University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Fall 2006 Instructor: Dr. Dan Garcia 2006-10-16

☺CS61C Midterm☺

Last Name	
First Name	
Student ID Number	
Login	cs61c-
Login First Letter (please circle)	a b c d e f g h i j k l m
Login Second Letter (please circle)	a b c d e f g h i j k l m n o p q r s t u v w x y z
*The name of your **SECTION** TA (please circle)*	Scott Aaron David P. Sameer David J.
Name of the person to your Left	
Name of the person to your Right	
*All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61C who have not taken it yet. **(please sign)***	

Instructions (Read Me!)

- Don't Panic!
- This booklet contains 8 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.

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**
Score								

Name: _____ Login: cs61c-____

Question 1: Warm up jog and stretch (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) How many different things can we represent with N bits?

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

...a binary number?	**`0b`**			
...an octal (base 8) number?	**`0`**			
...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 of the answer next to each statement. Some letters may never be used; others may be used more than once.

	The stack allocation increases
	`jr $ra`
	You give your variables names
	Your code is automatically optimized
	Jump statements are resolved
	Pseudo-instructions are replaced
	A memory leak occurs
	A **`jal`** instruction is executed
	Symbol and relocation tables are created
	The "Buddy System" might be used
	Machine code is copied from disk into memory
	Storage in C-originated-code is garbage collected
	MAL is produced

A) Never
B) During loading
C) During garbage collection
D) While writing higher-level code
E) During the compilation
F) During assembly
G) During linking
H) When **`malloc`** is called
I) When **`free`** is called
J) When a function is called
K) When a function returns
L) When registers are spilled
M) During mark and sweep
N) When there are no more references to allocated memory

Name: _____ Login: cs61c-____

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 IEEE floating point numbers you call a **`quarter`** (for quarter precision floats). It has all the properties of IEEE 754 (including denorms, NaNs and ± ∞) 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.

	Write the binary bit pattern that corresponds to the answer.	Write the base 10 decimal representation; expressions w/exponents and fractions ok.
a) What is the largest number smaller than ∞?	**0b**	DECIMAL
b) What negative denorm is closest to 0? (but not -0)	**0b**	DECIMAL

Show your work here

You find it neat how rounding modes affect computation, if at all. You remember that 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 ¾.

Show your work here

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

q1 = -0.25;  /* -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(TOWARD_ZERO);
e = (q1 + q2) + q3;
```

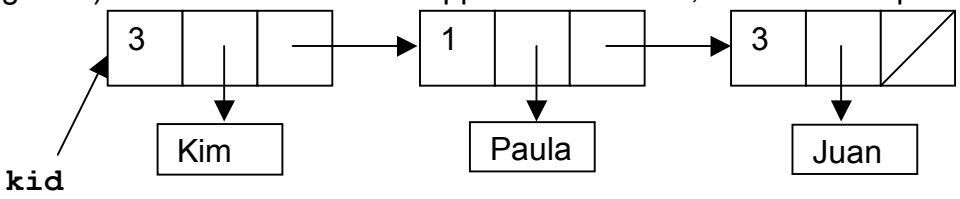
`c`	
`d`	
`e`	

Name: _____ Login: cs61c-____

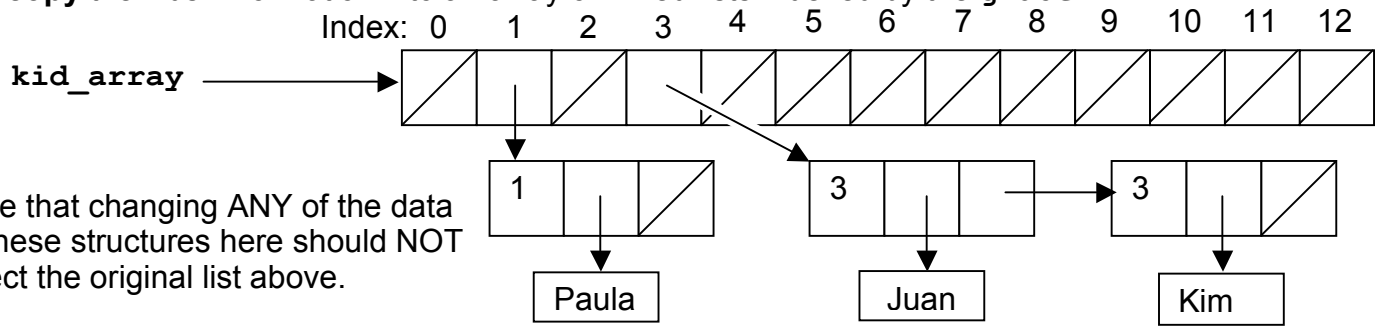
Question 3: You must be **kid**ding! (groan) (15 pts, 40 min)

We have a simple linked list that consists of kids' **name**s (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:

```
typedef 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.

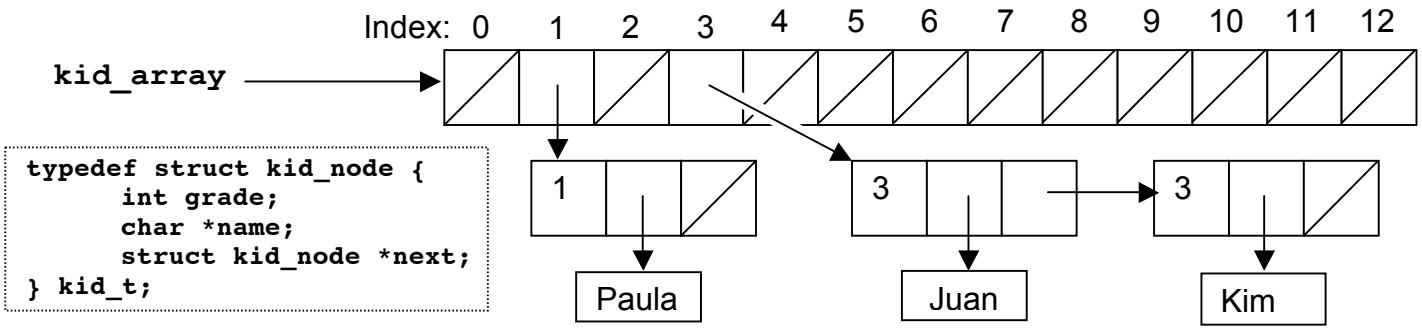
```
#define MAXGRADE 12
kid_t **create_kid_array(kid_t *kid) {
      int i;       /* in case you need an int */
      kid_t *tmp; /* or kid_t ptr somewhere  */
      kid_t **kid_array = (kid_t **) malloc ( _____ );
      if (kid_array == NULL) return NULL; /* malloc has no space! */
      /* Additional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

```
      while (_____!= NULL) { /* populate the array */
```

```
      }
      return kid_array;
}
```

Name: _____ Login: cs61c-____

Question 3: You must be **kid**ding! (groan) ...continued... (15 pts, 40 min)




```
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 *recursive* function **free_kid_list** which frees a single kid list.

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

      ____________________________________________________________
}

/* Remember, this has to be a RECURSIVE function. */
void free_kid_list(kid_t *kid) {
      ┌──────────────────────────────────────────────┐
      │                    /* Declare temp variables */ │
      └──────────────────────────────────────────────┘
      if (kid == NULL)
            return;
      ┌──────────────────────────────────────────────┐
      │                                              │
      │                                              │
      │                                              │
      │                                              │
      │                                              │
      │                                              │
      │                                              │
      │                                              │
      └──────────────────────────────────────────────┘
}
```

Name: _____ Login: cs61c-____

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 have had. Conveniently, we can calculate that by simply summing all the **grade** fields from our new linked list of **kid**s.
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 any lines; you may leave lines blank.

```
grade_sum:                          ### Feel free to write comments below

      beq   _____, _____, NULL_CASE

      _________________________

      _________________________

      _________________________

      _________________________

      jal grade_sum

      lw $a0, 0($sp)

      _________________________

      _________________________

      _________________________

      _________________________

      _________________________

NULL_CASE:

      _________________________

      _________________________
```

Name: _____ Login: cs61c-____

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 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 placed in register* **rd***.)* Compilers 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, $a0, 1
        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…

Name: _____ Login: cs61c-____

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. Requests 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 few rows are filled in 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) Lets 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 minimum number of requests Slab can satisfy before failure?	
What is the maximum number of requests Buddy can satisfy successfully?	
What is the minimum 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 wouldn't 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.