Andrew login ID: _____

Full Name: _____

15-213, Fall 2007

Midterm Exam

October 17, 2007, 1:00pm-2:20pm

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew login 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 of each problem is indicated. Pile up the easy points quickly and then go back to the harder problems.
- This exam is OPEN BOOK. You may use any books or notes you like. Calculators are allowed, but no other electronic devices. Good luck!

Do not write below this line

Problem	Possible Points	Your Score
1	16	
2	18	
3	16	
4	12	
5	12	
6	16	
7	10	
Total	100	

Problem 1. (16 points):

Assume we are running code on an IA32 machine, which has a 32-bit word size and uses two's complement arithmetic for signed integers. Consider the following definitions:

```
int x = foo();
unsigned ux = x;
```

Fill in the empty boxes in the table below. For each of the C expressions in the first column, either:

- State that it is true of all possible values returned by foo(), or
- Give an example where it is not true.

Puzzle	True / Counterexample
`x > 0  ⇒  (x*2) > 0`	False (TMax)
`((~(ux>>31)) & ux) != ux`	
`x >= 0  ⇒  ((x ^ (x>>31))) <= 0`	
`((-(!x)) & x) == 0`	
`x < 0  ⇒  ((~x)+1) > 0`	
`x > 0  ⇒  (x+ux) > 0U`	
`x < 0  ⇒  (x-1) < 0`	
`x >= 0  ⇒  -(((~x) >> 1) + 1) > 0`	
`((~(x>>31))+1) == (x<0)`	

Problem 2. (18 points):

Consider the following 9-bit floating point representation based on the IEEE floating point format:

- The most-significant bit is the *sign bit*.
- The next **four** bits are the *exponent*. The exponent bias is 7.
- The least-significant **four** bits are the *significand*.

The rules are like those in the IEEE standard (normalized, denormalized, representation of 0, infinity, and NAN).

Please fill in the empty boxes in the table below. For the *Value* field, you can write the value either as a fraction (e.g. $\frac{3}{4}$ or $4\frac{1}{2}$) or as an integer times a power of 2 (e.g. 3×2^{-2} or 9×2^{-1}).

Number	Value	sign bit	exponent	significand
zero	0.0	0	0000	0000
closest negative to zero				
largest positive				
n/a	-5.0			
n/a	$-2^{-4} + 2^{-6}$			
n/a	$(1\frac{9}{16}) \cdot 2^{-2}$			
n/a	$(1\frac{5}{16}) \cdot 2^{1} + (1\frac{3}{16}) \cdot 2^{-2}$			

Problem 3. (16 points):

Consider the following C function's x86-64 assembly code:

```
  # On entry:
  #    %rdi = 1st argument to the procedure
  #    %esi = 2nd argument to the procedure
<foo>:
 <foo+0>:     cmpl   $0x1,(%rdi)
 <foo+4>:     je     <foo+20>
 <foo+10>:    movl   $0x0,%eax
 <foo+15>:    jmp    <foo+73>
 <foo+20>:    movl   $0x1,%ebx
 <foo+25>:    cmp    %esi,%ebx
 <foo+27>:    jge    <foo+68>
 <foo+33>:    movslq %ebx,%rdx
 <foo+36>:    mov    0xfffffffffffffffc(%rdi,%rdx,4),%ecx
 <foo+40>:    add    %ecx,%ecx
 <foo+42>:    cmp    %ecx,(%rdi,%rdx,4)
 <foo+45>:    je     <foo+61>
 <foo+51>:    movl   $0x0,%eax
 <foo+56>:    jmp    <foo+73>
 <foo+61>:    inc    %ebx
 <foo+63>:    jmp    <foo+25>
 <foo+68>:    movl   $0x1,%eax
 <foo+73>:    retq
```

Please fill in the corresponding C code:

```
int foo(int *array, int n) {
     int i;

     if (____________________) {
          return _______;
     }

     for (i = _______; i _________; _________) {
          if (_____________________________) {
               return _______;
          }
     }

     return _______;
}
```

Problem 4. (12 points):

Consider the C code below, where H and J are constants declared with `#define`.

```
int array1[H][J];
int array2[J][H];

int copy_array(int x, int y) {
    array2[y][x] = array1[x][y];

    return 1;
}
```

Suppose the above C code generates the x86-64 assembly code shown below. (Note that "`line A`" and "`line B`" are not part of the code, but are comments that will be used later to refer back to specific lines.)

```
# On entry:
#    %edi = x
#    %esi = y
#
copy_array:
    movslq  %edi,%rcx
    movslq  %esi,%rdx
    mov     %rcx,%rbp
    shl     $0x3,%rbp
    sub     %rcx,%rbp   # line A
    add     %rdx,%rbp
    movslq  %edi,%rcx
    movslq  %esi,%rdx
    mov     %rdx,%rax
    add     %rax,%rax
    add     %rdx,%rax   # line B
    lea     (%rax,%rcx,1),%rsi
    mov     array1(,%rbp,4),%eax
    mov     %eax,array2(,%rsi,4)
    mov     $0x1,%eax
    retq
```

Give the valid C language expression that best describes the contents of the following registers. A valid C language expression is one that can be parsed by a C compiler (e.g., `811`, `x+1`, `y*3+x`, etc.).

- `%rbp` *immediately after* the instruction at line `A` has executed:
- `%rax` *immediately after* the instruction at line `B` has executed:
- The constant value `H`:
- The constant value `J`:

Problem 5. (12 points):

Consider the following C declarations:

```
typedef struct {
    int x;
    int y;
    int sensor_id;
} Sensor;

typedef struct {
    int sensor_id;
    double data;
} Data;

typedef struct {
    Sensor *sensor;
    char status;
} Sensor_Status;

Sensor sensor;
Data data;
Sensor_Status status_array[10];
```

Assume x86-64 alignment constraints. For each of the three data structures listed in the table below, please indicate the total amount of space (in bytes) that is *allocated* and *wasted*:

Data Structure	Total *Allocated* Space (in bytes)	Total *Wasted* Space (in bytes)
`sensor`		
`data`		
`status_array`		

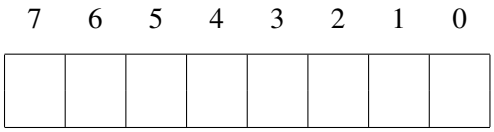
Problem 6. (16 points):

Consider a computer with an 8-bit address space and an associative 64-byte data cache, with a LRU replacement policy. Assume that the cache is 2-way set associative and has 8-byte cache lines. The boxes below represent the bit-format of an address. In each box, indicate which field that bit represents (it is possible that a field does not exist). Here are the fields:

CO: The byte offset within the cache line

CI: The cache (set) index

CT: The cache tag



The table below on the left shows a trace of load addresses accessed in the data cache. Below on the right is a list of possible final states of the cache, showing the hex value of the tag for each cache line in each set. Which is the correct final cache state? How many of the loads were hits? Assume that initially all cache lines are invalid (represented by X), and that **sets are filled from left to right**.

Load No.	Hex Address	Binary Address
1	1a	0001 1010
2	55	0101 0101
3	e6	1110 0110
4	53	0101 0011
5	77	0111 0111
6	28	0010 1000
7	94	1001 0100
8	a6	1010 0110
9	75	0111 0101
10	c7	1100 0111
11	56	0101 0110

(a)

Set 3		Set 2		Set 1		Set 0	
0	X	4	2	1	X	6	5

(b)

Set 3		Set 2		Set 1		Set 0	
0	X	2	3	1	X	6	5

(c)

Set 3		Set 2		Set 1		Set 0	
0	X	2	4	1	X	6	5

(d)

Set 3		Set 2		Set 1		Set 0	
0	X	2	3	1	X	7	6

(e)

Set 3		Set 2		Set 1		Set 0	
0	X	4	2	1	X	7	6

(f)

Set 3		Set 2		Set 1		Set 0	
X	0	5	2	1	3	7	6

(g)

Set 3		Set 2		Set 1		Set 0	
0	2	4	2	1	3	5	6

Correct final state: _____ Number of hits: _____

Problem 7. (10 points):

Consider the C code below:

```
int forker(int x) {
    int pid;

    printf("A");
    if (x > 0) {
        pid = fork();
        printf("B");
        if (pid == 0) {
            printf("C");
        } else {
            waitpid(pid, NULL, 0);
            printf("D");
        }
    }
    printf("E");
    exit(4);
}
```

Consider each of the following outputs and circle the ones that could be produced by the code above (after all processes are terminated).

AAEE

ABCDE

ABCBEDE

ABCBDEE

ABCEBDE