The Single Cycle Processor
MIPS fixed sized instruction formats

### R - Format

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

### I - Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>value or offset</th>
</tr>
</thead>
</table>

### J - Format

<table>
<thead>
<tr>
<th>op</th>
<th>absolute address</th>
</tr>
</thead>
</table>

### ALU

- alu $rd,$rs,$rt

### ALUi

- alui $rt,$rs,value

### Data Transfer

- lw $rt,offset($rs)
- sw $rt,offset($rs)

### Branch

- beq $rs,$rt,offset

### Jump

- j address

### Jump&Link

- jal address

Appendix A & A.10
Review: MIPS instruction formats

Arithmetic
addi $rt, $rs, value
add $rd,$rs,$rt
jr $rs

Data Transfer
lw $rt,offset($rs)
sw $rt,offset($rs)

Conditional branch
beq $rs,$rt,offset
bne $rs,$rt,offset

Unconditional jump
j address
jal address
<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td><code>alu $rd,$rs,$rt</code></td>
<td>$rd = $rs &lt;alu&gt; $rt</td>
</tr>
<tr>
<td>JR</td>
<td><code>jr $rs</code></td>
<td>$pc = $rs</td>
</tr>
<tr>
<td>ALUi</td>
<td><code>alui $rd,$rs,value16</code></td>
<td>$rd = $rs &lt;alu&gt; $\delta^{32}(value16)$</td>
</tr>
<tr>
<td>Data Transfer</td>
<td><code>lw $rt,offset16($rs)</code></td>
<td>$rt = Mem[$rs + $\delta^{32}(offset16)]</td>
</tr>
<tr>
<td></td>
<td><code>sw $rt,offset16($rs)</code></td>
<td>Mem[$rs + $\delta^{32}(offset16)]=$rt</td>
</tr>
<tr>
<td>Branch</td>
<td><code>beq $rs,$rt,offset16</code></td>
<td>$pc = ($rt == $rs)? ($pc+4+($\delta^{32}(offset16)&lt;&lt;2))):($pc+4);</td>
</tr>
<tr>
<td>Jump</td>
<td><code>j address</code></td>
<td>$pc=($pc+4 &amp; 0xF0000000)(addr&lt;&lt;2)</td>
</tr>
<tr>
<td>Jump&amp;Link</td>
<td><code>jal address</code></td>
<td>$ra = $pc+4; $pc=($pc+4 &amp; 0xF0000000)(addr&lt;&lt;2)</td>
</tr>
</tbody>
</table>

$\delta^{32} = \text{sign extend 16 bit number to 32 bits}$
Assembling JUMP Instructions

Suppose the \( \text{fib\_exit} = 0x81fc084C \), \( \text{pc} = 0x81fc08124 \),

\[
\text{j fib\_exit}
\]

Then \( \text{addr} \gg 2 = \text{fib\_exit} \gg 2 = 0x81fc084C \gg 2 \)

\[
= 1000\ 0001\ 1111\ 1100\ 0000\ 1000\ 0100\ 1100 \gg 2
\]

\[
= 0010\ 0000\ 0111\ 1111\ 0000\ 0010\ 0010\ 0011
\]

\[
= 0x27f0223 \text{ (upper 6 bits will be lost for opcode!)}
\]
**Executing JUMP Instructions**

| Jump  | j address | $pc = ($pc+4 & 0xF0000000) | (addr<<2) |

Suppose the pc = 0x81fc08124,

\[ j \quad 0x07f0223 \]

Then address = 0x007f0223

Then address << 2 = 0x01fc084C

Then $pc+4 = 0x81fc08128$

Then $pc+4 & 0xF0000000 = 0x800000000$

Then \[ $pc = ($pc+4 & 0xF0000000) | (addr<<2) = 0x800000000 | 0x01fc084C \]

= 0x81fc084C

Jump \[ j \quad \text{address} \quad $pc = ($pc+4 & 0xF0000000) | (addr<<2) \]
Executing JUMP Instructions different pc

<table>
<thead>
<tr>
<th>Jump</th>
<th>address</th>
<th>$pc = ($pc+4 &amp; 0xF000000)</th>
<th>(addr&lt;&lt;2)</th>
</tr>
</thead>
</table>

Same instruction but instead of pc = 0x81fc08124,
we have pc = 0x31fc08124

\[ j \ 0x07f0223 \]

Then address = 0x007f0223

Then address \( \ll 2 \) = 0x01fc084C

Then $pc+4 = 0x31fc08128$

Then $pc+4 \& 0xF000000 = 0x300000000$

Then $pc = ($pc+4 \& 0xF000000) | (addr<<2) = 0x300000000 | 0x01fc084C

= 0x31fc084C

Completely different address! The jump is memory block dependant.
Book example

Jump \ j \ address \ \$pc= (\$pc+4 \& 0xF000000) \ | \ (addr<<2)\)

Book example (decimal)

\[ \begin{align*}
    j & \quad 10000 \\
    \text{Then jump address} & \quad = 10000_{10} \\
    \text{Then address } & \ll 2 \text{ (or divide by 4)} & \quad = 2500_{10} \\
    \text{assembly} & \quad <j,2500_{10}> \text{ or } <2:2500_{10}> \\
    \text{Execution assume } \text{pc}=0x01fc08124, & \\
    \text{Then } \text{pc}+4 & \quad = 0x01fc0812C \\
    \text{Then } \text{pc}+4 \& 0xF000000 & \quad = 0x000000000 \\
    \text{Then } \text{pc}= (\text{pc}+4 \& 0xF000000) \ | \ (addr<<2) & \quad = 0x000000000 | 10000_{10} \\
    & \quad = 10000_{10}
\end{align*} \]
## Review: MIPS registers and conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Conventional usage</th>
<th>SPARC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0,$zero</td>
<td>0</td>
<td>Constant 0</td>
<td>%g0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>Temporary for pseudo-instructions</td>
<td></td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>Function results</td>
<td>%o0-%o1</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Function arguments 1 to 4</td>
<td>%o0-%o6</td>
</tr>
<tr>
<td>$t1-$t9</td>
<td>8-15,24,35</td>
<td>Temporary</td>
<td>%g1-%g7</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>Saved Temporary</td>
<td>%L0-%L7</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>26-27</td>
<td>Reserved for OS kernel</td>
<td></td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Pointer to global area</td>
<td></td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack pointer</td>
<td></td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame pointer</td>
<td></td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Function Return address</td>
<td>%o7</td>
</tr>
</tbody>
</table>
Review: Function calling

**Calling conventions**

- \$v0:\$v1 = f(\$a0, \$a1, \$a2, \$a3, 0(\$sp), 4(\$sp),...)

**Caller Bookkeeping:**

- Arguments \$a0-\$a3, 0(\$sp), n-4(\$sp)
- Return address \$ra
- Call function jal label # \$ra=pc+4; pc=label

**Callee Bookkeeping:**

- Not restored \$t0 - \$t9, \$a0-\$a3, \$at
- Restore caller’s \$s0 - \$s7, \$sp, \$fp
- Return value \$v0, \$v1
- Return jr \$ra # pc = \$ra
Abstract view of major functional units

Instruction memory

PC

Address

Instruction

Registers

Data

Register #

Register #

Register #

ALU

Address

Data

Data memory

ALU control

ALU result

Zero

3

ALU control

RegWrite

Read register 1

Read register 2

Write register

Write data

Read data 1

Read data 2

Data
R-type instruction datapath

**R - Format**

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

**ALU**

```
func $rd, $rs, $rt
```

**ALU control**

```
3
```

**Zero**

```
ALU result
```

**RegWrite**

```
32
```

**Read register 1**

```
Write register
```

**Read data 1**

```
Write data
```

**Read data 2**

```
```

**ALU result**

```
32
```

**ALU control**

```
3
```

**RegWrite**

```
32
```

**Read data 1**

```
32
```

**Write register**

```
```

**Read register 2**

```
5
```

**Read register 1**

```
5
```

**Write data**

```
5
```

**Read data 2**

```
5
```

**Read data 1**

```
5
```
Lw I-type instruction datapath

I - Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
</table>

Data Transfer

\[ \text{lw } $rt, \text{offset($rs)} \]

Operations:
- Read register 1
- Read data 1
- Write register
- Write data
- Read register 2
- Read data 2
- RegWrite
- MemRead
- MemWrite
- ALU
- ALU control
- Zero
- ALU result
- Address
- Write data
- Data memory
- Data
- Memory
- Sign extend
- 16
- 32
- 3
Von Neuman & Harvard Architectures

**Von Neuman architecture**
Area efficient but requires higher bus bandwidth because instructions and data must compete for memory.

**Harvard architecture** was coined to describe machines with separate memories. Speed efficient: Increased parallelism.
Sw I-type instruction datapath

I - Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
</table>

Data Transfer:  

```
sw $rt, offset($rs)
```
Branch I-type instruction datapath

I - Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
</table>

Data Transfer: \(\text{sw } \$rt, \text{offset}(\$rs)\)

Diagram:
- **ALU control**
- **Sign extend**
- **Shift left 2**
- **RegWrite**
- **Read register 1**
- **Read register 2**
- **Write register**
- **Read data 1**
- **Read data 2**
- **Write data**
- **PC +4**
- **PC**

CWRU EECS 322  16
Muxiplexors: Combining datapaths
Instruction Fetch: pc=pc+4
Combine Branch logic

- **Add**
- **Read address**
- **Instruction**
- **Instruction memory**
- **Read register 1**
- **Read register 2**
- **Write register**
- **Write data**
- **ALU result**
- **Shift left 2**
- **ALU operation**
- **Zero**
- **ALU**
- **Data memory**
- **Read data**
- **Write data**
- **MemRead**
- **MemWrite**
- **RegWrite**
- **Sign extend**
- **16 32**
- **Mux**
- **MemtoReg**
- **3 PCs**
Single Cycle Datapath

**Single-cycle model** (non-overlapping)
- Each instruction executes in a single cycle
- Every instruction and clock-cycle must be stretched to accommodate the slowest instruction (p.438)

Adder2: \( PC \leftarrow PC + \text{signext}(IR[15-0]) \ll 2 \)

Adder1: \( PC \leftarrow PC + 4 \)

Adder3: Arithmetic ALU

Single Cycle = 2 adders + 1 ALU + 4 muxes
Reduced Instruction Set Computer

RISC - Reduced Instruction Set Computer

• By reducing the number of instructions that a processor supports and thereby reducing the complexity of the chip,

• it is possible to make individual instructions execute faster and achieve a net gain in performance

• even though more instructions might be required to accomplish a task.

RISC trades-off

instruction set complexity for instruction execution timing.
RISC Features

- **Large register set**: having more registers allows memory access to be minimized.

- **Load/Store architecture**: operating data in memory directly is one of the most expensive in terms of clock cycle.

- **Fixed length instruction encoding**: This simplifies instruction fetching and decoding logic and allows easy implementation of pipelining.

  All instructions are register-to-register format except Load/Store which access memory

  All instructions execute in a single cycle save branch instructions which require two.

  Almost all single instruction size & same format.
Complex Instruction Set Computer

CISC - Complex Instruction Set Computer

Philosophy: Hardware is always faster than the software.

Objective: Instruction set should be as powerful as possible

With a power instruction set, fewer instructions needed to complete (and less memory) the same task as RISC.

- CISC was developed at a time (early 60’s), when memory technology was not so advanced.
- Memory was small (in terms of kilobytes) and expensive.

But for embedded systems, especially Internet Appliances, memory efficiency comes into play again, especially in chip area and power.
Reality Check: Intel 8086 clock cycles (1978)

Arithmetic
- 3 clocks: add reg16, reg16
- 118-133 clocks: mul dx:ax, reg16 very slow!!
- 128-154 clocks: imul dx:ax, reg16
- 114-162 clocks: div dx:ax, reg16
- 165-184 clocks: idiv dx:ax, reg16

Data Transfer
- 14 clocks: mov reg16, mem16
- 15 clocks: mov mem16, reg16

Conditional Branch
- 4/16 clocks: je displacement8

Unconditional Jump
- 15 clocks: jmp segment:offset16
## Comparison

<table>
<thead>
<tr>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Any</strong> instruction may reference memory</td>
<td>Only load/store references memory</td>
</tr>
<tr>
<td><strong>Many</strong> instructions &amp; addressing modes</td>
<td>Few instructions &amp; addressing modes</td>
</tr>
<tr>
<td>Variable instruction formats</td>
<td>Fixed instruction formats</td>
</tr>
<tr>
<td>Single register set</td>
<td>Multiple register sets</td>
</tr>
<tr>
<td><strong>Multi-clock</strong> cycle instructions</td>
<td><strong>Single-clock</strong> cycle instructions</td>
</tr>
<tr>
<td>Less to no pipelining</td>
<td>Highly pipelined</td>
</tr>
<tr>
<td>Program code size small</td>
<td>Program code size <strong>large</strong></td>
</tr>
<tr>
<td>Micro-program interprets instructions</td>
<td>Hardware (FSM) executes instructions</td>
</tr>
<tr>
<td>Complexity is in the micro-program</td>
<td><strong>Complexity</strong> is in the compiler</td>
</tr>
</tbody>
</table>
Which is better...

RISC

Or

CISC

?
RISC versus CISC

RISC machines: SUN SPARC, SGI Mips, HP PA-RISC

CISC machines: Intel 80x86, Motorola 680x0

What really distinguishes RISC from CISC these days lies in the architecture and not in the instruction set.

CISC occurs whenever there is a disparity in speed:
- between CPU operations and memory accesses
- due to technology or cost.

What about combining both ideas?
Intel 8086 Pentium P6 architecture is externally CISC but internally RISC & CISC!

Intel IA-64 executes many instructions in parallel.