(B&O Ch. 2.1, K&R Chapter 2.9)
Write a single C expression that swaps both 16-bit halves of an integer i.

```
int i;
```

(B&O Ch. 3.4, Problem 3.1)
Consider the following values stored at the indicated memory addresses and registers:

Address	Value
0x100	0x89
0x108	0xAB
0x110	0xCD
0x118	0xEF

Register	Value
%rax	0x108
%rdx	0x2

What are the values for the following source operands when used with a movl instruction? (i.e. What does `%rbx` contain after executing "`movq S,%rbx`" when `S` is each of the operands below?)

a) `$0x110`

b) `0x110`

c) `(%rax)`

d) `-8(%rax)`

e) `-8(%rax,%rdx,8)`

f) If the value in the `%rcx` register is 8, then what is the hexadecimal value in `%rax` after this instruction: `leaq 4( ,%rcx, 4), %rax`

(B&O Chapter 3.4, Problem 3.5)
Consider the following assembly routine

```
dx:
    movq %rdx, %rax
    subq %rsi, %rax
    movq %rax,(%rdi)
    retq
```

Fill in the missing lines of the following C function. Include the return value.

```
long dx(long* xp, long y, long z) {




}
```

(B&O Chapter 3.4, Problem 3.5)
Consider the following assembly routine

```
fx:
    movq      %rdi, %rax
    imulq     %rsi, %rax
    addq      $5, %rax
    retq
```

Fill in the the corresponding C function. Only one statement is required.

```
long fx(long x, long y) {



}
```

(B&O Ch. 3.5, Problem 3.6)
Suppose `%rax` contains `x` and `%rcx` contains `y` at the beginning of each instruction below. What would %rdx contain after each instruction is executed?

a) `leaq -8(%rax), %rdx`

b) `leaq (%rax, %rcx, 4), %rdx`

c) `leaq 4(%rax, %rcx, 2), %rdx`

d) `leaq 0xA(,%rcx,2), %rdx`

(B&O Ch. 3.5, Problem 3.10, 3.58)

Consider the assembly code implemention of a C function:

```
arith:
     subq    %rdx, %rdi              ; t1
     addq    %rdx, %rsi              ; t2
     leaq    (%rsi,%rsi,4), %rax
     addq    %rax, %rax              ; t3
     addq    %rdi, %rax              ; t4
     retq                            ;
```

The C it was generated from is listed below with the expressions that are calculated replaced by blanks. Based on the assembly code, fill in these blanks:

```
long arith(long x, long y, long z) {

     long t1 = _________;

     long t2 = _________;

     long t3 = _________;

     long t4 = _________;

     return ________;

}
```

(B&O 3.6, Problem 3.18)

Consider the assembly code implemention of a C function:

```
iffy:
     leaq    (%rdi,%rsi), %rax       ; result
     cmpq    $5, %rax                ; if
     jge     .L2
     leaq    4(%rdx, %rax), %rax     ; then
     retq
.L2:
     subq    %rdx, %rax              ; else
     retq
```

The C it was generated from is listed below with the expressions that are calculated replaced by blanks. Based on the assembly code, fill in these blanks:

```
long iffy(long x, long y, long z) {

     long result = ___________;

     if ( _________ )

          _____________________;

     else

          _____________________;

     return result;

}
```

(B&O Ch. 3.6, Problem 3.28)
Consider the following assembly routine that takes two input parameters `x` and `y`. Recall that the `cmpq` instruction (`cmpq` S_1, S_2) sets the condition flags by performing $S_2 - S_1$

```
forloop:
        xorq    %rax, %rax
        xorq    %rdx, %rdx
        addq    %rsi, %rdi
        jmp     .L2
.L3:
        addq    %rdi, %rax
        addq    $1, %rdx
.L2:
        cmpq    $31, %rdx
        jl      .L3
        ret
```

Fill in the 3 missing statements in the C code below that was used to generate this assembly code. Do not use local variables.

```
long forloop(long a, long b) {

     long i;

     long result=0;

     for (________ ; _________; i++) {

          ____________________ ;

     }

     return result;

}
```

(B&O Chapter 3.7, Problem 3.35)

The assembly routine below implements a recursive function:

```
rfun:
        testq   %rdi, %rdi
        je      .L3
        subq    $8, %rsp
        subq    $1, %rdi
        call    rfun
        addq    $1, %rax
        jmp     .L2
.L3:
        movq    %rsi, %rax
        retq
.L2:
        addq    $8, %rsp
        retq
```

Fill in the corresponding C function:

```
long rfun(long a, long b) {
     if ( _____________ )

          return ______________________;

     else

          return ______________________;

}
```

(B&O Chapter 3.6, Problem 3.31, 3.62, 3.63)
The assembly routine below implements a switch statement using a jump table

```
switcher:
     movq %rdi,%rax
     subq $30,%rax
     cmpq $4,%rax
     ja .L5
     jmp *.L7(,%rax,8)
.L2
     movq $2,%rax
     jmp .L6
.L3
     movq $3,%rax
     jmp .L6
.L4
     movq $4,%rax
     jmp .L6
.L5
     movq $0,%rax
.L6
     retq
.L7:
     .long .L2
     .long .L3
     .long .L4
     .long .L3
     .long .L2
```

Write the corresponding C function for this routine including the appropriate types and return values.

(B&O Chapter 3.8, Problem 3.36, 3.37)
Consider the following declaration

```
short S[15];
double *W[4];
```

a) What is the total size of the array `S` in bytes?

b) Assuming the address of `S` is stored in `%rbx` and `i` is stored in `%rdx`, write a single `movw` instruction using the scaled index memory mode that loads `S[i]` into `%rax`

c) What is the total size of the array `W` in bytes?

d) Assuming the address of `W` is stored in `%rbx` and `i` is stored in `%rdx`, write a single `movq` instruction using the scaled index memory mode that loads `W[i]` into `%rax`

(B&O Ch. 3.8, Problem 3.38)
Consider the following C code, where `M` and `N` are constants declared with `#define`

```
long P[M][N];
long Q[N][M];
long sum_element(long i, long j) {
    return P[i][j] + Q[j][i];
}
```

In compiling this program, `gcc` generates the following:

```
sum_element:
        leaq    (%rdi,%rdi,8), %rdx
        addq    %rsi, %rdx
        leaq    (%rsi,%rsi,2), %rax
        addq    %rax, %rdi
        movq    Q(,%rdi,8), %rax
        addq    P(,%rdx,8), %rax
        retq
```

Use your reverse engineering skills to determine the values of `M` and `N`.

a) M =

b) N =

(B&O Ch. 3.9, Problem 3.41)
Consider the following declarations:

```
typedef struct {
     int i;
     double d;
     char c;
} s_t;

typedef union {
     int i;
     double d;
     char c;
} u_t;

s_t  s, *sp, sa[5];
u_t  u, *up, ua[5];
```

What are the size (in bytes) of the following variables?

a) `s`

b) `sp`

c) `u`

d) `up`

e) `s.c`

f) `&s.c`

Suppose the address of `sa` is `0x100` and the address of `ua` is `0x200`. What are the addresses in hex of the following?

g) `&sa[2].c`

h) `&ua[2].c`

(B&O Chapter 3.8, 3.9, Problem 3.44)
Consider the following structure definitions on an x86-64 machine. Determine the total size of each structure

a) `struct P1 { long l ; char c; int i; char d };`

b) `struct P2 { float f ; char c; char d; long l };`

c) `struct P3 { short w[3]; int *c[3] };`

(B&O Chapter 3.8, 3.9, Problem 3.45)
Reorganize this structure in the space next to it in order to minimize its size

```
struct rec {
     short a;
     char *b;
     double c;
     char d;
     int e;
};
```

(B&O Chapter 3.4)
Consider the following code using embedded assembly

```
long myasm(long x, long y) {
        long result;

        asm("imulq %1,%2; xorq $15,%2; movq %2,%0"
                : "=r" (result)
                : "r" (x), “r” (y)
        );
        return result;
}
main() {
        long k;
        k = myasm(8,3);
        printf("%ld\n",k);
}
```

What is the output of the `printf` statement?

(B&O Chapter 3.9, 3.10)

a) For this union,

```
union {
     long i;
     char c;
} u;
```

What is `sizeof(u)` ?

b) For this structure,

```
struct S5 {
     char *c1;
     char c2;
     int i;
} p;
```

What is `sizeof(p)` ?

(B&O Chapter 5.4)

Assume that `a` and `b` are integer arrays. Use code motion and reimplement the following loop in a more efficient manner.

```
int vsum(int n) {
     int i,j;
     for (i = 0; i < n; i++)
          for (j = 0; j < n; j++)
               b[i][j] = a[i*n+j];
}
```

(B&O Chapter 5.6)

Reimplement the following routine so that it eliminates unneeded memory references.

```
int a[N];
void sum_a() {
     a[0] = 0;
     for (i=1; i<N; i++)
          a[0] += a[i];
}
```

(B&O Chapter 5.8, Problem 5.14)
Consider the following routine

```
int vsum(int n) {
    int i, sum=0;
    for (i = 0; i < n; i++)
        sum += a[i]*b[i];
    return sum;
}
```

Assume that `a` and `b` are integer arrays and `n` is a multiple of 3. Rewrite this routine using 3x1 loop unrolling.

(B&O Chapter 5.8, Problem 5.14)
Consider the following routine

```
int vsum(int n) {
    int i, sum=0;
    for (i = 0; i < n; i++)
        sum += a[i]*b[i];
    return sum;
}
```

Assume that `a` and `b` are integer arrays and `n` is a multiple of 3. Rewrite this routine using 3x3 loop unrolling (parallel accumulators).

(B&O Chapter 5.8, Problem 5.14)

Consider the following routine

```
int vsum(int n) {
     int i, prod=1;
     for (i = 0; i < n; i++)
          prod *= a[i]*b[i];
     return prod;
}
```

Assume that `a` and `b` are integer arrays and `n` is a multiple of 8. Rewrite this routine using 8x1a loop unrolling (reassociation with a single accumulator)

(B&O Chapter 6, Problem 6.12-6.15)

Consider the 2-way set associative cache below with (S,E,B,m) = (8,2,4,13). Cache lines that are blank are invalid.

Set	Tag	Data0-3	Tag	Data0-3
0	09	86 30 3F 10	00	
1	45	60 4F E0 23	38	00 BC 0B 37
2	EB		0B	
3	06		32	12 08 7B AD
4	C7	06 78 07 C5	05	40 67 C2 3B
5	71	0B DE 18 4B	6E	
6	91	A0 B7 26 2D	F0	
7	46		DE	12 C0 88 37

a) Consider an access to 0x037A

b) What is the block offset of this address in hex?

c) What is the set index of this address in hex?

d) What is the cache tag of this address in hex?

e) Does this access hit or miss in the cache?

f) What value is returned if it is a hit?

(B&O Chapter 6, Problem 6.16)
Consider the 2-way set associative cache below with (S,E,B,m) = (8,2,4,13). Cache lines that are blank are invalid. List all addresses that will hit in Set 0

Set	Tag	Data0-3	Tag	Data0-3
0	09	86 30 3F 10	00	
1	45	60 4F E0 23	38	00 BC 0B 37
2	EB		0B	
3	06		32	12 08 7B AD
4	C7	06 78 07 C5	05	40 67 C2 3B
5	71	0B DE 18 4B	6E	
6	91	A0 B7 26 2D	F0	
7	46		DE	12 C0 88 37

(B&O Chapter 8, Problem 8.2, 8.13)
Consider the code below. How many times is "Bye" printed when the program is executed?

```
#include <stdio.h>
int main()
{
     if (fork() == 0) {
          if (fork() == 0) {
               fork();
          }
     }
     printf("Bye\n");
}
```

(B&O Chapter 8, Problem 8.4, 8.7, 8.21)

Consider the code below. What is the output of the program when executed?

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int counter = 5;

void handler_chld() {
      int child_status;
      wait(&child_status);
      printf("%d\n",counter);
      exit(0);
}

int main() {
      signal(SIGCHLD,handler_chld);
      if (fork()==0) {
              counter--;
              if (fork()==0) {
                  counter--;
                  printf("%d\n",counter);
              }
              else
                  pause();
      } else {
              counter++;
              pause();
      }
}
```

(B&O Chapter 8, Problem 8.4, 8.7)
Consider the code below. What is the output of the program when executed?

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>

void handler_alrm() {
      printf("D\n");
      exit(0);
}
void handler_chld() {
      int child_status;
      wait(&child_status);
      printf("E: %d\n", WEXITSTATUS(child_status));
      exit(0);
}
int main() {
      int child_status;
      int pid;
      signal(SIGALRM,handler_alrm);
      signal(SIGCHLD,handler_chld);

      pid = fork();

      if (pid == 0) {
              pause();
              printf("A\n");
      }
      else {
              kill(pid,SIGALRM);
              pause();
              printf("B\n");
      }
      printf("C\n");
      exit(0);
}
```