 Name: _____

Problem 5 (20 points): The LC-3 Instruction Set

Parts A and B refer to the LC-3 code below (execution starts at x3000 and ends when the HALT trap is decoded).

Address	Contents	Instruction
x3000	1110 000000000101	LEA R0, #5
x3001	0110 010000000001	LDR R2, R0, #1
x3002	0111 010000000000	STR R2, R0, #0
x3003	1010 011000000001	LDI R3, #1
x3004	1111 000000100101	TRAP x25 (HALT)
x3005	0011 000000001000	*not executed*
x3006	1011 101010101101	*not executed*
x3007	0110 000000001100	*not executed*
x3008	0011 010101101010	*not executed*

Part A (4 points): How are the contents of registers and the memory locations shown above modified by the code when it runs? Specify the final value stored at each memory location and register changed by the code.

Part B (4 points): You also need to figure out how long this program takes to execute. Knowing that the majority of the time spent will be on memory accesses, you decide to count the number of memory accesses to estimate the time.

How many times does LC-3 need to access memory to execute this snippet of code (until decode of the TRAP)? Justify your answer.

 Name: _____

***** Part D** (4 points): Explain what the code below does. A description that requires more than a few words is a good hint that you have the wrong idea. *Hint: R0 and R1 are positive inputs, and R5 is the output.*

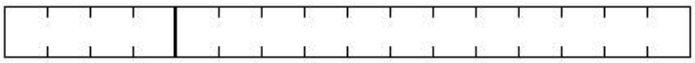
Address	Data	Instruction
x3000	0101 010000100000	AND R2, R0, #0
x3001	0001 101010100001	ADD R5, R2, #1
x3002	0001 011000100000	ADD R3, R0, #0
x3003	0001 010010000101	ADD R2, R2, R5
x3004	0001 011011111111	ADD R3, R3, #-1
x3005	0000 001111111101	BRp #-3
x3006	0001 101010100000	ADD R5, R2, #0
x3007	0101 010010100000	AND R2, R2, #0
x3008	0001 001001111111	ADD R1, R1, #-1
x3009	0000 001111111000	BRp #-8
x300A	1111 000000100101	TRAP x25

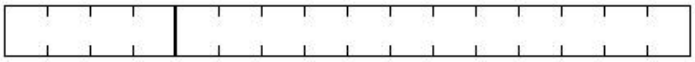
Problem 1 (20 points): Short Answers

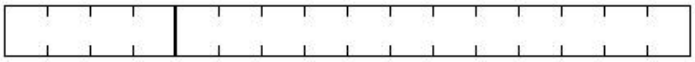
Please answer concisely. If you find yourself writing more than a few words or a simple drawing, your answer is probably wrong.

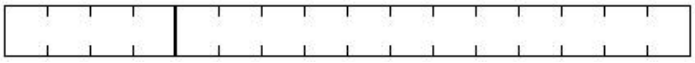
Part C (5 points): Using one or more LC-3 instructions, implement a branch if positive (BRp) to an address outside the range of the branch instruction. Assume that the address to which you want to branch is stored in R5. Write your instruction(s) in binary. *Note: you may not need all the lines provided below.*

Address Instruction

x3000: 

x3001: 

x3002: 

x3003: 

S18-honors:
write assembly code,
not bits --SL

 Name: _____

Problem 1 (40 points): Short Answer

Part A. Rewrite the following LC-3 code replacing the use of the direct and register-offset addressing modes with the indirect address mode. Provide your answer in the box.

```
      LD   R0, LOC
      LDR  R1, R0, #0
      ADD  R1, R1, #1
      STR  R1, R0, #0
      HALT
LOC   .FILL x7654
```

Part D. If we never bothered to check the ready bit when writing to the Display Data Register, what is the likely outcome?

 Name: _____

Part G. Consider the following code snippet to multiply R0 by R1 using repeated addition (assuming both are positive values). The product is kept in R2. How many instructions are executed when this program is run? Provide your answer in terms of the values of R0 and R1.

```
            AND  R2, R2, #0
LOOP        ADD  R2, R2, R1
            ADD  R0, R0, #-1
            BRp  LOOP
```

Part H. Consider two 16-bit 2's complement numbers A and B. What does it indicate if the expression NOT(A + NOT(B)) equals zero? Hint: recall that NOT(B) + 1 = -B.

 Name: _____

Problem 2 (20 points): LC-3 Assembly Programming

In this problem, you will help complete an LC-3 assembly program to remove all spaces in a character string. For example, if you are given a string "`Hello␣␣World␣!`", the program will convert the string to "`HelloWorld!`". Here, "␣" indicates the space character (ASCII 32). The string is terminated by a NULL character (ASCII 0) and is stored in memory at the memory location indicated by the symbol STRING.

The Algorithm works as follows: We will keep two memory addresses to track the string. One is called "Current Read" address, which is stored in R0. The other is called "Current Write" address, which is stored in R1. In the beginning, both R0 and R1 will contain the starting address of the string. R4 will contain the value -32, which we will use in our comparison tests to check for the space character.

At each iteration, we read the string at the "Current Read" location and test for the space character. If the character is a space, we only need to advance the "Current Read". If the character is not a space, we write the character to the "Current Write" location, and advance both the "Current Read" and "Current Write" locations. We then test for the end of string. If the character is a NULL, we are done. If it is not, we start another iteration.

Part A (15 points): Complete the program by filling in the missing information.

```
            .ORIG x3000
            LEA R0, STRING ; R0 contains "Current Read" location
            ADD R1, R0, #0 ; R1 contains "Current Write" location
            ___ R4, SPACE  ; R4 contains -32 (minus ASCII for space)

NEXT        LDR R2, R0, #0       ; R2 contains current character
            ADD R3, R2, R4       ; R3 is a temporary value
            BR__ NOTSPACE
            ADD R0, R0, ___      ; We have a space
            BR NEXT

NOTSPACE    STR ___, R1, ___     ; Write to "Current Write" location
            ADD ___, R0, #1
            ADD R1,  R1, #1
            _____________        ; Test for end of string
            BR__ NEXT

DONE        HALT

SPACE       .FILL #32
STRING      .STRINGZ "ECE 190 !"
            .END
```

 Name: _____

Problem (20 points): LC-3 Assembly

In this problem you will create an LC-3 Assembly Language program to add two 32-bit unsigned numbers. The input values are provided in locations x5000—x5003, and the results are to be placed in locations x5004 and x5005 as indicated below.

```
x5000          Input A[15:0]
x5001          Input A[31:16]
x5002          Input B[15:0]
x5003          Input B[31:16]
x5004          Output S[15:0]
x5005          Output S[31:16]
```

The algorithm for this addition is very straightforward:

```
S[15:0]  = A[15:0] + B[15:0];
S[31:15] = A[31:16] + B[31:16] + Carry[16];
```

The tricky part of the code involves generating Carry[16], the carry out from the addition of the lower 16 bits that is the carry into the addition for the upper 16 bits. Below is a code template:

```
            .ORIG x3000
            ; Load input data

            ; Calculate S[15:0] = A[15:0] + B[15:0]

            ; Calculate S[31:16] = A[31:16] + B[31:16]

            ; Correct S[31:16] = S[31:16] + Carry[16]

            ; Store results to Output locations

            HALT            ; Stop program execution
POINTER     .FILL x5000     ; Pointer to input data
```

Part A (5 points) Write code to **Load** the input data from memory. You may use the contents the location POINTER to help you. (Should be 5 instructions)

 Name: _____

Part B (1 point): Write the code for generating S[15:0] and S[31:16] in the **Calculate** blocks in the template. (Should be 2 instructions)

Part C (4 point): Fill in the truth table for S[15] below as a first step to understanding how to write the code for generating Carry[16]. Carry[16] is the carry used for generating S[16].

Carry[15]	A[15]	B[15]	S[15]	Carry[16]
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Part D (8 points): For this portion, provide the instruction to **Correct** S[31:16] by adding in the Carry[16]. (Hint: rework the solution for Part C. Should be less than 15 lines)

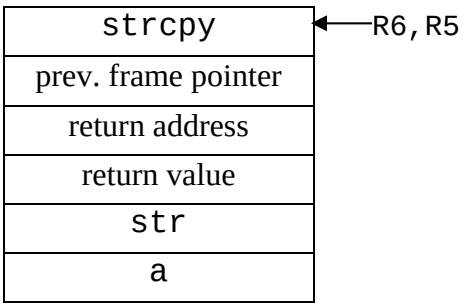
Part E (1 point): **Store** the result into the desired memory locations. (Should be 2 or 3 lines).

Problem 4 (20 points): From C to LC-3 and Back Again

Part A (10 points): translate the C function below to LC-3 assembly instructions. The diagram of the stack frame for the function call has been provided for you.

Translate the `while` and `return` statements from the function body **independently, with no register values shared between sections**. The stack frame management and register save/store has been done for you (not shown in figures).

```
char* find_char
(char* str, char a)
{
   char* strcpy = str;

   while(*strcpy != a){
      strcpy++;
   }
   return strcpy;
}
```



stack frame for `find_char`

; create stack frame and save registers
…
; char* strcpy = str;
; DO NOT WRITE IN THIS BOX

; translation for while loop
;while(*strcpy != a) {
; strcpy++;
; }

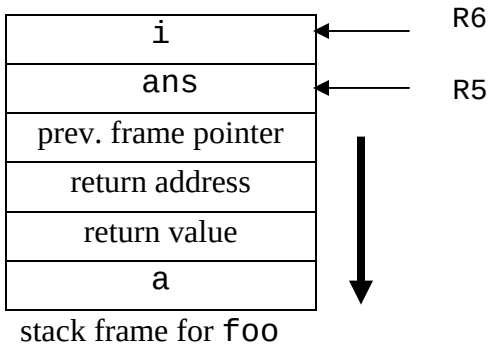
; translation for
; return i;

; restore registers and tear down stack frame
…
; DO NOT WRITE IN THIS BOX

Problem 4, continued:

Given below is part of the LC-3 translation of a C function `foo` and part of the function `foo` itself. Also given is the stack frame (activation record) for `foo`. Remember that memory addresses increase in the direction of the arrow. Answer the questions below.

```
int foo (int a)
{
  int ans = 0;
  int i;
  for (i = a; 0 < i; i--)  {

   /* body of loop written
    * by you in Part B
    */
  }
  return ans;
}
```



stack frame for `foo`

```
            .
            .
            .
          AND R0, R0, #0  ;1
          STR R0, R5, #0  ;2
          LDR R0, R5, #4  ;3
          STR R0, R5, #-1 ;4
LOOP
          LDR R0, R5, #-1 ;5
          BRnz DONE       ;6
          LDR R0, R5, #-1 ;7
          LDR R1, R5, #0  ;8
          ADD R1, R1, R0  ;9
          STR R1, R5, #0  ;10
          LDR R0, R5, #-1 ;11
          ADD R0, R0, #-1 ;12
          STR R0, R5, #-1 ;13
          BRnzp LOOP      ;14

DONE
          LDR R0, R5, #0  ;15
          STR R0, R5, #3  ;16
            .
            .
            .
```

Part B (6 points): Which LC-3 instructions correspond to (give the instruction numbers shown in the comments):

a. The initialization of the `for` loop?

b. The test part of the `for` loop?

c. The update (re-initialization) of the `for` loop?

Part C (4 points): Using the LC-3 translation of `foo`, write the body of the `for` loop here.

 Name: _____

Problem 1 (20 points): Short Answer

Part A (5 points): Suppose an I/O event (e.g., a keystroke) occurs infrequently, and at irregularly distributed times. Would a polled or interrupt-driven approach to processing the event be a better design choice? Give two reasons why the approach that you chose is the better one.

Part B (5 points): A certain C function `bar()` accepts a variable number of non-negative parameters/arguments of type `int`. Explain two ways to design `bar()` so that the callee can determine the actual number of parameters passed.

Problem 1, continued

Part C (5 points): What is wrong with the following function?

```
int* sum_of_int (int x, int y)
{
    int sum = x + y;
    return &sum;
}
```

Part D (5 points): Complete the output from the program fragments below.

```
int x = 0;
int i = 4;
for ( i = 0; 10 > i; i++ ) {
    i++;
    x++;
}
printf ("x: %d\ni: %d\n", x, i);
```

Output:

x: _____

i: _____

```
int x = 0;
int i = 4;
while ( 10 > i )
{
    if ( 3 < x ) { break; }
    x++;
    i++;
}
printf ("x: %d\ni: %d\n", x, i);
```

Output:

x: _____

i: _____

 Name: _____

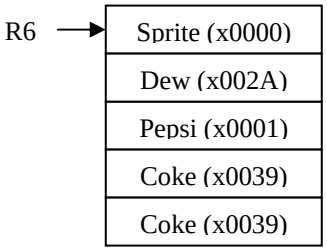
Problem 3 (20 points): Assembly Programming using Subroutines

You are given a vending machine filled with four different types of sodas. The vending machine behaves like a stack: when you buy a drink, it is "popped" off the stack and given to you. Unfortunately, you cannot open the machine, so the only way to remove the sodas is to "pop" them off, and the only way to put sodas back in the machine is to "push" them in. Your job is to inventory the sodas, restore the machine to its initial state, and then print out the total number of each soda.

You are provided with 2 stacks. One stack is a representation of the soda dispenser (*Stack 1*), and the other stack is empty (*Stack 2*). The stack pointer for *Stack 1* is R6, and the stack pointer for *Stack 2* is R5. *Stack 1* (the soda dispenser) has an arbitrary number of sodas in it. Each of the four types of drinks is represented on the stack by a unique value.

Type of Soda	*Encoded Value*
Coke	x0039
Pepsi	x0001
Sprite	x0000
Mountain Dew	x002A

Example of Stack 1



You are provided with the following subroutines:

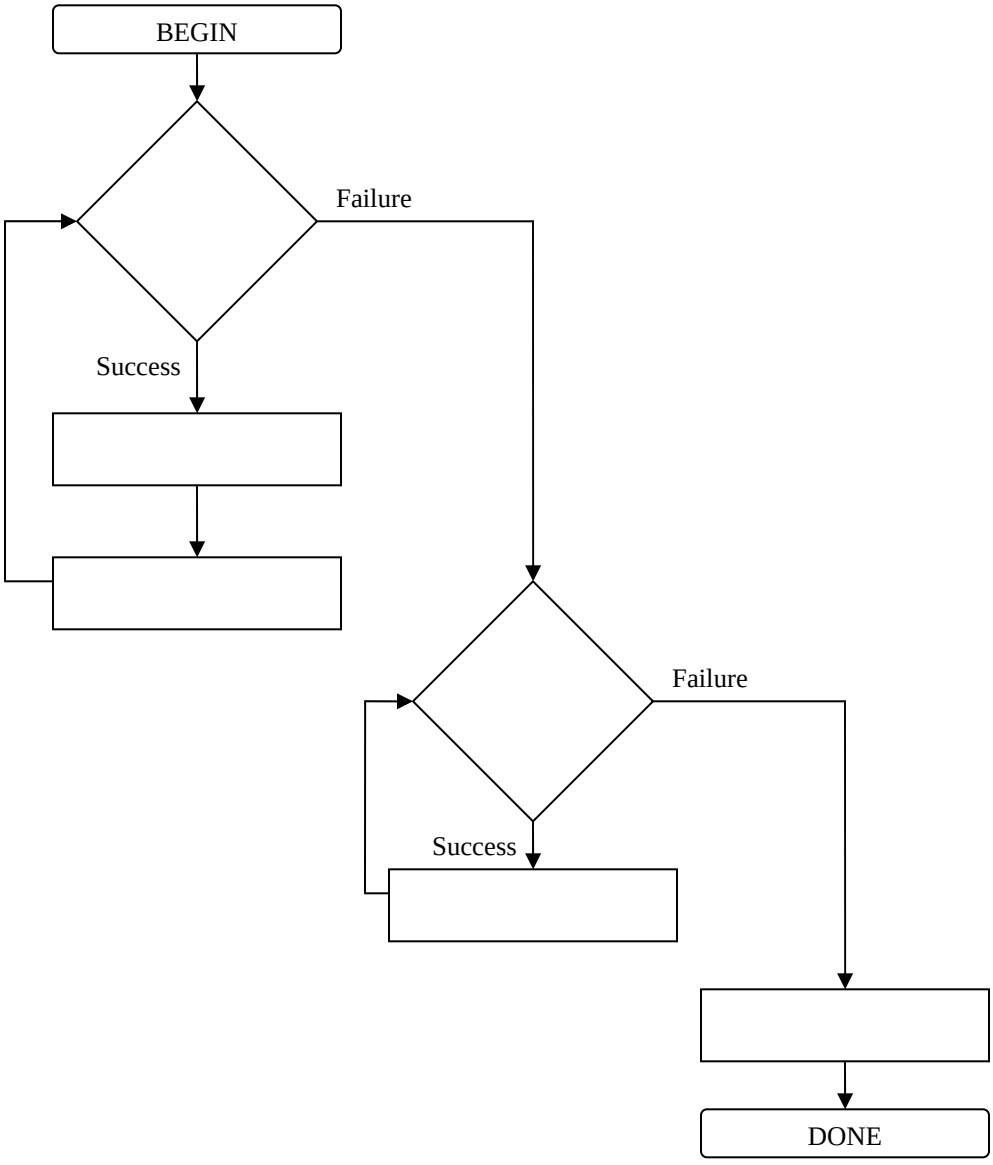
PUSH1 and PUSH2	
Description	Pushes the value in R0 onto *Stack 1* (PUSH1) or *Stack 2* (PUSH2)
Input	R0 (the value to be pushed)
Returns	Success: R0 contains 0
	Failure: R0 contains -1 (the stack is full)
POP1 and POP2	
Description	Pops a value off the stack into R0 (POP1 for *Stack 1*; POP2 for *Stack 2*)
Input	None
Returns	Success: R0 contains popped value
	Failure: R0 contains -1 (the stack is empty)
ADD_BIN	
Description	Adds the drink to a histogram stored at memory location x4000
Input	R0 contains the encoded value of the drink to add
Returns	Nothing
PRINT_HIST	
Description	Prints out the histogram created by ADD_BIN
Input	None
Returns	Nothing

 Name: _____

Problem 3, continued

Part A (6 points): Again, your job is to inventory the sodas in *Stack 1*, restore the machine to its initial state, and then print out the total number of each soda. Complete the systematic decomposition below by writing the name of one of the subroutines provided to you in each box of the flowchart below.

You may assume that neither PUSH1 nor PUSH2 ever fails.



Problem 3, continued

Part B (8 points): Using the subroutines provided, write LC-3 assembly code for the task assigned to you: again, count the number of each type of soda on *Stack 1*, restore *Stack 1* to its initial state, and print out the total number of each type of soda. Note that

- you may assume that neither PUSH1 nor PUSH2 ever fails,
- you may not access the stacks using their stack pointers directly,
- you may assume that registers R0, R1, R2, R3, and R4 are callee saved, and that R5 and R6 are only affected as specified by the subroutine semantics,
- you may make **no other assumptions** about any subroutine's implementation,
- you do not need to include .ORIG, HALT, or .END, and
- if you write more than fifteen or twenty lines, you're doing it wrong.

Problem 3, continued

Part C (6 points): Write the subroutine PRINT_HIST, which prints out the data stored in the histogram (the total number of each type of drink) located at memory locations x4000 to x4003. You may use TRAP x26 to print out decimal numbers; note that the version given to you **also prints a line feed** after the number. *You may use R0 through R3 without saving them for this part; other register values must be preserved.*

Example output for PRINT_HIST:

```
0
4
3
1
```

TRAP x26	
Description	Prints R0 to the screen as a decimal number followed by a linefeed
Input	R0 contains decimal number to be printed
Returns	Nothing

 Name: _____

Problem 4 (20 points): LC-3 and C

Part A (6 points): Explain why a compiler that does not make use of library subroutines requires more LC-3 instructions to implement the statement below on the left than the statement below on the right. The variable `i` is an `int`.

```
i = (i >> 1);
```

```
i = (i << 1);
```

 Name: _____

Problem 5 (20 points): C Programming

```
 0 int foo (int x, int y) {
 1    int z, val = x;
 2
 3    if (0 > x || 0 > y) {
 4       return -1;
 5    }
 6
 7    if (x > y) {
 8       z = x / y;
 9       val = x – y * z;
10    }
11    return val;
12 }
```

Most parts of this problem pertain to the C function shown above. The line numbers are only for reference and are not part of the program.

Part A (3 points): What is the return value of `foo (10,3)` ? _____

Part B (3 points): What is the return value of `foo (452,500)` ? _____

Part C (4 points): Using no multiplication operators, fill in a simple C expression below such that we can replace lines 7 through 10 of the function with the statement below without affecting the results returned from the function.

`val` = _____ ;

Part D (5 points): List the five types of information typically stored in a C function's stack frame (also called an activation record in the textbook)?

Part E (5 points): How many LC-3 memory locations does the stack frame for function `foo()` require? Assume that each `int` occupies one memory location. Justify your answer.

Problem 1 (20 points): Short Answers

Please answer concisely. If your answer requires more than a few words or a simple figure, it is probably wrong.

Part A (5 points): Consider the following C function.

```
int mystery ()
{
    int x = 1;
    int y = 10;

    do {
        y = y + x;
        if (4 < x) {
            break;
        }
        x = x * 2;
    } while (20 > y);
    printf ("%d\n", y); /* A(i)  refers to this line. */
    return x;           /* A(ii) refers to this line. */
}
```

i) What number does the call to `printf` in the `mystery` function output to the display?

ii) What value does the `mystery` function return?

Part B (5 points): Consider the C code snippet below.

```
int a = 10;
int b = 5;
int c = 10;

c = (a++) + (--b) + c;
```

Write the values of the three variables after all of the assignments have completed.

a _____ b _____ c _____

Part D (5 points): Most functions require a specific number of arguments. A few, such as `scanf`, take a variable number of arguments. In the C declaration below, the ellipsis ("...") following the first argument indicates that a variable number of additional arguments can be passed.

```
void variable_args (int num_args, ...);
```

For the `variable_args` function, the `num_args` argument indicates the number of additional arguments. For example, the call `variable_args (2, 4, 5)` indicates that two additional arguments (4 and 5) are being passed.

When a compiler generates assembly code for a call to the `variable_args` function, should the `num_args` argument be pushed first or last (or does it not matter, if all compilers are required make the same choice)? Explain your answer.

Problem 2 (15 points): Assemblers and Assembly Language

This exam you are now taking was written by a program outputting the text to the console and redirecting that to a text file (think of comparisons you made for MP2). The first function we wrote was HEADER, which is shown below. **There are no typographical errors in the code.**

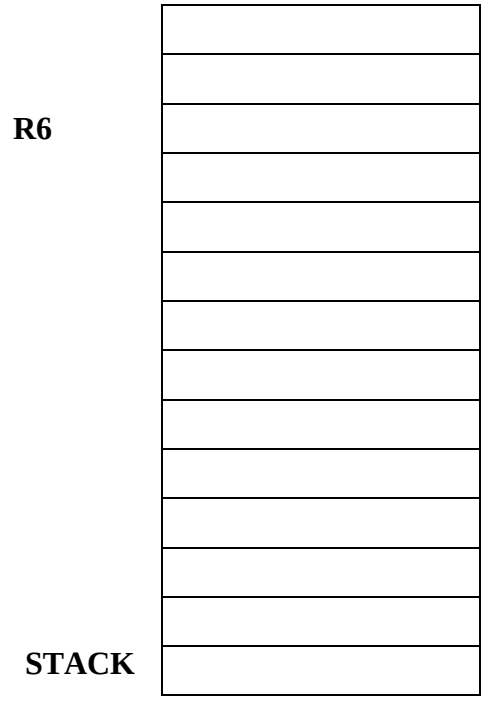
```
          .ORIG x3000
MAIN      LEA R1,TABLE
          JSR HEADER
          LEA R0,NICE
          HALT
NICE      .STRINGZ "Good Luck!"

HEADER    ST R7,TEMP
          LDR R0,R1,#0
          BRz DONE
          ADD R1,R1,#1
          PUTS
          LD R0,LINEFEED
          OUT
          BRnzp HEADER
DONE      LD R7,TEMP
          RET

TEMP      .BLKW #1
LINEFEED  .FILL x0A
TABLE     .FILL FIRST       ; address of first string
          .FILL SECOND      ; address of second string
          .FILL x0000
FIRST     .STRINGZ "ECE 190"
SECOND    .STRINGZ "Midterm 2"
          .END
```

Part B (7 points): Show the output of the program and describe the program's behavior.

Problem 3 (25 points): Stacks and Subroutines

This problem pertains to the LC-3 assembly program shown on the left below.

```
        .ORIG x3000
        LEA R6,STACK
        LEA R1,STRING

        ADD R2,R1,#1

PART1   LDR R0,R1,#0
        BRz PART2
        ADD R6,R6,#-1
        STR R0,R6,#0
        ADD R1,R1,#2
        BRnzp PART1

PART2   LDR R0,R6,#0
        ADD R6,R6,#1
        OUT
        LDR R0,R2,#0
        OUT
        ADD R2,R2,#2
        LDR R0,R2,#-1
        BRnp PART2

        HALT
MESSAGE .STRINGZ "SOOGTALUTARINNC!"
        .BLKW #20
STACK
        .END
```



Part A (7 points): Assuming that an LC-3 processor has executed the program until it first reaches the instruction at PART2, fill in the contents of the stack diagram on the right above with one ASCII character per location. Any location not changed by the program must be left blank. Note the position of the STACK label, and **draw an arrow from R6** in the diagram to the memory location to which it currently points.

Part B (5 points): Write the output of the program.

Part C (6 points): The string provided in the program above has a length of 16 ASCII characters (not counting NUL). For what string lengths does this program do something predictable?

Problem 3, continued:

Part D (5 points): Add necessary instructions before and after the main portion of the program to turn it into an **assembly subroutine** (not a C subroutine—you do NOT need to create a stack frame/activation record).

R1 holds the address of a string of appropriate length when the subroutine is called, and R6 points to a stack with plenty of space left on top.

Your subroutine may not change the value of ANY register (except R7); R0, R1, R2, R3, R4, R5, and R6 **must be returned with their original values** at the end of your subroutine.

*******(2 points): For full credit on this problem, add only instructions in your code (no new directives such as .BLKW, no .FILL, no .STRINGZ, *etc.*).

```
STRINGSUB
; you may add code here









        ADD R2,R1,#1    ; this code was copied from previous page

PART1   LDR R0,R1,#0
        BRz PART2
        ADD R6,R6,#-1
        STR R0,R6,#0
        ADD R1,R1,#2
        BRnzp PART1

PART2   LDR R0,R6,#0
        ADD R6,R6,#1
        OUT
        LDR R0,R2,#0
        OUT
        ADD R2,R2,#2
        LDR R0,R2,#-1
        BRnp PART2

; you may also add code here
```

Problem 4 (20 points): I/O and Systematic Decomposition

While working at a summer internship, your boss informs you that the company has just bought several LC-3 machines. Your company is working on a top-secret defense contract, and you are charged with writing the password entry subroutine. As you might expect, the public version of the LC-3 operating system cannot be used on a secure machine, so **your code must interact directly with the devices**.

The display registers on your machines behave identically to a standard LC-3 platform: DSR[15] indicates that the display is ready to receive a new ASCII character; when the display is ready, writing an ASCII character to DDR[7:0] delivers it to the monitor.

The keyboard registers are similar, but have been extended with a fingerprint scanner. KBSR now returns one of the following bit patterns (only bits 15 and 3 are defined; others may hold any value):

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	
0	x	x	x	x	x	x	x	x	x	x	x	0	x	x	x	no key, no fingerprint
0	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x	impossible
1	x	x	x	x	x	x	x	x	x	x	x	0	x	x	x	keystroke—read KBDR
1	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x	fingerprint match

When a key is ready, the 8-bit ASCII code can be read from KBDR[7:0]. No additional data is available when a fingerprint match is indicated by KBSR, thus KBDR holds no meaningful bits in that case.

For confidentiality reasons, every time the user types a password character, an asterisk ('*') must be echoed to the monitor in place of the character actually typed, while the character typed must be recorded in memory for later processing. After typing their password, a user swipes their finger over the scanner and, if authorized to use the machine, produces a fingerprint match response from KBSR.

Part A (10 points): Complete the systematic decomposition of the password entry routine on the next page by drawing missing arrows and filling in the boxes with brief labels in English and the symbolic names listed below.

KBSR KBDR DSR DDR

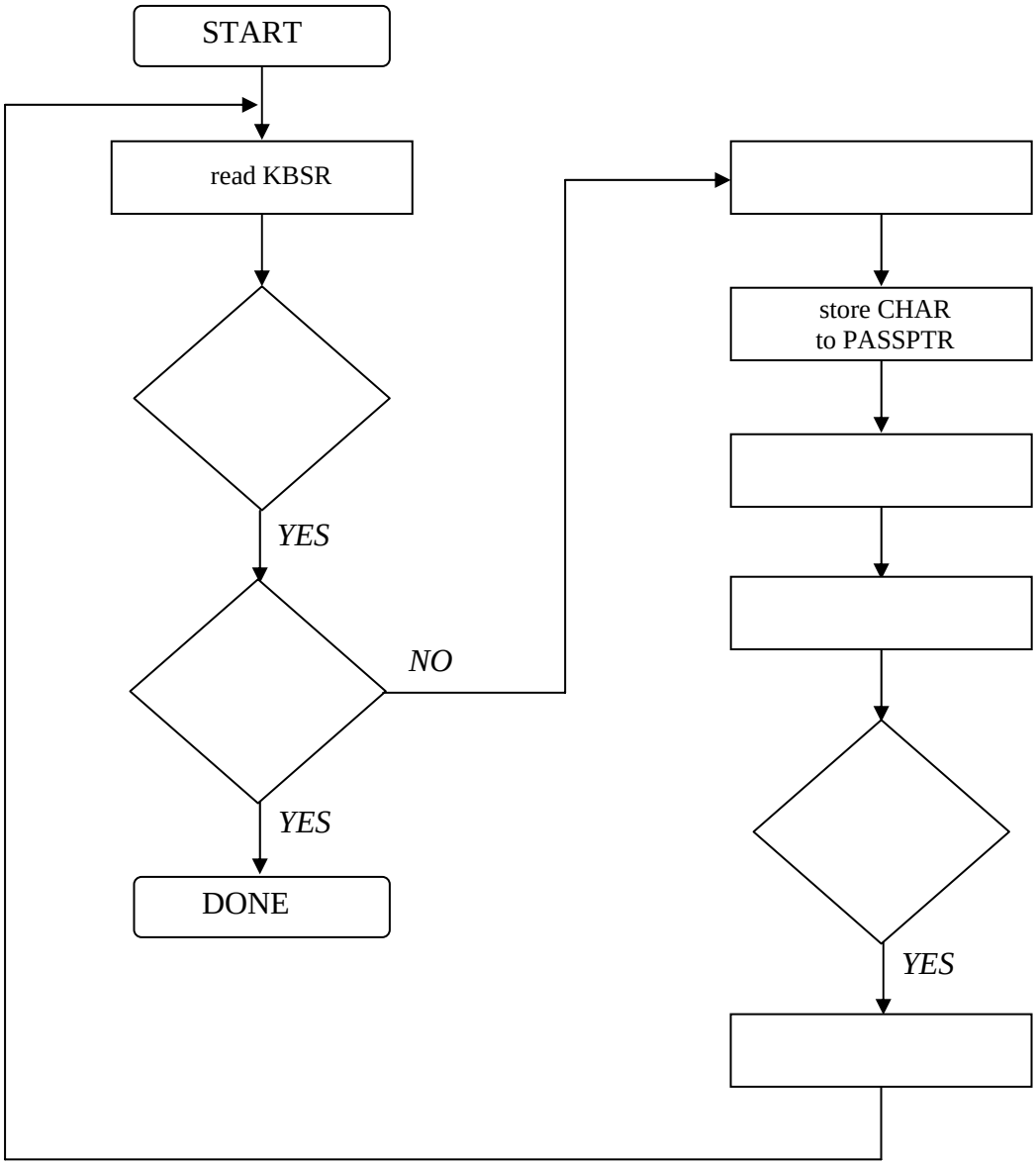
READY — the ready condition (as read from either KBSR or DSR)
FINGER — the fingerprint match flag (as read from KBSR)

CHAR — a character read from the keyboard
PASSPTR — the address of the first location in a block of memory into which you must store the password characters
ASTER — a register holding the ASCII character '*'

Note that labels in the form of LC-3 instructions earn no credit.

Problem 4, continued:

Part A answer diagram



Problem 4, continued:

Part B (10 points): Now write an LC-3 assembly subroutine to implement your password entry routine.

Some specifics for your subroutine:

- Device register addresses constants are provided for your use.
- R2 is an input to your subroutine and holds the address of the first location in the block of memory into which you must store the password characters (PASSPTR in **Part A**).
- R3 is loaded for you with an asterisk (ASTER in **Part A**).
- You may use any register that you like for the character read from KBDR (CHAR in **Part A**), reading the device registers, and so forth.
- You need not preserve any register values for this subroutine (all registers are caller-saved).

- **You may NOT add other constants or space for storage.**

```
PASS_ENTER
        LD R3,ASTERISK
        ; your code goes here
```

```
        RET
ASTERISK .FILL x002A
KBSR     .FILL xFE00
KBDR     .FILL xFE02
DSR      .FILL xFE04
DDR      .FILL xFE06
```

Problem 5 (20 points): C to LC-3

Consider the following C function.

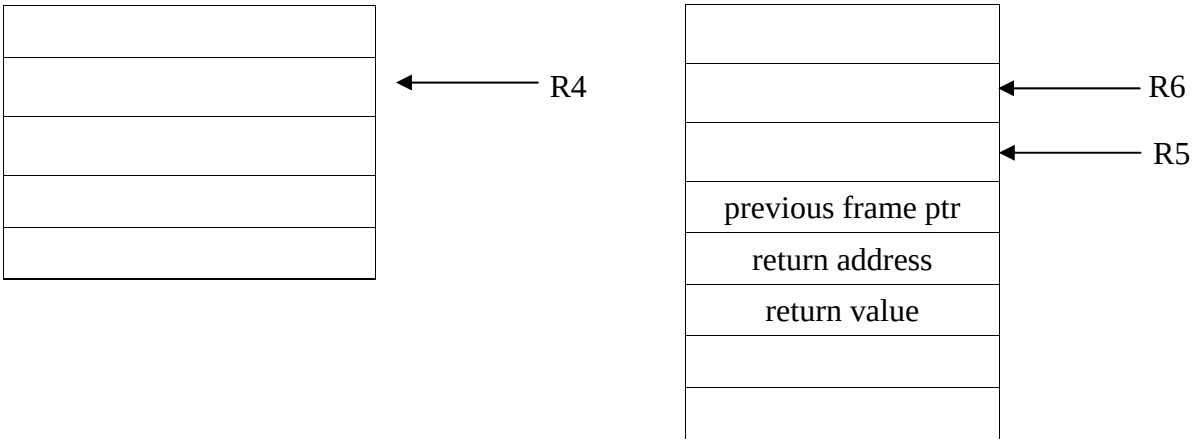
```
int a;

int a_function (int arg)
{
    int x;
    int y;

    x = arg - 1;
    y = (x << 3);
    if (10 > y) {
        y = a + 20;
    }
    return y;
}
```

Remember that in the LC-3 calling convention, C programs use R4 to point to the beginning (base) address of the global data section, R5 to point to the stack frame/activation record, and R6 to point to the top of the stack.

Part A (2 points): For the code above, fill the blank spaces in the partial symbol table shown below.

Identifier	Type	Offset	Scope
a	int	2	global
x	int	0	a_function
y	int	-1	a_function
arg	int		

Part B (6 points): Based on the symbol table, mark the locations of a, x, y and arg in the global data section diagram on the left and the stack frame diagram on the right below. Do not mark other boxes.



Part C (12 points): Fill in these five boxes with LC-3 assembly code for each of the statements listed at the top of the column. Write each section independently of the others: your code may not rely on registers loaded in a previous section (other than R4 and R5), and must update variables appropriately rather than leaving results in registers. **You may overwrite the values in R0 and R1 in your code for this problem.**

`x = arg - 1;`	`y = (x << 3);`

`if (10 > y) – branch if FALSE to NOPE`	`y = a + 20;`

`return y;`
`NOPE ; branch target for if test failure (above left)`

 Name: _____

Problem 2 (25 points): Analyzing C programs

Part A (5 points): Provide the output generated by the following program

```
int main()
{
  int ii;
  int jj;

  for (ii = 0; ii < 5; ii++)
    {
      for (jj = ii; jj < 5; jj++)
        printf("*");
      printf("\n");
    }
}
```

Output:

Line 1:

Line 2:

Line 3:

Line 4:

Line 5:

Line 6:

Line 7:

Line 8:

Line 9:

 Name: _____

Part B (5 points): Provide the output generated by the following program.

```
int main()
{
  int ii;
  int jj;

  for (ii = 1; ii != 64; ii = ii * 2)
    {
      jj = 1;
      while ( ii > jj )
        {
          if (!(ii % jj))
            printf("*");
          jj = jj << 1;
        }
      printf("\n");
    }
}
```

Output:

Line 1:

Line 2:

Line 3:

Line 4:

Line 5:

Line 6:

Line 7:

Line 8:

Line 9:

 Name: _____

Part C (5 points): Provide the output generated by the following program.

```
int main()
{
  char cc;
  char dd;
  char ee;

  cc = '2';
  dd = '3';

  ee = cc + dd;

  printf("%c\n", ee);
}
```

Output:

Part D (5 points): Provide the output generated by the following program.

```
int main()
{
  int ii;
  int jj;

  /* The following declaration includes an initializer that
     initalizes the array list to the indicated values.  After
     the initialization, list[0] = 8, list[1] = 2, etc.. */
  int list[10] = { 8, 2, 3, 1, 7, 4, 0, 6, 9, 5 };
  int pre[10];

  for (ii = 0; ii < 10; ii++)
    {
      pre[ii] = 0;
      for (jj = 0; jj < ii; jj++)
        pre[ii] = pre[ii] + list[jj];
    }

  for (ii = 0; ii < 10; ii++)
    printf("%d ", pre[ii]);
  printf("\n");
}
```

Output:

Problem 3 (45 points): Understanding functions in C

Shown below is a simple C program. In Parts A and B of this problem, you will fill in values for the run-time stack as the code below executes. Parts C through F deal with the LC-3 code generated for functions **`main`** and **`foo`**. For this problem, you may assume that the LC-3 compiler treats **`int`**s as 16-bit data types. Also, you cannot rely on any prewritten subroutines – you must provide all required code.

```
int main()
{
      int x = 4;
      int y = 9;
      int z = 16;

      x = foo(y, z);
}

int foo(int a, int b)
{
      int x, y;

      y = 20;
      x = bar(a, y);

      return 1;
}

int bar(int a, int b)
{
      int z = 7;

      return 10;
}
```

 Name: _____

Part A (20 points)**:** The table below contains a snapshot of the LC-3 run-time stack during execution of the given C code. Provide the values of the run-time stack just after the last instruction in the function **`bar`** executes. Provide exact values when possible (use UNKNOWN if you don't know an exact value). Hint: You might need to use information from Parts C through F to determine some addresses. To start you off, part of the activation record for **`main`** has been provided. In the column labeled "Field", indicate the purpose of the stack entry by denoting one of the following:

(A) Local variable **(B)** Argument **(C)** Dynamic link **(D)** Return value **(E)** Return address

Address	Memory Values	Field
xFD05		
xFD06		
xFD07		
xFD08		
xFD09		
xFD0A		
xFD0B		
xFD0C		
xFD0D		
xFD0E		
xFD0F		
xFD10		
xFD11		
xFD12		
xFD13		
xFD14		
xFD15		
xFD16	16	(A) Local variable z in main
xFD17	9	(A) Local variable y in main
xFD18	4	(A) Local variable x in main

Part B (5 points): What are the values of R5, R6, and R7 at the same point in execution as Part A.

R5 =

R6 =

R7 =

 Name: _____

Shown below are the LC-3 subroutines for the C functions **`main`**, **`foo`** and **`bar`**. Some of the assembly code has been omitted. For parts C through F you will write the assembly code needed to set up and tear down the activation record for **`foo`**. In your code, use only R0 as a temporary register. No need to worry about stack overflow or underflow.

Part C (5 points): Write the LC-3 code in **`main`** that corresponds to the call to **`foo`** in the table below.

Part D (5 points): Write a sequence of assembly instruction to POP information off the run-time stack after **`foo`** has returned in the table below.

```
        ; function main
x3000   :
        ; Initial code for main
        :
        ; Provide Code for Part C here









x3020   JSR FOO  ; Jumps to x3050
        ; Code for Part D starts here









        :
        ; Ending code for main
        :
        RET
```

 Name: _____

Part E (5 points): In the table below, write the sequence of LC-3 instructions that are executed upon entry into the function **`foo`** that prepare **`foo`**'s activation record.

Part F (5 points): In the table below, write the LC-3 code that corresponds to the **`return`** statement in **`foo`**.

```
        ; Function foo
x3050   ; Provide code for part E here
        :
        ; Code for call to BAR
        :
x3060   JSR BAR ; Jump to x3071
        :
        ; Code for return from BAR
        :
        ; Provide code for Part F here
x3070   RET
        ; Function bar
X3071
```

Problem 1, continued:

Part D (5 points): Assume that an LC-3 processor's R4 (register #4) holds the value 0. Without making any other assumptions, what is the minimum number of LC-3 instructions necessary to change R4's value to 16. Explain your answer.

Problem 5 (20 points): LC-3 Instructions

Part A (3 points): Given an ST (store) instruction at address x5000, what is the smallest memory address that the store can change? What is the largest?

Part B (4 points): Consider the following code snippet (RTL on the right):

```
ADD   R2, R1, R0   ; R2 ← R1 + R0
ADD   R2, R2, #1   ; R2 ← R2 + 1
BRz   #10          ; if Z: PC ← PC + 10
```

If the branch is taken (*i.e.*, if the PC changes due to the branch), what do you know about the relationship between R0 and R1? (*An equation will not earn full credit.*)

Problem 1 (20 points): Short Answers

Please answer concisely. If you find yourself writing more than a few words or a simple drawing, your answer is probably wrong.

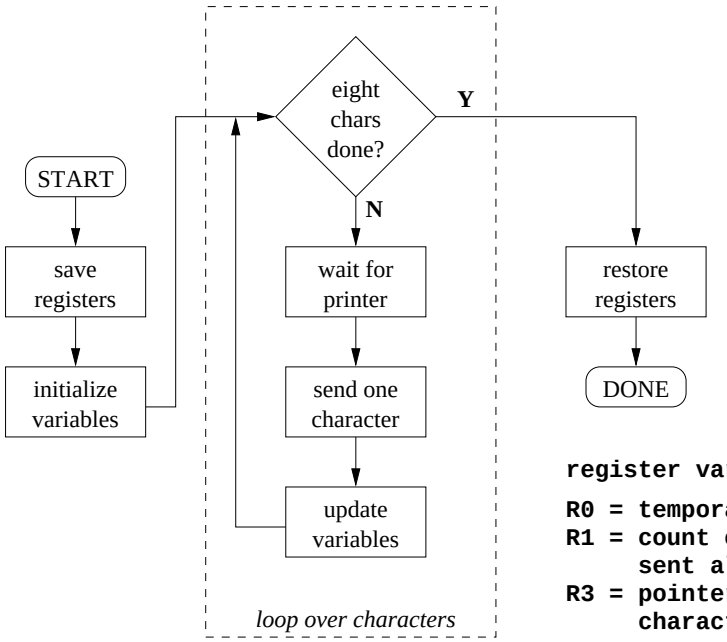
Part A (5 points): Consider the following C function:

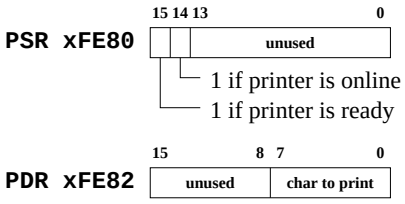
```
void /* returns nothing */
func (int x)
{
    switch ((5 < x) - (3 > x)) {
        case -1:
            printf ("Too cold\n");
            break;
        case 1:
            printf ("Too hot\n");
            break;
        case 0:
            printf ("Just right\n");
            break;
        default:
            printf ("Weird weather!\n");
            break;
    }
}
```

Fill in the blanks below to re-implement the function using `if` statements.

```
void /* returns nothing */
func (int x)
{
    if (_____) {
        printf ("Too cold\n");

    } else if (_____) {
        printf ("Too hot\n");

    } else if (_____) {
        printf ("Just right\n");
    } else {
        printf ("Weird weather!\n");
    }
}
```

Problem 2 (20 points): Systematic Decomposition to LC-3 Assembly






```
register variables for trap routine
R0 = temporary values
R1 = count of characters
     sent already
R3 = pointer to next
     character to send
```

Prof. Lumetta needs your help: a new printer device has been added to the LC-3, but he has not been able to write one of the trap routines, and the next ECE190 assignment requires that trap routine! The trap routine in question sends a sequence of eight characters stored in memory starting at R3 (an input value) to the printer. Before sending each character to the printer, the trap routine must wait until both the online and ready bits of the PSR are equal to 1. The character can then be written to PDR.

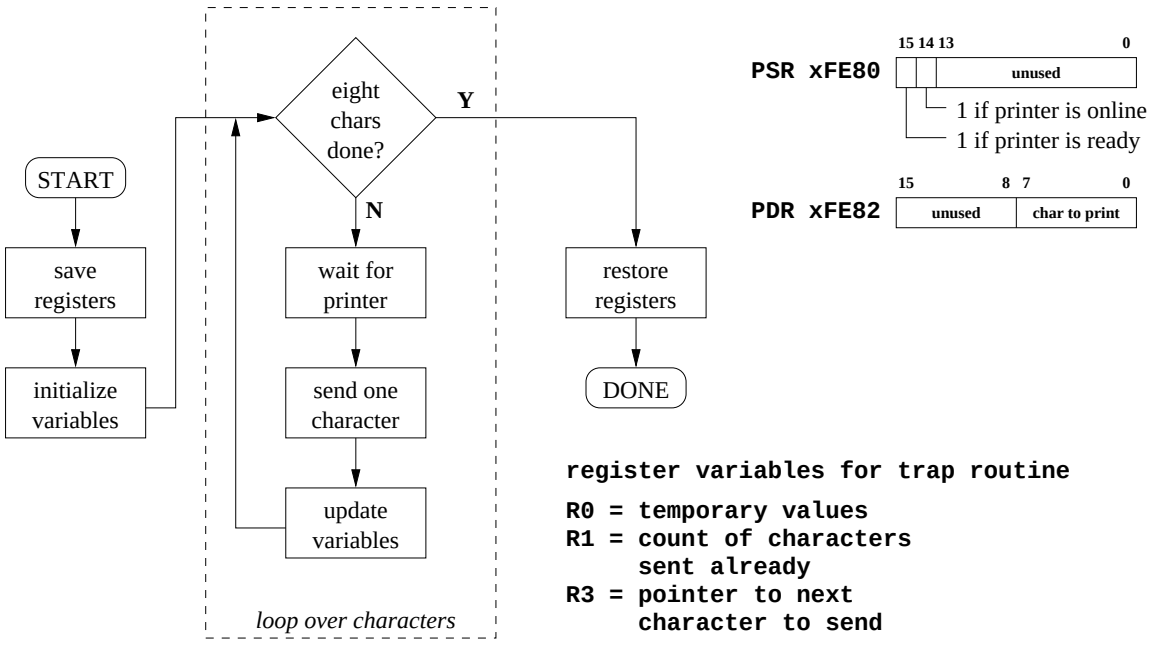
The figure above shows three things: on the left, a partial systematic decomposition for the trap routine (partial because it requires more than one LC-3 instruction for each box); in the upper right, the addresses and pictures of the new Printer Status Register (PSR) and Printer Data Register (PDR); in the lower right, the mapping from registers to data values that you'll need to use in the trap routine.

Part A (5 points): First protect the registers. The trap should preserve **all** register values. Fill in the code and allocate storage as necessary below to accomplish this goal. Two data values have been provided for Parts B and C.

```
; save registers (FILL IN)




; initialize variables and loop over characters (Part C)
; restore registers (FILL IN)




RET ; DONE
; data needed for trap routine (FILL IN)




TRAP_PSR .FILL xFE80
TRAP_PDR .FILL xFE82
```

Problem 2, continued:

Part B (5 points): The next step is to decompose the "wait for printer" box to the level of individual LC-3 instructions. Before sending a character to the printer, the trap routine must wait until both the online and ready bits of the PSR are equal to 1. Draw your answer as a flow chart with RTL or assembly inside each statement or test. For example, you might label a test with "BR" and write N, Z, and P on the appropriate output arcs. Use the register mapping shown in the figure (replicated below). Use data values from **Part A** (you should not need any others).




```
register variables for trap routine

R0 = temporary values
R1 = count of characters
     sent already
R3 = pointer to next
     character to send
```

Problem 2, continued:

Part C (8 points): You are now ready to write the main body of the code. Do so below. Remember that R3 initially points to the first of the eight characters to be sent, and that the others are in consecutive memory locations.

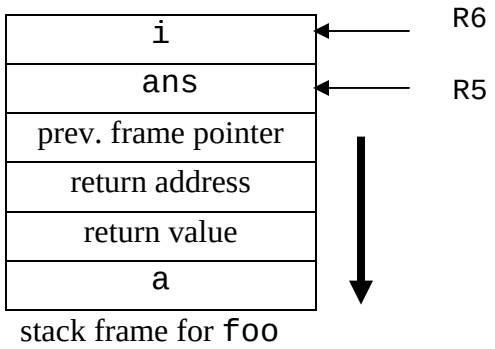
```
; save registers (Part A; NO NEED TO REWRITE)
; initialize variables (FILL IN; SEE REGISTER MAP FOR CONTENTS)




; all characters done?  (FILL IN)




; wait for printer (FILL IN from Part B)









; send one character (FILL IN)




; update variables (FILL IN)




; restore registers, DONE, and data (Part A; NO NEED TO REWRITE)
```

Part D*** (2 points): The printer has a button that turns it online/offline under human control. Using the protocol described, the printer **must** buffer one character even if the character is sent to PDR when the printer is offline. Explain why this buffering is necessary for correct behavior even though your code checks for the online bit before writing to PDR.

Problem 4, continued:

Given below is part of the LC-3 translation of a C function `foo` and part of the function `foo` itself. Also given is the stack frame (activation record) for `foo`. Remember that memory addresses increase in the direction of the arrow. Answer the questions below.

```
int foo (int a)
{
  int ans = 0;
  int i;
  for (i = a; 0 < i; i--)  {

   /* body of loop written
    * by you in Part B
    */
  }
  return ans;
}
```



stack frame for `foo`

```
            .
            .
            .
          AND R0, R0, #0  ;1
          STR R0, R5, #0  ;2
          LDR R0, R5, #4  ;3
          STR R0, R5, #-1 ;4
LOOP
          LDR R0, R5, #-1 ;5
          BRnz DONE       ;6
          LDR R0, R5, #-1 ;7
          LDR R1, R5, #0  ;8
          ADD R1, R1, R0  ;9
          STR R1, R5, #0  ;10
          LDR R0, R5, #-1 ;11
          ADD R0, R0, #-1 ;12
          STR R0, R5, #-1 ;13
          BRnzp LOOP      ;14

DONE
          LDR R0, R5, #0  ;15
          STR R0, R5, #3  ;16
            .
            .
            .
```

Part B (6 points): Which LC-3 instructions correspond to (give the instruction numbers shown in the comments):

a. The initialization of the `for` loop?

b. The test part of the `for` loop?

c. The update (re-initialization) of the `for` loop?

Part C (4 points): Using the LC-3 translation of `foo`, write the body of the `for` loop here.

Problem 5 (20 points): C and Stack Frames

This question focuses on the program below, and particularly on the stack frames (also called activation records) that are used by each function in the program.

```
#include <stdio.h>

/* function declarations */
int bar (int a, int b);
int foo (int* p);

int bar (int a, int b)
{
    int x = a + b;

    if (0 < a) {
        printf ("%d\n", a * b);
    }
    return x;
}

int foo (int* p)
{
    *p = bar (-4, 11);
    return 6;
}

int main ()
{
    int x = 0;
    int y;

    y = foo (&x);
    bar (x, y);
    return 0;
}
```

Part A (3 points): When someone runs the program, what is the order of subroutine calls for the program, starting from `main`? In other words, what is the sequence of JSR target over the whole program execution? Give a comma-separated list, including only the `main`, `foo`, and `bar` functions.

`main`,

Part B (3 points): What, if anything, is printed by the program?

Problem 5, continued:

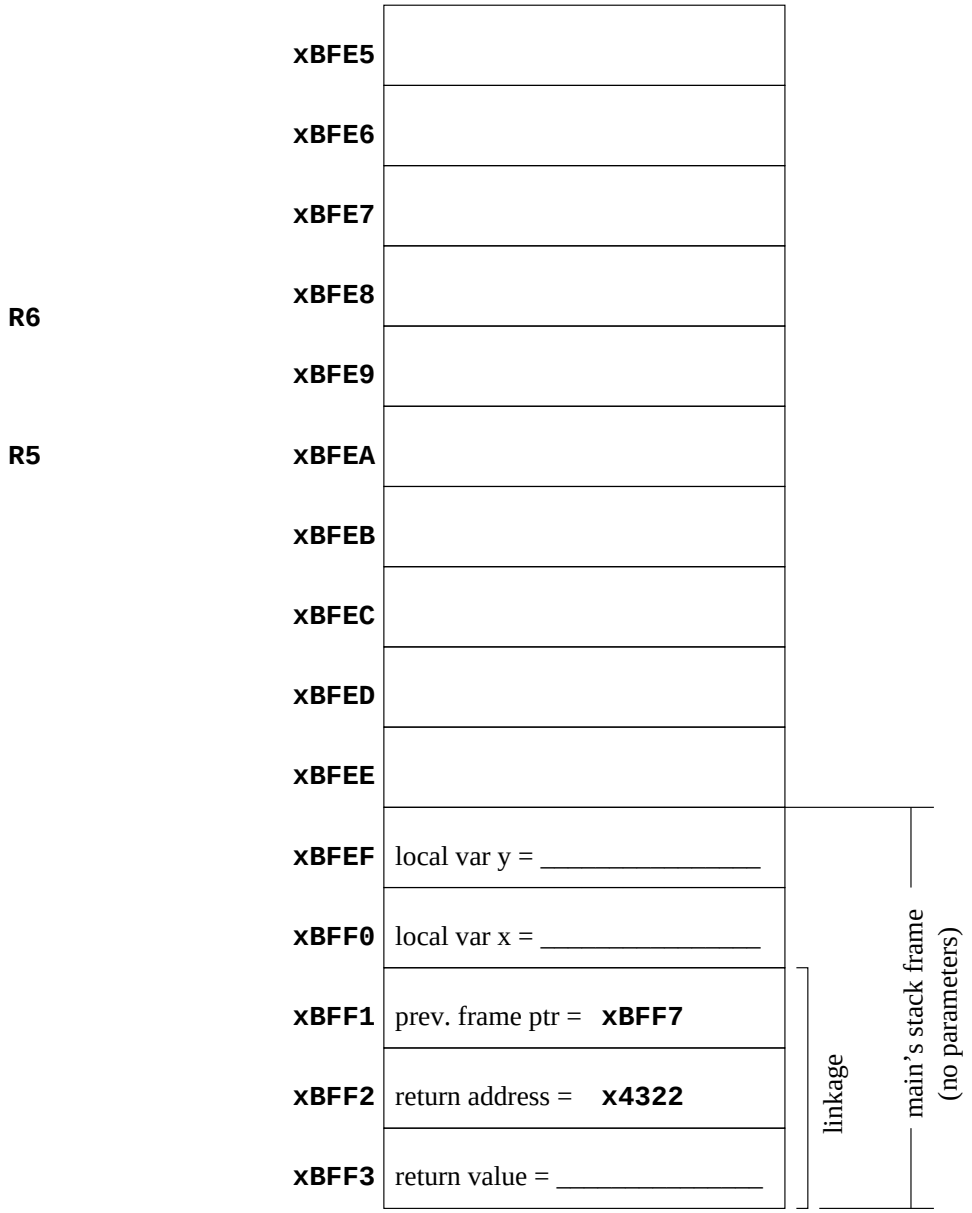
Part C (14 points): The stack frame for the `main` function is shown below. During execution of `main`, the stack pointer `R6=xBFEF`, and the frame pointer `R5=xBFF0`.

Use the figure to draw the stack just after completion of the `return` statement in the `bar` function **when it is called from** `main`, *i.e.*, just before `bar`'s stack frame is torn down and the subroutine returns to `main`.

Draw arrows to indicate the values of `R6` and `R5` at the point of program execution just described. For each memory location included in the stack (*i.e.*, between the stack pointer and the bottom of the figure), label the location with the type of information **and** the value stored there. If a memory location's value **cannot** be known, put a question mark by the description, *e.g.*, "x=?".

Do not mark or label any locations above the stack pointer, even if you know the values in those locations!

The address of the `JSR bar` instruction in `main` is `x3040`.



Problem 1 (18 points): Short Answers

Please answer concisely. If your answer requires more than a few words or a simple figure, it is probably wrong. Assume in all cases that necessary header files have been included.

Part B (4 points): The following line appears in an LC-3 assembly program:

```
.FILL x41
```

What could this fill value represent (circle **one answer**)?

a) the ASCII character for 'A'

b) the unsigned integer 65

c) the signed integer 65

d) all of the above (that is, it could be any of (a) through (c))

e) none of the above, this is a HEX number

Problem 6 (20 points): LC-3 to C

One of your TAs got excited about your MP5 and decided to extend it to support `while` loops in C. The questions on the next page refer to the code below, which was generated for a `while` loop using this extended version. The variable `num` is local to `main`'s stack frame (R5) with offset 0. `num` is initialized (before the code shown) to the value 0. Recall that R6 is the stack pointer.

```
LBL2                    ; Piece 1
        LDR R0,R5,#0
        ADD R6,R6,#-1
        STR R0,R6,#0
;--------------------------------
        LD R0,LBL3      ; Piece 2
        ADD R6,R6,#-1
        STR R0,R6,#0
        BRnzp LBL4
LBL3
        .FILL #10
LBL4
;-------------------------------
        LDR R1,R6,#0    ; Piece 3
        ADD R6,R6,#1
        LDR R0,R6,#0
        ADD R6,R6,#1
        AND R2,R2,#0
        NOT R1,R1
        ADD R1,R1,#1
        ADD R0,R0,R1
        BRzp LBL5
        ADD R2,R2,#1
LBL5
        ADD R6,R6,#-1
        STR R2,R6,#0
;-------------------------------
        LDR R0,R6,#0    ; Piece 4
        ADD R6,R6,#1
        ADD R0,R0,#0
        BRnp LBL7
        LD R3,LBL6
        JMP R3
LBL6
        .FILL LBL1
LBL7
;-------------------------------
        ADD R0,R5,#0    ; Piece 5
        LDR R1,R0,#0
        ADD R2,R1,#1
        STR R2,R0,#0
        ADD R6,R6,#-1
        STR R1,R6,#0
        ADD R6,R6,#1
;--------------------------------
        LD R3,LBL8      ; Piece 6
        JMP R3
LBL8
        .FILL LBL2
LBL1
```

WARNING: I did not solve this problem, so I'm not sure that it corresponds to C that you have learned yet. --SL

Problem 6, continued:

Part A (2 points)**:** What does the code section labeled "Piece 1" do?

Part B (3 points)**:** What does the code section labeled "Piece 2" do?

Part C (6 points)**:** Write **a single C expression** that corresponds to the code sections 1 through 3.

Part D (4 points)**:** Write **a single C statement** that corresponds to the code section labeled "Piece 5."

Part E (5 points)**:** Write the C code corresponding to the LC-3 assembly code shown on the previous page (using a few lines).

WARNING: I did not solve this problem, so I'm not sure that it corresponds to C that you have learned yet.
--SL

Problem 1 (25 points): C Operators and Loops

Read the following C function.

```
void func (int num)
{
    int i;
    while (1 < num) {
        for (i = 2; num > i; i++) {
            if (num == i * (num / i)) {
                break;
            }
        }
        printf ("%d,", i);
        num = num / i;
    }
}
```

Circle EXACTLY ONE ANSWER for each question.

(5 points): What is printed by the function call `func (-42)`?

A) "4,2," B) `func` has a syntax error C) "-1,2,3,7," D) nothing E) "2,4,6,8,10,"

(5 points): What is printed by the function call `func (42)`?

A) `func` has a syntax error B) "4,2," C) "2,3,7," D) "2,4,6,8,10," E) "42,"

(5 points): What is printed by the function call `func (200)`?

A) "2,4,6,8,10,12," B) nothing C) `func` has a syntax error D) "2,0,0," E) "2,2,2,5,5,"

(5 points) One of the loops below is slightly different from the rest. Circle EXACTLY ONE ANSWER to indicate which loop is DIFFERENT.

A)
```
int i;
for (i = 0; 12 > i; i++) {
    // loop body
}
```

B)
```
int i = 0;
do {
    // loop body
} while (12 > ++i);
```

C)
```
int i = 0;
while (12 > i++) {
    // loop body
}
```

D)
```
int i = -1;
do {
    i++;
    // loop body
} while (11 != i);
```

E)
```
int i = 0;
while (12 > i) {
    // loop body
    i++;
}
```

Problem 1 (25 points): C Operators and Loops

Read the following C function.

```
void func (int start, int count)
{
    while (0 < count--) {
        if (0 == (start % 2)) {
            start = start / 2;
        } else {
            start = 3 * start + 1;
        }
        printf ("%d,", start);
    }
}
```

Circle EXACTLY ONE ANSWER for each question.

1. (5 points) What is printed by the function call `func (42, 1)`?

A) "21," B) `func` has a syntax error C) "21,64," D) nothing E) "127,"

2. (5 points) What is printed by the function call `func (19, 3)`?

A) `func` has a syntax error B) "," C) "9,4,13," D) "58,29,88," E) "58,29,88,44,"

3. (5 points) What is printed by the function call `func (200, 0)`?

A) "200," B) "4,2,1," C) `func` has a syntax error D) "100," E) nothing

4. (5 points) For one of the loops below, the behavior within the loop body is slightly different from the other loops. Circle EXACTLY ONE ANSWER to indicate which loop is DIFFERENT.

A)
```
int i;
for (i = 0; 12 > i; i++) {
    // loop body
}
```

B)
```
int i = 0;
do {
    // loop body
} while (i++ < 12);
```

C)
```
int i = -1;
while (12 > i++) {
    // loop body
}
```

D)
```
int i = -1;
do {
    ++i;
    // loop body
} while (12 != i);
```

E)
```
int i = 0;
while (i <= 12) {
    // loop body
    ++i;
}
```

Problem 1, continued:

Read the program below.

```
#define PI 3.1415926

void main()
{
    int y    = 3.14;
    int x    = 8;
    int temp = 90;
    int humi = 80;

    if (y != PI) {
        printf ("y is not PI.\n");               // FIRST printf
    }

    if (!(x = 9)) {
        printf ("x is not 9.\n");                // SECOND printf
    }

    if ((temp >= 80) && (humi >= 50)) {
        printf ("Wow, it's hot!\n");             // THIRD printf
    }

    if ((temp < 60) || (temp > 80)) {
        printf ("The room is uncomfortable.\n"); // FOURTH printf
    }
}
```

(5 points) The programmer who wrote the code above expected all four `printf` statements to execute. Circle EXACTLY ONE ANSWER to indicate which `printf` statement is not executed.

A) `FIRST` B) `SECOND` C) `THIRD` D) `FOURTH` E) All four print.

Problem 1, continued:

Read the program below.

```
#define PI 3.1415926

void main()
{
    int y    = 0x880;
    int x    = 8;
    int mask = 0x888;
    int val  = 0xFFC;
    int pie  = PI;

    if (x + y == mask) {
        printf ("Sum checks out.\n");            // FIRST printf
    }

    if ((mask >> 8) == x) {
        printf ("x is largest hex of mask.\n");  // SECOND printf
    }

    if ((val == (mask | val)) && (mask == (val & mask))) {
        printf ("Mask and val are equal!\n");    // THIRD printf
    }

    if (pie == PI) {
        printf ("PI has not changed.\n");        // FOURTH printf
    }
}
```

5. (5 points) The programmer who wrote the code above expected all four `printf` statements to execute. Circle EXACTLY ONE ANSWER to indicate which `printf` statement is NOT executed.

A) All four print. B) `FIRST` C) `SECOND` D) `THIRD` E) `FOURTH`

Problem 1, continued:

Read the program below.

```
#define PI 3.1415926

void main()
{
    int    y    = 0x410;
    int    x    = 4;
    int    mask = 0x414;
    int    val  = 0xFFC;
    double pie  = PI;

    if (x + y == mask) {
        printf ("Sum checks out.\n");            // FIRST printf
    }

    if ((mask >> 2) == x) {
        printf ("x is largest hex of mask.\n");  // SECOND printf
    }

    if ((val == (mask | val)) && (mask == (val & mask))) {
        printf ("Mask and val are equal!\n");    // THIRD printf
    }

    if (pie == PI) {
        printf ("PI has not changed.\n");        // FOURTH printf
    }
}
```

5. (5 points) The programmer who wrote the code above expected all four `printf` statements to execute. Circle EXACTLY ONE ANSWER to indicate which `printf` statement is NOT executed.

A) All four print. B) `FOURTH` C) `THIRD` D) `SECOND` E) `FIRST`

Problem 2 (25 points): Stack Frames and Assembly Code

(5 points) A call is made to a function with the following function signature:

```
int zap (int b, int a, int c);
```

Circle EXACTLY ONE ANSWER below to indicate which stack frame might represent that of the function `zap` at some point during its execution.

A)

R6 →	x4565	a local variable	777
R5 →	x4566	another local variable	100
	x4567	previous frame pointer	x457F
	x4568	return address	x3222
	x4569	return value	(bits)
	x456A	a	92
	x456B	b	15
	x456C	c	-17

B)

R6 →	x5090	a local variable	222
R5 →	x5091	another local variable	199
	x5092	previous frame pointer	x4FF0
	x5093	return address	x3127
	x5094	return value	77
	x5095	b	41
	x5096	a	-8
	x5097	c	99

C)

R5,R6 →	x4566	a local variable	100
	x4567	previous frame pointer	x4570
	x4568	return address	x3117
	x4569	return value	93
	x456A	b	42
	x456B	a	-5
	x456C	c	7

D)

R6 →	x5CCC	a local variable	101
	x5CCD	previous frame pointer	x5D03
	x5CCE	return address	x3099
	x5CCF	return value	(bits)
	x5CD0	a	1000
	x5CD1	b	2000
R5 →	x5CD2	c	3000

E)

R5,R6 →	x4568	a local variable	999
	x4569	previous frame pointer	x4601
	x456A	return address	x3333
	x456B	return value	(bits)
	x456C	c	17
	x456D	a	-100
	x456E	b	-1000

Problem 2 (25 points): Stack Frames and Assembly Code

1. (5 points) A call is made to a function with the following function signature:

```
int zap (int n, int g, int p);
```

Circle EXACTLY ONE ANSWER below to indicate which stack frame might represent that of the function `zap` at some point during its execution.

A)

R5,R6 →	x5090	a local variable	222
	x5091	another local variable	xECE
	x5092	previous frame pointer	x50B0
	x5093	return address	x3127
	x5094	return value	77
	x5095	g	41
	x5096	n	-8
	x5097	p	99

B)

R6 →	x4568	a local variable	999
	x4569	previous frame pointer	x4580
	x456A	return address	x3333
	x456B	return value	(bits)
R5 →	x456C	p	17
	x456D	g	-100
	x456E	n	-1000

C)

R5,R6 →	x4566	a local variable	100
	x4567	previous frame pointer	x4570
	x4568	return address	x3117
	x4569	return value	93
	x456A	n	42
	x456B	g	-5
	x456C	p	7

D)

R6 →	x4565	a local variable	777
R5 →	x4566	another local variable	122
	x4567	previous frame pointer	x4570
	x4568	return address	x322C
	x4569	return value	(bits)
	x456A	g	92
	x456B	n	100
	x456C	p	-17

E)

R6 →	x5CCC	a local variable	101
	x5CCD	another local variable	731
	x5CCE	a third local variable	442
	x5CCF	previous frame pointer	x5D03
	x5CD0	return address	x3099
	x5CD1	return value	(bits)
R5 →	x5CD2	n	1000
	x5CD3	g	2000
	x5CD4	p	3000

Problem 2, continued:

Prof. Lumetta has translated the C code below into LC-3 assembly language shown beneath the C code.

```
int to_upper (char* s)
{
    int cnt = 0;
    while ('\0' != *s) {
        if ('a' <= *s && 'z' >= *s) {       // TEST LINE
            *s = *s - 'a' + 'A';            // UPDATE LINE
            cnt++;
        }
        s++;
    }
    return cnt;
}
```

```
TO_UPPER                // 01                    ADD R2,R2,#1   // 22
        ADD R6,R6,#-4   // 02                    BRn CCCCC      // 23
        STR R5,R6,#1    // 03                    LD  R1,EEEEE   // 24
        STR R7,R6,#2    // 04                    NOT R1,R1      // 25
        ADD R5,R6,#0    // 05                    ADD R0,R1,R0   // 26
        AND R0,R0,#0    // 06                    ADD R0,R0,#1   // 27
        STR R0,R5,#0    // 07                    BRp CCCCC      // 28
AAAAA   LDR R0,R5,#4    // 08                    LD  R0,FFFFF   // 29
        LDR R0,R0,#0    // 09                    ADD R2,R2,R0   // 30
        BRnp BBBBB      // 10                    LDR R0,R5,#4   // 31
        LDR R0,R5,#0    // 11                    STR R2,R0,#0   // 32
        STR R0,R5,#3    // 12                    LDR R0,R5,#0   // 33
        LDR R7,R5,#2    // 13                    ADD R0,R0,#1   // 34
        LDR R5,R5,#1    // 14                    STR R0,R5,#0   // 35
        ADD R6,R6,#3    // 15            CCCCC   LDR R0,R5,#4   // 36
        RET             // 16                    ADD R0,R0,#1   // 37
BBBBB   LDR R0,R5,#4    // 17                    STR R0,R5,#4   // 38
        LDR R0,R0,#0    // 18                    BRnzp AAAAA    // 39
        LD  R1,DDDDD    // 19            DDDDD   .FILL 'a'      // 40
        NOT R1,R1       // 20            EEEEE   .FILL 'z'      // 41
        ADD R2,R1,R0    // 21            FFFFF   .FILL 'A'      // 42
```

Circle EXACTLY ONE ANSWER for each question.

(5 points): Which lines of the assembly code set up the stack frame?

A) 02 to 07 B) 29 to 39 C) 11 to 16 D) 08 to 10 E) 02 to 05

(5 points): Which lines of the assembly code tear down the stack frame?

A) 29 to 39 B) 02 to 05 C) 13 to 15 D) 08 to 10 E) 36 to 39

(5 points): Which lines of the assembly code correspond to the “TEST LINE” in the C code?

A) 29 to 35 B) 17 to 28 C) 11 to 15 D) 08 to 10 E) 24 to 39

(5 points): Which line of the assembly code performs the write to *s in the “UPDATE LINE” of the C code?

A) 12 B) 07 C) 38 D) 32 E) 35

Problem 2, continued:

Prof. Lumetta has translated the C code below into LC-3 assembly language shown beneath the C code.

```
char* fill (int a, int b, char* s)
{
    int c;
    int i;

    if (a <= b) {                    // AB COMPARISON
        c = a;                       // INIT TO A
    } else {
        c = b;
    }
    for (i = 0; c > i; i++) {
        *(s++) = '+';                // LOOP BODY
    }
    return s;                        // RETURN S
}
```

```
FILL    ADD R6,R6,#-5     // 01
        STR R5,R6,#2      // 02
        STR R7,R6,#3      // 03
        ADD R5,R6,#1      // 04
        LDR R0,R5,#4      // 05
        LDR R1,R5,#5      // 06
        NOT R1,R1         // 07
        ADD R0,R0,R1      // 08
        ADD R0,R0,#1      // 09
        BRp AAAAA         // 10
        LDR R0,R5,#4      // 11
        STR R0,R5,#-1     // 12
        BRnzp BBBBB       // 13
AAAAA   LDR R0,R5,#5      // 14
        STR R0,R5,#-1     // 15
BBBBB   AND R0,R0,#0      // 16
        STR R0,R5,#0      // 17
CCCCC   LDR R0,R5,#-1     // 18
        LDR R1,R5,#0      // 19
        NOT R1,R1         // 20
                  ADD R0,R0,R1       // 21
                  ADD R0,R0,#1       // 22
                  BRp DDDDD          // 23
                  LDR R0,R5,#6       // 24
                  STR R0,R5,#3       // 25
                  LDR R7,R5,#2       // 26
                  LDR R5,R5,#1       // 27
                  ADD R6,R6,#4       // 28
                  RET                // 29
        DDDDD     LDR R0,R5,#6       // 30
                  LD R1,EEEEE        // 31
                  STR R1,R0,#0       // 32
                  ADD R0,R0,#1       // 33
                  STR R0,R5,#6       // 34
                  LDR R0,R5,#0       // 35
                  ADD R0,R0,#1       // 36
                  STR R0,R5,#0       // 37
                  BRnzp CCCCC        // 38
        EEEEE     .FILL x002B ; '+'  // 39
```

Circle EXACTLY ONE ANSWER for each question.

2. (5 points) Which lines of the assembly code perform the "AB COMPARISON" test in the C code?

A) `18 to 22` B) `30 to 38` C) `05 to 09` D) `11 to 15` E) `01 to 04`

3. (5 points) Which lines of the assembly code perform the "INIT TO A" statement in the C code?

A) `16 to 17` B) `11 to 12` C) `14 to 15` D) `30 to 34` E) `01 to 10`

Problem 2, continued: (code replicated for your convenience)

```
char* fill (int a, int b, char* s)
{
    int c;
    int i;

    if (a <= b) {                       // AB COMPARISON
        c = a;                          // INIT TO A
    } else {
        c = b;
    }
    for (i = 0; c > i; i++) {
        *(s++) = '+';                   // LOOP BODY
    }
    return s;                           // RETURN S
}
```

```
FILL    ADD R6,R6,#-5     // 01                ADD R0,R0,R1       // 21
        STR R5,R6,#2      // 02                ADD R0,R0,#1       // 22
        STR R7,R6,#3      // 03                BRp DDDDD          // 23
        ADD R5,R6,#1      // 04                LDR R0,R5,#6       // 24
        LDR R0,R5,#4      // 05                STR R0,R5,#3       // 25
        LDR R1,R5,#5      // 06                LDR R7,R5,#2       // 26
        NOT R1,R1         // 07                LDR R5,R5,#1       // 27
        ADD R0,R0,R1      // 08                ADD R6,R6,#4       // 28
        ADD R0,R0,#1      // 09                RET                // 29
        BRp AAAAA         // 10         DDDDD  LDR R0,R5,#6       // 30
        LDR R0,R5,#4      // 11                LD R1,EEEEE        // 31
        STR R0,R5,#-1     // 12                STR R1,R0,#0       // 32
        BRnzp BBBBB       // 13                ADD R0,R0,#1       // 33
AAAAA   LDR R0,R5,#5      // 14                STR R0,R5,#6       // 34
        STR R0,R5,#-1     // 15                LDR R0,R5,#0       // 35
BBBBB   AND R0,R0,#0      // 16                ADD R0,R0,#1       // 36
        STR R0,R5,#0      // 17                STR R0,R5,#0       // 37
CCCCC   LDR R0,R5,#-1     // 18                BRnzp CCCCC        // 38
        LDR R1,R5,#0      // 19         EEEEE  .FILL x002B ; '+'  // 39
        NOT R1,R1         // 20
```

Circle EXACTLY ONE ANSWER for each question.

4. (5 points) Which line of the assembly code performs ADD operation for the increment of `s` in the "LOOP BODY" of the C code?

A) `09` B) `04` C) `33` D) `22` E) `36`

5. (5 points) Which lines of the assembly code perform the "RETURN S" statement in the C code?

A) `18 to 22` B) `26 to 29` C) `24 to 25` D) `09 to 14` E) `30 to 34`

Problem 2, continued:

Prof. Lumetta has translated the C code below into LC-3 assembly language shown beneath the C code.

```
char* fill (int a, int b, char* s)
{
    int c;
    int i;

    if (a <= b) {
        c = a;
    } else {
        c = b;                          // INIT TO B
    }
    for (i = 0; c > i; i++) {           // FOR LOOP
        *(s++) = '+';
    }
    return s;
}
```

```
FILL    ADD R6,R6,#-5     // 01                ADD R0,R0,R1       // 21
        STR R5,R6,#2      // 02                ADD R0,R0,#1       // 22
        STR R7,R6,#3      // 03                BRp DDDDD          // 23
        ADD R5,R6,#1      // 04                LDR R0,R5,#6       // 24
        LDR R0,R5,#4      // 05                STR R0,R5,#3       // 25
        LDR R1,R5,#5      // 06                LDR R7,R5,#2       // 26
        NOT R1,R1         // 07                LDR R5,R5,#1       // 27
        ADD R0,R0,R1      // 08                ADD R6,R6,#4       // 28
        ADD R0,R0,#1      // 09                RET                // 29
        BRp AAAAA         // 10         DDDDD  LDR R0,R5,#6       // 30
        LDR R0,R5,#4      // 11                LD R1,EEEEE        // 31
        STR R0,R5,#-1     // 12                STR R1,R0,#0       // 32
        BRnzp BBBBB       // 13                ADD R0,R0,#1       // 33
AAAAA   LDR R0,R5,#5      // 14                STR R0,R5,#6       // 34
        STR R0,R5,#-1     // 15                LDR R0,R5,#0       // 35
BBBBB   AND R0,R0,#0      // 16                ADD R0,R0,#1       // 36
        STR R0,R5,#0      // 17                STR R0,R5,#0       // 37
CCCCC   LDR R0,R5,#-1     // 18                BRnzp CCCCC        // 38
        LDR R1,R5,#0      // 19         EEEEE  .FILL x002B ; '+'  // 39
        NOT R1,R1         // 20
```

Circle EXACTLY ONE ANSWER for each question.

2. (5 points) Which lines of the assembly code set up the stack frame?

A) `01 to 06` B) `30 to 38` C) `24 to 28` D) `14 to 17` E) `01 to 04`

3. (5 points) Which lines of the assembly code tear down the stack frame?

A) `01 to 06` B) `30 to 38` C) `26 to 28` D) `24 to 28` E) `18 to 22`

Problem 2, continued: (code replicated for your convenience)

```
char* fill (int a, int b, char* s)
{
    int c;
    int i;

    if (a <= b) {
        c = a;
    } else {
        c = b;                          // INIT TO B
    }
    for (i = 0; c > i; i++) {           // FOR LOOP
        *(s++) = '+';
    }
    return s;
}
```

```
FILL    ADD R6,R6,#-5     // 01                ADD R0,R0,R1       // 21
        STR R5,R6,#2      // 02                ADD R0,R0,#1       // 22
        STR R7,R6,#3      // 03                BRp DDDDD          // 23
        ADD R5,R6,#1      // 04                LDR R0,R5,#6       // 24
        LDR R0,R5,#4      // 05                STR R0,R5,#3       // 25
        LDR R1,R5,#5      // 06                LDR R7,R5,#2       // 26
        NOT R1,R1         // 07                LDR R5,R5,#1       // 27
        ADD R0,R0,R1      // 08                ADD R6,R6,#4       // 28
        ADD R0,R0,#1      // 09                RET                // 29
        BRp AAAAA         // 10         DDDDD  LDR R0,R5,#6       // 30
        LDR R0,R5,#4      // 11                LD R1,EEEEE        // 31
        STR R0,R5,#-1     // 12                STR R1,R0,#0       // 32
        BRnzp BBBBB       // 13                ADD R0,R0,#1       // 33
AAAAA   LDR R0,R5,#5      // 14                STR R0,R5,#6       // 34
        STR R0,R5,#-1     // 15                LDR R0,R5,#0       // 35
BBBBB   AND R0,R0,#0      // 16                ADD R0,R0,#1       // 36
        STR R0,R5,#0      // 17                STR R0,R5,#0       // 37
CCCCC   LDR R0,R5,#-1     // 18                BRnzp CCCCC        // 38
        LDR R1,R5,#0      // 19         EEEEE  .FILL x002B ; '+'  // 39
        NOT R1,R1         // 20
```

Circle EXACTLY ONE ANSWER for each question.

4. (5 points) Which line of the assembly code performs ADD operation for the increment of `i` in the "FOR LOOP" of the C code?

A) `09` B) `33` C) `04` D) `22` E) `36`

5. (5 points) Which lines of the assembly code perform the "INIT TO B" statement in the C code?

A) `16 to 17` B) `24 to 25` C) `11 to 12` D) `14 to 15` E) `30 to 34`