CS 61C Fall 2015
Guerrilla Section 1: Number Representation & C

Question 0: *Silly Rabbit, Trits Are for Kids*

A new memory technology with three distinct states is exploding into the technology industry! Let's see if we can't develop some new number representations to take advantage of this new development.

(a) First, define a rule, analogous to what we use for binary numerals, for determining the unsigned value of a ternary numeral d_n, d_{n-1} . . . d_0, and use this rule to convert 2102_3 into decimal:

unsigned(d_n, d_{n-1} . . . d_0) = _____

unsigned(2102_3) = _____

(b) Next we'd like to define an analogue to two's complement for ternary numerals, which we'll call three's complement. Three's complement numbers should be as evenly distributed between positive and negative as possible (favor negative if necessary), should have a zero at 0_3, and should increase in value when incremented as an unsigned value (except in the case of overflow). Define a rule for negating a three's complement number.

(c) What is the most positive possible three's complement 8-trit number? Using this result, specify a rule for determining if a three's complement number is positive or negative.

(d) There are two different two's complement numbers who are their own inverse. Specify these numbers.

(e) Which numbers in three's complement are their own inverse?

(f) What arithmetic operation is a shift left logical equivalent to with three's complement numbers?

(g) What arithmetic operation is a shift right arithmetic equivalent to with two's complement numbers?

Question 1: *Number Representation*

1) Convert the following 8-bit two's complement numbers from hexadecimal to decimal:

0x80 = _____
0xF4 = _____
0x0E = _____

2) What's the biggest change to the PC as the result of a jump on a 32-bit MIPS system?

3) Assume that the most significant bit (MSB) of x is a 0. We store the result of flipping x's bits into y. Interpreted in the following number representations, how large is the magnitude of y relative to the magnitude of x? Circle ONE choice per row.

Unsigned	|y| < |x|	|y| = |x|	|y| >|x|	Can't Tell
One's Complement	|y| < |x|	|y| = |x|	|y| >|x|	Can't Tell
Two's Complement	|y| < |x|	|y| = |x|	|y| >|x|	Can't Tell
Sign and Magnitude	|y| < |x|	|y| = |x|	|y| >|x|	Can't Tell

Question 2: *C strings/pointers (Fa03, Q2)*

a. (2pts) Given the following declarations:

```
char a[14] = "pointers in c";
char c = 'b';
char *p1 = &c, **p2 = &p1;
```

Cross out any of the following statements that are not correct C:

```
p1 = a + 5;

&p1 = &a[0];

p2 = a;

*(a + 10) = 't';

*p2 = %c;
```

b. (3pts) Consider the following C program.

```
#include <stdio.h>

char* set(char c, int i) {
    /* See below for line to insert here */
    str[i] = c; Return
    str;
}

int main(){
    char* output;

    output = set('o', 2); output
    = set('w', 0); output =
    set('r', 1);

    printf("%s", output);

    return 0;
}
```

For each of the following lines inserted as indicated into procedure **set**, what is printed when the program executes? (If the program causes an error during compilation, say "compilation error"; if it causes an error or undefined results while running, say "runtime error.")

a1) static char str[] = "thing";

a2) char str[] = "thing";

a3) char *str = malloc(6);
strcpy(str, "thing")

Question 3: *Memory management in C (sp15, dis2, q1)*

1. In which memory sections (**C**ODE, **ST**ATIC, **H**EAP, **S**TACK) do the following reside?

```
#define C 2
const int val = 16;
char arr[] = "foo";
void foo(int arg){
    char *str = (char *) malloc (C*val);
    char *ptr = arr;
}
```

arg [ ] str [ ]

arr [ ] *str [ ]

val [ ] C [ ]

2. What is wrong with the C code below?

```
int* ptr = malloc(4 * sizeof(int));
if(extra_large) ptr = malloc(10 * sizeof(int));
return ptr;
```

3. Write code to prepend (add to the start) to a linked list, and to free/empty the entire list.
`struct ll_node { struct ll_node* next; int value; }`

free_ll(struct ll_node** list)	prepend(struct ll_node** list, int value)

Note: `list` *points to the first element of the list, or to* `NULL` *if the list is empty.*

Question 4: *Memory management in C (Fa06, M2)*

A bignum is a data structure designed to represent large integers. It does so by abstractly considering all of the bits in the `num` array as part of one very large integer. This code is run on a standard 32-bit MIPS machine, where a `word` (defined below) is 32 bits wide and a `halfword` is 16 bits wide.

```
typedef unsigned int   word;
typedef unsigned short halfword;
typedef struct bignum_struct {
   int length;     // number of words
   word *num;      // the actual data
} bignum;
```

This function shows how bignums are used:

```
void print_bignum(bignum *b) {
   printf("0x"); // Print hex prefix
   for (int i = b->length-1; i>=0; i--)
      printf("%08x", b->num[i]);
}
```

a) Is the ordering of `words` in the `num` array BIG or LITTLE endian? (circle one)

b) How many bytes would be used in the *static*, *stack* and *heap* areas as the result of lines `1`, `3` and `4` below? **Treat each line independently!** E.g., For line `3`, don't count the space allocated in line `1`.

```
1 bignum biggie;
2 int main(int argc, char *argv[]) {
3   bignum bigTriple[3], *bigArray[4];
4   bigArray[1] = (bignum *) malloc (sizeof(bignum) * 2);
```

	static	*stack*	*heap*
Line 1			
Line 3			
Line 4			

b) Complete the `add` function for two bignums, which you may assume **are the same** `length`. Our C compiler translates `z = x + y` (where `x,y,z` are `words`) to `add` (not `addu`, as is customary) and thus could generate a hardware (HW) overflow we don't want, as we're running on untrusted HW. Your code should be written so that `words` never overflow in HW (so we do all adding in the `halfword`).

```
void add(bignum *a, bignum *b, bignum *sum, word carry_in, word *carry_out) {
   // reserve space for num array. Remember a and b are the SAME length...
   sum->num =
   for (int i=0; i < a->length; i++) {  // word-by-word do addition of lo, hi halfwords
      // add lo halfwords of a,b
      word lo     =
      // add hi halfwords of a,b (but in the safe, low halfword area so no HW overflow)
      word hi     =
      // combine low and hi halfwords (put back in their places), like a lui-ori
      sum->num[i] = (hi << 16) | (halfword) lo;
      // what's the carry_in for the next word?
      carry_in    =
   }
   sum->length = a->length;
   *carry_out  = carry_in;
}
```