Computer Organization and Structure

Bing-Yu Chen
National Taiwan University
Instructions:  
Language of the Computer

- Operations and Operands  
  - of the Computer Hardware
- Signed and Unsigned Numbers
- Representing Instructions  
  - in the Computer
- Logical Operations
- Instructions for Making Decisions
- Supporting Procedures  
  - in Computer Hardware
- Communicating with People
- MIPS Addressing  
  - for 32-Bit Immediates and Addresses
- Translating and Starting a Program
- Arrays vs. Pointers
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, 3 operands
  - 2 sources and 1 destination

- Operand order is fixed
  - Destination first
  - All arithmetic operations have this form

- Example:
  - C code: \[ a = b + c \]
  - MIPS code: \[ add a, b, c \]
Arithmetic Operations

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

- compiling two C assignments into MIPS
  - C code: \[ a = b + c; \]
    \[ d = a - e; \]
  - MIPS code: \[
    \text{add } a, b, c \\
    \text{sub } d, a, e
  \]

- compiling a complex C assignment into MIPS
  - C code: \[ f = (g + h) - (i + j) \]
  - MIPS code: \[
    \text{add } $t0, g, h \quad \text{# temp } t0 = g + h \\
    \text{add } $t1, i, j \quad \text{# temp } t1 = i + j \\
    \text{sub } f, $t0, $t1 \quad \text{# } f = t0 - t1
  \]
Register Operands

- Of course this complicates some things...
  - C code: \( a = b + c + d; \)
  - MIPS code: `add a, b, c
               add a, a, d`
    - where \( a \) & \( b \) & \( c \) & \( d \) mean **registers**

- Arithmetic instructions use register operands
  - operands must be **registers**
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: \[ f = (g + h) - (i + j) \]
  - assume \( f, \ldots, j \) in \( s0, \ldots, s4 \)

- MIPS code: 
  - `add $t0, $s1, $s2`
  - `add $t1, $s3, $s4`
  - `sub $s0, $t0, $t1`
Registers vs. Memory

- Arithmetic instructions operands must be registers
  - only 32 registers 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
- 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
Big Endian vs. Little Endian

Big Endianness

Little Endianness

Data

Memory
Load & Store Instructions

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

- MIPS code:  
  
lw  $t0, 32($s3)
  
  add $s1, $s2, $t0
  
  index 8 requires offset of 32
  
  4 bytes per word

- can refer to registers by name (e.g., $s2, $t0) instead of number
Load & Store Instructions

  - h in $s2, base address of A in $s3

- MIPS code:  
  - \( \text{lw} \quad $t0, 32($s3) \)
  - \( \text{add} \quad $t0, \quad $s2, \quad $t0 \)
  - \( \text{sw} \quad $t0, \quad 48($s3) \)

- store word has destination last
- remember arithmetic operands are registers, not memory

  - can’t write:  \( \text{add} \quad 48($s3), \quad $s2, \quad 32($s3) \)
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

*e.g. subi*
The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - add $t2, $s1, $zero
    - e.g., move between registers
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
  - \[ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1011_2 \]
  - \[ = 0 + \ldots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \]
  - \[ = 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10} \]

- Using 32 bits
  - 0 to \( +4,294,967,295 \)
2’s-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 $+2,147,483,647$
2’s-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 2’s-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 $1 \rightarrow 0$, $0 \rightarrow 1$
    \[
    x + \overline{x} = 1111\ldots111_2 = -1
    \]
    \[
    \overline{x} + 1 = -x
    \]

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

- “negate” and “complement” are quite different!
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

- **op** = operation code (opcode)  
  - basic operation of the instruction
- **rs / rt / rd**  
  - register source / destination operand
- **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>

- add $t0, $s1, $s2

<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>

\[ \begin{array}{cccccc}
000000 & 10001 & 10010 & 01000 & 00000 & 100000 \\
\end{array} \]

\[ \begin{array}{cccccc}
00000010001100100100000000010000002 \\
= \ 02324020_{16} \]

30
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
  - rs / rt: source or destination 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
### 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>

- lw $t0, 32($s2)

<table>
<thead>
<tr>
<th>lw</th>
<th>$s2</th>
<th>$t0</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>18</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>100011</td>
<td>10010</td>
<td>01000</td>
<td>00000000000100000</td>
</tr>
</tbody>
</table>
C / MIPS / Machine Languages

- C: \[ A[300] = h + A[300] \]
- MIPS: \(lw \ $t0, 1200($t1)\)
  \(add \ $t0, $s2, $t0\)
  \(sw \ $t0, 1200($t1)\)
- Machine Language:

<table>
<thead>
<tr>
<th>35</th>
<th>9</th>
<th>8</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>18</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>43</td>
<td>9</td>
<td>8</td>
<td>1200</td>
</tr>
</tbody>
</table>
Stored Program Concept

- 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

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting program</td>
</tr>
<tr>
<td>(machine code)</td>
</tr>
<tr>
<td>Editor program</td>
</tr>
<tr>
<td>(machine code)</td>
</tr>
<tr>
<td>C compiler</td>
</tr>
<tr>
<td>(machine code)</td>
</tr>
<tr>
<td>Payroll data</td>
</tr>
<tr>
<td>Book text</td>
</tr>
<tr>
<td>C code for editor program</td>
</tr>
</tbody>
</table>

memory for data, programs, compilers, editors, etc.
Logical Operations

- Instructions for bitwise manipulation

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>srl</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bitwise NOT</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
  - `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)
Shift Operations

- NOTICE
  - shift left/right logical is not *I-type*

- Example: `sll $t2, $s0, 4`

- Machine Language:

<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>special</td>
<td>none</td>
<td>$s0</td>
<td>$t2</td>
<td>4</td>
<td>sll</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>16</td>
<td>10</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
AND Operations

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

- and $t0$, $t1$, $t2$
  - $t2 = 0000 0000 0000 0000 0000 1101 1100 0000$
  - $t1 = 0000 0000 0000 0000 0011 1100 0000 0000$
  - $t0 = 0000 0000 0000 0000 0000 0000 1100 0000 0000$
OR Operations

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

- or $t0$, $t1$, $t2$
  - $t2 = 0000 0000 0000 0000 0000 0000 1101 1100 0000$
  - $t1 = 0000 0000 0000 0000 0000 0000 0011 1100 0000 0000$
  - $t0 = 0000 0000 0000 0000 0000 0000 0011 1101 1100 0000$
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
  - $t1 = 0000 0000 0000 0000 0011 1100 0000 0000
  - $t0 = 1111 1111 1111 1111 1100 0011 1111 1111

43
Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- MIPS **conditional branch** instructions:
  - `bne $t0, $t1, Label`
  - `beq $t0, $t1, Label`

Example:

```plaintext
if (i==j) h = i + j;
```

```plaintext
bne $s0, $s1, Label
add $s3, $s0, $s1
Label: ....
```
Unconditional Operations

- MIPS unconditional branch instructions:
  - j Label

- (Un-)Conditional Branch Example:
  if (i==j)
    f=g+h;
  else
    j Exit
  else:
    sub $s0, $s1, $s2
  Exit: ...

- Can you build a simple for / while loop?
Compiling Loop Statements

C:
While (save [i] == k) i += 1;
- assume i in $s3, k in $s5, address of save in $s6

MIPS:
Loop: sll $t1, $s3, 2
      # $t1=4*i
add $t1, $t1, $s6
      # $t1=addr. of save[i]
lw $t0, 0($t1)
      # $t0=save[i]
bne $t0, $s5, Exit
      # go to Exit if save[i]!=k
addi $s3, $s3, 1
      # i+=1
j Loop
      # go to 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 on less than:
  
  \[
  \text{if} \ ($s_3 < s_4) \quad \text{slti} \quad t_1, \ s_3, \ s_4 \\
  t_1 = 1; \\
  \text{else} \\
  t_1 = 0;
  \]

- can use this instruction to build
  "\text{blt} \ s_1, \ s_2, \ Label"
  
  can now build general control structures

- NOTE
  
  - the assembler needs a register to do this,
  - there are policy of use conventions for registers
Branch Instruction Design

- Why not blt, bge, etc?
- Hardware for $<$, $\geq$, ... slower than $=$, $\neq$
  - 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
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
  - $t0 = 1
    - $-1 < +1 \Rightarrow $t0 = 1
- sltu $t0, $s0, $s1  # unsigned
  - $t0 = 0
    - $+4,294,967,295 > +1 \Rightarrow $t0 = 0
Procedure Calling

- Steps required
  - Place parameters in registers
  - Transfer control to procedure
  - Acquire storage for procedure
  - Perform procedure’s operations
  - Place result in register for caller
  - Return to place of call
## Register Usage

<table>
<thead>
<tr>
<th>Name</th>
<th>Register No.</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results &amp; expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries (can be overwritten by callee)</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved (must be saved/restored by callee)</td>
</tr>
<tr>
<td>$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 for assembler, 26-27 for operating system.
Procedure Call Instructions

- **Procedure call: jump and link**
  - `jal ProcedureLabel`
    - Address of following instruction put in $ra
    - Jumps to target address

- **Procedure return: jump register**
  - `jr $ra`
    - Copies $ra to program counter
    - Can also be used for computed jumps
      - e.g., for case/switch statements
Leaf Procedure Example

int leaf_example (int g, int h, int i, int j) {
    int f;

    f = (g+h)-(i+j);
    return f;
}

- Assume
  - Arguments g, ..., j in $a0, ..., $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0
Leaf Procedure Example

```
addi $sp, $sp, -4       # adjust stack for saving $s0
sw $s0, 0($sp)
add $t0, $a0, $a1      # g+h
add $t1, $a2, $a3      # i+j
sub $s0, $t0, $t1      # (g+h)-(i+j)
add $v0, $s0, $zero    # return f ($v0=$s0+0)
lw $s0, 0($sp)
addi $sp, $sp, 4       # adjust stack again
jr $ra                  # jump back to calling routine
```
Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call
Non-Leaf Procedure Example

```c
int fact (int n) {
    if (n < 1)
        return 1;
    else
        return (n * fact (n - 1));
}
```

☐ Assume
- Argument n in $a0
- Result in $v0
Non-Leaf Procedure Example

fact:

```
addi $sp, $sp, -8  # adjust stack for 2 items
sw $ra, 4($sp)    # save the return address
sw $a0, 0($sp)    # save the argument n
slti $t0, $a0, 1  # test for n < 1
beq $t0, $zero, L1 # if n >= 1, go to L1
addi $sp, $sp, 8  # pop 2 items off stack
addi $v0, $zero, 1 # return 1
jr $ra             # return to after jal
L1: addi $a0, $a0, -1 # n >= 1: argument gets (n - 1)
jal fact           # call fact with (n - 1)
lw $a0, 0($sp)     # return from jal: restore argument n
lw $ra, 4($sp)     # restore the return address
addi $sp, $sp, 8   # adjust stack pointer to pop 2 items
mul $v0, $a0, $v0  # return n * fact (n - 1)
jr $ra             # return to the caller
```
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

High address

Low address

$fp

$sp

$fp

saved argument registers (if any)

saved return address

saved saved registers (if any)

local arrays and structures (if any)
Memory Layout

- **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
- **Stack**: automatic storage

![Memory Layout Diagram](image-url)
Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in C++ wide characters, ...
  - Most of the world’s alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings
Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

- `lb rt, offset(rs)`   `lh rt, offset(rs)`
  - Sign extend to 32 bits in rt
- `lbu rt, offset(rs)`   `lhu rt, offset(rs)`
  - Zero extend to 32 bits in rt
- `sb rt, offset(rs)`   `sh rt, offset(rs)`
  - Store just rightmost byte/halfword
String Copy Example

```c
void strcpy (char x[], char y []) {
    int i;

    i = 0;
    while (x[i] = y[i] != '¥0') {
        i = i + 1;
    }
}
```

- Assume
  - Null-terminated string
  - Addresses of x, y in $a0, $a1, i in $s0
String Copy Example

```
addi       $sp, $sp, -4   # i = 0
sw         $s0, 0($sp)
add        $s0, $zero, $zero
L1:add     $t1, $s0, $a1   # address of y[i] in $t1
lb         $t2, 0($t1)    # $t2 = y[i]
add        $t3, $s0, $a0  # address of x[i] in $t3
sb         $t2, 0($t3)    # x[i] = y[i]
beq        $t2, $zero, L2 # if y[i] == 0, go to L2
addi       $s0, $s0, 1    # i = i + 1
j          L1             # go to L1
L2:lw      $s0, 0($sp)    # restore old $s0
addi       $sp, $sp, 4
jr         $ra
```
32-bit Constants

☐ Most constants are small
  ■ 16-bit immediate is sufficient

☐ For the occasional 32-bit constant
  ■ lui rt, constant
    ☐ Copies 16-bit constant to left 16 bits of rt
    ☐ Clears right 16 bits of rt to 0

\[\text{lui } \$s0, 61\]
\[\text{ori } \$s0, \$s0, 2304\]
Branch Addressing

Instructions:
- bne  $s0,$s1,L1
- beq  $s0,$s1,L2

Formats:

<table>
<thead>
<tr>
<th>I</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit number</th>
</tr>
</thead>
</table>

Most branch targets are near branch
- Forward or backward

PC-relative addressing
- Target address = PC + offset × 4
- PC already incremented by 4 by this time
Jump Addressing

- Instructions:
  - j  L1
  - jal  L2

- Formats:
  - J  op  |  26 bit number

- Jump targets could be anywhere in text segment
  - Encode full address in instruction

- (Pseudo)Direct jump addressing
  - Target address = PC_{31...28} : (address × 4)
Target Addressing Example

C:

while (save [i] == k) i += 1;

MIPS:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll</td>
<td>$t1, $s3, 2</td>
<td>80000</td>
</tr>
<tr>
<td>add</td>
<td>$t1, $t1, $s6</td>
<td>80004</td>
</tr>
<tr>
<td>lw</td>
<td>$t0, 0($t1)</td>
<td>80008</td>
</tr>
<tr>
<td>bne</td>
<td>$t0, $s5, Exit</td>
<td>80012</td>
</tr>
<tr>
<td>addi</td>
<td>$s3, $s3, 1</td>
<td>80016</td>
</tr>
<tr>
<td>j</td>
<td>Loop</td>
<td>80020</td>
</tr>
</tbody>
</table>

Exit: 80024...
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: ... 
```
Addressing Mode Summary

- Immediate addressing
  
  $\begin{array}{|c|c|c|c|}
  \hline
  \text{op} & \text{rs} & \text{rt} & \text{immediate} \\
  \hline
  \end{array}$

- Register addressing
  
  $\begin{array}{|c|c|c|c|c|c|}
  \hline
  \text{op} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{funct} \\
  \hline
  \end{array}$
Addressing Mode Summary

- Base addressing

```
<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
</table>
```

Register

Memory

- Byte
- Halfword
- Word

+
Addressing Mode Summary

- PC-relative addressing

- Pseudodirect addressing
Decoding Machine Code

What is the assembly language statement corresponding to this machine instruction?

00af8020\text{\_hex}\n\Rightarrow 0000 0000 1010 1111 1000 0000 0010 0000

- op = 000000 ⇒ R-format
- rs = 00101 (a1)/ rt = 01111 (t7)/ rd = 10000 (s0)
- shamt = 00000 / funct = 100000 ⇒ add

⇒ add $s0, $a1, $t7
C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function

- Swap procedure (leaf)
  - `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

swap:  
sll $t1, $a1, 2  # $t1=k*4
add $t1, $a0, $t1  # $t1=v+(k*4)
               # (addr. of v[k])

lw $t0, 0($t1)  # $t0=v[k]
lw $t2, 4($t1)  # $t2=v[k+1]
sw $t2, 0($t1)  # v[k]=$t2
sw $t0, 4($t1)  # v[k+1] = $t0
jr $ra         # return to
               # calling routine
The Sort Procedure in C

- Non-leaf (calls swap)
  - 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, k in $a1, i in $s0, j in $s1
The Procedure Body

```assembly
move $s2, $a0  # save $a0 into $s2
move $s3, $a1  # save $a1 into $s3
move $s0, $zero  # i = 0

for1tst:
  slt $t0, $s0, $s3  # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
  beq $t0, $zero, exit1  # go to exit1 if $s0 ≥ $s3 (i ≥ n)
  addi $s1, $s0, -1  # j = i - 1

for2tst:
  slti $t0, $s1, 0  # $t0 = 1 if $s1 < 0 (j < 0)
  bne $t0, $zero, exit2  # go to exit2 if $s1 < 0 (j < 0)
  sll $t1, $s1, 2  # $t1 = j * 4
  add $t2, $s2, $t1  # $t2 = v + (j * 4)
  lw $t3, 0($t2)  # $t3 = v[j]
  lw $t4, 4($t2)  # $t4 = v[j + 1]
  slt $t0, $t4, $t3  # $t0 = 0 if $t4 ≥ $t3
  beq $t0, $zero, exit2  # go to exit2 if $t4 ≥ $t3

move $a0, $s2  # 1st param of swap is v (old $a0)
move $a1, $s1  # 2nd param of swap is j
jal swap  # call swap procedure
addi $s1, $s1, -1  # j -= 1

j for2tst  # jump to test of inner loop

exit2:
  addi $s0, $s0, 1  # i += 1
  j for1tst  # jump to test of outer loop

exit1:
```
The Full Procedure

sort: addi $sp,$sp, -20  # make room on stack for 5 registers
    sw $ra, 16($sp)    # save $ra on stack
    sw $s3,12($sp)    # save $s3 on stack
    sw $s2, 8($sp)    # save $s2 on stack
    sw $s1, 4($sp)    # save $s1 on stack
    sw $s0, 0($sp)    # save $s0 on stack
    ...
    ...

exit1: lw $s0, 0($sp)    # restore $s0 from stack
    lw $s1, 4($sp)    # restore $s1 from stack
    lw $s2, 8($sp)    # restore $s2 from stack
    lw $s3,12($sp)    # restore $s3 from stack
    lw $ra,16($sp)    # restore $ra from stack
    addi $sp,$sp, 20  # restore stack pointer
    jr $ra            # return to calling routine
Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address

- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity
Array vs. Pointers in C

void clear1 (int array[], int size) {
    int i;
    for (i = 0; i < size; i += 1)
        array[i] = 0;
}

void clear2 (int *array, int size) {
    int *p;
    for (p = &array[0]; p < &array[size]; p += 1)
        *p = 0;
}
Array Version of Clear in MIPS

add  $t0, $zero, $zero

loop1:  sll  $t1, $t0, 2
add  $t2, $a0, $t1
sw  $zero, 0($t2)
addi  $t0, $t0, 1
slt  $t3, $t0, $a1
bne  $t3, $zero, loop1
Pointer Version of Clear in MIPS

```
add    $t0, $a0, $zero
loop2:  sw    $zero, 0($t0)
     addi   $t0, $t0, 4
     sll    $t1, $a1, 2
     add    $t2, $a0, $t1
     slt    $t3, $t0, $t2
     bne    $t3, $zero, loop2
```
New Pointer Version of Clear

add   $t0, $a0, $zero
sll   $t1, $a1, 2
add   $t2, $a0, $t1

loop2:  sw   $zero, 0($t0)
addi  $t0, $t0, 4
slt   $t3, $t0, $t2
bne   $t3, $zero, loop2
## Comparing the Two Versions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Version 1</th>
<th>Instruction</th>
<th>Version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$t0, $zero, $zero</td>
<td>add</td>
<td>$t0, $a0, $zero</td>
</tr>
<tr>
<td><strong>lp1:</strong></td>
<td>sll $t1, $t0, 2</td>
<td>sll $t1, $a1, 2</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>$t2, $a0, $t1</td>
<td>add</td>
<td>$t2, $a0, $t1</td>
</tr>
<tr>
<td>sw</td>
<td>$zero, 0($t2)</td>
<td>sw</td>
<td>$zero, 0($t0)</td>
</tr>
<tr>
<td>addi</td>
<td>$t0, $t0, 1</td>
<td>addi</td>
<td>$t0, $t0, 4</td>
</tr>
<tr>
<td>slt</td>
<td>$t3, $t0, $a1</td>
<td>slt</td>
<td>$t3, $t0, $t2</td>
</tr>
<tr>
<td>bne</td>
<td>$t3, $zero, <strong>lp1</strong></td>
<td>bne</td>
<td>$t3, $zero, <strong>lp2</strong></td>
</tr>
</tbody>
</table>
Comparison of Array vs. Pointer

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer