1	15/15
2	17/17
3	10/10
4	18/18
5	17/17
6	8/8
7	15/15

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.S077: Introduction to Low-level Programming in C and Assembly
Spring 2022

Name	*Athena login name*	*Score*
Anne Surs	**answers**	***100***

Please enter your name and Athena login name above. Enter your answers in the spaces provided below. Show your work for potential partial credit. You can use the extra white space and the backs of the pages for scratch work.

Problem 1. Binary Arithmetic (15 Points)

(A) (2 points) What is (0x4B & 0xF3) ^ 0x2D, where & is bitwise AND and ^ is bitwise XOR? Provide your result in both binary and hexadecimal.

Result in binary (0b):________ 0110_1110 ________

Result in hexadecimal (0x):________6E______________

(B) (3 points) What is 19 in 8-bit 2's complement notation? What is –25 in 8-bit 2's complement notation? Show how to compute 19–25 using 2's complement addition. What is the result in 8-bit 2's complement notation?

19 in 8-bit 2's complement notation (0b):_____0001_0011__________

–25 in 8-bit 2's complement notation (0b):_____1110_0111___________

19–25 in 8-bit 2's complement notation (show your work) (0b):______1111_1010__________

(C) (2 points) What range of numbers encoded using **two's complement** representation can be expressed using 6 bits? Provide your answer in decimal.

Smallest 6-bit two's complement number (in decimal):_______-32________

Largest 6-bit two's complement number (in decimal):______31_________

(D) (2 points) What range of numbers encoded using **unsigned binary** representation can be expressed using 6 bits? Provide your answer in decimal.

Smallest 6-bit unsigned binary number (in decimal):______**0**_________

Largest 6-bit unsigned binary number (in decimal):______**63**_________

(E) (2 points) Multiply 9 by 5 using unsigned binary numbers. **Show all of your work by filling in the missing rows in the table below** and provide your final answer in 8-bit two's complement notation.

0	0	0	0	1	0	0	1
0	0	0	0	0	1	0	1
				1	0	0	1
			0	0	0	0	0
		1	0	0	1	0	0
	0	0	0	0	0	0	0
0	0	1	0	1	1	0	1

9 x 5 (0b):__**0010_1101**_____________

(F) (2 Points) Translate the number 6.25 to 32-bit binary floating point representation. Then convert your answer to hexadecimal. Show your work.

6.25 = 110.01 = 1.1001 * 2^2
Sign = 0
Mantissa = 2 + 127 = 129 = 0b1000_0001
Fraction = 1001
0_10000001_10010000000000000000000
0100_0000_1100_1000_0000_0000_0000_0000

6.25 in binary floating point representation (0b):_**0100_0000_1100_1000_0000_0000_0000_0000**__

6.25 in binary floating point representation (0x):__**40C80000**_____________

(G) (2 points) Given an unknown **binary** number with digits `ghijk`, where each of `g`, `h`, `i`, `j`, and `k` could be a 0 or a 1, determine the two intermediate bitwise operations that should be performed on this number in order to end up with the result **`g1i0k`**. In other words, clear bit 1 and set bit 3 and leave the other bits unchanged. For each intermediate operation, specify both the operator and the value of the second operand (e.g., xor 11111).

First bitwise operation to perform on `ghijk`:____**| 0b01000**________________

Second bitwise operation (performed on result of first operation):___ **& 0b11101** _______________

Problem 2. (17 points) C structs

A struct called `Pixel` is defined below to represent the RGB (red, green, blue) color values of a pixel in an image.

```
struct Pixel{
  uint8_t r; //red value
  uint8_t g; //green value
  uint8_t b; //blue value
};
```

(A) (2 points) How many unique colors can be represented by this struct?

2^24. 24 bits of possibility

(B) (4 points) The luminance (perceived brightness) of a RGB value is based on the following expression:

$$L = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

Where R, G, and B are color values (meaning maximum red corresponds to $R = 255$ and minimal red corresponds to $R = 0$). Luminance L will range from 0 to 255.0 as a result.

Write a function `getLuminance` that takes in a single Pixel struct p by value and returns the Luminance (a value ranging from 0 to 255.0)

```
float getLuminance(struct Pixel p){

  return 0.299 * p.r + 0.587 * p.g + 0.114 * p.b;

}
```

(C) (5 points) The equation in part B requires (or should have required…retroactive hint) floating point calculations and those can be slower to compute. In the interest of speed, sometimes compromises are made in calculating luminance and the following equation can be a good approximation:

$$L = 0.375 \cdot R + 0.5 \cdot G + 0.125 \cdot B$$

Taking advantage of this, write a modified `getLuminance` that takes in a single `Pixel struct p` by value and returns the Luminance that is mapped over a one byte unsigned value (ranging from 0 to 255). **Your function is only allowed to use addition, subtraction, and shift operators.**

Hint: Think about how you can make these fractions with powers of 2.

```
uint8_t getLuminance(struct Pixel p){


  return (p.r)>>2+(p.r)>>3 + (p.g)>>1 + (p.b)>>3;










}
```

(D) (6 points) Consider a case where somebody makes an array of 100 pixels:

```
struct Pixel picture[100];
```

We proceed to “load” an image into this 100 pixel image by calling some function `loadPicture` on this array so that it now contains the pixel information for all 100 pixels in that image.

```
loadPicture(picture);
```

Create a function defined below that takes in an array of Pixel structs as well as the length of the array and returns a struct Pixel representing the average red, green, and blue values in the image:

```
struct Pixel getAverageColor(struct Pixel* p, int len);
```

You can use any **integer** arithmetic operations for this function.

```
struct Pixel getAverageColor(struct Pixel* p, int len){

  struct Pixel new;
  uint32_t red = 0;
  uint32_t green = 0;
  uint32_t blue = 0;
  for (int i= 0; i<len; i++){
    red += p->r;
    green += p->g;
    blue += p->b;
    p++;
  }
  new.r = red/len;
  new.g = green/len;
  new.b = blue/len;
  return new;










}
```

Problem 3. (10 points) Mystery Functions

Consider the following two functions, the second being a slight variation of the first: An ASCII table is provided to you for reference.

```c
#include<stdlib.h>
#include<stdio.h>

void mysteryFunc(const char* si, char* so){
  char* sr = si;
  char* ss = si;
  char* st = so;
  while(*sr != '\0'){
    sr++;
  }
  *(sr-si+so) = 0;
  sr--;
  while(sr != ss){
    *so = *sr;
    so++;
    sr--;
  }
  *so = *sr;
}

void mysteryFunc2(const char* si, char* so){
  char* sr = si;
  char* ss = si;
  char* st = so;
  while(*sr != '0'){ //changed!
    sr++;
  }
  *(sr-si+so) = 0;
  sr--;
  while(sr != ss){
    *so = (*sr)-1; //changed!
    so++;
    sr--;
  }
  *so = (*sr)-1; //changed!
}
```

Study these functions and determine what they do. Consider the test code on the following page:

```
  char c[] = "6.0004";
  char d[100];

  mysteryFunc(c,d);
  printf("%s\n",d); //Print 1!

  mysteryFunc2(c,d);
  printf("%s\n",d); //Print 2!
```

Determine what will get printed on the lines denoted `Print 1!` and `Print 2!`.

What gets printed at `Print 1!`

4000.6

What gets printed at `Print 2!`

-5

Problem 4. (18 points) More Pointers

Consider the four functions written below:

```
#include<stdio.h>
#include<stdlib.h>

int* A(int* a){
  return a+1;
}
int* B(int* a){
  return a-1;
}
void E(int* a, int* b){
  int* d = b;
  b = a;
  a = d;
}
void F(int* a, int* b){
  int d = *b;
  *b = *a;
  *a = d;
}
```

For each of the following code segments, specify the value of arrays x and y at the line indicated by: // LINE #.

(A) (3 points) @LINE 1:

```
const int LEN = 5;
int main(){
  int x[LEN] = {0,1,2,3,4};
  int y[LEN] = {9,8,7,6,5};
  int* c;
  int* d;
  c = A(A(x));
  *c = 3;
  // LINE 1
}
```

```
x[] = {0,1,3,3,4}


y[] = { 9,8,7,6,5}
```

(B) (3 points) @LINE 2:

```
const int LEN = 5;
int main(){
  int x[LEN] = {0,1,2,3,4};
  int y[LEN] = {9,8,7,6,5};
  int* c;
  int* d;
  c = x + 2;
  d = A(B(c));
  *d = 2*(*d);
  // LINE 2
}
```

```
x[] = { 0,1,4,3,4}


y[] = { 9,8,7,6,5}
```

(C) (3 points) @LINE 3:

```
const int LEN = 5;
int main(){
  int x[LEN] = {0,1,2,3,4};
  int y[LEN] = {9,8,7,6,5};
  int* c = x;
  int* d = y+2;
  E(c,d);
  // LINE 3
}
```

```
x[] = { 0,1,2,3,4}


y[] = { 9,8,7,6,5}
```

(D) (3 points) @LINE 4:

```
const int LEN = 5;
int main(){
  int x[LEN] = {0,1,2,3,4};
  int y[LEN] = {9,8,7,6,5};
  int* c = x;
  int* d = y;

  *A(c) = *A(d)+6;
  // LINE 4
}
```

```
x[] = { 0,14,2,3,4}


y[] = { 9,8,7,6,5}
```

(E) (3 points) @LINE 5:

```
const int LEN = 5;
int main(){
  int x[LEN] = {0,1,2,3,4};
  int y[LEN] = {9,8,7,6,5};

  *B(y+3) = *(B(y+2))*2;
  // LINE 5
}
```

```
x[] = { 0,1,2,3,4}


y[] = { 9,8,16,6,5 }
```

(F) (3 points) @LINE 6:

```
const int LEN = 5;
int main(){
  int x[LEN] = {0,1,2,3,4};
  int y[LEN] = {9,8,7,6,5};

  F(A(x),A(A(y)));
  // LINE 6
}
```

```
x[] = { 0,7,2,3,4}


y[] = { 9,8,1,6,5}
```

Problem 5. RISC-V Assembly (17 points)

(A) (3 points) What is the **hexadecimal** encoding of the instruction **`sw t1, -8(t3)`**? You can use the template below to help you with the encoding. Please show your work for partial credit.

[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

8 = 0000_0000_1000
-8 = 1111_1111_0111 + 1 = 1111_1111_1000
rs2 = t1 = x6 = 0b00110
rs1 = t3 = x28 = 0b11100
funct3 = 010
opcode = 0100011
1111_1110_0110_1110_0010_1100_0010_0011

`sw t1, -8(t3)` instruction encoding (0x):____FE6E2C23________________

For each of the following code snippets, provide the value left in each register **after executing the entire code snippet** (i.e., when the processor reaches the instruction at the `end` label), or specify **CAN'T TELL** if it is impossible to tell the value of a particular register. The code snippets are independent of each other.

(B) (3 points)

```
code_start:
  li x1, 0x23
  lui x2, 0x28
  blt x2, x1, L1
  addi x1, x1, 1
L1:
  add x1, x1, zero
end:
```

x1: (0x) ____24____________________

x2: (0x) ____28000____________________

pc: (0x) _____CAN'T TELL__________________

(C) (4 points)

```
    . = 0x100
    li x4, 0x55
    slli x5, x4, 3
    xor x4, x4, x5
    addi x6, zero, -2
  end:
```

x4: (0x) ______2FD________________

x5: (0x) _____2A8_________________

x6: (0x) ___FFFFFFFE____________

pc: (0x) _____110_________________

(D) (3 points)

```
    . = 0x100
    addi x7, zero, 0x414
    li x8, 5
    lw x9, -4(x7)
    sw x8, 4(x7)
  end:

  . = 0x410
  .word 0x01010101
  .word 0xAAAAAAAA
  .word 0x77777777
```

x9: (0x) _____01010101___________

Which address in memory is written to: (0x) _______418_______________

What value is written to memory: (0x) ______5________________

(E) (4 points) Translate the following C code to RISC-V assembly. Assume x is in register a1, y is in register a2, and z is in register a3. Only use a registers in your answer.

```
for (int i = 0; i < 10; i++) {
      x = x << 7;
      y = y | 0x55555;
      z = x - 3;
}
```

```
  li a0, 0
  li a4, 10
  li a5, 0x55555
loop:
  bge a0, a4, end
  slli a1, a1, 7
  or a2, a2, a5
  addi a3, a1, -3
  addi a0, a0, 1
  j loop
end:
```

Problem 6. RISC-V Calling Convention (8 points)

Brian and Dennis have decided to branch out from C and write a program using RISC-V assembly language. Brian has written and tested a single assembly procedure, `is_zero`, shown below, that works as expected. However, things go awry when he writes a function, `count_zeros`, that calls `is_zero` as shown below. Dennis suggests that Brian may not be following RISC-V calling convention. Brian agrees.

Brian's Code:

```
# Inputs:
# (1) value to compare to zero

is_zero:
    beq a0, zero, set
    li a0, 0
    j end
set:
    li a0, 1
end:
    ret
```

Brian's Code:

```
# count_zeros arguments:
# (1) base address of array
# (2) array length

count_zeros:
    li s0, 0
    li t0, 0
loop:
    slli t1, s0, 2
    add t1, t1, a0
    lw a0, 0(t1)
    jal is_zero
    add t0, t0, a0
    addi s0, s0, 1
    blt s0, a1, loop
end:
    mv a0, t0
    ret
```

Please add appropriate instructions into the blank spaces on the right, below, to make `count_zeros` follow calling convention. If the procedure already follows calling convention, write **NO INSTRUCTIONS NEEDED**. For full credit, you should only save registers that *must* be saved onto the stack and avoid unnecessary loads and stores.

You can assume that the program will work correctly if it follows calling convention. Do not remove or modify any of the original instructions. Also assume that `is_zero` *works as expected and follows calling convention.*

Brian's Code (repeated):

```
# count_zeros arguments:
# (1) base address of array
# (2) array length

count_zeros:
    li s0, 0
    li t0, 0
loop:
    slli t1, s0, 2
    add t1, t1, a0
    lw a0, 0(t1)
    jal is_zero
    add t0, t0, a0
    addi s0, s0, 1
    blt s0, a1, loop
end:
    mv a0, t0
    ret
```

Answer:

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



    li s0, 0
    li t0, 0



loop:

    slli t1, s0, 2


    lw a0, 8(sp)

    add t1, t1, a0




    lw a0, 0(t1)



    sw t0, 16(sp)
    jal is_zero
    lw t0, 16(sp)
```

Brian's Code (repeated):

```
# count_zeros arguments:
# (1) base address of array
# (2) array length

count_zeros:
    li s0, 0
    li t0, 0
loop:
    slli t1, s0, 2
    add t1, t1, a0
    lw a0, 0(t1)
    jal is_zero
    add t0, t0, a0
    addi s0, s0, 1
    blt s0, a1, loop
end:
    mv a0, t0
    ret
```

```
(continued on next page)



    add t0, t0, a0





    addi s0, s0, 1


    lw a1, 12(sp)


    blt s0, a1, loop



end:





    mv a0, t0


    lw ra, 0(sp)
    lw s0, 4(sp)
    addi sp, sp, 20
    ret
```

Problem 7. Stack Detective (15 points)

The following C program computes the sum of all elements in an array. The corresponding assembly program is shown on the right.

```
int arraySum(int* a, int b){
    // int *a: pointer to array
    // int b: length of array
    if (b == 1) return a[0];
    else {
        return a[0] + arraySum(a+1, b-1);
    }
 }
```

```
arraySum:
     lw a2, 0(a0)
     li a3, 1
     beq a1, a3, end

     addi sp, sp, -8
     sw ra, 0(sp)
     sw a2, 4(sp)
     addi a0, a0, 4

     _______________

     jal arraySum
     lw a2, 4(sp)
     add a2, a2, a0
     lw ra, 0(sp)
     addi sp, sp, 8

 end:
     mv a0, a2
     ret
```

(A) (1 point) What RISC-V assembly instruction should go in the blank line in order to make the assembly implementation match the C program?

Instruction: ___add a1, a1, -1____________

The program's *initial call* to `arraySum` is made from outside of the function. `arraySum` is interrupted during a recursive call, just prior to the execution of `mv a0, a2`. The diagram to the right shows the contents of the stack at this point in time.

	0x78
	0x7
SP →	0x3c
	0x2
	0x3c
	0x1
	0x3c
	0x6
	0x7324
	0x3
	0x7828

(B) (8 points) What are the (hexadecimal) values of the following variables at the time of the *initial call* to arraySum? Write "CAN'T TELL" if you cannot tell a value from the information provided.

Initial value of a: ____CAN'T TELL_______

Initial value of *a: ____3______________

Initial value of a[1]: ____6______________

Initial value of b: ____5______________

(C) (4 points) What are the (hexadecimal) values in the following registers when the program is halted? Write "CAN'T TELL" if you cannot tell a value from the information provided.

Current value of ra (0x): ____3C___________

Current value of a2 (0x): ____CAN'T TELL___________

(D) (2 points) What is the hexedecimal address of the "call arraySum" instruction that made the *initial call* to arraySum?

Address of initial call (0x): ___0x7320_____________

END OF QUIZ!