Lab 6: Integer Mathematics

Joseph Salazar
College of Engineering
University of Miami
Miami, United States
jes409@miami.edu

***Abstract*— The ARM assembly language is widely used by professionals in the computer engineering field. However, there is very little material that introduces entry-level persons into assembly language. To help individuals better understand assembly for their futures in engineering, we propose following a quick and efficient program provided by the University of Miami's College of Engineering. This program allows individuals to practice assembly by coding a multiplier using only add and shift operations.**

Keywords—ARM, assembly,

I. Introduction

The purpose of this lab is to advance ones' knowledge with the DE1's I/O devices, as well as improving their understanding of integer math. By becoming familiar with the I/O devices, one can utilize their board more effectively and in many more ways. In addition to this, understanding integer math will optimize run time of ones' code if done properly. To demonstrate the importance of this, a programmer will be asked to write a code in ARM assembly that multiplies two numbers without using "mul" instructions. The operation is dependent on "shift" and "add" operations as it is very efficient. Once a result is obtained, the programmer should display it on the board's LED's. By the end, the programmer should have learned a different approach to what seems to be a simple task.

II. Method/Experient

A. Overview

Before attempting this lab, one should have Altera Monitor Program installed before continuing. Then, power on the De1 board and connect it to the computer. Once connected, one should proceed to set-up their coding environment/document so that the code remains organized. After one's ".s" file is created, the programmer should include the data, global, text, start, and end directives to ensure compilation goes smoothly after code has been added. If one is not familiar with this routine, referencing the DE1 manual would be most beneficial [2].

Before attempting the code, the programmer should observe where the LED's and switches are located in memory. In order to utilize them, one will have to initialize them and create a pointer to effectively read and write values. Once this is understood, the programmer can begin writing their program.

To begin this lab, one is asked to write a short program that multiplies two integers. The programmer should start by allocating two registers to their multiplicands. It will be easy to assign different values very quickly if a move instruction is utilized. Next, a programmer should be familiar with how multiplication works by only using shift and add. The programmer should first check that neither integer is zero, and then start checking the bits of one of the numbers. If a LSB is a 1, then the value of that number is added to a register dedicated to the result, and that number is shifted left once while the other integer is shifted right once. If a LSB is a 0, then the integers are shifted, but nothing is added to the result. This should be repeated until the integer shifted right reaches 0. Visually, this can be seen in Figure 3. The programmer is also prohibited from hard-coding a number of iterations that this happens. The loop should end on its own, and store the correct value in a different register.

Lastly, the programmer is asked to add another condition where all the LED's turn on if the result is zero. This is like the previous step, and it should reiterate the importance of knowing how to code effectively and efficiently in assembly.

B. Results

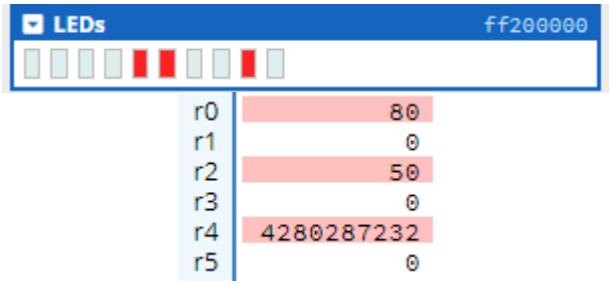

Figure 1. The following figure shows five multiplied by ten, and the result is stored in register 2.

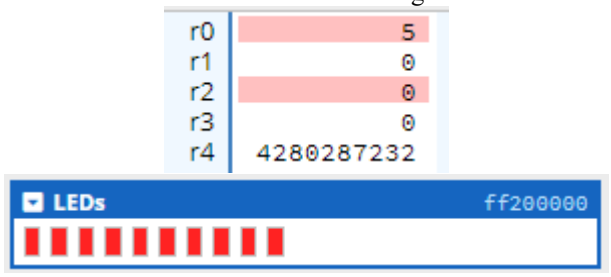

Figure 2. The following figure shows the result in r2 where one of the integers was a zero.

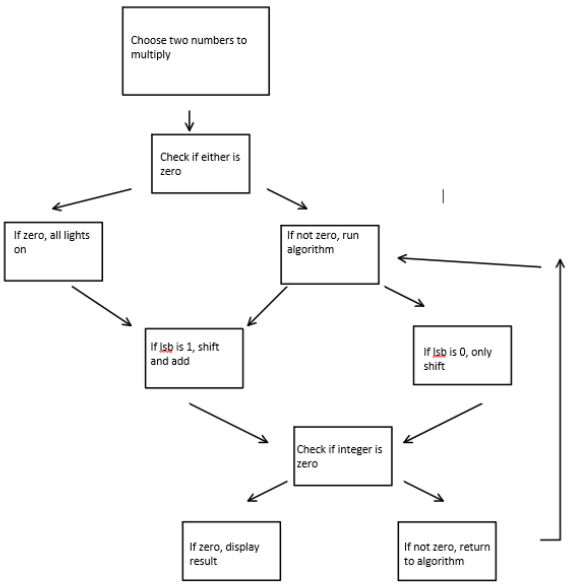

Figure 3. The following is a flow-chart for the steps taken during this exercise.

C. *Discussion*

After completing the previous tasks, one could reflect and take many lessons away. These exercises emphasize the importance of knowing how to correctly utilize a DE1's switches and LED's to create a visual coding experience. This could be very important to know in case one is tasked with an assignment involving the switches and/or LED's. As a computer engineer, one should be able to be well-rounded in all aspects of ARM assembly.

Ultimately, this exercise should not provide too much difficulty because it only includes writing a relatively simple program. The most difficult task is determining which loops should be executed based on the multipliers LSB. One should take their time to ensure that they fully understand the shift and add algorithm of multiplication. With this being said, the exercise is very well-written, but a programmer should try to build upon it on his own in order to completely understand the material. The exercise should not change, but extra steps, such as creating a divider, should be explored.

III. Conclusion

In sum, this exercise introduces instructions in ARM assembly that are essential for more difficult programs. After following the instructions, the programmer will have walked away with a deeper understanding of a DE1's I/O and how to write algorithms that are more efficient than some of ARM's built in operations. The results gathered from this exercise reinforce this idea by forcing the programmer to thoroughly think through an algorithm that is new to them. Ultimately, this will help the programmer excel as a computer engineer by learning new and important content in ARM assembly code.

University of Miami

References

[1] ARM Limited. ARM Architecture Reference Manual. 1996.

[2] *Documentation–ArmDeveloper*, developer.arm.com/documentation/ddi0388/latest/.

[3] "GNU Manuals Online - GNU Project - Free Software Foundation." *[A GNU Head]* , www.gnu.org/manual/manual.en.html.

[4] Pyeatt, Larry D. *Modern Assembly Language Programming with the ARM Processor*. Newnes, 2016.

APPENDIX

```
.data
led:        .word     0xff200000
lights:     .word     1023
.global _start
.text
_start:
ldr r4,=led
ldr r4, [r4]
ldr r8,=lights
ldr r8,[r8]
mov r0, #5 //x
mov r1, #0  //y
mov r2, #0  //a
//operands

cmp r0, #0
beq done
cmp r1, #0
beq lightup
//if one of them is zero, end

checklsb:
mov r10, #0
AND r10, r1, #1
cmp r10, #1
beq accumulate
bne lsbzero

accumulate:
add r2, r2, r0 //accumulate a
b lsbzero

lsbzero:
lsl r0,#1      //shift x left
lsr r1,#1    //shift y right
cmp r1,#0
bne checklsb //continue checking bits
beq display          // end

display:
cmp r2, #0
beq lightup
str r2,[r4]
b done

lightup:
str r8,[r4]
b done

done:
b           done

.end
```