Quick Review: Instruction Set Architecture

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory
Abstraction Hierarchy

Application Software

Operating System *(threads, files, exceptions)*

**Computer Architecture (instruction set)**

Micro-Architecture (execution pipeline)

Logic (adders, multipliers, FSMs)

Digital Circuits (gates)

Analog Circuits (amplifiers)

Process and Devices (MOSFET transistors)

Physics (electrons, holes, tunneling, band gap)

Source: Alex Bronstein
Instruction Set Architecture (ISA)
- Indicating which resources (of processor) are needed
- Explain how instructions can be encoded as a bitstream

Levels of Program Code
- High Level Language (C++, Python, Java)
  - `x = a + b;`
- Assembly Language (ARM, MIPS, x86)
  - `add R1, R2, R3`
- Machine Language
  - `1001 1110 0110 1010`

What needs to be built
- Use a wide spectrum of techniques to make CPU faster

How to program a machine
- Processor executes instructions as a stream (in a sequence)

Software

Hardware

Abstraction layer

Application
For a given level of function, however, that system is best in which one can specify things with the most simplicity and straightforwardness. ... Simplicity and straightforwardness proceed from conceptual integrity. ... Ease of use, then, dictates unity of design, conceptual integrity.

The Mythical Man-Month, Brooks, pg 44
For a given level of function, however, that system is best in which one can specify things with the most simplicity and straightforwardness. ... Simplicity and straightforwardness proceed from conceptual integrity. ... Ease of use, then, dictates unity of design, conceptual integrity.

The Mythical Man-Month, Brooks, pg 44
(von Neumann) Processor Organization

Datapath needs to have the
- Components – the functional units and storage (e.g., register file) needed to execute instructions
- Interconnects - components connected so that the instructions can be accomplished and so that data can be loaded from and stored to Memory

Control needs to
1. Bring input instructions from Memory
2. Issue signals to control the information flow between the Datapath components and to control what operations they perform
3. Manage instruction sequencing
RISC
Reduced Instruction Set Computer

Example
MIPS R3000 ISA
RISC – **Reduced Instruction Set Computer**

**RISC philosophy**
- Fixed instruction lengths
- Load-store instruction sets
- Limited addressing modes
- Limited operations

**Design goals**: Speed, cost (design, fabrication, test, packaging), size, power consumption, reliability, memory space (embedded systems)

---

Nintendo® 64 (MIPS R4300i)  
Sony® PSP® (MIPS R4000)  
Sony® PlayStation® 2 (MIPS R5900)
Overview of MIPS R3000 ISA

**Instruction** = Opcode + Operand specifiers

> Pointed by Program Counter (PC) register
Overview of MIPS R3000 ISA

- Instruction Categories
  - Arithmetic
  - Memory Access (Load/Store)
  - Control Flow
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

3 Instruction Formats: all 32 bits wide

<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>R format</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>OP</th>
<th>jump target</th>
</tr>
</thead>
<tbody>
<tr>
<td>J format</td>
<td></td>
</tr>
</tbody>
</table>
MIPS Registers

Program Counter

Memory

Bus Interface Unit

Registers

Control Unit

ALU / FPU

Register File

32 bits

32 locations

write data

write control

src1 addr

src2 addr

dst addr

write data

src1 data

src2 data
MIPS Register File

Registers are

- **Faster** than main memory
  But register files with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)

- Easier for a compiler to use
  e.g., \((A\times B) - (C\times D) - (E\times F)\) can do multiplies in any order vs. stack

- Can hold variables so that
  Code density improves (since register are named with fewer bits than a memory location)
### MIPS Register Convention

- **Order:** \{Register number, Name, Usage\}

<table>
<thead>
<tr>
<th>Register</th>
<th>Number</th>
<th>Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>$0</td>
<td>$0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>R1</td>
<td>$at</td>
<td>$at</td>
<td>Reserved for assembler</td>
</tr>
<tr>
<td>R2</td>
<td>$v0</td>
<td>$v0</td>
<td>Returned values</td>
</tr>
<tr>
<td>R3</td>
<td>$v1</td>
<td>$v1</td>
<td>Returned values</td>
</tr>
<tr>
<td>R4</td>
<td>$a0</td>
<td>$a0</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>R5</td>
<td>$a1</td>
<td>$a1</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>R6</td>
<td>$a2</td>
<td>$a2</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>R7</td>
<td>$a3</td>
<td>$a3</td>
<td>Procedure arguments</td>
</tr>
</tbody>
</table>
Spilling Registers

- What if the caller needs more registers?
- What if the procedure is recursive?

**Uses a stack** – a last-in-first-out queue – in memory for passing additional values or saving (recursive) return address(es)

### MIPS Register File

- **$0**, **$at**, **$v0**, **$v1**, **$a0**, **$a1**, **$a2**, **$a3**
- **$sp**

### Memory

- **high addr**, **low addr**
- **Top of stack**

### Push and Pop

- **Push**
- **Pop**
MIPS Register Convention

Order: {Register number, Name, Usage}

The **caller may save** these register values and pass via the callee

The **callee may save** these register values and return for the caller

---

The diagram illustrates the register conventions of MIPS. The registers are grouped into three categories:

1. **Constant 0**: Register $0
2. **Reserved for assembler**: Register $at
3. **Returned values**: Registers $v0, $v1
4. **Procedure arguments**: Registers $a0, $a1, $a2, $a3
5. **Caller Save (Temporaries)**: Registers $t0, $t1, $t2, $t3, $t4, $t5, $t6, $t7
6. **Caller Save (Saved registers)**: Registers $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7

The diagram also shows the calling convention for functions Foo() and Bar(). In the Bar() function, the values of $s0 are saved and passed to the callee. The callee modifies $s0 and returns 0 to the caller.

Memory (stack) is used to pass arguments and return values between the caller and callee.

---

**Calling convention**

- **Foo()**
- **Bar()**
  - $a = b + c;
  - Return 0;
MIPS Register Convention

➢ Order: {Register number, Name, Usage}

- **R0**: $0 (Constant 0)
- **R1**: $at (Reserved for assembler)
- **R2**: $v0
- **R3**: $v1
- **R4**: $a0
- **R5**: $a1
- **R6**: $a2
- **R7**: $a3
- **R8**: $t0
- **R9**: $t1
- **R10**: $t2
- **R11**: $t3
- **R12**: $t4
- **R13**: $t5
- **R14**: $t6
- **R15**: $t7
- **R16**: $s0
- **R17**: $s1
- **R18**: $s2
- **R19**: $s3
- **R20**: $s4
- **R21**: $s5
- **R22**: $s6
- **R23**: $s7
- **R24**: $t8
- **R25**: $t9
- **R26**: $k0
- **R27**: $k1
- **R28**: $gp
- **R29**: $sp
- **R30**: $fp
- **R31**: $ra

- **Caller Save (Temporarily)**
- **Caller Save (Saved registers)**
- **Callee Save (Register number, Name, Usage)**

- **Caller Save (Temp)**
- **Reserved for OS**
- **Global Pointer**
- **Stack Pointer**
- **Frame Pointer**
- **Return Address**
MIPS Memory

Alignment restriction
The memory addr of a word must be a multiple of 4

Byte addresses

Endian-based

Memory

2^30 words

32 bits (8 bits x 4)

word address (binary)
Little Endian vs. Big Endian

Way to write/read four bytes within a word (in memory)

- Little Endian version
  - Rightmost byte is word address
  - (Intel 80x86, DEC Vax, DEC Alpha)
- Big Endian version
  - Leftmost byte is word address
  - (IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA)

Memory

Data (32 bit)

| 0x0E | 0x82 | 0xFF | 0x14 |

msb lsb
MIPS Memory Access Instructions

Load: move data from memory to a register in the register file (5bit address)

Store: move data from register to memory (32bit address)
### MIPS Memory Access Instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instr</th>
<th>Op Code</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Transfer (I format)</td>
<td>load word</td>
<td>35</td>
<td>lw $s1, 24($s2)</td>
<td>$s1 = Memory($s2+24)</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>43</td>
<td>sw $s1, 24($s2)</td>
<td>Memory($s2+24) = $s1</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>32</td>
<td>lb $s1, 25($s2)</td>
<td>$s1 = Memory($s2+25)</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>40</td>
<td>sb $s1, 25($s2)</td>
<td>Memory($s2+25) = $s1</td>
</tr>
<tr>
<td></td>
<td>load upper imm</td>
<td>15</td>
<td>lui $s1, 6</td>
<td>$s1 = 6 * $2^{16}</td>
</tr>
</tbody>
</table>

Assembly order is fixed as

Load/Store  <Register number>  <Memory address>

By adding the contents of the **base address register** to the **offset** value
Machine Language - \textit{Load Inst.}

\begin{itemize}
  \item \textbf{Load/Store Instruction Format (I format)}
  \item (I) type accept two registers and one fixed value (immediate)
\end{itemize}

- \textbf{Load/Store} <Register number> <Base register> <Offset>

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode [6 bit] rs [5 bit] rt [5 bit] Immediate (offset) [16 bit]</td>
</tr>
</tbody>
</table>

- \textbf{rs} 5-bits register file address of the \textit{base register}
- \textbf{rt} 5-bits register file address of the \textit{target}
- \textbf{Im} 16-bits offset value (Immediate)

- A 16-bit field meaning access is \textbf{limited} to 8,192 words \((\pm2^{15}\) or 32,768 bytes) of the address in the base register
- Note that the offset can be \textbf{positive or negative}
Machine Language - *Load Inst.*

- Example

\[ \text{lw} \quad \text{rt0}, \quad 24 \quad (\text{rt2}) \]

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit offset</th>
</tr>
</thead>
</table>

\[ 24_{10} + \text{rt2} = \]

\[ \ldots 0001 \quad 1000 \]
\[ + \quad \ldots 1001 \quad 0100 \]
\[ \ldots 1010 \quad 1100 = 0x120040ac \]

Memory

<table>
<thead>
<tr>
<th>data</th>
<th>word address (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xf f f f f f f f</td>
</tr>
<tr>
<td></td>
<td>0x120040ac</td>
</tr>
<tr>
<td></td>
<td>0x12004094</td>
</tr>
<tr>
<td></td>
<td>0x00000000c</td>
</tr>
<tr>
<td></td>
<td>0x00000008</td>
</tr>
<tr>
<td></td>
<td>0x00000004</td>
</tr>
<tr>
<td></td>
<td>0x00000000</td>
</tr>
</tbody>
</table>
Loading and Storing Bytes

MIPS provides special instructions to move bytes

<table>
<thead>
<tr>
<th>Category</th>
<th>Instr</th>
<th>Op Code</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Transfer</td>
<td>load byte</td>
<td>32</td>
<td>lb $s1, 25($s2)</td>
<td>$s1 = Memory($s2+25)</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>40</td>
<td>sb $s1, 25($s2)</td>
<td>Memory($s2+25) = $s1</td>
</tr>
</tbody>
</table>

What 8 bits get loaded and stored?

- Load byte places the byte from memory in the rightmost 8 bits of the destination register
  - What happens to the other bits in the register?
- Store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
  - What happens to the other bits in the memory word?
MIPS Immediate Instructions

- Small constants are used often in typical code
- Possible approaches?
  1. put “typical constants” in memory and load them
  2. create hard-wired registers (like $zero) for constants like 1
  3. have special instructions that contain constants!

Special instructions that contain constant!

```
addi $sp,$sp,4  #$sp = $sp + 4
slti $t0,$s2,15  #$t0 = 1 if $s2 < 15
```

NOTE: Immediate format limits values to the range $+2^{15}-1$ to $-2^{15}$
Aside: How About Larger Constants?

- We’d also like to be able to load a 32 bit constant into a register, for this we must use two instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instr</th>
<th>Op Code</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Transfer</td>
<td>load word</td>
<td>35</td>
<td>lw $s1, 24($s2)</td>
<td>$s1 = Memory($s2+24)</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>43</td>
<td>sw $s1, 24($s2)</td>
<td>Memory($s2+24) = $s1</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>32</td>
<td>lb $s1, 25($s2)</td>
<td>$s1 = Memory($s2+25)</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>40</td>
<td>sb $s1, 25($s2)</td>
<td>Memory($s2+25) = $s1</td>
</tr>
<tr>
<td></td>
<td>load upper imm</td>
<td>15</td>
<td>lui $s1, 6</td>
<td>$s1 = 6 * 2^{16}</td>
</tr>
</tbody>
</table>

- A new “load upper immediate instruction”

```text
lui $t0, 1010101010101010
ori $t0, $t0, 1010101010101010
```

### Terms
- **lui**: Load Upper Immediate
- **ori**: Or Immediate
MIPS Arithmetic Instructions

Arithmetic / logical computation
## MIPS Arithmetic Instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instr</th>
<th>Op Code</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic (R &amp; I format)</td>
<td>add</td>
<td>0 and 32</td>
<td>add $t0, $s1, $s2</td>
<td>$t0 = $s1 + $s2</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>0 and 34</td>
<td>sub $t0, $s1, $s2</td>
<td>$t0 = $s1 - $s2</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>8</td>
<td>add $t0, $s1, 6</td>
<td>$t0 = $s1 + 6</td>
</tr>
<tr>
<td></td>
<td>or immediate</td>
<td>13</td>
<td>ori $t0, $s1, 6</td>
<td>$t0 = $s1 v 6</td>
</tr>
</tbody>
</table>

Assembly order is fixed as:

Add/Sub <Destination> <Source1> <Source2>

**Destination first**

All contained in the datapath’s register file – indicated by $
### Machine Language - Add Inst.

- **Arithmetic Instruction Format** (R format)
- Register (R) type instructions accept three registers

#### Opcode

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>6-bit</td>
<td>6-bits opcode that specifies the operation</td>
</tr>
<tr>
<td>rs</td>
<td>5-bit</td>
<td>5-bits register file address of the first source operand</td>
</tr>
<tr>
<td>rt</td>
<td>5-bit</td>
<td>5-bits register file address of the second source operand</td>
</tr>
<tr>
<td>rd</td>
<td>5-bit</td>
<td>5-bits register file address of the result’s destination</td>
</tr>
<tr>
<td>shamnt</td>
<td>5-bit</td>
<td>5-bits shift amount (for shift instructions)</td>
</tr>
<tr>
<td>funct</td>
<td>6-bit</td>
<td>6-bits function code augmenting the opcode</td>
</tr>
</tbody>
</table>

#### Format Diagram

```
Add/Sub <Destination> <Source1> <Source2>
```

```
| Bit | 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|     | 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
| Opcode |     | rs  |     | rt  |     | rd  |     | shamnt | funct |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
```
MIPS Control Flow Instructions

PC
Inst.
Inst.
Control (e.g., branch)

Bus Interface Unit
Registers
Control Unit
ALU / FPU
## Control Flow Instructions (Jump)

<table>
<thead>
<tr>
<th>Category</th>
<th>Instr</th>
<th>Op Code</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncond. Jump (J &amp; R format)</td>
<td>jump</td>
<td>2</td>
<td>j</td>
<td>2500 go to 10000</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>0 and 8</td>
<td>jr</td>
<td>$t1 go to $t1</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>3</td>
<td>jal</td>
<td>2500 go to 10000; $ra=PC+4</td>
</tr>
</tbody>
</table>

Assembly order is fixed as

**Jump** <Label>

Similar to memory access instruction (load/store)

Branch destination = Base register (PC) + Offset
Jump Instruction Format (J format)

(J) type instructions receive a fixed (immediate) address

Jump destination address calculation
## Control Flow Instructions (Branch)

<table>
<thead>
<tr>
<th>Category</th>
<th>Instr</th>
<th>Op Code</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond. Branch (I &amp; R format)</td>
<td>br on equal</td>
<td>4</td>
<td>beq $s1, $s2, L</td>
<td>if ($s1==𝑠2) go to L</td>
</tr>
<tr>
<td></td>
<td>br on not equal</td>
<td>5</td>
<td>bne $s1, $s2, L</td>
<td>if ($s1 !=$s2) go to L</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>0 and 42</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2&lt;$s3) $s1=1 else $s1=0</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>10</td>
<td>slti $s1, $s2, 6</td>
<td>if ($s2&lt;6) $s1=1 else $s1=0</td>
</tr>
</tbody>
</table>

Assembly order is fixed as

Branch `<Source1>` `<Source2>` `<Label>`

Comparison targets

Similar to memory access instruction (load/store)

Branch destination = Base register \((PC)\) + Offset
**Machine Language - Branch Inst.**

- **Branch Instruction Format (I format)**
- **Similar to memory access instruction (Load/Store)**

### Branch Instruction Format (I format)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|
| Branch | Opcode [6 bit] | rs [5 bit] | rt [5 bit] | Immediate (offset) [16 bit] | <Source1> | <Source2> | <Offset> |

*Again, branch instruction uses instruction address register (the PC)*
Specifying Branch Destinations

- Instruction Address Register (the PC)
- PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction

Branch destination address calculation

**Instruction**

<table>
<thead>
<tr>
<th>31 30 29 28 27</th>
<th>26 25 24 23 22</th>
<th>21 20 19 18 17</th>
<th>16 15 14 13 12</th>
<th>11 10 9 8 7</th>
<th>6 5 4 3 2</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode [6 bit]</td>
<td>rs [5 bit]</td>
<td>rt [5 bit]</td>
<td>Immediate (offset) [16 bit]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- sign-extend
- PC for Branched Instruction
- Add
- Add
- branch dst address

\[ \text{branch dst address} \]
Aside: Branching Far Away

What if the branch destination is further away that can be captured in 16 bits?

The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition:

```
beq  $s0, $s1, L1
```

becomes

```
bne  $s0, $s1, L2
j    L1
```

Extended to 26bits!
Aside: Accessing Procedures

- MIPS `procedure call` instruction:
  
  ```
  jal ProcedureAddress    #jump and link
  ```

- Saves PC+4 in register $ra to have a link to the next instruction for the procedure return

- Machine format (J format):
  
  ```
  op  26 bit address
  ```

- Then can do procedure `return` with a
  
  ```
  jr  $ra    #return
  ```

- Instruction format (R format):
  
  ```
  op  rs  funct
  ```
MIPS *assemblers* support pseudo-instructions that give the illusion of a more expressive instruction set, but are actually translated into one or more simpler, “real” instructions.

It’s why the assembler needs a reserved register ($at)

**Branch-if-less-than (blt)**

- **Cause use** `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create other conditions.

- **Set on less than instruction**
  
  ```
  slt $t0, $s0, $s1  # if $s0 < $s1 then
  # $t0 = 1  else
  # $t0 = 0
  ```

- **Branch-if-less-then (blt $s1,$s2,Label)**
  
  ```
  slt $at, $s1, $s2  #$at set to 1 if
  bne $at, $zero, Label  # $s1 < $s2
  ```
MIPS Organization So Far

Processor

- **Register File**: 32 registers ($zero - $ra)
- **src1 addr**: 5 bits
- **src2 addr**: 5 bits
- **dst addr**: 5 bits
- **write data**: 32 bits
- **branch offset**: 4 bits
- **PC**: 32 bits
- **Add**: 32 bits
- **ALU**: 32 bits
- **Fetch**: PC = PC+4
- **Decode**: 32 bits
- **Exec**: 32 bits

Memory

- **read/write addr**: 32 bits
- **read data**: 32 bits
- **write data**: 32 bits
- **byte address**: 32 bits (big Endian)
- **word address**: 32 bits
- **0...1100**
- **0...1000**
- **0...0100**
- **0...0000**
- **2^{30}** words

- **32 bits**
- **0...0000**
- **1...1100**
RISC-V
(Pronounced “risk-five”)

Open Source Instruction Set
RISC-V vs. MIPS

- RISC-V and MIPS (a variation of RISC-I) are effectively same except for
- RISC-V is designed as modular; instead of taking simple but monolithic, the modular design enables a more flexible implementation that suit specific applications

### RISC-V Instruction Set Architecture

- Compressed
- Dynamically translated languages
- Transactional memory
- Packed SIMD
- User-level interrupts

#### Instructions

- **C**: Base integer
- **M**: Integer multiplication and division
- **A**: Atomic operations
- **F**: Floating point – single
- **D**: Floating point – double
- **Q**: Floating point – quad
Aside: Create Custom Processor

- Example: Simple integer calculator
- Only I & M modules are required

RISC-V Instruction Set Architecture

- Compressed
- Dynamically translated languages
- Transactional memory
- Packed SIMD
- User-level interrupts

I  Base integer

M  Integer multiplication and division

C
J
T
P
N

A
F
D
Q

Atomic operations
Floating point – single
Floating point – double
Floating point – quad
### RISC-V vs. MIPS

- RISC-V and MIPS (a variation of RISC-I) are effectively same except for:
  - The number of RISC-V’s default format of is “**SIX**” (while MIPS is three; I-type, R-type and J-type)

#### Instruction Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Bits</th>
<th>Opcodes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R-type</strong></td>
<td>32</td>
<td>Arithmetic Instructions</td>
</tr>
<tr>
<td><strong>I-type</strong></td>
<td>32</td>
<td>Load Instructions</td>
</tr>
<tr>
<td><strong>S-type</strong></td>
<td>32</td>
<td>Store Instructions</td>
</tr>
<tr>
<td><strong>B-type</strong></td>
<td>32</td>
<td>Conditional Branches</td>
</tr>
<tr>
<td><strong>U-type</strong></td>
<td>32</td>
<td>LUI (Load Upper Immediate) Instructions</td>
</tr>
<tr>
<td><strong>J-type</strong></td>
<td>32</td>
<td>Uncoditional Jumps</td>
</tr>
</tbody>
</table>

#### Example Instructions:
- **R-type**:
  - `add rs1, rs2, rd` (ex: Arithmetic Instructions)
- **I-type**:
  - `addi rs1, rs2, imm` (ex: Load Instructions)
- **S-type**:
  - `addi rs1, rs2, imm` (ex: Store Instructions)
- **B-type**:
  - `beq rs1, rs2, imm` (ex: Conditional Branches)
- **U-type**:
  - `lui rs1, imm` (ex: LUI Instructions)
- **J-type**:
  - `j imm` (ex: Uncoditional Jumps)
Aside: RISC-V vs. MIPS Instruction Set

- **Register-register**
  - **RISC-V (R-type)**: Func7(7) | Rs2(5) | Rs1(5) | Funct(3) | Rd(5) | Opcode(7)
  - **MIPS (R-type)**: Op(6) | Rs1(5) | Rs2(5) | Rd(5) | Const(5) | Op(6)

- **Load**
  - **RISC-V (B-type)**: Immediate(12) | Rs1(5) | Funct(3) | Rd(5) | Opcode(7)
  - **MIPS (I-type)**: Op(6) | Rs1(5) | Rs2(5) | Const(16)

- **Branch**
  - **RISC-V (S-type)**: Immediate(7) | Rs2(5) | Rs1(5) | Funct(3) | Immediate(5) | Opcode(7)
  - **MIPS (I-type)**: Op(6) | Rs1(5) | Op(6) | Rs2(5) | Const(16)
RISC-V vs. MIPS

- RISC-V and MIPS (a variation of RISC-I) are effectively same except for
- Some pseudo instructions are not allowed in RISC-V (E.g., all 0 operands are noop, but not valid in RISC-V)

![Table Comparing MIPS and RISC-V NOOP Instructions](image)

- **MIPS NOOP instruction**
  - Description: Performs no operation.
  - Operation: advance_pc (4);
  - Syntax: noop
  - Encoding: 0000 0000 0000 0000 0000 0000 0000 0000

- **RISC-V NOOP instruction**
  - Encoded as **ADDI $x0,$x0,0**
Wrap up

Review: Addressing Modes of MIPS
Review of MIPS **Operand Addressing Modes**

- **Register addressing** – operand is in a register

  \[
  \begin{array}{c|c|c|c|c|c}
  \text{op} & \text{rs} & \text{rt} & \text{rd} & \text{funct} \\
  \end{array}
  \]

  Register

  word operand

- **Base (displacement) addressing** – operand is at the memory location whose address is the sum of a register and a 16-bit constant contained within the instruction

  \[
  \begin{array}{c|c|c|c|c|c}
  \text{op} & \text{rs} & \text{rt} & \text{offset} \\
  \end{array}
  \]

  Memory

  word or byte operand

- **Immediate addressing** – operand is a 16-bit constant contained within the instruction

  \[
  \begin{array}{c|c|c|c|c}
  \text{op} & \text{rs} & \text{rt} & \text{operand} \\
  \end{array}
  \]
Review of MIPS Instruction Addressing Modes

- **PC-relative** addressing – instruction address is the sum of the PC and a 16-bit constant contained within the instruction.

- **Pseudo-direct** addressing – instruction address is the 26-bit constant contained within the instruction concatenated with the upper 4 bits of the PC.
Quick Review: Instruction Set Architecture

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory