

# Arithmetic Implemented By MIPS Logic Operations

Edmond Lin

San Jose State University

edmond.lin@sjsu.edu

Abstract-- This report will focus on the Implementation of basic arithmetic operations (addition, subtraction, multiplication, and division) using: 1) MIPS standard operation, And 2) using logical procedures in MARS (MIPS Assembler and Runtime Simulator).

## 1. Introduction

This guide will grant users the ability to calculate basic arithmetic operations by using the standard MIPS library set operations or by the implementation of logical procedures implemented in MARS. After Implementation, the user will be able to test their results to see if both their normal and logical operations match, if the test passes, the user will get a score of 40/40. Many advanced hardware revolve around basic arithmetic, and that is why it is vital to learn the functionalities of how arithmetic digit circuits work. This project will imitate a processor's digit circuitry when doing arithmetic operations through MARS. With basic knowledge, It can be very useful for any engineer or scientist to improve digital circuits which can lead to passing faster electrical signal travel time resulting in a decrease in runtime while running programs.

## II. INSTALLATION AND SETUP

### A. Installation of Simulator

MARS is free to download, follow the link provided

<https://courses.missouristate.edu/KenVollmar/MARS/download.htm>. Unzip the MARS4\_5.jar file and download MARS.

### B. Load the Project into Simulator

Download the zip file provided from Canvas at <https://sjsu.instructure.com/courses/1324477/assignments/5056985> and unzip it. The file should contain five different files:

1) cs47common\_macro.asm

2) cs47\_proj\_alu\_logical.asm  
3) cs47\_proj\_alu\_normal.asm  
4) cs47\_proj\_macro.asm  
5) cs47\_proj\_procs.asm  
6) cs47\_proj-auto-test.asm

Proceed to open MARS, click on the file icon (located on the top left corner of the simulator), and navigate to the unzipped CS47Project file that contains all six .asm files that will be needed to do this project.

The three files that we will be writing code in will be:

1) cs47\_alu\_logical.asm

This file will contain the macros that will be used in both the normal and logical procedures.

2) cs47\_proj\_alu\_normal.asm

This file will contain the normal arithmetic procedure using MARS instruction set.

3) cs47\_proj\_alu\_logical.asm

This file will contain the logical arithmetic implementation using logical procedures.

### C. Setting the simulation tool

Locate Mars "Settings" menu bar (fourth column), keep the default checked settings, and click on 'Initialization Program Counter to global "main" if defined.' After clicking a check mark should appear.

## Boolean Algebra

A boolean variable assumes two values true or false.

True is denoted as 1 and false is denoted as 0.

There are three basic boolean algebraic operations AND (.), OR (+), NOT('). Boolean Algebra is needed for digit logic arithmetic handling.

## AND OPERATION

## III. Requirements of Normal Arithmetic Procedure

In this project, there will be two types of arithmetic operations, normal and logical procedures. As discussed previously, both implementations will allow the user to perform addition, subtraction, multiplication, and division operations.

#### A. Normal Procedure

The normal procedure is to be completed under the `cs47_proj_alu_normal.asm` file where the procedure will use MIPS basic instruction set. MIPS has a standard arithmetic operations, and we will be using these to do the mathematical expressions.

##### 1) Register \$a0

This argument is the first operand in the mathematical expression

##### 2) Register \$a1

This argument is the second operand in the mathematical expression

##### 3) Register \$a2 is the operator provided in ASCII code and it determines whether it is an add, subtract, multiplication, or division operation.

##### 1) Register \$v0

The results of the addition and subtraction operations will be stored into register \$v0, because it will result in a 32-bit yield. For multiplication and division it will result in a 64-bit result which requires two registers to hold all 64 bits. In multiplication, \$v0 will hold the 'LO', the lower 32 bits of the result, and for division it will hold the quotient.

##### 2) Register \$v1

During multiplication, \$v1 will be used to hold the upper most significant 32 bits. And in division, \$v1 will hold the remainder 32 bits. Two registers hold total 64 bits, as wanted.

#### MIPS NORMAL INSTRUCTION FUNCTIONALITY

The normal procedure requires the operations "add", "sub", "mul", and "div." First the user must move the stack frame down by 12 spaces which allows the user to store the frame pointer and return address for each function call. By using the branch instruction provided by MIPS, MARS will recognize the operator symbol '+', '-', '\*', '/'.

##### 1) Operator '+'

Represents "add" in MIPS and once '+' symbol is recognized as an operand it will jump into the to normal add procedure and return the sum.

##### 2) Operator '-'

Represents "sub" in MIPS and once '-' symbol is recognized as an operand it will jump into the normal subtraction procedure and return the difference,

##### 3) Operator '\*'

Represents "mul" operation in MIPS and once '\*' symbol is recognized as an operand it will jump into the multiplication result.

##### 4) Operator '/'

Represents "div" operation in MIPS and once '/' symbol is recognized as operand it will jump into division procedure returning the quotient and remainder.

##### 1) Additional Implementation

Adds "\$a0" and "\$a1" operands values and returns the sum in \$v0.

##### 2) Subtraction Implementation

Subtracts "\$a0" and "\$a1" operands and returns the difference in \$v0.

##### 3) Multiplication Implementation

Multiplies "\$a0" and "\$a1" operand value, and returns a 64 bit value. Upper 32 bits will be stored in special registers "HI" and lower 32 bits in "LO" register. To move the values into \$v0, and \$v1 "mfhi" and "mflo" operation MIPS instructions are used. "mfhi" moves the "HI" register value into \$v0 and "mflo" operation moves "LO" register value into \$v1.

##### 4) Division Implementation

Divides "\$a0" and "\$a1" operand values, and returns a 64 bit value. Quotient is stored in "LO" and the remainder will be stored in "HI". The 32-bit value in "LO" will need to move with "mflo" operation provided by MIPS to replace \$v0 register with corrected remainder value, and "mfhi" operation to replace \$v1 with corrected remainder value from "HI" register.

When the function call is finished calling, meaning the procedure is completed and results are returned, the stack frame must be restored. Load the frame \$fp and \$ra. Move the stack pointer back up and call jr \$ra which jumps the user back to the next instruction of "caller" instruction.

#### Project Macros

In this portion, we will focus on the macros used in this project. Macro are pattern matching and replacement facilities, meaning they match the string

pattern of the sequence of code. Once the name is defined by the programmer, the macro is called. It is important to note that macros do not serve as function calls, but are just shells of sequences of codes. It's main serve is to make code cleaner, by reducing the size of code. Each time a macro is called the code is still stepped through, it is just for visualization in the main code calling that macro that makes it seem shorter since it will only have to refer to that one line of macro initialization versus the many lines of code actually hidden in the macro call section.

#### 1) .macro extract\_nth\_bit(\$regD, \$regS, \$regT)

The purpose of this macro is to extract the nth bit from a specified bit pattern. In order to do this, the macro will have to take 3 registers \$regD, \$regS, \$regT. \$regD represents the destination register and ultimately will contain 0x0 or 0x1 in the 0th bit position. \$regS is denoted for source pattern, the code pattern a user supplies. And \$regT is the amount of bits the pattern is shifted, \$regT controls which value in the 32 bit will be extracted. To do in MARS assembly, we must move the "source" bit pattern into a temporary register, example shows \$t0. Then a user must shift the temporary register holding the source by the amount wanted which is the number inside \$regT, and saving that value back into \$t0. Then after the shifted amount, the value wanted is in the LSB(0th position). This is where a user assigns a mask, where the last value is 1 and a and operation will be used. Since AND is only 1 when both values A and B are 1 the right most bit will be either a 0 or 1 depending on the shifted value while the rest of the higher 31 bits will remain 0 indefinitely.

```
.macro insert_to_nth_bit($regD, $regS, $regT, $maskReg)
    li $maskReg, 1
    sllv $maskReg, $maskReg, $regS
    not $maskReg, $maskReg
    and $regD, $regD, $maskReg
    sllv $regT, $regT, $regS
    or $regD, $regD, $regT
.end_macro
```

#### 2) .macro insert\_to\_nth\_bit(\$regD, \$regS, \$regT, \$maskReg)

The purpose of this macro is to insert the value bit 0x0 or 0x1 into a bit pattern. \$regD represents the bit pattern in which 1 or 0 will be inserted in the nth position, \$regS is which position the bit to be inserted (0-31), \$regT is the register that contains the 0x1 or

0x0 bit value to be inserted, and \$maskReg is just a register to hold the temporary mask that will be used.

To do in MARS, first set the mask value to 1 (\$maskReg) by loading the value 1 into \$maskreg. Then shift by using the "sllv" operation which shifts the mask to the position to what \$regS is. After \$maskReg is set to the corrected insert bit position, invert \$maskReg by using calling the "not" operation on \$maskReg and store it back into \$maskReg, Then perform "and" to \$regD and \$maskReg store back in \$regD. All these operations have allowed us to replace the insertion bit value with "0".

```

      0
      V
1 1 0 0 1 1 0 0 <---- D , n=3, b = 0
-----
0 0 0 0 0 0 0 1 <---- M == Mask
0 0 0 0 1 0 0 0 <---- M = M << n
1 1 1 1 0 1 1 1 <---- M = !M
& 1 1 0 0 1 1 0 0 <---- D
-----
1 1 0 0 0 1 0 0

```

From this illustration, we can see that we are changing the bit position labeled V to 0 by performed the shift and inverted operations.

Now we shift left \$regT(actual value 0x0 or 0x1) by the insertion value stored in \$regS. We perform a or operation, and since we have updated \$regD to have a '0' bit in the insertion position, the insert bit position will be updated with the value in \$regT, because in an OR operation as displayed in a previous diagram since boolean value 'A' is indefinitely 0 it will rely on value 'B' to either a 0 or 1. If it is 1 insert value will be updated it with 1, if it is 0 it will remain the same.

#### 3) .macro saved\_stack

This macro pushes the stack down, and moves the stack frame up to store \$fp, \$ra, all argument values (\$a0-\$a3), and saved registers (\$s0-\$s7). Everytime a function needs registers saved in the stack, we will call this macro, and since this project does not take in account of the memory usage, we are allowed to store these registers to avoid confusion of which registers to save.

#### 4) .macro restored\_stack

After saving all the stack once the procedure is done, we will need to push the stack frame back up, restore all registers by loading the values \$fp, \$ra, all arguments values, and saved stack values (\$sp) back out of the stack.

## B) Implementation Details

### Utility Procedures

#### 1) twos\_complement and twos\_complement\_if\_neg

```
# Utility Procedures
# Two's complement ~$reg + 1
twos_complement:
    saved_stack
twos_complement_no_revert:
    not $a0, $a0                # Inverse :
    li $a1, 1                  # move 1 in
    jal add_logical             # add $a0 i
    restored_stack
# If $a0 is < 0 then return pos value by using two's comp
twos_complement_if_neg:
    saved_stack
    bltz $a0, twos_complement_no_revert #less than
    move $v0, $a0                # move invi
    restored_stack
```

Twos\_complement takes the argument \$a0 and returns the two's complement of \$a0 in \$v0. To compute a two's complement, we will invert the argument \$a0, set \$a1 to 1. We will use the adder logical here, which will be discussed later. The adder logical and the add normal procedure serve as both the same functionalities just different ways of implementation, and the result is the sum of inverted \$a0 and 1 which is the twos\_complementary format.

Twos\_complement\_if\_neg relies on the twos\_complement it checks if the value \$a0 is less than 0 or not, if the \$a0 is less than 0 then it will jump to the twos\_complementary procedure which will take negative binary value and return a positive value. If the bit pattern is already positive, move that value into the \$v0 return register.

#### 2) Twos\_complement\_64bit

This procedure creates the twos\_complement form for 64-bits hi and lo for multiplication. It takes arguments \$a0 (LO of the number), and \$a1 (HI of the number). It returns \$v0 the LO part of the 2's complement 64 bit, and \$v1 returns the HI part of the 2's complement 64 bit. This is possible in MARS by inverting \$a0 (LO value) and adding the value 1 into the add\_logical function, which returns a 32 bit LO two's complement with the overflow carry bit in \$v1. The carry bit is then added to the inverted HI register to get the

complement form. The complement form will be returned in \$v0, and \$v1 all 64 bits.

#### 3) bit\_replicator

If the \$a0 is equivalent to 0 then jump to rep\_zero which returns 0x0 in \$v0, and if it is not 0 then return 0v1 in \$v0.

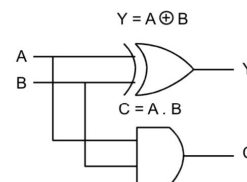
### Addition and Subtraction Logical Implementation

Both addition and subtraction logical will rely on a full adder circuit application, and full adder replies on half adder to make add logic functionality work.

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



In this illustration a half adder is implemented which involves two input bits. No carry in bit is considered. It results in one sum bit and one carry out bit. Logic replies on an XOR operation two inputs result in sum (Y), and a AND operation outputs carry (C). It is good to note that the full adder is comprised of two half adders, the fuller adder will continue to used (Y) and (C) from the first half adder into the second half adder to get (Y: full sum) (CO: carry out).

Binary Three Single Bit Addition Result

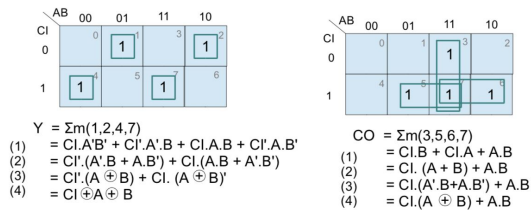
	Bit 1 (Ci) Carry In	Bit 2 (A)	Bit 3 (B)	Sum Bit (Y)	Carry Bit (CO) Carry Out
m0	0	0	0	0	0
m1	0	0	1	1	0
m2	0	1	0	1	0
m3	0	1	1	0	1
m4	1	0	0	1	0
m5	1	0	1	0	1
m6	1	1	0	0	1
m7	1	1	1	1	1

Full Addition

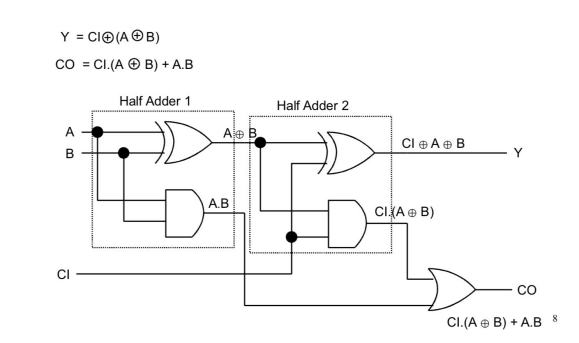
$$Y = \Sigma m(1,2,4,7)$$

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

This table determines the combinational circuit implementation. Find the minterms that cause the outputs to be 1 (sum), and the outputs that represent the carry out represented in POS format.



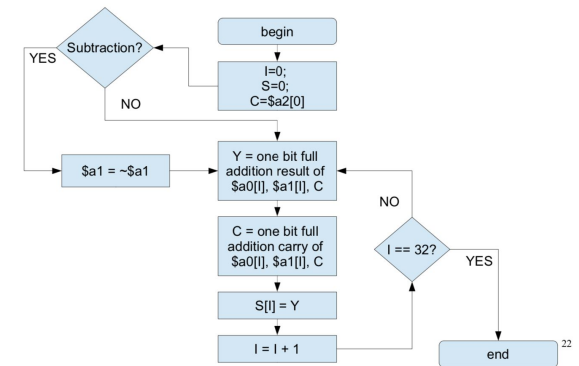
After we identify the POS format, we will use K-Map operation to simplify the logical operations, and after simplification we have resulted in  $(C_i \oplus A \oplus B)$  for the sum and  $(C_i(A \oplus B) + A.B)$  for the carry out bit. We will need to imitate this circuitry logic in MARS to develop the full adder circuit.



From this diagram we can see a clear view of the XOR and AND logicals being used. First half adder is implemented as a middle ( $Y: A \text{ xor } B$ ) and ( $C: A.B$ ) ground to get the values to get the Full adder SUM ( $Y$ ) and the CARRY ( $CO$ ).

Now that we understand how we got the full adder logic, we will follow the algorithm chart and implement the add logical in MARS which the sub logical also relies on.

### Write common Add/Sub



To implement this in software, we will follow this flow chart. To begin, MARS must first identify whether the

operation is an addition or subtraction. If the operation is positive the the first carry in bit is set to 0, and if the operation is subtraction 2's complement will set \$a1 into 2's complement format making the first carry in bit value 1. Then we will extract\_nth\_bit position to perform the half adder by using the LSB of the extracted operand values at nth position and the specified carry in value. That will give the values in the partial half adder sum, and we will use those values to complete the full adder logic. Once we have the corrected extracted summed or differences bit we will insert it into register \$v0 which is the return register, and \$v1 will be updated with the updated carry in bit at the end of the full adder logical. Then we must repeat this ripple process for all 32 bits. Everytime iteration, the carry bit will be updated and sum bit will be saved in \$v0, which will lead to all 32 bit summed or difference value, with the last overflow bit saved in register \$v1.

#### add\_logical:

```

saved_stack
li $s0, 0
li $v1, 0
li $v0, 0
jal add_sub_logical
restored_stack

```

#### add\_sub\_logical:

```

li $s1 32
beq $s0, $s1, exit_add_sub
extract_nth_bit($s2, $a0, $s0)
extract_nth_bit($s3, $a1, $s0)
xor $s4, $s2, $s3
and $s6, $s2, $s3
and $s7, $v1, $s4
xor $s5, $v1, $s4
or $v1, $s7, $s6
insert_to_nth_bit($v0, $s0, $s5, $t1)
addi $s0, $s0, 1
j add_sub_logical
exit_add_sub:
jr $ra

```

#### 1) Logical Add Operation

Initialize register \$s0 as the counter value, \$v1 as the first carry bit value (0), and initialize \$v0 as the updated sum bit that will return the full add value. Jump in the add\_sub\_logical procedure which was extract the LSB of of operand \$a0 and \$a1. Period the full adder operation which was output the sum of two bits \$a0 and \$a1 depending and updated carry in bit depending on the nth position. Insert the sum bit into \$v0 using insert macro, and carry in updated every iteration of add\_sub\_logical. Keep branching to add\_sub\_logical until all 32 bits are completed. Final

value should be the sum of all 32 bits from operand \$a0, and \$a1 in \$v0. Final carry in bit which is saved in \$v1 saving the overflow value.

```
sub_logical:
    saved_stack
    move $s0, $a0
    move $a0, $a1
    jal twos_complement
    move $a0, $s0
    move $a1, $v0
    jal add_logical
    restored_stack
```

## 2) Logical Sub Operation

Uses the same add\_sub\_logical procedure as add\_logical, but the initial carry in bit is set to 1, and \$a1 value is inverted + 1 (twos\_complement format). Returns sub value of all 32 bits in \$v0, and last carry in bit is saved in \$v1 if overflow occurs.

## Logical Multiplication Operation

This is logical implementation of multiplication, Which will take the operand values \$a0 (Multiplicand) and \$a1 (Multiplier) multiply the two numbers return a 64-bit product which will be stored in \$v0 (Lo part of the result) and \$v1 (Hi part of the result).

### 1) Sign Handling

In the logical multiplication procedure we have to handle signed values. To do this, we must take the two's complement values of a negative binary number, and change it into a positive number. Save the positive operands into new registers. Then we must extract the MSB of the two original operands. From there from figure 12 we can see that the sign assignment matches that of an XOR logical, so we are allowed to use the XOR logical between the MSB of the original operand (\$a0) with the second operand (\$a1) to determine the sign value. We will later apply that sign value to the result of the unsigned 64-bit output that use the two binary positive numbers as operands. Illustration \_ shows explicitly why

```
// Sign assignment
if (A[31] == 1) and (B[31] == 1) then P = P1;

// P = INV(P1) + 1 => 2's complement of P1
if (A[31] == 1) and (B[31] == 0) then P = -P1;

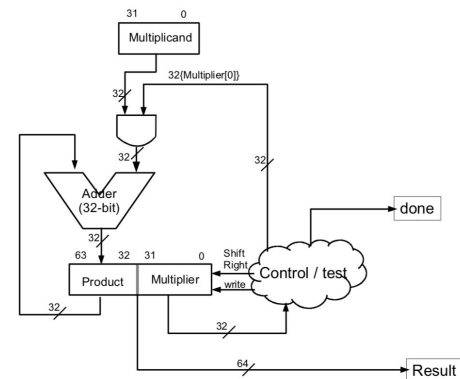
// P = INV(P1) + 1 => 2's complement of P1
if (A[31] == 0) and (B[31] == 1) then P = -P1;

if (A[31] == 0) and (B[31] == 0) then P = P1;
```

A[31]	B[31]	P's sign (S)
0	0	0
0	1	1
1	0	1
1	1	0

$S = A[31] \text{ xor } B[31]$

## Simplified Sequential Multiplier



### 2) Simplified Unsigned Circuit

For the simplified multiplier, we store the multiplicand in a 32-bit register. Store the product in a higher 32 bit, and multiplier in a lower 32 lower bit. The product and multiplier are connected in a sense, because the product LSB of the product will be shifted into the MSB of the multiplier, and the multiplier will be shifted out of scope. We will use a 32-bit adder to add every multiplier and multiplicand result and store in into the product. Each iteration we take the LSB of the multiplier and replicate that bit 32 times. The value will be a 0 or 1 replicated 32 times to be multiplied with the multiplicand resulting into a partial product that will be added into the product register, which will use a 32-bit adder, and then the product and multiplier will be shifted right after the multiplier and product has been updated. Repeat 32 times.



## Signed Multiplication Circuit

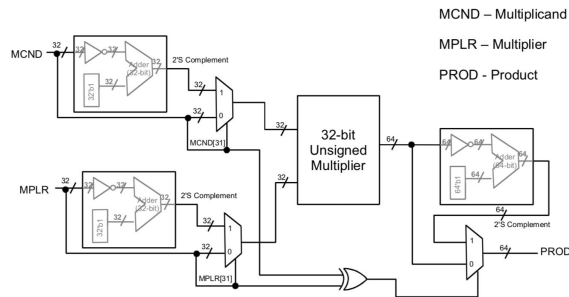
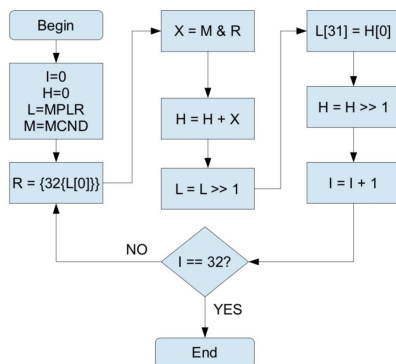


Figure 13, shows us the digital circuitry for the signed multiplication circuit. This shows what I have explained in signed handling portion, two's complement will change the value of the operands if it is negative into positive. MSB of the original operands are saved, and signed value is determined logically using an XOR gate. Once the value of the 64-bit unsigned value, we use a 64-bit two's-complement circuit that uses a 64 bit adder, to make a two's-complement form. We will pick up the negative or positive 64-bit value depending on the sign bit we have determined from the MSB and LSB of the original operand using XOR logical.

Now we will follow the flow chart, and implement the multiplication circuit into MARS.

## Unsigned Multiplication



First, we will need a saved register to store the counter, product (H), multiplier (L), and multiplicand (M). We will then extract the LSB of the multiplicand and replicate it 32 times. Then we use an AND logical between the 32-bit LSB (1 or 0) with the multiplicand. Multiplicand stays constant throughout the entirety of

the loop. We will then add the partial products to the product register (HI result). Then we will shift the multiplier register by 1 pushing the LSB of the multiplier out of scope. Save the LSB of the product register and set the MSB of the multiplier register (LO result) to the LSB of the product register. Shift H (product HI register) by 1, and increment the counter by 1. Loop until counter equals 32. By the end of the 32 bits, the initial multiplier values will be shifted out, and what is left is the lo 32-bit and the hi 32-bit which is what we desire. We keep shifting and adding following this pattern until we get the final product result.

```

mul_logical:
    saved_stack
    jal twos_complement_if_neg
    move $s0, $v0
    move $a0, $a1
    jal twos_complement_if_neg
    move $a0, $s0
    move $a1, $v0

mul_unsigned:
    li $s0, 0
    li $s2, 0
    move $s1, $a1
    move $s3, $a0
    lw $a0, 52($sp)
    lw $a1, 48($sp)
    li $s6, 31
    extract_nth_bit($t7, $a0, $s6)
    extract_nth_bit($t8, $a1, $s6)
    xor $s7, $t7, $t8

extract_first_bit_mplr:
    extract_nth_bit($a0, $s1, $zero)
    jal bit_replicator
    and $s4, $s3, $v0
    move $a0, $s2
    move $a1, $s4
    jal add_logical
    move $s2, $v0
    srl $s1, $s1, 1
    extract_nth_bit($s5, $s2, $zero)
    li $s6, 31
    insert_to_nth_bit($s1, $s6, $s5, $t1)
    srl $s2, $s2, 1
    addi $s0, $s0, 1
    li $s6, 32
    beq $s0, $s6, exit_multloop
    j extract_first_bit_mplr

exit_multloop:
    move $v0, $s1

    move $v1, $s2
    beqz $s7, return_mult
    move $a0, $v0
    move $a1, $v1
    jal twos_complement_64bit

return_mult:
    restored_stack
  
```

Here is the actual code from MARS in figure \_ , my first step was to change the two operands given in \$v0, and \$v1 to both positive numbers by using the twos\_complement\_if\_neg macro which was already established. If you refer back to twos\_complment\_if\_neg macro it takes the values in \$a0 to be two's complement, and returns the value in \$v0. We must save the result of the \$a0 complemented form temporarily, so we can put \$a1 in to \$a0 and get the two's complement form, and at the end we restore \$a0 and \$a1 with the positive binary integers.

Note: We will follow the registers used in the picture with the multiplication code (Figure ), but users are not limited to these registers.

Then we can move on the unsigned portion of the multiplication, where we make registers to hold the counter (\$s0), product (\$s2 HI result), multiplier (\$s1 LO result), and multiplicand (\$s3 constant). We will then take the original \$a0, and \$a1 two values take their MSB values, and use an xor logical to get the sign bit result and save it into a register. We will enter the extract\_first\_bit\_mplr loop to extract the LSB of the multiplier and replicate it 32 times. We will and the 32-bit (0 or 1) with the multiplicand. Add the partial product to the product to \$s2 (product). Shift right logical multiplier (\$s1). Extract the LSB of the product and move it into the MSB of the multiplier. Shift the product by 1, and increment the loop. When all 32 bits are done looping move the hi and lo values into return registers \$v0, and \$v1. We need to check whether or not we need to return the twos\_complement 64 bit of the result or not. To do this we refer back to the saved sign bit which will either have a 1 or 0 saved. If it is equal to 0 just turn the mult as is, if it is 1 convert the twos\_complement\_64 bit result by jumping into twos\_complement\_64bit macro.

#### Logical Division Implementation

This is the logical division implementation, which is a replication of the normal division. It divides the dividend (\$a0), and the divisor (\$a1), and returns that result into \$v0 (quotient) and \$v1 (remainder). Total 64 bit, quotient and remainder split into 2 registers.

#### Sign Handling of Logical Division

For the sign handling we will use the same methodology as we used for the multiplication sign.

We will XOR the MSB of the dividend(\$a0) and divisor(\$a1). Save that sign bit value into a save register, 1 meaning it is negative, and 0 meaning it is positive. We will only use the saved sign bit value for the quotient (\$v0). For the remainder, the signed value is will be the same as the dividend so to determine the remainder sign value, we will extract the MSB of the dividend resulting in (1 pos, 0 neg). We will have to convert both the dividend and the divisor into positive numbers and use an unsigned divider logical before we add back the sign logic and the end where we have the unsigned result.

#### Simplified Binary Division Circuit

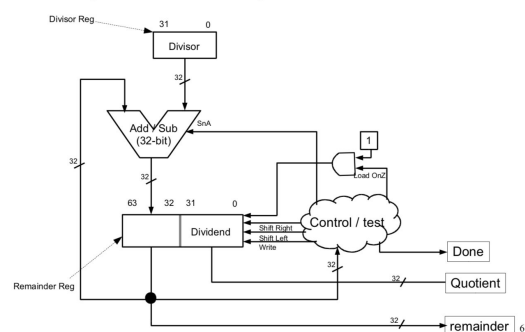
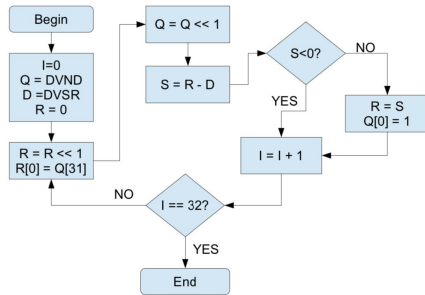


Figure \_ shows the simplified binary division circuit. We need to store the divisor into a 32 bit register (stays constant), and we need two registers for the upper 32 bit (remainder) and lower 32 bit (quotient). We will first place the dividend into lower 32 bit that will later house the quotient once the divisor loop has been completed. In this circuit we will have a left and right shift capability, we will have to left shift and do a subtraction between the remainder and divisor each iteration. If the value is negative, we will "roll back" which means we return back to the shifted remainder and quotient value iterate and continue shifting until the difference between the remainder and divisor is a positive number. Once the remainder is a positive between the differences, we will need to shift out the 0 in the LSB, so we need to right shift the remainder and quotient and insert a 1. And this why we need a control that shifts left and write with the capability to write into the remainder register. Repeating the process 32 times.



## Unsigned Division



10

We will follow this unsigned divisor diagram. It calls for a register to hold the counter (I), dividend (Q), divisor(D), and remainder (R). We will left shift the remainder, and extract the MSB of the divisor (quotient register) and insert it into the LSB of the remainder register. Then we will left shift the quotient register, and use the subtraction logical that we have implemented previously to subtract the remainder and the divisor. If the value is not less than 0, we will set the remainder to the subtracted result, and then we will take the LSB of the quotient register and insert 1. If the value is less than 0 then we proceed to increment and continue shifting left each iteration until we a difference that is not negative. We keep incrementing until we have reached a difference value between the remainder and dividend that is positive. Each time that it is a positive difference, the remainder is set to the differenced value, and the quotient is inserted with 1. Repeat this process 32 times, and we will end up with the unsigned reminder (higher 32 bit register) and the quotient (lower 32 bit register).

```

div_logical:
    saved_stack
    jal twos_complement_if_neg
    move $s0, $v0
    move $a0, $a1
    jal twos_complement_if_neg
    move $a0, $s0
    move $a1, $v0

unsigned_div:
    li $s0, 0
    move $s1, $a0
    move $s2, $a1
    li $s3, 0
    lw $a0, 52($sp)
    lw $a1, 48($sp)
    li $s6, 31
    extract_nth_bit($t7, $a0, $s6)
    extract_nth_bit($t8, $a1, $s6)
    xor $s7, $t7, $t8

div_loop:
    sll $s3, $s3, 1
    li $s6, 31
    extract_nth_bit($t1, $s1, $s6)
    insert_to_nth_bit($s3, $zero, $t1, $t2)
    sll $s1, $s1, 1
    move $a0, $s3
    move $a1, $s2
    jal sub_logical
    move $s4, $v0
    bltz $s4, counter
    move $s3, $s4
    li $s6, 1
    insert_to_nth_bit($s1, $zero, $s6, $t2)

counter:
    addi $s0, $s0, 1

    li $s6, 32
    beq $s0, $s6, exit_div
    j div_loop

exit_div:
    move $v0, $s1
    move $v1, $s3
    beqz $s7, sign_remainder
    move $a0, $s1
    jal twos_complement
    move $s1, $v0

sign_remainder:
    beqz $t7, skip
    move $a0, $s3
    jal twos_complement
    move $s3, $v0

skip:
    move $v0, $s1
    move $v1, $s3
    restored_stack
  
```

Figure \_ shows the logical division circuit being implemented into MARS.

To implement the division circuit into MARS , we will have to change the divisor and quotient into positive values if it is a negative value by using the twos\_complement\_if\_neg macro. Move the positive

two's complement values in the appropriate \$a0 (dividend) and \$a1 (divisor) registers. We will have counter register (\$s0), \$s1 (unsigned dividend), \$s2 (unsigned divisor), and \$s3 (remainder). We will extract the MSB of the original dividend (\$a0) and the divisor (\$a1), and perform an XOR operation to save the sign bit in a temporary register (\$t7 & \$t8) for the quotient once we are done with unsigned division logical. We will also use the MSB of the dividend to correct our sign for the remainder (stored in \$t7). Shift the dividend register left logical ((\$s3) quotient when completed) by 1. Extract the MSB of the quotient register, and insert it into the LSB of the remainder register. Left shift the quotient register by 1 (\$s3). Now compute the difference between the remainder and divisor by setting the values of \$a0 (remainder) and \$a1 as the (divisor) then jump into the sub\_logical procedure which will return a value which is either negative or positive. Check if that value is less than zero or not with the bltz operation which is provided by MIPS instruction set. If the difference is less than 0 then jump to the counter loop and continue iterating until counter is 32 repeating the processes listed above. If the difference is more than 0 then, set the remainder value to be the difference between the remainder and the divisor, and add 1 to the LSB of the quotient (this is shown in the previous sections diagram). After looping 32 times, we will end with the unsigned 32 bit remainder, and 32 bit quotient. We will exit the unsigned\_div loop and move the unsigned quotient in \$s1 into \$v0, and move the unsigned remainder in \$s3 into \$v1. Here we will need to handle the 32 twos\_complement values of the quotient and remainder. We will use a branch statement similar to the multiplication procedure, if the signed register result is 1 return the twos\_complement form by jumping into the twos\_complement procedure which will change the quotient to a negative value. If it is 0 keep the positive value, it is fine as is, and jump to the sign\_remainder to get the sign value for the remainder which is determined by the MSB of the dividend (we saved this in temporary register when we initialized the hold values). If the MSB of the dividend is 0 then skip the twos\_complement of the remainder if not then do the twos\_complement of the remainder. Move the corrected signed values into \$v0, and \$v1 which will hold the signed quotient and remainder.