Chapter 2
Instructions:

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

“The natural number of operands for an operation like addition is three...requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple”
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:  
\text{add} $t0, $s1, $s2
\text{add} $s0, $t0, $s3
\text{sub} $s4, $s5, $s0
MIPS arithmetic

- 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

Diagram:

- Control
- Datapath
- Memory
- Input
- Output
  - Processor
  - I/O
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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>1</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>2</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>3</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>4</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>5</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>6</td>
<td>8 bits of data</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.

<table>
<thead>
<tr>
<th></th>
<th>32 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

... Registers hold 32 bits of data
Memory Organization

- $2^{32}$ 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:  
  \[
  \text{lw } \$t0, 32(\$s3) \\
  \text{add } \$t0, \$s2, \$t0 \\
  \text{sw } \$t0, 32(\$s3)
  \]

- Store word has destination last

- Remember arithmetic operands are registers, not memory!

Can’t write:  
\[
\text{add } 48(\$s3), \$s2, 32(\$s3)
\]
Our First Example

Can we figure out the code?

```c
swap(int v[], int k);
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```assembly
muli $2, $5, 4
add $2, $4, $2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
jr $31
```
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) | $s1 = Memory[+$s2+100]
sw $s1, 100($s2) | Memory[+$s2+100] = $s1
Machine Language

- 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></th>
<th>000000</th>
<th>10001</th>
<th>10010</th>
<th>01001</th>
<th>00000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shamt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Can you guess what the field names stand for?
Consider the load-word and store-word instructions,

- What would the regularity principle have us do?

- New principle: Good design demands a compromise
Machine Language

- 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)}

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>18</td>
<td>9</td>
<td>32</td>
</tr>
</tbody>
</table>

- Where's the compromise?
Instructions are bits

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

memory for data, programs, compilers, editors, etc.
Stored Program Concept

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

- **Formats:**

<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>R</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
</tr>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>16 bit number</td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>26 bit address</td>
</tr>
</tbody>
</table>

Electrical & Computer Engineering
School of Engineering
THE COLLEGE OF NEW JERSEY

Spring Semester, 2004
Control Flow

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

  \[
  \text{if } \quad \text{\textbf{if } } s1 < s2 \text{ then }
  \quad t0 = 1
  \]

  \[
  \text{\textbf{else }}
  \quad t0 = 0
  \]

  \[
  \text{\textbf{slt } } t0, s1, s2
  \]

- 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>
<th>Preserved on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results and expression evaluation</td>
<td>no</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
<td>yes</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>

Register 1 ($at) reserved for assembler, 26-27 for operating system
Supporting Procedures in Computer Hardware

High address

Low address

a.

b.

c.

$sp

Contents of register $s0

Contents of register $t1

Contents of register $t0

Contents of register $s0

Spring Semester, 2004

Electrical & Computer Engineering
School of Engineering
THE COLLEGE OF NEW JERSEY
Starting a Program

C program

Compiler

Assembly language program

Assembler

Object: Machine language module

Object: Library routine (machine language)

Linker

Executable: Machine language program

Loader

Memory
Small constants are used quite frequently (50% of operands)

\[ A = A + 5; \]
\[ B = B + 1; \]
\[ C = C - 18; \]

Solutions? Why not?

- put 'typical constants' in memory and load them.
- create hard-wired registers (like $zero) for constants like one.
Constants

- MIPS Instructions:

  addi $29, $29, 4
  slti $8, $18, 10
  andi $29, $29, 6
  ori $29, $29, 4

- How do we make this work?

- Design Principle: Make the common case fast. Which format?
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:
  \[ \text{lui } \$t0, \quad 1010101010101010 \]
  filled with zeros

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

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

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

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

\[
\begin{array}{c|c}
1010101010101010 & 1010101010101010 \\
\end{array}
\]
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 Language vs. Machine Language

- 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

- Discussed in your assembly language programming labs:
  - 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 more about later
- We’ll talk about compiler optimizations when we hit chapter 4
Overview of MIPS

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

R

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

I

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

J

<table>
<thead>
<tr>
<th>op</th>
<th>26 bit address</th>
</tr>
</thead>
</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`  
  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>I</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>op</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 ≠ $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 number</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 registers</td>
<td>$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>2^30 memory words</td>
<td>Memory[0], Memory[4], ..., Memory[4294967292]</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so 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></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>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></td>
<td>store word</td>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Word from register memory</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>lb $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>sb $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Byte from register memory</td>
</tr>
<tr>
<td></td>
<td>load upper immediate</td>
<td>lui $s1, 100</td>
<td>$s1 = 100 * 2^16</td>
<td>Loads constant in upper 16 bits</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></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>set on 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>set less than immediate</td>
<td>slti $s1, $s2, 100</td>
<td>if ($s2 &lt; 100) $s1 = 1; else $s1 = 0</td>
<td>Compare less than constant</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>jump</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>Jump to target address</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>jr $ra</td>
<td>go to $ra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal 2500</td>
<td>$ra = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
1. Immediate addressing

```
| op | rs | rt | Immediate |
```

2. Register addressing

```
| op | rs | rt | rd | funct |
```

3. Base addressing

```
| op | rs | rt | Address |
```

4. PC-relative addressing

```
| op | rs | rt | Address |
```

5. Pseudodirect addressing

```
| op | Address |
| PC |
```

Electrical & Computer Engineering
School of Engineering
THE COLLEGE OF NEW JERSEY
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

"The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions"
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 Intel Architecture (IA)
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
Intel Architecture

- 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)
Intel Architecture

- 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
Intel Architecture

“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”
A dominant architecture: 80x86

- 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 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>
IA-32 Register Restrictions

- Registers are not “general purpose” – note the restrictions below

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Register restrictions</th>
<th>MIPS equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Indirect</td>
<td>Address is in a register.</td>
<td>not ESP or EBP</td>
<td>lw $s0, 0($s1)</td>
</tr>
<tr>
<td>Based mode with 6- or 32-bit displacement</td>
<td>Address is contents of base register plus displacement.</td>
<td>not ESP or EBP</td>
<td>lw $50, 100($s1) #16-bit displacement</td>
</tr>
<tr>
<td>Base plus scaled index</td>
<td>The address is Base + (2^{Index} x Index) where Scale has the value 0, 1, 2, or 3.</td>
<td>Base: any GPR Index: not ESP</td>
<td>mul $t0, $s2,4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>add $t0, $t0, $s1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lw $50, 0($t0)</td>
</tr>
<tr>
<td>Base plus scaled index with 8- or 32-bit displacement</td>
<td>The address is Base + (2^{Index} x Index) + displacement where Scale has the value 0, 1, 2, or 3.</td>
<td>Base: any GPR Index: not ESP</td>
<td>mul $t0, $s2,4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>add $t0, $t0, $s1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lw $50, 100($t0) #16-bit displacement</td>
</tr>
</tbody>
</table>

**FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code.** The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.24 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine 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>JE name</td>
<td>if equal(condition code) {EIP=name}; EIP-128 ≤ name ≤ EIP+128</td>
</tr>
<tr>
<td>JMP name</td>
<td>EIP=name</td>
</tr>
<tr>
<td>CALL name</td>
<td>SP=SP-4; M[SP]=EIP+5; EIP=name;</td>
</tr>
<tr>
<td>MOV EBX,[EDI+45]</td>
<td>EBX=M[EDI+45]</td>
</tr>
<tr>
<td>PUSH ESI</td>
<td>SP=SP-4; M[SP]=ESI</td>
</tr>
<tr>
<td>POP EDI</td>
<td>EDI=M[SP]; SP=SP+4</td>
</tr>
<tr>
<td>ADD EAX,#6765</td>
<td>EAX=EAX+6765</td>
</tr>
<tr>
<td>TEST EDX,#42</td>
<td>Set condition code (flags) with EDX and 42</td>
</tr>
<tr>
<td>MOVS L</td>
<td>M[EDI]=M[ESI]; EDI-EDI+4; ES=ESI+4</td>
</tr>
</tbody>
</table>

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

- **Typical formats:**
  (notice the different lengths)

a. JE EIP + displacement

<table>
<thead>
<tr>
<th>JE</th>
<th>Condition</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

b. CALL

<table>
<thead>
<tr>
<th>CALL</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

c. MOV EBX, [EDI + 45]

<table>
<thead>
<tr>
<th>MOV</th>
<th>d</th>
<th>w</th>
<th>r/m</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

d. PUSH ESI

<table>
<thead>
<tr>
<th>PUSH</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

e. ADD EAX, #6765

<table>
<thead>
<tr>
<th>ADD</th>
<th>Reg</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

f. TEST EDX, #42

<table>
<thead>
<tr>
<th>TEST</th>
<th>w</th>
<th>Postbyte</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>
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!