ECE 220 Summer 2017 Midterm 2

On-campus and Western Hemisphere students

- Absolutely no interaction between students or any external help is allowed!

- You are not allowed to use any applications on your desktop outside the Exam VM.
- You can use any applications inside the Exam VM.

- You are permitted one double-sided page of **hand-written** notes.

- Read entire problem description before starting to work on the solution.

- **IMPORTANT: Once done, you must commit your work to Subversion:**

```
cd ~/midterm2
svn commit -m "I am done"
```

- To verify your submission, point web browser inside the Exam VM to the following page: **https://subversion.ews.illinois.edu/svn/su17-ece220/NETID/midterm2** where NETID is your actual NetID.

- If your attempt to commit/verify fails, make sure you are still connected to the UIUC network via the VPN client. If your connection to UIUC network is terminated during the exam, you must reconnect again.

Good luck!

Part 1: C to LC-3 with run-time stack (40%)

Problem Statement

Convert the following C program given in file `part1/part1.c` from C to LC-3 assembly language. Note that the majority of the code is already converted and you just need to finish parts marked with **IMPLEMENT THIS**. Do not modify other parts of the assembly code, especially labels!

```
int main()
{
    int a, b=5;
    int array[5] = { 5, 4, 3, 2, 1 };

    a = function(array, &b);

    printf("a=%d, b=%d\n", a, b);

    return 0;
}

int function(int array[], int *n)
{
    /* terminal case */
    if (*n == 0) return 0;

    /* reduction case */
    *n = *n - 1;
    return array[*n] + function(array, n);
}
```

Complete your code in `part1.asm` in your `part1` folder. **Do not forget to commit your work!**

Implementation requirements

You must use the run-time stack convention presented in the textbook and lectures to invoke the subroutine. To receive full credit, `part1.asm` should correctly run in lc3sim (or lc3sim-tk).

Your code will be graded for functionality as well as the correct construction of the activation record in the run-time stack. Consequently, you MAY NOT perform any optimizations that impact the contents of the stack. ALL local variables used in the C functions should be stored in the run-time stack appropriately. The local variables may NOT be stored locally in memory reserved by .FILL, .BLKW, etc. You may receive zero points if your code does not assemble or does not behave as specified.

Feel free to compile and run the provided C code to observe its behavior and output. The same result must be produced by your LC-3 program and stored in the memory allocated on the run-time stack for variable `a`.

Important: your code for each "IMPLEMENT THIS" section must be between the designated lines, e.g.:

```
; IMPLEMENT THIS: push &b onto the stack
; ----- place your code between these lines only ----

code for pushing &b onto the stack must be written between these two lines

; ----- place your code between these lines only ----
```

Part 2: Arrays and pointers (40%)

Problem Statement

You are provided with *unfinished* code that detects and labels object's boundary pixels. Your task is to implement function, `pixelType`, that classifies image pixel specified by its coordinates `(x,y)` into one of three categories: belonging to an object, to the background, or to the object's boundary.

```
void pixelType(unsigned char *image, int height, int width, int x, int y,
               int *background, int *object, int *boundary);
```

This function accepts base address of image array, `image`, image dimensions, `height` and `width`, and coordinates `x` and `y` of a pixel to be classified. It sets (assigns value of 1) one of the output variables provided by their pointers, *`background`, *`object`, or *`boundary`, to indicate the pixel type and clears (assigns value of 0) the remaining two variables. **Do not modify this function prototype!**

Image representation details

In this assignment, a 2-dimensional *black-and-white* image of size `width` by `height` pixels is stored in memory in row-major order as a 1-dimensonal array. Thus, `image[0]` contains value of pixel (0, 0), `image[1]` contains value of pixel (1, 0), `image[2]` contains value of pixel (2, 0), and so on. Here the first coordinate, `x`, is pixel's *column index* and the second coordinate, `y`, is the pixel's *row index*.

Pixel values in the input image are either 0 or 1. Pixel value 0 means the pixel is part of the background. Pixel value 1 means the pixel is part of an object. Shown below on the left is a printout of a sample *16 columns* by *8 rows* image. Shown on the right is the same image with object boundary pixels labeled with value 2 (also highlighted by yellow color). The image on the right is the expected output of your program once you correctly implement `pixelType` function.

```
0000001111000000        000000222200000
0000001111000000        000000211200000
0000011111100000        000002112200000
0000111111110000        000022111122000
0001111111110000        000221111120000
0001111111110000        000211122220000
0001111100000000        000222220000000
0000000000000000        000000000000000
```

Pixel is a *boundary pixel* if 1) it belongs to an object and 2) at least one of its 8 immediate neighbor pixels belong to the background. All pixels outside the image are treated as belonging to the background.

Implementation requirements

You are provided with several files in `part2` directory, including `Makefile`. Your implementation must be done in `image.c` file and you may use `main.c` to write test code in addition to the already provided code, if needed. Study the code in `main.c` and `image.c` to understand the use and requirements of `pixelType` function that you must implement. **Do not modify `contour` function.**

To compile, type `make`. If successful, this will produce `part2` executable.

To run, type `./part2`. This will execute `contour` function on a sample input image and will print the original image and the modified image for your visual examination. Note that sample image provided in the exam is different from the above example. **Do not forget to commit your work!**

Part 3: Recursion (20%)

Problem statement

Given integers a, n, and m with $n \geq 0$ and $0 \leq a < m$, write a recursive function that computes $a^n \pmod m$. For example, $3^2 \pmod{10} = 9$, $3^2 \pmod 4 = 1$, $3^2 \pmod 5 = 4$.

Algorithm

Implement Fast Exponential algorithm for computing $a^n \pmod m$ using the following recursive relation for a^n:

$$a^n = \begin{cases} 1, & if\ n = 0 \\ \left(a^{n/2}\right)^2, & if\ n\ is\ even \\ a\left(a^{(n-1)/2}\right)^2, & if\ n\ is\ odd \end{cases}$$

Implementation requirements

Provide function prototype and implement function `FastExp` in `part3/part3.c` file. To receive full credit, your code must compile, run, and produce correct results. Do not modify `main()` function.

part3/part3.c file

```
#include <stdio.h>

/* IMPLEMENT ME: write function prototype here */

/* ! DO NOT MODIFY MAIN() ! */
int main()
{
    int a, n, m;

    printf("Enter three positive numbers a n m: ");
    scanf("%d %d %d", &a, &n, &m);
    printf("%d^%d(mod %d) is %d\n", a, n, m, FastExp(a, n, m));

    return 0;
}

/* Fast Exponential algorithm pseudocode:
   FastExp(a, 0, m) is 1
   FastExp(a, 1, m) is a
   x = FastExp(a, n/2, m)
   FastExp(a, n, m) is x^2(mod m) if n is even
   FastExp(a, n, m) is (x^2a)(mod m) if n is odd
 */
int FastExp(int a, int n, int m)
{
 /* implement me */
}
```

Do not forget to commit your work!