[结构组成 2](#_Toc492318249)

[数据传输 4](#_Toc492318250)

[寄存器之间的数据传送 4](#_Toc492318251)

[主存与CPU之间的数据传送 4](#_Toc492318252)

[执行算数运算或逻辑运算 4](#_Toc492318253)

[啊 4](#_Toc492318254)

[运算指令执行流程 5](#_Toc492318255)

[整数 6](#_Toc492318256)

[数据表示与存储 6](#_Toc492318257)

[补码的本质 7](#_Toc492318258)

[整数运算 8](#_Toc492318259)

[定点数 9](#_Toc492318260)

[数据 9](#_Toc492318261)

[二进制表示 9](#_Toc492318262)

[存储 10](#_Toc492318263)

[定点运算 11](#_Toc492318264)

[移位运算 11](#_Toc492318265)

[补码加减法 12](#_Toc492318266)

[乘法运算 13](#_Toc492318267)

[浮点数 14](#_Toc492318268)

[数据 14](#_Toc492318269)

[二进制表示 14](#_Toc492318270)

[存储 15](#_Toc492318271)

[浮点数的IEEE754标准 16](#_Toc492318272)

[浮点运算 17](#_Toc492318273)

# 结构组成

运算器接收从控制器送来的命令并执行相应的动作，对数据进行加工和处理。

|  |  |  |  |
| --- | --- | --- | --- |
| 算数逻辑单元  ALU | 进行算数/逻辑运算  在CPU内部必须给ALU提供数据，因此ALU必须可直接访问MDR和用户可见的寄存器  ALU外围还可以有另一些寄存器，这些寄存器用于ALU的输入/输出以及用于和MDR及用户可见寄存器交换数据 | | |
| 移位器 | 对操作数或运算结果进行移位运算 | | |
| 计数器CT | 控制乘除运算的操作步数 | | |
| 程序状态字寄存器  PSW |  | | |
| 暂存寄存器 | 暂存从主存读来的数据，这个数据不能存放在通用寄存器中，否则会破坏其原有的内容  暂存寄存器对程序员是透明的 | | |
| 通用寄存器 | 累加寄存器  ACC |  | 暂时存放ALU预算的结果信息  作为加法运算的一个输入端 |
| 堆栈指针  SP |  |  |
| AX |  | 存放操作数  存放地址 |
| BX |  |  |
| CX |  |  |
| DX |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | ACC | MQ | X |
| 加法 | 被加数  结果和 |  | 加数 |
| 减法 | 被减数  结果差 |  | 减数 |
| 乘法 | 结果积的高位 | 乘数  结果积的低位 | 被乘数 |
| 除法 | 被除数  余数 | 商 | 除数 |
|  |  |  |  |
|  |  |  |  |

## 工作过程

|  |  |
| --- | --- |
| 加法 | 初态：ACC中已经有了被加数 |
| 从给定的内存单元取出加数，保存到X寄存器  [M]🡪X |
| ALU执行加法操作：  [ACC]+[X]  结果存储到ACC中 |
| 减法 | 初态：ACC中已经有了被减数 |
| 从给定的内存单元取出减数，保存到X寄存器  [M]🡪X |
| ALU执行减法操作：  [ACC]-[X]  结果存储到ACC中 |
| 乘法 | 初态：ACC中已经有了被乘数 |
| 从给定的内存单元取出乘数，保存到MQ寄存器  (1)[M]🡪MQ  (2)[ACC]🡪X  (3)0🡪ACC  (4)[X]×[MQ]🡪ACC//MQ，结果的高位放在ACC，低位放在MQ  1和4 可以互换顺序，可以同时进行  总体执行是由顺序的  顺序有CU控制 |
| 除法 | 初态：ACC中已经有了被除数 |
| 从给定的内存单元取出除数，保存到X寄存器  (1)[M]🡪X  (2)[ACC]÷[X](商)🡪MQ  余数🡪ACC |

MQ：乘商寄存器

# 数据传输

## 寄存器之间的数据传送

通过CPU内部总线完成

某寄存器X的输出和输入分别有Xout和Xin控制

以PC寄存器为例，把PC内容送至MAR，实现传送操作的流程即控制信号为：

PC🡪Bus：PCout有效，PC内容送至总线

Bus🡪MAR：MARin有效，总线内容送至MAR

## 主存与CPU之间的数据传送

以CPU从主存读取指令为例说明数据在数据通路中的传送过程

PC🡪Bus🡪MAR：PCout和MAR有效，现行指令地址🡪MAR

1🡪R：CU发出读命令

MEM(MAR)🡪MDR：MDRin有效

MDR🡪Bus🡪IR：MDRout和IRin有效，现行指令🡪IR

## 执行算数运算或逻辑运算

当执行算术或逻辑操作时，由于ALU本身是没有内部存储功能的组合电路，因此如果要执行加法运算，被相加的两个数必须在ALU的两个输入端同时有效

暂存器Y及用于该目的

现将一个操作数经CPU内部总线送入暂存器Y保存起来，Y的内容在ALU的左输入端始终有效，再将另一个操作数经总线直接送到ALU的右输入端。这样两个操作数都送入了ALU，运算结果暂存在暂存器Z中

Ad(IR)🡪Bus🡪MAR：MDRout和MARin有效

1🡪R：CU发出读命令

MEM🡪数据线🡪MDR：操作数从存储器🡪数据线🡪MDR

MDR🡪Bus🡪Y：MDRout和Yin有效，操作数🡪Y

(ACC)+Y🡪Z：ACCout和ALUin有效，CU向ALU发加命令，结果🡪Z

Z🡪ACC：Zout和ACCin有效，结果🡪ACC

数据通路结构直接影响CPU内各种信息的传送路径，数据通路不同，指令执行过程的微操做序列的安排也不同，它关系着微操做信号形成部件的设计。

## 啊

CPU的读写控制信号的作用是决定是从存储器读还是向存储器写如：

决定数据总线上的数据流向

控制存储器操作的读写类型

控制流入、流出存储器的信息的方向

## 运算指令执行流程

加法运算指令执行过程：ADD (R1),R0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 时钟周期 | 数据流 | | | 其他操作 |
| 信号内容 | | 传输路径 |
| C5 | 控制信号 | R1\_out有效 | CU🡪片内总线🡪R1 |  |
| MAR\_in有效 | CU🡪片内总线🡪MAR |
| 主存地址 | (R1) | R1🡪片内总线🡪MAR |
| C6 | 控制信号 | MDR\_in有效 | CU🡪片内总线🡪MDR |  |
| 读命令  (包含MAR中的信息) | CU🡪系统控制总线🡪主存 |
| 主存数据 | ((R1)) | 主存🡪系统数据总线🡪MDR |
| C7 | 控制信号 | MDR\_out有效 | CU🡪片内总线🡪MDR | ALU一端是寄存器A，MDR或R0中必须有一个先写入A中  如MDR |
| A\_in有效 | CU🡪片内总线🡪A |
| 寄存器数据 | ((R1)) | MDR🡪A |
| C8 | 控制信号 | R0\_out有效 | CU🡪片内总线🡪R0 | 执行加法操作 |
| ACC\_in有效 | CU🡪片内总线🡪ACC |
| 寄存器数据 | (R0) | R0🡪片内总线🡪ACC |
| 控制信号 | Add | CU🡪片内总线🡪ACC |
| 寄存器数据 | (A)+(ACC) | ACC |
| C9 | 控制信号 | ACC\_out有效 | CU🡪片内总线🡪ACC |  |
| MDR\_in有效 | CU🡪片内总线🡪MDR |  |
| 寄存器数据 | (ACC) | ACC🡪片内总线🡪MDR |  |
| C10 | 控制信号 | MDR\_out有效 | CU🡪片内总线🡪MDR | 加法结果写回到R1的内容所指的主存单元， MAR中的内容没有改变 |
| 写命令  (包含MAR中的信息) | CU🡪系统控制总线🡪主存 |
| 寄存器数据 | (MDR) | MDR🡪系统数据总线🡪主存 |

指令 的操作数一个在主存中，一个在寄存器中，运算结果在主存中。

根据指令功能，要读出R1的内容所指的主存单元，必须先将R1的内容送至MAR，即(R1)🡪MAR。

而读出的数据必须经过MDR，即M(MAR)🡪MDR

有效控制信号的安排并不难，只需要看数据是流入还是流出，如果流入寄存器X就是Xin，流出寄存器X就是Xout

还需注意特殊控制信号如：PC+1、Add等

整数

# 数据表示与存储

|  |  |  |  |
| --- | --- | --- | --- |
| 整数 | 计算机的**整数数据类型**定义的位长：  又称为计算机的机器字长，即计算机一个存储单元的大小  这是由计算机的硬件决定的一个固有属性 | | |
| 无符号整数 | 二进制表示 | 无符号整数的表示方法只有一种  直接将数据按照机器字长展开为二进制数 | |
| 存储 | 存储单元的每一位，bit[N-1:0]，都用来存放数值  如机器字长为16位，那么存储的范围是00…0B到11…1B，  其中00…0B表示最小值0，11…1B表示最大值65535；所以无符号整数范围是0~65535 | |
| 有符号整数 | 二进制表示 | 原码 | 符号位：最高位，0表示正，1表示负  数据位：其余位，数据的绝对值按照机器字长展开为二进制数，作为数据位  0有两种表示形式： |
| 反码 | 正数：同原码  负数：符号位=1，数据位是原码的数据位各位取反  0有两种表示形式： |
| 补码 | 正数：同原码  负数：反码加1  所以 ，0的补码只有一种表示形式 |
| 移码 | 符号位：取反  数据位：同补码 |
| 存储 | 存储单元的最高位，bit[N-1]，存储符号位  存储单元的其余位，bit[N-2:0]，存储补码的数据位  如机器字长为16位，那么存储的范围是00…0B到11…1B，  10…0B表示最小值-32768，-32768的补码其实是1 1000 0000 0000 0000，最高位溢出  10…1B表示-32767，……，11…1B表示-1，00…0B表示0  00…1B表示最大值32767，  00…2B表示32766，……，01…1B表示1  所以有符号整数范围是-32768~32767 | |

## 补码的本质

|  |  |  |
| --- | --- | --- |
|  | 概念 | 计算方法 |
| 补数 | 补数的本质是一个循环  确定一个模值n，那么所有的数的补数都在范围[0,n]内，  并且，这是一个循环的范围，即上限与下限是相连的，所以  如： | 由此得到补数的计算方法：  一个负数对应一个正补数，  一个正数的补数是它的本身，  0的补数是模值  如： |
| 补码 | 计算机中的补码是用到了数学中的补数的概念 |  |

# 整数运算

定点数

# 数据

## 二进制表示

|  |  |  |
| --- | --- | --- |
| 采用定点数的计算机称为定点机 | 数符 |  |
| 数值部分 |  |

|  |  |
| --- | --- |
| 纯小数 | 小数点在数符与数值部分中间 |
| 纯整数 | 小数点在数值部分的后面 |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| 原码 | 符号位：最高位，0表示正，1表示负  小数点：在最高位的后面  数据位：其余位，小数位数值按照机器字长展开为二进制数，作为数据位 |
| 反码 | 符号位：不变  小数点：在最高位的后面  数据位：其余位，原码的数据位的各位按位取反，作为反码的数据位 |
| 补码 | 符号位：不变  小数点：在最高位的后面  数据位：其余位，原码的数据位取反，然后加1，作为补码的数据位 |
| 移码 | 符号位：取反  小数点：在最高位的后面  数据位：其余位，原码的数据位取反，然后加1，作为补码的数据位 |

## 存储

|  |  |
| --- | --- |
| 存储结构 | 存储单元的最高位，bit[N-1]，存储符号位  存储单元的最低位，bit[N-2:0]，存储小数部分的最高位  小数点固定在存储单元的最低位bit0的后面 |
| 范围 | ，N是机器位长 |

# 定点运算

## 移位运算

|  |  |  |  |
| --- | --- | --- | --- |
|  | 二进制表示的机器数：  小数点左移n位：该数乘以  小数点右移n位：该数除以 | | |
| 移位规则 | 正数 | 数符：不变  数值部分：溢出的数值丢弃，空出的位置补0 | |
| 负数 | 原码 | 数符：不变  数值部分：溢出的数值丢弃，空出的位置补0 |
| 补码 | 数符：不变  数值部分：溢出的数值丢弃，空出的低位补0，空处的高位补1 |
| 反码 | 数符：不变  数值部分：溢出的数值丢弃，空出的位置补1 |

## 补码加减法

|  |  |  |
| --- | --- | --- |
| 公式 | 整数： （mod ）  小数： （mod ） | |
| 溢出判断 | 用一位符号位 | 对于加法，仅当两个符号位相同的数相加时，才可能溢出  对于减法，仅当两个符号位相异的数相减时，才可能溢出  由于减法运算使用加法器实现的，所以可以得出总结：  加法器中参与运算的两个操作数符号数相同，结果的符号数又与原操作数不同，判为溢出 |
| 用两位符号位 |  |
| 硬件配置 |  | |
| 控制流程 |  | |

## 乘法运算

|  |  |  |
| --- | --- | --- |
| 笔算乘法 | 改进前 | 将4个位积一次相加，机器难以实现  乘积位数正常了一倍，这将造成器材的浪费和运算时间的增加。 |
| 改进后 | 使用移位运算和加法运算来实现，  如两个4位数相乘，总共需要进行4次加法运算和4次移位运算  由乘数的末位值确定被乘数是否原部分积相加，然后右移一位面，形成新的部分积，同时，乘数也右移一位，由低次位作新的末位，空出的最高位放部分积的最低位  每次做加法时，被乘数仅仅与原部分积的高位相加，其低位被移至乘数所空出的高位位置。 |
| 原码乘法 | 源码一位乘 |  |
| 源码两位乘 |  |
| 补码乘法 |  | |
|  |  | |

浮点数

# 数据

## 二进制表示

|  |  |  |
| --- | --- | --- |
| S× | S：尾数 | 0.1\*\*\* |
| j：阶码 |  |
| r：基数 | 基数取2的幂，如2、4、8、16…  查看下面的浮点数存储可以看到并没有存储基数，因为基数的取值与计算机相关，浮点机的基数一旦确定后，就不再改变；  所以，不同基数的浮点数存储形式是完全相同的。  基数越大，浮点数的精度越差； |
|  |  |

## 存储

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 存储结构 |  | 名称 | | 符号 | 位长 | 说明 |
| bit[N-1] | 阶码 | 阶符 |  | 1 |  |
| bit[N-2:N-m-1] | 数值 |  | m=N-n-2 | 阶码的数值部分 |
| bit[n-2] | 尾数 | 数符 |  | 1 | 表示尾数的正负 |
| bit[n-1:0] | 数值 |  | n=N-m-2 | 尾数的数值部分  bit[n-1]=1 |
| 精度 |  | | | | | |
| 范围 | 正数范围：  负数范围：  基数越大，可表示的浮点数范围越大，且表示的数的个数越多 | | | | | |

|  |  |
| --- | --- |
| 尾数规格化 | 尾数必须为纯小数：即尾数范围是[0,1]  尾数必须为规格化数：尾数的小数部分中的最高位是1  如果不满足上述条件，就要对尾数进行规格化，即将非规格化数转换成规格化数的过程 |
| …… |

## 浮点数的IEEE754标准

现代计算机中的浮点数一般采用IEEE指定的国际标准，这种标准形式如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 位长 | 符号位 | 阶码 | 尾数 |
| 短实数 | 32 | 1 | 8 | 23 |
| 长实数 | 64 | 1 | 11 | 52 |
| 临时实数 | 80 | 1 | 15 | 64 |

短实数：1位符号位S，8位阶码，23位尾数，总位数=32

长实数：1位符号位S，8位阶码，23位尾数，总位数=32

临时实数：1位符号位S，8位阶码，23位尾数，总位数=32

# 浮点运算