Arithmetic Operations Created

By MIPS Logic Operations

Harsh Milan Singh

San Jose State University

*Abstract—*This report captures and explains the implementation and the testing of four basic mathematical operands (addition, subtraction, multiplication, and division) created with MIPS natural logical operations in MARS – MIPS Assembler and Runtime Simulator.

# Introduction

In this project, we will quantify four basic mathematical functions which are addition, subtraction, multiplication, and division using standard MIPS operations.

The objectives of this project are as follows:

1. Setup and create a blank workspace in MARS. This objective sets up the project software requirements and the creation of a default environment for the use of assembly language.

2.Create user made functions that perform arithmetic calculations. The primary operations used are logical operations. The additional operations involve add, sub, mult and div. However, the condition is that these operations cannot be used in the logical section of this project since doing so we defeat the purpose of making a logical section.

3. Test the code and provide its validity and possible sources of errors.

# Project requirement

These are the steps and information needed to follow the implementation of the project.

## MARS Download

The project workspace, MARS, can be found online at: http://courses.missouristate.edu/KenVollmar/MARS/.

This project was done on the current latest version 4.5 of the Windows download. Run the .exe file and let it be installed following the steps. Opening it for the first time should open a black workspace.

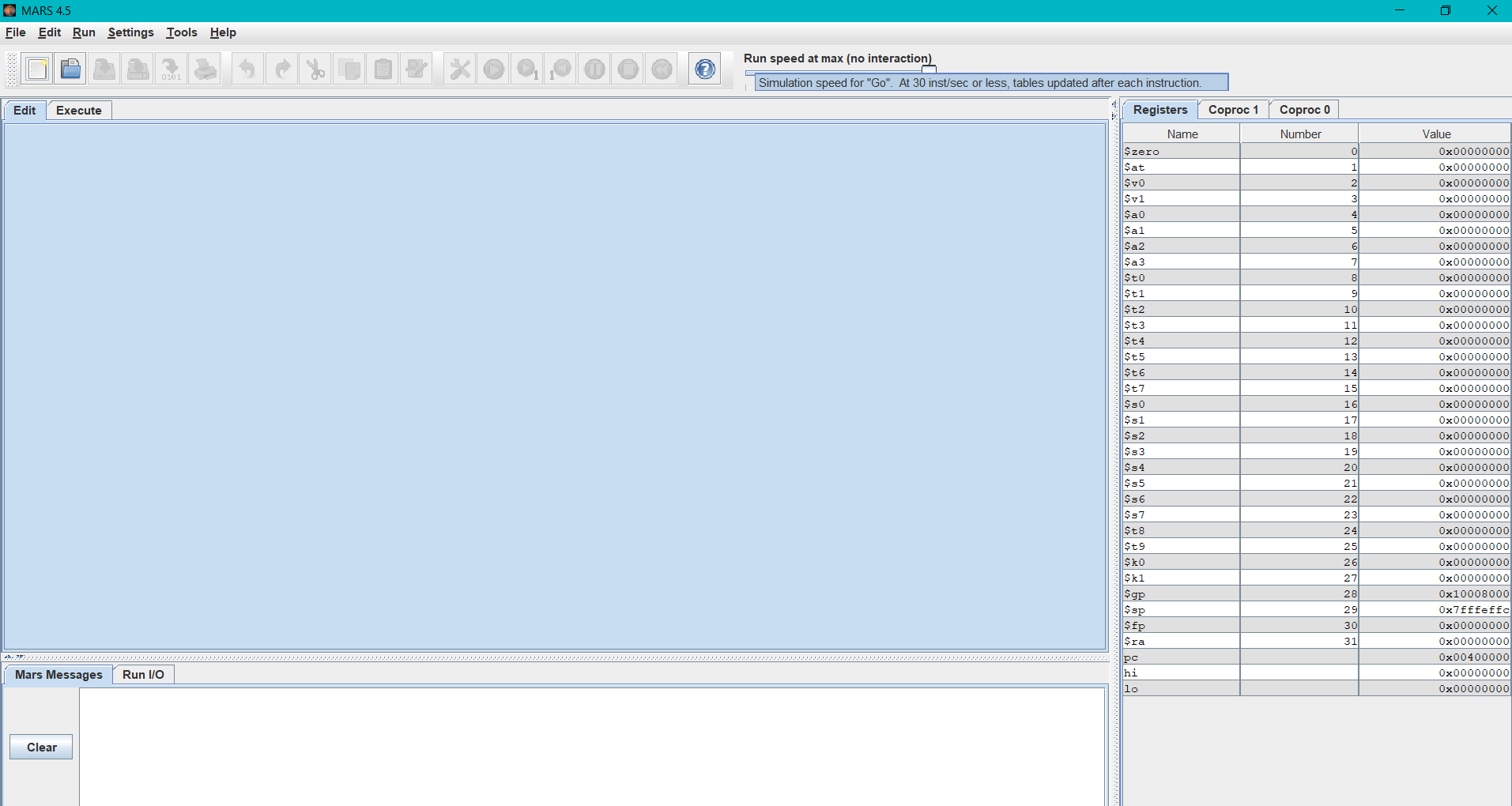


Fig. 1. Blank MARS screen

## Project Pre-requistes and files:

On the upper left corner of the blank MARS window, click “File” and then “Open”. In the new Open window, navigate to find the directory with the described files and individually open them all as shown in fig. 2.

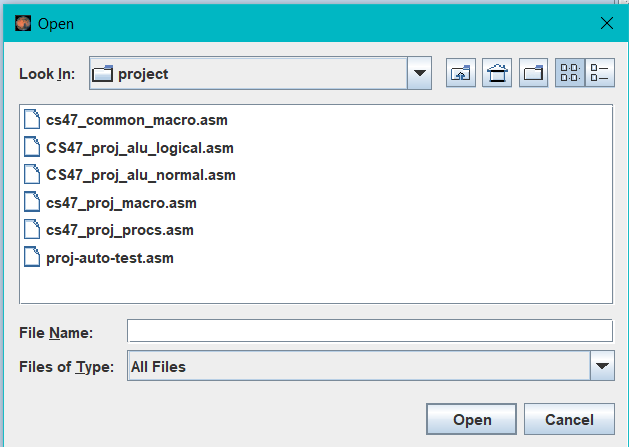


Fig. 2. Opening Files

They should automatically open in separate tabs and we can switch through them from the tabs just below the upper menu.

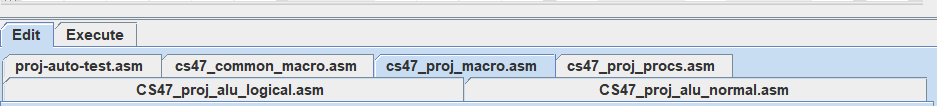


Fig. 3. Tabbed files

There are six .asm files available to download in Canvas. These are then unzipped and put in singular directory named “Project\_CS47”. Load up MARS and open all these six files. Each files’ description is provided below:

1. cs47\_common\_macro.asm

Predefined file containing macro definitions for printing and reading string, integers, register values, pushing/popping and exiting the program. No further changes were made to this file.

1. cs47\_proj\_procs.asm

Predefined file containing functions required to print various data types such floating values, string, characters, integers etc. No further changes were made to this file.

1. proj-auto-test.asm

The tester file containing main functions to test the completed project. There are 40 tests and overall pass and fail texts to test the validity of the code. No further changes were made to this file.

1. cs47\_proj\_macro.asm

File for user made macro definitions. This file contains macros for printing integers, strings, pushing and popping. Additionally, it contains methods get\_bit which extracts the nth bit from the given register and method insert\_bit which inserts the nth bit into the given bit place.

1. CS47\_proj\_alu\_normal

Incomplete file used to implement the normal arithmetic operations using “add, sub, mult, div” pseudocodes. Completing this file is the objective of this project.

1. CS47\_proj\_alu\_logical

Incomplete file used to implement the logical arithmetic operations. The use of “add, sub, mult, div” pseudocodes is prohibited in this file. Completing this file is the objective of this project.

*C. Configuration Pre-requisites*

Move to Settings tab in the menu and turn on “Assembles all files in directory” and “Initialize program counter to global main if defined” for proper execution of the files. The rest of the settings are configured by default but one can verify them with fig 4.

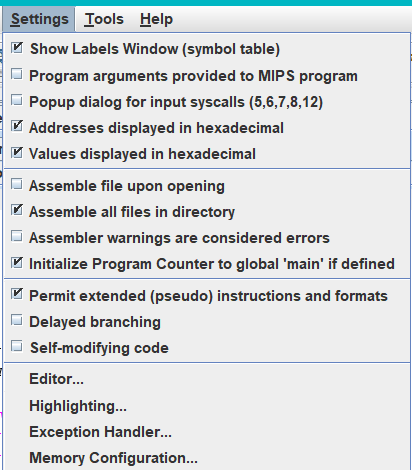


Fig. 4. Settings tab

# Arithmetic procedures Description

There two cases of implementations. The first one, normal procedure is defined in the file “CS47\_proj\_alu\_normal”. This procedure uses MIPS standard operations.

The second case, the logical procedure is defined in the file “CS47\_proj\_alu\_logical”. This procedure contains the definitions for mathematical symbols represented in logical operations such as AND, OR, NOT, and XOR. On the contrary to *Normal Portion*, this implementation does not contain standard MIPS instructions for the actual terms.

## Normal Portion

As mentioned in the title comments, this file consists of three arguments:

* Register $a0 : First Number
* $a1: Second number
* $a2: operation code ('+': add, '-': sub, '\*': mult, '/': div)

In return, the output will be stored and later used for testing in the following registers:

* $v0: Variable storage for ($a0+$a1) | ($a0-$a1) | ($a0\*$a1): (contains LO part) | ($a0 / $a1) (Quotient)
* $v1: Variable storage for ($a0 \* $a1): (contains HI part) | ($a0 % $a1) (Remainder)

## Logical Portion

As mentioned in the title comments of alu\_logical, this file consists of three input arguments:

* $a0: First number
* $a1: Second number
* $a2:operation code ('+': add, '-': sub, '\*': mult, '/': div)

In return, the output will be stored and later used for testing in the following registers:

* $v0: Variable storage for ($a0+$a1) | ($a0-$a1) | ($a0\*$a1): (contains LO part) | ($a0 / $a1)(Quotient)
* $v1: Variable storage for ($a0 \* $a1): (contains HI part) | ($a0 % $a1)(Remainder)

Since both multiply and division are 64 bit operations, both 32 bit variable storages are used to mimic a single 64 bit variable.

# code design of alu\_normal

The attempted code design for the use of MIPS standard operations as self-made functions is done by dividing the given prompt into five functions.

1. au\_normal:

Caller function for the 4 operation functions. The code first branches to add\_normal if register $a2 is equal to 0x2B, which corresponds to ASCII character ‘+’.The second line branches to sub\_normal if the register $a2 is equal to 0x2D, which corresponds to ASCII character ‘-’. The third line branches to mult\_normal if the register $a2 is equal to 0x2A, which corresponds to ASCII character ‘\*’. The fourth line branches to div\_normal if the register $a2 is equal to 0x2F, which corresponds to ASCII character ‘/’.

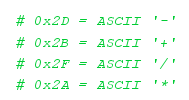
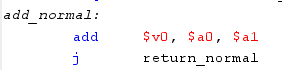


Fig. 5. ASCII Representation of hex values



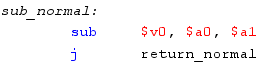
1. add\_normal:

Adds register $a0 and $a1 and stores the value in $v0 using ‘add’. Returns $v0 into return\_normal.



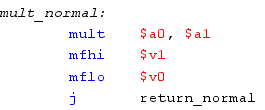
1. sub\_normal:

Subtracts register $a0 and $a1 and stores the value in $v0 using ‘sub’. Returns $v0 into return\_normal.



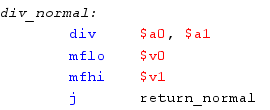
1. mult\_normal:

Multiplies register $a0 and $a1 and stores the HI value in $v1 and LO value in $v0 using “mult, mfhi, mflo”. Returns $v0 and $v1 into return\_normal.



1. div\_normal:

Divides register $a0 and $a1 and stores the LO value (Quotient) in $v0 and HI value (Remainder) in $v1 using “div, mfhi, mflo”. Returns $v0 and $v1 into return\_normal.



1. return\_normal:

Helper function that returns control to the caller function at the address $ra.



##### V. code design of alu\_logical

1. *Set up for alu\_logical*

Unlike its counterpart, the alu\_logical requires far more steps and conditions. Structurally alu\_logical is congruent to alu\_normal, however, the procedure requires more labels to improve readability.

Therefore, there are a total of 21 labels each described below.

1. *au\_logical*

Initially we set $sp to the end of the stack and store registers $a0-$a3, $s0-$s5, $ra, $fp accordingly using the store word mnemonic in addresses in multiples of 4. This is for maintaining format and so none of the registers overlap due to address overlap. Lastly we shift $fp to the top of the stack of registers.

After this initial setup, we need to initialize our arguments and return registers ($t0, $s0, $v0, $v1) to zero. This process eliminates any garbage values previously stored in them. The value zero also becomes a Sentinel value which can be used to check if the loops are working correctly or not.

From there on, au\_logical mimics au\_normal and branches into add\_logical, sub\_logical, mult\_logical, and div\_logical if the value of $a2 corresponds to 0x2B (ASCII ‘+’), 0x2D (ASCII ‘-‘), 0x2A (ASCII ‘\*’), or 0x2F (ASCII ‘/’).

The last line jumps to restore\_return\_logical, restores the stack of registers by loading them as outputs and jumps to $ra.

1. *add\_logical and add\_logical\_loop*

In add\_logical, we initialize $s1 to zero and jump to add\_logical\_loop.

In add\_logical\_loop, we iterate to find carry in(sum) part of the sum through an AND instruction and use XOR to find its carry-out bit. This can be done using a half Adder.

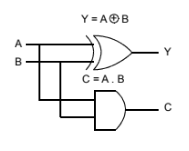


Fig.6. Half Adder Logic Gate Config.

We need such division for cases of 1 + 1 and 1 + 1 +1. The addition of 1 + 1 returns sum(carry in) as 0 and carry out as 1 whereas the addition of 1 + 1 + 1 returns sum 1 and carry out as 1. However, since half Adder only contain two inputs, we need multiple half Adders for three inputs. To account for both carry out and the sum, we must use two half adders or one full adder to store the sum of the two arguments.

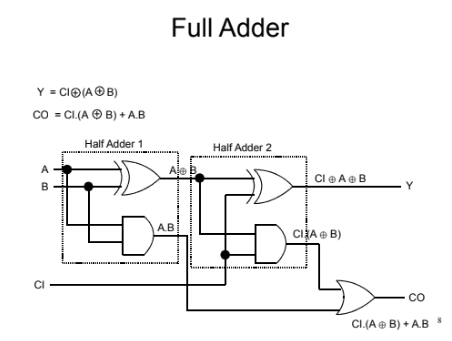


Fig.7. Full Adder Logic Config.

The given truth tables show each individual case of addition between binary inputs.

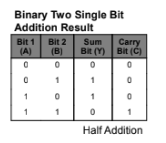


Fig. 8. Truth Table for two inputs

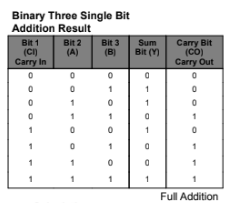


Fig. 9. Truth Table for three inputs(carry out as third input)

The K-map displayed below is used provide the previous full Adder configuration.

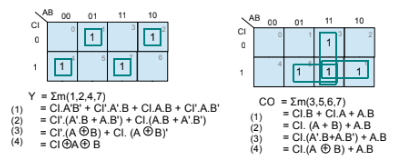
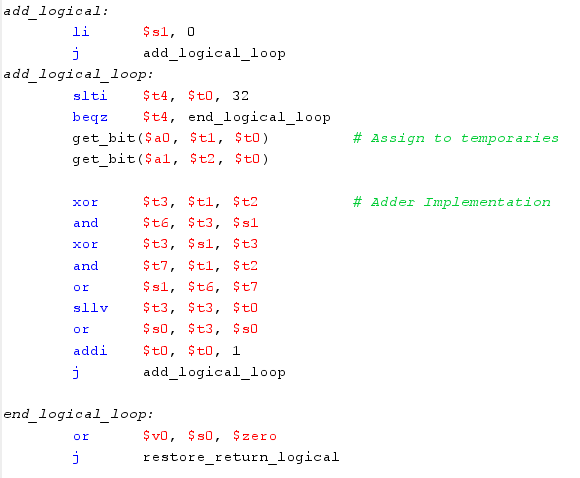


Fig. 10. K-map reduction for full Adder

Instead of using the two arguments given $a0 and $a1, we use temporary variables $t1 and $t2 (with the helper method get\_bit) to extract the value from $a0 and $a1 and store the addition in $s1. To combine the values of carry in and carry out we use ‘or’ and ‘sllv’(shift left by value) instructions. Lastly, to end the loop we increment the value of $t0 by 1 each time the loop works leading it to exceed 32 to terminate the loop.



This design is to detect the presence of carry in bit of 1. If it contains 1, then the process gets inverted for subtraction as it becomes a signed value.

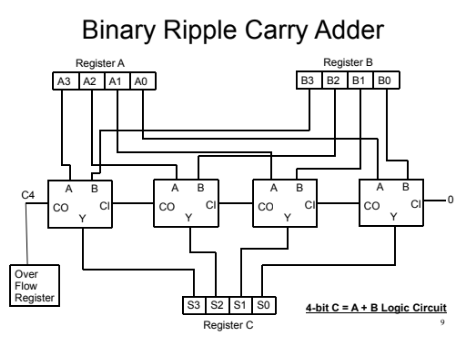
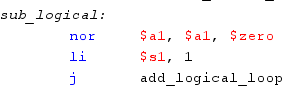


Fig.11. Full Adder

It is important to take note of the fact that this design does not detect overflow and just discards it.

1. *sub\_logical*

sub\_logical is logically identical to add\_logical as subtraction of two variables can be interpreted as addition of positive and negative values. Therefore sub\_logical calls add\_logical\_loop as well. The only difference is that the arguments will be signed values.



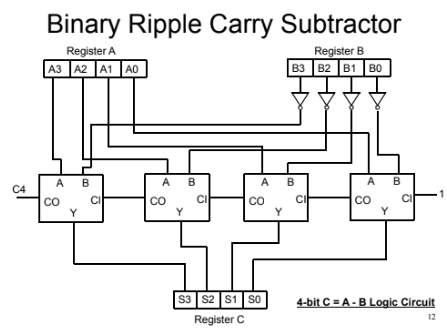


Fig. 12. Full Adder modification for subtraction



We use the 2’s complement form for subtraction which is done by a NOR gate and the full adder.

$a0 = (INV)$a0 +1

To differentiate between addition and subtraction, if $s1 contains 0 then addition takes place. If $s1 contains 1 then subtraction takes place.

1. *mult\_logical*

This procedure is responsible for the process of multiplication, either signed or unsigned. In multiplication, the product can be significantly larger than its multiples. Therefore, this function returns a 64 bit digit as two 32 bit digits in the form of HI and LO.

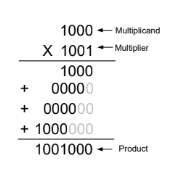
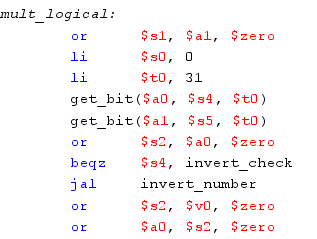


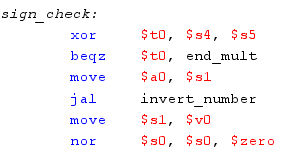
Fig. 13. Multiplication process

Just as in addition/subtraction, we start by saving the given arguments $a0 and $a1 into temporary values. However, we may need these values multiple times so saved temporaries $s4 and $s5 are used.

For unsigned multiplication, the upper half of the multiplier register gets equal to zero and then AND the multipler with the lower bits of the multiplier.



For signed values, the sign\_check method uses XOR to provide correct signs to the operands. Operands may be converted to their two’s complement form if the sign returned is negative. After the checks, the multiplication process is the same of unsigned values.



The invert\_check compares the sign of both arguments and appends the arguments to $zero to make them 64 bit values.

The AND operator is used on the multiplicand with the multiplier's 0th bit 32 times (due to 32-bit register). This is due to the fact that the AND operation for binary bit pairs is analogous to how binary multiplication operates.



The loop of mult\_logical\_loop uses $s3 as its counter and runs 32 times, one time for each bit multiplication. The contents of the multiplier register will be moved to the right once to replicate the 64-bit register with two 32-bit registers. The first bit from the product register would be removed and put into the multiplier register's MSB (most significant bit) (31st bit).

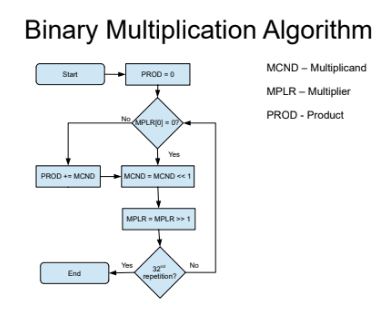
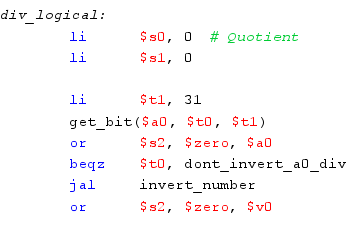


Fig. 14. Flow chart of logical multiplication

1. *div\_logical*

Similar to mult\_logical, this procedure takes in two arguments $a0 and $a1 as dividend and divisor instead of multiplicand and multiplier. In return, two 32 bit registers are used to imitate a 64 bit register. $v0 is the result quotient and $v1 is the result remainder.



To create the process of division, the relation Dividend = Quotient \* Divisor + Remainder has been used. The process below shows the binary division.

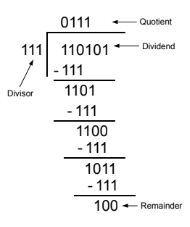


Fig. 15. Bit pattern for division

In the process of div\_logical\_loop, we use $s4 as the counter and increment it till the MSB value(31). The loop then moves the contents of the remainder register to left and inserts quotient from MSB to LSB. In this process, the dividend will be thrown out bit by bit till all bits of the register have be replaced by the quotient and the remainder.

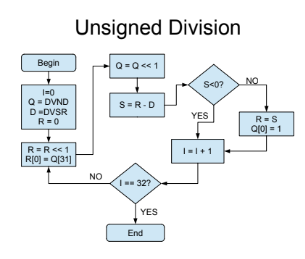


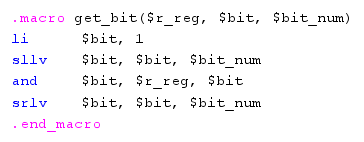
Fig. Flowchart for unsigned division

For signed division, similar to mult\_logical, sign invert checks are run on both the arguments and if the variable needs to be negative, we follow the same division loop with the variables two’s complement form.

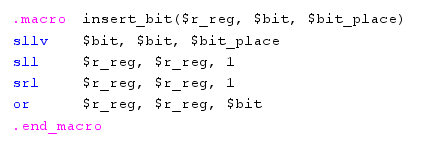
1. *Helper Methods*

These are the helper functions used throughout the project.

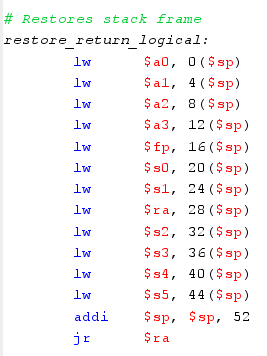
get\_bit: Extracts the nth bit from a given register argument. It achieves this either by shifting left or right and appending the bit into the result register using AND. This method was primarily used to get nth bit from the arguments $a0 and $a1 and store them into temporaries and saved temporaries.



insert\_bit: The contrary method to get\_bit. This method inserts a bit from the register argument to the bit place. This method also shifts left and right but inserts the bit using OR.



Restore\_return\_logical: A function is alu\_logical used to restore the stack form originally created in au\_logical. It is called in all the fundamental operations to end the process.

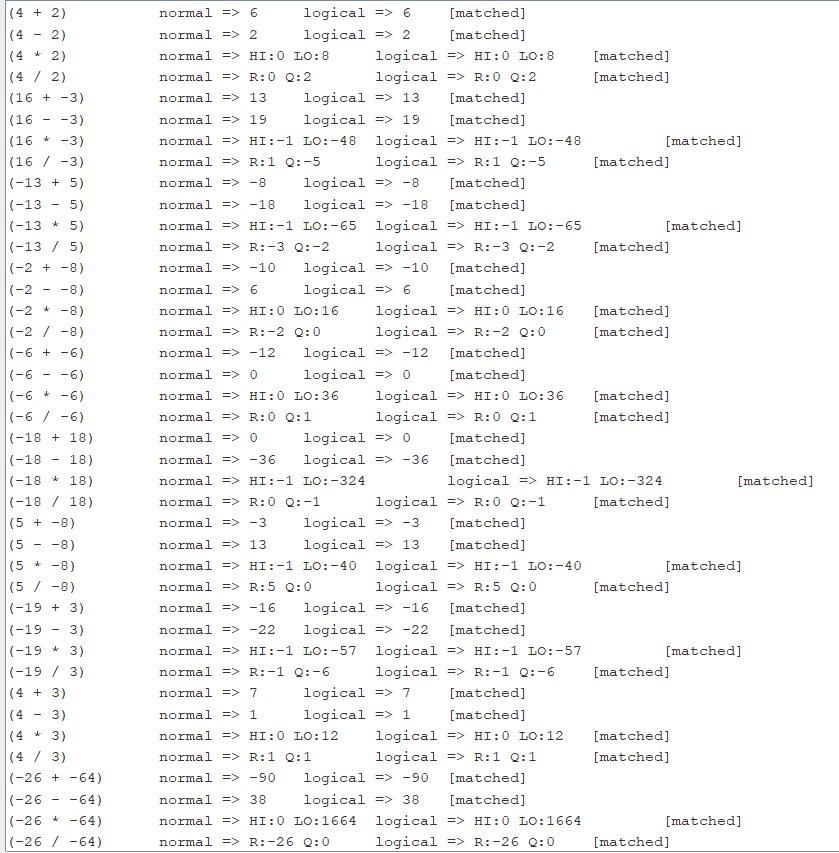


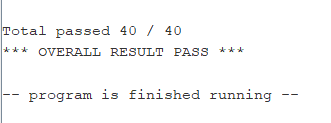
Invert\_number: Changes a number into its two’s complement form. Since the range for signed values is less (24 bit), this function creates and restores its own separate stack frame.



##### VI. Testing

Having completed the second objective of this project, we need to test the user made functions from both alu\_normal and alu\_logical. As mentioned in the requirements section, we are given an auto tester file named “proj-auto-test.asm”. Save all the files and compile this file. After compiling, run the file and open the Run I/O tab.





A large documentation of test cases are now displayed with the last line indicating the overall result. There are total of 40 cases and each case tests both normal and logical outputs and returns correct only if they are mathematical corrected and matching outputs. If the procedures were implemented correctly, the console should say “Total passed 40/40 Overall Result Pass”.

Throughout the debugging process, we may ponder upon errors. Here is a documentation of the sources of error I faced during the process of making the project.

1. Incomplete restoration of the stack frame: For each operation, the entire stack frame must be loaded and restored as they may contain values from previous test runs creating incorrect output which may or may not show up as an error.
2. Large values: Since there is no overflow detection, it is important to notice the range of inputs that can be provided. For addition and subtraction specially, larger integer values should be avoided during testing as the result may not be a 32 bit value and can lead to an error.

##### VII. Conclusion

The provided report presents application to material learned in CS 47 class. The use of assembly language, MARS environment, the implementation of ALU and register file system and procedural oriented programming has been highlighted in this project. The process of building and testing the code multiple times and debugging minor errors has exemplified my understanding of the assembly language. The code length and complexity showcases the primitivity of assembly language. However, the use of calculations at the binary level provides a look at the inner working of a computer and the speed of working with the register file memory. These fundamental arithmetic operations are used in almost every program in the computer. The design of a computer is amazing due to the fact that we can use just 0s and 1s to map an entire world.

San Jose, CA, May, 2021.