

Example of RISC Architecture: MIPS

Norman Matloff
University of California at Davis
©2002-2007, N. Matloff

December 11, 2006

Contents

1	Introduction	2
2	A Definition of RISC	3
3	Beneficial Effects for Compiler Writers	4
4	Introduction to the MIPS Architecture	4
4.1	Register Set	5
4.2	Example Code	5
4.3	MIPS Assembler Pseudoinstructions	7
4.4	Programs Tend to Be Longer on RISC Machines	8
4.5	Instruction Formats	9
4.6	Arithmetic and Logic Instruction Set	10
4.7	Conditional Branches in MIPS	10
5	Some MIPS Op Codes	11
6	Dealing with Branch Delays	11
6.1	Branch Prediction	11
6.2	Delayed Branch	11

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 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.³ 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,⁴ 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 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.

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

²It 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.

³Actually, 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.

⁴You 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.

2 A DEFINITION OF RISC

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.⁵

At about the same time, Professor John Hennessy at Stanford University (who was later to become president of the university) was conducting his own RISC project, leading to his founding of the MIPS Corporation, which produced his RISC chip.

Silicon Graphics, Inc. eventually bought MIPS Inc. and SGI uses MIPS processors in its servers and so on, and though SGI is no longer one of the big players in those segments of the industry, MIPS processors are popular in the world of embedded applications.

Today virtually every computer manufacturer has placed a major emphasis on RISC. Even Intel, the ultimate CISC company, has turned to RISC for its most powerful chips. Sun Microsystems has the SPARC chip, IBM and Motorola have the Power PC chip series used in Macs,⁶ and so on.

Of course, 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)
```

Instructions 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.
- **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 only available operands are the registers EAX, ECX and EDI; one is not allowed to use any other register in place of these.

⁵Published as *Reduced Instruction Set Computer Architectures for VLSI*, M. Katevenis, MIT Press, 1984.

⁶Though Apple has announced plans to switch to Intel starting 2008.

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

4 INTRODUCTION TO THE MIPS ARCHITECTURE

- 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 Beneficial Effects for 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.

All of this inhibits development of compilers which produce very fast code.

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.

4 Introduction to the MIPS Architecture

MIPS is one of the most popular RISC chips, used for example in SGI workstations and in a number of embedded applications.⁹

⁸Again, we will not discuss microcode here.

⁹MIPS is also the subject of a simulator, SPIM, which is used in many university computer architecture courses. See <http://www.cs.wisc.edu/~larus/spim.html>.

Every MIPS instruction is 32 bits, i.e. one word, in length, and every instruction which does not access memory executes in one cycle. Those that do access memory are assumed to “usually” take two cycles, meaning that this will be the case if the desired memory access results in a cache hit; otherwise, the CPU enters a **stall** mode until the memory access is satisfied.

4.1 Register Set

There are 32 general registers, named \$0, \$1, ..., \$31. The assembler also has an alternate set of names for the registers, which show their suggested uses; here is a partial list:

\$0	zero	contains the hard-wired constant 0
\$1-\$2	v0-v1	for expression evaluation and function return values
\$4-\$7	a0-a3	for passing arguments
\$8-\$15	t0-t7	for temporary results
...		
\$28	gp	pointer to global area
\$29	sp	stack pointer
\$30	fp	frame pointer (not used)
\$31	ra	return address

The usages listed above, e.g. \$29 as the stack pointer, are merely suggested; in keeping with the RISC philosophy of orthogonality, the registers are treated uniformly by the architecture.

4.2 Example Code

To introduce the MIPS architecture, here is a C program which calls a MIPS assembly language subroutine, **addone()**:

TryAddOne.c:

```
int x;

main()
{
    x = 7;
    addone(&x);
    printf("%d\n",x); // should print out 8
    exit(1);
}
```

AddOneMIPS.s:

```
.text
    .globl addone
addone:
    lw     $2, 0($4)
    addu  $2, $2, 1
    sw     $2, 0($4)
    j     $31
```

To begin, let’s look at the compiled code we obtain by applying **gcc -S** to TryAddOne.c. The source lines

```
x = 7;
addone (&x);
```

translate to:

```
# x = 7;
li    $2,7
sw    $2,x

# addone (&x);
la    $4,x
la    $25,addone
jal   $31,$25
```

The **li** (Load Immediate) instruction puts 7 into the register \$2.¹⁰

The **sw** (Store Word) instruction then copies what is in \$2 to the memory location *x*.

Now, what about the call? Many RISC architectures are less stack-oriented than are the classical CISC ones. This is because stacks are in memory, and memory access is slow.¹¹ MIPS is an example of this philosophy. Both the argument to **addone()**, as well as the return address, will be put in registers, rather than on the stack.

The **la** (Load Address) instruction here puts the argument, the address of *x*, into register \$4.¹² We then use this same instruction to put the jump target, i.e. the address of **addone()**, into \$25. We then call **addone()** with the instruction

```
jal   $31,$25
```

which says to “jump and link” (i.e. do a subroutine call) to the subroutine pointed to by \$25, saving the return address (the address of the instruction following the **jal**) in \$31.

The return from the subroutine thus makes use of \$31:

```
j     $31
```

This is an unconditional jump, to the location pointed to by \$31.

Note that operands have different meanings depending on the operation. In **sw \$2, x**, the *x* meant the memory location named *x*, whereas in **la \$4, x**, *x* meant the address of *x*.

Recall that **main()** had placed the argument in \$4. The subroutine then uses it:

```
lw    $2,0($4)
addu  $2,$2,1
sw    $2,0($4)
```

¹⁰Actually, **li** is not a real MIPS instruction, but is instead a macro, from which the assembler will generate an **ori** instruction. More on this later.

¹¹Some architectures, such as SPARC, try to solve this problem by actually having space on the CPU chip for the top few elements of the stack.

¹²Recall that registers \$4-\$7 are used for passing arguments. If there are more arguments, we can store them in memory, and one of the arguments can be a pointer to them.

The **lw** (Load Word) instruction is the opposite of **sw**, fetching a word from memory rather than storing it. Both the **lw** and **sw** here use the more general format, which is based addressing. For example,

```
lw      $2, 0($4)
```

adds \$4 to 0, and then treats that as an address, fetching the contents of that address and putting it in register \$2.

The **addu** (Add Unsigned) instruction has three operands; it adds the second and third, and places the sum in the first.

4.3 MIPS Assembler Pseudoinstructions

Recall that most assembly languages allow **macros**. MIPS assemblers go a step further, offering the programmer use of **pseudoinstructions**; these appear to be machine instructions, but are actually like macros, which the assembler translates to one or more real MIPS instructions.

Several “instructions” which we saw in the example above are actually pseudoinstructions. The first of those was **li**, allegedly the Load Immediate instruction. There actually is no such instruction, and our line containing **li** was actually treated by the assembler as

```
ori $2, $0, 7
```

where **ori** is the MIPS’ OR Immediate instruction. Remember, the register \$0 contained the hardwired value 0, so the instruction above applies the logical OR operation to 0 and 7, yielding 7, and thus placing 7 into the register \$2, just as desired. So, why couldn’t the designers of the MIPS architecture include **li** Load Immediate instruction?

To see the answer, suppose we wish to load a large number like 0xabcd1234, which is 32 bits in length. Remember, all MIPS instructions are 32 bits long, so this operand would occupy the entire instruction, with no room for an op code!

Indeed, if our assembly language source code contains a line

```
li $2, 0xabcd1234
```

the assembler will actually treat it as if it were two instructions:

```
lui $1, 0xabcd
ori $2, $1, 0x1234
```

The MIPS instruction **lui** places the given immediate constant, 0xabcd in this case, and placing it in upper 16 bits of the destination, the register \$1 here, and places 0s in the lower 16 bits. The MIPS instruction **ori** then puts 0x1234 in the lower 16 bits of a copy of \$1, then placing the result in \$2. This gets 0xabcd1234 into \$2, as desired.

So, the use of pseudoinstructions here frees the programmer from having to think about the size of the operand she is using.

Similarly, in the line

4 INTRODUCTION TO THE MIPS ARCHITECTURE Programs Tend to Be Longer on RISC Machines

```
la $4, x
```

from our example code above, **la** is just a pseudoinstruction, not a real MIPS machine instruction. One of the issues here is like the one with **li** above; the address of `x` is a 32-bit quantity, and thus could not be fit into one instruction. The other issue is that we need a way of letting the assembler know that “`x`” here means the address of `x`, and our use of the pseudoinstruction **la** tells the assembler that.

Finally, there is even some chicanery in the line

```
sw $2, x
```

Actually, **sw** IS a real MIPS instruction. But the real instruction has three operands, as in our other line,

```
sw $2, 0($4)
```

(the three operands are `$2`, `0` and `$4`), whereas

```
sw $2, x
```

has only two operands. The fact that there are only two operands here is a signal to the assembler that this too is a pseudoinstruction, in spite of the fact that there is a real MIPS instruction **sw**. So, the assembler will convert this to an **lui** and **ori** and then a (real) **sw**.

4.4 Programs Tend to Be Longer on RISC Machines

Note in our example above that it took two instructions

```
li    $2, 7
sw    $2, x
```

to do what, for example, on Intel would take only one:

```
movl $7, x
```

In fact, we later found that those two instructions were actually pseudoinstructions, so here MIPS is taking four instructions to do what Intel does in one.

This is typical of RISC machines, and means that RISC programs tend to be longer than CISC ones. For example, the compiled version of `TryAddOne.c` above¹³ is 1276 bytes long on an SGI machine but only 1048 on an Intel platform.

¹³This is `TryAddOne.o`, generated from applying `gcc -c` to `TryAddOne.c`.

4.5 Instruction Formats

One very “RISC-y” feature of MIPS is that it has only three instruction formats.¹⁴ To present them, let us number the most- and least-significant bits in a word as 31 and 0, respectively. Also, we will use the following abbreviations:

op	op code
---	-----
rs	register operand
rt	register operand
rd	register operand
imm	immediate constant
shamt	shift amount
funct	special function (like an additional op code)
dist	distance factor to branch (i.e. jump) target

the formats are as follows.

- I (“immediate”) format:
 - op: bits 31-26
 - rs (source): bits 25-21
 - rt (destination): bits 20-16
 - imm: bits 15-0
- J (“jump”) format:
 - op: bits 31-26
 - dist: bits 25-0
- R (“register”) format:
 - op: bits 31-26
 - rs (source): bits 25-21
 - rt (source): bits 20-16
 - rd (destination): bits 15-11
 - shamt: bits 10-6
 - funct: bits 5-0

The **shamt** field is used if the instruction is to perform a right- or left-shift; the shift will be **shamt** bits in the specified direction.

The **dist** field in J instructions is somewhat unusual, but actually clever: It is equal to 1/4 of the distance to the branch target.¹⁵ Since all instructions are 4 bytes long, then every instruction address is a multiple of 4,

¹⁴By contrast, the Intel chip, a CISC, has a large number of instruction formats, and even the RISC SPARC chip has six.

¹⁵Recall that *branch* is a synonym for *jump*.

and thus the distance from any branch instruction to its target is a multiple of 4, i.e. the distance has two 0 bits at the right end. That last point means that there is no point in storing the last two bits of the distance; they are simply tacked on by the CPU when the instruction is executed. This effectively multiplies the jump range by 4.

The J format is used only for unconditional branches, including **call**. Conditional branches use the I or R format.

Note that the **imm** field can have two interpretations, depending on **op**. The **addi** instruction, for example, treats the immediate constant as a signed number, while **addiu** treats it as unsigned. Since MIPS has word size 32 bits, its ALU deals with 32-bit quantities, not the 16-bit quantity in an **imm** field. Thus the latter must be extended to 32 bits before entering the ALU, by copying the leftmost bit to the leading 16 bits. Thus for instance in the instructions

```
addi $6, $7, 0xb123
addiu $9, $10, 0xb123
```

in the first case the 0xb123 becomes 0xffffb123 while in the second case it becomes 0x0000b123.

4.6 Arithmetic and Logic Instruction Set

Here is a partial listing of the MIPS instruction set for arithmetic and logic operations:

```
add rd, rs, rt           # rd <-- rs + rt
addu rd, rs, rt         # rd <-- rs + rt (without overflow)
addi rd, rs, imm        # rd <-- rs + imm

and rd, rs, rt          # rd <-- rs AND rt
andi rd, rs, imm       # rd <-- rs AND imm

not rd, rs              # rs <-- NOT rd

or rd, rs, rt           # rd <-- rs OR rt
ori rd, rs, imm        # rd <-- rs OR imm

sll rd, rt, shamt      # rd <-- rt << shamt
srl rd, rt, shamt      # rd <-- rt >> shamt

sub rd, rs, rt          # rd <-- rs - rt
subu rd, rs, rt        # rd <-- rs - rt (without overflow)
```

4.7 Conditional Branches in MIPS

On a MIPS machine, applying **gcc -S** to

```
if (i == 20) j = 2;
```

produces

```
lw     $2,i
li     $3,20
bne    $2,$3,.L4
li     $2,2
sw     $2,j
```

6 DEALING WITH BRANCH DELAYS

where `.L4` is a label of the branch target. The `bne` instruction compares the two registers and then branches to the target if they are not equal. (There is nothing like Intel's Z and S Flags in MIPS.)

5 Some MIPS Op Codes

```
ORI    001101
LUI    001111
SW     101011
JAL    000011
J      000010
BNE    000101
ADDI   001000
ADDU   001001
```

6 Dealing with Branch Delays

RISC architectures are especially good at having smooth pipeline operation, due to the uniformity of instruction execution time: Every instruction executes in one clock cycle, except for branches and load/store instructions, which access memory and thus take longer. Even to go to the cache takes up an extra clock cycle, and going to full memory takes even longer. Thus the pipeline is delayed.

For example, consider this MIPS code:

```
subiu $2, $2, $1
bne   $4, $3, yyy
addiu $5, $5, 12
```

While the fetch/decode/execute cycle for the `bne` is taking place, the CPU is prefetching the `addiu`. But if the branch is taken, the instruction at `yyy` will be the one we need, and it will not have been prefetched.

6.1 Branch Prediction

One possible way to deal with the branch-delay problem is to design the hardware to try to predict whether a conditional branch will be taken or not. The hardware will prefetch from the instruction which it predicts will be executed next. If it is correct, there is no delay; if the prediction is false, we must fetch the other instruction, incurring the delay.

Even non-RISC architectures do this. Intel CPUs, for example do the following when a given conditional branch instruction is executed for the first time: If it is a branch backward, the prediction is that the branch will be taken; this design is motivated by the assumption that backward branches will typically be the last instruction within a loop, and usually such branches will be taken. If the branch is forward, the prediction is that it will not be taken. In subsequent times this particular branch instruction is executed, a complicated prediction algorithm is applied, based on previous history.

6.2 Delayed Branch

MIPS—as well as most RISC chips—solves this problem by using something called **delayed branch**. The CPU actually executes the instruction which sequentially follows the branch instruction in memory, in this


case the **addiu**, before executing the instruction at the branch target. That way useful work is being done during the delay involving fetching of the branch target.

But wait! This can't work as it stands. After all, if the branch is taken, we do not want to execute that **addiu**! So, the assembler (or compiler, in the case of C/C++) must put in a "do nothing" instruction: **nop**, pronounced "no-op" for "no operation."¹⁶

```
subiu $2, $2, $1
bne   $4, $3, yyy
nop
addiu $5, $5, 12
```

But now we are back to the original problem — no useful work is being done while the instruction at yyy is being fetched. So, next an optimizer within the assembler (or compiler) will try to actually rearrange the instructions. It will look for some instruction elsewhere in the code which can be moved into the slot currently occupied by the **nop**. In this case, the **subiu** can be safely moved:

```
bne $4, $3, yyy
subiu $2, $2, $1
addiu $5, $5, 12
```

The reasoning is this: The **bne** is not affected by the outcome of the **subiu** which precedes it (this would not be true if the **bne** compared \$4 with \$2, for example), so it doesn't matter if we move the **subiu**. Remember, the latter will still be executed, because the CPU is designed to execute the instruction which immediately follows the branch in memory. 

The setup for load/store instructions is similar. The MIPS CPU will execute the instruction which immediately follows a load/store instruction in memory while performing the load/store. This is done because the latter requires a time-consuming memory access.

But as with branches, we need to watch out for dependencies. For example, consider

```
lw $2, 0($4)
addiu $2, $2, 1
```

The CPU would execute the add simultaneously with fetching register \$2, with potentially disastrous consequences. Thus the assembler will at first change this to

```
lw $2, 0($4)
nop
addiu $2, $2, 1
```

and then the optimizer will try to find some instruction elsewhere to move to the position now occupied by the **nop**.

The instruction position following a branch or load/store 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:

¹⁶Almost every CPU architecture has this kind of instruction. On MIPS, its op code is all 0s.

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 $\approx 20\%$ of the cycles, optimized RISC code only loses about 6% of them.