**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* | Answer Key |
| *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<br>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?

| |
|---|
| $2^N$ |

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

<table>
<tr><td>...a binary number?</td><td colspan="4">1000 0001 0001 1111 0000 0000 1111 1110</td></tr>
<tr><td>...an octal (base 8) number?</td><td colspan="4">20107600376</td></tr>
<tr><td>...four unsigned bytes?</td><td>129</td><td>31</td><td>0</td><td>254</td></tr>
<tr><td>...four two's complement bytes?</td><td>-127</td><td>31</td><td>0</td><td>-2</td></tr>
<tr><td>...a MIPS instruction?<br>Use register names (e.g,. <b>$a0</b>).</td><td colspan="4">1000 00|01 000|1 1111 |0000 0000 1111 1110<br>lb $ra, 254($t0)</td></tr>
</table>

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.

| | |
|---|---|
| L | The stack allocation increases |
| K | **jr $ra** |
| D | You give your variables names |
| E | Your code is automatically optimized |
| G | Jump statements are resolved |
| F | Pseudo-instructions are replaced |
| N | A memory leak occurs |
| J | A **jal** instruction is executed |
| F | Symbol and relocation tables are created |
| H,I | The "Buddy System" might be used |
| B | Machine code is copied from disk into memory |
| A | Storage in C-originated-code is garbage collected |
| E | 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

```
;;;;;;;
;; META
;;;;;;;
```

GS  = Grading Standard
CM  = Common Mistakes
PC  = Partial Credit
NPC = No Partial Credit
NB  = Nota Bene (http://en.wikipedia.org/wiki/Nota_Bene)

```
;;;;;;;;;;;;;
;; Question 1
;;;;;;;;;;;;;
```

a) 2^N

(This one may be the easiest question we've ever asked.)
Each bit is an independent binary choice, so we have 2*2*...*2 (n times) = 2^N different possibilities, and each bit pattern maps to a different thing.
GS: 1/2 pt, NPC

b) 0x811F00FE

...binary

Breaking this into binary, we simply map each hex digit into its binary nibble (these answers are on the green sheet!)
0b1000 0001 0001 1111 0000 0000 1111 1110
GS: 1/2 pt, NPC

...octal

We simply work from the *right*, and re-cluster the binary number into threes

```
 1000  0001   0001  1111  0000   0000  1111  1110            (becomes)
10|00 0|001| 000|1 11|11 0|000| 000|0 11|11 1|110
```

which when converted to octal digits becomes (with invisible leading 0s)
020107600376
GS: 1/2 pt, full PC if you had binary wrong but binary->octal correct; else NPC

...four unsigned bytes

We take each byte (two nibbles) and have four separate calculations:
0x81 0x1F 0x00 0xFE
which are simply base-16 values. I.e., 0xXY = X * 16^1 + Y * 16^0 = 16X+Y

```
8*16+1 1*16+15 0*16+0 15*16+15
   129      31      0      254
```

But another way is to think in binary, and remember some common bit-patterns.

```
0b1000 0001 ==> We recall the 8th bit is 128, + 1 = 129
0b0001 1111 ==> This is the biggest 5-bit value = 2^5-1 = 31
0b0000 0000 ==> 0, easy
0b1111 1110 ==> This is one less than the biggest unsigned byte = 2^8-1-1 = 254
```

GS: 1 pt, full PC if you had binary wrong but conversion correct; else NPC
NB: All the TAs were able to do these in their heads given the binary patterns, so that's something you might want to shoot for...

...four two's complement bytes

We chose the numbers to be relatively easy:

```
0b1000 0001 ==> This is one up from the smallest negative number, which we
                recall is -128, so this is -127. No math needed.
0b0001 1111 ==> This is a positive number, so we remember that it will be the
                same as the unsigned byte case.
0b0000 0000 ==> 0, easy
0b1111 1110 ==> This is one less than the biggest unsigned byte = 2^8-1-1 = 254
```

GS: 1 pt, full PC if you had binary wrong but conversion correct; else NPC

...a MIPS instruction?

```
0b1000 0001 0001 1111 0000 0000 1111 1110
smushed together becomes
0b10000001000111110000000011111110
and we look at the left 6 bits for the opcode to figure out what type of inst:
0b 100000 01000111110000000011111110
which is not 0 (R-type) or 2,3 (J-type), so it must be I-type. Let's look up
this number (0b100000 = 32) in the green sheet. Hmm, let's see, it's a "lb".
What's the format for lb? Oh, I remember, it's the same as "lw", which is
lb $rt, OFFSET($rs) -- lbu is described on the other side of the green sheet,
and lb is the same except for sign vs zero-extension of the loaded byte.
Ok, so let's break the instruction up into fields: OPCODE(6) RS(5) RT(5) IMM(16)
(this breakdown is ALSO from the green sheet, man that's a handy reference!)
    opcode|  rs   |  rt   |  immediate
0b 100000 | 01000 | 11111 | 0000000011111110
=     32  |   16  |   31  |     254
=   "lb"  |  $s0  |  $ra  |     254
So that means the instruction represents:
lb $ra 254($s0)
```

GS: 1 pt, full PC if you had binary wrong but conversion correct; else NPC

c) Each box was worth .5 points. If there were multiple possible answers to a question, only one was necessary to receive full credit.

## 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.

| | | |
|---|---|---|
| a) What is the largest number smaller than ∞? | 01101111 | 0b01101111 = 1.1111 * $2^3$ = 1111.1 = 15.5 |
| b) What negative denorm is closest to 0? (but not -0) | 10000001 | 10000001 = -0.0001 * $2^{-2}$ = $-2^{-6}$ = -1/64 |

*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** | -4 ¼ |
| **d** | -4 ½ |
| **e** | -4 ¼ |

;;;;;;;;;;;;;
;; Question 2
;;;;;;;;;;;;;

a) Largest number smaller than infinity...
In quarter (floating point), infinity's representation is 0b01110000, where the exponent field is maxed out and the mantissa is 0. The largest number smaller than infinity? Subtract 1! The result: 0b01101111. Now translate to decimal for the second column.
1.1111 * 2 ^ 3 = 1111.1 = 15.5

b) Negative denorm closest to 0 (but not -0)...
In a quarter, -0's representation is 0b10000000. The smallest number of higher magnitude than -0? Add 1! The result: 0b10000001. NB this is still a denorm because the exponent bits are 0.

GS (for parts a and b):
  1 pt for correct bit pattern (NPC for first column).
  2 pt for correct decimal representation.
  1 pt for decimal representation that is correct with respect to an incorrect bit pattern.

BOTTOM HALF:
q1 = -0.25
q2 = -4.0
q3 = -0.125

For these three problems, q3 will always trump the other values when adding them together because it is so large. What numbers can be represented in a number that has the same exponent field as -4.0? Well, -4.0 is 2^2, so our exponent is 2. Thus, -4.0's representation is 0b11010000. What is it's least significant bit's value? 2^2 * 2^-4 = 2^-2 = 0.25. Thus, adding -0.25 is no problem for a quarter like this, since it fits without a problem in the LSB of the quarter. It's only when we try to add -0.125 that we have to deal with rounding.

NB: Adding -0.125 will put you exactly half-way between two numbers with exponent 2.

c) Default rounding mode == unbiased == round toward even/0
q2 + q3 happens first. -4.0 + -0.125 puts me exactly half-way between -4.0 and -4.25, numbers which can be represented in a quarter with exponent 2! So, which way do we round?
-4.0  = 0b11010000
-4.125 falls right in between
-4.25 = 0b11010001
Round toward even tells us that we should round toward the quarter with a 0 in the LSB, so we round to -4.0.
Finally, add q1, and we end up with -4.25
Answer: -4.25

d) q1 + q2 = -4.25. No problems there
add on q3 (-0.125) puts us exactly half-way between -4.25 and -4.50. Which way do we round using the default rounding mode?
-4.25 = 0b11010001
-4.375 falls right in between
-4.50 = 0b11010010

Round toward a 0 in the LSB, so we round to -4.50.
Answer: -4.50

e) q1 + q2 = -4.25. Again, no problems here, we can represent -4.25.
add on q3 (-0.125), puts us between -4.25 and -4.50. Round toward 0 tells us to round toward the lower magnitude, which would be -4.25.
Answer: -4.25

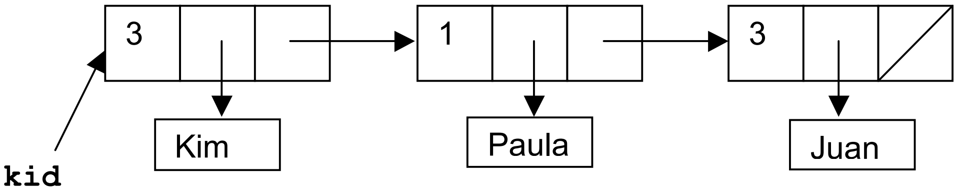
GS: NPC, 2pts per correct answer on parts c, d, and e.

Name: ______________________________ Login: cs61c-____

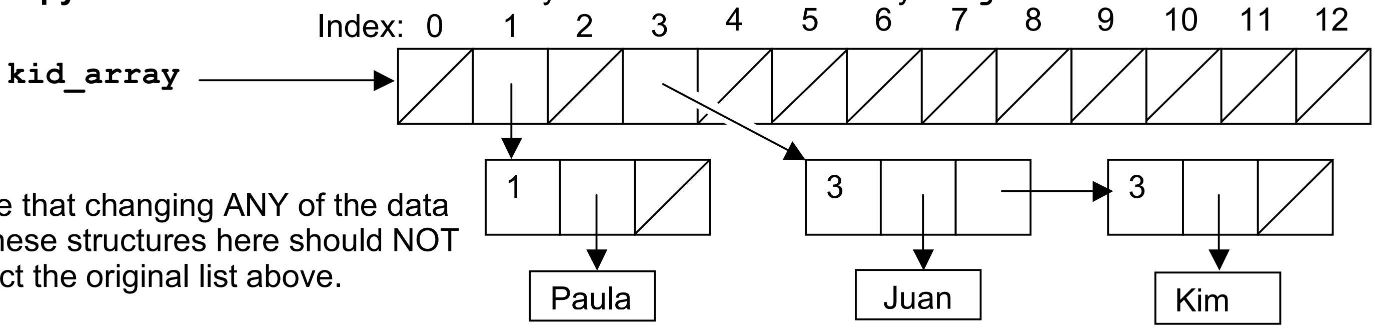
## Question 3: You must be **kid**ding! (groan) (15 pts, 40 min)

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

```
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**.



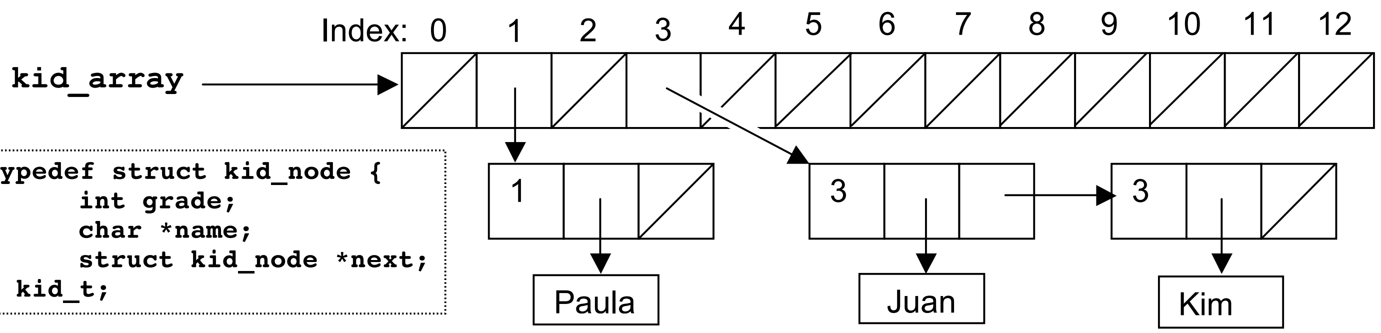
Note that changing ANY of the data in these structures here should NOT affect the original list above.

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 ((MAXGRADE+1)*sizeof(kid_t *));
     if (kid_array == NULL) return NULL; /* malloc has no space! */
     /* Additional initializing - add some code below */
        tmp = (kid_t *) malloc (sizeof(kid_t));
        tmp->grade = kid->grade;
        tmp->name = (char *) malloc (sizeof(char) * (strlen(kid->name)+1));
        strcpy(tmp->name, kid->name);
        tmp->next = kid_array[kid->grade];
        kid_array[kid->grade] = tmp;
        kid = kid->next;
     ret
}
```

## **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 */
      free(kid_array);
      _____________________________________________________________
}

/* Remember, this has to be a RECURSIVE function. */
void free_kid_list(kid_t *kid) {
                                        /* Declare temp variables */
}
```

```
;;;;;;;;;;;;;
;; Question 3
;;;;;;;;;;;;;
```

This question was probably one of the most difficult on the exam. We gave lots of PC in every part of the question.

a)
BLANK 1: (MAXGRADE + 1) * sizeof(kid_t *)
GS: -1 pt if anything here is missing. The most common errors were to forget to use MAXGRADE or to forget to add 1. It's important to use a named constant rather than a literal in your code so that if this number should ever change, it need only be fixed once.

BLANK 2: for(i=0;i<=MAXGRADE;i++) kid_array[i]=NULL;
The step that we were looking for in this blank was the initialization of the kid_array.
NB: after malloc-ing anything, we can make no assumptions about the values in the acquired memory. Thus, it's important to initialize every index of the array to NULL. This step is especially necessary to make part b) function properly (testing NULL as a base case for the recursive free-ing). Further, in order to build an array like the one in the diagrams, every element that's not a linked list of kid_t's needs to be NULL.
GS: Cannot lose more than 2 points...
 -2 if the initialization of the array is missing.
 -1 for other mistakes, such as only going to <MAXGRADE or using a number instead of MAXGRADE.

BLANK 2: kid
GS: -1 pt if this does not somehow allow iteration through the linked list in your algorithm. We did the best we could to figure out what your algorithm was and generally only took a point if the loop termination was missing.

BLANK 3:
GS: -1 pt for each of the following missing steps. Cannot lose more than 6 pts. If neither a new kid_t nor the name are malloc-ed, -5 (cannot earn more than 1 pt for this blank).

Steps we were expecting:
1) malloc a new kid_t (NB: don't forget sizeof!)
2) copy the grade from the original to the new kid_t
3) malloc a new string for the name (NB: don't forget to allocate an extra char for the null terminator!) (NB: also, don't forget that you must always multiply by sizeof(char))
4) copy the name from the original to the new kid_t (strcpy if you did 3, = otherwise)
5) attach the new node to the linked list in the kid_array
6) (if you placed the node at the end of the linked list, this is unnecessary) modify kid_array to point to the new node
7) move to the next node in the original linked list of kids (NB: kid++ would be used to move to the next index in an array, but is not appropriate for a linked list. Use kid=kid->next.)

b)
BLANK 1: free(kid_array);
GS: -1 pt if kid_array not freed

NB: Do not forget, just because you freed the linked lists that the array contains, you can't assume that the array itself was freed.

BLANK 2: Optional (use if you need an addiitonal variable)

BLANK 3:
GS: cannot lose more than 6 pts
-2 pts if missing the recursive call (free_kid_list(kid->next))
-2 pts if missing the free of the kid
-2 pts if missing the free of the kid's name (we only took off 1 pt if name wasn't malloc-ed)
-3 pts if all components are there but the order is wrong (NB: Cannot dereference freed memory, so one mustn't free the kid before his name, or the kid before the rest of the list, unless the pointers were saved before freeing)
-1 pts for other small errors

GS: First page was out of 8, Second page was out of 7

## 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 **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 any lines; you may leave lines blank.

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

      beq    _____, _____, NULL_CASE beq $a0, $0, NULL_CASE

      ____________________________   addi $sp, $sp, -8

      ____________________________   sw $ra, 4($sp)

      ____________________________   sw $a0, 0($sp)

      ____________________________   lw $a0, 8($a0)

      jal grade_sum

      lw $a0, 0($sp)

      ____________________________   lw $t0, 0($a0)

      ____________________________   lw $ra, 4($sp)

      ____________________________   addu $v0, $v0, $t0

      ____________________________   addi $sp, $sp, 8

      ____________________________   jr $ra

NULL_CASE:

      ____________________________   move $v0, $zero

      ____________________________   jr $ra
```

```
;;;;;;;;;;;;;
;; Question 4
;;;;;;;;;;;;;
```

Most of the problem was knowing register conventions. Six of the 12 instructions involved this.

CM: Getting $a0 set to what should be kid->next proved to be most common mistake. In the solution this is done by lw $a0, 8($a0). Many people had:

```
addi $a0, $a0, 8.
```

The problem is that kid->next is a pointer. If you used addi, it is calculating the address of the pointer, while if you used lw, it actually gets the value of the pointer.

Many people forgot to save $ra on the stack. It must be saved if you call another function (even if it is yourself).

GS: 12pts total, PC possible
Since the solution uses 12 instructions, 1 point was awarded for each correct instruction. If you did the same net operation but with a different instruction, 1 point was awarded for each instruction you accomplished the equivalent of. For very wrong solutions it was harder to award partial credit because it was difficult to determine what some registers were intended to be used as.

NB: Almost across the board, those who wrote good comments did better on this problem. Forcing yourself to write good comments really makes you think about what you are doing and helps you understand it better. It also makes it easier for graders to see what you were trying to do if you got it wrong. Almost every midterm with solid comments got near or full points on the problem. Coincidence? probably not....

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 */

     while (x & 1){
          x = x >> 1;
          i++;
     }

     return (i == 32) ? 0 :
          (x | 1) << i;

}
```

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: addiu $v0, $a0, 1

      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…

**rube** is the same, except with an input of all 1s, i.e., **rube(2^32-1)** = **rube(-1)** now hangs (runs forever in **loop** waiting for a 0 on the LSB)

```
;;;;;;;;;;;;;
;; Question 5
;;;;;;;;;;;;;
```

a)

```
int rube (unsigned int x){
     int i = 0; /* i is $t0  */
     while (x & 1){
          x = x>>1;
          i++;
     }
     return (i==32)? 0: (x | 1) << i;
}
```

A pretty direct translation of the original MIPS. We take a few shortcuts by not comparing (x&1) to 0 and constructing the loop as well as compressing the return statement into a single inline conditional.

GS: 7pts. total
1 pt. PC for each of the salient features in the MIPS seen in the C.

- loop condition
- shifting x right by 1 (or dividing by 2)
- incrementing i
- checking i vs 32
- oring x with 1
- shifting x left by i
- returing 0 or x

b)

```
addiu $v0, $a0, 1
```

GS: 2 pts total
1 point PC for getting the correct add (addiu), 1 point PC for the correct arguments. Note that there is no partial credit (NPC) given for answers not involving an add instruction.

c) rube is the same for all arguments other than -1 (all 1s in the binary representation). If there were a 0 anywhere in the bit pattern, it would be shifted right and eventually found. Then any 1s acquired through sign extension would be removed when the result is shifted left again. Accordingly, rube(-1) will now infinitely loop, since shifting right makes no progress.

GS: 3 pts total
2 PC points for recognizing that the error will occur on -1
1 PC point for recognizing that the error is an infinite loop

CM: There is no danger of stack overflow in this function, since there is no recursive call. Without decrementing the stack pointer, it is impossible to create a stack overflow.

## 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)`** | C | B | B | | | | | B | B | C | | | C | B | B | | | |
| **`D = malloc(1)`** | C | B | B | D | | | | B | B | C | D | | C | B | B | D | | |
| **`E = malloc(2)`** | C | B | B | D | E | E | | B | B | C | D | | C | B | B | D | E | E |
| **`free(B)`** | C | | | D | E | E | | | | C | D | | C | | | D | E | E |
| **`free(C)`** | | | | D | E | E | | | | | D | | | | | D | E | E |
| **`free(E)`** | | | | D | | | | | | | D | | | | | D | | |
| **`F = malloc(2)`** | F | F | | D | | | F | F | | | D | | | | | D | F | F |
| **`G = malloc(2)`** | F | F | | D | G | G | F | F | G | G | D | | G | G | | D | F | F |

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? | 13 |
| What is the minimum number of requests Slab can satisfy before failure? | 0 |
| What is the maximum number of requests Buddy can satisfy successfully? | 128 |
| What is the minimum number of requests Buddy can satisfy before failure? | 2 |

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.

`256 MebiBytes`

;;;;;;;;;;;;;
;; Question 6
;;;;;;;;;;;;;

a)
First-Fit: As you can see the answer has simply placed the memory in the most left columns possible (first fit).
Next-Fit: After every allocation a "next fit" arrow can be kept after each memory block. Place memory there if it fits - and if it doesn't, move the arrow forward (it can circle around) until it does.
Best-Fit: Simply place each memory block in the most compact spot it can fit in.

GS: 2 points for each part. All or nothing
CM: Some students had assumed memory is circular (i.e. in next-fit would place E in both the first and last box).
Also on best fit, many students allocated FF in the left 2 boxes. Note though that where it is placed in the solution is in fact the "best fit" (2 slots rather than 3).

b) Maximum for slab: The maximum is the total number of blocks, which is 13.
   Minimum for slab: 0. If we request a block larger than 32, slab will fail (as the largest block is 32 bytes)
   Maximum for buddy: 128. If we request 128 1 byte blocks, all will fit - until buddy fills up.
   Minimum for buddy: 2. The largest size we can request is 64 bytes (recall that we round up to the next power of 2). Therefore, we must allocate twice to fail.
GS: 1 point for each part. All or nothing (NPC).

c) jal takes a 26 bit immediate. To calculate the new PC the following formula is used: newPC = (PC & 0xf) | immediate << 2 (See green sheet). In other words, the 26 bit immediate is word addressed, which means we can effectively change (4 bytes in a word) 28 bits of the PC. And 2^28 bytes = 256 Mebibytes.
GS: 1 point for each part. All or nothing (NPC).