MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

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

Spring 2025, Quarter 4

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

#1 (15)	**15**
#2 (13)	**13**
#3 (14)	**14**
#4 (15)	**15**
#5 (15)	**15**
#6 (16)	**16**
#7 (12)	**12**
Total (100)	**100**

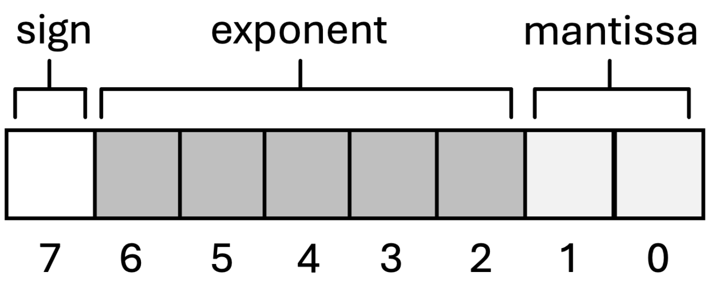
Exam content is on **BOTH SIDES** of the exam sheets.

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. Mini Float (15 points)

A **`float8_t`** is an 8-bit floating point type that consists of 1 sign bit, 5 exponent bits, and 2 mantissa bits. It uses an exponent bias of 15. Recall that the exponent is encoded as an unsigned number.



$$(-1)^{\text{sign}} \cdot 2^{\text{exp}-15} \cdot \left(1 + \sum_{i=1}^{2} b_{2-i} 2^{-i}\right)$$

A. (2 points) Answer the following (you should leave your answers in simplified power-of-2 scientific notation, e.g. $1.5 \cdot 2^3$):

What is the largest possible value that a **`float8_t`** can represent?	**$1.75 * 2^{16}$**
What is the smallest **positive** value the **`float8_t`** can represent?	**$1.0 * 2^{-15}$**

B. (2 points) Given the **`float8_t`** value `0b11000101`, what is its decimal representation?

```
1_10001_01

sign is negative
exponent = 10001 = 17
mantissa = 01

2^(17-15) * (1 + 0.25) = 5 → negative = -5
```

C. (3 points) Consider the code below:

```
float8_t x = 0b11000101;
float8_t y = 0b01000011;
float8_t z = x+y;
```

What is the 8 bit binary representation of `float8_t z`?

```
1_10001_01 → -5
0_10000_11 → 3.5

-5 + 3.5 = -1.5

1_01111_10
```

D. (1 point) Express your answer from part C in hexadecimal.

```
0xBE
```

E. (2 points) Given the variable **`float8_t X`**, write an expression for the absolute value of **`X`** as a **`float8_t`** using only bitwise operators.

```
X & 0x7F
```

F. (5 points) Let's imagine we're on a computer that does not natively support `float8_t` data types. We would still like to work with them, so we'll handle them using `uint8_t` variables, just like we did in Lab 3 where we used `uint8_t` variables to hold locations for `Snake` segments and food.

Write a function, `divideByPowerOfTwo`, which takes in a `uint8_t val` containing a number encoded as a `float8_t` and a second argument `uint8_t pow`, which is used to specify a power-of-two. Specifically, the function should interpret the bits of `val` as a `float8_t`, divide that value by 2 raised to the power `pow`, and finally return the result in `float8_t` format. For example: if `val` encodes `32.0` and `pow=3`, the result should be the `float8_t` encoding of `4.0`.

If the result of the division is too small to be represented using the `float8_t` encoding, the function should return the smallest possible `float8_t` value with the correct sign.

```
uint8_t divideByPowerOfTwo(uint8_t val, uint8_t pow) {


  uint8_t exp = (val & 0x7C)>>2;
  if (exp>=pow){
    uint8_t final = val & 0x83;//clear exponent
    final |= (exp - pow)<<2;
    return final;
  } else{
    uint8_t final = val & 0x80; //keep only sign and do smallest value
    return final;
  }

}
```

This page intentionally left blank

Problem 2. Delivery System (13 points)

Your friends want to build a grocery delivery system where customers can place orders and get products delivered to their home. They implemented the following `structs` to store the data on a 32-bit system:

```
struct Product{
  char name[40];          // name of the product
  float price;            // price of the product
};

struct Customer{
  char name[40];         // customer name
  char phone_number[12]; // customer phone number
  char address[100];     // delivery address
};

struct Order{
  uint16_t id;           // order ID
  uint8_t num_products;  // number of products ordered by the customer
  uint8_t status;        // delivery status
  struct Customer customer;     // customer data
  struct Product* products[15]; // an array of Product pointers

};
```

A. (3 points) First, they need to evaluate how much memory each struct type uses. Using the `sizeof` operator, complete the table below. You can assume no byte-alignment or padding in the structs.

Operation	Value
`sizeof(struct Product)`	**44**
`sizeof(struct Customer)`	**152**
`sizeof(struct Order)`	**216**

B. (4 points) Next, they want to print the name of a product given a variable `order` of type `struct Order`. Assume that the product at index 1 exists and is valid. **Circle all the correct ways** to access the name of this product. You will earn points for each correct answer you circle, but lose points for each incorrect answer you circle.

1.	`order.products[1]->name`	2.	`(*order.products[1]).name`
3.	`*order->products[1].name`	4.	`order->products[1]->name`
5.	`(*(order.products + 1))->name`	6.	`(*(order.products + 4))->name`
7.	`*(order.products[1]->name)`	8.	`(**(order.products + 1)).name`

C. (6 points) After employing the system, they notice a bug. The names of **some** customers have no space between the first and last name, but are instead separated by an underscore ('_'). For example, `"Alex Alibaba"` is mistakenly stored as `"Alex_Alibaba"`. This causes an error in the payment system. Help your friends write a function to replace the first underscore ('_') with space in the names of all customers, given an array of `struct Order`, by filling in the blank with the correct line of code.

```
#include<string.h>

__BLANK1__ replaceUnderscore(__BLANK2__ orders, int num_orders){
    char *ptr;
    for(int i = 0; i < num_orders; i++){
        ptr = __BLANK3__; // use an appropriate function from string.h here
        if (__BLANK4__){
            __BLANK5__;
        }
    }
    return;
}
```

Blank #:	**Line of code:**
__BLANK1__	`void`
__BLANK2__	`struct Order*`
__BLANK3__	`strchr(orders[i].customer.name, '_')`
__BLANK4__	`ptr != NULL`
__BLANK5__	`*ptr = ‘ ’`

Problem 3. Get a Grep (14 points)

The function `findDir` below is designed to search through the directories (i.e. folders) of a simplified file structure to find a matching directory. In this file structure, every directory is represented by a `Directory` struct:

```
  struct Directory {
    uint32_t dir_id;                  // directory ID
    struct Directory* children[10];  // an array of Directory pointers...
                                     // representing all children...
                                     // of this directory
  };
```

Within the `children` array, each element is a pointer to another `Directory` struct or a `NULL` pointer; if a directory has fewer than 10 children, all unused spaces in the children array will hold a `NULL` pointer. This function runs recursively to search each child of **`base_dir`** until it finds a child directory with a matching **`target_id`**. **It returns the depth of the recursion at which the directory is found.** If the directory is not found, the function returns the value -1.

```
int findDir(struct Directory *base_dir, uint32_t target_id) {
  if (base_dir->dir_id == target_id){
    return 1;
  }
  for(int i = 0; i < 10; i++){
     if (base_dir->children[i] != NULL){
       int n = findDir(base_dir->children[i], target_id);
       if (n >= 0){
          return n + 1;
      }
    }
  }
  // if none of the children find the directory, return -1
  return -1;
}
```

The following RISC-V assembly code is a translation of the above C function, adhering to calling convention. In this RISC-V assembly code, in order to access the **`dir_id`** member of a `Directory`, one must access the data at the address **40 bytes offset** from the pointer to the `Directory` struct. The `children` array begins at the same address as the base address of the `Directory` struct, and each of its 10 elements uses 4 bytes of space for a pointer to another `Directory` struct.

The special instruction **`stack`** indicates the execution point where the code was interrupted to capture the stack data displayed on the following page.

```
findDir:
  addi sp, sp, -20
  sw a0, 0(sp)
  sw a1, 4(sp)
  sw ra, 8(sp)
  sw s0, 12(sp)
  sw s1, 16(sp)

  lw s0, 40(a0)    # access base_dir->dir_id
  addi s1, x0, 0   # int i = 0;
  bne s0, a1, findDir_loop

  stack
  addi a0, x0, 1
  jal x0, findDir_end

findDir_loop:
  slli t0, s1, 2
  add t0, a0, t0
  lw a0, 0(t0)

  beq a0, x0, findDir_finishloop
  jal ra, findDir

  blt a0, x0, findDir_finishloop

  addi a0, a0, 1
  jal x0, findDir_end

findDir_finishloop:

  lw a0, 0(sp)   # restore caller-saved registers we care about
  lw a1, 4(sp)

  addi s1, s1, 1
  addi t1, x0, 10
  blt s1, t1, findDir_loop

  addi a0, x0, -1

findDir_end:

  lw ra, 8(sp)
  lw s0, 12(sp)
  lw s1, 16(sp)
  addi sp, sp, 20

  jalr x0, 0(ra)
```

The **`findDir`** function is called, and the capture of stack memory displayed below is recorded at the first and only moment when the **`stack`** instruction is passed, inside of a recursive call to `findDir`. At the time of capture, the value of `sp` was `0x4110`. Use this stack information and the assembly implementation to answer the following questions about this call to `findDir`.

Address	Contents	stack value meaning
0x4100	0x0000000F	
0x4104	0x00009988	
0x4108	0x00004324	
0x410C	0x000000F7	
0x4110	0x00003494	a0: final base_dir pointer
0x4114	0x00000066	a1: target_id
0x4118	0x0000512C	recursive ra
0x411C	0x00000031	saved s0: prev call's dir_id
0x4120	0x00000001	saved s1: prev call's i
0x4124	0x00003440	a0: recursive base_dir pointer
0x4128	0x00000066	a1: target_id
0x412C	0x0000512C	recursive ra
0x4130	0x0000018F	saved s0: prev call's dir_id
0x4134	0x00000004	saved s1: prev call's i
0x4138	0x00003300	a0: initial base_dir pointer
0x413C	0x00000066	a1: target_id
0x4140	0x00005544	original ra
0x4144	0x00000771	s0: unknown contents
0x4148	0xFF09FF83	s1: unknown contents
0x414C	0x00000009	
0x4150	0x00003394	
0x4154	0xFFFFFFFE	

A. (1 point) What is the memory address corresponding to the instruction that made the original (non-recursive) call to `findDir` ?

Memory Address: `0x`**`5540`**

B. (2 points) What is the memory address corresponding to the label `findDir_finishloop`?

Memory Address: `0x5138`

C. (2 points) What were the original values `base_dir` and `target_id` passed into the initial function call?

`base_dir`: `0x3300`	**`target_id`**: `0x66`

D. (2 points) What is the memory address of the `Directory` struct that matched the search condition of the function call?

Memory Address: `0x3494`

E. (2 points) What depth value will be returned by the initial call to `findDir`?

Depth: ____3_____

F. (3 points) What are the IDs of all the directories in the path from the original base directory to the matching directory? List them in the order of base directory to the matching directory, including both ends.

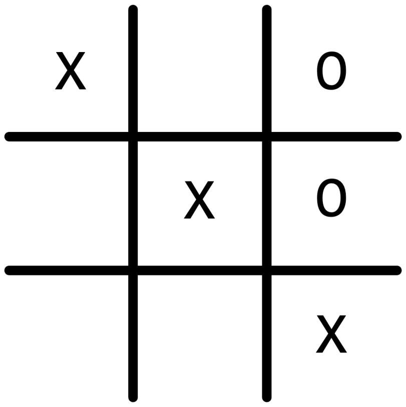
List of IDs in path:___`0x18F, 0x31, 0x66`___________

G. (2 points) In the initial call to `findDir`, how many direct child directories were searched unsuccessfully before finding the matching directory?

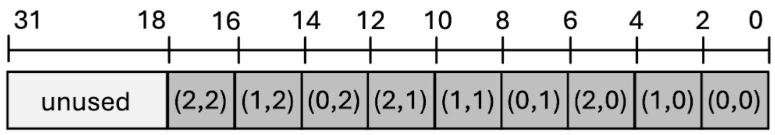
Number of child directories searched: ____4_____

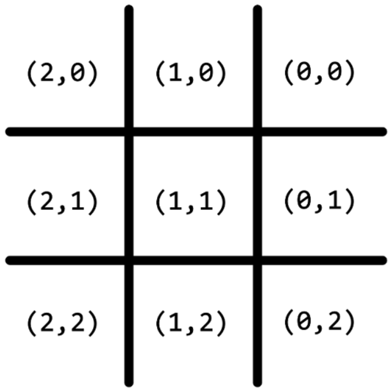
Problem 4. Tic-Tac-Toe (15 points)

Consider a game of tic-tac-toe. The game board consists of nine cells, with each cell being either empty, having an X, or having an O. Starting with a blank board, players take turns placing X's and O's in one of nine cells until one of them achieves a three-in-a-row victory. Here's a game where X has won by having three-in-a-row on the diagonal.



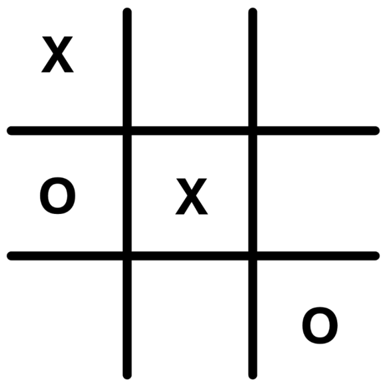
We'd like to implement tic-tac-toe on our 32 bit RISC-V system. To do this we're going to encode the entire 3x3 game board in a single `uint32_t`. To keep things clean, the bottom 18 bits will be used to encode all nine-cells in the following order, with the game board indices `(i, j)` (i.e., column `i`, row `j`) shown on the right below:



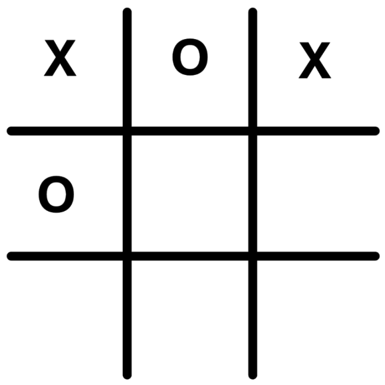


- Each cell is encoded as a 2-bit value with the following encoding scheme:
    - An empty cell is encoded as: `0b10`
    - A cell containing an `'O'` is encoded as: `0b01`
    - A cell containing an `'X'` is encoded as: `0b11`
    - `0b00` is an invalid state
- There exists a global pointer to the game board called `gb` that holds the `uint32_t` of game state.
- This game has two players, each represented by a single character: `'X'` and `'O'`.

Two example encodings are provided below (note the upper 14 bits are set to 0 here for the sake of clarity, but as stated on the previous page, they are unused and ignored):



0x000297BA



0x0002A6B7

Your main task is to complete the provided functions, in the spaces where there are blanks (e.g. `__BLANK1__`). These functions interact with this pre-defined, compact game state. For this problem, understanding the game *logic* of tic-tac-toe is not expected or required.

A. (2 points) Write a function `resetBoard` to initialize or reset the game state, setting all cells on the board back to their empty state.

```
void resetBoard(uint32_t *gb){

    *gb = 0x2AAAA;






}
```

B. (7 points) Complete the function `updateGameBoard` below so that it returns a 0 if the player tries to move to an occupied cell. If the player tries to move to an empty cell, then it should replace the empty cell with the player's encoding in the game board and return a 1.

```
// Parameters:
// * gb:     A pointer to the game board.
// * player: A char representing the player making the move
// * i:      The column index (0-2) of the move.
// * j:      The row index (0-2) of the move.
// Behavior:
// * If the cell at (i, j) is already occupied: the function should return 0 and
//   not modify the board.
// * If the cell at (i, j) is empty: the function should update the board by
//   placing the player's mark using the correct 2-bit encoding and return 1.

uint8_t updateGameBoard(uint32_t *gb, char player, uint8_t i, uint8_t j){
    // Calculate the bit position for cell at coordinates (i, j)
    uint8_t shift_amount = __BLANK1__ ;

    // Check if this cell is already occupied. Otherwise, populate that cell.
    if (__BLANK2__){
        return 0;
    } else {
        if (player == 0x58){
            __BLANK3__
        }
        if (player == 0x4F){
            __BLANK4__
        }
        return 1;
  }
}
```

Blank #:	Line of code:
`__BLANK1__`	`(j * 3 + i) * 2`
`__BLANK2__`	`((*gb >> shift_amount) & 0b11) != 0b10`

`__BLANK3__`
 	
`(Multiple lines OK)`
	`*gb |= (0b11 << shift_amount);`

`__BLANK4__`
 	
`(Multiple lines OK)`
	`*gb &= ~(0b11 << shift_amount)); // clear cell`
	`*gb |= 0b01 << shift_amount;     // set set`

C. (6 points) Complete the function `checkForWinner` to detect if a player has won. It should return the winner's `char` (`'X'` or `'O'`) if found, otherwise the null character. For convenience, we've provided an array containing sub-arrays whose indexes correspond to each of the eight winning index combinations:

```
uint8_t widx[8][3] =
{
    {0,1,2},    // Top row
    {3,4,5},    // Middle row
    {6,7,8},    // Bottom row
    {0,3,6},    // Right column
    {1,4,7},    // Middle column
    {2,5,8},    // Left column
    {0,4,8},    // Diagonal (top-right to bottom-left)
    {2,4,6},    // Diagonal (top-left to bottom-right)
};
```

Cell Index Mapping:

2	1	0
5	4	3
8	7	6

```
char checkForWinner(uint32_t *gb){
    // Iterate through the 8 possible winning lines
    for (int k = 0; k < 8; k++){
        // Extract the 2-bit values of the three cells in the line
        uint8_t c1 = __BLANK1__ ;
        uint8_t c2 = __BLANK2__ ;
        uint8_t c3 = __BLANK3__ ;

        // Check if all three cells are the same AND are not the empty cell.
        if (__BLANK4__){
            // If so, return the appropriate character.
            if (__BLANK5__){
                return 'O';
            }else{
                return 'X';
            }
        }
    }
    // If the loop completes without finding a winner, return null character.
    return NULL;
}
```

Blank #:	Line of code:
__BLANK1__	`(*gb >> (widx[k][0] * 2)) & 0b11`
__BLANK2__	`(*gb >> (widx[k][1] * 2)) & 0b11`
__BLANK3__	`(*gb >> (widx[k][2] * 2)) & 0b11`
__BLANK4__	`(c1 == c2 && c2 == c3) && (c1 != 0b10)`
__BLANK5__	`c1 == 0b01 // or c2 or c3`

Problem 5. RISC-V Assembly (15 points)

A. (6 points) We ran the following function through a buggy C compiler and it produced the following buggy RISC-V assembly code. Please help us correct it by identifying the **4 incorrect lines** and replacing them with a correct RISC-V instruction in the right hand column. **Do not use pseudoinstructions.**

```
// Original C function

int mystery(int x, int y, int z) {
    if (z/2 < y) {
        x += z;
        return x;
    } else if (x % 2 == 0) {
        y = x - z;
        return y;
    }
    return x;
}
```

# Assembly output **# HINT: There are 4 incorrect lines** mystery:	**For each incorrect line of the function, write the correct line of assembly code in its corresponding blank box below (no pseudoinstructions):**
`slli t0, a2, 1`	`srai t0, a2, 1`
`bge t0, a1, label1`	
`add t1, a0, a2`	`add a0, a0, a2`
`jal ra, label2`	`jal zero, label2`
`label1:`	
`addi t0, a0, 1`	`andi t0, a0, 1`
`bne t0, zero, label2`	
`sub a0, a0, a2`	
`label2:`	
`jalr x0, 0(ra)`	

B. (4 points) Rewrite each of the four code sequences below with a single RISC-V instruction that produces the same results for the `a0-a7` registers. Note that the resulting values of the `t0-t6` registers does not need to match across the two implementations. **Do not use pseudoinstructions.**

`# Original assembly code`	**Single RISC-V instruction that produces equivalent results in `a0-a7` (no pseudoinstructions):**

`addi a0, zero, 0x37`
`slli a0, a0, 12`
	`lui a0, 0x37`

`not a2, a1`	`xori a2, a1, -1`

`li t0, 0xE`
`srli t0, t0, 3`
`beq t0, zero, done`
`add a1, a1, t0`
 	
`done:`
	`addi a1, a1, 1`

`addi t1, zero, 0xFFE`
`li t0, 47`
`sub a2, t0, t1`
	`addi a2, zero, 49 (or equivalent)`

C. (5 points) The following code snippet is run until the code reaches the `end` label. Fill in the requested values in the table below after the code is run:

```
. = 0x100
    addi t3, zero, 0x2C
    li a1, 0x61904
    lw a2, 0x600(t3)
    xori a5, t3, 0x74
    beq a1, a1, end

. = 0x620
    .word 0x12345678
    .word 0x33333333
    .word 0x88664422
    .word 0xABCDEF01
    .word 0x55337799
    .word 0x45456767

end:
```

Question	Answer
Address of `lw a2, 0x600(t3)` instruction:	`0x10C`
32-bit encoding of `xori a5, t3, 0x74` instruction:	`0x074E4793`
`a2 =`	`0xABCDEF01`
`a5 =`	`0x58`

This page intentionally left blank

Problem 6. Pythagorean Protocol Pitfalls (16 points)

The following assembly function, **`hypotenuse`**, is a **buggy implementation** that tries to compute the hypotenuse of a right triangle given the length of each of its legs. It calls two helper functions, **`square`** and **`sqrt`** whose full implementations are omitted. Assume both helper functions only take one argument and properly adhere to the RISC-V calling convention.

```
Instruction   # hypotenuse
Address       # ARGUMENTS:
              #   a0: leg1
              #   a1: leg2
              # RETURNS: $\sqrt{(leg1^2 + leg2^2)}$

              hypotenuse:
       0x500    addi sp, sp, 4
       0x504    sw ra, 0(sp)

       0x508    call square

       0x50C    mv s0, a0
       0x510    sw ra, 0(sp)
       0x514    call square

       0x518    add a0, a0, s0
       0x51C    sw ra, 0(sp)
       0x520    call sqrt

       0x524    lw ra, 0(sp)
       0x528    addi sp, sp, -4

       0x52C    ret

              square:
                # IMPLEMENTATION OMITTED
                ret

              sqrt:
                # IMPLEMENTATION OMITTED
                ret
```

Assume that the original instruction call to **`hypotenuse`** was made from address `0x200` and that the stack pointer register `sp = 0x620` at the time of the original call.

A. (1 point) Considering the original call to **`hypotenuse`**, after executing the `sw ra, 0(sp)` at address `0x504`, what is the value of the `sp` register and what is stored at the memory location that it points to?

`sp =`	`0x624`	`Mem[sp] =`	`0x204`

B. (2 points) After executing the `sw ra, 0(sp)` at address `0x510`, what is the value of the `sp` register and what is stored at the memory location that it points to?

`sp =`	`0x624`	`Mem[sp] =`	`0x50C`

C. (1 point) After executing the `sw ra, 0(sp)` at address `0x51C`, what is the value of the `sp` register and what is stored at the memory location that it points to?

`sp =`	`0x624`	`Mem[sp] =`	`0x518`

D. (2 points) After executing the `ret` at address `0x52C`, what is the value of the `sp` register and what is the value of `pc` register?

`sp =`	`0x620`	`pc =`	`0x518`

E. (2 points) Is the return address, **`ra`**, handled correctly? Explain in a few sentences.

No, the original ra saved in line 0x504 is repeatedly overwritten to different values (0x510, 0x51c).

F. (2 points) As written, list **two** issues with how the stack and the stack pointer, **`sp`**, are being handled.

1. "addi sp, sp, 4" and "addi sp, sp, -4" are swapped
2. We are causing a stack underflow by repeatedly calling "addi sp, sp, -4" in an infinite loop

G. (6 points) Our original **hypotenuse** function contains several errors. Please use the blank right column to rewrite the **hypotenuse** function so that it both adheres to the RISC-V calling convention and is functionally correct.

```
# hypotenuse
# ARGUMENTS:
#   a0: leg1
#   a1: leg2
# RETURNS: $\sqrt{(leg1^2 + leg2^2)}$

hypotenuse:
  addi sp, sp, 4
  sw ra, 0(sp)

  call square

  mv s0, a0
  sw ra, 0(sp)
  call square

  add a0, a0, s0
  sw ra, 0(sp)
  call sqrt

  lw ra, 0(sp)
  addi sp, sp, -4
  ret

square:
  # IMPLEMENTATION OMITTED
  ret

sqrt:
  # IMPLEMENTATION OMITTED
  ret
```

```
# hypotenuse
# ARGUMENTS:
#   a0: leg1
#   a1: leg2
# RETURNS: $\sqrt{(leg1^2 + leg2^2)}$

hypotenuse:


addi sp, sp, -8
sw ra, 0(sp)
sw s0, 4(sp)
mv s0, a1 #save first leg
call square #a0 now has first leg squared
mv t1, s0 #swap stuff
mv s0, a0 #etc…
mv a0, t1 #put leg two
call square #square leg two
add a0, a0, s0 #a0 now has sum of each square
call sqrt #call square root answer in a0
lw ra, 0(sp)
lw s0, 4(sp)
addi sp, sp, 8  #put the campsite back

  ret

square:
  # IMPLEMENTATION OMITTED
  ret

sqrt:
  # IMPLEMENTATION OMITTED
  ret
```

This page intentionally left blank

Problem 7. Pointer Detective (12 points)

Consider the code below:

```
#include<stdio.h>
#include<stdint.h>
#include<string.h>

int main(void){
  uint32_t a = 19;
  uint16_t b = 17;
  uint8_t c = 38;

  uint32_t * p1;
  uint16_t * p2;
  uint8_t * p3;

  p1 = &b;
  p2 = &c;
  p3 = &a;


  // note: %08x formats for 8 digits of hexadecimal padded with leading 0s

  printf(" p1: %08x\n", (int)(p1));
  printf(" p2: %08x\n", (int)(p2));
  printf(" p3: %08x\n", (int)(p3));
  printf("*p1: %08x\n", (int)*p1);
  printf("*p2: %04x\n", (int)*p2);
  printf("*p3: %02x\n", (int)*p3);

  p1 = p1+1;
  p2 = p2+1;
  p3 = p3-2;

  printf(" p1: %08x\n", (int)(p1));
  printf(" p2: %08x\n", (int)(p2));
  printf(" p3: %08x\n", (int)(p3));
  printf("*p1: %08x\n", (int)*p1);
  printf("*p2: %04x\n", (int)*p2);
  printf("*p3: %02x\n", (int)*p3);

}
```

When run, the following incomplete printout is generated:

```
 p1: 6ce0b3fa
 p2: 6ce0b3f9
 p3: 6ce0b3fc
*p1: 00130011
*p2: 1126
*p3: 13
 p1: __BLANK1__
 p2: __BLANK2__
 p3: __BLANK3__
*p1: __BLANK4__
*p2: __BLANK5__
*p3: __BLANK6__
```

What is printed in the six blanks? If not enough information is available, write "CAN'T TELL".

__BLANK1__: 6ce0b3fe
__BLANK2__: 6ce0b3fb
__BLANK3__: 6ce0b3fa
__BLANK4__: CAN'T TELL
__BLANK5__: 1300
__BLANK6__: 11