Andrew ID (print clearly!):_____

Full Name:_____

15-213/18-213, Fall 2011

Exam 1

Tuesday, October 18, 2011

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew ID and full name on the front.
- This exam is closed book, closed notes (except for 1 double-sided note sheet). You may not use any electronic devices.
- 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 66 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Good luck!

1 (08):	
2 (08):	
3 (08):	
4 (04):	
5 (10):	
6 (08):	
7 (10):	
8 (04):	
9 (06):	
TOTAL (66):	

Problem 1. (8 points):

Multiple choice. Write your answer for each question in the following table:

1	2	3	4	5	6	7	8

1. What is the minimum (most negative) value of a 32-bit two's complement integer?

   (a) -2^{32}

   (b) $-2^{32} + 1$

   (c) -2^{31}

   (d) $-2^{31} + 1$

2. What is the difference between the `mov` and `lea` instructions?

   (a) `lea` dereferences an address, while `mov` doesn't.

   (b) `mov` dereferences an address, while `lea` doesn't.

   (c) `lea` can be used to copy a register into another register, while `mov` cannot.

   (d) `mov` can be used to copy a register into another register, while `lea` cannnot.

3. After executing the following code, which of the variables are equal to 0?

```
unsigned int a = 0xffffffff;
unsigned int b = 1;
unsigned int c = a + b;
unsigned long d = a + b;
unsigned long e = (unsigned long)a + b;
```

   (Assume `int`s are 32 bits wide and `long`s are 64 bits wide.)

   (a) None of them

   (b) `c`

   (c) `c` and `d`

   (d) `c`, `d`, and `e`

4. Assume a function `foo` takes two arguments. When calling `foo(arg1, arg2)`, which is the correct order of operations assuming x86 calling conventions and that `foo` must allocate stack space (implies that we must save the `%ebp`)?

   (a) `push arg1, push arg2, call foo, push %ebp`
   (b) `push arg1, push arg2, push %ebp, call foo`
   (c) `push arg2, push arg1, call foo, push %ebp`
   (d) `push arg2, push arg1, push %ebp, call foo`

5. Which one of the following statements is NOT true?

   (a) x86-64 provides a larger virtual address space than x86.
   (b) The stack disciplines for x86 and x86-64 are different.
   (c) x86 uses `%ebp` as the base pointer for the stack frame.
   (d) x86-64 uses `%rbp` as the base pointer for the stack frame.

6. Consider a 4-way set associative cache ($E = 4$). Which one of the following statements it true?

   (a) The cache has 4 blocks per line.
   (b) The cache has 4 sets per line.
   (c) The cache has 4 lines per set.
   (d) The cache has 4 sets per block.
   (e) None of the above.

7. Which one of the following statements about cache memories is true?

   (a) Fully associative caches offer better latency, while direct-mapped caches have lower miss rates.
   (b) Fully associative caches offer lower miss rates, while direct-mapped caches have better latency.
   (c) Direct-mapped caches have both better miss rates and better latency.
   (d) Both generally have similar latency and miss rates.

8. Consider an SRAM-based cache for a DRAM-based main memory. Neglect the possibility of other caches or levels of the memory hierarchy below main memory. If a cache is improved, increasing the typical hit rate from 98% to 99%, which one of the following would best characterize the likely decrease in typical access time?

   (a) 0%
   (b) 1%
   (c) 10%
   (d) 100%

Problem 2. (8 points):

Integer encoding. Fill in the blanks in the table below with the number described in the first column of each row. You can give your answers as unexpanded simple arithmetic expressions (such as $15^{213} + 42$); you should not have trouble fitting your answers into the space provided.

For this problem, assume a **6-bit word size**.

Description	Number
U_{max}	
T_{min}	
`(unsigned) ((int) 4)`	
`(unsigned) ((int) -7)`	
`(((unsigned) 0x21) << 1) & 0x3F)`	
`(int) (20 + 12)`	
`12 && 4`	
`(! 0x15) > 16`	

Problem 3. (8 points):

Floating point encoding. Consider the following 5-bit floating point representation based on the IEEE floating point format. This format does not have a sign bit – it can only represent nonnegative numbers.

- There are $k = 3$ exponent bits. The exponent bias is 3.
- There are $n = 2$ fraction bits.

Recall that numeric values are encoded as a value of the form $V = M \times 2^E$, where E is the exponent after biasing, and M is the significand value. The fraction bits encode the significand value M using either a denormalized (exponent field 0) or a normalized representation (exponent field nonzero). The exponent E is given by $E = 1 - \textit{Bias}$ for denormalized values and $E = e - \textit{Bias}$ for normalized values, where e is the value of the exponent field `exp` interpreted as an unsigned number.

Below, you are given some decimal values, and your task it to encode them in floating point format. In addition, you should give the rounded value of the encoded floating point number. To get credit, you must give these as whole numbers (e.g., 17) or as fractions in reduced form (e.g., $3/4$). Any rounding of the significand is based on ***round-to-even***, which rounds an unrepresentable value that lies halfway between two representable values to the nearest even representable value.

Value	Floating Point Bits	Rounded value
$9/32$	`001 00`	1/4
3		
9		
$3/16$		
$15/2$		

Problem 4. (4 points):

Functions. I never learned to properly comment my code, and now I've forgotten what this function does. Help me out by looking at the assembly code and reconstructing the C code for this recursive function. Fill in the blanks:

```
unsigned mystery1(unsigned n) {
     if(_________)
          return 1;
     else
          return 1 + mystery1(_________);
}
```

```
mystery1:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    cmpl $0, 8(%ebp)
    jne .L2
    movl $1, -4(%ebp)
    jmp .L3
.L2:
    movl 8(%ebp), %eax
    shrl %eax
    movl %eax, (%esp)
    call mystery1
    addl $1, %eax
    movl %eax, -4(%ebp)
.L3:
    movl -4(%ebp), %eax
    leave
    ret
```

Problem 5. (10 points):

Stack discipline. Consider the following C code and assembly code for a recursive function:

```
int gcd(int a, int b)
{
    if(!b)
    {
        return a;
    }

    return gcd(b, a % b);
}
```

```
0x08048394 <+0>:    push   %ebp
0x08048395 <+1>:    mov    %esp,%ebp
0x08048397 <+3>:    sub    $0x10,%esp
0x0804839a <+6>:    mov    0x8(%ebp),%eax
0x0804839d <+9>:    mov    0xc(%ebp),%ecx
0x080483a0 <+12>:   test   %ecx,%ecx
0x080483a2 <+14>:   je     0x80483b7 <gcd+35>
0x080483a4 <+16>:   mov    %eax,%edx
0x080483a6 <+18>:   sar    $0x1f,%edx
0x080483a9 <+21>:   idiv   %ecx
0x080483ab <+23>:   mov    %edx,0x4(%esp)
0x080483af <+27>:   mov    %ecx,(%esp)
0x080483b2 <+30>:   call   0x8048394 <gcd>
0x080483b7 <+35>:   leave
0x080483b8 <+36>:   ret
```

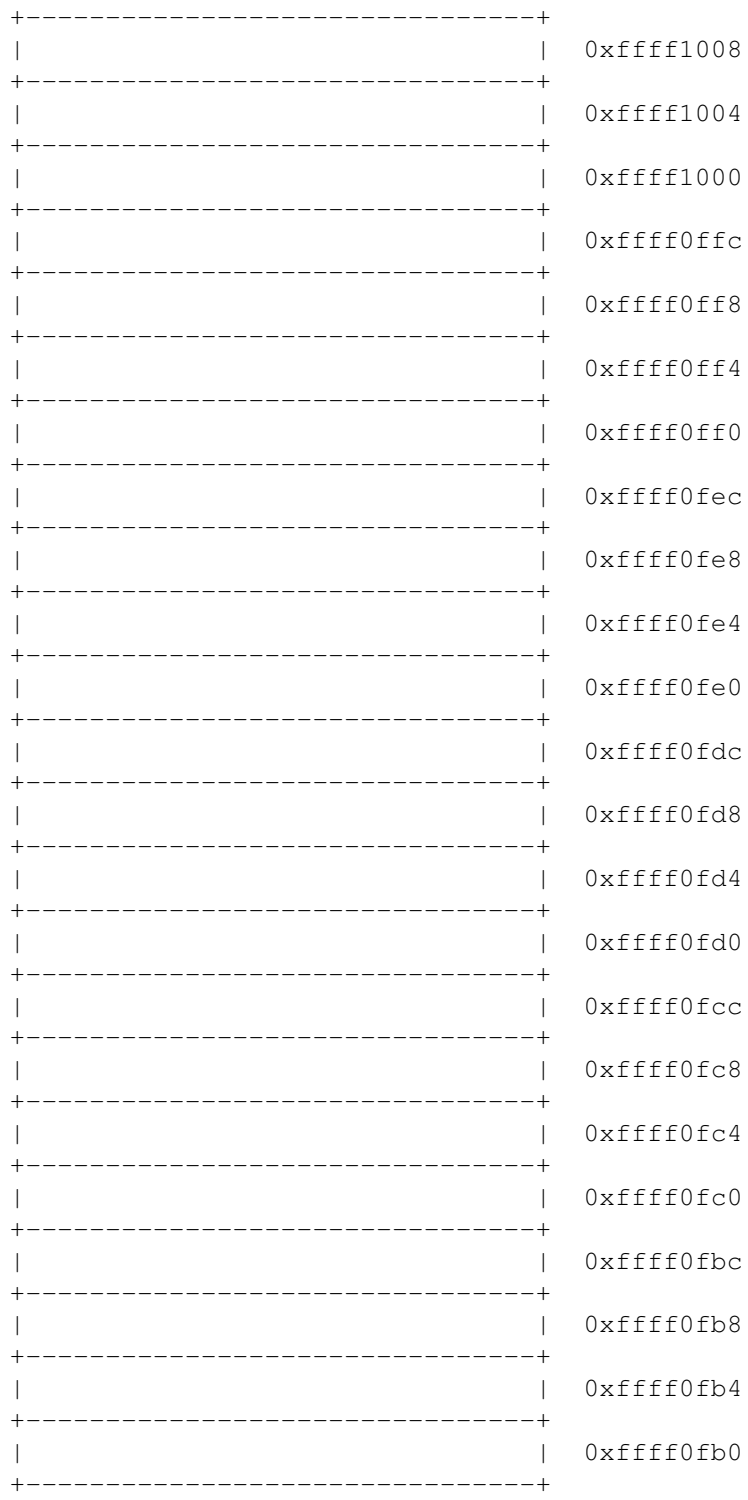
Imagine that a program makes the procedure call `gcd(213, 18)`. Also imagine that prior to the invocation, the value of %esp is `0xffff1000`—that is, `0xffff1000` is the value of %esp *immediately before* the execution of the `call` instruction.

A. Note that the call `gcd(213, 18)` will result in the following function invocations: `gcd(213, 18)`, `gcd(18, 15)`, `gcd(15, 3)`, and `gcd(3, 0)`. Using the provided code and your knowledge of IA32 stack discipline, fill in the stack diagram with the values that would be present immediately before the execution of the `leave` instruction for `gcd(15, 3)`. Supply numerical values wherever possible, and cross out each blank for which there is insufficient information to complete with a numerical value.

   Hints: The following set of C style statements describes an approximation of the operation of the instruction `idiv %ecx`, where `'/'` is the division operator and `'%'` is the modulo operator:

   ```
   %eax = %eax / %ecx
   %edx = %eax % %ecx
   ```

   Also, recall that `leave` is equivalent to `movl %ebp, %esp; popl %ebp`



B. What are the values of %esp and %ebp immediately before the execution of the `ret` instruction for `gcd(15, 3)`?

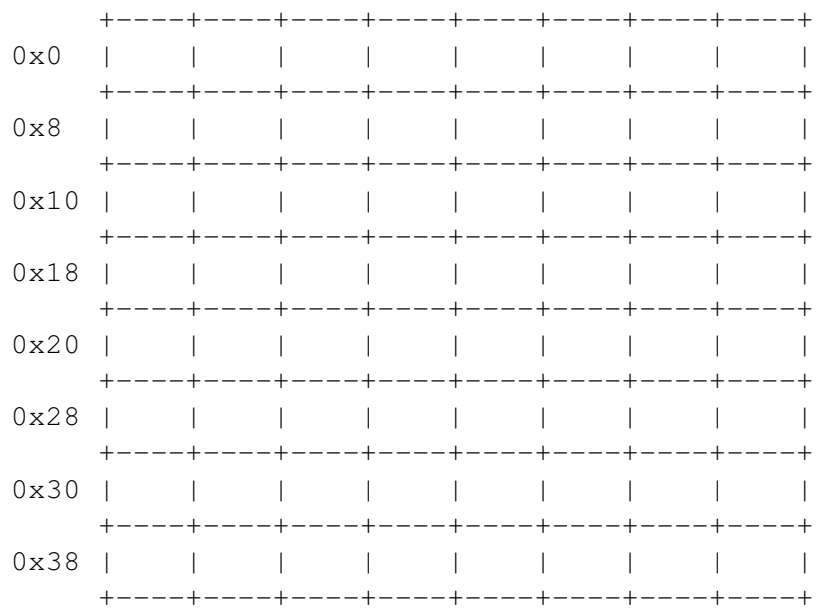
Problem 6. (8 points):

Structs. Consider the following struct on an x86-64 Linux machine:

```
struct my_struct {
    char a;
    long long b;
    short c;
    float *d[2];
    unsigned char e[3];
    float f;
};
```

A. (4 points)

Please lay out the struct in memory below. Shade in any bytes used for padding and **clearly indicate the end of the** `struct`.



B. (3 points)

Consider the following C function:

```
void foo(struct my_struct *st) {
    st->a = 'e';
    st->d[0] = NULL;
    st->c = 0x213;

    printf("%lld %p %hhu\n", st->b, &st->f, st->e[1]);
}
```

Fill in the missing pieces of the diassembled version of `foo`.

```
(gdb) disassemble foo
Dump of assembler code for function foo:
0x00000000004004e4 <+0>:    sub    $0x8,%rsp
0x00000000004004e8 <+4>:    movb   $0x65,_________(%rdi)
0x00000000004004eb <+7>:    movq   $0x0,_________(%rdi)
0x00000000004004f3 <+15>:   movw   $0x213,_________(%rdi)
0x00000000004004f9 <+21>:   movzbl _________(%rdi),%ecx
0x00000000004004fd <+25>:   lea    _________(%rdi),%rdx
0x0000000000400501 <+29>:   mov    _________(%rdi),%rsi
0x0000000000400505 <+33>:   mov    $0x40062c,%edi
0x000000000040050a <+38>:   mov    $0x0,%eax
0x000000000040050f <+43>:   callq  0x4003e0 <printf@plt>
0x0000000000400514 <+48>:   add    $0x8,%rsp
0x0000000000400518 <+52>:   retq
End of assembler dump.
```

C. (1 point)

How many bytes is the smallest possible struct containing the same elements as `my_struct`?

(a) 48

(b) 36

(c) 40

(d) None of the above

Problem 7. (10 points):

Switch statements. The problem concerns code generated by GCC for a function involving a switch statement. The code uses a jump to index into the jump table:

```
  0x4004b7:       jmpq   *0x400600(,%rax,8)
```

Using GDB, we extract the 8-entry jump table:

```
  0x400600: 0x00000000004004d1 0x00000000004004c8
  0x400610: 0x00000000004004c8 0x00000000004004be
  0x400620: 0x00000000004004c1 0x00000000004004d7
  0x400630: 0x00000000004004c8 0x00000000004004be
```

Here is the block of disassembled code implementing the switch statement:

```
# on entry: %rdi = x, %rsi = y, %rdx = z
  0x4004b0:       cmp    $0x7,%edx
  0x4004b3:       ja     0x4004c8
  0x4004b5:       mov    %edx,%eax
  0x4004b7:       jmpq   *0x400600(,%rax,8)
  0x4004be:       mov    %edi,%eax
  0x4004c0:       retq
  0x4004c1:       mov    $0x3,%eax
  0x4004c6:       jmp    0x4004da
  0x4004c8:       mov    %esi,%eax
  0x4004ca:       nopw   0x(%rax,%rax,1)
  0x4004d0:       retq
  0x4004d1:       mov    %edi,%eax
  0x4004d3:       and    $0x19,%eax
  0x4004d6:       retq
  0x4004d7:       lea    (%rdi,%rdi,1),%eax
  0x4004da:       add    %esi,%eax
  0x4004dc:       retq
```

Fill in the blank portions of the C code below to reproduce the function corresponding to this object code.

```
int test(int x, int y, int z)
{
  int result = 3;
  switch(z)
  {
    case ___:

      _______________;

    case ___:

    case ___:

      result = _______________;

      break;

    case ___:

      result = _______________;

    case ___:

      result = _______________;

      break;

    default:

      result = _______________;
  }
  return result;
}
```

Problem 8. (4 points):

Cache operation. Assume a cache memory with the following properties:

- The cache size (C) is 512 bytes (contains 512 data bytes)
- The cache uses an LRU (least-recently used) policy for eviction.
- The cache is initially empty.

Suppose that for the following sequence of addresses sent to the cache, **0, 2, 4, 8, 16, 32**, the hit rate is **0.33**. Then what is the block size (B) of the cache?

A. $B = 4$ bytes

B. $B = 8$ bytes

C. $B = 16$ bytes

D. None of the above.

Problem 9. (6 points):

Miss rate analysis. Listed below are two matrix multiply functions. The first, `matrix_multiply` computes $C = AB$, a standard matrix multiply. The second, `matrix_multiply_t`, computes $C = AB^T$, A times the transpose of B.

```
void matrix_multiply(float A[N][N], float B[N][N], float C[N][N])
{
    /* Computes C = A*B.  Assumes C starts as all zeros. */
    int i,j,k;
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            for (k=0; k<N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void matrix_multiply_t(float A[N][N], float B[N][N], float C[N][N])
{
    /* Computes C = A*transpose(B).  Assumes C starts as all zeros. */
    int i,j,k;
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            for (k=0; k<N; k++) {
                C[i][j] += A[i][k] * B[j][k];
            }
        }
    }
}
```

Assumptions:

- 8 MB 16-way cache with a block size of 64 bytes.
- N is very large, so that a single row or column cannot fit in the cache.
- `sizeof(float) == 4`.
- `C[i][j]` is stored in a register.

A. Approximately what miss rate do you expect the function `matrix_multiply` to have for large values of `N`?

(a) $\frac{1}{16}$
(b) $\frac{1}{8}$
(c) $\frac{1}{4}$
(d) $\frac{1}{2}$
(e) 1

B. Approximately what miss rate do you expect the function `matrix_multiply_t` to have for large values of `N`?

(a) $\frac{1}{16}$
(b) $\frac{1}{8}$
(c) $\frac{1}{4}$
(d) $\frac{1}{2}$
(e) 1