MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

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

Spring 2024, Quarter 3

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

#1 (14)	
#2 (13)	
#3 (16)	
#4 (16)	
#5 (18)	
#6 (13)	
#7 (10)	
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.

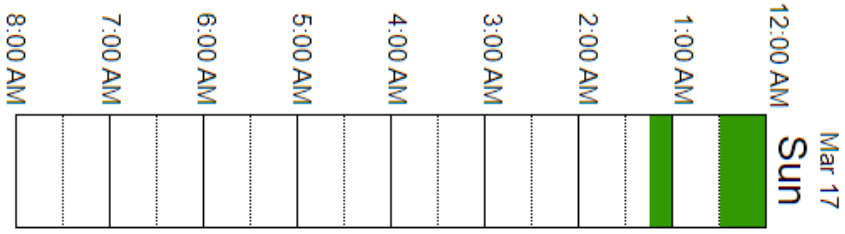
IMPORTANT: Avoid talking about and communicating the contents of this exam with other students until we have announced it is ok to do so on Piazza. Failure to do so will be considered an academic policy violation.

This page intentionally left blank

Problem 1. When Two Hydrants Meet (14 points)

The 6.1903 website logs show that most students work on assignments from 00:00 to 08:00 in 24-hour time (midnight-8:00 am in US format). You and your friends would like to meet up during this window of time to pset together. Together, you decide to encode your availabilities in that eight-hour time window in fifteen-minute increments using a 32-bit integer where the least significant bit represents the 00:00-00:15 time slot. 1 means available. 0 means not available. As an example, someone with an availability from 00:00 to 00:30 and from 01:00-01:15 would have a representation of:

`0b0000_0000_0000_0000_0000_0000_0001_0011`



For now, the platform has two users, Albert and Balbert. Their availability for the 00:00-08:00 time window is declared and initialized in `int a` and `int b` for Albert and Balbert, respectively.

A. (2 points) For the scenario below, write an equivalent C expression using only:

- **Bitwise operators (`|` , `&` , `^` , `~` , `<<` , `>>`)**
- **Constants**
- **Variables (`a, b`)**

Generate a schedule of when either Albert or Balbert are available, before 03:45	`(a | b) & 0x7FFF`

B. (2 points) After some changes, Albert's availability is shown to be `0x0000_003C`. You print out Albert's availability with the `%d` format. Convert this to 32-bit two's complement binary and printf output. *You must indicate all 32 bits.*

32-bit two's complement binary (`0b`): **`0b0000_0000_0000_0000_0000_0000_0011_1100`**
`printf("%d", a);` gives: **60**

C. (3 points) After some changes, Balbert's availability is shown to be -128. Convert this to 32-bit two's complement binary and hexadecimal encoding. *You must indicate all 32 bits.*

32-bit two's complement binary (0b): **0b1111_1111_1111_1111_1111_1111_1000_0000**
32-bit two's complement hexadecimal (0x): **0xFFFF_FF80**

D. (4 points) Write a function, `updateAvailability`, that will set or clear availability in a provided `availability`. `updateAvailability` should appropriately modify the fifteen-minute interval specified by `hour` and `quarter` (of an hour) to be `val`, without changing the availability for any other time intervals.

```
updateAvailability(&a, 4, 3, 0); //clears 04:45-05:00 for albert
updateAvailability(&b, 5, 1, 1); //sets   05:15-05:30 for balbert
```

```
void updateAvailability(int* availability, uint8_t hour,
                        uint8_t quarter, uint8_t val) {




   int avail = *availability;
   int bit = 4*hour + quarter;

   if (val) avail |= (1 << bit);
   else avail &= ~(1 << bit);

   *availability = avail;









}
```

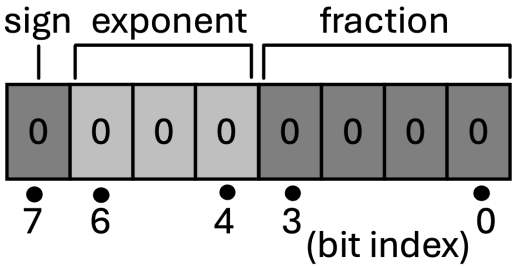
E. (3 points) Running on a 32 bit processor, the platform has been a huge hit, and now has **3 billion** users. Each user's availability is stored in a massive array database. Write a loop to count the number of users available from 00:00 to 00:15 by filling in the blanks below:

```
1 __BLANK1__ count_available(int* database, __BLANK1__ num_users) {
2     __BLANK1__ count = 0;
3     for (__BLANK1__ i = 0; i < num_users; i++) {
4         count += __BLANK2__;
5     }
6     return count;
7 }
```

Blank #:	**Line of Code:**
BLANK1	uint32_t
BLANK2	database[i] & 1

Problem 2. Baby Floats (13 points)

You've created a new data type called `float8_t`. It is similar to the standard float discussed in class, but it takes up only one byte rather than four. It uses the encoding shown below:



$$\text{Value} = (-1)^{\text{sign}} \cdot 2^{\text{exp}-3} \cdot \left(1 + \sum_{i=1}^{4} b_{4-i} \cdot 2^{-i}\right)$$

A. (7 points) Answer the following questions about values that can be represented using this format.

Can 1.0 be represented and if so what is the 8 bit encoding in binary?	**Yes, 0_011_0000**
Can 2.0 be represented and if so what is the 8 bit encoding in binary?	**Yes, 0_100_0000**
What is the largest positive value that can be represented? (decimal format, i.e. 5.5)	**0_111_1111 = 2^4(1 + ½ + ¼ + ⅛ + 1/16) = 31**
What is the most negative value that can be represented?	**1_111_1111 = -2^4(1 + ½ + ¼ + ⅛ + 1/16) = -31**
What is the smallest absolute value that can be represented?	**0_000_0000 = 2^{-3}(1) = 1/8**

B. (6 points) Consider this code below:

```
uint8_t i1 = 68;
uint8_t i2 = i1 << 1;
float8_t* fp1 = (float8_t*) &i1;
float8_t* fp2 = (float8_t*) &i2;
```

What are the final values of `*fp1` and `*fp2`? Show your work.

`*fp1`	**i1 = 68 = 0b_0100_0100** ***fp1 = $2^{1}(1 + ¼) = 5/2$**
`*fp2`	**i2 = 0b_1000_1000** ***fp2 = $-2^{-3}(1 + 1/2) = -(⅛)(3/2) = -3/16$**

This page intentionally left blank

Problem 3. Binary Search Again (16 points)

Here is a C implementation of a function `binarySearch` that searches for a value `val` in a **sorted** array `arr`. The function returns the index of `val` in the array if it exists and `-1` otherwise. The function searches for `val` in `arr` from index `start` (inclusive) to index `end` (exclusive).

```
int binarySearch(int *arr, int start, int end, int val) {
    if (start >= end) return -1;
    int mid = (start + end)/2;

    if(arr[mid] == val) {
        return mid;
    } else if(arr[mid] > val) {
        return binarySearch(arr, start, mid, val);
    } else {
        return binarySearch(arr, mid+1, end, val);
    }
}
```

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).

```
binarySearch:
    blt a1, a2, body
    stack                 # Get stack trace
    addi a0, zero, -1
    jalr zero, 0(ra)
body:
    addi sp, sp, -8
    sw ra, 4(sp)
    sw s0, 0(sp)
    stack                 # Get stack trace
    add a4, a1, a2
    srai a4, a4, 1
    slli s0, a4, 2
    add s0, a0, s0
    lw s0, 0(s0)
    blt a3, s0, left
    blt s0, a3, right
    add a0, zero, a4
    jal zero, end
left:
    add a2, zero, a4
    jal ra, binarySearch
    jal zero, end
right:
    __________________
    jal ra, binarySearch
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: `__addi a1, a4, 1_____`

We obtain the stack trace just before the first call to the function `binarySearch` occurred (**`TIME 0`** in the table). Then we run `binarySearch` to search for a value in an array. 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. Note that the execution may not have reached completion by time point TIME 5.

Address	TIME 0	TIME 1	TIME 2	TIME 3	TIME 4	TIME 5
0x3fc93f0c	0x00000004	0x00000004	0x00000004	0x00000004	0x00000004	0x00000004
0x3fc93f10	0x000007c2	0x000007c2	0x000007c2	0x000007c2	0x000007c2	0x000007c2
0x3fc93f14	0x00000123	0x00000123	0x00000123	0x00000123	0x00000035	0x00000035
0x3fc93f18	0xffffffff	0xffffffff	0xffffffff	0xffffffff	0x42107b0c	0x42107b0c
0x3fc93f1c	0x00000123	0x00000123	0x00000123	0x00000030	0x00000030	0x00000030
0x3fc93f20	0x420165b0	0x420165b0	0x420165b0	0x42107b0c	0x42107b0c	0x42107b0c
0x3fc93f24	0x3fc91000	0x3fc91000	0x00000040	0x00000040	0x00000040	0x00000040
0x3fc93f28	0x3fc91000	0x3fc91000	0x42107b00	0x42107b00	0x42107b00	0x42107b00
0x3fc93f2c	0x00000011	0x00000012	0x00000012	0x00000012	0x00000012	0x00000012
0x3fc93f30	0x00000000	0x42176bf8	0x42176bf8	0x42176bf8	0x42176bf8	0x42176bf8
0x3fc93f34	0x00000111	0x00000111	0x00000111	0x00000111	0x00000111	0x00000111

sp	0x3fc93f34	0x3fc93f2c	0x3fc93f24	0x3fc93f1c	0x3fc93f14	0x3fc93f14

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

Address: 0x___0x42176bf4____

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

Address: 0x___0x42107af8____

D. (3 points) Can you determine which value (argument `val`) is the function looking for in the array? If yes, what is the value? If not, give an interval in which the value lies.

YES/NO (circle one)
if YES provide value:_____
if NO provide interval: _0x35__< value < __0x40__

E. (3 points) Is the value the function is searching for present in the array?

YES/NO ____NO_____

Time 4 to time 5, the stack remains the same and stack pointer also remains the same. This means that in that call, nothing was allocated and stored on the stack. That means it went in the if(start >=end) branch. This means that the value is not in the array

F. (3 points) List all the values that you know are present in the array based on the information in the stack trace.

0x40, 0x30, 0x35

Problem 4: RISC-V Re-Vise (16 points)

We ran the C functions **`add_or_multiply`** and **`rewrite_array`** through a buggy C-to-RISC-V-assembly compiler, resulting in the following assembly functions. Review these assembly functions below and revise them so they align with their respective C functions. **No additional lines are permitted, only specific line rewrites. The number of incorrect lines is provided in the HINT of each subproblem.**

DO NOT USE PSEUDOINSTRUCTIONS IN ANY PART OF THIS PROBLEM!

A. (4 points)

```
// Original C function

int add_or_multiply(int x, int y) {
    if (x <= y) {
        x += y;
        return x;
    } else if (y == 3) {
        y *= 9;
        return y;
    }
    return x;
}
```

`# Assembly output`
`# HINT: There are 2 incorrect lines`
 	
`add_or_multiply:`
	For each incorrect line of the function, write the correct line of assembly code in its corresponding blank box below:

`blt a0, a1, label1`	`[1] blt a1, a0, label1`
`add a0, a0, a1`	
`jal x0, label2`	
`label1:`	
`addi a2, x0, 3`	
`bne a1, a2, label2`	

`slli a3, a1, 3`	`[2] addi a3, x0, 27`
	`[2] addi a3, a1, 24`
	`[2] ori a3, a1, 24`

`add a0, x0, a3`	`[2] add a0, a1, a3`
	`[2] addi a0, a3, 3`
	`[2] add a0, a3, a2`

`label2:`	
`jalr x0, 0(ra)`	

B. (6 points)

REMINDER: DO NOT USE PSEUDOINSTRUCTIONS IN ANY PART OF THIS PROBLEM!

```
// Original C function

void rewrite_array(int* arr, int length, int start) {
    for (int i = start; i < length; i++) {
        arr[i] = i;
    }
}
```

`# Assembly output`
`# HINT: There are 3 incorrect lines`
 	
`rewrite_array:`
	For each incorrect line of the function, write the correct line of assembly code in its corresponding blank box below:

`add a3, x0, a2`	
`label3:`	
`bge a2, a1, label4`	`[1] bge a3, a1, label4`
`slli a4, a3, 4`	`[2] slli a4, a3, 2`
`add a5, a0, a4`	

`add a0, a5, x0`	`[3] sw a3, 0(a5)`
`addi a3, a3, 1`	
`jal x0, label3`	`[1] blt a3, a1, label3`
`label4:`	
`jalr x0, 0(ra)`	

C. (6 points) Turns out our buggy translator supports two-way translation! We ran the RISC-V assembly function **`set_or_add`** through the translator and it resulted in the following C code. Review the C code below and revise it so it aligns with the **`set_or_add`** assembly function. **Again, no additional lines are permitted, only specific line rewrites.**

```
# Original assembly function

set_or_add:
   lui a1, 0x60009
   slli a2, a0, 2
   add a1, a2, a1
   lw a3, 0(a1)
   bge x0, a0, label5
   addi a4, x0, 3
   slli a5, a4, 8
   or a3, a3, a5
   sw a3, 0(a1)
   jal x0, label6
label5:
   bne a0, x0, label6
   addi a4, x0, 1
   slli a5, a4, 8
   add a3, a3, a5
   sw a3, 0(a1)
```

```
label6:
   jalr x0, 0(ra)
```

`// C output`
`// HINT: There are 3 incorrect lines`
 	
`void set_or_add(int mode) {`
	For each incorrect line of the function, write the correct line of C code in its corresponding blank box below:

`  int base = 0x60009;`	`int base = 0x60009000;`
`  int* addr = (int*) (base + (4*mode));`	
`  if (mode > 0) {`	
`    addr |= (0b11 << 8);`	`*addr |= (0b11 << 8);`
`  } else if (mode == 0) {`	
`    addr += (1 << 8);`	`*addr += (1 << 8);`
`  }` `}`	

This page intentionally left blank

Problem 5. Fibbing is Fun (18 points)

The C implementation of the Fibonacci function is provided for you below together with an implementation of this function in RISC-V assembly. Unfortunately, the person who wrote the RISC-V assembly completely forgot about calling convention. Your job is to correct the code so that it properly follows all RISC-V calling convention rules and properly implements the fibonacci function. To do this you are **only allowed to add operations of the following two forms inside the blank boxes**.

- `sw` *`reg`*`, `*`constant`*`(sp)`
- `lw` *`reg`*`, `*`constant`*`(sp)`

You may add 0 or multiple `lw` and `sw` operations per blank box. Assume that the number of bytes allocated on the stack, `X`, is correct for your implementation.

You may not change any of the registers currently being used in the implementation. Do not store any registers that are not strictly required to be stored by the RISC-V calling convention. ***For full credit, minimize the number of `lw` and `sw` instructions inside the loop.***

C Implementation:

```
int fib(int n)
{
    int a = 0;
    int b = 1;
    int c, i;
    if (n == 0)
        return a;
    for (i = 2; i <= n; i++) {
        c = sum(a, b);
        a = b;
        b = c;
    }
    return b;
}
```

Assume function `sum` is available to your code and follows RISC-V calling conventions.

C Implementation reproduced here for convenience:

```
int fib(int n)
{
    int a = 0;
    int b = 1;
    int c, i;
    if (n == 0)
        return a;
    for (i = 2; i <= n; i++) {
        c = sum(a, b);
        a = b;
        b = c;
    }
    return b;
}
```

RISC-V Implementation (spans two pages)

```
# a in a0
# b in t0
# c in s1
# i in t1
# n in a0 (note: a0 is used for both n and a)
fib:
   addi sp, sp, -X  # allocate space on stack, assume X is correct
   sw ra, 0(sp)     # save ra
   sw s1, 4(sp)     # save s1
   sw a0, 8(sp)     # save n

   li a0, 0         # a = 0
   li t0, 1         # b = 1
   sw a0, 12(sp)    # save a before overwriting a0
                    # don't save b yet
   lw a0, 8(sp)     # restore a0 = n from stack before beq
   beq a0, zero, iszero # if n == 0 goto iszero
   li t1, 2         # i = 2
   lw a0, 12(sp)    # restore a0 = a before jumping to cmp

   j cmp
```

```
loop:
   lw a0, 12(sp)     # restore a0 = a from stack
                     # can also be done after mv a1, t0

   mv a1, t0         # second arg is b

   sw t0, 16(sp)     # need to save t0 (or a1) = b on stack
   sw t1, 20(sp)     # need to save t1 = i on stack
                     # these can also go above mv a1, t0

   call sum

   lw t0, 16(sp)     # must restore t0 = b from stack
   lw t1, 20(sp)     # need to restore t1 = i from stack
                     # do not restore a0 = a

   mv s1, a0         # c = a + b
   mv a0, t0         # a = b
   mv t0, s1         # b = c
   addi t1, t1, 1    # increment i

cmp:
   sw a0, 12(sp)     # must save a0 = a
   lw a0, 8(sp)      # restore a0 = n from stack before ble

   ble t1, a0, loop # branch to loop if i <= n
   mv a0, t0         # a0 = b
end:
                     # do not restore a0 = a here!
   lw s1, 4(sp)      # restore s1
   lw ra, 0(sp)      # restore ra

   addi sp, sp, X    # restore sp
   ret

iszero:
   lw a0, 12(sp)     # need to restore a0 = a before jump to end

   j end
```

Problem 6. Stringing Dormspam (13 points)

You are writing a program that processes an email passed in using `email_text`. It determines if the email is dormspam or not by checking if `"bcc'ed to dorms"` is contained in the email text, and summarizes the email by copying the first two sentences into `first_two_sents`. Write the program by calling the appropriate string functions with the right arguments! **You must use variables as arguments.** It is guaranteed that `email_text` has at least two sentences and is less than 5,000 characters long, and the capacity of `first_two_sents` is 5,000 characters. If the function doesn't use all three arguments, leave the unused cells blank. Appendix 1 in the reference packet contains `string.h` definitions. Appendix 2 in the reference package contains a reference diagram for `strtok` operation.

```
 1 void summarizeEmails(
 2     char* email_text,
 3     int* is_dormspam,
 4     char* first_two_sents)
 5 {
 6     char filter[] = "bcc'ed to dorms";
 7     *is_dormspam = (__BLANK1__ != NULL);
 8     char *sentence_1, *sentence_2;
 9     char dot = '.';
10     char dot_str[] = {dot, '\0'};
11     sentence_1 = __BLANK2__;
12     sentence_2 = __BLANK3__;
13     // copy sentence_1 followed by sentence_2 into first_two_sents
14     __BLANK4__;
15     __BLANK5__;
16 }
```

Blank #:	Function:	Argument #1:	Argument #2:	Argument #3:
BLANK1	strstr	email_text	filter	
BLANK2	strtok	email_text	dot_str	
BLANK3	strtok	NULL	dot_str	
BLANK4	strcpy	first_two_sents	sentence_1	
BLANK5	strcat	first_two_sents	sentence_2	

Problem 7: It's a Mystery (10 points)

You are handed this mystery function to study:

```
 1 #include <stdio.h>
 2
 3
 4 void mystery1(char *input, int input_len, char *output) {
 5     uint8_t a = 0;
 6     uint8_t b = 0;
 7     int i;
 8     for (i = 0; i < input_len; i++) {
 9         char c = input[input_len - i - 1];
10         a += c;
11         if (a < c) {
12             b += 1;
13         }
14         output[i] = c;
15     }
16     output[i] = input[i-1] - (a + b);
17     output[i + 1] = 0;
18 }
```

Consider the test code below:

```
char *x = "sIR";
char y[100];
mystery1(x, 3, y);
printf("%s\n", y);            // PRINT A
```

Determine what will get printed by the line `PRINT A.` (An ASCII table is provided in the reference packet Appendix 3.)

RIsC

This page intentionally left blank