Instruction Set Design II

Tom Kelliher, CS 240

Feb. 1, 2002

1 Administrivia

Announcements

Assignment

Read 3.5, A.1–5, A.9.

From Last Time

Instruction Set Design I.

Outline

1. Instruction operands: registers.
2. Accessing memory.
3. Instruction formats.

Coming Up

Conditional instructions, SPIM.

2 Instruction Set Design

2.1 Instruction Operands

(Continued.)

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.

   Other register files: x86, SPARC and the register window (Berkeley RISC, about 128 registers, spilling).

   Register renaming: ISA registers vs. physical registers.

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

   Implications: size of address space, datapath width.

3. General purpose vs. special purpose.

   MIPS, M68000, x86.

2.2 Using MIPS Registers

See pg. A-23 for the full register naming convention. Note the limited number of `s` and `t` registers.

Our simple convention:

1. Use `$s0`, `$s1`, etc. for C variables.

2. Use `$t0`, `$t1`, etc. for temps.

Recall:

```
f = (g + h) - (i + j);
```

Assume `f` through `j` are in `$s0` through `$s4`, respectively. Compile the statement.

2.3 Memory Addressing

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

5. Endianess:

   (a) Big endian machines: HP PA-RISC, IBM Power, MIPS, SPARC.

   (b) Little Endian: x86.

   (c) MIPS can go either way!

Compile the following:

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

where `g` is in `$s1`, `h` is in `$s2`, and the base address of `A`, an array of 100 words, is in `$s3`.

Base, offset addressing.

Compile each of the following:

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

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

Notes:

1. Base, offset addressing, using constant offsets, is similarly useful for accessing members of structures.
2. Variables in registers are simpler to use and faster than variables in memory. Compilers must be clever in optimizing register use. *Spilling* registers.

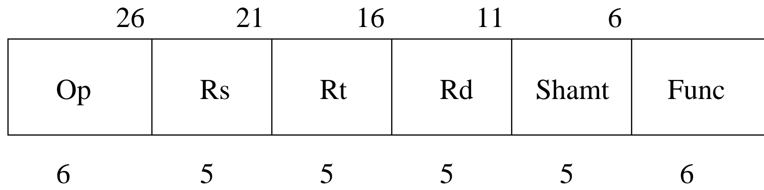
3 Instruction Formats

How do we encode instructions for storage in memory? Issues:

1. Compactness — smaller programs use less memory bus bandwidth and less memory space.
2. Decoding ease — simplify the instruction decoder.
3. Length — fixed or variable.
4. Size — one or multiple words.

3.1 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. **Memorize** field positions and sizes for all three formats. Necessary later.
2. Example encodings:

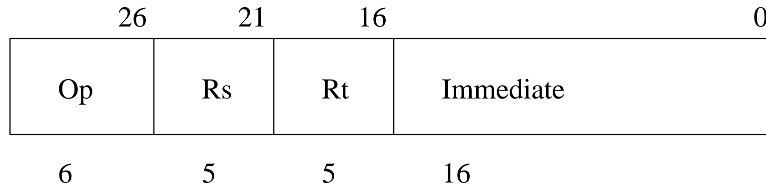
Assembly	**Op**	**Rs**	**Rt**	**Rd**	**Shamt**	**Func**
`add $1, $2, $3`	0	2	3	1	0	32
`sub $4, $5, $6`	0	5	6	4	0	34

3.2 MIPS I-Format

1. How do we fit Memory instructions into r-format? We can't!
2. Design principle 3: Good design demands good compromises.

3. Introduce another format — added complexity tradeoff.

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?

In assembly: `Op/Func Rt, address(Rs)`

Notes:

1. This format also used for immediate operands: `addi $1, $2, 123`.
2. Example encodings:

Assembly	Op	Rs	Rt	Address
`lw $1, 1000($2)`	35	2	1	1000
`sw $3, -12($4)`	43	4	3	-12