Basic Calculator Implemented By MIPS Logic Operations

Parameswaran Ranganathan, Computer Scientist, San Jose State University

***Abstract*—This write up explains the implementation of basic math operations, including addition, subtraction, multiplication and division, using the MIPS logical and normal procedures in Mars 4.5. Pictures of the code are included to help understand the program.**

I. Introduction

THE purpose of logical operators is to calculate various expressions, which is the basis of digital circuits in computer hardware. This project serves to implement a simple calculator that can perform addition, subtraction, multiplication, and division through the usage of logical operators.

The calculator that is being implemented is written using the MIPS. MIPS is a type of assembly language and the MARS software (IDE) is what is used to simulate the MIPS.

II. Requirements

Section II discusses the necessary software needed to write the basic mathematical calculator and also provides the needed background information to create the calculator and understand how it works.

A. Necessary Software's

To run MIPS assembly language, one must use the MARS software as their interactive development environment (IDE). Mars is a simulator, which acts as a runtime environment for MIPS. MARS can be downloaded on Missouri State University website as it is developed by them.

B. Setting up the project

Go to SJSU Canvas and download the provided zip files.

[https://sjsu.instructure.com/courses/1208160/assignments/4252547 - submit](https://sjsu.instructure.com/courses/1208160/assignments/4252547#submit)

Download the "CS47project1.zip" and unzip it. It should include the following files.

1. *Cs47_common_macro.asm*
2. *Cs47_proj_alu_logical.asm*
3. *Cs47_proj_alu_normal.asm*
4. *Cs47_proj_macro.asm*
5. *Cs47_proj_procs.asm*
6. *Proj-auto-test.asm*

Open the Mars 4_5 jar file downloaded from the Missouri state university webpage. Go to "File" and click on "open".

Find your way to the directory, which contains the unzipped files. Since MARS does not allow you to load all the files at once, load each of them separately. After all the files have been loaded and opened, MARS should look like this.

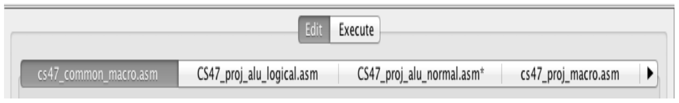


Fig.1. Opening/loading files onto MARS

C. Boolean Algebra/logic

Implemented circuitry in computer hardware systems requires a deep understanding of Boolean logic and algebra. In circuits, only the numbers 1 and 0 exists, nothing else. One must use truth tables to find out the output of a Boolean expression.

Here is an example of the AND Boolean operation where t represents a T and 0 represents a F. AND is the multiplication of two values.

Table 1: Truth Table for Logical AND

A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

In addition to the AND operation, there exists an OR operation. OR is used for addition of two binary integers in Boolean algebra. The OR operation returns 0 if both A and B are 0. The following is the truth table of Logical OR.

Table 2: Truth Table for Logical OR

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

Finally, there exists an XOR operation, which means exclusive OR. XOR returns 0 if both A and B are either 0 or 1. The XOR operation returns 1 if only one of the two is 1.

Table 3: Truth Table for Logical XOR

A	B	A.B
0	0	1
0	1	1
1	0	1
1	1	0

D. *Binary*

Binary is a number system with base 2, unlike the decimal system, which uses base 10. Since this system is base 2, it consists of only 2 symbols that can be used, 0 or 1. For example, if we used a 4 bit long word, 0 is represented as 0000 and 1 is represented as 0001. Once a digit reaches 1, it becomes 0 and rolls over to the next larger digit and places a 1 there. Thus, 2 would be represented as 0010. The binary system is extremely useful when it comes to computers as a computer system can only read 1's and 0's.

Now the question is, how does one represent a negative number in binary? This is where the "two's compliment" form comes into play. In two's compliment, if a 1 exists in the MSB position, that means the number is negative. If a 0 exists in the MSB position, then the number is positive. For example, 3 in twos compliment is 0011, where as -3 is 1101. To obtain the negative of a number, inverse all the bits of the positive number and add 1. If we inverse all the bits of +3, we get 1100. Add 1 to this, and we obtain 1101, which is -3.

III. Design and Implementation

Section 3 discusses the design of the arithmetic calculator and how it's being implemented in MIPS. This calculator implements addition, subtraction, multiplication, and division operations.

A. *Design*

The design section talks about the design of the addition, subtraction, multiplication, and division operations.

1) *Addition and subtraction*

Addition and subtraction both relate in the sense that subtraction is the addition of the negation of the second integer. For example, A-B = A + (-B). For the addition operation, a full adder must be implemented since the carry bit has to be taken into consideration. A half adder can only add two bits.

The way addition works is a carry bit starts off as 0 and is added with the LSB bits of the two inputs. The sum is put into the LSB position of the answer, and the carry bit is found and added to the next two bits of the inputs. To get the nth digit, the logical XOR is called on the first bits and is stored. Then, the logical XOR is called on the stored value and the carry in. This results in the sum bit and is stored temporarily. To find the carry bit, first initialize the carry bit to 0. To find the carry bit, first we use the XOR operation on the two inputs and store it in lets say \$t7. AND is called on \$t7 and the initial carry bit (\$v1) and is stored in \$t8. Then, AND is used on the nth bits of the two inputs. Finally, OR is called on \$t8 and \$t1 and is stored back in the carry bit register which is \$v1.

These operations are performed 32 times, since there are 32 bits in each register. Finally, the sum of the two inputs will be calculated.

For subtraction, the only change that must be made is the complement of the second input must be taken. Then pass the first input and the complement of the second input into the add loop and the difference (in this case a sum) will be calculated.

2) *Multiplication*

For unsigned multiplication, a counter (\$t0) is initially set to 0. The first input, which is the multiplicand, is saved in a temporary register (\$t4). The second input, which is the multiplier, is put into a temporary register (\$t1). This represents the lo of the multiplication. The high is initially set to 0 and put into temporary register \$t2.

In short, binary multiplication is simply repeated addition. What we must know is that 0 x 0 = 0, 1 x 0 = 0, and 1 x 1 = 1. If the current bit of the multiplier is 0, then 0 gets placed in that bit position of the answer. If the multiplier is 1, then simply put the multiplicand into that bit position of the answer.

3) *Division*

First, we must align the divisor with the MSB bits of the dividend. Compare these bits to the divisor. If these bits are greater than the divisor, the quotient bit is set to 1 and then subtraction is performed with MSB bits – divisor. If the MSB bits are less than the divisor, the quotient bit is set to 0 and subtraction is not performed. The divisor is then shifted one bit to the right and we compare the MSB bits to the divisor again. This continues until division is over.

B. *Implementation*

To implement the calculator operations, many macros and utility procedures are designed to help make the implementation easier.

1) *Utility Macros*

The 4 macros created are extract_nth_bit, insert_to_nth_bit, store_stack_all, and restore_stack_all.

i. *Extract_nth_bit*

The extract_nth_bit macro obtains a bit value at a given position for any integer.

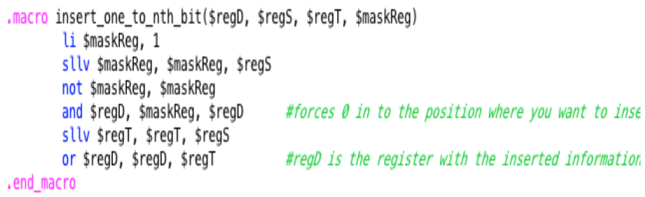
```
.macro extract_nth_bit($regD, $regS, $regT)
        la $s0, ($regS)
        srav $s0, $s0, $regT
        andi $regD, $s0, 1
.end_macro
```

The macro contains three arguments passed in as regD, regS, regT. regD is the destination register where the result will be stored. regS is the source register which contains the bit pattern which the calculator will be extracting from. regT holds the position of the bit that will be extracted.

First, \$regS is moved into \$s0 just so that the original source register does not get messed up. Then, \$s0 is shifted right by the bit position number and is stored back into \$s0. Finally, AND is used on \$s0 and 1 to extract the bit and is put into the destination register. Basically, if \$s0 is 0x0, AND'ing it with 1 will result in 0. If \$s0, is 0x1, AND'ing it with 1 will result in 1. This properly extracts the needed bit.

ii. *Insert_to_nth_bit*

The insert_to_nth_bit macro inserts a given bit into the nth position of a bit pattern and returns the bit pattern.

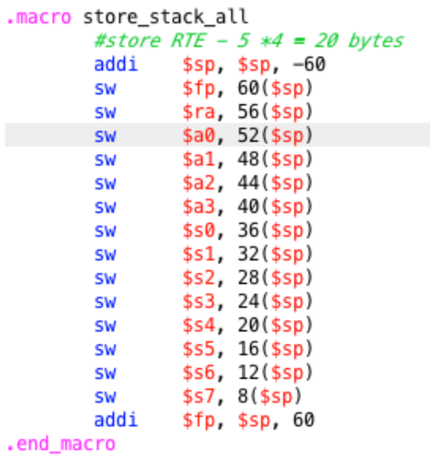

```
.macro insert_one_to_nth_bit($regD, $regS, $regT, $maskReg)
        li $maskReg, 1
        sllv $maskReg, $maskReg, $regS
        not $maskReg, $maskReg
        and $regD, $maskReg, $regD      #forces 0 in to the position where you want to inse
        sllv $regT, $regT, $regS
        or $regD, $regD, $regT          #regD is the register with the inserted informatio
.end_macro
```

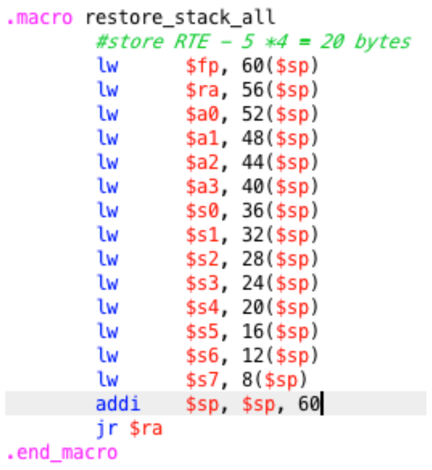
The macro contains four arguments passed in as regD, regS, regT, and maskReg. regD contains the bit pattern in which the proper bit will be inserted into. regS contains the insertion position. regT contains what bit will be inserted, either a 1 or 0. Finally, maskReg is any temporary register in which a mask will be created.

Initially the mask register is set to 1 and is then shifted left by the regS amount, which is the insertion position. All of the bits in the mask register are then inverted by using the NOT operation. Next, AND is used on the mask register and the bit pattern in which we are inserting to force 0 into the position in which we are inserting. regT, which contains the bit to be inserted is then shifted left by the regS amount to get into the right position. Finally, or is used on regD and regT to finish the insertion process.

iii. Store_stack_all and restore_stack_all

These two macros serve to store and restore registers $fp, $ra, $a0-$a3 and $s0-$s7. Unlike the other macros, these macros don't take in any arguments.


```
.macro store_stack_all
        #store RTE - 5 *4 = 20 bytes
        addi    $sp, $sp, -60
        sw      $fp, 60($sp)
        sw      $ra, 56($sp)
        sw      $a0, 52($sp)
        sw      $a1, 48($sp)
        sw      $a2, 44($sp)
        sw      $a3, 40($sp)
        sw      $s0, 36($sp)
        sw      $s1, 32($sp)
        sw      $s2, 28($sp)
        sw      $s3, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        addi    $fp, $sp, 60
.end_macro
```


```
.macro restore_stack_all
        #store RTE - 5 *4 = 20 bytes
        lw      $fp, 60($sp)
        lw      $ra, 56($sp)
        lw      $a0, 52($sp)
        lw      $a1, 48($sp)
        lw      $a2, 44($sp)
        lw      $a3, 40($sp)
        lw      $s0, 36($sp)
        lw      $s1, 32($sp)
        lw      $s2, 28($sp)
        lw      $s3, 24($sp)
        lw      $s4, 20($sp)
        lw      $s5, 16($sp)
        lw      $s6, 12($sp)
        lw      $s7, 8($sp)
        addi    $sp, $sp, 60
        jr $ra
.end_macro
```

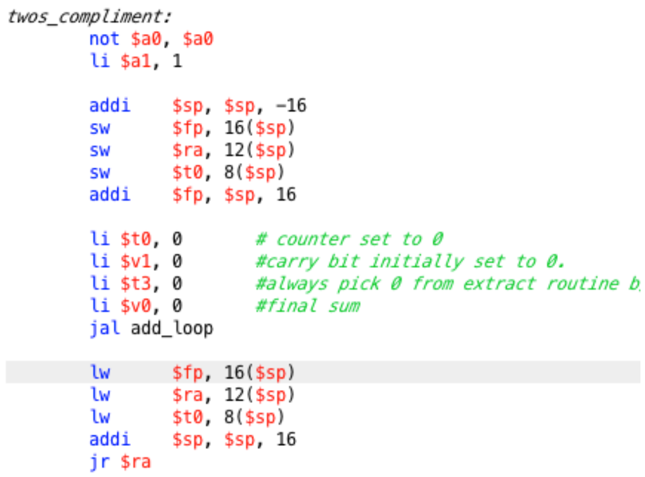
Although it is not necessary to always store and restore these registers, these macros still do it regardless.

2) Utility Procedures

i. twos_complement

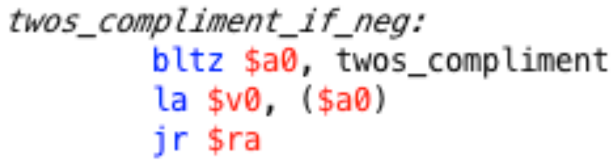
An argument $a0 is passed into the twos_complement subroutine and the complement of the argument is returned in the $v0 register.

In the twos_complement subroutine, the NOT of $a0 is taken and stored back in $a0. The number 1 is then put into $a1 and both $a0 and $a1 are passed into the add_loop method. This way, the complement of $a0 is taken.


```
twos_compliment:
        not $a0, $a0
        li $a1, 1

        addi    $sp, $sp, -16
        sw      $fp, 16($sp)
        sw      $ra, 12($sp)
        sw      $t0, 8($sp)
        addi    $fp, $sp, 16

        li $t0, 0       # counter set to 0
        li $v1, 0       #carry bit initially set to 0.
        li $t3, 0       #always pick 0 from extract routine b
        li $v0, 0       #final sum
        jal add_loop

        lw      $fp, 16($sp)
        lw      $ra, 12($sp)
        lw      $t0, 8($sp)
        addi    $sp, $sp, 16
        jr $ra
```

ii. Twos_complement_if_neg

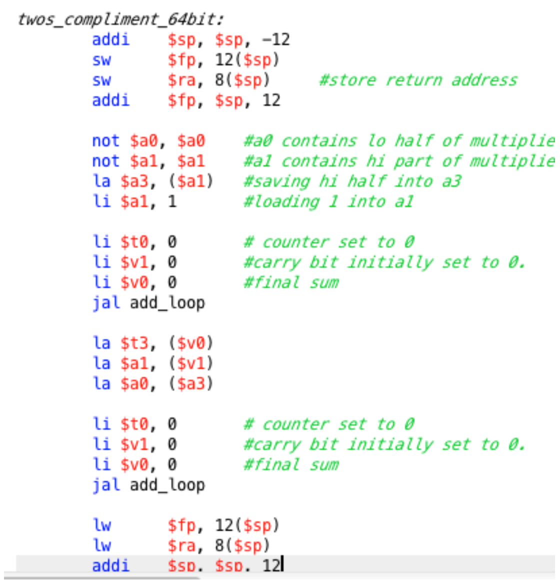
Twos_complement_if_neg branches to twos_complement if the argument is less than 0, or returns the argument itself if it is above 0.


```
twos_compliment_if_neg:
        bltz $a0, twos_compliment
        la $v0, ($a0)
        jr $ra
```

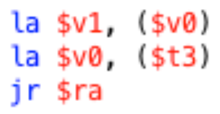
iii. *Twos_complement_64bit*

Twos_complement_64bit returns the complement of the lo and hi of the multiplied result. The lo exists in the $a0 register and the hi exists in the $a1 register.

$a0 and $a0 are inverted using not. $a1 is then saved in $a3 and 1 is loaded into $a1. Add_loop is then called on $a0 and $a1, which is the lo and 1 respectively. Once add_loop is done, twos_complement_64bit is returned to and the answer ($v0) is stored in $t0. The carry bit ($v1) is saved in $a1 and $a3 is moved back into $a0. Add_loop is then called again with the new $a0 and $a1. After this, the twos_complement_64bit is done.



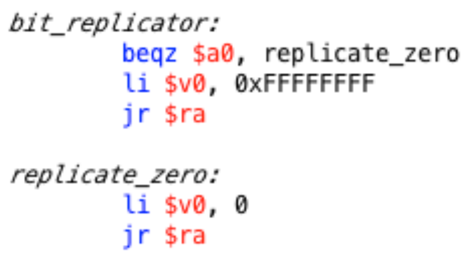
```
twos_compliment_64bit:
        addi    $sp, $sp, -12
        sw      $fp, 12($sp)
        sw      $ra, 8($sp)     #store return address
        addi    $fp, $sp, 12

        not $a0, $a0        #a0 contains lo half of multiplie
        not $a1, $a1        #a1 contains hi part of multiplie
        la $a3, ($a1)       #saving hi half into a3
        li $a1, 1           #loading 1 into a1

        li $t0, 0           # counter set to 0
        li $v1, 0           #carry bit initially set to 0.
        li $v0, 0           #final sum
        jal add_loop

        la $t3, ($v0)
        la $a1, ($v1)
        la $a0, ($a3)

        li $t0, 0           # counter set to 0
        li $v1, 0           #carry bit initially set to 0.
        li $v0, 0           #final sum
        jal add_loop

        lw      $fp, 12($sp)
        lw      $ra, 8($sp)
        addi    $sp, $sp, 12
```



```
la $v1, ($v0)
la $v0, ($t3)
jr $ra
```

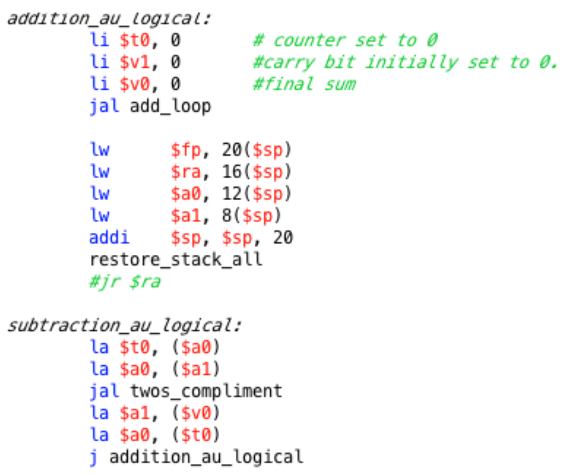
iv. *bit_replicator and replicate*

These two procedures replicate bits of a given register. Bit_replicator checks if the argument is 0, and if it is, branches to replicate_zero. Replicate_zero will load 0 into $v0 and returns back to the caller address. If bit_replicator doesn't branch, -1 is loaded into $v0 and returns back to the caller address.

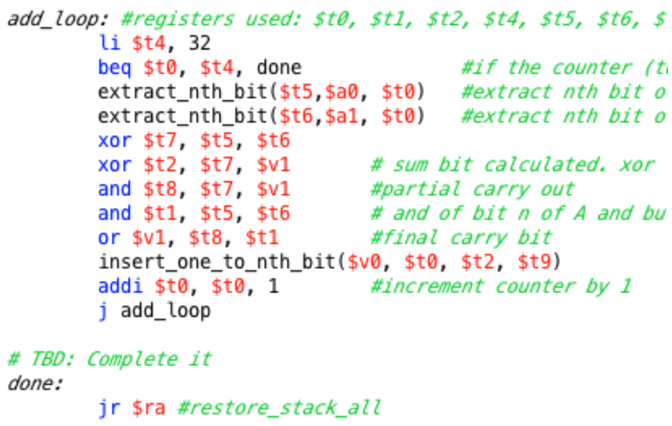


```
bit_replicator:
        beqz $a0, replicate_zero
        li $v0, 0xFFFFFFFF
        jr $ra

replicate_zero:
        li $v0, 0
        jr $ra
```

3) Addition/subtraction implementation



```
addition_au_logical:
        li $t0, 0           # counter set to 0
        li $v1, 0           #carry bit initially set to 0.
        li $v0, 0           #final sum
        jal add_loop

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

subtraction_au_logical:
        la $t0, ($a0)
        la $a0, ($a1)
        jal twos_compliment
        la $a1, ($v0)
        la $a0, ($t0)
        j addition_au_logical
```



```
add_loop: #registers used: $t0, $t1, $t2, $t4, $t5, $t6, $
        li $t4, 32
        beq $t0, $t4, done                      #if the counter (t
        extract_nth_bit($t5,$a0, $t0)   #extract nth bit o
        extract_nth_bit($t6,$a1, $t0)   #extract nth bit o
        xor $t7, $t5, $t6
        xor $t2, $t7, $v1           # sum bit calculated. xor
        and $t8, $t7, $v1           #partial carry out
        and $t1, $t5, $t6           # and of bit n of A and bu
        or $v1, $t8, $t1            #final carry bit
        insert_one_to_nth_bit($v0, $t0, $t2, $t9)
        addi $t0, $t0, 1            #increment counter by 1
        j add_loop

# TBD: Complete it
done:
        jr $ra #restore_stack_all
```

Addition_au_logical sets up the addition process. It creates the necessary variables and jumps to the add_loop. For subtraction_au_logical, the second input is moved into a0, and the twos complement is taken. Then all the variables are stored back in their correct position and the method jumps to addition_au_logical.

Add_loop is where the addition actually takes place. This loop calculates the sum of the $a0 and $a1 registers. The method first sets a register to the immediate value of 32 so that the loop has a value to compare to. The loop checks if the counter is equal to 32. If it is, it branches to "done" and returns to the caller. If it does not equal 32, the loop is executed. The nth bits of the two inputs are extracted and stored in $t5 and $t6 respectively. XOR is used to calculated the sum bit and AND/OR are used to calculate the final carry out bit. The sum bit is then inserted into the correct position of the final answer ($v0) and the counter is then incremented by 1. The procedure then jumps back to itself.

3) *Multiplication implementation*

```
multiplication_au_logical:
        la $a3, ($a1)    #saving a1 into a3
        jal twos_compliment_if_neg
        la $t0, ($v0)    #2's compliment of a0 (mcnd)
        la $a0, ($a3)
        jal twos_compliment_if_neg      #get 2's compl
        la $a1, ($v0)
        la $a0, ($t0)
        j mul_unsigned

mul_unsigned:
#a0 = mcnd, a1 = mplr
        li $t0, 0        #counter
        la $t1, ($a1)    #lo (l)
        li $t2, 0        #hi (h)
        la $t4, ($a0)    #save mcnd


extract_beginning:
        extract_nth_bit($a0, $t1, $zero)        #a0 gets the 0th
        jal bit_replicator      #v0 gets the replicated bit, whic
        and $t5, $t4, $v0
        la $a0, ($t2)           #move old h value into a0
        la $a1, ($t5)   #move the 0 or mcnd (t5) into a1

        #now add old H with mcnd or 0, depending on what the 0th
        addi    $sp, $sp, -44
        sw      $fp, 44($sp)
        sw      $ra, 40($sp)
        sw      $t0, 36($sp)
        sw      $t1, 32($sp)
        sw      $t2, 28($sp)
        sw      $t3, 24($sp)
        sw      $t4, 20($sp)
        sw      $t5, 16($sp)
        sw      $a0, 12($sp)
        sw      $a1, 8($sp)
        addi    $fp, $sp, 44
        li $t0, 0       # counter set to 0
        li $v1, 0       #carry bit initially set to 0.
        li $v0, 0       #final sum

        jal add_loop


       lw      $fp, 44($sp)
       lw      $ra, 40($sp)
       lw      $t0, 36($sp)
       lw      $t1, 32($sp)
       lw      $t2, 28($sp)
       lw      $t3, 24($sp)
       lw      $t4, 20($sp)
       lw      $t5, 16($sp)
       lw      $a0, 12($sp)
       lw      $a1, 8($sp)
       addi    $sp, $sp, 44

       la $t2, ($v0)    #h = h + x
       #next part is shifting 64 bits to the right
       srl  $t1, $t1, 1          #shift mplr right by 1 bit
       extract_nth_bit($t7, $t2, $zero)         #extracting bit i
       li $t8, 31
       insert_one_to_nth_bit($t1, $t8, $t7, $t9) #move bit 0 of
       srl  $t2, $t2, 1                 #shift hi right by 1 bit
       addi $t0, $t0, 1                 #increment loop counter i
       li $t8, 32
       beq $t0, $t8, done_mult          #quit of counter = 32
       j extract_beginning
```

```
done_mult:
        la $v0, ($t1)   #this moves the lo, 32 bit result, into v0
        la $v1, ($t2)   #this moves the hi, 32 bit result, into v1

        lw      $fp, 20($sp)
        lw      $ra, 16($sp)
        lw      $a0, 12($sp)
        lw      $a1, 8($sp)
        addi    $sp, $sp, 20
        #restore_stack_all
        li $t8, 31
        extract_nth_bit($t1, $a0, $t8)  #extract bit 31 of a0
        extract_nth_bit($t2, $a1, $t8)  #extract bit 31 of a1
        xor $t6, $t1, $t2       #if xor is 1, that means only one number is
        beqz $t6, positive
        la $a0, ($v0)
        la $a1, ($v1)
        jal twos_compliment_64bit
        restore_stack_all
        #jr $ra

positive:
        restore_stack_all       #restore will take u to caller
        #jr $ra
```

i. *Multiplication_au_logical*

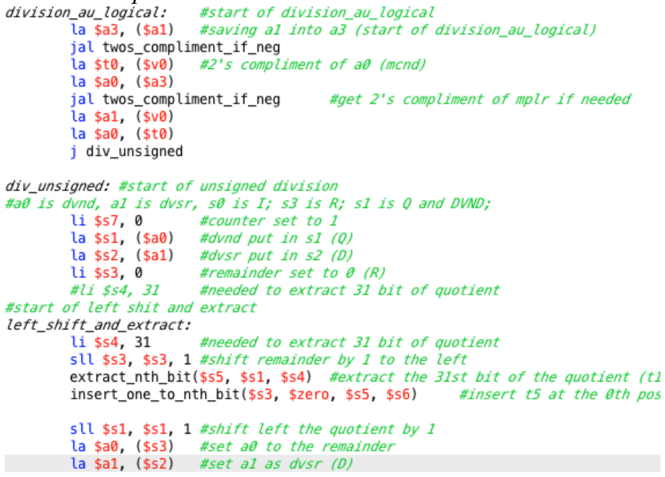
Multiplication_au_logical sets up the entire multiplication process. It first checks to see if both number are negative by using twos_complement_if_neg, then jumps to mul_unsigned.

Mul_unsigned gets all the necessary registers needed for storage and continues on to extract_beginning. Extract_beginning first gets the 0th bit of the lo and stores it in $a0. $a0 is then replicated using bit_replicator and is stored in $v0. AND is then called on the replicated bit pattern and $t4, which holds $a0 (MCND). Now we want to add the original Hi with the mcnd or 0, depending on what the 0th bit was, and the value is stored back into the register, which holds the Hi. Next, the Lo is shifted right by 1 bit and the 31st bit of Lo gets the 0th bit of the Hi. Then, the Hi is shifted right by 1 bit and the counter is incremented by 1. Next, we check if the counter equals 32, and if it does, that means all the multiplicating is over and we jump to “done_mult”. If it does not equal 32, we jump back to the “extract_beginning”.

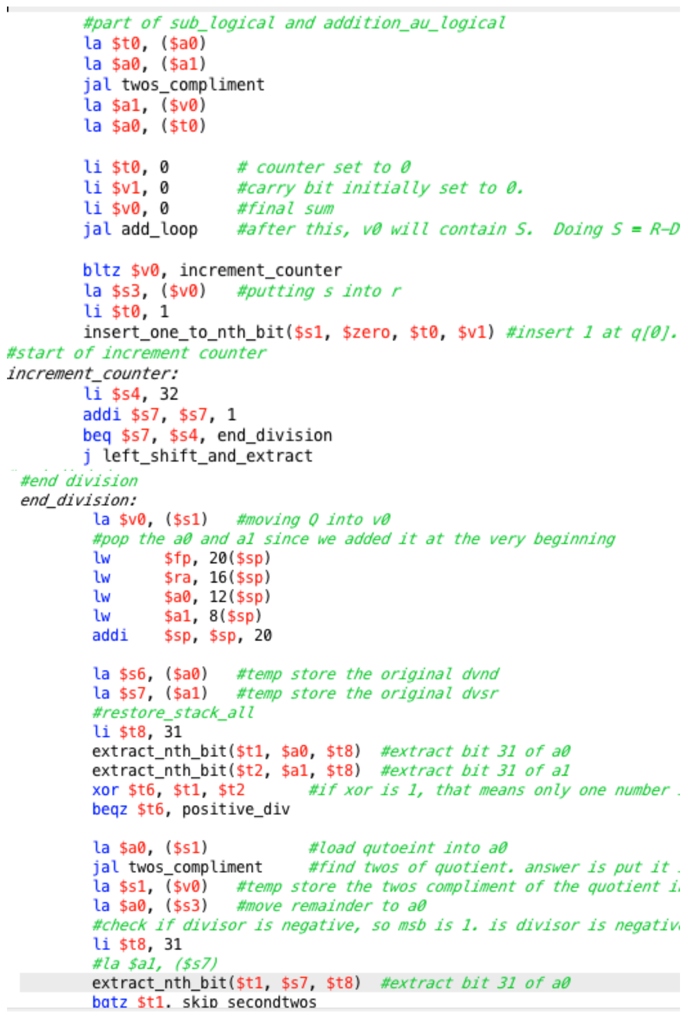
ii. *done_mult*

Done_mult begins with storing the lo ($t1) into $v0 and storing the hi ($t2) into $v1. Next, the code finds out if one of the initial inputs were negative or if both are either positive or negative. We check to see if both are negative or if both are positive by calling an XOR on the 31st bit of each number. If the XOR is equal to 1, that means only one of the inputs is negative, meaning that the sign bit should be negative. If the XOR is 0, we branch to “positive” which restores the stack and jumps back to the caller. If the code does not jump to positive, twos complement 64 bit is called on hi and lo and then the stack is restored and returned back to the caller.

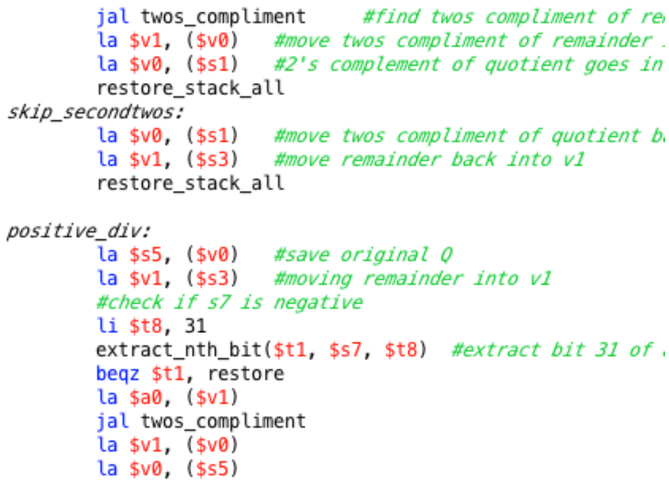
4) *Division implementation*



```
division_au_logical:    #start of division_au_logical
        la $a3, ($a1)   #saving a1 into a3 (start of division_au_logical)
        jal twos_compliment_if_neg
        la $t0, ($v0)   #2's compliment of a0 (mcnd)
        la $a0, ($a3)
        jal twos_compliment_if_neg      #get 2's compliment of mplr if needed
        la $a1, ($v0)
        la $a0, ($t0)
        j div_unsigned

div_unsigned: #start of unsigned division
#a0 is dvnd, a1 is dvsr, s0 is I; s3 is R; s1 is Q and DVND;
        li $s7, 0       #counter set to 1
        la $s1, ($a0)   #dvnd put in s1 (Q)
        la $s2, ($a1)   #dvsr put in s2 (D)
        li $s3, 0       #remainder set to 0 (R)
        #li $s4, 31     #needed to extract 31 bit of quotient
#start of left shit and extract
left_shift_and_extract:
        li $s4, 31      #needed to extract 31 bit of quotient
        sll $s3, $s3, 1 #shift remainder by 1 to the left
        extract_nth_bit($s5, $s1, $s4)  #extract the 31st bit of the quotient (t1
        insert_one_to_nth_bit($s3, $zero, $s5, $s6)     #insert t5 at the 0th pos

        sll $s1, $s1, 1 #shift left the quotient by 1
        la $a0, ($s3)   #set a0 to the remainder
        la $a1, ($s2)   #set a1 as dvsr (D)
```



```
        #part of sub_logical and addition_au_logical
        la $t0, ($a0)
        la $a0, ($a1)
        jal twos_compliment
        la $a1, ($v0)
        la $a0, ($t0)

        li $t0, 0       # counter set to 0
        li $v1, 0       #carry bit initially set to 0.
        li $v0, 0       #final sum
        jal add_loop    #after this, v0 will contain S.  Doing S = R-D

        bltz $v0, increment_counter
        la $s3, ($v0)   #putting s into r
        li $t0, 1
        insert_one_to_nth_bit($s1, $zero, $t0, $v1) #insert 1 at q[0].
#start of increment counter
increment_counter:
        li $s4, 32
        addi $s7, $s7, 1
        beq $s7, $s4, end_division
        j left_shift_and_extract
 #end division
 end_division:
        la $v0, ($s1)   #moving Q into v0
        #pop the a0 and a1 since we added it at the very beginning
        lw      $fp, 20($sp)
        lw      $ra, 16($sp)
        lw      $a0, 12($sp)
        lw      $a1, 8($sp)
        addi    $sp, $sp, 20

        la $s6, ($a0)   #temp store the original dvnd
        la $s7, ($a1)   #temp store the original dvsr
        #restore_stack_all
        li $t8, 31
        extract_nth_bit($t1, $a0, $t8)  #extract bit 31 of a0
        extract_nth_bit($t2, $a1, $t8)  #extract bit 31 of a1
        xor $t6, $t1, $t2       #if xor is 1, that means only one number
        beqz $t6, positive_div

        la $a0, ($s1)           #load qutoeint into a0
        jal twos_compliment     #find twos of quotient. answer is put it
        la $s1, ($v0)   #temp store the twos compliment of the quotient i
        la $a0, ($s3)   #move remainder to a0
        #check if divisor is negative, so msb is 1. is divisor is negativ
        li $t8, 31
        #la $a1, ($s7)
        extract_nth_bit($t1, $s7, $t8)  #extract bit 31 of a0
        bgtz $t1, skip_secondtwos
```



```
        jal twos_compliment     #find twos compliment of re
        la $v1, ($v0)   #move twos compliment of remainder
        la $v0, ($s1)   #2's complement of quotient goes in
        restore_stack_all
skip_secondtwos:
        la $v0, ($s1)   #move twos compliment of quotient b
        la $v1, ($s3)   #move remainder back into v1
        restore_stack_all

positive_div:
        la $s5, ($v0)   #save original Q
        la $v1, ($s3)   #moving remainder into v1
        #check if s7 is negative
        li $t8, 31
        extract_nth_bit($t1, $s7, $t8)  #extract bit 31 of
        beqz $t1, restore
        la $a0, ($v1)
        jal twos_compliment
        la $v1, ($v0)
        la $v0, ($s5)
```

i) *Division_au_logical*

Division_au_logical obtains the twos complement of the two inputs if they are below 0. Then the method jumps to div_unsigned.

ii) div_unsigned

Div_unsigned begins by setting up the proper arguments as variables. A counter, $s7, is initialized to 0. $s0, which holds the Quotient, is set to $s1. $a1, which holds the dvsr, is set to $s2. And finally $s3, which is the remainder, is set to 0.

iii) *left_shift_and_extract*

In left_shift_and_extract, the remainder is shifted left by 1 bit. Then the 31st bit of the quotient ($s1) is put into the 0th bit position of the remainder ($s3). The quotient is then shifted left by one bit. Next, an intermediate value (S) is calculated by subtracting the dvsr from the remainder. I was not able to call the actual sub_logical routine since that will always jump back to the caller code, so I copy and pasted by sub_logical code into this method. The difference is taken and is stored into S ($v0). Next, we check if S is 0, basically if the Dvsr is greater than the remainder. If it is less than 0, we jump to increment counter. If S is not greater than 0 , we set R to equal S and make the 0th bit of Q a 1. Then we jump to increment counter and do the same checking.

iv) *increment_counter*

Increment_counter increments the counter by 1 and compares it to 32. If the counter equals 32, we jump to end_division, or else we go back to left_shift_and_extract.

v) *end_division*

End_division takes care of adding the proper sign to the quotient and remainder. The 31st bits of the original inputs are extracted and XOR is called on both of them. If the XOR is 1, that means only one number is negative, meaning that the sign bit should be negative. If the XOR'd answer is equal to 0, we branch to positive_div. If it equals 1, we figure out the correct signs to put on the remainder and the quotient by using various twos_complements calls.

vi) positive_div

Positive_div restores the stack and returns to the caller.

IV. Testing

Another class, called “proj_alu_normal” is created. This is also an arithmetic calculator, but does not use logic to perform the operations. Instead, it uses MIPS inbuilt arithmetic operations such as “add”, “sub”, “mul”, and “div”.

A) Implementation

1) addition_au_normal

Addition_au_normal calls the MIPS add instruction to add the two inputs. Just like in au_logical, the arguments are passed through the method in $a0 and $a1, and the result is stored in $v0.

2) subtraction_au_normal

Subtraction_au_normal calls the MIPS subtraction instruction to subtract the two inputs. Similarly to addition_au_normal, the arguments are passed through the method in $a0 and $a1, and the result is stored in $v0.

3) Multiplication_au_normal

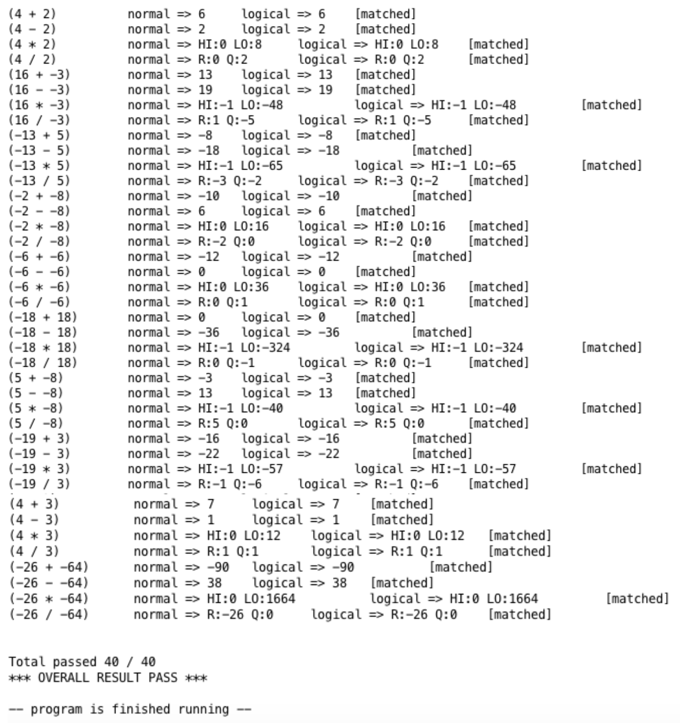
Multiplication_au_normal calls the MIPS “mul” instruction and stores the result in $v0. The hi value is then moved into the $v1 register.

4) Division_au_normal

Division_au_normal calls the MIPS “div” instruction on $a0 and $a1, which are the two inputs. The hi value contains the remainder and the lo value contains the quotient. The remainder is moved into $v1 and the quotient is moved into $v0.

B) Proj-auto-test

Professor Patra wrote assembly code which tests that the calculator works properly. The code provides sample inputs and does the operations using the au_normal and au_logical. The file see’s if the au_normal results match the au_logical results. The following is a picture of the proj_auto_test 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: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 --
```

V. Conclusion

This project, although a tough and tedious one, was made to implement a calculator using logical operations only. It was basically writing software that simulates the hardware, which performs these logical operations. The program was written in the MIPS assembly language and tested using a tester file that Professor Patra provided.