Control

• Decision making instructions
  – alter the control flow,
  – i.e., change the "next" instruction to be executed

• MIPS conditional branch instructions:
  
  ```
  bne $t0, $t1, Label
  beq $t0, $t1, Label
  ```

• Example: if (i==j) h = i + j;
  
  ```
  bne $s0, $s1, Label
  add $s3, $s0, $s1
  Label: ....
  ```

Control

• MIPS unconditional branch instructions:
  
  ```
  j label
  ```

• Example:
  
  ```
  if (i!=j) beq $s4, $s5, Lab1
  h=i+j; add $s3, $s4, $s5
  else
  j Lab2
  h=i-j; Lab1: sub $s3, $s4, $s5
  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)`  $s1 = Memory[$s2+100]
  - `sw $s1,100($s2)`  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 bit address |
  - **J**
    | op | 26 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
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

```
  lui $t0, 1010101010101010
  ```

- Then must get the lower order bits right, i.e.,

```
  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>R</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>16 bit address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td>---</td>
<td>26 bit address</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  
  beq $t4,$t5,Label  
  j Label

  Next instruction is at Label if $t4 \neq $t5
  Next instruction is at Label if $t4 = $t5
  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, $t5, Label` Next instruction is at `Label` if `$t4 ≠ $t5`
  - `beq $t4, $t5, Label` Next instruction is at `Label` if `$t4 =$t5`

- Formats:

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

- 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:

### MIPS operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 register</td>
<td><code>$s0-$s7</code>, <code>$t0-$t9</code>, <code>$zero</code>, <code>$a0-$a3</code>, <code>$v0-$v1</code>, <code>$gp</code>, <code>$fp</code>, <code>$sp</code>, <code>$ra</code>, <code>$at</code></td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register file is large enough to hold data temporarily before it is moved to the memory to handle data permanently.</td>
</tr>
<tr>
<td>Memory[0]</td>
<td><code>$fp</code>, <code>$sp</code>, <code>$ra</code>, <code>$at</code></td>
<td>Reserved for the assembler to handle large constants.</td>
</tr>
</tbody>
</table>

### 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>Arithmetic</td>
<td>add</td>
<td><code>$s1</code>, <code>$s2</code>, <code>$s3</code></td>
<td><code>$s1 = $s2 + $s3</code></td>
<td>Three operands, data in registers.</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td><code>$s1</code>, <code>$s2</code>, <code>$s3</code></td>
<td><code>$s1 = $s2 - $s3</code></td>
<td>Three operands, data in registers.</td>
</tr>
<tr>
<td>Datamem</td>
<td>lw</td>
<td><code>$s1</code>, <code>100($s2)</code></td>
<td><code>$s1 = Memory[0x100 + $s2]</code> Word from memory to register.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td><code>$s1</code>, <code>100($s2)</code></td>
<td><code>$s1 = Memory[0x100 + $s2]</code> Word from register to memory.</td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>lui</td>
<td><code>$s1</code>, <code>100</code></td>
<td><code>$s1 = 100 * 2^{16}</code> Loads constant in upper 16 bits.</td>
<td></td>
</tr>
<tr>
<td>Conditional Branch</td>
<td>bne</td>
<td><code>$s1$, </code>$s2<code>, </code>25`</td>
<td>If <code>$s1 ≠ $s2</code> go to PC + 4 + 100 Code relative branch.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>beq</td>
<td><code>$s1$, </code>$s2<code>, </code>25`</td>
<td>If <code>$s1 =$s2</code> go to PC + 4 + 100 Code relative branch.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>slt</td>
<td><code>$s1</code>, <code>$s2</code>, <code>$s3</code></td>
<td><code>$s1</code> = 1; else 0 Compare less than; for <code>beq</code>, <code>bne</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>slti</td>
<td><code>$s1</code>, <code>$s2</code>, <code>100</code></td>
<td><code>$s1</code> = 1; else 0 Compare less than constant</td>
<td></td>
</tr>
<tr>
<td></td>
<td>j</td>
<td><code>2500</code></td>
<td>Jump to target address</td>
<td></td>
</tr>
<tr>
<td></td>
<td>jal</td>
<td><code>2500</code></td>
<td><code>$ra = PC + 4; go to 10000</code> For procedure call</td>
<td></td>
</tr>
</tbody>
</table>

©1998 Morgan Kaufmann Publishers

1. Immediate addressing

| Op | Rn | Rl | Immediate |

2. Register addressing

| Op | Rn | Rl | Register |

3. Base addressing

| Op | Rn | Rl | Address |

4. PC-relative addressing

| Op | Rn | Rl | Address |

5. Pseudodirect addressing

| Op | Address |

---

**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=\text{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=\text{Memory}[\$s3+4]; \$s3=\$s3+4
  - What do we have to do in MIPS?

- Others:
  - load multiple/store multiple
  - a special counter register “bc Loop”
    \begin{verbatim}
    decrement counter, if not 0 goto loop
    \end{verbatim}

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”