[12 marks] 2. Consider the ARM assembly-language program that is shown below.

```
          .global   _start
_start:
          MOV       R0, #0
          MOV       R2, #N
          LDR       R3, [R2]
LOOP:     ADDS      R3, R3
          BCC       CONT
          ADD       R0, #1
CONT:     MOVS      R3, R3
          BNE       LOOP

          STR       R0, [R2, #4]
END:      B         END

N:        .word     0xAF13           // the data
Result:   .word     0                // space for the result
```

(a) Using as few words as possible explain what this code "does". That is, for a given value of N, what corresponding value will the program generate and store in *Result*? Note that the condition `CC` used in the `BCC` instruction means *Carry Clear*. Hence, the `BCC` branch will be taken when the *c* flag is 0.

Answer:

Counts the number of 1 bits in the word of memory at N, and stores the count in Result.

Part (b) of this question is on the next page . . .

[10 marks] 3. For this question you are to write an assembly-language program for the ARM processor. The program should perform the following task: in an infinite loop read from the SW port (address 0xFF200040). If the value read from SW is 0, 1, 2, or 3 then you should display the characters 'g', 'o', 'o', and 'd', respectively, on the HEX0 display (address 0xFF200020). You should *error-check* the value read from the SW port; if it is greater than 3 then you should display a '-' on HEX0.

For the 7-segment display recall that segment 0 is at the top and then the segments are ordered from 1 to 5 in the clockwise direction, with segment 6 in the middle.

Show your ARM assembly-language code in the space below:

```
// Displays g, o, o, d on 7-seg HEX0 based on SW setting
.global _start
_start:
                MOV   R2, #HEX_ADDR   // base address of HEX3_0 port
                LDR   R2, [R2]
                MOV   R3, #SW_ADDR    // base address of SW port
                LDR   R3, [R3]
                MOV   R4, #HEX_bits

DO_DISPLAY:     LDR   R0, [R3]        // load SW switches
                MOV   R1, #0b01000000 // '-' character
                CMP   R0, #3
                BGT   DASH
                LDRB  R1, [R4, R0]
DASH:           STR   R1, [R2]        // write to HEX0
                B     DO_DISPLAY

HEX_ADDR: .word   0xFF200020
SW_ADDR:  .word   0xFF200040
HEX_bits: .word   0x5E5C5C6F    // 'd','o', 'o', 'g'
                // or
          .byte 0x6f, 0x5c, 0x5c, 0x5e // because little endian
```

[8 marks] 4. Consider the ARM code shown below. Note that the address of each instruction in the memory is shown to the left of the code. The directive `.asciz "ha ho ha ho"` places the given ASCII characters (bytes) in memory, including a 0 byte to designate the end of the string.

```
                         .text
                         .global _start

00000000 _start:         MOV     R0, #S1
00000004                 MOV     R1, #S2
00000008                 BL      Goober
0000000C Stop:           B       Stop

00000010 Goober:         MOV     R2, R0
00000014 Freakshow:      LDRB    R3, [R2]
00000018                 CMP     R3, #0
0000001C                 BEQ     DoDat
00000020                 ADD     R2, #1
00000024                 B       Freakshow

00000028 DoDat:          CMP     R2, R0
0000002C                 BEQ     NoMoe

00000030                 SUB     R2, #1
00000034                 LDRB    R3, [R2]
00000038                 STRB    R3, [R1]
0000003C                 ADD     R1, #1
00000040                 B       DoDat
00000044 NoMoe:          MOV     R3, #0
00000048                 STRB    R3, [R1]
0000004C                 MOV     PC, LR

00000050 S1:            .asciz "ha ho ha ho"
0000005C S2:            .asciz " "

                        .end
```

(a) If this program is executed on the ARM processor, what would be the values that would be shown in a debugger the **first** time the code reaches the instruction at address 0x28.

R0	0x50	R1	0x5C	R2	0x5B
R3	0x0	R14	0xC	R15	0x28

(b) What does this code "do"? That is, given the string "ha ho ha ho" what does the program produce?

Answer

Reverses the string, and writes it to S2

"ha ho ha ho" -> "oh ah oh ah"

[7 marks] 5. An ARM program is shown below, which is supposed to work as follows. There is a main program that reads two integers, A and B, from the memory, multiplies them to produce $C = A \times B$, and then stores the result in memory. Instead of using the ARM `MUL` instruction, this code does the multiplication by using a subroutine, called `MULTIPLY`, which performs repeated addition (similar to the code that you wrote in your lab exercises that used repeated subtraction to do division).

Unfortunately, the `MULTIPLY` subroutine has been written by an aging professor, who has "lost it," and the subroutine code contains some errors. Your job is to find the errors, and fix them.

```
          .text
          .global _start
_start:   MOV     R0, #A
          LDR     R0, [R0]
          MOV     R1, #B
          LDR     R1, [R1]
          BL      MULTIPLY
          MOV     R2, #C
          STR     R0, [R2]
END:      MOV     R15, #END

MULTIPLY: MOV     R3, R1
          BEQ     EMULT
          MOV     R3, R0
CONT:     CMP     R1, #0
          BNE     EMULT
          ADD     R3, R3
          SUB     R1, #1
          B       CONT
EMULT:
          MOV     PC, #END

A:        .word   10
B:        .word   10
C:        .word   0
```

Answer the questions on the next page.

(a) How many errors did you find in the subroutine code?

Answer 6

(b) Briefly describe each of the errors that you found, in the space below. Note that the lines of code are numbered for convenience of reference.

Line 12: movs instead of mov

Line 13: need to decrement r1 either before or after this instruction

Line 16: should be beq instead of bne

Line 17: should be add r3, r3, r0 (or just add r3, r0)

Line 20: need mov r0, r3 to properly place return value

Line 21: need to use mov PC, LR to return from a subroutine

(c) In the space below provide a corrected version of the `MULTIPLY` subroutine.

```
MULTIPLY:
            MOVS   R3, R1
            BEQ    EMULT   // return 0
            SUB    R1, #1  // control how many times to add
            MOV    R3, R0  // use R3 to accumulate the result
CONT:       CMP    R1, #0
            BEQ    EMULT
            ADD    R3, R0
            SUB    R1, #1
            B      CONT
EMULT:      MOV    R0, R3
            MOV    PC, LR
```