CS3350B Computer Architecture
MIPS Introduction

Ning Xie

http://www.csd.uwo.ca/courses/CS3350b
Department of Computer Science
University of Western Ontario, Canada

Winter, 2016
Abstraction of Machine Structures

- Levels of representation

```
<table>
<thead>
<tr>
<th>Software</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Application (ex: browser)</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>Compiler</td>
<td></td>
</tr>
<tr>
<td>Assembler</td>
<td>Processor</td>
</tr>
<tr>
<td>Operating System (Mac OSX)</td>
<td>Memory</td>
</tr>
<tr>
<td></td>
<td>I/O system</td>
</tr>
<tr>
<td></td>
<td>Datapath &amp; Control</td>
</tr>
<tr>
<td></td>
<td>Digital Design</td>
</tr>
<tr>
<td></td>
<td>Circuit Design</td>
</tr>
<tr>
<td></td>
<td>Transistors</td>
</tr>
</tbody>
</table>
```

Ning Xie (http://www.csd.uwo.ca/courses/CS3350B) Computer Architecture MIPS Intro Winter, 2016
Instructions: Language of the Computer
The repertoire of instructions of a computer

Different computers have different instruction sets
  - But with many aspects in common

Early computers had very simple instruction sets
  - Simplified implementation

Many modern computers also have simple instruction sets
The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (http://www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendixes B and E
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
Arithmetic Operations

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

- 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
  - c.f. main memory: millions of locations
Register Operand Example

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

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

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressable
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Store the most significant byte in the smallest address
  - c.f. Little Endian: least-significant byte in the smallest address

Ning Xie (http://www.csd.uwo.ca/courses/CS3350B Computer Architecture  MIPS Intro  Winter, 2016)
Memory Operand Example 1

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

- Compiled MIPS code:
  
  - Assume 4 bytes per word, then index 8 requires offset of 32
  
  ```mips
  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:

  - Assume 4 bytes per word, then index 8 requires offset of 32

  ```
  lw  $t0, 32($s3)   # load word
  add $t0, $s2, $t0
  sw  $t0, 48($s3)  # store word
  ```

Ning Xie (http://www.csd.uwo.ca/courses/CS3350B) Computer Architecture MIPS Intro
Winter, 2016 13 / 24
Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - 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
  
  \[ \text{addi } $s3, $s3, 4 \]

- No subtract immediate instruction
  - Just use a negative constant
    
    \[ \text{addi } $s2, $s1, -1 \]

- **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
  - E.g., move between registers
    
    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>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>OP</th>
<th>jump target</th>
</tr>
</thead>
<tbody>
<tr>
<td>J-format</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Instr</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>add</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
</tr>
<tr>
<td></td>
<td>or immediate</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>load word</td>
</tr>
<tr>
<td></td>
<td>store word</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
</tr>
<tr>
<td></td>
<td>load upper imm</td>
</tr>
<tr>
<td>Cond. Branch</td>
<td>br on equal</td>
</tr>
<tr>
<td></td>
<td>br on not equal</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
</tr>
<tr>
<td></td>
<td>set on less than immediate</td>
</tr>
<tr>
<td>Uncond. Jump</td>
<td>jump</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
</tr>
<tr>
<td></td>
<td>jump and link</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
  \[ x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_12^1 + x_02^0 \]
- Range: 0 to \( +2^n - 1 \)
- Example
  \[
  \begin{align*}
  0000 0000 0000 0000 0000 0000 0000 1011_2 \\
  &= 0 + \cdots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
  &= 0 + \cdots + 8 + 0 + 2 + 1 = 11_{10}
  \end{align*}
  \]
- Using 32 bits: 0 to +4,294,967,295
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

1111 1111 1111 1111 1111 1111 1111 1100₂

\[ = -1 \times 2^{31} + 1 \times 2^{30} + \cdots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \]
\[ = -2,147,483,648 + 2,147,483,644 = -4_{10} \]

- 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 $1 \rightarrow 0$, $0 \rightarrow 1$
    \[
    x + \overline{x} = 1111 \ldots 111_2 = -1
    \]
    \[
    \overline{x} + 1 = -x
    \]
  - 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
  - `beq, bne`: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 ⇒ 0000 0000 0000 0010
  - -2: 1111 1110 ⇒ 1111 1111 1111 1110