Proc. Tenth International Symposium on Computer Architecture, pp. 108-116 (June 1983).

ARCHITECTURE OF A VLSI INSTRUCTION CACHE FOR A RISC

David A. Patterson, Phil Garrison, Mark Hill, Dimitris Lioupis, Chris Nyberg, Tim Sippel, and Korbin Van Dyke

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

INTRODUCTION

A cache was first used in a commercial computer in 1968, [1] and researchers have spent the last 15 years analyzing caches and suggesting improvements. In designing a VLSI instruction cache for a RISC microprocessor we have uncovered four ideas potentially applicable to other VLSI machines. These ideas provide expansible cache memory, increased cache speed, reduced program code size, and decreased manufacturing costs. These improvements blur the habitual distinction between an instruction cache and an instruction fetch unit.

RISC stands for Reduced Instruction Set Computer, an architectural philosophy promising higher performance using simpler hardware. [2] Examples of RISC's are the IBM 801, [3] the Berkeley RISC I, [4] and the Stanford MIPS. [5] RISC I is the first by-product of a new graduate curriculum in which students propose and evaluate architectural concepts, learn Mead/Conway VLSI design, form teams to build the system, and then test their design. This 44,500 transistor integrated circuit has one minor design error, worked on the first good silicon, and runs programs faster than commercial microprocessors. [6]

The by-product of the second offering of these courses is a VLSI instruction cache. A cache is a high-speed buffer between main memory and the CPU. Each entry in the cache contains an *address tag* as well as buffered data in the cache *block*. Each memory access is mapped to a cache entry; the comparison of the address to the tag determines if there is a *hit*. The popular organizations either map a memory word onto only one cache block (*direct mapped*), allow a memory word to map onto any cache block (*fully associative*), or map a memory word onto a small number of blocks in the cache (*set associative*). All caches also need a bit per block to tell whether the data in the block is valid or not. Generally, cache invalidations occur only on startup and on process switches. We will assume that the cache receives non-virtual addresses, although these ideas work well with a virtual address cache. Writing causes difficulties in caches, but an instruction cache avoids those problems because code is read-only. An excellent survey of cache issues was written by Smith. [7]

The instruction cache is designed to work with a more efficient implementation of RISC I, called RISC II. [8] RISC II has more registers, uses less area, and should have higher performance than RISC I. It also has the minor changes needed for this instruction cache.

Our long-term goal is to design a single chip that combines the CPU with an instruction cache, as this combination reduces off-chip references. The silicon processing available to us precludes an on-chip cache. Thus this separate cache chip should be considered a research prototype rather than a potential commercial product.

The next four sections present the four architectural ideas, followed by a section on performance evaluation of each idea. We then describe the implementation of the cache and finally summarize the results.

Interestingly, the first architectural idea is possible only because we cannot fit the cache and the CPU on the same chip.

EXPANSIBLE CACHE

Designers must select a cache memory size large enough for a wide range of applications and small enough so most customers can afford it. Just as a customer buys varying amounts of main memory depending on his budget and application, he should be able to buy only the amount of cache he needs. Custom VLSI designs differ from traditional hardware in their high

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.

design costs but low replication costs. Thus, the ideal VLSI cache would consist of one or more identical chips, the number being determined by the desired cache size.

The RISC II instruction cache is such a chip. The controller of each chip just checks the ***chip select*** line to see if it is enabled; if so, it services the memory access. This approach requires a cached memory location appear only in one chip. An external decoder generates the chip-select signals, implementing a direct mapped cache — a word is always cached in the same chip (Figure 1).

A simple improvement is to allow a word to appear in any chip. Every chip tries to service each address request~~, but o~~nly the chip with the matching address supplies the data. To assure that the same data does not appear in more than one chip at the same time, a token identifies the chip that fetches the data from memory when there is a miss. The token is then passed to the next cache chip. The implements a set associative cache, with the number of sets equal to the number of chips. In addition to giving higher hit ratios (due to fewer restrictions on locations of cacheable data), this approach works with any number of chips, whereas direct mapping works best with 2^i chips.

Associative mapping requires a smarter cache controller, a bit of memory to hold the token, a few pins to pass the token, plus one pin to determine where to start the token on system initialization. Figure 2 shows associatively-mapped cache chips connected in a ring. Note that in general a set associative cache can be expanded either wider, by increasing the number of sets, or deeper, by increasing the number of blocks per set. The former is more general and thus has a higher hit rate at the cost of more comparators. Since a compartor must be included in each chip, Figure 2 uses the wide option.

A fully associative mapping over multiple cache chips, where a memory word could be located in any block on any chip, is achieved by the organization in Figure 2 if each cache chip uses fully associative mapping internally.

FAULT TOLERANCE

The semiconductor industry is sometimes called the "jelly bean" industry because of its high volume manufacturing techniques. Usually a majority of the chips manufactured at such a rate are faulty. Chips are discarded even if only one transistor fails out of tens of thousands. Rather than dwell on failure, manufacturers accentuate the positive by measuring *yield*. Yield is a function of the size of the chip and the density of the circuits inside the chip. The yield of the chip affects the manufacturing costs and, indirectly, limits the size of the chip.

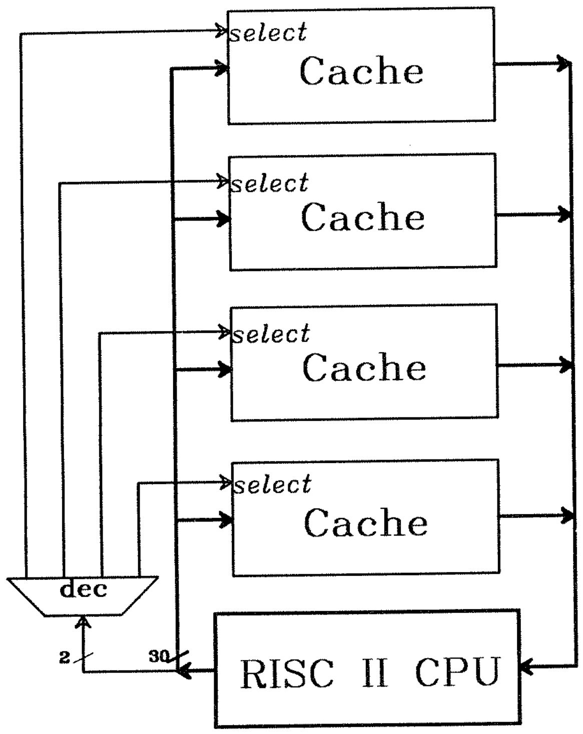

Figure 1. Direct Mapped Cache Chips.

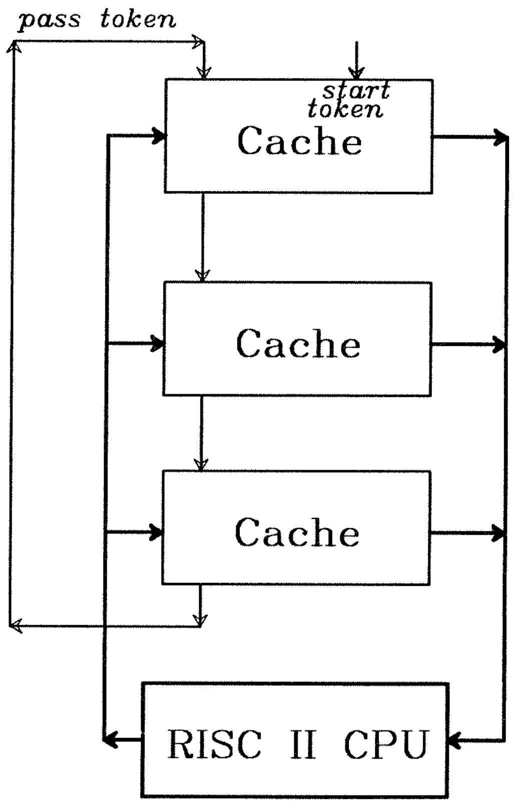

Figure 2. Associatively Mapped Cache Chips.

The RISC II cache includes a minor architectural change that improves yield. Most of the chip area comprises the memory for the address tag and cache data (illustrated in Figure 3). As mentioned above, each cache block must have an invalid bit. Alan Paeth of Xerox PARC suggested we add a second invalid bit, the ***fault tolerant*** bit, that can permanently invalidate a cache block. The result of an access to a bad block is simply a miss; the data is then supplied from main memory. Each chip would be tested to determine functioning words, and on power up, each chip would be loaded with the proper fault tolerant bits. We chose to provide initialization by connecting the fault tolerant bits as a shift register, so the fault pattern is shifted in one bit per block.

Bad blocks effect performance depending on the mapping strategies. In fully associative mapping an N word cache with a single bad word becomes an N-1 word cache, which for practical purposes is as effective as before. At first glance direct mapping should not work with a chip with "broken" cache blocks; memory words can be found only in one cache entry, so a broken cache block means the corresponding data would never be in the cache. However, because caches improve performance statistically rather than absolutely, a direct mapped cache can still perform acceptably even if it consistently misses on a few addresses. Moreover, multiple cache chips connected associatively will lock out addresses only if all chips have a defect in the same word.

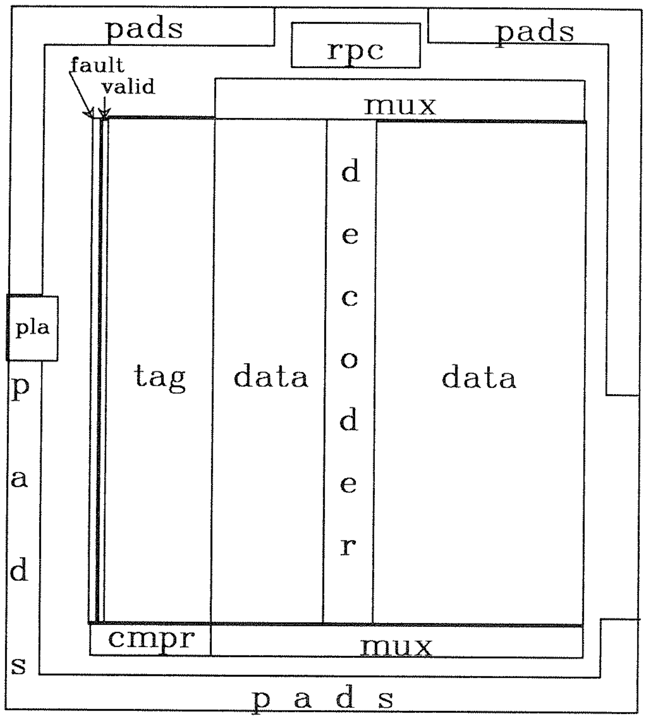

Figure 3. Chip Plan of RISC II Cache.

Note that this redundancy does not cover all single bit memory errors. Chips with faults in the fault tolerant or valid bits will be discarded. Also, we have relied on the availability of a path around the instruction cache to access memory when we have a fault. Some cache organizations require blocks to be loaded into the cache before they are read by the CPU, and are incompatible with our ideas for direct mapped caches.

REMOTE PROGRAM COUNTER

One design goal of VLSI machines is to reduce off-chip accesses. This goal motivated Manolis Katevenis' SAMOS architecture, [9] in which copies of pointer-registers are kept with the memory chips, rather than only in the CPU. The best example of such a pointer is the Program Counter (PC). Subsequent values of the PC – sent out of the CPU over the address bus – contain little new information, since they are usually just the previous value plus a small offset. The RISC II instruction cache uses a *Remote Program Counter (RPC)*, inspired by the SAMOS architecture, to take advantage of the predictable nature of PC addresses.

Figure 4 shows the timing of the RPC cache compared to a normal cache. It nearly doubles the time available to read the cache data and address tag, the critical path of a cache.

Predicting the next PC value is easy for sequential execution, but obviously the next instruction address is not always the next sequential address. Rather than compare the tag to the RPC, every cache access compares to the real PC to assure access of the correct word. If there is no match, we take another full cycle to see if the correct address is in the cache. Thus the number of cache misses as well as the number of mispredicted addresses affect the performance of the RISC II instruction cache.

There are two reasons for non-sequential addresses – interrupts and jumps. Interrupts are completely unpredictable but, fortunately, they are infrequent. Jumps, on the other hand, are predictable. RISC I and RISC II include PC-relative addressing for jumps and calls, accounting for almost all non-sequential addresses. Since there is a copy of the PC on the cache chip, we can calculate jump addresses from the instruction in the chip. If it is a jump, then we add the offset to the RPC as the new predicted address. Thus the cache properly predicts sequential instruction accesses and unconditional PC-relative jumps and calls. This RPC requires the addition of a register, adder, and multiplexer to a cache chip.

Conditional jumps are another matter. Designers have invented many ways to predict conditional jumps, and any would work with this proposal. [10, 11] We have defined an unused bit of the jump instruction to help predict conditional jumps. This bit is named *likely*, meaning the RPC will be loaded with with the jump address if the bit is set and loaded with the next sequential address otherwise. This allows the compiler to give clues to the cache. [12] For example, the branches associated with loops would be marked likely while branches that exit a loop would be marked unlikely. We have not yet modified the compiler to supply this information so we always set the likely bit.

COMPACTION

In RISC I all instructions are 32-bits, simplifying instruction fetch and decode, thereby improving the performance of RISC I. Many have argued that smaller programs mean better performance. Although no experimental evidence has been presented to support this claim, the arguments are that smaller programs mean fewer cache faults and fewer page faults. Rather than add variable length instructions to RISC II CPU, Phil Garrison and Korbin Van Dyke investigated uses of the cache to transform compact code from main memory into 32-bit RISC I instructions. [13]

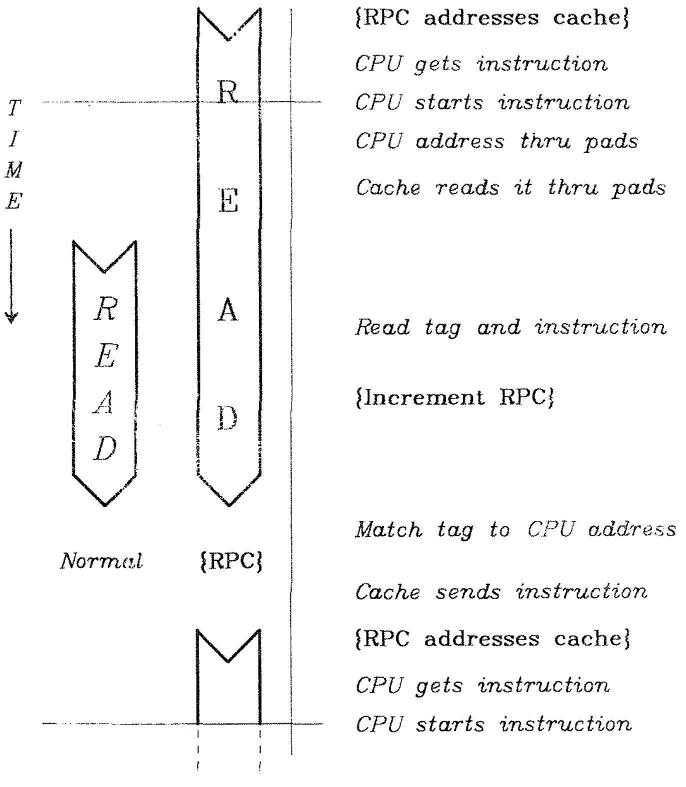

Figure 4. This timing diagram compares a normal cache to the RPC cache. The arrow on the left illustrates the time available for a read with a normal cache. The words on the right side of the figure in *italics* show the steps that occur when sending an instruction to the CPU in the normal case. The arrow on the right shows the increased time for reading the tag and data with the RPC cache. The words on the right in in curly brackets and **bold** type show the extra steps that occur for this case.

Having the cache handle variable length instructions has the potential of maintaining the high performance of simple instruction fetch and decode while obtaining the potential advantages of reduced code size in fewer cache and page faults. A similar idea has been used by the S-1 Mark II. It increases the instruction cache entry from 36 to 56 bits to reduce instruction decoding time.

This raises three questions:

1. Should instruction lengths be bit-variable or byte-variable?
2. Should instructions be expanded 'on-the-fly' between the cache and the CPU, or 'on-the-miss' between cache and memory?
3. What can be done to avoid the problems caused by variable length instructions that span cache and page boundaries?

1. Instruction Length. The first decision involves density versus simplicity. Huffman encoded, bit-variable instructions will result in the smallest programs, but at what cost? Huffman encoding of the RISC I instruction set resulted in instructions varying from 4 bits to 67 bits. As each field of the instruction is separately encoded in 2 to 17 bits (see Table 1), practical implementations of instruction expansion required one clock step per field. This means 5 clock steps for most RISC I instructions. Bit variable instructions imply a bit-addressable PC in the CPU plus the ability to add bit offsets of 4 to 67 to access the next instruction. The cache/CPU interface then needs 7 ($\log_2 67$) more bits to pass the correct PC bit increment along with the expanded 32-bit instruction. The wider PC affects branch instructions – either the offsets are considered bit offsets, reducing the addressing range by a factor of 8, or branch targets are padded to byte boundaries.

An alternative is instructions lengths that are multiples of bytes, removing the problems of the PC and branches. Garrison and Van Dyke proposed 11 new instructions and 5 new instruction formats to support 8-, 16-, and 24-bit instructions (see Table 2). Most new instructions occurred by having the opcode specify the use of register 0, a register that always contains a hardwired 0. The cache still needs to tell the CPU the size of the instruction, but only 2 bits are necessary to distinguish 8-, 16-, 24-, and 32-bit instructions.

2. Expansion. Figure 5 illustrates the two alternatives on instruction expansion. The on-the-fly scheme has the disadvantage of expanding the instructions during the access, lengthening the cache cycle. Since the main purpose of cache is fast access, long expansion time means an ineffective cache.

Table 1. Huffman encoding sizes

Field	Length (bits) Min	Max	Avg	(RISC I)	Avg Freq per Instr	Savings
Opcode	3	17	4.02	(8)	1	12.5%
No Immediate	0	-	-	(8)	.25	10.7%
Shorter Constants	5	9	6.9	(13/18)	.67	8.4%
Register Specification	2	14	4.1	(5)	1.98	6.1%
Condition Code Field	2	11	2.4	(5)	.15	1.2%
Operand Class in Opcode	0	0	0	(1)	.44	1.4%
Fewer Operands (# regs)	0	3	1.3	(1/2/3)	-	8.4%
Byte Alignment Padding	1	7	4	(0)	.20	-2.5%
Byte Alignment Start Bit	1	1	1	(0)	1	-3.1%
Total Savings						43%

Table 2. Original Proposal for Byte-Variable Short Instructions

Special Instruction ($const < \pm 2^8$)	RISC I equivalent ($r0 \equiv 0$)	Length (bits)	% Savings
move rN,rM	**add** rN,r0,rM	16	7.1%
jmprs *const*	**jmpr** *const*	16	5.2%
ldshi rN,*const*	**ldhi** rN,*const*	16	3.9%
clr rN	**add** r0,r0,rN	8	3.9%
nop	**add** r0,r0,r0	8	3.9%
compare rN,rM	**sub** rN,rM,r0,{c}	16	3.7%
rets rN	**ret** 8(rN)	8	1.7%
moveimm *const*,rN	**add** r0,*const*,rN	16	1.7%
add1 rN	**add** #1,rN,rN	8	1.5%
noimmop rN,rM,rP	**noimmop** rN,rM,rP	24	1.1%
add2imm *const*,rN	**add** *const*,rN,rN	16	1.0%
TOTAL			34.8%

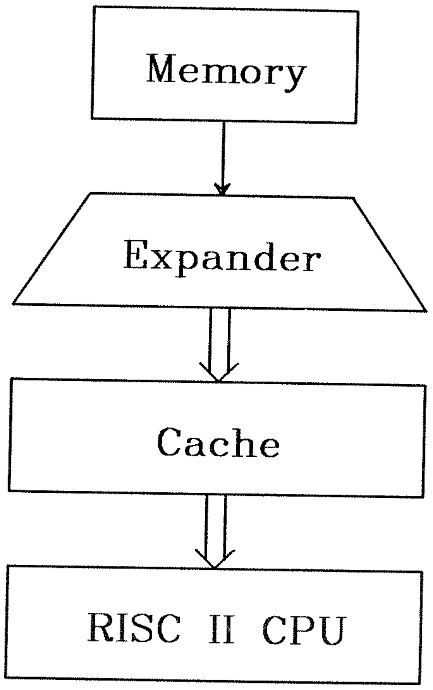

Figure 5(a). Expander using 'On-the-miss' Expansion.

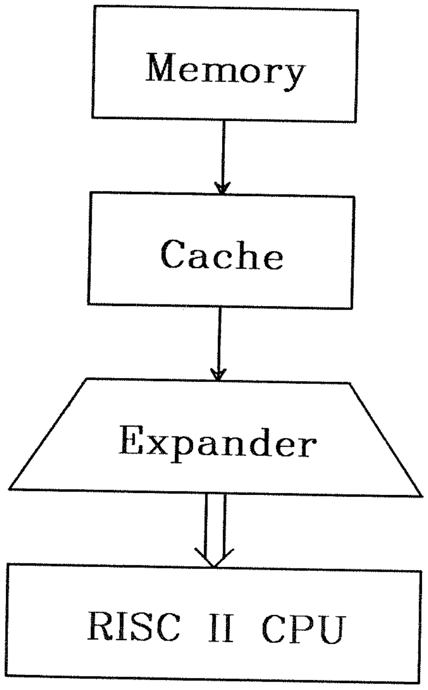

Figure 5(b). Expander using 'On-the-fly' Expansion

In on-the-miss expansion, an instruction in memory changes size when in the cache. Thus the cache must be able to store a full 32-bit expanded instruction for every location that can contain an instruction. The straightforward implementation for byte-variable instructions is an on-the-miss cache memory four times the size of an on-the-fly cache, with most of that extra memory usually empty.

A practical solution to the problem of a cache that is four times the size of an on-the-fly cache is to limit each on-the-miss cache entry to a single instruction, instead of two. Tables 1 and 2 indicate that compact programs are approximately 40 percent (35 to 43 percent) smaller than expanded programs. We must therefore increase the number of cache entries by 40 percent to hold the same number of instructions as the on-the-fly cache. This solution introduces a further cost: there is only a single instruction per address tag in the cache, instead of two instructions per tag, so fewer instructions can share the address tag portion of the cache. For the 32-bit RISC I address, an on-the-miss cache containing the same number of instructions as an on-the-fly cache of 64 blocks by 64 bits would need

$$\frac{1.4 * 128 * (26+32)}{64 * (23+64)} = 1.8$$

times more memory for tags and data. That is, a new cache would contain 128 entries, each consisting of 26 bits of address tag and 32 bits of data memory per entry, instead of 64 entries consisting of 23 bits of address and 64 bits of data per entry. The cache also needed to be 40 percent larger, as noted above, to accommodate the equivalent number of instructions.

Tables 1 and 2 show that byte-variable instructions reduce code size 35 percent while bit-variable instructions condense code only 8 percent more. The complexity of bit-variable instructions and the enlarged memory of the on-the-miss cache led Garrison and Van Dyke to recommend byte-variable instructions using an on-the-miss expansion strategy.

3. Crossing Boundaries. Both schemes have instructions that can span page boundaries; thus, unlike RISC I, a valid starting PC address does not guarantee uninterrupted instruction fetch. Variable length instructions also allow instructions to cross cache blocks. As in the paging problem, the cache no longer always delivers a complete instruction if the PC points to a location in the cache.

This difficulty can be avoided by requiring the assembler to prevent instructions from spanning cache blocks. Since the page size is a multiple of cache blocks, this also means that instructions cannot span page boundaries, avoiding faults during an instruction fetch. The consequences of that restriction is to pad sequences with NOP's similar to the way 15-bit and 30-bit instructions pack into 60-bit words on the CDC 6600. In addition to reducing code density, this approach reduces performance because extra unnecessary NOP instructions are executed.

EVALUATION

This section evaluates the performance of each of the four new ideas. RISC II is designed to execute an instruction every 400 ns leaving about 200 ns to fetch an instruction after the address is valid. The performance of each idea is measured by its impact on the effective access time.

In each case we have measured RISC II executing the Portable C Compiler, the largest C program that we can run on our cache simulators. We assume that a single chip is direct mapped and contains 64 blocks with 64 bits per block. A miss takes 1400 ns with a 600 ns main memory.

Expansibility. Table 3 below shows how the miss ratio decreases as we expand the number of chips for the cache. Note that the number of chips are doubled with each line in the table.

Table 3. Miss fraction for 2 Mappings

Number of Chips	Miss Fraction Associative	Direct	Ratio D/A
1	.124	.124	1
2	.105	.099	1.06
4	.078	.082	1.05
8	.050	.064	1.28

Fault Tolerance. We can calculate the improvement in yield due to the fault tolerant bit by using a formula to predict yield. AMI forecasts the net dies per wafer (N) based on the gross number of dies on the wafer (G), the area per die (A), the defect density (D), and number of critical mask layers (n) using the formula[14]

$$N = \frac{G}{(1 + A * D)^n}$$

Typical defect densities are 2 to 8 per square inch. Using the AMI formula, the normal number of dies for RISC II cache assuming a 4-inch wafer, 5 critical layers, and 4 defects per square inch is

$$\frac{118}{(1+(.275)(.285) * 4)^5} = 30$$

dies, or 25% yield.

Memory is 52 % of the area of the RISC II cache chip (Figure 3). Recalculating the formula assuming that only 48 % of the area can have defects predicts 58 dies. All flaws in the memory area cannot be corrected. If, for example, we can recover from two-thirds of the flaws then the formula predicts 50 dies, an increase of 67 %. With 2 defects per square inch the improvement is 30 % and 135 % for 8 defects per square inch. Furthermore, this memory, the densest part of the chip, corresponds to more than 85 % of the transistors. We believe the rest of the area is less sensitive to flaws, and thus expect an even greater manufacturing improvement.

The VLSI consequences of the fault tolerant bit are minor. (We included this feature on the chip within two days after Paeth's suggestion.) The extra invalidate line increases the size of the RISC II cache by 1 percent.

Remote Program Counter. Table 4 shows the fraction of instructions that are successfully predicted using the Remote Program Counter. There are 4 small benchmarks plus the large Portable C Compiler.

Table 4. Jump Prediction Accuracy.

acker(2,6)	.894
benchE	.912
sieve	.915
towers(9)	.875
PCC compiling benchE	.925

The next instruction address is correctly predicted 90 % of the time. When the instruction cache mispredicts it takes an extra 400 ns CPU cycle to fetch the correct instruction. Thus we can can approximately double the time for the reading the cache data at a cost of only 10 percent performance reduction.

The high prediction fraction suggests the use of RPC provided it does not need too much area. At first glance it would seem we would need a 32-bit adder and 32-bit register to implement RPC. Because we compare the cache tag to the PC address coming from the CPU rather than from the RPC, the RPC need only be large enough to address the on-chip cache memory. The RISC II cache RPC is 7 bits, and less than 3 percent of the area of the chip.

Compaction. The "less is more" philosophy that guided RISC has also guided the design of the RISC II cache. We simplified the expansion phase by limiting RISC II to only one short instruction format. Table 5 shows that this 16-bit format still reduces code size 30 %. The percentage savings shown in Tables 1, 2, and 5 is based on the average static frequency of instructions in a half-dozen large RISC I programs. [13] The seven 16-bit instructions in Table 5 were the most popular instructions in that study, and required most of the opcodes in the single new format. The rest of the candidates for short instructions reduced code density by less than 1 percent or required another instruction format.

As mentioned above, we can solve the block and page crossing problem with the assembler. We can avoid padding with NOP's by remembering that every 16-bit instruction has a corresponding 32-bit instruction that is just as fast. Using the 32-bit form of the final 16-bit instruction in the cache block eliminates the NOP. Figure 6 shows the packing of instruction sequences for a 64-bit cache block.

Old Sequence		(Size)	New Sequence		(Size)
MOVE	Ra,Rb	(16)	ADD	Ra,#0,Rb	(32)
LOAD	Ra,X	(32)	LOAD	Ra,X	(32)
LOAD	Rb,Y	(32)	LOAD	Rb,Y	(32)
LOAD	Ra,X	(32)	LOAD	Ra,X	(32)
MOVE	Ra,Rb	(16)	ADD	Ra,#0,Rb	(32)
LOAD	Rb,Y	(32)	LOAD	Rb,Y	(32)
MOVE	Ra,Rb	(16)	MOVE	Ra,Rb	(16)
MOVE	Rc,Rd	(16)	MOVE	Rc,Rd	(16)
MOVE	Re,Rf	(16)	ADD	Re,#0,Rf	(32)
LOAD	Ra,X	(32)	LOAD	Ra,X	(32)

Figure 6. Padding 64-bit Cache Blocks with 32-bit versions of the short instructions. With an even number of short instructions no padding is necessary. The *MOVE* instruction is 16-bit and *LOAD* is 32-bits. The 32-bit version of *MOVE Ra, Rb* is *ADD Ra, #0, Rb*.

The increase in code size due to the restriction that instructions cannot cross block boundaries can be estimated by looking at all combinations of 16-bit and 32-bit instruction sequences and finding what fraction of sequences require padding. For a 64-bit block, 5 sequences out of 16 will not fit. Thus on the average we would expect

Table 5. RISC II Short Instructions

Special Instruction (*const* < $\pm 2^8$)	RISC I equivalent (r0 ≡ 0)	Length (bits)	% Savings
move rN,rM	**add** rN,r0,rM	16	12.3%
jmprs *const*	**jmpr** *const*	16	5.2%
ldshi rN,*const*	**ldhi** rN,*const*	16	3.9%
compare rN,rM	**sub** rN,rM,r0,{c}	16	3.7%
add2imm *const*,rN	**add** *const*,rN,rN	16	2.0%
moveimm *const*,rN	**add** r0,*const*,rN	16	1.7%
rets *const*(rN)	**ret** *const*(rN)	16	1.2%
TOTAL			30.0%

$$\frac{\frac{5}{16} * 16}{64}$$

or 7.8 % padding. The assumption of random distribution with 128-bit blocks yields 3.9 % padding. We measured the impact on real programs by scanning through thousands of lines of RISC II assembly language and found 7.6% and 3.8% padding for 64- and 128-bit blocks, respectively. A more sophisticated scanner could check semantics of code sequences to re-order instructions to improve packing.

Such padding increases code size, but we still expect RISC II programs to be 20 to 25 % smaller than RISC I using compacted instructions that do not cross cache boundaries. Since RISC I C programs are 10 to 30 % larger than the PDP-11, [4] this should make RISC II programs about the same size as PDP-11.

Compaction also improves cache performance, as shown in Table 6. (By expanding cache blocks from 8 bytes to 11 bytes we have the same number of instructions in the simulated cache as would be in a cache with compacted instructions.) The 30% reduction in code size not only increases hits, it increases the bandwidth of the cache bus by bringing more instructions on each miss.

Table 6. Miss fraction for Compacted Cache.

Number	Miss Fraction		Ratio
of Chips	compacted	uncompacted	U/C
1	.092	.124	1.35
2	.078	.105	1.35
4	.056	.082	1.46
8	.040	.064	1.60

Short instructions have two VLSI consequences: (1) the cache must supply 16-bit as well as 32-bit data, and (2) the expansion logic and time delay become a problem. Limiting to one 16-bit format simplifies expansion hardware, and we hope the comparison of the tag to the real address during expansion hides this delay.

IMPLEMENTATION

Mark Hill, Dimitris Lioupis, Chris Nyberg, and Tim Sippel implemented the RISC II cache chip in 4 micron NMOS using Mead/Conway design style. [15] This 46,500 transistor chip implements cache expansibility, fault tolerant bit, Remote PC, and 16-bit instructions. This chip uses a very conservative 7-transistor semi-static memory cell, whose size limits us to blocks each containing 64-bits. The chip uses direct mapping to manage internal cache blocks, allowing us to use the same memory cell for tag and data.

The only idea presented in this paper that was not implemented is the hardware that expands the 16-bit short instructions into the normal 32-bit RISC instructions.

The project began April 1982 and design was completed in August. The chip is currently being fabricated. Figure 3 shows the chip plan of the cache.

SUMMARY

We believe that the four main ideas in the RISC II instruction cache are of general interest to computer designers. The fault-tolerant bit should at least double the yield of cache at an increase of only 1% of the chip area. The Remote PC predicts 90% of all instruction addresses effectively providing a fast cache from slow memory elements for only 3 % of the area. Given our block size and bus, loading the cache with compact instructions significantly improves the hit ratio and the memory-cache bandwidth. The CPU can still execute easy-to-decode RISC I instructions and the cache and memory can contain programs that are 20% to 25 % smaller. There is clearly a spectrum of decoding opportunities from all in the CPU to all in the cache; we believe the partial decoding in the cache to deliver equal sized instructions will be of interest to other designers. A minor architecture change to an off-chip cache allows several chips to be connected to a CPU thereby improving the hit ratio.

ACKNOWLEDGMENTS

As mentioned above, this cache was designed as part of CS 292R, the Experimental Architecture class at Berkeley, and implemented from scratch as part of CS 292X, the VLSI Systems course. A preliminary version of the cache was designed by David Buchanan, Gordon Hamachi, Mark Hill, Tom Quarles, and Barry Roitblat in CS 248, the VLSI layout class taught by John Ousterhout. We would like to thank everyone involved in those classes for their help.

People not involved in those classes helped our design. The Remote Program Counter was inspired by Manolis Katevenis' SAMOS architecture. Gaetano Borriello and Alan Paeth of Xerox PARC reviewed the VLSI design of the cache chip, and Paeth suggested the fault tolerant bit during that visit.

Several people gave valuable suggestions that improved the quality of this paper: Barbara Borske, Doug Clark, Helen Davis, Dave Ditzel, Jim Goodman, Paul Hansen, Manolis Katevenis, John Ousterhout, and Carlo Séquin.

This research was sponsored by Defense Advance Research Projects Agency (DoD), ARPA Order No. 3803, monitored by Naval Electronic System Command under Contract No. N00039-81-K-0251.

References

1. J.S. Liptay, "Structural Aspects of the System/360 Model 85, Part II: The Cache," *IBM Systems Journal* **7**(1) pp. 15-21 (1968).
2. D.A. Patterson and D.R. Ditzel, "The Case for the Reduced Instruction Set Computer," *Computer Architecture News* **8**(6) pp. 25-33 (15 October 1980).
3. G. Radin, "The 801 Minicomputer," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 39-47 (March 1-3, 1982).
4. D.A. Patterson and C.H. Séquin, "A VLSI RISC," *Computer* **15** (9) pp. 8-21 (September 1982).
5. J. Hennessy, N. Jouppi, F. Baskett, A. Strong, T. Gross, C. Rowen, and J. Gill, "The MIPS Machine," *Proc. Compcon*, (February 1982).
6. J.K. Foderaro, K.S. Van Dyke, and D.A. Patterson, "Running RISC's," *VLSI Design* **III**(5) pp. 27-32 (September/October, 1982).
7. A.J. Smith, "Cache Memories," *Computing Surveys* **14**(3) pp. 473 - 530 (September, 1982).
8. M.G.H. Katevenis, R.W. Sherburne, D.A. Patterson, and C.H. Séquin, "The RISC II Micro-Architecture," *Submitted to the VLSI 83 Conference*, (August 1983).
9. M. Katevenis, *SAMOS: a SmArt MemOry computer System (outline of first general ideas)*, U.C.Berkeley Internal Working Paper June 1981.
10. D. Morris and R.N. Ibbett, *The MU-5 Computer System*, Springer-Verlag, 1979.
11. J.E. Smith, "A Study of Branch Prediction Stratagies," *Proc. Eighth International Symposium on Computer Architecture*, pp. 135-148 (May 1981).
12. D.R. Ditzel and D.A. Patterson, "Retrospective on High-Level Language Computer Architecture," *Proc. Seventh Annual International Symposium on Computer Architecture*, pp. 97-104 (May 6-8, 1980).
13. P. Garrison and K.S. Van Dyke, *Compact RISC*, CS292R Final Report December 6, 1981.
14. S. McMinn, "Semiconductor Manufacturing for VLSI Designers," *VLSI Design* **III**(4)(July/August 1982).
15. M. Hill, D. Lioupis, C. Nyberg, and T. Sippel, *RISC Cache Project: Final Report on Architecture and Implementation of a VLSI Cache Chip*, CS292X Final Report