CPE 390 – Microprocessor Systems
Lab 2

Michael Savo

"I pledge my honor that I have abided by the Stevens Honor System."

Exercises:

1. Consider memory storage of a 32-bit word stored at memory word 102 in a byte-addressable memory.

(a) What is the byte address of memory word 102?

4*102 = 408 in decimal and 0x198 in hexadecimal.

(b) What are the byte addresses that memory word 102 spans? (this should include your answer from part a with the additional appropriate addresses)

102 would span 4 addresses: 0x198, 0x199, 0x19A, 0x19B

(c) Draw the number 0x12345678 stored at word 102 in both big-endian and little-endian machines. Clearly label the byte address corresponding to each data byte value.

Big-endian:

0x198	12
0x199	34
0x19A	56
0x19B	78

Little-endian:

0x19B	78
0x19A	56
0x199	34
0x198	12

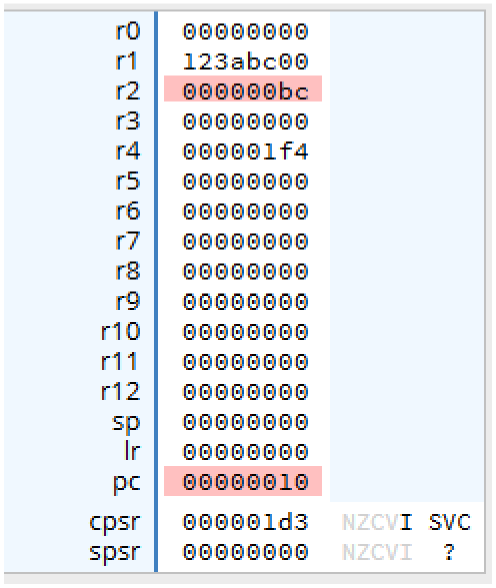
2. (a) Explain how the following ARM program can be used to determine whether a computer is big-endian or little-endian:

```
MOV r4, #500
LDR r1, =0x123ABC00
STR r1, [r4]
LDRB r2, [r4, #1]
```

This will set the register r4 to correspond with the memory address 500 and it will load the word 0x123ABC00 into the r1 register. STR r1, [r4] will store that word into the memory address 500 because r4 is attached to it. LDRB loads a register with a specific byte. Specifically, this has a #1 offset so it will show what byte is in memory address

501 and load it to r2. If it contains the least significant byte, it will indicate a little-endian system, while the most significant byte would indicate a big-endian system.

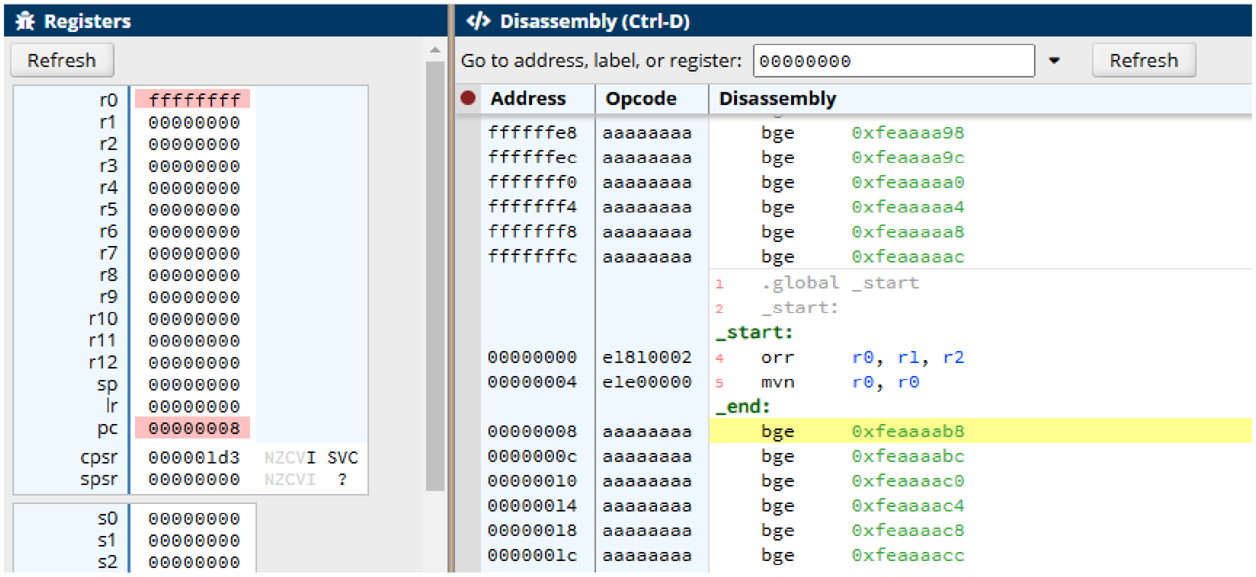
(b) Use the following ARM simulator to verify your explanation: https://cpulator.01xz.net/?sys=arm Include a screenshot of your results and state whether the system is big-endian or little-endian.
After using the ARM simulator, I found that it uses a little-endian system because it stores the byte bc in r2 which is towards the end of the word. The system starts with the least significant byte.



```
  r0   00000000
  r1   123abc00
  r2   000000bc
  r3   00000000
  r4   000001f4
  r5   00000000
  r6   00000000
  r7   00000000
  r8   00000000
  r9   00000000
 r10   00000000
 r11   00000000
 r12   00000000
  sp   00000000
  lr   00000000
  pc   00000010

cpsr   000001d3   NZCVI SVC
spsr   00000000   NZCVI  ?
```

3. The NOR instruction is not part of the ARM instruction set, because the same functionality can be implemented using existing instructions. Write a short assembly code snippet that has the following functionality: R0 = R1 NOR R2. Use as few instructions as possible. Include a screenshot of the code and explain your process.

You can do:
ORR R0, R1, R2
MVN R0, R0



```
  r0   ffffffff
  r1   00000000
  r2   00000000
  r3   00000000
  r4   00000000
  r5   00000000
  r6   00000000
  r7   00000000
  r8   00000000
  r9   00000000
 r10   00000000
 r11   00000000
 r12   00000000
  sp   00000000
  lr   00000000
  pc   00000008

cpsr   000001d3   NZCVI SVC
spsr   00000000   NZCVI  ?

  s0   00000000
  s1   00000000
  s2   00000000
```

```
Address    Opcode     Disassembly
ffffffe8   aaaaaaaa      bge      0xfeaaaa98
ffffffec   aaaaaaaa      bge      0xfeaaaa9c
fffffff0   aaaaaaaa      bge      0xfeaaaaa0
fffffff4   aaaaaaaa      bge      0xfeaaaaa4
fffffff8   aaaaaaaa      bge      0xfeaaaaa8
fffffffc   aaaaaaaa      bge      0xfeaaaaac
                         .global _start
                         _start:
                      _start:
00000000   e1810002      orr      r0, r1, r2
00000004   e1e00000      mvn      r0, r0
                      _end:
00000008   aaaaaaaa      bge      0xfeaaaab8
0000000c   aaaaaaaa      bge      0xfeaaaabc
00000010   aaaaaaaa      bge      0xfeaaaac0
00000014   aaaaaaaa      bge      0xfeaaaac4
00000018   aaaaaaaa      bge      0xfeaaaac8
0000001c   aaaaaaaa      bge      0xfeaaaacc
```

You can see that the first line of code computes r1 and r2 and stores it in r0. Then, the second line takes the result in r0 and inverts the result so it would be equivalent to a NOR.

4. Matrices do not directly exist in the ARM architecture, but we can make our own with a little bit of work. Assume matrix 1 is a 2x2 matrix stored from addresses 0x20000000 through 0x2000000C and matrix 2 is a 2x2 matrix stored from addresses 0x20000010 through 0x2000001C. We want to add the two matrices together by loading from one spot of matrix 1 into register 1, loading from the same spot of matrix 2 into register 2, and adding those values while keeping the result of the addition in register 3. This result in register 3 should be stored to the corresponding matrix 3 location, which is located at addresses 0x20000020 through 0x2000002C. The “matrices” below show what the full process should re-create. Write assembly code to accomplish this objective! Include a screenshot of the code and explain your process.

Matrix 1:

The value at 0x20000000	The value at 0x20000004
The value at 0x20000008	The value at 0x2000000C

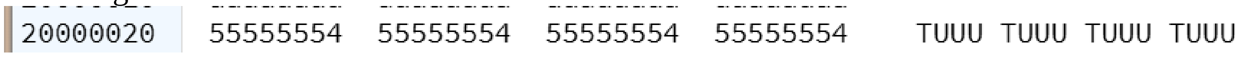
Matrix 2:

The value at 0x20000010	The value at 0x20000014
The value at 0x20000018	The value at 0x2000001C

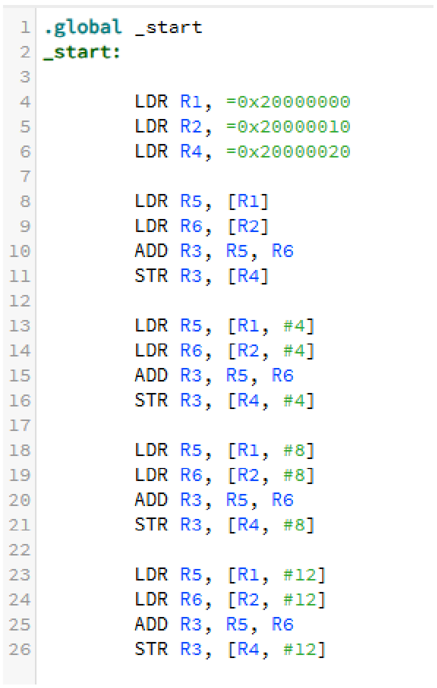
Matrix 3:

The sum of the value at 0x20000000 and the value at 0x20000010, now stored at 0x20000020	The sum of the value at 0x20000004 and the value at 0x20000014, now stored at 0x20000024
The sum of the value at 0x20000008 and the value at 0x20000018, now stored at 0x20000028	The sum of the value at 0x2000000C and the value at 0x2000001C, now stored at 0x2000002C

If you “go to” address 20000000 in the Memory window of your simulator, you should see the following at the end:



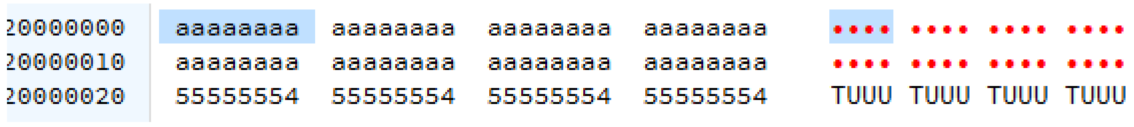
```
20000020   55555554  55555554  55555554  55555554    TUUU TUUU TUUU TUUU
```



```
.global _start
_start:

        LDR R1, =0x20000000
        LDR R2, =0x20000010
        LDR R4, =0x20000020

        LDR R5, [R1]
        LDR R6, [R2]
        ADD R3, R5, R6
        STR R3, [R4]

        LDR R5, [R1, #4]
        LDR R6, [R2, #4]
        ADD R3, R5, R6
        STR R3, [R4, #4]

        LDR R5, [R1, #8]
        LDR R6, [R2, #8]
        ADD R3, R5, R6
        STR R3, [R4, #8]

        LDR R5, [R1, #12]
        LDR R6, [R2, #12]
        ADD R3, R5, R6
        STR R3, [R4, #12]
```

The first three lines of code are to load the addresses of the three matrices to a corresponding register. The first two matrices are loaded onto register r1 and r2 because the problem indicated that I use those, and the elements of matrix 3 will be loaded onto r4 temporarily while r3 contains the sum of matrix 1 and 2.

The code is then split into 4 sections of loading, adding and storing. The first two LDR lines are for loading the corresponding matrix element from the first and second matrix. ADD R3, R5, R6 adds the elements in the matrix and places the result in r3. Finally, the addition result is stored in matrix 3. This is repeated three more times to cover the other elements in the 2x2 matrices.



```
20000000   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    .... .... .... ....
20000010   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    .... .... .... ....
20000020   55555554  55555554  55555554  55555554    TUUU TUUU TUUU TUUU
```