Chapter 2

Instructions: Language of the Computer
Instruction Set

- 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 ([www.mips.com](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
Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination
  - add a, b, c  # a gets b + c
- All arithmetic operations have this form
- Design Principle 1: Simplicity favours 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 × 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
- 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 addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - c.f. Little Endian: least-significant byte at least address
Memory Operand Example 1

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

- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word
  \[
  \text{lw } \$t0, 32(\$s3) \quad \# \text{ load word} \\
  \text{add } \$s1, \$s2, \$t0
  \]
Memory Operand Example 2

- **C code:**
  
  \[
  \]
  
  - `h` in `$s2`, base address of `A` in `$s3`

- **Compiled MIPS code:**
  
  - Index 8 requires offset of 32
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw $t0, 32($s3)</code></td>
<td>load word</td>
</tr>
<tr>
<td><code>add $t0, $s2, $t0</code></td>
<td></td>
</tr>
<tr>
<td><code>sw $t0, 48($s3)</code></td>
<td>store word</td>
</tr>
</tbody>
</table>
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
  ```
  addi $s3, $s3, 4
  ```

- 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
  - E.g., move between registers
    
    ```
    add $t2, $s1, $zero
    ```
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

  - 0000 0000 0000 0000 0000 0000 0000 1011₂
  
  \[ = 0 + \ldots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \]
  
  \[ = 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10} \]

- 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

\[ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2 \]

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

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

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 $1 \rightarrow 0$, $0 \rightarrow 1$

\[
\bar{x} + x = 1111\ldots111_2 = -1
\]
\[
\bar{x} + 1 = -x
\]

- 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
  - 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>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>

- **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

#### add $t0, $s1, $s2

<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>

<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>0</td>
<td>32</td>
</tr>
</tbody>
</table>

| 0000000 | 10001 | 10010 | 01000 | 00000 | 100000 |

$00000010001100100100000000100000_2 = 02324020_{16}$
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>
</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>
</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>
</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>
</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>
</tr>
</tbody>
</table>

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

<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>

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: $-2^{15}$ 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 32-bit instructions uniformly
  - Keep formats as similar as possible
Stored Program Computers

The BIG Picture

- 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

- **shamt**: how many positions to shift
- **Shift left logical**
  - Shift left and fill with 0 bits
  - \( s11 \) 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

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

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

or $t0, $t1, $t2

<table>
<thead>
<tr>
<th>$t2</th>
<th>0000 0000 0000 0000 0000 0000 1101 1100 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t1</td>
<td>0000 0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
<tr>
<td>$t0</td>
<td>0000 0000 0000 0000 0000 0011 1101 1100 0000</td>
</tr>
</tbody>
</table>
NOT Operations

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

```
nor $t0, $t1, $zero
```

<table>
<thead>
<tr>
<th>$t0</th>
<th>1111 1111 1111 1111 1100 0011 1111 1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t1</td>
<td>0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
</tbody>
</table>

Register 0: always read as zero
Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1
Compiling If Statements

- **C code:**
  
  ```c
  if (i==j) f = g+h;
  else f = g-h;
  ```

  - f, g, ... in $s0, $s1, ...

- **Compiled MIPS code:**
  
  ```
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
  Else: sub $s0, $s1, $s2
  Exit: ...
  ```

  **Assembler calculates addresses**
Compiling Loop Statements

- C code:
  ```c
  while (save[i] == k) i += 1;
  ```
  - i in $s3, k in $s5, address of save in $s6

- Compiled MIPS code:
  ```mips
  Loop: sll $t1, $s3, 2
       add $t1, $t1, $s6
       lw  $t0, 0($t1)
       bne $t0, $s5, Exit
       addi $s3, $s3, 1
       j   Loop
  Exit: ...
  ```
Basic Blocks

- A basic block is a sequence of instructions with:
  - No embedded branches (except at end)
  - No branch targets (except at beginning)

- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks
More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with `beq, bne`
  - `slt $t0, $s1, $s2  # if ($s1 < $s2)`
  - `bne $t0, $zero, L  # branch to L`
Branch Instruction Design

- Why not blt, bge, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- beq and bne are the common case
- This is a good design compromise