MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

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

Fall 2026 Exam 1

Name: *Saul U. Shuns*	**Kerberos:** *saulushuns*
	MIT ID #: *123456789*

#1 (16)	**16**
#2 (14)	**14**
#3 (13)	**13**
#4 (12)	**12**
#5 (17)	**17**
#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. Binary Encoding (16 points)

Consider the 8-bit binary value: 0b10001101

A. (2 Points)

What is the hexadecimal representation of this 8-bit binary number?	`0x8D`

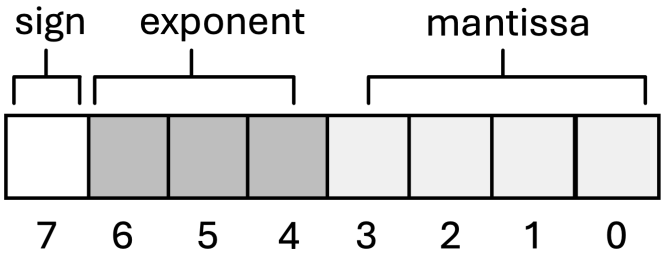
B. (2 Points)

What is the decimal equivalent of this number if it is treated as an unsigned binary number?	**`141`**

C. (4 Points)

What is the decimal equivalent of this number if it is treated as a signed two's complement number?	**`-115`**
What would be the 12-bit binary representation of this signed two's complement number?	**`0b1111_1000_1101`**

D. (8 Points) Assume a modified 8-bit version of floating point (`float8_t`) with a sign bit, 3 exponent bits, and 4 mantissa bits (illustrated below). Answer the following questions using this new data type.



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

What is the decimal equivalent of `0b10001101` if it is treated as a `float8_t` as defined above? You can leave your answers unsimplified e.g. `(1/2 + 1/16) · 2^5`	`-⅛*(1+1/2+1/4+1/16)` Or `-29/128`
If you declare: `float8_t x = 8.5`, what would be the 8-bit binary representation of `x`?	`0b0110_0001`
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) · 2^5`	`2^4*(1+1/2+1/4+1/8+1/16)`
Can you represent the number 8.0 with this format?	`yes`
Can you represent the number 8.25 with this format?	`no`

Problem 2. Pointy Pointers (14 points)

Reminder:
`printf("%d\n", value)` prints `value` in decimal.
`printf("%x\n", value)` prints `value` in hexadecimal.
`printf("%p\n", pointer)` prints `pointer` in hexadecimal.
`printf("%s\n", string)` prints `string`.
`printf("%c\n", character)` prints `character`.

A. (3 Points) Consider the following code. What gets printed?

```
uint8_t x[] = { 0xf0, 0xff };
int16_t *y = (int16_t*) x;
printf("*y = %d\n", *y);
```

```
"*y = -16\n"
```

B. (3 Points) Consider the following code. What does it print?

```
int x = 1;
int y[4] = {5, 22, 10, 32};
int *x_ptr = &x;
int **x_ptr_ptr = &x_ptr;
*x_ptr_ptr = y+x;
printf("x_ptr[2] = %d\n", x_ptr[2]);
```

```
"x_ptr[2] = 32\n"
```

C. (4 Points) Consider the following code. What does it print?

```
void do_stuff(char *array) {
    array += 2;
    *array = 'W';
}

void app_main() {
    char array[] = "abcdefg";
    do_stuff(array);
    printf("array is '%s'\n", array);
}
```

"array is 'abWdefg'\n"

D. (4 Points) Consider the following code. What does it print?

```
int x[] = { 2, 4, 6, 8, 10, 12, 14, 16 };
int *y = x + 1;
y += *y;
printf("*y = %d\n", *y);
```

"*y = 12\n"

Problem 3. Bitty Bits (13 points)

A. (4 Points) Given a `uint32_t a`, provide a C expression **comprised of only bitwise operators** that evaluates to:

`1` if `a` is even and `0` if `a` is odd.	`~a & 1`
`1` if `a` is cleanly divisible by `8` and `0` if `a` is not, where "cleanly divisible" means there is a remainder of 0 when `a` is divided by `8`.	`~(a | (a>>1) | (a>>2)) & 1`

B. (4 points) You are given the following skeleton of a function `expand` that takes as input a variable `x` and returns the expansion of `x`, which is defined to be the 32-bit unsigned integer where bit `2*i` is equal to bit `i` of `x` and bit `2*i+1` is 0. Complete the implementation of `expand` by filling in the missing line(s):

```
uint32_t expand(uint32_t x) {
    uint32_t r = 0;
    for (int i = 0; i < 16; i++) {


        r = r | ((x & (1 << i)) << i);

        //or

         r |= ((x >> i) & 1) << (2*i);

        //or

         r |= ((x >> i) & 1) << (i<<1);




    }
    return r;
}
```

C. (5 points) You are given the following skeleton of a function `shuffle` that takes as input a variable `x` and returns the shuffle of `x`, which is defined to be the 32-bit unsigned integer where bit `2*i` is equal to bit `i` of `x` and bit `2*i+1` is equal to bit `i + 16` of `x`. Complete the implementation of `shuffle` by filling in the missing line(s):

```
uint32_t shuffle(uint32_t x) {
    uint32_t r = 0;
    for (int i = 0; i < 16; i++) {









        r = r | (((x & (1 << i)) >> i) << 2*i);
        r = r | (((x & (1 << (i+16))) >> (i+16)) << ((2*i) + 1));

        //or more properly (but accept either):
        r = r | (((x & (1u << (i+16))) >> (i+16)) << ((2*i) + 1));
    }
    return r;
}
```

Problem 4. Back to the Lab Again (12 points)

A. (3 Points) For the postlab 3 extension, Andy wants to make the screen flash. To help accomplish this, he writes a function that sets every pixel in the grid on. Assume all called functions behave properly. Below is his implementation:

```
uint32_t screen_buffer[8];
void fillScreen() {
  for (uint8_t loc = 0; loc < 256; loc++) {
      setPixel(loc, 1); //updates screen_buffer at location loc with a 1
  }
  drawBuffer(); //renders the contents of screen_buffer on the display
  printf("Display updated!\n");
  return;
}
```

He runs the function on his lab kit but the screen did not update and nothing printed to the serial monitor. What is wrong with Andy's implementation and how would you fix it?

`loc` is a `uint8_t` which can only represent the values `[0, 255]`. After the iteration where `loc` reaches `255`, it overflows to `0`. This means the for loop's condition (`loc < 256`) will never be `false` because `loc` can never reach above `255`. make `loc` a `uint16_t` or `int`

B. (4 Points) Bella writes a helper function called `resetSnake` to reset the snake for postlab 3.

```
/* For reference
struct Snake {
    uint8_t body[SCREEN_COLS*SCREEN_ROWS];
    uint8_t direction;
    uint8_t length;
};
*/

void resetSnake(struct Snake snake) {
  struct Snake* snake_ptr = &snake;
  for (int i=0; i<3; i++){
    setX(&snake_ptr->body[i], 5);
    setY(&snake_ptr->body[i], 3-i);
  }
  snake_ptr->length = 3;
  snake_ptr->direction = left;
}

void app_main (){

  // Initialize snake
  struct Snake snake;
  resetSnake(snake);

  while(1) { // Game loop
  // ...
  }
}
```

After giving herself a pat on the back, Bella turns on her ESP32 and realizes the `resetSnake` function does not work properly. Why did the snake not properly reset?

`resetSnake` makes a copy of the input snake struct so the pointer inside the `resetSnake` function is a pointer to the copy, not a pointer to the original input snake. This means when the function operates on `snake_ptr`, it is not actually modifying the input snake.

This page intentionally left blank

C. (5 Points) Carl has an idea for an extension. He wants to add bombs that damage the head of the snake. He asks for your help implementing the function `damageSnake`. This function takes a pointer to a `Snake` struct and removes the first `dmg` elements of the snake's body. The first element of the remaining body becomes the new head of the snake. You may assume `dmg` is strictly less than the snake's current length. Assume `direction` does not change.

```
/* For reference
struct Snake {
    uint8_t body[SCREEN_COLS*SCREEN_ROWS];
    uint8_t direction;
    uint8_t length;
};
*/

void damageSnake(struct Snake *snake, uint32_t dmg) {




    for (int i = 0; i < snake->length-dmg; i++) {
        snake->body[i] = snake->body[i+dmg];
    }

    snake->length = snake->length - dmg;









}
```

Problem 5. Deep Structural Problems (17 Points)

Consider the following program:

```
struct MyStruct {
  int array[5];
  int* ptr;
};

void app_main() {
    struct MyStruct myobj = {
      .array = {6, 5, 4, 7, 2}
    };

    myobj.ptr = myobj.array + 1;

    printf("%p\n", &myobj.array);   // Outputs 0x3fc93f44
    printf("%p\n", &myobj.ptr);     // Outputs 0x3fc93f58

    // PAUSE

    printf("%x\n", myobj.array[2]);  // third printf
    printf("%x\n", myobj.ptr[3]);    // fourth printf
    printf("%x\n", myobj.ptr[4]);    // fifth printf
}
```

Suppose the program is run until the line annotated with `PAUSE`. The outputs of the first and second `printf` calls are:

```
0x3fc93f44
0x3fc93f58
```

A. (4 Points)

What is `sizeof(myobj.array)`?	20
What is `sizeof(myobj.ptr)`?	4

B. (6 Points) Fill in the following 32-bit words in memory as hexadecimal values.

Address:	Value:
0x3fc93f44:	0x 6
0x3fc93f48:	0x 5
0x3fc93f4c:	0x 4
0x3fc93f50:	0x 7
0x3fc93f54:	0x 2
0x3fc93f58:	0x 3fc93f48

C. (3 Points) Assume the program continues running past `PAUSE` and completes execution.

What does the third `printf` output?	**4**
What does the fourth `printf` output?	**2**
What does the fifth `printf` output?	**3fc93f48**

D. (4 Points)

Write a function `link` that takes in three arguments:

- `struct MyStruct *dst`
- `struct MyStruct *src`
- `uint32_t i`

The function should point the `ptr` of `dst` to the `i`th element of the array in `src` such that the following behavior is achieved:

```
/* For reference/reminder:
struct MyStruct {
  int array[5];
  int* ptr;
};
*/

struct MyStruct a = { .array = {6, 5, 4, 7, 2} };
// a.array is at 0x3fc93f44 for ref
struct MyStruct b = { .array = {1, 3, 8, 9, 5} };
// b.array is at 0x3fc93f2c for ref

link(&a, &b, 2);
printf("%p\n", a.ptr);      // 0x3fc93f34
printf("%d\n", *a.ptr);     // 8
printf("%d\n", a.ptr[1]);   // 9

link(&b, &a, 0);
printf("%p\n", b.ptr);      // 0x3fc93f44
printf("%d\n", b.ptr[3]);   // 7
```

```
void link(struct MyStruct *dst, struct MyStruct *src, uint32_t i) {

  dst->ptr = &src->array[i];

}
```

Problem 6. String Processing (16 points)

A. (8 points) Write a function `stringSplitter` that splits the null-terminated string `source` into its individual sentences in-place. In other words, your function should:

- Modify `source` such that each sentence becomes its own null-terminated string after the split.
- Store a pointer to each new string in the `dest` array.
- Return the total number of strings that were identified and split.

The inputs to the function will be:

- `source`: a null-terminated string containing at least one sentence. This string is to be split and modified by the function.
- `dest`: a sufficiently large array of pointers the function will use to store the pointers to the individual null-terminated strings in `source`.

Some notes:

- Assume all periods `'.'` in `source` will be followed by a single space `' '` except for the last period, which ends the string. Your solution should keep the period in each split string, but not the space.
- **Your solution must use `strstr` from `string.h`.**

```
uint32_t stringSplitter(char * source, char ** dest) {

     // YOUR CODE BELOW

    char * period_ptr = strstr(source, ". ");
    char * src_ptr = source;
    uint32_t num_strings = 0;

    while (period_ptr != NULL){
        *(period_ptr+1)  = '\0';

        dest[num_strings] = src_ptr;
        src_ptr = period_ptr+2;
        period_ptr = strstr(src_ptr, ". ");
        num_strings+=1;
    }

   dest[num_strings] = src_ptr;
   num_strings++;

   return num_strings;




}
```

B. (8 points) On the next page, write a function named `textFormatter` that takes in an array of non-empty C-strings (`source`) and concatenates modified versions of these strings into `dest` starting at `dest[0]`. The following changes should be applied to the contents from each string in `source` before being appended to `dest`:

- Capitalize the first character if it is a lowercase letter; leave it the same otherwise.
- Modify all other alphabetic characters to be lowercase if they are not already. Non-alphabetic characters should remain unchanged.

Write the function such that each string from `source` is followed by a semicolon `';'` once it has been appended to `dest` (including the final string). The function input `num_strings` specifies the number of strings in `source` to be appended to `dest`.

Note that you should not modify `source` itself. After the function has been run, `dest` should contain a single, properly formatted string containing the concatenated, modified strings. You may assume that `dest` points to an array that has a large enough capacity to store all necessary characters.

You *MAY NOT* assume that string.h has been included for you in Part B.

```
void textFormatter(const char ** source, char * dest,
                                uint32_t num_strings) {

     // YOUR CODE BELOW
    int dest_indx = 0;
    for (int i = 0; i< num_strings; i++){
        if ((0x61 <= source[i][0]) && (source[i][0]<= 0x7A)) {
            dest[dest_indx] = source[i][0] - 32;
        } else {
            dest[dest_indx] = source[i][0];
        }
        dest_indx++;
        int j = 1;
        while (source[i][j] != '\0'){
            char curr_char = source[i][j];
            if ( (0x41 <= curr_char) && (curr_char<= 0x5A)) {
                curr_char = curr_char + 32;
            }
            dest[dest_indx] = curr_char;
            j++;
            dest_indx++;
        }
        dest[dest_indx] = ';';
        dest_indx++;
    }
    dest[dest_indx] = 0;









}
```

This page intentionally left blank

Problem 7. A Candy Colored Clown They Call the Sandman (12 points)

Oh. Hey. How are you doing? Sorry, the exam isn't done yet. I know we want it to be, but that's just not what's gonna happen. Here's a snapshot of memory:

```
Address         Word
0x00001000    0x00001024
0x00001004    0xFFFFFFFE
0x00001008    0x00000007
0x0000100C    0x0000000F
0x00001010    0x6C6C6548
0x00001014    0x0000216F
0x00001018    0x00001010
0x0000101C    0x0000101A
0x00001020    0x41424344
0x00001024    0x00000000
0x00001028    0x00001020
0x0000102C    0xDEADBEEF
0x00001030    0x0000102C
0x00001034    0x12345678
0x00001038    0x00001014
0x0000103C    0xCAFEF00D
```

This code runs with the memory in this state.

```
int      *p  = (int *)     0x00001004;
uint8_t  *b  = (uint8_t *) 0x0000102C;
char     *s  = (char *)    0x00001010;
char    **pp = (char **)   0x00001018;

printf("%d\n",  *p);        // QA
printf("%d\n",  p[2]);      // QB
printf("%x\n",  b[1]);      // QC
printf("%s\n",  s + 2);     // QD
printf("%p\n",  pp+4);      // QE: %p prints a ptr in hex (see prob. 5)
printf("%s\n",  *(pp+4));   // QF
```

What gets printed for the six `printf` calls?

QA `-2`	**QB** `15`
QC `be or BE`	**QD** `llo!`
QE `0x1028 or 0x00001028`	**QF** `DCBA`

This page intentionally left blank