ECE 255, 2018 Chapter: TA Lab #2 Progress Report

By: Philip B. Alipour

TAs: Philip B. Alipour and Amir Khatibzadeh

General Lab Notes (log excluded or irrelevant):

1. **What is expected to happen in lab experiment #2?**

i. Following our last session, due to the thanksgiving holiday, the remainder of some groups for 1st lab evaluation will commence during the 2nd lab.

ii. Next session will be conducted with all students to be marked prior to their report submission two weeks after the session ends.

iii. The focus of this lab is to write your code in assembly and generate **prime numbers**. For a successful code, 45% is granted and as you explain the code, another 5% is added. For an attempted code without major errors, 30% assuming most of the objectives are accomplished. **I will also give a hint code (see next page) which you need to dig deeper on the internet in order to complement it. Note:** this is an **assembly line-by-line comment style algorithm** for you to interpret and translate as a successful code. If your code is based on this, as far as it fulfills all of the objectives, the 45% is granted.

iv. Another 10% would be granted if you successfully show an **optimized version of your code** with shorter time execution (2% to display the time even if it is a slow code compared to others; 7% for the best time possible and how you have implemented it).

v. I and Amir shall continue to assist students on their machines as technical problems occur on the debugger and software as well as giving hints on the lab and performance evaluation per group. Lab 2 performance is one week after the next session.

vi. We will include bonus points, a maximum of 5% in order to compensate the overall session grade for those who have lost points during and after session (such as in the lab report due next week.)

Notes for the Attending Students on the forthcoming lab sessions:

2.1. Primary challenges and info on Lab 2

- As usual you need to take the same steps for configuring your debugger similar to Labs 0 and 1 in **eclipse** to compile and run the code series successfully. But now you need to write up your own assembly code to generate your **prime numbers** in an array successfully and efficiently (in terms of machine performance).
- Now you can appreciate how extensive the process is on assembly level checking out the **register content** being updated by operators systematically, as they are compared (**cmp**), branched into (e.g. **bge**) like a **switch** (**cases** in C language), popped from or into, etc. compared to the high-level representation of a simple code (finding prime numbers using modulus, for example).
- The following is the comment-based and actual assembly code (most lines need your input to complete the missing parts i.e. the actual code and

not the comments) for Lab 2 problem, where you need to translate it properly in your code!

- **Note:** registers in the following code could be any other number depending on your code assumption… **if you have your own solution, you may compare and have it both to present, you shall gain bonus marks as well for the effort (of course, commented code):**

```
// This file is part of the GNU ARM Eclipse distribution.
// Copyright (c) 2014 Liviu Ionescu.
//
// Lab 2 prime numbers Hint Code:
// Parts have been appended as a hint code and will not compile properly if taken as it
// is (for the purposes of the ECE 255 course at UVic, Canada in Oct. 2018
//
// Author: Philip B. Alipour
// ----------------------------------------------------------------------
                .data
primeArray:     .space                  @type in a value for your space appropriately
                                        @(relevant to STM 32 architecture and the data you
                                        @ need to store and parse).

        .text
        .global main

main:                           //change main to the file you create to avoid main filename
                                                    // conflict in eclipse e.g. ComputePrimes
          //and make sure to link this file to the main.c via extern function in that file
                push {r3,r4,r5,r6,r7}
                mov     r4, lr
                push {r4}
                                                        //initialize i, n, load array address
                                                                    @ use mov and ldr
//end
/*********************************************************************************************/
//1st 'for' loop
for1:
              // use cmp, branch command and a specific logical shift register to discover
                // primes; further hint: study section 2 of http://www.toves.org/books/arm
                // which you might find it useful to implement your algorithm or this one.
                                                                                @ given n<30
                                          // now specify range (Limit) for your numbers to
                                                    // avoid establishing an infinite loop
//2nd 'for' loop
for2:
                                      @ if j < Limit
                bge     endfor2       @ j>= Limit
                mov                 @ temporarily put i to a register to calculate remainder
/*********************************************************************************************/
//3rd 'for' loop to calculate the remainder i mod j
for3:                           @ need to compute the remainder = i mod j (r2 mod r1)
                                @ if some register >0
                                @ if the same register >=0
                sub             @ then same register <- i-j
                b               @ then branch to (goto) a for loop
endfor3:
/*********************************************************************************************/
//verify i is prime or not
if1:
                cmp             @ if remainder is equal to some value
                bne             @ remainder not equal to the same value
```

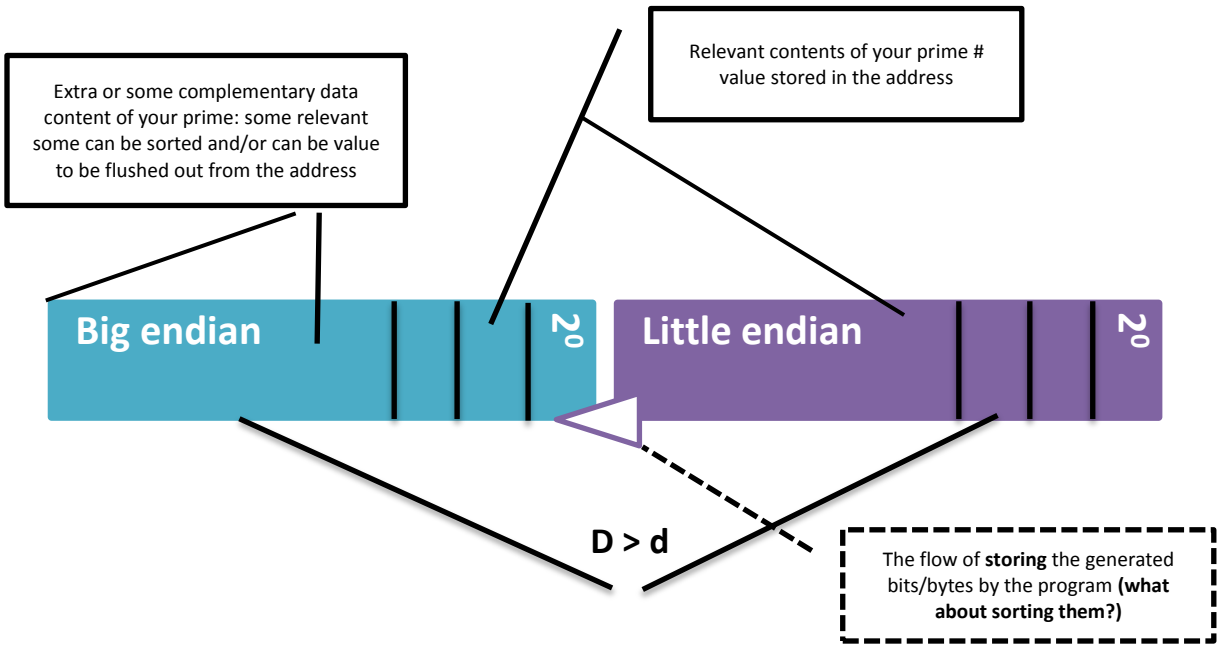
```
                mov             @ move
                b              @ then branch to (goto) a for loop
fi1:
                add             @ a register <- j++
                b                       @ goto for2
endfor2:
if2:
                  // use similar command to the previous including str to store your primes
                        // once you catch one adjust primeArray address to store next prime
                                                                              // number (use add).
fi2:
                                @ i+=2
                b               @ goto for1
endfor1:
stop:
                pop {r4}
                mov    lr, r4
                pop {r3,r4,r5,r6,r7}
                bx lr

                        .align
primeArrary_addr:       .word   primeArray

/*****************************************************************************/
//once you complete your code, make sure to link it to the main.c file according to the
//last page of your Lab #2 manual and run your code successfully. It should generate your
// primes and can be checked using the registry list (memory map) containing the numbers
// step-by-step run. Also for full marks, the timer must be displayed.
```

2.2. Hardware/software challenges

- Compared to the TA's sorting algorithm, for the full 10% portion of the 60% you need to have either a sorting algorithm or a way in partitioning your space (the first two lines after the current header above (or see lab 1, part 3 code)) and manage/sort data relative to what is actually being stored/read in your address (to deal with the big O algorithm complexity issue of your looping algorithm… see below).
- To make your code the most efficient (which is beyond the scope of this lab… but if you do it, you gain bonus marks on top of the regular total as well!) you need to know the difference between the times/steps e.g.,

T(n)= O(n^2) and **O(n).**

The aim is to achieve the latter and in a perfect world, **O(n-{1, 2, 3, …})** when the code executes. For example, as depicted below, if an operation is performed from right-to-left (in your eclipse, the memory map is displayed and partitioned from left-to-right), the **little endian** distance is shorter than the **big endian**. If the relevant contents representing your prime numbers is stored in the targeted address, yet with extra bits (the way **space** is partitioned in your code as you define it), then it inevitably stores the content line-by-line for the ascending prime numbers with much greater distances (accessing the big endian addresses) **D**. This will impede or slow down the level of code execution i.e. greater algorithmic complexity or higher O's.



The aim is to gain minimum distances of d's rather than **D > d** which pertains to bigger O's (something to avoid or address upon).

And the relation between distance **d** or **D** to processing speed or performance is frequency **f** of the input/output of data to/from registers, such that

---

$d*f = p_s$ or processing speed, where
f=1/T(n) and is measured in Hertz (or cycles per second).

So the more steps in frequency f to process data items with distance D, the worst your computational performance.

- Another issue that could be addressed is changing the flow and order of your **For loops** as well as **nested loops** based on a condition which could be instead of satisfying long iterations (longer distances) satisfy shorter ones if possible.
- The proper solution to achieve an optimized code would be typing up a, e.g. a sorting algorithm like Bubble Sort (http://en.wikipedia.org/wiki/Bubble_sort) to address $O(n^2)$ and satisfy O(n) levels of execution.
- The timing is generated for a successful loop series by the main.c file, and not the prime numbers source code where you are supposed to work on as your prime numbers solution.

Notes on the C to assembly translator:

- In C, both **%** and **mod** operations are possible, whereas the latter depending on the compiler might be not accepted so this function must be typed up or brought into code by a library function (**#include**…).
  - Another problem is the division in ARM assembly. Here is a useful article to get this done for your mod function from C to assembly, which also addresses the division by 0 problem usually encountered in writing a loop for a specified range setting by the programmer (you the master/creator of your code): http://www.tofla.iconbar.com/tofla/arm/arm02/

Other notes to do with your course material as well as Lab #2:

- Some might have questions about the course material, but since the time is condensed, and the class is big compared to other TA labs, I won't be able to answer outside of the lab scope. A question was raised on the two's complement, which is a lengthy mathematical discussion to cover, however, the short version can be explained, hereby exemplified: https://www.youtube.com/watch?v=NED9IIpteXA (this link also discusses about the complementary results in different machines as well as overflow (focus is on operation as well as size. In any case, size is always an issue, either in a buffer, stack or number stored by a register: "In a computer, the amount by which a calculated value is greater in magnitude than that which a given register or storage location can store or represent" http://en.wikipedia.org/wiki/Arithmetic_overflow)) which could be useful for your midterm exam as well.
- Also operands during operations will change the register content when written in assembly (or even C). For instance, what is the difference between in the order and result of execution by the operators in the following expression? (what is prioritized in the operation in this example?):

  (2+3)*5 against **2+3*5**

- The purpose of **push** and **pop** is also of importance during operations. For instance, in the operand case above, which value will be pushed first and popped out in the queue of operation over a stack, or an array of two or three registers?
- Lab manual and the relevant files are available at http://www.ece.uvic.ca/~ece255/lab/.
- Make sure to download all including the metadata and save into your workspace. Your workspace is created once you run (execute) Eclipse.
- However, based on my experience, the communication between the **microcontroller** and the **workstation**/PC sometimes create unstable responses, hangs and thus an on-hand real-time debugging solution is required. On occasion, we need to disconnect the microcontroller from the station and resume all over again by logging off and take the same steps.
- To investigate memory contents thoroughly between the memory map in eclipse and thereby study the hardware firsthand, run the **STM-32**

hardware program on your desktop. You may depict and mention these comparisons in your report next time if it helps your discussion/ analysis material.

Have a productive week,

Good luck,

Philip

==========

Philip B. Alipour,
Ph.D. Candidate in Electrical, Computer Engineering and Quantum Physics,
Dept. of Electrical and Computer Engineering, University of Victoria, V8W 3P6, Canada,
Office: ELW Room # A358,
Email: phibal12@uvic.ca or philipbaback_orbsix@msn.com
Homepage: http://web.uvic.ca/~phibal12/