Designing a Calculator Using MIPS in MARS

Kunwarpreet Singh Behar
San Jose State University
San Jose, CA

Abstract—**This report goes over the steps to implement the basic operations of a calculator; addition, subtraction, multiplication and division. The program will be written MIPS normal and logical operations in MARS.**

I. INTRODUCTION

We will be using MIPS to implement our code in MARS which is also known as MIPS Assembler and Runtime Simulator. Then using MIPS assembly language and the MARS platform we will design a code to run addition, subtraction, multiplication, and division problems; both using normal and logical operations. Throughout the process of this project we will complete the following:

1) Install MARS and learn the basic functionalities of the assmebler

2) Implement the basic caluclator functions using normal operations

3) Implement the basic caluclator functions using logical operations

4) Test the code we designed to confirm our program performs as expected

II. INSTALLING MARS AND THE BASICS

A. Installation

To install MARS, begin by doing a simple Google search for "MARS MIPS Simulator. The fist link should be the one you need.
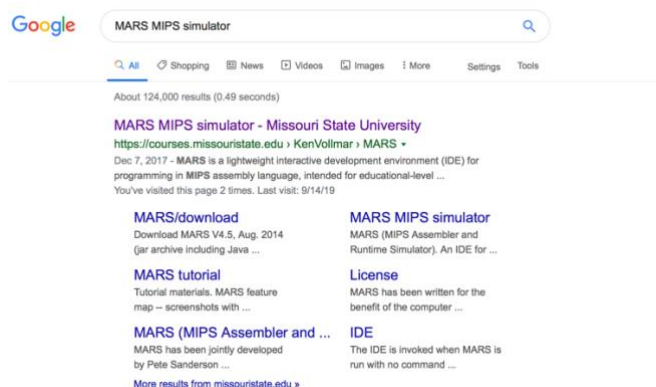
However, if this does not show up for you please visit https://courses.missouristate.edu/KenVollmar/MARS/. On the left-hand side of the website you will find a button that says "download." Click that button and then click the option that says, "Download MARS."

B. The Setup

Once MARS is installed you will be able to launch the jar file to begin programming. However, before you begin programming you should make sure that you have the correct settings.
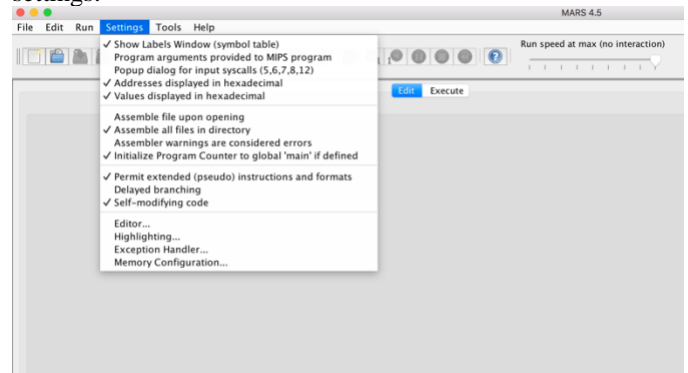
Make sure to have the following settings checked; show labels window (symbol table), addresses displayed in hexadecimal, values displayed in hexadecimal, assemble all files in directory, initialize program counter to global 'main' if defined, permit extended (pseudo) instructions and formats, and self-modifying code.
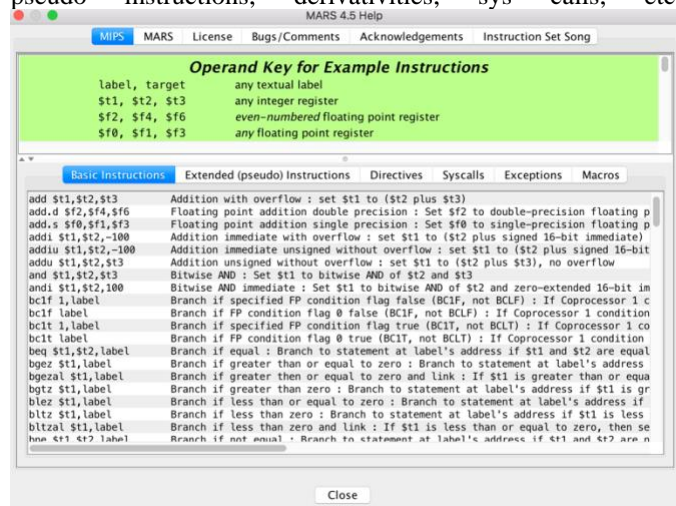
C. The Basics of MARS

In order to program a calculator on the MARS assembler we just downloaded we must go over the basics of how to use and navigate MARS. To begin with, this is how the assembler will look when you start it:
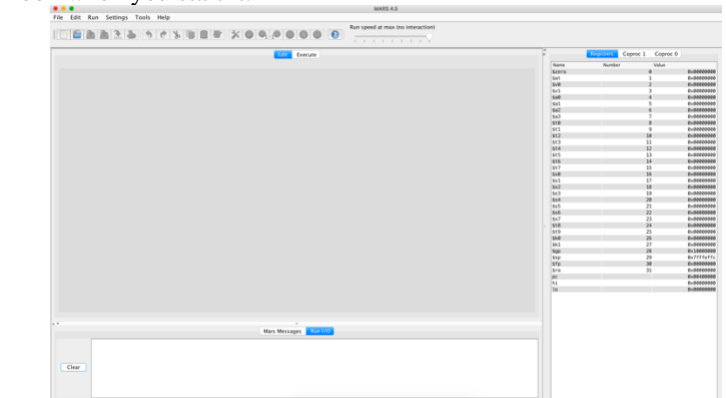
Going across the top you have your basics, easy to navigate buttons: File, Edit, Settings, Tools, and Help. Help is a very useful and important button that we will get back to. Right below this set of buttons you have another row. From left to right the buttons are used to; create a new file for editing, open

a file for editing, save the current file, save the current file with a different name, dump machine code or data in available format, print current file, undo last edit, redo last edit, cut, copy, past, find/replace, run the current program, run one step at a time, undo the last step, pause the currently running program, stop the currently running program, reset MIPS memory and registers, and lastly, help once again. Below these buttons you have an edit view and an execution view which are used as needed (for editing and execution). On the bottom of the assembler, there is a rectangular box which gives your any messages from MARS, run time errors, and the run time output. Lastly, on the right side there is an option which allows you to view the values in your registers and coprocessors. An important feature of MARS that you must know before you begin your project is the "Help" button. Clicking this button will bring up all the basic instructions used in MIPS, as well as pseudo instructions, derivativities, sys calls, etc.

III. Installing/Opening Starter Files and Understanding Project Instructions

Now that we know the basics of MARS, we can begin programming the calculator. To start, visit the "Project 1" assignment on canvas and download the zip file. If you cannot find the assignment use the following link: https://sjsu.instructure.com/courses/1324477/assignments/5056985?module_item_id=10137487. However, this link will only work if the project is open to you. After installing the zip file make sure to extract it. Doing so should give you an extracted file including: cs47_common_macro.asm, CS47_proj_alu_logical.asm, CS47_proj_alu_normal.asm, cs47_proj_macro .asm, cs47_proj_procs.asm, proj-auto-test.asm.

Now that we got the starter files installed, we should go over the instructions of what exactly needs to be done. The only files that need to be (or should be) edited during the process of writing this program are CS47_proj_alu_logical.asm, CS47_proj_alu_normal.asm, and cs47_proj_macro .asm. The other files are important to testing and running your code, however you must make sure that you do not edit or change these files. The

A. Instructions for Normal

You must edit CS47_proj_alu_normal.asm which takes three arguments $a0 first operand, $a1 second operand, and $a2 the operation code. This part of the program should use normal math operations in MIPS (add, sub, mul and div). to make the basic functions of the calculator. The result of operation will be stored into register $v0 for addition and subtraction. For multiplication, the result of HI will be stored into $v1 and LO will be stored into $v0. Lastly, for division, the quotient will be stored into $v0 and the remainder will be stored into $v1.

B. Instrcutions for Logical

For this step you must edit CS47_proj_alu_logical.asm which also take the same three arguments as the normal implementation. The outputs should also be the same as the normal implementation. However, the difference between alu_normal and alu_logical is the implementation itself. In alu_logical you may not use MIPS mathematical operations, but instead use only MIPS logic. Alu_logical may use regular math operations for uses such as incrementing and initializing.

- Complete the following two procedures in **CS47_proj_alu_normal.asm** and **CS47_proj_alu_logical.asm**. You may include your required macros (those you write) in the **cs47_proj_macro.asm**. **Do not update proj-auto-test.asm** or **cs47_proj_procs.asm** or **cs47_common_macro.asm**.
  - au_normal (in **CS47_proj_alu_normal.asm**): It takes three arguments as $a0 (First operand), $a1 (Second operand), $a2 (Operation code '+', '-', '*', '/' - ASCII code). It returns result in $v0 and $v1 (for multiplication $v1 it will contain HI, for division $v1 will contain remainder). This procedure uses normal math operations of MIPS to compute the result (add, sub, mul and div).
  - au_logical (in **CS47_proj_alu_logical.asm**): It takes three arguments as $a0 (First operand), $a1 (Second operand), $a2 (Operation code '+', '-', '*', '/' - ASCII code). It returns result in $v0 and $v1 (for multiplication $v1 it will contain HI, for division $v1 will contain remainder). The evaluation of mathematical operations should use MIPS logic operations only (**result should not be generated directly using MIPS mathematical operations**). The implementation needs to follow the digital algorithm implemented in hardware to implement the mathematical operations.

Before writing the code, you will get a 0/40 on the test cases and after the code is written correctly, you should receive a 40/40.

C. Opening Starter Files

Since we know the instructions and know what we have to do, let's begin with opening the starter files. To open the files, begin with starting up MARS. Once you are there, choose the button which is used to "Open a file for editing".

After this, navigate to the unzipped folder and open each asm file one by one. After opening each file in MARS, you should have something which looks like this above your edit view:
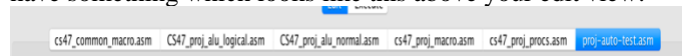
IV. Implementation of The Normal Procedure

In this part we will cover how to implement the alu_normal part of this project.

A. Creating the differnet operation symbols

Begin with creating a frame and then an if statement which will do the following: "if $a2 (operation symbol) = '+' then jump to add". Do this for all operation symbols (+,-,*, and /).

- '+' represents additions
- '-' represents subtractions
- '*' represents multiplication
- '/' represents division

```
au_normal:

    # TBD: Complete it
    #create frame
    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

    #body
    beq $a2, '+' , add #if operation is + jump to add
    beq $a2, '-' , subtract #if operation is - jump to subtract
    beq $a2, '*' , multiply #if operation is * jump to multiply
    beq $a2, '/' , divide #if operation is / jump to divide
```

B. Add

For the addition, simply use the MIPS instruction 'add' and store the addition into $v0.

C. Subtract

For the subtraction, use the MIPS instruction 'sub' and store the subtraction into $v0.

D. Multiply

For multiplication, multiply the operands together using 'mult' which will automatically store the HI value into HI and the LO value into LO. The 'mflo' (move from lo) and 'mfhi' (move from hi) to be the values into $v0 and $v1 as instructed.

E. Divide

For division, divide the operands using 'div' which will automatically move the quotient to LO and the remainder to HI. Then, as you did in multiply, use 'mflo' and mfhi' to move the values into $v0 and $v1.

```
add:
    add $v0, $a0, $a1 #$v0=$a0+$a1
    j end #jump to end where program restores frame

subtract:
    sub $v0, $a0, $a1 #v0=$a0-$a1
    j end #jump to end where program restores frame

multiply:
    mult  $a0, $a1 #multiply $a0 and $a1
    mflo $v0 #v0 = LO
    mfhi $v1 #v1 = HI
    j end #jump to end where program restores frame

divide:
    div $a0, $a1 #divide $a0 by $a1 where LO is quotient and HI is remainder
    mflo $v0 #quotient
    mfhi $v1 #remainder
    j end #jump to end where program restores frame
end:

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

Once you have completed the implementation of alu_normal you should be able to get 1/40 on the test case.

V. Implementation of The Logical Procedure

In this part we will go over the implementation of alu_logical.

A. Macros

To make the process a little simpler we will create some macros: extract_nth_bit($regD,$regS,$regT) and insert_to_nth _bit($regD,$regS,$regT,$maskReg).

Extract_nth_bit($regD,$regS,$regT) should be able to extract the bit at the $regT position (0-31) from $regS (the source bit pattern) and this should result into $regD being 0x1 or 0x0. To implement this macro use 'srlv' to shift right on $regS by $regT and store that value into $regD. Then you AND that value with 0x1 to get the value of the bit. This macro should look something like this:

```
.macro extract_nth_bit($regD, $regS, $regT) #as described in video
    srlv  $regD, $regS, $regT
    and $regD, $regD, 1
    .end_macro
```

Insert_to_nth_bit($regD,$regS,$regT,$maskReg) should insert the value from $regT (register containing the bit to insert, 0 or 1) into $regD at $regS (the nth position). $regMask is used to hold a temporary mask. Begin by initializing the $regMask to 1. Then, left shifting mask by n bits. After you have done that, invert $maskReg. Next, do an AND operation with your new $maskReg and $regD. After this, you shift $regT n times and then perform an OR operation with $regD.

```
                           1
                           v
            1 1 1 1 0 1 1 0 <--- D; n=3; b=1

    ---------------------------------------------
            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 1 1 0 1 1 0 <--- D
    ---------------------------
            1 1 1 1 0 1 1 0
        |   0 0 0 0 1 0 0 0 <--- b = b << n
    ---------------------------
            1 1 1 1 1 1 1 0
```

After following these steps, you should get a macro looking like this:

```
.macro  insert_to_nth_bit ($regD, $regS, $regT, $maskReg) #as described in video
    li $maskReg, 1
    sllv $maskReg, $maskReg, $regS
    not $maskReg, $maskReg
    and $regD, $regD, $maskReg
    sllv $regT, $regT, $regS
    or $regD, $regD, $regT
    .end_macro
```

B. Creating the differnet operation symbols

Similar to the way we created if then statements using beq in the alu_normal, we will do the same here.

```
au_logical:
# TBD: Complete it
    #body
    beq     $a2, '+' , addition #if operation is + jump to add_sub
    beq     $a2, '-' , subtract #if operation is - jump to add_sub
    beq     $a2, '*' , multiply #if operation is * jump to multiply
    beq     $a2, '/' , divide #if operation is / jump to divide
    ....
```

C. Addition and Subtraction

The process for addition and subtraction is very similar, so we will attempt to combine some parts of the process for these two operations.

$a0 is the first operand, $a1 is the second operand and $a2 is a "mode" where it is set to 0x00000000 if the operation is addition and 0xFFFFFFFF if the operation is subtraction. To begin with, the if statement should branch to a label called addition or subtraction depending on the operation which should alter the value in $a2 register as needed. These two should come back together by branching to a label, 'add_sub' which initializes the counter, summation, and C (the first bit of $a2). Check to see what the value of C is. If C is 0 then the procedure addition, so jump to 'add_sub_process'. On the other hand, if C is 1 then the procedure is subtraction, so invert the second operand and then jump to 'add_sub_process'.

Inside of the 'add_sub_process' you perform one-bit full addition. One-bit full addition is where you pick up one bit at a time from each operand and from C and then XOR between them. C is the carry bit of the XOR between the previous one-bit addition. Since you are doing one bit at a time for the entire number you should have a loop going from 0 to 31. Use the macros created to extract one bit at a time, add them and then insert each bit where it belongs inside of $v0, which should include your final answer. Repeat this process by incrementing I until you finish with the MSB (Incrementor = 31). Once I = 31 branch to 'add_sub_end' where you restore the frame. The algorithm of this process is visualized as the following:
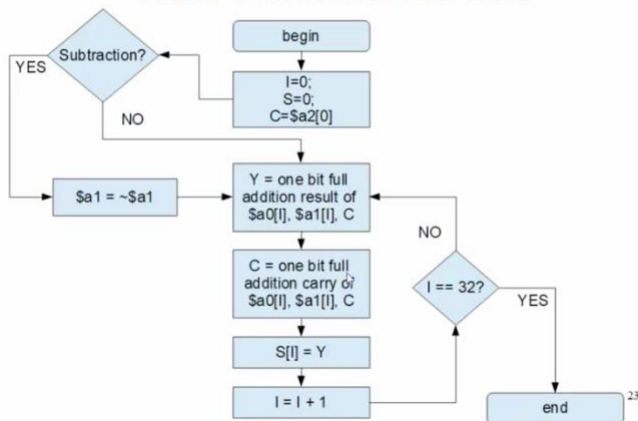
Once completed, your implementation should look similar to the one shown below. However, it may or may not differ depending on if you followed this algorithm or not. Also, it may differ depending on the instructions and pseudo instructions you used and in what order you performed your operations.

```
24   addition:
25          li      $a2,0x00000000
26
27          j       add_sub
28
29   subtract:
30          li      $a2,0xFFFFFFFF
31
32          j       add_sub
33
34   add_sub:
35          #create frame
36          addi    $sp, $sp, -32
37          sw      $fp, 32($sp)
38          sw      $ra, 28($sp)
39          sw      $t0, 24($sp)
40          sw      $t1, 20($sp)
41          sw      $t2, 16($sp)
42          sw      $t3, 12($sp)
43          sw      $t4, 8($sp)
44          addi    $fp, $sp, 32
45
46          li      $t0, 0x0 #index=0
47          li      $v0, 0x0 #summation = 0
48
49          extract_nth_bit($t1,$a2,$t0) #c=$a2[0]
50
51          beq     $a2, 0x0, add_sub_process
52
53          j       invert
54
```

```
55   invert:
56          nor     $a1, $a1, $zero
57
58          j       add_sub_process
59
60   add_sub_process:
61          extract_nth_bit($t2,$a0,$t0)
62          extract_nth_bit($t3,$a1,$t0)
63
64          xor     $t4, $t2, $t3 #op 1 xor op 2
65          and     $t5, $t2,$t3
66          xor     $t6, $t4, $t1 #c xor (op1 xor op 2)
67          and     $t1,$t1,$t4
68
69          or      $t1, $t1, $t5
70
71          insert_to_nth_bit($v0,$t0,$t6,$t3)
72
73          addi    $t0, $t0, 0x1 #increment index
74          beq     $t0, 32, add_sub_end #if index = 32 then break loop
75
76          j       add_sub_process #loop
77

78   add_sub_end:
79          #restore frame
80          lw      $fp, 32($sp)
81          lw      $ra, 28($sp)
82          lw      $t0, 24($sp)
83          lw      $t1, 20($sp)
84          lw      $t2, 16($sp)
85          lw      $t3, 12($sp)
86          lw      $t4, 8($sp)
87          addi    $sp, $sp, 32
88
89          jr      $ra
90
```

The multi bit addition and subtraction diagram/flow chart looks as shown below:
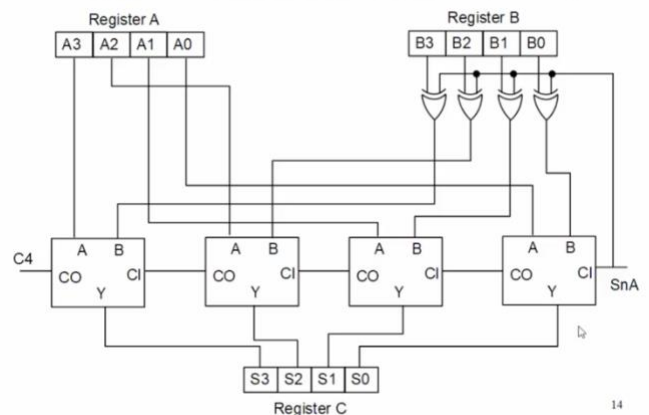
Once you have completed the implementation of the alu_normal and the alu_logical addition and subtraction part you should be able to get a 20/40 on the test cases. You should be halfway done with the project (P.S this was the easier half, the next part gets crazy).

D. Multiplication

The algorithm of the multiplication processes goes as follows. You start and assign product (a 64-bit output) as 0. Then, you look at the LSB of the multiplier and test if it is 0 or not. If the multiplier is 1, add the product with the multiplicand, so product = product + multiplicand and then left shift multiplicand by one bit. If the multiplier is 0 then skip straight

to shifting the multiplicand by one bit. Next, you shift the multiplier to the right by one bit. You do this on a loop, with an incrementor. Once, you hit 31 you stop. The visual representation of the algorithm looks like this:

## Binary Multiplication Algorithm
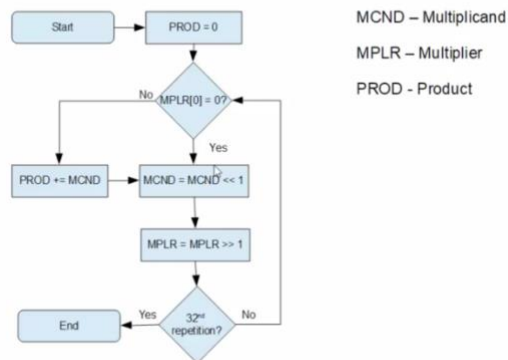
A 32-bit unsigned multiplier has a diagram which make it a lot easier to visualize what the multiplier is actually doing.

Simplified Sequential Multiplier
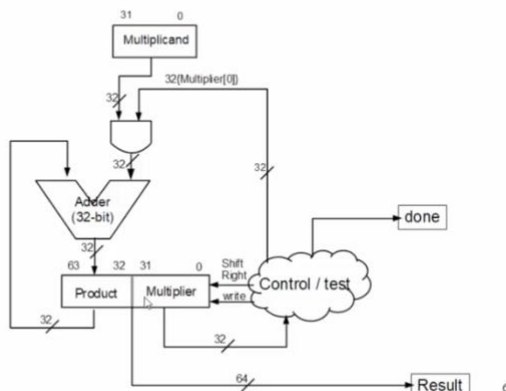
However, this is only for a 32-bit unsigned multiplier. We also want to construct signed multiplication. Unsigned multiplication will ignore the signs of the operands and only give you a positive answer, thus having a signed multiplier in our program is essential. A signed multiplier visual looks like the following:

Signed Multiplication Circuit
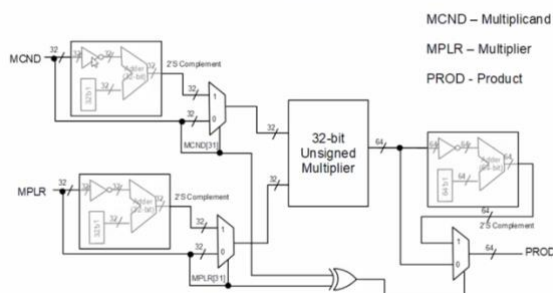
To implement this in MARS we must create a few procedures:

1) Twos_complement- This procedure should return the twos complement of the argument ($a0). In this procedure you use 'NOT' and your previosly designed addition procedure to compute ~$a0+1.

```
205   twos_complement:
206           not     $a0, $a0
207           li      $a1, 1
208           j       addition #~$a0 + 1
209
```

2) Twos_complement_if_negative- Depending on if the bit pattern is negative use this procedure. It should return the twos complement of $a0 if $a0 is negative. If the incoming bit pattern is negative then do twos_complement otherwise do not. This procedure should simply call twos_complement if necessary.

```
210   twos_complement_if_negative:
211           bltz    $a0, twos_complement
212           jr      $ra
213
```

3) Twos_comp_64bit- the argument $a0 is the lower half of the adress and the argument $a1 is the hi of the number. It should return $v0, the lo part of the 2s complement 64 bit and $v1, the hi part of the the 2s complement 64 bit. First invert both $a0 and $a1. Then, use your previosly created addition procedure to add 1 to $a0. Lastly, add carry from previous step to $a1 (also using the addition procedure you created). Make sure to not just take the complement of the upper and lower half because this will not work.

```
214   twos_comp_64bit:
215           #create frame
216           addi    $sp, $sp, -16
217           sw      $fp, 16($sp)
218           sw      $ra, 12($sp)
219           sw      $t0,  8($sp)
220           addi    $fp, $sp, 16
221
222           nor     $t0, $a1, $zero
223           nor     $a0, $a0, $zero
224           add     $a1, $zero, 1
225
226           jal     addition
227
228           move    $a0, $t0
229           move    $t0, $v0
230           move    $a1, $v1
231
232           jal     addition
233
234           move    $v1, $v0
235           move    $v0, $t0
236
237           #restore frame
238           lw      $fp, 16($sp)
239           lw      $ra, 12($sp)
240           lw      $t0,  8($sp)
241           addi    $sp, $sp, 16
242
243           jr      $ra
244
```

4) Bit_replicator- replicates a given bit value to 32 times. This takes an argument of $a0 which is 0x0 or 0x1 and sets $v0 to 0x00000000 if $a0 is 0x0 or 0xFFFFFFFF if $a0 is 0x1.
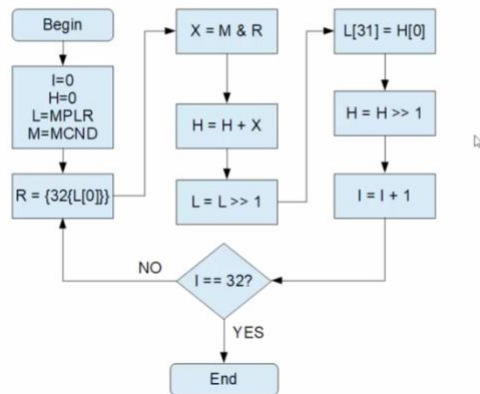
```
245   bit_replicator:
246          beq     $a0, 0x0, replicate_0
247          li      $v0, 0xFFFFFFFF
248          jr      $ra
249
250   replicate_0:
251          li      $v0, 0
252          jr      $ra
253
```

5) Multiply_unsigned- this should take $a0 (the multiplicand) and $a1 (the multiplier) as arguments. Its should return lo through $v0 and high through $v1. The algorithm looks like this:

Unsigned Multiplication

The code for unsigned multiply should look like the following:

```
139   multiply_unsigned:
140          #create frame
141          addi    $sp, $sp, -48
142          sw      $fp, 48($sp)
143          sw      $ra, 44($sp)
144          sw      $s0, 40($sp)
145          sw      $s1, 36($sp)
146          sw      $s2, 32($sp)
147          sw      $s3, 28($sp)
148          sw      $s4, 24($sp)
149          sw      $s5, 20($sp)
150          sw      $t0, 16($sp)
151          sw      $t1, 12($sp)
152          sw      $t2, 8($sp)
153          addi    $fp, $sp, 48
154
155          add     $s0, $zero, $zero #I
156          add     $s1, $zero, $zero #H
157          move    $s2, $a0 #$a0 is multiplicand = M
158          move    $s3, $a1 #$a1 is multiplier = L
159
160          j       multiply_process
161
```

6) Signed Multiplication: Set n1 to $a0 and n2 to $a1. Make n1 and n2 twos complemenet if they are negative.Then, call unsigned multiplication using n1 and n2. Then determine the sign of the bit by extracting the MSB of $a0 and $a and xor between them. The xor result is S. If S is 1 then use

twos_comp_64bit to determine the twos complement of the number.

Following the algorithms and diagrams above your code for multiplication should appear as this:

```
91   multiply:
92          #create frame
93          addi    $sp, $sp, -28
94          sw      $fp, 28($sp)
95          sw      $ra, 24($sp)
96          sw      $s2, 20($sp)
97          sw      $s1, 16($sp)
98          sw      $s0, 12($sp)
99          sw      $t0, 8($sp)
100         addi    $fp, $sp, 28
101
102         move    $s0, $a0 #n1
103         move    $s1, $a1 #n2
104         move    $v0, $a0
105
106         jal     twos_complement_if_negative     #if negative returns in $v0
107
108         move    $s2, $v0 #n1 is stored in s2
109         move    $a0, $s1
110         move    $v0, $s1
111
112         jal     twos_complement_if_negative
113
114         move    $a1, $v0 #n2 = a1
115         move    $a0, $s2
116
117         jal     multiply_unsigned #calls unsigned multiplication with $a1 and $a0
118
119         move    $a0, $v0 #quotent
120         move    $a1, $v1 #remainder
121         add     $t0, $zero, 31
122
123         extract_nth_bit($s0, $s0, $t0) #sign of a0 in s0
124         extract_nth_bit($s1, $s1, $t0) #sign of a1 in s1
125
126         xor     $s0, $s0, $s1 #s in s0
127         beq     $s0, 1, twos_comp_64bit #get complement if s is 1
128
129         #restore frame
130         lw      $fp, 28($sp)
131         lw      $ra, 24($sp)
132         lw      $s2, 20($sp)
133         lw      $s1, 16($sp)
134         lw      $s0, 12($sp)
135         lw      $t0, 8($sp)
136         addi    $sp, $sp, 28
137         jr      $ra
138
162   multiply_process:
163         extract_nth_bit($s4, $s3, $zero)
164
165         move    $a0, $s4
166
167         jal     bit_replicator #replicate the value in R 32 times, returns in $v0
168
169         move    $s4, $v0
170         and     $s5, $s2, $s4 #X = M & R
171         add     $s1, $s1, $s5 #H = H + X
172         srl     $s3, $s3, 1 #L = L >> 1
173
174         extract_nth_bit($t0, $s1, $zero) #extract 0th bit from H
175
176         add     $t1, $zero, 31
177
178         insert_to_nth_bit($s3,$t1,$t0,$t2) #L[31] = H[0]
179
180         srl     $s1, $s1, 1 #H = H >> 1
181         add     $s0, $s0, 1 #increment counter
182         beq     $s0, 32, multiply_end #if index = 32 then break loop
183
184         j       multiply_process #loop
185
186   multiply_end:
187         move    $v0, $s3 #lo
188         move    $v1, $s1 #hi
189
190         #restore frame
191         lw      $fp, 48($sp)
192         lw      $ra, 44($sp)
193         lw      $s0, 40($sp)
194         lw      $s1, 36($sp)
195         lw      $s2, 32($sp)
196         lw      $s3, 28($sp)
197         lw      $s4, 24($sp)
198         lw      $s5, 20($sp)
199         lw      $t0, 16($sp)
200         lw      $t1, 12($sp)
201         lw      $t2, 8($sp)
202         addi    $sp, $sp, 48
203         jr      $ra
204
```
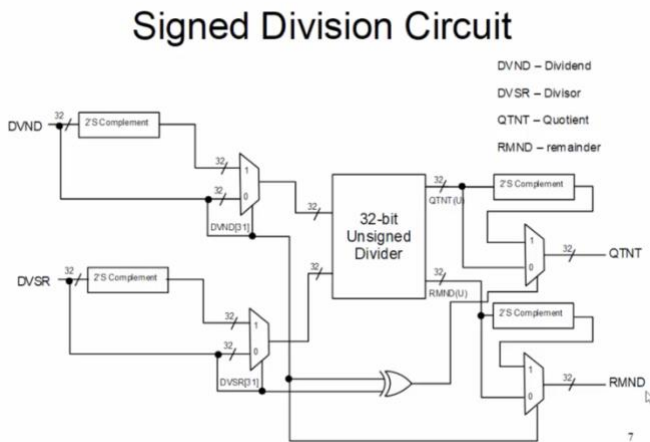
This code above should be accompanied with the procedures we went over.

E. Division

The algorithm should begin by loading the remainder register (64 bits) with the dividend in the lower half and 32 0's in the upper half and loading the divisor register (64 bits) with the divisor in the upper half and 32 0's in the lower half. Next, you do remainder = remainder – divisor (64) using a 64-bit subtractor. Next, if the remainder is not less than 0 then left shift quotient by one and then OR with 1. On the other hand, if the remainder is less than one after doing remainder = remainder – divisor, then you do remainder = remainder + divisor and then left shift quotient by 1. Then, we right shift divisor by 1. You do this until 33rd repetition. The visual representation of the algorithm looks as following:


Binary Division Algorithm

DVND – Dividend
DVSR – Divisor
QTNT – Quotient
RMND – Remainder
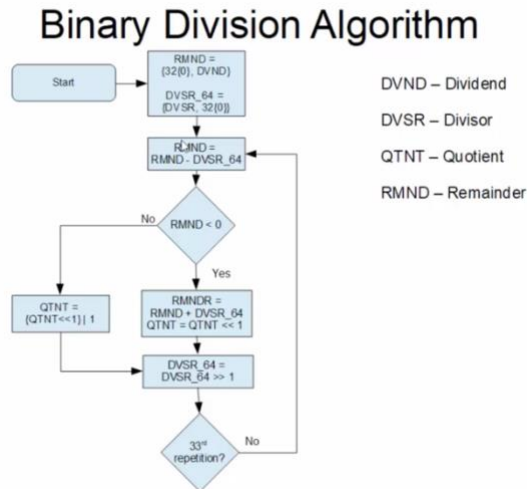
A visual representation of a signed division circuit looks like this:

Signed Division Circuit

DVND – Dividend
DVSR – Divisor
QTNT – Quotient
RMND – remainder

Your code for alu_divide should include the following procedures:

1) Divide_unassigned- this takes the aruguments $a0 (dividend) and $a1 (divisor) and returns $vo (quotient) and $v1 (remainder). The approach to writing the algorithm for divide_unassigned is as follows:

Unsigned Division
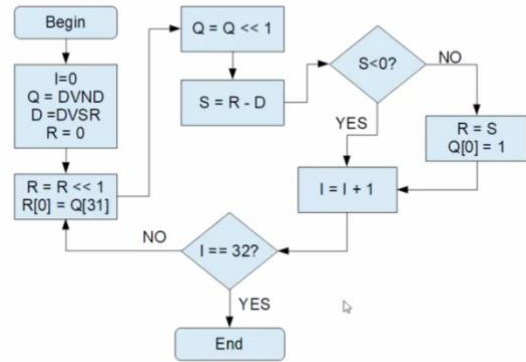
The steps of signed division are as follows:

Signed Division

- **Steps**
  - N1 = $a0, N2 = $a1
  - Make N1 two's complement if negative
  - Make N2 two's complement if negative
  - Call unsigned Division using N1, N2. Say the result is Q and R
  - Determine sign S of Q
    - Extract $a0[31] and $a1[31] bits and xor between them. The xor result is S.
    - If S is 1, use the 'twos_complement' to determine two's complement form of Q.
  - Determine sign S of R
    - Extract $a0[31] and assign this to S
    - If S is 1, use the 'twos_complement' to determine two's complement form of R.

Once completed, your code for divide should look like this:

```
206  divide:
207         #create frame
208         addi    $sp, $sp, -28
209         sw      $fp, 28($sp)
210         sw      $ra, 24($sp)
211         sw      $s2, 20($sp)
212         sw      $s1, 16($sp)
213         sw      $s0, 12($sp)
214         sw      $t0, 8($sp)
215         addi    $fp, $sp, 28
216
217         la      $s0, ($a0) #N1
218         la      $s1, ($a1) #N2
219         move    $v0, $a0
220
221         jal     twos_complement_if_negative #if negative then take complement
222
223         move    $s2, $v0 #N1 = s2
224         move    $a0, $s1
225         move    $v0, $s1
226
227         jal     twos_complement_if_negative #if negative then take complement
228
229         move    $a1, $v0 #N2 = a1 divisor
230         move    $a0, $s2 #N1 = a0 dividend
231
232         jal     divide_unsigned #returns $v0 quotient and $v1 remainder
233
234         move    $a0, $v0 #Q
235         move    $a1, $v1 #R
236
237         #determine sign
238         add     $t0, $zero, 31
239
240         extract_nth_bit($s0, $s0, $t0) #sign of a0 in t2
241         extract_nth_bit($s1, $s1, $t0) #sign of a1 in s1
242
243         xor     $t6, $s0, $s1 #S in t2
244         and     $t7, $t6, $s0
245         beq     $t7, 1, complement2 #if both are 1 then negate both
246         beq     $t6, 1 , twos_complement
247         beq     $s0, 1, complement
248
249         #restore frame
250         lw      $fp, 28($sp)
251         lw      $ra, 24($sp)
252         lw      $s2, 20($sp)
253         lw      $s1, 16($sp)
```

```
253        lw      $s1, 16($sp)
254        lw      $s0, 12($sp)
255        lw      $t0, 8($sp)
256        addi    $sp, $sp, 28
257        jr $ra
258
259 divide_unsigned:
260        addi    $sp, $sp, -48
261        sw      $fp,  48($sp)
262        sw      $ra,  44($sp)
263        sw      $s0,  40($sp)
264        sw      $s1,  36($sp)
265        sw      $s2,  32($sp)
266        sw      $s3,  28($sp)
267        sw      $s4,  24($sp)
268        sw      $s5,  20($sp)
269        sw      $t0,  16($sp)
270        sw      $t1,  12($sp)
271        sw      $t2,  8($sp)
272        addi    $fp, $sp, 48
273
274        add     $s0, $zero, $zero# I  (counter)
275        add     $s1, $zero, $zero# R Remaninder
276        move    $s2, $a0# Q
277        move    $s3, $a1# D
278
279        j           divide_process
280
281 divide_process:
282        sll     $s1, $s1, 1 #R = R << 1
283        li      $t0, 31
284
285        extract_nth_bit($s4, $s2, $t0) #extract 31st bit from Q
286        insert_to_nth_bit ($s1, $zero, $s4, $t9) #insert at R[0]
287
288        sll     $s2, $s2, 1 #Q = Q << 1
289        move    $a0, $s1 #move R into $a0
290        move    $a1, $s3 #move D into $a1
291
292        jal     subtract
293
294        move    $s5, $v0 #solution stored in $s5 after subtracting S
295        la      $a0, ($s2)
296        bltz    $s5, check_loop
297        move    $s1, $s5
298        li      $t1, 1
299
300        insert_to_nth_bit ($s2, $zero, $t1, $t9)
301
302        j           check_loop
303
304 check_loop:
305        addi    $s0, $s0, 1
306        li      $t2, 32
307        beq     $s0, $t2, divide_process_end #if index = 32 then break loop
308        j       divide_process #loop
309
310 divide_process_end:
311        move    $v1, $s1 #remainder
312        move    $v0, $s2 #quotent
313
314        #restore frame
315        lw      $fp,  48($sp)
316        lw      $ra,  44($sp)
317        lw      $s0,  40($sp)
318        lw      $s1,  36($sp)
319        lw      $s2,  32($sp)
320        lw      $s3,  28($sp)
321        lw      $s4,  24($sp)
322        lw      $s5,  20($sp)
323        lw      $t0,  16($sp)
324        lw      $t1,  12($sp)
325        lw      $t2,  8($sp)
326        addi    $sp, $sp, 48
327        jr      $ra
328
```

F. Testing

Now you should be finished with implementing alu_normal and alu_logical so we could test if our code works as expected. To test the code, make sure all files are saved as the latest version and then run the code. To run the code, go to the proj-

auto-test.asm file (on MARS) and hit the 'assemble the current file and clear breakpoints' button. Then, hit the 'run the current program' button. Assuming you did everything correct, your output should be:

```
(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 --
```