Operations, Operands, and Instructions

Tom Kelliher, CS 220

Sept. 14, 2009

1 Administrivia

Announcements

Assignment

Read 2.6–2.7.

From Last Time

Macro-architectural trends; IC fab.

Outline

1. Introduction.
2. Simple arithmetic operations
3. Operands: registers, memory.
4. R-Format and I-Format instruction formats.

Coming Up

Logical and conditional instructions in MIPS.

2 Introduction

1. Stored program concept. Is it data, address of data, instruction, or what?

2. Instruction set.

3. Operands.

4. Collected design principles:

   (a) *Simplicity favors regularity.* A simple architecture will result in regular, clean implementations.

   Example: arithmetic operations that have two source operands.

   (b) *Smaller is faster.* Storage capacity vs. speed tradeoffs.

   Examples: Register file size. Word size. Instruction set cardinality.

   (c) *Make the common case fast.*

   Example: The most common immediate values are small. Store them in the instruction.

   (d) *Good design demands good compromises.* We can't optimize everything.

   Example: Having only a few instruction formats, all of which are regular.

3 Arithmetic Instructions

Instruction semantics:

```
add a, b, c        # This, BTW, is a comment.
sub a, a, b
```

De-compile into a single HLL statement:

```
add a, b, c
add a, a, d
add a, a, e
```

Compile each of the following:

```
a = b + c;
d = a - e;
f = (g + h) - (i + j);
```

4 Instruction Operands

Where are the operands?

4.1 Registers

`$0` to `$31` or `$s0`, `$t0`, etc.

Example:

```
add $1, $2, $3
```

Properties of registers:

1. Number of registers. 32 for MIPS, including the hardwired register. Two ways of naming: numbers, convention "nicknames". Why not more? Size of register file, size of operand fields within instructions.

2. Number of bits/register. 32. Word size.

   32 bits is 4 bytes.

   Implications: size of address space, datapath width.

3. General purpose vs. special purpose.

   MIPS, M68000, x86.

4.2 Memory Operands

1. HLL have complex data structures such as arrays and structs. How are they handled?

2. Data transfer instructions: load, store. operands: memory address, register.

3. Actual MIPS instructions: `lw`, `sw`.

   Base and offset addressing: `lw $s0, 8($s1)`

4. MIPS memory is byte addressable, so word addresses differ by 4:

Word Address	Byte Address			
0	0	1	2	3
4	4	5	6	7
8	8	9	10	11
12	12	13	14	15
	msB			lsB

   Alignment restriction — Word addresses start on word boundaries. A good compromise.

   This isn't the only way to number the bytes within word. "Big endian."

Base, offset addressing. Usefulness when accessing a member of an object.

Compile each of the following:

```
A[12] = h + A[8];

A[j] = h + A[i];
```

where `h` is in `$1`, `i` is in `$2`, `j` is in `$3`, and the base address of `A`, an `int` array of 100 words, is in `$4`.

Variables in registers are simpler to use and faster than variables in memory. Compilers must be clever in optimizing register use.

Example:

```
lw $1, 4($0)
lw $2, 8($0)
add $1, $1, $2
sw $1 0($0)
```

Explain what this is doing, in plain English.

Another example:

```
lw $t0, 0($s0)
addi $s0, $s0, 4
lw $t1, 0($s0)
add $t0, $t0, $t1
addi $s0, $s0, 4
sw $t0, 0($s0)
```

Again, explain.

5 Representing Numbers

Basics:

1. Conventional number systems are weighted and positional.

2. A base x systems uses x numerals (symbols).

3. Representing negative values is tricky — we cheat in decimal.

4. The hexadecimal system and its relationship to binary.

5. Base conversions.

6. msb and lsb.

5.1 Unsigned Integers

1. Consider a four bit unsigned binary integer: $b_3b_2b_1b_0$. What does b_i contribute to the value of the number?

2. What range of values can be represented using just four bits? Eight bits? n bits?

5.2 Signed Integers

How might we designate that a binary number is negative?

Two problems with this approach:

1. Two representations for zero.

2. Adder design becomes more difficult.

This is sign-magnitude representation.

5.3 Two's Complement Representation

Again, msb is sign bit.

1. Positive values have same representation as for unsigned case.

2. To compute the inverse, complement all the bits and add 1.

3. One 0.

4. Asymmetric range: one more negative value.

5.4 Two's Complement: Negation, Sign Extension

1. Negation:

   (a) Two's complement is called by that name because

   $$2^n - x = -x$$

   Which is okay because $2^n = 0$. Wait — Why?

   Obviously,

   $$(2^n - 1) - x = \overline{x}$$

   So:

   $$\begin{aligned} -x &= 2^n - x \\ &= (2^n - 1) - x + 1 \\ &= \overline{x} + 1 \end{aligned}$$

   Hence our algorithm for negating a two's complement number.

2. Sign Extension:

   (a) How do you place a 16-bit *signed* immediate into a register?

   (b) How do you load a *signed* byte into register?

   (c) The sign extension algorithm.

   (d) Load byte *unsigned*.

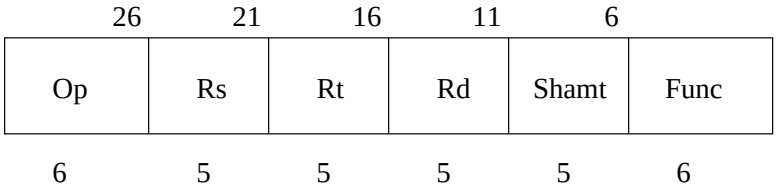
   (e) How can this simple procedure preserve the value of a negative number? Proof?

3. Signed & unsigned comparison. Or, sometimes 1111 is less than 0000 and sometimes it's not.

6 Representing Instructions

6.1 MIPS R-Format

Example instruction: `add $s2, $s0, $s1`



Fields:

1. Op: Opcode.
2. Rs: First *source* operand.
3. Rt: Second source operand.
4. Rd: *Destination* operand.
5. Shamt: Shift amount — ignore for now.
6. Func: Function. Further specification of the opcode.

In assembly: `Op/Func Rd, Rs, Rt`

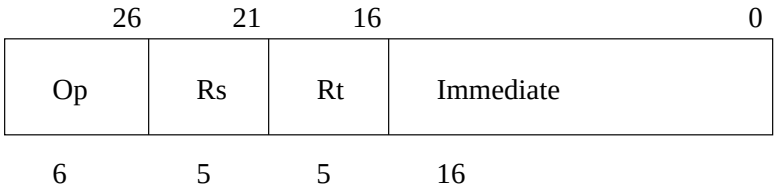
Notes:

1. Become familiar with field positions and sizes for all three formats.
2. Example encodings:

Assembly	**Op**	**Rs**	**Rt**	**Rd**	**Shamt**	**Func**
`add $1, $2, $3`	0	2	3	1	D.C.	32
`sub $4, $5, $6`	0	5	6	4	D.C.	34

6.2 MIPS I-Format

Example instruction: `lw $s0 8($s1)`



Fields:

1. Op: Opcode.
2. Rs: Source register.
3. Rt: *Destination* register.
4. Address: 16-bit signed immediate value.

   Offset range?

Example encoding of:

```
A[20] += 72;
```

Assembly	**Op**	**Rs**	**Rt**	**Immediate**
`lw $t1, 80($s0)`	35	16	9	80
`addi $t2, $t1, 72`	8	9	10	72
`sw $t2, 80($s0)`	43	16	10	80