Arithmetic Implementation of ALU Using MIPS

Jyothi Shankar
San Jose State University
jyothi.shankar@sjsu.edu

***Abstract* – This report presents two ways to perform basic arithmetic operations – addition, subtraction, multiplication, and division -- in MIPS using normal and logical procedures is MARS IDE. The normal procedure involves using the normal mathematical operations provided by MIPS. The logical procedure on the other hand involves following the ALU's hardware implementation to compute the arithmetic operations, and therefore it cannot use the provided MIPS math operations.**

I. Introduction

The purpose of this project is to follow how the Arithmetic Logic Unit, or ALU, uses logical operators to calculate arithmetic operations. The assembly language MIPS (Microprocessor without Interlocked Pipelined Stages) will be used to perform these calculations. In this project, we will do these calculations in two different ways: one in which we use the normal operations given to us by MIPS to simplify the calculation process (add, sub, mult, div) and one in which we need to compute using the process of the processor, which is through logical operations (AND, OR, NOT, etc.).

For this project, we are using the MARS software, which is an integrated development environment that will simulate MIPS.

II. Requirements

The project is required to be completed on MARS IDE using the MIPS assembly language. To set up the files and settings for the project:

1. Download the provided CS47Projectl.zip folder.
2. After downloading, unzip the folder to separate the individual files. The individual files include:
   a) *cs47_common_macro.asm*:
      File that provides commonly used macros which can be accessed by your code if necessary. This file cannot be altered.
   b) *CS47_proj_alu_logical.asm*:
      File in which you write the logical implementation for arithmetic operations.
   c) *CS47_proj_alu_normal.asm*:
      File in which you write the implementation for arithmetic operations using MIPS mathematical instructions.
   d) *cs47_proj_macro.asm*:
      File in which you may define macros you wish to use in your logical implementation.
   e) *cs47_proj_procs.asm*:
      File which creates procedures that are used by proj-auto-test. This file cannot be altered and does not have much to do with the requirements of the project.
   f) *proj-auto-test.asm*:
      Provided file in which the normal and logical implementation are being derived and compared to each other. This file generates operands and operation codes, and will compute 40 unique arithmetic operations using your normal procedure and logical procedure. It will determine your score and whether or not you passed the test depending on how many results from the logical implementation match the results from the normal implementation. This file cannot be altered.
3. Download and open MARS IDE, go to "Settings", and turn on "Assembles all files in directory" and "Initialize program counter to global main if defined".

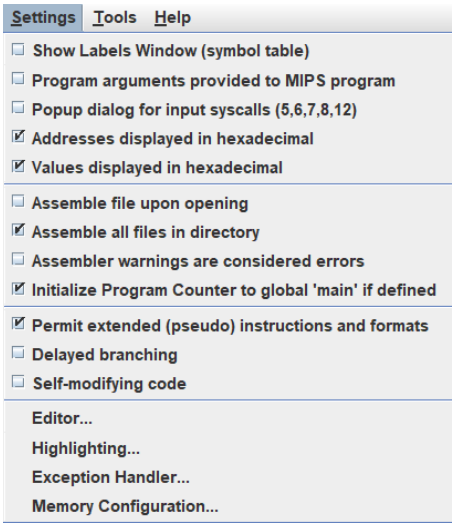

Figure 1: MARS Settings

4. Open all six files within CS47Projectl.zip in MARS.

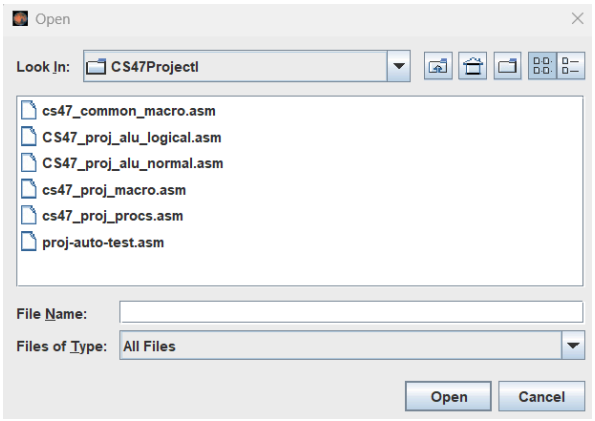

Figure 2: Opening files

As mentioned, this project requires two main components: a normal and a logical procedure. Both procedures need to correctly compute the mathematical operations derived from the file proj-auto-test, which contains the "main" and calls on both procedures to test them with various integer values and mathematical operations. Each procedure, however, has different requirements:

A. *Same for Both:*

Both the normal and logical procedure take three arguments into three registers: $a0, $a1, and $a2, and return the necessary values into $v0 and $v1.

a) $a0: Register $a0 is an argument register which stores the first operand, or first number.
b) $a1: Register $a1 is an argument register which stores the second operand, or second number.
c) $a2: Register $a3 is an argument register which will store the operation code, or an ASCII code, which helps to direct which arithmetic operation will be performed using the two operations: addition, subtraction, multiplication, or division.
d) $v0: The value register $v0 holds depends on what arithmetic operation is being performed. The register $v0 will hold the sum if addition is being performed on the two numbers, and it will hold the difference if subtraction is being performed. In the case of multiplication, since the operation involves HI and LO registers, $v0 will hold the value of the LO register. For division, $v0 will hold the quotient.
e) $v1: The value register $v1 is used only for multiplication and division operations. For multiplication, this register will hold the value of the HI register once the arithmetic operation is performed, and for division it will hold the remainder.

B. *Normal Procedure:*

The normal procedure, au_normal, needs to be created in the file *CS47_proj_alu_normal.asm*. It needs to use the MIPS mathematical operations "add", "sub", "mult", and "div" to compute the arithmetic operations. This will only require a singular procedure to complete.

C. *Logical Procedure:*

The logical procedure, au_logical, needs to be created in the file *CS47_proj_alu_logical.asm*. It needs to use logical operations like "AND", "OR", "NOT", etc. in order to compute the arithmetic operations for addition, subtraction, multiplication, and division. MIPS mathematical operations "add", "sub", "mul", "div", etc. may not be used in this procedure unless it is being used for purposes that are not directly connected to the computation of the operations, such as incrementing an index. Because this process will be much longer and more complex than the normal procedure, it will require multiple procedures to complete.

III. Design and Implementation

A. *Normal Procedure:*

This procedure collects the ASCII code from register $a2 to determine which arithmetic operation is being performed on the two operands ($a0 and $a1): addition, subtraction, multiplication, or division. It then branches to the corresponding label which will perform that arithmetic operation on the two numbers.

```
au_normal:
        beq $a2, 43, add_label        #goes to add_label if $a2 = '+'
        beq $a2, 45, subtract_label   #goes to subtract_label if $a2 = '-'
        beq $a2, 42, multiply_label   #goes to multiply_label if $a2 = '*'
        beq $a2, 47, divide_label     #goes to divide_label if $a2 = '/'
```

Figure 3: Branching in normal procedure

Each label will use the mathematical instructions provided by MIPS: "add", "sub", "mult", and "div", to compute the sum, difference, product, or quotient/remainder.

```
add_label:                    #label that will add the two values
        add $v0, $a0, $a1
        jr $ra

subtract_label:               #label that will subtract the two values
        sub $v0, $a0, $a1
        jr $ra

multiply_label:               #label that will multiply the two values
        mult $a0, $a1
        mflo $v0              #value will be stored in register $v0 and $v1
        mfhi $v1
        jr $ra

divide_label:                 #label that will divide the two values
        div $a0, $a1
        mflo $v0              #quotient stored in $v0
        mfhi $v1              #remainder stores in $1
        jr $ra
```

Figure 4: Normal procedure computation using MIPS instructions.

The resulting answers for each will be stored into $v0, and if multiplication or division, also into $v1.

B. *Macros Used in Logical Procedure*

Because the logical procedure is much more complex than the normal procedure and it will require many repetitive steps that are used throughout, defining macros will be useful. Macros are meant to make the main code, which is in *CS47_proj_alu_logical.asm* in this case, less cluttered and repetitive. The macros will be defined in the file *cs47_proj_macro.asm.*

1) *extract_nth_bit*

This macro extracts the value of a bit at a specified position from a bit pattern. This is a useful macro because it will be used for addition, subtraction, multiplication, and division in the logical procedure in order to manipulate the bit patterns of the two operands and compute the operation.

The macro takes three register arguments. The register $regSourceBit is the bit pattern from which the bit is being extracted from. The register $regPos holds the position of the bit that needs to be extracted. The $regVal register will contain a 1 or 0 depending on what the extracted bit is. Making is used to hold the value of $regSourceBit shifted right by the amount of the but position, and then the "and" operation is used with $regSourceBit and the immediate value '1' in order to determine whether the value extracted is a 1 or a 0, which is stored in $regVal.

```
#Usage: extracts the bit from nth position
.macro extract_nth_bit ($regVal, $regSourceBit, $regPos)
#move content of $regSourceBit to $s0 so that $regSourceBit value isn't manipulated directly
move $s0, $regSourceBit
#shifts to right based on position of bit that is being extracted
srav $s0, $s0, $regPos
#will use "AND" operation with 1 and extracted bit, resulting in 1 or 0
and $regVal, $s0, 1
.end_macro
```

Figure 5: Macro to extract nth bit

2) *insert_to_nth_bit*

This macro inserts a specified bit at a specified position in a bit pattern. This is useful when manipulating a bit pattern in order to add, subtract, multiply, or divide two numbers.

The macro takes four arguments. All the registers have the same function as they do for the macro *extract_nth_bit*, but this macro also has an argument $maskReg, which holds a temporary bit pattern that is manipulated to insert a bit into the specified bit pattern. $maskReg begins with a value of 1, the 1 gets shifted the position of the bit that is to be inserted, the bit pattern is inverted and an "and" operation is performed on the $maskReg and the original source bit. The bit that is being inserted is shifted left to the position it needs to be inserted into, and an "or" operation is used to create the final bit pattern with the inserted bit.

```
.macro insert_to_nth_bit ($regVal, $regSourceBit, $regPos, $maskReg)
li $maskReg, 1
#shifts to left based on position of bit that is being added
sllv $maskReg, $maskReg, $regPos
#will invert the bit pattern in maskReg (all bits will be 1 except bit at nth position)
not $maskReg, $maskReg
#performs "AND" operation with inverted mask and the original source bit to reset the nth bit to zero
and $regSourceBit, $regSourceBit, $maskReg
#creates a bit pattern made up of the bit 1 or 0 (depending on what bit is being inserted)
sllv $regVal, $regVal,$regPos
#performs "OR" operation to create the bit pattern with the inserted nth bit
or $regSourceBit, $regSourceBit, $regVal
.end_macro
```

Figure 6: Macro to insert bit into nth position

3) *store_RTE*

The *store_RTE* macro stores the frame, which includes the frame pointer register ($fp), the return address register ($ra), as well as the registers $a0-$a3 and $s0-$s7. This macro does not take in any arguments unlike the previous macros discussed. Although some of the arguments and saved temporary registers may not be used for every single procedure and therefore do not necessarily need to be saved, storing all of them in one go is more convenient and efficient rather than rewriting these lines of code for every procedure every time the frame needs to be stored.

```
#store RTE - 14 *4 = 56 bytes
#storing fp, ra, $a0-#a3, $s0-$s7
.macro store_RTE
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
.end_macro
```

Figure 7: Macro to store frame

4) *restore_RTE*

Similar to the *store_RTE* macro, the *restore_RTE* macro serves the same purpose of making the process of restoring the frame more efficient and less repetitive. This macro will restore all the values held in each register.

```
#restore frame
.macro restore_RTE
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
.end_macro
```

Figure 8: Macro to restore frame

5) *twos_complement_checker*

This macro is meant for the multiplication and division logic portion of the logic procedure. Because using *twos_complement_if_negative,* the procedure that will be discussed later, jumps to many different procedures and changes the values held in many registers, there were issues with the program running an infinite loop. This macro helps avoid issues that I ran into when I was jumping and linking too much and losing track of values stored in my registers. It checks if a number is negative, and if it is, it will convert the number to twos complement, which will be discussed later on.

```
#Usage: if number is negative, this macro converts it to twos complement
.macro twos_complement_checker($regS, $regFinal)
bgt $regS, 0, positive_val
not $regS, $regS
#addi $regFinal, $regS, 1
la $a0, ($regS)
li $a1, 1
li $a2, 0
jal add_sub_logical
la $regFinal, ($v0)
j end_checker

positive_val:
        move $regFinal, $regS
end_checker:
.end macro
```

Figure 9: Macro checks if number needs to convert to two's complement

6) *twos_complement_convert*

Unlike the macro *twos_complement_checker*, this macro converts the number to two's complement without checking whether or not it is a negative or positive number. This macro is utilized for division logic.

```
#Usage: converts number to twos complement regardless of whether or not it is +/-
.macro twos_complement_convert($regS, $regFinal)
not $regS, $regS
la $a0, ($regS)
li $a1, 1
li $a2, 0
jal add_sub_logical
la $regFinal, ($v0)
.end macro
```

Figure 10: Macro to convert number to two's complement

C. *Logical Procedure*

This procedure, just like the normal procedure, starts by collecting the ASCII code from register $a2 to determine which arithmetic operation is being performed on the two operands: addition, subtraction, multiplication, or division. It then branches to the corresponding label which will perform that arithmetic operation on the two numbers.

```
au_logical:
        store_RTE
        beq     $a2, 43, add_logical    #goes to add_label if $a2 = '+'
        beq     $a2, 45, sub_logical    #goes to subtract_label if $a2 = '-'
        beq     $a2, 42, mul_logic      #goes to multiply_label if $a2 = '*'
        beq     $a2, 47, div_logic      #goes to divide_label if $a2 = '/'
```

Figure 11: Branching in logical procedure

1) *add_logical*

This procedure loads a mode of 0 into $a2. The mode determines whether addition or subtraction will be performed on the two numbers, and 0 is used for addition. It then jumps to the label *add_sub_logical* in order to prepare to compute the sum.

```
add_logical:
        store_RTE
        li $a2, 0x00000000      #load 0 into $a2
        jal add_sub_logical     #jump and link to add_sub_logic
        restore_RTE
```

Figure 12: Mode loaded for addition

2) *sub_logical*

This procedure loads a mode of 0xFFFFFFFF into $a2. This mode means that subtraction will be performed on the two numbers. It then jumps to the label *add_sub_logical* in order to prepare to compute the subtraction.

```
sub_logical:
        store_RTE
        li $a2, 0xFFFFFFFF      #load 0xFFFFFFFF into $a2
        jal add_sub_logical     #jump and link to add_sub_logic
        restore RTE
```

Figure 13: Mode loaded for subtraction

3) *subtraction*

This short procedure allows for subtraction to be treated as addition when it reaches *add_loop*. Because C = A - B is the same thing as C = A + (-B+1), subtraction can be seen as addition except one operand is getting added to the inverse sign of the second operand.

This procedure also makes the carry bit, which is stored in $t2 a 1, instead of a 0 like it is for addition. This is because in two's complement form, the carry bit for subtraction needs to start off as a 1. The binary ripple carry subtractor below demonstrates this logic.

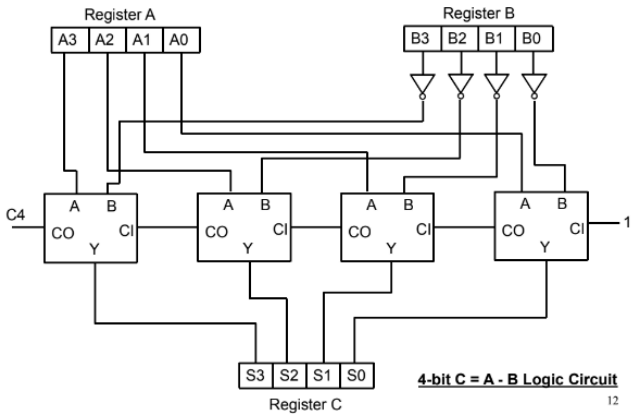

Figure 14: Binary ripple carry subtractor demonstrates subtraction logic

```
subtraction:
        not $a1, $a1
        add $t2,$t2,1
```

Figure 15: Conversion to twos complement for subtraction

4) *add_sub_logical*

This procedure will determine where to branch depending on the mode that is loaded into $a2. If the mode is 0 in $a2, then it will directly branch to *add_loop*, in which the actual addition operation will take place. If the mode indicates that the operands need to be subtracted, then *add_sub_logic* will direct to the *subtraction* label for a few extra steps before it proceeds to *add_loop*. Prior to branching, it sets the index, which will be incremented in *add_loop*, to 0, the sum is set to 0, and the carry bit is set to 0 as this is the carry bit for addition.

```
add_sub_logical:
        store_RTE
        add $t0, $zero, $zero   # ($t0 = index) set to 0
        add $t1,$zero, $zero    # ($t1 = sum) set to 0
        add $t2, $zero, $zero   # ($t2 = carry) bit initialized at 0
        beq $a2, 0x00000000, add_loop  #if $a2 is 0, go directly to add_loop
        beq $a2, 0xFFFFFFFF, subtraction #if $a2 is not 0, needs to go to subtraction
```

Figure 16: Procedure to check if addition or subtraction

5) *add_loop*

This procedure will calculate the sum of the two operands, which are in $a0 and $a1, and return the sum in $v0.

The images below illustrate the basic logic of addition. Corresponding nth bits, starting from the least significant bit and then shifting left, are added together using binary addition. Binary addition involves carry bits, which are bits that are carried from the addition of the prior two bits. The sum is calculated by doing an XOR operation between the two nth bits. An “and” operation is used between the resulting bit of the XOR operation and the carry bit.

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

Figure 17: Binary addition logic[1]

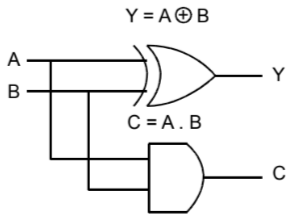

Figure 18: Binary addition logic gates[1]

To get into more detail, this procedure begins by checking to see if the index has reached 32. Because there are only 32 bits in each operand, the binary addition needs to end when it reaches the last bit. If the index is not yet 32, the nth bits of the two operands are extracted. An XOR operation between the two extracted nth bits and another XOR between the resulting bit and the carry bit will provide the output which is then inserted into the nth bit of the sum. To find the carry-out bit which will be used for the next nth bit, an AND operation is performed between the XOR of the nth bits of the two numbers and the carry bit and stored in a temporary register. An AND operation is also performed on the nth bits of the two operands and stored in a temporary register. Finally, an OR operation is done with the values held in the two temporary registers, which results in the carry-out bit used in the next loop cycle. The index is then incremented and the loop calls itself to repeat the process until the index reaches 32.

The diagram below is a ripple carry adder-subtractor, and it demonstrates how addition and subtraction are done using carry bits. The diagram demonstrates the processes of *add_sub_logic, subtraction,* and *add_loop*.

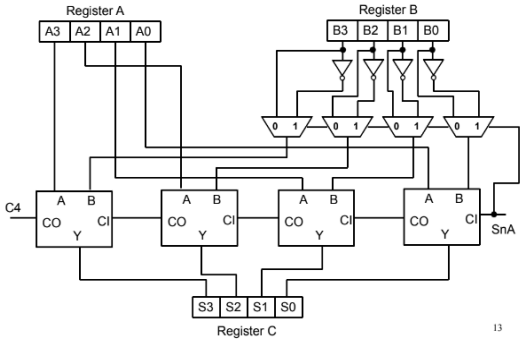

Figure 19: Ripple carry adder-subtractor demonstrates add-sub logic[1]

Finally, once the loop is exited when the index reaches 32, the final sum is loaded into $v0, and the last carry bit is loaded into $v1. This carry-out bit in $v1 is not used in addition or subtraction, but it is used later on for multiplication and division.

```
add_loop:
        beq $t0, 32, exit_loop
        extract_nth_bit($t3, $a0,$t0)
        extract_nth_bit ($t4, $a1, $t0)
        #to find the Y (nth bit that is inserted)
        xor $t5, $t3, $t4
        xor $t6, $t5, $t2
        insert_to_nth_bit($t6, $t1, $t0, $t7)
        #to find the carry out
        and $t7, $t5, $t2
        and $t6, $t3, $t4
        or $t2, $t7, $t6
        addi $t0, $t0, 0x1      #increment index by 1
        j add_loop              #loop again

exit_loop:              #comes here when index = 32
        la $v0, ($t1)   #loads sum to $v0
        la $v1, ($t2)   #carry bit stored in $v1
        restore_RTE
```

Figure 20: Procedure to compute sum of two operands

6) *twos_complement_if_negative*

This procedure checks to see if a number is negative or not. If the number is negative, it will branch to *twos_complement* to convert the number to two's complement. Otherwise, it will move on

```
twos_complement_if_negative:
#use twos_complement
        store_RTE
        #if $a0 !< 0, value in $a0 will be passed into $v0
        move $v0, $a0
        #branch to twos_complement if $a0 < 0
        blt $a0, $zero, twos_complement
        restore_RTE
```

Figure 21: Checking for negative number

7) *twos_complement*

This procedure converts the number to two's complement. This is done by performing a NOT operation on the first number, setting the second number to 1, and adding the two numbers by jumping to *add_sub_logical*.

```
twos_complement:
#use add_logical and "not"
#compute "not $a0" + 1
        store_RTE
        not $a0, $a0    #performs not on first number
        li $a1, 1       #sets second number to 1
        li $a2, 0
        j add_sub_logical       #goes to loop
```

Figure 22: Two's complement converter

9) *twos_complement_64_bit*

This procedure is used in multiplication logic in order to create a 64-bit result when dealing with two 32-bit registers. This is because multiplication has a HI and LO register, each of which are 32-bits.

The procedure takes the arguments $a0 and $a1. The register $a0 will contain the contents of the LO register and $a1 will contain the contents of the HI register. First, both $a0 and $a1 should be inverted to their NOT form, $a0 should be added to 1 using the add procedure defined above, and the resulting carry bit from the previous add operation should be added to $a1, once again using the add procedure.

The register $v0 will hold the LO register's 32-bits of the two's complement result, and the $v1 register will hold the HI register's 32-bits of the two's complement result.

```
twos_complement_64_bit:
        store_RTE
        not $a0, $a0
        not $s0, $a1
        li $a2, 0
        li $a1, 1
        jal add_sub_logical
        move $s3, $v0
        move $a0, $s0
        move $a1, $v1
        jal add_sub_logical
        move $v1, $v0
        move $v0, $s3
        restore_RTE #Macro call to restore frame
```

Figure 23: Procedure to perform two's complement with 64-bit result

10) *bit_replicator / zero_bit_replicator*

This procedure is used for multiplication logic, and its purpose is to replicate a bit 32 times.

These procedures load the register $v0 with either 0x00000000 or 0xFFFFFFFF depending on what the register $a0 holds after going through *twos_complement_if_negative*.

```
bit_replicator:
        store_RTE
        #branches to zero_bit_replicator if $a0 holds value  of 0
        beq $a0, 0x0, zero_bit_replicator
        #if $a0 != 0, $v0 should hold value 0xFFFFFFFF
        li $v0, 0xFFFFFFFF
        restore_RTE

zero_bit_replicator:
#if $a0 = 0, $v0 should hold 0
        li $v0, 0x00000000
        restore_RTE
```

Figure 24: Procedure to replicate a bit 32 times

11) *mul_logic*

This procedure first determines if the two operands need to be converted to two's complement, converts them if they do need to be, and also sets the index, sum, multiplier, and multiplicand prior to entering the loop.

The register $a0 will hold the multiplicand and $a1 will hold the multiplier, or LO register value. The index is initially set to 0 and will be incremented within the loop. Two temporary registers will hold an untransformed version of the multiplier and multiplicand as the original bit pattern is needed later on in the multiplication process and the changed versions of both registers after they go through the loop cannot be used as their values will be different. It will then move on to *mul_loop*.

```
mul_logic:
        store_RTE
        move $s4, $zero # INDEX
        move $s1, $zero # s1 = HI
        move $s2, $a1 # MPLR = LO
        move $t8,$s2 # untransformed $a1 (lo)
        move $s3, $a0 # MCND
        move $t9, $s3 # untransformed $a0 (MCND)
        twos_complement_checker($s2, $s2)
        twos_complement_checker($s3, $s3)
```

Figure 25: Procedure to convert to two's complement prior to multiplication

12) *mul_loop*

This procedure performs multiplication on the two operands, stored in $a0 and $a1, and the resulting product will be stored in $v0 and $v1. The register $v0 will hold the value in the LO register, and $v1 will hold the value of the HI register.

This procedure computes unsigned multiplication. The diagram below illustrates the process used for unsigned multiplication and how it converts two 32-bit values into one 64-bit result.

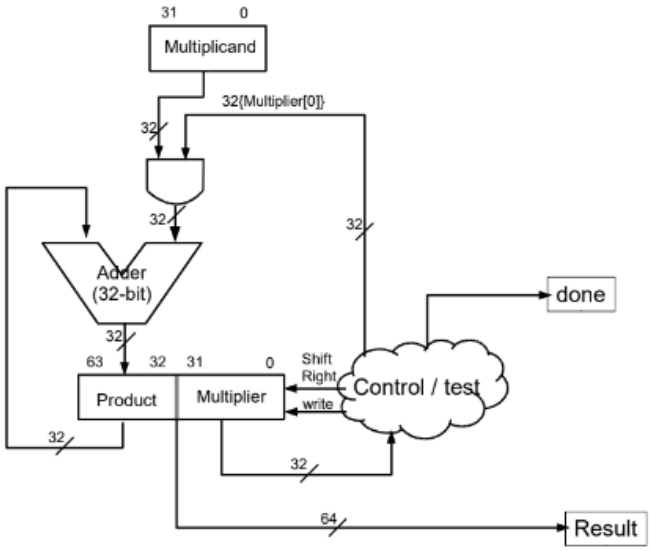

Figure 26: Diagram for unsigned multiplication[2]

First, the procedure *mul_loop* checks the index to ensure it is not 32. If it has not reached 32, it means there is more multiplication to be done. It uses *bit_replicator* to replicate the least significant bit of the multiplier, and then uses AND operation on the replicated bit pattern and the multiplicand. This result is stored in a temporary register and it is later added to the value in the HI register using the add procedure. The LO register value is shifted right as seen in the above diagram. The least significant bit will then be extracted from the updated HI register and inserted into the most significant bit of the LO register, and then the HI register is shifted.

This process of shifting is done in order to incrementally move the two 32-bit register values into a 64-bit result. The index is then incremented and the loop calls itself to repeat the cycle until the index reaches 32.

```
mul_loop:
        beq $s4, 32, exit_mul_loop      #if index = 32, exits loop
        extract_nth_bit($a0, $s2, $zero) # a0 = L[0] = MPLR[0]
        jal bit_replicator # result should be in $v0
        and $t0, $s3, $v0 # x = M & R ; $t0 = X
        move $a0, $s1 # $a0 = HI
        move $a1, $t0 # $a1 = X
        li $a2, 0
        jal add_sub_logical # H = H + X --> result stored in $v0
        move $s1, $v0 # update hi
        sra $s2, $s2, 1 # L >> 1 ; $S2 = L
        extract_nth_bit($t1,$s1,$zero) # $t1 = H[0]
        li $t2, 31 # setting up for insertion (L[32])
        insert_to_nth_bit($t1, $s2, $t2, $t3) # L[31] = h[0] ; $T3
        sra $s1, $s1, 1 # HI >> 1 ; $S1 = HI
        addi $s4, $s4, 1 # I = I + 1
        j mul_loop
```

Figure 27: Unsigned multiplication

13) *exit_mul_loop* (signed multiplication)

This procedure takes care of signed multiplication. Signed multiplication operates differently than unsigned multiplication as it uses the procedure *twos_complement_64_bit* in order to get the complement form of the HI and LO register. The diagram below demonstrates the process of signed multiplication, which utilizes unsigned multiplication but includes the additional steps required for two's complement

Figure 28: Diagram for signed multiplication[2]

The procedure extracts the most significant bits of the original multiplier and multiplicand which were stored in an unaltered temporary register, and an XOR operation is performed between the two resulting bits. The *twos_complement_64_bit* is only performed if the result of the XOR operation is 1, indicating that the number is negative.

```
exit_mul_loop:
        move $v0, $s2
        move $v1, $s1
        li $t0, 31
        extract_nth_bit($t1, $t8, $t0)
        extract_nth_bit($t2, $t9, $t0)
        xor $t0, $t1, $t2
        beqz $t0, done
        move $a0, $v0
        move $a1, $v1
        jal twos_complement_64_bit
```

Figure 29: Procedure to determine if multiplication is signed or unsigned

14) *div_logic*

The *div_logic* procedure will determine if the two operands need to be be converted to two's complement, it converts them if they did, and it also sets the index, dividend, divisor, and remainder prior to entering the loop.

The register $a0 will hold the dividend and $a1 will hold the divisor. The index is initially set to 0 and will be incremented within the loop. Two temporary registers will hold an untransformed version of the dividend and divisor as the original bit pattern is needed later on in the division process and the changed versions of both registers after they go through the loop cannot be used as their values will be different. It will then move on to the *div_unsigned* loop.

```
div_logic:
        store_RTE
        move $s1, $zero # ($t0 = index) set to 0
        move $s2, $a0   #Q = DVND
        move $s3, $a1   #D = DVSR
        move $s4, $zero #R = 0
        move $t8, $a0 # untransformed $a0
        move $t9, $a1 # untransformed $a1
        twos_complement_checker($s2, $s2)
        twos_complement_checker($s3,$s3)
```

Figure 30: Procedure to convert to two's complement prior to division

15) *div_unsigned*

This procedure performs division on the two operands, stored in $a0 and $a1, and the resulting quotient will be stored in $v0, and the remainder will be stored in $v1.

This procedure computes unsigned division. The diagram below illustrates the process used for unsigned division and how it reaches the outputs of the quotient and remainder.

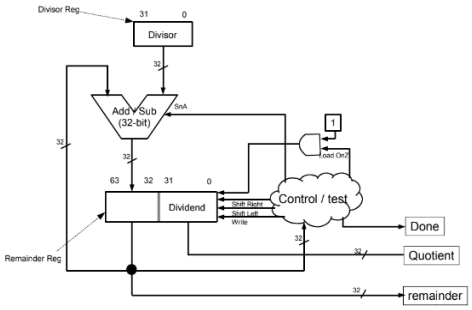

Figure 31: Diagram for unsigned division[3]

The procedure shifts the remainder to the left by 1 and inserts the most significant bit of the dividend to the remainder's least significant bit. It then shifts the dividend. This shifting simulated the dividend getting shifted out as a 64-bit result is being formed. The dividend is then subtracted from the remainder using subtraction logic and stored in a register. If this difference is positive, then a 1 will be inserted in the corresponding location of the remainder. If the difference is negative, then this additional step is disregarded and it simply increments the index and loops again.

The purpose of the shifting process in division is to keep shifting out the dividend until it is completely shifted into the quotient, modeling how 64-bit results for division operations work.

```
div_unsigned:
        sll $s4, $s4, 1 # R<<1
        li $t4, 31
        extract_nth_bit($t5, $s2, $t4) # Q[31]
        insert_to_nth_bit($t5,$s4, $zero, $t6) # R[0] = Q[31]
        sll $s2, $s2, 1 # Q << 1
        la $a0, ($s4)
        la $a1, ($s3)
        li $a2, 0XFFFFFFFF
        jal add_sub_logical # S = R-D
        move $s5, $v0 # $t7 = S (R-D)
        bltz $s5, extra_step # if S < 0 --> go to extra step
        move $s4, $s5 # R = S
        li $s7, 1
        insert_to_nth_bit($s7, $s2, $zero, $t6)

extra_step:
        addi $s1, $s1, 1
        beq $s1, 32, div_done
        j div_unsigned
```

Figure 32: Unsigned division

16) *div_done* (signed division)

This procedure takes care of signed division. Signed division works differently from unsigned division because it uses the macro *twos_complement_convert* in order to get the result of the quotient and remainder in its two's complement form. The diagram below demonstrates how unsigned division works, utilizing unsigned division but including extra steps to convert the numbers.

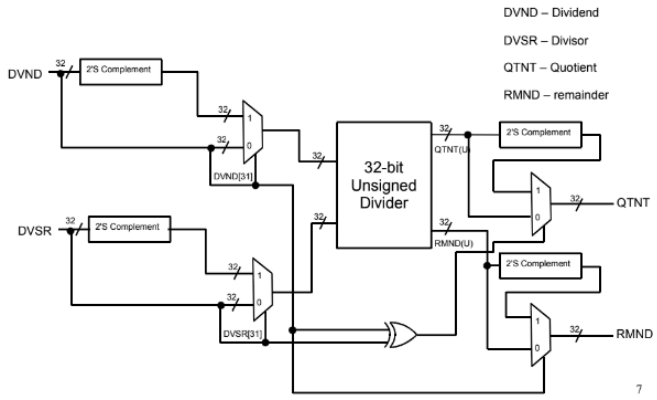

Figure 33: Diagram for signed division[3]

The procedure extracts the most significant bits of the original dividend and divisor, which were stored in an unaltered temporary register, and an XOR operation is performed between the two resulting bits, similar to the process for signed multiplication. If the result of the XOR is 1, then the quotient has to be converted to two's complement through the macro *twos_complement_convert*. If the most significant bit is a 1, then the remainder is also converted to two's complement. The quotient is then loaded into $v0 and the remainder is loaded into $v1, and the procedure ends.

```
div_done:
        # ---- determine signs of Q and R, etc.
        li $t0, 31
        # Q
        extract_nth_bit($t1, $t8, $t0)
        extract_nth_bit($t2, $t9, $t0)
        xor $t0, $t1, $t2 # $t0 = S of Q
        move $s0, $t1
        beq $t0, 1, sign_q
        j check_r

sign_q:
        twos_complement_convert($s2,$s2)

check_r:
        beq $s0, 1, sign_r
        j finish

sign_r:
        twos_complement_convert($s4,$s4)

finish:
        move $v0, $s2
        move $v1, $s4
        restore_RTE
```

Figure 34: Procedure to determine if division is signed or unsigned

IV. Testing

Prior to testing your results, it is important to check that the frame has been stored and restored properly so that the registers that are holding important values are not carried on to other procedures and compute operations incorrectly. A common mistake is that your values can get lost when jumping to different labels or changing the values in temporary and saved temporary registers. This can also cause infinite loops as the value of the register is not changing the way it should. It could cause runtime errors as well in which it goes out of bounds.

To test the program, first the files have to be assembled. Because the "Assemble all files in directory" option is chosen in Settings, each file does not have to be individually assembled. The assemble button looks like a wrench at the bar at the top of MARS. Then, to run the program, choose the green button with the arrow.

The results indicate whether or not your logical and normal procedure answers match, and this is determined by the *proj-auto-test.asm* file. The operation is computed for each set of operands in both the normal and logical procedures. The results are printed for both procedures. Then the answers are matched up with each other; if the answers are the same, it means both the logical and normal procedure work correctly despite being implemented in different ways. The overall results are at the bottom of the message, and a 40/40 indicates that all test cases passed.

```
(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]

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

-- program is finished running --
```

Figure 35: Testing results passed

V. Conclusion

This project helped me understand how to apply the information learned in class to a real life example. Prior to this class, I never thought about the internal operations of a computer, and prior to this project, I never thought about how computers cannot just add, subtract, multiply, and divide the way we do. It was very interesting to learn about how logic gates are used to perform mathematical calculations.

This project specifically helped me understand the purpose of the frame and why it needs to be stored and restored. I ran into problems many times because my registers were not being stored and then I would use them in another procedure which would change its values and produce an incorrect output. I also learned the importance of defining macros in order to avoid repeating the same lines of code throughout procedures. It helped save a lot of time and made my code more efficient/easier to read.

This project also helped me appreciate my computer. It is easy to take it for granted because it does all these operations discreetly, but it is eye-opening to know that our computers are performing all these functions plus more, and it does them so quickly and it gives us accurate results every time. Implementing the normal procedure and the logical procedure helped me see that although it is easier and more efficient to use helpful tools and methods to make a task easier, it is also important to learn the actual process in order to truly understand how and why it works the way it does. It is also very fascinating to learn how the computer is able to perform so many different functions and operations using just 0's and 1's. I look forward to experimenting more with MIPS and MARS and understanding more about the processor's functionalities.

References

[1] K. Patra. CS 47. “Addition / Subtraction Logic”. San Jose State University, San Jose, CA. 04 May, 2023.

[2] K. Patra. CS 47. “Multiplication Logic”. San Jose State University, San Jose, CA. 04 May, 2023.

[3] K. Patra. CS 47. “Division Logic”. San Jose State University, San Jose, CA. 04 May, 2023.