Logical Implementation of ALU using MIPS Assembly Programming

Vedant Sawal
Department Of Computer Science
San Jose State University
San Jose, California
vedant.sawal@sjsu.edu

***Abstract*—This report demonstrates how to implement an ALU, which are found on every computer. The main objective is to carry out addition, subtraction, multiplication, and division first using arithmetic operations and then using only logical operations. The MARS Simulator is used to write the assembly code. The main idea is to only use a combination of simple logic gates to perform complex operations.**

***Index Terms*—MIPS, MARS, Assembly Programming, ALU, Logic Gates, Arithmetic Operations**

I. Introduction

This project's goal is to create a simple calculator using logical operators in order to demonstrate how digital logic can be utilized to perform basic arithmetic. The calculator is capable of performing addition, subtraction, multiplication, and division. The implementation is done in MIPS assembly code on an Integrated Development Environment (IDE) called MARS (MIPS Assembly and Runtime Simulator).

II. MARS Installation and Setup

MARS is a lightweight IDE and Simulator for MIPS Assembly Language Programming intended for educational-level use. MARS can be downloaded for free from [1]. MARS is developed using Java so to use MARS on your machine, please ensure that you have Java Runtime Environment JRE-8 or higher installed on your machine. For more information on how to install JRE, please refer to [2].

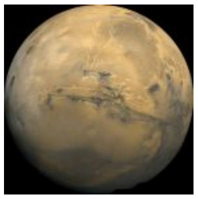


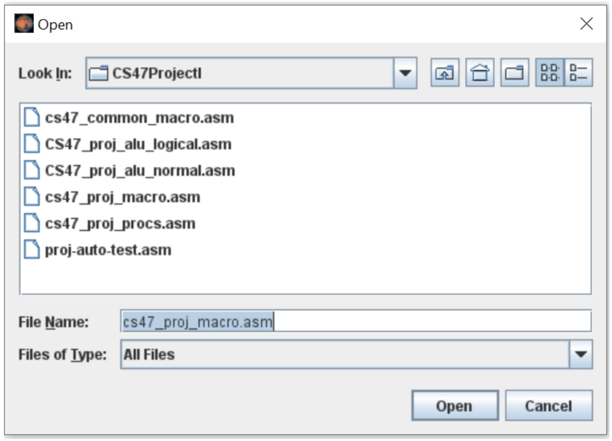
Fig. 1. MARS IDE Icon

A. *Installation*

MARS is a Java application available in a Java Archive (JAR) format which is equivalent to an executable file. So there is no installation required to use MARS. All we need to do is to download the package `MARS4_5.jar` from [1] and execute it. Once downloaded, double-click on the MARS icon and the MARS IDE will open up.

B. *Project Setup*

Download the CS47 project starter code zip file from Canvas and extract the files. Figure 2 shows the file menu in MARS IDE with all the extracted files from the zip.



Open

Look In: CS47ProjectI

cs47_common_macro.asm
CS47_proj_alu_logical.asm
CS47_proj_alu_normal.asm
cs47_proj_macro.asm
cs47_proj_procs.asm
proj-auto-test.asm

File Name: cs47_proj_macro.asm
Files of Type: All Files

Open Cancel

Fig. 2. CS47 Project Files in MARS IDE

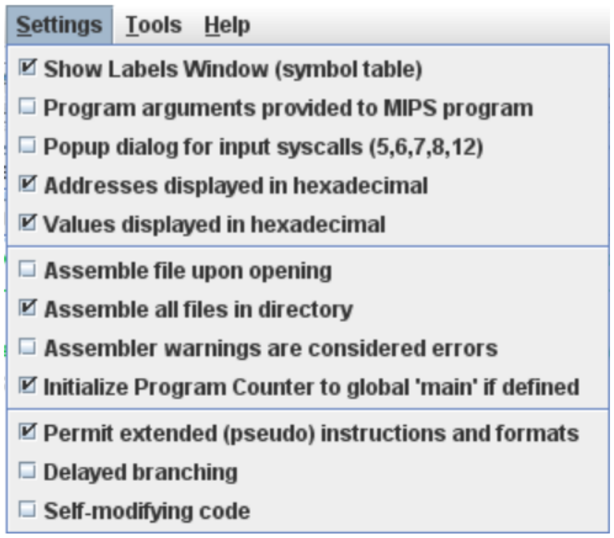
To complete the project, following relevant files are required to be modified:

- `cs47_proj_alu_logical.asm`: ALU implementation using logical commands
- `cs47_proj_alu_normal.asm`: ALU implementation using arithmetic commands
- `cs47_proj_common_macros.asm`: Utility macros defined by the user

For this project to run correctly, some default settings are required to be changed. As shown in the figure, select the following settings:

- *Assemble all files in the directory*: This setting assembles all the files in the directory. No need to individually assemble each file.
- *Initialize Program Counter to global main if defined*: This setting enables the system to start the program at the address defined by `main` instead of the first line of the first encountered file.

- *Permit extended (pseudo) instructions and formats*: This setting enables the use of MIPS pseudo/non-native instructions in MARS IDE. The MIPS assembler transforms the pseudo-instructions into native instructions prior to execution.



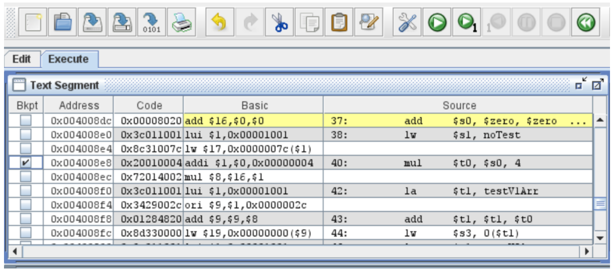
☑ Show Labels Window (symbol table)
☐ Program arguments provided to MIPS program
☐ Popup dialog for input syscalls (5,6,7,8,12)
☑ Addresses displayed in hexadecimal
☑ Values displayed in hexadecimal
☐ Assemble file upon opening
☑ Assemble all files in directory
☐ Assembler warnings are considered errors
☑ Initialize Program Counter to global 'main' if defined
☑ Permit extended (pseudo) instructions and formats
☐ Delayed branching
☐ Self-modifying code

Fig. 3. MARS Settings

C. *Debugging*

Debugging assembly code is quite challenging. To assist the developers in debugging, MARS provides two useful features in the IDE:

- Ability to set breakpoints: You can pause the execution of a mips assembly program by setting a **bkpt** in the *Execute* tab in the IDE editor.
- Forward single-stepping and Backward single-stepping: MARS provides the ability to execute instructions one step at a time in forward and backward direction.



Bkpt	Address	Code	Basic	Source
☐	0x004008dc	0x00008020	add $16,$0,$0	37: add $s0, $zero, $zero ...
☐	0x004008e0	0x3c011001	lui $1,0x00001001	38: lw $s1, noTest
☐	0x004008e4	0x8c31007c	lw $17,0x0000007c($1)	
✓	0x004008e8	0x20010004	addi $1,$0,0x00000004	40: mul $t0, $s0, 4
☐	0x004008ec	0x72014002	mul $8,$16,$1	
☐	0x004008f0	0x3c011001	lui $1,0x00001001	42: la $t1, testV1Arr
☐	0x004008f4	0x3429002c	ori $9,$1,0x0000002c	
☐	0x004008f8	0x01284820	add $9,$9,$8	43: add $t1, $t1, $t0
☐	0x004008fc	0x8d330000	lw $19,0x00000000($9)	44: lw $s3, 0($t1)

Fig. 4. Breakpoint and Single Stepping

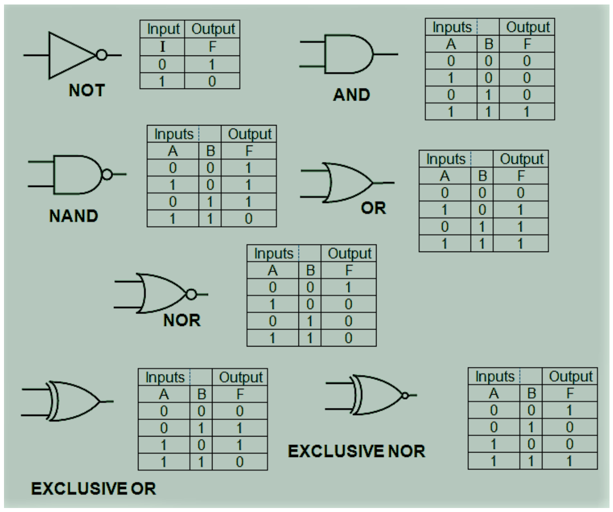
Figure 4 shows the breakpoint and single stepping features in MARS.

III. Requirements

In order to achieve successful completion of this project, it is imperative that we first satisfy several prerequisites. These prerequisites comprise comprehension and mastery of certain mathematical concepts, as well as familiarity with the precise project specifications, which are outlined in the subsequent subsections.

A. *Knowledge Base*

- **Binary Numeral System**: The binary numeral system is a base-2 number system that uses only two digits, 0 and 1, to represent numbers. Each digit in a binary number represents a power of 2, with the rightmost digit representing 2^0 (1), the next representing 2^1 (2), the next 2^2 (4), and so on. The use of binary numbers allows for more efficient and reliable electronic circuits, as binary signals are less susceptible to interference and noise than analog signals. Binary numbers also make it easier to perform logical operations and to represent complex data structures in computer programming.
- **Boolean Logic**: Boolean logic is a form of algebra that deals with variables that can have only two possible values: true or false (often represented by 1 or 0). It is based on the work of George Boole and is widely used in digital electronics and computer programming.



Input	Output
I	F
0	1
1	0

Inputs		Output
A	B	F
0	0	0
1	0	0
0	1	0
1	1	1

Inputs		Output
A	B	F
0	0	1
1	0	1
0	1	1
1	1	0

Inputs		Output
A	B	F
0	0	0
1	0	1
0	1	1
1	1	1

Inputs		Output
A	B	F
0	0	1
1	0	0
0	1	0
1	1	0

Inputs		Output
A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Inputs		Output
A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

Fig. 5. Logic Gates

Boolean logic includes several basic operations, including:

- **AND**: A logical operation that returns true only if both inputs are true.
- **OR**: A logical operation that returns true if either input is true.
- **NOT**: A logical operation that returns the opposite of the input value (e.g., not true is false).

These operations can be combined to create more complex expressions, such as:

- **NAND**: The NOT-AND operation, which returns true unless both inputs are true.
- **NOR**: The NOT-OR operation, which returns true only if both inputs are false.
- **XOR**: The exclusive OR operation, which returns true if exactly one input is true.

Figure 5 shows symbols and truth tables for the above-mentioned logic gates.

- **Computer Architecture**: To write efficient MIPS assembly code, you'll need a good understanding of computer architecture, including concepts such as memory management, registers, and instruction sets.

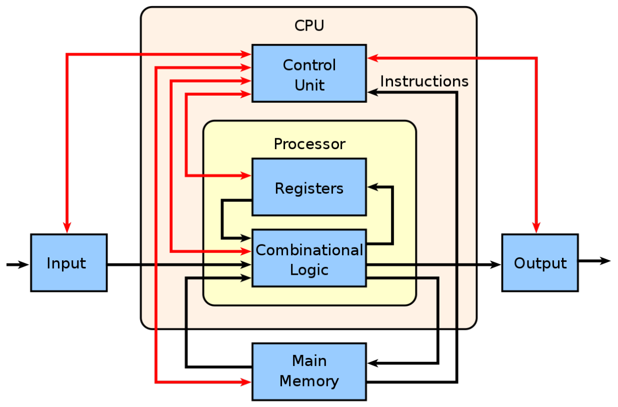


Fig. 6. Simplified Computer Architecture

B. Project Requirements

There are two distinct methods to implement all the necessary functions and logic. The four mathematical operations `ADDITION`, `SUBTRACTION`, `MULTIPLICATION`, `DIVISION` can be implemented through actual arithmetic operations or via Boolean logic. Each method requires different functions to complete the algorithm and carry out the operations.

1) Arithmetic Procedure: Arithmetic Procedure is required to be implemented using arithmetic operators. The procedure `au_normal` is the entry point for implementing the functionality using arithmetic operators. This procedure is located in file `cs47_proj_alu_normal.asm` of the project zip.

2) Logical Procedure: Logical Procedure requires us to implement the same functionality using only logical operators. No arithmetic operators are permitted. The procedure `au_logical` is what we have to develop. This procedure is located in file `cs47_proj_alu_logical.asm` of the project zip.

IV. Design and Implementation

A. Design Methodology

The following design principles have been used to design and structure the code in this project:

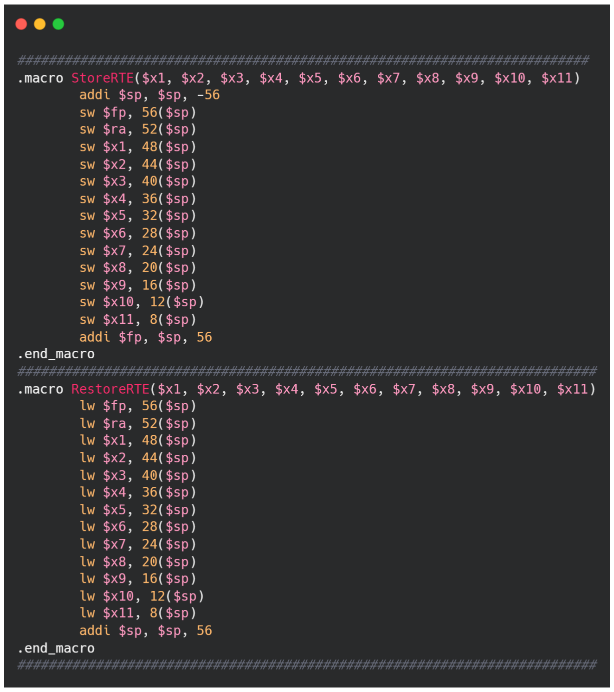
- Code Modularity: Create small self-contained code modules.
- Code Reusability: If a piece of code is required at more than one place, create a procedure or a macro for it. Don't duplicate the same piece of code at multiple places.
- Ease of Debugging: Unit test your code modules as you're developing them. Don't serialize the development and testing steps.

B. Code Implementation

This section describes the details of MIPS assembly code implemented to fulfill the requirements.

*1) **Utility Macros**:* For code readability and modularity, common functionality is implemented in the utility macro file `cs47_proj_macro.asm`. The following macros are implemented:

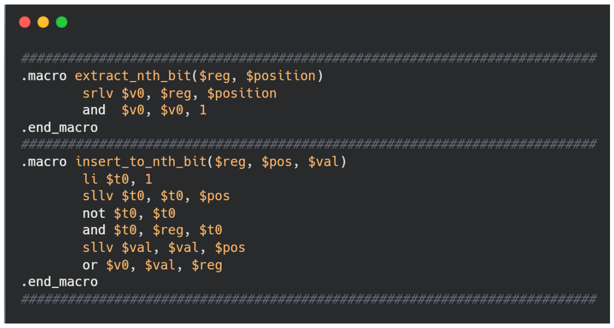
- Runtime environment: To support procedure calls, storing and restoring the runtime environment such as SP, FP, RA, and registers is a vital requirement. Without this, you may run into runtime failures, exceptions or undesired output. Since this functionality is required in every procedure, creating a macro for it makes the most sense.



```
##########################################################################
.macro StoreRTE($x1, $x2, $x3, $x4, $x5, $x6, $x7, $x8, $x9, $x10, $x11)
        addi $sp, $sp, -56
        sw $fp, 56($sp)
        sw $ra, 52($sp)
        sw $x1, 48($sp)
        sw $x2, 44($sp)
        sw $x3, 40($sp)
        sw $x4, 36($sp)
        sw $x5, 32($sp)
        sw $x6, 28($sp)
        sw $x7, 24($sp)
        sw $x8, 20($sp)
        sw $x9, 16($sp)
        sw $x10, 12($sp)
        sw $x11, 8($sp)
        addi $fp, $sp, 56
.end_macro
##########################################################################
.macro RestoreRTE($x1, $x2, $x3, $x4, $x5, $x6, $x7, $x8, $x9, $x10, $x11)
        lw $fp, 56($sp)
        lw $ra, 52($sp)
        lw $x1, 48($sp)
        lw $x2, 44($sp)
        lw $x3, 40($sp)
        lw $x4, 36($sp)
        lw $x5, 32($sp)
        lw $x6, 28($sp)
        lw $x7, 24($sp)
        lw $x8, 20($sp)
        lw $x9, 16($sp)
        lw $x10, 12($sp)
        lw $x11, 8($sp)
        addi $sp, $sp, 56
.end_macro
##########################################################################
```

Fig. 7. Runtime environment macro

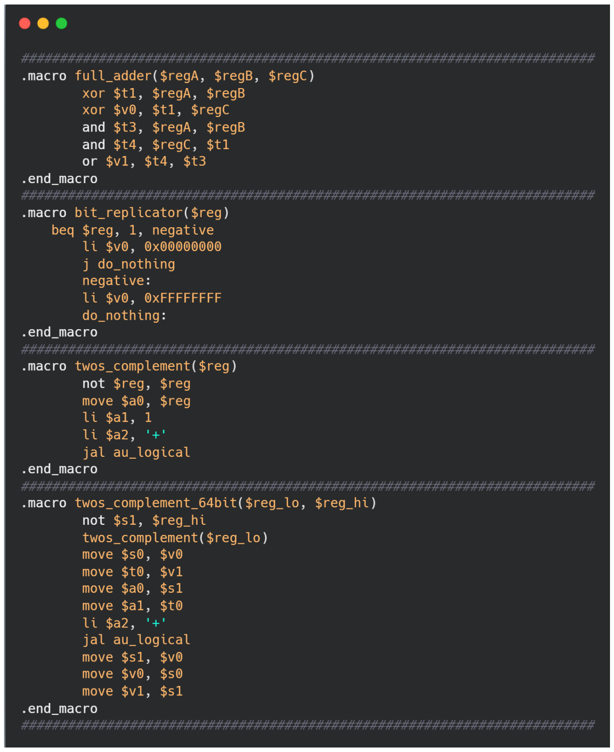
- Bit manipulation on registers: Since we are working with bits, bit manipulation operations on registers such as extraction, insertion, and modification a prime candidate for a macro.
  - `extract_nth_bit`: Uses a masking technique to extract bit value of the bit position
  - `Insert_to_nth_bit`: Macro to place a specific value into a specific bit position of a register.
  - `bit_replicator`: Used to replicate a specific bit 32 times to make 32 bits.



```
###########################################################################
.macro extract_nth_bit($reg, $position)
        srlv $v0, $reg, $position
        and  $v0, $v0, 1
.end_macro
###########################################################################
.macro insert_to_nth_bit($reg, $pos, $val)
        li $t0, 1
        sllv $t0, $t0, $pos
        not $t0, $t0
        and $t0, $reg, $t0
        sllv $val, $val, $pos
        or $v0, $val, $reg
.end_macro
###########################################################################
```

Fig. 8. Bit manipulation macro

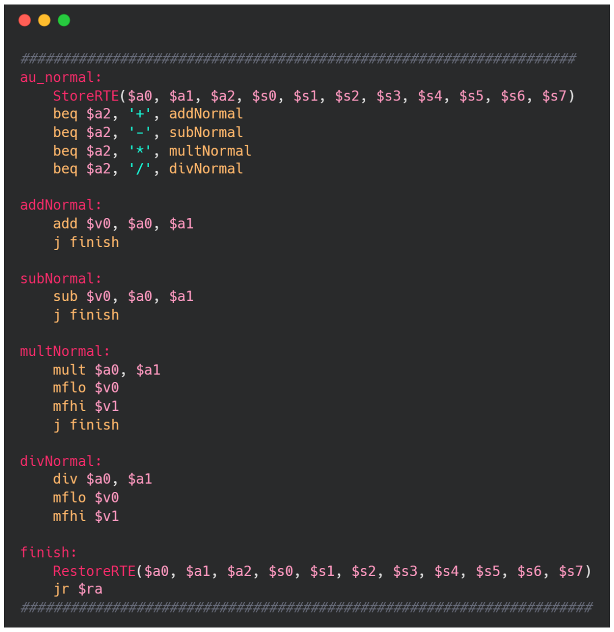
- Other macros: This macro contains common functionality like 2's complement for 16 bit and 32 bit inputs, one-bit adder and bit replicator.
  - `full_adder`: Macro to add a full bit with a carry.
  - `twos_complement`: Converts a 32-bit value into its corresponding twos complement representation.
  - `twos_complement_64bit`: Converts a 64-bit value into its corresponding twos complement representation.



```
###########################################################################
.macro full_adder($regA, $regB, $regC)
        xor $t1, $regA, $regB
        xor $v0, $t1, $regC
        and $t3, $regA, $regB
        and $t4, $regC, $t1
        or $v1, $t4, $t3
.end_macro
###########################################################################
.macro bit_replicator($reg)
    beq $reg, 1, negative
        li $v0, 0x00000000
        j do_nothing
        negative:
        li $v0, 0xFFFFFFFF
        do_nothing:
.end_macro
###########################################################################
.macro twos_complement($reg)
        not $reg, $reg
        move $a0, $reg
        li $a1, 1
        li $a2, '+'
        jal au_logical
.end_macro
###########################################################################
.macro twos_complement_64bit($reg_lo, $reg_hi)
        not $s1, $reg_hi
        twos_complement($reg_lo)
        move $s0, $v0
        move $t0, $v1
        move $a0, $s1
        move $a1, $t0
        li $a2, '+'
        jal au_logical
        move $s1, $v0
        move $v0, $s0
        move $v1, $s1
.end_macro
###########################################################################
```

Fig. 9. Twos complement macro

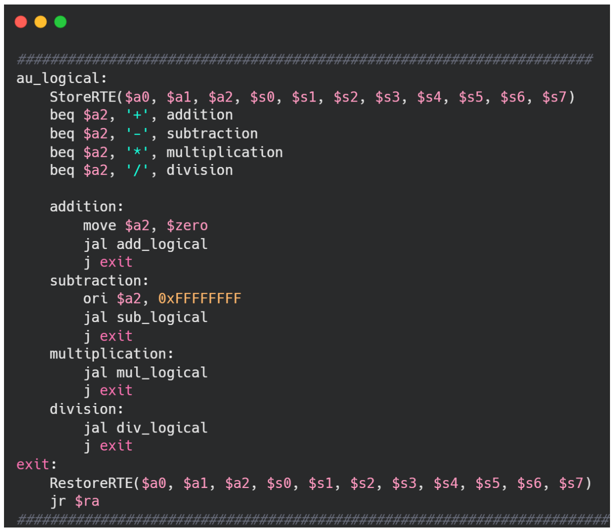
*2) **Arithmetic Procedure**:* The `au_normal` procedure is the entry point for the implementation of addition, subtraction, multiplication, and division using the corresponding arithmetic operators. The procedure works with three arguments, two operands, and an operator. Based on the type of operation, the procedure branches to the appropriate code label and executes the instructions. The output is placed in `$v0` and `$v1`.



```
###########################################################################
au_normal:
    StoreRTE($a0, $a1, $a2, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7)
    beq $a2, '+', addNormal
    beq $a2, '-', subNormal
    beq $a2, '*', multNormal
    beq $a2, '/', divNormal

addNormal:
    add $v0, $a0, $a1
    j finish

subNormal:
    sub $v0, $a0, $a1
    j finish

multNormal:
    mult $a0, $a1
    mflo $v0
    mfhi $v1
    j finish

divNormal:
    div $a0, $a1
    mflo $v0
    mfhi $v1

finish:
    RestoreRTE($a0, $a1, $a2, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7)
    jr $ra
###########################################################################
```

Fig. 10. Arithmetic Procedure

*3) **Logical Procedure**:* Logical procedure `au_logical` is functionally similar to that of its arithmetic counterpart except for the fact that only logical operations are employed to achieve the arithmetic operation. The procedure also works with three arguments, two operands, and an operator. Based on the type of operation, the procedure branches to the appropriate code label and executes the instructions. The output is placed in `$v0` and `$v1`. MIPS code snippet that implements the logical procedure is available in figure 11.



```
###########################################################################
au_logical:
    StoreRTE($a0, $a1, $a2, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7)
    beq $a2, '+', addition
    beq $a2, '-', subtraction
    beq $a2, '*', multiplication
    beq $a2, '/', division

    addition:
        move $a2, $zero
        jal add_logical
        j exit
    subtraction:
        ori $a2, 0xFFFFFFFF
        jal sub_logical
        j exit
    multiplication:
        jal mul_logical
        j exit
    division:
        jal div_logical
        j exit
exit:
    RestoreRTE($a0, $a1, $a2, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7)
    jr $ra
###########################################################################
```

Fig. 11. Logical Procedure

*4) **Logical Addition**:* Addition using logical operators can be summarized in the following steps:

1) Use the XOR operation to add the binary digits: Starting from the rightmost digit, use the XOR operation to add the two binary digits. The XOR operation returns a 1 if the two inputs are different and a 0 if they are the same.
2) Use the AND operation to determine the carry: Use the AND operation to determine whether a carry is needed for each addition. The AND operation returns a 1 if both inputs are 1 and a 0 otherwise.
3) Shift the carry: If a carry is needed, shift it to the left and add it to the next column. Repeat this process until all digits have been added.

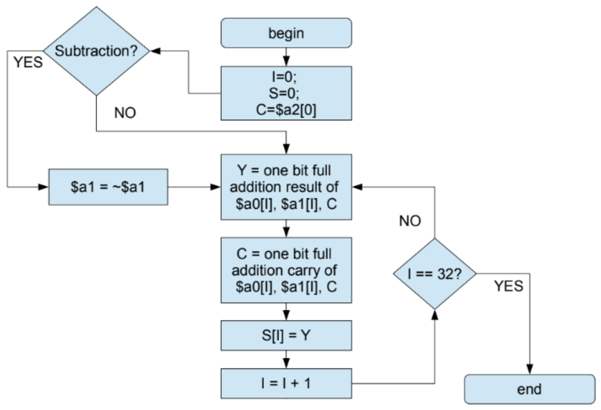
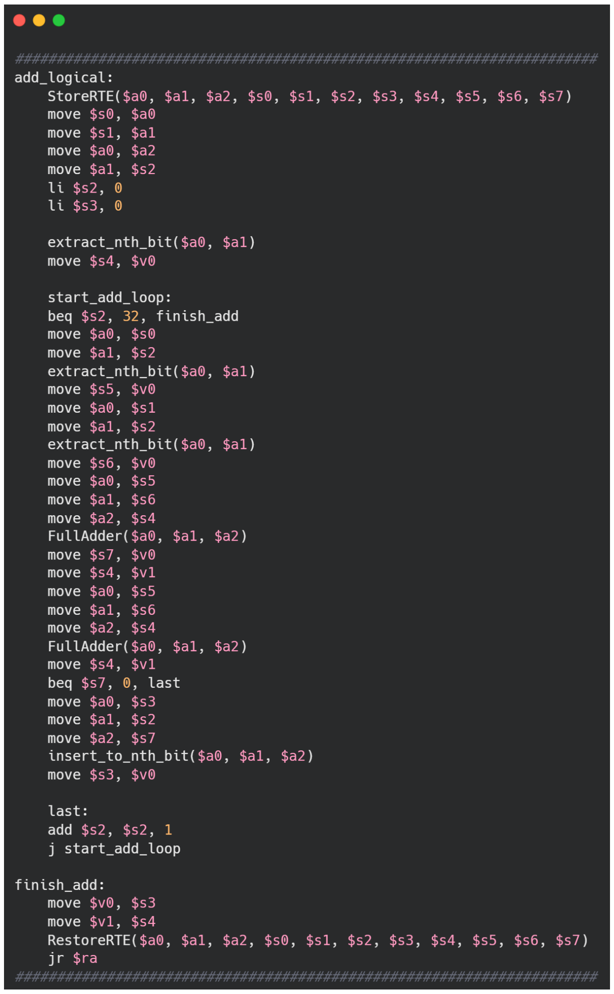

Subtraction?
YES
NO
begin
I=0;
S=0;
C=$a2[0]
$a1 = ~$a1
Y = one bit full addition result of $a0[I], $a1[I], C
C = one bit full addition carry of $a0[I], $a1[I], C
S[I] = Y
I = I + 1
NO
I == 32?
YES
end

Fig. 12. Add Sub Algo

In order to incorporate both the carry-in and carry-out bits in the addition or subtraction process, a full adder is used. The full adder is just a combination of two half adders. In essence, as shown in the code in figure 13, we utilize the above-mentioned algorithm in a loop 32 times keeping track of carry-in and carry-out. Care must be taken for the final carry bit i.e. the overflow bit, which needs to be stored in another register. Finally, the output of addition is moved to `$v0 & $v1` and the control returns back to the caller.

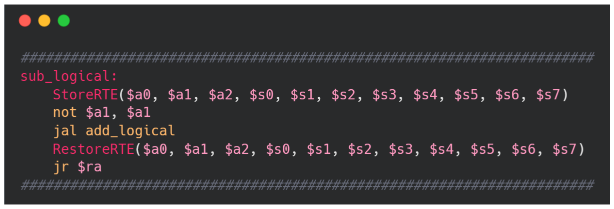
Figure 12 shows the algorithmic flow of the steps involved in logical addition and logical subtraction. Note that there is no functional difference between addition and subtraction. Thus, addition procedures can be repurposed to implement logical subtraction.

*5) **Logical Subtraction**:* Conceptually, subtraction is effectively nothing but an addition of one or more negative operands. This observation conveniently carries over (no pun intended) to logical operations and hence logical subtraction is an extension of logical addition with some pre-processing of the input argument. Thus, to perform subtraction, the second operand should be converted to its two's complement form before invoking the `add_logical` procedure. As shown in the implementation of `sub_logical` code snippet in figure 14, the procedure converts operand `$a1` into it's two's complement and calls logical addition procedure. As usual, the output of the logical subtraction is placed in `$v0 & $v1`.



```
######################################################################
add_logical:
    StoreRTE($a0, $a1, $a2, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7)
    move $s0, $a0
    move $s1, $a1
    move $a0, $a2
    move $a1, $s2
    li $s2, 0
    li $s3, 0

    extract_nth_bit($a0, $a1)
    move $s4, $v0

    start_add_loop:
    beq $s2, 32, finish_add
    move $a0, $s0
    move $a1, $s2
    extract_nth_bit($a0, $a1)
    move $s5, $v0
    move $a0, $s1
    move $a1, $s2
    extract_nth_bit($a0, $a1)
    move $s6, $v0
    move $a0, $s5
    move $a1, $s6
    move $a2, $s4
    FullAdder($a0, $a1, $a2)
    move $s7, $v0
    move $s4, $v1
    move $a0, $s5
    move $a1, $s6
    move $a2, $s4
    FullAdder($a0, $a1, $a2)
    move $s4, $v1
    beq $s7, 0, last
    move $a0, $s3
    move $a1, $s2
    move $a2, $s7
    insert_to_nth_bit($a0, $a1, $a2)
    move $s3, $v0

    last:
    add $s2, $s2, 1
    j start_add_loop

finish_add:
    move $v0, $s3
    move $v1, $s4
    RestoreRTE($a0, $a1, $a2, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7)
    jr $ra
######################################################################
```

Fig. 13. Add logical Implementation



```
######################################################################
sub_logical:
    StoreRTE($a0, $a1, $a2, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7)
    not $a1, $a1
    jal add_logical
    RestoreRTE($a0, $a1, $a2, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7)
    jr $ra
######################################################################
```

Fig. 14. Sub Logical Implementation

*6) **Logical Multiplication**:* To understand logical multiplication, let's first go over the binary multiplication algorithm shown in figure 15. Before the algorithm enters the loop, the product is set to `0`. For a 32-bit operand, the loop is executed 31 times. During each iteration of the loop, the following operations take place:

- Least Significant bit of the multiplier is checked to see if it's 1
- If the LSB of the multiplier is indeed 1, the product is added to the multiplicand
- Shift the multiplicand one bit left
- Shift the multiplier one bit right

- Repeat 31 times

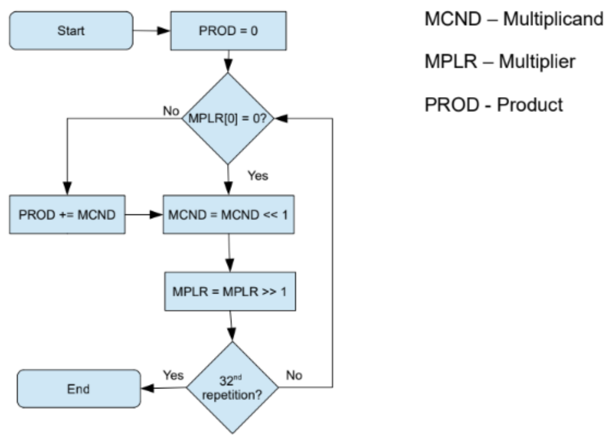


Fig. 15. Binary Multiplication Algorithm

Once the loop ends, the product argument contains the product of the two operands. Note that this algorithm is for unsigned integer multiplication. Implementation of the unsigned multiplication is done in the macro `mul_unsigned` as seen in the binary sequential multiplier in figure 16. To multiply two unsigned numbers, the least significant bit of the multiplier is used to create a mask. Depending on the bit value of the multiplier, the mask value changes. If the bit is 0, the mask is set to 0; if the bit is 1, the mask is set to -1. The mask and the multiplicand are then ANDed together and added to a `hi` register. Next, the multiplier and hi counter are shifted to the left by 1. The `hi` register's dropped-off bit is then inserted into the multiplier's most significant bit. This process is repeated 32 times to obtain the final output.

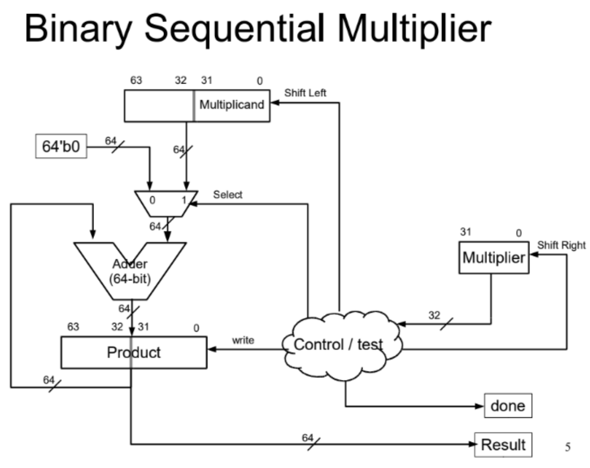
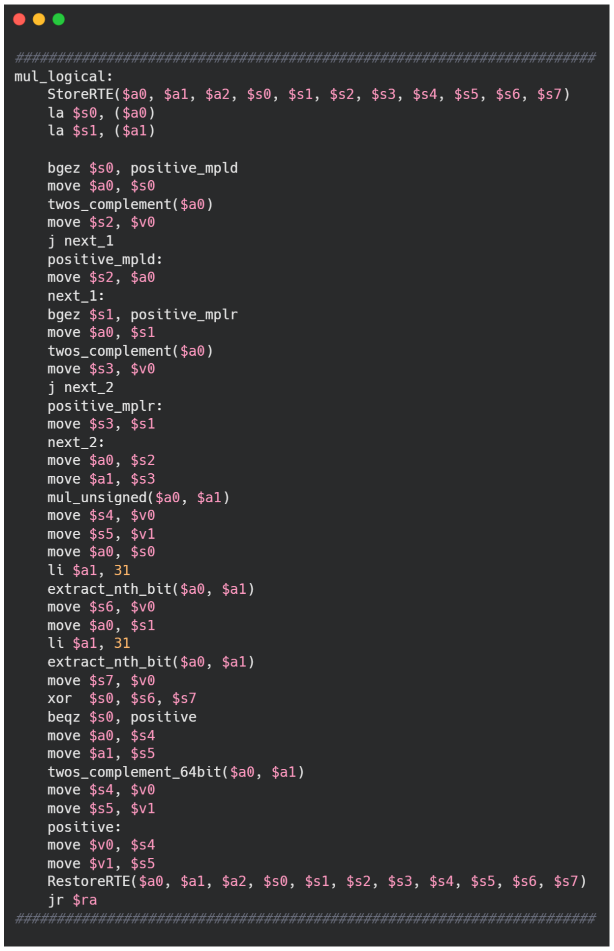


Fig. 16. Binary Sequential Multiplier Diagram

For signed multiplication, if either the multiplier or the multiplicand is negative, they are both made positive and unsigned multiplication is performed. The final sign of the result is then determined by performing the XOR operation on the most significant bits of the original inputs. If the sign value is 1, the result of the unsigned multiplication is complemented to obtain the correct signed result. Full implementation of logical multiplication code `mul_logical` is shown in figure 17. Similar to other procedures, it takes two 32-bit operands and returns a signed 64-bit result.



```
########################################################################
mul_logical:
    StoreRTE($a0, $a1, $a2, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7)
    la $s0, ($a0)
    la $s1, ($a1)

    bgez $s0, positive_mpld
    move $a0, $s0
    twos_complement($a0)
    move $s2, $v0
    j next_1
    positive_mpld:
    move $s2, $a0
    next_1:
    bgez $s1, positive_mplr
    move $a0, $s1
    twos_complement($a0)
    move $s3, $v0
    j next_2
    positive_mplr:
    move $s3, $s1
    next_2:
    move $a0, $s2
    move $a1, $s3
    mul_unsigned($a0, $a1)
    move $s4, $v0
    move $s5, $v1
    move $a0, $s0
    li $a1, 31
    extract_nth_bit($a0, $a1)
    move $s6, $v0
    move $a0, $s1
    li $a1, 31
    extract_nth_bit($a0, $a1)
    move $s7, $v0
    xor  $s0, $s6, $s7
    beqz $s0, positive
    move $a0, $s4
    move $a1, $s5
    twos_complement_64bit($a0, $a1)
    move $s4, $v0
    move $s5, $v1
    positive:
    move $v0, $s4
    move $v1, $s5
    RestoreRTE($a0, $a1, $a2, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7)
    jr $ra
########################################################################
```

Fig. 17. Mul Logical Implementation

*7) **Logical Division**:* The binary division algorithm is shown in figure 18 and contains a pre-processing step where the 32-bit divisor and 32-bit dividend are placed in the upper and lower halves of a 64 bit register respectively. A 32 count loop is executed with the following data processing in each loop:

- Subtract the divisor from the remainder
- If the result from previous step is negative, add the the divisor back and shift quotient register one bit left with 0 inserted in LSB.
- If the subtraction result is positive, keep the result and shift quotient register one bit left with 1 inserted in LSB.
- Repeat 32 times

Implementation of the above mentioned unsigned division algorithm is realized in a logical circuit shown in figure 19 and implemented in the macro `div_unsigned`.

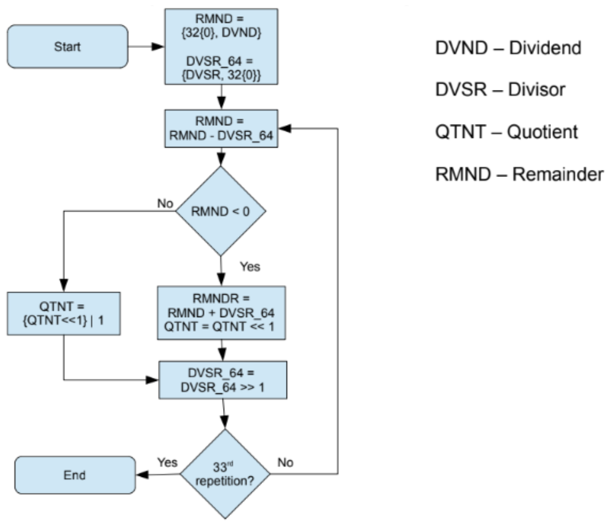


Fig. 18. Binary Division Algorithm

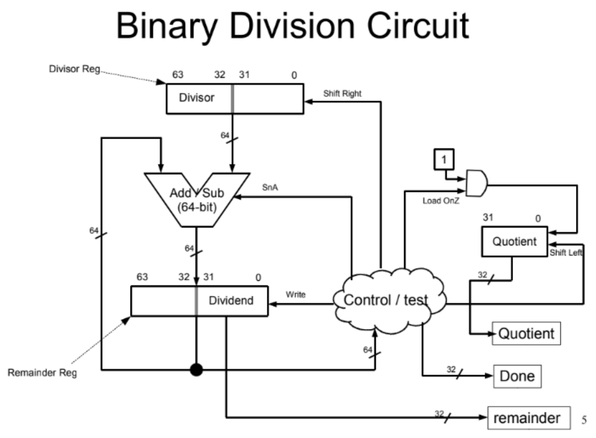
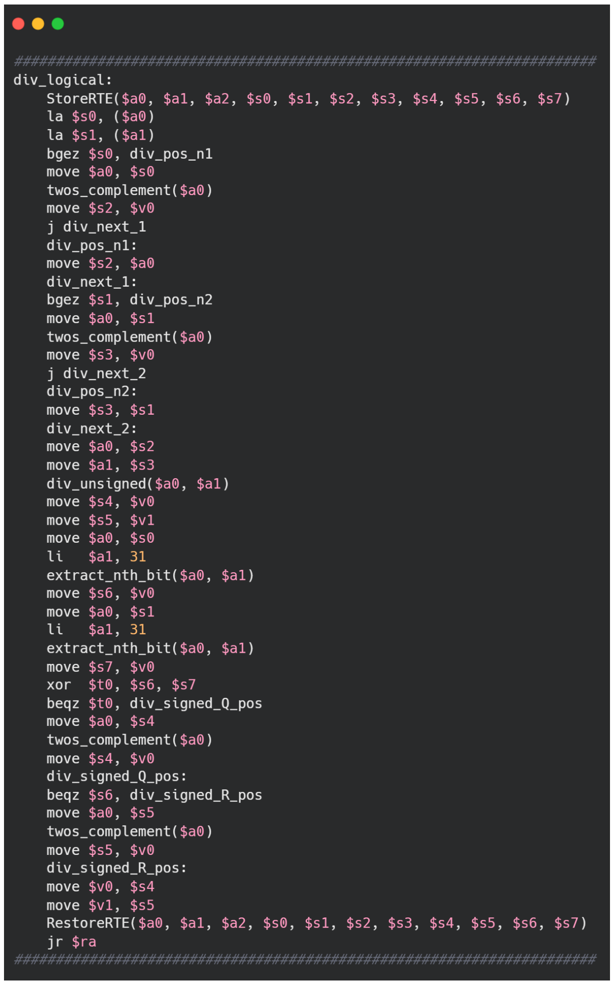


Fig. 19. Binary Division Circuit

The final `div_logical` procedure performs division on signed arguments and employs `div_unsigned` for doing the unsigned division. It also takes two arguments `$a0` as dividend and `$a1` as divisor. To perform division in the project, two registers are used to store the dividend and quotient, and together they can simulate a 64-bit register. The division process starts at the left-most bit of the dividend, where the divisor is subtracted. If the subtraction yields a positive result, a 1 is placed in the corresponding bit position of the result. If the subtraction leads to a negative result, a 0 is placed instead. This process is repeated as the dividend is shifted out and replaced by the quotient, until the entire dividend has been processed.

V. Testing

Testing is a vital step in the completion of this project. Generally speaking, testing can be categorized into the following main categories:



```
########################################################################
div_logical:
    StoreRTE($a0, $a1, $a2, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7)
    la $s0, ($a0)
    la $s1, ($a1)
    bgez $s0, div_pos_n1
    move $a0, $s0
    twos_complement($a0)
    move $s2, $v0
    j div_next_1
    div_pos_n1:
    move $s2, $a0
    div_next_1:
    bgez $s1, div_pos_n2
    move $a0, $s1
    twos_complement($a0)
    move $s3, $v0
    j div_next_2
    div_pos_n2:
    move $s3, $s1
    div_next_2:
    move $a0, $s2
    move $a1, $s3
    div_unsigned($a0, $a1)
    move $s4, $v0
    move $s5, $v1
    move $a0, $s0
    li   $a1, 31
    extract_nth_bit($a0, $a1)
    move $s6, $v0
    move $a0, $s1
    li   $a1, 31
    extract_nth_bit($a0, $a1)
    move $s7, $v0
    xor  $t0, $s6, $s7
    beqz $t0, div_signed_Q_pos
    move $a0, $s4
    twos_complement($a0)
    move $s4, $v0
    div_signed_Q_pos:
    beqz $s6, div_signed_R_pos
    move $a0, $s5
    twos_complement($a0)
    move $s5, $v0
    div_signed_R_pos:
    move $v0, $s4
    move $v1, $s5
    RestoreRTE($a0, $a1, $a2, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7)
    jr $ra
########################################################################
```

Fig. 20. Div Logical Implementation

1) Unit testing: Test small stand-alone functionality. For example, unit test each utility macro.
2) Integration testing: Test a feature that is integrated together by combining multiple small functionalities. For example, integration testing of `au_normal`.
3) System testing: Full feature testing of the overall system.

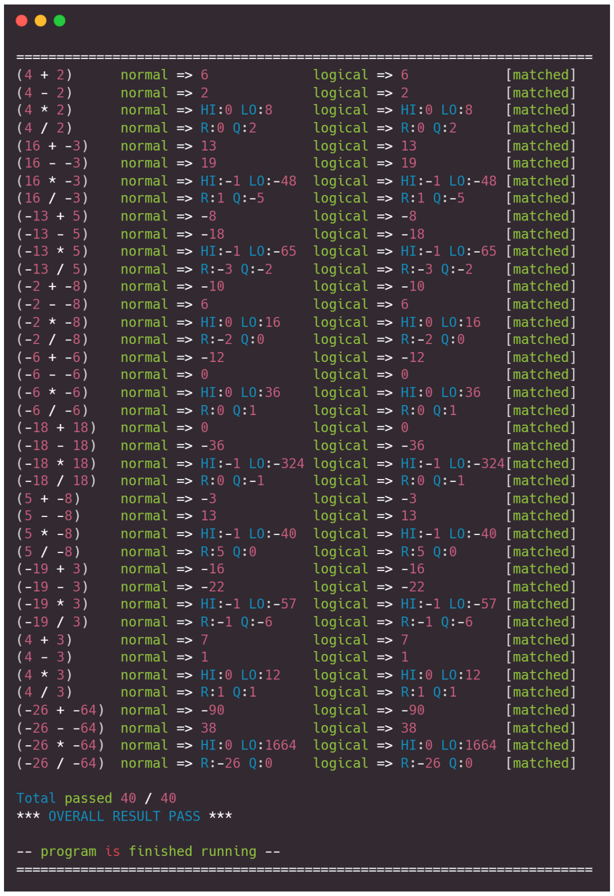
As mentioned earlier, debugging errors and issues in MIPS assembly code is not easy. Even with available tools in MARS, debugging may be very time consuming. So the best thing to do is to implement small code modules and test them right away. Unit testing can help in this step. Once tested, these small modules shall be used to build more complex functionality and ultimately complete the project requirements successfully.

As part of the Project Starter code, a test suite file `proj-auto-tests.asm` of 40 test cases has been provided in the project zip file. This test file can be thought of as an automated system test to ensure that all procedures and functions are working as desired. Each test case tests a specific requirement of the project. Some examples include:

- Add/Subtract/Multiply/Divide two positive integers

- Add/Subtract/Multiply/Divide a positive and a negative integers
- Add/Subtract/Multiply/Divide two negative integers

Moreover, each test case cross-validates the results from the arithmetic operation and the logical operation since both types of operations should yield the same result. All 40 test cases must pass for the project to be successful. Test results for my implementation are displayed in Figure 21.



```
=========================================================================
(4 + 2)     normal => 6            logical => 6            [matched]
(4 - 2)     normal => 2            logical => 2            [matched]
(4 * 2)     normal => HI:0 LO:8    logical => HI:0 LO:8    [matched]
(4 / 2)     normal => R:0 Q:2      logical => R:0 Q:2      [matched]
(16 + -3)   normal => 13           logical => 13           [matched]
(16 - -3)   normal => 19           logical => 19           [matched]
(16 * -3)   normal => HI:-1 LO:-48 logical => HI:-1 LO:-48 [matched]
(16 / -3)   normal => R:1 Q:-5     logical => R:1 Q:-5     [matched]
(-13 + 5)   normal => -8           logical => -8           [matched]
(-13 - 5)   normal => -18          logical => -18          [matched]
(-13 * 5)   normal => HI:-1 LO:-65 logical => HI:-1 LO:-65 [matched]
(-13 / 5)   normal => R:-3 Q:-2    logical => R:-3 Q:-2    [matched]
(-2 + -8)   normal => -10          logical => -10          [matched]
(-2 - -8)   normal => 6            logical => 6            [matched]
(-2 * -8)   normal => HI:0 LO:16   logical => HI:0 LO:16   [matched]
(-2 / -8)   normal => R:-2 Q:0     logical => R:-2 Q:0     [matched]
(-6 + -6)   normal => -12          logical => -12          [matched]
(-6 - -6)   normal => 0            logical => 0            [matched]
(-6 * -6)   normal => HI:0 LO:36   logical => HI:0 LO:36   [matched]
(-6 / -6)   normal => R:0 Q:1      logical => R:0 Q:1      [matched]
(-18 + 18)  normal => 0            logical => 0            [matched]
(-18 - 18)  normal => -36          logical => -36          [matched]
(-18 * 18)  normal => HI:-1 LO:-324 logical => HI:-1 LO:-324[matched]
(-18 / 18)  normal => R:0 Q:-1     logical => R:0 Q:-1     [matched]
(5 + -8)    normal => -3           logical => -3           [matched]
(5 - -8)    normal => 13           logical => 13           [matched]
(5 * -8)    normal => HI:-1 LO:-40 logical => HI:-1 LO:-40 [matched]
(5 / -8)    normal => R:5 Q:0      logical => R:5 Q:0      [matched]
(-19 + 3)   normal => -16          logical => -16          [matched]
(-19 - 3)   normal => -22          logical => -22          [matched]
(-19 * 3)   normal => HI:-1 LO:-57 logical => HI:-1 LO:-57 [matched]
(-19 / 3)   normal => R:-1 Q:-6    logical => R:-1 Q:-6    [matched]
(4 + 3)     normal => 7            logical => 7            [matched]
(4 - 3)     normal => 1            logical => 1            [matched]
(4 * 3)     normal => HI:0 LO:12   logical => HI:0 LO:12   [matched]
(4 / 3)     normal => R:1 Q:1      logical => R:1 Q:1      [matched]
(-26 + -64) normal => -90          logical => -90          [matched]
(-26 - -64) normal => 38           logical => 38           [matched]
(-26 * -64) normal => HI:0 LO:1664 logical => HI:0 LO:1664 [matched]
(-26 / -64) normal => R:-26 Q:0    logical => R:-26 Q:0    [matched]

Total passed 40 / 40
*** OVERALL RESULT PASS ***

-- program is finished running --
=========================================================================
```

Fig. 21. Test Output

VI. Conclusion

This project has been a valuable learning experience for me, as I have gained a deep understanding of the MARS IDE and MIPS operations. I learned about the software development life-cycle process pertinent to low level MIPS assembly programming. Similar to Lego blocks, simple functionality is combined to create more complex functions and features to fulfill the requirements. I've also learned new debugging techniques as I encountered various errors, which gave me the opportunity to learn and understand their root causes. The project has also provided insight into the complex calculations that computers perform for seemingly simple tasks. One potential avenue for future expansion of this project could be the implementation of more sophisticated mathematical operations in the project.

Overall, I found the project to be an interesting and enlightening experience.

References

[1] K. Vollmar, "MARS (MIPS Assembler and Runtime Simulator)," Mars mips simulator - missouri state university, Aug-2014. [Online]. Available: http://courses.missouristate.edu/kenvollmar/mars/. [Accessed: 29-Apr-2023].

[2] "Java Downloads for All Operating Systems," Java.com. [Online]. Available: https://www.java.com/en/download/manual.jsp. [Accessed: 03-May-2023].

[3] D. A. Patterson, Computer Organization and Design. San Francisco: Elsevier Science and Technology, 2013.

[4] K. Patra. CS 47. Class Lecture, Topic: "Addition Subtraction Logic." San Jose State University, San Jose, CA, April 14, 2016.

[5] K. Patra. CS 47. Class Lecture, Topic: "Multiplication Logic." San Jose State University, San Jose, CA, April 19, 2016.

[6] K. Patra. CS 47. Class Lecture, Topic: "Division Logic." San Jose State University, San Jose, CA, April 21, 2016.

[7] "Computer architecture," Wikipedia, 07-Apr-2023. [Online]. Available: https://en.wikipedia.org/wiki/Computer_architecture. [Accessed: 04-May-2023].