### 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>$s1 = Memory[$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 ≠ $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
Instructions, like registers and words of data, are also 32 bits long
- Example: \texttt{add $t0, $s1, $s2}
- registers have numbers, $t0=9$, $s1=17$, $s2=18$

Instruction Format:

\begin{center}
\begin{tabular}{cccccc}
op & rs & rt & rd & shamt & funct \\
000000 & 1001 & 10010 & 01000 & 00000 & 100000 \\
\end{tabular}
\end{center}

R 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: \texttt{lw $t0, 32($s2)}

\begin{center}
\begin{tabular}{cccc}
35 & 18 & 9 & 32 \\
\hline
op & rs & rt & 16 bit number \\
\end{tabular}
\end{center}

Where’s the compromise?
Machine Language: J Format

- Jump (j), 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

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

  filled with zeros

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

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

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

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

  \[
  1010101010101010 \\
  1010101010101010 \\
  \]

Addresses in Branches and Jumps

- Instructions:
  - `bne $t4,$t5,Label`  
  - Next instruction is at Label if $t4 \neq t5$
  - `beq $t4,$t5,Label`  
  - Next instruction is at Label if $t4 = t5$
  - `j Label`  
  - Next instruction is at Label

- Formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit address</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td>26 bit address</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`  
  - Next instruction is at Label if $t4 \neq t5$
  - `beq $t4,$t5,Label`  
  - Next instruction is at Label if $t4 = t5$

- Formats:

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

©1998 Morgan Kaufmann Publishers
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

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 $2^{28} = 256$ MB block of addresses
  - Loader and linker must be careful to avoid placing a program across an address boundary of 256 MB

To summarize:

<table>
<thead>
<tr>
<th>MIPS operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>32 register</td>
</tr>
<tr>
<td>8th memory word</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MIPS assembly language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
</tr>
<tr>
<td>Arithmetic</td>
</tr>
<tr>
<td>Subtrahend</td>
</tr>
<tr>
<td>Add immediate</td>
</tr>
<tr>
<td>Load word</td>
</tr>
<tr>
<td>Store word</td>
</tr>
<tr>
<td>Load byte</td>
</tr>
<tr>
<td>Store byte</td>
</tr>
<tr>
<td>Add upper immediate</td>
</tr>
<tr>
<td>Conditional branch</td>
</tr>
<tr>
<td>Branch on not equal</td>
</tr>
<tr>
<td>Branch on less than</td>
</tr>
<tr>
<td>Branch on less than or equal</td>
</tr>
<tr>
<td>Jump</td>
</tr>
<tr>
<td>Jump register</td>
</tr>
<tr>
<td>Jump and link</td>
</tr>
</tbody>
</table>
Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower CPU
  - 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 IA-32
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

IA - 32

- 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: 57 new “MMX” instructions are added, Pentium II
- 1999: The Pentium III added another 70 instructions (SSE)
- 2001: Another 144 instructions (SSE2)
- 2003: AMD extends the architecture to increase address space to 64 bits,
  widens all registers to 64 bits and other changes (AMD64)
- 2004: Intel capitulates and embraces AMD64 (calls it EM64T) and adds
  more media extensions

- “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”
IA-32 Overview

- 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”

IA-32 Registers and Data Addressing

- Registers in the 32-bit subset that originated with 80386

<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 1</td>
</tr>
<tr>
<td>ES</td>
<td>Data segment pointer 2</td>
</tr>
<tr>
<td>FS</td>
<td>Data segment pointer 3</td>
</tr>
<tr>
<td>GS</td>
<td>Data segment pointer 1</td>
</tr>
<tr>
<td>EIP</td>
<td>Instruction pointer (PC)</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>Condition codes</td>
</tr>
</tbody>
</table>
### IA-32 Register Restrictions

- Registers are not "general purpose" – note the restrictions below

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Instruction Set Restrictions</th>
<th>MIPS equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register indirect</td>
<td>ADD/ADDI is 32-bit register</td>
<td>ADD/ADDI</td>
<td>bx ($R0,0xFF)</td>
</tr>
<tr>
<td>Global mode with r or R6-8</td>
<td>Address in contents of base register plus</td>
<td>ADD/ADDI</td>
<td>bx + ($R0,0xFF)</td>
</tr>
<tr>
<td>displacement</td>
<td>displacement</td>
<td></td>
<td>dx + ($R0,0xFF)</td>
</tr>
<tr>
<td>Data plus scaled index</td>
<td>The address is</td>
<td>ADD/ADDI</td>
<td>bx + ($R0,0xFF)</td>
</tr>
<tr>
<td></td>
<td>base + (scale * index)</td>
<td></td>
<td>dx + ($R0,0xFF)</td>
</tr>
<tr>
<td></td>
<td>where scale has the values 0, 1, 2, or 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data plus immediate with</td>
<td>base + (immediate)</td>
<td>ADD/ADDI</td>
<td>bx + ($R0,0xFF)</td>
</tr>
<tr>
<td>8- or 32-bit displacement</td>
<td>displacement</td>
<td></td>
<td>dx + ($R0,0xFF)</td>
</tr>
<tr>
<td></td>
<td>where displacement has the values 0, 1, 2, or 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 2.42** IA-32 32-bit addressing modes with register restrictions and the emulate MIPS code. The emulated code inside a following code is in MIPS format, as it would exceed the multiple by first mode 1 of 10 to turn into an index to a register into a byte of data (see Figure 2.34 and 2.35). In a byte of 8 bits, the first bit is the sign bit, and is called a bit of the end bit data that has 8 or else the address is not scaled. If the sign bit is larger than 1 bit in the second or fourth modes, then the MIPS equivalent code would need two more instructions, b to load the upper 16 bits of the displacement and not 0 to read the upper address with the byte register 1-3, second given two different meanings for what is called an addressing mode—base and index—but they are essentially identical and we consider them here.

### IA-32 Typical Instructions

- Four major types of integer instructions:
  - Data movement including move, push, pop
  - Arithmetic and logical (destination register or memory)
  - Control flow (use of condition codes / flags)
  - String instructions, including string move and string compare

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD, ADDI</td>
<td>32-bit addition</td>
</tr>
<tr>
<td>SUB, SUBI</td>
<td>32-bit subtraction</td>
</tr>
<tr>
<td>MUL</td>
<td>32-bit multiplication</td>
</tr>
<tr>
<td>IDIV</td>
<td>32-bit division</td>
</tr>
<tr>
<td>DIV</td>
<td>32-bit division</td>
</tr>
<tr>
<td>SHR</td>
<td>32-bit shift</td>
</tr>
<tr>
<td>SHL</td>
<td>32-bit shift</td>
</tr>
<tr>
<td>SRL</td>
<td>32-bit shift</td>
</tr>
<tr>
<td>SAR</td>
<td>32-bit shift</td>
</tr>
<tr>
<td>ADDI</td>
<td>Immediate addition</td>
</tr>
<tr>
<td>SUBI</td>
<td>Immediate subtraction</td>
</tr>
<tr>
<td>MULI</td>
<td>Immediate multiplication</td>
</tr>
<tr>
<td>IDIVI</td>
<td>Immediate division</td>
</tr>
<tr>
<td>DIVI</td>
<td>Immediate division</td>
</tr>
<tr>
<td>SHRi</td>
<td>Immediate shift</td>
</tr>
<tr>
<td>SHLi</td>
<td>Immediate shift</td>
</tr>
<tr>
<td>SAR</td>
<td>Immediate shift</td>
</tr>
<tr>
<td>SRL</td>
<td>Immediate shift</td>
</tr>
</tbody>
</table>

**FIGURE 2.43** Some typical IA-32 instructions and their functions. A list of special operations appears in Figure 2.44. The CALL saves the ESP of the next instruction on the stack (ESP in the local PC).
IA-32 instruction Formats

- Typical formats: (notice the different lengths)

  a. JE EIP + displacement
     
     |   Je |  4  |  Displacement |
     |-----|----|--------------|

  b. CALL
     
     |   CALL |  8 |  Offset |

  c. MOV EBX, [EDI + 45]
     
     |   MOV  |  8  | Displacement |
     |        |    |              |

  d. PUSH EDX
     
     |   PUSH  |  5  |  EAX |

  e. ADD EAX, [EBX]
     
     |   ADD   |  4  |  ImmAdd |

  f. TEST EDX, #100
     
     |   TEST  |  2  |  Postbyte |  ImmAdd |

Summary

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