CS 61C, Fall, 2006, Midterm 1, Garcia

Problem	0	1	2	3	4	5	6	Total
Minutes	0	20	30	40	30	30	30	180
Points	1	11	12	15	12	12	12	75

Question 0:
Name:
Login:
SID:
Section TA:
Person to Left:
Person to Right:
Signature:

Question 1: Warm up jog and Stretch (11 pts, 20 min.)

a) How many different things can we represent with N bits?

b) Given the number 0x811F00FE, what is it interpreted as:

…a binary number?
…an octal (base 8) number?
…four unsigned bytes?
…four two's complement bytes?
…a MIPS instruction?
Use register names (e.g., $a0)

c) During which phase of the process from coding to execution do each of the following things happen? Place the most appropriate letter to the answer next to each statement. Some letters may never be used; others may be used more than once.

__ The stack allocation increases	a) Never
__ jr $ra	b) during loading
__ You give your variables names	c) during garbage collection
__ Your code is automatically optimized	d) while writing higher-level code
__ Jump statements are resolved	e) during the compilation
__ Pseudo-instructions are replaced	f) during assembly
__ a memory leak occurs	g) during linking
__ a jal instruction is executed	h) when malloc is called
__ Symbol and relocation tables are created	i) when free is called
__ The "Buddy System" might be used	j) when a function is called
__ Machine code is copied from disk into memory	k) when a function returns
__ Storage in C-originated-code is garbage collected	l) when registers are spilled
__ MAL is produced	m) during mark and sweep
	n) When there are no more references to allocated memory

Question 2: Old-School Quarter Arcade (12 pts, 30 min)
Early processors had no hardware support for floating point numbers. Suppose you are a game developer for the original 8-bit Nintendo Entertainment System (NES) and wish to represent fractional numbers. You and your engineering team decide to create a variant on the IEEE floating point numbers you call a quarter (for quarter precision floats). It has all the properties of IEEE754 (including denorms, NaN and $\pm\infty$) just with different ranges, precision & representations.

A quarter is a single byte split into the following fields (1 sign, 3 exponent, 4 mantissa): SEEEMMMM the bias of the exponent is 3, and the implicit exponent for denorms is -2.

a) What is the largest number smaller than ∞?

b) What negative denorm is closest to 0? (but not -0)

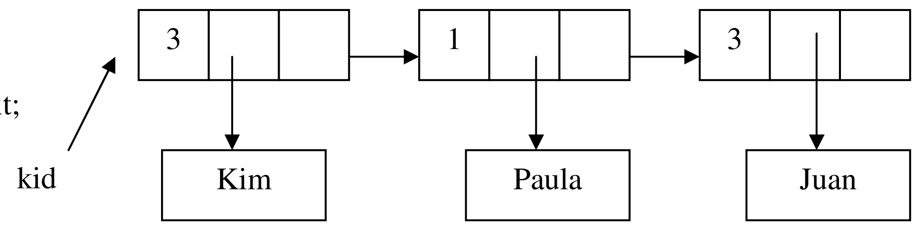
You find it neat how rounding modes affect computation, if at all. You remember the NES carries *one extra guard bit* for computation, so you write the following code to run on your NES. What is the value of c, d, and e? Please express your answer in decimal, but fractions are ok. E.g., -5 3/4 .

```
quarter q1, q2, q3, c, d, e;

q1 = -.25;      /* -1/4  */
q2 = -4.0;
q3 = -0.125;    /* -1/8  */

/* Default rounding mode */
c = q1 + (q2 + q3);

/* Default rounding mode */
d = (q1 + q2) + q3;

SetRoundingMode(TOWARDS_ZERO);
e = (q1 + q2) + q3;
```

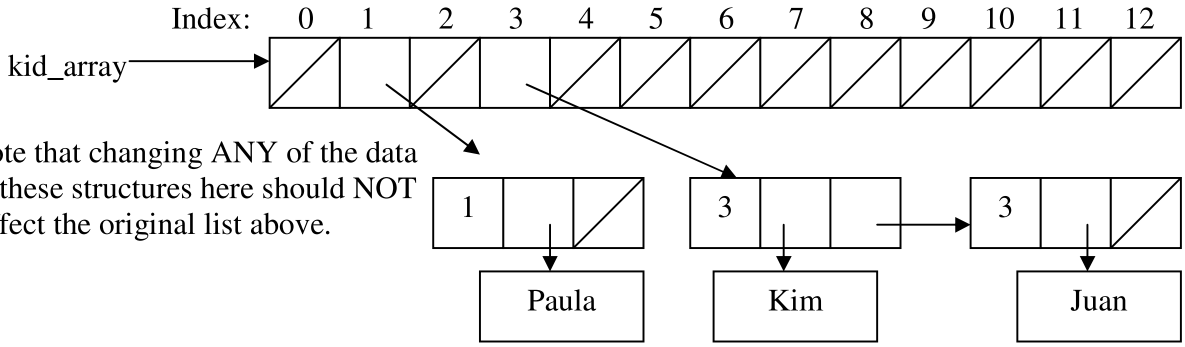
c	
d	
e	

Question 3: You must be kidding! (groan) (15 pts, 40 min)
We have a simple linked list that consists of kids' names (a standard C string) and the grade they are in – an integer between 0 (Kindergarten) and 12. The structure appears as follows, with an example:

```
typdef struct kid_node {
        int grade;
        char *name;
        struct kid_node *next;
} kid_t;
```



For "administrative reasons", we'd like to categorize our kids by **grade.**
We **copy** the kids' information into an array of linked lists indexed by the grade.



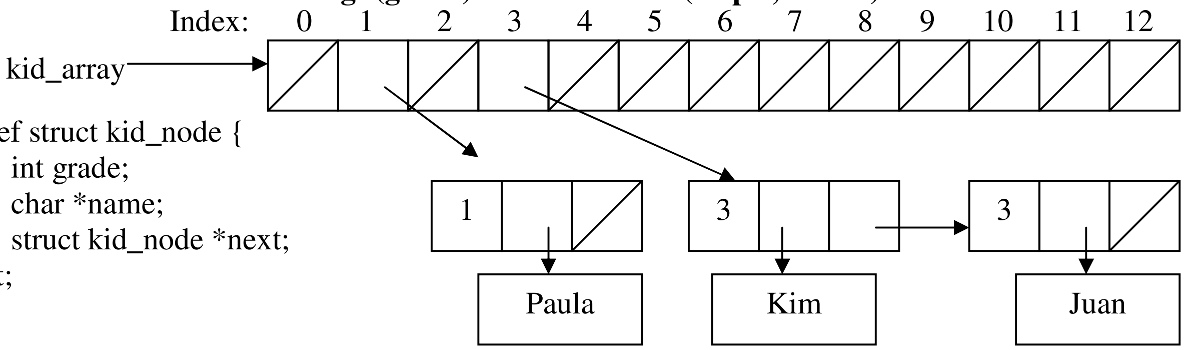
Note that changing ANY of the data
In these structures here should NOT
Affect the original list above.

Fill in the blanks in the below code:

a) The **create_kid_array** function will return a pointer to the new array. Remember, the range of grades is 0-12, inclusive, and the original list MUST remain unchanged.

```
#define MAXGRADE 12
kid_t   **create_kid_array (kid_t *kid) {
        int i;              /* in case you need an int */
        kid_t *temp;   /* or kid_t ptr somewhere */
        kid_t **kid_array = (kid_t **) malloc (________________________);
        if (kid_array == NULL) return NULL; /* malloc has no space! */
        /* Additional initializing – add some code below */

        ret
}
```

Question 3: You must be kidding! (groan) …continued… (15 pts, 40min)




```
Typedef struct kid_node {
        int grade;
        char *name;
        struct kid_node *next;
} kid_t;
```

b) For every Yin, there is a Yang. Now that we have a function for **creating** kid arrays, we must create a function that **frees** all memory associated with the structure. Fill in the following functions. **free_kid_array** calls the *<u>recursive</u>* function **free_kid_list** which frees a single kid list.

```
void free_kid_array(kit_t *kid_array[]){
    int i;
    for (i = 0; i<= MAXGRADE; i++){
            free_kid_list(kid_array[i]);
    }
    /* Clean up if necessary */
    ____________________________________________
}

void free_kid_list(kid_t *kid) {
    [                                   /* Declare temp variables */ ]

    if (kid == NULL)
            return;

    [                                                                ]

}
```

Question 4: That's sum grade you got there, kid! (12 pts, 30 min)

We wish to find out how many cumulative years of schooling our kids had. Conveniently we can calculate that by simply summing all the grade fields from our new linked list of kids. Translate the following recursive C code into recursive MAL-level MIPS.

```
Typedef struct kid_node {
        int grade;
        char *name;
        struct kid_node *next;
} kid_t;
```

```
int grade_sum (kid_t *kid) {
        If (kid == NULL)
                Return 0;
        else
                Return kid->grade + grade_sum(kid->next);
}
```

We started you off; Fill in the blanks. You may not add lines; you may leave lines blank

```
grade_sum:
        beq  _____, _____, NULL_CASE

        ______________________________

        ______________________________

        ______________________________

        ______________________________

        jal grade_sum

        lw      $a0,    0($sp)

        ______________________________

        ______________________________

        ______________________________

        ______________________________

        ______________________________

NULL_CASE:

        ______________________________

        ______________________________
```

Question 5: A little MIPS to C action… (12 pts, 30 min)

You may find this definition handy: sllv (Shift Left Logical Variable): sllv rd, rt, rs *(the contents of the general register* rt *are shifted left by the number of bits specified by the low order five bits contained as contents of general register* rs*, inserting zero into the low order bits of* rt. *The result is place din register* rd.*)* Compiles translate z = x<<y into sllv zReg,xReg,yReg

```
rube:  li     $t0,   0
loop:  andi   $t1,   $a0,   1
       beq    $t1,   $zero, done
       srl    $a0,   $a0,   1
       addiu  $t0,   $t0,   1
       j loop
done:  li     $v0,   0
       li     $t2,   32
       beq    $t2,   $t0,   home
       ori    $v0,   $v0,   $t0
       sllv   $v0,   $v0,   $t0
home:  jr $ra
```

```
int rube (unsigned int x) {
        int i = 0;        /* i is $t0 */








}
```

a) In the box, fill in the C code for rube.

b) Rube can be rewritten as *two TAL instructions!* We've provided the second: what's the first?

```
rube:

       jr      $ra
```

c) How would rube change if we swapped the **srl** with **sra**? Examples might be:

- Rube doesn't change
- Rube now crashes on all input
- Rube is the same, except rube(5) now overflows the stack
- Rube now returns -3 always
- Etc …

Question 6: Memory, all-ocate in the moonlight… (12 pts, 30 min.)

a) Fill in the following table according to the given memory allocation scheme. Show the changes that are made to the memory, if any. Request for memory are in the left column. If a request can't be satisfied, the memory and internal state (e.g., where *next-fit* will start) shouldn't change. Likewise, if there is no prior allocation made for a given **free** call, ignore the action for the given scheme. Assume that if *best-fit* has multiple choices, it will take the first valid one starting from the left. The first rows are filled as an example.

Memory Action	First-Fit						Next-Fit						Best-Fit					
A = malloc(1)	A						A						A					
B = malloc(2)	A	B	B				A	B	B				A	B	B			
free(A)		B	B					B	B					B	B			
C = malloc(1)																		
D = malloc(1)																		
E = malloc(2)																		
Free(B)																		
Free(C)																		
Free(E)																		
F = malloc(2)																		
G = malloc(2)																		

b) Let's compare the performance of a Slab Allocator and Buddy System for 128 bytes of memory (it will be fun!). The Slab Allocator has 2 32-byte blocks, 7 8-byte blocks, and 4 2-byte blocks. All requests to memory are at least 1 byte, and are no more than 64 bytes. For ideal requests (of your choosing), find the limits:

What is the maximum number of requests Slab can satisfy successfully? __________

What is the maximum number of requests Slab can satisfy before failure? __________

What is the maximum number of requests Buddy can satisfy successfully? __________

What is the maximum number of requests Buddy can satisfy before failure? __________

c) My code segment is SOOO big. (audience: How big is it?) It's SOOO big that if I added one more instruction, I would be able to jal to it. (currently I can jal to anywhere in my program). How big is my code segment? Use IEC language, like 16 KibiBytes, 128 YobiBytes, etc.