MIPS Arithmetic and Logic Instructions

Computer Architecture

Riad Bourbia

Computer Sciences department

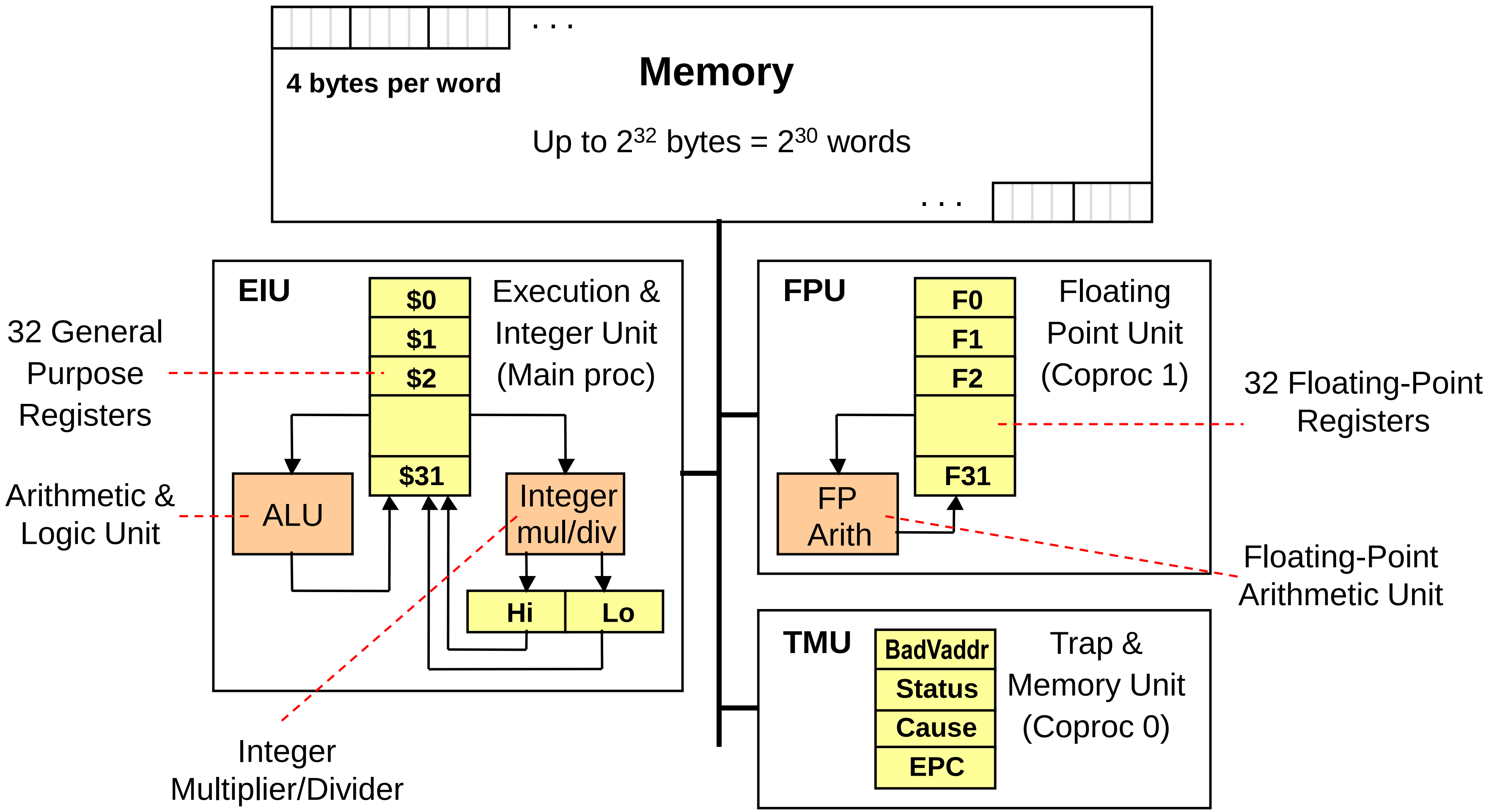
Guelma University

[Adapted from slides of Dr. A. El-maleh]

Presentation Outline

- **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
- Multiplication and Division Instructions

Overview of the MIPS Architecture



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)

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

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

R-Type Instruction Format

Op^6	Rs^5	Rt^5	Rd^5	sa^5	funct^6

- **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

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**

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)`**

```
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

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

Logic Bitwise Operations

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

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

- 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?

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

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

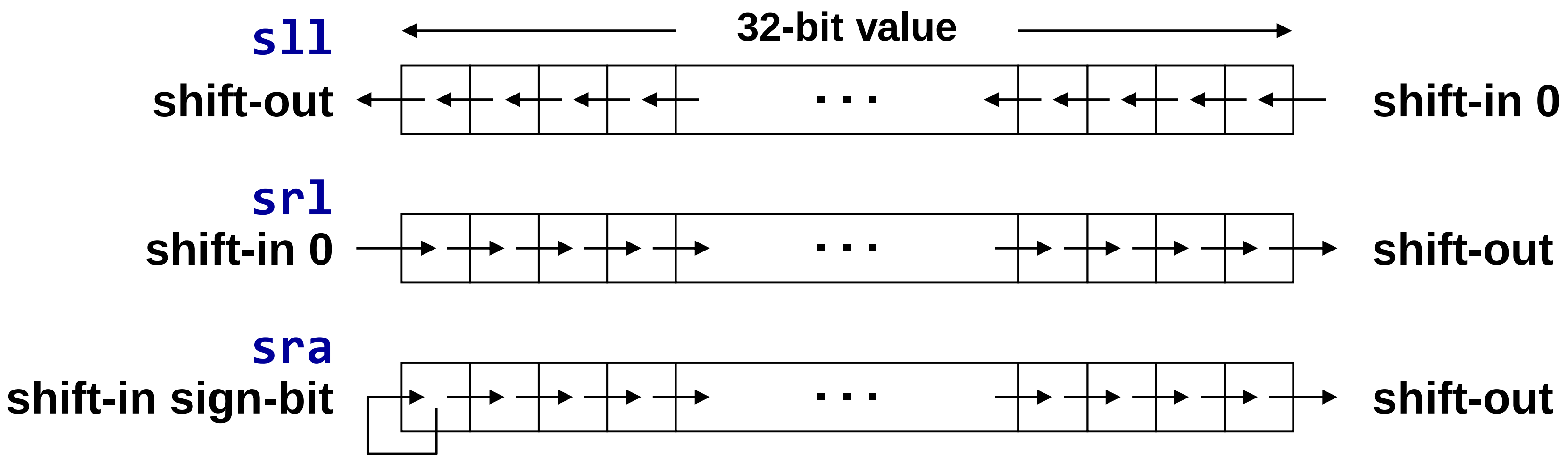
- Examples:

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

```
and $t0, $t1, $t2        # $t0 = 0xabcd0000
or  $t0, $t1, $t2        # $t0 = 0xffff1234
xor $t0, $t1, $t2        # $t0 = 0x54321234
nor $t0, $t1, $t2        # $t0 = 0x0000edcb
```

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

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**

$t0*36 = $t0*(4 + 32) = $t0*4 + $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
```

I-Type Instruction Format

- Constants 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

Op^{6}	Rs^{5}	Rt^{5}	immediate^{16}

- 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 ^ 25`	0xe	$t2	$t1	25
`lui $t1, 25`	`$t1 = 25 << 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
- 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
`A = B + 5;`	`addiu $t0, $t1, 5`
`C = B – 1;`	`addiu $t2, $t1, -1`
`A = B & 0xf;`	`andi  $t0, $t1, 0xf`
`C = B | 0xf;`	`ori   $t2, $t1, 0xf`
`C = 5;`	`addiu $t2, $zero, 5`
`A = B;`	`addiu $t0, $t1, 0`

Op = addiu	Rs = $t1	Rt = $t2	-1 = 0b1111111111111111

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

Op6	Rs5	Rt5	immediate16

- 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

		Upper 16 bits	Lower 16 bits
`lui $t1, 0xAC51`	`$t1`	`0xAC51`	`0x0000`
`ori $t1, $t1, 0x65D9`	`$t1`	`0xAC51`	`0x65D9`

Pseudo-Instructions

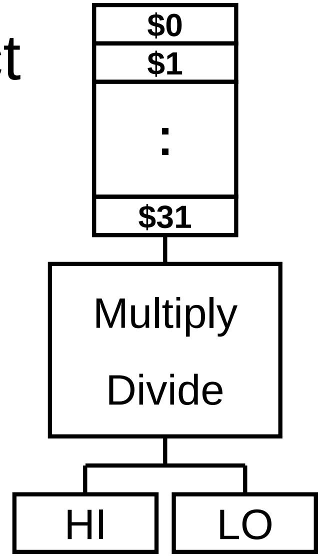
- ❖ Introduced by the assembler as if they were real instructions
- ❖ Facilitate assembly language programming

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

The MARS tool has a long list of pseudo-instructions

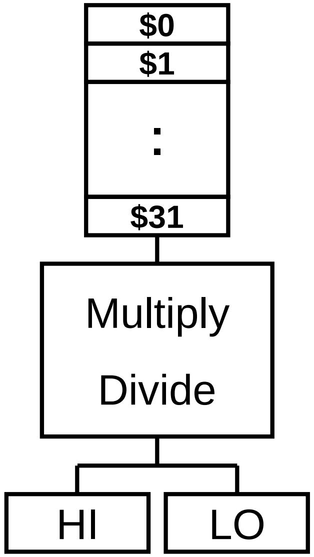
Integer Multiplication in MIPS

- Multiply instructions
  - `mult  $s1,$s2` **Signed multiplication**
  - `multu $s1,$s2` **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   $s0,$s1,$s2` **$s0 = $s1 × $s2**
  - **Put low-order 32 bits into destination register**
  - **HI & LO are undefined**



Integer Division in MIPS

- Divide instructions
  - **`div $s1,$s2`** **Signed division**
  - **`divu $s1,$s2`** **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`** (move from HI to Rd)
  - **`mflo Rd`** (move from LO to Rd)



Integer Multiply/Divide Instructions

Instruction	Meaning	Format					
mult Rs, Rt	Hi, Lo = Rs × Rt	$op^6 = 0$	Rs^5	Rt^5	0	0	0x18
multu Rs, Rt	Hi, Lo = Rs × Rt	$op^6 = 0$	Rs^5	Rt^5	0	0	0x19
mul Rd, Rs, Rt	Rd = Rs × Rt	0x1c	Rs^5	Rt^5	Rd^5	0	0x02
div Rs, Rt	Hi, Lo = Rs / Rt	$op^6 = 0$	Rs^5	Rt^5	0	0	0x1a
divu Rs, Rt	Hi, Lo = Rs / Rt	$op^6 = 0$	Rs^5	Rt^5	0	0	0x1b
mfhi Rd	Rd = Hi	$op^6 = 0$	0	0	Rd^5	0	0x10
mflo Rd	Rd = Lo	$op^6 = 0$	0	0	Rd^5	0	0x12

- Signed arithmetic: mult, div (Rs and Rt are signed)
  - LO = 32-bit low-order and HI = 32-bit high-order of multiplication
  - LO = 32-bit quotient and HI = 32-bit remainder of division
- Unsigned arithmetic: multu, divu (Rs and Rt are unsigned)
- NO arithmetic exception can occur