Instruction Set Design I

Tom Kelliher, CS 240

Jan. 30, 2002

1 Administrivia

Announcements

Distribute shell account info for phoenix.

Distribute homework I.

Assignment

Read 3.5.

From Last Time

Calculating performance.

Outline

1. Design principles.
2. Classes of instructions.
3. Operands.

4. MIPS registers.

5. Memory addressing.

Coming Up

Conditional instructions in MIPS.

2 Instruction Set Design

Things to consider:

1. General design principles. E. g., simplicity.

2. Operations.

3. Operands: number, memory addressing modes.

2.1 Design Principles

1. Simplicity favors regularity. Example: the layout of a WalMart store.

2. Smaller is faster. Example: L1 vs. L2 caches.

We'll see the implications in what comes next.

2.2 Classes of Instructions

Consider your favorite HLL. What classes of instructions (operations) are required?

Now, consider a multi-user OS such as Unix? What *additional* classes of instructions are required?

Any general purpose architecture must support these classes.

2.3 Arithmetic Instructions

Consider the two simplest:

```
add
sub
```

Consider an HLL statement:

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

1. Should an instruction set directly support complex arithmetic statements?
2. How about directly supporting a variable number of operands?
3. How many operands should arithmetic instruction take? 3? 2? 1? 0? Tradeoffs.

2.3.1 Instruction Semantics

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

De-compile each of the following:

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

De-compile further into a single HLL statement.

Compile each of the following:

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

2.4 Instruction Operands

Consider operands within an HLL:

```
#include <stdio.h>

int main()
{
   int foo = 1234;

   printf("%d,   %p\n", foo, &foo);

   return 0;
}
```

A variable is an abstraction which the compiler/OS binds to a memory address.

RISC instruction sets don't ordinarily support memory operands. Why not?

Where are the operands? Registers.

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.

2.5 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.

2.6 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.