Basic Arithmetic Logic Unit Implemented in MIPS

Arthur Chan, San Jose State University

***Abstract*—This report focuses on the implementation of basic mathematical operations such as addition, subtraction, multiplication, and division using instructions that are built into MIPS and through logical operations using the MARS simulator, forming a basic arithmetic logic unit.**

I. INTRODUCTION

The point of this project is to understand the basis for many of the basic mathematical operations that take place in common programming. Using the MARS simulator, we are able to understand how addition, subtraction, multiplication, and division work at a basic level through designing a basic calculator covering these operations. This report will cover how to run and install MARS, assemble and run the included files, cover the concepts of Boolean algebra, and how it is implemented. Discussion over testing and findings will take place at the end.

II. INSTALLATION

A. *Installation of MARS*

The MARS simulator is used as an IDE often used for teaching the basic of assembly language. This can be found on http://courses.missouristate.edu/KenVollmar/mars/. Unzipping the folder reveals a .jar executable file and will require Java to already be installed. The latest revision can be found on https://java.com/en/ under the "downloads" and will auto-detect the correct operating system.

B. Loading the Project

In order to load the project in MARS, navigate to the "file" tab and find "open". Navigate to the CSProjectI folder was downloaded and open the following files: CS47_proj_alu_normal.asm, CS47_proj_alu_logical.asm and cs47_proj_macro.asm.

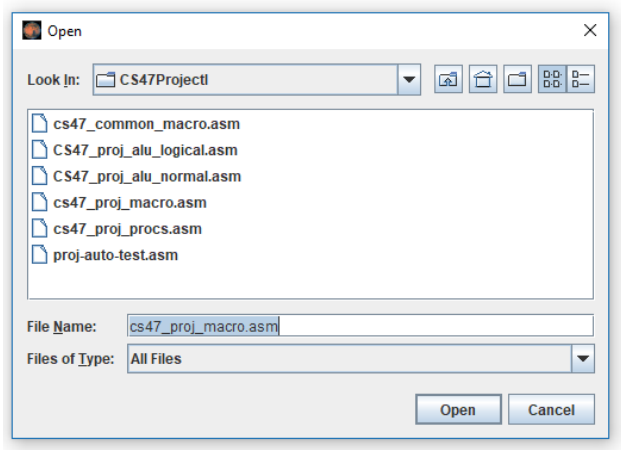


Figure 1. Locating the files

C. Changing the Settings

Navigate to the settings tab and enable the "Initialize Program Counter to global 'main' if defined" and the "Assemble all files in directory" options. The first option will have the starting point of the program to start at the label "main, " instead of the first line. The latter option has all .asm files in the directory be assembled in order to avoid errors with some labels not being present in the symbol table.

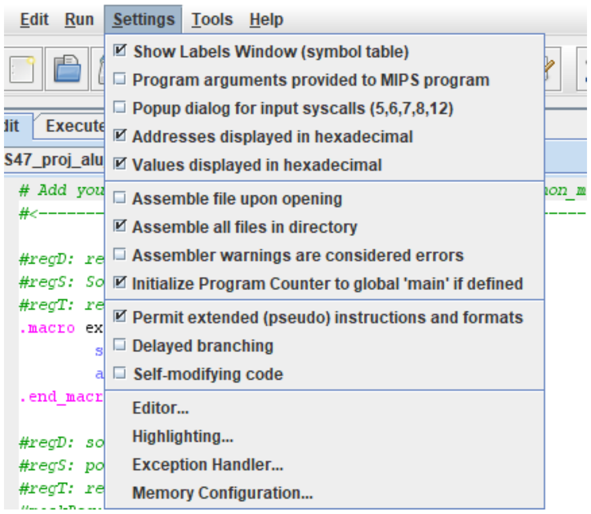

Edit Run Settings Tools Help
Execute
☑ Show Labels Window (symbol table)
☐ Program arguments provided to MIPS program
☐ Popup dialog for input syscalls (5,6,7,8,12)
☑ Addresses displayed in hexadecimal
☑ Values displayed in hexadecimal
☐ Assemble file upon opening
☑ Assemble all files in directory
☐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

Figure 2. The settings menu

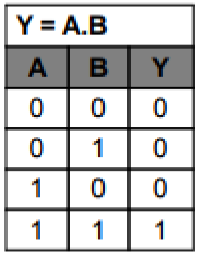
III. Requirements

A. *Logic Operations*

In order to understand the operation here, the basic logical operations must be defined. There are three basic operations, being AND, OR, and XOR, each taking two inputs each. Typically, in circuits and programming, 0 is represented as false while 1 represents true.

1) AND

The AND operation will return 1 only if both inputs are 1, otherwise 0 is returned for all other instances.



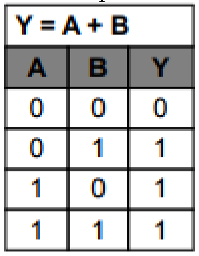
Y = A.B

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Figure 3. Truth Table of AND

2) OR

The OR operation will return 1 in most situations but return 0 in the case that both inputs are 0.



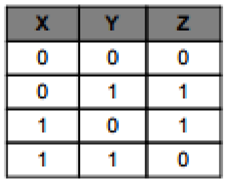
Y = A + B

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4. Truth Table of OR

3) XOR

The XOR operation will return 1 when its inputs do not match and will return 0 when they do.



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5. Truth Table of XOR

B. *Binary System*

In addition, much of the operations are under binary, a number system in which a single bit can represent either 0 or 1. This means that a binary number has a base 2, compared to other number types such as decimal or hex, which have base 10 and 16 respectively. In binary, the bits are combined in order to make larger numbers. 1 in binary just means 1 while 11 in binary means 3. Each step up of binary in the case of 10 or 1000 is determined by the position of the 1 from left to right, starting from the 0th position. As such, in order to calculate 1000, the formula for doing so would be 2^3 which equals 8. In order to determine the value of binary numbers with multiple 1's in them, it would just require simple addition. In order to calculate 1010, the formula for this would be 2^3 + 2^1 = 10.

While the above works well for positive numbers, negative numbers are a different story. Negative numbers are represented in a slightly different format, where the leading number, or most significant bit (MSB), represents the sign of the number. Having a 1 as the MSB would mean that the number is negative while a 0 signifies positive. In order to convert a normal number into a two's complement, the bits of the number must be inverted, then 1 is added to it. For example, 0011100 which represents 28 will be turned into 1100100, representing -28.

C. *Binary Logic and Mathematics*

The logical operations that will be implemented are done with bit manipulation and using binary, where knowing how to preform addition, subtraction, multiplication, and division between two binary number extremely important.

1) Addition

The addition focuses in the addition of two binary addresses.

With the above example, whatever result of the binary gets recorded down into that bit position. For instance, adding 0 and 0 will result in the bit for the new address to be 0. Adding 1 and 0 result in 1 instead. In the case that adding 1 and 1 will result in the binary number 10, where 0 is recorded for that bit position and the 1 will act as the carryover, being carried over into the next position will be taken into consideration for the addition there. The situation is the same for adding 1 and 1 with the carryout of 1, where the result will be 11, where 1 will be the carryout and 1 will be the resultant bit position.

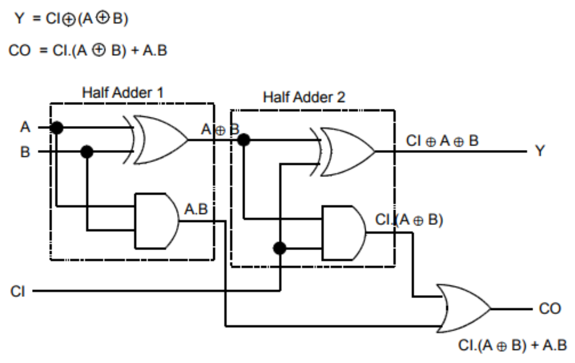


Figure 6. Circuit of a Full Adder

How to this is represented in a circuit is through half adders. Half adders only take in two inputs and output the sum and the carryout. The AND gate represented in [insert figure here] does the carryout while the XOR gate handles the sum. However, a half adder will only solve part of the equation, as the bit carry-in bit is considered. A full adder is used, taking the carry-in bit into consideration with the use of an additional half adder. This circuit is formed through the process of constructing a truth table with the rules of binary addition with the carry-in bit and the two other operands. The K-map is formed and a circuit is formed as shown in figure 6. This allows for multiple full adders to be stitched together in order to form a result of appropriate bits, as seen in figure 7, forming the binary ripple carry adder.

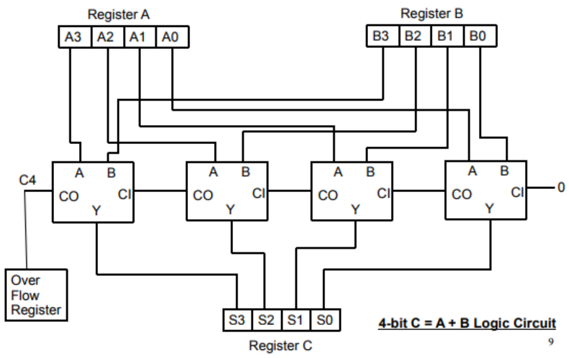


Figure 7. The Binary Ripple Adder

2) Subtraction

Subtraction works in a similar fashion, although the nature of the carryover works differently. In the case of 0 - 1, the 1 will be carried over from other bits to the right.

Even down to the rules, the binary ripple carry adder can be used for subtraction without much modification. Addition can still be performed only converting the second operand into its two's complement form, eliminating the need for an additional circuit and is the reason why the binary ripple carry adder needs only slight changes. The implementation of subtraction in the logical implementation closely follows the binary ripple adder in figure 7, a more accurate chart in figure 8 shows SnA attracting is the control signal and as another input where 0 results in addition and 1 will invert the value of the other inputs, resulting in subtraction.

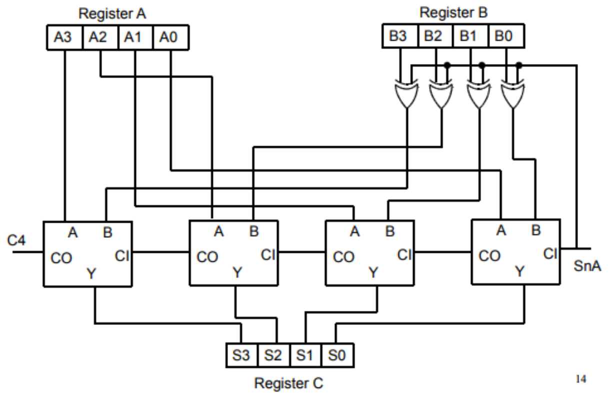


Figure 8. Updated Binary Ripple Adder

3) Multiplication

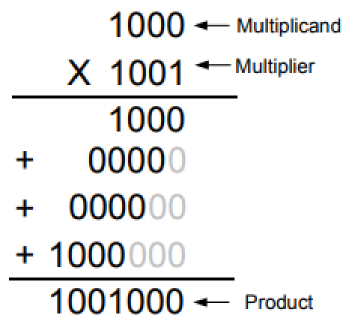


Figure 9. On-paper Binary Multiplication

Binary multiplication works very similar to the conventional way of doing it. Using figure 9 as an example, basic rules still stand such as 1 * 0 = 0 and 1 * 1 = 1. With each partial product calculated, the number is shifted to the left and the moves on to the next bit for multiplication. As a result, the product be of length equal to the sum of the length of both the multiplicand and the multiplier. As a result, in typical 32-bit system, the product will end up being of length 64 bit and is split into two 32-bit registers. In MIPS, the lower 32 bits are stored in the Lo register while the upper 32 bits are stored in the Hi register, both being specialized registers for this purpose.

4) Division

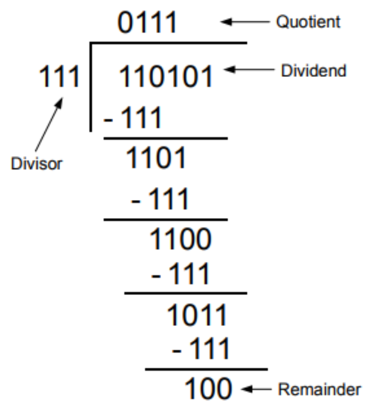


Figure 10. On-paper Binary Division

Division also works similar to typically mathematical procedures. It starts with placing the divisor, starting with the most significant bit of the dividend. If the section the divisor is compared to is greater than the divisor, then no subtraction takes place and a 0 is put into the quotient. Otherwise, a 1 is marked into the quotient and subtraction takes place. For example, in figure 10, the first step is comparing 110 and 111. Since 111 is greater, 0 is marked and no subtraction takes place. The result is then shifted. Now the next step would be comparing 1101 and 111. Since 1101 is greater than the dividend of 111, subtraction takes place and 1 is added to the quotient. This goes on until it reaches to least significant bit, where the remainder is the result, or lack thereof, of the subtraction should it not equal 0.

IV. Implementation

A. Structure and Format

The normal implementation is under the name "au_normal" where calculations are done using the mathematical instructions already included within the MIPS language. The logical implementation is done in "au_logical." Both implementations follow the same general format when accepting arguments:

1) Register $a0
The first operand in the mathematical operation
2) Register $a1
The second operand in the mathematical operation
3) Register $a2
The character signifying the operation, and will be in the form of "+", "-", "*", or "/"

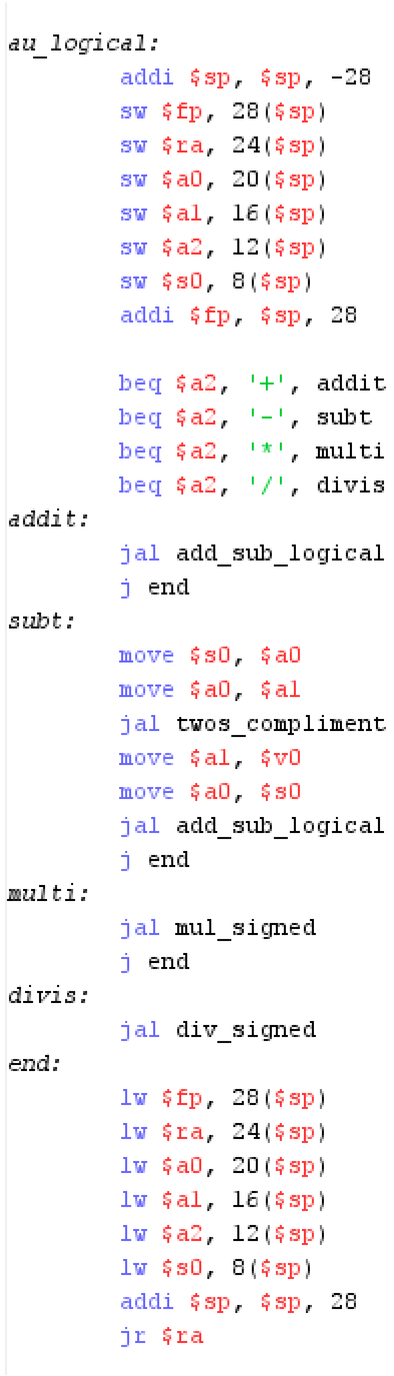
The return registers $v0 and $v1 are used as well, depending on which operation is taking place. For addition, $v0 will contain the result of $a0+$a1 and subtraction will have $v0 contain the result of $a0-$a1. By the nature of multiplication and division, $v1 must be used to hold the additional result of the two operations. For multiplication, $v0 will contain the LO part of the resultant value while $v1 will contain the HI part. For division, $v0 will contain the quotient and $v1 will contain the remainder.

B. Normal Implementation

The normal procedure is straightforward. This implementation is contained within "CS47_proj_alu_normal.asm". With its arguments, it checks $a2 for the operator character is run it through a series of four conditionals, perform the operation, and then jump to the appropriate section. Afterwards, it then returns to the caller.

C. Logical Implementation:

The logical procedure is found within "CS47_proj_alu_logical.asm". However, unlike the normal implementation of a basic ALU, it much more complex and requires multiple macros and procedures in order to work. However, the section under "au_logical" still fundamentally follows the same structure as "au_normal" as it features conditionals in order to differentiate the operations and will return to the caller once it is done with its operations. In most cases, once the conditional is triggered, it will call the appropriate form of logical operation, except for subtraction, where twos_complement is called before add_sub_logical in order to convert $a1 into a negative, so that only addition needs to be done.



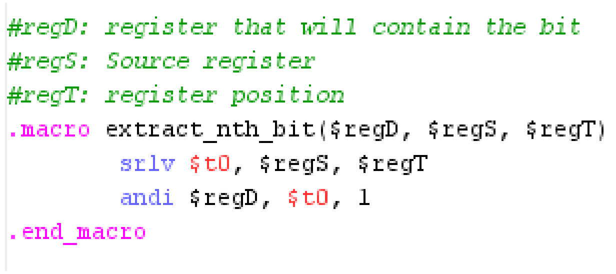
```
au_logical:
	addi $sp, $sp, -28
	sw $fp, 28($sp)
	sw $ra, 24($sp)
	sw $a0, 20($sp)
	sw $a1, 16($sp)
	sw $a2, 12($sp)
	sw $s0, 8($sp)
	addi $fp, $sp, 28

	beq $a2, '+', addit
	beq $a2, '-', subt
	beq $a2, '*', multi
	beq $a2, '/', divis
addit:
	jal add_sub_logical
	j end
subt:
	move $s0, $a0
	move $a0, $a1
	jal twos_compliment
	move $a1, $v0
	move $a0, $s0
	jal add_sub_logical
	j end
multi:
	jal mul_signed
	j end
divis:
	jal div_signed
end:
	lw $fp, 28($sp)
	lw $ra, 24($sp)
	lw $a0, 20($sp)
	lw $a1, 16($sp)
	lw $a2, 12($sp)
	lw $s0, 8($sp)
	addi $sp, $sp, 28
	jr $ra
```

Figure 11. Implementation of au_logical

1) Utility Macros

i. extract_nth_bit

Extract_nth_bit returns the bit value contained in a given position for a register. This macro takes in three arguments, one to store the result, one being the source register, and one containing which position to take from, being in the range of 0 to 31. The source register is placed into a temporary register where it is then shifted to the right depending on the position given. Then it is combined with AND and 1 in order to obtain the required bit and stored in $regD.



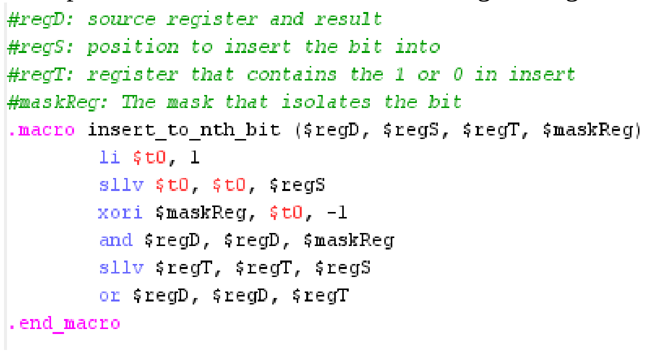
```
#regD: register that will contain the bit
#regS: Source register
#regT: register position
.macro extract_nth_bit($regD, $regS, $regT)
        srlv $t0, $regS, $regT
        andi $regD, $t0, 1
.end_macro
```

Figure 12. Implementation of extract_nth_bit

ii. insert_to_nth_bit

This macro will insert the given bit into a register given the position. A total of four registers are taken in, $regD will be the result of the insertion and the source register, $regS will be the position of the bit to insert, $regT contains either 0 or 1 to insert, and $maskReg will be used to mask the bits such that it only affects the position where the bit will be to insert.

A temporary register is used, containing 1, and is shifted to the left by the amount contained within $regS. Inverting the bits with XOR and -1 with this register will give the $maskReg. $regD is then overwritten with itself and $maskReg using the AND operation. $regT is then shifted to its position and then combined with $regD using OR.



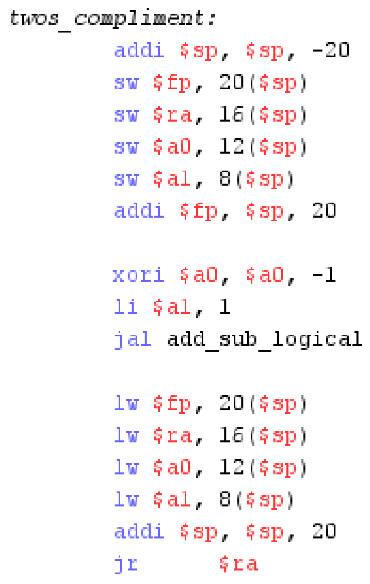
```
#regD: source register and result
#regS: position to insert the bit into
#regT: register that contains the 1 or 0 in insert
#maskReg: The mask that isolates the bit
.macro insert_to_nth_bit ($regD, $regS, $regT, $maskReg)
        li $t0, 1
        sllv $t0, $t0, $regS
        xori $maskReg, $t0, -1
        and $regD, $regD, $maskReg
        sllv $regT, $regT, $regS
        or $regD, $regD, $regT
.end_macro
```

Figure 13. Implementation of insert_to_nth_bit

2) Utility Procedures

i. twos_compliment and twos_compliment_if_neg

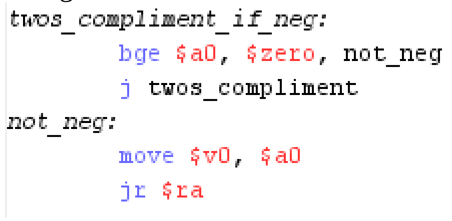
Twos_complement takes the argument contained within $a0 and converts it over to its two's compliment form. This is done through using XOR to invert it and then adding one to the register. $v0 will contains the results of this operation.



```
twos_compliment:
        addi $sp, $sp, -20
        sw $fp, 20($sp)
        sw $ra, 16($sp)
        sw $a0, 12($sp)
        sw $a1, 8($sp)
        addi $fp, $sp, 20

        xori $a0, $a0, -1
        li $a1, 1
        jal add_sub_logical

        lw $fp, 20($sp)
        lw $ra, 16($sp)
        lw $a0, 12($sp)
        lw $a1, 8($sp)
        addi $sp, $sp, 20
        jr      $ra
```

Figure 14. Implementation of twos_compliment

Twos_complement_if_neg is a simple check for if the argument in $a0 if negative and if so, will call twos_complement. Otherwise it will return the argument unchanged.

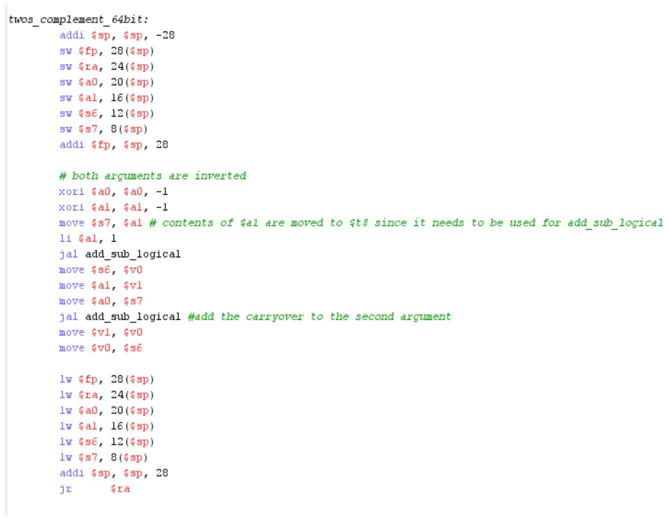


```
twos_compliment_if_neg:
        bge $a0, $zero, not_neg
        j twos_compliment
not_neg:
        move $v0, $a0
        jr $ra
```

Figure 15. Implementation of twos_compliment_if_neg

i. twos_compliment_64bit

This procedure takes in the HI and LO from the results of mul_unsigned and converts both of the results into their two's complement form. $a0 contains the LO part of the number while $a1 contains the HI part of the product. Both of the arguments are inverted, then add_sub_logical is called on $a0 with 1 being the other argument for the function. The final carryout from the addition is then added to $a1 with add_sub_logical again. $v0 returns LO in two's complement and $v1 returns HI in two's complement.

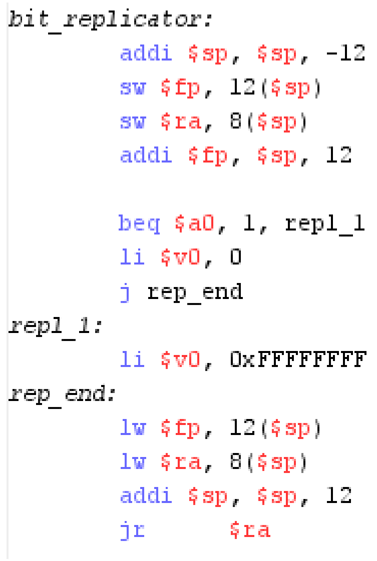


```
twos_complement_64bit:
        addi $sp, $sp, -28
        sw $fp, 28($sp)
        sw $ra, 24($sp)
        sw $a0, 20($sp)
        sw $a1, 16($sp)
        sw $s6, 12($sp)
        sw $s7, 8($sp)
        addi $fp, $sp, 28

        # both arguments are inverted
        xori $a0, $a0, -1
        xori $a1, $a1, -1
        move $s7, $a1 # contents of $a1 are moved to $t8 since it needs to be used for add_sub_logical
        li $a1, 1
        jal add_sub_logical
        move $s6, $v0
        move $a1, $v1
        move $a0, $s7
        jal add_sub_logical #add the carryover to the second argument
        move $v1, $v0
        move $v0, $s6

        lw $fp, 28($sp)
        lw $ra, 24($sp)
        lw $a0, 20($sp)
        lw $a1, 16($sp)
        lw $s6, 12($sp)
        lw $s7, 8($sp)
        addi $sp, $sp, 28
        jr      $ra
```

Figure 16. Implementation of twos_compliment_64bit

i. bit_replicator

This procedure takes in $a0 as the argument, which will be 0 or 1. What is returned is the bit replicated to all 32 bits of the register. In the case that 1 is the argument, 0xFFFFFFFF is returned in $v0.



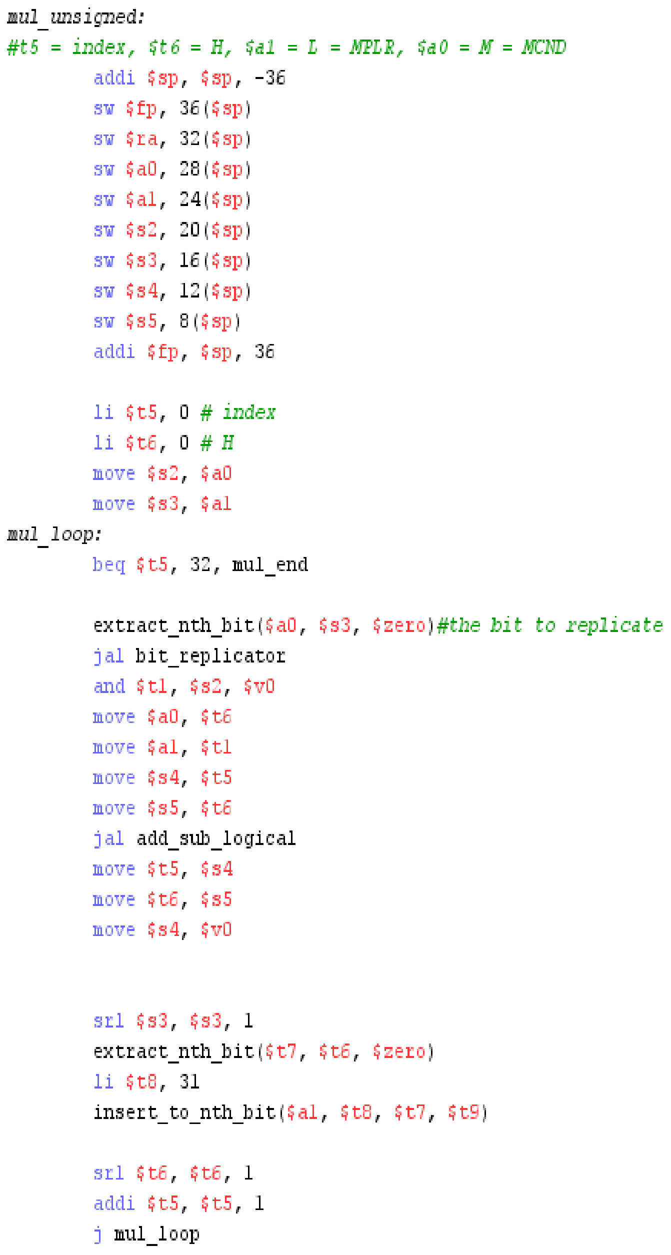
```
bit_replicator:
        addi $sp, $sp, -12
        sw $fp, 12($sp)
        sw $ra, 8($sp)
        addi $fp, $sp, 12

        beq $a0, 1, repl_1
        li $v0, 0
        j rep_end
repl_1:
        li $v0, 0xFFFFFFFF
rep_end:
        lw $fp, 12($sp)
        lw $ra, 8($sp)
        addi $sp, $sp, 12
        jr      $ra
```

Figure 17. Implementation of bit_replicator

1) Utility Macros

i. add_sub_logical

Add_sub_logical accepts two arguments, being both operands and will return the sum of those operands. This procedure goes through an iterative loop, and has an index to keep track of how many times to iterate. Once the index reaches 32, the address of the sum will have fully been built and return to the caller. The procedure beings by extracting the bit in both arguments according to the position of the index using the extract_nth_bit macro. Then they are added through logic operations similar to that of the full adder in figure []. The insert_to_nth_bit macro is used in order to insert the sum depending on the value of the index. The new carryout value also stored. The index is incremented and will continue until it reaches 32. The procedure will return not complete sum and the final carryout value for use in the twos_complement_64bit procedure.

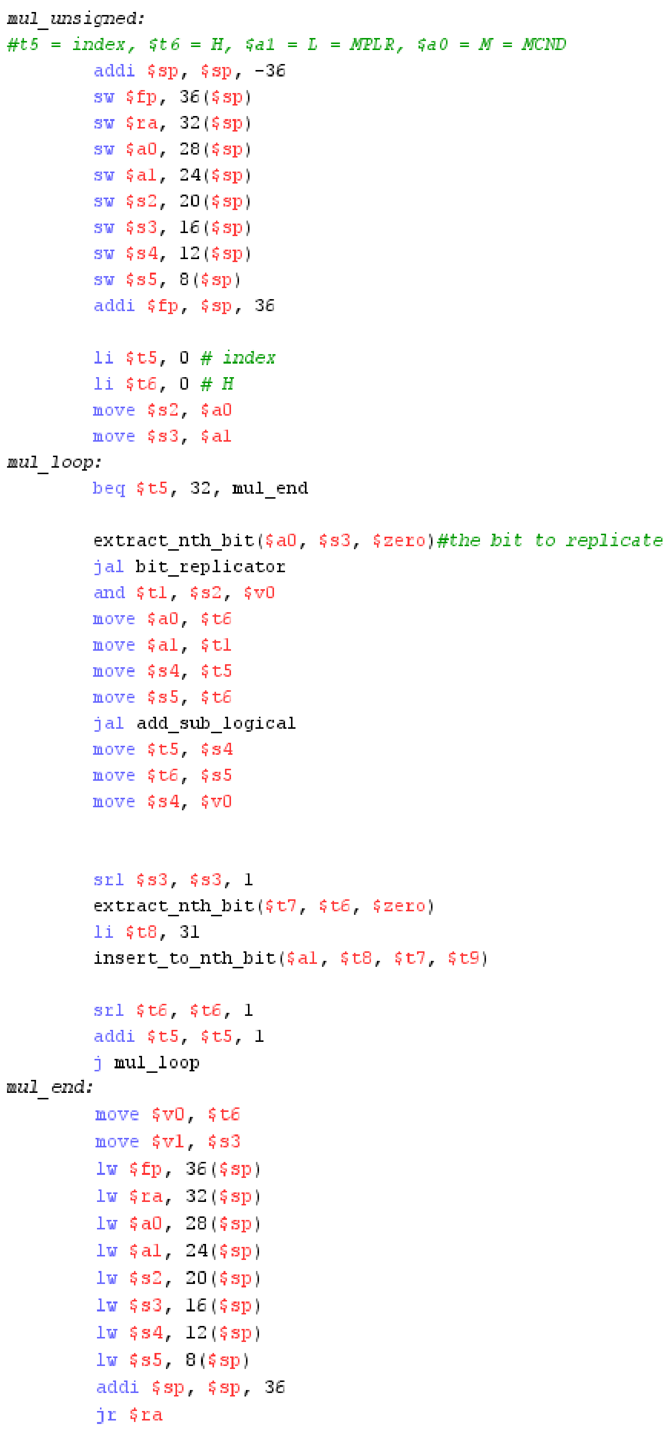


```
mul_unsigned:
#t5 = index, $t6 = H, $a1 = L = MPLR, $a0 = M = MCND
        addi $sp, $sp, -36
        sw $fp, 36($sp)
        sw $ra, 32($sp)
        sw $a0, 28($sp)
        sw $a1, 24($sp)
        sw $s2, 20($sp)
        sw $s3, 16($sp)
        sw $s4, 12($sp)
        sw $s5, 8($sp)
        addi $fp, $sp, 36

        li $t5, 0 # index
        li $t6, 0 # H
        move $s2, $a0
        move $s3, $a1
mul_loop:
        beq $t5, 32, mul_end

        extract_nth_bit($a0, $s3, $zero)#the bit to replicate
        jal bit_replicator
        and $t1, $s2, $v0
        move $a0, $t6
        move $a1, $t1
        move $s4, $t5
        move $s5, $t6
        jal add_sub_logical
        move $t5, $s4
        move $t6, $s5
        move $s4, $v0


        srl $s3, $s3, 1
        extract_nth_bit($t7, $t6, $zero)
        li $t8, 31
        insert_to_nth_bit($a1, $t8, $t7, $t9)

        srl $t6, $t6, 1
        addi $t5, $t5, 1
        j mul_loop
```

Figure 18. Implementation of add_sub_logical

i. mul_unsigned

This procedure starts by initializing two registers to 0. $t5 will keep track of the index while $t6 acts as the eventual Hi for the result. The multiplier contained within $a1 will actual as the resultant Lo value. What is done first is extracting the LSB of the multiplier and replicating it using the bit_replicator procedure. After which the result of is used with AND and the multiplicand in order to make the first result. The result of the AND operation is then added together with Hi, then the multiplier is shifted to the right. The LSB of Hi is then read using the extract_nth_bit macro and then assigned to the 31st bit of the multiplier. The index is then incremented to j back to the mul_loop label and repeats until to the 32nd iteration, where the $t6 and $a1 are return as the Hi and Lo, respectively.

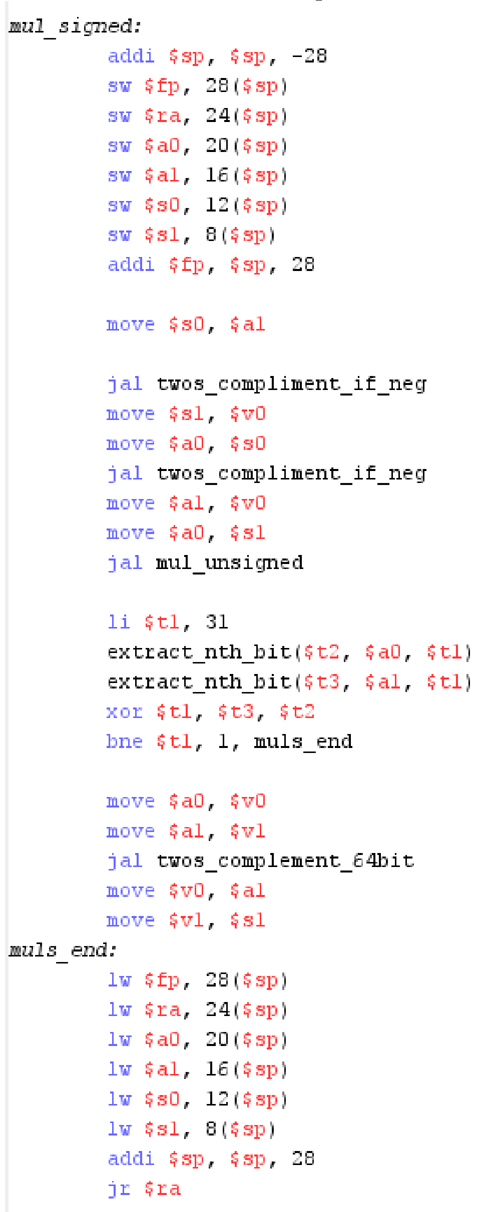


```
mul_unsigned:
#t5 = index, $t6 = H, $a1 = L = MPLR, $a0 = M = MCND
        addi $sp, $sp, -36
        sw $fp, 36($sp)
        sw $ra, 32($sp)
        sw $a0, 28($sp)
        sw $a1, 24($sp)
        sw $s2, 20($sp)
        sw $s3, 16($sp)
        sw $s4, 12($sp)
        sw $s5, 8($sp)
        addi $fp, $sp, 36

        li $t5, 0 # index
        li $t6, 0 # H
        move $s2, $a0
        move $s3, $a1
mul_loop:
        beq $t5, 32, mul_end

        extract_nth_bit($a0, $s3, $zero)#the bit to replicate
        jal bit_replicator
        and $t1, $s2, $v0
        move $a0, $t6
        move $a1, $t1
        move $s4, $t5
        move $s5, $t6
        jal add_sub_logical
        move $t5, $s4
        move $t6, $s5
        move $s4, $v0


        srl $s3, $s3, 1
        extract_nth_bit($t7, $t6, $zero)
        li $t8, 31
        insert_to_nth_bit($a1, $t8, $t7, $t9)

        srl $t6, $t6, 1
        addi $t5, $t5, 1
        j mul_loop
mul_end:
        move $v0, $t6
        move $v1, $s3
        lw $fp, 36($sp)
        lw $ra, 32($sp)
        lw $a0, 28($sp)
        lw $a1, 24($sp)
        lw $s2, 20($sp)
        lw $s3, 16($sp)
        lw $s4, 12($sp)
        lw $s5, 8($sp)
        addi $sp, $sp, 36
        jr $ra
```

Figure 19. Implementation of mul_unsigned

i. *mul_signed*

This procedure is directly called in "au_logical". mul_unsigned is only called within this function. In order to handle positives and negatives in multiplication and to make sure if the result is the right sign, both arguments given are tested in twos_compliment_if_neg, then mul_unsigned is called. then to determine the sign of the result, both of the MSBs of the original arguments are used. XOR is called on both of them. Should the result of this operation be 1, that notifies that the result should be negative, and both return values from mul_unsigned are run through the twos_compliment_64bit procedure in order to turn them into their two's compliment form.

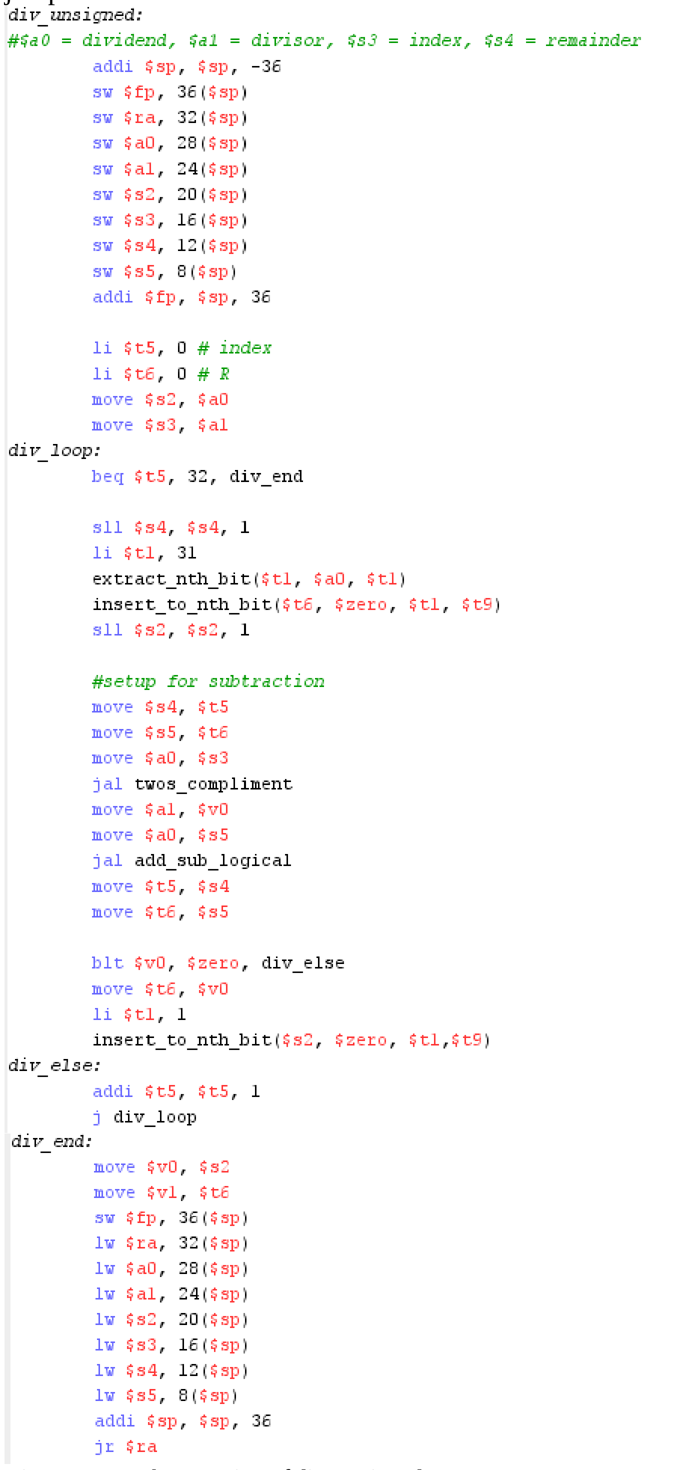


```
mul_signed:
        addi $sp, $sp, -28
        sw $fp, 28($sp)
        sw $ra, 24($sp)
        sw $a0, 20($sp)
        sw $a1, 16($sp)
        sw $s0, 12($sp)
        sw $s1, 8($sp)
        addi $fp, $sp, 28

        move $s0, $a1

        jal twos_compliment_if_neg
        move $s1, $v0
        move $a0, $s0
        jal twos_compliment_if_neg
        move $a1, $v0
        move $a0, $s1
        jal mul_unsigned

        li $t1, 31
        extract_nth_bit($t2, $a0, $t1)
        extract_nth_bit($t3, $a1, $t1)
        xor $t1, $t3, $t2
        bne $t1, 1, muls_end

        move $a0, $v0
        move $a1, $v1
        jal twos_complement_64bit
        move $v0, $a1
        move $v1, $s1
muls_end:
        lw $fp, 28($sp)
        lw $ra, 24($sp)
        lw $a0, 20($sp)
        lw $a1, 16($sp)
        lw $s0, 12($sp)
        lw $s1, 8($sp)
        addi $sp, $sp, 28
        jr $ra
```

Figure 20. Implementation of mul_signed

i. *div_unsigned*

The procedure starts by initializing the remainder and the index to 0. The remainder is shifted to the left as the MSB of the dividend is assigned to the LSB of the remainder, then the dividend is shifted to the left. Then the registers are set up in order to call add_sub_logical in order to subtract the remainder and the divisor. The setup works similarly to how subtraction is handled in "au_logical" where the second argument is converted into two's compliment and then added together after calling add_sub_logical. The result from this operation is compared whether it is less than 0 or not. Should the result be less than 0, the index is merely incremented and repeats the procedure again, effectively canceling out the results of the subtraction. Should 0 or a positive number be the result, the remainder is assigned to the contents of this results and the quotient's LSB is set to 1, then the index is incremented. This will repeat until the 32nd iteration where it will then jump back to the caller.

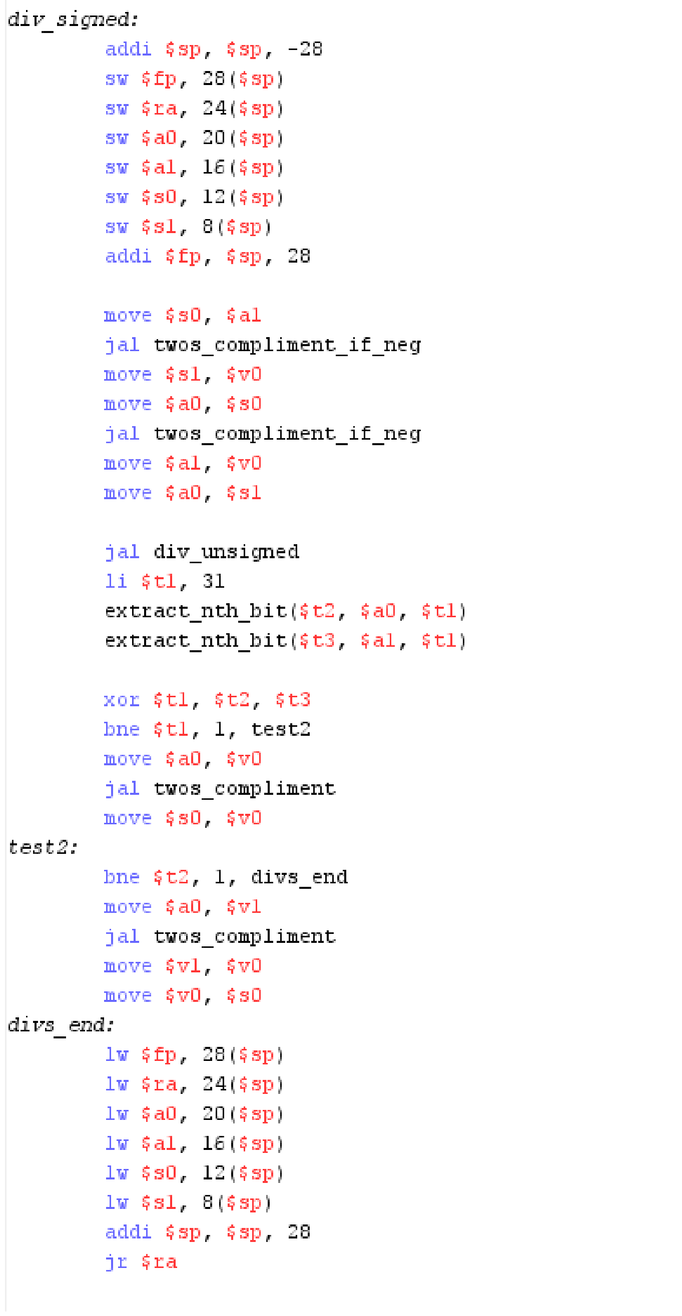


```
div_unsigned:
#$a0 = dividend, $a1 = divisor, $s3 = index, $s4 = remainder
        addi $sp, $sp, -36
        sw $fp, 36($sp)
        sw $ra, 32($sp)
        sw $a0, 28($sp)
        sw $a1, 24($sp)
        sw $s2, 20($sp)
        sw $s3, 16($sp)
        sw $s4, 12($sp)
        sw $s5, 8($sp)
        addi $fp, $sp, 36

        li $t5, 0 # index
        li $t6, 0 # R
        move $s2, $a0
        move $s3, $a1
div_loop:
        beq $t5, 32, div_end

        sll $s4, $s4, 1
        li $t1, 31
        extract_nth_bit($t1, $a0, $t1)
        insert_to_nth_bit($t6, $zero, $t1, $t9)
        sll $s2, $s2, 1

        #setup for subtraction
        move $s4, $t5
        move $s5, $t6
        move $a0, $s3
        jal twos_compliment
        move $a1, $v0
        move $a0, $s5
        jal add_sub_logical
        move $t5, $s4
        move $t6, $s5

        blt $v0, $zero, div_else
        move $t6, $v0
        li $t1, 1
        insert_to_nth_bit($s2, $zero, $t1,$t9)
div_else:
        addi $t5, $t5, 1
        j div_loop
div_end:
        move $v0, $s2
        move $v1, $t6
        sw $fp, 36($sp)
        lw $ra, 32($sp)
        lw $a0, 28($sp)
        lw $a1, 24($sp)
        lw $s2, 20($sp)
        lw $s3, 16($sp)
        lw $s4, 12($sp)
        lw $s5, 8($sp)
        addi $sp, $sp, 36
        jr $ra
```

Figure 21. Implementation of div_unsigned

i. *div_signed*

Unlike its unsigned counterpart, div_signed works very similar to mul_signed. First both arguments are tested in twos_compliment_if_neg in order to turn them into their positive forms for div_unsigned. Then the testing of the sign takes place, only for both the quotient and the remainder the be potentially converted into their two's compliment form. For the quotient, if the XOR operations for the MSB of both original arguments turns out the equal 1, twos_compliment is called to convert it. For the remainder, if the MSB of the dividend is equal to 1, then it is run through twos_compliment. then procedure then returns to the caller.

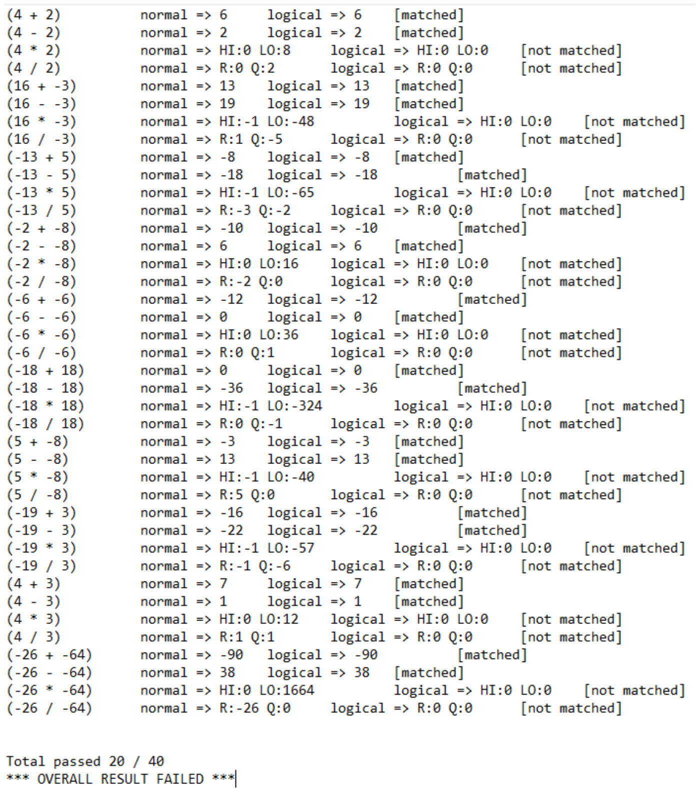


```
div_signed:
        addi $sp, $sp, -28
        sw $fp, 28($sp)
        sw $ra, 24($sp)
        sw $a0, 20($sp)
        sw $a1, 16($sp)
        sw $s0, 12($sp)
        sw $s1, 8($sp)
        addi $fp, $sp, 28

        move $s0, $a1
        jal twos_compliment_if_neg
        move $s1, $v0
        move $a0, $s0
        jal twos_compliment_if_neg
        move $a1, $v0
        move $a0, $s1

        jal div_unsigned
        li $t1, 31
        extract_nth_bit($t2, $a0, $t1)
        extract_nth_bit($t3, $a1, $t1)

        xor $t1, $t2, $t3
        bne $t1, 1, test2
        move $a0, $v0
        jal twos_compliment
        move $s0, $v0
test2:
        bne $t2, 1, divs_end
        move $a0, $v1
        jal twos_compliment
        move $v1, $v0
        move $v0, $s0
divs_end:
        lw $fp, 28($sp)
        lw $ra, 24($sp)
        lw $a0, 20($sp)
        lw $a1, 16($sp)
        lw $s0, 12($sp)
        lw $s1, 8($sp)
        addi $sp, $sp, 28
        jr $ra
```

Figure 22. Implementation of div_signed

V. Testing

In order to run the program, proj-auto-test.asm should be assembled and ran in order to test out the program. There are sets of two operands that will run through each of the four mathematical operations and print out the result. Unfortunately, only half of the results turned to pass the test. The cause of which is what is assumed to be improper use of storing and restoring the frame. This happened during first writing the add_sub_logical when running the program would lead to 0's on the side of the logical output.



```
(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:0    [not matched]
(4 / 2)         normal => R:0 Q:2       logical => R:0 Q:0      [not matched]
(16 + -3)       normal => 13    logical => 13   [matched]
(16 - -3)       normal => 19    logical => 19   [matched]
(16 * -3)       normal => HI:-1 LO:-48          logical => HI:0 LO:0    [not matched]
(16 / -3)       normal => R:1 Q:-5      logical => R:0 Q:0      [not matched]
(-13 + 5)       normal => -8    logical => -8   [matched]
(-13 - 5)       normal => -18   logical => -18          [matched]
(-13 * 5)       normal => HI:-1 LO:-65          logical => HI:0 LO:0    [not matched]
(-13 / 5)       normal => R:-3 Q:-2     logical => R:0 Q:0      [not 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:0    [not matched]
(-2 / -8)       normal => R:-2 Q:0      logical => R:0 Q:0      [not 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:0    [not matched]
(-6 / -6)       normal => R:0 Q:1       logical => R:0 Q:0      [not matched]
(-18 + 18)      normal => 0     logical => 0    [matched]
(-18 - 18)      normal => -36   logical => -36          [matched]
(-18 * 18)      normal => HI:-1 LO:-324         logical => HI:0 LO:0    [not matched]
(-18 / 18)      normal => R:0 Q:-1      logical => R:0 Q:0      [not matched]
(5 + -8)        normal => -3    logical => -3   [matched]
(5 - -8)        normal => 13    logical => 13   [matched]
(5 * -8)        normal => HI:-1 LO:-40          logical => HI:0 LO:0    [not matched]
(5 / -8)        normal => R:5 Q:0       logical => R:0 Q:0      [not matched]
(-19 + 3)       normal => -16   logical => -16          [matched]
(-19 - 3)       normal => -22   logical => -22          [matched]
(-19 * 3)       normal => HI:-1 LO:-57          logical => HI:0 LO:0    [not matched]
(-19 / 3)       normal => R:-1 Q:-6     logical => R:0 Q:0      [not 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:0    [not matched]
(4 / 3)         normal => R:1 Q:1       logical => R:0 Q:0      [not 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:0    [not matched]
(-26 / -64)     normal => R:-26 Q:0     logical => R:0 Q:0      [not matched]


Total passed 20 / 40
*** OVERALL RESULT FAILED ***
```

Figure 23. The Testing Results

VI. Conclusion

At the end of this project, despite not fulfilling all of the results during testing, there was enjoyment that was fulfilled from learning how basic math operations worked at a much lower level, down to the bits. Though the most trouble was the handling of frames, this has allowed to gain a better understanding of bit manipulation and logical operations as a whole.