15-213 Introduction to Computer Systems

Exam 1

February 28, 2006

Name: **Model Solution**

Andrew User ID: **fp**

Recitation Section: _____

- This is an open-book exam. Notes are permitted, but not computers.
- Write your answer legibly in the space provided.
- You have 80 minutes for this exam.

Problem	Max	Score
1	**10**	
2	**10**	
3	**10**	
4	**15**	
5	**10**	
6	**20**	
Total	**75**	

1. Integers (10 points)

For each of the following propositions, indicate if they are true or false. If false, give a counterexample. We assume that the variables are declared as follows

```
int x,y;
unsigned u,v;
```

and initialized to some unknown value. You should formulate your counterexamples in terms of the word size w. We have given the first answer as an example. You may assume right shift is arithmetical.

If `x > 0` then `x + 1 > 0`	false	$x = 2^w - 1$
If `x < 0` then `x * 2 < 0`	**false**	$x = -2^{w-1}$
If `x > 0` then `x * x > 0`	**false**	$x = 2^{w/2}$
`u >= 0`	**true**	
`u <= -1`	**true**	
If `x > y` then `-x < -y`	**false**	$x = 0$ **and** $y = -2^{w-1}$
If `u > v` then `-u > -v`	**false**	$u = 2$ **and** $v = 1$
If `x >= 0` then `-x <= 0`	**true**	
If `x < 0` then `-x > 0`	**false**	$x = -2^{w-1}$
`(unsigned) x == x`	**true**	
`(x << 1) >> 1 == x`	**false**	2^{w-2}

2. Floating Point (10 points)

Fill in the blank entries in the following table. We assume the standard representation of single-precision floating point numbers with 1 sign bit, $k = 8$ bits for the exponent and $n = 23$ bits for the fractional value. This means the bias is $2^7 - 1 = 127$.

Value	Form $(-1)^s \times M \times 2^E$ for $1 \leq M < 2$	Hexadecimal Representation
132	1.00001×2^7	`0x43040000`
$-1\frac{1}{2}$	$-1 \times 1.1 \times 2^0$	**`0xBFC00000`**
$\frac{1}{2}$	1.0×2^{-1}	**`0x3F000000`**
$\frac{3}{8}$	1.1×2^{-2}	`0x3EC00000`
2^{-149}	1.0×2^{-149}	**`0x00000001`**

3. Structures and Alignment (10 points)

Consider the following C declaration which is part of a small cache simulator.

```
typedef struct {
  char valid;
  char dirty;
  int tag;
  char block[32];
  int stamp;
} cache_line;
```

A `cache_line` structure will be 44 bytes long.

1. (5 points) Assuming that a `cache_line` structure `c` is allocated at address `0x08000000`, give the values each of the following expressions.

`&c.valid`	**0x08000000**
`&c.dirty`	**0x08000001**
`&c.tag`	**0x08000004**
`&c.block`	**0x08000008**
`&c.stamp`	**0x08000028**

2. (5 points) Which of the following is always true, always false, or undefined (for example, if it could yield different answers at different times, or if it compares elements of different type).

`c.block[31] == *(c.block+31)`	**always true**
`&c.valid == &c.dirty`	**always false**
`&c.block[32] == &c.stamp`	**undefined**
`*&c.dirty == c.dirty`	**always true***
`&*c.block == c.block`	**always true**

$*$ **There was a typographical error in this line, so we also accepted *undefined* as a correct answer.**

4. Caches (15 points)

We continue the cache simulator. We recall the declaration for `cache_line` for reference and then declare the representation of main memory, sets of cache lines, and the cache itself. We also declare a counter, to be incremented on every cache access to implement an LRU cache line replacement policy.

```
typedef struct {
  char valid;
  char dirty;
  int tag;
  char block[32];
  int stamp;
} cache_line;

char main_memory[1<<16];   /* array representing main memory */
typedef cache_line cache_set[4]; /* set of cache lines */
cache_set cache[16];             /* cache, uninitialized */
int counter = 0;           /* counter for LRU replacement policy */
```

1. (5 points) Fill in the blanks:

   The simulated cache is _____**4** way associative,

   where each cache block contains _____**32** bytes. In total,

   the cache holds _____**2048** bytes. The simulated main

   memory holds _____2^{16} bytes and addresses may

   be _____**16** bits wide.

2. (10 points) Complete the following code to load a byte from our simulated memory hierarchy. Assume a function `int lru(int i);` which, when given a set index `i`, returns the least recently used cache line in set `i`. Assume moreover, that if the dirty bit of this cache line is on, the `lru` function will write it back to memory, implementing a write-back policy.

   We also assume a function
   `void copy_block(char* src, char* dest);`
   which copies a cache block from `src` to `dest`.

```
char load_byte(unsigned short addr) {
  int j;
  int tag = addr & 0xFE00;
  int i = (addr & 0x01E0) >> 5;
  int offset = addr & 0x001F;
  for (j = 0; j < 4; j++) {
    if (cache[i][j].valid && tag == cache[i][j].tag) {
      cache[i][j].stamp = counter++;
      return (cache[i][j].block)[offset];
    }
  }
  j = lru(i);
  copy_block(&main_memory[addr & 0xFFE0], cache[i][j].block);
  cache[i][j].valid = 1;
  cache[i][j].dirty = 0;
  cache[i][j].tag = tag;
  cache[i][j].stamp = counter++;
  return (cache[i][j].block)[offset];
}
```

5. Assembly Language (10 points)

For reference, we repeat the relevant declarations from the previous question.

```
typedef struct {
  char valid;
  char dirty;
  int tag;
  char block[32];
  int stamp;
} cache_line;
typedef cache_line set[4];
set cache[16];
```

Fill in the missing parts of the C program and the assembly code so that the assembly code implements the C function. The C function is supposed to reset the cache by clearing all `valid` flags. Recall that on the x86-64, the `imulq` instruction can take a constant, a source register, and a destination register in that order.

```
void reset() {
  int i,j;
  for (i = 0; i < 16; i++)
    for (j = 0; j < 4; j++) {
      cache[i][j].valid = 0;
    }
}
```

```
reset:
        xorl    %ecx, %ecx
.L9:
        movslq  %ecx,%rax
        movl    $3, %edx
        imulq   $176, %rax, %rax
.L8:
        movb    $0, cache(%rax)
        addq    $44, %rax
        decl    %edx
        jns     .L8
        incl    %ecx
        cmpl    $15, %ecx
        jle     .L9
        ret
```

6. Optimization (20 points)

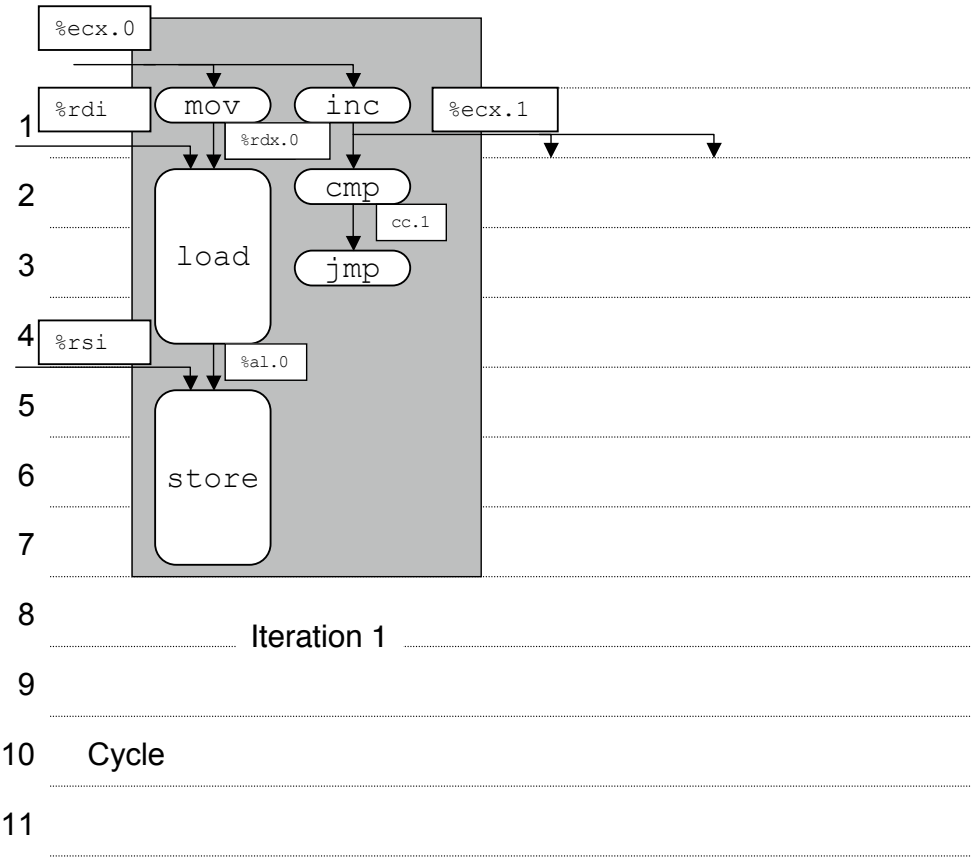
Consider the following version of the `copy_block` function that we assumed in Problem 4 for copying a cache block to and from memory. Recall that `%al` represents the lowest byte of the `%rax` register.

```
copy_block:
        xorl    %ecx, %ecx
.L19:
        movslq  %ecx,%rdx
        incl    %ecx
        movzbl  (%rdx,%rdi), %eax
        cmpl    $31, %ecx
        movb    %al, (%rdx,%rsi)
        jle     .L19
        ret
```

1. (6 pts) For the inner loop, show the corresponding processor operations using the register renaming notation used in class and in the textbook. Do not rename registers that are invariant throughout multiple loop iterations.

```
movslq  %ecx,%rdx          movslq     %ecx.0          →  %rdx.0

incl    %ecx               incl       %ecx.0          →  %ecx.1

movzbl  (%rdx,%rdi), %eax  load       (%rdx.0,%rdi)   →  %eax.0

cmpl    $31, %ecx          cmpl       $31, %ecx.1     →  cc.1

movb    %al, (%rdx,%rsi)   store      %al.0           →  (%rdx.0,%rsi)

jle     .L19               jle-taken  cc.1
```

2. (10 pts) Label the following timed data dependency diagram with operations from the program (boxes with rounded corners) and possibly renamed registers (square boxes).



3. (2 pts) This code may execute more slowly if there is aliasing between the source and destination block. Briefly explain why.

Because the load on the next iteration may have to wait for the store from the previous iteration if they access the same memory location.

4. (2 pts) Despite the fact that this code has good locality because it strides through memory in increments of one, it is not particularly efficient. Describe one way to improve its efficiency significantly. You may show source code if it helps your explanation, but you are not required to do so.

It's slow, because it only moves one byte at a type. Moving 8 4-byte double words or 4 8-byte quad words would likely be much faster.