University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Science

Spring 2004 Instructor: Dan Garcia 2004-03-08

☺ CS61C Midterm ☺

Last Name	
First Name	
Student ID Number	
Login	`cs61c-`
The name of your TA (please circle)	Alex Chema Jeremy Paul Roy
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

- This booklet contains 6 numbered pages including the cover page plus photocopied pages from COD and K&R. 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 points if done incorrectly) involves filling in the front of this page and putting your name & login on every sheet of paper.
- You have 180 minutes to complete this exam. The times listed by each problem will allow you to finish with 45 (!) minutes left to check your answers. The exam is closed book, no computers, PDAs or calculators. You may use one page (US Letter, front and back) of notes.
- 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**	**7**	**8**	**9**	**Total**
Min		2	8	20	30	30	30	15	1	14	150
Max	**0/-1**	**2**	**4**	**8**	**16**	**14**	**18**	**6**	**3**	**4**	**75**
Score											

Question 1: Big Ideas (2 Points – 2 minutes)

We've discussed four design principles that guide the authors of instruction sets (& played a part in MIPS design). **What is one of them and how did it affect the design?** Be as brief as possible. We've shown one of them for you to refresh your memory.

Design Principle	How was the MIPS design affected?
Smaller is faster	*MIPS has 32 registers, rather than many more.*

Question 2: Numerical Representation (4 Points – 8 minutes)

a) Below is a table corresponding to the different systems for representing #s. Fill in the six blanks in the table. Each column should contain the same #, written different ways. Show your work below.

Decimal (base 10)	**-3_{10}**	
8 bit Sign-Magnitude (in hex)		
8 bit One's Complement (in hex)		**0x80**
8 bit Two's Complement (in hex)		

Scratch space

b) We've seen the decimal point and the binary point, but as you can guess, there's also a hex point. Fill in the table below. NB: This is a different question than (a) above – there is no encoding here.

Decimal (base 10)	**-18.25_{10}**	
Hexadecimal (base 16)		**20.2_{16}**

Scratch space

Question 3: Floating Point (8 Points – 20 minutes)

a) Shown below is a number whose value is described by the fields (sign, exponent, significand) of the IEEE 754 32-bit floating-point standard. What is the next-largest (closest to it but larger [closer to +∞] than it) number that can be represented? Write it in the same format in the blanks below.

1_2	**11100000_2**	**$00\ldots0_2$**	**Next-largest ⇒**				

b) Using IEEE 754 32-bit floating point, what is the **largest positive number *x*** that makes this expression true: ***x* + 1.0 = 1.0**? Assume there is no rounding with extra guard/rounding bits (i.e., we truncate the bits outside the given fractional mantissa field). Write the answer in the same format as in (a) above. Show all work!

Show your work below

Name: ____________________________ Login: cs61c-____

Question 4: C (16 Points – 30 minutes)

We've written **matchSubStr** below in C. Some of the lines are buggy and some are perfectly fine. Circle BUGGY or OK for each labeled C statement and if buggy, why it is and provide the fix. A line may be buggy for multiple reasons, so be sure you're descriptive.

Use the comments near each statement as a guide for what the line SHOULD do. If the code is buggy and you have a more clever/intuitive way of doing the same thing, feel free to do it your way. Note: You can assume only valid input will be provided (two non-empty, null-terminated strings).

```
/* This function tries to find a substring (sub) within another (string).
 * If matchSubStr() finds the substring, it returns the index of the start
 * of the substring.  If there is more than one match, it returns the first.
 * This is the scheme-equivalent of an equal? match (not eq? match) */
int matchSubStr(char sub[], char string[]) {

/* Holds the location we're checking (and will return if a match). */
A:  int loc;        || BUGGY If buggy, why?
                    || OK    If buggy, fix:

    /* These variables are pointers to the chars in sub/string */
B:  char *c1, c2;   || BUGGY If buggy, why?
                    || OK    If buggy, fix:

    /* We want to iterate through the string looking for a match, so we start at
     * loc=0 (beginning) and keep going as long as we have characters remaining */
C:  for(loc=0; strlen(string[loc]); loc++) { || BUGGY If buggy, why?
                                             || OK    If buggy, fix:

        /* We step through the substring using c1 and c2 to reference the
         * letters in sub and string.  We stop when we have either exhausted
         * all the characters in sub (and thus found a match) or when we
         * encounter two characters that are not equivalent. */

D:      for(c1 = sub, c2 = string&loc;   || BUGGY If buggy, why?
                                         || OK    If buggy, fix:

E:          ;                            || BUGGY If buggy, why?
                                         || OK    If buggy, fix:

F:          c1++, c2++) {                || BUGGY If buggy, why?
                                         || OK    If buggy, fix:

            /* If we didn't find a match, we break out */
G:          if(c1 != c2) {  || BUGGY If buggy, why?
                            || OK    If buggy, fix:
                break;
            }
        }

        /* We return the location if we found a match */
H:      if(1) {             || BUGGY If buggy, why?
                            || OK    If buggy, fix:
            return loc;
        }
    }
    /* Return -1 if we didn't find a match */
    return -1;
}
```

Question 5: MIPS Assembly Language (14 Points – 30 minutes)

a) Below is a function written in C and the same function partially compiled into MAL. Fill in the blanks (and the comments!) to complete the compilation. Use register names, not #s. (8 points)

```
int *replaceInt(int *array, int toReplace, int replaceWith) {
      for(;*array; array++) {
            if(*array == toReplace) {
                  array[0] = replaceWith;
                  return array;
            }
      }
      return NULL;
}
```

```
line #
  0    replaceInt:    ___     ___ ___(____)
  1                   beq     ___ ___ endLoop        # We're done
  2                   beq     ___ ___ doReplace      # Let's replace it
  3                   addiu   ___ $a0 ___            # _______________________
  4                   j       ________________
  5    doReplace:     ____    ___ ___(____)          # _______________________
  6                   ____    ___ ___                # _______________________
  7                   j       ret
  8    endLoop:       move    ___ ___                # return NULL
  9    ret:           jr      $ra
```

b) Now, provide us with the MIPS code that would correspond to the following C function call. You may not need all the lines (or blanks) below. Note: `myArray` starts at `8($sp)`. (3 points)

```
replaceInt(myArray, 1, 2)
```

____

____

____

____

____

c) Optimize the code above and reduce the number of instructions to fewer than 10. You can do this through slight adjustments of fewer than *four* lines of code. Your answer should be in the form of directives that tell us how the code will be changed: "`Move line __ to __ (and change __)`". If your destination is *between* two lines, use fractional line numbers. Your last command should be "`Delete line __`". E.g., if you wanted to move line 2 right after line 4 (but have *it* now be labeled `doReplace`) you would write: "`move line` **`2`** `to` **`4.5`** `and change` **`the doReplace label to be at line 4.5`**". You may not need all the lines below; leave "`and change __`" blank if not nec.). (3 pts)

```
Move line __ to __ (and change _____________________________________________)
Move line __ to __ (and change _____________________________________________)
Move line __ to __ (and change _____________________________________________)
Delete line __
```

Name: ______________________________ Login: cs61c-____

Question 6: MIPS Reverse-Engineering (18 Points – 30 Minutes)

a) You have heard of Jedi hackers reverse-engineering programs. Prove you belong in that elite group by converting the following MIPS function mystery into C code. Show your work by adding comments to the code to help you understand it. (10 points)

```
 1 mystery: bnez $a0, recur    #
 2          li   $v0, 0        #
 3          jr   $ra           #
 4 recur:   sub  $sp, $sp, 8   #
 5          sw   $ra, 4($sp)   #
 6          sub  $a0, $a0, 1   #
 7          jal  mystery       #
 8          sw   $v0, 0($sp)   #
 9          jal  mystery       #
10          lw   $t0, 0($sp)   #
11          addu $v0, $v0, $t0 #
12          addu $v0, $v0, 1   #
13          add  $a0, $a0, 1   #
14          lw   $ra, 4($sp)   #
15          add  $sp, $sp, 8   #
16          jr   $ra           #
```

```
int mystery(int n) {
```

b) You may have noticed that `mystery` doesn't follow proper register conventions but somehow works. Tell us which line in particular is the most blatant offender and what's wrong with it? (2 pts)

Line #	**How does it violate register conventions?**

c) What is printed as a result of `printf("%d", mystery(32))`? Show your work. (4 points)

d) What is printed as a result of `printf("%d", mystery(34))`? Show your work. (2 points)

Question 7: MIPS → Binary (6 points, 15 minutes)

Assemble the following code assuming that the label '`code`' corresponds to the address `0x00080000`. You should fill in the table below with the hexadecimal value for the instruction. You must show your work to receive credit.

```
code:        addiu $v0, $zero, 0      # Inst 01 → 0x24020000
while_loop:  lw $v1,-4($a0)           # Inst 02 → ____________
             beq $v1,$0,loop_exit     # Inst 03 → ____________
             # There are 10 instructions here
             j while_loop             # Inst 14 → ____________

loop_exit:   …
```

Question 8: Starting a Program (3 points, 1 minute)

For each of the tasks below, label it with the first two letters of the system whose job it is: COmpiler, ASsembler, LInker, or LOader.

___ Resolve undefined labels using the relocation information and symbol table

___ Copy the parameters (if any) to the main program onto the stack

___ Change `move $t0,$t1` into `add $t0,$zero,$t1`

Question 9: Memory Management (4 points, 14 minutes)

Assume a simplistic view of 5 bytes of memory, and no header overhead. Further, assume *best-fit*, when given multiple identical "best" options, chooses the space closest to the head of the freelist. Also assume both *first-fit* and *best-fit* always "flush-left" within a chunk of free space (i.e., allocates the memory closest to the head of the freelist). These are shown below with two identical memories:

```
        FIRST-FIT  BEST-FIT  COMMAND        /* COMMENT */
LABEL:   01234      01234
MEMORY:  -----      -----                   /* Begin, 5 contiguous bytes of memory free */
         AA---      AA---    A=malloc(2);   /* Both first- and best-fit "flush-left" */
         AAB--      AAB--    B=malloc(1);   /* A simple request */
         AABC-      AABC-    C=malloc(1);   /* Another simple request…memory almost full */
         AA-C-      AA-C-    free(B);       /* Freeing B fragments our memory: 2 1-bytes */
         AADC-      AADC-    D=malloc(1);   /* Note best-fit chose space 2 over 4 */
```

Will *first-fit* ever succeed with a call to `malloc` where *best-fit* would fail? If so, draw memories (as above, except you may initialize them to any state, as long as they are equal) and give a **short sequence** of `malloc` and `free` calls that proves your point. If not, explain **in at most 3 sentences**.

Yes, *first-fit* will succeed where *best-fit* would fail. (fill in a table below just like shown above)	**No**, *first-fit* will never succeed where *best-fit* would fail. (answer below in at most 3 sentences)
`FIRST-FIT  BEST-FIT  COMMAND` `LABEL:   01234      01234`	