Toy ISA and microarchitecture

Instruction Fetch and Decode → Registers → ALU → Memory
Instruction Set Architecture (HW/SW Interface)

**Processor**
- Instruction Logic
- Registers

**Memory**
- Encoded Instructions
- Data

**Instructions**
- Names, Encodings
- Effects
- Arguments, Results

**Local storage**
- Names, Size
- How many

**Large storage**
- Addresses, Locations

Computer
Toy Instruction Set Architecture
(a.k.a. Mini-MIPS)

- **Word size = 16 bits, data bus = 16 bits.**
  - Register size = 16 bits.
  - ALU computes on 16-bit values.
- **Memory is byte-addressable**, also access words (byte pairs).
- **16 registers: R0 - R15**
  - R0 always holds hardcoded 0
  - R1 always holds hardcoded 1
  - R2 – R15: general purpose
- **Instructions are 1 word in size.**
- **Separate instruction memory.**
  - Each instruction executes in a single clock cycle.
- **Special Program Counter (PC) register**
  - holds address of next instruction to execute.

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>First instruction, low-order byte</td>
</tr>
<tr>
<td>1</td>
<td>First instruction, high-order byte</td>
</tr>
<tr>
<td>2</td>
<td>Second instruction, low-order byte</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### Instruction Set

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Opcode</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD Rs, Rt, Rd</td>
<td>$R[d] \leftarrow R[s] + R[t]$</td>
<td>0010</td>
<td>s</td>
<td>t</td>
<td>d</td>
</tr>
<tr>
<td>SUB Rs, Rt, Rd</td>
<td>$R[d] \leftarrow R[s] - R[t]$</td>
<td>0011</td>
<td>s</td>
<td>t</td>
<td>d</td>
</tr>
<tr>
<td>AND Rs, Rt, Rd</td>
<td>$R[d] \leftarrow R[s] &amp; R[t]$</td>
<td>0100</td>
<td>s</td>
<td>t</td>
<td>d</td>
</tr>
<tr>
<td>OR Rs, Rt, Rd</td>
<td>$R[d] \leftarrow R[s] \mid R[t]$</td>
<td>0101</td>
<td>s</td>
<td>t</td>
<td>d</td>
</tr>
<tr>
<td>LW Rt, offset(Rs)</td>
<td>$R[t] \leftarrow M[R[s] + offset]$</td>
<td>0000</td>
<td>s</td>
<td>t</td>
<td>offset</td>
</tr>
<tr>
<td>SW Rt, offset(Rs)</td>
<td>$M[R[s] + offset] \leftarrow R[t]$</td>
<td>0001</td>
<td>s</td>
<td>t</td>
<td>offset</td>
</tr>
<tr>
<td>BEQ Rs, Rt, offset</td>
<td>If $R[s] == R[t]$ then [PC] $\leftarrow [PC]+2 + offset*2$ Else [PC] $\leftarrow [PC]+2$</td>
<td>0111</td>
<td>s</td>
<td>t</td>
<td>offset</td>
</tr>
<tr>
<td>JMP offset</td>
<td>$[PC] \leftarrow offset*2$</td>
<td>1000</td>
<td>o</td>
<td>f</td>
<td>f</td>
</tr>
</tbody>
</table>

(R = register file, M = memory)
Instruction Encoding: 3 formats

Arithmetic instructions:
- 2 source register IDs (Rs, Rt)
- 1 destination register ID (Rd)

Memory/branch instructions:
- address/source register ID (Rs)
- data/source register ID (Rt)
- 4-bit offset

Jump instruction:
- 12-bit offset

All have 4-bit opcode in MSBs
Instruction Fetch

Fetch instruction from memory.
Increment program counter (PC) to point to the next instruction.

Instruction Memory
Read Address
Instruction
## Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Opcode</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD Rs, Rt, Rd</td>
<td>( R[d] \leftarrow R[s] + R[t] )</td>
<td>0010</td>
<td>0-15</td>
<td>0-15</td>
<td>0-15</td>
</tr>
<tr>
<td>SUB Rs, Rt, Rd</td>
<td>( R[d] \leftarrow R[s] - R[t] )</td>
<td>0011</td>
<td>0-15</td>
<td>0-15</td>
<td>0-15</td>
</tr>
<tr>
<td>AND Rs, Rt, Rd</td>
<td>( R[d] \leftarrow R[s] &amp; R[t] )</td>
<td>0100</td>
<td>0-15</td>
<td>0-15</td>
<td>0-15</td>
</tr>
<tr>
<td>OR Rs, Rt, Rd</td>
<td>( Rd \leftarrow R[s] \mid R[t] )</td>
<td>0101</td>
<td>0-15</td>
<td>0-15</td>
<td>0-15</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 16-bit Encoding

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>0011</td>
<td>0110</td>
<td>1000</td>
</tr>
</tbody>
</table>
Instruction Decode, Register Access, ALU

Control Unit

Write Enable

ALU control

Instruction

16

Op 4

Register File

Read Addr 1

Read Addr 2

Read Data 1

Read Data 2

Write Addr

Write Data

16

4

16

4

4

16

4

ALU

ALU result

overflow

zero

16

16

16
## Memory Instructions

### LW Rt, offset(Rs)

- **Op**: 0000
- **Rs**: 0-15
- **Rt**: 0-15
- **Rd**: offset

**Meaning**: 
\[ R[t] \leftarrow \text{Mem}[R[s] + \text{offset}] \]

### SW Rt, offset(Rs)

- **Op**: 0001
- **Rs**: 0-15
- **Rt**: 0-15
- **Rd**: offset

**Meaning**: 
\[ \text{Mem}[R[s] + \text{offset}] \leftarrow R[t] \]

...
Memory access

How can we support arithmetic and memory instructions? What's shared?

Control Unit

Write Enable

ALU control

MemWrite

Op

4

16

4

Rs

4

Rt

16

Rt

16

Rd

4

Sign extend

4

16

16

16

16

4

4

16

16

Address

Data Memory

Rd (offset)

4

Write Addr

Write Data

Read Addr 1

Read Data 1

Write Data

Read Addr 2

Read Data 2

ALU

Data

Read Data

Register File
MUXes to the rescue!

![Diagram of a computer architecture with MUXes used to select data for operations involving instruction execution, memory access, and register file interactions.](image)
# Control-flow Instructions

BEQ R1, R2, 28

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Op</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ Rs, Rt, offset</td>
<td>If Rs == Rt then PC ← PC+2 + offset*2</td>
<td>0111</td>
<td>0001</td>
<td>0010</td>
<td>1110</td>
</tr>
<tr>
<td></td>
<td>Else PC ← PC+2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP offset</td>
<td>PC ← offset*2</td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>off</td>
<td>f s</td>
<td>e t</td>
</tr>
</tbody>
</table>

16-bit Encoding

Use these to implement: conditionals, loops, etc.
Compute branch target
Make branch decision

- Instruction Memory Read Address
- PC
- Instruction
- Read Addr 1
- Read Addr 2
- Write Addr
- Write Data
- Read Data 1
- Read Data 2
- ALU control
- ALU
- Overflow
- Zero
- Branch?
All together now...
Microarchitecture

Single-cycle architecture
• Simple, "easily" fits on a slide (and in your head).
• One instruction takes one clock cycle.
• Slowest instruction determines minimum clock cycle.
• Inefficient.

Could it be better?
• How? Performance, energy, debugging, security, reconfigurability, ...
• Pipelining
• OoO: out-of-order execution
• SIMD: single instruction multiple data
• Caching
• Microcode vs. direct hardware implementation
• ... enormous, interesting design space of Computer Architecture