ECE 220 Summer 2016 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/su16-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 gets lost 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!

```
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`.

Supplied part1.asm code

```
.ORIG x3000

;int main()
MAIN
    ; setup stack
    LD R5, STACKTOP   ; R5 - frame pointer for main()
    LD R6, STACKTOP   ; R6 - ToS pointer
    ADD R6 , R6 , #1  ; stack is initially empty

    ; allocate and initialize local variables
    ADD R6, R6, #-1  ; push a
    ADD R6, R6, #-1  ; push b
    AND R0, R0, #0   ; b = 5;
    ADD R0, R0, #5
    STR R0, R6, #0
    ADD R6, R6, #-5  ; push array
    AND R0, R0, #0
    ADD R0, R0, #5   ; array[0]=5
    STR R0, R6, #0
    ADD R0, R0, #-1   ; array[1]=4
    STR R0, R6, #1
    ADD R0, R0, #-1   ; array[2]=3
    STR R0, R6, #2
    ADD R0, R0, #-1   ; array[3]=2
    STR R0, R6, #3
    ADD R0, R0, #-1   ; array[4]=1
    STR R0, R6, #4

    ; call subroutine
    ; IMPLEMENT THIS: push &b onto the stack

    ; IMPLEMENT THIS: push array base address onto the stack

    ; call subroutine
    JSR FUNCTION

    ; get return value
    LDR R0, R6, #0
    ADD R6, R6, #1
    STR R0, R5, #0 ; a = return value

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

    ; free stack
    ADD R6, R6, #2 ; pop arguments
    ADD R6, R6, #7 ; pop local variables

HALT           ; return

STACKTOP .FILL x30FF
```

Continue on the next page

```
;int function(int array[], int *n)
FUNCTION
    ; IMPLEMENT THIS: push bookkeeping info onto the stack

    ; setup frame pointer
    ADD R5, R6, #-1

TERMINAL_CASE
    ;    if (*n == 0) return 0;
    LDR R0, R5, #5
    LDR R1, R0, #0
    BRp REDUCTION_CASE
    ; return 0
    AND R0, R0, #0
    STR R0, R5, #3
    BR DOWN

REDUCTION_CASE
    ; IMPLEMENT THIS: *n = *n - 1;

    ;    return array[*n] + function(array, n);
    ; IMPLEMENT THIS: R0 <- array[*n]

    ; temporary store array[*n] (R0) on the stack
    STR R0, R5, #3   ; save array[*n] in temp storage

    ; setup function call
    LDR R0, R5, #5
    ADD R6, R6, #-1
    STR R0, R6, #0
    LDR R0, R5, #4
    ADD R6, R6, #-1
    STR R0, R6, #0

    JSR FUNCTION

    ; IMPLEMENT THIS: get return value
                      ; R0 <- function(array, n)
                      ; pop return value and 2 function arguments

    LDR R1, R5, #3   ; read array[*n] from temp storage

    ADD R0, R0, R1   ; R0 <- array[*n] + function(array, n)
    STR R0, R5, #3   ; store return value

DOWN
    ; restore R5/R7 and return
    LDR R5, R6, #0
    ADD R6, R6, #1 ; pop R5
    LDR R7, R6, #0
    ADD R6, R6, #1 ; pop R7
RET

.END
```

Part 2: Arrays and pointers (35%)

Problem Statement

Implement the following three functions:

```
void getPixel(int *image, int height, int width, int x, int y, int *r, int
*g, int *b);
```

This function reads image pixel (x, y) values r, g, and b.

```
void setPixel(int *image, int height, int width, int x, int y, int r, int g,
int b);
```

This function sets image pixel (x, y) values to r, g, and b.

```
void invertImage(int *inImage, int *outImage, int height, int width);
```

This function converts image to negative.

Image representation details

In this assignment, a 2-dimensional image of size `width` by `height` pixels is stored in memory in row-major order as a 1-dimensonal array. Each 3 consecutive values in the image array store *red*, *green*, and *blue* color channel components for a single pixel. Thus, `image[0]` contains *red* component of pixel (0, 0), `image[1]` contains *green* component of pixel (0, 0), and `image[2]` contains *blue* component of pixel (0, 0). Similarly, `image[3]` contains *red* component of pixel (0, 1), `image[4]` contains *green* component of pixel (0, 1), and `image[5]` contains *blue* component of pixel (0, 1).

Implementation requirements

`getPixel` and `setPixel` functions require pointer to the `image` array and image size, `height` and `width`, as inputs. They also require pixel coordinates, `x` and `y`, as inputs. `setPixel` function sets the pixel at the specified coordinates `(x, y)` to the specified color `(r, g, b)`. `getPixel` function retrieves the `(r, g, b)` color value of the pixel at the specified location `(x, y)`.

`invertImage` converts input image passed to it as pointer `inImage` to a *negative* representation returned in `outImage` by subtracting each color channel value for each pixel from 255. For example, negative of pixel (r=0, g=200, b=255) is (r=255, g=55, b=0). `invertImage` function **must use** `getPixel` and `setPixel` functions to get and set pixel values.

You are provided with several files in `part2` directory, including `Makefile`. Your implementation must be done in `image.c` file and you **must not modify** any other files, but you are welcome to examine them. **Do not forget to commit your work!**

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

To run, type `./part2 example.ppm result.ppm <x> <y>`, where <x> and <y> are pixel coordinates for which you would like to see output of your program. For example, `./part2 example.ppm result.ppm 25 125` should produce the following output:

```
before: image[ 25, 125] = 202, 207, 226
 after: image[ 25, 125] =  53,  48,  29
```

`example.ppm` file contains sample image used in this assignment to test your implementation. `result.ppm` will contain the negative image produced by your program. You can open these images with image viewer `qiv` to examine their content.

Part 3: Recursion (25%)

Problem statement

Write a **recursive function** that computes greatest common divisor (GCD) of two *non-negative* integers. Greatest common divisor of two integers, where at least one of them is not zero, is the largest positive integer that divides the numbers without a remainder. For example, the GCD of 8 and 12 is 4.

Algorithm

Implement Euclid's algorithm for computing GCD using the following recursive relation:

```
GCD(M, 0) is M.
GCD(M, N) is GCD(N, M) if M < N.
GCD(M, N) is N if N ≤ M and N divides M.
GCD(M, N) is GCD(N, remainder of M divided by N) otherwise.
```

Implementation requirements

Provide function prototype and implement function `GCD` in `part3/part3.c` file. To receive full credit, your code must compile, run, and produce correct results. Do not modify `main()` function. **Do not forget to commit your work!**

part3/part3.c file

```c
#include <stdio.h>

/* IMPLEMENT ME: write function prototype here */


/* ! DO NOT MODIFY MAIN() ! */
int main()
{
    int M, N;

    printf("Enter two positive numbers: ");
    scanf("%d %d", &M, &N);
    printf("GCD of %d and %d is %d\n",  M, N, GCD(M, N));

    return 0;
}


/*
 * GCD(M, 0) is M.
 * GCD(M, N) is GCD(N, M) if M < N.
 * GCD(M, N) is N if N ≤ M and N divides M.
 * GCD(M, N) is GCD(N, remainder of M divided by N) otherwise.
 */
int GCD(int M, int N)
{
 /* implement me */
}
```