CS3350B Computer Architecture
MIPS Introduction

Marc Moreno Maza

http://www.csd.uwo.ca/~moreno/cs3350_moreno/index.html
Department of Computer Science
University of Western Ontario, Canada

Tuesday January 24, 2017
Abstraction of machine structures

- Levels of representation
Instructions: Language of the Computer
Instruction Set

- **Machine instructions** form the language of the Computer, known as the *instruction set*
- Different computers have different instruction sets:
  - but with many aspects in common;
- early computers had very simple instruction sets
  - due to simplified implementation w.r.t. today’s computers
- Nevertheless, many modern computers also have simple instruction sets
The MIPS instruction set

- Used as the example throughout this course
- For history, see https://en.wikipedia.org/wiki/MIPS_instruction_set
- MIPS stand for *Microprocessor without Interlocked Pipeline Stages.*
- MIPS has a large share of the *embedded core* market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- MIPS is typical of many modern ISAs
  - See the MIPS Reference card.
spim assembler and simulator

- spim is a simulator that runs MIPS32 assembly language programs
  - It provides a simple assembler, debugger and a simple set of operating system services
  - Interfaces: Spim, XSpim, PCSpim, QtSpim (new UI, cross-platform)
- See installation and user guide at
- We recommend to use the command line versions: Spim and XSpim
Arithmetic operations

- Add and subtract have three operands
  - two sources and one destination
    \[
    \text{add } a, b, c \quad \# \text{ a gets } b + c
    \]
- All arithmetic operations have this form
- **Design principle 1**: simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost
ArithmeticeExample

- C code:
  \[ f = (g + h) - (i + j); \]
- Compiled MIPS code:
  
  ```
  add t0, g, h  # temp t0 = g + h
  add t1, i, j  # temp t1 = i + j
  sub f, t0, t1  # f = t0 - t1
  ```
Register Operands

- Arithmetic instructions use register operands
- MIPS has a $32 \times 32$-bit register file
  - use for frequently accessed data
  - numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $t0, t1, \ldots, t9$ for temporary values
  - $s0, s1, \ldots, s7$ for saved variables
- Design Principle 2: smaller is faster
  - in comparison to main memory which has millions of locations
Register operand example

- C code:
  \[
  f = (g + h) - (i + j);
  \]
  - \( f, \ldots, j \) in $s0, \ldots, $s4

- Compiled MIPS code:
  
  add $t0, $s1, $s2
  add $t1, $s3, $s4
  sub $s0, $t0, $t1
Memory operands

- Main memory used for storing composite data:
  - Arrays, structures, dynamic data
- To apply an arithmetic operation, we need to
  - \textbf{load} values from memory into registers, and
  - \textbf{store} the result from register to memory
- Memory is \textbf{byte addressable}
  - Each address identifies a word (= 4 bytes = 32 bits)
- each word is \textbf{aligned} in memory, that is,
  - its address must be a multiple of \textbf{4}
- MIPS is Big Endian
  - that is, it stores the most significant byte in the smallest address,
  - in contrast, with \textit{little endian}, the least-significant byte is at the smallest address.
  - Read https://en.wikipedia.org/wiki/Endianness
Memory operand example 1

- **C code:**
  ```c
  g = h + A[8];
  ```
  - assume g in $s1, h in $s2, and the base address of A in $s3

- **Compiled MIPS code:**
  ```mips
  With 4 bytes per word, the index 8 requires an offset of 32
  lw  $t0, 32($s3)  # load word
  add $s1, $s2, $t0
  ```
Memory Operand example 2

- C code:
  
  \[
  \]

  - \(h\) in \$s2, base address of \(A\) in \$s3

- Compiled MIPS code:

  
  ```mips
  lw $t0, 32($s3)       # load word
  add $t0, $s2, $t0
  sw  $t0, 48($s3)      # store word
  ```
Registers vs. memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - Thus more instructions to be executed
- Compiler must use registers for variables as much as possible
  - only spill to memory for less frequently used variables
  - register optimization is important!
Immediate operands

- **Constant data** specified in an instruction
  
  ```
  addi $s3, $s3, 4
  ```

- There is no subtract immediate instruction
  
  - just use a negative constant
    
    ```
    addi $s2, $s1, -1
    ```

- **Design Principle 3**: make the common case fast
  
  - small constants are common
  
  - immediate operand avoids a load instruction
The constant zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - for instance, for copying between registers
    \[ \text{add } \$t2, \$s1, \$zero \]
Overview: MIPS R3000 ISA

- Instruction categories
  - computational
  - load/Store
  - jump and Branch
  - floating point coprocessor
  - memory management
  - special

- 3 basic instruction formats: all 32 bits wide

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sha</th>
<th>funct</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R0 - R31</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>HI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
<th>R-format</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OP</th>
<th>jump target</th>
<th>J-format</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## MIPS ISA: selected instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instr</th>
<th>OP/ funct</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add R</td>
<td>0/32</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td></td>
<td>subtract R</td>
<td>0/34</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td></td>
<td>add immediate I</td>
<td>8</td>
<td>addi $s1, $s2, 6</td>
<td>$s1 = $s2 + 6</td>
</tr>
<tr>
<td></td>
<td>or immediate I</td>
<td>13</td>
<td>ori $s1, $s2, 6</td>
<td>$s1 = $s2 ∧ 6</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>load word I</td>
<td>35</td>
<td>lw $s1, 24($s2)</td>
<td>$s1 = Memory($s2+24)</td>
</tr>
<tr>
<td></td>
<td>store word I</td>
<td>43</td>
<td>sw $s1, 24($s2)</td>
<td>Memory($s2+24) = $s1</td>
</tr>
<tr>
<td></td>
<td>load byte I</td>
<td>32</td>
<td>lb $s1, 25($s2)</td>
<td>$s1 = Memory($s2+25)</td>
</tr>
<tr>
<td></td>
<td>store byte I</td>
<td>40</td>
<td>sb $s1, 25($s2)</td>
<td>Memory($s2+25) = $s1</td>
</tr>
<tr>
<td></td>
<td>load upper imm I</td>
<td>15</td>
<td>lui $s1, 6</td>
<td>$s1 = 6 * 2^{16}</td>
</tr>
<tr>
<td>Cond. Branch</td>
<td>br on equal I</td>
<td>4</td>
<td>beq $s1, $s2, L</td>
<td>if ($s1===$s2) go to L</td>
</tr>
<tr>
<td></td>
<td>br on not equalI</td>
<td>5</td>
<td>bne $s1, $s2, L</td>
<td>if ($s1 !=$s2) go to L</td>
</tr>
<tr>
<td></td>
<td>set on less than R 0/42</td>
<td>0/42</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2&lt;$s3) $s1=1 else $s1=0</td>
</tr>
<tr>
<td></td>
<td>set on less than immediate I 10</td>
<td>10</td>
<td>slti $s1, $s2, 6</td>
<td>if ($s2&lt;6) $s1=1 else $s1=0</td>
</tr>
<tr>
<td>Uncond. Jump</td>
<td>jump J</td>
<td>2</td>
<td>j 250</td>
<td>go to 1000</td>
</tr>
<tr>
<td></td>
<td>jump register R</td>
<td>0/8</td>
<td>jr $t1</td>
<td>go to $t1</td>
</tr>
<tr>
<td></td>
<td>jump and link J</td>
<td>3</td>
<td>jal 250</td>
<td>go to 1000; $ra=PC+4</td>
</tr>
</tbody>
</table>
## MIPS register convention

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserve on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0 (hardware)</td>
<td>n.a.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>returned values</td>
<td>no</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16-23</td>
<td>saved values</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$k</td>
<td>26-27</td>
<td>Interrupt/trap handler</td>
<td>yes</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return addr (hardware)</td>
<td>yes</td>
</tr>
</tbody>
</table>
Unsigned binary integers

- Given an n-bit number

\[ x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ldots + x_12^1 + x_02^0 \]

- Range: 0 to + 2^m - 1

- Example

\[
\begin{align*}
0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1011_2 &= 0 + \ldots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
&= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}
\end{align*}
\]

- Using 32 bits: 0 to +4,294,967,295
2s-complement signed integers

- Given an n-bit number

\[ x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_12^1 + x_02^0 \]

- Range: \(-2^{n-1}\) to \(+2^{n-1}-1\)

- Example

\[
\begin{align*}
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2 &= -1 \times 2^{31} + 1 \times 2^{30} + \cdots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\
&= -2,147,483,648 + 2,147,483,644 = -4_{10}
\end{align*}
\]

- Using 32 bits: -2,147,483,648 to +2,147,483,647
2s-complement signed integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- \([-(-2^n - 1)\) can’t be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 ... 0000
  - -1: 1111 1111 ... 1111
  - Most-negative: 1000 0000 ... 0000
  - Most-positive: 0111 1111 ... 1111
Signed negation

- **Complement and add 1**
  - Complement means $1 \rightarrow 0$, $0 \rightarrow 1$
    - $x + \bar{x} = 1111 \ldots 111_2 = -1$
    - $\bar{x} + 1 = -x$
  - Example: negate $+2$
    - $+2 = 0000 \ 0000 \ldots \ 0010_2$
    - $-2 = 1111 \ 1111 \ldots \ 1101_2 + 1$
      - $= 1111 \ 1111 \ldots \ 1110_2$
Sign extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb, lh`: extend loaded byte/halfword
- Replicate the sign bit to the left
  - unsigned values are extended with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 ⇒ 0000 0000 0000 0010
  - -2: 1111 1110 ⇒ 1111 1111 1111 1110