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 Application Specific Abstractions Easy for Humans 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 Language/Thought General Purpose Abstractions 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 Hides some machine details like alignment, address calculations 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 Difficult for Humans</td>
</tr>
</tbody>
</table>
Computer Science in Perspective

Application Programs
High-Level Language
Assembly Language (ISA)

People

Computer Human Interaction, User Interfaces
CS1/CS2, Programming, Data Structures
Programming Languages, Compilers
Computer Architecture

INTERPRETATION
A CORE theme all throughout Computer Science

Spring 2003
INEL 4206 Microprocessors
Lecture 5
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 ++;
        }
    }
}
## 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>$AC \leftarrow \text{not } AC$</td>
<td>Comp</td>
<td>$AC \leftarrow \text{not } AC$</td>
</tr>
<tr>
<td>ShR</td>
<td>00 001</td>
<td>$AC \leftarrow AC / 2$</td>
<td>ShR</td>
<td>$AC \leftarrow AC / 2$</td>
</tr>
<tr>
<td>BrNi</td>
<td>00 010</td>
<td>$AC &lt; 0 \Rightarrow PC \leftarrow X$</td>
<td>BrN</td>
<td>$AC &lt; 0 \Rightarrow PC \leftarrow MEM[X]$</td>
</tr>
<tr>
<td>Jump</td>
<td>00 011</td>
<td>$PC \leftarrow X$</td>
<td>Jump</td>
<td>$PC \leftarrow MEM[X]$</td>
</tr>
<tr>
<td>Storei</td>
<td>00 100</td>
<td>$MEM[X] \leftarrow AC$</td>
<td>Store</td>
<td>$MEM[MEM[X]] \leftarrow AC$</td>
</tr>
<tr>
<td>Loadi</td>
<td>00 101</td>
<td>$AC \leftarrow MEM[X]$</td>
<td>Load</td>
<td>$AC \leftarrow MEM[MEM[X]]$</td>
</tr>
<tr>
<td>Andi</td>
<td>00 110</td>
<td>$AC \leftarrow AC \text{ and } X$</td>
<td>And</td>
<td>$AC \leftarrow AC \text{ and } MEM[X]$</td>
</tr>
<tr>
<td>Addi</td>
<td>00 111</td>
<td>$AC \leftarrow AC + X$</td>
<td>Add</td>
<td>$AC \leftarrow AC + MEM[X]$</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 ++;
        }
    }
}
```

Spring 2003
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:   andi   0 # AC = 0
     addi   12
     storei 1000 # a = 12 (a stored @ 1000)
     andi   0 # AC = 0
     addi   4
     storei 1004 # b = 4 (b stored @ 1004)
     andi   0 # AC = 0
     storei 1008 # result = 0 (result @ 1008)
```

Issues
- Memory allocation
- Data Alignment
- Data Sizing

Spring 2003
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: Conditionals
If-Then

```assembly
0: andi 0 # AC = 0
addi 12
storei 1000 # a = 12 (a stored @ 1000)
andi 0 # AC = 0
addi 4
storei 1004 # b = 4 (b stored @ 1004)
andi 0 # AC = 0
storei 1008 # result = 0 (result @ 1008)

main: loadi 1004 # compute a - b in AC
comp # using 2’s complement add
addi 1
add 1000
brni exit # exit if AC negative

exit:
```

Issues
• Must translate HLL boolean expression into ISA-level branching condition
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: Iteration (loops)

```
0: andi 0 # AC = 0
addi 12
storei 1000 # a = 12 (a stored @ 1000)
andi 0 # AC = 0
addi 4
storei 1004 # b = 4 (b stored @ 1004)
andi 0 # AC = 0
storei 1008 # result = 0 (result @ 1008)
main: loadi 1004 # compute a - b in AC
comp # using 2’s complement add
addi 1
add 1000
brni exit # exit if AC negative
loop: loadi 1000
brni endloop

jump loop
endloop:
exit:
```
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: Arithmetic Ops

0:  andi   0      # AC = 0
    addi   12
    storei 1000   # a = 12  (a stored @ 1000)
    andi   0      # AC = 0
    addi   4
    storei 1004   # b = 4   (b stored @ 1004)
    andi   0      # AC = 0
    storei 1008   # result = 0  (result @ 1008)
main: loadi 1004  # compute a - b in AC
      comp
      addi   1
      add    1000
      brni   exit  # exit if AC negative
loop: loadi 1000
      brni   endloop
      loadi 1004  # compute a - b in AC
      comp
      addi   1
      add    1000  # Uses indirect bit I = 1
jumpi  loop
endloop: exit:

C++
HLL

Easy-I
Assembly Language

Spring 2003
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: Assignments

0: andi 0 # AC = 0
addi 12
storei 1000 # a = 12 (a stored @ 1000)
andi 0 # AC = 0
addi 4
storei 1004 # b = 4 (b stored @ 1004)
andi 0 # AC = 0
storei 1008 # result = 0 (result @ 1008)
main: loadi 1004 # compute a – b in AC
comp # using 2’s complement add
addi 1
add 1000
brni exit # exit if AC negative
loop: loadi 1000
brni endloop
loadi 1004 # compute a – b in AC
comp # using 2’s complement add
addi 1
add 1000 # Uses indirect bit I = 1
storei 1000
jump loop
endloop:
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: Increments

```
0:  andi   0       # AC = 0
    addi   12
    storei 1000  # a = 12 (a stored @ 1000)
    andi   0       # AC = 0
    addi   4
    storei 1004  # b = 4  (b stored @ 1004)
    andi   0       # AC = 0
    storei 1008  # result = 0 (result @ 1008)
main:  loadi  1004  # compute a – b in AC
       comp                      # using 2’s complement add
       addi   1
       add    1000
       brni   exit               # exit if AC negative
loop:  loadi  1000
       brni   endloop
       loadi  1004  # compute a – b in AC
       comp                      # using 2’s complement add
       addi   1
       add    1000               # Uses indirect bit I = 1
       storei 1000
       loadi  1008  # result = result + 1
       addi   1
       storei 1008
       jumpi  loop
endloop:
exit:
```
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>1000</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>00 010</td>
<td>46</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>46</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>0</td>
<td>00 111</td>
<td>1</td>
</tr>
<tr>
<td>36</td>
<td>0</td>
<td>00 100</td>
<td>1000</td>
</tr>
<tr>
<td>38</td>
<td>0</td>
<td>00 101</td>
<td>1008</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>00 100</td>
<td>1008</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.
Computing Integer Division
Iterative C++ Version

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

Revised Version
Optimization at the HLL level

<table>
<thead>
<tr>
<th>Assembly Language</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>andi 0</td>
<td># AC = 0</td>
</tr>
<tr>
<td></td>
<td>addi 12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>storei 1000</td>
<td># a = 12 (a stored @ 1000)</td>
</tr>
<tr>
<td></td>
<td>andi 0</td>
<td># AC = 0</td>
</tr>
<tr>
<td></td>
<td>addi 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>storei 1004</td>
<td># b = 4 (b stored @ 1004)</td>
</tr>
<tr>
<td></td>
<td>andi 0</td>
<td># AC = 0</td>
</tr>
<tr>
<td></td>
<td>storei 1008</td>
<td># result = 0 (result @ 1008)</td>
</tr>
<tr>
<td>main:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop:</td>
<td>loadi 1004</td>
<td># compute a – b in AC</td>
</tr>
<tr>
<td></td>
<td>comp</td>
<td># using 2’s complement add</td>
</tr>
<tr>
<td></td>
<td>addi 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add 1000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>brni exit</td>
<td># exit if AC negative</td>
</tr>
<tr>
<td></td>
<td>loadi 1004</td>
<td># compute a – b in AC</td>
</tr>
<tr>
<td></td>
<td>comp</td>
<td># using 2’s complement add</td>
</tr>
<tr>
<td></td>
<td>addi 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add 1000</td>
<td># Uses indirect bit I = 1</td>
</tr>
<tr>
<td></td>
<td>storei 1000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>loadi 1000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>storei 1008</td>
<td></td>
</tr>
<tr>
<td>endloop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exit:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Translating Conditional Expressions

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

Translating Logical Expressions

```
loop exit condition

~(a >= b) ⇔ ~((a - b) >= 0) ⇔ ((a - b) < 0)
```

What if Loop Exit Condition was:

```
~(a < b) ⇔
```

Computing Integer Division
Iterative C++ Version

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

C++

HLL

Easy-I
Assembly Language

Peephole Optimization
Optimization at Assembly level

0:  andi   0  # AC = 0
    addi   12
    storei 1000  # a = 12 (a stored @ 1000)
    andi   0  # AC = 0
    addi   4
    storei 1004  # b = 4  (b stored @ 1004)
    andi   0  # AC = 0
    storei 1008  # result = 0 (result @ 1008)

main:
    loadi 1004  # compute a – b in AC
    comp
    addi   1
    add    1000
    brni   exit  # exit if AC negative

endloop:

exit:

Iterative C++ Version

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

Computing Integer Division
Iterative C++ Version
The MIPS Architecture
ISA at a Glance

- Reduced Instruction Set Computer (RISC)
- 32 general purpose 32-bit registers
- Load-store architecture: Operands in registers
- Byte Addressable
- 32-bit address space
## The MIPS Architecture

### 32 Register Set (32-bit registers)

<table>
<thead>
<tr>
<th>Register #</th>
<th>Reg Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>r0</td>
<td>Zero constant</td>
</tr>
<tr>
<td>r4-r7</td>
<td>a0-a3</td>
<td>Function arguments</td>
</tr>
<tr>
<td>r1</td>
<td>at</td>
<td>Reserved for Operating Systems</td>
</tr>
<tr>
<td>r30</td>
<td>fp</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>r28</td>
<td>gp</td>
<td>Global memory pointer</td>
</tr>
<tr>
<td>r26-r27</td>
<td>k0-k1</td>
<td>Reserved for OS Kernel</td>
</tr>
<tr>
<td>r31</td>
<td>ra</td>
<td>Function return address</td>
</tr>
<tr>
<td>r16-r23</td>
<td>s0-s7</td>
<td>Callee saved registers</td>
</tr>
<tr>
<td>r29</td>
<td>sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>r8-r15</td>
<td>t0-t7</td>
<td>Temporary variables</td>
</tr>
<tr>
<td>r24-r25</td>
<td>t8-t9</td>
<td>Temporary variables</td>
</tr>
<tr>
<td>r2-r3</td>
<td>v0-v1</td>
<td>Function return values</td>
</tr>
</tbody>
</table>
The MIPS Architecture
Main Instruction Formats

Simple and uniform 32-bit 3-operand instruction formats

–R Format: Arithmetic/Logic operations on registers

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

–I Format: Branches, loads and stores

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>Address/Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

–J Format: Jump Instruction

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>Address/Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
The MIPS Architecture

Examples of **Native** Instruction Set

<table>
<thead>
<tr>
<th>Instruction Group</th>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic/Logic</strong></td>
<td>add $s1,$s2,$s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td></td>
<td>addi $s1,$s2,K</td>
<td>$s1 = $s2 + K</td>
</tr>
<tr>
<td><strong>Load/Store</strong></td>
<td>lw $s1,K($s2)</td>
<td>$s1 = MEM[$s2+$K]</td>
</tr>
<tr>
<td></td>
<td>sw $s1,K($s2)</td>
<td>MEM[$s2+$K] = $s1</td>
</tr>
<tr>
<td><strong>Jumps and</strong></td>
<td>beq $s1,$s2,K</td>
<td>if ($s1=$s2) goto PC + 4 + K</td>
</tr>
<tr>
<td><strong>Conditional</strong></td>
<td>slt $s1,$s2,$s3</td>
<td>if ($s2&lt;$s3) $s1=1 else $s1=0</td>
</tr>
<tr>
<td><strong>Branches</strong></td>
<td>j K</td>
<td>goto K</td>
</tr>
<tr>
<td><strong>Procedures</strong></td>
<td>jal K</td>
<td>$ra = PC + 4; goto K</td>
</tr>
<tr>
<td></td>
<td>jr $ra</td>
<td>goto $ra</td>
</tr>
</tbody>
</table>
### The SPIM Assembler

#### Examples of Pseudo-Instruction Set

<table>
<thead>
<tr>
<th>Instruction Group</th>
<th>Syntax</th>
<th>Translates to:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic/Logic</strong></td>
<td>neg $s1, $s2</td>
<td>sub $s1, $r0, $s2</td>
</tr>
<tr>
<td></td>
<td>not $s1, $s2</td>
<td>nor $17, $18, $0</td>
</tr>
<tr>
<td><strong>Load/Store</strong></td>
<td>li $s1, K</td>
<td>ori $s1, $0, K</td>
</tr>
<tr>
<td></td>
<td>la $s1, K</td>
<td>lui $at, 152</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ori $s1, $at, -27008</td>
</tr>
<tr>
<td></td>
<td>move $s1, $s2</td>
<td></td>
</tr>
<tr>
<td><strong>Jumps and Conditional Branches</strong></td>
<td>bgt $s1, $s2, K</td>
<td>slt $at, $s1, $s2</td>
</tr>
<tr>
<td></td>
<td>sge $s1, $s2, $s3</td>
<td>bne $at, $0, K</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Instructions:** translated to native instructions by Assembler
# The SPIM Assembler

## Examples of Assembler Directives

<table>
<thead>
<tr>
<th>Group</th>
<th>Directive</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Segmentation</td>
<td>.data &lt;addr&gt;</td>
<td>Data Segment starting at</td>
</tr>
<tr>
<td></td>
<td>.text &lt;addr&gt;</td>
<td>Text (program) Segment</td>
</tr>
<tr>
<td></td>
<td>.stack &lt;addr&gt;</td>
<td>Stack Segment</td>
</tr>
<tr>
<td></td>
<td>.ktext &lt;addr&gt;</td>
<td>Kernel Text Segment</td>
</tr>
<tr>
<td></td>
<td>.kdata &lt;addr&gt;</td>
<td>Kernel Data Segment</td>
</tr>
<tr>
<td>Data Allocation</td>
<td>x: .word &lt;value&gt;</td>
<td>Allocates 32-bit variable</td>
</tr>
<tr>
<td></td>
<td>x: .byte &lt;value&gt;</td>
<td>Allocates 8-bit variable</td>
</tr>
<tr>
<td></td>
<td>x: .ascii “hello”</td>
<td>Allocates 8-bit cell array</td>
</tr>
<tr>
<td>Other</td>
<td>.globl x</td>
<td>x is external symbol</td>
</tr>
</tbody>
</table>

**Assembler Directives**: Provide assembler additional info to generate machine code.
Handy MIPS ISA References

- Appendix A: Patterson & Hennessy
- SPIM ISA Summary on class website
- Patterson & Hennessy Back Cover
The MIPS Architecture
Memory Model

32-bit byte addressable address space
Computing Integer Division
Iterative C++ Version

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

MIPS/SPIM Version

```mips
.data # Use HLL program as a comment
x: .word 12 # int x = 12;
y: .word 4 # int y = 4;
res: .word 0 # int res = 0;
.globl main
.text
main: la $s0, x # Allocate registers for globals
    lw $s1, 0($s0) # x in $s1
    lw $s2, 4($s0) # y in $s2
    lw $s3, 8($s0) # res in $s3
while: bgt $s2, $s1, endwhile # while (x >= y) {
    sub $s1, $s1, $s2 # x = x - y;
    addi $s3, $s3, 1 # res ++;
    j while # }
endwhile:
    la $s0, x # Update variables in memory
    sw $s1, 0($s0)
    sw $s2, 4($s0)
    sw $s3, 8($s0)
```

Computing Integer Division
Iterative C++ Version

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

MIPS/SPIM Version
Input/Output in SPIM

```mips
.data
x: .word 12 # int x = 12;
y: .word 4 # int y = 4;
res: .word 0 # int res = 0;
pf1: .asciiz "Result = "
globl main
.text
main: la $s0, x # Allocate registers for globals
    lw $s1, 0($s0) #   x in $s1
    lw $s2, 4($s0) #   y in $s2
    lw $s3, 8($s0) #   res in $s3
while: bgt $s2, $s1, endwhile # while (x >= y) {
    sub $s1, $s1, $s2 #   x = x - y;
    addi $s3, $s3, 1 #   res ++;
    j while # }
endwhile:
    la $a0, pf1 # printf("Result = %d \n");
    li $v0, 4 # //system call to print_str
    syscall
move $a0, $s3 # //system call to print_int
syscall
la $s0, x # Update variables in memory
sw $s1, 0($s0)
sw $s2, 4($s0)
sw $s3, 8($s0)
```

Spring 2003
SPIM Assembler Abstractions

• Symbolic Labels
  – Instruction addresses and memory locations

• Assembler Directives
  – Memory allocation
  – Memory segments

• Pseudo-Instructions
  – Extend native instruction set without complicating architecture

• Macros
Implementing Procedures

• Why procedures?
  – Abstraction
  – Modularity
  – Code re-use

• Initial Goal
  – Write segments of assembly code that can be re-used, or “called” from different points in the main program.
  – KISS: Keep It Simple Stupid:
    • no parameters, no recursion, no locals, no return values
Procedure Linkage
Approach I

• Problem
  – procedure must determine where to return after servicing the call

• Solution: Architecture Support
  – Add a jump instruction that saves the return address in some place known to callee
    • MIPS: jal instruction saves return address in register $ra
  – Add an instruction that can jump to return address
    • MIPS: jr instruction jumps to the address contained in its argument register
Computing Integer Division (Procedure Version)

Iterative C++ Version

```cpp
int a = 0;
int b = 0;
int res = 0;
main () {
a = 12;
b = 5;
res = 0;
div();
printf("Res = %d",res);
}

void div(void) {
while (a >= b) {
a = a - b;
res ++;
}
}
```

MIPS Assembly Language

```
data
x: .word 0
y: .word 0
res: .word 0
pf1: .asciiz "Result = 
pf2: .asciiz "Remainder = 
.globl main
.text
main: # int main() {
    // main assumes registers sx unused
    la $s0, x # x = 12;
    li $s1, 12
    sw $s1, 0($s0)
    la $s0, y # y = 5;
    li $s2, 5
    sw $s2, 0($s0)
    la $s0, res # res = 0;
    li $s3, 0
    sw $s3, 0($s0)
    jal d # div();
    lw $s3, 0($s0)
    la $a0, pf1 # printf("Result = %d \n");
    li $v0, 4 # //system call to print_str
    syscall
    move $a0, $s3
    li $v0, 1 # //system call to print_int
    syscall
    la $a0, pf2 # printf("Remainder = %d \n");
    li $v0, 4 # //system call to print_str
    syscall
    move $a0, $s1
    li $v0, 1 # //system call to print_int
    syscall
    jr $ra # return // TO Operating System
```
Computing Integer Division (Procedure Version)
Iterative C++ Version

```c
int a = 0;
int b = 0;
int res = 0;
main () {
    a = 12;
b = 5;
    res = 0;
div();
    printf("Res = %d",res);
}
void div(void) {
    while (a >= b) {
        a = a - b;
        res ++;
    }
}
```

MIPS Assembly Language
# div function
# PROBLEM: Must save args and registers before using them
d:
#   // Allocate registers for globals
    # void d(void) {
    #   // x in $s1
        la $s0, x
lw $s1, 0($s0)
la $s0, y
lw $s2, 0($s0)
la $s0, res
lw $s3, 0($s0)
while: bgt $s2, $s1, ewhile
        # while (x <= y) {
        sub $s1, $s1, $s2
        addi $s3, $s3, 1
        j while
        # }
        # // Update variables in memory
ewhile: 
        la $s0, x
sw $s1, 0($s0)
la $s0, y
sw $s2, 0($s0)
la $s0, res
sw $s3, 0($s0)
enddiv: jr $ra
        # return;
        # }
```
Pending Problems With Linkage Approach I

- Registers shared by all procedures
  - procedures must save/restore registers (use stack)
- Procedures should be able to call other procedures
  - save multiple return addresses (use stack)
- Lack of parameters forces access to globals
  - pass parameters in registers
- Recursion requires multiple copies of local data
  - store multiple procedure activation records (use stack)
- Need a convention for returning function values
  - return values in registers
Recursion Basics

```c
int fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        return (fact(n-1) * n);
    }
}
```

```
int fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        return (fact(n-1) * n);
    }
}
```
Solution: Use Stacks of Procedure Frames

• Stack frame contains:
  – Saved arguments
  – Saved registers
  – Return address
  – Local variables
Anatomy of a Stack Frame

Contract: Every function must leave the stack the way it found it
Example: Function Linkage using Stack Frames

int x = 0;
int y = 0;
int res = 0;
main () {
    x = 12;
y = 5;
    res = div(x,y);
    printf("Res = %d",res);
}
int div(int a,int b) {
    int res = 0;
    if (a >= b) {
        res = div(a-b,b) + 1;
    } else {
        res = 0;
    }
    return res;
}
Example: Function Linkage using Stack Frames
MIPS: Procedure Linkage Summary

• First 4 arguments passed in $a0-$a3
• Other arguments passed on the stack
• Return address passed in $ra
• Return value(s) returned in $v0-$v1
• Sx registers saved by callee
• Tx registers saved by caller