3.1

%eax的值为0x100；0x104的值为0xAB；$0x108的值为0x108；

(%eax)的值即为0x100的值，即为0xFF；4(%eax)的值即为0x104的值，即为0xAB；

9(%eax,%edx)的值即为0x10C的值，即为0x11,；260转化为0x104，于是260(%ecx,%edx) 的值为0x108的值，即为0x13；0xFC(,%ecx,4)的值即为0xFC+4\*%ecx=0x100的值，为0xFF；

(%eax,%edx,4)即为%eax+4\*%edx，即为0x10C的值，为0x11；

因此表格如下：

|  |  |
| --- | --- |
| 操作数 | 值 |
| %eax | 0x100 |
| 0x104 | 0xAB |
| $0x108 | 0x108 |
| (%eax) | 0xFF |
| 4(%eax) | 0xAB |
| 9(%eax,%edx) | 0x11 |
| 260(%ecx,%edx) | 0x13 |
| 0xFC(,%ecx,4) | 0xFF |
| (%eax,%edx,4) | 0x11 |

3.3

第一行，不能使用%bl作为地址寄存器；

第二行，movl和%ax不匹配，应当用movw；

第三行，源与目标不能同时作为存储器的引用；

第四行，不存在叫做%sh的寄存器；

第五行，立即数$0x123不能作为目标数；

第六行，目标为%dx，与movl不匹配；

第七行，movb和%si不匹配；

3.6

通过指令leal的规定格式可以很容易得出下表：

|  |  |
| --- | --- |
| 指令 | 结果 |
| leal 6(%eax),%edx | 6+x |
| leal (%eax,%ecx),%edx | x+y |
| leal (%eax,%ecx,4),%edx | x+4\*y |
| leal 7(%eax,%eax,8),%edx | 7+9\*x |
| leal 0xA(,%ecx,4),%edx | 10+4\*y |
| leal 9(%eax,%ecx,2),%edx | 9+x+2\*y |

3.7

值得注意的是最后两个指令的目的因为没有()，不求值。

|  |  |  |
| --- | --- | --- |
| 指令 | 目的 | 值 |
| addl %ecx,(%eax) | 0x100 | 0x100 |
| subl %edx,4(%eax) | 0x104 | 0xA8 |
| imull $16,(%eax,%edx,4) | 0x10C | 0x110 |
| incl 8(%eax) | 0x108 | 0x14 |
| decl %ecx | %ecx | 0x0 |
| subl %edx,%eax | %eax | 0xFD |

3.9

movl只是将位于%ebp+12的转移到%eax，不显示在C代码中；xorl将%eax中的y与8(%ebp)中的x进行异或操作，将x^y储存在%eax中（记为t1）；sarl将t1向右移位3个单位，记为t2；notl将t2取反，记为t3；subl将t3减去z，记为t4；

因此填写如下：

第5行：int t1 = x^y;

第6行：int t2 = t1>>3;

第7行：int t3 = ~t2;

第8行：int t4 = t3-z;

3.12

1. 从第4行的mull可知num\_t为无符号数，又因为需要进行两次乘法和读取两次内存，为64位数的乘法，故num\_t为unsigned long long类型；
2. 因为y为64位数，记为y中高32位的值，记为y中低32位的值，故

故，将生成的96位数取低64位。乘积得到的结果的低位部分即为（64位）的低位部分，高位部分即为的高位部分加上的低32位，由此得到一个64位数。

3.13

1. 由setl知，COMP是对补码的小于(<)操作，并且是有符号数，因此又由cmpl（32位）可判断出data\_t为int；
2. 由setge知，COMP是对补码的大于等于(>=)操作，并且是有符号数，因此又由cmpw（16位）可判断出data\_t为short；
3. 由setb知，COMP是对无符号数的低于(<)操作，又由cmpb知data\_t为unsigned char；
4. 由setne知，COMP是不等(!=)操作，data\_t可能是int,unsigned或long类型；

3.15

1. je的目标即为0x8048291+0x05=0x8048296
2. 0x59+0xe7=0x40，故jb指令的目标为0x8048340
3. mov指令的地址为0x8048391-0x12=0x804837f
4. 反着读e9 e0 ff ff ff为0xffffffe0，再加上0x80482c4，即jmp的目标为0x80482a4
5. ff 25为间接跳转指令，用小端法的机器中fc 9f 04 08即为0x08049ffc，即题中的0x8049ffc

3.16

1. void cond(int a, int \*p)

{

if (p == 0)

goto done;

if (a <= 0)

goto done;

\*p += a;

done:

return;

}

B.&&前的条件分支如果测试失败，则不再测试&&之后的a>0；

3.18

int test(int x, int y) {

int val = x ^ y;

if (x < -3) {

if (x > y)

val = x \* y;

else

val = x + y;

} else if ( x > 2)

val = x - y;

return val;

}

3.21

1. 在第6行（初始化）中，%edx的值为a+b，在循环体中由于a=a+1，为了保持%edx的值为a+b，其本身也要加一；

B.

|  |  |  |
| --- | --- | --- |
| 寄存器 | 程序值 | 初始值 |
| %eax | result | 1 |
| %ebx | b | b |
| %ecx | a | a |
| %edx | temp | a+b |

C.

加了注释后的汇编代码：

movl 8(%ebp), %ecx //将a赋值给寄存器%ecx

movl 12(%ebp), %ebx //将b赋值给寄存器%ebx

movl $1, %eax //将result初始化为1

cmpl %ebx, %ecx //作差：a-b，对其符号进行判断

jge .L11 //若a-b>=0 即a >= b，则直接return result，结束函数进程

leal (%ebx, %ecx), %edx //创建临时变量temp，初始化为a+b，放入寄存器%edx

movl $1, %eax //再将result初始化为1

.L12:

imull %edx, %eax //将result与1+b相乘，并赋给result

addl $1, %ecx //a++

addl $1, %edx //将result也加一，使其始终为a+b

cmpl %ecx, %ebx //作差：b-a，对其符号进行判断

jg .L12 //若b > a，则继续循环，反之退出循环

.L11: //结束

D.

int loop\_while (int a, int b)

{

int result = 1;

if (a >= b)

goto done;

int temp = a + b;

loop:

result \*= temp;

a++;

temp++;

if ( a < b)

goto loop;

done:

return result;

}

3.22

A.

int fun\_a(unsigned x) {

int val = 0;

while ( x ) {

val = val ^ x;

x = x >> 1;

}

return val & 0x1;

}

B.

这个函数充当奇校验，若x中1的个数为奇数，则return 1，否则 return 0；

3.23

A.

int fun\_b(unsigned x) {

int val = 0;

int i;

for (i = 0; i != 32; i++) {

int temp = 2 \* val;

val = x;

val = val & 0x1;

val = val | temp;

x = x >> 1;

}

return val;

}

B.

主要计算为 val = (val << 1) | (x & 0x1)和x >>= 1，将x中的位反过来创造镜像

3.27

int test(int x, int y) {

int val = x+y ;

if ( y > 0 ) {

if ( x < y)

val = x - y;

else

val = x ^ y;

} else if (y >= -2)

val = 4\*x;

return val;

}

3.28

A.

起初x=x+2，随后判断x-6的符号，若x>6，则跳转到switch的默认分支，否则进入switch选择，最大情况标号为4，最小情况标号为-2.而跳转表中第3行指令与.L2相同，为缺失项，故情况标号值为-2,0,1,2,3,4

B.由跳转表易知.L6的情况标号为2、3；

3.54

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

{

z -= y;

int temp = z;

temp = ( temp << 15) >> 15;

z ^= x;

temp \*= z;

return temp;

}

3.55

movl 12(%ebp), %esi //获得x

movl 20(%ebp), %eax //获得y

movl %eax, %edx //temp1=y

sarl $31,%edx //temp1=y>>31

movl %edx, %ecx //temp2=temp1=y>>31

imull %esi, %ecx //temp2=(y>>31)\*x=x的低位\*y的符号位

movl 16(%ebp), %ebx //temp3=x的高位

imull %eax, %ebx //temp3=x的高位\*y

addl %ebx, %ecx //temp2=x的高位\*y+x的低位\*y的符号位

mull %esi //temp1=(x的低位\*y)的高位，y=(x的低位\*y)的低位

leal (%ecx,%edx), %edx //temp1=temp2+temp1=(x的低位\*y)的高位+x的低位\*y的符号位

movl 8(%ebp), %ecx //temp2=dest

movl %eax, (%ecx)

movl %edx, 4(%ecx) //此时高位为(x\*y\_u)\_h-x，低位为(x\*y\_u)\_l

3.56

A.x保存在%esi，n保存在%ebx，result保存在%edi，mask保存在%edx；

B.result的初始值为1431655765，即0x55555555，mask的初始值是-2147483648，即0x80000000；

C.mask的测试条件为 mask != 0;

D.mask的修改代码为mask>>= n;

E.result的修改代码为result ^= mask & x;

F.

int loop(int x, int n)

{

int result = 0x55555555;

int mask;

for (mask = 0x80000000; mask != 0; (unsigned)mask>>n) {

result ^= (mask & x);

}

return result;

}

3.58

int switch3(int \*p1, int \*p2, mode\_t action)

{

int result = 0;

switch(action) {

case MODE\_A:

result = \*p1;

\*p1 = \*p2;

break;

case MODE\_B:

result = \*p2 + \*p1;

\*p2 = result;

break;

case MODE\_C:

\*p2 = 15;

result = \*p1;

break;

case MODE\_D:

\*p2 = \*p1;

result = 17;

break;

case MODE\_E:

result = 17;

break;

default:

result = -1;

break;

}

return result;

}

3.59

分析思路：在第4行中，参数n被加载到寄存器%eax中；

第5行：%eax -= 0x28，即n -= 40；

第6行：作差n-5，也即开始的n0-45；

由此可知上限为45，又由跳转表有6个4字节的字可知switch有6个分支，预设置为40,41,42,43,44,45,46，其中可能有缺失的。

第7行：若n>45，则跳至第9行，重新加载%eax后跳至第24行，result+=0x11，这就是default的情形；

若n<=45，则看第8行，跳至跳转表的0x80485f0段，首先是0x08048442（n==40），跳至第14行，result <<= 3，随后跳至第25行return result；然后是0x08048442，以此类推，按序跳转即可。

因此补全的C程序如下：

int switch\_prob(int x, int n)

{

int result = x;

switch( n ) {

case 40:

case 42:

result <<= 3;

break;

case 43:

result >>= 3;

break;

case 44:

result <<= 3;

result -= x;

result \*= result;

break;

case 45:

result \*= result;

break;

default:

result += 0x11;

break;

}

return result;

}