RISC-V Assembly Programming Guide

Introduction

This guide provides comprehensive instruction for programming in RISC-V assembly for the 16-bit processor simulator. It covers everything from basic syntax to advanced programming techniques, with practical examples and best practices.

Table of Contents

Assembly Language Basics

Syntax Rules

```
# Comments start with '#'

# Labels end with ':'

# Instructions are lowercase

# Registers use 'x' prefix or ABI names



main:                   # Label definition

    addi x1, x0, 10     # instruction rd, rs1, immediate

    add x2, x1, x1      # instruction rd, rs1, rs2

    halt                # Stop execution
```

Basic Structure

```
# Program structure
main:                   # Entry point
    # Initialization
    addi x1, x0, 0      # Initialize variables

    # Main logic
    # ...

    # Cleanup and exit
    halt                # Always end with halt
```

Number Formats

```
# Decimal (default)
addi x1, x0, 10         # x1 = 10

# Hexadecimal (0x prefix)
addi x1, x0, 0xA        # x1 = 10

# Binary (0b prefix)
addi x1, x0, 0b1010     # x1 = 10

# Negative numbers (2's complement)
addi x1, x0, -1         # x1 = 0xFFFF (16-bit)
```

Instruction Set Reference

R-Type Instructions (Register-Register)

ADD - Addition

add rd, rs1, rs2   # rd = rs1 + rs2

add x3, x1, x2   # x3 = x1 + x2

SUB - Subtraction

sub rd, rs1, rs2   # rd = rs1 - rs2

sub x3, x1, x2   # x3 = x1 - x2

AND - Bitwise AND

and rd, rs1, rs2   # rd = rs1 & rs2

and x3, x1, x2   # x3 = x1 & x2

OR - Bitwise OR

or rd, rs1, rs2   # rd = rs1 | rs2

or x3, x1, x2   # x3 = x1 | x2

XOR - Bitwise XOR

xor rd, rs1, rs2   # rd = rs1 ^ rs2

xor x3, x1, x2   # x3 = x1 ^ x2

I-Type Instructions (Immediate)

ADDI - Add Immediate

addi rd, rs1, imm   # rd = rs1 + imm

addi x1, x0, 15   # x1 = 0 + 15 = 15

ANDI - AND Immediate

andi rd, rs1, imm   # rd = rs1 & imm

andi x2, x1, 0xF   # x2 = x1 & 15

ORI - OR Immediate

ori rd, rs1, imm   # rd = rs1 | imm

ori x2, x1, 0x8   # x2 = x1 | 8

LW - Load Word

lw rd, offset(rs1)   # rd = memory[rs1 + offset]

lw x2, 0(x1)   # x2 = memory[x1 + 0]

lw x3, 4(x1)   # x3 = memory[x1 + 4]

S-Type Instructions (Store)

SW - Store Word

sw rs2, offset(rs1)   # memory[rs1 + offset] = rs2

sw x2, 0(x1)   # memory[x1 + 0] = x2

sw x3, 4(x1)   # memory[x1 + 4] = x3

B-Type Instructions (Branch)

BEQ - Branch if Equal

beq rs1, rs2, label   # if (rs1 == rs2) goto label

beq x1, x2, end   # if x1 equals x2, jump to 'end'

BNE - Branch if Not Equal

bne rs1, rs2, label   # if (rs1 != rs2) goto label bne x1, x0, loop # if x1 not zero, jump to 'loop'

J-Type Instructions (Jump)

JAL - Jump and Link

jal rd, label # rd = PC + 1; PC = label

jal x1, function # Call function, store return in x1

Special Instructions

NOP - No Operation

nop # Do nothing for one cycle

HALT - Stop Execution

halt # Stop processor execution

Register Usage Conventions

Register Map with ABI Names

Register	ABI Name	Purpose	Caller/Callee Saved
x0	zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee

x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5-x7	t0-t2	Temporaries	Caller
x8-x9	s0-s1	Saved registers	Callee
x10-x14	a0-a4	Function arguments	Caller
x15	a7	System call number	Caller

Register Usage Examples

```
# Using numeric names
addi x1, x0, 10       # Use x1 as return address
addi x2, x0, 100      # Use x2 as stack pointer

# Using ABI names (preferred)
addi ra, zero, 10     # Same as above
addi sp, zero, 100    # More readable

# Function arguments
addi a0, zero, 5      # First argument
addi a1, zero, 10     # Second argument
jal ra, multiply      # Call function
```

```
multiply:
    add a0, a0, a1     # Result in a0
    # Return (ra contains return address)
```

Programming Patterns

1. Variable Assignment

```
# Simple assignment: var = 42
addi x1, x0, 42       # x1 = 42

# Copy variable: var2 = var1
add x2, x1, x0        # x2 = x1 + 0 = x1
```

2. Arithmetic Operations

```
# Addition: result = a + b + c
add x4, x1, x2        # temp = a + b
add x4, x4, x3        # result = temp + c

# Multiplication by powers of 2
add x2, x1, x1        # x2 = x1 * 2
add x3, x2, x2        # x3 = x1 * 4
add x4, x3, x3        # x4 = x1 * 8

# Multiplication by constant (e.g., x * 5 = x * 4 + x)
add x2, x1, x1        # x2 = x * 2
add x2, x2, x2        # x2 = x * 4
```

add x2, x2, x1 # x2 = x * 5

3. Bit Manipulation

Set bit (OR with power of 2)

ori x2, x1, 0b0001 # Set bit 0

ori x2, x1, 0b0010 # Set bit 1

ori x2, x1, 0b1000 # Set bit 3

Clear bit (AND with inverted mask)

andi x2, x1, 0b1110 # Clear bit 0

andi x2, x1, 0b1101 # Clear bit 1

Toggle bit (XOR with power of 2)

xori x2, x1, 0b0001 # Toggle bit 0

Test bit

andi x2, x1, 0b0001 # x2 = 0 if bit 0 clear, 1 if set

Memory Operations

1. Simple Load/Store

Store value to memory

addi x1, x0, 42 # Value to store

sw x1, 0(x0) # Store at memory address 0x1000

Load value from memory

lw x2, 0(x0)   # Load from memory address 0x1000

2. Array Operations

Array initialization: arr[0] = 10, arr[1] = 20, arr[2] = 30

addi x1, x0, 10   # arr[0] = 10

addi x2, x0, 20   # arr[1] = 20

addi x3, x0, 30   # arr[2] = 30

sw x1, 0(x0)   # Store arr[0] at base address

sw x2, 1(x0)   # Store arr[1] at base + 1

sw x3, 2(x0)   # Store arr[2] at base + 2

Array access: sum = arr[0] + arr[1] + arr[2]

lw x4, 0(x0)   # Load arr[0]

lw x5, 1(x0)   # Load arr[1]

lw x6, 2(x0)   # Load arr[2]

add x7, x4, x5   # sum = arr[0] + arr[1]

add x7, x7, x6   # sum = sum + arr[2]

3. Dynamic Array Access

Access arr[index] where index is in x1

Base address is 0x1000 (memory base)

add x2, x0, x1   # x2 = base + index

lw x3, 0(x2)   # x3 = arr[index]

Control Flow

1. Conditional Execution

```
# if (x1 == x2) then x3 = 1 else x3 = 0

beq x1, x2, then_case

addi x3, x0, 0       # else case: x3 = 0

beq x0, x0, end_if   # jump to end

then_case:

    addi x3, x0, 1   # then case: x3 = 1

end_if:

    # continue...
```

2. Loops

Simple Count Loop

```
# for (i = 0; i < 5; i++)

addi x1, x0, 0       # i = 0

addi x2, x0, 5       # limit = 5


loop:

    beq x1, x2, end_loop  # if i == limit, exit


    # Loop body here

    # ... do something with x1


    addi x1, x1, 1   # i++

    beq x0, x0, loop  # continue loop
```

```
end_loop:
    # continue after loop
```

While Loop

```
# while (x1 != 0)
while_loop:
    beq x1, x0, end_while  # if x1 == 0, exit

    # Loop body
    # ... modify x1

    beq x0, x0, while_loop # continue loop

end_while:
    # continue after loop
```

Do-While Loop

```
# do { ... } while (x1 != 0)
do_loop:
    # Loop body
    # ... modify x1

    bne x1, x0, do_loop   # if x1 != 0, continue

    # continue after loop
```

3. Function Calls

```
# Function definition
main:
    addi a0, x0, 5       # First argument
    addi a1, x0, 3       # Second argument
    jal ra, add_function  # Call function
    # Result is now in a0
    halt

add_function:
    add a0, a0, a1       # result = arg1 + arg2
    # Return (ra contains return address)
    # Note: In real RISC-V, we'd use 'jr ra' but we simulate return
```

Advanced Techniques

1. Fibonacci Sequence

```
# Generate Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13...
main:
    addi x1, x0, 1       # fib(n-2) = 1
    addi x2, x0, 1       # fib(n-1) = 1
    addi x3, x0, 8       # counter = 8 (generate 8 numbers)

    sw x1, 0(x0)        # Store first number
    sw x2, 1(x0)        # Store second number
    addi x4, x0, 2      # memory index = 2
```

```
fib_loop:
    beq x3, x0, fib_done # if counter == 0, done

    add x5, x1, x2       # next_fib = fib(n-2) + fib(n-1)
    sw x5, 0(x4)         # Store in memory[index]

    add x1, x2, x0       # fib(n-2) = fib(n-1)
    add x2, x5, x0       # fib(n-1) = next_fib
    addi x4, x4, 1       # index++
    addi x3, x3, -1      # counter--

    beq x0, x0, fib_loop # continue

fib_done:
    halt
```

2. Bubble Sort

```
# Bubble sort algorithm for array of 5 elements
main:
    # Initialize array: [5, 2, 8, 1, 9]
    addi x1, x0, 5
    addi x2, x0, 2
    addi x3, x0, 8
    addi x4, x0, 1
    addi x5, x0,
```