COURSEWORK ASSIGNMENT | **UNIVERSITY OF EAST ANGLIA**
School of Computing Sciences

MODULE: CMP-5013A
ASSIGNMENT TITLE: Assembly Language Programming

DATE SET	**:**	**Week 7**		
DATE DUE	**:**	**15:00 Thursday 15/12/16 (via Blackboard)**		
RETURN DATE	**:**	**Week 1 (Spring Semester)**		
ASSIGNMENT VALUE	**:**	**SUMMATIVE**		
SET BY	**:**	**MHF**	**SIGNED:**	Mark Fisher
CHECKED BY	**:**	**MM**	**SIGNED:**	

Aim:
The purpose of this assignment is to help you learn about computer architecture, assembly language programming, and testing strategies. It also will give you the opportunity to learn more about the GNU/Unix programming tools, especially `vi`, `bash`, `make`, `gcc`, and `gas` for C and assembly language programs.

Learning outcomes:
- To become familiar with the ARMv8-A instruction set architecture.

Description of assignment:

See Attached Sheet.

Marking Scheme:

Task	Marks
Exercise 1a	10
Exercise 1b	30
Exercise 2a	5
Exercise 2b	5
Exercise 2c	30
Exercise 2d	20
Total	100

The Assignment contributes 30% of the overall mark for the module.

UNIVERSITY OF EAST ANGLIA

School of Computing Sciences

CMP-5013A — Architectures and Operating Systems

ASSIGNMENT S1 (V1.5) — ARM Assembly Language Programming

The purpose of this assignment is to help you learn about computer architecture and in particular, assembly language programming. It also will give you the opportunity to learn more about the GNU/Unix programming tools, especially `bash`, `make`, `gcc`, `gas`, and `gdb`.

Task 1: A Word Counting Program in Assembly Language

The Unix operating system has a command named `wc` (word count). In its simplest form, `wc` reads characters from the standard input stream until end-of-file, and prints to the standard output stream a count of how many lines, words, and characters it has read. A word is a sequence of characters that is delimited by one or more white space characters. For example, if the file named `proverb.txt` contains these characters:

```
Learning is a
treasure which
accompanies its
owner everywhere.
--  Chinese proverb
```

Then the command:

```
$ wc < proverb
```

prints this line to the standard output stream:

```
 5 12 83
```

The program `mywc.c` represents a naïve implementation of `wc`.

Exercise 1a: Test `mywc.c`

Write a `makefile` to build `mywc.c` and then write a bash script named `results.sh` that accepts a flag `-r` followed by the name of a command to run and a flag `-f` followed by a list of file names to print. e.g. `results.sh -r "./mywc < proverb.txt" -f "results.sh mywc.c makefile"`

1. Prints `results.sh` to `stdout`.
2. Prints `mywc.c` to `stdout`.

3. Prints the `makefile` to `stdout`.

4. Executes `make`.

5. Runs the program on `proverb.txt`.

Run `results.sh` redirecting output to a file named `resultsEx1a.txt` (i.e. use `>`).

Exercise 1b: Translate to Assembly Language

Translate `mywc.c` into assembly language, thus creating a file named `mywc.s`. If necessary, it is acceptable to use global (i.e. bss section or data section) variables in `mywc.s`. Translate the C code statement-by-statement and document the code by including the original source code statements as comments. Your assembly language program should have exactly the same behaviour (i.e. should write exactly the same characters to the standard output stream) as the given C program. Your assembly language program may call functions defined in `stdio.h`. Think in Geek [1] is a useful tutorial blog on Raspberry Pi Assembly Language; chapters 22 and 19 are particularly relevant.

Note: The Raspberry Pi 3 uses a Broadcom BCM2837 SoC with a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor [2]. **Use the AARM32 T32 (Thumb) ARM instruction set** [2] Part F. Write a `makefile` to build `mywc.s` and then use your bash script to:

1. Print `mywc.s` to `stdout`.

2. Print the `makefile` to `stdout`.

3. Execute `make`

4. Runs the program on `proverb.txt`.

Run `results.sh` redirecting output to a file named `resultsEx2a.txt`. (i.e. use `>`).

Task 2: Beat the Compiler

Many programming environments contain modules to handle high-precision integer arithmetic. For example, the Java Development Kit (JDK) contains the BigDecimal and BigInteger classes.

Fibonacci numbers are often used in computer science, see [3] for some background information on Fibonacci numbers. Note in particular that Fibonacci numbers can be very large integers.

This part of the assignment asks you to write a minimal high-precision integer arithmetic module, and use it to compute large Fibonacci numbers.

Exercise 2a: Add `BigInt` Objects Using C Code

Suppose you must compute Fibonacci number 500000, that is, fib(500000)...

The CMP-5013A blackboard assignment folder contains a C program that computes Fibonacci numbers. It consists of two modules: a client module and a `BigInt` ADT (Abstract Data Type). The client consists of the file `fib.c`. The client accepts an integer `n` as a command-line argument, validates it, and computes and prints `fib(n)` to `stdout` as a hexadecimal number. It prints to the standard error stream the amount of CPU time consumed while performing the computation. The client module delegates most of its work to `BigInt` objects.

The `BigInt` ADT performs high precision integer arithmetic. It is a minimal ADT; essentially it implements only an 'add' operation. The `BigInt` ADT consists of four files:

- `bigint.h` is the interface. Note that the ADT makes four functions available to clients: `BigInt_new`, `BigInt_free`, `BigInt_add`, and `BigInt_writeHex`.
- `bigint.c` contains implementations of the `BigInt_new`, `BigInt_free`, and `BigInt_writeHex` functions.
- `bigintadd.c` contains an implementation of the `BigInt_add` function.
- `bigintprivate.h` is a "private header file" – private in the sense that clients never use it. It allows code sharing between the two implementation files, `bigint.c` and `bigintadd.c`.

Write a `makefile` to build the program with no optimization and then use your bash script to:

1. Print the `makefile` to `stdout`.
2. Execute `make`
3. Run the program to compute `fib(500000)`.

Run the script redirecting output to a file named `resultsEx2a.txt`. (i.e. use `>`). Note the amount of CPU time consumed.

Exercise 2b: Add `BigInt` Objects Using C Code with Compiler Optimization

Suppose you decide that the amount of CPU time consumed is unacceptably large. You decide to command the compiler to optimize the code that it produces...

Write a `makefile` to build the program using optimization. Specifically, specify the `-D NDEBUG` option so the preprocessor disables the assert macro, and the `-O3` option so the compiler generates optimized code. Use your bash script to:

1. Print the `makefile` to `stdout`.
2. Executes `make`
3. Run the program to compute `fib(500000)`.

Run the script redirecting output to a file named `resultsEx2b.txt`. (i.e. use `>`). Note the amount of CPU time consumed.

Exercise 2c: Add `BigInt` Objects Using Assembly Language Code

Suppose, further analysis shows that most CPU time is spent executing the `BigInt_add` function. In an attempt to gain speed, you decide manually to code the `BigInt_add` function in assembly language...

Use the AARM32 A32 ARM instruction set. Manually translate the C code in the `bigintadd.c` file into assembly language, thus creating the file `bigintadd.s`. You need not translate the code in other files into assembly language.

Your assembly language code should store all variables in memory. It should contain definitions of the `BigInt_add` and `BigInt_larger` functions; the former should call the latter, just as the C code does.

Note that assert is a parameterized macro, not a function. (See Section 14.3 of the King book for a description of parameterized macros.) So you cannot call assert from assembly language code. When translating `bigintadd.c` to assembly language, simply pretend that the calls of assert are not in the C code.

Build the program consisting of the files `fib.c`, `bigint.c`, and `bigintadd.s` using the `-D NDEBUG` and `-O3` options. Write a `makefile` to the program consisting of the files `fib.c`, `bigint.c`, and `bigintadd.s` using the `-D NDEBUG` and `-O3` options. Use your bash script to:

1. Print `bigintadd.s` to `stdout`.
2. Print the `makefile` to `stdout`.
3. Execute `make`.
4. Run the program to compute `fib(500000)`.

Run the script redirecting output to a file named `resultsEx2c.txt`. (i.e. use `>`). Note the amount of CPU time consumed.

Part 2d: Add `BigInt` Objects Using Optimized Assembly Language Code

Suppose, to your horror, you discover that you have taken a step backward: the CPU time consumed by your assembly language code is approximately the same as that of the non-optimized compiler-generated code. So you decide to optimize your assembly language code...

Use the AARM32 A32 or AARM32 T32 ARM instruction set. Manually optimize your assembly language code in `bigintaddopt.s`, thus creating the file `bigintaddopt.s`. Specifically, perform these optimizations:

- If you're not already doing so, store local variables (but not parameters) in registers.
- 'inline' the call of the `BigInt_larger` function. That is, eliminate the `BigInt_larger` function, placing its code within the `BigInt_add` function.
- (Optionally) translate your ARM code into Thumb (You should find Thumb code runs faster).

Build the program consisting of the files `fib.c`, `bigint.c`, and `bigintaddopt.s` using the `-D NDEBUG` and `-O3` options. Write a `makefile` to the program consisting of the files `fib.c`, `bigint.c`, and `bigintadd.s` using the `-D NDEBUG` and `-O3` options. Use your bash script to:

1. Print `bigintaddopt.s` to `stdout`.
2. Print the `makefile` to `stdout`.
3. Execute `make`
4. Run the program to compute `fib(500000)`.

Run the script redirecting output to a file named `resultsEx2d.txt`. (i.e. use `>`). Note the amount of CPU time consumed.

What to submit

You should submit:

- Files `resultsEx1a.txt`, `resultsEx1b.txt`, `resultsEx2a.txt`, `resultsEx2b.txt`, `resultsEx2c.txt` and `resultsEx2d.txt`.
- A file named `readme.txt`.

Your `readme.txt` file should contain:

- Your name.
- A description of whatever help (if any) you received from others while doing the assignment, and the names of any individuals with whom you collaborated.
- The times consumed by the fib programs, as specified above.
- (Optionally) An indication of how much time you spent doing the assignment.
- (Optionally) Your assessment of the assignment.
- (Optionally) Any information that will help us to grade your work in the most favorable light. In particular you should describe all known bugs.

Submit your work electronically by uploading to Blackboard.

References

[1] Think in Geek, ARM assembler in Raspberry Pi,
http://thinkingeek.com/arm-assembler-raspberry-pi/
Last Accessed, Nov. 2016.

[2] ARM, ARM Architecture Reference Manual: ARMv8, for ARMv8-A architecture profile,
`http://115.28.165.193/down/arm/arch/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf`
Last Accessed, Nov. 2016.

[3] Anon. Fibonacci number,
`/wiki/Fibonacci_numbers`
Last Accessed, Nov. 2016.

Dr. Mark Fisher
Nov. 2016