Full Name: _____

Andrew ID: _____

CS 15-213, Fall 1998

Final Exam

December 11, 1998

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew ID on the front.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 100 points.
- The problems are of varying difficulty. The point value indicated for each problem is proportional to its difficulty. Pile up the easy points quickly and then come back to the harder problems.
- Each problem tests a skill from a lab or homework assignment. The problems are listed in course order.
- This exam is OPEN BOOK. You may use any books or notes you like. Good luck!

1:	/05
2:	/10
3:	/18
4:	/10
5:	/14
6:	/10
7:	/16
8:	/17
TOTAL:	/100

Problem 1. (5 points):

Write a bit mask generator. Assume that $0 \leq low \leq high \leq 31$. Examples: `make_mask(12,9)` should return `0x000000E0`, while `make_mask(31,31)` should return `0x80000000`.

```
int make_mask(unsigned int high, unsigned int low)
{
    int mask;









    return mask;
}
```

Problem 2. (10 points):

Consider the following Alpha assembly code for a procedure `foo()`:

```
       foo:
  0x0:  47ff0413        bis     zero, zero, a3
  0x4:  ee20000c        ble     a1, 0x38
  0x8:  a4300000        ldq     t0, 0(a0)
  0xc:  424105a2        cmpeq   a2, t0, t1
  0x10: f4400009        bne     t1, 0x38
  0x14: 42700640        s8addq  a3, a0, v0
  0x18: b7e00000        stq     zero, 0(v0)
  0x1c: 42603413        addq    a3, 0x1, a3
  0x20: 40011400        addq    v0, 0x8, v0
  0x24: 427109a3        cmplt   a3, a1, t2
  0x28: e4600003        beq     t2, 0x38
  0x2c: a4800000        ldq     t3, 0(v0)
  0x30: 424405a5        cmpeq   a2, t3, t4
  0x34: e4bffff8        beq     t4, 0x18
  0x38: 46730400        bis     a3, a3, v0
  0x3c: 6bfa8001        ret     zero, (ra), 1
```

Based on the assembly code above, fill in the blanks below in its corresponding C source code. (Note: you may only use symbolic variables a, len, val, and i, from the source code in your expressions below—do *not* use register names.)

```
long int foo(long int *a, long int len, long int val)
{
  long int i;

  for (i = _________; _____________________________ ; i =___________) {
    a[i] = 0;
  }
  return i;
}
```

Problem 3. (18 points):

Dr. Evil has returned! He has placed a binary bomb in this exam! Once again, Dr. Evil has made the disasterous mistake of leaving behind some of his source code. Can you save all of mankind (or at least your grade on this question), and tell us what this bomb does? Turn the page to find out how...

The C source code Dr. Evil forgot to erase:

```
/* bomb.c:  Use new computer technology to blow up exams! -- Dr. Evil */
#include <stdio.h>
#include <stdlib.h>

extern long secret_unsolvable_puzzle_fn(long input);

void explode_bomb()
{
    printf("You fail!  Mwhahahahaha!!!\n");
    exit(8);
}

int main(int argc,
         char *argv[])
{
    if(argc != 2) {
        printf("Usage: %s <magic password>\n", argv[0]);
        explode_bomb();
    }

    if(secret_unsolvable_puzzle_fn(atol(argv[1])) != 1) {
        explode_bomb();
    }

    printf("Curses, foiled again!  Good work!\n");

    return 0;
}
```

The Alpha dissasembly for the stuff Dr. Evil did erase:

```
        secret_unsolvable_puzzle_fn:
  0x120001320:  addq    a0, a0, t0
  0x120001324:  lda     t1, 15213(zero)
  0x120001328:  cmpult  t0, t1, t2
  0x12000132c:  beq     t2, 0x12000133c
  0x120001330:  addq    t0, a0, t0
  0x120001334:  cmpult  t0, t1, t3
  0x120001338:  bne     t3, 0x120001330
  0x12000133c:  lda     v0, -15213(t0)
  0x120001340:  cmpeq   zero, v0, v0
  0x120001344:  ret     zero, (ra), 1
```

(a) Does the bomb use floating point arithmetic? (Yes/No)

(b) Does the function `secret_unsolvable_puzzle_fn()` contain any recursion? (Yes/No)

(c) You think that `secret_unsolvable_puzzle_fn()` is too slow, and you want to replace it by a lookup table which exhaustively lists all possible inputs and maps them to all possible outputs. How many entries will your lookup table have? What values can `secret_unsolvable_puzzle_fn()` return?

entries in table =

possible return values =

(d) Just to see what happens, you try running the program "`./bomb 0`". What happens? Does the bomb explode?

(e) For each of the following input values, circle whether it defuses or explodes the bomb:

input = 2 defuses explodes

input = 11 defuses explodes

input = 1381 defuses explodes

input = 5071 defuses explodes

Problem 4. (10 points):

You have been assigned the task of writing a C function to compute a floating point representation of 2^x. You realize that the best way to do this is to directly construct the IEEE single precision representation of the result. When x is too small, your routine will return 0.0. When x is too large, it will return $+\infty$. Fill in the blank portions of the following code to compute the correct result:

```
/* Compute 2**x */
float fpwr2(int x) {

  union {
    unsigned u;
    float f;
  } result;

  unsigned exp, sig;  /* Result exponent and significand */

  if (x < ______) {
    /* Too small.  Return 0.0 */
    exp = ____________;
    sig = ____________;
  } else if (x < ______) {
    /* Denormalized result */
    exp = ____________;
    sig = ____________;
  } else if (x < ______) {
    /* Normalized result. */
    exp = ____________;
    sig = ____________;
  } else {
    /* Too big.  Return +oo */
    exp = ____________;
    sig = ____________;
  }

  result.u = exp << 23 | sig;
  return result.f;
}
```

Problem 5. (14 points):

Welcome to the C-Memory-Layout Question. The context for all questions is the Alpha, as usual.

```
struct s1 {
    char     c;
    short    s;
    double   d;
    int      i[4];
};

struct s2 {
    int       a[2][3];
    struct s1 b;
    long      c[2][2];
};

struct s2 thing;
```

a) Fill in the following table to describe the memory footprint of `thing`. Do not include wasted space in $length$. To help you, we've filled in two of the blanks (well, ok - only one is useful).

Field	Start	Length
a	0	
b.c		
b.s		
b.d		
b.i		
c	56	

b) In an effort to create job security for himself, a deranged programmer at your company had "optimized" his code by referencing fields of `thing` via out-of-range elements of `thing.a` and `thing.c`. Needless to say, as soon as this was discovered, he was fired. Now, you must figure out what the programmer was actually referencing.

First, figure out the *offset* of the access from the beginning of the array accessed. For example, the offset of `thing.b.i[1]` is 4. Using the offset and your knowledge of the structure's layout, you can figure out how the programmer *should* have referred to those memory locations. Write one **or more** expressions indicating what fields were being referenced. Array indices you give should be within the bounds as given in the declaration. It may help you to draw a picture on scratch paper.

Store-to	Relative Offset	Fields/Elements Overwritten
`thing.a[3][2]`		
`thing.c[-1][-3]`		
`thing.b.i[6]`		

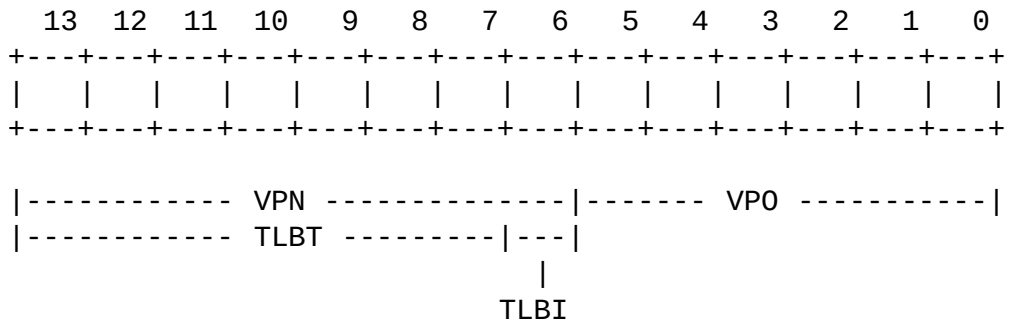
Translation Lookaside Buffers

Translation Lookaside Buffers (TLBs) are often managed by the operating system virtual memory manager using special instructions to update TLB entries. If there are bugs in this software, chaotic behavior can occur. Consider the task of writing a "consistency checker" that would scan through the TLB and detect anomalous entries.

Assume a virtually-addressed memory system with the following properties:

- Virtual addresses are 14 bits wide.
- Physical addresses are 12 bits wide.
- The page size is 64 bytes.
- The TLB is 8-way set associative with 16 total entries.
- There should be no aliasing in the address mapping, i.e., under no condition should two virtual addresses map to the same physical address.

Observe that for this virtual address format, the upper 8 bits denote the virtual page number (VPN), while the lower 6 bits denote the virtual page offset (VPO). For indexing into the TLB, the low order bit of the VPN indexes the set (TLBI), and the upper 7 bits form the tag (TLBT), as diagrammed below:



The physical address is split into two fields: the upper 6 bits denote the physical page number (PPN), while the lower 6 bits denote the physical page offset (PPO).

The page table (in hexadecimal) for the first 16 pages is as follows:

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	2B	0	09	17	1
02	33	1	0A	09	0
03	02	1	0B	1A	0
04	2A	0	0C	27	0
05	16	1	0D	2D	1
06	04	0	0E	11	1
07	26	0	0F	0D	1

Problem 6. (10 points):

In this problem, you are given a series of TLB entries *for set index 1*. For each entry, indicate the virtual page number (VPN) represented (in hexadecimal), whether the entry is valid (i.e, there is nothing wrong with its format or with respect to the portion of the page table given), and a brief explanation. For valid entries, your explanation should describe what the combination of TLB and page table indicates about this page. For invalid entries, you should describe why they are invalid.

A.

Tag	PPN	Valid
03	31	1

VPN:
OK?:
Explanation:

B.

Tag	PPN	Valid
06	2D	1

VPN:
OK?:
Explanation:

C.

Tag	PPN	Valid
04	28	0

VPN:
OK?:
Explanation:

D.

Tag	PPN	Valid
1F	33	1

VPN:
OK?:
Explanation:

E.

Tag	PPN	Valid
F0	40	1

VPN:
OK?:
Explanation:

Caches

In the next problem you will determine the number of read and write misses for a small transpose routine, given data caches of sizes 64 and 256 bytes. Throughout this problem you should assume the following:

- Like Lab L3, the data cache is direct mapped, write through, write allocate, and the block size is 32 bytes.
- The *src* array starts at address 0 and the *dst* array starts at address 128.
- Accesses to the *src* and *dst* arrays are the only sources of read and write misses, respectively.
- Unlike Lab L3, the arrays are 2d arrays of longs, where `sizeof(long)` = 8 bytes.

Problem 7. (16 points):

```
typedef long array[4][4];

void foo(array dst, array src) {
  int i, j;

  for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
      dst[j][i] = src[i][j];
    }
  }
}
```

A. For a cache with a *total size of 64 data bytes* and for each i and j, indicate whether each access to $dst[i][j]$ and $src[i][j]$ is a hit (h) or a miss (m). For example, reading $src[0][0]$ is a miss and writing $dst[0][0]$ is also a miss.

dst array				
	0	1	2	3
0	m			
1				
2				
3				

src array				
	0	1	2	3
0	m			
1				
2				
3				

B. Repeat part A for a cache with a *total size of 256 data bytes*.

dst array				
	0	1	2	3
0				
1				
2				
3				

src array				
	0	1	2	3
0				
1				
2				
3				

Problem 8. (Of course, 17 points):

The designers of the cookienet protocol just can't stop: after their proprietary protocol was reverse engineered, they designed CookieNet 2000. In this problem, you'll reverse engineer this protocol. They've advertised that they've improved encryption by varying the key on every packet, and increased the number of simultaneous cookie messages possible.

You start by using the manufacturer's program, `pfw2k` to write the following message to the network, specifying port `0x12345678`:

```
the quick brown fox 1
the quick brown fox 2222
the quick brown fox 33333333
```

Here's a dump of the packets that were sent to the network, both in hex and the character representation, without the Ethernet header.

```
packet 1:
00 00 00 00 12 34 56 78 00 16 00 00 74 68 65 20 71 75 69 63 6B 20 62
      72 6F 77 6E 20 66 6F 78 20 31 0A
.....4Vx....the quick brown fox 1.

packet 2:
00 00 00 01 12 34 56 78 00 19 00 00 75 69 66 20 72 76 6A 64 6C 20 63
73 70 78 6F 20 67 70 79 20 32 32 32 32 0A
.....4Vx....uif rvjdl cspxo gpy 2222.

packet 3:
00 00 00 02 12 34 56 78 00 1D 00 00 76 6A 67 20 73 77 6B 65 6D 20 64
74 71 79 70 20 68 71 7A 20 33 33 33 33 33 33 33 33 0A
.....4Vx....vjg swkem dtqyp hqz 33333333.
```

a) Name the fields present in the CookieNet 2000 packet header. Does any field serve more than one purpose?

b) Fill in the C structure below with the format of the packet. If you need a placeholder, clearly label it as such.

```
struct packet {









    unsigned char payload[1]; /* actual payload may be longer */
};
```

c) Write, in hex, the header for a packet that contains `0x12C` (300) characters and has sequence number `0x10000` (65536), on port number `0x19` (25). You may not need all the spaces. Put one byte per box. Indicate where the start of the payload would be.