# 计算机系统导论 知识点总结

### 信息的表示和处理

1. 计算机采用8位的块（字节,Byte）作为最小可寻址的内存单位，机器级程序将内存视为一个非常大的字节数组，称为虚拟内存，内存的每个字节由一个唯一的数字来标识，称为它的地址，所有可能地址的集合称为虚拟地址空间。
2. 编译器运行时系统会将存储器划分为更可管理的单元，来存储不同的程序对象，这种管理完全是在虚拟地址空间里完成的。
3. C语言中一个指针的值都是某个存储块的第一个字节的虚拟地址。
4. 每个程序对象可以简单地视为一个字节块，而程序本身就是一个字节序列。
5. 在C语言中，以0x或0X开头的数字常量被认为是16进制的值，字母a~f既可以是大写，也可以是小写，甚至可以是大小写混合。
6. 每台计算机都有一个字长（word size），字长决定了虚拟地址空间的最大大小，即对于一个字长为w位的机器来说，虚拟地址的范围为0~2^w-1，程序最多访问2^w个字节。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| C声明 | | 字节数 | | 取值范围（上32位，下64位） | |
| 有符号 | 无符号 | 32位 | 64位 | 有符号 | 无符号 |
| [signed] char | unsigned char | 1 | 1 | -128~127  -128~127 | 0~255  0~255 |
| short | unsigned short | 2 | 2 | -32768~32767  -32768~32767 | 0~65535  0~65535 |
| int | unsigned int | 4 | 4 | -2147483648~2147483647  -2147483648~2147483647 | 0~4294967295  0~4294967295 |
| long | unsigned long | 4 | 8 | -2147483648~2147483647  -9223372036854775808~9223372036854775807 | 0~4294967295  0~18446744073709551615 |
| int32\_t | uint32\_t | 4 | 4 | -2147483648~2147483647  -2147483648~2147483647 | 0~4294967295  0~4294967295 |
| int64\_t | uint64\_t | 8 | 8 | -9223372036854775808~9223372036854775807  -9223372036854775808~9223372036854775807 | 0~18446744073709551615  0~18446744073709551615 |
| char\* |  | 4 | 8 | 略 | 略 |
| float |  | 4 | 4 | 略 | 略 |
| double |  | 8 | 8 | 略 | 略 |

7.大多数64位机器也可以运行32位机器编译的程序，这种特性称为向后兼容，我们所说的“xx位程序”区分的是编译方式而不是其运行的机器。

8.上表中int32\_t与int64\_t是确定大小的整数类型，不随编译器和机器设置而变化；大部分数据类型默认为有符号的，但char不保证这一点，很多情况下程序对char是否有符号并不敏感。

9.大多数机器中多字节对象被存储为连续的字节序列，对象的地址为所使用字节中最小的地址，而存储的规则有两种：大端法与小端法。大端法即最高有效字节在最前面的方式，即最高有效字节所处的地址较小，这称为大端法；小端法即最高有效字节在最后面的方式，即最高有效字节所处的地址较大。大多数Intel兼容机采用的是小端模式，而大多数IBM和Oracle采取的是大端模式。但在网络应用程序中，为了避免字节出现反序，网络应用程序的代码编写必须最受已建立的关于字节顺序的规则，以确保发送方机器将它的内部表示转换成网络标准，而接收方及其将网络标准转换为它的内部表示。

10.在使用ASCII码作为字符码的任何系统上都将得到相同的结果，与字节顺序和字大小规则无关，于是文本数据比二进制数据有更强的平台独立性；相应地，二进制代码是不兼容的，二进制代码很少能在不同的机器和操作系统之间移植。

11.可以通过执行命令man ascii来得到一张ASCII字符码的表

12.C语言中字符串被编码为一个以null（0）为结尾的字符数组，每个字符都以某个标准编码表示，最常见的是ASCII字符码。在使用ASCII码作为字符码的任何系统上都将得到相同的结果，与字节顺序和字大小规则无关，因而文本数据比二进制数据具有更强的平台独立性。不同的机器类型使用不同的且不兼容的指令和编码方式，即使是完全一样的进程，运行在不同的操作系统上也会有不同的编码规则，因此二进制代码是不兼容的，二进制代码很少能在不同的机器和操作系统组合之间移植。

13.计算机系统的一个基本概念是从机器的角度来看，程序仅仅只是字节序列。

14.布尔代数的运算规则：a&(b|c)=(a&b)|(a&c)，a|(b&c)=(a|b)&(a|c)，(a^b)^a=b

15.位级运算的一个常见用法就是实现掩码运算，如表达式~0将生成一个全1的掩码，若在32位机上使用0xFFFFFFFF，这样的代码是不可移植的。

16.只有当参数限制为0或1时按位运算才与对应的逻辑运算有相同的结果

17.如果对第一个参数求值就能确定表达式的结果，逻辑运算就不会对第二个参数求值。

18.因为运算是从左至右可结合的，即x<<j<<k等价于(x<<j)<<k；逻辑右移在最高位补0，算术右移在左侧补最高位的值。

19.几乎所有编译器都对有符号数使用算术右移（左侧补最高位），对于无符号数必须使用逻辑右移（左侧补0）；特别地，在java中，x>>k执行的是算术右移，而x>>>k执行的是逻辑右移。

20.如果移位的位数超过了数据类型的大小，这样的行为是没有定义的，因而不应采用。

21.补码编码：设机器的字长为w，则x=(-xw-1)2w-1+∑xk2k，补码的范围是不对称的，|Tmin|=|Tmax|+1=-2w-1，这里的原因是一半的位模式表示负数，另一半表示非负数，由于非负数区多了0，所以表示不对称；Umax=2|Tmax|+1=2|Tmin|-1=2w-1，而-1和Umax有相同的位表示，0的位表示在这些情况下均唯一。

22.C语言标准并没有要求要用补码形式来表示有符号正数，但几乎所有的机器都是这么做的，程序员如果希望代码有最大的可移植性，能够在所有可能的机器上运行，那么除上述范围外我们不应假设任何可表示的数据范围，也不应假设有符号数会使用任何特殊的表示方式。另一方面，许多程序的书写都假设用补码来表示有符号数，并且具有上表中的典型的取值范围，如C语言中的<limits.h>库定义了一组常量来描述有符号与无符号整数的范围

23.ISO C99标准在文件<stdint.h>中引入了intN\_t和uintN\_t来定义N位有符号和无符号整数，使用宏能保证，不论代码是如何编译的，都能生成正确的格式字符串。

24.对于整数数据类型的取值范围和表示，Java的标准是非常明确的，要求采用补码表示，取值范围与上表中64位情况相同，在Java中，单字节的数据类型为byte而非char，这些要求保证了无论在什么机器上，Java程序都能表现一致。

25.另外的编码方式：原码（用最高有效位表示符号用以确定剩余部分的正负，剩余部分作为无符号数计算出绝对值）；反码（最高有效位权值为2w-1-1，其余与补码相同）；这两种表示方式对0都有两种表示方式，在浮点数中有使用原码编码

26.强制类型转换并不改变位值，只改变了解释这些位的方式。

27.通常大多数数字都默认是有符号的，要创建无符号常量必须加上后缀字符U或u

28.当执行一个运算时，如果两个参数一个是有符号的而一个是无符号的，那么C语言会隐式地将有符号数转变为无符号数参与运算，并假设这两个数都是非负的，于是对于诸如大小比较的运算可能会导致错误。

29.在C语言中，TMin必须写成-2147483647-1而不是-2147483648或0x80000000，补码表示的不对称性和C语言的转换规则之间奇怪的交互，迫使我们用这种不寻常的方式来写TMin

30.要将一个无符号数转换为一个更大的数据类型，我们只要简单地在表示的开头添加0，这种运算称为零扩展；要将一个补码数字转换为一个更大的数据类型，可以执行一个符号扩展，在表示中添加最高有效位的值。

31.从一个数据大小到另一个数据大小的转换，以及无符号和有符号数之间的转换的相对顺序能够影响一个程序的行为，如将short转换为unsigned时，我们要先改变大小，之后再完成有符号到无符号之间的转换，也就是说(unsigned) a <=> (unsigned int) (int) a

32.将一个较长的数字截断为一个较短的数字时，我们会丢弃其高位中超过的部分，截断一个数字可能会改变它的值——溢出的一种形式，对于一个无符号数很容易，对于一个补码编码截断原理相同，而截断后的最高位要解释为符号位。

33.需要注意到有符号数与无符号数之间的隐式转换会带来一些非直观的结果，有一些隐式的无符号数，比如size\_t，sizeof，strlen等定义或计算出的结果均为unsigned结果，在代码中出现值得注意。避免这类错误的一种方法是绝不使用无符号数，实际上，除了C语言以外很少有语言支持无符号正数。但当我们要把字仅仅看做是位的集合而没有任何数学意义时，无符号数值是非常有用的；当实现模运算和多精度运算的数学包时，数字是由字的数组来表示的，无符号值也会非常有用。

34.要想完整地保持算数运算的结果，我们不能对字长做任何限制，一些编程语言如Lisp实际上支持无限精度的运算，允许任意的（在内存范围内的）整数运算，但更常见的是编程语言支持固定精度的运算，因此加法和乘法这样的运算是不同于其在整数上的相应运算的。

35.无符号数的加法是将两个数相加的结果截断为对应字长的结果，对于两个无符号数相加，若结果小于这两个加数，则说明发生了溢出（充要条件）。

36.模数加法形成了一种名为阿贝尔群的数学结构，满足可交换、可结合、有零元和逆元（其中w位无符号数x的逆元是2w-x，0的逆元是0）

37.补码加法是将两个数相加后截断为对应长度的结果，如果两个正数相加得到负数称为正溢出（截断的结果是原结果-2w），如果两个负数相加得到正数称为负溢出（截断的结果是原结果+2w）。两个数相加按无符号相加和按补码相加有完全相同的位表示法，大多数计算机使用相同的机器指令来执行无符号或有符号加法。需要注意有符号加法是阿贝尔群，即若x+y=s，则一定有s-x=y,s-y=x。

38.在补码之中，每个数都有相反数，其中TMin+TMin=0，其余数的相反数为-x；计算一个补码的相反数第一种方法是每一位取反后+1，在C语言中-x=~x+1；第二种方法是先找，到二进制表示中最右边的1，然后将这一位左侧的所有位（不包括这一位）取反。

39.无符号数的乘法定义为两个数相乘的结果截断为对应的字长，而补码同理，但需要注意的是无符号乘积和补码乘积的完整的位表示不同，但是截断后的位表示相同。

40.检查补码乘法x\*y=p是否溢出只需判断x=0或p/x=y即可。

41.在大多数机器上，乘法运算相当慢，在本书的参考机器上乘法需要3个时钟周期，而除法更慢，需要30个或更多的时钟周期。

41.左移一个数值等价于执行一个与2的幂相乘的无符号乘法，即使在溢出的时候，我们通过移位得到的结果也是一样的。在乘以常数时，将一个乘以常数的运算替换为多个移位和加法，无论是无符号还是补码甚至在溢出时都会得到一样的结果，更好的情况是可以用减法来减少移位和加法的次数。大多数编译器只在需要少量移位、加法和减法运算就足够的时候才使用这种优化。

42.除以2的幂可以用右移来实现，无符号数和补码分别用逻辑移位和算术移位来达到目的，表达式x>>k得到数值x/2k（向下取整），即向下舍入一个数，对于正数这没有问题，对于负数在不需要舍入的情况没有影响，在需要舍入时会将负数向下舍入（如-771.25会被舍入为-772），于是欲使结果向上舍入时，需要加上偏置量bias，即(x+(1<<k)-1)>>k；表明对于使用算术右移的补码机器，在C表达式中，(x<0?(x+(1<<k)-1):x)>>k==x/2k，但是和乘2的幂不同，我们不能用除以2的幂的除法来表示除以任意常数的除法。

43.目前所有的计算机都支持IEEE浮点的标准，这大大提高了科学应用程序在不同机器上的可移植性。

44.形如0.1111111...2的数表示的是刚好小于1的数，可以表示为1.0-ε；很多小数不能用二进制小数精确表示，只能近似地表示，增加二进制表示的长度可以提高表示的精度。

45.IEEE的浮点标准用V=(-1)s×M×2E的形式表示一个数，其中s为符号位，s=1表示为负数，s=0表示为正数，数值0的符号位解释为特殊情况处理；尾数M是一个二进制小数，其范围是1~2-ε或0~1-ε；阶码E的作用是为浮点数加权，这个权重是2的E次幂（可能是负数）。标准将浮点数的位表示划分为3个字段，分别对这些值进行编码：单独的符号位s直接编码s，k位的阶码字段编码阶码E，n位小数字段frac=fn-1...f1f0编码尾数M，但编码出来的值依赖于阶码位是否为0。

46.在单精度浮点格式（C语言的float）中s,exp,frac的字段分别为1位，8位和23位，得到一个32位的表示；在双精度浮点格式（C语言的double）中，s,exp,frac的字段分别为1位，11位和52位，得到一个64位的表示。

47.根据阶码的情况可以将被编码的值分成三种不同的情况：

情况1：规格化的值。当exp的值既不全为0也不全为1时，都属于这种情况，这种情况下阶码字段被解释成以偏置量bias表示的有符号整数，阶码的值是E=e-bias，其中e是阶码编码出的无符号值，bias=2k-1-1（单精度为127，双精度为1023），由此产生指数E的取值范围：单精度-126~+127，双精度-1022~+1023；小数字段frac解释为小数值f，其中f∈[0,1)，而尾数M=1+f，这种方式称为隐含的1开头表示，可以获得一个额外的精度位

情况2：非规格化的值。当exp的值全为0时，表示的数是非规格化的，此时阶码的值为E=1-bias（这提供了一种从非规格化值平滑转换到规格化值的方法），而尾数M=f。非规格化的值提供了表示数值0的方法，+0.0的浮点表示的位模式为全0，于是M=f=0，但符号位为1时会得到-0.0，这两种表示在某些方面被认为是不同的，在其他方面是相同的。

情况3：特殊值。当exp的值全为1时，若小数域全为0，得到的值表示无穷，当s=0时为+∞，当s=-1时得到的值为-∞，当我们把两个大数相乘或除以0时，无穷能表示溢出的结果；当小数域非0时，结果值称为NaN，一些运算的结果不能是实数或无穷，就会返回NaN，比如√-1或∞-∞时，在某些应用中，表示未初始化的数据时，它们也很有用处。

1. 可以看到，在数值较小时，能精确表示的数值很稠密；当数值较大时，能精确表示的数据点比较稀疏。
2. 在非负的情况下，最小的非规格化数为0，最大的非规格化数为0 0000（阶码全为0） 11111（小数部分全为1），最小的规格化数为0 00001（阶码最后一位为1）00000（小数部分全为0），最大的规格化数为0 111110（阶码最后一位为0） 111111（小数部分全为1），如果我们将这个位表示解释为无符号数，他们是按升序排列的，浮点数也是按升序排列的。
3. 有k位阶码和n位小数的浮点表示有如下一般属性：
4. +0.0总有一个全为0的位表示
5. 最小的正非规格化的值的表示M=f=2-n和E=-2k-1+2，于是V=2-n-2^(k-1)+2
6. 最大的非规格化的值的表示M=f=1-2-n和E=-2k-1+2，于是V=22-2^(k-1)(1-2-n)
7. 最小的正规格化的值的表示M=1，f=0，E=-2k-1+2，于是V=22-2^(k-1)
8. 值1.0的位表示的阶码字段除最高有效位为0以外其余所有位都等于1，其尾数值M=1，f=0
9. 最大的规格化的值的小数值f=1-2-n，尾数M=2-2-n，阶码值E=2k-1-1，于是V=(1-2-n-1)22^(k-1)
10. 可以通过将一个正整数的二进制表示的小数点左移适当的位数得到这个数的规格化表示，然后按IEEE标准构造即可。整数与对应浮点数的位级表示在等于1的最高有效位之前停止，和浮点的小数部分的高位是匹配的。
11. 向偶数舍入，也称为向最近的值舍入，是默认的方式，其采用的方法是它向数字向上或向下舍入，使得结果的最低有效数字是偶数，其他三种方式（向零舍入，向下舍入和向上舍入）在一些数值处理之中相当有用。向偶数舍入在大多数现实情况中可以避免统计误差，向偶数舍入应用在二进制小数上就是认为最后一位为0表示偶数，最后一位为1表示奇数，比如在舍入的过程中会将0.101和0.011都舍入到0.10。
12. 浮点运算会将浮点值看成实数，在实数上计算出结果后进行舍入得到最终的结果，定义1/0=+∞，1/-0=-∞，实数上的加法也形成了阿贝尔群，这个运算是可交换的但是不可结合的，大多数值在浮点加法的意义下都有负元，但是无穷和NaN是例外情况，因为x+NaN=NaN
13. 浮点加法满足了单调性属性，即若a≥b，则x+a≥x+b（除了NaN），而无符号或补码加法不满足这样的单调性。
14. 浮点乘法在乘法中是封闭的，可交换的但不可结合（可能发生溢出或舍入失去精度），有单位元1.0，浮点乘法在加法上不具有分配性，浮点乘法有单调性：a≥b，c≥0，则a\*c≥b\*c；a≥b，c≤0，则a\*c≤b\*c；a\*a≥0（上述规则NaN除外）；无符号或补码的乘法没有这些单调性属性。缺乏结合性和分配性是很严重的问题。
15. C语言版本提供两种不同的浮点数据类型float和double，在支持IEEE浮点格式的机器上这些数据类型对应于单精度和双精度浮点，这类机器使用向偶数舍入的舍入方式。但C语言不要求机器使用IEEE浮点，因此没有标准的方法改变舍入方式或得到-0，+∞，-∞，NaN之类的特殊值，大多数系统提供include（‘.h’）文件和读取这些特征的过程库，但是细节随系统的不同而不同。
16. 当在int、float和double格式之间进行强制类型转换时，程序改变数值和位模式的原则如下：
17. int转变为float，数字不会溢出但可能被舍入
18. int或float转变为double，数字既不会溢出也不会被舍入
19. double转变为float值可能溢出成+∞或-∞，还可能被舍入
20. float或double转变为int值会向零舍入，同时值可能会溢出，C语言标准对这种状况没有作出规定，与Intel兼容的微处理器指定位模式[100...0000]（TMin）为整数不确定值，一个浮点数到整数的转换如果不能为该浮点数找到一个合理的整数近似值就会产生这样一个值
21. 程序的机器级表示
22. 计算机执行机器代码，用字节序列编码低级的操作，包括处理数据、管理内存、读写存储设备上的数据，以及利用网络通信。编译器基于目标语言的规则、目标机器的指令集和操作系统遵循的惯例，经过一系列的阶段生成机器代码。GCC C语言编译器以汇编代码的形式产生输出，汇编代码是机器代码的文本表示，给出程序中的每一条指令，然后GCC调用汇编器和链接器，根据汇编代码生成可执行的机器代码。
23. 通常情况下，使用现代的优化编译器产生的代码至少与一个熟练的汇编语言程序员手工编写的代码一样有效，但用高级语言编写的程序可以在很多不同的机器上编译和执行，而汇编代码则是与特定机器密切相关的。
24. 每个后继处理器的设计都是后向兼容的——较早版本上编译的代码可以在较新的处理器上运行。
25. 编译选项-Og告诉编译器使用会生成符合原始C代码整体结构的机器代码的优化等级（GCC 4.8引入，在较早版本的GCC和一些非GNU的编译器中使用-o1可能是最好的选择）
26. gcc命令调用了一整套程序，将源代码转化成可执行代码：C预处理器扩展源代码，插入所有用#include命令指定的文件和扩展所有用#define声明指定的宏->编译器产生源文件的汇编代码，命名为.s->汇编器会将汇编代码转化成二进制目标代码文件.o，目标代码是机器代码的一种形式，它包含所有指令的二进制表示，但还没有填入全局值的地址->链接器将目标代码文件与实现库函数的代码合并，并产生最终的可执行代码文件（由命令行提示符-o 指定的）
27. 计算机系统使用了多种不同形式的抽象，利用更简单的抽象模型来隐藏实现的细节。对于机器级编程而言，其中有两种抽象尤为重要，第一种是指令集体系结构/指令集架构（ISA）来定义机器级程序的格式和行为，它定义了处理器状态、指令的格式以及每条指令对状态的影响。大多数ISA（包括x86-64），将程序的行为描述成好像每条指令都是按顺序执行的，一条结束后下一条再开始，但处理器的硬件远比描述的更精细复杂，它们并发地执行许多指令，但是可以采取措施保证整体行为与ISA指定的顺序执行的行为完全一致。第二种抽象是，机器级程序使用的内存地址是虚拟地址，提供的内存模型看上去是一个非常大的字节数组。存储器系统的实际实现是将多个硬件存储器和操作系统软件组合起来。
28. x86-64的机器代码和原始的C代码的差别非常大，一些通常对C语言程序员隐藏的处理器状态都是可见的：
29. 程序计数器（PC，x86-64中用%rip表示）给出将要执行的下一条指令在内存中的地址
30. 整数寄存器文件包含16个命名的位置，分别存储64位的值，这些寄存器可以存储地址（对应于C语言的指针）或整数数据。
31. 条件码寄存器保存着最近执行的算术或逻辑指令的状态信息，它们用来实现数据流中的条件变化，比如说用来实现if和while语句。
32. 一组向量寄存器可以存放一个或多个正数或浮点数值。
33. 机器代码只是简单地将内存看做一个很大的、按字节寻址的数组。C语言中的聚合数据类型（数组和结构）在机器代码中用一组连续的字节表示，即使对标量数据类型，汇编代码也不区分有符号或无符号整数，不区分各种类型的指针，甚至不区分指针和整数。
34. 程序内存包含：程序的可执行机器代码，操作系统需要的一些信息，用来管理过程调用和返回的运行时栈，以及用户分配的内存块（比如用malloc库函数分配的）。在任意给定的时刻，只有有限的一部分虚拟地址被认为是合法的，比如x86-64的虚拟地址是由64位的字来表示的，在目前的实现中，这些地址的高16位必须设置为0，所以一个地址实际上能够指定的是248或256TB范围的一个字节。操作系统负责管理虚拟地址空间，将虚拟地址翻译成实际处理器内存中的地址。
35. 在命令行上使用“-S”选项就能看到C语言编译产生的汇编代码：“gcc -Og -S xxx.c”
36. 如果使用“-c”命令行选项，GCC会编译并汇编改代码：“gcc -Og -c xxx.c”，产生.o文件，它是二进制格式的，所以无法直接查看。
37. 机器执行的程序只是一个字节序列，它是对一系列指令的编码，机器对产生这些指令的源代码几乎一无所知。
38. 要展示程序的二进制目标代码，我们要先用反汇编器确定该过程的代码字节长度为p，然后在.o文件上运行GNU调试工具GDB，输入命令x/pxb func，这条命令告诉GDB显示（简写为x）从函数func所处地址开始的14个十六进制格式表示（也简写为x）的字节（简写为b）。
39. 要查看机器代码文件的内容，有一类称为反汇编器的程序非常有用，这些程序根据机器代码产生一种类似于汇编代码的格式。在Linux系统中，带“-d”命令行标志的程序OBJDUMP（表示“object dump”）可以充当这个角色。
40. x86-64的指令长度从1到15个字节不等，常用的指令以及操作数较少的指令所需的字节数少，而那些不太常用或操作数较多的指令所需字节数较多。
41. 设计指令格式的方式是，从某个给定位置开始，可以将字节唯一地解码成机器指令。
42. 反汇编器知识基于机器代码文件中的字节序列来确定汇编代码，它不需要访问该程序的源代码或汇编代码。
43. 反汇编器使用的指令命名规则和GCC生成的汇编代码使用的有细微的差别。它会省略一些指令结尾的q而回给call和ret这样的指令后面加上q
44. 生成实际可执行的代码需要对一组目标代码文件运行链接器，而这一组目标代码文件中必须含有一个main函数，链接器会将代码段的地址移到不同的地址范围之中，还会为函数调用找到匹配的函数的可执行代码的位置，还会插入一些指令使函数代码变为16字节，使得就存储器系统性能而言，能更好地放置下一个代码块。
45. 所有以“.”开头的都是指导汇编器和链接器工作的指令，我们通常可以忽略这些行，另一方面，也没有关于指令的用途以及它们与源代码之间的关系的解释说明。
46. 每一行的左边有编号供引用，右边是注释，简单地描述指令的效果以及它与原始C语言代码中计算操作的关系，这是一种汇编语言程序员写代码的风格。
47. 我们学习的是ATT格式的汇编代码，这是GCC、OBJDUMP和一些其他工具的默认格式，但其他工具，包括Microsoft的工具和Intel的文档其汇编是Intel格式的，在许多方面有所不同，使用命令“-masm=intel”可以生成Intel格式的汇编代码，其与ATT格式的区别如下：
48. 省略指示大小的后缀
49. 省略寄存器名字前面的%
50. 用不同的方式来描述内存中的位置，如QWORD PTR [rbx]而不是(%rbx)
51. 在带有多个操作数的指令情况下，列出操作数的顺序相反。
52. 每次x86-64的处理器执行算术或逻辑运算后，如果得到的运算结果的低8位中有偶数个1，就会把一个名为PF的1位条件码设置成1，否则会设置成0。
53. 在C语言中插入汇编代码有两种方法，第一种是我们可以编写完整的函数，放进一个独立的汇编代码文件中，让汇编器和链接器把它和用C语言编写的代码合并起来；第二种是我们可以使用GCC的内联汇编特性，用asm伪指令可以在C语言中包含简短的汇编代码，这种方法的好处是减少了与机器相关的代码量，当然，在C程序中包含汇编代码使得这些待遇与某类特殊的机器相关（如x86-64），所以只应该在想要的特性只能以此方法才能访问到时使用它。
54. Intel用术语“字”（word）表示16位数据类型，因此称32位数为双字，64位数为四字。
55. x86家族的历史上实现过对一种特殊的80位（10字节）的浮点格式进行全套的浮点运算，可以在C程序中用声明long double来指定这种格式，不过我们不建议使用这种格式，它不能移植到其他类型的机器上，而且实现的硬件也不如单精度和双精度算术运算的高效。
56. 汇编代码使用后缀l表示8字节双精度浮点数和s表示4字节单精度浮点数，这不会产生歧义因为浮点数使用的是一组完全不同的指令和寄存器。
57. 寄存器的名称及作用

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4字（64bit，8Byte） | 2字（32bit，4Byte） | 1字（16bit，2Byte） | 1字节（8bit，1Byte） | 功能与特性 |
| %rax | %eax | %ax | %al | 存储返回值（调用者保存） |
| %rbx | %ebx | %bx | %bl | 帧指针，被调用者保存 |
| %rcx | %ecx | %cx | %cl | 第4个参数（调用者保存） |
| %rdx | %edx | %dx | %dl | 第3个参数（调用者保存） |
| %rsi | %esi | %si | %sil | 第2个参数（调用者保存） |
| %rdi | %edi | %di | %dil | 第1个参数（调用者保存） |
| %rbp | %ebp | %bp | %bpl | 被调用者保存 |
| %rsp | %esp | %sp | %spl | 栈指针 |
| %r8 | %r8d | %r8w | %r8b | 第5个参数（调用者保存） |
| %r9 | %r9d | %r9w | %r9b | 第6个参数（调用者保存） |
| %r10 | %r10d | %r10w | %r10b | 调用者保存 |
| %r11 | %r11d | %r11w | %r11b | 调用者保存 |
| %r12 | %r12d | %r12w | %r12b | 被调用者保存 |
| %r13 | %r13d | %r13w | %r13b | 被调用者保存 |
| %r14 | %r14d | %r14w | %r14b | 被调用者保存 |
| %r15 | %r15d | %r15w | %r15b | 被调用者保存 |

1. 对于生成小于8字节结果的指令，生成1字节和2字节的指令会保持剩下的字节不变，生成4字节数字的指令会把高位4个字节设置成0.
2. 大多数指令有一个或多个操作数，指示出执行一个操作中要使用的源数据值，以及放置结果的目的位置。源数据值可以以常数形式给出，或是从寄存器或内存中读出，结果可以存放在寄存器或内存中。各种不同的操作数被分为三种类型，第一种是立即数，用来表示常数值，在ATT格式中，立即数用$+C表示法表示的数字。不同的指令允许的立即数数值范围不同，汇编器会自动选择最紧凑的方式进行数值编码。第二种类型是寄存器，它表示某个寄存器的内容，16个寄存器的低位1字节、2字节、4字节或8字节中的一个作为操作数，这些字节数分别对应于8、16、32、64位。第三种类型是内存引用，它会根据计算出来的地址（通常称为有效地址）访问某个内存位置。用ra表示寄存器，R[ra]表示寄存器里的值，用Add表示地址，M[add]表示内存中地址add处的值
3. 有不同的寻址模式，其中Imm(rb,ri,s)是最常用的，表示M[R[rb]+s\*R[ri]+Imm]，其中s只能是1,2,4或8，其余情况都是这种情况的特殊情况。
4. 数据传送指令可以将数据从一个位置复制到另一个位置。MOV类数据传送指令可以将数据复制到目的位置而不做任何变化。指令为MOV S D，其中源操作数指定的值可以是一个立即数或一个存储在寄存器或内存中的数，目的操作数指定一个位置（寄存器或内存地址）。
5. x86-64增加了限制，传送指令的操作数不能都指向内存位置，于是将一个值从一个内存地址复制到另一个内存地址需要两条指令——一条指令将源值加载到寄存器中，另一条将寄存器写入目的位置。
6. 这些指令的寄存器操作数可以是16个寄存器有标号部分中的任意一个，寄存器部分的大小必须与指令最后一个字符指定的大小匹配，大多数情况下MOV指令只会更新目的操作数指定的那些寄存器字节或内存位置，但movl指令以寄存器作为目标时会把该寄存器的高位4字节设置为0。
7. 常规的movq指令只能以表示为32位补码数字的立即数作为源操作数，然后把这个值符号扩展得到64位的值，放到目的位置，movabsq指令能够以任意64位立即数作为源操作数，并且只能以寄存器作为目的。
8. 在将较小的源值复制到较大的目的时使用两类MOVZ（零扩展）和MOVS（符号扩展）指令，每条指令的最后两个字符都是大小指示符，第一个指定源的大小，第二个指定目的的大小。
9. 并没有一条明确的指令把4字节源值零扩展到8字节目的，因为这可以通过movl的特性实现。
10. cltq指令总是以%eax为源，%rax为目的进行符号扩展，其效果与指令movslq %eax，%rax完全一致，但编码更紧凑。
11. 数据传送指令：

b（1 Byte）,w(2 Byte),l(4Byte),q(8 Byte)，表格中x,y为标识符

|  |  |
| --- | --- |
| mov+x A B | A-->B |
| movz+x y A B | A(零扩展）-->B |
| movs+x y A B | A(符号扩展)-->B |
| movabsq A B | A-->B（绝对四字，B仅寄存器） |
| cltq %eax %rax | %eax(符号扩展）-->%rax |

1. C语言中所谓的“指针”其实就是地址，间接引用指针就是将该指针放在一个寄存器中，然后在内存引用中使用这个寄存器。
2. 局部变量通常保存在寄存器中而不是内存中，访问寄存器比访问内存要快得多。
3. 弹出的值永远是最近被压入且仍然在栈中的值；栈向下增长，栈顶元素的地址是所有栈中元素地址最低的；无论如何，%rsp的值总是指向栈顶。
4. 栈操作：（pushq编码为1个字节，而替代的两条指令要编码为8个字节）

|  |  |  |
| --- | --- | --- |
| pushq S | R[%rsp]-8 ->R[%rsp]  S -> M[R[%rsp]] | 将4字压入栈 |
| popq D | M[R[%rsp]]-> D  R[%rsp]+8->R[%rsp] | 将四字弹出栈 |

1. 因为栈和程序代码以及其他形式的程序数据都是放在同一内存中，所以程序可以用标准的内存寻址方式访问栈内的任意位置。
2. 加载有效地址leaq指令形式是将内存读数据到寄存器，但它根本没有引用内存，它是将第一个操作数的有效地址写入目的寄存器，在C语言中可以用&来描述这种运算，除此之外还可以用来进行简单的计算，编译器经常发现leaq的一些灵活运算，根本与有效地址运算无关，其目的操作数必须是一个寄存器。
3. 二元操作的第二个操作数既是源又是目的，第一个操作数可以是立即数、寄存器或内存位置，第二个操作数可以是寄存器或内存位置，当第二个操作数为内存地址时，处理器必须从内存读出值，执行操作，再把结果写回内存。
4. 移位操作想给出移位量，第二项是要移位的数，移位量可以是一个立即数，或者放在单字节寄存器%cl中（只允许以这个寄存器作为操作数）。x86-64中，移位操作对w位长的数据进行操作，移位量是由%cl寄存器的低m位决定的，其中2m=w，高位会被忽略。移位操作的目的操作数可以是一个寄存器或是一个内存位置。
5. 大多数运算指令既可以用于无符号运算，也可以用于补码运算，只有右移操作要求区分有符号和无符号数，这个特性使得补码运算称为实现有符号整数运算的一种比较好的方法的原因之一。
6. 通常，编译器产生的代码中，会用一个寄存器存放多个程序值，还会在寄存器之间传递程序值。
7. 一般的算术操作：

|  |  |
| --- | --- |
| Leaq S D | 加载有效地址，&S-->D |
| Inc/Dec/Neg/Not D | +1/-1/求负数/取反 |
| Add/Sub/Imul/Xor/Or/Andx A B | B+/-/\*/^/|/&A-->B |
| SAL/SHL/SAR/SHR k A | A<</<</算术 >>/逻辑 >>k -->A |

1. 两个64位有符号或无符号正数相乘得到的乘积需要128位来表示，x86-64指令集对128位（16字节）数的操作提供有限的支持，Intel把16字节的数称为8字（oct word）
2. imulq指令有两种形式，一种是如上表所示的双操作数乘法指令，由两个64位操作数产生一个64位乘积；而mulq（无符号）和imulq（补码）是两种单操作数乘法指令，要求一个参数必须在寄存器%rax中，另一个由作为指令的源操作数给出，然后乘积存放在%rdx（高64位）和%rax（低64位）中。在存储这个乘积时，需要注意如果使用小端法机器，那么%rax中的值应当被存放在较低的地址。
3. 除法或取模操作是由单操作数除法命令来提供的，有符号除法指令idivl将寄存器%rdx（作为高64位）和寄存器%rax（作为低64位）中的128位数作为被除数，而除数作为指令的操作数给出，指令将商存储在寄存器%rax中，将余数存储在寄存器%rdx中。
4. 对于大多数64位除法应用来说，被除数常常是一个64位的值，这个值应当存放在%rax中，而%rdx应该由%rax符号扩展（有符号）或清零（无符号）得到。符号扩展的指令可以由cqto完成，这条指令不需要操作数（注意在Intel中这条指令为cqo）。
5. 无符号除法使用divq指令，通常寄存器%rdx会事先设置成0
6. 乘除法的操作：

|  |  |
| --- | --- |
| imulq S | S\*R[%rax]->R[%rdx]:R[%rax] |
| mulq S | S\*R[%rax]->R[%rdx]:R[%rax] |
| cqto | R[%rax]-（符号扩展）->R[%rdx]:R[%rax] |
| idivq S | R[%rdx]:R[%rax] mod S -> R[%rdx]  R[%rdx]:R[%rax] / S -> R[%rax] |
| divq S | R[%rdx]:R[%rax] mod S -> R[%rdx]  R[%rdx]:R[%rax] / S -> R[%rax] |

57.条件码

|  |  |
| --- | --- |
| CF | 进位标志，最近的操作使最高位产生了进位，可用来检查无符号操作的溢出 |
| ZF | 零标志，最近的操作得出的结果为0 |
| SF | 符号标志，最近的操作得到的结果为负数 |
| OF | 溢出标志，最近的操作导致一个补码溢出（正溢出或负溢出） |

58.leaq指令并不会设置任何条件码，因为它进行的是地址计算，而其余所有运算都改变条件码；对于逻辑操作，进位标志和溢出标志会设置为0；对于移位操作，进位标志会设置成最后一个被移出的位，溢出标志会设置为0；INC和DEC操作会设置溢出和零标志，但是不会改变进位标志。

59.有两类cmp和test指令，其作用只改变条件码而不改变其他寄存器，注意cmp A B指令计算的是B-A而非A-B

60.条件码的访问：

|  |  |  |  |
| --- | --- | --- | --- |
| set(n)e  /set(n)z D | 判断（不）为0 | ZF->D | ~ZF->D |
| set(n)s D | 判断（不）为负 | SF->D | ~SF->D |
| setg(e)  /setnl[e] D | 判断大于（大于等于）（有符号） | ~(SF^OF)&~ZF->D | ~(SF^OF)->D |
| setl(e)  /setng[e] D | 判断小于（小于等于）（有符号） | SF^OF->D | (SF^OF)|ZF->D |
| seta(e)  /setnb[e] D | 判断超过（超过或相等）（无符号） | ~CF&~ZF->D | ~CF->D |
| setb(e)  /setna[e] D | 判断低于（低于或相等）（无符号） | CF->D | CF|ZF->D |

1. 跳转指令及跳转条件：

|  |  |  |  |
| --- | --- | --- | --- |
| jmp label/\*operand | 无条件跳转 | 无 | 无 |
| j(n)e/j(n)z label | 若（不）为0跳转 | ZF | ~ZF |
| j(n)s label | 若（不）为负数跳转 | SF | ~SF |
| jg(e)/jnl[e] label | 按大于（大于等于）跳转（有符号） | ~(SF^OF)&~ZF | ~(SF^OF) |
| jl(e)/jng[e] label | 按小于（小于等于）跳转（有符号） | SF^OF | (SF^OF)|ZF |
| ja(e)/jnb[e] label | 按超过（超过或相等）跳转（无符号） | ~CF&~ZF | ~CF |
| jb(e)/jna[e] label | 按低于（低于或相等）跳转（无符号） | CF | CF|ZF |

62.jmp是无条件跳转，可以是直接跳转（以标号作为跳转目标，作为指令一部分编码），也可以是间接跳转（跳转目标从寄存器或内存中读出，用\*表示）；但条件跳转只能是直接跳转。

63.跳转指令最常见的编码方法是PC相对的，即将目标指令的地址与紧跟在跳转指令后面的那条指令的地址之差（而不是跳转指令本身的地址）作为编码，这个偏移量可编码为1、2、4个字节；第二种编码方式会给出绝对地址，用4个字节直接指定目标。

64.在相对寻址的过程中，应当注意我们观察字节编码使，偏移量是使用补码编码的，应当注意正负的问题。

65.在汇编代码中可以放心地无视rep或repz指令。

66.用条件控制实现条件分支的方式类似在C语言中使用goto语句。

67.条件传送语句及传送条件：

|  |  |  |  |
| --- | --- | --- | --- |
| cmov(n)e/  cmov(n)z  A B | 当（不）为0时进行A->B | ZF | ~ZF |
| cmov(n)s A B | 当（不）为负数时进行 A->B | SF | ~SF |
| cmovg(e)/  cmovnl[e]  A B | 当大于（大于等于）时进行A->B（有符号） | ~(SF^OF)&~ZF | ~(SF^OF) |
| cmovl(e)/  cmovng[e]  A B | 当小于（小于等于）时进行A->B（有符号） | SF^OF | (SF^OF)|ZF |
| cmova(e)/  cmovnb[e]  A B | 当超过（超过或相等）时进行A->B（无符号） | ~CF&~ZF | ~CF |
| cmovb(e)/  cmovna[e]  A B | 当低于（低于或相等）时进行A->B（无符号） | CF | CF|ZF |

1. 处理器使用流水线来获得高性能，在流水线中，一条指令的处理要经过一系列的阶段，每个阶段执行所需操作的一小部分，这种方法通过重叠连续指令的方法来获得高性能。要做到这一点，要求能够事先确定要执行的指令序列，这样才能保持流水线中充满了待执行的指令，当机器遇到条件跳转时，只有当分支条件求值完成后，才能决定分支往哪边走，处理器采用非常精密的分支预测逻辑来猜测每条跳转指令是否会执行，只要它的猜测还比较可靠，指令流水线中就会充满指令。另一方面，错误预测一个跳转要求处理器丢掉它为该跳转指令后所有指令已做的工作，然后再开始用从正确位置处起始的指令去填充流水线，这样一个错误会导致大约15~30个时钟周期的惩罚。
2. 假设分支预测错误的概率是p，没有预测错误用时为T1，预测错误处罚为T2（即在预测错误时程序运行时间为T1+T2），则执行代码的平均时间为T=T1+pT2，这样在其中一些量已知时可以计算出另一些量。
3. 无论测试数据是什么，编译出来使用条件传送的代码所需的时间都大致相同，控制流不依赖于数据，这使得处理器更容易保持流水线是满的。
4. 条件传送的源可以是寄存器或内存，目的是寄存器，它们可以是16、32、64位，不支持单字节传送，汇编器可以从目标寄存器的名字推断出条件传送指令的操作数长度，所以对所有的操作数长度，都可以使用同一个指令名字。
5. 处理器无需预测测试的结果就可以执行条件传送，处理器只是读原值、检查条件码，然后要么更新目的寄存器，要么保持不变。
6. 不是所有的条件表达式都可以用条件传送编译，无论测试结果如何，我们都会对两种情况的表达式求值，如果这两个表达式中的任意一个可能产生错误条件或者副作用，就会导致非法的行为。
7. 只有当两个表达式都很容易计算是，例如表达式分别都只是一条加法指令，它才会是用条件传送。即使许多条件分支预测错误的开销会超过更复杂的计算，GCC还是会使用条件控制传送。
8. 在根据汇编逆向工程循环的时候，关键是找到程序值和寄存器值之间的关系，比较困难的情况是为了节约资源编译器会重组计算，导致忽略一些原本存在的值或引入一些原本不存在的值，同时编译器还可能试图用一个寄存器来存储多个程序值。为了理解这一过程，我们应当关注的是循环之前如何初始化寄存器，循环中如何更新和测试寄存器，循环之后如何使用寄存器。需要注意的是，gcc常常执行一些变换，非但不能带来好处，甚至可能降低代码性能。
9. while循环有两种实现方法，一种是jump to middle，也就是先执行无条件跳转跳到循环结尾的测试来执行初始的测试；另一种是guarded-do，即先用条件分支，如果初始条件不成立就跳过循环，这样把代码转变成了do-while循环，当使用较高等级编译时，例如使用命令行-O1，GCC会采用这种策略。利用这种实现策略，编译器常常可以优化初始的测试，例如认为测试条件总是满足。
10. C语言标准说明while循环与for循环的等价性，但是要注意for中有continue的情况。
11. switch可以根据一个整数索引值进行多重分支，通过使用跳转表这种数据结构使得实现更加高效。跳转表是一个数组，表项i是一个代码段的地址，这个代码段实现当开关索引等于i时程序应该采取的动作。
12. 使用跳转表的优点是执行开关语句的时间与开关情况的数量无关，GCC根据开关情况的数量和开关情况值的稀疏程度来翻译开关语句，当开关情况数量比较多（如4个以上），并且值的跨度范围比较小的时候，就会使用跳转表。
13. 在将开关变量映射到下标时，会将开关变量视为一个无符号数，然后减去一个数把取值范围映射到0~M。
14. 执行switch语句的关键步骤是通过跳转表来访问代码位置，jmp指令的操作数有前缀\*表明是一个间接跳转。跳转表对重复情况的处理就是简单地应用同样的代码标号，对缺失情况就是使用默认的标号。
15. 跳转表一般保存在“.rodata”的目标代码文件段中。
16. 假设过程P调用过程Q，Q执行后返回P，这些动作包括下面一个或多个机制：
17. 传递控制：在进入过程Q的时候，程序计数器必须被设置为Q的代码的起始地址，然后在返回时，要把程序计数器设置为P中调用Q后面的那条指令的地址；
18. 传递数据：P必须能够向Q提供一个或多个参数，Q必须能够向P返回一个值。
19. 分配和释放内存：在开始时，Q可能需要为局部变量分配空间，而在返回前，又必须释放这些存储空间。
20. C语言过程调用机制的一个关键特性在于使用了栈数据结构提供的后进后出的内存管理原则。当P调用Q时，控制和数据信息添加到栈尾，当P返回时，这些信息会释放掉。
21. x86-64的栈向低地址方向增长。而栈指针%rsp指向栈顶元素，可以用pushq或popq来将数据存入栈中或从栈中取出，将栈指针减小一个适当的量可以为没有指定初始值的数据在栈上分配空间。类似地，可以通过增加栈指针来释放空间。
22. 当x86-64的过程所需要的存储空间超出寄存器能够存放的大小时，就会在栈上分配空间，这个部分称为过程的栈帧。当前正在执行的过程的帧总是在栈顶，当过程P调用过程Q时，会把返回地址压入栈中，我们把这个返回地址当做P的栈帧的一部分。大多数过程的栈帧都是定长的，但有些过程需要变长的帧。
23. 通过寄存器，过程P可以传递最多6个整数值（也就是指针和整数），如果Q需要更多的参数，P可以在调用Q之前在自己的栈帧里存好这些参数。x86-64过程只分配自己所需要的栈帧部分，许多函数甚至根本不需要栈帧，当所有的局部变量都可以保存在寄存器里，而且该函数不会调用任何其它函数（有时候称为叶子过程），就可以这样处理。
24. call Q调用Q会把地址A压入栈中，并将PC设置为Q的起始地址，压入的地址A称为返回地址，是紧跟在call指令后面那条指令的地址，对应的指令ret会从栈中弹出地址A，并把PC设置为A。（在OBJDUMP产生的反汇编输出中这些指令会添加后缀q，只是为了强调这是x86-64版本的调用和返回，而不是IA32的，在x86-64的汇编代码中这两种版本可以互换）。call的跳转可以是直接的，也可以是间接的。
25. x86-64中，可以通过寄存器传递至多6个整型（整数或指针），寄存器的使用有特殊顺序，寄存器使用的名字取决于要传递的数据类型的大小。如果一个函数有大于6个整型参数，超出6个的部分就要通过栈来传递，假设P调用Q有n个整型参数且n>6，则P的代码分配的栈帧必须能容纳7~n号参数的存储空间，要把参数1~6复制到对应的寄存器，把参数7~n放到栈上，而参数7位于栈顶。通过栈传递参数时，所有数据大小都向8的倍数对齐。
26. 局部数据必须存放在内存中的情况：
27. 寄存器不够存放所有的本地数据
28. 对一个局部变量使用地址运算符&，因此必须能够为它产生一个地址
29. 某些局部变量是数组或结构，因此必须能够通过数组或结构引用被访问到。
30. 寄存器是被所有过程共享的资源，虽然给定时刻只有一个过程是活动的，我们必须确保当一个过程（调用者caller）调用另一个过程（callee）时，被调用者不会覆盖调用者稍后会使用的寄存器值。为此，x86-64采用了一组统一的寄存器使用管理，所有过程（包括程序库）都必须遵守。
31. 被调用者保存的寄存器指当过程P调用过程Q时，Q必须保存这些寄存器的值，保证它们的值在Q返回到P时与Q被调用时是一样的。过程Q保持一个寄存器的值不变，要么就根本不去改变它，要么就把原始值压入栈中，改变寄存器的值，最后在返回前从栈中弹出旧值。
32. 调用者保存的寄存器是指若P在此类寄存器中保存了一个数据，然后调用Q，那么保存这个寄存器的值是P的责任。
33. 每个过程调用在栈中都有自己的私有空间，多个未完成调用的局部变量不会相互影响，这使得x86-64的过程能递归调用它们自身。递归调用一个函数本身和调用其它函数是一样的。
34. 对数组声明为T A[N]，其起始地址为add，这个声明会在内存中分配一个L\*N字节的连续区域，L是数据类型T的大小（以字节为单位），数组元素i会被存放在地址为add+L·i的地方。
35. C语言允许对指针进行运算，而计算出来的值会根据该指针引用的数据类型的大小进行伸缩，也就是说如果p是指向类型T的指针，p的值为add，那么表达式p+i的值为add+L\*i，这里L是数据类型T的大小。
36. 操作符&和\*可以产生指针和间接引用指针，对于一个表示某个对象的表达式expr，&expr给出给对象地址的一个指针，而对于一个表示地址的表达式Aexpr，\*Aexpr给出该地址处的值，因此表达式\*&expr和expr是等价的，数组引用A[i]等同于表达式\*(A+i)。
37. 可以计算出同一个数据结构中的两个指针之差，结果的数据类型为long，值等于两个地址之差除以该数据类型的大小。如&E[i]-E=i。
38. 数组在内存中按照“行优先”的顺序排列，第0行的所有元素可以写成A[0]，要访问多维数组的元素，编译器会以数组起始为基地址，（可能需要经过伸缩的）偏移量为索引，产生计算期望的元素的偏移量。通常来说，对于一个声明如下的数组T D[R][C]，其元素&D[i][j]=add+L(C·i+j)，这里add是起始地址，L是数据类型T以字节为单位的大小。
39. 当称需要用一个常数作为数组的维度或缓冲区的大小时，最好通过#define声明将这个常数与一个名字联系起来，然后在后面一直使用这个名字代替常数的数值，这样一来，如果需要修改这个值，只用简单的修改这个#define的声明就可以了。
40. 如果使用定长数组，程序可以把地址的乘法运算改为简单的加法运算，这样能大大提高程序效率。
41. ISO C99引入了一种功能，允许数组的维度是表达式，在数组被分配时才计算出来，需要注意的是如果数组和数组大小同时作为参数传入，应当先传入大小再传入数组。这样的数组由于改变了大小，需要用乘法指令来计算偏移量而不是简单的leaq地址计算。但是在一个循环中引入变长数组时，编译器常常可以根据访问模式的规律性来优化索引计算，比如每次移动n步，这样也能避免乘法的使用。
42. 结构的所有组成部分都存放在内存中一段连续的区域内，而指向结构的指针就是结构第一个字节的地址。
43. 在引用结构的指针时，必须写成(\*obj).name的形式，因为编译器会将\*obj.name解释成\*(obj.name)，而第一个形式可以有替代写法obj->name。
44. 联合允许以多种类型来引用一个对象，用不同的字段引用相同的内存块。一个联合总的大小是其最大字段的大小加上适当的对齐。
45. 一种应用联合的场景是我们事先知道对一个数据结构中两个不同字段的使用是互斥的，那么将这两个字段声明为联合的一部分而不是结构的一部分会减小分配空间的总量。另一种场景是联合可以用来访问不同数据类型的位模式。也即用一种数据类型来存储联合中的参数，而以另一种数据类型进行访问，这样就能得到相同位表示的不同解释。
46. 数据对齐限制简化了形成处理器和内存系统之间接口的硬件设计，无论数据是否对齐，x86-64都能正确工作，不过数据对齐是建议的以提高系统性能，对齐原则是任何K字节的基本对象的地址都必须是K的倍数。
47. 编译器会在汇编代码中放入命令，指明全局数据所需的对齐，用.align表示
48. 对于结构，编译器可能需要在字段的分配中插入间隙和结尾插入填充以保证每个结构元素都满足它的对齐要求，而结构本身对它的起始地址也有一些对齐要求。
49. 在考虑对齐的情况下，想使结构占内存最小可以将结构中元素按字节大小从大到小排列。
50. 如果数据没有对齐，某些型号的Intel和AMD处理器对有些实现多媒体操作的SSE指令就无法正确执行，这些指令要求内存地址必须是16的倍数，因此任何针对x86-64处理器的编译器和运行时系统都必须保证分配用来保存可能会被SSE读写的数据结构的内存必须满足16字节对齐，于是任何内存分配函数生成的块的起始地址都必须是16的倍数，大多数函数的栈帧的边界都必须是16的倍数（有一些例外）。除了提供SSE的超集，支持AVX的指令并没有强制性的对齐要求。
51. 在C语言中，每个指针都对应一个类型表示这个指针指向哪一类对象，如果对象类型为T，则指针的类型为T\*，特殊类型void\*表示通用指针，比如malloc返回一个通用指针然后转换成一个有类型的指针，指针类型不是机器代码中的一部分。
52. 每个指针都有一个值，表示其所指向的对象的地址，NULL表示指针没有指向任何地方。
53. 指针可以用&运算符创建，其可以应用到任何lvalue的C表达式上，&运算符的机器代码常常用leaq指令计算地址。\*操作符可以间接引用指针，其结果是一个值，其类型与该指针的类型是一样的。
54. 一个数组的名字可以像一个指针变量引用，将指针从一种类型强制转化成另一种类型只改变其类型而不改变其值。改变其类型的效果是影响指针的伸缩运算。
55. 指针可以指向函数，者提供了很强大的存储和向代码传递引用的功能。函数指针的值是该函数机器代码表示中第一条指令的地址。函数指针应当被声明成T (\*f)()而非T\* f()，因为后者会被解释成一个返回值为T\*的正常函数。
56. C对于数组引用不做任何边界检查，而局部变量和状态信息都存放在栈中，这可能导致严重的程序错误，对越界的数组元素的写操作会破坏存储在栈中的状态信息，当程序使用这个被破坏的状态，试图重新加载寄存器或执行ret指令时，就会出现很严重的错误。一种特别常见的状态破坏称为缓冲区溢出，通常在栈中分配某个字符数组用来保存一个字符串，但是字符串的长度超过了为数组分配的空间。比如如果存储的返回地址的值被破坏了，那么ret指令会导致程序跳转到一个完全意想不到的位置。通常，使用gets或任何其他能导致存储溢出的函数（strcpy，strcat，sprintf）都不是一个好的变成习惯，因为它们不需要得知目标缓冲区的大小就产生一个字节序列。
57. 缓冲区溢出的一个更加致命的使用就是让程序执行它本来不愿意执行的函数，这是一种最常见的通过计算机网络攻击系统安全的方法。通常，输入给程序一个字符串，这个字符串包含一些可执行代码的字节编码，称为攻击代码，还有一些字节会用一个指向攻击代码的指针覆盖返回地址，那么执行ret指令的效果就是跳转到攻击代码。
58. 在一种攻击形式中，攻击代码会使用系统调用启动一个shell程序，给攻击者提供一组操作系统函数；在另一种攻击形式中，攻击代码会执行一些未授权的任务，修复对栈的破坏，然后第二次执行ret指令，表面上正常返回到调用者。
59. 蠕虫（worm）可以自己运行，并且将自己的等效副本传播到其他机器；而病毒（virus）能将自己添加到包括操作系统在内的其他程序中，但它不能独立运行。
60. 现代的编译器和操作系统实现了很多机制来避免遭受缓冲区溢出的攻击，限制入侵者通过缓冲区溢出攻击获得系统控制的方式。
61. 栈随机化：为了在系统中插入代码，攻击者既要插入代码，也要插入指向这段代码的指针，这个指针也是攻击字符串的一部分，产生这个指针需要知道这个字符串放置的栈地址。在过去，程序的栈地址非常容易预测，对于所有运行同样程序和操作系统版本的系统来说，在不同的机器之间，栈的位置是相当固定的。这会导致许多系统都容易受到同一种病毒的攻击，这种现象常被称为安全单一化。

栈随机化的思想使得栈的位置在程序每次运行时都有变化，实现的方式是程序开始时在栈上分配一段0~n字节随机大小的空间，比如使用alloca分配。分配的范围n必须足够大才能获得足够多栈地址的变化，但又要足够小不至于浪费程序太多的空间。

栈随机化在Linux中已经成为了标准型为，它是更大一类名为地址空间随机化ASLR的技术中的一种，ASLR每次运行时程序的不同部分，包括程序代码、库代码、栈、全局变量和堆数据都会被加载到内存的不同区域。

而破解这种防护的一种常见的把戏就是在实际的攻击代码前插入很长一段nop操作，称为空操作雪橇（nop sled），执行这种指令只会使程序计数器加一，这样只要攻击者能够猜对这段序列中的某个地址程序就会经过这个序列到达攻击代码。比如如果我们建立一个256字节的nop sled，那么枚举2^15个起始地址就可以破解n=23的随机化。

②.最新的GCC版本在产生的代码中加入了一种栈保护者机制，来检测缓冲区越界。其思想是在栈帧中任何局部缓冲区与栈状态之间存储一个特殊的金丝雀（canary）值，也称为了哨兵值，是程序随机产生的，攻击者没有简单的办法得知它是什么，而在恢复寄存器状态和从函数中返回之前，程序检查这个金丝雀值是否被函数的某个操作改变了，如果是，那么程序异常终止。使用命令行“-fno-stack-protector”来阻止GCC产生栈保护代码。

指令参数%fs:xx指明金丝雀值是用段寻址从内存中读入的，段寻址机制可以追溯到80286的寻址。将金丝雀值存放在一个特殊的段中，标记为“只读”，这样攻击者就不能覆盖存储的金丝雀值。GCC只在函数中有局部char类型缓冲区时才插入这样的代码。

③.限制可执行代码区域：在典型的程序中，只有保存编译器产生的代码那部分内存才是可执行的，其他部分被限制为只允许读或写。虚拟内存空间在逻辑上被分成了页（page），典型的每页是2048或4096个字节，硬件支持多种形式的内存保护，能够指明用户程序和操作系统内核所允许的访问形式。许多系统允许控制三种访问形式：读（从内存读数据）、写（存储数据到内存）和执行（将内存的内容看做机器级代码）。

以前x86体系结构将读和执行访问控制合并成一个1位的标志，这样任何被标记为可读的页也都是可执行的。栈必须是既可读又可写的，因此栈一定是可执行的。已经实现的很多机制，能够限制一些页是可读的但不可执行的，但这些机制通常会带来严重的性能损失。

最近，AMD为它的64位处理器引入了NX位，将读和执行访问模式分开，Intel也跟进了，这样栈可以标记成可读和可写但不可执行，检查页是否可执行由硬件来检查，性能上没有损失。

有些类型的程序要求动态产生和执行代码的能力，比如即时编译技术为解释语言（如Java）编写的程序动态地产生代码，以提高执行性能。是否能够将可执行代码限制在由编译器在创建原始程序产生的那个部分中，取决于语言和操作系统。

上面三个技术是用于最小化程序缓冲区溢出攻击漏洞三种最常见的机制，它们都不需要程序员做任何特殊的努力，带来的性能代价都非常小甚至没有，单独每一种机制都降低了漏洞的等级，而组合起来更加有效，但不幸的是仍然有办法能够攻击计算机。

1. 很多过程编译器能预先确定需要为战阵分配多少空间，但有些函数需要的局部存储是变长的，比如alloca（一个标准库函数，可以在栈上分配任意字节数量的存储）或声明一个局部变长数组。
2. 为了管理变长栈帧，x86-64使用寄存器%rbp作为帧指针（有时称为基指针），当使用帧指针时，代码必须把%rbp之前的值保存到栈中，因为它是一个被调用者保存的寄存器，然后在整个函数执行过程中，都使得%rbp指向那个时刻栈的位置，然后用固定长度的局部变量相对于%rbp的偏移量来引用它们。最后leave指令会将帧指针恢复到它之前的值，等价于movq %rbp,%rsp;popq %rbp。
3. 在较早的x86代码中，每个函数调用都使用了帧指针，而现在只有栈帧长可变的情况下才会使用，历史上大多数编译器在生成IA32代码时会使用帧指针，而最新的GCC版本放弃了这个惯例。可以看到把使用帧指针的代码和不使用帧指针的代码混在一起是可以的，只要所有的函数都把%rbp当做被调用者保存的寄存器来处理即可。
4. 链接
5. 大多数编译系统提供编译器驱动程序，代表用户在需要时调用语言与处理器、编译器、汇编器和链接器，驱动程序首先运行C预处理器（cpp），将main.c翻译成一个ASCII的中间文件main.i，接下来运行C编译器（ccl）将main.i翻译成一个ASCII汇编文件main.s，然后运行汇编器（as）将main.s翻译成一个可重定位目标文件main.o，最后运行链接器程序ld，将.o文件及一些必要的系统目标文件组合起来，创建一个可执行目标文件，在运行时shell会调用操作系统中一个叫做加载器loader的函数，将可执行文件的代码和数据复制到内存，然后将控制转移到这个程序的开头
6. Linux LD程序这样的静态链接器以一组可重定位目标文件和命令行参数作为输入，生成一个完全连接的、可以加载和运行的可执行目标文件作为输出，输入的可重定位目标文件由不同的代码和数据节section组成，每一节都是一个连续的字节序列，指令在一节中，初始化了的全局变量在一节中，而未初始化的变量又在另外一节中。
7. 链接器要完成的主要任务有两个：一个是符号解析：目标文件定义和引用的符号，每个符号对应了一个函数、一个全局变量或一个静态变量（C语言中用static声明），符号解析的目的是将每个符号引用正好和一个符号定义关联起来；另一个是重定位：编译器和汇编器生成从地址0开始的代码和数据节，连接器通过把每个符号定义与一个内存位置关联起来，从而重定位这些节，然后修改所有对这些符号的引用，使得他们指向这个内存位置。链接器使用汇编器产生的重定位条目的详细指令，不加甄别的执行这样的重定位。
8. 目标文件纯粹是字节块的集合，这些块中有些包含程序代码，有些包含程序数据，而其他的包含引导连接器和加载器的数据结构，链接器对目标机器了解甚少，产生目标文件的编译器和汇编器已经完成了大部分工作。
9. 目标文件有三种形式：可重定位目标文件（包含二进制代码和数据，其形式可以在编译时与其他可重定位目标文件结合起来来创建一个可执行目标文件）；可执行目标文件（包含二进制代码和数据，其形式可以被直接复制到内存并执行）；共享目标文件（一种特殊类型的可重定位目标文件，可以在加载或运行时被动态地加载进内存并链接）。
10. 编译器和汇编器生成可重定位目标文件（包括共享目标文件），链接器生成可执行目标文件，技术上说一个目标模块就是一个字节序列，而一个目标文件就是一个以文件形式存放在磁盘中的目标模块，但这两个术语会互换使用。
11. 各个系统的目标文件的组织格式是不同的，第一个Unix系统使用的是a.out格式（今日可执行文件仍称为a.out文件），Windows使用可移植可执行（PE）格式，Mac-OS-X使用Mach-O格式，现代x86-64 Linux和Unix使用可执行可链接格式（ELF）。
12. ELF文件的格式如下图所示

|  |
| --- |
| ELF头：以一个16字节的序列开始，描述生成该文件的系统的字的大小和字节顺序，剩下的部分包含帮助链接器语法分析和解释目标文件的信息，其中包括ELF头的大小、目标文件的类型、机器类型、节头部表的文件偏移，以及节头部表中条目的大小和数量。 |
| .text：已编译程序的机器代码 |
| .rodata：只读数据，比如printf语句中的格式串和开关语句的跳转表 |
| .data：已初始化的全局变量和静态C变量，局部C变量在运行时被保存在栈中，既不出现在.data节也不出现在.bss节中 |
| .bss：未初始化的全局和静态C变量，以及所有被初始化为0的全局或静态变量，这个节不占实际的空间而仅仅是一个占位符，这种区分是为了空间效率，未初始化的变量不需要占据任何实际的磁盘空间，运行时直接在内存中分配这些变量，初始值为0 |
| .symtab：一个符号表，存放在程序中定义和引用的函数和全局变量的信息，不需要通过-g选项来得到符号表信息，但和编译器中的符号表不同，.symtab中不包含局部变量的条目 |
| .rel.text：一个.text节中位置的列表，当链接器把这个目标文件和其它文件组合时，需要修改这些位置，任何调用外部函数或引用全局变量的指令都需要修改，但调用本地函数的指令则不需要修改。可执行目标文件中并不需要重定位信息，因此通常省略，除非用户显示地指示链接器包含这些信息 |
| .ret.data：被模块引用或定义的所有全局变量的重定位信息，任何已初始化的全局变量如果初始值是一个全局变量的地址或者外部定义函数的地址，都需要被修改 |
| .debug：一个调试符号表，其条目是程序中定义的局部变量和类型定义，程序中定义和引用的全局变量以及原始的C源文件，只有用-g选项编译的时候才会得到这张表 |
| .line：原始C源程序中的行号和.text节中机器指令之间的映射，用-g得到 |
| .strtab：一个字符串表，其内容包括.symtab和.debug节中的符号表，以及节头部中的节名字，字符串表就是以null为结尾的字符串的序列 |
| 节头部表：描述不同节的位置和大小，目标文件中每个节都有一个固定大小的条目（entry） |

1. 每个可重定位目标模块都有一个符号表，包含该文件定义和引用的符号的信息，有三种不同的符号：
2. 由该文件定义并能被其他模块引用的全局符号，全局连接器符号对应于非静态的C函数和全局变量
3. 由其他模块定义并被该模块引用的全局符号，这些符号称为外部符号，对应于在其他模块中定义的非静态的C函数和全局变量
4. 只被本模块定义和引用的局部符号，它们对应于带static属性的C函数和全局变量，这些符号在本模块中任何位置都可见，但是不能被其他模块引用。

本地链接器符号和本地程序变量是不同的，.symtab中的符号表不包含对应于本地非静态程序变量的任何符号，这些符号在运行时在栈中被管理，链接器对此类符号不感兴趣，但定义为static属性的本地过程变量是不在栈中管理的，编译器在.data或.bss中为每个定义分配空间，对于不同函数中定义的同名静态变量，编译器会向汇编器输出两个不同名字的局部连接器符号。任何用static声明的全局变量或函数都是模块私有的，尽可能用static属性来保护你的变量和函数是很好地编程习惯。

1. 符号表是由汇编器构造的，符号表包含一个条目的数组，其中每个条目格式如下：

|  |
| --- |
| name：字符串表的字节偏移，指向符号的以null为结尾的字符串名字 |
| type（4bit）：要么是数据，要么是函数 |
| binding（4bit）：符号是本地的还是全局的 |
| reserved（未使用，1byte） |
| section：每个符号被分配到目标文件中的哪个节，也是一个到节头部表的索引，有三个特殊的伪节，在节头部表中没有条目：ABS表示不该被重定位的符号，UNDEF表示未定义的符号，也就是在本目标模块中引用，但在其他地方定义的符号，COMMON表示还未被分配位置的未初始化的数据目标，对此，value字段给出对齐要求，size给出最小的大小，只有可重定位目标文件中才有这些伪节，可执行目标文件中是没有的。 |
| value：符号的地址，对于可重定位模块来说，它是距离定义目标的节的起始位置的偏移；对于可执行文件来说，它是一个绝对运行时地址 |
| size：目标的大小（以字节为单位） |

1. 现代GCC版本将未初始化的全局变量分配到COMMON中，将未初始化的静态变量和初始化为0的全局或静态变量分配到.bss中
2. 链接器解析符号引用的方法是将每个引用与它输入的可重定位目标文件的符号表中的一个确定的符号定义关联起来，编译器只允许每个模块中每个局部符号有一个定义，静态局部变量也会有本地链接器符号，编译器还要确保它们拥有唯一的名字。但当编译器遇到一个不是在当前模块中定义的符号（变量或函数名）时，会假设该符号是在其他摸个模块中定义的，生成一个链接器符号表条目，并把它交给连接器处理，如果链接器在它的任何输入模块中都找不到这个被引用符号的定义，就输出一条错误信息并终止。
3. 对全局符号的解析很棘手，因为多个目标文件可能会定义相同名字的全局符号。
4. 链接器的输入是一组可中定位目标模块，每个模块定义一组符号，如果多个模块定义同名的全局符号，Linux系统采用的策略如下：
5. 编译时编译器向汇编器输出每个全局符号，分为强和弱，汇编器把这个信息隐含地编码在可重定位目标文件的符号表里，函数和已初始化的全局变量是强符号，而未初始化的全局变量是弱符号。
6. 不允许有多个同名的强符号；如果有一个强符号和多个弱符号同名，那么选择强符号；如果有多个弱符号同名，那么从这些弱符号中任意选择一个，而链接器通常不会标明它检测到多个同名符号的定义。

在怀疑出现此类错误时，用GCC-fno-common标志这样的选项调用链接器，遇到多重

定义的全局符号时触发一个错误，或者用-Werror选项，会把所有的警告视为错误

1. 采用上述分配COMMON和.bss的规则是由于在某些情况中链接器允许多个模块定义同名的全局符号，当编译器在某个模块中遇到了弱全局符号，它无法判断其他模块是否也定义了该符号，也无法确定该选择哪一个定义，因此编译器把x分配为COMMON，将决定权留给链接器，而如果该符号有初始化为0，那么其是一个强符号，因此可以被很自信地分配为.bss，静态符号的构造必须是唯一的，所以只会被分配进.data或.bss
2. 将所有相关的目标模块打包称一个单独的文件称为静态库，可以用作链接器的输入，链接器构造一个输出的可执行文件时，只复制静态库里被应用程序引用的目标模块。
3. 如果不使用静态库，一种方法是要求编译器识别出对标准函数的调用并直接生成相应代码（如Pascal采用的方式），但这对C是不合适的，因为C标准中有大量的标准函数，这样会使编译器更加复杂，同时对标准函数的修改会导致必须对编译器的修改；而如果将所有的C函数都放在一个可重定位目标模块中，那么系统中的每个可执行目标文件都包含标准函数集合的完全副本，这对磁盘空间是很大的浪费，而且每个正在运行的程序都将它自己的这些函数的副本放在内存上，这是对内存的浪费，而且每次修改标准函数都要求库的开发人员重新编译整个源文件；而如果对每个标准函数创建一个独立的可重定位文件，但是这种方法要求应用程序员显式地链接合适的目标模块到它们的可执行文件，这容易出错而且很耗时。
4. 相关的函数可以被编译为独立的目标模块，然后封装成一个单独的静态库文件，然后应用程序通过在命令行上指定单独的文件名字来使用这些在库中定义的函数，在链接时链接器只复制被程序引用的目标模块，这样减少了可执行文件在磁盘和内存中的大小，同时应用程序员只需要包含较少的库文件的名字。
5. 在Linux系统中，静态库以一种称为存档archive的特殊文件形式存放在磁盘中，存档文件是一组连接起来的可重定位目标文件的集合，有一个头部用来描述每个成员目标文件的大小和位置，存档文件名由后缀.a标识
6. 与静态库链接时，-static命令告诉编译器驱动程序链接器应该构建一个完全连接的可执行目标文件，可以加载到内存并运行，加载时无须更进一步的链接。
7. 在符号解析阶段，编译器从左到右按照它们在编译器驱动程序命令行上出现的顺序来扫描可重定位目标文件和存档文件，链接器维护一个可重定位目标文件的集合E（这个集合中的文件会被合并起来形成可执行文件），一个未解析的符号U和一个在前面的输入文件中已定义的符号集合D，初始时E,U,D均为空
8. 对于命令行上每个输入文件f，链接器会判断f是一个目标文件还是一个存档文件，如果是目标文件，链接器会把f添加到E，修改U和D来反应f中的符号定义和引用，并继续下一个输入文件
9. 如果f是一个存档文件，链接器就尝试匹配U中未解析的符号和由存档文件成员定义的符号，如果某个存档文件成员m定义了一个符号解析U中的一个引用，则将m加到E中，并且链接器修改U和D来反应m中的符号定义和引用直到U和D都不再发生变化，此时不包含在E中的成员目标文件都被丢弃，链接器继续处理下一个输入文件
10. 链接器完成对命令行上输入文件的扫描后，U是非空的，那么链接器就会输出一个错误并终止，否则其会合并和重定位E中的目标文件，构建输出的可执行文件。
11. 在命令行中，如果定义一个符号的库出现在引用这个符号的目标文件之前，那么引用就不能被解析，链接会失败，关于库的一般规则是将它们放在命令行的结尾，如果各个库的成员是相互独立的，那么这些库就可以以任意顺序放置在命令行的结尾处，另一方面，如果库不是相互独立的，那么必须对他们排序，如果需要满足依赖要求，可以在命令行上重复库。