MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.1903: Introduction to Low-level Programming in C and Assembly

Spring 2025, Quarter 3

Name: Solutions	**Kerberos:** solutions
	MIT ID #: solutions

#1 (11)	11
#2 (14)	14
#3 (7)	7
#4 (15)	15
#5 (13)	13
#6 (14)	14
#7 (14)	14
#8 (12)	12
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. Did You Get Recitation Credit? (11 points)

The 6.1903 TAs are devising a new way for the Institute to encode recitation attendance using 32-bit binary encodings. (Course 6 was suddenly granted enough funding so that recitation section sizes are capped at 32.) Student X's attendance is encoded in bit X (from the right), where 1 means present and 0 means absent. Any excess bits are left at zero (for enrollments less than 32). For instance, if the recitation has 10 students and everyone is present except Student 3, then this class's attendance can be represented as:

```
0b0000_0000_0000_0000_0000_0011_1111_0111
```

All the attendance encodings for a recitation section `A` with `N` expected weeks are defined using the following data structure:

```
int rec_A[N] = {0};
```

6.1903's Recitation section `A` has 20 students and expects 5 weeks throughout the term. It has already had its first two sessions (the zeroth and first week). Its encodings are stored in array `rec_A[5].` Assume that the roster never changes.

A. (4 points) For the scenarios below, write an equivalent C expression using only:

- **Bitwise or Logical Operators**
- **Constants**
- **Variables (`rec_A`)**

Generate a 32-bit encoding that sets the X-th bit to 1 if student X attended both weeks 0 and 1, and 0 otherwise.	`rec_A[0] & rec_A[1]`
Generate a boolean value that is 1 if student 6 attended the zeroth week, and 0 otherwise.	`rec_A[0] >> 6 & 1`

B. (3 points) For the second week, students 0, 2, 6, 7, and 10 attended. You print out the recitation attendance with the `%d` format. Encode the attendance with both 32-bit binary and hexadecimal representation, then evaluate what you would print. *You must indicate all 32 bits.*

32-bit binary (`0b`):   **0b0000_0000_0000_0000_0000_0100_1100_0101**
32-bit hexadecimal (`0x`):   **0x000004C5**
`printf("%d", rec_A[2]);` gives: **1221**

C. (4 points) Write a function, `setAttendance`, which will set Student X's attendance for `week_num`-th class. For example:

```
setAttendance(rec_A, 9, 0, 1);  // marks student 9 present for the 0th
                                 // week of recitation rec_A
setAttendance(rec_B, 15, 3, 0); // marks student 15 absent for the 3rd
                                 // week of recitation rec_B
```

```
void setAttendance(int * rec_X, uint8_t student_index,
                   uint32_t week_num, uint8_t present) {

    if (present) {
        rec_X[week_num] |= 1 << student_index;

    } else {
        rec_X[week_num] &= ~(1 << student_index);

    }

    // OR

    rec_X[week_num] &= ~(1 << student_index);
    rec_X[week_num] |= present << student_index;

}
```

Problem 2: RISC-V Riddles (14 points)

A. (8 points) Given the following initial values for registers `x0-x7`, determine the resulting hexadecimal value for each of the registers specified.

x0 = 0x0	x1 = 0x104	x2 = 0x5A	x3 = 0x333
x4 = 0x44	x5 = 0x4	x6 = 0x66	x7 = 0xABCD

```
  li x0, 3
  addi x1, x1, 0x258
  ori x2, x2, 0x31
  lui x3, 0x34
  ori x4, x4, 0x800
  slli x5, x5, 1
  lw x6, 0x63C(x5)
  xori x7, x7, -1
  j end

. = 0x640
.word 0x55335533
.word 0x11223344
.word 0xFFAAFF00
.word 0x87654321

end:
```

# Results of running assembly code	What is the value of each of the following registers after running the code snippet above?
x0 =	0x0
x1 =	0x35C
x2 =	0x7B
x3 =	0x34000
x4 =	0xFFFFF844
x5 =	0x8
x6 =	0x11223344
x7 =	0xFFFF5432

B. (6 points) One of your friends tries to translate the following snippet of C code into RISC-V assembly. However, they need your help to get it right. Finish the translation of the following C code into assembly.

```
// Original C code
// arr is an int array
for (int i = 0; i < 4; i++) {
        arr[i] &= i;
    }
}
```

```
# Complete the assembly code below so that it implements the C for loop above
```

```
   mv a1, zero         # i
   addi a2, zero, 4
   li a3, 0x500        # array arr begins at address 0x500
loop:
   bge a1, a2, end
   # Your code here
   slli a4, a1, 2      # i*4
   add a5, a3, a4      # address of arr[i]
   lw a6, 0(a5)
   and a6, a6, a1
   sw a6, 0(a5)
   addi a1, a1, 1
   j loop
end:
```

Problem 3. Leftovers (7 points)

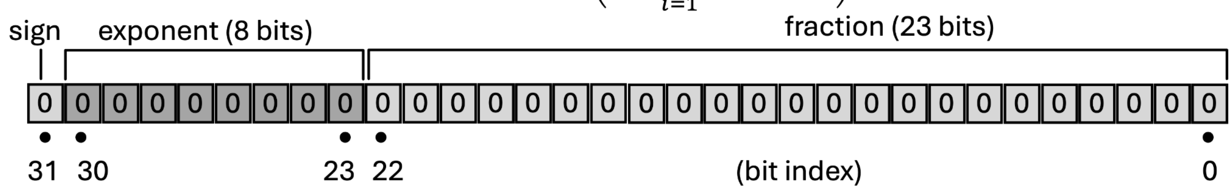
A. (2 points) Write the value of the `result` as a decimal value. Assume each cell is executed independently. Pay attention to order of operations!

```
int8_t alpha = 0b00001110;

int8_t result = -128 == 3 << alpha / 2 | 2 ^ 4;
```

6

B. (5 points) Tim the Beaver wants to represent Pi as a floating point number using IEEE 754 standard.

$$\text{Value} = (-1)^{\text{sign}} \cdot 2^{\text{exp}-127} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} \cdot 2^{-i}\right)$$



Tim approximates as: $\pi \approx 3 + 2^{-3} + 2^{-6} + 2^{-10} \approx 3.1416$. **Determine the floating point representation of Tim's approximation of Pi. Provide your answer in hexadecimal. Show your work.**

= 0b0_10000000_10010010001000000000000

= 0x40491000

Problem 4. (15 points) Array Maze Architect

For a future 6.1903 lab, students will be developing a maze game that they will be able to navigate using the buttons on their control board. With your advanced knowledge of the C programming language and pointer arithmetic, you have been tasked with creating an advanced maze generation algorithm for the lab.

A. (5 points) You structure your maze as a 1D array of `uint8_t` integers, with "1"s representing walls and "0"s representing paths. For this problem, assume the following global declarations:

```
#define ROWS 7
#define COLS 6

uint8_t maze[ROWS*COLS] = {
    1, 1, 1, 1, 1, 1,   // row 0
    1, 1, 1, 1, 1, 1,   // row 1
    1, 1, 1, 1, 1, 1,   // row 2
    1, 1, 1, 1, 1, 1,   // row 3
    1, 1, 1, 1, 1, 1,   // row 4
    1, 1, 1, 1, 1, 1,   // row 5
    1, 1, 1, 1, 1, 1,   // row 6
};
```

Recall that the `sizeof` operation 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:**

Operation	Result
`sizeof(maze)`	42 bytes
`sizeof(&maze)`	4 bytes
`sizeof(&maze[2])`	4 bytes
`sizeof(maze[0])`	1 byte
`sizeof(maze[8])`	1 byte

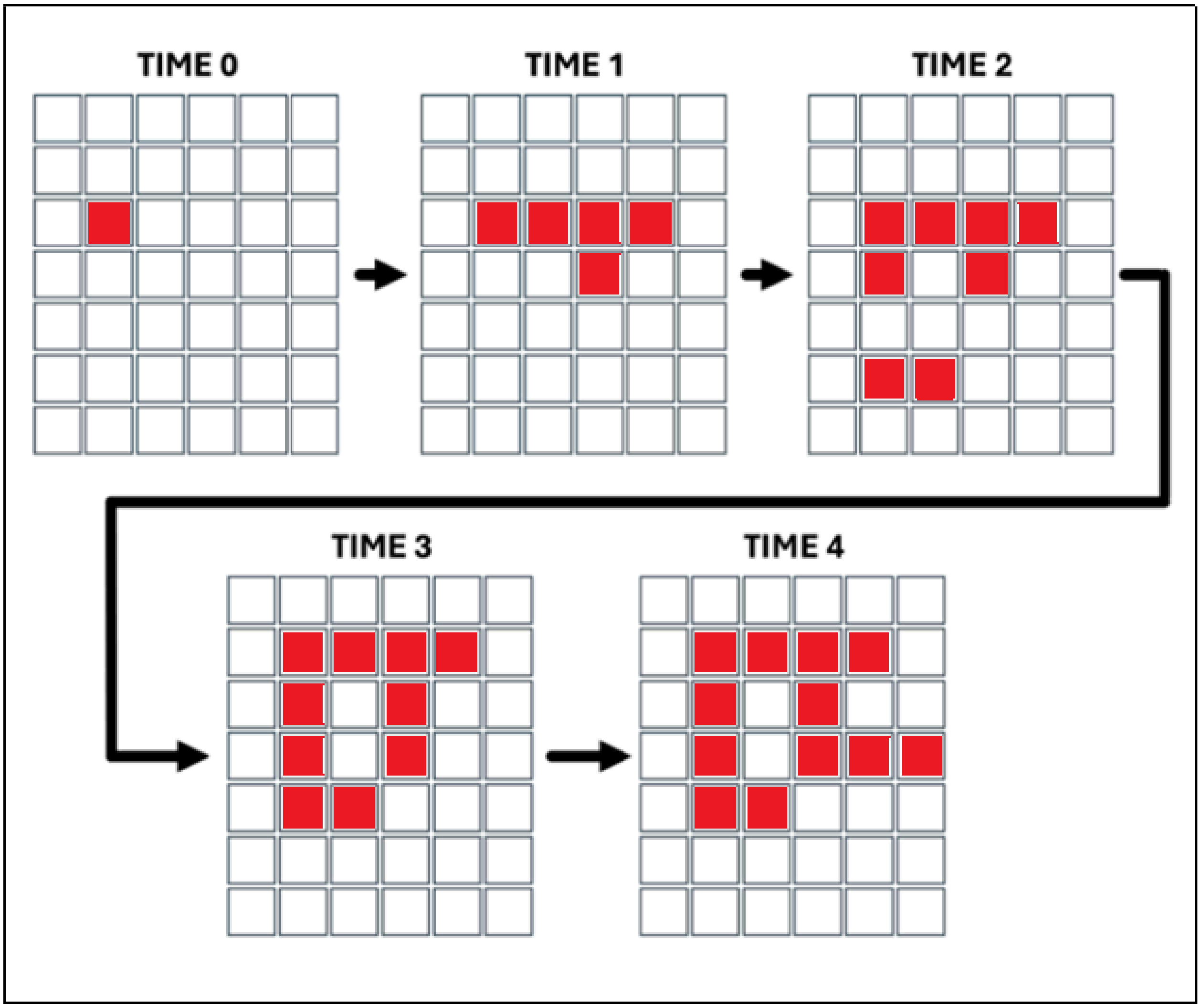
B. (7 points) After declaring an initial maze array with only walls, you develop a function `mazeSculptor` to carve out the maze's tunnels. Your function takes in a start position pointer pointing somewhere within the maze array.

```
void mazeSculptor(uint8_t* start) {
    uint8_t* maze_pointer = start;
    *(maze_pointer) = 0;

    // TIME 0

    for (int i = 1; i < COLS-2; i++) {
        *(maze_pointer + i) = 0;
    }
    *(maze_pointer + 8) = 0;

    // TIME 1

    maze_pointer = start + COLS;
    *(maze_pointer) = 0;
    *(maze_pointer + 2*COLS) = 0;
    *(maze_pointer + 2*COLS + 1) = 0;

    // TIME 2

    maze_pointer = start + 2*COLS;
    for (int i = 0; i < COLS; i++) {
        *(maze_pointer + i) = *(start + COLS + i);
    }

    // TIME 3

    maze_pointer[3] = 0;
    maze_pointer[4] = 0;

    // TIME 4

    printf("Maze generated successfully!\n");
}
```

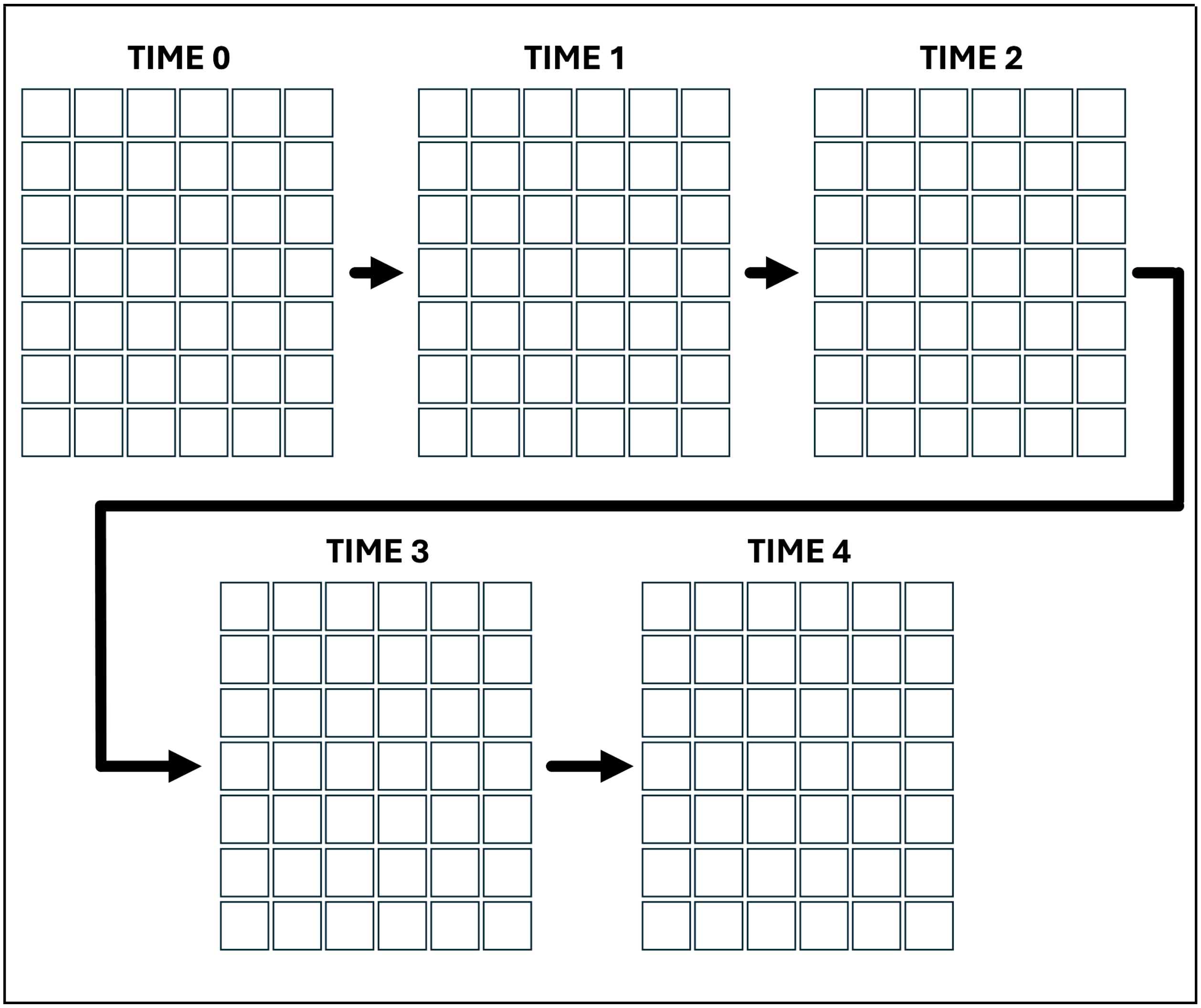
Suppose you call the mazeSculptor function in the following way:

```
mazeSculptor(maze + 2*COLS + 1)
```

At every state indicated by comments in the function above, **clearly write a '0' in every cell in the maze that would contain a path at that TIME.** An extra copy of this figure is provided on the following page in case you make a mistake. Clearly mark which one should be graded.



Here is an extra copy of the figure on the previous page in case you make a mistake and want to start over. Clearly mark which one should be graded.



C. (2 points) You call your `mazeSculptor` function again but this time with a different start position. **Describe at least two potential outcomes of making the following function call:**

```
mazeSculptor(maze + 6*COLS + 7);
```

You will be accessing memory outside of the maze array. This can have many potential outcomes:

- **Segmentation fault**
- **Memory corruption**
- **No errors → "Maze generated successfully!" printed**

D. (1 point) You decide to call `mazeSculptor` in an alternative way:

```
mazeSculptor(maze[2*COLS+1]);
```

Is this equivalent to our original command in Part B, `mazeSculptor(maze + 2*COLS + 1)`**? Explain why / why not.**

No, this will throw a type error because you are passing an integer instead of a pointer here

This page intentionally left blank

Problem 5. Call Me Sometime (13 points)

The function `fraction2float` converts its numerator and denominator arguments into their floating-point representation. Its C implementation is shown below, followed by its RISC-V implementation on the next page.

Reference C implementation:

```
float fraction2float(int numerator, int denominator){

  // determine sign bit of floating-point output
  int sign = 0;
  if (numerator < 0) {
      sign = sign ^ 1;
      numerator = -numerator;
  }
  if (denominator < 0) {
      sign = sign ^ 1;
      denominator = -denominator;
  }

  // call sub-functions to combine output
  int exponent = getFloatExponent(numerator, denominator);
  int mantissa = getFloatMantissa(numerator, denominator, exponent);


  // build 32-bit output using sign, exponent, and mantissa
  int output = sign << 31;
  output = output | (exponent << 23);
  output = output | mantissa;

  return output;

}
```

The equivalent assembly procedure on the next page, `fraction2float`, calls two helper functions: `getFloatExponent` and `getFloatMantissa`. Currently, `fraction2float` **does not** adhere to RISC-V calling convention. Assume the two helper functions do adhere to calling convention.

```
# fraction2float
  # ARGUMENTS:
  #   a0: numerator
  #   a1: denominator

  # RETURNS: 32 bit floating point (IEEE 754) representation
  #          of the fraction specified by (numerator/denominator)

fraction2float:



  li s0, 0                  # sign = 0
  bge a0, zero, skip_numerator_flip
  xori s0, s0, 1            # sign = sign ^ 1
  sub a0, zero, a0          # numerator = -numerator
skip_numerator_flip:
  bge a1, zero, skip_denominator_flip
  xori s0, s0, 1            # sign = sign ^ 1
  sub a1, zero, a1          # denominator = -denominator
skip_denominator_flip:



  call getFloatExponent
  mv s1, a0                 # save exponent in s1



  mv a2, s1
  call getFloatMantissa
  mv s2, a0                 # save mantissa in s2



  slli a0, s0, 31
  slli t0, s1, 23
  or a0, a0, t0
  or a0, a0, s2



  ret
```

A. (2 points) As the function is currently written (with no corrections to calling convention), what instruction will be executed immediately after the final `ret` statement?

```
mv s2, a0     # save mantissa in s2
```

B. (2 points) As the function is currently written (with no corrections to calling convention), list the caller-saved and callee-saved registers used anywhere in the function `fraction2float`. (Include any registers used by pseudoinstructions).

Caller-saved	Callee-saved
ra a0 a1 a2 t0	s0 s1 s2

C. (9 points) In the assembly code box below, add in the necessary assembly code to utilize the stack and make the function `fraction2float` adhere to RISC-V calling convention. Do not modify the function behavior, only add lines that ensure calling convention is followed. You should only use 3 types of instructions:

- Modifications to the stack pointer,
- `lw` instructions to read from the stack,
- `sw` instructions to write to the stack.

```
  # fraction2float
  # ARGUMENTS:
  #   a0: numerator
  #   a1: denominator

  # RETURNS: 32 bit floating point (IEEE 754) representation
  #          of the fraction specified by (numerator/denominator)

fraction2float:

  # one of multiple solutions
  addi sp, sp, -24
  sw ra, 0(sp)
  sw s0, 4(sp)
  sw s1, 8(sp)
  sw s2, 12(sp)

  li s0, 0                  # sign = 0
  bge a0, zero, skip_numerator_flip
  xori s0, s0, 1            # sign = sign ^ 1
  sub a0, zero, a0          # numerator = -numerator
skip_numerator_flip:
  bge a1, zero, skip_denominator_flip
  xori s0, s0, 1            # sign = sign ^ 1
  sub a1, zero, a1          # denominator = -denominator
skip_denominator_flip:



  sw a0, 16(sp)
  sw a1, 20(sp)


  call getFloatExponent
  mv s1, a0                 # save exponent in s1
```

```
lw a0, 16(sp)
lw a1, 20(sp)



mv a2, s1
call getFloatMantissa
mv s2, a0                # save mantissa in s2









slli a0, s0, 31
slli t0, s1, 23
or a0, a0, t0
or a0, a0, s2




lw ra, 0(sp)
lw s0, 4(sp)
lw s1, 8(sp)
lw s2, 12(sp)
addi sp, sp, 24






ret
```

Problem 6. (14 points) The Bitville Library

The Bitville Library is undergoing renovations and has hired you to help digitize its book management system. They have defined several C structs and functions, which you will work with in this problem. A struct called `Book` is defined below to represent a book in the library. Interestingly, in Bitville, there are only two book genres: fiction and nonfiction. **You may assume the library `string.h` has been included for all parts of this problem.**

```
struct Book {
  char title[100];        // Array to store the book title.
                          // Contains only ASCII characters.
  char author[50];        // Array to store the author's name.
                          // Contains only ASCII characters.
  uint32_t pub_year;      // Year the book was published.
  bool is_fiction;        // True if the book is fiction, false if it's
                          // non-fiction.
  bool is_checked_out;    // True if the book is currently checked out,
                          // false otherwise
};
```

A. (5 points) Write a function called **`createBook`** that takes the title, author, publication year, and fiction flag as parameters and returns a `Book` struct initialized with these values. Take careful notice of the parameters. *Assume that `is_checked_out` is `false` by default and that the provided `title` and `author` strings are properly sized and null-terminated. Assume that the title and author string will contain only ASCII characters.*

```
struct Book createBook(const char *title, const char *author,
                        uint32_t pub_year, bool is_fiction){
      struct Book new_book; // Create a new book instance
      // YOUR CODE BELOW

      // Set title and author fields:
      // (strcpy is safe to use per the assumption that title and
      // author are properly sized and null-terminated)
      strcpy(new_book.title, title);
      strcpy(new_book.author, author);

      // Set publication year and fiction status from parameters:
      new_book.pub_year = pub_year;
      new_book.is_fiction = is_fiction;

      // Set initial checkout status to available:
      new_book.is_checked_out = false;

      return new_book;
}
```

B. (4 points) Before adding new books to the library, Bitville librarians require that each book's title contain only lowercase letters (a-z) and spaces. Write a function called **`validateTitle`** that validates this condition and returns true if valid and false if otherwise. *Like in part A, assume that the `title` string is properly null-terminated and contains only ASCII characters.*

```
// Returns true if the Book's title contains only lowercase letters (a-z)
// and the space character, false otherwise.
bool validateTitle(struct Book *book) {
      // YOUR CODE BELOW

      // Get pointer to book title string
      char *title = book->title;

      // Iterate through each character until null terminator is reached
      for (int i = 0; title[i] != '\0'; i++) {
            char c = title[i];

            // Check if character is lowercase letter (a-z) or space
            // (Comparing against int equivalent is fine too: 'a' = 97,
            // 'z' = 122, ' ' = 32)
            bool is_lowercase = (c >= 'a' && c <= 'z');
            bool is_space = (c == ' ');

            // Return false if character is not lowercase or space
            if (!is_lowercase && !is_space) {
                  return false;
            }
      }

      // All characters were valid, return true
      return true;

}
```

C. (5 points) A struct called **`Library`** is defined below to represent the library:

```
struct Library {
  Book **books;          // Array of pointers to Book structs representing
                         // the books in the library.
  uint32_t count;        // Current number of books in the library /
                         // current size of books array.
  uint32_t capacity;     // Max books allowed; full when count == capacity.
};
```

Before adding a book to the library, librarians ensure the following:

- The book meets the title validation criteria from Part B.
- No other book in the library has the same title.
- Adding the book does not exceed the library's capacity.

Write a function called **`addBook`** on the next page that verifies these conditions and, if all are true, adds the book to the books array. This function should return true if the book was successfully added and false otherwise. You may call your function from Part B as needed. *Assume that your function from Part B is fully functional. Assume that the `Library` struct has been properly initialized (with valid pointers), and that `count` is initially set to be less than `capacity`.*

For convenience, the `Book` struct definition is reproduced here:

```
struct Book {
  char title[100];         // Array to store the book title.
                           // Contains only ASCII characters.
  char author[50];         // Array to store the author's name.
                           // Contains only ASCII characters.
  uint32_t pub_year;       // Year the book was published.
  bool is_fiction;         // True if the book is fiction, false if it’s
                           // non-fiction.
  bool is_checked_out;     // True if the book is currently checked out,
                           // false otherwise
};
```

```c
bool addBook(struct Library *lib, struct Book *book) {
        // YOUR CODE BELOW

        // Note: The ordering below (1-3) can be done in any order.

        // (1) Check if library has space
        if (lib->count >= lib->capacity) { // (use of == is fine too)
              return false;
        }

        // (2) Check book title validity
        bool is_valid = validateTitle(book);
        if (!is_valid) {
              return false;
        }

        // (3) Check for duplicate title
        for (uint32_t i = 0; i < lib->count; i++) {
              if (strcmp(lib->books[i]->title, book->title) == 0) {
                    return false;
              }
        }

        // Add the book and update count
        lib->books[lib->count] = book;
        lib->count++;

        // Successfully added
        return true;

}
```

This page intentionally left blank

Problem 7: (14 points) Some Algebra Practice

Here is a C implementation of a function `linearConvergence` that determines if a series of operations on `input` leads to convergence to `0`. The function returns `depth` if the input is ever equal to `0` and `0` if the maximum recursive depth is exceeded.

```
int linearConvergence(int input, int alpha, int beta, int depth) {
    if (depth >= MAXIMUM_DEPTH) {
         return 0;
    }

    int shifted = input >> alpha;

    if (input == 0) {
        return depth;
    } else if (input > 0) {
        return linearConvergence(shifted - beta, alpha, beta, depth+1);
    } else {
        return linearConvergence(shifted + beta, alpha, beta, depth+1);
    }
}
```

In RISC-V, this function has the equivalent implementation shown on the following page. `stack` is a custom instruction that prints the stack trace and the value of stack pointer whenever it executes. It does not modify the state of the system (registers, memory etc. in any way).

```
linearConvergence:
    addi t0, zero, MAXIMUM_DEPTH
    blt a3, t0, body
    stack                   # Get stack trace
    addi a0, zero, 0
    jalr zero, 0(ra)
body:
    addi sp, sp, -8
    sw s0, 0(sp)
    sw ra, 4(sp)
    stack                   # Get stack trace
    sra s0, a0, a1
    blt a0, zero, negative
    blt zero, a0, positive
    _______________
    jal zero, end
negative:
    add a0, s0, a2
    addi a3, a3, 1
    jal ra, linearConvergence
    jal zero, end
positive:
    sub a0, s0, a2
    addi a3, a3, 1
    jal ra, linearConvergence
End:
    lw s0, 0(sp)
    lw ra, 4(sp)
    addi sp, sp, 8
    stack                   # Get stack trace
    jalr zero, 0(ra)
```

A. (2 points) Complete the line left blank in the code so that the assembly implementation matches the C implementation.

Line: **mv a0, a3 (or) addi a0, a3, 0 (or) add a0, a3, zero/x0**

We obtain the stack trace just before the first call to the function `linearConvergence` occurred (**`TIME 0`** in the table). Then we run `linearConvergence` to determine convergence. Some stack traces are produced as given in the table below. The last row of the table also gives you the value in the register `sp` at the corresponding time. Execution may not have reached completion by time point **`TIME 5.`**

Address	**TIME 0**	**TIME 1**	**TIME 2**	**TIME 3**	**TIME 4**	**TIME 5**
0x000802a0	0x000a05a0	0x000a05a0	0x000a05a0	0x000a05a0	0x00000003	0x00000003
0x000802a4	0x00000211	0x00000211	0x00000211	0x00000211	0x00000270	0x00000270
0x000802a8	0xfffffffe	0xfffffffe	0xfffffffe	0xfffffffe	0xfffffffe	0xfffffffe
0x000802ac	0x00000000	0x00000000	0x00000000	0x00000260	0x00000260	0x00000260
0x000802b0	0x8f000010	0x8f000010	0x0000000b	0x0000000b	0x0000000b	0x0000000b
0x000802b4	0x0000001a	0x0000001a	0x00000270	0x00000270	0x00000270	0x00000270
0x000802b8	0x00001986	0x00000019	0x00000019	0x00000019	0x00000019	0x00000019
0x000802bc	0x00001986	0x00000214	0x00000214	0x00000214	0x00000214	0x00000214
0x000802c0	0x00000a50	0x00000a50	0x00000a50	0x00000a50	0x00000a50	0x00000a50
0x000802c4	0xffffffff	0xffffffff	0xffffffff	0xffffffff	0xffffffff	0xffffffff
0x000802c8	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000802cc	0x00000214	0x00000214	0x00000214	0x00000214	0x00000214	0x00000214

sp	0x000802c0	0x000802b8	0x000802b0	0x000802a8	0x000802a0	0x000802a0

B. (2 points) What is the address of the instruction that initiates the first call to `linearConvergence`?

Address: 0x00000210

C. (3 points) What address does the label `negative` correspond to?

Address: 0x00000254

D. (4 points) If `alpha` is 2 and `beta` is 17, list two possibilities for the initial value of `input:`

Possible initial value for `input`:
One of 44, 45, 46, 47 (in decimal) or 0x2c, 0x2d, 0x2e, 0x2f (in hex)
Another possible initial value for `input`:
A different one of the four above values

E. (3 points) Assume that `linearConvergence` is initially called with a depth of 0. Does the function converge in time? If yes, then how many recursive calls are necessary (value of depth when the function completes)? If not, what is our value of `MAXIMUM_DEPTH`?

Yes or No?
NO
Value:
MAXIMUM_DEPTH = 4

This page intentionally left blank

Problem 8. Summy and Char (12 points)

Here's some code:

```
int sumCharArray(const char * ar) {
   int a = 0;
   char * c = ar;
   while (*c != 0){
        a += *c;
        c++;
   }
   return a;
}

int m1(const char * ar) {
   int a = 0;
   int b = 0;
   char * c = ar;
   while (*c != 0){
       a += *c;
       b++;
       c++;
   }
   return a/b;
}

int m2(const char * ar1, const char * ar2, char * ar3) {
   int a = 0;
   int b = 0;
   char * c = ar1;
   char * d = ar2;
   char * e = ar3;
   while (*c != 0 && *d != 0){
       *e = (*c + *d )/2;
       a++;
       c++;
       d++;
       e++;
   }
   *e = 0;
   return a;
}
```

We then run this code:

```
int main(void){
   char szn[] = "SUMMer"; //season
   char mit[] = "MIT";    //MIT
   char cit[] = "CIT";    //Caltech
   char git[] = "GIT";    //rambling wreck

   szn[4] = 0;

   printf("print1: %d\n", sumCharArray(szn));
   printf("print2: %d\n", sumCharArray(mit) - sumCharArray(cit));
   printf("print3: %d\n", sumCharArray(cit) - sumCharArray(git));
   printf("print4: %c\n", m1(szn) + 5);

   char c[100];
   int cap = m2(szn, mit, c);
   printf("print5: %d\n", cap);
   printf("print6: %s\n", c);
 }
```

Fill in the print transcripts below (pay attention to the string formatting arguments).

```
print1: 322

print2: 10

Print3: -4

Print4: U

Print5: 3

Print6: POP
```