Instructions:

- Language of the Machine
- More primitive than higher level languages
  e.g., no sophisticated control flow
- Very restrictive
  e.g., MIPS Arithmetic Instructions

- We’ll be working with the MIPS instruction set architecture
  – similar to other architectures developed since the 1980’s
  – used by NEC, Nintendo, Silicon Graphics, Sony

Design goals: maximize performance and minimize cost, reduce design time

MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: \[ A = B + C \]
MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)
MIPS arithmetic

- **Design Principle**: simplicity favors regularity. Why?
- Of course this complicates some things...

  C code:
  
  ```
  A = B + C + D;
  E = F - A;
  ```

  MIPS code:
  
  ```
  add $t0, $s1, $s2
  add $s0, $t0, $s3
  sub $s4, $s5, $s0
  ```

- Operands must be registers, only 32 registers provided
- **Design Principle**: smaller is faster. Why?

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 Organization

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

Memory Organization

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

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

Registers hold 32 bits of data

• \(2^{32}\) bytes with byte addresses from 0 to \(2^{32}-1\)
• \(2^{30}\) words with byte addresses 0, 4, 8, ... \(2^{32}-4\)
• Words are aligned
  i.e., what are the least 2 significant bits of a word address?
Instructions

• Load and store instructions
• Example:


  MIPS code:
  \[
  \begin{align*}
  &\text{lw }$t0, 32($s3) \\
  &\text{add }$t0, $s2, $t0 \\
  &\text{sw }$t0, 32($s3)
  \end{align*}
  \]

• Store word has destination last
• Remember arithmetic operands are registers, not memory!

Our First Example

• Can we figure out the code?

  \begin{verbatim}
  swap(int v[], int k);
  { int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
  }
  \end{verbatim}

  swap:
  \[
  \begin{align*}
  &\text{mul }$2, $5, 4 \\
  &\text{add }$2, $4, $2 \\
  &\text{lw }$15, 0($2) \\
  &\text{lw }$16, 4($2) \\
  &\text{sw }$16, 0($2) \\
  &\text{sw }$15, 4($2) \\
  &\text{jr }$31
  \end{align*}
  \]
So far we’ve learned:

- MIPS
  - loading words but addressing bytes
  - arithmetic on registers only

- Instruction | Meaning
  - `add $s1, $s2, $s3` | `$s1 = $s2 + $s3`  
  - `sub $s1, $s2, $s3` | `$s1 = $s2 - $s3`  
  - `lw $s1, 100($s2)` | `$s1 = Memory[\text{s2}+100]`  
  - `sw $s1, 100($s2)` | `Memory[\text{s2}+100] = $s1`  

• Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t0, $s1, $s2`  
    - registers have numbers, $t0=9$, $s1=17$, $s2=18$

- Instruction 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>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

- Can you guess what the field names stand for?
Machine Language

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise

- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - Other format was R-type for register

- Example: \( \text{lw} \; \$t0, 32(\$s2) \)

\[
\begin{array}{cccc}
35 & 18 & 9 & 32 \\
\end{array}
\]

- Where's the compromise?

Stored Program Concept

- Instructions are bits
- Programs are stored in memory
  — to be read or written just like data

- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the "next" instruction and continue
• Decision making instructions
  – alter the control flow,
  – i.e., change the "next" instruction to be executed

• MIPS conditional branch instructions:
  \[ \text{bne $t0, t1, Label} \]
  \[ \text{beq $t0, t1, Label} \]

• Example: \( \text{if (i==j) } h = i + j; \)
  \[ \text{bne } s0, s1, Label \]
  \[ \text{add } s3, s0, s1 \]
  \[ \text{Label: ...} \]

• MIPS unconditional branch instructions:
  \[ \text{j label} \]

• Example:
  \[ \text{if (i!=j) } \]
  \[ \text{beq } s4, s5, Lab1 \]
  \[ \text{add } s3, s4, s5 \]
  \[ \text{else } \]
  \[ \text{j Lab2} \]
  \[ \text{sub } s3, s4, s5 \]
  \[ \text{Lab1: ...} \]
  \[ \text{Lab2: ...} \]

• Can you build a simple for loop?
So far:

- **Instruction** | **Meaning**
  - `add $s1,$s2,$s3` | $s1 = $s2 + $s3$
  - `sub $s1,$s2,$s3` | $s1 = $s2 - $s3$
  - `lw $s1,100($s2)` | $s1 = \text{Memory}[\$s2+100]$
  - `sw $s1,100($s2)` | \text{Memory}[\$s2+100] = $s1$
  - `bne $s4,$s5,L` | Next instr. is at Label if $s4 \neq s5$
  - `beq $s4,$s5,L` | Next instr. is at Label if $s4 = s5$
  - `j Label` | Next instr. is at Label

- **Formats:**
  - **R**
    - `op rs rt rd shamt funct`
  - **I**
    - `op rs rt 16\text{ bit address}`
  - **J**
    - `op 26\text{ bit address}`

Control Flow

- We have: beq, bne, what about Branch-if-less-than?

- New instruction:
  ```
  if $s1 < $s2 then
  $t0 = 1$
  else
  $t0 = 0$
  ```

- Can use this instruction to build "blt $s1, $s2, Label"
  — can now build general control structures

- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers
Policy of Use 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>

Constants

- Small constants are used quite frequently (50% of operands)
  e.g.,
  \[ A = A + 5; \]
  \[ B = B + 1; \]
  \[ C = C - 18; \]
- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.
- MIPS Instructions:
  - `addi $29, $29, 4`
  - `slti $8, $18, 10`
  - `andi $29, $29, 6`
  - `ori $29, $29, 4`
- How do we make this work?
How about larger constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new “load upper immediate” instruction
  \[
  \text{li } \$t0, \ 1010101010101010
  \]
  filled with zeros

- Then must get the lower order bits right, i.e.,
  \[
  \text{ori } \$t0, \$t0, \ 1010101010101010
  \]

Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first

- Machine language is the underlying reality
  - e.g., destination is no longer first

- Assembly can provide ‘pseudoinstructions’
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”

- When considering performance you should count real instructions
Other Issues

- Things we are not going to cover
  - support for procedures
  - linkers, loaders, memory layout
  - stacks, frames, recursion
  - manipulating strings and pointers
  - interrupts and exceptions
  - system calls and conventions

- Some of these we'll talk about later

- We've focused on architectural issues
  - basics of MIPS assembly language and machine code
  - we'll build a processor to execute these instructions.

Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

<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>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- rely on compiler to achieve performance
  - what are the compiler’s goals?
- help compiler where we can
Addresses in Branches and Jumps

- **Instructions:**
  - `bne $t4,$t5,Label` \(\text{Next instruction is at Label if } t4 \neq t5\)
  - `beq $t4,$t5,Label` \(\text{Next instruction is at Label if } t4 = t5\)
  - `j Label` \(\text{Next instruction is at Label}\)

- **Formats:**
  - | \(I\) | op | rs | rt | 16 bit address |
  - | \(J\) | op | 26 bit address |

- **Addresses are not 32 bits**
  - How do we handle this with load and store instructions?

Addresses in Branches

- **Instructions:**
  - `bne $t4,$t5,Label` \(\text{Next instruction is at Label if } t4 \neq t5\)
  - `beq $t4,$t5,Label` \(\text{Next instruction is at Label if } t4 = t5\)

- **Formats:**
  - | \(I\) | op | rs | rt | 16 bit address |

- **Could specify a register (like lw and sw) and add it to address**
  - use Instruction Address Register (PC = program counter)
  - most branches are local (principle of locality)

- **Jump instructions just use high order bits of PC**
  - address boundaries of 256 MB
To summarize:

<table>
<thead>
<tr>
<th>MIPS operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>32 registers</strong></td>
</tr>
<tr>
<td>$s0-$s7, $t0-$t9, $zero, $at, $fp, $sp, $ra, $s8, $s9, $t8, $t9, $zero, $at</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td>$a0-$a3, $v0-$v1, $gp, $at, $fp, $sp, $ra, $s8, $s9, $t8, $t9, $zero, $at</td>
</tr>
</tbody>
</table>

- Fast locations for data; in MIPS, data must be in registers to perform arithmetic: MIPS register form always equals 0. Register $at is reserved for the assembler to handle large constants.

- Access only by data transfer instructions; MIPS uses byte addresses, so unaligned words differ by 2. Memory holds data structures, such as arrays, unaligned data such as extra space at procedure call.

---

**MIPS assembly language**

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic</strong></td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>addi $s1, $s2, 100</td>
<td>$s1 = $s2 + 100</td>
<td>Used to add constants</td>
</tr>
<tr>
<td></td>
<td>load word</td>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Only word accessible &amp; memory</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Word from register &amp; memory</td>
</tr>
<tr>
<td></td>
<td>load immediate</td>
<td>lui $s1, 100</td>
<td>$s1 = 100 * 2^16</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td><strong>Conditional branch</strong></td>
<td>branch on equal</td>
<td>beq $s1, $s2, 25</td>
<td>if ($s1 == $s2) go to PC + 4 + 100</td>
<td>Equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1, $s2, 25</td>
<td>if ($s1 != $s2) go to PC + 4 + 100</td>
<td>Not equal test; PC-relative</td>
</tr>
<tr>
<td></td>
<td>branch less than</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than; for beq, bne</td>
</tr>
<tr>
<td></td>
<td>jump</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>Jump to target address</td>
</tr>
<tr>
<td><strong>Unconditional jump</strong></td>
<td>jump and link</td>
<td>jal 2500</td>
<td>$ra = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>

---

1. Immediate addressing
2. Register addressing
3. Base addressing
4. PC-relative addressing
5. Pseudodirect addressing
Alternative Architectures

• Design alternative:
  – provide more powerful operations
  – goal is to reduce number of instructions executed
  – danger is a slower cycle time and/or a higher CPI

• Sometimes referred to as “RISC vs. CISC”
  – virtually all new instruction sets since 1982 have been RISC
  – VAX: minimize code size, make assembly language easy
    
    instructions from 1 to 54 bytes long!

• We’ll look at PowerPC and 80x86

PowerPC

• Indexed addressing
  – example: \texttt{lw \$t1,\$a0+\$s3} # \$t1=Memory[\$a0+\$s3]
  – What do we have to do in MIPS?

• Update addressing
  – update a register as part of load (for marching through arrays)
  – example: \texttt{lwu \$t0,4(\$s3)} #\$t0=Memory[\$s3+4];\$s3=\$s3+4
  – What do we have to do in MIPS?

• Others:
  – load multiple/store multiple
  – a special counter register “\texttt{bc Loop}”
    
    decrement counter, if not 0 goto loop
80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: MMX is added

“This history illustrates the impact of the “golden handcuffs” of compatibility

“adding new features as someone might add clothing to a packed bag”

“an architecture that is difficult to explain and impossible to love”

A dominant architecture: 80x86

- See your textbook for a more detailed description

- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”

- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

“what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective”
Summary

• Instruction complexity is only one variable
  – lower instruction count vs. higher CPI / lower clock rate

• Design Principles:
  – simplicity favors regularity
  – smaller is faster
  – good design demands compromise
  – make the common case fast

• Instruction set architecture
  – a very important abstraction indeed!