Implementing a Calculator using MIPS Assembly and Logic Operations

Johann Kwon
Computer Science
San Jose State University
San Jose, United States
johann.kwon@sjsu.edu

***Abstract*—This report explains the creation and implementation of a calculator with basic functions (addition, subtraction, multiplication, and division) using MIPS assembly language and logical procedures.**

I. Introduction

By using MARS, we are able to calculate mathematical expressions using MIPS protocol. There are two different methods for the four basic operations (addition, subtraction, multiplication, and division): Using MIPS normal operations (add, sub, mult, and div, respectively) and MIPS logical operations (using Boolean operations such as AND and OR). The project objectives are such that:

1) To successfully execute and use MARS.
2) To implement two modules to do arithmetic calculations using both MIPS mathematical operations and MIPS logical operations.
3) To test and verify procedures and results.

II. Setup and Requirements

A. Installation

First, we must install MARS, an assembly language simulator. It can be downloaded for free from the following site: http://courses.missouristate.edu/KenVollmar/MARS/.

B. Loading the Project

Download the project files, which are in a zip file titled "CS47Project1.zip" from Canvas using the following link: https://sjsu.instructure.com/courses/1364573/assignments/5215393. The following files are contained inside:

1) "cs47_common_macro.asm": This file contains the common macros for the files "cs47_proj_procs" and "proj-auto-test.asm."

2) "CS47_proj_alu_normal.asm": This file contains the implementation of MIPS normal operations (add, sub, mult, and div).

3) "CS47_proj_alu_logical.asm": This file contains the implementation of MIPS logical operations by using Boolean logic such as AND and OR.

4) "cs47_proj_macro.asm": This file contains self-written macro files for the two files "CS47_proj_alu_normal.asm" and "CS47_proj_alu_lo gical.asm."

5) "cs47_proj_procs.asm": This file contains many printf procedures for "proj-auto-test.asm."

6) "proj-auto-test.asm": This file is an automatic tester that checks to see if your implemented procedures and operations function properly and produce expected results.

MARS should also be configured to certain settings. The default settings are located in the "Settings" option in the top menu when MARS is open (Fig. 1). When clicked, the following options should be selected: "Assembles all files in directory" and "Initialize program counter to global main if defined."

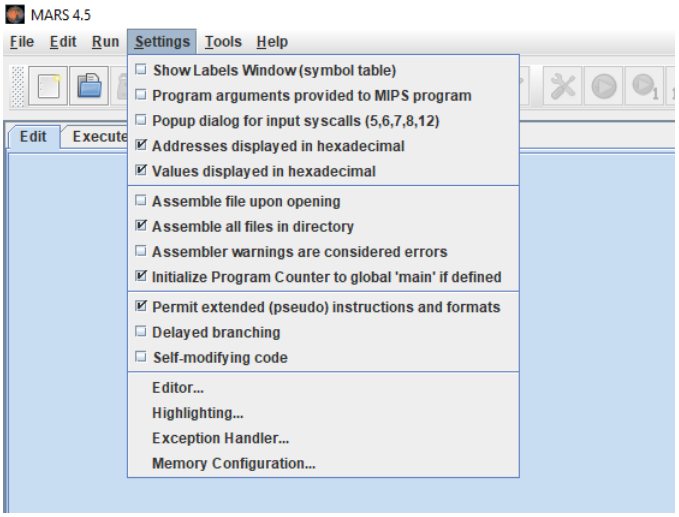

Fig. 1. Configured settings.

III. Normal Arithmetic Procedures

A. Initial Setup

The normal procedure begins with three basic registers as inputs in "CS47_proj_alu_normal":

- $a0, which contains the first operand
- $a1, which contains the second operand

- $a2, which contains the ASCII symbol that decides the operation being done. The ASCII symbols accepted are '+', '-', '*', and '/'.

We also use $v0 and $v1, which are the output variables. Each arithmetic operation will store different values in the output variables, and will be chosen depending on $a2.

B. *Addition*

'+' corresponds to addition, where we use the MIPS "add" instruction to add $a0 and $a1 together ($a0 + $a1). The sum will be stored in $v0.

C. *Subtraction*

'-' corresponds to subtraction, where we use the MIPS "sub" instruction to subtract $a1 from $a0 ($a0 - $a1). The difference will be stored in $v0.

D. *Multiplication*

'*' corresponds to multiplication, where we use the MIPS "mult" instruction to multiply $a0 and $a1 ($a0 * $a1). The LO of the product will be stored in $v0 using "mflo," and the HI of the product will be stored in $v1 using "mfhi."

E. *Division*

'/' corresponds to division, where we use the MIPS "div" instruction to divide $a0 by $a1 ($a0 / $a1). The quotient will be stored in $v0 and the remainder will be stored in $v1.

We can then implement these and use the MIPS instruction "beq" (branch on equal) to access addition, subtraction, multiplication, or division based on $a2. Here (Fig. 2.), the procedures are add_normal, subtract_normal, multiply_normal, and divide_normal, respectively.

```
beq $a2, '+', add_normal
beq $a2, '-', subtract_normal
beq $a2, '*', multiply_normal
beq $a2, '/', divide_normal
jr $ra
```

Fig. 2. Using "beq" to access arithmetic operations in the file "CS47_proj_alu_normal"

IV. LOGICAL ARITHMETIC PROCEDURES

A. *Initial Setup*

The logical procedure will be in nature similar to the normal procedures. However, the methods are different and procedures will be calling multiple procedures. We can use the general format of "CS47_proj_alu_normal" (Fig. 3) and branch off from there.

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

Fig. 3. Using similar logic from "CS47_proj_alu_normal" in "CS47_proj_alu_logical.asm."

We also need to be able to store the values of our saved registers (Fig. 4). We can do this by using the "sw" (store word) instruction to create a store frame. To do this we must shift $sp to the end of the stack by allocating space, and then storing the various registers we are going to use ($fp, $ra, $a0, $a1, $a2 as indicated in Fig. 4). The frame pointer must also be set up, and we can do this by shifting it to the top of the stack.

```
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
```

Fig. 4. Allocating space in the store frame.

Likewise we must also create a load frame (Fig. 5) using "lw" (load word). We restore the same registers as seen in Fig. 4 and shift $sp back up the stack.

```
lw $fp, 24($sp)
lw $ra, 20($sp)
lw $a0, 16($sp)
lw $a1, 12($sp)
lw $a2, 8($sp)
add $sp, $sp, 24
```

Fig. 5. Restoring previously stored registers in the load frame.

We must do this in procedures in order to keep the registers and the values in the registers the same so that the output does not produce unwanted results, especially when the registers are from before the procedure was called.

B. *Addition (add_logical)*

To understand how to implement addition, we must first look at the logic behind it.

1) Half Addition and Full Addition: There are two types of binary addition: half addition and full addition. Binary-wise, when you add two bits together, it produces a sum bit and a carry bit. Half addition considers only the two input bits, while full addition uses a carry bit (from a previous addition operation) while also using the two input bits. Fig. 6 shows the resultant sum and carry bits from half addition, while Fig. 7 shows the logic diagram, showing that the sum bit (Y) is an XOR operation between A and B (Y = A XOR B), while the carry bit (C) is an AND operation between A and B (C = A AND B).

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

Fig. 6. Half addition. [1]

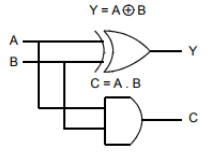

Fig. 7. Logic diagram for half addition. [1]

Fig. 8 shows the resultant sum bits (Y) and carry bits (now labeled CO) from A and B. Notice how there is an additional input: the carry in (CI) bit. CI is the carry bit that full addition uses from a previous addition operation. The carry out bits are for the next full addition operation.

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

$Y = \Sigma m(1,2,4,7)$

$CO = \Sigma m(3,5,6,7)$

Fig. 8. Full addition. [1]

To get our logic diagram, we must identify the associated minterm that causes the resulting sum and carry out bits to be 1. In Fig. 8, we can see that m1, m2, m4, and m7 cause the sum bit to be 1, hence $Y = \Sigma m(1,2,4,7)$. Likewise, m3, m5, m6, and m7 cause the carry out bit to be 1, resulting in $CO = \Sigma m(3,5,6,7)$.

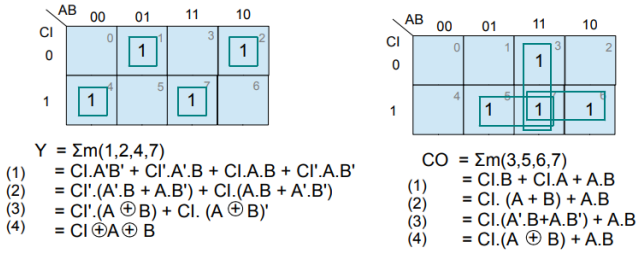

Fig. 9. The K-maps and reduced Product of Sums (POS) form. [1]

Now we can use a K-map to "plot" our minterms and reduce our equation. As seen in Fig. 9, $Y = \Sigma m(1,2,4,7)$ and $CO = \Sigma m(3,5,6,7)$ result in different POS forms:

- Y = CI XOR A XOR B
- CO = CI AND (A XOR B) OR (A AND B)

This represents our full adder in our logical procedure, for which the logic diagram can be created, as seen in Fig. 10.

$Y = CI \oplus (A \oplus B)$

$CO = CI.(A \oplus B) + A.B$

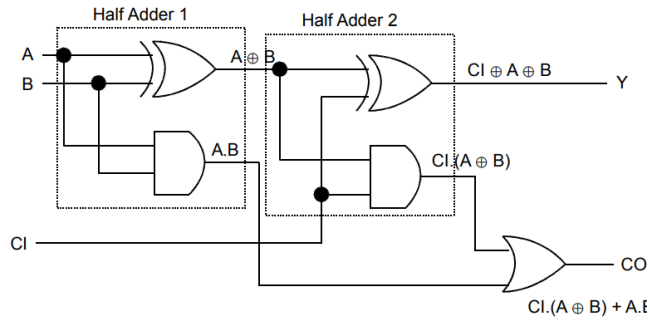

Fig. 10. Logic diagram for full addition. [1]

Consider also that this logic is for single bit use. To full add muti-bit numbers, we can combine multiple single bit full adders. The resulting adder is called a binary ripple carry adder. As a general formula, to get an *n*-bit adder we combine *n* 1-bit full adders. For example, to add integers, we need to combine 32 single bit full adders, and use an initial CI = 0. Fig. 11 shows the combination of four full adders to get a 4-bit number.

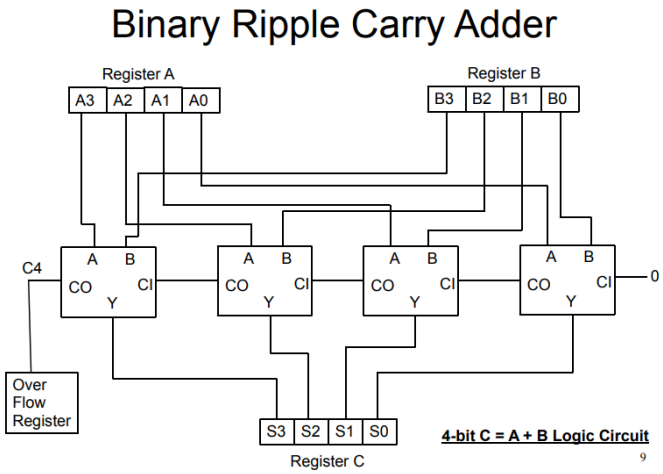

Fig. 11. A binary ripple carry adder for a 4-bit number. [1]

2) Macros: In order to implement addition into our "CS47_proj_alu_logical.asm," there are some helpful macros we need to put into place first: extract_nth_bit, extract_nth _bit_0, insert_to_nth_bit, half_adder and full_adder. Fig. 12 shows the implementation of the macros.

```
.macro extract_nth_bit($regD, $regS, $regT)
        srlv $regS, $regS, $regT                # shift bit pattern ($regS)
        li $regD, 1                             # $regD = 1
        and $regD, $regS, $regD                 # $regD = $regS and $regD,
.end_macro

.macro extract_nth_bit_0($regD, $regS)
        li $regD, 1                             # $regD = 1
        and $regD, $regS, $regD                 # mask to get $regS
        srl $regS, $regS, 1                     # shift $regS to right by 1
.end_macro

.macro insert_to_nth_bit($regD, $regsS, $regT, $maskReg)
        move $maskReg, $regT                    # move $regT into $maskReg
        sllv $maskReg, $maskReg, $regS          # shift bit left by n
        or $regD, $regD, $maskReg               # $regD = $regD or $maskReg
.end_macro

.macro half_adder($a, $b, $y, $c)
        xor $y, $a, $b                          # $y = $a xor $b
        and $c, $a, $b                          # $c = $a and $b
.end_macro

.macro full_adder($a, $b, $ab, $y, $ci, $co)
        half_adder($a, $b, $y, $ab)             # $ab is the carry bit
        and $co, $ci, $y                        # $co = $ci and ($a xor $b)
        xor $y, $ci, $y                         # $y = $ci xor ($a xor $b)
        xor $co, $co, $ab                       # $co = $ci and ($a xor $b)
.end_macro
```

Fig. 12. Implementation of extract_nth_bit, extract_nth_bit_0, insert_to_nth_bit, half_adder and full_adder.

3) Implementation: After creating our helper macros and allocating/deallocating space for various registers, we can implement our addition procedure. As mentioned above, in order to add integers we must combine 32 single bit adders

(since integers are 32-bit). We can create a loop (as seen in Fig. 13) to run through our full adder 32 times, essentially combining single bit adders 32 times.

```
li $t0, 0                       # hold $a2
li $t1, 0                       # hold $a1
li $t2, 0                       # $s1, position of $s0
li $t4, 0                       # carry holder ($y)
li $t5, 0                       # bit answer ($ab), hold
li $t6, 0                       # carry in bit ($ci) for
li $t7, 0                       # carry out bit ($co)
add $s0, $zero, $               # save result $s0

add_loop:
        beq $t2, 32, end_add_loop
        extract_nth_bit_0($t0, $a0)             # first
        extract_nth_bit_0($t1, $a1)             # first
        full_adder($t0, $t1, $t5, $t4, $t6, $t7)
        insert_to_nth_bit($s0, $t2, $t5, $t6)
        move $t6, $t7
        addi $t2, $t2, 1
        j add_loop

end_add_loop:
        move $v0, $s0
        move $v1, $t7
```

Fig. 13. Implementation of add_logical.

4) Overflow: We also must consider overflow when we add. Overflow is when the sum of two numbers can't be represented with the given number of bits, so the result is larger than it is. Fig. 14 demonstrates how overflow can change the result of our process.

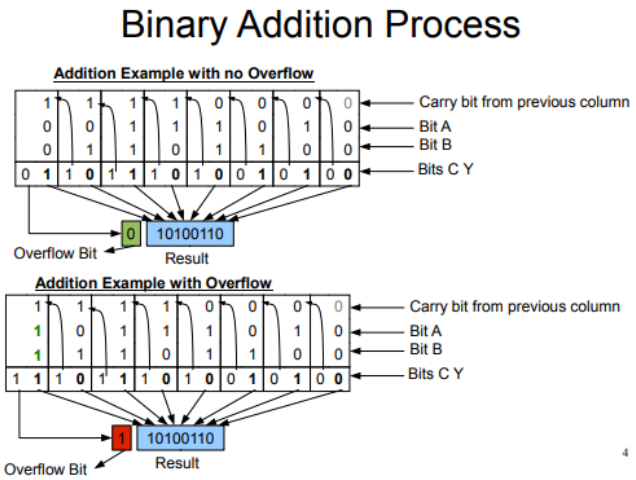

Fig. 14. Overflow in binary addition. [1]

C. Subtraction

Consider that subtraction is addition but using a negative operand on the right (E.g. 1 - 1 = 1 + -1). This means we can use add_logical to complete our sub_logical. To do so, we have to change the operator from '+' to '-', or from 0x00000000 to 0xFFFFFFFF. This means we can do addition with a negative right operand, as seen in Fig. 15.

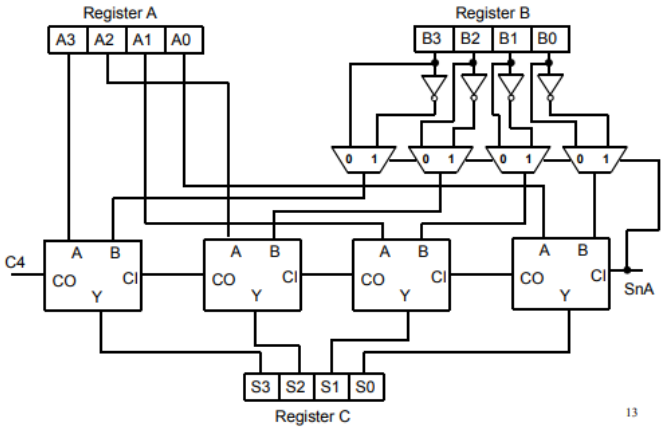

Fig. 15. The combination of a binary ripple carry adder and subtractor. [1]

Fig. 16 shows the implementation of sub_logical. Notice it follows the negation of the second operator $a1 and puts it into add_logical.

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

        neg $a1, $a1
        jal add_logical

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

Fig. 16. Implementation of sub_logical.

D. Multiplication

Moving on to multiplication, we must first go over the logic. We can start by looking at binary based, paper-pencil basic multiplication, which is simply going through the process of multiplying each operand and the places of each digit. Looking at Fig. 17, we can see that the top operand is called the multiplicand, and the bottom operand is the multiplier. At each step, we have to compute a partial product (made up of several sums), which are shifted over by one at each consecutive step. We add straight down in order to get the final product.

Although I could not get the bulk of multiplication to function correctly in my code, the logic behind it and its various macros can be explained.

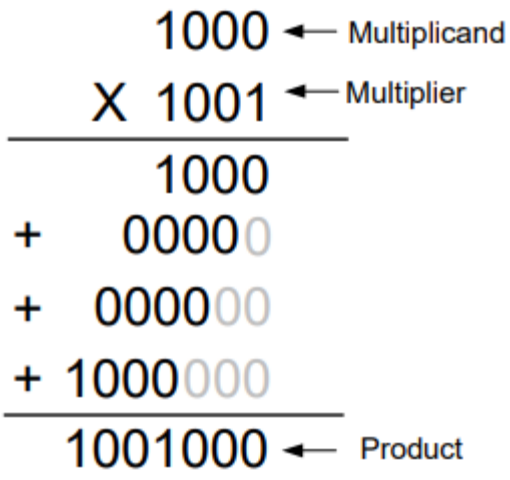

Fig. 17. Paper-pencil binary multiplication. [2]

Now we can move on to the algorithm. Fig. 18 shows us the algorithm and process we need to go through in order to find our final product.

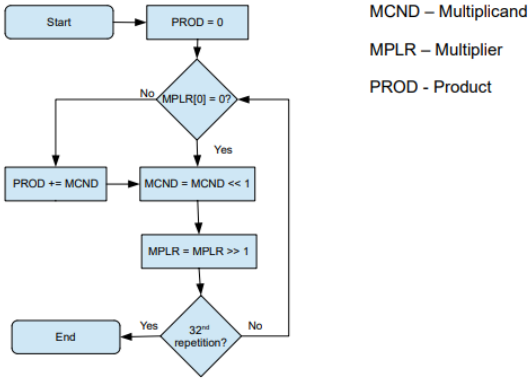

Fig. 18. Binary multiplication. [2]

Essentially, we start at a product = 0. Then we look at the least significant bit (LSB) of the multiplier, and check if it is 0. If not, it is 1, so we add the product with the multiplicand, and then left shift the multiplicand by 1 bit. If it is 0, then we don't add anything and just shift it left. We also shift the multiplier to the right by 1. Note that we have to go through this process 32 times in order for a 32-bit multiplier. Also consider that if the product is 64-bit, the multiplicand and multiplier will also be 64-bit.

1) Utility Procedures: In order to implement the multiplication procedure, we can implement some utility procedures to help with the process.

- twos_complement: This procedure creates the two's complement of a given number in $a0. It then puts the number in $v0.
- twos_complement_if_neg: If the given number is negative, then we use twos_complement_if_neg to calculate the two's complement of the number. It will always return a positive bit pattern.
- twos_complement_64bit: We take in the Lo and Hi of a number and return the Lo and Hi of the two's complemented number as a 64-bit result.
- bit_replicator: bit_replicator simply replicates the given bit value into a 32-bit output.

Now we can go into the macros for unsigned and signed multiplication.

2) Unsigned Multiplication: It is important to remember that the multiplication we have done so far is unsigned. Remember, we take in two 32-bit values (the multiplicand and the multiplier) as operands and produce a 64-bit result. In mul_unsigned, the multiplicand is taken through $a0 and the multiplier through $a1. The result will be a Lo and Hi of the product, in $v0 and $v1 respectively. Fig. 19 describes the process step by step.

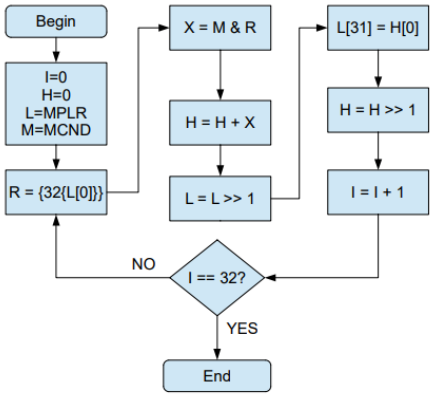

Fig. 19. Unsigned multiplication. [2]

We can see in Fig. 19 that the index of our loop and Hi both start equal to 0. The LSB of the multiplier (R) goes through the process, becoming M AND R, and is added to Hi. Then Lo will be shifted to the right, making Lo's most significant bit (MSB) equal to the LSB of Hi. This then loops around 31 more times.

3) Signed Multiplication: Signed multiplication can involve negative numbers as well (hence the - "sign"). Essentially you take two operands, and if either of them is negative, you take the two's complement of that number. Then you take the original signs of the operands, and from there assign the product's sign based on those. You do this by using XOR in the form A XOR B, where A is $a0 and B is $a1. If the result is 1, then the product is negative. If the result is 0, then the product is positive.

E. Division

Similarly to multiplication, division takes an operation and repeats it however many times you need to. In division's case, however, we use subtraction (and shift to the left) instead of addition and shifting to the right.

I was also unable to finish division as well, but the logic behind it can still be explained.

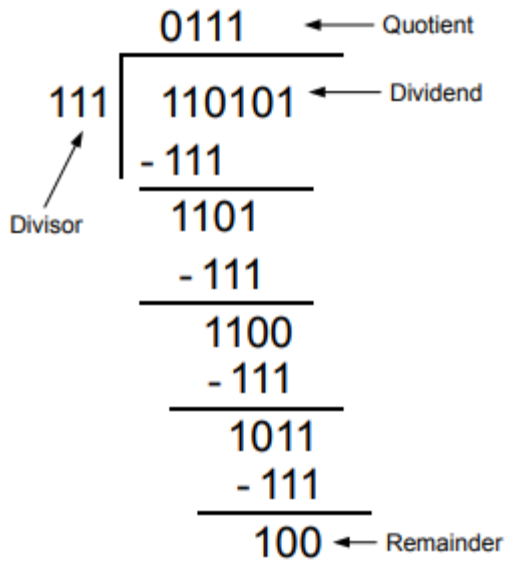

Fig. 20. Paper-pencil binary division. [3]

If we take a look at Fig. 20, which demonstrates a simple binary based paper-pencil division process, we see that the two operands are the dividend and the divisor (the number you're dividing the dividend by). We can also see that we need to compute a partial quotient (made up of differences) at each step, and shift over for the next one. The quotient is found by seeing however many times you can "fit" the divisor in into the dividend.

In other words, we align the divisor with the most significant end of the dividend. Then we compare the selected portion of the dividend (which is from the MSB to the bit aligned with the LSB of the divisor), and the divisor. If the selected portion is larger than or equal to the divisor, then we perform a subtraction, subtracting the divisor from the portion. The quotient bit will be 1. If the selected portion is smaller than the divisor, then the quotient bit is 0, and no subtraction occurs. Then we move on to the next bit and repeat the process, until we cover all of the dividend.

Division will produce a quotient and a remainder (if possible).

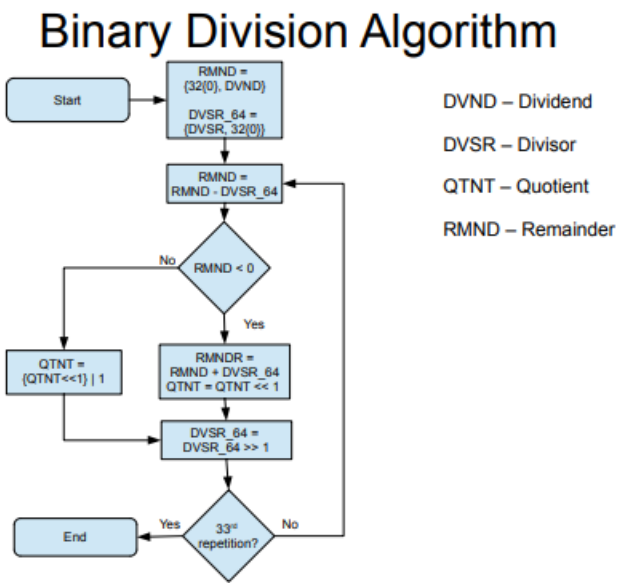

Fig. 21. Binary division. [3]

Taking a look at Fig. 21, we can see that we start with a 64-bit remainder register (labeled RMND). We put a 32-bit (being all 0's) the upper half and the dividend in the lower half. We also get the divisor register (being 64-bit, labeled DVSR_64), and load it with a divisor 32-bit bit pattern in the upper half, and fill the lower half with a 32-bit (being all 0's).

Continuing on, we set RMND to RMND – DVSR_64, which will involve 64-bit subtractor. If the RMND is greater than 0, then we left shift the quotient bit (labeled QTNT) and insert 1 into the quotient bit. Then we right shift DVSR_64 by 1. If the remainder is less than 0, then we set the remainder equal to the remainder plus the divisor register, and left shift the quotient by 1. Then we right shift the divisor register by 1. We continue this in a loop until we go through this process *33* times, not 32.

Now we can go into our procedures.

1) div_unsigned: Looking at Fig. 22, we can see the general process for unsigned division.

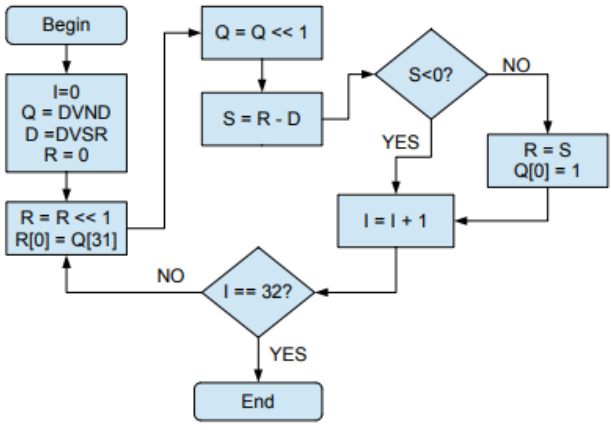

Fig. 22. Unsigned division. [3]

We start off with an index of 0 and our divisor (32-bit) and dividend (32-bit). We left shift our remainder (R) by 1 and put the MSB of the dividend into the LSB of the remainder. Then we left shift the quotient and subtract the divisor from the remainder. If the difference is greater than or equal to 0, then the remainder becomes the difference, and we put 1 into the quotient LSB. Then we increment the index by 1. If the difference is less than 0, then we just increase the index by 1. Then we repeat this process, restarting where we left shift the register by 1.

2) div_signed: Here we do signed division, which can involve negative numbers. Similarly to multiplication, we take the two's complement of our operands and do unsigned division. Then we need to find the sign of the quotient. We take the MSB of the original operands and do an XOR. If the result is 1, meaning it's negative, then we can use our "twos_complement" procedure to find the two's complement of the quotient. If positive, we leave it be.

We also need to need to find the sign of the remainder, and so we take the MSB of the first operand and if it is 1 (negative), we find its two's complement.

V. Testing

To test if a procedure is working properly or not, we must first save and assemble the procedure at hand. This will assemble "proj_auto_test.asm" and we can move on to running it by clicking the "Run" icon on the top menu. Essentially, "proj_auto_test" goes through "CS47_proj_alu_normal.asm" and "CS47_proj_alu_logical.asm," as well as several test cases. "If successful, the following output will be produced, as seen in Fig. 23.

```
(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. 23. Expected output after assembly and run.

Should a different output occur, such as error messages, or "Total passed x/40," we know we must continue to test and alter our procedures logically. There are many possible reasons as to why the output may not match the expected result. Perhaps the implementation of an operation design was not correct, or the use of temporary registers lost a certain value. It may not be certain, which is why debugging becomes necessary.

VI. Conclusion

To conclude, the objectives of the project were clear: To implement basic arithmetic operations through the use of MIPS assembly, as well as understand the logical operations behind them. As a result, this project helped me understand more about the how logical operations and designs inside a computer perform. Although I did not finish some of the last parts of the logical procedure, I still learned a lot about logical processes, as well as the usage of various MIPS instructions. Overall, the project was an interesting experience and gave me more experience regarding low level assembly, and how computer science projects function.

VII. References

[1] K. Patra. CS 47. Class Lecture, Topic: "Addition Subtraction Logic." San Jose State University, San Jose, CA, April 16, 2020

[2] K. Patra. CS 47. Class Lecture, Topic: "Multiplication Logic" San Jose State University, San Jose, CA, April 21, 2020

[3] K. Patra. CS 47. Class Lecture, Topic: "Division Logic." San Jose State University, San Jose, CA, April 23, 2020