考核方式

习题、小测验等平时成绩 10%

Lab实验 20%

Project 40%

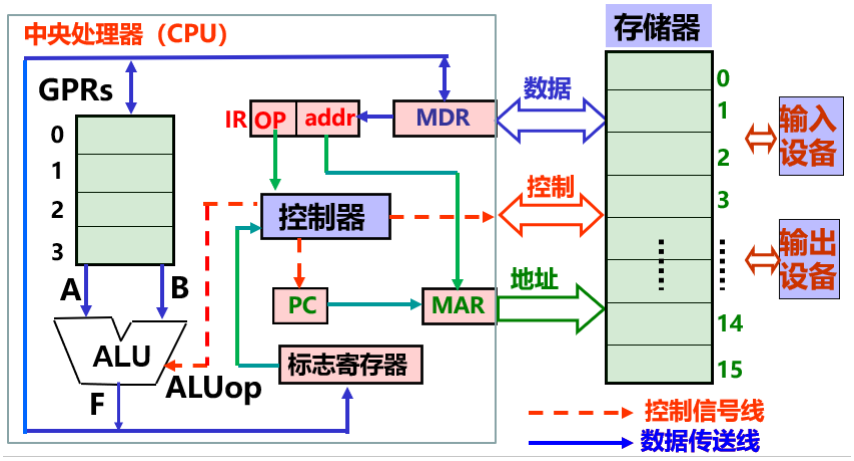
期末考试（开卷） 30%

Chapter1 绪论

CPU：中央处理器；PC：程序计数器；MAR：存储器地址寄存器

ALU：算术逻辑部件；IR：指令寄存器；MDR：存储器数据寄存器

GPRs：通用寄存器组（由若干通用寄存器组成，早期就是累加器）



开始执行程序

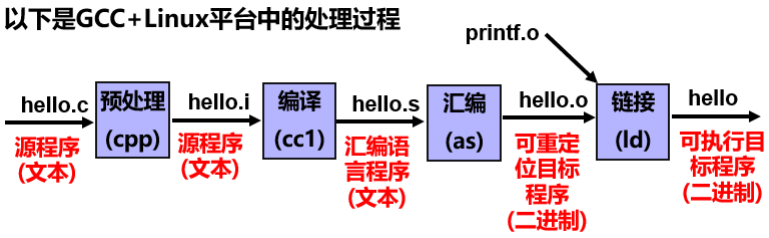
第一步：根据PC取指令 第二步：指令译码 第三步：取操作数

第四步：指令执行 第五步：回写结果 第六步：修改PC的值

继续执行下一条指令

怎样区分读的是**指令还是数据**：只在取指令时，读到的才是**指令**

程序的开发和执行过程：



I/O采用异步方式：（1）外设是慢速设备；（2）执行命令时钟周期的不确定性

ISA指**Instruction Set Architecture**，即**指令集体系结构**，ISA是一种规约，它规定了如何使用硬件

ISA是硬件（计算机组成）对外界的接口，计算机组成是ISA的实现！

不同ISA规定的指令集不同，如，IA-32、MIPS、ARM等

计算机组成必须能够实现ISA规定的功能，如提供GPR、标志、运算电路等

同一种ISA可以有不同的计算机组成，如乘法指令可用ALU或乘法器实现

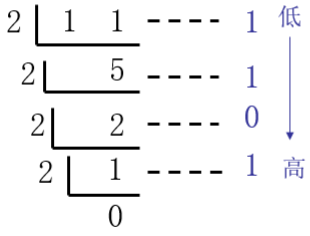
Chapter2 数据的机器级表示和处理

第一讲：数值数据的表示

数值数据表示的三要素：进位计数制，定/浮点表示，定点数的编码

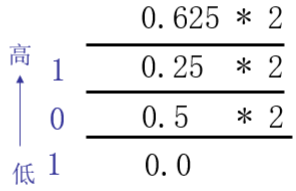
进制转换：

- **十进制转二进制**
  - **整数部分除2取余**



除尽为止 1011

小数部分乘2取整



求得位数满足要求为止

补码 - 模运算：在一个模运算系统中，一个数与它除以"模"后的余数等价。

结论1：一个负数的补码等于模减该负数的绝对值。

结论2：对于某一确定的模，某数减去小于模的另一数，总可以用该数加上另一数负数的补码来代替。

结论3：二进制中，一个负数的补码等于对应正数补码的"各位取反、末位加1"

一般在全部是正数运算且不出现负值结果的场合下，可使用无符号数表示。例如，地址运算，编号表示，等等

无符号整数的编码中没有符号位，能表示的最大值大于位数相同的带符号整数的最大值

常量的默认类型

- **C90**

范围	类型
$0\sim2^{31}-1$	int
$2^{31}\sim2^{32}-1$	unsigned int
$2^{32}\sim2^{63}-1$	long long
$2^{63}\sim2^{64}-1$	unsigned long long

- **C99**

范围	类型
$0\sim2^{31}-1$	int
$2^{31}\sim2^{63}-1$	long long
$2^{63}\sim2^{64}-1$	unsigned long long

C语言默认类型转换顺序（32位）

高　　double　←←　float

↑　　↑

↑　　unsigned long long

↑　　↑

↑　　long long

↑　　↑

↑　　unsigned int（unsigned long）

↑　　↑

低　　int（long）　←←　char, short，unsigned short

float计算前必然转换为double；

char，short计算前必然转换为int；

Exponent	Significand	Object
0	0	+/-0
0	nonzero	Denorms
1-254	anything implicit leading 1	Norms
255	0	+/- infinity
255	nonzero	NaN

单精度浮点数的有效位数为7，双精度浮点数的有效位数为17

第二讲：非数值数据的表示、数据的存储

西文：ASCII码、ISO 8859、ISO/IEC 10646 (Unicode)：UTF - 8变长：1，2，3，4；UTF - 16变长：2，4。

中文：GB2312、CJK(中日韩)统一汉字字符集、GBK、GB18030（单，双，四）

输入码：对汉字用相应按键进行编码表示，用于输入

内码：用于在系统中进行存储、查找、传送等处理

字模点阵或轮廓描述: 描述汉字字模点阵或轮廓，用于显示/打印

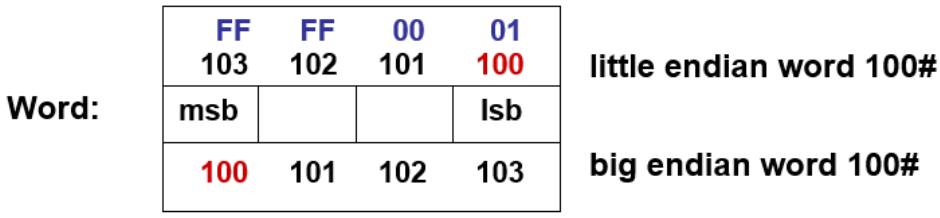
”字长”等于CPU内部总线的宽度、运算器的位数、通用寄存器的宽度等。

C语言中数值数据类型的宽度 (单位：字节)

C声明	典型32位机器	Compaq Alpha机器
char short int int long int	1 2 4 4	1 2 4 8
char*	4	8
float double	4 8	4 8

65535=2^{16}-1

$[-65535]_{补}$ = FFFF0001H



大端方式（Big Endian）：MSB所在的地址是数的地址（MSB在小地址）

小端方式（ Little Endian）：LSB所在的地址是数的地址（LSB在小地址）

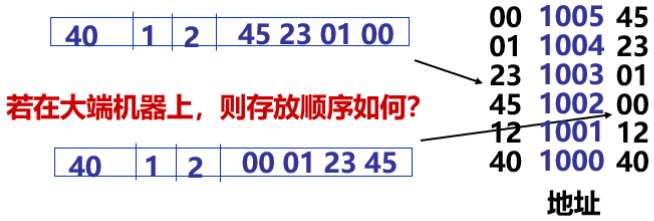
对于指令：**只需要考虑指令中立即数的顺序！**

BIG Endian versus Little Endian

Ex3: Memory layout of an instruction located in 1000

假定小端机器中指令：mov AX, 0x12345(BX)

其中操作码mov为40H，寄存器AX和BX的编号分别为0001B和0010B，立即数占32位，则存放顺序为：



只需要考虑指令中立即数的顺序！

第三讲：数据的运算

按位运算

用途：对位串实现“掩码”（mask）操作或相应的其他处理（主要用于对多媒体数据或状态/控制信息进行处理）

操作：按位或：“|”，按位与：“&”，按位取反：“~”，按位异或：“

逻辑运算

用途：用于关系表达式的运算

操作：“ll”表示“OR”运算，“&&”表示“AND”运算，“!”表示“NOT”运算

移位运算

用途：提取部分信息，扩大或缩小数值的2、4、8...倍

操作：左移：x<<k，右移： x>>k（不区分是逻辑移位还是算术移位，由x的类型确定）

无符号数：逻辑左移、逻辑右移

高（低）位移出，低（高）位补0，可能溢出！

若高位移出的是1，则左移时发生溢出

带符号整数：算术左移、算术右移

左移：高位移出，低位补0。可能溢出！

溢出判断：若移出的位不等于新的符号位，则溢出。

右移：低位移出，高位补符，可能发生有效数据丢失。

位扩展和位截断运算

用途：类型转换时可能需要数据扩展或截断

操作：没有专门操作运算符，根据类型转换前后数据长短确定是扩展还是截断

扩展：短转长

无符号数：0扩展，前面补0

带符号整数：符号扩展，前面补符

截断：长转短

强行将高位丢弃，故可能发生“溢出”

扩展时，”听右边的“！右边unsigned补0，signed补1

n位整数加/减运算器

无符号整数和带符号整数的加、减运算电路完全一样，这个运算电路称为整数加减运算部件，基于带标志加法器实现。最基本的加法器，因为只有n位，所以是一种模2n运算系统！

重要认识1：计算机中所有算术运算都基于加法器实现！

重要认识2：加法器不知道所运算的是带符号数还是无符号数。

重要认识3：加法器不判定对错，总是取低n位作为结果，并生成标志信息。

OF：若A与B同号但与Sum不同号，则为1；否则为0。**SF**：Sum符号

ZF：如Sum为0，则为1，否则为0。**CF**：Cout ⊕ sub。

带符号溢出：（1）最高位和次高位的进位不同 (2) 和的符号位和加数的符号位不同

无符号减溢出：差为负数，即借位CF=1

无符号减公式：

$$\text{result}=\begin{cases} x-y & (x-y>0) \\ x-y+2^n & (x-y<0) \end{cases}$$

带符号减公式：

$$\text{result}=\begin{cases} x-y-2^n & (2^{n-1}\leqslant x-y) & \text{正溢出} \\ x-y & (-2^{n-1}\leqslant x-y<2^{n-1}) & \text{正常} \\ x-y+2^n & (x-y<-2^{n-1}) & \text{负溢出} \end{cases}$$

无符号数：

发生溢出时，一定满足 result<x and result<y
否则，若$x+y-2^n\geq x$，则 $y\geq 2^n$，这是不可能的！

整数的乘运算

无论是带符号还是无符号乘法，运算低32位都是一样的！是否带符号只影响高32位！

布斯乘法： 部分积公式：$P_i=2^{-1}(P_{i-1}+(y_{i-1}-y_i)X)$

例3.4 已知 $[X]_{补}$ = 1 101， $[Y]_{补}$ = 0 110，要求用布斯乘法计算$[X\times Y]_{补}$。

解： $[-X]_{补}$ = 0 011，布斯乘法过程如下：

P	Y	y_{-1}	说明
0 0 0 0	0 1 1 0	0	设 $y_{-1}=0$，$[P_0]_{补}=0$
			$y_0\ y_{-1}=00$，P、Y 直接右移一位
0 0 0 0	0 0 1 1	0	得$[P_1]_{补}$
+0 0 1 1			$y_1\ y_0=10$，$+[-X]_{补}$
0 0 1 1			P、Y 同时右移一位
0 0 0 1	1 0 0 1	1	得$[P_2]_{补}$
			$y_2\ y_1=11$，P、Y 直接右移一位
0 0 0 0	1 1 0 0	1	得$[P_3]_{补}$
+1 1 0 1			$y_3\ y_2=01$，$+[X]_{补}$
1 1 0 1			P、Y 同时右移一位
1 1 1 0	1 1 1 0	0	得$[P_4]_{补}$

因此，$[X\times Y]_{补}$=1110 1110

X*Y的高n位可以用来判断溢出，规则如下：

（1）无符号：若高n位全0，则不溢出，否则溢出

（2）带符号：若高n位全0或全1且等于低n位的最高位，则不溢出

```
int mul(int x, int y)
{
    int z=x*y;
    return z;
}
```

若x、y和z都改成unsigned类型，则判断方式为

乘积的高n位为全0，则不溢出

如何判断返回的z是正确值？

当 !x || z/x==y 为真时

什么情况下，乘积是正确的呢？

当 $-2^{n-1} \le x*y < 2^{n-1}$ （不溢出）时

即：乘积的高n位为全0或全1，并等于低n位的最高位！

即：乘积的高n+1位为全0或全1

编译器在处理变量与常数相乘时，往往以移位、加法和减法的组合运算来代替乘法运算。

不管是无符号数还是带符号整数的乘法，即使乘积溢出时，利用移位和加减运算组合的方式得到的结果都是和采用直接相乘的结果是一样的。

整数的除运算

对于带符号整数来说，n位整数除以n位整数，除$-2^{n-1}/-1=2^{n-1}$会发生溢出外，其余情况都不会发生溢出

因为商的绝对值不可能比被除数的绝对值更大，因而不会发生溢出，也就不会像整数乘法运算那样发生整数溢出漏洞

整数除法，其商也是整数，所以，在不能整除时需要进行舍入，通常按照朝0方向舍入，即正数商取比自身小的最接近整数（Floor，地板），负数商取比自身大的最接近整数（Ceiling，天板）

为了缩短除法运算的时间，编译器在处理一个变量与一个2的幂次形式的整数相除时，常采用右移运算来实现。

无符号：逻辑右移

带符号：算术右移

无符号数、带符号正整数（地板）：移出的低位直接丢弃

带符号负整数（天板）：加偏移量(2^k-1)，然后再右移 k 位，低位截断（这里K 是右移位数）

浮点数运算

进行尾数加减运算前，必须“对阶”！最后还要考虑舍入

“对阶”操作：目的是使两数阶码相等

小阶向大阶看齐，阶小的那个数的尾数右移，右移位数等于两个阶码差的绝对值

IEEE 754尾数右移时，**要将隐含的“1”移到小数部分，高位补0，移出的低位保留到特定的“附加位”上**

设两个规格化浮点数分别为 $A=M_a \cdot 2^{Ea}$　$B=M_b \cdot 2^{Eb}$,则：

$A \pm B = (M_a \pm M_b \cdot 2^{-(Ea-Eb)}) \cdot 2^{Ea}$　(假设Ea>=Eb)

$A*B = (M_a * M_b) \cdot 2^{Ea+Eb}$　　SP最大指数为多少？ 127！

$A/B = (M_a / M_b) \cdot 2^{Ea-Eb}$　　SP最小指数为多少？ -126！

上述运算结果可能出现以下几种情况：

阶码上溢：一个正指数超过了最大允许值 =〉+∞/-∞/溢出

阶码下溢：一个负指数超过了最小允许值 =〉+0/-0

尾数溢出：最高有效位有进位 =〉右规 尾数溢出，结果不一定溢出

非规格化尾数：数值部分高位为0 =〉左规

右规或对阶时，右段有效位丢失 =〉尾数舍入　运算过程中添加保护位

（假定：Xm、Ym分别是X和Y的尾数，　Xe和Ye 分别是X和Y的阶码 ）

(1) 求阶差：$\Delta e = Ye - Xe$ (若Ye > Xe，则结果的阶码为Ye)

(2) 对阶：将Xm右移Δe位，尾数变为 $Xm*2^{Xe-Ye}$（保留右移部分：附加位）

(3) 尾数加减： $Xm*2^{Xe-Ye} \pm Ym$

(4) 规格化：

当尾数高位为0，则需左规：尾数左移一次，阶码减1，直到MSB为1

每次阶码减1后要判断阶码是否下溢（比最小可表示的阶码还要小）

当尾数最高位有进位，需右规：尾数右移一次，阶码加1，直到MSB为1

每次阶码加1后要判断阶码是否上溢（比最大可表示的阶码还要大）

阶码溢出异常处理：阶码上溢，则结果溢出；阶码下溢，则结果为0

(5) 如果尾数比规定位数长，则需考虑舍入（有多种舍入方式）

(6) 若运算结果尾数是0，则需要将阶码也置0。为什么？

尾数为0说明结果应该为0（阶码和尾数为全0）。

加减运算右规时最多只需一次，符号位单独处理

- 从int转换为float时，不会发生溢出，但可能有数据被舍入
- 从int或 float转换为double时，因为double的有效位数更多，故能保留精确值
- 从double转换为float和int时，可能发生溢出，此外，由于有效位数变少，故可能被舍入
- 从float 或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