Chapt. 2 CENG 255 TA Lab Log Progress Report

TA: Philip B. Alipour

General Log and Notes:

1. **What happened in the lab?**

i. Following our last session, an update was arranged by me and Leonard for the subsequent session which took place last week (closing day):
   - o Session unofficially started at 2:30 PM until 5:30 PM on Tuesday 14th, **Leonard** and I were available at the lab and it was open for those who were interested in working on the solution to access and ask questions regarding the second lab. No grades or assessments were made except some hints as we shall point out in this log in full.
   - o To this account, **tomorrow's session** will begin an hour before the formal hour for **2nd lab performance evaluation** and those pre-labs which have not been marked yet (for those still unmarked as indicated in your grades pdf file sent to you before,
   - o Please bring your **pre-labs** to this session.)
   - o Students can now have the chance to work longer and record their codes for their reports next week.

ii. Session will officially start at 2:30PM, unofficially at 1:30 PM. The formal duration is 2:30 to 5:20PM according to http://www.ece.uvic.ca/~ceng255/lab/information.html#schedule.

iii. Tomorrow session will be conducted with all students to be marked prior to their report submission next week on 28 Oct. 2014, 11:59PM.

iv. Marking results handouts and comments based on our agreement will be given to each individual with only their student number visible on screen or posted via email with the individual's grade.

v. The focus of this lab is to write your code in assembly to 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). **Note:** this is an assembly line-by-line comment style algorithm for you to translate and successfully run the code. If your code is based on this, as far as it fulfills all of the objectives, the 45% is granted.

vi. 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).

vii. I shall continue to assist students on their machines as problems occur on the debugger and software as well as giving hints on the lab and performance evaluation per group.

viii. It is expected to have 11 groups present. The time of evaluation will be compressed as each person must attend and perform. The other TA **Leonard**, shall assist mainly in his session. So some can stay at Leonard's class after 5:30PM if their code is still under development, since there are 4 more stations available in his lab. So evaluation can be fully done on 4 groups if left behind (no penalty considered).

ix. I 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 as of Lab 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).
- To understand how a high-level language can be translated to assembly and thus interpreted on a machine level, an email was sent to you with a way to translate on http://assembly.ynh.io/ as well as the 2nd lab manual code including all the commands and comments attached in that email. You might find both useful for the current solution.
- 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 and it will work accordingly!
- **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):

```
.data
primeArray:    .space         @type in a value for your space

       .text
       .global main

main:
               push {r3,r4,r5,r6,r7}
               mov    r4, lr
               push {r4}
//initialize i, n, load array address
               mov    r2, #3  @ r2 <- i=3
               mov    r3, #0  @ r3 <- n=0
                ldr    r0, primeArrary_add  @r0 <-obtain primeArrary address and put in r0
//end
/*****************************************************************************/
//1st 'for' loop
for1:
               cmp                    @ n<30
               bge                    @ if n>= 20, then stop loop
               mov                    @ r5 <- prime=1
               mov                    @ r6 <- put i to r6
               lsr                    @ r6 <- i/2
               mov r7, r6             @ r7 <- Limit=i/2
               mov r1, #2             @ r1 <- j=2
//2nd 'for' loop
```

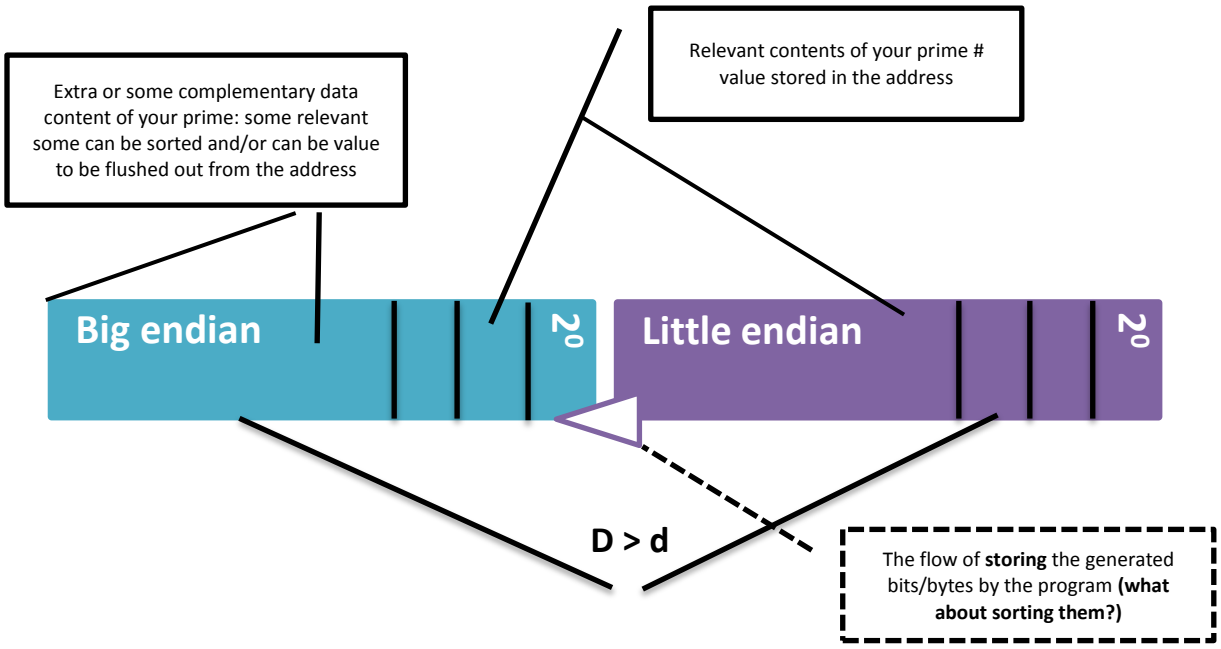
```
for2:
                                    @ if j < Limit
                bge     endfor2     @ j>= Limit
                mov             @ r4 <- r2, temporarily put i to r4 to calculate remainder
/*********************************************************************************/
//3rd 'for' loop to calculate the remainder i mod j
for3:                           @ need to compute the remainder = i mod j (r2 mod r1)
                                @ if r4 >0
                                @ if r4 >=0
                sub             @ r4 <- r4-r1 (r4 <- i-j)
                b       for3            @
endfor3:
/*********************************************************************************/
//verify i is prime or not
if1:
                cmp             @ if remainder == 0
                bne                             @ remainder != 0
                mov                     @ r5 <- prime = 0
                b       endfor2         @ end for2
fi1:
                add             @ r1 <- j++
                b                       @ goto for2
endfor2:
if2:
                cmp             @ if prime==1
                bne                     @ prime!=1
                                @ n++
                str             @ store i to primeArray
                add r0, r0, #4 @ adjust primeArray address to store next prime number
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
```

2.2. Hardware/software challenges

- Compared to our (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 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).

- 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:

- The http://assembly.ynh.io/ website will run and show you the flow of the logic behind the code on a different machine (the Hint Code text file sent to you before as you copy-pasted it and tried different translator settings, and subsequently highlighted specific portions of your code as you clicked on the line of code in C to Assembly).
- Although on the STM-32 microcontroller, the behavior is different and copy-pasting won't work from the translator during compilation (you'll get errors). However, it is a good resourceful indicator to figure out your code's flow and logic in implementing the code in aim of accomplishing your Lab 2 objective (Sec. 2.5 of the lab manual). So try to map the translated part to your code especially where the mod operation is concerned.
- 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 have had 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 as hereby exemplified: https://www.youtube.com/watch?v=SXAr35BiqK8 and https://www.youtube.com/watch?v=Hof95YlLQk0 (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/~ceng255/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 created unstable responses, hangs and thus an on-hand real-time debugging solution is required. On occasion (especially when Leonard had his group session), I had 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 on the Have a productive week,

Good luck,

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