Basic Calculator Implementation Through MIPS Logical Operations

John Paul Tran
San Jose State University
johnpaul.tran@sjsu.edu

Abstract--**This report includes the explanation and detail behind how basic calculator functions are implemented by logical operations in MIPS. The four basic math operations (add, subtract, multiply, and divide) are implemented through normal and logical procedures.**

I. Introduction

The focus of this project is to execute addition, subtraction, multiplication, and division through two procedures in MIPS assembly language. To accomplish our goals, we will be using MIPS Assembler and Runtime Simulator (MARS). Normal procedures will be done through built-in MIPS operations (add, sub, mult, etc). Logical procedures will be done with logical operations (and, or, xor, etc). The steps are as follows:

1. Download and set up MARS software correctly
2. Implement normal operations
3. Implement logical operations
4. Test the results of logical operations by comparing them to the results of normal operations

II. Requirements

This part will allow us to correctly set up the conditions necessary to perform our tasks. This is essential for ensuring success.

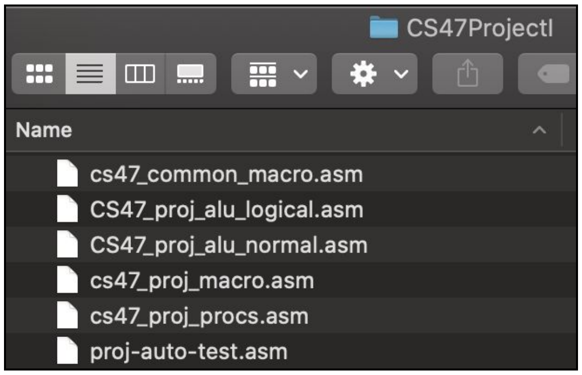
A. Download MARS

First off, MARS is the simulator we will use to program in MIPS assembly language. You can download it through Missouri State University at the following link: https://courses.missouristate.edu/KenVollmar/MARS/download.htm

- A prerequisite of downloading MARS is to have Java J2SE 1.5 (or later) SDK installed.
- If you do not have this, you can download the latest version at: https://www.oracle.com/technetwork/java/javase/downloads/index.html

B. Download Project Files

To locate the project files, you must have an account on Canvas and be enrolled in Professor Patra’s CS 47 class. Once logged into Canvas, click on the course tab and navigate to the assignments page. There, you will find the ‘Project 1’ assignment that contains the zipped file ***CS47Project1.zip***. Download and unzip it to see the following six files:

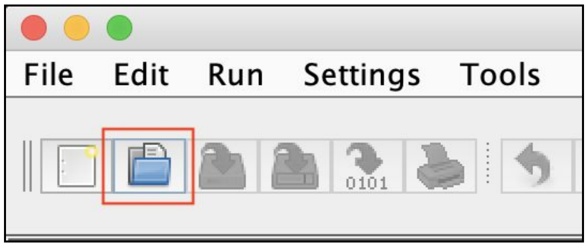


cs47_common_macro.asm
CS47_proj_alu_logical.asm
CS47_proj_alu_normal.asm
cs47_proj_macro.asm
cs47_proj_procs.asm
proj-auto-test.asm

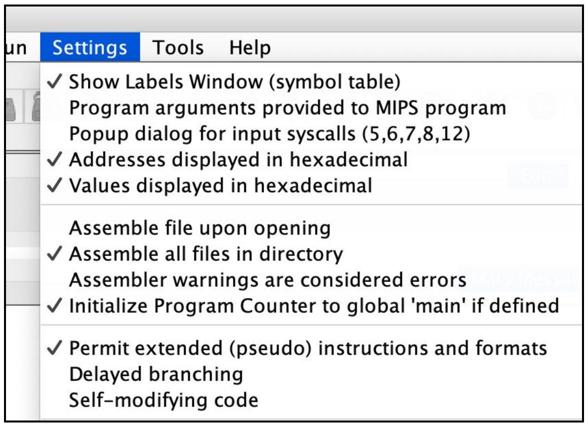
1. *Cs47_common_macro.asm*
   a. This file has the macros to print the results.
2. *Cs47_proj_alu_logical.asm*
   a. This file will contain the logical procedures for basic calculator functions.
3. *Cs47_proj_alu_normal.asm*
   a. This file will contain the normal procedures for basic calculator functions.
4. *Cs47_proj_macro.asm*
   a. This file will contain our own macros to help us with logical implementation.
5. *Cs47_proj_procs.asm*
   a. This file will contain the project procedures.
6. *Proj-auto-test.asm*
   a. This file will be used to compare the normal results with the logical results.

C. *Setting Up Mars*

Once you have the software and necessary files downloaded, go ahead and launch MARS. Upon opening, click the blue folder on the upper left to find your unzipped ***CS47Project1*** and open the six files.



Before beginning, be sure to check if you have the right settings. Click on settings at the bar at the top and you should see the following:



✓ 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

Make sure that these two are checked:

- Assemble all files in directory
- Initialize Program Counter to global ‘main’ if defined

By assembling all files in the directory, all the files will link together. This allows for the tester to still work even if all the needed files are not open. Initializing the program counter to global ‘main’ would allow for the program to begin at certain points defined by a main rather than just the beginning. It is now ready to start.

III. MIPS Registers

Unlike Java or C, MIPS uses registers for input and output storage. Each register can hold up to a maximum of 32 bits. When an operation result exceeds the 32 bit limit, we would have to use separate registers to store the results. The result could be split into two registers called $Lo and $Hi. The $Lo register will hold the lower 32 bit half of the result, and the $Hi register will hold the upper 32 bit half of the result.

A. Arguments

Both normal and logical procedures take the following three arguments:

1. *Register $a0* - the first operand in our arithmetic operation.
2. *Register $a1* - the second operand in our arithmetic operation.
3. *Register $a2* - the operation symbol or opcode in our arithmetic operation
    a. Based on basic math operations
    b. Addition (+), subtraction (-), multiplication (*), and division (/)

B. Results

After performing our arithmetic operations, the results will be stored in the following registers:

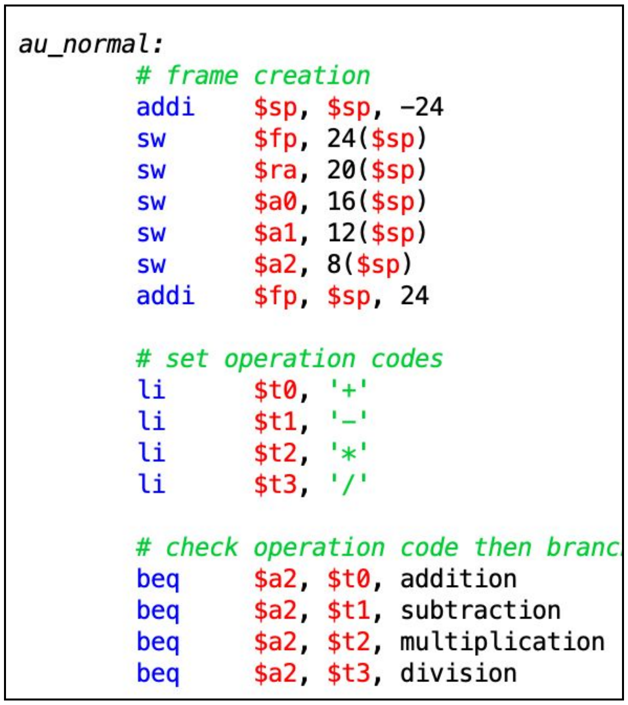
1. *Register $v0*
    a. It will hold the result of the sum or difference between $a0 and $a1.
    b. It will hold the Lo 32 bits of the result of $a0 * $a1.
    c. It will hold the quotient of $a0 / $a1.
2. *Register $v1*
    a. It will hold the Hi 32 bits of the result of $a0 * $a1.
    b. It will hold the remainder of $a0 / $a1.

IV. Design and Implementation

Once again, our main goal is to implement basic calculator functions in two different ways. Since the normal procedure will already have built-in MIPS functions for arithmetic operations, we will be using it as a comparison to the logical procedure. The logical procedure will use be implemented differently with logical operations (AND, XOR, and OR).

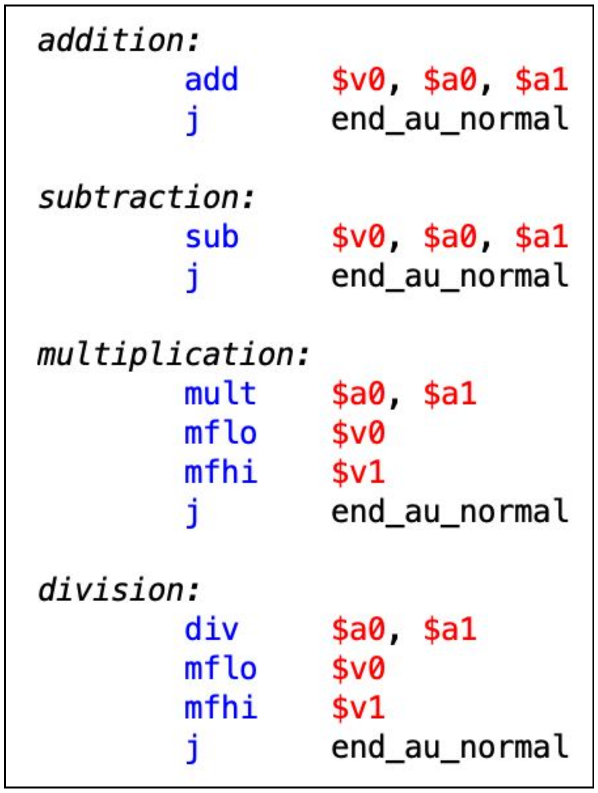
A. Normal Procedure

For starters, we have to create a frame. The frame allows space for push and pop operations on the stack. When $a2 matches with an operation code, we branch to the label of that specific operation to carry out the task at hand.



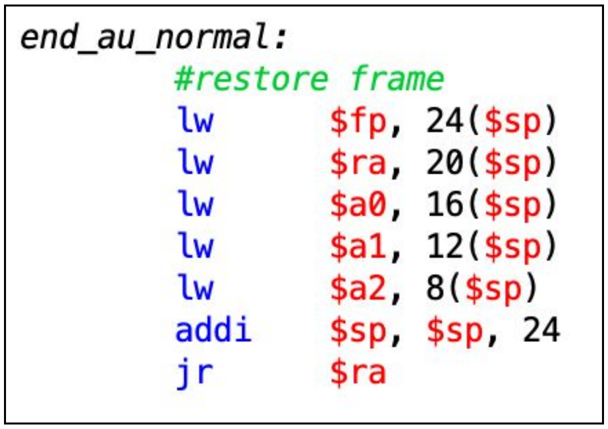
```
au_normal:
        # frame creation
        addi    $sp, $sp, -24
        sw      $fp, 24($sp)
        sw      $ra, 20($sp)
        sw      $a0, 16($sp)
        sw      $a1, 12($sp)
        sw      $a2, 8($sp)
        addi    $fp, $sp, 24

        # set operation codes
        li      $t0, '+'
        li      $t1, '-'
        li      $t2, '*'
        li      $t3, '/'

        # check operation code then branc
        beq     $a2, $t0, addition
        beq     $a2, $t1, subtraction
        beq     $a2, $t2, multiplication
        beq     $a2, $t3, division
```

For example, if there is a '-' in $a2, then it will branch to the *subtraction* label and carry out said function there. The same applies for all the other operation symbols.



```
addition:
        add     $v0, $a0, $a1
        j       end_au_normal

subtraction:
        sub     $v0, $a0, $a1
        j       end_au_normal

multiplication:
        mult    $a0, $a1
        mflo    $v0
        mfhi    $v1
        j       end_au_normal

division:
        div     $a0, $a1
        mflo    $v0
        mfhi    $v1
        j       end_au_normal
```

Since the functions are already defined in MIPS, you just have to call upon them to execute in each branch. Make sure you move the Lo and Hi bits to the right registers when doing multiplication and division. After a function is completed, caller will jump to the end label.



```
end_au_normal:
        #restore frame
        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
```

Finally, in the end label, we have to restore the frame in order to free up space in the stack.

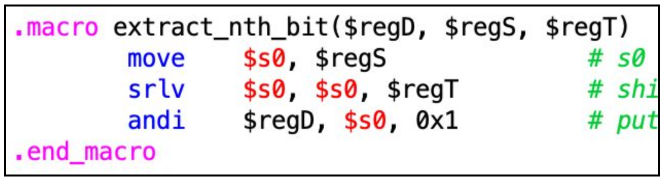
B. *Utility Macros & Other Procedures*

Before we begin the logical procedure, let's look at some utility macros and procedures that will help us along the way.

1. *extract_nth_bit*

This macro is intended to take out the nth bit (bit at nth position) of a source register. Once extracted, take that nth bit and place it in another register. It has three arguments:

- $regD: the resulting bit of 0 or 1
- $regS: the source of bit to extract
- $regT: the index

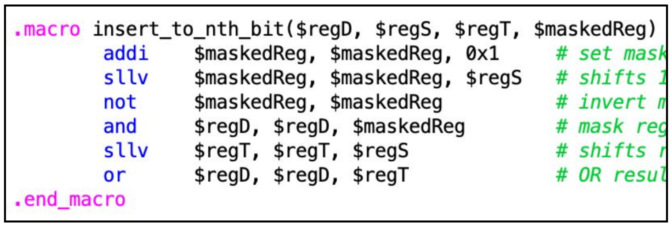


```
.macro extract_nth_bit($regD, $regS, $regT)
        move    $s0, $regS              # s0
        srlv    $s0, $s0, $regT         # shi
        andi    $regD, $s0, 0x1         # put
.end_macro
```

2. *insert_to_nth_bit*

This macro is intended to insert a certain bit (0 or 1) into another register from a source register. It has four arguments:

- $regD: the register of bit to insert
- $regS: the source of number to insert
- $regT: the index
- $maskedReg: the temporary register to mask and help shift

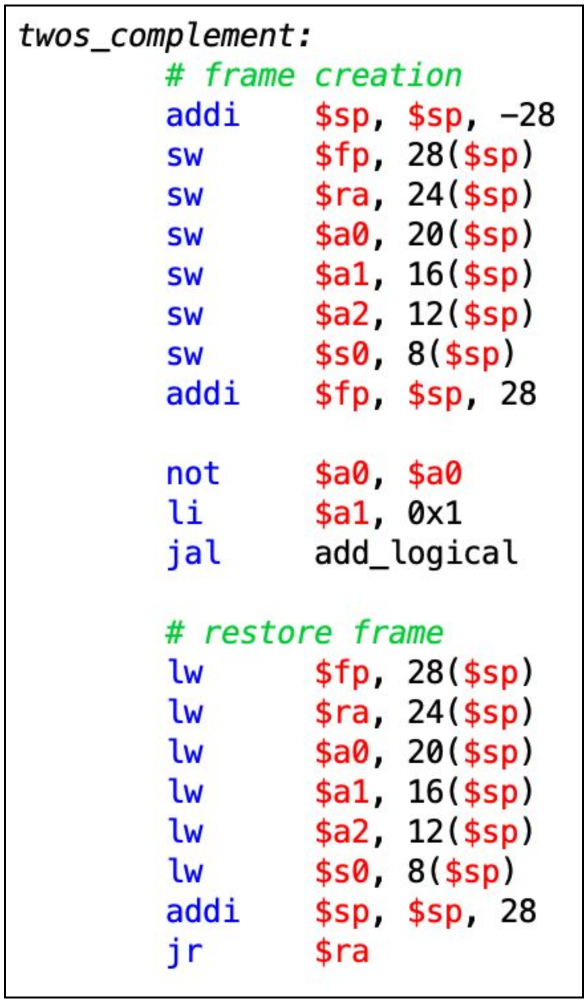


```
.macro insert_to_nth_bit($regD, $regS, $regT, $maskedReg)
        addi    $maskedReg, $maskedReg, 0x1     # set mask
        sllv    $maskedReg, $maskedReg, $regS   # shifts 1
        not     $maskedReg, $maskedReg          # invert m
        and     $regD, $regD, $maskedReg        # mask reg
        sllv    $regT, $regT, $regS             # shifts r
        or      $regD, $regD, $regT             # OR resul
.end_macro
```

3. *twos_complement*

This procedure is intended to convert from negative to positive or positive to negative. It starts with input $a0 and then inverts it. After inversion, add 1 to gain the two's complement.

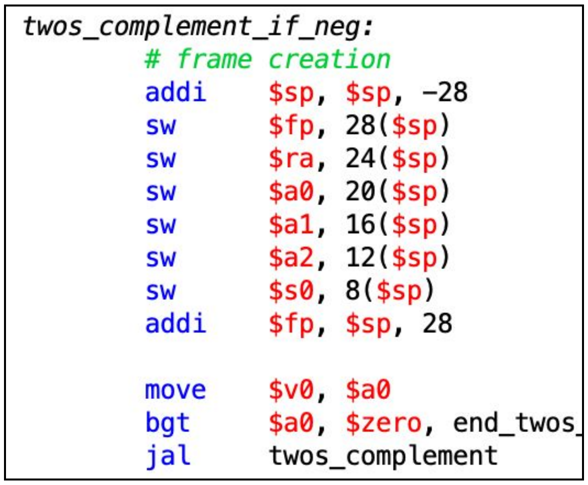
- ~$a0 + 1



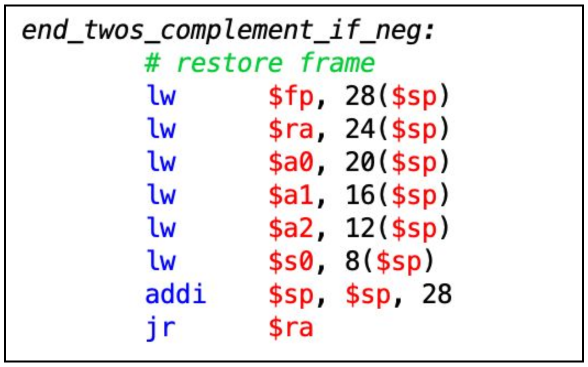
```
twos_complement:
        # frame creation
        addi    $sp, $sp, -28
        sw      $fp, 28($sp)
        sw      $ra, 24($sp)
        sw      $a0, 20($sp)
        sw      $a1, 16($sp)
        sw      $a2, 12($sp)
        sw      $s0, 8($sp)
        addi    $fp, $sp, 28

        not     $a0, $a0
        li      $a1, 0x1
        jal     add_logical

        # restore frame
        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
```

4. *twos_complement_if_neg*

This procedure is intended to check if the integer is negative. If it is greater than zero, you end the procedure. If it is less than zero, then we call upon the previous procedure.



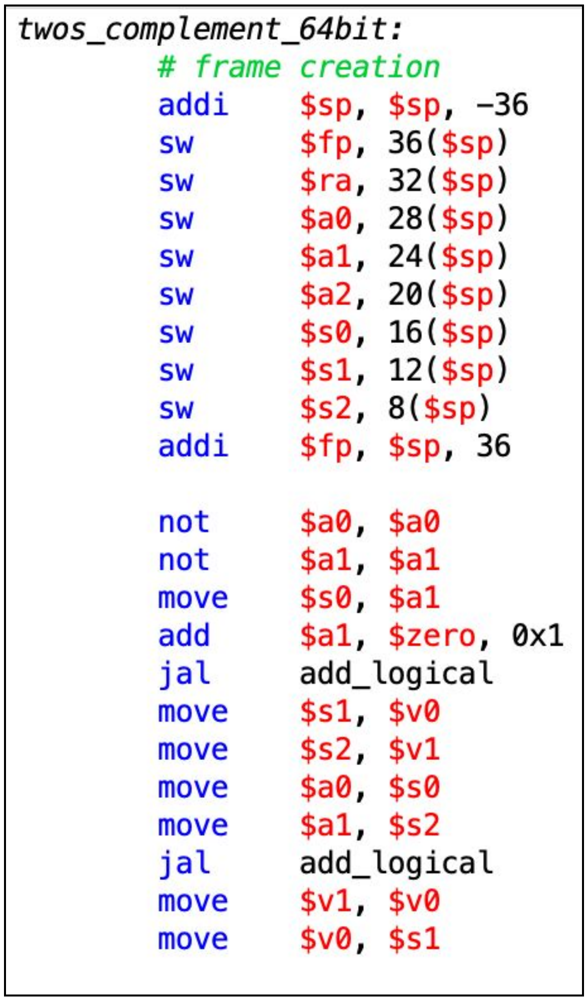
```
twos_complement_if_neg:
        # frame creation
        addi    $sp, $sp, -28
        sw      $fp, 28($sp)
        sw      $ra, 24($sp)
        sw      $a0, 20($sp)
        sw      $a1, 16($sp)
        sw      $a2, 12($sp)
        sw      $s0, 8($sp)
        addi    $fp, $sp, 28

        move    $v0, $a0
        bgt     $a0, $zero, end_twos_
        jal     twos_complement
```



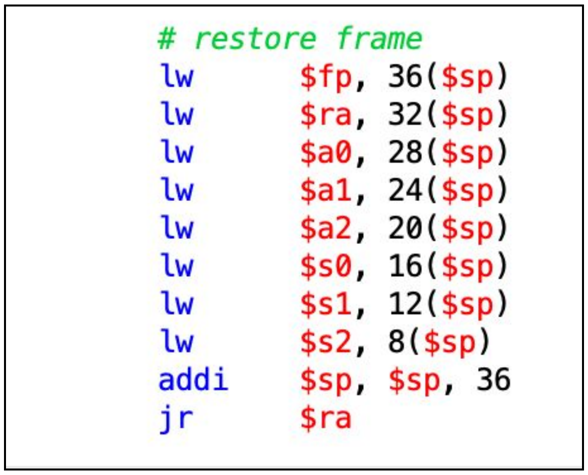
```
end_twos_complement_if_neg:
        # restore frame
        lw      $fp, 28($sp)
        lw      $ra, 24($sp)
        lw      $a0, 20($sp)
        lw      $a1, 16($sp)
        lw      $a2, 12($sp)
        lw      $s0, 8($sp)
        addi    $sp, $sp, 28
        jr      $ra
```

5. *twos_complement_64bit*

This procedure is intended for 64 bit representation of a two's complement integer. This is essential for multiplication and division because they both use the $Lo and $Hi registers. $Lo result is stored in $v0 while $Hi result is in $v1.



```
twos_complement_64bit:
        # frame creation
        addi    $sp, $sp, -36
        sw      $fp, 36($sp)
        sw      $ra, 32($sp)
        sw      $a0, 28($sp)
        sw      $a1, 24($sp)
        sw      $a2, 20($sp)
        sw      $s0, 16($sp)
        sw      $s1, 12($sp)
        sw      $s2, 8($sp)
        addi    $fp, $sp, 36

        not     $a0, $a0
        not     $a1, $a1
        move    $s0, $a1
        add     $a1, $zero, 0x1
        jal     add_logical
        move    $s1, $v0
        move    $s2, $v1
        move    $a0, $s0
        move    $a1, $s2
        jal     add_logical
        move    $v1, $v0
        move    $v0, $s1
```

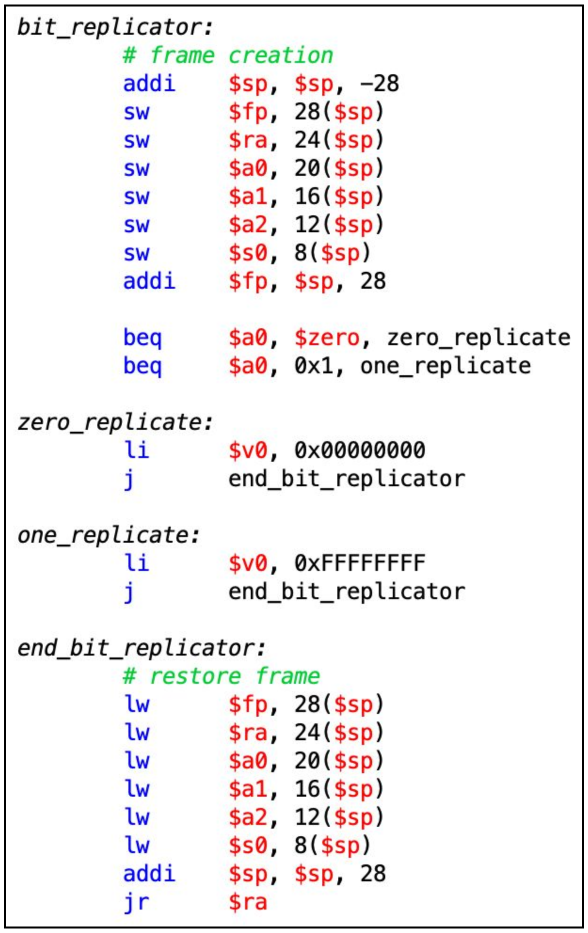


```
# restore frame
lw      $fp, 36($sp)
lw      $ra, 32($sp)
lw      $a0, 28($sp)
lw      $a1, 24($sp)
lw      $a2, 20($sp)
lw      $s0, 16($sp)
lw      $s1, 12($sp)
lw      $s2, 8($sp)
addi    $sp, $sp, 36
jr      $ra
```

a. Twos_complement_64bit takes two arguments $a0 (Lo) and $a1 (Hi).
b. Invert both $a0 and $a1. Move $a1 into $s0 and set $a1 to 1 in order to execute ~$a0 + 1. We will use add_logical (explained later) rather than built in MIPS function.
c. Move result $v0 to $s1 and $v1 to $s2 to save the original value and carry value from the last addition operation temporarily.
d. Move $s1 and $s2 to $a0 and $a1 for the next addition step in add_logical.
e. Move the result, $v0, directly into $v1 to be the two's complement of the 32 $Hi bits.
f. Move $s1 into $v0 to be the two's complement of the 32 $Lo bits.

6. bit_replicator

This procedure is intended for multiplication. It replicates a bit 32 times in order to fill its register. It takes one operand $a0 (bit with a value of 0 or 1) and replicates it. If $a0 is 0, it branches to the label *zero_replicate* and the result $v0 is 0x00000000.

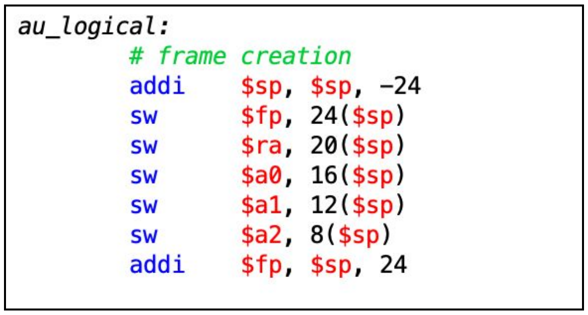


```
bit_replicator:
        # frame creation
        addi    $sp, $sp, -28
        sw      $fp, 28($sp)
        sw      $ra, 24($sp)
        sw      $a0, 20($sp)
        sw      $a1, 16($sp)
        sw      $a2, 12($sp)
        sw      $s0, 8($sp)
        addi    $fp, $sp, 28

        beq     $a0, $zero, zero_replicate
        beq     $a0, 0x1, one_replicate

zero_replicate:
        li      $v0, 0x00000000
        j       end_bit_replicator

one_replicate:
        li      $v0, 0xFFFFFFFF
        j       end_bit_replicator

end_bit_replicator:
        # restore frame
        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
```

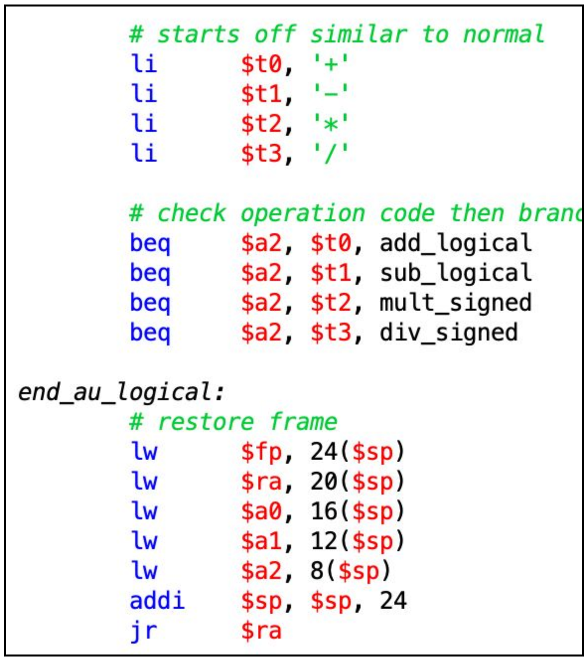
If $a0 is 1, then it branches to label *one_replicate* and the result $v0 is 0xFFFFFFFF.

C. Logical Procedures

Au_logical starts off very similar to au_normal except now it branches directly to the different logical operations based on their operation symbol. As always, we need to create a frame first.



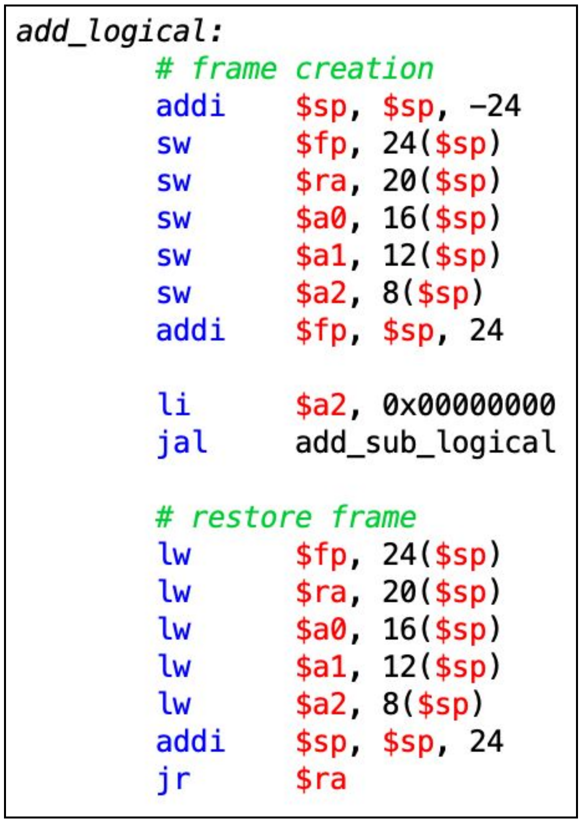
```
au_logical:
        # frame creation
        addi    $sp, $sp, -24
        sw      $fp, 24($sp)
        sw      $ra, 20($sp)
        sw      $a0, 16($sp)
        sw      $a1, 12($sp)
        sw      $a2, 8($sp)
        addi    $fp, $sp, 24
```



```
        # starts off similar to normal
        li      $t0, '+'
        li      $t1, '-'
        li      $t2, '*'
        li      $t3, '/'

        # check operation code then branc
        beq     $a2, $t0, add_logical
        beq     $a2, $t1, sub_logical
        beq     $a2, $t2, mult_signed
        beq     $a2, $t3, div_signed

end_au_logical:
        # restore frame
        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
```

1. *add_logical*

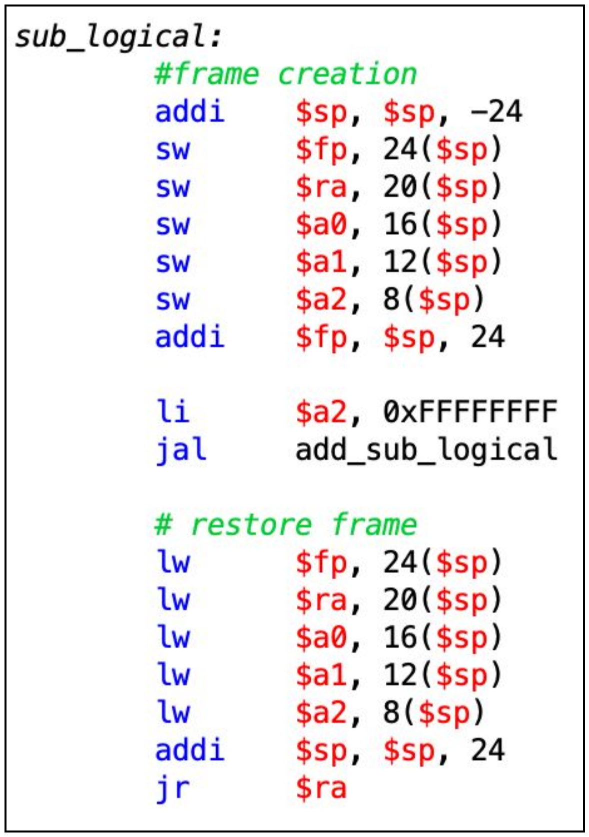
This procedure specifies the addition operation when $a2 is equal to 0x00000000. It calls to add sub logical to carry out addition.



```
add_logical:
        # frame creation
        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

        li      $a2, 0x00000000
        jal     add_sub_logical

        # restore frame
        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
```

2. *sub_logical*

This procedure specifies the subtraction operation when $a2 is equal to 0xFFFFFFFF. It calls to add_sub_logical to carry out subtraction.



```
sub_logical:
        #frame creation
        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

        li      $a2, 0xFFFFFFFF
        jal     add_sub_logical

        # restore frame
        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
```

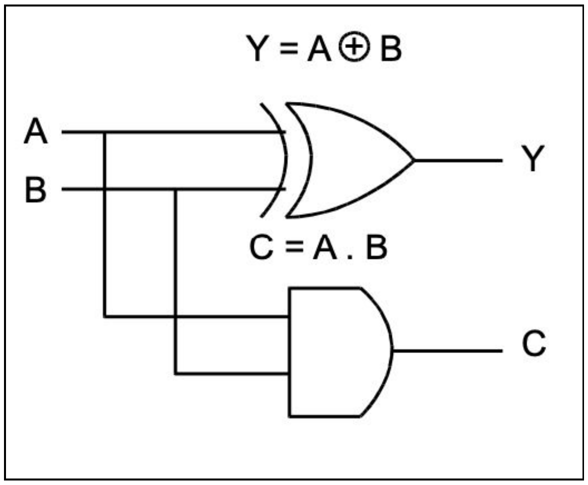
3. *add_sub_logical*

This procedure is intended to calculate 32 bit addition. Subtraction can be seen as addition when written as $a0 + (~$a1), so it is also addition in this case. It takes the following three operands:

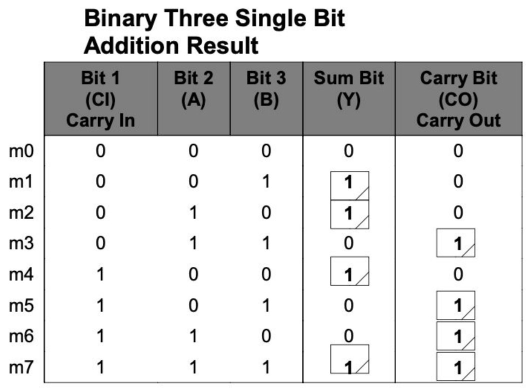
- $a0: the first number
- $a1: the second number
- $a2: the operation mode (either 0x00000000 for addition or 0xFFFFFFFF for subtraction)

Register $v0 will contain the result.
The addition operation begins with the half-adder. Due to the binary system, MIPS only allows for the addition of one bit of the

operand at a time while also keeping in mind the carry-out of the binary adding operations. The half adder, however, adds up two binary bits and stores the carry-out to be added to the next bit of the operand. A XOR operation determines the sum while an AND operation determines the carry-out bit.



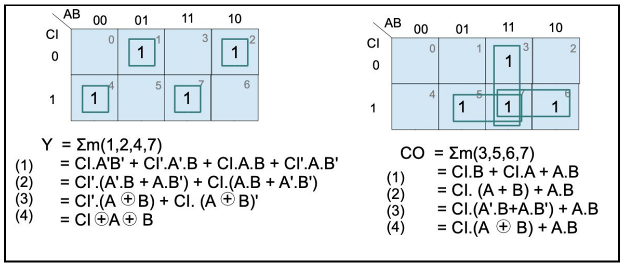
To add all of the operands up, we need to implement a full adder to do full number addition.


Binary Three Single Bit Addition Result

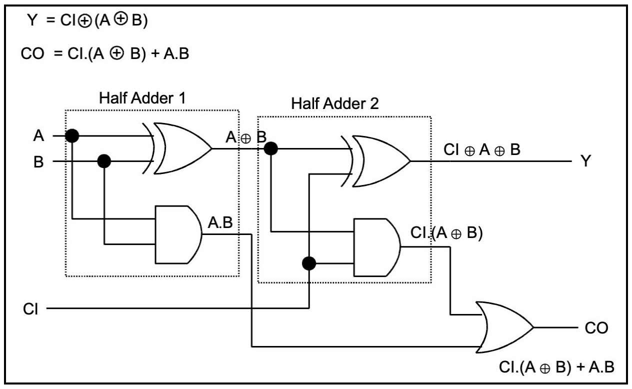
	Bit 1 (CI) Carry In	Bit 2 (A)	Bit 3 (B)	Sum Bit (Y)	Carry Bit (CO) Carry Out
m0	0	0	0	0	0
m1	0	0	1	1	0
m2	0	1	0	1	0
m3	0	1	1	0	1
m4	1	0	0	1	0
m5	1	0	1	0	1
m6	1	1	0	0	1
m7	1	1	1	1	1

Full Addition

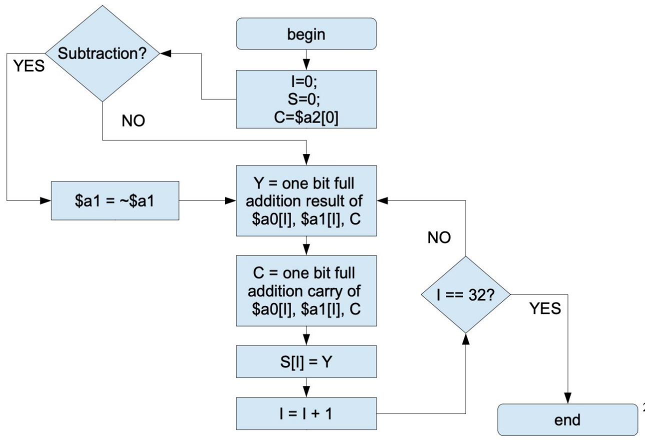
The first carry-out bit will need to become the carry-in bit for the next addition operation. Using the truth table for a full adder enables us to draw a Karnaugh map.


AB
CI
00
01
11
10
0
1
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

A K-map will allow us to see the full adder's logical design once we simplify the expressions.



Y (the new sum) is computed by the XOR of the following two digits and the carry-in bit. An OR operation between the carry-out bit from the two half adders will determine the final carry-out bit. Depending on the operation mode, $a1 may or may not be inverted. To fill out the register, the operation is looped 32 times.


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]

This diagram as well as the circuits is translated into the following code:

```
add_sub_logical:
        # frame creation
        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

        add     $t0, $zero, $zero
        extract_nth_bit($t1, $a2, $zero)
        add     $t2, $zero, $zero
        beq     $a2, 0xFFFFFFFF, subtraction_mode
        j       loop

subtraction_mode:
        not     $a1, $a1
        j       loop
loop:
        beq     $t0, 0x20, end_add_sub_logical
        extract_nth_bit($t3, $a0, $t0)
        extract_nth_bit($t4, $a1, $t0)
        # one bit full adder logical equation
        xor     $t5, $t3, $t4
        xor     $t6, $t5, $t1
        and     $t7, $t3, $t4
        and     $t1, $t1, $t5
        or      $t1, $t1, $t7
        insert_to_nth_bit($t2, $t0, $t6, $t8)
        addi    $t0, $t0, 0x1
        j       loop
```

```
end_add_sub_logical:
        move    $v0, $t2
        move    $v1, $t1
        # restore frame
        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
```

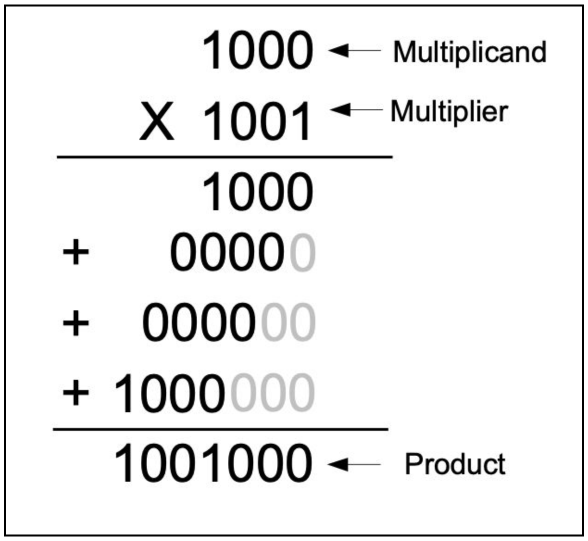
4. *mult_unsigned*

Binary multiplication is very similar to that of decimal numbers. Mult_unsigned is a 64 bit operation that has the following arguments:

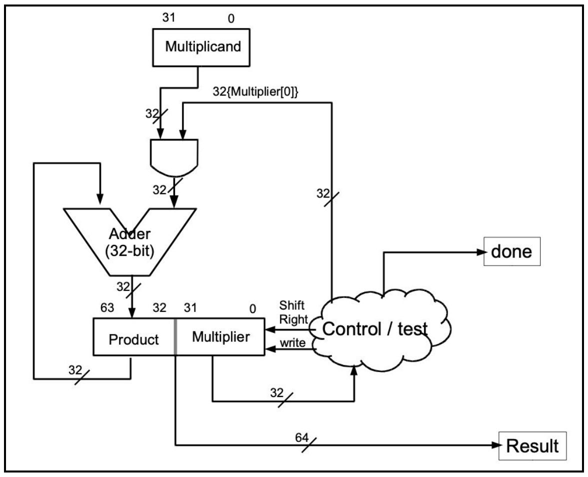
- $a0: the multiplicant
- $a1: the multiplier

Then it returns the following:

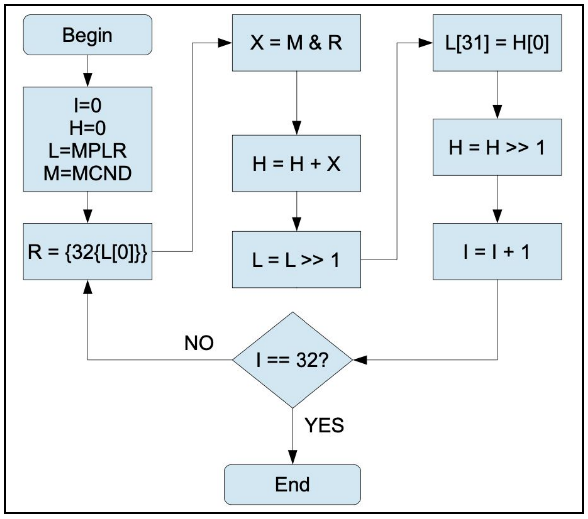
- $v0: the Lo 32 bits of the result
- $v1: the Hi 32 bits of the result



The final product can be found by an AND operation between the multiplicand and multiplier. We must then right shift the multiplier by 1 in order to AND the correct bits as seen in the paper example above. We must follow this format until each bit of the multiplicand and multiplier has been used. In our case, since $a0 and $a1 have 32 bits each, we would AND the multiplicand and multiplier's LSB bits 32 times.



The following flowchart shows the process of mul_unsigned:



The implementation is as follows:

```
mult_unsigned:
        # frame creation
        addi    $sp, $sp, -60
        sw      $fp, 60($sp)
        sw      $ra, 56($sp)
        sw      $a0, 52($sp)
        sw      $a1, 48($sp)
        sw      $a2, 44($sp)
        sw      $a3, 40($sp)
        sw      $s0, 36($sp)
        sw      $s1, 32($sp)
        sw      $s2, 28($sp)
        sw      $s3, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        addi    $fp, $sp, 60

        add     $s1, $zero, $zero
        add     $s2, $zero, $zero
        move    $s3, $a1
        move    $s4, $a0
```

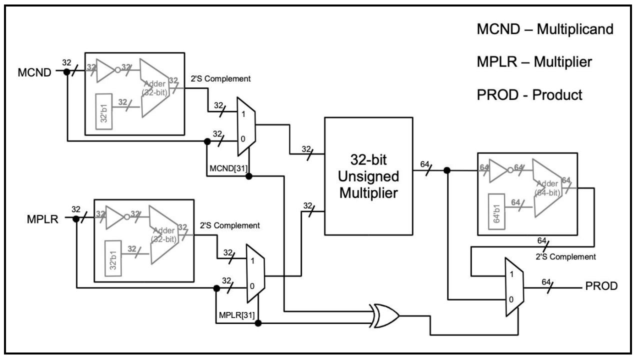
```
mult_unsigned_loop:
        beq     $s1, 0x20, end_mult_unsigned
        extract_nth_bit($a0, $s3, $zero)
        jal     bit_replicator
        move    $s5, $v0
        and     $s6, $s4, $s5
        move    $a0, $s2
        move    $a1, $s6
        jal     add_logical
        move    $s2, $v0
        srl     $s3, $s3, 0x1
        extract_nth_bit($s7, $s2, $zero)
        add     $t0, $zero, 31
        insert_to_nth_bit($s3, $t0, $s7, $t1)
        srl     $s2, $s2, 0x1
        addi    $s1, $s1, 0x1
        j       mult_unsigned_loop
```

```
end_mult_unsigned:
        move    $v0, $s3
        move    $v1, $s2
        # restore frame
        lw      $fp, 60($sp)
        lw      $ra, 56($sp)
        lw      $a0, 52($sp)
        lw      $a1, 48($sp)
        lw      $a2, 44($sp)
        lw      $a3, 40($sp)
        lw      $s0, 36($sp)
        lw      $s1, 32($sp)
        lw      $s2, 28($sp)
        lw      $s3, 24($sp)
        lw      $s4, 20($sp)
        lw      $s5, 16($sp)
        lw      $s6, 12($sp)
        lw      $s7, 8($sp)
        addi    $sp, $sp, 60
        jr      $ra
```

The loop repeats 32 times. In the loop, add_logical acts as the full adder. To mirror the shifting in pencil multiplication, the extraction of the product's bits is put into the lower bits of the 64 bit result.

5. *Mult_signed*

This procedure is almost identical to mult_unsigned, except it can also multiply negative numbers.



First off, if the operands are negative then they need to be translated into their two's complement form. Once they are in two's complement form, we can call to mult_unsigned to multiply. Once multiplication is done we need to use sign extension and 64 bit complement utilities in order to have two 32 bit results in $v0 and $v1. A XOR between the original MCND and MLPR's signs will determine the signs of $v0 and $v1. Since the result will be in 64-bit, the Lo 32 bits will be stored in $v0 and the Hi 32 bits will be stored in $v1. This is because each register can only hold 32 bits, and by doing so it ensures that mult_signed returns the correct value.

```
mult_signed:
        # frame creation
        addi    $sp, $sp, -60
        sw      $fp, 60($sp)
        sw      $ra, 56($sp)
        sw      $a0, 52($sp)
        sw      $a1, 48($sp)
        sw      $a2, 44($sp)
        sw      $a3, 40($sp)
        sw      $s0, 36($sp)
        sw      $s1, 32($sp)
        sw      $s2, 28($sp)
        sw      $s3, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        addi    $fp, $sp, 60
```

```
        move    $s3, $a0
        move    $a2, $a0
        move    $s4, $a1
        move    $a3, $a1
        jal     twos_complement_if_neg
        move    $s3, $v0
        move    $a0, $s4
        jal     twos_complement_if_neg
        move    $s4, $v0
        move    $a0, $s3
        move    $a1, $s4
        jal     mult_unsigned
        move    $s3, $v0
        move    $s4, $v1
        add     $t0, $zero, 0x1F
        extract_nth_bit($t1, $a2, $t0)
        extract_nth_bit($t2, $a3, $t0)
        xor     $t3, $t1, $t2
        # if S = 1, use 2's complement 64 bit t
        beq     $t3, 0x1, twos_comp_64bit
        j       end_mult_unsigned
```

```
twos_comp_64bit:
        move    $a0, $s3
        move    $a1, $s4
        jal     twos_complement_64bit
        move    $s3, $v0
        move    $s4, $v1

end_mult_signed:
        move    $v0, $s3
        move    $v1, $s4
        # restore frame
        lw      $fp, 60($sp)
        lw      $ra, 56($sp)
        lw      $a0, 52($sp)
        lw      $a1, 48($sp)
        lw      $a2, 44($sp)
        lw      $a3, 40($sp)
        lw      $s0, 36($sp)
        lw      $s1, 32($sp)
        lw      $s2, 28($sp)
        lw      $s3, 24($sp)
        lw      $s4, 20($sp)
        lw      $s5, 16($sp)
        lw      $s6, 12($sp)
        lw      $s7, 8($sp)
        addi    $sp, $sp, 60
        jr      $ra
```

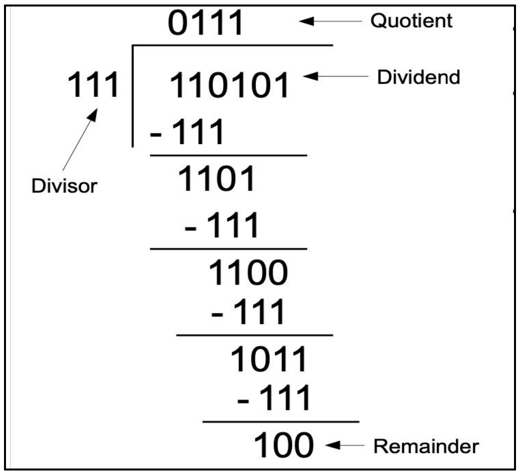
6. *div_unsigned*

This procedure is like mult_unsigned because binary division is also similar to decimal division. Div_unsigned takes the following three arguments:

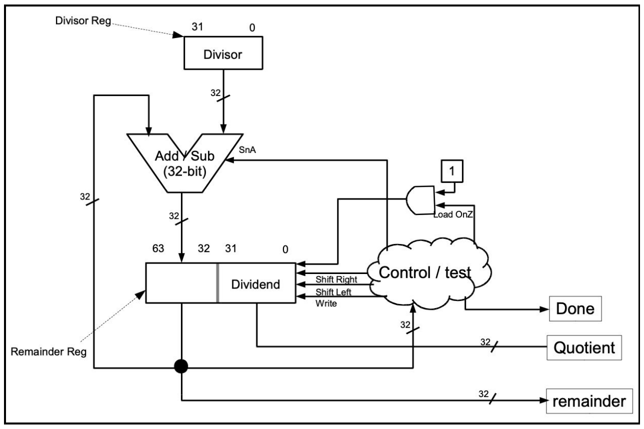
- $a0: the dividend
- $a1: the divisor

Then it returns the following:

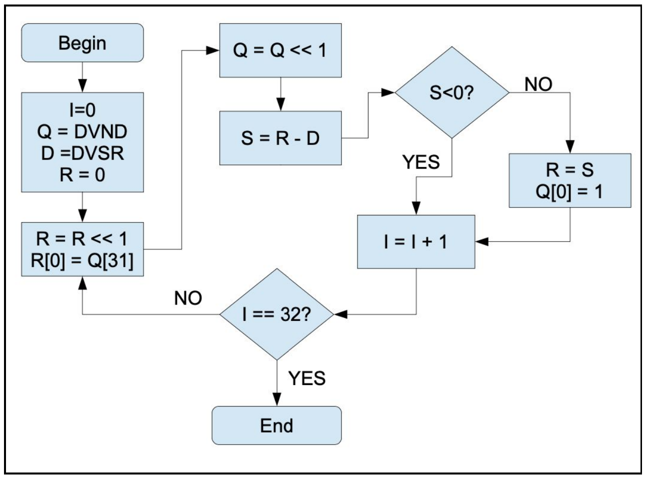
- $v0: the quotient
- $v1: the remainder



To divide 64-bit numbers, the remainder register value needs to be left shifted in order for the last bit of the quotient register to be placed into the first position. As a result, the quotient register would also be shifted left. We need to loop this 32 times to get the correct results.



The following flow chart shows the process of div_unsigned:



The implementation is as follows:

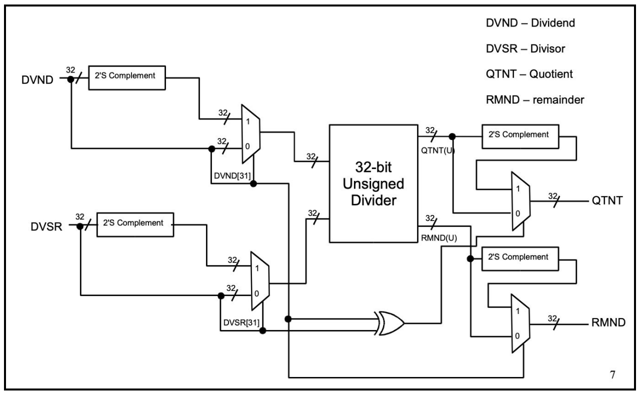
```
div_unsigned:
        # frame creation
        addi    $sp, $sp, -60
        sw      $fp, 60($sp)
        sw      $ra, 56($sp)
        sw      $a0, 52($sp)
        sw      $a1, 48($sp)
        sw      $a2, 44($sp)
        sw      $a3, 40($sp)
        sw      $s0, 36($sp)
        sw      $s1, 32($sp)
        sw      $s2, 28($sp)
        sw      $s3, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        addi    $fp, $sp, 60
```

```
        add     $s1, $zero, $zero
        move    $s2, $a0
        move    $s3, $a1
        add     $s4, $zero, $zero

div_unsigned_loop:
        beq     $s1, 0x20, end_div_unsigned
        sll     $s4, $s4, 0x1
        addi    $s5, $zero, 0x1F
        extract_nth_bit($s6, $s2, $s5)
        add     $t9, $zero, $zero
        insert_to_nth_bit($s4, $zero, $s6, $t9)
        sll     $s2, $s2, 0x1
        move    $a0, $s4
        move    $a1, $s3
        jal     sub_logical
        move    $s7, $v0
        bltz    $s7, end_div_unsigned_loop
        move    $s4, $s7
        add     $t4, $zero, $zero
        addi    $t3, $zero, 0x1
        insert_to_nth_bit($s2, $zero, $t3, $t4)

end_div_unsigned_loop:
        addi    $s1, $s1, 0x1
        jal     div_unsigned_loop
```

```
end_div_unsigned:
        move    $v0, $s2
        move    $v1, $s4
        # restore frame
        lw      $fp, 60($sp)
        lw      $ra, 56($sp)
        lw      $a0, 52($sp)
        lw      $a1, 48($sp)
        lw      $a2, 44($sp)
        lw      $a3, 40($sp)
        lw      $s0, 36($sp)
        lw      $s1, 32($sp)
        lw      $s2, 28($sp)
        lw      $s3, 24($sp)
        lw      $s4, 20($sp)
        lw      $s5, 16($sp)
        lw      $s6, 12($sp)
        lw      $s7, 8($sp)
        addi    $sp, $sp, 60
        jr      $ra
```

7. *div_signed*

This procedure handles the division of positive numbers as well as negative numbers.



First we check the operands to see if they are negative and then we compute their two's complement. After that, we call to div_unsigned to carry out division. Last, the complemented numbers go back to their signed state to maintain consistency.

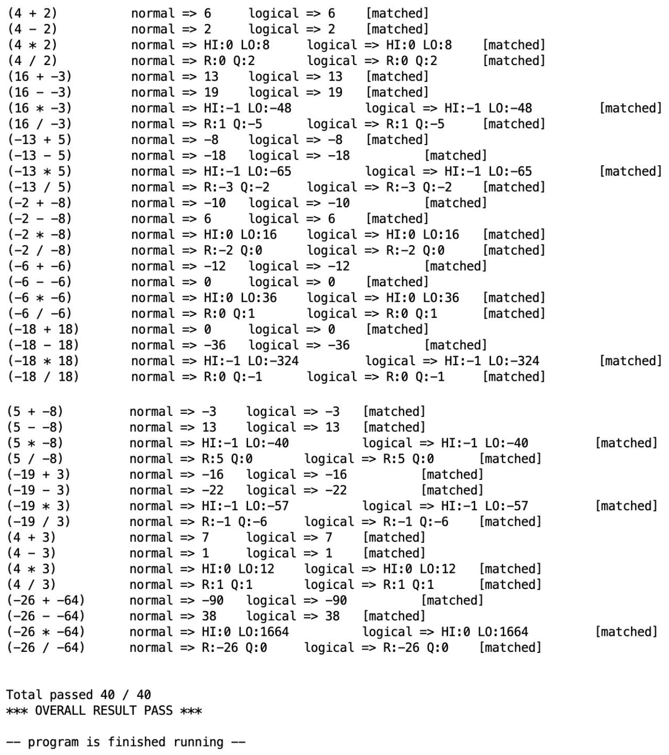
```
div_signed:
        # frame creation
        addi    $sp, $sp, -60
        sw      $fp, 60($sp)
        sw      $ra, 56($sp)
        sw      $a0, 52($sp)
        sw      $a1, 48($sp)
        sw      $a2, 44($sp)
        sw      $a3, 40($sp)
        sw      $s0, 36($sp)
        sw      $s1, 32($sp)
        sw      $s2, 28($sp)
        sw      $s3, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        addi    $fp, $sp, 60
```

```
        move    $s3, $a0
        move    $s4, $a1
        jal     twos_complement_if_neg
        move    $s1, $v0
        move    $a0, $a1
        jal     twos_complement_if_neg
        move    $s2, $v0
        move    $a0, $s1
        move    $a1, $s2
        jal     div_unsigned
        move    $s1, $v0
        move    $s2, $v1
        # find sign of Q
        addi    $t0, $zero, 0x1F
        extract_nth_bit($s5, $s3, $t0)
        extract_nth_bit($s6, $s4, $t0)
        xor     $s7, $s5, $s6
        beq     $s7, 0x1, twos_comp_q
        bne     $s7, 0x1, twos_comp_r

twos_comp_q:
        move    $a0, $s1
        jal     twos_complement
        move    $s1, $v0
        j       twos_comp_r

        # find sign of R
twos_comp_r:
        move    $s7, $s5
        bne     $s7, 0x1, end_div_signed
        move    $a0, $s2
        jal     twos_complement
        move    $s2, $v0
        j       end_div_signed
```

```
end_div_signed:
        move    $v0, $s1
        move    $v1, $s2
        # restore frame
        lw      $fp, 60($sp)
        lw      $ra, 56($sp)
        lw      $a0, 52($sp)
        lw      $a1, 48($sp)
        lw      $a2, 44($sp)
        lw      $a3, 40($sp)
        lw      $s0, 36($sp)
        lw      $s1, 32($sp)
        lw      $s2, 28($sp)
        lw      $s3, 24($sp)
        lw      $s4, 20($sp)
        lw      $s5, 16($sp)
        lw      $s6, 12($sp)
        lw      $s7, 8($sp)
        addi    $sp, $sp, 60
        jr      $ra
```

V. Testing

Once you have finished the logical implementation, you can go ahead and assemble the proj-auto-test.asm and run it to see your results. This will run a comparison between your logical implementation and normal procedures to see if they match up.

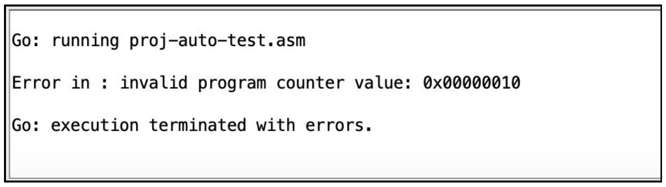


```
(4 + 2)         normal => 6     logical => 6    [matched]
(4 - 2)         normal => 2     logical => 2    [matched]
(4 * 2)         normal => HI:0 LO:8     logical => HI:0 LO:8    [matched]
(4 / 2)         normal => R:0 Q:2       logical => R:0 Q:2      [matched]
(16 + -3)       normal => 13    logical => 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 --
```

If your implementation is correct, it should show 'Total passed 40/40' as seen above.

VI. Common Mistakes

One common mistake that I experienced was creating and storing the right frames. If your stack frame is off, then the frame pointer would be incorrect. This was one of my biggest issues that took me a while to debug. I would get the following error:



```
Go: running proj-auto-test.asm

Error in : invalid program counter value: 0x00000010

Go: execution terminated with errors.
```

An important step in frame restoration is to remember to include 'jr $ra'. If you forget, then your procedure would not return to the caller correctly which would cause memory location problems when you are trying to return something.

Another common mistake is to not have consistent registers throughout your code. You need to make sure that the registers you are manipulating are holding the intended values that you want to use. For example, make sure to move computed results out of $v0 or $v1 and into other registers if you want to use them later. Make sure the registers you use match up throughout your whole code.

VII. Conclusion

In this project, I learned a lot about lower level design. I would have never imagined computing a simple operation like 16 / 4 would have such complicated underlying procedures. I was also exposed to MIPS assembly language and how different it is from Java and C. Using MIPS registers, branching methods, and procedures can get you lost pretty fast. The project can seem impossible at times when you are trying to debug and cannot seem to come up with a solution. Make sure you always double check your code and write comments along the way so that you can look back at what you've done. Overall, this project not only taught me about lower level programming, but it helped me develop better attention to detail and a higher understanding of what is really happening in my code.

References:

[1] K. Patra. CS 47. Class Lecture, Topic: "Addition Subtraction Logic." San Jose State University, San Jose, CA, November 27, 2019.

[2] K. Patra. CS 47. Class Lecture, Topic: "Multiplication Logic." San Jose State University, San Jose, CA, November 27, 2019.

[3] K. Patra. CS 47. Class Lecture, Topic: "Division Logic." San Jose State University, San Jose, CA, November 27, 2019.