Problem 1. (12 points):

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

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

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             j×4.
        leal 0(,%eax,8),%edx         8×i
        subl %eax,%edx             7i
        leal (%eax,%eax,4),%eax        4i+i → 5i.
        movl mat2(%ecx,%eax,4),%eax    (4×5i+4j.)    4(5i+j).
        addl mat1(%ecx,%edx,4),%eax    4×7i + 4j.
        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				
0				4								12.				16				20				24				28.			

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.    __A___ NodeTree[i]-> i

II.    __C___ NodeTree[i]-> l -> l -> c

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

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

-2 -2

D

none

Linux/IA32 Assembly:

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

4i

(eax+4i)

(eax+4i)+16

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

4i.

eax+4i.

24+4i

24+4i+20
+20

```
C:        sall $2,%edx
          leal (%eax,%edx),%eax
          movl 20(%eax),%eax
          movl 20(%eax),%eax
          movsbl (%eax),%eax
```

```
D:        sall $2,%edx
          leal (%eax,%edx),%eax
          movl (%eax),%eax
          movl 16(%eax),%eax
```

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

```
F:        sall $2, %edx
          leal (%eax,%edx),%eax
          movl (%eax),%eax
          movl 12(%eax),%eax
          movl 12(%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];
    double d[2];
    short s[3];
    unsigned u;
    int i;
} uni1;
```

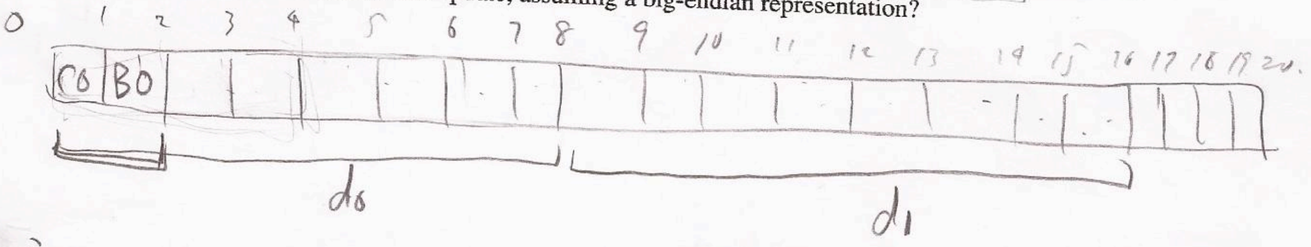
A. How much memory space in bytes would need to be allocated for uni1?

since char c [20] : 20 bytes double d [2] 16 bytes

short s[3] 6 bytes. U : 4 bytes., i 4 bytes.

Thus it needs 20 bytes for uni1.

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?



since d[0] was $-5.5 = M \times 2^E \times (-1)$

$E = 2$. $M = \frac{11}{8}$

for ~~sh~~ s[0]. 0X C0B0 ⇒ 0X 40B0.

There fore, the fist byte charge from C0 → 40.

Thus the only differen is the first bit.

Thus d[0] ⇒ $\boxed{5.5}$

4 2 1

1 0 0

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);
}
```

2

func1 2.

3 +1 = 4.

(i×2)+1

(i+1)+1

i+2.

temp = $\boxed{i+1}$

return temp+1

(i×2) +1

returnnn temp+2

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

✓ Sime fp-a and fp-b ~~,~~ meny be identical, therefore, in create func pointer () memony aliasing can occur.

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

-2 If no memeny aliasing occured, fp-b points to func 2, and fp-a points to func 1 or func 2 depeds on the value of a, so compiler will return (i+1)+1 for a > 0 and (i×2)+1 for a < 0

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

✓ If memory aliasing occurred, fp-b and fb-a will alway point to func=1 and return i+2 as always, Even though the correct answer may be (i×2)+1

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?

The update will lead to a greater degree of compiler optimization since the function calls are fixed, and compiler can simply return (arg +1) × 2.

-4

Performance Optimization

Problem 5. (14 points): 10

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;
                 2
  for (i = n; i > 0; i--)
    result = result * i;
                      a
  return result;        2
}
```

3 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.

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

Sinve the recursive versim would be like

```
int   fact (int n)
{  if (n ≤ 1)
     return 1
   return n x fac(n-1);
   }.
```

which has a bad locality, and redundant procedure calls.

The processor needs to read value from a lower level cache and operate instruction such as pop, ret and push.

But for iterativer version, it has a nice stride-1 pattern, with good spaial and temporal locality. Thus it's faster.

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;
}
```

0 1 1 2

4

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, sime it will be decrement by 2 Every time, so when n is reduced to 1, it will still be passed in loop, and (i-1) term will be 0, which cause 0 return value.

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)

change

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 _X_ _2_;
}
```

5

2 3 5 2

5×5 3

2×1=2.

1×2

2 2

5×4×3×2.

2

(<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:` y	`.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).

since the int multiplicatin latency is 4 and issue time is 1. and since in loop 1, Y is predetermined outside the loop, so that pipeline processors can carry out multiplication every clock cycle, but for loop 2, X is updated every time after multiplicatim, thus processors has to finish the whole multiplicatio before starts the next.

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 1 is optimized because every multiplication does not depend on each other. Therefore loop unrolling can make presser to operater more instruction at the same time, which speeds up Loop 1.

3

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

the loop 2 is unchanged since in X = X x a[i] X is depend on previous X, so that the processor has to know the previous x before execute a new multiplication. Thus even with unrolling, it will not help to speed up loop 2.

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:

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

2^3

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 48 56 + 1=89 11

128 64 32 16 8 4 2 1

A. Physical address format (one bit per box) 8 4 2 1

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

2.1

B. Physical memory reference

Parameter	Value
Cache Offset (CO)	0x 0
Cache Index (CI)	0x 11 -1
Cache Tag (CT)	0x 3B
Cache Hit? (Y/N)	Y
Cache Byte returned	0x 66

0x3

16x16 = 256 x 4 = ~~20~~ 10.24

4 (8)

128x8

16

6

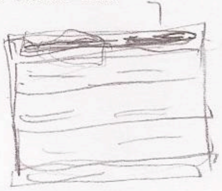
24

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.

16x8 = 2?6 7.

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;
```

4 bytes

D 128 by b=3

128 = 1 x 8 x S.

42 / 7|128 42

b=2.

A 32 / 4|128/4

16 / 8|128 / 8 16

Also assume that

S=3 b=2

S=4 b=3.

16x4=64

4|128

- `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

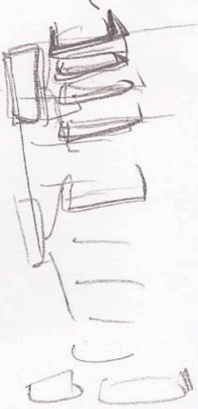
0 8

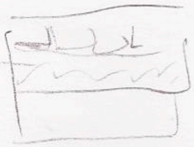
32

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

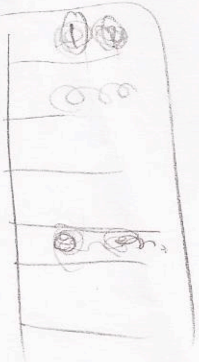
```
for (i = 0; i < 16; i ++){
    for (j = 0; j < 16; j ++){
        pixel[i][j].r = 0;
        pixel[i][j].g = 0;
        pixel[i][j].b = 0;
        pixel[i][j].a = 0;
    }
}
```

32





4



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;
     }
}
```

$\frac{128}{8} = 16$ $\frac{64}{8} = 8$

16

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 %