Question 1 : *What's that funky smell?! Oh yeah, it's potpourri...*

We need to think big, 64 bits big. In the next generation MIPS, called MIPS64, we propose to have a 64-bit address space (so naturally, instructions will be 64 bits wide too), and 64 64-bit-wide registers.

a. How many bytes are stored in all the registers taken together? (IEC format) ____________________

b. Let's say we wanted to keep a 6-bit opcode field (for backward compatibility), and for R-type operators, we use however many bits we need to specify the three registers and shamt, and the remaining bits we give to the function field. How many different MIPS64 (R + I + J) operations can we have? (You don't need a calculator for this question, simply leave your answer as an expression, then say what it approximates in IEC format, something like "2^3 + 2^10 ≈ 1 Kibi instructions") Just for clarification, **`addu $t0 $t1 $t2`** and **`addu $s4 $t9 $0`** are considered the same instruction, **`addu`**.

   __________ + ________________ ≈ __________________ instructions

c. What would the original MIPS language designers say about the opportunity to have many more operations in their language -- would they welcome them or shun them and why?

Suppose we use **`1`** sign bit, **`E`** exponent bits, and **`S`** significand bits to represent floating point numbers. If we give a significand bit to the exponent; we'd now have **`1`** sign bit, **`E+1`** exponent bits, and **`S-1`** significand bits.

d. How would the # of real numbers we could represent change and why? (e.g., shrink by 12 because...)

e. Would the number of underflow cases increase, decrease, or remain the same? Explain briefly.

We wish to find out roughly how much free memory we have that we could **`malloc`**. Sure, we could iterate by **`malloc`**ing and **`free`**ing until we found the exact number, but there's a better way to get a good estimate.

f. Fill in the blank to complete the subroutine **`FreeSpace()`**

```
float PI = 3.14;
int main(int argc, char *argv[]) {
  int i;
  printf("You can probably malloc around %u bytes\n", FreeSpace());
}

unsigned int FreeSpace() {
  int temp;      // In case you need a local variable for some reason

  return ( ________________________________________________________ );
}
```

g. Given the clock frequency and cycles per instruction (Ideal CPI) for two computers, can we predict which one would run a particular C program faster on the same input? Why (tell us how to do it) or why not (tell us all that is missing to be able to make a good prediction).

h. True or False. Although the linker resolves all PC-relative addresses, it is unable to resolve absolute addresses as it doesn't know exactly where things will be in the address space when we actually run the program. Explain briefly but completely.

i. What is the meaning of following nibble **N** if it is interpreted as a...

	N = 0b1101
Sign/Magnitude number (write your answer in decimal)	
Twos Complement number (write your answer in decimal)	
Unsigned number (write your answer in decimal)	
Float with SEEM format (1 Sign (S) bit, 2 Exponent (E) bits, 1 Significant (M) bit, bias = 1)	
MIPS instruction if **N** is the lower bits of the opcode (bits 29:26) with the rest zeros	
What is the value of this expression written in hex? → **((N & 0x6) | 0x9) >> 1**	

j. The following MIPS instructions listed in the table below are located at memory location **0x70000008**. Assume **$t0** contains **0xDEADBEEF**. What is the address of the next instruction that will be executed in each case?

Question	**Machine Code**	**Next Instruction's Address**
(1)	`000010 11111 11100 00000 00000 000011`	
(2)	`000000 01000 00000 00000 00000 001000`	
(3)	`000101 01000 00000 11111 11111 111110`	
(4)	`000000 00000 01000 01000 00000 100011`	

Question 2 : *How can I bring you to the C of Madness?*

We wish to implement a *nibble* (4-bit) array, where we can read and write a particular *nibble.* Normally for read/write array access, we would just use bracket notation (e.g., **`x=A[5]; A[5]=y;`**), but since a nibble is *smaller* than the smallest datatype in C, we have to design our own **`GetNibble()`** and **`NewNibbleArray()`** functions. We'll use the following typedefs to make our job easier:

```
typedef uint8_t nibble_t;  // If it's a single nibble, value is in least signif. nibble.
typedef uint32_t index_t;  // The index into a nibble_t array to select which one is used
```

E.g., imagine a nibble array with 4 nibbles (2 bytes): **`nibble_t A[2]; A[1]=0x8B;`** **`A[0]=0x3F;`** Internally, **`A`** would look like the table below. **`GetNibble(A,0)`** would return **`0xF`**, **`GetNibble(A,1)`** would return **`0x3`**, **`GetNibble(A,2)`** would return **`0xB`**, and **`GetNibble(A,3)`** would return **`0x8`**.

Nibble index	3	2	1	0
Array A	8	B	3	F

a. Write **`GetNibble`** in C in one line. Three points deducted if you use **`if/else`** or **`?/:`** format.

```
GetNibble(nibble_t A[], index_t i) {

    return ( ________________________________________________
);
}
```

b. We authored **`NewNibbleArray`** in C that takes in the number of nibbles the user wants to store and returns a pointer to the contiguous region of memory that can hold it. We didn't like the MIPS code that the compiler spit out so we hand-edited the MIPS **`.s`** file and put this in its place. When we test this from **`main`** it works fine. Answer three quick questions below (1) what's the bug, (2) why did it work from **`main`**? (3) how can we fix it? (in English, not MIPS)

```
nibble_t *NewNibbleArray(index_t number_of_nibbles) {
   ### This is the MIPS that our C code was translated into
   NewNibbleArray:   addi $a0 $a0 1   ### Even 1 nibble needs a whole
byte
                     srl $a0 $a0 1    ### num bytes =
⌈number_of_nibbles/2⌉
                     sub $sp $sp $a0  ### Make room for nibble array
                     add $v0 $0 $sp   ### Return ptr to the array
                     jr $ra
}
|<--- answer 1 here --->|<--- answer 2 here --->|<--- answer 3 here
--->|
```

Question 3 : *Let's take a look under the hood, eh?...* `cs61c-__`

We wish to write **`sum`** in MAL MIPS, which sums up all the elements in a linked list, whose node we define below. We've coded up a recursive **`sum`** routine in C.

```
struct Node {
  int n;              // If there's a ptr to this struct, n is in the first word
  struct Node *next; // and next is 4 bytes away. So you can think of this
};                    // stored in memory like an array. A[0] is n, A[1] is next.

int sum (struct Node *head) {
  if (head == NULL)
     return 0;
  else
     return head->n + sum(head->next);
}
```

Now, fill in the blanks to author the *recursive* MIPS that implements **`sum`** for us. Wherever we've written a comment, you *must* fill in the equivalent MIPS. Where there's no comment, you can do what you want. Our solution uses every line -- if you need to use more or fewer, you should think twice about it.

```
sum:  ___________________________

      ___________________________  # if head == NULL goto done

      ___________________________  # make room for data we'll need later

      ___________________________

      ___________________________

      ___________________________

      ___________________________  # recursive call

      ___________________________

      ___________________________

      ___________________________  # give the room back

      ___________________________  # head->n

      ___________________________  # head->n + sum(head->next)

done: ___________________________  # return to caller
```

Question 4 : *Cache, money, dollar bill y'all...*

Take a look at the following C function `sum_iter` run on a 32-bit MIPS machine. On this system, these `struct`s are aligned to two-word boundaries since `sizeof(struct Node) = 8`. Assume the total space taken up by the linked list is greater than (and a *multiple* of) the cache size.

```
struct Node {                         // copied from Q3 for your convenience
  int n;
  struct Node *next;
};

int sum_iter (struct Node *head) {  // iterative version of sum from Q3
    int sum = 0;
    while (head != NULL) {
        sum += head->n;               // load from head+0
        head = head->next;            // load from head+4
    }
    return sum;
}
```

Given a direct-mapped data cache with this configuration: **INDEX: 13 bits, OFFSET: 7 bits**

a. How many *words* are in a block? ____________________________

b. How many *bytes of data* does this cache hold? (in IEC format) ________________________

Let's define A and B as your answers to (a) and (b) above, respectively. For questions (c) and (d) below, use these variables in your answer if necessary (this way if you get A and/or B wrong, you could still get full points!) Also, when we mention hit rate below, we're talking about accessing *data* (not instructions).

c. What is the *lowest possible cache hit rate* for the `while` loop in `sum_iter`? ____________________

d. What is the *highest possible cache hit rate* for the `while` loop in `sum_iter`? ____________________

e. To achieve this maximum hit rate, we obviously could have every Node next to every other node, like an array. However, that's too strict a constraint -- we can *still* achieve this hit rate if that's not the case. What is the *loosest* constraint for how the Nodes are distributed in memory to get the best hit rate?