1 MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

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

Spring 2024, Quarter 4

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

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

This page intentionally left blank

Problem 1. Operator (That's Not the Way it Feels) (12 points)

Consider the code below.

```
uint8_t u1 = 0x0E;
uint8_t u2 = 90;
uint8_t u3 = u1 & u2;
uint8_t u4 = u1 - u2;
uint8_t u5 = u1 * u2;
uint8_t u6 = u1 / u2;
uint8_t u7 = u1 | u2;
uint8_t u8 = u1 ^ u2;
```

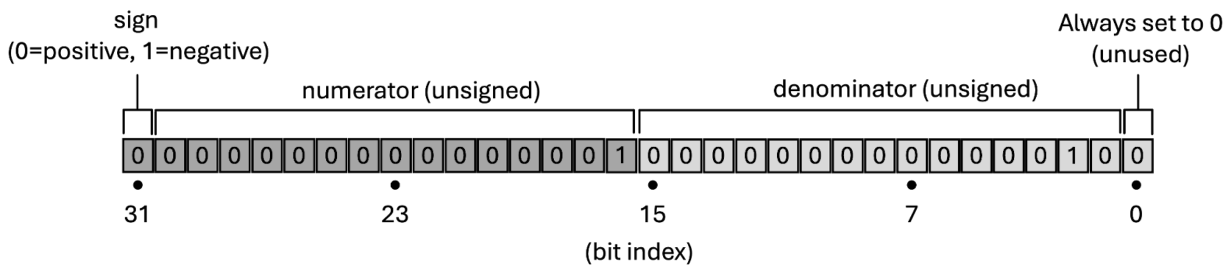
Specify (**in hexadecimal**) the value of each variable after the code runs:

u3	0x0A
u4	0xB4
u5	0xEC
u6	0x00
u7	0x5E
u8	0x54

Problem 2. Fractious Fractions (15 points)

You decide to use 32 bits to represent fractions. One bit is used to represent the sign, 15 bits are used to represent the numerator as an unsigned integer, and 15 bits are used to represent the denominator as an unsigned integer. One bit is left-over and is unused.

The fraction shown in the example encoding below corresponds to $+\frac{1}{2}$



For this problem you can ignore cases where the denominator is 0, and you can also assume all numbers will stay within their assigned bits (don't worry about overflow). Answer the following questions:

A. (2 points) What is the largest positive number that can be represented?

2**15-1

B. (2 points) What is the smallest non-zero positive number that can be represented?

1/(2**15-1)

C. (2 points) Encode $-\frac{13}{31}$. Specify your answer in hexadecimal:

0x800D003E

D. (2 points) Using only the following operators: (`|,&,^,~,<<,>>,>,<,>=,<=`) write a function that takes in a 32 bit data type encoding a fraction as specified on the previous page and returns a **negative of the input fraction**. For example, if the input fraction is -½ it returns ½. `if/else` and ternary operations are not allowed.

```
uint32_t negateFraction(uint32_t a){

   return a^0x80000000;

}
```

E. (7 points) Using only the following operators: (`|,&,^,~,<<,>>,*,/, >,<,>=,<=`) write a function that takes in two 32 bit data types encoding two fractions as specified on the previous page and returns their product also in the same fraction form. `if/else` and ternary operations are not allowed. **Do not simplify the numerator and denominator of your result.** *You can assume the components of both input arguments are small enough to avoid concerns about overflow in the resulting fraction.*

```
uint32_t multiplyFraction(uint32_t a, uint32_t b){

   uint32_t as = a&0x80000000;
   uint32_t bs = b&0x80000000;
   uint32_t s = as^bs;
   uint32_t da = (a & 0xFFFE)>>1;
   uint32_t db = (b & 0xFFFE)>>1;
   uint32_t d = da*db;
   uint32_t na = (a>>16)&0x7FFF;
   uint32_t nb = (b>>16)&0x7FFF;
   uint32_t n = na*nb;
   return s | (n<<16) | (d<<1);

}
```

Problem 3: RISC-Vy Behavior (16 points)

For each of the code snippets below, specify what value ends up in each of the listed registers and memory locations after the code snippet is run to completion.

A. (4 points) Assume that all registers are initialized to 0.

`addi a1, zero, 0x34`
`lui a1, 5`
`addi a2, a1, 0x803`
`li a3, 36`
`xori a4, a3, 0x55`
	Resulting value in hexadecimal:

a1 =	0x5000
a2 =	0x4803
a3 =	0x24
a4 =	0x71

B. (4 points) Assume that all registers are initialized to 0.

```
. = 0x0
      addi a1, zero, 11
      addi a2, zero, 0x10
      blt a2, a1, here
      srli a3, a2, 3
      jal a4, there
      and a4, a4, a4

. = 0x100
here:
      addi a4, a4, 7

. = 0x200
there:
      xori a2, a2, 0x33
```

(code above)	Resulting value in hexadecimal:
a1 =	0xB
a2 =	0x23
a3 =	0x2
a4 =	0x14

C. (6 points) Assume that all registers are initialized to 0.

```
      addi a1, zero, 0x10
loop:
      bgt a2, a1, end
      addi a2, a2, 4
      srli a1, a1, 1
      lw a3, 0x504(a2)
      slli a4, a3, 4
      sw a4, 0x500(a2)
      jal x0, loop

. = 0x500
.word 0x12345678
.word 0xDEADBEEF
.word 0x50505050
.word 0x77773333
.word 0x12345678

end:
```
	Resulting value in hexadecimal:

a1 =	0x4
a2 =	0x8
Mem[0x500]	0x12345678
Mem[0x504]	0x05050500
Mem[0x508]	0x77733330
Mem[0x50C]	0x77773333

D. (2 points) What instruction is encoded by the 32 bit value `0x2096A823`?

Encoded Instruction: `0x2096A823`	Decoded Instruction: `sw x9, 0x210(x13)` `sw s1, 0x210(a3)`

Problem 4: Stringing You Along (6 points)

Your best friend, Charlotte Starr has this weird string code she's been handed, and she wants your help figuring out what's going on. Based on the information given, answer the following questions about the output of the code. ***Recall that if the delimiter string includes multiple characters, then each character is a valid delimiter.***

```
char str[200] = "an interesting string, with A weird catch.";
str[3] = 0;
printf("0x%x\n", (int) str);      // prints 0x2000
char* out = strtok(str + 5, str);
while(out != NULL) {
    printf("%s\n", out);          // print statement A
    printf("0x%x\n", (int) out);  // print statement B
    out = strtok(NULL, str);
}
```

Fill in the following table with the outputs of each print statement each time the loop is executed. You may not need to fill all rows of the table.

Loop #	Output of print statement A	Output of print statement B
1	teresti	0x2005
2	g	0x200D
3	stri	0x200F
4	g,	0x2014
5	with	0x2017
6	A	0x201C
7	weird	0x201E
8	c	0x2024
9	tch.	0x2026
10		
11		
12		

Problem 5: To Save Or Not To Save (14 points)

A. (8 points) Translate the C function **`operations`** into RISC-V assembly. The RISC-V **`func_x`** and **`func_y`** functions are predefined and follow calling convention. Full credit will be awarded to implementations that (1) **follow calling convention** and (2) utilize **no more than 8 bytes of stack space**.

```
// operations C function

int operations (int* a){
    return func_y(func_x(a, *a), *(a+1));
}
```

```
# your RISC-V implementation

operations:
    addi sp, sp, -8
    sw ra, 0(sp)
    sw a0, 4(sp)
    lw a1, 0(a0)
    call func_x
    lw t0, 4(sp)
    lw a1, 4(t0)
    call func_y
    lw ra, 0(sp)
    addi sp, sp, 8
    ret
```

B. (6 points) We have given you an implementation of **`save_identifier`** that is almost correct. Specifically, the logic is correct, but it does not follow RISC-V calling convention. Fixing the calling convention (and changing nothing else) will make the procedure work as expected.

Your task is to add instructions to **`save_identifier`** so that it follows RISC-V calling convention. Specifically, you may only add instructions that:

- Increment/decrement the stack pointer
- Put elements on the stack
- Read elements from the stack.

You may only write 1 instruction per line. You also **do not** have to fill every line. Full credit will be awarded to answers **that minimize the number of instructions** added. **Do not use pseudo-instructions.**

```
# save_identifier has no arguments or return values

save_identifier:

    addi t1, x0, -4
    add ra, ra, t1
    lw s0, 0(ra)
    andi s0, s0, 0x40
    lui t1, 0x60004
    sw s0, 0(t1)
    jalr x0, 4(ra)
```

```
save_identifier:

    ________________________

    ________________________

    addi t1, x0, -4

    ________________________

    ________________________

    add ra, ra, t1

    addi sp, sp, -4_________

    sw s0, 0(sp)____________

    lw s0, 0(ra)

    ________________________

    ________________________

    andi s0, s0, 0x40


    ________________________

    ________________________
    lui t1, 0x60004


    ________________________

    ________________________

    sw s0, 0(t1)


    lw s0, 0(sp)____________

    addi sp, sp, 4__________

    jalr x0, 4(ra)
```

Problem 6. Fast Exponentiation (15 points)

The simplest way to compute a^n, for non-negative integers a and n, would be multiplying $a \times a \times a \ldots n\ times$, resulting in $n - 1$ total multiplications.

A faster way is to do it recursively as follows:

1. If n is even, first find $a^{n/2}$ and then square it. $a^n = (a^{n/2})^2$.
2. If n is odd, first find $a^{(n-1)/2}$, square it and multiply it by a. $a^n = a \times (a^{(n-1)/2})^2$.

In this way, we group several smaller powers into one value and square it, saving a lot of multiplications.

Here is a C implementation of the function `fastPow` that takes in two integers `a` and `n` and finds a^n using the methodology described above.

```
int fastPow(int a, int n) {
   if(n == 0){
       return 1;
   }
   int res = fastPow(a, n / 2);
   res = res * res;
   if(n % 2 == 1){
     res = a * res;
   }
   return res;
}
```

In RISC-V, this function has the equivalent implementation shown on the following page. Note that we also need a `mult` procedure to perform the multiplication between two numbers as we do not have an instruction for it in RV32I from class. We have not provided the implementation for simplicity but you can assume that **`mult` does not store anything on the stack, nor does it modify the stack pointer.**
Here is a C declaration for this function.

```
int mult(int a, int b); // returns a * b
```

```
fastPow:
    bne a1, x0, fp_body
    addi a0, x0, 1
    jalr x0, 0(ra)
fp_body:
    addi sp, sp, -12
    sw ra, 0(sp)
    sw a0, 4(sp)
    sw s0, 8(sp)
    ___BLANK__
    srli a1, a1, 1
    jal ra, fastPow
    addi a1, a0, 0
    jal ra, mult         # compute res*res
    beq s0, x0, fp_done
    lw a1, 4(sp)
    jal ra, mult         # compute a*res
fp_done:
    lw ra, 0(sp)
    lw s0, 8(sp)
    addi sp, sp, 12
    jalr x0, 0(ra)
```

A. (2 points) Complete the blank in the assembly code with the correct RISC-V instruction. **Hint: this instruction will be related to line 7 in the equivalent C code.**

```
andi s0, a1, 1
```

`fastPow` is run to find the 26th power of an unknown number (i.e. n = 26 for the initial call to `fastPow`.) Execution was halted just before a call to `mult`. We get the following snapshot of registers `a0` and `a1` as well as a relevant portion of the stack.We also know that `sp` is between `0x31480` and `0x314ff`.

Again, as a reminder, the `mult` function does not modify the stack or the stack pointer at all.

```
          a0 =0x00000040, a1 =0x00000040
                Address: Data:
                      ...
                      ...
             0x31478: 0x00003420
             0x3147c: 0x00000002
             0x31480: 0x00000000
             0x31484: 0x00003420
             0x31488: 0x00000002
             0x3148c: 0x00000001
             0x31490: 0x00003420
             0x31494: 0x00000002
             0x31498: 0x00000000
             0x3149c: 0x0000723c
             0x314a0: 0x00000002
             0x314a4: 0x00000000
             0x314a8: 0x12010fff
             0x314ac: 0x12951a7f
```

B. (2 points) What is the address of the label `fp_done`?

`0x3434`

C. (1 point) What is the value of the argument `a` (base of the exponent) to the initial call to `fastPow`?

`a = 0x2`

D. (4 points) Knowing that n = 26 for the initial call to `fastPow`, what is the value stored in register `sp` at the moment the execution was halted? If you cannot determine this value, write **CAN'T TELL.**

`sp = 0x31490`

E. (6 points) We have reproduced the stack shown on the last page and added a few blanks. Fill in the values stored in the stack at the given memory addresses. If you cannot determine the value, write **CAN'T TELL.**

Address	**Data**
`0x31460`	**CAN'T TELL**
`0x31464`	**CAN'T TELL**
`0x31468`	**CAN'T TELL**
`0x3146c`	`0x00003420`
`0x31470`	`0x00000002`
`0x31474`	`0x00000001`
`0x31478`	`0x00003420`
`0x3147c`	`0x00000002`
`0x31480`	`0x00000000`
`0x31484`	`0x00003420`
`0x31488`	`0x00000002`
`0x3148c`	`0x00000001`
`0x31490`	`0x00003420`
`0x31494`	`0x00000002`
`0x31498`	`0x00000000`
`0x3149c`	`0x0000723c`
`0x314a0`	`0x00000002`
`0x314a4`	`0x00000000`
`0x314a8`	`0x12010fff`
`0x314ac`	`0x12951a7f`

Problem 7. Listicles (14 points)

Your friend is tasked with writing a list data structure `listicle.h` that wraps the C array and adds helper functions, to try to emulate a Python list. Defining a variable of type `struct list` produces a struct with the following elements:

```
struct list {
    int array[15]; // can store up to 15 items
    int size;
};
```

A. (4 points) The following functions are already implemented: **`list_append`**, which appends one integer to the end of the array, and **`list_extend`**, which adds the contents of one list to the end of another. Here are their function signatures.

```
void list_append(struct list * original, int new_value);
void list_extend(struct list * original, struct list * new_values);
```

Your friend wants you to test these functions out, and challenges you to use these two functions to complete the code below. Fill in the blank lines with function calls in order to give the intended output.

```
#include <stdio.h>
#include "listicle.h"
void main() {
    struct list a;
    struct list b;
    a.size = 0;
    b.size = 0;
    for (int i = 0; i < 5; i++) {
        __BLANK 1__
        __BLANK 2__
    }
    for (int j = 0; j < a.size; j++) {
        printf("%d ", a.array[j]);
    }
}
```

```
Intended output: 1 1 2 1 2 3 1 2 3 4 1 2 3 4 5
```

BLANK 1: `list_append(&b, i+1);`

BLANK 2: `list_extend(&a, &b);`

B. (4 points) Later, your friend comes up with the **`list_insert`** function, which can insert a `value` right in front of a particular index `i`.

```
// inserts new_value IN FRONT OF original.array[index]
// this makes new_value the new original.array[index]
void list_insert(struct list * original, int index, int new_value);
```

Assuming that the function keeps `size` correct, and that the compiler does not do any padding for structures, **specify the value of each variable listed in the table below after the following is run. Leave a box blank if it is not possible to know.**

```
#include <stdio.h>
#include "listicle.h"
void main() {
    struct list c;
    c.size = 0;
    for (int i = 0; i < 5; i++) {
        list_insert(&c, i/2, i);
    }
}
```

Variable	Value
`c.array[0]`	1
`c.array[1]`	3
`c.array[2]`	4
`c.array[3]`	2
`c.array[4]`	0
`c.array[5]`	
`c.array[6]`	
`c.size`	5

C. (6 points) It's 4am. Your friend is getting tired. Please help them write **`list_remove`**, which removes the first occurrence of `value` in the list. If a list `d` represents `[3, 1, 4, 1, 5]`, calling `list_remove(&d, 1)` should change what the list represents to `[3, 4, 1, 5]`. Other requirements include:

- If `value` cannot be found in `original`, the list stays unchanged
- The list `size` must be updated

```
void list_remove(struct list * original, int value) {
    // solution 1
    for (int i = 0; i < original->size; i++) {
        if (original->array[i] == value) {
            for (int j = i; j < original->size - 1; j++) {
                original->array[j] = original->array[j+1];
            }
            original->size--;
            return;
        }
    }
    // solution 2
    int index = -1;
    for (int i = 0; i < original->size; i++) {
        if (original->array[i] == value) {
            index = i;
            break;
        }
    }
    if (index != -1) {
        for (int i = index; i < original->size; i++) {
            original->array[i] = original->array[i+1];
        }
        original->size -= 1;
    }
    // solution 3 below, there are many other correct solutions as well!
    int found = 0;
    for (int i = 0; i < original->size; i++) {
        if (original->array[i] == value) found = 1;
        if (found) original->array[i] = original->array[i+1];
    }
    if (found) original->size = original->size - 1;
}
```

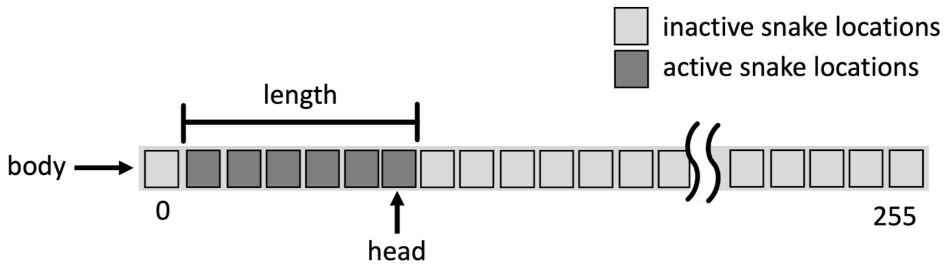
This page intentionally left blank

Problem 8: Faster Snake (8 points)

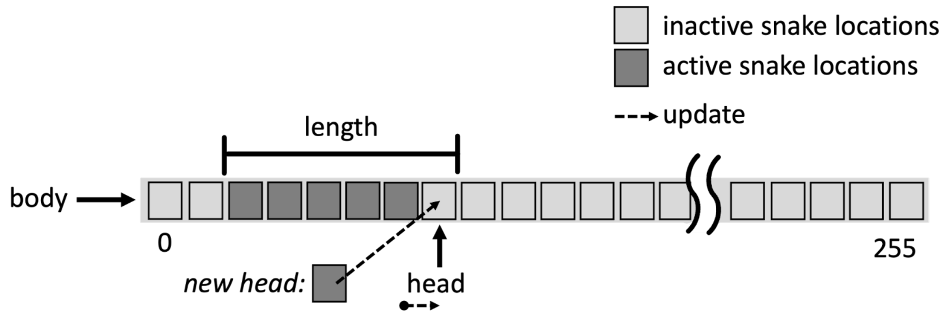
In lab 3, updating the snake required copying the entire body of the snake at every update. This is inefficient. To address this, Ben created a new `Snake` struct. In this representation, the active segments of the snake are no longer anchored at `body[0]` with the head implied at `body[0]`. Instead, the beginning of the snake is now represented by an index stored in the variable `head`.

```
// Snake struct definition
struct Snake {
    uint8_t body[256];
    uint8_t direction;
    uint8_t length;
    uint8_t head;
};
```

The diagram below shows this new snake representation, where the length is 6 and the head is at index 6.



Updating the position of the entire snake now requires only writing one new location and updating the `head`, regardless of the length of the snake, meaning the code will run much more consistently at any length. The updated snake shown below continues to have a length of 6, but the index of head is now 7.



Help Ben appropriately complete the `updateSnake` function to correctly accommodate the new `Snake` struct. For reference, the function signatures used by `updateSnake` are provided here:

```
uint8_t getX(uint8_t location);
uint8_t getY(uint8_t location);
void setX(uint8_t *location, uint8_t new_x);
void setY(uint8_t *location, uint8_t new_y);
void updateSnake(struct Snake *snake) {
    /* Function to update the snake on each step based on its direction
    Arguments:
        struct Snake *snake: pointer to snake struct
    */
    uint8_t head = snake->body[snake->head];
    uint8_t x = getX(head); //extract x value from head location
    uint8_t y = getY(head); //extract y value from head location
    if (snake->direction == up) {
        y--;
        y &= 0b111;  //same as % 8
    } else if (snake->direction == down) {
        y++;
        y &= 0b111;  //same as % 8
    } else if (snake->direction == left) {
        x++;
        x &= 0b11111; //same as % 32
    } else if (snake->direction == right) {
        x--;
        x &= 0b11111; //same as % 32
    }
    snake->head = __BLANK1__;

    setX(__BLANK2__, x); //update x portion of location reference
    setY(__BLANK2__, y); //update y portion of location reference
}
```

Complete the new implementation of `updateSnake()` by providing a correct line of code for both `BLANK1` and `BLANK2`.

Blank #:	**Line of Code:**
BLANK1	`snake->head + 1`
BLANK2	`snake->body + snake->head`