Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Fall 2001

Exam 1

October 9, 2001

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 64 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. You may use any books or notes you like. You may use a calculator, but no laptops or other wireless devices. Good luck!

1 (09):	
2 (12):	
3 (03):	
4 (03):	
5 (08):	
6 (08):	
7 (09):	
8 (12):	
TOTAL (64):	

Problem 1. (9 points):

Assume we are running code on a 6-bit machine using two's complement arithmetic for signed integers. A "short" integer is encoded using 3 bits. Fill in the empty boxes in the table below. The following definitions are used in the table:

```
short sy = -3;
int y = sy;
int x = -17;
unsigned ux = x;
```

Note: You need not fill in entries marked with "–".

Expression	Decimal Representation	Binary Representation
Zero	0	
–	-6	
–		`01 0010`
ux		
y		
$x >> 1$		
TMax		
$-$TMin		
TMin $+$ TMin		

Problem 2. (12 points):

Consider the following 8-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next 3 bits are the exponent. The exponent bias is $2^{3-1} - 1 = 3$.
- The last 4 bits are the fraction.
- The representation encodes numbers of the form: $V = (-1)^s \times M \times 2^E$, where M is the significand and E is the biased exponent.

The rules are like those in the IEEE standard(normalized, denormalized, representation of 0, infinity, and NAN). FILL in the table below. Here are the instructions for each field:

- **Binary:** The 8 bit binary representation.
- **M:** The value of the significand. This should be a number of the form x or $\frac{x}{y}$, where x is an integer, and y is an integral power of 2. Examples include 0, $\frac{3}{4}$.
- **E:** The integer value of the exponent.
- **Value:**The numeric value represented.

  Note: you need not fill in entries marked with ”—”.

Description	Binary	M	E	Value
Minus zero				-0.0
—	`0 100 0101`			
Smallest denormalized (negative)				
Largest normalized (positive)				
One				1.0
—				5.5
Positive infinity		—	—	$+\infty$

Problem 3. (3 points):

Consider the following C functions and assembly code:

```
int fun7(int a)
{
    return a * 30;
}

int fun8(int a)
{
    return a * 34;
}

int fun9(int a)
{
    return a * 18;
}
```

```
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%eax
sall $4,%eax
addl 8(%ebp),%eax
addl %eax,%eax
movl %ebp,%esp
popl %ebp
ret
```

Which of the functions compiled into the assembly code shown?

Problem 4. (3 points):

Consider the following C functions and assembly code:

```
int fun4(int *ap, int *bp)
{
    int a = *ap;
    int b = *bp;
    return a+b;
}

int fun5(int *ap, int *bp)
{
    int b = *bp;
    *bp += *ap;
    return b;
}

int fun6(int *ap, int *bp)
{
    int a = *ap;
    *bp += *ap;
    return a;
}
```

```
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%edx
movl 12(%ebp),%eax
movl %ebp,%esp
movl (%edx),%edx
addl %edx,(%eax)
movl %edx,%eax
popl %ebp
ret
```

Which of the functions compiled into the assembly code shown?

Problem 5. (8 points):

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

```
int array1[M][N];
int array2[N][M];

int copy(int i, int j)
{
    array1[i][j] = array2[j][i];
}
```

Suppose the above code generates the following assembly code:

```
copy:
  pushl %ebp
  movl %esp,%ebp
  pushl %ebx
  movl 8(%ebp),%ecx
  movl 12(%ebp),%ebx
  leal (%ecx,%ecx,8),%edx
  sall $2,%edx
  movl %ebx,%eax
  sall $4,%eax
  subl %ebx,%eax
  sall $2,%eax
  movl array2(%eax,%ecx,4),%eax
  movl %eax,array1(%edx,%ebx,4)
  popl %ebx
  movl %ebp,%esp
  popl %ebp
  ret
```

What are the values of M and N?

M =

N =

Problem 6. (8 points):

Consider the following assembly representation of a function `foo` containing a `for` loop:

```
foo:
  pushl %ebp
  movl %esp,%ebp
  pushl %ebx
  movl 8(%ebp),%ebx
  leal 2(%ebx),%edx
  xorl %ecx,%ecx
  cmpl %ebx,%ecx
  jge .L4
.L6:
  leal 5(%ecx,%edx),%edx
  leal 3(%ecx),%eax
  imull %eax,%edx
  incl %ecx
  cmpl %ebx,%ecx
  jl .L6
.L4:
  movl %edx,%eax
  popl %ebx
  movl %ebp,%esp
  popl %ebp
  ret
```

Fill in the blanks to provide the functionality of the loop:

```
int foo(int a)
{
    int i;
    int result = _____________;

    for( ________; ________; i++ ) {

        __________________;

        __________________;

    }
    return result;
}
```

Problem 7. (9 points):

Consider the following C declarations:

```
typedef struct {
    short code;
    long start;
    char raw[3];
    double data;
} OldSensorData;

typedef struct {
    short code;
    short start;
    char raw[5];
    short sense;
    short ext;
    double data;
} NewSensorData;
```

A. Using the templates below (allowing a maximum of 24 bytes), indicate the allocation of data for structs of type `OldSensorData` `NewSensorData`. Mark off and label the areas for each individual element (arrays may be labeled as a single element). **Cross hatch the parts that are allocated, but not used (to satisfy alignment).**

Assume the Linux alignment rules discussed in class. **Clearly indicate the right hand boundary of the data structure with a vertical line.**

`OldSensorData:`

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

`NewSensorData:`

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. Now consider the following C code fragment:

```
void foo(OldSensorData *oldData)
{
    NewSensorData *newData;

    /* this zeros out all the space allocated for oldData */
    bzero((void *)oldData, sizeof(oldData));

    oldData->code = 0x104f;
    oldData->start = 0x80501ab8;
    oldData->raw[0] = 0xe1;
    oldData->raw[1] = 0xe2;
    oldData->raw[2] = 0x8f;
    oldData->raw[-5] = 0xff;
    oldData->data = 1.5;

    newData = (NewSensorData *) oldData;

    ...
```

Once this code has run, we begin to access the elements of newData. Below, give the value of each element of `newData` that is listed. Assume that this code is run on a Little-Endian machine such as a Linux/x86 machine. You must give your answer in hexadecimal format. **Be careful about byte ordering!**.

(a) `newData->start  = 0x`__________________

(b) `newData->raw[0] = 0x`__________________

(c) `newData->raw[2] = 0x`__________________

(d) `newData->raw[4] = 0x`__________________

(e) `newData->sense  = 0x`__________________

The next problem concerns the following C code. This program reads a string on standard input and prints an integer in hexadecimal format based on the input string it read.

```
#include <stdio.h>

/* Read a string from stdin into buf */
int evil_read_string()
{
    int buf[2];

    scanf("%s",buf);
    return buf[1];
}

int main()
{
    printf("0x%x\n", evil_read_string());
}
```

Here is the corresponding machine code on a Linux/x86 machine:

```
08048414 <evil_read_string>:
 8048414:  55                     push   %ebp
 8048415:  89 e5                  mov    %esp,%ebp
 8048417:  83 ec 14               sub    $0x14,%esp
 804841a:  53                     push   %ebx
 804841b:  83 c4 f8               add    $0xfffffff8,%esp
 804841e:  8d 5d f8               lea    0xfffffff8(%ebp),%ebx
 8048421:  53                     push   %ebx                 address arg for scanf
 8048422:  68 b8 84 04 08         push   $0x80484b8           format string for scanf
 8048427:  e8 e0 fe ff ff         call   804830c <_init+0x50>  call scanf
 804842c:  8b 43 04               mov    0x4(%ebx),%eax
 804842f:  8b 5d e8               mov    0xffffffe8(%ebp),%ebx
 8048432:  89 ec                  mov    %ebp,%esp
 8048434:  5d                     pop    %ebp
 8048435:  c3                     ret

08048438 <main>:
 8048438:  55                     push   %ebp
 8048439:  89 e5                  mov    %esp,%ebp
 804843b:  83 ec 08               sub    $0x8,%esp
 804843e:  83 c4 f8               add    $0xfffffff8,%esp
 8048441:  e8 ce ff ff ff         call   8048414 <evil_read_string>
 8048446:  50                     push   %eax                 integer arg for printf
 8048447:  68 bb 84 04 08         push   $0x80484bb           format string for printf
 804844c:  e8 eb fe ff ff         call   804833c <_init+0x80>  call printf
 8048451:  89 ec                  mov    %ebp,%esp
 8048453:  5d                     pop    %ebp
 8048454:  c3                     ret
```

Problem 8. (12 points):

This problem tests your understanding of the stack discipline and byte ordering. Here are some notes to help you work the problem:

- `scanf("%s", buf)` reads an input string from the standard input stream (stdin) and stores it at address `buf` (including the terminating '`\0`' character). It does **not** check the size of the destination buffer.
- `printf("0x%x", i)` prints the integer i in hexadecimal format preceded by "`0x`".
- Recall that Linux/x86 machines are Little Endian.
- You will need to know the hex values of the following characters:

Character	Hex value	Character	Hex value
`'d'`	`0x64`	`'v'`	`0x76`
`'r'`	`0x72`	`'i'`	`0x69`
`'.'`	`0x2e`	`'l'`	`0x6c`
`'e'`	`0x65`	`'\0'`	`0x00`
		`'s'`	`0x73`

A. Suppose we run this program on a Linux/x86 machine, and give it the string "`dr.evil`" as input on stdin.

Here is a template for the stack, showing the locations of `buf[0]` and `buf[1]`. Fill in the value of `buf[1]` (in hexadecimal) and indicate where `ebp` points just **after** `scanf` returns to `evil_read_string`.

```
               |<- buf[0]->|<-buf[1] ->|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

What is the 4-byte integer (in hex) printed by the `printf` inside main?

`0x`________________________

B. Suppose now we give it the input “`dr.evil.lives`” (again on a Linux/x86 machine).

(a) List the contents of the following memory locations just **after** `scanf` returns to `evil_read_string`. Each answer should be an unsigned 4-byte integer expressed as 8 hex digits.

`buf[0] = 0x`______________________

`buf[3] = 0x`______________________

(b) Immediately **before** the `ret` instruction at address `0x08048435` executes, what is the value of the frame pointer register `%ebp`?

`%ebp  = 0x`______________________

You can use the following template of the stack as *scratch space*. *Note:* this does **not** have to be filled out to receive full credit.

```
               <- buf[0] -><- buf[1] ->
--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
```

Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Fall 2002

Exam 1

October 8, 2002

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 66 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. You may use any books or notes you like. You may use a calculator, but no laptops or other wireless devices. Good luck!

1 (06):
2 (16):
3 (08):
4 (12):
5 (06):
6 (12):
7 (06):
TOTAL (66):

Problem 1. (6 points):

Assume we are running code on a 5-bit machine using two's complement arithmetic for signed integers. Fill in the empty boxes in the table below. The following definitions are used in the table:

```
int y = -9;
unsigned z = y;
```

Note: You need not fill in entries marked with "–".

Expression	Decimal Representation	Binary Representation
Zero	0	
–	-5	
–		`1 0010`
y		
z		
$y - z$		
TMax		
TMin		

Problem 2. (16 points):

Consider the following 10-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next $k = 4$ bits are the exponent. The exponent bias is 7.
- The last $n = 5$ bits are the significand.

Numeric values are encoded in this format as a value of the form $V = (-1)^s \times M \times 2^E$, where s is the sign bit, E is exponent after biasing, and M is the significand.

Part I

Answer the following problems using either decimal (e.g., 1.375) or fractional (e.g., $11/8$) representations for numbers that are not integers.

A. For denormalized numbers:

(a) What is the value E of the exponent after biasing? __________

(b) What is the largest value M of the significand? __________

B. For normalized numbers:

(a) What is the smallest value E of the exponent after biasing? __________

(b) What is the largest value E of the exponent after biasing? __________

(c) What is the largest value M of the significand? __________

Part II

Fill in the blank entries in the following table giving the encodings for some interesting numbers.

Description	E	M	V	Binary Encoding
Zero		0	0	`0 0000 00000`
Smallest Positive (nonzero)				
Largest denormalized				
Smallest positive normalized				
One			1	
Largest odd integer				
Largest finite number				
Infinity	—	—	$+\infty$	

Problem 3. (8 points):

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

```
int array1[M][N];
int array2[N][M];

int copy(int i, int j)
{
    array1[i][j] = array2[j][i];
}
```

Suppose the above code generates the following assembly code:

```
copy:
  pushl %ebp
  movl %esp,%ebp
  pushl %ebx
  movl 8(%ebp),%ecx
  movl 12(%ebp),%ebx
  leal (%ecx,%ecx,8),%edx
  sall $2,%edx
  leal (%ebx,%ebx,2),%eax
  sall $2,%eax
  movl array2(%eax,%ecx,4),%eax
  movl %eax,array1(%edx,%ebx,4)
  popl %ebx
  movl %ebp,%esp
  popl %ebp
  ret
```

What are the values of M and N?

M =

N =

Problem 4. (12 points):

Consider the following C declarations:

```
typedef struct {
  char           name[5];
  unsigned short type;
  int            model;
  char           color;
  double         price;
} Product_Struct1;

typedef struct {
  char           *name;
  unsigned short type;
  char           color;
  unsigned short model;
  float          price;
} Product_Struct2;

typedef union {
  unsigned int      product_id;
  Product_Struct1   one;
  Product_Struct2   two;
} Product_Union;
```

A. Using the templates below (allowing a maximum of 24 bytes), indicate the allocation of data for structs of type `Product_Struct1` and `Product_Struct2`. Mark off and label the areas for each individual element (arrays may be labeled as a single element). **Cross hatch the parts that are allocated, but not used. Assume the Linux alignment rules discussed in class.**

`Product_Struct1:`

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

`Product_Struct2:`

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. How many bytes are allocated for objects of type `Product_Struct1`, `Product_Struct2` and `Product_Union`, respectively?

(a) `sizeof(Product_Struct1) =` ________________

(b) `sizeof(Product_Struct2) =` ________________

(c) `sizeof(Product_Union)   =` ________________

C. Now consider the following C code fragment:

```
void init(Product_Union *p)
{
    /* This will zero all the space allocated for *p */
    bzero((void *)p, sizeof(Product_Union));

    p->one.type = 0xbeef;
    p->one.model = 0x10302ace;
    p->one.color = 0x8a;
    p->one.price = 1.25;
    strcpy (p->one.name, "abcdef");
    /* ’a’ = 0x61  ’b’ = 0x62  ’c’ = 0x63
       ’d’ = 0x64  ’e’ = 0x65  ’f’ = 0x66 */
```

After this code has run, please give the value of each element of `Product_Union` listed below. Assume that this code is run on a Little-Endian machine such as a Linux/x86 machine. You must give your answer in hexadecimal format. **Be careful about byte ordering!**

(a) `p->product_id = 0x`________________

(b) `p->two.name   = 0x`________________

(c) `p->two.type   = 0x`________________

(d) `p->two.color  = 0x`________________

(e) `p->two.model  = 0x`________________

Problem 5. (6 points):

This problem tests your ability of matching assembly code to the corresponding C pointer code. Note that some of the C code below doesn’t do anything useful.

```
int fun4(int ap, int bp)
{
    int a = ap;
    int b = bp;
    return *(&a + b);
}

int fun5(int *ap, int bp)
{
    int *a = ap;
    int b = bp;
    return *(a + b);
}

int fun6(int ap, int *bp)
{
    int a = ap;
    int b = *bp;
    return *(&a + b);
}
```

```
pushl %ebp
movl %esp,%ebp
subl $24,%esp
movl 12(%ebp),%edx
movl 8(%ebp),%eax
movl %eax,-4(%ebp)
movl (%edx),%eax
sall $2,%eax
movl -4(%eax,%ebp),%eax
movl %ebp,%esp
popl %ebp
ret
```

Which of the functions compiled into the assembly code shown?

A) `fun4` **B)** `fun5` **C)** `fun6`

Problem 6. (12 points):

This problem tests your understanding of the stack discipline and byte ordering. Consider the following C functions and assembly code:

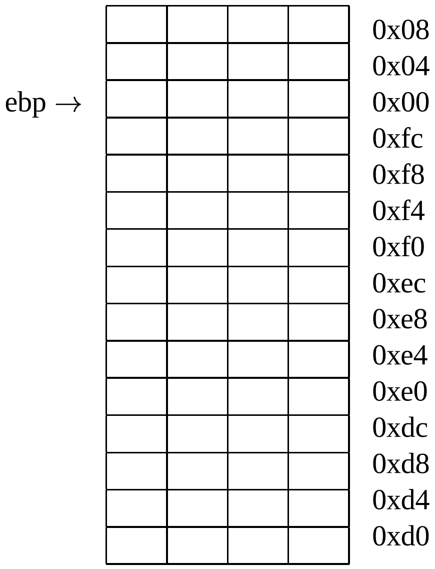
```
void check_password()
{
  char buf[8];
  scanf("%s", buf);
  if(0 != string_compare(buf, "Biggles"))
  {
    exit(1);
  }
}

int main()
{
  printf("Enter your password: ");
  check_password();
  printf("Welcome to my evil lair!\n");
  return 0;
}
```

```
80484ac <check_password>:
80484ac:    55                   push   %ebp
80484ad:    89 e5                mov    %esp,%ebp
80484af:    83 ec 24             sub    $0x24,%esp
80484b2:    53                   push   %ebx
80484b3:    83 c4 f8             add    $0xfffffff8,%esp
80484b6:    8d 5d f8             lea    0xfffffff8(%ebp),%ebx
80484b9:    53                   push   %ebx
80484ba:    68 78 85 04 08       push   $0x8048578
80484bf:    e8 a0 fe ff ff       call   8048364 <scanf>
80484c4:    83 c4 f8             add    $0xfffffff8,%esp
80484c7:    68 7b 85 04 08       push   $0x804857b
80484cc:    53                   push   %ebx
80484cd:    e8 be ff ff ff       call   8048490 <string_compare>
80484d2:    83 c4 20             add    $0x20,%esp
80484d5:    85 c0                test   %eax,%eax
80484d7:    74 0a                je     80484e3 <check_password+0x37>
80484d9:    83 c4 f4             add    $0xfffffff4,%esp
80484dc:    6a 01                push   $0x1
80484de:    e8 c1 fe ff ff       call   80483a4 <exit>
80484e3:    8b 5d d8             mov    0xffffffd8(%ebp),%ebx
80484e6:    89 ec                mov    %ebp,%esp
80484e8:    5d                   pop    %ebp
80484e9:    c3                   ret
```

Here are some notes to help you work the problem:

- `scanf("%s", buf)` reads an input string from the standard input stream (stdin) and stores it at address `buf` (including the terminating `\0` character). It does **not** check the size of the destination buffer.
- `string_compare(s1, s2)` returns `0` if `s1` equals `s2`.
- `exit(1)` halts execution of the current process without returning.
- Recall that Linux/x86 machines are Little Endian.

You may find the following diagram helpful to work out your answers. However, when grading we will **not** consider anything that you write in it.



A. **Circle the address** (relative to `ebp`) of the following items. Assume that the code has just finished executing the prolog for `check_password` (through the `push` instruction at `0x80484b2`).

return address:	0x08	0x04	0x00	0xfc	0xf8	0xf4	0xf0	...	0xdc	0xd8	0xd4	0xd0
saved `%ebp`:	0x08	0x04	0x00	0xfc	0xf8	0xf4	0xf0	...	0xdc	0xd8	0xd4	0xd0
`&buf`:	0x08	0x04	0x00	0xfc	0xf8	0xf4	0xf0	...	0xdc	0xd8	0xd4	0xd0
saved `%ebx`:	0x08	0x04	0x00	0xfc	0xf8	0xf4	0xf0	...	0xdc	0xd8	0xd4	0xd0
`%esp`:	0x08	0x04	0x00	0xfc	0xf8	0xf4	0xf0	...	0xdc	0xd8	0xd4	0xd0

B. Let us enter the string “`Bigglesworth`” (not including the quotes) as a password. Inside the `check_password` function `scanf` will read this string from stdin, writing it value into `buf`. Afterwards what will be the value in the 4-byte word pointed to by `%ebp`? You should answer in hexadecimal notation.

The following table shows the hexadecimal value for relevant ASCII characters.

Character	Hex value	Character	Hex value
'B'	0x42	'i'	0x69
'g'	0x67	'l'	0x6c
'e'	0x65	's'	0x73
'w'	0x77	'o'	0x6f
'r'	0x72	't'	0x74
'h'	0x68	\0	0x00

`(%ebp) = 0x`________________________

C. The `push` instruction at `0x80484b2` saves the value of the callee-save register `%ebx` on the stack. Give the address of the instruction that restores the value of `%ebx`. You should answer in hexadecimal notation.

`0x`________________________

Problem 7. (6 points):

This problem tests your understanding of how `for` loops in C relate to IA32 machine code. Consider the following IA32 assembly code for a procedure `foo()`:

```
foo:
        pushl %ebp
        movl %esp,%ebp
        movl 16(%ebp),%ecx
        movl 12(%ebp),%eax
        movl 8(%ebp),%edx
        cmpl %ecx,%edx
        jl .L19
.L21:
        addl %edx,%eax
        decl %edx
        cmpl %ecx,%edx
        jge .L21
.L19:
        movl %ebp,%esp
        popl %ebp
        ret
```

Based on the assembly code, fill in the blanks below in its corresponding C source code. (Note: you may only use symbolic variables x, y, z, i, and $result$, from the source code in your expressions below —do *not* use register names.)

```
int foo(int x, int y, int z)
{
  int i, result;

  result = ________;

  for (i = ________; ______________; ________)  {

       result = __________________;
     }
  }
  return result;
}
```

Andrew login ID:_____________________________________

Full Name:_____________________________________

CS 15-213, Fall 2003

Exam 1

October 7, 2003

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 55 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. You may use any books or notes you like. You may use a calculator, but no laptops or other wireless devices. Good luck!

1 (09):
2 (10):
3 (04):
4 (08):
5 (06):
6 (06):
7 (08):
8 (04):
TOTAL (55):

Problem 1. (9 points):

For this problem, assume the following:

- We are running code on a 6-bit machine using two's complement arithmetic for signed integers.
- `short` integers are encoded using 3 bits.
- Sign extension is performed whenever a `short` is casted to an `int`
- Right shifts `int`s are arithmetic.

Fill in the empty boxes in the table below. The following definitions are used in the table:

```
short sy = -3;
int y = sy;
int x = -17;
unsigned ux = x;
```

Note: You need not fill in entries marked with "–".

Expression	Decimal Representation	Binary Representation
Zero	0	
–	-6	
–		`01 0010`
ux		
y		
$x >> 1$		
TMax		
$-$TMin		
TMin + TMin		

Problem 2. (10 points):

Consider the following 12-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next $k = 4$ bits are the exponent. The exponent bias is 7.
- The last $n = 7$ bits are the significand.

Numeric values are encoded in this format as a value of the form $V = (-1)^s \times M \times 2^E$, where s is the sign bit, E is exponent after biasing, and M is the significand.

Part I

How many FP numbers are in the following intervals $[a, b)$?

For each interval $[a, b)$, count the number of x such that $a \leq x < b$.

A. Interval $[1, 2)$: __________

B. Interval $[2, 3)$: __________

Part II

Answer the following problems using either decimal (e.g., 1.375) or fractional (e.g., $11/8$) representations for numbers that are not integers.

A. For denormalized numbers:

(a) What is the value E of the exponent after biasing? __________

(b) What is the largest value M of the significand? __________

B. For normalized numbers:

(a) What is the smallest value E of the exponent after biasing? __________

(b) What is the largest value E of the exponent after biasing? __________

(c) What is the largest value M of the significand? __________

Part II

Fill in the blank entries in the following table giving the encodings for some interesting numbers.

Description	E	M	V	Binary Encoding
Zero		0	0	`0 0000 0000000`
Smallest Positive (nonzero)				
Largest denormalized				
Smallest positive normalized				

Problem 3. (4 points):

Consider the following C functions and assembly code:

```
int fun4(int *ap, int *bp)
{
    int a = *ap;
    int b = *bp;
    return a+b;
}

int fun5(int *ap, int *bp)
{
    int b = *bp;
    *bp += *ap;
    return b;
}

int fun6(int *ap, int *bp)
{
    int a = *ap;
    *bp += *ap;
    return a;
}
```

```
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%edx
movl 12(%ebp),%eax
movl %ebp,%esp
movl (%edx),%edx
addl %edx,(%eax)
movl %edx,%eax
popl %ebp
ret
```

Which of the functions compiled into the assembly code shown? ____________

Problem 4. (8 points):

Consider the following four C and IA32 functions. Next to each of the four IA32 functions, write the name of the C function that it implemnts. If the assembly routine doesn't match any of the above functions, write NONE. To save space, the startup code for each IA32 function is omitted:

```
   pushl %ebp
   movl %esp,%ebp
```

```
int winter(int foo[8][12],
           int i, int j)
{
  return foo[i][j];
}
```

```
int *spring(int foo[8][12],
            int i, int j)
{
  return foo[i];
}
```

```
int summer(int** foo,
           int i, int j)
{
  return foo[i][j];
}
```

```
int *fall(int** foo,
          int i, int j)
{
  return foo[i];
}
```

```
movl 8(%ebp),%edx
movl 12(%ebp),%eax
movl %ebp,%esp
popl %ebp
movl (%edx,%eax,4),%eax
ret
```

```
movl 8(%ebp),%ecx
movl 12(%ebp),%eax
movl 16(%ebp),%edx
movl (%ecx,%eax,4),%eax
movl %ebp,%esp
popl %ebp
movl (%eax,%edx,4),%eax
ret
```

```
movl 12(%ebp),%eax
leal (%eax,%eax,2),%eax
sall $4,%eax
addl 8(%ebp),%eax
movl %ebp,%esp
popl %ebp
ret
```

```
movl 8(%ebp),%eax
movl 12(%ebp),%edx
movl 16(%ebp),%ecx
sall $2,%ecx
leal (%edx,%edx,2),%edx
sall $4,%edx
addl %edx,%ecx
movl %ebp,%esp
popl %ebp
movl (%eax,%ecx),%eax
ret
```

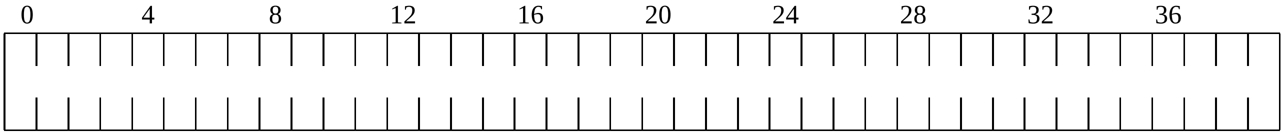
Problem 5. (6 points):

Consider the following data type definition:

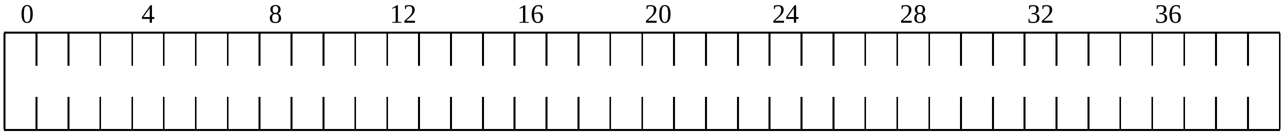
```
typedef struct {
    char c;
    double d;
    short s;
    double *pd;
    float f;
    char *pc;
} struct1;
```

Using the template below (allowing a maximum of 40 bytes), indicate the allocation of data for a structure of type `struct1`. Mark off and label the areas for each individual element (there are 6 of them). Cross hatch the parts that are allocated, but not used (to satisfy alignment). **Clearly indicate the right hand (end) boundary of the data structure with a vertical line**.

Byte Offset - Assume IA32 Windows alignment conventions



Byte Offset - Assume IA32 Linux alignment conventions



Problem 6. (6 points):

Consider the following IA32 code for a procedure called `mystery`:

```
mystery:
  pushl %ebp
  movl %esp,%ebp
  movl 12(%ebp),%edx
  movl 16(%ebp),%eax
  addl 8(%ebp),%edx
  testl %edx,%edx
  jle .L4
.L6:
  incl %eax
  decl %edx
  testl %edx,%edx
  jg .L6
.L4:
  movl %ebp,%esp
  popl %ebp
  ret
```

Based on the assembly code, fill in the blanks below in `mystery`'s C source code. (Note: you may only use symbolic variables from the source code in your expressions below - do *not* use register names.)

```
int mystery(int a, int b, int c) {
    int x, y;

    y = _____________;


    for (____________; ____________; _____________) {


         _____________;

    }


   return ___________;
}
```

The next problem concerns the following C code:

```
/* copy string x to buf */
void foo(char *x) {
  int buf[1];
  strcpy((char *)buf, x);
}

void callfoo() {
  foo("abcdefghi");
}
```

Here is the corresponding machine code on a Linux/x86 machine:

```
080484f4 <foo>:
080484f4: 55                pushl  %ebp
080484f5: 89 e5             movl   %esp,%ebp
080484f7: 83 ec 18          subl   $0x18,%esp
080484fa: 8b 45 08          movl   0x8(%ebp),%eax
080484fd: 83 c4 f8          addl   $0xfffffff8,%esp
08048500: 50                pushl  %eax
08048501: 8d 45 fc          leal   0xfffffffc(%ebp),%eax
08048504: 50                pushl  %eax
08048505: e8 ba fe ff ff    call   80483c4 <strcpy>
0804850a: 89 ec             movl   %ebp,%esp
0804850c: 5d                popl   %ebp
0804850d: c3                ret

08048510 <callfoo>:
08048510: 55                pushl  %ebp
08048511: 89 e5             movl   %esp,%ebp
08048513: 83 ec 08          subl   $0x8,%esp
08048516: 83 c4 f4          addl   $0xfffffff4,%esp
08048519: 68 9c 85 04 08    pushl  $0x804859c   # push string address
0804851e: e8 d1 ff ff ff    call   80484f4 <foo>
08048523: 89 ec             movl   %ebp,%esp
08048525: 5d                popl   %ebp
08048526: c3                ret
```

Problem 7. (8 points):

This problem tests your understanding of the stack discipline and byte ordering. Here are some notes to help you work the problem:

- `strcpy(char *dst, char *src)` copies the string at address `src` (including the terminating '`\0`' character) to address `dst`. It does **not** check the size of the destination buffer.
- Recall that Linux/x86 machines are Little Endian.
- You will need to know the hex values of the following characters:

Character	Hex value	Character	Hex value
'a'	0x61	'f'	0x66
'b'	0x62	'g'	0x67
'c'	0x63	'h'	0x68
'd'	0x64	'i'	0x69
'e'	0x65	'\0'	0x00

Now consider what happens on a Linux/x86 machine when `callfoo` calls `foo` with the input string "`abcdefghi`".

A. List the contents of the following memory locations immediately after `strcpy` returns to `foo`. Each answer should be an unsigned 4-byte integer expressed as 8 hex digits.

`buf[0] = 0x`______________________

`buf[1] = 0x`______________________

`buf[2] = 0x`______________________

B. Immediately **before** the `ret` instruction at address `0x0804850d` executes, what is the value of the frame pointer register `%ebp`?

`%ebp  = 0x`______________________

C. Immediately **after** the `ret` instruction at address `0x0804850d` executes, what is the value of the program counter register `%eip`?

`%eip  = 0x`______________________

Problem 8. (4 points):

Consider the following fragment of IA32 code taken directly from the C standard library:

```
0x400446e3: call   0x400446e8
0x400446e8: popl   %eax
```

After the `popl` instruction completes, what hex value does register `%eax` contain?

Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Fall 2004

Exam 1

Tuesday October 12, 2004

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 70 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. You may use any books or notes you like. No electronic devices are allowed. Good luck!

1 (12):	
2 (10):	
3 (15):	
4 (8):	
5 (10):	
6 (8):	
7 (7):	
TOTAL (70):	

Problem 1. (12 points):

Consider a 9-bit variant of the IEEE floating point format as follows:

- Sign bit
- 4-bit exponent with a bias of -7.
- 4-bit significand

All of the rules for IEEE (normalized, denormalized, special numbers, etc.) apply.

Fill in the numeric value represented by the following bit patterns. Write your numbers in either fractional form (e.g., $-11/8$) or decimal form (e.g., 1.375).

Bit Pattern	Numeric Value
`1 0000 0000`	
`0 0000 1111`	
`1 0001 0001`	
`0 1010 1010`	
`1 1110 1111`	
`0 1111 0000`	

Problem 2. (10 points):

You are given the following C code to compute integer absolute value:

```
int abs(int x)
{
  return x < 0 ? -x : x;
}
```

You've concerned, however, that mispredicted branches cause your machine to run slowly. So, knowing that your machine uses a two's complement representation, you try the following (recall that `sizeof(int)` returns the number of bytes in an `int`):

```
int opt_abs(int x)
{
  int mask = x >> (sizeof(int)*8-1);
  int comp = x ˆ mask;
  return comp;
}
```

A. What bit pattern does `mask` have, as a function of `x`?

B. What numeric value does `mask` have, as a function of `x`?

C. For what values of `x` do functions `abs` and `opt_abs` return identical results?

D. For the cases where they produce different results, how are the two results related?

.

E. Show that with the addition of just one single arithmetic operation (any C operation is allowed) that you can fix `opt_abs`. Show your modifications on the original code.

F. Are there any values of `x` such that `abs` returns a value that is *not* greater than 0? Which value(s)?

Problem 3. (15 points):

This question will test your ability to reconstruct C code from the assembled output. On the opposing page, there is asm code for a routine called `bunny`. It comes from a C routine with the following outline.
Don't fill in the outline yet.

```
static int bunny(int l, int r, int *A) {
    int x = ______;
    int i = ______;
    int j = ______;
    while(______) {
        do  j--;  while(______);
        do  i++;  while(______);
        if(______) {
            int t = A[i];
            A[i] = A[j];
            A[j] = t;
        }
    }
    return ______;
}
```

A. (3 points): Fill in the following table of register usage. Use the variable names from the outline. If a register gets used to store two different things, just list both of them. I've filled in two blanks to show examples. This will help you understand the code; do this before part C.

Register	Variable
`%eax`	
`%ebx`	
`%ecx`	
`%edx`	
`%esi`	
`%edi`	
`%esp`	
`%ebp`	

B. (3 points): Why does `bunny` push `%edi`, `%esi`, and `%ebx` on to the stack?

```
bunny:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %edi
        pushl   %esi
        pushl   %ebx
        movl    8(%ebp), %eax
        movl    16(%ebp), %esi
        movl    (%esi,%eax,4), %edi
        leal    -1(%eax), %ecx
        movl    12(%ebp), %ebx
        incl    %ebx
        cmpl    %ebx, %ecx
        jge     .L3
.L16:
        decl    %ebx
        cmpl    %edi, (%esi,%ebx,4)
        jg      .L16
.L7:
        incl    %ecx
        cmpl    %edi, (%esi,%ecx,4)
        jl      .L7
        cmpl    %ebx, %ecx
        jge     .L3
        movl    (%esi,%ecx,4), %edx
        movl    (%esi,%ebx,4), %eax
        movl    %eax, (%esi,%ecx,4)
        movl    %edx, (%esi,%ebx,4)
        jmp     .L16
.L3:
        movl    %ebx, %eax
        popl    %ebx
        popl    %esi
        popl    %edi
        popl    %ebp
        ret
```

C. (5 points): Fill in the blanks on the outline (on the previous page).

D. (4 points): Look at `draft_horse` and write out the control flow structure. As an example, the control flow structure of `bunny` would be:

```
bunny() {
    while() {
        while() { }
        while() { }
        if() { }
    }
    return;
}
```

I want to know about any `if`, `while`, function calls, and returns, but I don't care about anything else. Do not use `goto`.

```
draft_horse() {




}
```

E. (bragging rights): What algorithm is this code implementing?

```
draft_horse:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $28, %esp
        movl    %ebx, -12(%ebp)
        movl    %esi, -8(%ebp)
        movl    %edi, -4(%ebp)
        movl    8(%ebp), %ebx
        movl    12(%ebp), %esi
        movl    16(%ebp), %edi
        cmpl    %esi, %ebx
        jge     .L17
        movl    %edi, 8(%esp)
        movl    %esi, 4(%esp)
        movl    %ebx, (%esp)
        call    bunny
        movl    %eax, -16(%ebp)
        movl    %edi, 8(%esp)
        movl    %eax, 4(%esp)
        movl    %ebx, (%esp)
        call    draft_horse
        movl    %edi, 8(%esp)
        movl    %esi, 4(%esp)
        movl    -16(%ebp), %eax
        incl    %eax
        movl    %eax, (%esp)
        call    draft_horse
.L17:
        movl    -12(%ebp), %ebx
        movl    -8(%ebp), %esi
        movl    -4(%ebp), %edi
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Problem 4. (8 points):

Given the following code:

```
int
calcHash(char *str) {
    unsigned int i;
    int hash = 0;

    for(i = 0; i < strlen(str); i++) {
        hash += str[i] * 32 + i;

    }





    return hash;
}
```

A savvy programmer has re-written it to read as follows:

```
int
calcHash(char *str) {
    unsigned int i, len = strlen(str);
    int hashA = 0, hashB = 0;

    for(i = 0; i < len - 1; i += 2) {
        hashA += (str[i] << 5) + i;
        hashB += (str[i + 1] << 5) + i + 1;
    }

    if(i == len - 1) {
        hashA += (str[i] << 5) + i;
    }

    return hashA + hashB;
}
```

Answer the questions about this code on the following page.

A. Explain in one or two sentences how moving strlen(str) from line 6 to line 3 improves the performance of this code.

Would this transformation would preserve the exact functionality of the original code? Explain.

B. Explain in one or two sentences how creating two separate add instructions on lines 7 and 8 improves the performance of this code.

Is this an optimization that a compiler could perform? Why or why not?

C. Point out one other optimization that was added to this code and explain how it improves performance.

Problem 5. (10 points):

This problem will test your knowledge of stack discipline and byte ordering. As in Lab 3, you will perform a buffer overflow attack on the following C code. Your goal is to call `secret` and make the program execute the infinite loop.

```
int read_string() {
  char buf[8];
  scanf("%s", &buf);
  return buf[1];
}

int main() {
  printf("0x%x\n", read_string());
  return 0;
}

void secret(int arg) {
  if(arg == 0x15213)
    while(1);
  exit(-1);
}
```

Things to keep in mind while working on this problem.

- `scanf("%s", buf)` reads an input string from stdin and stores it at address `buf` (including the terminating '`\0`' character). It does **not** check the size of the destination buffer.
- Linux/x86 machines are Little Endian.

A. Suppose we gave the program the codes `69 6c 75 76 32 31 33`. Using the stack template below, indicate where `%ebp` points to, and fill in the stack with the values that were just read in **after** the call to `scanf`. **Addresses increase from left to right**.

```
     |<---------------- buf ---------------->|
+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+
```

B. Using the assembly code on the next page, fill in the stack template below with codes that will cause the program to execute `secret` and make it believe that `arg` has the value 0x15213. **Addresses increase from left to right and from top to bottom.**

```
     |<---------------- buf ---------------->|
+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+

+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+
```

C. What is the value of `%ebp` when the instruction at `0x80483e2` is executed?

```
08048398 <read_string>:
 8048398: 55                    push   %ebp
 8048399: 89 e5                 mov    %esp,%ebp
 804839b: 83 ec 18              sub    $0x18,%esp
 804839e: 8d 45 f8              lea    0xfffffff8(%ebp),%eax
 80483a1: 89 44 24 04           mov    %eax,0x4(%esp,1)
 80483a5: c7 04 24 74 84 04 08  movl   $0x8048474,(%esp,1)
 80483ac: e8 ef fe ff ff        call   80482a0 <scanf>
 80483b1: 8b 45 fc              mov    0xfffffffc(%ebp),%eax
 80483b4: c9                    leave
 80483b5: c3                    ret

080483b6 <main>:
 80483b6: 55                    push   %ebp
 80483b7: 89 e5                 mov    %esp,%ebp
 80483b9: 83 ec 08              sub    $0x8,%esp
 80483bc: 83 e4 f0              and    $0xfffffff0,%esp
 80483bf: b8 00 00 00 00        mov    $0x0,%eax
 80483c4: 29 c4                 sub    %eax,%esp
 80483c6: e8 cd ff ff ff        call   8048398 <read_string>
 80483cb: 89 44 24 04           mov    %eax,0x4(%esp,1)
 80483cf: c7 04 24 77 84 04 08  movl   $0x8048477,(%esp,1)
 80483d6: e8 e5 fe ff ff        call   80482c0 <printf>
 80483db: b8 00 00 00 00        mov    $0x0,%eax
 80483e0: c9                    leave
 80483e1: c3                    ret

080483e2 <secret>:
 80483e2: 55                    push   %ebp
 80483e3: 89 e5                 mov    %esp,%ebp
 80483e5: 83 ec 08              sub    $0x8,%esp
 80483e8: 81 7d 08 13 52 01 00  cmpl   $0x15213,0x8(%ebp)
 80483ef: 75 02                 jne    80483f3 <secret+0x11>
 80483f1: eb fe                 jmp    80483f1 <secret+0xf>
 80483f3: c7 04 24 ff ff ff ff  movl   $0xffffffff,(%esp,1)
 80483fa: e8 d1 fe ff ff        call   80482d0 <exit>
 80483ff: 90                    nop
```

Problem 6. (8 points):

Consider the following C declarations:

```
typedef union{          typedef struct {
    char state[3];          short WID;
    char cncode[4];         char name[5];
    int  index;             Location address;
} Location;                 double balance;
                            char domestic;
                            char *note;
                        } Warehouse;
```

A. Using the templates below (allowing a maximum of 28 bytes), indicate the allocation of data for structs of type `Warehouse`. Mark off and label the areas for each individual element (arrays may be labeled as a single element). **Cross hatch the parts that are allocated, but not used, and be sure to clearly indicate the end of the structure. Assume the Linux alignment rules discussed in class.**

```
Warehouse:

  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
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                                   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. How would you define the `Compact` structure to minimize the number of bytes allocated for the structure using the same fields as the `Warehouse` structure?

```
typedef struct {




 } Compact;
```

C. What is the value of `sizeof(Compact)`?

D. Now consider the IA-32 Windows alignment convention. How would you define the `Win_Compact` structure to minimize the number of bytes allocated for the structure using the same fields as the `Warehouse` structure?

```
typedef struct {





} Win_Compact;
```

E. Consider the following C code fragment:

```
Warehouse company1;

strcpy(company1.address.cncode, "CAN"); /* ’C’ = 43, ’A’ = 41, ’N’ = 4e */
```

After this code has been executed,

`company1.address.index = 0x`____________

Assume that this code is running on a little-endian machine such as a Linux/x86 machine. You must give your answer in hexadecimal format.

Problem 7. (7 points):

Answer **true** or **false** for each of the statements below. For full credit your answer must be correct and you must write the entire word (either **true** or **false** in the answer space. You will be given $+1.0$ point for each correct answer, and -0.5 points for each incorrect answer, so wild guessing doesn't pay.

1. `int a[10], x;`
   `x = &(a[5]) - &(a[1]);`
   `x` is always 4.

2. All Intel IA-32 instructions have the same length.

3. Processors with longer pipelines tend to make branch instructions more costly.

4. To swap the values of two variables in C always requires using some kind of temporary storage location.

5. In C, the variable `m` is declared as: `int m[100][100]`. Depending on the CPU architecture, the address for m[9][99] can sometimes be greater than the address for m[10][0].

6. IEEE floating point numbers are evenly distributed for values `0.5 < x < 1.0`.

7. IEEE floating point operations always round toward the nearest FP number.

Andrew login ID:_____________________________________

Full Name:_____________________________________

CS 15-213, Fall 2005

Exam 1

Tuesday October 11, 2005

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 58 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. You may use any books or notes you like. No electronic devices are allowed. Good luck!

1 (10):	
2 (12):	
3 (04):	
4 (05):	
5 (06):	
6 (05):	
7 (08):	
8 (08):	
TOTAL (58):	

Problem 1. (10 points):

Assume we are running code on a 7-bit machine using two's complement arithmetic for signed integers. Fill in the empty boxes in the table below. The following definitions are used in the table:

```
int x = -16;
unsigned uy = x;
```

- You need not fill in entries marked with "–".
- TMax denotes the largest positive two's complement number and TMin denotes the smallest negative two's complement number.
- Hint: Be careful with the promotion rules that C uses for signed and unsigned ints.

Expression	Decimal Representation	Binary Representation
–	-2	
–		`001 0011`
x		
uy		
$x - uy$		
TMax + 1		
TMin - 1		
-TMin		
TMin + TMin		
TMax + TMin		

Problem 2. (12 points):

Consider the following two 7-bit floating point representations based on the IEEE floating point format. Neither of them have sign bits—they can only represent nonnegative numbers.

1. Format A
   - There are $k = 3$ exponent bits. The exponent bias is 3.
   - There are $n = 4$ fraction bits.
2. Format B
   - There are $k = 4$ exponent bits. The exponent bias is 7.
   - There are $n = 3$ fraction bits.

Numeric values are encoded in both of these formats as a value of the form $V = M \times 2^E$, where E is exponent after biasing, and M is the significand value. The fraction bits encode the significand value M using either a denormalized (exponent field 0) or a normalized representation (exponent field nonzero).

Below, you are given some bit patterns in Format A, and your task is to convert them to the closest value in Format B. If rounding is necessary you should *round upward*. In addition, give the values of numbers given by the Format A and Format B bit patterns. Give these as whole numbers (e.g., 17) or as fractions (e.g., $17/64$).

Format A		Format B	
Bits	Value	Bits	Value
`011 0000`	1	`0111 000`	1
`101 1110`			
`010 1001`			
`110 1111`			
`000 0001`			

Problem 3. (4 points):

This problem will test your knowledge of buffer overflows. In Lab 3, you performed an overflow attack against a program that read user input. The input was read by `getbuf()` and your goal was to create an exploit string that called `smoke()`.

```
int getbuf()
{
    char buf[32];
    Gets(buf);
    return 1;
}

void smoke()
{
    printf(‘‘Smoke!: You called smoke()\n’’);
    validate(0);
    exit(0);
}
```

Creating a workable exploit string against a program like the `bufbomb` usually requires converting the executable file into human readable assembly (using `objdump`) and generating a sequence of raw, often unprintable, bytes (using a program like `hex2raw`).

However, with the `bufbomb`, you may have noticed that any 40 character string will result in `smoke()` being called.

```
unix> ./bufbomb -t ngm
Type string:It is easy to love 213 when you’re a TA.
Smoke!: You called smoke()
VALID
NICE JOB!
```

A. Why will any 40-character string result in `smoke()` being called?

The following information may help you in answering this question. Hints:

- Recall that `getbuf()` is called from `test()`.
- Also recall that C strings are always terminated by the NULL character.

```
0000000000400f66 <test>:
...
  400f72:       b8 00 00 00 00          mov    $0x0,%eax
  400f77:       e8 54 00 00 00          callq  400fd0 <getbuf>
  400f7c:       89 c2                   mov    %eax,%edx
...

0000000000400f00 <smoke>:
  400f00:       48 83 ec 08             sub    $0x8,%rsp
  400f04:       bf 1c 25 40 00          mov    $0x40251c,%edi
  400f09:       e8 fa fe ff ff          callq  400e08 <puts@plt>
  400f0e:       bf 00 00 00 00          mov    $0x0,%edi
  400f13:       e8 0c 07 00 00          callq  401624 <validate>
  400f18:       bf 00 00 00 00          mov    $0x0,%edi
  400f1d:       e8 76 fe ff ff          callq  400d98 <exit@plt>

0000000000400fd0 <getbuf>:
  400fd0:       48 83 ec 28             sub    $0x28,%rsp
  400fd4:       48 89 e7                mov    %rsp,%rdi
  400fd7:       e8 ff 00 00 00          callq  4010db <Gets>
  400fdc:       b8 01 00 00 00          mov    $0x1,%eax
  400fe1:       48 83 c4 28             add    $0x28,%rsp
  400fe5:       c3                      retq
```

Problem 4. (5 points):

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

```
int array1[L][M][N];
int array2[M][N][L];

int copy(int i, int j, int k)
{
        array1[i][j][k] = array2[j][k][i];
}
```

Suppose the above code generates the following assembly code:

```
copy:
        movslq  %edi,%rdi
        movslq  %esi,%rsi
        movslq  %edx,%rdx
        movq    %rdi, %rax
        salq    $5, %rax
        addq    %rdi, %rax
        addq    %rsi, %rax
        leaq    (%rsi,%rsi,8), %rsi
        leaq    (%rdx,%rax,2), %rax
        leaq    (%rdx,%rdx,8), %rdx
        leaq    (%rdx,%rsi,2), %rsi
        addq    %rdi, %rsi
        movl    array2(,%rsi,4), %edx
        movl    %edx, array1(,%rax,4)
        ret
```

What are the values of `L`, `M`, and `N`?

```
L =

M =

N =
```

Problem 5. (6 points):

Consider the following C function and its corresponding x86-64 assembly code:

```
int foo(int x, int i)
{
  switch(i)
  {
    case 1:
      x -= 10;
    case 2:
      x *= 8;
      break;
    case 3:
      x += 5;
    case 5:
      x /= 2;
      break;
    case 0:
      x &= 1;
    default:
      x += i;
  }
  return x;
}
```

```
00000000004004a8 <foo>:
  4004a8:       mov    %edi,%edx
  4004aa:       cmp    $0x5,%esi
  4004ad:       ja     4004d4 <foo+0x2c>
  4004af:       mov    %esi,%eax
  4004b1:       jmpq   *0x400690(,%rax,8)
  4004b8:       sub    $0xa,%edx
  4004bb:       shl    $0x3,%edx
  4004be:       jmp    4004d6 <foo+0x2e>
  4004c0:       add    $0x5,%edx
  4004c3:       mov    %edx,%eax
  4004c5:       shr    $0x1f,%eax
  4004c8:       lea    (%rdx,%rax,1),%eax
  4004cb:       mov    %eax,%edx
  4004cd:       sar    %edx
  4004cf:       jmp    4004d6 <foo+0x2e>
  4004d1:       and    $0x1,%edx
  4004d4:       add    %esi,%edx
  4004d6:       mov    %edx,%eax
  4004d8:       retq
```

Recall that the gdb command `x/g $rsp` will examine an 8-byte word starting at address in `$rsp`. Please fill in the switch jump table as printed out via the following gdb command:

```
>(gdb) x/6g 0x400690

0x400690:   0x__________________     0x__________________

0x4006a0:   0x__________________     0x__________________

0x4006b0:   0x__________________     0x__________________
```

Problem 6. (5 points):

Consider the following function's assembly code:

```
0040050a <bar>:
  40050d:       b9 00 00 00 00          mov    $0x0,%ecx
  400512:       8d 47 03                lea    0x3(%rdi),%eax
  400515:       83 ff ff                cmp    $0xffffffffffffffff,%edi
  400518:       0f 4e f8                cmovle %eax,%edi
  40051b:       89 fa                   mov    %edi,%edx
  40051d:       c1 fa 02                sar    $0x2,%edx
  400520:       85 d2                   test   %edx,%edx
  400522:       7e 14                   jle    400538 <bar2+0x2b>
  400524:       8d 42 03                lea    0x3(%rdx),%eax
  400527:       83 fa ff                cmp    $0xffffffffffffffff,%edx
  40052a:       0f 4f c2                cmovg  %edx,%eax
  40052d:       89 c2                   mov    %eax,%edx
  40052f:       c1 fa 02                sar    $0x2,%edx
  400532:       ff c1                   inc    %ecx
  400534:       85 d2                   test   %edx,%edx
  400536:       7f ec                   jg     400524 <bar2+0x17>
  400538:       89 c8                   mov    %ecx,%eax
  40053a:       c3                      retq
```

Please fill in the corresponding C code:

```
int bar(int x)
{
  int y = 0;
  int z = ________;

  for( ; ________ ; ________)
  {
    z = ________;
  }

  return ________;
}
```

Problem 7. (8 points):

Consider the following data structure declarations:

```
struct alpha {
    int array[3];
    int i;
};
```

Below are four C and four x86-64 functions. Next to each of the x86-64 functions, write the name of the C function that it implements.

```
int *jan(struct alpha *p)
{
    return &p->i;
}                                   movslq  12(%rdi),%rax
                                    leaq    (%rdi,%rax,4), %rax
                                    ret

int feb(struct alpha *p)
{
    return p->i;                    leaq    12(%rdi), %rax
}                                   ret


int mar(struct alpha *p)            movl    12(%rdi), %eax
{                                   ret
    return p->array[p->i];
}

                                    movslq  12(%rdi),%rax
                                    movl    (%rdi,%rax,4), %eax
int *apr(struct alpha *p)           ret
{
    return &p->array[p->i];
}
```

Problem 8. (8 points):

Consider the following C declarations:

```
typedef struct Order {
  char id;
  short code;
  float amount;
  char name[3];
  long data;
  char initial;
  struct Order *next;
  char address[5];
} Order;

typedef union {
  unsigned int      value;
  char              buf[20];
  Order             new_order;
} Union_1;
```

A. Using the templates below (allowing a maximum of 64 bytes), indicate the allocation of data for the Order struct `Order`. Mark off and label the areas for each individual element (arrays may be labeled as a single element). **Cross hatch the parts that are allocated, but not used. Assume the 64 bit alignment rules discussed in class.**

`Order:`

```
  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
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. How many bytes are allocated for an object of type `Union_1`?

(a) `sizeof(Union_1) =` ________________

Now consider the following C code fragment:

```
void init(Union_1 *u)
{
    /* This will zero all the space allocated for *u */
    bzero((void *)u, sizeof(Union_1));

    strcpy(u->buf, "Hello World");
    strcpy(u->new_order.name, "SM");

    printf("Output #1 is u->value                = %x", u->value);
    printf("Output #2 is u->buf                  = %s", u->buf);

    u->new_order.code = 256;

    printf("Output #3 is u->buf                  = %s", u->buf);

    /* 'H' = 0x48  'e' = 0x65  'l' = 0x6c   'o' = 0x6f
       'W' = 0x57  'r' = 0x72  'd' = 0x64   'S' = 0x63
       'M' = 0x4d  space = 0x20 */
```

After this code has run, please complete the output given below. Assume that this code is run on a Little-Endian machine such as a Linux/x86-64 machine. **Be careful about byte ordering!**

C. (a) `Output #1 is u->value                =`

(b) `Output #2 is u->buf                  =`

(c) `Output #3 is u->buf                  =`

Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Fall 2006

Exam 1

Wednesday October 4, 2006

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 56 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. You may use any books or notes you like. Calculators are allowed, but no other electronic devices. Good luck!

1 (8):	
2 (8):	
3 (8):	
4 (6):	
5 (8):	
6 (8):	
7 (10):	
TOTAL (56):	

Problem 1. (8 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 argument values, or
- Give an example where it is not true.

Puzzle	True / Counterexample
x < 0 ⇒ (x*2) < 0	False (TMin)
x > 0 ⇒ (x+1) > 0	
x > 0 ⇒ (˜x + 2) <= 0	
(x>>31) == -1 ⇒ x < 0U	
x < 0 ⇒ ((x ˆ x>>31) + 1) > 0	
((x>>31)+1) == (x>=0)	
x >= 0 ⇒ ((!x - 1) & x) == x	
((int)(ux >> 31) + ˜0) == -1	
-(x | (˜x + 1)) > 0	

Problem 2. (8 points):

Consider the following 5-bit floating point representations based on the IEEE floating point format. This format does not have a sign bit – it can only represent nonnegative numbers.

- There are $k = 3$ exponent bits. The exponent bias is 3.
- There are $n = 2$ fraction bits.

Numeric values are encoded as a value of the form $V = M \times 2^E$, where E is exponent after biasing, and M is the significand value. The fraction bits encode the significand value M using either a denormalized (exponent field 0) or a normalized representation (exponent field nonzero).

Below, you are given some decimal values, and your task it to encode them in floating point format. If rounding is necessary, you should use *round-to-even*, as you did in Lab 1 for the `float_i2f` puzzle. In addition, you should give the rounded value of the encoded floating point number. Give these as whole numbers (e.g., 17) or as fractions in reduced form (e.g., $3/4$).

Value	Floating Point Bits	Rounded value
$9/32$	`001 00`	1/4
$7/8$		
$15/16$		
9		
10		

Problem 3. (8 points):

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

```
  # On entry %edi = n
  #
0000000004004a8 <foo>:
  4004a8:   b8 00 00 00 00          mov    $0x0,%eax
  4004ad:   83 ff 01                cmp    $0x1,%edi
  4004b0:   7e 1a                   jle    4004cc <foo+0x24>
  4004b2:   01 f8                   add    %edi,%eax
  4004b4:   ba 00 00 00 00          mov    $0x0,%edx
  4004b9:   39 fa                   cmp    %edi,%edx
  4004bb:   7d 08                   jge    4004c5 <foo+0x1d>
  4004bd:   01 d0                   add    %edx,%eax
  4004bf:   ff c2                   inc    %edx
  4004c1:   39 fa                   cmp    %edi,%edx
  4004c3:   7c f8                   jl     4004bd <foo+0x15>
  4004c5:   ff cf                   dec    %edi
  4004c7:   83 ff 01                cmp    $0x1,%edi
  4004ca:   7f e6                   jg     4004b2 <foo+0xa>
  4004cc:   f3 c3                   repz retq  # treat repz as a no-op
```

Please fill in the corresponding C code:

```
int foo (int n) {
    int a, i;

    a = 0;
    for (; n > _____; _____) {
        a = a + _____;
        for (i = _____; i < _____; _____)
            a = a + _____;
    }
    return _____;
}
```

Problem 4. (6 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 following x86-64 assembly code:

```
# On entry:
#    %edi = x
#    %esi = y
#
copy_array:
    movslq  %esi,%rsi
    movslq  %edi,%rdi
    movq    %rsi, %rax
    salq    $7, %rax
    subq    %rsi, %rax
    addq    %rdi, %rax
    leaq    (%rdi,%rdi,2), %rdi
    addq    %rsi, %rdi
    movl    array1(,%rdi,4), %edx
    movl    %edx, array2(,%rax,4)
    movl    $1, %eax
    ret
```

What are the values of H and J?

H =

J =

Problem 5. (8 points):

Consider the following C declaration:

```
typedef struct WineNode {
    int vintages[3];
    double cost;
    char z;
    WineNode *next;
    short ages[5];
    int type;
    char a;
} WineNode;
```

A. Using the template below (allowing a maximum of 80 bytes), indicate the allocation of data for the struct `WineNode`. Mark off and label the areas for each element (arrays may be labeled as a single element). **Cross hatch the parts that are allocated, but not used. Clearly mark the end of the struct. Assume the 64 bit alignment rules and X86-64 data structure sizes discussed in class.**

`WineNode:`

```
  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
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. How many bytes of space in `WineNode` are wasted?_________________

C. Now rewrite the `WineNode` struct in the space provided below **so that the amount of wasted allocated space in `WineNode` is minimized.**

```
typedef struct WineNode {







} WineNode;
```

D. Now rewrite the allocation for `WineNode` as you did before using this new specification.

```
WineNode:

  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
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

E. How many bytes of space in this new `WineNode` are wasted?_________________

Problem 6. (8 points):

Consider the following data structure declarations:

```
struct node {                     struct data {
    struct data d;                    int x;
    struct node *next;                char str[6];
};                                };
```

Below are given four C functions and four x86-64 code blocks. Next to each of the x86-64 code blocks, write the name of the C function that it implements.

```
int alpha(struct node *ptr) {
    return ptr->d.x;
}
                                          movq    16(%rdi), %rax
                                          addq    $4, %rax
char *beta(struct node *ptr) {            ret
    ptr = ptr->next;
    return ptr->d.str;
}                                         movq    %rdi, %rax
                                          ret

char gamma (struct node *ptr) {
    return ptr->d.str[4];                 movl    (%rdi), %eax
}                                         ret


int *delta (struct node *ptr) {           movsbl  8(%rdi),%eax
    struct data *dp =                     ret
        (struct data *) ptr;
    return &dp->x;
}
```

Reverse Engineering Switch Code

The next problem concerns the code generated by GCC for a function involving a switch statement. Following a bounds check, the code uses a jump to index into the jump table

```
  400476:  ff 24 d5 a0 05 40 00  jmpq  *0x4005a0(,%rdx,8)
```

Using GDB, we extract the 8-entry jump table as:

```
0x4005a0: 0x0000000000400480  0x0000000000400491
0x4005b0: 0x0000000000400480  0x0000000000400496
0x4005c0: 0x0000000000400480  0x0000000000400489
0x4005d0: 0x0000000000400485  0x0000000000400496
```

The following block of disassembled code implements the branches of the switch statement

```
  400480:  48 8d 04 3f  lea    (%rdi,%rdi,1),%rax
  400484:  c3           retq
  400485:  48 0f af f7  imul   %rdi,%rsi
  400489:  48 89 f8     mov    %rdi,%rax
  40048c:  48 21 f0     and    %rsi,%rax
  40048f:  90           nop
  400490:  c3           retq
  400491:  48 8d 04 37  lea    (%rdi,%rsi,1),%rax
  400495:  c3           retq
  400496:  48 8d 46 ff  lea    0xffffffffffffffff(%rsi),%rax
  40049a:  c3           retq
```

Problem 7. (10 points):

Fill in the blank portions of the C code below to reproduce the function corresponding to this object code. You can assume that the first entry in the jump table is for the case when s equals 0. Parameters a, b, and s are passed in registers %rdi, %rsi, and %rdx, respectively.

```
long fun(long a, long b, long s)
{
    long result = 0;
    switch (s) {
    case ___:
    case ___:
        result = _____;
        break;
    case ___:
        b = _____;
        /* Fall through */
    case ___:
        result = _____;
        break;
    case ___:
        result = _____;
        break;
    default:
        result = _____;
    }
    return result;
}
```

Andrew login ID:______________________________________

Full Name:______________________________________

Recitation Section:______________________________________

CS 15-213, Fall 2008
Exam 1

Thurs. September 25, 2008

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–H) on the front.
- Write your answers in the space provided for the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 72 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. You may use any books or notes you like. No calculators or other electronic devices are allowed.
- Good luck!

1 (8):
2 (10):
3 (12):
4 (9):
5 (6):
6 (8):
7 (11):
8 (8):
TOTAL (72):

Problem 1. (8 points):

For this problem, assume the following:

- We are running code on an 8-bit machine using two's complement arithmetic for signed integers.
- `short` integers are encoded using 4 bits.
- Sign extension is performed whenever a `short` is cast to an `int`

The following definitions are used in the table below:

```
short sa = -6;
int b = 2*sa;
short sc = (short)b;
int x = -64;
unsigned ux = x;
```

Fill in the empty boxes in the table. If the expression is cast to or stored in a `short`, use a 4-bit binary representation. Otherwise assume an 8-bit binary representation. The first 2 lines are given to you as examples, and you need not fill in entries marked with "—".

Expression	Decimal Representation	Binary Representation
Zero	0	`0000 0000`
`(short)0`	0	`0000`
—	-17	
—		`0010 1001`
`sa`		
`b`		
`sc`		
`ux`		
TMax		
TMax $-$ TMin		

Problem 2. (10 points):

Assume we are using a machine where data type `int` uses a 32-bit, two's complement representation, and right shifting is performed arithmetically. Data type `float` uses a 32-bit IEEE floating-point representation.

Consider the following definitions.

```
int i = hello();
float fi = i;
```

Answer the following questions. For each C-language expression in the first column, either

1. Mark that it is TRUE of all possible values returned by function `hello()`, and *provide an explanation of why it is true*.
2. Mark that it is possibly FALSE, and provide a counter-example.

Puzzle	**True/False**	**Explanation/Counter-example**
`(i ^ ~(i >> 31)) < 0`		
`-(i | (~i + 1)) > 0`		
`i > 0 ⇒ i + (int) fi > 0`		
`fi > 0 ⇒ fi + (float) i > 0`		
`i & 1 == ((int) fi) & 1`		

Problem 3. (12 points):

Consider the following two 8-bit floating point representations based on the IEEE floating point format. Neither has a sign bit—they can only represent nonnegative numbers.

1. Format A
   - There are $k = 3$ exponent bits. The exponent bias is 3.
   - There are $n = 5$ fraction bits.
2. Format B
   - There are $k = 5$ exponent bits. The exponent bias is 15.
   - There are $n = 3$ fraction bits.

Fill in the blanks in the table below by converting the given values in each format to the closest possible value in the other format. Express values as whole numbers (e.g., 17) or as fractions (e.g., 17/64). If necessary, you should apply the round-to-even rounding rule.

Format A		Format B	
Bits	Value	Bits	Value
`011 00000`	1	`01111 000`	1
			15
	$\frac{53}{16}$		
		`10100 110`	
`000 00001`			

Problem 4. (9 points):

Consider the following x86_64 assembly code:

```
# On entry: %rdi = M, %esi = n
# Note: nopl is simply a nop instruction for alignment purposes
0000000000400500 <func>:
  400500: 85 f6                test   %esi,%esi
  400502: 7e 2a                jle    40052e <func+0x2e>
  400504: 31 c0                xor    %eax,%eax
  400506: 48 8b 0f             mov    (%rdi),%rcx
  400509: 31 d2                xor    %edx,%edx
  40050b: 0f 1f 44 00 00       nopl   0x0(%rax,%rax,1)
  400510: 44 8b 01             mov    (%rcx),%r8d
  400513: 45 85 c0             test   %r8d,%r8d
  400516: 7f 18                jg     400530 <func+0x30>
  400518: 83 c2 01             add    $0x1,%edx
  40051b: 48 83 c1 04          add    $0x4,%rcx
  40051f: 39 c2                cmp    %eax,%edx
  400521: 7e ed                jle    400510 <func+0x10>
  400523: 83 c0 01             add    $0x1,%eax
  400526: 48 83 c7 08          add    $0x8,%rdi
  40052a: 39 c6                cmp    %eax,%esi
  40052c: 7f d8                jg     400506 <func+0x6>
  40052e: 31 c0                xor    %eax,%eax
  400530: f3 c3                repz retq
```

Fill in the blanks of the corresponding C function:

```
int func(______ M, int n) {
    int i, j;
    for (i = 0; ______; i++) {
        for (j = 0; ______; j++) {
            if (________________)
                return ____;
        }
    }
    return ____;
}
```

Problem 5. (6 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 following x86-64 assembly code:

```
# On entry:
#    %edi = x
#    %esi = y
#
copy_array:
    movslq  %edi,%rdi
    movslq  %esi,%rsi
    movq    %rdi, %rax
    leaq    (%rsi,%rsi,2), %rdx
    salq    $5, %rax
    subq    %rdi, %rax
    leaq    (%rdi,%rdx,2), %rdx
    addq    %rsi, %rax
    movl    array1(,%rax,4), %eax
    movl    %eax, array2(,%rdx,4)
    movl    $1, %eax
    ret
```

What are the values of H and J?

H =

J =

Problem 6. (8 points):

Consider the following data structure declarations:

```
struct node {
  struct entry e;
  struct node *next;
}
```

```
struct entry {
  char a;
  char b;
  long c[2];
}
```

Below are given four C functions and five x86-64 code blocks.

```
char *one(struct node *ptr){
  return &(ptr->e.a)+1;
}
```

```
long two(struct node *ptr){
  return ((ptr->e.c)[0] = ptr->next);
}
```

```
char *three(struct node *ptr){
  return &(ptr->next->e.a);
}
```

```
char four(struct node *ptr){
  return ptr->e.b;
}
```

```
A | mov   0x18(%rdi), %rax
```

```
B | lea   0x18(%rdi), %rax
```

```
C | lea   0x1(%rdi), %rax
```

```
D | mov   0x18(%rdi), %rax
  | mov   %rax, 0x8(%rdi)
```

```
E | movsbl  0x1(%rdi), %rax
```

In the following table, next to the name of each C function, write the name of the x86-64 block that implements it.

Function Name	**Code Block**
`one`	
`two`	
`three`	
`four`	

Problem 7. (11 points):

The next problem concerns code generated by GCC for a function involving a switch statement. The code uses a jump to index into the jump table:

```
  400519: jmpq   *0x400640(,%rdi,8)
```

Using GDB, we extract the 8-entry jump table as:

```
0x400640:    0x0000000000400530
0x400648:    0x0000000000400529
0x400650:    0x0000000000400520
0x400658:    0x0000000000400529
0x400660:    0x0000000000400535
0x400668:    0x000000000040052a
0x400670:    0x0000000000400529
0x400678:    0x0000000000400530
```

The following block of disassembled code implements the branches of the switch statement:

```
# on entry: %rdi = a, %rsi = b, %rdx = c
  400510: mov    $0x5,%rax
  400513: cmp    $0x7,%rdi
  400517: ja     400529
  400519: jmpq   *0x400640(,%rdi,8)
  400520: mov    %rdx,%rax
  400523: add    %rsi,%rax
  400526: salq   $0x2,%rax
  400529: retq
  40052a: mov    %rsi,%rdx
  40052d: xor    $0xf,%rdx
  400530: lea    0x70(%rdx),%rax
  400534: retq
  400535: mov    $0xc,%rax
  400538: retq
```

Fill in the blank portions of C code below to reproduce the function corresponding to this object code. You can assume that the first entry in the jump table is for the case when a equals 0.

```
long test(long a, long b, long c)
{
  long answer = _____;
  switch(a)
  {
    case ___:
      c = _____;
      /* Fall through */
    case ___:
    case ___:
      answer = _____;
      break;
    case ___:
      answer = _____;
      break;
    case ___:
      answer = _____;
      break;
    default:
      answer = _____;
  }

  return answer;
}
```

Problem 8. (8 points):

```
struct {
    char *a;
    short b;
    double c;
    char d;
    float e;
    char f;
    long g;
    void *h;
} foo;
```

A. Show how the struct above would appear on a 32-bit Windows machine (primitives of size k are k-byte aligned). Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks to indicate bytes that are allocated in the struct but are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. Rearrange the above fields in `foo` to conserve the most space in the memory below. Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks to indicate bytes that are allocated in the struct but are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

C. How many bytes of the struct are wasted in part A?

D. How many bytes of the struct are wasted in part B?

Andrew login ID:______________________________________

Full Name:______________________________________

Recitation Section:______________________________________

CS 15-213, Fall 2009
Exam 1

Thursday, September 24, 2009

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–J) on the front.
- Write your answers in the space provided for the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 76 points.
- The problems are of varying difficulty. The point value of each problem is indicated (instructors reserve the right to change these values). Pile up the easy points quickly and then come back to the harder problems.
- This exam is OPEN BOOK. You may use any books or notes you like. No calculators or other electronic devices are allowed.
- QUESTIONS: If you have a question, write it (clearly) on an index card and raise your hand. We will take the card and write a reply.
- Good luck!

1 (10):	
2 (8):	
3 (12):	
4 (11):	
5 (8):	
6 (12):	
7 (9):	
8 (6):	
Extra (4):	
TOTAL (76):	

Problem 1. (10 points):

Part A

Fill in the blanks in the table below with the number described in the first column of each row. You can give your answers as unexpanded simple arithmetic expressions (such as $15^{213} + 42$); you should not have trouble fitting your answers into the space provided.

Remember that 32-bit floats have 8 bits of exponent and 23 bits of mantissa.

Description	Number
`int x=0; float *f = (float *)&x;` What is the value of `*f`?	
`int x=-1; float *f = (float *)&x;` What is the value of `*f`?	
Smallest positive, non-zero denormalized 32-bit `float`	

Part B

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 definition:

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

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 argument values, or
- Give an example where it is not true.

Puzzle	True / Counterexample
`(x >> 31) ^ ((-x) >> 31) == 0`	
`x ^ ~(x >> 31) < 0`	
`x ^ y ^ (~x) - y == y ^ x ^ (~y) - x`	
`(((!!ux)) << 31) >> 31) == (((!!x) << 31) >> 31)`	

Problem 2. (8 points):

```
struct {
    char a[9];
    short b[3];
    float c;
    char d;
    int e;
    char *f;
    short g;
} foo;
```

A. Show how the struct above would appear on a 64-bit ("x86_64") Windows machine (primitives of size k are k-byte aligned). Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks or x's to indicate bytes that are allocated in the struct but are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. Rearrange the above fields in `foo` to conserve the most space in the memory below. Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks or x's to indicate bytes that are allocated in the struct but are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

C. How many bytes are wasted in part A, inside and after the struct, if the next memory value is a pointer?

D. How many bytes are wasted in part B, inside and after the struct, if the next memory value is a pointer?

Problem 3. (12 points):

Consider the following two 8-bit floating point representations based on the IEEE floating point format. Neither has a sign bit—they can only represent nonnegative numbers.

1. Format A
   - There are $k = 4$ exponent bits. The exponent bias is 7.
   - There are $n = 4$ fraction bits.
2. Format B
   - There are $k = 6$ exponent bits. The exponent bias is 31.
   - There are $n = 2$ fraction bits.

Fill in the blanks in the table below by converting the given values in each format to the closest possible value in the other format. Express values as whole numbers (e.g., 17) or as fractions (e.g., 17/64). If necessary, you should apply the round-to-even rounding rule. If conversion would cause an overflow, follow the IEEE standard convention for representing +Infinity. You should also assume IEEE conventions for representing denormalized values.

Format A		Format B	
Bits	Value	Bits	Value
0111 0000	1	011111 00	1
			112
	$\frac{23}{32}$		
		110111 10	
0000 0101			

Problem 4. (11 points):

Your friend Harry Bovik, who hasn't taken 15-213, is in need of your help. He was writing a function to do strange arithmetic for a project of his, but accidentally deleted his source code file, and also spilled his drink across the sheet of paper with his scratch work on it, leaving him with only half-legible code and an executable file that he compiled just recently. Being the clever student that you are, you ask to see his scratchwork and executable file.

```
int foo(_____________ a)
{
    _____________ b = 0;

    switch (_____________) {
        case 0:

            b = _____________;

            _____________;
        case 1:

            b = _____________;

            _____________;
        case 2:

            b = _____________;

            _____________;
        case 3:

            b = _____________;

            _____________;
        case 4:

            b = _____________;

            _____________;
    }
    return b;
}
```

Feeding the executable to your trusty debugger, you find the following relevant information:

```
(gdb) disassemble foo
Dump of assembler code for function foo:
0x0000000000400508 <foo+0>:     mov    $0x0,%edx
0x000000000040050d <foo+5>:     lea    0x1(%rdi),%eax     # %rdi: first argument
0x0000000000400510 <foo+8>:     cmp    $0x4,%eax
0x0000000000400513 <foo+11>:    ja     0x40052c <foo+36>
0x0000000000400515 <foo+13>:    mov    %eax,%eax
0x0000000000400517 <foo+15>:    jmpq   *0x4006d0(,%rax,8)
0x000000000040051e <foo+22>:    mov    %edi,%edx
0x0000000000400520 <foo+24>:    shr    %edx
0x0000000000400522 <foo+26>:    not    %edx
0x0000000000400524 <foo+28>:    neg    %edx
0x0000000000400526 <foo+30>:    jmp    0x40052c <foo+36>
0x0000000000400528 <foo+32>:    mov    %edi,%edx
0x000000000040052a <foo+34>:    xor    %edi,%edx
0x000000000040052c <foo+36>:    mov    %edx,%eax
0x000000000040052e <foo+38>:    retq
End of assembler dump.
(gdb) x/8g 0x4006d0
0x4006d0:       0x000000000040051e      0x0000000000400522
0x4006e0:       0x0000000000400524      0x0000000000400528
0x4006f0:       0x000000000040052a      0x3b031b01000a6425
0x400700:       0x0000000400000028      0x00000044fffffe0c
```

1. Unfortunately Harry's scratch work has `break` statements hastily scribbled in and crossed out again in every case, and he can't remember which cases are supposed to have them. Using the assembly dump of his function, figure out which cases had `break`s at the end of them. (Write either "`break`" or nothing at all in the last blank of each case block.)

2. The scratch work you were handed also failed to note what types `a` and `b` are, but fortunately some of the opcodes give it away. Figure out what types Harry meant for his variables to be.

3. Using the disassembly of `foo` and the jump table you found, reconstruct the rest of the switch statement.

4. What values will `foo` return for each possible input `a`?

Problem 5. (8 points):

Consider the following data structure declarations:

```
struct node {
  unsigned uid;
  union data d;
  struct node *next;
};
```

```
union data {
  int x[3];
  long y[3];
};
```

Below are given four C functions and five x86_64 code blocks, compiled on Linux using GCC.

```
int odin(struct node *ptr) {
  return (ptr->d.x[2]);
}
```

```
unsigned dva(struct node *ptr) {
  return (ptr->uid = (long)ptr->next);
}
```

```
long tri(struct node *ptr) {
  union data *dptr =
    (union data *)ptr->next;
   return dptr->y[1];
}
```

```
long *chetyre(struct node *ptr) {
  return &ptr->next->d.y[0];
}
```

```
A | mov   0x20(%rdi),%rax
  | mov   0x8(%rax),%rax
```

```
B | mov   0x10(%rdi),%eax
```

```
C | mov   0xc(%rdi),%rax
```

```
D | mov   0x20(%rdi),%rax
  | add   $0x8,%rax
```

```
E | mov   0x20(%rdi),%rax
  | mov   %eax,(%rdi)
```

In the following table, next to the name of each C function, write the name of the x86_64 block that implements it.

Function Name	Code Block
odin	
dva	
tri	
chetyre	

Problem 6. (12 points):

Below is some assembly code to a famous algorithm. Please briefly read the code then answer the questions on the following page.

```
0000000000400498 <mystery>:
  400498:  41 b8 00 00 00 00  mov    $0x0,%r8d
  40049e:  eb 22              jmp    4004c2 <mystery+0x2a>
  4004a0:  89 c8              mov    %ecx,%eax
  4004a2:  c1 e8 1f           shr    $0x1f,%eax
  4004a5:  01 c8              add    %ecx,%eax
  4004a7:  d1 f8              sar    %eax          ; arith shift right 1 bit
  4004a9:  42 8d 0c 00        lea    (%rax,%r8,1),%ecx
  4004ad:  48 63 c1           movslq %ecx,%rax
  4004b0:  8b 04 87           mov    (%rdi,%rax,4),%eax
  4004b3:  39 d0              cmp    %edx,%eax
  4004b5:  7d 05              jge    4004bc <mystery+0x24>
  4004b7:  41 89 c8           mov    %ecx,%r8d
  4004ba:  eb 06              jmp    4004c2 <mystery+0x2a>
  4004bc:  39 d0              cmp    %edx,%eax
  4004be:  7e 10              jle    4004d0 <mystery+0x38>
  4004c0:  89 ce              mov    %ecx,%esi
  4004c2:  89 f1              mov    %esi,%ecx
  4004c4:  44 29 c1           sub    %r8d,%ecx
  4004c7:  85 c9              test   %ecx,%ecx
  4004c9:  7f d5              jg     4004a0 <mystery+0x8>
  4004cb:  b9 ff ff ff ff     mov    $0xffffffff,%ecx
  4004d0:  89 c8              mov    %ecx,%eax
  4004d2:  c3                 retq
```

a) Please write a single line of C code to represent the instruction `lea (%rax,%r8,1),%ecx` (Use C variables named rax,r8, and ecx, you can ignore types).

b) Please write a single line of C code to represent the instruction `mov (%rdi,%rax,4),%eax` (Use C variables named rdi and rax, you can ignore types).

c) Commonly found in assembly is the `leave` instruction; why is that instruction not in this code?

d) You learned about two different architectures in class, IA32 and x86_64. What architecture is this code written for and what major downside would occur from using the other architecture?

e) Now for the fun part! Please fill in the blanks in the following C code to match the assembly above.

```
int mystery (_______ * array, size_t size, int e){
    int a;

    int __________ = size;

    int __________ = 0;

    while (____________________> 0){

        a = ______________________;

        if(________________________________________){

            ________ = a;

        }else if (_______________________________){

            ________ = a;
        }else{

            return _________
        }
    }

    return __________;
}
```

f) What famous algorithm is this?

Problem 7. (9 points):

Circle the correct answer. Assume IA32 unless stated otherwise.

1. Here is a small C program:

```
struct foo { int bar; int baz; };

int get_baz(struct foo *foo_ptr)
{
    return foo_ptr->baz;
}
```

After compiling the code, disassembling `get_baz`, and adding a few comments, we get:

```
get_baz: push   %ebp            ; save old frame base pointer
         mov    %esp,%ebp       ; set frame base pointer
         mov    0x8(%ebp),%eax  ; move foo_ptr to %eax
         –Mystery Instruction Goes Here–
         leave
         ret
```

What is the *Mystery Instruction*?

(a) `mov $baz(%eax),%eax`

(b) `mov 0x4(%eax),%eax`

(c) `lea 0x4(%eax),%eax`

(d) `mov 0xc(%ebp),%eax`

2. The function `bitsy` is declared in C as

```
int bitsy(int x);
```

and the (correctly) compiled IA32 code is:

```
bitsy: push   %ebp
       mov    %esp,%ebp
       sub    $0x8,%esp
       mov    0x8(%ebp),%eax
       not    %eax
       inc    %eax
       leave
       ret
```

What is the result (denoted here by a C expression) returned by bitsy?

(a) `!(x + 1)`

(b) `*(1 - x)`

(c) `-x`

(d) `(x > 0 ? -x : -x + 1)`

3. Which of the following is true:

   (a) There are no IEEE float representations exactly equal to zero.

   (b) There is one IEEE float representation exactly equal to zero.

   (c) There are two IEEE float representations exactly equal to zero.

   (d) There are many IEEE float representations exactly equal to zero.

4. Which one of the following is true:

   (a) Denormalized floats must be normalized before a floating point computation is complete.

   (b) Denormalized floats represent magnitudes smaller than those of normalized floats.

   (c) Denormalized floats signal a computation error or an undefined result.

   (d) Denormalized floats represent magnitudes greater than those of normalized floats.

5. Why does the compiler sometimes generate `xorl %eax,%eax` rather than `movl $0x0,%eax`?

   (a) Using `xorl` allows the binary code to run on both IA32 and x86-64 architectures.

   (b) The `xorl` form is faster and/or uses fewer bytes than `movl`.

   (c) The `movl` form requires a zero to be accessed from memory location 0.

   (d) The `xorl` form stalls the processor until the the result value is stored in %eax and ready for use by the next instruction.

6. On x86-64, `addl %ebx,%eax` has the following effect:

   (a) %eax gets %eax + %ebx, high-order 32 bits of %rax are zeroed

   (b) %eax gets %eax + %ebx, %rax is unchanged

   (c) %eax gets %eax + %ebx, high-order 32 bits of %rax are sign-extended

   (d) %rax gets %eax + %ebx

7. If %esp has the value 0xBFFF0000 before a call instruction, the value immediately after the call instruction (before the first instruction of the called function) is:

   (a) 0xBFFEFFFC
   (b) 0xBFFF0004
   (c) 0xBFFF0000
   (d) The address of the instruction after the call instruction.

8. Which of the following is true:

   (a) A function can immediately clear *any* “callee save” registers.
   (b) The caller must always save *all* “caller save” registers before calling a function.
   (c) The called function must immediately save *all* “callee save” registers on the stack and restore them before returning.
   (d) A function can always ignore the initial values of *all* “caller save” registers.

9. Which of the following is true:

   (a) A 32-bit IEEE float can represent any 32-bit integer to within 0.5.
   (b) All 32-bit IEEE floats with integer values are encoded with the binary point at the rightmost bit, so E (the exponent) is 0 and exp (the 8-bit exponent field) is E + bias = 127.
   (c) No decimal integer has an exact representation in IEEE floating point (10 is not a power of 2).
   (d) There is no exact representation in IEEE floating point of most decimal fractions.

Problem 8. (6 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 following x86-64 assembly code:

```
# On entry:
#    %edi = x
#    %esi = y
#
copy_array:
  movslq  %edi,%rdi
  movslq  %esi,%rsi
  movq    %rdi, %rax
  leaq    0(,%rsi,8), %rdx
  salq    $4, %rax     ; arith shift left
  subq    %rdi, %rax
  subq    %rsi, %rdx
  addq    %rsi, %rax
  leaq    (%rdi,%rdx,4), %rdx
  movl    array1(,%rax,4), %eax
  movl    %eax, array2(,%rdx,4)
  movl    $1, %eax
  ret
```

What are the values of H and J?

H =

J =

Extra Credit (4 points)

This problem is Extra Credit; do not attempt it until you have finished all other questions on the exam. This question is based on knowledge this class does not cover, and you are not expected to know how to solve it.

This problem deals with a tricky problem with GCC when run with high levels of optimization. This code in particular is compiled with

```
$ gcc -O3 input.c
```

One of your friends who hasn't taken 213 comes to you with a program, wanting your help. They tell you that they have been debugging it for hours, finally removing all their intricate code and just putting a single printf statement inside their loop. They show you this relevant code:

```
short a = 1024;
short b;

for(b=1000;;b++){
    if(a+b < 0){
        printf("Overflow!, stopping\n");
        break;
    }
    printf("%d ",a+b);
}
```

Their code never breaks and runs in an infinite loop. You, being a 213 student of course immediately ask to see the assembly dump:

```
08048380 <main>:
 8048380: 8d 4c 24 04           lea    0x4(%esp),%ecx
 8048384: 83 e4 f0              and    $0xfffffff0,%esp
 8048387: ff 71 fc              pushl  0xfffffffc(%ecx)
 804838a: 55                    push   %ebp
 804838b: 89 e5                 mov    %esp,%ebp
 804838d: 53                    push   %ebx
 804838e: bb e8 07 00 00        mov    $0x7e8,%ebx
 8048393: 51                    push   %ecx
 8048394: 83 ec 10              sub    $0x10,%esp
 8048397: 89 5c 24 04           mov    %ebx,0x4(%esp)
 804839b: 83 c3 01              add    $0x1,%ebx
 804839e: c7 04 24 70 84 04 08  movl   $0x8048470,(%esp)
 80483a5: e8 2e ff ff ff        call   80482d8 <printf@plt>
 80483aa: eb eb                 jmp    8048397 <main+0x17>
 80483ac: 90                    nop
 80483ad: 90                    nop
 80483ae: 90                    nop
 80483af: 90                    nop
```

1. From the programmer's point of view, what is wrong with this assembly code?

2. Why do you think gcc did this? (hint: we never mentioned this in class)

3. Please write the assembly code necessary to achieve the behavior intended by the programmer, and tell us where you would insert the code.

Andrew login ID:_____________________________________

Full Name:_____________________________________

Section:_____________________________________

15-213/18-243, Fall 2010

Exam 1 - Version A

Tuesday, September 28, 2010

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew login ID, full name, and section on the front.
- This exam is closed book, closed notes, although you may use a single 8 1/2 x 11 sheet of paper with your own notes. You may not use any electronic devices.
- 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 60 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Good luck!

1 (10):	
2 (10):	
3 (6):	
4 (6):	
5 (4):	
6 (10):	
7 (14):	
TOTAL (60):	

Problem 1. (10 points):

General systems concepts. Write the correct answer for each question in the following table:

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20

1. Consider the following code, what is the output of the printf?

```
int x = 0x15213F10 >> 4;
char y = (char) x;
unsigned char z = (unsigned char) x;
printf("%d, %u", y, z);
```

   (a) -241, 15
   (b) -15, 241
   (c) -241, 241
   (d) -15, 15

2. In two's compliment, what is $-TMin$?

   (a) $Tmin$
   (b) $Tmax$
   (c) 0
   (d) -1

3. Let $int\ x = -31/8$ and $int\ y = -31 >> 3$. What are the values of x and y?

   (a) $x = -3, y = -3$
   (b) $x = -4, y = -4$
   (c) $x = -3, y = -4$
   (d) $x = -4, y = -3$

4. In C, the expression "$15213U > -1$" evaluates to:

   (a) True (1)
   (b) False (0)

5. In two's compliment, what is the minimum number of bits needed to represent the numbers -1 and the number 1 respectively?

   (a) 1 and 2
   (b) 2 and 2
   (c) 2 and 1
   (d) 1 and 1

6. Consider the following program. Assuming the user correctly types an integer into stdin, what will the program output in the end?

```
#include <stdio.h>
int main(){
    int x = 0;
    printf("Please input an integer:");
    scanf("%d",x);
    printf("%d", (!!x)<<31);
}
```

   (a) 0
   (b) $TMin$
   (c) Depends on the integer read from stdin
   (d) Segmentation fault

7. By default, on Intel x86, the stack

   (a) Is located at the bottom of memory.
   (b) Grows down towards smaller addresses
   (c) Grows up towards larger addresses
   (d) Is located in the heap

8. Which of the following registers stores the return value of functions in Intel x86_64?

   (a) %rax
   (b) %rcx
   (c) %rdx
   (d) %rip
   (e) %cr3

9. The `leave` instruction is effectively the same as which of the following:

   (a) `mov %ebp, %esp`
   `pop %ebp`
   (b) `pop %eip`
   (c) `mov %esp, %ebp`
   `pop %esp`
   (d) `ret`

10. Arguments to a function, in Intel IA32 assembly, are passed via

   (a) The stack
   (b) Registers
   (c) Physical memory
   (d) The `.text` section
   (e) A combination of the stack and registers.

11. A buffer overflow attack can only be executed against programs that use the `gets` function.

    (a) True
    (b) False

12. Intel x86_64 systems are

    (a) Little endian
    (b) Big endian
    (c) Have no endianess
    (d) Depend on the operating system

13. Please fill in the return value for the following function calls on both an Intel IA32 and Intel x86_64 system:

Function	Intel IA32	Intel x86_64
`sizeof(char)`		
`sizeof(int)`		
`sizeof(void *)`		
`sizeof(int *)`		

14. Select the two's complement negation of the following binary value: `0000101101`:

    (a) `1111010011`
    (b) `1111010010`
    (c) `1000101101`
    (d) `1111011011`

15. Which line of C-code will perform the same operation as `leal 0x10(%rax,%rcx,4),%rax`?

    (a) `rax = 16 + rax + 4*rcx`
    (b) `rax = *(16 + rax + 4*rcx)`
    (c) `rax = 16 + *(rax + 4*rcx)`
    (d) `*(16 + rcx + 4*rax) = rax`
    (e) `rax = 16 + 4*rax + rcx`

16. Which line of Intel x86-64 assembly will perform the same operation as `rcx = ((int *)rax)[rcx]`?

    (a) `mov (%rax,%rcx,4),%rcx`
    (b) `lea (%rax,%rcx,4),%rcx`
    (c) `lea (%rax,4,%rcx),%rcx`
    (d) `mov (%rax,4,%rcx),%rcx`

17. If `a` is of type `(int)` and *b* is of type `(unsigned int)`, then `(a < b)` will perform

    (a) An unsigned comparison.
    (b) A signed comparison.
    (c) A segmentation fault.
    (d) A compiler error.

18. Denormalized floating point numbers are

    (a) Very close to zero (small magnitude)
    (b) Very far from zero (large magnitude)
    (c) Un-representable on a number line
    (d) Zero.

19. What is the difference between an arithmetic and logical right shift?

    (a) C uses arithmetic right shift; Java uses logical right shift.
    (b) Logical shift works on 32 bit data; arithmetic shift works on 64 bit data.
    (c) They fill in different bits on the left
    (d) They are the same.

20. Which of the following assembly instructions is invalid in Intel IA32 Assembly?

    (a) `pop %eip`
    (b) `pop %ebp`
    (c) `mov (%esp),%ebp`
    (d) `lea 0x10(%esp),%ebp`

Problem 2. (10 points):

Floating point encoding. Consider the following 5-bit floating point representation based on the IEEE floating point format. This format does not have a sign bit – it can only represent nonnegative numbers.

- There are $k = 3$ exponent bits. The exponent bias is 3.
- There are $n = 2$ fraction bits.

Recall that numeric values are encoded as a value of the form $V = M \times 2^E$, where E is the exponent after biasing, and M is the significand value. The fraction bits encode the significand value M using either a denormalized (exponent field 0) or a normalized representation (exponent field nonzero). The exponent E is given by $E = 1 - \mathit{Bias}$ for denormalized values and $E = e - \mathit{Bias}$ for normalized values, where e is the value of the exponent field `exp` interpreted as an unsigned number.

Below, you are given some decimal values, and your task it to encode them in floating point format. In addition, you should give the rounded value of the encoded floating point number. To get credit, you must give these as whole numbers (e.g., 17) or as fractions in reduced form (e.g., $3/4$). Any rounding of the significand is based on ***round-to-even***, which rounds an unrepresentable value that lies halfway between two representable values to the nearest even representable value.

Value	Floating Point Bits	Rounded value
$9/32$	`001 00`	1/4
1		
12		
11		
$1/8$		
$7/32$		

Problem 3. (6 points):

Accessing arrays. 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 following x86-64 assembly code:

```
# On entry:
#    %edi = x
#    %esi = y
#
copy_array:
        movslq  %esi,%rsi
        movslq  %edi,%rdi
        movq    %rsi, %rax
        salq    $4, %rax
        subq    %rsi, %rax
        addq    %rdi, %rax
        leaq    (%rdi,%rdi,2), %rdi
        addq    %rsi, %rdi
        movl    array1(,%rdi,4), %edx
        movl    %edx, array2(,%rax,4)
        movl    $1, %eax
        ret
```

What are the values of H and J?

H =

J =

Problem 4. (6 points):

Structure alignment. Consider the following C struct:

```
struct {
    char a, b;
    short c;
    long d;
    int *e;
    char f;
    float g;
} foo;
```

1. Show how the struct above would appear on a 32 bit Windows machine (primitives of size k are k-byte aligned). Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks to indicate bytes that are allocated in the struct but are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

2. Rearrange the above fields in foo to conserve the most space in the memory below. Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks to indicate bytes that are allcoated in the struct that are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Problem 5. (4 points):

Structure access. Consider the following data structure declaration:

```
struct ms_pacman{
  short wire;
  int resistor;
  union transistor{
    char bjt;
    int* mosfet;
    long vacuum_tube[2];
  }transistor;
  struct ms_pacman* connector;
};
```

Below are given four C functions and four x86-64 code blocks.

```
char* inky(struct ms_pacman *ptr){
  return &(ptr->transistor.bjt);
}

long blinky(struct ms_pacman *ptr){
  return ptr->connector->
         transistor.vacuum_tube[1];
}

int pinky(struct ms_pacman *ptr){
  return ptr->resistor;
}

int clyde(struct ms_pacman *ptr){
  return *(ptr->transistor.mosfet);
}
```

```
A   mov  0x8(%rdi), %rax
    mov  (%rax),%eax
    retq
```

```
B   lea  0x8(%rdi), %rax
    retq
```

```
C   mov  0x4(%rdi), %eax
    retq
```

```
D   mov  0x18(%rdi),%rax
    mov  0x10(%rax),%rax
    retq
```

In the following table, next to the name of each x86-64 code block, write the name of the C function that it implements.

Code Block	**Function Name**
A	
B	
C	
D	

Problem 6. (10 points):

Switch statement encoding. Consider the following C code and assembly code for a strange but simple function:

```
int lol(int a, int b)       40045c <lol>:
{                           40045c: lea    -0xd2(%rdi),%eax
    switch(a)               400462: cmp    $0x9,%eax
    {                       400465: ja     40048a <lol+0x2e>
        case 210:           400467: mov    %eax,%eax
            b *= 13;        400469: jmpq   *0x400590(,%rax,8)
            ________        400470: lea    (%rsi,%rsi,2),%eax
        case 213:           400473: lea    (%rsi,%rax,4),%eax
            b = 18243;      400476: retq
            ________        400477: mov    $0x4743,%esi
        case 214:           40047c: mov    %esi,%eax
            b *= b;         40047e: imul   %esi,%eax
            ________        400481: retq
        case 216:           400482: mov    %esi,%eax
        case 218:           400484: sub    %edi,%eax
            b -= a;         400486: retq
            ________        400487: add    $0xd,%esi
        case 219:           40048a: lea    -0x9(%rsi),%eax
            b += 13;        40048d: retq
            ________
        default:
            b -= 9;
    }

    return b;
}
```

Using the available information, fill in the jump table below. (Feel free to omit leading zeros.) Also, for each case in the `switch` block which should have a `break`, write `break` on the corresponding blank line.

Hint: `0xd2 = 210` and `0x4743 = 18243`.

```
0x400590:   ________________        0x400598:   ________________

0x4005a0:   ________________        0x4005a8:   ________________

0x4005b0:   ________________        0x4005b8:   ________________

0x4005c0:   ________________        0x4005c8:   ________________

0x4005d0:   ________________        0x4005d8:   ________________
```

Problem 7. (14 points):

Stack discipline. This problem concerns the following C code, compiled on a 32-bit machine:

```
void foo(char * str, int a) {

  int buf[2];
  a = a; /* Keep GCC happy */
  strcpy((char *) buf, str);

}

/*
  The base pointer for the stack
  frame of caller() is: 0xffffd3e8
*/
void caller() {

  foo(‘‘0123456’’, 0xdeadbeef);

}
```

Here is the corresponding machine code on a 32-bit Linux/x86 machine:

```
080483c8 <foo>:
080483c8 <foo+0>:     push   %ebp
080483c9 <foo+1>:     mov    %esp,%ebp
080483cb <foo+3>:     sub    $0x18,%esp
080483ce <foo+6>:     lea    -0x8(%ebp),%edx
080483d1 <foo+9>:     mov    0x8(%ebp),%eax
080483d4 <foo+12>:    mov    %eax,0x4(%esp)
080483d8 <foo+16>:    mov    %edx,(%esp)
080483db <foo+19>:    call   0x80482c0 <strcpy@plt>
080483e0 <foo+24>:    leave
080483e1 <foo+25>:    ret

080483e2 <caller>:
080483e2 <caller+0>:  push   %ebp
080483e3 <caller+1>:  mov    %esp,%ebp
080483e5 <caller+3>:  sub    $0x8,%esp
080483e8 <caller+6>:  movl   $0xdeadbeef,0x4(%esp)
080483f0 <caller+14>: movl   $0x80484d0,(%esp)
080483f7 <caller+21>: call   0x80483c8 <foo>
080483fc <caller+26>: leave
080483fd <caller+27>: ret
```

This problem tests your understanding of the stack discipline and byte ordering. Here are some notes to help you work the problem:

- `strcpy(char *dst, char *src)` copies the string at address `src` (including the terminating '`\0`' character) to address `dst`.
- Keep endianness in mind.
- You will need to know the hex values of the following characters:

Character	Hex value	Character	Hex value
'0'	0x30	'4'	0x34
'1'	0x31	'5'	0x35
'2'	0x32	'6'	0x36
'3'	0x33	'`\0`'	0x00

Now consider what happens on a Linux/x86 machine when `caller` calls `foo`.

A. Stack Concepts:
   a) Briefly describe the difference between the x86 instructions `call` and `jmp`.

   b) Why doesn't `ret` take an address to return to, like `jmp` takes an address to jump to?

B. Just before `foo` calls `strcpy`, what integer `x`, if any, can you guarantee that `buf[x] == a` ?

C. At what memory address is the string "0123456" stored (before it is `strcpy`'d)?

   We encourage you to use this space to draw pictures:

D. Just after `strcpy` returns to `foo`, fill in the following with hex values:

```
buf[0] = 0x_____ _____ _____ _____

buf[1] = 0x_____ _____ _____ _____

buf[2] = 0x_____ _____ _____ _____

buf[3] = 0x_____ _____ _____ _____

buf[4] = 0x_____ _____ _____ _____
```

E. Immediately before the call to `strcpy`, what is the the value at `%ebp` (not what is `%ebp`)?

F. Immediately before `foo`'s `ret` call, what is the value at `%esp` (what's on the top of the stack)?

G. Will a function that calls `caller()` segfault or notice any stack corruption? Explain.

Andrew ID (print clearly!):______________________________________

Full Name:______________________________________

15-213/18-213, Fall 2011

Exam 1

Tuesday, October 18, 2011

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew ID and full name on the front.
- This exam is closed book, closed notes (except for 1 double-sided note sheet). You may not use any electronic devices.
- 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 66 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Good luck!

1 (08):
2 (08):
3 (08):
4 (04):
5 (10):
6 (08):
7 (10):
8 (04):
9 (06):
TOTAL (66):

Problem 1. (8 points):

Multiple choice. Write your answer for each question in the following table:

1	2	3	4	5	6	7	8

1. What is the minimum (most negative) value of a 32-bit two's complement integer?

   (a) -2^{32}

   (b) $-2^{32} + 1$

   (c) -2^{31}

   (d) $-2^{31} + 1$

2. What is the difference between the `mov` and `lea` instructions?

   (a) `lea` dereferences an address, while `mov` doesn't.

   (b) `mov` dereferences an address, while `lea` doesn't.

   (c) `lea` can be used to copy a register into another register, while `mov` cannot.

   (d) `mov` can be used to copy a register into another register, while `lea` cannnot.

3. After executing the following code, which of the variables are equal to 0?

```
unsigned int a = 0xffffffff;
unsigned int b = 1;
unsigned int c = a + b;
unsigned long d = a + b;
unsigned long e = (unsigned long)a + b;
```

   (Assume `int`s are 32 bits wide and `long`s are 64 bits wide.)

   (a) None of them

   (b) `c`

   (c) `c` and `d`

   (d) `c`, `d`, and `e`

4. Assume a function `foo` takes two arguments. When calling `foo(arg1, arg2)`, which is the correct order of operations assuming x86 calling conventions and that `foo` must allocate stack space (implies that we must save the `%ebp`)?

   (a) `push arg1, push arg2, call foo, push %ebp`
   (b) `push arg1, push arg2, push %ebp, call foo`
   (c) `push arg2, push arg1, call foo, push %ebp`
   (d) `push arg2, push arg1, push %ebp, call foo`

5. Which one of the following statements is NOT true?

   (a) x86-64 provides a larger virtual address space than x86.
   (b) The stack disciplines for x86 and x86-64 are different.
   (c) x86 uses `%ebp` as the base pointer for the stack frame.
   (d) x86-64 uses `%rbp` as the base pointer for the stack frame.

6. Consider a 4-way set associative cache ($E = 4$). Which one of the following statements it true?

   (a) The cache has 4 blocks per line.
   (b) The cache has 4 sets per line.
   (c) The cache has 4 lines per set.
   (d) The cache has 4 sets per block.
   (e) None of the above.

7. Which one of the following statements about cache memories is true?

   (a) Fully associative caches offer better latency, while direct-mapped caches have lower miss rates.
   (b) Fully associative caches offer lower miss rates, while direct-mapped caches have better latency.
   (c) Direct-mapped caches have both better miss rates and better latency.
   (d) Both generally have similar latency and miss rates.

8. Consider an SRAM-based cache for a DRAM-based main memory. Neglect the possibility of other caches or levels of the memory hierarchy below main memory. If a cache is improved, increasing the typical hit rate from 98% to 99%, which one of the following would best characterize the likely decrease in typical access time?

   (a) 0%
   (b) 1%
   (c) 10%
   (d) 100%

Problem 2. (8 points):

Integer encoding. Fill in the blanks in the table below with the number described in the first column of each row. You can give your answers as unexpanded simple arithmetic expressions (such as $15^{213} + 42$); you should not have trouble fitting your answers into the space provided.

For this problem, assume a **6-bit word size**.

Description	Number
U_{max}	
T_{min}	
`(unsigned) ((int) 4)`	
`(unsigned) ((int) -7)`	
`(((unsigned) 0x21) << 1) & 0x3F)`	
`(int) (20 + 12)`	
`12 && 4`	
`(! 0x15) > 16`	

Problem 3. (8 points):

Floating point encoding. Consider the following 5-bit floating point representation based on the IEEE floating point format. This format does not have a sign bit – it can only represent nonnegative numbers.

- There are $k = 3$ exponent bits. The exponent bias is 3.
- There are $n = 2$ fraction bits.

Recall that numeric values are encoded as a value of the form $V = M \times 2^E$, where E is the exponent after biasing, and M is the significand value. The fraction bits encode the significand value M using either a denormalized (exponent field 0) or a normalized representation (exponent field nonzero). The exponent E is given by $E = 1 - \textit{Bias}$ for denormalized values and $E = e - \textit{Bias}$ for normalized values, where e is the value of the exponent field `exp` interpreted as an unsigned number.

Below, you are given some decimal values, and your task it to encode them in floating point format. In addition, you should give the rounded value of the encoded floating point number. To get credit, you must give these as whole numbers (e.g., 17) or as fractions in reduced form (e.g., $3/4$). Any rounding of the significand is based on ***round-to-even***, which rounds an unrepresentable value that lies halfway between two representable values to the nearest even representable value.

Value	Floating Point Bits	Rounded value
$9/32$	`001 00`	1/4
3		
9		
$3/16$		
$15/2$		

Problem 4. (4 points):

Functions. I never learned to properly comment my code, and now I've forgotten what this function does. Help me out by looking at the assembly code and reconstructing the C code for this recursive function. Fill in the blanks:

```
unsigned mystery1(unsigned n) {
     if(_________)
          return 1;
     else
          return 1 + mystery1(_________);
}
```

```
mystery1:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    cmpl $0, 8(%ebp)
    jne .L2
    movl $1, -4(%ebp)
    jmp .L3
.L2:
    movl 8(%ebp), %eax
    shrl %eax
    movl %eax, (%esp)
    call mystery1
    addl $1, %eax
    movl %eax, -4(%ebp)
.L3:
    movl -4(%ebp), %eax
    leave
    ret
```

Problem 5. (10 points):

Stack discipline. Consider the following C code and assembly code for a recursive function:

```
int gcd(int a, int b)       0x08048394 <+0>:    push   %ebp
{                           0x08048395 <+1>:    mov    %esp,%ebp
    if(!b)                  0x08048397 <+3>:    sub    $0x10,%esp
    {                       0x0804839a <+6>:    mov    0x8(%ebp),%eax
        return a;           0x0804839d <+9>:    mov    0xc(%ebp),%ecx
    }                       0x080483a0 <+12>:   test   %ecx,%ecx
                            0x080483a2 <+14>:   je     0x80483b7 <gcd+35>
    return gcd(b, a % b);   0x080483a4 <+16>:   mov    %eax,%edx
}                           0x080483a6 <+18>:   sar    $0x1f,%edx
                            0x080483a9 <+21>:   idiv   %ecx
                            0x080483ab <+23>:   mov    %edx,0x4(%esp)
                            0x080483af <+27>:   mov    %ecx,(%esp)
                            0x080483b2 <+30>:   call   0x8048394 <gcd>
                            0x080483b7 <+35>:   leave
                            0x080483b8 <+36>:   ret
```

Imagine that a program makes the procedure call `gcd(213, 18)`. Also imagine that prior to the invocation, the value of %esp is `0xffff1000`—that is, `0xffff1000` is the value of %esp *immediately before* the execution of the `call` instruction.

A. Note that the call `gcd(213, 18)` will result in the following function invocations: `gcd(213, 18)`, `gcd(18, 15)`, `gcd(15, 3)`, and `gcd(3, 0)`. Using the provided code and your knowledge of IA32 stack discipline, fill in the stack diagram with the values that would be present immediately before the execution of the `leave` instruction for `gcd(15, 3)`. Supply numerical values wherever possible, and cross out each blank for which there is insufficient information to complete with a numerical value.

   Hints: The following set of C style statements describes an approximation of the operation of the instruction `idiv %ecx`, where `'/'` is the division operator and `'%'` is the modulo operator:

   ```
   %eax = %eax / %ecx
   %edx = %eax % %ecx
   ```

   Also, recall that `leave` is equivalent to `movl %ebp, %esp; popl %ebp`

```
+---------------------------------+
|                                 |  0xffff1008
+---------------------------------+
|                                 |  0xffff1004
+---------------------------------+
|                                 |  0xffff1000
+---------------------------------+
|                                 |  0xffff0ffc
+---------------------------------+
|                                 |  0xffff0ff8
+---------------------------------+
|                                 |  0xffff0ff4
+---------------------------------+
|                                 |  0xffff0ff0
+---------------------------------+
|                                 |  0xffff0fec
+---------------------------------+
|                                 |  0xffff0fe8
+---------------------------------+
|                                 |  0xffff0fe4
+---------------------------------+
|                                 |  0xffff0fe0
+---------------------------------+
|                                 |  0xffff0fdc
+---------------------------------+
|                                 |  0xffff0fd8
+---------------------------------+
|                                 |  0xffff0fd4
+---------------------------------+
|                                 |  0xffff0fd0
+---------------------------------+
|                                 |  0xffff0fcc
+---------------------------------+
|                                 |  0xffff0fc8
+---------------------------------+
|                                 |  0xffff0fc4
+---------------------------------+
|                                 |  0xffff0fc0
+---------------------------------+
|                                 |  0xffff0fbc
+---------------------------------+
|                                 |  0xffff0fb8
+---------------------------------+
|                                 |  0xffff0fb4
+---------------------------------+
|                                 |  0xffff0fb0
+---------------------------------+
```

B. What are the values of %esp and %ebp immediately before the execution of the `ret` instruction for `gcd(15, 3)`?

Problem 6. (8 points):

Structs. Consider the following struct on an x86-64 Linux machine:

```
struct my_struct {
    char a;
    long long b;
    short c;
    float *d[2];
    unsigned char e[3];
    float f;
};
```

A. (4 points)

Please lay out the struct in memory below. Shade in any bytes used for padding and **clearly indicate the end of the** `struct`.

```
     +----+----+----+----+----+----+----+----+
0x0  |    |    |    |    |    |    |    |    |
     +----+----+----+----+----+----+----+----+
0x8  |    |    |    |    |    |    |    |    |
     +----+----+----+----+----+----+----+----+
0x10 |    |    |    |    |    |    |    |    |
     +----+----+----+----+----+----+----+----+
0x18 |    |    |    |    |    |    |    |    |
     +----+----+----+----+----+----+----+----+
0x20 |    |    |    |    |    |    |    |    |
     +----+----+----+----+----+----+----+----+
0x28 |    |    |    |    |    |    |    |    |
     +----+----+----+----+----+----+----+----+
0x30 |    |    |    |    |    |    |    |    |
     +----+----+----+----+----+----+----+----+
0x38 |    |    |    |    |    |    |    |    |
     +----+----+----+----+----+----+----+----+
```

B. (3 points)

Consider the following C function:

```
void foo(struct my_struct *st) {
    st->a = ’e’;
    st->d[0] = NULL;
    st->c = 0x213;

    printf("%lld %p %hhu\n", st->b, &st->f, st->e[1]);
}
```

Fill in the missing pieces of the diassembled version of `foo`.

```
(gdb) disassemble foo
Dump of assembler code for function foo:
0x00000000004004e4 <+0>:    sub    $0x8,%rsp
0x00000000004004e8 <+4>:    movb   $0x65,________(%rdi)
0x00000000004004eb <+7>:    movq   $0x0,________(%rdi)
0x00000000004004f3 <+15>:   movw   $0x213,________(%rdi)
0x00000000004004f9 <+21>:   movzbl ________(%rdi),%ecx
0x00000000004004fd <+25>:   lea    ________(%rdi),%rdx
0x0000000000400501 <+29>:   mov    ________(%rdi),%rsi
0x0000000000400505 <+33>:   mov    $0x40062c,%edi
0x000000000040050a <+38>:   mov    $0x0,%eax
0x000000000040050f <+43>:   callq  0x4003e0 <printf@plt>
0x0000000000400514 <+48>:   add    $0x8,%rsp
0x0000000000400518 <+52>:   retq
End of assembler dump.
```

C. (1 point)

How many bytes is the smallest possible struct containing the same elements as `my_struct`?

(a) 48

(b) 36

(c) 40

(d) None of the above

Problem 7. (10 points):

Switch statements. The problem concerns code generated by GCC for a function involving a switch statement. The code uses a jump to index into the jump table:

```
  0x4004b7:       jmpq   *0x400600(,%rax,8)
```

Using GDB, we extract the 8-entry jump table:

```
  0x400600: 0x00000000004004d1 0x00000000004004c8
  0x400610: 0x00000000004004c8 0x00000000004004be
  0x400620: 0x00000000004004c1 0x00000000004004d7
  0x400630: 0x00000000004004c8 0x00000000004004be
```

Here is the block of disassembled code implementing the switch statement:

```
# on entry: %rdi = x, %rsi = y, %rdx = z
  0x4004b0:       cmp    $0x7,%edx
  0x4004b3:       ja     0x4004c8
  0x4004b5:       mov    %edx,%eax
  0x4004b7:       jmpq   *0x400600(,%rax,8)
  0x4004be:       mov    %edi,%eax
  0x4004c0:       retq
  0x4004c1:       mov    $0x3,%eax
  0x4004c6:       jmp    0x4004da
  0x4004c8:       mov    %esi,%eax
  0x4004ca:       nopw   0x(%rax,%rax,1)
  0x4004d0:       retq
  0x4004d1:       mov    %edi,%eax
  0x4004d3:       and    $0x19,%eax
  0x4004d6:       retq
  0x4004d7:       lea    (%rdi,%rdi,1),%eax
  0x4004da:       add    %esi,%eax
  0x4004dc:       retq
```

Fill in the blank portions of the C code below to reproduce the function corresponding to this object code.

```
int test(int x, int y, int z)
{
  int result = 3;
  switch(z)
  {
    case ___:

      _____________;

    case ___:

    case ___:

      result = _____________;

      break;

    case ___:

      result = _____________;

    case ___:

      result = _____________;

      break;

    default:

      result = _____________;
  }
  return result;
}
```

Problem 8. (4 points):

Cache operation. Assume a cache memory with the following properties:

- The cache size (C) is 512 bytes (contains 512 data bytes)
- The cache uses an LRU (least-recently used) policy for eviction.
- The cache is initially empty.

Suppose that for the following sequence of addresses sent to the cache, **0, 2, 4, 8, 16, 32**, the hit rate is **0.33**. Then what is the block size (B) of the cache?

A. $B = 4$ bytes

B. $B = 8$ bytes

C. $B = 16$ bytes

D. None of the above.

Problem 9. (6 points):

Miss rate analysis. Listed below are two matrix multiply functions. The first, matrix_multiply computes $C = AB$, a standard matrix multiply. The second, matrix_multiply_t, computes $C = AB^T$, A times the transpose of B.

```
void matrix_multiply(float A[N][N], float B[N][N], float C[N][N])
{
    /* Computes C = A*B.  Assumes C starts as all zeros. */
    int i,j,k;
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            for (k=0; k<N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void matrix_multiply_t(float A[N][N], float B[N][N], float C[N][N])
{
    /* Computes C = A*transpose(B).  Assumes C starts as all zeros. */
    int i,j,k;
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            for (k=0; k<N; k++) {
                C[i][j] += A[i][k] * B[j][k];
            }
        }
    }
}
```

Assumptions:

- 8 MB 16-way cache with a block size of 64 bytes.
- N is very large, so that a single row or column cannot fit in the cache.
- `sizeof(float) == 4.`
- `C[i][j] is stored in a register.`

A. Approximately what miss rate do you expect the function `matrix_multiply` to have for large values of `N`?

(a) $\frac{1}{16}$
(b) $\frac{1}{8}$
(c) $\frac{1}{4}$
(d) $\frac{1}{2}$
(e) 1

B. Approximately what miss rate do you expect the function `matrix_multiply_t` to have for large values of `N`?

(a) $\frac{1}{16}$
(b) $\frac{1}{8}$
(c) $\frac{1}{4}$
(d) $\frac{1}{2}$
(e) 1

Andrew ID (print clearly!):______________________________________

Full Name:______________________________________

15-213/18-213, Fall 2012

Midterm Exam

Tuesday, October 16, 2012

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew ID and full name on the front.
- This exam is closed book, closed notes (except for 1 double-sided note sheet). You may not use any electronic devices.
- 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 71 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Good luck!

1 (10):	
2 (07):	
3 (06):	
4 (08):	
5 (08):	
6 (04):	
7 (08):	
8 (10):	
9 (10):	
TOTAL (71):	

Problem 1. (10 points):

Multiple choice. Write your answer for each question in the following table:

1	2	3	4	5	6	7	8	9	10

1. What is the output of the following code?
   Assume that int is 32 bits, short is 16 bits, and the representation is two's complement.

   ```
   unsigned int x = 0xDEADBEEF;
   unsigned short y = 0xFFFF;
   signed int z = -1;
   if (x > (signed short) y)
       printf("Hello");
   if (x > z)
       printf("World");
   ```

   (a) Prints nothing.

   (b) Prints ”Hello”

   (c) Prints ”World”

   (d) Prints ”HelloWorld”

2.
   ```
   1) mov (%eax, %eax, 4), %eax
   2) lea (%eax, %eax, 4), %eax
   ```

   Which of the above accomplishes the following: `%eax = 5 * %eax`?

   (a) Neither 1 nor 2.

   (b) Only 1.

   (c) Only 2.

   (d) Both 1 and 2.

3. The x86-64 instruction `test` is best described as which of the following:

   (a) Same as `sub`.

   (b) Same as `sub`, but doesn't keep the result (only sets flags).

   (c) Same as `and`.

   (d) Same as `and`, but doesn't keep the result (only sets flags).

4. In the following code, what order of loops exhibits the best locality?

```
// int a[X][Y][Z] is declared earlier
int i, j, k, sum = 0;
for (i = 0; i < Y; i++)
    for (j = 0; j < Z; j++)
        for (k = 0; k < X; k++)
            sum += a[k][i][j];
```

(a) i on the outside, j in the middle, k on the inside (as is).

(b) j on the outside, k in the middle, i on the inside.

(c) k on the outside, i in the middle, j on the inside.

(d) The order does not matter.

5. Which expression will evaluate to `0x1` if `x` is a multiple of 32 and `0x0` otherwise? Assume that `x` is an unsigned `int`.

(a) `!(x & 0x1f)`

(b) `!(x & 0x3f)`

(c) `(x & 0x1f)`

(d) `(x | 0x3f)`

(e) `!(x ^ 0x1f)`

6. On a 32-bit Linux system, what is the size of a `long`?

(a) 2 bytes

(b) 4 bytes

(c) 6 bytes

(d) 8 bytes

(e) 16 bytes

7. Consider the C declaration

```
int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Suppose that the compiler has placed the variable `array` in the `%ecx` register. How do you move the value at `array[3]` into the `%eax` register? Assume that `%ebx` is 3.

(a) `leal 12(%ecx),%eax`

(b) `leal (%ecx,%ebx,4),%eax`

(c) `movl (%ecx,%ebx,4),%eax`

(d) `movl 8(%ecx,%ebx,2),%eax`

(e) `leal 4(%ecx,%ebx,1),%eax`

8. Why does the technique called “blocking” help with cache utilization when transposing a matrix?

   (a) Inductive locality

   (b) Spatial locality

   (c) Monadic locality

   (d) Temporal locality

   (e) Internet locality

9. What is NOT true about 64-bit Linux systems?

   (a) `%rax` is used for function return values

   (b) There are more registers than there are in 32-bit systems

   (c) All function arguments are passed on the stack

   (d) `%rbp` can be used like any other register; there is no base pointer

   (e) `%eax` and `%ebx` can be used like in a 32-bit system.

10. On a 64-bit system, if `%rsp` has the value `0x7ffff0000` immediately before a `retq` instruction, what is the value of `%rsp` immediately after the `retq`?

    (a) `0x7ffffefff8`

    (b) `0x7fffff0000`

    (c) `0x7fffff0004`

    (d) `0x7fffff0008`

    (e) The return address

Problem 2. (7 points):

Integer encoding. Assume we are running code on two machines using two's complement arithmetic for signed integers. Machine 1 has 4-bit integers and Machine 2 has 6-bit integers. Fill in the empty boxes in the table below. The following definitions are used in the table:

```
int x = -5;
unsigned ux = x;
```

Expression	4-bit decimal	4-bit binary	6-bit decimal	6-bit binary
-8	-8		-8	
$-$TMin				
$x >> 1$				
$(-x\hat{}(-1)) >> 2$				

Problem 3. (6 points):

Floating point encoding. In this problem, you will work with floating point numbers based on the IEEE floating point format. We consider two different 6-bit formats:

Format A:

- There is one sign bit s.
- There are $k = 3$ exponent bits. The bias is $2^{k-1} - 1 = 3$.
- There are $n = 2$ fraction bits.

Format B:

- There is one sign bit s.
- There are $k = 2$ exponent bits. The bias is $2^{k-1} - 1 = 1$.
- There are $n = 3$ fraction bits.

For formats A and B, please write down the binary representation for the following (use round-to-even). Recall that for denormalized numbers, $E = 1 - \text{bias}$. For normalized numbers, $E = e - \text{bias}$.

Value	Format A Bits	Format B Bits
Zero	`0 000 00`	`0 00 000`
One		
$1/2$		
$11/8$		

Problem 4. (8 points):

Loops. Consider the following x86 assembly code:

```
(gdb) disassemble transform
   0x080483d0 <+0>:      push   %ebp
   0x080483d1 <+1>:      mov    %esp,%ebp
   0x080483d3 <+3>:      mov    0x8(%ebp),%edx
   0x080483d6 <+6>:      mov    $0x0,%eax
   0x080483db <+11>:     test   %edx,%edx
   0x080483dd <+13>:     je     0x80483ec <transform+28>
   0x080483df <+15>:     test   $0x1,%dl
   0x080483e2 <+18>:     je     0x80483e8 <transform+24>
   0x080483e4 <+20>:     lea    0x1(%eax,%eax,1),%eax
   0x080483e8 <+24>:     shr    %edx
   0x080483ea <+26>:     jne    0x80483df <transform+15>
   0x080483ec <+28>:     pop    %ebp
   0x080483ed <+29>:     ret
```

Given this assembly code, reconstruct the C `transform` function.

- Recall that `%dl` is the low-order byte of `%edx`.
- Recall that if a shift amount is not specified in the `shr` instruction, a default shift amount of 1 is used. Hence, the `shr %edx` instruction updates the `%edx` register by shifting its value to the right by one bit position.

```
unsigned transform(unsigned n)
{
    int b, m;

    for (m = _______; ___________________; ____________________) {

        b =_______________________;

        if (b == 0) {

            __________________________________;
        }

        __________________________________;
    }

    return m;
}
```

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct {
  char a;
  long b;
  float c;
  char d[3];
  int *e;
  short *f;
} foo;
```

1. Show how `foo` would be allocated in memory on an x86-64 Linux system. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

2. Rearrange the elements of `foo` to conserve the most space in memory. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Problem 6. (4 points):

Struct Access. Now for something totally different... Dr. Grave O'Danger is now head of the computer science department and has decided to make it impossible for you to graduate.

```
struct confuse {
  char systems;
  long theory;
  struct applications {
    char web[3];
  } database;
  int *languages;
  struct confuse *math;
};
```

Below are three C functions and three x86-64 Linux code blocks.

```
char *bachelors(struct confuse *ptr) {
  return &(ptr->database.web[2]);
}

int masters(struct confuse *ptr) {
  return *(ptr->languages);
}

long phd(struct confuse *ptr) {
  return ptr->math->theory;
}
```

```
A | mov    0x20(%rdi), %rax
  | mov    0x8(%rax), %rax
  | retq
```

```
B | lea    0x12(%rdi),%rax
  | retq
```

```
C | mov    0x18(%rdi), %rax
  | mov    (%rax), %eax
  | retq
```

In the following table, next to the name of each x86-64 code block, write the name of the C function that it implements.

Code Block	Function Name
A	
B	
C	

Problem 7. (8 points):

Switch statements. The following problem tests your understanding of switch statements that use jump tables.

Consider a switch statement with the following implementation. The code uses this `jmpq` instruction to index into the jump table:

```
0x40047b      jmpq   *0x400598(,%rdi,8)
```

Using GDB we extract the jump table:

```
0x400598:    0x0000000000400488          0x0000000000400488
0x4005a8:    0x000000000040048b          0x0000000000400493
0x4005b8:    0x000000000040049a          0x0000000000400482
0x4005c8:    0x000000000040049a          0x0000000000400498
```

Here is the assembly code for the switch statement:

```
  #on entry :  %rdx = c and %rsi = b
   0x400474 :    cmp    $0x7,%edi
   0x400477 :    ja     0x40049a
   0x400479 :    mov    %edi,%edi
   0x40047b :    jmpq   *0x400598(,%rdi,8)
   0x400482 :    mov    $0x15213,%eax
   0x400487 :    retq
   0x400488 :    sub    $0x5,%edx
   0x40048b :    lea    0x0(,%rdx,4),%eax
   0x400492 :    retq
   0x400493 :    mov    $0x2,%edx
   0x400498 :    and    %edx,%esi
   0x40049a :    lea    0x4(%rsi),%eax
   0x40049d :    retq
```

Fill in the C code implementing this switch statement:

```
int main(int a, int b, int c){
     int result = 4;

     switch(a){
        case 0:

        case 1:

             __________;

        case __:

             __________;

             break;

        case __:

             result = __________;

             break;

        case 3:

             __________;

        case 7:

             __________;

        default:

             __________;

    }

    return result;
}
```

Problem 8. (10 points):

Stack discipline. Consider the following C code and its corresponding 32-bit x86 machine code. Please complete the stack diagram on the following page.

```
int bar (int a, int b) {
    return a + b;
}

int foo(int n, int m, int c) {
    c += bar(m, n);
    return c;
}
```

```
08048374 <bar>:
 8048374:       55                      push   %ebp
 8048375:       89 e5                   mov    %esp,%ebp
 8048377:       8b 45 0c                mov    0xc(%ebp),%eax
 804837a:       03 45 08                add    0x8(%ebp),%eax
 804837d:       5d                      pop    %ebp
 804837e:       c3                      ret

0804837f <foo>:
 804837f:       55                      push   %ebp
 8048380:       89 e5                   mov    %esp,%ebp
 8048382:       83 ec 08                sub    $0x8,%esp
 8048385:       8b 45 08                mov    0x8(%ebp),%eax
 8048388:       89 44 24 04             mov    %eax,0x4(%esp)
 804838c:       8b 45 0c                mov    0xc(%ebp),%eax
 804838f:       89 04 24                mov    %eax,(%esp)
 8048392:       e8 dd ff ff ff          call   8048374 <bar>
 8048397:       03 45 10                add    0x10(%ebp),%eax
 804839a:       c9                      leave
 804839b:       c3                      ret
```

A. Draw a detailed picture of the stack, starting with the caller invoking `foo(3, 4, 5)`, and ending immediately **before** execution of the `ret` instruction in `bar`.

- The stack diagram should begin with the three arguments for `foo` that the caller has placed on the stack. To help you get started, we have given you the first one.
- Use the actual values for function arguments, rather than variable names. For example, use 3 or 4 instead of `n` or `m`.
- Always label `%ebp` and give its value when it is pushed to the stack, e.g., `%ebp: 0xffff1400`.
- You may not need to fill in all of the boxes in the diagram.

```
Value of %ebp when foo is called: 0xffffd858
Return address in function that called foo: 0x080483c9

Stack      The diagram starts with the
addresss   arguments for foo()
           +-----------------------------------+
0xffffd850 |                5                  |
           +-----------------------------------+
0xffffd84c |                                   |
           +-----------------------------------+
0xffffd848 |                                   |
           +-----------------------------------+
0xffffd844 |                                   |
           +-----------------------------------+
0xffffd840 |                                   |
           +-----------------------------------+
0xffffd83c |                                   |
           +-----------------------------------+
0xffffd838 |                                   |
           +-----------------------------------+
0xffffd834 |                                   |
           +-----------------------------------+
0xffffd830 |                                   |
           +-----------------------------------+
```

B. What is the final value of `%ebp`, immediately **before** execution of the `ret` instruction in `bar`?

`%ebp=0x`_________________________

C. What is the final value of `%esp`, immediately **before** execution of the `ret` instruction in `bar`?

`%esp=0x`________________________

Problem 9. (10 points):

Caches. Consider a computer with an **8-bit address space** and a **direct-mapped 64-byte** data cache with **8-byte cache blocks**.

A. 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) by labeling them as follows:

B: Block Offset

S: Set Index

T: Cache Tag

7	6	5	4	3	2	1	0

B. The table below shows a trace of load addresses accessed in the data cache. Assume the cache is initially empty. For each row in the table, please complete the two rightmost columns, indicating (i) the *set number* (in decimal notation) for that particular load, and (ii) whether that loads *hits* (H) or *misses* (M) in the cache (circle either "H" or "M" accordingly).

Load	Hex	Binary	Set Number?	Hit or Miss?
No.	Address	Address	(in Decimal)	(Circle one)
1	43	0100 0011		H M
2	b2	1011 0010		H M
3	40	0100 0000		H M
4	f9	1111 1001		H M
5	b2	1011 0010		H M
6	93	1001 0011		H M
7	d0	1101 0000		H M
8	b0	1011 0000		H M
9	67	0110 0111		H M
10	07	0000 0111		H M

C. For the trace of load addresses shown in Part B, below is a list of possible final states for the cache, showing the hex value of the tag for each cache block in each set. Assume that initially all cache blocks are invalid (represented by X).

(a)

Set:	0	1	2	3	4	5	6	7
Tag:	0	X	2	1	X	X	2	3

(b)

Set:	0	1	2	3	4	5	6	7
Tag:	0	3	3	X	1	X	2	X

(c)

Set:	0	1	2	3	4	5	6	7
Tag:	0	X	3	X	1	X	2	3

(d)

Set:	0	1	2	3	4	5	6	7
Tag:	X	0	X	X	1	X	1	3

(e)

Set:	0	1	2	3	4	5	6	7
Tag:	0	X	0	X	1	X	X	3

(f)

Set:	0	1	2	3	4	5	6	7
Tag:	0	4	X	2	1	X	X	4

(g)

Set:	0	1	2	3	4	5	6	7
Tag:	1	X	2	X	1	X	2	3

Which of the choices above is the correct final state of the cache? ______

Andrew login ID:____________________________________

Full Name:____________________________________

CS 15-213, Spring 2002

Exam 1

Febrary 26, 2002

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 84 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. You may use any books or notes you like. You may use a calculator, but no laptops or other wireless devices. Good luck!

1 (11):	
2 (19):	
3 (9):	
4 (10):	
5 (11):	
6 (12):	
7 (12):	
TOTAL (84):	

Problem 1. (11 points):

Assume we are running code on a 6-bit machine using two's complement arithmetic for signed integers. A "short" integer is encoded using 3 bits. Fill in the empty boxes in the table below. The following definitions are used in the table:

```
short sy = -3;
int y = sy;
int x = -13;
unsigned ux = x;
```

Note: You need not fill in entries marked with "–".

Expression	Decimal Representation	Binary Representation
Zero	0	
–	-3	
–		`11 0010`
ux		
y		
$x >> 2$		
TMax		

Problem 2. (19 points):

Consider the following 8-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next 3 bits are the exponent. The exponent bias is $2^{3-1} - 1 = 3$.
- The last 4 bits are the fraction.
- The representation encodes numbers of the form: $V = (-1)^s \times M \times 2^E$, where M is the significand and E is the biased exponent.

The rules are like those in the IEEE standard(normalized, denormalized, representation of 0, infinity, and NAN). FILL in the table below. Here are the instructions for each field:

- **Binary:** The 8 bit binary representation.
- **M:** The value of the significand. This should be a number of the form x or $\frac{x}{y}$, where x is an integer, and y is an integral power of 2. Examples include 0, $\frac{3}{4}$.
- **E:** The integer value of the exponent.
- **Value:**The numeric value represented.

  Note: you need not fill in entries marked with ”—”.
- **Smallest/Largest:** These refer to the absolute value. The number’s relationship with 0 is indicated with positive/negative.

Description	Binary	M	E	Value
Positive zero				$+0.0$
—	`0 000 0101`			
Largest denormalized (positive)				
Smallest normalized (negative)				
Negative Two				-2.0
—				-14.5
Negative infinity		—	—	$-\infty$

Problem 3. (9 points):

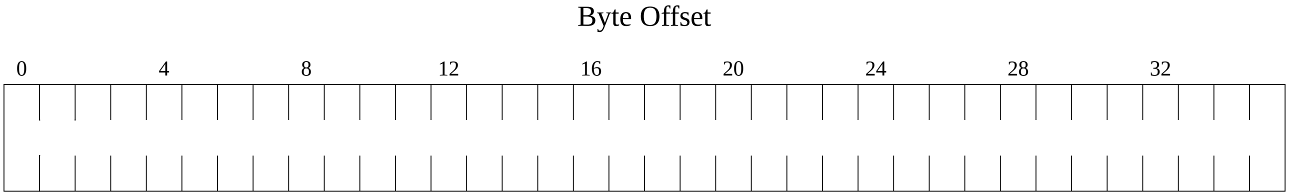
Consider the following C type definitions:

```
typedef struct {        typdef union {
     char c[2];              int i;
     int *intp;              char buf[6];
     union1 u;               float f;
     double d;          } union1
     short s;
} struct1
```

Note: Assume the IA-32 **Windows** alignment convention for this problem – in particular, values of the type **double** must be 8-byte aligned. (vs. Linux where they are only 4-byte aligned).

A. What byte alignment is necessary for an object of type union1?

B. On the below template, draw out the allocation of data for an object of type struct1. Allowing for a maximum of 36 bytes, mark off and label each element in the struct as well as cross-hatch the unused, but allocated space. Clearly indicate the right hand boundary with a vertical line.



C. How many bytes need to be allocated for an object of type struct1?

D. Think about redefining the fields of struct1 in a different order so as to minimize the amount of unused, but allocated space for each variable of type struct1. Now how many bytes are required for an struct1 object?

Problem 4. (10 points):

Fill in the blanks of the C code. The assembly code and the memory status for the C code are given as follows.

Assembly code:

```
 08048430 <foo>:                              8048454:   mov    %ecx,%eax
  8048430:   push   %ebp                      8048456:   or     %edx,%eax
  8048431:   mov    %esp,%ebp                 8048458:   jmp    804846f <foo+0x3f>
  8048433:   push   %ebx                      804845a:   mov    %esi,%esi
  8048434:   mov    0x8(%ebp),%ebx            804845c:   mov    %ecx,%eax
  8048437:   xor    %eax,%eax                 804845e:   xor    %edx,%eax
  8048439:   cmp    $0x4,%ebx                 8048460:   jmp    804846f <foo+0x3f>
  804843c:   mov    0xc(%ebp),%ecx            8048462:   mov    %esi,%esi
  804843f:   mov    0x10(%ebp),%edx           8048464:   mov    %ecx,%eax
  8048442:   ja     804846f <foo+0x3f>        8048466:   not    %eax
  8048444:   jmp    *0x8048500(,%ebx,4)       8048468:   jmp    804846f <foo+0x3f>
  804844b:   nop                              804846a:   mov    %esi,%esi
  804844c:   mov    %ecx,%eax                 804846c:   lea    (%edx,%ecx,1),%eax
  804844e:   and    %edx,%eax                 804846f:   mov    (%esp,1),%ebx
  8048450:   jmp    804846f <foo+0x3f>        8048472:   leave
  8048452:   mov    %esi,%esi                 8048473:   ret
```

Memory information given by gdb:

```
>gdb foo
(gdb) x /8w 0x8048500
0x8048500 : 0x0804844c 0x08048454 0x0804845c 0x08048464
0x8048510 : 0x0804846c 0x00000000 0x00000000 0x00000000
```

C code:

```
int foo(int op, int a, int b)
{
    int result = 0;
    switch (op)
    {
        case 0:_________________________________;

        case 1:_________________________________;

        case 2:_________________________________;

        case 3:_________________________________;

        case 4:_________________________________;
    }
    return result;
}
```

Problem 5. (11 points):

The memory stack can become a large performance bottleneck for programs. For instance, each function must at least save the old base pointer as well as restore it once it is done executing. This leaves out other stack interactions, such as function calls.

One proposed way of increasing the performance of stack operations is to emulate a stack using registers on the processor itself. The Sun SPARC and Intel Itanium architectures both use register stacks in this fashion.

Now suppose you work for a processor company that has created a very specific processor for the scientific community. Hence, it only needs to deal with 4 byte quantities.

The company is the final stages of completing its processor, and is finishing up the documentation for their product. However, the technical writers are having problems understanding the register stack and they want you to explain it to them. Basically, they want to explain to C programmers how the processor emulates stack operations.

The following are the relevant operations the processor can perform:

- alloc: similar to operations performed at the beginning of a function
- push: a stack push operation
- pop: a stack pop operation
- call: used to call a function, similar to call from IA-32

The following are the registers the processor can use:

- RET*n*: return address registers
- BP*n*: stack frame pointer registers
- STK*n*: general purpose stack registers

Note that, although there is no reserved base pointer register (such as ebp), there is a notion of a base pointer.

A. Using the information above, fill in what each of the processor's stack operations perform.

- alloc

- push

- pop

- call

B. What is the problem with using registers in this fashion?

C. Why is the notion of a base pointer still required?

Problem 6. (12 points):

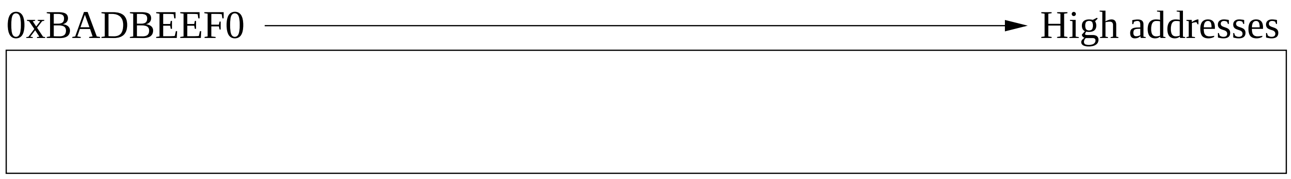
Dr. Evil was very impressed by your performance with the binary bomb and he has decided to hire you as a computer security consultant.

You are presented with the following code written by his minions. The provided code is responsible for handling incoming command requests from foreign hosts. The network connections have been tied to standard input (stdin), so input is handled exactly as it would have been had a user entered the input from the keyboard.

A. Which lines of code are weak? Please fix them.

B. Given the code as shown below, please show how you could position an exploit string in memory to ensure that it is executed.

Specifically, assume that the box below represents your input. The beginning of the buffer is located at memory address 0xbadbeef0. You have determined that your exploit code requires 32 bytes. Please draw lines to separate this box into sections, with one section for each component of the input. The label each box to indicate what it stores and its offset within the input string. For example, one section should be the text of the exploit. In other words, the hex code representing the assembly, Please don't forget to show the padding between the sections.



```
#define NUM_CMDS 10
extern char *cmds[NUM_CMDS];

1. int readcmd(void)
2.
3.          char *cmdname;
4.          char *result_fname;
5.          int index, fd;
6.          char buffer[512];
7.
8.          gets(buffer);
9.
10.         sscanf(buffer, "%d", &index);
11.
12.         /* Get command name */
13.         cmdname = cmds[index];
14.
15.         result_fname = process_cmd(cmdname,arg);
16.
17.         /* Log command under result for later analysis */
18.
19.         fd = open(result_fname, O_CREAT | O_WRONLY | O_APPEND);
20.         write(fd, buffer, sizeof(buffer));
21.
22.         return 0;
23.
```

Problem 7. (GG points):

Consider the following C function:

```
int FindMin(int *A, int size)
{
    int i;
    int min = A[0];

    for(i = 0; i < size; i++)
    {
       if(A[i] < min) {
          min = A[i];
       }
    }

    return min;
}
```

Fill in the assembly code for the body of the function.

```
FindMin:
   push %ebp
   mov %esp, %ebp
   mov 0x8(%ebp), %ecx
   mov 0xc(%ebp), %edx
```

```
   mov %ebp, %esp
   pop %ebp
   ret
```

Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Spring 2003

Exam 1

February 27, 2003

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 60 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. You may use any books or notes you like. No electronic devices are allowed. Good luck!

1 (1):	
2 (9):	
3 (8):	
4 (4):	
5 (8):	
6 (12):	
7 (10):	
8 (8):	
TOTAL (60):	

Problem 1. (1 points):

The correct answer to this problem is worth 1 point. An incorrect answer is worth -2 points. No answer will be scored as 0 points. Note: The answer to this question was given in lecture.

The correct answer to this problem is: ______________

Problem 2. (9 points):

Assume we are running code on an 8-bit machine using two's complement arithmetic for signed integers. Short integers are encoded using 4 bits. Sign extension is performed whenever a short is casted to an int. For this problem, assume that all shift operations are arithmetic. Fill in the empty boxes in the table below.

```
int i = -11;
unsigned ui = i;
short s = -2;
unsigned short us = s;
```

Note: You need not fill in entries marked with "–". TMax denotes the largest positive two's complement number and TMin denotes the minimum negative two's complement number. Finally, you may use hexidecimal notation for your answers in the "Binary Representation" column.

Expression	Decimal Representation	Binary Representation
Zero	0	
–	-3	
i		
i >> 4		
ui		
(int) s		
(int)(s ˆ 7)		
(int) us		
TMax		
TMin		

Problem 3. (8 points):

Consider the following 7-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next 3 bits are the exponent. The exponent bias is 3.
- The last 3 bits are the fraction.
- The representation encodes numbers of the form: $V = (-1)^s \times M \times 2^E$, where M is the significand and E the integer value of the exponent.

Please fill in the table below. You do not have to fill in boxes with ”——” in them. **If a number is NAN or infinity, you may disregard the M, E, and V fields below.** However, fill the Description and Binary fields with valid data.

Here are some guidelines for each field:

- **Description -** A verbal description if the number has a special meaning
- **Binary -** Binary representation of the number
- M **-** Significand (same as the M in the formula above)
- E - Exponent (same as the E in 2^E)
- V - Fractional Value represented

Please fill the M, E, and V fields below with rational numbers (fractions) rather than decimals or binary decimals

Description	Binary	M	E	V
——	`0 010 010`			
$2\ \frac{3}{8}$				
	`1 111 000`			
Most Negative **Normalized**				
Smallest Positive **Denormalized**				

Problem 4. (4 points):

Consider the following assembly instructions and C functions:

```
080483b4 <funcX>:
 80483b4:  55                     push   %ebp
 80483b5:  89 e5                  mov    %esp,%ebp
 80483b7:  8b 45 08               mov    0x8(%ebp),%eax
 80483ba:  8d 04 80               lea    (%eax,%eax,4),%eax
 80483bd:  8d 04 85 f6 ff ff ff   lea    0xfffffff6(,%eax,4),%eax
 80483c4:  89 ec                  mov    %ebp,%esp
 80483c6:  5d                     pop    %ebp
 80483c7:  c3                     ret
```

Circle the C function below that generates the above assembly instructions:

```
int func1(int n) {

  return n * 20 - 10;

}

int func2(int n) {

  return n * 24 + 6;

}

int func3(int n) {

  return n * 16 - 4;

}
```

Problem 5. (8 points):

Consider the following pairs of C functions and assembly code. Draw a line connecting each C function with the block(s) of assembly code that implements it. **There is not necessarily a one-to-one correspondence. Draw an X through any C and/or assembly code fragments that don't have a match.**

```
int scooby(int *a)
{
    return a[4];
}
```

```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %eax
sall    $2, %eax
popl    %ebp
ret
```

```
int dooby(int a)
{
    return a*4;
}
```

```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %eax
movsbl  4(%eax),%eax
popl    %ebp
ret
```

```
int doo(int a)
{
    return a<<4;
}
```

```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %eax
sall    $4, %eax
popl    %ebp
ret
```

```
char scrappy(char *a)
{
    return a[4];
}
```

```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %eax
leal    0(,%eax,4), %eax
popl    %ebp
ret
```

Problem 6. (12 points):

Recently, Microsoft's SQL Server was hit by the SQL Slammer worm, which exploits a known buffer overflow in the SQL Resolution Service. Today, we'll be writing our own *213 Slammer* that exploits the vulnerability introduced in `bufbomb`, the executable used in your Lab 3 assignment. And as such, `Gets` has the same functionality as in Lab 3 except that it strips off the newline character before storing the input string.

Consider the following exploit code, which runs the program into an infinite loop:

```
infinite.o:     file format elf32-i386
Disassembly of section .text:

00000000 <.text>:
   0:   68 fc b2 ff bf          push   $0xbfffb2fc
   5:   c3                      ret
   6:   89 f6                   mov    %esi,%esi
```

Here is a disassembled version of the `getbuf` function in `bufbomb`, along with the values of the relevant registers and a printout of the stack **before** the call to `Gets()`.

```
(gdb) disas
Dump of assembler code for function getbuf:
0x8048a44 <getbuf>:     push   %ebp
0x8048a45 <getbuf+1>:   mov    %esp,%ebp
0x8048a47 <getbuf+3>:   sub    $0x18,%esp
0x8048a4a <getbuf+6>:   add    $0xfffffff4,%esp
0x8048a4d <getbuf+9>:   lea    0xfffffff4(%ebp),%eax
0x8048a50 <getbuf+12>:  push   %eax
0x8048a51 <getbuf+13>:  call   0x8048b50 <Gets>
0x8048a56 <getbuf+18>:  mov    $0x1,%eax
0x8048a5b <getbuf+23>:  mov    %ebp,%esp
0x8048a5d <getbuf+25>:  pop    %ebp
0x8048a5e <getbuf+26>:  ret
0x8048a5f <getbuf+27>:  nop
End of assembler dump.

(gdb) info registers
eax            0xbfffb2fc       ecx            0xffffffff
edx            0x0              ebx            0x0
esp            0xbfffb2e0       ebp            0xbfffb308
esi            0xffffffff       edi            0x804b820

(gdb) x/20xb $ebp-12
0xbfffb2fc:   0xf0  0x17  0x02  0x40  0x18  0xb3  0xff  0xbf
0xbfffb304:   0x50  0x80  0x06  0x40  0x28  0xb3  0xff  0xbf
0xbfffb30c:   0xee  0x89  0x04  0x08  0x24  0xb3  0xff  0xbf
```

Here are the questions:

1. Write down the address of the location on the stack which contains the return address where `getbuf` is supposed to return to:

   0x______________________

2. Using the exploit code illustrated above, fill in the the following blanks on the stack **after** the call to `Gets()`. All the numbers must be in a **two character hexadecimal** representation of a byte. We've already filled in the terminating `\0` (0x00) character for you.

   ```
   (gdb) x/20xb $ebp-12

   0xbfffb2fc:   0x___  0x___  0x___  0x___  0x___  0x___  0x___  0x___

   0xbfffb304:   0x___  0x___  0x___  0x___  0x___  0x___  0x___  0x___

   0xbfffb30c:   0x___  0x___  0x___  0x___  0x00  0xb3  0xff  0xbf
   ```

3. During the infinite loop, what is the value of `%ebp`?

   0x____________________________

Problem 7. (10 points):

This problem tests your understanding of both control flow and multidimensional array layout. Consider the following assembly code for a procedure `moo()`:

```
moo:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %edi
        pushl   %esi
        pushl   %ebx
        movl    $0, %ecx
        movl    $arr1, %edi
        movl    $arr2+8, %esi
        movl    8(%ebp), %eax
        leal    0(,%eax,4), %ebx
.L5:
        leal    (%ecx,%ecx,2), %eax
        sall    $2, %eax
        movl    %ebx, %edx
        addl    (%eax,%esi), %edx
        movl    %edx, (%eax,%edi)
        incl    %ecx
        cmpl    $10, %ecx
        jle     .L5
        movl    %ecx, %eax
        popl    %ebx
        popl    %esi
        popl    %edi
        popl    %ebp
        ret
```

Based on the assembly code, fill in the blanks below in `moo`'s C source code. (Note: you may only use symbolic variables from the source code in your expressions below-do *not* use register names.) **Hint:** First figure out what the loop variable (i) is in the assembly and what the value of M is.

```
#define M _____

#define N _____


int arr1[M][N];


int arr2[M][N];


int moo(int x)
{
    int i;

    for(__________; i < M; ________ )
    {

       arr1[_____][_____] = arr2[_____][_____]+______________;
    }

    return _______;
}
```

Problem 8. (8 points):

Consider the following C declarations:

```
typedef struct {
  char          ID[2];
  double        weight;
  double        *components;
  short         momentum;
} Projectile;
```

A. Using the templates below (allowing a maximum of 24 bytes), indicate the allocation of data for structs of type `Projectile`. Mark off and label the areas for each individual element (arrays may be labeled as a single element). **Cross hatch the parts that are allocated, but not used, and be sure to clearly indicate the end of the structure. Assume the Linux alignment rules discussed in class.**

`Projectile:`

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. How would you define the `Modified` structure to minimize the number of bytes allocated for the structure using the same fields as the `Projectile` structure?

```
 typedef struct {



} Modified;
```

C. What is the value of `sizeof(Modified)`?

Andrew login ID:____________________________________

Full Name:____________________________________

CS 15-213, Spring 2004

Exam 1

February 26, 2004

Instructions:

- Make sure that your exam is not missing any sheets (there should be 15), 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 80 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. You may use any books or notes you like. No electronic devices are allowed. Good luck!

1 (9):	
2 (14):	
3 (12):	
4 (8):	
5 (12):	
6 (12):	
7 (6):	
8 (7):	
TOTAL (80):	

Problem 1. (9 points):

Assume we are running code on a 10-bit machine using two’s complement arithmetic for signed integers. Short integers are encoded using 5 bits. Sign extension is performed whenever a short is casted to an int. For this problem, assume that all shift operations are arithmetic. Fill in the empty boxes in the table below.

```
int i = -42;
unsigned ui = i;
short s = -7;
unsigned short us = s;
```

Note: You need not fill in entries marked with ”–”. TMax denotes the largest positive two’s complement number and TMin denotes the minimum negative two’s complement number. Finally, you must use hexidecimal notation for your answers in the “Hex Representation” column, failure to do so will result in being marked incorrect on that portion of the question.

Expression	Decimal Representation	Hex Representation
Zero	0	**000**
–	-9	**3f7**
i	**-42**	**3d6**
i >> 5	**-2**	**3fe**
ui	**982**	**3d6**
(int) s	**-7**	**3f9**
(int)(s ˆ -12)	**13**	**00d**
(int) us	**25**	**019**
TMax	**511**	**1ff**
TMin	**-512**	**200**

Problem 2. (14 points):

Consider the following 11-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next $k = 4$ bits are the exponent. The exponent bias is 7.
- The last $n = 6$ bits are the significand.

Numeric values are encoded in this format as a value of the form $V = (-1)^s \times M \times 2^E$, where s is the sign bit, E is exponent after biasing, and M is the significand.

Part I

How many FP numbers are in the following intervals $[a, b)$?

For each interval $[a, b)$, count the number of x such that $a \leq x < b$.

A. Interval $[1, 2)$: 2^6**=64**

B. Interval $[2, 2.5)$: 2^4**=16**

Part II

Answer the following problems using either decimal (e.g., 1.375) or fractional (e.g., $11/8$) representations for numbers that are not integers.

A. For denormalized numbers:

(a) What is the smallest value E of the exponent after biasing? **-6**

(b) What is the largest value M of the significand? **63/64**

B. For normalized numbers:

(a) What is the smallest value E of the exponent after biasing? **-6**

(b) What is the largest value E of the exponent after biasing? **7**

(c) What is the smallest value M of the significand? **1**

(d) What is the largest value M of the significand? **127/64**

Part III

Fill in the blank entries in the following table giving the encodings for some interesting numbers.

Description	E	M	V	Binary Encoding
Zero	-6	0	0	`0 0000 000000`
Smallest Positive (nonzero)	**-6**	**1/64**	**1/4096**	**0 0000 000001**
Largest denormalized	**-6**	**63/64**	**63/4096**	**0 0000 111111**
Smallest positive normalized	**-6**	**1**	**1/64**	**0 0001 000000**
Positive Infinity	—	—	—	**0 1111 000000**
Negative Infinity	—	—	—	**1 1111 000000**

Problem 3. (12 points):

Consider the following assembly code:

```
08048333 <func>:
 8048333:       55                      push   %ebp
 8048334:       89 e5                   mov    %esp,%ebp
 8048336:       83 ec 0c                sub    $0xc,%esp
 8048339:       c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
 8048340:       8b 45 08                mov    0x8(%ebp),%eax
 8048343:       0f af 45 0c             imul   0xc(%ebp),%eax
 8048347:       39 45 fc                cmp    %eax,0xfffffffc(%ebp)
 804834a:       72 02                   jb     804834e <func+0x1b>
 804834c:       eb 2f                   jmp    804837d <func+0x4a>
 804834e:       8b 45 08                mov    0x8(%ebp),%eax
 8048351:       89 c2                   mov    %eax,%edx
 8048353:       0f af 55 0c             imul   0xc(%ebp),%edx
 8048357:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
 804835a:       01 10                   add    %edx,(%eax)
 804835c:       8b 45 08                mov    0x8(%ebp),%eax
 804835f:       48                      dec    %eax
 8048360:       89 44 24 04             mov    %eax,0x4(%esp,1)
 8048364:       8b 45 0c                mov    0xc(%ebp),%eax
 8048367:       03 45 08                add    0x8(%ebp),%eax
 804836a:       d1 e8                   shr    $0x1,%eax
 804836c:       89 04 24                mov    %eax,(%esp,1)
 804836f:       e8 bf ff ff ff          call   8048333 <func>
 8048374:       89 c2                   mov    %eax,%edx
 8048376:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
 8048379:       29 10                   sub    %edx,(%eax)
 804837b:       eb c3                   jmp    8048340 <func+0xd>
 804837d:       8b 45 fc                mov    0xfffffffc(%ebp),%eax
 8048380:       c9                      leave
 8048381:       c3                      ret
```

The assembly on the previous page corresponds to the C code below. Fill in the blanks in the C code to match the operations done in the assembly.

```
unsigned int func(unsigned int a, unsigned int b)
{
  unsigned int result = 0;
  while (result ____< a*b____) {
    result += _____a*b____;
    result -= func(___(a+b)/2__, _____a-1____);
  }
  return result;
}
```

Problem 4. (8 points):

Consider the following C declarations:

```
struct a_struct {
        char             a;
        struct a_strcut *b;
};

struct b_struct {
        char            c;
        int             i;
        double *        d;
        short           e[3];
        struct a_struct m;
};
```

A. Using the templates below (allowing a maximum of 32 bytes), indicate the allocation of data for `struct b_struct`. Mark off and label the areas for each individual element (arrays may be labeled as a single element). **Cross hatch the parts that are allocated, but not used, and be sure to clearly indicate the end of the structure. Assume the Linux alignment rules discussed in class.**

```
struct b_struct:

  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| c  X  X  X| i  i  i  i| d  d  d  d| e  e  e  e|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| e  e  X  X| a  X  X  X| b  b  b  b|           |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. How would you define the `struct b2_struct` structure to minimize the number of bytes allocated for the structure using the same fields as the `struct b_struct` structure?

```
struct b2_struct {

        char            c;
        short           e[3];
        int             i;
        double *        d;
        struct a_struct m;
```

(Most solutions with e and c adjacent work.)

```
};
```

C. What is the value of `sizeof(struct b2_struct)`? **24**

Problem 5. (12 points):

Consider the following C code:

```
struct triple
{
    int x;
    char c;
    int y;
};

int mystery1(int x);
int mystery2(int x);
int mystery3(struct triple* t);

int main()
{
    struct triple t = {35, ’q’, 10};

    int result1 = mystery1(42);
    int result2 = mystery2(19);
    int result3 = mystery3(&t);

    printf("result1 = %d\n", result1);
    printf("result2 = %d\n", result2);
    printf("result3 = %d\n", result3);

    return 0;
}
```

Using the assembly code for mystery1, mystery2, and mystery3 on the next page, fill in the proper values in this program’s output:

result1 = **3**

result2 = **19**

result3 = **350**

```
080483d0 <mystery1>:
 80483d0:       55                      push   %ebp
 80483d1:       89 e5                   mov    %esp,%ebp
 80483d3:       53                      push   %ebx
 80483d4:       8b 45 08                mov    0x8(%ebp),%eax
 80483d7:       89 c3                   mov    %eax,%ebx
 80483d9:       83 e3 01                and    $0x1,%ebx
 80483dc:       85 c0                   test   %eax,%eax
 80483de:       74 0b                   je     80483eb <mystery1+0x1b>
 80483e0:       c1 f8 01                sar    $0x1,%eax
 80483e3:       50                      push   %eax
 80483e4:       e8 e7 ff ff ff          call   80483d0 <mystery1>
 80483e9:       01 c3                   add    %eax,%ebx
 80483eb:       89 d8                   mov    %ebx,%eax
 80483ed:       8b 5d fc                mov    0xfffffffc(%ebp),%ebx
 80483f0:       c9                      leave
 80483f1:       c3                      ret

080483f4 <mystery2>:
 80483f4:       55                      push   %ebp
 80483f5:       89 e5                   mov    %esp,%ebp
 80483f7:       8b 55 08                mov    0x8(%ebp),%edx
 80483fa:       31 c0                   xor    %eax,%eax
 80483fc:       85 d2                   test   %edx,%edx
 80483fe:       7e 06                   jle    8048406 <mystery2+0x12>
 8048400:       40                      inc    %eax
 8048401:       4a                      dec    %edx
 8048402:       85 d2                   test   %edx,%edx
 8048404:       7f fa                   jg     8048400 <mystery2+0xc>
 8048406:       c9                      leave
 8048407:       c3                      ret

08048408 <mystery3>:
 8048408:       55                      push   %ebp
 8048409:       89 e5                   mov    %esp,%ebp
 804840b:       8b 45 08                mov    0x8(%ebp),%eax
 804840e:       8b 10                   mov    (%eax),%edx
 8048410:       0f af 50 08             imul   0x8(%eax),%edx
 8048414:       89 d0                   mov    %edx,%eax
 8048416:       c9                      leave
 8048417:       c3                      ret
```

Problem 6. (12 points):

This problem tests your understanding of byte ordering and the stack discipline. The following program reads a string from standard input and prints an integer in it's hexadecimal format based on the input it was given.

```
#include <stdio.h>

int get_key () {
    int key;
    scanf ("%s", &key);
    return key;
}

int main () {
    printf ("0x%8x\n", get_key());
    return 0;
}
```

Here is the corresponding machine code on a Linux/x86 machine:

```
08048414 <get_key>:
 8048414:   55                push   %ebp
 8048415:   89 e5             mov    %esp,%ebp
 8048417:   83 ec 18          sub    $0x18,%esp
 804841a:   83 c4 f8          add    $0xfffffff8,%esp
 804841d:   8d 45 fc          lea    0xfffffffc(%ebp),%eax
 8048420:   50                push   %eax       address arg for scanf
 8048421:   68 b8 84 04 08    push   $0x80484b8  format string for scanf
 8048426:   e8 e1 fe ff ff    call   804830c <_init+0x50>  call scanf
 804842b:   8b 45 fc          mov    0xfffffffc(%ebp),%eax
 804842e:   89 ec             mov    %ebp,%esp
 8048430:   5d                pop    %ebp
 8048431:   c3                ret

08048434 <main>:
 8048434:   55                push   %ebp
 8048435:   89 e5             mov    %esp,%ebp
 8048437:   83 ec 08          sub    $0x8,%esp
 804843a:   83 c4 f8          add    $0xfffffff8,%esp
 804843d:   e8 d2 ff ff ff    call   8048414 <get_key>
 8048442:   50                push   %eax       integer arg of printf
 8048443:   68 bb 84 04 08    push   $0x80484bb  format string for printf
 8048448:   e8 ef fe ff ff    call   804833c <_init+0x80>  call printf
 804844d:   31 c0             xor    %eax,%eax
 804844f:   89 ec             mov    %ebp,%esp
 8048451:   5d                pop    %ebp
 8048452:   c3                ret
```

Here are a few notes to help you with the problem:

- `scanf (‘‘%s’’, i)` reads a string from the standard input stream and stores it at stores it at address `i` (including the terminating `’\0’` character. It does **not** check the size of the destination buffer.
- `printf ("0x%8x\n", j)` prints 8 digits of the integer `i` in hexadecimal format as `0x`$xxxxxx$
- Recall that Linux/x86 machines are Little Endian.
- You will need to know the hex values for the following characters:

Character	Hex Value	Character	Hex Value
`’E’`	`0x45`	`’s’`	`0x73`
`’v’`	`0x76`	`’h’`	`0x68`
`’i’`	`0x69`	`’!’`	`0x21`
`’l’`	`0x6c`	`’\0’`	`0x00`

A. Suppose we run this program on a Linux/x86 machine with the input string “`Evil!`”.

Here is a template for the stack, showing the location of `key`. Indicate with a labelled arrow where `%ebp` points to, and fill in the stack with the values that were just read in *after* the call to scanf (**addresses increase from left to right**).

```
                    |<------ key ------>|
+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    | 45 | 76 | 69 | 6c | 21 | 00 |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+
                                        ^
                                      %ebp
```

What is the 4-byte integer (in hex) printed by `printf` inside main?

0x **6c697645**

B. Suppose we instead gave it the input string “`Evillish!`”.

For the remaining problems, each answer should be an unsigned 4-byte integer expressed as 8 hexadecimal digits.

(a) What is the value of `(&key)[1]` just **after** `scanf` returns to `get_key`?

`(&key)[1] = 0x` **6873696c**

(b) What is the value of `%ebp` immediately **before** the execution of the `ret` instruction of `get_key`?

`%ebp = 0x` **6873696c**

You can use the following template of the stack as *scratch space*. This **will not** be considered for credit.

```
+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+
```

Problem 7. (6 points):

Consider the following code for a matrix multiplication function:

```
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<m; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

with matricies `a[n][m]`, `b[m][n]` and `c[n,n]`.

1. Assume that `m` is twice as large as `n`. Is the above loop optimally arranged for preserving locality? If not, state the optimal nesting of these loops.

No,

Nest them as

```
    for ( ... i ...)
      for ( ... k ... )
        for ( ... j ... )
```

see class13, slides 30-40

2. Assume that `n` is twice as large as `m`. Is the above loop optimally arranged for preserving locality? If not, state the optimal nesting of these loops.

No,

Nest them as

```
    for ( ... i ...)
      for ( ... k ... )
        for ( ... j ... )
```

see class13, slides 30-40

Problem 8. (7 points):

Answer **true** or **false** for each of the statements below. For full credit your answer must be correct and you must write the entire word (either **true** or **false** in the answer space.

1. If it takes less time to do either a cache access or a main memory access than to perform a branch, then loop unrolling always improves performance. **false**

2. On the Fish machines the most effective way to time a short routine is to use the cycle counter. **true**

3. Loop invariant code motion can be applied to expressions involving the loop induction variable **false**

4a.
```
typedef int (*a)(int *);
typedef a b[10];
typedef b* (*c)();
c d;
int *(*(*)(int *))[10] (*e)();
```
`d` and `e` have the same type. **either**

4b. `e` and `c` have the same type. **false**

5. In C, the variable `f` is declared as: `int f[12][17]`. The address for f[3][8] can sometimes be greater than the address for f[8][3]. **false**

6. The largest possible finite denormalized IEEE floating number is greater than the smallest possible positive normalized IEEE floating number: **false**

15-213 Introduction to Computer Systems

Exam 1

February 22, 2005

Name: ______________________________

Andrew User ID: ____________________

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)
        fmuls   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.

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?

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

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

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.

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 `exp`n 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	
n	
y	

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

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

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;
```

1. (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

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

2. (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.

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

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;
     }
   }
   ```

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)++;
       ______________;
     }
   }
   ```

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.

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.

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.

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.

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.

1. (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	10	9	8	7	6	5	4	3	2	1	0

2. (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 infered from the information in the cache.

Operation	Address	Hit?	Read Value (or Unknown)
Read	0x834		
Write	0x836		(not applicable)
Read	0xFFD		

15-213 Introduction to Computer Systems

Exam 1

February 28, 2006

Name: ______________________

Andrew User ID: ______________________

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`		
If `x > 0` then `x * x > 0`		
`u >= 0`		
`u <= -1`		
If `x > y` then `-x < -y`		
If `u > v` then `-u > -v`		
If `x >= 0` then `-x <= 0`		
If `x < 0` then `-x > 0`		
`(unsigned) x == x`		
`(x << 1) >> 1 == x`		

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
	1.00001×2^7	`0x43040000`
$-1\frac{1}{2}$		
$\frac{1}{2}$		
$\frac{3}{8}$		`0x3EC00000`
2^{-149}	1.0×2^{-149}	

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	
&c.dirty	
&c.tag	
&c.block	
&c.stamp	

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)	
&c.valid == &c.dirty	
&c.block[32] == &c.stamp	
*&c.dirty == c.dirty	
&*c.block == c.block	

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 ________ way associative,

    where each cache block contains ________ bytes. In total,

    the cache holds ________ bytes. The simulated main

    memory holds ________ bytes and addresses may

    be ________ 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 = ___________________________ ;  /* obtain tag */

  int i = ________________________ ;       /* obtain set index */

  int offset = __________________________; /* obtain block offset */

  for (j = 0; j < 4; j++) {

    if (_____________________________________________) {
      /* cache hit */
      cache[i][j].stamp = counter++;

      return _____________________________________________ ;
    }
  }
  /* cache miss */
  j = lru(i);            /* get LRU cache line j in set i */

  copy_block(________________________________, 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 ___________________________________________ ;
}
```

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++)

      _____________________________;
}
```

```
reset:
        xorl    %ecx, %ecx
.L9:
        movslq  %ecx,%rax

        movl    ____, %edx

        imulq   ____, %rax, %rax
.L8:
        movb    $0, cache(%rax)

        _________________________
        decl    %edx
        jns     .L8
        incl    %ecx

        _________________________
        jle     .L9
        ret
```

6. Optimization (20 points)

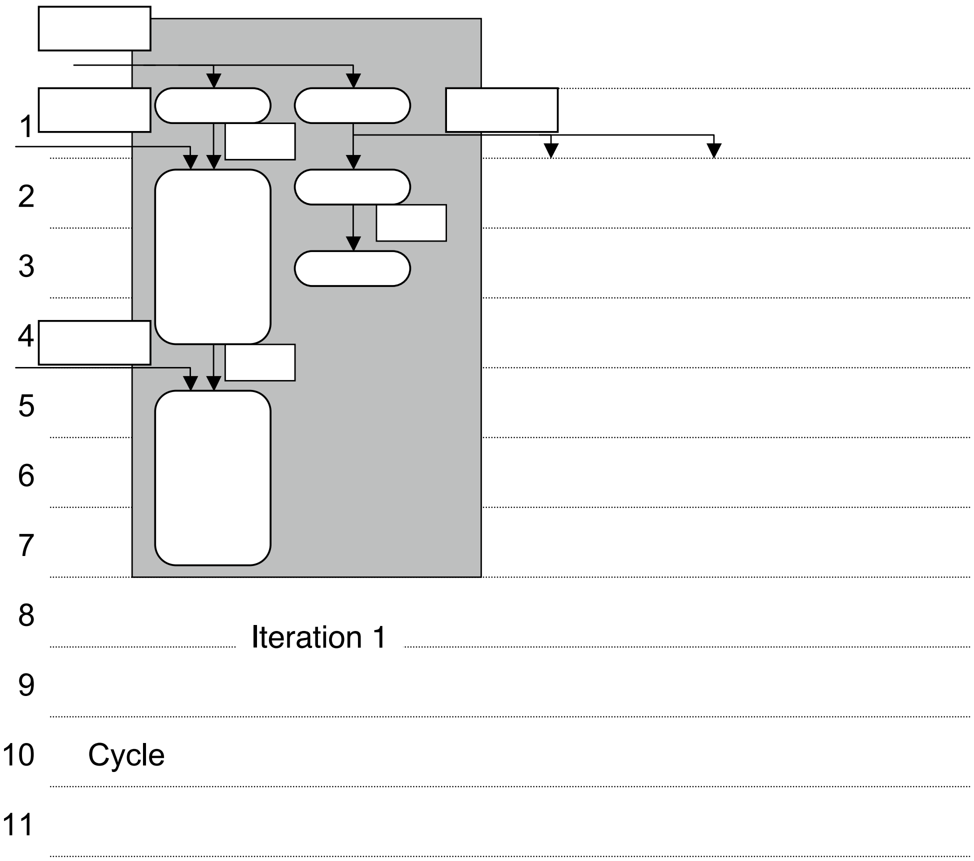
Consider the following version of the `copy_block` function that we assumed in Problem 4 for copying a cache block to and from memory. Recall that `%al` represents the lowest byte of the `%rax` register.

```
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

incl    %ecx

movzbl  (%rdx,%rdi), %eax

cmpl    $31, %ecx

movb    %al, (%rdx,%rsi)

jle     .L19
```

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.

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.

15-213 Introduction to Computer Systems

Exam 1

February 27, 2007

Name:

Andrew User ID:

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**
Integers	1	**10**	
Floating Point	2	**15**	
Assembly Language	3	**15**	
Calling Conventions	4	**10**	
Structures and Alignment	5	**10**	
Out-of-Order Execution	6	**15**	
	Total	**75**	

1. Integers (10 points)

For each of the following propositions, write in **all** comparisons that make it **always** true among the four possibilities:

```
<  >  ==  !=
```

If none are guaranteed to hold, please indicate that explicitly by marking it with an **X**. We have filled in the first two for you as examples. Assume the variables are declared with

```
int x,y;
```

and initialized to some unknown values. You may assume that `int`'s are 32 bits wide, `char`'s are 8 bits wide and that right shift is arithmetical on signed numbers and logical on unsigned numbers.

If `x > y` then `y`	`<`
	`!=`
		`x`

If `x < 0` then `x+1`	**X**	`0`
If `x > 0` then `x+1`		`0`
If `(x >> 31) < 0` then `x`		`0`
If `((x << 31) >> 31) < 0` then `x & 1`		`0`
If `x > y` then `(unsigned)x`		`y`
If `((unsigned char)x >> 1) < 64` then `(char)x`		`0`

2. Floating Point (15 points)

1. (10 pts) Fill in the blank entries in the following table. We assume an IEEE representation of floating point numbers with 1 sign bit, $k = 3$ bits for the exponent and $n = 4$ bits for the fractional value. This means the bias is $2^{3-1} - 1 = 3$.

   We are only considering positive numbers.

Value	Form $M \times 2^E$ for $1 \leq M < 2$	Hexadecimal representation	Decimal fraction
Smallest denorm.		`0x01`	
Smallest norm.			
One	1.0×2^0		1
Largest norm.			
Infinity	XXXXX		XXXXX

2. (5 pts) With standard single precision floating point numbers with 1 sign bit, $k = 8$ bits for the exponent, and $n = 23$ bits for the significand, what is the smallest positive value for `n` declared with `int n;` such that

   `(int)(float)n != n`?

3. Assembly Language (15 points)

Consider the following recursive C program that sorts a segment of an integer array into ascending order in place.

```
/* qsort(A, low, high) sorts subarray A[low]..A[high] */
void qsort (int* A, int low, int high) {
  int pivot, i, k;
  if (low >= high) return;
  pivot = A[high];
  i = low;
  k = high;
  while (i < k) {
    if (A[i] < pivot) {
      i++;
    } else {
      A[k] = A[i];
      k--;
      A[i] = A[k];
    }
  }
  A[k] = pivot;
  qsort(A, low, k-1);
  qsort(A, k+1, high);
}
```

The `while` loop is compiled to the following assembly language instructions. **Hint:** The entry point of the loop is at label `.L8`. **Hint:** Do not try to fill in the missing instructions until you have answered questions 1 and 2 on the next page.

```
.L13:
        incl    %edx
        jmp     .L8
.L7:
        movslq  %edx, %rcx
        movl    (%rbp,%rcx,4), %eax
        cmpl    %r8d, %eax
        jl      .L13
        movl    %eax, (%rbp,%rdi,4)
        decl    %ebx

        ___________________________ <first missing instruction>

        ___________________________ <second missing instruction>
        movl    %eax, (%rbp,%rcx,4)
.L8:
        cmpl    %ebx, %edx
        jl      .L7
```

1. (5 pts) Complete the following table associating C variable or expressions with registers.

C Expression	Register
`pivot`	`%r8d`
`A`	
`i`	
`k`	
`(long)k`	
`(long)i`	

2. (5 pts) The register `%eax` holds temporary values during the loop computation. List all the expressions in the original C program whose values it holds.

3. (5 pts) Fill in the two missing instructions.

4. Calling Conventions (10 points)

After the code of the `while` loop shown before, we find the following instructions to implement the first recursive call.

```
movl    %r8d, (%rbp,%rdi,4)
leal    -1(%rbx), %edx
movq    %rbp, %rdi
call    qsort
```

1. (5 pts) By the x86-64 calling conventions, which of the mentioned registers are guaranteed to have the same value when `qsort` returns as right before the call? Write **yes** or **no** into the space.

Register	Same?
%r8d	
%rbp	
%rdi	
%rbx	
%edx	

We are surprised that following the first recursive call, we find no further recursive call, but instead the instructions

```
leal    1(%rbx), %esi
jmp     .L14
```

where `.L14` is a label near the beginning of the `qsort` procedure. The compiler used an optimization to eliminate the second recursive call in favor of a jump instruction.

2. (5 pts) Complete the following rendering of the compiler's optimization in C in the form of a new `while` loop.

```
void qsort (int* A, int low, int high) {
 int pivot, i, k;
 /* eliminated here: if (low >= high) return; */
 while (___________) {
   pivot = A[high];
   i = low;
   k = high;
   while (i < k) { ... as before ... }
   A[k] = pivot;
   qsort(A, low, k-1);

   ___________________   /* was: qsort(A, k+1, high); */
 }
}
```

5. Structures and Alignment (10 points)

This question concerns alignment on an x86-64 architecture and pointer arithmetic. Consider the following C structure declaration.

```
struct box {
  int tag; /* 0 = int, 1 = long, 2 = float, 3 = double */
  union {
    int i;
    long l;
    float f;
    double d;
  } data;
};
```

1. (3 pts) What is the total size of a `box` structure on an x86-64, expressed in bytes?

2. (2 pts) An element of type `struct box` must be aligned at 0 modulo __________.

3. (5 pts) Write a C function `int high4 (long x);` that extracts the high 4 bytes of a `long` as an `int`. For example, `high (0x1234567890abcdef)` should return `0x12345678`. Use a `box` structure and pointer arithmetic. Your code may ignore the `tag` field.

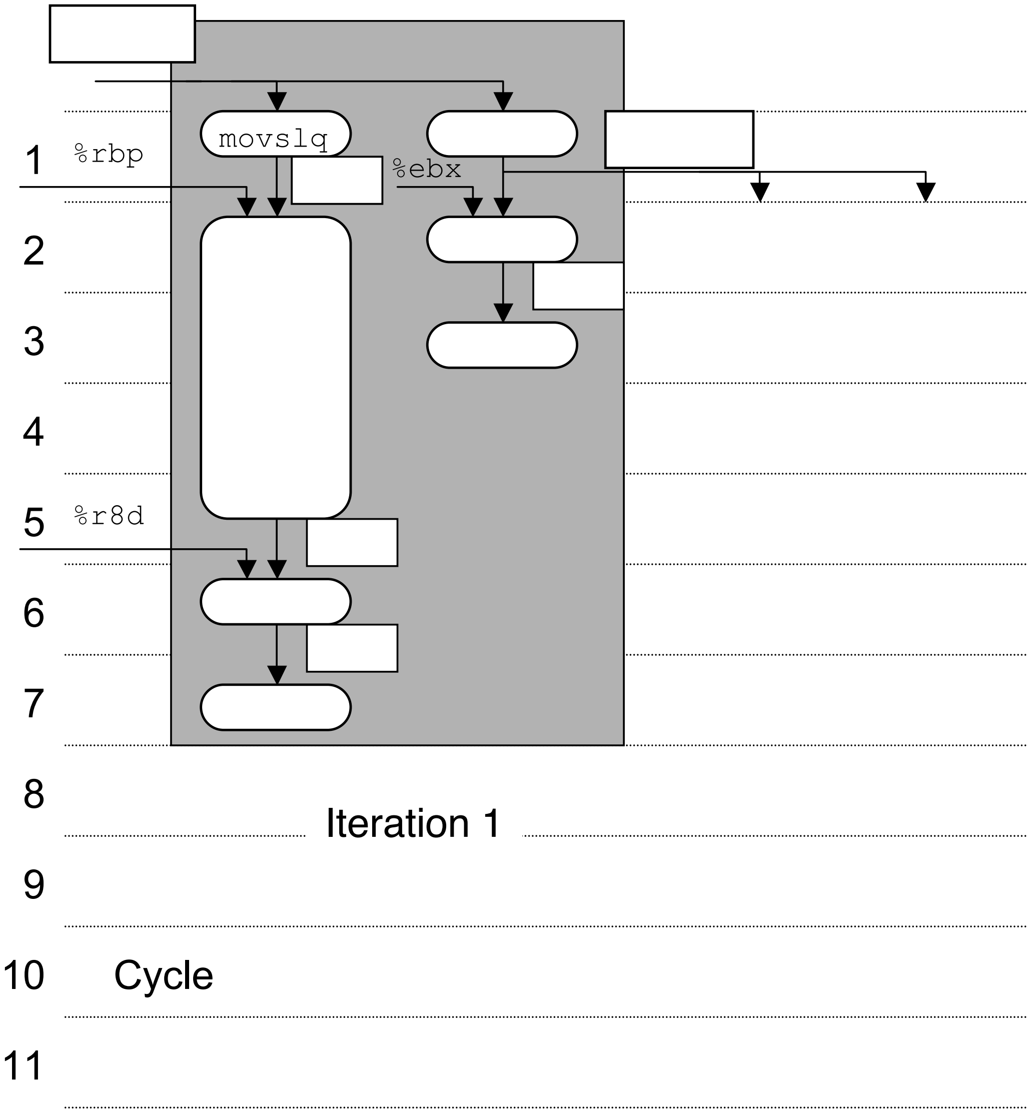
6. Out-of-Order Execution (15 points)

We now return to the earlier quicksort procedure. If the array is already sorted, the first conditional branch instruction in the code fragment below will always be taken.

1. (5 pts) Assume that the processor correctly predicts this, as well as the second branching instruction (for example, because they both go backwards). On the right-hand side, show the translation of the assembly language instructions into execution unit operations with register renaming. Do not rename registers that are invariant throughout multiple loop iterations. We have already filled in the translations of the conditional branches; the unconditional jump is handled by the instruction fetch unit. Consider `.L7` as the starting point of an iteration.

```
.L13:                                 |
        incl    %edx                  |
        jmp     .L8                   | (handled in fetch unit)
.L7:                                  |
        movslq  %edx, %rcx            |
        movl    (%rbp,%rcx,4), %eax   |
        cmpl    %r8d, %eax            |
        jl      .L13                  | jl-taken cc.1a
        <omitted code>                |
.L8:                                  |
        cmpl    %ebx, %edx            |
        jl      .L7                   | jl-taken cc.1b
```

2. (7 pts) Complete the labeling in the following timed data dependency diagram. We have already filled in the registers that are not renamed and one register move operation.



3. (3 pts) What is the theoretical CPE for this loop, assuming perfect branch prediction and no resource limitions?

Worksheet

Worksheet

Worksheet

Andrew login ID:______________________________________

Full Name:______________________________________

Recitation Section:______________________________________

CS 15-213, Spring 2008
Exam 1

Tue. February 26, 2008

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–H) 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 70 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. You may use any books or notes you like. No calculators or other electronic devices are allowed.
- Good luck!

1 (8):	
2 (8):	
3 (10):	
4 (9):	
5 (8):	
6 (8):	
7 (11):	
8 (8):	
TOTAL (70):	

Problem 1. (8 points):

For this problem, assume the following:

- We are running code on a 6-bit machine using two's complement arithmetic for signed integers.
- `short` integers are encoded using 3 bits.
- Sign extension is performed whenever a `short` is casted to an `int`
- Right shifts of `int`s are arithmetic.

Fill in the empty boxes in the table below. The following definitions are used in the table:

```
int a = -29;
short b = (short)a;
unsigned ua = a;
int x = -21;
short y = (short)x;
unsigned ux = x;
```

Note: You need not fill in entries marked with "—".

Expression	Decimal Representation	Binary Representation
—	27	
—		100100
x		
y		
ux		
a >> 2		
ua >> 2		
b << 1		
−TMin		

Problem 2. (8 points):

Part A

Fill in the blanks in the table below with the number described in the first column of each row. You can give your answers as unexpanded simple arithmetic expressions (such as $15^{213} + 42$); you should not have trouble fitting your answers into the space provided.

Description	Number
`int x=1; float *f = (float *)&x;` What is the value of `*f`?	
`int x=-1; float *f = (float *)&x;` What is the value of `*f`?	
Smallest positive integer that cannot be represented as a 32-bit `float`	

Part B

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 definition:

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

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 argument values, or
- Give an example where it is not true.

Puzzle	True / Counterexample
`x < 0` $\Rightarrow$ `-x > 0`	
`x ^ ~x < 0`	
`(x ^ (x >> 31)) + 1 > 0`	
`(((!!x) << 31) >> 31) & x == x`	

Problem 3. (10 points):

Consider an 8-bit IEEE floating-point representation with:

- 1 sign bit
- 3 exponent bits (therefore the bias $B = 2^{3-1} - 1 = 3$)
- 4 mantissa bits

A. Fill in the blanks in the following table. Express numerical values as fractions (e.g., $277/512$).

Number	**Bit representation**
$3/8$	
$-\infty$	
$9/2$	
	`0 010 0101`
	`1 000 1000`
	`0 111 0010`

B. Give the bit representation and numerical value of the largest number representable in this format as a *denormalized* floating-point number.

C. Give the bit representation and numerical value of the largest number representable in this format as a *normalized* floating-point number.

Problem 4. (9 points):

Consider the following x86-64 assembly code:

```
# on entry: %rdi = n, %rsi = A
000000000040056e <bar>:
  40056e:       41 b8 00 00 00 00       mov    $0x0,%r8d
  400574:       41 b9 00 00 00 00       mov    $0x0,%r9d
  40057a:       41 39 f9                cmp    %edi,%r9d
  40057d:       7d 30                   jge    4005af <bar+0x41>
  40057f:       ba 00 00 00 00          mov    $0x0,%edx
  400584:       39 fa                   cmp    %edi,%edx
  400586:       7d 1f                   jge    4005a7 <bar+0x39>
  400588:       49 63 c1                movslq %r9d,%rax
  40058b:       48 8b 0c c6             mov    (%rsi,%rax,8),%rcx
  40058f:       48 63 c2                movslq %edx,%rax
  400592:       8b 04 81                mov    (%rcx,%rax,4),%eax
  400595:       41 0f af c1             imul   %r9d,%eax
  400599:       0f af c2                imul   %edx,%eax
  # Instruction "cltq" is equivalent to "movslq %eax, %rax"
  40059c:       48 98                   cltq
  40059e:       49 01 c0                add    %rax,%r8
  4005a1:       ff c2                   inc    %edx
  4005a3:       39 fa                   cmp    %edi,%edx
  4005a5:       7c e8                   jl     40058f <bar+0x21>
  4005a7:       41 ff c1                inc    %r9d
  4005aa:       41 39 f9                cmp    %edi,%r9d
  4005ad:       7c d0                   jl     40057f <bar+0x11>
  4005af:       4c 89 c0                mov    %r8,%rax
  4005b2:       c3                      retq
```

Fill in BOTH of the blanks below for the corresponding C code.

```
long bar(int n, ________________ A) {         // Fill in type of A
    long sum = 0;
    int i,j;
    for (i = 0; i < n; i++) {
        for(j = 0; j < n; j++)

            sum += _____________________;   // Fill in expression
    }
    return sum;
}
```

Problem 5. (8 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 following x86-64 assembly code:

```
# On entry:
#    %edi = x
#    %esi = y
#
copy_array:
    movslq  %edi,%rdi
    movslq  %esi,%rsi
    movq    %rdi, %rax
    leaq    (%rsi,%rsi,8), %rdx
    salq    $5, %rax
    subq    %rdi, %rax
    addq    %rdi, %rdx
    leaq    (%rsi,%rax,2), %rax
    movl    array1(,%rax,4), %eax
    movl    %eax, array2(,%rdx,4)
    movl    $1, %eax
    ret
```

What are the values of H and J?

H =

J =

Problem 6. (8 points):

Consider the following data structure declaration:

```
struct node{
  char x;
  int array[2];
  int idx;
  struct node *next;
};
```

Below are given four C functions and four x86-64 code blocks.

```
char *mon(struct node *ptr){
  return &ptr->x;
}
```

```
int tue(struct node *ptr){
  return ptr->array[ptr->idx];
}
```

```
void wed(struct node *ptr){
  ptr->idx = ptr->array[1];
  return;
}
```

```
char thu(struct node *ptr){
  ptr = ptr->next;
  return ptr->x;
}
```

```
A  movl  8(%rdi), %eax
   movl  %eax, 12(%rdi)
```

```
B  movq  %rdi, %rax
```

```
C  movq  16(%rdi), %rax
   movsbl  (%rax),%eax
```

```
D  movslq  12(%rdi),%rax
   movl  4(%rdi,%rax,4), %eax
```

In the following table, next to the name of each x86-64 code block, write the name of the C function that it implements.

Code Block	Function Name
A	
B	
C	
D	

Problem 7. (11 points):

The next problem concerns code generated by GCC for a function involving a switch statement. The code uses a jump to index into the jump table:

```
  400518: ff 24 d5 40 06 40 00  jmpq   *0x400640(,%rdx,8)
```

Using GDB, we extract the 8-entry jump table as:

```
0x400640:    0x0000000000400527
0x400648:    0x0000000000400525
0x400650:    0x0000000000400531
0x400658:    0x000000000040051f
0x400660:    0x0000000000400525
0x400668:    0x0000000000400525
0x400670:    0x000000000040052a
0x400678:    0x0000000000400531
```

The following block of disassembled code implements the branches of the switch statement:

```
# on entry: %rdi = a, %rsi = b, %rdx = c
  400510: 31 c0                 xor    %eax,%eax
  400512: 48 83 fa 07           cmp    $0x7,%rdx
  400516: 77 0d                 ja     400525 <_Z4testlll+0x15>
  400518: ff 24 d5 40 06 40 00  jmpq   *0x400640(,%rdx,8)
  40051f: 48 89 f0              mov    %rsi,%rax
  400522: 48 29 f8              sub    %rdi,%rax
  400525: f3 c3                 repz retq # repz is a no-op here
  400527: 48 01 f7              add    %rsi,%rdi
  40052a: 48 89 f8              mov    %rdi,%rax
  40052d: 48 31 f0              xor    %rsi,%rax
  400530: c3                    retq
  400531: 48 8d 46 2a           lea    0x2a(%rsi),%rax
  400535: c3                    retq
```

Fill in the blank portions of C code below to reproduce the function corresponding to this object code. You can assume that the first entry in the jump table is for the case when `c` equals 0.

```
long test(long a, long b, long c)
{
  long answer = _____;
  switch(c)
  {
    case ___:
      answer = _____;
      break;
    case ___:
    case ___:
      answer = _____;
      break;
    case ___:
      a = _____;
      /* Fall through */
    case ___:
      answer = _____;
      break;
    default:
      answer = _____;
  }

  return answer;
}
```

Problem 8. (8 points):

```
struct BOOKLIST {
    char a;
    short US;
    char b;
    short CA;
    char c;
    double EU;
    char d;
    int UK;
} booklist;
```

A. Show how the struct above would appear on a 32 bit Windows machine (primitives of size k are k byte aligned). Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks to indicate bytes that are allocated in the struct but are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. Rearrange the above fields in `booklist` to conserve the most space in the memory below. Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks to indicate bytes that are allocated in the struct but are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

C. How many bytes of the struct are wasted in part A?

D. How many bytes of the struct are wasted in part B?

Andrew login ID:______________________________________

Full Name:______________________________________

Recitation Section:______________________________________

CS 15-213, Spring 2009
Exam 1

Tuesday, February 24, 2009

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–J) on the front.
- Write your answers in the space provided for 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. You may use any books or notes you like. No calculators or other electronic devices are allowed.
- Good luck!

1 (16):
2 (22):
3 (13):
4 (13):
5 (22):
6 (14):
TOTAL (100):

Problem 1. (16 points):

Consider a new floating point format that follows the IEEE spec you should be familiar, except with 3 exponent bits and 2 fraction bits (and 1 sign bit). Fill in all blank cells in the table below. *I*f, in the process of converting a decimal number to a float, you have to round, write the rounded value next to the original decimal as well.

Description	**Decimal**	**Binary Representation**
Bias		-----
Smallest positive number		
Lowest finite		
Smallest positive normalized		
-----	$-\frac{7}{16}$	
-----	$\frac{5}{4}$	
-----		`1 010 01`
-----	13	

Problem 2. (22 points):

Consider the C code written below and compiled on a 32-bit Linux system using GCC.

```
struct s1
{
  short x;
  int y;
};

struct s2
{
  struct s1 a;
  struct s1 *b;
  int x;
  char c;
  int y;
  char e[3];
  int z;
};

short fun1(struct s2 *s)
{
  return s->a.x;
}

void *fun2(struct s2 *s)
{
  return &s->z;
}

int fun3(struct s2 *s)
{
  return s->z;
}

short fun4(struct s2 *s)
{
  return s->b->x;
}
```

a) What is the size of `struct s2`?

b) How many bytes are wasted for padding?

You may use the rest of the space on this page for scratch space to help with the rest of this problem. Nothing written below this line will be graded.

c) Which of the following correspond to functions `fun1`, `fun2`, `fun3`, and `fun4`?

```
push   %ebp
mov    %esp,%ebp
mov    0x8(%ebp),%eax
add    $0x1c,%eax
pop    %ebp
ret
```

ANSWER: ________

```
push   %ebp
mov    %esp,%ebp
mov    0x8(%ebp),%eax
mov    0x8(%eax),%eax
movswl (%eax),%eax
pop    %ebp
ret
```

ANSWER: ________

```
push   %ebp
mov    %esp,%ebp
mov    0x8(%ebp),%eax
mov    0x1c(%eax),%eax
pop    %ebp
ret
```

ANSWER: ________

```
push   %ebp
mov    %esp,%ebp
mov    0x8(%ebp),%eax
movswl (%eax),%eax
pop    %ebp
ret
```

ANSWER: ________

d) Assume a variable is declared as `struct s2 myS2;` and the storage for this variable begins at address `0xbfb2ffc0`.

```
(gdb) x/20w &myS2
0xbfb2ffc0: 0x0000000f  0x000000d5  0xbfb2ffe8  0x00000000
0xbfb2ffd0: 0xb7f173ff  0x0000012c  0xbf030102  0x0000000c
0xbfb2ffe0: 0xb7e2dfd5  0xb7f15ff4  0xbfb30012  0x000000f3
0xbfb2fff0: 0xb7e2e0b9  0xb7f15ff4  0xbfb30058  0xb7e1adce
0xbfb30000: 0x00000001  0xbfb30084  0xbfb3008c  0xbfb30010
```

Fill in all the blanks below.

HINTS: Label the fields. Not all 20 words are used. Remember endianness!

What would be returned by:

`fun1(&myS2) = 0x`______________

`fun2(&myS2) = 0x`______________

`fun3(&myS2) = 0x`______________

`fun4(&myS2) = 0x`______________

What is the value of:

`myS2.b->y = 0x`______________

`myS2.a.y  = 0x`______________

`myS2.z    = 0x`______________

`myS2.e[1] = 0x`______________

Problem 3. (13 points):

Given the memory dump and disassembly from GDB on the next page, fill in the C skeleton of the function switchfn:

```
int switchfn(int a, long b) {

  int y = 0, x = ____________;
  switch (a * b) {

  case 1:
    return 24;

  case 6:

    a = _______________;
    return a;

  case 0:
    return a + b;

  case 4:
    x = a;
    y *= b;
    break;

  case ____:
    a = y == x;

  case 3:

    b = y ____ x;

  case 5:

    return a ____ b;
  }

  return x == y;
}
```

There may be a few instructions you haven't seen before in this assembly dump. data16 is functionally equivalent to nop. set*cc* functions similarly to j*cc* except it will set its operand to 1 or 0 instead of jumping or not jumping, respectively. cqto is the 64-bit equivalent of cltd.

```
(gdb) x/7xg 0x4005c0
0x4005c0 <_IO_stdin_used+8>:    0x00000000004004a1      0x0000000000400494
0x4005d0 <_IO_stdin_used+24>:   0x00000000004004ac      0x00000000004004b4
0x4005e0 <_IO_stdin_used+40>:   0x00000000004004a5      0x00000000004004bc
0x4005f0 <_IO_stdin_used+56>:   0x000000000040049a

0x0000000000400476 <switchfn+0>:        mov    $0x0,%ecx
0x000000000040047b <switchfn+5>:        mov    $0xdeadbeef,%edx
0x0000000000400480 <switchfn+10>:       movslq %edi,%rax
0x0000000000400483 <switchfn+13>:       imul   %rsi,%rax
0x0000000000400487 <switchfn+17>:       cmp    $0x6,%rax
0x000000000040048b <switchfn+21>:       ja     0x4004c5 <switchfn+79>
0x000000000040048d <switchfn+23>:       jmpq   *0x4005c0(,%rax,8)
0x0000000000400494 <switchfn+30>:       mov    $0x18,%eax
0x0000000000400499 <switchfn+35>:       retq
0x000000000040049a <switchfn+36>:       lea    (%rdx,%rsi,4),%eax
0x000000000040049d <switchfn+39>:       data16
0x000000000040049e <switchfn+40>:       data16
0x000000000040049f <switchfn+41>:       nop
0x00000000004004a0 <switchfn+42>:       retq
0x00000000004004a1 <switchfn+43>:       lea    (%rdi,%rsi,1),%eax
0x00000000004004a4 <switchfn+46>:       retq
0x00000000004004a5 <switchfn+47>:       mov    %edi,%edx
0x00000000004004a7 <switchfn+49>:       imul   %esi,%ecx
0x00000000004004aa <switchfn+52>:       jmp    0x4004c5 <switchfn+79>
0x00000000004004ac <switchfn+54>:       cmp    %edx,%ecx
0x00000000004004ae <switchfn+56>:       sete   %al
0x00000000004004b1 <switchfn+59>:       movzbl %al,%edi
0x00000000004004b4 <switchfn+62>:       cmp    %edx,%ecx
0x00000000004004b6 <switchfn+64>:       setl   %al
0x00000000004004b9 <switchfn+67>:       movzbl %al,%esi
0x00000000004004bc <switchfn+70>:       movslq %edi,%rax
0x00000000004004bf <switchfn+73>:       cqto
0x00000000004004c1 <switchfn+75>:       idiv   %rsi
0x00000000004004c4 <switchfn+78>:       retq
0x00000000004004c5 <switchfn+79>:       cmp    %ecx,%edx
0x00000000004004c7 <switchfn+81>:       sete   %al
0x00000000004004ca <switchfn+84>:       movzbl %al,%eax
0x00000000004004cd <switchfn+87>:       retq
```

Problem 4. (13 points):

The function below is hand-written assembly code for a sorting algorithm. Fill in the blanks on the next page by converting this assembly to C code.

```
.globl mystery_sort      # exports the symbol so other .c files
                         # can call the function

mystery_sort:
        jmp     loop1_check

loop1:
        xor     %rdx, %rdx
        mov     %rsi, %rcx
        jmp     loop2_check

loop2:
        mov     (%rdi, %rcx, 8), %rax
        cmp     %rax, (%rdi, %rdx, 8)
        jg      loop2_check
        mov     %rcx, %rdx

loop2_check:
        dec     %rcx
        test    %rcx, %rcx
        jnz     loop2

        dec     %rsi
        mov     (%rdi, %rsi, 8), %rax
        mov     (%rdi, %rdx, 8), %rcx
        mov     %rcx, (%rdi, %rsi, 8)
        mov     %rax, (%rdi, %rdx, 8)

loop1_check:
        test    %rsi, %rsi
        jnz     loop1

        ret
```

```
void mystery_sort (long* array, long len)
{
  long a, b, tmp;

  while (_____ > _____)
  {

    a = _____;

    for (b = _____; b > _____; b--)
    {

      if (array[_____] > array{_____])
      {

         _____ = _____;
      }
    }

    len--;

    tmp = array[_____];

    array[_____] = array[_____];

    array[_____] = tmp;
  }
}
```

Problem 5. (22 points):

Circle the correct answer.

1. What sequence of operations does the leave instruction execute?

   (a)
   ```
   mov %ebp,%esp
   pop %ebp
   ```
   (b)
   ```
   pop %ebp
   mov %ebp,%esp
   ```
   (c)
   ```
   pop %esp
   mov %ebp,%esp
   ```
   (d)
   ```
   push %ebp
   mov %esp,%ebp
   ```

2. Who is responsible for storing the return address of a function call?

   (a) the caller

   (b) the callee

   (c) the kernel

   (d) the CPU

3. On what variable types does C perform logical right shifts?

   (a) signed types

   (b) unsigned types

   (c) signed and unsigned types

   (d) C does not perform logical right shifts

4. What is the difference between the rbx and the ebx register on an x86_64 machine?

   (a) nothing, they are the same register

   (b) ebx refers to only the low order 32 bits of the rbx register

   (c) they are totally different registers

   (d) ebx refers to only the high order 32 bits of the rbx register

5. Which of the following is the name for the optimization performed when you pull code outside of a loop?

   (a) code motion

   (b) loop expansion

   (c) dynamic programming

   (d) loop unrolling

6. On 32-bit x86 systems, where is the value of %ebp saved in relation to the current value of %ebp? (Assume a pointer size of 32 bits.)

   (a) there is no relation between where the current base pointer and old base pointer are saved.
   (b) old ebp = (ebp - 4)
   (c) old ebp = (ebp + 4)
   (d) old ebp = (ebp)

7. Which of the following mov instructions is invalid?

   (a) `mov %esp, %ebp`
   (b) `mov $0xdeadbeef, %eax`
   (c) `mov (0xdeadbeef), %esp`
   (d) `mov $0xdeadbeef, 0x08048c5f`
   (e) `mov %ebx, 0x08048c5f`

8. In C, the result of shifting a value by greater than its type's width is:

   (a) illegal
   (b) undefined
   (c) 0
   (d) Encouraged by the C1x standard.

9. Extending the stack can by done by

   (a) swapping the base pointer and the stack pointer
   (b) subtracting a value from your stack pointer
   (c) adding a value to your stack pointer
   (d) executing the ret instruction

10. 64-bit systems can support 32-bit assembly code

    (a) TRUE
    (b) FALSE

11. Assuming the register %rbx contains the value 0xfaaafbbbfcccfddd, which instruction would cause the register %rdi to contain the value 0x00000000fcccfddd?

    (a) `movl %ebx, %rdi`
    (b) `movslq %ebx, %rdi`
    (c) `movzlq %ebx, %rdi`
    (d) `lea %ebx, %rdi`

Problem 6. (14 points):

Throughout this question, remember that it might help you to draw a picture. It helps us see what you're thinking when we grade you, and you'll be more likely to get partial credit if your answers are wrong.

Consider the following C code:

```
void foo(int a, int b, int c, int d) {
  int buf[16];
  buf[0] = a;
  buf[1] = b;
  buf[2] = c;
  buf[3] = d;
  return;
}

void bar() {
  foo(0x15213, 0x18243, 0xdeadbeef, 0xcafebabe)
}
```

When compiled with default options (32-bit), it gives the following assembly:

```
00000000 <foo>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 40                sub    $0x40,%esp

   6:   8b 45 08                mov    _____(%ebp),%eax //temp = a;
   9:   89 45 c0                mov    %eax,-0x40(%ebp) //buf[0] = temp;

   c:   8b 45 0c                mov    _____(%ebp),%eax //temp = b;
   f:   89 45 c4                mov    %eax,-0x3c(%ebp) //buf[1] = temp;

  12:   8b 45 10                mov    _____(%ebp),%eax //temp = c;
  15:   89 45 c8                mov    %eax,-0x38(%ebp) //buf[2] = temp;

  18:   8b 45 14                mov    _____(%ebp),%eax //temp = d;
  1b:   89 45 cc                mov    %eax,-0x34(%ebp) //buf[3] = temp;
  1e:   c9                      leave
  1f:   c3                      ret

00000020 <bar>:
  20:   55                      push   %ebp
  21:   89 e5                   mov    %esp,%ebp
  23:   83 ec 10                sub    $0x10,%esp
  26:   c7 44 24 0c be ba fe ca movl   $0xcafebabe,0xc(%esp)
  2e:   c7 44 24 08 ef be ad de movl   $0xdeadbeef,0x8(%esp)
  36:   c7 44 24 04 43 82 01 00 movl   $0x18243,0x4(%esp)
  3e:   c7 04 24 13 52 01 00    movl   $0x15213,(%esp)
  45:   e8 fc ff ff ff          call   foo
  4a:   c9                      leave
  4b:   c3                      ret
```

a) Very briefly explain what purpose is served by the first three lines of the disassembly of foo (just repeating the code in words is not sufficient). No more than one sentence should be necessary here.

b) Note that in foo (C version), each of the four arguments are accessed in turn. The assembly dump of foo is commented to show where this is done. Recall that the current `%ebp` value points to where the pushed old base pointer resides, and immediately above that is the return address from the function call. Write into the gaps in the disassembly of foo the offsets from `%ebp` needed to access each of the four arguments `a`, `b`, `c`, and `d`. (Hint: Look at how they are arranged in bar before the call.)

GCC has a compile option called `-fomit-frame-pointer`. When given this flag in addition to the previous flags, the function foo is compiled like this:

```
00000000 <foo>
83 ec 40       sub    $0x40,%esp

8b 44 24 44    mov    ____(%esp),%eax //temp = a;
89 04 24       mov    %eax,(%esp)     //buf[0] = temp;

8b 44 24 48    mov    ____(%esp),%eax //temp = b;
89 44 24 04    mov    %eax,0x4(%esp)  //buf[1] = temp;

8b 44 24 4c    mov    ____(%esp),%eax //temp = c;
89 44 24 08    mov    %eax,0x8(%esp)  //buf[2] = temp;

8b 44 24 50    mov    ____(%esp),%eax //temp = d;
89 44 24 0c    mov    %eax,0xc(%esp)  //buf[3] = temp;
83 c4 40       add    $0x40,%esp
c3             ret
```

c) What is the difference between the first few lines of foo in the first compilation and in this compilation? What does this mean about what the stack frame looks like? (Consider drawing a before/after picture.)

d) Note what has changed in how the arguments `a`, `b`, `c`, `d` and the stack-allocated buffer are accessed: they are now accessed relative to `%esp` instead of `%ebp`. Considering that the arguments are in the same place when foo starts as last time, and recalling what has changed about the stack this time around (note: the pushed return address is still there!), fill in the blanks on the previous page to correctly access the function's arguments.

e) Consider what the compiler has done: `foo` is now using its stack frame without dealing with the base pointer at all... and, in fact, all functions in the program compiled with `-fomit-frame-pointer` also do this. What is a benefit of doing this? (0-point bonus question: What is a drawback?)

Andrew login ID:______________________________

Full Name:______________________________

Recitation Section:______________________________

CS 15-213 / ECE 18-243, Spring 2010
Exam 1

Version 1100101
Tuesday, March 2nd, 2010

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–J) on the front. Read all instructions and sign the statement below.
- Write your answers in the space provided for 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 (instructors reserve the right to change these values). Pile up the easy points quickly and then come back to the harder problems.
- You may not use any books or notes on this exam. Reference material is located at the end of this exam. No calculators or other electronic devices are allowed.
- Good luck!

I understand the CMU policy on cheating applies in full to this exam. ____________________________________

1- Multiple Choice (14):	
2- Peephole (16):	
3- Floating Point (14):	
4- Structs (14):	
5- Stacks (15):	
6- Buffer Overflow (17):	
7- Assembly (10):	
TOTAL (100):	

Problem 1. (14 points):

1. Which of the following lines of C code performs the same operation as the assembly statement `lea 0xffffffff(%esi),%eax`.

   (a) `*(esi-1) = eax`

   (b) `esi = eax + 0xffffffff`

   (c) `eax = esi - 1`

   (d) `eax = *(esi -1)`

2. `test %eax, %eax`
   `jne 3d<function+0x3d>`
   Which of the following values of `%eax` would cause the jump to be taken?

   (a) 1

   (b) 0

   (c) Any value of `%eax`

   (d) No value of `%eax` would cause the jump to be taken.

3. Which of the following are legitimate advantages of x86_64 over IA32? (Circle 0-3)

   (a) x86_64 is able to make use of a larger address space than IA32

   (b) x86_64 is able to make use of more registers than IA32

   (c) x86_64 is able to make use of larger registers than IA32

4. T/F: Any sequence of IA32 instructions can be executed on an x86_64 processor?

   (a) True

   (b) False

5. What sequence of operations does the `leave` instruction execute?

   (a)
   ```
   mov %ebp,%esp
   pop %ebp
   ```

   (b)
   ```
   pop %ebp
   mov %ebp,%esp
   ```

   (c)
   ```
   pop %esp
   mov %ebp,%esp
   ```

   (d)
   ```
   push %ebp
   mov %esp,%ebp
   ```

6. What is the difference between the `%rbx` and the `%ebx` register on an x86_64 machine?

   (a) nothing, they are the same register

   (b) `%ebx` refers to only the low order 32 bits of the `%rbx` register

   (c) they are totally different registers

   (d) `%ebx` refers to only the high order 32 bits of the `%rbx` register

7. On IA32 systems, where is the value of old `%ebp` saved in relation to the current value of `%ebp`?

   (a) there is no relation between where the current base pointer and old base pointer are saved.

   (b) old `%ebp` is stored at (`%ebp - 4`)

   (c) old `%ebp` is stored at (`%ebp + 4`)

   (d) old `%ebp` is stored at (`%ebp`)

Problem 2. (16 points):

Consider the following assembly code:

```
08048334 <mystery>:
 8048334: 55                      push   %ebp
 8048335: 89 e5                   mov    %esp,%ebp
 8048337: 83 ec 0c                sub    $0xc,%esp
 804833a: 8b 45 08                mov    0x8(%ebp),%eax
 804833d: c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
 8048344: 3b 45 fc                cmp    0xfffffffc(%ebp),%eax
 8048347: 75 09                   jne    8048352 <mystery+0x1e>
 8048349: c7 45 f8 00 00 00 00    movl   $0x0,0xfffffff8(%ebp)
 8048350: eb 12                   jmp    8048364 <mystery+0x30>
 8048352: 8b 45 08                mov    0x8(%ebp),%eax
 8048355: 48                      dec    %eax
 8048356: 89 04 24                mov    %eax,(%esp)
 8048359: e8 d6 ff ff ff          call   8048334 <mystery>
 804835e: 03 45 08                add    0x8(%ebp),%eax
 8048361: 89 45 f8                mov    %eax,0xfffffff8(%ebp)
 8048364: 8b 45 f8                mov    0xfffffff8(%ebp),%eax
 8048367: c9                      leave
 8048368: c3                      ret
```

1. Fill in the blanks of the corresponding C function:

```
int  mystery(int  i)
{
  if (______________) return ______________________;

  return _____________________;
}
```

2. Peephole optimizations are a kind of optimization which looks at a small number of assembly instructions and tries to optimize those instructions. Care must be taken to not affect the behavior of the rest of the program. Write an optimized version of the assembly instructions at addresses `0x804833d` and `0x8048344`.

3. If we look at the addresses `0x8048361` and `0x8048364` it seems like we can can eliminate both instructions or replace the instructions with nops. Explain why we can't implement this peephole optimization without affecting the behavior of the rest of the function.

Problem 3. (14 points):

Your friend, Harry Q. Bovik, encounters a function named mystery when runnning gdb on a 32-bit binary that was compiled on the fish machines. Use the gdb output below and the function prototype for mystery to complete this question.

```
int mystery(float arg1, float arg2, float arg3, float arg4);
```

```
Breakpoint 1, 0x08048366 in mystery ()
(gdb) x/20 $esp
0xffd3d1e0:     0xf7f3fff4      0xf7f3e204      0xffd3d208      0x080483cd
0xffd3d1f0:     0x41700000      0x3de00000      0x7f800010      0x00000001
0xffd3d200:     0x7f7fffff      0xffd3d220      0xffd3d278      0xf7e13e9c
0xffd3d210:     0xf7f5fca0      0x080483f0      0xffd3d278      0xf7e13e9c
0xffd3d220:     0x00000001      0xffd3d2a4      0xffd3d2ac      0xf7f60810
(gdb) print $ebp
$1 = (void *) 0xffd3d1e8
```

1. What is on the stack where %ebp is pointing (in hex)?

2. What is the return address of the function mystery?

Fill in the below table. Hexadecimal may be used in the address column. The value column may not contain any binary. Instead of calculating large powers of two you may use exponentials in the value column but your answer must fit within the table boundaries.

	address	value
arg1		
arg2		
arg3		
arg4		

Problem 4. (14 points):

Take the struct below compiled on Linux 32-bit:

```
struct my_struct {
    short b;
    int x;
    short s;
    long z;
    char c[5];
    long long a;
    char q;
}
```

1. Please lay out the struct in memory below (each cell is 1 byte). Please shade in boxes used for padding.

```
+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+
+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+
+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+
+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+
+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+
```

Given the following gdb interaction (where ms is a struct my_struct).

```
(gdb) x/40b  &ms
0xffffcde0:  0xbb  0x00  0x86  0x47  0xf9  0xd9  0x01  0x00
0xffffcde8:  0x6d  0x3b  0xff  0xff  0xbe  0xba  0xef  0xbe
0xffffcdf0:  0x68  0x6c  0x70  0x6d  0x65  0x00  0x00  0x00
0xffffcdf8:  0x1e  0xab  0xdf  0x1e  0xff  0xe1  0xaf  0xde
0xffffce00:  0x21  0x00  0x00  0x00  0xf4  0x7f  0x86  0x47
```

2. Label the fields above and fill in the values below.

   - ms.b = 0x______________
   - ms.x = 0x______________
   - ms.s = 0x______________
   - ms.z = 0x______________
   - ms.c = _______ , _______ , _______ , _______ , _______
   - ms.a = 0x______________
   - ms.q = 0x______________

3. Define a struct with the same elements that has a total size of less than 30 bytes.

```
struct my_compressed_struct {









}
```

4. What is the size of `my_compressed_struct` that you wrote above?

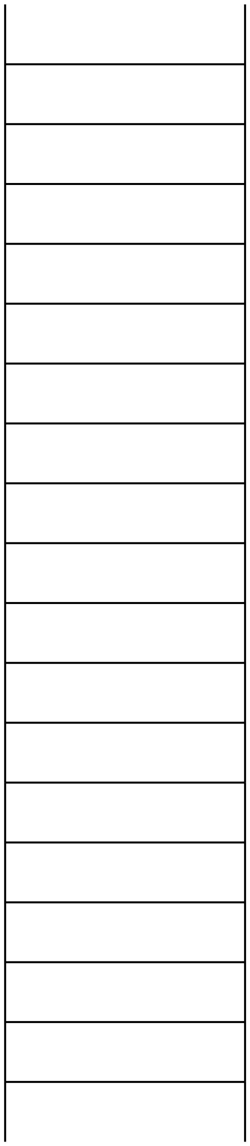
Problem 5. (15 points):

Below is the C code and assembly code for a simple function.

```
000000af <doSomething>:
af:   push   %ebp
b0:   mov    %esp,%ebp
b2:   sub    $0xc,%esp
b5:   mov    0x8(%ebp),%ecx
b8:   mov    $0x1,%eax
bd:   test   %ecx,%ecx
bf:   je     de <doSomething+0x2f>
c1:   mov    %ecx,%edx
c3:   shr    $0x1f,%edx
c6:   lea    (%ecx,%edx,1),%edx
c9:   sar    %edx
cb:   mov    0x10(%ebp),%eax
ce:   mov    %eax,0x8(%esp)
d2:   mov    %ecx,0x4(%esp)
d6:   mov    %edx,(%esp)
d9:   call   da <doSomething+0x2b>
de:   leave
df:   ret
```

```
int doSomething(int a, int b, int c){
    int d;
    if (a == 0){ return 1;}
    d = a/2;
    c = doSomething(d,a,c);
    return c;
}
```

Please draw a detailed stack diagram for this function in Figure 1 on the next page, starting with a function that calls this function and continuing for 2 recursive calls of this function. (That is, at least two stack frames that belong to this function). Please label everything you can.



Problem 6. (17 points):

As a security engineer for a software company it is your job to perform attacks against your company's software and try to break it. One of your developers, Harry Q. Bovik, has written a password validator that he thinks is unbreakable! Below is the front-end to his system:

```
int main(){
    char buffer[20];

    printf("Enter your password >");
    scanf("%s",buffer);
    if(validate(buffer)){
        getOnTheBoat();
        exit(0);
    }
    printf("Sorry, you do not have access :(\n");
    return 0;
}
```

Step 0: Briefly explain how you could attack this program with a buffer overflow. (25 words or less).

Harry then mentions that you actually cannot perform that attack because he runs this on a special system where the stack is not-executable. This means that it is impossible to execute any code on the stack, making the typical attack you performed in buffer-lab now impossible.

You can still do this though! You are going to perform a RETURN TO LIBC attack! This attack relies on pre-existing code in the program that will allow you to execute arbitrary instructions. There are a few important things you need to know about first:

The C function `system(char * command)` will execute the string `command` as if you had typed it into a shell prompt.

Using GDB you discover:

```
(gdb) print system
$1 = {<text variable, no debug info>} 0xf7e263a0 <system>
```

In every program executable, your environment variables are loaded at runtime. And part of your environment variables is your current SHELL:

```
(gdb) print (char *) 0xff89d957
$2 = 0xff89d957 "SHELL=/bin/bash"
```

Using this information, you can successfully launch a shell from Harry's program, proving that you can execute arbitrary code with his program's privelage level!

Step 1:

- What is the address of the system() function?

- What is the address of the string "/bin/bash"?

Step 2: Design your exploit string (keep in mind where arguments go for IA32). We're looking for an drawing of what you can pass as input to this program causing it to launch a shell. Don't worry about exact sizes/lengths.

Step 3: Explain how your exploit string will allow you to execute a shell on Harry's program. This combined with your answer to Step 2 should be enough to prove Harry wrong. (This will be graded independently of your Step 2).

Problem 7. (10 points):

Use the x86_64 assembly to fill in the C function below

```
0x0000000000400498 <mystery+0>:    push   %r13
0x000000000040049a <mystery+2>:    push   %r12
0x000000000040049c <mystery+4>:    push   %rbp
0x000000000040049d <mystery+5>:    push   %rbx
0x000000000040049e <mystery+6>:    sub    $0x8,%rsp
0x00000000004004a2 <mystery+10>:   mov    %rdi,%r13
0x00000000004004a5 <mystery+13>:   mov    %edx,%r12d
0x00000000004004a8 <mystery+16>:   test   %edx,%edx
0x00000000004004aa <mystery+18>:   jle    0x4004c7 <mystery+47>
0x00000000004004ac <mystery+20>:   mov    %rsi,%rbx
0x00000000004004af <mystery+23>:   mov    $0x0,%ebp
0x00000000004004b4 <mystery+28>:   mov    (%rbx),%edi
0x00000000004004b6 <mystery+30>:   callq  *%r13
0x00000000004004b9 <mystery+33>:   mov    %eax,(%rbx)
0x00000000004004bb <mystery+35>:   add    $0x1,%ebp
0x00000000004004be <mystery+38>:   add    $0x4,%rbx
0x00000000004004c2 <mystery+42>:   cmp    %r12d,%ebp
0x00000000004004c5 <mystery+45>:   jne    0x4004b4 <mystery+28>
0x00000000004004c7 <mystery+47>:   add    $0x8,%rsp
0x00000000004004cb <mystery+51>:   pop    %rbx
0x00000000004004cc <mystery+52>:   pop    %rbp
0x00000000004004cd <mystery+53>:   pop    %r12
0x00000000004004cf <mystery+55>:   pop    %r13
0x00000000004004d1 <mystery+57>:   retq
```

```
void mystery(int (*funcP)(int), int a[], int n) {




}
```

Andrew login ID:____

Full Name:____

Section:____

15-213/18-243, Spring 2011

Exam 1

Thursday, March 3, 2011 (v1)

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew login ID, full name, and section on the front.
- This exam is closed book, closed notes. You may not use any electronic devices.
- 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. Good luck!

1 (12):
2 (17):
3 (13):
4 (11):
5 (20):
6 (12):
7 (15):
TOTAL (100):

Problem 1. (12 points):

Multiple choice.

Write the correct answer for each question in the following table:

1	2	3	4	5	6	7	8	9	10
11	12								

1. Consider an `int *a` and an `int n`. If the value of %ecx is `a` and the value of %edx is `n`, which of the following assembly snippets best corresponds to the C statement `return a[n]`?

   (a) `ret (%ecx,%edx,4)`

   (b) `leal (%ecx,%edx,4),%eax`
   `ret`

   (c) `mov (%ecx,%edx,4),%eax`
   `ret`

   (d) `mov (%ecx,%edx,1),%eax`
   `ret`

2. Which of the following 8 bit floating point numbers (1 sign, 3 exponent, 4 fraction) represent NaN?

   (a) 1 000 1111

   (b) 0 111 1111

   (c) 0 100 0000

   (d) 1 111 0000

3. `%rsp` is `0xdeadbeefdeadd0d0`. What is the value in `%rsp` after the following instruction executes?

   `pushq %rbx`

   (a) `0xdeadbeefdeadd0d4`

   (b) `0xdeadbeefdeadd0d8`

   (c) `0xdeadbeefdeadd0cc`

   (d) `0xdeadbeefdeadd0c8`

4. How many lines does a direct-mapped cache have in a set?

   (a) 0

   (b) 1

   (c) 2

   (d) 4

5. On an x86_64 Linux system, which of these takes up the most bytes in memory?

   (a) char a[7]
   (b) short b[3]
   (c) int *c
   (d) float d

6. Two-dimensional arrays are stored in ________ order, to help with cache performance.

   (a) column-major
   (b) row-major
   (c) diagonal-major
   (d) Art-major

7. Which register holds the first argument when an argument is called in IA32 (32 bit) architecture?

   (a) edi
   (b) esi
   (c) eax
   (d) None of the above

8. What is the C equivalent of `mov 0x10(%rax,%rcx,4),%rdx`

   (a) `rdx = rax + rcx + 4 + 10`
   (b) `*(rax + rcx + 4 + 10) = rdx`
   (c) `rdx = *(rax + rcx*4 + 0x10)`
   (d) `rdx = *(rax + rcx + 4 + 0x10)`

9. What is the C equivalent of `leal 0x10(%rax,%rcx,4),%rdx`

   (a) `rdx = 10 + rax + rcx + 4`
   (b) `rdx = 0x10 + rax + rcx*4`
   (c) `rdx = *(0x10 + rax + rcx*4)`
   (d) `*(0x10 + rax + rcx + 4) = rdx`

10. What is the C equivalent of `mov %rax,%rcx`

    (a) `rcx = rax`
    (b) `rax = rcx`
    (c) `rax = *rcx`
    (d) `rcx = *rax`

11. In x86 (IA32) an application's stack grows from

    (a) High memory addresses to low memory addresses
    (b) Low memory addresses to high memory addresses
    (c) Both towards higher and lower addresses depending on the action
    (d) Stacks are a fixed size and do not grow.

12. True or False: In x86_64 the `%rbp` register can be used as a general purpose register.

    - True
    - False

Problem 2. (17 points):

Bits.

A. Convert the following from decimal to 8-bit two's complement.

```
67  =
-35 =
```

B. Please solve the following are datalab-style puzzle. Please write brief and clear comments. You may use large constants. eg. instead of saying (1 << 16), you may use 0x10000.

```
/*
 * reverseBytes - reverse bytes
 *   Example: reverseBytes(0x12345678) = 0x78563412
 *   Legal ops: ! ~ & ^ | + << >>
 */
int reverseBytes(int x)
{






}
```

C. Assume x and y are of type int. For each expression below, give values for x and y which make the expression false, or write "none" if the expression is always true.

- `((x ^ y) < 0)`
- `((~(x | (~x + 1)) >> 31) & 0x1) == !x`
- `(x ^ (x>>31)) - (x>>31) > 0`
- `((x >> 31) + 1) >= 0`
- `(!x | !!y) == 1`

Problem 3. (13 points):

Floats.

Consider a 6-bit floating point data type with 3 exponent bits and 3 fraction bits (there is no sign bit, so the data type can only represent positive numbers). Assume that this data type uses the conventions presented in class, including representations on NaN, infinity, and denormalized values.

A. What is the bias?

B. What is the largest value, other than infinity, that can be represented?

C. What is the smallest value, other than zero, that can be represented?

D. Fill in the following table. Use round-to-even. If a number is too big to represent, use the representation of infinity, and if it is too small to represent, use the representation of 0. Value should be written in decimal.

Bits	Value	Bits	Value
011 000	1		5
	17	111 010	
110 001			3/32
	9 1/2		8 1/2

Problem 4. (11 points):

Structs.

Consider the following struct:

```
typedef struct
{
    char a[3];
    short b[3];
    double c;
    long double d;
    int* e;
    int f;
} JBOB;
```

A. Show how the struct above would appear on a 64-bit (“x86_64”) Linux machine. Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use x’s to indicate bytes that are allocated in the struct but are not used. A long double is 16 bytes long.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. Rearrange the above fields in `foo` to conserve the most space in the memory below. Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks or x's to indicate bytes that are allocated in the struct but are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

C. How many bytes are wasted in part A, inside and after the struct, if the next memory value is a pointer?

D. How many bytes are wasted in part B, inside and after the struct, if the next memory value is a pointer?

Problem 5. (20 points):

Assembly/C translation. Given the x86 assembly dump, please fill in the blank lines for the function in the provided C code:

```
int lolwut(char *s)
{
    int i, n;

    n = 0;

    for(i = 0; ________________; i++)
    {
        if(________________________________)
        {
            return -1;
        }
        n = ________________;
    }
    return ________;
}
```

```
080483a4 <lolwut>:
 80483a4:   55                      push   %ebp
 80483a5:   89 e5                   mov    %esp,%ebp
 80483a7:   53                      push   %ebx
 80483a8:   8b 5d 08                mov    0x8(%ebp),%ebx
 80483ab:   0f b6 0b                movzbl (%ebx),%ecx
 80483ae:   ba 00 00 00 00          mov    $0x0,%edx
 80483b3:   84 c9                   test   %cl,%cl
 80483b5:   74 31                   je     80483e8 <lolwut+0x44>
 80483b7:   8d 41 d0                lea    -48(%ecx),%eax
 80483ba:   ba 00 00 00 00          mov    $0x0,%edx
 80483bf:   3c 09                   cmp    $0x9,%al
 80483c1:   76 0c                   jbe    80483cf <lolwut+0x2b>
 80483c3:   eb 1e                   jmp    80483e3 <lolwut+0x3f>
 80483c5:   83 c3 01                add    $0x1,%ebx
 80483c8:   8d 41 d0                lea    -48(%ecx),%eax
 80483cb:   3c 09                   cmp    $0x9,%al
 80483cd:   77 14                   ja     80483e3 <lolwut+0x3f>
 80483cf:   8d 14 92                lea    (%edx,%edx,4),%edx
 80483d2:   0f be c1                movsbl %cl,%eax
 80483d5:   8d 54 50 d0             lea    -48(%eax,%edx,2),%edx
 80483d9:   0f b6 4b 01             movzbl 0x1(%ebx),%ecx
 80483dd:   84 c9                   test   %cl,%cl
 80483df:   75 e4                   jne    80483c5 <lolwut+0x21>
 80483e1:   eb 05                   jmp    80483e8 <lolwut+0x44>
 80483e3:   ba ff ff ff ff          mov    -1,%edx
 80483e8:   89 d0                   mov    %edx,%eax
 80483ea:   5b                      pop    %ebx
 80483eb:   5d                      pop    %ebp
 80483ec:   c3                      ret
```

A. At address `0x080483a7` we see the instruction `push %ebx`. Name two things that happen as a result of executing that instruction, and explain why the instruction is necessary.

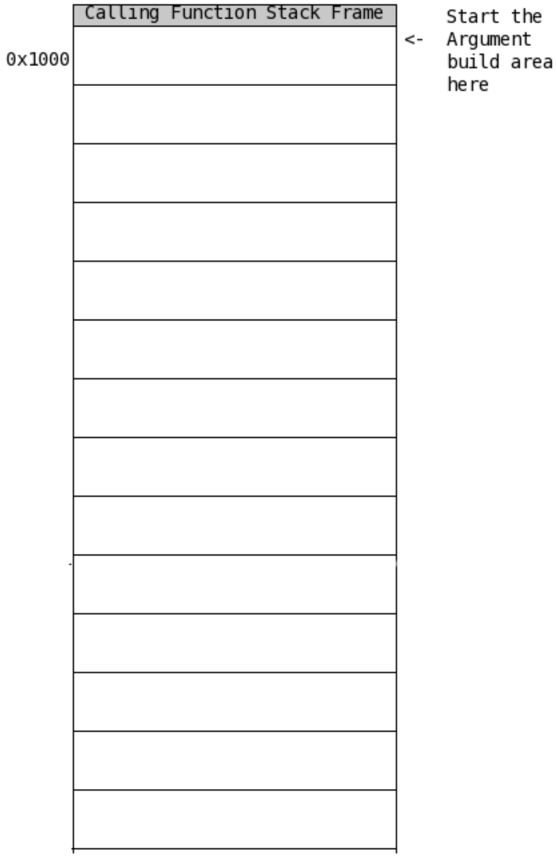
B. Assume that immediately after executing the instruction at address `0x080483a7` (`push %ebx`), the value of `%esp` is `0xffff0000`. If that is the case, at which address would one find the argument `s`?

Problem 6. (12 points):

Stacks.

Given the following function prototypes, and initial lines of IA32 assembly for each function, fill in the stack frame diagram with

- any arguments to the function `foo`
- the return address
- Any registers stored on the stack by the asm fragment (register names not values)
- The location on the stack pointed to by `%esp` and `%ebp` after the exection of the `sub` instruction.


Calling Function Stack Frame
0x1000
<- Start the Argument build area here

```
int foo(int a, int b, int c, int d);
push %ebp
mov  %esp,%ebp
push %ebx
sub $0x10,$esp
```

Problem 7. (15 points):

The Hit or Miss Question

Given a 32-bit Linux system that has a 2-way associative cache of size 128 bytes with 32 bytes per block. Long longs are 8 bytes. For all parts, assume that `table` starts at address 0x0.

```
int i;
int j;
long long table[4][8];
for (j = 0; j < 8; j++) {
  for (i = 0; i < 4; i++) {
    table[i][j] = i + j;
  }
}
```

A. This problem refers to code sample 1. In the table below write down in each space whether that element's access will be a hit or a miss. Indicate hits with a 'H' and misses with a 'M'

	0	1	2	3	4	5	6	7
0								
1								
2								
3								

What is the miss rate of this code sample?

```
int i;
int j;
int table[4][8];
for (j = 0; j < 8; j++) {
  for (i = 0; i < 4; i++) {
    table[i][j] = i + j;
  }
}
```

B. This problem refers to code sample above. In the table below write down in each space whether that element's access will be a hit or a miss. Indicate hits with a 'H' and misses with a 'M'

	0	1	2	3	4	5	6	7
0								
1								
2								
3								

What is the miss rate of this code sample?

C. One code sample performs better than the other. Why is this?

Andrew login ID:_______________________________________

Full Name:_______________________________________

Section:_______________________________________

15-213/18-243, Spring 2011

Exam 1

Thursday, March 3, 2011 (v2)

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew login ID, full name, and section on the front.
- This exam is closed book, closed notes. You may not use any electronic devices.
- 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. Good luck!

1 (12):	
2 (17):	
3 (13):	
4 (11):	
5 (20):	
6 (12):	
7 (15):	
TOTAL (100):	

Problem 1. (12 points):

Multiple choice.

Write the correct answer for each question in the following table:

1	2	3	4	5	6	7	8	9	10
11	12								

1. Consider an `int *a` and an `int n`. If the value of %ecx is `a` and the value of %edx is `n`, which of the following assembly snippets best corresponds to the C statement `return a[n]`?

   (a) `ret (%ecx,%edx,4)`

   (b)
   ```
   leal (%ecx,%edx,4),%eax
   ret
   ```

   (c)
   ```
   mov (%ecx,%edx,4),%eax
   ret
   ```

   (d)
   ```
   mov (%ecx,%edx,1),%eax
   ret
   ```

2. Which of the following 8 bit floating point numbers (1 sign, 3 exponent, 4 fraction) represent NaN?

   (a) 1 000 1111

   (b) 0 111 1111

   (c) 0 100 0000

   (d) 1 111 0000

3. `%rsp` is `0xdeadbeefdeadd0d0`. What is the value in `%rsp` after the following instruction executes?

   ```
   pushq %rbx
   ```

   (a) `0xdeadbeefdeadd0d4`

   (b) `0xdeadbeefdeadd0d8`

   (c) `0xdeadbeefdeadd0cc`

   (d) `0xdeadbeefdeadd0c8`

4. How many lines does a direct-mapped cache have in a set?

   (a) 0

   (b) 1

   (c) 2

   (d) 4

5. On an x86_64 Linux system, which of these takes up the most bytes in memory?

   (a) char a[7]
   (b) short b[3]
   (c) int *c
   (d) float d

6. Two-dimensional arrays are stored in ________ order, to help with cache performance.

   (a) column-major
   (b) row-major
   (c) diagonal-major
   (d) Art-major

7. Which register holds the first argument when an argument is called in IA32 (32 bit) architecture?

   (a) edi
   (b) esi
   (c) eax
   (d) None of the above

8. What is the C equivalent of `mov 0x10(%rax,%rcx,4),%rdx`

   (a) `rdx = rax + rcx + 4 + 10`
   (b) `*(rax + rcx + 4 + 10) = rdx`
   (c) `rdx = *(rax + rcx*4 + 0x10)`
   (d) `rdx = *(rax + rcx + 4 + 0x10)`

9. What is the C equivalent of `leal 0x10(%rax,%rcx,4),%rdx`

   (a) `rdx = 10 + rax + rcx + 4`
   (b) `rdx = 0x10 + rax + rcx*4`
   (c) `rdx = *(0x10 + rax + rcx*4)`
   (d) `*(0x10 + rax + rcx + 4) = rdx`

10. What is the C equivalent of `mov %rax,%rcx`

   (a) `rcx = rax`
   (b) `rax = rcx`
   (c) `rax = *rcx`
   (d) `rcx = *rax`

11. In x86 (IA32) an application's stack grows from

    (a) High memory addresses to low memory addresses
    (b) Low memory addresses to high memory addresses
    (c) Both towards higher and lower addresses depending on the action
    (d) Stacks are a fixed size and do not grow.

12. True or False: In x86_64 the `%rbp` register can be used as a general purpose register.

    - True
    - False

Problem 2. (17 points):

Bits.

A. Convert the following from decimal to 8-bit two's complement.

```
67  =
-35 =
```

B. Please solve the following are datalab-style puzzle. Please write brief and clear comments. You may use large constants. eg. instead of saying (1 << 16), you may use 0x10000.

```
/*
 * reverseBytes - reverse bytes
 *   Example: reverseBytes(0x12345678) = 0x78563412
 *   Legal ops: ! ~ & ^ | + << >>
 */
int reverseBytes(int x)
{









}
```

C. Assume x and y are of type int. For each expression below, give values for x and y which make the expression false, or write "none" if the expression is always true.

- `((x ^ y) < 0)`
- `((~(x | (~x + 1)) >> 31) & 0x1) == !x`
- `(x ^ (x>>31)) - (x>>31) > 0`
- `((x >> 31) + 1) >= 0`
- `(!x | !!y) == 1`

Problem 3. (13 points):

Floats.

Consider a 6-bit floating point data type with 3 exponent bits and 3 fraction bits (there is no sign bit, so the data type can only represent positive numbers). Assume that this data type uses the conventions presented in class, including representations on NaN, infinity, and denormalized values.

A. What is the bias?

B. What is the largest value, other than infinity, that can be represented?

C. What is the smallest value, other than zero, that can be represented?

D. Fill in the following table. Use round-to-even. If a number is too big to represent, use the representation of infinity, and if it is too small to represent, use the representation of 0. Value should be written in decimal.

Bits	Value	Bits	Value
011 000	1		5
	17	111 010	
110 001			3/32
	9 1/2		8 1/2

Problem 4. (11 points):

Structs.

Consider the following struct:

```
typedef struct
{
    char a[3];
    short b[3];
    double c;
    long double d;
    int* e;
    int f;
} JBOB;
```

A. Show how the struct above would appear on a 64-bit ("x86_64") Linux machine. Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use x's to indicate bytes that are allocated in the struct but are not used. A long double is 16 bytes long.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. Rearrange the above fields in `foo` to conserve the most space in the memory below. Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks or x's to indicate bytes that are allocated in the struct but are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

C. How many bytes are wasted in part A, inside and after the struct, if the next memory value is a pointer?

D. How many bytes are wasted in part B, inside and after the struct, if the next memory value is a pointer?

Problem 5. (20 points):

Assembly/C translation.

Given the following x86 assembly dump, please reconstruct the function in the provided C code.

```
int mystery(int (*f)(int, int), int* arr, int c)
{
    int i, x;

    if(_________________)
        return ______________;

    x = _________________;

    for(i = ______________; ______________; __________)
        x = _______________________________;

    return x;
}
```

```
(gdb) disas mystery
Dump of assembler code for function mystery:
0x080483a4 <mystery+0>:push    %ebp
0x080483a5 <mystery+1>:mov     %esp,%ebp
0x080483a7 <mystery+3>:push    %edi
0x080483a8 <mystery+4>:push    %esi
0x080483a9 <mystery+5>:push    %ebx
0x080483aa <mystery+6>:sub     $0xc,%esp
0x080483ad <mystery+9>:mov     0xc(%ebp),%edi
0x080483b0 <mystery+12>:mov    0x10(%ebp),%esi
0x080483b3 <mystery+15>:test   %esi,%esi
0x080483b5 <mystery+17>:jle    0x80483db <mystery+55>
0x080483b7 <mystery+19>:mov    (%edi),%edx
0x080483b9 <mystery+21>:cmp    $0x1,%esi
0x080483bc <mystery+24>:jle    0x80483d9 <mystery+53>
0x080483be <mystery+26>:mov    $0x1,%ebx
0x080483c3 <mystery+31>:mov    (%edi,%ebx,4),%eax
0x080483c6 <mystery+34>:mov    %eax,0x4(%esp)
0x080483ca <mystery+38>:mov    %edx,(%esp)
0x080483cd <mystery+41>:call   *0x8(%ebp)
0x080483d0 <mystery+44>:mov    %eax,%edx
0x080483d2 <mystery+46>:add    $0x1,%ebx
0x080483d5 <mystery+49>:cmp    %ebx,%esi
0x080483d7 <mystery+51>:jg     0x80483c3 <mystery+31>
0x080483d9 <mystery+53>:mov    %edx,%esi
0x080483db <mystery+55>:mov    %esi,%eax
0x080483dd <mystery+57>:add    $0xc,%esp
0x080483e0 <mystery+60>:pop    %ebx
0x080483e1 <mystery+61>:pop    %esi
0x080483e2 <mystery+62>:pop    %edi
0x080483e3 <mystery+63>:pop    %ebp
0x080483e4 <mystery+64>:ret
End of assembler dump.
```

A. At address `0x080483a9` we see the instruction `push %ebx`. Name two things that happen as a result of executing that instruction, and explain why the instruction is necessary.

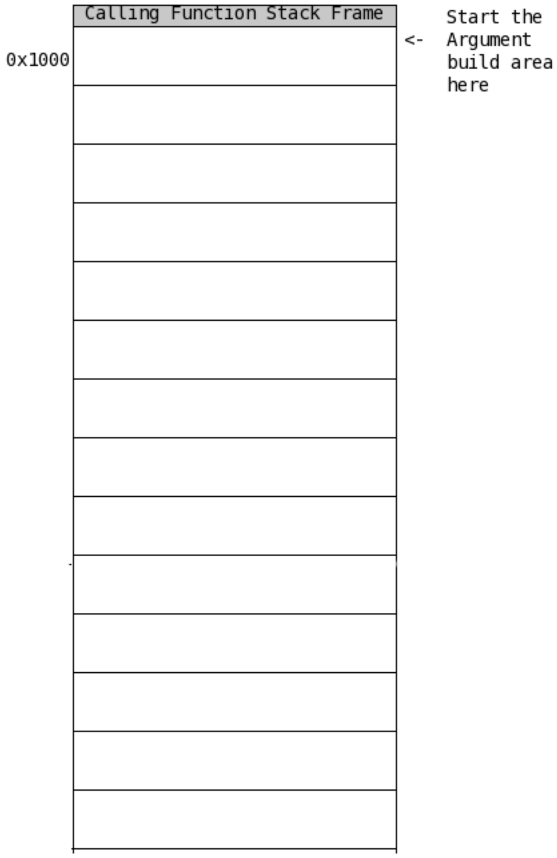
B. Assume that immediately after executing the instruction at address `0x080483a9` (`push %ebx`), the value of `%esp` is `0xffff0000`. If that is the case, at which address would one find the argument `f`?

Problem 6. (12 points):

Stacks.

Given the following function prototypes, and initial lines of IA32 assembly for each function, fill in the stack frame diagram with

- any arguments to the function `foo`
- the return address
- Any registers stored on the stack by the asm fragment (register names not values)
- The location on the stack pointed to by `%esp` and `%ebp` after the exection of the `sub` instruction.


Calling Function Stack Frame
0x1000
<- Start the Argument build area here

```
void foo(char *a, int b);
push %ebp
mov %esp,%ebp
sub $0x10,$esp
```

Problem 7. (15 points):

We will consider performance issues associated with caching the reads from array A. Assume other variables are stored in registers. Also assume A is cache-aligned, and that the cache is cold before running the code.
Consider the following code:

```
#define N 128

int myst(int[] A)
{
   int i, result;

   for (i = 0; i < N; i++)
       result += A[i]*A[n-i-1];

   return result;
}
```

A. Consider a 64-byte, direct mapped cache with 4 sets. Fill in the values that will be stored in this cache when the code reaches the point `return result;`

B. Consider a 64-byte, two-way set associative cache with 4 sets. Fill in the values that will be stored in this cache when the code reaches the point `return result;` Each rectangle in the table represents 4 bytes.

Andrew login ID:______________________________________

Full Name:______________________________________

Section:______________________________________

15-213/18-243, Spring 2011

Exam 2

Thursday, April 21, 2011 v1

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew login ID, full name, and section on the front.
- This exam is closed book and closed notes. A notes sheet is attached to the back.
- 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. Good luck!

1 (12):
2 (15):
3 (20):
4 (18):
5 (10):
6 (15):
7 (10):
TOTAL (100):

Problem 1. (12 points):

Multiple choice.

Write the correct answer for each question in the following table:

1	2	3	4	5	6	7	8	9	10
11	12	13 X	14 X	15 X	16 X	17 X	18 X	19 X	20 X

1. What kind of process can be reaped?

   (a) Exited

   (b) Running

   (c) Stopped

   (d) Both (a) and (c)

2. Which of the following functions will always return exactly once?

   (a) `exec`

   (b) `exit`

   (c) `fork`

   (d) None of the above

3. Given an arbitrary malloc implementation that does not coalesce and a second implementation that is identical except it does coalesce, which of the following is true about their utilization scores on an arbitrary trace? (You may assume the first implementation stores enough information to make coalescing possible, so the only difference is that the second implementation actually performs the coalescing.)

   (a) The coalescing malloc will definitely get a better utilization score.

   (b) The coalescing malloc might get a better utilization score and might get the same utilization score, but it cannot get a worse utilization score.

   (c) The coalescing malloc might get a better utilization score, might get the same utilization score, and might get a worse utilization score.

   (d) The coalescing malloc will definitely get a worse utilization score.

4. Which of the following is a reason next-fit might perform better than first-fit?

   (a) If a large number of small blocks are stored at the beginning of the free list, next-fit avoids walking through those small blocks upon every allocation.
   (b) First-fit requires a traversal of the entire free list, but next-fit does not.
   (c) First-first requires that both allocated and unallocated blocks be examined, and next-fit examines only free blocks.
   (d) Next-fit is an approximation of best-fit, so it reduces internal fragmentation compared to first-fit.

5. How much virtual memory can be addressed by a 32-bit system?

   (a) 2GB
   (b) 4GB
   (c) 8GB
   (d) 240TB

6. Which of the following is a reason why a virtual memory translation would fault?

   (a) Page is not present
   (b) Page is read only
   (c) Page is empty
   (d) All of the above

7. How many bits are needed for the Virtual Page Offset if page size is 5000 bytes?

   (a) 10
   (b) 11
   (c) 12
   (d) 13

8. Which of the following is preserved across exec?

   (a) Signal handlers
   (b) Blocked signals
   (c) a and b
   (d) Neither

9. In what section of an ELF binary are initialized variables located?

   (a) .symtab
   (b) .data
   (c) .bss
   (d) .text

10. What does the call `dup2(oldfd, newfd);` do?

    (a) `newfd` and `oldfd` now both refer to `oldfd`'s entry in the open file table.

    (b) A copy of `oldfd`'s open file table entry is made, and `newfd` points to the copy.

    (c) A copy of the file `oldfd` is pointing to is made on the filesystem. The file is then opened, and `newfd` points to that open file entry.

    (d) The numerical value in oldfd is copied into newfd. No changes are made in the system.

11. Which of the following are shared between a parent and child process immediately following a fork?

    (a) Writeable physical memory

    (b) File descriptor tables

    (c) Instruction pointer

    (d) Open file structs

12. Which signals cannot be handled by the process?

    (a) SIGTSTP

    (b) SIGKILL

    (c) SIGTERM

    (d) All of the Above

Problem 2. (15 points):

Process control.

What are the possible output sequences from the following program:

```
int main() {
    if (fork() == 0) {
        printf("a");
        exit(0);
    }
    else {
        printf("b");
        waitpid(-1, NULL, 0);
    }
    printf("c");
    exit(0);
}
```

A. Circle the possible output sequences: abc acb bac bca cab cba

B. What is the output of the following program?

```
pid_t pid;
int counter = 2;

void handler1(int sig) {
    counter = counter - 1;
    printf("%d", counter);
    fflush(stdout);
    exit(0);
}

int main() {
    signal(SIGUSR1, handler1);
    printf("%d", counter);
    fflush(stdout);

    if ((pid = fork()) == 0) {
        while(1) {};
    }

    kill(pid, SIGUSR1);
    waitpid(-1, NULL, 0);
    counter = counter + 1;
    printf("%d", counter);
    exit(0);
}

OUTPUT: __________
```

Problem 3. (20 points):

Dynamic memory allocation.

In this question, we will consider the utilization score of various malloc implementations on the following code:

```
#define N 64

void *pointers[N];
int i;

for (i = 0; i < N; i++) {
    pointers[i] = malloc(8);
}
for (i = 0; i < N; i++) {
    free(pointers[i]);
}
for (i = 0; i < N; i++) {
    pointers[i] = malloc(24);
}
```

A. Consider a malloc implementation that uses an implicit list with headers of size 8 bytes and no footers. In order to keep payloads aligned to 8 bytes, every block is always constrained to have size a multiple of 8. The header of each block stores the size of the block, and since the 3 lowest order bits are guaranteed to be 0, the lowest order bit is used to store whether the block is allocated or free. A first-fit allocation policy is used. If no unallocated block of a large enough size to service the request is found, `sbrk` is called for the smallest multiple of 8 that can service the request. No coalescing or block splitting is done. NOTE: You do NOT need to simplify any mathematical expressions. Your final answer may include multilpliations, additions, and divisions.

1. After the given code sample is run, how many total bytes have been requested from `sbrk`?

2. How many of those bytes are used for currently allocated blocks, including internal fragmentation and header information?

3. How many of those bytes are used to store free blocks, including header information?

4. Give the fraction of the total number of bytes requested by the user by the end of the trace (not including calls to `malloc` that have subsequently been freed) over total number of bytes allocated by `sbrk`. You do not need to simplify the fraction.

B. Consider another malloc implementation that never calls `sbrk` for a size less than 32 bytes. In every other way the implementation is identical to the implementation in question A. Note that since no block splitting is done, this means the size of each block, including the header, will always be at least 32 bytes. Again, there is no need to simplify mathematical expressions.

   1. After the given code sample is run, how many total bytes have been requested from `sbrk`?

   2. How many of those bytes are used for currently allocated blocks, including internal fragmentation and header information?

   3. How many of those bytes are used to store free blocks, including header information?

   4. Give the fraction of the total number of bytes requested by the user by the end of the trace (not including calls to `malloc` that have subsequently been freed) over total number of bytes allocated by `sbrk`. You do not need to simplify the fraction.

Problem 4. (18 points):

Virtual Memory.

Consider a 32-bit system with a page size of 4KB. A certain kernel designer wishes to analyze the merits of using 2-level page tables.

A. How many entries are there in the page directory?

B. How much virtual memory is reachable from a single page directory entry? (i.e.: 4KB are reachable from a single page table entry).

C. Consider a process with the following sections of memory in its address space:

```
+-----------------------------+ 0xFFFFFFFF
|   9MB     Stack             |
+-----------------------------+
|                             |
|   ...     Unused            |
|                             |
+-----------------------------+
|   6MB     Heap              |
+-----------------------------+
|   4MB     Unused            |
+-----------------------------+
|   12MB    Text and Data     |
+-----------------------------+
|   16MB    Kernel memory     |
+-----------------------------+ 0x00000000
```

Fill in each entry of the page directory below with the name of the corresponding section of memory. The sections are: `unallocated`, `stack`, `heap`, `text and data`, or `kernel memory`.

Page Directory

Index	Entry
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
...	
n−12	
n−11	
n−10	
n−9	
n−8	
n−7	
n−6	
n−5	
n−4	
n−3	
n−2	
n−1	

Problem 5. (10 points):

Assume a System that has

1. A two way set associative TLB
2. A TLB with 8 total entries
3. 2^8 byte page size
4. 2^{16} bytes of virtual memory
5. one (or more) boats

TLB			
Index	Tag	Frame Number	Valid
0	0x13	0x30	1
	0x34	0x58	0
1	0x1F	0x80	0
	0x2A	0x72	1
2	0x1F	0x95	1
	0x20	0xAA	0
3	0x3F	0x20	1
	0x3E	0xFF	0

A. Use the TLB to fill in the table. Strike out anything that you don't have enough information to fill in.

Virtual Address	Physical Address
0x7E85	
0xD301	
	0x3020
0xD040	
	0x5830

Problem 6. (12 points):

Linking.

For each of the following code snippets, write down all symbols in the resulting object files from compilation. Write whether it is a weak global, strong global, or local variable, and what section of the final compiled ELF binary the variable will go into. Fill in the value if you have enough information to determine the value.

A.

main.c

```
#include <stdio.h>

int x;
int y;
int z = 0;

int main() {
  printf("%x\n", x);
  printf("%x\n", y);
  x = 0xdeadbeef;
  printf("%x\n", x);
  printf("%x\n", y);
  return 0;
}
```

foo.c

```
short x = 5;
short y = 2;
```

File	Symbol	Strength and scope	Value	ELF section
main.o				
foo.o				

Problem 7. (10 points):

I/O

Consider the following code. Assume all system calls succeed, and that calls to read() and write() are atomic with respect to each other.

The contents of `foo.txt` are “ABCDEFG”.

A.
```
void read_and_print_one(int fd)
{
    char c;
    read(fd, &c, 1);
    printf("%c", c); fflush(stdout);
}

int main(int argc, char *argv[])
{
    int fd1 = open("foo.txt", O_RDONLY);
    int fd2 = open("foo.txt", O_RDONLY);
    read_and_print_one(fd1);
    read_and_print_one(fd2);

    if(!fork()) {
        read_and_print_one(fd2);
        read_and_print_one(fd2);
        close(fd2);
        fd2 = dup(fd1);
        read_and_print_one(fd2);
    }
    else {
        wait(NULL);
        read_and_print_one(fd1);
        read_and_print_one(fd2);
        printf("\n");
    }

    close(fd1);
    close(fd2);
    return 0;
}
```

Write out the output of this code, and the final contents of the file `foo.txt`.

Andrew login ID:______________________________________

Full Name:______________________________________

Section:______________________________________

15-213/18-243, Spring 2011

Exam 2

Thursday, April 21, 2011 v2

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew login ID, full name, and section on the front.
- This exam is closed book and closed notes. A notes sheet is attached to the back.
- 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. Good luck!

1 (12):
2 (15):
3 (20):
4 (18):
5 (10):
6 (15):
7 (10):
TOTAL (100):

Problem 1. (12 points):

Multiple choice.

Write the correct answer for each question in the following table:

1	2	3	4	5	6	7	8	9	10
11	12	13 X	14 X	15 X	16 X	17 X	18 X	19 X	20 X

1. What kind of process can be reaped?

   (a) Exited

   (b) Running

   (c) Stopped

   (d) Both (a) and (c)

2. Which of the following functions will always return exactly once?

   (a) `exec`

   (b) `exit`

   (c) `fork`

   (d) None of the above

3. Given an arbitrary malloc implementation that does not coalesce and a second implementation that is identical except it does coalesce, which of the following is true about their utilization scores on an arbitrary trace? (You may assume the first implementation stores enough information to make coalescing possible, so the only difference is that the second implementation actually performs the coalescing.)

   (a) The coalescing malloc will definitely get a better utilization score.

   (b) The coalescing malloc might get a better utilization score and might get the same utilization score, but it cannot get a worse utilization score.

   (c) The coalescing malloc might get a better utilization score, might get the same utilization score, and might get a worse utilization score.

   (d) The coalescing malloc will definitely get a worse utilization score.

4. Which of the following is a reason next-fit might perform better than first-fit?

   (a) If a large number of small blocks are stored at the beginning of the free list, next-fit avoids walking through those small blocks upon every allocation.
   (b) First-fit requires a traversal of the entire free list, but next-fit does not.
   (c) First-first requires that both allocated and unallocated blocks be examined, and next-fit examines only free blocks.
   (d) Next-fit is an approximation of best-fit, so it reduces internal fragmentation compared to first-fit.

5. How much virtual memory can be addressed by a 32-bit system?

   (a) 2GB
   (b) 4GB
   (c) 8GB
   (d) 240TB

6. Which of the following is a reason why a virtual memory translation would fault?

   (a) Page is not present
   (b) Page is read only
   (c) Page is empty
   (d) All of the above

7. How many bits are needed for the Virtual Page Offset if page size is 5000 bytes?

   (a) 10
   (b) 11
   (c) 12
   (d) 13

8. Which of the following is preserved across exec?

   (a) Signal handlers
   (b) Blocked signals
   (c) a and b
   (d) Neither

9. In what section of an ELF binary are initialized variables located?

   (a) .symtab
   (b) .data
   (c) .bss
   (d) .text

10. What does the call `dup2(oldfd, newfd);` do?

    (a) `newfd` and `oldfd` now both refer to `oldfd`'s entry in the open file table.

    (b) A copy of `oldfd`'s open file table entry is made, and `newfd` points to the copy.

    (c) A copy of the file `oldfd` is pointing to is made on the filesystem. The file is then opened, and `newfd` points to that open file entry.

    (d) The numerical value in oldfd is copied into newfd. No changes are made in the system.

11. Which of the following are shared between a parent and child process immediately following a fork?

    (a) Writeable physical memory

    (b) File descriptor tables

    (c) Instruction pointer

    (d) Open file structs

12. Which signals cannot be handled by the process?

    (a) SIGTSTP

    (b) SIGKILL

    (c) SIGTERM

    (d) All of the Above

Problem 2. (15 points):

Process control.

Consider the following code sample. You may assume that no call to `fork`, `exec`, `wait`, or `printf` will ever fail, and that stdout is immediately flushed following every call to `printf`.

```
int global_x = 0;

int main(int argc, char *argv[])
{
    global_x = 17;

    /* Assume fork never fails */
    if(!fork()) {
        global_x++;
        printf("Child: %d\n", global_x);
    }
    else {
        wait(NULL);
        global_x--;
        printf("Parent: %d\n", global_x);
    }

    return 0;
}
```

A. What is printed by this program?

B. How might the output change if we removed the call to `wait`? *Note: Keep your answer short, you will be penalized for excessively long answers.*

Now, consider the following two code samples. Again, you may assume that none of the previously mentioned function calls can fail. You should also note that the source for `my_child` follows the source for the invoked program.

```
int global_x = 0;

int main(int argc, char *argv[])
{
    global_x = 17;

    /* Assume fork never fails */
    if(!fork()) {
        global_x++;
        /* Assume exec never fails */
        execl("./my_child", "./my_child", NULL);
        printf("Child finished\n");
    }
    else {
        wait(NULL);
        global_x--;
        printf("Parent: %d\n", global_x);
    }

    return 0;
}
```

Code Listing for my_child:

```
int global_x;

int main(int argc, char *argv[])
{
    printf("Child: %d\n", global_x);
    return 0;
}
```

C. Is the output for this program the same as for the previous? Why or why not? *Note: Again, your answer to this question should be short.*

Problem 3. (20 points):

Dynamic memory allocation.

In this question, we will consider the utilization score of various malloc implementations on the following code:

```
#define N 32

void *pointers[N];
int i;

for (i = 0; i < N; i++) {
    pointers[i] = malloc(4);
}
for (i = 0; i < N; i++) {
    free(pointers[i]);
}
for (i = 0; i < N; i++) {
    pointers[i] = malloc(56);
}
```

A. Consider a malloc implementation that uses an implicit list with headers of size 8 bytes and no footers. In order to keep payloads aligned to 8 bytes, every block is always constrained to have size a multiple of 8. The header of each block stores the size of the block, and since the 3 lowest order bits are guaranteed to be 0, the lowest order bit is used to store whether the block is allocated or free. A first-fit allocation policy is used. If no unallocated block of a large enough size to service the request is found, `sbrk` is called for the smallest multiple of 8 that can service the request. No coalescing or block splitting is done. NOTE: You do NOT need to simplify any mathematical expressions. Your final answer may include multilpliations, additions, and divisions.

1. After the given code sample is run, how many total bytes have been requested from `sbrk`?

2. How many of those bytes are used for currently allocated blocks, including internal fragmentation and header information?

3. How many of those bytes are used to store free blocks, including header information?

4. Give the fraction of the total number of bytes requested by the user by the end of the trace (not including calls to `malloc` that have subsequently been freed) over total number of bytes allocated by `sbrk`. You do not need to simplify the fraction.

B. Consider another malloc implementation that never calls `sbrk` for a size less than 64 bytes. In every other way the implementation is identical to the implementation in question A. Note that since no block splitting is done, this means the size of each block, including the header, will always be at least 64 bytes. Again, there is no need to simplify mathematical expressions.

   1. After the given code sample is run, how many total bytes have been requested from `sbrk`?

   2. How many of those bytes are used for currently allocated blocks, including internal fragmentation and header information?

   3. How many of those bytes are used to store free blocks, including header information?

   4. Give the fraction of the total number of bytes requested by the user by the end of the trace (not including calls to `malloc` that have subsequently been freed) over total number of bytes allocated by `sbrk`. You do not need to simplify the fraction.

Problem 4. (18 points):

Virtual Memory.

Consider a 32-bit system with a page size of 4KB. A certain kernel designer wishes to analyze the merits of using 2-level page tables.

A. Consider the above system utilizing a 2-level page table (the page directory and page tables are 1 page in size).

   (a) How many entries are there in the page directory?

   (b) How many entries are there in the page table?

   (c) If a process were to use all 4GB of available virtual memory, how many page tables would be in use?

   (d) How much memory would the page directory and page tables occupy?

B. Consider the same system but now utilizing a 1-level page table.

   (a) How many entries are there in the page table?

   (b) How much memory would the page table occupy?

C. Why would the kernel designer opt for a 2-level page table when a full 2-level page table takes up more memory than a full 1-level page table?

Problem 5. (10 points):

Assume a System that has

1. A two way set associative TLB
2. A TLB with 8 total entries
3. 2^8 byte page size
4. 2^{16} bytes of virtual memory
5. one (or more) boats

TLB			
Index	Tag	Frame Number	Valid
0	0x13	0x30	1
	0x34	0x58	0
1	0x1F	0x80	0
	0x2A	0x72	1
2	0x1F	0x95	1
	0x20	0xAA	0
3	0x3F	0x20	1
	0x3E	0xFF	0

A. Use the TLB to fill in the table. Strike out anything that you don't have enough information to fill in.

Virtual Address	Physical Address
0x7E85	
0xD301	
	0x3020
0xD040	
	0x5830

Problem 6. (12 points):

Linking.

For each of the following code snippets, write down all symbols in the resulting object files from compilation. Write whether it is a weak global, strong global, or local variable, and what section of the final compiled ELF binary the variable will go into. Fill in the value if you have enough information to determine the value.

A.

main.c

```
int x = 5;
int y;
static int z = 3;

int main() {
  printf("%x\n", z());
  x = 0xdeadbeef;
  printf("%x\n", z());
}
```

foo.c

```
short x;
int y = 0x12345678;

int z() {
  return y;
}
```

File	Symbol	Strength and scope	Value	ELF section
main.o				
foo.o				

Problem 7. (10 points):

I/O

Consider the following code. Assume all system calls succeed, and that calls to read() and write() are atomic with respect to each other.

The contents of `foo.txt` are “ABCDEFG”.

A.
```
void read_and_print_one(int fd)
{
    char c;
    read(fd, &c, 1);
    printf("%c", c); fflush(stdout);
}

int main(int argc, char *argv[])
{
    int fd1 = open("foo.txt", O_RDONLY);
    int fd2 = open("foo.txt", O_RDONLY);
    read_and_print_one(fd1);
    read_and_print_one(fd2);

    if(!fork()) {
        read_and_print_one(fd2);
        read_and_print_one(fd2);
        close(fd2);
        fd2 = dup(fd1);
        read_and_print_one(fd2);
    }
    else {
        wait(NULL);
        read_and_print_one(fd1);
        read_and_print_one(fd2);
        printf("\n");
    }

    close(fd1);
    close(fd2);
    return 0;
}
```

Write out the output of this code, and the final contents of the file `foo.txt`.

Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Fall 2002

Exam 2

November 12, 2002

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 66 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. You may use any books or notes you like. You may use a calculator, but no laptops or other wireless devices. Good luck!

1 (09):	
2 (08):	
3 (08):	
4 (12):	
5 (10):	
6 (10):	
7 (09):	
TOTAL (66):	

Problem 1. (9 points):

This problem tests your understanding of code optimization. Consider the following function for computing the product of an array of n integers. We have unrolled the loop by a factor of 4.

```
int aprod (int a[], int n)
{
    int i, w, x, y, z, r=1;

    for (i = 0; i < n-3; i += 4) {
        w = a[i]; x = a[i+1]; y = a[i+2]; z = a[i+3];
        r = r * w * x * y * z;  // Product computation
    }
    for (; i < n; i++)
        r *= a[i];
    return r;
}
```

For the line labeled `Product computation`, we can use parentheses to create 3 different associations of the computation, as follows:

```
        r = (((r * w) * x) * y) * z;   // A1
        r = (r * w) * ((x * y) * z);   // A2
        r = r * (w * (x * (y * z)));   // A3
```

Complete the following table with the theoretical CPE (cycles per element) of each of these associations. Assume that this machine has an infinite number of integer multipliers, all capable of operating in parallel with each other. Also, assume that integer multiplication on this machine has a latency of 4 cycles and an issue time of 1 cycle.

Version	Theoretical CPE
A1	
A2	
A3	

Here are some hints:

- Recall that the CPE measure assumes that the run time, measured in clock cycles, for an array of length n is a function of the form $Cn + K$, where C is the CPE.
- "Theoretical CPE" means the performance that would be achieved if the only limiting factors were the data dependences of computation and the latency and issue time of the integer multiplier.

Problem 2. (8 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 14 bits wide.
- The cache is 4-way set associative, with a 4-byte block size and 64 total lines.

In the following tables, **all numbers are given in hexadecimal**. The *Index* column contains the set index for each set of 4 lines. The *Tag* columns contain the tag value for each line. The *V* column contains the valid bit for each line. The *Bytes 0–3* columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

The contents of the cache are as follows:

4-way Set Associative Cache												
Index	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3
0	0C	0	03 3E CD 38	A0	0	16 7B ED 5A	40	0	8E 4C DF 18	58	0	FB B7 12 02
1	3A	1	A9 76 2B EE	54	0	BC 91 D5 92	98	1	80 BA 9B F6	84	1	48 16 81 0A
2	26	0	75 F7 3F C6	78	1	9E 3A 0F DA	26	1	00 4C B6 A8	5E	1	92 04 E5 2E
3	B8	1	E0 22 19 3A	D2	0	02 B3 8F B6	D4	1	25 31 E1 02	C2	0	18 09 73 02
4	54	1	86 B8 F0 C6	4C	1	AA 29 AE 16	56	1	76 46 80 6E	1C	1	13 EA A8 66
5	F6	0	04 2A 32 6A	9E	0	B1 86 56 0E	CC	0	96 30 47 F2	06	1	F8 1D 42 30
6	BE	0	2F 7E 3D A8	C0	0	27 95 A4 74	C4	1	07 11 6B D8	8A	1	C7 B7 AF C2
7	A0	0	D6 A4 89 92	10	0	FD FE D6 DA	76	0	DE D5 CD 4A	E2	0	7C 68 3A 1A
8	F0	1	ED 32 0A A2	E4	1	BF 80 1D FC	14	1	EF 09 86 2A	BC	1	25 44 6F 1A
9	30	1	1E C2 AE 60	08	0	5C 3E DF F2	CA	0	25 CF 84 DA	5C	1	F1 6B DC DE
A	38	1	5D 4D F7 DA	82	1	69 C2 8C 74	9C	1	A8 CE 7F DA	3E	1	FA 93 EB 48
B	3A	1	61 C6 5E 74	64	0	03 97 BA 62	80	1	F8 11 72 12	E0	1	C5 EC 76 4E
C	D4	0	17 52 75 2C	AE	0	62 89 EF 18	8E	0	BB 7D 8C 7C	68	0	26 57 7F C2
D	DC	1	54 9E 1E FA	B6	1	DC 81 B2 14	00	0	B6 1F 7B 44	74	0	10 F5 B8 2E
E	D6	0	14 9A 0D 4A	EA	1	C8 1D E6 6E	38	1	F3 38 F3 5C	64	0	6C 8F BD A8
F	7E	1	32 21 1C 2C	FA	1	22 C2 DC 34	BE	1	BA DD 37 D8	B8	0	E7 A2 39 BA

Part 1

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

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

Part 2

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: 2BB2

A. Physical address format (one bit per box)

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

B. Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Physical address: 098B

A. Physical address format (one bit per box)

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

B. Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Problem 3. (8 points):

This problem tests your understanding of cache conflict misses. Consider the following matrix transpose routine

```
typedef int array[2][2];

void transpose(array dst, array src) {
  int i, j;

  for (j = 0; j < 2; j++) {
    for (i = 0; i < 2; i++) {
      dst[i][j] = src[j][i];
    }
  }
}
```

running on a hypothetical machine with the following properties:

- `sizeof(int) == 4`.
- The `src` array starts at address 0 and the `dst` array starts at address 16 (decimal).
- There is a single L1 cache that is direct mapped and write-allocate, with a block size of 8 bytes.
- Accesses to the `src` and `dst` arrays are the only sources of read and write misses, respectively.

A. Suppose the cache has a total size of 16 data bytes (i.e., the block size times the number of sets is 16 bytes) and that the cache is initially empty. Then for each `row` and `col`, indicate whether each access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.

`src` array		
	col 0	col 1
row 0	m	
row 1		

`dst` array		
	col 0	col 1
row 0	m	
row 1		

B. Repeat part A for a cache with a total size of 32 data bytes.

`src` array		
	col 0	col 1
row 0	m	
row 1		

`dst` array		
	col 0	col 1
row 0	m	
row 1		

The following problem concerns the cache performance of three functions that compute the sum of all elements in an $N \times N$ array for different values of N.

```
typedef int array_t[N][N];

int sumA(array_t a)
{
    int i, j;
    int sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++) {
            sum += a[i][j];
        }
    return sum;
}

int sumB(array_t a)
{
    int i, j;
    int sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < N; i++) {
            sum += a[i][j];
        }
    return sum;
}

int sumC(array_t a)
{
    int i, j;
    int sum = 0;
    for (j = 0; j < N; j+=2)
        for (i = 0; i < N; i+=2) {
            sum += (a[i][j] + a[i+1][j]
                    + a[i][j+1] + a[i+1][j+1]);
        }
    return sum;
}
```

Problem 4. (12 points):

This problem tests your ability to analyze the cache behavior of C code. Assume we execute the three summation functions shown on the previous page under the following conditions:

- `sizeof(int) == 4`.
- The machine has a 4KB direct-mapped cache with a 16-byte block size.
- Within the two loops, the code uses memory accesses only for the array data. The loop indices, and the value `sum` are held in registers.
- Array `a` is stored starting at memory address `0x08000000`.

Fill in the table for the approximate cache miss rate for the two cases: $N = 64$ and $N = 60$.

Function	`N = 64`	`N = 60`
`sumA`		
`sumB`		
`sumC`		

Problem 5. (10 points):

This problem concerns the way virtual addresses are translated into physical addresses. Imagine a system has the following parameters:

- Virtual addresses are 20 bits wide.
- Physical addresses are 18 bits wide.
- The page size is 1024 bytes.
- The TLB is 2-way set associative with 16 total entries.

The contents of the TLB and the first 32 entries of the page table are shown as follows. **All numbers are given in hexadecimal**.

TLB			
Index	Tag	PPN	Valid
0	03	C3	1
	01	71	0
1	00	28	1
	01	35	1
2	02	68	1
	3A	F1	0
3	03	12	1
	02	30	1
4	7F	05	0
	01	A1	0
5	00	53	1
	03	4E	1
6	1B	34	0
	00	1F	1
7	03	38	1
	32	09	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
000	71	1	010	60	0
001	28	1	011	57	0
002	93	1	012	68	1
003	AB	0	013	30	1
004	D6	0	014	0D	0
005	53	1	015	2B	0
006	1F	1	016	9F	0
007	80	1	017	62	0
008	02	0	018	C3	1
009	35	1	019	04	0
00A	41	0	01A	F1	1
00B	86	1	01B	12	1
00C	A1	1	01C	30	0
00D	D5	1	01D	4E	1
00E	8E	0	01E	57	1
00F	D4	0	01F	38	1

Part 1

1. The diagram below shows the format of a virtual address. Please indicate the following fields by labeling the diagram: (If a field does not exist, do not draw it on the diagram.)

VPO	The virtual page offset
VPN	The virtual page number
TLBI	The TLB index
TLBT	The TLB tag

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. The diagram below shows the format of a physical address. Please indicate the following fields by labeling the diagram: (If a field does not exist, do not draw it on the diagram.)

PPO	The physical page offset
PPN	The physical page number

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Part 2

For the given virtual addresses, please indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs. If there is a page fault, enter “-” for “PPN” and leave the physical address blank.

Virtual address: `078E6`

1. Virtual address (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. Address translation

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	PPN	0x

3. Physical address(one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Virtual address: `04AA4`

1. Virtual address (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. Address translation

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	PPN	0x

3. Physical address(one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Problem 6. (10 points):

This problem tests your understanding of Unix process control.

Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main()
{
    int status;
    int counter = 1;

    if (fork() == 0) {
        counter++;
        printf("%d",counter);
    }
    else {
        if (fork() == 0) {
            printf("5");
            counter--;
            printf("%d",counter);
            exit(0);
        }
        else {
            if (wait(&status) > 0) {
                printf("6");
            }
        }
    }

    printf("3");
    exit(0);
}
```

For each of the following strings, circle whether (Y) or not (N) this string is a possible output of the program.

A. `253063`   Y   N

B. `251633`   Y   N

C. `520633`   Y   N

D. `263503`   Y   N

E. `506323`   Y   N

Problem 7. (9 points):

This question tests your understanding of signals and signal handlers.

It presents 3 different snippets of C code. Assume that all functions and procedures return correctly and that all variables are declared and initialized properly. Also, assume that an arbitrary number of SIGINT signals, and only SIGINT signals, can be sent to the code snippets randomly from some external source.

For each code snippet, circle the value(s) of `i` that could possibly be printed by the `printf` command at the end of each program. *Careful: There may be more than one correct answer for each question. Circle all the answers that could be correct.*

Code Snippet 1:

```
int i = 0;

void handler(int sig) {
  i = 0;
}

int main() {
  int j;

  signal(SIGINT, handler);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  printf("i = %d\n", i);
  exit(0);
}
```

1. Circle possible values of `i` printed by snippet 1:

A. 0

B. 1

C. 50

D. 100

E. 101

F. None of the above

Code Snippet 2:

```
int i = 0;

void handler(int sig) {
  i = 0;
}

int main () {
  int j;
  sigset_t s;

  signal(SIGINT, handler);

  /* Assume that s has been
     initialized and declared
     properly for SIGINT */

  sigprocmask(SIG_BLOCK, &s, 0);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  sigprocmask(SIG_UNBLOCK, &s, 0);
  printf("i = %d\n", i);
  exit(0);
}
```

2. Circle possible values of `i` printed by snippet 2:

A. 0

B. 1

C. 50

D. 100

E. 101

F. None of the above

Code Snippet 3:

```
int i = 0;

void handler(int sig) {
  i = 0;
  sleep(1);
}


int main () {
  int j;
  sigset_t s;

  /* Assume that s has been
     initialized and declared
     properly for SIGINT */

  sigprocmask(SIG_BLOCK, &s, 0);
  signal(SIGINT, handler);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  printf("i = %d\n", i);
  sigprocmask(SIG_UNBLOCK, &s, 0);
  exit(0);
}
```

3. Circle possible values of `i` printed by snippet 3:

A. 0

B. 1

C. 50

D. 100

E. 101

F. None of the above

Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Fall 2003

Exam 2

November 18, 2003

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 66 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. You may use any books or notes you like. You may use a calculator, but no laptops or other wireless devices. Good luck!

1 (10):
2 (10):
3 (12):
4 (11):
5 (09):
6 (08):
7 (06):
TOTAL (66):

Problem 1. (10 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 13 bits wide.
- The cache is 4-way set associative, with a 4-byte block size and 32 total lines.

In the following tables, **all numbers are given in hexadecimal**. The *Index* column contains the set index for each set of 4 lines. The *Tag* columns contain the tag value for each line. The *V* column contains the valid bit for each line. The *Bytes 0–3* columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

The contents of the cache are as follows:

4-way Set Associative Cache												
Index	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3
0	F0	1	ED 32 0A A2	8A	1	BF 80 1D FC	14	1	EF 09 86 2A	BC	0	25 44 6F 1A
1	0C	0	03 3E CD 38	A0	0	16 7B ED 5A	8A	1	8E 4C DF 18	E4	1	FB B7 12 02
2	8A	1	54 9E 1E FA	B6	1	DC 81 B2 14	00	1	B6 1F 7B 44	74	0	10 F5 B8 2E
3	BE	0	2F 7E 3D A8	C0	1	27 95 A4 74	C4	0	07 11 6B D8	8A	1	C7 B7 AF C2
4	7E	1	32 21 1C 2C	8A	1	22 C2 DC 34	BE	1	BA DD 37 D8	DC	0	E7 A2 39 BA
5	98	0	A9 76 2B EE	54	0	BC 91 D5 92	98	1	80 BA 9B F6	8A	1	48 16 81 0A
6	38	1	5D 4D F7 DA	82	1	69 C2 8C 74	8A	1	A8 CE 7F DA	3E	1	FA 93 EB 48
7	8A	1	04 2A 32 6A	9E	0	B1 86 56 0E	CC	1	96 30 47 F2	06	1	F8 1D 42 30

Part 1

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

12	11	10	9	8	7	6	5	4	3	2	1	0

Part 2

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: `0x1314`

Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0

Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Physical address: `0x08DF`

Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0

Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Part 3

For the given contents of the cache, list all of the hex physical memory addresses that will hit in Set 3. To save space, you should express contiguous addresses as a range. For example, you would write the four addresses `0x1314`, `0x1315`, `0x1316`, `0x1317` as `0x1314--0x1317`.

Answer: ____

The following templates are provided as scratch space:

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

Part 4

For the given contents of the cache, what is the probability (expressed as a percentage) of a cache hit when the physical memory address ranges between `0x1140` - `0x115F`. Assume that all addresses are equally likely to be referenced.

Probability = ___________%

The following templates are provided as scratch space:

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

Problem 2. (10 points):

The following problem concerns the way virtual addresses are translated into physical addresses.

- The memory is byte addressable.
- Memory accesses are to 4-byte words.
- Virtual addresses are 22 bits wide.
- Physical addresses are 18 bits wide.
- The page size is 2048 bytes.
- The TLB is 2-way set associative with 16 total entries.

In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB and the page table for the first 32 pages are as follows:

TLB			
Index	Tag	PPN	Valid
0	003	EB	1
	007	46	0
1	028	D3	1
	001	2F	0
2	031	E0	1
	012	D3	0
3	001	5C	0
	00B	D1	1
4	02A	BA	0
	011	F1	0
5	01F	18	1
	002	4A	1
6	007	63	1
	03F	AF	0
7	010	0D	0
	032	10	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	37	1	10	16	0
01	58	1	11	37	0
02	19	1	12	28	0
03	2A	1	13	53	0
04	56	0	14	1D	0
05	33	0	15	4A	1
06	61	0	16	49	0
07	28	0	17	26	0
08	42	0	18	0C	1
09	63	0	19	04	1
0A	31	1	1A	1F	0
0B	5C	0	1B	22	1
0C	5A	1	1C	40	0
0D	2D	0	1D	0E	1
0E	4E	0	1E	35	1
0F	1D	1	1F	03	1

A. Part 1

(a) The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

VPO	The virtual page offset
VPN	The virtual page number
TLBI	The TLB index
TLBT	The TLB tag

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(b) 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:

PPO	The physical page offset
PPN	The physical page number

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

B. Part 2

For the given virtual addresses, indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs.

If there is a page fault, enter "-" for "PPN" and leave part C blank.

Virtual address: `0x005EEC`

(a) Virtual address format (one bit per box)

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(b) Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

(c) Physical address format (one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Virtual address: `0x00AF9D`

(a) Virtual address format (one bit per box)

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(b) Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

(c) Physical address format (one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Problem 3. (12 points):

This problem tests your understanding of basic cache operations. Harry Q. Bovik has written the mother of all game-of-life programs. The Game-of-life is a computer game that was originally described by John H. Conway in the April 1970 issue of Scientific American. The game is played on a 2 dimensional array of cells that can either be alive (= has value 1) or dead (= has value 0). Each cell is surrounded by 8 neighbors. If a life cell is surrounded by 2 or 3 life cells, it survives the next generation, otherwise it dies. If a dead cell is surrounded by exactly 3 neighbors, it will be born in the next generation.

Harry uses a very, very large $N \times N$ array of `int`'s, where N is an integral power of 2. It is so large that you don't need to worry about any boundary conditions. The inner loop uses two int-pointers src and dst that scan the cell array. There are two arrays: src is scanning the current generation while dst is writing the next generation. Thus Harry's inner loop looks like this:

```
...
   int *src, *dst;
...
  {     int n;

    /* Count life neigbors */
    n  = src[ 1    ];
    n += src[ 1 - N];
    n += src[   - N];
    n += src[-1 - N];
    n += src[-1    ];
    n += src[-1 + N];
    n += src[     N];
    n += src[ 1 + N];

    /* update the next generation */
    *dst = (((*src != 0) && (n == 2)) || (n == 3)) ? 1 : 0;

    dst++;
    src++;
  }
...
```

You should assume that the pointers src and dst are kept in registers and that the counter variable n is also in a register. Furthermore, Harry's machine is fairly old and uses a write-through cache with no-write-allocate policy. Therefore, you do *not* need to worry about the write operation for the next generation.

Each cache line on Harry's machine holds 4 int's (16 Bytes). The cache size is 16 KBytes, which is too small to hold even one row of Harry's game of life arrays. Hint: each row has N elements, where N is a power of 2.

Figure 1 shows how Harry’s program is scanning the game of life array. The thick vertical bars represent the boundaries of cache lines: four consecutive horizontal squares are one cache line. A neighborhood consists of the 9 squares (cells) that are not marked with an X. The single gray square is the `int` cell that is currently pointed to by src.

The 2 neighborhoods shown in Figure 1 represent 2 successive iterations (case A and B) through the inner loop. The src pointer is incremented one cell at a time and moves from left to right in these pictures.

You shall mark each of the 9 squares those with either a ’H’ or a ’M’ indicating if the corresponding memory read operation hits (H) or misses (M) in the cache. Cells that contain an X do not belong to the neighborhood that is being evaluated and you should not mark these.

Part 1

In this part, assume that the cache is organized as a direct mapped cache. Please mark the left column in Figure 1 with your answer. The right column may be used as scratch while you reason about your answer. We will grade the left column only.

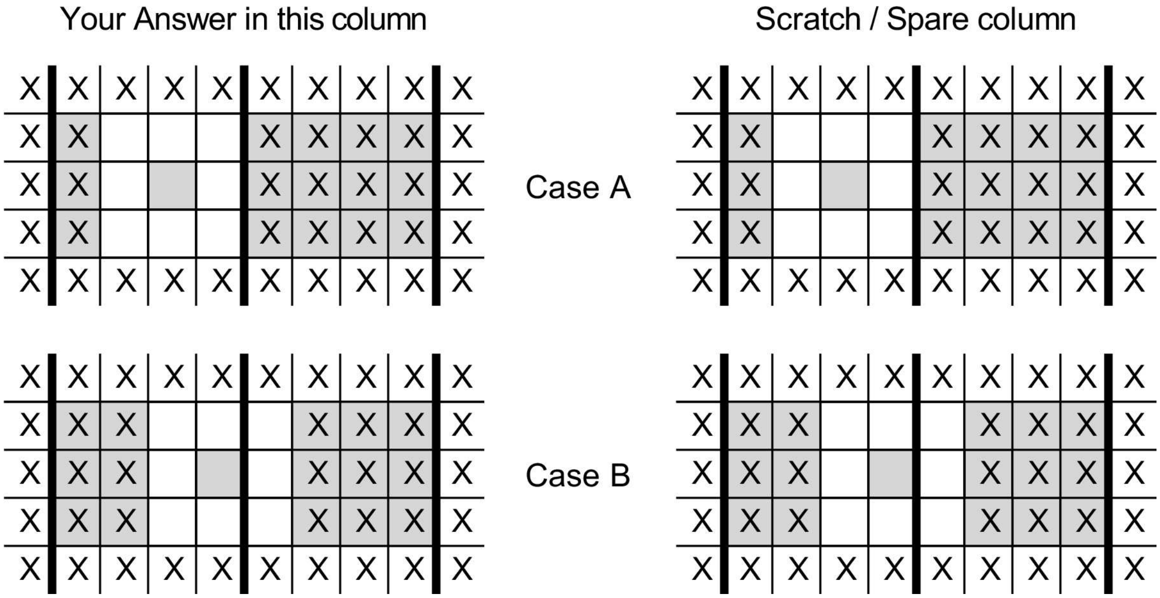


Figure 1: Game of Life with a direct mapped cache

Part 2

In this part, assume a 3-way, set-associative cache with true Least Recently Used replacement policy (LRU). As in Part 1 of this question, please provide your answer by marking the empty squares of the left column in Figure 2 with your solution.

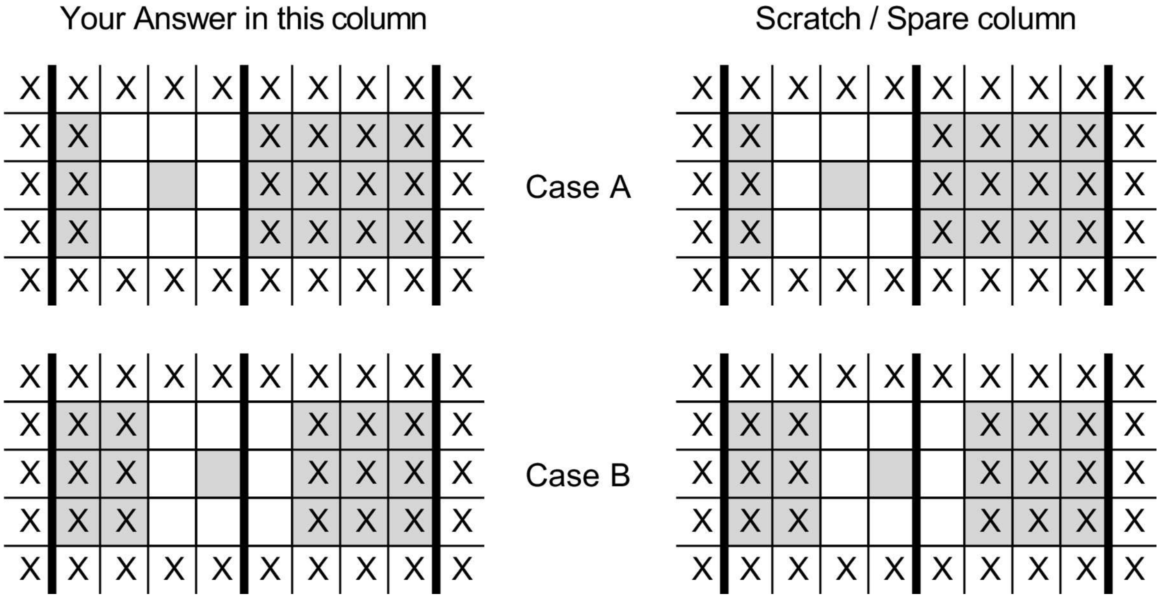


Figure 2: Game of Life with a set associative cache

Problem 4. (11 points):

This problem requires you to analyze the behavior of the inner loops from a simple linear algebra package. The int-vector X of length N is added to all rows of the $N \times N$ int-matrix A:

```
...
   int X[N] = {0};
   int A[N][N] = {0};
...
  {   int i, j;
...
      for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
          int t;
          t = A[i][j];
          t += X[j];
          A[i][j] = t;
        }
     }
...
  }
```

X is allocated first and is directly followed by the matrix A. In other words, the address of $X[N]$ is the address of $A[0][0]$. You may assume that X is aligned so that $X[0]$ maps to the first set of the cache.

Part 1

Assume a 1K byte direct-mapped cache with 16 byte blocks. Each `int` is 4 bytes long. You should assume that i, j, and t are kept in registers and are not polluting the cache. Furthermore, you should ignore all instruction fetches: you are only concerned with the data cache.

Fill in the table below for the following problems size, estimating the miss-rate expressed as a percentage of all load/store operations to X and A. For the percentage, 2 digits of precision suffices.

N	Total # of memory refs to A and X	# of misses to X	# of misses to A	Miss rate (in %)
8				
64				
60				

Part 2

Consider lines 10-12 of the program. How can you rewrite this part of the loop to improve performance?

Part 3

A Victim-Cache (VC) is a small extra cache that is used for lines that are evicted from the main cache. VC's tend to be small and fully associative. On a read, the processor is looking up the victim cache and the main cache in parallel. If there is a hit in the victim cache, the line is transferred back to the main cache and the line from the main cache is now stored in the VC (line swap). On a cache miss (both VC and main cache), the data is fetched from memory and placed in the main cache. The evicted line is then copied to the VC. Assume that this machine has a 1-line victim cache. What is the miss rate of this system for $N = 64$?

Miss rate = ____________________%

Problem 5. (9 points):

This problem tests your understanding of memory bugs. Each of the code sequences below may or may not contain memory bugs. The code all compiles without warnings or errors. If you think there is a bug, please circle **YES** and indicate the type of bug from the list below of memory bugs. Otherwise, if you think there are no memory bugs in the code, please circle **NO**.

Bugs:

1. Potential buffer overflow error
2. Memory leak
3. Potential for dereferencing a bad pointer
4. Incorrect use of free
5. Incorrect use of realloc
6. Misaligned access to memory
7. Other memory bug

Part A

```
/*
 * strndup - An attempt to write a safe version of strdup
 *
 * Note: For this problem, assume that if the function returns a
 * non-NULL pointer to dest, then the caller eventually frees the dest buffer.
 */
char *strndup(char *src, int max)
{
    char *dest;
    int i;

    if (!src || max <= 0)
        return NULL;
    dest = malloc(max+1);
    for (i=0; i < max && src[i] != 0; i++)
        dest[i] = src[i];
    dest[i] = 0;
    return dest;
}
```

NO **YES** Type of bug: ____________________

Part B

```
/* Note: For this problem, asssume that if the function returns a non-NULL
 * pointer to node, then the caller eventually frees node. */
struct Node {
    int data;
    struct Node *next;
};

struct List {
    struct Node *head;
};

struct Node *push(struct List *list, int data)
{
    struct Node *node = (struct Node *)malloc(sizeof(struct Node));

    if (!(list && node))
        return NULL;
    node->data = data;
    node->next = list->head;
    list->head = node;
    return node;
}
```

NO **YES** Type of bug: ____________________

Part C

```
/* print_shortest - prints the shortest of two strings */
void print_shortest(char *str1, char *str2)
{
    printf("The shortest string is %s\n", shortest(str1, str2));
}

char *shortest(char *str1, char *str2)
{
    char *equal = "equal";
    int len1 = strlen(str1);
    int len2 = strlen(str2);

    if (len1 == len2)
        return equal;
    else
        return (len1 < len2 ? str1 : str2);
}
```

NO **YES** Type of bug: ____________________

Problem 6. (8 points):

This problem tests your understanding of Unix process control. Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main()
{
  int status;
  int counter = 2;
  pid_t pid;

  if ((pid = fork()) == 0) {
      counter += !fork();
      printf("%d", counter);
      fflush(stdout);
      counter++;
  }
  else {
      if (waitpid(pid, &status, 0) > 0) {
          printf("6");
          fflush(stdout);
      }
      counter += 2;
  }

  printf("%d", counter);
  fflush(stdout);
  exit(0);
}
```

For each of the following strings, circle whether (Y) or not (N) this string is a possible output of the program.

```
A. 264343      Y    N
B. 236434      Y    N
C. 243643      Y    N
```

How many possible strings start with 234... ? (Give only the number of strings.)

Answer = _______

Problem 7. (6 points):

Suppose the file foo.txt contains the text ”yweixtr”, bar.txt contains the text ”ounazvs”, and baz.txt does not yet exist. Examine the following C code, and answer the two questions below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main() {
    int fd1, fd2, fd3, fd4, status;
    pid_t pid;
    char c;

    fd1 = open("foo.txt", O_RDONLY, 0);
    fd2 = open("foo.txt", O_RDONLY, 0);
    fd3 = open("bar.txt", O_RDONLY, 0);
    fd4 = open("baz.txt", O_WRONLY | O_CREAT, DEF_MODE); /* r/w */

    dup2(fd4, STDOUT_FILENO);

    if ((pid = fork()) == 0) {
        dup2(fd3, fd2);
        dup2(STDOUT_FILENO, fd4);
        read(fd1, &c, 1);
        printf("%c", c);
        read(fd2, &c, 1);
        printf("%c", c);
        read(fd3, &c, 1);
        printf("%c", c);
        printf("\n");
        exit(0);
    }

    waitpid(pid, &status, 0);
    read(fd1, &c, 1);
    printf("%c", c);
    read(fd2, &c, 1);
    printf("%c", c);
    read(fd3, &c, 1);
    printf("%c", c);
    printf("\n");
    return 0;
}
```

A. What will the contents of baz.txt be after the program completes?

B. What will be printed on stdout?

```
A. Contents of baz.txt:                  B. Printed on stdout:
```

Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Fall 2004

Exam 2

Tuesday November 16, 2004

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 74 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. You may use any books or notes you like. You may use a calculator, but not a laptop, PDA, or any wireless devices. Good luck!

1 (10):	
2 (12):	
3 (12):	
4 (10):	
5 (6):	
6 (12):	
7 (12):	
TOTAL (74):	

Problem 1. (10 points):

This problem tests your understanding of programming in the presence of signals – in particular, race conditions. The questions are based on the library below which maintains a sorted list of integers.

The programs are compiled without optimization: the machine instructions executed are a straightforward translation of the C code. In particular, statements are never reordered.

Remember that `malloc` is not guaranteed to zero the memory it returns.

```
struct list {
    struct list *next;
    int value;
};
static struct list global_header = { NULL, 0 };

/* Find the largest datum < value */
static struct list *find_prev(int value) {
    struct list *prev = &global_header;
    while(prev->next) {
        if(prev->next->value >= value) break;
        prev = prev->next;
    }
    return prev;
}

/* Add ’value’ into the list */
void add_datum(int value) {
    struct list *prev = find_prev(value);
    struct list *node = malloc(sizeof(*node));
    struct list *next = prev->next;

    if(!node) return;

    prev->next = node;
    node->next = next;
    node->value = value;
}

/* Remove the first datum >= ’value’ */
void remove_datum(int value) {
    struct list *prev, *node, *next;

    prev = find_prev(value);
    node = prev->next;
    if(!node) return;
    next = node->next;

    prev->next = next;
    free(node);
}
```

```
/* Count the number of data in the list */
int count(void) {
    struct list *node = global_header.next;
    int cnt = 0;
    while(node) {
        cnt++;
        node = node->next;
    }
    return cnt;
}
```

A.

```
void handler(int sig) {
    printf("we have %d elements\n", count());
}
int main() {
    char buf[256];
    signal(SIGUSR2, handler);

    while(fgets(buf, sizeof(buf), stdin)) {
        add_datum(atoi(buf));
    }
    return 0;
}
```

Does this have a race condition that can cause the program to crash? Why or why not? (2 points)

B.

```
void handler(int sig) {
    printf("we have %d elements\n", count());
}
int main() {
    char buf[256];
    int n = 100; while(n > 0) { add_datum(n--); }
    signal(SIGUSR2, handler);

    while(fgets(buf, sizeof(buf), stdin)) {
        remove_datum(atoi(buf));
    }
    return 0;
}
```

Does this have a race condition that can cause the program to crash? Why or why not? (2 points)

C.

```
void handler(int sig) {
    add_datum(random());
}
int main() {
    char buf[256];
    signal(SIGUSR2, handler);

    while(fgets(buf, sizeof(buf), stdin)) {
        remove_datum(atoi(buf));
    }
    return 0;
}
```

This code has a race condition. While it won’t cause a crash, it could cause a memory leak. Why? (2 points)

D. Explain one way to avoid all the crashes and undesired behaviour in all the examples above. (4 points)

Problem 2. (12 points):

This problem concerns the way virtual addresses are translated into physical addresses. Imagine a system with the following parameters:

- Virtual addresses are 20 bits wide.
- Physical addresses are 18 bits wide.
- The page size is 4096 bytes.
- The TLB is 2-way set associative with 16 total entries.

The contents of the TLB and the first 32 entries of the page table are shown as follows. **All numbers are given in hexadecimal**.

TLB			
Index	Tag	PPN	Valid
0	16	13	1
	1B	2D	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0
4	06	08	1
	0F	19	1
5	0A	09	1
	1F	20	1
6	02	13	0
	18	12	1
7	0C	0B	0
	1E	24	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	2D	0
05	13	1	15	1B	0
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	1
0C	01	1	1C	1E	0
0D	15	1	1D	3E	1
0E	0C	0	1E	27	1
0F	14	0	1F	18	1

Part 1

1. The diagram below shows the bits of a virtual address. Please indicate the locations of the following fields by placing an ’X’ in the corresponding boxes of that field’s row. For example, if the virtual page offset were computed from the 2 most significant bits of the virtual address, you would mark the ’O’ (offset) column as shown:

	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
O	X	X																		

O The virtual page **o**ffset
N The virtual page **n**umber
I The TLB **i**ndex
T The TLB **t**ag

	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
O																				
N																				
I																				
T																				

2. The diagram below shows the format of a physical address. Please indicate the locations of the following fields by placing an ’X’ in the corresponding boxes of that field’s row.

O The physical page **o**ffset
N The physical page **n**umber

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
O																		
N																		

Part 2

For the given virtual addresses, please indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs. If there is a page fault, enter “-” for “PPN” and leave the physical address blank.

Virtual address: `0xD8AC3`

1. Virtual address (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. Address translation

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	PPN	0x

3. Physical address(one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Virtual address: `0x1665D`

1. Virtual address (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. Address translation

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	PPN	0x

3. Physical address(one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Problem 3. (12 points):

This problem tests your understanding of Unix process control. Consider the following C program. For space reasons, we are not checking error return codes, so assume that all functions return normally. Assume that `printf` is unbuffered.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
  int i = 0;
  pid_t pid1, pid2;

  if((pid1 = fork()) == 0) {
    i++;
    if((pid2 = fork()) == 0) {
      i++;
      printf("i: %d\n", ++i);
      exit(0);
    }
    printf("i: %d\n", i);
  }
  else {
    if(waitpid(pid1, NULL, 0) > 0) {
      printf("i: %d\n", ++i);
    }
  }
  printf("i: %d\n", ++i);
  exit(0);
}
```

Draw an **X** through any column which does not represent a valid possible output of this program.

i: 1 i: 3 i: 2 i: 1 i: 2	i: 1 i: 1 i: 2 i: 3	i: 1 i: 2 i: 3 i: 2 i: 3 i: 2	i: 3 i: 3 i: 3 i: 1 i: 2	i: 1 i: 2 i: 1 i: 2 i: 3	i: 1 i: 2 i: 3 i: 4 i: 2 i: 3

Problem 4. (10 points):

This problem tests your understanding of the the cache organization and performance. To simplify your reasoning, you may assume the following:

1. `sizeof(int) = 4`
2. x begins at memory address 0 and is stored in row-major order.
3. The cache is initially empty.
4. The only memory accesses are to the entries of the array x. All variables are stored in registers.
5. All code is compiled with -O0 flag (no optimizations).

Consider the following C code:

```
int x[2][128];
int i;
sum = 0;

for (i = 0; i < 128; i ++){
    sum += x[0][i] * x[1][i];
}
```

Case 1

1. Assume your cache is a 512-byte direct-mapped data cache with 16-byte cache blocks. What is the cache **miss rate**? (3 pts)

miss rate = _______%

2. If the cache were twice as big, what would be the miss rate? (2 pts)

miss rate = _______%

Case 2

1. Assume your cache is 512-byte 2-way set associative using an LRU replacement policy with 16-byte cache blocks. What is the cache miss rate? (3 pts)

miss rate = _______%

2. Will larger **cache size** help to reduce the miss rate? (Yes / No) (1 pt)

3. Will larger **cache line** help to reduce the miss rate? (Yes / No) (1 pt)

Problem 5. (6 points):

True/False:

1. A linker needs access to the source code to determine what global variables are referenced by a code file.
2. After a fork occurs, the memory updates made by the child process can affect the behavior of the parent process.
3. After a fork occurs, the file reads made by the child process can affect the behavior of the parent process.
4. When the `kill` function is invoked, a termination signal will be sent to the indicated process.
5. The availability of pointer casting in C makes it impossible for a garbage collector to identify all inaccessible data.
6. Implementing the free list as a doubly linked list makes it possible to implement the `free` operation in $O(1)$ time while still doing block coalescing.

Problem 6. (12 points):

This problem tests your understanding of basic cache operations. A frequent operation in image processing programs is to convolve a large array (= the image) with a small, constant matrix (= the kernel). Filter operations such as sharpening, blurring, edge-enhancement can be implemented by choosing the kernel elements. In the simplest case, the kernel is a 3x3 matrix.

For this problem, you should assume a very large $N \times N$ image array of `unsigned short`'s, where N happens to be an integral power of 2. The inner loop uses two int-pointers src and dst that scan the image array. There are two arrays: src is scanning the source image while dst is pointing to the resulting image after the convolution kernel has been applied. Thus the inner loop looks like this:

```
...
   unsigned short *src, *dst;
...
  { int new_pix;

    /* compute one pixel */
    new_pix  = src[     0] * A;

    new_pix += src[   - N] * B;
    new_pix += src[-1    ] * B;
    new_pix += src[ 1    ] * B;
    new_pix += src[   + N] * B;

    new_pix += src[-1 - N] * C;
    new_pix += src[ 1 - N] * C;
    new_pix += src[-1 + N] * C;
    new_pix += src[ 1 + N] * C;

    /* save the new pixel */
    *dst = new_pix;

    dst++;
    src++;
  }
...
```

This code utilizes the fact that many 3x3 kernels have rotational symmetry and have only 3 distinct values A, B, C. You should assume these values are kept in three registers. Likewise, the pointers src and dst are also stored in registers, as are the variables needed to control the inner loop. The target machine is fairly old and uses a write-through cache with no-write-allocate policy. Therefore, you do *not* need to worry about the write operations to the destination image array.

Each cache line on Harry's machine holds 4 `unsigned shorts` (8 Bytes). The cache size is 16 KBytes, which is too small to hold even one row of the image. Hint: each row has N elements, where N is a power of 2.

Figure 1 shows how this filter is scanning the source image array. The thick vertical bars represent the boundaries of cache lines: four consecutive horizontal squares are one cache line. The convolution kernel is represented by the 9 squares that are not marked with an X.

The 2 kernels shown in Figure 1 represent 2 successive iterations (case A and B) through the inner loop. The src pointer is incremented one cell at a time and moves from left to right in these pictures.

You shall mark each of the 9 squares those with either a ’H’ or a ’M’ indicating if the corresponding memory read operation hits (H) or misses (M) in the cache. Cells that contain an X do not belong to the convolution step that is being computed and you should not mark these.

Part 1

In this part, assume that the cache is organized as a direct mapped cache. Please mark the left column in Figure 1 with your answer. The right column may be used as scratch while you reason about your answer. We will grade the left column only.

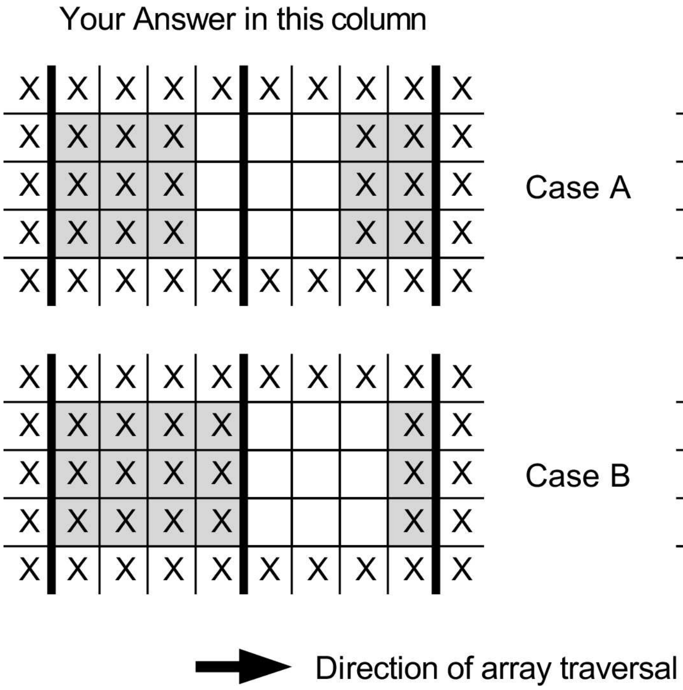


Figure 1: Convolution with a direct mapped cache

Part 2

In this part, assume a 3-way, set-associative cache with true Least Recently Used replacement policy (LRU). As in Part 1 of this question, please provide your answer by marking the empty squares of the left column in Figure 2 with your solution.

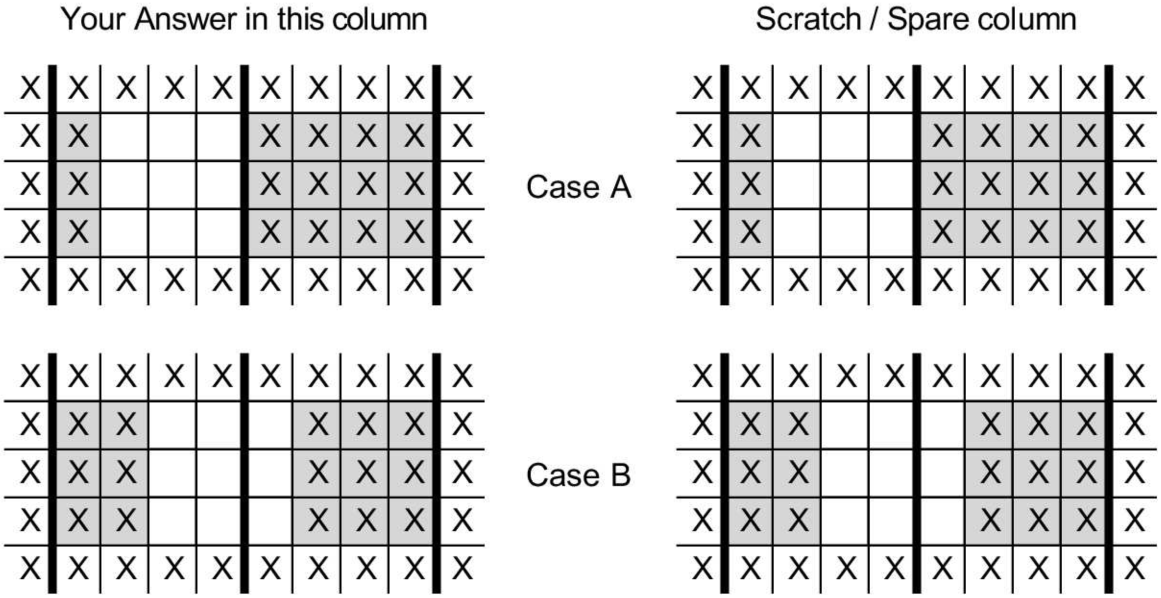


Figure 2: Convolution with a set associative cache

Problem 7. (12 points):

This problem tests your knowledge of block coalescing.

Harry Q. Bovik is trying to write his `malloc` implementation. Assume his allocator does following:

- Uses an implicit list of blocks.
- All headers, footers, and pointers are 4 bytes in size
- All memory blocks have a size of at least 16 bytes: (one header + one footer + payload of at least 8).
- All memory blocks have their *total* size rounded up to a multiple of 8.
- Allocated blocks consist of a header, a payload, and a footer.
- All free block coalescing happens in the `free()` function.
- The heap contains prologue and epilogue blocks which are allocated in the `malloc` initialization function (guaranteed to be called before any call to `malloc()` or `free()`). The prologue and epilogue are never freed although the epilogue is shifted appropriately when the heap is extended.

A. Simple Header (4 points)

Let the header store the size of the block in bytes. Since the block size is always a multiple of 8, the three least significant bits of the size are always 0. Harry decides to use these bits to store allocation info. The low order bit is set to 0 when the block is free and 1 when the block is allocated. To correctly access the size of the block's payload, he must mask the lower 3 bits to 0's. The footer value is set to match the header value.

Harry wants to make his `free()` function coalesce newly freed blocks with any and all adjacent free blocks.

How many memory reads must `free()` do to *decide whether to coalesce* this block with the ones around it? Assume that the only piece of data currently available is the base pointer of the block that is currently being freed. ___________

What is the maximum number of writes that `free()` must do to create the final coalesced block? ________

This question continues on the next page.

B. Packed Header (4 points)

Harry's implementation works, but it is a little slow. Like the clever 213 student that he is, he decides to try and decrease the number of memory accesses made by his program by utilizing all three free bits in the header:

- The header consists of the payload size, in bytes, OR'ed with the three flags described below. This can be done since the payload size is always a multiple of 8. To access the size of the block's payload, you must mask the lower 3 bits to 0's.
- Let the lower three bits be b_2, b_1, and b_0, respectively with b_0 corresponding to the least significant bit of the header.
- b_2 indicates whether the previous block on the heap is allocated. This bit is 1 when the previous block is allocated and 0 when the previous block is free. Assume that there is a prologue block that is maintained, so the first allocatable block on the heap will have this bit set to 1.
- b_1 indicates whether the next block on the heap is allocated. This bit is 1 when the next block is allocated and 0 when the next block is free. Assume that there is an epilogue block that is maintained, so the last allocatable block on the heap will have this bit set to 1.
- b_0 indicates whether this block on the heap is allocated. This bit is 1 when this block is allocated and 0 when this block is free.

The footer value is set to match the header value.

How many memory reads must `free()` do to *decide whether to coalesce* this block with the ones around it? Assume that the only piece of data currently available is the base pointer of the block that is currently being freed. __________

What is the maximum number of writes that `free()` must do to create the final coalesced block? ________

C. Custom Footer (4 points)

Consider the header format specified in Part B. But now, instead of having the footer mirror the header value, Harry uses the footer to store a *pointer* to the header of the block.

How many memory reads must `free()` do to *decide whether to coalesce* this block with the ones around it? Assume that the only piece of data currently available is the base pointer of the block that is currently being freed. __________

What is the maximum number of writes that `free()` must do to create the final coalesced block? ________

Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Fall 2005

Exam 2

Tuesday Nov 22, 2005

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 51 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. You may use any books or notes you like. No electronic devices are allowed. Good luck!

1 (04):	
2 (09):	
3 (10):	
4 (10):	
5 (10):	
6 (08):	
TOTAL (51):	

Problem 1. (4 points):

This problem tests your understanding of how Linux represents and shares files. You are asked to show what each of the following programs prints as output:

- Assume that file `infile.txt` contains the ASCII text characters ”`123456`”.
- The system function `int dup(int oldfd)` is a variant of `dup2` that copies descriptor `oldfd` to the lowest-numbered unused descriptor, and then returns the index of the new descriptor. For example, suppose that the lowest-numbered unused descriptor is 5. Then `newfd = dup(3)` copies descriptor 3 to descriptor 5, returns the integer value 5, and assigns it to variable `newfd`.

A.

```
int main() {
    int fd1, fd2;
    char c;

    fd1 = open("infile.txt", O_RDONLY, 0);
    fd2 = open("infile.txt", O_RDONLY, 0);

    read(fd1, &c, 1);
    read(fd2, &c, 1);

    printf("c = %c\n", c);
    exit(0);
}
```

c = ______________

B.

```
int main() {
    int fd1, fd2;
    char c;

    fd1 = open("infile.txt", O_RDONLY, 0);
    fd2 = dup(fd1);

    read(fd1, &c, 1);
    read(fd2, &c, 1);

    printf("c = %c\n", c);
    exit(0);
}
```

c = ______________

Problem 2. (9 points):

This problem will test your knowledge of process control and signals. Each program below produces a single output line each time it runs. However, because of non-determinism in the scheduling of processes and signals, each run may produce different output lines. You are asked to list all possible output lines.

A. (3 pts) Assume that `main()` calls the following function `test()` exactly once.

```
void test(void)
{
  if ( fork() == 0 )
  {
      printf(‘‘0’’);
      exit(0);
  }
  printf(‘‘1’’);
}
```

List all possible output lines:

B. (3 pts) Assume that `main()` calls the following function `test()` exactly once.

```
void test(void)
{
  int status;
  int counter = 0;

  if ( fork() == 0 )
  {
    counter++;
    printf(‘‘%d’’, counter);
    exit(0);
  }
  wait(&status);

  if ( fork() == 0 )
  {
    counter++;
    printf(‘‘%d’’, counter);
    exit(0);
  }
  wait(&status);
}
```

List all possible output lines:

C. (3 pts) Assume that `main()` calls the following function `test()` exactly once. Assume that `sigusr1_handler()` is installed as the signal handler for SIGUSR1 and that `block_all_signals()` uses `sigprocmask()` to block all signals.

```
void sigusr1_handler(int n)
{
  printf(‘‘1’’);
  exit(0);
}

void test(void)
{
  int i, status;

  for (i = 0; i < 3; i++)
  {
    pid_t pid = fork();
    if ( pid == 0 )
    {
      block_all_signals();
      printf(‘‘0’’);
      exit(0);
    }
    kill( pid, SIGUSR1 );
    wait(&status);
  }
}
```

List all possible output lines:

Problem 3. (10 points):

The following problem concerns basic cache lookups.

- The memory is byte addressable.
- Memory accesses are to **1-byte words**
- Physical addresses are 13 bits wide.
- The cache is 4-way set associative, with a 4-byte block size and 8 sets.

In the following tables, **all numbers are given in hexadecimal**. The *Index* column contains the set index for each set of 4 lines. The *Tag* columns contain the tag value for each line. The *V* column contains the valid bit for each line. The *Bytes 0–3* columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

The contents of the cache are as follows:

4-way Set Associative Cache												
Index	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3
0	F0	1	ED 32 0A A2	8A	1	BF 80 1D FC	14	1	EF 09 86 2A	BC	0	25 44 6F 1A
1	BC	0	03 3E CD 38	A0	0	16 7B ED 5A	BC	1	8E 4C DF 18	E4	1	FB B7 12 02
2	BC	1	54 9E 1E FA	B6	1	DC 81 B2 14	00	0	B6 1F 7B 44	74	0	10 F5 B8 2E
3	BE	0	2F 7E 3D A8	C0	1	27 95 A4 74	C4	0	07 11 6B D8	BC	0	C7 B7 AF C2
4	7E	1	32 21 1C 2C	8A	1	22 C2 DC 34	BC	1	BA DD 37 D8	DC	0	E7 A2 39 BA
5	98	0	A9 76 2B EE	54	0	BC 91 D5 92	98	1	80 BA 9B F6	BC	1	48 16 81 0A
6	38	0	5D 4D F7 DA	BC	1	69 C2 8C 74	8A	1	A8 CE 7F DA	38	1	FA 93 EB 48
7	8A	1	04 2A 32 6A	9E	0	B1 86 56 0E	CC	1	96 30 47 F2	BC	1	F8 1D 42 30

Part 1

A) Warmup problem

What is the (data) size of this cache in bytes?

Answer: `C = _________ bytes`

B) 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

12	11	10	9	8	7	6	5	4	3	2	1	0

Part 2

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”.

Hint: Pay attention to those valid bits!

Physical address: `0x71A`

Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0

Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Physical address: `0x16E8`

Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0

Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Part 3

For the given contents of the cache, list the eight hex physical memory addresses that will **hit in Set 2**.

To save space, you should express contiguous addresses as a range. For example, you would write the four addresses `0x1314`, `0x1315`, `0x1316`, `0x1317` as `0x1314--0x1317`.

Answer: ____

The following templates are provided as scratch space:

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

Problem 4. (10 points):

This problem requires you to analyze the cache behavior of two small segments of code that access an $N \times M$ integer matrix arr. For this problem, $N = 4$ and $M = 3$. Assume that the loop variables i and j, the accumulator variable sum, and the temporary variables $temp1$ and $temp2$ are all stored in registers.

```
#define N 4
#define M 3

int arr[N][M];

int arr_sum1()
{
  int i, j;
  int sum = 0;

  for (i=0; i<N; i+=2){
    for (j=0; j<M; j++){
      int temp1 = arr[i][j];
      int temp2 = arr[i+1][j];
      sum += (temp1 * temp2);
    }
  }

  return sum;
}
```

We are interested in how this function will interact with a simple cache. Assume that the cache is cold when the function is called and that the array has been initialized elsewhere. **The cache is direct-mapped with two sets and a block size of 8 bytes.** Assume that arr is aligned so that the first two elements are stored in the same cache block. Fill out the table below to indicate if the corresponding memory access in arr will be a hit (**H**) or a miss (**M**).

arr	Col 0	Col 1	Col 2
Row 0			
Row 1			
Row 2			
Row 3			

Now consider a slightly different function that accesses the same integer matrix arr. Again, assume **i**, **j**, **sum**, **temp1**, and **temp2** are all stored in registers.

```
#define N 4
#define M 3

int arr[N][M];

int arr_sum2()
{
  int i, j;
  int sum = 0;

  for(i=0; i<N/2; i++){
    for(j=0; j<M; j++){
      int temp1 = arr[i][j];
      int temp2 = arr[i+(N/2)][j];
      sum += (temp1 * temp2);
    }
  }

  return sum;
}
```

Once again, fill out the table below to indicate if the corresponding memory access in arr will be a hit (**H**) or a miss (**M**). Assume the cache is cold when the function is called and the array has already been initialized. **Like the previous problem, the cache is direct-mapped with two sets and a block size of 8 bytes.**

arr	Col 0	Col 1	Col 2
Row 0			
Row 1			
Row 2			
Row 3			

Problem 5. (10 points):

The following problem concerns virtual memory and the way virtual addresses are translated into physical addresses. Below are the specifications of the system on which the translation occurs.

- Memory is byte addressable and memory accesses are to 4-byte words.
- The system is configured with 256MB of virtual memory.
- The system has only 64MB of physical memory.
- The page size is 4KB.
- The TLB is 2-way set associative with 8 total sets.

The contents of the TLB and the relevant sections of the page tables are shown below. In the following tables, **all numbers are given in hexadecimal**.

TLB			
Index	Tag	PPN	Valid
0	0003	03EB	1
	1A10	0D46	0
1	0107	0AD3	1
	1D01	052F	0
2	0106	0D4E	1
	1213	01D3	0
3	0301	005C	0
	0A0B	0231	1
4	1211	0819	1
	0108	03D8	0
5	011F	0218	1
	1102	0A4A	1
6	1FE7	0263	1
	103F	0FAF	0
7	0211	030D	0
	10D2	0310	0

Page Table		
VPN	PPN	Valid
0837	1457	0
0838	0D31	0
0839	0AD3	1
0840	035A	1
0841	16F3	0
0842	0D4E	1
0843	031D	1
0844	078F	1
0845	0487	1
0846	0819	0
0847	136B	1
0848	04BA	1
0849	0D33	0
0850	0061	0
0851	0328	0
0852	0F42	0

A. Part 1 - Warmup questions

(a) How many bits are needed to represent the virtual address space?

(b) How many bits are needed to represent the physical address space?

(c) What is the total number of page table entries?

B. Part 2 - Virtual memory address translation

(a) Please step through the following address translation. You may indicate a page fault by entering a '–' for Physical Page Number and Physical Address.

Parameter	Value
Virtual Address Accessed:	0x844044
Virtual Page Number:	0x
TLB Index:	0x
TLB Tag:	0x
TLB Hit or Miss:	
Physical Page Number:	0x
Physical Address:	0x

Please use the layout below as scratch space if necessary.

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

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

(b) Please step through the following address translation. You may indicate a page fault by entering in ’-’ for Physical Page Number and Physical Address.

Parameter	Value
Virtual Address Accessed:	0x839108
Virtual Page Number:	0x
TLB Index:	0x
TLB Tag:	0x
TLB Hit or Miss:	
Physical Page Number:	0x
Physical Address:	0x

Please use the layout below as scratch space as necessary.

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

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

Performance Evaluation

The following code evaluates a polynomial of degree `degree` over `x` for a set of coefficients `coeff`.

```
/* Unroll 2x */
data_t peval2(data_t coeff[], data_t x, int degree)
{
    data_t result = 0;
    int i;

    /* Unroll by 2X */
    for (i = degree; i >= 1; i -= 2) {
        /* Version 1 Main Loop Computation */
        result = (((result * x) + coeff[i]) * x) + coeff[i-1];
    }

    /* Finish off remaining element(s) */
    for (; i >= 0; i -= 1) {
        result = result * x + coeff[i];
    }
    return result;
}
```

The code uses loop unrolling to compute two terms of the polynomial per iteration. It uses a technique known as *Horner's Rule* to reduce the number of multiplications and additions. Data type `data_t` can be defined to different types using a `typedef` declaration.

Running on an Intel Pentium 4, and with `data_t` defined to be `float`, this code achieves a CPE (Cycles Per Element) of 12.0. Considering that this machine has a latency of 7 cycles for single-precision multiplication, and 5 for single-precision addition, we can see that the CPE is dictated by these latencies, the data dependencies in the main loop (the line labeled "`Version 1 Main Loop Computation`"), and the fact that each iteration computes two "elements".

Problem 6. (8 points):

Below are four alternative versions of the main loop computation. For each version, write down the CPE that will result. The possible choices are: 6.0, 8.5, 9.5, and 12.0. You can determine the answer knowing only the operation latencies and the data dependencies that each line entails. **You can ignore other factors such as the issue time and the number of functional units.**

A. Version 2

```
result = (((result * x) * x) + (coeff[i] * x)) + coeff[i-1];
```

CPE = ________________

B. Version 3

```
result = ((result * (x * x)) + (coeff[i] * x)) + coeff[i-1];
```

CPE = ________________

C. Version 4

```
result = ((result * x) * x) + ((coeff[i] * x) + coeff[i-1]);
```

CPE = ________________

D. Version 5

```
result = (result * (x * x)) + ((coeff[i] * x) + coeff[i-1]);
```

CPE = ________________

Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Fall 2006

Exam 2

Wednesday Nov 15, 2006

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 55 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. You may use any books or notes you like. Calculators are allowed, but no other electronic devices. Good luck!

1 (11):	
2 (08):	
3 (08):	
4 (12):	
5 (08):	
6 (08):	
TOTAL (55):	

Problem 1. (11 points):

Consider the following C function:

```
data_t psum(data_t a[], data_t b[], data_t c[], int cnt)
{
    data_t r = 0;
    int i;
    for (i = 0; i < cnt; i++) {
        /* Inner loop expression */
        r =   r  +  a[i]  *  b[i]  +  c[i]  ;
    }
    return r;
}
```

In this code, data type `data_t` can be defined to different types using a `typedef` declaration.

According to the C rules for operator precedence and associativity, the line labeled “Inner loop expression” will be computed according to the following parenthesization:

```
        r =  (r  + (a[i]  *  b[i]))+  c[i]  ;
```

In all, there are 5 different parenthesizations for this expression, They will not all compute the same result.

Imagine we run this code on a machine in which multiplication requires 7 cycles, while addition requires 5. Assume that these latencies are the only factors constraining the performance of the program. Don't worry about the cost of memory references or integer operations, resource limitations, etc.

A. For each parenthesization listed below, write down the CPE that the function would achieve. **Hint:** All of your answers will be in the set $\{5, 7, 10, 12, 14, 15, 17, 19\}$.

```
// P1.  CPE =
r = ((r  +  a[i]) *  b[i]) +  c[i]  ;

// P2.  CPE =
r =  (r  + (a[i]  *  b[i]))+  c[i]  ;

// P3.  CPE =
r =   r  +((a[i]  *  b[i]) +  c[i]) ;

// P4.  CPE =
r =  (r  +  a[i]) * (b[i]  +  c[i]) ;

// P5.  CPE =
r =   r  + (a[i]  * (b[i]  +  c[i]));
```

B. Of the parenthesizations that give the same result as the original function, which has the best CPE? Assume that addition and multiplication are associative for data type `data_t`.

Problem 2. (8 points):

This problem requires you to analyze the cache behavior of a function that sums the elements of an array A:

```
int A[2][4];

int sum()
{
    int i, j, sum=0;

    for (j=0; j<4; j++) {
        for (i=0; i<2; i++) {
            sum += A[i][j];
        }
    }
    return sum;
}
```

Assume the following:

- The memory system consists of registers, a single L1 cache, and main memory.
- The cache is cold when the function is called and the array has been initialized elsewhere.
- Variables i, j, and sum are all stored in registers.
- The array A is aligned in memory such that the first two array elements map to the same cache block.
- `sizeof(int) == 4`.
- The cache is direct mapped, with a block size of 8 bytes.

A. Suppose that the cache consists of 2 sets. Fill out the table to indicate if the corresponding memory access in A will be a hit (**h**) or a miss (**m**).

A	Col 0	Col 1	Col 2	Col 3
Row 0	**m**			
Row 1				

B. What is the pattern of hits and misses if the cache consists of 4 sets instead of 2 sets?

A	Col 0	Col 1	Col 2	Col 3
Row 0	**m**			
Row 1				

Problem 3. (8 points):

This problem tests your understanding of the the cache organization and performance. Assume the following:

1. `sizeof(int) = 4`
2. Array `x` begins at memory address 0.
3. The cache is initially empty.
4. The only memory accesses are to the entries of the array `x`. All variables are stored in registers.

Consider the following C code:

```
int x[128];
int i, j;
int sum = 0;

for (i = 0; i < 64; i ++){
    j = i + 64;
    sum += x[i] * x[j];
}
```

Case 1

1. Assume your cache is a 256-byte direct-mapped data cache with 8-byte cache blocks. What is the cache **miss rate**? (2 pts)

miss rate = _______%

2. If the cache were twice as big, what would be the miss rate? (1 pts)

miss rate = _______%

Case 2

1. Assume your cache is 256-byte 2-way set associative using an LRU replacement policy with 8-byte cache blocks. What is the cache miss rate? (3 pts)

miss rate = _______%

2. Will larger **cache size** help to reduce the miss rate? (Yes / No) (1 pt)

3. Will larger **cache line** help to reduce the miss rate? (Yes / No) (1 pt)

Problem 4. (12 points):

Imagine a system with the following attributes:

- The system has 1MB of virtual memory
- The system has 256KB of physical memory
- The page size is 4KB
- The TLB is 2-way set associative with 16 total entries.

The contents of the TLB and the first 32 entries of the page table are given below. **All numbers are in hexadecimal**.

TLB			
Index	Tag	PPN	Valid
0	16	13	1
	1B	2D	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0
4	06	08	1
	0F	19	1
5	0A	09	1
	1F	20	1
6	02	13	0
	18	12	1
7	0C	0B	0
	1E	24	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	2D	0
05	13	1	15	1B	0
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	1
0C	01	1	1C	1E	0
0D	15	1	1D	3E	1
0E	0C	0	1E	27	1
0F	14	0	1F	18	1

A. Warmup Questions

(a) How many bits are needed to represent the virtual address space? ______

(b) How many bits are needed to represent the physical address space? ______

(c) What is the total number of page table entries? ______

B. Virtual Address Translation I

Please step through the following address translation. Indicate a page fault by entering '-' for Physical Address.

Virtual address: `0xFAA3F`

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	Physical Address	0x

Use the layout below as scratch space for the virtual address bits. To allow us to give you partial credit, clearly mark the bits that correspond to the VPN, TLB index (TLBI), and TLB tag (TLBT).

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(Please go to the next page for part C)

C. Virtual Address Translation II

Please step through the following address translation. Indicate a page fault by entering '-' for Physical Address.

Virtual address: `0x162A4`

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	Physical Address	0x

Use the layout below as scratch space for the virtual address bits. To allow us to give you partial credit, clearly mark the bits that correspond to the VPN, TLB index (TLBI), and TLB tag (TLBT).

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Problem 5. (8 points):

This problem tests your understanding of pointer arithmetic, pointer dereferencing, and malloc implementation.

Harry Q. Bovik has implemented a simple explicit-list allocator. You may assume that his implementation follows the usual restrictions that you had to comply with in L6, such as the 8-byte alignment rule.

The following is a description of Harry's block structure:

HDR	PAYLOAD	FTR

- HDR - Header of the block (4 bytes)
- PAYLOAD - Payload of the block (arbitrary size)
- FTR - Footer of the block (4 bytes)

The size of the **payload** of each block is stored in the header and the footer of the block. Since there is an 8-byte alignment requirement, the least significant of the 3 unused bits is used to indicate whether the block is free (0) or allocated (1).

This problem consists of two parts.

For the first part of the problem, you can assume that:

- `sizeof(int) == 4 bytes`
- `sizeof(char) == 1 byte`
- `sizeof(short) == 2 bytes`
- `sizeof(long) == 4 bytes`
- The size of any pointer (e.g. `char *`) is 4 bytes.

Note that for the first part Harry is working on a 32-bit machine. Also, assume that the block pointer `bp` points to the first byte of the payload.

Part One. Your task is to help Harry compute the correct payload size (using the function `get_payload_size()`), by indicating which of the following implementations of the `GET_HDR` macro are corect. For each of the proposed solutions listed below, fill in the blank with either **C** for correct, or **I** for incorrect.

```
/* get_payload_size returns the actual size of payload.
   bp is pointing to the first byte of a block
   returned from Harry’s malloc() */

#define GET_HDR(p)       ??
#define GET_SIZE(p)      (GET_HDR(p) & ~0x7)

int get_payload_size(void *bp)
{
    return (int)(GET_SIZE(bp));
}

/* (1) */
   #define GET_HDR(p)  (*(int *)((int *)(p) - 1))    ________

/* (2) */
   #define GET_HDR(p)  (*(int *)((char *)(p) - 1))   ________

/* (3) */
   #define GET_HDR(p)  (*(int *)((char **)(p) - 1))  ________

/* (4) */
   #define GET_HDR(p)  (*(char *)((int)(p) - 1))     ________

/* (5) */
   #define GET_HDR(p)  (*(long *)((long *)(p) - 1)   ________

/* (6) */
   #define GET_HDR(p)  (*(int *)((int)(p) - 4))      ________

/* (7) */
   #define GET_HDR(p)  (*(int *)((short)(p) - 2))    ________

/* (8) */
   #define GET_HDR(p)  (*(short *)((int *)(p) - 1))  ________
```

Part Two. Harry now wants to port his `GET_HDR` macro to a 64-bit machine. On 64-bit machines,

- `sizeof(long) == 8 bytes`
- The size of any pointer (e.g. `char *`) is 8 bytes.

and the other types remain the same. As before, for each proposed solution, fill in the blanks with either **C** for correct, or **I** for incorrect.

```
#define GET_HDR(p)   ??
#define GET_SIZE(p)  (GET_HDR(p) & ~0x7)

int get_payload_size(void *bp)
{
    return (int)(GET_SIZE(bp));
}

/* (1) */
   #define GET_HDR(p)  (*(int *)((int *)(p) - 1))     _________

/* (2) */
   #define GET_HDR(p)  (*(int *)((char *)(p) - 1))    _________

/* (3) */
   #define GET_HDR(p)  (*(int *)((char **)(p) - 1))   _________

/* (4) */
   #define GET_HDR(p)  (*(char *)((int)(p) - 1))      _________

/* (5) */
   #define GET_HDR(p)  (*(long *)((long *)(p) - 1)    _________

/* (6) */
   #define GET_HDR(p)  (*(int *)((int)(p) - 4))       _________

/* (7) */
   #define GET_HDR(p)  (*(int *)((short)(p) - 2))     _________

/* (8) */
   #define GET_HDR(p)  (*(short *)((int *)(p) - 1))   _________
```

Problem 6. (8 points):

This question will test your understanding of Unix process control and signals. Consider the following C program. For space reasons, we are not checking error return codes. You can assume that all functions return normally.

```
int val = 3;
void Exit(int val) {
    printf("%d", val);
    exit(0);
}

void usr1_handler(int sig) {
    Exit(val);
}

int main() {
    int pid;

    signal(SIGUSR1, usr1_handler);

    if ((pid = fork()) == 0) {
        setpgid(0, 0);

        if (fork())
            Exit(val + 1);
        else
            Exit(val - 1);
    }

    kill(-pid, SIGUSR1);
}
```

List all the outputs possible from this program:

______________   ______________   ______________   ______________   ______________

______________   ______________   ______________   ______________   ______________

______________   ______________   ______________   ______________   ______________

Andrew login ID:____________________________________

Full Name:____________________________________

Recitation Section:____________________________________

CS 15-213, Fall 2008
Exam 2

Thurs. Oct 30, 2008

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–H) on the front.
- Write your answers in the space provided for the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 60 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. You may use any books or notes you like. No calculators or other electronic devices are allowed.
- Good luck!

1 (6):	
2 (9):	
3 (6):	
4 (8):	
5 (10):	
6 (6):	
7 (7):	
8 (8):	
TOTAL (60):	

Problem 1. (6 points):

In **buflab**, you performed various buffer overflow attacks against a vulnerable function `gets` that writes into a small buffer. However, in practice, a decent compiler (such as `gcc`) warns about the vulnerabilities of `gets`, and most programmers tend to take the advice.

Harry Q. Bovik thinks that his code is invulnerable against buffer overflow attacks as long as he stays away from unsafe functions such as `gets`.

Here is a piece of code Bovik wrote; it compiled without warnings under a 32-bit little-endian machine:

```
// str.c (headers omitted)

int main()
{
    char buf[7];
    scanf("%s", buf);
    return 0;
}

void remove_later()
{
    printf("You have found my weakness!!!\n");
}
```

Your goal is to prove Bovik wrong by jumping to the `remove_later` function. Do not worry about how the program would behave upon exiting the function.

Relevant assembly output from `objdump` of the `str` program:

```
080483c0 <main>:
 80483c0:   55                      push   %ebp
 80483c1:   89 e5                   mov    %esp,%ebp
 80483c3:   83 ec 28                sub    $0x28,%esp
 80483c6:   83 e4 f0                and    $0xfffffff0,%esp
 80483c9:   8d 45 e8                lea    0xffffffe8(%ebp),%eax
 80483cc:   83 ec 10                sub    $0x10,%esp
 80483cf:   89 44 24 04             mov    %eax,0x4(%esp)
 80483d3:   c7 04 24 e8 84 04 08    movl   $0x80484e8,(%esp)
 80483da:   e8 f5 fe ff ff          call   80482d4 <scanf@plt>
 80483df:   c9                      leave
 80483e0:   31 c0                   xor    %eax,%eax
 80483e2:   c3                      ret

080483f0 <remove_later>:
 80483f0:   55                      push   %ebp
 80483f1:   89 e5                   mov    %esp,%ebp
 80483f3:   83 ec 08                sub    $0x8,%esp
 80483f6:   c7 04 24 eb 84 04 08    movl   $0x80484eb,(%esp)
 80483fd:   e8 c2 fe ff ff          call   80482c4 <puts@plt>
 8048402:   c9                      leave
 8048403:   c3                      ret
```

Assume that you are allowed to work under the same directory where Bovik created `str`, and you are executing `./hex2raw < exploit | ./str`, where `exploit` contains your attack code in hexadecimal.

Write down the contents of your `exploit`, and use `[n]` to denote `n` consecutive arbitrary bytes:

Problem 2. (9 points):

Consider the following C function to sum all the elements of a 5×5 matrix. Note that it is iterating over the matrix **column-wise**.

```
char sum_matrix(char matrix[5][5]) {
  int row, col;
  char sum = 0;
  for (col = 0; col < 5; col++) {
    for (row = 0; row < 5; row++) {
      sum += matrix[row][col];
    }
  }
  return sum;
}
```

Suppose we run this code on a machine whose memory system has the following characteristics:

- Memory is byte-addressable.
- There are registers, an L1 cache, and main memory.
- A char is stored as a single byte.
- The cache is direct-mapped, with 4 sets and 2-byte blocks.

You should also assume:

- `matrix` begins at address 0.
- `sum`, `row` and `col` are in registers; that is, the only memory accesses during the execution of this function are to `matrix`.
- The cache is initially cold and the array has been initialized elsewhere.

Fill in the table below. In each cell, write "**h**" if there is a cache hit when accessing the corresponding element of the matrix, or "**m**" if there is a cache miss.

	0	1	2	3	4
0					
1					
2					
3					
4					

Problem 3. (6 points):

Using pointers

Give the output for the following code snippet, assuming that it was compiled on an IA-32 machine. Variable i, j, and k have memory addresses 200, 300 and 400, respectively.

```
#include <stdio.h>

int main() {
  // Assume that i is stored at memory address 200
  int i = 4;
  // Assume that j is stored at memory address 300
  int *j = &i;
  // Assume that k is stored at memory address 400
  int *k = (int *) i;

  printf("%d,%d,%d", (int) i, (int) &i, (int) (i+1));
  printf("\n");

  printf("%d,%d,%d", (int) j, (int) &j, (int) (j+1));
  printf("\n");

  printf("%d,%d,%d", (int) k, (int) &k, (int) (k+1));
  printf("\n");
  return 0;
}
```

This program prints out three lines. Each line has three values that are separated by a comma. What is the output?

Problem 4. (8 points):

Consider the following C program, with line numbers:

```
1   int main() {
2     int counter = 0;
3     int pid;
4
5     while (counter < 2 && !(pid = fork())) {
6       counter++;
7       printf("%d", counter);
8     }
9
10    if (counter > 0) {
11      printf("%d", counter);
12    }
13
14    if (pid) {
15      waitpid(pid, NULL, 0);
16      counter += 4;
17      printf("%d", counter);
18    }
29  }
```

Use the following assumptions to answer the questions:

- All processes run to completion and no system calls will fail.
- `printf()` is atomic and calls `fflush(stdout)` after printing argument(s) but before returning.
- Logical operators such as && evaluate their operands from left to right and only evaluate the smallest number of operands necessary to determine the result.

A. List all possible outputs of the program in the following blanks.

(You might not use all the blanks.)

B. If we modified line 10 of the code to change the `>` comparison to `>=`, it would cause the program flow to print out zero counter values. With this change, how many possible outputs are there?

(Just give a number, you do not need to list them all.)

```
NEW NUMBER OF POSSIBLE OUTPUTS = __________
```

Problem 5. (10 points):

Consider the following C program:

```
void handler1(int sig) {
    printf("Pirate\n");
    exit(0);
}

int main()
{
    pid_t pid1;

    signal(SIGUSR1, handler1);

    if((pid1 = fork()) == 0) {
        printf("Ghost\n");
        exit(0);
    }
    kill(pid1, SIGUSR1);
    printf("Ninja\n");
    return 0;
}
```

Use the following assumptions to answer the questions:

- All processes run to completion and no system calls will fail.
- `printf()` is atomic and calls `fflush(stdout)` after printing argument(s) but before returning.

Mark each column that represents a valid possible output of this program with 'Yes' and each column which is impossible with 'No'.

Ninja Pirate Ghost	Ghost Ninja Pirate	Ninja Ghost	Ninja Pirate Ninja	Pirate Ninja

Problem 6. (6 points):

Consider a system with 10 GB of physical memory (with a 4 KB page size) and a 50 GB disk drive with the following characteristics:

- 512-byte sectors
- 800 sectors/track
- 15,000 RPM (i.e., 4ms to complete one full revolution)
- 3ms average seek time

Imagine an application that MALLOC()s nearly 50 GB of space, initializes it to all zeros, and then randomly selects integers from across the full space and increments them. (Assume that there are no other processes.)

A. What percentage of the integers selected would result in page faults? ______

B. What is the average time to service a page fault? (round to the nearest millisecond) ______

C. Approximately how many integers can be incremented per second? (again, rounding is fine) ______

D. If an additional 15 GB of physical memory were available, how many integers could be incremented incremented per second? (again, rounding is fine) ______

Problem 7. (7 points):

Consider the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    char c;
    int file1 = open("buffer.txt", O_RDONLY);
    int file2;
    int file3 = open("buffer.txt", O_RDONLY);

    read(file1, &c, 1);
    file2 = dup(file1);
    read(file1, &c, 1);
    read(file2, &c, 1);

    printf("1 = %c\n", c);

    int pid = fork();
    if(pid == 0) {
        read(file3, &c, 1);
        printf("2 = %c\n", c);

        dup2(file2, file3);
        close(file1);
        read(file3, &c, 1);
        printf("3 = %c\n", c);

        file1 = open("buffer.txt", O_RDONLY);
        read(file1, &c, 1);
        printf("4 = %c\n", c);
    } else {
        waitpid(pid, NULL, 0);
        printf("5 = %c\n", c);

        read(file3, &c, 1);
        printf("6 = %c\n", c);
        close(file2);
        dup2(file1, file2);
        read(file1, &c, 1);
        printf("7 = %c\n", c);
    }
    return 0;
}
```

Assume that the disk file `buffer.txt` contains the string of bytes `COMPUTER` . Also assume that all system calls succeed. What will be output when this code is compiled and run? You may not need all the lines in the table given below.

Output Line Number	**Output**
1^{st} line of output	
2^{nd} line of output	
3^{rd} line of output	
4^{th} line of output	
5^{th} line of output	
6^{th} line of output	
7^{th} line of output	

Problem 8. (8 points):

Imagine a system with the following attributes:

- The system has 1MB of virtual memory
- The system has 256KB of physical memory
- The page size is 4KB
- The TLB is 2-way set associative with 8 total entries.

The contents of the TLB and the first 32 entries of the page table are given below. **All numbers are in hexadecimal**.

TLB			
Index	Tag	PPN	Valid
0	05	13	1
	3F	15	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	13	1
05	13	0	15	1B	1
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	0
0C	01	0	1C	1E	0
0D	15	0	1D	3E	1
0E	0C	0	1E	27	1
0F	2B	1	1F	18	1

A. Warmup Questions

(a) How many bits are needed to represent the virtual address space? ______

(b) How many bits are needed to represent the physical address space? ______

(c) How many bits are needed to represent a page table offset? ______

B. Virtual Address Translation I

Please step through the following address translation. Indicate a page fault by entering '-' for Physical Address.

Virtual address: `0x15213`

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	Physical Address	0x

Use the layout below as scratch space for the virtual address bits. To allow us to give you partial credit, clearly mark the bits that correspond to the VPN, TLB index (TLBI), and TLB tag (TLBT).

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(Please go to the next page for part C)

C. Virtual Address Translation II

Please step through the following address translation. Indicate a page fault by entering '-' for Physical Address.

Virtual address: `0x7E213`

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	Physical Address	0x

Use the layout below as scratch space for the virtual address bits. To allow us to give you partial credit, clearly mark the bits that correspond to the VPN, TLB index (TLBI), and TLB tag (TLBT).

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Andrew login ID:______________________________________

Full Name:______________________________________

Recitation Section:______________________________________

CS 15-213/18-243, Fall 2009
Exam 2

Thursday, October 29th, 2009

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–J) on the front.
- **Do not write any part of your answers outside of the space given below each question. Write clearly and at a reasonable size. If we have trouble reading your handwriting you will receive no credit on that problem.**
- The exam has a maximum score of XXX 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. You may use any books or notes you like. No calculators or other electronic devices are allowed.
- Good luck!

1 (21):	
2 (4):	
3 (15):	
4 (16):	
5 (12):	
6 (16):	

TOTAL (84):	

Problem 1. (21 points):

1. What is the most likely immediate result of executing the following code:

   ```
   int foo[10]
   int *p = (int *) malloc(4*sizeof(int));
   p = p - 1;
   *p = foo[0];
   ```

   (a) Initialize the first array element to 4

   (b) Segmentation fault

   (c) Reset the pointer p to point to the array named foo

   (d) Corruption of malloc header information

2. What is the maximum number of page faults per second that can be serviced in a system that has a disk with an average access time of 10ms?

   (a) 10

   (b) 100

   (c) 50

   (d) Depends on the percentage of memory accesses that are page faults

3. Why does Count Dracula not have to worry about his program's memory addresses overlapping those of other processes run on the same system?

   (a) Each process has its own page table

   (b) The linker carefully lays out address spaces to avoid overlap

   (c) The loader carefully lays out address spaces to avoid overlap

   (d) He does need to worry

4. Dr. Frankenstein has a disk that rotates at 7,200 RPM (8ms per full revolution), has an average seek time of 5ms, and has 1000 sectors per track. How long (approximately) does the average 1-sector access take?

   (a) Not enough information to determine the answer

   (b) 13ms

   (c) 9ms

   (d) 10.5ms

5. How many times does exec() return?

   (a) 0

   (b) 1

   (c) 2

   (d) 0 or 1, depending on whether or not an error occurs

6. Which of the following is **not** a default action for any signal type?

   (a) The process terminates.

   (b) The process reaps the zombies in the waitlist.

   (c) The process stops until restarted by a SIGCONT signal.

   (d) The process ignores the signal.

   (e) The process terminates and dumps core.

7. Imagine a process (called ”process A”) that calls fork() three times. If all three child processes terminate before process A is picked by the kernel to be run again, how many times could process A receive SIGCHLD?

   (a) 0

   (b) 1

   (c) 3

   (d) 1 or 3

   (e) Not enough information to determine

Problem 2. (4 points):

1. Consider the following program compiled for x86-64:

```
#include <malloc.h>

int main()
{
    int a = 0;
    int *b = malloc(sizeof(int));

    if ((&a) > b) {
        printf("Trick!\n");
    } else {
        printf("Treat!\n");
    }

    return 0;
}
```

What does this program print out and why? (You can assume that the malloc() call does not fail)

Problem 3. (15 points):

You are provided with several files, each of which contains a simple text string without any whitespace or special characters. The list of files with their respective contents is given below:

one.txt	`abc`
two.txt	`nidoking`
three.txt	`conflageration`

You are also presented with the `main()` function of three small programs (header includes omitted), each of which uses simple and familiar functions that perform file i/o operations. For each program, determine what will be printed on `stdout` based on the code and the contents of the file. Assume that calls to `open()` succeed, and that each program is run from the directory containing the above files. (The program execution order does not matter; the programs are independent.)

Program 1:

```
void main() {
  char c0 = ’x’, c1 = ’y’, c2 = ’z’;
  int r, r2 = open("one.txt", O_RDONLY);

  read(r2, &c0, 1);
  r = dup(r2);
  read(r2, &c1, 1);
  close(r2);
  read(r,  &c2, 1);

  printf("%c%c%c", c0, c1, c2);
}
```

output to `stdout` from Program 1:	

Program 2:

```
void main() {
  char c0 = ’x’, c1 = ’y’, c2 = ’z’;
  char scrap[4];
  int pid, r, r2 = open("two.txt", O_RDONLY);
  r = dup(r2);

  if (!(pid = fork())) {
    read(r, &c0, 1);
    close(r2);
    r2 = open("two.txt", O_RDONLY);
    read(r2, &scrap, 4);
  } else {
    waitpid(pid, NULL, 0);
    read(r,  &c1, 1);
    read(r2, &c2, 1);
  }

  printf("%c%c%c", c0, c1, c2);
}
```

output to `stdout` from Program 2:	

Program 3:

```
void main() {
  char c[3] = {’x’, ’y’, ’z’};
  int r, r2, r3;

  r  = open("three.txt", O_RDONLY);
  r2 = open("three.txt", O_RDWR);
  dup2(1, r3);
  dup2(r2, 1);

  read(r, &c[0], 1);
  printf("elephant");
  fflush(stdout);
  read(r,   &c[1], 1);
  read(r2,  &c[2], 1);
  write(r3, &c[0], 3);

  printf("%c%c%c", c[0], c[1], c[2]);
}
```

output to `stdout` from Program 3:	

Problem 4. (16 points):

Your evil TA Punter Hitelka has redesigned the fish machines to make buflab impossible! Normally, on x86 systems, a program's stack grows down, to lower memory addresses, making a called function have a **lower** stack address than the calling function. The new fish machines have stack frames that grow up, this means that a called function has a **higher** stack address than the calling function.

For example, under stack-down convention, having main() call foo() would create

```
0x0f0 +---------------+
      |  main's stack |
      |      frame    |
0x0e0 +---------------+
      |  foo's stack  |
      |      frame    |  | Stack Growing Down |
0x0d0 +---------------+  V                    V
```

Under the new stack-up convention, having main() call foo() would create

```
0x110 +---------------+
      |  foo's stack  | ^                  ^
      |     frame     | | Stack Growing Up |
0x100 +---------------+
      |  main's stack |
      |     frame     |
0x0f0 +---------------+
```

This means that a push instruction would increment %esp, and a pop instruction would decrement %esp. Bufflab now contains the following function, which Punter claims to be un-exploitable:

```
int exploitMe(){
    char password[100];

    /*prompt the user for the password*/
    printf("what is the password?\n");

    /*read it in*/
    gets(password);

    printf("You shall not pass!\n");
    return false;
}
```

1

First, let's go back to the old model of the stack growing down. Please draw a stack diagram from the perspective of the gets() function. Assume that main() calls exploitMe.

```
+-------------------------------------+
|         Ret Addr to Main             |
+-------------------------------------+
|                                      |  Stack Growing Down! |
|                                      |                      V
|                                      |
|                                      |
```

2

Describe a buffer overflow exploit you could use to make exploitMe return true if the stack grew down. You do not need to write the exploit, just describe how it would work.

3

Now, draw the stack diagram under the new stack-grows-up scheme. (Hint: this should be very easy given the answer to part 1). We will grade this based off your answer to part 1)

```
|                                     |
|                                     |                    ^
|                                     |  Stack Growing UP! |
+-------------------------------------+
|         Ret Addr to Main            |
+-------------------------------------+
```

4

Is it possible that Punter is wrong and this is exploitable? If so, please describe an exploit that would make exploitMe return true. Otherwise explain why it is impossible.

Problem 5. (12 points):

Consider the following C program, with line numbers:

```
1    int main() {
2      int counter = 0;
3      int pid;
4
5      if( !(pid = fork()) ) {
6        while((counter < 2) && (pid = fork()) ) {
7            counter++;
8            printf("%d", counter)
9        }
10       if (counter > 0) {
11           printf("%d", counter);
12       }
13     }
14     if(pid) {
15       waitpid(pid, NULL, 0);
16      counter = counter << 1;
17      printf("%d", counter)
18     }
19   }
```

Use the following assumptions to answer the questions:

- All processes run to completion and no system calls will fail.
- `printf()` is atomic and calls `fflush(stdout)` after printing argument(s) but before returning.
- Logical operators such as `&&` evaluate their operands from left to right and only evaluate the smallest number of operands necessary to determine the result.

A. List all possible outputs of the program in the following blanks.

(You might not use all the blanks.)

________________________________   ________________________________

________________________________   ________________________________

________________________________   ________________________________

________________________________   ________________________________

________________________________   ________________________________

B. If we modified line 10 of the code to change the > comparison to >=, it would cause the program flow to print out zero counter values. With this change, how many possible outputs are there?

(Just give a number, you do not need to list them all.)

```
NEW NUMBER OF POSSIBLE OUTPUTS = ____________
```

Problem 6. (16 points):

Your friend Goger Ganberg was inspired by shell lab, and has, in an effort to better understand process management, written the following program:

```
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>

extern int do_stuff();

void magic()
{
    if (fork()) exit(0);
    signal(SIGHUP, SIG_IGN);
    setpgid(0, 0);
    chdir("/");
}

int main()
{
    magic();
    return do_stuff();
}
```

The main functionality of the program is implemented in `do_stuff`, defined in another file, but he seems more intent on showing off the `magic` function called beforehand.

Suppose that you run Goger's program in a shell whose process ID is 1000, and the program is assigned process ID 1001.

1. Assume the call to `fork` in this program returns 1002. What will be the PIDs of all processes that run `do_stuff` after `magic` returns, and for each one, what will be the PID of its parent?

2. Suppose `do_stuff` takes a long time to execute. Will the shell next emit a prompt only after `do_stuff` returns, or might it do so sooner? Why?

3. Assume now that `fork` instead returns -1. What will be different from the case in which `fork` succeeds? What will be the same?

The `SIGHUP` signal is delivered to a process when its controlling terminal goes away, and the default action is to exit. For example, if you open a terminal and run `vim` in it, then close the terminal, you would want your now-useless `vim` process to not stick around.

4. What is the purpose of the second and third lines of `magic`?

5. **Bonus question** (1 point): What sort of program would use a function like `magic`? (An example is sufficient.)

6. **Bonus question** (1 point): What else might it be useful for a function such as `magic` to do?

Andrew login ID:____________________________________

Full Name:____________________________________

Section:____________________________________

15-213/18-243, Fall 2010

Exam 2

Tuesday, November 9. 2010

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew login ID, full name, and section on the front.
- This exam is closed book, closed notes, although you may use a single 8 1/2 x 11 sheet of paper with your own notes. You may not use any electronic devices.
- 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 67 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Good luck!

1 (10):
2 (06):
3 (12):
4 (09):
5 (09):
6 (12):
7 (09):
TOTAL (67):

Problem 1. (10 points):

Multiple choice on a variety of topics. Write the correct answer for each question in the following table:

1	2	3	4	5	6	7	8	9	10

1. For the Unix linker, which of the following accurately describes the difference between global symbols and local symbols?

   (a) There is no functional difference as to how they can be used or how they are declared.

   (b) Global symbols can be referenced by other modules (files), but local symbols can only be referenced by the module that defines them.

   (c) Global symbols refer to variables that are stored in `.data` or `.bss`, while local symbols refer to variables that are stored on the stack.

   (d) Both global and local symbols can be accessed from external modules, but local symbols are declared with the "static" keyword.

2. Consider the following C variable declaration

   ```
   int *(*f[3])();
   ```

   Then `f` is

   (a) an array of pointers to pointers to functions that return int

   (b) a pointer to an array of functions that return pointers to int

   (c) a function that returns a pointer to an array of pointers to int

   (d) an array of pointers to functions that return pointers to int

   (e) a pointer to a function that returns an array of pointers to int

   (f) a pointer to an array of pointers to functions that return int

   Hint: Recall from the K&R book that `[]` and `()` have higher precedence than `*`.

3. Which of the following is true concerning dynamic memory allocation?

   (a) External fragmentation is caused by chunks which are marked as allocated but actually cannot being used.

   (b) Internal fragmentation is caused by padding for alignment purposes and by overhead to maintain the heap data structure (such as headers and footers).

   (c) Coalescing while traversing the list during calls to malloc is known as immediate coalescing.

   (d) Garbage collection, employed by calloc, refers to the practice of zeroing memory before use.

For the next 3 questions, consider the following code running on a 32-bit Linux system. Recall that `calloc` zeros the memory that it allocates.

```
int main()
{
    long a, *b, c;
    char **p;

    p = calloc(8, sizeof(char)); /* calloc returns 0x1dce1000 */

    a = (long) (p + 0x100);
    b = (long *) (*p + 0x200);
    c = (int) (b + 0x300);

    printf("p=%p a=%x b=%p c=%x\n", p, a, b, c);

    exit(0);
}
```

4. When `printf` is called, what is the hex value of variable a?

   (a) Can't tell

   (b) 0x1dce1100

   (c) 0x1dce1400

   (d) 0x1dce1800

5. When `printf` is called, what is the hex value of variable b?

   (a) Can't tell

   (b) 0x1dce1200

   (c) 0x1dce2000

   (d) 0x200

   (e) 0x800

6. When `printf` is called, what is the hex value of variable c?

   (a) Can't tell

   (b) 0x1dce2a00

   (c) 0x1dce4400

   (d) 0xc00

   (e) 0xe00

7. Which of the following is **not** a default action for any signal type?

   (a) The process terminates.

   (b) The process reaps the zombies in the waitlist.

   (c) The process stops until restarted by a SIGCONT signal.

   (d) The process ignores the signal.

   (e) The process terminates and dumps core.

8. A system uses a two-way set-associative cache with 16 sets and 64-byte blocks. Which set does the byte with the address `0xdeadbeef` map to?

   (a) Set 7

   (b) Set 11

   (c) Set 13

   (d) Set 14

9. When it succeeds, `execve` is called once and returns how many times?

   (a) 0

   (b) 1

   (c) 2

   (d) 3

10. When it suceeds, `fork` is called once and returns how many times?

   (a) 0

   (b) 1

   (c) 2

   (d) 3

Problem 2. (6 points):

Linking. Consider the executable object file `a.out`, which is compiled and linked using the command

```
  unix> gcc -o a.out main.c foo.c
```

and where the files `main.c` and `foo.c` consist of the following code:

```
/* main.c */
#include <stdio.h>

static int a = 1;
int b = 2;
int c;

int main()
{
    int c = 3;

    foo();
    printf("a=%d, b=%d, c=%d\n", a, b, c);
    return 0;
}
```

```
/* foo.c */
int a, b, c;

void foo()
{
    a = 4;
    b = 5;
    c = 6;
}
```

What is the output of `a.out`?

Answer: `a=____, b=_____, c=_____`

Problem 3. (12 points):

Cache Operation In this problem, you are asked to simulate the operation of a cache. You can make the following assumptions:

- There is only one level of cache
- Physical addresses are 8 bits long ($m = 8$)
- The block size is 4 bytes ($B = 4$)
- The cache has 4 sets ($S = 4$)
- The cache is direct mapped ($E = 1$)

(a) What is the total capacity of the cache? (in number of data bytes)

(b) How long is a tag? (in number of bits)

(c) Assuming that the cache starts clean (all lines invalid), please fill in the following tables, describing what happens with each operation. Addresses are given in both hex and binary for your convenience.

Operation	Set index?	Hit or Miss?	Eviction?
load 0x00 $(0000\ 0000)_2$	0	miss	no
load 0x04 $(0000\ 0100)_2$		miss	
load 0x08 $(0000\ 1000)_2$			
store 0x12 $(0001\ 0010)_2$	0		
load 0x16 $(0001\ 0110)_2$			
store 0x06 $(0000\ 0110)_2$			
load 0x18 $(0001\ 1000)_2$	2		
load 0x20 $(0010\ 0000)_2$			
store 0x1A $(0001\ 1010)_2$			

Problem 4. (9 points):

Signals. Consider the following three different snippets of C code. Assume that all functions and procedures return correctly and that all variables are declared and initialized properly. Also, assume that an arbitrary number of SIGINT signals, and only SIGINT signals, can be sent to the code snippets randomly from some external source.

For each code snippet, circle the value(s) of `i` that could possibly be printed by the `printf` command at the end of each program. *Careful: There may be more than one correct answer for each question. Circle all the answers that could be correct.*

Code Snippet 1:

```
int i = 0;

void handler(int sig) {
  i = 0;
}

int main() {
  int j;

  signal(SIGINT, handler);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  printf("i = %d\n", i);
  exit(0);
}
```

1. Circle possible values of `i` printed by snippet 1:

1. 0
2. 1
3. 50
4. 100
5. 101
6. Terminates with no output.

Code Snippet 2:

```
int i = 0;

void handler(int sig) {
  i = 0;
}

int main () {
  int j;
  sigset_t s;

  signal(SIGINT, handler);

  /* Assume that s has been
     initialized and declared
     properly for SIGINT */

  sigprocmask(SIG_BLOCK, &s, 0);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  sigprocmask(SIG_UNBLOCK, &s, 0);
  printf("i = %d\n", i);
  exit(0);
}
```

2. Circle possible values of `i` printed by snippet 2:

1. 0
2. 1
3. 50
4. 100
5. 101
6. Terminates with no output.

Code Snippet 3:

```
int i = 0;

void handler(int sig) {
  i = 0;
  sleep(1);
}


int main () {
  int j;
  sigset_t s;

  /* Assume that s has been
     initialized and declared
     properly for SIGINT */

  sigprocmask(SIG_BLOCK, &s, 0);
  signal(SIGINT, handler);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  printf("i = %d\n", i);
  sigprocmask(SIG_UNBLOCK, &s, 0);
  exit(0);
}
```

3. Circle possible values of `i` printed by snippet 3:

1. 0
2. 1
3. 50
4. 100
5. 101
6. Terminates with no output.

Problem 5. (9 points):

Unix process control. Consider the C program below. For space reasons, we are not checking error return codes, so assume that all functions return normally.

Part 1

```
int main () {
    if (fork() == 0) {
        if (fork() == 0) {
            printf("9");
            exit(1);
        }
        else
            printf("5");
    }
    else {
        pid_t pid;
        if ((pid = wait(NULL)) > 0) {
             printf("3");
        }
    }
    printf("0");
    return 0;
}
```

For each of the following strings, circle whether (Y) or not (N) this string is a possible output of the program.

A. 93050 Y N

B. 53090 Y N

C. 50930 Y N

D. 39500 Y N

E. 59300 Y N

Part 2

Consider the C program below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int i = 0;

int main () {
    int j;
    pid_t pid;

    if ((pid = fork()) == 0) {
        for (j = 0; j < 20; j++)
            i++;
    }
    else {
        wait(NULL);
        i = -1;
    }

    if (i < 0)
        i = 10;

    if (pid > 0)
        printf("Parent: i = %d\n", i);
    else
        printf("Child: i = %d\n", i);

    exit(0);
}
```

What are the outputs of the two `printf` statements?

`Parent: i =` __________

`Child : i =` __________

Problem 6. (0xc points):

Address translation. This problem deals with virtual memory address translation using a multi-level page table, in particular the 2-level page table for a 32-bit Intel system with 4 KByte pages tables. The following diagrams are direct from the Intel System Programmers guide and should be used on this problem:

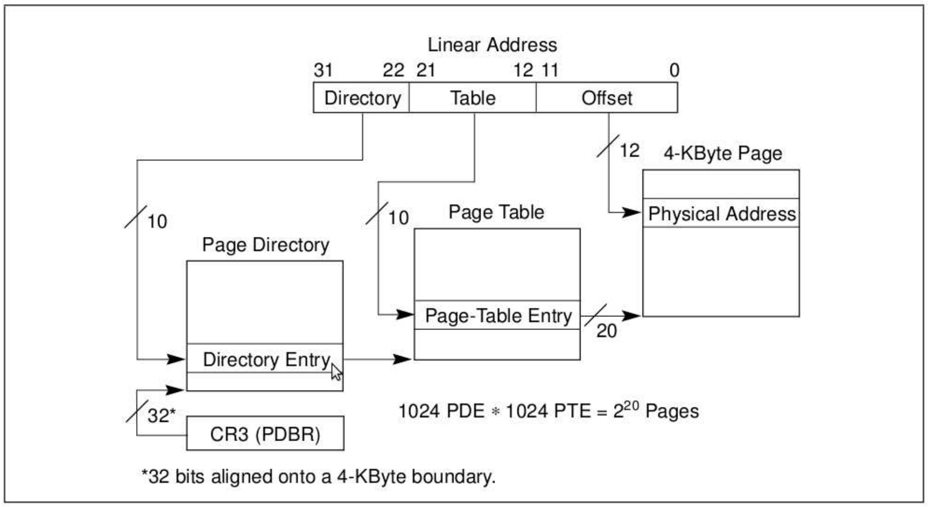


Figure 3-12. Linear Address Translation (4-KByte Pages)

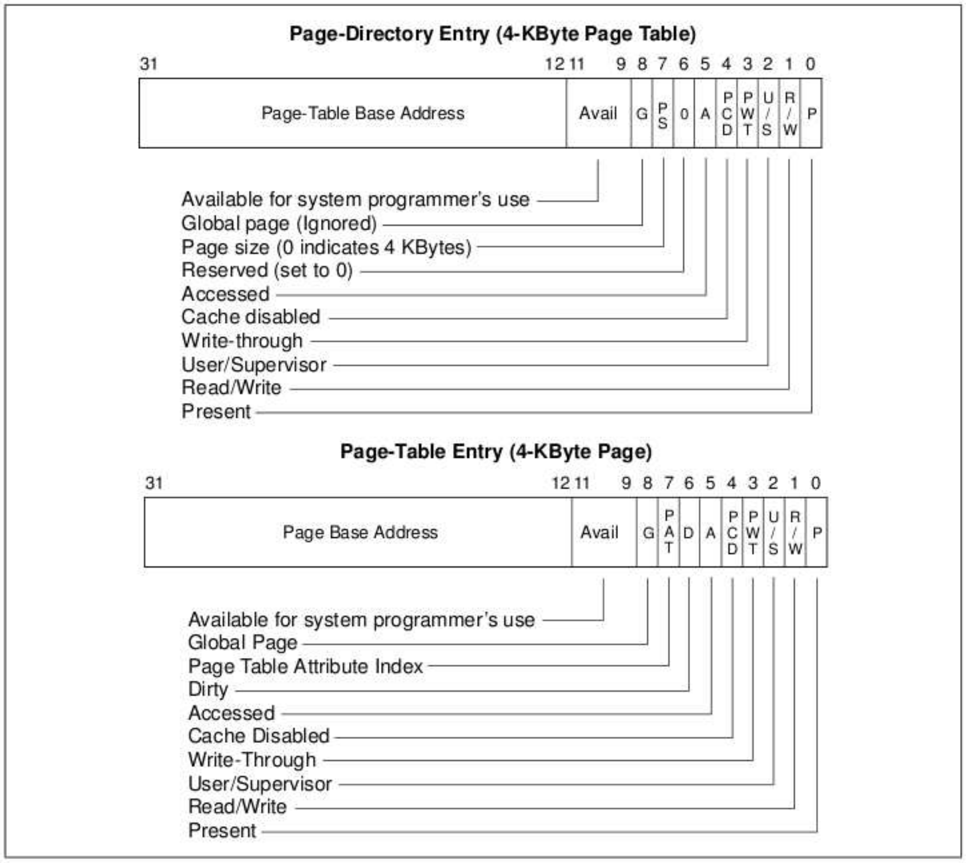


Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

The contents of the relevant sections of memory are shown on this page. All numbers are given in **hexadecimal**. Any memory not shown can be assumed to be zero. The Page Directory Base Address is `0x0c23b000`.

For each of the following problems, perform the virtual to physical address translation. If an error occurs at any point in the address translation process that would prevent the system from performing the lookup, then indicate this by circling FAILURE and noting the physical address of the table entry that caused the failure.

For example, if you were to detect that the present bit in the PDE is set to zero, then you would leave the PTE address in (b) empty, and circle FAILURE in (c), noting the physical address of the offending PDE.

Address	**Contents**
00023000	beefbee0
00023120	12fdc883
00023200	debcfd23
00023320	d2e52933
00023FFF	bcdeff29
00055004	8974d003
0005545c	457bc293
00055460	457bd293
00055464	457be293
0c23b020	01288b53
0c23b040	012aab53
0c23b080	00055d01
0c23b09d	0FF2d303
0c23b274	00023d03
0c23b9fc	2314d222
2314d200	0fdc1223
2314d220	d21345a9
2314d4a0	d388bcbd
2314d890	00b32d00
24AEE520	b58cdad1
29DE2504	56ffad02
29DE4400	2ab45cd0
29DE9402	d4732000
29DEE500	1a23cdb0

1. Read from virtual address `0x080016ba`.
   Scratch space:

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

(a) Physical address of PDE: 0x

(b) Physical address of PTE: 0x

(c) (SUCCESS) The physical address accessed is: 0x

OR

(FAILURE) The physical address of the table entry causing the failure is: 0x

2. Read from virtual address `0x9fd28c10`. Scratch space:

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

(a) Physical address of PDE: 0x

(b) Physical address of PTE: 0x

(c) (SUCCESS) The physical address accessed is: 0x

OR

(FAILURE) The physical address of the table entry causing the failure is: 0x

Problem 7. (9 points):

Unix I/O. You are given two text files `file_1.txt` and `file_2.txt`. Each of them contains exactly a single word without any whitespace or special characters. Their contents are shown in the table below.

File name	File contents
file_1.txt	`file`
file_2.txt	`descriptors`

You are also given two programs (headers omitted), both of which are independent of one another, i.e., the order of execution does not matter. They use the simple and familiar UNIX system functions to perform file I/O operations.

For each program, determine what will be the output to `stdout`, based on the file contents as shown above. Assume that after each program finish execution, the file contents will be reset to that shown above. Also, assume that all system calls will succeed and the files are in the same directory as the two programs.

Program 1: (4 points)

```
/* buf is initialized to be all zeroes */
char buf[20] = {0};

int main(int argc, char* argv[]) {
    int fd1, fd2 = open("file_1.txt", O_RDONLY);

    fd1 = dup(fd2);
    read(fd2, buf, 3);
    close(fd2);
    read(fd1, &buf[3], 1);

    printf("%s", buf);

    /* Don't worry about file descriptors not being closed */
    return 0;
}
```

output to `stdout` from Program 1:	

Program 2: (5 points)

```
/* buf is initialized to be all zeroes */
char buf[20] = {0};

int main(int argc, char* argv[]) {
    pid_t pid;
    int fd1, fd2, fd3 = open("file_2.txt", O_RDONLY);

    read(fd3, buf, 1);
    fd1 = dup(fd3);

    if ((pid = fork()) > 0){
        waitpid(pid, NULL, 0);
        read(fd1, &buf[1], 2);
    } else {
        fd2 = open("file_2.txt", O_RDONLY);
        read(fd1, &buf[1], 1);
        read(fd2, &buf[2], 2);
    }

    printf("%s", buf);

    /* Don’t worry about file descriptors not being closed */
    return 0;
}
```

output to `stdout` from Program 2:

Full Name:_______________________________________

CS 15-213, Spring 2002

Exam 2

March 28, 2002

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 54 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. You may use any books or notes you like. You may use a calculator, but no laptops or other wireless devices. Good luck!

1 (4):	
2 (12):	
3 (13):	
4 (7):	
5 (6):	
6 (12):	
TOTAL (54):	

In this problem, you will compare the performance of direct mapped and 4-way associative caches for the initialization of 2-dimensional array of data structures. Both caches have a size of 1024 bytes. The direct mapped cache has 64-byte lines while the 4-way associative cache has 32-byte lines.

You are given the following definitions

```
typedef struct{
    float irr[3];
    short theta;
    short phi;
} photon_t;

photon_t  surface[16][16];
register int i, j, k;
```

Also assume that

- `sizeof(short) = 2`
- `sizeof(float) = 4`
- `surface` begins at memory address `0`
- Both caches are initially empty
- The array is stored in row-major order
- Variables `i`,`j`,`k` 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 ++)
    {
        for(k = 0; k < 3; k ++)
        {
            surface[i][j].irr[k] = 0.;
        }
        surface[i][j].theta = 0;
        surface[i][j].phi = 0;
    }
}
```

Miss rate for writes to surface:_______________%

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 surface: _______________%

```
for (i = 0; i < 16; i ++)
{
    for (j = 0; j < 16; j ++)
    {
        for (k = 0; k < 16; k ++)
        {
            surface[j][i].irr[k] = 0;
        }
        surface[j][i].theta = 0;
        surface[j][i].phi = 0;
    }
}
```

Miss rate for writes to surface:________________%

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 surface:________________%

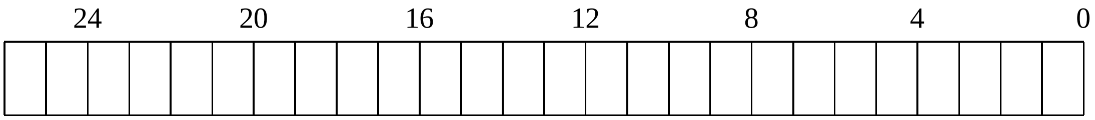
The following problem concerns various aspects of virtual memory.

Part I.

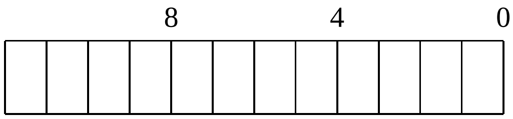
The following are attributes of the machine that you will need to consider:

- Memory is byte addressable
- Virtual Addresses are 26 bits wide
- Physical Addresses are 12 bits wide
- Pages are 512 bytes
- Each Page Table Entry contains:
  - Physical Page Number
  - Valid Bit

A. The box below shows the format of a virtual address. Indicate the bits used for the VPN (Virtual Page Number) and VPO (Virtual Page Offset).



B. The box below shows the format for a physical address. Indicate the bits used for the PPN (Physical Page Number) and PPO (Physical Page Offset)



C. **Note:** For the questions below, answers of the form 2^i are acceptable. Also, please note the units of each answer

How much *virtual* memory is addressable? ____________________ bytes

How much *physical* memory is addressable? ____________________ bytes

How many bits is each Page Table Entry? ____________________ bits

How large is the Page Table? ____________________ bytes

(4 points)

Application images for the operating system used on the machine in part I are formed with a subset of ELF. They only contain the `.text` and `.data` regions.

When a process uses `fork()` to create a new process image in memory, the operating system maintains each process' view that it has full control of the virtual address space. To the programmer, the amount of physical memory used by the two processes together is twice that which is used by a single process.

****NOTE** Read both questions below before answering**

A. How can the operating system conservatively save physical memory when creating the new process image during a call to `fork()` with respect to the `.text` and `.data` regions?

B. Imagine a process that acts in the following fashion:

```
int my_array[HUGE_SIZE];

int main() {

  /* Code to initialize my_array */

  if(fork() == 0) {
    exit(0);
  } else {

    /* Code that calculates and prints
     * the sum of the elements in my_array
     */
  }
}
```

How could the operating system be aggressive by temporarily saving memory beyond what was saved in part A in this case? *Hint 1: Note that the child doesn't change my_array, but the operating system has to be prepared for such an event since it doesn't know the future. Hint 2: Think about protection bits and page faults.*

The following problem concerns the way virtual addresses are translated into physical addresses.

- The memory is byte addressable, and memory accesses are to 1-byte {**not 4-byte**} words.
- Virtual addresses are 17 bits wide.
- Physical addresses are 12 bits wide.
- The page size is 256 bytes.
- The TLB is 4-way set associative with 16 total entries.
- The cache is 2-way set associative, with a 4-byte line size and 64 total entries.

In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB and the page table for the first 32 pages, and the cache are as follows:

TLB			
Index	Tag	PPN	Valid
0	55	6	0
	48	F	1
	00	C	0
	77	9	1
1	01	4	1
	32	A	1
	02	F	0
	73	0	1
2	02	3	1
	0F	B	0
	04	3	0
	26	C	0
3	00	8	1
	7A	2	1
	21	1	0
	17	E	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
000	C	0	010	1	1
001	7	1	011	8	1
002	3	1	012	3	0
003	8	1	013	E	1
004	0	0	014	6	0
005	5	0	015	C	0
006	C	1	016	7	0
007	4	1	017	2	1
008	D	1	018	9	1
009	F	0	019	A	0
00A	3	1	01A	B	0
00B	0	1	01B	3	1
00C	0	0	01C	2	1
00D	F	1	01D	9	0
00E	4	0	01E	5	0
00F	7	1	01F	B	1

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	7A	1	09	EE	12	64	00	0	99	04	03	48
1	02	0	60	17	18	19	38	1	00	BC	0B	37
2	55	1	30	EB	C2	0D	0B	0	8F	E2	05	BD
3	07	1	03	04	05	06	5D	1	7A	08	03	22
4	12	0	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	B0	39	D3	F7
6	91	1	A0	B7	26	2D	F0	0	0C	71	40	10
7	46	0	B1	0A	32	0F	DE	1	12	C0	88	37

1. The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

VPO	The virtual page offset
VPN	The virtual page number
TLBI	The TLB index
TLBT	The TLB tag

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. 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:

PPO	The physical page offset
PPN	The physical page number
CO	The Cache Block Offset
CI	The Cache Index
CT	The Cache Tab

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

For the given virtual addresses, indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs.

If there is a cache miss, enter “-” for “Cache Byte Returned.” If there is a page fault, enter “-” for “PPN” and leave part C blank.

Virtual address: `01FAD`

1. Virtual address format (one bit per box)

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

3. Physical address format (one bit per box)

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

4. Physical memory reference

Parameter	Value
Block Offset	0x
Cache Index	0x
Cache Tag	0x
Cache Hit? (Y/N)	
Value of Cache Byte Returned	0x

Consider the C program below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main ()
{
  int j = 0;

  for ( j = 0; j < 3; j++ )
    {
      if ( fork() == 0 )
        printf ( "%i", j + 1 );
      else
        printf ( "%i", j + 4 );

      printf ( "%i", j );
    }

  fflush ( stdout );
  return 0;
}
```

Tell how many of each number will be printed:

0 ________________

1 ________________

2 ________________

3 ________________

4 ________________

5 ________________

6 ________________

The following program performs a shell-like file copying. It forks out a child process to copy the file. The parent process waits for the child process to finish and, at the same time, captures the SIGINT signal. The signal handler asks the user whether to kill the copying or not. If the user answers ’y’, the child process will be killed. Otherwise, the signal handler just silently returns from the signal handler.

```
 1: pid_t   child_pid = 0;

 2: void  handler(int sig)
 3: {
 4:    char answer;

 5:    printf("Stop copying?\n");
 6:    scanf("%c", &answer);
 7:    if (answer == ’y’ && child_pid != 0)

 8:         _____________________________________________________;
 9: }

10: void  copy()
11: {

12:     _____________________________________________________;

13:    if ((child_pid = fork()) == 0) {

14:         __________________________________________________;
15:         copy_file();
16:         exit(0);
17:    }
18:    else {

19:        __________________________________________________;
20:        if (wait(NULL) == child_pid) {
21:            printf("Child process %d ended.\n", child_pid);
22:        }
23:    }
24: }
```

A. Fill in the blank lines to make the code work. Please fill in ”(empty)” if no code is needed.

B. What happens if the user presses Ctrl-C when the program execution is at line 6? And why?

Introduction

Below we have provided the source code for a simple C Language program. Additionally we have provided an excerpt of its disassembled assembly language source code, its Procedure Linkage Table (PLT), and snapshots of its Global Offset Table (GOT) at three different stages of its execution.

C Source Code:

```
 1. #include <stdio.h>
 2. #include <math.h>
 3.
 4. int main (int argc, char *argv)
 5. {
 6.   char *cptr;
 7.
 8.   cptr = (char *) malloc (1024);
 9.
10.   sprintf (cptr, "Hi Mom\n");
11.   printf ("%s\n", cptr);
12.   fflush (stdout);
13.
14.   return 0;
15. }
```

Questions

A. Please label each entry in the PLT as one of the following:

   (a) `malloc()`

   (b) `sprintf()`

   (c) `printf()`

   (d) `fflush()`

   (e) code to invoke the dynamic linker, itself

   (f) none of the above

B. Please label each entry in the first GOT as one of the following:

   (a) `malloc()`

   (b) `sprintf()`

   (c) `printf()`

   (d) `fflush()`

   (e) code to invoke the dynamic linker, itself

   (f) none of the above

C. Each of the three GOTs provided was extracted within gdb at three different stages of the program's execution. Please indicate when each snapshot might have been taken by indicating the last line to completely execute before the snapshot was recorded. If the snapshot was recorded before or after main() executes, please indicate this by recording ”before execution”, ”before main” or ”after main” in place of a line number.

```
int main (int argc, char *argv)
{
    080484a4 <main>:
     80484a4:       55                      push   %ebp
     80484a5:       89 e5                   mov    %esp,%ebp
     80484a7:       83 ec 14                sub    $0x14,%esp
     80484aa:       53                      push   %ebx

 char *cptr;
 cptr = (char *) malloc (1024);

     80484ab:       83 c4 f4                add    $0xfffffff4,%esp
     80484ae:       68 00 04 00 00          push   $0x400
     80484b3:       e8 d8 fe ff ff          call   8048390 <_init+0x60>
     80484b8:       89 c3                   mov    %eax,%ebx

 sprintf (cptr, "Hi Mom\n");
     80484ba:       83 c4 f8                add    $0xfffffff8,%esp
     80484bd:       68 48 85 04 08          push   $0x8048548
     80484c2:       53                      push   %ebx
     80484c3:       e8 08 ff ff ff          call   80483d0 <_init+0xa0>

 printf ("%s\n", cptr);
     80484c8:       83 c4 20                add    $0x20,%esp
     80484cb:       83 c4 f8                add    $0xfffffff8,%esp
     80484ce:       53                      push   %ebx
     80484cf:       68 50 85 04 08          push   $0x8048550
     80484d4:       e8 e7 fe ff ff          call   80483c0 <_init+0x90>

 fflush (stdout);
     80484d9:       a1 40 96 04 08          mov    0x8049640,%eax
     80484de:       83 c4 f4                add    $0xfffffff4,%esp
     80484e1:       50                      push   %eax
     80484e2:       e8 99 fe ff ff          call   8048380 <_init+0x50>

 return 0;
     80484e7:       8b 5d e8                mov    0xffffffe8(%ebp),%ebx
     80484ea:       31 c0                   xor    %eax,%eax
     80484ec:       89 ec                   mov    %ebp,%esp
     80484ee:       5d                      pop    %ebp
     80484ef:       c3                      ret
}
```

This entry belongs to ____

```
8048360:       ff 35 78 95 04 08       pushl  0x8049578
8048366:       ff 25 7c 95 04 08       jmp    *0x804957c
804836c:       00 00                   add    %al,(%eax)
804836e:       00 00                   add    %al,(%eax)
```

This entry belongs to ____

```
8048370:       ff 25 80 95 04 08       jmp    *0x8049580
8048376:       68 00 00 00 00          push   $0x0
804837b:       e9 e0 ff ff ff          jmp    8048360 <_init+0x30>
```

This entry belongs to ____

```
8048380:       ff 25 84 95 04 08       jmp    *0x8049584
8048386:       68 08 00 00 00          push   $0x8
804838b:       e9 d0 ff ff ff          jmp    8048360 <_init+0x30>
```

This entry belongs to ____

```
8048390:       ff 25 88 95 04 08       jmp    *0x8049588
8048396:       68 10 00 00 00          push   $0x10
804839b:       e9 c0 ff ff ff          jmp    8048360 <_init+0x30>
```

This entry belongs to ____

```
80483a0:       ff 25 8c 95 04 08       jmp    *0x804958c
80483a6:       68 18 00 00 00          push   $0x18
80483ab:       e9 b0 ff ff ff          jmp    8048360 <_init+0x30>
```

This entry belongs to ____

```
80483b0:       ff 25 90 95 04 08       jmp    *0x8049590
80483b6:       68 20 00 00 00          push   $0x20
80483bb:       e9 a0 ff ff ff          jmp    8048360 <_init+0x30>
```

This entry belongs to ____

```
80483c0:       ff 25 94 95 04 08       jmp    *0x8049594
80483c6:       68 28 00 00 00          push   $0x28
80483cb:       e9 90 ff ff ff          jmp    8048360 <_init+0x30>
```

This entry belongs to ____

```
80483d0:       ff 25 98 95 04 08       jmp    *0x8049598
80483d6:       68 30 00 00 00          push   $0x30
80483db:       e9 80 ff ff ff          jmp    8048360 <_init+0x30>
```

```
0x8049574 <_GLOBAL_OFFSET_TABLE_>:      0x080495a0  ____________________
0x8049578 <_GLOBAL_OFFSET_TABLE_+4>:    0x00000000  ____________________
0x804957c <_GLOBAL_OFFSET_TABLE_+8>:    0x00000000  ____________________
0x8049580 <_GLOBAL_OFFSET_TABLE_+12>:   0x08048376  ____________________
0x8049584 <_GLOBAL_OFFSET_TABLE_+16>:   0x08048386  ____________________
0x8049588 <_GLOBAL_OFFSET_TABLE_+20>:   0x08048396  ____________________
0x804958c <_GLOBAL_OFFSET_TABLE_+24>:   0x080483a6  ____________________
0x8049590 <_GLOBAL_OFFSET_TABLE_+28>:   0x080483b6  ____________________
0x8049594 <_GLOBAL_OFFSET_TABLE_+32>:   0x080483c6  ____________________
0x8049598 <_GLOBAL_OFFSET_TABLE_+36>:   0x080483d6  ____________________
```

B. gdb dump of .got at ______________________________

```
0x8049574 <_GLOBAL_OFFSET_TABLE_>:      0x080495a0
0x8049578 <_GLOBAL_OFFSET_TABLE_+4>:    0x40013ed0
0x804957c <_GLOBAL_OFFSET_TABLE_+8>:    0x4000a960
0x8049580 <_GLOBAL_OFFSET_TABLE_+12>:   0x08048376
0x8049584 <_GLOBAL_OFFSET_TABLE_+16>:   0x08048386
0x8049588 <_GLOBAL_OFFSET_TABLE_+20>:   0x08048396
0x804958c <_GLOBAL_OFFSET_TABLE_+24>:   0x080483a6
0x8049590 <_GLOBAL_OFFSET_TABLE_+28>:   0x080483b6
0x8049594 <_GLOBAL_OFFSET_TABLE_+32>:   0x080483c6
0x8049598 <_GLOBAL_OFFSET_TABLE_+36>:   0x080483d6
```

C. gdb dump of .got at ______________________________

```
0x8049574 <_GLOBAL_OFFSET_TABLE_>:      0x080495a0
0x8049578 <_GLOBAL_OFFSET_TABLE_+4>:    0x40013ed0
0x804957c <_GLOBAL_OFFSET_TABLE_+8>:    0x4000a960
0x8049580 <_GLOBAL_OFFSET_TABLE_+12>:   0x400fa530
0x8049584 <_GLOBAL_OFFSET_TABLE_+16>:   0x08048386
0x8049588 <_GLOBAL_OFFSET_TABLE_+20>:   0x400734e0
0x804958c <_GLOBAL_OFFSET_TABLE_+24>:   0x080483a6
0x8049590 <_GLOBAL_OFFSET_TABLE_+28>:   0x400328cc
0x8049594 <_GLOBAL_OFFSET_TABLE_+32>:   0x080483c6
0x8049598 <_GLOBAL_OFFSET_TABLE_+36>:   0x40068080
```

Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Spring 2003

Exam 2

April 10, 2003

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 77 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. You may use any books or notes you like. You may not use a calculator, laptop or other wireless device. Good luck!

1 (8):	
2 (10):	
3 (9):	
4 (20):	
5 (8):	
6 (6):	
7 (8):	
8 (8):	
TOTAL (77):	

Problem 1. (8 points):

Here's a familiar function that returns the nth fibonacci number:

```
int fibo(int n) {
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    if (n == 2)
        return 1;
    return fibo(n-1) + fibo(n-2);
}
```

This implementation is way too slow. Which of the following are the **two main** contributing factors? (Circle the correct answer)

(a) There are two recursive calls per iteration.

(b) There are too many if checks before the actual recursion.

(c) The overhead of a recursion (calling functions) is too high as compared to the actual work done.

(d) The recursive calls only goes down by 1 or 2 each round. (i.e. $n - 1$ and $n - 2$)

You have been hired recently to create the world's fastest fibonacci number generator to be run on a **Intel Pentium III** just like our fish machines.

You wrote your first draft as shown below. Assume n takes only natural numbers.

```
int fibo2(int n) {
    int x = 0;
    int y = 1;
    int tmp, i;
    if (n < 2)
        return n;

    for (i = 2; i <= n; ++i) {
        tmp = y;
        y = x + y;
        x = tmp;
    }
    return y;
}
```

This program gave a terrific improvement over the recursive implementation. However, unsatisfied with the result, you seek to further improve the program. Consider the following two alternatives, `fibo3` and `fibo4`. Assume n to be very large (e.g. > 10000).

```
int fibo3(int n) {
    int x = 0;
    int y = 1;
    int i;
    if (n < 2)
        return n;

    for (i = 2; i <= n; i+=2) {
        x = x + y;
        y = x + y;
    }
    if (i == n+1)
        return y;
    return x;
}
```

Is `fibo3` faster than `fibo2`? **Yes** **No**

Why? Please give short answers (one sentence and/or indicative keywords will suffice)

Here’s another one:

```
int fibo4(int n) {
    int p = 0;    int q = 1;    int r = 1;    int s = 2;
    int k = 3;    int l = 5;    int m = 8;    int o = 13;

    int i;
    if (n < 2)  return n;
    if (n == 2) return r;
    if (n == 3) return s;
    if (n == 4) return k;
    if (n == 5) return l;
    if (n == 6) return m;
    if (n == 7) return o;

    for (i = 8; i <= n; i+=8) {
        p = m + o;
        q = o + p;
        r = p + q;
        s = q + r;
        k = r + s;
        l = s + k;
        m = k + l;
        o = l + m;
    }

    if (i == n+1) return o;
    if (i == n+2) return m;
    if (i == n+3) return l;
    if (i == n+4) return k;
    if (i == n+5) return s;
    if (i == n+6) return r;
    if (i == n+7) return q;
    return p;
}
```

Is `fibo4` faster than `fibo3`? **Yes** **No**

Why? Please give short answers (one sentence and/or indicative keywords will suffice)

Problem 2. (10 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 *Index* column contains the set index for each set of 4 lines. The *Tag* columns contain the tag value for each line. The *V* column contains the valid bit for each line. The *Bytes 0–1* columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

The contents of the cache are as follows:

4-way Set Associative Cache												
Index	Tag	V	Bytes 0–1	Tag	V	Bytes 0–1	Tag	V	Bytes 0–1	Tag	V	Bytes 0–1
0	30	1	4E 47	4B	0	1A D6	77	1	5A B3	EA	0	OD C3
1	09	1	F8 88	AF	1	CA 4A	6C	0	8B 58	47	1	3A 17
2	80	1	4B 59	3B	0	84 0D	A6	1	B4 5B	EE	1	FF 75
3	C4	1	77 CF	77	0	61 DC	3A	1	6B D5	C3	0	1A 9F
4	0D	0	87 2A	66	1	CE 64	7D	1	4E AF	D6	0	89 29
5	E3	1	30 E8	3F	1	1E E8	B4	0	E2 5F	84	1	59 C0
6	60	0	93 2B	35	0	56 46	D4	1	64 CD	FE	0	CA 98
7	A7	1	B2 9B	1B	0	1F 0E	35	1	9E 44	08	0	04 12

Part 1

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

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

Part 2

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: `EE4`

A. Physical address format (one bit per box)

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

B. Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Physical address: `B4A`

A. Physical address format (one bit per box)

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

B. Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Problem 3. (9 points):

This problem tests your understanding of memory bugs. Each of the code sequences below may or may not contain memory bugs. The code all compiles without warnings or errors. If you think there is a bug, please circle **YES** and indicate the type of bug from the list below of memory bugs. Otherwise, if you think there are no memory bugs in the code, please circle **NO**.

Bugs:

1. buffer overflow error
2. memory leak
3. dereference of possibly bad pointer
4. incorrect use of free
5. incorrect use of realloc
6. misaligned access to memory
7. Other memory bug

A.
```
typedef struct _stackelem* StackElem;

struct _stackelem {
        void* ptr;
        StackElem next;
};

typedef struct _stack* Stack;
struct _stack {
        StackElem top;
};

void* popptr(Stack s)
{
        void* ret;
        if (s == NULL) return NULL;
        if (s->top != NULL) {
                ret = s->top->ptr;
                s->top = s->top->next;
                return ret;
        }
        return NULL;
}
```

NO **YES** Type of bug(s): ____________________

B.
```
void readints(int* count, int* vals)
{
        int i;

        scanf("%d\n", count);
        vals = malloc(*count * sizeof(int));
        for (i = 0; i<*count; i++) {
                scanf("%d\n", vals+i);
        }
}

void caller(void)
{
        int count;
        int* vals;

        /* some stuff */
        readints(&count, vals);
        /* some more stuff */

        /* all done with inputs */
        free(vals);
}
```

NO **YES** Type of bug(s): ____________________

C.
```
struct IntList {
        int val;
        struct IntList* next;
};

struct IntList*
addToList(struct IntList* old, int v)
{
        struct IntList* newone;

        newone = malloc(sizeof(struct IntList));
        newone->val = v;
        newone->next = old;
        return newone;
}
```

NO **YES** Type of bug(s): ____________________

Problem 4. (20 points):

Imagine that you are implementing a console driver. The job of a console driver is to take characters and output them to the screen. The contents of the console is controlled by a region of main memory (memory mapped I/O). Each character on the console is represented in this region by a pair of integers. The first number in this pair is simply the character itself. The second integer controls the foreground and background colors used to draw the character. These integer pairs are stored in row major order. For your console, there will be 25 rows of 80 characters each.

We define the following structure:

```
struct consoleChar
{
  int character;
  int color;
}

struct consoleChar buffer[25][80];
```

To simplify the reasoning about this problem, you can assume the following:

- `sizeof(int) = 4`
- Buffer begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array buffer. All variables are stored in registers.
- All code is compiled with -O0 flag (no optimizations).

Note: Be careful about the number of accesses per loop.

An obvious thing that we might to do is to clear the console - this requires writing a black colored ’space’ to all positions of the console. Consider the following algorithm to clear the console:

```
for(row=0; row<25; row++)
  for(col=0; col<80; col++)
  {
    buffer[row][col].character = ’ ’;
    buffer[row][col].color = 0x00;
  }
```

Assume your cache is 1024-byte direct-mapped data cache with 64-byte lines. What is the cache miss rate?

Assume your cache is 1024-byte 4-way set associative using an LRU replacement policy with 32-byte lines. What is the cache miss rate?

Now, we change the algorithm a little - instead of clearing the screen by rows, we will do so by columns. Thus we change the algorithm to be the following:

```
for(col=0; col<80; col++)
  for(row=0; row<25; row++)
  {
    buffer[row][col].character = ’ ’;
    buffer[row][col].color = 0x00;
  }
```

Assume your cache is 512-byte direct-mapped data cache with 64-byte lines. What is the cache miss rate?

Assume your cache is 1024-byte 2-way set associative using an LRU replacement policy with 64-byte lines. What is the cache miss rate?

Another thing we might want is to count how many non-blank characters we have. Consider the following algorithm:

```
for(row=0; row<25; row++)
  for(col=0; col<80; col++)
  {
    cChar = buffer[row][col].character;
    cColor = buffer[row][col].color;
    if (cChar != ’ ’ || cColor != 0x00)
      nonBlankCounter++;
  }
```

Assume your cache is 128-byte direct-mapped data cache with 16-byte lines. What is the cache miss rate?

Now we add another feature to our driver - scrolling. We implement it via the following algorithm:

```
for(row = 0; row < 24; row++)
  for(col = 0; col < 80; col++)
  {
    buffer[row][col].character = buffer[row+1][col].character;
    buffer[row][col].color = buffer[row+1][col].color;
  }
```

Assume your cache is 640-byte direct-mapped data cache with 640-byte lines. What is the cache miss rate?

Assume your cache is 1280-byte 2-way set associative using an LRU replacement policy with 640-byte lines. What is the cache miss rate?

Problem 5. (8 points):

The following problem concerns various aspects of virtual memory.

Part I.

For this part only, the following are attributes of the machine that you will need to consider:

- Memory is byte addressable
- Virtual Addresses are 24 bits wide
- Physical Addresses are 16 bits wide
- Pages are 1KB
- Each Page Table Entry contains:
  - Physical Page Number
  - Valid Bit, Read Only bit (1 for Read Only, 0 for Read/Write)

A. The box below shows the format of a virtual address. Indicate the bits used for the VPN (Virtual Page Number) and VPO (Virtual Page Offset).

			20				16				12				8				4				0

B. The box below shows the format for a physical address. Indicate the bits used for the PPN (Physical Page Number) and PPO (Physical Page Offset)

			12				8				4				0

C. **Note:** For the questions below, answers of the form 2^i are acceptable. Also, please note the units of each answer

How much *virtual* memory is addressable?

______________________ bytes

How much *physical* memory is addressable?

______________________ bytes

How many bits is each Page Table Entry?

______________________ bits

How large is the Page Table?

______________________ bytes

Part II

The Xbox video game console comes configured with a Pentium III 733MHz processor with 64MB of RAM. Games running on the Xbox console do not make use of a virtual memory subsystem, because they are performance sensitive. Give two aspects of a virtual memory subsystem that would cause the designers to forego using virtual memory at all. **Note:** Short, simple answers are fine, but be clear as to how performance suffers.

Problem 6. (6 points):

This problem tests your understanding of Unix process control.

Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.) Assume that printf is unbuffered.

```
int main()
{
    int i = 1;

    printf("%d",i);

    if(fork() == 0)
    {
        i++;
        printf("%d",i);
        i++;
    }
    else
    {
        i+=3;
        printf("%d",i);
        wait(NULL);
    }

    printf("%d",i);
}
```

List all possible outputs of this program.

Problem 7. (8 points):

This question tests your understanding of signals and signal handlers.
Assume that all functions and procedures return correctly. If the `pid` argument of `kill` equals 0, then the `sig` argument is sent to every process in the process group of the current process. Assume that printf is unbuffered.

```
void handler(int sig)
{
    printf("hello"\n");
}

int main()
{
    int pid;
    setpgid(0,0);
    signal(SIGCHLD, handler);
    if((pid = fork()) == 0)
    {
        kill(0, SIGCHLD);
        printf("bonjour\n");
        exit(0);
    }
    else
    {
        waitpid(pid, NULL, 0);
    }
    printf("hola\n");
}
```

Draw an **X** through any column which does not represent a valid possible output of this program.

hello	hello	hello	hello	hello	hello
hello	hello	hello	hello	bonjour	bonjour
bonjour	bonjour	hello	hola	hola	hello
hello	hola	bonjour	bonjour		hello
hola		hola			hola

Problem 8. (8 points):

Answer the following short answer questions with **no more** than 2 sentences.

A. Where would a conservative garbage collect for C find the root set of pointers?

B. What advantage does a mark-and-sweep collector have over a reference-counting collector?

C. What could go wrong when the following two files are linked together?

```
int x;                                      static int y;
int y;                                      double x;
foo(int a) {                                bar(int b) {
        x=a;                                        x = (double)b;
        y=a*a;                                      y = b * b;
}                                           }
```

Answer the following questions by circling the correct answer:

D. When finding a free block, the segregated free-list algorithm aproximates best-fit algorithm in memory usage, but has a time complexity similar to the first-fit algorithm.

True False

E. In the call `realloc(p, 128)`, current value of `p` does not have to be the result of a previous malloc or realloc.

True False

F. A genius programmer in 213 found a way to eliminate all internal fragmentation by eliminating the need for a header and a boundary tag. The resulting malloc package achieves 100% peak memory utilization.

True False

G. Deferred coalescing often performs better than immediate coalescing, but can result in an increase in the amortized time for a free operation.

True False

H. Deferred coalescing reduces false fragmentation

True False

Andrew login ID:_______________________________________

Full Name:_______________________________________

CS 15-213, Spring 2003

Exam 2

April 10, 2003

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 77 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. You may use any books or notes you like. You may not use a calculator, laptop or other wireless device. Good luck!

1 (8):	
2 (10):	
3 (9):	
4 (20):	
5 (8):	
6 (6):	
7 (8):	
8 (8):	
TOTAL (77):	

Problem 1. (8 points):

Here's a familiar function that returns the nth fibonacci number:

```
int fibo(int n) {
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    if (n == 2)
        return 1;
    return fibo(n-1) + fibo(n-2);
}
```

This implementation is way too slow. Which of the following are the **two main** contributing factors? (Circle the correct answer)

(a) There are two recursive calls per iteration.

(b) There are too many if checks before the actual recursion.

(c) The overhead of a recursion (calling functions) is too high as compared to the actual work done.

(d) The recursive calls only goes down by 1 or 2 each round. (i.e. $n-1$ and $n-2$)

You have been hired recently to create the world's fastest fibonacci number generator to be run on a **Intel Pentium III** just like our fish machines.

You wrote your first draft as shown below. Assume n takes only natural numbers.

```
int fibo2(int n) {
    int x = 0;
    int y = 1;
    int tmp, i;
    if (n < 2)
        return n;

    for (i = 2; i <= n; ++i) {
        tmp = y;
        y = x + y;
        x = tmp;
    }
    return y;
}
```

This program gave a terrific improvement over the recursive implementation. However, unsatisfied with the result, you seek to further improve the program. Consider the following two alternatives, `fibo3` and `fibo4`. Assume n to be very large (e.g. > 10000).

```
int fibo3(int n) {
    int x = 0;
    int y = 1;
    int i;
    if (n < 2)
        return n;

    for (i = 2; i <= n; i+=2) {
        x = x + y;
        y = x + y;
    }
    if (i == n+1)
        return y;
    return x;
}
```

Is `fibo3` faster than `fibo2`? **Yes** **No**

Why? Please give short answers (one sentence and/or indicative keywords will suffice)

Here's another one:

```
int fibo4(int n) {
    int p = 0;    int q = 1;    int r = 1;    int s = 2;
    int k = 3;    int l = 5;    int m = 8;    int o = 13;

    int i;
    if (n < 2)  return n;
    if (n == 2) return r;
    if (n == 3) return s;
    if (n == 4) return k;
    if (n == 5) return l;
    if (n == 6) return m;
    if (n == 7) return o;

    for (i = 8; i <= n; i+=8) {
        p = m + o;
        q = o + p;
        r = p + q;
        s = q + r;
        k = r + s;
        l = s + k;
        m = k + l;
        o = l + m;
    }

    if (i == n+1) return o;
    if (i == n+2) return m;
    if (i == n+3) return l;
    if (i == n+4) return k;
    if (i == n+5) return s;
    if (i == n+6) return r;
    if (i == n+7) return q;
    return p;
}
```

Is `fibo4` faster than `fibo3`? **Yes** **No**

Why? Please give short answers (one sentence and/or indicative keywords will suffice)

Problem 2. (10 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 *Index* column contains the set index for each set of 4 lines. The *Tag* columns contain the tag value for each line. The *V* column contains the valid bit for each line. The *Bytes 0–1* columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

The contents of the cache are as follows:

4-way Set Associative Cache												
Index	Tag	V	Bytes 0–1	Tag	V	Bytes 0–1	Tag	V	Bytes 0–1	Tag	V	Bytes 0–1
0	30	1	4E 47	4B	0	1A D6	77	1	5A B3	EA	0	OD C3
1	09	1	F8 88	AF	1	CA 4A	6C	0	8B 58	47	1	3A 17
2	80	1	4B 59	3B	0	84 0D	A6	1	B4 5B	EE	1	FF 75
3	C4	1	77 CF	77	0	61 DC	3A	1	6B D5	C3	0	1A 9F
4	0D	0	87 2A	66	1	CE 64	7D	1	4E AF	D6	0	89 29
5	E3	1	30 E8	3F	1	1E E8	B4	0	E2 5F	84	1	59 C0
6	60	0	93 2B	35	0	56 46	D4	1	64 CD	FE	0	CA 98
7	A7	1	B2 9B	1B	0	1F 0E	35	1	9E 44	08	0	04 12

Part 1

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

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

Part 2

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: EE4

A. Physical address format (one bit per box)

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

B. Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Physical address: B4A

A. Physical address format (one bit per box)

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

B. Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Problem 3. (9 points):

This problem tests your understanding of memory bugs. Each of the code sequences below may or may not contain memory bugs. The code all compiles without warnings or errors. If you think there is a bug, please circle **YES** and indicate the type of bug from the list below of memory bugs. Otherwise, if you think there are no memory bugs in the code, please circle **NO**.

Bugs:

1. buffer overflow error
2. memory leak
3. dereference of possibly bad pointer
4. incorrect use of free
5. incorrect use of realloc
6. misaligned access to memory
7. Other memory bug

A.
```
typedef struct _stackelem* StackElem;

struct _stackelem {
        void* ptr;
        StackElem next;
};

typedef struct _stack* Stack;
struct _stack {
        StackElem top;
};

void* popptr(Stack s)
{
        void* ret;
        if (s == NULL) return NULL;
        if (s->top != NULL) {
                ret = s->top->ptr;
                s->top = s->top->next;
                return ret;
        }
        return NULL;
}
```

NO **YES** Type of bug(s): ____________________

B.
```
void readints(int* count, int* vals)
{
        int i;

        scanf("%d\n", count);
        vals = malloc(*count * sizeof(int));
        for (i = 0; i<*count; i++) {
                scanf("%d\n", vals+i);
        }
}

void caller(void)
{
        int count;
        int* vals;

        /* some stuff */
        readints(&count, vals);
        /* some more stuff */

        /* all done with inputs */
        free(vals);
}
```

NO **YES** Type of bug(s): ____________________

C.
```
struct IntList {
        int val;
        struct IntList* next;
};

struct IntList*
addToList(struct IntList* old, int v)
{
        struct IntList* newone;

        newone = malloc(sizeof(struct IntList));
        newone->val = v;
        newone->next = old;
        return newone;
}
```

NO **YES** Type of bug(s): ____________________

Problem 4. (20 points):

Imagine that you are implementing a console driver. The job of a console driver is to take characters and output them to the screen. The contents of the console is controlled by a region of main memory (memory mapped I/O). Each character on the console is represented in this region by a pair of integers. The first number in this pair is simply the character itself. The second integer controls the foreground and background colors used to draw the character. These integer pairs are stored in row major order. For your console, there will be 25 rows of 80 characters each.

We define the following structure:

```
struct consoleChar
{
  int character;
  int color;
}

struct consoleChar buffer[25][80];
```

To simplify the reasoning about this problem, you can assume the following:

- `sizeof(int) = 4`
- Buffer begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array buffer. All variables are stored in registers.
- All code is compiled with -O0 flag (no optimizations).

Note: Be careful about the number of accesses per loop.

An obvious thing that we might to do is to clear the console - this requires writing a black colored ’space’ to all positions of the console. Consider the following algorithm to clear the console:

```
for(row=0; row<25; row++)
  for(col=0; col<80; col++)
  {
    buffer[row][col].character = ’ ’;
    buffer[row][col].color = 0x00;
  }
```

Assume your cache is 1024-byte direct-mapped data cache with 64-byte lines. What is the cache miss rate?

Assume your cache is 1024-byte 4-way set associative using an LRU replacement policy with 32-byte lines. What is the cache miss rate?

Now, we change the algorithm a little - instead of clearing the screen by rows, we will do so by columns. Thus we change the algorithm to be the following:

```
for(col=0; col<80; col++)
  for(row=0; row<25; row++)
  {
    buffer[row][col].character = ’ ’;
    buffer[row][col].color = 0x00;
  }
```

Assume your cache is 512-byte direct-mapped data cache with 64-byte lines. What is the cache miss rate?

Assume your cache is 1024-byte 2-way set associative using an LRU replacement policy with 64-byte lines. What is the cache miss rate?

Another thing we might want is to count how many non-blank characters we have. Consider the following algorithm:

```
for(row=0; row<25; row++)
  for(col=0; col<80; col++)
  {
    cChar = buffer[row][col].character;
    cColor = buffer[row][col].color;
    if (cChar != ’ ’ || cColor != 0x00)
      nonBlankCounter++;
  }
```

Assume your cache is 128-byte direct-mapped data cache with 16-byte lines. What is the cache miss rate?

Now we add another feature to our driver - scrolling. We implement it via the following algorithm:

```
for(row = 0; row < 24; row++)
  for(col = 0; col < 80; col++)
  {
    buffer[row][col].character = buffer[row+1][col].character;
    buffer[row][col].color = buffer[row+1][col].color;
  }
```

Assume your cache is 640-byte direct-mapped data cache with 640-byte lines. What is the cache miss rate?

Assume your cache is 1280-byte 2-way set associative using an LRU replacement policy with 640-byte lines. What is the cache miss rate?

Problem 5. (8 points):

The following problem concerns various aspects of virtual memory.

Part I.

For this part only, the following are attributes of the machine that you will need to consider:

- Memory is byte addressable
- Virtual Addresses are 24 bits wide
- Physical Addresses are 16 bits wide
- Pages are 1KB
- Each Page Table Entry contains:
  - Physical Page Number
  - Valid Bit, Read Only bit (1 for Read Only, 0 for Read/Write)

A. The box below shows the format of a virtual address. Indicate the bits used for the VPN (Virtual Page Number) and VPO (Virtual Page Offset).

			20				16				12				8				4				0

B. The box below shows the format for a physical address. Indicate the bits used for the PPN (Physical Page Number) and PPO (Physical Page Offset)

			12				8				4				0

C. **Note:** For the questions below, answers of the form 2^i are acceptable. Also, please note the units of each answer

How much *virtual* memory is addressable?

______________________ bytes

How much *physical* memory is addressable?

______________________ bytes

How many bits is each Page Table Entry?

______________________ bits

How large is the Page Table?

______________________ bytes

Part II

The Xbox video game console comes configured with a Pentium III 733MHz processor with 64MB of RAM. Games running on the Xbox console do not make use of a virtual memory subsystem, because they are performance sensitive. Give two aspects of a virtual memory subsystem that would cause the designers to forego using virtual memory at all. **Note:** Short, simple answers are fine, but be clear as to how performance suffers.

Problem 6. (6 points):

This problem tests your understanding of Unix process control.
Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.) Assume that printf is unbuffered.

```
int main()
{
    int i = 1;

    printf("%d",i);

    if(fork() == 0)
    {
        i++;
        printf("%d",i);
        i++;
    }
    else
    {
        i+=3;
        printf("%d",i);
        wait(NULL);
    }

    printf("%d",i);
}
```

List all possible outputs of this program.

Problem 7. (8 points):

This question tests your understanding of signals and signal handlers.
Assume that all functions and procedures return correctly. If the `pid` argument of `kill` equals 0, then the `sig` argument is sent to every process in the process group of the current process. Assume that printf is unbuffered.

```
void handler(int sig)
{
    printf("hello"\n");
}

int main()
{
    int pid;
    setpgid(0,0);
    signal(SIGCHLD, handler);
    if((pid = fork()) == 0)
    {
        kill(0, SIGCHLD);
        printf("bonjour\n");
        exit(0);
    }
    else
    {
        waitpid(pid, NULL, 0);
    }
    printf("hola\n");
}
```

Draw an **X** through any column which does not represent a valid possible output of this program.

hello	hello	hello	hello	hello	hello
hello	hello	hello	hello	bonjour	bonjour
bonjour	bonjour	hello	hola	hola	hello
hello	hola	bonjour	bonjour		hello
hola		hola			hola

Problem 8. (8 points):

Answer the following short answer questions with **no more** than 2 sentences.

A. Where would a conservative garbage collect for C find the root set of pointers?

B. What advantage does a mark-and-sweep collector have over a reference-counting collector?

C. What could go wrong when the following two files are linked together?

```
int x;                                      static int y;
int y;                                      double x;
foo(int a) {                                bar(int b) {
        x=a;                                        x = (double)b;
        y=a*a;                                      y = b * b;
}                                           }
```

Answer the following questions by circling the correct answer:

D. When finding a free block, the segregated free-list algorithm aproximates best-fit algorithm in memory usage, but has a time complexity similar to the first-fit algorithm.

True False

E. In the call `realloc(p, 128)`, current value of `p` does not have to be the result of a previous malloc or realloc.

True False

F. A genius programmer in 213 found a way to eliminate all internal fragmentation by eliminating the need for a header and a boundary tag. The resulting malloc package achieves 100% peak memory utilization.

True False

G. Deferred coalescing often performs better than immediate coalescing, but can result in an increase in the amortized time for a free operation.

True False

H. Deferred coalescing reduces false fragmentation

True False

Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Spring 2004

Exam 2

April 8, 2004

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 79 points and a total of **17** pages.
- 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. You may use any books or notes you like. You may not use a calculator, laptop or other wireless device. Good luck!

1 (16):	
2 (10):	
3 (8):	
4 (12):	
5 (9):	
6 (10):	
7 (14):	
TOTAL (79):	

Problem 1. (16 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 13 bits wide.
- The cache is 4-way set associative, with a 4-byte block size and 32 total lines.

In the following tables, **all numbers are given in hexadecimal**. The *Index* column contains the set index for each set of 4 lines. The *Tag* columns contain the tag value for each line. The *V* column contains the valid bit for each line. The *Bytes 0–3* columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

The contents of the cache are as follows:

4-way Set Associative Cache												
Index	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3
0	84	1	ED 32 0A A2	9E	0	BF 80 1D FC	10	0	EF 09 86 2A	E8	0	25 44 6F 1A
1	18	1	03 3E CD 38	E4	0	16 7B ED 5A	02	0	8E 4C DF 18	E4	1	FB B7 12 02
2	84	0	54 9E 1E FA	84	1	DC 81 B2 14	48	0	B6 1F 7B 44	89	1	10 F5 B8 2E
3	92	0	2F 7E 3D A8	9F	0	27 95 A4 74	57	1	07 11 FF D8	93	1	C7 B7 AF C2
4	84	1	32 21 1C 2C	FA	1	22 C2 DC 34	73	0	BA DD 37 D8	28	1	E7 A2 39 BA
5	A7	1	A9 76 2B EE	73	0	BC 91 D5 92	28	1	80 BA 9B F6	6B	0	48 16 81 0A
6	8B	1	5D 4D F7 DA	29	1	69 C2 8C 74	B5	1	A8 CE 7F DA	BF	0	FA 93 EB 48
7	84	1	04 2A 32 6A	96	0	B1 86 56 0E	CC	0	96 30 47 F2	91	1	F8 1D 42 30

Part 1

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:

O The block **o**ffset within the cache line
I The cache **i**ndex
T The cache **t**ag

12	11	10	9	8	7	6	5	4	3	2	1	0

Part 2

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: `0x0D74`

Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0

Physical memory reference

Parameter	Value
Cache Offset (CO)	0x______________
Cache Index (CI)	0x______________
Cache Tag (CT)	0x______________
Cache Hit? (Y/N)	______________
Cache Byte returned	0x______________

Physical address: `0x0AEE`

Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0

Physical memory reference

Parameter	Value
Cache Offset (CO)	0x______________
Cache Index (CI)	0x______________
Cache Tag (CT)	0x______________
Cache Hit? (Y/N)	______________
Cache Byte returned	0x______________

Part 3

For the given contents of the cache, list all of the hex physical memory addresses that will hit in Set 7. To save space, you should express contiguous addresses as a range. For example, you would write the four addresses `0x1314`, `0x1315`, `0x1316`, `0x1317` as `0x1314--0x1317`.

Answer: ____

The following templates are provided as scratch space:

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

Part 4

For the given contents of the cache, what is the probability (expressed as a percentage) of a cache hit when the physical memory address ranges between `0x1080 - 0x109F`. Assume that all addresses are equally likely to be referenced.

Probability = ______________%

The following templates are provided as scratch space:

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

Problem 2. (10 points):

This problem requires you to analyze the behavior of the program below which transposes the $N \times N$ int-matrix A.For this problem, $N = 4$. For this problem, you should assume that the loop variables **x** and **y** are kept in registers and do not cause memory accesses. Likewise, the temporary variable **t** which is used to exchange two array elements is also stored in a register and does not cause any load/store from/to the memory system or the caches.

```
  #define N 4  /* Array size */
...
  int A[N][N] = {0};
...
 {   int x, y;
...
     for (y = 0; y < N; y++) {
       for (x = y + 1; x < N; x++) {
         int t;
         t = A[y][x];
         A[y][x] = A[x][y];
         A[x][y] = t;
       }
    }
...
 }
```

You are supposed to analyze how this program will interact with a simple cache. The cache line size is **2*sizeof(int)**. The cache is cold when the program starts. Further more, the array A is aligned so that the first two elements are stored in the same cache line.

You are supposed to fill out the tables below. For each load (ld) and store (st) operation to an element of the array A, you should indicate if this operation misses (**M**) or hits (**H**) in the cache. Note that this program does not touch the diagonal array elements.

1. The cache is direct mapped and has two (2) lines:

	ld= st=	ld= st=	ld= st=
ld= st=		ld= st=	ld= st=
ld= st=	ld= st=		ld= st=
ld= st=	ld= st=	ld= st=	

2. The cache is 2-way set-associative and has one set. It uses the least recently used (LRU) replacement policy:

	ld= st=	ld= st=	ld= st=
ld= st=		ld= st=	ld= st=
ld= st=	ld= st=		ld= st=
ld= st=	ld= st=	ld= st=	

Problem 3. (8 points):

Consider the following C programs. (For space reasons, we are not checking error return codes, so assume that all functions return normally.) Assume that `printf` is unbuffered and that each call to `printf` executes atomically.

```
/* ecf.c */
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

#define MAXLEN 32

int main(int argc, char *argv[])
{
   int i, status, limit;
   char num[MAXLEN+1];
   pid_t pid;

   limit = atoi(argv[1]);

   for(i = 0; i < limit; i++) {
      if((pid = fork()) == 0) {
         snprintf(num, MAXLEN, "%d", i+10);
         execl("./kid", "./kid", num, NULL);
      }
      if(i == (limit - 2)) {
         waitpid(pid, &status, 0);
         printf("%d\n", status);
      }
   }

   return 0;
}

-------------------------------------------
/* kid.c */
#include <stdio.h>

int main(int argc, char *argv[])
{
   printf("%d\n", atoi(argv[1]));
   return 0;
}
```

Assume that ecf.c is compiled into ecf and kid.c is compiled into kid in the same directory. List **all** possible outputs (note the newlines) of the following command:

```
[user@host directory]$ ./ecf 3
```

Problem 4. (12 points):

This problem concerns the way virtual addresses are translated into physical addresses. Imagine a system has the following parameters:

- Virtual addresses are 18 bits wide.
- Physical addresses are 16 bits wide.
- The page size is 1024 bytes.
- The TLB is 4-way set associative with 16 total entries.

The contents of the TLB and the first 32 entries of the page table are shown as follows. **All numbers are given in hexadecimal**.

TLB			
Index	Tag	PPN	Valid
0	05	13	0
	1E	14	1
	10	0F	1
	0F	1E	0
1	1F	01	1
	11	1F	0
	03	2B	1
	1D	23	0
2	06	08	1
	0F	19	1
	0A	09	1
	1F	20	1
3	03	13	0
	13	12	1
	0C	0B	0
	2E	24	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	2D	0
05	13	1	15	1B	0
06	0F	1	16	0C	0
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	01	0	1A	0C	1
0B	16	1	1B	12	1
0C	01	1	1C	1E	0
0D	15	1	1D	0E	1
0E	0C	0	1E	27	1
0F	14	0	1F	18	1

Part 1

1. The diagram below shows the format of a virtual address. Please indicate the following fields by labeling the diagram: (If a field does not exist, do not draw it on the diagram.)

O	The virtual page **o**ffset
N	The virtual page **n**umber
I	The TLB **i**ndex
T	The TLB **t**ag

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. The diagram below shows the format of a physical address. Please indicate the following fields by labeling the diagram: (If a field does not exist, do not draw it on the diagram.)

O	The physical page **o**ffset
N	The physical page **n**umber

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

Part 2

For the given virtual addresses, please indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs. If there is a page fault, enter "-" for "PPN" and leave the physical address blank.

Virtual address: `0x0718F`

1. Virtual address (one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. Address translation

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	PPN	0x

3. Physical address(one bit per box)

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

Virtual address: `0x04AA4`

1. Virtual address (one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. Address translation

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	PPN	0x

3. Physical address(one bit per box)

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

Problem 5. (9 points):

This problem tests your understanding of Unix signals.
Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.) Assume that printf is unbuffered, and that latency in receiving signals is negligable.

```
/* signal.c */

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

sigset_t s1;
sigset_t s2;
int i;

void handler1(int sig);
void handler2(int sig);
void func0();
void func1();
void func2();

int main(int argc, char** argv)
{
    int n = atoi(argv[1]);

    signal(SIGINT, handler1);
    signal(SIGTSTP, handler2);

    sigemptyset(&s1);
    sigaddset(&s1, SIGINT);
    sigemptyset(&s2);
    sigaddset(&s2, SIGTSTP);

    if(n == 0)
        func0();
    else if(n == 1)
        func1();
    else if(n == 2)
        func2();

    return 0;
}
```

```
/* signal.c (continued) */

void handler1(int sig)
{
    int j;
    printf("hello\n");
    for(j = 0; j < 3; j++)
        kill(0, SIGTSTP);
}

void handler2(int sig)
{
    printf("greetings\n");
}

void func0()
{
    sigprocmask(SIG_BLOCK, &s1, NULL);
    for(i = 0; i < 5; i++)
        kill(0, SIGINT);
    sigprocmask(SIG_UNBLOCK, &s1, NULL);
}

void func1()
{
    sigprocmask(SIG_BLOCK, &s2, NULL);
    for(i = 0; i < 5; i++)
        kill(0, SIGINT);
    sigprocmask(SIG_UNBLOCK, &s2, NULL);
}

void func2()
{
    for(i = 0; i < 5; i++)
    {
        sigprocmask(SIG_BLOCK, &s1, NULL);
        kill(0, SIGINT);
        sigprocmask(SIG_BLOCK, &s2, NULL);
        sigprocmask(SIG_UNBLOCK, &s1, NULL);
        sigprocmask(SIG_UNBLOCK, &s2, NULL);
    }
}
```

For each commandline listed below, write the output which would result:

```
unix> ./signal 0
```

```
unix> ./signal 1
```

```
unix> ./signal 2
```

Problem 6. (10 points):

Suppose the file `foo.txt` contains the text ”`123456`”, `bar.txt` contains the text ”`abcdef`”, and `baz.txt` does not yet exist. Examine the following C code, and answer the questions below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main() {
    int fd1, fd2, fd3, fd4, fd5, fd6;
    int status;
    pid_t pid;
    char c;

    /* foo.txt has "123456" */
    fd1 = open("foo.txt", O_RDONLY, 0);
    fd2 = open("foo.txt", O_RDONLY, 0);

    /* bar.txt has "abcdef" */
    fd3 = open("bar.txt", O_RDWR, 0);
    fd4 = open("bar.txt", O_RDWR, 0);

    /* baz.txt doesn’t exist initially */
    fd5 = open("baz.txt", O_WRONLY | O_CREAT | O_TRUNC,
                          S_IRUSR | S_IWUSR); /* r/w */

    fd6 = dup(STDOUT_FILENO);
    dup2(fd5, STDOUT_FILENO);

    if ((pid = fork()) == 0) {
        dup2(fd3, fd2);

        read(fd3, &c, 1); printf("%c", c);
        write(fd4, "!@#$%ˆ", 6);
        read(fd3, &c, 1); printf("%c", c);
        read(fd1, &c, 1); printf("%c", c);
        read(fd2, &c, 1); printf("%c\n", c);
        exit(0);
    }

    wait(NULL);
    read(fd1, &c, 1); printf("%c", c);
    fflush(stdout);

    dup2(fd6, STDOUT_FILENO);
    printf("done.\n");
    return 0;
}
```

A. What will the contents of `baz.txt` be after the program completes?

B. What will be printed on `stdout`?

Problem 7. (14 points):

Answer the following short answer questions with **no more** than 2 sentences.

A. What characteristics of a disk subsystem determine the time it takes to access a sector on disk?

B. What is the CPU-Memory gap? And, is it getting bigger or smaller over time?

C. Does the declaration “`static int x;`” generate an entry in the .o file? If so, what kind of entry would it be?

D. List up to two positive reasons and two negative reasons for using a mark and sweep garbage collection system instead of an explicit memory allocator in a real-time system?

E. The following two files are linked together and the program is run.

```
main()
{
   int x;

   x  = 40;
   foo();
   printf("A: %d\n", x);
}

extern int x;

bar()
{
   x = 2 * x;
   foobar();
}

int x = 2;
```

```
static int x = 5;

foo()
{
   x = x + 10;
   bar();
}

foobar()
{
   extern int x;

   x+=3;
   printf("B: %d\n", x);
}
```

What is the output:

F. What is printed out in the following code:

```
#include <setjmp.h>

jmp_buf buf;
int x = 0;

main()
{
   switch (setjmp(buf))
   {
   case 0: x = 2;
           foo();
           x++;
   case 1: bar();
           x++;
   case 2: printf("A: %d\n", x);
   }
}
```

```
foo()
{
   x++;
   if (x < 10) foo();
   longjmp(buf, 1);
   x++;
   printf("B: %d\n", x);
}

bar()
{
   x++;
   longjmp(buf, 2);
   printf("C: %d\n", x);
}
```

Output:

Andrew login ID:_______________________________________

Full Name:_______________________________________

CS 15-213, Spring 2004

Exam 2

April 8, 2004

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 79 points and a total of **17** pages.
- 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. You may use any books or notes you like. You may not use a calculator, laptop or other wireless device. Good luck!

1 (16):	
2 (10):	
3 (8):	
4 (12):	
5 (9):	
6 (10):	
7 (14):	
TOTAL (79):	

Problem 1. (16 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 13 bits wide.
- The cache is 4-way set associative, with a 4-byte block size and 32 total lines.

In the following tables, **all numbers are given in hexadecimal**. The *Index* column contains the set index for each set of 4 lines. The *Tag* columns contain the tag value for each line. The *V* column contains the valid bit for each line. The *Bytes 0–3* columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

The contents of the cache are as follows:

4-way Set Associative Cache												
Index	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3
0	84	1	ED 32 0A A2	9E	0	BF 80 1D FC	10	0	EF 09 86 2A	E8	0	25 44 6F 1A
1	18	1	03 3E CD 38	E4	0	16 7B ED 5A	02	0	8E 4C DF 18	E4	1	FB B7 12 02
2	84	0	54 9E 1E FA	84	1	DC 81 B2 14	48	0	B6 1F 7B 44	89	1	10 F5 B8 2E
3	92	0	2F 7E 3D A8	9F	0	27 95 A4 74	57	1	07 11 FF D8	93	1	C7 B7 AF C2
4	84	1	32 21 1C 2C	FA	1	22 C2 DC 34	73	0	BA DD 37 D8	28	1	E7 A2 39 BA
5	A7	1	A9 76 2B EE	73	0	BC 91 D5 92	28	1	80 BA 9B F6	6B	0	48 16 81 0A
6	8B	1	5D 4D F7 DA	29	1	69 C2 8C 74	B5	1	A8 CE 7F DA	BF	0	FA 93 EB 48
7	84	1	04 2A 32 6A	96	0	B1 86 56 0E	CC	0	96 30 47 F2	91	1	F8 1D 42 30

Part 1

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:

O The block **o**ffset within the cache line
I The cache **i**ndex
T The cache **t**ag

12	11	10	9	8	7	6	5	4	3	2	1	0
T	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**I**	**I**	**I**	**O**	**O**

Part 2

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: `0x0D74`

Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0
0	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**0**

Physical memory reference

Parameter	Value
Cache Offset (CO)	0x**00**
Cache Index (CI)	0x**05**
Cache Tag (CT)	0x**6B**
Cache Hit? (Y/N)	**N (dirty)**
Cache Byte returned	0x**-**

Physical address: `0x0AEE`

Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0
0	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**0**

Physical memory reference

Parameter	Value
Cache Offset (CO)	0x**02**
Cache Index (CI)	0x**03**
Cache Tag (CT)	0x**57**
Cache Hit? (Y/N)	**Y**
Cache Byte returned	0x**FF**

Part 3

For the given contents of the cache, list all of the hex physical memory addresses that will hit in Set 7. To save space, you should express contiguous addresses as a range. For example, you would write the four addresses `0x1314`, `0x1315`, `0x1316`, `0x1317` as `0x1314--0x1317`.

Answer: **0x109C – 0x109F, 0x123C – 0x123F**

The following templates are provided as scratch space:

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

Part 4

For the given contents of the cache, what is the probability (expressed as a percentage) of a cache hit when the physical memory address ranges between `0x1080 - 0x109F`. Assume that all addresses are equally likely to be referenced.

Probability = **50** %

The following templates are provided as scratch space:

12	11	10	9	8	7	6	5	4	3	2	1	0

12	11	10	9	8	7	6	5	4	3	2	1	0

Problem 2. (10 points):

This problem requires you to analyze the behavior of the program below which transposes the $N \times N$ int-matrix A.For this problem, $N = 4$. For this problem, you should assume that the loop variables **x** and **y** are kept in registers and do not cause memory accesses. Likewise, the temporary variable **t** which is used to exchange two array elements is also stored in a register and does not cause any load/store from/to the memory system or the caches.

```
  #define N 4  /* Array size */
...
  int A[N][N] = {0};
...
 {   int x, y;
...
     for (y = 0; y < N; y++) {
       for (x = y + 1; x < N; x++) {
         int t;
         t = A[y][x];
         A[y][x] = A[x][y];
         A[x][y] = t;
       }
    }
...
 }
```

You are supposed to analyze how this program will interact with a simple cache. The cache line size is **`2*sizeof(int)`**. The cache is cold when the program starts. Further more, the array A is aligned so that the first two elements are stored in the same cache line.

You are supposed to fill out the tables below. For each load (ld) and store (st) operation to an element of the array A, you should indicate if this operation misses (**M**) or hits (**H**) in the cache. Note that this program does not touch the diagonal array elements.

1. The cache is direct mapped and has two (2) lines:

	ld=**M** st=**M**	ld=**M** st=**H**	ld=**H** st=**H**
ld=**M** st=**M**		ld=**M** st=**H**	ld=**H** st=**H**
ld=**M** st=**H**	ld=**M** st=**H**		ld=**M** st=**M**
ld=**M** st=**H**	ld=**M** st=**H**	ld=**M** st=**M**	

2. The cache is 2-way set-associative and has one set. It uses the least recently used (LRU) replacement policy:

	ld=**M** st=**H**	ld=**M** st=**H**	ld=**H** st=**H**
ld=**M** st=**H**		ld=**M** st=**H**	ld=**H** st=**H**
ld=**M** st=**H**	ld=**M** st=**H**		ld=**M** st=**H**
ld=**M** st=**H**	ld=**M** st=**H**	ld=**M** st=**H**	

Problem 3. (8 points):

Consider the following C programs. (For space reasons, we are not checking error return codes, so assume that all functions return normally.) Assume that `printf` is unbuffered and that each call to `printf` executes atomically.

```
/* ecf.c */
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

#define MAXLEN 32

int main(int argc, char *argv[])
{
   int i, status, limit;
   char num[MAXLEN+1];
   pid_t pid;

   limit = atoi(argv[1]);

   for(i = 0; i < limit; i++) {
      if((pid = fork()) == 0) {
         snprintf(num, MAXLEN, "%d", i+10);
         execl("./kid", "./kid", num, NULL);
      }
      if(i == (limit - 2)) {
         waitpid(pid, &status, 0);
         printf("%d\n", status);
      }
   }

   return 0;
}

-------------------------------------------
/* kid.c */
#include <stdio.h>

int main(int argc, char *argv[])
{
   printf("%d\n", atoi(argv[1]));
   return 0;
}
```

Assume that ecf.c is compiled into ecf and kid.c is compiled into kid in the same directory. List **all** possible outputs (note the newlines) of the following command:

```
[user@host directory]$ ./ecf 3
```

```
10   11   11   11
11   10    0    0
 0    0   10   12
12   12   12   10
```

Problem 4. (12 points):

This problem concerns the way virtual addresses are translated into physical addresses. Imagine a system has the following parameters:

- Virtual addresses are 18 bits wide.
- Physical addresses are 16 bits wide.
- The page size is 1024 bytes.
- The TLB is 4-way set associative with 16 total entries.

The contents of the TLB and the first 32 entries of the page table are shown as follows. **All numbers are given in hexadecimal**.

TLB			
Index	Tag	PPN	Valid
0	05	13	0
	1E	14	1
	10	0F	1
	0F	1E	0
1	1F	01	1
	11	1F	0
	03	2B	1
	1D	23	0
2	06	08	1
	0F	19	1
	0A	09	1
	1F	20	1
3	03	13	0
	13	12	1
	0C	0B	0
	2E	24	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	2D	0
05	13	1	15	1B	0
06	0F	1	16	0C	0
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	01	0	1A	0C	1
0B	16	1	1B	12	1
0C	01	1	1C	1E	0
0D	15	1	1D	0E	1
0E	0C	0	1E	27	1
0F	14	0	1F	18	1

Part 1

1. The diagram below shows the format of a virtual address. Please indicate the following fields by labeling the diagram: (If a field does not exist, do not draw it on the diagram.)

   O The virtual page **o**ffset
   N The virtual page **n**umber
   I The TLB **i**ndex
   T The TLB **t**ag

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T/N	**T/N**	**T/N**	**T/N**	**T/N**	**T/N**	**I/N**	**I/N**	**O**	**O**	**O**	**O**	**O**	**O**	**O**	**O**	**O**	**O**

2. The diagram below shows the format of a physical address. Please indicate the following fields by labeling the diagram: (If a field does not exist, do not draw it on the diagram.)

   O The physical page **o**ffset
   N The physical page **n**umber

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

Part 2

For the given virtual addresses, please indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs. If there is a page fault, enter “-” for “PPN” and leave the physical address blank.

Virtual address: `0x0718F`

1. Virtual address (one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	**1**

2. Address translation

Parameter	Value	Parameter	Value
VPN	0x**1C**	TLB Hit? (Y/N)	**No**
TLB Index	0x**0**	Page Fault? (Y/N)	**Yes**
TLB Tag	0x**07**	PPN	0x**–**

3. Physical address(one bit per box)

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

Virtual address: `0x04AA4`

1. Virtual address (one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**0**

2. Address translation

Parameter	Value	Parameter	Value
VPN	0x**12**	TLB Hit? (Y/N)	**No**
TLB Index	0x**2**	Page Fault? (Y/N)	**No**
TLB Tag	0x**04**	PPN	0x**0E**

3. Physical address(one bit per box)

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

Problem 5. (9 points):

This problem tests your understanding of Unix signals.
Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.) Assume that printf is unbuffered, and that latency in receiving signals is negligable.

```
/* signal.c */

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

sigset_t s1;
sigset_t s2;
int i;

void handler1(int sig);
void handler2(int sig);
void func0();
void func1();
void func2();

int main(int argc, char** argv)
{
    int n = atoi(argv[1]);

    signal(SIGINT, handler1);
    signal(SIGTSTP, handler2);

    sigemptyset(&s1);
    sigaddset(&s1, SIGINT);
    sigemptyset(&s2);
    sigaddset(&s2, SIGTSTP);

    if(n == 0)
        func0();
    else if(n == 1)
        func1();
    else if(n == 2)
        func2();

    return 0;
}
```

```
/* signal.c (continued) */

void handler1(int sig)
{
    int j;
    printf("hello\n");
    for(j = 0; j < 3; j++)
        kill(0, SIGTSTP);
}

void handler2(int sig)
{
    printf("greetings\n");
}

void func0()
{
    sigprocmask(SIG_BLOCK, &s1, NULL);
    for(i = 0; i < 5; i++)
        kill(0, SIGINT);
    sigprocmask(SIG_UNBLOCK, &s1, NULL);
}

void func1()
{
    sigprocmask(SIG_BLOCK, &s2, NULL);
    for(i = 0; i < 5; i++)
        kill(0, SIGINT);
    sigprocmask(SIG_UNBLOCK, &s2, NULL);
}

void func2()
{
    for(i = 0; i < 5; i++)
    {
        sigprocmask(SIG_BLOCK, &s1, NULL);
        kill(0, SIGINT);
        sigprocmask(SIG_BLOCK, &s2, NULL);
        sigprocmask(SIG_UNBLOCK, &s1, NULL);
        sigprocmask(SIG_UNBLOCK, &s2, NULL);
    }
}
```

For each commandline listed below, write the output which would result:

```
unix> ./signal 0
```

```
hello
greetings
greetings
greetings
```

```
unix> ./signal 1
```

```
hello
hello
hello
hello
hello
greetings
```

```
unix> ./signal 2
```

```
hello
greetings
hello
greetings
hello
greetings
hello
greetings
hello
greetings
```

Problem 6. (10 points):

Suppose the file `foo.txt` contains the text ”`123456`”, `bar.txt` contains the text ”`abcdef`”, and `baz.txt` does not yet exist. Examine the following C code, and answer the questions below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main() {
    int fd1, fd2, fd3, fd4, fd5, fd6;
    int status;
    pid_t pid;
    char c;

    /* foo.txt has "123456" */
    fd1 = open("foo.txt", O_RDONLY, 0);
    fd2 = open("foo.txt", O_RDONLY, 0);

    /* bar.txt has "abcdef" */
    fd3 = open("bar.txt", O_RDWR, 0);
    fd4 = open("bar.txt", O_RDWR, 0);

    /* baz.txt doesn’t exist initially */
    fd5 = open("baz.txt", O_WRONLY | O_CREAT | O_TRUNC,
                          S_IRUSR | S_IWUSR); /* r/w */

    fd6 = dup(STDOUT_FILENO);
    dup2(fd5, STDOUT_FILENO);

    if ((pid = fork()) == 0) {
        dup2(fd3, fd2);

        read(fd3, &c, 1); printf("%c", c);
        write(fd4, "!@#$%ˆ", 6);
        read(fd3, &c, 1); printf("%c", c);
        read(fd1, &c, 1); printf("%c", c);
        read(fd2, &c, 1); printf("%c\n", c);
        exit(0);
    }

    wait(NULL);
    read(fd1, &c, 1); printf("%c", c);
    fflush(stdout);

    dup2(fd6, STDOUT_FILENO);
    printf("done.\n");
    return 0;
}
```

A. What will the contents of `baz.txt` be after the program completes?

```
a@1#
2
```

B. What will be printed on `stdout`?

```
done.
```

Problem 7. (14 points):

Answer the following short answer questions with **no more** than 2 sentences.

A. What characteristics of a disk subsystem determine the time it takes to access a sector on disk?

Seek time, rotation speed, transfer time

B. What is the CPU-Memory gap? And, is it getting bigger or smaller over time?

The difference in time between the CPU clock rate and the dram access time. It is getting bigger.

C. Does the declaration "`static int x;`" generate an entry in the .o file? If so, what kind of entry would it be?

yes. A relocatable entry in the data segment.

D. List up to two positive reasons and two negative reasons for using a mark and sweep garbage collection system instead of an explicit memory allocator in a real-time system?

Postive: fast code development, fewer memory errors. Negatives: unpredictable pauses in system execution.

E. The following two files are linked together and the program is run.

```
main()
{
   int x;

   x  = 40;
   foo();
   printf("A: %d\n", x);
}

extern int x;

bar()
{
   x = 2 * x;
   foobar();
}

int x = 2;
```

```
static int x = 5;

foo()
{
   x = x + 10;
   bar();
}

foobar()
{
   extern int x;

   x+=3;
   printf("B: %d\n", x);
}
```

What is the output:

B: 18 A: 40

F. What is printed out in the following code:

```
#include <setjmp.h>

jmp_buf buf;
int x = 0;

main()
{
   switch (setjmp(buf))
   {
   case 0: x = 2;
           foo();
           x++;
   case 1: bar();
           x++;
   case 2: printf("A: %d\n", x);
   }
}
```

```
foo()
{
   x++;
   if (x < 10) foo();
   longjmp(buf, 1);
   x++;
   printf("B: %d\n", x);
}

bar()
{
   x++;
   longjmp(buf, 2);
   printf("C: %d\n", x);
}
```

Output:

A: 11

15-213 Introduction to Computer Systems

Exam 2

April 5, 2005

Name:	**Model Solution**
Andrew User ID:	fp
Recitation Section:	________________

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

Problem	**Max**	**Score**
1	**14**	
2	**18**	
3	**12**	
4	**8**	
5	**12**	
6	**11**	
Total	**75**	

1. Symbols and Linking (14 points)

Consider the following two files, `fib1.c` and `fib2.c`:

```
/* fib1.c */
#define MAXFIB 1024

int table[MAXFIB];
int fib(int n);

int main(int argc, char **argv) {
  int n;
  table[0] = 0;
  table[1] = 1;
  argc--; argv++; /* skip command name */
  while (argc > 0) {
    if (sscanf(*argv, "%d", &n) != 1 || n < 0 || n >= MAXFIB) {
      printf ("Error: %s not an int or out of range\n", *argv);
      exit (0);
    }
    printf("fib(%d) = %d\n", n, fib(n));
    argc--; argv++;
  }
}

/* fib2.c */
int* table;

int fib(int n) {
  static int num = 2;
  if (n >= num) {
    int i = num;
    while (i <= n) {
      table[i] = table[i-1] + table[i-2];
      i++;
    }
    num = i;
  }
  return table[n];
}
```

1. (8 points) Fill in the following tables by stating for each name whether it is local or global, whether it is strong or weak, and the section it occupies in that module (`.text`, `.data`, or`.bss`). Cross out any box in the table that does not apply. For example, cross out the first box in a line of the symbol is not in the symbol table, or cross out the second box in a line if the symbol is not global (and therefore neither weak nor strong).

   `fib1.c`

	Local or Global?	Strong or Weak?	Which segment?
`table`			
`fib`			
`num`			

   `fib2.c`

	Local or Global?	Strong or Weak?	Which segment?
`table`			
`fib`			
`num`			

2. (3 points) When the two files are linked together, symbols will be resolved. For each symbol below, show which module it will be defined in (write `fib1` or `fib2` or `not determined`).

	Defined in module?
`table`	
`fib`	
`num`	

3. (3 points) The code which is generated by `gcc -o fib fib1.c fib2.c` may not execute correctly. Explain succinctly why.

2. Virtual Address Translation (18 points)

We consider a virtual address system with the following parameters.

- The memory is byte addressable.
- Virtual addresses are 20 bits wide.
- Physical addresses are 16 bits wide.
- The page size is 4096 bytes.
- The TLB is 4-way set associative with 16 total entries.

In the following tables, all numbers are given in hexadecimal. The contents of the TLB and the page table for the first 16 virtual pages are as follows. If a VPN is not listed in the page table, assume it generates a page fault.

TLB

Index	Tag	PPN	Valid
0	03	B	1
	07	6	0
	28	3	1
	01	F	0
1	31	0	1
	12	3	0
	07	E	1
	0B	1	1
2	2A	A	0
	11	1	0
	1F	8	1
	07	5	1
3	07	3	1
	3F	F	0
	10	D	0
	32	0	0

Page Table

VPN	PPN	Valid
00	7	1
01	8	1
02	9	1
03	A	1
04	6	0
05	3	0
06	1	0
07	8	0
08	2	0
09	3	0
0A	1	1
0B	6	1
0C	C	1
0D	D	0
0E	E	0
0F	D	1

1. (4 points) In the four rows below, mark the bits that constitute the indicated part of the virtual address with an **X**. Leave the remaining bits of each row blank.

Virtual Page Number

	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VPN																				

Virtual Page Offset

	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VPO																				

TLB Tag

	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TLBT																				

TLB Index

	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TLBI																				

2. (7 points) For the virtual address `0x7E37C`, indicate the physical address and various results of the translation. If there is a page fault, enter "—" for the PPN and Physical Address. All answers should be given in hexadecimal.

Virtual Address (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Parameter	Value
VPN	
TLB Tag	
TLB Index	
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	
Physical Address	

3. (7 points) For the virtual address `0x16A48`, indicate the physical address and various results of the translation. If there is a page fault, enter "—" for the PPN and Physical Address. All answers should be given in hexadecimal.

Virtual Address (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Parameter	Value
VPN	
TLB Tag	
TLB Index	
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	
Physical Address	

3. Process Control (12 points)

Consider the following C program. For space reasons, we do not check return codes, so assume that all functions return normally. Also assume that `printf` is unbuffered.

```
void handler(int sig) {
  printf("H\n");
  exit(0);
}

int main() {
  pid_t pid1, pid2;

  signal(SIGUSR1,handler);

  pid1 = fork();
  if (pid1 == 0) {
    pid2 = fork();
    printf("A\n");
    if (pid2 == 0) {
      printf("B\n");
      exit(0);
    }
    printf("C\n");
    kill(pid2,SIGUSR1);
    exit(0);
  }
  if (waitpid(pid1, NULL, 0) > 0) {
    printf("D\n");
  }
  exit(0);
}
```

Mark each column that represents a valid possible output of this program with 'Yes' and each column which is impossible with 'No'.

A A B C D	A C D H	A B H C D	A A B C H D	A A C B D	A A C H D

4. Exceptional Control Flow (8 points)

The following C program computes an array v by a call to an external function init_vector, sums up the elements of v, and prints the result.

```
#include <setjmp.h>
#define VSIZE 1024
double v[VSIZE];

jmp_buf k;

double sum(int n, double *v) {
  int i;
  double x = 0.0;
  for (i = 0; i < n; i++) {
   /* place additional code here */


    x += v[i];
  }
  return x;
}

int main () {
  init_vector(VSIZE, v);
  printf("%f\n", sum(VSIZE,v));
  exit(0);
}
/* put new version of main below */
```

We want to change the sum function to indicate an error if any of the elements of the input vector is negative by using a long jump to k.

Add a line of code to sum in the indicated place and write a new version of main that prints the same output if there is no error condition and prints Illegal vector if any of the vector elements is negative. Your main function must still call sum.

5. Garbage Collection (12 points)

In this problem we consider a tiny list processing machine in which each memory word consists of two bytes: the first byte is a pointer to the tail of the list and the second byte is a data element. The end of a list is marked by a pointer of `0x00`. We assume that the data element is never a pointer.

1. (6 points) In the first question we consider a copying collector.

   We start with the memory state on the left, where the range `0x10`–`0x1F` is the from-space and the range `0x20`–`0x2F` is the to-space. All addresses and values in the diagram are in hexadecimal.

   Write in the state of memory after a copying collector is called with root pointers `0x12` and `0x14`. You may leave cells that remain unchanged blank.

Before GC

Addr	Ptr	Data
10	00	00
12	1C	3F
14	1E	0E
16	04	44
18	1C	01
1A	14	20
1C	18	02
1E	00	00

After GC

Addr	Ptr	Data	Addr	Ptr	Data
10			20		
12			22		
14			24		
16			26		
18			28		
1A			2A		
1C			2C		
1E			2E		

2. (6 points) In the second question we consider a mark and sweep collector.

   We use the lowest bit of the pointer as the mark bit, because it is normally always zero since pointers must be word-aligned.

   Assume the garbage collector is once again called with root pointers `0x12` and `0x14`. Write in the state of memory after the mark phase, and then again after the sweep phase. You may leave cells that remain unchanged blank. Assume that the free list starts at the lowest unoccupied address and is terminated by a NULL (`0x00`) pointer.

Before GC

Addr	Ptr	Data
10	00	00
12	1C	3F
14	1E	0E
16	04	44
18	1C	01
1A	14	20
1C	18	02
1E	00	00

After Marking Phase

Addr	Ptr	Data
10		
12		
14		
16		
18		
1A		
1C		
1E		

After Sweep Phase

Addr	Ptr	Data
10		
12		
14		
16		
18		
1A		
1C		
1E		

6. Cyclone (11 points)

Consider the following C program which initializes a linked list `p0` with a call to `init_list`, then counts the number of positive members in `p0` and prints the result.

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

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

int main () {
  int k = 0;
  int *w = &k;
  List* p0 = init_list();
  count_pos(p0, w);
  printf ("%d\n", k);
  return 0;
}
```

1. (6 points) When this program is ported to Cyclone, each pointer variable must be considered to see which attributes it should be assigned. Indicate which attributes apply by writing "yes" or "no" in the appropriate box.

   You may assume that `init_list();` returns a pointer to a (possibly empty) linked list, allocated on the heap. Recall that a thin pointer is simply a bounded pointer with bound 1, written as `@numelts(1)`.

	@numelts(1)	@notnull
w		
p0		
p		

2. (5 points) Now consider the function `upto (`n`, `p`)` which allocates a linked list $0, \ldots, n-1$ followed by the tail p and returns a pointer to it. Therefore, if we call it with `upto (`n`, NULL)` it will return a pointer to the list $0, \ldots, n-1$.

```
List* upto (int n, List* p) {
  List* q;
  if (n > 0) {
    q = (List *) malloc(sizeof(List));
    q->data = n-1;
    q->next = p;
    return upto(n-1, q);
  } else {
    return p;
  }
}

List* init_list() {
  List* q0 = upto (10, NULL);
  return q0;
}

int main () {
  int k = 0;
  int *w = &k;
  List* p0 = init_list();
  count_pos(p0, w);
  printf ("%d\n", k);
  return 0;
}
```

For each pointer variable, indicate which region it points to. Recall that `'H` is the notation for the global heap region, and that `'f` is the notation for the stack region of function `f`.

Variable	Region
`p`	
`q`	
`q0`	
`w`	
`p0`	

15-213 Introduction to Computer Systems

Exam 2

April 5, 2005

Name: **Model Solution**
Andrew User ID: fp
Recitation Section: ____________________

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

Problem	Max	Score
1	**14**	
2	**18**	
3	**12**	
4	**8**	
5	**12**	
6	**11**	
Total	**75**	

1. Symbols and Linking (14 points)

Consider the following two files, `fib1.c` and `fib2.c`:

```
/* fib1.c */
#define MAXFIB 1024

int table[MAXFIB];
int fib(int n);

int main(int argc, char **argv) {
  int n;
  table[0] = 0;
  table[1] = 1;
  argc--; argv++; /* skip command name */
  while (argc > 0) {
    if (sscanf(*argv, "%d", &n) != 1 || n < 0 || n >= MAXFIB) {
      printf ("Error: %s not an int or out of range\n", *argv);
      exit (0);
    }
    printf("fib(%d) = %d\n", n, fib(n));
    argc--; argv++;
  }
}

/* fib2.c */
int* table;

int fib(int n) {
  static int num = 2;
  if (n >= num) {
    int i = num;
    while (i <= n) {
      table[i] = table[i-1] + table[i-2];
      i++;
    }
    num = i;
  }
  return table[n];
}
```

1. (8 points) Fill in the following tables by stating for each name whether it is local or global, whether it is strong or weak, and the section it occupies in that module (`.text`, `.data`, or`.bss`). Cross out any box in the table that does not apply. For example, cross out the first box in a line of the symbol is not in the symbol table, or cross out the second box in a line if the symbol is not global (and therefore neither weak nor strong).

   `fib1.c`

	Local or Global?	Strong or Weak?	Which segment?
`table`	**global**	**weak**	**`.bss`**
`fib`	**global**	**weak**	**X (or .bss)**
`num`	**X**	**X**	**X**

   `fib2.c`

	Local or Global?	Strong or Weak?	Which segment?
`table`	**global**	**weak**	**`.bss`**
`fib`	**global**	**strong**	**`.text`**
`num`	**local**	**X**	**`.data`**

2. (3 points) When the two files are linked together, symbols will be resolved. For each symbol below, show which module it will be defined in (write `fib1` or `fib2` or `not determined`).

	Defined in module?
`table`	**not determined**
`fib`	**fib2**
`num`	**fib2**

3. (3 points) The code which is generated by `gcc -o fib fib1.c fib2.c` may not execute correctly. Explain succinctly why.

   The symbol `table` is weak in both modules, and therefore the linker picks arbitrarily whether to resolve to `fib1` or `fib2`. However, the module `fib2` reserves only a single word (4 bytes) for it, which is too small, and a segmentation violation is likely to ensue if `fib2` is prefered over `fib1`.

2. Virtual Address Translation (18 points)

We consider a virtual address system with the following parameters.

- The memory is byte addressable.
- Virtual addresses are 20 bits wide.
- Physical addresses are 16 bits wide.
- The page size is 4096 bytes.
- The TLB is 4-way set associative with 16 total entries.

In the following tables, all numbers are given in hexadecimal. The contents of the TLB and the page table for the first 16 virtual pages are as follows. If a VPN is not listed in the page table, assume it generates a page fault.

TLB

Index	Tag	PPN	Valid
0	03	B	1
	07	6	0
	28	3	1
	01	F	0
1	31	0	1
	12	3	0
	07	E	1
	0B	1	1
2	2A	A	0
	11	1	0
	1F	8	1
	07	5	1
3	07	3	1
	3F	F	0
	10	D	0
	32	0	0

Page Table

VPN	PPN	Valid
00	7	1
01	8	1
02	9	1
03	A	1
04	6	0
05	3	0
06	1	0
07	8	0
08	2	0
09	3	0
0A	1	1
0B	6	1
0C	C	1
0D	D	0
0E	E	0
0F	D	1

1. (4 points) In the four rows below, mark the bits that constitute the indicated part of the virtual address with an **X**. Leave the remaining bits of each row blank.

Virtual Page Number

	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VPN	x	x	x	x	x	x	x	x												

Virtual Page Offset

	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VPO									x	x	x	x	x	x	x	x	x	x	x	x

TLB Tag

	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TLBT	x	x	x	x	x	x														

TLB Index

	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TLBI							x	x												

2. (7 points) For the virtual address `0x7E37C`, indicate the physical address and various results of the translation. If there is a page fault, enter "—" for the PPN and Physical Address. All answers should be given in hexadecimal.

Virtual Address (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Parameter	Value
VPN	**7E**
TLB Tag	**1F**
TLB Index	**2**
TLB Hit? (Y/N)	**Y**
Page Fault? (Y/N)	**N**
PPN	**8**
Physical Address	**0x837C**

3. (7 points) For the virtual address `0x16A48`, indicate the physical address and various results of the translation. If there is a page fault, enter "—" for the PPN and Physical Address. All answers should be given in hexadecimal.

Virtual Address (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Parameter	Value
VPN	**16**
TLB Tag	**05**
TLB Index	**2**
TLB Hit? (Y/N)	**N**
Page Fault? (Y/N)	**Y**
PPN	**—**
Physical Address	**—**

3. Process Control (12 points)

Consider the following C program. For space reasons, we do not check return codes, so assume that all functions return normally. Also assume that `printf` is unbuffered.

```
void handler(int sig) {
  printf("H\n");
  exit(0);
}

int main() {
  pid_t pid1, pid2;

  signal(SIGUSR1,handler);

  pid1 = fork();
  if (pid1 == 0) {
    pid2 = fork();
    printf("A\n");
    if (pid2 == 0) {
      printf("B\n");
      exit(0);
    }
    printf("C\n");
    kill(pid2,SIGUSR1);
    exit(0);
  }
  if (waitpid(pid1, NULL, 0) > 0) {
    printf("D\n");
  }
  exit(0);
}
```

Mark each column that represents a valid possible output of this program with 'Yes' and each column which is impossible with 'No'.

Yes	**Yes**	**No**	**Yes**	**Yes**	**Yes**
A	A	A	A	A	A
A	C	B	A	A	A
B	D	H	B	C	C
C	H	C	C	B	H
D		D	H	D	D
			D		

4. Exceptional Control Flow (8 points)

The following C program computes an array v by a call to an external function init_vector, sums up the elements of v, and prints the result.

```
#include <setjmp.h>
#define VSIZE 1024
double v[VSIZE];

jmp_buf k;

double sum(int n, double *v) {
  int i;
  double x = 0.0;
  for (i = 0; i < n; i++) {
   /* place additional code here */


    x += v[i];
  }
  return x;
}

int main () {
  init_vector(VSIZE, v);
  printf("%f\n", sum(VSIZE,v));
  exit(0);
}
/* put new version of main below */
```

We want to change the sum function to indicate an error if any of the elements of the input vector is negative by using a long jump to k.

Add a line of code to sum in the indicated place and write a new version of main that prints the same output if there is no error condition and prints Illegal vector if any of the vector elements is negative. Your main function must still call sum.

Add

```
if (v[i] < 0.0) longjmp(k, 1);
```

and

```
int main () {
  init_vector(VSIZE, v);
  if (setjmp(k) == 0)
    printf("%f\n", sum(VSIZE,v));
  else
    printf("Illegal vector\n");
  exit(0);
}
```

5. Garbage Collection (12 points)

In this problem we consider a tiny list processing machine in which each memory word consists of two bytes: the first byte is a pointer to the tail of the list and the second byte is a data element. The end of a list is marked by a pointer of `0x00`. We assume that the data element is never a pointer.

1. (6 points) In the first question we consider a copying collector.

   We start with the memory state on the left, where the range `0x10`–`0x1F` is the from-space and the range `0x20`–`0x2F` is the to-space. All addresses and values in the diagram are in hexadecimal.

   Write in the state of memory after a copying collector is called with root pointers `0x12` and `0x14`. You may leave cells that remain unchanged blank.

Before GC

Addr	Ptr	Data
10	00	00
12	1C	3F
14	1E	0E
16	04	44
18	1C	01
1A	14	20
1C	18	02
1E	00	00

After GC

Addr	Ptr	Data	Addr	Ptr	Data
10			20	**22**	**3F**
12	**20**		22	**24**	**02**
14	**26**		24	**22**	**01**
16			26	**28**	**0E**
18	**24**		28	**00**	**00**
1A			2A		
1C	**22**		2C		
1E	**28**		2E		

Addresses `2A`–`2F` are now free, and the range `10`–`1F` becomes the to-space the next time the garbage collector is called.

2. (6 points) In the second question we consider a mark and sweep collector.

   We use the lowest bit of the pointer as the mark bit, because it is normally always zero since pointers must be word-aligned.

   Assume the garbage collector is once again called with root pointers `0x12` and `0x14`. Write in the state of memory after the mark phase, and then again after the sweep phase. You may leave cells that remain unchanged blank. Assume that the free list starts at the lowest unoccupied address and is terminated by a NULL (`0x00`) pointer.

Before GC

Addr	Ptr	Data
10	00	00
12	1C	3F
14	1E	0E
16	04	44
18	1C	01
1A	14	20
1C	18	02
1E	00	00

After Marking Phase

Addr	Ptr	Data
10		
12	**1D**	
14	**1F**	
16		
18	**1D**	
1A		
1C	**19**	
1E	**01**	

After Sweep Phase

Addr	Ptr	Data
10	**16**	
12	**1C**	
14	**1E**	
16	**1A**	
18	**1C**	
1A	**00**	
1C	**18**	
1E	**00**	

The linked free list starts at `10` and ends at `1A`.

6. Cyclone (11 points)

Consider the following C program which initializes a linked list p0 with a call to init_list, then counts the number of positive members in p0 and prints the result.

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

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

int main () {
  int k = 0;
  int *w = &k;
  List* p0 = init_list();
  count_pos(p0, w);
  printf ("%d\n", k);
  return 0;
}
```

1. (6 points) When this program is ported to Cyclone, each pointer variable must be considered to see which attributes it should be assigned. Indicate which attributes apply by writing "yes" or "no" in the appropriate box.

   You may assume that init_list(); returns a pointer to a (possibly empty) linked list, allocated on the heap. Recall that a thin pointer is simply a bounded pointer with bound 1, written as @numelts(1).

	@numelts(1)	@notnull
w	**yes**	**yes**
p0	**yes**	**no**
p	**yes**	**no**

2. (5 points) Now consider the function `upto (`n`, `p`)` which allocates a linked list $0, \ldots, n-1$ followed by the tail p and returns a pointer to it. Therefore, if we call it with `upto (`n`, NULL)` it will return a pointer to the list $0, \ldots, n-1$.

```
List* upto (int n, List* p) {
  List* q;
  if (n > 0) {
    q = (List *) malloc(sizeof(List));
    q->data = n-1;
    q->next = p;
    return upto(n-1, q);
  } else {
    return p;
  }
}

List* init_list() {
  List* q0 = upto (10, NULL);
  return q0;
}

int main () {
  int k = 0;
  int *w = &k;
  List* p0 = init_list();
  count_pos(p0, w);
  printf ("%d\n", k);
  return 0;
}
```

For each pointer variable, indicate which region it points to. Recall that `'H` is the notation for the global heap region, and that `'f` is the notation for the stack region of function `f`.

Variable	Region
p	**'H**
q	**'H**
q0	**'H**
w	**'main**
p0	**'H**

15-213 Introduction to Computer Systems

Exam 2

April 11, 2006

Name: ____________________

Andrew User ID: ____________________

Recitation Section: ____________________

- This is an open-book exam.
- Notes and calculators are permitted, but not computers.
- Write your answer legibly in the space provided.
- You have 80 minutes for this exam.
- We will drop your lowest score among questions 1–6.

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

1. Symbols and Linking (15 points)

Consider the following three files, `main.c`, `fib.c`, and `bignat.c`:

```
/* main.c */
void fib (int n);
int main (int argc, char** argv) {
  int n = 0;
  sscanf(argv[1], "%d", &n);
  fib(n);
}

/* fib.c */
#define N 16

static unsigned int ring[3][N];

static void print_bignat(unsigned int* a) {
  int i;
  for (i = N-1; i >= 0; i--)
    printf("%u ", a[i]);        /* print a[i] as unsigned int */
  printf("\n");
}

void fib (int n) {
  int i, carry;
  from_int(N, 0, ring[0]);      /* fib(0) = 0 */
  from_int(N, 1, ring[1]);      /* fib(1) = 1 */
  for (i = 0; i <= n-2; i++) {
    carry = plus(N, ring[i%3], ring[(i+1)%3], ring[(i+2)%3]);
    if (carry) { printf("Overflow at fib(%d)\n", i+2); exit(0); }
  }
  print_bignat(ring[n%3]);
}
```

Furthermore assume that a file `bignat.c` defines functions `plus` and `from_int` of the form

```
int plus (int n, unsigned int* a, unsigned int* b, unsigned int* c);
void from_int (int n, unsigned int k, unsigned int* a);
```

A possible definition of these functions is given in Problem 6, although this is not relevant here.

1. (9 points) Fill in the following tables by stating for each name whether it is local or global, and whether it is strong or weak. Cross out any box in the table that does not apply. For example, cross out the first box in a line if the symbol is not in the symbol table, or cross out the second box in a line if the symbol is not global (and therefore neither weak nor strong). Recall that in C, external functions do not need to be declared.

   `main.c`

	Local or Global?	Strong or Weak?
`fib`		
`main`		

   `fib.c`

	Local or Global?	Strong or Weak?
`ring`		
`print_bignat`		
`fib`		
`plus`		

2. (3 points) Now assume that the file `bignat.c` is compiled to a static library in archive format, `bignat.a` exporting the symbols `plus` and `from_int`.

   For each of the following calls to `gcc`, state if it

   (A) compiles and links correctly, or

   (B) linking fails due to undefined references, or

   (C) linking fails due to multiple definitions .

Command	Result (A, B, or C)
`gcc -o fib main.c fib.c bignat.a`	
`gcc -o fib bignat.a main.c fib.c`	
`gcc -o fib fib.c main.c bignat.a`	

3. (3 points) Consider the case where the programmer accidentally declared the variable `ring` in the file `fib.c` with

   ```
   static int ring[3][N];
   ```

   instead of `static unsigned int ring[3][N]`. Mark each of the following statements as true or false.

   - The files all still compile correctly. True False
   - The files can all still be linked correctly. True False
   - The resulting executable will still run correctly. True False

2. Virtual Address Translation (15 points)

We consider a virtual address system with the following parameters.

- The memory is byte addressable.
- Virtual addresses are 16 bits wide.
- Physical addresses are 16 bits wide.
- The page size is 1024 bytes.
- The TLB is fully associative with 16 total entries.

Recall that a fully associative cache has just one set of entries. In the following tables, all numbers are given in hexadecimal. The contents of the TLB and the page table for the first 16 virtual pages are as follows. If a VPN is not listed in the page table, assume it generates a page fault.

TLB

Tag	PPN	Valid
03	1B	1
06	06	0
28	23	1
01	18	0
31	01	1
12	00	0
07	3D	1
0B	11	1
2A	2C	0
11	1C	0
1F	03	1
08	14	1
09	2A	1
3F	30	0
10	0D	0
32	11	0

Page Table

VPN	PPN	Valid
00	27	1
01	0F	1
02	19	1
03	1B	1
04	06	0
05	03	0
06	06	0
07	3D	0
08	14	1
09	2A	1
0A	21	1
0B	11	1
0C	1C	1
0D	2D	0
0E	0E	0
0F	04	1

1. (5 points) In the four rows below, mark the bits that constitute the indicated part of the virtual address with an **X**. Leave the remaining bits of each row blank.

Virtual Page Number

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

Virtual Page Offset

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

TLB Tag

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

2. (5 points) For the virtual address `0xC7A4`, indicate the physical address and various results of the translation. If there is a page fault, enter "—" for the PPN and Physical Address. All answers should be given in hexadecimal.

   Virtual Address (one bit per box)

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

Parameter	Value
VPN	
TLB Tag	
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	
Physical Address	

3. (5 points) For the virtual address `0x05DD`, indicate the physical address and various results of the translation. If there is a page fault, enter "—" for the PPN and Physical Address. All answers should be given in hexadecimal.

   Virtual Address (one bit per box)

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

Parameter	Value
VPN	
TLB Tag	
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	
Physical Address	

3. Process Control (15 points)

Consider the following C program. For space reasons, we do not check return codes, so assume that all functions return normally. Also assume that `printf` is unbuffered.

```
int main() {
  pid_t pid1, pid2;

  pid1 = fork();
  pid2 = fork();
  if (pid1 && pid2) printf("A\n");
  if (pid1 || pid2) printf("B\n");
  exit(0);
}
```

Mark each column that represents a valid possible output of this program with 'Yes' and each column which is impossible with 'No'.

A	B	B	B	B
B	A	B	B	B
B	B	A	B	B
B	B	B	A	B

4. Signals (15 points)

Consider the following code.

```
        int i = 1;

        void handler (int sig) {
          i++;
        }

        int main() {
          pid_t pid;
          sigset_t s;
          sigemptyset(&s);
          sigaddset(&s, SIGUSR1);
          signal(SIGUSR1, handler);
          sigprocmask(SIG_BLOCK, &s, 0);
          pid = fork();
<LINE A>
          if (pid != 0) {
            i = 2;
<LINE B>
          } else {
            i = 3;
<LINE C>
          }
          sigprocmask(SIG_UNBLOCK, &s, 0);
          pause();    /* pause to allow all signals to arrive */
          printf("%d\n", i);
          exit(0);
        }
```

We assume that `pause();` pauses long enough that all signals in a process arrive before the following `printf` command is executed and that concurrently running processes proceed to their `pause();` command if they are not already there. We also assume that `fork();` is successful and that all processes run to successful completion.

Now consider the effect of adding the command

```
kill(pid, SIGUSR1);
```

either at `<LINE A>`, `<LINE B>`, or `<LINE C>`. Recall that if the first argument to `kill` is 0, it sends the signal to all processes in the current process group. For each resulting program, list the possible values that may be printed for any given run. You may assume that no other process sends a `SIGUSR1` signal.

1. (5 pts) `<LINE A>`

2. (5 pts) `<LINE B>`

3. (5 pts) `<LINE C>`

5. Garbage Collection (15 points)

In this problem we consider a tiny list processing machine in which each memory word consists of two bytes: the first byte is a pointer to the tail of the list and the second byte is a data element. The end of a list is marked by a pointer of `0x00`. We assume that the data element is never a pointer.

1. (8 points) In the first question we consider a copying collector.

   We start with the memory state on the left, where the range `0x10`–`0x1F` is the from-space and the range `0x20`–`0x2F` is the to-space. All addresses and values in the diagram are in hexadecimal.

   Write in the state of memory after a copying collector is called with root pointers `0x12` and `0x1A` and answer the subsequent question. You may leave cells that remain unchanged blank.

Before GC

Addr	Ptr	Data
10	12	2C
12	18	FF
14	12	0E
16	1C	AB
18	16	10
1A	00	00
1C	12	1D
1E	1A	00

After GC

Addr	Ptr	Data	Addr	Ptr	Data
10			20		
12			22		
14			24		
16			26		
18			28		
1A			2A		
1C			2C		
1E			2E		

After garbage collection, free space starts as address ______

2. (7 points) In the second question we consider a mark and sweep collector.

   We use the lowest bit of the pointer as the mark bit, because it is normally always zero since pointers must be word-aligned.

   Assume the garbage collector is once again called with root pointers `0x12` and `0x1A`. Write in the state of memory after the mark phase, and then again after the sweep phase and answer the subsequent question. You may leave cells that remain unchanged blank.

Before GC

Addr	Ptr	Data
10	12	2C
12	18	FF
14	12	0E
16	1C	AB
18	16	10
1A	00	00
1C	12	1D
1E	1A	00

After Marking Phase

Addr	Ptr	Data
10		
12		
14		
16		
18		
1A		
1C		
1E		

After Sweep Phase

Addr	Ptr	Data
10		
12		
14		
16		
18		
1A		
1C		
1E		

The free list now starts at address ______.

6. Cyclone (15 points)

Now consider the file `bignat.c` that implements addition and conversion of an unsigned integer to a bignat representation as we assumed in Problem 1.

```
typedef unsigned int uint;

int plus (int n, uint* a, uint* b, uint* c) {
  int i;
  int carry = 0;
  for (i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
    if (carry) {
      c[i]++;
      carry = (c[i] <= a[i]);
    } else {
      carry = (c[i] < a[i]);
    }
  }
  return carry;
}

void from_int (int n, uint k, uint* a) {
  int i;
  a[0] = k;
  for (i = 1; i < n; i++)
    a[i] = 0;
  return;
}
```

1. (3 points) Is it legal for `a`, `b`, and `c` to be pointers to different regions of memory? Circle one

   Yes   No

2. (4 points) Assume `a`, `b`, and `c` are declared as fat pointers. Use the notation $\texttt{curr}(p)$, $\texttt{lower}(p)$, and $\texttt{upper}(p)$ to denote the current value of a fat pointer p and its lower and upper bounds. The bounds are *inclusive*: $\texttt{lower}(p) \leq \texttt{curr}(p) \leq \texttt{upper}(p)$, and you may assume all fat pointers will always satisfy this invariant. Under which conditions on `n`, `a`, `b`, and `c` will this code execute without an illegal pointer access exception?

In order to avoid the still possible overflow, we would like to implement arbitrarily large unsigned integers as linked lists of unsigned integers. In order to avoid always using linked lists, we represent bignats as a tagged union, either consisting just of an integer or a pointer to a linked list representation.

```
struct List {
  unsigned int head;
  struct List* tail;
};

@tagged union Bignat {
  unsigned int x;
  struct List* l;
};
```

4. (5 points) Recall that Cyclone translates its source into C code. Give a representation of the tagged union construct above that would be a plausible result of a translation from Cyclone to C.

5. (3 points) Give a plausible translation of the following Cyclone code fragment using your translation of the tagged union.

   ```
   union Bignat a;
   a.x = 15213;
   ```

15-213 Introduction to Computer Systems

Exam 2

April 11, 2006

Name: **Model Solution**

Andrew User ID: **fp**

Recitation Section: ____________

- This is an open-book exam.
- Notes and calculators are permitted, but not computers.
- Write your answer legibly in the space provided.
- You have 80 minutes for this exam.
- We will drop your lowest score among questions 1–6.

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

1. Symbols and Linking (15 points)

Consider the following three files, `main.c`, `fib.c`, and `bignat.c`:

```
/* main.c */
void fib (int n);
int main (int argc, char** argv) {
  int n = 0;
  sscanf(argv[1], "%d", &n);
  fib(n);
}

/* fib.c */
#define N 16

static unsigned int ring[3][N];

static void print_bignat(unsigned int* a) {
  int i;
  for (i = N-1; i >= 0; i--)
    printf("%u ", a[i]);        /* print a[i] as unsigned int */
  printf("\n");
}

void fib (int n) {
  int i, carry;
  from_int(N, 0, ring[0]);      /* fib(0) = 0 */
  from_int(N, 1, ring[1]);      /* fib(1) = 1 */
  for (i = 0; i <= n-2; i++) {
    carry = plus(N, ring[i%3], ring[(i+1)%3], ring[(i+2)%3]);
    if (carry) { printf("Overflow at fib(%d)\n", i+2); exit(0); }
  }
  print_bignat(ring[n%3]);
}
```

Furthermore assume that a file `bignat.c` defines functions `plus` and `from_int` of the form

```
int plus (int n, unsigned int* a, unsigned int* b, unsigned int* c);
void from_int (int n, unsigned int k, unsigned int* a);
```

A possible definition of these functions is given in Problem 6, although this is not relevant here.

1. (9 points) Fill in the following tables by stating for each name whether it is local or global, and whether it is strong or weak. Cross out any box in the table that does not apply. For example, cross out the first box in a line if the symbol is not in the symbol table, or cross out the second box in a line if the symbol is not global (and therefore neither weak nor strong). Recall that in C, external functions do not need to be declared.

   `main.c`

	Local or Global?	Strong or Weak?
`fib`	**global**	**weak**
`main`	**global**	**strong**

   `fib.c`

	Local or Global?	Strong or Weak?
`ring`	**local**	**X**
`print_bignat`	**local**	**X**
`fib`	**global**	**strong**
`plus`	**global**	**weak**

2. (3 points) Now assume that the file `bignat.c` is compiled to a static library in archive format, `bignat.a` exporting the symbols `plus` and `from_int`.

   For each of the following calls to `gcc`, state if it

   (A) compiles and links correctly, or

   (B) linking fails due to undefined references, or

   (C) linking fails due to multiple definitions .

Command	Result (A, B, or C)
`gcc -o fib main.c fib.c bignat.a`	**A**
`gcc -o fib bignat.a main.c fib.c`	**B**
`gcc -o fib fib.c main.c bignat.a`	**A**

3. (3 points) Consider the case where the programmer accidentally declared the variable `ring` in the file `fib.c` with

   ```
   static int ring[3][N];
   ```

   instead of `static unsigned int ring[3][N]`. Mark each of the following statements as true or false.

   - The files all still compile correctly. True False **True**
   - The files can all still be linked correctly. True False **True**
   - The resulting executable will still run correctly. True False **True**

   Because conversions between signed and unsigned integers do not change the representation, and their sizes are identical, calls to the library functions will behave exactly as before. Comparisons will behave differently, but they are not used in this code.

2. Virtual Address Translation (15 points)

We consider a virtual address system with the following parameters.

- The memory is byte addressable.
- Virtual addresses are 16 bits wide.
- Physical addresses are 16 bits wide.
- The page size is 1024 bytes.
- The TLB is fully associative with 16 total entries.

Recall that a fully associative cache has just one set of entries. In the following tables, all numbers are given in hexadecimal. The contents of the TLB and the page table for the first 16 virtual pages are as follows. If a VPN is not listed in the page table, assume it generates a page fault.

TLB

Tag	PPN	Valid
03	1B	1
06	06	0
28	23	1
01	18	0
31	01	1
12	00	0
07	3D	1
0B	11	1
2A	2C	0
11	1C	0
1F	03	1
08	14	1
09	2A	1
3F	30	0
10	0D	0
32	11	0

Page Table

VPN	PPN	Valid
00	27	1
01	0F	1
02	19	1
03	1B	1
04	06	0
05	03	0
06	06	0
07	3D	0
08	14	1
09	2A	1
0A	21	1
0B	11	1
0C	1C	1
0D	2D	0
0E	0E	0
0F	04	1

1. (5 points) In the four rows below, mark the bits that constitute the indicated part of the virtual address with an **X**. Leave the remaining bits of each row blank.

Virtual Page Number

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

Virtual Page Offset

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

TLB Tag

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

2. (5 points) For the virtual address `0xC7A4`, indicate the physical address and various results of the translation. If there is a page fault, enter "—" for the PPN and Physical Address. All answers should be given in hexadecimal.

Virtual Address (one bit per box)

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

Parameter	Value
VPN	**31**
TLB Tag	**31**
TLB Hit? (Y/N)	**Y**
Page Fault? (Y/N)	**N**
PPN	**01**
Physical Address	**0x07A4**

3. (5 points) For the virtual address `0x05DD`, indicate the physical address and various results of the translation. If there is a page fault, enter "—" for the PPN and Physical Address. All answers should be given in hexadecimal.

Virtual Address (one bit per box)

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

Parameter	Value
VPN	**01**
TLB Tag	**01**
TLB Hit? (Y/N)	**N**
Page Fault? (Y/N)	**N**
PPN	**0F**
Physical Address	**0x3DDD**

3. Process Control (15 points)

Consider the following C program. For space reasons, we do not check return codes, so assume that all functions return normally. Also assume that `printf` is unbuffered.

```
int main() {
  pid_t pid1, pid2;

  pid1 = fork();
  pid2 = fork();
  if (pid1 && pid2) printf("A\n");
  if (pid1 || pid2) printf("B\n");
  exit(0);
}
```

Mark each column that represents a valid possible output of this program with 'Yes' and each column which is impossible with 'No'.

Yes	**Yes**	**Yes**	**No**	**No**
A	B	B	B	B
B	A	B	B	B
B	B	A	B	B
B	B	B	A	B

4. Signals (15 points)

Consider the following code.

```
        int i = 1;

        void handler (int sig) {
          i++;
        }

        int main() {
          pid_t pid;
          sigset_t s;
          sigemptyset(&s);
          sigaddset(&s, SIGUSR1);
          signal(SIGUSR1, handler);
          sigprocmask(SIG_BLOCK, &s, 0);
          pid = fork();
<LINE A>
          if (pid != 0) {
            i = 2;
<LINE B>
          } else {
            i = 3;
<LINE C>
          }
          sigprocmask(SIG_UNBLOCK, &s, 0);
          pause();    /* pause to allow all signals to arrive */
          printf("%d\n", i);
          exit(0);
        }
```

We assume that `pause();` pauses long enough that all signals in a process arrive before the following `printf` command is executed and that concurrently running processes proceed to their `pause();` command if they are not already there. We also assume that `fork();` is successful and that all processes run to successful completion.

Now consider the effect of adding the command

```
          kill(pid, SIGUSR1);
```

either at `<LINE A>`, `<LINE B>`, or `<LINE C>`. Recall that if the first argument to `kill` is 0, it sends the signal to all processes in the current process group. For each resulting program, list the possible values that may be printed for any given run. You may assume that no other process sends a `SIGUSR1` signal.

1. (5 pts) `<LINE A>`

3 4, 4 3, 3 5, or 5 3

2. (5 pts) `<LINE B>`

2 4 or 4 2

3. (5 pts) `<LINE C>`

3 4 or 4 3

5. Garbage Collection (15 points)

In this problem we consider a tiny list processing machine in which each memory word consists of two bytes: the first byte is a pointer to the tail of the list and the second byte is a data element. The end of a list is marked by a pointer of `0x00`. We assume that the data element is never a pointer.

1. (8 points) In the first question we consider a copying collector.

   We start with the memory state on the left, where the range `0x10`–`0x1F` is the from-space and the range `0x20`–`0x2F` is the to-space. All addresses and values in the diagram are in hexadecimal.

   Write in the state of memory after a copying collector is called with root pointers `0x12` and `0x1A` and answer the subsequent question. You may leave cells that remain unchanged blank.

Before GC

Addr	Ptr	Data
10	12	2C
12	18	FF
14	12	0E
16	1C	AB
18	16	10
1A	00	00
1C	12	1D
1E	1A	00

After GC

Addr	Ptr	Data	Addr	Ptr	Data
10			20	**24**	**FF**
12	**20**		22	**00**	**00**
14			24	**26**	**10**
16	**26**		26	**28**	**AB**
18	**24**		28	**20**	**1D**
1A	**22**		2A		
1C	**28**		2C		
1E			2E		

After garbage collection, free space starts as address **2A**

2. (7 points) In the second question we consider a mark and sweep collector.

   We use the lowest bit of the pointer as the mark bit, because it is normally always zero since pointers must be word-aligned.

   Assume the garbage collector is once again called with root pointers `0x12` and `0x1A`. Write in the state of memory after the mark phase, and then again after the sweep phase and answer the subsequent question. You may leave cells that remain unchanged blank.

Before GC

Addr	Ptr	Data
10	12	2C
12	18	FF
14	12	0E
16	1C	AB
18	16	10
1A	00	00
1C	12	1D
1E	1A	00

After Marking Phase

Addr	Ptr	Data
10		
12	**19**	
14		
16	**1D**	
18	**17**	
1A	**01**	
1C	**13**	
1E		

After Sweep Phase

Addr	Ptr	Data
10	**14**	
12	**18**	
14	**1E**	
16	**1C**	
18	**16**	
1A	**00**	
1C	**12**	
1E	**00**	

The free list now starts at address **10**.

6. Cyclone (15 points)

Now consider the file `bignat.c` that implements addition and conversion of an unsigned integer to a bignat representation as we assumed in Problem 1.

```
typedef unsigned int uint;

int plus (int n, uint* a, uint* b, uint* c) {
  int i;
  int carry = 0;
  for (i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
    if (carry) {
      c[i]++;
      carry = (c[i] <= a[i]);
    } else {
      carry = (c[i] < a[i]);
    }
  }
  return carry;
}

void from_int (int n, uint k, uint* a) {
  int i;
  a[0] = k;
  for (i = 1; i < n; i++)
    a[i] = 0;
  return;
}
```

1. (3 points) Is it legal for `a`, `b`, and `c` to be pointers to different regions of memory? Circle one

   Yes   No   **Yes**

2. (4 points) Assume `a`, `b`, and `c` are declared as fat pointers. Use the notation $\texttt{curr}(p)$, $\texttt{lower}(p)$, and $\texttt{upper}(p)$ to denote the current value of a fat pointer p and its lower and upper bounds. The bounds are *inclusive*: $\texttt{lower}(p) \leq \texttt{curr}(p) \leq \texttt{upper}(p)$, and you may assume all fat pointers will always satisfy this invariant. Under which conditions on `n`, `a`, `b`, and `c` will this code execute without an illegal pointer access exception?

   $\texttt{curr}(p) + \texttt{n} - 1 \leq \texttt{upper}(p)$ for $p \in \{\texttt{a}, \texttt{b}, \texttt{c}\}$.

In order to avoid the still possible overflow, we would like to implement arbitrarily large unsigned integers as linked lists of unsigned integers. In order to avoid always using linked lists, we represent bignats as a tagged union, either consisting just of an integer or a pointer to a linked list representation.

```
struct List {
  unsigned int head;
  struct List* tail;
};

@tagged union Bignat {
  unsigned int x;
  struct List* l;
};
```

4. (5 points) Recall that Cyclone translates its source into C code. Give a representation of the tagged union construct above that would be a plausible result of a translation from Cyclone to C.

   ```
   enum Tag {INT, LIST};
   struct Bignat {
     Tag t;
     union U {
       unsigned int x;
       struct List* l;
     } u;
   };
   ```

5. (3 points) Give a plausible translation of the following Cyclone code fragment using your translation of the tagged union.

   ```
   union Bignat a;
   a.x = 15213;
   ```

   ```
   struct Bignat a;
   a.t = INT;
   a.u.x = 15213;
   ```

15-213 Introduction to Computer Systems

Exam 2

April 10, 2007

Name: ____________________

Andrew User ID: ____________________

Recitation Section: ____________________

- This is an open-book exam.
- Notes and calculators are permitted, but not computers.
- Write your answer legibly in the space provided.
- You have 80 minutes for this exam.
- We will drop your lowest score among questions 1–6.

	Problem	**Max**	**Score**
Symbols and Linking	1	**15**	
Virtual Address Translation	2	**15**	
Process Control	3	**15**	
Signals	4	**15**	
Garbage Collection	5	**15**	
Cyclone	6	**15**	
	Total	**75**	

1. Symbols and Linking (15 points)

In this problem we consider issues of symbols and linking. Consider the following three files.

File polygon.h:

```
struct Node {
  float pos[2];
  int marked;                   /* only for GC */
  struct Node* next;
  struct Node* prev;
};
typedef struct Node node;

node* alloc();                  /* allocated new node */
void init();                    /* initialize store */
void gc();                      /* call garbage collector */
```

File main.c (with portions of the function elided)

```
#include "polygon.h"
node* root_ptr;                /* root pointer, for GC */

int main () {
  node* p;
  init();
  p = alloc();
  root_ptr = p;                /* GC root is first allocated pointer */
  ...
  gc();
  ...
  return 0;
}
```

File gc.c (with function bodies elided)

```
#include "polygon.h"
#define N (1<<20)
static node polygon[N];         /* polygon array */
static node* free_ptr;          /* free list */
static node* root_ptr;          /* root pointer, for GC */

void mark(node* v) {...}
void sweep () {...}
void gc () {...}
void init () {...}
node* alloc () {...}
```

Possible definition of the functions in `gc.c` are revisited in Problems 5 and 6 although this is not relevant here.

Recall that a line `#include "`*file*`"` will literally be replaced by the contents of `"`*file*`"` by the C preprocessor. We compile the files to obtain an executable object file `polygon` with `gcc -o polygon main.c gc.c`. Questions related to symbols and linking therefore refer to the *expanded* versions of the files.

1. (9 points) Fill in the following tables by stating for each name whether it is local or global, and whether it is strong or weak. Cross out any box in the table that does not apply. For example, cross out the first box in a line if the symbol is not in the symbol table, or cross out the second box in a line if the symbol is not global (and therefore neither weak nor strong).

   `main.c`

	Local or Global?	Strong or Weak?
`root_ptr`		
`init`		
`main`		

   `gc.c`

	Local or Global?	Strong or Weak?
`N`		
`polygon`		
`root_ptr`		
`alloc`		

2. (3 points) Explain why linking succeeds to create an executable despite the fact that some symbols are declared in both files.

3. (3 points) Explain why the executable fails to be correct despite the fact that it compiles and links without warning. Propose how to fix the bug. Be clear in your suggestion correction (for example, *replace the line . . . with . . .*).

2. Virtual Address Translation (15 points)

In this problem we consider virtual address translation using a **two-level page table**. The parameters of the machine are as follows:

- The memory is byte addressable.
- Virtual addresses are 32 bits wide.
- The 32 bits of the virtual address are divided into 8 bits for VPN$_1$, 8 bits for VPN$_2$, and 16 bits for the VPO.
- Physical addresses are 24 bits wide.

1. (3 pts) Given the parameters above, fill in the blanks below.

   (i) The page size is ____________.

   (ii) The maximal physical memory size is ____________.

   (iii) Assuming a page table entry is 32 bits, the size of a page table is ____________.

Next we consider the behavior of address translation using the two-level page table. We assume that page table entries have the following form, where unspecified bits are irrelevant for this problem:

- Level 1:
  - Bits 31–16: High 16 bits of level 2 page table physical base address
  - Bit 0: 1 = Present in physical memory
- Level 2:
  - Bits 31–24: PPN
  - Bit 0: 1 = Present in physical memory

For the following questions, assume that the VPN is not cached in the TLB, so we have to consult the page tables. The PTBR is set at `0xC80000`.

Address	Content
`0xC80000`	`0xCA1C0001`
`0xC80004`	`0xCA1B0000`
`0xC80008`	`0xCA1D0001`
`0xC8000C`	`0xCA1B0001`
`0xC80010`	`0x00000000`
`0xC80014`	`0xCA1B0001`
`0xC80018`	`0xC8000000`
`0xC8001C`	`0xCA1B0001`

Address	Content
`0xCA1B00`	`0x07000001`
`0xCA1B04`	`0xFF000000`
`0xCA1B08`	`0x08000001`
`0xCA1B0C`	`0xCA000001`
`0xCA1B10`	`0x01000000`
`0xCA1B14`	`0x00000000`
`0xCA1B18`	`0x09000001`
`0xCA1B1C`	`0xCA000000`

2. (6 points) For the virtual address `0x0300F218`, indicate the physical address and various results of the translation. If there is a page fault, enter "—" for the answer and all subsequent results. All answers should be given in hexadecimal.

Parameter	Value
VPN_1	
VPN_2	
PTE (level 1)	
Page Fault? (Y/N)	
PTE (level 2)	
Page Fault? (Y/N)	
PPN	
Physical Address	

3. (6 points) For the virtual address `0x0507EE00`, indicate the physical address and various results of the translation. If there is a page fault, enter "—" for the PPN and Physical Address. All answers should be given in hexadecimal.

Parameter	Value
VPN_1	
VPN_2	
PTE (level 1)	
Page Fault? (Y/N)	
PTE (level 2)	
Page Fault? (Y/N)	
PPN	
Physical Address	

3. Process Control (15 points)

Consider the following C program. For space reasons, we do not check return codes, so assume that all functions return normally. Also assume that `printf` is unbuffered, and that each process successfully runs to completion.

```
int main() {
  int pid;

  pid = fork() && fork();
  if (!pid)
    printf("A\n");
  else
    printf("B\n");
  exit(0);
}
```

Mark each column that represents a valid possible output of this program with 'Yes' and each column which is impossible with 'No'.

A	A	A	A	A
A	A	A	A	A
	A	B	B	A
			B	B

4. Signals (15 points)

Consider the following code.

```
int val = 1;

void handler(int sig) {
  waitpid(-1, NULL, 0);
  val++;
}

int main() {
  int pid;
  signal(SIGCHLD, handler);
  pid = fork();
  if (pid == 0) {
    val = 0;
    exit(0);
  }
  printf("%d\n", val);
  exit(0);
}
```

Assume that all system calls succeed, and that all processes terminate normally. List all possible outputs of this program.

5. Garbage Collection (15 points)

In this problem we consider code for a specialized mark-and-sweep garbage collector, a skeleton of which was introduced in Problem 1. We first explain the data structures. A *polygon* is implemented as a doubly linked list of vertices, where each vertex has a link to its successor and predecessor vertices (the `next` and `prev` pointers, respectively). In addition the x and y position of each vertex are stored in `pos[0]` and `pos[1]`.

```
struct Node {
  float pos[2];
  int marked;                    /* only for GC */
  struct Node* next;
  struct Node* prev;
};
typedef struct Node node;
```

Finally, we have a *mark* (field `marked`) which is used exclusively by the garbage collector and should not be touched by the user program. For simplicity, we store this as a whole `int` instead of as a bit.

There is a static array `polygon` of size `N` containing all vertices. There is also a pointer to the beginning of a list of free vertices, `free_ptr`, and a single root pointer, `root_ptr` pointing to the polygon. The free pointer must be maintained by the garbage collector; the root pointer is maintained by the user program and may not be changed by the garbage collector.

```
#define N (1<<20)
node polygon[N];                 /* polygon store, max 1M vertices */
node* free_ptr;                  /* free list */
node* root_ptr;                  /* root pointer */
```

First, the garbage collector. This function is invoked either when the free list is empty when a node supposed to be allocated, or explicitly from the user program to "clean up" the store.

```
void gc () {                     /* call when free_ptr == NULL */
  mark(root_ptr);
  sweep();
}
```

1. (3 pts) Traversal of the heap during the marking phase of a mark-and-sweep garbage collector is (circle correct answer)

   (i) Depth first

   (ii) Breadth first

   (iii) Best first

   (iv) Arbitrary

2. (5 pts) The mark function takes a node as argument and marks all nodes reachable from the given node. Rather than using pointer reversal techniques, this mark function should be recursive. Complete the definition of mark. There are many solutions—for calibration, our solution adds 5 lines to the function body.

```
void mark(node* v) {











}
```

3. (7 pts) Next, complete the function sweep. Maintain the free list as a **singly linked** list with next pointing to the next free element. There are many solutions–for calibration, our solutions adds 7 lines to the function body.

```
void sweep () {
  int i;
  node* v;                /* use if needed */
  free_ptr = NULL;        /* initialize free list */
  for (i=0; i<N; i++) {














  }
}
```

6. Cyclone (15 points)

In this problem we reconsider the garbage collector sketched above and port it from C to Cyclone. We have left out some type declarations for you to fill in.

```
struct Node {
  float pos[2];
  int marked;                    /* only for GC */
  struct Node* next;
  struct Node* prev;
};
typedef struct Node node;

#define N (1<<20)
node polygon[N];                 /* polygon store, max 1M vertices */

________ free_ptr;  /* 1 */      /* free list */

________ root_ptr;  /* 2 */      /* root pointer */


________ alloc () { /* 3 */      /* return pointer to new vertex */

  ________ new_ptr;  /* 4 */
  if (!free_ptr) {
    gc();
    if (!free_ptr) {
      printf("Out of space!\n");
      exit(0);
    }
  }
  new_ptr = free_ptr;
  free_ptr = free_ptr->next;
  new_ptr->next = NULL;          /* init next pointer */
  new_ptr->prev = NULL;          /* init prev pointer */
  new_ptr->marked = 0;           /* unmarked */
  /* do not initialize pos[2] */
  return new_ptr;
}
```

```
int main () {

  ________ p1;   /* 5 */

  ________ p2;   /* 6 */
  init();                         /* initialize polygon store */
  p1 = alloc();                   /* create two-vertex "polygon" */
  p2 = alloc();
  p1->next = p2; p1->prev = p2;
  p2->next = p1; p2->next = p1;
  root_ptr = p1;
  gc();
  return 0;
}
```

1. (6 pts) For each of the 6 missing types, fill in the most precise type among the following which makes the code type-check without any implicit casts (and hence without warnings).

   (i) node@ (most precise)

   (ii) node*

   (iii) node? (least precise)

2. (3 pts) Assume that Cyclone considers global variables as living on the heap (region `H). Each of the following expressions denotes a pointer. Indicate the region that this pointer points to, or write *unknown* if this cannot be determined.

   (i) p1 in function main. ____________

   (ii) &p1 in function main. ____________

   (iii) new_ptr in function alloc. ____________

3. (3 pts) The pos[] fields are not initialized in the alloc function. Explain why this does not compromise safety of Cyclone.

15-213 Introduction to Computer Systems

Exam 2

April 10, 2007

Name: **Model Solution**

Andrew User ID: **fp**

Recitation Section: ____________________

- This is an open-book exam.
- Notes and calculators are permitted, but not computers.
- Write your answer legibly in the space provided.
- You have 80 minutes for this exam.
- We will drop your lowest score among questions 1–6.

	Problem	**Max**	**Score**
Symbols and Linking	1	**15**	**15**
Virtual Address Translation	2	**15**	**15**
Process Control	3	**15**	**15**
Signals	4	**15**	**15**
Garbage Collection	5	**15**	**15**
Cyclone	6	**15**	**12**
	Total	**75**	

1. Symbols and Linking (15 points)

In this problem we consider issues of symbols and linking. Consider the following three files.

File `polygon.h`:

```
struct Node {
  float pos[2];
  int marked;                   /* only for GC */
  struct Node* next;
  struct Node* prev;
};
typedef struct Node node;

node* alloc();                  /* allocated new node */
void init();                    /* initialize store */
void gc();                      /* call garbage collector */
```

File `main.c` (with portions of the function elided)

```
#include "polygon.h"
node* root_ptr;                /* root pointer, for GC */

int main () {
  node* p;
  init();
  p = alloc();
  root_ptr = p;                /* GC root is first allocated pointer */
  ...
  gc();
  ...
  return 0;
}
```

File `gc.c` (with function bodies elided)

```
#include "polygon.h"
#define N (1<<20)
static node polygon[N];         /* polygon array */
static node* free_ptr;          /* free list */
static node* root_ptr;          /* root pointer, for GC */

void mark(node* v) {...}
void sweep () {...}
void gc () {...}
void init () {...}
node* alloc () {...}
```

Possible definition of the functions in `gc.c` are revisited in Problems 5 and 6 although this is not relevant here.

Recall that a line `#include "`*file*`"` will literally be replaced by the contents of `"`*file*`"` by the C preprocessor. We compile the files to obtain an executable object file `polygon` with `gcc -o polygon main.c gc.c`. Questions related to symbols and linking therefore refer to the *expanded* versions of the files.

1. (9 points) Fill in the following tables by stating for each name whether it is local or global, and whether it is strong or weak. Cross out any box in the table that does not apply. For example, cross out the first box in a line if the symbol is not in the symbol table, or cross out the second box in a line if the symbol is not global (and therefore neither weak nor strong).

   `main.c`

	Local or Global?	Strong or Weak?
`root_ptr`	**global**	**weak**
`init`	**global**	**weak**
`main`	**global**	**strong**

   `gc.c`

	Local or Global?	Strong or Weak?
`N`	**X**	**X**
`polygon`	**local**	**X**
`root_ptr`	**local**	**X**
`alloc`	**global**	**strong**

2. (3 points) Explain why linking succeeds to create an executable despite the fact that some symbols are declared in both files.

> The symbols `alloc`, `init`, and `gc` are weak in `main.c` and strong in `gc.c`. The symbol `root_ptr` is local in `gc.c` and therefore does not conflict with the same symbol in `main.c`.

3. (3 points) Explain why the executable fails to be correct despite the fact that it compiles and links without warning. Propose how to fix the bug. Be clear in your suggestion correction (for example, *replace the line . . . with . . .*).

> The value assigned to `root_ptr` in `main.c` will not be seen by the garbage collector, since the symbols are different in the two files. One solution would be to hoist the declaration of `root_ptr` to `polygon.h`; another to remove its qualification as `static` in `gc.c`.

2. Virtual Address Translation (15 points)

In this problem we consider virtual address translation using a **two-level page table**. The parameters of the machine are as follows:

- The memory is byte addressable.
- Virtual addresses are 32 bits wide.
- The 32 bits of the virtual address are divided into 8 bits for VPN_1, 8 bits for VPN_2, and 16 bits for the VPO.
- Physical addresses are 24 bits wide.

1. (3 pts) Given the parameters above, fill in the blanks below.

   (i) The page size is **64KB**.

   (ii) The maximal physical memory size is **16MB**.

   (iii) Assuming a page table entry is 32 bits, the size of a page table is **1KB**.

Next we consider the behavior of address translation using the two-level page table. We assume that page table entries have the following form, where unspecified bits are irrelevant for this problem:

- Level 1:
  - Bits 31–16: High 16 bits of level 2 page table physical base address
  - Bit 0: 1 = Present in physical memory
- Level 2:
  - Bits 31–24: PPN
  - Bit 0: 1 = Present in physical memory

For the following questions, assume that the VPN is not cached in the TLB, so we have to consult the page tables. The PTBR is set at `0xC80000`.

Address	Content
`0xC80000`	`0xCA1C0001`
`0xC80004`	`0xCA1B0000`
`0xC80008`	`0xCA1D0001`
`0xC8000C`	`0xCA1B0001`
`0xC80010`	`0x00000000`
`0xC80014`	`0xCA1B0001`
`0xC80018`	`0xC8000000`
`0xC8001C`	`0xCA1B0001`

Address	Content
`0xCA1B00`	`0x07000001`
`0xCA1B04`	`0xFF000000`
`0xCA1B08`	`0x08000001`
`0xCA1B0C`	`0xCA000001`
`0xCA1B10`	`0x01000000`
`0xCA1B14`	`0x00000000`
`0xCA1B18`	`0x09000001`
`0xCA1B1C`	`0xCA000000`

2. (6 points) For the virtual address `0x0300F218`, indicate the physical address and various results of the translation. If there is a page fault, enter "—" for the answer and all subsequent results. All answers should be given in hexadecimal.

Parameter	Value
VPN_1	**03**
VPN_2	**00**
PTE (level 1)	**0xCA1B0001**
Page Fault? (Y/N)	**N**
PTE (level 2)	**0x07000001**
Page Fault? (Y/N)	**N**
PPN	**07**
Physical Address	**0x07F218**

3. (6 points) For the virtual address `0x0507EE00`, indicate the physical address and various results of the translation. If there is a page fault, enter "—" for the PPN and Physical Address. All answers should be given in hexadecimal.

Parameter	Value
VPN_1	**05**
VPN_2	**07**
PTE (level 1)	**0xCA1B0001**
Page Fault? (Y/N)	**N**
PTE (level 2)	**0xCA000000**
Page Fault? (Y/N)	**Y**
PPN	**—**
Physical Address	**—**

3. Process Control (15 points)

Consider the following C program. For space reasons, we do not check return codes, so assume that all functions return normally. Also assume that `printf` is unbuffered, and that each process successfully runs to completion.

```
int main() {
  int pid;

  pid = fork() && fork();
  if (!pid)
    printf("A\n");
  else
    printf("B\n");
  exit(0);
}
```

Mark each column that represents a valid possible output of this program with 'Yes' and each column which is impossible with 'No'.

No	**No**	**Yes**	**No**	**No**
A	A	A	A	A
A	A	A	A	A
	A	B	B	A
			B	B

4. Signals (15 points)

Consider the following code.

```
int val = 1;

void handler(int sig) {
  waitpid(-1, NULL, 0);
  val++;
}

int main() {
  int pid;
  signal(SIGCHLD, handler);
  pid = fork();
  if (pid == 0) {
    val = 0;
    exit(0);
  }
  printf("%d\n", val);
  exit(0);
}
```

Assume that all system calls succeed, and that all processes terminate normally. List all possible outputs of this program.

1 or 2

5. Garbage Collection (15 points)

In this problem we consider code for a specialized mark-and-sweep garbage collector, a skeleton of which was introduced in Problem 1. We first explain the data structures. A *polygon* is implemented as a doubly linked list of vertices, where each vertex has a link to its successor and predecessor vertices (the `next` and `prev` pointers, respectively). In addition the x and y position of each vertex are stored in `pos[0]` and `pos[1]`.

```
struct Node {
  float pos[2];
  int marked;                  /* only for GC */
  struct Node* next;
  struct Node* prev;
};
typedef struct Node node;
```

Finally, we have a *mark* (field `marked`) which is used exclusively by the garbage collector and should not be touched by the user program. For simplicity, we store this as a whole `int` instead of as a bit.

There is a static array `polygon` of size `N` containing all vertices. There is also a pointer to the beginning of a list of free vertices, `free_ptr`, and a single root pointer, `root_ptr` pointing to the polygon. The free pointer must be maintained by the garbage collector; the root pointer is maintained by the user program and may not be changed by the garbage collector.

```
#define N (1<<20)
node polygon[N];               /* polygon store, max 1M vertices */
node* free_ptr;                /* free list */
node* root_ptr;                /* root pointer */
```

First, the garbage collector. This function is invoked either when the free list is empty when a node supposed to be allocated, or explicitly from the user program to "clean up" the store.

```
void gc () {                   /* call when free_ptr == NULL */
  mark(root_ptr);
  sweep();
}
```

1. (3 pts) Traversal of the heap during the marking phase of a mark-and-sweep garbage collector is (circle correct answer)

   (i) Depth first **yes, depth first**

   (ii) Breadth first

   (iii) Best first

   (iv) Arbitrary

2. (5 pts) The `mark` function takes a node as argument and marks all nodes reachable from the given node. Rather than using pointer reversal techniques, this `mark` function should be recursive. Complete the definition of `mark`. There are many solutions—for calibration, our solution adds 5 lines to the function body.

```
void mark(node* v) {
  if (!v || v->marked)
    return;
  v->marked = 1;
  mark(v->next);
  mark(v->prev);
}
```

3. (7 pts) Next, complete the function `sweep`. Maintain the free list as a **singly linked** list with `next` pointing to the next free element. There are many solutions–for calibration, our solutions adds 7 lines to the function body.

```
void sweep () {
  int i;
  node* v;
  free_ptr = NULL;        /* initialize free list */
  for (i=0; i<N; i++) {
    v = &polygon[i];
    if (v->marked)
      v->marked = 0;
    else {
      v->next = free_ptr;
      free_ptr = v;
    }
  }
}
```

6. Cyclone (15 points)

In this problem we reconsider the garbage collector sketched above and port it from C to Cyclone. We have left out some type declarations for you to fill in.

```
struct Node {
  float pos[2];
  int marked;                    /* only for GC */
  struct Node* next;
  struct Node* prev;
};
typedef struct Node node;

#define N (1<<20)
node polygon[N];                 /* polygon store, max 1M vertices */

________ free_ptr;  /* 1 */      /* free list */

________ root_ptr;  /* 2 */      /* root pointer */


________ alloc () { /* 3 */      /* return pointer to new vertex */

  ________ new_ptr;  /* 4 */
  if (!free_ptr) {
    gc();
    if (!free_ptr) {
      printf("Out of space!\n");
      exit(0);
    }
  }
  new_ptr = free_ptr;
  free_ptr = free_ptr->next;
  new_ptr->next = NULL;          /* init next pointer */
  new_ptr->prev = NULL;          /* init prev pointer */
  new_ptr->marked = 0;           /* unmarked */
  /* do not initialize pos[2] */
  return new_ptr;
}
```

```
int main () {

  ________ p1;   /* 5 */

  ________ p2;   /* 6 */
  init();                          /* initialize polygon store */
  p1 = alloc();                    /* create two-vertex "polygon" */
  p2 = alloc();
  p1->next = p2; p1->prev = p2;
  p2->next = p1; p2->next = p1;
  root_ptr = p1;
  gc();
  return 0;
}
```

1. (6 pts) For each of the 6 missing types, fill in the most precise type among the following which makes the code type-check without any implicit casts (and hence without warnings).

   (i) `node@` (most precise)

   (ii) `node*`

   (iii) `node?` (least precise)

1 `node*` 2 `node*` 3 `node@` 4 `node@` 5 `node@` 6 `node@`

2. (3 pts) Assume that Cyclone considers global variables as living on the heap (region `` `H ``). Each of the following expressions denotes a pointer. Indicate the region that this pointer points to, or write *unknown* if this cannot be determined.

   (i) `p1` in function `main`. ____**`H**____

   (ii) `&p1` in function `main`. ____**`main**____

   (iii) `new_ptr` in function `alloc`. ____**`H**____

3. (3 pts) The `pos[]` fields are not initialized in the `alloc` function. Explain why this does not compromise safety of Cyclone.

   They are floating point numbers, so computation with them cannot lead to a memory error: the bit pattern in these fields can always be interpreted as a floating point number of some form.

Andrew login ID:____

Full Name:____

Recitation Section:____

CS 15-213, Spring 2008
Exam 2

Thu. April 3, 2008

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–H) 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 59 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. You may use any books or notes you like. No calculators or other electronic devices are allowed.
- Good luck!

1 (8):	
2 (9):	
3 (10):	
4 (12):	
5 (6):	
6 (8):	
7 (6):	
TOTAL (59):	

Problem 1. (8 points):

Consider the following C function:

```
data_t psum(data_t a[], data_t b[], data_t c[], int cnt)
{
    data_t r = 0;
    int i;
    for (i = 0; i < cnt; i++) {
        /* Inner loop expression */
        r =   r  +  b[i]  *  b[i]  +  a[i] * c[i] ;
    }
    return r;
}
```

In this code, data type `data_t` can be defined to different types using a `typedef` declaration.

According to the C rules for operator precedence and associativity, the line labeled "Inner loop expression" will be computed according to the following parenthesization:

```
        r =  (r  + (b[i]  *  b[i]))  +  (a[i]  *  c[i]) ;
```

Imagine we run this code on a machine in which multiplication requires 5 cycles, while addition requires 3. Assume that these latencies are the only factors constraining the performance of the program. Don't worry about the cost of memory references, the cost of operations that compute and check the loop index, resource limitations, etc.

We list 4 different parenthesizations for the "Inner loop expression" below; they will not all compute the same result. For each parenthesization, write down the CPE (cycles per element) that the function would achieve. **Hint:** All of your answers will be in the set $\{3, 5, 6, 8, 10, 9, 11, 13, 15\}$.

```
        // P1.  CPE =
        r =  (r  + (b[i]  *  b[i]))  +  (a[i]  *  c[i]) ;

        // P2.  CPE =
        r =  ((r  +  b[i])  *  b[i])  +  (a[i]  *  c[i]) ;

        // P3.  CPE =
        r =  r  +  ((b[i]  *  b[i])  +  (a[i]  *  c[i])) ;

        // P4.  CPE =
        r =  (r  +  b[i])  *  (b[i]  +  (a[i]  *  c[i])) ;
```

Problem 2. (9 points):

Consider the following C function to sum all the elements of a 3×3 matrix. Note that it is iterating over the matrix **column-wise**.

```
char sum_matrix(char matrix[3][3]) {
  int row, col;
  char sum = 0;
  for (col = 0; col < 3; col++) {
    for (row = 0; row < 3; row++) {
      sum += matrix[row][col];
    }
  }
  return sum;
}
```

Suppose we run this code on a machine whose memory system has the following characteristics:

- Memory is byte-addressable.
- There are registers, an L1 cache, and main memory.
- Char's are 8 bits wide.
- The cache is direct-mapped, with 4 sets and 4-byte blocks.

You should also assume:

- `matrix` begins at address 0.
- `sum`, `row` and `col` are in registers; that is, the only memory accesses during the execution of this function are to `matrix`.
- The cache is initially cold and the array has been initialized elsewhere.

Fill in the table below. In each cell, write "**h**" if there is a cache hit when accessing the corresponding element of the matrix, or "**m**" if there is a cache miss.

	0	1	2
0			
1			
2			

Problem 3. (10 points):

Consider the following C function that calculates a value analogous to a matrix version of the dot-product. Note the different access patterns of A and B. Assume $X = 64$, which implies the total number of accesses is $2 * X^2 = 8192$.

```
int A[X][X], B[X][X];

int matrix_dot_product()
{
  int i, j, s = 0;
  for (i = 0; i < X; i++)
    for (j = 0; j < X; j++)
      s += A[i][j] * B[j][i];

  return s;
}
```

Assume the following:

- Memory consists of registers, an L1 cache, and main memory.
- The cache is cold when the function is called and the arrays have been initialized elsewhere.
- Variables i, j, s, and pointers to A and B are all stored in registers.
- The arrays A and B are aligned such $A[0][0]$ begins a cache block, and $B[0][0]$ comes immediately following $A[X-1][X-1]$. Assume in the above code that $A[i][j]$ is accessed before $B[j][i]$ for each value of i and j.

Case 1: Assume the cache is a 16KB direct-mapped data cache with 8-byte blocks. Calculate the cache miss rate by giving the number of **cache misses**. For partial credit briefly explain (one to two sentences) where collisions in the cache occur w.r.t. this access pattern (using locations of the arrays as a guide).

```
miss rate = __________ / 8192 (3 pts)
```

For your reference, $X = 64$ and here's the code snippet again:

```
int A[X][X], B[X][X];

int matrix_dot_product()
{
  int i, j, s = 0;
  for (i = 0; i < X; i++)
    for (j = 0; j < X; j++)
      s += A[i][j] * B[j][i];

  return s;
}
```

Case 2: Assume the cache has as many sets as in case 1, but is two-way set associative using an LRU replacement policy with 8-byte blocks. Calculate the cache miss rate by giving the number of **cache misses**.

miss rate = __________ / 8192 (3 pts)

1. Would a larger cache size, with the same cache-line size, reduce the miss rate? (Yes / No)

Case 1: _____   Case 2: _____ (2 pts)

2. Would a larger cache-line, with the same cache size, reduce the miss rate? (Yes / No)

Case 1: _____   Case 2: _____ (2 pts)

Problem 4. (12 points):

The following problem concerns virtual memory and the way virtual addresses are translated into physical addresses. Below are the specifications of the system on which the translation occurs.

- The system is a 32-bit machine - words are 4 bytes.
- Memory is byte addressable.
- The maximum size of a virtual address space is 4GB.
- The system is configured with 64MB of physical memory.
- The page size is 4KB.
- The system uses a two-level page tables. Tables at both levels are 4096 bytes (1 page) and entries in both tables are 4 bytes as shown below.

In this problem, you are given parts of a memory dump of this system running 2 processes. In each part of this question, one of the processes will issue a single memory operation (read or write of one byte) to a single virtual address (as indicated in each part). Your job is to figure out which physical addresses are accessed by the process if any, or determine if an error is encountered.

Entries in the first and second level tables have in their low-order bits flags denoting various access permissions.

31		2	1	0
Page Table Base Address				P

Page Directory Entry

Page Address		U	W	P

Page Table Entry

- P = 1 ⇒ Present
- W = 1 ⇒ Writable
- U = 1 ⇒ User-mode

The contents of relevant sections of memory is shown on the next page. All numbers are given in **hexadecimal**.

Address	Contents
001AC021	07693003
001AC084	00142003
0021A020	0481C001
0021A080	04A95001
0021A2FF	06128001
0021A300	05711001
0021ABFC	05176001
0021AC00	001AC001
0021B020	01FAC9DA
0021B080	052DB001
0021B2C0	0B2B36C2
0021B2FF	05A11001
0021B300	01FCF001
0021BBFC	06213001
0021BC00	001AC001
01FCF021	00382003
0481C048	0523A005
04A95048	048B8005
04A95120	07D6A005
051760F0	0E33F007
051763C0	08BF1007
052DB04A	09A62006
052DB128	0D718006
05711021	00113003
05A110F0	01133007
061280F0	0A114007
0614504A	0B183006
062133C0	052F1007

For the purposes of this problem, omitted entries have contents = 0.

Process 1 is a process in **user** mode (e.g. executing part of `main()`) and has page directory base address `0x0021A000`.
Process 2 is a process in **kernel** mode (e.g. executing a `read()` system call) and has page directory base address `0x0021B000`.

For each of the following memory accesses, first calculate and fill in the address of the page directory entry and the page table entry. Then, if the lookup is successful, give the physical address accessed. Otherwise, circle the reason for the failure and give the address of the table entry causing the failure. You may use the 32-bit breakdown table if you wish, but you are not required to fill it in.

1. Process 1 writes to `0xBFCF0145`.
   Scratch space:

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

(a) Address of PDE: 0x

(b) Address of PTE: 0x

(c) (SUCCESS) The address accessed is: 0x

OR

(FAILURE) The address of the table entry causing the failure is: 0x

The access failed because (circle one):
page not present \ page not writable \ illegal non-supervisor access

2. Process 2 writes to `0x0804A1F0`. Scratch space:

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

(a) Address of PDE: 0x

(b) Address of PTE: 0x

(c) (SUCCESS) The address accessed is: 0x

OR

(FAILURE) The address of the table entry causing the failure is: 0x

The access failed because (circle one):
page not present \ page not writable \ illegal non-supervisor access

Problem 5. (6 points):

Pink Bowler Software

A. Harry Bovik, a programmer at Pink Bowler software, has written some code that uses the following two globals and a function called tophat(). In which sections of the ELF file would they be stored? Please just write the name of the section.

   (a) `int valid;`

   (b) `int last_pid = -1;`

   (c) `int tophat(int size, double brim_width) { .... }`

B. While working one day, Harry Bovik notices that one of his scurrilous employees has made an important project print out “Bovik couldn’t make it through 213.” Bovik traces the offensive string to a `.h` file of string literals. He plans to overwrite this string in memory while the program is running. Can he do it? Why or not?

C. The following shows excerpts from three different files.

```
file0.c
     ...
     static int var;
     ...
file1.c
     ...
     int var = 0;
     double b;
     ...
file2.c
     int var;
     int foo(void) {
         int c;
         int error = printf("How’s your exam going?\n");
         return error;
     }
     ...
```

These files are compiled with the command line

```
gcc -o code file0.c file1.c file2.c
```

(a) Would the linker report an error resolving the symbol ’`var`’?

(b) If so, explain why. If not, describe how the three declarations of this symbol would be resolved. (A few sentences will suffice.)

D. In the above code, where would the string “`How’s your exam going?\n`” appear?

```
( ) .bss
( ) .rodata
( ) .text
( ) .data
```

Problem 6. (8 points):

Read over the following program:

```
int main(int argc, char *argv[]) {
    printf("PB ");
    pid_t pid;
    if (pid = fork()) {
        printf("PS ");
        printf("PR ");
        kill(pid, SIGCONT);
    } else {
        printf("CS ");
        kill(getpid(), SIGSTOP);
        printf("CR ");
    }
    return 0;
}
```

Fill in the blanks below as indicated, with the following assumptions:

- No system calls will fail.
- No other processes in the system will send signals to either the parent or the child.
- `printf()` is atomic, and will call `fflush(stdout)` after printing its argument(s) but before returning.

Write all possible outputs of this program in the following blanks: (you might not use all of them)

____________________________ ____________________________

____________________________ ____________________________

____________________________ ____________________________

____________________________ ____________________________

____________________________ ____________________________

Problem 7. (6 points):

A. Given:

```
char *foo() {
    char *copy;
    char *mystring = "Pink > Red";
    copy = malloc(strlen(mystring));
    return strcpy(copy, mystring);
}
```

Assume the call to `malloc` is successful. Is this a correct snippet of code? Why or why not?

B. Harry Bovik has written the following function. He'd like you to look it over before he commits it.

```
#define SUCCESS 0
#define ERROR (-1)
#define UNLOCKED 0

typedef struct {
    int locked;
} mutex_t;

int mutex_init(mutex_t *p) {
    if ((p = malloc(sizeof(mutex_t))) == NULL)
        return ERROR;
    p->locked = UNLOCKED;
    return SUCCESS;
}

mutex_t *new_mutex() {
    mutex_t *new = malloc(sizeof(mutex_t));
    if ((new == NULL) ||(mutex_init(new) != SUCCESS))
        return NULL;
    return new;
}
```

Describe the two most serious problems this code has. 1–2 sentences for each should suffice.

Andrew login ID:____________________________________

Full Name:____________________________________

Recitation Section:____________________________________

CS 15-213, Spring 2009
Exam 2

Tuesday, April 7th, 2009

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–J) on the front.
- **Do not write any part of your answers outside of the space given below each question. Write clearly and at a reasonable size. If we have trouble reading your handwriting you will receive no credit on that problem.**
- 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. You may use any books or notes you like. No calculators or other electronic devices are allowed.
- Good luck!

1 (20):
2 (15):
3 (15):
4 (15):
5 (10):
6 (25):
TOTAL (100):

Problem 1. (20 points):

In this problem, you will perform cache analysis for three code sequences. Assume a very small direct mapped 16 byte data cache with two cache lines. We assume a float requires 4 bytes. Drawing the cache helps.

For each code sequence, we assume a cold cache and that the array X is cache aligned (that is, $X[0]$ is loaded into the the beginning of the first cache line. All other variables are held in registers.

Recall that miss rate is defined as $\frac{\#\text{misses}}{\#\text{accesses}}$.

1. Code 1:

```
float X[8], t = 0;
for(int j = 0; j < 2; j++)
  for(int i = 0; i < 8; i++)
    t += X[i];
```

   Answer the following:

   (a) Miss rate:

   (b) What types ot types of locality does this code have with respect to this cache?

2. Code 2:

```
float X[8], t = 0;
for(int j = 0; j < 2; j++)
{
  for(int i = 0; i < 7; i += 2)
    t += X[i];
  for(int i = 1; i < 8; i += 2)
    t += X[i];
}
```

   Answer the following:

   (a) Miss rate:

   (b) What types ot types of locality does this code have with respect to this cache?

3. Code 3:

```
float X[8], float t = 0;
for(i = 0; i < 2; i++)
  for(k = 0; k < 2; k++)
    for(j = 0; j < 4; j++)
      t += X[j + i * 4];
```

Answer the following:

(a) Miss rate:

(b) What types ot types of locality does this code have with respect to this cache?

4. Changing Cache:

All three code fragments above perform the same computation. Assume you could change the code any way you want to perform the same computation and also change the cache as you wish.

(a) What is the minimum number of cache misses achievable?

(b) How would the cache look like to achieve this?

Problem 2. (15 points):

1. If a direct mapped cache is 8KB in size, and has 32 byte cache blocks, how many lines are there in each set?

   (a) 256
   (b) 64
   (c) 32
   (d) 1

2. You have a 32-bit virtual memory system with 4KB page frames, with a TLB with 4 sets, each of which is 8-way set associative. How many bits of the virtual address form the TLBi (TLB index)?

   (a) 2
   (b) 4
   (c) 8
   (d) 12

3. Which of these features in a system **best** justifies the use of a two level page table structure, as opposed to a one level page table structure?

   (a) Small page sizes
   (b) Frequent memory accesses
   (c) High degree of spatial locality in programs
   (d) Sparse memory usage patterns

4. Which section of an ELF file contains the compiled functions from a program?

   (a) .data
   (b) .rodata
   (c) .text
   (d) .bss

5. Which of the following is true?

   (a) Every signal that is sent is also received
   (b) Signals are always received immediately since they cause an interrupt
   (c) Signals can only be received after a context switch
   (d) Signals can only be received upon returning from system mode

6. Consider a theoretical computer architecture with 50-bit virtual addresses and 16kb pages. What is the maximum number of levels of page tables that could be used in the virtual memory system?

   (a) 16

   (b) 36

   (c) 2

   (d) 50

7. Which blocks the signal SIGKILL?

   (a) signal(SIGKILL, SIG_IGN);

   (b) sigfillset(&set); sigprocmask(SIG_BLOCK, set);

   (c) a and b

   (d) none of the above

8. Which of the following will reduce the number of compulsory (cold) cache misses in a program?

   (a) Increasing the associativity

   (b) Increasing line size

   (c) Both a & b

   (d) None of the above

9. Linkers can take as input which of the following file types? (circle all correct answers)

   (a) .c

   (b) .o

   (c) .a

   (d) .s

10. Blocking matrix-matrix multiplication can increase what type(s) of locality?

   (a) Temporal

   (b) Spatial

   (c) Both

11. Give two functions that don't return (two completely different functions, not variations of one).

Problem 3. (15 points):

Suppose we have the following two .c files:

alarm.c

```
int counter;

void sigalrm_handler (int num) {
  counter += 1;
}

int main (void) {
  signal(SIGALRM, &sigalrm_handler);
  counter = 2;
  alarm(1);
  sleep(1);
  counter -= 3;
  exit(counter);
  return 0;
}
```

fork.c

```
int counter;

void sigchld_handler(int num) {
  int i;
  wait(&i);
  counter += WEXITSTATUS(i);
}

int main (void) {
  signal(SIGCHLD, &sigchld_handler);
  counter = 3;
  if (!fork()) {
    counter++;
    execl("alarm", "alarm", NULL);
  }
  sleep(2);
  counter *= 3;
  printf("%d\n", counter);
  exit(0);
}
```

Assume that all system calls succeed and that all C arithmetic statements are atomic.

The files are compiled as follows:

```
gcc -o alarm alarm.c
```

```
gcc -o fork fork.c
```

Suppose we run `./fork` at the terminal. What are the possible outputs to the terminal?

Problem 4. (15 points):

Harry Q. Bovik builds and runs a C program from the following two files:

```
----------------------------------------
main.c:

#include <stdio.h>

long a = 1;
const long b = 2;
long c;
long d = -1;

int main(int argc, char *argv[]) {
    printf("a: %p\nb: %p\nc: %p\nd: %p\n", &a, &b, &c, &d);
    printf("%ld\n", c);
    return 0;
}
----------------------------------------
data.c:

unsigned int c[2] = {...};
----------------------------------------
```

And sees this output:

```
a: 0x601020
b: 0x400650
c: 0x601030
d: 0x601028
4294967297
```

Harry was expecting the variables to be in order, one after another. Obviously, he was very wrong. Help him figure out what's happening using your knowledge of linking and executable layouts. Be specific but concise with your answers.

(a) How many symbols does `main.c` generate in the executable program's symbol table?

(b) What are the strong symbols from `main.c`, and what are the weak symbols from `main.c`?

(c) Note the address of b. Why is it far removed from the addresses of the other variables?

(d) Why is c located after d in memory, even though it's before d in Harry's program?

(e) Note the output given by the final printf. Was Harry compiling and running the code on x86 or x86-64? How do you know?

(f) Given that $4294967297 = 2^{32} + 1$, what would be output by

```
printf("{%d, %d}", c[0], c[1]);
```

if it were executed in data.c?

Problem 5. (10 points):

Assume a System that has

1. A two way set associative TLB
2. A TLB with 8 total entries
3. 2^8 byte page size
4. 2^{16} bytes of virtual memory
5. one (or more) boats

TLB			
Index	Tag	Frame Number	Valid
0	0x27	0xC6	1
	0x29	0x73	1
1	0x11	0xFF	0
	0x0A	0xEC	1
2	0x29	0xCD	1
	0x3A	0xAB	1
3	0x32	0xFB	0
	0x23	0x46	0

Use the TLB to fill in the table. Strike out anything that you don't have enough information to fill in.

Virtual Address	Physical Address
0x8F0F	
	0x4690
	0x7300
0x2933	
0x2839	

Problem 6. (25 points):

Suppose a processor can support up to 64-bits of physical memory, but for binary backwards-compatibility uses 32-bit virtual addresses. You have been assigned the task of designing a virtual memory scheme for this processor.

You must meet the following requirements:

1. The scheme must be able to map any 32-bit virtual address to any 64-bit physical address.
2. The scheme must use a 16KB page size.

You must meet the following performance constraints in addition to the above requirements:

1. The scheme must minimize the number of physical memory accesses needed to resolve a virtual address (that is, the scheme must minimize the number of levels of page tables).
2. The size of a page table at any level must not exceed 512 bytes.

This is your design. You may specify anything not already defined, including the size of page tables, the size of page table entries, etc. Design your system, and then answer the following questions:

How many levels of page tables are there?

What is the size of the table at each level?

Show the mapping of the bits of a virtual address to each level. To illustrate the notation we expect, please follow the convention shown in the example below (the mapping for standard two-level Intel x86-32 virtual memory). Notice that the first set of bits maps into the top level page table, the second set maps into the second level page table, etc.

```
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                             |                             |                                   |
|           VPN 1             |           VPN 2             |              OFFSET               |
|                             |                             |                                   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Draw your mapping here:

```
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                                               |
|                                                                                               |
|                                                                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

What is the size of a page table entry at each level?