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

```
add  f, g, h  
sub  f, f, i  
sub  f, f, j  
add  t0, g, h  
add  t1, 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
A set of registers in the processor core

- An index is used to identify each register

```plaintext
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

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

\[
\begin{align*}
\text{int } &a, b, c, d[10] \\
\end{align*}
\]

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

- Example:
  - \( c = 8163 \) → \( (00000000 \ 00000000 \ 00011111 \ 11100011)_{\text{bin}} \)
  → \( (00 \ 00 \ 1F \ E3)_{\text{hex}} = 0X00001FE3 \)
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`
The format of a load instruction:

- Destination register
- Source address
- \texttt{lw} $t0, 8(t3)$
- Any register
- A constant added to the register in brackets
The format of a store instruction:

- source register
- source address
- `sw $t0, 8($t3)`
- any register
- a constant added to the register in brackets
```c
int a, b, c, d[10];

addi $t0, $zero, 1000  # assume that data is stored at 
    # base address 1000; placed in $t0; 
    # $zero is a register that always 
    # equals zero
lw  $s1, 0($t0)        # brings value of a into register $s1
lw  $s2, 4($t0)        # brings value of b into register $s2
lw  $s3, 8($t0)        # brings value of c into register $s3
lw  $s4, 12($t0)       # brings value of d[0] into register $s4
lw  $s5, 16($t0)       # brings value of d[1] into register $s5
```
Example

- Convert the following C code to assembly
Example

- Convert the following C code to assembly

Assembly (same assumptions as previous example):

- \(\text{lw} \quad $s0, 0($t0)\)  \#  a is brought into $s0
- \(\text{lw} \quad $s1, 20($t0)\)  \#  d[2] is brought into $s1
- \(\text{add} \quad $t1, $s0, $s1\)  \#  the sum is in $t1
- \(\text{sw} \quad $t1, 24($t0)\)  \#  $t1 is stored into d[3]
Memory Organization

- The space allocated on stack by a procedure is termed the activation record (includes saved values and data local to the procedure) – frame pointer points to the start of the record and stack pointer points to the end – variable addresses are specified relative to $fp$ as $sp$ may change during the execution of the procedure.
- $gp$ points to area in memory that saves global variables.
- Dynamically allocated storage (with malloc()) is placed on the heap.
Recap – Numeric Representations

- **Decimal**  \( 35_{10} = 3 \times 10^1 + 5 \times 10^0 \)
- **Binary**  \( 00100011_2 = 1 \times 2^5 + 1 \times 2^1 + 1 \times 2^0 \)
- **Hexadecimal (compact representation)**
  \( 0x\ 23 \text{ or } 23_{\text{hex}} = 2 \times 16^1 + 3 \times 16^0 \)

0-15 (decimal) \( \rightarrow \) 0-9, a-f (hex)

<table>
<thead>
<tr>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>00</td>
<td>4</td>
<td>0100</td>
<td>04</td>
<td>8</td>
<td>1000</td>
<td>08</td>
<td>12</td>
<td>1100</td>
<td>0c</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>01</td>
<td>5</td>
<td>0101</td>
<td>05</td>
<td>9</td>
<td>1001</td>
<td>09</td>
<td>13</td>
<td>1101</td>
<td>0d</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>02</td>
<td>6</td>
<td>0110</td>
<td>06</td>
<td>10</td>
<td>1010</td>
<td>0a</td>
<td>14</td>
<td>1110</td>
<td>0e</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>03</td>
<td>7</td>
<td>0111</td>
<td>07</td>
<td>11</td>
<td>1011</td>
<td>0b</td>
<td>15</td>
<td>1111</td>
<td>0f</td>
</tr>
</tbody>
</table>
Instruction Formats

Instructions are represented as 32-bit numbers (one word), broken into 6 fields

\[ R\text{-type instruction} \quad \text{add} \quad \$t0, \$s1, \$s2 \]

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td>opcode</td>
<td>source</td>
<td>source</td>
<td>dest</td>
<td>shift</td>
<td>amt</td>
</tr>
</tbody>
</table>

\[ I\text{-type instruction} \quad \text{lw} \quad \$t0, 32(\$s3) \]

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>constant</td>
</tr>
</tbody>
</table>
## Logical Operations

<table>
<thead>
<tr>
<th>Logical ops</th>
<th>C operators</th>
<th>Java operators</th>
<th>MIPS instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift Left</td>
<td><code>&lt;&lt;</code></td>
<td><code>&lt;&lt;</code></td>
<td><code>sll</code></td>
</tr>
<tr>
<td>Shift Right</td>
<td><code>&gt;&gt;</code></td>
<td><code>&gt;&gt;&gt;</code></td>
<td><code>srl</code></td>
</tr>
<tr>
<td>Bit-by-bit AND</td>
<td><code>&amp;</code></td>
<td><code>&amp;</code></td>
<td><code>and, andi</code></td>
</tr>
<tr>
<td>Bit-by-bit OR</td>
<td>`</td>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td>Bit-by-bit NOT</td>
<td><code>~</code></td>
<td><code>~</code></td>
<td><code>nor</code></td>
</tr>
</tbody>
</table>
Control Instructions

• Conditional branch: Jump to instruction L1 if register1 equals register2:
  \[ \text{beq register1, register2, L1} \]
  Similarly, \( \text{bne} \) and \( \text{slt} \) (set-on-less-than)

• Unconditional branch:
  \[ \text{j L1} \]
  \[ \text{jr } \$s0 \]  (useful for large case statements and big jumps)

Convert to assembly:

  \[
  \text{if } (i == j) \\
  \quad f = g+h; \\
  \text{else} \\
  \quad f = g-h;
  \]
Control Instructions

• Conditional branch: Jump to instruction L1 if register1 equals register2:

\[
\text{beq register1, register2, L1}
\]
Similarly, \text{bne} and \text{slt} (set-on-less-than)

• Unconditional branch:

\[
\text{j L1}
\]
\[
\text{jr } \$s0
\]  (useful for large case statements and big jumps)

Convert to assembly:

\[
\text{if (i == j)} \quad \text{bne } \$s3, \$s4, \text{Else}
\]
\[
f = g+h; \quad \text{add } \$s0, \$s1, \$s2
\]
\[
\text{else} \quad \text{j Exit}
\]
\[
f = g-h; \quad \text{Else: sub } \$s0, \$s1, \$s2
\]
\[
\text{Exit:}
\]