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) = $\sum_{i=0}^{n} 3^i \cdot d_i$

unsigned(2102_3) = 65

(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.

"flip" all of the trits (replace 2s with 0s, 0s with 2s), and add 1

(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.

$1 \ldots 1_3$. A number is negative if it has unsigned magnitude greater than $1 \ldots 1_3$

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

0, 0b10…0

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

0

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

Multiplication by a power of three

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

Division by a power by two, followed by a floor

Question 1: *Number Representation*

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

0x0E = 14

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

256 mebibytes

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

• In unsigned, a number with the MSB of 1 is always greater than one with a MSB of 0.
• In one's complement, flipping all of the bits is the negation procedure, so the magnitude will be the same. • In two's complement, y is a negative number. Its magnitude can be found by applying the negation procedure, which is flipping the bits and then adding 1, resulting in a larger magnitude than x.
• In sign and magnitude, the 2nd MSB bit will determine the relative magnitudes of x and y, so you can't tell for certain.

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;
The "a" without a subscript means &a[0], a constant of type pointer-to-char. Adding an integer to a pointer is legal, and returns a pointer-to-char, which matches the type of p1, so the assignment is LEGAL.

&p1 = &a[0];
Any expression starting with & is a constant, not a variable, so this is an attempt to assign a value to a constant, like saying 3 = 4; so it's ILLEGAL.

p2 = a;
Again, "a" means &a[0], a constant of type pointer-to-char. But p2 is of type pointer-to-pointer-to-char, so this is a type mismatch and is ILLEGAL. It would be legal, although weird, with an explicit cast: p2 = (char **)a;

***(a + 10) = 't';**
The "a+10" is a valid expression of type pointer-to-char, like a+5 in the first statement. So *(a+10) is a variable of type char, and we are assigning a char value to it, so this is LEGAL.

***p2 = %c;**
p2 is of type pointer-to-pointer-to-char, so *p2 is a variable of type pointer-to-char. &c is a constant of type pointer-to-char. These match, so this assignment is

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";

This allocates a six-byte character array (including a byte for the null at the end) in global data space. Every call to set() refers to this same array. So each of the three calls changes one character: thing -> thong -> whong -> wrong. So the result that's printed is "wrong".

a2) char str[] = "thing";

This allocates a new six-byte array on the stack for each call, then returns the address of that stack array, but the stack frame containing it is deallocated when set() returns. So what will be printed is whatever the call to printf() puts at that address on the stack! The result is therefore undefined, or a runtime error.

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

This heap-allocates a new six-byte array for each call. Each array has the initial value "thing" and then one character is changed. So the first two calls have essentially no effect; the third call changes thing -> tring, and "tring" is printed

Question 3: *C Memory management*

1. In which memory sections (**C**ODE, S**T**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 [ S ]     str  [ S ]
arr [ T ]     *str [ H ]
val [ C ]     C    [ C ]
```

2. What is wrong with the C code below?

```
int* ptr = malloc(4 * sizeof(int));
if(extra_large) ptr = malloc(10 * sizeof(int)); // Memory leak if extra_large is true
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)`

`if(*list) {`
`  free_ll(&((*list)->next));`
`  free(*list);`
`}`
 	
`*list = NULL;`
	`struct ll_node* item = (struct ll_node*)`
	`  malloc(sizeof(struct ll_node));`
	`item->value = value;`
	`item->next = *list;`
	`*list = item;`

Note: `*list` *points to the first element of the list, or is* `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 belonging to 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 `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	8	0	0
Line 3	0	3*8 + 4*4 = 40	0
Line 4	0	0	2*8 = 16

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 =  (word *) malloc (a->length * sizeof(word))

   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     = (a->num[i]&0xffff) + (b->num[i]&0xffff) + carry_in;

      // add hi halfwords of a,b (but in the safe, low halfword area so no HW overflow)

      word hi     = (a->num[i] >> 16 ) + (b->num[i] >> 16 ) + (lo >> 16);

      // 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    = hi >> 16;
   }
   sum->length = a->length;
   *carry_out  = carry_in;
}
```