RISC Architectures

Norman Matloff
University of California at Davis

February 27, 2002

Contents

1 Introduction

The term **RISC** is an acronym for reduced instruction set computer, the antonym being **CISC**, for complex instruction set computer.

During the 1970s and early 1980s, computers became more and more CISC-like, with richer and richer instruction sets. The most cited example is that of the VAX family (highly popular in the early 1980s), whose architecture included 304 instructions, including such highly specialized instructions as **poly**, which evaluates polynomials. By contrast, the UC Berkeley RISC I architecture, which later became the basis for what is now the commercial SPARC chip in Sun Microsystems computers, had only 32 instructions.

At the same time, the single-chip CPU was starting out on its path to dominance of the market. The more components a computer has, the more expensive it is to manufacture, so single-chip CPUs became very attractive.[1] But the problem then became that the very limited space on a chip—this space is colloquially called "real estate"—made it very difficult, if not impossible, to fit a CISC architecture onto a single chip.[2]

The VAX was a clear casualty of this. When the Digital Equipment Corporation (DEC) tried to produce a "VAX-on-a-chip," they found that they could not fit the entire instruction set on one chip. They were then forced to implement the remaining instructions in software. For example, when the CPU encountered a **poly** instruction, that would cause an "illegal op code" condition, which triggered a **trap** (i.e. internal interrupt); the trap-service routine would then be a procedure consisting of ordinary **add** and **mul** instructions which compute the polynomial.

At this point, some researchers at IBM[3] decided to take a fresh look at the whole question of instruction set formulation. The main point is optimal use of real estate. In a CISC CPU chip, a large portion of the chip is devoted to circuitry which is rarely, if ever used.[4] For example, most users of VAX machines did not make use of the **poly** instruction, and yet the circuitry for that instruction would occupy valuable-but-wasted real estate in the chips comprising a VAX CPU.

True, for those programs which did use the **poly** instruction, execution speed increased somewhat,[5] but for most programs the **poly** instruction was just getting in the way of fast performance.

Other factors came into play. For example, the richer the instruction set, the longer the decode portion of the instruction cycle would take. Even more important is the issue of pipelining, which is much harder to do in CISC machine, due to lack of uniformity in the instruction set (in a CISC machine, instructions tend to have different lengths).

For this reason, the IBM people asked, "Why do we need all this complexity," and they built the first RISC machine. Unfortunately, IBM scrapped their RISC project, but later David Patterson of the Division of

[1] And there is a performance gain as well. Multi-chip CPUs suffer from the problem that off-chip communication is slower than within-chip.

[2] It would seem that the solution is to simply make larger chips. However, this idea has not yet worked. For example, larger chips have lower **yield** rates, i.e. lower rates of flaw-free chips. Of course, one can try to make chips denser as well, but there are limits here too.

[3] Thoug Professor Patterson at UC Berkeley is the one who really popularized the RISC idea, it was actually invented at IBM first. Actually, the old CDC machines designed by Seymour Cray had many RISC characteristics, i.e. small instruction sets and load/store architecture.

[4] Actually, most CISC chips use a technique called **microcode**, which would make the use of the word "circuitry" here a bit overly simplistic, but we will not pursue that here.

[5] You should think about this. First, **poly**, being a single instruction, would require only one instruction fetch, compared to many fetches which would be needed if one wrote a function consisting of many instructions which would collectively perform the work that **poly** does. Second, at the circuitry level, we might be able to perform more than one register transfer at once.

Computer Science at UC Berkeley became interested in the idea, and developed two RISC CPU chips, RISC I and RISC II, which later became the basis of the commercial SPARC chip, as mentioned earlier.

Patterson's team did not invent the RISC concept, nor did they invent the specific features of their design such as delayed branch (see below). Nevertheless, these people singlehandedly changed the thinking of the entire computer industry, by showing that RISC could really work well. The team considered the problem as a whole, investigating all aspects, ranging from the questions of chip layout and VLSI electronics technology to issues of compiler writing, and through this integrated approach were able to make a good case for the RISC concept. Manolis Katevenis, a Ph.D. student who wrote his dissertation on RISC under Professor Patterson, won the annual ACM award for the best dissertation in Computer Science.[6]

Today virtually every computer manufacturer has placed a major emphasis on RISC. Sun Microsystems has the SPARC chip, SGI uses MIPS, IBM and Motorola have the Power PC chip series, and so on. (On the other hand, this does not imply that RISC is inherently "superior," and as electronic technologies advance in the future, there may well be some resurgence of CISC ideas, at least to some degree.)

2 A Definition of RISC

There is no universally-agreed-upon definition of RISC, but most people would agree at least to the following characteristics:

- Most instructions execute in a single clock cycle.
- "Load/store architecture"—the only instructions which access memory are of the LOAD and STORE type, i.e. instructions that merely copy from memory to a register or vice versa, such as Intel's

  ```
  movl (%eax), %ebx
  ```

  and

  ```
  movl %ecx, (%edx)
  ```

  For example, memory-to-register adds like Intel's

  ```
  addl %eax, (%ebx)
  ```

  are not possible.
- Every instruction has the same length, e.g. 4 bytes. By contrast, Intel instructions range in length from 1 to several bytes.

[6] Published as *Reduced Instruction Set Computer Architectures for VLSI*, M. Katevenis, MIT Press, 1984.

- "Orthogonal architecture"—any operation can use any operand. This is in contrast, for example, to the Intel instruction STOS, which stores multiple copies of a string.[7] Here the operands are the registers EAX, ECX and EDI; one is not allowed to use any other register in place of these.
- The instruction set is limited to only those instructions which are truly justified in terms of performance/space/pipeline tradeoffs. The SPARC chip, for instance, does not even have a multiply instruction. If the compiler sees an expression like, say,

```
I = J * K;
```

  in a source code file, the compiler must generate a set of add and shift instructions to synthesize the multiply operation, compared to the Intel case, in which the compiler would simply generate an **imull** instruction.
- Hardwired implementation, i.e. not microcoded.[8]

All but the last of the traits listed above make pipelining smoother and easier.

3 Effects on Compiler Writers

Computer architects used to partly justify including instructions like **poly** in the instruction set by saying that this made the job of compiler writers easier. However, it was discovered that this was not the case, for several reasons.

- Compiler writers found it difficult to write compilers which would automatically recognize situations in (say) C source code in which instructions like **poly** could be used.
- Compiler writers found that some specialized instructions that were actually motivated by high-level language constructs did not match the latter well. For example, the VAX had a CASE instruction, obviously motivated by the Pascal **case** and C **switch** constructs; yet it turned out to be too restrictive to use well for compiling those constructs.
- CISC generally led to very nonorthogonal architectures, which made the job of compiler writers quite difficult. For instance, in Intel, the requirement that the EAX, ECX and EDI registers be used in STOS, plus the requirement that some of these registers be used in certain other instructions, implies that compiler writers "don't have any spare registers to work with," and thus they must have the compiler generate code to save and restore "popular" registers such as EAX; this is a headache for the compiler writer.

Most RISC machines do ask that the compiler writers do additional work of other kinds, though, in that they have to try to fill **delay slots**; more on this below.

[7] In this context it must be used with the REP prefix.

[8] Again, we will not discuss microcode here.

4 Dealing With the Branch Problem

As you know, a major problem with pipelined CPUs is that the pipeline must be drained when a branch is taken. If for example there are n stages to the pipe, and the instruction is a conditional branch (so that we won't know until the last stage whether the branch will be taken), we will have n-1 empty clock cycles in which no instruction is executed.

4.1 Branch Delay Slots

Many RISC processors use something called a **delayed branch** to deal with the branch problem. The term means that the execution of the **branch target**, i.e. the place to which we jump on the branch , is delayed by one cycle. This is accomplished in an odd but actually quite clever way: Regardless of whether a branch is taken or not, we still execute the instruction following the branch.

For example, suppose we have a 2-stage pipe, with Stage 0 being the instruction fetch and Stage 1 being decode and execute; while one instruction is executing, the next one is being fetched. Note that this implicitly means that we are assuming that it takes one pipe-stage time to read memory.[9] Now consider the following generic (not RISC I/II) code:

```
ADD R5,R3
ADD R2,R4
JNEG T
n.s.i.
```

Here n.s.i. stands for "next sequential instruction," i.e. the instruction following JNEG in memory. Using the normal scheme, if the condition in JNEG is satisfied and we jump to T, we will have one empty clock cycle in which no instruction is executed, because we will have to fetch the instruction at T (which had not been prefetched).

Under the delayed-branch scheme, the n.s.i. will be executed regardless of whether we jump to T! By the time we start execution of the n.s.i., we will know whether the condition in the JNEG was satisfied, and so during this time we either fetch from T (if it was satisfied) or from the location following the n.s.i. (if it was not satisfied). In either case, we will have no empty clock cycles.

A potential problem arises in that we may not want to have the n.s.i. executed. Thus the compiler (or assembly-language programmer) would first tentatively produce code like this

```
ADD R5,R3
ADD R2,R4
JNEG T
NOP
```

so that the delayed-branch mechanism will not harm correct execution of the program. (Recall that most machines have an NOP instruction, pronounced "no-opp," which does literally that, i.e. no operation at all.)

This code would work properly, but it would not exploit the parallelism idea of delayed branches. Instead, what we would like to do is change the order of the instructions:

[9] Or possibly to read from a cache.

```
ADD R2,R4
JNEG T
ADD R5,R3
```

Here the ADD to R5 has been moved to follow the JNEG, and this ADD will be done right after the JNEG. The change in order won't affect the correct execution of the code,[10], but the key point is that in the case the branch is taken, we save one cycle! The time that would have been wasted fetching the instruction at T is now usefully employed by execution of the ADD at R5 at the same time. And of course if the branch is not taken, no harm is done.

The instruction position following a branch is called a **delay slot**. Machines such as MIPS and SPARC have one delay slot, but some other machines have more than one. Clearly, the more delay slots there are, the harder it is to "fill" them, i.e. to find instructions to move to put in the slots, and thus avoid simply putting wasteful NOPs in them.

How hard is it to fill even one delay slot? The Katevenis dissertation on RISC says:

> Measurements have shown [citation given] that the [compiler] optimizer is able to remove about 90% of the no-ops following unconditional transfers and 40% to 60% of those following conditional branches. The unconditional and conditional transfer-instructions each represent approximately 10% of all executed instructions (20% total). Thus, while a conventional pipeline would lose ≈ 20% of the cycles, optimized RISC code only loses about 6% of them.

5 SPARC Example: Delayed Branch

```
 1 Script started on Sun Mar 10 16:19:06 1996
 2 taco 1: m a.c
 3
 4 int a,b;
 5
 6 main(argc,argv)
 7    int argc; char **argv;
 8
 9 {  int x,y;
10
11
12    a = atoi(argv[1]);
13    b = atoi(argv[2]);
14    if (a < b) x = a;
15    else y = a+b;
16 }
17
18 taco 2:
19 taco 2:
```

[10] If we had moved the other ADD, there would have been an effect.

```
20 taco 2:
21 taco 2: cc -S a.c
22 taco 3: m a.s
23 LL0:
24         .seg    "data"
25         .common _a,0x4,"data"
26         .common _b,0x4,"data"
27         .seg    "text"
28         .proc 04
29         .global _main
30 _main:
31         !#PROLOGUE# 0
32         sethi   %hi(LF14),%g1
33         add     %g1,%lo(LF14),%g1
34         save    %sp,%g1,%sp
35         !#PROLOGUE# 1
36         st      %i0,[%fp+0x44]
37         st      %i1,[%fp+0x48]
38         ld      [%fp+0x48],%o0
39         ld      [%o0+0x4],%o0
40         call    _atoi,1
41         nop
42         sethi   %hi(_a),%o1
43         st      %o0,[%o1+%lo(_a)]
44         ld      [%fp+0x48],%o2
45         ld      [%o2+0x8],%o0
46         call    _atoi,1
47         nop
48         sethi   %hi(_b),%o3
49         st      %o0,[%o3+%lo(_b)]
50         sethi   %hi(_a),%o4
51         ld      [%o4+%lo(_a)],%o4
52         sethi   %hi(_b),%o5
53         ld      [%o5+%lo(_b)],%o5
54         cmp     %o4,%o5
55         bge     L17
56         nop
57         sethi   %hi(_a),%o7
58         ld      [%o7+%lo(_a)],%o7
59         st      %o7,[%fp+-0x4]
60         b       L18
61         nop
62 L17:
63         sethi   %hi(_a),%l0
64         ld      [%l0+%lo(_a)],%l0
```

```
 65         sethi   %hi(_b),%l1
 66         ld      [%l1+%lo(_b)],%l1
 67         add     %l0,%l1,%l2
 68         st      %l2,[%fp+-0x8]
 69 L18:
 70 LE14:
 71         ret
 72         restore
 73        LF14 = -104
 74         LP14 = 96
 75         LST14 = 96
 76         LT14 = 96
 77         .seg    "data"
 78 taco 4:
 79
 80
 81 taco 4:
 82 taco 4:
 83 taco 4: cc -S -O1 a.c
 84 taco 5: m a.s
 85         .seg    "text"                  ! [internal]
 86         .proc   4
 87         .global _main
 88 _main:
 89 !#PROLOGUE# 0
 90 !#PROLOGUE# 1
 91         save    %sp,-104,%sp
 92         st      %i0,[%fp+68]
 93         st      %i1,[%fp+72]
 94         call    _atoi,1
 95         ld      [%i1+4],%o0
 96         sethi   %hi(_a),%l0             ! [internal]
 97         st      %o0,[%l0+%lo(_a)]
 98         ld      [%fp+72],%o0
 99         call    _atoi,1
100         ld      [%o0+8],%o0
101         sethi   %hi(_b),%o1             ! [internal]
102         st      %o0,[%o1+%lo(_b)]
103         ld      [%l0+%lo(_a)],%l0
104         ld      [%o1+%lo(_b)],%o1
105         cmp     %l0,%o1
106         bge,a   LY1
107         sethi   %hi(_a),%l0
108         sethi   %hi(_a),%o7
109         ld      [%o7+%lo(_a)],%o7
```

```
110         b       LE14
111         st      %o7,[%fp-4]
112 LY1:                                    ! [internal]
113         ld      [%l0+%lo(_a)],%l0
114         sethi   %hi(_b),%l1
115         ld      [%l1+%lo(_b)],%l1
116         add     %l0,%l1,%l0
117         st      %l0,[%fp-8]
118 LE14:
119         ret
120         restore
121         .seg    "data"                  ! [internal]
122         .common _a,4,"data"
123         .common _b,4,"data"
124 taco 6:
125 taco 6:
126 taco 6: e
127 exit
128
129 script done on Sun Mar 10 16:20:11 1996
```

In lines 3-17, we have a simple C source file a.c, which, in line 21 we compile, using the -S option. This option produces a file a.s, which shows us the assembly language produced in the compilation. We look at that file starting at line 22 (in my .cshrc file, I have aliased ‘m’ to ‘more’).

There are a number of assembler directives which have been inserted by the compiler, such as those seen in lines 24-29, but they do not concern us here.

Recall that the SPARC was modeled on RISC I/II. In particular, it uses register windows. Remember that registers r31-r26 in RISC I/II were used to send parameters to the child function, i.e. as “output” to the called function. So, in SPARC these are denoted %o0, %o1, and so on. Similarly, r15-r10 in RISC I/II are for receiving parameters from the function which called the current one, so the analogs in SPARC are denoted %i0, %i1, etc. The registers named %gn are analogous to RISC I/II’s global registers r9-r0.

The %fp register is the frame pointer, which is a second copy of the stack pointer. So the expression %fp+0x48 in line 38 refers to the location 0x48 bytes into the stack frame, for instance.

The assembly-language notation lists the destination operand last. For example, in line 39, the destination operand is %o0.

Let’s go right to line 36, which (together with the lines following it) is the result of compiling line 12, the call to atoi(). The call occurs at line 40, but first we have to get the parameter ready. This occurs in lines 36-39:

Keep in mind that main() is *itself* a function, whose parameters are argc and argv. So, argc and argv will be in some %i registers, apparently %i0 and %i1, which are copied to the stack frame in lines 36 and 37.

Recall (line 7) that argv is a pointer-to-a-pointer. After line 38 is executed, %o0 now points to argv[0], %o0+4 points to argv[1], %o0+8 points to argv[2], and so on. This is reflected in lines 39 and 45.

After line 39 sets up the parameter, line 40 then is the actual call to the function. Of particular interest to us

here is that the compiler has placed a **nop** in line 41, because after all, the **call** in line 40 is a form of branch. Since we did not request any optimization in our command line (line 21), the compiler has made no attempt to fill this branch-delay slot.

The function's return value will be in %o0. We now have to store it in the variable 'a', which is done in lines 42 and 43:

The **sethi** ("set high") instruction (line 42) puts a value into the high 22 bits of the destination register. Typically the value consists of the high bits of an address, and **sethi** is executed in preparation for a load or store to that address, in this case the **st** in line 43.

Here are the details of line 43. The high 22 bits of the address of 'a' already have been copied (by line 42) to the high 22 bits of the register %o1. By adding the low 10 bits of the address of 'a', we then get the full address of 'a'. Line 43 therefore stores to 'a'.

Note, by the way, that the reason we must resort to this piecemeal way to set up the address of 'a' is that under the RISC philosophy all instructions ought to be the same length, in this case 32 bits. Since the address of 'a' is already 32 bits, we could not fit both it and an op code into one 32-bit instruction!

Now, in line 83, we have recompiled the program, using first-level optimization (-O1), to force the compiler to attempt to fill the branch-delay slots.

To see an example of how the code has changed, note that the **ld** which used to be just before the call, in line 39, now appears just after the call, in line 95. (It also now has slightly different form, because the compiler has done some other optimizations in addition to trying to fill delay slots.)

Now look at lines 106-107. The **bge** instruction in line 55 has been changed to **bge,a,** which means the following: If the branch in line 55 is taken, do execute line 56; if the branch is not taken, then do not execute line 56. Look at the lines following line 63 and line 113, and you will see why things are set up this way.

6 So, How Well Does RISC Work?

Some statistics from Katevenis:

- Due to the simple, orthogonal instruction set, the load/store architecture, etc., the C compiler for the machine was easy to write, taking only 6 person-months, including the optimization components.
- Though RISC's simpler instruction set means we sometimes need to use more instructions to accomplish a given task, the RISC I/II code was typically no more than 50% larger than that of the VAX.
- The ratio of execution times of hand-coded assembly language to C code was about 90% for RISC I/II, but around 45% for the VAX. In other words, on a VAX you pay a much higher speed penalty for the convenience of coding in a high-level language.
- The **control area** of the RISC II chip consists just of the opcode decoder, and occupied only 0.5% of the chip. By contrast, in the Motorola 68000,[11] the control area, in this case meaning the microcode, occupied 68% of the chip.
- The chip layout design for RISC II was done by students in 12 person-months. For the 68000 the process was done by experienced professionals, and took 70 person-months.

[11] The chip family formerly used in Macintoshes.

- RISC II and a PDP-11/70, having the same clock speed and the same register-to-register addition time, were compared over 11 programs. RISC II was on the average 2.1 times faster.

7 What to Do With All That Extra Space?

Now that RISC has liberated all this real estate, what might we do with it?

- cache
- more registers
- on-chip stack
- another CPU!