

MIPS Arithmetic and Logic Instructions

COE 233

Logic Design and Computer Organization

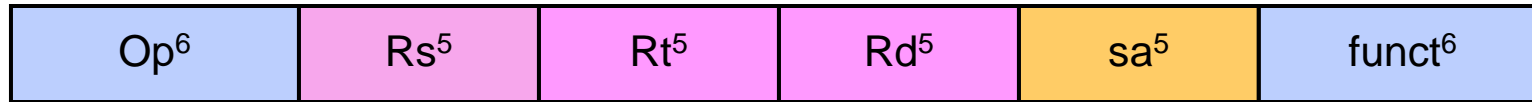
Dr. Muhamed Mudawar

King Fahd University of Petroleum and Minerals

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 numbers and registers
- ❖ Miscellaneous
 - ✧ Instructions that transfer control to/from exception handlers
 - ✧ Memory management instructions

R-Type Instruction Format



❖ **Op**: operation code (opcode)

- ✧ Specifies the operation of the instruction
- ✧ Also specifies the format of the instruction
- ✧ Up to $2^6 = 64$ opcodes → Not sufficient to define all instructions

❖ **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

Instruction	Meaning	Op	Rs	Rt	Rd	sa	func
add \$t1, \$t2, \$t3	\$t1 = \$t2 + \$t3	0	\$t2	\$t3	\$t1	0	0x20
addu \$t1, \$t2, \$t3	\$t1 = \$t2 + \$t3	0	\$t2	\$t3	\$t1	0	0x21
sub \$t1, \$t2, \$t3	\$t1 = \$t2 - \$t3	0	\$t2	\$t3	\$t1	0	0x22
subu \$t1, \$t2, \$t3	\$t1 = \$t2 - \$t3	0	\$t2	\$t3	\$t1	0	0x23

❖ **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**

Carry versus 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)

	11111	1			1	11		1
	1000	0100	0000	0000	1110	0001	0100	0001
+	1111	1111	0000	0000	1111	0101	0010	0000
<hr/>								
	1000	0011	0000	0001	1101	0110	0110	0001

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

More Examples of Carry and Overflow

❖ Example 2: Carry = 0, Overflow = 1

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

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

❖ Example 3: Carry = 1, Overflow = 1

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

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: $\$t0=f$, $\$t1=g$, $\$t2=h$, $\$t3=i$, and $\$t4=j$
- ❖ Called temporary registers: $\$t0=\8 , $\$t1=\9 , ...
- ❖ Translation of: $f = (g+h)-(i+j)$

 $\text{addu } \$t5, \$t1, \$t2 \quad \# \ \$t5 = g + h$
 $\text{addu } \$t6, \$t3, \$t4 \quad \# \ \$t6 = i + j$
 $\text{subu } \$t0, \$t5, \$t6 \quad \# \ f = (g+h)-(i+j)$
- ❖ Assembler translates $\text{addu } \$t5, \$t1, \$t2$ into binary code

Op	$\$t1$	$\$t2$	$\$t5$	sa	addu
000000	01001	01010	01101	00000	100001

Logic Bitwise Instructions

Instruction	Meaning	Op	Rs	Rt	Rd	sa	func
and \$t1, \$t2, \$t3	\$t1 = \$t2 & \$t3	0	\$t2	\$t3	\$t1	0	0x24
or \$t1, \$t2, \$t3	\$t1 = \$t2 \$t3	0	\$t2	\$t3	\$t1	0	0x25
xor \$t1, \$t2, \$t3	\$t1 = \$t2 ^ \$t3	0	\$t2	\$t3	\$t1	0	0x26
nor \$t1, \$t2, \$t3	\$t1 = ~(\$t2 \$t3)	0	\$t2	\$t3	\$t1	0	0x27

x	y	x and y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x or y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x xor y
0	0	0
0	1	1
1	0	1
1	1	0

x	y	x nor y
0	0	1
0	1	0
1	0	0
1	1	0

NOT instruction is not needed, because

not \$t1, \$t2 is equivalent to: **nor \$t1, \$t2, \$t2**

Logic Bitwise Instructions

- ❖ AND instruction is used to clear bits: $x \text{ and } 0 \rightarrow 0$
- ❖ OR instruction is used to set bits: $x \text{ or } 1 \rightarrow 1$
- ❖ XOR instruction is used to toggle bits: $x \text{ xor } 1 \rightarrow \text{not } x$

❖ Examples:

Given: $\$t1 = 0xabcd1234$ and $\$t2 = 0xffff0000$

$\text{and } \$t0, \$t1, \$t2$ $\# \$t0 = 0xabcd0000$

$\text{or } \$t0, \$t1, \$t2$ $\# \$t0 = 0xffff1234$

$\text{xor } \$t0, \$t1, \$t2$ $\# \$t0 = 0x54321234$

$\text{nor } \$t0, \$t1, \$t2$ $\# \$t0 = 0x0000edcb$

I-Type Instruction Format

- ❖ Constants are used quite frequently in programs
- ❖ I-Type: Instructions with Immediate Operands



- ❖ 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

Instruction	Meaning	Op	Rs	Rt	Immediate
addi \$t1, \$t2, 25	$\$t1 = \$t2 + 25$	0x8	\$t2	\$t1	25
addiu \$t1, \$t2, 25	$\$t1 = \$t2 + 25$	0x9	\$t2	\$t1	25
andi \$t1, \$t2, 25	$\$t1 = \$t2 \& 25$	0xc	\$t2	\$t1	25
ori \$t1, \$t2, 25	$\$t1 = \$t2 25$	0xd	\$t2	\$t1	25
xori \$t1, \$t2, 25	$\$t1 = \$t2 \wedge 25$	0xe	\$t2	\$t1	25
lui \$t1, 25	$\$t1 = 25 \ll 16$	0xf	0	\$t1	25

❖ **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 → Use negative immediate


❖ 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

Expression	Equivalent MIPS Instruction
<code>A = B + 5;</code>	<code>addiu \$t0, \$t1, 5</code>
<code>C = B - 1;</code>	<code>addiu \$t2, \$t1, -1</code>
<code>A = B & 0xf;</code>	<code>andi \$t0, \$t1, 0xf</code>
<code>C = B 0xf;</code>	<code>ori \$t2, \$t1, 0xf</code>
<code>C = 5;</code>	<code>addiu \$t2, \$zero, 5</code>
<code>A = B;</code>	<code>addiu \$t0, \$t1, 0</code>

Op = <code>addiu</code>	Rs = <code>\$t1</code>	Rt = <code>\$t2</code>	-1 = <code>0b1111111111111111</code>
-------------------------	------------------------	------------------------	--------------------------------------



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

Register `$zero` has always the value `0`

32-bit Constants

- ❖ 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 ☹**

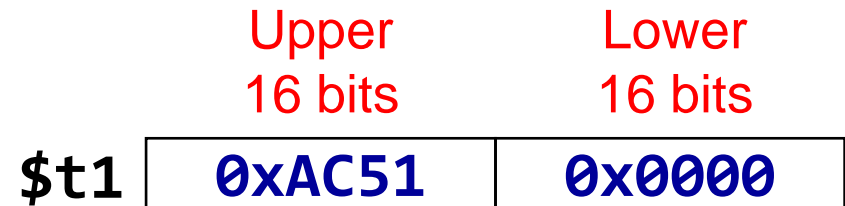
✧ The sizes of all instructions are fixed to 32 bits

- ❖ **Solution: use two instructions instead of one 😊**

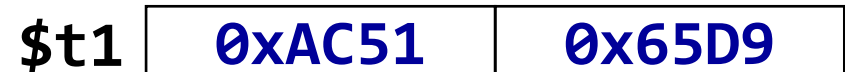
- ❖ Suppose we want: **\$t1 = 0xAC5165D9** (32-bit constant)

lui: load upper immediate

lui \$t1, 0xAC51



ori \$t1, \$t1, 0x65D9



Pseudo-Instructions

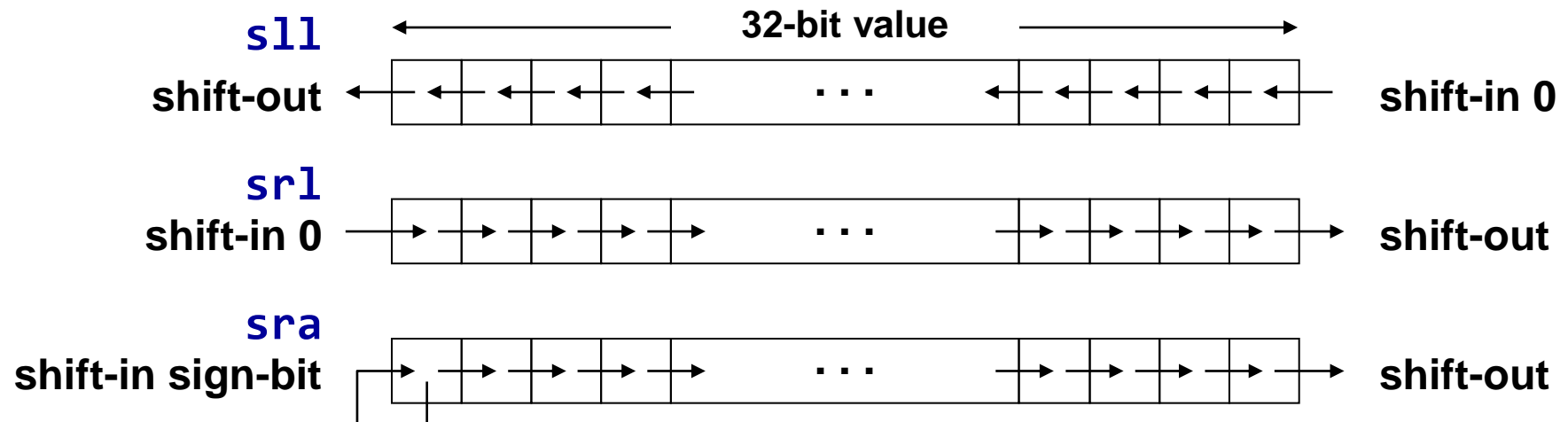
- ❖ Introduced by the assembler as if they were real instructions
- ❖ Facilitate assembly language programming

Pseudo-Instruction	Equivalent MIPS Instruction
<code>move \$t1, \$t2</code>	<code>addu \$t1, \$t2, \$zero</code>
<code>not \$t1, \$t2</code>	<code>nor \$t1, \$t2, \$zero</code>
<code>neg \$t1, \$t2</code>	<code>sub \$t1, \$zero, \$t2</code>
<code>li \$t1, -5</code>	<code>addiu \$t1, \$zero, -5</code>
<code>li \$t1, 0xabcd1234</code>	<code>lui \$t1, 0xabcd</code> <code>ori \$t1, \$t1, 0x1234</code>

The MARS tool has a long list of pseudo-instructions

Shift Operations

- ❖ Shifting is to move the 32 bits of a number left or right
- ❖ **sll** means **shift left logical** (insert zero from the right)
- ❖ **srl** means **shift right logical** (insert zero from the left)
- ❖ **sra** means **shift right arithmetic** (insert sign-bit)
- ❖ The **5-bit shift amount** field is used by these instructions



Shift Instructions (R-type only)

Instruction	Meaning	Op	Rs	Rt	Rd	sa	func
sll \$t1,\$t2,10	\$t1 = \$t2 << 10	0	0	\$t2	\$t1	10	0
srl \$t1,\$t2,10	\$t1 = \$t2 >>> 10	0	0	\$t2	\$t1	10	2
sra \$t1,\$t2,10	\$t1 = \$t2 >> 10	0	0	\$t2	\$t1	10	3
sllv \$t1,\$t2,\$t3	\$t1 = \$t2 << \$t3	0	\$t3	\$t2	\$t1	0	4
srlv \$t1,\$t2,\$t3	\$t1 = \$t2 >>>\$t3	0	\$t3	\$t2	\$t1	0	6
srav \$t1,\$t2,\$t3	\$t1 = \$t2 >> \$t3	0	\$t3	\$t2	\$t1	0	7

❖ **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 $t1, $t2, 8` $\$t1 = 0xcd123400$

`srl $t1, $t2, 4` $\$t1 = 0x0abcd123$

`sra $t1, $t2, 4` $\$t1 = 0xfabcd123$

`srlv $t1, $t2, $t3` $\$t1 = 0x0000abcd$



Op	Rs = \$t3	Rt = \$t2	Rd = \$t1	sa	srlv
000000	01011	01010	01001	00000	000110

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**

$$\text{\$t0} * 36 = \text{\$t0} * (4 + 32) = \text{\$t0} * 4 + \text{\$t0} * 32$$

```
sll  $t1, $t0, 2           # $t1 = $t0 * 4
```

```
sll  $t2, $t0, 5           # $t2 = $t0 * 32
```

```
addu $t3, $t1, $t2         # $t3 = $t0 * 36
```

Your Turn . . .

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

Hint: **26 = 2 + 8 + 16**

```
sll    $t1, $t0, 1           # $t1 = $t0 * 2
sll    $t2, $t0, 3           # $t2 = $t0 * 8
sll    $t3, $t0, 4           # $t3 = $t0 * 16
addu   $t4, $t1, $t2         # $t4 = $t0 * 10
addu   $t5, $t4, $t3         # $t5 = $t0 * 26
```

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

```
sll    $t1, $t0, 5           # $t1 = $t0 * 32
subu   $t2, $t1, $t0         # $t2 = $t0 * 31
```

Integer Multiplication in MIPS

❖ Multiply instructions

✧ `mult Rs, Rt` **Signed multiplication**

✧ `multu Rs, Rt` **Unsigned multiplication**

❖ 32-bit multiplication produces a 64-bit Product

❖ Separate pair of 32-bit registers

✧ **HI = high-order 32-bit of product**

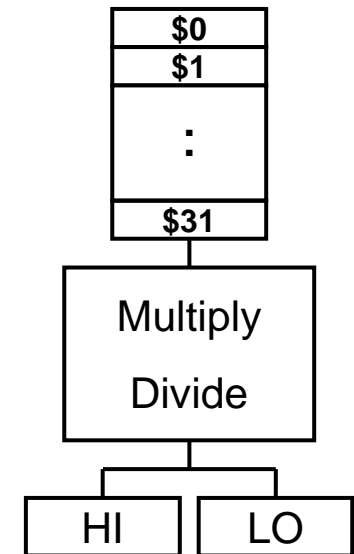
✧ **LO = low-order 32-bit of product**

❖ MIPS also has a special `mul` instruction

✧ `mul Rd, Rs, Rt` **$Rd = Rs \times Rt$**

✧ Copy **LO** into destination register **Rd**

✧ Useful when the product is small (32 bits) and **HI** is not needed



Integer Division in MIPS

❖ Divide instructions

✧ `div Rs, Rt` **Signed division**

✧ `divu Rs, Rt` **Unsigned division**

❖ Division produces quotient and remainder

❖ Separate pair of 32-bit registers

✧ **HI = 32-bit remainder**

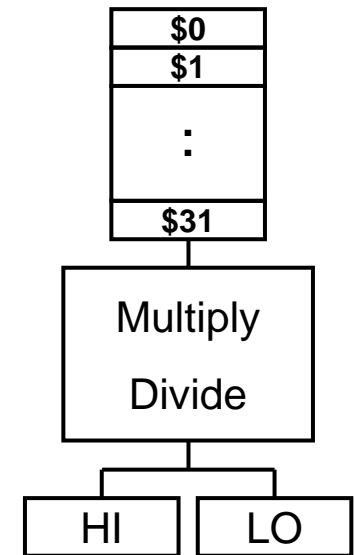
✧ **LO = 32-bit quotient**

✧ If divisor is 0 then result is **unpredictable**

❖ Moving data from **HI**, **LO** to MIPS registers

✧ `mfhi Rd` ($Rd = HI$)

✧ `mflo Rd` ($Rd = LO$)



Integer Multiply and Divide Instructions

Instruction		Meaning	Format					
mult	Rs, Rt	HI, LO = Rs \times_s Rt	Op = 0	Rs	Rt	0	0	0x18
multu	Rs, Rt	HI, LO = Rs \times_u Rt	Op = 0	Rs	Rt	0	0	0x19
mul	Rd, Rs, Rt	Rd = Rs \times_s Rt	0x1c	Rs	Rt	Rd	0	2
div	Rs, Rt	HI, LO = Rs $/_s$ Rt	Op = 0	Rs	Rt	0	0	0x1a
divu	Rs, Rt	HI, LO = Rs $/_u$ Rt	Op = 0	Rs	Rt	0	0	0x1b
mfhi	Rd	Rd = HI	Op = 0	0	0	Rd	0	0x10
mflo	Rd	Rd = LO	Op = 0	0	0	Rd	0	0x12
mthi	Rs	HI = Rs	Op = 0	Rs	0	0	0	0x11
mtlo	Rs	LO = Rs	Op = 0	Rs	0	0	0	0x13

\times_s = Signed multiplication, \times_u = Unsigned multiplication

$/_s$ = Signed division, $/_u$ = Unsigned division

NO arithmetic exception can occur

Signed Integer Division

- ❖ Simplest way is to remember the signs
- ❖ Convert the dividend and divisor to positive
 - ✧ Obtain the 2's complement if they are negative
- ❖ Do the unsigned division
- ❖ Compute the signs of the quotient and remainder
 - ✧ Quotient sign = Dividend sign XOR Divisor sign
 - ✧ Remainder sign = Dividend sign
- ❖ Negate the quotient and remainder if their sign is negative
 - ✧ Obtain the 2's complement to convert them to negative

Signed Integer Division Examples

1. **Positive** Dividend and **Positive** Divisor

✧ Example: $+17 / +3$ Quotient = $+5$ Remainder = $+2$

2. **Positive** Dividend and **Negative** Divisor

✧ Example: $+17 / -3$ Quotient = -5 Remainder = $+2$

3. **Negative** Dividend and **Positive** Divisor

✧ Example: $-17 / +3$ Quotient = -5 Remainder = -2

4. **Negative** Dividend and **Negative** Divisor

✧ Example: $-17 / -3$ Quotient = $+5$ Remainder = -2

The following equation must always hold:

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$