1.
What is loop unrolling? How can it make your program more efficient? How can it make your program less efficient?

2.
What affects the data stored on the stack? Think registers and instructions!

3.
The following table gives the parameters for a different number of caches. For each cache, fill in the missing fields in the table.

- *m* is the number of physical address bits
- *C* is the cache size
- *B* is the block size
- *E* is the associativity
- *S* is the number of the cache sets
- *t* is the number of tag bits
- *s* is the number of set index bits
- *b* is the number of block offset bits

Cache	*m*	*C*	*B*	*E*	*S*	*t*	*s*	*b*
1	32	1024	4	4				
2	32	2048			128	23	7	
3	32	1024	8	1				
4	32	1024	8	128				
5	32	1024				25	4	3
6	32	1024	32	4				

4.
The provided function `func_one` takes as input two pointers, that are actually each individually pointing to the first element in a N by M array of integers.

```
int func_one(char* one, char* two, int N, int M) {
      int i, j, k;
      int sum = 0;

      char* ptr1 = one;
      char* ptr2 = two;

      for (k = 0; k < 4; k++) {
         for (j = 0; j < M; j++) {
            for (i = 0; i < N; i++) {

               char one = *(ptr1 + k + j*4 + i*4*M);
               int masked = one & 0xFF;

               int shift = k << 3;
               int shifted = masked << shift;

               *(ptr2 + k + j*4 + i*4*M) = masked;
               sum += shifted;
            }
         }
      }

      return sum;
}
```

In what ways can we optimize the above function?

5.

The provided code below is an optimization of the previous problem. Fill in the blanks.

```
int func_two(char* one, char* two, int N, int M) {
      int i, j, k;
      int sum = 0;
      int temp = 0;

      char* ptr1 = one;
      char* ptr2 = two;

      for (i = 0; i < N; i++) {
         for (j = 0; j < M; j++) {
            temp = (0xFF & *ptr1);
            sum += ______________;
            *ptr2 = _____________;
            ptr1++;
            ptr2++;

            temp = ______________;
            sum += ______________;
            *ptr2 = _____________;
            ptr1++;
            ptr2++;

            _____________________;
            _____________________;
            _____________________;
            _____________________;
            _____________________;

            _____________________;
            _____________________;
            _____________________;
            _____________________;
            _____________________;
         }
      }

      return sum;
}
```

FYI.

jjm3105 — ssh myong@lnxsrv09.seas.ucla.edu — 80×24

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 71.85     23.48    23.48        1    23.48    23.48  func_one
 16.20     28.78     5.29        1     5.29     5.29  func_one_opt
  6.46     30.89     2.11                             main
  5.96     32.83     1.95        1     1.95     1.95  func_two

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

"gprof.output" 157L, 6493C                                    2,0-1         Top
```

The above shows the running time of the functions discussed in problems 5 and 6. `func_one` is the code from problem 5 as is, `func_one_opt` is one optimization of `func_one`, and `func_two` is the completed code from problem 6.

The results were generated using gprof

OPTIONAL
6.
Assume the following:

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not to 4-byte words).
- Addresses are 13 bits wide.
- The cache is two-way set associative (E = 2), with a 4-byte block size (B = 4) and eight sets (S = 8).

The contents of the cache are as follows, with all numbers given in hexadecimal notation.

	Line 0						Line 1					
Set Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	09	1	86	30	3F	10	00	0				
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0					0B	0				
3	06	0					32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0				
6	91	1	A0	B7	26	2D	F0	0				
7	46	0					DE	1	12	C0	88	37

Suppose a program running on a machine with such a cache references the 1 byte word at the address 0x0E34.

What is the resulting of the following?

Cache block offset:
Cache set index:
Cache tag:
Cache hit?:
Byte returned: