CS61C Midterm

After the exam, indicate on the line above where you fall in the emotion spectrum between “sad” & “smiley”...

<table>
<thead>
<tr>
<th>Last Name</th>
<th>Answer Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Name</td>
<td></td>
</tr>
<tr>
<td>Student ID Number</td>
<td></td>
</tr>
<tr>
<td>Login</td>
<td>cs61c-</td>
</tr>
<tr>
<td>Login First Letter (please circle)</td>
<td>a b c d e f g h i j k l m</td>
</tr>
<tr>
<td>Login Second Letter (please circle)</td>
<td>a b c d e f g h i j k l m n o p q r s t u v w x y z</td>
</tr>
<tr>
<td>The name of your SECTION TA (please circle)</td>
<td>Aaron Alex Brian David Matt Michael Valerie</td>
</tr>
<tr>
<td>Name of the person to your Left</td>
<td></td>
</tr>
<tr>
<td>Name of the person to your Right</td>
<td></td>
</tr>
</tbody>
</table>

All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61C who have not taken it yet. (please sign)

Instructions (Read Me!)

- Don’t Panic!
- This booklet contains 7 numbered pages including the cover page. Put all answers on these pages; don’t hand in any stray pieces of paper.
- Please turn off all pagers, cell phones & beepers. Remove all hats & headphones. Place your backpacks, laptops and jackets at the front. Sit in every other seat. Nothing may be placed in the “no fly zone” spare seat/desk between students.
- Question 0 (1 point) involves filling in the front of this page and putting your name & login on every front sheet of paper.
- You have 180 minutes to complete this exam. The exam is closed book, no computers, PDAs or calculators. You may use one page (US Letter, front and back) of notes and the green sheet.
- There may be partial credit for incomplete answers; write as much of the solution as you can. We will deduct points if your solution is far more complicated than necessary. When we provide a blank, please fit your answer within the space provided. You have 3 hours...relax.

<table>
<thead>
<tr>
<th>Question</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minutes</td>
<td>1</td>
<td>36</td>
<td>36</td>
<td>36</td>
<td>36</td>
<td>36</td>
<td>180</td>
</tr>
<tr>
<td>Points</td>
<td>1</td>
<td>15</td>
<td>14</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>75</td>
</tr>
<tr>
<td>Score</td>
<td>1</td>
<td>15</td>
<td>14</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>75</td>
</tr>
</tbody>
</table>
Name: ___________ Answers ___________________ Login: cs61c-____

**Question 1: Is this the best midterm in memory? No, we freed it! (15 pts, 36 min)**

a) I have N bits to represent data, and every bit pattern has a unique meaning. I want to represent 3 times as many things. How many more bits do I need? 2

b) We are given two nibbles, A (=0xF) and B (=0b0010), and we wish to calculate their SUM = A + B. We only have a nibble to store the SUM result. What is SUM if all three nibbles (A, B, SUM) were...

<table>
<thead>
<tr>
<th>SUM (single hex character)</th>
<th>Decimal number SUM encodes</th>
<th>Was there overflow?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xD</td>
<td>-5</td>
<td>no</td>
</tr>
<tr>
<td>0x2</td>
<td>2</td>
<td>no</td>
</tr>
<tr>
<td>0x1</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>0xA</td>
<td>3</td>
<td>No</td>
</tr>
</tbody>
</table>

Algorithm: SUM = encode( decode-into-decimal(A) + decode-into-decimal(B) )

S/M:-7+2=-5==>1101, 1s:-0+2=2==>0010, us:15+2=17==>10001, 2s:-1+2=1==>0001, b:8+-5=3==>1010

c) Put the following in chronological order. We’ve started it for you.

6 Code and Data from various places are stitched together.
1 A CS61C student is assigned a project that implements big_nums.
9 Execution begins at main.
2 The student writes his or her code in C.
5 Link tables are produced.
4 MAL is translated into TAL.
8 Static, code, and global space are reserved/initialized in memory.
3 The student’s C code is translated into MIPS.
7 Links are “edited”

d) Assume we have just enough bits to byte-address 51210 zebibytes. We want to define some number of the most-significant bits to encode 910 x 250 things, and some number of the least-significant bits to encode 2,00010 things. How many things can we encode with the remaining bits? Use IEC language, like “16 mebibithings”. Show your work.

79-54-11=14 bits
16 kibithings

e) For every line of code on the right, we want to know if any memory is used, and if so, where and how much. If zero, leave it blank.

<table>
<thead>
<tr>
<th>Static</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

// more code below
Question 2: If swimming in the 61\(\rightarrow\)6+1\(\rightarrow\)7 Cs, keep “a float” (14 pts, 36 min)

The figure below shows the layout of the different types of 32-bit float numbers on a not-to-scale real number line (with zero in the center, and NaNs considered to be further from zero than infinity).

For each of the (a-f) values below, draw marks below the number line (like we’ve done for 0) indicating where the value would fall. **Label the mark with the corresponding letter and actual value below it.**

a) The **sign magnitude** number closest to \(\infty\).

b) The result of casting the **double number** closest to \(-\infty\) (but is not \(-\infty\)) into a float.

c) \(1/(X+1)\), where \(X\) is the largest **ones complement** number.

d) The number represented by the **float bits**
0x80000001
(using normal floating point multiply \texttt{mul.s})

e) The number represented by the **float bits**
0x00000000
times the number represented by the **float bits**
0xFFFFFPPP
(flip the sign)

f) The difference between the **int value closest**
to \(-\infty\) and the **float** that can most closely
represent that int’s value.

---

The largest unsigned int is \(2^{32} - 1\), which falls between floats \(2^{32} - 2^{16}\) and \(2^{32} - 2^8\). Thus, halfway between those two floats is \(2^{32} - 2^7\), whose bits look like \([0 \mid \text{bits-encoding-exponent-32} \mid 11...11 \] 1, and rounding to ‘even’ means rounding away from odd, which is rounding up.
Question 3: Goodness, Grandma, what bignums you have! (15 pts, 36 min)

Part A: After your extensive C bignum experience, you were hired by Lawrence Berkeley Labs to make an arbitrary precision math package. The scientists use scientific notation and keep track of significant figures. The scientists have written a function called sci_bignum_cmp as shown below. Unfortunately, the implementation has at least one bug. In the boxes at the bottom, briefly explain all of them and give sample values for a and b (i.e. 1.23 x 10^4) that causes sci_bignum_cmp() to reveal the bug. You may not necessarily use all the boxes.

```c
#define POS '+'
#define NEG '-'
typedef struct sci_bignum {
    char sign; // POS or NEG
    char *significand; // null-terminated string of decimal digits ('.' implicit)
    unsigned int num_sigfigs; // equal to strlen(significand)
    int exponent;
} sci_bignum_t;

// Compare a to b; return <0 if a < b, 0 if a == b, or >0 if a > b (just like strcmp)
int sci_bignum_cmp(sci_bignum_t *a, sci_bignum_t *b) {
    if (a->exponent != b->exponent) {
        return (a->exponent < b->exponent ? -1 : 1);
    } else if (a->sign != b->sign) {
        return (a->sign < b->sign ? -1 : 1);
    } else {
        for(int i = 0; a->significand[i]; i++) {
            if(a->significand[i] != b->significand[i]){
                return (a->significand[i] < b->significand[i] ? -1 : 1);
            }
        }
    }
}
```

As an example, to store the number 1.23 x 10^4, the sign would be the char '+', the significand would be the null-terminated string "123", numSigfigs would be 3, and the exponent would be 4. There is an implicit decimal point after the first significand digit.

<table>
<thead>
<tr>
<th>Bug Description</th>
<th>Values for a and b that reveal the bug</th>
<th>What a correct sci_bignum_cmp should return</th>
<th>What this buggy sci_bignum_cmp returns / does</th>
</tr>
</thead>
</table>
| exponent compared before sign | a = -1.23 x 10^4  
b = 1.23 x 10^-4 | -1 | 1 |
| Wrong return value when exponents are equal, but a->sign != b->sign since char '+' < '-' | a = -1.23 x 10^4  
b = +1.23 x 10^4 | -1 | 1 |
| Wrong return value when two negative numbers passed in | a = -1.2 x 10^4  
b = -1.4 x 10^4 | 1 | -1 |
| No return value when a == b (should be 0) | a = 1.23 x 10^4  
b = 1.23 x 10^4 | 0 | $v0 not set so returns garbage. Might not compile |

4/7
Question 3: Goodness, Grandma, what bignums you have! (cont’d)

Part B: The scientists also use vectors quite a bit, so they wrote the following C struct:

```c
typedef struct sci_vector {
    sci_bignum_t *elts;
    unsigned int num_elts;
} sci_vector_t;
```

The scientists want the elements of a given vector to have the same number of significant digits. In other words, each element in a vector should be truncated to the smallest `num_sigfigs` in the vector. They want you to help them write a function (by filling in the blanks) that will “clean up” its argument vector by modifying all its `sci_bignum`s to have the appropriate number of significant digits. Any excess allocated space should be freed so there’s no wasted memory. Avoid memory leaks.

```c
#define MIN(a, b) ((a)<(b)?(a):(b))

void clean_vector(sci_vector_t *vec) {
    unsigned int min_sigfigs = 0xFFFFFFFF; // Initialize to biggest unsigned int

    /* get min significant digits */
    for (unsigned int i = 0; i < vec->num_elts; i++)
        min_sigfigs = MIN(min_sigfigs, vec->elts[i].num_sigfigs);

    /* truncate all elts to have min_sigfigs */
    for (unsigned int i = 0; i < vec->num_elts; i++) {
        sci_bignum_t *b = vec->elts + i; // convenient reference

        if (b->num_sigfigs > min_sigfigs) {
            char *new_significand = malloc((min_sigfigs + 1)*sizeof(char));
            b->significand[min_sigfigs] = '\0';
            strcpy(new_significand, b->significand);
            free(b->significand);
            b->significand = new_significand;
            b->num_sigfigs = min_sigfigs;
        }
    }
}
```
Question 4: fun with MIPS ... more naughty bits! (15 pts, 36 min)

What follows is an inefficient MIPS function. Read it carefully, and answer the questions below.
The definition of div can be found in your green sheet, column ↑ (div a,b ∏ lo=a/b, hi=a%b).

fun: mov $v0, $0
     li $s0, 1
loop: beq $a1, $0, end
     addiu $a1, $a1, -1
     sll $s0, $s0, 1
     div $a0, $s0
     mfhi $s1
     or $v0, $v0, $s1
     j loop
end: jr $ra

// Precondition: y < 31

unsigned int fun(unsigned int x, unsigned int y)
{
    return x & ((1 << y) - 1);
}

a) Briefly, explain what fun returns (assuming y < 31).

Don't describe the algorithm; explain how the return value relates to x and y.

b) Write optimized C code for fun in the box (make it as compact and efficient as possible).

That is, think of all the C tricks you know and try to author it in the fewest characters possible.

c) Uh oh, we've broken some calling conventions! What should we add to the
beginning (before mov $v0, $0) and end (before jr $ra) of fun to correct this? Help!

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>addiu $sp $sp -8</td>
<td>lw $s0 0($sp)</td>
</tr>
<tr>
<td>sw $s0 0($sp)</td>
<td>lw $s1 4($sp)</td>
</tr>
<tr>
<td>sw $s1 4($sp)</td>
<td>addiu $sp $sp 8</td>
</tr>
</tbody>
</table>
Question 5: He’s a unix. He’s definitely a unix. He’s dead! (15 pts, 36 min)

Assume (for simplification) that main returns a value to its caller (Unix) through standard MIPS procedure calling conventions. We wish to see how long our command-line inputs arguments are:

```
unix% count_argument_characters
0

unix% count_argument_characters I love cs61c!
11
```

Implement count_argument_characters in MAL MIPS. Follow the hints given by the comments; you may not need to use all the lines.

```
li $v0, 0
main: ______________________   # ans=0
        $a0, $a0
word:  addiu __________, -1    # Decrement
        $a0, $0
        beq ____________, done    # We’re done!
        sll        $a0   $2
        ______ $t0, ____, ____  # change $t0
        addiu $t1, $a1, $t0
        __________
        lw $t1, 0($t0)          #
        __________
        lb $t2, 0($t1)          #
letr: ______________________   #
        beq $t2, _______, word   # end of word
        addiu $v0, $v0, 1        #
        __________
        addiu $t1, $t1, 1       # increment $t1
        __________
        letr
        j ____________          # keep processing

done: jr $ra
```