

15-213 Introduction to Computer Systems

Exam 1

February 22, 2005

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	15	
2	15	
3	15	
4	15	
5	7	
6	8	
Total	75	

1. Floating Point (15 points)

Consider the following function in assembly language.

```
problem1:  
    pushl    %ebp  
    movl    %esp, %ebp  
    flds    16(%ebp)  
    fadds   12(%ebp)  
    fmul    8(%ebp)  
    leave  
    ret
```

Recall that `fld` pushes the argument onto the floating point register stack, `fadd` pushes and adds, and `fmul` pushes and multiplies. The suffix `s` determines size of the operands to be 32 bits.

1. (5 points) Write out a corresponding function definition in C.

```
float problem1 (float x, float y, float z) {  
    return x * (y + z);  
}
```

2. (1 points) 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. What is the bias?

$$2^{k-1} - 1 = 127$$

3. (5 points) Give the hexadecimal representation of the number $\frac{1}{4}$.

0x3E800000

4. (4 points) Give the representation of 0xBE000000 as a fraction.

$$-\frac{1}{8}$$

2. Pointers and Functions (15 points)

The following somewhat misguided code tries to optimize the exponential function on positive integers by creating an array of function pointers called `table` and using them if the exponent is less than 4.

```
int exp0 (int x) { return 1; }
int exp1 (int x) { return x; }
int exp2 (int x) { return x * x; }
int exp3 (int x) { return x * x * x; }
```

```
_____ = {
    &exp0, &exp1, &exp2, &exp3
};
```

```
int exp (int x, int n) {
    int y = 1;
    if (n < 4)
        return _____ ;
    while (n > 0) {
        y *= x;
        n--;
    }
    return y;
}
```

1. (5 points) Fill in the missing declaration of `table` and complete the return statement.

```
int (*table[4]) (int x)
(*table[n])(x)
```

2. (3 points) Note the assignment of program variables to registers in the following assembly code produced by gcc for `exp`. We have elided some alignment instructions and the specialized `expn` functions.

```

table:
    .long    exp0
    .long    exp1
    .long    exp2
    .long    exp3

exp:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    12(%ebp), %edx
    cmpl    $3, %edx
    movl    8(%ebp), %ecx
    movl    $1, %eax
    jle     _____
    testl   %edx, %edx
    jle     _____

.L11:
    decl    %edx
    imull   %ecx, %eax
    testl   %edx, %edx
    jg      .L11

.L6:
    leave
    ret

.L14:
    subl    $12, %esp
    pushl   %ecx
    call    _____
    jmp     .L6

```

Variable	Register
x	%ecx
n	%edx
y	%eax

3. (3 points) Justify the use of particular registers chosen by gcc.

`%ecx`, `%edx`, and `%eax` are caller-save registers, which means the code of `exp` does not have to save the prior values on the stack. In addition, `%eax` is a good choice for `y`, since it always holds the return value of a function.

4. (4 points) Fill in the three missing lines of assembly code.

```
.L14  
.L6  
*table( ,%edx,4)
```

3. Structures and Alignment (15 points)

Consider the following C declaration:

```
typedef struct {
    unsigned short id;
    char* name;
    char andrew_id[9];
    char year;
    int raw_score;
    double percent;
} Student;
```

- (5 points) On the template below, show the layout of the Student structure. Delineate and label the areas for each component of the structure, cross-hatching the parts that are allocated but not used. Clearly mark the end of the structure. You should assume Linux alignment rules. Do **not** fill in the data fields; you will need them to answer the next question.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	00														

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	FE			01	02	00	00								

0-1 id; 2-3 x; 4-7 name; 8-16 andrew_id; 17 year; 18-19 x; 20-23 raw_score; 24-31 percent

- (3 points) Show the state of an instance of the Student structure after the following code is executed. Write in the assigned values by filling them into the assigned squares above and leave the remaining squares blank. Assume a Linux/x86 architecture and use hexadecimal format.

```
Student jenn;
jenn.id = 16;
jenn.year = -2;
jenn.raw_score = 513;
```

3. (5 points) Rewrite the declaration to use the minimal amount of space for the structure with the same components. You should make sure that the amount of space is minimal for both Linux and Windows alignment rules.

```
typedef struct {  
    char* name;  
    char andrew_id[9];  
    char year;  
    unsigned short id;  
    double percent;  
    int raw_score;  
} CompactStudent;
```

Note that `double percent` cannot be last (at offset 20), because under the Windows alignment rules its address must be a multiple of 8.

4. (2 points) How many bytes does your new structure require?

28

4. Optimization (15 points)

Consider the following declaration of a linked list data structure in C:

```
typedef struct LIST {
    struct LIST *next;
    int data;
} List;
```

We use a next pointer of NULL to mark the end of the list, and we assume we have a function `int length (List* p);` to calculate the length of a non-circular linked list. You may assume all linked lists in this problem are non-circular.

1. (3 points) The function `count_pos1` is supposed to count the number of positive elements in the list at `p` and store it at `k`, but it has a serious bug which may cause it not to traverse the whole list. Insert one line and change one line to fix this problem.

```
void count_pos1 (List *p, int *k) {
    int i;
    *k = 0;
    for (i = 0; i < length(p); i++) {
        if (p->data > 0)
            (*k)++;
        p = p->next;
    }
}
```

Insert `int n = length(p);` before the loop and change the bounds check to `i < n`

2. (5 points) Further improve the efficiency of the corrected function from part 1 by eliminating the iteration variable `i`, changing it to a iteration using only pointers instead. Fill in the template below.

```
void count_pos2 (List *p, int *k) {
    *k = 0;
    while (_____) {
        if (p->data > 0)
            (*k)++;
        _____;
    }
}
```



```
p and p = p->next;
```

3. (2 points) Your function from part 2 still has a bug in that it does not always “count the number of positive elements in the list at p and stores it at k ”. Explain the problem.

It may sometimes be incorrect, because of the possibility of aliasing. If k is a pointer to an element of the linked list, the answer may be incorrect since one of the list elements is changed before it starts counting.

4. (5 points) Fix the problem you identified in part 3. Your function should also run faster by reducing memory accesses when compared to the function in part 2.

```
void count_pos3 (List *p, int *k) {
    int i;
    while (p) {
        if (p->data > 0)
            i++;
        p = p->next;
    }
    *k = i;
}
```

5. Out-of-Order Execution (7 points)

1. (5 points) The inner loop corresponding to our answer to part 4 of the previous problem has the following form:

```
.L48:
    movl    4(%eax), %ecx    # load 4(%eax.0) --> %ecx.1
    testl  %ecx, %ecx
    jle    .L47
    incl  %edx

.L47:
    movl    (%eax), %eax
    testl  %eax, %eax
    jne    .L48
```

Annotate each line with the execution unit operations for one iteration, assuming the inner branch is **not** taken. To get you started, we have filled in the first operation.

<pre>.L48: movl 4(%eax), %ecx testl %ecx, %ecx jle .L47 incl %edx .L47: movl (%eax), %eax testl %eax, %eax jne .L48</pre>	<pre>load 4(%eax.0) → %ecx.1 testl %exc.1, %ecx.1 → cc.1 jne-not-taken cc.1 incl %edx.0 → %edx.1 load (%eax.0) → %eax.1 testl %eax.1, %eax.1 → cc.2 jne-taken cc.2</pre>
---	--

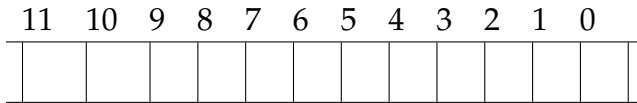
2. (2 points) Assuming most numbers in a list are positive and branch predictions are correct, give a plausible lower bound on the CPE for the inner loop based on the execution unit operations. Optimistically assume memory accesses are cache hits.

The two load and subsequent test and jump operations are independent and, given enough processor resources, can be done in parallel. Moreover, there is no data dependency between the increment of %edx and the above operations. However, both loads within an iteration depend on the result of the second load in the previous iteration. This means we cannot attain a CPE below 3.

6. Cache Memory (8 points)

Assume we have byte-addressable memory with addresses that are 12 bits wide. We have a 2-way set associative cache with with 4 byte block size and 4 sets.

- (3 points) On the template below, show the portions of an address that make up the tag, the set index, and the block offset.



11-4 tag, 3-2 set index, 1-0 block offset

- (5 points) Consider the following cache state, where all addresses, tags, and values are given in hexadecimal format.

Set Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	00	1	40	41	42	43
	83	1	FE	97	CC	D0
1	00	1	44	45	46	47
	83	0	-	-	-	-
2	00	1	48	49	4A	4B
	40	0	-	-	-	-
3	FF	1	9A	C0	03	FF
	00	0	-	-	-	-

For each of following memory accesses indicate if it will be a cache hit or miss, when they are **carried out in sequence** as listed. Also give the value of a read if it can be inferred from the information in the cache.

Operation	Address	Hit?	Read Value (or Unknown)
Read	0x834	No	Unknown
Write	0x836	Yes	(not applicable)
Read	0xFFD	Yes	C0