Last Name: Paskalov

First Name: Daniel

UID: [redacted]

Section: 1B

Computer Science 33, Winter 2012

Midterm 2

February 28, 2012

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name 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 come back to the harder problems.
- This exam is open book and open notes.
- You have 1 hour and 50 minutes to complete the test. Good luck!

0 (1):	1
1 (12):	12
2 (15):	13
3 (12):	12
4 (13):	13
5 (14):	11
6 (9):	7
7 (12):	10
8 (12):	12
TOTAL (100):	91

Problem 1. (12 points):

Consider the source code below, where M and N are constants declared with #define.

```
        5  7
int mat1[M][N];
int mat2[N][M];

int sum_element(int i, int j)
{
  return mat1[i][j] +  mat2[i][j];
}             1  7          1  5
```

A. Suppose the above code generates the following assembly code:

```
sum_element:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax           i
        movl 12(%ebp),%ecx          j
        sall $2,%ecx                4j
        leal 0(,%eax,8),%edx          8i
        subl %eax,%edx              7i
        leal (%eax,%eax,4),%eax       5i
        movl mat2(%ecx,%eax,4),%eax   mat2 + 4(5i + j)      mat2-5 cols
        addl mat1(%ecx,%edx,4),%eax  +mat1 + 4(7i + j)      mat1 = 7 cols
        movl %ebp,%esp
        popl %ebp
        ret
```

What are the values of M and N?

M = 5

N = 7 ✓

Problem 2. (15 points):

Consider the following C declaration, assuming a 32 bit machine (1 byte size for char, 4 byte size for int, 8 bytes for double):

```
struct Node{
    char c;
    double d;
    struct Node* n;
    int i;
    struct Node* l;
    struct Node* r;
};

typedef struct Node* pNode;

/* NodeTree is an array of N pointers to Node structs */
pNode NodeTree[N];
```

A. Using the template below (allowing a maximum of 32 bytes), indicate the allocation of data for a Node struct. Mark off and label the areas for each individual element (there are 6 of them) using the letter of the variable to indicate the byte positions spanned by the element (for example, for element "int i" you would have iiii across four spaces). Indicate with an "x" the parts that are allocated to the struct, but not used for storing meaningful data (i.e. space within the struct allocated to satisfy alignment).

Assume the Linux alignment rules discussed in class (1 byte alignment for char, 4 byte alignment for int, double, and pointers). **Clearly indicate the right hand boundary of the data structure with a vertical line.**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
c	x	x	x	d	d	d	d	d	d	d	d	n	n	n	n	i	i	i	i	l	l	l	l	r	r	r	r				

B. For each of the four C references below, please indicate which assembly code section (labeled A – F) places the value of that C reference into register `%eax`. If no match is found, please write "NONE" next to the C reference.

The initial register-to-variable mapping for each assembly code section is:

```
  %eax = starting address of the NodeTree array
  %edx = i
```

```
-------------------------------------------------------------------
C References:

I.    __D__ NodeTree[i]-> i

-2 II.   __C__ NodeTree[i]-> l -> l -> c

III.  __F__ NodeTree[i]-> n -> n -> i

IV.   __B__ NodeTree[i]-> r -> l -> l
-------------------------------------------------------------------
```

Linux/IA32 Assembly:

$\boxed{i}$ ↑

```
A.        sall $2, %edx                  B.  sall $2,%edx
          leal (%eax,%edx),%eax              leal (%eax,%edx),%eax
          movl 16(%eax),%eax                 movl (%eax),%eax
                                             movl 24(%eax),%eax
                                             movl 20(%eax),%eax
                                             movl 20(%eax),%eax

C:        sall $2,%edx                   D:  sall $2,%edx
          leal (%eax,%edx),%eax              leal (%eax,%edx),%eax
          ~~movl 20(%eax),%eax~~             movl (%eax),%eax
          movl 20(%eax),%eax                 movl 16(%eax),%eax
          movsbl (%eax),%eax

E:        sall $2, %edx                  F:  sall $2, %edx
          leal (%eax,%edx),%eax              leal (%eax,%edx),%eax
          movl (%eax),%eax                   movl (%eax),%eax
          movl 16(%eax),%eax                 movl 12(%eax),%eax
          movl 16(%eax),%eax                 movl 12(%eax),%eax
          movl 20(%eax),%eax                 movl 16(%eax),%eax
```

Problem 3. (12 points):

Assume the following sizes for this problem:
double: 8 bytes
int/unsigned: 4 bytes
short: 2 bytes
char: 1 byte

Consider the following C declaration:

```
union Uni {
   char c[20];  20
   double d[2]; 16
   short s[3];  6
   unsigned u;  4
   int i;       4
} uni1;
```

A. How much memory space in bytes would need to be allocated for uni1?

20 bytes because the largest data is the array of characters each being a byte long

B. Assume uni1.d[0] is initialized to -5.5, and then the value of uni1.s[0] is updated from 0xC0B0 to 0x40B0. What is the value of uni1.d[0] after the update, assuming a big-endian representation?

$5\frac{1}{2}$ $\frac{11}{2}$

1100 0000 1101 0000
0100 0000 1101 0000

$\boxed{+5.5}$

Since the update to s[0] changes the sign bit of d[0] only since they share the same space

C. Which of the following functions (assert1, assert2, assert3) always returns true (i.e. never returns 0)? Write the name of the function that satisfies this condition in the blank below, or write "none" if you believe that none of these functions always returns true.

Assert 1

```
short assert1(Uni uni, int num) {
   if (uni.i == num) {
      return ((int) uni.u) == num;
   }
}


short assert2(Uni uni, short sh) {
   if (uni.s[0] == sh) {
      return ((short) uni.i) == sh;
   }
}


short assert3(Uni uni, double dbl) {
   if (uni.d[0] == dbl) {
      return ((double) uni.i) == dbl;
   }
}
```

Problem 4. (13 points):
Consider the following C code involving function pointers:

```
#include <stdio.h>

int (*fp_a)(int);
int (*fp_b)(int);

int rand(void);

int execute_funcs(int arg) {
   int temp;
   temp = fp_a(arg);
   return fp_b(temp);
}

int func1(int i) {
   return i + 1;
}


int func2(int i) {
   return i * 2;
}

void create_func_pointers(int test) {
   if (test >= 0 )
     fp_a = func1;
   else
     fp_a = func2;

   fp_b = func1;
}

void main() {
  int a, b;
  a = rand();
  b = 2;
  create_func_pointers(a);
  execute_funcs(b);
}
```

A. Explain how memory aliasing can occur in execute_funcs().

The compiler doesn't know what function fp_a points to.
If fp_a points to func1, then we have memory aliasing since
fp_b also points to func1

B. What value would be returned from execute_funcs if no memory aliasing has occurred?

2*2+1 = 5

C. What value would be returned from execute_funcs if memory aliasing has occurred?

2+1+2 = 4

Consider the following re-implementation of execute_funcs:

```
int execute_funcs(int arg) {
   int temp;
   temp = func1(arg);
   return func2(temp);
}
```

D. Would this new implementation of execute_funcs allow a greater degree of compiler optimization, a lesser degree of compiler optimization, or would this update have no effect on the compiler's ability to optimize the code?

There won't be a difference since function calls are treated as a black box and the compiler must always call them, regardless of what they do

Performance Optimization

Problem 5. (14 points): 11

The following problem concerns optimizing a procedure for maximum performance on an Intel Pentium III. The following are the performance characteristics of the functional units for this machine across various operations:

Operation	Latency	Issue Time
Shifts	1	1
Integer Add	1	1
Integer Multiply	4	1
Integer Divide	36	36
Floating Point Add	3	1
Floating Point Multiply	5	2
Floating Point Divide	38	38
Load or Store (Cache Hit)	1	1

You've just joined a programming team that is trying to develop the world's fastest factorial routine. Starting with recursive factorial, they've converted the code to use iteration:

```
int fact(int n)
{
  int i;
  int result = 1;

  for (i = n; i > 0; i--)
    result = result * i;

  return result;
}
```

By doing so, they have reduced the number of cycles per element (CPE) for the function from around 63 to around 4 (really!). Still, they would like to do better.

4 A. Explain why the iterative version of fact would perform so much better than a recursive version.

The recursive version will have a lot of overhead to deal with all the ensuing function calls and the compiler has less options to optimize because of all the function calls.

One of the programmers heard about loop unrolling. He generated the following code:

```
int fact_u2(int n)
{
  int i;
  int result = 1;

  for (i = n; i > 0; i-=2) {
    result = (result * i) * (i-1);
  }

  return result;
}
```

3·1·2

0

6·1·0

Unfortunately, the team has discovered that this code returns 0 for some values of argument n.

3

B. For what values of n will `fact_u2` and `fact` return different values?

when n is odd it will do an extra iteration multiplying the result by 0 (for fact_u2)

3

C. Show how to fix `fact_u2` so that its behavior is identical to `fact`. Your update must only change a single character of the original code.

for(i=n; i>1; i-=2){

1

D. Suppose it could be assumed that the value of the parameter n would always be greater than 1, and that the function would be extremely heavily used to the point where even saving a couple of loop iterations and/or cycles of latency would help overall performance. A new version of the "fact" function below, fact_ng1(int n), attempts to take advantage of this knowledge by taking the code for n=2 outside of the loop. Fill in the blanks in the return statement to incorporate the same effect as the n=2 loop iteration had in the original function, incorporating a reduction in strength optimization. Each blank should contain either an operator, a variable name, or a constant value.

```
int fact_ng1(int n) {
   int i;
   int result = 1;

   for (int i = n; i > 2; i--) {
     result = result * i;
   }

   return result _*_  _2_;
}
```

The following problem concerns optimizing a procedure for maximum performance on an Intel Pentium III. Using the same performance characteristics for operations as in Problem 5, consider the following two procedures:

Loop 1	Loop 2
`int loop1(int *a, int x, int n)`	`int loop2(int *a, int x, int n)`
`{`	`{`
`  int y = x*x;`	`  int y = x*x;`
`  int i;`	`  int i;`
`  for (i = 0; i < n; i++)`	`  for (i = 0; i < n; i++)`
`    x = y * a[i];`	`    x = x * a[i];`
`  return x*y;`	`  return x*y;`
`}`	`}`

When compiled with GCC, we obtain the following assembly code for the inner loop:

Loop 1	Loop 2
`.L21:`	`.L27:`
`    movl %ecx,%eax`	`    imull (%esi,%edx,4),%eax`
`    imull (%esi,%edx,4),%eax`	`    incl %edx`
`    incl %edx`	`    cmpl %ebx,%edx`
`    cmpl %ebx,%edx`	`    jl .L27`
`    jl .L21`	

Problem 6. (9 points): 7

Running on a Pentium III Xeon server, we find that Loop 1 requires 3.0 clock cycles per iteration, while Loop 2 requires 4.0.

3 A. Explain how it is that Loop 1 is faster than Loop 2, even though it has one more instruction. (Hint: consider how the latency of the multiplication operation might affect one of the loops more than another, given that pipelined processors have multiple instructions in execution at once whenever possible and that all operands must be known for an instruction to enter the pipeline).

~~The~~ The second loop needs the value of x to get the answer and must wait for the previous iteration to complete while y is constant and ~~the~~ the first loop can begin computing y*a[i] while the previous loop finishes updating x

1 B. By using the compiler flag `-funroll-loops`, we can compile the code to use 4-way loop unrolling. This speeds up Loop 1. Explain why.

The loop can compute y*a[i] + y*a[i+1] + ... + y*a[i+3] and then update x at once. The addition takes less time then having to wait for x to update

3 C. Even with loop unrolling, we find the performance of Loop 2 remains the same. Explain why.

Each iteration still has to wait for the value of x regardless.

Problem 7. (12 points):

The following problem concerns basic cache lookups.

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Physical addresses are 12 bits wide.
- The cache is 4-way set associative, with a 2-byte block size and 32 total lines.

In the following tables, **all numbers are given in hexadecimal**. The contents of the cache are as follows:

4-way Set Associative Cache																
Index	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1
0	29	0	34	29	87	0	39	AE	7D	1	68	F2	8B	1	64	38
1	F3	1	0D	8F	3D	1	0C	3A	4A	1	A4	DB	D9	1	A5	3C
2	A7	1	E2	04	AB	1	D2	04	E3	0	3C	A4	01	0	EE	05
3	3B	0	AC	1F	E0	0	B5	70	3B	1	66	95	37	1	49	F3
4	80	1	60	35	2B	0	19	57	49	1	8D	0E	00	0	70	AB
5	EA	1	B4	17	CC	1	67	DB	8A	0	DE	AA	18	1	2C	D3
6	1C	0	3F	A4	01	0	3A	C1	F0	0	20	13	7F	1	DF	05
7	0F	0	00	FF	AF	1	B1	5F	99	0	AC	96	3A	1	22	79

Part I

The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

S=8
b=1

CO The block offset within the cache line
CI The cache index
CT The cache tag

11	10	9	8	7	6	5	4	3	2	1	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO

Part II

For the given physical address, indicate the cache entry accessed and the cache byte value returned **in hex**. Indicate whether a cache miss occurs.

If there is a cache miss, enter "-" for "Cache Byte returned".

Physical address: `3B6`

A. Physical address format (one bit per box)

11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	1	0	1	1	0

B. Physical memory reference

Parameter	Value	
Cache Offset (CO)	0x0	
Cache Index (CI)	0x3	
Cache Tag (CT)	0x3B	
Cache Hit? (Y/N)	N	-1
Cache Byte returned	0x –	-1

Problem 8. (12 points):

A bitmap image is composed of pixels. Each pixel in the image is represented as four values: three for the primary colors(red, green and blue - RGB) and one for the transparency information defined as an alpha channel.

In this problem, you will compare the performance of direct mapped and 4-way associative caches for a square bitmap image initialization. Both caches have a size of 128 bytes. The direct mapped cache has 8-byte blocks while the 4-way associative cache has 4-byte blocks.

You are given the following definitions

```
typedef struct{
    unsigned char r;
    unsigned char g;
    unsigned char b;
    unsigned char a;
}pixel_t;

pixel_t  pixel[16][16];
register int i, j;
```

$\frac{2^7}{2^3} = 2^4 = 16$

$\frac{2^7}{2^4} = 2^3$

Also assume that

- `sizeof(unsigned char) = 1`
- `pixel` begins at memory address 0
- Both caches are initially empty
- The array is stored in row-major order
- Variables `i,j` are stored in registers and any access to these variables does not cause a cache miss

A. What fraction of the writes in the following code will result in a miss in the direct mapped cache?

```
for (i = 0; i < 16; i ++){
    for (j = 0; j < 16; j ++){
        pixel[i][j].r = 0;
        pixel[i][j].g = 0;
        pixel[i][j].b = 0;
        pixel[i][j].a = 0;
    }
}
```

$8 \cdot 16$

$2^3 \cdot 2^4 = 2^7 = 128$

$1 \ \frac{2^7}{2^3} = 16$

Miss rate for writes to pixel: 12.5 %

B. Using code in part A, what fraction of the writes will result in a miss in the 4-way associative cache?

Miss rate for writes to pixel: 25 %

C. What fraction of the writes in the following code will result in a miss in the direct mapped cache?

```
for (i = 0; i < 16; i ++){
    for (j = 0; j < 16; j ++){
        pixel[j][i].r = 0;
        pixel[j][i].g = 0;
        pixel[j][i].b = 0;
        pixel[j][i].a = 0;
    }
}
```

Miss rate for writes to pixel: 25 %

D. Using code in part C, what fraction of the writes will result in a miss in the 4-way associative cache?

Miss rate for writes to pixel: 25 %