University of California, Berkeley – College of Engineering
Department of Electrical Engineering and Computer Sciences
Summer 2018 Instructors: Steven Ho, Nick Riasanovsky 2018-07-03

CS61C MIDTERM 1

Last *Name (Please print clearly)*	Petra	
First *Name (Please print clearly)*	Perfect	
Student ID Number		
Circle the name of your Lab TA	**Damon Jonathan Sean Sruthi Emaan Suvansh Sukrit**	
Name of the person to your: Left | Right		
All my work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61C who haven't taken it yet. ***(please sign)***		

Instructions

- This booklet contains 9 pages including this cover page. **The back of each page is blank and can be used for scratch work, but will not be graded** (i.e. not even scanned into Gradescope).
- Please turn off all cell phones, smartwatches, and other mobile devices. Remove all hats, headphones, and watches. Place *everything* except your writing utensil(s), cheat sheet, and beverage underneath your seat.
- You have 80 minutes to complete this exam. The exam is closed book: no computers, tablets, cell phones, wearable devices, or calculators. You are allowed one page (US Letter, double-sided) of *handwritten* notes.
- There may be partial credit for incomplete answers; write as much of the solution as you can.
- Please write your answers within the boxes and blanks provided within each problem!

Question	**1**	**2**	**3**	**4**	**5**	**Total**
Possible Points	12	16	20	20	12	80

If you have the time, **feel free to doodle on the front page!**

This page is intentionally blank. Draw here if you are bored.

Question 1: Number Representation and Floating Point (12 pts)

Given the following bit string 0b1111 1100, answer the following questions:

1) What is this bitstring's value if it was interpreted as an **unsigned number**?
(2^8 - 1) - 3 = 252

2) What is this bitstring's value if it was interpreted in **two's complement**?
(Flip all the bits and add 1) * -1 = (0b0000 0011 + 1) * -1 = -4

3) Suppose the bit string was represented as **fixed point** where the bits following the dot (.) represent the base (2) to the power of a negative exponent. What number does the bitstring 0b1111.1100 represent?
0b1111 = 15; 0b.1100 = 0.5 + 0.25 = 0.75 -> 15.75

4) Now let's devise a scheme for interpreting fixed point numbers as positive or negative values. Complete the following sentence:
Given an 8-bit fixed point bitstring 0bXXXX.XXXX with a value of Y, in order to compute -Y, we must flip all the bits of Y and add:
The idea in two's complement is that you add 0b00...001 to the flipped bitstring.
0b0000.0001 in fixed point has a value of 1/16. 1/16

5) What is the value of 0b1111.1100 given the **two's complement fixed point** representation described above?
. (Flip all the bits and add 1/16) * -1 = (0b0000.0011 + 1/16) * -1 = -0.25

6) You are given the following field breakdown and specifications of an 8-bit floating point, which **follows the same rules** as standard 32-bit IEEE floats, except with different field lengths:

Sign: 1 bit
Exponent: 3 bits
Significand: 4 bits

Exponent Value	Significand Value	Floating Point Value
Smallest	Zero, Non-Zero	±0, Denormalized
Largest	Zero, Non-zero	±Infinity, NaN

What is the floating point value of 0b1111 1100:

Exponent field is the largest exponent (0b111) and the significant is non-zero-> NaN

7) We now modify the floating point description in part 6 so that the exponent field is now in **two's complement** instead of in bias notation. Compute the floating point value of 0b1111 1100.

Exponent: 0b111 = -1, Signif: 0.75 = (-1) x 2^-1 x 1.75 = -0.875

Question 2: C Memory Management (16 pts)

```
char *mood;
char *copy_message (char *msg) {
     char *x = malloc (sizeof (char) * (strlen (msg) + 1));
     strncpy (x, msg, strlen (msg));
     x[strlen (x)] = '/0';        / **** 6 ****/
     return x;
}
void print_int (int *p) {
     printf ("%d\n", *p);                /**** 7 ****/
}
void print_msg (char *str) {
     char *cpy = calloc (strlen (str) + 1, 1);
     strncpy (cpy, str, strlen (str));
     printf ("%s\n", cpy);          /**** 8 ****/
}
char *a () {
     char res[7] = " rules";
     return res;
}
char *b () {
     char *var = "cs 61c";
     return var;
}
void c () {
     printf ("%s\n", a ());       /**** 9 ****/
     printf ("%s\n", b ());      /**** 10 ****/
}
int main () {
     int y;
     mood = malloc (3);
     strcpy (mood, "hi");
     copy_message (mood);
     print_int (&y);
     print_msg (mood);
     c ();
}
```

Each of the following values below evaluates to an address in the C code on the previous page. Select the region of memory that the address points to (notice each function is called exactly once).

1.	`mood`	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap
2.	`&mood`	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap
3.	`var`	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap
4.	`res`	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap
5.	`print_int`	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap

On the previous page there are comments on lines with numbers from 7-11. Each of these refers to a line of code that requires a dereference of a pointer to be performed. What we want to do is characterize if these memory accesses are legal c. We will use the following terminology

Legal: All addresses dereferenced are addresses that the program is allowed to read.

Initialized: Is there actual meaningful data in contents (data at each address) or is it garbage.

Always Illegal: This line will always dereference an address the program doesn't have explicit access to

Possibly Legal: The operation could result in only dereferences of legal addresses but it's also possible that in other runs on the program illegal accesses occur.

For each of lines that have the numbered comment select the best answer from

- A. Legal and Initialized
- B. Legal and Uninitialized
- C. Possibly Legal
- D. Illegal

For example for question 6 you should answer about the line with the /**** 6 ****/ comment from when the program runs.

6.	Ⓐ	Ⓑ	Ⓒ	Ⓓ
7.	Ⓐ	Ⓑ	Ⓒ	Ⓓ
8.	Ⓐ	Ⓑ	Ⓒ	Ⓓ

9. Ⓐ Ⓑ Ⓒ Ⓓ

10. Ⓐ Ⓑ Ⓒ Ⓓ

Question 3: RISC-V Coding (20 pts)

---

1. Fill in the following RISC-V code so that it properly follows convention. Assume that all labels not currently in the code are external functions. You may not need all the lines provided.

```
Pro:
      addi  sp sp -16
      sw    ra 0(sp)
      sw    s1 4(sp)
      sw    s2 8(sp)
      sw    s3 12(sp)
Body:
      mv s1 a0
      jal ra foo
      mv s2 a0
      addi a0 x0 6
Loop:
      beq a0 x0 Epi
      addi a0 a0 -1
      mv s3 a0
      jal ra foo
      addi s2 s2 a0
      mv a0 s3
      j Loop
Epi:
      lw    s3 12(sp)
      lw    s2 8(sp)
      lw    s1 4(sp)
      lw    ra 0(sp)
      addi sp sp 16
      jr ra
```

2.

```
foo:
            slli  t6 a0 2
            sub   sp sp t6
            mv    t4 sp
            sw    zero 0(t4)
            addi  t1 zero 1
L1:         bge   t1 a0 Next
            andi  t2 t1 1
            slli  t3 t1 2
            add   t3 t3 t4
            sw    t2 0(t3)
            addi  t1 t1 1
            j     L1
Next:       mv    t1 zero
            mv    t2 zero
            slli  a0 a0 2
L2:         bge   t1 a0 End
            add   t3 t4 t1
            lw    t3 0(t3)
            add   t2 t2 t3
            addi  t1 t1 4
            j     L2
End:        mv    a0 t2
            add   sp sp t6
            jr    ra
```

Translate the RISC-V Assembly on the left into C code to complete the function foo:

```
unsigned foo(unsigned n) {
  unsigned arr[n];
  unsigned total = 0;
  unsigned *ptr = arr;
  ptr[0] = 0;
  for (int i = 1; i < n; i++) {
    ptr[i] = i & 1;
  }
  for (int i = 0; i < n; i++) {
    total +=  ptr[i];
  }
  return total;
}
```

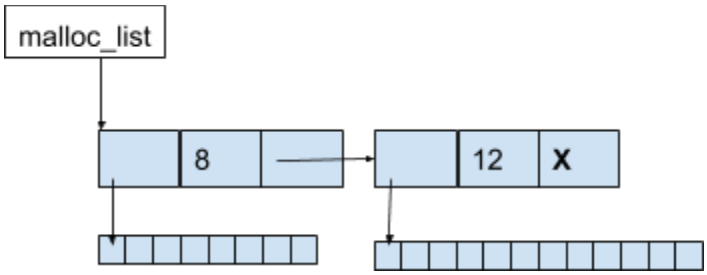
Question 4: C Coding (20 pts)

Recall that in C, pointers have no sense of their own bounds. Nick doesn't like this, so he decided to replace malloc and free with helper functions to keep track of memory bounds. To do so he decides to create wrapper functions for malloc and free that he will call instead of just malloc or free. To do so he creates a struct:

```
typedef struct malloc_node {
      void *data_ptr;
      size_t length;
      struct malloc_node *next;
} m_node;
```

Example:

```
user_malloc (12);
user_malloc (8);
```



This holds the value of the malloc'ed pointer along with the original length requested in bytes as a linked list node. He also creates a global variable:

```
m_node *malloc_list; // Assume this is initialized to NULL
```

Using these globally accessible structures you will implement a form of malloc and free that can keep track of the bounds on heap pointers. For this question assume all mallocs succeed. For both questions you may not need all lines.

/ The user wrapper function for malloc. This function is called instead of malloc when asking for N bytes of heap memory. This function should make a call to malloc to produce this memory, but it should also add a node to malloc_list to store the additional size information. A visual is shown above.*
*Hint: You may find it easier to add an element to the front of a linked list rather than the end . */*

```
void *user_malloc (size_t n) {
      m_node *node = malloc (sizeof (m_node));
      node->data_ptr = malloc (n);
      node->length = n;
```

```
     node->next = malloc_list;
     malloc_list = node;
     return node->data_ptr;
}
```

Finally we need an implementation of free which also frees this metadata.

```
void user_free (void *ptr) {
     if (! remove_ptr (& malloc_list, ptr)) {
          illict_free ();       // Assume this handles any errors from
illegal
                                // attempts to free.
     }
}
```

Implement remove_ptr. This should free the PTR, remove the metanode node from the linked list and cleanup any metadata if the free is legal. It should return true if it was successful and otherwise false.

/ Takes in a NODE_PTR which points to a part of the list and a ptr and if the node stores the info about pointer handles any appropriate freeing and removes the node holding that pointer from the list. */*

```
bool remove_ptr (m_node **node_addr, void *ptr) {
     if (node_addr && *node_addr && ptr) {
          if ((*node_addr)->data_ptr != ptr) {
               return remove_ptr (&(*node_addr)->next, ptr);
          }
          m_node *temp = (*node_addr);
          free (temp->data_ptr);
          *node_addr = temp->next;
          free (temp);
          return true;
     }
     return false;
}
```

Nick is considering revising the `struct malloc_node` definition by adding another field (bolded for your convenience):

```
typedef struct malloc_node {
     void *data_ptr;
     size_t length;
     struct malloc_node *next;
     size_t num_bytes;
} m_node;
```

Given this new struct definition, the value returned by `sizeof(next)` changes. Ⓣ Ⓕ

Question 5: RISC-V Instruction Formats (12 pts)

You are given the following RISC-V code:

```
Loop:     andi  t2 t1 1
          srli  t3 t1 1
          bltu  t1 a0 Loop
          jalr s0 s1 MAX_POS_IMM
          ...
```

1) What is the value of the **byte offset** that would be stored in the immediate field of the `bltu` instruction?

   Two instructions away = -8 bytes

2) What is the binary encoding of the `bltu` instruction? Feel free to use the following space for scratch work—it will not be graded. Put your final answer in hexadecimal.

   31 0

   1111 1110 1010 0011 0110 1100 1110 0011

   0xFEA36CE3

As a curious 61C student, you question why there are so many possible opcode, but only 47 instructions. Thus, you propose a revision to the standard 32-bit RISC-V instruction formats where **each instruction has a unique opcode (which still is 7 bits)**. You believe this justifies taking out the funct3 field from the R, I, S, and SB instructions, allowing you to allocate bits to other instruction fields **except the opcode field**.

1) What is the largest number of registers that can now be supported in hardware?

Now that we eliminate the funct3 field, we have 3 bits at our disposal for the R, I, S, and SB instructions. Since the R type instruction has three registers fields, we could add a bit to each of those fields and thus increase the number of registers to 64.

2) With the new register sizes, how far can a `jal` instruction jump to **(in halfwords)**?

Since register fields now have 6 bits, then in the UJ type, the jump immediate field is now reduced to 19 bits instead of 20. Thus, we can only jump to $\pm(2^{18})$ halfwords

`jal` jump range: [-2^{18}, $2^{18} - 1$]

3) Assume register `s0` = 0x1000 0000, `s1` = 0x4000 0000, `PC` = 0xA000 0000. Let's analyze the instruction:

```
jalr s0, s1, MAX_POS_IMM
```

where `MAX_POS_IMM` is the maximum possible positive immediate for `jalr`.

Once again, use the new register sizes from part 1. After the instruction executes, what are the values in the following registers?

Once again, we know that rd and rs1 fields are now 6 bits. `jalr` is an I-type instruction, so we take out the funct3 bits but we give each of rd and rs1 fields 1 bit, meaning we have 1 bit leftover to give to the immediate field. Thus, we now have a 13-bit immediate. Thus, the maximum possible immediate a jalr instruction can hold is $+2^{12} - 1$ halfwords away, which is represented as 0b0 1111 1111 1111, which is 0x0FFF.

s0 is the linking register—it's value is PC + 4
s1 does not get written into so it stays the same
PC = R[s1] + 0x0FFF

s0 = 0xA000 0004 s1 = 0x4000 0000 PC = 0x4000 0FFF