ECE 220: Computer Systems and Programming

Spring 2021 – Midterm Exam 2

April 8, 2021

1. This is a closed-book, closed-notes exam
2. Absolutely no interaction between students is allowed
3. Illegible handwriting will be graded as incorrect
4. You must put your name and NetID on your submission page
5. **Use a separate page for each question**
6. Submission is only accepted through Gradescope

Question 1 (36 points): _____

Question 2 (30 points): _____

Question 3 (20 points): A)_____; B)_____

Question 4 (14 points): 1)_____; 2)_____

Total Score: _____

Write down your answers in the following format. Each question should be on a separate page. Tag each question on your Gradescope submission.

Name:
NetID:

Q1 (**write down the # and entire line of code highlighted in yellow**)

Q2 (**write down the # and entire line of code highlighted in yellow**)

Q3 (**write down the # and entire line of code highlighted in yellow**)
A)
B)

Q4
1)
2)

Problem 1 (36 points): Array

Given an m x n 2D matrix, validate if the sum of the left diagonal is larger than the sum of the right diagonal in every 2 x 2 submatrices. Return True if the given matrix satisfies this statement. Otherwise, return False.

Example Input: a 5 x 4 matrix (m = 5, n = 4)

4	1	2	0
3	5	4	9
5	1	4	11
9	6	9	8
1	12	3	6

Output: False

Explanation:
1st submatrix: 4+5(left diagonal sum) > 1+3(right diagonal sum)
2nd submatrix: 2+9 > 0+4
3rd submatrix: 5+6 > 1+9
4th submatrix: 4+8 < 11+9
The fifth row that cannot form a 2x2 submatrix, therefore it is ignored

Note:

- Implement a helper function that converts 2D index to 1D index assuming that the 2D matrix is stored in row-major order.
- Extra row or column that cannot form a 2x2 submatrix should be ignored.

Algorithm for checkSubMatrix() function:

- Start with the upper left cell in each 2x2 submatrix
- Calculate the upper bounds for row and column index within each 2x2 submatrix
- If the bounds are within range, calculate left diagonal sum and right diagonal sum
- Return false if left diagonal sum is less than or equal to right diagonal sum
- Return true if left diagonal sum is greater than right diagonal sum for every 2x2 submatrix

/* FILL IN THE BLANKS BELOW TO FINISH THE PROGRAM */

```
int linearInd(int i, int j, int length){
     /* Given the 2D index(i, j) and the length of each
        row, return its 1D index in row-major order */
     return (1)____________;
}

bool checkSubMatrix(int* mat, int m, int n){
   int i,j;
   for (i = 0; (2) ___________; i=i+2){
      for (j = 0; (3) ___________; j=j+2){

          /* calculate the upper bounds for row and
             column index within the 2x2 submatrix*/
          int row_bound = (4)______;
          int col_bound = (5)______;

          /* check if row_bound and col_bound are within
             range of the matrix */
          if((6)_____________________________________){
               /*calculate the left diagonal sum */
               int left_sum = (7)_____________________;
               /* calculate the right diagonal sum */
               int right_sum = (8)____________________;
               if(left_sum <= right_sum){
                    (9)___________;
               }
          }
      }
   }

   (10)_____________;
}
```

Problem 2 (30 points): Recursion

The easter bunny has contracted you to create a program to map out where eggs can be hidden for next year's easter egg hunt (they really start preparations early). To facilitate this, you must fill in a recursive function **place_eggs** which uses basic backtracking to find a proper placement, if there is one. This function will go through the rows one at a time placing an egg if possible until it runs through all rows or fails to find a solution.
The rules for the placement are as follows:

1. The grid is a 7x7 field
2. Empty spots where eggs can be placed are marked 'o'
3. Existing eggs are marked 'x'
4. Blocked locations (like walls/shrubs/etc) where eggs cannot be placed are marked '+'
5. There must be exactly one egg in each row
6. No eggs may share a column

To aid in your solution, the helper functions **is_spot_valid** and **is_row_valid** have been provided. Their function prototype is shown below. The first helper function determines if a spot can be used for an egg given all the above constraints, the second function determines if a row can have an egg placed in it.

```
/* returns 1 for spot which can be used, 0 for spot which
   does not meet above requirements */
  int is_spot_valid(char grid[7][7], int row, int col);

/* returns 1 for row which can be used, 0 for row with
   egg already in it */
  int is_row_valid(char grid[7][7], int row);
```

Two example runs are shown below of the function:

```
Printing start grid:

++oo+oo
o+oo+oo
oo++++o
ooo++oo
o++++oo
oo+++oo
ooooooo

A solution was found! Easter is saved!!

Printing end grid:

++xo+oo
o+ox+oo
xo++++o
oxo++oo
o++++xo
oo+++ox
oooxoo
```

```
Printing start grid:

++oo+oo
o+oo+oo
oo++++o
ooo++oo
o++++oo
oo+++oo
oooo+oo

No solution found, sad easter bunny...

Printing end grid:

++oo+oo
o+oo+oo
oo++++o
ooo++oo
o++++oo
oo+++oo
oooo+oo
```

FILL IN THE BLANKS BELOW TO FINISH THE PROGRAM

```
/* function to place eggs into grid
   backtracks with each row
   returns 1 on success, 0 on failure */
int place_eggs(char grid[7][7]){

     /* part 1: find the row to fill */
     int row;
     for(row=0;row(1)_______;row++){
          //exit if we find valid row
          if(is_row_valid((2)______,______))
                (3)______;
     }

     /* base case:
        there is no need to fill a row (we are done),
        happens when no valid row found */
     if((4)______)
          return 1;

     /* part 2: test every possible cell in the row */
     int col;
     for(col=0;col(5)______;col++){
          /* check if spot is valid */
          if(is_spot_valid((6)____,____,____)){
               (7)_______________; /* fill with egg */

               /* recursively try to solve grid */
               if((8)___________)
                    return (9)________;

               (10)_____________; /* backtrack */
          }
     }

     return 0; /* if every choice fails */
}
```

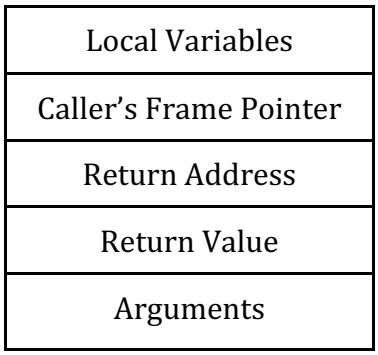
Problem 3 (20 points): C to LC-3 Conversion

For this question, you will be converting a function `foo` from C to LC-3.

Here is the code that we ask you to convert (note that you only need to convert `foo` to LC-3). You must use and conform to the LC-3 calling conventions we have described in class.

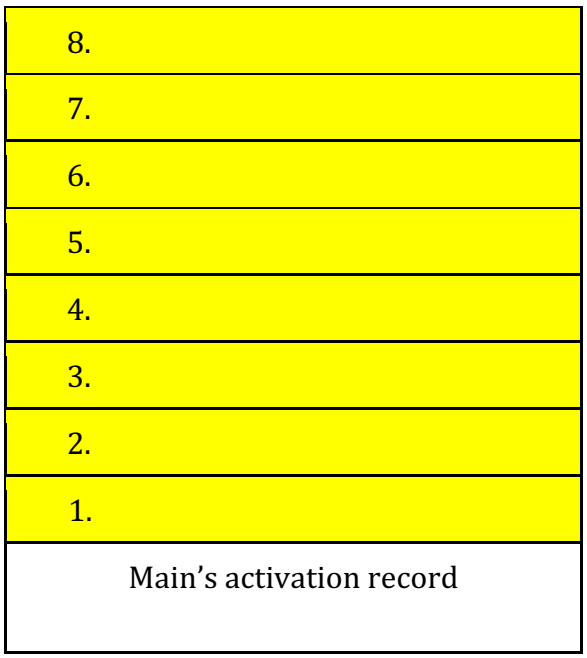
```
int foo(int a, int b) {
     int res_1, res_2, final_result;

     /* code omitted for simplicity */

     return final_result;
}

int main() {
     int answer = foo(1,2);

     /* do something with answer */

     return 0;
}
```

Recall that a function's activation record has the following format:



Part A (8 points).

Draw the complete run-time stack activation record (stack frame) for a call of `foo(1,2)`. You MUST conform to the LC-3 calling conventions we have described in class, including callee activation record build-up. **Use labels (variable names) instead of values whenever possible.**



Part B (12 points).

For this part, you should convert `foo` from C to LC-3 by filling in the blank lines. Consider that the caller (i.e. main function) has pushed the arguments into the activation record and the program control is just transferred to the callee function (i.e. foo). We have provided comments for what you have to write for each line of code. For each line of code, **you must implement what is described in the comments for that line**. Note that "var." stands for "variable", "ptr." stands for "pointer", "ret." stands for return, and "addr." stands for address.

```
; You may assume R0~R5 all contain zeros.
; R6 is the stack pointer. R5 is the frame pointer.
; R7 contains the return address.

FOO
;;Callee set-up
(1) _______________ ; allocate space for ret. value and ret. addr.
(2) _______________ ; save ret. addr. to stack
(3) _______________ ; allocate space for caller's frame ptr.
(4) _______________ ; save caller's frame ptr. to stack
(5) _______________ ; allocate space for local variables
(6) _______________ ; update frame ptr.


;;function logic omitted for simplicity


;;set ret. value
(7) _______________ ; Using R5, load var. final_result into R0
(8) _______________ ; Using R5, save ret. value of foo to stack


;;Callee tear-down
(9) _______________ ; pop local variables off stack
(10) _______________; Using R6, restore caller's frame ptr.
(11) _______________; Using R6, restore ret. addr.
(12) _______________ ; pop caller's frame ptr. and ret. addr.

RET
```

Problem 4 (14 points): Concepts

1. (5 points) The following function call to **`sum()`** will compute the sum of the array elements in an integer array. What is the size of the first argument of the function **`sum()`** in bytes? (Assume you are on a 32-bit system and the size of int is 32 bits.)

```
int main() {
     int array[5] = {1,2,3,4,5};
     int result;
     result = sum(array, 5);
     return 0;
}
```

Your Answer: _____Bytes

2. (9 points) Determine whether the statements are true for the following code.

```
typedef struct CourseStruct{
     int year;
     int class_number;
} course;

int main(){
     course A[2];
     course* B;
     course* C;

     A[0].year = 2021;
     A[0].class_number = 220;
     A[1].year = 2020;
     A[1].class_number = 120;

     B = A;
     C = &(A[1]);

     return 0;
}
```

Your Answer:

a. B == &(A[0]) TRUE FALSE

b. B->class_number == 120 TRUE FALSE

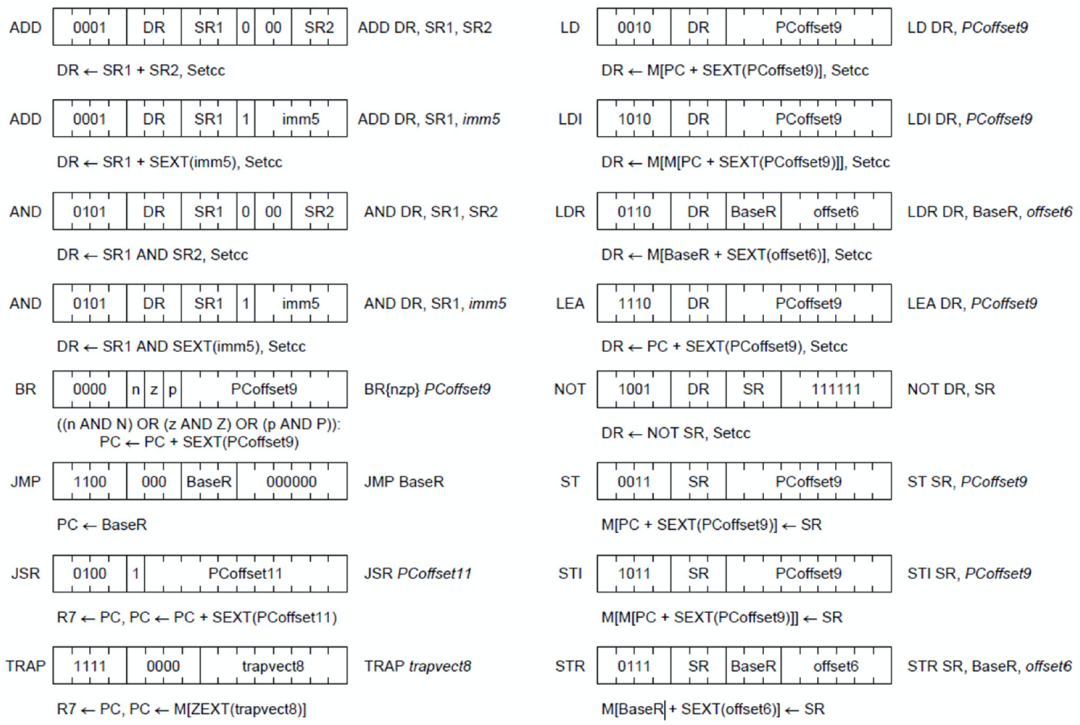
c. (B+1) == C TRUE FALSE

Table E.2 The Standard ASCII Table

ASCII			ASCII			ASCII			ASCII		
Character	Dec	Hex	Character	Dec	Hex	Character	Dec	Hex	Character	Dec	Hex
nul	0	00	sp	32	20	@	64	40	`	96	60
soh	1	01	!	33	21	A	65	41	a	97	61
stx	2	02	"	34	22	B	66	42	b	98	62
etx	3	03	#	35	23	C	67	43	c	99	63
eot	4	04	$	36	24	D	68	44	d	100	64
enq	5	05	%	37	25	E	69	45	e	101	65
ack	6	06	&	38	26	F	70	46	f	102	66
bel	7	07	'	39	27	G	71	47	g	103	67
bs	8	08	(	40	28	H	72	48	h	104	68
ht	9	09	)	41	29	I	73	49	i	105	69
lf	10	0A	*	42	2A	J	74	4A	j	106	6A
vt	11	0B	+	43	2B	K	75	4B	k	107	6B
ff	12	0C	'	44	2C	L	76	4C	l	108	6C
cr	13	0D	-	45	2D	M	77	4D	m	109	6D
so	14	0E	.	46	2E	N	78	4E	n	110	6E
si	15	0F	/	47	2F	O	79	4F	o	111	6F
dle	16	10	0	48	30	P	80	50	p	112	70
dc1	17	11	1	49	31	Q	81	51	q	113	71
dc2	18	12	2	50	32	R	82	52	r	114	72
dc3	19	13	3	51	33	S	83	53	s	115	73
dc4	20	14	4	52	34	T	84	54	t	116	74
nak	21	15	5	53	35	U	85	55	u	117	75
syn	22	16	6	54	36	V	86	56	v	118	76
etb	23	17	7	55	37	W	87	57	w	119	77
can	24	18	8	56	38	X	88	58	x	120	78
em	25	19	9	57	39	Y	89	59	y	121	79
sub	26	1A	:	58	3A	Z	90	5A	z	122	7A
esc	27	1B	;	59	3B	[	91	5B	{	123	7B
fs	28	1C	<	60	3C	\	92	5C	|	124	7C
gs	29	1D	=	61	3D	]	93	5D	}	125	7D
rs	30	1E	>	62	3E	^	94	5E	~	126	7E
us	31	1F	?	63	3F	_	95	5F	del	127	7F

LC-3 Instructions

NOTES: RTL corresponds to execution (after fetch!); JSRR not shown



End of ECE 220 Midterm Exam 2