**Submit only a .pdf file for the answer to these questions.**

**Q0 (10 points)** Answer briefly to the following questions.

1. What is mov in assembly? What is the difference between movl and movq?

**The MOV instruction moves a value from one register/memory location to another register/memory location (though you cannot move from memory to memory). ‘l’ and ‘q’ are suffixes that indicate the size of the value being moved. ‘l’ is a long word of 32 bits (8 bytes), ‘q’ is quad word of 64 bits (16 bytes).**

1. What are the three main types/categories of assembly operations/instructions?

**Arithmetic and logical operators, comparison operators, and jump instructions. You might also consider movement operators a category.**

**Q1 (10 points)** Each of the following lines of code generates an error message when we invoke the assembler. Explain what is wrong with each line.

1 movb $0xF, (%ql) **%ql is not a register**

2 movl %eax,$0xFA **You cannot move a value into an immediate**

3 movl %eax,%bx **%bx is only for w suffixes, and %eax is l.**

4 movb %ah,%sh **Neither are registers**

5 movw 4(%esp), (%ebx) **Can’t move from a memory location to a memory location.**

**Q2 (20 points)** You are given the following information. A function with prototype

void decode3(int \*xp, int \*yp, int \*zp);

is compiled into assembly code. The body of the code is as follows:

# *xp at %ebp+8, yp at %ebp+12, zp at %ebp+16*

movl 8(%ebp), %edi

movl 12(%ebp), %edx

movl 16(%ebp), %ecx

movl (%ecx), %esi

movl (%edx), %ebx

movl (%edi), %eax

movl %eax, (%edx)

movl %esi, (%edi)

movl %ebx, (%ecx)

Parameters xp, yp, and zp are stored at memory locations with offsets 8, 12, and

16, respectively, relative to the address in register %ebp.

Write C code for decode3 that will have an effect equivalent to the assembly

code above.

void decode(int \*xp, int \*yp, int \*zp) {

int temp = \*zp;

int temp2 = \*yp;

int temp3 = \*xp;

\*yp = temp3;

\*xp = temp;

\*zp = temp2;

}

**Q3 (10 points)** Suppose register %eax holds the value of variable named a and %ecx holds the value of variable named b. Fill in the table below with formulas indicating the value that will be stored in register %edx for each of the given assembly code instructions based on a and b:

|  |  |
| --- | --- |
| Instruction | Result |
| leal 4(%eax), %edx | 4+a |
| leal (%ecx,%eax), %edx | a+b |
| leal (%ecx,%eax,3), %edx | a\*3+b |
| leal 3(%eax,%eax,2), %edx | a\*3+3 |
| leal 3(%ecx,%eax,2), %edx | a\*2+b+3 |

**Q4 (20 points)**

Assume the following values are stored at the indicated memory addresses and registers:

|  |  |  |  |
| --- | --- | --- | --- |
| Address | Value | Register | Value |
| 0x100 | 0xFA | %eax | 0x109 |
| 0x104 | 0xAE | %ecx | 0xF |
| 0x108 | 0x13 | %edx | 0x3 |
| 0x10C | 0x5 |  |  |

Fill in the following table showing the effects of the following instructions, both in terms of the register or memory location that will be updated and the resulting value.

|  |  |  |
| --- | --- | --- |
| Instruction | Destination | Value |
| addl %ecx,(%eax) | 0x100 | 0x100 |
| subl %edx,4(%eax) | 0x104 | 0xAB |
| imull $16,(%eax,%edx,4) | 0x10C | 0x50 **(check 0x5\*16)** |
| incl 8(%eax) | 0x108 | 0x14 |
| decl %ecx | %ecx | 0xE |

**Q5 (20 points)**. Consider the following assembly code:

*# x at %ebp+8, n at %ebp+12*

movl 8(%ebp), %esi **x in %esi**

movl 12(%ebp), %ebx **n in %ebx**

movl $-1, %edi **-1 in a %edi**

movl $1, %edx **1 in mask**

.L2:

movl %edx, %eax **result = mask**

andl %esi, %eax **result = result & x**

xorl %eax, %edi **a = result ^ a**

movl %ebx, %ecx **n in b**

sall %cl, %edx **mask =** **mask** << **n**

testl %edx, %edx **mask == 0**

jne .L2

movl %edi, %eax

The preceding code was generated by compiling C code that had the following

overall form:

int loop(int x, int n)

{

int result = **-1**;

int mask;

for (mask = **1**; mask != **0**; mask = **mask << n**) {

result ^= **mask & x**;

}

return result;

}

Your task is to fill in the missing parts of the C code to get a program equivalent to the generated assembly code. Recall that the result of the function is returned in register %eax. You will find it helpful to examine the assembly code before, during, and after the loop to form a consistent mapping between the registers and the program variables. Answering these questions will help you to complete the C code.

A. Which registers hold program values x, n, result, and mask?

B. What are the initial values of result and mask?

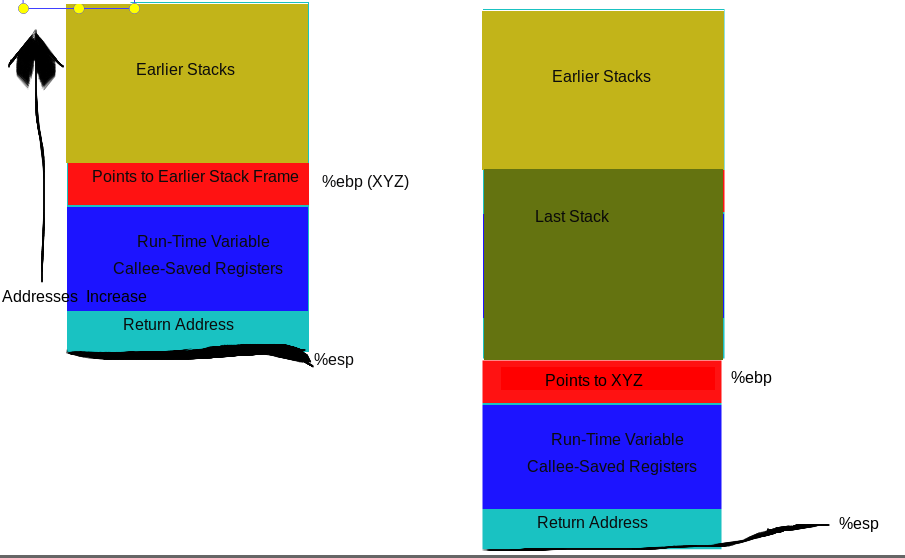
C. What is the test condition for mask?

D. How does mask get updated?

E. How does result get updated?

F. Fill in all the missing parts of the C code.

**Q6 (10 points)** Draw a stack, and show changes in the stack before and after a procedure call in 32 bit systems. Particularly show i) What are the changes in %esp, %ebp? ii) Where will the arguments be stored?



I forgot to include them, but function arguments will actually be stored below the return address, right above %esp. (Sorry for the hack job.)

**Q7 (15 point)**. based on the following assembly code, answer the questions.

**08048384 <swap>:**

**8048384: push %ebp**

**8048385: mov %esp,%ebp**

**8048387: push %ebx**

**8048388: mov 0x8(%ebp),%edx**

**804838b: mov 0xc(%ebp),%ecx**

**804838e: mov (%edx),%ebx**

**8048390: mov (%ecx),%eax**

**8048392: mov %eax,(%edx)**

**8048394: mov %ebx,(%ecx)**

**8048396: pop %ebx**

**8048397: pop %ebp**

**8048398: ret**

**80483b4: movl $0x8049658,0x4(%esp)**

**80483bc: movl $0x8049654,(%esp)**

**80483c3: call 8048384 <swap>**

**80483c8: leave # Prepare to return**

**80483c9: ret**

1. Why %ebx is pushed and poped?

**%ebx is a caller-saved register, meaning that swap is responsible for making sure it returns to the callee function as it was originally. It can do this by either pushing itself onto the stack and popping itself at the end of the function, or by just being left unchanged.**

1. After the call to swap is over, what is the value in %ebp?

**The top of whatever stack frame (B) it referred to before swap (C) was called, which itself points to whatever stack frame the next function is called from (A). (That’s recursion for you.)**

1. Assume %esp has the value 0x70412345 before call 8048384 <swap>. What would be the value of %esp after the call is over?

**The same value, 0x70412345, because popping it simply reverts it back to its old value.**