## MIPS Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1,$s2,$s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1,$s2,$s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>addi $s1,$s2,4</td>
<td>$s1 = $s2 + 4</td>
</tr>
<tr>
<td>ori $s1,$s2,4</td>
<td>$s2 = $s2</td>
</tr>
<tr>
<td>lw $s1,100($s2)</td>
<td>Memory[$s2+100]</td>
</tr>
<tr>
<td>sw $s1,100($s2)</td>
<td>Memory[$s2+100] = $s1</td>
</tr>
<tr>
<td>bne $s4,$s5,Label</td>
<td>Next instr. is at Label if $s4 \neq $s5</td>
</tr>
<tr>
<td>beq $s4,$s5,Label</td>
<td>Next instr. is at Label if $s4 = $s5</td>
</tr>
<tr>
<td>slt $t1,$s2,$s3</td>
<td>if $s2 &lt; $s3, $t1 = 1 else $t1 = 0</td>
</tr>
<tr>
<td>j Label</td>
<td>Next instr. is at Label</td>
</tr>
<tr>
<td>jr $s1</td>
<td>Next instr is in register $s1</td>
</tr>
<tr>
<td>jal Label</td>
<td>Jump and link procedure at Label</td>
</tr>
</tbody>
</table>
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”

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
Instruction Set Architecture: What Must be Specified?

- Instruction Format or Encoding
  - how is it decoded?
- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data type and Size
- Operations
  - what are supported
- Successor instruction
  - jumps, conditions, branches

- fetch-decode-execute is implicit!

MIPS Design Principles

- Reduced Instruction Set Computers (RISC) design philosophy
- Principles guiding Instruction Set Design
  - Smaller is faster
    - Example: Only 32 registers in MIPS
  - Simplicity favors regularity
  - Good design demands compromise
  - Make the common case fast
Machine Language: R Format

- 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>000000</th>
<th>1001</th>
<th>10010</th>
<th>01000</th>
<th>00000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
</tbody>
</table>

- R Format

Machine Language: I format

- 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: lw $t0, 32($s2)

<table>
<thead>
<tr>
<th>000000</th>
<th>1001</th>
<th>10010</th>
<th>01000</th>
<th>00000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>16 bit number</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Where’s the compromise?
Machine Language: J Format

- Jump (j) and Jump and link (jal) instructions have two fields
  - Opcode
  - Address
- Instruction should be 32 bits (Regularity principle)
  - 6 bits for opcode
  - 26 bits for address

```
J  op  26 bit address
```

MIPS Instruction Formats

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

```
R  op  rs  rt  rd  shamt  funct
I  op  rs  rt  16 bit address
J  op  26 bit address
```
What about other instructions

- `slt $t0, $s1, $s2`
  - 3 operands all registers ⇒ use R format
- `beq $s1,$s2, Label`
  - 2 registers + address ⇒ use I format
- `addi $s1,$s1, 4`
  - 2 registers + immediate value ⇒ use I format
- `jr $t1`
  - 1 register
  - R format

Implications of design choices

- Using I format for arithmetic instructions with immediate operands
  - Only 16 bits for immediate field
  - Constants have to fit in 16 bits
- Using I format for branch instructions
  - Only 16 bits in immediate field
  - But 32 bits needed for branch address
- J format
  - Only 26 bits for address field
  - But 32 bits needed for Jump address
Constants

- Small constants are used quite frequently (50% of operands)
  e.g., \( A = A + 5; \)
  \( B = B + 1; \)
  \( C = C - 18; \)
- So in most programs, constants will fit in 16 bits allocated for immediate field
- Design Principle: Make the common case fast
  - Common case: constant is small
  - Only need to use one instruction in the common case

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
  
  \[
  \text{lui } \$t0, 1010101010101010 \\
  \text{ori } \$t0, \$t0, 1010101010101010
  \]

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

\[
\begin{array}{c}
1010101010101010 \\
0000000000000000
\end{array}
\]

\[
\begin{array}{c}
0000000000000000 \\
1010101010101010
\end{array}
\]

\[
\begin{array}{c}
1010101010101010 \\
1010101010101010
\end{array}
\]
Addresses in Branches and Jumps

- Instructions:
  
  bne $t4,$t5,Label  \hspace{1cm} \text{Next instruction is at Label if } \$t4 \neq \$t5
  
  beq $t4,$t5,Label  \hspace{1cm} \text{Next instruction is at Label if } \$t4 = \$t5
  
  j Label  \hspace{1cm} \text{Next instruction is at Label}

- Formats:
  
<table>
<thead>
<tr>
<th>I</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit address</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>op</td>
<td>26 bit address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Addresses in Branches

- Instructions:
  
  bne $t4,$t5,Label  \hspace{1cm} \text{Next instruction is at Label if } \$t4 \neq \$t5
  
  beq $t4,$t5,Label  \hspace{1cm} \text{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)
  
- Jump instructions just use high order bits of PC
  
  - address boundaries of 256 MB
Addressing in branches

- Immediate field is 16 bits but we need an address that is 32 bits
- Obtain address using **PC-relative addressing**
  - On branch, new PC = PC + immediate field in branch instruction
  - Actually, new PC = (PC+4) + immediate field in branch instruction

80000  Loop:  mult $9, $19, $10
80004  lw   $8, Sstart($9)
80008  bne  $8, $21, Exit
80012  add  $19, $19, $20
80016  j   Loop
80020  Exit:

Immediate field contains the distance in words between PC+4 and branch target address

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>21</td>
<td>2</td>
</tr>
</tbody>
</table>

Uncommon Case for branches

- beq $18, $19, L1
  replaced by

  bne $18, $19, L2
  j   L1

L2:

Make the common case fast
one instruction for most branches
Addressing in Jumps

- J format has 26 bits in address field
  - How to get 32 bits?
- Assume that jump address is a word address
- 26 + 2 (least significant bits) = 28
- Get 4 most significant bits from PC
  - \( 4 + 26 + 2 = 32 \)
  - Implication: can only jump within a \( 2^{28} = 256 \text{ MB} \) block of addresses
- Loader and linker must be careful to avoid placing a program across an address boundary 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 registers</td>
<td>$s0$-$s7$, $t0$-$t9$, $zero$</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero$ always equals 0.</td>
</tr>
<tr>
<td>16 additional words</td>
<td>$a0$-$a3$, $v0$-$v1$, $gp$, $fp$, $sp$, $ra$, $at$</td>
<td>Registered for the assembler to handle large constants.</td>
</tr>
<tr>
<td>Memory [0]</td>
<td>$a0$-$a3$, $v0$-$v1$, $gp$, $fp$, $sp$, $ra$, $at$</td>
<td>Reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>Memory [4]</td>
<td>...</td>
<td>Sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.</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>$add$ $s1$, $s2$, $s3$</td>
<td>$s1 = s2 + s3$</td>
<td>Three operands, data in registers</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>$lw$ $s1$, ( 100(s2) )</td>
<td>$s1 = Memory[s2 + 100]$</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td>Data transfer</td>
<td>store word</td>
<td>$sw$ $s1$, ( 100(s2) )</td>
<td>Memory[s2 + 100] = $s1$</td>
<td>Word from register to memory</td>
</tr>
<tr>
<td>Conditional branch</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>Conditional branch</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>Conditional branch</td>
<td>branch on less than</td>
<td>$slt$ $s1$, $s2$, ( 100(s3) )</td>
<td>if ((s1 &lt; s2)) go to (PC + 4 + 100)</td>
<td>Compare less than; PC-relative</td>
</tr>
</tbody>
</table>

©1998 Morgan Kaufmann Publishers
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: 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: 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 “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

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