Instruction Set Design I

Tom Kelliher, CS 240

Feb. 4, 2004

1 Administrivia

Announcements

Distribute shell account info for phoenix.

AIM screen name.

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.