Question 1

int decode2(int x, int y, int z)

```
1:   movl 16(%ebp), %eax
2:   movl 12(%ebp), %edx
3:   subl %eax, %edx
4:   movl %edx, %eax
5:   imull 8(%ebp), %edx
6:   sall $31, %eax
7:   sarl $31, %eax
8:   xorl %edx, %eax
```

Q. Write the corresponding C code

Answer:

```
int decode2(int x, int y, int z) {
     int t1 = y - z;
     int t2 = x * t1;
     int t3 = (t1 << 31) >> 31;
     int t4 = t3 ^ t2;
     return t4;
}
```

Question 2

```
1: int absdiff2(int x, int y) {
2:  int result;
3:  if (x < y)
4:      result = y - x;
5:  else
6:      result = x - y;
7:  return result;
8: }
```

Corresponding assembly code

```
1:  movl 8(%ebp), %edx
2:  movl 12(%ebp), %ecx
3:  movl %edx, %eax
4:  subl %ecx, %eax
5:  cmpl %ecx, %eax
6:  jge .L3
7:  movl %ecx, %eax
8:  subl %edx, %eax
9:  .L3
```

a. What subtractions are performed when x < y? When x>= y?

b. In what way does this code deviate from the standard implementation of an if-else?

c. Using C syntax (including goto's), show the general form of this translation

d. What restrictions **must** be imposed on the use o this translation to guarantee that it has the behavior specified by the C code?

Answers:
a. when x < y it first computes x – y and then y – x, when x >= y it computes y – x.
b. The code for the then-statement is executed unconditionally.

c.
then-statement
t = test-expression
if (t)
  goto done
else-statement

done:

Question 3:

Switch statements are particularly tricky to reverse engineer from the object code. In the following procedure, the body of the switch statement has been removed:

```
1:  int switch_prob(int x) {
2:      int result = x;
3:      switch (x) {
4:          /* Fill in code here … */
5:      }
6:      return result;
7:  }
```

Figure 3.38 shows the disassembled object code for the procedure. We are only interested in the part of the code shown on lines 4 through 16. We can see on line 4 that parameter x (at offset 8 relative to %ebp) is loaded into register %eax, corresponding to program variable result. The "lea 0x0(%esi), %esi" instruction on line 11 is a nop instruction inserted to make the instruction on line 12 start on an address that is a multiple of 16.

The jump table resides in a different area of memory. Using the debugger GDB, we can examine the six 4-byte words of memory starting at address 0x8048468 with the command x/6w 0x8048468. GDB prints the following:

(gdb) x/6w

```
0x8048468: 0x080483d5    0x080483eb    0x080483d5    0x0x80483e0
0x8048478: 0x080483e5    0x080483e8
```

(gdb)

Assembly Code:

```
1:  080483c0 <switch_prob>:
2:    80483c0:        push %ebp
3:    80483c1:        mov %esp,%ebp
4:    80483c3:        mov 0x8(%ebp),%eax
5:    80483c6:        lea 0xfffffce(%eax),%edx
6:    80483c9:        cmp $0x5, %edx
7:    80483cc:        ja 80483eb <switch_prob+0x2b>
8:    80483ce:        jmp *0x8048468(,%edx,4)
9:    80483d5:        shl $0x2, %eax
10:   80483d8:        jmp 80483ee <switch_prob+0x2e>
11:   80483da:        lea 0x0(%esi),%esi
12:   80483e0:        sar $0x2, %eax
13:   80483e3:        jmp 80483ee <switch_prob+0x2e>
14:   80483e5:        lea (%eax, %eax, 2), %eax
15:   80483e8:        imul %eax, %eax
16:   80483eb:        add $0xa, %eax
17:   80483ee:        mov %ebp, %esp
18:   80483f0:        pop %ebp
19:   80483f1:        ret
```

```
20:   80483f2:         mov %esi, %esi
```

Answer:

```
int switch_prob(int x) {
  int result = x;
  switch (x) {
    case 50:
    case 52:
       result <<= 2;
    break;

    case 53:
      result >>= 2;
          break;

    case 54:
       result *= 3;
     break;

         default:
     result += 10;
  }
  return result;
}
```

Question 4:

```
typedef struct {
    int left;
    a_struct a[CNT];
    int right;
} b_struct;

void test(int i, b_struct *bp) {
   a_struct *ap = &bp->a[i];
   int n = bp->left + bp->right;
   ap->x[ap->idx] = n;
}
```

Unfortunately, the ".h" file defining the compile-time constant CNT and the structure a_struct are in files for which you do not have access privileges. Fortunately you have access to the a ".o" version of the code, which you are able to disassemble with the objdump program, yielding the disassembly shown below.

```
0  00000000<test>
1     0:   push %ebp
2     1:   mov %esp, %ebp
3     3:   push %ebx
4     4:   mov 0x8(%ebp), %eax
5     7:   mov 0xc(%ebp), %ecx
6     a:   lea (%eax, %eax, 4), %eax
7     d:   lea 0x4(%ecx, %eax, 4), %eax
8     11:  mov (%eax), %edx
9     13:  shl $0x2, %edx
10    16:  mov 0xb8(%ecx), %ebx
11    1c:  add (%ecx), %ebx
12    1e:  mov %ebx, 0x4(%edx, %eax, 1)
13    22:  pop %ebx
14    23:  mov %ebp, %esp
15    25:  pop %ebp
16    26:  ret
```

Determine

a. What is the value of CNT?

b. A complete declaration of structure a_struct. Assume the only fields in this structure are idx and x.

Answers:
a. CNT is 9

b.
```
typedef struct {
   int idx;
   int x[4];
} a_struct;
```

Question 5:

```
union ele {
     struct {
          int *p;
          int y;
     } e1;
     struct {
          int x;
          union ele *next;
     } e2;
};
```

This declaration illustrates that structures can be embedded within unions.

The following procedure (with some expressions omitted) operates on a linked list consisting of these unions as list elements:

```
void proc (union ele *up)
{
     up → ___________ = *(up → _____________) - up → _________;
}
```

a. What would be the offsets (in bytes) of the following fields:

e1.p:
e1.y:
e2.x:
e2.next:

b. How many total bytes would the structure require?

c. The compile generates the following assembly code for proc:

```
1    movl 8(%ebp), %eax
2    movl 4(%eax), %edx
3    movl (%edx), %ecx
4    movl %ebp, %esp
5    movl 8(%eax), %eax
6    movl 8(%ecx), %ecx
7    subl %eax, %ecx
8    movl %ecx, 4(%edx)
```

Use this code to fill in the blanks for the C source. There is only one answer that does not perform any casting and does not violate any type constraints.

Answers:
This problem very clearly shows that unions are simply a way to associate multiple names (and types) with a single storage location.

a. Draw out the layout in memory

e1.p	e1.y
e2.x	e2.next

b. 8 bytes

c.
Line 1: get up
Line 2: up->e1.y (no) or up->e2.next
Line 3: up->e2.next->e1.p or up->e2.next->e2.x (no)
Line 4: up->e1.p (no) or up->e2.x
Line 5: *(up->e2.next->e1.p)
Line 6: *(up->e2.next->e1.p) – up->e2.x
Line 7: Store in up->e2.next->e1.y

Question 6:

This question is testing your understanding of the stack frame structure.

a. The following memory image embeds a binary tree with root node at `0x804961c`. Please draw the logical organization of the tree in the same format as the shown example. Please indicate the address and key value (in hexidecimal) of all the tree nodes and the pointers from parent nodes to child nodes. The declaration of the tree node structure is as follows.

```
struct tree_node {
      int key;
      struct tree_node *left;
      struct tree_Node *right;
};

/* address of the root */
tree_node *root;

Memory:
<address> <value>
0x80495f8: 0x0000000c
0x80495fc: 0x00000000
0x8049600: 0x00000000
0x8049604: 0x0000001f
0x8049608: 0x080495f8
0x804960c: 0x08049610
0x8049610: 0x00000022
0x8049614: 0x00000000
0x8049618: 0x00000000
0x804961c: 0x00000037
0x8049620: 0x08049604
0x8049624: 0x08049628
0x8049628: 0x0000003c
0x804962c: 0x00000000
0x8049630: 0x08049634
0x8049634: 0x0000004e
0x8049638: 0x00000000
0x804963c: 0x00000000
```

```
1: struct tree_node * search(struct tree_node * node, int value)
2: {
3:    if (node->key == value)
4:     return node;
5:    else if (node->key > value) {
6:     if (node->left == NULL)
7:       return NULL;
8:     else
9:      return search(node->left, value);
10:   } else {
11:    if (node->right == NULL)
12:      return NULL;
```

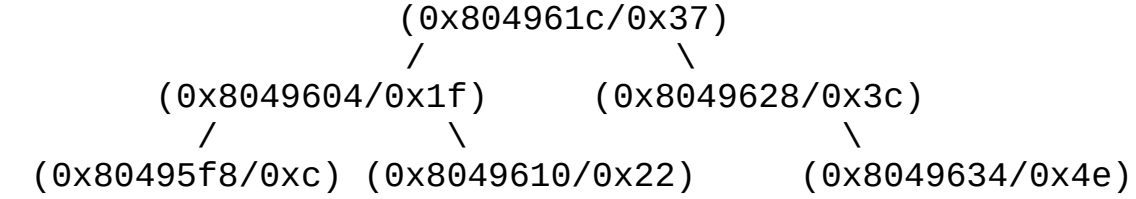
```
13:    else
14:      return search(node->right, value);
15:   }
16: }
```

b. Suppose we call search(root, 0x4e). Fill in the blanks the value of these memory location so that it shows the stack when the execution is at line 4. (More space than needed is provided.) You can assume that the stack stores only arguments, return address, and the ebp register value. The value of ebp is 0xbffff880 when the program calls the function. Write ”rtn addr” for return addresses.

Address	Value
0xbffff800	0x4e
0xbffff7fc	0x804961c
0xbffff7f8	rtn_addr
0xbffff7f4	
0xbffff7f0	
0xbffff7ec	
0xbffff7e8	
0xbffff7e4	
0xbffff7e0	
0xbffff7dc	
0xbffff7d8	
0xbffff7d4	
0xbffff7d0	
0xbffff7cc	
0xbffff7c8	
0xbffff7c4	

Answers:

a.



b.

<Address>	<Value>
0xbffff800	0x4e
0xbffff7fc	0x804961c
0xbffff7f8	rtn_addr
0xbffff7f4	0xbffff880
0xbffff7f0	0x4e

```
0xbffff7ec   0x8049628
0xbffff7e8   rtn_addr
0xbffff7e4   0xbffff7f4
0xbffff7e0   0x4e
0xbffff7dc   0x8049634
0xbffff7d8   rtn_addr
0xbffff7d4   0xbffff7e4
```

Question 7:

a.
Which of the following x86 instructions can be used to add two registers and store the result without overwriting either of the original values?
(a) mov
(b) add
(c) lea
(d) None of above

b.
The register rax is currently storing a NULL pointer. Which of the following x86 instructions will cause a segmentation fault because of an invalid memory access?
(a) mov (%rax), %rcx
(b) lea (%rax), %rcx
(c) None of the above

c. A programmer wishes to compare the contents of a string called my_str to the string "GET". She writes the following C code:

```
if (my_str == "GET") ...
```

Which of the following apply?

(a) my str is a pointer to the first character of a string in memory
(b) my str is the ASCII value of the first character of a string in memory
(c) my str is a register containing all of the characters in the string
(d) "GET" will compile to a pointer to a string in memory
(e) "GET" will compile to the ASCII value for the letter "G"
(f) "GET" will compile to a register containing the string "GET" represented as an integer
(g) The comparison will always work as expected
(h) The comparison will not necessarily work as expected
(i) The comparison itself will cause the program to crash

d. The function `foo()` is declared in a C program as follows: `void foo(int int_param, char *str_param);`

A programmer calls `foo()` from within the function `bar()` as follows:

```
foo(my_int, my_string);
```

Which of the following is/are true:

(a) If `foo()` changes the value of `int param`, the change will propagate back to the calling function`bar()`, in other words, the value of `my int` will also change.

(b) If `foo()` changes the second character of `str param`, the change will propagate back to the calling function `bar()`, in other words, the second character of `my string` will also change.

(c) If `foo()` changes the address of `str param` to point to a different string, the change will propagate back to the calling function `bar()`, in other words, `my string` will now point to a different string.

(d) None of the above.

e. A programmer has declared an array in a C program as follows:

```
int my_array[100];
```

Which of the following give(s) the address of the eighth element in the array (bearing in mind that the first element in the array is at index zero):

(a) `my array[7]`
(b) `&my array[7]`
(c) `my array + 7`
(d) `my array + 28`
(e) None of the above

g. A programmer has stored an 8-bit value in memory. The pointer:

char *ptr;

points to the location where it is stored. He or she now wants to retrieve the value and store it into the variable:

int value;

Which of the following (if any) will achieve this properly?

(a) value = ptr;
(b) value = *ptr;
(c) value = (int)ptr;
(d) value = (int *)ptr;
(e) value = *(int *)ptr;
(f) None of the above

h. Given code for an implicit list memory allocator, you claim you can improve performance by creating a linked list of free blocks. Why does this help increase performance?

(a) Traversing a linked list is significantly faster than moving from block to block in the implicit list.
(b) The implicit list had to include every block in memory, but the linked list could just include the free

blocks.
(c) The compiler knows how to optimize the code for a linked list by unrolling loops, but wasn't able to do
this for the implicit list.
(d) Having a linked list made coalescing significantly faster.
(e) None of the above.

Answer(s):

Answers:

Question 8: Synchronization

A multithreaded program has two global data structures that will be shared among the threads. The data structures are not necessarily accessed at the same time. Which of the following is/are true (if any)?

(a) If the program has only one semaphore, and threads call P on that single semaphore before using either
of the data structures, the code will not work correctly.

(b) Having one semaphore will work, but having two, one per shared data structure, may allow for increased
performance.

(c) If the machine has only one processor, only one of the threads can run at a time, so semaphores are not
necessary in that case.

(d) None of the above.

Correct answer(s):

Answers:

Question 9: Synchronization

A barbershop consists of n waiting chairs and the one barber chair. If there are no customers, the barber waits. If a customer enters, and all the waiting chairs are occupied, then the customer leaves the shop. If the barber is busy, but waiting chairs are available, then the customer sits in one of the free waiting chairs. We provide you the skeleton code without any synchronization.

```
extern int N;
int customers = 0;

void *customer() {


      if (customers > N) {

            return NULL;
      }

      customers += 1;

      getHairCut();

      customers -= 1;

      return NULL;
}


void *barber() {
      while (1) {

            cutHair();

      }
}
```

Our solution uses **three** binary sempahores:

`mutex:` **to control access to the global variable customers**
`customer:` **to signal a customer is in the shop**
`barber:` **to signal the barber is busy**

a. What are the initial values for the three semaphores?
b. Complete the above without changing ANY existing code but only by filling in as many copies of the following lines as you need.

```
P(&mutex);
V(&mutex);
P(&customer);
V(&customer);
P(&barber);
V(&barber);
```

Answers:

Here is *one* solution, for which the initial values are mutex = 1 (variable customers may be accessed), customer = 0 (no customers) and barber = 0 (barber is not busy).

```
void* customer() {
        P(&mutex);
        if (customers > N) {
                V(&mutex);
                return NULL;
        }

        customers += 1;
        V(&mutex);
        V(&customer);

        P(&barber);
        getHairCut();

        P(&mutex);
        customers -= 1;
        V(&mutex);

        return NULL;
}

void* barber() {
        while(1) {
                P(&customer);
                V(&barber);
                cutHair();
        }
}
```

Question 10: Synchronization #2

Consider the following three threads and semaphores

```
/* Initializing semaphores */
s1 = 1;
s2 = 0;
s3 = 0;

/* Initialize x */
x = 0;
```

```
void thread1() {
     x = x + 1;
}
```

```
void thread2() {
     x = x + 2;
}
```

```
void thread3() {
     x = x * 2;
}
```

Add P() and V() calls (and change nothing else) to the three threads such that at the end of any sequence of concurrent execution of the three threads, x will always be = 6.

Answer

```
/* Initializing semaphores */
s1 = 1;
s2 = 0;
s3 = 0;

/* Initialize x */
x = 0;
```

```
void thread1() {
     P(s1);

     x = x + 1;

     V(s1);
     V(s2);
}
```

```
void thread2() {
     P(s1);

     x = x + 2;

     V(s1);
     V(s3);
}
```

```
void thread3() {
     P(s2);
     P(s3);

     x = x * 2;

     V(s3);
     V(s2);
}
```

Question 11: Synchronization #3

Consider the following three threads and semaphores

```
/* Initializing semaphores */
s1 =
s2 =
s3 =
s4 =

/* Initialize x */
x = 1;
```

```
void thread1() {
   while (x != 360) {

      x = x * 2;

   }
   exit(0);
}
```

```
void thread2() {
   while (x != 360) {

      x = x * 3;

   }
   exit(0);
}
```

```
void thread3() {
   while (x != 360) {

      x = x * 5;

   }
   exit(0);
}
```

Provide initial values for the four semaphores and add P(), V() semaphore operations (using the four semaphores) in the code for thread 1, 2 and 3 such that the process is guaranteed to terminate.

Answer

```
/* Initializing semaphores */
s1 = 3
s2 = 2
s3 = 1
s4 = 1

/* Initialize x */
x = 1;
```

```
void thread1() {
   while (x != 360) {
      P(s1);
      P(s4);
      x = x * 2;
      V(s4);
   }
   exit(0);
}
```

```
void thread2() {
   while (x != 360) {
      P(s2);
      P(s4);
      x = x * 3;
      V(s4);
   }
   exit(0);
}
```

```
void thread3() {
   while (x != 360) {
      P(s3);
      P(s4);
      x = x * 5;
      V(s4);
   }
   exit(0);
}
```