Problem 1 (5 points)
Given the following LC-3 program:

```
.ORIG x3000
AND R5, R5, #0
AND R3, R3, #0
ADD R3, R3, #8
LDI R1, A ; R1 = A
ADD R2, R1, #0 ; R2 = A
AG ADD R2, R2, R2 ; R2 = R2 << 1
ADD R3, R3, #-1 ; done 8 times
BRnp AG
LD R4, B ; R4 = B
AND R1, R1, R4 ; R1 = A AND B
NOT R1, R1
ADD R1, R1, #1 ; R1 = - (A AND B)
ADD R2, R2, R1 ; R2 = A << 8 - (A AND B)
BRnp NO
ADD R5, R5, #1
NO HALT
B .FILL xFF00
A .FILL x4000
.END
```

a. What does the above program do? (1 point)
Compress if highest 8 bits = lowest 8 bits of value pointed by A. If so, then set r5 = 1. else, r5 = 0

b. Create the symbol table generated by the assembler when translating the above routine into machine code: (2 points)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>0x3005</td>
</tr>
<tr>
<td>NO</td>
<td>0x300F</td>
</tr>
<tr>
<td>B</td>
<td>0x3010</td>
</tr>
<tr>
<td>A</td>
<td>0x3011</td>
</tr>
</tbody>
</table>

c. The following is the machine code for the above program. Fill in the blank lines with the appropriate machine code. (2 points)

```
0101 1011 0110 0000 ; AND R5, R5, #0
0101 0110 1110 0000 ; AND R3, R3, #0
```
Problem 2 (3 points)
The following code has three syntax errors in it. Mark each instruction that has an error and explain why it is wrong.

```assembly
.ORIG x3000
NOT R0, R0, #0  ; May not take 3 args / immediate val
LD R2, ZERO
LD R0, M0
LD R1, M1
LOOP
BRz DONE
ADD R2, R2, R0
ADD R1, R1, #30  ; #30 out of immediate range
BR x3003  ; x3003 out of PCoffset range
DONE
ST R2, RESULT
HALT
RESULT .FILL x0000
ZERO .FILL x0000
M0 .FILL x0004
M1 .FILL x0803
.END
```

The following code has three syntax errors in it. Mark each instruction that has an error and explain why it is wrong.

```assembly
0001 0110 1110 1000 ; ADD R3, R3, #8
1010 0010 0000 1101 ; LDI R1, A
0001 0100 0110 0000 ; ADD R2, R1, #0
0001 0100 1000 0010 ; AG ADD R2, R2, R2
0001 0110 1111 1111 ; ADD R3, R3, #-1
0000 1011 1111 1101 ; BRnp AG
0101 0010 0000 1101 ; LD R4, B
1001 0010 0111 1111 ; NOT R1, R1
0001 0100 0110 0001 ; ADD R1, R1, #1
0001 0100 1000 0001 ; ADD R2, R2, R1
0000 1010 0000 0001 ; BRnp NO
0001 1011 0110 0001 ; NO HALT
1111 1111 0000 0000 ;B .FILL xFF00
0100 0000 0000 0000 ;A .FILL x4000
```
Problem 3 (4 points)
The following subroutines implement a data structure called a stack at memory location STACK that holds up to 100 16-bit integers. When an element is “pushed” into the stack, it is inserted at the “top” of the data structure. When an element is “popped” off the stack the “top” element of the stack is removed. Assume the value at STACK_PTR holds the address of the top of the stack where the next element will be inserted. STACK_PTR increases as the stack grows and decreases as numbers in the stack are removed. MAX_SIZE is the maximum number of elements that can be stored in the stack.

Complete the push and pop subroutines given in the descriptions below:

a. The push subroutine takes R0 as the element to be inserted into the stack. The stack is full if STACK_PTR = STACK + MAX_SIZE. If it is full, we return without adding the element. If it is not full, we store R0 in the location specified by STACK_PTR and increment STACK_PTR. Fill in the following blanks. (2 points)

;Push: Inserts R0 into the stack
PUSH ST R1, R1_TMP
ST R2, R2_TMP
LD R1, MAX_SIZE
LEA R2, STACK
ADD R2, R1, R2 ; R2 = STACK+MAX_SIZE
NOT R2, R2 ; R2 = !(STACK+MAX_SIZE)
LD R1, STACK_PTR ; R1 = STACK_PTR
AND R2, R1, R2

ALTERNATIVELY:
ADD R2, R2, #1
ADD R2, R1, R2
BRz PUSH_RET ; If 0, full
STR R0, R1, #0
ADD R1, R1, #1
ST R1, STACK_PTR
LD R2, R2_TMP
LD R1, R1_TMP
PUSH_RET RET
b. The pop subroutine will first check if the stack is empty by comparing STACK with STACK_PTR. If the stack is empty, return from the function without doing anything. If it is not empty, remove the top element of the stack by decrementing STACK_PTR by 1. **Fill in the following blanks.** (2 points)

;Pop: Remove the most recently inserted element from the stack

```assembly
POP ST R0, R0_TMP
ST R1, R1_TMP
LEA R0, STACK
LD R1, STACK_PTR ; R1 = STACK_PTR
NOT R0, R0
ADD R0, R0, #1
ADD R0, R0, R1 ; R0 = STACK_PTR - STACK
BRz POP_RET
ADD R1, R1, #-1
ST R1, STACK_PTR
LD R1, R1_TMP
LD R0, R0_TMP
POP_RET RET
```
Problem 4 (8 points)

In communication networks it is necessary to check for errors in a message sent from another device. One way to do this is by performing a cyclic redundancy check (CRC) on the message. This is done by performing the CRC calculation on a received message and checking whether the result is 0. If it is 0 there is no error in the message, and if it is non-zero there is an error in the message. The following pseudo code describes the algorithm for the CRC calculation using a 3-bit CRC divisor. The subroutine is given one input parameter, which is the message on which to perform the calculation.

```
CRC_calc(message){
    num_shifted = 0 // # of times result has been left shifted
    divisor = 0xB000
    result = message
    loop while (num_shifted < 13){
        if ( MSB of result = 1)
            result = result XOR divisor
        left_shift(result)
        num_shifted = num_shifted + 1
    }
    result = high3(result); // Get highest 3 bits of result
    return result
}
```

Download the template code for this problem: hw7_p4.asm. Modify the template according to the tasks below. **DO NOT CHANGE THE LABELS.**

a. (2 points) Implement the XOR subroutine in LC-3 assembly code. Assume the following:
   - Z = A XOR B = \overline{AB} + AB \quad \text{(Hint: DeMorgan’s Law)}
   - R0 and R1 are the registers holding the operands for the XOR operation
   - R2 is the register that holds the result of the XOR subroutine

b. (6 points) Implement the subroutine CRC_CALC as described above in LC-3 assembly code. Assume the following:
   - R0 is the register that holds the input message
   - R1 is the register that holds the result of the subroutine

The solution code is available here: hw7_p4_SOL.asm
Problem 5 (6 points)
It is possible to implement division with a constant number of loops. Typically this is less than using the subtraction method of homework 6. For instance, dividing 512 by 1 using the old method would require 512 loops, but using the following algorithm would only require 16. The pseudocode below describes the new process for the division of two unsigned 16-bit input values that stores its outputs as an unsigned 16-bit quotient and an unsigned 16-bit remainder.

```plaintext
// quotient = n / d
// remainder = remainder n / d
DIVIDE(n, d){
    if (divisor = 0){
        stop the program. Error!
    } else{
        quotient = 0  // Initialize quotient and remainder
        remainder = 0
        mask = 0x8000  // The mask will be used to select bits from n
        for i = 16 to 1 {
            remainder = left_shift(remainder)  // left shift R by 1 bit
            if ( (mask AND n) != 0){
                remainder = remainder + 1
            }
            if (remainder >= d){
                remainder = remainder - d
                quotient = quotient + mask
            }
            mask = right_shift(mask)  // Right shift mask by 1 bit.
        }
    }
}
```

Download the template code for this problem: hw7_p5.asm. DO NOT CHANGE THE LABELS.

Implement the DIVIDE program as depicted above. You are given the RSHMASK subroutine which will take the value stored at MASK, right shift by one bit, and store the result back into MASK. Assume the following:
- Your inputs are stored at N (dividend) and D (divisor)
- Your outputs should be stored at Q (quotient) and R (remainder)

The solution code is available here: hw7_p5_SOL.asm
Problem 6 (4 points)
The following LC-3 program counts the number of ones in a 16-bit word stored at WORD:

```
.ORIG x3000
AND   R0, R0, #0
LD    R1, LABEL
LD    R2, WORD
A    BRzp B
   ADD  R0, R0, #1
B    ADD  R1, R1, -1
   BRz  C
   ADD  R2, R2, R2
   BR   A
C    ST   R0, FINAL
END   HALT

LABEL  .FILL #16
WORD   .BLKW #1
FINAL  .BLKW #1
.END
```

1. (3 points) Fill in the blanks above. There should be only one instruction per line.
2. (1 point) Which instruction can be changed so that the program will count the number of 0s in WORD instead of 1s? Show the new instruction. Only one instruction should be changed. (Example: ADD R0, R0, 0 -> AND R0, R0, 0)

BRzp B → BRn B