第三次小作业

3.1

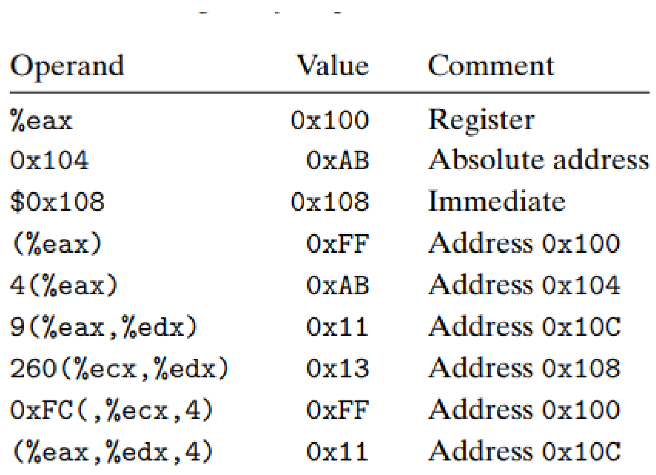
Assume the following values are stored at the indicated memory addresses and registers:

Address	**Value**	**Register**	**Value**
0x100	0xFF	%eax	0x100
0x104	0xAB	%ecx	0x1
0x108	0x13	%edx	0x3
0x10C	0x11		

Fill in the following table showing the values for the indicated operands:

Operand	**Value**
%eax	
0x104	
$0x108	
(%eax)	
4(%eax)	
9(%eax,%edx)	
260(%ecx,%edx)	
0xFC(,%ecx,4)	
(%eax,%edx,4)	

答案：



Operand	Value	Comment
%eax	0x100	Register
0x104	0xAB	Absolute address
$0x108	0x108	Immediate
(%eax)	0xFF	Address 0x100
4(%eax)	0xAB	Address 0x104
9(%eax,%edx)	0x11	Address 0x10C
260(%ecx,%edx)	0x13	Address 0x108
0xFC(,%ecx,4)	0xFF	Address 0x100
(%eax,%edx,4)	0x11	Address 0x10C

3.15

In the following excerpts from a disassembled binary, some of the information has been replaced by Xs. Answer the following questions about these instructions.

A. What is the target of the je instruction below? (You don't need to know anything about the call instruction here.)

```
804828f:    74 05                   je      XXXXXXX
8048291:    e8 1e 00 00 00          call    80482b4
```

B. What is the target of the jb instruction below**?**

```
8048357:    72 e7                   jb      XXXXXXX
8048359:    c6 05 10 a0 04 08 01    movb    $0x1,0x804a010
```

C. What is the address of the mov instruction?

```
XXXXXXX:    74 12                   je      8048391
XXXXXXX:    b8 00 00 00 00          mov     $0x0,%eax
```

D. In the code that follows, the jump target is encoded in PC-relative form as a 4-byte, two's-complement number. The bytes are listed from least significant to most, reflecting the little-endian byte ordering of IA32. What is the address of the jump target?

```
80482bf:    e9 e0 ff ff ff          jmp     XXXXXXX
80482c4:    90                      nop
```

E. Explain the relation between the annotation on the right and the byte coding on the left.

```
80482aa:    ff 25 fc 9f 04 08       jmp     *0x8049ffc
```

答案：

A. The je instruction has as target 0x8048291 + 0x05. As the original disassembled code shows, this is 0x8048296:

```
804828f:      74 05                 je     8048296
8048291:      e8 1e 00 00 00        call   80482b4
```

B. The jb instruction has as target 0x8048359 − 25 (since 0xe7 is the 1-byte, two's-complement representation of −25). As the original disassembled code shows, this is 0x8048340:

```
8048357:      72 e7                 jb     8048340
8048359:      c6 05 10 a0 04 08 01  movb   $0x1,0x804a010
```

C. According to the annotation produced by the disassembler, the jump target is at absolute address 0x8048391. According to the byte encoding, this must be at an address 0x12 bytes beyond that of the mov instruction. Subtracting these gives address 0x804837f, as confirmed by the disassembled code:

```
804837d:      74 12                 je     8048391
804837f:      b8 00 00 00 00        mov    $0x0,%eax
```

D. Reading the bytes in reverse order, we see that the target offset is 0xffffffe0, or decimal −32. Adding this to 0x80482c4 (the address of the nop instruction) gives address 0x80482a4:

```
80482bf:      e9 e0 ff ff ff        jmp    80482a4
80482c4:      90                    nop
```

E. An indirect jump is denoted by instruction code ff 25. The address from which the jump target is to be read is encoded explicitly by the following 4 bytes. Since the machine is little endian, these are given in reverse order as fc 9f 04 08.

3.34

For a C function having the general structure

```c
int rfun(unsigned x) {
    if (________)
    return ________;
    unsigned nx = ________;
    int rv = rfun(nx);
    return ________;
}
```

gcc generates the following assembly code (with the setup and completion code omitted):

```
movl    8(%ebp), %ebx
movl    $0, %eax
testl   %ebx, %ebx
```

```
je      .L3
movl    %ebx, %eax
shrl    %eax                          Shift right by 1
movl    %eax, (%esp)
call    rfun
movl    %ebx, %edx
andl    $1, %edx
leal    (%edx,%eax), %eax
.L3:
```

A. What value does rfun store in the callee-save register %ebx?

B. Fill in the missing expressions in the C code shown above.

C. Describe in English what function this code computes

答案：

A. Register `%ebx` holds the value of parameter `x`, so that it can be used to compute the result expression.

B. The assembly code was generated from the following C code:

```
int rfun(unsigned x) {
    if (x == 0)
        return 0;
    unsigned nx = x>>1;
    int rv = rfun(nx);
    return (x & 0x1) + rv;
}
```

C. Like the code of Problem 3.49, this function computes the sum of the bits in argument `x`. It recursively computes the sum of all but the least significant bit, and then it adds the least significant bit to get the result.

3.56

Consider the following assembly code:

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

```
movl    8(%ebp), %esi
movl    12(%ebp), %ebx
movl    $-1, %edi
movl    $1, %edx
.L2:
movl    %edx, %eax
andl    %esi, %eax
xorl    %eax, %edi
movl    %ebx, %ecx
sall    %cl, %edx
testl   %edx, %edx
```

```
jne     .L2
movl    %edi, %eax
```

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

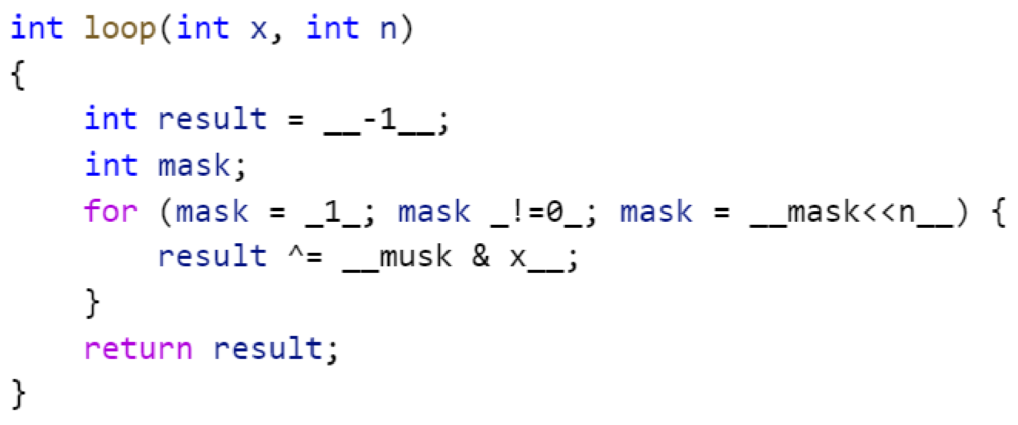
```c
int loop(int x, int n)
{
    int result = _________;
        int mask;
    for (mask = _________; mask _________; mask = _________) {
        result ^= _________;
    }
    return result;
}
```

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.

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

答案:

A. x: %esi   n:%ebx result:%edi   mask:%edx
B. result=-1, mask=1
C. mask != 0
D. mask << n
E. result ^= x&mask



```c
int loop(int x, int n)
{
    int result = __-1__;
    int mask;
    for (mask = _1_; mask _!=0_; mask = __mask<<n__) {
        result ^= __musk & x__;
    }
    return result;
}
```

F.

3.57

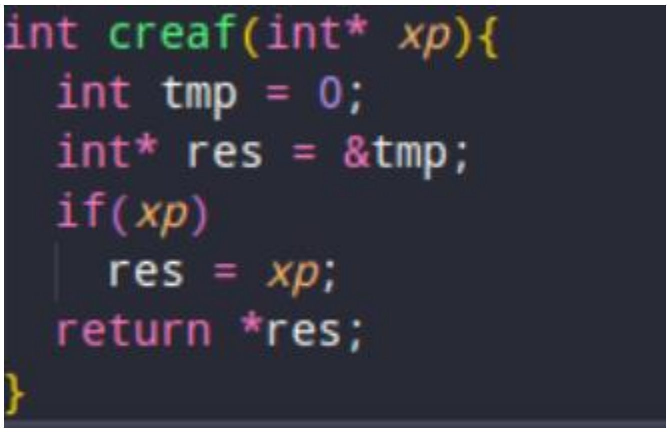
In Section 3.6.6, we examined the following code as a candidate for the use of conditional data transfer:

```
int cread(int *xp) {
    return (xp ? *xp : 0);
}
```

We showed a trial implementation using a conditional move instruction but argued that it was not valid, since it could attempt to read from a null address.

Write a C function cread_alt that has the same behavior as cread, except that it can be compiled to use conditional data transfer. When compiled with the command-line option '-march=i686', the generated code should use a conditional move instruction rather than one of the jump instructions.

答案:



```
int creaf(int* xp){
  int tmp = 0;
  int* res = &tmp;
  if(xp)
    res = xp;
  return *res;
}
```