INSTRUCTION SET ARCHITECTURE

Mahdi Nazm Bojnordi

Assistant Professor

School of Computing

University of Utah
Overview

- This lecture
  - Instruction Operands
  - Registers
  - Basics of Memory
  - Memory Access Instructions
Recall: Example MIPS Instruction

- Translate this one

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

- Assembly

```
add f, g, h  
sub f, f, i  
sub f, f, j  
```

```
add t0, g, h  
at1, i, j     
sub f, t0, t1
``` 

- In summary
  - operations are not necessarily associative and commutative
  - More instructions than C statements
  - Usually fixed number of operands per instruction
Operands

- 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
Register File

- A set of registers in the processor core
  - An index is used to identify each register

  ```
  add a, b, c
  add $3, $4, $1
  $3 ← $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

int a, b, c, d[10]

\[
c = 8163 \rightarrow (00000000 00000000 00011111 11100011)_{\text{bin}}
\]
Memory Address

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

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

- `c = 8163` → `00000000 00000000 00011111 11100011)_bin`
- `c = 8163` → `(00 00 1F E3)_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

Memory

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

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

\[
\begin{align*}
  c &= 8163 \\
  &\Rightarrow (00000000 \ 00000000 \ 00011111 \ 11100011)_{\text{bin}} \\
  &\Rightarrow (00 \ 00 \ 1F \ E3)_{\text{hex}} = 0X00001FE3
\end{align*}
\]