1. The following problem concerns basic cache lookups and layouts.

   In all the problems, the memory is byte adressable, and memory accesses are all to bytes (not words).

   (a) **[5 Points]** If the cache is 4-way set associative, with an 8-byte block size and 512 total bytes, label the parts of the address uses as the *block offset* (BO) within the line, the *cache set index* (CI) and the *cache tag* (CT).

11	10	9	8	7	6	5	4	3	2	1	0

   (b) **[5 Points]** If the cache is 2-way set associative, with a 4-byte block size and 256 total bytes, label the parts of the address uses as the *block offset* (BO) within the line, the *cache set index* (CI) and the *cache tag* (CT).

11	10	9	8	7	6	5	4	3	2	1	0

   (c) **[5 Points]** Assume you're designing a cache for a computer that will mainly be used to run workloads with strong spatial locality. For a cache of a given size, what attribute of the cache is likely to be the most important?

2. The following program implements *matrix transposition*.

```
float m[4][4] = { {2,   1,-1,  8}, {-3, -1, 2, -11},
  {-2,  1, 2, -3}, {10, -2, 4, 8} };

for (int i = 0; i < rows; i++) {
  for (int j = i+1; j < cols; j++) {
    float t = m[i][j];
    m[i][j] = m[j][i];
    m[j][i] = t;
  }
}
```

You should assume:

- Floats take 4 bytes.
- Stores to memory do not cause cache misses if the address memory is not already in the cache.
- The array 'm' starts at address 0; memory addresses at 12 bits long.
- The variables 'i', 'j', 'k' and 't' are held in registers
- Your cache is 2-way set associate with 8 byte lines, and a total size of 32 bytes

Below, list the addres the first 12 references and indicate if it it a hit or miss in the cache. Use decimal numbers throughout.

[12 Points]

Ref #	Address	Hit?
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

[8 Points] Below, draw a diagram to show the state of the cache at the end of the references above. You should clearly indicate each set and the value of each line in each set. For each cache line, you should indicate if the entry is valid and the appropriate tag for that line (if valid). To simplify problem, use the starting address for the line rather than the tag.

3. The next problem concerns the following C code. This program reads a string on standard input and prints an integer in hexadecimal format based on the input string it read.

```
#include <stdio.h>

/* Read a string from stdin into buf */
int evil_read_string()
{
    int buf[2];

    scanf("%s",buf);
    return buf[1];
}

int main()
{
    printf("0x%x\n", evil_read_string());
}
```

Here is the corresponding machine code on a Linux/x86 machine:

```
08048414 <evil_read_string>:
 8048414:  55                  push   %ebp
 8048415:  89 e5               mov    %esp,%ebp
 8048417:  83 ec 14            sub    $0x14,%esp
 804841a:  53                  push   %ebx
 804841b:  83 c4 f8            add    $0xfffffff8,%esp
 804841e:  8d 5d f8            lea    0xfffffff8(%ebp),%ebx
 8048421:  53                  push   %ebx                address arg for scanf
 8048422:  68 b8 84 04 08      push   $0x80484b8          format string for scanf
 8048427:  e8 e0 fe ff ff      call   804830c <_init+0x50>  call scanf
 804842c:  8b 43 04            mov    0x4(%ebx),%eax
 804842f:  8b 5d e8            mov    0xffffffe8(%ebp),%ebx
 8048432:  89 ec               mov    %ebp,%esp
 8048434:  5d                  pop    %ebp
 8048435:  c3                  ret

08048438 <main>:
 8048438:  55                  push   %ebp
 8048439:  89 e5               mov    %esp,%ebp
 804843b:  83 ec 08            sub    $0x8,%esp
 804843e:  83 c4 f8            add    $0xfffffff8,%esp
 8048441:  e8 ce ff ff ff      call   8048414 <evil_read_string>
 8048446:  50                  push   %eax                integer arg for printf
 8048447:  68 bb 84 04 08      push   $0x80484bb          format string for printf
 804844c:  e8 eb fe ff ff      call   804833c <_init+0x80>  call printf
 8048451:  89 ec               mov    %ebp,%esp
 8048453:  5d                  pop    %ebp
 8048454:  c3                  ret
```

This problem tests your understanding of the stack discipline and byte ordering. Here are some notes to help you work the problem:

- `scanf("%s", buf)` reads an input string from the standard input stream (stdin) and stores it at address `buf` (including the terminating `'\0'` character). It does **not** check the size of the destination buffer.
- `printf("0x%x", i)` prints the integer i in hexadecimal format preceded by "`0x`".
- Recall that Linux/x86 machines are Little Endian.
- You will need to know the hex values of the following characters:

Character	Hex value	Character	Hex value
`'d'`	`0x64`	`'v'`	`0x76`
`'r'`	`0x72`	`'i'`	`0x69`
`'.'`	`0x2e`	`'l'`	`0x6c`
`'e'`	`0x65`	`'\0'`	`0x00`
		`'s'`	`0x73`

(a) **[5 Points]** Suppose we run this program on a Linux/x86 machine, and give it the string "`dr.evil`" as input on stdin.

Here is a template for the stack, showing the locations of `buf[0]` and `buf[1]`. Fill in the value of `buf[1]` (in hexadecimal) and indicate where `ebp` points just **after** `scanf` returns to `evil_read_string`.

```
                |<- buf[0]->|<-buf[1] ->|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

(b) **[5 Points]** What is the 4-byte integer (in hex) printed by the `printf` inside main?

`0x`________________________

(c) Suppose now we give it the input "`dr.evil.lives`" (again on a Linux/x86 machine).

i. **[5 Points]** List the contents of the following memory locations just **after** `scanf` returns to `evil_read_string`. Each answer should be an unsigned 4-byte integer expressed as 8 hex digits.

`buf[0] = 0x`________________________

`buf[3] = 0x`________________________

ii. **[5 Points]** Immediately **before** the `ret` instruction at address `0x08048435` executes, what is the value of the frame pointer register `%ebp`?

`%ebp  = 0x`________________________

You can use the following template of the stack as *scratch space*. *Note:* this does **not** have to be filled out to receive full credit.

```
               <- buf[0] -><- buf[1] ->
--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
```

4. The following problem concerns optimizing a procedure for maximum performance on an Intel Pentium III. Recall the following performance characteristics of the functional units for this machine:

Operation	Latency	Issue Time
Integer Add	1	1
Integer Multiply	4	1
Integer Divide	36	36
Floating Point Add	3	1
Floating Point Multiply	5	2
Floating Point Divide	38	38
Load or Store (Cache Hit)	1	1

Consider the following two procedures:

Loop 1

```
int loop1(int *a, int x, int n)
{
  int y = x*x;
  int i;
  for (i = 0; i < n; i++)
    x = y * a[i];
  return x*y;
}
```

Loop 2

```
int loop2(int *a, int x, int n)
{
  int y = x*x;
  int i;
  for (i = 0; i < n; i++)
    x = x * a[i];
  return x*y;
}
```

When compiled with GCC, we obtain the following assembly code for the inner loop:

Loop 1

```
.L21:
   movl %ecx,%eax
   imull (%esi,%edx,4),%eax
   incl %edx
   cmpl %ebx,%edx
   jl .L21
```

Loop 2

```
.L27:
   imull (%esi,%edx,4),%eax
   incl %edx
   cmpl %ebx,%edx
   jl .L27
```

Running on a vintage piece of hardware, we find that Loop 1 requires 3.0 clock cycles per iteration, while Loop 2 requires 4.0.

(a) **[10 Points]** Explain how it is that Loop 1 is faster than Loop 2, even though it has one more instruction

(b) **[5 Points]** By using the compiler flag `-funroll-loops`, we can compile the code to use 4-way loop unrolling. This speeds up Loop 1. Explain why.

(c) **[5 Points]** Even with loop unrolling, we find the performance of Loop 2 remains the same. Explain why.