CS 61C, Fall, 2005, Midterm 1, Garcia

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?

Big	little

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?

```
addi $t7, $k0, 0x6821
```

c) Answer the following questions in one sentence (or even a few short words).

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 ± ∞. 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.

```
bfni $a0 done
```

e) Whose job is it to do that expansion? Circle one.

Interpreter **Compiler** **Assembler** **Linker** **Loader**

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.

Current	Proposed (changes in **bold**)

`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 = ??;`
	`}`

# of R-type	# of I-type	# of J-type	# of R-type	# of I-type	# of J-type
		2			

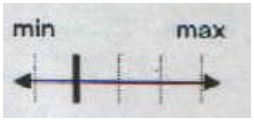
The pro is…	1.
The cons are…	1. 2.

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.

	Unsigned Int	Sign-magnitude	2s Complement	`float`
Show your work in these boxes →				
Decimal value (in *simplest* notation... you can use 2^)				
Where does it lie on the LFNL?	min max	min max	min max	min max
If *on* a dashed mark, circle LEFT, ON, or RIGHT.	LEFT ON RIGHT	LEFT ON RIGHT	LEFT ON RIGHT	LEFT ON RIGHT

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.

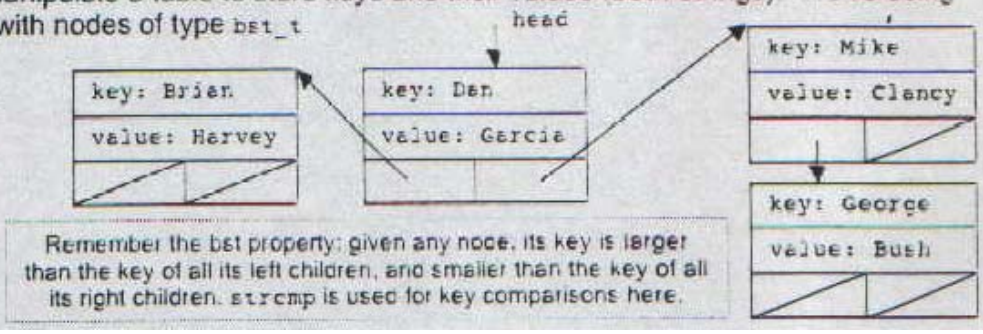


	Unsigned *char*	Sign-magnitude *byte*	2s complement signed *char*	`float`
Show your work in these boxes →				
Decimal value (in *simplest* notation... you can use 2^)				
Hex value	0x	0x	0x	0x

Question 4: Are you the keymaster? bst of burden? (17 pts, 40 min)

We're authoring code to manipulate a table to store *keys* and their *values* (both strings). We're using a *binary search tree* (bst) with nodes of type `bst_t` as defined below; a sample bst is on the right.

```
typedef struct node {
    char key[80];
    char *value;
    struct node *left;
    struct node *right;
} bst_t;
```



Remember the bst property: given any node, its key is larger than the key of all its left children, and smaller than the key of all its right children. `strcmp` is used for key comparisons here.

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 `key`s 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.

```
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 }
```

```
Typedef struct node {
   char key[80];
   char *value;
   struct node *left;
   struct node *right;
} bst_t;
```

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…

	static	stack	heap
`Line 1`			
`Line 3`			
`Line 4`			

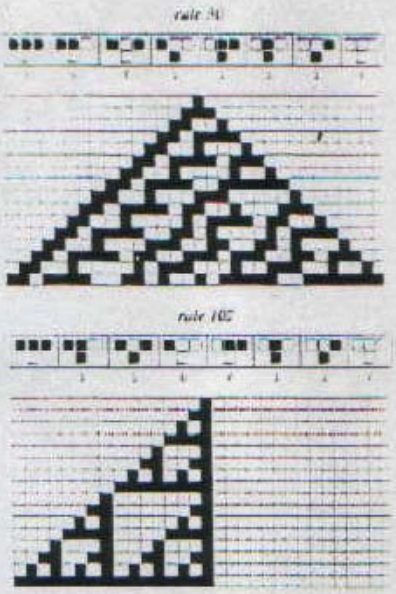
c) Finally, we would like to delete the **full structure**. When deleting, assume that the OS immediately fills any freed space with garbage, so you cannot access freed heap contents. Finiah the `Delete` function to completely delete the table. You may find the `typedef` above handy.

```
void Delete (bst_t *table) {









   return;
}
```

Name: _____ Login: cs61c-___

Question 5: Life1D, revisited... (10 pts, 35 min)

Hopefully you remember Homework 2 with fondness. Now that you've learned about MIPS and bit manipulation, you want to code the central `IsLive` function from that assignment directly into MIPS. All your code essentially boils down to two functions: `main` and `IsLive`. (The variables `nw`, `n`, and `ne` encode the Live/Dead status of the pixels directly above you to your northwest, north and northeast respectively). The graphics to the right show the examples for rules 30 and 102 and should remind you of everything you need to do.



```
typedef unsigned char bit_t;
typedef unsigned char rule_t;
int main(int argc, char *argv[]) {
  rule_t rule;  /* This encodes our rule [0,255] */
  bit_t  ne, n, nw, pixel, IsLive(); /* 0=dead or 1=live */
  /* Do initial setup, read args, set rule, etc */
  /* for each pixel in the row (nw, n, ne, rule given) */
     pixel = IsLive(ne, n, nw, rule); /* 0=dead or 1=live */
  /* Print out the row */
}

bit_t IsLive(bit_t ne, bit_t n, bit_t nw, rule_t rule) {
   return(rule & (1 << ((nw << 2) | (n << 1) | ne)) ? 1 : 0); /*return 0=dead or 1=live*/
}
```

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.

```
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
```

```
________  ___________ (                    )
{
    if ( ___________________________________ ) {



    } else {



    }
}
```

XOR Truth Table

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

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.