Instruction Set Architectures
Part II: x86, RISC, and CISC

Readings: 2.16-2.18
Quiz 1 Recap

**Question 3: Multiple Answer**

Which of the following statements are true about instruction set architectures?

**Correct Answers**

- The ISA defines a contract between the program and the hardware.
- The ISA determines how the processor will implement the operations the ISA defines.
- The ISA defines a set of basic operations that the processor can perform.
- The ISA that a computer uses determines which programming languages can be used to program the computer.

<table>
<thead>
<tr>
<th>Percent Correct</th>
<th>Percent Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>77.778%</td>
<td>22.222%</td>
</tr>
<tr>
<td>83.333%</td>
<td>16.667%</td>
</tr>
<tr>
<td>95.833%</td>
<td>4.167%</td>
</tr>
<tr>
<td>94.444%</td>
<td>5.556%</td>
</tr>
</tbody>
</table>

Average Score 0.59 points
Quiz 1 Recap

**Question 4: Multiple Answer**

Which of the following statements are true about the MIPS ISA?

**Correct Answers**

- MIPS is the dominant ISA in use in computers today.
- MIPS instructions are 8 bytes long.
- MIPS instructions can read up to 2 register values from the register file and write at most 1 register.
- Regardless of instruction type, the opcode is always in the same location in MIPS instructions.
- MIPS is more complex than the x86 ISA.

Average Score 0.67 points

<table>
<thead>
<tr>
<th>Percent Correct</th>
<th>Percent Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>97.222%</td>
<td>2.778%</td>
</tr>
<tr>
<td>90.278%</td>
<td>9.722%</td>
</tr>
<tr>
<td>81.944%</td>
<td>18.056%</td>
</tr>
<tr>
<td>97.222%</td>
<td>2.778%</td>
</tr>
<tr>
<td>100%</td>
<td>0%</td>
</tr>
</tbody>
</table>
Quiz 1 Recap

Question 5: True/False

Moore's Law specifies that processor performance double every 18-24 months.

<table>
<thead>
<tr>
<th>Correct Answers</th>
<th>Percent Answered</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>49.315%</td>
</tr>
<tr>
<td>False</td>
<td>47.945%</td>
</tr>
</tbody>
</table>

Unanswered 2.74%

Average Score 0.48 points
Quiz 1 Recap

Question 6: Multiple Choice

Performance scaling of microprocessors slowed around 2004 for which reason?

- Correct
  - Moore's Law ended.
  - Power constraints caused the end of clock speed scaling.
  - Users stopped being interested in processor performance.
  - Chip manufacturers started building chip-multiprocessor because they were easier to program.

Unanswered

Average Score 0.93 points

Percent Answered
- 0%
- 93.151%
- 0%
- 4.11%
- 2.74%
Question 7: Essay

In class I suggested (joking) that if I had two cars, I should be able to drive twice as fast as if I had one car. Clearly, that is not true in terms of speed measured in miles per hour. But what about other kinds of "performance"? For what kinds of "performance" does having two cars improve performance? Under what circumstances are two cars actually twice as good as one?

Example Answer
### Question 8: Multiple Answer

Which of the following are key components of a stored program computer?

<table>
<thead>
<tr>
<th>Correct Answers</th>
<th>Percent Correct</th>
<th>Percent Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory.</td>
<td>93.056%</td>
<td>6.944%</td>
</tr>
<tr>
<td>IO devices.</td>
<td>63.889%</td>
<td>36.111%</td>
</tr>
<tr>
<td>The program counter.</td>
<td>83.333%</td>
<td>16.667%</td>
</tr>
<tr>
<td>A register file.</td>
<td>48.611%</td>
<td>51.389%</td>
</tr>
<tr>
<td>Reconfigurable hardware.</td>
<td>86.111%</td>
<td>13.889%</td>
</tr>
<tr>
<td>The ability to execute java programs.</td>
<td>94.444%</td>
<td>5.556%</td>
</tr>
<tr>
<td>A processor that executes a series of discrete instructions.</td>
<td>81.944%</td>
<td>18.056%</td>
</tr>
</tbody>
</table>

Average Score: 0.44 points
Question 9: Multiple Choice

What will the values of $t2$ and $t3$ be after executing this MIPS code?
ori $t0$, $zero$, 4
sw $t0$, 0($sp)
beq $t0$, $t0$, foo
lw $t1$, 0($sp)
ori $t2$, $zero$, 0

foo:
or $t2$, $t1$, $zero$
or $t3$, $t1$, $zero$

Correct
$t2 = 0$
$t3 = 0$

Percent Answered
9.589%

$\checkmark$

$t2 = 0$
$t3 = 4$

$t2 = 4$
$t3 = 0$
$t2 = 4$
$t3 = 4$

None of the above.

Unanswered
Goals for this Class

• Understand how CPUs run programs
  • How do we express the computation the CPU?
  • How does the CPU execute it?
  • How does the CPU support other system components (e.g., the OS)?
  • What techniques and technologies are involved and how do they work?

• Understand why CPU performance varies
  • How does CPU design impact performance?
  • What trade-offs are involved in designing a CPU?
  • How can we meaningfully measure and compare computer performance?

• Understand why program performance varies
  • How do program characteristics affect performance?
  • How can we improve a program's performance by considering the CPU running it?
  • How do other system components impact program performance?
Goals

- Start learning to read x86 assembly
- Understand the design trade-offs involved in crafting an ISA
- Understand RISC and CISC
  - Motivations
  - Origins
- Learn something about other current ISAs
  - Very long instruction word (VLIW)
  - Arm and Thumb
The Stack Frame

- A function’s “stack frame” holds
  - It’s local variables
  - Copies of callee-saved registers (if needs to used them)
  - Copies of caller-saved registers (when it makes function calls).
- The frame pointer ($fp) points to the base of the frame stack frame.
- The frame pointer in action.
  - Adjust the stack pointer to allocate the frame
  - Save the $fp into the frame (it’s callee-saved)
  - Copy from the $sp to the $fp
  - Use the $sp as needed for function calls.
  - Refer to local variables relative to $fp.
  - Clean up when you’re done.

Example

```
main:
  addiu $sp,$sp,-32
  sw $fp,24($sp)
  move $fp,$sp
  sw $0,8($fp)
  li $v0,1
  sw $v0,12($fp)
  li $v0,2
  sw $v0,16($fp)
  lw $3,12($fp)
  lw $v0,16($fp)
  addu $v0,$3,$v0
  sw $v0,8($fp)
  lw $v0,8($fp)
  move $sp,$fp
  lw $fp,24($sp)
  addiu $sp,$sp,32
  j $ra
```
The Stack Frame

main:
  addiu $sp,$sp,-32
  sw $fp,24($sp)
  move $fp,$sp
  sw $0,8($fp)
  li $v0,1
  sw $v0,12($fp)
  li $v0,2
  sw $v0,16($fp)
  lw $3,12($fp)
  lw $v0,16($fp)
  addu $v0,$3,$v0
  sw $v0,8($fp)
  lw $v0,8($fp)
  move $sp,$fp
  lw $fp,24($sp)
  addiu $sp,$sp,32
  j $ra

<table>
<thead>
<tr>
<th>...</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
</tr>
<tr>
<td>0x1018</td>
<td></td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
</tr>
<tr>
<td>0x100C</td>
<td></td>
</tr>
<tr>
<td>0x1008</td>
<td></td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
</tr>
</tbody>
</table>
The Stack Frame

main:
  addiu $sp,$sp,-32
  sw $fp,24($sp)
  move $fp,$sp
  sw $0,8($fp)
  li $v0,1
  sw $v0,12($fp)
  li $v0,2
  sw $v0,16($fp)
  lw $3,12($fp)
  lw $v0,16($fp)
  addu $v0,$3,$v0
  sw $v0,8($fp)
  lw $v0,8($fp)
  move $sp,$fp
  lw $fp,24($sp)
  addiu $sp,$sp,32
  j $ra

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x1020</td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
</tr>
<tr>
<td>0x1018</td>
<td></td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
</tr>
<tr>
<td>0x100C</td>
<td></td>
</tr>
<tr>
<td>0x1008</td>
<td></td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
</tr>
</tbody>
</table>
The Stack Frame

main:
PC->

addiu $sp,$sp,-32
sw $fp,24($sp)
move $fp,$sp
sw $0,8($fp)
li $v0,1
sw $v0,12($fp)
li $v0,2
sw $v0,16($fp)
lw $3,12($fp)
lw $v0,16($fp)
addu $v0,$3,$v0
sw $v0,8($fp)
lw $v0,8($fp)
move $sp,$fp
lw $fp,24($sp)
addiu $sp,$sp,32
j $ra

<table>
<thead>
<tr>
<th>sp-&gt;</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x1020</td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
</tr>
<tr>
<td>0x1018</td>
<td></td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
</tr>
<tr>
<td>0x100C</td>
<td></td>
</tr>
<tr>
<td>0x1008</td>
<td></td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
</tr>
</tbody>
</table>
The Stack Frame

main:

PC->

addiu $sp,$sp,-32
sw $fp,24($sp)
move $fp,$sp
sw $0,8($fp)
li $v0,1
sw $v0,12($fp)
li $v0,2
sw $v0,16($fp)
lw $3,12($fp)
lw $v0,16($fp)
addu $v0,$3,$v0
sw $v0,8($fp)
lw $v0,8($fp)
move $sp,$fp
lw $fp,24($sp)
addiu $sp,$sp,32
j $ra


<table>
<thead>
<tr>
<th>...</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
</tr>
<tr>
<td>0x1018</td>
<td></td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
</tr>
<tr>
<td>0x100C</td>
<td></td>
</tr>
<tr>
<td>0x1008</td>
<td></td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
</tr>
</tbody>
</table>
The Stack Frame

main:
  addiu $sp,$sp,-32
  lw $v0,2($fp)
  li $v0,1
  sw $v0,12($fp)
  li $v0,2
  lw $v0,16($fp)
  lw $v0,16($fp)
  addu $v0,$3,$v0
  lw $v0,8($fp)
  lw $v0,8($fp)
  move $sp,$fp
  lw $fp,24($sp)
  addiu $sp,$sp,32
  j $ra

PC->

<table>
<thead>
<tr>
<th>...</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
</tr>
<tr>
<td>0x1018</td>
<td></td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
</tr>
<tr>
<td>0x100C</td>
<td></td>
</tr>
<tr>
<td>0x1008</td>
<td></td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
</tr>
</tbody>
</table>
The Stack Frame

main:
  addiu $sp,$sp,-32
  sw $fp,24($sp)
  move $fp,$sp
  sw $0,8($fp)
  li $v0,1
  sw $v0,12($fp)
  li $v0,2
  sw $v0,16($fp)
  lw $3,12($fp)
  lw $v0,16($fp)
  addu $v0,$3,$v0
  sw $v0,8($fp)
  lw $v0,8($fp)
  move $sp,$fp
  lw $fp,24($sp)
  addiu $sp,$sp,32
  j $ra

PC->

<table>
<thead>
<tr>
<th>...</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
</tr>
<tr>
<td>0x1018</td>
<td></td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
</tr>
<tr>
<td>0x100C</td>
<td></td>
</tr>
<tr>
<td>0x1008</td>
<td></td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
</tr>
</tbody>
</table>

sp->

... value
0x1020
0x101C
0x1018 old fp
0x1014
0x1010
0x100C
0x1008
0x1004
0x1000
0x0FFC
The Stack Frame

main:

 punched  $sp,$sp,-32
sw       $fp,24($sp)
move     $fp,$sp
sw       $0,8($fp)
li       $v0,1
sw       $v0,12($fp)
li       $v0,2
sw       $v0,16($fp)
lw       $3,12($fp)
lw       $v0,16($fp)
addu     $v0,$3,$v0
sw       $v0,8($fp)
lw       $v0,8($fp)
move     $sp,$fp
lw       $fp,24($sp)
addiu    $sp,$sp,32
j         $ra

<table>
<thead>
<tr>
<th>...</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
</tr>
<tr>
<td>0x1018</td>
<td>old fp</td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
</tr>
<tr>
<td>0x100C</td>
<td></td>
</tr>
<tr>
<td>0x1008</td>
<td></td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
</tr>
</tbody>
</table>
The Stack Frame

main:

```
addiu $sp,$sp,-32
sw $fp,24($sp)
move $fp,$sp
sw $0,8($fp)
li $v0,1
sw $v0,12($fp)
li $v0,2
sw $v0,16($fp)
lw $3,12($fp)
lw $v0,16($fp)
addu $v0,$3,$v0
sw $v0,8($fp)
lw $v0,8($fp)
move $sp,$fp
lw $fp,24($sp)
addiu $sp,$sp,32
j $ra
```
The Stack Frame

```
main:
    addiu $sp,$sp,-32
    sw $fp,24($sp)
    move $fp,$sp
    sw $0,8($fp)
    li $v0,1
    sw $v0,12($fp)
    li $v0,2
    sw $v0,16($fp)
    lw $3,12($fp)
    lw $v0,16($fp)
    addu $v0,$3,$v0
    sw $v0,8($fp)
    lw $v0,8($fp)
    move $sp,$fp
    lw $fp,24($sp)
    addiu $sp,$sp,32
    j $ra
```
The Stack Frame

main:
  addiu $sp,$sp,-32
  sw $fp,24($sp)
  move $fp,$sp
  sw $0,8($fp)
  li $v0,1
  sw $v0,12($fp)
  li $v0,2
  sw $v0,16($fp)
  lw $3,12($fp)
  lw $v0,16($fp)
  addu $v0,$3,$v0
  sw $v0,8($fp)
  lw $v0,8($fp)
  move $sp,$fp
  lw $fp,24($sp)
  addiu $sp,$sp,32
  j $ra

PC-> fp-> sp->

<table>
<thead>
<tr>
<th>...</th>
<th>value</th>
<th>fp-relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
<td>+32</td>
</tr>
<tr>
<td>0x1018</td>
<td>old fp</td>
<td>+24</td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
<td>+20</td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
<td>+16</td>
</tr>
<tr>
<td>0x100C</td>
<td></td>
<td>+12</td>
</tr>
<tr>
<td>0x1008</td>
<td></td>
<td>+8</td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
<td>+4</td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
<td>+0</td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The stack frame
The Stack Frame

```
main:
    addiu $sp, $sp, -32
    sw $fp, 24($sp)
    move $fp, $sp
    sw $0, 8($fp)
    li $v0, 1
    sw $v0, 12($fp)
    li $v0, 2
    sw $v0, 16($fp)
    lw $3, 12($fp)
    lw $v0, 16($fp)
    addu $v0, $3, $v0
    sw $v0, 8($fp)
    lw $v0, 8($fp)
    move $sp, $fp
    lw $fp, 24($sp)
    addiu $sp, $sp, 32
    j $ra
```

The stack frame:

<table>
<thead>
<tr>
<th>...</th>
<th>value</th>
<th>fp-relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
<td>+32</td>
</tr>
<tr>
<td>0x1018</td>
<td>old fp</td>
<td>+24</td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
<td>+20</td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
<td>+16</td>
</tr>
<tr>
<td>0x100C</td>
<td></td>
<td>+12</td>
</tr>
<tr>
<td>0x1008</td>
<td>0</td>
<td>+8</td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
<td>+4</td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
<td>+0</td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Stack Frame

main:
  addiu $sp,$sp,-32
  sw $fp,24($sp)
  move $fp,$sp
  sw $0,8($fp)
  li $v0,1
  sw $v0,12($fp)
  li $v0,2
  sw $v0,16($fp)
  lw $3,12($fp)
  lw $v0,16($fp)
  addu $v0,$3,$v0
  sw $v0,8($fp)
  lw $v0,8($fp)
  move $sp,$fp
  lw $fp,24($sp)
  addiu $sp,$sp,32
  j $ra
The Stack Frame

main:
  addiu $sp,$sp,-32
  sw $fp,24($sp)
  move $fp,$sp
  sw $0,8($fp)
  li $v0,1
  sw $v0,12($fp)
  li $v0,2
  sw $v0,16($fp)
  lw $3,12($fp)
  lw $v0,16($fp)
  addu $v0,$3,$v0
  sw $v0,8($fp)
  lw $v0,8($fp)
  move $sp,$fp
  lw $fp,24($sp)
  addiu $sp,$sp,32
  j $ra

<table>
<thead>
<tr>
<th>...</th>
<th>value</th>
<th>fp-relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
<td>+32</td>
</tr>
<tr>
<td>0x1018</td>
<td>old fp</td>
<td>+24</td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
<td>+20</td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
<td>+16</td>
</tr>
<tr>
<td>0x100C</td>
<td>1</td>
<td>+12</td>
</tr>
<tr>
<td>0x1008</td>
<td>0</td>
<td>+8</td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
<td>+4</td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
<td>+0</td>
</tr>
</tbody>
</table>

The stack frame
The Stack Frame

**main:**

- `addiu $sp, $sp, -32`
- `sw $fp, 24($sp)`
- `move $fp, $sp`
- `sw $0, 8($fp)`
- `li $v0, 1`
- `sw $v0, 12($fp)`
- `li $v0, 2`
- `sw $v0, 16($fp)`
- `lw $3, 12($fp)`
- `lw $v0, 16($fp)`
- `addu $v0, $3, $v0`
- `sw $v0, 8($fp)`
- `lw $v0, 8($fp)`
- `move $sp, $fp`
- `lw $fp, 24($sp)`
- `addiu $sp, $sp, 32`
- `j $ra`

**Table:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>fp-relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
<td>+32</td>
</tr>
<tr>
<td>0x1018</td>
<td>old fp</td>
<td>+24</td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
<td>+20</td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
<td>+16</td>
</tr>
<tr>
<td>0x100C</td>
<td>1</td>
<td>+12</td>
</tr>
<tr>
<td>0x1008</td>
<td>0</td>
<td>+8</td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
<td>+4</td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
<td>+0</td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**The stack frame**

PC->

fp-> sp->
The Stack Frame

main:
    addiu $sp,$sp,-32
    sw $fp,24($sp)
    move $fp,$sp
    sw $0,8($fp)
    li $v0,1
    sw $v0,12($fp)
    li $v0,2
    sw $v0,16($fp)
    lw $3,12($fp)
    lw $v0,16($fp)
    addu $v0,$3,$v0
    sw $v0,8($fp)
    lw $v0,8($fp)
    move $sp,$fp
    lw $fp,24($sp)
    addiu $sp,$sp,32
    j $ra

PC->

<table>
<thead>
<tr>
<th>...</th>
<th>value</th>
<th>fp-relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
<td>+32</td>
</tr>
<tr>
<td>0x1018</td>
<td>old fp</td>
<td>+24</td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
<td>+20</td>
</tr>
<tr>
<td>0x1010</td>
<td>2</td>
<td>+16</td>
</tr>
<tr>
<td>0x100C</td>
<td>1</td>
<td>+12</td>
</tr>
<tr>
<td>0x1008</td>
<td>0</td>
<td>+8</td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
<td>+4</td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
<td>+0</td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The stack frame
The Stack Frame

main:
addiu $sp,$sp,-32  
sw $fp,24($sp) 
move $fp,$sp 
sw $0,8($fp)  
li $v0,1 
sw $v0,12($fp)  
li $v0,2 
sw $v0,16($fp)  
lw $3,12($fp)  
lw $v0,16($fp)  
addu $v0,$3,$v0 
sw $v0,8($fp)  
lw $v0,8($fp)  
move $sp,$fp  
lw $fp,24($sp)  
addiu $sp,$sp,32  
j $ra

The stack frame

<table>
<thead>
<tr>
<th>...</th>
<th>value</th>
<th>fp-relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
<td>+32</td>
</tr>
<tr>
<td>0x1018</td>
<td>old fp</td>
<td>+24</td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
<td>+20</td>
</tr>
<tr>
<td>0x1010</td>
<td>2</td>
<td>+16</td>
</tr>
<tr>
<td>0x100C</td>
<td>1</td>
<td>+12</td>
</tr>
<tr>
<td>0x1008</td>
<td>0</td>
<td>+8</td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
<td>+4</td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
<td>+0</td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Stack Frame

main:
addiu $sp,$sp,-32
sw $fp,24($sp)
move $fp,$sp
sw $0,8($fp)
li $v0,1
sw $v0,12($fp)
li $v0,2
sw $v0,16($fp)
lw $3,12($fp)
lw $v0,16($fp)
addu $v0,$3,$v0
sw $v0,8($fp)
lw $v0,8($fp)
move $sp,$fp
lw $fp,24($sp)
addiu $sp,$sp,32
j $ra

PC-> fp-> sp->

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>fp-relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
<td>+32</td>
</tr>
<tr>
<td>0x1018</td>
<td>old fp</td>
<td>+24</td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
<td>+20</td>
</tr>
<tr>
<td>0x1010</td>
<td>2</td>
<td>+16</td>
</tr>
<tr>
<td>0x100C</td>
<td>1</td>
<td>+12</td>
</tr>
<tr>
<td>0x1008</td>
<td>3</td>
<td>+8</td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
<td>+4</td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
<td>+0</td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Stack Frame

main:
    addiu $sp,$sp,-32
    sw $fp,24($sp)
    move $fp,$sp
    sw $0,8($fp)
    li $v0,1
    sw $v0,12($fp)
    li $v0,2
    sw $v0,16($fp)
    lw $3,12($fp)
    lw $v0,16($fp)
    addu $v0,$3,$v0
    sw $v0,8($fp)
    lw $v0,8($fp)
    move $sp,$fp
    lw $fp,24($sp)
    addiu $sp,$sp,32
    j $ra

<table>
<thead>
<tr>
<th>PC-&gt;</th>
<th>fp-&gt;</th>
<th>sp-&gt;</th>
</tr>
</thead>
</table>

The stack frame

<table>
<thead>
<tr>
<th>...</th>
<th>value</th>
<th>fp-relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
<td>+32</td>
</tr>
<tr>
<td>0x1018</td>
<td>old fp</td>
<td>+24</td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
<td>+20</td>
</tr>
<tr>
<td>0x1010</td>
<td>2</td>
<td>+16</td>
</tr>
<tr>
<td>0x100C</td>
<td>1</td>
<td>+12</td>
</tr>
<tr>
<td>0x1008</td>
<td>3</td>
<td>+8</td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
<td>+4</td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
<td>+0</td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Stack Frame

main:

```
addiu $sp,$sp,-32
sw $fp,24($sp)
move $fp,$sp
sw $0,8($fp)
li $v0,1
sw $v0,12($fp)
li $v0,2
sw $v0,16($fp)
lw $3,12($fp)
lw $v0,16($fp)
addu $v0,$3,$v0
sw $v0,8($fp)
lw $v0,8($fp)
move $sp,$fp
lw $fp,24($sp)
addiu $sp,$sp,32
j $ra
```

The stack frame:

<table>
<thead>
<tr>
<th>...</th>
<th>value</th>
<th>fp-relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
<td>+32</td>
</tr>
<tr>
<td>0x1018</td>
<td>old fp</td>
<td>+24</td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
<td>+20</td>
</tr>
<tr>
<td>0x1010</td>
<td>2</td>
<td>+16</td>
</tr>
<tr>
<td>0x100C</td>
<td>1</td>
<td>+12</td>
</tr>
<tr>
<td>0x1008</td>
<td>3</td>
<td>+8</td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
<td>+4</td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
<td>+0</td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Stack Frame

**main:**
- `addiu $sp,$sp,-32`
- `sw $fp,24($sp)`
- `move $fp,$sp`
- `sw $0,8($fp)`
- `li $v0,1`
- `sw $v0,12($fp)`
- `li $v0,2`
- `sw $v0,16($fp)`
- `lw $3,12($fp)`
- `lw $v0,16($fp)`
- `addu $v0,$3,$v0`
- `sw $v0,8($fp)`
- `lw $v0,8($fp)`
- `move $sp,$fp`
- `lw $fp,24($sp)`
- `addiu $sp,$sp,32`
- `j $ra`

---

### The stack frame

<table>
<thead>
<tr>
<th>...</th>
<th>value</th>
<th>fp-relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x101C</td>
<td></td>
<td>+32</td>
</tr>
<tr>
<td>0x1018</td>
<td>old fp</td>
<td>+24</td>
</tr>
<tr>
<td>0x1014</td>
<td></td>
<td>+20</td>
</tr>
<tr>
<td>0x1010</td>
<td>2</td>
<td>+16</td>
</tr>
<tr>
<td>0x100C</td>
<td>1</td>
<td>+12</td>
</tr>
<tr>
<td>0x1008</td>
<td>3</td>
<td>+8</td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
<td>+4</td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
<td>+0</td>
</tr>
<tr>
<td>0x0FFC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Stack Frame

main:
  addiu $sp,$sp,-32
  sw $fp,24($sp)
  move $fp,$sp
  sw $0,8($fp)
  li $v0,1
  sw $v0,12($fp)
  li $v0,2
  sw $v0,16($fp)
  lw $3,12($fp)
  lw $v0,16($fp)
  addu $v0,$3,$v0
  sw $v0,8($fp)
  lw $v0,8($fp)
  move $sp,$fp
  lw $fp,24($sp)
  addiu $sp,$sp,32
  j $ra

   ...     |  value  | fp-relative
   --|--------|----------
  0x1020  |        |          
  0x101C  |        | +32      
  0x1018  | old fp | +24      
  0x1014  |        | +20      
  0x1010  | 2      | +16      
  0x100C  | 1      | +12      
  0x1008  | 3      | +8       
  0x1004  |        | +4       
  0x1000  |        | +0       
  0x0FFC  |        |          

The stack frame
Dead Demo

http://cseweb.ucsd.edu/classes/wi12/cse141-a/asm_examples/
x86 Assembly
x86 ISA Caveats

• x86 is a poorly-designed ISA
  • It breaks almost every rule of good ISA design.
  • There is nothing “regular” or predictable about its syntax.
  • We don’t have time to learn how to write x86 with any kind of thoroughness.

• It is the most widely used ISA in the world today.
  • It is the ISA you are most likely to see in the “real world”
  • So it’s useful to study.

• Intel and AMD have managed to engineer (at considerable cost) their CPUs so that this ugliness has relatively little impact on their processors’ performance (more on this later)
Some Differences Between MIPS and x86

• x86 instructions can operate on memory or registers or both
• x86 is a “two address” ISA
  • Both arguments are sources.
  • One is also the destination
• x86 has (lots of) special-purpose registers
• x86 has variable-length instructions
  • Between 1 and 17 bytes
There are two syntaxes for x86 assembly
We will use the “gnu assembler (gas) syntax”, aka “AT&T syntax”. This different than “Intel Syntax”
The most confusing difference: argument order
- AT&T/gas
  - <instruction> <src> <dst>
- Intel
  - <instruction> <dst> <src>
Also, different instruction names
There are some other differences too (see http://en.wikipedia.org/wiki/X86_assembly_language#Syntax)
If you go looking for help online, make sure it uses the AT&T syntax (or at least be aware, if it doesn’t)!
## Registers

<table>
<thead>
<tr>
<th>8-bit</th>
<th>16-bit</th>
<th>32-bit</th>
<th>64-bit</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%AL</td>
<td>%AX</td>
<td>%EAX</td>
<td>%RAX</td>
<td>The accumulator register</td>
<td></td>
</tr>
<tr>
<td>%BL</td>
<td>%BX</td>
<td>%EBX</td>
<td>%RBX</td>
<td>The base register</td>
<td></td>
</tr>
<tr>
<td>%CL</td>
<td>%CX</td>
<td>%ECX</td>
<td>%RCX</td>
<td>The counter</td>
<td></td>
</tr>
<tr>
<td>%DL</td>
<td>%DX</td>
<td>%EDX</td>
<td>%RDX</td>
<td>The data register</td>
<td></td>
</tr>
<tr>
<td>%SPL</td>
<td>%SP</td>
<td>%ESP</td>
<td>%RSP</td>
<td>Stack pointer</td>
<td></td>
</tr>
<tr>
<td>%SBP</td>
<td>%BP</td>
<td>%EBP</td>
<td>%RBP</td>
<td>Points to the base of the stack frame</td>
<td></td>
</tr>
<tr>
<td>%RnB</td>
<td>%RnW</td>
<td>%RnD</td>
<td>%Rn</td>
<td>(n = 8...15) General purpose registers</td>
<td></td>
</tr>
<tr>
<td>%SIL</td>
<td>%SI</td>
<td>%ESI</td>
<td>%RSI</td>
<td>Source index for string operations</td>
<td></td>
</tr>
<tr>
<td>%DIL</td>
<td>%DI</td>
<td>%EDI</td>
<td>%RDI</td>
<td>Destination index for string operations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>%IP</td>
<td>%EIP</td>
<td>%RIP</td>
<td>Instruction Pointer</td>
<td></td>
</tr>
<tr>
<td>%FLAGS</td>
<td></td>
<td></td>
<td></td>
<td>Condition codes</td>
<td></td>
</tr>
</tbody>
</table>

These can be used more or less interchangeably, like the registers in MIPS.

Different names (e.g. %AX vs. %EAX vs. %RAX) refer to different parts of the same register.
Instruction Suffixes

<table>
<thead>
<tr>
<th>Instruction Suffixes</th>
<th>Description</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>byte</td>
<td>8</td>
</tr>
<tr>
<td>s</td>
<td>short</td>
<td>16</td>
</tr>
<tr>
<td>w</td>
<td>word</td>
<td>16</td>
</tr>
<tr>
<td>l</td>
<td>long</td>
<td>32</td>
</tr>
<tr>
<td>q</td>
<td>quad</td>
<td>64</td>
</tr>
</tbody>
</table>

Example

```
addb $4, %al
addw $4, %ax
addl $4, %eax
addq %rcx, %rax
```
## Arguments/Addressing Modes

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>%&lt;reg&gt;</td>
<td>R[&lt;reg&gt;]</td>
<td>%RAX</td>
</tr>
<tr>
<td>Immediate</td>
<td>$nnn</td>
<td>constant</td>
<td>$42</td>
</tr>
<tr>
<td>Label</td>
<td>$label</td>
<td>label</td>
<td>$foobar</td>
</tr>
<tr>
<td>Base-Offset</td>
<td>(%r1, %r2)</td>
<td>Mem[R[&lt;reg&gt;] + %R[&lt;reg&gt;]]</td>
<td>(%RAX,%AL)</td>
</tr>
<tr>
<td>Scaled Offset</td>
<td>(%r1, %r2, 2^n)</td>
<td>Mem[R[&lt;reg&gt;] + %R[&lt;reg&gt;] * 2^n]</td>
<td>(%RAX,%AL, 4)</td>
</tr>
<tr>
<td>Scaled Offset Displacement</td>
<td>k(%r1, %r2, 2^n)</td>
<td>Mem[R[&lt;reg&gt;] + %R[&lt;reg&gt;] * 2^n + k]</td>
<td>-4(%RAX,%AL, 2)</td>
</tr>
</tbody>
</table>
x86 does not have loads and stores. It has `mov`.

<table>
<thead>
<tr>
<th>x86 Instruction</th>
<th>RTL</th>
<th>MIPS Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb $0x05, %al</td>
<td>R[al] = 0x05</td>
<td>ori $t0, $t0, 5</td>
</tr>
<tr>
<td>movl -4(%ebp), %eax</td>
<td>mem[R[ebp] -4] = R[eax]</td>
<td>lw $t0, -4($t1)</td>
</tr>
<tr>
<td>movl %eax, -4(%ebp)</td>
<td>R[eax] = mem[R[ebp] -4]</td>
<td>sw $t0, -4($t1)</td>
</tr>
<tr>
<td>movl $LC0, (%esp)</td>
<td>mem[R[esp]] = $LC0</td>
<td>la $at, LC0 sw $at, 0($t0)</td>
</tr>
<tr>
<td>movl %R0, -4(%R1,%R2,4)</td>
<td>mem[R[%R1] + R[%R2] * (2^n + k)] = %R0</td>
<td>slr $at, $t2, 2 add $at, $at, $t1 sw $t0, k($at)</td>
</tr>
<tr>
<td>movl %R0, %R1</td>
<td>R[%R1] = R[%R0]</td>
<td>ori $t0, $t0, $zero</td>
</tr>
</tbody>
</table>
# Arithmetic

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>subl $0x05, %eax</td>
<td>R[eax] = R[eax] - 0x05</td>
</tr>
</tbody>
</table>
## Stack Management

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>x86 Equivalent</th>
<th>MIPS equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl %eax</td>
<td>Push %eax onto the stack</td>
<td>subl $4, %esp; movl %eax, (%esp)</td>
<td>subi $sp, $sp, 4 sw $t0, ($sp)</td>
</tr>
<tr>
<td>popl %eax</td>
<td>Pop %eax off the stack</td>
<td>movl (%esp), %eax addl $4, %esp</td>
<td>lw $t0, ($sp) addi $sp, $sp, 4</td>
</tr>
<tr>
<td>enter n</td>
<td>Save stack pointer, allocate stack frame with n bytes for locals</td>
<td>push %BP mov %SP, %BP sub $n, %SP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>movl %ebp, %esp pop %ebp</td>
<td></td>
</tr>
</tbody>
</table>

None of these are pseudo instructions. They are real instructions, just very complex.
The Stack Frame

- A function’s “stack frame” holds
  - It’s local variables
  - Copies of callee-saved registers (if needs to used them)
  - Copies of caller-saved registers (when it makes function calls).
- The base pointer (%ebp) points to the base of the frame stack frame.
- The base pointer in action
  - Save the old stack pointer.
  - Align the stack pointer
  - Save the old %ebp
  - Copy from the %esp to the %ebp
  - Allocate the frame by decrementing %esp
  - Refer to local variables relative to %ebp
  - Clean up when you’re done.

Example

main:

```
leal 4(%esp), %ecx
andl $-16, %esp
pushl -4(%ecx)
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl $0, -16(%ebp)
movl $1, -12(%ebp)
movl $2, -8(%ebp)
movl -8(%ebp), %eax
addl -12(%ebp), %eax
movl %eax, -16(%ebp)
movl -16(%ebp), %eax
addl $16, %esp
popl %ebp
leal -4(%ecx), %esp
ret
```
Branches

- x86 uses condition codes for branches
  - Condition codes are special-purpose bits that make up the flags register
  - Arithmetic ops set the flags register
  - carry, parity, zero, sign, overflow

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpl %r1 %r2</td>
<td>Set flags register for %r1 - %r2</td>
</tr>
<tr>
<td>jmp &lt;location&gt;</td>
<td>Jump to &lt;location&gt;</td>
</tr>
<tr>
<td>je &lt;location&gt;</td>
<td>Jump to &lt;location&gt; if the equal flag is set</td>
</tr>
<tr>
<td>jg, jge, jl, jle, jnz, ...</td>
<td>jump if {&gt;, &gt;=, &lt;, &lt;=, != 0,}</td>
</tr>
</tbody>
</table>
# Function Calls

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>call &lt;label&gt;</td>
<td>Push the return address onto the stack.</td>
<td>Homework?</td>
</tr>
<tr>
<td></td>
<td>Jump to the function.</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>Pop the return address off the stack and</td>
<td>lw $at, 0($sp)</td>
</tr>
<tr>
<td></td>
<td>jump to it.</td>
<td>addi $sp, $sp, 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>jr $at</td>
</tr>
</tbody>
</table>

- Return address goes on the stack (rather than a register as in MIPS)
- Arguments are passed on the stack (with push)
- Return value in `%eax/%rax`

**Example**

```c
int foo(int x, int y);
...
d = foo(a, b);
pushq %R9
pushq %R8
call foo
movq %eax, d
```
These slides don’t cover everything you’ll need for the homeworks on x86 assembly
- There’s too many ugly details to cover in class.
- But you may still encounter this code in real life (or on the homeworks).

You’ll need to do some looking of your own to find the missing bits
- The text book.

Make sure you know if the resources you find are AT&T or Intel syntax!
- If there aren’t any “%”, it’s probably Intel, and the dst comes first, rather than last.
MIPS vs. x86: Arithmetic

http://cseweb.ucsd.edu/classes/wi12/cse141-a/asm_examples/1.html

http://cseweb.ucsd.edu/classes/wi12/cse141-a/asm_examples/5.html
MIPS vs. x86: Branches

http://cseweb.ucsd.edu/classes/wi12/cse141-a/assembling/asm_examples/7.html
MIPS vs. x86: Caller

http://cseweb.ucsd.edu/classes/wi12/cse141-a/asm_examples/caller.html
MIPS vs. x86: Callee

http://cseweb.ucsd.edu/classes/wi12/cse141-a/asm_examples/callee.html
MIPS vs. x86: Structs

http://cseweb.ucsd.edu/classes/wi12/cse141-a/asm_examples/struct.html
Other ISAs
Designing an ISA to Improve Performance

• The PE tells us that we can improve performance by reducing CPI. Can we get CPI to be less than 1?
  • Yes, but it means we must execute more than one instruction per cycle.
  • That means parallelism.

• How can we modify the ISA to support the execution of multiple instructions each cycle?

• Later, we’ll look at modifying the processor implementation to do the same thing without changing the ISA.
Very Long Instruction Word (VLIW)

- Put two (or more) instructions in one!

- Each sub-instruction is just like a normal instruction.
- The instructions execute at the same time.
- The processor can treat them as a single unit.
- Typical VLIW widths are 2-4 instructions, but some machine have been much higher
Very Long Instruction Word (VLIW)

- Put two (or more) instructions in one!

<table>
<thead>
<tr>
<th>32 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
</tr>
<tr>
<td>rs</td>
</tr>
<tr>
<td>rt</td>
</tr>
<tr>
<td>rd</td>
</tr>
<tr>
<td>shamt</td>
</tr>
<tr>
<td>funct</td>
</tr>
</tbody>
</table>

- Each sub-instruction is just like a normal instruction.
- The instructions execute *at the same time*.
- The processor can treat them as a single unit.
- Typical VLIW widths are 2-4 instructions, but some machines have been much higher.
Very Long Instruction Word (VLIW)

- Put two (or more) instructions in one!

Each sub-instruction is just like a normal instruction.

The instructions execute at the same time.

The processor can treat them as a single unit.

Typical VLIW widths are 2-4 instructions, but some machine have been much higher.
Very Long Instruction Word (VLIW)

• Put two (or more) instructions in one!

- Each sub-instruction is just like a normal instruction.
- The instructions execute *at the same time*.
- The processor can treat them as a single unit.
- Typical VLIW widths are 2-4 instructions, but some machine have been much higher.
Very Long Instruction Word (VLIW)

- Put two (or more) instructions in one!

Each sub-instruction is just like a normal instruction.
- The instructions execute at the same time.
- The processor can treat them as a single unit.
- Typical VLIW widths are 2-4 instructions, but some machines have been much higher.
VLIW Example

- VLIW-MIPS
  - Two MIPS instruction/VLIW instruction word
  - Not a real VLIW ISA.

MIPS Code
ori $s2, $zero, 6
ori $s3, $zero, 4
add $s2, $s2, $s3
sub $s4, $s2, $s3

Results:
$s2 = 10$
$s4 = 6$

Since the add and sub execute sequentially, the sub sees the new value for $s2$

VLIW-MIPS Code
<ori $s2, $zero,6; ori $s3, $zero, 4>
<add $s2, $s2, $s3; sub $s4, $s2, $s3>

Results:
$s2 = 10$
$s4 = 2$

Since the add and sub execute at the same time they both see the original value of $s2$
VLIW Challenges

- VLIW has been around for a long time, but it’s not seen mainstream success.
- The main challenging is finding instructions to fill the VLIW slots.
- This is tortuous by hand, and difficult for the compiler.

VLIW-MIPS Code

```mips
<ori $s2, $zero,6;  ori $s3, $zero, 4>
<add $s2, $s2, $s3; nop
<sub $s4, $s2, $s3; nop
```

Results:
$s2 = 10$
$s4 = 6$

Now, the add and sub execute sequentially, but we’ve wasted space and resources executing nops.
VLIW’s History

- VLIW has been around for a long time
  - It’s the simplest way to get CPI < 1.
  - The ISA specifies the parallelism, the hardware can be very simple
  - When hardware was expensive, this seemed like a good idea.
- However, the compiler problem (previous slide) is extremely hard.
  - There end up being lots of noops in the long instruction words.
  - Especially for “branchy” code (word processors, compilers, games, etc.)
- As a result, they have either
  - 1. met with limited commercial success as general purpose machines (many companies) or,
  - 2. Become very complicated in new and interesting ways (for instance, by providing special registers and instructions to eliminate branches), or
  - 3. Both 1 and 2 -- See the Itanium from intel.
VLIW’s Success Stories

- VLIW’s main success is in digital signal processing
  - DSP applications mostly comprise very regular loops
    - Constant loop bounds,
    - Simple data access patterns
    - Non-data-dependent computation
  - Since these kinds of loops make up almost all (i.e., $x$ is almost 1.0) of the applications, Amdahl’s Laws says writing the code by hand is worthwhile.
- These applications are cost and power sensitive
  - VLIW processors are simple
    - Simple means small, cheap, and efficient.
- I would not be surprised if there’s a VLIW processor in your cell phone.
The ARM ISA

- The ARM ISA is in most of today’s cool mobile gadgets
- It got started at about the same time as MIPS
  - ARM Holdings, Inc. owns the ISA and licenses it to other companies.
  - It does not actually build chips.
- There are ARM chips available from many vendors
  - The vendors compete or other features (e.g., integrated graphics co-processors)
  - Drives down cost.
- There’s an ARM version of your text book.
MIPS vs. ARM

- MIPS and ARM are both modern, relatively clean ISAs
- ARM has
  - Fixed-length instruction words (mostly. More in moment)
  - General-purpose registers (although only 16 of them)
  - A similar set of instructions.
- But there are some differences...
MIPS vs. ARM: Addressing Modes

- MIPS has 3 “addressing modes”
  - Register -- $s1
  - Displacement -- 4($s1)
  - Immediate -- 4

- ARM has several more

<table>
<thead>
<tr>
<th>ARM Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR r0,[r1,#8]</td>
<td>R[r0] = Mem[R[r1] + 8]</td>
</tr>
<tr>
<td>LDR r0,[r1,#8]!</td>
<td>R[r0] = Mem[R[r1] + 8]; R[r1] = R[r1] + 8</td>
</tr>
<tr>
<td>LDR r0,[r1],#8</td>
<td>R[r0] = Mem[R[r1]]; R[r1] = R[r1] + 8</td>
</tr>
</tbody>
</table>
MIPS vs. ARM: Shifts

- ARM likes to perform shift operations
- The second src operand of most instructions can be shifted before use
- MIPS is less shift-happy.

<table>
<thead>
<tr>
<th>ARM Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add r1,r2,r3, LSL #4</td>
<td>$R[r1] = R[r2] + (R[r3] &lt;&lt; 4)$</td>
</tr>
<tr>
<td>Add r1,r2,r3, LSL r4</td>
<td>$R[r1] = R[r2] + (R[r3] &lt;&lt; R[r4])$</td>
</tr>
</tbody>
</table>
MIPS vs. ARM: Branches

- ARM uses condition codes and predication for branches
  - Condition codes: negative, zero, carry, overflow
  - Instruction set them
- Instruction can be made conditional on one of the condition codes
  - The corresponding condition code is set, the instruction will execute.
  - Otherwise, the instruction will be a nop.
  - An instruction suffix specifies the condition code
- This eliminates many branches.
  - We’ll see later on in this class that branches can slow down execution.

C Code

```c
if (x == y)
    p = q + r
```

MIPS Assembly

```assembly
bne $s0, $s1, foo
add $s2, $s3, $s4
foo:
```

ARM Assembly

```assembly
CMP r0, r1
ADDEQ r2, r3, r4
```
ISA Alternatives

• 2-address code
  • add r1, r2 means R[r1] = R[r1] + R[r2]
  • + few operands, so more bits for each.
  • - lots of extra copy instructions

• 1-address -- Accumulator architectures
  • An “accumulator” is a source and destination for all operations
  • add r1 means acc = acc + R[r1]
  • setacc r1 mean acc = R[r1]
  • getacc r1 mean R[r1] = acc

• “0-address” code -- Stack-based architectures
Stack-based ISA

- A stack holds arguments
- Some instruction manipulate the stack
  - push -- add something to the stack
  - pop -- remove the top item.
  - swap -- swaps the top two items
- Most instructions operate on the contents of the stack
  - Zero-operand instructions
    - add --> t1 = pop; t2 = pop; push t1 + t2;
- Elegant in theory
- Clumsy in hardware.
  - How big is the stack?
- Java and Python “byte code” are stack-based ISAs
  - Infinite stack, but it runs in a VM
  - More on this later.
Stack Example: \( A = X \times Y - B \times C \)

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - \(+, -, \ast, \ldots\) -- Replace top two values with the result
  - Store -- Store the top of the stack

```
Push 12 (BP)
Push 8 (BP)
Mult
Push 0 (BP)
Push 4 (BP)
Mult
Sub
Store 16 (BP)
Pop
```

[Diagram showing stack operations and memory layout]
Stack Example: $A = X \times Y - B \times C$

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - $+, -, \times, \ldots$ -- Replace top two values with the result
  - Store -- Store the top of the stack
Stack Example: \( A = X \times Y - B \times C \)

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - \(+, -, \times, \ldots\) -- Replace top two values with the result
  - Store -- Store the top of the stack
Stack Example: $A = X \times Y - B \times C$

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - $+, -, \times, \ldots$ -- Replace top two values with the result
  - Store -- Store the top of the stack

--

```
Push 12(BP)
Push 8(BP)
Mult
Push 0(BP)
Push 4(BP)
Mult
Sub
Store 16(BP)
Pop
```
Stack Example: $A = X \times Y - B \times C$

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - $+, -, \times, \ldots$ -- Replace top two values with the result
  - Store -- Store the top of the stack

```
Push 12(BP)
Push 8(BP)
Mult
Push 0(BP)
Push 4(BP)
Mult
Sub
Store 16(BP)
Pop
```
Stack Example: $A = X \times Y - B \times C$

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - $+,-,\times,...$ -- Replace top two values with the result
  - Store -- Store the top of the stack

```
Push 12(BP)
PUSH 8(BP)
Mult
Push 0(BP)
PUSH 4(BP)
Mult
Sub
Store 16(BP)
Pop
```
Stack Example: \( A = X \times Y - B \times C \)

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - \(+, -, \times, \ldots\) -- Replace top two values with the result
  - Store -- Store the top of the stack

```
Push 12(BP)
Push 8(BP)
Mult
Push 0(BP)
Push 4(BP)
Mult
Sub
Store 16(BP)
Pop
```

```
X*Y
B*C

Memory

BP
+4
+8
X
+12
B
+16
C

Base ptr (BP) 0x1000
```
Stack Example: \( A = X \times Y - B \times C \)

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result
  - Store -- Store the top of the stack

```
Push 12(BP)
Push 8(BP)
Mult
Push 0(BP)
Push 4(BP)
Mult
Sub
Store 16(BP)
Pop
```

```
Memory

| 0x1000 | Base ptr (BP) |
|----------------|
| X            |
| Y            |
| B            |
| C            |
| A            |
+4
+8
+12
+16

X*Y-B*C
```
compute $A = X \times Y - B \times C$

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - $+, -, \times, \ldots$ -- Replace top two values with the result
  - Store -- Store the top of the stack

```
Push 12(BP)
Push 8(BP)
Mult
Push 0(BP)
Push 4(BP)
Mult
Sub
Store 16(BP)
Pop
```
RISC vs CISC
In the Beginning...

- 1964 -- The first ISA appears on the IBM System 360
- In the “good” old days
  - Initially, the focus was on usability by humans.
  - Lots of “user-friendly” instructions (remember the x86 addressing modes).
  - Memory was expensive, so code-density mattered.
  - Many processors were microcoded -- each instruction actually triggered the execution of a builtin function in the CPU. Simple hardware to execute complex instructions (but CPIs are very, very high)
  - Microcoding saved hardware (which was expensive)
    - You only needed one adder.
    - How many adders are in our simple MIPS pipeline?
- ...SO...
  - Many, many different instructions, lots of bells and whistles
  - Variable-length instruction encoding to save space.
- ... their success had some downsides...
  - ISAs evolved organically.
  - They got messier, and more complex.
Things Changed

• In the modern era
  • Compilers write code, not humans.
  • Memory is cheap. Code density is unimportant.
  • Hardware is cheap. E.g. Adders are essentially free.
  • Low CPI should be possible, but only for simple instructions
  • We learned a lot about how to design ISAs, how to let them evolve gracefully, etc.

• So, architects started with with a clean slate...
Reduced Instruction Set Computing (RISC)

- Simple, regular ISAs, mean simple CPUs, and simple CPUs can go fast.
  - Fast clocks.
  - Low CPI.
  - Simple ISAs will also mean more instruction (increasing IC), but the benefits should outweigh this.

- Compiler-friendly, not user-friendly.
  - Simple, regular ISAs, will be easy for compilers to use
  - A few, simple, flexible, fast operations that compiler can combine easily.
  - Separate memory access and data manipulation
    - Instructions access memory or manipulate register values. Not both.
    - “Load-store architectures” (like MIPS)
  - No (or at least not many) special cases!
RISC: MIPS

- MIPS is the prototypical RISC ISA.
  - 3 instruction formats. Fixed length.
  - Very simple instructions.
  - Separate memory access and arithmetic instructions (This is called a “load-store architecture”)
  - All registers are general-purpose.
  - Originally, very few instructions

- MIPS targeted maximum performance
  - Fast clocks!
  - Memory was cheap, so code density was not an issue.
  - The simpler, the better, because simple is fast.

- In 141L you’ll see the impact of its simplicity in hardware
MIPS is RISC!

- 3 instruction formats: I, R, and J.
  - R-type: Register-register Arithmetic
  - I-type: immediate arithmetic; loads/stores
  - J-type: Non-conditional, non-relative branches
  - opcodes are always in the same place
  - rs and rt are always in the same place
  - The immediate is always in the same place

- Similar amounts of work per instruction
  - 1 read from instruction memory
  - ≤ 1 arithmetic operations
  - ≤ 2 register reads
  - ≤ 1 register write
  - ≤ 1 data store/load

- Fixed instruction length
MIPS is RISC!

- 3 instruction formats: I, R, and J.
  - R-type: Register-register Arithmetic
  - I-type: immediate arithmetic; loads/stores
  - J-type: Non-conditional, non-relative branches
  - opcodes are always in the same place
  - rs and rt are always in the same place
  - The immediate is always in the same place

- Similar amounts of work per instruction
  - 1 read from instruction memory
  - <= 1 arithmetic operations
  - <= 2 register reads
  - <= 1 register write
  - <= 1 data store/load

- Fixed instruction length
CISC: x86

- x86 is the prime example of CISC (there were many others long ago)
  - Many, many instruction formats. Variable length.
  - Many complex rules about which register can be used when, and which addressing modes are valid where.
  - Very complex instructions
  - Combined memory/arithmetic.
  - Special-purpose registers.
  - Many, many instructions.
- Implementing x86 correctly is almost intractable
Mostly RISC: ARM

• ARM is somewhere in between
  • Four instruction formats. Fixed length.
  • General purpose registers (except the condition codes)
  • Moderately complex instructions, but they are still “regular” -- all instructions look more or less the same.

• ARM targeted embedded systems
  • Code density is important
  • Performance (and clock speed) is less critical
  • Both of these argue for more complex instructions.
  • But they can still be regular, easy to decode, and crafted to minimize hardware complexity

• Implementing an ARM processor is also tractable for 141L, but it would be harder than MIPS
RISCing the CISC

• Everyone believes that RISC ISAs are better for building fast processors.
• So, how do Intel and AMD build fast x86 processors?
  • Despite using a CISC ISA, these processors are actually RISC processors inside
  • Internally, they convert x86 instructions into MIPS-like micro-ops (uops), and feed them to a RISC-style processor

<table>
<thead>
<tr>
<th>x86 Code</th>
<th>uops</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb $0x05, %al</td>
<td></td>
</tr>
<tr>
<td>movl -4(%ebp), %eax</td>
<td></td>
</tr>
<tr>
<td>movl %eax, -4(%ebp)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, -4(%R1,%R2,4)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, %R1</td>
<td></td>
</tr>
</tbody>
</table>
RISCing the CISC

• Everyone believes that RISC ISAs are better for building fast processors.
• So, how do Intel and AMD build fast x86 processors?
  • Despite using a CISC ISA, these processors are actually RISC processors inside
  • Internally, they convert x86 instructions into MIPS-like micro-ops (uops), and feed them to a RISC-style processor

<table>
<thead>
<tr>
<th>PC-</th>
<th>x86 Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb $0x05, %al</td>
<td></td>
</tr>
<tr>
<td>movl -4(%ebp), %eax</td>
<td></td>
</tr>
<tr>
<td>movl %eax, -4(%ebp)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, -4(%R1,%R2,4)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, %R1</td>
<td></td>
</tr>
</tbody>
</table>
RISCing the CISC

- Everyone believes that RISC ISAs are better for building fast processors.
- So, how do Intel and AMD build fast x86 processors?
  - Despite using a CISC ISA, these processors are actually RISC processors inside
  - Internally, they convert x86 instructions into MIPS-like micro-ops (uops), and feed them to a RISC-style processor

<table>
<thead>
<tr>
<th>x86 Code</th>
<th>uops</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb $0x05, %al</td>
<td>ori $t0, $t0, 5</td>
</tr>
<tr>
<td>movl -4(%ebp), %eax</td>
<td></td>
</tr>
<tr>
<td>movl %eax, -4(%ebp)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, -4(%R1,%R2,4)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, %R1</td>
<td></td>
</tr>
</tbody>
</table>
RISCing the CISC

- Everyone believes that RISC ISAs are better for building fast processors.
- So, how do Intel and AMD build fast x86 processors?
  - Despite using a CISC ISA, these processors are actually RISC processors inside
  - Internally, they convert x86 instructions into MIPS-like micro-ops (uops), and feed them to a RISC-style processor

<table>
<thead>
<tr>
<th>x86 Code</th>
<th>uops</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb $0x05, %al</td>
<td>ori $t0, $t0, 5</td>
</tr>
<tr>
<td>movl -4(%ebp), %eax</td>
<td></td>
</tr>
<tr>
<td>movl %eax, -4(%ebp)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, -4(%R1,%R2,4)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, %R1</td>
<td></td>
</tr>
</tbody>
</table>
RISCing the CISC

• Everyone believes that RISC ISAs are better for building fast processors.

• So, how do Intel and AMD build fast x86 processors?
  • Despite using a CISC ISA, these processors are actually RISC processors inside
  • Internally, they convert x86 instructions into MIPS-like micro-ops (uops), and feed them to a RISC-style processor

<table>
<thead>
<tr>
<th>x86 Code</th>
<th>uops</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb $0x05, %al</td>
<td>ori $t0, $t0, 5</td>
</tr>
<tr>
<td>movl -4(%ebp), %eax</td>
<td>lw $t0, -4($t1)</td>
</tr>
<tr>
<td>movl %eax, -4(%ebp)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, -4(%R1,%R2,4)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, %R1</td>
<td></td>
</tr>
</tbody>
</table>

PC->
RISCing the CISC

• Everyone believes that RISC ISAs are better for building fast processors.
• So, how do Intel and AMD build fast x86 processors?
  • Despite using a CISC ISA, these processors are actually RISC processors inside
  • Internally, they convert x86 instructions into MIPS-like micro-ops (uops), and feed them to a RISC-style processor

<table>
<thead>
<tr>
<th>x86 Code</th>
<th>uops</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb $0x05, %al</td>
<td>ori $t0, $t0, 5</td>
</tr>
<tr>
<td>movl -4(%ebp), %eax</td>
<td>lw $t0, -4($t1)</td>
</tr>
<tr>
<td>movl %eax, -4(%ebp)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, -4(%R1,%R2,4)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, %R1</td>
<td></td>
</tr>
</tbody>
</table>
RISCing the CISC

- Everyone believes that RISC ISAs are better for building fast processors.
- So, how do Intel and AMD build fast x86 processors?
  - Despite using a CISC ISA, these processors are actually RISC processors inside
  - Internally, they convert x86 instructions into MIPS-like micro-ops (uops), and feed them to a RISC-style processor

<table>
<thead>
<tr>
<th>x86 Code</th>
<th>uops</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb $0x05, %al</td>
<td>ori $t0, $t0, 5</td>
</tr>
<tr>
<td>movl -4(%ebp), %eax</td>
<td>lw $t0, -4($t1)</td>
</tr>
<tr>
<td>movl %eax, -4(%ebp)</td>
<td>sw $t0, -4($t1)</td>
</tr>
<tr>
<td>movl %R0, -4(%R1,%R2,4)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, %R1</td>
<td></td>
</tr>
</tbody>
</table>
RISCing the CISC

- Everyone believes that RISC ISAs are better for building fast processors.
- So, how do Intel and AMD build fast x86 processors?
  - Despite using a CISC ISA, these processors are actually RISC processors inside
  - Internally, they convert x86 instructions into MIPS-like micro-ops (uops), and feed them to a RISC-style processor

<table>
<thead>
<tr>
<th>x86 Code</th>
<th>uops</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb $0x05, %al</td>
<td>ori $t0, $t0, 5</td>
</tr>
<tr>
<td>movl -4(%ebp), %eax</td>
<td>lw $t0, -4($t1)</td>
</tr>
<tr>
<td>movl %eax, -4(%ebp)</td>
<td>sw $t0, -4($t1)</td>
</tr>
<tr>
<td>movl %R0, -4(%R1,%R2,4)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, %R1</td>
<td></td>
</tr>
</tbody>
</table>
RISCing the CISC

- Everyone believes that RISC ISAs are better for building fast processors.
- So, how do Intel and AMD build fast x86 processors?
  - Despite using a CISC ISA, these processors are actually RISC processors inside
  - Internally, they convert x86 instructions into MIPS-like micro-ops (uops), and feed them to a RISC-style processor

<table>
<thead>
<tr>
<th>x86 Code</th>
<th>uops</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb $0x05, %al</td>
<td>ori $t0, $t0, 5</td>
</tr>
<tr>
<td>movl -4(%ebp), %eax</td>
<td>lw $t0, -4($t1)</td>
</tr>
<tr>
<td>movl %eax, -4(%ebp)</td>
<td>sw $t0, -4($t1)</td>
</tr>
<tr>
<td>movl %R0, -4(%R1,%R2,4)</td>
<td>slr $at, $t2, 2</td>
</tr>
<tr>
<td>movl %R0, %R1</td>
<td>add $at, $at, $t1</td>
</tr>
<tr>
<td></td>
<td>sw $t0, k($at)</td>
</tr>
</tbody>
</table>
RISCing the CISC

- Everyone believes that RISC ISAs are better for building fast processors.
- So, how do Intel and AMD build fast x86 processors?
  - Despite using a CISC ISA, these processors are actually RISC processors inside
  - Internally, they convert x86 instructions into MIPS-like micro-ops (uops), and feed them to a RISC-style processor

<table>
<thead>
<tr>
<th>x86 Code</th>
<th>uops</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb $0x05, %al</td>
<td>ori $t0, $t0, 5</td>
</tr>
<tr>
<td>movl -4(%ebp), %eax</td>
<td>lw $t0, -4($t1)</td>
</tr>
<tr>
<td>movl %eax, -4(%ebp)</td>
<td>sw $t0, -4($t1)</td>
</tr>
<tr>
<td>movl %R0, -4(%R1,%R2,4)</td>
<td>slr $at, $t2, 2</td>
</tr>
<tr>
<td></td>
<td>add $at, $at, $t1</td>
</tr>
<tr>
<td></td>
<td>sw $t0, k($at)</td>
</tr>
<tr>
<td>movl %R0, %R1</td>
<td></td>
</tr>
</tbody>
</table>
RISCing the CISC

- Everyone believes that RISC ISAs are better for building fast processors.
- So, how do Intel and AMD build fast x86 processors?
  - Despite using a CISC ISA, these processors are actually RISC processors inside
  - Internally, they convert x86 instructions into MIPS-like micro-ops (uops), and feed them to a RISC-style processor

<table>
<thead>
<tr>
<th>x86 Code</th>
<th>uops</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb $0x05, %al</td>
<td>ori $t0, $t0, 5</td>
</tr>
<tr>
<td>movl -4(%ebp), %eax</td>
<td>lw $t0, -4($t1)</td>
</tr>
<tr>
<td>movl %eax, -4(%ebp)</td>
<td>sw $t0, -4($t1)</td>
</tr>
<tr>
<td>movl %R0, -4(%R1,%R2,4)</td>
<td>slr $at, $t2, 2</td>
</tr>
<tr>
<td>movl %R0, %R1</td>
<td>add $at, $at, $t1</td>
</tr>
<tr>
<td></td>
<td>sw $t0, k($at)</td>
</tr>
<tr>
<td></td>
<td>ori $t0, $t0, $zero</td>
</tr>
</tbody>
</table>
RISCing the CISC

- Everyone believes that RISC ISAs are better for building fast processors.
- So, how do Intel and AMD build fast x86 processors?
  - Despite using a CISC ISA, these processors are actually RISC processors inside.
  - Internally, they convert x86 instructions into MIPS-like micro-ops (uops), and feed them to a RISC-style processor.

### x86 Code

<table>
<thead>
<tr>
<th>x86 Instruction</th>
<th>MIPS uops</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb $0x05, %al</td>
<td></td>
</tr>
<tr>
<td>movl -4(%ebp), %eax</td>
<td></td>
</tr>
<tr>
<td>movl %eax, -4(%ebp)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, -4(%R1,%R2,4)</td>
<td></td>
</tr>
<tr>
<td>movl %R0, %R1</td>
<td></td>
</tr>
</tbody>
</table>

The preceding was a dramatization. MIPS instructions were used for clarity and because I had some laying around.

No x86 instruction were harmed in the production of this slide.
VLIWing the CISC

- We can also get rid of x86 in software.
- Transmeta did this.
  - They built a processor that was completely hidden behind a “soft” implementation of the x86 instruction set.
  - Their system would translate x86 instruction into an internal VLIW instruction set and execute that instead.
  - Originally, their aim was high performance.
  - That turned out to be hard, so they focused low power instead.
The End