MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.1904: Introduction to Low-level Programming in C and Assembly

Spring 2026, Quarter 4

Name: Answers	**Kerberos:** answers
	MIT ID #: 0x123456789

#1 (11)	11
#2 (12)	12
#3 (14)	14
#4 (14)	14
#5 (14)	14
#6 (11)	11
#7 (11)	11
#8 (13)	13
Total (100)	100

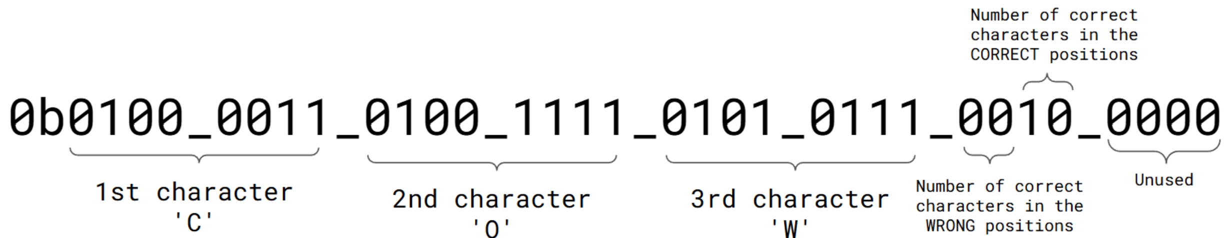
Exam content is on **BOTH SIDES** of the exam sheets.

Enter your answers in the boxes designated for each problem. Show your work for potential partial credit.

IMPORTANT: Avoid talking about and communicating the contents of this exam with other students until we have announced it is ok to do so on Piazza. Failure to do so will be considered an academic policy violation.

Problem 1. Mini Wordle (11 points)

You are tasked with creating a mini Wordle game, where the player guesses a hidden 3-character word using binary representations. The objective is to guess the secret word using the following unsigned 32-bit format for storing the guess:



- Bits 31-24: Stores the ASCII value of the first character
- Bits 23-16: Stores the ASCII value of the second character
- Bits 15-8: Stores the ASCII value of the third character
- Bits 7-6: Stores the number of correct characters in the wrong positions
- Bits 5-4: Stores the number of correct characters in the correct positions
- Bits 3-0: Unused

For example, suppose the guess is `"COW"` but the hidden word is `"BOW"`. Since `'O'` and `'W'` are correct characters in the correct positions, bits 5-4 store the value 2. There are no correct characters in the wrong positions, so bits 7-6 store the value 0.

All characters in the words are unique and uppercase. The player wins by correctly guessing all characters in their correct positions.

A. (3 points) Write the binary representation for the guess `"ANT"` given that the hidden word is `"NOT"`. **Indicate all 32 bits.**

`0b0100_0001_0100_1110_0101_0100_0101_0000`

B. (2 points) Later, you decide to extend the game to 4-character words and store only 7 bits per ASCII character, since the most significant ASCII bit is always 0. Considering that some fields may need more bits for 4-character words and that you may use unused bits for extra space, is it possible to represent the game using an unsigned 32-bit integer following the same design idea as Part A? Explain why or why not in no more than 2 sentences.

No, because the 4 letters already use 28 bits, and to handle a 4-letter word we need 3 bits each for "correct letters in wrong position" and "correct letters in correct position". So, 32 bits are not enough to store 34 bits of information

C. (6 points) You now realize that since the word contains only uppercase characters A-Z, each character can be mapped to a value from 1-26 where A=1, B=2, ..., Z=26. This means each character only requires 5 bits, so a 4-character word needs 20 bits total.

Using this information, you redesign the encoding system to use a 32-bit variable as follows:

- Bits 31-27: Represent the first character
- Bits 26-22: Represent the second character
- Bits 21-17: Represent the third character
- Bits 16-12: Represent the fourth character

The remaining bits are used to store the following in this order (from most to least significant):

- The number of correct characters in the wrong positions
- The number of correct characters in the correct positions
- Zeros for unused bits

Use the minimum number of bits necessary to store these values.

For each of the scenarios on the next page, you are given a variable of type `uint32_t` named `guess` that already contains the encoded word, along with the filled-in counts for correct and misplaced characters.

Problem continued on next page.

Notes:

- For the following questions, you **may not** use arithmetic operators (`+, -, /, *, %`).
- Your answer should be in terms of the variable `guess`.

Write an expression that evaluates to `0` if the word has been guessed correctly, and `1` otherwise.

```
uint32_t answer = ((guess >> 6) & 0b111) != 4;
//left outer parentheses are needed
```

Write an expression that evaluates to `1` if the first character is `'B'`, and `0` otherwise.

```
uint32_t answer = ((guess >> 27) & 0b11111) == 2;
//left outer parentheses are needed
```

Write an expression that changes the second character to `'X'` while keeping everything else unchanged.

```
uint32_t answer = (guess & 0xF83FFFFF) | (24 << 22);
```

Problem 2. Stringy Way Galaxy (12 points)

You are travelling with your spaceship in the Stringy Way Galaxy, in which distances between planets are recorded as characters, where each character corresponds to its **ASCII decimal value** in miles. For instance, if the recorded distance between planet **A** and planet **B** is `'h'`, the distance in miles is 104 (you may check the ASCII table to confirm). **You may assume the library `string.h` has been included for all parts of this problem.**

A. (3 points) You went through a trip between multiple planets, and you recorded your journey in a **C string** variable `trip_record`, with each character in `trip_record` representing the distances you travelled in order. Write a function that computes the total distance you travelled in miles.

```
#include <string.h>

uint32_t computeDistance(char* trip_record) {
      // YOUR CODE BELOW


    int distance = 0;
    int i = 0;
    while(trip_record[i]) {
        distance += (int) trip_record[i];
        i++;
    }
    return distance;




}
```

Problem continued on next page.

B. (5 points) Now, assume that throughout your trip, you filled your spaceship's gas multiple times. Each time you did it, you recorded it as an `'_'` in your `new_trip_record` (`new_trip_record` is a properly-terminated **C string** that looks like this: `"ahgt_HnO_RTT_"`).

Assume the following:

- Each time you fill your tank, you fill it completely.
- You end your trip with filling your tank.
- Each mile requires $1 worth of gas.
- You'll never encounter "back-to-back" fill-ups so two sequential `'_'` will not occur (e.g. `"__"` is impossible)
- A trip will never start with a fill-up, so no string will start with `'_'`.

Fill in the blanks in the function below to calculate how many dollars you pay for the entire trip. Your function may modify `new_trip_record`. You have access to `computeDistance` from the previous part.

```
#include <string.h>

uint32_t tripCost(char* new_trip_record) {
    uint32_t cost = 0;
    char *trip_segment;
    char *delim = __BLANK1__;
    trip_segment = __BLANK2__;

    while (trip_segment != __BLANK3__) {
        cost += computeDistance(trip_segment);
        trip_segment = __BLANK4__;
    }
    return __BLANK5__;
}
```

Blank #:	Line of code:
`__BLANK1__`	`"_"`
`__BLANK2__`	`strtok(new_trip_record, delim)`
`__BLANK3__`	`NULL`
`__BLANK4__`	`strtok(NULL, delim)`
`__BLANK5__`	`cost`

C. (4 points) You now want to record how much you paid at each gas station in a compact "stringy" format. Assume that the cost paid at each station is a single-digit integer (1–9). You are given an integer array `costs`, where `costs[i]` is the amount paid at the `i`-th station, and an integer `num_stations` representing the number of stations visited.

Write a function that constructs a null-terminated string `cost_record` such that each cost is encoded as its corresponding character digit.

For example, if `costs = {1,4,9,8,7}`, then `cost_record` points to the null-terminated string `"14987"`. You may assume that `cost_record` has enough space to store the result.

```
#include <string.h>

void recordCosts(char *cost_record, uint32_t *costs, uint32_t num_stations)
{

      for (int i = 0; i < num_stations; i++) {
            cost_record[i] = costs[i] + '0';
      }
      cost_record[num_stations] = '\0';

}
```

This page intentionally left blank

Problem 3. MIT Student Database (14 points)

You are helping MIT construct a new database that tracks subjects taken by each student. A struct called `Student` is defined below to represent a student in the system. A struct called `Subject` stores information about the class.

You may assume the library `string.h` has been included for all parts of this problem.

```
struct Subject {
  uint8_t num_units;         // Number of units a subject has
  char    subject_name[100]; // subject_name format is [major].[number]: [name]
                             //e.g., "6.1904: Introduction to Low-level Program..."
};
struct Student {
  uint32_t student_id;             // Unique student ID
  char     major[4];               // Major of the student (e.g., "6" or "21W")
  struct   Subject subjects[100];  // Array of subjects taken by student so far
  uint32_t num_subjects;           // Number of subjects taken so far
};
```

A. (3 points) Write a function that takes the `student_id` and `major` and returns a `Student` struct initialized with these values. The number of subjects taken should be initialized to `0` by default. Assume that the provided `major` is properly sized and null-terminated. `subjects` may remain uninitialized.

```
#include <string.h>

struct Student createStudent(const uint32_t student_id, const char* major)
{
  struct Student new_student; // Create a new student instance
  // YOUR CODE BELOW


    new_student.student_id = student_id;
    strcpy(new_student.major, major);
    new_student.num_subjects = 0;









  return new_student;
}
```

Problem continued on next page.

B. (2 points) As described in the `Subject` struct definition above, the `subject_name` member in `Subject` is in the format `"6.1904: Introduction to Low-level Programming in C and Assembly"`. Given a C-string `subject_name` and an output buffer `out`, we would like to extract the major and put it in the output buffer. The major is defined as everything before the first period `'.'`. For example, for the above, `out` should be the C-string `"6"`. Assume `subject_name` contains at least one `'.'` and `out` has enough space to store the extracted major and the null terminator. `subject_name` **may not** be modified. Write this function below.

```
#include <string.h>

// Copies major to variable out
void copyMajor(const char *subject_name, char *out) {
  // YOUR CODE BELOW
    int i = 0;
    while (subject_name[i] != '.') {
        out[i] = subject_name[i];
        i++;
    }
    out[i] = '\0';


}
```

C. (3 points) A student would like to add a new subject to their record and update their major to match the major prefix of that subject. For example, if the added subject has `subject_name` `"6.1904: Introduction to Low-level Programming in C and Assembly"`, then the student's major should be updated to `"6"`. `num_subjects` should also be updated appropriately. Write the function below. You can assume that the student has taken fewer than 100 classes. **You should use `copyMajor` from Part B.**

```
// Adds new subject to student and updates student's major
void addSubjectAndUpdateMajor(struct Student *student, struct Subject
*subject) {
  // YOUR CODE BELOW
    student->subjects[student->num_subjects] = *subject;
    student->num_subjects++;
    copyMajor(subject->subject_name, student->major);




}
```

D. (6 points) We would now like to create a function that checks whether a student has taken enough subjects to graduate. To graduate, the following conditions need to be satisfied:

- Must take at least 360 units total
- Must take at least 120 units from your major (so a major `"6"` student must have taken at least 120 units from `"6.XXXX"` classes)
- Must have taken at least 40 units outside of your major

To be precise, a subject counts toward the major if the substring before the first `'.'` in `subject_name` matches the student's `major` member exactly. Write a function that returns 1 if a student can graduate based on the classes taken so far, and 0 otherwise. **You should use `copyMajor` from Part B.**

```
#include <string.h>

int canGraduate(struct Student *student) {
  // YOUR CODE BELOW
    uint32_t total_units = 0;
    uint32_t major_units = 0;
    uint32_t non_major_units = 0;
    char subject_major[20]; //should minimally only need 4 here as per problem spec
    for (uint32_t i = 0; i < student->num_subjects; i++) {
        uint32_t units = student->subjects[i].num_units;
        total_units += units;
        copyMajor(student->subjects[i].subject_name, subject_major);
        if (strcmp(subject_major, student->major) == 0) {
            major_units += units;
        } else {
            non_major_units += units;
        }
    }
    if (total_units >= 360 && major_units >= 120
                           && non_major_units >= 40) {
        return 1;
    }
    return 0;
}
```

Problem 4: RISC-V Re-Vise (14 points)

A. (8 points) Convert the following C expressions into equivalent RV32I (32-bit RISC-V integer) assembly language instructions. Full credit will be given for correct solutions that use at most **three** RV32I instructions. Note: you may **not** use pseudo-instructions in this part.

Assume the following initial declarations:

```
int x[10];
int *y;
int z;
int out;
```

Assume `a0` holds the base address of `x`, `a1` holds the pointer `y`, and `a2` holds integer `z`. For any expression assigning to `out`, place the final value in `a3`.

Registers `t0-t6` are available for storing temporary values.

`out = x[3] + (*y);`	`lw t0, 12(a0)`
	`lw t1, 0(a1)`
	`add a3, t0, t1`

`*y = z + *(y+2);`	`lw t0, 8(a1)`
	`add t1, a2, t0`
	`sw t1, 0(a1)`

`*y = 5 * z;`	`slli t0, a2, 2`
	`add t0, t0, a2`
	`sw t0, 0(a1)`

`out = 0x47654321 + z;`	`lui t0, 0x47654`
	`addi t0, t0, 0x321`
	`add a3, t0, a2`

B. (6 points) A friend tries to translate the following assembly procedure into an equivalent C function but makes a couple of mistakes. Identify and correct your friend's **two buggy lines** of C code translation?

```
f:  li t0, 0            # i = 0
loop:
    bge t0, a1, done    # if i >= length goto done
    slli t1, t0, 2      # t1 = i * 4
    add t2, a0, t1      # t2 = addr of array[i]
    lw t3, 0(t2)        # t3 = array[i]
    andi t4, t0, 1
    bne t4, zero, label1
    slli t5, t3, 1      # i is even
    or t3, t3, t5       # t5 = 2 * array[i]
    sw t3, 0(t2)        # array[i] |= 2 * array[i]
    j next
label1:
    andi t5, t0, 0x1F   # t5 = shift = i % 32
    addi t6, t0, -1     # t6 = i - 1
    slli t6, t6, 2      # t6 = (i - 1) * 4
    add t6, a0, t6      # t6 = addr of array[i-1]
    lw t6, 0(t6)        # t6 = array[i-1]
    sll t6, t6, t5      # temp = array[i-1] << shift
    add t6, t3, t6      # t6 = array[i] + temp
    and t3, t3, t6      # array[i] &= (array[i] + temp)...continued...
    sw t3, 0(t2)        # array[i] &= (array[i] + temp)
next:
    addi t0, t0, 1      # i++
    j loop
done:
    ret
```

// Attempted C translation
// HINT: There are 2 incorrect lines
`void f(int *array, int length) {`
	For each incorrect line of the function, write the correct line of C code in its corresponding blank box below:

`for (int i = 0; i < length; i++) {`	
`if (length % 2 == 0) {`	**`if (i % 2 == 0) {`**
`array[i] |= 2 * array[i];`	
`} else {`	
`int shift = i%32;`	
`int temp = array[i] << shift;`	**`int temp = array[i-1] << shift;`**
`array[i] &= array[i] + temp;`	
`}` `}` `}`	

Problem 5. Stack Stroll (14 Points)

Your friend learned about **1-dimensional** random walks in class and wants to model them using C and RISC-V assembly. The function `randomWalk` takes in the walker's current position, target destination, and current number of steps. On each call, the function randomly takes a step left (negative) or right (positive) and keeps going until the position reaches the destination or the number of steps hits the recursion limit. If the step limit is hit, `randomWalk` will return `-1`, otherwise it will return the number of steps needed to reach the target destination.

Below is a C implementation of `randomWalk`. The code makes use of a memory address named `RANDOM_VALUE_ADDR` (address `0x100`) that generates random values (used in a similar way to generating random food locations in the Snake lab from class).

An equivalent RISC-V Assembly implementation is provided on the following page.

```
1   int randomWalk(int position, int target, int steps) {
2     if (steps == LIMIT) {
3       return -1;
4     } else if (position == target) {
5       return steps;
6     }
7
8     int rand = *((int*) RANDOM_VALUE_ADDR); // Generate random int
9
10    if (rand >= 0) {      // Move right
11      return randomWalk(position + 1, target, steps + 1);
12    } else {              // Move left
13      return randomWalk(position - 1, target, steps + 1);
14    }
15  }
```

```
1   randomWalk:
2       addi sp, sp, -8
3       sw ra, 0(sp)
4       sw a0, 4(sp)
5       li t0, LIMIT # Secret value
6       blt a2, t0, underLimit
7       addi sp, sp, 8
8       li a0, -1
9       jalr x0, 0(ra)
10  underLimit:
11      bne a0, a1, notAtTarget
12      addi sp, sp, 8
13      addi a0, a2, 0
14      jalr x0, 0(ra)
15  notAtTarget:
16      lw t0, 0x100(x0) # RANDOM_VALUE_ADDR=0x100
17      addi a2, a2, 1
18      blt t0, x0, moveLeft
19      addi a0, a0, 1
20      jal ra, randomWalk
21      jal x0, end
22  moveLeft:
23      addi a0, a0, -1
24      jal ra, randomWalk
25  end:
26      lw ra, 0(sp)
27      addi sp, sp, 8
28      jalr x0, 0(ra)
```

Problem continued on next page.

`randomWalk` is initially called with `steps=0`. Execution was halted somewhere during a recursive call to `randomWalk`. After halting, the state of program execution is printed below, which includes a relevant portion of the stack, the current value of the program counter (`pc`), and the current value of register `a1`:

pc = 0x00002044	a1 = 0x0000000F

Address	Data
...	...
0x00002FC8	0xDEEEAAAD
0x00002FCC	0xBEEEEEEF
0x00002FD0	0xDEEEAAAD
0x00002FD4	0xBEEEEEEF
0x00002FD8	0x00002060
0x00002FDC	0x0000000D
0x00002FE0	0x????????
0x00002FE4	0x????????
0x00002FE8	0x????????
0x00002FEC	0x????????
0x00002FF0	0x0000206C
0x00002FF4	0x0000000D
0x00002FF8	0x00002060
0x00002FFC	0x0000000E
0x00003000	0x00000F28
0x00003004	0x0000000D
...	...

A. (2 Points) What is the address of the call instruction that made the initial call to `randomWalk`?

`0x00000F24`

B. (2 Points) What instruction is the program counter (`pc`) pointing to when execution is halted?

`addi a0, a2, 0`

C. (2 Points) Does the call to `randomWalk` reach its target or hit the recursion limit? Circle one option:

REACHES TARGET / **HITS LIMIT** / **CAN'T TELL**

- If **REACHES TARGET**, how many steps did it take (i.e. what is the return value of the initial call)?
- If **HITS LIMIT**, what is the value of `LIMIT`?
- If **CAN'T TELL**, briefly explain why.

4

Problem continued on next page.

D. (6 Points) A copy of the stack diagram is provided below. Fill in the four blank entries. If the value cannot be determined, then fill in the box with **CAN'T TELL.**

Address	Data
...	...
0x00002FC8	0xDEEEAAAD
0x00002FCC	0xBEEEEEEF
0x00002FD0	0xDEEEAAAD
0x00002FD4	0xBEEEEEEF
0x00002FD8	0x00002060
0x00002FDC	0x0000000D
0x00002FE0	**0x00002060**
0x00002FE4	**0x0000000F**
0x00002FE8	**0x00002060**
0x00002FEC	**0x0000000E**
0x00002FF0	0x0000206C
0x00002FF4	0x0000000D
0x00002FF8	0x00002060
0x00002FFC	0x0000000E
0x00003000	0x00000F28
0x00003004	0x0000000D
...	...

E. (2 Points) What is the value of the stack pointer (`sp`) when execution is halted?

0x**00002FE8**

This page intentionally left blank

Problem 6. RISC-V Assembly (11 points)

A. (2 points) Tim does not like pseudoinstructions. He wants to write a procedure `foo` that returns the value `0x1704` **without using pseudoinstructions**. Help Tim write the instruction(s) to do this.

```
foo:

    lui a0, 0x1
    addi a0, a0, 0x704

    jalr x0, 0(ra)
```

B. (3 points) Next, he wants to write a procedure `bar` that returns the value `0x1904` **without using pseudoinstructions**. Help Tim write the instruction(s) to do this.

```
bar:

    lui a0, 0x1904
    srli a0, a0, 12

    #Alternative solution (with addi):
    lui a0, 0x2
    addi a0, a0, -0x6FC

    jalr x0, 0(ra)
```

C. (3 points) What is the 32-bit RISC-V instruction encoding of `sw s2, -8(a1)`? Write your final answer in hexadecimal

0xFF25AC23

D. (3 points) The assembly code below is run to completion of the `sw` instruction. Fill in the blanks.

```
. = 0x100
    li a1, 0x123
    li a2, 1
    slli a5, a2, 2
    slli a4, a2, 3
    lw a6, 0x500(a4)
    or a6, a6, a5
    sw a6, 0x500(a5)


. = 0x500
.word 0xDEADBEEF
.word 0x67676767
.word 0x12345678
.word 0x24446666
```

Register/Memory Address	Final value in hex (`0x`)
`MEM[0x500]=`	`0x`**DEADBEEF**
`MEM[0x504]=`	`0x`**1234567C**
`MEM[0x508]=`	`0x`**12345678**
`MEM[0x50C]=`	`0x`**24446666**

Problem 7: Some Helpful Pointers (11 points)

A. (3 points) Consider the following code:

```
int arr[] = {7, 8, 4, 9, 3, 5, 6, 2, 1, 0};
int *s = arr;

int i = __START__;
int count = 0;
while(*(s + i) != 0) {
    i = *(s + i);
    count++;
}

printf("%d", count);
```

For each of the following values of `__START__`, what gets printed when the above code is executed? Note that the `while` loop might run forever; in that case, write "INFINITE LOOP" instead.

Value of `__START__`	Printed Output
0	5
1	INFINIE LOOP
2	3

B. (2 points) The following snippet of code is executed and prints "`0xED767BC8`".

```
int val = 0x2123252B;
int *n = &val;
char *c = (char*) n;
printf("0x%x", n);       // print n in hex format
```

Assuming the same 32-bit ESP32 system we've been working with, **complete the table below**:

Print Statement	Printed Output
`printf("0x%x", n + 4);`	0xED767BD8
`printf("0x%x", c + 4);`	0xED767BCC

C. (6 points) Consider the following code:

```
char box[3][4] = {
    {'a', 'b', 'c', 'd'},
    {'l', 'm', 'n', 'o'},
    {'w', 'x', 'y', 'z'}
};
char *p = &box[0][0];
char **q = &p;
```

Recall that the `sizeof` operator provides the size of a variable or data type in **bytes**. Assuming the same 32-bit ESP32 system we've been working with, **complete the table below**:

Expression	**Value**
`sizeof(box)`	**12**
`sizeof(*q)`	**4**
`sizeof(**q)`	**1**
`sizeof(&q)`	**4**

Can you use the variable `p` to modify the character in `box[0][2]`?
If so, write the code to modify it to be `'G'`. If not, explain why not.

Yes. `*(p + 2) = 'G';`

Problem 8. RISC-V Re-CALL (13 points)

We can define the parity of an integer as whether its binary representation has an even or odd number of bits that are 1. If the number of bits in the integer that are 1 is odd, the parity of the integer is 1. If the number of bits in the integer that are 1 is even, the parity of the integer is 0. The integer 4 would have a parity of 1, and the integer 5 would have a parity of 0, for example.

A. (4 points) We've implemented a function called `computeIntParity` that computes and returns an input integer's parity. The assembly reflects the intended algorithm, but it does not properly follow the RISC-V calling convention.

The C function and RISC-V procedure implementations are provided below and on the next page. Identify and fix any errors in the assembly code translation on the next page. You are **only** allowed to change register usage.

```
uint32_t computeIntParity (uint32_t x) {
    uint32_t parity = 0;
    while (x!=0){
        if (x&1) {
            parity = !parity;
        }
        x = x >> 1;
    }
    return parity;
}
```

Re-write the code below such that `computeIntParity` correctly follows the calling convention and behaves as expected. Again, you may **only** modify the register usage in doing so.

For **full credit**, you should only modify registers with usage violating the calling convention. Fill in only the registers that need to change; unchanged registers may be omitted.

Original	Rewritten
`computeIntParity:`	`computeIntParity:`
`    li sp, 0`	`    li _t2_, 0`
`    li t0, 1`	`    li _____, 1`
`while:`	`while:`
`    beqz a0, endwhile`	`    beqz _____, endwhile`
`    andi t1, a0, 1`	`    andi _____, _____, 1`
`    blt t1, t0, afterif`	`    blt  _____, _____, afterif`
`    xori sp, sp, 1`	`    xori _t2_, _t2_, 1`
`afterif:`	`afterif:`
`    srli a0, a0, 1`	`    srli _____, _____, 1`
`    jal ra, while`	`    jal  _x0__, while`
`endwhile:`	`endwhile:`
`    mv a0, sp`	`    mv   _____, _t2_`
`    jalr a0, 0(ra)`	`    jalr _x0_, 0(_____)`

B. (9 points) Provided below is the function `arrayParityCheck` that takes in three inputs: two integer array pointers and these arrays' corresponding length (you can assume both arrays will always have the same length, and not be empty). `arrayParityCheck` populates `parity_array` and returns an integer representing a metric called `global_parity`.

A working C implementation is provided below for reference:

```
// Computes and returns the global parity of an array
// Also populates parity_array with the parity of each int_array entry

uint32_t arrayParityCheck (uint32_t *int_array, uint32_t *parity_array,
uint32_t array_length) {

    uint32_t global_parity = 0;
    for (int i = 0; i<array_length; i++) {
        uint32_t val    = int_array[i];
        uint32_t parity = computeIntParity(val);
        parity_array[i] = parity;
        global_parity   ^= parity;
    }
    return global_parity;
}
```

The RISC-V assembly implementation of the function is provided below. Again, while the function's implementation is a correct translation of the C function, it does not adhere to the RISC-V calling convention. You may assume that `array_length` will always be greater than 0.

Your job will be to add instructions to the code such that `arrayParityCheck` adheres to the calling convention. ***However***, you may ***only***:

- increment/decrement the stack pointer
- load word(s) from stack
- save word(s) to stack.

You may only write **1 instruction per line**. It is possible that not all lines will need to be filled.
For full credit, your answer should ***minimize*** the total number of instructions (loads and stores to the stack, stack pointer manipulations).

You ***may*** assume that the implementation for `computeIntParity` properly follows RISC-V calling convention and has the same functionality as part A.
You **SHOULD NOT** assume that `computeIntParity`'s implementation exactly matches part A's.

We have provided a non-RISC-V adhering version of `arrayParityCheck` here for your convenience:

```
arrayParityCheck:
    mv s0, a2 # s0 = array_length
    addi a3, x0, 0 # i in a3
    addi a4, x0, 0 # global_parity in a4

    loopBegin:

        slli s1, a3, 2 # s1 = i*4
        add a6, s1, a0 # a6 = &int_array[i]
        lw a0, 0(a6) # a0 = int_array[i]

        call computeIntParity # outputs parity in a0

        xor a4, a4, a0 # global_parity ^= output of computeIntParity

        add a7, s1, a1 # a7 = &parity_array[i] = i*4 + a1
        sw a0, 0(a7) # store output parity at a7 address

        addi a3, a3, 1 # i++
        blt a3, s0, loopBegin # loop again
    end:

    mv a0, a4

    jalr x0, 0(ra)
```

This page intentionally left blank

Fill in the blanks in the *implementation below* to receive credit:

```
arrayParityCheck:

    ________________________________

    addi sp, sp, -28      # allocate stack space

    sw   ra, 0(sp)        # save ra

    sw   s0, 4(sp)        # save s0

    sw   s1, 8(sp)        # save s1

    sw   a0, 12(sp)       # save a0

    sw   a1, 16(sp)       # save a1

    ________________________________

    ________________________________

    mv s0, a2 # s0 = array_length
    addi a3, x0, 0 # i in a3
    addi a4, x0, 0 # global_parity in a4

    loopBegin:
    ________________________________

    ________________________________

    ________________________________

    ________________________________

       slli s1, a3, 2 # s1 = i*4
       add a6, s1, a0 # a6 = &int_array[i]
       lw a0, 0(a6) # a0 = int_array[i]

    ________________________________

       sw a3, 20(sp)

       sw a4, 24(sp)
```

```
______________________________

______________________________

______________________________

______________________________

 call computeIntParity # outputs parity in a0

 lw a3, 20(sp)

 lw a4, 24(sp)

 lw a1, 16(sp)

______________________________

______________________________

______________________________

______________________________

  xor a4, a4, a0 # global_parity ^= output of computeIntParity
  add a7, s1, a1 # a7 = i*4 + a1
  sw a0, 0(a7) # store output parity at a7 address

  lw a0, 12(sp)

______________________________

______________________________

______________________________

   addi a3, a3, 1 # i++
   blt a3, s0, loopBegin # loop again

end:

mv a0, a4
```

```
________________________________

________________________________

________________________________

lw   ra, 0(sp)

lw   s0, 4(sp)

lw   s1, 8(sp)

addi sp, sp, 28

________________________________

________________________________

jalr x0, 0(ra)
```