Arithmetic Implemented by MIPS Logical Operations

Shreyass Prem Sankar
San Jose State University
shreyass.premsankar@sjsu.edu

***Abstract*—This report outlines the format and implementation of mathematical operations such as addition, subtraction, multiplication, and division. In the report/project, we use MARS IDE which simulates MIPS procedures and use both normal and logical procedures.**

I. Introduction (*Heading 1*)

With machine code/assembly language, we are able to combine various normal and logical operations in MIPS/MARS to simulate basic mathematical operations such as addition, subtraction, multiplication, and division.

In this project, we aim to simulate a barebones calculator with basic operations by using our MIPS logic operations in machine code. Stated below are the three main objectives/steps of this project:

1. Install and setup required software (MARS)
2. Simulate basic mathematical operations (addition, subtraction, multiplication, division) with MIPS operations
3. Test our coded mathematical operations in MARS

The project report will include instructions on how to setup MARS and explain how to implement the machine code behind each mathematical operation. Finally, the report overviews the test cases provided to ensure that the machine code is void of any errors or bugs and that the output is correct.

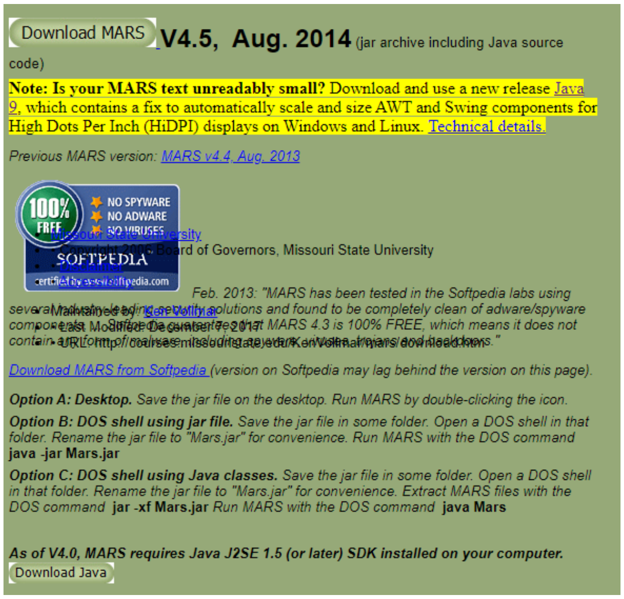
II. MARS Installation/setup

A. *Installing MARS*

Click the link below to go to the webpage install MARS:

https://courses.missouristate.edu/KenVollmar/mars/download.htm. Click the "Download MARS" button and run MARS4_5.jar using Java once the file has finished downloading.

Also, click the Download Java button as seen at the bottom of the following image if needed. Run the Java file before the MARS file.



V4.5, Aug. 2014 (jar archive including Java source code)

Note: Is your MARS text unreadably small? Download and use a new release Java 9, which contains a fix to automatically scale and size AWT and Swing components for High Dots Per Inch (HiDPI) displays on Windows and Linux. Technical details.

Previous MARS version: MARS v4.4, Aug. 2013

- Missouri State University
- Copyright 2006 Board of Governors, Missouri State University
- Disclaimer
- Accessibility
- Maintained by: Ken Vollmar
- Last Modified: December 7, 2017
- URL: http://courses.missouristate.edu/KenVollmar/mars/download.htm

Feb. 2013: "MARS has been tested in the Softpedia labs using several industry-leading security solutions and found to be completely clean of adware/spyware components. Softpedia guarantees that MARS 4.3 is 100% FREE, which means it does not contain any form of malware, including adware, viruses, trojans and backdoors."

Download MARS from Softpedia (version on Softpedia may lag behind the version on this page).

***Option A: Desktop.** Save the jar file on the desktop. Run MARS by double-clicking the icon.*

***Option B: DOS shell using jar file.** Save the jar file in some folder. Open a DOS shell in that folder. Rename the jar file to "Mars.jar" for convenience. Run MARS with the DOS command* **java -jar Mars.jar**

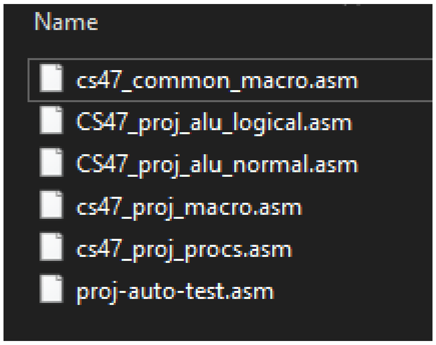
***Option C: DOS shell using Java classes.** Save the jar file in some folder. Open a DOS shell in that folder. Rename the jar file to "Mars.jar" for convenience. Extract MARS files with the DOS command* **jar -xf Mars.jar** *Run MARS with the DOS command* **java Mars**

As of V4.0, MARS requires Java J2SE 1.5 (or later) SDK installed on your computer.

B. *Open Project in MARS*

Download CS47Projectl.zip from the link below: https://sjsu.instructure.com/courses/1474044/files/66532978/download?wrap=1

After downloading the file, extract all of the files and you should see the following list of files:



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

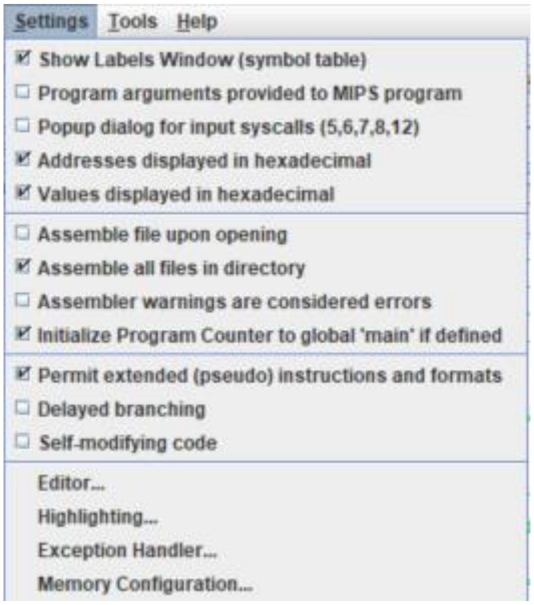
1. cs47_common_macro.asm – has common macros needed for the program, do not modify
2. CS47_proj_alu_logical.asm – implement logical operations for the arithmetic calculator in this file

3. CS47_proj_alu_normal.asm – the generic implementation of the calculator is done here, able to use add, sub, etc. for your operations in order to compare them and check the outputs of CS47_proj_alu_logical
4. cs47_proj_macro.asm – any macros needed for the program, do not modify
5. cs47_proj_procs.asm – contains the project macros needed for the program, do not modify
6. proj-auto-test.asm – the testing program used to compare the outputs of CS47_proj_alu_logical.asm and the CS47_proj_alu_normal.asm to see if they match

Open these files in MARS by clicking file in the upper right corner of the MARS application and selecting open from the drop-down menu. After that, a pop-up window will open to navigate to the files and open them on-by-one.

Make sure to have these default settings checked in MARS(under settings drop-down menu):

1. Assemble all files in directory
2. Initialize program counter to global main if defined



☑ 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

III. Operation implementation

For this project, there are two main program files, one(CS47_proj_alu_normal) which is used to check the validity of the results of your implementation of the arithmetic operations with MIPS normal operations. Additionally, we have the bulk of the code, also known as our implementation of the arithmetic calculations: CS47_proj_alu_logical which uses MIPS logical operations.

A. *Procedures*

1. Addition

In MIPS logical addition, we find the sum of two numbers by using a full adder which is able to determine our carry bit and sum bit. Now the number of times this procedure is done depends on the number of bits, in this case, 32. Once the code has run 32 times, the machine is able to compute the sum of two values, which is in total, 32 bits.

2. Subtraction

In MIPS logical subtraction, we find the sum of two numbers by using a full adder which is able to determine our carry bit and sum bit. Now the number of times this procedure is done depends on the number of bits, in this case, 32. Once the code has run 32 times, the machine is able to compute the sum of two values, which is in total, 32 bits. Very similar to addition, but for subtraction, we negate the second value before adding to the first.

3. Multiplication

In MIPS logical multiplication, we find the product of two numbers: the multiplicand and the multiplier. Unlike addition or subtraction, the product in multiplication is 64 bits long, which requires us to use Hi and Lo to split the upper and lower domains of the final value.

We start of with the least significant bit or LSB of the multiplier and multiply with the multiplicand, which does not give us the whole answer, but just a partial product. We then proceed to shift the multiplier one to the right and the multiplicand one to the left. Then, we multiply the modified values to receive yet another partial product. Finally, we add these two partial products to gain a partial sum, representative of a small part of the entire product. This process occurs 32 times, to end up with a 64-bit-long product.

4. Division

In MIPS logical division, we find a quotient and a remainder using two values: the dividend and the divisor which are both 32-bit-long.

We start the division process with the most significant bit or MSB of the dividend and align it with the divisor. The section of the dividend is compared to the divisor: if the section of the dividend is equal to or greater than the divisor, the section of dividend is subtracted from the divisor with the quotient bit as one. On the other hand, if the section if the dividend is less than the divisor, no subtraction occurs and the quotient bit is zero. The divisor shifts right one bit and is aligned again with the dividend to compare.

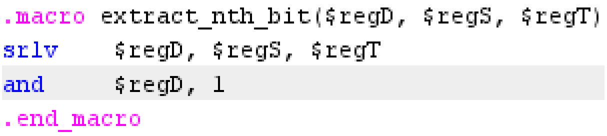
This process occurs 32 times, to end up with a 32-bit quotient and a 32-bit remainder.

B. *Code Implementation*

1. Utility Macros

The following macros are in the file cs47_proj_macro.asm to assist the code implementation of the arithmetic operations.

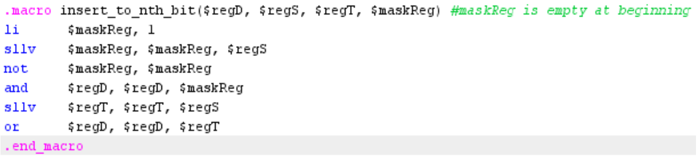
a) extract_nth_bit



```
.macro extract_nth_bit($regD, $regS, $regT)
srlv    $regD, $regS, $regT
and     $regD, 1
.end_macro
```

What the extract_nth_bit macro does is use srlv to take $regS and shift it by the amount specified in $regT. After shifting, $regD is used to store the bit at the point.

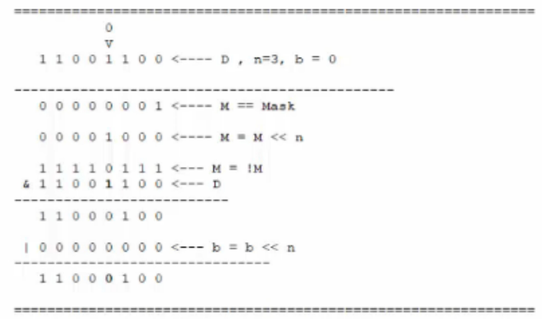
b) insert_to_nth_bit



```
.macro insert_to_nth_bit($regD, $regS, $regT, $maskReg) #maskReg is empty at beginning
li      $maskReg, 1
sllv    $maskReg, $maskReg, $regS
not     $maskReg, $maskReg
and     $regD, $regD, $maskReg
sllv    $regT, $regT, $regS
or      $regD, $regD, $regT
.end_macro
```

What insert_to_nth_bit does is it takes $regT, inserts it into $regD and the position specified by $regS. To start off the procedure, $maskReg is inverted after it is shifted to the left by the amount specified in $regS. We then use the *and* operation on $regD and $maskReg and store the end result in $regD, as shown above. Next, we perform a similar operation on $regT, which is shifted to the left and overwritten. And to complete the procedure/macro, we insert $regT into $regD and overwrite $regD.

We can also look at a more visual representation of this process below:

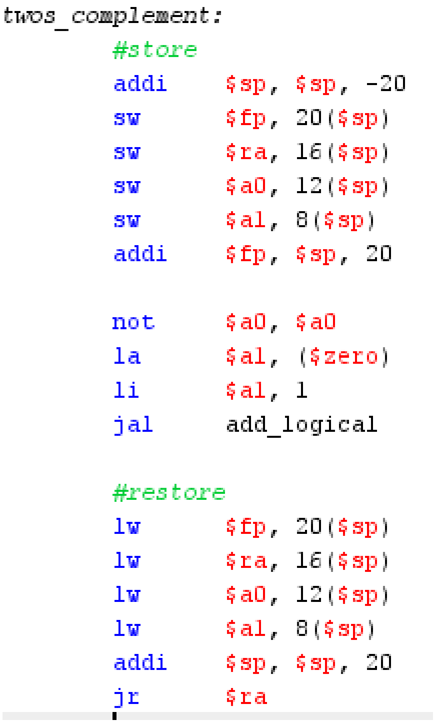


```
==============================================================
             0
             V
   1 1 0 0 1 1 0 0 <---- D , n=3, b = 0

---------------------------------------------
   0 0 0 0 0 0 0 1 <---- M == Mask

   0 0 0 0 1 0 0 0 <---- M = M << n

   1 1 1 1 0 1 1 1 <--- M = !M
 & 1 1 0 0 1 1 0 0 <--- D
-------------------------
   1 1 0 0 0 1 0 0

 | 0 0 0 0 0 0 0 0 <--- b = b << n
------------------------------
   1 1 0 0 0 1 0 0

==============================================================
```

2. *Utility Procedures*

The following procedures are in the file cs47_proj_alu_logical.asm.

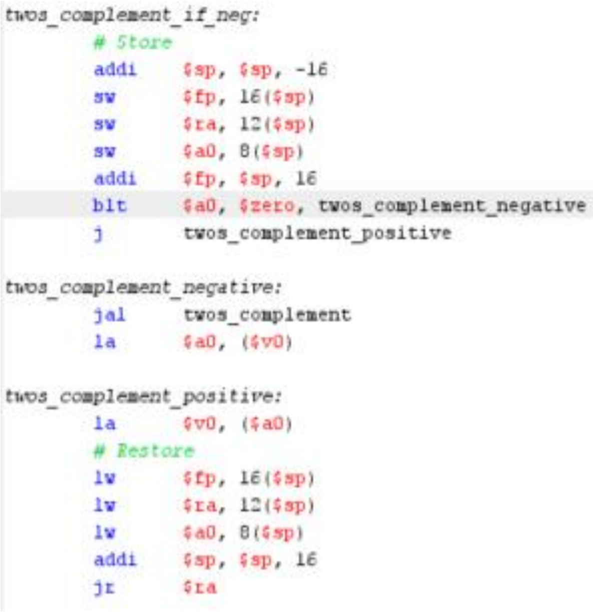
a) *twos_complement*



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

        not     $a0, $a0
        la      $a1, ($zero)
        li      $a1, 1
        jal     add_logical

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

The twos_complement procedure takes a value, in this case, $a0 and converts it into it's twos complement, which is ~$a0 + 1. We start the procedure with setup, we create the RTE frame and then complement $a0 using not. Then we load $a1 with 0x1 and add it with the now complemented $a0 in order to use add_logical. When we use add_logical, we add ~$a0 and 1 to $v0 and store it there. Once add_logical completes and returns the 2s complement of the original number($v0), we restore the RTE frame.

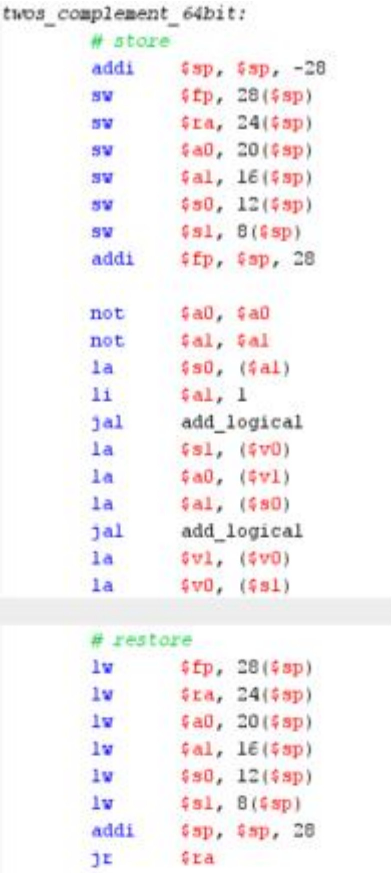
b) *twos_complement_if_neg(+helper macros)*



```
twos_complement_if_neg:
        # Store
        addi    $sp, $sp, -16
        sw      $fp, 16($sp)
        sw      $ra, 12($sp)
        sw      $a0, 8($sp)
        addi    $fp, $sp, 16
        blt     $a0, $zero, twos_complement_negative
        j       twos_complement_positive

twos_complement_negative:
        jal     twos_complement
        la      $a0, ($v0)

twos_complement_positive:
        la      $v0, ($a0)
        # Restore
        lw      $fp, 16($sp)
        lw      $ra, 12($sp)
        lw      $a0, 8($sp)
        addi    $sp, $sp, 16
        jr      $ra
```

This procedure is used to check of the sign of the value being converted to twos complement. If positive, then twos_complement_positive is called, and if the value is negative, twos_complement_negative is called. To start the procedure, we store the RTE frame and use blt to verify what sign the value/$a0 is negative, if so, the procedure moves to twos_complement_negative and the result is stored in $a0. If the value passes the blt, we know it’s positive and jump to twos_complement_positive which has the frame restoration and the return. To end off the procedure, we store $a0 in $v0 to return it and restore the frame to end the procedure.

c) twos_complement_64bit



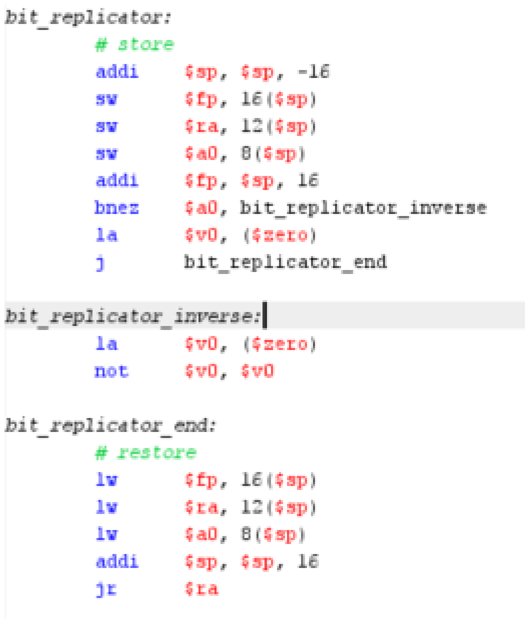
```
twos_complement_64bit:
        # store
        addi    $sp, $sp, -28
        sw      $fp, 28($sp)
        sw      $ra, 24($sp)
        sw      $a0, 20($sp)
        sw      $a1, 16($sp)
        sw      $s0, 12($sp)
        sw      $s1, 8($sp)
        addi    $fp, $sp, 28

        not     $a0, $a0
        not     $a1, $a1
        la      $s0, ($a1)
        li      $a1, 1
        jal     add_logical
        la      $s1, ($v0)
        la      $a0, ($v1)
        la      $a1, ($s0)
        jal     add_logical
        la      $v1, ($v0)
        la      $v0, ($s1)

        # restore
        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
```

How twos_complement_64bit differs from the normal twos_complement is pretty self-explanatory as this twos complement procedure is for converting 64-bit values into twos complement. The procedure is made use of in the logical multiplication since it’s product is 64-bits-long.

In this procedure we have $a0 which acts as the lo part of the given value and $a1 which acts as the hi part of the value. Once we create the frame, on order to convert $a0 and $a1, we have to invert both of them using not. We then combine $a0 and 1 with load immediate and use add_logical to compute the lo portion of the overall twos complement value. We then store this in $s1 and store the carry in $a0 to use it again for calculating the twos complement for the hi portion.

Using add_logical, we add $a0 to $a1 and save that value in $v1. To complete the procedure, we store $s1 back to $v0, while $v1 already contains the hi portion. We then store the frame to end the procedure.

d) bit_replicator



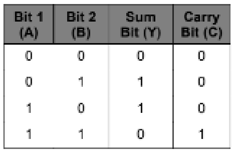
```
bit_replicator:
        # store
        addi    $sp, $sp, -16
        sw      $fp, 16($sp)
        sw      $ra, 12($sp)
        sw      $a0, 8($sp)
        addi    $fp, $sp, 16
        bnez    $a0, bit_replicator_inverse
        la      $v0, ($zero)
        j       bit_replicator_end

bit_replicator_inverse:
        la      $v0, ($zero)
        not     $v0, $v0

bit_replicator_end:
        # restore
        lw      $fp, 16($sp)
        lw      $ra, 12($sp)
        lw      $a0, 8($sp)
        addi    $sp, $sp, 16
        jr      $ra
```

Bit_replicator is a somewhat straightforward procedure in the sense that all we do is replicate a singular bit 32 times. As usual, we start the procedure by creating the frame, we then check to see if $a0 contains 0x1. If it's 0, then $v0 is stored full of zeroes and we end the procedure. However, if it's one, then we jump to bit_replicator_inverse where $v0 is again stored full of zeros, but this time inverted after doing so. We end the procedure by restoring the frame.

3. *Addition/Subtraction*

Before looking at the implementations in machine code, here's a visual representation of how binary addition works with 1's and 0's:

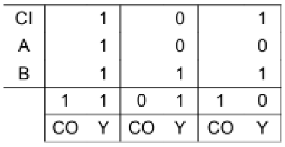
Binary Two Single Bit Addition Result



Bit 1 (A)	Bit 2 (B)	Sum Bit (Y)	Carry Bit (C)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

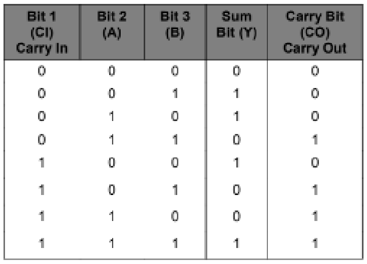
Half Addition

Example



CI		1		0		1
A		1		0		0
B		1		1		1
	1	1	0	1	1	0
	CO	Y	CO	Y	CO	Y

Binary Three Single Bit Addition Result



Bit 1 (CI) Carry In	Bit 2 (A)	Bit 3 (B)	Sum Bit (Y)	Carry Bit (CO) Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Full Addition

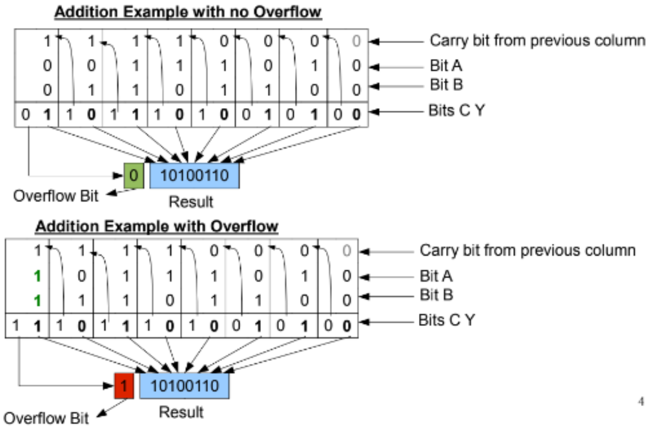
Calculation

1 + 1 + 1 = 10 + 1 = 11

0 + 0 + 1 = 00 + 1 = 01

1 + 0 + 1 = 01 + 1 = 10

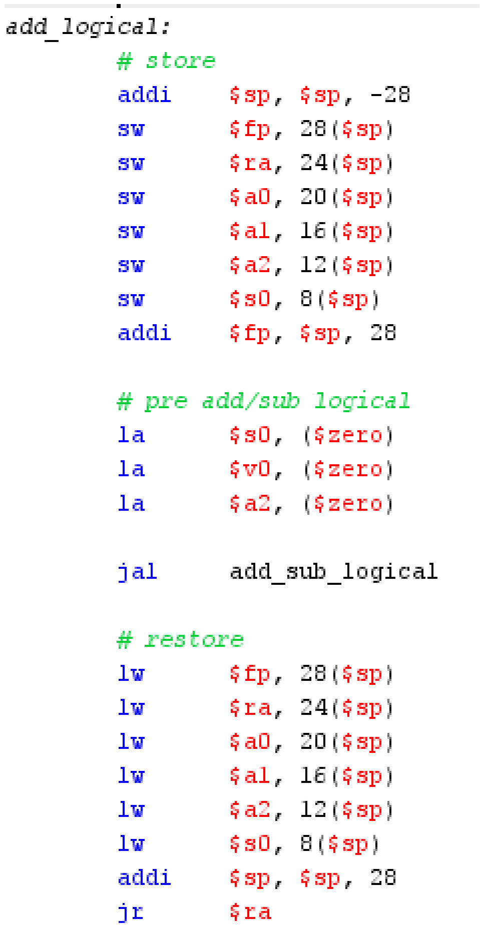
As well as with and without overflow:



Addition Example with no Overflow
Carry bit from previous column
Bit A
Bit B
Bits C Y
0
10100110
Overflow Bit
Result
Addition Example with Overflow
Carry bit from previous column
Bit A
Bit B
Bits C Y
1
10100110
Overflow Bit
Result
4

Hopefully the two preceding images will help to understand how the procedures work a little better.

a) *add_logical*

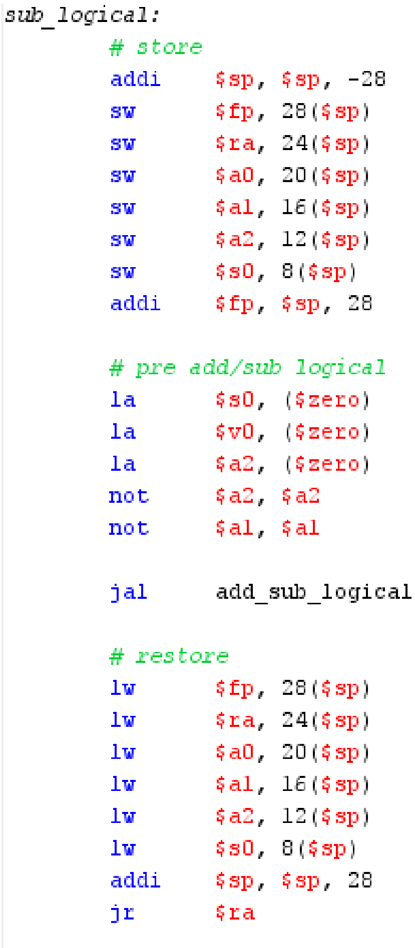


```
add_logical:
            # store
            addi    $sp, $sp, -28
            sw      $fp, 28($sp)
            sw      $ra, 24($sp)
            sw      $a0, 20($sp)
            sw      $a1, 16($sp)
            sw      $a2, 12($sp)
            sw      $s0, 8($sp)
            addi    $fp, $sp, 28

            # pre add/sub logical
            la      $s0, ($zero)
            la      $v0, ($zero)
            la      $a2, ($zero)

            jal     add_sub_logical

            # restore
            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
```

This procedure acts as a pre-procedure to add_sub_logical, which adds two values with logic operations. This procedure starts with two values to add, $a0 and $a1, and return the sum of them, with that final result in $v1 and the carry portion in $v1.

As with all procedures, we start by storing the frame. We initialize all the variables/registers needed to call add_sub_logical to zero, sum to $v0, counter to $s0, and carry to $a2. After this we jump to add_sub_logical since the prep is complete, and when we return to this method, we return $v0 with the sum and $v1 with the carry and restore the frame.

b) *sub_logical*

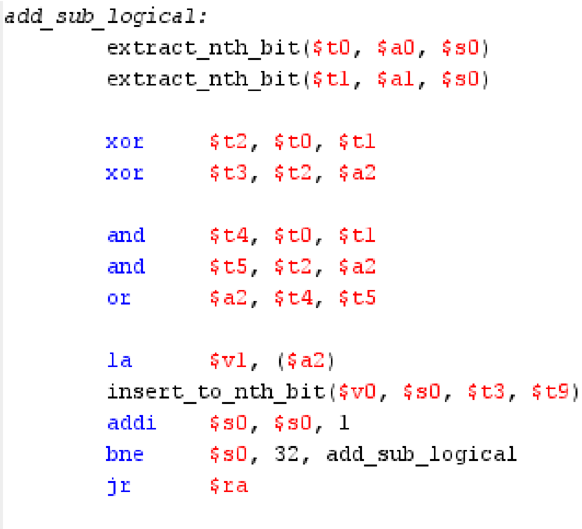


```
sub_logical:
        # store
        addi    $sp, $sp, -28
        sw      $fp, 28($sp)
        sw      $ra, 24($sp)
        sw      $a0, 20($sp)
        sw      $a1, 16($sp)
        sw      $a2, 12($sp)
        sw      $s0, 8($sp)
        addi    $fp, $sp, 28

        # pre add/sub logical
        la      $s0, ($zero)
        la      $v0, ($zero)
        la      $a2, ($zero)
        not     $a2, $a2
        not     $a1, $a1

        jal     add_sub_logical

        # restore
        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
```

This procedure acts as a pre-procedure to add_sub_logical, which will essentially subtract one value from the other with logic operations. This procedure starts with two values to subtract, $a0 and $a1, and return the $a0 - $a1, with that final result in $v1 and the carry portion in $v1.

As with all procedures, we start by storing the frame. We initialize all the variables/registers needed to call add_sub_logical to zero, difference to $v0, counter to $s0, and carry to $a2. Since this is subtraction, we invert the carry and $a1, since we can make use of the addition procedures we created by treating subtraction as just the addition of a negative number. After this we jump to add_sub_logical since the prep is complete, and when we return to this method, we return $v0 with the difference and $v1 with the carry and restore the frame.

c) *add_sub_logical*



```
add_sub_logical:
        extract_nth_bit($t0, $a0, $s0)
        extract_nth_bit($t1, $a1, $s0)

        xor     $t2, $t0, $t1
        xor     $t3, $t2, $a2

        and     $t4, $t0, $t1
        and     $t5, $t2, $a2
        or      $a2, $t4, $t5

        la      $v1, ($a2)
        insert_to_nth_bit($v0, $s0, $t3, $t9)
        addi    $s0, $s0, 1
        bne     $s0, 32, add_sub_logical
        jr      $ra
```

The main portion of the addition/subtraction code is done in add_sub_logical, since both add_logical and sub_logical are both like prep procedures for this one. Add_logical preps $a0 and $a1 as positive while sub_logical preps $a0 as positive and $a1 as negative or inverted, which are both now input values for this procedure.

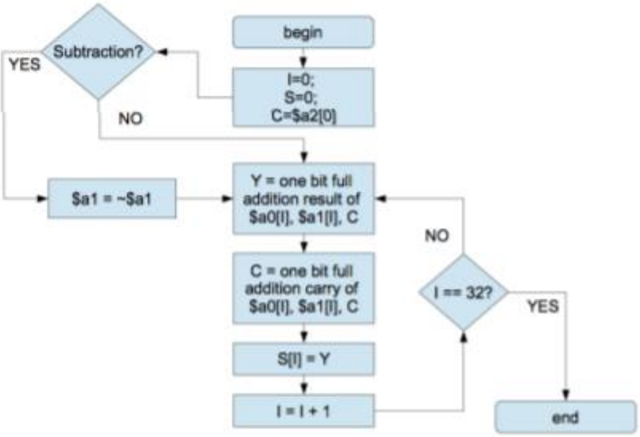
First we call extract_nth_bit, and take the bit specified by $s0 in $a0/$a1 and store it in $t0/$t1. Then we move on to finding the sum by using xor.

We use xor on the two values stored in $t0 and $t1 and store in $t2. We again use xor on $a2 and the newly overwritten $t2 and store the result in $t3 which is the sum/difference bit.

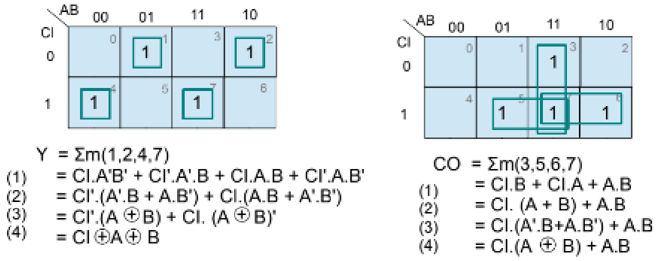
We use the and/or operations next to find the carry bit, starting by using and on $t0 and $t1, and saving the result in $t4. We repeat the procedure for $t2 and $a2 and store it in $t5. Now we use the or between $t4 and $t5 and store the result in $a2 which is the carry bit.

To end the procedure, store the newly computed values into their respective registers. The sum bit or $t3 needs to be inserted at $s0, which is why we call the utility macro insert_to_nth_bit. This process is done 32 times and returns to the procedure, add_logical or sub_logical, from where it was called.

We can see a more visual representation of this code below:


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

Provided below is also the Karnaugh-map and logical design for the add/sub operations:

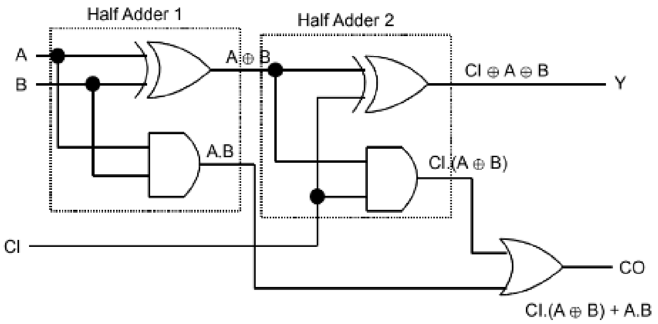


Y = Σm(1,2,4,7)
(1) = CI.A'B' + CI'.A'.B + CI.A.B + CI'.A.B'
(2) = CI'.(A'.B + A.B') + CI.(A.B + A'.B')
(3) = CI'.(A ⊕ B) + CI. (A ⊕ B)'
(4) = CI ⊕ A ⊕ B

CO = Σm(3,5,6,7)
(1) = CI.B + CI.A + A.B
(2) = CI. (A + B) + A.B
(3) = CI.(A'.B+A.B') + A.B
(4) = CI.(A ⊕ B) + A.B

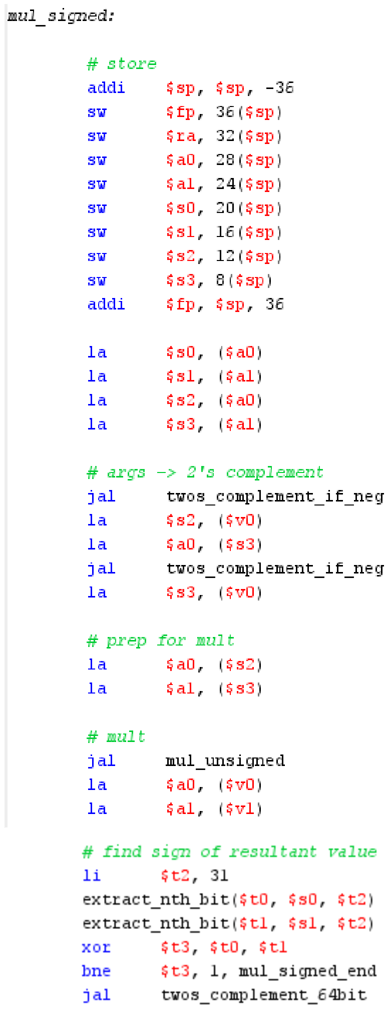
Y = CI⊕(A⊕B)

CO = CI.(A ⊕ B) + A.B

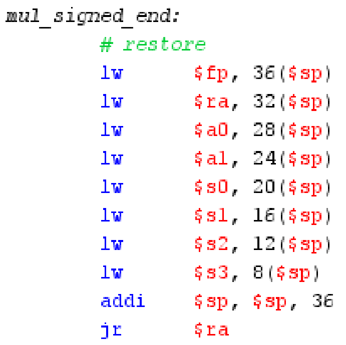


4. *Multiplication*

a) *mul_signed/mul_signed_end*



```
mul_signed:

        # store
        addi    $sp, $sp, -36
        sw      $fp, 36($sp)
        sw      $ra, 32($sp)
        sw      $a0, 28($sp)
        sw      $a1, 24($sp)
        sw      $s0, 20($sp)
        sw      $s1, 16($sp)
        sw      $s2, 12($sp)
        sw      $s3, 8($sp)
        addi    $fp, $sp, 36

        la      $s0, ($a0)
        la      $s1, ($a1)
        la      $s2, ($a0)
        la      $s3, ($a1)

        # args -> 2's complement
        jal     twos_complement_if_neg
        la      $s2, ($v0)
        la      $a0, ($s3)
        jal     twos_complement_if_neg
        la      $s3, ($v0)

        # prep for mult
        la      $a0, ($s2)
        la      $a1, ($s3)

        # mult
        jal     mul_unsigned
        la      $a0, ($v0)
        la      $a1, ($v1)

       # find sign of resultant value
       li      $t2, 31
       extract_nth_bit($t0, $s0, $t2)
       extract_nth_bit($t1, $s1, $t2)
       xor     $t3, $t0, $t1
       bne     $t3, 1, mul_signed_end
       jal     twos_complement_64bit
```



```
mul_signed_end:
        # restore
        lw      $fp, 36($sp)
        lw      $ra, 32($sp)
        lw      $a0, 28($sp)
        lw      $a1, 24($sp)
        lw      $s0, 20($sp)
        lw      $s1, 16($sp)
        lw      $s2, 12($sp)
        lw      $s3, 8($sp)
        addi    $sp, $sp, 36
        jr      $ra
```

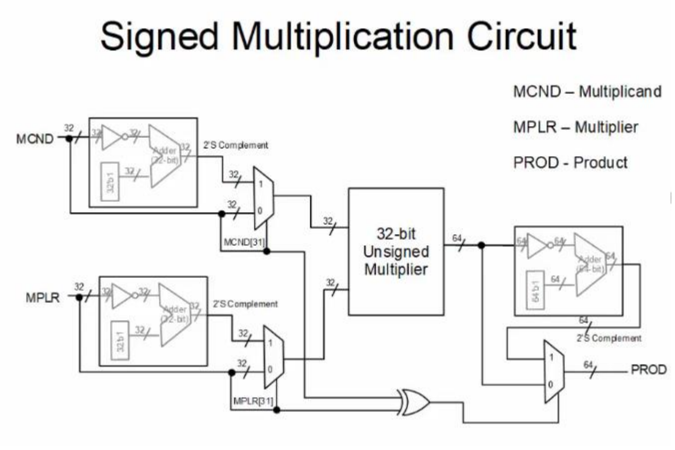
Like the other procedures, mul_signed takes in two values, $a0 and $a1, but this time, we multiply these values instead of adding/subtracting them, and like the input, we also return the output in two pieces. The two pieces being the hi part($v1) and the lo part($v0). In this procedure, we make sure all the values/argument are checked to see if they have the correct form(ex. Twos complement) before multiplying and returning the product.

Start by storing the frame and storing the input values $a0 and $a1 into other variables for later use, in this case, we use $s2 for $a0 and $s3 for $a1. To check if the value has to be in twos complement, we call twos_complement_if_neg and store the new results(if it's called) into the copied registers $s2 and $s3.

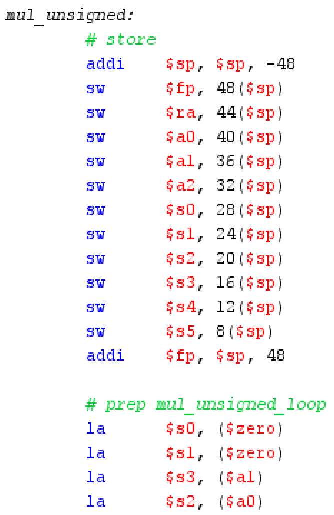
We then prepare for the multiplication step by loading the $s2 and $s3 values to $a0 and $a1 respectively. Then we call mul_unsigned, after which, the lo part of the result is in $a0 and the hi on $a1.

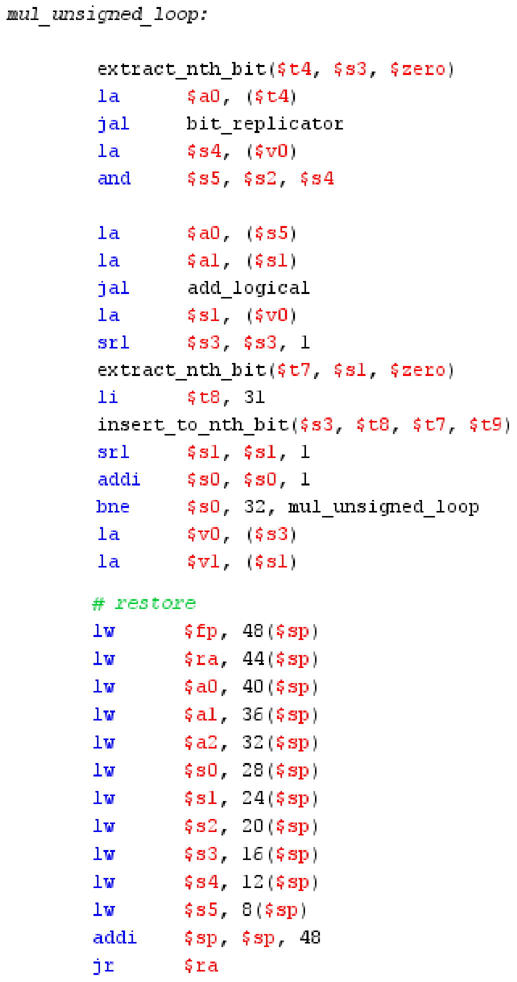
We then verify/check the sign of the values produced by mul_unsigned by calling extract_nth_bit(which then takes the determining sign bit), and then using xor. The result of this chain of operations is stored in $t3. When compared to 1, if $t3 is positive, we jump to mul_signed_end also known as the frame restoration and return the values of the product. Otherwise, if $t3 is negative, we use twos_complement_64bit to convert the value into twos complement before the frame restoration.

We can see a more visual representation of mul_signed on the circuit-side below:



b) *mul_unsigned*


```
mul_unsigned:
        # store
        addi    $sp, $sp, -48
        sw      $fp, 48($sp)
        sw      $ra, 44($sp)
        sw      $a0, 40($sp)
        sw      $a1, 36($sp)
        sw      $a2, 32($sp)
        sw      $s0, 28($sp)
        sw      $s1, 24($sp)
        sw      $s2, 20($sp)
        sw      $s3, 16($sp)
        sw      $s4, 12($sp)
        sw      $s5, 8($sp)
        addi    $fp, $sp, 48

        # prep mul_unsigned_loop
        la      $s0, ($zero)
        la      $s1, ($zero)
        la      $s3, ($a1)
        la      $s2, ($a0)
```


```
mul_unsigned_loop:

        extract_nth_bit($t4, $s3, $zero)
        la      $a0, ($t4)
        jal     bit_replicator
        la      $s4, ($v0)
        and     $s5, $s2, $s4

        la      $a0, ($s5)
        la      $a1, ($s1)
        jal     add_logical
        la      $s1, ($v0)
        srl     $s3, $s3, 1
        extract_nth_bit($t7, $s1, $zero)
        li      $t8, 31
        insert_to_nth_bit($s3, $t8, $t7, $t9)
        srl     $s1, $s1, 1
        addi    $s0, $s0, 1
        bne     $s0, 32, mul_unsigned_loop
        la      $v0, ($s3)
        la      $v1, ($s1)

        # restore
        lw      $fp, 48($sp)
        lw      $ra, 44($sp)
        lw      $a0, 40($sp)
        lw      $a1, 36($sp)
        lw      $a2, 32($sp)
        lw      $s0, 28($sp)
        lw      $s1, 24($sp)
        lw      $s2, 20($sp)
        lw      $s3, 16($sp)
        lw      $s4, 12($sp)
        lw      $s5, 8($sp)
        addi    $sp, $sp, 48
        jr      $ra
```

Mul_unsigned will compute the product of $a0 and $a1 using logic operations, and return the lo part of said product in $v0 and the hi in $v1.

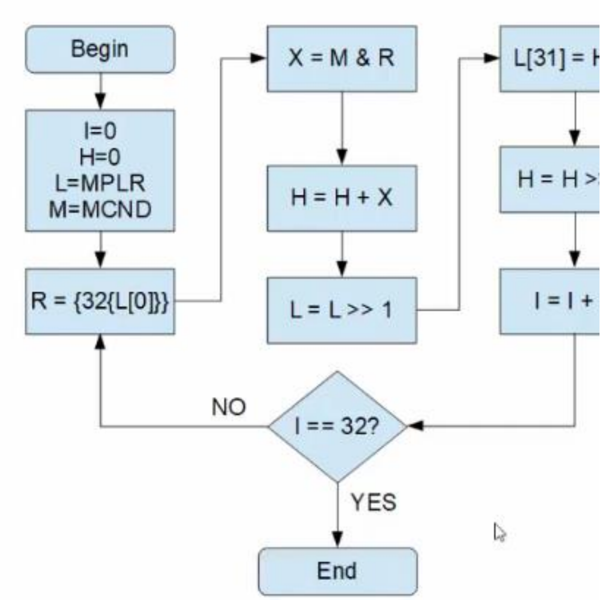
To prep mul_unsigned, we store the frame, load $a0 and $s1 with zero, and then save the $a0 to $s3(multiplier) and save $a1 to $s2(multiplicand).

To start the multiplication process, mul_unsigned_loop first calls our utility macro extract_nth_bit in order to find the least significant bit of the multiplier($s3) which is then stored in $t4. We then call bit_replicator to replicate $t4 32 times and we use load address to store that result in $s4. We use and between $s4 and $s2(multiplicand) and store that result in $s5.

We then add up $s1(Hi) and $s5 using add_logical after storing them in $a1 and $a0 respectively. We store add_logical's result in $s1 and shift $s3 one to the right. We then use extract_nth_bit on $s1 and store it in $t7. We then call insert_to_nth_bit to insert $t7 into the most significant bit of $s3. Now, we shift $s1 one to the right.

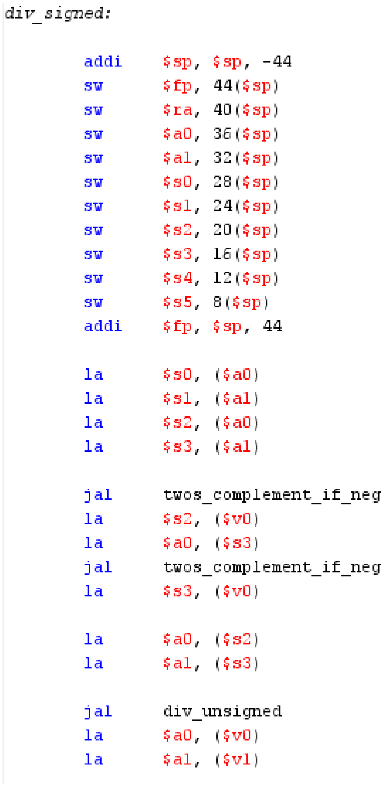
This process occurs 32 times, and after, the result is stored in $v0 and in v1.

A more visual representation of the code for mul_unsigned can be seen below:

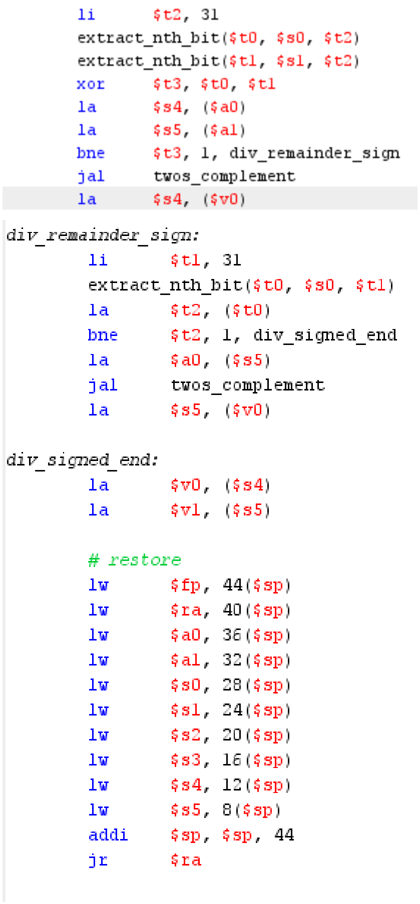


5. *Division*

a) *div_signed*



```
div_signed:

        addi    $sp, $sp, -44
        sw      $fp, 44($sp)
        sw      $ra, 40($sp)
        sw      $a0, 36($sp)
        sw      $a1, 32($sp)
        sw      $s0, 28($sp)
        sw      $s1, 24($sp)
        sw      $s2, 20($sp)
        sw      $s3, 16($sp)
        sw      $s4, 12($sp)
        sw      $s5, 8($sp)
        addi    $fp, $sp, 44

        la      $s0, ($a0)
        la      $s1, ($a1)
        la      $s2, ($a0)
        la      $s3, ($a1)

        jal     twos_complement_if_neg
        la      $s2, ($v0)
        la      $a0, ($s3)
        jal     twos_complement_if_neg
        la      $s3, ($v0)

        la      $a0, ($s2)
        la      $a1, ($s3)

        jal     div_unsigned
        la      $a0, ($v0)
        la      $a1, ($v1)
```



```
        li      $t2, 31
        extract_nth_bit($t0, $s0, $t2)
        extract_nth_bit($t1, $s1, $t2)
        xor     $t3, $t0, $t1
        la      $s4, ($a0)
        la      $s5, ($a1)
        bne     $t3, 1, div_remainder_sign
        jal     twos_complement
        la      $s4, ($v0)

div_remainder_sign:
        li      $t1, 31
        extract_nth_bit($t0, $s0, $t1)
        la      $t2, ($t0)
        bne     $t2, 1, div_signed_end
        la      $a0, ($s5)
        jal     twos_complement
        la      $s5, ($v0)

div_signed_end:
        la      $v0, ($s4)
        la      $v1, ($s5)

        # restore
        lw      $fp, 44($sp)
        lw      $ra, 40($sp)
        lw      $a0, 36($sp)
        lw      $a1, 32($sp)
        lw      $s0, 28($sp)
        lw      $s1, 24($sp)
        lw      $s2, 20($sp)
        lw      $s3, 16($sp)
        lw      $s4, 12($sp)
        lw      $s5, 8($sp)
        addi    $sp, $sp, 44
        jr      $ra
```

The procedure div_signed acts as prep for div_unsigned and also determines the signs of the results. We again use the registers $a0 and $a1 for input, and store the outputs in $v0 and $v1, which are the quotient and remainder respectively.

Start by storing the frame and storing the input values $a0 and $a1 into other variables for later use, in this case, we use $s2 for $a0 and $s3 for $a1. To check if the value has to be in twos complement, we call twos_complement_if_neg and store the new results(if it’s needed/negative) into the copied registers $s2 and $s3.

Now we let div_unsigned do the heavy lifting and put $s2 into $a0 and $s3 into $a1. Then the results are transferred from $a0 and $a1 to $v0(quotient) and $v1(remainder).

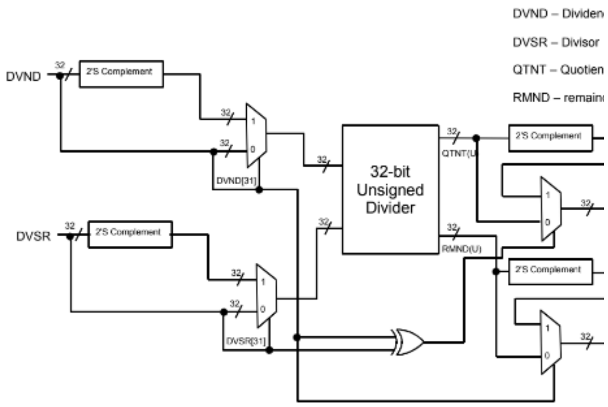
We move on to verifying the signs of our values, the quotient’s sign is determined by calling extract_nth_bit to find the most significant bit of each argument. We then call xor on both and store the resutl in $t3. If $t3 is negative when it is compared to 1, twos_complement is called, otherwise, we start finding the sign of the remainder.

Div_remainder_sign uses extract_nth_bit to find the most significant bit, which can lead to twos_complement being called should it be required.

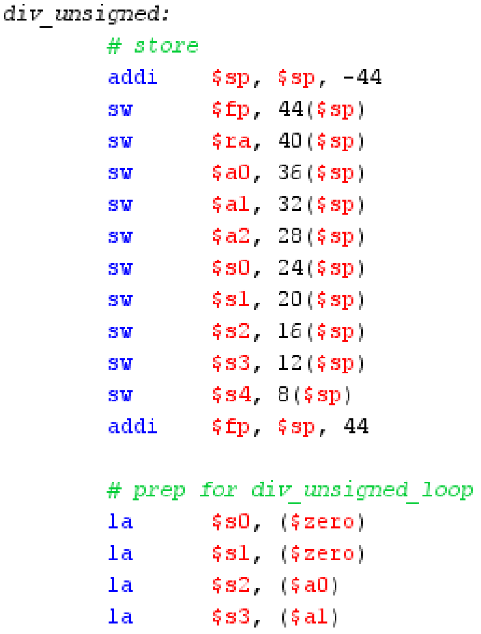
We go to div_signed_end and store our results, $s4 and $s5 into their respective spots, $v0 stores the quotient and

$v1 stores the remainder, which then ends by restoring the frame.

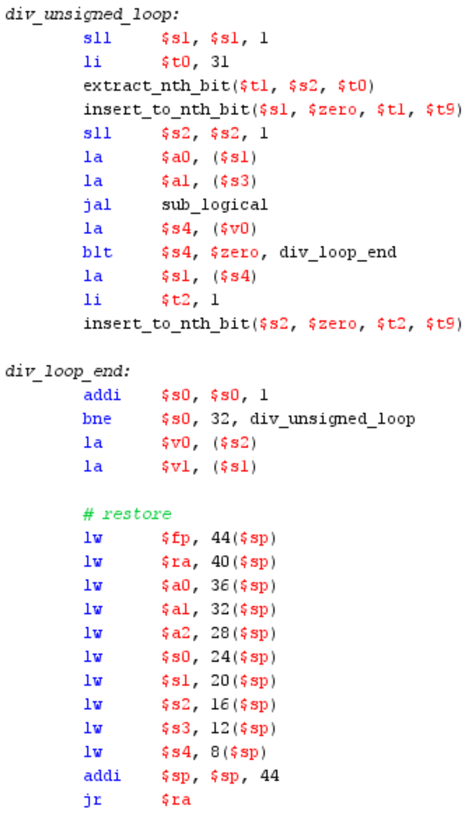
A more visual representation of the logical design of signed division(circuit design) can be seen below:



b) *div_unsigned*



```
div_unsigned:
        # store
        addi    $sp, $sp, -44
        sw      $fp, 44($sp)
        sw      $ra, 40($sp)
        sw      $a0, 36($sp)
        sw      $a1, 32($sp)
        sw      $a2, 28($sp)
        sw      $s0, 24($sp)
        sw      $s1, 20($sp)
        sw      $s2, 16($sp)
        sw      $s3, 12($sp)
        sw      $s4, 8($sp)
        addi    $fp, $sp, 44

        # prep for div_unsigned_loop
        la      $s0, ($zero)
        la      $s1, ($zero)
        la      $s2, ($a0)
        la      $s3, ($a1)
```



```
div_unsigned_loop:
        sll     $s1, $s1, 1
        li      $t0, 31
        extract_nth_bit($t1, $s2, $t0)
        insert_to_nth_bit($s1, $zero, $t1, $t9)
        sll     $s2, $s2, 1
        la      $a0, ($s1)
        la      $a1, ($s3)
        jal     sub_logical
        la      $s4, ($v0)
        blt     $s4, $zero, div_loop_end
        la      $s1, ($s4)
        li      $t2, 1
        insert_to_nth_bit($s2, $zero, $t2, $t9)

div_loop_end:
        addi    $s0, $s0, 1
        bne     $s0, 32, div_unsigned_loop
        la      $v0, ($s2)
        la      $v1, ($s1)

        # restore
        lw      $fp, 44($sp)
        lw      $ra, 40($sp)
        lw      $a0, 36($sp)
        lw      $a1, 32($sp)
        lw      $a2, 28($sp)
        lw      $s0, 24($sp)
        lw      $s1, 20($sp)
        lw      $s2, 16($sp)
        lw      $s3, 12($sp)
        lw      $s4, 8($sp)
        addi    $sp, $sp, 44
        jr      $ra
```

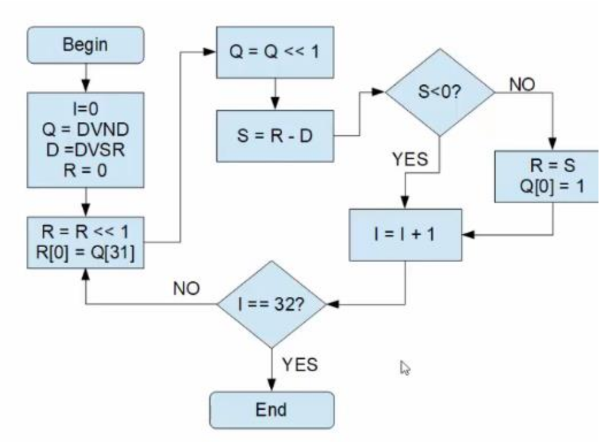
Div_unsigned will compute the quotient of $a0 and $a1 using logic operations, and return the quotient in $v0 and the remainder in $v1.

To prep div_unsigned, we store the frame, load $s0 and $s1 with zero, and then save the $a0 to $s2(dividend) and save $a1 to $s3(divisor).

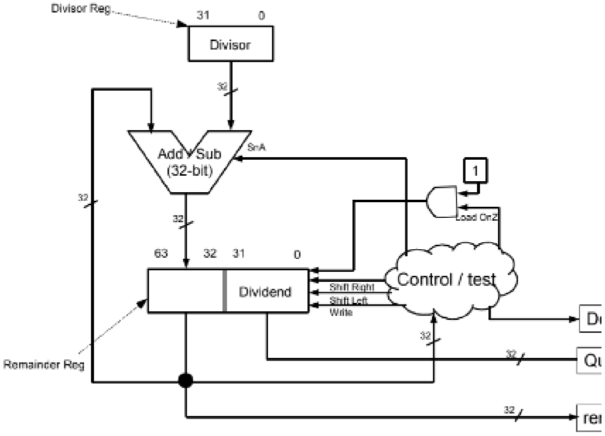
Once div_unsigned_loop starts, we shift $s1/remainder one to the left. We use utility macro extract_nth_bit to find the most significant bit of $s2/quotient and insert that bit, using utility macro insert_nth_bit, into the least significant bit of $s1/remainder. We then use sll to shift the dividend one bit to the left.

We load $s1 and $s3 to $a0 and $a1 respectively and then call sub_logical to find the difference between the remainder and the divisor(remainder-divisor), which is stored in $s4. We then use blt to compare $s4 to zero, if positive/equal to zero, $s4 becomes the remainder and the dividend's/$s2's least significant bit becomes one. Otherwise, if $s4 is less than zero, the loop moves to the next run through/iteration. This iteration is one of 32, after which, the quotient is stored to $v0 and the remainder is stored to $v1.

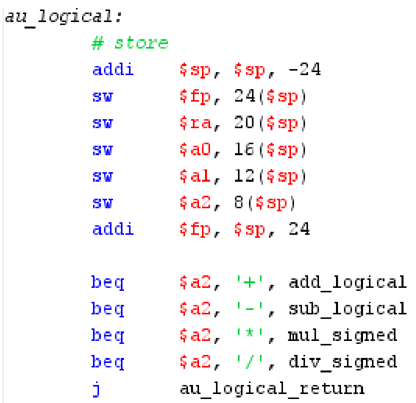
A more visual representation of div_unsigned operation can be seen above the second column of this page:

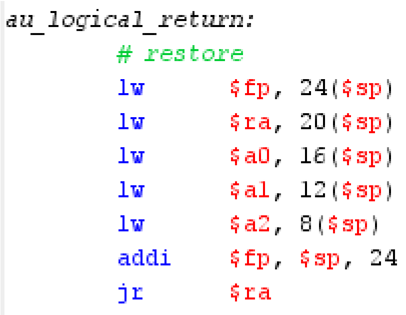


I have also provided a visual representation of what happens on the circuit-side/logical-design-side of unsigned division:



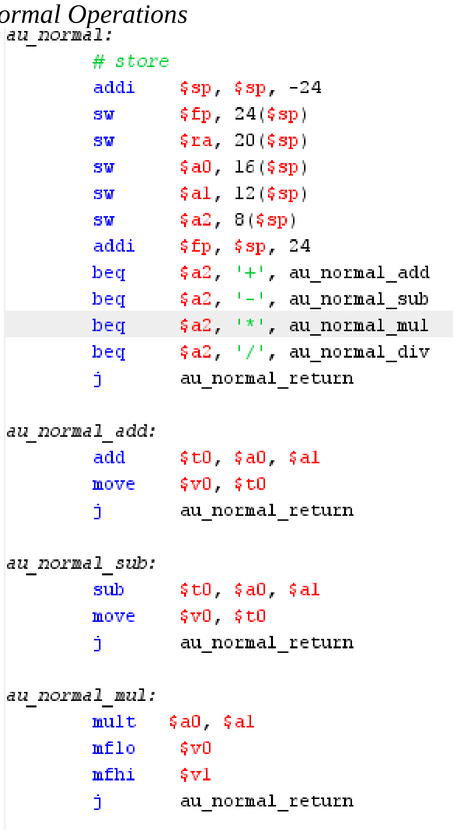
6. *Logic operations*

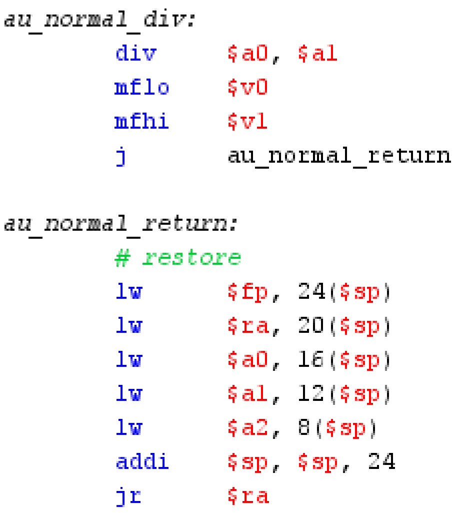

```
au_logical:
        # store
        addi    $sp, $sp, -24
        sw      $fp, 24($sp)
        sw      $ra, 20($sp)
        sw      $a0, 16($sp)
        sw      $a1, 12($sp)
        sw      $a2, 8($sp)
        addi    $fp, $sp, 24

        beq     $a2, '+', add_logical
        beq     $a2, '-', sub_logical
        beq     $a2, '*', mul_signed
        beq     $a2, '/', div_signed
        j       au_logical_return
```


```
au_logical_return:
        # restore
        lw      $fp, 24($sp)
        lw      $ra, 20($sp)
        lw      $a0, 16($sp)
        lw      $a1, 12($sp)
        lw      $a2, 8($sp)
        addi    $fp, $sp, 24
        jr      $ra
```

Au_logical is where we decide what operation to call, whether it be addition, subtraction, multiplication, or division. Au_logical starts by storing the frame and the using a set of beqs to match signs to operations. Once the operation/operations are completed, we jump to au_logical_return which restores the frame.

7. *Normal Operations*


```
au_normal:
        # store
        addi    $sp, $sp, -24
        sw      $fp, 24($sp)
        sw      $ra, 20($sp)
        sw      $a0, 16($sp)
        sw      $a1, 12($sp)
        sw      $a2, 8($sp)
        addi    $fp, $sp, 24
        beq     $a2, '+', au_normal_add
        beq     $a2, '-', au_normal_sub
        beq     $a2, '*', au_normal_mul
        beq     $a2, '/', au_normal_div
        j       au_normal_return

au_normal_add:
        add     $t0, $a0, $a1
        move    $v0, $t0
        j       au_normal_return

au_normal_sub:
        sub     $t0, $a0, $a1
        move    $v0, $t0
        j       au_normal_return

au_normal_mul:
        mult   $a0, $a1
        mflo    $v0
        mfhi    $v1
        j       au_normal_return
```



```
au_normal_div:
        div     $a0, $a1
        mflo    $v0
        mfhi    $v1
        j       au_normal_return

au_normal_return:
        # restore
        lw      $fp, 24($sp)
        lw      $ra, 20($sp)
        lw      $a0, 16($sp)
        lw      $a1, 12($sp)
        lw      $a2, 8($sp)
        addi    $sp, $sp, 24
        jr      $ra
```

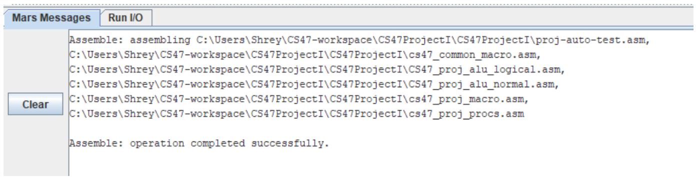
Au_normal procedures are used to check the validity of the results of our version of the arithmetic operations coded in CS47_proj_alu_logical.

We start by storing the frame and then going to the branch equals lines which compares $a2 to the respective arithmetic operation. At the end of whichever operation is called, we have the results stored in $v0 (or in the case of multiplication/division $v1 as well). The registers are saved and we call au_normal_return to restore the frame.

IV. Testing

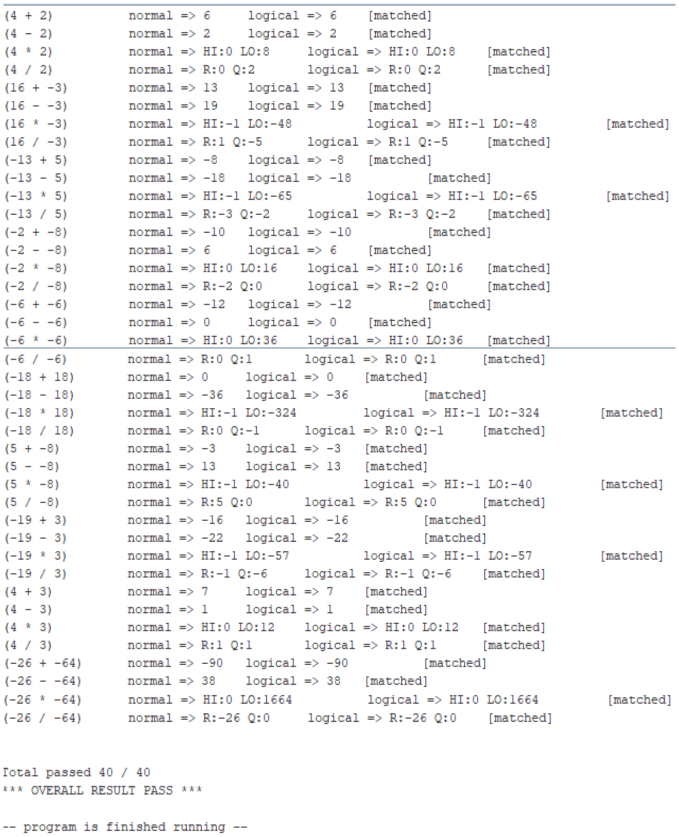
After finishing the code, we test our implementation of the arithmetic operations coded in CS47_proj_alu_logical against the ones coded in CS47_proj_alu_normal using the provided proj_auto_test file. We make sure to save all the files we edited and make sure we didn't edit any files we were not supposed to. We then assemble the program and run it by clicking the wrench/screwdriver icon and the green play button respectively. If our code works correctly, the output should look like the one above the next column:

After assembling:



```
Assemble: assembling C:\Users\Shrey\CS47-workspace\CS47ProjectI\CS47ProjectI\proj-auto-test.asm,
C:\Users\Shrey\CS47-workspace\CS47ProjectI\CS47ProjectI\cs47_common_macro.asm,
C:\Users\Shrey\CS47-workspace\CS47ProjectI\CS47ProjectI\CS47_proj_alu_logical.asm,
C:\Users\Shrey\CS47-workspace\CS47ProjectI\CS47ProjectI\CS47_proj_alu_normal.asm,
C:\Users\Shrey\CS47-workspace\CS47ProjectI\CS47ProjectI\cs47_proj_macro.asm,
C:\Users\Shrey\CS47-workspace\CS47ProjectI\CS47ProjectI\cs47_proj_procs.asm

Assemble: operation completed successfully.
```

After running:



```
(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

After completing this project, my knowledge of MIPS/MARS has definitely increased. Till this project I had no clue there was even a way to make breakpoints and/or debug in MARS. With each error I faced, I was able to learn to debug my code better/more efficiently. I also learned a lot about the common procedures, the main one being storing and restoring the frame, which was commonly seen throughout the code. This project also alloweed me to appreciate what really goes behind the scenes of a calculator as I try to punch in numbers since I'm not quite confident what 2+2 is at the moment. Something like addition we take for granted since we can just count the fingers on our hand and call it a day, but what goes on in the computer to achieve the same simple procedure is a much more arduous task. I believe that with this project, I am finally able to list MIPS/assembly language on my resume as a skill and this project to back up my knowledge.