a) Instructions (Read Me!)

- Don’t Panic!
- This booklet contains 6 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.
Question 1: Potpourri: hard to spell, nice to smell… (14 pts, 36 min)

Questions (a) and (b) refer to the C code to the right; pretend you don’t know about MIPS yet.

a) In which memory sections (code, static, heap, stack) do the following reside?

<table>
<thead>
<tr>
<th>arg</th>
<th>stack</th>
<th>arr</th>
<th>static</th>
</tr>
</thead>
<tbody>
<tr>
<td>*str</td>
<td>heap</td>
<td>val</td>
<td>code</td>
</tr>
</tbody>
</table>

b) Name a C operation that would treat arr and ptr differently: ____________________________

You peek into the text part of an a.out file and see that the left six bits of an instruction are 0x02. As a result of executing this instruction...

c) What’s the most that your PC could change? Be exact.________________________________________

d) What is the least? ____________________________________

e) Write a getPC function, which returns the address of the jal instruction calling it.

(two instructions should be sufficient)

getPC: ______________________________________________________________

f) Which of the best-, first-, next-fit schemes would succeed for all 5 of the following sequence of malloc and free requests on a malloc-able region of memory only 8 bytes long? Circle those that would and show the resulting contents of memory for each one. E.g., After the "a=malloc(4)" call, all schemes should have the leftmost 4 boxes labeled "a". A pencil is useful (or draw "a" lightly).

a = malloc(4); b = malloc(1); free(a); c = malloc(3); d = malloc(4);

```
 d d d d b c c c  
```

(best-fit)

d d d d b c c c  

first-fit

d d d d b c c c  

(next-fit)

g) In one sentence, why can’t we use automatic memory management in C?

C is weakly typed; any variable could be a pointer.

h) To reduce complexity for your software company, you delete the Compiler, Assembler and Linker and replace them with a single program, CAL, that takes all the source code in a project and does the job of all three for all the files given to it. Overall, is this a good idea or bad idea? Why or why not?

BAD idea! A change to only one file requires recompiling/reassembling all!
Question 2: Player’s got a brand new bag… (15 pts, 36 min)

We want to add an inventory system to the adventure game so that the player can collect items. First, we’ll implement a bag data structure that holds items in a linked list. Each item has an associated weight, and each bag has a max_weight that determines its holding capacity (see the definitions below). In the left text area for item_node_t, define the necessary data type to serve as the nodes in a linked list of items, and in the right text area, add any necessary fields to the bag definition.

typedef struct item {
    int weight;
    // other fields not shown
} item_t;

typedef struct item_node {
    // (a) FILL IN HERE
    item_t *item;
    struct item_node *next;
} item_node_t;

typedef struct bag {
    int max_weight;
    int current_weight;
    // add other fields necessary
    // (b) FILL IN HERE
    item_node_t *contents;
} bag_t;

c) Complete the add_item() function, which should add item into bag only if adding the item would not cause the weight of the bag contents to exceed the bag’s max_weight. The function should return 0 if the item could not be added, or 1 if it succeeded. Be sure to update the bag’s current_weight. You do not need to check if malloc() returns NULL. Insert the new item into the list wherever you wish.

int add_item(item_t *item, bag_t *bag) {
    if (item->weight + bag->current_weight > bag->max_weight) {
        return 0;
    }
    item_node_t *new_node = ________________;
    // Add more code below...
    new_node->item = item;
    new_node->next = bag->contents;
    bag->contents = new_node;
    bag->current_weight += item->weight;
    return 1;
}

(d) Finally, we want an empty_bag() function that frees the bag’s linked list but NOT the memory of the items themselves and NOT the bag itself. The bag should then be “reset”, ready for add_item. Assume that the operating system immediately fills any freed memory with garbage. Fill in the functions below.

void empty_bag(bag_t *bag) {
    free_contents( ______________________ );
    // FILL IN HERE
    bag->current_weight = 0;
    bag->contents = NULL;
}

void free_contents( _________________ ) {
    // FILL IN HERE
    if (c == NULL) return;
    free_contents(c->next);
    free(c);
}
Question 3: You won’t mind this question one bit! (15 pts, 36 min)

We wish to implement a bit array, where we can read and write a particular bit. Normally for read/write array access, we would just use bracket notation (e.g., x=A[5]; A[5]=y), but since a bit is smaller than the smallest datatype in C, we have to design our own GetBit() and SetBit() functions. We'll use the following typedefs to make our job easier:

typedef uint8_t bit_t; // If it's a single bit, value is in least significant bit.
typedef uint32_t index_t; // The index into a bit_t array to select which bit is used

E.g., imagine a 16-bit bit array: bit_t A[2]; A[1]=0x82; A[0]=0x1F; Internally, A would look like this:

<table>
<thead>
<tr>
<th>Array A</th>
<th>8</th>
<th>2</th>
<th>1</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0</td>
<td>0 0 1 0</td>
<td>0 0 0 1</td>
<td>1 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

GetBit(A,0) would return 1, as would GetBit(A,1), GetBit(A,2), GetBit(A,3), and GetBit(A,4). GetBit(A,5) would return 0, as would GetBit(A,6), GetBit(A,7), and GetBit(A,8). Etc.

a) How much space would the largest usable bit array take up? 512 MiB

Express your answer in IEC format. E.g., 128 KiB, 32 TiB, etc.

b) Write SetBit in C. You may not need to use all the lines.

```c
void SetBit(bit_t A[], index_t n, bit_t b) {
    A[n/8] = (A[n/8] & ~(1 << (n%8))) | (b << (n%8)); // The one liner. Or …
    uint32_t byte_index = n/8; // (1) Find out which byte we want
    uint8_t bit_index = n%8; // (2) Find where within that byte is the bit
    A[byte_index] &= ~(1 << bit_index); // (3) Reset that bit in the byte
    A[byte_index] |= b << bit_index; // (4) Assign that bit in the byte
}
```

c) Write GetBit(bit_t A[], index_t n) in MAL; $v0 should be 1 if the bit is on, and 0 if it’s off.

Hint: it might help if you start from the srlv and work backwards.

```c
GetBit:
    srlv $v0,$t2,$t3   # $v0 = byte >> bit_index (slide bit to lsb slot)
    srlv $v0,$t2,$t3   # “srlv rd,rt,rs” means (in C): rd = rt >> rs
    andi $v0,$v0,1     # $v0 &= 1 (mask out the lsb bit)
    j $ra
```

# $t0 = byte_index = n/8
# $t1 = A + byte_index
# $t2 = byte = *(A+byte_index) = A[byte_index]
# $t3 = bit_index = n%8
# $v0 better be either a 0 or 1
Name: ___________________________ Login: cs61c-____

**Question 4: Did somebody say “Free Lunch”?! (15 pts, 36 min)**
Consider two competing 5-bit floating point formats. Each contains the same fields (sign, exponent, significand) and follows the same general rules as the 32-bit IEEE standard (denorms, biased exponent, non-numeric values, etc.), but allocates its bits differently.

Implementation “LEFT”:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>S E E F</td>
</tr>
</tbody>
</table>

scratch space (show all work here)

Exponent Bias: $2^{2^1-1} = 2^{2^{-1}} = 1$

Denorm implicit exponent: 

Number of NANS:

<table>
<thead>
<tr>
<th>What</th>
<th>Number</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest non-zero pos denom</td>
<td>$2^0 \times 0.01 = \frac{1}{8}$</td>
<td>0b00001=0x1</td>
</tr>
<tr>
<td>Largest non-infinite pos value</td>
<td>$2^{1} \times 1.11 = 3.5$</td>
<td>0b01011=0xB</td>
</tr>
<tr>
<td>Negative Infinity</td>
<td>$-\infty$</td>
<td>0b11100=0x1C</td>
</tr>
</tbody>
</table>

Mark every representable number in the range [+0,1] as a vertical line on the number line below. We’ve already done it for +0.

<table>
<thead>
<tr>
<th>0</th>
<th>1/8</th>
<th>1/4</th>
<th>1/2</th>
<th>3/4</th>
<th>1</th>
</tr>
</thead>
</table>

Which implementation is able to represent more integers, LEFT or (RIGHT) ? (circle one)

Implementation “RIGHT”:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>S E E F</td>
</tr>
</tbody>
</table>

scratch space (show all work here)

Exponent Bias: $2^{3^1-1} = 2^{3^{-1}} = 3$

Denorm implicit exponent:

Number of NANS:

<table>
<thead>
<tr>
<th>What</th>
<th>Number</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest non-zero pos denom</td>
<td>$2^{-2} \times 0.1 = 1/8$</td>
<td>0b00001=0x1</td>
</tr>
<tr>
<td>Largest non-infinite pos value</td>
<td>$2^{2} \times 1.1 = 12$</td>
<td>0b01101=0xD</td>
</tr>
<tr>
<td>Negative Infinity</td>
<td>$-\infty$</td>
<td>0b11110=0x1E</td>
</tr>
</tbody>
</table>

Mark every representable number in the range [+0,1] as a vertical line on the number line below. We’ve already done it for +0.

<table>
<thead>
<tr>
<th>0</th>
<th>1/8</th>
<th>1/4</th>
<th>1/2</th>
<th>3/4</th>
<th>1</th>
</tr>
</thead>
</table>
Question 5: Three’s a Crowd… (15 pts, 36 min)

Breaking news! We have just developed hardware that has 3-states: {false=0, true=1, and maybe=2}! Now we can store all our numbers in base 3. The race is on to develop a good encoding scheme for integer values.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Ternary</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12&lt;sub&gt;three&lt;/sub&gt;</td>
</tr>
<tr>
<td>26</td>
<td>222&lt;sub&gt;three&lt;/sub&gt;</td>
</tr>
<tr>
<td>27</td>
<td>1000&lt;sub&gt;three&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

a) To warm up, first do some simple conversions between decimal and unsigned ternary. We’ve done one for you.

b) Suppose we have N ternary digits (tets, for short). What is the largest unsigned integer that can be stored? ____________ 

3<sup>N</sup> – 1

Ok, now that we’ve got unsigned numbers nailed down, let’s tackle the negatives. We’ll look to binary representations for inspiration.

c) Name two disadvantages of a sign and magnitude approach in ternary. Suppose a leading 0 means positive, and a leading 1 means negative, similar to what we did in the binary days.

There are only two signs, but three digits (we waste 1/3 of our numbers)

Wierd gap behavior -> now we jump to 2’s mystery zone from 11…11 + 1.

d) Maybe three’s complement will be more promising. To make sure we understand what that means, let’s begin with a very small example – say a 2-tet number. Fill in the following number ring of tet-patterns with the values we’d like them to represent (just as in two’s complement, we want all zeros to be zero, and want a balanced number of positive and negative values).

Going clockwise from 00<sub>three</sub> : 0, 1, 2, 3, 4, -4, -3, -2, -1

![Diagram of ternary complement]

- 00<sub>three</sub> - 11<sub>three</sub>
- 21<sub>three</sub> - 02<sub>three</sub>
- 12<sub>three</sub> - 10<sub>three</sub>
- 11<sub>three</sub> - 11<sub>three</sub>

e) Recall that for an N-bit two’s complement number, the bit-pattern of the largest positive number looks like 011…11. For an N-tet three’s complement number, what does the tet-pattern of the largest positive number look like?

11…11

f) Provide (in pseudocode) an algorithm for negating an N-tet three’s complement number.

Flip all tets across 1. (i.e., 0 -> 2, 1-> 1, 2 -> 0)

Add 1. (same as 2’s complement!)