Complete MIPS Assembly Reference Guide for Lab Exams

Table of Contents

---

1. Basic Program Structure

Minimal MIPS Program Template

```
.data
    # Global variables and constants go here

.text
.globl main
main:
    # Your code here
```

```
    # Always exit properly
    li $v0, 10
    syscall
```

Program Sections

- **.data**: Global variables, strings, arrays
- **.text**: Executable code
- **.globl main**: Makes main function globally visible

---

2. Data Types and Memory

Data Declaration Directives

```
.data
# Integers
num1:       .word 42                  # 32-bit integer
array:      .word 1, 2, 3, 4, 5       # Integer array
big_array:  .space 400                # Reserve 400 bytes (100 integers)

# Smaller data types
byte_val:   .byte 0xFF                # 8-bit value
half_val:   .half 1024                # 16-bit value

# Strings
message:    .asciiz "Hello World!"    # Null-terminated string
buffer:     .space 100                # String buffer (100 chars)

# Floating point
pi:         .float 3.14159            # Single precision
e:          .double 2.718281828       # Double precision

# Constants
.eqv SIZE 10                          # Define constant SIZE = 10
.eqv NEWLINE 0x0A                     # ASCII newline
```

Memory Access Instructions

```
# Load operations
lw $t0, address        # Load word (32-bit)
lh $t0, address        # Load halfword (16-bit, sign extended)
lhu $t0, address       # Load halfword unsigned
lb $t0, address        # Load byte (8-bit, sign extended)
lbu $t0, address       # Load byte unsigned

# Store operations
sw $t0, address        # Store word
sh $t0, address        # Store halfword
sb $t0, address        # Store byte

# Address modes
lw $t0, label          # Load from label address
lw $t0, 0($s0)         # Load from address in $s0
lw $t0, 4($s0)         # Load from $s0 + 4 (offset)
```

Memory Alignment

- **Words** (4 bytes): Must be aligned to addresses divisible by 4
- **Halfwords** (2 bytes): Must be aligned to addresses divisible by 2
- **Bytes**: No alignment required

---

3. Registers and Conventions

Register Categories

```
# Zero register
$zero ($0)           # Always contains 0, cannot be changed

# Return values
$v0-$v1 ($2-$3)       # Function return values, syscall codes

# Arguments
$a0-$a3 ($4-$7)       # Function arguments (first 4)
```

```
# Temporary registers (caller-saved)
$t0-$t9 ($8-$15, $24-$25)  # Not preserved across function calls

# Saved registers (callee-saved)
$s0-$s7 ($16-$23)      # Must be preserved across function calls

# Special purpose
$gp ($28)            # Global pointer
$sp ($29)            # Stack pointer
$fp ($30)            # Frame pointer
$ra ($31)            # Return address
```

Register Usage Guidelines

```
# Good practice examples
move $s0, $a0         # Save argument in saved register
li $t0, 100          # Use temporary for immediate calculations
jal function         # $ra automatically saved by jal
```

---

4. System Calls (Syscalls)

Essential Syscalls

```
# Print integer
li $v0, 1            # Service code
move $a0, $t0         # Integer to print
syscall

# Print string
li $v0, 4            # Service code
la $a0, message       # Address of string
syscall

# Read integer
li $v0, 5            # Service code
syscall              # Result in $v0
```

```
# Read string
li $v0, 8            # Service code
la $a0, buffer       # Buffer address
li $a1, 100          # Maximum length
syscall

# Print character
li $v0, 11           # Service code
li $a0, 'A'          # Character to print
syscall

# Read character
li $v0, 12           # Service code
syscall              # Result in $v0

# Exit program
li $v0, 10           # Service code
syscall
```

Floating Point Syscalls

```
# Print float
li $v0, 2            # Service code
mov.s $f12, $f0      # Float value in $f12
syscall

# Read float
li $v0, 6            # Service code
syscall              # Result in $f0

# Print double
li $v0, 3            # Service code
mov.d $f12, $f0      # Double value in $f12
syscall

# Read double
```

```
li $v0, 7            # Service code
syscall              # Result in $f0
```

Memory Allocation Syscall

```
# Allocate memory (sbrk)
li $v0, 9            # Service code
li $a0, 100          # Number of bytes to allocate
syscall              # Address of allocated memory in $v0
```

---

5. Arithmetic and Logic Operations

Basic Arithmetic

```
# Addition
add $t0, $s1, $s2      # $t0 = $s1 + $s2
addi $t0, $s1, 10      # $t0 = $s1 + 10 (immediate)
addu $t0, $s1, $s2     # Unsigned addition

# Subtraction
sub $t0, $s1, $s2      # $t0 = $s1 - $s2
subu $t0, $s1, $s2     # Unsigned subtraction

# Multiplication
mul $t0, $s1, $s2      # $t0 = $s1 * $s2 (32-bit result)
mult $s1, $s2          # Full multiply: result in HI:LO
mfhi $t0               # Move from HI register
mflo $t1               # Move from LO register

# Division
div $s1, $s2           # $s1 / $s2: quotient→LO, remainder→HI
mflo $t0               # $t0 = quotient
mfhi $t1               # $t1 = remainder

# Optimized division by powers of 2
```

```
srl $t0, $s1, 2        # $t0 = $s1 / 4 (right shift by 2)
sll $t0, $s1, 3        # $t0 = $s1 * 8 (left shift by 3)
```

Logical Operations

```
# Bitwise operations
and $t0, $s1, $s2      # $t0 = $s1 & $s2
andi $t0, $s1, 0xFF    # $t0 = $s1 & 0xFF
or $t0, $s1, $s2       # $t0 = $s1 | $s2
ori $t0, $s1, 0x10     # $t0 = $s1 | 0x10
xor $t0, $s1, $s2      # $t0 = $s1 ^ $s2
nor $t0, $s1, $s2      # $t0 = ~($s1 | $s2)

# Bit shifts
sll $t0, $s1, 2        # Shift left logical by 2
srl $t0, $s1, 2        # Shift right logical by 2
sra $t0, $s1, 2        # Shift right arithmetic by 2
```

Comparison Operations

```
# Set on less than
slt $t0, $s1, $s2      # $t0 = 1 if $s1 < $s2, else 0
slti $t0, $s1, 10      # $t0 = 1 if $s1 < 10, else 0
sltu $t0, $s1, $s2     # Unsigned comparison

# Pseudo-instructions for convenience
sge $t0, $s1, $s2      # $t0 = 1 if $s1 >= $s2
seq $t0, $s1, $s2      # $t0 = 1 if $s1 == $s2
sne $t0, $s1, $s2      # $t0 = 1 if $s1 != $s2
```

---

6. Control Flow (Branches and Loops)

Conditional Branches

```
# Basic comparisons
beq $s0, $s1, label    # Branch if equal
```

```
bne $s0, $s1, label    # Branch if not equal
blt $s0, $s1, label    # Branch if less than
ble $s0, $s1, label    # Branch if less than or equal
bgt $s0, $s1, label    # Branch if greater than
bge $s0, $s1, label    # Branch if greater than or equal

# Compare with zero (optimized)
beqz $s0, label        # Branch if $s0 == 0
bnez $s0, label        # Branch if $s0 != 0
blez $s0, label        # Branch if $s0 <= 0
bgez $s0, label        # Branch if $s0 >= 0
bltz $s0, label        # Branch if $s0 < 0
bgtz $s0, label        # Branch if $s0 > 0
```

Unconditional Jumps

```
j label                # Jump to label
jr $ra                 # Jump to address in register (usually return)
jal function           # Jump and link (function call)
jalr $t0               # Jump and link register
```

Loop Implementations

For Loop Pattern

```
# for (int i = 0; i < n; i++)
li $t0, 0              # i = 0
for_loop:
    bge $t0, $s0, for_done    # if i >= n, exit

    # Loop body here

    addi $t0, $t0, 1   # i++
    j for_loop
for_done:
```

While Loop Pattern

```
# while (condition)
while_loop:
    # Check condition
    beq $s0, $zero, while_done    # if condition false, exit

    # Loop body here

    j while_loop
while_done:
```

Do-While Loop Pattern

```
# do { ... } while (condition)
do_while_loop:
    # Loop body here

    # Check condition
    bne $s0, $zero, do_while_loop    # if condition true, repeat
```

If-Else Pattern

```
# if (condition) { ... } else { ... }
    bne $s0, $s1, else_part    # if condition false, go to else

    # if part
    j end_if

else_part:
    # else part

end_if:
```

---

7. Functions and Stack Management

Function Call Convention

```
# Caller responsibilities:
# 1. Save $t registers if needed
# 2. Set up arguments in $a0-$a3
# 3. Call function with jal
# 4. Handle return value in $v0-$v1

# Callee responsibilities:
# 1. Save $ra, $fp, and $s registers
# 2. Set up stack frame
# 3. Execute function
# 4. Restore saved registers
# 5. Return with jr $ra
```

Complete Function Template

```
function_name:
    # Prologue: Save context
    addi $sp, $sp, -16     # Allocate stack space
    sw $ra, 12($sp)        # Save return address
    sw $fp, 8($sp)         # Save frame pointer
    sw $s0, 4($sp)         # Save $s0 if used
    sw $s1, 0($sp)         # Save $s1 if used
    addi $fp, $sp, 16      # Set frame pointer

    # Function body
    move $s0, $a0          # Save argument
    # ... function logic ...

    # Set return value
    move $v0, $s0          # Return value in $v0

    # Epilogue: Restore context
    lw $s1, 0($sp)         # Restore $s1
    lw $s0, 4($sp)         # Restore $s0
    lw $fp, 8($sp)         # Restore frame pointer
    lw $ra, 12($sp)        # Restore return address
```

```
    addi $sp, $sp, 16      # Deallocate stack space
    jr $ra                 # Return
```

Recursive Function Example (Factorial)

```
factorial:
    addi $sp, $sp, -8      # Allocate space
    sw $ra, 4($sp)         # Save return address
    sw $a0, 0($sp)         # Save n

    # Base case
    li $v0, 1              # Default return 1
    ble $a0, 1, fact_done  # if n <= 1, return 1

    # Recursive case
    addi $a0, $a0, -1      # n-1
    jal factorial          # factorial(n-1)

    # Multiply n * factorial(n-1)
    lw $a0, 0($sp)         # Restore n
    mul $v0, $a0, $v0      # n * factorial(n-1)

fact_done:
    lw $ra, 4($sp)         # Restore return address
    addi $sp, $sp, 8       # Deallocate space
    jr $ra
```

---

8. Dynamic Memory Allocation

Basic Memory Allocation

```
# Allocate memory for n integers
sll $a0, $s0, 2        # n * 4 bytes
li $v0, 9              # syscall 9 (sbrk)
syscall
move $s1, $v0          # Save base address
```

```
# Use allocated memory
sw $t0, 0($s1)        # Store at base
sw $t1, 4($s1)        # Store at base + 4
```

Dynamic Array Implementation

```
.data
prompt: .asciiz "Enter array size: "

.text
main:
    # Get array size
    li $v0, 4
    la $a0, prompt
    syscall

    li $v0, 5
    syscall
    move $s0, $v0         # $s0 = array size

    # Allocate memory
    sll $a0, $s0, 2       # size * 4 bytes
    li $v0, 9
    syscall
    move $s1, $v0         # $s1 = array base address

    # Fill array
    li $t0, 0             # index
    move $t1, $s1         # current address
fill_loop:
    beq $t0, $s0, fill_done

    # Read value
    li $v0, 5
    syscall
    sw $v0, 0($t1)        # Store in array

    addi $t1, $t1, 4      # Next element
```

```
    addi $t0, $t0, 1       # Increment index
    j fill_loop

fill_done:
    # Array is now filled and ready to use
```

Linked List Node Creation

```
create_node:
    # Input: $a0 = data value
    # Output: $v0 = node address

    addi $sp, $sp, -4
    sw $a0, 0($sp)         # Save data

    # Allocate node (8 bytes: data + next pointer)
    li $a0, 8
    li $v0, 9
    syscall
    move $t0, $v0          # Node address

    # Initialize node
    lw $a0, 0($sp)         # Restore data
    sw $a0, 0($t0)         # node→data = data
    sw $zero, 4($t0)       # node→next = NULL

    move $v0, $t0          # Return node address
    addi $sp, $sp, 4
    jr $ra
```

9. Floating Point Operations

FPU Register System

```
# Single precision registers: $f0, $f1, $f2, ... $f31
# Double precision uses pairs: $f0-$f1, $f2-$f3, etc.
```

Basic Float Operations

```
.data
pi:     .float 3.14159
num1:   .float 10.5
num2:   .float 2.0

.text
main:
    # Load floats
    l.s $f0, pi           # Load single precision
    l.s $f1, num1
    l.s $f2, num2

    # Arithmetic operations
    add.s $f3, $f1, $f2    # $f3 = $f1 + $f2
    sub.s $f4, $f1, $f2    # $f4 = $f1 - $f2
    mul.s $f5, $f1, $f2    # $f5 = $f1 * $f2
    div.s $f6, $f1, $f2    # $f6 = $f1 / $f2

    # Math functions
    sqrt.s $f7, $f1       # Square root
    abs.s $f8, $f1        # Absolute value
    neg.s $f9, $f1        # Negation

    # Comparisons
    c.eq.s $f1, $f2       # Compare equal
    bc1t equal_label      # Branch if true
    c.lt.s $f1, $f2       # Compare less than
    bc1t less_label       # Branch if true
```

Integer-Float Conversions

```
# Integer to float
lw $t0, integer_var       # Load integer
mtc1 $t0, $f0             # Move to FPU
cvt.s.w $f0, $f0          # Convert to single precision
```

```
# Float to integer
cvt.w.s $f1, $f0        # Convert float to word
mfc1 $t0, $f1           # Move to CPU register
```

Floating Point I/O

```
# Read float
li $v0, 6
syscall                 # Result in $f0

# Print float
mov.s $f12, $f0         # Move to print register
li $v0, 2
syscall
```

---

10. Arrays and Data Structures

Static Array Access

```
.data
array: .word 1, 2, 3, 4, 5

.text
main:
    la $s0, array         # Base address
    li $t0, 2             # Index

    # Calculate address: base + (index * 4)
    sll $t1, $t0, 2       # index * 4
    add $t1, $s0, $t1     # base + offset
    lw $t2, 0($t1)        # Load array[index]

    # Alternative: direct offset
    lw $t3, 8($s0)        # Load array[2] directly
```

2D Array Access

```
.data
matrix: .word 1, 2, 3, 4, 5, 6, 7, 8, 9  # 3×3 matrix

.text
main:
    # Access matrix[row][col] where matrix is rows x cols
    # Address = base + (row * cols + col) * 4

    li $t0, 1             # row = 1
    li $t1, 2             # col = 2
    li $t2, 3             # cols = 3

    mul $t3, $t0, $t2     # row * cols
    add $t3, $t3, $t1     # row * cols + col
    sll $t3, $t3, 2       # * 4 for byte offset

    la $s0, matrix
    add $t3, $s0, $t3     # base + offset
    lw $t4, 0($t3)        # Load matrix[1][2]
```

Stack Implementation

```
.data
stack_array: .space 400   # Stack storage (100 integers)
stack_ptr: .word stack_array  # Stack pointer

.text
push:
    # Input: $a0 = value to push
    lw $t0, stack_ptr     # Get current stack pointer
    sw $a0, 0($t0)        # Store value
    addi $t0, $t0, 4      # Increment pointer
    sw $t0, stack_ptr     # Save new pointer
    jr $ra

pop:
    # Output: $v0 = popped value
```

```
    lw $t0, stack_ptr     # Get current stack pointer
    addi $t0, $t0, -4     # Decrement pointer
    lw $v0, 0($t0)        # Load value
    sw $t0, stack_ptr     # Save new pointer
    jr $ra
```

---

11. String Operations

String Length

```
strlen:
    # Input: $a0 = string address
    # Output: $v0 = length

    move $t0, $a0         # Current position
    li $v0, 0             # Length counter

strlen_loop:
    lb $t1, 0($t0)        # Load character
    beqz $t1, strlen_done # If null terminator, done
    addi $v0, $v0, 1      # Increment length
    addi $t0, $t0, 1      # Next character
    j strlen_loop

strlen_done:
    jr $ra
```

String Copy

```
strcpy:
    # Input: $a0 = destination, $a1 = source

    move $t0, $a0         # Destination pointer
    move $t1, $a1         # Source pointer

strcpy_loop:
    lb $t2, 0($t1)        # Load source character
```

```
    sb $t2, 0($t0)        # Store to destination
    beqz $t2, strcpy_done # If null terminator, done
    addi $t0, $t0, 1      # Next destination
    addi $t1, $t1, 1      # Next source
    j strcpy_loop

strcpy_done:
    jr $ra
```

String Compare

```
strcmp:
    # Input: $a0 = string1, $a1 = string2
    # Output: $v0 = 0 (equal), <0 (s1<s2), >0 (s1>s2)

    move $t0, $a0         # String1 pointer
    move $t1, $a1         # String2 pointer

strcmp_loop:
    lb $t2, 0($t0)        # Load char from string1
    lb $t3, 0($t1)        # Load char from string2

    bne $t2, $t3, strcmp_diff  # If different, compute difference
    beqz $t2, strcmp_equal     # If both null, equal

    addi $t0, $t0, 1      # Next char in string1
    addi $t1, $t1, 1      # Next char in string2
    j strcmp_loop

strcmp_diff:
    sub $v0, $t2, $t3     # Difference
    jr $ra

strcmp_equal:
    li $v0, 0             # Equal
    jr $ra
```

12. Common Programming Patterns

Bubble Sort

```
bubble_sort:
    # Input: $a0 = array address, $a1 = size

    move $s0, $a0        # Array base
    move $s1, $a1        # Array size
    li $s2, 0            # i = 0

outer_loop:
    bge $s2, $s1, sort_done

    li $s3, 0            # j = 0
    sub $t0, $s1, $s2    # size - i
    addi $t0, $t0, -1    # size - i - 1

inner_loop:
    bge $s3, $t0, inner_done

    # Calculate addresses
    sll $t1, $s3, 2      # j * 4
    add $t1, $s0, $t1    # array + j*4
    lw $t2, 0($t1)       # array[j]
    lw $t3, 4($t1)       # array[j+1]

    # Compare and swap if needed
    ble $t2, $t3, no_swap
    sw $t3, 0($t1)       # array[j] = array[j+1]
    sw $t2, 4($t1)       # array[j+1] = temp

no_swap:
    addi $s3, $s3, 1     # j++
    j inner_loop

inner_done:
    addi $s2, $s2, 1     # i++
```

```
    j outer_loop

sort_done:
    jr $ra
```

Binary Search

```
binary_search:
    # Input: $a0 = array, $a1 = size, $a2 = target
    # Output: $v0 = index (-1 if not found)

    li $t0, 0           # left = 0
    addi $t1, $a1, -1   # right = size - 1

search_loop:
    bgt $t0, $t1, not_found

    add $t2, $t0, $t1   # left + right
    srl $t2, $t2, 1     # mid = (left + right) / 2

    # Get array[mid]
    sll $t3, $t2, 2     # mid * 4
    add $t3, $a0, $t3   # array + mid*4
    lw $t4, 0($t3)      # array[mid]

    beq $t4, $a2, found  # if array[mid] == target
    blt $t4, $a2, search_right

    # Search left half
    addi $t1, $t2, -1   # right = mid - 1
    j search_loop

search_right:
    addi $t0, $t2, 1    # left = mid + 1
    j search_loop

found:
    move $v0, $t2       # Return index
```

```
    jr $ra

not_found:
    li $v0, -1          # Return -1
    jr $ra
```

Matrix Multiplication

```
matrix_multiply:
    # Multiply two 3×3 matrices
    # Input: $a0 = matrix A, $a1 = matrix B, $a2 = result matrix C

    li $t0, 0           # i = 0

mult_i_loop:
    bge $t0, 3, mult_done
    li $t1, 0           # j = 0

mult_j_loop:
    bge $t1, 3, mult_i_next
    li $t2, 0           # k = 0
    li $t3, 0           # sum = 0

mult_k_loop:
    bge $t2, 3, mult_k_done

    # Calculate A[i][k]
    mul $t4, $t0, 3      # i * 3
    add $t4, $t4, $t2    # i * 3 + k
    sll $t4, $t4, 2      # * 4
    add $t4, $a0, $t4    # A + offset
    lw $t5, 0($t4)       # A[i][k]

    # Calculate B[k][j]
    mul $t4, $t2, 3      # k * 3
    add $t4, $t4, $t1    # k * 3 + j
    sll $t4, $t4, 2      # * 4
    add $t4, $a1, $t4    # B + offset
```

```
    lw $t6, 0($t4)       # B[k][j]

    # sum += A[i][k] * B[k][j]
    mul $t4, $t5, $t6
    add $t3, $t3, $t4

    addi $t2, $t2, 1     # k++
    j mult_k_loop

mult_k_done:
    # Store C[i][j] = sum
    mul $t4, $t0, 3      # i * 3
    add $t4, $t4, $t1    # i * 3 + j
    sll $t4, $t4, 2      # * 4
    add $t4, $a2, $t4    # C + offset
    sw $t3, 0($t4)       # C[i][j] = sum

    addi $t1, $t1, 1     # j++
    j mult_j_loop

mult_i_next:
    addi $t0, $t0, 1     # i++
    j mult_i_loop

mult_done:
    jr $ra
```

13. Debugging and Best Practices

Debugging Strategies

```
# Use descriptive labels and comments
calculate_average:       # Clear function name
    # Calculate sum of array elements
    li $t0, 0            # sum = 0 (comment explains purpose)
    li $t1, 0            # index = 0
```

```
# Add debug prints during development
debug_print_register:
    li $v0, 1
    move $a0, $t0
    syscall
```