Question 1: For those about to test, we salute you…(11 pts, 20 min.)

You’re coming straight from a wild pre-midterm toga party and are not quite focused on the exam. Fear not, this question will get you warmed up and ready to rock, 61C style.

a) J-Lo whispers her age to you: “thirty six”. Was that big-endian or little-endian language?

b) You “peek” into memory and see the instruction shown on the right. What would this be if we interpreted this instead as:

…a single hexadecimal number?

…a sequence of four characters?

What’s the #1 reason why copying is better than mark and sweep?

What’s the #1 reason why reference counting is better than copying?

When would the buddy system be better than the slab allocator?

When would the slab allocator be better than the K&R freelist alone?

d) Suppose we want to add a new Pseudo-Instruction “branch if float is Nan or Infinity”: bfni. This instruction will take two arguments, a float (stored in a register) and a label, e.g.,

bfni $a0 done. It takes the branch if the floating point number in the register is NaN or $\pm \infty$. To what TAL instructions (from the “core instruction set”) could this be expanded? It can be done in four (or fewer!) instructions; you may not use any more.

e) Whose job is it to do that expansion? Circle one.
**Question 2: Bit fields and bit fields of wheat…(7 pts, 15 min.)**

You believe the encoding of MIPS opcodes, functions and bit field widths should be modified. What we currently use is on the left and what you propose is on the right – note you have not changed the code for the R and I cases, just reordered them. We’ve left out the code (replacing it with “??”) for determining which R and which I instruction it is, assume that can be worked out later. How many R-type, I-type and J-type instructions did we have and will we have and what is one big pro and two big cons of this proposal? Consider how this subtle change could change the bit fields for the instructions for better or worse. Yes, we already know we’d probably have to reprint things like the green sheet. When counting instructions, only count the number of different operators. E.g., you should count `jr $v0` and `jr $a1` as the same instruction (same `jr` operator). We’ve filled one in for you already.

<table>
<thead>
<tr>
<th>Current</th>
<th>Proposed (changes in <strong>bold</strong>)</th>
</tr>
</thead>
</table>
| if ((bit[31]...bit[26] == 0) {
  // Look elsewhere for which R it is...
  inst_type = Rl inst = ??;
} else if ((bit[31]...bit[26] == 2) {
  inst_type = J; inst = jump;
) else if ((bit[31]...bit[26] == 3) {
  inst_type = Jl inst = jal;
) else {
  // Remaining ops are I
  // Ignore Floating Pt formats FR,FI
  inst_type = I; inst = ??;
} | if (bit[31] == 1) { // Most-signif. Bit
  inst_type = J
  if (bit[30] == 1)
    inst = jump;
  else
    inst = jal;
} else if ((bit[31]...bit[26] == 0) {
  // Look elsewhere for which R it is...
  inst_type = Rl inst = ??;
} else {
  // Remaining ops are I
  // Ignore Floating Pt formats FR,FI
  inst_type = I; inst = ??;
} |

<table>
<thead>
<tr>
<th># of R-type</th>
<th># of I-type</th>
<th># of J-type</th>
<th># of R-type</th>
<th># of I-type</th>
<th># of J-type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The pro is…

1. 

The cons are…

1. 

2.
**Question 3: This Q brought to you by the # 0x80000008...(16 pts, 35 min.)**

a) Your favorite 32-bit hex quantity is 0x80000008. For each of the encodings, tell us what the decimal value of this number is, and where on a linear, finite number line (LFNL) that number would lie given all the possible finite numbers that can be encoded. There are five dashed marks on our LFNL: the leftmost dash marks “min”, the smallest encodable finite value (closest to -∞) and the rightmost dash marks “max”, the largest. The middle dash marks the halfway point between min and max (not necessarily zero!), and the other two dashes mark ¼ and ¾ of the distance from min to max. If the mark you make falls pretty much on one of the existing dashed marks, in the space below the LFNL circle whether (if we were to zoom in) your mark would be to the “LEFT”, “ON”, or to the “RIGHT” of our dashed mark. E.g., if the 32-bit hex quantity were 0x0, the Sign-magnitude we’d put our mark directly on the middle dashed mark and circle “ON”. If it were 0x1, it’s be the same mark but we’d circle “RIGHT”. Remember, this number line is linear & finite! Show your work.

![LFNL Diagram]

b) Let’s now turn the problem on its head. We’re giving you the LFNL (as explained in the last problem) with a mark ¼ of the way from min to max as shown here. What is the corresponding number and its hex value for each encoding? If you have a choice between two numbers (i.e., the ¼ mark doesn’t exactly lie on a number in that encoding), always choose the neighboring simpler number (the one with the fewest 1s in its binary representation). Again, show your work.

![LFNL Diagram]
a) We want to be able to insert a node. Finish the Insert function which will insert a (key, value) pair in the right place. It is Insert’s job to copy these string arguments into its internal structure. You may not write any additional functions. We want the tightest, cleanest code possible (measured by the number of statements which terminate in semicolons). You can assume all keys will be fewer than 80 characters, and that if given a key already in the table, Insert should not make a new node, but instead clobber the old value. Here’s how we could call Insert; we’ll ask you to write Delete on the next page. We’ve included two potentially useful String functions below.

```c
int main() {
    bst_t *heat = NULL;
    head = Fill(); /* Somehow fill the table via user input */
    head = Insert(head, "Tiger", "Woods"); /* Insert another entry into table */
    Print(head); /* Somehow print the table */
    Delete(head); /* Delete the table (code on next page) */
    DoSomething(); /* Do something, but we need the space back! */
}

bst_t *Insert(bst_t *table, const char *key, const char *value) {
    if () { /* New */
    } else if () { /* Clobber! */
    } else if () { /* Left */
    } else { /* Right */
        return table;
    }
}

char *strcpy(char *s, const char *ct) : Copy string ct to string s, including '\0'; return s
int strcmp(const char *cs, const char *ct) : Compare string cs to string ct;
    return < 0 if cs<ct, 0 if cs=ct, or >0 if cs>ct
size_t strlen(const char *cs) : Return length of cs
```
Question 4 (cont’d): Are you the keymaster? bst of burden?

(The typedef for the struct declaration is reprinted on the right.) Now that you’ve written the Insert function, we’ll call it form main() as shown below:

```
1 bst_t *head = NULL;
2 int main(int argc, char *argv[]) {
3    bst_t mybst;
4    head = Insert(head, “61C”, “Awesome”);
5    printf(“Key: %s, Value: %s\n”, head->key, head->value);
6 }
```

b) Assume you’ve written Insert as efficiently as possible. At the time of the call to printf, how many bytes would be used in the static, stack and heap areas as the result of…

<table>
<thead>
<tr>
<th>static</th>
<th>stack</th>
<th>heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

c) Finally, we would like to delete the full structure. When deleting, assume that the OS immediately fills any freed space with garbage, so you cannot access freed heap contents. Finish the Delete function to completely delete the table. You may find the typedef above handy.

```
void Delete (bst_t *table) {

    return;
}
```
Implement the `IsLive` function in MIPS. Use as few lines as possible (you may not need every blank). Assume the pseudoinstructions `sllv, srlv` (shift left & right logical variable) exit. Wherever there is a comment below, you must write a MIPS instruction that calculates exactly that value. Your code below doesn’t have to be a literal translation of the C return statement, just compute the same value.

```mips
IsLive:    ____ $a1, ____, ____ #
           ____ $t0, ____, ____ #
           ____ $a2, ____, ____ #
           ____ ____, ____, ____ # t0 = (nw << 2) | (n << 1) | ne
           ____ ____, ____ #
           ____ ____, ____ # t1 = 1 << ((nw << 2) | (n << 1) | ne)
           ____ ____, ____ #
           ____ ____, ____ #
           ____ ____, ____, ____ # set v0 to be 0=dead or 1=live
           jr   $ra
```
Question 6: Magical mystery MIPS, step right this way… (13 pts, 35 min)

Main: ......
   # Set up $a0
   jal mystery
 ......

mystery:
   addi $sp, $sp, -8
   sw $ra, 0($sp)
   sw $s0, 4($sp)
   lw $s0, 0($a0)
   beq $s0, $0, mystery_label1
   addi $a0, $a0, 4
   jal mystery
   xor $v0, $v0, $s0
   andi $v0, $v0, 1
   j mystery_label2

mystery_label1:
   li $v0, 0

mystery_label2:
   lw $ra, 0($sp)
   lw $s0, 4($sp)
   addi $sp, $sp, 8
   jr $ra

---

a) In the box above, fill in the C code for the function mystery. Do not use any loop constructs like while or for. Be sure to include arguments and return values, along with their types.

b) What does the function mystery return?

c) What would mystery return if we changed the “li $v0, 0” to read “li $v0, 1”?

d) What would mystery return if we changed the “li $v0, 0” to the TAL instruction “0x00000000”? Be very specific.

---

XOR Truth Table

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a XOR b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

---
