Review: Computer Organization

Instruction: Language of the Machine

MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: \[ A = B + C \]

MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)
MIPS arithmetic

• Design Principle: simplicity favors regularity. Why?
• Of course this complicates some things...
  
  C code:  \[\begin{align*}  A &= B + C + D; \\  E &= F - A; \end{align*}\]
  
  MIPS code:  \begin{align*}  \text{add} & \quad \$t0, \quad \$s1, \quad \$s2 \\  \text{add} & \quad \$s0, \quad \$t0, \quad \$s3 \\  \text{sub} & \quad \$s4, \quad \$s5, \quad \$s0 \end{align*}\n
• Operands must be registers, only 32 registers provided
• Design Principle: smaller is faster. Why?

MIPS arithmetic

• How about “memory operand”?
• For example
  – my_account = my_account + deposit_amount;

• Data
  – Data is stored in memory
  – Each variable has its address

• Translate to MIPS code
  – \[160] = \[160] + \[200]\;

• But, MIPS does not allow memory operands in the arithmetic operation

<table>
<thead>
<tr>
<th>Memory</th>
<th>address</th>
<th>content</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>160</td>
<td>149000</td>
<td>my_account</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>100</td>
<td>deposit_amount</td>
</tr>
</tbody>
</table>
Data Transfer (Memory) Instructions

- Load and store instructions
- The only instructions to access memory
- Example:

  C code: \( A = B + C \)

  MIPS code: lw $s0, ($s3)
  lw $s1, ($s4)
  lw $s2, ($s5)
  add $s0, $s1, $s2

  $s3 : the data itself = “160”
  ($s3) : data in memory addressed by $s3 = “00024608”

- Store word has destination last

So far we’ve learned:

- MIPS
  - loading words but addressing bytes
  - arithmetic on registers only

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2+100]</td>
</tr>
<tr>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2+100] = $s1</td>
</tr>
</tbody>
</table>
2.4 Representing Instructions in the Computer

- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t0, $s1, $s2`
  - registers have numbers, $t0=8$, $s1=17$, $s2=18$
    (page 140, Table 3.13)
- Instruction Format (machine code):
  - Can you guess what the field names stand for?
  - How many registers can it specify (rs, rt, rd)?
  - How many operations can it support?
  - How many functions can it support?

Machine Language

- Consider the load-word and store-word instructions,
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`
- Regularity principle?
So far…

<table>
<thead>
<tr>
<th>Instruction</th>
<th>format</th>
<th>(destination)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R-type</td>
<td>0 reg reg reg 0 32</td>
</tr>
<tr>
<td>sub</td>
<td>R-type</td>
<td>0 reg reg reg 0 34</td>
</tr>
<tr>
<td>lw</td>
<td>I-type</td>
<td>35 reg reg address</td>
</tr>
<tr>
<td>sw</td>
<td>I-type</td>
<td>43 reg reg address</td>
</tr>
</tbody>
</table>

Read memory ($t0$) to $t1$ => lw $t1$, ($t0$) => 35 / 9 / 8 / 0
Write $t1$ to memory ($t0$) = Write memory ($t0$) from $t1$ => sw $t1$, ($t0$) => 43 / 9 / 8 / 0

Example

- $A[300] = h + A[300]$
- Assembly code ($t1 :$ base of the array $A$, $s2 : h$)
  - lw $t0$, 1200($t1$) # temporary register $t0$ = $A[300]$
  - add $t0$, $s2$, $t0$ # temporary register $t0$ = $h + A[300]$
  - sw $t0$, 1200($t1$) # store back into $A[300]$

- Machine code (hexa & binary format) ?

```
lw 35  9  8  1200
add 0  18  8  8  0  32
sw 43  9  8  1200
```

- “Where are those instructions are stored?”
2.5 MIPS logical instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and $1,$2,$3</td>
<td>$1 = $2 &amp; $3</td>
<td>3 reg. operands; Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>or $1,$2,$3</td>
<td>$1 = $2</td>
<td>$3</td>
</tr>
<tr>
<td>nor</td>
<td>nor $1,$2,$3</td>
<td>$1 = \neg (2 \land 3)$</td>
<td>3 reg. operands; Logical NOR</td>
</tr>
<tr>
<td>xor</td>
<td>xor $1,$2,$3</td>
<td>$1 = \oplus 2 \oplus 3$</td>
<td>3 reg. operands; Logical XOR</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sll $1,$2,10</td>
<td>$1 = 2 &lt;&lt; 10</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srl $1,$2,10</td>
<td>$1 = 2 &gt;&gt; 10</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift left logical</td>
<td>slv $1,$2,3</td>
<td>$1 = 2 &lt;&lt; 3</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srlv $1,$2,3</td>
<td>$1 = 2 &gt;&gt; 3</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td>shift right arithm.</td>
<td>sra $1,$2,10</td>
<td>$1 = 2 &gt;&gt; 10</td>
<td>Shift right (sign extend)</td>
</tr>
<tr>
<td>shift right arithm.</td>
<td>srav $1,$2,3</td>
<td>$1 = 2 &gt;&gt; 3</td>
<td>Shift right arith. by variable</td>
</tr>
</tbody>
</table>

(Immediate instruction would be introduced later ...)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and immediate</td>
<td>andi $1,$2,10</td>
<td>$1 = 2 &amp; 10</td>
<td>Logical AND reg, constant</td>
</tr>
<tr>
<td>or immediate</td>
<td>orli $1,$2,10</td>
<td>$1 = 2</td>
<td>10</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xorli $1,$2,10</td>
<td>$1 = 2 \oplus 10</td>
<td>Logical XOR reg, constant</td>
</tr>
</tbody>
</table>

2.6 Instructions for Making Decisions

- Control flow instructions \(\leftarrow\) in addition to arithmetic inst. & memory inst.
  - alter the control flow,
  - i.e., change the "next" instruction to be executed (otherwise, it is address of current inst.+4)

- MIPS conditional branch instructions (compare&branch type):
  
  \begin{verbatim}
  bne $t0, $t1, Label
  beq $t0, $t1, Label
  \end{verbatim}

- Example: if \((i==j)\) \(h = i + j\);
  
  \begin{verbatim}
  bne $s0, $s1, Label
  add $s3, $s0, $s1
  Label: ....
  \end{verbatim}
Representing beq/bne Instructions

- \texttt{bne \$t0, \$t1, Label}
- \texttt{beq \$t0, \$t1, Label}

\begin{verbatim}
beq  
\hspace{1cm} 4 \hspace{1cm} \text{reg} \hspace{1cm} \text{reg} \hspace{1cm} \text{address}
\end{verbatim}

\begin{verbatim}
bne  
\hspace{1cm} 5 \hspace{1cm} \text{reg} \hspace{1cm} \text{reg} \hspace{1cm} \text{address}
\end{verbatim}

Offset as in \texttt{lw/sw} instructions
What’s the base address in this case?
⇒ “PC”: it is called “PC-relative addressing”
⇒ Ranges: 0–2^{16} (64K)
- Can be negative? “Must be”
- Can be byte-boundary? “Cannot be”
- Branch address = PC + offset\{00
- Ranges: \(-2^{17} – 2^{17}\)

* Addresses in Branches

- Instructions:
  - \texttt{bne \$t4, \$t5, Label} \hspace{0.5cm} \text{Next instruction is at Label if \$t4!=\$t5}
  - \texttt{beq \$t4, \$t5, Label} \hspace{0.5cm} \text{Next instruction is at Label if \$t4=\$t5}

- Formats:

\begin{verbatim}
I  
\hspace{1cm} \text{op} \hspace{1cm} \text{rs} \hspace{1cm} \text{rt} \hspace{1cm} 16 \text{bit address}
\end{verbatim}

- Next PC = current PC + Label : “PC-relative addressing”
  - PC = program counter
  - most branches are local (principle of locality)
  - Branching range is ???
    - When “current PC”=20000h
      - The jumping address is 20000~2fff ???
      - The jumping address is 20000-7fff ~ 20000+7fff = 18001–27ff
      => 20000-7ff*4 ~ 20000+7ff*4 = \(9 \times 40000\) (-4)
Representing j Instructions

- \textit{j Label}

\begin{center}
\begin{tikzpicture}
  \draw[->] (0,0) -- (0,2); \draw[->] (0,0) -- (2,0);
  \node at (0,0) {\texttt{j}};
  \node at (0,2) {6}; \node at (2,0) {26};
  \node at (1,1) {2}; \node at (1,0) {address};
\end{tikzpicture}
\end{center}

PC-relative addressing as in beq/bne instructions? NO. Simply because it can specify a larger range. Branch address = offset||00

\Rightarrow\text{Range: } 0-2^{30} (256\text{MB})

Not enough: Branch address (32-bit full address)

= Upper 4-bit from PC \parallel offset \parallel 00

\textit{“Pseudo-direct addressing”}

Addresses in Jumps

Memory is virtually divided into sixteen 256MB chunks and jump address is limited to the chunk where the \textit{“j”} instruction is located.
Loop

- Loop: \( g = g + A[i] \)
  
  \( l = l + j; \)
  
  if \( (l \neq h) \) go to Loop;

- \( A \) is an array of 100 elements, base in \$s5\)
- \( g, h, l \) and \( j \) are stored to \$s1, \$s2, \$s3 and \$s4\)

- Loop:
  
  add \$t1, \$s3, \$s3 #
  
  add \$t1, \$t1, \$t1 #
  
  add \$t1, \$t1, \$s5 # \( \$t1 = ??? \)
  
  lw \$t0, 0(\$t1)

  add \$s1, \$s1, \$t0
  
  add \$s3, \$s3, \$s4 # step is \$s4 (j)

  bne \$s3, \$s2, Loop

Control Flow

- We have: beq, bne, what about Branch-if-less-than ("blt")?
- New instruction:

  \[
  \text{if } \$s1 < \$s2 \text{ then} \]
  
  \[
  \$t0 = 1 \]
  
  \[
  \text{else} \]
  
  \[
  \$t0 = 0 \]

  slt \$t0, \$s1, \$s2

- Can use this instruction to build "$blt \$s1, \$s2, Label$"

  \( \text{(slt + bne = blt)} \)

  — can now build general control structures

- Note that the assembler needs a register to do this,

  — there are policy of use conventions for registers
So far:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1,$s2,$s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1,$s2,$s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>lw $s1,100($s2)</td>
<td>$s1 = \text{Memory}[$s2+100]</td>
</tr>
<tr>
<td>sw $s1,100($s2)</td>
<td>\text{Memory}[$s2+100] = $s1</td>
</tr>
<tr>
<td>bne $s4,$s5,L</td>
<td>Next instr. is at Label if $s4 \neq $s5</td>
</tr>
<tr>
<td>beq $s4,$s5,L</td>
<td>Next instr. is at Label if $s4 = $s5</td>
</tr>
<tr>
<td>j Label</td>
<td>Next instr. is at Label</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formats:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>op</td>
</tr>
<tr>
<td></td>
<td>(add, sub)</td>
</tr>
<tr>
<td>I</td>
<td>op</td>
</tr>
<tr>
<td></td>
<td>(lw, sw)</td>
</tr>
<tr>
<td>J</td>
<td>op</td>
</tr>
<tr>
<td></td>
<td>(j)</td>
</tr>
</tbody>
</table>

Which type is bne, beq, or slt inst.?