, it is possible to pass the 17-bit offset constant to a coprocessor as an instruction. The instruction would include a 3-bit field to specify the coprocessor being addressed, although the processor does not need to know the format of these instructions. This scheme has several advantages over our earlier ideas. A coprocessor instruction bus is not required, since the instructions are sent out over the address pins. Only one extra pin is required to tell the memory system to ignore the cycle. Additional pins can now be used for alleviating the pin bandwidth problem in other parts of the system. Using coprocessor load and store instructions, data can be directly transferred between processors by making the coprocessor supply or read data on the data bus instead of the memory. Also, the coprocessor instructions can be cached just like all the other instructions. The disadvantages of this scheme are that there are fewer bits to specify the coprocessor instructions, and all data to and from the coprocessor's registers must be transferred through the main processor registers first before it can be sent to memory.

Having to transfer all data through the main processor registers was still thought to be inefficient for heavy floating point computation. This lead to a further modification of the instruction set to add *load floating* and *store floating* instructions. These instructions provide one special coprocessor with its own load and store instructions, which we assume will be a floating point unit (FPU). The interface now allows one special coprocessor to load and store its registers directly to memory, without passing through the main processor, in a single instruction. All other coprocessors require one extra cycle for memory loads/stores.

One final tweaking of the interface was to remove the coprocessor branch instructions. The main reason for their removal was the problem of saving state in the coprocessors across exceptions. The solution was to just read a coprocessor status register into a main processor register and then branch according to the value of that register. This change eliminated the last set of problems we had discovered with the coprocessor instructions.

By using the address lines, the resulting coprocessor interface has instructions that can be cached, does not require a large coprocessor bus, allows efficient communication between the processor registers and the coprocessor registers, and lets a single coprocessor have direct access to memory.

Branches

Having set out the initial architecture of the machine, we quickly ran into the problem of branches, and branch delays. Branches have a considerable effect on the performance of a computer especially one that is pipelined as deeply as MIPS-X. The effects of branches in a pipelined machine are particularly noticeable because branches interrupt the flow of the pipeline. Decisions about the design of the pipeline and the type of branch scheme used are not independent. Control complexity is a serious issue.

We very quickly decided to eliminate the use of condition codes in MIPS-X if possible. This decision was motivated by two facts. First, instruction trace statistics indicated that a prior compute operation infrequently generated the condition code needed for a branch. In roughly 80% of the branches an explicit compare operation must be performed to set the condition codes. A previous analysis[7] of empirical data showed that the number of instructions saved by condition codes was very small and essentially useless. Second, condition codes generate state that needs to be saved and restored during exceptions. Handling condition codes in a pipelined machine is difficult because when an exception occurs, great care must be taken to ensure that the correct condition codes are saved. It seemed to us that condition codes provide little benefit and have potential complexity problems. In particular, generating code to use condition codes efficiently is not as straightforward as one might expect. All the branch schemes considered for MIPS-X contained an explicit compare in the branch. This actually reduces the amount of control logic required because there is no need to worry about how to save this state.

Two arithmetic operations are required to execute a branch instruction. One is to compute the branch condition and the other is to compute the branch destination. A machine that uses condition codes computes the branch condition before the actual branch instruction and saves the condition in a condition code register. The first idea conceived for implementing branches in MIPS-X computed the condition in the branch instruction, but did not compute the branch destination. Instead the branch destination was made explicitly visible in the architecture. The user would have to load a register called *PC+1* with the branch destination. The branch instruction computes a condition and then selects *PC+1* or the next sequential instruction depending on the computed condition. An observation was made that many inner loops contain several forward branches due to constructs like if-then-else statements so it would be good to have several *PC+1* registers. Four was felt to be sufficient. This would allow the compiler to hoist the destination address calculations out of the loop. Without this feature, the contents of *PC+1* would have to be loaded from a register for each branch within the loop for each iteration of the loop.

This scheme still had the problem that there was some state that must be saved (the *PC+1* registers) when an exception occurred. Also, deciding how to use the *PC+1* registers could be cumbersome for the compiler system. Finally, with four special registers, it was no longer clear that this solution was easier to implement than simply including a separate adder to compute the destination while the ALU performed the comparison. At this point in the design, adding a little hardware to the datapath to make the control simpler was the wisest choice so we added the separate adder to compute the destination.

During this period we also became concerned about the effect of the branch delay slots on the machine's performance. Often in a pipelined machine one or more instructions following a branch are fetched before the result of the condition evaluation is known. If these instructions are executed, then the machine is said to have a *delayed branch* meaning the effect of the branch occurs after the actual branch instruction. The number of cycles or *delay slots* that execute after the branch instruction and before the actual branch occurs is called the *branch delay*. Filling these delay slots is not a simple task[8, 9, 10] and affects the overall performance.

In the MIPS-X pipeline, it is most straightforward to implement a branch with a delay of two. The ALU is used to compute the branch condition during the third (ALU) pipestage. Filling two delay slots did not seem very promising. Using data from MIPS instruction traces, we expected over 50% of the slots to remain empty[8]. This performance problem lead to discussions about how to reduce the branch delay to 1 cycle, and whether we could use branch prediction to help reduce the wasted cycles[11, 12].

A *quick compare*[3] was proposed as a method to reduce the branch delay. In this scheme, simple comparisons between the two source registers are done before the ALU cycle. This comparison would be performed at the end of the RF cycle by placing a comparator on the output of the register file. Only equality and sign comparisons can be obtained using this method since there is not enough time for an arithmetic operation. Other conditions such as *greater than* would require two steps. The ALU operation is done first and the result is stored in a register. This result is then used in a quick sign compare instruction.

The main question that needed to be resolved initially was what percentage of branches could be handled by a quick compare. Statistics from Katevenis's thesis indicate that by changing the compiler slightly, about 80% of all branches can be converted into quick compares[3], but this means that 20% of all branches take two cycles. Our initial statistics indicated that the number of branches that could be handled using a quick compare was between 70% and 80%.

The quick compare was eventually dropped because it could potentially lengthen the processor cycle time. The comparator circuit must operate on the source buses leading to the ALU and since the values on the buses could come from a bypass source it was possible that the buses would not be stable until late into that cycle, particularly for a previous memory fetch because the data would only be back at the very end of the cycle. For the quick compare to operate, we would need to perform a compare on these values and then use this result to select the correct address of the next instruction. The potential increase in cycle time discounted its slight advantage in the average number of cycles it takes to complete a branch. In retrospect, our decision was correct. In the final machine, the delay from the generation of the branch signal to driving the correct value on the PC Bus is long (measured to be about 20 ns). Even providing a full phase to drive this path leaves it on a critical path.

Left with a branch delay of 2, we investigated branch prediction as a way to reduce the effective branch delay. There were two prediction algorithms tried: branch cache, and static prediction. The branch cache was quickly discarded

when we discovered that it had to be fairly large (much greater than 16 entries) to get a high hit rate. It would also affect the size of our instruction cache. Besides, it never did much better than static prediction and was much more complex. Static prediction would use information at compile time (possibly with profiling) to predict which way a branch would go.

To make use of the prediction information we considered implementing *squashing*, the ability to convert an instruction into a *no-op* if the branch did not go in the predicted direction. In MIPS, the instructions in the branch delay slots are always executed. The strategy for choosing instructions is to first try to move an instruction from before the branch into the slot. If no instructions can be moved past the branch the next choice is to find instructions from the destination or the sequential path that have no effect if the branch goes the wrong way. Thus if you predict correctly, the slot performs a useful instruction and if the branch goes the other way, the slot instruction is simply wasted. The last alternative is to place a *no-op* instruction in the slot. Squashing relaxes the restriction on the second choice for instructions. It allows any instruction from the branch destination to be placed in the slot, even when there is an adverse effect if the branch goes the wrong way. The machine squashes the instruction (turns it into a *no-op*) if the branch goes the wrong way.

With squashing there are three options for dealing with the instructions in the delay slots giving three possible branch types: *no squash* where the slot instructions are always executed, *squash if don't go* where the slot instructions are executed if the branch takes and *squash if go* where the slot instructions are executed if the branch does not take. Since we decided to use static prediction, and in the static case most branches go, MIPS-X only has the first two types of branches. This requires only one bit in the instruction to specify how to deal with the instructions in the slots.

Various combinations of one and two-slot schemes with and without squashing were evaluated. The results are shown in Table 1. The *no squash* scheme is the same as used in MIPS where the instructions in the slots are always executed. The *always squash* scheme only uses the *squash if go* and *squash if don't go* actions for the instructions in the branch slots. The *squash optional* scheme includes the use of branches with *no squash* instructions in the slots as well as having branches with squashing. It can be seen that by allowing squashing the efficiency of branches is much better.

Branch Scheme	Cycles/Branch[2]
2-slot no squash	2.0
2-slot always squash	1.5
2-slot squash optional	1.3
1-slot no squash	1.4
1-slot always squash	1.3
1-slot squash optional	1.1

Table 1: Average Cycles per Branch Instruction for Various Branch Schemes

[2]If all of the branch delay slots could be filled with useful instructions, then we would achieve the ideal of a 1 cycle branch. Any no-op instructions in the branch delay slots are attributed to the cost of the branch so a branch with 2 no-ops in its two delay slots is deemed to have a cost of 3.

The scheme we finally chose uses the full compare and *squash optional* with two slots. Our initial estimates about the cost of the double slots turned out to be slightly optimistic. Where we predicted the average branch would take 1.3 cycles, results using the actual reorganizer showed that the average branch took about 1.5 cycles for small benchmarks using traditional optimization. However, we have since developed better optimization techniques and our most recent results show that even with large Pascal and Lisp benchmarks the average branch takes 1.27 cycles.

Implementing squashing was a gamble because we were not completely sure how it would affect exception handling at the time we made the commitment to use it. It turned out that they mesh together very well as described in the next section.

Exception Handling

As the design of the machine progressed, our concentration shifted from the functions the machine was going to perform to how these functions were going to be controlled. MIPS-X benefited greatly from the experience gained during the MIPS design. Handling exceptions in MIPS caused the most complexity in the machine because of the large number of possible states in the processor during an exception. These states were the result of the processor trying to complete the instructions that occurred conceptually before the fault but still in the pipeline, and reloading the partially full pipeline on a return from an exception. The goal for MIPS-X was to require as few states as possible to handle an exception so the state machine design would not be difficult. The underlining rule was to *keep it simple, stupid*[13].

In some ways exception handling in MIPS-X followed the MIPS model. Exceptions are not vectored so the exception handler must first determine the cause of the exception. On MIPS there was an on-chip *surprise register* where this information was stored. MIPS-X relies instead on a separate off-chip interrupt control unit that contains this information. The PSW does contain bits that determine whether the exception was caused by an interrupt, arithmetic overflow or a non-maskable interrupt.

MIPS-X differed from MIPS in how exceptions affected the pipeline. The MIPS exception sequence started with the pipeline being flushed of as many instructions as possible that were already executing. Then the program counter (PC) was zeroed and the return PCs saved from the PC chain. The flushing of the pipeline caused a great many extra states and added a lot of complexity.

In MIPS-X the pipeline is halted when an exception occurs. No instructions are completed. The PC is immediately set to zero and the shift chain of old PC values is frozen, saving the addresses of the instructions that are still in the pipeline. The current PSW is placed in PSWold, interrupts are turned off and the machine is placed into system mode. The exception routine, located at address zero in system space, begins execution by first saving the three PCs from the PC chain and PSWold onto the system stack. Once the state of the interrupted process is saved, then PC shifting can be enabled and interrupts unmasked if desired. The restart sequence involves reloading the PC chain with the three saved PCs and then doing three special jumps using the contents of the PC chain; the PC chain is used to store the

return addresses during the return sequence. Interrupts must be disabled both during machine state saving and restoring.

During the discussions about how branches were to be implemented, there was some concern about the effects the branch implementation would have on exception handling. The original feeling was that having more branch slots would require more state in the machine and implementing squashing branches would make the state machine even more complicated. The squash proponents argued that the hardware needed to freeze the pipeline during an exception could be used to implement squashing branches. They not only convinced the design team, they also turned out to be correct. Squashing two branch slots only requires a single extra input to the squashing finite state machine that is used to handle exceptions. Branch squashing and squashing for exceptions are very similar.

The general scheme used to no-op an instruction is quite simple. All that needs to be done is to set a bit in the destination specifier for that instruction. This bit is used by the register file to determine whether to perform a write or not. There are 2 lines in the machine that can set this bit, Exception and Squash. Exception no-ops the instructions in the ALU and MEM stages of the pipeline, while Squash no-ops the instructions currently in the IF and RF stages of the pipeline. The only added complexity occurs with the Mult/Div register and the PSW which contains the only visible state outside of the register file. Writes to these locations are also prevented by Exception and Squash.

There is only one exception generated on chip and it is a trap on overflow in the ALU or the multiplication/division hardware. At the start of the design it was felt that detecting overflows and generating a trap was too complex to do. The original solution was the concept of a *sticky overflow* bit. If an overflow occurred then the sticky overflow bit would be set in the PSW. This bit could then be checked at a later time to determine whether an overflow had occurred. This meant that it would not be possible to precisely detect the occurrence of the overflow but at least it was possible to indicate the presence of an incorrect result. We began looking for other overflow mechanisms when we discovered that the sticky overflow bit interacted badly with bypassing. Instead of making the hardware simple, it seemed to make the PSW harder to design.

Several other simple schemes were then proposed. One was a *SetOnAddOverflow* instruction that just routed the overflow bit from the ALU into the most significant bit of the ALU result. This instruction could then be used to determine whether the addition causes an overflow by simply testing for the sign of the result. Another suggestion was a *Branch on Overflow* instruction that caused a branch if the result of the branch comparison overflowed. These were minimal hardware solutions that would provide some small support for overflow detection.

At this point the exception hardware had been designed and we observed that generating a true *trap on overflow* was not difficult; in fact it was simpler than the original sticky overflow bit. We decided to abandon the sticky overflow bit for a maskable trap on overflow.

Control

Our overriding goal for the control section was to keep it as simple as possible. In part we accomplished our goal by eliminating hardware features that would complicate the machine without providing significant performance advantages. We also tried to keep a uniform view of the hardware, trying to reuse the same control mechanism for many features. Merging exceptions and squashing, and merging memory instructions and coprocessor operations were examples of this strategy. Finally, we eliminated the global controller for the machine and replaced it with a set of smaller controllers, one for each section of the datapath. We further partitioned the design so that a single designer was responsible for both the datapath and control in his section, giving each designer the incentive to make his control section simpler. Most of the machine control is simple decoders, many generated automatically using PLA generators.

One technique that MIPS-X used to great advantage was a qualified clock, called $\psi 1$, to latch the control state of the machine. This clock is the $\phi 1$ clock qualified with *not external cache miss* and *not internal cache miss*. When either cache misses, the $\psi 1$ clock does not rise, and the control state does not shift down the pipeline control latches. The lack of a $\psi 1$ clock causes the machine to execute the *previous* $\phi 2$ phase before retrying the $\phi 1$ phase. This simple technique made temporary stalling of the entire pipeline very easy, and allowed us to implement the late miss described earlier without greatly increasing the machine complexity. Since the $\psi 1$ clock is only allowed to clock control state latches, its pulse width can be quite narrow (about 10 ns). As long as the miss signal is monotonic, it is possible to detect a cache hit after the data has been latched in the machine without stalling the machine.

Together these control techniques were quite successful. The control was nicely divided among the 4 main datapath sections, with the only two finite state machines (FSMs) residing in the PC unit. These FSMs handle instruction cache misses and instruction squashing during exceptions and squashed branches. The state diagrams for the two machines are shown in Figures 3 and 4. These FSMs are implemented as simple shift registers with a very small amount of random logic and occupy less than 0.2% of the total area of the chip.

Status and Conclusions

The MIPS-X project began in earnest during the summer of 1984. By January 1985, we had settled on an initial version of the instruction set, and had written an instruction level simulator for the machine. We were able to use much of the software system that was created for MIPS for MIPS-X as well. This greatly reduced the software development effort. The compiler/simulator system generated instruction traces that we used to gather cache statistics and fine tune the architecture. By April 1985, the architecture had stabilized and work on the detailed design accelerated. We ran our first instruction through a detailed functional simulator of the entire processor during the summer. The final design was taped out at the end of April 1986 and we received first silicon back in October.

The processor was designed to run at a clock rate of 20

MHz, executing an instruction every cycle, yielding a peak performance of 20 MIPs. Timing analysis showed that the version that was shipped in April would run at about 16 MHz. Initial timing tests have shown that the part is fully functional and it runs at the projected 16 MHz clock rate. We are now fixing the critical paths so that we can achieve our goal of 20 MHz. The die is 8.5 mm by 8 mm and has a total of 108 pins of which 84 are for signals and 24 are for power and ground. There are about 150K transistors, two thirds of which are in the instruction cache. The power dissipation is less than 1 W.

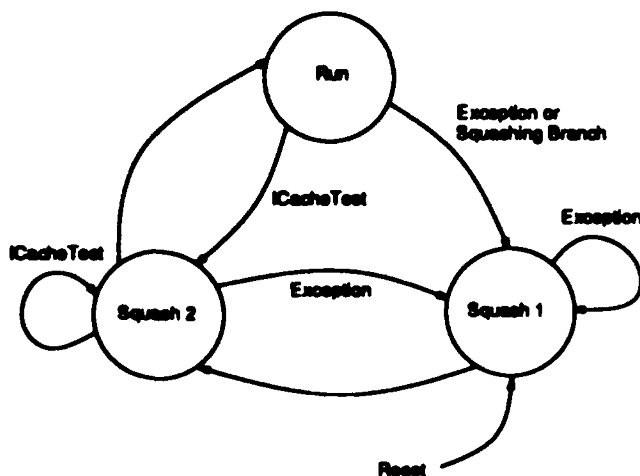


Figure 3: Squash Finite State Machine

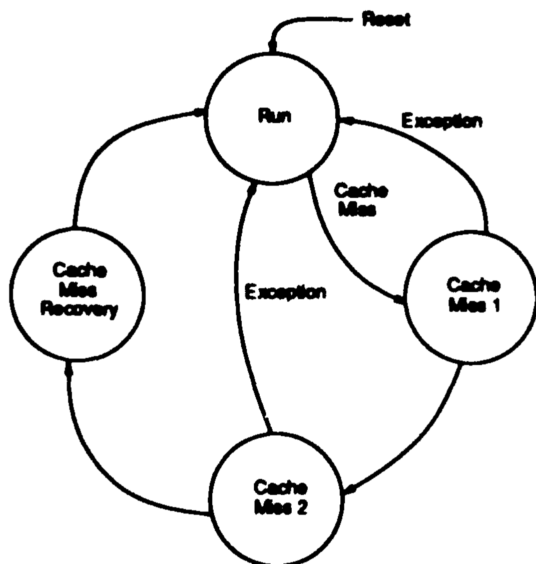


Figure 4: Cache Miss Finite State Machine

Simulations of our large Pascal benchmarks show that 15.6% of all instructions are no-ops due to unused branch delays or other pipeline interlocks that cannot be optimized away. For Lisp, this number increases slightly to 18.3% due to a larger number of jumps and many load-load interlocks caused by chasing car and cdr chains[14]. When the memory system overhead is included (delays from Icache and Ecache misses), the average instruction requires about 1.7 cycles meaning MIPS-X should have a sustained throughput above 11 MIPs. Our benchmark programs have static code sizes in the range of 50 KBytes to 270 KBytes so we cannot get exact numbers for the effects of the external cache because most of the benchmarks fit entirely. Smith's numbers[15] are not large enough so we used much larger traces[16] to derive the Ecache effects.

The performance of a machine is based on three factors: the number of instructions executed (path length), the number of cycles per instruction and the cycle time. Ideally, all three factors should be minimized but we have shown that by having simple instruction decode we can significantly decrease the latter two factors without adversely affecting the path length. Comparison of Pascal programs with a VAX 11/780 shows that MIPS-X executes about 25% more instructions but executes the programs about 14 times faster for unoptimized code. The static code size for MIPS-X is also about 25% greater than VAX code. The Stanford compiler system was used and the only difference was in the back end code generators. However, when MIPS-X code is compared to the Berkeley Pascal compiler, the path length is 80% longer and the speedup is only 10 times faster than the VAX. Much of this difference may be due to poorer code from our VAX code generator. We feel that when we get the results for optimized code, the numbers will be somewhere inbetween.

The goal of the MIPS-X project from the beginning was to learn from MIPS and design a simpler yet faster processor. The emphasis in all design decisions throughout the project was simplicity: minimize state and keep the control simple. The implementation of MIPS-X has shown that it is possible to implement a high performance microprocessor that supports coprocessors, without requiring complex control or hundreds of pins.

Acknowledgements

The MIPS-X research project has been supported by the Defense Advanced Research Projects Agency under contract #MDA903-83-C-0335. Paul Chow was partially supported by a postdoctoral fellowship from the Natural Sciences and Engineering Research Council of Canada.

Many people have contributed to the MIPS-X research effort. Malcolm Wing, Arturo Salz, Karen Huyser, Anant Agarwal, Scott McFarling, C.Y. Chu, Steve Richardson, Steve Tjiang, John Acken, Richard Simoni, Glenn Gulak, Kathy Cuderman, Takeshi Tokuda, Eugen Reithmann, Steven Przybylski, Chris Rowen, Norm Jouppi, Thomas Gross, John Gill and John Hennessy deserve special thanks for their contributions to the project.

References

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

2. D. Patterson and C. Sequin, "A VLSI RISC", *Computer*, September, 1982, pp. 8-21.

3. M. Katevenis, "Reduced Instruction Set Computer Architectures for VLSI", Computer Science Division (EECS) UCB/CSD 83/141, Univ. of CA at Berkeley, October 1983.

4. J. Hennessy, et al., "The MIPS Machine", *COMPCON*, IEEE, Spring 1982, pp. 2-7.

5. S. Przybylski, T. Gross, J. Hennessy, N. Jouppi, C. Rowen, "Organization and VLSI Implementation of MIPS", *Journal of VLSI and Computer Systems*, Vol. 1, No. 2, December, 1984, pp. 170-208.

6. Anant Agarwal, Paul Chow, Mark Horowitz, John Acken Arturo Salz and John Hennessy, "On-chip Instruction Caches for High Performance Processors", *Proceedings, Stanford Conference on Advanced Research in VLSI*, March 1987, pp. 1-24.

7. J.L. Hennessy, N. Jouppi, F. Baskett, T.R. Gross and J. Gill, "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.

8. Thomas Gross, *Code Optimization of Pipeline Constraints*, PhD dissertation, Stanford University, December 1983, Available as Stanford University CSL Technical Report 83-255.

9. John Hennessy and Thomas Gross, "Postpass Code Optimization of Pipeline Constraints", *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, July, 1983, pp. 422-448.

10. Scott McFarling and John Hennessy, "Reducing the Cost of Branches", *Proceedings, 13th Symposium on Computer Architecture*, June 1986, pp. 396-403.

11. J.E. Smith, "A Study of Branch Predition Strategies", *Proceedings, Eighth Symposium on Computer Architecture*, May 1981, pp. 135-148.

12. Johnny K. F. Lee, Alan Jay Smith, "Branch Prediction Strategies and Branch Target Buffer Design", *Computer*, January, 1984, pp. 6-22.

13. Butler W. Lampson, "Hints for Computer System Design", *IEEE Software*, Vol. 1, No. 1, January, 1984, pp. 11-30.

14. Peter Steenkiste, *LISP on a Reduced-Instruction-Set Processor: Characterization and Optimization*, PhD dissertation, Stanford University, 1987, To appear in 1987.

15. Alan Jay Smith, "Cache Memories", *Computing Surveys*, Vol. 14, No. 3, September, 1982, pp. 473-530.

16. Anant Agarwal, Richard L. Sites and Mark Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode", *13th Annual International Symposium on Computer Architecture*, IEEE, June 1986, pp. 119-127.

Special
A-1

END
8-87
DTIC