Full Name:_____

Course Name, Fall/Spring 200x

Master Exam

Date, 200x

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name on the front.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 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. Good luck!

1 (10):
2 (12):
3 (10):
4 (06):
5 (08):
6 (03):
7 (10):
8 (08):
TOTAL (67):

Problem 1. (6 points):

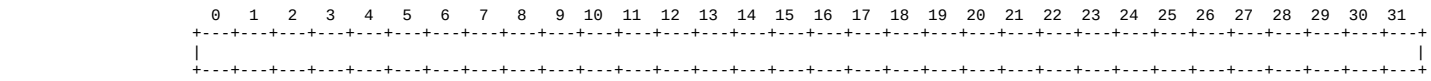
Consider the following datatype definitions on an IA32 (x86) machine.

```
typedef struct {
  char  c;
  double *p;
  int i;
  double d;
  short s;
} struct1;
```

```
typedef union {
  char  c;
  double *p;
  int i;
  double d;
  short s;
} union1;
```

A. Using the template below (allowing a maximum of 32 bytes), indicate the allocation of data for a structure of type `struct1`. Mark off and label the areas for each individual element (there are 5 of them). Cross hatch the parts that are allocated, but not used (to satisfy alignment).

Assume the alignment rules discussed in lecture: data types of size x must be aligned on x-byte boundaries. **Clearly indicate the right hand boundary of the data structure with a vertical line**.



B. How many bytes are allocated for an object of type `struct1`?

C. What alignment is required for an object of type `struct1`? (If an object must be aligned on an x-byte boundary, then your answer should be x.)

D. If we define the fields of `struct1` in a different order, we can reduce the number of bytes wasted by each variable of type `struct1`. What is the number of **unused, allocated** bytes in the best case?

E. How many bytes are allocated for an object of type `union1`?

F. What alignment is required for an object of type `union1`? (If an object must be aligned on an x-byte boundary, then your answer should be x.)

Problem 2. (12 points):

In the following questions assume the variables `a` and `b` are signed integers and that the machine uses two's complement representation. Also assume that `MAX_INT` is the maximum integer, `MIN_INT` is the minimum integer, and `W` is one less than the word length (e.g., `W` = 31 for 32-bit integers).

Match each of the descriptions on the left with a line of code on the right (write in the letter). You will be given 2 points for each correct match.

1. One's complement of `a`

_____

2. `a`.

_____

3. `a & b`.

_____

4. `a * 7`.

_____

5. `a / 4` .

_____

6. `(a < 0) ? 1 : -1` .

_____

a. `~(~a | (b ^ (MIN_INT + MAX_INT)))`

b. `((a ^ b) & ~b) | (~(a ^ b) & b)`

c. `1 + (a << 3) + ~a`

d. `(a << 4) + (a << 2) + (a << 1)`

e. `((a < 0) ? (a + 3) : a) >> 2`

f. `a ^ (MIN_INT + MAX_INT)`

g. `~((a | (~a + 1)) >> W) & 1`

h. `~((a >> W) << 1)`

i. `a >> 2`

Problem 3. (8 points):

The following procedure takes a single-precision floating point number in IEEE format and prints out information about what category of number it is. Fill in the missing code so that it performs this classification correctly.

```
void classify_float(float f)
{
   /* Unsigned value u has same bit pattern as f */
   unsigned u = *(unsigned *) &f;
   /* Split u into the different parts */
   int sign = (u >> 31) & 0x1;      // The sign bit

   int exp  = _________________;    // The exponent field

   int frac = _________________;    // The fraction field

   /* The remaining expressions can be written in terms of the
   values of sign, exp, and frac */

   if (_____________________)
     printf("Plus or minus zero\");

   else if (_____________________)
     printf("Nonzero, denormalized\");

   else if (_____________________)
     printf("Plus or minus infinity\");

   else if (_____________________)
     printf("NaN\");

   else if (_____________________)
     printf("Greater than -1.0 and less than 1.0\");

   else if (_____________________)
     printf("Less than or equal to -1.0\");
   else
     printf("Greater than or equal to 1.0\");
}
```

Problem 4. (12 points):

Consider the following 16-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next seven bits are the exponent. The exponent bias is 63.
- The last eight bits are the significand.

The rules are like those in the IEEE standard (normalized, denormalized, representation of 0, infinity, and NAN).
As described in Class 10, we consider the floating point format to encode numbers in a form:

$$(-1)^s \times m \times 2^E$$

where m is the *mantissa* and E is the exponent.
Fill in the table below for the following numbers, with the following instructions for each column:

Hex: The 4 hexadecimal digits describing the encoded form.

m**:** The fractional value of the mantissa. This should be a number of the form x or x/y, where x is an integer, and y is an integral power of 2. Examples include: 0, 67/64, and 1/256.

E**:** The integer value of the exponent.

Value: The numeric value represented. Use the notation x or $x \times 2^z$, where x and z are integers.

As an example, to represent the number $7/2$, we would have $s = 0$, $m = 7/4$, and $E = 1$. Our number would therefore have an exponent field of `0x40` (decimal value $63 + 1 = 64$) and a significand field `0xC0` (binary 11000000_2), giving a hex representation `40C0`.
You need not fill in entries marked "—".

Description	Hex	m	E	Value
-0				—
Smallest value > 1				
Largest Denormalized				
$-\infty$		—	—	—
Number with hex representation `3AA0`	—			

Problem 5. (12 points):

Consider the following 5-bit floating point representation based on the IEEE floating point format. There is a sign bit in the most significant bit. The next three bits are the exponent, with an exponent bias is 3. The last bit is the fraction. The rules are like those in the IEEE standard (normalized, denormalized, representation of 0, infinity, and NAN).

As described in Handout #1, we consider the floating point format to encode numbers in a form:

$$V \quad = \quad (-1)^s \times M \times 2^E$$

where M is the *significand* and E is the *exponent*.

Fill in missing entries in the table below with the following instructions for each column:

Description: Some unique property of this number, such as, "The largest denormalized value."

Binary: The 5 bit representation.

M**:** The value of the Mantissa written in decimal format.

E**:** The integer value of the exponent.

Value: The numeric value represented, written in decimal format.

You need not fill in entries marked "—". For the arithmetic expressions, recall that the rule with IEEE format is to round to the number nearest the exact result. Use "round-to-even" rounding.

Description	Binary	M	E	Value
Minus Zero				-0.0
Positive Infinity		—	—	$+\infty$
	`01101`			
Smallest number > 0				
One				1.0
$4.0 - 0.75$				
$2.0 + 3.0$				

Problem 6. (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 7. (20 points):

We are running programs on a machine with the following characteristics:

- Values of type `int` are 32 bits. They are represented in two's complement, and they are right shifted arithmetically. Values of type `unsigned` are 32 bits.
- Values of type `float` are represented using the 32-bit IEEE floating point format, while values of type `double` use the 64-bit IEEE floating point format.

We generate arbitrary values `x`, `y`, and `z`, and convert them to other forms as follows:

```
/* Create some arbitrary values */
int x = random();
int y = random();
int z = random();
/* Convert to other forms */
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
double   dx = (double) x;
double   dy = (double) y;
double   dz = (double) z;
```

For each of the following C expressions, you are to indicate whether or not the expression *always* yields 1. If so, circle "Y". If not, circle "N". You will be graded on each problem as follows:

- If you circle no value, you get 0 points.
- If you circle the right value, you get 2 points.
- If you circle the wrong value, you get -1 points (so don't just guess wildly).

Expression	Always True?
`(x<y) == (-x>-y)`	Y N
`((x+y)<<4) + y-x == 17*y+15*x`	Y N
`~x+~y+1 == ~(x+y)`	Y N
`ux-uy == -(y-x)`	Y N
`(x >= 0) || (x < ux)`	Y N
`((x >> 1) << 1) <= x`	Y N
`(double)(float) x == (double) x`	Y N
`dx + dy == (double) (y+x)`	Y N
`dx + dy + dz == dz + dy + dx`	Y N
`dx * dy * dz == dz * dy * dx`	Y N

Problem 8. (9 points):

Consider the following C declarations:

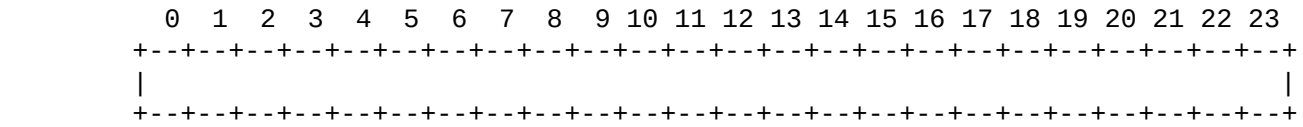
```
typedef struct {
    short code;
    long start;
    char raw[3];
    double data;
} OldSensorData;

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

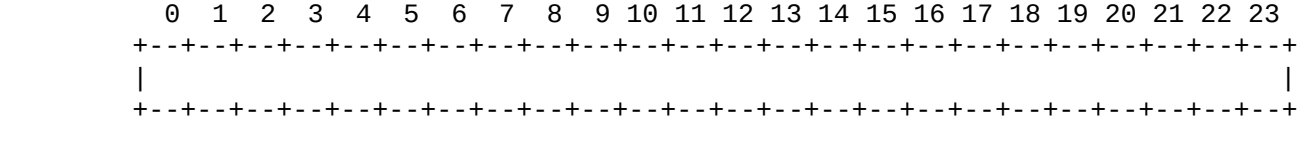
A. Using the templates below (allowing a maximum of 24 bytes), indicate the allocation of data for structs of type `OldSensorData` `NewSensorData`. Mark off and label the areas for each individual element (arrays may be labeled as a single element). **Cross hatch the parts that are allocated, but not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



`NewSensorData:`



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`_____

Problem 9. (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 10. (8 points):

Consider the following 5-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next two bits are the exponent. The exponent bias is 1.
- The last two bits are the significand.

The rules are like those in the IEEE standard (normalized, denormalized, representation of 0, ∞, and NAN). As described in Class 10, the floating point format encodes numbers in a form:

$$(-1)^s \times m \times 2^E$$

where m is the *mantissa* and E is the exponent. The table below enumerates the entire non-negative range for this 5-bit floating point representation. Fill in the blank table entries using the following directions:

E**:** The integer value of the exponent.

m**:** The fractional value of the mantissa. **Your answer must be expressed as a fraction of the form** $x/4$**.**

Value: The numeric value represented. **Your answer must be expressed as a fraction of the form** $x/4$.

You need not fill in entries marked "—".

Bits	E	m	Value
0 00 00	—	—	0
0 00 01			
0 00 10			
0 00 11			
0 01 00			
0 01 01			
0 01 10			
0 01 11			
0 10 00	1	$4/4$	$8/4$
0 10 01			
0 10 10			
0 10 11			

Problem 11. (8 points):

Consider a **5-bit** two's complement representation. Fill in the empty boxes in the following table. Addition and subtraction should be performed based on the rules for 5-bit, two's complement arithmetic

Number	Decimal Representation	Binary Representation
Zero	0	
n/a	-2	
n/a	9	
n/a	-14	
n/a		0 1100
n/a		1 0100
TMax		
TMin		
TMin+TMin		
TMin+1		
TMax+1		
$-$TMax		
$-$TMin		

Problem 12. (10 points):

Consider a **6-bit** two's complement representation. Fill in the empty boxes in the following table:

Number	Decimal Representation	Binary Representation
Zero	0	
n/a	-1	
n/a	5	
n/a	-10	
n/a		01 1010
n/a		10 0110
TMax		
TMin		
TMax+TMax		
TMin+TMin		
TMin+1		
TMin−1		
TMax+1		
−TMax		
−TMin		

Problem 13. (8 points):

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

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

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

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

```
sum_element:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax
        movl 12(%ebp),%ecx
        sall $2,%ecx
        leal 0(,%eax,8),%edx
        subl %eax,%edx
        leal (%eax,%eax,4),%eax
        movl mat2(%ecx,%eax,4),%eax
        addl mat1(%ecx,%edx,4),%eax
        movl %ebp,%esp
        popl %ebp
        ret
```

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

`M` =

`N` =

Problem 14. (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 15. (8 points):

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

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

int copy_element(int i, int j)
{
    mat1[i][j] = mat2[j][i];
}
```

This generates the following assembly code:

```
copy_element:
        pushl %ebp
        movl %esp,%ebp
        pushl %ebx
        movl 8(%ebp),%ecx
        movl 12(%ebp),%ebx
        movl %ecx,%edx
        leal (%ebx,%ebx,8),%eax
        sall $4,%edx
        sall $2,%eax
        subl %ecx,%edx
        movl mat2(%eax,%ecx,4),%eax
        sall $2,%edx
        movl %eax,mat1(%edx,%ebx,4)
        movl -4(%ebp),%ebx
        movl %ebp,%esp
        popl %ebp
        ret
```

A. What is the value of `M`:

B. What is the value of `N`:

Problem 16. (6 points):

Consider the following code fragment containing the incomplete definition of a data type `matrix_entry` with 4 fields.

```
struct matrix_entry{

  ____ a;

  ____ b;

  int  c;

  ____ d;

};

struct matrix_entry matrix[2][5];

int return_entry(int i, int j){
   return matrix[i][j].c;
}
```

Complete the above definition of `matrix_entry` so that the following assembly code could be generated from it on a Linux/x86 machine:

```
return_entry:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax
        leal (%eax,%eax,4),%eax
        addl 12(%ebp),%eax
        sall $4,%eax
        movl matrix+4(%eax),%eax
        movl %ebp,%esp
        popl %ebp
        ret
```

Notes

- Note that there are multiple correct answers.
- Choose your answers from the following types, assuming the following sizes and alignments:

Type	Size (bytes)	Alignment (bytes)
char	1	1
short	2	2
int	4	4
double	8	4

Problem 17. (8 points):

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

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

void 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),%eax
  leal 0(,%eax,4),%ebx
  leal 0(,%ecx,8),%edx
  subl %ecx,%edx
  addl %ebx,%eax
  sall $2,%eax
  movl array2(%eax,%ecx,4),%eax
  movl %eax,array1(%ebx,%edx,4)
  popl %ebx
  movl %ebp,%esp
  popl %ebp
  ret
```

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

`M` =

`N` =

Problem 18. (14 points):

Consider the source code below, used to keep track of the rooms currently reserved in a family-run hotel. Each entry in the `residents` array stores a name of the customer reserving the room. `FLOORS` represents the number of floors in the hotel. `ROOMS` represents the number of rooms per floor. Both are constants declared with `#define`. LEN, the maximum number of bytes allocated for a name, is defined to be 12.

```
char residents[FLOORS][ROOMS][LEN];

void
reserve_room(int floor, int room, char *custname)
{
        strcpy(residents[floor][room], custname);
}
```

The assembly code for the function `reserve_room` looks like this:

```
reserve_room:
        pushl %ebp
        movl %esp,%ebp
        movl 12(%ebp),%eax
        movl 16(%ebp),%edx
        pushl %edx
        movl 8(%ebp),%edx
        sall $4,%edx
        subl 8(%ebp),%edx
        leal (%eax,%eax,2),%eax
        leal residents(,%eax,4),%eax
        leal (%eax,%edx,4),%edx
        pushl %edx
        call strcpy
        movl %ebp,%esp
        popl %ebp
        ret
```

A. What is the value of `ROOMS`?

B. Due to a strange bug, the program accesses `residents[0][1][-2]`. What value is actually being accessed? (Express your answer as an *integer triplet (-,-,-).* You may assume that FLOORS and ROOMS are both greater than 1)

C. The programmer realizes that this implementation is wasteful of memory. Successive fires in several memory chip factories in Taiwan drive up memory prices and finally convince him to improve the memory efficiency of his implementation to maintain the competitiveness of the family hotel.

The declaration of `residents` is changed to be a two dimensional array of pointers to character strings (names). The new code allocates memory for customer names only for those rooms that are actually reserved. Otherwise, `residents[f][r]` stores a NULL pointer. **For simplicity, assume there is no storage overhead due to** `malloc`.

The new declaration looks like this:

```
char *residents[FLOORS][ROOMS];

void
reserve_room(int floor, int room, char *custname)
{
    residents[floor][room] = malloc(LEN);
    strcpy(residents[floor][room], custname);
}
```

After a few months. The programmer goes back to review the memory savings of his improved scheme. During that period, the hotel was 20% reserved. The programmer is delighted because the savings are found to be 168 bytes! How many floors does this hotel have? (that is, what is the value of FLOORS?)

In the following problem, you are given the task of reconstructing C code based on some declarations of C structures and unions, and the IA32 assembly code generated when compiling the C code.

Below are the data structure declarations. (Note that this is a single declaration which includes several data structures; they are shown horizontally rather than vertically simply so that they fit on one page.)

```
struct s1 {
  char a[3];
  union u1 b;
  int c;
};
```

```
struct s2 {
  struct s1 *d;
  char e;
  int f[4];
  struct s2 *g;
};
```

```
union u1 {
  struct s1 *h;
  struct s2 *i;
  char j;
};
```

You may find it helpful to diagram these data structures in the space below:

Problem 19. (12 points):

For each IA32 assembly code sequence below on the left, fill in the missing portion of corresponding C source line on the right.

A.
```
proc1:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 12(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

```
int proc1(struct s2 *x)
{
  return x->_____________________ ;
}
```

B.
```
proc2:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 4(%eax),%eax
    movl 20(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

```
int proc2(struct s1 *x)
{
  return x->_____________________ ;
}
```

C.
```
proc3:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%eax
    movsbl 4(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

```
char proc3(union u1 *x)
{
  return x->_____________________ ;
}
```

D.
```
proc4:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%eax
    movl 24(%eax),%eax
    movl (%eax),%eax
    movsbl 1(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

```
char proc4(union u1 *x)
{
  return x->_____________________ ;
}
```

Problem 20. (8 points):

Condider the following assembly code for a C `for` loop:

```
loop:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%ecx
        movl 12(%ebp),%edx
        xorl %eax,%eax
        cmpl %edx,%ecx
        jle .L4
.L6:
        decl %ecx
        incl %edx
        incl %eax
        cmpl %edx,%ecx
        jg .L6
.L4:
        incl %eax
        movl %ebp,%esp
        popl %ebp
        ret
```

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

```
int loop(int x, int y)
{
   int result;

   for (_____________; ____________; result++ ) {

      __________;

      __________;
   }

   __________;

   return result;
}
```

Problem 21. (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 22. (8 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 12(%ebp),%ecx
        xorl %eax,%eax
        movl 8(%ebp),%edx
        cmpl %ecx,%edx
        jle .L3
        .align 4
.L5:
        addl %edx,%eax
        decl %edx
        cmpl %ecx,%edx
        jg .L5
.L3:
        leave
        ret
```

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

```
int foo(int x, int y)
{
  int i, result=0;

  for (i=________; _____________________________; _________) {

    _______________;

  }

  return result;
}
```

Problem 23. (10 points):

Condider the following assembly code for a C `for` loop:

```
loop:
        pushl %ebp
        movl %esp,%ebp
        movl 0x8(%ebp),%edx
        movl %edx,%eax
        addl 0xc(%ebp),%eax
        leal 0xffffffff(%eax),%ecx
        cmpl %ecx,%edx
        jae .L4
.L6:
        movb (%edx),%al
        xorb (%ecx),%al
        movb %al,(%edx)
        xorb (%ecx),%al
        movb %al,(%ecx)
        xorb %al,(%edx)
        incl %edx
        decl %ecx
        cmpl %ecx,%edx
        jb .L6
.L4:
        movl %ebp,%esp
        popl %ebp
        ret
```

Based on the assembly code above, fill in the blanks below in its corresponding C source code. (Note: you may only use the symbolic variables `h`, `t` and `len` in your expressions below — *do not use register names.*)

```
void loop(char *h, int len)
{
   char *t;

   for (_____________; ____________; h++,t--) {

      _________________;

      _________________;

      _________________;
   }

   return;
}
```

Problem 24. (10 points):

A C function `looper` and the assembly code it compiles to on an IA-32 machine running Linux/GAS is shown below:

```
looper:
  pushl %ebp
  movl %esp,%ebp
  pushl %esi
  pushl %ebx
  movl 8(%ebp),%ebx
  movl 12(%ebp),%esi
  xorl %edx,%edx
  xorl %ecx,%ecx
  cmpl %ebx,%edx
  jge .L25
.L27:
  movl (%esi,%ecx,4),%eax
  cmpl %edx,%eax
  jle .L28
  movl %eax,%edx
.L28:
  incl %edx
  incl %ecx
  cmpl %ebx,%ecx
  jl .L27
.L25:
  movl %edx,%eax
  popl %ebx
  popl %esi
  movl %ebp,%esp
  popl %ebp
  ret
```

```
int looper(int n, int *a) {
  int i;
  int x = ________________;

  for(i = _____________;

      ___________________;

      i++) {

    if (_____________________)

       x = ___________________;


    ______________________;
  }

  return x;
}
```

Based on the assembly code, fill in the blanks in the C source code.

Notes:

- You may only use the C variable names `n`, `a`, `i` and `x`, not register names.
- Use array notation in showing accesses or updates to elements of `a`.

Problem 25. (8 points):

Consider the following IA32 code for a procedure `foo()`:

```
  foo:
      pushl %ebp
      movl %esp,%ebp
      movl 8(%ebp),%ecx
      movl 16(%ebp),%edx
      movl 12(%ebp),%eax
      decl %eax
      js .L3
 .L7:
      cmpl %edx,(%ecx,%eax,4)
      jne .L3
      decl %eax
      jns .L7
 .L3:
      movl %ebp,%esp
      popl %ebp
      ret
```

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

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

  for (i = _________; _____________________________ ; i =___________) {
     ;
  }
  return i;
}
```

Buffer overflow

The next problem concerns the following C code, excerpted from Dr. Evil's best-selling autobiography, "World Domination My Way". He calls the program *NukeJr*, his baby nuclear bomb phase.

```
/*
 * NukeJr - Dr. Evil's baby nuke
 */
#include <stdio.h>

int overflow(void);
int one = 1;

/* main - NukeJr's main routine */
int main() {
  int val = overflow();

  val += one;
  if (val != 15213)
    printf("Boom!\n");
  else
    printf("Curses! You've defused NukeJr!\n");
  _exit(0); /* syscall version of exit that doesn't need %ebp */
}

/* overflow - writes to stack buffer and returns 15213 */
int overflow() {
  char buf[4];
  int val, i=0;

  while(scanf("%x", &val) != EOF)
    buf[i++] = (char)val;
  return 15213;
}
```

Buffer overflow (cont)

Here is the corresponding machine code for NukeJr when compiled and linked on a Linux/x86 machine:

```
08048560 <main>:
 8048560:       55              pushl  %ebp
 8048561:       89 e5           movl   %esp,%ebp
 8048563:       83 ec 08        subl   $0x8,%esp
 8048566:       e8 31 00 00 00  call   804859c <overflow>
 804856b:       03 05 90 96 04  addl   0x8049690,%eax        # val += one;
 8048570:       08
 8048571:       3d 6d 3b 00 00  cmpl   $0x3b6d,%eax          # val == 15213?
 8048576:       74 0a           je     8048582 <main+0x22>
 8048578:       83 c4 f4        addl   $0xfffffff4,%esp
 804857b:       68 40 86 04 08  pushl  $0x8048640
 8048580:       eb 08           jmp    804858a <main+0x2a>
 8048582:       83 c4 f4        addl   $0xfffffff4,%esp
 8048585:       68 60 86 04 08  pushl  $0x8048660
 804858a:       e8 75 fe ff ff  call   8048404 <_init+0x44>  # call printf
 804858f:       83 c4 10        addl   $0x10,%esp
 8048592:       83 c4 f4        addl   $0xfffffff4,%esp
 8048595:       6a 00           pushl  $0x0
 8048597:       e8 b8 fe ff ff  call   8048454 <_init+0x94>  # call _exit

0804859c <overflow>:
 804859c:       55              pushl  %ebp
 804859d:       89 e5           movl   %esp,%ebp
 804859f:       83 ec 10        subl   $0x10,%esp
 80485a2:       56              pushl  %esi
 80485a3:       53              pushl  %ebx
 80485a4:       31 f6           xorl   %esi,%esi
 80485a6:       8d 5d f8        leal   0xfffffff8(%ebp),%ebx
 80485a9:       eb 0d           jmp    80485b8 <overflow+0x1c>
 80485ab:       90              nop
 80485ac:       8d 74 26 00     leal   0x0(%esi,1),%esi
 80485b0:       8a 45 f8        movb   0xfffffff8(%ebp),%al    # L1: loop start
 80485b3:       88 44 2e fc     movb   %al,0xfffffffc(%esi,%ebp,1)
 80485b7:       46              incl   %esi
 80485b8:       83 c4 f8        addl   $0xfffffff8,%esp
 80485bb:       53              pushl  %ebx
 80485bc:       68 80 86 04 08  pushl  $0x8048680
 80485c1:       e8 6e fe ff ff  call   8048434 <_init+0x74>    # call scanf
 80485c6:       83 c4 10        addl   $0x10,%esp
 80485c9:       83 f8 ff        cmpl   $0xffffffff,%eax
 80485cc:       75 e2           jne    80485b0 <overflow+0x14> # goto L1
 80485ce:       b8 6d 3b 00 00  movl   $0x3b6d,%eax
 80485d3:       8d 65 e8        leal   0xffffffe8(%ebp),%esp
 80485d6:       5b              popl   %ebx
 80485d7:       5e              popl   %esi
 80485d8:       89 ec           movl   %ebp,%esp
 80485da:       5d              popl   %ebp
 80485db:       c3              ret
```

Problem 26. (10 points):

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

- Recall that Linux/x86 machines are Little Endian.
- The `scanf("%x", &val)` function reads a whitespace-delimited sequence of characters from `stdin` that represents a hex integer, converts the sequence to a 32-bit `int`, and assigns the result to `val`. The call to `scanf` returns either 1 (if it converted a sequence) or EOF (if no more sequences on `stdin`).

  For example, calling `scanf` four time on the input string `"0 a ff"` would have the following result:

  - 1st call to `scanf`: `val=0x0` and `scanf` returns 1.
  - 2nd call to `scanf`: `val=0xa` and `scanf` returns 1.
  - 3rd call to `scanf`: `val=0xff` and `scanf` returns 1.
  - 4th call to `scanf`: `val=?` and `scanf` returns EOF.

A. After the `subl` instruction at address `0x804859f` in function `overflow` completes, the stack contains a number of objects which are shown in the table below. Determine the address of each object as a byte offset from `buf[0]`.

Stack object	Address of stack object
return address	`&buf[0] + ________`
old %ebp	`&buf[0] + ________`
buf[3]	`&buf[0] + ________`
buf[2]	`&buf[0] + ________`
buf[1]	`&buf[0] + 1`
buf[0]	`&buf[0] + 0`

B. What input string would defuse NukeJr by causing the call to `overflow` to return to address `0x8048571` instead of `804856b`? *Notes: (i) Your solution is allowed to trash the contents of the* `%ebp` *register. (ii) Each underscore is a one or two digit hex number.*

`Answer: "0 0 0 0 ___ ___ ___ ___ ___ ___ ___ ___ "`

Problem 27. (7 points):

Match each of the assembler routines on the left with the equivalent C function on the right.

```
foo1:
     pushl %ebp
     movl %esp,%ebp
     movl 8(%ebp),%eax
     sall $4,%eax
     subl 8(%ebp),%eax
     movl %ebp,%esp
     popl %ebp
     ret

foo2:
     pushl %ebp
     movl %esp,%ebp
     movl 8(%ebp),%eax
     testl %eax,%eax
     jge .L4
     addl $15,%eax
.L4:
     sarl $4,%eax
     movl %ebp,%esp
     popl %ebp
     ret

foo3:
     pushl %ebp
     movl %esp,%ebp
     movl 8(%ebp),%eax
     shrl $31,%eax
     movl %ebp,%esp
     popl %ebp
     ret
```

```
int choice1(int x)
{
    return (x < 0);
}

int choice2(int x)
{
    return (x << 31) & 1;
}

int choice3(int x)
{
    return 15 * x;
}

int choice4(int x)
{
    return (x + 15) /4
}

int choice5(int x)
{
    return x / 16;
}

int choice6(int x)
{
    return (x >> 31);
}
```

Fill in your answers here:

foo1 corresponds to choice _____.

foo2 corresponds to choice _____.

foo3 corresponds to choice _____.

Problem 28. (3 points):

Consider the following C functions and assembly code:

```
int fun1(int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}

int fun2(int a, int b)
{
    if (b < a)
        return b;
    else
        return a;
}

int fun3(int a, int b)
{
    unsigned ua = (unsigned) a;
    if (ua < b)
        return b;
    else
        return ua;
}
```

```
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%edx
  movl 12(%ebp),%eax
  cmpl %eax,%edx
  jge .L9
  movl %edx,%eax
.L9:
  movl %ebp,%esp
  popl %ebp
  ret
```

Which of the functions compiled into the assembly code shown?

Problem 29. (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 30. (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?

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 31. (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____________________
```

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 32. (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] ->
--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
```

Problem 33. (4 points):

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

```
0x400446e3 <malloc+7>: call   0x400446e8 <malloc+12>
0x400446e8 <malloc+12>: popl   %eax
```

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

This next problem will test your understanding of stack frames. It is based on the following recursive C function:

```
int silly(int n, int *p)
{
    int val, val2;

    if (n > 0)
        val2 = silly(n << 1, &val);
    else
        val = val2 = 0;

    *p = val + val2 + n;

    return val + val2;
}
```

This yields the following machine code:

```
silly:
        pushl %ebp
        movl %esp,%ebp
        subl $20,%esp
        pushl %ebx
        movl 8(%ebp),%ebx
        testl %ebx,%ebx
        jle .L3
        addl $-8,%esp
        leal -4(%ebp),%eax
        pushl %eax
        leal (%ebx,%ebx),%eax
        pushl %eax
        call silly
        jmp .L4
        .p2align 4,,7
.L3:
        xorl %eax,%eax
        movl %eax,-4(%ebp)
.L4:
        movl -4(%ebp),%edx
        addl %eax,%edx
        movl 12(%ebp),%eax
        addl %edx,%ebx
        movl %ebx,(%eax)
        movl -24(%ebp),%ebx
        movl %edx,%eax
        movl %ebp,%esp
        popl %ebp
        ret
```

Problem 34. (6 points):

A. Is the variable `val` stored on the stack? If so, at what byte offset (relative to `%ebp`) is it stored, and why is it necessary to store it on the stack?

B. Is the variable `val2` stored on the stack? If so, at what byte offset (relative to `%ebp`) is it stored, and why is it necessary to store it on the stack?

C. What (if anything) is stored at `-24(%ebp)`? If something is stored there, why is it necessary to store it?

D. What (if anything) is stored at `-8(%ebp)`? If something is stored there, why is it necessary to store it?

The following problem concerns the following, low-quality code:

```
void foo(int x)
{
  int a[3];
  char buf[4];
  a[0] = 0xF0F1F2F3;
  a[1] = x;
  gets(buf);
  printf("a[0] = 0x%x, a[1] = 0x%x, buf = %s\n", a[0], a[1], buf);
}
```

In a program containing this code, procedure `foo` has the following disassembled form on an IA32 machine:

```
 080485d0 <foo>:
 80485d0: 55              pushl  %ebp
 80485d1: 89 e5           movl   %esp,%ebp
 80485d3: 83 ec 10        subl   $0x10,%esp
 80485d6: 53              pushl  %ebx
 80485d7: 8b 45 08        movl   0x8(%ebp),%eax
 80485da: c7 45 f4 f3 f2  movl   $0xf0f1f2f3,0xfffffff4(%ebp)
 80485df: f1 f0
 80485e1: 89 45 f8        movl   %eax,0xfffffff8(%ebp)
 80485e4: 8d 5d f0        leal   0xfffffff0(%ebp),%ebx
 80485e7: 53              pushl  %ebx
 80485e8: e8 b7 fe ff ff  call   80484a4 <_init+0x54>  # gets
 80485ed: 53              pushl  %ebx
 80485ee: 8b 45 f8        movl   0xfffffff8(%ebp),%eax
 80485f1: 50              pushl  %eax
 80485f2: 8b 45 f4        movl   0xfffffff4(%ebp),%eax
 80485f5: 50              pushl  %eax
 80485f6: 68 ec 90 04 08  pushl  $0x80490ec
 80485fb: e8 94 fe ff ff  call   8048494 <_init+0x44>  # printf
 8048600: 8b 5d ec        movl   0xffffffec(%ebp),%ebx
 8048603: 89 ec           movl   %ebp,%esp
 8048605: 5d              popl   %ebp
 8048606: c3              ret
 8048607: 90              nop
```

For the following questions, recall that:

- `gets` is a standard C library routine.
- IA32 machines are little-endian.
- C strings are null-terminated (i.e., terminated by a character with value 0x00).
- Characters '`0`' through '`9`' have ASCII codes `0x30` through `0x39`.

Problem 35. (6 points):

Fill in the following table indicating where on the stack the following program values are located. Express these as decimal offsets (positive or negative) relative to register `%ebp`:

Program Value	Decimal Offset
`a`	
`a[2]`	
`x`	
`buf`	
`buf[3]`	
Saved value of register `%ebx`	

The following problem concerns the following, low-quality code:

```
void foo(int x)
{
  int a[3];
  char buf[4];
  a[0] = 0xF0F1F2F3;
  a[1] = x;
  gets(buf);
  printf("a[0] = 0x%x, a[1] = 0x%x, buf = %s\n", a[0], a[1], buf);
}
```

In a program containing this code, procedure `foo` has the following disassembled form on an IA32 machine:

```
 080485d0 <foo>:
 80485d0: 55              pushl  %ebp
 80485d1: 89 e5           movl   %esp,%ebp
 80485d3: 83 ec 10        subl   $0x10,%esp
 80485d6: 53              pushl  %ebx
 80485d7: 8b 45 08        movl   0x8(%ebp),%eax
 80485da: c7 45 f4 f3 f2  movl   $0xf0f1f2f3,0xfffffff4(%ebp)
 80485df: f1 f0
 80485e1: 89 45 f8        movl   %eax,0xfffffff8(%ebp)
 80485e4: 8d 5d f0        leal   0xfffffff0(%ebp),%ebx
 80485e7: 53              pushl  %ebx
 80485e8: e8 b7 fe ff ff  call   80484a4 <_init+0x54>  # gets
 80485ed: 53              pushl  %ebx
 80485ee: 8b 45 f8        movl   0xfffffff8(%ebp),%eax
 80485f1: 50              pushl  %eax
 80485f2: 8b 45 f4        movl   0xfffffff4(%ebp),%eax
 80485f5: 50              pushl  %eax
 80485f6: 68 ec 90 04 08  pushl  $0x80490ec
 80485fb: e8 94 fe ff ff  call   8048494 <_init+0x44>  # printf
 8048600: 8b 5d ec        movl   0xffffffec(%ebp),%ebx
 8048603: 89 ec           movl   %ebp,%esp
 8048605: 5d              popl   %ebp
 8048606: c3              ret
 8048607: 90              nop
```

For the following questions, recall that:

- `gets` is a standard C library routine.
- IA32 machines are little-endian.
- C strings are null-terminated (i.e., terminated by a character with value 0x00).
- Characters '`0`' through '`9`' have ASCII codes `0x30` through `0x39`.

Problem 36. (8 points):

Consider the case where procedure `foo` is called with argument `x` equal to `0xE3E2E1E0`, and we type "`123456789`" in response to `gets`.

A. Fill in the following table indicating which program values are/are not corrupted by the response from `gets`, i.e., their values were altered by some action within the call to `gets`.

Program Value	Corrupted? (Y/N)
`a[0]`	
`a[1]`	
`a[2]`	
`x`	
Saved value of register `%ebp`	
Saved value of register `%ebx`	

B. What will the `printf` function print for the following:

- `a[0]` (hexadecimal): _____
- `a[1]` (hexadecimal): _____
- `buf` (ASCII): _____

Problem 37. (10 points):

Consider the following C declaration:

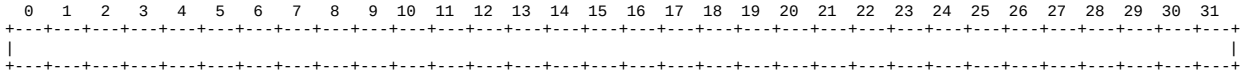
```
struct Node{
   char c;
   double value;
   struct Node* next;
   int flag;
   struct Node* left;
   struct Node* right;
};

typedef struct Node* pNode;

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

A. Using the template below (allowing a maximum of 32 bytes), indicate the allocation of data for a `Node` struct. Mark off and label the areas for each individual element (there are 6 of them). Cross hatch the parts that are allocated, but not used (to satisfy alignment).

Assume the Linux alignment rules discussed in Class 9. **Clearly indicate the right hand boundary of the data structure with a vertical line**.



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

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

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

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

1.  _______ NodeTree[i]->flag

2.  _______ NodeTree[i]->left->left->c

3.  _______ NodeTree[i]->next->next->flag

4.  _______ NodeTree[i]->right->left->left
-----------------------------------------------------------------

Linux/IA32 Assembly:

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

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

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

Problem 38. (10 points):

Consider the following incomplete definition of a C struct along with the incomplete code for a function `func` given below.

```
typedef struct node {

  _________________ x;

  _________________ y;

  struct node *next;

  struct node *prev;

} node_t;
```

```
node_t n;

void func() {

  node_t *m;

  m = ____________________________;

  m->y /= 16;

  return;
}
```

When this C code was compiled on an IA-32 machine running Linux, the following assembly code was generated for function `func`.

```
func:
  pushl %ebp
  movl n+12,%eax
  movl 16(%eax),%eax
  movl %esp,%ebp
  movl %ebp,%esp
  shrw $4,8(%eax)
  popl %ebp
  ret
```

Given these code fragments, fill in the blanks in the C code given above. Note that there is a unique answer.

The types must be chosen from the following table, assuming the sizes and alignment given.

Type	Size (bytes)	Alignment (bytes)
char	1	1
short	2	2
unsigned short	2	2
int	4	4
unsigned int	4	4
double	8	4

Performance Optimization

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

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

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

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

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

  return result;
}
```

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

Problem 39. (8 points):

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

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

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

  return result;
}
```

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

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

B. Show how to fix `fact_u2` so that its behavior is identical to `fact`. [Hint: there is a special trick for this procedure that involves modifying just a single character.]

C. Benchmarking `fact_u2` shows no improvement in performance. How would you explain that?

D. You modify the line inside the loop to read:

```
result = result * (i * (i-1));
```

To everyone's astonishment, the measured performance now has a CPE of 2.5. How do you explain this performance improvement?

Problem 40. (9 points):

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

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

Consider the following two procedures:

Loop 1

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

Loop 2

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

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

Loop 1

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

Loop 2

```
.L27:
   imull (%esi,%edx,4),%eax
   incl %edx
   cmpl %ebx,%edx
   jl .L27
```

Running on one of the Fish machines, we find that Loop 1 requires 3.0 clock cycles per iteration, while Loop 2 requires 4.0.

A. Explain how it is that Loop 1 is faster than Loop 2, even though it has one more instruction

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

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

Problem 41. (10 points):

Consider the following function for computing the product of an array of n integers. We have unrolled the loop by a factor of 3.

```
int aprod(int a[], int n)
{
    int i, x, y, z;
    int r = 1;
    for (i = 0; i < n-2; i+= 3) {
        x = a[i]; y = a[i+1]; z = a[i+2];
        r = r * 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 5 different associations of the computation, as follows:

```
        r = ((r * x) * y) * z; // A1
        r = (r * (x * y)) * z; // A2
        r = r * ((x * y) * z); // A3
        r = r * (x * (y * z)); // A4
        r = (r * x) * (y * z); // A5
```

We express the performance of the function in terms of the number of cycles per element (CPE). As described in the book, this measure assumes 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.

We measured the 5 versions of the function on an Intel Pentium III. Recall that the integer multiplication operation on this machine has a latency of 4 cycles and an issue time of 1 cycle.

(continued)

The following table shows some values of the CPE, and other values missing. The measured CPE values are those that were actually observed. "Theoretical CPE" means that performance that would be achieved if the only limiting factor were the latency and issue time of the integer multiplier.

Version	Measured CPE	Theoretical CPE
A1	4.00	
A2	2.67	
A3		$4/3 = 1.33$
A4	1.67	
A5		$8/3 = 2.67$

Fill in the missing entries. For the missing values of the measured CPE, you can use the values from other versions that would have the same computational behavior. For the values of the theoretical CPE, you can determine the number of cycles that would be required for an iteration considering only the latency and issue time of the multiplier, and then divide by 3.

Problem 42. (5 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 2-way set associative, with a 4 byte line size and 16 total lines.

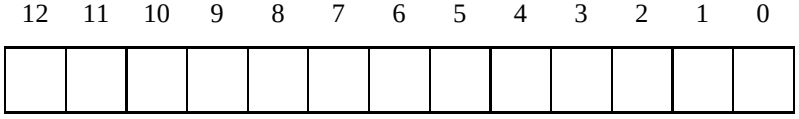
In the following tables, **all numbers are given in hexadecimal**. The contents of the cache are as follows:

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	09	1	86	30	3F	10	00	0	99	04	03	48
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	2F	81	FD	09	0B	0	8F	E2	05	BD
3	06	0	3D	94	9B	F7	32	1	12	08	7B	AD
4	C7	1	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

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



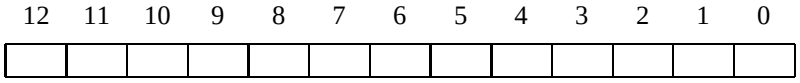
Part 2

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: `0E34`

A. Physical address format (one bit per box)



B. Physical memory reference

Parameter	Value
Byte offset	0x
Cache Index	0x
Cache Tag	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Problem 43. (5 points):

The following problem concerns basic cache lookups.

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

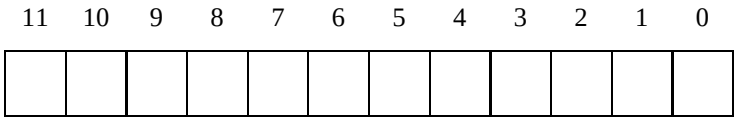
In the following tables, **all numbers are given in hexadecimal**. The contents of the cache are as follows:

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

Part 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



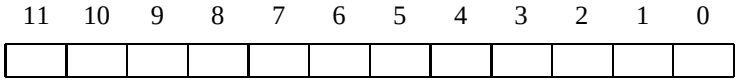
Part 2

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: `3B6`

A. Physical address format (one bit per box)



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 44. (8 points):

You are writing a new 3D game that you hope will earn you fame and fortune. You are currently working on a function to blank the screen buffer before drawing the next frame. The screen you are working with is a 640x480 array of pixels. The machine you are working on has a 64 KB direct mapped cache with 4 byte lines. The C structures you are using are:

```
struct pixel {
    char r;
    char g;
    char b;
    char a;
};

struct pixel buffer[480][640];
register int i, j;
register char *cptr;
register int *iptr;
```

Assume:

- `sizeof(char) = 1`
- `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`. Variables `i`, `j`, `cptr`, and `iptr` are stored in registers.

A. What percentage of the writes in the following code will miss in the cache?

```
for (j=0; j < 640; j++) {
    for (i=0; i < 480; i++){
        buffer[i][j].r = 0;
        buffer[i][j].g = 0;
        buffer[i][j].b = 0;
        buffer[i][j].a = 0;
    }
}
```

Miss rate for writes to `buffer`: _____ %

B. What percentage of the writes in the following code will miss in the cache?

```
char *cptr;
cptr = (char *) buffer;
for (; cptr < (((char *) buffer) + 640 * 480 * 4); cptr++)
    *cptr = 0;
```

Miss rate for writes to `buffer`: _____ %

C. What percentage of the writes in the following code will miss in the cache?

```
int *iptr;
iptr = (int *) buffer;
for (; iptr < (buffer + 640 * 480); iptr++)
    *iptr = 0;
```

Miss rate for writes to `buffer`: _____ %

D. Which code (A, B, or C) should be the fastest? _____

Problem 45. (6 points):

The following table gives the parameters for a number of different caches, where m is the number of physical address bits, C is the cache size (number of data bytes), B is the block size in bytes, and E is the number of lines per set. For each cache, determine the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b).

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	4				
2.	32	1024	4	256				
3.	32	1024	8	1				
4.	32	1024	8	128				
5.	32	1024	32	1				
6.	32	1024	32	4				

Problem 46. (7 points):

After watching the presidential election you decide to start a business in developing software for electronic voting. The software will run on a machine with a 1024-byte direct-mapped data cache with 64 byte blocks.
You are implementing a prototype of your software that assumes that there are 7 candidates. The C-structures you are using are:

```
struct vote {
    int candidates[7];
    int valid;
};

struct vote vote_array[16][16];
register int i, j, k;
```

You have to decide between two alternative implementations of the routine that initializes the array `vote_array`. You want to choose the one with the better cache performance.
You can assume:

- `sizeof(int) = 4`
- `vote_array` begins at memory address 0
- The cache is initially empty.
- The only memory accesses are to the entries of the array `vote_array`. Variables `i`, `j` and `k` are stored in registers.

A. What percentage of the writes in the following code will miss in the cache?

```
for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
        vote_array[i][j].valid=0;
    }
}

for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
        for (k=0; k<7; k++) {
            vote_array[i][j].candidates[k] = 0;
        }
    }
}
```

Total number of misses in the first loop: _____ %

Total number of misses in the second loop: _____ %

Overall miss rate for writes to `vote_array`: _____ %

B. What percentage of the writes in the following code will miss in the cache?

```
for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
        for (k=0; k<7; k++) {
            vote_array[i][j].candidates[k] = 0;
        }
        vote_array[i][j].valid=0;
    }
}
```

Miss rate for writes to `vote_array`: _____ %

Problem 47. (8 points):

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

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

You are given the following definitions

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

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

Also assume that

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

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

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

Miss rate for writes to pixel:_____%

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

Miss rate for writes to pixel: _____ %

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

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

Miss rate for writes to pixel:_____%

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

Miss rate for writes to pixel:_____%

Problem 48. (12 points):

3M decides to make Post-Its by printing yellow squares on white pieces of paper. As part of the printing process, they need to set the CMYK (cyan, magenta, yellow, black) value for every point in the square. 3M hires you to determine the efficiency of the following algorithms on a machine with a 2048-byte direct-mapped data cache with 32 byte blocks.

You are given the following definitions:

```
struct point_color {
    int c;
    int m;
    int y;
    int k;
};

struct point_color square[16][16];
register int i, j;
```

Assume:

- `sizeof(int) = 4`
- `square` begins at memory address 0
- The cache is initially empty.
- The only memory accesses are to the entries of the array `square`. Variables `i` and `j` are stored in registers.

A. What percentage of the writes in the following code will miss in the cache?

```
for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
        square[i][j].c = 0;
        square[i][j].m = 0;
        square[i][j].y = 1;
        square[i][j].k = 0;
    }
}
```

Miss rate for writes to `square`: _____ %

B. What percentage of the writes in the following code will miss in the cache?

```
for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
        square[j][i].c = 0;
        square[j][i].m = 0;
        square[j][i].y = 1;
        square[j][i].k = 0;
    }
}
```

Miss rate for writes to `square`: _____ %

C. What percentage of the writes in the following code will miss in the cache?

```
for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
        square[i][j].y = 1;
    }
}
for (i=0; i<16; i++) {
    for (j=0; j<16; j++) {
        square[i][j].c = 0;
        square[i][j].m = 0;
        square[i][j].k = 0;
    }
}
```

Miss rate for writes to `square`: _____ %

Problem 49. (10 points):

After a stressful semester you suddenly realize that you haven't bought a single christmas present yet. Fortunately, you see that one of the big electronic stores has CD's on sale. You don't have much time to decide which CD will make the best presents for which friend, so you decide to automatize the decision process. For that, you use a database containing an entry for each of your friends. It is is implemented as an 8×8 matrix using a data structure `person`. You add to this data structure a field for each CD that you consider:

```
struct person{

  char name[16];

  int age;
  int male;

  short nsync;
  short britney_spears;
  short dolly_parton;
  short garth_brooks;
}

struct person db[8][8];
register int i, j;
```

Part 1

After thinking for a while you come up with the following smart routine that finds the ideal present for everyone.

```
void generate_presents(){

  for (j=0; j<8; j++){
      for (i=0; i<8; i++) {
          db[i][j].nsync=0;
          db[i][j].britney_spears=0;
          db[i][j].garth_brooks=0;
          db[i][j].dolly_parton=0;
      }
   }

  for (j=0; j<8; j++){
      for (i=0; i<8; i++) {

          if(db[i][j].age < 30){
             if(db[i][j].male)
                   db[i][j].britney_spears = 1;
             else  db[i][j].nsync = 1;
          }
          else{
             if(db[i][j].male)
                   db[i][j].dolly_parton =1;
             else  db[i][j].garth_brooks = 1;
          }
      }
   }

}
```

Of course, runtime is important in this time-critical application, so you decide to analyze the cache performance of your routine. You assume that

- your machine has a 512-byte direct-mapped data cache with 64 byte blocks.
- `db` begins at memory address 0
- The cache is initially empty.
- The only memory accesses are to the entries of the array `db`. Variables `i`, and `j` are stored in registers.

Answer the following questions:

A. What is the total number of read and write accesses? _____.

B. What is the total number of read and write accesses that miss in the cache? _____ .

C. So the fraction of all accesses that miss in the cache is: _____.

Part 2

Then you consider the following alternative implementation of the same algorithm:

```
void generate_presents(){

  for (i=0; i<8; i++){
      for (j=0; j<8; j++) {

          if(db[i][j].age < 30)
             if(db[i][j].male) {
                db[i][j].nsync=0;
                db[i][j].britney_spears=1;
                db[i][j].garth_brooks=0;
                db[i][j].dolly_parton=0;
             }

             else
                db[i][j].nsync=1;
                db[i][j].britney_spears=0;
                db[i][j].garth_brooks=0;
                db[i][j].dolly_parton=0;
             }
          }

          else{
             if(db[i][j].male) {
                db[i][j].nsync=0;
                db[i][j].britney_spears=0;
                db[i][j].garth_brooks=0;
                db[i][j].dolly_parton=1;
             }

             else{
                db[i][j].nsync=0;
                db[i][j].britney_spears=0;
                db[i][j].garth_brooks=1;
                db[i][j].dolly_parton=0;
             }
          }


      }
   }

}
```

Making the same assumptions as in Part 1, answer the following questions.

A. What is the total number of read and write accesses? _____

B. What is the total number of read and write accesses that miss in the cache? _____

C. So the fraction of all accesses that miss in the cache is: _____.

Problem 50. (14 points):

Consider a direct mapped cache of size 64K with block size of 16 bytes. Furthermore, the cache is write-back and write-allocate. You will calculate the miss rate for the following code using this cache. Remember that `sizeof(int) == 4`. Assume that the cache starts empty and that local variables and computations take place completely within the registers and do not spill onto the stack.

A. Now consider the following code to copy one matrix to another. Assume that the `src` matrix starts at address 0 and that the `dest` matrix follows immediately follows it.

```
void copy_matrix(int dest[ROWS][COLS], int src[ROWS][COLS])
{
    int i, j;

    for (i=0; i<ROWS; i++) {
        for (j=0; j<COLS; j++) {
            dest[i][j] = src[i][j];
        }
    }
}
```

1. What is the cache miss rate if `ROWS = 128` and `COLS = 128`?
   Miss rate = _____%
2. What is the cache miss rate if `ROWS = 128` and `COLS = 192`?
   Miss rate = _____%
3. What is the cache miss rate if `ROWS = 128` and `COLS = 256`?
   Miss rate = _____%

B. Now consider the following two implementations of a horizontal flip and copy of the matrix. Again assume that the `src` matrix starts at address 0 and that the `dest` matrix follows immediately follows it.

```
void copy_n_flip_matrix1(int dest[ROWS][COLS], int src[ROWS][COLS])
{
    int i, j;

    for (i=0; i<ROWS; i++) {
        for (j=0; j<COLS; j++) {
            dest[i][COLS - 1 - j] = src[i][j];
        }
    }
}
```

1. What is the cache miss rate if `ROWS = 128` and `COLS = 128`?
   Miss rate = _____%
2. What is the cache miss rate if `ROWS = 128` and `COLS = 192`?
   Miss rate = _____%

```
void copy_n_flip_matrix2(int dest[ROWS][COLS], int src[ROWS][COLS])
{
    int i, j;

    for (j=0; j<COLS; j++) {
        for (i=0; i<ROWS; i++) {
            dest[i][COLS - 1 - j] = src[i][j];
        }
    }
}
```

1. What is the cache miss rate if `ROWS = 128` and `COLS = 128`?
   Miss rate = _____%
2. What is the cache miss rate if `ROWS = 192` and `COLS = 128`?
   Miss rate = _____%

Problem 51. (12 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 (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
      dst[j][i] = src[i][j];
    }
  }
}
```

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.

`dst` array		
	col 0	col 1
row 0	m	
row 1		

`src` 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.

`dst` array		
	col 0	col 1
row 0	m	
row 1		

`src` array		
	col 0	col 1
row 0	m	
row 1		

Problem 52. (8 points):

This problem tests your understanding of conflict misses. Consider the following transpose routine

```
typedef int array[2][2];

void transpose(array dst, array src) {
  int i, j;

  for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
      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.

`dst` array		
	col 0	col 1
row 0	m	
row 1		

`src` 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.

`dst` array		
	col 0	col 1
row 0	m	
row 1		

`src` array		
	col 0	col 1
row 0	m	
row 1		

Problem 53. (5 points):

Consider the C program below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
main() {

  if (fork() == 0) {
    if (fork() == 0) {
      printf("3");
    }
    else {
      pid_t pid; int status;
      if ((pid = wait(&status)) > 0) {
        printf("4");
      }
    }
  }
  else {
    if (fork() == 0) {
      printf("1");
      exit(0);
    }
    printf("2");
  }

  printf("0");

  return 0;
}
```

Out of the 5 outputs listed below, circle only the valid outputs of this program. Assume that all processes run to normal completion.

A. 2030401 B. 1234000 C. 2300140

D. 2034012 E. 3200410

Problem 54. (4 points):

Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
pid_t pid;

void handler1(int sig) {
  printf("zip");
  fflush(stdout);   /* Flushes the printed string to stdout */
  kill(pid, SIGUSR1);
}

void handler2(int sig) {
  printf("zap");
  exit(0);
}

main() {
  signal(SIGUSR1, handler1);
  if ((pid = fork()) == 0) {
    signal(SIGUSR1, handler2);
    kill(getppid(), SIGUSR1);
    while(1) {};
  }
  else {
    pid_t p; int status;
    if ((p = wait(&status)) > 0) {
      printf("zoom");
    }
  }
}
```

What is the output string that this program prints?

_____

Problem 55. (10 points):

Consider the C program below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main () {
    if (fork() == 0) {
        if (fork() == 0) {
            printf("3");
        }
        else {
            pid_t pid; int status;
            if ((pid = wait(&status)) > 0) {
                printf("4");
            }
        }
    }
    else {
        printf("2");
        exit(0);
    }
    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. `32040` Y N

B. `34002` Y N

C. `30402` Y N

D. `23040` Y N

E. `40302` Y N

Problem 56. (5 points):

Consider the following C program:

```
#include <sys/wait.h>

main() {
  int status;

  printf("%s\n", "Hello");
  printf("%d\n", !fork());

  if(wait(&status) != -1)
    printf("%d\n", WEXITSTATUS(status));

  printf("%s\n", "Bye");

  exit(2);
}
```

Recall the following:

- Function `fork` returns 0 to the child process and the child's process Id to the parent.
- Function `wait` returns `-1` when there is an error, e.g., when the executing process has no child.
- Macro `WEXITSTATUS` extracts the exit status of the terminating process.

What is a valid output of this program? *Hint: there are several correct solutions.*

Problem 57. (8 points):

This problem tests your understanding of exceptional control flow in C programs.
For problems A-C, indicate how many "hello" output lines the program would print.
Caution: Don't overlook the `printf` *function in* `main`.

Problem A

```
void doit() {
  fork();
  fork();
  printf("hello\n");
  return;
}
```

Answer: _____ output lines.

```
int main() {
  doit();
  printf("hello\n");
  exit(0);
}
```

Problem B

```
void doit() {
  if (fork() == 0) {
    fork();
    printf("hello\n");
    exit(0);
  }
  return;
}
```

Answer: _____ output lines.

```
int main() {
  doit();
  printf("hello\n");
  exit(0);
}
```

Problem C

```
void doit() {
  if (fork() == 0) {
    fork();
    printf("hello\n");
    return;
  }
  return;
}
```

Answer: _____ output lines.

```
int main() {
  doit();
  printf("hello\n");
  exit(0);
}
```

For problem E, indicate the value of the `counter` variable that the program would print.

Problem D

```
int counter = 1;

int main() {

  if (fork() == 0) {
    counter--;
    exit(0);
  }
  else {
    wait(NULL);
    counter++;
    printf("counter = %d\n", counter);
  }
  exit(0);
}
```

Answer: `counter = _____`.

Problem 58. (16 points):

This problem tests your understanding of exceptional control flow in C programs. Assume we are running code on a Unix machine. The following problems all concern the value of the variable `counter`.

Part I (6 points)

```
int counter = 0;

int  main()
{
    int i;

    for (i = 0; i < 2; i ++){
        fork();
        counter ++;
        printf("counter = %d\n", counter);
    }

    printf("counter = %d\n", counter);
    return 0;
}
```

A. How many times would the value of `counter` be printed: _____

B. What is the value of `counter` printed in the first line? _____

C. What is the value of `counter` printed in the last line? _____

Part II (6 points)

```
pid_t pid;
int counter = 0;

void handler1(int sig)
{
    counter ++;
    printf("counter = %d\n", counter);
    fflush(stdout);   /* Flushes the printed string to stdout */
    kill(pid, SIGUSR1);
}

void handler2(int sig)
{
    counter += 3;
    printf("counter = %d\n", counter);
    exit(0);
}

main() {
    signal(SIGUSR1, handler1);
    if ((pid = fork()) == 0) {
        signal(SIGUSR1, handler2);
        kill(getppid(), SIGUSR1);
        while(1) {};
    }
    else {
        pid_t p; int status;
        if ((p = wait(&status)) > 0) {
            counter += 2;
            printf("counter = %d\n", counter);
        }
    }
}
```

What is the output of this program?

Part III (4 points)

```
int counter = 0;

void handler(int sig)
{
    counter ++;
}


int  main()
{
    int i;

    signal(SIGCHLD, handler);

    for (i = 0; i < 5; i ++){
        if (fork() == 0){
            exit(0);
        }
    }

    /* wait for all children to die */
    while (wait(NULL) != -1);

    printf("counter = %d\n", counter);
    return 0;
}
```

A. Does the program output the same value of `counter` every time we run it? `Yes` `No`

B. If the answer to A is `Yes`, indicate the value of the `counter` variable. Otherwise, list all possible values of the `counter` variable.

Answer: `counter` = _____

Problem 59. (4 points):

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 = 10;

void handler(sig)
{
    val += 5;
    return;
}

int main()
{
    int pid;

    signal(SIGCHLD, handler);
    if ((pid = fork()) == 0) {
        val -= 3;
        exit(0);
    }
    waitpid(pid, NULL, 0);
    printf("val = %d\n", val);
    exit(0);
}
```

What is the output of this program? `val =` _____

Process control

The next problem concerns the following four versions of the `tfgets` routine, a timeout version of the Unix `fgets` routine.

The `tfgets` routine waits for the user to type in a string and hit the return key. If the user enters the string within 5 seconds, the `tfgets` returns normally with a pointer to the string. Otherwise, the routine "times out" and returns a NULL string.

`tfgets`: Version A

```
void handler(int sig) {
  siglongjmp(env, 1);
}

char *tfgets(char *s, int size, FILE *stream) {
  pid_t pid;
  signal(SIGCHLD, handler);

  if (!sigsetjmp(env, 1)) {
    pid = fork();
    if (pid == 0) {
      return fgets(s, size, stream);
    }
    else {
      sleep(5);
      kill(pid, SIGKILL);
      wait(NULL);
      return NULL;
    }
  }
  else {
    wait(NULL);
    exit(0);
  }
}
```

`tfgets`: **Version B**

```
void handler(int sig) {
  wait(NULL);
  siglongjmp(env,1);
}

char *tfgets(char *s, int size, FILE *stream) {
  pid_t pid;

  signal(SIGUSR2, handler);
  if (sigsetjmp(env, 1) != 0)
    return NULL;
  if ((pid = fork()) == 0) {
    sleep(5);
    kill(getppid(), SIGUSR2);
    exit(0);
  }
  fgets(s, size, stream);
  kill(pid, SIGKILL);
  wait(NULL);
  return s;
}
```

`tfgets`: **Version C**

```
void handler(int sig) {
  wait(NULL);
  siglongjmp(env, 1);
}

char *
tfgets(char *s, int size, FILE *stream) {
  pid_t pid;
  str = NULL;
  signal(SIGCHLD, handler);

  if ((pid = fork()) ==  0) {
    sleep(5);
    exit(0);
  }
  else {
    if (sigsetjmp(env, 1) == 0) {
      str = fgets(s, size, stream);
      kill(pid, SIGKILL);
      pause();
    }
    return str;
  }
}
```

`tfgets`: Version D

```
void handler(int sig) {
  wait(NULL);
  siglongjmp(env, 1);
}

char *
tfgets(char *s, int size, FILE *stream) {
  pid_t pid;
  str = NULL;
  signal(SIGCHLD, handler);

  if ((pid = fork()) ==  0) {
    sleep(5);
    return NULL;
  }
  else {
    if (sigsetjmp(env, 1) == 0) {
      str = fgets(s, size, stream);
      kill(pid, SIGKILL);
      pause();
    }
    return str;
  }
}
```

Problem 60. (8 points):

This problem concerns the four versions of `tfgets` from the previous pages. Some of them are correct, and others are flawed because the author didn't understand basic concepts of concurrency and signaling.
Circle the versions that are correct, in the sense that they return the input string if typed within 5 seconds, timeout after 5 seconds by returning NULL, and correctly reap their terminated children.

`Version A` `Version B` `Version C` `Version D`

Note: The `pause` function sleeps until a signal is received and then returns.

Problem 61. (10 points):

The following problem concerns the way virtual addresses are translated into physical addresses.

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Virtual addresses are 16 bits wide.
- Physical addresses are 14 bits wide.
- The page size is 1024 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 32 pages are as follows:

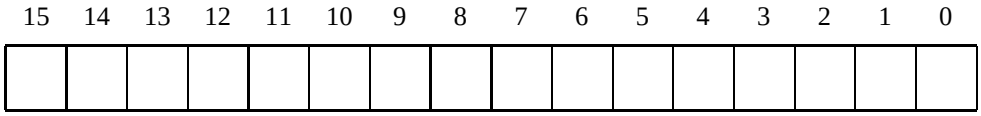
TLB			
Index	Tag	PPN	Valid
0	8	7	1
	F	6	1
	0	3	0
	1	F	1
1	1	E	1
	2	7	0
	7	3	0
	B	1	1
2	0	0	0
	C	1	0
	F	8	1
	7	6	1
3	8	4	0
	3	5	0
	0	D	1
	2	9	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	2	0	10	1	1
01	5	1	11	3	0
02	7	1	12	9	0
03	9	0	13	7	1
04	F	1	14	D	1
05	3	1	15	5	0
06	B	0	16	E	1
07	D	1	17	6	0
08	7	1	18	1	0
09	C	0	19	0	1
0A	3	0	1A	8	1
0B	1	1	1B	C	0
0C	0	1	1C	0	0
0D	D	0	1D	2	1
0E	0	0	1E	7	0
0F	1	0	1F	3	0

Part 1

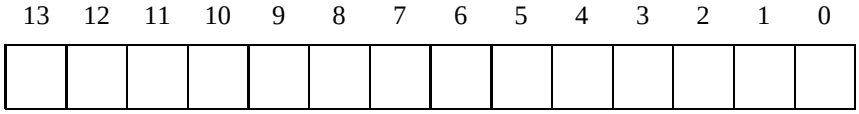
A. The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

VPO The virtual page offset
VPN The virtual page number
TLBI The TLB index
TLBT The TLB tag



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



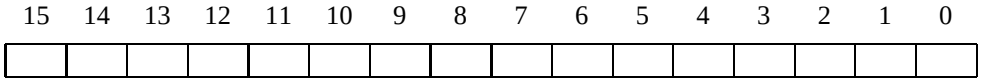
Part 2

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: `2F09`

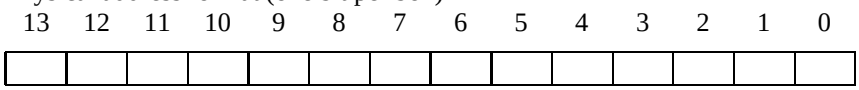
A. Virtual address format (one bit per box)



B. Address translation

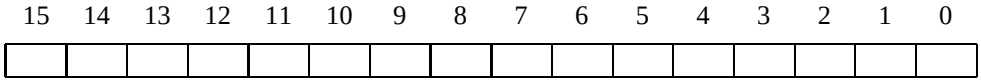
Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

C. Physical address format (one bit per box)



Virtual address: `0C53`

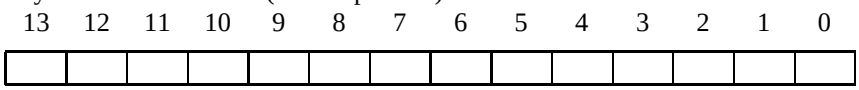
A. Virtual address format (one bit per box)



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)



Problem 62. (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 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 32 pages are as follows:

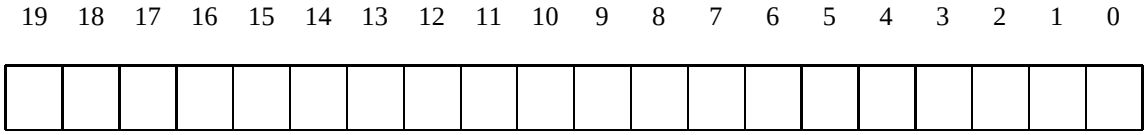
TLB			
Index	Tag	PPN	Valid
0	03	B	1
	07	6	0
	28	3	1
	01	F	0
1	31	0	1
	12	3	0
	07	E	1
	0B	1	1
2	2A	A	0
	11	1	0
	1F	8	1
	07	5	1
3	07	3	1
	3F	F	0
	10	D	0
	32	0	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	7	1	10	6	0
01	8	1	11	7	0
02	9	1	12	8	0
03	A	1	13	3	0
04	6	0	14	D	0
05	3	0	15	B	0
06	1	0	16	9	0
07	8	0	17	6	0
08	2	0	18	C	1
09	3	0	19	4	1
0A	1	1	1A	F	0
0B	6	1	1B	2	1
0C	A	1	1C	0	0
0D	D	0	1D	E	1
0E	E	0	1E	5	1
0F	D	1	1F	3	1

A. Part 1

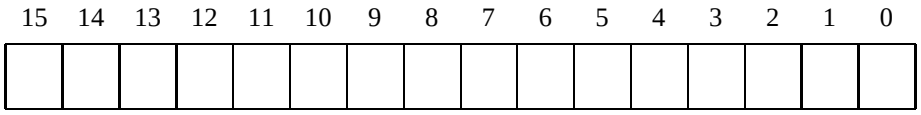
(a) The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

VPO The virtual page offset
VPN The virtual page number
TLBI The TLB index
TLBT The TLB tag



(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



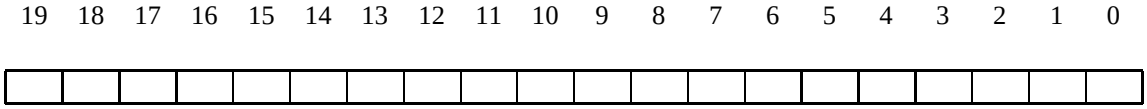
B. Part 2

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: `7E37C`

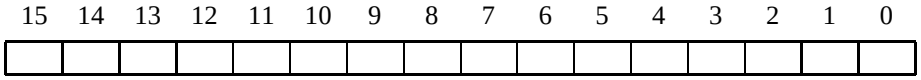
(a) Virtual address format (one bit per box)



(b) Address translation

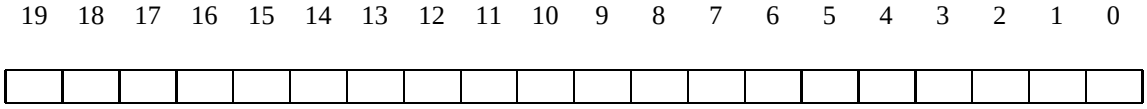
Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

(c) Physical address format (one bit per box)



Virtual address: `16A48`

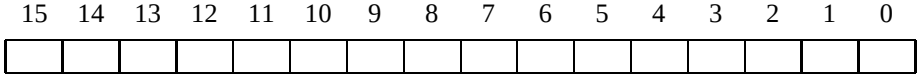
(a) Virtual address format (one bit per box)



(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)



Problem 63. (12 points):

The following problem concerns the way virtual addresses are translated into physical addresses.

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Virtual addresses are 16 bits wide.
- Physical addresses are 13 bits wide.
- The page size is 512 bytes.
- The TLB is 8-way set associative with 16 total entries.
- The cache is 2-way set associative, with a 4 byte line size and 16 total lines.

In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB, the page table for the first 32 pages, and the cache are as follows:

TLB			
Index	Tag	PPN	Valid
0	09	4	1
	12	2	1
	10	0	1
	08	5	1
	05	7	1
	13	1	0
	10	3	0
	18	3	0
1	04	1	0
	0C	1	0
	12	0	0
	08	1	0
	06	7	0
	03	1	0
	07	5	0
	02	2	0

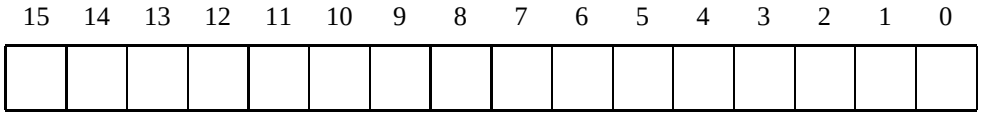
Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	6	1	10	0	1
01	5	0	11	5	0
02	3	1	12	2	1
03	4	1	13	4	0
04	2	0	14	6	0
05	7	1	15	2	0
06	1	0	16	4	0
07	3	0	17	6	0
08	5	1	18	1	1
09	4	0	19	2	0
0A	3	0	1A	5	0
0B	2	0	1B	7	0
0C	5	0	1C	6	0
0D	6	0	1D	2	0
0E	1	1	1E	3	0
0F	0	0	1F	1	0

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	19	1	99	11	23	11	00	0	99	11	23	11
1	15	0	4F	22	EC	11	2F	1	55	59	0B	41
2	1B	1	00	02	04	08	0B	1	01	03	05	07
3	06	0	84	06	B2	9C	12	0	84	06	B2	9C
4	07	0	43	6D	8F	09	05	0	43	6D	8F	09
5	0D	1	36	32	00	78	1E	1	A1	B2	C4	DE
6	11	0	A2	37	68	31	00	1	BB	77	33	00
7	16	1	11	C2	11	33	1E	1	00	C0	0F	00

Part 1

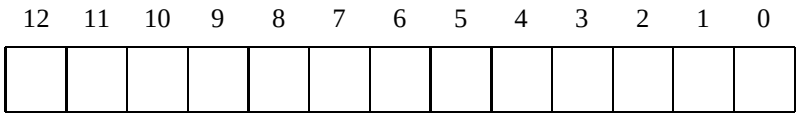
A. The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

VPO The virtual page offset
VPN The virtual page number
TLBI The TLB index
TLBT The TLB tag



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



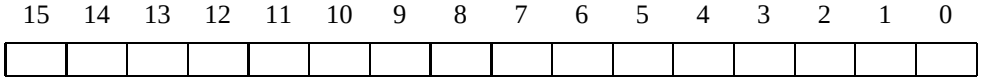
Part 2

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

If there is a cache miss, enter "-" for "Cache Byte returned". If there is a page fault, enter "-" for "PPN" and leave parts C and D blank.

Virtual address: `1DDE`

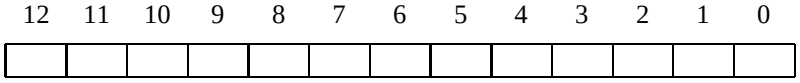
A. Virtual address format (one bit per box)



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)



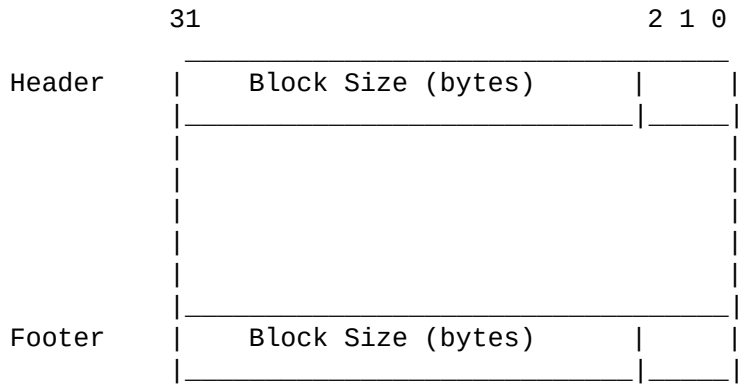
D. Physical memory reference

Parameter	Value
Byte offset	0x
Cache Index	0x
Cache Tag	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Dynamic storage allocation

The following problem concerns dynamic storage allocation.

Consider an allocator that uses an implicit free list. The layout of each allocated and free memory block is as follows:



Each memory block, either allocated or free, has a size that is a multiple of eight bytes. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer. The usage of the remaining 3 lower order bits is as follows:

- `bit 0` indicates the use of the current block: 1 for allocated, 0 for free.
- `bit 1` indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- `bit 2` is unused and is always set to be 0.

Problem 64. (8 points):

Given the contents of the heap shown on the left, show the new contents of the heap (in the right table) after a call to `free(0x400b010)` is executed. Your answers should be given as hex values. Note that the address grows from bottom up. Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed.

Address	
0x400b028	0x00000012
0x400b024	0x400b611c
0x400b020	0x400b512c
0x400b01c	0x00000012
0x400b018	0x00000013
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	0x00000013
0x400b008	0x00000013
0x400b004	0x400b601c
0x400b000	0x400b511c
0x400affc	0x00000013

Address	
0x400b028	
0x400b024	0x400b611c
0x400b020	0x400b512c
0x400b01c	
0x400b018	
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	
0x400b008	
0x400b004	0x400b601c
0x400b000	0x400b511c
0x400affc	

Problem 65. (10 points):

Consider an allocator that uses an implicit free list. Each memory block, either allocated or free, has a size that is a multiple of eight bytes. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer and is represented in units of bytes. The usage of the remaining 3 lower order bits is as follows:

- `bit 0` indicates the use of the current block: 1 for allocated, 0 for free.
- `bit 1` indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- `bit 2` is unused and is always set to be 0.

Five helper routines are defined to facilitate the implementation of `free(void *p)`. The functionality of each routine is explained in the comment above the function definition. Fill in the body of the helper routines the code section label that implement the corresponding functionality correctly.

```
/* given a pointer p to an allocated block, i.e., p is a
   pointer returned by some previous malloc()/realloc() call;
   returns the pointer to the header of the block*/
void * header(void* p)
{
  void *ptr;

  _______;
  return ptr;
}

A. ptr=p-1
B. ptr=(void *)((int *)p-1)
C. ptr=(void *)((int *)p-4)


/* given a pointer to a valid block header or footer,
   returns the size of the block */
int size(void *hp)
{
  int result;

  _______;
  return result;
}

A. result=(*hp)&(~7)
B. result=((*(char *)hp)&(~5))<<2
C. result=(*(int *)hp)&(~7)
```

```
/* given a pointer p to an allocated block,i.e. p is
   a pointer returned by some previous malloc()/realloc() call;
   returns the pointer to the footer of the block*/
void * footer(void *p)
{
  void *ptr;

  _______;
  return ptr;
}
```

A. ptr=p+size(header(p))-8
B. ptr=p+size(header(p))-4
C. ptr=(int *)p+size(header(p))-2

```
/* given a pointer to a valid block header or footer,
   returns the usage of the currect block,
   1 for allocated, 0 for free */
int allocated(void *hp)
{
  int result;

  ______;
  return result;
}
```

A. result=(*(int *)hp)&1
B. result=(*(int *hp)&0
C. result=(*(int *)hp)|1

```
/* given a pointer to a valid block header,
   returns the pointer to the header of previous block in memory */
void * prev(void *hp)
{
  void *ptr;

  ______;
  return ptr;
}
```

A. ptr = hp - size(hp)
B. ptr = hp - size(hp-4)
C. ptr = hp - size(hp-4) + 4