计算机系统基础 by gMr

> **注：一、二章较详细，三、四章非常省略**
>
> 因为期末周课实在是太多，这课几乎是考前一两天速通的（除了第三章没法速通，之前就认真看了几天），最后还熬了个通宵。一二章认真学了，第三章之前看了第四章到最后实在看不下去了所以这两章没咋写笔记。具体可以从下文的详细程度看出。
>
> 考前留言：完全不是人的课，知识量真的太大了，我只求老师把我捞到 3.7 谢谢，，，
>
> 出分感言：嗯竟然捞到 3.9 了，也是辛苦老师了，期末 89 总评 92，而且看了眼教学班排名，7/48，看来分普遍还是给的很高的，是我实力不行，大家引以为戒，这课真的很难，不适合速通，可能是我上了三学期学过的最难的课（
>
> 以及本人没有选上计基实验，可能选了的话这门课会好学一点。

第一章

冯·诺依曼计算机的组成部分、各部分的功能

控制器：用于控制主动执行指令；

运算器：用于执行指令；

存储器：存放数据和指令；

输入输出设备：通过输入输出设备使用计算机；

存储程序

程序执行前，需将程序包含的**指令和数据**先送入主存，一旦启动程序执行，则计算机必须能够在不需操作人员干预下自动完成逐条指令取出和执行的任务

指令的执行过程

1. 从主存取指令
2. 对指令进行译码
3. PC 增量
4. 取操作数并执行
5. 将结果送主存或寄存器保存

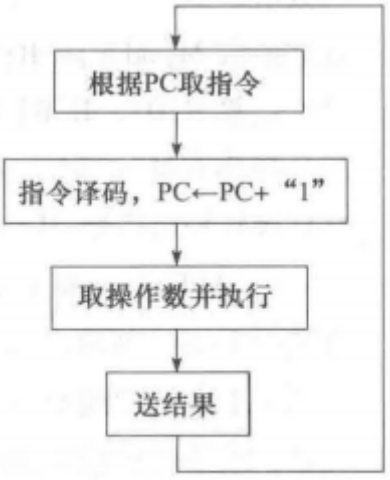


图 1.4　程序执行过程

计算机系统的的划分及用户

层次

电路设计、数字设计、ISA、汇编程序、编译程序、应用程序、操作系统

用户

最终用户：应用程序层

系统管理员：操作系统

应用程序员：编译程序

系统程序员：汇编程序与 ISA 之间

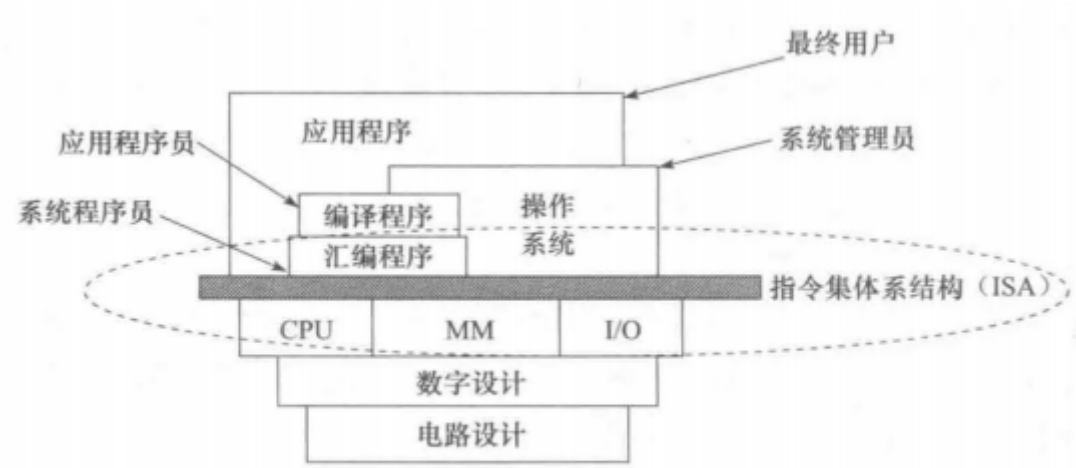


图 1.11　计算机系统的层次化结构

计算机性能

响应时间

响应时间=等待时间+执行时间

执行时间=CPU时间+其他时间

- **CPU 时间**：

  用户 CPU 时间：真正用于运行用户程序代码时间

  系统 CPU 时间：为了执行用户程序而需要 CPU 运行操作系统程序的时间

- **其他时间**：

  I/O 操作时间或 CPU 执行其他用户程序的时间

CPU 时间的计算

- **时钟周期**：CPU 的主脉冲信号
- **时钟频率**：即 CPU 的主频，是时钟周期倒数，即单位时间的时钟周期数量
- **程序的 CPI**：CPI (Cycles Per Instruction)，表示**执行一条指令所需的时钟周期数**

> CPI 与 **总时钟周期数、指令条数** 有关
>
> ↑书上以及答案写的，感觉好扯淡啊，这式子只是关系式又不是定义，难道不应该和指令集体系结构、指令类型、硬件什么有关吗

$$\text{用户}\mathrm{CPU}\text{时间} = \text{程序总时间周期数} \div \text{时钟频率} = \text{程序总时钟周期数} \times \text{时钟周期}$$

第 i 种指令条数和 CPI 分别为 $\mathrm{C_i}, \mathrm{CPI_i}$ 时：

$$\text{程序总时间周期数} = \sum_{i=1}^{n} C_i \times CPI_i$$

第 i 种指令所占比例为 F_i 时，综合 CPI：

$$\text{综合}\mathrm{CPI} = \sum_{i=1}^{n}(CPI_i \times F_i) = \text{程序总时钟周期数} \div \text{程序总指令条数}$$

因此，若已知综合 CPI，则：

$$\text{用户}\mathrm{CPU}\text{时间} = \mathrm{CPI} \times \text{程序总指令条数} \times \text{时钟周期}$$

指令执行速度

- **指令速度的计量单位**：MIPS（Million Instructions Per Second），**平均每秒执行多少百万条指令**

  $$\mathrm{MIPS} = 1 \div (\mathrm{CPI} \times \text{时钟周期}) \div 10^6 = \text{时钟频率} \div \mathrm{CP} \div 10^6$$

  > 因此 CPI 越小越好，CPI 最小时可计算出 **峰值** MIPS

  为什么不能很好反映计算机性能：

  没有考虑时钟周期、指令条数、 CPI 三个因素。

  首先，不同机器的指令集是不同的，而且指令的功能也是不同的，也许在机器 1 上一条指令完成的功能机器 2 需要多条指令。

  其次，不同机器的 CPI 和时钟周期也是不同的，因此同一条指令在不同的机器上所用的时间也不同。

吞吐率

表示在单位时间内所完成的工作量

阿姆达尔定律

$$\text{改进后的执行时间} = \text{改进部分执行时间} \div \text{改进部分的改进倍数} + \text{未改进部分执行时间}$$

或

$$\text{整体改进倍数} = 1 \div (\text{改进部分执行时间比例} \div \text{改进部分的改进倍数} + \text{未改进部分的执行比例})$$

> 这二臂定律感觉我也能提出来

> 考后总结：这一章主要考了计算机层次结构的概念题和计算机性能的计算题，前面的概念背一背，计算机性能这块几个概念要打心底理解（很重要，我觉得结合全称记忆很好记），就够了

第二章

定点数与浮点数

- **定点数**

  小数点位置固定，ez

  注意 8421 码，最常用的 BCD 码，非常好理解，连续四位二进制表示一位十进制

  $(1001\ 0101.0001)_{8421} = (95.1)_{10}$

- **浮点数**

  $X = (-1)^S \times M \times R^E$

  S 为符号位， M 为二进制定点小数称为尾数，R 为基数，E 为阶或指数

  若 n, m 分别为尾数和指数的位数，基数为 2 ，易得

  绝对值最小值为 $0.0\cdots01 \times R^{-11\cdots1}$ ，最大值为 $0.1\cdots11 \times R^{11\cdots1}$

  $2^{-n} \times 2^{-(2^m-1)} \leq |X| \leq (1-2^{-n}) \times 2^{(2^m-1)}$

  > 初看疑问：这里的 m 疑似没有考虑指数的符号位？后面发现是用移码，那没事了。
  >
  > 又往后看了一点，呃其实还是很看表示规范的，先不要计较

定点数的表示方法

- **原码**

  第一位为符号位，0 为正 1 为负，n 为位数

  $-2^{n-1} < X < 2^{n-1}$

- **补码（2 - 补码）**

  由符号位跟上真值取模 2^n ， n 为位数

  $[X_T]_{补} = 2^n + X_T (-2^{n-1} \leq X_T < 2^{n-1},\ \mod\ 2^n)$

  计算时超过表示范围称为 **溢出**

  可以看出补码比原码多表示一个数，这是因为原码 0 的不唯一性

  - **补码与真值的转换**

    真值 -> 补码：

    正数：符号位为 0 ，其余各位同真值

    负数：符号位为 1 ，各位取反，末位加 1

    补码 -> 真值：

    符号位为 0 ： 取其余各位

    符号位为 1 ： 各位取反，末位加 1

  补码的优点：可以将符号位和其他位统一处理。

- **反码（1 - 补码）**

  与补码类似，只是负数取反后不加 1

- **移码**

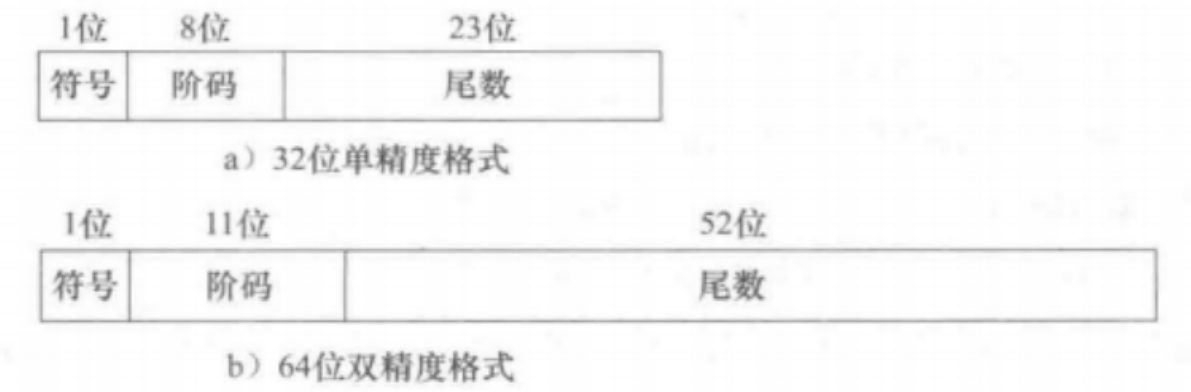
  浮点数表示中，一个定点小数表示尾数，一个定点整数表示阶。浮点数的阶用 **移码** 表示，阶的编码称为 **阶码** 。

  阶可正可负，在浮点数运算时必须先“对阶”，使两个阶的大小相等，为了简化比较操作，使之不涉及符号问题，对每个阶加上一个 **偏置常数** 。（移码的优点）

  $[E]_{移} = 偏置常数 + E$ ，通常偏置常数取 2^{n-1} 或 $2^{n-1}-1$ 。所以要算回来其实要减。

  计算一个数的移码就是加上偏置常数然后转二进制，应该是认为没有符号位存在。

IEEE 754 浮点数标准



- **规格化尾数**

  就是使得第一位总为 1，使得尾数可表示的位数多了一位

  要注意的是，**IEEE 754 规定隐藏 1 的位置在小数点之前**

  例如 0.101，尾数其实是 01，虽然其表示的定点小数为 1.01。这很好理解，就是尾数省略了最前面的一个 1，增加了精度

- **偏置常数**

  可以看到是 $2^{n-1}-1$，因此可表示范围更大（我也不知道为什么）

- **特殊规定**

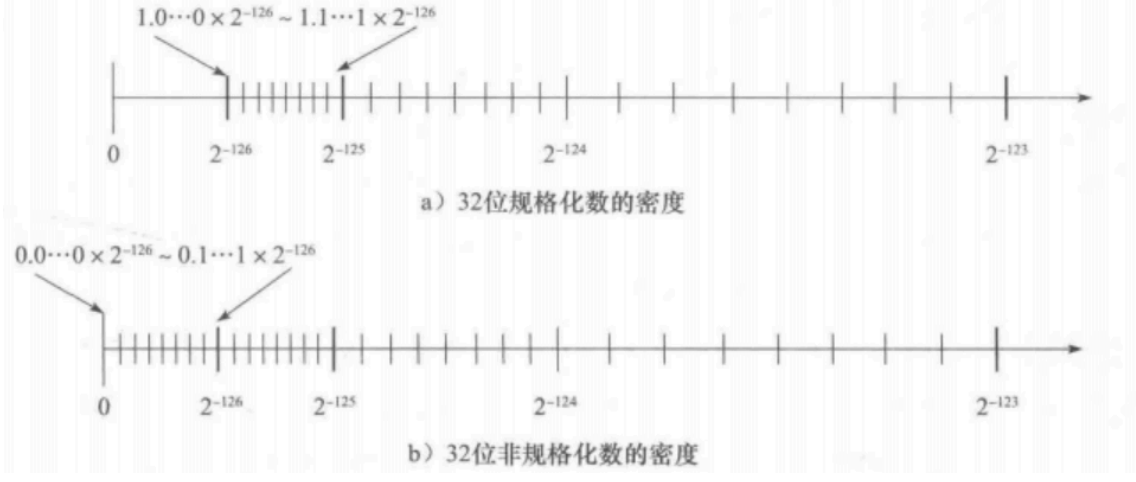
  可以主要到一些神秘数字因为规格化没法表示了，更何况一些本来无法表示的数，所以规定如下：

表 2.2　IEEE 754 浮点数的解释

值的类型	单精度（32 位）				双精度（64 位）			
	符号	阶码	尾数	值	符号	阶码	尾数	值
正零	0	0	0	0	0	0	0	0
负零	1	0	0	−0	1	0	0	−0
正无穷大	0	255（全 1）	0	∞	0	2047（全 1）	0	∞
负无穷大	1	255（全 1）	0	−∞	1	2047（全 1）	0	−∞
无定义数（非数）	0 或 1	255（全 1）	≠0	NaN	0 或 1	2047（全 1）	≠0	NaN
规格化非零正数	0	$0<e<255$	f	$2^{e-127}(1.f)$	0	$0<e<2047$	f	$2^{e-1023}(1.f)$
规格化非零负数	1	$0<e<255$	f	$-2^{e-127}(1.f)$	1	$0<e<2047$	f	$-2^{e-1023}(1.f)$
非规格化正数	0	0	$f\neq0$	$2^{-126}(0.f)$	0	0	$f\neq0$	$2^{-1022}(0.f)$
非规格化负数	1	0	$f\neq0$	$-2^{-126}(0.f)$	1	0	$f\neq0$	$-2^{-1022}(0.f)$

> 这个表应该不至于夸张到考吧，大概了解一下

非规格化是为了处理规格化不能表示的过于小的数。



至于为什么是 2^{-126} ，这个图给出了一定的解释，因为要和规格化数连在一起，，，？

这部分有点太夸张了，我觉得大致了解一下就行了，，，

浮点数的精度问题

- **从 int 转换为 float 时，不会发生溢出，但有效数字可能被舍去**

  int 的 31 位 比 float 尾数 的 23 位 多

- **从 int float 转换为 double 时，因为 double 的有效位数更多，故能保留精确值**

  double 有效位数更多

- **从 double 转换为 float 时，因为 float 表示范围更小，故可能发生溢出，此外，由于有效位数变少，故数据可能被舍入**

- **从 float double 转换为 int 时，因为 int 没有小数部分，所以数据可能会向 0 方向被截断**

  例如，1.9999 被转换为 1 ，-1.9999 被转换为 -1 ，此外，因为 int 的表示范围更小，故可能发生溢出，将大的浮点数转换为整数可能会导致程序错误

浮点数的运算

- **加减法**

  对阶、尾数加减、规格化、舍入

  - **对阶**

    小阶向大阶看齐，阶小的那个尾数右移

  - **尾数加减**

    字面意思，本质上是定点原码小数的加减运算

  - **规格化**

    右规：尾数右移一位，阶码加 1

    左规：尾数左移一位，阶码减 1

    总之就是让尾数表示的数变成 $1.bbbb\cdots$ 的形式

  - **舍入**

    尾数右边：保护位、舍入位、粘位，右规时保留三位

    粘位：只要舍入位右边有任何非 0 数字则设为 1，否则设为 0

    1. **就近舍入到偶数**

       这里的偶数应该指的是尾数最后一位的奇偶性。

    2. **朝 $+\infty$ 方向舍入**
    3. **朝 $-\infty$ 方向舍入**
    4. **朝 0 方向舍入**

  - **阶码溢出**

    上溢：设置为 $\pm\infty$

    下溢：阶码减一变为 0 时，尾数不变，保持非规格化形式。

- **乘除法**

  无需对阶，尾数乘除，阶数加减，然后规格化、舍入，直接算。

整数的运算

- **加减**

  直接运算，根据 OF 判断是否溢出

  带符号整数和无符号整数都是在同一个整数加减运算部件中进行运算的。等效于把两个 int 类型的数转为 unsigned 然后运算再转回来

- **乘除**

  - **无符号数**

    两个 n 位的直接乘出一个 $2n$ 位的，然后截断，如果高 n 位全零则结果正确未溢出

  - **有符号数**

    直接算出正确结果，转成 $2n$ 位的补码，然后截断，如果高 n 位全都与低 n 位的最高位相同，则结果正确未溢出

    > 书上讲的二臂吧

常见数值数据类型的宽度

C 声明	典型的 32 位 机器	64 位 机器
char	1	1
short int	2	2
int	4	4
long int	4	8
char *	4	8
float	4	4
double	8	8

> 应该所有指针的字节数都是一样的?

大小端方式

在所有计算机中，多字节数据都被存放在连续地址中，根据数据各字节在连续地址中排列顺序的不同，可有两种排列方式：大端 (big endian) 和小端 (little endian)

图 2.6　大端方式和小端方式

- **大端方式**

  最高有效字节存放在小地址单元，最低有效字节存放在大地址单元

  从小到大存储

- **小端方式**

  最高有效字节存放在大地址单元，最低有效字节存放在小地址单元

  从大到小存储

> x86 是小端方式

对齐

- **什么是对齐?**
  数据对齐是指将数据存储在满足特定内存地址边界（如 2 字节、4 字节、8 字节）的地址上。比如，4 字节对齐时，数据的起始地址必须是 4 的倍数。
- **为什么要对齐?**
  1. **提高访问效率**：对齐的数据能被 CPU 在一次内存操作中直接访问，未对齐的数据可能需要多次访问。
  2. **硬件要求**：某些处理器（如 ARM）不支持未对齐数据，强制要求数据对齐，否则可能报错。
  3. **跨平台兼容性**：对齐规则有助于提高程序在不同系统上的兼容性。

> 这几把书上也没有啊，问 chatgpt 的，稍微看下吧

各种标志位

- **零标志 ZF(Zero Flag)**

  0 标志，ZF = 1 表示结果所有位都为 0 ，**有无符号都有意义**

- **溢出标志 OF(Overflow Flag)**

  当**带**符号整数 X, Y 的最高位相同且不同于结果的最高位时，OF = 1

- **符号标志 SF(Sign Flag)**

  表示**带**符号整数加减运算结果的符号位，即结果的最高位

- **进/借位标志 CF(Carry Flag)**

  表示**无**符号整数加减运算时的进/借位。加法时 CF 等于进位输出 C，减法时应将进位输出 C 取反来作为借位标志。

  减法时，$CF = 1$ 等效于 $A < B$

> 考后总结：印象里考了几种定点数的表示，标志位的概念，IEEE 754 浮点数的表示之类的（但是比较浅，就是给你一个数让你用双精度表示出来），感觉这块还是得从头重点理解一下，内容还是比较多的，不懂就看书

第三章

IA-32中通用寄存器的名称、宽度，注意EAX、AX、AH、AL的位宽差别；

IA-32中EFLAGS、EIP的功能；

IA-32中各类寻址方式及有效地址的计算方式；

了解IA-32中各类传输指令功能，包括MOV、PUSH/POP、LEA、PUSHF/POPF等；

了解IA-32中各类定点算术运算指令的功能，包括ADD/SUB，INC/DEC、NEG、CMP、MUL/DIV、IMUL/IDIV；

了解IA-32中各类逻辑运算指令的功能，包括NOT、AND、OR、XOR、TEST等；

了解IA-32中各类移位指令的功能，包括SHL/SHR、SAL/SAR等。

了解IA-32中各类控制转移指令的功能，包括JMP、Jcc等；

了解IA-32中调用和返回指令的功能，包括CALL、RET等。

※理解C语言中过程调用的执行步骤；

※理解堆、栈、栈帧的概念，能画出过程调用中栈帧的变化；

※牢记EAX是过程调用return的返回值；

※牢记在栈帧中EBP+8一般是指向第一个入口参数，EBP+4指向返回地址，EBP指向EBP的旧值，EBP-4指向被调用者的第一个非静态局部变量；

理解过程调用中按值传递参数和按地址传递参数的本质区别；

※掌握常用C语言与汇编语言之间的对应关系：if else；switch；条件表达式；while；for等。

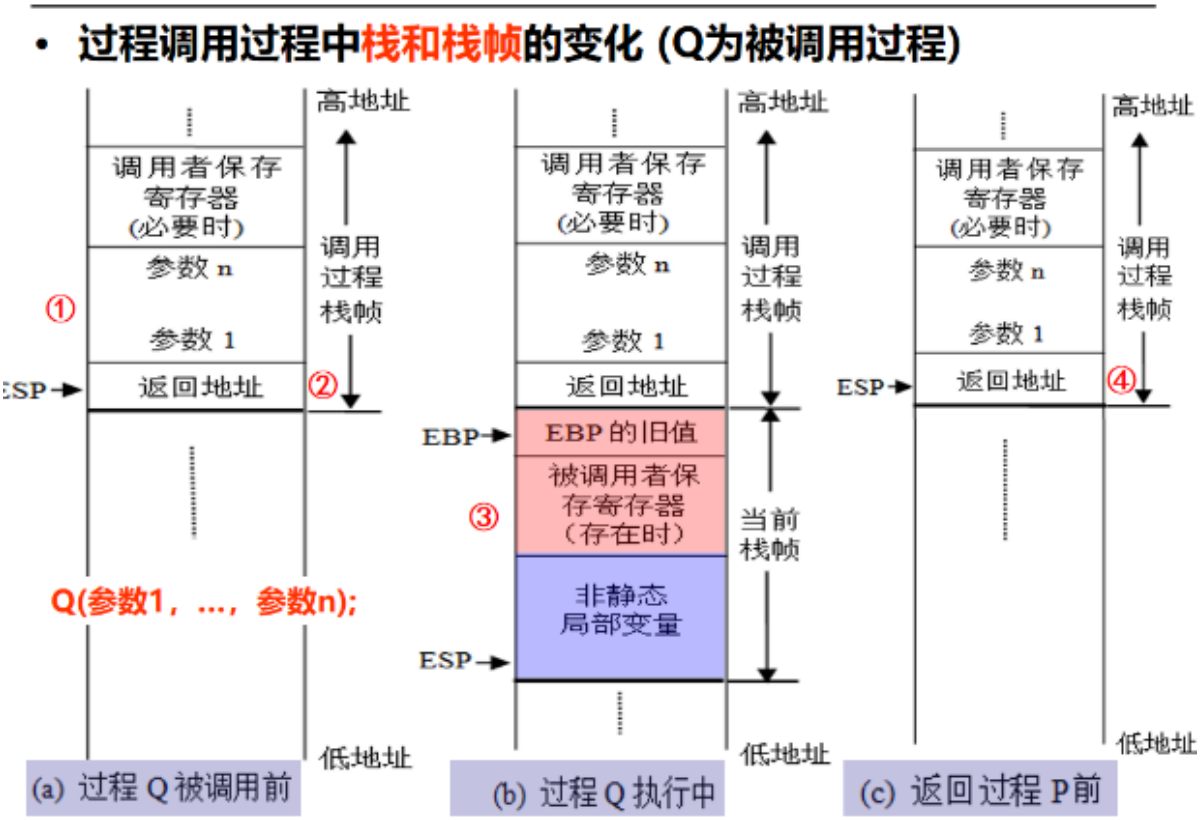
※难点：能在C语言与汇编语言之间做手动转换，重点是读懂汇编程序，反汇编为对应的C代码。也能够根据C语言，手动汇编为对应的汇编程序——要求较低。参考作业题：13和14等。

了解数组元素、结构体数据、联合体数据在存储空间中的存放方式及访问机制。

越界访问和缓冲器溢出攻击机器及其防范方法。

> 这是人？？？？？？？我觉得写不完了，自己去翻书吧，感觉这个部分理解起来非常吃力，还是要自己读书慢慢看，当时我第一遍看了好久（其实当时四章只学了第三章）大概理解一下，再加上考试貌似会给出汇编指令解释，应该不会很难吧（希望如此）。

i386 System V ABI 规范规定，寄存器 EAX、ECX 和 EDX 是**调用者保存寄存器**（caller saved register）。当过程 P 调用过程 Q 时，Q 可以直接使用这三个寄存器，不用将它们的值保存到栈中，这也意味着，如果 P 在从 Q 返回后还要用这三个寄存器的话，P 应在转到 Q 之前先保存它们的值，并在从 Q 返回后先恢复它们的值再使用。寄存器 EBX、ESI、EDI 是**被调用者保存寄存器**（callee saved register），Q 必须先将它们的值保存到栈中再使用它们，并在返回 P 之前先恢复它们的值。还有另外两个寄存器 EBP 和 ESP 则分别是帧指针寄存器和栈指针寄存器，分别用来指向当前栈帧的底部和顶部。



> 考后总结：这块主要是要会读汇编和对过程调用要有理解，考的一题是汇编转 C，一题是一个函数调用时具体栈和栈帧的变化，两个事情都是比较折磨的，建议硬嗑课后题，不懂的看书或者网课
>
> 读汇编的话每一条指令对应含义会在试卷后标明，这很良心，极大减少了要记忆的点
>
> 要求是比较低的，课后题一定要看懂，就够了
>
> 我听说如果自己认真做计基实验对做这章的题很有帮助

第四章

强弱符号

强、弱符号的定义如下：函数名和已初始化的全局变量名是强符号，未初始化的全局变量名是弱符号。

规则 1： 强符号不能多次定义，也即强符号只能被定义一次，否则链接错误

规则 2：若一个符号被说明为一次强符号定义和多次弱符号定义，则按强符号定义为准

规则 3：若有多个弱符号定义，则任选其中一个

.text： 目标代码部分

.rodata： 只读数据

.data： 已初始化的全局变量

.bss： 未初始化的全局变量。

链接器的作用是什么?

ELF可重定位目标文件的格式。

ELF可执行文件的格式。

符号表和符号解析。

重定位过程。

静态链接与动态链接的概念。

> 考后感言：这章没认真学，当时马上就考试了随便看了几眼几个概念就上了，记得考了强弱符号，全局、外部、本地符号，以及课后题最后一题的改编，很综合的一道题，我做出来一个一看就是错的答案（