INSTRUCTION SET ARCHITECTURE

Mahdi Nazm Bojnordi

Assistant Professor
School of Computing
University of Utah
Overview

- Homework 2 due on Jan 24th (midnight)

- One more TA added
  - Please check the class webpage for office hours

- This lecture
  - Instruction set architecture (ISA)
Recall: Example MIPS Instruction

- Translate this one

\[ f = (g + h) - (i + j); \]

- Assembly

<table>
<thead>
<tr>
<th>MIPS Instruction</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>add f, g, h</td>
<td>add t0, g, h</td>
</tr>
<tr>
<td>sub f, f, i</td>
<td>add t1, i, j</td>
</tr>
<tr>
<td>sub f, f, j</td>
<td>sub f, t0, t1</td>
</tr>
</tbody>
</table>

- In summary

- operations are not necessarily associative and commutative
- More instructions than C statements
- Usually fixed number of operands per instruction
In a high level language, each variable is a location in memory.

You may define a large number of operands (variables) in a high-level program.

The number of operands in assembly is fixed (registers).
To simplify hardware, let’s require each instruction (add, sub) only operate on registers.

For example:
- MIPS ISA has 32 registers
- x86 has 8 registers

32-bit registers
- Modern 64-bit architectures

Every 32-bit stores a **word**
A set of registers in the processor core

- An index is used to identify each register

\[
\text{add a, b, c}
\]
\[\text{add } \$3, \$4, \$1\]
\[\$3 \leftarrow \$4 + \$1\]

For more readability

- Registers are partitioned as $s0$-$s7$ (C/Java variables), $t0$-$t9$ (temporary variables)...
Memory Access

- Values must be fetched from memory before (add and sub) instructions can operate on them

- Memory operations
  - Read
    - Returns *data* stored at location *address*
  - Write
    - Stores *data* at location *address*
Memory Access

- Values must be fetched from memory before (add and sub) instructions can operate on them

- Load word
  - `lw $t0, memory-address`

- Store word
  - `sw $t0, memory-address`

- How is memory-address determined?
The compiler organizes data in memory... it knows the location of every variable (saved in a table)... it can fill in the appropriate mem-address for load-store instructions

```c
int a, b, c, d[10]
```
The compiler organizes data in memory... it knows the location of every variable (saved in a table)... it can fill in the appropriate mem-address for load-store instructions.
Memory Address

- Each word is referred to with the address of a single byte

```plaintext
int a, b, c, d[10]
```

 Glory A = 8163 $\rightarrow$ (00000000 00000000 00011111 11100011)$_{bin}$
Memory Address

- Each word is referred to with the address of a single byte

```plaintext
int a, b, c, d[10]
```

Memory

```plaintext
c = 8163  \rightarrow (00000000 00000000 00011111 11100011)_\text{bin}
            \rightarrow (00 00 1F E3)_\text{hex} = 0X00001FE3
```
Memory Address

- Each word is referred to with the address of a single byte

- **Big Endian**
  - MIPS, IBM 360/370,
  - Motorola 68k, Sparc,
  - HP PA, ARMv8

- \[ c = 8163 \rightarrow (00000000 \ 00000000 \ 00011111 \ 11100011)_{\text{bin}} \]
- \[ \rightarrow (00 \ 00 \ 1F \ E3)_{\text{hex}} = 0X00001FE3 \]
Memory Address

- Each word is referred to with the address of a single byte
  - Little Endian
    - Intel x86, DEC VAX
    - DEC Alpha

Memory

\[
c = 8163 \rightarrow (00000000\ 00000000\ 00011111\ 11100011)_{\text{bin}}\ \\
\rightarrow (00\ 00\ 1F\ E3)_{\text{hex}} = 0X00001FE3
\]
Immediate Operand

- An instruction may require a constant as input.

- An immediate instruction uses a constant number as one of the inputs (instead of a register operand).

- Putting a constant in a register requires addition to register $zero (a special register that always has zero in it) -- since every instruction requires at least one operand to be a register.

- For example, putting the constant 1000 into a register:
  - `addi $s0, $zero, 1000`