# 2GA3 Assignment 3

Elite Lu

There is no Question 1, as that was a question meant to learn assembly. In addition to this, I will be writing the comments and how my assembly code works in this document.

## Question 2

For the original outputs when I entered in the value 4, I would get 0x04 for both the C and the assembly code. I modified the assembly code to accommodate for 0xBD5B7DDE by loading it as an int, and then ldr (load function) the address into a variable. After that, I was able to print out the value.
In order to shift this signed integer constant of the hexadecimal value 0xBD5B7DDE, I will first determine the binary, which is 10111101010110110111110111011110. With this, I will consider the first bit as that is the sign. I will shift every bit right by removing the 0 at the end, making it 10111101010110110111111011101111. Then, I will the sign bit at the front, making it 11011110101011011011111011101111, which is written as DEADBEEF in hexadecimal (ha ha).

## Question 3

For this question, I used the bitwise operator ^ in C. This operator checks each corresponding bit for two values and performs the XOR operation. This operation is exclusive or, which means that it will only return true when only one variable returns true. In assembly, I used a similar method by using the operator EOR, which is ARM's equivalent of XOR. From this, I was able to generate the outputs by printing out the values together. I compared them manually and found that the outputs are the same. The outputs are shown together on each print line statement of the C code.

## Question 4

### Part A

The difference between horizontal and vertical microcode is that vertical microcode is sequential and controls one functional unit at a time whereas horizontal microcode controls multiple functional units at a time and less intuitive, requiring more understanding of the hardware.

### Part B

Here the operations the methods operands are addressed in a computer:

1) The immediate value encoded in the instruction
2) The value is in the register, register ID, and the instruction
3) The value is in memory and the memory address is in the instruction
4) The value is in memory, the memory address is in the register, and the register ID is in the instruction
5) The value is in memory, the address is in a different memory location encoded in the instruction

The fastest one would be 1, as having the immediate value encoded in the instruction.

### Part C

A two-stage assembler has three main steps, as the first "step" is not considered a step. This includes the following

0) Run pre-processor
1) Assign an address to every instruction and construct a label symbol table
2) Translate the instructions to binary and substitute label with location from symbol tab

For example, if I was given an instruction set, I would first run the pre-processor, then for each type of operation, I would assign an opcode. The number of bits would depend on the number of operations in the

opcode set. In addition to this, I will designate a set number of bits for the operands and the offset, which is dependent on the remaining bits. Finally, I will translate the instructions into binary and replace labels with locations from the symbol table. Some examples would be to replace the words main in the main function, for in a for loop, or even if from an if statement with a binary representation. Here is a demonstration with some sample code. Assume that we have these commands (from the lecture slides):

| Instruction | Opcode | r0 | r1 | Offset |
|---|---|---|---|---|
| **add** | 000 | XXX | XXX | ------- |
| **load** | 001 | XXX | XXX | XXXXXXX |
| **store** | 010 | XXX | XXX | XXXXXXX |
| **cmp** | 011 | XXX | XXX | ------- |
| **bne** | 100 | --- | --- | XXXXXXX |
| **be** | 101 | --- | --- | XXXXXXX |
| **jmp** | 110 | --- | --- | XXXXXXX |
| **imd** | 111 | XXX | --- | XXXXXXX |

This type of encoding ensures that the least number of bits are used for the opcode, r0, and r1 to optimize the number of bits for the offset. Above assumes that there are only 8 registers and 8 commands along with only having access to 16 bits.
Next, here is some sample code:

```
main:   imd r0, 10
        imd r1, 1
        imd r2, 0
for:    cmp r0 , r2
        be endfor
        add r2 , r1 #increment r2
        jmp for #loop again
endfor: jmp main
```

With this, I will encode each line. In addition to this, I will make a symbol table for the labels. The corresponding bytes will be present:

| Address | Byte 1 | Byte 2 |
|---|---|---|
| **0x00** | 111 000 00 | 0 0001010 |
| **0x02** | 111 001 00 | 0 0000001 |
| **0x04** | 111 010 00 | 0 0000000 |
| **0x06** | 011 000 01 | 0 0000000 |
| **0x08** | 101 000 00 | 0 0000110 |
| **0x0a** | 000 010 00 | 1 0000000 |
| **0x0c** | 110 000 00 | 0 1111010 |
| **0x0e** | 110 000 00 | 0 1110010 |

After this, I will add the corresponding symbol tree:

| Address | Label |
|---|---|
| **0x00** | main |
| **0x06** | for |
| **0x0e** | endfor |

I added spaces between the values to represent the bits used for the opcode, operands, and the offset. To calculate the offset for the operations of branching, I subtract the destination byte with the current byte.