Programming Universal Computers
Instruction Sets

Lecture 5

Prof. Bienvenido Velez
What do we know?

From

Instruction Set Architecture

To

Processor Implementation

What Next?

How do we get here in the first place?

Instruction Set Design
Outline

• Virtual Machines: Interpretation Revisited
• Example: From HLL to Machine Code
• Implementing HLL Abstractions
  – Control structures
  – Data Structures
  – Procedures and Functions
# Virtual Machines (VM’s)

<table>
<thead>
<tr>
<th>Type of Virtual Machine</th>
<th>Examples</th>
<th>Instruction Elements</th>
<th>Data Elements</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Programs</td>
<td>Spreadsheet, Word Processor</td>
<td>Drag &amp; Drop, GUI ops, macros</td>
<td>cells, paragraphs, sections</td>
<td>Visual, Graphical, Interactive</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Application Specific Abstractions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Easy for Humans</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Hides HLL Level</td>
</tr>
<tr>
<td>High-Level Language</td>
<td>C, C++, Java, FORTRAN, Pascal</td>
<td>if-then-else, procedures, loops</td>
<td>arrays, structures</td>
<td>Modular, Structured, Model Human</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Language/Thought</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>General Purpose Abstractions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Hides Lower Levels</td>
</tr>
<tr>
<td>Assembly-Level</td>
<td>SPIM, MASM</td>
<td>directives, pseudo-instructions, macros</td>
<td>registers, labelled memory cells</td>
<td>Symbolic Instructions/Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Hides some machine details like</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>alignment, address calculations</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Exposes Machine ISA</td>
</tr>
<tr>
<td>Machine-Level (ISA)</td>
<td>MIPS, Intel 80x86</td>
<td>load, store, add, branch</td>
<td>bits, binary addresses</td>
<td>Numeric, Binary</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Difficult for Humans</td>
</tr>
</tbody>
</table>

- **Virtual Machines (VM’s)**: The study of computer systems that abstract away the hardware details and make it easier for humans to interact with software. VMs are crucial in modern computing, allowing for the execution of different operating systems and applications on a single machine.

- **Types of Virtual Machines**:
  - **Application Programs**: These are programs that run directly on the operating system and interact with the user interface. Examples include spreadsheets and word processors.
  - **High-Level Language**: These are programming languages like C++, Java, and FORTRAN, which are closer to human language and abstract away some of the machine details.
  - **Assembly-Level**: These are assembly languages like MIPS and Intel 80x86, which are very close to the machine level and require detailed knowledge of the hardware.

- **Data Elements**:
  - Cells, paragraphs, sections: These are data elements that are easily understandable by humans, such as data stored in documents.
  - Arrays, structures: These are more complex data structures that are still understandable by humans.
  - Registers, labelled memory cells: These are data elements that are very close to the hardware and require specific knowledge.

- **Comments**:
  - Visual, Graphical, Interactive: VMs provide a more interactive and visual interface for users.
  - Modular, Structured, Model Human Language/Thought: These VMs abstract away some of the machine details, making them easier to use.
  - Symbolic Instructions/Data: These VMs provide a more symbolic and human-readable representation of the data.
  - Numeric, Binary: These VMs are very close to the hardware and are difficult for humans to use directly.
Computer Science in Perspective

A CORE theme all throughout Computer Science
Computing Integer Division
Iterative C++ Version

```cpp
int a = 12;
int b = 4;
int result = 0;
main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}
```

We ignore procedures and I/O for now
# Easy I

## A Simple Accumulator Processor

### Instruction Set Architecture (ISA)

## Instruction Set

<table>
<thead>
<tr>
<th>Symbolic Name</th>
<th>Opcode</th>
<th>Action $I=0$</th>
<th>Symbolic Name</th>
<th>Action $I=1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp</td>
<td>00 000</td>
<td>AC ? not AC</td>
<td>Comp</td>
<td>AC &lt;- not AC</td>
</tr>
<tr>
<td>ShR</td>
<td>00 001</td>
<td>AC ? AC / 2</td>
<td>ShR</td>
<td>AC ? AC / 2</td>
</tr>
<tr>
<td>BrN</td>
<td>00 010</td>
<td>AC &lt; 0 ? PC ? X</td>
<td>BrN</td>
<td>AC &lt; 0 ? PC ? MEM[X]</td>
</tr>
<tr>
<td>Jump</td>
<td>00 011</td>
<td>PC ? X</td>
<td>Jump</td>
<td>PC ? MEM[X]</td>
</tr>
<tr>
<td>Store</td>
<td>00 100</td>
<td>MEM[X] ? AC</td>
<td>Store</td>
<td>MEM[MEM[X]] ? AC</td>
</tr>
<tr>
<td>Load</td>
<td>00 101</td>
<td>AC ? MEM[X]</td>
<td>Load</td>
<td>AC ? MEM[MEM[X]]</td>
</tr>
<tr>
<td>Andc</td>
<td>00 110</td>
<td>AC ? AC and X</td>
<td>And</td>
<td>AC ? AC and MEM[X]</td>
</tr>
<tr>
<td>Addc</td>
<td>00 111</td>
<td>AC ? AC + X</td>
<td>Add</td>
<td>AC ? AC + MEM[X]</td>
</tr>
</tbody>
</table>
int a = 12;
int b = 4;
int result = 0;
main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}
Computing Integer Division
Iterative C++ Version

```cpp
int a = 12;
int b = 4;
int result = 0;
main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}
```

Translate Data: Global Layout

```
0:       andc 0    # AC = 0
         addc 12
         store 1000    # a = 12 (a stored @ 1000)
         andc 0    # AC = 0
         addc 4
         store 1004    # b = 12 (b stored @ 1004)
         andc 0    # AC = 0
         store 1008    # result = 0 (result @ 1008)
```

Issues
• Memory allocation
• Data Alignment
• Data Sizing
Computing Integer Division
Iterative C++ Version

```
int a = 12;
int b = 4;
int result = 0;
main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}
```

Translate Code: Conditionals
If-Then

```
0:  andc 0     # AC = 0
addc 12     # a = 12 (a stored @ 1000)
store 1000   # AC = 0
addc 4      # b = 12 (b stored @ 1004)
store 1004   # AC = 0
store 1008   # result = 0 (result @ 1008)
main: load 1004 # compute a – b in AC
comp         # using 2’s complement add
addc 1       # exit if AC negative
add 1004
brn exit
```

Issues
• Must translate HLL boolean expression into ISA-level branching condition
Computing Integer Division
Iterative C++ Version

```
int a = 12;
int b = 4;
int result = 0;

main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}
```

Translate Code: Iteration (loops)

```
0:  andc 0  # AC = 0
    addc 12  # AC = 0
    store 1000  # a = 12 (a stored @ 1000)
    andc 0  # AC = 0
    addc 4  # AC = 0
    store 1004  # b = 12 (b stored @ 1004)
    andc 0  # AC = 0
    store 1008  # result = 0 (result @ 1008)

main:  load 1004  # compute a - b in AC
        comp  # using 2’s complement add
        addc 1  # AC = 0
        add 1004
        brn exit  # exit if AC negative

loop:   load 1000
        brn endloop

endloop:  jump loop

exit:
```
Computing Integer Division
Iterative C++ Version

```cpp
int a = 12;
int b = 4;
int result = 0;
main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}
```

Translate Code: Arithmetic Ops

<table>
<thead>
<tr>
<th>C++ Code</th>
<th>Easy-I Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a = 12;</td>
<td>store 1000 # a = 12 (a stored @ 1000)</td>
</tr>
<tr>
<td>int b = 4;</td>
<td>store 1004 # b = 12 (b stored @ 1004)</td>
</tr>
<tr>
<td>int result = 0;</td>
<td>store 1008 # result = 0 (result @ 1008)</td>
</tr>
<tr>
<td>main () {</td>
<td>main: load 1004 # compute a – b in AC</td>
</tr>
<tr>
<td>if (a &gt;= b) {</td>
<td>comp</td>
</tr>
<tr>
<td>while (a &gt; 0) {</td>
<td>addc 1</td>
</tr>
<tr>
<td>a = a - b;</td>
<td>add 1004</td>
</tr>
<tr>
<td>result ++;</td>
<td>brn exit</td>
</tr>
<tr>
<td>}</td>
<td>main: load 1000</td>
</tr>
<tr>
<td>}</td>
<td>comp</td>
</tr>
<tr>
<td></td>
<td>add 1004</td>
</tr>
<tr>
<td></td>
<td>brn endloop</td>
</tr>
<tr>
<td></td>
<td>load 1004</td>
</tr>
<tr>
<td></td>
<td>comp</td>
</tr>
<tr>
<td></td>
<td>add 1004 # Uses indirect bit I = 1</td>
</tr>
<tr>
<td></td>
<td>jump loop</td>
</tr>
<tr>
<td></td>
<td>endloop:</td>
</tr>
<tr>
<td></td>
<td>exit:</td>
</tr>
</tbody>
</table>
### Computing Integer Division

**Iterative C++ Version**

```cpp
int a = 12;
int b = 4;
int result = 0;
main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}
```

### Translate Code: Assignments

<table>
<thead>
<tr>
<th>C++ Code</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int a = 12;</code></td>
<td><code>andc 0</code> # AC = 0</td>
</tr>
<tr>
<td><code>int b = 4;</code></td>
<td><code>addc 12</code> # a = 12 (a stored @ 1000)</td>
</tr>
<tr>
<td><code>int result = 0;</code></td>
<td><code>andc 0</code> # AC = 0</td>
</tr>
<tr>
<td><code>main () {</code></td>
<td><code>store 1000</code> # b = 12 (b stored @ 1004)</td>
</tr>
<tr>
<td><code>    if (a &gt;= b) {</code></td>
<td><code>addc 4</code> # AC = 0</td>
</tr>
<tr>
<td><code>        while (a &gt; 0) {</code></td>
<td><code>store 1004</code> # result = 0 (result @ 1008)</td>
</tr>
<tr>
<td><code>            a = a - b;</code></td>
<td><code>andc 0</code> # AC = 0</td>
</tr>
<tr>
<td><code>            result ++;</code></td>
<td><code>store 1008</code> # result = 0 (result @ 1008)</td>
</tr>
<tr>
<td><code>        }</code></td>
<td><code>main: load 1004</code> # compute a - b in AC</td>
</tr>
<tr>
<td><code>    }</code></td>
<td><code>comp</code> # using 2’s complement add</td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>addc 1</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>add 1004</code></td>
</tr>
<tr>
<td><code>)</code></td>
<td><code>brn exit</code> # exit if AC negative</td>
</tr>
</tbody>
</table>

**Easy-I Assembly Language**

```
0: andc 0 # AC = 0
    addc 12 # a = 12 (a stored @ 1000)
    store 1000 # a = 12 (a stored @ 1000)
    andc 0 # AC = 0
    store 1004 # b = 12 (b stored @ 1004)
    andc 0 # AC = 0
    store 1008 # result = 0 (result @ 1008)
main: load 1004 # compute a - b in AC
comp # using 2’s complement add
addc 1
add 1004
brn exit # exit if AC negative
loop: load 1000
brn endloop
load 1004
comp
add 1004 # Uses indirect bit I = 1
store 1000
jump loop
endloop: jump loop
exit:
```
# Computing Integer Division

## Iterative C++ Version

```c++
int a = 12;
int b = 4;
int result = 0;
main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}
```

## Translate Code: Increments

<table>
<thead>
<tr>
<th>C++ Code</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a = 12;</td>
<td>addc 12 # a = 12 (a stored @ 1000)</td>
</tr>
<tr>
<td>int b = 4;</td>
<td>andc 0 # AC = 0</td>
</tr>
<tr>
<td>int result = 0;</td>
<td>addc 4 # b = 12 (b stored @ 1004)</td>
</tr>
<tr>
<td>main () {</td>
<td>store 1004 # AC = 0</td>
</tr>
<tr>
<td>if (a &gt;= b) {</td>
<td>store 1008 # result = 0 (result @ 1008)</td>
</tr>
<tr>
<td>while (a &gt; 0) {</td>
<td>main: load 1004 # compute a – b in AC</td>
</tr>
<tr>
<td>a = a - b;</td>
<td>comp</td>
</tr>
<tr>
<td>result ++;</td>
<td>addc 1 # using 2’s complement add</td>
</tr>
<tr>
<td>}</td>
<td>add 1004</td>
</tr>
<tr>
<td>}</td>
<td>brn exit # exit if AC negative</td>
</tr>
<tr>
<td>}</td>
<td>loop: load 1000</td>
</tr>
<tr>
<td></td>
<td>brn endloop</td>
</tr>
<tr>
<td></td>
<td>load 1004 comp</td>
</tr>
<tr>
<td></td>
<td>add 1004</td>
</tr>
<tr>
<td></td>
<td>store 1000</td>
</tr>
<tr>
<td></td>
<td>load 1012 # Uses indirect bit I = 1</td>
</tr>
<tr>
<td></td>
<td>addc 1</td>
</tr>
<tr>
<td></td>
<td>store 1012</td>
</tr>
<tr>
<td></td>
<td>jump loop</td>
</tr>
<tr>
<td></td>
<td>endloop:</td>
</tr>
<tr>
<td></td>
<td>exit:</td>
</tr>
</tbody>
</table>

---

Spring 2002
## Computing Integer Division

### Easy I Machine Code

#### Data

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>a</td>
</tr>
<tr>
<td>1004</td>
<td>b</td>
</tr>
<tr>
<td>1008</td>
<td>result</td>
</tr>
</tbody>
</table>

#### Program

<table>
<thead>
<tr>
<th>Address</th>
<th>I Bit</th>
<th>Opcode (binary)</th>
<th>X (base 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 110</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>00 111</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>00 100</td>
<td>1000</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>00 110</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>00 111</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00 100</td>
<td>1004</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>00 110</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>00 100</td>
<td>1008</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>00 101</td>
<td>1004</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>00 000</td>
<td>unused</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>00 111</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>00 111</td>
<td>1004</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>00 010</td>
<td>44</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>00 101</td>
<td>1000</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td>00 010</td>
<td>44</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>00 101</td>
<td>1004</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>00 000</td>
<td>unused</td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td>00 111</td>
<td>1004</td>
</tr>
<tr>
<td>36</td>
<td>0</td>
<td>00 100</td>
<td>1008</td>
</tr>
<tr>
<td>38</td>
<td>0</td>
<td>00 101</td>
<td>1012</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>00 111</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>0</td>
<td>00 101</td>
<td>1000</td>
</tr>
<tr>
<td>44</td>
<td>0</td>
<td>00 011</td>
<td>26</td>
</tr>
</tbody>
</table>

#### Challenge

Make this program as small and fast as possible.
The MIPS Architecture

- Reduced Instruction Set Computer (RISC)
- 32 general purpose registers
- Load-store architecture: Operands in registers
- Simple and Uniform instruction formats
  - **R Format**: Arithmetic/Logic operations on registers
  - **I Format**: Branches, loads and stores
SPIM Assembler

- Symbolic Labels
- Assembler Directives
- Pseudo-Instructions
- Macros
Computing Integer Division
Iterative C++ Version

### C++

```c++
int a = 12;
int b = 4;
int result = 0;
main () {
if (a >= b) {
    while (a > 0) {
        a = a - b;
        result ++;
    }
}
}
```

### SPIM Assembler

```assembly
.data
a .word 12
b .word 4
result .word 0
.glob main #global symbol
```
Computing Integer Division
Iterative C++ Version

C++

```cpp
int a = 12;
int b = 4;
int result = 0;
main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}
```

SPIM Assembler

```assembly
.data                # data segment
a:       .word  12
b:       .word  4
result:  .word  0
.glob main          # global symbols
    .text            # text segment
main:
```

Step 2: Translate If-Then-Else