Instruction formats

3 instruction formats: all 32 bits

- **R-type**: register
  - arithmetic and logical

- **I-type**: immediate
  - use constant in instruction
  - arithmetic, logical, conditional branch

- **J-type**: jump
  - unconditional branch

Design principle #3: "Good design demands good compromises."
Size of instruction vs. number of formats
Register conventions

Register conventions and mnemonics

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$zero</td>
<td>hardwired 0 value</td>
</tr>
<tr>
<td>1</td>
<td>$at</td>
<td>used by assembler (pseudo-instructions)</td>
</tr>
<tr>
<td>2-3</td>
<td>$v0-1</td>
<td>subroutine return value</td>
</tr>
<tr>
<td>4-7</td>
<td>$a0-3</td>
<td>arguments: subroutine parameter value</td>
</tr>
<tr>
<td>8-15</td>
<td>$t0-7</td>
<td>temp: can be used by subroutine without saving</td>
</tr>
<tr>
<td>16-23</td>
<td>$s0-7</td>
<td>saved: must be saved and restored by subroutine</td>
</tr>
<tr>
<td>24-25</td>
<td>$t8-9</td>
<td>temp</td>
</tr>
<tr>
<td>26-27</td>
<td>$k0-1</td>
<td>kernel: interrupt/trap handler</td>
</tr>
<tr>
<td>28</td>
<td>$gp</td>
<td>global pointer (static or extern variables)</td>
</tr>
<tr>
<td>29</td>
<td>$sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>$fp</td>
<td>frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>$ra</td>
<td>return address for subroutine</td>
</tr>
<tr>
<td></td>
<td>Hi, Lo</td>
<td>used in multiplication (provide 64 bits for result)</td>
</tr>
</tbody>
</table>

Hidden registers

**PC**, the program counter, which stores the current address of the instruction being executed

**IR**, which stores the instruction being executed
**Instruction formats: R-type, I-type**

**R-type**: register

```
addu    $r10,$r8,$r9  # add 2 numbers
```

![Instruction Format Diagram](image)

- **opcode**
- **$rs**
- **$rt**
- **$rd**
- **shamt**
- **function**

3 registers: 2 source, 1 destination

operation: opcode and function
**Instruction formats: R-type, I-type**

**R-type: register**

```
addu    $r10, $r8, $r9  # add 2 numbers
```

```
000000 01000 01001 01010 00000 100001
```

- $r10: Destination register
- $r8, $r9: Source registers
- `addu`: Addition with unsigned overflow

**I-type: immediate**

```
addi    $rt, $rs, immed  # add a constant to a register
```

```
001000 01000 01001 01010 00000 100001
```

- $rt: Destination register
- $rs: Source register
- `immed`: Immediate value

**Semantics:**

- **R[t] = R[s] + (IR_{15})^{16} :: IR_{15-0}**
  - Sign-extend the immediate value to 32 bits
  - Add it (using signed addition) to register R[s]
  - Store the result in register R[t]
**Instruction formats: J-type**

**J-type: jump**

```
j  target  # jump to target address
```

<table>
<thead>
<tr>
<th>opcode</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>01000</td>
</tr>
<tr>
<td>01001</td>
<td>01010</td>
</tr>
<tr>
<td>00000</td>
<td>100001</td>
</tr>
</tbody>
</table>

**b_{31-26}**  **b_{25-0}**

**opcode**  **target**

**semantics:**

\[
\text{PC} \leftarrow \text{PC}_{31-28} :: \text{IR}_{25-0} :: 00
\]

update the PC by using:

- upper 4 bits of the program counter
- 26 bits of the target (lower 26 bits of instruction register)
- two 0's
  (creates a 32-bit address)

**Why 2 0's?**
Other instruction formats: non-MIPS

Other possible formats

older formats were designed to minimize the number of bits in an instruction

3-register format (MIPS)

`addu $r10, $r8, $r9`

2-register format (CISC)

`add2 $r1, $r2`


same register used for source AND target

fewer bits necessary

1-register format (accumulator)

`add1 $r2`

semantics: \( Acc = Acc + R[2] \)

accumulator: special register used to hold results, implicit in instruction

0-register format (stack)

`add0`

semantics:

\[
\text{Stack[Top-4]} = \text{Stack[Top]} + \text{Stack[Top-4]} \\
\text{Top} = \text{Top - 4}
\]

replace top of stack with sum of top 2 values

requires push and pop operations

must go back to memory to reuse value