Arithmetic Implemented By MIPS Logic Operations

Adam Golab
San Jose State University
emon.golab@live.com

Abstract— **This document contains a detailed thorough explanation regarding basic mathematical operations including addition, subtraction, multiplication, and division, using MIPS assembly language. Included within the report, is an explanation regarding both normal and logical procedures.**

I. Introduction

As found on the missouri state website, MARS is short for MIPS Assembler and Runtime Simulator. True to its description, MARS is a relatively small interactive development IDE for developing in MIPS assembly language. Using this tool, we will not only utilize the MIPS assembly language itself, but also the functions that MARS provides for us to perform these mathematical calculations. The project objectives are as follows:

- ➢ Accurately install and run the simulator
- ➢ Implement basic arithmetic calculations with both the provided MIPS mathematical and logical operations
- ➢ Test and ensure accuracy of the results using the simulator

II. Installation and Setup

A. Downloading MARS

MARS is provided publicly for students to utilize. The program can be downloaded at any time from the following link: http://courses.missouristate.edu/KenVollmar/mars/. The version used in this report is MARS 4.5.

B. Downloading and Extracting the Project

To begin, start out by downloading the given zip for the project from the SJSU Canvas website. The direct link to the assignment can be found here. Once downloaded, unzip the folder and you should notice 6 files as shown in Fig. 1.

After the file has been successfully unzipped, you should notice 6 asm files in the specified directory:

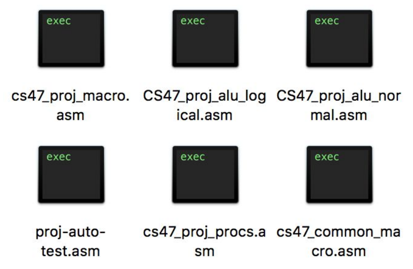


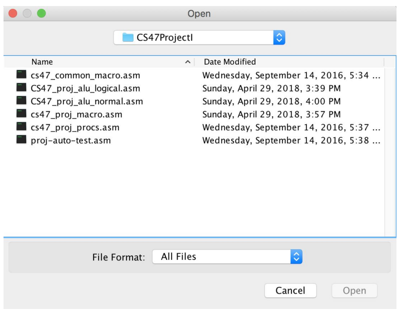
Fig. 1. Files

C. Opening the Project

Open Mars4_5.jar, then navigate to the extracted folder containing 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"*

Click on 'File' and then 'Open' and select each of the 6 files one and a time as shown in Fig. 2



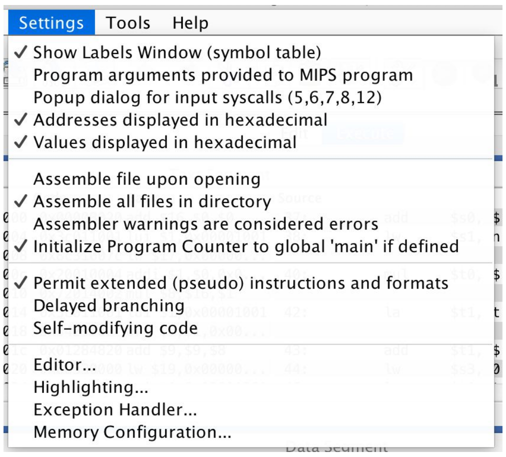
```
Open
CS47ProjectI

Name                          Date Modified
cs47_common_macro.asm         Wednesday, September 14, 2016, 5:34 ...
CS47_proj_alu_logical.asm     Sunday, April 29, 2018, 3:39 PM
CS47_proj_alu_normal.asm      Sunday, April 29, 2018, 4:00 PM
cs47_proj_macro.asm           Sunday, April 29, 2018, 3:57 PM
cs47_proj_procs.asm           Wednesday, September 14, 2016, 5:37 ...
proj-auto-test.asm            Wednesday, September 14, 2016, 5:38 ...

File Format: All Files
```

Fig. 2. Folder Navigation

Once all 6 files have been successfully opened, you should see tabs that can be switched from one to another that include the file names.

III. Environment Configuration

As specified in the project instructions, we must properly set up our programming environment. To do this, we must navigate to the 'Settings' tab, and make sure all of the selections as shown below in Fig. 3 are selected. We do this not only because of the project specifics, but for our own convenience down the line.



```
Settings  Tools  Help
✓ 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...
```

Fig. 3. Environment Settings

A. Environment Settings

Once all of the settings above are selected, we have completed our first objective and are ready to move on to begin the next step.

IV. Arguments and I/O of Arithmetic Procedures

We'll begin by initially discussing the normal procedure. This task will be completed and implemented in the *"cs47_proj_alu_normal.asm"* file. Speaking arguments and output, this procedure will take in a total of three arguments and return with two outputs. The three arguments are as follow:

1. *$a0* - holds the first value
2. *$a1* - holds the second value
3. *$a2* - holds the specific operation code

The outputs on the other hand, include register $v0 and register $v1. Where with addition and subtraction, the result will be stored in $v0. Multiplication will utilize both $v0 and $v1 where $v0 will contain the LO portion and $v1 will contain the HI portion. With Division, $v0 will contain the quotient, and $v1 will contain the remainder.

V. Implementation of Normal Arithmetic Procedures

Before beginning, it's important to be aware that in most programming scenarios, the standard MIPS arithmetic procedures will suffice for basic calculations.

In this case, the basic MIPS arithmetic procedures will suffice for our Normal Procedure program. With the given three arguments, we must initially check with operation we're going to complete by looking at $a2. It can be broken down into 4 cases:

1. ***$a2 equals '+'***
   If the a2 equals '+' in ascii, then it should jump to addition instruction.

2. ***$a2 equals '-'***
   If the a2 equals '-' in ascii, then it should jump to subtraction instruction.

3. ***$a2 equals '*'***
   If the a2 equals '*' in ascii, then it should jump to multiplication instruction

4. ***$a2 equals '/'***
   If the a2 equals '/' in ascii, then it should jump to division instruction

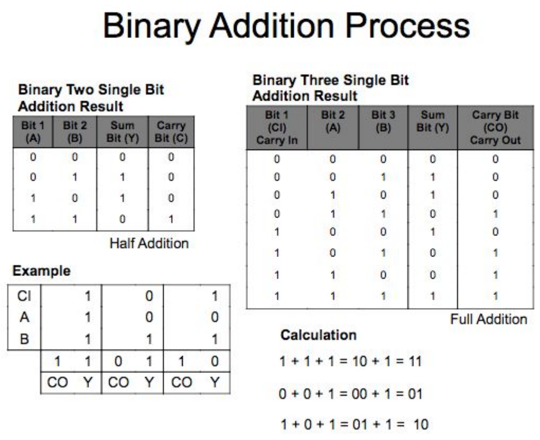
```
au_normal:
        beq     $a2, '+', ADD
        beq     $a2, '-', SUB
        beq     $a2, '*', MUL
        beq     $a2, '/', DIV
ADD:
        add     $v0, $a0, $a1
        jr      $ra
SUB:
        sub     $v0, $a0, $a1
        jr      $ra
MUL:
        mult    $a0, $a1
        mflo    $v0
        mfhi    $v1
        jr      $ra
DIV:
        div     $a0, $a1
        mflo    $v0
        mfhi    $v1
        jr      $ra
```

Fig. 4. Implementation of Normal Arithmetic

VI. Implementation of Logical Arithmetic Procedures

Similar to the implementation of the normal arithmetic procedures, the logical implementation mirrors the basics with just a few minor differences. The logical arithmetic procedures utilize a few utility macros that function 'behind the scenes' in the provided *"cs47_proj_macro.asm"* file. Before getting into actual assembly implementation, it's important to understand the underlying logical behind each of the operations themselves.

A. Addition and Subtraction Operations



Binary Addition Process

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

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

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

Calculation

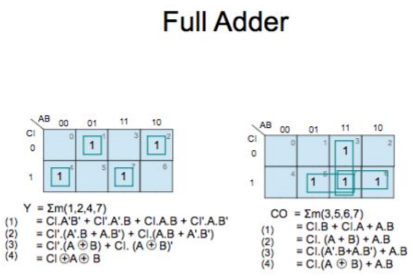
1 + 1 + 1 = 10 + 1 = 11

0 + 0 + 1 = 00 + 1 = 01

1 + 0 + 1 = 01 + 1 = 10

Fig. 5. Binary Addition Process

With Binary Addition, a sum bit and a carry bit will be produced as shown above in Fig. 5. This process is known as half addition being that it doesn't consider the previous carry bit from the steps before. To achieve full bit by bit addition, there must be two arguments including the two input bits, and a third argument that contains a carry bit from the previous bit position. For full addition, the general method of circuit implementation can be used. Start by gathering minterms for each output and express the equation in POS format. Then, we can use a K-map to reduce as shown below:



$Y = \Sigma m(1,2,4,7)$
(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) $= CI'.(A \oplus B) + CI.(A \oplus B)'$
(4) $= CI \oplus A \oplus B$

$CO = \Sigma m(3,5,6,7)$
(1) $= CI.B + CI.A + A.B$
(2) $= CI.(A + B) + A.B$
(3) $= CI.(A'.B+A.B') + A.B$
(4) $= CI.(A \oplus B) + A.B$

Fig. 6. K-map Reduction

Once the equation is determined a reduced, the digital implementation can be done. However, it is also important to note for later, that when dealing with 2's complement, an overflow condition might occur; which eventually should be accounted for.

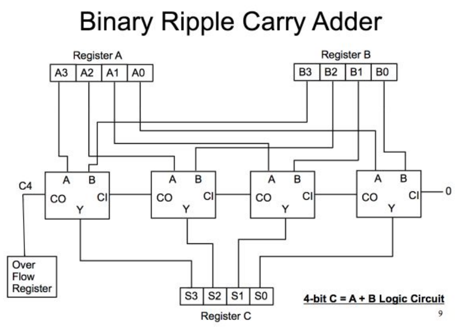


Fig. 7. Binary Ripple Carry Adder

Looking at Fig. 7, you'll notice adding multiple single bit full-adders can be grouped together to form one complete adder.

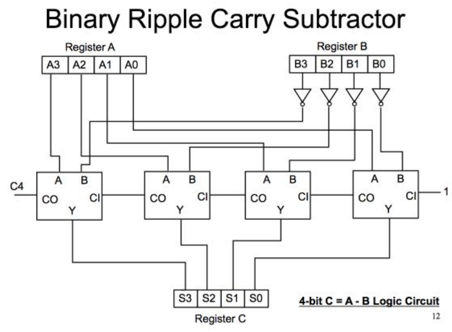


Fig. 8. Binary Ripple Carry Subtractor

When comparing Fig. 8 and Fig. 7, they're identical aside from a few minor differences. The two differences being, B is inverted for subtraction, and the carry is 1 incase of subtraction, where as for the addition the carry is 0.

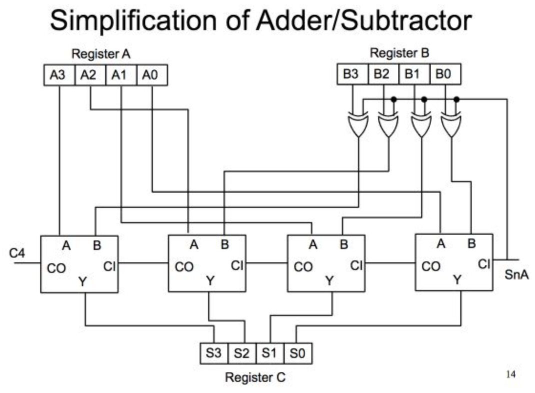
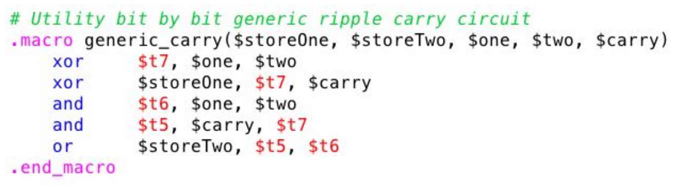


Fig. 9. Simplified Bit by Bit Adder/Subtractor

A few simplifications later as seen in Fig. 9, the two can be merged into a more efficient cleaner bit by bit generic carry adder and subtractor. The translation and implementation in assembly for this generic ripple carry circuit is fairly straightforward.



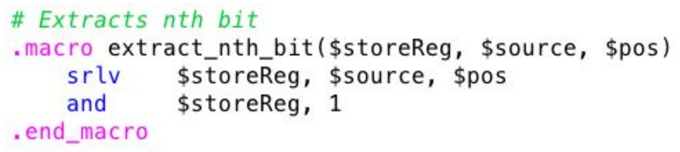
```
# Utility bit by bit generic ripple carry circuit
.macro generic_carry($storeOne, $storeTwo, $one, $two, $carry)
    xor     $t7, $one, $two
    xor     $storeOne, $t7, $carry
    and     $t6, $one, $two
    and     $t5, $carry, $t7
    or      $storeTwo, $t5, $t6
.end_macro
```

Fig. 10. Generic Ripple Carry Adder-Subtractor

The generic_carry macro as shown in Fig. 10 is used alongside with two other utility macros when dealing with the logical addition and subtraction procedure. The two other utility macros include:

1. *extract_nth_bit*

This macro will return a value at a certain index in a provided bit pattern. This macro default returns locally. Meaning, if the extracted_nth_bit needs to be saved, it needs to be load addressed into a saved register. This macro is given a $source which is a source register, which contains a bit pattern in which it will get extracted from. It is also given a $pos which holds the bit position, as well as a $storeReg which is where the extracted bit will go.

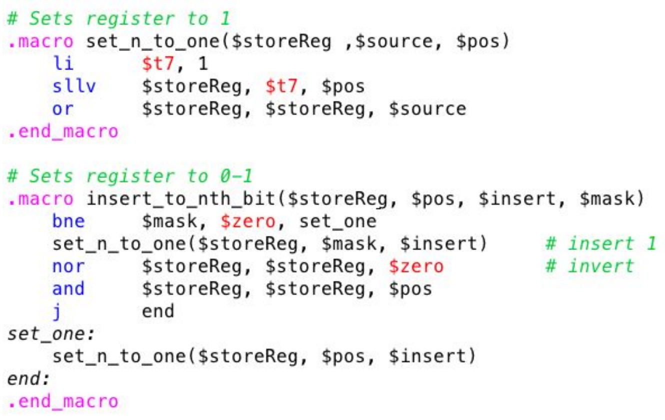


```
# Extracts nth bit
.macro extract_nth_bit($storeReg, $source, $pos)
    srlv    $storeReg, $source, $pos
    and     $storeReg, 1
.end_macro
```

Fig. 11. Extraction Implementation

2. *insert_to_nth_bit*

This macro is used to insert either a 0 or 1 into a specified bit pattern anywhere throughout one of its 32 indexes which can range from 0 up to 31. Like the previous macro above, this macro also default returns locally. Meaning, its result if wanted needs to be saved into a local saved register via load address. This macro is given a $pos which is the insertion position, a $insert which contains the bit that will be inserted, and finally a $mask which is a temporary register where the mask will be altered and created. It also contains $storeReg which is where the output will be stored.



```
# Sets register to 1
.macro set_n_to_one($storeReg ,$source, $pos)
    li      $t7, 1
    sllv    $storeReg, $t7, $pos
    or      $storeReg, $storeReg, $source
.end_macro

# Sets register to 0-1
.macro insert_to_nth_bit($storeReg, $pos, $insert, $mask)
    bne     $mask, $zero, set_one
    set_n_to_one($storeReg, $mask, $insert)     # insert 1
    nor     $storeReg, $storeReg, $zero         # invert
    and     $storeReg, $storeReg, $pos
    j       end
set_one:
    set_n_to_one($storeReg, $pos, $insert)
end:
.end_macro
```

Fig. 12. Insertion Implementation

With those few utility macros in hand, we can now move on towards implementing our common addition and subtraction logical procedure. Looking below at the provided flowchart Fig. 13, for our common procedure. The translation to assembly becomes once again fairly straight forward so long as we follow the chart;

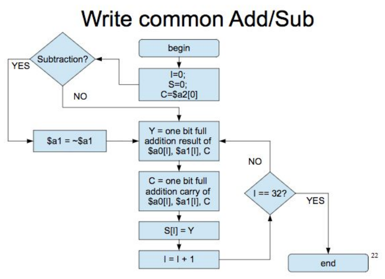

Write common Add/Sub
begin
I=0;
S=0;
C=$a2[0]
Subtraction?
YES
NO
$a1 = ~$a1
Y = one bit full addition result of $a0[I], $a1[I], C
C = one bit full addition carry of $a0[I], $a1[I], C
S[I] = Y
I = I + 1
I == 32?
NO
YES
end
22

Fig. 13. Flowchart

With the chart in mind, we can write a common macro for addition and subtraction being:

1. *add_sub_logical*

```
add_sub_logical:
        addi    $s2, $zero, 0           # i = 0
        addi    $s3, $zero, 0           # s = 0
        la      $s0, 0($a1)             # Entered Argument
        la      $s1, 0($a0)             # Entered Argument
        la      $a0, 0($a2)
        extract_nth_bit($s4, $a0, $zero)# Save LSB
        beqz    $s4, adding
        nor     $s0, $s0, $zero         # $s0 = ~$s0
adding:
        la      $a0, 0($s1)
        la      $a1, 0($s2)
        extract_nth_bit($s5, $a0, $a1)  # Save ith bit of Argument
        la      $a0, 0($s0)
        la      $a1, 0($s2)
        extract_nth_bit($t0, $a0, $a1)  # Save ith bit of Argument
        la      $a0, 0($s5)
        la      $a1, 0($t0)
        la      $a2, 0($s4)
        generic_carry($t1, $s4, $a0, $a1, $a2)  # Save $t1 for insertion check
        la      $a0, 0($s5)                     # and store $s4 overflow bit
        la      $a1, 0($t0)
        la      $a2, 0($s4)
        generic_carry($zero, $s4, $a0, $a1, $a2) # Save overflow bit into $s4
        beqz    $t1, ignore_insert      # if $t1 = 0, dont insert anything
        la      $a0, 0($s3)
        la      $a1, 0($s2)
        la      $a2, 0($t1)
        insert_to_nth_bit($s3, $a0, $a1, $a2) # Save S[i]
ignore_insert:
        addi    $s2, $s2, 1             # i++
        bge     $s2, 32, end
        j       adding
end:
        la      $v0, 0($s3)             # S
        la      $v1, 0($s4)             # Overflow bit
        jr      $ra
```

Fig. 14. Common Add/Sub Implementation

Looking above at Fig. 14, you'll notice since we have 32-bits, an initial loop is created to run 32 times. Recalling from earlier above, we also must be aware of the overflow condition, and save the overflow bit.

Onwards, we have two logical procedures that will end up calling add_sub_logical. The two logically procedures are as follow:

1. *add_logical*

This procedure will eventually call on the add_sub_logical procedure to perform addition. Being that $a2 is the determining register for the specific operation, it will be set to 0x00000000.

```
add_logical:
        addi    $a2, $zero, 0           # $a2 = 0x00000000
        jal     add_sub_logical
        j       exit
```

Fig. 15. Addition Implementation

2. *sub_logical*

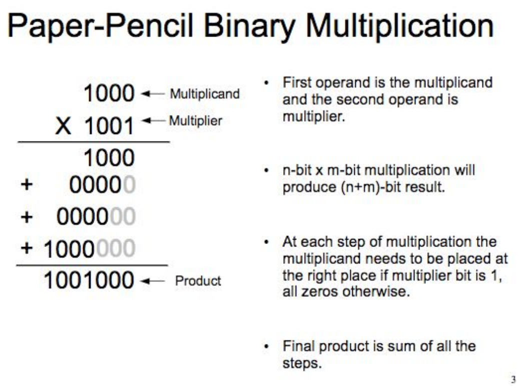
Likewise, this procedure will also eventually call on the add_sub_logical procedure however to perform subtraction. Being that $a2 is the determining register for the specific operation, it will be set to 0xFFFFFFFF.

```
sub_logical:
        addi    $a2, $zero, 0xFFFFFFFF  # $a2 = 0xFFFFFFFF
        jal     add_sub_logical
        j       exit
```

Fig. 16. Subtraction Implementation

B. *Multiplication Operation*

With the completion of addition and subtraction, we're ready to move on towards multiplication. This is where the HI and LO register come in hand. Once again, before getting into the raw assembly code, it's important to understand the background and logic of what is taking place. We'll stick with unsigned multiplication to start out with, because ultimately with the help of XOR of the MSB in both the operands, we'll be able to tell if the result should be positive or negative. We start out by looking below at Fig. 17 for an example of binary multiplication:



Paper-Pencil Binary Multiplication

```
     1000  ← Multiplicand
   X 1001  ← Multiplier
   -------
     1000
 +  00000
 + 000000
 + 1000000
   -------
   1001000 ← Product
```

- First operand is the multiplicand and the second operand is multiplier.
- n-bit x m-bit multiplication will produce (n+m)-bit result.
- At each step of multiplication the multiplicand needs to be placed at the right place if multiplier bit is 1, all zeros otherwise.
- Final product is sum of all the steps.

3

Fig. 17. Binary Multiplication

With that concept from Fig. 17 in mind, the flowchart for the binary multiplication algorithm becomes fairly straight forward:

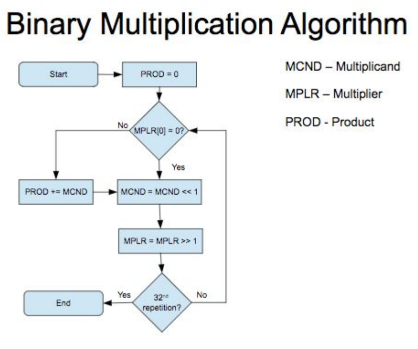


Fig. 18. Multiplication Flowchart

From the flowchart, it becomes even easier to implement the algorithm circuit wise. What's important to take away from this however, is that we have a multiplicand and a multiplier. Both of which are 32-bit numbers. That being said, when the product and multiplier come together, it results in a single 64-bit register. The implemented circuit from the flowchart above resembles Fig. 19 below:

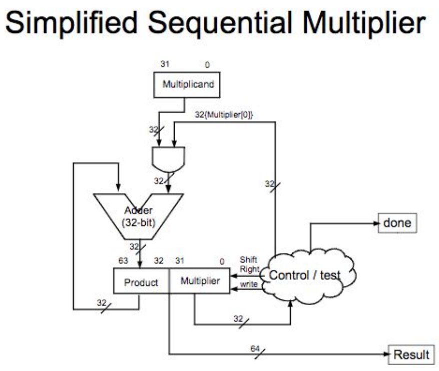
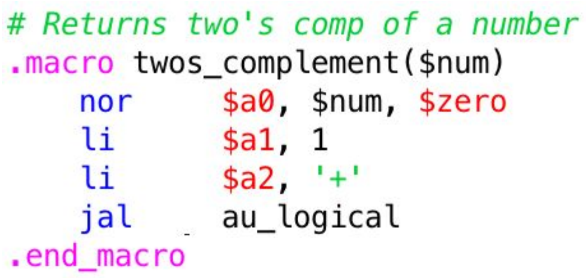


Fig. 19. Multiplier Circuit

With that, we're almost ready to begin implementing the multiplication procedure in MARS. Before doing so, we need the help of a few macros:

1. *twos_complement($num)*

The two's complement procedure takes in a $num and outputs its inverse. Being a positive number to a negative number. Likewise a negative number to a positive number.

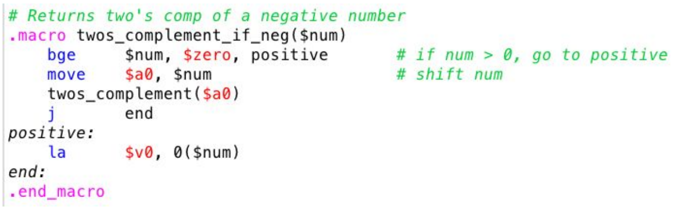


```
# Returns two's comp of a number
.macro twos_complement($num)
    nor     $a0, $num, $zero
    li      $a1, 1
    li      $a2, '+'
    jal     au_logical
.end_macro
```

Fig. 20. Implementation of twos_complement

2. *twos_complement_if_neg($num)*

The two's complement if negative procedure does essentially the same thing, but only if $num is negative. It checks if the number is negative or not, and will act appropriately if it's not negative.

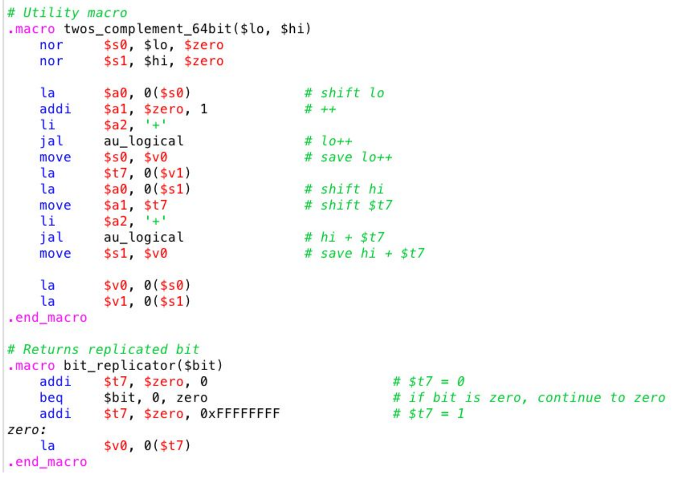


```
# Returns two's comp of a negative number
.macro twos_complement_if_neg($num)
    bge     $num, $zero, positive       # if num > 0, go to positive
    move    $a0, $num                   # shift num
    twos_complement($a0)
    j       end
positive:
    la      $v0, 0($num)
end:
.end_macro
```

Fig. 21. Implementation of twos_complement_if_neg

3. *twos_complement_64bit($lo, $hi)*

The two's complement 64 bit procedure will output the inverse of an entered 64 bit number. This is achieved with shifting and adding with the inverse of the $lo and the $hi.

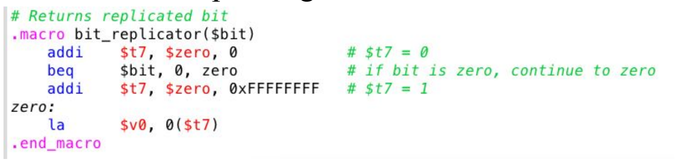


```
# Utility macro
.macro twos_complement_64bit($lo, $hi)
    nor     $s0, $lo, $zero
    nor     $s1, $hi, $zero

    la      $a0, 0($s0)             # shift lo
    addi    $a1, $zero, 1           # ++
    li      $a2, '+'
    jal     au_logical              # lo++
    move    $s0, $v0                # save lo++
    la      $t7, 0($v1)
    la      $a0, 0($s1)             # shift hi
    move    $a1, $t7                # shift $t7
    li      $a2, '+'
    jal     au_logical              # hi + $t7
    move    $s1, $v0                # save hi + $t7

    la      $v0, 0($s0)
    la      $v1, 0($s1)
.end_macro

# Returns replicated bit
.macro bit_replicator($bit)
    addi    $t7, $zero, 0                   # $t7 = 0
    beq     $bit, 0, zero                   # if bit is zero, continue to zero
    addi    $t7, $zero, 0xFFFFFFFF          # $t7 = 1
zero:
    la      $v0, 0($t7)
.end_macro
```

Fig. 22. Implementation of twos_complement_64bit

4. *bit_replicator($bit)*

The bit replicator procedure will take in either a 0x1 or a 0x0 bit pattern and repeat either of them. This procedure will result in an 0x00000000 or an 0xFFFFFFFF depending on the entered bit.



```
# Returns replicated bit
.macro bit_replicator($bit)
    addi    $t7, $zero, 0            # $t7 = 0
    beq     $bit, 0, zero            # if bit is zero, continue to zero
    addi    $t7, $zero, 0xFFFFFFFF   # $t7 = 1
zero:
    la      $v0, 0($t7)
.end_macro
```

Fig. 23. Implementation of bit_replicator

Glancing up back towards Fig. 19, we can generate a flowchart that will help us further implement the unsigned multiplication procedure.

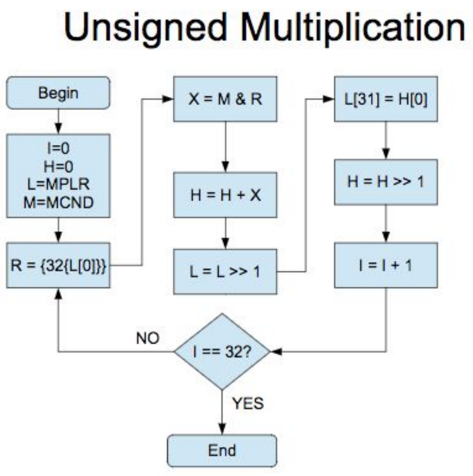
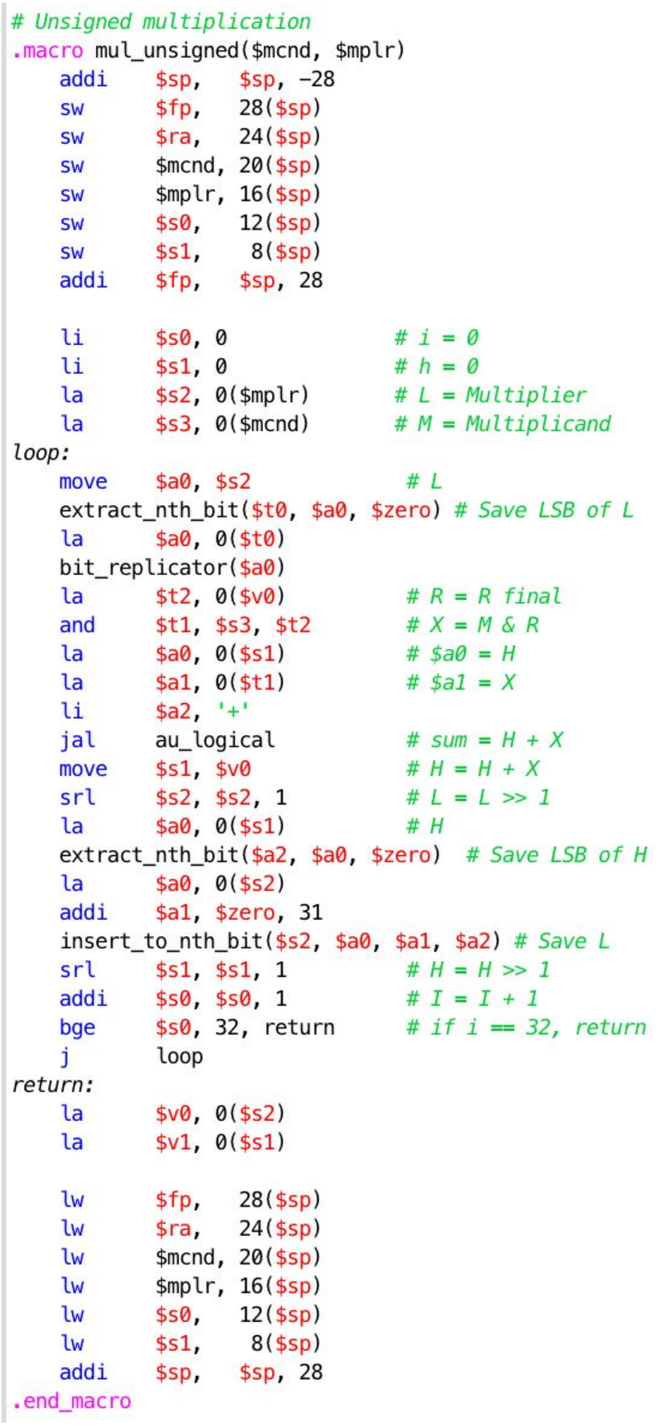


Fig. 24. Unsigned Multiplication Flowchart

We initially begin by having the upper half of the multiplier register equal zero. We then AND the multiplier with the Least Significant Bit of the multiplier which comes to get repeated where it eventually becomes X = M & R. That being said, it then get loaded into the upper half of the multiplier. LO then gets shifted which in turn makes the Most Significant Bit of LO equal the Least Significant Bit of HI. HI then in turn gets shifted, which eradicates the Least Significant Bit. This process gets repeated following this instance 31 more times. Looking back up at the flowchart above, it can be translated into assembly like so:



```
# Unsigned multiplication
.macro mul_unsigned($mcnd, $mplr)
    addi    $sp,   $sp, -28
    sw      $fp,   28($sp)
    sw      $ra,   24($sp)
    sw      $mcnd, 20($sp)
    sw      $mplr, 16($sp)
    sw      $s0,   12($sp)
    sw      $s1,    8($sp)
    addi    $fp,   $sp, 28

    li      $s0, 0              # i = 0
    li      $s1, 0              # h = 0
    la      $s2, 0($mplr)       # L = Multiplier
    la      $s3, 0($mcnd)       # M = Multiplicand
loop:
    move    $a0, $s2             # L
    extract_nth_bit($t0, $a0, $zero) # Save LSB of L
    la      $a0, 0($t0)
    bit_replicator($a0)
    la      $t2, 0($v0)          # R = R final
    and     $t1, $s3, $t2        # X = M & R
    la      $a0, 0($s1)          # $a0 = H
    la      $a1, 0($t1)          # $a1 = X
    li      $a2, '+'
    jal     au_logical           # sum = H + X
    move    $s1, $v0             # H = H + X
    srl     $s2, $s2, 1          # L = L >> 1
    la      $a0, 0($s1)          # H
    extract_nth_bit($a2, $a0, $zero)  # Save LSB of H
    la      $a0, 0($s2)
    addi    $a1, $zero, 31
    insert_to_nth_bit($s2, $a0, $a1, $a2) # Save L
    srl     $s1, $s1, 1          # H = H >> 1
    addi    $s0, $s0, 1          # I = I + 1
    bge     $s0, 32, return      # if i == 32, return
    j       loop
return:
    la      $v0, 0($s2)
    la      $v1, 0($s1)

    lw      $fp,   28($sp)
    lw      $ra,   24($sp)
    lw      $mcnd, 20($sp)
    lw      $mplr, 16($sp)
    lw      $s0,   12($sp)
    lw      $s1,    8($sp)
    addi    $sp,   $sp, 28
.end_macro
```

Fig. 25. Unsigned Multiplication Implementation

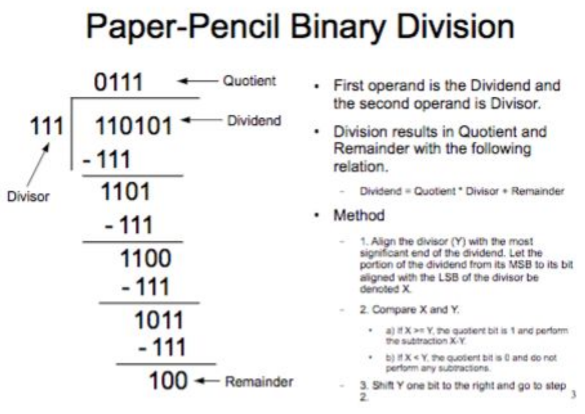
From there recalling from above, the process of determining whether the result should be positive or negative is relatively simple. We can utilize an XOR with both of the original arguments Most Significant Bits and determine whether or not the result should be positive or negative. Looking below, depending on the output of the XOR, we'll either return the output or the two's complement of the output for whatever it should be:

```
# Signed multiplication
.macro mul_signed($mcnd, $mplr)
    addi    $sp,    $sp, -28
    sw      $fp,    28($sp)
    sw      $ra,    24($sp)
    sw      $mcnd, 20($sp)
    sw      $mplr, 16($sp)
    sw      $s0,    12($sp)
    sw      $s1,     8($sp)
    addi    $fp,    $sp, 28

    la      $s0, 0($mcnd)     # initial N1
    la      $s1, 0($mplr)     # initial N2

    la      $a0, 0($s0)
    twos_complement_if_neg($a0)  # two's comp for N1
    la      $s2, 0($v0)       # new N1
    move    $a0, $s1
    twos_complement_if_neg($a0)  # two's comp for N2
    la      $t7, 0($v0)       # new N2
    la      $a0, 0($s2)
    la      $a1, 0($t7)
    mul_unsigned($a0, $a1)    # N1 * N2
    la      $t6, 0($v0)       # Rlo
    la      $t0, 0($v1)       # Rhi
    la      $a0, 0($s0)       # initial N1
    addi    $a1, $zero, 31
    extract_nth_bit($s6, $a0, $a1)  # Rlo - $s6
    la      $a0, 0($s1)       # initial N2
    addi    $a1, $zero, 31
    extract_nth_bit($s7, $a0, $a1)  # Rhi - $s7
    xor     $s0, $s6, $s7     # $s0 = $a0[31] xor $a1[31]
    bne     $s0, 1, return
    la      $a0, 0($t6)       # $a0 = Rlo
    la      $a1, 0($t0)       # $a1 = Rhi
    twos_complement_64bit($a0, $a1)
    la      $t6, 0($v0)       # new Rlo
    la      $t0, 0($v1)       # new Rhi
return:
    move    $v0, $t6          # return Rlo
    move    $v1, $t0          # return Rhi
end:
    lw      $fp,    28($sp)
    lw      $ra,    24($sp)
    lw      $mcnd, 20($sp)
    lw      $mplr, 16($sp)
    lw      $s0,    12($sp)
    lw      $s1,     8($sp)
    addi    $sp,    $sp, 28
.end_macro
```

Fig. 26. Signed Multiplication Implementation

C. Division Operation

Before moving onward to complete the final step of division, it is important to be sure that we have properly implemented the addition, subtraction, and multiplication operations. Division in a way, is similar to addition and shifting. The only difference is, it utilizes subtracting and shifting to the left.



- First operand is the Dividend and the second operand is Divisor.
- Division results in Quotient and Remainder with the following relation.
  - Dividend = Quotient * Divisor + Remainder
- Method
  - 1. Align the divisor (Y) with the most significant end of the dividend. Let the portion of the dividend from its MSB to its bit aligned with the LSB of the divisor be denoted X.
  - 2. Compare X and Y.
    - a) If X >= Y, the quotient bit is 1 and perform the subtraction X-Y.
    - b) If X < Y, the quotient bit is 0 and do not perform any subtractions.
  - 3. Shift Y one bit to the right and go to step 2.

Fig. 27. Binary Division

From Fig. 27, we can furthermore create a binary division algorithm that looks as follows:

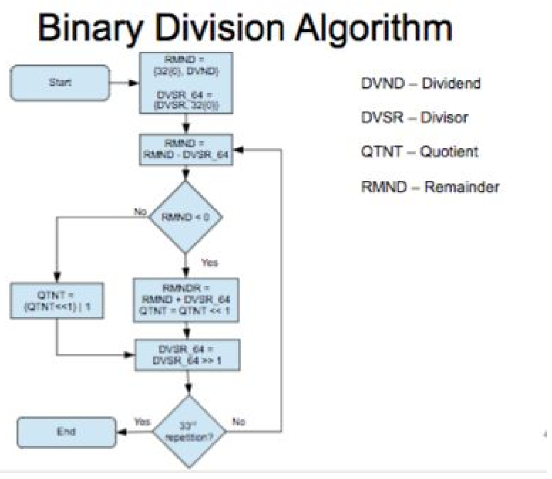


Fig. 28. Binary Division Algorithm

Looking at Fig. 28, the flowchart represents a 32-bit Binary Division Algorithm. That being said, we'll need a single 32-bit register as well as a single 64-bit register. The single 64-bit register will be two 32-bit registers combined. These registers will resemble the Quotient and the Remainder. As shown above, the algorithm will find the largest number it is able to divide by, then repeat following that instance another 31 times resulting in a Quotient and a final Remainder. From the flowchart above, we can construct a simplified binary division circuit as follows:

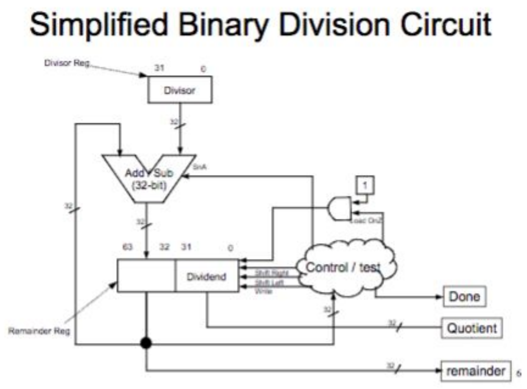


Fig. 29. Binary Division Circuit

From the circuit above, Fig. 29, we can come to deriving an Unsigned Division algorithm to implement for MIPS as follows:

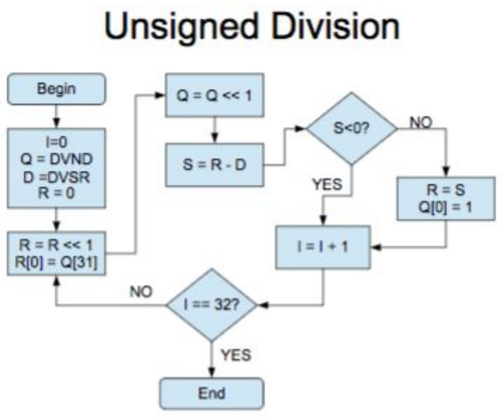
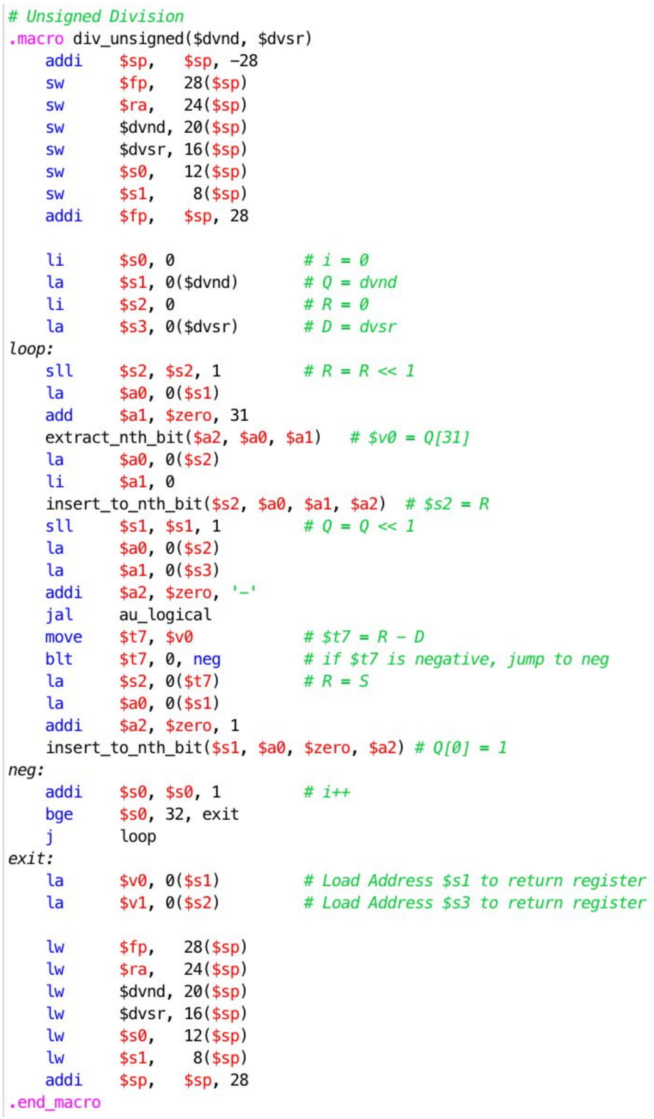


Fig. 30. Unsigned Division Algorithm

Looking at the algorithm above, we notice it starts out by initially shifting the remainder to the left. Following that, the Most Significant Bit gets pulled from the dividend. Following that, the dividend shifts to eradicate the Most Significant Bit. The result of the remainder is then stored and then subtracted by the divisor. That result will then be checked if it is a positive number, and if so will be placed in the remainders register. Finally, finishing a single iteration, a counter gets incremented. Following this single instance, the process will continue to repeat for another 31 iterations. The Unsigned Division implementation in MIPS looks as follows:

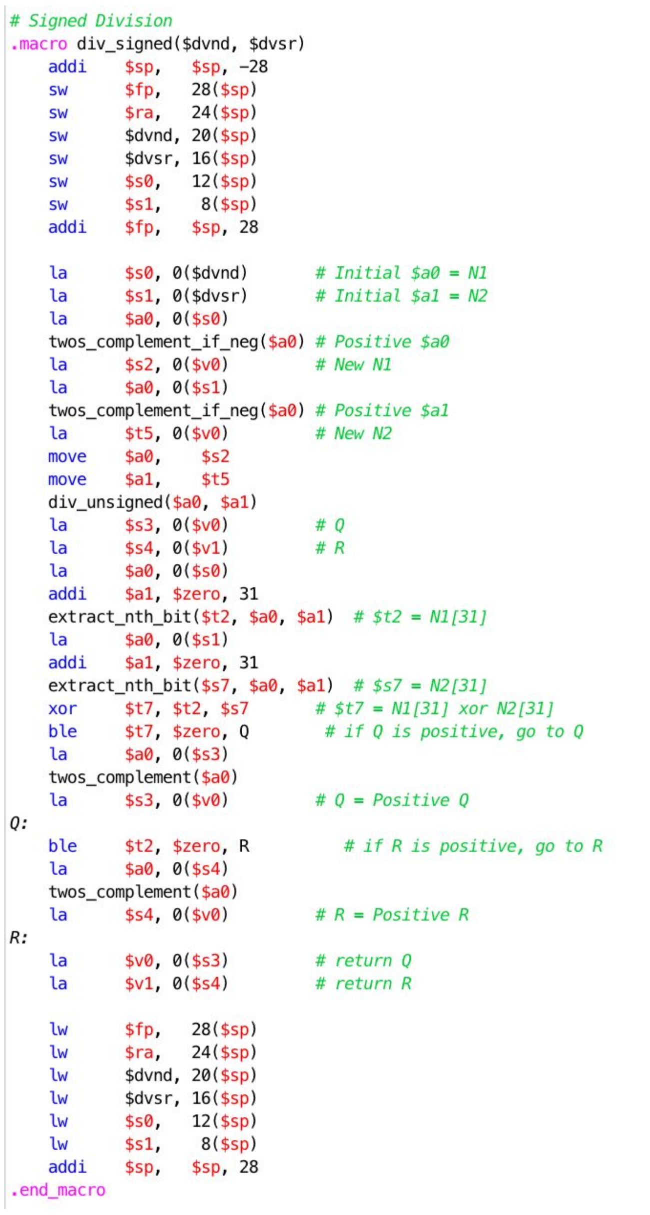


```
# Unsigned Division
.macro div_unsigned($dvnd, $dvsr)
    addi    $sp,   $sp, -28
    sw      $fp,   28($sp)
    sw      $ra,   24($sp)
    sw      $dvnd, 20($sp)
    sw      $dvsr, 16($sp)
    sw      $s0,   12($sp)
    sw      $s1,    8($sp)
    addi    $fp,   $sp, 28

    li      $s0, 0              # i = 0
    la      $s1, 0($dvnd)       # Q = dvnd
    li      $s2, 0              # R = 0
    la      $s3, 0($dvsr)       # D = dvsr
loop:
    sll     $s2, $s2, 1         # R = R << 1
    la      $a0, 0($s1)
    add     $a1, $zero, 31
    extract_nth_bit($a2, $a0, $a1)   # $v0 = Q[31]
    la      $a0, 0($s2)
    li      $a1, 0
    insert_to_nth_bit($s2, $a0, $a1, $a2)  # $s2 = R
    sll     $s1, $s1, 1         # Q = Q << 1
    la      $a0, 0($s2)
    la      $a1, 0($s3)
    addi    $a2, $zero, '-'
    jal     au_logical
    move    $t7, $v0            # $t7 = R - D
    blt     $t7, 0, neg         # if $t7 is negative, jump to neg
    la      $s2, 0($t7)         # R = S
    la      $a0, 0($s1)
    addi    $a2, $zero, 1
    insert_to_nth_bit($s1, $a0, $zero, $a2) # Q[0] = 1
neg:
    addi    $s0, $s0, 1         # i++
    bge     $s0, 32, exit
    j       loop
exit:
    la      $v0, 0($s1)         # Load Address $s1 to return register
    la      $v1, 0($s2)         # Load Address $s3 to return register

    lw      $fp,   28($sp)
    lw      $ra,   24($sp)
    lw      $dvnd, 20($sp)
    lw      $dvsr, 16($sp)
    lw      $s0,   12($sp)
    lw      $s1,    8($sp)
    addi    $sp,   $sp, 28
.end_macro
```

Fig. 31. div_unsigned Implementation

Note that the same concept with multiplication regarding the unsigned and signed translation also applies for division. We can simply find the Most Significant Bit for both of our original arguments, and toss them in an XOR. Depending on the output of the XOR, we'll know whether the result of the division needs to be marked as a positive number or a negative number.

The implementation for div_signed in MIPS looks as follows:

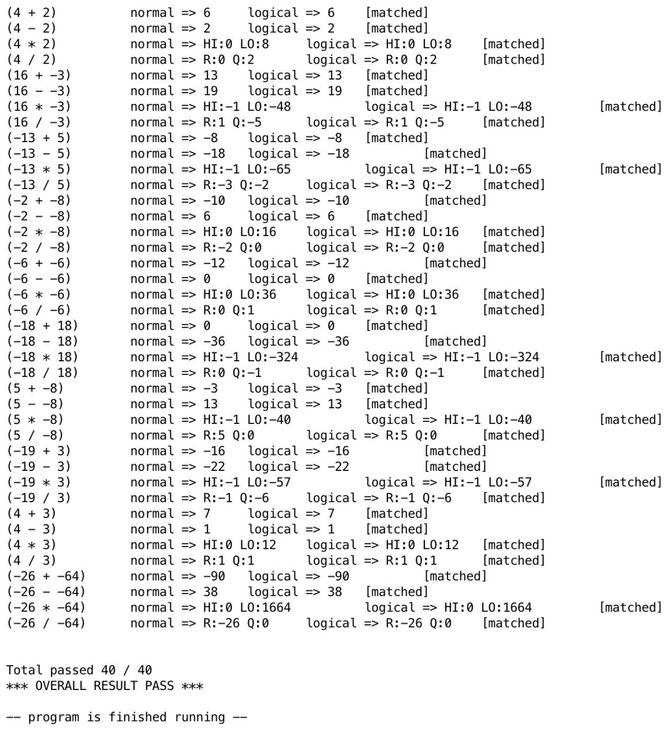


```
# Signed Division
.macro div_signed($dvnd, $dvsr)
    addi    $sp,    $sp, -28
    sw      $fp,    28($sp)
    sw      $ra,    24($sp)
    sw      $dvnd, 20($sp)
    sw      $dvsr, 16($sp)
    sw      $s0,    12($sp)
    sw      $s1,     8($sp)
    addi    $fp,    $sp, 28

    la      $s0, 0($dvnd)       # Initial $a0 = N1
    la      $s1, 0($dvsr)       # Initial $a1 = N2
    la      $a0, 0($s0)
    twos_complement_if_neg($a0) # Positive $a0
    la      $s2, 0($v0)         # New N1
    la      $a0, 0($s1)
    twos_complement_if_neg($a0) # Positive $a1
    la      $t5, 0($v0)         # New N2
    move    $a0,    $s2
    move    $a1,    $t5
    div_unsigned($a0, $a1)
    la      $s3, 0($v0)         # Q
    la      $s4, 0($v1)         # R
    la      $a0, 0($s0)
    addi    $a1, $zero, 31
    extract_nth_bit($t2, $a0, $a1)  # $t2 = N1[31]
    la      $a0, 0($s1)
    addi    $a1, $zero, 31
    extract_nth_bit($s7, $a0, $a1)  # $s7 = N2[31]
    xor     $t7, $t2, $s7       # $t7 = N1[31] xor N2[31]
    ble     $t7, $zero, Q        # if Q is positive, go to Q
    la      $a0, 0($s3)
    twos_complement($a0)
    la      $s3, 0($v0)         # Q = Positive Q
Q:
    ble     $t2, $zero, R          # if R is positive, go to R
    la      $a0, 0($s4)
    twos_complement($a0)
    la      $s4, 0($v0)         # R = Positive R
R:
    la      $v0, 0($s3)         # return Q
    la      $v1, 0($s4)         # return R

    lw      $fp,    28($sp)
    lw      $ra,    24($sp)
    lw      $dvnd, 20($sp)
    lw      $dvsr, 16($sp)
    lw      $s0,    12($sp)
    lw      $s1,     8($sp)
    addi    $sp,    $sp, 28
.end_macro
```

Fig. 32. div_signed Implementation

VII. Testing

With the completion of addition, subtraction, multiplication, and division, we have successfully completed the second objective. Throughout the entirety of the project, we were not to alter the tester file that was provided to us. The tester file being, *'proj-auto-test.asm'*. To test the procedures, we must ensure that all previous steps including set up, and proper environment have been completed. Once done, by clicking 'Assemble' then by clicking 'Run' the file should return a few calculations in the console. The purpose of this file is to test both the au_normal and the au_logical and match the results to check for accuracy possible mistakes. Should everything work as planned, the result at the bottom of the console should pass with a score of 40/40:



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


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

-- program is finished running --
```

Fig. 33. Test Results

A collection of mistakes that I ran into while working on this project include:

1. *Organization*

Initially when I first began to work on the assignment, I was struggling to keep track of everything that was being written procedure wise. Keeping track of all the registers was difficult in the main alu_logical.asm file. After hours on end of debugging, I decided to make things cleaner and more efficient by utilizing macros. With the use of the macros, I was able to more efficiently and accurately debug.

2. *Frame Storage with Proper Registers*

With another handful of hours spent debugging, I came to realize storing and restoring the proper registers apart of the frame is a necessity. Errors can occur with counters, and values not saving in registers, as well as not knowing where things are actually being stored.

VIII. Conclusion

I spent a large majority of my time working on this assignment, and I can confidently say I walked away with a better more extensive understanding of a few things. I’m far more confident with my ability and understanding of MIPS and assembly. I have come upon far too many errors regarding Stack Frames, so I most definitely feel like I’ve gained a better understanding of the implementation. Overall, this project did a good job of reinforcing a collection of all the theories we went over in class regarding low level programming. It was a nice change, compared to the high level Java classes that I’ve taken so far.

References

[1] K. Patra. CS 47. Class Lecture, Topic: “Addition Subtraction Logic.” San Jose State University, San Jose, CA, April 18 2018

[2] K. Patra. CS 47. Class Lecture, Topic: “Multiplication Logic.” San Jose State University, San Jose, CA, April 23 2018

[3] K. Patra. CS 47. Class Lecture, Topic: “Division Logic.” San Jose State University, San Jose, CA, April 25 2018

[4] Chapter 3 of ‘Computer Organization & Design’ by Hennesy, Patterson

[5] Chapter 4 of ‘Logic and Computer Design Fundamentals’ by Mano, Kime