Lecture 2

Instructions: Language of the Computer
(Chapter 2 of the textbook)
Instructions: tell computers what to do

Chapter 2 — Instructions: Language of the Computer — 2
Introduction

Chapter 2.1
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
  - 32-bit Computer, i.e., MIPS-32
- Stanford MIPS commercialized by MIPS Technologies (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 A and E
Represent numbers in computer

Chapter 2.4
How to represent integers in a computer?

- Unsigned integer
- Signed integer
  - Two’s complement
  - Sign extension
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 1011_2
    \[ = 0 + \ldots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \]
    \[ = 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10} \]
- Using 32 bits
  - 0 to +4,294,967,295
2s-Complement Signed Integers

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

- 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

- Most significant bit 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 $1 \rightarrow 0$, $0 \rightarrow 1$

\[ x + \overline{x} = 1111...111_{2} = -1 \]
\[ -x = x + 1 \]

- 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. zero extension: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - -2: 1111 1110 => 1111 1111 1111 1110
Overflow

- The result of an arithmetic operation is beyond the representation
- How to decide if overflow occurs
  - Unsigned operations
    - Check if there is a carry (in addition) or borrow (in subtraction)
  - Signed operations
    - Check the signs of the operands and the results
Overflow

- Addition with signed operands
  - $C = A + B$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>No overflow</td>
</tr>
<tr>
<td>+</td>
<td>-</td>
<td>Don’t care</td>
<td>No overflow</td>
</tr>
<tr>
<td>-</td>
<td>+</td>
<td>Don’t care</td>
<td>No overflow</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>+</td>
<td>Overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-</td>
<td>No overflow</td>
</tr>
</tbody>
</table>
Overflow

- Subtraction with signed operands
  - \( C = A - B \)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>Don’t care</td>
<td>No overflow</td>
</tr>
<tr>
<td>+</td>
<td>-</td>
<td>-</td>
<td>No overflow</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>+</td>
<td>No overflow</td>
</tr>
<tr>
<td>-</td>
<td>+</td>
<td>Don’t care</td>
<td>No overflow</td>
</tr>
</tbody>
</table>
Representation of Memory Addresses

- Memory addresses are given in the unit of byte
- A multiple-byte word is stored in multiple consecutive bytes
  - The address of a word matches the address of one of the multiple bytes within the word
    - Use the smallest address in general
    - In MIPS, words must start at addresses of multiples of 4

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>99</td>
</tr>
<tr>
<td>4</td>
<td>152</td>
</tr>
<tr>
<td>8</td>
<td>85</td>
</tr>
<tr>
<td>12</td>
<td>2586</td>
</tr>
</tbody>
</table>
Store words into memory

Most significant byte \( \rightarrow \) Least significant byte

\[
\begin{array}{c|c|c|c}
0A0B0C0D & \vdots & \vdots & 0A0B0C0D \\
\hline
a+3: & 0D & a+3: & 0A \\
\hline
a+2: & 0C & a+2: & 0B \\
\hline
a+1: & 0B & a+1: & 0C \\
\hline
a: & 0A & a: & 0D \\
\hline
\end{array}
\]

Big-endian \( \rightarrow \) Little-endian
Two Key Principles of Machine Design

1. Instructions are represented as numbers and, as such, are indistinguishable from data
2. Programs are stored in alterable memory (that can be read or written to) just like data

- Stored-program concept
  - Programs can be shipped as files of binary numbers
  - Computers can inherit ready-made software provided they are compatible with an existing ISA – leads industry to align around a small number of ISAs
Representing Instructions in the Computer

Chapter 2.5
MIPS-32 ISA

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
  - Memory Management
  - Special

3 Instruction Formats: all 32 bits wide

- R format
- I format
- J format
## 32 Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserved by callee?</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 - $a3</td>
<td>4-7</td>
<td>arguments</td>
<td>no</td>
</tr>
<tr>
<td>$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 - $t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$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 address</td>
<td>no</td>
</tr>
</tbody>
</table>
MIPS Register File

- Holds thirty-two 32-bit registers
  - Two read ports
  - One write port

- Registers are
  - Faster than main memory
    - Register files with more locations are slower
  - Easier for a compiler to use
    - \((A \times B) + (C \times D) - (E \times F)\) can do multiplications in any order
Registers vs. Memory

- Registers are faster to access than memory
  - Operating on memory data requires loads and stores
- Compiler must use registers for variables as much as possible
  - Only spill the less frequently used variables to memory
  - Register optimization is important!
MIPS (RISC) Design Principles

- Simplicity favors regularity
  - Fixed size instructions
  - Small number of instruction formats
  - Opcode is always the 6 most significant bits

- Smaller is faster
  - Limited instruction set
  - Limited number of registers
  - Limited number of addressing mode

- Make the common case fast
  - Arithmetic operands from the register file (load-store machine)
  - Allow instructions to contain immediate operands

- Good design demands good compromises
  - Three instruction formats
MIPS R-format Instructions

- Instruction fields
  - \textbf{op}: operation code (opcode)
    - All R-type instructions use opcode $000000_2$
  - \textbf{rs}: register file address of the first source operand
  - \textbf{rt}: register file address of the second source operand
  - \textbf{rd}: register file address of the result’s destination
  - \textbf{shamt}: shift amount (for shift instructions)
  - \textbf{funct}: function code augmenting the opcode
### R-format Example

#### Table:

<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>

#### Instruction:

```
add $t0, $s1, $s2
```

#### Binary Representation:

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

\[0000,0010,0011,0010,0100,0000,0010,0000_2 = 02324020_{16}\]
R-type Arithmetic Instructions

- MIPS assembly language arithmetic statement
  
  \[
  \text{add} \ $t0, \ $s1, \ $s2 \\
  \text{sub} \ $t0, \ $s1, \ $s2
  \]

- Each arithmetic instruction performs **one** operation

- Each specifies exactly **three** operands that are all contained in the datapath’s register file \((t0, s1, s2)\)
  
  \[
  \text{destination} \leftarrow \text{source1} \ \text{op} \ \text{source2}
  \]

- Instruction Format (R format)
  
  \[
  \begin{array}{cccccc}
  0 & 17 & 18 & 8 & 0 & 0x22
  \end{array}
  \]
R-type Arithmetic Instructions

- MIPS assembly language arithmetic statement
  
  \[
  \text{add} \; \$t0, \; \$s1, \; \$s2
  \]

- Each arithmetic instruction performs one operation

- Each specifies exactly three operands that are all contained in the datapath's register file \((\$t0, \$s1, \$s2)\)

- Instruction Format (R format)

<table>
<thead>
<tr>
<th>0</th>
<th>17</th>
<th>18</th>
<th>8</th>
<th>0</th>
<th>0x22</th>
</tr>
</thead>
</table>

Chapter 2 — Instructions: Language of the Computer — 28
Register Operand Example

- C code:
  
  ```c
  f = (g + h) - (i + j);
  ```
  
  - f, g, h, i, j in $s0, $s1, $s2, $s3, $s4

- Compiled MIPS code:
  
  ```mips
  add $t0, $s1, $s2   # $t0 = g + h
  add $t1, $s3, $s4   # $t1 = i + j
  sub $s0, $t0, $t1   # f = $t0 - $t1
  ```
MIPS I-format Instructions

- Immediate arithmetic and load/store instructions
- Immediate arithmetic
  - addi $rt, $rs, Imm #R[rt]=R[rs]+SignExtImm
  - rt: destination register number
  - rs: source register number
  - immediate: \(-2^{15}\) to \(+2^{15} - 1\), sign extension to 32 bits

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate 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>

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0011</td>
<td>$0000,0000,0000,0011$</td>
<td>$0000,0000,0000,0011$</td>
</tr>
<tr>
<td>1111 1111 1111 1111</td>
<td>$1111,1111,1111,1111$</td>
<td>$1111,1111,1111,1111$</td>
</tr>
</tbody>
</table>

Chapter 2 — Instructions: Language of the Computer — 30
Immediate Operands

- Constant data specified in an instruction
  \[ \text{addi } \$s3, \$s3, 4 \]
- No subtract immediate instruction
  - Just use a negative constant
    \[ \text{addi } \$s2, \$s1, -1 \]

<table>
<thead>
<tr>
<th></th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>001000₂</td>
<td></td>
<td></td>
<td>FFFF₁₆</td>
</tr>
</tbody>
</table>
The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., copy between registers
    add $t2, $s1, $zero
Unsigned Arithmetic Instructions

- **addu**: Add Unsigned
  - `addu $rd, $rs, $rt # R[rd]=R[rs]+R[rt]`

- **addiu**: Add Immediate Unsigned
  - `addiu $rt, $rs, Imm # R[rt]=R[rs]+SignExtImm`

- **subu**: Subtract Unsigned
  - `subu $rd, $rs, $rt # R[rd]=R[rs]-R[rt]`
Memory Operations
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 the least address of a word
  - *c.f.* Little Endian: least-significant byte at the least address
MIPS Memory Access Instructions

- MIPS has two basic data transfer instructions for accessing memory:
  - `lw $t0, 4($s3)` # load a word from memory
  - `sw $t0, 8($s3)` # store a word to memory
  - The data are loaded into (lw) or stored from (sw) a register in the register file
- The memory address, a 32-bit address, is formed by adding the offset value to the content of the base address register:
  - A 16-bit field means accessing is limited to memory locations within a region of $\pm 2^{13}$ or $\pm 8,192$ words ($\pm 2^{15}$ or $\pm 32,768$ bytes) of the address in the base register
MIPS I-format Instructions

- load/store instructions
  - `lw/sw $rt, offset($rs)`
  - `rt`: destination (for load) or source (for store) register number
  - Address: offset added to base address in `rs`
  - Offset range: $-2^{15}$ to $+2^{15} - 1$
Machine Language – Load Instruction

- Load/Store Instruction Format (I format)

\[ lw \, st0, \, 24(\$s3) \]

\[
\begin{array}{c}
35 \\
19 \\
8 \\
24_{10}
\end{array}
\]

\[
24_{10} + \$s3 =
\]

\[
\ldots 0001\, 1000
\]

\[
+ \ldots 1001\, 0100
\]

\[
\ldots 1010\, 1100 = 0x120040ac
\]

Memory

\[
\begin{array}{c}
0xff\, ffff\, ffff \\
0x120040ac \\
0x12004094 \\
0x0000000c \\
0x00000008 \\
0x00000004 \\
0x00000000
\end{array}
\]

Chapter 2 — Instructions: Language of the Computer — 38
Memory Operand Example 1

- C code:
  
  \[
g = h + A[8];
  \]
  
  - A is an array of words
  - 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:
  
  \[ A[12] = h + A[8]; \]
  
  - h in \$s2, base address of A in \$s3

- Compiled MIPS code:
  
  - Index 8 requires offset of 32
  
  \[
  \begin{align*}
  \text{lw} & \quad \text{\$t0, 32($s3)} \quad \# \text{load word} \\
  \text{add} & \quad \text{\$t0, \$s2, \$t0} \\
  \text{sw} & \quad \text{\$t0, 48($s3)} \quad \# \text{store word}
  \end{align*}
  \]
Loading and Storing Bytes

- MIPS provides special instructions to move bytes
  - lb  $t0, 1($s3)  # load byte from memory
  - lbu $t0, 1($s3)  # load unsigned byte from memory
  - sb  $t0, 6($s3)  # store byte to memory

- What 8 bits get loaded and stored?
  - Load byte places the byte from memory in the rightmost 8 bits of the destination register
    - lb: the byte is sign-extended to 32 bits
    - lbu: the byte is zero-extended to 32 bits
  - Store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
    - Leave the other bits in the memory word intact
MIPS provides special instructions to move half words, i.e., 16 bits:

- `lh  $t0, 2($s3)  # load half word from memory`
- `lhu $t0, 4($s3)  # load unsigned half word from memory`
- `sh  $t0, 8($s3)  # store half word to memory`

The address of the half word has to be even:

| 0x29 | 19 | 8 | 16 bit offset |

| Byte_3 | Byte_2 | Byte_1 | Byte_0 |
Logical Operations

Chapter 2.6
### 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 - sll**

- **shamt**: how many positions to shift
- **Shift left logical**
  - Shift left and fill with 0 bits
  - **sll** by *i* bits: multiplies by $2^i$
  - $sll\ $rd, $rt, shamt \# R[rd]=R[rt]<<shamt$
  - $sll\ $t0, $t0, 4$

<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>
Shift Operations - srl

- **shamt**: how many positions to shift
- **Shift right logical**
  - Shift right and fill with 0 bits
  - srl by i bits: divides by $2^i$
    - For non-negative number only

```markdown
srl $rd, $rt, shamt # R[rd]=R[rt]>>shamt
srl $s0, $t0, 4
```

<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>0</th>
<th>0</th>
<th>8</th>
<th>16</th>
<th>4</th>
<th>2</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>
Bitwise Operations

- **Bitwise logical operations in MIPS ISA**
  - and $rd, $rs, $rt # $R[rd] = $R[rs] & $R[rt]
  - or  $rd, $rs, $rt # $R[rd] = $R[rs] | $R[rt]
  - nor $rd, $rs, $rt # $R[rd] = ~ ($R[rs] | $R[rt])

- **Instruction Format (R format)**

<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>

- nor $t0, $t1, $t2

  | 0 | 9 | 10 | 8 | 0 | 0x24 |
AND Operations

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

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

\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{\$t2} & 0000 & 0000 & 0000 & 0000 & 0000 & 1101 & 1100 0000 \\
\text{\$t1} & 0000 & 0000 & 0000 & 0000 & 0011 & 1100 0000 0000 \\
\text{\$t0} & 0000 & 0000 & 0000 & 0000 & 0000 & 1100 0000 0000 \\
\end{array}
\]
OR Operations

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

or $t0, $t1, $t2
NOT Operations

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

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

Register 0: always read as zero

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

- **Bitwise** logical operations with immediate
  - `andi $rt, $rs, Imm # R[rt] = R[rs] & ZeroExtImm`
  - `ori $rt, $rs, Imm # R[rt] = R[rs] | ZeroExtImm`

- **Instruction Format** (I format)

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

- `andi $t0, $t1, 0xFF00`

| 0x0C | 9   | 8   | 0xFF00 |
| 6 bits | 5 bits | 5 bits | 16 bits |
Branch and Jump Instructions

Chapter 2.7
Control Flow Instructions

- 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;

Instruction Format (I format)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Compiling If Statements

- **C code:**
  ```c
  if (i==j) f = g+h;
  else f = g-h;
  ```
  - `f,g,h,i,j` in `$s0,$s1,$s2,$s3,$s4`

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

Assembler calculates addresses
Compiling If Statements

- C code:
  ```c
  if (i==j) f = g+h;
  else f = g-h;
  ```
  - `f`, `g`, `h`, `i`, `j` in `$s0`, `$s1`, `$s2`, `$s3`, `$s4`

- Compiled MIPS code:
  ```
  beq $s3, $s4, If
  sub $s0, $s1, $s2
  j Exit
  If:
  add $s0, $s1, $s2
  Exit: ...
  ```

Assembler calculates addresses
Control Flow Instructions

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially

- `beq $rs, $rt, L1`
- `bne $rs, $rt, L1`

Instruction Format (I format)

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

- How to specify the branch destination address?
Specifying Branch Destinations

- Use a base address (like in `lw`) added to the 16-bit offset
  - Which register? Instruction Address Register (Program Counter)
    - Its use is automatically implied by instructions
    - PC gets updated (PC+4) during the fetch stage so that it holds the address of the next instruction
  - Instruction address is always the multiple of 4. The unit of the offset is word
    - Limit the branch distance to \(-2^{15}\text{to}+2^{15}-1\) (word) instructions from the instruction after the branch instruction from the low order 16 bits of the branch instruction
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:
  ```
  Loop: sll  $t1, $s3, 2
       add  $t1, $t1, $s6
       lw   $t0, 0($t1)
       bne  $t0, $s5, Exit
       addi $s3, $s3, 1
       j    Loop
  Exit: ...
  ```
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 are penalized!

beq and bne are the common case

This is a good design compromise
In Support of Branch Instructions

- We have beq, bne, but what about other kinds of branches (e.g., branch-if-less-than)?
- Set on less than instruction:
  - `slt $rd, $rs, $rt`
    - if (rs < rt) rd = 1; else rd = 0;
    - Instruction format: R format
- Alternate versions of `slt`
  - `sltu $rd, $rs, $rt`
  - `slti $rt, $rs, Imm`
  - `sltiu $rt, $rs, Imm`
Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`

**Example**

- $s0 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$
- $s1 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001$
- `slt $t0, $s0, $s1` # signed
  - $-1 < +1 \Rightarrow t0 = 1$
- `sltu $t0, $s0, $s1` # unsigned
  - $+4,294,967,295 > +1 \Rightarrow t0 = 0$
Pseudo Branch Instructions

- Can use `slt`, `beq`, `bne`, and the register `$zero` to create other conditions
  - less than: \[ \text{blt} \] $s1, \text{ } s2, \text{ } \text{Label} \]
  - \[ \text{slt} \] $at, \text{ } s1, \text{ } s2 \]
  - `bne` \[ $at, \text{ } \text{zero}, \text{ } \text{Label} \]
  - \[ $s1 < \text{ } s2 \]
- less than or equal to: \[ \text{ble} \] $s1, \text{ } s2, \text{ } \text{Label} \]
- greater than: \[ \text{bgt} \] $s1, \text{ } s2, \text{ } \text{Label} \]
- greater than or equal to: \[ \text{bge} \] $s1, \text{ } s2, \text{ } \text{Label} \]

- Such branches are included in the instruction set as pseudo instructions - recognized and expanded by the assembler
- Assembler needs a reserved register \(<at>)
Unconditional Jump

- Jump \((j)\) targets could be anywhere in text segment within 256 MB
  - Encode full address in instruction
  - Instruction format: \(J\) format

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- (Pseudo)Direct jump addressing
  - Target address = \((PC+4)_{31...28}:(address\times4)\)
  - PC is the address of the jump instruction
Form the target address

from the low order 26 bits of the jump instruction

Where the 4 bits come from?

- The upper 4 bits of the address of the instruction following the Jump instruction
### Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Mnemonic</th>
<th>Flags</th>
<th>Opcode</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll $t1, $s3, 2</td>
<td>80000</td>
<td>sll</td>
<td>0</td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>add $t1, $t1, $s6</td>
<td>80004</td>
<td>add</td>
<td>0</td>
<td>9</td>
<td>22</td>
</tr>
<tr>
<td>lw $t0, 0($t1)</td>
<td>80008</td>
<td>lw</td>
<td>35</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>bne $t0, $s5, Exit</td>
<td>80012</td>
<td>bne</td>
<td>5</td>
<td>8</td>
<td>21</td>
</tr>
<tr>
<td>addi $s3, $s3, 1</td>
<td>80016</td>
<td>addi</td>
<td>8</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>j Loop</td>
<td>80020</td>
<td>j</td>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Exit: ... 80024
Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Op</th>
<th>Func</th>
<th>Imm1</th>
<th>Imm2</th>
<th>Imm3</th>
<th>Imm4</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll $t1, $s3, 2</td>
<td>80000</td>
<td>0 0 19 9 2 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $t1, $t1, $s6</td>
<td>80004</td>
<td>0 9 22 9 0 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $t0, 0($t1)</td>
<td>80008</td>
<td>35 9 8 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne $t0, $s5, Exit</td>
<td>80012</td>
<td>5 8 21 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s3, $s3, 1</td>
<td>80016</td>
<td>8 19 19 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j Loop</td>
<td>80020</td>
<td>2 20000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exit: ...</td>
<td>80024</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code.
- Example:

```assembly
beq $s0,$s1, L1
↓
bne $s0,$s1, L2
j L1
L2: ...
```
Procedure Call Convention and Support

- Chapter 2.8
Six Steps in Execution of a Procedure

1. Main routine (callee) places parameters in a place where the procedure (callee) can access them
   - $a0$–$a3$: four argument registers
2. Caller transfers control to the callee
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where the caller can access it
   - $v0$–$v1$: two registers for result values
6. Callee returns control to the caller
   - $ra$: return address register to return to the point of origin
### 32 Registers Revisited

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserved by callee?</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 - $a3</td>
<td>4-7</td>
<td>arguments</td>
<td>no</td>
</tr>
<tr>
<td>$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 - $t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$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 address</td>
<td>no</td>
</tr>
</tbody>
</table>
Instructions for Accessing Procedures

- **MIPS procedure call instruction**
  - Saves PC+8 in register $ra to have a link to the second instruction following the jump instruction for the procedure to return
  
  ```
  jal ProcedureAddress #jump and link
  ```
  
  - Instruction format: J format

  ```
  0x03 26 bit address
  ```

- **Procedure can do a jump on register**

  ```
  jr $ra #return
  ```

  - Instruction format: R format

  ```
  0 rs 0x08
  ```
Calling Convention

- A set of rules for defining
  - Where data are placed during function calls
  - Who saves/restores data
  - What data should be stored
- All calling conventions have a bit in common
  - All store data in registers and on a stack
MIPS Memory Allocation

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage

![Memory Allocation Diagram]
Local Data on the Stack

- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage
Stack

- A stack is a LIFO data structure
  - LIFO = Last-In, First-Out
  - Basic stack operations: push, pop
  - Stack pointer: point to the “top” element in the stack

- Push:
  - Place an element on the top of the stack
  - Update the stack pointer

- Pop:
  - Remove an element from the top of the stack
  - Update the stack pointer
Stack Example

1: initial state

2: push(15)

3: push(-6)

4: pop

5: push(10)

6: push(16)

7: pop

8: push(-3)
Stack in Memory

- A stack is a last-in first-out (LIFO) data structure
- One of the general registers, $sp, is used to address the stack, which “grows” from high address to low address
- **Push** – add data onto the stack
  \[
  \text{sp} = \text{sp} - 4
  \]
  data on stack at new $sp
- **Pop** – remove data from the stack
  \[
  \text{data from stack at sp}
  \text{sp} = \text{sp} + 4
  \]
Procedures

- **Root**
  - Entry function of a program
  - Always a caller
- **Non-leaf procedure**
  - Both a caller and a callee
- **Leaf procedure**
  - Always a callee
Calling Conventions regarding Root

- Root is always a caller
- As a caller it needs to save on the stack:
  - Any arguments and temporaries needed after calling the callee
    - Arguments are given as the program is launched
    - Restore from the stack after calling the callee
- Root has no return address to save
Calling Conventions regarding Leaf Procedures

- A leaf procedure is a callee
- A callee needs to save on the stack:
  - Any saved registers it needs to use for computation
    - Restore from the stack before returning the control back to the caller
- A callee needs to restore the following registers before returning the control back to the caller:
  - Global pointer register
  - Frame pointer register
  - Stack pointer register
Leaf Procedure Example

- C code:
  ```c
  int leaf_example (int g, h, i, j)
  {
    int f;
    f = (g + h) - (i + j);
    return f;
  }
  ```

- Arguments `g, …, j` in `$a0, …, a3`
- `f` in `$s0`
  - The procedure needs to save `$s0` on stack before using it
- Result in `$v0`
**Leaf Procedure Example**

- **MIPS code:**

```mips
leaf_example:
    addi $sp, $sp, -4
    sw $s0, 0($sp)
    add $t0, $a0, $a1
    add $t1, $a2, $a3
    sub $s0, $t0, $t1
    add $v0, $s0, $zero
    lw $s0, 0($sp)
    addi $sp, $sp, 4
    jr $ra
```

- **Save $s0 on stack**
- **Procedure body**
- **Result**
- **Restore $s0**
- **Return**
String Copy Example

- C code (naïve):
  - Null-terminated string
  ```c
  void strcpy (char x[], char y[])
  {
    int i;
    i = 0;
    while (((x[i]=y[i])!='$\0$'))
      i += 1;
  }
  
  Addresses of x, y in $a0, $a1
  i in $s0
  ```
**String Copy Example**

- **MIPS code:**

```mips
strcpy:
    addi $sp, $sp, -4      # adjust stack for 1 item
    sw $s0, 0($sp)         # save $s0
    add $s0, $zero, $zero  # i = 0

L1: add $t1, $s0, $a1     # addr of y[i] in $t1
    lbu $t2, 0($t1)        # $t2 = y[i]
    add $t3, $s0, $a0      # addr of x[i] in $t3
    sb $t2, 0($t3)         # x[i] = y[i]
    beq $t2, $zero, L2     # exit loop if y[i] == 0

addi $s0, $s0, 1         # i = i + 1
    j L1                   # next iteration of loop

L2: lw $s0, 0($sp)        # restore saved $s0
    addi $sp, $sp, 4       # pop 1 item from stack
    jr $ra                 # and return
```
Non-Leaf Procedures

- Non-leaf procedure is both a caller and a callee

- As a caller it needs to save on the stack:
  - Its return address, and any arguments and temporaries needed after calling the callee
  - Restore from the stack after calling the callee

- As a callee it needs to restore the values of the following registers before returning
  - Saved register, global pointer register, frame pointer register, stack pointer register
Non-Leaf Procedure Example

- Factorial
  - \( n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n \)

- C code:
  ```c
  int fact (int n)
  {
    if (n < 2) return 1;
    else return n * fact(n - 1);
  }
  ```
  - Argument \( n \) in \( a0 \)
  - Result in \( v0 \)
Non-Leaf Procedure Example

- MIPS code:

```assembly
fact:
   addi $sp, $sp, -8     # adjust stack for 2 items
   sw   $ra, 4($sp)      # save return address
   sw   $a0, 0($sp)      # save argument
   slti $t0, $a0, 2      # test for n < 2
   beq  $t0, $zero, L1   # if (n>=2), jump to L1
   addi $v0, $zero, 1    # if (n<2), result is 1
   addi $sp, $sp, 8      # pop 2 items from stack
   jr   $ra              # and return
L1:  addi $a0, $a0, -1     # else decrement n
   jal  fact             # recursive call
   nop
   lw   $a0, 0($sp)      # restore original n
   lw   $ra, 4($sp)      # and return address
   addi $sp, $sp, 8      # pop 2 items from stack
   mul  $v0, $a0, $v0    # multiply to get result
   jr   $ra              # and return
```

A C Sort Example to Put It All

Chapter 2.13
C Sort Example

- Illustrates use of assembly instructions for a C *sinking sort* function
  - Sort a sequence into a non-decreasing order
- Swap procedure (leaf)
  ```c
  void swap(int v[], int k)
  {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
  }
  ```
  - v in $a0$, k in $a1$, temp in $t0$
The Procedure Swap

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sll $t1, $a1, 2</code></td>
<td><code>$t1 = k \times 4$</code></td>
</tr>
<tr>
<td><code>add $t1, $a0, $t1</code></td>
<td><code>$t1 = v + (k \times 4)$</code></td>
</tr>
<tr>
<td><code>lw $t0, 0($t1)</code></td>
<td><code>$t0$ (temp) = v[k]</code></td>
</tr>
<tr>
<td><code>lw $t2, 4($t1)</code></td>
<td><code>$t2 = v[k+1]$</code></td>
</tr>
<tr>
<td><code>sw $t2, 0($t1)</code></td>
<td><code>$v[k] = t2$ (v[k+1])</code></td>
</tr>
<tr>
<td><code>sw $t0, 4($t1)</code></td>
<td><code>$v[k+1] = t0$ (temp)</code></td>
</tr>
<tr>
<td><code>jr $ra</code></td>
<td># return to calling routine</td>
</tr>
</tbody>
</table>

# (address of v[k])
The Sort Procedure in C

- Non-leaf (calls swap)

```c
void sort (int v[], int n) {
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1;
            j >= 0 && v[j] > v[j + 1];
            j -= 1) {
            swap(v, j);
        }
    }
}
```

- `v` in $a0$, `n` in $a1$, `i` in $s0$, `j` in $s1$
- Use $s2$ and $s3$ to save `v` and `n`
## The Full Procedure

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sort: addi $sp, $sp, -20</code></td>
<td># make room on stack for 5 registers</td>
</tr>
<tr>
<td><code>sw $ra,16($sp)</code></td>
<td># save $ra on stack</td>
</tr>
<tr>
<td><code>sw $s3,12($sp)</code></td>
<td># save $s3 on stack</td>
</tr>
<tr>
<td><code>sw $s2, 8($sp)</code></td>
<td># save $s2 on stack</td>
</tr>
<tr>
<td><code>sw $s1, 4($sp)</code></td>
<td># save $s1 on stack</td>
</tr>
<tr>
<td><code>sw $s0, 0($sp)</code></td>
<td># save $s0 on stack</td>
</tr>
<tr>
<td>...</td>
<td># procedure body (see next slide)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td><code>exit1: lw $s0, 0($sp)</code></td>
<td># restore $s0 from stack</td>
</tr>
<tr>
<td><code>lw $s1, 4($sp)</code></td>
<td># restore $s1 from stack</td>
</tr>
<tr>
<td><code>lw $s2, 8($sp)</code></td>
<td># restore $s2 from stack</td>
</tr>
<tr>
<td><code>lw $s3,12($sp)</code></td>
<td># restore $s3 from stack</td>
</tr>
<tr>
<td><code>lw $ra,16($sp)</code></td>
<td># restore $ra from stack</td>
</tr>
<tr>
<td><code>addi $sp, $sp, 20</code></td>
<td># restore stack pointer</td>
</tr>
<tr>
<td><code>jr $ra</code></td>
<td># return to calling routine</td>
</tr>
</tbody>
</table>
### The Procedure Body

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add $s2, $a0, $zero</code></td>
<td>save $a0 into $s2 (v)</td>
<td>Pass params</td>
</tr>
<tr>
<td><code>add $s3, $a1, $zero</code></td>
<td>save $a1 into $s3 (n)</td>
<td>Pass params</td>
</tr>
<tr>
<td><code>add $s0, $zero, $zero</code></td>
<td>$i = 0</td>
<td>Outer loop</td>
</tr>
<tr>
<td><code>for1tst: slt $t0, $s0, $s3</code></td>
<td>$t0 = 0 if $s0 ≥ $s3 (i ≥ n)</td>
<td>Outer loop</td>
</tr>
<tr>
<td><code>beq $t0, $zero, exit1</code></td>
<td>go to exit1 if $s0 ≥ $s3 (i ≥ n)</td>
<td>Outer loop</td>
</tr>
<tr>
<td><code>addi $s1, $s0, -1</code></td>
<td>$j = i - 1</td>
<td>Pass params</td>
</tr>
<tr>
<td><code>for2tst: slti $t0, $s1, 0</code></td>
<td>$t0 = 1 if $s1 &lt; 0 (j &lt; 0)</td>
<td>Outer loop</td>
</tr>
<tr>
<td><code>bne $t0, $zero, exit2</code></td>
<td>go to exit2 if $s1 &lt; 0 (j &lt; 0)</td>
<td>Outer loop</td>
</tr>
<tr>
<td><code>sll $t1, $s1, 2</code></td>
<td>$t1 = j * 4</td>
<td>Pass params</td>
</tr>
<tr>
<td><code>add $t2, $s2, $t1</code></td>
<td>$t2 = v + (j * 4)</td>
<td>Pass params</td>
</tr>
<tr>
<td><code>lw $t3, 0($t2)</code></td>
<td>$t3 = v[j]</td>
<td>Pass params</td>
</tr>
<tr>
<td><code>lw $t4, 4($t2)</code></td>
<td>$t4 = v[j + 1]</td>
<td>Pass params</td>
</tr>
<tr>
<td><code>slt $t0, $t4, $t3</code></td>
<td>$t0 = 0 if $t4 ≥ $t3</td>
<td>Outer loop</td>
</tr>
<tr>
<td><code>beq $t0, $zero, exit2</code></td>
<td>go to exit2 if $t4 ≥ $t3</td>
<td>Outer loop</td>
</tr>
<tr>
<td><code>add $a0, $s2, $zero</code></td>
<td>1st param of swap is v (old $a0)</td>
<td>Pass params &amp; call</td>
</tr>
<tr>
<td><code>add $a1, $s1, $zero</code></td>
<td>2nd param of swap is j</td>
<td>Pass params &amp; call</td>
</tr>
<tr>
<td><code>jal swap</code></td>
<td>call swap procedure</td>
<td>Pass params &amp; call</td>
</tr>
<tr>
<td><code>nop</code></td>
<td></td>
<td>Pass params &amp; call</td>
</tr>
<tr>
<td><code>addi $s1, $s1, -1</code></td>
<td>$j = - 1</td>
<td>Pass params &amp; call</td>
</tr>
<tr>
<td><code>j for2tst</code></td>
<td>jump to test of inner loop</td>
<td>Outer loop</td>
</tr>
<tr>
<td><code>exit2: addi $s0, $s0, 1</code></td>
<td>$i =+ 1</td>
<td>Outer loop</td>
</tr>
<tr>
<td><code>j for1tst</code></td>
<td>jump to test of outer loop</td>
<td>Outer loop</td>
</tr>
</tbody>
</table>
Addressing Mode Summary

1. Immediate addressing

2. Register addressing

3. Base addressing

4. PC-relative addressing

5. Pseudodirect addressing

Chapter 2 — Instructions: Language of the Computer — 95
Delay slot

- **Delay slot** is an instruction slot that gets executed without the effects of a preceding instruction.
  - When an instruction is executed, the following instruction is put into the delay slot.
  - In hardware and some simulators, the instruction following jump and branch instructions is always executed.
    - Add ‘nop’ after the jump and branch instructions in your lab to simply coding.