Solutions to *CSAPP 2ed*

# Prictice Problems

### 3.1

本题考察对Figure 3.3(p.203)的理解。

|  |  |
| --- | --- |
| Operand | Value |
| %eax | 0x100 |
| 0x104 | 0xAB |
| $0x108 | 0x108 |
| (%eax) | M(0x100)=0xFF |
| 4(%eax) | M(4+0x100)=0xAB |
| 9(%eax, %edx) | M(9+0x100+0x3)=M(0x10C)=0x11 |
| 260(%ecx, %edx) | M(260+0x1+0x3)=M(264)=M(0x108)=0x13 |
| 0xFC(, %ecx, 4) | M(0xFC+0x1\*4)=M(0x100)=0xFF |
| (%eax, %edx, 4) | M(0x100+0x3\*4)=M(0x10C)=0x11 |

### 3.2

本题根据operands选择相应size的instraction。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | mov | %eax, (%esp) | movl | Register – Memory (Stack Pointer) |
| 2 | mov | (%eax), %dx | movw | Memory – Register (word) |
| 3 | mov | $0xFF, %bl | movb | Immediate – Register (word) |
| 4 | mov | (%esp, %edx, 4), %dh | movb | Memory – Register (byte) |
| 5 | push | $0xFF | pushl | Immediate – Memory (Stack) |
| 6 | mov | %dx, (%eax) | movw | Register – Memory |
| 7 | pop | %edi | popl | Memory – Register, movl (%esp), %edi |

### 3.3

本题从instractions的size出发，判断指令的使用错误。

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | movb | $0xF, (%bl) | %bl不是一个register. |
| 2 | movl | %ax, (%esp) | S的size是word, D是Memory，应该用movw |
| 3 | movw | (%eax), 4(%esp) | S和D是Memory，由于不能直接在memory内部使用mov，此处不合法。 |
| 4 | movb | %ah, %sh | Register中没有%sh |
| 5 | movl | %eax, $0x123 | D不能是constants |
| 6 | movl | %eax, %dx | S的size是32-bit，D的size是16-bit，两者不匹配。用movw可行吗？ |
| 7 | movb | %si, 8(%ebp) | S的size是word，应该用movw |

问题：Instruction的size取决于register，是这样吗？

### 3.4

本题要注意类型转换、类型对应的register sizes、指令对应的sizes。

参考figure 3.4.

|  |  |  |
| --- | --- | --- |
| src\_t | dest\_t | Instruction |
| int | int | movl %eax, (%edx) |
| char | int | movsbl %al, (%edx) |
| char | unsigned | ~~movzbl~~ %al, (%edx) movsbl %al, (%edx) |
| unsigned char | int | movzbl %al, (%edx) |
| int | char | movb %al, (%edx) |
| unsigned | unsigned char | movb %al, (%edx) |
| unsigned | int | movl %eax, (%edx) |

### 3.5

本题根据assembly code写出对应的C代码。

通过观察，我发现assembly code的语义是交换指针所指对象的值。

void decode1(int \*xp, int \*yp, int \*zp)

{

int x = \*xp; // 1, 4

int y = \*yp; // 2, 5

int z = \*zp; // 3, 6

/\* \*yp <- x

\* \*xp <- z

\* \*zp <- y

\*/

\*yp = x; // 7

\*xp = z; // 7

\*zp = y; // 8

}

### 3.6

本题主要考察Figure 3.3及leal指令。本题只需应用公式即可，参考Figure 3.3(p.203).

Leal指令将S的字面值复制到D。

|  |  |
| --- | --- |
| Instruction | Result |
| leal 6(%eax), %edx | x+6 |
| leal (%eax, %ecx), %edx | x+y |
| leal (%eax, %ecx, 4), %edx | x+4y |
| leal 7(%eax, %eax, 8), %edx | 9x+7 |
| leal 0xA(, %ecx, 4), %edx | 4y+10 |
| leal 9(%eax, %ecx, 2), %edx | x+2y+9 |

### 3.7

本题形式上与3.1相似。

|  |  |  |
| --- | --- | --- |
| Instruction | Destination | Value |
| addl %ecx, (%eax) | 0x100 | 0xFE |
| subl %edx, 4(%eax) | 0x104 | 0xA8 |
| imull $16, (%eax, %edx, 4) | 0x10C | 16 |
| incl 8(%eax) | 0x108 | 0x14 |
| decl %ecx | %ecx | 0x0 |
| subl %edx, %eax | %eax | 0xFC |

### 3.8

参照figure 3.7.

sal $2, %eax

sar %ecx, %eax

### 3.9

本题根据ASM写出C代码。

|  |  |
| --- | --- |
| 5 | y^x |
| 6 | t1>>3 |
| 7 | !t2 |
| 8 | t3-z |

### 3.10

本题是要干啥？

1. 该指令的两个operands都是%edx，执行效果是将%edx所有bit设为0。可以实现将变量值设为0.
2. 更直观的方式：movl $0, %edx。
3. 要比较的是：编码需要的bytes。两种方式的差异源于第一个operand。在第一种方式中，第一个operand是register；在第二种方式中，第一个operand(0)是一个int，需要32 bit编码。那么，一个register需要多少bit呢？根据对assembly code的观察，可以知道register的bit是少于32的。

### 3.11

本题参考p.218的代码，将第3行修改为: divl 12(%ebp)

### 3.12

本题要根据ASM的行为，分析变量的类型。

在第2行和第4行，用了两个register存储y变量的值，表明y 的类型是64bit，且不是floating-point。由于题中将64bit类型数据分到两个registers中做处理，因此，我认为该机器是32 bit。在32bit机器上，long long int和unsigned long long是64bit。由于代码第3行使用imull，表明是signed类型值。因此，我认为num\_t是long long int。

算法：由于x是unsigned，扩展到64 bit，便是zero extended。然后依据下列公式，针对x与y 的不同部分计算乘积，然后组合结果，等价于x\*y。

|  |  |  |
| --- | --- | --- |
|  | 64 bit | 32 bit |
|  |  |  |
| \* |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### 3.13

本题要根据指令写出相应语句，两者在语义上一致。参考Figure 3.10, 3.11。

1. 32 bits, data\_t = int, COMP = <
2. 16 bits, data\_t = short, unsigned short, COMP = >=
3. 8 bits, data\_t = char, COMP = <
4. 32 bits, data\_t = int, unsigned int, COMP = =

### 3.14

TEST可以为{=, !=, >, <, >=, <=}.

???

### 3.15

PC-relative

假设jmp指令占用一个byte。

1. Target被编码为0x05。

因此，结果为。

1. Target被编码为0xe7，根据Two’s Comp.,先转换为相应signed值，然后从8048359中减去该值。

因此，结果为。

1. 根据PC-Relative的规则，我们知道mov instraction的地址是

因此，结果为。

1. 本题将序列的排列设定为little-endian. addr = ff ff e0 e9.
2. ff -> jmp, 25 fc 9f 04 08 -> \*0x8049ffc.

### 3.16

本题综合考虑conditional instructions和jump instructions。参考Figure 3.10, 3.12。

1. 本题要写出goto版本的代码。先理解语义，给asm注释如下

|  |  |
| --- | --- |
| 1 | Get *x* |
| 2 | Get *y* |
| 3 | Check pointer *p* is null or not |
| 4 | if *p* is null, goto **done** |
| 5 | Check x is negative, zero or positive |
| 6 | if *x* less or equal to 0, goto **done** |
| 7 | Compute result = x + y |
| 8 | **done** |

1. C中用一个if语句包含两种情况。在ASM中，这两种情况要独立判断，因此，需要两个condional branches.

### 3.17

1. 本小题写goto版本代码。
2. 我认为要看各分支内部的逻辑，选择将逻辑较复杂的分支做为主干。

### 3.18

本题根据ASM和C代码结构，推断表达式。

我可以从ASM中执行顺序靠后的语句入手。

从C中知道，假设不发生跳转，那么最终返回val初始值。而在ASM的第7行，设置val的初值。于是，我们知道C中第2行的计算体现在ASM的第7行。

C中第3行if语句对于ASM的第3行；第4行if语句对于ASM的第5行。

### 3.19

A. 本题可以逆向思考：从1开始，连续乘积，何时overflow。

问题：怎么判断乘积是否overflow?

B. ？？

### 3.20

本题考察do-while loop及对应的asm，参考Figure 3.14作答。

A.

|  |  |  |
| --- | --- | --- |
| Register | Variable | Initially |
| %eax | x | x |
| %ecx | y | y |
| %edx | n | n |

B.

*test-expr*是(n>0) && (y<n)，对应到asm，是第8、10两行。

body statement是第3-5行，对应到asm，是第5-7行。

C.

|  |  |
| --- | --- |
|  | Arguments: x at %ebp+8, y at %ebp+12, n at %ebp+16 |
|  | Registers: x in %eax, y in %ecx, n in %edx, |
| 1 | Get x |
| 2 | Get y |
| 3 | Get n |
| 4 | loop: |
| 5 | Compute x +=n |
| 6 | Compute y \*=n |
| 7 | Compute n-- |
| 8 | Compare n:0 |
| 9 | If <=, goto done |
| 10 | Compare y:n |
| 11 | If <, goto loop |
| 12 | done: return x |

### 3.21

1. 该变量的值是a+b，每次循环后，该值增加1.
2. Lll

|  |  |  |
| --- | --- | --- |
| Register | Variable | Initially |
| %ecx | a | a |
| %ebx | b | b |
| %eax | result | 1 |
| %edx | a+b | a+b |

|  |  |  |
| --- | --- | --- |
|  | Arguments: *a* at %ebp+8, *b* at %ebp+12 | |
|  | Registers: *a* in %ecx, *b* in %ebx, *result* in %eax, *a+b* in %edx | |
| 1 | | Get *a* |
| 2 | | Get *b* |
| 3 | | Set *result* to 1 |
| 4 | | Compare *a:b* |
| 5 | | If >=, goto done |
| 6 | | Set a new variable to *a+b* |
| 7 | | Set *result* to 1 （为什么重复？） |
| 8 | loop: | |
| 9 | | Compute *result \*= (a+b)* |
| 10 | | Increament *a* |
| 11 | | Increament *(a+b)* |
| 12 | | Compare *a:b* |
| 13 | | If >, goto loop |
| 14 | done: | |
|  | | Return *result* |

1. 见代码

### 3.22

1. 本题要根据ASM写C代码。思路：。具体见代码fun\_a.c。

通过观察不同版本的fun\_a函数，可以发现不同语言在描述同一计算过程时的差异。

1. 对x所有bit做xor运算，最后返回一个bit。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | … |  |  |  |

该函数可以用来判断bit seq中，值为1的位的数量是偶数还是奇数。如果是奇数个，返回1；若是偶数个，返回0。

### 3.23

本题与3.22相同。

1. 先根据ASM写出goto version。具体见代码fun\_b.c。
2. 该函数的参数是一个unsigned，在将该翻转参数bit顺序后返回。

### 3.24

1. 代码：

int sum = 0;

int i;

i = 0; // init

while (i < 0) {

// body

if (i & 1)

continue;

sum += 1;

// update-expr

i++;

如果简单的转换成while form，那么update-expr可能会被忽略执行。

1. 可以考虑将update-expr做为独立的区块。具体见代码sum\_even.c。

### 3.25

1. 参考p.243 Aside的计算过程。
2. 当判断出错时，那么函数的消耗就是两个部分：mispredication penalty和成功执行所需时间。计算如下：

### 3.26

本题根据ASM分析语义，确定C代码。

通过分析，我认为。

注释

|  |  |
| --- | --- |
|  | Register: x in %edx |
| 1 | Compute 4\*x, store into %eax |
| 2 | Test x |
| 3 | If x >= 0, then let %eax = x |
| 4 | Compute %eax / 4, and return the value in %eax |

### 3.27

本题与3.18类似。具体见代码

### 3.28

本题根据ASM判断C代码的特征。

1. 我们知道，jump table的indices是连续的；ASM的第2-4行是判断参数是否超过index的上限。

假设参数为x，执行ASM第二行，得到x+2。在第三行，比较x+2与6。第四行，如果x+2大于6，则跳转到结束，可以知道，case lable的值小于等于6。并且，第四行，ja指令的含义表示unsigned类型比较。因此，可以知道case lable的值范围是[0, 6]。

根据以上，得出indices的范围是

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| -2 | -1 | 0 | 1 | 2 | 3 | 4 |

1. jump table第6、7两行重复。对应到C，就是case 2和case 3。

### 3.29

本题根据ASM写出C代码。

关键是case values的取值范围，这点可以从ASM第2行得知。在此基础上，结合jump table，可以知道每个case对应的ASM代码块，如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | .L7 | |  | Case |
| 2 | | .long | .L3 | 0 |
| 3 | | .long | .L2 | 1 |
| 4 | | .long | .L4 | 2 |
| 5 | | .long | .L2 | 3 |
| 6 | | .long | .L5 | 4 |
| 7 | | .long | .L6 | 5 |
| 8 | | .long | .L2 | 6 |
| 9 | | .long | .L4 | 7 |

由ASM第三行ja，可知L2对应default。

当case为2、7时，共调用L4两次，可知L4对应Case C, D。

接着，我们分析ASM代码之间的跳转关系，如下:

.L2

.L8

.L5

.L6

.L3

.L4

由于.L6与.L3之间关系，可知分别对应Case A，Case B。最后，剩下.L5对应Case E。

综合以上，得到

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | .L7 | |  | Case |  |
| 2 | | .long | .L3 | 0 | Case B |
| 3 | | .long | .L2 | 1 | default |
| 4 | | .long | .L4 | 2 | Case C |
| 5 | | .long | .L2 | 3 | default |
| 6 | | .long | .L5 | 4 | Case E |
| 7 | | .long | .L6 | 5 | Case A |
| 8 | | .long | .L2 | 6 | default |
| 9 | | .long | .L4 | 7 | Case D |

代码见switcher.c。

### 3.30

本题考察对instraction和调用机制的理解。

1. 等价于：。
2. 若是ret instruction，还得修改%ebp的值。
3. ？？

### 3.31

本题考察对函数调用机制的理解。

如果这些ASM是调用者的代码，完成在函数调用前的准备工作。其中，错误在于，搞反了caller-save和callee-save。在第2、3、4行，应当是保存%eax, %edx, %ecx到stack。

如果这些ASM是被调用函数的代码，那么没有问题。

### 3.32

本题关键是参数在stack frame中的排序。

稍加观察，可发现ASM的第三行对应：\*p=d；第6行对应：x-c。

结合参数在stack frame中的相对顺序，可知

|  |  |
| --- | --- |
| +20 | c |
| +16 | p |
| +12 | d |
| +8 | x |

然后，根据获取参数时使用的instractions确定参数类型。第1行的instraction是movsbl，可知d是char。第4行的instraction是movswl，可知x是16位的signed short。第5行的instraction是movl，假设c是int。

|  |  |  |
| --- | --- | --- |
| +20 | c | int |
| +16 | d | char |
| +12 | p | int\* |
| +8 | x | short |

最终，prototype如下：

int fun(short x, int\* p, char d, int c);

### 3.33

|  |  |
| --- | --- |
| Register | Value |
| %esp | 0x800040 |
| %ebp | 0x800060 |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| 0 | %ebp | Saved %edp | Stack frame for *proc* |
| -4 |  | x |
| -8 |  | y |
|  |  |  |
|  |  |  |
|  | %esp+8 | &x |
|  | %esp+4 | &y |
| -40 | %esp |  |

1. 在第3行，%ebp被赋值为%esp的初始值(0x800040)。
2. 在第4行，%esp的值被减去0x40，结果为0x800000。
3. 从第11和12行，知道x与y分别位于%ebp+4和%ebp+8，即0x800044，0x800048。
4. Scanf读入了0x46、0x53，分别存入到y与x。

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | caller |
|  |  |  |
| 0 | %ebp | Saved %edp | Stack frame for *proc* |
| -4 |  | x=0x53 |
| -8 |  | y=0x46 |
|  |  | unused |
|  |  |  |
| -32 | %esp+8 | &x |
| -36 | %esp+4 | &y |
| -40 | %esp |  | Stack frame for *scanf* |
|  |  |  |
|  |  |  |

1. 上图阴影部分未被proc使用。

### 3.34

本题考察对recursive函数调用的理解。

1. 根据Figure3.27，在执行第1行后，%ebx中的值是参数x。
2. 见代码rfun.c。关键是理解ASM。
3. 该函数将参数的所有bit相加。

### 3.35

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array | Element size | Total size | Start address | Element i |
| S | 2 | 14 |  |  |
| T | 4 | 12 |  |  |
| U | 4 | 24 |  |  |
| V | 8 | 64 |  |  |
| W | 4 | 16 |  |  |

根据[MSDN](https://msdn.microsoft.com/en-us/library/s3f49ktz.aspx)的资料，在VC++中，long double与double的size一样。

根据[GNU](http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html#Real-Number-Types)的资料，在GCC中，long double的size不小于float。

在本题，我们假设long double的size与double一样。

### 3.36

|  |  |  |  |
| --- | --- | --- | --- |
| Expression | Type | Value | Assembly code |
| S+1 | short\* |  | leal 1(%edx), %eax |
| S[3] | short |  | movw 3(%edx), %ax |
| &S[i] | short\* |  | leal (%edx, %ecx), %eax |
| S[4\*i+1] | short |  | movw 1(%edx, %ecx, 4), %ax |
| S+i-5 | short\* |  | leal -5(%edx, i), %eax |

### 3.37

本题考察二维数组的元素排列规则。

mat1和mat2的定义分别是：

结合公式(3.1)，可知

接着，我们分析ASM每个instraction的执行结果，如下

|  |  |
| --- | --- |
|  | i at %ebp+8, j at %ebp+12 |
| 1 | %ecx = i |
| 2 | %edx = j |
| 3 | %eax = 8i |
| 4 | %eax = 7i |
| 5 | %eax = j+7i |
| 6 | %edx = 5j |
| 7 | %edx = i+5j |
| 8 | %eax = |
| 9 | %eax = |

综合公式，可知

### 3.38

本题的关键是将二维数组看作一维序列。具体见代码fix\_set\_diag.c。

### 3.39

本题考察struct对象在内存中的表示。

1. 参考p.276底部的图，画出struct prob的布局，如下

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Offset | 0 | 4 |  | 12 |
| Contents | p | s.x | s.y | next |

1. 根据上图，struct prob对象的size是4bytes。
2. 先给出ASM的注释

|  |  |  |
| --- | --- | --- |
|  | sp at %ebp+8 |  |
| 1 | %eax = sp | 将%ebp+8中的值赋值到%eax，该值是sp。 |
| 2 | %edx = s.y | sp的值是其所指结构体第一个byte的地址。sp+8表示sp->s.y。 |
| 3 | s.x = s.y | sp+4表示获取sp->s.x。 |
| 4 | %edx = %eax+4 | %eax的值是\*sp的第一个byte的地址。%eax+4，表示s.x的地址。 |
| 5 | p = &s.x |  |
| 6 | next = sp | %eax = sp, %eax+12 = sp->next |

### 3.40

本题考察struct和union对象在内存中的布局。

|  |  |  |  |
| --- | --- | --- | --- |
| EXPR | TYPE | Code | 修正 |
| up->t1.s | int | movl 4(%eax), %eax |  |
|  |  | movl %eax, (%edx) |  |
|  |  |  |  |
| up->t1.v | short | movw 2(%eax), %ax | movw (%eax), %ax |
|  |  | movw %ax, (%edx) |  |
|  |  |  |  |
| &up->t1.d | short\* | leal 2(%eax), %eax |  |
|  |  | movl %eax, (%edx) |  |
|  |  |  |  |
| up->t2.a | int\* | movl %eax, (%edx) |  |
| = &up->t2.a[0] |  |  |  |
|  |  |  |  |
| up->t2.a[up->t1.s] | int | movl 4(%eax), %ecx |  |
|  |  | movl (%eax, %ecx, 4), %eax |  |
|  |  | movl %eax, (%edx) |  |
|  |  |  |  |
| \*up->t2.p | char | movl 8(%eax), %eax | movl 8(%eax), %eax |
|  |  | movl %eax, %edx | movb (%eax), %al |
|  |  |  | movb %al, (%edx) |

注释：由于不能在memory之间直接移动数据，因此，“movl 4(%eax), (%edx)”不合法。

### 3.41

本题考察结构体内部元素在内存中的alignment问题。具体分析见figures.xsl。

### 3.42

本题考察结构体内部元素在内存中的布局和alignment规则。

1. 按size的对元素进行分类。分析过程见figures.xsl。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Content | a | b | c | d | e | f | g | h |  |
| Offset | 0 | 4 | 8 | 16 | 20 | 24 | 32 | 40 | 48 |

1. 要注意byte序列的末尾要以8对齐。该结构的size是48。
2. 考虑到元素对齐的值均是2的倍数，那么，如果size较大的元素已经对齐，那么紧邻其后的元素必定也对齐。根据这一点，如果按照元素的size从大到小排列元素，就无需在序列中加入空白，从而可以实现总体size最小。但要注意，在序列的末尾，仍然要实现以8对齐。以下是offset

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Content | e | g | a | e | h | b | d | f |  |
| Offset | 0 | 8 | 16 | 20 | 24 | 28 | 30 | 31 | 33 |

### 3.43

本题考察在程序运行过程中出现的stack overflow问题。

1. 本小题要画出执行到第7行时，stack的情况

|  |  |  |
| --- | --- | --- |
|  | 08 04 86 43 | Return address |
| %ebp -> | 0xbffffc94 | Saved %ebp. |
| %edi -> | 0x3 |  |
| %esi -> | 0x2 |  |
| %ebx -> | 0x1 |  |
|  |  |  |

1. 本小题要画出执行到第10行时，stack的情况

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  | 08 04 86 43 | Return address |
| 0xbffffc94 – 0x28 = 0xbffffc6c | %ebp | 0xbffffc94 | Saved %ebp. |
|  | -4 | 0x3 | Saved %edi. |
|  | -8 | 0x2 | Saved %esi. |
|  | -c | 0x1 | Saved %ebx. |
|  | -10 |  |  |
| 0xbffffc6c – 14  = 0xbffffc59 | -14 |  |  |
|  | -18 |  |  |
|  | -1C |  |  |
|  | -20 |  |  |
|  | -24 |  |  |
| 0xbffffc6c – 28  = 0xbffffc45 | -28 | 0xbffffc59 | %esp |
|  | -2C |  |  |
|  |  |  |  |
|  |  |  |  |

1. 上图黄色区域是程序准备存储输入数据的位置。
2. 当输入字符串为“012345678901234567890123”时，该字符串的可见字符总长度是~~23~~24bytes。同时，在该字符串末尾有一个不可见字符NULL。因此，该字符串的总长度是25bytes。此时，callee-saved数据和return address最后一个byte均会被该字符串覆盖，导致被corrupted。

|  |  |
| --- | --- |
| 08 04 86 00 | Return address |
| 33 32 31 30 | Saved %ebp. |
| 39 38 37 36 | Saved %edi. |
| 35 34 33 32 | Saved %esi. |
| 31 30 39 38 | Saved %ebx. |
| 37 36 35 34 |  |
| 33 32 31 30 |  |

1. 根据答案提示，执行malloc时，参数应当在buf的length基础上再加1，以存储字符串末尾的NULL。

在本题，我出现以下错误：

1. 数错了字符串可见字符的数量
2. 忽略了字符串末尾的不可见字符NULL
3. 忽略了ASCII编码
4. 由于不了解strlen函数性质，不知道E的解答

### 3.44

本题考察stack address随机化。

1. 根据maximum与minimum的差值，range近似于。
2. Nop sled的作用，我理解为增加攻击性代码的size。一旦接触到Nop sled，就会将程序引导到攻击代码上。

我的理解还很模糊。

### 3.45

本题考察GCC stack protector facility。

A.先写出在without protector情况下的注释

|  |  |
| --- | --- |
| 1 | 扩展Stack |
| 2 | 获取参数x |
| 3 | 将x赋值到v |
| 4 | 获取变量v的地址 |
| 5 | 将v的地址放置到目标位置，准备调用iptoa |
| 6 | 计算数组buf的地址 |
| 7 | 将buf的地址放置到目标位置，准备调用iptoa |
| 8 | 调用iptoa |

根据以上分析，我们得知，在没有protector的情况下，stack frame的结构如下：

|  |  |
| --- | --- |
| +8 | x |
| +4 | Return address |
| 0， %ebp | Saved %ebp |
| -4 | Saved %ebx |
| -8 | v |
| -12 |  |
| -16 |  |
| -20 | buf |
| -24 |  |
| -28 |  |
| -32 | Address of v |
| -36，%esp | Address of buf |
|  |  |
|  |  |

以下是在with protector情况下的注释

|  |  |
| --- | --- |
| 1 | 扩展Stack |
| 2 | 获取canary |
| 3 | 将canary放入stack |
| 4 | 清空register |
| 5 | 获取参数x |
| 6 | 将x赋值到v |
| 7 | 获取变量v的地址 |
| 8 | 将v的地址放置到目标位置，准备调用iptoa |
| 9 | 计算数组buf的地址 |
| 10 | 将buf的地址放置到目标位置，准备调用iptoa |
| 11 | 调用iptoa |

在with protector的情况下，stack frame的结构如下：

|  |  |
| --- | --- |
| +8 | x |
| +4 | Return address |
| 0， %ebp | Saved %ebp |
| -4 | Saved %ebx |
| -8 | Canary |
| -12 |  |
| -16 |  |
| -20 | buf |
| -24 | v |
| -28 |  |
| -32 |  |
| -36 |  |
| -40 |  |
| -44 |  |
| -48 | Address of v |
| -52，%esp | Address of buf |
|  |  |

B. 将buf放到接近canary的地方，确实会更好。如果将buf放置在v下方，那么，当buf overflow后改变v的值但保持canary不变时，canary保护机制就没有作用。

### 3.46

本题考察对单位、技术发展趋势的理解。

1. 本题的关键是时间与单价的关系。问题等价于：什么时候，memory的价格会降到以下。

根据题意，从1980到2010年间，memory的价格从降低到了。假设每年价格与前一年的比值为x，有以下关系

可知。

现假设memory价格以相同趋势改变，在2010年的y年后，降低到，有以下关系

可知。也就是说，将在2034年达到这个价格。

1. 与上一小题相同。

现假设memory价格以相同趋势改变，在2010年的y年后，降低到，有以下关系

可知。也就是说，将在2062年达到这个价格。

1. ?？？

### 3.47

本题考察data movement instructions和type conversions。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| src\_t | dest\_t | Instruction | S | D | Explanation |
| long | long | movq | %rdi | %rax | No conversion |
| int | long | movslq | %edi | %rax | Sign-extended |
| char | long | movsbq | %dil | %rax | Sign-extended |
| unsigned int | unsigned long | mov | %edi | %rax | Extended double word to quad word |
| unsigned char | unsigned long | movzbq | %dil | %rax | Extended byte to quad word |
| long | int | movq | %rax | %edx | Curtaild quad word to double word |
| unsigned long | unsigned | movabsq | %rax | %edx | Curtaild quad word to double word |

修正

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| unsigned int | unsigned long | movl | %edi | %rax | Zero-extended |
| unsigned int | unsigned long | ? |  |  |  |
| long | int | movl | %edx | %edx | Zero-extended, 忽略%edx之外的bits |
| unsigned long | unsigned | movabsq | %rax | %edx | Curtaild quad word to double word |

我不熟悉从较大size到较小size的类型转换。

### 3.47

本题考察type conversions。解决方法是从ASM出发，确定参数的类型。

先给出注释

|  |  |  |
| --- | --- | --- |
| 1 | a, b, c and d are the appropriate regions of %rdi, %rsi, %rdx and %rcx | |
| 2 | | Sign extended *d,* from long(32-bit) to quad(64-bit) |
| 3 | | Compute *c\*d* |
| 4 | | Sign extended *b*, from byte to long |
| 5 | | Compute *a\*b* |
| 6 | | Sign extended *a\*b*, from long to quad |
| 7 | | Set *a\*b+c\*d* as return value |
| 8 | | Return |

根据以上分析，我们知道

|  |  |
| --- | --- |
| a | 32 bits |
| b | 8 bits |
| c | 64 bits |
| d | 32 bits |

因此，arithprob的原型可以是

long int arithprob(int a, char b, long int c, int d);

除此外，还有其他prototypes。

### 3.48

本题要求根据ASM写出C code，考察对相关instructions的理解。

先给出注释

|  |  |  |
| --- | --- | --- |
| 1 | x in %rdi | |
| 2 | | 设置val为0 |
| 3 | | 设置i为0 |
| 4 | | 设置常量 |
| 5 | .L2 | |
| 6 | | 获取参数x |
| 7 | | 与常量做&运算 |
| 8 | | 将运算结果保存val |
| 9 | | 对x做right shift运算 |
| 10 | | 循环变量i加1 |
| 11 | | 比较i与8 |
| 12 | | 如果<=，跳转到.L2 |
| 13 | |  |
| 14 | |  |
| 15 | |  |
| 16 | |  |
| 17 | |  |
| 18 | |  |
| 19 | |  |
| 20 | |  |
| 21 | |  |
| 22 | | 做mask运算，取出最后8个bit。由于最小的size是byte，因此不需要继续对折。 |
| 23 | |  |

代码是写出来了，但是不理解其内涵。原因是第7行代码看错了！！！

B. 该程序将一个quad变量的所有bits相加，得到一个数值，表示值为1的bit个数。这里的并行算法设计很有意思。

在第二章有一个程序，是对二进制序列所有bit做xor运算，最后得到一个bit。根据结果，我们可以判断序列中值为1的位是偶数个还是奇数个。

还有一个程序，要给出一个二进制序列第一个值为1的bit位的mask。我也是通过移动bit位的方式实现。

### 3.49

本题根据ASM和C函数的主体部分，写出C函数的prototype，其中可变部分是参数列表的顺序以及参数类型。

结合ASM的2、3、5行和C代码，x位于%edi，t位于%rdx，q位于%rsi。

根据argument registers的顺序，可以知道参数的顺序是x、q、t。

经过尝试，总结出以下两点

1. t是signed int\*
2. \*q的size是64bit

具体见代码incrprob.c。

### 3.50

本题考察type conversions对程序的影响。

具体见代码incrprob.c。

### 3.51

本题考察x86-64平台上，stack frame的使用规则。

1. 画出stack frame

|  |  |
| --- | --- |
| Stack pointer %rsp -> 0 |  |
| -8 |  |
| -16 | 7 |
| -24 | 5 |
| -32 | 3 |
| -40 | 2 |

1. 注释

|  |  |  |
| --- | --- | --- |
| 1 | i in %edi | |
| 2 | | a的第1个元素 |
| 3 | | a的第2个元素 |
| 4 | | a的第3个元素 |
| 5 | | a的第4个元素 |
| 6 | | 确保idx不会越界 |
| 7 | | (%rsp-40)表示第一个元素；%rdi的值是0、1、2和3，表示index；每个元素的size是64bits。因此，此处通过地址运算，获得相应元素。 |
| 8 | | 返回%rax |

1. 使用了stack pointer之外的memory，没有改变stack pointer。

### 3.52

本题考察x86-64平台下，stack frame的工作原理。

1. Rfact是递归函数，于是在执行过程中，需要重复利用registers。由于%rbx是callee-saved register，因此，需要在使用前将其值保存到stack中，这便是第2行和11行的作用。

在第一次调用rfact时，%rbx的值不确定。在递归调用rfact时，在第2行中，%rbx中的值为x+1，该值被保存到stack；在第3行，%rbx被赋值为x。同时，根据第9行以及对应的C代码可知，%rbx中的值为x。

1. Rfact是递归函数，于是在执行过程中，需要重复利用registers。由于%rbx是callee-saved register，因此，需要在使用前将其值保存到stack中，这是第2行的作用。在使用结束后，再从stack中恢复%rbx，这在第11行完成。

|  |  |  |
| --- | --- | --- |
|  | Argument: x in %rdi | |
| 1 | rfact: | |
| 2 | | 将值保存到stack |
| 3 | | 获取参数x，保存到%rbx |
| 4 | | 将返回值设为1 |
| 5 | | 比较x与0 |
| 6 | | 若x小于等于0，跳转到.L11，返回值为1 |
| 7 | | 令xml=x-1 |
| 8 | | 以xml为参数，递归调用rfact |
| 9 | | 将递归调用的返回值乘上x，做为函数返回值 |
| 10 | .L11 | |
| 11 | | 从stack中获取变量，恢复%rbx的值 |
| 12 | | 返回函数 |

1. 在之前的情况中，我们通过改变stack frame pointer的值管理stack frame。在这里，我们使用push和pop instructions。

### 3.53

本题考察在x86-64平台上，数据类型的size以及alignment问题。尤其注意当结构体中包含数组时，alignment有数组元素决定。

本题采用与3.41相同的方法解决，具体见figures.xls。

### 3.54

本题考察对instructions的理解，要求根据ASM写出对应的C代码。

### 3.55

本题考察对mov和arithmetic instructions的理解，以及和C代码的对应关系。

我要做两件事情：描述ASM的算法、添加注释。

算法：先对y进行sign extended。然后依据下列公式，针对x与y 的不同部分计算乘积，然后组合结果，等价于x\*y。

|  |  |  |
| --- | --- | --- |
|  | 64 bit | 32 bit |
|  |  |  |
| \* |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

结合以上公式，给出注释：

|  |  |
| --- | --- |
|  |  |
| 1 | 获取 |
| 2 | 获取y |
| 3 | 将y保存到%edx |
| 4 | 对%edx做arithmentic right shift，得到 |
| 5 | 移动 |
| 6 | 计算 |
| 7 | 获取 |
| 8 | 计算 |
| 9 | 计算 |
| 10 | 计算 |
| 11 | 计算 |
| 12 | 获取dest的值 |
| 13 | 将保存到memory |
| 14 | 将保存到memory |

### 3.56

本题考察对conditional instructions的理解。

基本性质：

1. 值不变的register，对应C中的不变量
2. test指令的operand对应循环变量
3. %eax值的来源，对应返回变量
4. 根据ASM的1、2行，容易确定x对应%esi，y对应%ebx。根据第3条性质，以及ASM第13行，可以确定result对应%edi。根据第2条性质，以及ASM第11行和C第5行，可以确定mask对应%edx。
5. result值保存在%edi中，根据ASM第3行，初始值是1431655765；

result值保存在%edi中，根据ASM第3行，初始值是1431655765；

1. 根据ASM第11、12行，可知test condition是mask<=0
2. 在ASM的循环主体中，只有第10行改变了%edx的值。结合第9、10两行，mask的update方式是

mask = mask >> ((char) n);

1. 在ASM的循环主体中，只有第8行改变了%edi的值。结合第6、7、8行，result的update方式是

result ^= x & mask;

1. 见代码。