Implementing Arithmetic Operation Using MIPS

Azael Zamora
San Jose State University
azael.zamora@sjsu.edu

The objective of this report, is to focus on the basic arithmetic operations such as multiplying, subtracting, adding, and dividing using MIPS logical and normal procedures in MARS

I. Introduction

Utilizing the MARS simulator, we will be able to calculate basic arithmetic operations using both the MIPS basic instructions such as; add, sub, mult, and div, as well as logical operations such as the boolean logic AND, NOT, and OR. For the purpose of this project, we will be using MARS to simulate the MIPS assembly language to preform the basic arithmetic operations. The objectives for the project are shown below:

1. *Install and setup the simulator for MIPS called MARS*

2. *Implement and design the basic arithmetic operations using both boolean operation provided by MIPS logical operations and the basic mathematical operations provided by MIPS*

3. *Test both implementations using the MARS simulator downloaded*

The report will provide the steps on how to install the MIPS simulator, and will also discuss the implementation process that used for the arithmetic operations.

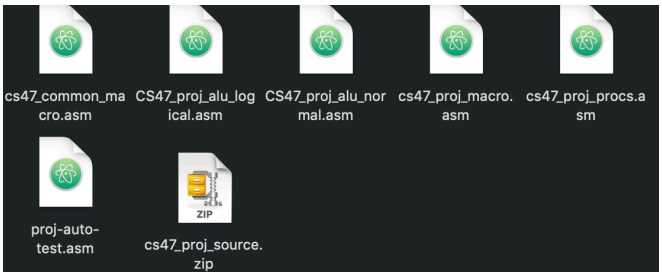
II. Setting up to use MIPS assembly language

1. Installing MARS

The MIPS simulator MARS is available for free by downloading it at the site: *https://www.softpedia.com/get/Programming/Coding-languages-Compilers/Vollmar-MARS.shtml*, and click on download, to start downloading it.

2. Downloading Project 1 to your computer

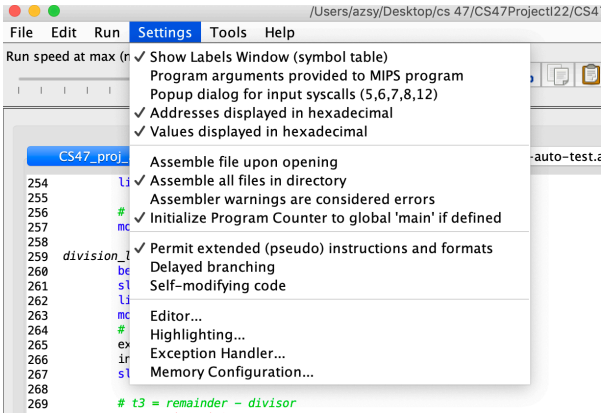
Download the zip file located on Canvas: https://sjsu.instructure.com/courses/1324477/assignments/5056985, and make sure to unzip the file. Once you unzip, there should be six files, ignore the zip file:



The files that should be included.

3. Settings for MIPS Simulator

To make it a bit more easier to debug the project, make sure to check the box to show line numbers, to help fix the lines the easier when testing and writing the program. On the top menu, go to "Settings" and be sure to turn on the following settings:



The settings that should be turned on for this project.

III. Requirements for the arithmetic operations

For the project, two different procedures are required to be created. As stated previously, the project requires the implementation of addition, subtraction, division, and multiplication, in two different ways, which need to implement procedures that are required to be utilized for this project.

A. Normal Procedure using MIPS basic Intructions

The procedure called au_normal, will consist the use of MIPS basic instructions to the calculate the mathematical expressions that are given in proj_auto_test.asm. The basic instructions that are required to use for the MIPS assembly language are; add, div, mult, and sub, in order to understand the instructions that include in MIPS. The procedure has the following arguments:

1. *Register $a0 (Argument #1)*

Register $a0 will hold the first number in the mathematical expression that will be calculated

2. *Register $a1 (Argument #2)*

This register will hold the second number in the mathematical expression that will be calculated.

3. Register $a2 (Argument #3)

This register will hold the operation code, for this I used the ASCII code representation of either; +, -, *, and /, to branch to the operation that will need to be performed.

Register $v0, will store the result of the sum of the addition operation or the difference of the subtraction operation. For both multiplication and division operations, registers $v0 and $v1 are required to be utilized. For the multiplication operation, $v0 will store LO portion of the product, while $v1 will store the HI portion of the product. In division, $v0 will store the quotient, while $v1 will store the remainder of the operation.

B. Logical Procedure using MIPS logical operations

This procedure called au_logical, is required to use multiple procedures in order to perform each basic arithmetic operation with only boolean operations. This includes boolean logic such as NOT, OR, AND, and also XOR. Basic MIPS instructions that were used in au_normal are prohibited when calculating in au_logical, only add is allowed to increment the index counter in the loops. The procedure will also utilize the three same arguments like in au_normal.

1. Register $a0 (Argument #1)

This register will hold the first number in the mathematical expression that will be calculated

2. Register $a1 (Argument #2)

This register will hold the second number in the mathematical expression that will be calculated.

3. Register $a2 (Argument #3)

This register will hold the operation code, for this I used the ASCII code representation of either; +, -, *, and /, to determine the operation that is needed to be performed.

Register $v0, will store the result of either the sum of the addition operation or the difference of the subtraction operation. For both multiplying and dividing, registers $v0 and $v1 will be required to used. For the multiplication operation, $v0 will store LO portion of the product, while $v1 will store the HI portion of the product. In division, $v0 will store the quotient, while the remainder will be contained in register $v1 for the operation. While au_logical will use the same registers as au_normal, au_logical will require the use of additional procedures in order to calculate the results similar to au_normal.

IV. DESIGNING AND IMPLEMENTING

A. au_normal

The normal procedure of the project was completed with a simple procedure called au_normal, which is capable of performing any of the four arithmetic operations. Based on the argument symbol that register $a2 holds, the corresponding MIPS instruction should be used to calculate the expression.

1. Operation code '-'

The '-' symbol references the use of subtraction, in which it is used to determine the difference the two numbers in registers $a0 and $a1. The 'sub' instruction in MIPS will be used to calculate the difference.

2. Operation code '+'

The '+' symbol references the use of addition, in which it is used to determine the sum between the numbers in registers $a0 and $a1. The 'add' instruction in MIPS will be used to calculate the sum.

3. Operation code '*'

The '*' symbol references the use of multiplication, in which it is used to determine the product of both the numbers in registers $a0 and $a1. The 'mult' instruction in MIPS will be used to calculate the product.

4. Operation code '/'

The '/' symbol reference the use of division, in which it is used to determine both the quotient and the remainder of the numbers in registers $a0 and $a1. The 'div' instruction in MIPS will be used to calculate both the quotient and the remainder.

```
16 ##################################################################
17 au_normal:
18         addi $sp, $sp, -24
19         sw $fp, 24($sp)
20         sw $ra, 20($sp)
21         sw $a2, 16($sp)
22         sw $a1, 12($sp)
23         sw $a0,  8($sp)
24         addi $fp, $sp, 24
25
26         li $t0, 42    # ascii value of *
27         beq $t0, $a2, multiply_au_normal        # if mult, jump to multiplication
28
29         li $t0, 43    # ascii value of +
30         beq $t0, $a2, add_au_normal             # if add, jump to addition
31
32         li $t0, 45    # ascii value of -
33         beq $t0, $a2, subtract_au_normal        # if sub, jump to subtracition
34
35         li $t0, 47    # ascii value of /
36         beq $t0, $a2, divide_au_normal          # if div, jump to division
37
38         j end_au_normal
39
```

Implementing branches and the operations.

B. au_logical

Similar to the au_normal procedure, the au_logical procedure will also use branches to determine the arithmetic operation that is needed to be performed. Unlike the au_normal procedure, this procedure will require the use of additional procedures as well as the use of logical operations to perform the arithmetic operations necessary.

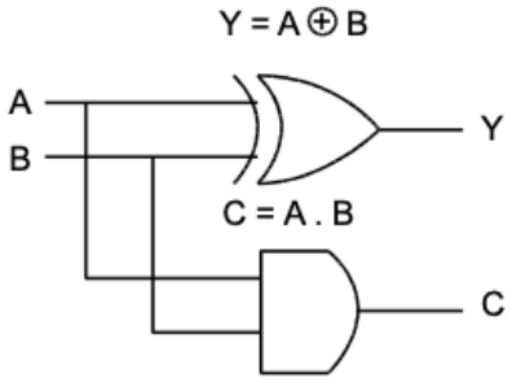
```
18         addi $sp, $sp, -24
19         sw $fp, 24($sp)
20         sw $ra, 20($sp)
21         sw $a2, 16($sp)
22         sw $a1, 12($sp)
23         sw $a0, 8($sp)
24         addi $fp, $sp, 24
25
26         li $t0, 42    # ascii value of *
27         beq $t0, $a2, mul_signed                # if mult, jump to multiplication
28
29         li $t0, 43    # ascii value of +
30         beq $t0, $a2, add_logical               # if add, jump to addition
31
32         li $t0, 45    # ascii value of -
33         beq $t0, $a2, sub_logical               # if sub, jump to subtracition
34
35         li $t0, 47    # ascii value of /
36         beq $t0, $a2, div_signed                # if div, jump to division
37
```

Implementing branches and the operations

1. add_sub

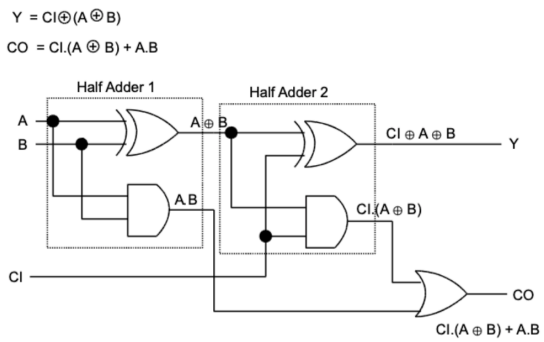
The procedure will calculate the sum of the numbers in registers $a0 and $a1. The register $a2 will be used in add_logical and sub_logical to determine if whether adding or subtracting will be used. The register $a2 will be used to branch into the operation that should be done by the symbol that is currently stored. If $a2 has the symbol '+', then addition will be performed, and the sum in this case will be the addition of both numbers. If $a2 has the symbol '-', then subtracting will be performed. If it is subtraction, then $a1 will be converted to its two's complement form, and sum should be $a0 + ~$a1. Two half adders are required to do the addition of both $a0 and $a1

The half adder can only do single bit addition and only considers the carry-out bit. It should be noted that using AND operation can determine the carry-out bit, while the sum is determined by utilizing the XOR boolean operation.

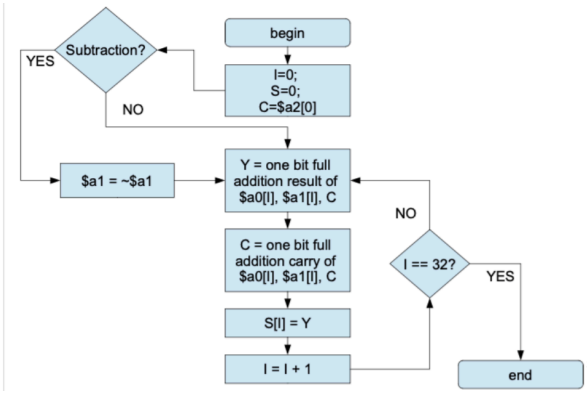


Implementation of the Half Adder

In order to take into account both carry-in and carry-out bits in one-bit addition, a full adder must be used. A full adder is made from two half adders. The sum of single bit addition is done by using the XOR boolean operation on the carry-in-bit, which requires using both the 1st bit of both operands ($a0, and $a1). The last carry-out bit of the add_sub is the result from using OR between both the carry-out bits using the full adder. It should be noted that most of the Boolean operations are accomplished by using a single half adder.



Implementation of the Full Adder



The loop for Addition/Subtraction

To implement the addition/subtraction operation, a loop will be required in order to run 32 times, since both $a0 and $a1 are 32-bit registers. If it is addition, it will branch to add_logical in which it proceed to do the addition between then two arguments. If it is subtraction, it will then branch to sub_logical in which $a1 is inverted into its two's complement form, and will then proceed to jump into the loop to calculate the sum of $a0 and ~$a1.

```
extract_0th_bit($t0, $a0)          # t0 = a0.first_bit
extract_0th_bit($t1, $a1)          # t1 = a1.first_bit

full_adder($t0, $t1, $t4, $t3, $t5, $t6) # full adder: $t4 = bit answer, $t6
insert_to_nth_bit($s0, $t2, $t4, $t5)    # $s0[$t2] = $t4

move    $t5, $t6                   # $t5 (carry in) = $t6 (carry out)
addi    $t2, $t2, 1                # t2++

beq     $t2, 32, add_sub_end       # if t$2 == 32 end the loop

j       add_sub                    # loop repeats if $t2 != 32
```

Implementation of the add_sub loop

2. add_logical

The procedure add_logical will utilize the procedure add_sub_logical to add $a0 and $a1, since the register $a2 will hold the symbol '+'.

```
gical:
  addi    $sp, $sp, -24
  sw      $fp, 24($sp)
  sw      $ra, 20($sp)
  sw      $s0, 16($sp)
  sw      $a1, 12($sp)
  sw      $a0, 8($sp)
  addi    $fp, $sp 24
  add     $t0, $t0, $zero          # t0 = hold $a0 bit
  add     $t1, $t1, $zero          # t1 = hold $a1 bit
  add     $t2, $t2, $zero          # position index
  add     $t3, $t3, $zero          # carry holder for full adder
  add     $t4, $t4, $zero          # bit answer holder for full adder
  add     $t5, $t5, $zero          # carry in for full adder
  add     $s0, $zero, $zero        # is for saving the final outcome
```

Implementation of add_logical

3. sub_logical

The procedure sub_logical will jump to add_sub_logical to do subtraction, but will first need to use MIPS instruction 'neg' to set the number in $a1 into its two's complement form. The procedure is used since the register $a2 will hold the symbol '-'.

```
ical:
  addi    $sp, $sp, -20
  sw      $fp, 20($sp)
  sw      $ra, 16($sp)
  sw      $a1, 12($sp)
  sw      $a0, 8($sp)
  addi    $fp, $sp 20
  neg     $a1, $a1        # $a1 = ~$a1, neg makes $a2 into its complement form
  jal     add_logical     # $a0 + ~$a1
  lw      $fp, 20($sp)
  lw      $ra, 16($sp)
  lw      $a1, 12($sp)
  lw      $a0, 8($sp)
  add     $sp, $sp, 20
  jr      $ra
```

Implementing the sub_logical procedure

4. twos_complement

The procedure twos_complement is used to convert $a0 into its two's complement form. The argument $a0 will be used, and will require the use of NOT to do bitwise inversion of the number in $a0, and will also require $a1 to store the number 1. The procedure add_sub will be used to calculate the two's complement form of $a0 by calculating the sum of $a0 and $a1 in twos_complement procedure.

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

    not     $a0, $a0        # $a0 = ~$a0
    li      $a1, 1          # $a1 = 1
    jal     add_sub         # add_sub will be used to calculate ~$a0

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

Implementation of twos_complement

5. twos_complement_64bit

The procedure twos_complement_64bit is utilized to invert a 64-bit number into its two's complement form using two 32-bit registers. Register $a0 will contain the LO of the number, while register $a1 will contain the HI of the product. To start, invert both $a0 and $a1 to their two's complement form. Then use add_sub to add $a0 with 1, and then use add_sub to add the carry out ($s0) with register $a1. Register $v0 should will store the LO number in two's complement form, while register $v1 should return the HI number in two's complement form.

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

        not     $a0, $a0        # invert a0
        not     $a1, $a1        # invert a1
        move    $s0, $a1        # s0 = a1
        li      $a1, 1
        jal     add_logical
        move    $a0, $v1        # set a0 to the overflow bit
        move    $a1, $s0        # a1 = s0
        move    $s1, $v0        # save Lo part of 64bit 2'complement in s1

        jal     add_logical
        move    $v1, $v0        # move Hi part of 64bit 2'complement in v1
        move    $v0, $s1

        lw      $s1, 28($sp)
        lw      $s0, 24($sp)
        lw      $a1, 20($sp)
        lw      $a0, 16($sp)
        lw      $ra, 12($sp)
        lw      $fp, 8($sp)
        addi    $sp, $sp, 28
        jr      $ra
```

MIPS implementation of twos_complement_64bit

6. bit_replicator

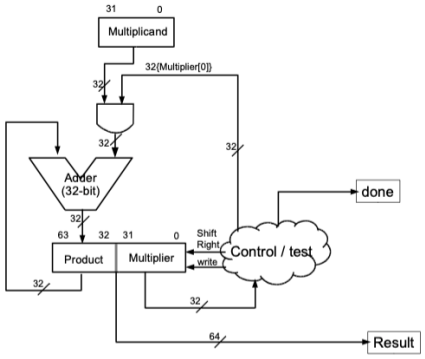
This procedure will replicate a bit that is stored in $a0 approximately 32 times in order to have a 32-bit value. This procedure will be used when calculating the product between both numbers in $a0 and $a1, when creating the multiplication operation. For this procedure, $a0 will be the argument in which it will store either a 0 or 1, and will return a 32-bit number that will consist of the value in $a0 replicated 32 times.

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

        extract_0th_bit($t0, $a0)
        beqz    $t0, end_bitreplicator  # if t0 = 0 t0 already the answer

        srl     $t0, $t0, 1     # know t1 should be all 0s
        move    $a0, $t0        # a0 = t0
        jal     twos_complement # $v0 is now all 1s

        end_bitreplicator:
        move    $v0, $t0

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

MIPS implementation of bit_replicator

7. mul_unsigned

The procedure will perform multiplication with registers $a0 and $a1, the arguments while not taking into account the signs of both arguments. While both the arguments $a0 and $a1 will values of up to 32-bits, the result will be in a 64-bit answer due to the fact that multiplication is a 64-bit operation.

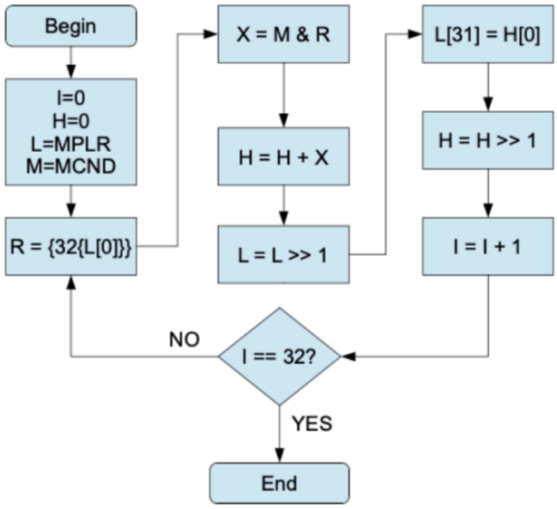


Logical design of Unsigned Multiplication

```
_mul_loop:
  # check for LSB of MCND
  extract_0th_bit($t0, $s5)
  beqz    $t0, shift_MCND_by_1
  # add product and MCND in 64 bit
  move    $a0, $s0
  move    $a1, $s1
  move    $a2, $s2
  move    $a3, $s3

  jal     bit64_adder

  move    $s0, $v0                    # $s0 = $v0
  move    $s1, $v1                    # $s1 = $v1

  beq     $s4, 32, end_unsign_mul_loop    # if $s4 == 32, exit
```

Implementation of the unsigned Multiplication Loop

In order to do multiplication, addition is required to do the operation. As the process of multiplication goes on, the multiplier noted by as $a1 will be shift to the right, to show that the individual bit position multiplication is finished. This process will continue, once the multiplier has shifted to the right 32 times, and the product is able to fill both registers $v0 and $v1.

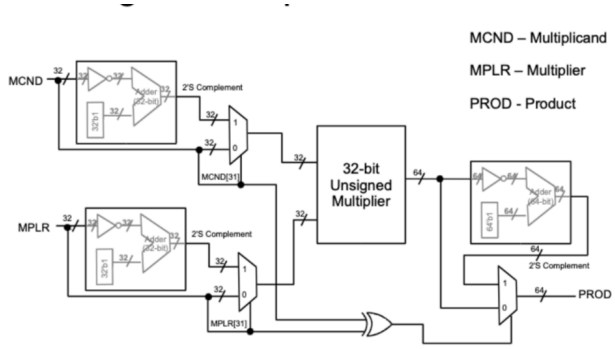


Logical Loop design for Multiplication

In order to replicate the multiplication process as shown in the picture, a loop will required to use in order to perform the necessary operations 32 times. To perform the '64-bit' multiplication process, the multiplier stored in register $a1 must be shifted to the right as the loop increments, while the 0th bit position in the product register will be inserted into the most significant bit position in the multiplier register. After, the product register will shift to the right.

8. mul_signed

Similar to mul_unsigned, the procedure will also handle multiplication, but will take both signed or unsigned numbers into consideration unlike the mul_unsigned procedure. The arguments will $a0 which will store the multiplicand, and $a1 which will store the multiplier. Operand that negative will be converted to positive using twos_complement procedure, but the sign will be remembered for later on.



Logical Design of Multiplication Signed

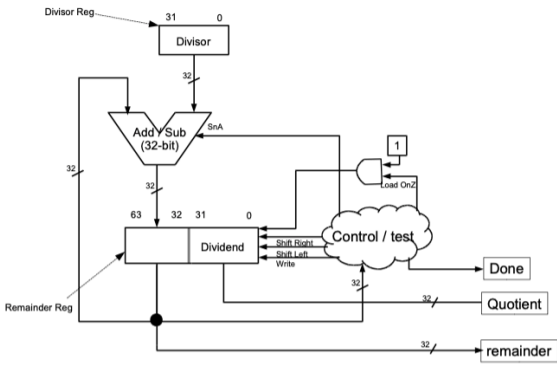
After utilizing mul_unsigned to perform the multiplication, the XOR operation will be used to check whether the product will either be a positive or negative number. The purpose of the using XOR is to check whether two positive or two negative numbers were used which leads to a positive product, or if one number was negative, then the product will be a negative number. If the XOR value is a 1, then twos_complement_64bit will be used to determine the two's complement form of the product.

```
mul_signed:
        addi    $sp, $sp, -32
        sw      $s6, 32($sp)
        sw      $a1, 28($sp)
        sw      $a0, 24($sp)
        sw      $s1, 20($sp)
        sw      $s0, 16($sp)
        sw      $ra, 12($sp)
        sw      $fp, 8($sp)
        addi    $fp, $sp, 32

        # get the MSD for both a0, a1
        move    $t1, $a0
        move    $t2, $a1
        li      $t3, 31
        extract_nth_bit($t0, $t1, $t3)
        extract_nth_bit($t4, $t2, $t3)
        # see if they are same sign or not
        # same -> positive, dif -> two's complement needed
        xor     $s6, $t0, $t4   # s6 save the result
        jal     mul_unsigned
        move    $s0, $v0        # Lo
        move    $s1, $v1        # Hi

        beqz    $s6, end_mul_signed
        move    $a0, $s0
        move    $a1, $s1
        jal     twos_complement_64bit
        move    $s0, $v0
        move    $s1, $v1

end_mul_signed:
        move    $v0, $s0
        move    $v1, $s1
```

Implementing Signed Multiplication in MIPS

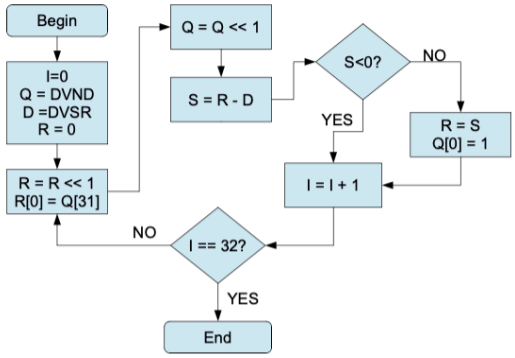
9. div_unsigned

Like the procedure mul_unsigned, the div_unsigned procedure will perform division without taking into consideration the signs. The procedure has two arguments, the dividend contained in $a0, and the divisor contained in $a1.



Logical Design of Unsigned Division

Division is a 64-bit arithmetic operation similar to multiplication, so both registers $a0 and $a1 are used in order to solve the 64-bit answer for division. The division operation requires the use of subtraction, since the dividend should be subtracted by the divisor, starting from the 31sr bit in the dividend. If the result from subtraction is a positive number, then the quotient will have a 1 in the current bit position, since the quotient will be the result. If the result from subtraction leads to a negative result, a 0 will be stored in the corresponding bit index in the quotient. The division operation will continue while the dividend is shifted to the left, while being replaced by the quotient.



Design of the Unsigned Division Loop

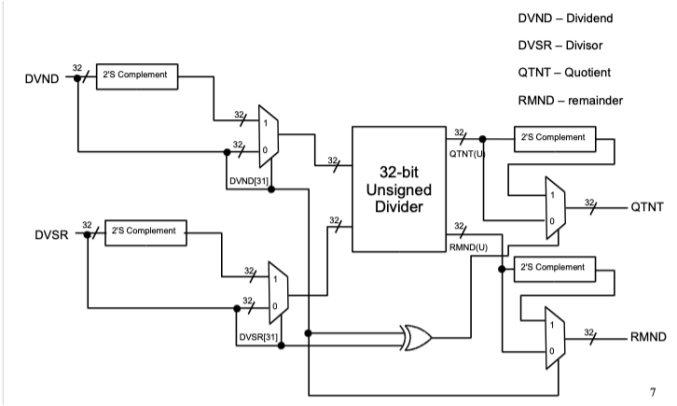
In order to replicate chart as shown in the picture, a for loop is necessary in order to repeat the operations 32 times. For division, the contents in the register containing the remainder $s3, will need to be shifted to the left, to allow the 31st bit of the register containing the quotient to be inserted at the 0th position of the remainder register. After, the quotient should shifted to the left, it should be noted that both the quotient and dividend are stored in the same register.

```
n_loop:
 beq     $s4, 32, end_division
 sll     $s3, $s3, 1     # remainder shift left by 1
 li      $t1, 31         # t1 = 31
 move    $t2, $s2        # t2 = s2
 # get the MSD from dividend insert it at LSD of remainder
 extract_nth_bit($t0, $t2, $t1)
 insert_to_nth_bit($s3, $zero, $t0, $t3)
 sll     $s2, $s2, 1     # shift dividend left by 1

 # t3 = remainder - divisor
 move    $a0, $s3
 move    $a1, $s1
 jal     sub_logical
 move    $t3, $v0

 bltz    $t3, not_large_enough
 # remainder = t3, remainder larger than divisor -> no longer remainder
 move    $s3, $t3
 # insert one to the quotation at the MSD(when finished)
 li      $t0, 1          # t0 = 1
 insert_to_nth_bit($s2, $zero, $t0, $t2)
```

Implementing the Unsigned Division in MIPS

10. div_signed

Similar to the mul_signed procedure, div_signed will do division, while taking the signs into consideration. The steps required to do signed division, is somewhat similar to the steps required in mul_signed.



Design implementing the logical operations of Signed Division

The numbers in $a0 and $a1will first be compared to see if they are positive or negative, and will be converted into their two's complement form if either one is negative. It will then call upon div_unsigned to do the division operation required, where $v0 will store the quotient, while $v1 will store the remainder. Afterwards the original signs of both the dividend and the divisor will be used to determine the signs of both the remainder and the quotient. Similar to mul_signed, if both $a0 and $a1 were positive or negative, then both $v0 and $v1 are positive numbers, else if one is a negative sign, then $v0 and $v1 are negative numbers. Using XOR, can help determine whether or not the quotient needs to be in its complement form, similar to how mul_signed used XOR.

```
        move    $t1, $a0
        move    $t2, $a1
        li      $t3, 31
        extract_nth_bit($s3, $t1, $t3)
        extract_nth_bit($s4, $t2, $t3)
        # see if they are same sign or not
        # same -> positive, dif -> two's complement needed
        xor     $s0, $s3, $s4   # s0 save the result

        jal     div_unsigned
        move    $s1, $v0        # quotient
        move    $s2, $v1        # remainder

        beqz    $s0, check_remainder_sign
        move    $a0, $s1
        jal     twos_complement
        move    $s1, $v0        # 2'c the quo then s1 = v0

check_remainder_sign:
        beqz    $s3, end_div_signed
        move    $a0, $s2
        jal     twos_complement
        move    $s2, $v0        # 2'c the rem then s2 = v0

end_div_signed:
        move    $v0, $s1        # v0 = quotient
        move    $v1, $s2        # v1 = remainder
```

Implementing the Signed Division in MIPS

V. TESTING

To test au_normal and au_logical procedures, first all the files must be saved. Proceed to assemble the proj_auto_test.asm, note that the file should not be modified in anyway, and run the MIPS program by pressing the "Run" button located within the icon menu. Both au_normal and au_logical procedures were created to match their output, since both should yield identical answers. If the procedures correctly compile, then the test cases of 40 mathematical problems by both the au_normal and au_logical will pass with a score of 40/40, with a message ending in "*** OVERALL RESULT PASS ***".

```
(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 --
```

This is what the output should look like when running the file

VI. Conclusion

To summarize, this project has made me realize how much work a computer is required to do for a simple calculation, as demonstrated in the both the au_logical procedure. I learned a lot on how to use MIPS operations, and how to be able to write logical procedures for basic arithmetic operations. This project was an interesting approach on learning how to apply the required procedures in order to implement the logical operations, as opposed to simply using the basic MIPS instructions to add, divide, multiply or subtract. In the end, this project was interesting to do, as well as really good learning experience towards the logical approach for the basic arithmetic operations we use on a regular basis.

References

1. Kaushik Patra. CS 47. Lecture, Topic: “Addition Subtraction Logic.” San Jose State University, San Jose, CA, November 13, 2019
2. Kaushik Patra. CS 47. Lecture, Topic: “Multiplication Logic.” San Jose State University, San Jose, CA, November 18, 2019
3. Kaushik Patra. CS 47. Lecture, Topic: “Division Subtraction Logic.” San Jose State University, San Jose, CA, November 20, 2019