Chapter 2

Instructions: Language of the Computer

Baback Izadi
ECE Department
bai@ engr.newpaltz.edu
Instruction Set

- Language of the Machine
- 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
Design Principle 1: simplicity favors regularity

- Regularity makes implementation simpler
- Simplicity enables higher performance at lower cost
- If code size is most important, use variable length instructions
- If performance is most important, use fixed length instructions
The MIPS Instruction Set

- We’ll be working with the MIPS instruction set architecture
- Used as the example throughout the book
- 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
MIPS Arithmetic Operations

- All instructions have 3 operands
  - Add and subtract, three operands
    - Two sources and one destination
- Operand order is fixed (destination first)

Example:

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

MIPS ‘code’: \[ add \ a, b, c \]

- Of course this complicates some things…

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

MIPS code: \[ add \ a, b, c \]
\[ add \ a, a, d \]
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  
  ```
Registers vs. Memory

- All new machines use general purpose registers
  - Registers are faster than memory
  - Registers are easier for a compiler to use
    - e.g. \((A*B) - (C*D) - (E*F)\) can do multiplies in any order vs. stack
  - Registers can hold variables
    - Memory traffic is reduced, so program is sped up (since registers are faster than memory)
    - Code density improves (since register named with fewer bits than memory location)
Registers vs. Memory

- **Design Principle 2**: smaller is faster
- Operands must be registers,
  - 32 registers provided
  - Each register contains 32 bits
- Arithmetic instructions operands must be registers
- Compiler associates variables with registers
- What about programs with lots of variables
  - c.f. main memory: millions of locations
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
  - $t0$, $t1$, …, $t9$ for temporary values
  - $s0$, $s1$, …, $s7$ for saved variables
Register Usage Conventions

<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>

Registers hold 32 bits of data

Register zero **always** has the value **zero** (even if you try to write it)

Register 1 ($at) reserved for assembler, 26-27 for operating system
Register Operand Example

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

- 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

<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>...</td>
<td></td>
</tr>
</tbody>
</table>
Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.
  - Registers hold 32 bits of data
  
  \[
  \begin{array}{c|c|c|c}
  \text{32 bits of data} & \hspace{1cm} & \hspace{1cm} \\
  \hline
  0 & 1 & 2 \\
  4 & 5 & 6 \\
  8 & 9 & 10 \\
  12 & 13 & 14 \\
  \end{array}
  \]

- \(2^{32}\) bytes with byte addresses from 0 to \(2^{32}-1\)
- \(2^{30}\) words with byte addresses 0, 4, 8, … \(2^{32}-4\)
Endianess and Alignment

• MIPS is Big Endian
  • Most-significant byte at least address of a word
  • c.f. Little Endian: least-significant byte at least address

• Words are aligned in memory
  • Alignment: require that objects fall on address that is multiple of their size.
  • Address must be a multiple of 4
  • The least two significant bits of a word address=00
Memory Operand Example 1

- **Memory address = content of a register + constant**
- **C code:**
  
  ```
  g = h + A[8];
  ```
  
  - g in $s1, h in $s2, base address of A in $s3
- **Compiled MIPS code:**
  
  ```
  lw $t0, 32($s3)    # load word
  add $s1, $s2, $t0
  ```
  
  - Index 8 requires offset of 32
    - 4 bytes per word
- offset
- base register
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
    
    lw  \$t0, 32($s3)    # load word
    add \$t0, \$s2, \$t0
    sw  \$t0, 48($s3)    # store word
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!
Summary so far

- **MIPS**
  - Loading words but addressing bytes
  - Arithmetic on registers only

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add $s1, $s2, $s3</code></td>
<td>$s1 &lt;= $s2 + $s3</td>
</tr>
<tr>
<td><code>sub $s1, $s2, $s3</code></td>
<td>$s1 &lt;= $s2 – $s3</td>
</tr>
<tr>
<td><code>lw $s1, 100($s2)</code></td>
<td>$s1 &lt;= Memory[$s2+100]</td>
</tr>
<tr>
<td><code>sw $s1, 100($s2)</code></td>
<td>$s1 =&gt; Memory[$s2+100]</td>
</tr>
</tbody>
</table>
Immediate Operands

- Constant data specified in an instruction
  \[
  \text{addi} \quad $s3, $s3, 4
  \]
- No subtract immediate instruction
  - Just use a negative constant
    \[
    \text{addi} \quad $s2, $s1, -1
    \]
- **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 1011_2
    - = 0 + \ldots + 1\times2^3 + 0\times2^2 +1\times2^1 +1\times2^0
    - = 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}

- Using 32 bits
  - 0 to +4,294,967,295
Signed Negation

- Complement and add 1
  - Complement means $1 \rightarrow 0$, $0 \rightarrow 1$

$$\overline{x + \overline{x}} = 1111...111_2 = -1$$
$$\overline{x} + 1 = -x$$

Example: negate $+2$

- $+2 = 0000\ 0000\ ...\ 0010_2$
- $-2 = 1111\ 1111\ ...\ 1101_2 + 1$
  $$= 1111\ 1111\ ...\ 1110_2$$
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\(_2\)
  - \(-0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100\)
  - \(-4\)\(_{10}\)

- 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
- 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
    - $-(2^n - 1)$ can’t be represented
Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
  - Why do we need to sign extend?
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110
- 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
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

- Instructions are also 32 bits long

<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

<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

### Special Form

<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>

| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |

$0000001000110010010000000100000_2 = 02324020_{16}$
Hexadecimal

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

<table>
<thead>
<tr>
<th></th>
<th>0 0000</th>
<th>4 0100</th>
<th>8 1000</th>
<th>c 1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>4</td>
<td>0100</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
<td>b</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
- Example: `lw $t0, 32($s2)`

<table>
<thead>
<tr>
<th>35</th>
<th>18</th>
<th>8</th>
<th>32</th>
</tr>
</thead>
</table>

- 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
  - `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
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 has NOR 3-operand instruction
  • \( a \text{ NOR } b = \text{ NOT}(a \text{ OR } b) \)

\text{nor} \; $t0, \; $t1, \; $zero

\begin{align*}
$\text{t1} & \quad 0000 \; 0000 \; 0000 \; 0000 \; 0011 \; 1100 \; 0000 \; 0000 \\
$\text{t0} & \quad 1111 \; 1111 \; 1111 \; 1111 \; 1100 \; 0011 \; 1111 \; 1111
\end{align*}

\text{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:
  
  ```assembly
  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 $<$, $\geq$, … 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
  • sltu $t0, $s0, $s1  # unsigned
    • $t0 = 0
Procedure Calling

- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure’s operations
  5. Place result in register for caller
  6. Return to place of call
Register Usage

- $a0 – $a3: arguments (reg’s 4 – 7)
- $v0, $v1: result values (reg’s 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)
Procedure Call Instructions

- Procedure call: jump and link
  \[ \text{jal ProcedureLabel} \]
  - Address of following instruction put in $ra
    - $ra ← PC +4
    - Where you want to return after completion of the procedure
  - Jumps to target address

- Procedure return: jump register
  \[ \text{jr } \ $ra \]
  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements
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 (hence, need to save $s0 on stack)
• Result in $v0
Leaf Procedure Example

- **MIPS code:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>MIPS Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>addi $sp, $sp, -4</td>
<td>Save $s0 on stack</td>
</tr>
<tr>
<td>sw</td>
<td>sw $s0, 0($sp)</td>
<td>Procedure body</td>
</tr>
<tr>
<td>add</td>
<td>add $t0, $a0, $a1 # g+h</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>add $t1, $a2, $a3 # i+j</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>sub $s0, $t0, $t1</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>add $v0, $s0, $zero</td>
<td>Result</td>
</tr>
<tr>
<td>lw</td>
<td>lw $s0, 0($sp)</td>
<td>Restore $s0</td>
</tr>
<tr>
<td>addi</td>
<td>addi $sp, $sp, 4</td>
<td></td>
</tr>
<tr>
<td>jr</td>
<td>jr $ra</td>
<td>Return</td>
</tr>
</tbody>
</table>
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 code:

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

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

- **MIPS code:**

```mips
 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, 1       # test for n < 1
   beq  $t0, $zero, L1
   addi $v0, $zero, 1     # if so, 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
       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
```
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
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, new in Java
- **Stack**: automatic storage
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 Java, 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 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:**

```assembly
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
```
32-bit Constants

- Most constants are small; 16-bit immediate is sufficient
- For the occasional 32-bit constant, \texttt{lui \ rt, constant}
  - Copies 16-bit constant to left 16 bits of \rt
  - Clears right 16 bits of \rt to 0
- To load a 32-bit constant into a register, load each 16-bit separately

\texttt{lui \ $t0, 1010101010101010} \quad // \text{First: "load upper immediate"}

\begin{align*}
\begin{array}{c|c}
1010101010101010 & 0000000000000000 \\
\end{array}
\end{align*}
\text{filled with zeros}

Then must get the lower order bits right, i.e.,
\texttt{ori \ $t0, \ t0, 1100001100111111} \quad // \text{OR immediate}

\begin{align*}
\begin{array}{c|c}
1010101010101010 & 0000000000000000 \\
0000000000000000 & 1100001100111111 \\
\end{array}
\end{align*}
\text{OR}

\begin{align*}
\begin{array}{c|c}
1010101010101010 & 1100001100111111 \\
\end{array}
\end{align*}
Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

<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>

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

- Jump (j and jal) targets could be anywhere in text segment
- Encode full address in instruction

- (Pseudo)Direct jump addressing
  - Target address = PC_{31...28} : (address \times 4)

\[ \text{op} \quad \text{address} \]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>
Target Addressing Example

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

Loop:             | $t1, $s3, 2 | 80000 |
add             | $t1, $t1, $s6 | 80004 |
lw              | $t0, 0($t1) | 80008 |
bne             | $t0, $s5, Exit | 80012 |
addi            | $s3, $s3, 1 | 80016 |
j Loop          |             | 80020 |
Exit:            |             | 80024 |

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>19</th>
<th>9</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>22</td>
<td>9</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>35</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>21</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td>19</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20000</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

  ```
  beq $s0,$s1, L1
  ↓
  bne $s0,$s1, L2
  j L1
  L2: ...
  ```
Addressing Mode Summary

1. Immediate addressing

2. Register addressing

3. Base addressing

4. PC-relative addressing

5. Pseudodirect addressing
Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don’t synchronize
    - Result depends on order of accesses

- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write

- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions
Synchronization in MIPS

- **Load linked**: \( \text{ll} \ rt, \text{offset}(rs) \)
- **Store conditional**: \( \text{sc} \ rt, \text{offset}(rs) \)
  - Succeeds if location not changed since the ll
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt

- **Example: atomic swap (to test/set lock variable)**

```
try:       add    $t0, $zero, $s4  #$t0=$s4 (exchange value)
          ll      $t1, 0($s1)  #load memory value to $t1
          sc      $t0, 0($s1)  #try to store exchange
                      #value to memory, if fail
                      #$t0 will be 0
        beq      $t0, $zero, try  #try again on failure
          add    $s4, $zero, $t1  #load value in $s4
```
Translation and Startup

Many compilers produce object modules directly

C program

Compiler

Assembly language program

Assembler

Object: Machine language module

Object: Library routine (machine language)

Linker

Executable: Machine language program

Loader

Memory

Static linking
Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler’s imagination
  
  \[
  \begin{align*}
  \text{move} & \quad $t0, $t1 \quad \rightarrow \quad \text{add} \quad $t0, $\text{zero}, $t1 \\
  \text{blt} & \quad $t0, $t1, L \quad \rightarrow \quad \text{slt} \quad \text{at}, $t0, $t1 \\
  & \quad \text{bne} \quad \text{at}, $\text{zero}, L \\
  \end{align*}
  \]

  - \(\text{at}\) (register 1): assembler temporary
Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code
Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs

- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space
Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main
     - When main returns, do exit syscall
Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions
Lazy Linkage

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code

a. First call to DLL routine

b. Subsequent calls to DLL routine
Starting Java Applications

- **Java program**
  - Compiler
  - Class files (Java bytecodes)
    - Compiled Java methods (machine language)
    - Interprets bytecodes
    - Compiled Java methods (machine language)
    - Java library routines (machine language)
    - Just In Time compiler
    - Java Virtual Machine
  - Simple portable instruction set for the JVM
    - Compiles bytecodes of “hot” methods into native code for host machine
C Sort Example

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

- 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>swap:</th>
<th>sll</th>
<th>$t1, $a1, 2</th>
<th># $t1 = k * 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add</td>
<td>$t1, $a0, $t1</td>
<td># $t1 = v + (k * 4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td># (address of v[k])</td>
</tr>
<tr>
<td>lw</td>
<td>$t0, 0($t1)</td>
<td># $t0 (temp) = v[k]</td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>$t2, 4($t1)</td>
<td># $t2 = v[k+1]</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>$t2, 0($t1)</td>
<td># v[k] = $t2 (v[k+1])</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>$t0, 4($t1)</td>
<td># v[k+1] = $t0 (temp)</td>
<td></td>
</tr>
<tr>
<td>jr</td>
<td>$ra</td>
<td># return to calling routine</td>
<td></td>
</tr>
</tbody>
</table>
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 $a0$, k in $a1$, i in $s0$, j in $s1$
## The Procedure Body

| move      | $s2, $a0         # save $a0 into $s2 |
| move      | $s3, $a1         # save $a1 into $s3 |
| move      | $s0, $zero       # i = 0 |
| 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 |
| 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 |
### The Full Procedure

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Registers</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addi</code></td>
<td>$sp, $sp, $-20`</td>
<td># make room on stack for 5 registers</td>
</tr>
<tr>
<td><code>sw</code></td>
<td>$ra, 16($sp)</td>
<td># save $ra on stack</td>
</tr>
<tr>
<td><code>sw</code></td>
<td>$s3, 12($sp)</td>
<td># save $s3 on stack</td>
</tr>
<tr>
<td><code>sw</code></td>
<td>$s2, 8($sp)</td>
<td># save $s2 on stack</td>
</tr>
<tr>
<td><code>sw</code></td>
<td>$s1, 4($sp)</td>
<td># save $s1 on stack</td>
</tr>
<tr>
<td><code>sw</code></td>
<td>$s0, 0($sp)</td>
<td># save $s0 on stack</td>
</tr>
<tr>
<td><code>lw</code></td>
<td>$s0, 0($sp)</td>
<td># restore $s0 from stack</td>
</tr>
<tr>
<td><code>lw</code></td>
<td>$s1, 4($sp)</td>
<td># restore $s1 from stack</td>
</tr>
<tr>
<td><code>lw</code></td>
<td>$s2, 8($sp)</td>
<td># restore $s2 from stack</td>
</tr>
<tr>
<td><code>lw</code></td>
<td>$s3, 12($sp)</td>
<td># restore $s3 from stack</td>
</tr>
<tr>
<td><code>lw</code></td>
<td>$ra, 16($sp)</td>
<td># restore $ra from stack</td>
</tr>
<tr>
<td><code>addi</code></td>
<td>$sp, $sp, 20</td>
<td># restore stack pointer</td>
</tr>
<tr>
<td><code>jr</code></td>
<td>$ra</td>
<td># return to calling routine</td>
</tr>
</tbody>
</table>
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

- **Relative Performance**: none, O1, O2, O3
- **Instruction count**: none, O1, O2, O3
- **Clock Cycles**: none, O1, O2, O3
- **CPI**: none, O1, O2, O3

Compiled with gcc for Pentium 4 under Linux
Effect of Language and Algorithm

Bubblesort Relative Performance

Quicksort Relative Performance

Quicksort vs. Bubblesort Speedup
Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!
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
### Example: Clearing and Array

<table>
<thead>
<tr>
<th>clear1(int array[], int size) {</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i;</td>
</tr>
<tr>
<td>for (i = 0; i &lt; size; i += 1)</td>
</tr>
<tr>
<td>array[i] = 0;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>clear2(int *array, int size) {</th>
</tr>
</thead>
<tbody>
<tr>
<td>int *p;</td>
</tr>
<tr>
<td>for (p = &amp;array[0]; p &lt; &amp;array[size]; p = p + 1)</td>
</tr>
<tr>
<td>*p = 0;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>move $t0,$zero  # i = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop1: sll $t1,$t0,2   # $t1 = i * 4</td>
</tr>
<tr>
<td>add $t2,$a0,$t1       # $t2 =</td>
</tr>
<tr>
<td># &amp;array[i]</td>
</tr>
<tr>
<td>sw $zero, 0($t2)      # array[i] = 0</td>
</tr>
<tr>
<td>addi $t0,$t0,1        # i = i + 1</td>
</tr>
<tr>
<td>selt $t3,$t0,$a1      # $t3 =</td>
</tr>
<tr>
<td># (i &lt; size)</td>
</tr>
<tr>
<td>bne $t3,$zero,loop1   # if (..)</td>
</tr>
<tr>
<td># goto loop1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>move $t0,$a0   # p = &amp; array[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop2: sw $zero, 0($t0) # Memory[p] = 0</td>
</tr>
<tr>
<td>addi $t0,$t0,4   # p = p + 4</td>
</tr>
<tr>
<td>selt $t3,$t0,$t2  # $t3 =</td>
</tr>
<tr>
<td># (p&lt;&amp;array[size])</td>
</tr>
<tr>
<td>bne $t3,$zero,loop2  # if (..)</td>
</tr>
<tr>
<td># goto loop2</td>
</tr>
</tbody>
</table>
Comparison of Array vs. Ptr

- 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
ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

<table>
<thead>
<tr>
<th></th>
<th>ARM</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date announced</td>
<td>1985</td>
<td>1985</td>
</tr>
<tr>
<td>Instruction size</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>Address space</td>
<td>32-bit flat</td>
<td>32-bit flat</td>
</tr>
<tr>
<td>Data alignment</td>
<td>Aligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>Data addressing modes</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Registers</td>
<td>15 × 32-bit</td>
<td>31 × 32-bit</td>
</tr>
<tr>
<td>Input/output</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
</tr>
</tbody>
</table>
Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions
Instruction Encoding

ARM

Register-register

MIPS

Data transfer

MIPS

Branch

ARM

Jump/Call

MIPS
The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments
The Intel x86 ISA

• Further evolution...
  • i486 (1989): pipelined, on-chip caches and FPU
    • Compatible competitors: AMD, Cyrix, …
  • Pentium (1993): superscalar, 64-bit datapath
    • Later versions added MMX (Multi-Media eXtension) instructions
    • The infamous FDIV bug
  • Pentium Pro (1995), Pentium II (1997)
    • New microarchitecture (see Colwell, *The Pentium Chronicles*)
  • Pentium III (1999)
    • Added SSE (Streaming SIMD Extensions) and associated registers
  • Pentium 4 (2001)
    • New microarchitecture
    • Added SSE2 instructions
The Intel x86 ISA

• And further...
  • AMD64 (2003): extended architecture to 64 bits
  • EM64T – Extended Memory 64 Technology (2004)
    • AMD64 adopted by Intel (with refinements)
    • Added SSE3 instructions
  • Intel Core (2006)
    • Added SSE4 instructions, virtual machine support
  • AMD64 (announced 2007): SSE5 instructions
    • Intel declined to follow, instead...
  • Advanced Vector Extension (announced 2008)
    • Longer SSE registers, more instructions

• If Intel didn’t extend with compatibility, its competitors would!
  • Technical elegance ≠ market success
## Basic x86 Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>GPR 0</td>
</tr>
<tr>
<td>ECX</td>
<td>GPR 1</td>
</tr>
<tr>
<td>EDX</td>
<td>GPR 2</td>
</tr>
<tr>
<td>EBX</td>
<td>GPR 3</td>
</tr>
<tr>
<td>ESP</td>
<td>GPR 4</td>
</tr>
<tr>
<td>EBP</td>
<td>GPR 5</td>
</tr>
<tr>
<td>ESI</td>
<td>GPR 6</td>
</tr>
<tr>
<td>EDI</td>
<td>GPR 7</td>
</tr>
<tr>
<td>CS</td>
<td>Code segment pointer</td>
</tr>
<tr>
<td>SS</td>
<td>Stack segment pointer (top of stack)</td>
</tr>
<tr>
<td>DS</td>
<td>Data segment pointer 0</td>
</tr>
<tr>
<td>ES</td>
<td>Data segment pointer 1</td>
</tr>
<tr>
<td>FS</td>
<td>Data segment pointer 2</td>
</tr>
<tr>
<td>GS</td>
<td>Data segment pointer 3</td>
</tr>
<tr>
<td>EIP</td>
<td>Instruction pointer (PC)</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>Condition codes</td>
</tr>
</tbody>
</table>
Basic x86 Addressing Modes

- Two operands per instruction

<table>
<thead>
<tr>
<th>Source/dest operand</th>
<th>Second source operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Register</td>
</tr>
<tr>
<td>Register</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>Memory</td>
</tr>
<tr>
<td>Memory</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

- Memory addressing modes
  - Address in register
  - Address = \( R_{\text{base}} + \text{displacement} \)
  - Address = \( R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}} \) (scale = 0, 1, 2, or 3)
  - Address = \( R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}} + \text{displacement} \)
x86 Instruction Encoding

- Variable length encoding
- Postfix bytes specify addressing mode
- Prefix bytes modify operation
  - Operand length, repetition, locking, ...

a. JE EIP + displacement
   4  4 8
   JE Condition Displacement

b. CALL
   8 32
   CALL Offset

c. MOV EBX, [EDI + 45]
   6  1  1  8  8
   MOV d w r/m Postbyte Displacement

d. PUSH ESI
   5  3
   PUSH Reg

e. ADD EAX, #6765
   4  3  1 32
   ADD Reg w Immediate

f. TEST EDX, #42
   7  1  8 32
   TEST w Postbyte Immediate
Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions
Fallacies

- Powerful instruction $\Rightarrow$ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions

- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code $\Rightarrow$ more errors and less productivity
Fallacies

- Backward compatibility ⇒ instruction set doesn’t change
- But they do accrete more instructions
Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped
Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

- Layers of software/hardware
  - Compiler, assembler, hardware

- MIPS: typical of RISC ISAs
  - c.f. x86
# Concluding Remarks

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>MIPS examples</th>
<th>SPEC2006 Int</th>
<th>SPEC2006 FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add, sub, addi</td>
<td>16%</td>
<td>48%</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw, sw, lb, lbu, lh, lhu, sb, lui</td>
<td>35%</td>
<td>36%</td>
</tr>
<tr>
<td>Logical</td>
<td>and, or, nor, andi, ori, sll, srl</td>
<td>12%</td>
<td>4%</td>
</tr>
<tr>
<td>Cond. Branch</td>
<td>beq, bne, slt, slti, sltiu</td>
<td>34%</td>
<td>8%</td>
</tr>
<tr>
<td>Jump</td>
<td>j, jr, jal</td>
<td>2%</td>
<td>0%</td>
</tr>
</tbody>
</table>