MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

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

Spring 2026, Quarter 3

Name:	**Kerberos:**
	MIT ID #:

#1 (11)	
#2 (12)	
#3 (13)	
#4 (15)	
#5 (16)	
#6 (6)	
#7 (13)	
#8 (14)	
Total (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.

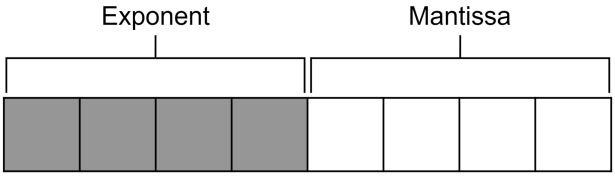
IMPORTANT: Avoid talking about and communicating the contents of this exam with other students until we have announced it is ok to do so on Piazza. Failure to do so will be considered an academic policy violation.

Problem 1. CPI Record-keeping (11 points)

A. (2 Points) The Cambridge Polytechnic Institute is trying to keep an accurate record of the number of students in each class while minimizing memory space. They anticipate that every class is going to have up to 120 students.

What is the minimum number of bits needed to represent the possible number of students in a class [0, 120]?	

B. (6 Points) CPI has decided to update its recording-keeping system to use a modified 8-bit version of floating point (`float8_t`) with no sign bit, no exponent offset, 4 exponent bits, and 4 mantissa bits (illustrated below). Answer the following questions using this new data type.



$$2^{\text{exp}} \cdot \left(1 + \sum_{i=1}^{4} b_{4-i} 2^{-i}\right)$$

Convert decimal number 2.5 into 8-bit floating point format. Write your answer in binary.	
Convert hexadecimal number 0x3C from `float8_t` to decimal.	
What is the maximum value that can be represented by `float8_t`? Answer in decimal. You can leave your answers unsimplified e.g. $(1/2 + 1/16) \cdot 2^4$	

C. (3 Points) Satisfied with the new format, the registrar writes a quick test that adds a new student to a class to make sure the new system is working:

```
void main() {
    float8_t fp1 = 256.0;
    float8_t fp2 = fp1 + ((float8_t) 1.0);

    if (fp2 > fp1) {
        printf("Test 1 Passed!");
    } else {
        printf("Test 1 Failed!");
    }
}
```

The program runs and it prints
`"Test 1 Failed!"`

Provide a detailed explanation for why the test case failed.

Problem 2. Re-Assembly (12 points)

TIM was trying to execute the following expression in RISC-V assembly:

```
a = ((0x6190 << 12) + 731) ^ b
```

`a` is stored at memory address `0x6004` and `b` is stored at memory address `0x6008.`

Unfortunately, TIM forgot to save his progress, and the code got corrupted! Here's a partial reconstruction of the code:

```
__BLANK1__
lui a1, 0x6
__BLANK2__
lw a2, 8(a1)
__BLANK3__
addi a5, x0, 731
__BLANK4__
add a6, a6, a5
__BLANK5__
xor a6, a6, a2
__BLANK6__
```

A. (3 points) TIM finds two lines of additional code, but he only has them saved as 32-bit instructions. He wants to figure out if they belong in the code. Help TIM decode the instructions!

`0x0105A023:`	
`0x06190837:`	

Problem continued on next page.

B. (9 points) TIM remembers the two instructions in part A definitely belong in the code. The problem is TIM doesn't know which instruction goes where, and it turns out one of the instructions is slightly *incorrect*. Answer the following questions.

i. (3 points) Please identify the incorrect instruction and explain why it's incorrect.

ii. (3 points) Encode the correct instruction as a hexadecimal.

`0x`

iii. (3 points) For each instruction, indicate in which Blank it should be placed in TIM's original code (assuming all errors are corrected). There may be more than one correct answer.

Instruction (hexadecimal or assembly)	Blank

Problem 3. Hidden Messages (13 points)

A. (6 points) You are given a string and you need to encode it by shifting each character by a fixed number of positions in the alphabet. The shift wraps around the alphabet, preserves the original capitalization (upper case stays upper case; lower case stays lower case), and leaves spaces unchanged. For example, with shift amount = 2: the string `"AEY z"` becomes `"CGA b"` and with shift amount = -2: the string `"AEY z"` becomes `"YCW x"`.

On the next page, write a function that modifies the string *in place*. The inputs to the function will be:

- **`str:`** a null-terminated string containing only alphabetic characters and spaces.
- **`shift_num:`** an integer shift amount.

Some notes:

- The shift amount may be **any 32 bit integer.** Remember that in C, the `%` operator is a remainder operator, not a mathematical modulo. **It will return a negative value if the dividend (left side) is negative** (e.g., `-1 % 5 = -1`).
- Ensure your logic correctly handles "wrap-around" cases so that characters do not fall outside the `'A'-'Z'` or `'a'-'z'` range.
- **You CANNOT assume `string.h` or the string functions listed in the `reference packet` have been included for you in Part A.**

```
void transformString(char* str, int shift_num) {
      // YOUR CODE BELOW









}
```

B. (7 points) After transforming a series of messages, you need to store them in a single contiguous destination string (`dest`), with a single null terminator, and with a size (in bytes) specified by the `capacity` argument. The function should iterate through the `num_messages` provided in the `messages` array, appending each full message to `dest` if space allows. If `dest` does not have sufficient remaining capacity to hold the full message, as much of that message as possible should be copied into `dest`. The function will return the total number of messages successfully added to the destination buffer regardless of whether a message is added fully or partially. Fill in the blanks to complete the function. **The string functions listed in the reference packet have been included for you in Part B.**

```
int concatenateWords(char* dest, __BLANK1__ messages,
                     int num_messages, int capacity) {
    int count = 0;
    int current_len = 0;
    // initialize the destination string to an empty string
    if (capacity > 0) { __BLANK2__ ;}

    for (int i = 0; i < num_messages; i++) {
        int msg_len = __BLANK3__;

        // check if the whole message fits (including '\0')
        if (__BLANK4__) {
            __BLANK5__;
            current_len += msg_len;
            count++;
        }
        else {
        // max chars to append without exceeding capacity (including '\0')
            int space_left = __BLANK6__;
            if (space_left > 0) {
                __BLANK7__;
                count++;
            }
            break; // memory is full, stop
        }
    }
    return count;
}
```

Blank #:	Line of code:
__BLANK1__	
__BLANK2__	
__BLANK3__	
__BLANK4__	
__BLANK5__	
__BLANK6__	
__BLANK7__	

Problem 4. Fun With Pointers (15 points)

Consider the code below.

```
#define ROWS 6
#define COLS 6

void foo(char* messages) {
    //[LOCATION 1]
}

void bar(char character) {
    //[LOCATION 2]
}

char messages[ROWS][COLS] = {
    "HELLO",   // row 0
    "WORLD",   // row 1
    "ARRAY",   // row 2
    "SCOPE",   // row 3
    "STACK",   // row 4
    "QUEUE",   // row 5
};

void app_main() { // Assume any needed setup done
    //[LOCATION 3]
    bar(messages[0][0]);
    for (int i = 0; i < ROWS; i++) {
        foo(messages[i]);
    }
}
```

Answer the questions on the following page.

A. (7 points) Recall that the `sizeof` operator provides the size of a variable or data type in **bytes**. Tim prints sizeof in various locations indicated by **[LOCATION #]** in the code above. Assuming the same 32-bit ESP32 system we've been working with, **complete the table below:**

Operation	Location	Result
`sizeof(messages)`	**3**	
`sizeof(messages[0])`	**3**	
`sizeof(&messages)`	**3**	
`sizeof(messages)`	**1**	
`sizeof(messages[0])`	**1**	
`sizeof(&messages)`	**1**	
`sizeof(character)`	**2**	

B. (2 points) The following code runs to completion. What will be the values of `arr` after the following code is executed?

```
int arr[5] = {0, 1, 2, 3, 4};
int *p = arr;
*(p + 1) = 10;
*(p + 0) = 2;
```

```
arr = [_____, _____, _____, _____, _____]
```

Problem continued on next page

C. (2 points) After the code in part B is done, you execute the following line:

```
printf("%d\n", p);
```

What will it print? If you cannot determine the value, explain why.

D. (2 points) Continuing on the previous code, you execute the following line:

```
int **q = &p;
```

Can you use the variable `q` to modify index 0 in the `arr`? If so, write the code to modify it to be `6190`. If not, explain why not.

E. (2 points) Can you use the variable `q` to modify index 3 in the `arr`? If so, write the code to modify it to be `6190`. If not, explain why not.

Problem 5. Call Me (16 points)

Ben Bitdiddle wants to translate the following C functions into RISC-V Assembly procedures.

```
int find(int *array, int length, int value) {
  for (int i = 0; i < length; i++) {
    if (array[i] == value) return i;
  }
  return -1; // return -1 if value not found in array
}
void delete(int *array, int *length, int value) {
  int index = find(array, *length, value);
  if (index == -1) return;
  for (int i = index; i < (*length)-1; i++) {
    array[i] = array[i+1];
  }
  *length = *length - 1;
}
```

A. (6 points) Ben is rusty with his assembly so his first attempt to write the **`find`** function does not quite work. Modify Ben's implementation **only by changing registers used** so that it works as expected and follows the calling convention. **You only need to fill in the blanks for modified registers.** You may assume `length` is at least `1`.

```
find:
  mv s0, zero
loop:
  slli a4, s0, 2

  add a4, a1, a4

  lw a4, 0(a4)

  beq a4, a0, done

  addi a3, a3, 1

  blt a3, a2, loop

  li a1, -1
  ret
done:
  mv t0, a3
  ret
```

```
find:
  mv  _____, _____
loop:
  slli _____, _____, 2

  add _____, _____, _____

  lw _____, 0(_____)

  beq _____, _____, done

  addi _____, _____, 1

  blt _____, _____, loop

  li _____, -1
  ret
done:
  mv _____, _____
  ret
```

B. (10 points). The C implementation for `delete` has been repeated below for your convenience.

```
void delete(int *array, int *length, int value) {
  int index = find(array, *length, value);
  if (index == -1) return;
  for (int i = index; i < (*length)-1; i++) {
    array[i] = array[i+1];
  }
  *length = *length - 1;
}
```

The following is Ben's implementation for **`delete`**. Unfortunately, the program does not adhere to the calling convention.

```
delete:

  lw a1, 0(a1)
  call find
  li s0, -1
  beq a0, s0, delete_end

  addi s1, a0, 0     # i = index
  lw a4, 0(a1)       # a4 = *length
  addi a4, a4, -1    # a4 = (*length)-1

delete_loop:
  bge s1, a4, cont

  # calculate the address
  slli t0, s1, 2
  add t0, a0, t0     # array + 4*i

  # update the array
  lw t1, 4(t0)       # array[i+1]
  sw t1, 0(t0)       # array[i] = array[i+1]

  addi s1, s1, 1     # i++
  j delete_loop

cont:
  sw a4, 0(a1)       # *length = *length - 1
delete_end:
  ret
```

Assume that **`delete`** calls a correct implementation of `find`. **Add appropriate instructions into the blank spaces <u>on the next two pages</u>** to make `delete` follow the calling convention. You may only:

- increment/decrement stack pointer
- load word from stack
- save word to stack.

You may assume that the implementation will work as expected once it follows the calling convention. You may not assume any further details about `find` (e.g. the implementations may not be the same as part A and may alter caller-saved registers). You may not need all the blank lines. For full credit, your implementation should minimize the total number of accesses to the stack during the execution of `delete`.

```
delete:

  _____________________________________

  _____________________________________

  _____________________________________

  _____________________________________

  _____________________________________

  _____________________________________

  _____________________________________

  _____________________________________

  lw a1, 0(a1)
  call find
  li s0, -1
  beq a0, s0, delete_end

  _____________________________________

  _____________________________________

  _____________________________________

  addi s1, a0, 0     # i = index
  lw a4, 0(a1)       # a4 = *length
  addi a4, a4, -1    # a4 = (*length)-1
```

```
  _____________________________________

  _____________________________________

  _____________________________________

  _____________________________________

delete_loop:

  _____________________________________

  _____________________________________

  _____________________________________


  bge s1, a4, cont

  slli t0, s1, 2
  add t0, a0, t0     # array + 4*i

  lw t1, 4(t0)       # array[i+1]
  sw t1, 0(t0)       # array[i] = array[i+1]

  addi s1, s1, 1     # i++
  j delete_loop

cont:
  sw a4, 0(a1)       # *length = *length - 1
delete_end:

  _____________________________________

  _____________________________________

  _____________________________________

  _____________________________________

  _____________________________________

  _____________________________________

  ret
```

Problem 6. A Tasty Byte (6 points).

In full RISC-V, there are instructions that allow you to load and store single bytes. Unfortunately, in the 6.1903 ISA, we only have `lw` and `sw`, which make individual byte manipulation quite difficult. To remedy this, we can define two functions shown below, with `byteRead` fully defined in assembly and `byteWrite` partially defined in assembly. Your job is to complete `byteWrite` and make sure it works consistently with `byteRead` as it is defined.

```
//reads a byte from an array arr at index byte_index and returns its value
uint8_t byteRead(uint8_t *arr, uint8_t byte_index);

//writes the byte value to an array arr at index byte_index
void byteWrite(uint8_t *arr, uint8_t byte_index, uint8_t value);
```

You can assume that `arr` is always *word-aligned*, meaning that index 0 is in a byte address that has either `0x0`, `0x4`, `0x8`, or `0xC` in the lower four bits of its address.

```
byteRead:
    andi t0, a1, 0x3
    slli t0, t0, 0x3
    andi t1, a1, 0xFFC
    add  t1, a0, t1
    lw   t2, 0(t1)
    srl  t2, t2, t0
    andi a0, t2, 0xFF
    jalr zero, 0(ra)
```

```
byteWrite:
    andi t0, a1, 0x3
    slli t0, t0, 0x3
    srli t1, a1, __BLANK1__
    slli t1, t1, __BLANK2__
    add  t1, a0, t1
    lw   a4, 0(t1)
    addi t3, zero, __BLANK3__
    sll  t3, t3, __BLANK4__
    xori t3, t3, __BLANK5__
    and  a4, a4, t3
    sll  t4, a2, t0
    or   t2, __BLANK6__, t4
    sw   t2, 0(t1)
    jalr zero, 0(ra)
```

Fill in the 6 missing values so that the RISC-V implementation of `byteWrite` will work appropriately.

BLANK	Value	BLANK	Value
__BLANK1__		__BLANK4__	
__BLANK2__		__BLANK5__	
__BLANK3__		__BLANK6__	

Problem 7. Stack Criminal (13 points)

Below is some assembly.

```
.= 0x0500
addi a0, zero, 6 #Line 1
addi a1, zero, 8 #Line 2
jal ra, f2 #Line 3
jal ra, f1 #Line 4
jal ra, f3 #Line 5
#...
#...


.= 0x1000
f1:
    addi sp, sp, -8
    sw ra, 0(sp)
    sw a1, 4(sp)
    jal ra, f3
    lw a1, 4(sp)
    add a0, a1, a0
    lw ra, 0(sp)
    addi sp, sp, 8
    jalr zero, 0(ra)
.= 0x1100
f2:
    addi a0, a0, 1
    srli a0, a0, 1
    jalr zero, 0(ra)
.= 0x1200
f3 :
    addi t0, zero, 2
    blt a0, t0, done
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, f2
    jal ra, f3
    addi a0, a0, 1
    lw ra, 0(sp)
    addi sp, sp, 4
    jalr zero, 0(ra)
done:
    addi a0, zero, 0
    jalr zero, 0(ra)
```

Continued on next page.

A. (4 points) Specify the value of the following four bits.

Bit of Interest	Value
Bit 31 of the instruction at `0x100C`	
Bit 31 of the instruction at `0x1214`	
Bit 31 of the instruction at `0x1224`	
Bit 31 of the instruction at `0x1208`	

B. (9 points) The code is run starting at address Line 1 and completes after running Line 5 (marked by comments). A portion of memory and register `sp` are shown on the next page before executing Line 1. Answer the following questions. *Note the blank spots in the table on the next page are left as scratch space and may be used in assigning partial credit.*

Question	Answer		
What is the lowest value sp takes on during the execution of Lines 1 to 5?			
Which addresses in the table have different values right after Line 5 compared to what they were right after Line 4? List the addresses and their values after running Line 4 and running Line 5.	**Address**	**After Line 4**	**After Line 5**
How many total `sw` instructions are executed across all five lines?			
What is the value of `a0` immediately before `f1` executes `jalr zero, 0(ra)`?			

Address	Before Line 1	After Line 3	After Line 4	After Line 5
0x00001FDC	0x41CF7D5E			
0x00001FE0	0x00000020			
0x00001FE4	0xABCDEF01			
0x00001FE8	0x006100DB			
0x00001FEC	0x19861985			
0x00001FF0	0x41CF7D5E			
0x00001FF4	0x00001FF4			
0x00001FF8	0x00000020			
0x00001FFC	0xDEADBEEF			
0x00002000	0x00000000			
0x00002004	0xA3F7D14B			

sp	0x00002000			

Problem 8. TimGPT (14 points)

MIT is developing TimGPT, a chatbot for MIT students, and has requested your help in developing a user management system. They have defined several C structs and functions. A struct called `User` is defined below to represent a user in the user management system. A struct called `Message` stores a message instance, which contains the user message, the response from the AI chatbot, and the time stamp. **You may assume the string functions defined in the reference packet have been included for all parts of this problem.**

```
struct Message {
    char user_message[100];    // Array to store a single message from the user
    char ai_response[100];     // Array to store a response from the AI chatbot
    uint32_t time_stamp;       // Time stamp that stores the time of the day
                               //(hours, minutes, seconds) the message was sent
};
struct User {
    uint32_t user_id;             // Unique user ID (Kerberos ID)
    char name[50];                // Array to store the student's name.
    struct Message messages[50];  // Array of messages for a user
    uint8_t num_messages;         // Number of messages
};
```

A. (2 points) Write a function called `createUser` that takes the `user_id`, `name`, and returns a `User` struct initialized with these values. The number of messages should be initialized to 0 by default. Assume that the provided `name` is properly sized and null-terminated.

```
struct User createUser(uint32_t user_id, const char *name) {
      struct User new_user; // Create a new user instance
      // YOUR CODE BELOW









      return new_user;
}
```

B. (3 points) The user management system stores each message instance in a **`Message`** struct that contains the user message, the response from the AI chatbot, as well as a time stamp. The time stamp only stores the hour (0-23), minutes (0-59), and seconds (0-59) (i.e., the year/month/day does not matter). To save space, the system uses a single `uint32_t` value to store all the information as follows:

```
[ unused ... ]  [ HHHHH ]  [ MMMMMM ]  [ SSSSSS ]
   15 bits       5 bits      6 bits      6 bits
```

That is, the least significant 6 bits encode the seconds, the next 6 bits encode the minutes, and the next 5 bits encode the hour. The leading 15 bits are not used (and set to 0). Write a function that takes in the hour, minutes, seconds (all given as unsigned 32-bit integers) and returns the time stamp in the above format. You should assume that you are given a valid time (hours will range from 0 to 23, minutes will range from 0 to 59, and seconds will range from 0 to 59).

```
// Returns the time stamp as a 32-bit integer
uint32_t createTimeStamp( uint32_t hour,
                          uint32_t minutes,
                          uint32_t seconds) {
      // YOUR CODE BELOW









}
```

C. (5 points) You now need to create a function that inserts a new user message, the associated AI response, and the time (hour, minutes, seconds), into the `User` struct as well as updates it. You should use the function `createTimeStamp` you implemented in the previous page to construct the time stamp. And you can assume there is enough space for the new message (`num_messages<50`).

```
void addMessage(struct User *user, const char *user_message,
                const char *ai_response, uint32_t hour,
                uint32_t minutes, uint32_t seconds){
    // YOUR CODE BELOW









}
```

This page intentionally left blank

D. (4 points) We assume that a new `Message` has been inserted into the `messages` array at `message_index`. In order to prevent students from using the chatbot to do their homework, we would like to manually overwrite the AI response in that message (i.e., `ai_response`) to say the following if the user message (i.e., `user_message`) contains the keyword "`homework`":

`"I'm sorry [name]. I'm afraid I can't do that."`

So if a student named `Dave` sends in a message like:

`"Can you help me with my homework?"`

We would like to replace the original `ai_response` with:

`"I'm sorry Dave. I'm afraid I can't do that."`

```
void overwriteResponse(struct User *user, int message_index) {
        // YOUR CODE BELOW









}
```