

# University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Spring 2010

Instructor: Dr. Dan Garcia

2010-03-09



After the exam, indicate on the line above where you fall in the emotion spectrum between “sad” & “smiley”...

<i>Last Name</i>	
<i>First Name</i>	
<i>Student ID Number</i>	
<i>Login</i>	cs61c-
<i>Login First Letter (please circle)</i>	a b c d e f g h i j k l m
<i>Login Second Letter (please circle)</i>	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
<i>The name of your LAB TA (please circle)</i>	Bing Eric Long Michael Scott
<i>Name of the person to your Left</i>	
<i>Name of the person to your Right</i>	
<i>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)</i>	

## a) Instructions (Read Me!)

- Don't Panic!
- This booklet contains 7 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 login on every sheet of paper.
- You have 180 minutes to complete this exam. The exam is open book, no computers, PDAs, calculators.
- 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.

Question	0	1	2	3	4	5	Total
Minutes	1	45	15	30	30	30	(+30 to review) = 180
Points	1	20	10	18	11	15	75
Score							

## Question 1: Brian Harvey would be proud... (20 pts, 45 min)

Working with a partner, you want to implement a scheme *sentence* ADT (a linked list of words), and you're going to start with the constructor `se(word, sentence)`. Your partner writes the interface, and shows you how they want to call `se`, and your job will be to write the subroutine. You agree on the standard `struct` definition on the right.

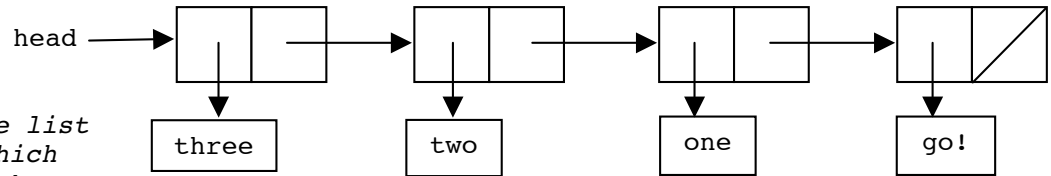
```
// Assume compiler packs tight
typedef struct sentence_node {
    char *word;
    struct sentence_node *next;
} sentence_t;
```

Here's how your friend wants to call `se`:

```
#include <string.h>
#include <stdio.h>
int i;
int main() {
    char word[8];
    sentence_t *head = (sentence_t *) malloc (sizeof(sentence_t)); // assume succeeds...
    head->word = "go!";
    head->next = NULL;
    for (i=1;i<=3;i++) {
        printf("Word? "); scanf("%s",word); se(word,head);
    }
    // ← Right here, if you ran this program, and typed the following:
}
```

```
// Word? one
// Word? two
// Word? three
```

```
// ...it'd create the list
// to the right, which
// in scheme would be:
// (three two one go!)
```



- a) Complete `se` so it will work with your friend's `main` code, even if he later supports longer words. Think about error checking you might need, and call `exit_with_msg` below, if needed. *Your solution should not need any looping/recursion, just statements that connect things up correctly.*

```
void se(char *word, sentence_t *head) {
    sentence_t *new = (sentence_t *) malloc (sizeof(sentence_t));
```

```
    new->word =
    new->next =
```

```
}
```

```
void exit_with_msg(char *msg) { printf("Error! %s\n",msg); exit(1); }
```

- b) At “← Right here”, how much space (Bytes) is used by `i`, `word`, & `head` (and their data) together?

Stack	Heap	Static	Code

**Question 1: Brian Harvey would be proud...** (continued) (20 pts, 45 min)

That interface is counterintuitive, you say. You decide the interface should be *changed* so that the boxed call to `se` looks like this: `head = se(word, head)` and you now ask *them* to write `se`, but to save space, only keep ONE copy of any word. That is, they should check if the `word` is already in your sentence, and if it is, to *share* it. They write the following but you suspect `share_or_new` has bugs.

```

sentence_t *se(char *word, sentence_t *head) {
    sentence_t *new = (sentence_t *) malloc (sizeof(sentence_t)); // assume succeeds...
    new->word = share_or_new(word, head);
    new->next = head;
    return new;
}

char *share_or_new(char *word, sentence_t *head) {
1     if (word = head->word)           // Found it, share!
2         return head->word;
3     if(head == NULL) {               // Didn't find it, create one
4         char new[10];
5         strcpy(new,word);
6         return new;
7     }
8     return share_or_new(word, head->next); // Keep looking..
}

```

- c) Simulate running the program with the same user input as before. If the code returns without error, show in scheme format, like (three two one go!), what head is at “**← Right here**”. If the code crashes, indicate the value of `i` when the error occurs. \_\_\_\_\_
- d) Just for fun, change the “=” in line 1 to “==” (make that change permanent in the code above). Instead of our previous user input, we type the word `go!` three times. If there is **no** crash by the time we get to “**← Right here**”, draw the *box-and-pointer diagram* for `head` (just like on the previous page). If there **is** a crash, list the functions in the stack frame before the error. E.g., if `main` called `sub`, which called `subsub`, which crashed, you’d write: `main→sub→subsub→ERROR`.
- 
- e) Finally, fix all the bugs so `se` works in general (not just for this particular `main`). Your fixes can be of the form: “change line `i` to (fill in the blank)”, or “insert (fill in the blank) before/after line `i`”, or “move lines `i` through `j` above line `k`”. You may not need all the blanks.
- 
- 
- 
- 
- 
-

**Question 2: L1 and L2 below are Booth needed for the algorithm... (10 pts, 15 min)**

Below is a MIPS function. Read it carefully and use it to answer the following questions.

```
myst:  move $v0, $0
L1:    andi $t0, $a0, 1
       beq  $t0, $0, L2
       addu $v0, $v0, $a1
L2:    sll  $a1, $a1, 1
       srl  $a0, $a0, 1
       bne  $a0, $0, L1
       jr  $ra
```

- a) In one sentence, briefly summarize what `myst` does, precisely (assume the inputs are unsigned). **Don't** describe the algorithm, abstract at a high level what the algorithm is doing. (e.g., "it returns the # of bits in common of its three arguments" or "sets `$a1` to `-$a0`")

**For question (b) only**, assume `$a1` is a two's complement negative number.

- b) Would `myst` still work? Briefly explain why or why not.

**For question (c) and (d)**, let's change the `srl` to `sra`.

- c) For what values of its inputs would `myst` still do what you said it does in (a)?

- d) When it *doesn't* return the same answer, what does `myst` return / do?

**Question 3: Tasting Menu... (18 pts, 30 min)**

Thanks to a breakthrough, we can store 3 values per digit instead of the usual 2. Rather than encoding 0, 1, and 2, we choose to encode -1, 0, and 1 (We call these binary-with-negative base 2 digits *binets*). To make notation cleaner, we'll use 4 to represent -1, and we'll precede binets with 0bn. Binet numbers allow us to express negative numbers with an "unsigned" encoding, which is great. However, now some numbers have multiple representations, like  $1_{10} = 0bn0001 = 0bn001\pm$  (i.e.,  $1*2^1 + -1*2^0 = 1$ )

a) How many unique numbers can be represented using N unsigned binets? \_\_\_\_\_

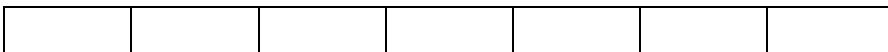
b) Recall that we negated 2's complement binary numbers by inverting each bit and then adding a constant offset (1). It turns out we can negate unsigned binets with a similar technique! Fill in the truth table for what the binet "inversion" function should do *per bit*, and the constant offset that should be added at the end to make it work perfectly.

Original bit	New bit	Constant Offset
4		
0		
1		

c) List all the representation(s) for zero for a binet nibble (4 binets). \_\_\_\_\_

Assume the heap has 7 contiguous free bytes left (below), and *best fit*, given two equally good candidates, will pick the leftmost one.

d) Fill in the gaps in the C snippet on the right such that the location of *e* is *different* depending on whether `malloc` uses first-fit, next-fit, or best-fit. Indicate in the diagram below where the remaining data is (by a single letter) and where the fits would place *e* (by writing `FIRST`, `NEXT` and `BEST` in the appropriate slot).



```

char *a, *b, *c, *d, *e;
a = (char*) malloc(____);
b = (char*) malloc(____);
c = (char*) malloc(____);
d = (char*) malloc(____);
free(____);
free(____);
e = (char*) malloc(1);
    
```

e) We're using C99 and need to store *n* integers, but can't decide how to reserve space for them. What is the **best argument** for using each one? Also, if `$_s0` contains the value of *n*, what *minimal* MAL MIPS would the C translate to, that reserves space and puts `&A[0]` (i.e., where *A* points) in `$_s1`? All volatile registers have been saved & we'll soon need `$_s0` and `$_s1`. (Note: `malloc` is just a function)

	<code>int A[n];</code>	<code>int *A = (int *)malloc(sizeof(int)*n);</code>
Why?		
MAL MIPS		

f) How many *total different instructions* could we list on the green sheet if, instead of an explicit `shamt` field, we stored the shift amt in the unused `rs`, register and expanded the `funct` field to use the `shamt` bits also? Be exact. \_\_\_\_\_

**Question 4: Did somebody say “Free Lunch”?! (11 pts, 30 min)**

Consider two competing 8-bit floating point formats. Each contains the same fields (sign, exponent, significand) and follows the same general rules as the 32-bit IEEE standard (denorms, biased exponent, non-numeric values, etc.), but allocates its bits differently. To save you time, *you only need to complete and circle the (LEFT or RIGHT) blank whose value is closest to zero*, that’s the only one we’ll grade! (If they’re the same value, write the answer in both, & circle both). E.g.,. The number represented by  $0_{\times}00$  was 0 for both, so we circled both. But for “exponent bias”, just from the # of EE...E bits in each, we know  $|\text{LEFT’s bias}| < |\text{RIGHT’s bias}|$ , so there’s no need to calculate or write the answer on the RIGHT.

“LEFT” format:



scratch space (show all work here)

“RIGHT” format:



scratch space (show all work here)

Number represented by  $0_{\times}00$ : 0

Exponent Bias: 1

a) # Numbers ( $0 \leq n < 1$ ): \_\_\_\_\_

b) # Numbers ( $1 \leq n < 2$ ): \_\_\_\_\_

c) Difference between two smallest positive values: \_\_\_\_\_

d) Difference between two biggest non- $\infty$  values: \_\_\_\_\_

e) Positive Integer closest to 0 it cannot represent: \_\_\_\_\_

Number represented by  $0_{\times}00$ : 0

Exponent Bias: 31 (not graded, so no need to write)

# Numbers ( $0 \leq n < 1$ ): \_\_\_\_\_

# Numbers ( $1 \leq n < 2$ ): \_\_\_\_\_

Difference between two smallest positive values: \_\_\_\_\_

Difference between two biggest non- $\infty$  values: \_\_\_\_\_

Positive Integer closest to 0 it cannot represent: \_\_\_\_\_

f) Which implementation is better for *approximating*  $\pi$ , LEFT or RIGHT ? (circle one)

**Question 5: Euclid’s Revenge...** (15 pts, 30 min)

The *Euclidean Algorithm* is used to find the greatest common divisor (GCD) of two numbers  $a$  and  $b$ . E.g.,  $\text{GCD}(8,6)=2$ . The algorithm works because ( $a \geq b$ ):  $\text{GCD}(a,0)=a$ ,  $\text{GCD}(a, b) = \text{GCD}(b, a \bmod b)$ .

a) Implement the Euclidean Algorithm below recursively (but as efficiently as you can) in MAL MIPS. Follow the hints given by the comments; you may not need all the lines. To assist you, assume that `mod` is correctly implemented as a MIPS subroutine that returns  $a \bmod b$  in  $\$v0$ . The first letter of the first MIPS instruction is `b`.

```
GCD:  b _____ # Handle base case first; we assume a ≥ b
_____
_____
_____
j done
rec:  _____ # Recursive case
_____
_____
_____
_____
_____
_____
_____
_____
jal GCD
_____
_____
_____
_____
done: jr $ra
```

For questions (b) & (c), assume the address of GCD is 0x1000008

- b) Translate the first MIPS instruction to hex: \_\_\_\_\_
- c) Translate the `jal GCD` to hex: \_\_\_\_\_
- d) You notice your code sometimes crashes when you call GCD with large numbers. Circle the instruction that causes the crash, and in *two words*, explain why does it crash. \_\_\_\_\_
- e) In *two words*, what can you do to combat this instability? \_\_\_\_\_