MIPS Arithmetic
and Logic Instructions

COE 301
Computer Organization
Prof. Muhamed Mudawar

College of Computer Sciences and Engineering
King Fahd University of Petroleum and Minerals
Overview of the MIPS Architecture

R-Type Instruction Format

R-type Arithmetic, Logical, and Shift Instructions

I-Type Instruction Format and Immediate Constants

I-type Arithmetic and Logical Instructions

Pseudo Instructions
Overview of the MIPS Architecture

Memory

- Up to $2^{32}$ bytes = $2^{30}$ words
- 4 bytes per word

EIU
- $0$, $1$, $2$, $31$
- Execution & Integer Unit (Main proc)
- ALU
- Integer mul/div
- Hi, Lo

Execution & Integer Unit

FPU
- F0, F1, F2, F31
- Floating Point Unit (Coproc 1)
- FP Arith

Floating Point Unit

TMU
- BadVaddr, Status, Cause, EPC
- Trap & Memory Unit (Coproc 0)

Floating Point Unit

32 General Purpose Registers

Arithmetic & Logic Unit

Integer Multiplier/Divider

EPC

Cause

Status

BadVaddr

TMU

Execution & Integer Unit

32 Floating-Point Registers

Floating-Point Arithmetic Unit

EIU

FPU
MIPS General-Purpose Registers

- 32 General Purpose Registers (GPRs)
  - All registers are 32-bit wide in the MIPS 32-bit architecture
  - Software defines names for registers to standardize their use
  - Assembler can refer to registers by name or by number ($ notation)

<table>
<thead>
<tr>
<th>Name</th>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>$0</td>
<td>Always 0 (forced by hardware)</td>
</tr>
<tr>
<td>$at</td>
<td>$1</td>
<td>Reserved for assembler use</td>
</tr>
<tr>
<td>$v0 – $v1</td>
<td>$2 – $3</td>
<td>Result values of a function</td>
</tr>
<tr>
<td>$a0 – $a3</td>
<td>$4 – $7</td>
<td>Arguments of a function</td>
</tr>
<tr>
<td>$t0 – $t7</td>
<td>$8 – $15</td>
<td>Temporary Values</td>
</tr>
<tr>
<td>$s0 – $s7</td>
<td>$16 – $23</td>
<td>Saved registers (preserved across call)</td>
</tr>
<tr>
<td>$t8 – $t9</td>
<td>$24 – $25</td>
<td>More temporaries</td>
</tr>
<tr>
<td>$k0 – $k1</td>
<td>$26 – $27</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>$28</td>
<td>Global pointer (points to global data)</td>
</tr>
<tr>
<td>$sp</td>
<td>$29</td>
<td>Stack pointer (points to top of stack)</td>
</tr>
<tr>
<td>$fp</td>
<td>$30</td>
<td>Frame pointer (points to stack frame)</td>
</tr>
<tr>
<td>$ra</td>
<td>$31</td>
<td>Return address (used for function call)</td>
</tr>
</tbody>
</table>
Instruction Categories

- Integer Arithmetic (our focus in this presentation)
  - Arithmetic, logic, and shift instructions
- Data Transfer
  - Load and store instructions that access memory
  - Data movement and conversions
- Jump and Branch
  - Flow-control instructions that alter the sequential sequence
- Floating Point Arithmetic
  - Instructions that operate on floating-point registers
- Miscellaneous
  - Instructions that transfer control to/from exception handlers
  - Memory management instructions
Overview of the MIPS Architecture

R-Type Instruction Format

R-type Arithmetic, Logical, and Shift Instructions

I-Type Instruction Format and Immediate Constants

I-type Arithmetic and Logical Instructions

Pseudo Instructions
**R-Type Instruction Format**

- **Op**: operation code (opcode)
  - Specifies the operation of the instruction
  - Also specifies the format of the instruction

- **funct**: function code – extends the opcode
  - Up to $2^6 = 64$ functions can be defined for the same opcode
  - MIPS uses opcode 0 to define many R-type instructions

- Three Register Operands (common to many instructions)
  - **Rs, Rt**: first and second source operands
  - **Rd**: destination operand
  - **sa**: the shift amount used by shift instructions
R-Type Integer Add and Subtract

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Op</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th>sa</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $t1, $t2, $t3</td>
<td>$t1 = $t2 + $t3</td>
<td>0</td>
<td>$t2</td>
<td>$t3</td>
<td>$t1</td>
<td>0</td>
<td>0x20</td>
</tr>
<tr>
<td>addu $t1, $t2, $t3</td>
<td>$t1 = $t2 + $t3</td>
<td>0</td>
<td>$t2</td>
<td>$t3</td>
<td>$t1</td>
<td>0</td>
<td>0x21</td>
</tr>
<tr>
<td>sub $t1, $t2, $t3</td>
<td>$t1 = $t2 – $t3</td>
<td>0</td>
<td>$t2</td>
<td>$t3</td>
<td>$t1</td>
<td>0</td>
<td>0x22</td>
</tr>
<tr>
<td>subu $t1, $t2, $t3</td>
<td>$t1 = $t2 – $t3</td>
<td>0</td>
<td>$t2</td>
<td>$t3</td>
<td>$t1</td>
<td>0</td>
<td>0x23</td>
</tr>
</tbody>
</table>

- **add, sub**: arithmetic overflow causes an **exception**
  - In case of overflow, result is not written to destination register
- **addu, subu**: arithmetic overflow is **ignored**
- **addu, subu**: compute the same result as **add, sub**
- Many programming languages ignore overflow
  - The + operator is translated into **addu**
  - The – operator is translated into **subu**
Bits have NO meaning. The same \( n \) bits stored in a register can represent an unsigned or a signed integer.

- **Unsigned Integers:** \( n \)-bit representation

- **Signed Integers:** \( n \)-bit 2's complement representation

### Finite Set of Signed Integers
- Min: \(-2^{n-1}\)
- Max: \(2^{n-1}-1\)

### Finite Set of Unsigned Integers
- Min: \(0\)
- Max: \(2^n-1\)
Carry and Overflow

- Carry is useful when adding (subtracting) unsigned integers
  - Carry indicates that the **unsigned sum** is out of range
- Overflow is useful when adding (subtracting) signed integers
  - Overflow indicates that the **signed sum** is out of range
- Range for 32-bit unsigned integers = 0 to \(2^{32} - 1\)
- Range for 32-bit signed integers = \(-2^{31}\) to \(2^{31} - 1\)
- Example 1: Carry = 1, Overflow = 0 (NO overflow)

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
+ & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]

Unsigned sum is out-of-range, but the Signed sum is correct
More Examples of Carry and Overflow

- Example 2: Carry = 0, Overflow = 1

\[
\begin{array}{cccccccccccccccc}
01111 & 1 & & & & & & & & 11 & 1 \\
0010 & 0100 & 0000 & 0100 & 1011 & 0001 & 0100 & 0100 \\
+ & 0111 & 1111 & 0111 & 0000 & 0011 & 0101 & 0000 & 0010 \\
\hline
1010 & 0011 & 0111 & 0100 & 1110 & 0110 & 0100 & 0110 \\
\end{array}
\]

Unsigned sum is correct, but the Signed sum is out-of-range.

- Example 3: Carry = 1, Overflow = 1

\[
\begin{array}{cccccccccccccccc}
1 & 11 & 1 & & & & & & & & 11 & 1 \\
1000 & 0100 & 0000 & 0100 & 1011 & 0001 & 0100 & 0100 \\
+ & 1001 & 1111 & 0111 & 0000 & 0011 & 0101 & 0000 & 0010 \\
\hline
0010 & 0011 & 0111 & 0100 & 1110 & 0110 & 0100 & 0110 \\
\end{array}
\]

Both the Unsigned and Signed sums are out-of-range.
Using Add / Subtract Instructions

- Consider the translation of: \( f = (g+h)-(i+j) \)
- Programmer / Compiler allocates registers to variables
- Given that: \( $t0=f, $t1=g, $t2=h, $t3=i, \text{ and } $t4=j \)
- Called temporary registers: \( $t0=8, $t1=9, \ldots \)
- Translation of: \( f = (g+h)-(i+j) \)

```
addu $t5, $t1, $t2 # $t5 = g + h
addu $t6, $t3, $t4 # $t6 = i + j
subu $t0, $t5, $t6 # f = (g+h)-(i+j)
```

- Assembler translates `addu $t5,$t1,$t2` into binary code

<table>
<thead>
<tr>
<th>Op</th>
<th>$t1</th>
<th>$t2</th>
<th>$t5</th>
<th>sa</th>
<th>addu</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>01001</td>
<td>01010</td>
<td>01101</td>
<td>00000</td>
<td>100001</td>
</tr>
</tbody>
</table>
Logic Bitwise Operations

- Logic bitwise operations: **and, or, xor, nor**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x and y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x or y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x xor y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x nor y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- AND instruction is used to clear bits: **x and 0 → 0**
- OR instruction is used to set bits: **x or 1 → 1**
- XOR instruction is used to toggle bits: **x xor 1 → not x**
- NOT instruction is not needed, why?

```plaintext
not $t1, $t2 is equivalent to: nor $t1, $t2, $t2
```
# Logic Bitwise Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Op</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th>sa</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>and $t1, $t2, $t3</td>
<td>$t1 = $t2 &amp; $t3</td>
<td>0</td>
<td>$t2</td>
<td>$t3</td>
<td>$t1</td>
<td>0</td>
<td>0x24</td>
</tr>
<tr>
<td>or  $t1, $t2, $t3</td>
<td>$t1 = $t2</td>
<td>$t3</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0x25</td>
</tr>
<tr>
<td>xor $t1, $t2, $t3</td>
<td>$t1 = $t2 ^ $t3</td>
<td>0</td>
<td>$t2</td>
<td>$t3</td>
<td>$t1</td>
<td>0</td>
<td>0x26</td>
</tr>
<tr>
<td>nor $t1, $t2, $t3</td>
<td>$t1 = ~(t2</td>
<td>t3)</td>
<td>0</td>
<td>$t2</td>
<td>$t3</td>
<td>$t1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Examples:**

Given: \( t1 = 0x{ab}cd{12}34 \) and \( t2 = 0xffffffff \)

- and \( t0, t1, t2 \) # \( t0 = 0x{ab}cd{00}00 \)
- or \( t0, t1, t2 \) # \( t0 = 0xffffffff \)
- xor \( t0, t1, t2 \) # \( t0 = 0x5432{12}34 \)
- nor \( t0, t1, t2 \) # \( t0 = 0x0000{ed}cb \)
Shift Operations

- Shifting is to move the 32 bits of a number left or right
- \texttt{sll} means \textit{shift left logical} (insert zero from the right)
- \texttt{srl} means \textit{shift right logical} (insert zero from the left)
- \texttt{sra} means \textit{shift right arithmetic} (insert sign-bit)
- The \textbf{5-bit shift amount} field is used by these instructions
## Shift Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Op</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th>sa</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sll</code></td>
<td><code>$t1 = $t2 &lt;&lt; 10</code></td>
<td>0</td>
<td>0</td>
<td>$t2</td>
<td>$t1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td><code>srl</code></td>
<td><code>$t1 = $t2 &gt;&gt;&gt; 10</code></td>
<td>0</td>
<td>0</td>
<td>$t2</td>
<td>$t1</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td><code>sra</code></td>
<td><code>$t1 = $t2 &gt;&gt; 10</code></td>
<td>0</td>
<td>0</td>
<td>$t2</td>
<td>$t1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td><code>sllv</code></td>
<td><code>$t1 = $t2 &lt;&lt; $t3</code></td>
<td>0</td>
<td>$t3</td>
<td>$t2</td>
<td>$t1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td><code>srlv</code></td>
<td><code>$t1 = $t2 &gt;&gt;&gt;$t3</code></td>
<td>0</td>
<td>$t3</td>
<td>$t2</td>
<td>$t1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td><code>srav</code></td>
<td><code>$t1 = $t2 &gt;&gt; $t3</code></td>
<td>0</td>
<td>$t3</td>
<td>$t2</td>
<td>$t1</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

- **sll, srl, sra:** shift by a constant amount
  - The shift amount (sa) field specifies a number between 0 and 31

- **sllv, srlv, srav:** shift by a variable amount
  - A source register specifies the variable shift amount between 0 and 31
  - Only the lower 5 bits of the source register is used as the shift amount
Shift Instruction Examples

- Given that: $t2 = 0xabcd1234$ and $t3 = 16$

\[
sll \quad t1, \quad t2, \quad 8 \quad \Rightarrow \quad t1 = 0xcd123400
\]
\[
srl \quad t1, \quad t2, \quad 4 \quad \Rightarrow \quad t1 = 0x0abcd123
\]
\[
sra \quad t1, \quad t2, \quad 4 \quad \Rightarrow \quad t1 = 0xfaabcd123
\]
\[
srlv \quad t1, \quad t2, \quad t3 \quad \Rightarrow \quad t1 = 0x0000abcd
\]
Binary Multiplication

- Shift Left Instruction (**sll**) can perform multiplication
  - When the multiplier is a power of 2
- You can factor any binary number into powers of 2
- Example: multiply $t0$ by 36
  
  $t0 \times 36 = t0 \times (4 + 32) = t0 \times 4 + t0 \times 32$

  ```plaintext
  sll $t1, t0, 2       # $t1 = t0 \times 4
  sll $t2, t0, 5       # $t2 = t0 \times 32
  addu $t3, $t1, $t2   # $t3 = t0 \times 36
  ```
Your Turn . . .

Multiply $t0$ by 26, using shift and add instructions

Hint: $26 = 2 + 8 + 16$

\[
\begin{align*}
\text{sll } & \text{$t1, $t0, 1} & \quad \# \quad \text{$t1 = $t0 * 2} \\
\text{sll } & \text{$t2, $t0, 3} & \quad \# \quad \text{$t2 = $t0 * 8} \\
\text{sll } & \text{$t3, $t0, 4} & \quad \# \quad \text{$t3 = $t0 * 16} \\
\text{addu } & \text{$t4, $t1, $t2} & \quad \# \quad \text{$t4 = $t0 * 10} \\
\text{addu } & \text{$t5, $t4, $t3} & \quad \# \quad \text{$t5 = $t0 * 26}
\end{align*}
\]

Multiply $t0$ by 31, Hint: $31 = 32 – 1$

\[
\begin{align*}
\text{sll } & \text{$t1, $t0, 5} & \quad \# \quad \text{$t1 = $t0 * 32} \\
\text{subu } & \text{$t2, $t1, $t0} & \quad \# \quad \text{$t2 = $t0 * 31}
\end{align*}
\]
Overview of the MIPS Architecture

R-Type Instruction Format

R-type Arithmetic, Logical, and Shift Instructions

I-Type Instruction Format and Immediate Constants

I-type Arithmetic and Logical Instructions

Pseudo Instructions
Constant are used quite frequently in programs

- The R-type shift instructions have a 5-bit shift amount constant
- What about other instructions that need a constant?

**I-Type: Instructions with Immediate Operands**

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs</th>
<th>Rt</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

16-bit immediate constant is stored inside the instruction

- Rs is the source register number
- Rt is now the **destination** register number (for R-type it was Rd)

**Examples of I-Type ALU Instructions:**

- Add immediate: `addi $t1, $t2, 5`  
  # $t1 = $t2 + 5
- OR immediate: `ori $t1, $t2, 5`   
  # $t1 = $t2 | 5
### I-Type ALU Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Op</th>
<th>Rs</th>
<th>Rt</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $t1, $t2, 25</td>
<td>$t1 = $t2 + 25</td>
<td>0x8</td>
<td>$t2</td>
<td>$t1</td>
<td>25</td>
</tr>
<tr>
<td>addiu $t1, $t2, 25</td>
<td>$t1 = $t2 + 25</td>
<td>0x9</td>
<td>$t2</td>
<td>$t1</td>
<td>25</td>
</tr>
<tr>
<td>andi $t1, $t2, 25</td>
<td>$t1 = $t2 &amp; 25</td>
<td>0xc</td>
<td>$t2</td>
<td>$t1</td>
<td>25</td>
</tr>
<tr>
<td>ori $t1, $t2, 25</td>
<td>$t1 = $t2</td>
<td>0xd</td>
<td>$t2</td>
<td>$t1</td>
<td>25</td>
</tr>
<tr>
<td>xori $t1, $t2, 25</td>
<td>$t1 = $t2 ^ 25</td>
<td>0xe</td>
<td>$t2</td>
<td>$t1</td>
<td>25</td>
</tr>
<tr>
<td>lui $t1, 25</td>
<td>$t1 = 25 &lt;&lt; 16</td>
<td>0xfe</td>
<td>0</td>
<td>$t1</td>
<td>25</td>
</tr>
</tbody>
</table>

- **addi**: overflow causes an arithmetic exception
  - In case of overflow, result is not written to destination register

- **addiu**: same operation as **addi** but overflow is ignored

- Immediate constant for **addi** and **addiu** is signed
  - No need for **subi** or **subiu** instructions

- Immediate constant for **andi**, **ori**, **xori** is unsigned
### Examples of I-Type ALU Instructions

- Given that registers $t0$, $t1$, $t2$ are used for $A$, $B$, $C$

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent MIPS Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A = B + 5;$</td>
<td>addiu $t0$, $t1$, 5</td>
</tr>
<tr>
<td>$C = B - 1;$</td>
<td>addiu $t2$, $t1$, -1</td>
</tr>
<tr>
<td>$A = B &amp; 0xf;$</td>
<td>andi $t0$, $t1$, 0xf</td>
</tr>
<tr>
<td>$C = B \mid 0xf;$</td>
<td>ori $t2$, $t1$, 0xf</td>
</tr>
<tr>
<td>$C = 5;$</td>
<td>addiu $t2$, $zero$, 5</td>
</tr>
<tr>
<td>$A = B;$</td>
<td>addiu $t0$, $t1$, 0</td>
</tr>
</tbody>
</table>

- No need for **subiu**, because **addiu** has **signed immediate**

- **$zero$** register has always the value $0$
I-Type instructions can have only 16-bit constants

What if we want to load a 32-bit constant into a register?

Can’t have a 32-bit constant in I-Type instructions 😞

The sizes of all instructions are fixed to 32 bits

Solution: use two instructions instead of one 😊

Suppose we want: $t1 = 0xAC5165D9$ (32-bit constant)

\[
\text{lui: load upper immediate} \\
\text{lui } t1, 0xAC51 \quad \text{Upper 16 bits} \\
\text{ori } t1, t1, 0x65D9 \quad \text{Lower 16 bits}
\]
Pseudo-Instructions

- Introduced by the assembler as if they were real instructions
- Facilitate assembly language programming

<table>
<thead>
<tr>
<th>Pseudo-Instruction</th>
<th>Equivalent MIPS Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>move $t1, $t2</td>
<td>addu $t1, $t2, $zero</td>
</tr>
<tr>
<td>not $t1, $t2</td>
<td>nor $t1, $t2, $zero</td>
</tr>
<tr>
<td>neg $t1, $t2</td>
<td>sub $t1, $zero, $t2</td>
</tr>
<tr>
<td>li $t1, -5</td>
<td>addiu $t1, $zero, -5</td>
</tr>
<tr>
<td>li $t1, 0xabcd1234</td>
<td>lui $t1, 0xabcd</td>
</tr>
<tr>
<td></td>
<td>ori $t1, $t1, 0x1234</td>
</tr>
</tbody>
</table>

The MARS tool has a long list of pseudo-instructions