滴水信息

滴水逆向培训

基础教程

昆山滴水信息技术有限公司

（内部学习资料）

地址：昆山市巴城镇学院路 88 号
邮编：215311
TEL：0512-57882866
官网：www.dtdishui.com
论坛：www.dtdebug.com
EMAIL：kunshandishui@163.COM

前言

在写这篇前言之前，首先非常感谢多年以来一直关心和支持滴水的朋友们，同时也非常感谢那些不停的鞭策我们勇往直前的批评者。在你们的掌声与谩骂声中我们一路走来。

2008 年滴水公司成立之日起，我们就面临着一个非常严峻的现实，巨大的工作量无人可以分担。既招不到合适的程序员，也找不到愿意同行的伙伴。在孤独与寂寞中我们慢慢的爬行。。。。。。

国内有不计其数的计算机专业毕业生，但是真正的程序员却少之又少。大量院校开办计算机专业，而真正酷爱编程的学习者却无处求学。只有少数人通过自学成为真正合格的程序员。业内的很多人其实都知道原因。大学缺乏合格的计算机专业老师，只会照本宣科、没有工程实践的老师永远也教不出合格的程序员。而真正经验丰富、适合做老师的程序员又没有教书育人的机会。现实需要有人能够改变这种不合理的现状。

2011 年经过深思与熟虑，滴水决定暂时放下产品开发，开办逆向培训。我们要做一些有意义的事情。即使我们是大海中的一滴水，也可以荡起一点点的涟漪。知行合一，立刻行动才是我们所需要的！

首先介绍一下滴水逆向培训的主讲老师-唐老师。2000 年毕业于西北一所不知名的大学，贸易经济专业毕业—似乎资深的程序员大多非计算机专业出身。有着 10 年以上的编程经验，取得过计算机软件发明专利，业内公认的杰出程序员。唐老师是滴水硬件调试器和 VT 调试器的开发者，成功逆向虚拟机 VMWare、加壳软件 Themida 和 VMProtect，并即将完成滴水动态变形壳的开发。

接下来说明一下本教程的编写历程。滴水逆向培训基础教程是唐老师自学成才的学习历程以及 10 多年软件开发经验的总结，并经过滴水逆向培训前几期的讲课实践整理编辑而成。以此让更多无法参加实地培训又渴望学习的人一起分享我们的经验。

在此需要说明的是此教程不同于我们以往常见的教材。为了尽可能的保持唐老师教学的原貌，本教程采用语录+章节体的风格编辑整理而成。先引述唐老师的讲课内容，然后再添加学生实际学习中整理、消化吸收的历程。为读者尽可能的创造一种身临其境、置身其中的学习氛围，共同探讨和学习本教程的内容。由于个人天赋的差异，以及文字表达的局限性，每个读者可能都会有不同的理解，甚至有些比较深奥的地方难以理解。这都是正常的，即使

参加实地培训的学员相互之间也存在着巨大的差距。在此只能靠每个读者细心的领悟。同时，由于编写的时间过于仓促，教程本身可能存在诸多错误，希望读者多多批评指正，我们将在后续版本中不断的加以纠正和完善。

最后，我们需要特别强调学习本教程的注意事项：

适合的读者：本教程适合于任何有志于从事编程的读者，包括零基础在内。零基础可能会比较吃力，但有志者事竟成！

教学的目的：帮助零基础的人跨过计算机门槛，为有志于从事计算机编程工作的读者打下坚实的基础。培养独立的自学能力。

学习效果：打下坚实的基础知识，可以独立学习任何自己感兴趣的东西。

学习的前提条件：一是数学要好；数学知识的掌握程度反映一个人的逻辑思维能力，这是学习编程的必备条件，否则会在后续的学习中非常吃力，难以达到预期的学习效果。如果你初中数学优异基本就可以证明具备一定的逻辑思维能力了。二是有兴趣；兴趣是最好的老师。如果这不是你终身希望做的事，学习本教程将是一件非常枯燥无味的事，最终会放弃的。三是肯用功；即使你的智商超过爱因斯坦也不能确保你一定成绩优异，天才+勤奋才是成功的秘诀。补充说明一点，学习编程和英语水平没有必然联系。因为编程涉及的英语单词有限，看的次数多了自然记得。

学习方法：勤学+苦练。练习到手都麻了为止。反复的练习是最佳的捷径，偷懒是你最大的敌人。任何人概莫能外，切记！我们将会在每一章节安排必要的练习，千万不可以偷懒。

教学内容及顺序：几乎所有教材的知识内容大致都是相同的，但是最后学习的效果却有天壤之别，很大一部分原因在于学习的顺序。正确的教学顺序才能让我们在计算机浩如烟海的知识体系中找到正确的方向，循序渐进的消化和吸收才可以真正掌握所学习的内容。大学计算机教育的失败很大一部分原因在于教学内容和顺序的安排错误。表面上所有的内容都已经学过了，但是学完之后全部都忘了，一无所获。过多的学习内容只会让学生望而却步、疲于应付。在没有很好的消化吸收前段学习内容的情况下，继续填鸭式的学习新的内容，割裂了各个知识点之间的联系，最终导致学习失败。所以我们要求，在没有充分掌握前一段知识的情况下，不要继续学习后续的内容。希望能够引以为戒。本教程的教学内容抛弃了知识体系中大量多余的内容，只讲述必要的内容，并进行有效的穿插，帮助读者把每个知识点前后联系起来，并最终牢牢的记住。这是我们所坚持的正确的教学安排，特意为之。学习完本教程，打下一定的基础之后，就可以根据自己的爱好和工作要求自学完成本教程之外暂时没有教授的知识了。所以教程没有涉及过多的应用知识，这是我们有意安排的。学完本教程，打

好基础之后，其他的应用知识可以自学完成。基础决定一切。挖多深的地基才能盖多高的楼。这就是我们为什么从汇编开始的原因。只有从底层的汇编开始学起，才是唯一正确的方法。汇编基础打好之后，其他知识就会学的越来越轻松。

此外，由于时间的限制，我们将会在编辑教程中引用其他优秀教材的部分内容，在此向他们表示诚挚的敬意！

最后再次感谢支持和鞭策滴水的朋友们，感谢滴水员工胡雨和陶捷为编写本教程所付出的辛勤工作。我们将在实际教学中不断完善本教程，对于一些自学能力相对较弱的读者，欢迎在情况允许的情况下报名参加滴水逆向实地培训。如果在学习本教程过程中遇到难以理解的问题也可以登录滴水论坛与大家一起探讨，我们将尽可能的给予解答。滴水官网地址：www.dtdishui.com;论坛地址：www.dtdebug.com；谢谢！

昆山滴水信息技术有限公司

2013 年 1 月 8 日

《滴水逆向培训基础教程》电子版仅供参考，书面教程更为详尽。

目录

开始学习本教程之前，请您务必先仔细阅读《前言》，将会对您接下来的学习有所帮助。

第一章 进制

引言：

进制跟我们生活是息息相关的，比如 时钟，星期 等，那么计算机也离不开进制，计算机是通过二进制进行操作和运算的。

我们为什么要学习进制？

方便我们了解计算机，了解计算机的运行，为以后的学习打下基础。

什么才是正确的学习方法？

忘掉呆板的十进制！说到进制，其时大家都会，只是生活中的运用把其它的进制都丢弃了，只留下十进制，这一章主要是带我们了解各种进制，找回应有的记忆。

本章必须要掌握的知识点：

1．各种进制的书写方法

2．进制间的运算

3．计算机中负数的表示

4．布尔代数

本章常犯的错误：

1．总是以十进制为依托去考虑其他进制

2．其他进制间判断大小时先转换成十进制

3．负数与符号位

1.1 数据进制

本节主要内容：

1．了解进制

2．各种进制的书写方法

老唐语录：

现在请一个同学上来写出 10 进制的 0-100

0 1 2 3 4 5 6 7 8 9

10 11 12 13 14 15 16 17 18 19

20 21 22 23 24 25 26 27 28 29

30 31 32 33 34 35 36 37 38 39

40 41 42 43 44 45 46 47 48 49

50 51 52 53 54 55 56 57 58 59

60 61 62 63 64 65 66 67 68 69

70 71 72 73 74 75 76 77 78 79

80 81 82 83 84 85 86 87 88 89

90 91 92 93 94 95 96 97 98 99

100

大家看着上面的的数字说一句话：

小陶 say：有 101 个数

小胡 say：十进制从 0 到 100。

……

其实这就是小学入学考试，只要写出 0-100 就可以参加学习我们的课程。

下面我来给十进制下个定义：

十进制是由 0、1、2、3、4、5、6、7、8、9 十个符号组成，逢十进一。

你们给九进制下个定义：

九进制定义：九进制是由 0、1、2、3、4、5、6、7、8 九个符号组成，最小是 0，最大是 8，逢九进一。

练习：

用九进制写出十进制的 101 个元素：

0 1 2 3 4 5 6 7 8

10 11 12 13 14 15 16 17 18

20 21 22 23 24 25 26 27 28

30 31 32 33 34 35 36 37 38

40 41 42 43 44 45 46 47 48

50 51 52 53 54 55 56 57 58

60 61 62 63 64 65 66 67 68

70 71 72 73 74 75 76 77 78

80 81 82 83 84 85 86 87 88

100 101 102 103 104 105 106 107 108

110 111 112 113 114 115 116 117 118

120 121

现在给七进制下个定义：

七进制是由 0、1、2、3、4、5、6 七个符号组成，最小是 0，最大是 6，逢七进一。

练习：

用七进制写出十进制的 101 个元素：

0 1 2 3 4 5 6

10 11 12 13 14 15 16

20 21 22 23 24 25 26

30 31 32 33 34 35 36

40 41 42 43 44 45 46

50 51 52 53 54 55 56

60 61 62 63 64 65 66

100 101 102 103 104 105 106

110 111 112 113 114 115 116

120 121 122 123 124 125 126

130 131 132 133 134 135 136

140 141 142 143 144 145 146

150 151 152 153 154 155 156

160 161 162 163 164 165 166

200 201 202

现在给十一进制下个定义：

十一进制是由 0、1、2、3、4、5、6、7、8、9 还差一个符号，用 X 也行，用 A 也行，共十一个符号组成，最小是 0，最大是 X（或 A），逢十一进一。

练习：用十一进制写出十进制的 101 个元素：

0 1 2 3 4 5 6 7 8 9 A

10 11 12 13 14 15 16 17 18 19 1A

20 21 22 23 24 25 26 27 28 29 2A

30 31 32 33 34 35 36 37 38 39 3A

40 41 42 43 44 45 46 47 48 49 4A

50 51 52 53 54 55 56 57 58 59 5A

60 61 62 63 64 65 66 67 68 69 6A

70 71 72 73 74 75 76 77 78 79 7A

80 81 82 83 84 85 86 87 88 89 8A

90 91

现在给三进制下个定义：三进制是由 0、1、2 共三个符号组成，最小是 0，最大是 2，逢三进一。

练习：

用三进制写出十进制的 101 个元素：

0 1 2

10 11 12

20 21 22

100 101 102

110 111 112

120 121 122

200 201 202

210 211 212

```
220 221 222
1000 1001 1002
1010 1011 1012
1020 1021 1022
1100 1101 1102
1110 1111 1112
1120 1121 1122
1200 1201 1202
1210 1211 1212
1220 1221 1222
2000 2001 2002
2010 2011 2012
2020 2021 2022
2100 2101 2102
2110 2111 2112
2120 2121 2122
2200 2201 2202
2210 2211 2212
2220 2221 2222
10000 10001 10002
10010 10011 10012
10020 10021 10022
10100 10101 10102
10110 10111 10112
10120 10121 10122
10200 10201
```

其实学计算机很简单，就是扳着手指头数，如果不是数出来，是算出来的都是错误的想法。数的本质是数出来的。

比如：2+2 = 4 我们可以用手指头一个一个数出来。

课后理解：

各种进制如表 1-1：

表 1-1：1-11 进制表示

进制	实例
1	0，00，000，0000，00000，000000…
2	0，1，10，11，100，101，110，111，1000…
3	0，1，2，10，11，12，20，21，22，100…
4	0，1，2，3，10，11，12，13，20，21，22…
5	0，1，2，3，4，10，11，12，13，14，20…
6	0，1，2，3，4，5，10，11，12，13，14…
7	0，1，2，3，4，5，6，10，11，12，13，14…
8	0，1，2，3，4，5，6，7，10，11，12，13…
9	0，1，2，3，4，5，6，7，8，10，11，12…
10	0，1，2，3，4，5，6，7，8，9，10，11，12…
11	0，1，2，3，4，5，6，7，8，9，A，10，11…

进制其实是 N 种符号组成的。

课后疑问：

本节没有疑问。

课后总结：

进制是由元素组成的，N 进制就是有 N 个元素组成，逢 N 进一

课后练习：

1．在纸上用 1 到 16 进制分别描述 100 个数

2．写一到十六进制,每进制 0-99

3．0 到 16 进制，每个进制写 100 个数。

1.2 进制运算

本节主要内容：

1. 二进制的好处
2. 进制间的运算

老唐语录：

十进制是大家小学时就会的：

0,1,2,3,4,5,6,7,8,9

那么九进制大家也应该都会：

0,1,2,3,4,5,6,7,8

十进制可以加减乘除，那么九进制照样可以加减乘除，直接算出结果，十一进制也可以。既然小于十进制都会，大于十进制都会，那就是说 N 进制都会。难道 N 进制都会，唯独不会二进制吗？

既然你们都会的话，书上为什么还要教二进制呢？还用进制间转换吗？你们可以看一下书上，有十进制转二进制，二进制转十进制，不是多此一举吗？三进制可以直接算出结果，九进制也可以直接算出结果，为什么二进制不可以呢？任何一种进制，他自身就是一个完美的体系结构，直接能加减乘除开方。

练习：

九进制加法：

7+8=16

九进制根本不需要去转换，可以直接算出结果，十进制也是，和其他进制没有关系，自成一个体系结构。任何进制都可以直接加减乘除。那么，同样的，三进制，二进制，八进制，十六进制都是一个完美的体系结构，都可以自己加减乘除，直接算出结果，如果谁去转换成二进制，那说明不懂。二进制可以直接加减乘除算结果，把结果拿来用就可以了。比如：你说你有“110”块钱不行吗？非要用十进制的吗？如果需要加减，可以直接加减，九进制可

以直接加减，为什么二进制不能呢？既然加减乘除都可以，为什么还要去转换呢？现在懂了没有？大家理解所有进制了吗？

小桃 say：“懂了。”

小胡 say：“会了。”

二进制就是由 0 和 1 共两个符号组成，最小是 0，最大是 1，逢二进一，就学完了。

那现在我们给 N 进制下个定义：

N 进制就是由 0、1、2、3、4……N-1 共 N 个符号组成，逢 N 进一。

我们说过了，人类只会数数，不会加减，所以说我们用进制不能用的太大，因为十进制有十个符号，相互之间加减是我们能记住的。二进制就更简单，只有两个符号，两个之间相互加减，都是可以记住的，为什么计算机用二进制？因为二进制最简单，需要记住的东西最少，人类不会计算，当然计算机更不会计算。所以说他只能用最简单的记忆方法来算，因为十进制需要记住的符号太多了。

比如在十进制中，5+4=9，2+7=9，1+8=9，3+6=9，2+8=10，两两相加，符号太多了，很麻烦，相比较而言，二进制要简单的多。严格的说，他只有两个结果 0 和 1，所以非常简单。而其他进制太复杂，三进制，四进制，五进制，六进制都很复杂，所以说基本都不用。人类最常用的是十进制，因为人类有十个手指头，数起来很方便。最方便的数数工具，就是手指。人类天生就会使用十进制，当然也有民族使用五进制的，因为五进制数起来也很方便，一个手五个手指头，一个手专门去数。现在大家都懂了进制了吧？

练习：

用一百进制从 0 写到十进制的 101？现在大家自己在纸上写，或者谁上来写。

小刘 say：“后面咋整？”

老唐 say：“随便你，不是学过了吗？九进制会了，十进制会了，十一进制会了，也就是说大于十进制的都会。小于十进制的都会。难道就不会一百进制吗？”

小李 say：“用什么来表示？”

老唐 say：“随便。”

小桃 say：“写到一百零一只进了一次”

老唐 say：“是啊。”

其实我们学到 0，1，2，3，4，5，6，7，8，9 是阿拉伯人的符号。零，一，二，三……

是中国人的符号，仅仅是个符号而已。

练习：

用一进制从 0 写到十进制的 20，结绳记事用的就是一进制。

0

00

000

0000

……

阴阳八卦就是二进制，宇宙中最和谐最完美的进制就是二进制。有和无，白和黑，无和点（点可以构成画，画可以构成所有）

练习：

用二进制从零写到十进制的 100。

0 1

10 11

100 101

110 111

1000 1001

1010 1011

1100 1101

1110 1111

10000 10001

10010 10011

10100 10101

10110 10111

11000 11001

11010 11011

11100 11101

11110 11111

100000 100001

100010 100011

100100 100101

100110 100111

101000 101001

101010 101011

101100 101101

101110 101111

110000 110001

110010 110011

110100 110101

110110 110111

111000 111001

111010 111011

111100 111101

111110 111111

1000000 1000001

1000010 1000011

1000100 1000101

1000110 1000111

1001000 1001001

1001010 1001011

1001100 1001101

1001110 1001111

1010000 1010001

1010010 1010011

1010100 1010101

1010110 1010111

1011000 1011001

1011010 1011011

1011100 1011101

1011110 1011111

1100000 1100001

1100010 1100011

1100100

练习：

九进制加减法

765-567

注：借一位是 9

练习：

七进制加法

654+321

进制的加减法都会了吧，现在我们来练习乘法

练习：

十进制乘法

8*9

练习：

五进制乘法

3*4

进制间的乘法都会了吧，那现在这道题给你们 10 分钟做出来

练习：

十一进制乘法

789A*A987

注：这个运算比较复杂，因为我们没有掌握它的乘法表，我们只知道十进制乘法表。现在我们来编写其他进制乘法表。

二进制

```
1*1 = 1
```

三进制

```
1*1 = 1
2*1=1 2*2=11
```

四进制

```
1*1=1
2*1=2 2*2=10
3*1=3 3*2=12 3*3=21
```

五进制

```
1*1=1
2*1=2 2*1=4
3*1=3 3*2=11 3*3=14
4*1=4 4*2=13 4*3=22 4*4=31
```

六进制

```
1*1=1
2*1=2 2*2=4
3*1=3 3*2=10 3*3=13
4*1=4 4*2=12 4*3=20 4*4=24
5*1=5 5*2=14 5*3=23 5*4=32 5*5=41
```

七进制

```
1*1=1
1*2=2 2*2=4
1*3=3 2*3=6  3*3=12
1*4=4 2*4=11 3*4=15 4*4=22
1*5=5 2*5=13 3*5=21 4*5=26 5*5=34
1*6=6 2*6=15 3*6=24 4*6=33 5*6=42 6*6=51
```

九进制

```
1*1=1
2*1=2 2*2=4
```

3*1=3 3*2=6 3*3=10

4*1=4 4*2=8 4*3=13 4*4=17

5*1=5 5*2=11 5*3=16 5*4=22 5*5=27

6*1=6 6*2=13 6*3=20 6*4=26 6*5=33 6*6=40

7*1=7 7*2=15 7*3=23 7*4=31 7*5=38 7*6=46 7*7=54

8*1=8 8*2=17 8*3=26 8*4=35 8*5=44 8*6=53 8*7=62 8*8=71

十二进制

1*1=1

2*1=2 2*2=4

3*1=3 3*2=6 3*3=9

4*1=4 4*2=8 4*3=10 4*4=14

5*1=5 5*2=a 5*3=13 5*4=18 5*5=21

6*1=6 6*2=10 6*3=16 6*4=20 6*5=26 6*6=30

7*1=7 7*2=12 7*3=19 7*4=24 7*5=2b 7*6=36 7*7=41

8*1=8 8*2=14 8*3=20 8*4=28 8*5=34 8*6=40 8*7=48 8*8=54

9*1=9 9*2=16 9*3=23 9*4=30 9*5=39 9*6=46 9*7=53 9*8=60 9*9=69

a*1=a a*2=18 a*3=26 a*4=34 a*5=42 a*6=50 a*7=5a a*8=68 a*9=76 a*a=84

b*1=b b*2=1a b*3=29 b*4=38 b*5=47 b*6=56 b*7=65 b*8=74 b*9=83 b*a=92 b*b=101

课后理解：

小陶 say：

我的理解：

我们可以通过十进制的乘法,从中摸索出规律：

456*456

先进行个位相乘：6*6=36，进 3 余 6，等等,我们怎么知道 6*6=36？ 是通过 99 乘法表得到的。同样除法也是如此。找到这个规律，我们就清楚接下来该做什么了：

七进制：

456*456=?

我们把七进制的乘法表写出来:

1*1 = 1

1*2 = 2

2*2 = ? 在 1*2 的基础上加 2 为 4

2*3 = ? 在 2*2 的基础上加 2.

这样，乘法演变成加法。所以很快得出七进制的乘法表:

1*1 = 1

1*2 = 2 2*2 = 4

1*3 = 3 2*3 = 6 3*3 = 12

1*4 = 4 2*4 = 11 3*4 = 15 4*4 = 22

1*5 = 5 2*5 = 13 3*5 = 21 4*5 = 26 5*5 = 34

1*6 = 6 2*6 = 15 3*6 = 24 4*6 = 33 5*6 = 42 6*6 = 51

小胡 say:

计算机只会加法，现实中为了计算方便所以建立了乘法表。

课后疑问:

111-111 = ?

有的人说等于 0。那么，如果我使用 0 作为一进制的符号:

00 + 000 = 00000

000 - 00 = 0,

那么 000 - 000 = ?

说明:

其实我们不用纠结于算数的结果,111-111 = 空,这个空可以用其他符号代替,只要不是该进制使用过的符号即可.

课后总结:

任何一种进制，他自身就是一个完美的体系结构，可以直接加减乘除开方。

课后练习：

1．对照 9*9 乘法表，建立 6*6 乘法表。

2．将 16 进制的元素用 2 进制的元素下定义

3．把 16 进制的元素和二进制对比，发现其中的规律

1.3 十六进制与数据宽度

本节主要内容：

1．进制间的运算

2．计算机计数与数学计数的区别

3．圆圈实现数字循环

4．计算机中负数的表示

老唐语录：

进制现在理解了吗？

在自然界都只有二进制存在。有二进制已经足够了，完全没必要有其他进制。其实自然界也是按这个发展的。学过生物的都知道细胞的分裂。没有说细胞一次性分成三个细胞，四个细胞的吧？只有一个细胞一次分成两个细胞，两个细胞再分成四个细胞，然后八个细胞。无论是单细胞还是多细胞，都是一个变二个，两个变四个，都是按二进制来的。呈二进制的指数增长。而不是十进制的指数增长，更不是其他进制的。十进制只是人类特殊创造的。

二进制有个问题，虽然很方便。也可以做很大，你看二进制书写很麻烦。

11111111111111111111111111111111，也就是说二进制表示现实生活中的数，比如你家里多少钱的时候，也并不需要多少位。二进制表示生活中的数也是很短的，并不是很长。32 位，已经很大了，4 个 G。我们说人类特别喜欢用十进制去写。比如 256，使用二进制写会很长（100000000B），你用二进制写会很长。既然你用二进制写太长了，要书写简化，我们书写简化一下二进制。比如二进制我们书写：

10101010111010

太长了，我们两位两位的书写，我们用个符号来表示，随便一个符号，比如用 B 来表示。有没有发现，两位两位的书写，最多出现四种可能是吧，00，01,10,11。这四种可能，那么两位两位的书写很方便啊，随便用四个符号代替就行。虽然说二进制有点长，但二进制毕竟是宇宙中最简单的。人类毕竟不习惯，为什么？因为太长。我们两位两位的书写。两位

最多要求 0,1,2,3.再没别的值了吧。也就是说只要定义四种符号去书写他就可以了。那我们还是觉得长，那好。我们三位三位的书写。那三位最多的可能呢？

100,101,110,111

最多只有八种，那我们用 A,B,C,D,E,F,G,H,去表示。010 在这是 C 是吧。111 是 H 是吧，010 是 C，101 是 F，这个 010 是 C，CFCHC。那好了，我们也可以同理，我们照样可以四位四位的书写。也可以五位五位的书写。六位六位的书写，七位七位的书写。但是有个问题有没有发现。我们四位四位的书写，就会发现一个问题，符号会越来越多。

1000，1001,1010,1011,1100,1101，1111，I,J,K,L,M,N,O。不需要排顺序，只要你使用这十六种符号表示这十六种值就可以。那好了，我们查表。1010，是 K，1011 是 L，1010 是 K，0010 是 C，CKCK。导致多位多位的书写。整体变得很短了。问题就是这边要记的符号越来越多了。是不是啊？那当然你还可以五位五位的书写。五位五位书写更短了是吧？那现在要记多少个符号啦，每多一位，符号要增一倍。两位两位的书写只要记 4 个符号。三位三位书写八个符号。四位四位书写 16 个符号。五位五位书写 32 个符号。六位六位书写 64 个符号。如果七位七位书写呢？那就 128 个符号了。那为什么当初不让你们用一百进制，那就是符号太多，记不住。人类只会记和数，不会算。二进制的书写对于人类来说毕竟是太长了，所以我们采用简单的方法。就是说，两位两位，三位三位，四位四位，或者五位五位，六位六位，七位七位的书写。到底你喜欢多少位书写呢？看你喜欢记的符号多还是少呢？比如：7 位 7 位的书写，你要记 128 个符号；八位八位的书写，你要记 256 个符号。如果你基本功扎实，记得这么多符号，那你写的时候就要容易一点。比如：你的文学功底比较深厚，字学的比较多，词语比较多，成语比较多，那你写的文章就漂亮，并且简短，精炼。那这样，只要你符号记得足够多，那么你书写起来就简短精炼。当然，你符号记得越少，你写的就越长。“记流水账嘛。”如果你字都不会写，就是写的再长，都是反复的这么说来说去，没什么内涵，也没什么文采。

二进制有两种，二进制本身是一种，还有一种是二进制的书写形式。书写形式有很多种：一位一位的书写，两位两位的书写，还是三位三位的书写，看你书写形式。

我们现在书写 A,B,C，D，我是两位两位的书写，自然这个 D 是 11，C 是 10，B 是 01，A 是 00。这两种书写形式表示的含义是一样的。上面是书写形式，下面是本质形式。当然本质形式也是其中一种书写形式。也就是说一个汉字他可以有草书，隶书。当然一个二进制也有很多种书写形式。本身二进制这东西只有一个，但是名字有很多种。比如：一个人有几

个名字，一个地方有几个名字，一个物体拍照片可以拍很多张，但是你都认识他。当然了，你也可以一位一位的书写，我用 A 表示 0，B 表示 1，就是 BBABBAAA……这也是二进制的书写形式。你可以另外定义两个符号，未必要用用原来的 0 和 1。另定义两个符号用 A 和 B 两个符号。也可以两位两位的书写，用 ABCD 来表示。或者是三位三位的书写。

那到底是几位几位的书写更方便呢？

其实很简单。就是因为在全世界阿拉伯数字的符号最简单，我们就不应该用 ABCD 这个符号去表示。我们到底使用 1 位 1 位，2 位 2 位还是 3 位 3 位 4 位 4 位 5 位 5 位书写呢，很简单，我们找这符号，这个符号不能太多，256,128,64,32，都显得多了，就算不多，那我们讨论到底多少适合，我们喜欢阿拉伯数字的符号，那我们就用阿拉伯人的符号来表示。阿拉伯人的符号太完美了，所以我们不能使用三位三位的书写，因为会浪费两个 8 和 9，太可惜了。所以我们一定要大于三位三位的书写。因为我们不要浪费这个符号。就使用四位四位的书写，才能保证把阿拉伯人的符号全部用上。那四位的书写，少了几个符号，那用谁的符号呢？我们发现其实英国人也很聪明，英国人的符号也不错。少几个符号我们用英国人的符号补上去。毕竟英国人没有古印度人聪明，我们补太多的符号，不便于记忆，所以使用四位四位的书写。这不是十六进制，这是四位四位的书写二进制。跟十六进制没有一点关系。你只要简单的记住这十六个符号来表示这十六种值就可以了。

以后写二进制四位四位的书写：

二进制从 0000 写到 1111

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
```

1011

1100

1101

1110

1111

然后改成 16 个符号。

如果按照这种新的书写方式，那你们发现如果要进位的话，要满 16 才进位。中国古代就知道用二进制数，并知道四位四位的书写。这就是我们平时说的半斤八两，其实就是二进制四位四位的书写导致的结果。

你们用过算盘没有？我们平时只用下面四个珠子，上面一个珠子，为什么上面两个珠子，下面五个珠子？这就是二进制，喜欢四位四位的书写，最小值是 0，最大值是 15，下面加上上面值为 15，不拨的时候 0，全拨的时候 15。这就是最早的计算机。有的银行现在仍然使用算盘，加减法还是算盘快，乘除法计算机快。

现在开始练习二进制的两种书写方式：一位一位的书写，四位四位的书写。

数学上的数，无论多长，都能表示出来，是没有宽度的。算盘宽度是有限的。

凡是涉及到机器上的数，宽度总是有限的，计算机中的数是用电路表示的，计算机成本是有限的，所以表示宽度是有限的。

这里为了书写方便我们只写 8 个位：

-1=0-1

00000000

-00000001

计算机在计算数之前，要规定这个数有多宽。-1 是 0xFF。

例：从 0，写到-50。

例：写 4 位，最大值与最小值可以交汇到一点，形成一个圆。

说明：见图 1-1：

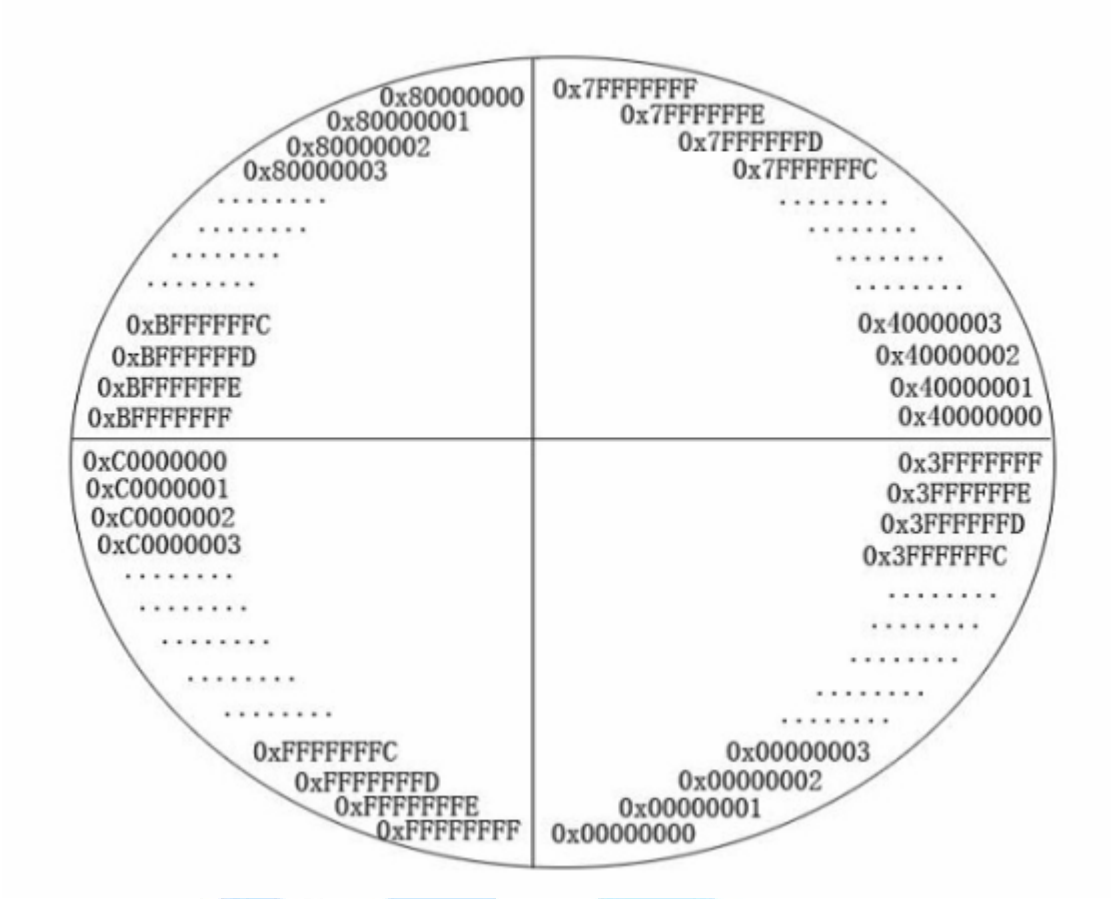


图 1-1：计算机宽度

课后理解：

在数学运算中，数值的大小（即宽度）是没有限制的，比如 100000*100000=10000000000。在计算机中，在对数值进行运算前要先规定其宽度,再进行运算.比如给定的一个数 0x123456789ABC，如果规定它的宽度为 32 位（这里指的 32 通常是二进制的 32 位），那么该数值的有效值为 0x56789ABC.如果运算结果超过其宽度将被略去，只保留有效位。

例：设给定的数值宽度为 8 位（二进制），则可以表示的最大值为 0xFF（11111111B）。此时，0x81+0x80= 1000 0001B+ 1000 0000B= 1 0000 0001（B） = 0000 0001（B） = 0x1。图 1-2 显示了计算机常用的数值宽度：

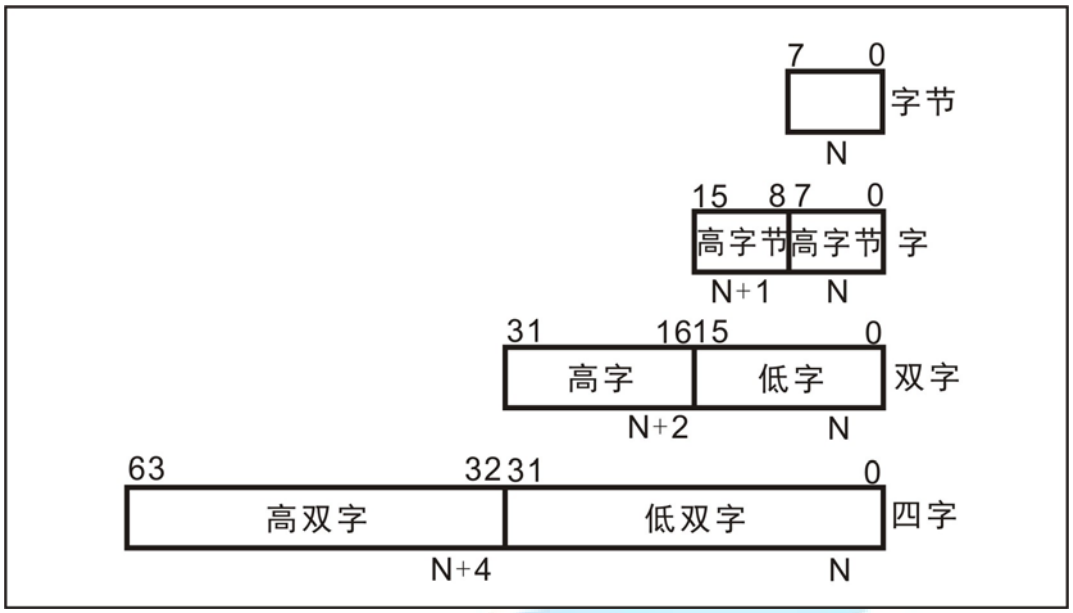


图 1-2：计算机数值宽度

如上图所示，字节（Byte）占据 8 位二进制宽度；字（Word）占据 16 位二进制宽度；双字（Doubleword）占据 32 位二进制宽度；四字（Quadword）占据 64 位二进制宽度。

在计算机的数值运算中，不仅要规定数值的宽度，同样要设置数值的符号和精确度。比如正数，负数和浮点数。针对这种情况，计算机将数值定义为有符号或无符号类型：当为有符号类型时，最高位作为符号位，最高位为 1，表示负数，反之为正数，见图 1-3：

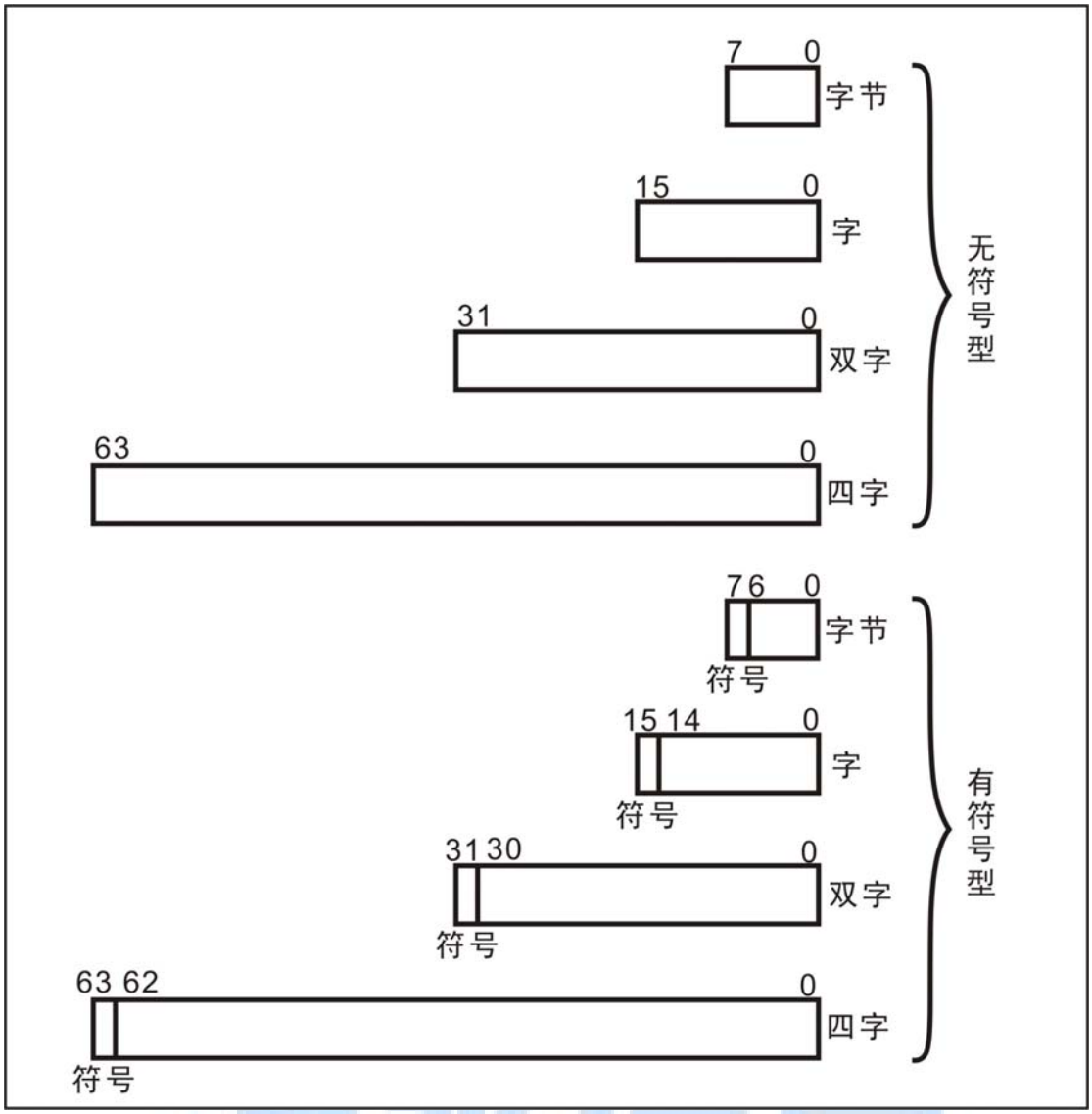


图 1-3：有符号数表示

有符号数值大小，见表 1-2：

表 1-2：有符号数

位宽度	最小值	最大值
8	0x80(-2^7)	0x7F(2^7-1)
16	0x8000(-2^{15})	0x7FFF(2^{15}-1)
32	0x8000 0000(-2^{31})	0x7FFF FFFF(2^{31}-1)

课后疑问：

本节没有疑问。

课后总结：

计算机是定宽的。

课后练习：

2，4，16 进制，每进制最大 32 位，写出 8 个主要点

1.4 逻辑运算

本节主要内容：

1．布尔代数与二进制的关系

2．布尔代数的运算

3．VC++6.0 的使用

老唐语录：

上节我们讲到，计算机是定宽的

比如：一个二进制数 1111 1111 1111 1111B

我们用四位，四位书写的式 FFFF 这样比较简单

今天我们讲，计算机除了算术运算外，还有逻辑运算，只有二进制才能进行逻辑运算。逻辑运算中只有错与对，成与败两个结果（也就是 0 和 1）。

或运算：

0+0=0

0+1=1

1+1=1（1+1=2 不等于 0，就是 1）

“+”等价于“或”，计算机中使用“|”符号代替。汇编语言使用“OR”代替

与运算：

0*0=0

0*1=0

1*0=0

1*1=1

“*”等价于“与”，计算机中使用“&”符号代替。汇编语言使用“AND”代替

异或运算：

0-0=0

1-0=1

0-1=1

1-1=0

“-”等价于“异或”，计算机中使用“^”符号代替。汇编语言使用“XOR”代替

三种运算在生活中怎样存在？

电路如图 1-4：

OR	0	1
0	0	1
1	1	1

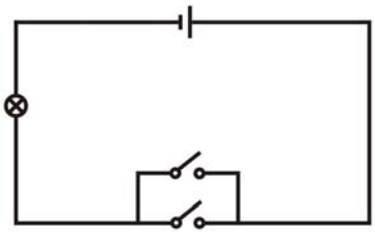


图 1-4：OR 运算与等效电路

说明：看图 1-4，这是一个并联电路图，任意一个开关按下时，灯泡都会亮。

所以说二进制运算是客观存在的，

是因为太冗长，所以发明了十进制

与运算电路如图 1-5：

AND	0	1
0	0	0
1	0	1

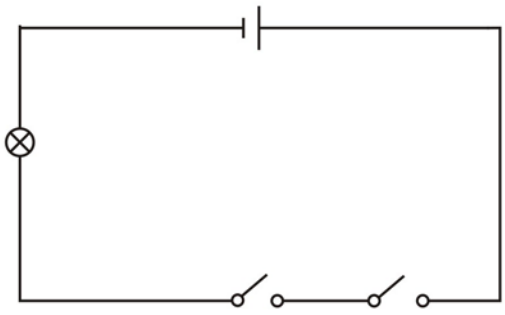
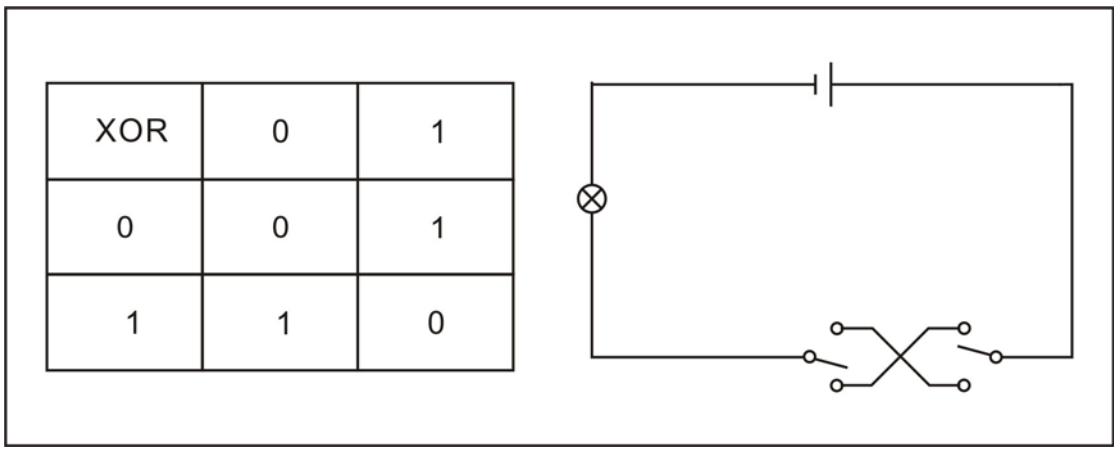


图 1-5：AND 运算与等效电路

说明：看图 1-5，这是一个串联电路图，只有两个开关同时按下时，灯泡才会亮。

异或运算电路如图 1-6：



XOR	0	1
0	0	1
1	1	0

图 1-6：XOR 运算与等效电路

说明：看图 1-6，这是一个交叉电路图，只有两个开关一上一下时，灯泡才会亮。

与运算：比如一个家庭，只有丈夫和妻子都有生育能力，才能有孩子。

异或运算：在法院势均力敌的两方无法分出胜负，只有一方高于另一方，才有胜负。

练习：

我们学习计算机必不可少的软件 VirtualC++6.0

现在我们新建一个工程，步骤如下：

1. File->new，弹出下面窗口，点击 Win32 Console Application，然后在 Project name 框中填入工程名，例如“HelloWorld”，点击 OK，如图 1-7：

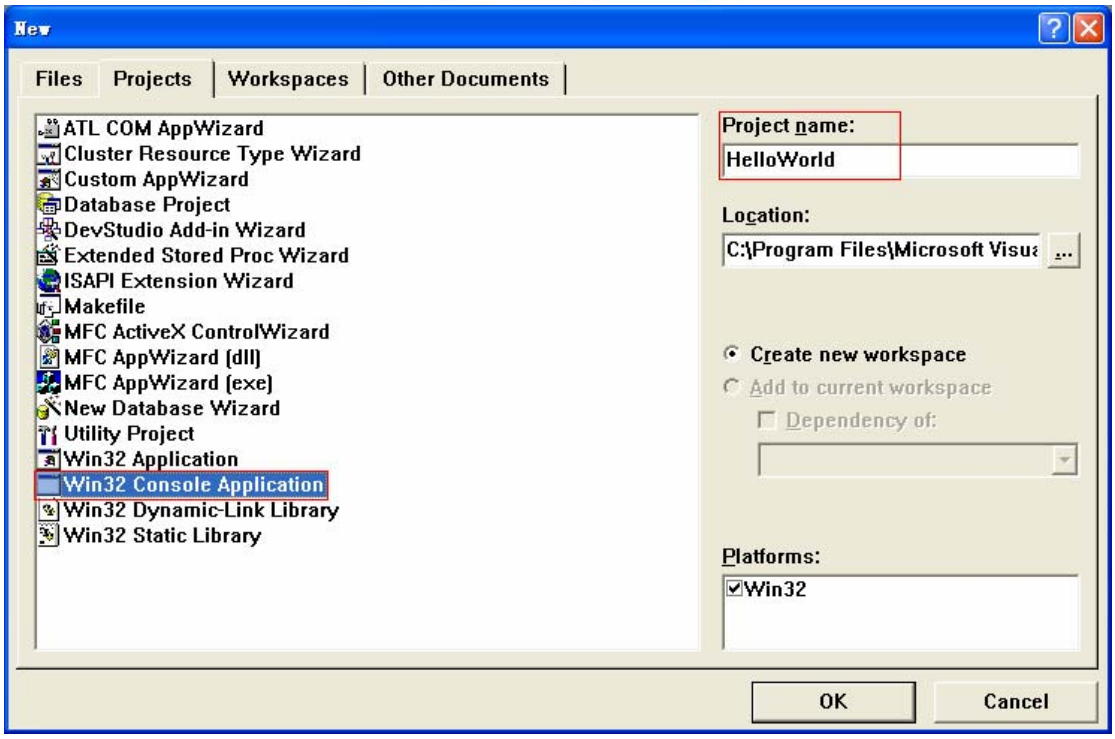


图 1-7：新建工程对话框

2.在图 1-8 中选择第三项：“A 'Hello,world' application”，然后点击“Finish”。

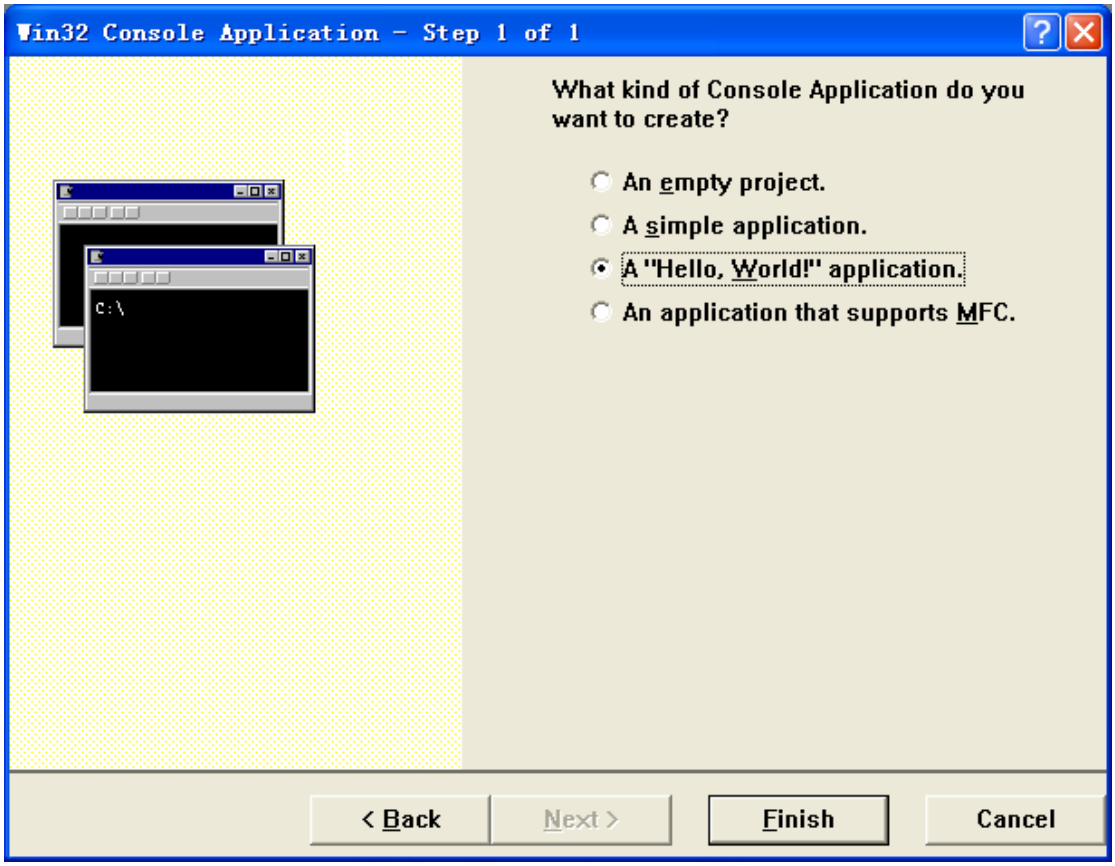


图 1-8：win32 Comsole Application

操作：

1> 按 F11 键单步跟踪每一条语句，并打开寄存器窗口查看寄存器的值变化，如图 1-9：

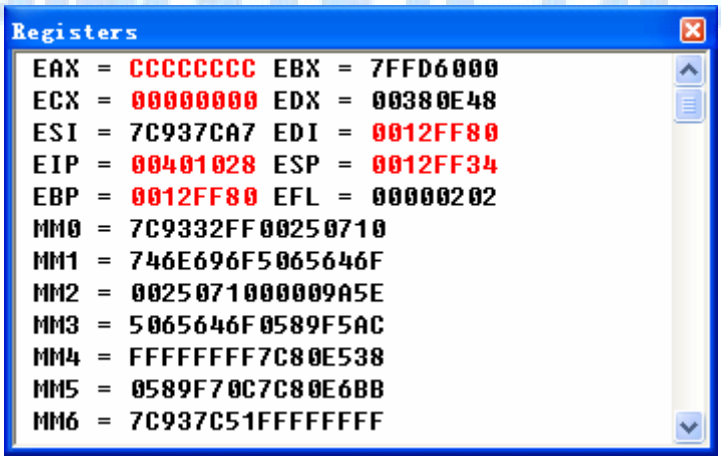


图 1-9：寄存器窗口

2> 添加如下语句，观察对应寄存器变化：

```
__asm{
mov eax,0x33434
mov ecx,0x124
...
}
```

课后理解：

小陶 say:

或运算（|）：只要有一个是 1，就是 1，其它为 0

与运算（&）： 两个为 1 才是 1，其它为 0

异或运算（^）：相同的为 0，不同的为 1

小胡 say:

看了楼主的理解，我懂了

课后疑问：

本节没有疑问。

课后总结：

二进制实现了逻辑运算和算术运算的统一。

课后练习：

1.2、4、6 进制，每个进制写 32 位。0-FFFFFFFF

2.写 10 个寄存器，背熟顺序

第二章 寄存器与汇编指令

引言

想要了解计算机，首先要了解的便是 cpu，cpu 是计算机最核心的部件，因为计算机的所有指令都是由 cpu 处理，而 cpu 的核心部件之一是寄存器。这一章我们就来认识一下寄存器以及寄存器是如何工作的。

本章必须要掌握的知识点：

1．8、16、32 位通用寄存器

2．寄存器与内存的区别

3．汇编语言的基础指令

本章常犯的错误：

1．内存的存储格式

2．溢出标志位（OF）的理解

3．pop 与 push 指令的理解

2.1 通用寄存器

本节主要内容：

1．8/16/32 位通用寄存器

2．汇编指令

老唐语录：

计算机最经典的指令就是移动指令：mov ecx，eax；

主要记住这 8 个寄存器：

eax

ecx

edx

ebx

esp

ebp

esi

edi

mov 指令可以任意移动这 8 个寄存器。

在 mov ecx，eax 中，后面的是源，前面的是目标，中间是逗号，不区分大小写。寄存器间相互移动。

如果计算机只有移动指令的话，那么什么事也干不了。

mov 是操作码，两个寄存器是操作数，操作码除了 mov 之外还有很多，你可以替换：加，ADD；减，SUB；与，AND；或，OR；异或，XOR；非，NOT。

比如：add eax,ecx。

当然了，操作码不仅仅只有这几个，还有很多很多，只要搞清楚操作数是任意两个寄存器，当然可以是同一个寄存器。

比如 or esi，esi：将 esi“或”esi 并将结果赋给 esi。操作数除了寄存器之外，还可以是一个数，只要保证是 32 位即可。

8 个寄存器是分段的。比如算盘，我们可以只用一半，或者四分之一。

eax 可以分成四部分：eax 共 0-31，其中 0-7 位叫做 AL，8-15 位叫做 AH。整个 0-15 位又称为 AX。AL：low；AH：high。ecx，edx，ebx 也是一样的，如图 2-1。

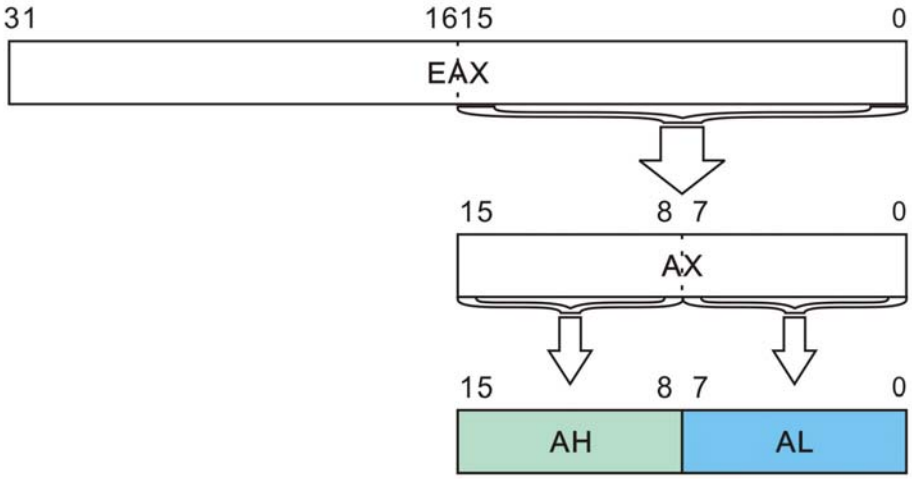


图 2-1：EAX 寄存器

以上四个寄存器都可以分成三段。

ESP,EBP,ESI,EDI，这四个寄存器只分成两段，比如 ESP 整体 0-31 位称为 ESP，0-15 位称为 SP。

通过以上，我们可以看出，32 位的寄存器有 8 个：EAX，ECX，EDX，EBX，ESP，EBP，ESI，EDI，每个寄存器都有一个编号：0 号，1 号，2 号，3 号……还有 16 位的寄存器：AX，CX，DX，BX，SP，BP，SI，DI。也是一样的：0 号，1 号，2 号，3 号…… 如表 2-1：

表 2-1：寄存器编号

寄存器			编号（二进制）	编号（十进制）
32 位	16 位	8 位		
EAX	AX	AL	000	0
ECX	CX	CL	001	1
EDX	DX	DL	010	2
EBX	BX	BL	011	3
ESP	SP	AH	100	4
EBP	BP	CH	101	5
ESI	SI	DH	110	6
EDI	DI	BH	111	7

32 位寄存器有自己的编号，16 位寄存器也有属于自己的独立的编号。当然，他们是重叠的，当改变了 32 位的寄存器，相应的 16 位寄存器也会跟着改变。

同样，也有 8 位的寄存器，第 0 号 AL，第 1 号，CL，DL，BL，AH，CH，DH，BH。顺序不能乱。当然还有两个寄存器：EIP 和 EFLAGS（又称为 EFL），8 号和 9 号寄存器，EIP 有 16 位，叫做 IP。EFL 的 16 位称为 FL。这两个寄存器使用相对较少。

练习：

除了 EIP 和 EFL 之外，其他寄存器都是可用的，使用 MOV，ADD，SUB……等指令做练习，比如 XOR AL，BH，后面随便跟两个寄存器，用逗号隔开，当然，前后宽度要一样。而不可以是 XOR AL，BX。后面的寄存器可以改成立即数。

课后理解：

通用寄存器即 cpu 常用的寄存器。Intel 手册给出了通用寄存器的功能：通用目的寄存器（General-Purpose Registers）主要实现逻辑和算术运算、地址计算和内存指针。

通用寄存器结构见图 2-2：

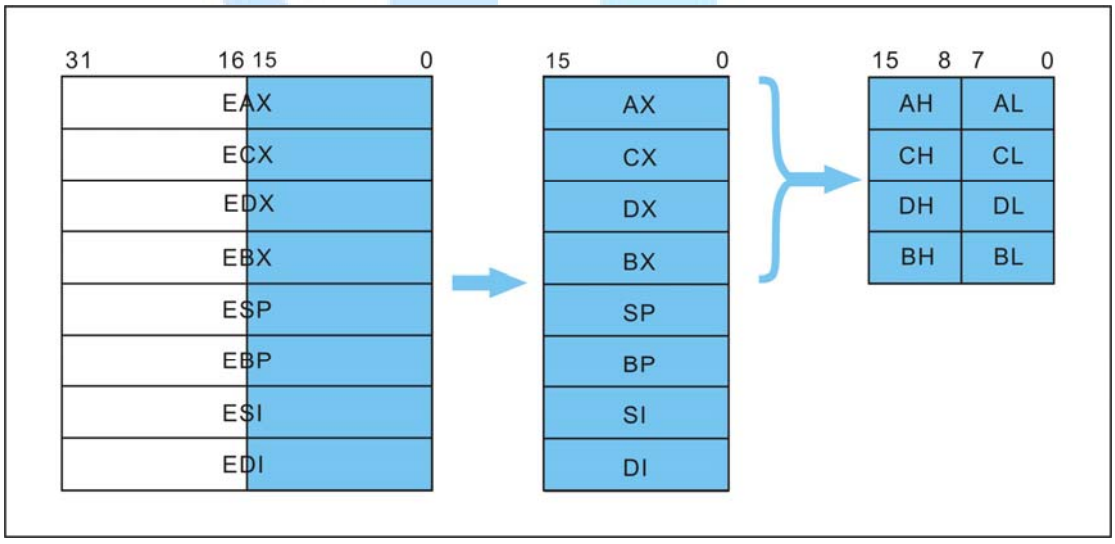


图 2-2：通用寄存器结构

为什么上图看上去比较奇怪，有的寄存器被分成一块一块？其实都是历史遗留问题，在早期的技术还没有现在成熟时，8 个通用寄存器宽度是 8 位（这里指的是二进制的 8 位）：AL，AH，CL，CH，DL，DH，BL，BH。后来 cpu 技术逐渐发展，由 8 位发展成 16 位，于是寄存器由 8 位演变成 16 位：AX，CX，DX，BX，SP，BP，SI，DI。但是为了兼容前面的 8 位技术，将之前的 8 个 8 位寄存器封装到 AX，CX，DX，BX 寄存器中。技术发展的脚

步不会停歇的，之后将 16 位寄存器都扩充了一倍，于是 32 位寄存器出现了，当然这里只截至 32 位，64 位和 128 位寄存器留给大家思考。

32 位通用目的寄存器的指定用途如下：

- EAX：累加器（Accumulator）
- ECX：计数（Counter）
- EDX：I/O 指针
- EBX：DS 段的数据指针
- ESP：堆栈（Stack）指针
- EBP：SS 段的数据指针
- ESI：字符串操作的源（Source）指针；SS 段的数据指针
- EDI：字符串操作的目标（Destination）指针；ES 段的数据指针

以上只是让大家对寄存器的缩写的含义了解一下，毕竟使用一样东西，不明白其中的道理，实在不高明，而且这东西是老外发明的，按中式逻辑很难猜出来。

课后疑问：

问题：EIP 在不断变化，我们不但要有命令去操作这些寄存器，还需要区分当前用的是哪一条命令，下一次再用哪一条命令。

回答：**cpu 具有判断指令长度和预处理指令的功能。**

为什么你写的寄存器顺序是 EAX，ECX，EDX，EBX……而不是 EAX，EBX，ECX，EDX……，不是更容易记住?

回答：其实我也想，可是 Intel 的技术员不这么认为，他们就是这么定义寄存器顺序的。**了解寄存器的顺序和编号，对后面将要介绍的汇编指令和硬编码尤为重要。**

课后总结：

汇编就是在“寄存器与寄存器”或者“寄存器与内存”之间来回移动数据。

课后练习：

32，16，8 位寄存器用 mov add AND sub or xor not 演算

滴水信息

2.2 内存

本节主要内容：

1．寄存器与内存的区别

2．寄存器,内存

3．内存修改

老唐语录：

快速算盘叫做寄存器。慢速的称为内存。其实他们的结构差不多，都是定宽的，最重要的一点，寄存器速度非常快，价格非常昂贵，所以在 CPU 内部。做的数量也很有限。常用的只做了 8 个：EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI。内存速度慢，很便宜，所以数量做的非常庞大。寄存器做的数量非常少，就可以为每个寄存器取个名字。而内存数量太庞大了，没办法给每一个都取上名字，所以只能编号。最后重要的是内存中寄存器的编号是 32 位的。这也是我们现在的计算机叫 32 位计算机的主要原因。如果按寄存器的宽度是 32 位的话，是不对的，因为还有很多寄存器是大于 32 位的。我们通用的 8 个是 32 位的。而内存地址是固定的 32 位。以前的计算机之所以称为 16 位计算机，就是因为他的内存单元编号是 16 位的。

我们学过的指令 MOV，能操作寄存器和内存。

所以说，汇编可以用一句话概括：汇编就是在寄存器和寄存器或寄存器和内存之间来回移动数据。就是指数据在内存和寄存器间来回流动，流动的多了，代表程序很复杂，比如 office 等一些大型软件。

mov eax，0x401000//给 eax 初始化

MOV 也可以改成 ADD（加），SUB（减），XOR（异或），OR（或），AND（与）。

后面的操作数为源操作数，前面的操作数为目标操作数，最开始是操作码。

其实 ADD……这些也是移动指令：把源加上目标然后移动到目标操作数里面去。

前面表示操作方式,后面表示寄存器和寄存器或寄存器和内存,但是只能出现一个内存。

比如：ds:[]里面写个数，表示内存单元。ptr 指的是指针 point。word 是两个字节，还有 byte 是一个字节，dword 是四个字节。

写法：byte/word/dword ptr ds:[32 位的数]。

其实[]里面不但可以写具体的一个数，还可以写某一个寄存器的值。

方括号的通用格式：reg+reg*数+立即数，只要这个值算出来指向哪一个内存单元，就是那个内存单元（现在先记住，不要管为什么，后面会讲）。

第一个寄存器叫做 BASE 寄存器（8 个寄存器都可以），第二个寄存器是 INDEX 也是 8 个寄存器之一，后面乘一个数 scale（1，2，4 或 8，也就是 2 的 0 次方，2 的 1 次方，2 的 2 次方，2 的 3 次方），后面再加一个数（DISP）。

BASE+INDEX*（1，2，4，8）+DISP

可以是有以下五种组合

BASE

INDEX*（1，2，4，8）

DISP

BASE+INDEX×（1，2，4，8）

BASE+INDEX×（1，2，4，8）+DISP

每一种组合都可以。

我们只有把最简单的东西用熟了，才能熟能生巧。

练习：

mov eax,ds:[0x401000+8*eax+eax];这个可以执行吗？

cpu 实现了将地址赋给寄存器：LEA 指令把方括号里面内存的编号给目标寄存器：load effective address。

当一个值不好确定宽度时，使用 dword ptr 或者 word ptr，byte ptr，源可以是内存或寄存器，也可以是立即数，但是目标不能是立即数。两方都可以是内存，但是内存只能在一个地方出现。两边的数据宽度要一样。所以指令是由操作码和操作数组成，操作码是表明我想干什么。复杂的指令也是由简单的指令组合而成。

课后理解：

内存同样由许多寄存器组成，但是此寄存器非彼寄存器，速度不及通用寄存器的几十分之一，宽度为 8 位。见图 2-3：

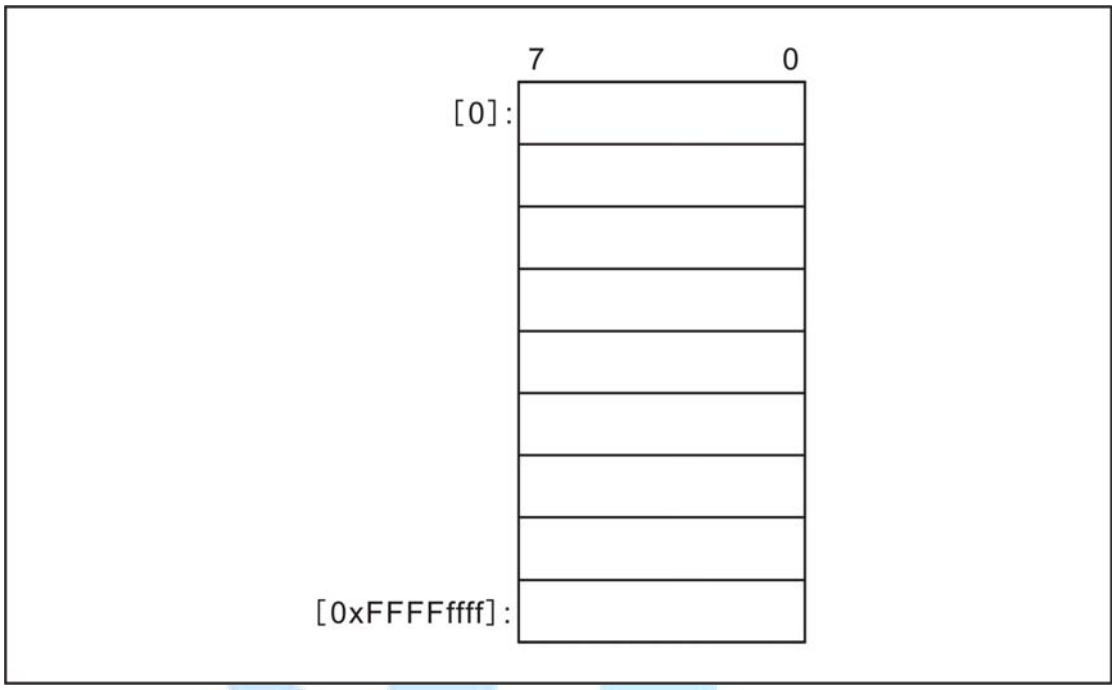


图 2-3：内存格式

如图 2-3，为了区别内存和 cpu 内部的寄存器，我们将内存中的寄存器打上“[]”，专业术语称之为“地址”。图中的内存大小从 0 到 0xFFFFffff，也就是说有 0xFFFFffff 个内存寄存器存在，内存地址从 0-0xFFFFffff。

注：此处 0xFFFFffff 写法为何不写成 oxFFFFFFFF？

十六进制不区分大小写，这样写是为了便于识别，我们可以很清楚的看出有 8 个 F。

内存地址的用处是什么？

当用户运行程序时，cpu 需要不停地去从存储区取代码和数据，这样非常耗时，于是 cpu 先将可能用到的代码和数据从存储区全部放入内存，再从内存中取数据和代码。看似多了一个过程，但是从内存中获取比存储区快得多，所以节省了很多时间。

我们如何使用内存地址呢？

我们通过实例来了解。

将 32 位数 0x12345678 存入内存中的 0x12FFB8 地址处：

说明：我们从图 2-4 可以看出：数值的高位存储在内存地址中的高位。

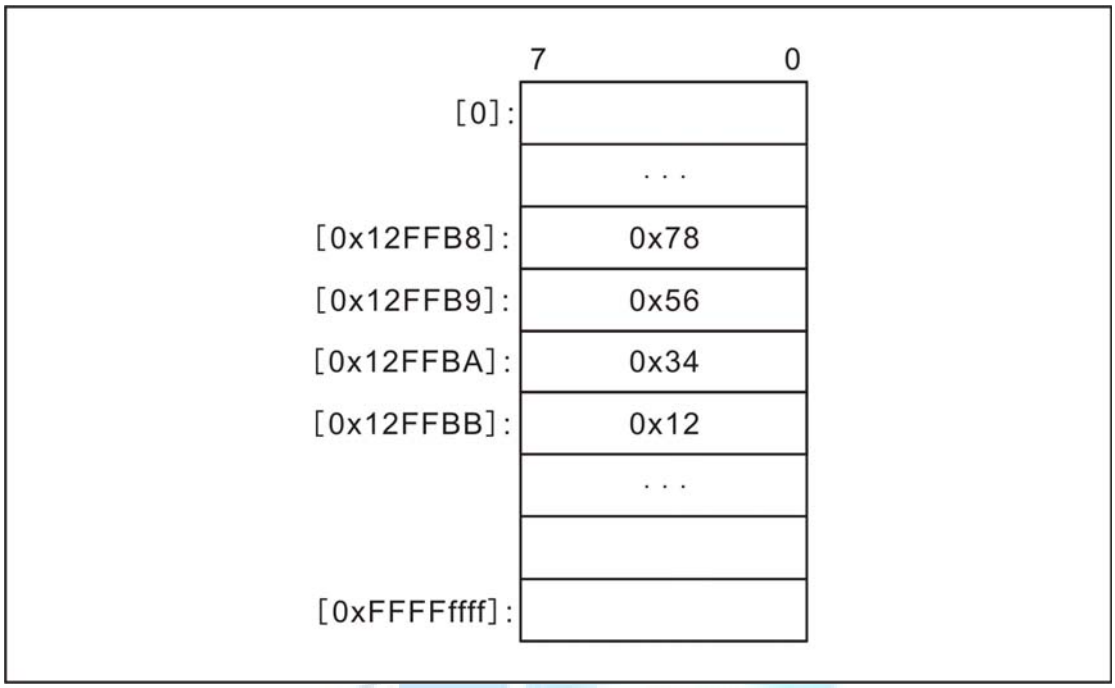


图 2-4：内存存储方式

内存地址的表示方法有哪些?

内存地址的表示方法有很多，除了上图中的表示方法外，还有其他四种表示方法。

以下是内存地址表示方法的组成成员：

- **位移**（Displacement） - 8 位、16 位或 32 位值
- **基**（Base） - 通用目的寄存器
- **索引**（Index） - 除 ESP 外的通用目的寄存器
- **比例因子**（Scale Factor） - 1，2，4 或 8

下列五种地址模式为常用组合（图 2-3）：

- 位移
- 基
- 基 + 位移
- （索引×比例因子）+ 位移
- 基 + （索引×比例因子）+ 位移

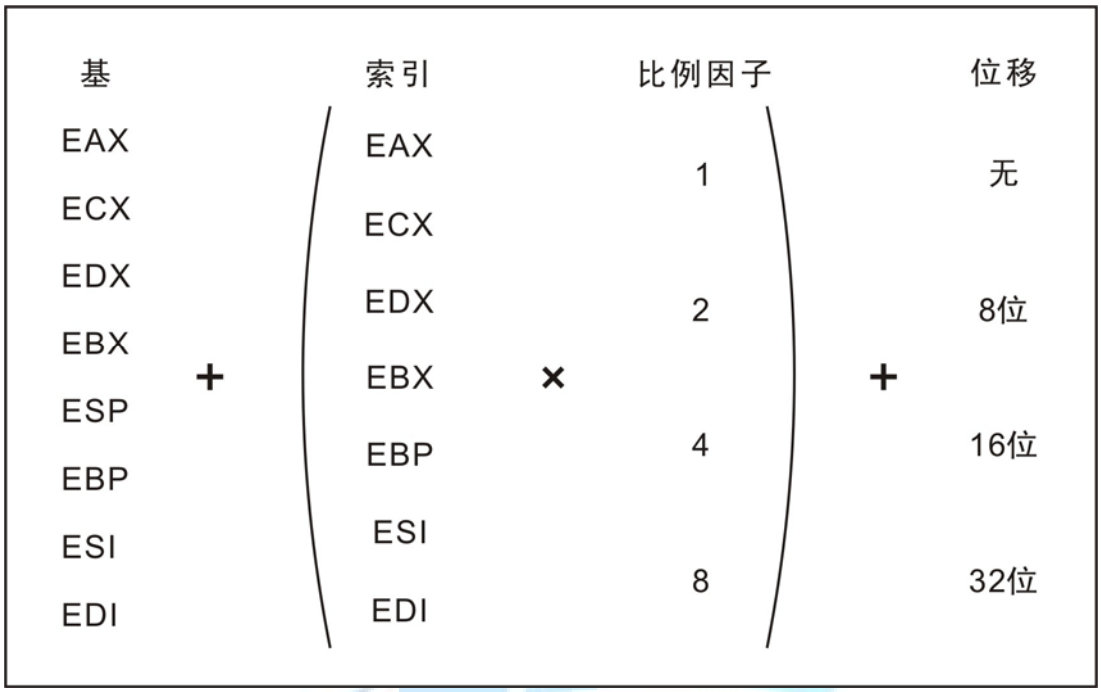


图 2-5：偏移（有效地址）计算

为什么只有五种表示方法，而且比例因子还有限制？

极有可能的原因是（猜测）：计算机只识别机器语言，所以我们要将内存地址的表示方法翻译成机器语言才能得到执行。组合越多，翻译起来越麻烦，cpu 的技术员们只好订个规矩：只能使用五种表示方法，否则一律不识别。

练习：

用实例表示以上五种组合方式。

1.[0x234791]//vc6 不支持

2.[reg]

3.[reg+0x10234]

4.[reg+reg*{1,2,4,8}]

5.[reg+reg*{1,2,4,8}+0x1234]

注：reg 表示通用寄存器。

课后疑问：

1. scale 可以是其他值吗？

回答：不可以。

2．如果算出的内存地址结果超过 32 位会怎样？

回答：如果结果超过 32 位溢出，则计算机只取 32 位。

课后总结：

内存的通用格式：reg+reg*数+立即数

课后练习：

练习复杂地址表达形式的操作如：`lea [eax +eax*0x02+0x12548],eax`

2.3 汇编指令

本节主要内容：

1. 内存地址
2. 汇编语言的基础指令（push、pop、call）

老唐语录：

复合指令：

```
lea esp,[esp-4]
mov [esp],eax
```

使用 vc6 调试观察内存的变化：

这两条指令的组合就是我们学的新指令：**push** eax。

```
mov eax,[esp]
```

lea esp,[esp+4]就是指令 **pop** eax。

练习：

将 eax 替换成其他 7 个寄存器并测试

总结：

pop ERX 也可以理解为：

```
lea esp,[esp+4]
mov ERX,[esp-4]
```

所以 pop esp 可以简化为

```
lea esp,[esp+4]
mov esp,[esp-4]
```

练习：

在 VC6 中单步过程时，可以在代码窗口右键 go to disassmbly，一步一步运行，并观察 eip 变化。

其实 eip 一直在变化。在 80x86CPU 中，eip 寄存器指向的是将要执行代码的位置，代码本身也是数据，也是由二进制构成。

Intel 指令的长度不同，最短的只占一个字节，最长 15 个字节。

比如:

```
xchg eax，eax
mov eax，0x1234567
```

通过 VC6 中的汇编窗口，右键 code bytes，可以看见指令的二进制内容。

xchg eax，eax 又称为 nop。

数据包含代码，代码只是数据的一部分，eip 指向的位置才称为代码。

前面提到计算机和数学（算盘）的不同点是计算机是定宽的。

第二个不同点是数学中的算盘只负责存储数据，并是由人操作的，而计算机相当于有两个算盘，其中一个存储数据，另一个负责操作算盘。

操作和运算本身也是代码，并存在于内存中。但是代码本身不能存放在寄存器，只能放在内存中，只有数据可以放在寄存器中。但是专门有一个寄存器负责指向执行代码（操作算盘的人在哪里，在内存中的哪个位置)。eip 是寄存器，所以他的值可以被改变。

计算机取名字是有规则的，以字母、数字或下划线组成，并且不以数字开始。

mov eax,offset 标号名（标签)；用名字的时候要加个 offset，其实就是立即数

每个地方都可以取个标签，如:

```
lab:  mov eax,ecx
      mov ecx,offset lab
```

既然汇编指令就是在寄存器和内存间移动指令，那么我们可以使用 mov 来修改 eip：

```
mov eip,寄存器/立即数
```

简写为 （取别名)：

```
jmp 寄存器/立即数
```

此外，

```
push offset lab
mov eip,eax
lab :...
```

称为 call eax

ret 就是

```
lea esp,[esp+4]
mov eip,[esp-4]//jmp [esp-4]
```

的简写形式。

ret 后面可以加一个数 0x04，0x08……这个数是两个字节的宽度：

ret IW，比如 ret 0x4

课后理解：

- **PUSH**：压入堆栈，见图 2-6：

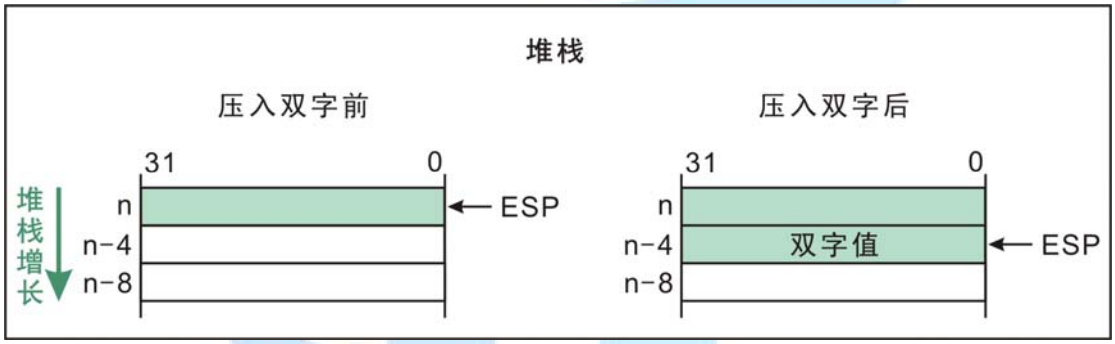


图 2-6：PUSH 指令图解

esp 指向的堆栈首地址，位置在内存中。由图 2-5 得出，所以当需要往堆栈中添加值时，堆栈首先要减 4 个字节（因为堆栈增长方向是由高地址到低地址），然后将值填入新位置。

简化 PUSH eax：

说明：

```
sub esp,4
mov esp,eax
```

- **POP**：弹出堆栈

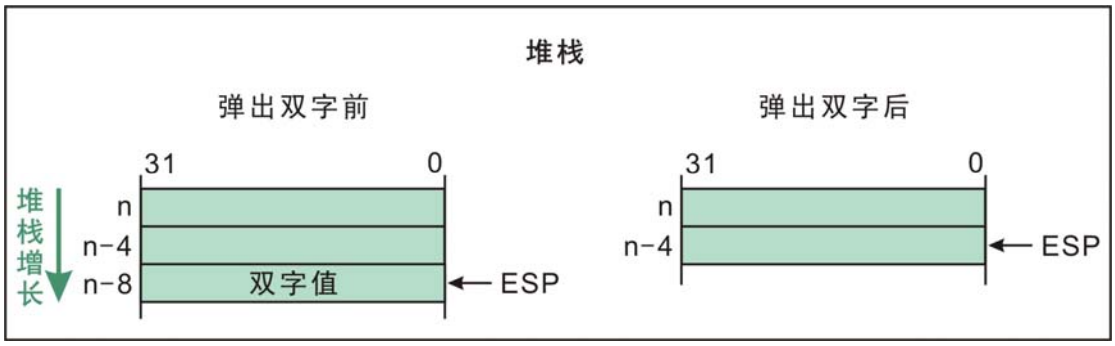


图 2-7：POP 指令图解

弹出堆栈与压入堆栈的操作相反。由图 2-7 得出：将要弹出的当前堆栈值赋给寄存器，然后堆栈加四个字节。POP eax 可以简化成：

```
mov eax,[esp];
add esp,4;
```

课后疑问：

POP ESP 是否也可以简化为指令方式？

说明：如果按上述的公式则写成

```
mov esp,[esp];
add esp,4;
```

可以看出，第一条汇编指令中得到的 esp 值被第二条指令修改，所以直接将 esp 带入是错误的。因为 pop eax 又可以等价于：

```
add esp,4
mov eax,[esp-4]
```

所以答案是：

```
add esp,4
mov esp,[esp-4]
```

课后总结：

操作和运算本身也是代码，并存在于内存中。

课后练习：

1．以下代码在纸上抄写 10 遍（目的抄会）

```
PUSH EAX
MOV DWORD PTR DS:[ESP-4],EAX
LEA ESP,DWORD PTR DS:[ESP-4]
POP ECX
LEA ESP,DWORD PTR DS:[ESP+4]
MOV ECX,DWORD PTR DS:[ESP-4]
```

2．PUSH ERX

POP ERX

ERX 做各种测试；单步执行语句，查看寄存器和内存的变化，且写出规律；

执行这些指令的时候 ESP 都是等于 0012FF34，这是什么原因呢？可以为其他的吗？

2.4 EFLAGS寄存器

本节主要内容：

1. 标志寄存器
2. 标志寄存器每个标志位的含义
3. 熟知标志寄存器的各种运算对于 CF,PF,AF,ZF,SF 这些标志位的影响

老唐语录：

下面我们开始讲标志寄存器 EFLAGS，如图 2-8：

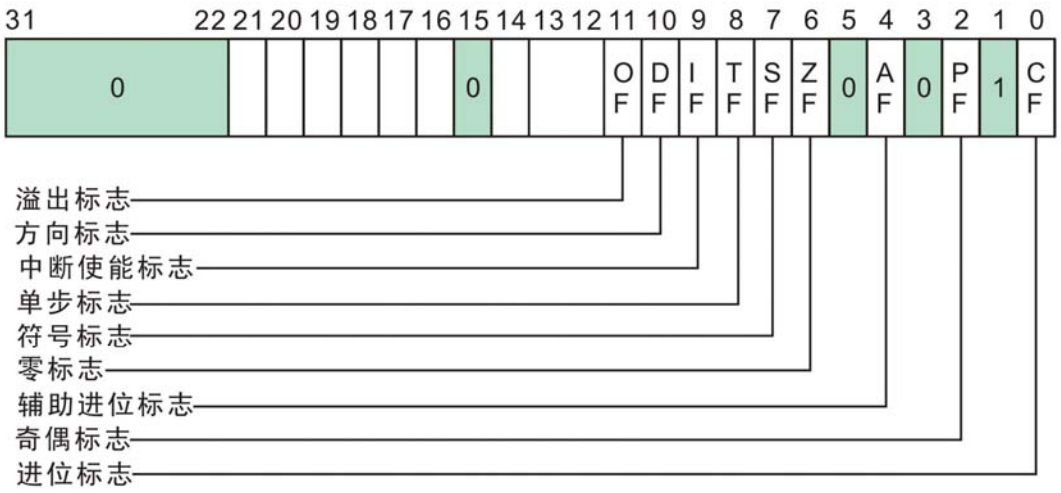


图 2-8：标志寄存器

两个数相加，如果最高位向高位进位，结果忽略这个位，那么这个位要放到 CF 里面。Carry Flag，进位标志。

减法运算，如果最高位向高位借位，则 CF 位为 1。

CF 表示加满了之后，向高位进位，但是计算机限制宽度，没法表示，所以把进的这个位暂时放在标志寄存器里面的一个位 CF，以后有待查证，减法同理。

CF 在 eflag 里面第 0 位。运算的宽度可以自定义：可以是 8 位，16 位也可以是 32

位的运算。

CF 如果本身是零，运算后无进位，则 CF 还是零，其实 CF 被改了，只是结果没有变而已。

第一位没有使用，永远是 1。

我们来看 **PF** 位，运算结果中 1 的个数是基数还是偶数，偶数置 1，基数置 0。PF 位只看结果（这里指二进制），parity flag，EFLAG 中的第二位。

AF 位：看低四位加减是否有进/借位，AF 置 1。没有进/借位也会改 AF（置 0）。

ZF 位只看结果，看结果是不是全是 0，全是 0，置 1，只要一位不是 0，则置 0，位置是第六位。

SF 位，最高位是多少，就置多少，是最高位的复制品，没有其他含义，如果是 8 位运算看第 7 位，16 位运算看第 15 位，32 位运算看第 31 位（从第 0 位开始）。

OF 位：最高位相同，看次高位是否向最高位进位，最高位不同，无溢出。

Intel 解释：

运算结果大于最大的正数，或小于最小的负数

我的理解（Intel 汇编语言程序设计）：

最高位的进位与次高位的进位异或。

我们之前一直认为两个操作数只能是在“寄存器和内存”或者“寄存器和寄存器”之间移动数据，不可以是在内存与内存之间。

其实也有特殊的指令允许出现两个内存：

mov [EDI],[ESI]可以简写为 movs

mov dword ptr[EDI],dword ptr[ESI] 简写为 movsd

mov word ptr[EDI],word ptr[ESI] 简写为 movsw

mov byte ptr[EDI],byte ptr[ESI] 简写为 movsb

练习：

mov edi,0x427c20

mov esi,0x427c48

movsd/movsw/movsb

观察哪些寄存器变化。

EFLAG 里面有一个 DF 位，movsb/w/d 每移动一次数据，都要对 ESI 和 EDI 进行加 1/2/4，或减 1/2/4，到底是加还是减，看 DF 位，当 DF 为 1 时，是减，当 DF 为 0 的时候是加。

所以我们使用 std（set DF）把 DF 置 1，cld（clear DF）把 DF 置 0，观察指令变化。

DF 影响的其他指令有：

stosb：stos byte ptr[edi],eax;(stosw，stosd)。

课后理解：

标志寄存器（EFLAGS）又称为程序状态和控制寄存器（Program Status and Control Register），主要用于记录当前的程序状态。

• 进位标志（CF）：在运算过程中出现进位或借位操作时标志位置 1

例：定义数值宽度为 32 位，则 0xFFFFffff+2 在运算过程中出现进位。

• 奇偶标志（PF）：运算结果值中 1 的个数为偶数置 1。这里的运算结果是站在计算机的角度看。

例：运算结果 0x4567 中 1 的个数是？

没有，那是站在你的角度；站在计算机的角度，它是 0100010101100111B。所以有 8 个 1，PF=1。

• 辅助进位标志（AF）：在算术运算中位 3 出现进位或借位置 1

• 零标志（ZF）：运算结果为零置 1

• 符号标志（SF）：符号位值（最高位）

• 单步标志（TF）：单步使能置 1，当该位置 1 时，可以对程序进行单步调试。

• 中断使能标志（IF）：响应可屏蔽中断置 1

• 方向标志（DF）：字符串指令（MOVS，CMPS，SCAS，LODS 和 STOS）处理字符串从高地址到低地址置 1（STD 和 CLD 指令设置和清除该标志位）。

说明：在汇编指令中，我们有些指令是对大块的内存操作，这个时候要选择从低地址开始还是从高地址开始，置 1 表示由高到低，置 0 则反之。

• 溢出标志（OF）：运算结果溢出置 1。区别于进位标志（在运算过程中）。在这里要引用 Intel 汇编语言程序设计中的一段话：CPU 是如何检测溢出的：在加法和减法运算完成

后，CPU 使用一种非常有趣的方法确定溢出标志的值：运算结果最高有效位向高位的进位值（CF）与到最高有效位的进位值异或，其结果放到溢出标志位中。

例如 8 位二进制数 10000000 和 11111110 相加，第 6 位向最高位有效位（第 7 位）无进位，但第 7 位向高位有进位值（CF=1）。如图 2-9：

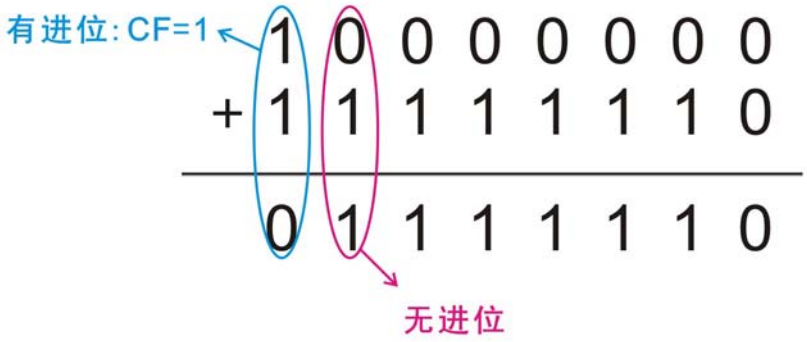


图 2-9：8 位二进制数加运算

由于 1 XOR 0 = 1，因此 of=1。

课后疑问：

AF 位是进位标志的一种吗？

回答：是。

加法或减法对 OF 位的影响有什么不同？

回答：一样的。

课后总结：

标志寄存器主要是记录当前的程序状态

课后练习：

举例改变标志寄存器的每个位

第 3 章 C语言

引言

在了解了汇编语言后，我们开始学习 C 语言，虽然两者看上去没什么关系，但是所有编译器都是先将高级语言（包括 C）转换成汇编，再由汇编转换成二进制。而且汇编语言语法简单，可以解释所有 C 语言中的语法。有人可能疑惑：既然汇编语言语法简单，为什么我们还要学习高级语言，直接使用汇编写程序不就可以了？当然可以，前提是你写的程序只有你一个人看，并且不打算运行在其他电脑上。这个前提诠释了汇编语言的诟病：很难阅读并且可移植性差。以往我们学习 C 语言总是和汇编扯不上半毛钱关系，这样理解的人大错特错，接下来我们通过汇编来揭开 C 语言神秘的面纱。

本章必须要掌握的知识点：

1．汇编与 C 的联系

2．函数的格式

3．条件执行语句

4．循环语句

5．表达式

6．数组

7．结构体

8．指针

本章常犯的错误：

1．内存的增长方向

2．一维数组和多位数组的区别

3．结构体定义和申明变量的使用、

4．指针和地址

3.1 C的汇编表示

本节主要内容：

1. 观察内存变化
2. 汇编与 C 的联系

老唐语录：

1>在我们之前的代码里总是出现 int main(),现在在 main 函数的上方写代码。

```
__declspec(naked) int abc(int a,int b)
{
  push ebx
  push esi
  push edi
  pop edi
  pop esi
  pop ebx
  ret
}
```

在 int main（）中添加

```
int r;
r=abc(2,3);
printf ("%d\n",r);
```

在 r=abc(2,3);处下断点然后按 F5，再按 F11 单步观察指令运行。

如果看懂每条指令的话，将每条指令改过的内存单元的编号记下来，写在纸上。

比如表 3-1：

表 3-1：内存地址书写格式

内存编号	内容
0x12ff2c	0x00000003

0x12ff28	0x00000002
0x12ff24	0x00401081
……	……

内存编号可以从小到大，也可以从大到小排列，每 4 个字节为一个单位，没有被改过的内存内容不用写，只记下被修改过的内容。

2>现在修改__declspec(naked) int abc(int a,int b)里面的内容如下：

```
push ebx
push esi
push edi
//在此处添加三条语句
mov eax,[esp+0x10]
mov ecx,[esp+0x14]
add eax,ecx
```

然后单步调试并观察变化。为什么最终打印的结果是 5，并将修改的内存画在纸上。

3>修改__declspec(naked) int abc(int a,int b)大括号里面的内容如下：

```
push ebp
mov ebp,esp
sub esp,4
push ebx
push esi
push edi
mov eax,[ebp+8]
add eax,[ebp+0xc]
mov [ebp-4],eax
pop edi
pop esi
pop ebx
mov eax,[ebp-4]
```

```
mov esp,ebp
pop ebp
ret
```

将修改的内存画在纸上。

修改 abc(2,3)为其他整数比如，abc(10,2)，观察结果变化。

问题：从 r=abc(2,3)开始到 printf 指令之前共有几条指令改变了内存单元？

回答：共九个。

4>修改__declspec(naked) int abc(int a,int b)大括号里面的内容如下：

```
push ebp
mov ebp,esp
sub esp,0x44
push ebx
push esi
push edi
mov eax,0xcccccccc
mov ecx,0x11
lea edi ,[esp+0xc]
rep stosd
mov eax,[ebp+8]
add eax,[ebp+0xc]
mov [ebp-4],eax
pop edi
pop esi
pop ebx
mov eax,[ebp-4]
mov esp,ebp
pop ebp
ret
```

将修改的内存画在纸上。

现在在 main()上面写下如下内容：

```
int abc(int a,int b,int c)
/*回车的地方都可以用空格代替，空格的地方可以用回车代替。也就是说，int a,int b,int c 写在一行和写在三行是一回事。*/
{
  int r;//每一句以分号结尾
  r=a+b+c;
  return r;
}
```

打比方，内存相当于算盘，但是算盘太多了，你根本就搞不清楚，给个地址，不好记，所以说，给每一个算盘取一个名字，比如：int a，int b，int c，int r，无论你取什么名字，都会分配给你四个字节的内存，然后在这个算盘上赋值，赋值很简单。我们称 a，b，c 为变量，也可以称为内存单元，赋值格式如下：

a	=	2	;
变量（内存单元）	等于	一个数，也可以是任意表达式	分号

不需要看 C，看汇编如何显示。

圆括号和大括号里面定义的变量都可以赋值和使用，也可以不使用。

课后理解：

当在对一个数进行运算前，需要先规定其宽度，在 C 语言中我们称之为申明。

申明语句格式如下：

数据类型 变量名；

其中数据类型即数据宽度，C 语言定义了以下几种数据类型：

int：32 位

short：16 位

char：8 位

注：C 语言除了十六进制数外，其他语句都区分大小写，这一点区别于汇编。

变量名也有要求：第一个字符必须是字母或下划线。可供使用的有小写字母、大写字母、

数字和下划线。

我们现在用实例来说明一个完整的 C 程序:

使用 VC6 打开并创建一个控制台的 HelloWorld 程序,VC6 自动为我们生成代码如下:

```
#include <stdio.h>
int main(void)/*程序入口*/
{
    int num;/*申明一个名为 num 的变量*/
    num = 1;/*赋值语句：num 值为 1*/
    printf("Hello World!\n");/*调用一个库函数*/
    return 0;/*返回*/
}
```

解析：

1〉首先我们要了解的是函数入口，即我们的程序从哪里开始运行：main 函数。注意一个新名词：函数。函数的英文名是 function，它的原意是功能、作用，所以“main 函数”又可以译为“主功能”。

2〉那我们第一条执行的指令是：int num;申明语句，该条语句给了我们三条信息：该变量名字是 num；宽度是 32 位；C 语言的每一条语句后都要以”;”结尾。

3〉第二条语句：num = 1;给该变量赋 1。既然 num 是变量，那么它的值可以再次被修改，这个留给大家测试。

4〉使用（专业术语是调用）一个 printf 函数，printf 译为打印，所以我们可以说调用一个打印功能。

5〉return 0;返回 0。该功能完成后，返回上一级。类似于我们去一个机关办事，办完事后，要从该机关出来才能去做下一件事。

课后疑问：

__declspec(naked)代表什么？

回答:代表裸函数，不要编译器帮我们构造函数框架，用户自己构造。

课后总结：

申明变量（比如 int a）就是给内存取名。

课后练习：

抄写汇编函数框架。

3.2 函数

本节主要内容：

1. C 语言函数的定义
2. 参数和变量的内存变化

老唐语录：

```
int abc(int e,int f,int c,int a,int b)
/*像前几讲一样 int 必须写，不需要理解他的意思，后面取一个函数的名字，这个名字只要是由字母数字或下划线组成并不以数字打头都可以，长度不受限制，只要不和别人冲突就可以。后面是空格，然后是左圆括号，右圆括号，中间是 int，然后空格（几个空格都可以，一般位 1 个），取个名字，作为标识符，可以取一排变量，用逗号隔开，最后没有分号。*/
//然后大括号
{
  int a;//int 后面是变量名，可以任意取，只要不和上面冲突就可以
  int x;//可以取一排，用分号隔开
  int y;
  a=b+y;
  x=1;
  y=2;
  x=a+y+e+f+c+a+b;
  return x+y+a;
}
```

如果前面是个 int，后面是个名称、左圆括号和右圆括号的话，我们称之为函数。就是因为这个导致产生一个 CALL。然后在调用的时候可以发现，他会往内存里面写参数，比如

你在括号里写一个 3，他会 push 3。在括号中定义多少个，他就会在往内存单元写多少个数，和他对应的是，从右边往左边开始，先给右边的赋值，所以在高地址，因为 push 指令会令会导致 esp-4，所以最右边的是地址是最高的内存单元。比如：

调用 abc(1,2,3)的汇编程序如下：

```
push        3
push        2
push        1
call        abc
```

每个单元 4 个字节。在圆括号里面定义的是 int 变量，每多定义一个，ebp 多减去 4 个字节，比如定义一个 int 变量，sub esp，0x44，定义两个 int 变量，sub esp，0x48。每一个名字（变量）都对应一个 4 字节的内存单元。比如：

int abc(int e,int f,int c,int a,int b)汇编程序如下：

```
push        ebp
mov         ebp,esp
sub         esp,40h
push        ebx
push        esi
push        edi
lea         edi,[ebp-40h]
mov         ecx,10h
mov         eax,0CCCCCCCCh
rep         stosd
```

注意看内存，里面有很多 0xCC，也就是说给一片单元，每四个字节取一个名字。无论是圆括号的里面定义的名字，还是大括号里面定义的名字，我们都把他叫做变量（variable）。给变量赋值是最基础的。无论做什么之前，都要对变量赋值：变量名=表达式；表达式可以是一个数。

变量名	等于号	表达式	分号
a	=	3	;

C 语言中的回车符只是为了便于阅读。将所有语句并做一行，代码仍然可以运行，只是

不便于阅读。

练习：

将 C 语言对应的汇编和内存变化写在纸上。

将 r=abc(1,2,3);修改为：

```
__asm
{
        sub esp,0xc
        mov dword ptr ds:[esp+0],1
        mov dword ptr ds:[esp+4],2
        mov dword ptr ds:[esp+8],3
        call abc
        add esp,0xc
        mov r,eax
}
```

再画出内存变化并运行。

课后理解：

下面简单讲一下函数的架构：

返回值 函数名（传入参数 1，传入参数 2，……）

```
{
    函数主体
}
```

比如：

```
int Add(int a,int b)
{
   int c;
   c = a+b;
   return c;
```

```
}
```

说明：这个函数实现了两数相加的功能，返回的是两个数的和。我们可以这样调用它：

```
int result;
result = Add(34,56);
```

这样就实现了 34+56 的和，并把这个结果传给了 result。

课后疑问：

函数的参数可以定义任意多个吗？

回答：理论上可以。一般情况下不要多余四个，这样便于阅读。

课后总结：

函数格式：前面是个 int，后面是个名称、左圆括号和右圆括号。

课后练习：

1. 更改参数查看内存变化
2. 更改变量查看内存变化

3.3 内存结构

本节主要内容：

1. 变量和参数在内存中的存放格式
2. 函数中嵌套多个函数

老唐语录：

```
int abc(int p1,p2,p3)
{
  int a;
  int b;
  int c;
  a=p1+p2+p3;
  b=p1-p2-p3;
  c=p1-p2+p3;
  return a+b+c;
}
```

写的多了，自然就清楚了，在汇编里写一个 call，或者在 C 语言里，main 函数中写一个函数名，左圆括号，参数，右圆括号，然后 call 指令地址加上本指令的长度就是下一条指令的首地址，放在内存里面去，如图 3-1：

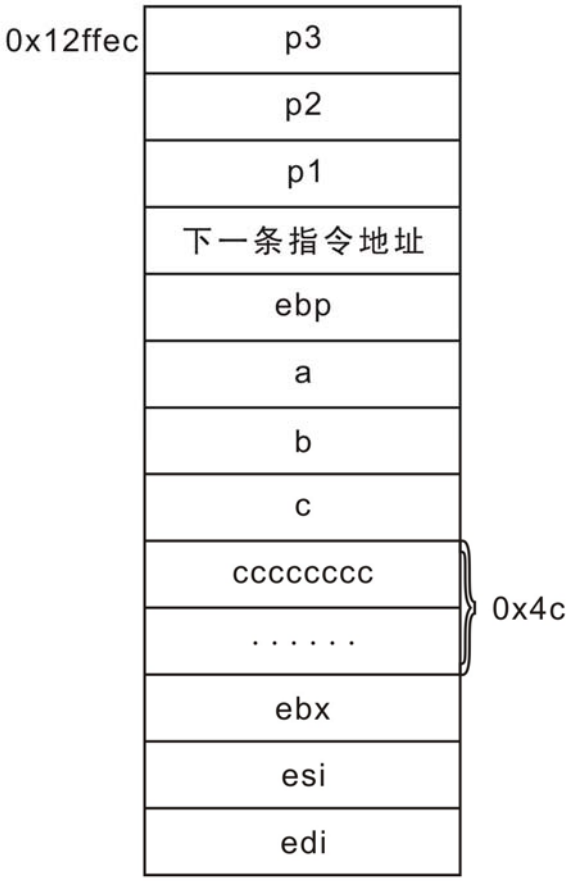


图 3-1：调用函数内存结构

练习：

```
int abc19(int a,int b,int c,int d,int e)
{
  return a+b+c+d+e;
}
……
int abc5(int a,int b,int c,int d,int e)
{
  int r;
  int v;
  v = abc6(a+5,b+5,c+5,d+5,e+5);
  r = a+b+c+d+e+v;
```

```
  return r;
}
int abc4(int a,int b,int c,int d,int e)
{
  int r;
  int v;
  v = abc5(a+5,b+5,c+5,d+5,e+5);
  r = a+b+c+d+e+v;
  return r;
}
int abc3(int a,int b,int c,int d,int e)
{
  int r;
  int v;
  v = abc4(a+5,b+5,c+5,d+5,e+5);
  r = a+b+c+d+e+v;
  return r;
}
int abc2(int a,int b,int c,int d,int e)
{
  int r;
  int v;
  v = abc3(a+5,b+5,c+5,d+5,e+5);
  r = a+b+c+d+e+v;
  return r;
}
int abc1(int a,int b,int c,int d,int e)
{
  int r;
```

```
  int v;
  v = abc2(a+5,b+5,c+5,d+5,e+5);
  r = a+b+c+d+e+v;
  return r;
}
int abc(int a,int b,int c,int d,int e)
{
  int r;
  int v;
  v = abc1(a+5,b+5,c+5,d+5,e+5);
  r = a+b+c+d+e+v;
  return r;
}
```

main 函数中的调用语句：

```
int r= abc(1,2,3,4,5);
printf("%d\n",r);
```

写出内存变化。

课后理解：

函数中嵌套有多个函数时，它的执行顺序是依据函数之间调用的关系由内到外的执行，其中用于运算的寄存器的顺序是 EAX，ECX，EDX，注意：在中间的函数的执行中，整个子程序从 PUSH 开始到最后的 RETURN 语句结束，再将返回值写入 EAX 传递给下一函数使用。函数的执行都是有始有终的。

课后疑问：

本节没有疑问。

课后总结：

函数调用都会引发 CALL 指令的执行。

课后练习：

练习嵌套函数并抄写反汇编语句。

3.4 条件执行

本节主要内容：

1. Jcc、SETcc、CMOVcc 指令
2. if 语句

老唐语录：

我们学习汇编到现在，其实就是学习了 mov 指令，在“寄存器和寄存器”或者“寄存器和内存”之间移动数据，之前学过很多指令影响标志位：ADD，ADC，AND，OR，XOR……

MOV eax,ebx 是无条件将 ebx 移动到 eax 里面去,现在学习有条件移动指令：CMOVcc，后面两个 cc 表示可以替换。比如 CMOVC eax，edx 表示当 CF 位为 1 时，数据从 edx 到 eax。如果 CF 为 0 时，这条指令什么都不做，也就是说在移动数据时，首先要判断标志位；而 CMOVNC 是 CF 为 0 时，数据从 edx 到 eax。如果 CF 位 1 时，不移动。除了 c、nc 之外还有 p、np，z、nz。“CMOVcc eip,”等于“Jcc”，Jcc 也有 16 种写法。所有条件见表 3-2：

表 3-2：Jcc 指令

指令	条件	描述
有符号条件跳转		
JA/JNBE	(CF+ZF) = 0	高于/不低于等于
JAE/JNB	CF = 0	高于或等于/不低于
JB/JNAE	CF = 1	低于/不高于等于
JBE/JNA	(CF+ZF) = 1	低于或等于/不高于
JC	CF = 1	进位
JE/JZ	ZF = 1	等于/零
JNC	CF = 0	不进位
JNE/JNZ	ZF = 0	不等于/非零

JNP/JPO	PF = 0	奇数
JP/JPE	PF = 1	偶数
JCXZ	CX = 0	寄存器 CX 为零
JECXZ	ECX = 0	寄存器 ECX 为零
无符号条件跳转		
JG/JNLE	((SF^OF)+ZF) = 0	大于/不小于等于
JGE/JNL	(SF^OF) = 0	大于或等于/不小于
JL/JNGE	(SF^OF) = 1	小于/不大于等于
JLE/JNG	((SF^OF)+ZF) = 1	小于或等于/不大于
JNO	OF = 0	不溢出
JNS	SF = 0	非负
JO	OF = 1	溢出
JS	SF = 1	负数

SETcc AL 是将条件赋给 AL，比如 SETC AL，将 CF 位赋给 AL，当 CF=1 时，AL=1，当 CF=0 时，AL=0。

练习：

将 CMOVcc 中的每一个条件都试一下。

注：cc 代表十六种条件，十六种写法。

练习：

画出内存变化：

```
int fun10(int a,int b,int c,int d,int e)
{
  int sum;
  int stub;
  if(a>=0x65)
  return 0;
  stub = fun10(a+10,b+10,c+10,d+10,e+10);
```

```
  sum = stub+a+b+c+d+e+(a+5+b+5+c+5+d+5+e+5);
  return sum;
}
```

调用语句：result = fun10(1,2,3,4,5);

课后理解：

cmovc edx,eax ： 如果 cf=1， 等同于 mov edx,eax 如果 cf=0,不处理

cmovnc edx,eax ： 如果 cf=0， 等同于 mov edx,eax 如果 cf=1,不处理

setcc oprd：oprd 可以是 8bit 寄存器、内存，不可为立即数

setz al:如果 zf=1,置 al 为 1， 如果 zf=0,不处理

setnz al:如果 zf=0,置 al 为 1， 如果 zf=1,不处理

课后疑问：

为什么 jcc [eax]不执行？

回答：vc 编译器不识别 jcc [寄存器]指令，但 test 编译器识别。

课后总结：

条件执行指令是根据 EFLAG 寄存器中相应的标志位为判断条件。

课后练习：

写程序练习条件执行语句。

3.5 移位指令

本节主要内容：

1．算数移位与逻辑移位指令

2．循环移位指令

老唐语录：

计算机的运算指令除了 ADD、SUB、ADC、SBB、OR、XOR、AND 之外，总会有其他指令，比如说操作数的取反指令 NOT。还有 CMP 指令，其实和 SUB（减指令）是一样的，只是不改变目标操作数，比如 SUB eax，edx 是 eax 的值减去 edx，结果放到 eax 里面去，影响标志位，而 cmp 只影响标志位，运算结果忽略，不影响 eax。

我们今天来学习移位指令。

逻辑移位：

SHR AL，2 向右面移两个位，如果 AL 值为 00101010，则向右移动两位后变成 00001010，简单的理解是，先移一位变成 00010101，再移一位变成 00001010，可移动的最大值为 0x1F，如果不论输入的值为多少，结果都要与 0x1F。也就是说只有低五位有效，如图 3-2：

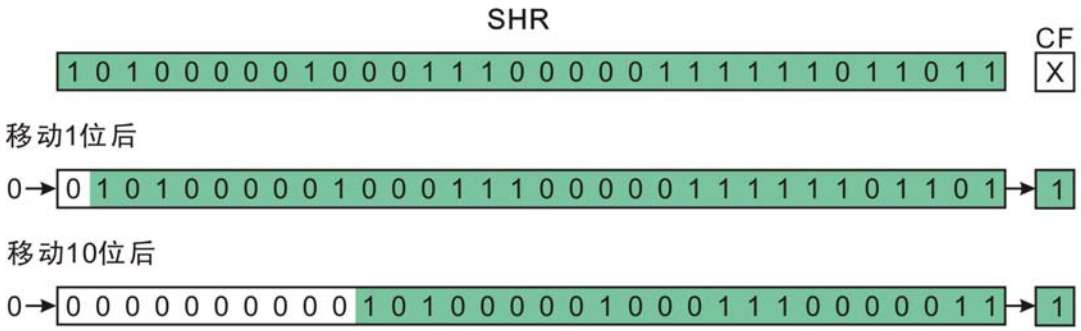


图 3-2：SHR

向左移：SHL ，只要会移动一位，移动 N 位就明白了，移动后的结果放在目标寄存器里面，当然该操作也会改变标志寄存器中的 CF，将移出来的位赋给 CF；右移也是一样，凡

是移出来的那个位都赋给 CF，如图 3-3：

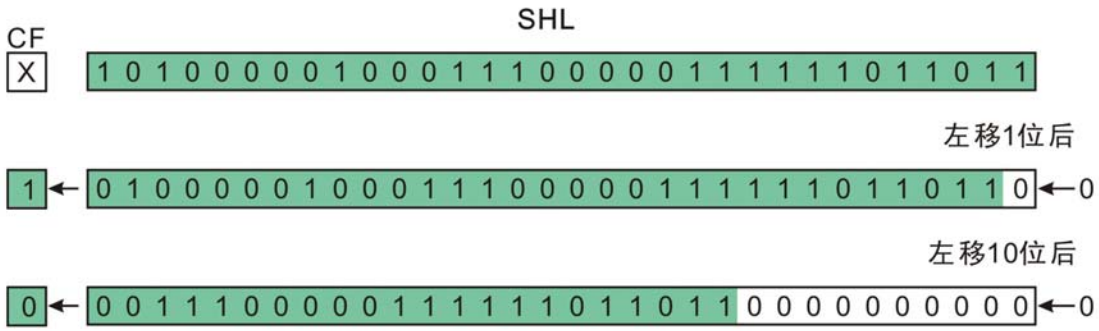


图 3-3：SHL

疑问：如果移动 N 位，哪一位赋给 CF？

移动 N 位，相当于 N 次移动一位，也就是说，每移动一位，就将移出来的那一位赋给 CF，移动了 N 位，就赋给 CF 位 N 次。所以结果很明显，就是最后移动的那一位。

当然移位指令照样影响 SF 位：将移动后剩下的最高位赋给 SF。还要看结果是否全为零，如果全为零，ZF 置 1，否则置 0。也影响 PF 位（1 的奇偶性）。不影响 OF 位。

移位指令的源操作数只能是 8 位的寄存器 CL 或立即数（ECX 中的 C 为 counter，计数）；目标操作数可以是字节/双字节/四字节，内存或寄存器。

算术移位：

算术移位和逻辑移位的区别：算术左移和逻辑左移没有区别；在右移时，高位移至低位，如果最高位补 0，是逻辑移位；最高位保持不变（符号），是算术移位，如图 3-4：

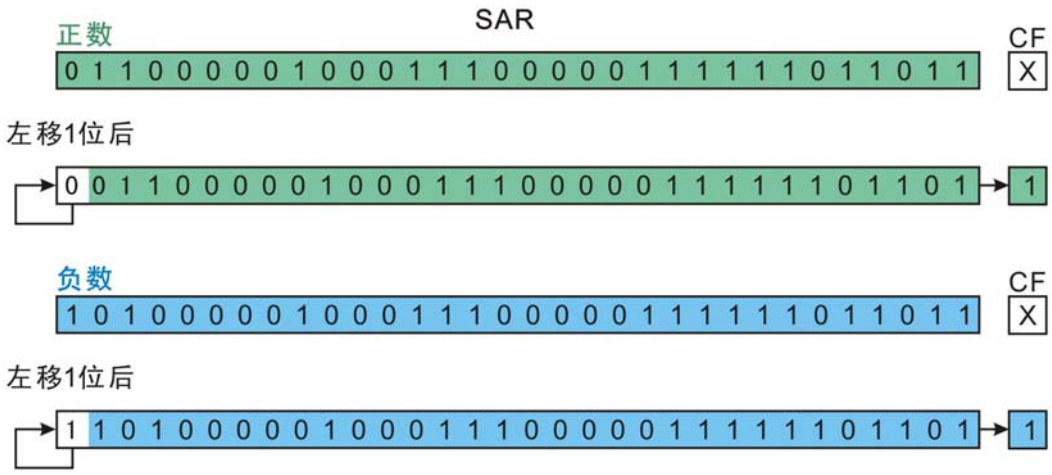


图 3-4：SAR

SHR：逻辑右移，SH 是 shift，R 代表 right，L 代表 left。

SAR：算术右移，A 是算术 arithmetic。

除了上述四条之外，还有循环移位：

循环右移：循环右移一位，31 位赋给 30 位，30 位赋给 29 位……最低位同时赋给最高位和 CF 位，导致最高位和 CF 位相同。当然也影响 SF，ZF，PF。如果移动五位看不懂，只需要移动一位，然后移动 5 次即可。

RCL 和 RCR：带进位的循环移位（C 代表 CF 位）：循环左移一位，将最高位给 CF，CF 值赋给最低位，最低位赋给次低位。

课后理解：

ROL/ROR 如图 3-5：

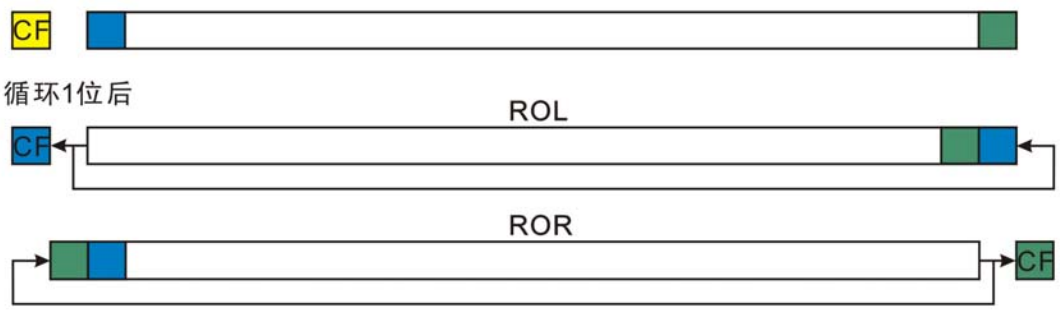


图 3-5：ROL/ROR

RCL/RCR 如图 3-6：

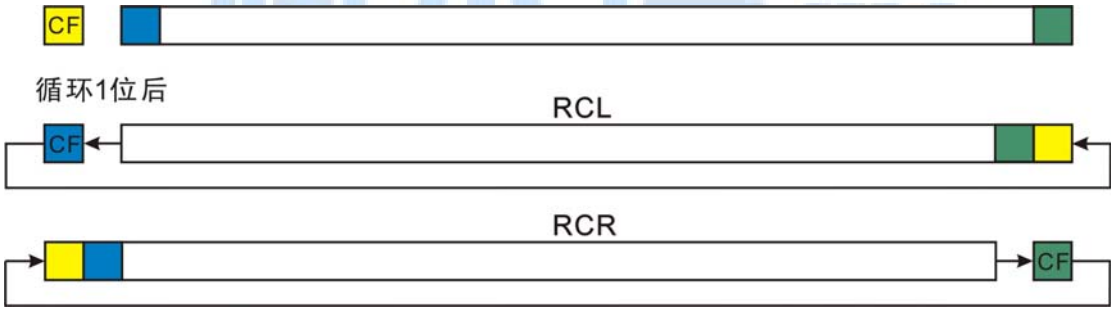


图 3-6：RCL/RCR

课后疑问：

C 语言中移位指令的书写格式？

左移<<，右移>>。

左移只会生成 SHL 指令，因为 SHL 和 SAL 等效

无符号右移（>>前的数）会生成 SHR，有符号数右移会生成 SAR

课后总结：

算术移位涉及符号位的运算，而逻辑移位不涉及。

课后练习：

画出下列指令执行时内存数据的变化情况：

```
int CumulativeHundred1(int a,int b,int c,int d,int e, int f,int g,int h,int i,int j);
int CumulativeHundred7(int a,int b,int c,int d,int e, int f,int g,int h,int i,int j)
{
 int sum;
 int stub;
 stub = (a>=90) ? 0:CumulativeHundred1 (a+20,b+20, c+20,d+20, e+20, f+20, g+20,h+20,i+20,j+20);
 sum = stub +g+h+i+j + ((a<<1)+10+(b<<1)+10+(c<<1) +10+(d<<1) +10+(e<<1) +10+(f<<1)+10+g+10+h+10+i+10+j+10);
 return sum;
}
int CumulativeHundred6(int a,int b,int c,int d,int e,int f,int g,int h,int i,int j)
{
 int sum;
 int stub;
 stub = (a>=90) ? 0:CumulativeHundred7(a+20,b+20, c+20,d+20, e+20,
```

```
f+20, g+20,h+20,i+20,j+20);
     sum = stub +f+g+h+i+j + ((a<<1)+10+(b<<1)+10+ (c<<1)+10+ (d<<1)
+10 +(e<<1)+10+f+10+g+10+h+10+i+10+j+10);
     return sum;
    }
    int CumulativeHundred5(int a,int b,int c,int d,int e, int f,int
g,int h,int i,int j)
    {
     int sum;
     int stub;
     stub = (a>=90) ? 0:CumulativeHundred6(a+20,b+20, c+20, d+20, e+20,
f+20, g+20,h+20,i+20,j+20);
     sum = stub +e+f+g+h+i+j + ((a<<1)+10+(b<<1) +10+(c<<1)+ 10+(d<<1)
+10 +e+10+f+10+g+10+h+10+i+10+j+10);
     return sum;
    }
    int CumulativeHundred4(int a,int b,int c,int d,int e, int f,int
g,int h,int i,int j)
    {
     int sum;
     int stub;
     stub = (a>=90) ? 0:CumulativeHundred5(a+20,b+20,c+20, d+20,e+20,
f+20, g+20,h+20,i+20,j+20);
     sum = stub +d+e+f+g+h+i+j + ((a<<1)+10+(b<<1)+10+ (c<<1)+10+d+
10+ e+ 10+f+10+g+10+h+10+i+10+j+10);
     return sum;
    }
    int CumulativeHundred3(int a,int b,int c,int d,int e, int f,int
g,int h,int i,int j)
```

```
    {
     int sum;
     int stub;
     stub = (a>=90) ? 0: CumulativeHundred4 (a+20,b+20, c+20,d+20, e+20,
f+20, g+20,h+20,i+20,j+20);
     sum = stub +c+d+e+f+g+h+i+j + ((a<<1) +10+ (b<<1) +10+c +10+d +10+e
+ 10 +f +10+g+10+h+10+i+10+j+10);
     return sum;
    }
    int CumulativeHundred2(int a,int b,int c,int d,int e, int f, int
g,int h,int i,int j)
    {
     int sum;
     int stub;
     stub = (a>=90) ? 0: CumulativeHundred3 (a+20, b+20, c+20, d+20,
e+20, f+20,g+20,h+20,i+20,j+20);
     sum = stub+b+c+d+e+f+g+h+i+j + ((a<<1) +10 +b +10 +c +10 +d +10
+e +10+f+10+g+10+h+10+i+10+j+10);
     return sum;
    }
    int CumulativeHundred1(int a,int b,int c,int d,int e, int f,int
g,int h,int i,int j)
    {
     int sum;
     int stub;
     stub = (a>=90) ? 0:CumulativeHundred2(a+20,b+20, c+20, d+20, e+20,
f+20, g+20,h+20,i+20,j+20);
     sum = stub +a+b+c+d+e+f+g+h+i+j + (a+10+b+10 +c+10+d +10+e+10
+f+10 +g+10+h+10+i+10+j+10);
```

```
 return sum;
}
```

3.6 表达式

本节主要内容：

1. 表达式的格式
2. 表达式的使用

老唐语录：

有些指令只操作两个寄存器，有些指令会操作几十甚至上百个寄存器。

cmp 和 SUB 指令功能是一样的，只是 cmp 不改变目标操作数的值，同样影响标志位。AND 的功能是按位与。TEST 与 AND 的功能相同，只是不改变目标操作数的值，也同样影响标志位。有些指令不能写立即数，有些指令可以全是内存，只要在书写时可编译通过，表示语法正确。

以后我们不需要写汇编，只需要看懂别人写的汇编代码即可。前面提到，定义一个 C 语言函数格式：int + 函数名 +（+定义的变量+……），可以称 C 语言是汇编语言的高级书写形式。

表达式的定义：**凡是由变量、常量和算术符号链接起来的都可以称为表达式。**

在 C 语言中，XOR 表示为“^”，OR 表示为“|”，AND 表示为“&”。数学里面的表达式有圆括号，中括号和大括号，在 C 语言中只有圆括号。

比如：

```
int _abcd(int ab,int cd,int name)
{
  int ba;//变量
  int dc;
  ba = ab+cd;//表达式
  return ab;//可以是一个值，也可以写表达式
}
```

练习：

用各种算术符号写表达式，越复杂越好，编译成功后，观察汇编。

如果看不到逻辑移位，只能看到算术移位，只需将 int 替换成 unsigned int。

注：int 是有符号整型，unsigned int 是无符号整型。

在表达式中，任意一个成员又可以表示成表达式，所以可以像嵌套一样无限膨胀。

练习：

```
int abc(int a,int b,int c)
{
  int var;
  int h;
  int j;
  var = 0;
  var = (var,0x5,var+0x5);//var = var+0x5;
  return 0;
}
```

“=”左边只能是个变量，不可以是常量或表达式。

逗号和等号也是算术运算符。

单目/双目/三目运算符：b=b+c 等价于 b+=c。

通过加圆括号和不加圆括号的运算来判断运算符的优先级。

练习：

乘法指令分为有符号乘法和无符号乘法：

计算机做有符号乘法（IMUL）时，首先判断最高位是否为 1，为 1（负数）时，将 1 提出来，变成 0 减去提出 1 后的数（此时为正数），乘法运算完成后，再用 0 减去结果。

当做乘法时，目标操作数默认为 EAX 或 EDX：通常乘法会使宽度扩大一倍，8 位的乘法需要两个字节保存，32 位的乘法需要 64 位保存。 1 个字节相乘时，结果放在 AX 里面，两个字节相乘时，结果放在 EAX 里面，4 个字节乘法时，结果放在 EAX 和 EDX 里面，其中 EAX 为高位。比如 MUL ecx；将 ecx 乘以 eax 并将结果放在 EAX 和 EDX 中。

除法里面，结果放在 EAX 里面，余数放在 EDX 里面。

V = ab%5+ba/7+ba*cd；// “%”：求余，“/”：除法，“*”：乘法

如果目标操作数不设定为 EAX，比如 IMUL ECX，EAX，则结果会溢出，没有意义。

IMUL 后面也可以含有 3 个操作数，比如 IMUL ECX，EAX，0x3；则第三个操作数必须是立即数。

比如：-5*-6 = -1*(0-(-5)) * -1*(0-(-6))。

练习：

写出下列 C 程序对应的汇编代码

```
int abc(int a,int b,int c)
{
  int v;
  int r;
  v = a+b+c;
  r = a|b&c;
  return v+r;
}
int ab(int a,int b,int c,int d)
{
  int v;
  v = abc(a+b,b+c,c+a)+3;
  return v+d+5;
}
```

调用函数的时候，函数的参数也可以是表达式。

观察函数的规律：

第一条指令都是：push ebp，然后是

```
push        ebp
mov         ebp,esp
sub         esp,40h
push        ebx
```

```
push       esi
push       edi
lea        edi,[ebp-40h]//没有定义变量为0x40
mov        ecx,10h
mov        eax,0CCCCCCCCh
rep stos    dword ptr [edi]
```

如果函数内部定义了变量，则[ebp-4]为第一个变量，[ebp-8]为第二个变量，[ebp+4]为函数返回后执行下一条指令的地址。

函数末尾结构如下：

```
pop        edi
pop        esi
pop        ebx
mov        esp,ebp
pop        ebp
ret
```

练习：

写一个函数，实现只有表达式由编译器生成汇编，其他部分由自己构造（汇编实现）。

```
v=a+b+c;r|=a&b|c;
```

说明：在写裸函数前需要在函数前加__declspec(naked)：

```
__declspec(naked) abc(int a,int b,int c)
{
  int v;int r;
  __asm
  {
    push ebp;
    mov ebp,esp
    sub esp,__LOCAL_SIZE(编译器的宏)//可以写成0x48
  }
```

```
v=a+b+c;
r|=a&b|c;
__asm
{
mov eax,r
mov esp,ebp
pop ebp
ret
}
}
```

高级语言的唯一方便之处就是表达式可以按数学思维书写，其他部分都只是固定的格式。

课后理解：

表达式可含有：

1.赋值符号： =

2.运算符号： +、-、*、/

3.不等号符号：==、>=、<=、!=、

课后疑问：

数字 3 是表达式吗？

回答：是的，属于常量表达式。

课后总结：

凡是由变量、常量和算术符号链接起来的都可以称为表达式。

课后练习：

找出下列对错

```
int #33;
int a;
int 2a;
int i;
int 3_Alafun()
{
  int v;
  int a;
  int b>a;
  v += i++++;
  v += ++++i;
  return 0;
}
```

3.7 if语句

本节主要内容：

1. if 语句的书写格式
2. if else 语句

老唐语录：

在函数调用时，用到的内存称为栈。无论在调用函数时，还是在函数内部，push 指令都要对应相应的 pop 指令，或者 esp 值最终都要被恢复到调用函数之前的值，我们称之为栈平衡。

函数框架：

```
int abc(int ab,int cd,int ef)//ebp+0:原 ebp 值；ebp+8：ab；ebp+0xc：cd；ebp+0x10：ef
{
  int v;//ebp-4
  int h;//ebp-8
  int r;
  ……
  return 0;//把表达式的值赋给 eax
}
```

C 语言中最常用的语句是 if 语句。

格式：

```
if ()//圆括号里面可以写任意表达式
{
//可以写任意多个表达式
}
```

既然 if 语句中间可以放任意语句，if 语句也属于语句，那么 if 语句中间也可以放 if 语句：

```
if ()
{
  if ()
  {
  }
}
```

if 语句的第二种格式：

```
if ()
{
}
else
{
  //也可以是任意语句
}
```

if 语句的第三种格式：

```
if ()
{
  //可以是任意语句
}
else if()
{
  //可以是任意语句
}
else if()
{
  //可以是任意语句
}
```

```
else if()
{
  //可以是任意语句
}
……//中间可以有任意多个 else if 语句
else //可以省略
{
  //可以是任意语句
}
```

其中圆括号里面都可以放任意表达式，且每个表达式之间没有任何关系。将 else if 格式省略掉，变成第二种格式；将 else if 格式和 else 格式都省略掉，变成第一种格式。

练习：

练习 if 语句的三种格式，并在圆括号和大括号里面写任意表达式，并用汇编查看。

说明：逗号表达式也属于表达式，比如 if(表达式 1，表达式 2，……)

Jcc lab10 指令等价于 CMOVcc eip，lab10//lab10 是标签

其实 C 语言中也含有功能等价于 jmp 的指令 goto：

goto lab10；等价于 jmp offset lab10

对于 if 语句，当圆括号里面的表达式值为非 0 时，执行大括号里面的程序。

课后理解：

```
if(表达式)
{
表达式;
}else if(表达式)
{
表达式;
}else
```

```
{
表达式;
}
```

其中表达式可以为空，数值，或判断语句。

课后疑问：

可以添加两个 else 语句，比如：

```
if ()
else
{
}
else
{
}
```

回答：不可以。因为 else 已经代表结束（包含了除上述情况外的所有情况），不需要再次结束。

课后总结：

if 语句又称选择语句，用于在可选择的几个动作之间进行选择。

课后练习：

写出下列函数中 FuncMove 函数每次执行完后变化内存的值

```
int FuncMove(int n,int nX,int nY)
{
 int nResult;
 int nTemp1;
 int nTemp2;
 int nTemp3;
```

```
 nResult = n;
 nTemp1 = 0;
 nTemp2 = 0;
 nTemp3 = 0;
 nTemp1 |= nY;
 nTemp2 = 0x3E2D2D00;
 nTemp2 |= nX;
 return nResult ;
}
int FuncHanoi(int n,int One,int Two,int Three)
{
 int nResult;
 nResult = 1;
 if (n==1)
 {
  FuncMove(n,One,Three);
 }
 else
 {
  FuncHanoi(n-1,One,Three,Two);
  FuncMove(n,One,Three);
  FuncHanoi(n-1,Two,One,Three);
 }
 return nResult;
}
```

3.8 循环语句

本节主要内容：

1. while 循环
2. for 循环

老唐语录：

在申明一个变量的时候没必要马上赋值。可以在使用的时候再进行初始化。

练习：

```
int v;
int i;
i=1;
v=0;
v+=i++++;//错误
(v=v+1)++;
(v=v+1)=(v=v+1)+1;
v+=+++++i;//编译器把落单的加号看作正号
v+=+++++++++i;//等价于 v=v+(++(++(++(++i))));
++i=i+1;//看汇编代码可知++优先级高于等号
```

总结：

```
5+i++//先进行 5+i，再运算 i+1
4+++i//先运算 i+1，再进行 4+i
```

程序往回转，可以通过 goto 语句实现。比如 goto lab（标签）；因为启用标号比较麻烦，所以 C 语言提供了新的语句 while。

格式：

```
while()/*圆括号里面可以是任意表达式，表达式的值成立，执行大括号里面的语句，
```

执行完后，再次判断圆括号里面的表达式值是否成立，若成立，继续执行大括号里面的语句……直至判断圆括号里面的表达式值不成立，跳出循环*/

```
{
  //可以是任意语句
}
```

练习：

练习 while 语句，并调试观察对应的汇编语句

当 while 语句中圆括号中值为 1 时，会永远执行，while 语句里面也可以嵌套 while 语句。

while 语句有两种形式，另外一种是 do while 语句：先执行，再判断，如果条件成立，继续执行：

```
do
{
}while();//圆括号里面不能为空
```

for 语句也是循环语句的一种，for 圆括号里面有三个表达式：

```
for (表达式 1;表达式 2;表达式 3)/*表达式可以不写，但是分号不能省略，表达式不写，则表示为 1 */
{
  //可以是任意语句
}
```

注：for 后面圆括号里面的表达式通常会改变某些变量的值，否则不执行或永远执行。表达式 1 只在开始时执行一次，表达式 2 和表达式 3 每次都执行。执行次序是：

1>表达式 1；

2>表达式 2；//表达式值成立

3>大括号里面内容；

4>表达式 3；

5>表达式 2；//表达式值成立

6>大括号里面内容；

7>表达式 3；

……

N>表达式 2；//表达式值不成立

N+1>跳出循环

练习：

构造 for 循环语句，观察其生成的汇编代码，并单步观察执行顺序

break 也是一条语句，通常放在 if，while 和 for 语句中间，作为中断使用，该功能可以由 goto 语句代替（需要设定标签）。如若 break 语句处于嵌套循环语句中，则只会跳出当前循环。

课后理解：

For 语句可以用 if 和 goto 语句实现，实现步骤如下：

```
1>for (i=begin,sum=0; end-i+1;i=i+1)
   { ... ... }
2>i=begin,sum=0;
   for ( ; end-i+1;i=i+1)
   { ... ... }
3>i=begin,sum=0;
   for ( ; end-i+1; )
   {
      ... ...
     i=i+1;
   }
4>i=begin,sum=0;
   lab:i=i+1;
   if ( end-i+1 )
   {
      ... ...
```

```
}else
 {
... ...
   goto lab;
 }
```

以上步骤可以说明，c 语言中所有句型都可以用 if 和 goto 实现。

for 语句格式来源：

在 for 循环中加入 continue 语句，当 continue 执行后，程序跳转到下一循环 for(...; ...;...)的第二个";"处 （从括号中的第三条语句开始执行）。

课后疑问：

在嵌套循环如：

```
for(;;)
{
  for(;;)
  {
    break;
  }
  lab1:……
}
lab2:……
```

如果执行 break 语句，则跳转的位置是 lab1 还是 lab2？

回答：lab2，break 语句只跳出当前循环。

课后总结：

c 语言中所有句型都可以用 if 和 goto 实现。

课后练习：

画出下列指令执行时内存数据的变化情况

```
int fun ( int a , int b)
{
  int v;
  int i;
  int j;
  i=a;
  v=0;
  while (i-b-1)
  {
    v+=i++;
  }
  return v;
}
int main(int argc, char* argv[])
{
  int result;
  result = fun(1,100);  // 从这里开始画栈
  printf("%d \n",result);
  return 0;
}
```

3.9 变量

本节主要内容：

1. 变量类型
2. 类型扩展

老唐语录：

C 语言除了赋值语句外，只有 if，while 和 for 语句。

```
int abc(int a,int b,int c)//ebp+8:a
{
  int v;//每多定义一个变量，ebp 要多减去 4 个字节
  int r;
  ......
}
```

其中 int 表示 4 个字节。int 可以改成 char，short，long。

char 表示变量的宽度只有 1 个字节

short 表示变量的宽度只有 2 个字节

long 表示变量的宽度只有 4 个字节（历史遗留问题）

练习：

自己编写程序，申明多个变量，将变量类型改为 char，short，int 和 long，观察汇编语句变化并调试。

观察汇编里面出现了 movsx 和 movzx 指令。

movsx 是将源操作数符号扩展为目标操作数的宽度并赋给目标操作数，movzx 是将源操作数 0 扩展为目标操作数的宽度并赋给目标操作数，如图 3-7。

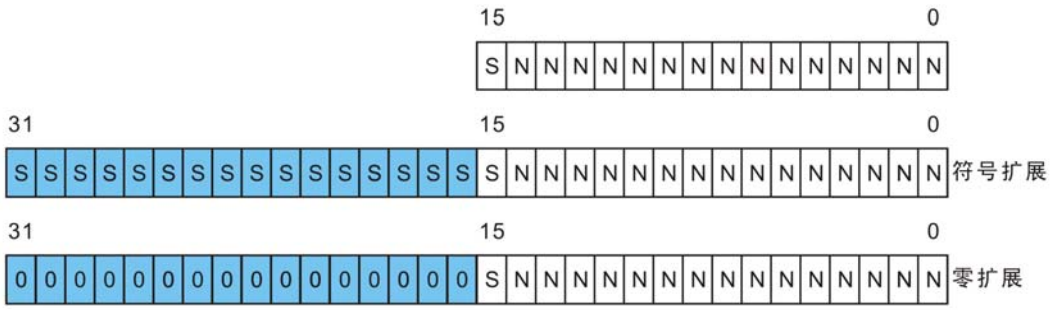


图 3-7：数值宽度扩展

```
int abc(int a,int b,int c)//a，b，c 为函数的参数
{
  int v;//定义的零时变量
  int r;//变量可以是不同类型，不同宽度
  v=a+b+c;//赋值语句
  r=v|a|b|c;//也可以是 if，while 或 for 语句
  return r;/*最后是返回语句，当返回值是 char（一个字节）时，返回值赋给 AL；为 short（两个字节）时，返回值赋给 AX；为 int 或 long 时，返回值赋给 EAX；若为 64 位（8 个字节）时，返回值赋给 EDX，EAX*/
}
```

当定义的变量是一个字节时，虽然计算机分配给该变量一个字节，但是该变量却占用了 4 个字节，为什么？

在计算机里面有个本机尺寸的概念，我们使用的是 32 位计算机，则计算机对 32 位支持比较好，主要类型是 32 位，cpu 针对 32 位会更快一些。针对其他位数，要进行复杂电路的改变，比如当使用 8 位时，计算机会先取 32 位，再屏蔽掉高位，只取低位，所以会更慢（早期的计算机）。现在计算机已经优化，但是编译器是早期写的，所以编译器还是遵循早期的规则，以本机尺寸为主。所以编译器在对一个字节宽度的变量分配 4 个字节，使用的时候只取一个字节。在我们在对一个字节做运算时，编译器通常将其扩展为 4 个字节赋给寄存器，运算结束后，再取低 8 位，因为编译器认为本机尺寸会运行更快。所以通常我们在写程序时，如非必要，通常定义成 int，而不定义为 char 或 short。因此定义成 char 并不节省空间反而会降低效率。

传入参数也可以作为局部变量使用，并不会改变原值。

```
int abc(void)//没有传入参数，void 也可以不写
{
  int i,r,j,h;
  i=1;//[ebp-4]:i
  r=2;//[ebp-8]:r
  h=3;//[ebp-0x10]:h
  j=4;//[ebp-0xc]:j
  j = abc(i,r,j);
/*
sub esp,0xc
mov dword ptr [esp],i
mov dword ptr [esp+4],r
mov dword ptr [esp+8],j
传递参数只是把表达式的值复制一份，所以子函数 abc 中使用和修改的只是 esp 至 esp+0xc 位置的值，和 ebp-4 到 ebp-0xc 没有任何关系*/
}
```

所以说，函数传参的过程只是一个复制的过程。

练习：

```
int ab(void)
{
  int a;
  int b;
  int c;
  int d;
  a=1;
  b=2;
  c=3;
```

```
  d=4;
  v = abc(a+0,b+0,c|0,d|0);//用汇编实现
  v += abc(a+2,b+2,c|2,d|2);//用汇编实现
  return v;
}
```

在汇编中，函数头部都会存在以下语句：

```
push ebp;
```

mov ebp,esp 是为了保证函数的稳定

返回值：当不需要返回值时，格式如下：

```
void abc(int a,int b)
{
  ......
  return;//也可以不写此句，写上更好，代表函数结束
}
```

问题：为什么汇编函数头要多减去 0x40，要执行 sub esp,0xXX,并赋值 0xcccccccc?

1> 都是 CC，代码即使出错，也不会引起错误，错误稳定，不会影响上一层。

2> 可以直观地观察栈的变化，快速裸眼定位。

问题：为什么函数开头要 push ebx,push esi,push edi?

当表达式比较简单时，只使用 eax，ecx 和 edx，在表达式比较复杂时，也会使用 ebx，esi，edi，所以要事先保存。

比如：

```
int abc(int a,int b)
{
  int v;
  v=a+b;//如果该处使用了 ebx，esi，edi，则破坏了原值
  ab(...);//如果这个函数依赖 ebx，esi，edi
  b=a+v;//则这条语句就会出错
```

```
}
```

编译器规定 eax，ecx，edx 用来做零时变量，当跨越到下一条 C 语句或函数时，不会对这三个寄存器值依赖，所以不用保存。

C 语言的函数调用协议：将 AL，AX，EAX 或 EDX 作为返回参数，且保证 EBP，ESP 以及 ebx，esi，edi 值保持不变，可以随便改变 eax，ecx 和 edx，并且会在子函数的上方（ebp+8 开始）来取参数变量。只要遵循该调度协议，使用汇编代码写函数可以有不同的书写方式。

当在函数前使用__declspec(naked)，自己定义函数结构。

练习：

自己写函数使入口参数从左向右传入。

说明：

```
__declpec(naked) abc(int a,int b)
{
  __asm
  {
    ......
    mov eax,dword ptr[ebp+0x8]
    mov ecx,dword ptr[ebp+0xc]
    mov dword ptr[ebp+0xc],eax
    mov dword ptr[ebp+0x8],ecx
  }
  ......
}
```

课后理解：

不同类型数据赋值时，使用以下指令：

Movsx : mov with sign-extension 符号位扩展

Movzx : mov with zero-extend ‘0’扩展

课后疑问：

本节没有疑问。

课后总结：

变量是内存中的一个位置，可以在该位置上存储值以供程序使用。

课后练习：

在汇编语言中查看各个数据类型的区别。

说明：

```
char sum;
char i;
sum  += i;
Mov    al,byte ptr [ebp-0x4]
Add    al,byte ptr [ebp-0x10]
Mov    byte ptr [ebp-0x4],al
sum = sum+i;
Movsx  ecx,byte ptr [ebp-0x4]
Movsx  edx,byte ptr [ebp-0x10]
Add    ecx,edx
Mov    byte ptr [ebp-0x4],cl
```

3.10 数组

本节主要内容：

1. 三目运算符
2. 数组
3. 多维数组

老唐语录：

```
v = a>b?a:b;//三目运算符，若 a 大于 b，取 a，否则取 b。
```

等价于：

```
if (a>b)
{
  v = a;
}
else
{
  v = b;
}
```

练习：

找出 8 个数中的最大值

```
int abc(int a,int b,int c)
{
  int v0,v1,v2,v3,v4,v5,v6,v7;
  int a;//存放最大数
  int b;//存放次大数
  int c;//存放第三大数
```

```
//给每个变量都赋一个初始值（省略）
  a=v0;
  a=a>v1?a:v1;
  a=a>v2?a:v2;
  a=a>v3?a:v3;
  a=a>v4?a:v4;
  a=a>v5?a:v5;
  a=a>v6?a:v6;
  a=a>v7?a:v7;
}
```

先在两个数中取小的那一个，肯定不等于 a 则肯定不是最大值，就可以赋给 b，

```
b=v0<v1?v0:v1;
if (v2<a && v2>b)//将 v2 改成 v3、v4……即可
{
  b=v2;
}
```

当需要的变量比较多时，比如一个学校的学生，则在定义时，会很麻烦，所以我们使用另一种简写的方法来定义一排变量。

```
int v[8];//表示定义 8 个变量，最小是 v[0]，最大是 v[7]
v[0] = 5;//初始化
v[1] = 2;
……
a=v[0];
a=a>v[0]?a:v[0];
a=a>v[1]?a:v[1];
a=a>v[2]?a:v[2];
a=a>v[3]?a:v[3];
a=a>v[4]?a:v[4];
a=a>v[5]?a:v[5];
```

```
a=a>v[6]?a:v[6];
a=a>v[7]?a:v[7];
```

其中，在申明一排变量时，方括号里面（个数）不可以是变量，必须是常量。

练习：

将 int v[8]改成 char/short/long v[8],查看汇编中 ESP 减少了多少？

在使用数组时，方括号内可以是表达式。

```
int abc(int a,int b)
{
  int score[20];
  int i;
  score[0] = 1;
  score[1] = 2;
  score[ ] = 3;//[]里面可以是任意表达式
  score[1,2,4,5]=10;//可以放逗号表达式,观察汇编，了解含义

}
```

把变量类型改为 char 或 short 时，并将方括号里面的常量改为变量，观察汇编变化，并观察函数头部 sub esp,后面的常量变化。

说明：

寻找对应的变量赋值：比如数组的首地址为 ebp-58h, 当为定义为 int 类型时，则 v[i]=1;汇编代码表示为：

```
sub esp,0x94//0x40+20*4+4(一个变量 i)
……
mov eax,i
mov dword ptr [ebp+eax*4-58h],1
```

若定义为 char，则 v[i]=1;汇编代码表示为：

```
sub esp,0x58//0x40+20+4(一个变量 i)
……
```

```
mov eax,i
mov byte ptr [ebp+eax-58h],1
```

当在对每个变量初始化时，如下：

```
v[0] = 0;
v[1] = 0;
v[2] = 0;
v[3] = 0;
v[4] = 0;
v[5] = 0;
v[6] = 0;
……
```

观察除了方括号里面数变化外，其他一律相同，我们可以使用循环语句代替：

```
for (i=0; i<20; i++)//i 为变量
{
  v[i] = 0;
}
```

当定义一排变量时，比如一个班有 5 个小组，每个小组有 9 人，则可以定义成：

```
int student[5*9];//5 和 9 都是常量
```

如果访问的值不在 student[0]到 student[44]之间，则没有意义，但是可以访问。C 语言上称之为越界。

也可以定义成：

```
int student[5][9];
```

比如一个县有 5 个学校，每个学校有 3 个年级，每个年级有 4 个班，每个班有 5 个组，每个组有 9 个人，则定义成：

```
int student[5][3][4][5][9];
```

现在要访问第 0 个学校的第 2 个年级第 0 个班第 3 个组第 2 个学生，则：

```
student[0][2][0][3][2]
```

练习：

将第 3 个学校第 0 个年级每个学生赋值，观察内存如何分配（先使用常量赋值，再使用变量，更易于观察汇编规律）。

```
int stud[3][2][5][7];//像这种结构称之为多维数组，等价于下条语句
int stud[3*2*5*7];
```

如果我要找到第 i 个年级，第 j 个班，第 k 个组的第 m 个学生，若定义为多维数组，则表示为：

```
stud[i][j][k][m]
```

若定义为一维数组，则表示为：

```
stud[i*(3*2*5)+j*(2*5)+k*5+m*1]
```

两种方法的功能实现一致。

练习：

给定变量值，手动计算多维数组对应的内存地址并验证。

说明：比如 `stud[0][0][0][i*(3*2*5)+j*(2*5)+k*5+m*1]`

注：函数的传入参数不能是数组。虽然能编译通过并使用，但不是数组，是另一种形式的简写（后面会提到）。

课后理解：

1.sizeof 获取变量或者数据类型的大小

2. 比如现在有 5 个班 9 个组 每组 8 个人 我们可以这样定义

```
int arry[5][9][8];
int _arry[5*9*8];
```

如果我们想访问 第 1 个班 第二个组 第三个人

```
arry[1][2][3] = 0;
```

等价于`_arry[ (1*9*8) + (2*8) + 3] = 0;`

课后疑问：

数组可以自加（++）或自减（--）吗？

回答：不可以，数组在内存中的位置是固定的，自加或自减后，内存地址改变，访问数

据不对。

课后总结：

数组表示一串变量。在定义数组时，如：int student[2][3][6][8/2];[ ]内在定义时必须为常量，应用时[]内可为表达式。

课后练习：

画出下列代码执行时内存值的变化：

```
unsigned char/int fun16(int number)
{
    unsigned char/int stud[3*6][45];
    stud[17][45-1] = 32;
    stud[17][45-2] = 45;
    stud[17][45-3] = 78;
    stud[17][0] = 99;
    stud[17][1] = 97;
    stud[17][2] = 91;
    stud[16][45-1] = 56;
    stud[16][45-2] = 72;
    stud[16][45-3] = 89;
    stud[16][0] = 11;
    stud[16][1] = 34;
    stud[16][2] = 35;
……
}
```

调用语句:ucr = fun16(10);

3.11 结构体

本节主要内容：

1. 掌握 C 语言结构体
2. 结构体占用空间计算

老唐语录：

我们现在只学习了三种数据宽度：char，一个字节；short，两个字节；int，long，4 个字节。如果我们想定义更大的数据宽度呢（数组只是将一种变量定义了很多遍）？

在函数外面写下列语句：

```
struct st//struct 是关键字，st 是用户自己定义的一个名字
{
  //可以定义多种类型
  int a;
  char b;
  short c;
};
```

在函数内部写下列语句：

```
st student1;
```

查看汇编中 sub esp,值变化。

struct 的含义不用理解，只要知道他把许多变量放在一起，变成一个大的数据类型。里面也可以定义一排变量（数组），也可以定义结构体。我们使用 sizeof（C 语言关键字）来判断变量的宽度，比如 sizeof(st)。

可以用 st 定义变量，又称 st 为变量类型。结构体可以作为返回值，也可以作为传入参数。数组不可以，因为返回值只能是一个变量，而不能是一排变量，同理，传入参数，一次只能 push 一个变量。也就是说，struct 和 int 没有区别，凡是使用 int 的地方，同

样可以使用 struct 定义变量。结构体内部不能使用自己。

要访问内部成员时，打“.”访问。比如 student1.a = 1;

练习 1：

```
struct student
{
  int a;
  int b;
};
struct stuBase
{
  int a;
  int b[10];
  student AB;
  stuBase sb;//错误，不能定义自身，因为无法确定大小
}
```

练习 2：

观察汇编：

```
struct student
{
  int a;
  int b;
  int c;
  int f;
  int d;
  int g;
};
int abc(student p)//返回值也可以是 student 数据类型
{
```

```
  student a;//正确，可以和内部成员同名
  a.a = 1;
  a.b = 2;
  a.c = 3;
  student p;//错误，变量名不可与传入参数相同
  return 0;
}
```

练习：

在定义结构体变量时，在函数内部定义和在函数外部定义的区别（可用汇编观察）

说明：

```
struct var
{
  int a;
  int score;
  char name[0x10];
  long add;
};
var a;
int abc(int a,int b)
{
  var v;
  var a;//如果在内部定义的变量名和函数外部的变量名相同，则系统默认为内部
  v.a=0;
  v.score = 0;
  v.add = v.add;
  return 0;
}
```

在函数外部定义的变量为全局变量，在内存中有一块固定的内存用来存放全局变量，被

称为**全局数据区**。而在函数内部定义的变量所使用的内存，将会被反反复复使用（函数每调用一次，使用一次），被称为**栈**。当函数返回时，定义的栈会失去作用，并被覆盖，只有 eax 作为返回值传出，而全局数据区永远有效，直至程序退出。

有符号型和无符号型只有比大小的时候有区别（可通过汇编观察）。

unsigned int/unsigned long 名称太长，可以取个别名。格式如下：

typedef unsigned int a;/*typedef 为关键字，unsigned int 为原来的名字，a 为新的名字。因此 a 就相当于数据类型 unsigned int*/

课后理解：

```
truct date
{
    char month;//占用 1 个字节
    char day;  //占用 1 个字节
    short year;//占用 4 个字节，short 型为 2 个字节
};
struct student
{
    int age;
    char name[15];//占用 16 个字节
    short score;  //占用 2 个字节，short 型为 2 个字节
    date dat;     //占用 6 个字节
    int number;
    char information[0x80];
};
```

说明：变量“占用”内存空间不仅与其定义的类型有关，和其定义的位置也有关系，所以相同类型的变量应放在一起，这样可以避免空间浪费；变量内部由低地址开始存储，变量之间由高地址开始存储。结构体为单个变量，数组为一串变量（由低地址存储）。

课后疑问：

```
struct st
{
  int a;
  char b;
  short c;
};
```

为什么使用 st.a=0;无法赋值？

回答：st 只是一个数据类型，相当于 int 或 char，必须用它申明一个变量使用。比如 st student;student.a = 0;

课后总结：

结构体就是不同数据类型的集合。变量“占用”内存空间不仅与其定义的类型有关，和其定义的位置也有关系，所以相同类型的变量应放在一起，这样可以避免空间浪费；变量内部由低地址开始存储，变量之间由高地址开始存储。结构体为单个变量，数组为一串变量（由低地址存储）。

课后练习：

画出下列代码执行时内存值的变化：

```
struct date
{
    char month;//占用 1 个字节
    char day;  //占用 1 个字节
    short year;//占用 4 个字节，short 型为 2 个字节
};
struct student
```

```
{
    int age;
    char name[15];//占用 16 个字节
    short score;  //占用 2 个字节，short 型为 2 个字节
    date dat;     //占用 6 个字节
    int number;
    char information[0x80];
};
int main(int argc, char* argv[])
{
    student a;
    student list[5][4][8];
    list[0][0][0].information[0] = 128;
    list[0][0][0].age = 20;
    list[0][0][1].age = 99;
    list[0][0][7].information[0x7C] = 0xAA;
//NOTE：此处超出数组定义范围，观察其写入内存位置为 list[0][1][0].age
    list[0][0][8].age =50;
    list[0][1][0].age =0x77;
    list[0][0][1].dat.day=10;
    list[0][0][1].information[0] = 1;
    a.age = 0;
    a.name[0] = 't';
    a.name[1] = 'a';
    a.name[2] = 'o';
    a.name[3] = '10';
    a.name[4] = 0x30;
    a.name[5] = 18;
    a.score = 55;
```

```
    a.dat.day = 56;
    a.dat.month = 10;
    a.dat.year = 2011;
    a.number = 1;
    return 0;
}
```

3.12 switch语句

本节主要内容：

1. switch 语句与 if 语句的联系
2. switch 中的大表与小表

老唐语录：

if()else()语句之间没有任何关系。但是在使用中通常自定义为有关系的。

练习：

书写 if else 语句，实现圆括号里面：

1> 是一个等号表达式

2> 等号左边的表达式相同

3> 等号右边的表达式是常量表达式

4> 常量表达式必须互不相同

```
if (表达式==常量表达式 1)
{
  语句 1;
}
else
if (表达式==常量表达式 2)
{
  语句 2;
}
else
if (表达式==常量表达式 3)
{
```

```
  语句 3;
}
else
{
  语句 4;
}
```

如果 if 语句满足上述条件，我们将之简化为：

```
switch(表达式)
{
  case 常量表达式 1:
       语句 1;
       break;
  case 常量表达式 2:
       语句 2;
       break;
  case 常量表达式 3:
       语句 3;
       break;
  default:
       语句 4;
       break;
}
```

switch 语句仅仅是 if 语句的一种简写方式。

例：

```
if (a+b==1)
{
  a++;
}
else
```

```
if (a+b==2)
{
  a+=2;
}
else
if (a+b==3)
{
  a+=3;
}
else
if (a+b==4)
{
  a+=4;
}
else
{
  b++;
}
```

可以改为：

```
switch(a+b)
{
  case 1:
       a++;
       break;
  case 2:
       a+=2;
       break;
  case 3:
       a+=3;
```

```
        break;
    case 4:
        a+=4;
        break;
    default:
        b++;
        break;
}
```

观察汇编运行。

执行下列操作：

1> 添加 case 后面的值，从 0 到 9，观察汇编变化。

```
switch(a+b)
{
    case 0:break;
    case 1:a++;break;
    case 2:a+=2;break;
    case 3:a+=3;break;
    case 4:a+=4;break;
    case 5:a+=5;break;
    case 6:a+=6;break;
    case 7:a+=7;break;
    case 8:a+=8;break;
    case 9:a+=9;break;
    default:b++;break;
}
```

3> 将 case 后面的值改成从 100 开始到 109，观察汇编变化

说明：多了一条 sub ecx,64h 语句，总结：在 switch 语句下方建一个表格，满足条件时，直接将地址赋给 eip（jmp），所以执行速度比 if 语句要快。

4> 将 case 后面的常量表达式改成毫不连续的值，观察汇编变化。

5> 将连续的十项中，抹去 1 项或两项，观察汇编有无变化

总结：case 后面的常量表达式顺序无关，编译器将常量表达式中的最小值和最大值取出来，只要从最小值到最大值是连续的，会建一张表，如果毫不连续，则就等价于 if 语句，如果将连续的十项中，抹去 1 项或两项，编译器仍然建一张表，只是将表中不存在的项填 default 的地址；如果将连续的二十项中，抹去较多项，编译器会建两张表，

6> 将连续的二十项中，抹去较多项。要求：一项一项删除，不要删最大项和最小项，并观察汇编，当删除到第多少项时，汇编中出现两张表。

说明：在 22 项（0 到 20+default）中，每减去一项，大表并没有减少，所以浪费了 4 个字节（地址为 4 个字节），如果全删了，那就全浪费了。删到一定程度时，建一张小表，每删除一项，只浪费一个字节。所以编译器在权衡空间和时间的浪费来决定建几张表。

课后理解：

在有规则且 case 值连续的 switch 语句中，汇编在代码 ret 返回的结尾处生成一张存储 case 值的表，每个 case 值占 4 个字节，称为大表，每次使用时调用。当按操作删除 case 值（最大值与最小值保留）到第 9 个 case 时，该表存储个数由 22 变为 13，此刻内存又生成一张表，每个 case 值占 1 个字节。为大表时，每去掉一个 case，系统浪费 4 个字节，而建立小表只需要 n（n 为 case 中的最大值减去最小值）个字节，依此判断何时改变表，因为系统的原则是节省空间。当然，每个编译器自行规定何时改变表的类型，所以 Switch 的执行速度快于 if 语句。

课后疑问：

default 语句后面可以省略 break 语句吗？

回答：可以，但是最好加上。

课后总结：

switch 语句是特殊条件下的 if 语句，但是比 if 语句执行效率高。

课后练习：

画出下列代码执行时内存值的变化情况：

```
int fun(int a, int b, int c)
{
switch(a + b + c)
{
case 108:
{
printf("108");
break;
}
case 105:
{
printf("105");
break;
}
case 107:
{
printf("107");
break;
}
case 100:
{
printf("101");
break;
}
case 115:
{
```

```
printf("115");
break;
}
case 118:
{
printf("118");
break;
}
case 128:
{
printf("128");
break;
}
case 104:
{
printf("104");
break;
}
default:
{
printf("other");
break;
}
}
return 0;
}
int main(int argc, char* argv[])
{
int temp ;
```

```
temp = fun(100,0,0);
return 0;
}
```

3.13 define与typedef

本节主要内容：

1. 掌握 define 语法
2. 掌握 typedef 语法
3. typedef 与 define 区别
4. #if 语句

老唐语录：

#define：给任何代码起一个别名，并在编译之前预处理。而 typedef 只能给合法的类型取别名。

比如：

```
#define pp int a,int c,int d//pp 是取的名字
```

则 int abc(int a,int c,int d)等价于 int abc(pp)

使用别名的原因：有些代码要经常使用，使用简短的别名代替，书写会更方便。也可以对常量取个别名，便于阅读，并且替换方便。

练习：

给 if 后面的括号取个别名。

注：只能给单行取别名，并不以一个单词的中间开始或结束,typdef 可以换行。

注释语句：

单行注释：//......

单行或多行注释：/*......*/

其他注释方法：

```
#if 表达式 1
c=a+b;
#elif 表达式 2
```

```
c=a+c;
#else
c=b+c;
#endif
```

解释：如果表达式1成立执行c=a+b;否则判断表达式2是否成立，成立则执行c=a+c;否则执行c=b+c;

#if defined (DD) 等价于 #ifdef DD//如果定义过DD

#if !defined 等价于 #ifndef DD //如果没有定义过DD

经典的算法：冒泡排序和折半查找

冒泡排序：

```
int arr[] = {5, 4, 1, 3, 6};
int i = 5, j;
int temp;
while(i > 0)
{
   for(j = 0; j < i - 1; j++)
   {
      if(arr[j] > arr[j + 1])
      {  temp = arr[j];
         arr[j] = arr[j + 1];
         arr[j + 1] = temp;
      }
   }
   i--;
}
```

折半查找：

```
int half_seek(int arr[], int low, int high, int num)
{
```

```
    int mid;
    mid = (low+high)/2;
    if(arr[mid]==num)
    {
        return mid;
    }
    else if(arr[mid]>num)
    {
        high = mid-1;
    }
    else
    {
        low = mid+1;
    }
    return half_seek(arr,low,high,num);//递归
}
```

变参函数：

我们曾使用过的 printf 属于变参函数。

```
void function(var1,var2,...)
{
int a;
...
return ; //无返回值的情况
}
```

可变参数函数

```
int fun(int a,int b,...)
{
... ...
}
```

调用语句：fun(1,2,3,4,5,6,7,无限多个);

参数：作为函数体本身，在被调用之前并不知道 user 传入参数个数与类型(例如 printf)

优点：多元化功能支持；可以省略参数类型；可以省略参数个数

使用：可由第一个传入参数规定参数的个数或类型

画出下列代码执行时内存值的变化：

```
int fun18(int n,int a,...)
{
int temp[9];
int i;
int p1;
if (1==n)
{
__asm
{
mov eax,dword ptr[ebp+0xc]
mov temp[9],eax
}
}
else if (2==n)
{
__asm
{
mov eax,dword ptr[ebp+0x10]
mov temp[8],eax
mov edx,dword ptr[ebp+0xc]
mov temp[9],edx
}
}
```

```
else if (9==n)
{
__asm
{
mov ecx,9
lea edi,[ebp-36]
lea esi,[ebp+12]
rep movsd
}
//或者使用 for 循环：
for (i=0;i<9;i++)
{
__asm
{
mov edx,i
mov edx,edx
mov edx,edx //以上操作完成将 i*4 附给 edx
mov eax,dword ptr [ebp+8+8+edx]
//此处不可以将[ebp+16+edx]的值直接附给 p1，目标和源不可同时为内存地址
mov p1,eax
}
temp[i] = p1;
}
}
return a;
}
```

调用语句:itemp = fun18(9,2,3,4,5,6,7,8,9,10);

栈平衡：

C 语言默认为为外平栈：_cdecl

_stdcall 表示内平栈，内平栈可以节省空间、运行速度快。

练习：

观察在函数头部加入_stdcall 后，汇编代码的变化。

课后理解：

#define PI 3.1415926（横线上的内容除末尾不能出现空格外无任何要求）

#define 与 typedef 的区别

前者只进行文本替换，无语法检测；后者进行语法检测。结构体，指针、函数定义用typedef。

预处理：

#if 常量

#else

#endif

或

#if 常量

#elif 常量

#endif

注释的艺术：

///*

... ...

//*/

取消该注释时，只需在开始处加//

#ifdef debug //检查该符号“debug”有无被定义过（与值大小无关）

课后疑问：

可以在函数外部使用#if+变量吗？

回答：不可以，只能使用常量。

课后总结：

给常量或表达式取别名只是为了方便书写或修改。

课后练习：

自己写一段程序，并且将所有表达式变量使用 `define` 或 `typedef` 表示。

3.14 指针

本节主要内容：

1．类型转换

2．指针申明与赋值

3．指针的加减运算

4．地址赋值运算

老唐语录：

一个变量最大的属性就是宽度，定义变量时可以在前面加 unsigned，还是原来的宽度，只是在比较大小的时候，使用的 jcc 检测条件不同。

我们现在学习指针，指针的标准写法：

```
int* p;//变量类型 + * + 空格 + 变量名 + 分号
```

“*”号个数可以任意多个，类型可以是 char、short，但是宽度永远是 4 个字节。

练习：

定义多个带“*”号的变量，观察汇编中 sub esp 的大小变化。

注：“*”号可以任意多个。

可以将任意表达式赋给等号左侧，只要两边的类型相同。赋值语句的标准写法：

```
char a;
a = (char)0;
```

需要在等号右侧先加上变量的类型，再跟上表达式。以前的类型简单，可以省略，带“*”号的不可以。比如：

```
char***** p;
p = (char*****)1;
```

一个变量的属性除了宽度之外还有其他属性，比如 unsigned int 和 int，在比大小时不同。一个带“*”号的变量，也可以自加（++）和自减（--）,但是含义不同。

练习：

分别定义 int、char、short 并且“*”号为一个或多个的变量，然后++和--，观察汇编变化。

说明：没有“*”号的变量++或--的时候，加 1 就是加整数 1，加 5 就是加整数 5，带“*”号的变量，p+1 就是 p 去掉一个“*”号后数据类型的宽度。所以：

```
char* p;//p+1 加的长度是 1
char** p;//p+1 加的长度是 4
int****** v[10];//定义了 10 个 int******的变量，一排变量
struct lv
{
  int*** a[10];
  char*** b[10];
};
lv* tt;//占 4 个字节
lv t2;//占 80 个字节
```

可以使用

#define charp char**********简化指令。

指针相减：指针之间可以相减，但是必须是完全相同的指针类型，且减去的结果要除以去掉一个“*”号的宽度。

练习：

看汇编

```
int abc(int a,int*** b)
{
  int****** p1;
  int****** p2;
  p1 = (int******)7;
  p2 = (int******)3;
  return (int)(p1-p2);
```

```
}
```

总结：如果 p1=p2+2 需要 2*4，那么 p1-p2=2 也是除以 4 的结果。

练习：

p2+2-p1-2 与 p2+2-(p1+2)结果是否相同？

说明：

p2+2-p1-2 等价于((p1+2) -p1)-2

（int 型）(（指针型）(p1+2) -p1)-2

p2+2-(p1+2) 等价于(p2+2)-(p1+2)）

（int 型）(指针型)（(p2+2)- （指针型）(p1+2)）

当两个指针比大小时，是当作无符号处理的，当指针做运算时，当作有符号处理。

```
int abc(int a,int b)
{
  int***** p1;
  int***** p2;
  p1 = (int*****)7;
  p2 = (int*****)5;
  if (p1>p2)//观察汇编中的 Jcc 指令，可知为无符号
  {
  }
  if (p1-p2)//加减是有符号的，可以看汇编中的 SAR 指令
  {
  }
}
```

&是地址符，类型是后面的变量类型加一个“*”号，比如 int a;&a 的类型是 int*，但是不能加减，也不能赋值，仅仅是个地址，一个数，不是变量。任何变量都可以取地址，无论有几个“*”号。

练习：

定义一个“*”和多个“*”的变量，并且加上取地址符，观察汇编。

练习：

定义一个结构类型，分别在前面加一至多个“*”号，观察区别。

总结：表达式前面可以加“*”号，比如**(&a+5)；int*** d;*d 的类型是 int**，**d 的类型是 int*。所以赋值语句**d = (int*)1;

课后理解：

在对变量赋值时，通常见到的是简化版，标准赋值语句如下：

```
int i;
char c;
i = (int)3;
c = (char)0;
```

简化版如：2+(3*4) 简化为 2+3*4，但是有些不能简化，如：2*(3+4) 不能简化为 2*3+4。

有时需要对变量进行类型转换，如：

```
int i;
char c;
i=3;
c=(char)(i+3);
struct {int a;int b;}V;
```

通常在对变量赋值时遵循以下步骤：

① 判断变量宽度：sizeof(V)

② 赋值：V.a = 1;V.b = 2;

指针，可以定义为带‘*’的数据类型，如：

```
char**************
short******
unsigned int*
```

申明指针变量：

```
char**************  cp;
```

```
short******         sp;
unsigned int*    ip;
cp = (char**************)1;
sp = (short******)3;
```

指针宽度为 4 个字节（32 位架构操作系统）。关于 void 数据类型避免使用，系统分配给 void 申明的变量 0 个字节，故不可用；void*为指针，占固定宽度 4 个字节。

关于地址，可以简单定义为一个数，在数组中，“&数组名“表示取整个数组的首地址；“&数组名[0]”表示取数组第一个元素的首地址，可以简化为“数组名”。

课后疑问：

如何给 int******* p;赋值并使用？

回答：

```
int****** p1;
int***** p2;
int**** p3;
int*** p4;
int** p5;
int* p6;
int p7;
p = &p1;
*p = &p2;
**p = &p3;
***p = &p4;
****p = &p5;
*****p = &p6;
******p = &p7;
p++;
```

课后总结：

1.带“*”号的变量宽度一定是 4

2.凡是带“*”号的变量，赋值时使用标准赋值语句，不能简写

3.带“*”号的变量的加减和普通变量有区别，加减的宽度是去掉一个“*”后新数据类型的宽度。

课后练习：

练习 1：

通过汇编代码观察指针宽度

```
struct aaa
{
    int**     a;
    char***   b;
    short**** c;
};
    void*  p; //4 bit
    void   ptr;   //没有宽度，或任意宽度，避免使用
    aaa    v1; //占 12 个字节
    aaa    v2[0x10]; //12*16 个字节
    aaa*** v3[0x10]; //4*16 个字节
    char   cp1;
    char*  cp2;
    char** cp3;
    char***   cp4;
    char****  cp5;
    char***** cp6;
```

```
char******   cp7;
char*******  cp8;
char******** cp9;
char*********   cp0;
short           sp1;
short****       sp2;
unsigned int    ip1;
unsigned int*   ip2;
int i=0,j;
p=(void*)i;
struct  aa
{
   int a;
   int b;
   int c;
}*** v[0x10],c;
sp1=(short)(5+3);
sp2=(short****)(3+11);
c.a=1;
v[0]=(aa***)-1;
v[1]=(aa***)(v[0]);
v[0]=(aa***)(c.a+3);
cp1=(char)3;
cp2=(char*)3;
cp3=(char**)3;
cp4=(char***)3;
cp5=(char****)3;
cp6=(char*****)3;
cp7=(char******)3;
```

```
    cp8=(char*******)3;
    cp9=(char********)3;
    cp0=(char*********)3;
    ip1=(unsigned int)3;
    ip2=(unsigned int*)3;
```

说明：指针宽度为 4 个字节（32 位架构操作系统）。关于 void 数据类型避免使用，系统分配给 void 申明的变量 0 个字节，故不可用；void*为指针，占固定宽度 4 个字节。

练习 2：

通过汇编代码观察指针加减的长度

```
int**** fun19(int**** a,int***** b)
{
        int i;
        i=3;
        i+=2;
        a=a+1;//加减 1 的长度为去掉一个*后数据类型的长度
        return(int****)3;
}
```

关于地址，可以简单定义为一个数，在数组中，“&数组名“表示取整个数组的首地址；“&数组名[0]”表示取数组第一个元素的首地址，可以简化为“数组名”。

练习 3：

画出下列代码执行时内存值的变化：

```
struct student
{
    char sex;
    char name[0xf];
    int age;

};// length 0x14
```

```
int**** fun19(int**** a,int***** b)
{
    char at0[0x10];
    int i;
    char* cp1;
    char* cp1_2;
    char** cp2;
    char** cp2_2;
    student stu3;
    student* stu1;
    student** stu2;
    stu1 =(student*)3;
    stu2 =(student**)stu1;
    stu2=(student**)stu1;
    stu3.age=1;
    i = (int)&stu3;
    i = (int)at0;
    i = (int)&at0;
    i = (int)&at0[0];
    i = (int)(at0+1);//add 0x1
    i = (int)(&at0+1);//add 0x10
    i = (int)(&at0[0]+1);//add 0x1
    ++stu1;
    cp1=(char*)1000000;
    cp1_2=(char*)10;
    cp1=cp1_2;
    cp2=(char**)2;
    cp2_2=(char**)1;
    stu2=(student**)&cp2;
```

```
    cp2_2=(char**)&i;
    i=(int)&stu1;
    cp1++;
    cp2+=2;
    i+=2;
    a++;
    return(int****)3;
}
```

调用语句:

```
fun19((int****)3,(int*****)4);
```

练习 4：

观察 if 语句后{}中的语句是否执行？

```
char****** p1;
char****** p2;
p1=(char******)3;
p2=(char******)5;
if (p2-p1==0)
{
    p2=0;
    p1=0;
}
```

练习 5：

观察当传参为数组时，其实际传入值。

```
void fun21(int ar[]) { ar++; }
```

调用语句:

```
int ir[5]={0,2,4,6,7};
int* ip=(int*)3;
fun21(ir);
```

```
fun21(ip);
```

说明：实际传入的参数是指针。所以数组不能作为传参。

练习 6：

看汇编，区别每一条语句

```
int i;
int********* d;
d=(int*********)&d;
i=(int)(d);
i=(int)(*d);
i=(int)(**d);
i=(int)(***d);
i=(int)(****d);
i=(int)(*****d);
i=(int)(******d);
i=(int)(*******d);
i=(int)(********d);
i=(int)(*********d);
```

练习 7：

看汇编，区别每一条语句

```
struct student
{
    char name[10];
    int age;
    char sex;
};
void main()
{
    int i;
```

```
    student********* d;
    student* d2;
    d=(student*********)&d;
    i=(int)(*********d).age;
    i=(int)(*********d).sex;
    i=(int)(*********d).name;
    i=(int)(*********d).name[0];
    (*********d).age=9;
    (********d)++;//加 1 的长度为结构体长度
    (*********d).age++;
    (*d2).age=5;
    d2->age=15;
}
```

说明：通过以上实例可以得出：

```
student********* d;
student* d2;
```

d2++ 等价于 (********d)++

(*********d).age = 1 等价于 (********d)->age = 1

练习 8:

简化下列赋值语句。

```
int d;
int* p1;
int** p2;
int*** p3;
int**** p4;
int***** p5;
int****** p6;
p1=(int*)&d;
```

```
p2=(int**)&d;
p3=(int***)&d;
p4=(int****)&d;
p5=(int*****)&d;
p6=(int******)&d;
```

说明：

```
p1=(int*)&d;//简化：p1=&d;
p2=(int**)&d;//p2=&p1;
p3=(int***)&d;//p3=&p2;
p4=(int****)&d;//p4=&p3;
p5=(int*****)&d;//p5=&p4;
p6=(int******)&d;//p6=&p5;
```

其中，int****** p6 的标准写法位 int(*(*(*(*(*(* p6))))))

练习 9：

观察下列 C 代码在汇编中区别。

```
int i=1;
*(p1+i) = 2;
p1[i] = 2;
*(*(p2+i)+i) = 3;
p2[i][i] = 3;
*(*(*(p3+i)+i)+i) = 4;
p3[i][i][i] = 4;
*(*(*(*(p4+i)+i)+i)+i) = 5;
p4[i][i][i][i] = 5;
*(*(*(*(*(p5+i)+i)+i)+i)+i) = 6;
p5[i][i][i][i][i] = 6;
*(*(*(*(*(*(p6+i)+i)+i)+i)+i)+i) = 7;
P6[i][i][i][i][i][i] = 7;
```

说明：指针符号*()可以由[]代替。

练习 10：

*()与[]相互转换。

```
*(*(p6+5)[4]) = p3;

*(*(*(p6[3]+4)[3]+3)[2]+2)= i;

(*p3[2]+2)[1] = 1;

(*(*((*p6-i)[-2]-3)[-10000]+3000))[-100] = 200000;
```

3.15 结构体指针

本节主要内容：

1. 掌握结构体指针
2. 数组指针和指针数组
3. 函数指针

老唐语录：

指针简写：*p =>*(p+0) => p[0]。

通用格式：*(p+i+j) => (p+i)[j] => (p+i)[j] => p[i+j]

练习：

练习指针的简写形式。

```
struct Arg
{
  int a;
  int b;
  int c;
};
```

定义一个结构体指针：

```
Arg* p;
```

则要给结构体指针内部的成员赋值使用(*p).a = 0。简写形式为：p->a = 0;

练习：

练习结构体指针的使用

```
struct intstam
{
  int a;
```

```
  int b;
  int c;
};
int abc2()
{
  intstam* p;
  char buffer[0xc];
  p = (intstam*)buffer;
  (*p).a=0;
  (p-1)[1].b=0;
  p->c = 0;
return 0;
}
```

数组指针与指针数组

```
int  array[10];//40byte
int** array[10];//40byte，指针数组
```

定义一排变量的指针：

```
int (***array)[10];//4byte，数组指针
int*** (****p)[10];//那么判断 sizeof(*****p)的大小等价于多少？
sizeof(*((****)p+0))=sizeof(****p[0])
void* p;//宽度是4个字节，可以指向任意宽度。
```

练习：

```
int abc(int a,int b[10][5][4])
{
  int c[10][5][4];
  c[1][2][3] = a;
  b[1][2][3] = 0;
  b++;//b可以自加，说明b不是数组，可以通过汇编观察，仅push了4个字节
```

```
  return 0;
}
```

所以说，函数名后面的圆括号内不能写数组，因为传入的是指针（int* b）。

任何数据都可以定义指针。

函数本身也占用空间，可以看作一个变量，也可以定义一个指针，格式：

```
int (*funName)(int a,int b);
```

当然，返回值也可以是一个指针：

```
int* (*funName)(int a,int b);
```

扩展：

```
int*** (****funName)(int a,int b);
```

练习：

将整数 5 赋给函数指针 funName（注意观察编译器的报错）。

由于函数指针不知道宽度，所以不能使用 sizeof，不可以使用*****funName。也不可以++或--。

课后理解：

```
struct aaa
{
    int**    a;
    char***   b;
    short**** c;
};
void*  p; //4 字节
void   ptr;   //没有宽度，或任意宽度，避免使用
aaa    v1;//占 12 个字节
aaa    v2[0x10]; //12*16 个字节
aaa*** v3[0x10]; //4*16 个字节
```

课后疑问：

定义变量 int*** (****arrayp)[10];如何对数组成员赋值？

回答：

```
int*** p1;
int** p2;
int* p3;
int p4;
arrayp = (int***(****)[10])&p1;
*arrayp = (int***(***)[10])&p2;
**arrayp = (int***(**)[10])&p3;
***arrayp = (int***(*)[10])&p4;
****arrayp[0] = (int***)1;
****arrayp[1] = (int***)2;
……
```

课后总结：

1.两个指针相减为指针间的距离，所以要将结果除以去掉一颗*号数据类型的长度。

2.当数组作为传参时，函数只将数组的首地址调入。

3.在结构体指针运用时，->可替代*. 取结构体内的子变量。

课后练习：

练习 1:

观察结构体数据类型指针使用

```
struct student
{
    int age;
```

```
    int score;
    char name[20];
    char sex;
};
void main()
{
    int id = 4;
    student* stu1 = NULL;
    student** stu2 = NULL;
    student*** stu3 = NULL;
    student**** stu4 = NULL;
    student***** stu5 = NULL;
    student****** stu6 = NULL;
    (student*)stu6++;
    (*****stu6)++;
    student i = {15,90,"TAO JIE",'m'};
    student ic[4];
    stu6[2] = stu5;
    *(stu6[-3]-5) = stu4;
    *(*(stu6[-2]+4)[3]-1) = stu2;
    *(*(stu6[-2]+4)[3]-1)[5] = stu1;
    *(*(stu6[-2]+4)[3]-1)[5] = &i;
    (**(*(stu6[-2]+4)[3]-1)[5]).age = 1;
    (*(*(stu6[-2]+4)[3]-1)[5])[1] =ic[1];
    (*(*(stu6[-2]+4)[3]-1)[5])[1] =*(ic+1);
    (*(*(stu6[-2]+4)[3]-1)[5])->age = 1;
    stu1 = &i;
    stu1->age = 1;
    *(&(*(stu1+4))) = ic[2];
```

```
    *(-id+stu6) = stu5;
    *(-id-3+stu6)[3] = stu4;
    ((ic+2)[2]).age = 3;
//p5 = &(&p3);//出错：常量不可以取地址
}
```

练习 2:

观察下列指针在 if 语句中的符号。

```
int* p1;
int* p2;
p1=(int*)5;
p2=(int*)3;
if (p1>p2+7)//指针比大小，无符号
{
    p1=0;
}
if (p1-p2>7)//指针相减，有符号
{
    p1=0;
}
unsigned int* p1;
unsigned int* p2;
p1=(unsigned int*)3;
p2=(unsigned int*)5;
if (p1-p2>7)//指针相减，有符号
{
    p1=0;
}
```

练习 3:

判断下列语句的可执行性。

```
void****** p;
p=(void******)5;
p++;
(****p)++;
(*****p)=0;//不能加但可以赋值
```

练习 4:

判断下列语句的可执行性。

```
stu d;
stu *p;
stu dd;
p=&dd;
*p=d;//结构体间也可以直接赋值
(*p).n = d.n;
p->n = d.n;
```

练习 5:

判断下列语句的可执行性。

```
struct stu
{
    int**** (*****p)[0x10][0x3][0x7];
    char**** cp;
    short***** ar[0x10];
};
void main()
{
stu**** (*****p)[0x10][0x3][0x7];
(*****((****(*(*(*(*****p)+15)+2)+6)).p))[15][2][6]=
=(int****)2 ;//正确
```

```
    *(*(*(*****((****(*(*(*(*****p)+15)+2)+6)).p))+15)+2)+6
=(int****)2 ;//错误
```

则解决方案由下：

```
    #define tang  (*****((****(*(*(*(*****p)+15)+2)+6)).p))
    tang[15][2][6] =(int****)2 ;
    (*(tang+15))[2][6] =(int****)2 ;
    (*(*(tang+15))+2)[6] =(int****)2 ;
    *((*(*((*****((****(*(*(*(((*****p)+15)+2)+6))).p))+15))+2)+6)
=(int****)2 ;
    }
```

说明：(*p-3)[2] = (*p-2)[1] = (*p-0)[-1] = (*p)[-1]= (*(p-0))[-1]

//脱去第一层从首地址开始，第二层从首地址-1*(去掉一颗*后数据类型长度)处开始

练习 6:

判断下列语句的可执行性。

```
    int*** funt(char a[0x10][0x5][0x3],int b)
    {
        int****** (******p)[0x10][0x8][0x6];
        *(int*)&p=b;
        a++;
        a[3][2][1]='A';
        p[6][5][4][3][2][1][3][2][1][1][2][3][4][5][6]=2;
        return 0;
    }
```

说明：

```
    int*** funt(char a[0x10][0x5][0x3],int b)
```

标准写法：int*** funt(char[0x5][0x3] (*a),int b)

2.类型强转不报错：*(int*)&p = b;

解析：

```
int** pp;
pp = (int*)&p;//p 为任意指针
*pp = b;
3. a[3][2][1]='A';//观察汇编
4. p[6][5][4][3][2][1][3][2][1][1][2][3][4][5][6]=2;
```

练习 7:

观察下列赋值语句的正确性和含义

```
stu dd;
stu *p;
stu** p2;
stu*** p3;
stu**** p4;
p++;
p = &dd;
p = (&dd+1);//+28
p2 = &p;
p3 = &p2;
p4 = &p3;
```

练习 8:

别名重定义

```
typedef unsigned long DWORD;
typedef unsigned long** DPP;
DPP a;
typedef  struct student
{
    int n;
    int age;
    short score;
```

```
} *** stup3;
stup3 sp;
typedef int*** (***aaa)[0x10][3][2];
aaa p6;
```

练习 9:

分析下列代码。

```
typedef  struct student{
int n;int age;short score;
}**** (***stup5)[3][2][1],*** stup3;
stup5*** (**td6)[4][5][6];
td6[3][2][1][5][6][7][1][1][1][4][3][1][2][3][2][2][1]->n     =
3;//分析
```

说明：对于指针，*(pp+i+j) <=> (pp+i)[j]；对于结构体，p-> <=> *(p)。

第四章 硬编码

引言

上一章我们理解了高级语言（主要是 C）如何转换成汇编，这一章我们将汇编转换成二进制码，这样我们可以全盘了解计算机的工作机制。

本章必须要掌握的知识点：

1. 固定长度的指令编码
2. 不定长度的指令编码
3. 指令结构中的 ModR/M 字段

本章常犯的错误：

1. 指令长度的判断
2. ModR/M 字段的理解

4.1 定长编码

本节主要内容：

1. 0x50-0x5f 编码
2. 0x40-0x4f 编码
3. 0xb0-0xbf 编码
4. 0x70-0x7f 编码
5. 0x0f 0x80-0x0f 0x8f 编码

老唐语录：

在 VC6 汇编窗口中右键 code bytes 可以看见指令对应的十六进制（编码），我们以前记过寄存器的顺序，eax，ecx，edx，ebx，esp，ebp，esi，edi，硬编码也按寄存

器的顺序递增，比如：

```
0x50:push eax
0x51:push ecx
0x52:push edx
0x53:push ebx
0x54:push esp
0x55:push ebp
0x56:push esi
0x57:push edi
0x58:pop eax
0x59:pop ecx
0x5a:pop edx
0x5b:pop ebx
0x5c:pop esp
0x5d:pop ebp
0x5e:pop esi
0x5f:pop edi
```

现在进行操作码编程，如果想书写 push ebp，则可以用

```
__asm
{
  _emit(0x55)//push ebp
}
```

push eax 到 pop edi 指令长度是一个字节

inc eax(eax 自加 1)的编码是 0x40，dec eax（eax 自减 1）是 0x48，则 0x40 到 0x4f 的指令是：

```
0x40:inc eax
0x41:inc ecx
0x42:inc edx
0x43:inc ebx
```

0x44:inc esp

0x45:inc ebp

0x46:inc esi

0x47:inc edi

0x48:dec eax

0x49:dec ecx

0x4a:dec edx

0x4b:dec ebx

0x4c:dec esp

0x4d:dec ebp

0x4e:dec esi

0x4f:dec edi

有些代码简单，有些编码复杂。

0x90:xchg eax,eax//使用汇编测试得出为 nop（交换 eax 和 eax 的值没有意义）

0x91:xchg ecx,eax

0x92:xchg edx,eax

0x93:xchg ebx,eax

0x94:xchg esp,eax

0x95:xchg ebp,eax

0x96:xchg esi,eax

0x97:xchg edi,eax

一个字节的编码指令很多，ret 的编码是 0xc3.int 3 的编码有两种，一种是 0xcc，一种是（0xcd，0x3）。int 0x01，单步指令，编码是 0xf1，或是（0xcd，0x1）。

两个字节编码：

0xB0 0x12:mov al,0x12 //一个字节立即数又称为 ib，所以可以写成 0xb0 ib

0xb1 ib:mov cl,ib

0xb2 ib:mov dl,ib

0xb3 ib:mov bl,ib

0xb4 ib:mov ah,ib

0xb5 ib:mov ch,ib

0xb6 ib:mov dh,ib

0xb7 ib:mov bh,ib

mov eax,0x12345678,则这条指令至少占 5 个字节：0xb8，0x78，0x56，0x34，0x12。其中四个字节的立即数也可以总括为 id。

0xb8 id:mov eax,id

0xb9 id:mov ecx,id

0xba id:mov edx,id

0xbb id:mov ebx,id

0xbc id:mov esp,id

0xbd id:mov ebp,id

0xbe id:mov esi,id

0xbf id:mov edi,id

练习：

写编码，看汇编。熟悉以上指令。

练习：

在 _emit()里面填写 0x70 到 0x7f 中的一个，再跟一个字节（随意填写），观察汇编。

说明：是条件跳转指令，16 种。

0x70 ib:jo ib//ib 为 1 个字节立即数：如果条件成立，跳转到本指令地址+本指令长度+ib（有符号型，大小从 0x80 到 0x7f）位置；条件不成立，执行下一条指令

0x71 ib:jno ib

0x72 ib:jb ib

0x73 ib:jae ib

0x74 ib:je ib

0x75 ib:jne ib

0x76 ib:jbe ib

0x77 ib:ja ib

0x78 ib:js ib

0x79 ib:jns ib

0x7a ib:jp ib

0x7b ib:jnp ib

0x7c ib:jl ib

0x7d ib:jge ib

0x7e ib:jle ib

0x7f ib:jg ib

以上跳转为单字节跳转，跳转位置为本指令向前 0x80，向后 0x7f。下面介绍的是四个字节跳转。

0x0f 0x80 id:jo id//前两个字节确定，后面跟四个字节立即数，整条指令长度为 6 个字节。

0x0f 0x81 id:jno id

0x0f 0x82 id:jb id

0x0f 0x83 id:jnb id

0x0f 0x84 id:je id

0x0f 0x85 id:jne id

0x0f 0x86 id:jbe id

0x0f 0x87 id:ja id

0x0f 0x88 id:js id

0x0f 0x89 id:jns id

0x0f 0x8a id:jp id

0x0f 0x8b id:jnp id

0x0f 0x8c id:jl id

0x0f 0x8d id:jge id

0x0f 0x8e id:jle id

0x0f 0x8f id:jg id

0x0f 0x80 到 0x0f 0x8f 是改变 eip 的指令编码。

0xe0 ib :loopne//先判断 zf 是否等于 1，条件成立，将 ecx 减 1，然后判断 ecx 是否为 0，条件成立，跳转

0xe1 ib:loope//功能与 loopne 相反

0xe2 ib:loop//只判断 ecx（先将 ecx-1）是否为 0

0xe3 ib:jecxz//只判断 ecx（ecx 不需要加减处理）是否为 0

0xe8 id:call id//五个字节长度。跳转地址位置：当前指令地址+指令长度（5 个字节）+id。

0xe9 id:jmp id//跳转指令，跳转地址位置：当前指令地址+指令长度（5 个字节）+id。

除了 8 个通用寄存器和 eflag，eip 寄存器外，还有 8 个段寄存器（顺序为 0-7）：

ES，CS，SS，DS，FS，GS，LDTR，TR，段寄存器是 96 位，定义为结构体形式：

```
struct segment
{
  word selctor;//可见，也就是说，汇编指令只能访问这 16 位
  word attributes;//隐藏
  dword base;//隐藏
  dword limit;//隐藏
};
```

0xea Ap:jmp cs:id//Ap 代表 6 个字节长度的数：高两个字节赋给 cs，低四个字节直接赋给 eip，指令长度为 7 个字节。

0xeb ib:jmp ib

0xc3:ret

0xc2 iw:ret iw//iw 代表 2 个字节立即数 esp+4+iw

0xcb:retf//从栈中弹出 8 个字节，低四个字节给 eip，高四个字节给 cs（只有低 2 个字节有效）,retf(ret far),ret(另一种写法是 retn:ret near)

0xca iw:retf iw//3 个字节长度，esp+8+iw，其他功能和 0xcb 相同

0x64:fs

0x65:gs

课后理解：

_asm

```
{
    _emit(0x40-0x47)inc eax – inc edi
    _emit(0x48-0x4f)dec eax – dec edi
    _emit(0x50-0x57)push eax – push edi
    _emit(0x58-0x5f)pop eax – pop edi
    _emit(0x90-0x97)xchg eax,eax - xchg edi,eax
    _emit(0xb0-0xb7)mov al,ib – mov bh,ib
    _emit(0x90-0x97)mov eax,id – mov edi,id
}
```

课后疑问：

本节没有疑问。

课后总结：

0x40-0x47:inc erx//erx 代表 eax 到 edi

0x48-0x4f:dec erx

0x50-0x57:push erx

0x58-0x5f:pop erx

0x90-0x97:xchg erx,eax

0xb0-0xb7:mov rb,ib//rb 为 8 位宽度的寄存器

0xb8-0xbf:mov erx,id

指令长度最短为 1 个字节，最长为 15 个字节。

课后练习：

练习 1：

使用 vc6 测试 0x60，0x61，0x64，0x65，0x66，0x67，0x68，0x6a，0x6c，0x6f，0x98 到 0x9f，0xa0 到 0xaf，0xc2 到 0xc7，0xe0 到 0xef，0xf0 到 0xf5，0xf8 到 0xfe 的功能实现。

练习 2：

写程序，把定长编码打印出来

4.2 不确定长度编码

本节主要内容：

1. 掌握 ModR/M 字段
2. 掌握 SIB 字段

老唐语录：

0x88:mov Eb,Gb

0x89:mov Ev,Gv

0x8a:mov Gb,Eb

0x8b:mov Gv,Ev

最短是 2 个字节，Gb（general register byte）代表通用单字节寄存器（AL 到 BH）。Gv 代表通用双/四字节寄存器（早期寄存器是双字节，现在使用四字节）。eb/ev 代表内存地址。

第一个字节是操作码，第二个字节是操作数，又称为 modR/M，格式如下：

7	6	5	4	3	2	1	0
Mod		reg(Gb/Gv)			R/M		

Eb 可以是内存，也可以是寄存器（AL 到 BH），如果指内存，就是指[eax 到 edi]。

Mod=11B，R/M 指寄存器，指令长度是 2 个字节。

Mod=00B，R/M 指内存，指令长度是 2 个字节。

Ev 可以是内存，也可以是寄存器（eax 到 edi），如果指内存，就是指[eax 到 edi]。

例：mov al,bh 的编码是 0x88，0xF8(11 111 000)，如果按 0x8a 来编码可以写成 0x8a，0xc7(11 000 111)。

mov [ecx],bh 的编码是 0x88，0x39(00 111 001)。

mov bh,[edx]的编码是 0x8a，0x3a(00 111 010)。

mov edi,[edx]的编码是 0x8b，0x3a(00 111 010)。

0x8a,0x39 的汇编指令为 mov edi,[ecx](00 111 001)。

mod 为 00 时，eb 不可以是 esp 或 ebp，出现 ebp（rm=5）之后，含义改变，指令长度变为 6 个字节，后添加的四个字节为立即数（d32），内存格式为[d32]。

Mod=01B，R/M 指内存，指令长度为 3 个字节，内存格式为[reg32+d8]。

Mod=10B，R/M 指内存，指令长度为 6 个字节，内存格式为[reg32+d32]。

注：mod 不等于 11B 时，都不允许出现 esp，如果出现，编码格式改变。

在函数内部，[ebp+0]处存储上一层 ebp 的值，将该值取出没有用处（因为上一层的 ebp 不确定性因素存在），所以将该编码换成其他形式。当 R/M=101（ebp）时，该 R/M 被废掉（无效），并在后面加上 4 个字节立即数，内存格式变成[d32]。

如果想继续使用[ebp]的内存格式时，可以使用[reg32+d8/d32]代替。

在函数内部使用变量时，我们通常使用 ebp 加减得到内存数值，而不是用 esp，因为 esp 在浮动（不断变化，比如调用函数时），所以 Mod 不等于 11B 时，将内存格式的 esp（R/M=100B）废掉，变成另一种含义：

如果 R/M 等于 4 时，在 ModR/M 字段后面跟一个字节，称为 SIB，格式如下：

7	6	5	4	3	2	1	0
Scale		Index			Base		

当 R/M=4，Mod=00B 时，指令长度为 3 个字节，内存格式为[Base+Index*2^{scale}]：

操作码	ModR/M			SIB		
-	00	-	101	Scale	Index	Base

当 R/M=4，Mod=01B 时，指令长度为 4 个字节，内存格式为[Base+Index*2^{scale}+d8]：

操作码	ModR/M			SIB			d8
-	01	-	101	Scale	Index	Base	-

当 R/M=4，Mod=10B 时，指令长度为 7 字节，内存格式为[Base+Index*2^{scale}+d32]：

操作码	ModR/M	SIB	d32

-	10	-	101	Scale	Index	Base	-

还有另外两种特殊情况：

1> index=100（esp）时，则被 0 替代，此时 SIB 中只有 Base 有效。

2> Base=101（ebp）时，则被 0 替代，此时 SIB 中只有 Scale 和 Index 有效。

总结：操作码决定有无 ModR/M 字段，ModR/M 字段决定有无 SIB 和 DISP（d8/d32）字段。没有带 ModR/M 字段的指令长度都是定长的。

课后理解：

ModR/M 字段如图 4-1：

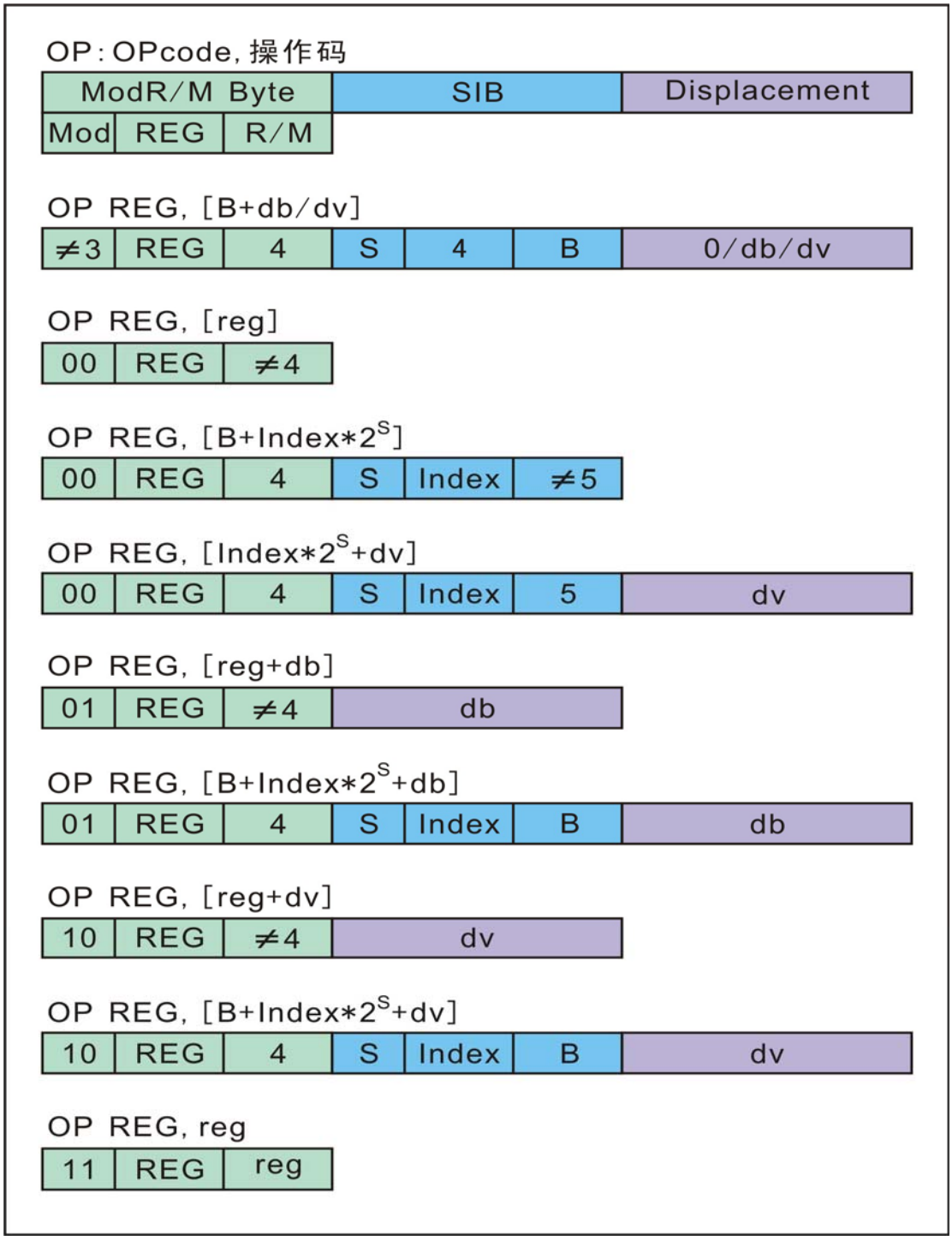


图 4-1：ModR/M 字段

课后疑问：

为什么有些指令有两种编码？

回答：历史遗留问题，早期的硬编码在编写的时候很有规律，也没有重叠，后期随着计

算机宽度的增加和向前兼容性，不得不将指令扩展，毕竟单字节操作码最多只能表示 256 个编码。

课后总结：

指令由指令前缀（Instruction Prefixes，最多加四个，每个 1 字节，不要求顺序）、主操作码（最多 3 个字节）、寻址格式符（addressing-form specifier，视需要而定）、位移（displacement，视需要而定）和立即数。

如图 4-2：

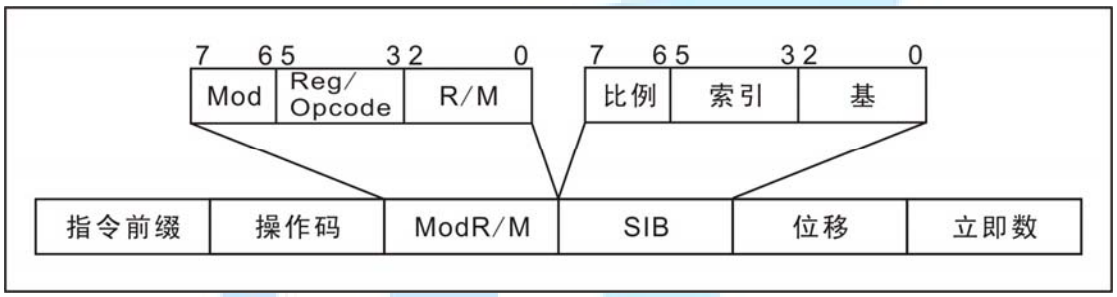


图 4-2：指令格式

课后练习：

1.用实例练习 ModR/M 字段对应的所有编码组合(比如当 R/M 为 4 时的内存寻址方式)。

1>当 mod=1 时，整个指令长度为 3 个字节

```
Eg.{_emit(0x89)}{_emit((1<<6)|(0<<3)|(7<<0))}{_emit(0x89)}:
mov [edi-0x77],eax
```

2>当 mod=2 时，整个指令长度为 6 个字节。

```
Eg.{_emit(0x89)}{_emit((2<<6)|(0<<3)|(7<<0))}{_emit(0x78)}{_emit(0x56)}{_emit(0x34)}{_emit(0x12)}:
mov [edi+0x12345678],eax
```

3>rm=4

```
Eg.{_emit(0x89)}{_emit((0<<6)|(0<<3)|(4<<0))}{_emit((1<<6)|(4<<3)|(1<<0))}:
```

Mov [ecx+ebx*2],eax

{_emit(0x89)}{_emit((1<<6)|(1<<3)|(4<<0))}{_emit((2<<6)|(5<<3)|(2<<0))}{_emit(0x12)}:

Mov [edx+ebp*4+12h],ecx

<1>当 index = 4 即 index=esp 时，esp 被 0 替代

{_emit(0x89)}{_emit((2<<6)|(2<<3)|(4<<0))}{_emit((3<<6)|(4<<3)|(3<<0))}{_emit(0x90)}{_emit(0x78)}{_emit(0x56)}{_emit(0x34)}:

Mov [ebx+34567890h],edx

<2>base=5 时，mov reg,[d32(base 被 d8/d32 代替)+index*2^{scale}]

{_emit(0x8b)}{_emit((00<<6)|(3<<3)|(4<<0))}{_emit((2<<6)|(3<<3)|(5<<0))}{_emit(0x78)}{_emit(0x56)}{_emit(0x34)}{_emit(0x12)}:

Mov ebx,[ebx*4+12345678h]

4.Mod=0&&rm=5

当 Mod=0&&rm=5 时，指令长度为 6 个字节:mov reg,[d32]

Eg.{_emit(0x89)}{_emit((0<<6)|(2<<3)|(5<<0))}{_emit(0x12)}{_emit(0x34)}{_emit(0x56)}{_emit(0x78)}:

Mov [0x78563412],edx

2. 写程序，把变长编码打印出来

4.3 其他指令编码

本节主要内容：

1. 单字节操作码指令
2. 双字节操作码指令

老唐语录：

0xc6:mov Eb,Ib

0xc7:mov Ev,Iv

例：mov eax，0x12345678 中，ModR/M 值为 11 000 000（reg 没用时通常为 0）。有些指令需要操作码和 ModR/M 中的 reg 字段同时确定。比如 xor dword ptr [eax]，0x12345678 与 or dword ptr[eax]，0x12345678 的操作码相同，都是 0x81。以 0x0f 开头则是两个字节操作码。

0x00:add Eb,Gb

0x01:add Ev,Gv

0x02:add Gb,Eb

0x03:add Gv,Ev//0x00 到 0x03 指令的操作数与 0x88 到 0x8b 的操作数一样。

0x04:add al,Ib

0x05:add eax,Id

0x06:push ES

0x07:pop ES

0x08:or Eb,Gb

0x09:or Ev,Gv

0x0a:or Gb,Eb

0x0b:or Gv,Ev

0x0c:or al,Ib

```
0x0d:or eax,Id
0x0e:push CS
0x0f:escape
0x10:adc Eb,Gb
0x11:adc Ev,Gv
0x12:adc Gb,Eb
0x13:adc Gv,Ev
0x14:adc al,Ib
0x15:adc eax,Id
0x16:push SS
0x17:pop SS
0x18:sbb Eb,Gb
0x19:sbb Ev,Gv
0x1a:sbb Gb,Eb
0x1b:sbb Gv,Ev
0x1c:sbb al,Ib
0x1d:sbb eax,Id
0x1e:push DS
0x1f:pop DS
0x20:and Eb,Gb
0x21:and Ev,Gv
0x22:and Gb,Eb
0x23:and Gv,Ev
0x24:and al,Ib
0x25:and eax,Id
0x26:ES(前缀)
0x27:DAA
0x28:sub Eb,Gb
0x29:sub Ev,Gv
```

0x2a:sub Gb,Eb

0x2b:sub Gv,Ev

0x2c:sub al,Ib

0x2d:sub eax,Id

0x2e:CS(前缀)

0x2f:DAS

0x30:xor Eb,Gb

0x31:xor Ev,Gv

0x32:xor Gb,Eb

0x33:xor Gv,Ev

0x34:xor al,Ib

0x35:xor eax,Id

0x36:SS(前缀)

0x37:AAA

0x38:cmp Eb,Gb

0x39:cmp Ev,Gv

0x3a:cmp Gb,Eb

0x3b:cmp Gv,Ev

0x3c:cmp al,Ib

0x3d:cmp eax,Id

0x3e:DS(前缀)

0x3f:AAS

方括号里面（表示内存）含有 ebp 或 esp 的，都是用 SS 做基址（前缀），其他情况下使用 DS。

在串操作数中使用 ES 做基址。比如 movs 指令时，使用 ES:[edi],ds:[esi]。

[EIP]指向的地方是指令，基址是 CS。

inc/dec 编码除了 0x40-0x4f 编码外还有 0xff：inc(0xff,0xc0-0xff,0xc7)；dec：(0xff,0xc8-0xff,0xcf)。

0x80 操作数：Eb,Ib

0x81 操作数：Ev，Iv

0x82 操作数：eb,Ib(含符号扩展)

0x83 操作数：Ev,Ib

0xc0 操作数：Eb,Ib

0xc1 操作数：Ev,Ib

0xd0 操作数：Eb,1

0xd1 操作数：Ev,1

0xd2 操作数：Eb,cl

0xd3 操作数：Ev,cl

0x80-0x83，0xc0-0xc1，d0-d3 操作码格式如表 4-1:

表 4-1：使用 reg 定位指令的操作码

操作码	Mod7,6	ModR/M 中的 reg 字段							
		000	001	010	011	100	101	110	111
80-83	00,01,10,11	add	or	adc	sbb	and	sub	xor	cmp
c0-c1 d0-d3	00,01,10,11	rol	ror	rcl	rcr	shl/sal	shr		sar

0xa0:mov al,byte ptr[d32]

0xa1:mov eax,dword ptr[d32]

0xa2:mov byte ptr[d32],al

0xa3:mov dword ptr[d32],eax

0x84:test Eb,Gb

0x85:test Ev,Gv

0x86:xchg Eb,Gb

0x87:xchg Ev,Gv

0x60:pushad(把 8 个寄存器全部压入堆栈)

0x61:popad(汇编查看，其 pop 的值改变的寄存器)

0x62:bound Gv,ma//ma 指 Mod 不等于 11B，功能实现判断寄存器的值是否在内存地址取出值高四个字节与低四个字节值之间（共取出 8 个字节）。

0x66:段前缀

0x67:数据前缀，前缀可以加多个。d32 会变成 d16。ModR/M 字段将发生改变（16 位模式），没有 SIB 字段，如表 4-2：

表 4-2：16 位宽度下的 ModR/M 字段

R/M	内存格式
0	[bx+si]
1	[bx+di]
2	[bp+si]
3	[bp+di]
4	[si]
5	[di]
6	[bp]
7	[bx]

0xd8-0xdf:浮点指令

0x70-0x7f:jcc jb

0x0f 0x80-0x8f:jcc jv(6 个字节)

0x0f 0x90-0x9f:setcc eb(条件成立，将 eb 置 1，否则置 0)

0x0f 0x40-0x4f:cmovcc Gv,Ev(条件成立，将 Ev 赋给 Gv)

0x0f 0xa0:push FS

0x0f 0xa1:pop FS

0x0f 0xa8:push GS

0x0f 0xa9:pop GS

0xc4:les Gv,mp//mp 表示 6 个字节，高两个字节给 es，低四个字节给目标寄存器

0xc5:lds Gv,mp

0x0f 0xb2:lss Gv,mp

0x0f 0xb4:lfs Gv,mp

0x0f 0xb5:lgs Gv,mp

cpu 执行指令步骤：

1> 取第 1 条指令

2> 分析第 1 条指令，取第 2 条指令

3> 执行第 1 条指令，分析第 2 条指令，取第 2 条指令

4> 执行第 2 条指令，分析第 3 条指令，取第 4 条指令

5> 执行第 3 条指令，分析第 4 条指令，取第 5 条指令

6> 执行第 4 条指令，分析第 5 条指令，取第 6 条指令

7> ……

一旦遇到 Jcc 指令，改变 eip，流水线被迫中断，再重新开始。可以使用 setcc 消除条件跳转。

课后理解：

```
Push   ebp => 0x55 01010 101 B  opcode&0xf8 == 0x50 push
                   A    5（寄存器号）
Pop    ebp => 0x5D 01011 101 B  opcode&0xf8 == 0x58 pop
                   B    5
Inc    ebp => 0x45 01000 101 B  opcode&0xf8 == 0x40 inc
                   8    5
dec    ebp => 0x4D 01001 101 B  opcode&0xf8 == 0x48 dec
                   9    5
```

得出：

Opcode&0xf8 = 0xb8 ： 指令长度为 5 个字节

Opcode&0xf8 = 0xb0 ： 指令长度为 2 个字节

课后疑问：

为什么在 VC6 下写许多语法正确的汇编指令会报错（比如 jmp Ap 指令）？

回答：VC6 不支持，可以使用 OD 或 IDA 等工具。

课后总结：

Intel 指令有成千上万条，平时常用的只有几百条。每个编译器对指令的支持不同，

在编译器不支持的情况下，只有通过硬编码查看，所以要熟记常用指令编码。

课后练习：

写一个反汇编引擎。

第五章 保护模式

引言

前面我们学过 C 语言、汇编、硬编码。今天我们来学习 CPU 的内部结构（段页）。

我们为什么要学习段页？

学了段页就是为了了解 CPU 的架构，一条指令是怎么运行的，内存的是怎么分布的，这一章实际上是为我们以后学习操作系统奠定基础。

什么才是正确的学习方法？

学习段页实际没有更好的捷径，关键是记的东西很多，什么是正确的方法，多抄多记多练习，按照我们每节讲的内容走。段页比较不容易理解，如果有什么不懂的，可以去滴水论坛提问 www.dtdebug.com。

本章必须要掌握的知识点：

1. 段寄存器及其结构
2. 段寄存器的读写
3. 改变 CS 的指令
4. 同时改变 CS EIP 的指令
5. 什么是门，门描述符结构
6. 调用门
7. 中断门，陷阱门
8. 任务段（TSS）
9. 任务门
10. 10-10-12 分页方式
11. 2-9-9-12 分页方式
12. TLB
13. CACHE
14. CR3 和 CR4 切换

5.1 段寄存器结构 1

本节主要内容：

1. 段寄存器及其结构
2. 探测段寄存器结构

老唐语录：

今天讲段，以前我们讲有 eax 到 edi 和 eflag、eip 共十个寄存器。这十个寄存器是 32 位，并且可以拆分成八位、十六位。读指令和寻址方式都是靠这 8 个寄存器工作，eip 是取指令，eflag 是执行指令中的标志位。方括号里面的值指的是内存单元，也是有效地址。当初这么说是不对的，cpu 里面有段寄存器作为基址，而基址一般是 0。

段寄存器有八个，也要记住顺序：ES、CS、SS、DS、FS、GS、LDTR、TR。

ES：扩展段。在串操作时（比如 cmovs）目标操作数的基址是 ES，源操作数是 DS。

CS：代码段，配合 EIP 使用。

SS：堆栈段，**凡是基址是 EBP 或 ESP 的，段前缀就是 SS**。

DS：数据段，默认的都是 DS。

FS、GS：80386 之后定义的。

计算机上的数是由宽度的，没有宽度的数没有意义。**段寄存器的宽度是 96 位**。使用结构体表示：

```
struct segment
{
  word Selector;/*十六位。当读段寄存器（如 mov ax,CS）的时候，只会返回这十六位。或者 push seg 操作针对的都是这十六位。如果目标操作数是 32 位（如 mov eax,CS），则将 16 位零扩展成 32 位赋给目标操作数。但是写的时候，就会涉及到 96 位*/
  word Attributes;//16 位
  dword Base;//32 位
  dword limit;//32 位
```

```
}
```

因为我们执行读操作的时候只能得到 16 位（selector），如何知道他是 96 位的？我们来探测并证明段寄存器确实是 96 位的。

```
int Gvar;
__asm
{
  mov AX,SS//当把 SS 修改为 CS 后，程序无法正常运行
  mov DS,AX
  mov Gvar,EAX
}
```

说明：上例说明，段除了 selector 之外，还有一个成员“属性（Attributes）”。表示这个段是可读，可写，还是可执行：可读表示可作为源操作数，可写表示可作为目标操作数，可执行表示可以使用 EIP 指向。Gvar 是个全局变量，默认情况下方括号里面是个立即数，编码的时候，Mod 字段是 00，R/M 等于 5，默认的寻址方式是 DS 作为前缀（DS:[]）。DS 本来是可访问的，SS 的属性被搬到 DS 里面（mov AX,SS mov DS,AX）。SS 的属性也是可读、可写、可执行。但是 CS 是不可写的（可读、可执行），所以程序会出错（CS 崩掉），但是反过来写不会：

```
__asm
{
  mov AX,CS
  mov DS,AX
  mov EAX,Gvar
  mov AX,ES//恢复 DS 的值防止程序崩溃
  mov DS,AX
}
```

段寄存器的选择子是可见的：

```
ES：0x32
CS：0x1B
SS：0x23
```

DS：0x23

FS：0x3B

GS：0

0x1B 对应段（CS）的属性是可读可执行不可写。

下面探测 Base 的存在性：

```
int Gvar;
__asm
{
  mov AX,FS//如果将 FS 改成 DS，观察程序是否正常执行
  mov GS,AX
  mov EAX,GS:[0]
  mov Gvar,EAX
}
```

为什么 GS 从 0 的位置也可以读出内容？说明无论哪一个段做前缀，[]都要加上 SEG.Base 才是真正的内存。FS 的 Base 不是 0，是个很大的值（使用 OD 查看）。

```
int Gvar1;
__asm
{
  mov EDX,DS:[0x7FFDF000]//方括号里填写 FS 基址。不同电脑的 FS 的基址不一定相同
  mov Gvar1,EDX
}
```

则 Gvar1 和 Gvar 的值相同。

注意：VC 有 BUG，程序不一定编译正确，可以使用其他工具（如 OD）做参考。

每个段寄存器的基址如表 5-1：

表 5-1：段寄存器结构内容

	基址	界限	属性	选择子
ES	0	0xffffFFFF	可读可写	0x23

CS	0	0xffffFFFF	可读可执行	0x1B
SS	0	0xffffFFFF	可读可写	0x23
DS	0	0xffffFFFF	可读可写	0x23
FS	不确定	0xFFF		0x38
GS	-	-		0

下面我们探测 Limit 的存在性：

```
int Gvar;
__asm
{
  mov AX,FS
  mov GS,AX
  mov EDX,DS:[0x7FFDF000+0x1000]
  mov EAX,GS:[0x1000]
  mov Gvar,EAX
}
```

在上例中，将 FS 赋给 GS，则 GS 就相当于 FS。GS:[0x1000]与 DS:[0x7FFDE000+0x1000]的地址相同，但是 mov EAX,GS:[0x1000]会出错。

注：有的测试程序不能调试，只能运行。

说明：一个段的 Base 表示这个段的基址从哪里开始可以访问。Limit（界限）表示[]里面的值最大允许多少。FS 的界限是 0xFFF，[0x1000]超出范围，所以程序出错，FS 的访问范围是从 0x7FFDF000-0x7FFDFFFF。DS 的访问地址是从 0-0xFFFFffff。如图 5-1：

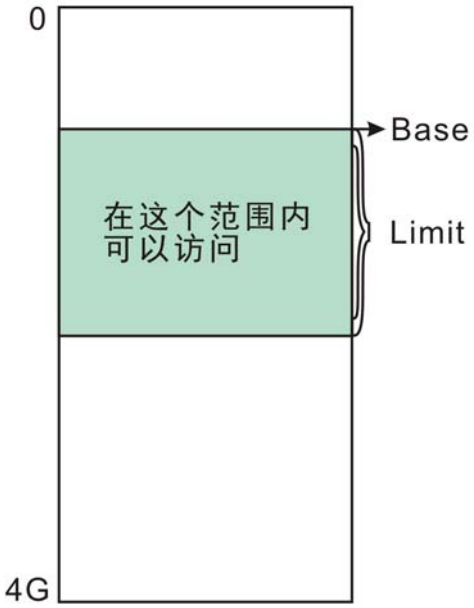


图 5-1：段基址与段界限

我们继续探测 limit：

```
int Gvar;
__asm
{
  mov AX,FS
  mov GS,AX
  mov AL,GS:[0xFFD]
}
```

1> 观察语句执行结果

2> 将 AL 改为 AX，观察执行结果.

3> 将 AL 改为 EAX，观察执行结果

说明：当访问 word 长度时，要保证每个字节都处于 limit 的范围内，所以从 0xFFD 读一个字节是位置 0xFFD,读两个字节是位置 0xFFD 和 0xFFE,读三个字节是位置 0xFFD、0xFFE 和 0xFFF，读四个字节是位置 0xFFD、0xFFE、0xFFF 和 0x1000。0x1000 超出界限，所以出错。

4> 将 0xFFD 改为 0xFFF（mov AL,GS:[0xFFF]），观察执行结果

5> 将 AL 改为 AX（mov AX,GS:[0xFFF]），观察执行结果

总结：0x1B 的属性是不可读不可写，所以赋给其任意一个段寄存器都会导致该段无法读写。

段寄存器的结构体位置到底在哪里？

《80x86 汇编语言设计》中有描述（后面章节会详述）。

为什么只将选择子赋给段，而段的属性却变了？

当选择子赋给段的时候，cpu 会去找原来的 96 位放在内存中的位置，然后修改为当前段选择子对应的 96 位结构体值。

除了 8 个段寄存器外，还要记住 GDTR，IDTR，这两个寄存器是 48 位，结构体如下：

```
struct GDTR_T
{
  dword Base;//高位
  word Limit;//低位
}
```

课后理解：

当一个段的选择子被加载到段寄存器的可视部分，处理器同样加载其他隐藏部分，这样处理器不需要再通过总线读段描述符来传输地址。

例 1. 通过 Mov eax,[ebx] = mov eax,[ds.base+ebx]

Mov ax,es = mov ax,es.selector，解释下列程序。

```
Mov ebx,0xffd
Mov eax,fs:[ebx]
```

说明：

```
Mov ebx,0xffd
Mov al,[fs.base+0xffd]
Mov ah,[fs.base+0xffe]
Mov eax>>16,[fs.base+0xfff]
Mov eax>>24,[fs.base+0x1000]//通过 OD 工具可以直观看出 fs 的 limit
```

课后总结：

段寄存器的宽度是 96 位。

课后练习：

将十个寄存器记熟。

5.2 段寄存器结构 2

本节主要内容：

存储段描述符表（GDTR）

老唐语录：

上节课我们要求记住段寄存器的每个字段，现在要求记住字段中的每一位：

selector（16 位）是个整体，不需要记，attributes（16 位）中每一位都不一样，limit（32 位）和 Base（32 位）都是一个整体。

往段寄存器里面读（如 mov AX,CS）只读取 16 位（selector），往段寄存器里面写（如 mov CS,0x1B）虽然只传入 16 位，却修改了 96 位。

想一想：16 位的值怎么会变成 96 位？

其实学习段页，要懂 cpu 的历史。不懂编码也不会真正的理解段页。编码在 8086 的时候就存在了，**在 8086 时代，段寄存器只有 16 位（selector）**。在 80286 时代，段寄存器是（selector：16 位，attributes：16 位，base：24 位，limit：16 位）。80386 之后，才发展成 96 位。软件绑架了 CPU，软件要兼容，指令格式要兼容，所以接口要稳定，不能变。所以当确定了原操作数是 16 位，目标操作数是 96 位时，如何执行？

Intel：针对 mov DS,AX，只是将 AX 的值赋给了 DS.Selector，然后 cpu 内部（软件不知道），GDTR 寄存器（48 位）有两个成员：Limit 和 Base。Base 指向内存的某个地方（**GDT 表**首地址），limit 表示这段内存有多长。再使用 AX 低三位清零，加上 GDTR.Base，得到一个内存地址，并在这该内存地址处取 8 个字节，把这八个字节赋给 CS 的其他位（Base、limit 和 attributes）。如图 5-2 所示：

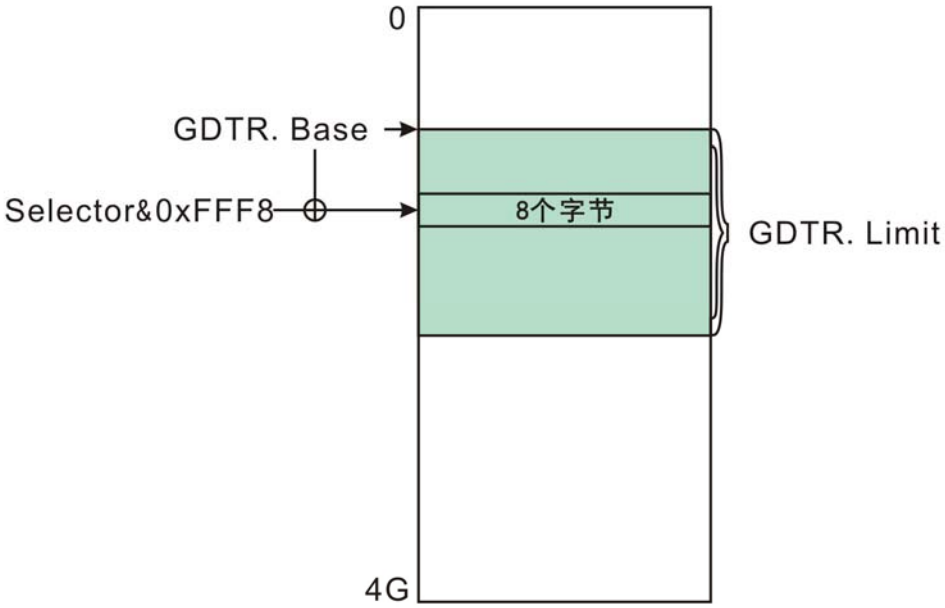


图 5-2：GDTR+Selector 定位段寄存器结构内容

补充：获取 GDTR 值的汇编指令为“SGDT mem”，写入 GDTR 的指令为 LGDT。

这 8 个字节的定义见图 5-3（见《80x86 汇编程序设计》366 页）：

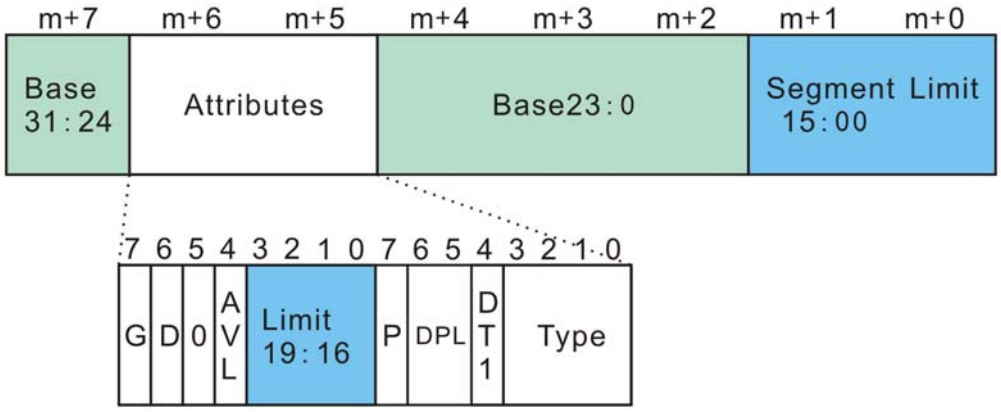


图 5-3：存储段描述符格式

limit 只有 20 个位，如何赋给 32 位？

当属性里的 G 位为 0 时，在 limit 高位添加 12 个 0，扩展为 32 位；当 G 位为 1 时，在 limit 低位添加 12 个 1（二进制），扩展成 32 位，如图 5-4:

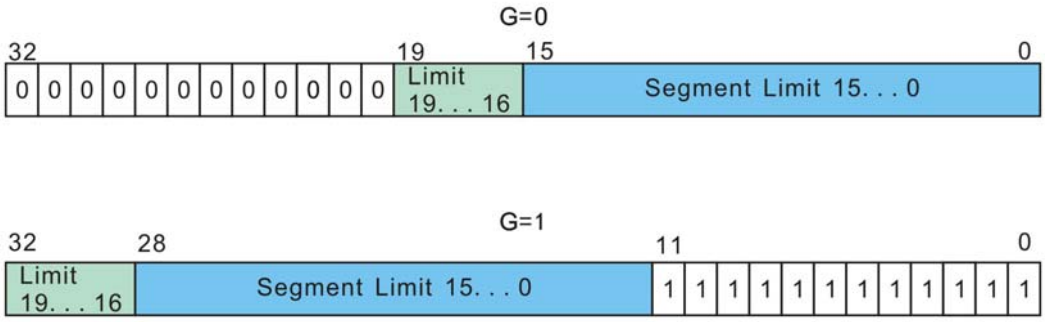


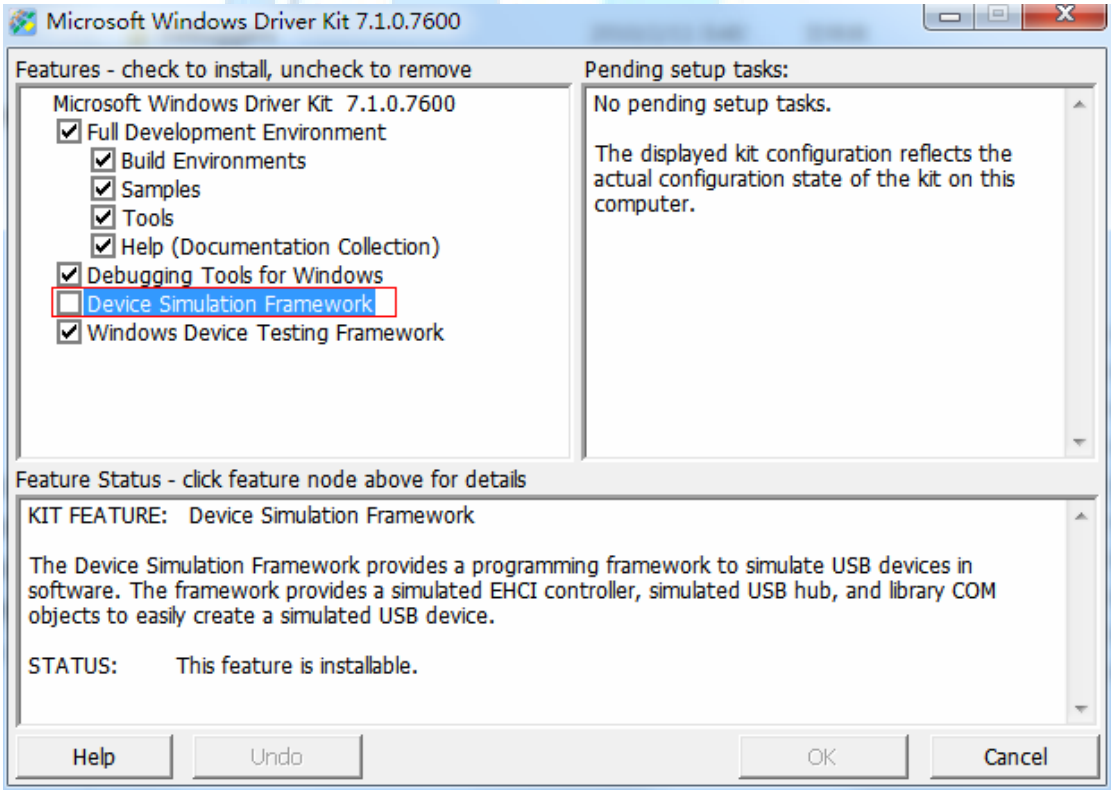
图 5-4：limit 扩展

安装和使用 windbg 使用步骤如下：

1）下载 windbg，网址：

http://download.microsoft.com/download/4/A/2/4A25C7D5-EFBE-4182-B6A9-AE6850409A78/GRMWDK_EN_7600_1.ISO

2）安装 KitSetup.exe 如图，针对 ghost 系统，Device Simulation Framework 不要勾选（出现重启后系统崩溃）。



3）将系统设置为 Debug 模式：

xp 系统：打开 C:\boot.ini（系统保护和隐藏文件），在尾端添加/debug，例如：

multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows XP" /execute=optin /fastdetect /debug

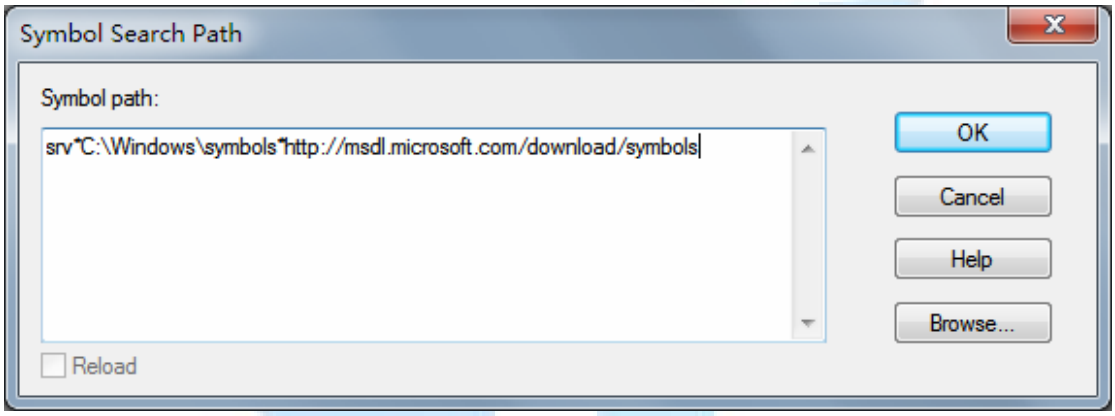
win7 系统：打开 cmd.exe 执行 bcdedit /debug ON（管理员权限下执行）

4）重启电脑

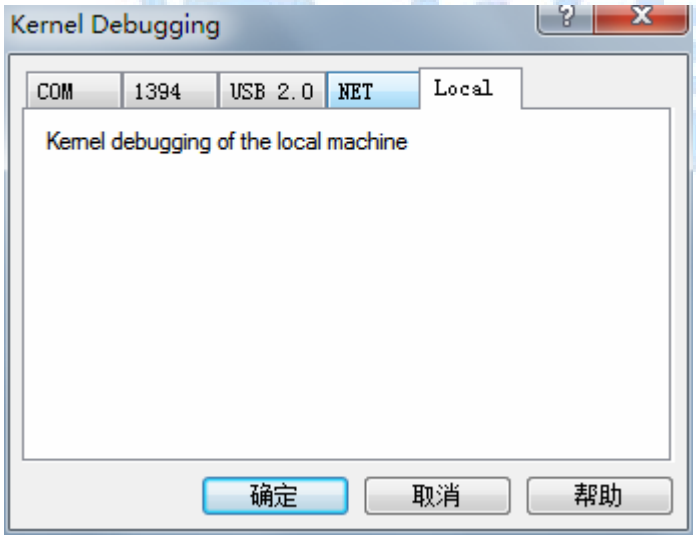
5）打开菜单-> Debugging Tools for Windows (x86)->WinDbg

6）WinDbg 打开后，添加符号文件：File->Symbol File Path.....弹出 Symbol Search Path 对话框，在 Symbol Path 下输入以下字符，并点击“OK”：

srv*C:\Windows\symbols*http://msdl.microsoft.com/download/symbols



7）打开本地内核调试：File->Kernel，弹出 Kernel Debugging 对话框，选择 Local 选项卡，点击“确定”。



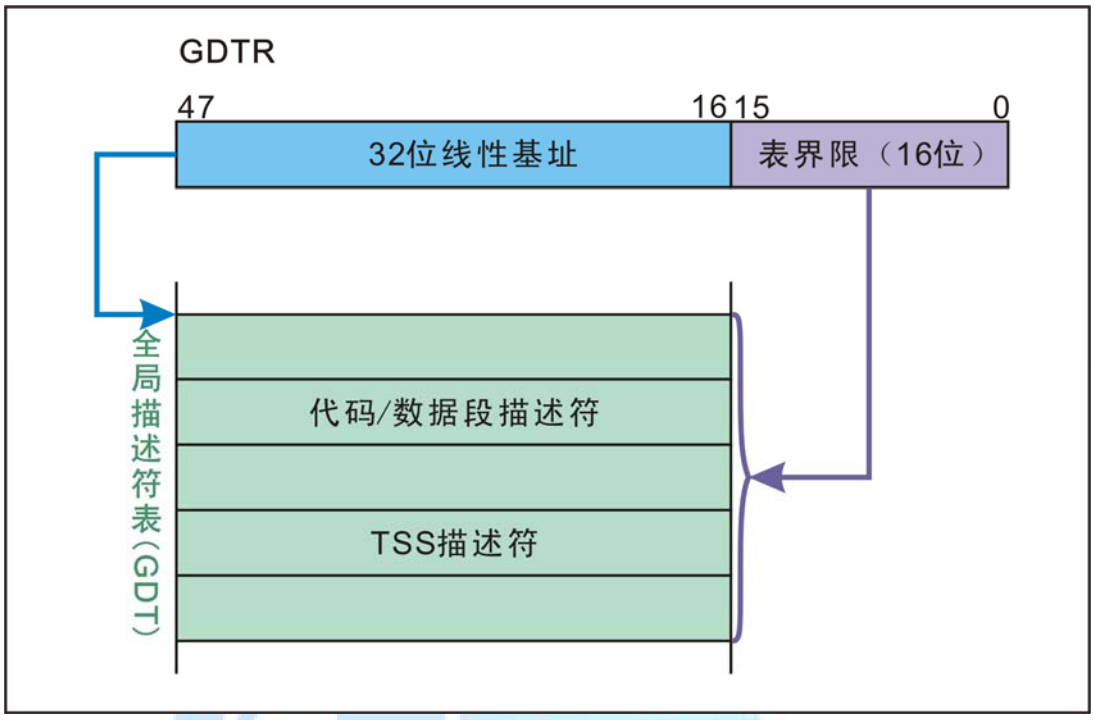
8）在 lkd>输入.reload 确认符号已经加载。然后输入!pcr 命令得到 GDT 表位置（推荐使用本节知识找到 GDT 表）：

```
lkd> !pcr
KPCR for Processor 0 at 83d7ac00:
... ...
IDT: 80b95400
GDT: 80b95000
TSS: 801e3000
```

9)输入指令 dq 0x80b95000 l10（指令解释：display QWORD 0x10 line：八个字节一组，共显示 16 组）查看 GDT 表内容：

```
lkd> dq 0x80b95000 l10
80b95000  00000000`00000000 00cf9b00`0000ffff
80b95010  00cf9300`0000ffff 00cffb00`0000ffff
80b95020  00cff300`0000ffff 80008b1e`300020ab
80b95030  834093d7`ac003748 0040f300`00000fff
80b95040  0000f200`0400ffff 00000000`00000000
80b95050  830089d7`80000068 830089d7`80680068
80b95060  00000000`00000000 00000000`00000000
80b95070  800092b9`500003ff 00000000`00000000
```

课后理解：



课后疑问：

为什么 Base 被分成两段？

回答：历史遗留原因。因为 80286 使用的 Base 是 24 位，386 扩展成 32 位，格式不能变，只能使用后面的保留位。

课后总结：

段描述符的 8 个字节赋给 96 位段寄存器。

课后练习：

安装 windbg，记住 GDT 中的每 8 个字节。

5.3 段寄存器结构 3

本节主要内容：

段寄存器中的属性（attributes）字段

老唐语录：

0x23（SS、DS、ES）的属性是读写，界限为-1（0xFFFFffff）。只有 FS 的基址通常不是 0，界限永远是 0xFFF。CS 可读可执行不可写。type 类型描述如表 5-2：

表 5-2：存储段描述符类型

类型域					描述
十进制	**数据段**				
		E	W	A	
0	0	0	0	0	只读
1	0	0	0	1	只读，可访问
2	0	0	1	0	可读，可写
3	0	0	1	1	可读，可写，可访问
4	0	1	0	0	只读，向低扩展
5	0	1	0	1	只读，向低扩展，可访问
6	0	1	1	0	可读，可写，向低扩展
7	0	1	1	1	可读，可写，向低扩展，可访问
	代码段				
		C	R	A	
8	1	0	0	0	只执行
9	1	0	0	1	只执行，可访问
10	1	0	1	0	可执行，可读
11	1	0	1	1	可执行，可读，可访问

12	1	1	0	0	只执行，一致码段
13	1	1	0	1	只执行，一致码段，可访问
14	1	1	1	0	可执行，只读，一致码段
15	1	1	1	1	可执行，只读，一致码段，可访问

表 5-2 中，可能不太理解向低扩展的含义。我们现在讲一下环帮助理解（凡是涉及段寄存器的低两位均指段选择子 selector 的低两位）。**CS 的低两位（CPL）**表示环：值为 0 表示 0 环；值为 3 表示 3 环。在保护模式下，**SS 的低两位和 CS 的低两位一定相同。**FS、DS、GS、ES 的低两位在数值上一定要大于等于 CS 低两位（权限小于等于 CS）。凡是值为 0（selector 的高 14 位为 0）视为无效，比如 GS，GDT 表里面的第一个 8 个字节全为 0，因为不会被使用。

段寄存器里的低两位（又称 **RPL**）可以理解成描述符里面的 **DPL**（其实含义不同，现在不要求理解）。

attributes 中的 P 位：如果 P 位为 0，表示描述符无效。

DT 值为 1。

TYPE 类型如表 5-2。

AVL 暂时没有使用。

G 位控制 limit。

D 位表示段是 32 位还是 16 位，无论是 0 还是 1，对 limit 和 Base 都没有影响。只对 SS 和 DS 有意义：如果指的是 CS 的 D 位，值为 1，[]里面的寻址方式是 REG32+REG32*(1,2,4,8)+D32；值为 0，[]里面的寻址方式是 SI+DI。如果指的是 SS 的 D 位，值为 1，push eax 可以转换成 lea esp,[esp-4];mov [esp],eax；值为 0，push eax 可以转换成 lea sp,[sp-4];mov dword ptr[sp],eax。比如：如果 esp=0，D 为 1 时（32 位），esp-4 = 0xFFFFfffC；D 为 0 时（16 位），sp-4 = 0xFFFC，零扩展为 32 位值是 0x0000FFFC。

向低扩展（向下扩展）：

以 FS 为例，如图 5-4：

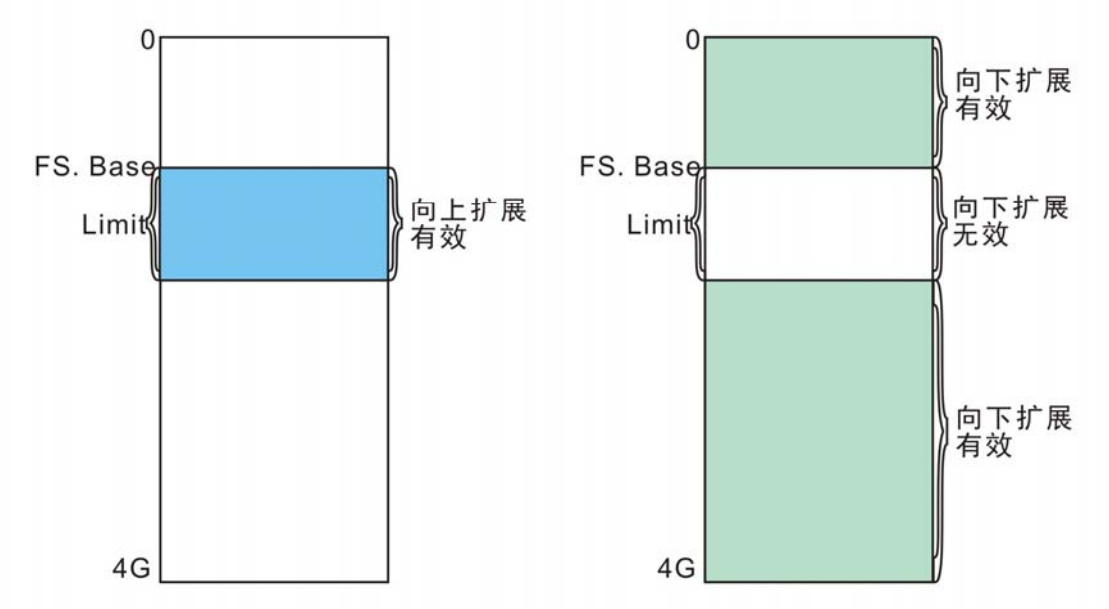


图 5-4：向下扩展

另外，向下扩展和 D 位有关：

D=1 时，最高位是 4G，这里的最高位指从 Base 开始偏移 4G，如图 5-5：

D=0 时，最高位是 64K，这里的最高位指从 Base 开始偏移 64K，如图 5-5：

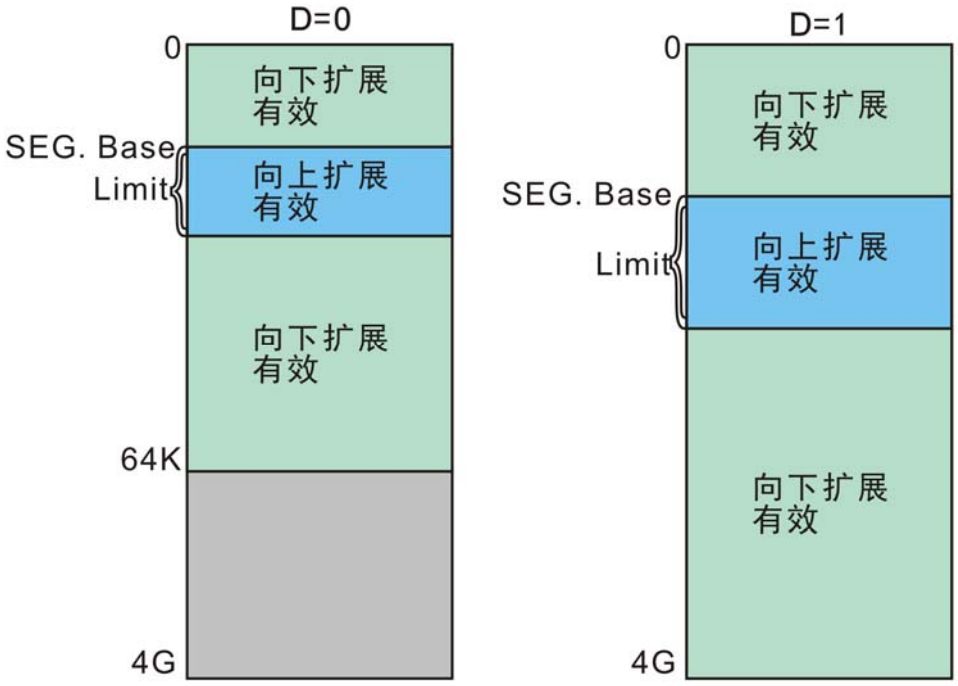


图 5-5：涉及 D 位的向下扩展

向下扩展就是指反向内存有效；D 位只是控制方括号[]里面的值。

前面提到，在针对寻找段寄存器对应的段描述符时，使用 selector 低三位清零

+GDT.Base。其中要求对 selector 低三位清零是因为 selector 的低两位表示权限（第 0 位和第 1 位），第 2 位是 TI 位，值为 0（值为 1 表示访问另一张表，windows 没有使用）。如图 5-6：

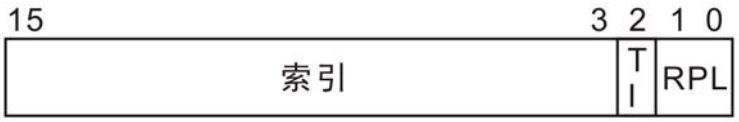


图 5-6：段选择子

```
mov AX,FS//mov AX,FS.Selector
push FS//push FS.Selector
```

mov FS,AX//将 AX 的低三位清零+GDTR.Base 得到地址取出 8 个字节赋给 FS，cpu 会检测 AX 值的合法性，8 个字节中 P 位是否为 0，DPL 值是否小于 RPL

练习：

1> 将 AX=0x200 赋给段寄存器，观察执行结果。

2> 将 AX=0x10 赋给段寄存器，观察执行结果。

DPL 在数值上一定要大于等于选择子的低两位（RPL）。

0 可以赋给段寄存器：表示清空。

0 不能赋给 SS，为什么？

回答：SS 和 CS 的低两位必须一致。

练习：

判断下列语句是否执行

```
mov AX,0x1B
mov SS,AX
```

说明：CS 的属性一定要可执行，SS 的属性一定要可读可写。在三环对 SS 赋 0x23，在 0 环赋 0x10.windows 的常见值：0x08，0x10，0x18，0x1B，0x23，0x30，0x3B。

三环 0x23，0x1B，0 环 0x08，0x10。为什么一个环有两个值？

回答：因为 SS 和 CS 的低两位必须一致，而属性又必须不同（CS 可读可执行，SS 可读可写），所以 0 环的 0x10 是给 SS 使用，0x08 给 CS 使用。三环的 0x1B 是给 CS 使用，0x23 给 SS 使用。其他段如 DS 或 ES 在三环和 0 环都可以 0x23。

mov CS,AX

为什么不能给 CS 赋值？

因为真正取指令的时候是 CS 的基址加上 EIP 取指令，并且取完当前指令后，马上取下一条，CS 一旦基址被改变，EIP 并没有改变（只是加上本条指令长度，偏移到下一条指令首地址），所以无法定位下一条指令，要想改变 CS，就要找到一条同时改变 CS 和 EIP 的指令：只改变 EIP 的指令：Jcc,e8,e9,,eb,e0,e1,e2,e3,c2,c3；同时改变 EIP 和 CS 的指令 0xCF：retf，首先 pop 四个字节给 EIP，再 pop 四个字节给 CS。

练习：

```
push 0x1B
push 0x401000
retf
```

也可以在 GDT 中空闲处手动写入和 0x1B 相同的 8 个字节，替换 0x1B（使用 windbg 工具，指令：eb，ew，ed，eq）。

iretd 也可以同时修改 CS 和 EIP:

练习：

```
pushfd
push 0x1B
push 0x401000
iretd
```

另外，jmp EP 和 call EP 也可以同时修改 CS 和 EIP。

课后理解：

B8(指 IDT 或 GDT)的首地址为 GDTR.base + gs.selector(由 ax 赋给)&0xFFF8。

```
GS.base = B8.base
GS.limit = B8.limit(G=0)或 B8.limit<<12|0xfff(G=1)
GS.attribute = B8.attribute
GS.selector = ax
```

课后总结：

SS 的低两位和 CS 的低两位一定相同。

课后练习：

解析 0x3B 对应的段寄存器值。

当 push 值为 0x3b 时，由 SGDT 指令访问 GDTR 值为 0x8003f00003ff

GDT 首地址 0x8003f000+0x3b&0xfff8 = 0x8003f038,GDT 的长度为 0x3ff

由 windebug-内核调试-local 指令 lkd>db 0x8003f038 得出 GDT 为

7f40f3fd`d0000fff

GS.base = 0x7ffdd000

GS.limit = 0xfff(G = 0)

GS.attribute = 0x40f3

GS.selector = 0x3b

由此得出 DT = 1，为段描述符。GS 的 type 类型为 3（读/写、已访问）不可执行。

5.4 段权限

本节主要内容：

1. CPL、DPL 和 RPL
2. CS 赋值指令

老唐语录：

1. TI（selector 中的第二位）不可以为 1。
2. 如果 selector 高 13 位为 0，表示空选择子，可以赋给段寄存器（CS、SS 除外）。
3. 选择子不能太大（不超过 GDTR.Limit）。

如图 5-7 所示：如果 Base= 0x1000,limit=0x200,ax=0x5000;AX 越界了。

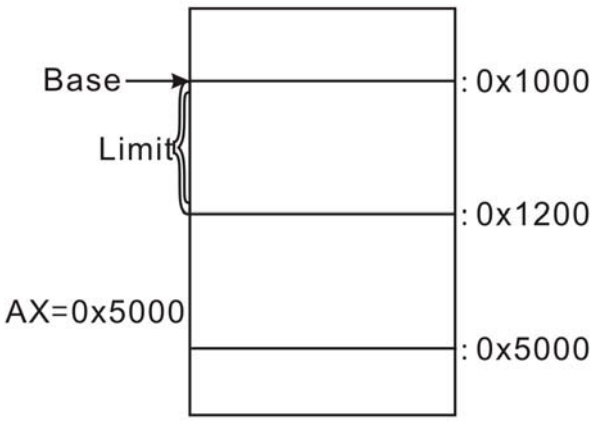


图 5-7：段界限

4. 8 个字节描述符中 P 位为 0，表示无效，不能赋给段寄存器。
5. DT 为 0，无法赋给段寄存器。
6. CPL（CS 低两位）权限要大于等于 DPL，RPL 权限要大于等于 DPL。
7. 如果给 SS 赋值，则 CPL 一定要等于 DPL，并且 TYPE 必须是可读可写。

提问：Base 是否可以等于 0xFFFFffff？

回答：可以，首地址是 0xFFFFffff，访问的下一位是 0x00000000。假设 FS.Base = 0xFFFFffff。则 mov EAX,FS:[0]将 0xFFFFffff 位置赋给 AL，0x0 赋给 AH...

给段寄存器赋值的指令：LES，LSS，LDS，LFS，LGS，但是没有 LCS。

练习：

在 OD 中给段寄存器赋值，观察。可以使用 0x08，0x10，0x1B，0x23，0x3B，也可以使用空闲区自己构建 8 个字节赋给段寄存器（使用 WinDbg，并且在真实机上实验，而非虚拟机）。

注：每个 cpu 都有一个 GDT 表，可以配置成单个 cpu 或修改所有位置。

给 CS 赋值的指令有哪些？

iretd（0xCF），retf（0xCA，0xCB），jmp far（0xEA），call far（0x9A），0xFF（CALLF Ep，JMPF Ep）。

练习：

使用上述指令修改 CS。

说明：

```
__asm
{
  pushfd
  push 0x1B//修改为 0x08，是否执行
  push 0x401000
  iretd
}
```

课后理解：

LTR ax**//修改 TR，没有权限执行**

Selector 的 0，1 位为权限 0-3 级，2 位为 TI 位，TI 为 0，表示从 GDT 中读取描述符，TI 为 1，表示从 LDT 中读取。

通过 selector,GDTR->GDT 中的 TR->TR.base(TSS 的首地址)->TSS。

可以在 winDebug 中使用 lkd>db TR.base 查看 TSS，共 108 个 byte。也可以通

过 eb 指令修改和设置内容，将 selector 的内容指向该段内容，可以做到修改段寄存器的内容。

课后总结：

段寄存器赋值会检查权限、位置等许多规则。

课后练习：

将 GDTR.base+0xc0 存储的段内容修改为 GDTR.base+0x38 存储的内容，然后执行：

```
Mov eax,0xc3
Mov es,ax
Mov ecx,0x100
```

5.5 调用门

本节主要内容：

1. 门的定义
2. 门描述符结构
3. 调用门

老唐语录

因为选择子和 eip 是操作数，所以都可以由用户指定。比如指令 mov EAX,0x90909090，如果用户执行修改 CS 的指令后跳至该指令中间位置（比如 0x90），则会导致系统蓝屏，所以 OS 规定修改 CS 只能在当前权限执行，不能跳至其他权限，如果想进入其他权限，则需要通过门进入（不能翻墙）。

切换权限时，JMP/CALL CS:EIP 只有前三个字节有效（选择子有用），后 4 个字节失效，EIP 要指定（因为要通过门），所以要使用新的段描述符，其中 limit 和 attributes 以及 EIP 都要指定。问题：段描述符已经不够使用（limit 使用 G 位扩展），于是再加上 8 个字节，共 16 个字节，为了不改变之前段描述符表（GDT）的格局，使用指针指向另一位置（selector 变成指针）。

DT=0 时，表示为门描述符（《80x86 汇编语言程序设计》P391 页），offset 可以看作 EIP，如图 5-7：

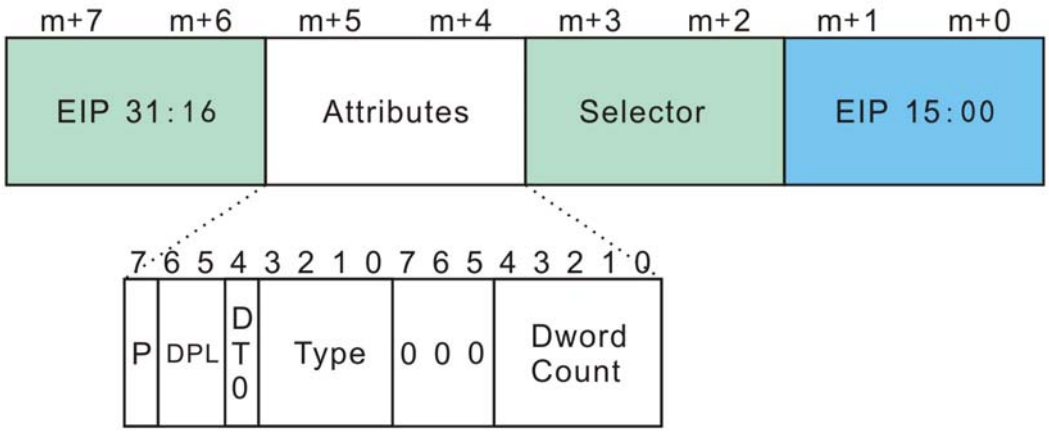


图 5-7：门描述符

练习：自己构建一个调用门（selector 可以填成 0x1B，也可以自己重新构造）。

总结：**调用门的目的就是废掉指令里面的 EIP**。

LDTR 和 TR 也是 96 位段寄存器。也遵循段格式，但不能使用 mov 指令赋值或读，也不可作为段跨越前缀（其他段寄存器可以），称为**系统段寄存器**。可以使用 SLDT/STR 指令读，比如 SLDT AX；STR AX。

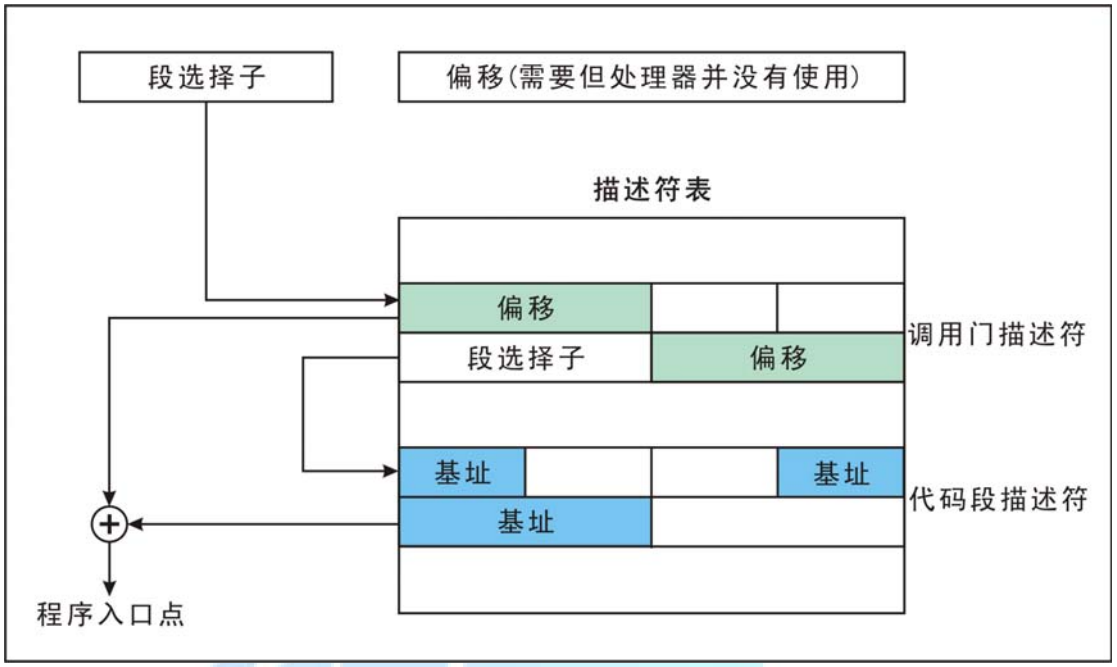
既然可以读，那么也可以赋值：指令为 LLDT，LTR。

系统段寄存器读取的段描述符成为**系统段描述符**，格式与门描述符相同，通过 TYPE 区别，TYPE 类型如下表：

表 5-3：系统段和门描述符 TYPE

	系统段和门描述符				
0	0	0	0	0	未定义
2	0	0	1	0	LDT
5	0	1	0	1	任务门
9	1	0	0	1	可用的 TSS
B	1	0	1	1	忙的 TSS（正在被系统使用）
C	1	1	0	0	调用门
E	1	1	1	0	中断门
F	1	1	1	1	陷阱门

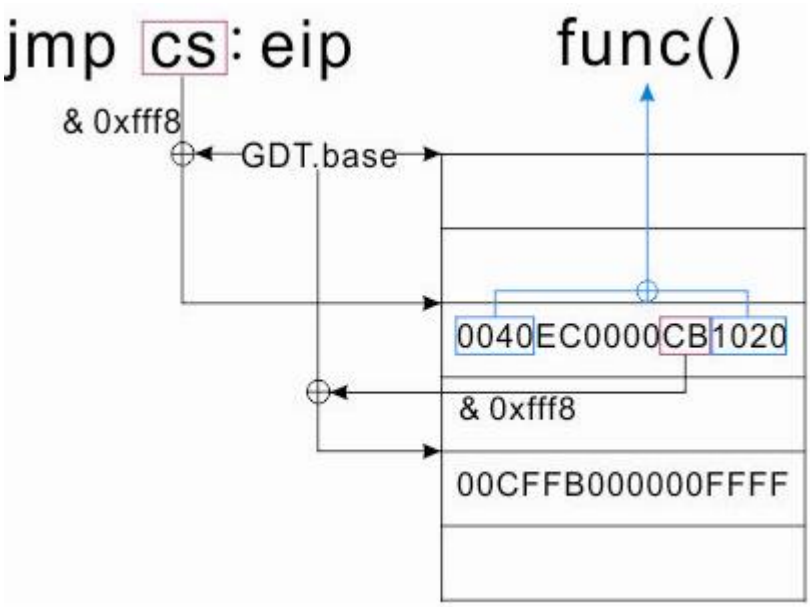
课后理解：




```
int func()//该处首地址为0x401020
{
    int a =1,b=2;
    return (a+b);
}
int main(int argc, char* argv[])
{
    char buff2[6];
    *(WORD*)&buff2[4] = 0xc3;
    __asm
    {
        Jmp fword ptr [buff2]
    }
    return 0;
}
```

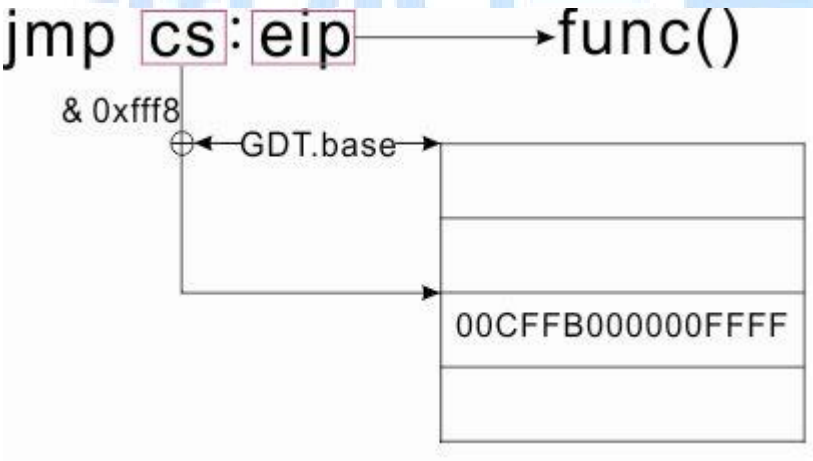
欲使 Jmp/call fword ptr [buff2]指令执行后跳转到 func 函数：

解<1>使用 windebug 将不用的 GDT 段修改为门描述符的结构，如下



解<2>

```
char buff[6];
*(DWORD*)&buff[0] = 0x401020;
*(WORD*)&buff[4] = 0x1b;
__asm
{
   call fword ptr[buff]
}
```



注：其中的 jmp/call 指令可以为 jmp 0xc3:0x401020,VC6 不支持该种显示格式，可用硬编码实现：

```
_emit(0xea)
_emit(0x10)
_emit(0x20)
_emit(0x30)
_emit(0x40)
_emit(0xc3)
_emit(0x00)
```

课后总结：

调用门的目的就是废掉指令里面的 EIP。

课后练习：

记住 GDT、IDT 表。

5.6 其他门描述符

本节主要内容：

1. 中断门、陷阱门和任务门
2. LDT 表
3. TSS 表

老唐语录：

中断门和陷阱门与调用门一样，也是忽略指令里面的 eip，放在 GDT 表里面叫做调用门，放在 IDT 表里面是中断门、陷阱门（GDTR 指向 GDT 表，IDTR 指向 IDT 表）。

windows 没有使用调用门。

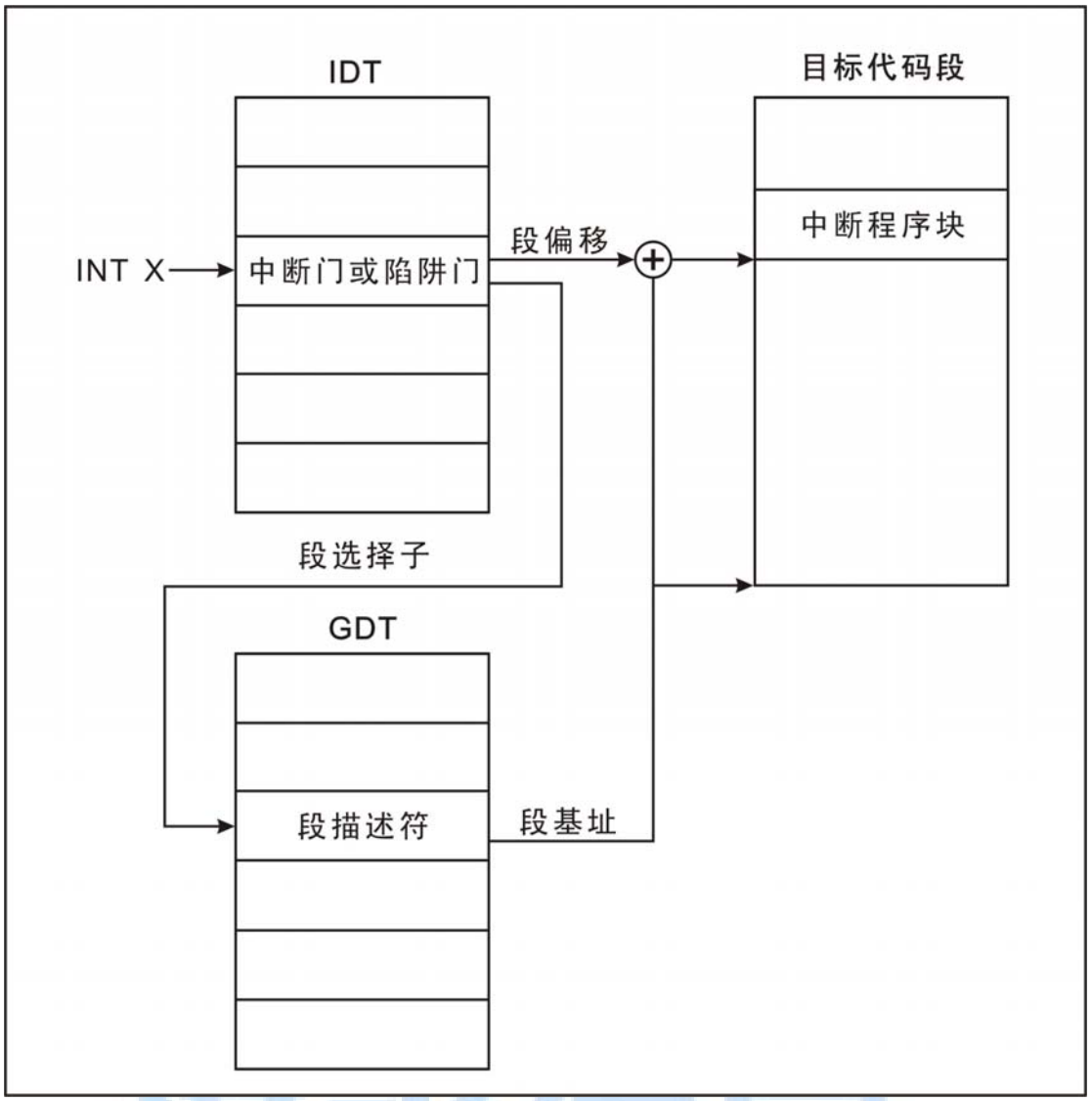
凡是使用 call、jmp 访问的是调用门。中断门单独使用一条指令 int ib。ib（一个字节立即数）表示要访问中断描述符表里面的第几个门描述符：ib=1 表示访问第一个 8 字节；ib=2 表示访问第二个 8 字节；一共可访问 256（0xFF）个门。

练习：

在 IDT 表里面放一个中断门或陷阱门（selector 可以填成 0x1B）。

课后理解：

陷阱门与调度门类似，只是访问的是 IDT 段，将对应 IDT 中的 type 类型改为 386 陷阱门，并使用 INT 指令代替 jmp/call。



代码实现：

(1)获取 IDT 段首地址。

```
Char buff[6];
__asm{SIDT fword ptr[buff]}
```

查看 buff 内存储的数据即为 IDT 首地址，多 CPU 会有多个，可通过 SetThreadAffinityMask(GetCurrentThread(),1);将 cpu 设置为单个，或将虚拟机设置为 1 个 cpu，在虚拟机中执行该代码。

读取子函数地址，将其放置在门描述符中的 offset 段。

```
Int *pFun = (int*)&func;
```

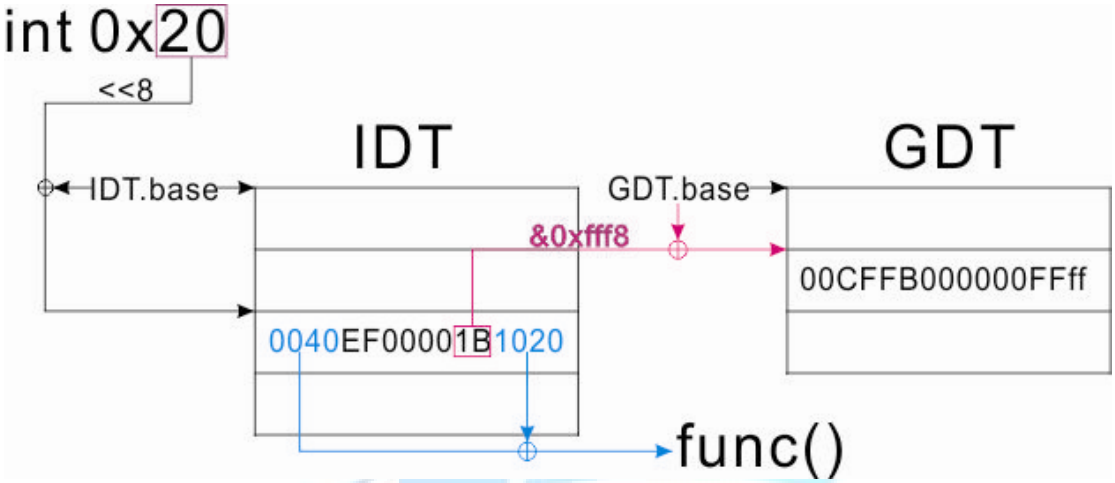
(2)寻找 IDT 段中的空闲段，将陷阱门描述符填入

假设获取的子函数地址为 0x00401020，则 offset31-16 位为 0x0040，p=1，DPL=3，

DT=0，TYPE=0xF（P390），Dword Count=0，Selector=0x1B（cs 属性），offset15-0 位为 0x1020（见 P391）。完整为：0040ef00 001b1020

(3)假设 IDT 段的空闲段为第 32 个段，则在 VC 中执行__asm{INT 32}。

(4)可以选择在子函数中打印[esp],[esp+4],[esp+8]的值。



完整代码及实现：

```
DWORD dwEsp;
DWORD ESP_0;
DWORD ESP_4;
DWORD ESP_8;
WORD dwCs;
void __declspec(naked)func()
{
    __asm
    {
        mov dwEsp,esp
        mov dwCs,CS
        mov dwEsp,esp
        mov eax,[esp+0]
        mov ESP_0,eax
        mov eax,[esp+4]
        mov ESP_4,eax
```

```
        mov eax,[esp+8]
        mov ESP_8,eax
        iretd
    }
}
int main(int argc, char* argv[])
{
    int* cFun = (int*)&func;
    char buff[6];
    SetThreadAffinityMask(GetCurrentThread(),1);
    __asm
    {
     SIDT fword ptr[buff]
     int 0x20
    }
    printf("dwEsp=%08X dwCs=%08X!\n",dwEsp,dwCs);
    printf("EFLAG:%08X\n",ESP_8);
    printf("CS:%08X\n",ESP_4);
    printf("EIP:%08X\n",ESP_0);
    return 0;
}
```

Windebug 部分：在内核调试状态下，查看哪些段没有被使用：

dq 8003f400 l50

修改闲置的 IDT 段：eq 8003f500

回车后在屏幕中显示初始值，且输入框变为 Input>

然后输入陷阱门描述符段：

```
Input> 0040ef00001b1020
```

此时地址被修改，且自动跳转到下一段地址待用户输入修改值，取消直接按回车键。

再次输入 dq 8003f500 查看空闲段已被修改：

```
lkd> dq 8003f500
8003f500  0040ef00`001b1020 00000000`00080000
```

最后运行 VC 程序，查看打印的 esp 值。

```
dwEsp=0012FF1C dwCs=0000001B!
esp8:00000212
esp4:0000001B
esp0:004010A5
```

通常情况下，[esp]= EIP,[esp+4]= CS,[esp+8]= EFLAG

<2>中断门与陷阱门类似，只将 IDT 中的 type 类型改为 386 中断门。

从 3 环到 0 环，再将门描述符 selector 改为 0x08

陷阱门 0 环打印：

```
dwEsp=EEEBADCC dwCs=00000008!
esp8:00000212
esp4:0000001B
esp0:00401085
```

课后总结：

中断门和陷阱门与调用门一样，也是忽略指令里面的 eip，放在 GDT 表里面叫做调用门，放在 IDT 表里面是中断门、陷阱门。

课后练习：

自己构成中断门，陷阱门。

5.7 TSS

本节主要内容：

1. 任务段
2. TSS 结构
3. 出错处理

老唐语录：

LDTR 描述符放在 GDT 表中，LDTR.Base 和 TR.Base 的值表示的含义是什么？

GDT 表里面可以放六个段描述符和调用门描述符。

LDT.Base 指向 LDT 表，里面也是 8 个字节一组存放描述符（windows 没有使用）。

所以选择子中当 TI=1 时，从 LDT 表取 8 个字节，TI=0，从 GDT 表取 8 个字节，如图 5-8 所示：

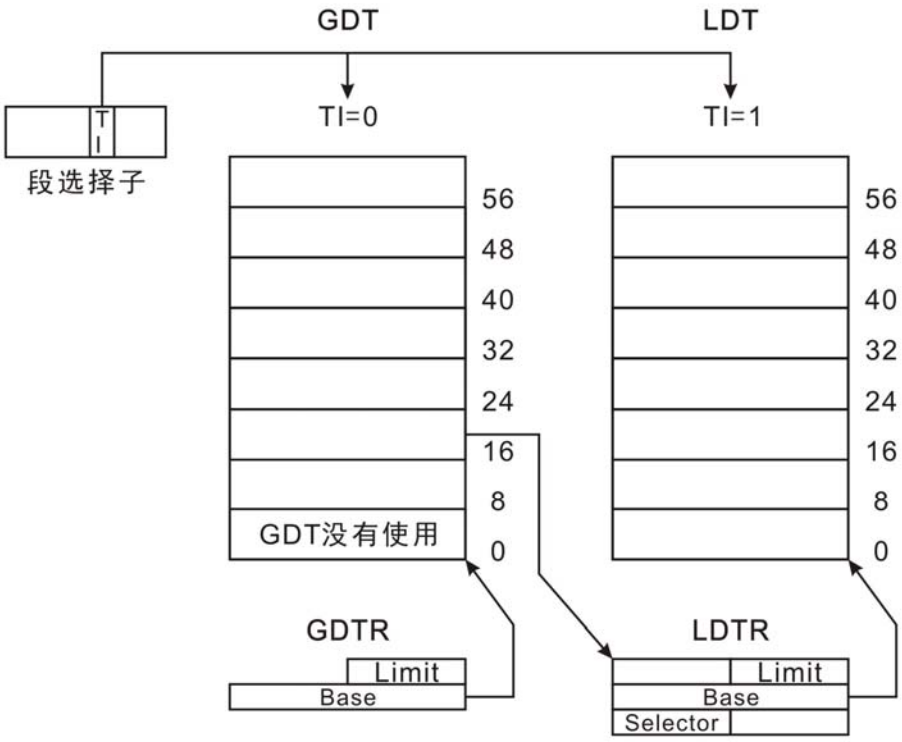


图 5-8：LDT

TR.Base（基址）和 TR.limit（段界限）指向一个结构体，称为 TSS，如图 5-9：

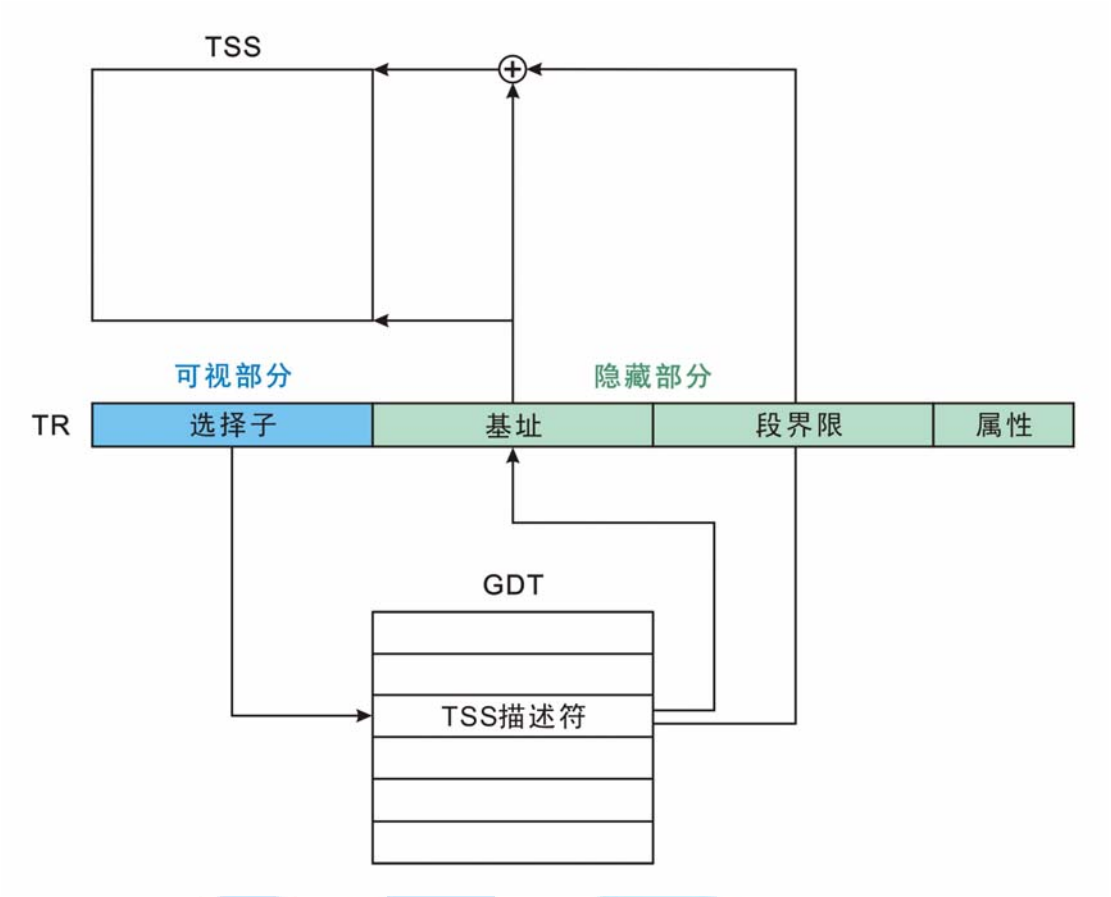


图 5-9：TR

练习：

使用 WinDbg 观察 TR 的 Base 和 Limit 值（首先获取 TR.Selector）。

TSS 结构如图 5-10：

图 5-10：TSS

将 GDT 表中 TR 描述符（0x28）清零后，计算机仍然可以正常工作，因为 TR 只加载一次。将 TSS 中的 SS0 或 ESP0 清零后，计算机会蓝屏。进 0 环只使用 SS0 和 ESP0。

INT 或者 Call 指令可以同时改 CS 和 SS。门描述符其实有 16 个字节，8 个字节存储 EIP，selector 指向的另外 8 个字节是代码段。则门描述符含有两个 DPL。CPL 数值上要小于等于 DPL，才可以使用门描述符。

练习：

实现中断门或陷阱门进 0 环（将门描述符里面的段选择子改为 0x08）。

既然实现了修改了 CS 的低两位，则 SS 的低两位一定被修改，所以使用扩展字段 TSS 提供 SS（0 环是 SS0，1 环是 SS1，2 环是 SS2）。

提问：为什么 TSS 里面有 SS0，SS1，SS2，没有 SS3？

因为三环到三环没有修改权限，CS 没有改变，SS 没有改变。

权限切换时，要保存信息：PUSH ESP 和 SS。

当通过门，权限（CS）不变的时候，使用 CALL 的时候，只会 push 两个值，当前的 CS 和下条指令的 EIP；权限发生改变的时候，push 四个值，push ss，esp，eip，CS。使用 INT 指令会 push 五个值：SS，esp，eflag，cs，eip。使用门时，不能使用 jmp，因为 eip 指定。

权限切换时，先切换栈（esp0 给 esp，ss0 给 ss），再 push 值。

门描述符里面有 dword count 字段，在切换栈时，有可能还会 push 参数。

如想在切换权限时，修改所有的寄存器，可以使用任务段。任务段描述符（类型 9），可以使用 LTR 访问。更深一层，可以使用 jmp/call EP 访问任务段（高两个字节可以是 CS 也可以是其他段）。区别：jmp 会改变所有的寄存器，LTR 只改变 TR。

代码段可以放到门里面去（调用门、中断门和陷阱门），那么任务段可以放到门里面去，叫做任务门。

任务段和任务门的区别？

如果当前指令出错，会执行 int 0xd（门，权限是 3），然后执行 d 对应描述符 eip 的指令，cpu 默认执行权限是 0 环，默认指令长度是 0（在保存堆栈和 eip 时，本指令并非代码段中的指令）。

如果 cpu 默认的指令出错，怎么办？

除零错，系统会执行 int 0，如果 int 0 出错，则会执行 int 8，如果 int 8 出错，系统直接重启。

int 0 到 int 0x1F 指错误处理：

异常	描述	程序状态改变
0	除 0 错	指令执行前产生，无除法错误
1	调试异常	指令执行前产生，无调试错误
3	断点异常	不影响任何寄存器和内存值

4	溢出异常	不影响任何寄存器和内存值
8	双重异常	关闭应用程序或处理器
13	段错误	指令执行前产生，无段错误
14	页错误异常	指令执行前产生，无页错误

普通段调用时，出错的概率很高，如果调用任务段，出错的概率很低，因为任务段将所有的寄存器和堆栈都换了一遍，系统将 int 0x8 做成任务门。

练习：

使用 windbg 查看 int 8 信息（指令 u）

外部设备断电会执行 int 0x2（也使用了任务段），如果 cpu 是多核，每个 cpu 的寄存器都是独立的，不能相互读取，只能通过内存通信（使用 int 0x2）。

段本身依赖 GDT 和 IDT，GDT 和 IDT 放在 2G 以上空间。可读（SGDT、SIDT）但是只能在 0 环可写（LGDT、LIDT、LTR）。

所有的 API 通过 int 0x2E 进入 0 环。

练习：

写一个嵌套循环，让程序运行 30 秒。然后将该程序放入虚拟机中运行，测试时间。

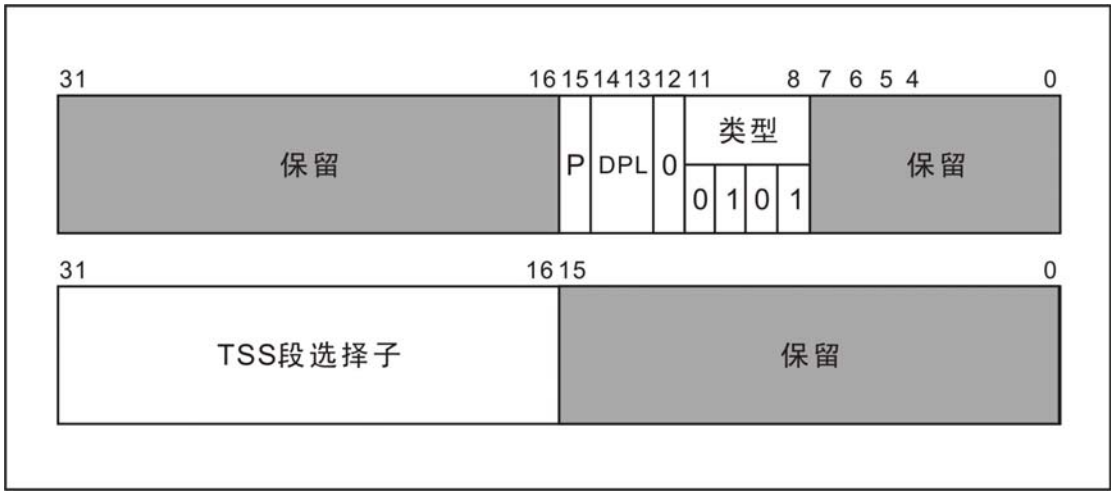
说明：结果相同，虚拟机使用的也是真实机，之所以感觉慢，是因为虚拟机的显卡是虚拟的，显示和刷新要慢。

那么将虚拟机里面的 GDT 和 IDT 抹零，为什么真实机不会死机？

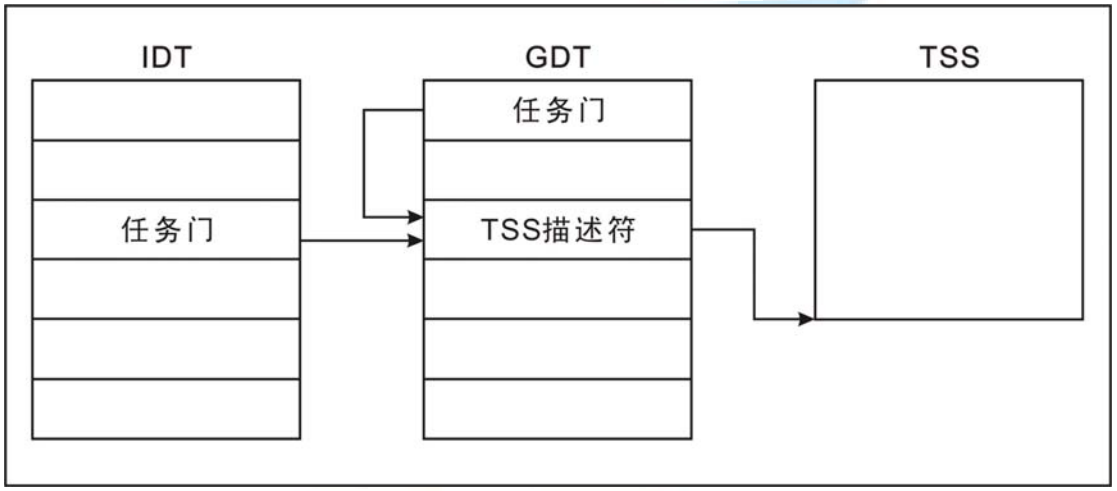
说明：VT 技术解决了段的问题。凡是对段的操作都由 int 0x3 接管，然后 int 0x3 挂钩模拟段的指令。

课后理解：

任务门：



访问 TSS 格式：



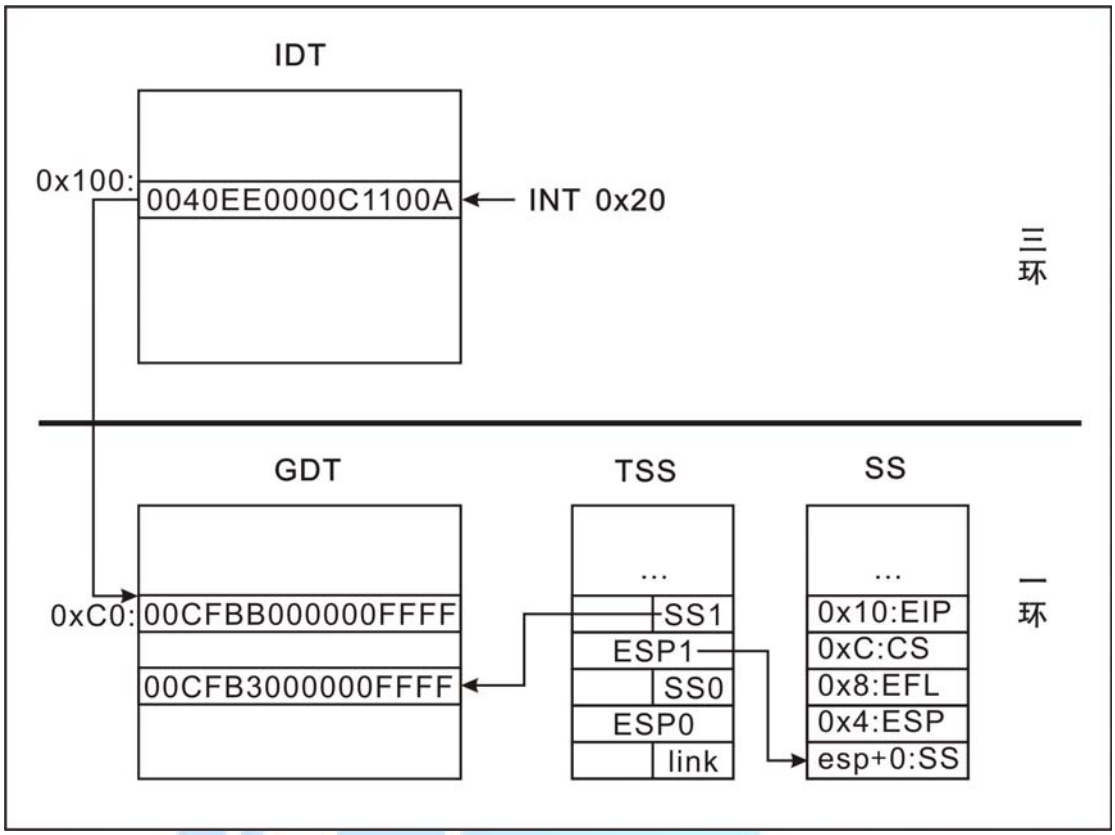
课后总结：

门里面的 DPL，表示谁可以进这个门，段里面的 DPL，表示进门之后的权限。

课后练习：

从三环切换到 1 环。

如下图：



完整代码：

```
#include "stdafx.h"
#include <Windows.h>
DWORD dwOK;
DWORD dwESP;
__declspec(naked)func()
{
dwOK = 1;
__asm
  {
    mov eax,esp
    mov dwESP,eax
    iretd
  }
}
```

```
int main(int argc, char* argv[])
{
  int* a = (int*)&func;//0040100a
  char buff[0x10];//12ff18
  int iCr3;
  SetThreadAffinityMask(GetCurrentThread(),1);
  printf("input CR3:\n");
  scanf("%x",&iCr3);
  DWORD iTss[0x68] = {
  0x00000000,//link
  0x00000000,//esp0
  0x00000000,//ss0
 (DWORD)buff,//esp1
  0x00000010,//ss1
  0x00000000,//esp2
  0x00000000,//ss2
 (DWORD)iCr3,//cr3
  0x0040100a,//eip
  0x00000000,//eflags
  0x00000000,//eax
  0x00000000,//ecx
  0x00000000,//edx
  0x00000000,//ebx
  (DWORD)buff,//esp
  0x00000000,//ebp
  0x00000000,//esi
  0x00000000,//edi
  0x00000023,//es
  0x00000008,//cs
```

```
  0x00000010,//ss
  0x00000023,//ds
  0x00000030,//fs
  0x00000000,//gs
  0x00000000,//ldt
  0x20ac0000
};
__asm{int 0x20}
printf("ESP = 0x%08x\nok = %d\n",dwESP,dwOK);
return 0;
}
```

5.8 页

本节主要内容：

1. 线性地址划分
2. 物理地址寻址规则

老唐语录：

一开始我们强调方括号里面是内存，后来我们发现，方括号里面并不是内存，是个有效地址（EA：effective address）。段基址（SEG.Base）+有效地址=线性地址（LA：Linear address）。其实线性地址还不是地址，因为我们发现 0-4G 空间，还有很多地址不能访问，比如你的机器可能是 512M 内存，2G 内存。其实还有个物理内存（PA：physical address）。物理地址还会转换成板卡地址（透明的）。

cpu 寄存器中只有 CR3 存放的是物理地址。

一个给定的线性地址是 32 位，我们将其分为 3 段如图 5-11：

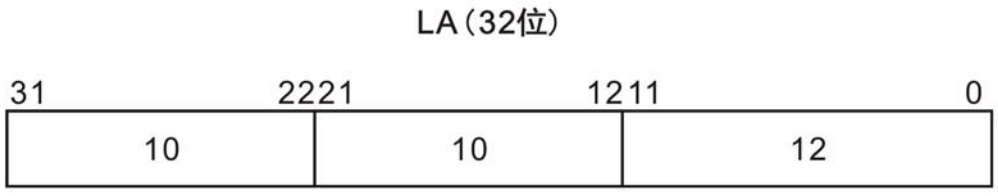


图 5-11：线性地址

那么一个线性地址如何转换成物理地址呢？

将 C:\boot.ini 文件中 noexecute 改为 execute，重启。

练习：使用！db/dw/dd/dq 访问物理地址。

写 VC6 程序，并嵌入 int 3（0xcc）断下程序，并观察自己的 CR3（使用双机调试）。

如何将 LA 转换成物理地址。现在写一个线性地址，

1）将其拆成 3 段（按图 5-11）。
2）CR3 指向一个物理页，一共 4096 个字节，共 1024 项。
3）在 PDI×4+CR3 的位置取 4 个字节，这四个字节称为 PDE。
4）将 PDE 低 12 位清零+PTI×4 的位置取 4 个字节，这四个字节称为 PTE。

5）将 PTE 低 12 位清零+offset = PA

练习：转换线性地址 0x402020 对应的物理地址。

每个程序都有一个 CR3，但是内存中 CR3 寄存器每一时间只加载一个 CR3。CR3 中有 1024 项：PDE0，PDE1，PDE2……，不同的 PTE 可以指向一个物理地址。PDE 的第 0 位如果是 0，代表 PDE 无意义，如果为 1，表示高 22 位指向一个物理地址（PTE 属于物理地址）。一个物理地址可以对应多个线性地址，但是一个线性地址只能对应一个物理地址（如图 5-12）。

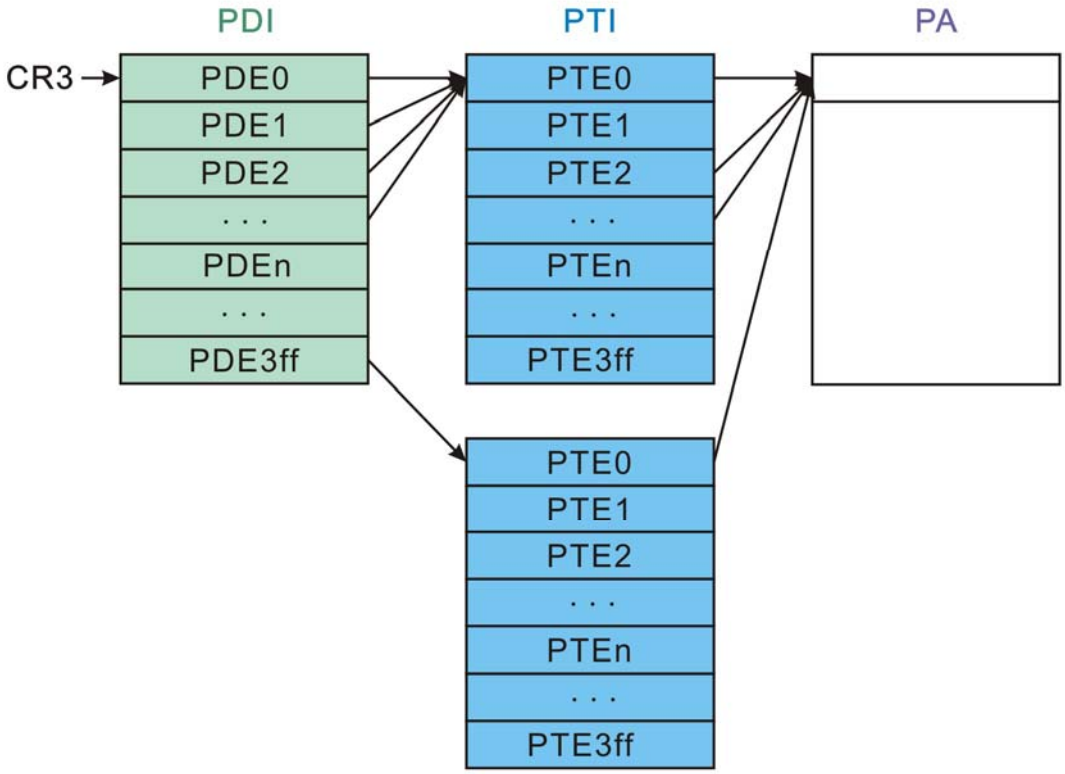


图 5-12：分页

假设内存地址只有 1G，要在高地址处存储一个值。

练习：在 PDE、PTE 里填入一个有效地址并在程序中使用。

PTE 可以指向任意物理地址，当然也可以指向自己。

注：每个进程 2G 以上的内存内容基本相同，2G 以下基本不同。

课后理解：

```
Mov eax,ds:[0x400FFE]; //找出线性地址所对应的物理地址
```

1. windbg 找出对应进程 cr3，即页基址 PDE0。

2．从 PDE0 处，以 4 个字节为单位，偏移到 PDI（线性地址目录）处，即 PDE。

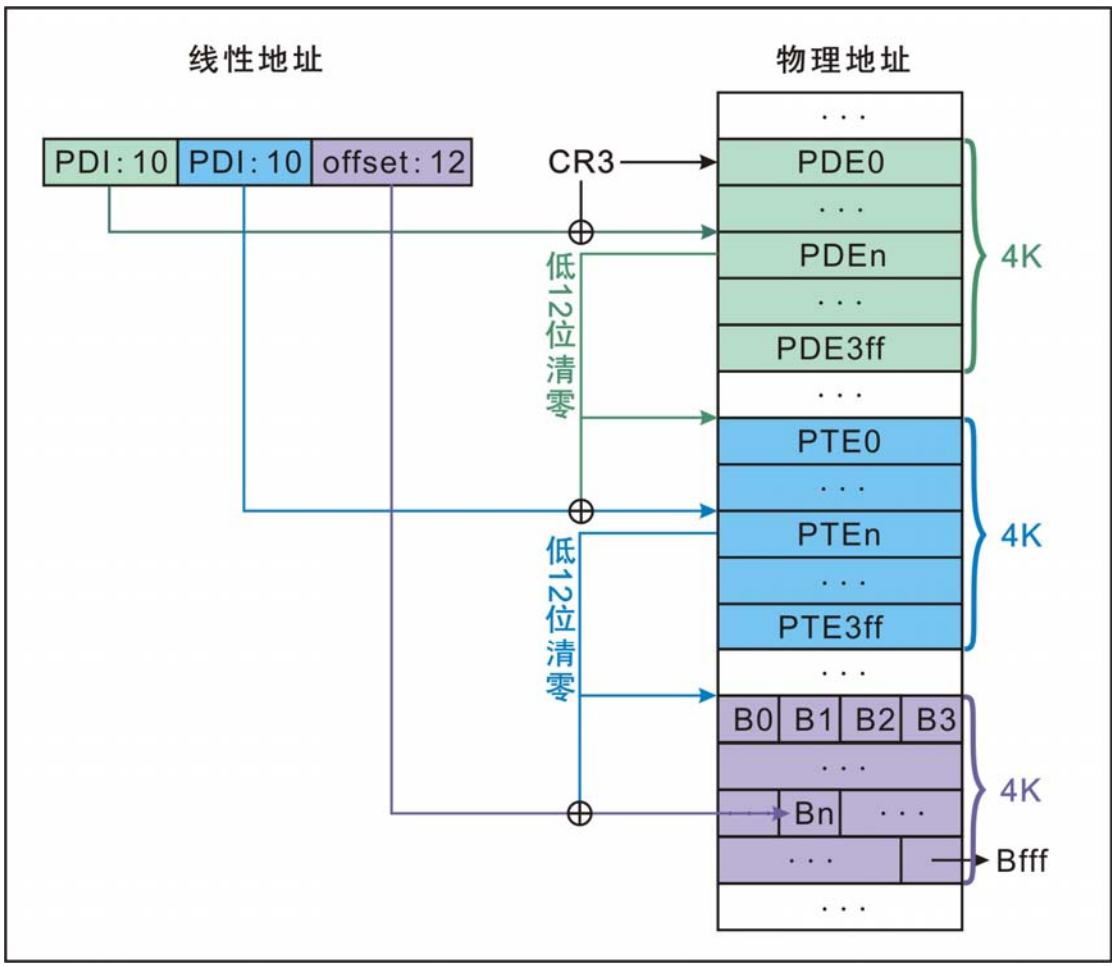
3．PDE 为 32 位，低 12 位清零，为新的页基址 PTE0。

4．从 PTE0 处，以 4 个字节为单位，偏移到 PTI（线性地址中的表）处，即 PTE。

5．PTE 为 32 位，低 12 位清零，为新的页基址 b0。

6．从 b0 处，以 1 个字节为单位，偏移到 offset（线性地址中的偏移）处，即 PA（物理地址 bN）。

具体步骤见下图：



课后总结：

页表可以任意指定物理页。

课后练习：

将 CR3 中每一个 PDE，PDE 中 PTE 拆分。

5.9 关键PTE

本节主要内容：

1. 如何改写物理地址
2. 页表属性

老唐语录：

当 PTE 的 0 位为 0 时，不可访问。

练习：写出 edx 值：

```
__asm
{
  mov eax,401FFE
  mov edx,[eax]
}
```

说明：将线性地址对应的 PDI、PTI、offset、PDE、PTE、PA 写出来。

注：地址跨页，地址可以拆分成 0x401FFE，0x401FFF，0x402000，0x402001。

PDE 低 12 位的属性如图 5-12：

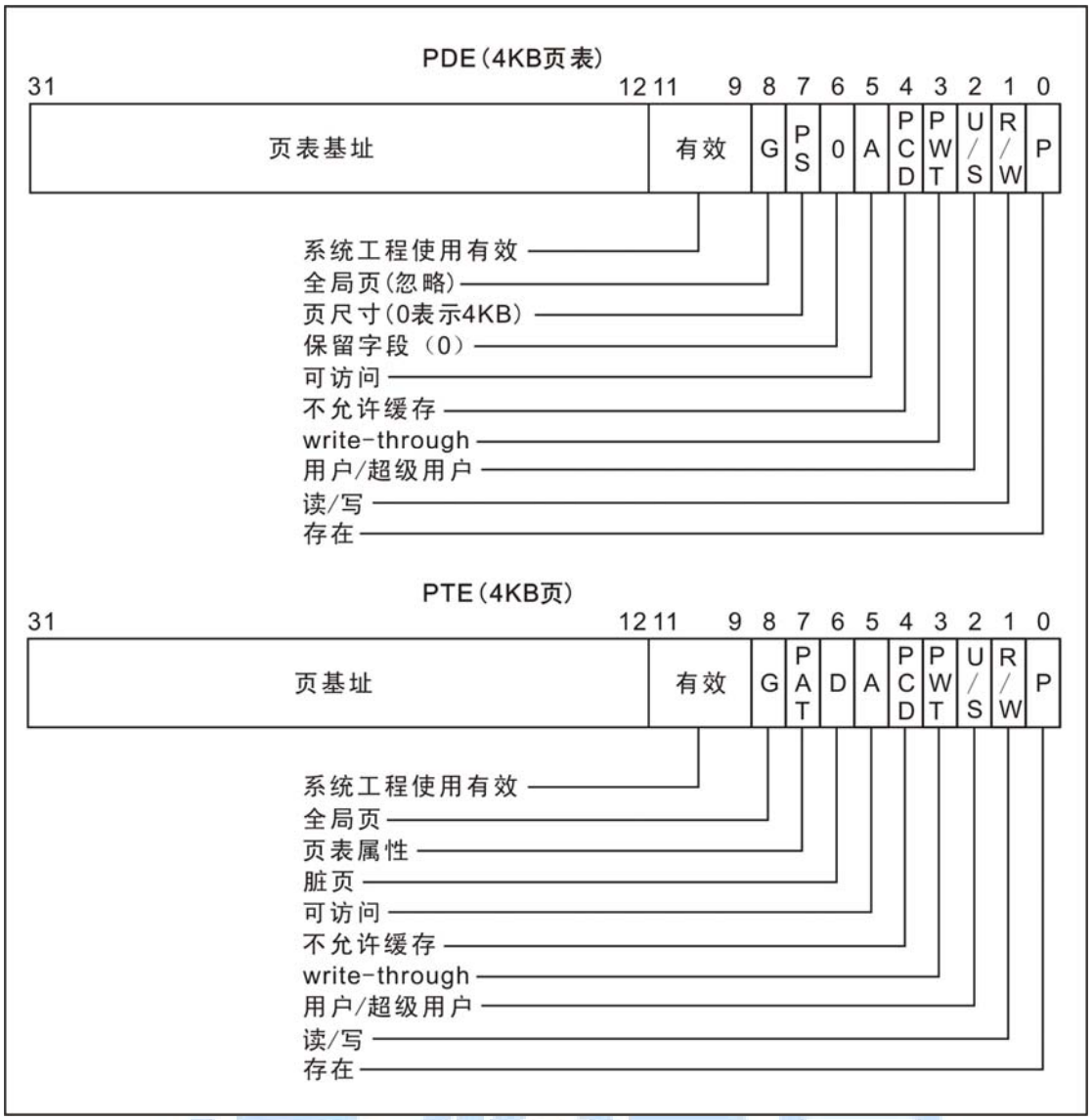


图 5-12：PDE 与 PTE

PDE 和 PTE 的结构相同。

P：存在位，当 P=0，表示不存在。

R/W：读写位。=1 比哦啊是可写，0 表示只读。

U\S：=0，表示系统，=1，表示用户

A：访问位，是否被访问过，如果被访问过，置 1。

D：脏位，=0 表示没有被写过。

PS：=1， PDE 低 22 位不要，线性地址分成两段：高 10 位是 PDI，没有 PTI，低 22 位是 offset（4M，大页）。

AVL：用户可使用位（cpu 不检测该位）。

想要访问一个物理地址，必须首先建立 PDE 和 PTE。PDE 和 PTE 也必须有对应的线性地址。2G 以上的空间 U\S=0，用户程序无法访问（改为 1 后可以访问）。

操作系统的 0 环或者是 3 环，仅仅是 CS 的低 2 位，将 0-2G 作为用户模式，2-4G 作为系统模式，而 Linux 是由系统自己设置。

我们接着讲页，如图 5-13：（PD 里面 PDE 指向很多 PT，然后是 PA）

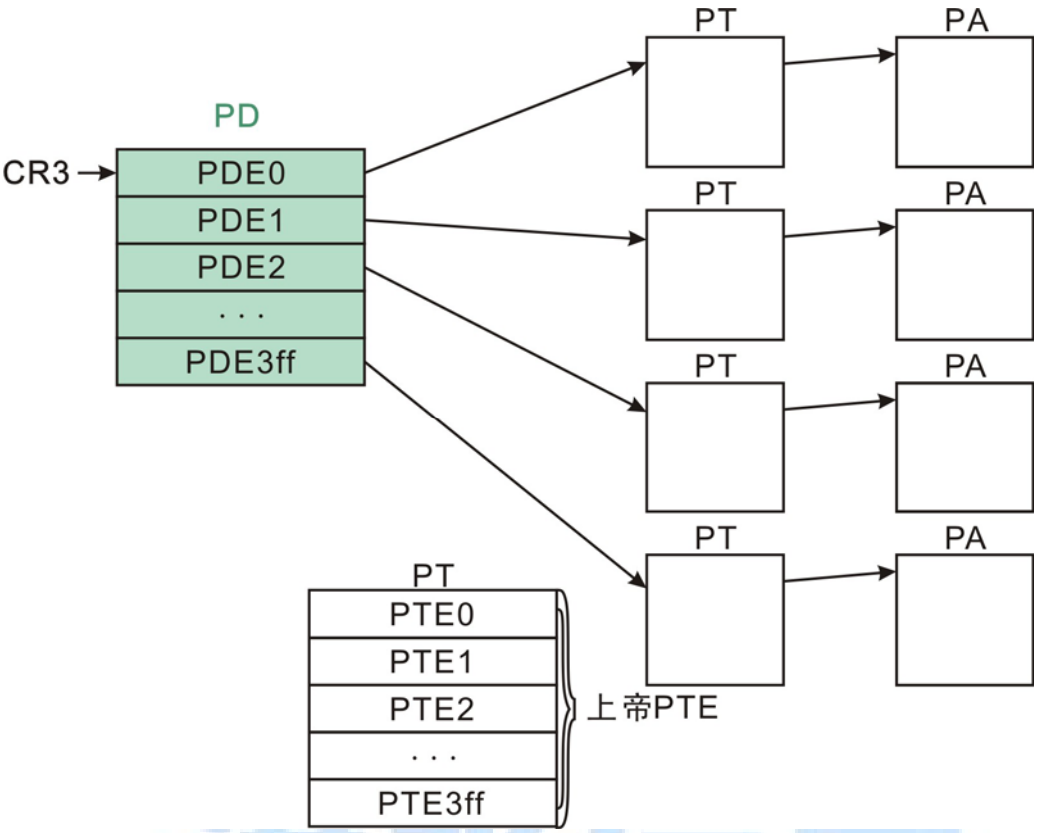


图 5-13：上帝 PTE

PTE 本身是物理地址，如果要得到访问，需要另一个 PTE，所以我们使用一个上帝 PTE，将所有 PTE 放在上帝 PTE 里面。

我们会发现，如果所有 PTE 挂入上帝 PTE 和挂入 PD（页目录表）的顺序一样时，那么 PDE0 和 PTE0 值一样。类似的，如果上帝 PTE 要得到访问，他也要将自身地址挂入自己 PT 里面。既然两个内容一致，那就可以去掉 PD，直接使用上帝 PTE。

我们拆 c030000 会发现，PDI=PTI。我们通过 c0300000 可以访问页目录本身。

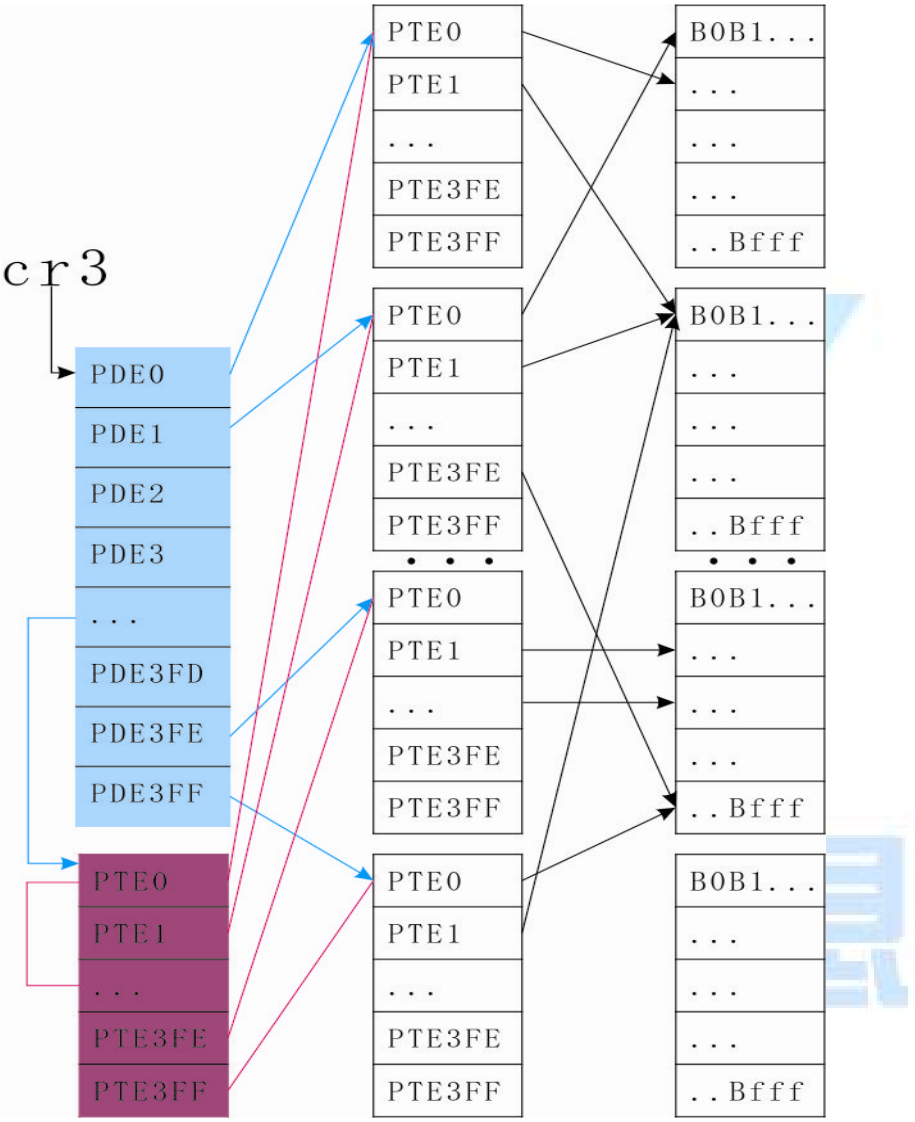
当创建进程 B 时，先在进程 A 中将 B 进程所有信息全部建好包括 CR3，然后切换 CR3 即可。

有了上帝页，可以访问和修改任意物理页。

在页里面没有 0 环和三环，只有用户和系统。系统包括：0、1、2 环，用户为 3 环。

可以解决很多问题：比如 0 地址为什么不能访问？是错误的，我们可以设置为可访问。

课后理解：



课后总结：

一个线性地址只能对应一个物理地址。

课后练习：

将应用程序 0-4G 拆分（PDI、PTI 对应的每一项）。

运行两个程序，并比较其在内存中的区别。

5.10 2-9-9-12 分页

本节主要内容：

1. 2-9-9-12 分页原理
2. TLB

老唐语录：

如果我们按之前讲的分页方式（10-10-12 分页），寻址方式是 32 位（4G）。为什么页以 4K 为单位？

如果将页目录和页表如果全部填满，则需要 4M 空间，换句话说，如果程序需要 4G 的运行空间，则要浪费 4M 的空间。所以如果页太小，浪费太多，如果页太大，不好分配。Intel 设计为 4k（ARM 为 2K）。

为了扩大访问地址，将 PDE 设为 64 位，PTE 是 64 位，则 PD 有 512 项 PDE，PT 为 512 项 PTE，在前面加了 4 个指针：PDPE。如图 5-14：

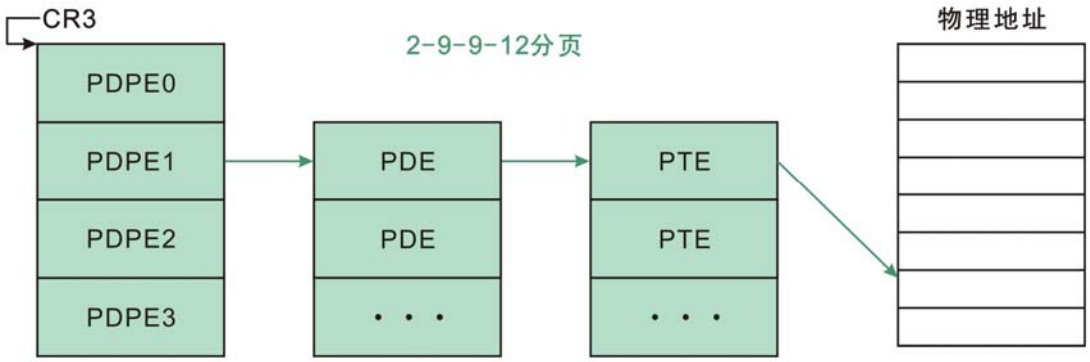


图 5-14：2-9-9-12 分页

PDPE => page directory pointer entry 页目录表指针

8 个字节的划分：低 12 位是属性，中间是物理地址，物理地址只用了 48 位，高位是保留的，现在只定义了最高位 X 如图 5-15 所示：

图 5-15：PDPE

我们原来学的断的时候是可执行可读可写，页的属性也有可执行可读可写，但是这样有一个问题啊，可读的一定可以执行，可写的页可以执行，在页里面没有可执行这个位，只有可读可写，那这样有个问题了，栈溢出漏洞， 栈返回的时候不是有个 ret 返回吗，它只要想个办法栈溢出把 RET 那个地方的 EIP 给覆盖了，跳到它指定的地方去，它指定的地方肯定是执行代码，那个代码是应用程序写进去的，肯定是可写的，

不可写，代码怎么进去，难道那么巧都是垃圾，漏洞都是依赖数据可执行。所以 intel 就做了个硬件保护的，所以说可写的就不可执行，就做了一个不可执行位。那你软件溢出了也没关系。EIP 蹦到你那里去，不可执行。

X = 1 表示这个页不可执行，堵漏洞的。

数据保护 DEP，打开我的电脑 => 属性 => 高级 => DEP （数据保护）设置这个必须是 2-9-9-12 分页方式才可以。

2-9-9-12 页目录关系都找到了没有？ 找到可以读写 4 个 PDPE 的线性地址就可以了，CR3 对应的线性地址找不到。

通过一个线性地址，访问一个物理内存比如：1 个字节。其实未必是真正的读的是 4 个字节，我们先读的 PDE,在读的 PTE 最后才读的 4 个字节的页（12 个字节）在 2-9-9-12 你会发现读了几个字节？你读了 24 个字节，这管理成本也太高了吧。以前我们算的是空间的问题，现在我们来算时间的问题。页尺寸是空间管理成本。

现在想一想，如果要访问一个字节，至少要读多少个字节 2-9-9-12 （8 个字节）最多要读多少个字节（48 个字节） 还要置 A 位，D 位，C 位。

如果访问两个字节以上还要牵扯到跨页的问题。 这个就需要读更多的字节了，管理成本太高了。

所以说管理成本太高了，没办法，只能做记录。

CPU 内部做了一个寄存器表，记录这些东西的寄存器表格，它记录的表格叫做 TLB，这个表格的内容如下：

LA	PA	ATTR	LRU
...	...	...	...

属性是 PDPE PDE PTE 三个属性 AND 起来的。

这个表很贵，比寄存器贵很多倍，它的速度是内存的 10 倍以上。

这个表太小没什么意义，因为还没有运行一会就要去拆分，太大也没什么意思，比如几百兆，经常访问的就是几十个页，几百个页，不可能是几千个页，几万个页， 如果这个表太多，查找的速度就很慢。还不如去拆分，CPU 是一个一个比较的。

所以有一个多路比较器（硬件）， 里面的每个线性地址都对应这一个比较器，它同时比较。有多少个线性地址，这个比较器就有多少个。

TLB 贵，所以不能太大，那到底有多少项了？

一般是机器是 64 条的 还有 128 条 256 条的 不同的 CPU 不一样

CPUID 这个指令可以看出 TLB 多少项

3．如果 TLB 满了，怎么办了？

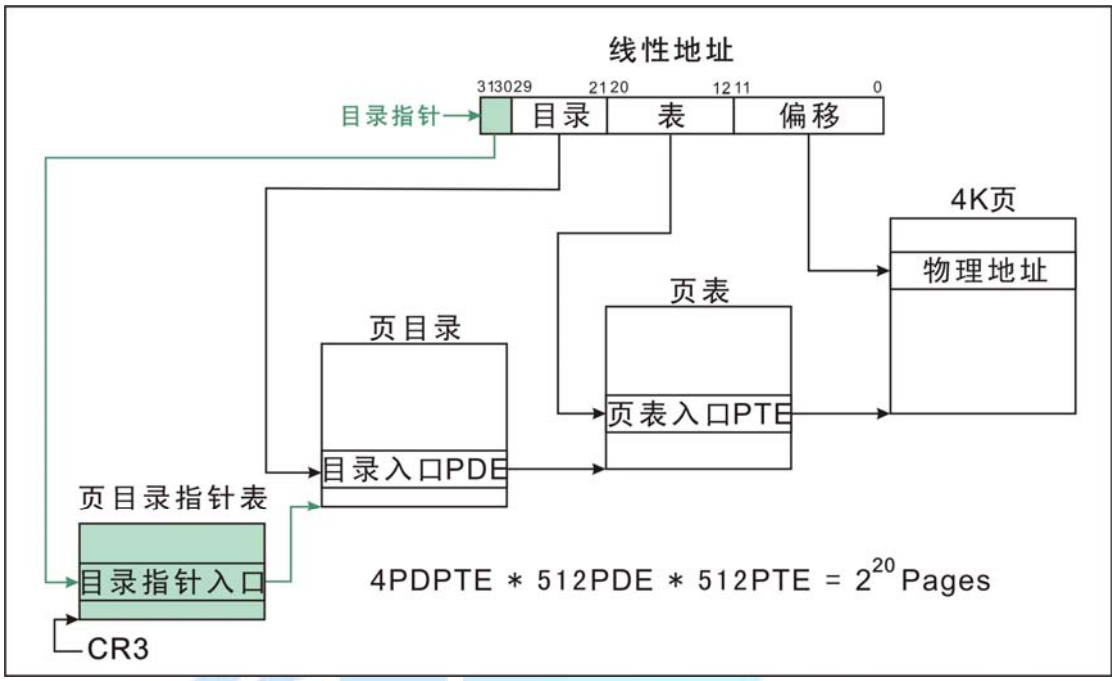
把不经常用的废弃了，看统计信息

4．只有改变 CR3 了，TLB 就立马刷新了，因为 CR3 改变了，那么线性地址对应的关系就乱套了，所以要重新刷新 TLB

这里就有一个问题了。2G 以上的物理地址都是映射到同一个地方，如果 CR3 改了，TLB 刷新了，又要重建 2G 以上的地址，这不是浪费吗？

所以说在 PTE 里面有个属性位 G 位 如果 PTE.G = 1 时不要刷新 物理地址是固定的，在切换 CR3 的时候，凡是 G 位= 1 的地址都不废弃，如 果 TLB 满了，还是要把最不常用废掉。

课后理解：



课后总结：

2-9-9-12 分页可以访问 64G 物理地址。页的空间管理寄存器 TLB。

课后练习：

1．按 2-9-9-12 分页拆分线性地址。

2．将应用程序 0-4G 拆分（PDI、PTI 对应的每一项：2-9-9-12 分页模式）。

5.11 控制寄存器

本节主要内容：

1．什么是 CACHE

2．掌握 CACHE 结构

老唐语录：

如果 p = 0 时 TLB 是不会记录的。

如果我们给 cr3 赋值 mov cr3,eax –> 这个 EAX 要很有讲究，如果原来分页是 2-9-9-12，现在分页也是 2-9-9-12 这个好办，只要保证切换的两个线性空间一样就行了，但是你看虚拟机就知道啦，外面的主机是配的 2-9-9-12 的分页方式，里面的虚拟机是可以配成 10-10-12 分页方式的，也可以配成 2-9-9-12 分页方式的。

当原来的 CR3 是 2-9-9-12 分页的，现在的 CR3 是 10-10-12 分页方式的 你要怎么建这个 CR3 才能切换过去了？ 自己在纸上画。可以在机器上试下怎么切。

这个分页方式为什么可以配了？这个分页方式，是由一个寄存器控制的。

我们原来学过 CR0 - CR4 吗？如图 5-16 所示：

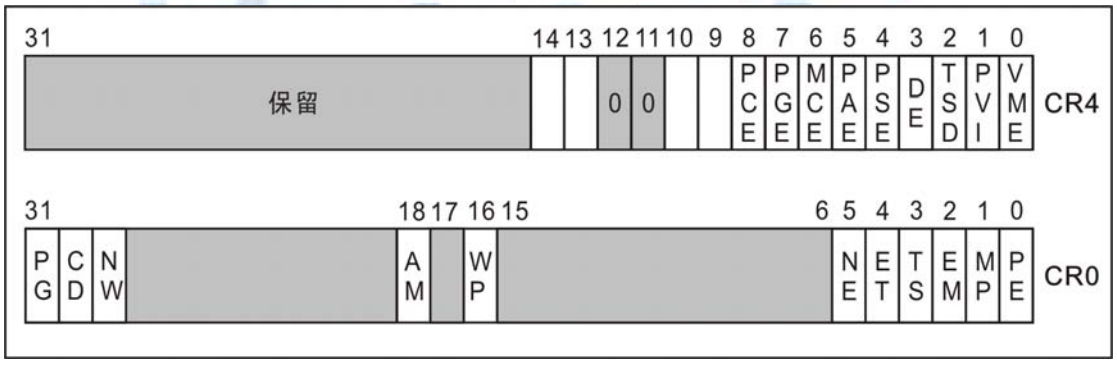


图 5-16 CR0-CR4

在 CR4 里面有一个位是控制 CPU 分页方式的

PAE = 1 时 控制 2-9-9-12 分页的

PAE = 0 时 控制 10-10-12 分页的

2-9-9-12 => ps 位 = 1 时 为 2M 页

CR4.PSE => PSE = 1 时 PDE 里的 ps 位有效

CR4.PSE => PSE = 0 时 PDE 里的 ps 无效

线性地址访问物理内存的时候，线性地址不是要吗？拆分太多次，时间成本过高，因为寄存器时间至少是内存时间的十倍以上，那个 TLB 也是几十倍以上，所以很快的，但是真正读内存的时候还是很慢啊，因为内存要比寄存器慢上十倍啊，mov eax,ecx 我们是尽量用寄存器做中间变量，但是 C 语言不一样啊，它不支持这样啊，C 语言的中间变量全部都是内存和栈啊，还是内存啊，写程序的时候，没有非要强调你用寄存器啊，都是用内存啊，因为内存容易修改，你用汇编写多麻烦啊，那内存比寄存器慢十倍，为什么还要用内存了，因为内存也不是真正的去读物理内存的，你刚刚读出来的内存它也有记录

它也是 CPU 内存寄存器记录的 CACHE (CPUID 可以查看)

LA - PA –>TLB。

PA - DATA ->CACHE。

PWT page write through (写通)。

这个位是写 cache 的时候也要把 cache 写入内存中。

当 pwt = 1 时 cache 和内存里面的值相同，读的时候 cache 有两份，但是写的时候不一样，写的时候只是改变 cache 里面的值，并没有改变内存里面的值，所以想要同时改变 cahce 和内存 就必须把 PWT 位 置 1

PCD page cache disabled 禁止页缓存

PCD = 1 时 禁止缓存，禁止某个页写缓存，直接写内存

1. 先换 CR3 还是先换 CR4 如图 5-17 所示：

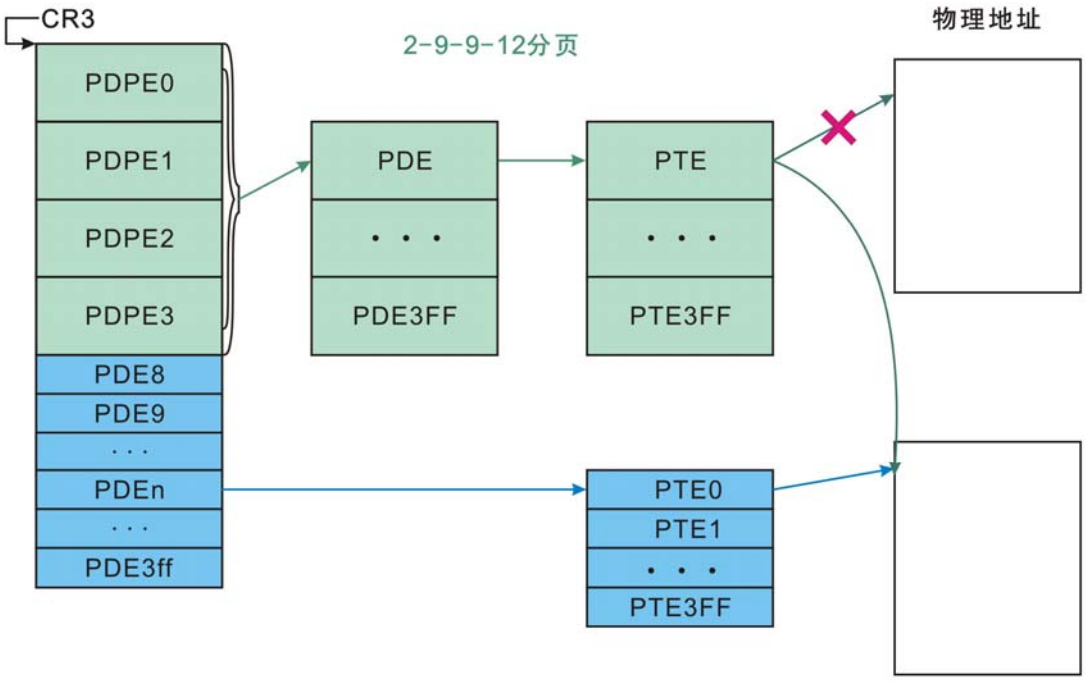
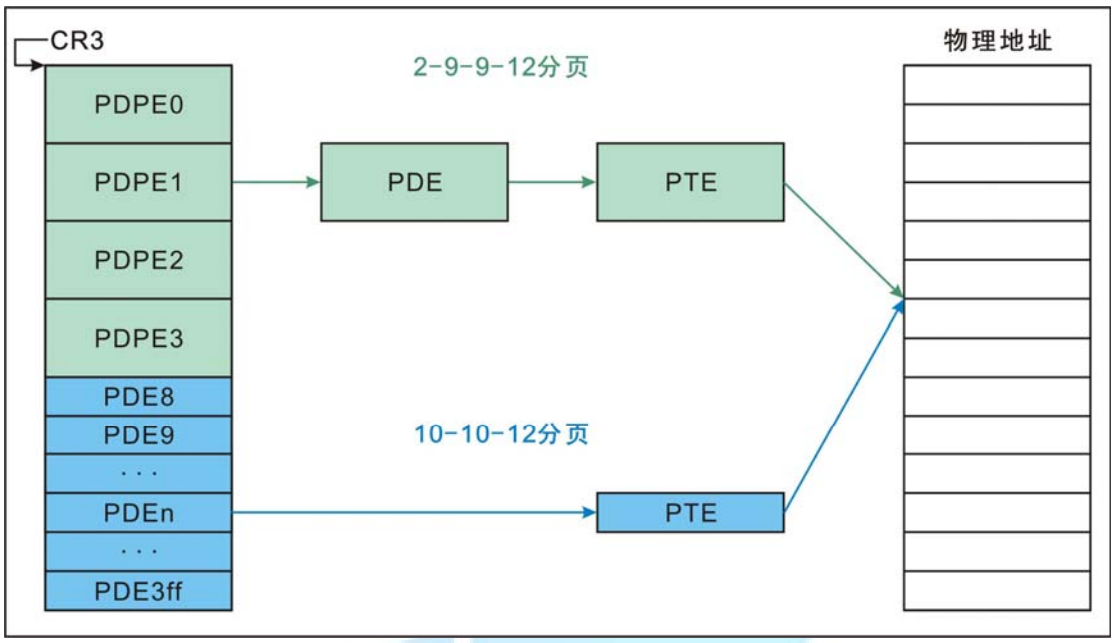


图 5-17

1．只要不用 32M 的线性地址都可以 （先换 CR3 还是先换 CR4）

2．其实只要不用 8M 的线性地址就可以了（线性地址超过 8M 距离就拉开了）。

课后理解：



课后总结：

页的时间管理寄存器 CACHE。

课后练习：

思考 10-10-12 和 2-9-9-12 是怎么转换的。

第六章 PE（略）

第七章 C++（略）

第八章 操作系统（略）