;;;;;;;;;;;;;;;;;;;;;;;;;
;; 2005Sp CS61C Midterm ;;
;;;;;;;;;;;;;;;;;;;;;;;;;

GS = "Grading Standard"

;;;;;;;;;;;;;
;; QUESTION 1
;;;;;;;;;;;;;

a) "#One" is in a register. Mapping the characters, one-by-one, into
hex values, we consult the table to find '#' => 0x23, 'O' => 0x4f,
'n' => 0x6e, and 'e' => 0x65. Concatenating them together yields
0x234f6e4f.

GS: 1 point. Half credit if you converted from character 'o'
rather than 'O'.

Converting them to an instruction involves mapping the word to a
binary representation, breaking it up into fields based on the leftmost
6-digit opcode, and translating each field into a value.

0b 0010 0011 0100 1111 0110 1110 0100 1111
The leftmost 6 opcode bits are "00 1000", so it's an addi I-type inst.
This has 4 fields of length [6 5 5 16], so breaking up the bits we get
= 00 1000 | 1 1010 | 0 1111 | 0110 1110 0100 1111
= addi | rs=$26 | rt=$15 | 0x6e4f
= addi | rs=$k0 | rt=$t7 | 0x6e4f
= addi $t7,$k0,0x6e4f

GS: 1 point. No deduction if you lost the half point in the first part
of the question and correctly converted that answer into an instruction.
Half credit if you flipped $t7 and $k0.

b) The number of exponent bits controls the range of the IEEE float and
the number of significant bits controls the precision. Thus, if we hand
some exponent bits to the significand, the pro will be "reduced range",
and the con will be "increased precision".

GS: 2 points. 1 point for each box. All or nothing.

Common Confusion: Reversing the answers.

c) To convert the pseudoinstruction "bfz $a0 done" to TAL, we have to
detect either 0, represented as 0x80000000 (-0) or 0x00000000 (0).
One clever way to do this is to shift the word left by one place
which would push that topmost bit (1 or 0) into the bitbucket and
insert a 0 on the right. Then we simply need to test whether that
resulting value is all zeros. We use our temporary register $at:

```
sll $at,$a0,1
beq $at,$0,done
```

GS: 2 points. Lots of interesting answers here... Anything that worked
as was straight forward enough for me to figure out got full credit. Basically,
1 point for acknowleding that there are two zeros and 1 point for correctly branching
on zero.

Common Confusion:
Forgetting that there are two zeros and just using a BEQ $a0, $0, done.
Using the ADDI instruction in ways that didn't make sense.
Trying to ANDI $at, $a0, 0x7FFFFFF (doesn't work because immediates can't be that big)
Shifting way, way too much (I saw some solutions with as many as 12 seperate shift instructions.)

d) The Buddy System and Slab Allocator are similar in that they use
their particular technique to handle small requests but the K&R
for large memory requests. Each introduces internal fragmentation,
and K&R introduces external fragmentation, so the answers are:

Buddy System : causes both types
Slab Allocator : causes both types
K&R (Free List Only) : causes external only

GS: 2 points. 1 point for one incorrect answer, 0 points for 2 or 3 incorrect answers

e) Since Copying only uses half of the memory at any one time, "full"
means M/2. The others use all the memory, so "full" means "M".
After GCing, in the best case, all systems are able to clear out all
unused data, so the least space the data would take up is 0 for all.
The most space their data could take up after GCing is the same as
the first column, if the GCer didn't recover any unused space. Wasted
space? Reference counting cannot handle circular references, so at worst
the entire memory could be filled (M) with such a structure. The table

looks like:

Reference Counting M | 0 | M | M
Mark and Sweep M | 0 | M | 0
Copying M/2 | 0 | M/2 | 0 or M/2

GS: 2 points. -1/2 for each incorrect box.

;;;;;;;;;;;;;
;; QUESTION 2
;;;;;;;;;;;;;

a) 3^n

GS: All or nothing, 1 point

b) (3^n) - 1

GS: All or nothing, 1 point

c) Ternary rep of -3: 22...220
# of positive ints: ((3^n) - 1) / 2
# of positive ints: ((3^n) - 1) / 2
# of zeros: 1

GS: All or nothing, 1/2 point each. Common misconceptions include forgetting
that we needed the representation for n bits (and instead just giving the answer for 3 digit numbers.

d) TernaryNegate(char *in, char *out) {
  char *tmp = out;
  while (*in)
    *tmp++ = InvertTernaryDigit(*in++);
  AddTernaryConstant(out,1);
}

InvertTernaryDigit(char c) {
  return('0' + '2' - c);
}

GS: 4 points: 2 points for TernaryNegate, 2 points for InvertTernaryDigit.
-1 for forgetting to add one (using AddTernaryConstant). Common misconceptions include doing string operations

(calling strlen, etc...) and getting lost that way. Some people didn't figure out what it means to invert a ternary digit
(tried rotating them, such as: 0->1, 1->2, 2->0).

e) int isNegative(char *in) {
```
  if (!*in || *in == '0')  /* If it's all 1s or starts with a 0, not neg */
    return(0);
  else if (*in == '2')     /* If it starts with a 2, negative */
    return(1);
  else                     /* Else, it starts with a 1, recurse */
    return(isNegative(in+1));
}
```

GS: 6 points: 2 points for getting the trivial cases (a '0' or a '2' at the beginning of the number) and 4 points for handling the recursion correctly--all of it, including base cases. Many misconceptions included only checking the first digit or the last digit. Handling the base case of the recursion incorrectly (not checking for a null character. Basically it was one point for each of the following: recursing to the next character (handling the '1' case, checking the null case, checking the '0' case, and checking the '2' case). Another mistake some people made was to think they could just find a 2 anywhere, then it'd be negative. It's only the case if the first non-1 digit is a 2 that it's negative.

;;;;;;;;;;;;;
;; QUESTION 3
;;;;;;;;;;;;;

a) 01: 0x2222FFFC
02: 0x1040000A
03: 0xAC830004

GS: 2 points each, 6 points total.
Each of these three instructions is and I-format and 1 point was given for the immediate (last 16 bits) and 1 point for the opcode and registers (first 16 bits)

For the branch I only accepted 0x1040 0000a as an answer but in retrospect I should also accept 0x1002 000a which corresponds to having the argument registers flipped. If you did this and lost points ask for a regrade.

b) 16 Exbirules

GS: 2 points, all or nothing

c) 2 Tebi

GS: 2 points, all or nothing. 0 points were given for saying 41 bits (or something to this extent).

;;;;;;;;;;;;;
;; QUESTION 4
;;;;;;;;;;;;;

a) Since the largest exponent is reserved for NaN/inf, the fields are:
0 110 111111111
= + 1.111111111 x 2^3
= + 1111.111111
= 15 63/64

GS: 3 points, all or nothing.

b) Denorm exponent is -2, so
0 000 000000001
= + 0.000000001 x 2^(-2)
= + 2^-9 x 2^-2
= 2^-11
= 1/2048

GS: 3 points, all or nothing.

c) 0x1000 = 1 000 00000000 = -0 (we also accepted 0)

GS: 1 point, all or nothing.

d) 0x1F00 = 1 111 10000000 = NaN (we also accepted -NaN)

GS: 1 point, all or nothing.

e) inf = 1 111 00000000 = 0x1E00

GS: 1 point, all or nothing.

f) -4.5 = -100.1 = -1.001 x 2^2 = 1 101 001000000 = 0x1A40

GS: 1 point, all or nothing.

;;;;;;;;;;;;;
;; QUESTION 5
;;;;;;;;;;;;;

```
void DeleteTable(keynode_t *table) {
  valuenode_t *head, *tmp;
  if (table != NULL) {
    DeleteTable(table->next);    /* Free the right part of the diagram */
    head = tmp = table->values;
    while (head != NULL) {       /* Free the bottom part of the diagram */
      tmp = head->next;
      free(head);
      head = tmp;
    }
    free(table);                 /* Free me (the upper-right node) */
  }
}
```

GS:
+2 points for the recursive call in the correct place (before free(table);)
+2 points for free(table); in the correct place
+2 points for just trying to free something on the valuenode_t lists
+2 points for correctly iterating through the valuenode list
+2 points for doing it in small number of statements (somewhere between 6 and 10 made sense)

This standard only seemed to work well for the students who sorta knew what they were doing. Students generally didn't get the last +2 unless they also generally demonstrated they knew what they were doing and had the correct basic structure.

Also, I gave out negative points in some cases that demonstrated misunderstanding.

-2 total if they tried derferencing pointers everywhere (but they got the above points for the frees)
-1 if they forgot to check table != NULL at the beginning
-2 total if they neglected to check for more null pointers

;;;;;;;;;;;;;
;; QUESTION 6
;;;;;;;;;;;;;

encryptThis:

```
 lbu   $t0, 0($a0)        #
 beq   $t0, $0,    done   #
 lw    $t1, 0($a1)        #
 addu  $t2, $t0,   $t1    #
 sw    $t2, 0($a2)        #
 addiu $a0, $a0,   1      #
 addiu $a1, $a1,   4      #
 addiu $a2, $a2,   4      #
 j encryptThis            #
done:
 sw    $0, 0($a2)         #
 jr    $ra                #
```

GS:

control [5]: included the lbu, beq, j instructions.

[2]: lbu. both lbu and lb were accepted; the fact that the ascii values were in the range 0-127 indicated that it did not matter. 1 point was deducted for using lw; 2 points were deducted for neglecting to dereference cleartext, and thus testing whether the cleartext pointer was null.

[3]: beq, j. a central part of this problem was setting the loop control to traverse to allocated arrays. because the problem explicitly stated to avoid recursion, a recursive call [jal] was not given any points.

encoding function [2]: included the lw, addu, sw instructions.

[2]: the main body of the code performed dereferences of the arrays, sums, and then a store back into an array. the lw, addu, sw are a fairly direct translation of that body.

increment [2]: included the addiu instructions.

[2]: the c call of encrypt this relied on the incrementation of the arguments. again, the addiu are a fairly direction translation of that process. 1 point was deducted for using an incorrect stride for any of the increments. note the argument types listed in the comments above provide the array stride lengths. 1 point was also deducted for having incorrect arguments- for instance, incrementing some register but writing into a different register.

epilogue [1]: included the sw, jr instructions.

[1]: the epilogue consisted of the base case of zero terminating cyphertext_buffer and the return to caller. any omission or error was a 1 point deduction.

as a note of subtlety, the correct operators for all of the additions were the 'unsigned' variants [more accurately, the 'do not signal overflow' variants]. c does not signal overflow exceptions and will always compile its additions to these 'unsigned' variants. this error was marked, but no points were deducted, as this was a common error, and had not been an emphasis of the class. any of the additions in this problem could have 'overflowed', that is, could have crossed the boundary from 0x7FFFFFFF to 0x80000000, for which the 'signed' variants would cause an exception.

all too common silliness:
  some silly things that many students did included:
   storing $a0, $a1, $a2, andor $ra. procedures are allowed to modify the contents of the $a_ registers, so storing them was unnecessary. since the problem explicitly forbade recursion, we know that we will not make a procedure call. thus, saving $ra was also unnecessary.
   doing a lw off of $a2, cyphertext_buffer, our destination buffer. reading from the destination buffer is an unnecessary operation. in this problem it was exclusively a location to be written to.
   having 'addu $t9 $0 $0' (or something similar) followed by some use of $t9 (rather than $0) to represent 0, such as in 'beq $t0 $t9 done'. this is not only unnecessary; it makes the code less clear, since it is not as obvious that $t9 represents 0.

;;;;;;;;;;;;;
;; QUESTION 7
;;;;;;;;;;;;;

```
a) char foo(char *s) {
    if (*s == '\0')
      return -1;
    else
      return (*s & ander(s+1));
  }

  /* which could be tighted up to be... */

  char ander(char *s) {
    return (*s == '\0') ? -1 : (*s & ander(s+1));
  }
```

b) The bitwise AND of all characters in a string.

c) @

d) Always return 0

GS:

6 points for C code
1 point for explaination of function
2 points for output of function
1 point for description of modified function.

The last three parts are scored all or nothing. Most students either got it or did not.

The code was (roughly) divided into three portions: type declarations (1 pt), base case (2 pts), and recursive calls/loops (4 pts). Take note of the fact that this adds up to 7 points when the code is worth 6 points. Points were deducted for mistakes for a maximum of 6 points, in the amounts specified for each part of the code.

Base case

---

Two points were assigned here for observing that the program branches when it detects '\0' in the character array. Basic C mistakes (such as using string* as opposed to *string) lead to a deduction of one point. Major C mistakes (such as checking "string == NULL" as opposed to "*string == '\0') lead to a deduction of two points. Returning a completely bizarre or strange value also was counted as -2.

Recursive calls/loops

---

Careful observation of the function will demonstrate that foo calls itself (the jal instruction) recursively. 4 points were allotted to see that this call was made, and that the return value was handled properly. Solutions that involved loops instead of recursive calls were not penalized if they produced the same result, albeit in a manner different from the MIPS code.

Major mistakes here involved not realizing that the current character was and'ed with the rest of the characters in the string (-4), calling foo but not using the return result (-4), and failing to accumulate properly, i.e. performing the operation on only the first or last character in the string (-4).

Minor mistakes only received a deuduction of one point. For example, some students did not know the syntax of the "&" operator and wrote "and" instead. More seriously, some students called foo(string++), where the correct call is foo(++string). foo(string++) will call foo with the SAME string that it as passed, incrementing it after passing the original value. This is an infinite loop! The string needs to be incremented before its value is passed, i.e. ++string.

Type declarations
---
The correct prototype for foo is
```
char foo (char *);
```

The most common mistake here was to specify int foo (char *). Students generally assumed that the output type of the register was int, since there was no indicator elsewhere. However, this poses a problem when it comes to recursive calls. By performing the operation "(some integer) & (some character)" in C, the character is SIGN EXTENDED to fill in the upper three bytes when the char is casted to an int (so the types match and they can be and'ed together). The MIPS code provided does no such sign extension- so the result return should be of the form 0x000A. With the sign extension and filling in of 1's, the output would end up as 0x111A (given an ASCII character with a large enough value).

Declaring the prototype as char foo (char *) but also delcaring temporary values as int leads to the same problems listed above, so one point was deducted as well. The only non-standard solutions involved casting values to (unsigned int) and (unsigned char), of which there were many unique and interested combinations.

Other ----------
In some cases, the base case/loops/recursive calls did not exist. Since these solutions could not traverse the string like the MIPS code, the code received no credit.