# MIPS Instruction Formats

- **for instance, “add r1, r2, r3” has**
  - 000000 00010 00011 00001 00000 100000
- **opcode (OP) tells the machine which format**
VAX Instruction Formats (x86 similar)

1 Byte OP code
- specifies number, type and length of operands

Each operand is 1 to many bytes
- first byte specifies addressing mode

“move” can be load, store, mem copy, jump,... depending on operands
How Many Operands?

• **Two-address code**: target is same as one operand
  - E.g., \( x = x + y \)

• **Three-address code**: target can be different
  - E.g. \( x = y + z \)
  - x86 doesn’t have three-address instructions; others do

• Some operands are also specified implicitly
  - “condition code” setting shows if result was +, 0, or –
  - PowerPC’s “Branch on count” uses special “count register”

• Well-known ISA’s have 0-4 (explicit) operands
  - PowerPC has “float multiply add”, \( r = x + y \times z \)
Addressing Modes

how do we specify the operand we want?

Immediate

#25
The operand (25) is part of the instruction

Register (direct)

R3
(sometimes written $3)
The operand is the contents of register 3

Register (indirect)

M[R3]
Use contents of R3 as address into memory; find the operand there. This is a special case of...

Base+Displacement

M[R3 + 160]
Add the displacement (160) to contents of R3, look in that memory location for the operand
- If register is PC, this is “PC-relative addressing”

All our example ISA’s have the above modes
# More Addressing Modes

*(not included in MIPS ISA)*

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base+Index</strong></td>
<td>$M[R3 + R4]$</td>
</tr>
<tr>
<td></td>
<td>Add contents of R3 and R4 to get memory address</td>
</tr>
<tr>
<td><strong>Autoincrement</strong></td>
<td>$M[R3++]$ (or $M[R3+=d]$)</td>
</tr>
<tr>
<td></td>
<td>Find value in memory location designated by R3, but also increment R3</td>
</tr>
<tr>
<td></td>
<td>- Useful for accessing array elements</td>
</tr>
<tr>
<td></td>
<td>- Autodecrement is similar</td>
</tr>
<tr>
<td><strong>Scaled Index</strong></td>
<td>$M[R3 + R4*d]$ (VAX and x86)</td>
</tr>
<tr>
<td></td>
<td>Multiply R4 by d (d is typically 1, 2, 4, or 8), then add R3 to get memory</td>
</tr>
<tr>
<td></td>
<td>address</td>
</tr>
<tr>
<td><strong>Memory Direct (absolute)</strong></td>
<td>$M[10000]$</td>
</tr>
<tr>
<td><strong>Memory Indirect</strong></td>
<td>$M[M[R3]]$ (only VAX)</td>
</tr>
<tr>
<td></td>
<td>Find number in memory location R3, use THAT as address into memory to find</td>
</tr>
</tbody>
</table>
VAX addressing mode usage

- Half of all references were register-mode.
- Remaining half distributed as follows:

<table>
<thead>
<tr>
<th>Program</th>
<th>Base + Displacement</th>
<th>Immediate</th>
<th>Scaled Index</th>
<th>Memory Indirect</th>
<th>All Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEX</td>
<td>56%</td>
<td>43%</td>
<td>0</td>
<td>1%</td>
<td>0</td>
</tr>
<tr>
<td>Spice</td>
<td>58%</td>
<td>17%</td>
<td>16%</td>
<td>6%</td>
<td>3%</td>
</tr>
<tr>
<td>GCC</td>
<td>51%</td>
<td>39%</td>
<td>6%</td>
<td>1%</td>
<td>3%</td>
</tr>
</tbody>
</table>

- Similar measurements show that 16 bits is enough for the immediate field 75% to 80% of the time.
- And 16 bits is enough for displacement 99% of the time.

Experiments on VAX binaries led to formulation of addressing modes in MIPS… unused things were dropped.
**MIPS addressing modes**

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

add $1, $2, $3

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

addi $1, $2, 35

lw $1, 24($2)  
\[ \text{register indirect} \  
\Rightarrow \text{disp} = 0 \  
\text{absolute} \  
\Rightarrow (rs) = 0 \]
MIPS ISA decisions

instruction length
  - all instructions are 32 bits long (one word)

how many registers?
  - 32 general purpose registers (R0 always 0)
  - (2 special purpose for multiply and divide)

where do operands reside?
  - load-store architecture.

instruction formats
  - three (r-, i-, and j-format).

operands
  - 3-address code.
  - immediate, register, and base+displacement modes.
Intel x86 evolution

- 1978: Intel 8086 announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor added
- 1982: The 80286 - more ops, 24-bit address space
- 1985: The 80386 - 32-bit address space + new modes
- 1989-1995: The 80486, Pentium, and Pentium Pro add a few instructions
- 1997: MMX is added (Pentium II is P. Pro + MMX)
- 1999 Pentium III (same architecture)
- 2000 Pentium 4 (144 new multimedia instructions)
- 2001 Itanium - new ISA (still can execute x86 code)
Anthropology of ISA’s

- VAX design goal was small code size and simple compilation (since all combinations of addressing modes were possible).
  - Met goals, but cheap & fast memories and better compilers made goals less important.
  - But couldn’t pipeline well. Replaced by Alpha (a RISC).
- x86: simpler than VAX, but still CISC
  - Ugly, but most common instructions can be implemented relatively efficiently.
- MIPS’ goal: simplicity, easy of decoding
  - 3% rule for adding new instructions
- PowerPC: RISC, Partitioned Machine, Superscalar. Some more complexity.
- IBM 801 “America” project - first RISC
Recent developments

- **VLIW** - Very Long Instruction Word
  - 1 "packet" has multiple instructions
  - Tera MTA has 26, 21, and 14 bit-long RISC operations (plus 3 "lookahead" bits) in 64 bits
  - Intel Itanium has three 41-bit RISC ops (plus 5 "type" bits) in 128-bit packet

- **JVM (Java Virtual Machine)**
  - a new level of abstraction
    - between Java language and ISA
  - stack based - is this a good choice??
MIPS ISA Tradeoffs

What if?
- 64 registers
- 20-bit immediates
- 4 operand instruction (e.g. \( Y = X + AB \))
I Type: Conditional Branches and
J Type: Unconditional Jumps

• How do you specify the destination of a branch/jump?
  - Studies show that almost all conditional branches go short distances from current PC
  • What program constructs cause conditional branches?
  - Specify a ____________ address in fewer bits than an ________________ address
I Type: Conditional Branches
beq $1, $2, 100

• How do you express this in MIPS?

• How does it change the PC?
But MIPS only supports 2-operands in branch for beq/bne!

• What if I want to emit code for
  - if ($R2 < $R3) goto somewhere;

• Vote:
  - A) Can't do it, too bad!
  - B) You must be able to do it, but I can't think how
  - C) You can, try using sub
  - D) You can, try using instruction ________
    (if you want to use another instruction, vote D, then yell out the answer when we review the poll)
beq/bne: if ($R2 < $R3)
Jumps, different from branches

- Not all changes in PC will be “close” to the current PC.
  - When?
    - Answer: jump to an ABSOLUTE address
    - Jump – j 10000
    - Jump and link – jal 10000
      - Used for procedure calls

- Jump Register (“JR”) (R type)
  - Used for returns / jump through pointer, very long jumps
Branch and Jump Addressing Modes

- Branch (e.g., beq) uses PC-relative addressing mode (uses few bits if address typically close). That is, target is PC+displacement mode.
  - If opcode is 6 bits, how many bits are available for displacement? How far can you jump?
Review - Instruction Execution in a CPU

Registers

<table>
<thead>
<tr>
<th>R0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>36</td>
</tr>
<tr>
<td>R2</td>
<td>60000</td>
</tr>
<tr>
<td>R3</td>
<td>45</td>
</tr>
<tr>
<td>R4</td>
<td>198</td>
</tr>
<tr>
<td>R5</td>
<td>12</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Program Counter

| 10000 |
| 10004 |
| 10008 |

ALU Operation

<table>
<thead>
<tr>
<th>in1</th>
<th>ALU</th>
<th>in2</th>
</tr>
</thead>
</table>

Immediate/Disp

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

func

Load/store unit

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
</table>

10001100010000110100111000100000
0000000001100001001010000100000
00000000000000000000000000111001
00000000000000000000000000111001
Key Points

- MIPS is a general-purpose register, load-store, fixed-instruction-length architecture.
- MIPS is optimized for fast pipelined performance, not for low instruction count
- Four principles of IS architecture
  - regularity produces simplicity
  - smaller is faster
  - good design demands compromise
  - make the common case fast