Example of instruction encoding:

- **Register R-type**
  - opcode
  - rs
  - rt
  - rd
  - shamt
  - funct

- **Immediate I-type**
  - opcode
  - rs
  - rt
  - immediate

- **Jump J-type**
  - opcode
  - target

**Example Instruction:**

- opcode=0, rs=1, rt=2, rd=5, sa=0, funct=32
- 000000 00001 00010 00101 00000 100000
- 0000000001000100010100000100000
- 0x00222420
Poll Q: Implications of the MIPS instruction format

What is the maximum number of unique operations MIPS can encode?

3, 64, 127, 128
Accessing the Operands
aka, what’s allowed to go here

- operands are generally in one of two places:
  - registers (32 options)
  - memory ($2^{32}$ locations)

- registers are
  - easy to specify
  - close to the processor (fast access)

- the idea that we want to use registers whenever possible led to load-store architectures.
  - normal arithmetic instructions only access registers
  - only access memory with explicit loads and stores

\[
\text{add } r5, r1, r2
\]
Poll Q: Accessing the Operands

There are typically two locations for operands: registers (internal storage - $t0, $a0) and memory. In each column we have which (reg or mem) is better.

**Which row is correct?**

<table>
<thead>
<tr>
<th></th>
<th>Faster access</th>
<th>Fewer bits to specify</th>
<th>More locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Mem</td>
<td>Mem</td>
<td>Reg</td>
</tr>
<tr>
<td>B</td>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
</tr>
<tr>
<td>C</td>
<td>Reg</td>
<td>Mem</td>
<td>Reg</td>
</tr>
<tr>
<td>D</td>
<td>Reg</td>
<td>Reg</td>
<td>Mem</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MIPS uses a load/store architecture to access operands

can do:

\[
\text{add } t0 = s1 + s2 \\
\text{and } lw \ t0, 32(s3)
\]

can’t do

\[
\text{add } t0 = s1, 32(s3)
\]

→ forces heavy dependence on registers, which is exactly what you want in today’s CPUs
→ more instructions + fast implementation (e.g., easy pipelining)

What pushes MIPS towards a load/store design? (hint: fixed instruction length)
How Many Operands?

*aka how many of these?*

- Most instructions have three operands (e.g., $z = x + y$).
- Well-known ISAs specify 0-3 (explicit) operands per instruction.
- Operands can be specified *implicitly* or *explicitly.*
Historically, many classes of ISAs have been explored, and trade off compactness, performance, and complexity.

<table>
<thead>
<tr>
<th>Style</th>
<th># Operands</th>
<th>Example</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>0</td>
<td>add</td>
<td>( \text{tos}<em>{(N-1)} \leftarrow \text{tos}</em>{(N)} + \text{tos}_{(N-1)} )</td>
</tr>
<tr>
<td>Accumulator</td>
<td>1</td>
<td>add A</td>
<td>( \text{acc} \leftarrow \text{acc} + \text{mem}[A] )</td>
</tr>
<tr>
<td>General Purpose</td>
<td>3</td>
<td>add A B Rc</td>
<td>( \text{mem}[A] \leftarrow \text{mem}[B] + \text{Rc} )</td>
</tr>
<tr>
<td>Register</td>
<td>2</td>
<td>add A Rc</td>
<td>( \text{mem}[A] \leftarrow \text{mem}[A] + \text{Rc} )</td>
</tr>
<tr>
<td>Load/Store:</td>
<td>3</td>
<td>add Ra Rb Rc</td>
<td>Ra \leftarrow Rb + Rc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>load Ra Rb</td>
<td>Ra \leftarrow \text{mem}[Rb]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>store Ra A</td>
<td>( \text{mem}[A] \leftarrow \text{Ra} )</td>
</tr>
</tbody>
</table>
Comparing the Number of Instructions

**Code sequence for C = A + B for four classes of instruction sets:**

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Accumulator</th>
<th>GP Register</th>
<th>GP Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(register-memory)</td>
<td>(load-store)</td>
</tr>
</tbody>
</table>
### Comparing the Number of Instructions

**Code sequence for C = A + B for four classes of instruction sets:**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>GP Register (register-memory)</th>
<th>GP Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Push B</td>
<td>Add</td>
<td>Pop C</td>
</tr>
</tbody>
</table>

Many slides adapted from Dean Tullsen
Comparing the Number of Instructions

Code sequence for $C = A + B$ for four classes of instruction sets:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>GP Register</th>
<th>GP Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>(register-memory)</td>
<td></td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparing the Number of Instructions

Code sequence for \( C = A + B \) for four classes of instruction sets:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>GP Register (register-memory)</th>
<th>GP Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>ADD C, A, B</td>
<td></td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparing the Number of Instructions

**Code sequence for C = A + B for four classes of instruction sets:**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>GP Register</th>
<th>GP Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>ADD C, A, B</td>
<td>Load R1, A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td></td>
<td>Load R2, B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td></td>
<td>Add R3, R1, R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td>Store C, R3</td>
</tr>
</tbody>
</table>
Exercise: Working through alternative ISAs

[if time]

Stack Architecture  Accumulator  GPR  GPR (Load-store)

\[ A = X \times Y - B \times C \]
Poll Q: The destination of a MIPS **add** operation can be...

- Only the top of the stack
- Only the accumulator register
- Any general purpose register,
- Any general purpose register or anywhere in memory
- Any general purpose register or the top of the stack
Addressing Modes
aka: how do we specify the operand we want?

- Register direct  \( R3 \)
- Immediate (literal)  \( \#25 \)
- Direct (absolute)  \( M[10000] \)
- Register indirect  \( M[R3] \)
- Base+Displacement  \( M[R3 + 10000] \)
- Base+Index  \( M[R3 + R4] \)
- Scaled Index  \( M[R3 + R4*d + 10000] \)
- Autoincrement  \( M[R3++] \)
- Autodecrement  \( M[R3 - -] \)
- Memory Indirect  \( M[ M[R3] ] \)
MIPS addressing modes and syntax

**register direct**

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

add $1, $2, $3

**immediate**

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

add $1, $2, #35

**base + displacement**

lw $1, disp($2)

(R1 = M[R2 + disp])

**register indirect**

$disp = 0$

absolute

$ (rs) = 0$
Is this sufficient?

• measurements on the VAX show that these addressing modes (immediate, direct, register indirect, and base+displacement) represent 88% of all addressing mode usage.
• similar measurements show that 16 bits is enough for the immediate 75 to 80% of the time
• and that 16 bits is enough of a displacement 99% of the time.
• (and when these are not sufficient, it typically means we need one more instruction)
What does memory look like anyway?

• 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 (address) points to a byte of memory.

```
  0  8 bits of data
  1  8 bits of data
  2  8 bits of data
  3  8 bits of data
  4  8 bits of data
  5  8 bits of data
  6  8 bits of data
...```

Memory accesses are (often) required to be “word-aligned” because of how buses and memory work

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

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

- Words are aligned
  i.e., what are the least 2 significant bits of a word address?
The MIPS ISA, so far

• fixed 32-bit instructions
• 3 instruction formats (R, I, J)
• 3-operand, load-store architecture
• 32 general-purpose registers
  - R0 always equals 0.
• 2 additional special-purpose integer registers, HI and LO, because multiply and divide produce more than 32 bits.
• registers are 32-bits wide (word)
• register, immediate, and base+displacement addressing modes
But what kinds of things do computers actually do?

- arithmetic
- logical
- data transfer
- conditional branch
- unconditional jump
Which kinds of instructions does (and doesn’t) the MIPS ISA support?

• arithmetic
  – add, subtract, multiply, divide
  – But not: \( \text{start} \), \( \text{mov} \), \( \text{sin} \), \( \text{cos} \), \( \text{adc} \), \( \text{add w/carry} \)

• logical
  – and, or, shift left, shift right
  – But not: \( \text{blt} \), \( \text{ble} \), \( \text{bne} \)

• data transfer
  – load word, store word
  – But not: \( \text{lw} \), \( \text{sw} \), \( \text{li} \), \( \text{lw} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \text{f} \)
“Control Flow” describes how programs execute

• Jumps
• Procedure call (jump subroutine)
• Conditional Branch
  – Used to implement, for example, if-then-else logic, loops, etc.

• Control flow must specify two things
  – Condition under which the jump or branch is taken
  – If take, the location to read the next instruction from (“target”)