Machine Instructions - I

Hwansoo Han
Computer Architecture

- **Architecture** defines
  - Instruction set architecture (ISA)
    - Instructions
    - Operand types
    - Data types (integer, floating-point, …)
    - Memory addressing modes
    - …
  - Registers and other states
  - Interrupt/execution model
  - Memory management and protection
  - Virtual and physical address layout
  - I/O model
  - …
Computer Architecture

- Microarchitecture
  - Organization of the machine below the visible level
    - Number/location of functional units
    - Pipeline/cache configurations
    - Programmer transparent techniques: prefetching, …
  - Must provide the same meaning (semantics) as the visible architecture model

- Implementation
  - Hardware realization
  - Logic circuits, VLSI technology, manufacturing process, …
Computer Architecture

Architecture Brand Name

XScale (IXA)

IA-32

Intel 64 (IA-32e, EM64T, x86-64)

Itanium (IA-64)

Microarchitecture Brand Name

P5
- Pentium
- Pentium MMX

P6
- Pentium Pro
- Pentium II
- Pentium III

NetBurst
- Pentium 4
- Pentium D

Mobile
- Pentium M

Core
- Core Duo/Solo

Processor Brand Name

Processor Code Name

- P5, P54C, P54CS
- P55C, Tillamook

- Pentium Pro
- Klamath, Deschutes
- Katmai, Coppermine, Tualatin

- Willamette, Northwood, Prescott
- Cedar Mill
- Smithfield, Presler

- Banias, Dothan

- Yonah

- Conroe, Allendale, Wolfdale
- Merom, Penryn
- Kentsfield, Yorkfield
Instruction Set

- Language of the Machine
  - Different computers have different instruction sets
  - But many aspects are common
- We’ll use the MIPS instruction set architecture
  - The most popular one is the ARM instruction set (which is similar to MIPS instructions)
  - Another popular one is the x86 instruction set
Arithmetic Instructions

- *Add* and *Subtract*. All instructions have 3 operands.
- Operand order is fixed (destination first)

  C code: \[ a = b + c \]

  MIPS code: `add a, b, c`

- **Design Principle 1**: Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost
Arithmetic Example

- Regularity complicates some things...

  C code: \[ f = (g + h) - (i + j); \]

  MIPS code:  
  ```
  add t0, g, h
  add t1, i, j
  sub f, t0, t1
  ```

- Operands must be registers, only 32 registers provided
- Each register contains 32 bit data
Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 x 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $t0, $t1, ... , $t9 for temporary values
  - $s0, $s1, ... , $s7 for saved variables
  - 14 registers for other purposes

- Design Principle 2: Smaller is faster
  - c.f. Main memory has millions of locations
## MIPS Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results and expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>

Register 1 ($at) reserved as an assembler temporary for pseudo instructions $k0–$k1(26-27) for operating system
Register Operand Example

- Assume
  \[ f, g, h, i, j \] in \[ s0, s1, s2, s3, s4 \]

C code: \[ f = (g + h) - (i + j); \]

MIPS code: add \[ t0, s1, s2 \]
  add \[ t1, s3, s4 \]
  sub \[ s0, t0, t1 \]
Memory

- The operands of arithmetic instructions must be registers
  - But only 32 registers are provided
- Compiler associates variables with registers
- What about programs with lots of variables?
Memory Operands

- Main memory used for
  - Composite data: arrays, structures, dynamic data
  - Global variables

- To apply arithmetic operations on memory operands
  - Load values from memory into registers
  - Store result from register to memory
Memory: Byte Addressed

- Viewed as a large, single-dimension array, with an address.
  - A memory address is an index into the array
  - "Byte addressing" means that the index points to a byte of memory.

<table>
<thead>
<tr>
<th></th>
<th>8 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Memory: Words are Aligned

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits (or 4 bytes).

\[
\begin{array}{c|c}
0 & 32 \text{ bits of data} \\
4 & 32 \text{ bits of data} \\
8 & 32 \text{ bits of data} \\
12 & 32 \text{ bits of data} \\
\vdots & \\
\end{array}
\]

- \(2^{32}\) bytes with byte addresses from 0 to \(2^{32} - 1\)
- \(2^{30}\) words with byte addresses 0, 4, 8, \ldots \(2^{32} - 4\)
- Words are aligned
  - Address must be a multiple of 4
Memory: Byte Ordering

- **Big Endian**
  - Most-significant byte at least address of a word

- **Little Endian**
  - least-significant byte at least address of a word

**Example**
- Variable $x$ has 4-byte representation $0x01234567$
- Address of $x$ (&$x$) is $0x100$

**Big Endian**

\[
\begin{array}{ccccc}
0x100 & 0x101 & 0x102 & 0x103 \\
\hline
\hline
& & & & \\
01 & 23 & 45 & 67 \\
& & & & \\
\end{array}
\]

**Little Endian**

\[
\begin{array}{ccccc}
0x100 & 0x101 & 0x102 & 0x103 \\
\hline
\hline
67 & 45 & 23 & 01 \\
& & & & \\
\end{array}
\]
Memory Operand: Load Instruction

- Assume:
  - g in $s1, h in $s2, base address of A in $s3
  - A[] is an array of integers
  - A[8]: index 8 requires offset of 32

C code: \[ g = h + A[8]; \]

MIPS code: \[
\text{lw } \$t0, 32(\$s3) \quad \# \text{ load word}
\text{add } \$s1, \$s2, \$t0
\]

- Remember arithmetic operands are registers, not memory! - Can’t write: \[ \text{add } \$s1, \$s2, 32(\$s3) \]
Memory Operand: Store Instruction

- Assume:
  - h in $s2, base address of A in $s3
  - A[] is an array of integers
- A[12]: index 12 requires offset of 48

C code:
\[ A[12] = h + A[8]; \]

MIPS code:
\[
\begin{align*}
\text{lw} & \quad \text{at 32($s3)} & \# \text{load word} \\
\text{add} & \quad \text{at 0, s2, at 0} \\
\text{sw} & \quad \text{at 0, 48($s3)} & \# \text{store word}
\end{align*}
\]

Remember arithmetic operands are registers, not memory! - Can’t write: \[ \text{add 48($s3), s2, 32($s3)} \]
Register vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed

- Compilers must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Compilers try to associate all local variables with registers
    - If run out of registers, register allocation phase selects register values to spill to the memory for a range of code
  - Compilers associate all global variables and composite data with memory
    - Use load and store instructions when operations for them are needed
Immediate Operands

- Constant data specified in an instruction

  \[
  \text{addi} \quad \$s3, \$s3, 4 \quad \# \text{add immediate}
  \]

- No \textit{subtract immediate} instruction
  - Use a negative constant

  \[
  \text{addi} \quad \$s2, \$s1, -1
  \]

- Design Principle 3: \textbf{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    # mov $t2, $s1
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

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

- 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

\[ \begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array} \]

\[ = -1 \times 2^{31} + 1 \times 2^{30} + \cdots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \]

\[ = -2,147,483,648 + 2,147,483,644 = -4_{10} \]

- Using 32 bits

- \(-2,147,483,648\) to \(+2,147,483,647\)
2s-Complement Signed Integers

- Bit 31 is a 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

- Obtain the number with the same absolute value but with opposite sign
  - Complement and add 1
  - Complement means $1 \rightarrow 0$, $0 \rightarrow 1$

\[
\bar{x} + 1 = 1111...111_2 = -1
\]

\[
\bar{x} + 1 = -x
\]

- Example: negate +2
  - $+2 = 0000\ 0000\ ...\ 0010_2$
  - $-2 = 1111\ 1111\ ...\ 1101_2 + 1$
    \[
    = 1111\ 1111\ ...\ 1110_2
    \]
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
Representing Instructions

- Instructions are encoded in binary
  - Called machine code

- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, ...
  - Regularity!

- Register numbers
  - $t0 - t7$ are reg’s 8 – 15
  - $t8 - t9$ are reg’s 24 – 25
  - $s0 - s7$ are reg’s 16 – 23
### MIPS R-format Instructions

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>operation code (opcode)</td>
<td>6</td>
</tr>
<tr>
<td>rs</td>
<td>first source register number</td>
<td>5</td>
</tr>
<tr>
<td>rt</td>
<td>second source register number</td>
<td>5</td>
</tr>
<tr>
<td>rd</td>
<td>destination register number</td>
<td>5</td>
</tr>
<tr>
<td>shamt</td>
<td>shift amount (00000 for now)</td>
<td>5</td>
</tr>
<tr>
<td>funct</td>
<td>function code (extends opcode)</td>
<td>6</td>
</tr>
</tbody>
</table>

**Instruction fields**

- **op**: operation code (opcode)
- **rs**: first source register number
- **rt**: second source register number
- **rd**: destination register number
- **shamt**: shift amount (00000 for now)
- **funct**: function code (extends opcode)
**R-format Example**

<table>
<thead>
<tr>
<th>op</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>

- **Example:** add $t0$, $s1$, $s2
- registers have numbers, $t0=8$, $s1=17$, $s2=18

<table>
<thead>
<tr>
<th>special</th>
<th>$s1$</th>
<th>$s2$</th>
<th>$t0$</th>
<th>0</th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>8</td>
<td>000000</td>
<td>32</td>
</tr>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

$0000000100011001001000000000100000_2 = 02324029_{16}$
Hexadecimal

- **Base 16**
  - Compact representation of bit strings
  - 4 bits per hex digit

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>4</td>
<td>0100</td>
<td>8</td>
<td>1000</td>
<td>c</td>
<td>1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
<td>9</td>
<td>1001</td>
<td>d</td>
<td>1101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
<td>a</td>
<td>1010</td>
<td>e</td>
<td>1110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
<td>b</td>
<td>1011</td>
<td>f</td>
<td>1111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Example:** eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000
MIPS I-format Instructions

- Immediate arithmetic and load/store instructions
  - \( rt \): destination or source register number
  - Constant: \(-2^{15} \text{ to } +2^{15} - 1\)
  - Address: offset added to base address in \( rs \)

**Design Principle 4:** Good design demands good compromises

- Different formats complicate decoding, but allow the length of all instructions to be the same (32-bits)
- Keep formats as similar as possible
## I-format Example

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- **Example:** `sw $t1, 1200($s3)`
  - registers have numbers, $t1=9$, $s3=19$

  
<table>
<thead>
<tr>
<th>sw</th>
<th>$s3$</th>
<th>$t1$</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>19</td>
<td>9</td>
<td>1200</td>
</tr>
</tbody>
</table>

- **Example:** `addi $s2, $s3, 4`

<table>
<thead>
<tr>
<th>addi</th>
<th>$s3$</th>
<th>$s2$</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>19</td>
<td>18</td>
<td>4</td>
</tr>
</tbody>
</table>
Stored Program Computers

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs
Logical Operations

Instructions for bitwise manipulation

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>Java</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
<td>srl</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>

Useful for extracting and inserting groups of bits in a word
### Shift Operations

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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>

- **shamt**: how many positions to shift

- **Shift left logical**
  - Shift left and fill with 0 bits
  - `sll by i bits`, multiplies by $2^i$

- **Shift right logical**
  - Shift right and fill with 0 bits
  - `srl by i bits`, divides by $2^i$ (unsigned only)
AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

and $t0$, $t1$, $t2$

$\begin{array}{c}
$t2$ \\
0000 0000 0000 0000 0000 0000 1101 1100 0000
\\
$t1$ \\
0000 0000 0000 0000 0000 0011 1100 0000 0000
\\
$t0$ \\
0000 0000 0000 0000 0000 0000 1100 0000 0000
\end{array}$ : mask
### OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```plaintext
or $t0, $t1, $t2
```

<table>
<thead>
<tr>
<th>$t2</th>
<th>0000 0000 0000 0000 0000 1101 1100 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t1</td>
<td>0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
<tr>
<td>$t0</td>
<td>0000 0000 0000 0000 0011 1101 1100 0000</td>
</tr>
</tbody>
</table>
```

: mask
NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - \( a \text{ NOR } b = \text{ NOT}(a \text{ OR } b) \)

\[
\text{nor } \$t0, \$t1, \$zero
\]

\[
\begin{array}{c}
\$t1 \\
0000 0000 0000 0000 0011 1100 0000 0000
\end{array}
\]

\[
\begin{array}{c}
\$t0 \\
1111 1111 1111 1111 1100 0011 1111 1111
\end{array}
\]

Register 0: always read as zero