Documentation for RISC-V Assembly Code Examples

Prepared by: Ayesha

Introduction:

This document provides examples of various arithmetic and logical operations implemented in RISC-V Assembly language. The examples cover fundamental operations like NOT, AND, OR, XOR, addition, subtraction, multiplication, and division. Each example includes the necessary assembly code along with comments explaining the code's functionality.

Table of Contents

1. Logical operations

1.1. NOT Operation (¬A)

The NOT operation takes a single operand and inverts its bits.

```
.data

.text
.global _start

_start:
li a0, 5       # Load immediate value 5 into register a0
not a1, a0     # Perform bitwise NOT on a0, store result in a1
```

1.2. AND Operation (A & B)

The AND operation takes two operands and performs a bitwise AND.

```
.data

.text
.global _start

_start:
li a0, 9       # Load immediate value 9 into register a0
li a1, 5       # Load immediate value 5 into register a1
and a2, a1, a0 # Perform bitwise AND on a1 and a0, store result in a2
```

1.3. OR Operation (A | B)

The OR operation takes two operands and performs a bitwise OR.

```
.data

.text
.global _start
```

```
_start:
li a0, 9       # Load immediate value 9 into register a0
li a1, 5       # Load immediate value 5 into register a1
or a2, a1, a0  # Perform bitwise OR on a1 and a0, store result in a2
```

1.4. XOR Operation (A ⊕ B)

The XOR operation takes two operands and performs a bitwise XOR.

```
.data

.text
.global _start

_start:
li a0, 9       # Load immediate value 9 into register a0
li a1, 5       # Load immediate value 5 into register a1
xor a2, a1, a0 # Perform bitwise XOR on a1 and a0, store result in a2
```

2. Arithmetic Operations

2.1. ADD Operation (A + B)

The ADD operation takes two operands and performs addition.

```
.data
w: .word 12    # Define word with value 12
d: .word 13    # Define word with value 13

.text
lw a0, w       # Load value of w into register a0
lw a1, d       # Load value of d into register a1
add a2, a1, a0 # Add values in a1 and a0, store result in a2
```

```
# Another example with a negative value

.data
w: .word 12
d: .word -13

.text
lw a0, w
lw a1, d
add a2, a1, a0
```

2.2. SUB Operation (A - B)

The SUB operation takes two operands and performs subtraction.

```
.data
w: .word 8     # Define word with value 8
d: .word 3     # Define word with value 3

.text
lw x10, w      # Load value of w into register x10
```

```
lw x11, d       # Load value of d into register x11
sub x11, x11, x10 # Subtract value in x10 from value in x11, store result in x11
```

3. Branch, Loops And Conditions

A for loop can be simulated using branch instructions.

```
.data
sum: .word 0

.text
.globl _start

_start:
    li t0, 0        # Initialize sum to 0
    li t1, 1        # Initialize counter to 1
    li t2, 10       # Initialize limit to 10

loop:
    add t0, t0, t1  # Add counter to sum
    addi t1, t1, 1  # Increment counter
    ble t1, t2, loop # If counter <= limit, branch to loop

    la t3, sum      # Load address of sum variable into t3
    sw t0, 0(t3)    # Store the sum in memory at sum

end:
    li a7, 93       # Load exit system call number into a7
    ecall           # Make system call to exit
```

4. Special Functions APIs

4.1. Multiplication Using Shift (A << B)

Multiplication can be performed using bitwise left shift operations.

```
.data
w: .word 9     # Define word with value 9
d: .word 1     # Define word with value 1
x: .word 113   # Define word with value 113

.text
lw t1, d       # Load value of d into register t1
lw t0, w       # Load value of w into register t0
lw a0, x       # Load value of x into register a0
sll t2, t1, t0 # Shift left t1 by the number of bits specified in t0, store result in t2
add a1, a0, t2 # Add values in a0 and t2, store result in a1
```

Note: sll also works for multiplication by shifting left. It considers last source register as the power of 2.

4.2. Division using logical shift right (A >> B) srl

Division can be performed using bitwise right shift operations.

```
.data

.text
.global _start
_start:
    li a0, 64       # Load immediate value 64 into register a0
    li a1, 2        # Load immediate value 2 into register a1
    srl a2, a0, a1  # Shift right on a0 by the number of positions in a1, store result in a2
```

Note: srl also works for division by shifting towards right. It considers last source register as the power of 2.

4.3. Square Root by using logical shift left (A * B^2) sll

Calculating the square of a number using shift operations.

```
.data
w: .word 3     # Define word with value 3
d: .word 4     # Define word with value 4

.text
lw t0, w       # Load value of w into register t0
lw t1, d       # Load value of d into register t1
sll t2, t1, t0 # Shift left t1 by the number of bits specified in t0, store result in t2
```

Note: sll also works for taking square and multiply by other source register by shifting towards left. It consider last source register as the power of 2 as (2^3) here shift will be 3 as t0=3.

4.4. Multiplication Using Addition (A * B)

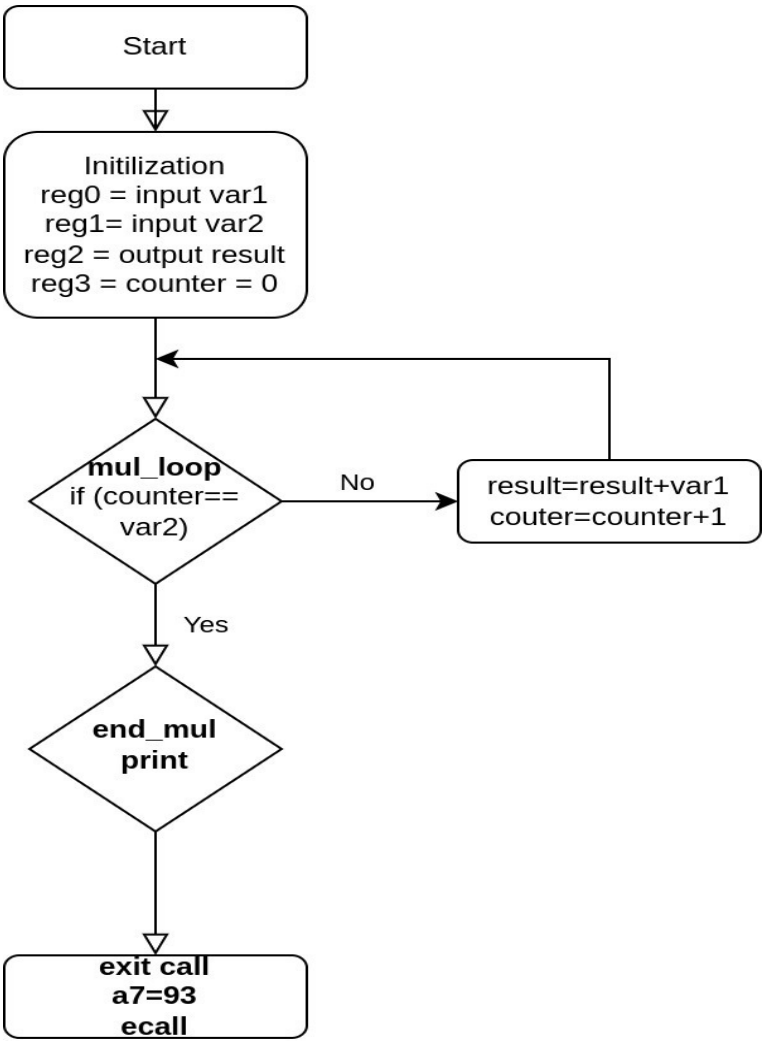
Multiplication can be performed using repeated addition.

```
.data

.text
.global _start
_start:
    li a0, 4         # Load immediate value 4 into register a0
    li a1, 9         # Load immediate value 9 into register a1
    li a2, 0         # Initialize result register a2 to 0
    li a3, 0         # Initialize counter register a3 to 0
multiply_loop:
    beq a3, a1, end_multiply # If counter equals multiplier a1, end loop
    add a2, a2, a0           # Add multiplicand(a0) to result
    addi a3, a3, 1           # Increment counter a3
    jal zero, multiply_loop  # Jump to the beginning of the loop
end_multiply:
    mv a4, a2                # Move result (a2) to a0 (return value)
    # Exit the program
    li a7, 93                # ecall code for exit in many RISC-V systems
    ecall                    # Make the system call to exit
```

Note:

RISC-V Proxy Kernel (pk) and Spike Simulator: They often use 93 as the exit code. In some RISC-V environments, the exit code might be different, such as 0 or 60 (common in Linux environments for the exit system call). The ecall instruction requires the system call number to be in register a7. You can use another register to temporarily hold the system call number, but before calling ecall, you must move that value into a7.

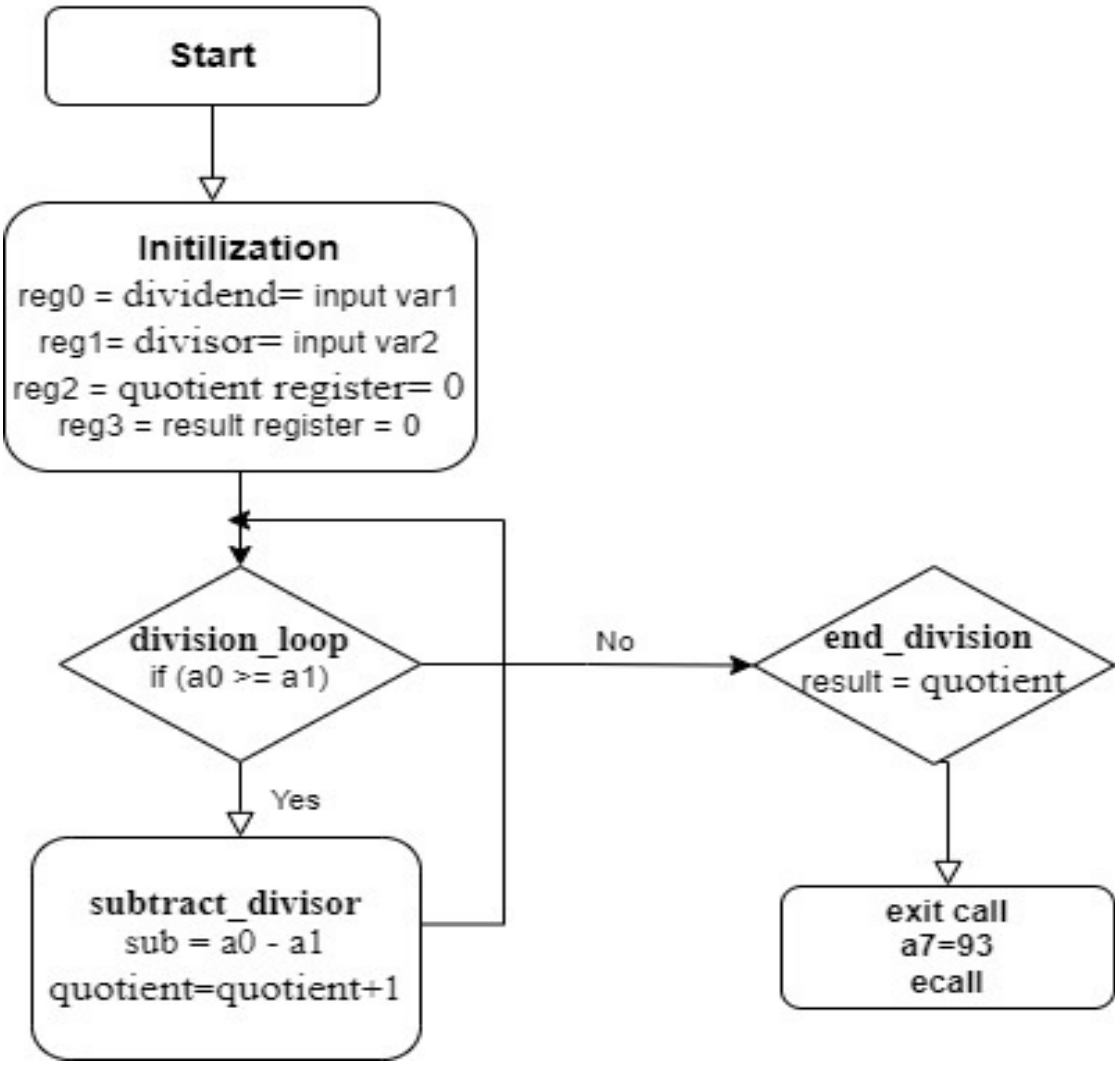
Flow Diagram for Multiplication by using Addition:



4.5. Division Using Subtraction (A / B)

Division can be performed using repeated subtraction.

```
.data
.text
.global _start

_start:
li a0, 18      # Load dividend (numerator) into register a0
li a1, 6       # Load divisor (denominator) into register a1
li a2, 0       # Initialize quotient register a2 to 0
li a3, 0       # Initialize result register a3 to 0

division_loop:
bge a0, a1, subtract_divisor # If dividend (a0) >= divisor (a1), continue loop
jal end_division  # Otherwise, jump to end_division

subtract_divisor:
sub a0, a0, a1 # Subtract divisor from dividend
addi a2, a2, 1 # Increment quotient
jal division_loop # Jump to the beginning of the loop

end_division:
mv a3, a2      # Move result into a3

# Exit the program
li a7, 93      # ecall code for exit in many RISC-V systems
ecall          # Make the system call to exit
```


Start
Initilization
reg0 = dividend= input var1
reg1= divisor= input var2
reg2 = quotient register= 0
reg3 = result register = 0
division_loop
if (a0 >= a1)
No
end_division
result = quotient
Yes
subtract_divisor
sub = a0 - a1
quotient=quotient+1
exit call
a7=93
ecall

5. Memory and IO Access

5.1. Accessing Memory through registers

The use of pointers (addresses) to load values from memory, perform an addition operation, and store the result back into memory.

```
.data

.text
.globl _start
_start:
    # Load immediate values 10 and 3 into registers t2 and t3
    li t2, 10        # Load immediate value 10 into t2
    li t3, 3         # Load immediate value 3 into t3

    # Load specific memory addresses into registers t0 and t1
    li t0, 0x44      # Load immediate address 0x44 into t0
    li t1, 0x55      # Load immediate address 0x55 into t1

    # Store the values from t2 and t3 into memory at addresses t0 and t1
    sw t2, 0(t0)     # Store word from t2 into memory address 0x44 (t0)
    sw t3, 0(t1)     # Store word from t3 into memory address 0x55 (t1)

    # Perform addition of the values in t2 and t3
    add t4, t2, t3   # t4 = t2 + t3

    # Store the result of the addition into memory address 0x32
    li t5, 0x32      # Load immediate address 0x32 into t5
    sw t4, 0(t5)     # Store word from t4 into memory address 0x32 (t5)

    # End of program with exit system call
    li a7, 93        # Load exit system call number into a7
    ecall            # Make system call to exit
```

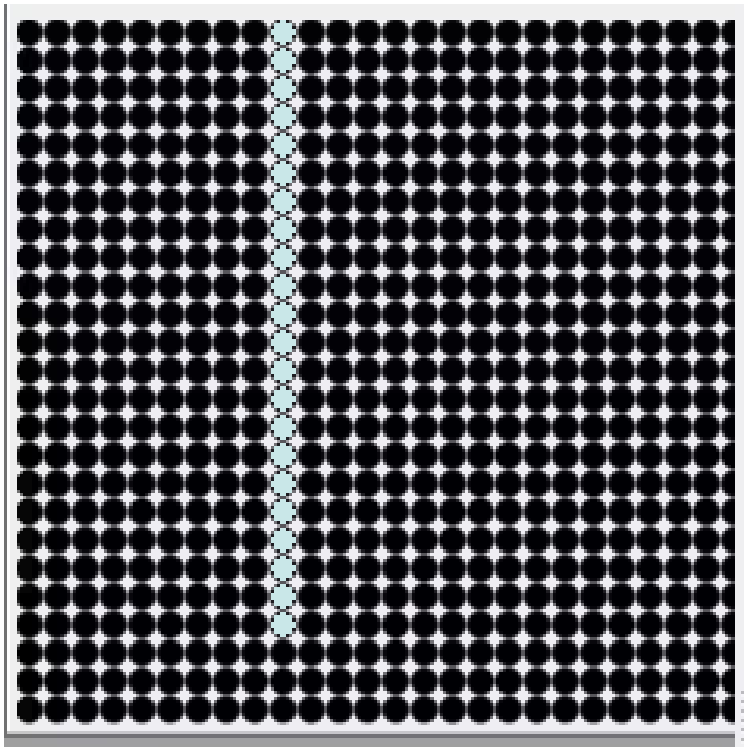
5.2. On first LED on matrix

```
# Initialize the base address of the LED matrix
li a0, LED_MATRIX_0_BASE
```

```
# Initialize the width and height of the LED matrix
li a1, LED_MATRIX_0_WIDTH
li a2, LED_MATRIX_0_HEIGHT

# Set the value of the first LED (turn it on)
li t0, 0xFFFFFF # White color (assuming 24-bit color: 0xRRGGBB)
sw t0, 0(a0) # Store the color value at the base address

li a7, 93
ecall
```

6. Task:

Write a Assembly code that display 1 on the screen



7. Conclusion:

This document provides a collection of RISC-V assembly code examples demonstrating various arithmetic and logical operations. These examples can serve as a reference for learning and understanding basic RISC-V assembly programming concepts.