

MIPS Instruction Set and Assembly Language

DSC 315: Computer Organization & Operating Systems

Dr. Laltu Sardar

School of Data Science,
Indian Institute of Science Education and Research Thiruvananthapuram (IISER TVM)



February 2, 2026



Instructions

Instruction Set and Computer Language

Key Idea

To command a computer's hardware, you must speak its language.

- The **words** of a computer's language are called **instructions**.
- The **complete vocabulary** of these instructions is known as the **instruction set**.
- Hardware understands and executes only the instructions defined in its instruction set.

Example of Instruction Set and Instructions

Instruction Set

An instruction set is the complete collection of machine level commands that a processor can understand and execute.

- Example instruction set: **x86, ARMv7, ARMv8, RISC-V, MIPS**
- Each instruction set defines:
 - **Allowed operations:** Define what actions the processor can perform, such as arithmetic, logic, data movement, and control flow operations.
 - **Instruction formats:** Specify how an instruction is encoded, including the size and position of fields like opcode, registers, and immediate values.
 - **Addressing modes:** Describe how an instruction identifies the location of its operands in registers, memory, or constants.

We study MIPS Instruction Set

Program Representation Levels

High-Level Language
(C, C++, Python)

⇓ Compiler

Assembly Language
(Human-readable instructions)

⇓ Assembler

Machine Language
(Binary instructions executed by hardware)

- Programs move from abstract descriptions to hardware-executable form.
- Each level hides complexity while preserving program meaning.

Learning Flow: Assembly to Machine Instructions

Assumed Assembly Instructions

(Formats and Syntax Given)



Machine Instruction Format

(Encoding and Syntax)



Assembly to Machine Translation

(Instruction Encoding)

Example: From C to Assembly to Machine Instruction

C Code

```
1      c = a + b;
```

Assembly Code

```
1      ADD x3, x1, x2
```

Machine Instruction (Binary Encoding)

```
1      0000000 00010 00001 000 00011 0110011
```

- High level code expresses intent.
- Assembly shows explicit register operations.

Example: $d = a + b + c$

C Code

```
1      d = a + b + c;
```

Assembly Code

```
1      ADD x4, x1, x2
2      ADD x5, x4, x3  // xi's are addresses of the registers
```

Machine Instructions (Conceptual Encoding)

```
1      00000000 00010 00001 000 00100 0110011
2      00000000 00011 00100 000 00101 0110011
```

- First instruction computes $a + b$.
- Second instruction adds c to the intermediate result.
- Final value is stored as d .

Program Execution and Instructions

Key Idea

When you execute a program, you are actually executing a **sequence of instructions**. Each instruction is fetched and executed one at a time.

→ Program Counter (PC)	Instruction Array
1001 →	Instruction 1
1002 →	Instruction 2
1003 →	Instruction 3
1004 →	Instruction 4

- The Program Counter holds the address of the current instruction.
- After execution, the PC moves to the next instruction.

MIPS Instruction Operands

- MIPS instructions ' typically use **three operands**.
- Operands specify where data comes from and where results are stored.

Types of Operands in MIPS

- **Registers:** Most operations use registers, for example \$t0, \$t1, \$t2.
- **Immediate values:** Constant values encoded directly in the instruction.
- **Memory operands:** Accessed using load and store instructions with base plus offset.

Example

```
1      add $t0, $t1, $t2
2      lw  $t0, 4($t1)
```

MIPS Registers and Their Names

Register	Name	Purpose
\$zero	\$0	Constant value 0
\$at	\$1	Assembler temporary
\$v0-\$v1	\$2-\$3	Function return values
\$a0-\$a3	\$4-\$7	Function arguments
\$t0-\$t7	\$8-\$15	Temporary registers
\$s0-\$s7	\$16-\$23	Saved registers
\$t8-\$t9	\$24-\$25	More temporaries
\$k0-\$k1	\$26-\$27	Operating system use
\$gp	\$28	Global pointer
\$sp	\$29	Stack pointer
\$fp	\$30	Frame pointer
\$ra	\$31	Return address

- MIPS has registers.

General Purpose vs Special Purpose Registers

General Purpose Registers

- Used for arithmetic and logic operations
- Hold operands and intermediate results
- Freely used by programmers and compilers
- Examples in MIPS:
 - \$t0-\$t9 (temporaries)
 - \$s0-\$s7 (saved registers)

Special Purpose Registers

- Have predefined roles in program execution
- Support control flow, memory, and system functions
- Usage follows strict conventions
- Examples in MIPS:
 - \$sp (stack pointer)
 - \$ra (return address)
 - \$gp (global pointer)
 - \$zero (constant zero)

- General purpose registers focus on computation.
- Special purpose registers support program structure and control.

Special Purpose Registers in MIPS

- **\$zero** Always holds the constant value 0. Useful for comparisons and moves.
- **\$at** Reserved for the assembler. Handles large constants.
- **\$gp** Global pointer. Provides fast access to global and static data.
- **\$sp** Stack pointer. Points to the top of the runtime stack.
- **\$fp** Frame pointer. Helps access local variables and function parameters.
- **\$ra** Return address. Stores the address to return after a function call.
- **\$k0, \$k1** Reserved for operating system and exception handling.
- These registers have predefined roles by convention.
- Programmers usually avoid modifying OS reserved registers.

Immediate Values in MIPS

- Immediate values are **constant operands embedded in the instruction**.
- They remove the need to load constants from memory.
- Immediate fields have limited size due to instruction width.

Example

```
addi $t0, $t1, 10
```

- Adds constant 10 directly to the contents of \$t1.
- Result is stored in \$t0.
- 16-bits kept for constants as signed int.

Memory Operands in MIPS

- MIPS uses **load** and **store** instructions to access memory.
- **Arithmetic instructions operate only on registers.**
- Memory addresses are computed using base plus offset.

Example

```
1      lw $t0, 8($t1)
```

```
2      sw $t0, 12($t1)
```

- **lw** loads data from memory into a register.
- **sw** stores data from a register into memory.
- MIPS uses a **32-bit address space**.
- Memory is **byte addressed**.

MIPS Assembly Instructions

MIPS Arithmetic Instructions

Instruction	Example	Meaning	Comments
add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three register operands
sub	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three register operands
addi	addi \$s1,\$s2,20	$\$s1 = \$s2 + 20$	Used to add constants

MIPS Data Transfer Instructions

Instruction	Example	Meaning	Comments
lw	lw \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Word memory to register
sw	sw \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Word register to memory
lh	lh \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword to register
lhu	lhu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword unsigned
sh	sh \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Halfword to memory
lb	lb \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte to register
lbu	lbu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte unsigned
sb	sb \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Byte to memory
ll	ll \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Atomic load
sc	sc \$s1,20(\$s2)	$\text{Memory}[\dots] = \$s1; \$s1=0 \text{ or } 1$	Atomic store
lui	lui \$s1,20	$\$s1 = 20 \times 2^{16}$	Load upper immediate

MIPS Logical Instructions

Instruction	Example	Meaning	Comments
and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Bitwise AND
or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \mid \$s3$	Bitwise OR
nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim(\$s2 \mid \$s3)$	Bitwise NOR
andi	andi \$s1,\$s2,20	$\$s1 = \$s2 \& 20$	AND with constant
ori	ori \$s1,\$s2,20	$\$s1 = \$s2 \mid 20$	OR with constant
sll	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Shift left logical
srl	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Shift right logical

MIPS Conditional Branch Instructions

Instruction	Example	Meaning	Comments
beq	beq \$s1,\$s2,25	if equal branch	PC relative branch
bne	bne \$s1,\$s2,25	if not equal branch	PC relative branch
slt	slt \$s1,\$s2,\$s3	$\$s1=1$ if $\$s2 < \$s3$	Signed compare
sltu	sltu \$s1,\$s2,\$s3	$\$s1=1$ if $\$s2 < \$s3$	Unsigned compare
slti	slti \$s1,\$s2,20	$\$s1=1$ if $\$s2 < 20$	Immediate compare
sltiu	sltiu \$s1,\$s2,20	$\$s1=1$ if $\$s2 < 20$	Unsigned immediate

MIPS Unconditional Jump Instructions

Instruction	Example	Meaning	Comments
j	j 2500	go to target	Jump instruction
jr	jr \$ra	go to \$ra	Procedure return
jal	jal 2500	\$ra = PC+4; jump	Procedure call

MIPS Unconditional Jump Instructions

Important Note to Students

- You do **not** need to memorize individual instructions.
- Focus on understanding instruction categories and usage.
- Learn how operands and addressing work.
- In examinations, the required set of instructions will be provided.

Example: C to Assembly Translation

C Code

```
1      f = (g + h) - (i + j);
```

Assumed Register Mapping

■ \$s1 = g \$s2 = h \$s3 = i \$s4 = j \$s0 = f

Assembly Code

```
1      add $t0, $s1, $s2
2      add $t1, $s3, $s4
3      sub $s0, $t0, $t1
```

- First addition computes $g + h$.
- Second addition computes $i + j$.
- Final subtraction produces the result f .

What will happen when the addresses are in the memory?

Computer		Programmers		
Address	Content	Name	Type	Value
90000000	00	sum	int (4 bytes)	000000FF (255 ₁₀)
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	age	short (2 bytes)	FFFF (-1 ₁₀)
90000005	FF			
90000006	1F			
90000007	FF			
90000008	FF	average	double (8 bytes)	1FFFFFFFFFFFFFFFFF (4.45015E-308 ₁₀)
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90			
9000000F	00			
90000010	00	ptrSum	int* (4 bytes)	90000000
90000011	00			

Note: All numbers in hexadecimal

Memory Address Rules

- Memory is **byte addressed**.
- Each memory address refers to exactly **one byte**.
- A word consists of **4 consecutive bytes**.

Alignment Rules

- Word addresses must be **aligned**.
- Word access addresses are multiples of 4.
- Misaligned accesses are either slow or not allowed.

Address Calculation

- Effective address = base register + offset.
- Offset is specified in bytes.



Example: $g = h + A[8]$

```
1      g = h + A[8];
```

Assumed Register Mapping

- $\$s1 = h$; $\$s0 = g$
- $\$s2 =$ base address of array A

Assembly Code

```
1      lw    $t0, 32($s2)
2      add   $s0, $s1, $t0
```

- Each array element is a word of 4 bytes.
- Index 8 corresponds to offset $8 \times 4 = 32$ bytes.
- The value of $A[8]$ is loaded first, then added to h .

Example: $A[12] = h + A[8]$

C Code

```
A[12] = h + A[8];
```

Assumed Register Mapping

- $\$s1 = h$
- $\$s2 = \text{base address of array } A$

Assembly Code

```
1      lw    $t0, 32($s2)
2      add   $t1, $s1, $t0
3      sw    $t1, 48($s2)
```

- Each array element occupies 4 bytes.
- $A[8]$ uses offset $8 \times 4 = 32$ bytes.
- $A[12]$ uses offset $12 \times 4 = 48$ bytes.
- Value is loaded, computed, and stored back to memory.

Problem: If-Else in Assembly

High Level Code

```
1      if (i == j)
2          f = g + h;
3      else
4          f = g - h;
```

Assumptions

- $\$s0 = i$ $\$s1 = j$
- $\$s2 = g$ $\$s3 = h$
- $\$s4 = f$

Solution: If-Else in Assembly

Assembly Code

```
1      bne $s0, $s1, Else
2      add $s4, $s2, $s3
3      j    EndIf
4      Else:
5      sub $s4, $s2, $s3
6      EndIf:
```

- Assembly has no if or else keywords.
- Control flow is created using branches and labels.
- **ELSE** is a label for the false branch.
- **ENDIF** is a label where both paths rejoin.
- Conditional branches jump to ELSE.
- Unconditional jumps skip ELSE and go to ENDIF.

Problem: While Loop in Assembly

High Level Code

```
1      while (A[i] != v)
2          i++;
```

Assumptions

- \$s0 = base address of array A
- \$s1 = i (loop index)
- \$s2 = v (search value)
- Each array element is a word (4 bytes)

Solution: While Loop in Assembly

Assembly Code

```
1      Loop:
2      sll $t0, $s1, 2
3      add $t0, $s0, $t0
4      lw  $t1, 0($t0)
5      beq $t1, $s2, Exit
6      addi $s1, $s1, 1
7      j   Loop
8      Exit:
```

- Index is scaled by 4 to get byte offset.
- Value $A[i]$ is loaded from memory.
- Loop continues until $A[i]$ equals v .
- Control exits when the condition becomes false.

Difference Between slt and sltu

Given Values (8-bit Registers): $a = 1000\ 0001$ $b = 0000\ 0001$

Assumptions:

- $\$s1 = a = 1000\ 0001$
- $\$s2 = b = 0000\ 0001$

```
1        slt    $t0, $s1, $s2  
2        sltu   $t1, $s1, $s2
```

What will be the outputs and why?

- **slt** treats operands as signed integers
- **sltu** treats operands as unsigned integers

Function in MIPS Assembly Instructions

Assembly Code for Function example()

C Function

```
1      int example(int a, int b, int c, int d){  
2          int s;  
3          s = (a+b) - (c+d);  
4          return s;  
5      }
```

Assumptions (MIPS Calling Convention)

- $\$a0 = a$, $\$a1 = b$, $\$a2 = c$, $\$a3 = d$
- Return value in $\$v0$
- Temporary registers $\$t0$, $\$t1$ used

Assembly Code for Function example()

```

1  int example(int a, int b, int c, int d){
2      int s;
3      s = (a+b)-(c+d);
4      return s;
5  }

```

```

1      example:
2      add  $t0, $a0, $a1
3      add  $t1, $a2, $a3
4      sub  $v0, $t0, $t1
5      jr   $ra

```

- 1 Since they a, b, c, d are function parameters `$a0` register is used
- 2 `$ra` is special register (for return value)

Problem solving in Assembly

Unsigned 64-bit Addition in MIPS32

Problem

MIPS32 supports only 32-bit registers, but we want to add two unsigned 64-bit integers.

Representation:

$$A = (A_{high} \ll 32) \mid A_{low} \quad B = (B_{high} \ll 32) \mid B_{low}$$

Register assignment:

- $\$t0 = A_low$; $\$t1 = A_high$
- $\$t2 = B_low$; $\$t3 = B_high$

Algorithm:

- 1 Add lower 32 bits
- 2 Detect carry
- 3 Add upper 32 bits and carry

Code for Unsigned 64-bit Addition in MIPS32

MIPS Assembly:

```
1      addu    $t4, $t0, $t2      # result_low
2      sltu    $t6, $t4, $t0      # carry
3      addu    $t5, $t1, $t3      # result_high
4      addu    $t5, $t5, $t6      # add carry
```

Signed 64-bit Addition in MIPS32

Problem

MIPS32 supports only 32-bit registers, but we want to add two **signed** 64-bit integers.

Observation: Signed 64-bit integers use two's complement. Carry propagation is the same as unsigned addition.

Steps:

- 1 Perform 64-bit addition using two 32-bit words
- 2 Detect signed overflow using sign bits

MIPS Assembly (Addition):

```
1      addu    $t4, $t0, $t2      # low word
2      sltu    $t6, $t4, $t0      # carry
3      addu    $t5, $t1, $t3      # high word
4      addu    $t5, $t5, $t6
```

Signed 64-bit Addition in MIPS32

Signed Overflow Detection:

$$\text{Overflow} = (\text{sign}_A = \text{sign}_B) \wedge (\text{sign}_{\text{result}} \neq \text{sign}_A)$$

```

1      srl    $t7, $t1, 31          # sign A
2      srl    $t8, $t3, 31          # sign B
3      srl    $t9, $t5, 31          # sign result
4      xor    $t7, $t7, $t8
5      bne    $t7, $zero, no_overflow
6      xor    $t7, $t9, $t1
7      andi   $t7, $t7, 1
8      bne    $t7, $zero, overflow

```


Unsigned Multiplication in MIPS32

Observation:

MIPS32 registers are 32-bit wide, but multiplying two unsigned 32-bit integers produces a 64-bit result.

Hardware support:

- `multu` performs unsigned multiplication
- Result is stored in special registers
 - LO : lower 32 bits
 - HI : upper 32 bits

Mathematical form:

$$P = A \times B, \quad 0 \leq P < 2^{64}$$

Unsigned Multiplication: Registers and Algorithm

Register assignment:

- \$t0 = A (unsigned 32-bit)
- \$t1 = B (unsigned 32-bit)
- \$t2 = result_low
- \$t3 = result_high

Algorithm:

- 1 Use multu to multiply operands
- 2 Move lower 32 bits from L0
- 3 Move upper 32 bits from HI

No overflow exception occurs for unsigned multiplication.

Unsigned Multiplication: MIPS Assembly Code

MIPS instructions:

```
1      multu $t0, $t1      # unsigned multiply
2      mflo   $t2           # result_low
3      mfhi   $t3           # result_high
```

Result representation:

$$\text{Product} = (\text{result_high} \ll 32) \mid \text{result_low}$$

This gives the full 64-bit unsigned product.

Addition vs Multiplication in MIPS32

Operation	Instruction	Result Handling
Unsigned Add	<code>addu</code>	Carry propagated manually
Unsigned Mul	<code>multu</code>	Result in HI and LO

Key difference:

- Addition uses general-purpose registers only
- Multiplication uses special HI and LO registers

Starting a program

What Does Starting a Program Mean?

Starting a program means preparing the computer hardware to execute instructions.

- Program must be placed in memory
- CPU must know where execution begins
- Registers must be initialized

Goal

Enable correct and controlled execution of instructions

Program as a Sequence of Instructions

A program consists of:

- Machine instructions
- Data used by instructions

These are stored in main memory before execution begins.

Key Point

CPU executes instructions sequentially from memory

Loading a Program into Memory

Before execution:

- Program is stored on secondary storage
- Operating system loads it into main memory
- A starting address is assigned

Loader

Responsible for placing program into memory

Role of the Program Counter (PC)

Program Counter

PC is a special register that holds the address of the next instruction.

- Updated after each instruction
- Controls the flow of execution

Starting a program means initializing the PC.

Initializing the Program Counter

- Loader sets PC to the first instruction address
- This address is called the entry point
- CPU fetches instruction from PC

Important

Without correct PC value, program cannot start correctly

Instruction Fetch Cycle

Once PC is initialized:

- 1 Fetch instruction from memory
- 2 Decode instruction
- 3 Execute instruction
- 4 Update PC

This cycle continues until program ends.

Why Do We Need a Stack?

Programs require temporary storage for:

- Function calls
- Local variables
- Return addresses

Solution

Use a stack data structure

Stack and Stack Pointer

Stack Pointer (SP)

SP holds the address of the top of the stack.

- Stack grows in a fixed direction
- SP is initialized before program execution

Correct stack setup is essential for procedure calls.

Memory Layout of a Program

A typical program memory layout includes:

- Code segment
- Data segment
- Heap
- Stack

Observation

Different regions serve different purposes

Stack vs Heap

Aspect	Stack	Heap
Purpose	Function calls, local variables	Dynamic memory allocation
Memory management	Automatic	Manual or managed
Access speed	Very fast	Slower than stack
Allocation pattern	LIFO	Arbitrary
Lifetime	Limited to function scope	Until explicitly freed
Fragmentation	No fragmentation	Possible fragmentation
Pointer used	Stack Pointer (SP)	Heap Pointer

First Instruction Execution

After initialization:

- PC points to first instruction
- Registers have defined values
- Stack is ready

CPU begins execution using the fetch decode execute cycle.

MIPS Instructions

Bitwise Layout of MIPS Instruction Formats

R-Type Instruction (32 bits)					
opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-Type Instruction (32 bits)			
opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

J-Type Instruction (32 bits)	
opcode	address
6 bits	26 bits

R-Type Instruction Format

R-type instructions are used for register-to-register arithmetic and logical operations.

Opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- **Opcode** Identifies the instruction type. For all R-type instructions, this field is set to 000000.
- **rs (source register)** Holds the number of the first source register.
- **rt (target register)** Holds the number of the second source register.
- **rd (destination register)** Specifies the register where the result is stored.
- **shamt (shift amount)** Used only for shift instructions to specify the number of bit positions to shift. Set to zero for non-shift operations.
- **funct (function code)** Determines the exact operation to be performed, such as add, sub, and, or.

R-Type Instruction Example: add

Consider the instruction:

add \$t0, \$t1, \$t2

This computes:

$$\$t0 \leftarrow \$t1 + \$t2$$

Opcode	rs	rt	rd	shamt	funct
000000	01001	01010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- **Opcode = 000000** Indicates an R-type instruction.
- **rs = \$t1 (01001)** First source register.
- **rt = \$t2 (01010)** Second source register.
- **rd = \$t0 (01000)** Destination register where the result is stored.
- **shamt = 00000** Not used for addition.
- **funct = 100000** Specifies the add operation.

I-Type Instruction Format

I-type instructions are used for immediate operations, load, store, and conditional branch instructions.

Opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- **Opcode** Specifies the instruction category such as addi, lw, sw, beq.
- **rs (source register)** Contains the first operand or base register.
- **rt (target register)** Destination register for immediate and load instructions, or source register for store and branch.
- **Immediate** A 16-bit constant or offset. It is sign-extended for arithmetic and branch instructions.

I-Type Instruction Example: addi

Consider the instruction:

```
addi $t0, $t1, 10
```

This computes:

$$\$t0 \leftarrow \$t1 + 10$$

Opcode	rs	rt	immediate
001000	01001	01000	0000 0000 0000 1010
6 bits	5 bits	5 bits	16 bits

- **Opcode = 001000** Identifies the addi instruction.
- **rs = \$t1 (01001)** Source register.
- **rt = \$t0 (01000)** Destination register.
- **Immediate = 10** Constant value added to the source register.

J-Type Instruction Format

J-type instructions are used for unconditional control transfer.

Opcode	address
6 bits	26 bits

- **Opcode** Specifies jump-type instructions such as j and jal.
- **Address** Contains the target address divided by 4. The actual jump address is formed using PC upper bits and this field.

J-Type Instruction Example: j

Consider the instruction:

j target

This causes an unconditional jump to the address labeled target.

Opcode	address
000010	target[25:0]
6 bits	26 bits

- **Opcode = 000010** Identifies the jump instruction.
- **Address field** Specifies the jump target location after left shifting by 2 bits and combining with PC upper bits.

Other Instruction Types and Their Formats

Instruction Type	Example	Format	Opcode
Load Word	lw \$t0, 4(\$t1)	I-type	100011
Store Word	sw \$t0, 4(\$t1)	I-type	101011
Branch if Equal	beq \$t0, \$t1, L	I-type	000100
Branch if Not Equal	bne \$t0, \$t1, L	I-type	000101
Add Immediate	addi \$t0, \$t1, imm	I-type	001000
AND Immediate	andi \$t0, \$t1, imm	I-type	001100
OR Immediate	ori \$t0, \$t1, imm	I-type	001101
Set Less Than Immediate	slti \$t0, \$t1, imm	I-type	001010
Jump	j target	J-type	000010
Jump and Link	jal target	J-type	000011
Shift Left Logical	sll \$t0, \$t1, sh	R-type	000000
Set Less Than	slt \$t0, \$t1, \$t2	R-type	000000

Opcode alone identifies the instruction class. For R-type instructions, the exact operation is selected using the funct field.

Load Instruction Example: lw

Instruction

lw \$t0, 8(\$t1)

Operation

$\$t0 \leftarrow \text{Memory}[\$t1 + 8]$

Opcode	rs	rt	immediate
100011	01001	01000	0000 0000 0000 1000
6 bits	5 bits	5 bits	16 bits

Store Instruction Example: sw

Instruction

sw \$t0, 12(\$t1)

Operation

Memory[\$t1 + 12] \leftarrow \$t0

Opcode	rs	rt	immediate
101011	01001	01000	0000 0000 0000 1100
6 bits	5 bits	5 bits	16 bits

Logical Instruction Example: and

Instruction

and \$t0, \$t1, \$t2

Operation

$\$t0 \leftarrow \$t1 \text{ AND } \$t2$

Opcode	rs	rt	rd	shamt	funct
000000	01001	01010	01000	00000	100100
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Logical Instruction Example: and

Instruction

and \$t0, \$t1, \$t2

Operation

$\$t0 \leftarrow \$t1 \text{ AND } \$t2$

Opcode	rs	rt	rd	shamt	funct
000000	01001	01010	01000	00000	100100
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Logical Instruction Example: or

Instruction

or \$t0, \$t1, \$t2

Operation

$\$t0 \leftarrow \$t1 \text{ OR } \$t2$

Opcode	rs	rt	rd	shamt	funct
000000	01001	01010	01000	00000	100101
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Logical Instruction Example: not

Instruction

```
nor $t0, $t1, $zero
```

Operation

$$\$t0 \leftarrow \text{NOT}(\$t1)$$

Opcode	rs	rt	rd	shamt	funct
000000	01001	00000	01000	00000	100111
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Branch Instruction Example: beq

Instruction

beq \$t0, \$t1, LABEL

Operation

if $\$t0 = \$t1$ then $PC \leftarrow LABEL$

Opcode	rs	rt	immediate
000100	01000	01001	offset
6 bits	5 bits	5 bits	16 bits

Branch Instruction Example: bne

Instruction

bne \$t0, \$t1, LABEL

Operation

if $\$t0 \neq \$t1$ then $PC \leftarrow LABEL$

Opcode	rs	rt	immediate
000101	01000	01001	offset
6 bits	5 bits	5 bits	16 bits

Comparison Instruction Example: slti

Instruction

```
slti $t0, $t1, 5
```

Operation

if $\$t1 < 5$ then $\$t0 \leftarrow 1$

Opcode	rs	rt	immediate
001010	01001	01000	0000 0000 0000 0101
6 bits	5 bits	5 bits	16 bits

Instructions in one table

Instruction	Type	Opcode	rs	rt	rd	shamt	funct / imm
lw \$t0, 8(\$t1)	I	100011	01001	01000	–	–	imm = 8
sw \$t0, 12(\$t1)	I	101011	01001	01000	–	–	imm = 12
and \$t0, \$t1, \$t2	R	000000	01001	01010	01000	00000	funct = 100100
or \$t0, \$t1, \$t2	R	000000	01001	01010	01000	00000	funct = 100101
nor \$t0, \$t1, \$zero	R	000000	01001	00000	01000	00000	funct = 100111
beq \$t0, \$t1, LABEL	I	000100	01000	01001	–	–	offset
bne \$t0, \$t1, LABEL	I	000101	01000	01001	–	–	offset
slti \$t0, \$t1, 5	I	001010	01001	01000	–	–	imm = 5

R-type instructions use the funct field, I-type instructions use the immediate / offset field.

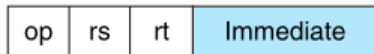
Addressing Modes

Section 2.10 in Book [1]

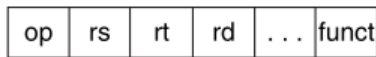
Addressing Modes

- An addressing mode defines how an instruction locates its operands.
- It specifies how the effective address or operand value is obtained.
- Addressing modes are part of the instruction set architecture.
- MIPS uses a small and simple set of addressing modes.

1. Immediate addressing



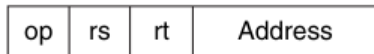
2. Register addressing



Registers

Register

3. Base addressing



Memory

Register

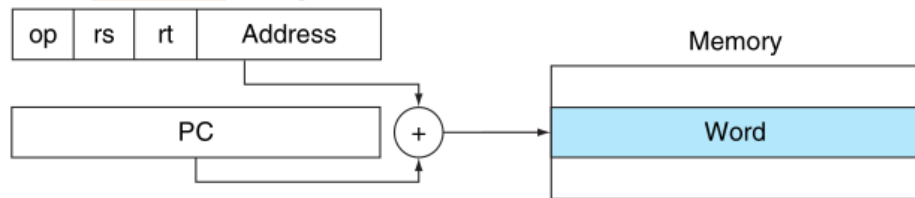
+

Byte

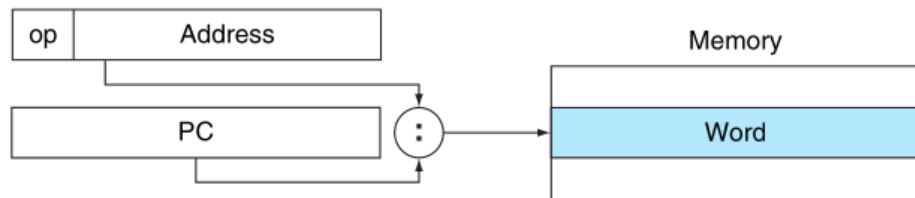
Halfword

Word

4. PC-relative addressing



5. Pseudodirect addressing



Register Addressing Mode

- Operands are stored in registers.
- Fastest addressing mode.
- Used by arithmetic and logical instructions.

Example

```
add $s0, $s1, $s2
```


Immediate Addressing Mode

- One operand is a constant value.
- Constant is encoded inside the instruction.
- Reduces memory accesses.

Example

```
addi $s0, $s1, 10
```

Base Plus Offset Addressing Mode

- Used for memory access.
- Effective address = base register + offset.
- Offset is specified in bytes.

Example

```
lw $t0, 16($s1)
```

PC Relative Addressing Mode

- Used by branch instructions.
- Target address is relative to the Program Counter.
- Supports position independent code.

Example

```
1      beq $s0, $s1, LABEL
```

Pseudodirect Addressing Mode

- Used by jump instructions.
- The instruction contains part of the target address.
- Remaining bits are taken from the Program Counter.

How the Target Address Is Formed

- Upper bits come from $PC + 4$.
- Lower 26 bits come from the instruction.
- Two zero bits are appended at the end.

Example

```
1      j    LABEL
2      jal  FUNCTION
```

Summary of Addressing Modes in MIPS

- Register addressing
- Immediate addressing
- Base plus offset addressing
- PC relative addressing
- Pseudodirect Addressing Mode

Important Note

- You do not need to memorize addressing modes.
- Focus on understanding how operands are accessed.
- Required details will be provided in examinations if needed.

Character Manipulation

Character Manipulation

- Characters are stored as **ASCII values**.
- Each character occupies **1 byte**.
- Character manipulation is done using byte load and store instructions.
- ASCII characters are manipulated as integers.
- No separate character type exists at hardware level.

ASCII Table (Common Characters)

Character	Decimal	Hex	Binary
'0'	48	0x30	0011 0000
'9'	57	0x39	0011 1001
'A'	65	0x41	0100 0001
'Z'	90	0x5A	0101 1010
'a'	97	0x61	0110 0001
'z'	122	0x7A	0111 1010
Space	32	0x20	0010 0000
Newline	10	0x0A	0000 1010

- Characters are stored as numeric ASCII codes.
- Uppercase and lowercase letters differ by 32.
- Characters are manipulated as integers in assembly.

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(72	48	110	H	104	68	150	h
9	9	11		41	29	51)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	
29	1D	35		61	3D	75	=	93	5D	135]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Loading and Storing Characters

- Use byte instructions to access characters.
- Memory is byte addressed.

Instructions

1 `lb $t0, 0($s0)`

2 `sb $t0, 0($s0)`

- `lb` loads a signed byte.
- `sb` stores the least significant byte.

Unsigned Character Access

- Some characters have ASCII values greater than 127.
- Unsigned load avoids sign extension.

Instruction

```
1      lbu $t0, 0($s0)
```

- lbu zero extends the loaded byte.
- Commonly used for character comparison.

Character Comparison Example

C Code

```
1      if (ch == 'A')  
2      flag = 1;
```

Assembly Code

```
1      lbu    $t0, 0($s0)  
2      addi   $t1, $zero, 65  
3      beq    $t0, $t1, Equal
```

- Characters are compared using their ASCII codes.

Character Case Conversion Example

- Uppercase and lowercase letters differ by 32 in ASCII.

Example: Convert Uppercase to Lowercase

```
1      lbu    $t0, 0($s0)
2      addi   $t0, $t0, 32
3      sb     $t0, 0($s0)
```

- Works only if the character is already uppercase.


- 1 Complete the adding unsigned integer
- 2 While loop of $A[i] == i$; 30
- 3 Writing function 34
- 4 Instruction format; reason
- 5 Addressing modes 76
- 6 starting a program.45
- 7 Possible number of supporting memory; looking at the format.



Dr. Laltu Sardar, Assistant Professor, IISER Thiruvananthapuram

`laltu.sardar@iisertvm.ac.in`, `laltu.sardar.crypto@gmail.com`

Course webpage: https://laltu-sardar.github.io/courses/corgos_2026.html.

-  Carl Hamacher, Zvonko Vranesic, and Safwat Zaky.
Computer Organization.
McGraw-Hill, 6th edition, 2012.
-  John L. Hennessy and David A. Patterson.
Computer Architecture: A Quantitative Approach.
Morgan Kaufmann, 6th edition, 2017.
-  Vincent P. Heuring and Harry F. Jordan.
Computer System Design and Architecture.
Pearson, 2nd edition, 2007.
-  David A. Patterson and John L. Hennessy.
Computer Organization and Design: The Hardware/Software Interface.
Morgan Kaufmann, 4th edition, 2014.
-  William Stallings.
Computer Organization and Architecture: Designing for Performance.
Pearson, 10th edition, 2016.