计算机组成原理

目录

概述

软件基于数据结构，数据结构又基于计算机，计算机由操作系统运行，计算机的硬件基于计算机组成原理，计算机之间通过计算机网络连接，这就是四个的关系。计算机组成原理是关于计算机硬件的最底层知识。

- 计算机系统=硬件+软件。
- 软件分为：
  - 系统软件：用来管理整个计算机系统，如操作系统、数据库管理系统（$DBMS$）、标准程程序库、网络软件、语言处理程序、服务程序。
  - 应用软件：按任务需要编制成的各种程序。

计算机的发展

硬件的发展

发展阶段	时间	逻辑元件	速度（次/秒）	内存	外存
第一代	1946-1957	电子管	几千-几方	汞延迟线、磁鼓	穿孔卡片、纸带
第二代	1958-1964	晶体管	几万-几千万	磁芯存储器	磁带
第三代	1964-1971	中小规模集成电路	几十万-几百万	半导体存储器	磁带、磁盘
第四代	1972-现在	大规模、超大规模集成电路	上千万-万亿	半导体存储器	磁盘、磁带、光盘、半导体存储器

- 第一代使用纸带磁带编程。
- 第二代出现了面向过程的程序设计语言$FORTRAN$，有了操作系统雏形。
- 第三代主要用于科学计算等专业用途，高级语言快速发展，开始有了分时系统。
- 第四代开始出现CPU、PC，如$Windows$、$MacOS$等。

软件的发展

- 机器语言：唯一可以被计算机识别和执行的语言。
- 汇编语言：必须经过汇编程序的翻译。
- 高级语言：转换为汇编程序或直接翻译为机器语言。

计算机的分类

电子计算机分为：

- 电子模拟计算机。

- 电子数字计算机：
  - 专用计算机。
  - 通用计算机：巨型机、大型机、中型机、小型机、微型机、单片机。

按指令和数据流分为：

- 单指令流和单数据流系统（$SISD$），即传统冯·诺依曼体系结构。
- 单指令流和多数据流系统（$SIMD$），包括阵列处理器和向量处理器系统。
- 多指令流和单数据流系统（$MISD$），这种计算机实际上不存在。
- 多指令流和多数据流系统（$MIMD$），包括多处理器和多计算机系统。

计算机系统层次结构

计算机结构

冯诺伊曼结构

1. 计算机由五大部件组成：
   - 输入设备：将信息转换成机器能识别的形式。
   - 存储器：存放数据和程序。
   - 运算器：算术运算和逻辑运算。
   - 控制器：协调其他部件与解析存储器中的程序或指令。
   - 输出设备：将结果转换为人类熟悉的形式。
2. 指令和数据以同等地位存于存储器，可按地址寻访。
3. 指令和数据用二进制表示。
4. 指令由操作码（指令序列号，用来表示处理的指令）和地址码（操作数据存储的地址）组成。
5. 存储程序：指将指令以二进制代码的形式事先输入计算机的主存储器，然后按其在存储器中的首地址执行程序的第一条指令，以后就按该程序的规定顺序执行其他指令，直至程序执行结束。
6. 以运算器为中心：输入/输出设备与存储器之间的数据传送通过运算器完成。

现代计算机结构

1. 计算机由两个部分组成：
   - 主机：
     - CPU：
       - 运算器。
       - 控制器。
     - 主存。
   - I/O设备：
     - 辅存。

- 输入设备。
- 输出设备。

2. 以存储器为中心。

硬件构成

I/O 设备

- 输入设备：将程序和数据以机器所能识别和接受的信息形式输入计算机。
- 输出设备：将计算机处理的结果以人们所能接受的形式或其他系统所要求的信息形式输出

存储器

用于存放程序和数据。分为主存和辅存，CPU直接访问主存，只有调入主存才能被CPU访问。

存储器结构：

- 存储体：存储数据的主体，按地址存取。（相联存储器按内容访存）
- MAR：（$Memory\ Address\ Register$）存储地址寄存器，用于访存地址，经过存放地址译码后所选的存储单元。MAR位数反映存储单元个数的幂值，与PC（程序计数器）长度相等（因为都是指向内存地址），如MAR有10位，则有$2^{10} = 1024$个存储单元。
- MDR：（$Memory\ Data\ Register$）存储数据寄存器，用于暂存要从存储器中读或写的信息。MDR位数=存储字长，一般为字节的二次幂的整数倍。（存储字长不等于数据字长，数据字长是数据总线一次性传输的信息位数）
- 时序控制逻辑：产生存储器操作时所需的各种时序信号。

存储器相关概念：

- 现代MAR和MDR被划分进入了CPU。
- 存储单元：每个存储单元存放一串二进制代码。
- 存储字（$word$）：存储单元中二进制代码的组合，即一个存储元存放的数据。
- 存储字长：存储单元中二进制代码的位数，为$1B$或字节的偶数倍。
- 存储元：即存储二进制的电子元件，每个存储元可存储$1bit$。

寄存器和高速缓冲寄存器$Cache$都集成在CPU上，离CPU越近速度越快，所以存取速度上寄存器>$Cache$>内存。

CPU包括运算器和控制器，不包括存储器（存储器不同于寄存器）。

运算器

用于算术运算和逻辑运算。

- ALU：算术逻辑单元，是核心单元，通过内部复杂的电路实现算数运算、逻辑运算。
- ACC：累加器，用于存放操作数，或运算结果。
- MQ：乘商寄存器，在乘、除运算时，用于存放操作数或运算结果。
- X：操作数寄存器，通用的操作数寄存器，用于存放操作数。
- PSW：程序状态寄存器，也称标志寄存器，用于存放ALU运算得到的一些标志信息或处理机的状态信息，如结果是否溢出、有无产生进位或借位、结果是否为负等。
- 还有变址寄存器（IX），主要用于存放存储单元在段内的偏移量，基址寄存器（BR），用来存放操作数或中间结果，以减少对存储器的访问次数的数据寄存器。这些都不一定具备。

	加	减	乘	除
ACC	被加数、和	被减数、差	乘积高位	被除数、余数
MQ			乘数、乘积低位	商
X	加数	减数	被乘数	除数

控制器

- CU：控制单元，分析指令，给出控制信号。
- IR：指令寄存器，存放当前执行的指令，内容来自主存的MDR。指令中操作码$OP(IR)$送到CU分析指令并发出各种微操作命令序列，地址码$Ad(IR)$送到MAR来取操作数。
- PC：程序计数器，存放下一条要指令地址，取指后自动加1（一条指令长度，不一定为1）以形成下一条指令地址的功能，与主存MAR有直接通路。

计算机工作过程

1. 把程序和数据装入主存储器。
2. 将源程序转成可执行问题。
3. 从可执行文件的首地址开始逐条执行指令。

完成一条指令：

1. 取指令$PC \rightarrow MAR \rightarrow M \rightarrow MDR \rightarrow IR$，并调用$PC$加1。
2. 将指令放入IR，并分析指令$OP(IR) \rightarrow CU$。
3. 调用CU协同执行指令$Ad(IR) \rightarrow MAR \rightarrow M \rightarrow MDR \rightarrow ACC$。

其中翻译包括：预处理；编译；汇编；链接四个阶段。

如何区分指令和数据：

- 通过不同的时间段来区分指令和数据，即在取指令阶段（或取指微程序）取出的为指令，在执行指令阶段（或相应微程序）取出的即为数据。

- 如果通过地址来源区分，由PC提供存储单元地址的取出的是指令，由指令地址码部分取出（提供存储单元地址或在指令中给出立即数）的是操作数。

已知：

```
int a=2,b=3,c=1,y=0;
int main(){
    y=a*b+c;
}
```

根据计算过程，a乘b再加上c，简单的高级语言背后的转换为机器语言就变成：

指令地址：

主存地址	操作码	地址码	注释
0	000001	0000000101	取数 a 至 ACC
1	000100	0000000110	乘 b 得 ab，存于 ACC 中
2	000011	0000000111	加 c 得 ab+c，存于 ACC 中
3	000010	0000001000	将 ab+c，存于主存单元
4	000110	0000000000	停机

数据地址，基本的赋值操作：

主存地址	数据值	注释
5	0000000000000010	a=2
6	0000000000000011	b=3
7	0000000000000001	c=1
8	0000000000000000	y=0

分析指令地址对应的五条指令执行流程，基本流程是取指令->获取指令操作码->获取指令地址->根据指令地址取数->放入寄存器->计算：

1. 执行地址为0的指令，取指令1到4，分析指令5，执行指令6到8：
    1. $(PC)=0$：程序计数器指向主存地址为0的指令。
    2. $(PC)\rightarrow MAR$，$(MAR)=0$：程序计数器将当前指向的指令地址通过系统总线传输给存储地址寄存器，从而存储地址寄存器的值现在置为0，表示它要处理的指令的主存地址为0。
    3. $M(MAR)\rightarrow MDR$，$(MDR)=000001\ 0000000101$：根据存储地址寄存器存储的地址0，在存储体中找到对应的指令$000001\ 0000000101$，并把它放入存储数据寄存器中。
    4. $(MDR)\rightarrow IR$，$(IR)=000001\ 0000000101$：将存储数据寄存器存储的指令放入指令寄存器中。

5. $OP(IR) \rightarrow CU$：将指令寄存器的000001操作码传输给控制单元中，控制单元根据操作码知道这是“取数”的指令。
6. $AD(IR) \rightarrow MAR$：控制单元根据取数指令，明白要从存储器中取出一个数，而取出的数的地址就是指令寄存器的0000000101，即取出存储地址为5的数据$a = 2$，将这个地址交给存储地址寄存器负责取出。
7. $M(MAR) \rightarrow MDR$，$(MDR) = 000000\ 0000000010 = 2$：存储体根据存储地址寄存器的0000000101地址找到地址为5的数据2，并把2放到存储数据寄存器中。
8. $(MDR) \rightarrow ACC$，$(ACC) = 000000\ 0000000010 = 2$：将存储数据寄存器的数据放入累加器中，完成0这条指令。

2. 执行地址为1的指令，取指令1到4，分析指令5，执行指令6到10：
   1. $(PC)+= 1$，$(PC) = 1$：程序计数器完成一条指令自动加一，从而现在指向地址为1的指令。
   2. $(PC) \rightarrow MAR$，$(MAR) = 1$：存储地址寄存器的值现在置为1，表示它要处理的指令的主存地址为1。
   3. $M(MAR) \rightarrow MDR$，$(MDR) = 000100\ 0000000110$：在存储体中找到地址为1的指令000100 0000000110，并把它放入存储数据寄存器中。
   4. $(MDR) \rightarrow IR$，$(IR) = 000100\ 0000000110$：将存储数据寄存器存储的指令放入指令寄存器中。
   5. $OP(IR) \rightarrow CU$：将指令寄存器的000100操作码传输给控制单元中，控制单元根据操作码知道这是“乘法”的指令。
   6. $AD(IR) \rightarrow MAR$：控制单元根据乘法指令，在第一条指令中就得到了其中一个乘数 a，现在要另一个乘数b，b的地址就是指令寄存器的0000000110，即取出存储地址为6的数据$b = 3$，将这个地址交给存储地址寄存器负责取出。
   7. $M(MAR) \rightarrow MDR$，$(MDR) = 000000\ 0000000011 = 3$：存储体根据存储地址寄存器的0000000110地址找到地址为6的数据3，并把3放到存储数据寄存器中。
   8. $(MDR) \rightarrow MQ$，$(MQ) = 000000\ 0000000011 = 3$：由于需要乘法操作，所以将存储数据寄存器的3放入乘商寄存器中。
   9. $(ACC) \rightarrow X$，$(X) = 2$：然后把a的值从累加器中放入通用寄存器中。（乘法操作中被乘数放入通用寄存器中，而乘数放入乘商寄存器中）。
   10. $(MQ) * (X) \rightarrow ACC$，$(ACC) = 6$：控制单元通知算术逻辑单元，通过算术逻辑单元对 ab 进行相乘后放入累加器中。如果乘积太大则需要乘商寄存器辅助存储。
3. 执行地址为2的指令，取指令1到4，分析指令5，执行指令6到9：
   1. $(PC)+= 1$，$(PC) = 2$：程序计数器完成一条指令自动加一，从而现在指向地址为 2 的指令。

2. $(PC) \rightarrow MAR$，$(MAR) = 2$：存储地址寄存器的值现在置为2，表示它要处理的指令的主存地址为 2。
3. $M(MAR) \rightarrow MDR$，$(MDR) = 000011\ 0000000111$：根据存储地址寄存器存储的地址2，在存储体中找到对应的指令$000011\ 0000000111$，并把它放入存储数据寄存器中。
4. $(MDR) \rightarrow IR$，$(IR) = 000011\ 0000000111$：将存储数据寄存器存储的指令放入指令寄存器中。
5. $OP(IR) \rightarrow CU$：将指令寄存器的000011操作码传输给控制单元中，控制单元根据操作码知道这是“加法”的指令。
6. $AD(IR) \rightarrow MAR$：控制单元根据取数指令，从存储器中取出指令寄存器的地址为0000000111的数，即取出存储地址为7的数据$c = 1$，将这个地址交给存储地址寄存器负责取出。
7. $M(MAR) \rightarrow MDR$，$(MDR) = 000000\ 0000000001 = 1$：存储体根据存储地址寄存器的0000000111地址找到地址为7的数据1，并把1放到存储数据寄存器中。
8. $(MDR) \rightarrow X$，$X = 000000\ 0000000001 = 1$：将存储数据寄存器的1放入通用寄存器中。（即加法操作中累加器存放被加数，通用寄存器中存放加数）。
9. $(ACC) + (X) \rightarrow ACC$，$(ACC) = 7$：控制单元通知算术逻辑单元，将$ab$与$c$相加并存回累加器中。

4. 执行地址为3的指令，取指令1到4，分析指令5，执行指令6到8：
   1. $(PC)+= 1$，$(PC) = 3$：程序计数器完成一条指令自动加一，从而现在指向地址为 3 的指令。
   2. $(PC) \rightarrow MAR$，$(MAR) = 3$：存储地址寄存器的值现在置为 3，表示它要处理的指令的主存地址为3。
   3. $M(MAR) \rightarrow MDR$，$(MDR) = 000010\ 0000001000$：根据存储地址寄存器存储的地址3，在存储体中找到对应的指令$000010\ 0000001000$，并把它放入存储数据寄存器中。
   4. $(MDR) \rightarrow IR$，$(IR) = 000010\ 0000001000$：将存储数据寄存器存储的指令放入指令寄存器中。
   5. $OP(IR) \rightarrow CU$：将指令寄存器的 000010 操作码传输给控制单元中，控制单元根据操作码知道这是“存数”的指令。
   6. $AD(IR) \rightarrow MAR$，$(MAR) = 0000001000 = 8$：控制单元根据存数指令，根据存储的目的地址为0000001000。
   7. $(ACC) \rightarrow MDR$，$(MDR) = 7$：将运算的结果从累加器中传输为存储数据寄存器中。
   8. $(MDR) \rightarrow$：地址为8的存储单元，$y = 7$：控制单元根据控制总线，将存储数据寄存器中的数据存储到存储地址寄存器中所指向的地址，所以地址码为8的y的值就变成了7。

5. 执行地址为4的指令，取指令1到4，分析指令5，执行指令6：
    1. $(PC)+=1$，$(PC)=4$：程序计数器完成一条指令自动加一，从而现在指向地址为4的指令。
    2. $(PC)\rightarrow MAR$，$(MAR)=4$：存储地址寄存器的值现在置为4，表示它要处理的指令的主存地址为 4。
    3. $M(MAR)\rightarrow MDR$，$(MDR)=000110\,0000000000$：根据存储地址寄存器存储的地址4，在存储体中找到对应的指令$000110\,0000000000$，并把它放入存储数据寄存器中。
    4. $(MDR)\rightarrow IR$，$(IR)=000110\,000000000$：将存储数据寄存器存储的指令放入指令寄存器中。
    5. $OP(IR)\rightarrow CU$：将指令寄存器的000110操作码传输给控制单元中，控制单元根据操作码知道这是“停机”的指令。
    6. 操作系统通过中断指令停止程序。

计算机层次

1. 微程序机器（微指令系统）：由硬件直接执行微指令。
2. 传统机器（用机器语言的机器）：执行二进制机器指令，微程序解释机器执行微指令。
3. 虚拟机器（操作系统机器）：由操作系统程序实现，向上提供“广义指令”即系统调用。也称为混合层。
4. 虚拟机器（汇编语言机器）：不能直接运行汇编语言，必须用汇编程序（汇编器）翻译成机器语言程序，所以是虚拟的。汇编语言与机器语言一一对应。
5. 虚拟机器（高级语言机器）：面向用户，必须用编译程序（编译器或解释器）翻译成汇编语言程序。

- 翻译程序：是指把高级语言源程序转换成机器语言程序（目标代码）的软件。包括编译程序和解释程序。
- 编译程序：将高级语言编写的源程序全部语句一次全部翻译成目标机器语言程序，而后再执行机器语言程序（只需翻译一次）。如C和CPP。
- 解释程序：将源程序的一条语句翻译成对应于机器语言的语句，并立即执行。紧接着再翻译下一句（每次执行都要翻译），所以不会形成目标程序文件。如JS和$Python$。
- 对于$Java$既可以编译也可以解释。

计算机性能指标

主存储器

- MAR位数反映内存（而不是外存）单元的个数（最多支持多少个，如果超过了则MAR也无法寻址到，等同于没用），反映计算机处理能力和并行能力。

- MDR位数=存储字长=每个存储单元的大小，反映计算机一次性处理的数据大小。
- 总容量=存储单元个数×存储字长。单位为bit。
- 如MAR为32位，MDR为8位，总容量=$2^{32} \times 8 \div 8 = 3^{32}Byte$，所以为$4GB$。

中央处理器

- CPU主频：CPU内数字脉冲信号振荡的频率。单位为赫兹。
- CPU时钟周期：CPU主频（时钟主频）=$1 \div CPU$时钟周期。一个时钟多少秒。单位为纳秒或微秒。
- CPI（$Clock\ cycle\ Per\ Instruction$）：执行一条指令所需的时钟周期数。
- 执行一条指令的耗时= $CPI \times CPU$时钟周期。
- CPU执行时间（整个程序的耗时）=CPU时钟周期数÷主频=(指令条数×CPI)÷主频。
- IPS（$Instructions\ Per\ Second$）：每秒执行多少条指令，$IPS$=主频÷平均$CPI$。也可以得到$MIPS$即每秒执行多少百万条指令。由于每个机器有不同指令集，所以使用此进行性能比较有缺陷。
- 指令周期：一条指令需要多少秒，为$1/IPS$。
- $FLOPS$（$Floating-point\ Operations\ Per\ Second$）：每秒执行多少次浮点运算。因此有$MFLOPS$、$GFLOPS$、$TFLOPS$、$EFLOPS$、$ZFLOPS$分别代表每秒$10^6$、$10^9$、$10^{12}$、$10^{15}$、$10^{18}$、$10^{21}$次浮点运算

系统整体

- 数据通路带宽：数据总线一次所能并行传送信息的位数（各硬件部件通过数据总线传输数据）。
- 吞吐量：指系统在单位时间内处理请求的数量。它取决于信息能多快地输入内存，CPU能多快地取指令，数据能多快地从内存取出或存入，以及所得结果能多快地从内存送给一台外部设备。这些步骤中的每一步都关系到主存，因此，系统吞吐量主要取决于主存的存取周期。
- 响应时间：指从用户向计算机发送一个请求，到系统对该请求做出响应并获得它所需要的结果的等待时间。通常包括CPU时间（运行一个程序所花费的时间）与等待时间（用于磁盘访问、存储器访问、I/O操作、操作系统开销等时间）。
- 基准程序是用来测量机算机处理速度的一种实用程序，以便于被测量的计算机性能可以与运行相向程序的其它计算机性能进行比较。但是也存在缺陷，因为基准程序的性能可能与某一段代码密切相关从而对此进行特别优化。

专业术语

- 系列机。具有基本相同的体系结构，使用相同基本指令系统的多个不同型号的计算机组成的一个产品系列。其特征就是指令系统向后兼容。

- 兼容。指计算机软件或硬件的通用性，即使用或运行在某个型号的计算机系统中的硬件/软件也能应用于另一个型号的计算机系统时，称这两台计算机在硬件或软件上存在兼容性。
- 软件可移植性。指把使用在某个系列计算机中的软件直接或进行很少的修改就能运行在另一个系列计算机中的可能性。
- 固件。将程序固定在ROM中组成的部件称为固件。固件是一种具有软件特性的硬件，固件的性能指标介于硬件与软件之间，吸收了软/硬件各自的优点，其执行速度快于软件，灵活性优于硬件，是软/硬件结合的产物。例如，目前操作系统已实现了部分固化（把软件永恒地存储于只读存储器中）。

数据表示与运算

数制与编码

进位计数制

- 真值：符合人类习惯的数字。
- 机器数：数字实际存到机器里的形式，正负号需要被“数字化”。

十进制

- 符号反映权重。
- 符号所在位置也反映权重。
- $K_nK_{n-1}\cdots K_2K_1K_0K_{-1}K_{-2}\cdots K_{-m}=K_n\times10^n+K_{n-1}\times10^{n-1}\cdots K_2\times10^2+K_1\times10^1+K_0\times10^0+K_{-1}\times10^{-1}+K_{-2}\times10^{-2}\cdots K_{-m}\times10^{-m}$。

R 进制

- 基数：每个数码位所用到的不同符号的个数，r进制的基数r。
- R进制转换为十进制：$K_nK_{n-1}\cdots K_2K_1K_0K_{-1}K_{-2}\cdots K_{-m}=K_n\times r^n+K_{n-1}\times r^{n-1}\cdots K_2\times r^2+K_1\times r^1+K_0\times r^0+K_{-1}\times r^{-1}+K_{-2}\times r^{-2}\cdots K_{-m}\times r^{-m}=\sum_{i=-m}^{n}K_i\times r^i$。
- 二进制：0、1。
- 使用二进制的原因：
  1. 可使用两个稳定状态的物理器件表示。
  2. 0，1正好对应逻辑值假、真。方便实现逻辑运算。
  3. 可很方便地使用逻辑门电路实现算术运算。（物理部件性质决定）
- 八进制：$0\sim7$。可以用下标方式表明，也可以用结束的O或开头的0表示。如$(1643)_8$、01643、$1643O$。
- 十进制：$0\sim9$。可以用下标方式表明，也可以用结束的D表示，如$(1643)_{10}$、$1643D$。
- 十六进制：$0\sim9$、A、B、C、D、E、F。可以用下标方式表明，也可以用结束的H或开头的$0x$表示。如$(1643)_{16}$、$0x1643$、$1643H$。

二进制与八进制或十六进制转换

- 二进制转换八进制：三位一组，每组转换成对应的八进制符号，整数部分不全则最高位用0填充，小数部分不全则最低位用0填充。
- 二进制转换十六进制：四位一组，每组转换成对应的十六进制符号，整数部分不全则最高位用0填充，小数部分不全则最低位用0填充。
- 八进制转换二进制：每位八进制对应的三位二进制。
- 十六进制转换二进制：每位十六进制对应的四位二进制。

十进制	二进制	八进制	十六进制
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

十进制转换为R进制

- 整数部分需要使用除基取余法，小数部分需要使用乘基取整法。
- 对于十进制数，需要把它分为整数部分和小数部分两个部分进行处理。
- 已知R进制转换为十进制的方法：$K = K_nK_{n-1}\cdots K_2K_1K_0K_{-1}K_{-2}\cdots K_{-m} = K_n \times r^n + K_{n-1} \times r^{n-1}\cdots K_2 \times r^2 + K_1 \times r^1 + K_0 \times r^0 + K_{-1} \times r^{-1} + K_{-2} \times r^{-2}\cdots K_{-m} \times r^{-m}$。
- 分为整数部分和小数部分：$K = N + F$。
- 首先把整数拿出来得到$N = K_n \times r^n + K_{n-1} \times r^{n-1}\cdots K_2 \times r^2 + K_1 \times r^1 + K_0 \times r^0$。
- 对这个数除以基数r，得到$X = K_n \times r^{n-1} + K_{n-1} \times r^{n-2}\cdots K_2 \times r^1 + K_1 \times r^0$，这时候就会得到一个余数$K_0$。所以$N = rX + K_0$，这时候就能算出$K_0$这个位数了。

- 同理再将得到的商X同样除以r，就能得到K_1，所以不断递归就会得到整数部分所有的K_i。商为0时结束。
- 得到的K_i是从低位排到高位。
- 对于小数部分$F = K_{-1} \times r^{-1} + K_{-2} \times r^{-2} \cdots K_{-m} \times r^{-m}$。
- 对这个数乘基数r，得到$K_{-1} \times r^{0} + K_{-2} \times r^{-1} \cdots K_{-m} \times r^{-m+1}$，取这个常数$K_{-1}$就是想要的答案。将乘积减去这个整数部分得到后面处理的数据
- 同理再将得到的数据不断乘基数，就能得到小数部分所有的K_i。积为1.0时结束。
- 得到的K_i是从高位排到低位。
- 将整数和小数合在一起就是最后的结果。
- 有时候小数会出现无法彻底转换的情况，需要考虑保留多少位。

基本上都是十进制转为二进制，然后由二进制转为十六进制或八进制，具体转换过程见例题。

同理也可以使用拼凑法将数字相减拼凑成对应的数值。这种方法对于只有整数的数值比较好用。

2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}
4096	2048	1024	512	256	128	64	32	16	8	4	2	1	0.5	0.25	0.125

真值与机器数

使用正负号表示正负数的就是真值。

将数的符号和数值一起来编码的，如原码、反码、补码就是机器数。

BCD 码

即$Binary - Coded\ Decimal$，用二进制编码的十进制。

使用$4bit$来表示0到9这十个数，而$4bit$能表示十六个数，所以会冗余六个组合。

8421 码

8421码是一种有权码，第1、2、3、4位分别对应8、4、2、1，使用常规的二进制来表示十进制，冗余最后六个：

0	1	2	3	4	5	6	7	8	9
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

8421码就是二进制表达十进制，只不过十进制只进行简单的对照二进制并进行一位位的拼接，与真正的二进制和十六进制不同。如十六进制的$46H$ =二进制的

0100 0110 =十进制的$38D$ = 8421码的0011 1000，985用8421码表示就是1001 1000 0101。

使用8421码表示的数字进行算术运算的方式是先按照二进制的方式进行运算，若最后结果不在映射表中，即落在没有定义的1010到1111中，就直接加上6（因为有六位无效，所以加上六位跳过无效的位数）从而向前面一个数字段进一位加一，高位补全0。每个段对应的数值合在一起就是原来的结果。

如$5 + 8 = 0101 + 1000 = 1101 = 13$，不在映射表中，则对1101进行二进制的加$6D = 0110$，即计算$1101 + 0110 = 1\,0011$，补齐得到0001 0011，而按照8421码，最高位的0001不再代表权值16，而代表十位的1，而后面是3，从而组合在一起就代表了13。

余三码

在8421码的基础上全部加上三：

0	1	2	3	4	5	6	7	8	9
0011	0100	0101	0110	0111	1000	1001	1010	1011	1100

余三码因为加上了三，所以每一位的权值映射关系就破坏了，所以这是一种无权码，不能分别对应8、4、2、1的值。

2421 码

与8421码一样都是一种有权码，但是映射的方式不同，第1、2、3、4位分别对应2、4、2、1。

0	1	2	3	4	5	6	7	8	9
0000	0001	0010	0011	0100	1011	1100	1101	1110	1111

字符与字符串

英文字符表示

*ASCII*码，数字英文符号一共128个字符，使用7位就可以表示128个字符，但是通常会高位补0凑足$1B$：

- 其中32到126是可印刷字符，其他都是控制或通信字符。
- 数字：48（0011 0000）到57（0011 1001），后面4位就是数字的8421码。
- 大写字母：65（0100 0001）到90（0101 1010），前面三位是010，后面五位代表1到26。
- 小写字母：97（0110 0001）到122（0111 1010），前面三位是011，后面五位代表1到26。

中文字符表示

- $GB\ 2312-80$：1980年推出汉字加符号共7445个表示的字符。中文有两个字节。
- 汉字编码包括汉字的输入编码、汉字内码、汉字字形码，用于输入、内部处理、输出。
  - 输入编码：如拼音、五笔等供人类输入，输入编码输入后转换为国标码再转成汉字内码存储。
    - 区位码：两个字节表示一个汉字，每字节用七位码，将汉字和符号排成94个区，每个区94位。四位十进制数，前两位是区码，后两位为位码。
    - 国标码：将十进制的区位码转换为十六进制。为了中文字符表示与英文字符表示共存，防止GB编码被认为是$ASCII$码的0到32位的控制或通信字符，所以在94区94位的基础上还要各自加上32，即$20H$，防止信息交换时冲突。国标码两字节最高位都是0。
  - 汉字内码：国标码只能用于信息传输，而如果是存储在计算机上，由于前128位已经被$ASCII$码占用了，所以在国标码的基础上再各自加$80H$（128），从而计算机识别字符时，看到0到128之间的就能辨认出是$ASCII$码，大于128的就是GB码。因为$ASCII$码高位是0，而GB码是两个字节且高位都是1。
    - 国标码=区位码H+$2020H$。
    - 汉字内码=国标码H+$8080H$。
  - 汉字字形码：把汉字输出成汉字的样子。

字符串

- 若一个计算机按字节编址，则每个地址对应$Q1B$。
- 很多语言中将$'\backslash 0'$即$00H$作为字符串结束标志。
- 大端模式：将数据的最高有效字节存放在低地址单元中。
- 小端模式：将数据的最高有效字节存放在高地址单元中。
- 计算机默认是小端模式，人类常用书写方式是大端模式。

数据校验

现在计算机组成原理不考数据校验，但是只是内容移动到计算机网络里。具体可以看计算机网络。

- 校验原理就是将更多字节映射到有限个合法状态，从而有多个冗余的非法状态，更容易判断是否非法。
- 码字：由若干位代码组成的一个字。
- 两个码字间的距离：将两个码字逐位进行对比，具有不同的位的个数。

- 码距：一种编码方案可能有若干个合法码字，即为各任意合法码字间最少变化的二进制最小位数。（如1000与1001码距为1，1111码距为3）

纠错理论

当码距= 1时，无检错能力；当码距= 2时，有检错能力；当码距≥ 3时，若设计合理，可能具有检错、纠错能力。

$L-1=D+C$且$D \geq C$。即编码最小码距L越大，其检测错误的位数D越大，纠正错误的位数C也越大，且纠错能力恒小于等于检错能力。

奇偶校验码

即通过在数据上添加一位校验位让所有数据为1的位的个数为奇数或偶数。

- 奇校验：让整个校验码（有效信息位加上校验位）中为1的个数为奇数。
- 偶校验：让整个校验码（有效信息位加上校验位）中为1的个数为偶数。
- 偶数个位数出错时无法校验。
- 对原始数据进行异或（模二加）运算，得到的结果即为校验位。
- 对所有数据进行异或运算，结果为0表示未出错，为1代表出错。
- 码距为2，只能检错，不能纠错。（为1的位数的个数为偶数或奇数，偶数和偶数，奇数和奇数之间最小差距为2，所以码距为2）

海明码

- 将信息分组进行偶校验，从而得到多个校验位，从而能携带多种状态信息。
- 设信息位为n，校验位为k，从而校验位能表达2^k种状态，而传输总数据位数=信息位+校验位一共$n+k$位，只错一位的状态种数加上一种正确状态为$n+k+1$，从而$2^k \geq n+k+1$。
- 令信息位为D_i，校验位为P_j，总海明码为H_k，其中校验位P_j必须放在海明码H_k位号位2^{j-1}的位置上，即1、2、4、8等。
- 如果没有发生错误，则每一位进行检错都是0，若出现1，则说明出错。
- 为了检测是一位错还是两位错，一般会加上一个全校验位，对整体进行偶校验。
- 具有一位的纠错和两位的检错能力，三位以上则不能检错。

n	1	2-4	5-11	12-26	27-57	58-120
k	2	3	4	5	6	7

循环冗余校验码

- 即CRC码，其思想是：
    1. 数据发送、接受方约定一个“除数”生成多项式$G(x)$。R为$G(x)$位数减一。

2. K个信息位+R个校验位作为“被除数”，添加校验位后需保证除法的余数为0。
3. 收到数据后，进行除法检查余数是否为0。

- 得到CRC码的方法：
  1. 确定K和R以及生成多项式对应的二进制码。其中R位生成多项式的最高次幂数。
  2. 将信息码左移R位，低位补0。
  3. 使用模二除法。（即相减时是异或运算，不能向上进位或借位；除法取余数时也不比较除数与被除数除第一位以外的低位大小，只需要相同的位数，这是由于异或加减法不进行进借位的特征）
  4. 余数就是校验位，只比多项式少一位。
  5. 对全部数据进行多项式除，余数为0代表无错。
  6. 若余数不为0，则出错。
  7. 若n个信息位，k个校验位，若生成多项式得当，且$2^k \geq n + k + 1$，则CRC码可纠正一位错。实际上基本上不怎么用来纠错。
- 检错：将收到的CRC码用生成多项式$G(x)$做模二除法，若除数为0则码字无误，若只有一位为1，则该位出错，直接取反，更多位则无法检错。即只具有一位的检错纠错能力。

定点数

指小数点的位置不变，使用常规计数法，如96.94。

定点数表示

无符号数

- 整个机器字长的全部二进制位均为数值位，没有符号位，相当于数的绝对值。
- n位无符号数表示范围为$[0, 2^n - 1]$。
- 无符号数不涉及小数。
- 无符号数只有原码的概念，没有反码、补码、移码。

有符号数

- 假设机器字长$n + 1$位，符号位1位，有效数字位n位。
- 定点整数：最高一位是符号位，0是正，1是负，小数点位置一般隐含在最后。范围为$[-(2^n - 1), 2^n - 1]$。
- 定点小数：最高一位是符号位，0是正，1是负，小数点位置隐含在符号位后面，是纯小数。范围为$[-(1 - 2^{-n}), -2^{-n}] \cup [2^{-n}, 1 - 2^{-n}]$。
- 数值部分也称为尾数。要保存一个非整数需要保存定点整数与定点小数两个部分。

原码

- 原码：用尾数表示真值的绝对值，符号位“0/1”对应“正/负”。
- 若机器字长为$n+1$位，则尾数占n位。
- 若使用$1B$来保存数值，则$+19D$就是0001 0011，$-19D$就是1001 0011。
- 有时1001 0011会写为1,0010011，其中的逗号只是为了标注正负号，本身是不存在的。
- 若未指明机器字长，则最开头的多个0可以省略。如1001 0011可以表示为1,10011。
- 同理小数也可以使用1.11表示，这是指-0.11而不是真值1.11。
- 若机器字长$n+1$位，则原码整数的表示范围是$[-(2^n-1), 2^n-1]$。
- 若机器字长$n+1$位，则原码小数的表示范围是$[-(1-2^{-n}), -2^{-n}] \cup [2^{-n}, 1-2^{-n}]$。
- 原码表示时真值0有$+0$和-0两种形式，因为符号位对0没有影响，所以造成了一定的空间浪费和错乱。

反码

- 反码：若符号位为0，则反码与原码相同，若符号位为1，则数值位全部取反。
- 可以转换为原码再取反。
- 若机器字长$n+1$位，则反码整数的表示范围是$[-(2^n-1), 2^n-1]$。
- 若机器字长$n+1$位，则反码小数的表示范围是$[-(1-2^{-n}), -2^{-n}] \cup [2^{-n}, 1-2^{-n}]$。
- 范围表示没有变化。
- 反码表示时真值0有$+0$和-0两种形式。
- 反码只是由原码转换为补码的一个中间态，实际上并没有作用。

补码

- 补码：若符号位为0，则反码与原码相同，若符号位为1，则数值位全部取反再加一，即反码加一。
- 补码表示时真值0只有一种形式0000 0000。
- 多出来的一种形式1000 0000表示正数的-2^7和小数的-1。
- 若机器字长$n+1$位，则补码整数的表示范围是$[-2^n, 2^n-1]$。（比原码多个-128）
- 若机器字长$n+1$位，则补码小数的表示范围是$[-1, 1-2^{-n}]$。（比原码多个-1）
- 负数补码转回原码：尾数取反，末位加一；或是负数补码中，最右边的1以及右边不变，最右边的1的左边取反。
- 数值的补码求其负数的补码：全部位包括符号位取反，末位加一。
- 对一个整数的补码再求补码，等于该整数自身。

- 补码算术移位：将补码的符号位与数值位一起右移一位并保持原符号位的值不变，表示除二。
- 变形补码：又称为模四补码，即双符号位的补码小数，用00表示正，11表示负，用于完成算术运算的ALU部件中。

移码

- 一般是在补码的基础上只将符号位取反，即在真值上加上一个常数偏移量2^n。也可能加上不同的偏移量。
- 移码只能用于表示整数，而不能表示定点小数。一般用来表示浮点数的阶码。
- 若机器字长$n+1$位，则移码整数的表示范围是$[-2^n, 2^n-1]$。
- 若机器字长$n+1$位，则移码小数的表示范围是$[-1, 1-2^{-n}]$。
- 只有一个零的表示$10\cdots0$。
- 移码由于负数的最高位为0，正数的最高位为1，所以保证了原数据大小顺序，从而能更方便对比大小。

机器数	无符号数	原码	反码	补码	移码
0000 0000	0	+0	+0	0	-128
0000 0001	1	+1	+1	+1	-127
…					
0111 1101	125	+125	+125	+125	-3
0111 1110	126	+126	+126	+126	-2
0111 1111	127	+127	+127	+127	-1
1000 0000	128	-0	-127	-128	0
1000 0001	129	-1	-126	-127	1
1000 0010	130	-2	-125	-126	2
…					
1111 1101	253	-125	-2	-3	125
1111 1110	254	-126	-1	-2	126
1111 1111	255	-127	-0	-1	127

补码作用

- 原码在计算时由于首位表示的是符号，所以需要考虑将加减运算转换的问题，而减法实现起来比较困难，就考虑是否可以将减法通过加法来实现。
- 由于计算机码操作若最高位进一就被舍弃，则天然是进行模运算，所以可以通过数学模运算来实现机器码的运算。
- 带余除法：设$x, m \in Z$，$m > 0$则存在唯一决定的整数q和r，使得$x = q \cdot m + r, 0 \le r < m$。
- 若两个数绝对值之和为模，则互为补数。即模-数的绝对值=数的补数（正数）。从而数加上数的补数就得到了模。

- 补码就是正数不变，负数取模的结果。如$-66 = -0100\,0010$，而$(1000\,0000 - 0100\,0010) \bmod (1111\,1111) = 1011\,1110$，也就是其补码。
- 从而就可以用补数的加法（被减数转为其负数的补码）替代原码转换的加减法。
- 所以补码可以让减法操作转换为加法操作，减少硬件成本。

定点数运算

定点数移位运算

- 算术移位：通过改变各个数码位和小数点的相对位置，从而改变各数码位的位权。可用移位运算实现乘法、除法。
  - 原码的算数移位——符号位保持不变，仅对数值位进行移位：
    - 若右移高位补0，低位舍弃，右移代表除2，若移出的是0则刚好整除，若移出的是1则会整除余1丢失精度。
    - 若左移低位补0，高位舍弃，左移代表乘2，若移出的是0则刚好乘2，若移出的是1，则会溢出严重误差。
  - 反码的算术移位——正数的反码与原码相同，所以正数的处理跟原码一样。而对于负数而言，反码的1等于原码的0，反码的0等于原码的1：
    - 若右移低位补1，高位舍弃。
    - 若左移高位补1，低位舍弃。
  - 补码的算术移位——正数的补码与原码相同，所以正数的处理跟原码一样。由于负数补码=反码末位加一，导致反码最右边几个连续的1都因进位而变为0，直到进位碰到第一个0为止。所以得到规律：
    - 负数补码中，最右边的1及其右边同原码一样。最右边的1的左边同反码一样。
    - 右移同反码，高位补1，低位舍弃。
    - 左移同原码，低位补0，高位舍弃。
- 逻辑移位，逻辑的移位可以视为对无符号数的算术移位：
  - 逻辑右移：高位补0，低位舍弃。
  - 逻辑左移：低位补0，高位舍弃。
- 循环移位，将移出的一位放到另一端的端点，类似队列：
  - 进位位CF：保存计算是否进位。1代表产生进位，0代表未产生进位。
  - 带CF就是大循环，不带CF的就是小循环。
  - 循环右移：将最低位的一位移出放到最高位，其余右移一位。
  - 循环左移：将最高位的一位移出放到最低位，其余左移一位。
  - 带进位位的循环左移：需要加上进位位数值的循环左移。
  - 带进位位的循环右移：需要加上进位位数值的循环右移。
  - 适合将数据的低字节数据和高字节数据互换。

定点数加减运算

- 注意机器字长，若溢出则将溢出位丢弃。
- 原码的加减，由加法器和减法器两个硬件来实现，因为最高位为符号位，所以不能直接进行加减：
  - 原码的加法：
    - 正数+正数：绝对值做加法，结果为正数。
    - 负数+负数：绝对值做加法，结果为负数。
    - 正数+负数：绝对值大的减绝对值小的，符号同绝对值大的数。
  - 原码的减法，减数符号取反，转变为加法：
    - 正-负→正+正。
    - 负-正→负+负。
    - 正-正→正+负。
    - 负-负→负+正。
- 补码的加减，由于减法器的硬件实现比较困难，所以原码的减法操作可以由补码来更简单实现，不用考虑符号位的问题，直接全部参与运算：
  - 参与运算的两个操作数均用补码表示。
  - 按二进制运算规则运算，逢二进一（与模二加减法不同需要进位）。
  - 符号位与数值位按同样规则一起参与运算，符号位运算产生的进位要丢掉，结果的符号位由运算得出。
  - 补码加减运算依据下面的公式进行。当参加运算的数是定点小数时，模$M=2$；当参加运算的数是定点整数时，模$M=2^{n+1}$。$[A+B]_{补}=[A]_{补}+[B]_{补}$（$\bmod M$）；$[A-B]_{补}=[A]_{补}+[-B]_{补}$（$\bmod M$）。
  - $\bmod M$运算是为了将溢出位丢掉。也就是说，若做加法，则两数的补码直接相加；若做减法，则将被减数与减数的机器负数相加。
  - 补码运算的结果亦为补码。
- 溢出判断，由于使用补码进行加减操作都会变成加法，所以只用考虑加法溢出的处理。
  - 小于最小值就是下溢。只有负数+负数才会下溢得到正数。如$-24-124=1110\,1000+1000\,0100=0110\,1100=108$。
  - 大于最大值就是上溢。只有正数+正数才会上溢得到负数。如$15+124=0000\,1111+0111\,1100=1000\,1011=-117$。
  - 对于小数，其绝对值小于等于1。
  - 方法一，采用一位符号位（模二补码），根据符号位判断：
    - 设A的符号为A_s，B的符号为B_s，运算结果的符号为S_s。
    - 则溢出逻辑表达式为$V=A_sB_s\overline{S_s}+\overline{A_sB_s}S_s$（即$V=A_s\&\&B_s\&\&!(S_s)\,||\,!(A_s)\&\&!(B_s)\&\&S_s$，前面判断下+溢，后面判断上溢）。
    - 若$V=0$，表示无溢出，若$V=1$，表示有溢出。

        - 由于只有操作数同号才能溢出，即判断标准为，两个操作数是否同号，结果符号是否与原操作数相同，这两个条件必须同时满足。
        - 如$-24-124=108$产生了溢出，$A_s=1$、$B_s=1$、$S_s=0$，$V=A_sB_s\overline{S_s}+\overline{A_sB_s}S_s=1\,1\,1+0\,0\,0=1+0=1$，所以产生了溢出。
    - 方法二，采用一位符号位（模二补码），根据数据位进位1情况判断：
        - 符号位的进位$C_s=0$，最高数值位的进位$C_1=1$时产生了上溢。
        - 符号位的进位$C_s=1$，最高数值位的进位$C_1=0$时产生了下溢。
        - 原理同方法一类似，如果进位产生并符号与期待符号不同则发生溢出。
        - 如$-24-124=1110\,1000+1000\,0100=0110\,1100$中符号位都为1相加结果为0，所以符号位进位，$C_s=1$，而最高数值位为$1+0=1$，没有进位，所以$C_1=0$，所以就产生了下溢。
    - 方法三，采用双符号位（模四补码），正数符号为00，负数符号为11。
        - 若两个符号位不同，则表示溢出，第一个符号位表示应该得到的符号位，第二个符号位代表实际得到的符号位。
        - 如$-24-124=11,110\,1000+11,000\,0100=10,110\,1100=108$。下溢。
        - 如$15+124=00,000\,1111+00,111\,1100=01,000\,1011=-117$。上溢。
        - 实际存储时只存储一个符号位，运算时会复制一个符号位。
- 符号扩展：防止溢出的一个方法就是将短数据扩展为长数据，把数据全部拓展为等长。（正数补码全部补0，负数补码高位补1低位补0）
    - 整数扩展，在原符号位和数值位中间添加新位，正数都填充0，对于负数：
        - 原码：扩展补0。
        - 反码：扩展补1。
        - 补码：扩展补1。
    - 小数扩展，在最后面添加新位，正数都填充0，对于负数：
        - 原码：扩展补0。
        - 反码：扩展补1。
        - 补码：扩展补0。

定点数乘法运算

二进制乘法运算可以参考十进制的乘法运算，将乘数一位一位的乘被乘数然后再全部错位相加得到的就是答案。

```
  318
× 211
-------
  318
 318
636
-------
67098
```

为什么要错位？因为乘数每一位的数值的位权不同，为基数的幂，所以本质应该是：

```
       318
×(200+10+1)
-----------
       318
      3180
     63600
-----------
     67098
```

而使用二进制的一位位乘法显然比十进制的一位位乘更简单：

```
    1101
×   1011
--------
    1101
   1101
  0000
 1101
--------
10001111
```

所以此时分析笔算二进制乘法就被拆解出了二进制乘法流程：将乘法变为加法和移位运算，将乘数一位位的向前移动并与被乘数相乘，由于是二进制所以只有0和1，遇到乘数当前位值为1就在原来的和上加上被乘数，并向前移动一位，如果是0就加上0继续移动一位，当乘数位数访问完成则乘法完成，所有的和相加成为最后的积。

这就是原码乘法的原理，可以考虑用机器实现，但是还是要考虑以下问题：如何处理符号？乘积位数扩大一倍是否可能超过机器字长？各个位积如何相加变成最后的乘积？

原码一位乘法：

- 一般使用原码一位乘法，即每次只乘一位的数据。
- 在原码乘法时，可以先符号位单独处理，将两个符号进行异或操作，得到的结果就是最后的结果的符号。然后对数据的绝对值（去除符号位）进行一位位的乘法（位积）然后相加。

- 由于运算时可能存在绝对值大于1但是不是溢出的情况，所以部分积和被乘数使用双符号位。
- 在运算器的组成时出现一个表格，说明在进行乘运算时，ACC保存乘积高位，MQ保存乘数与乘积低位，X保存被乘数。
- 原码一位乘法机器实现时就是按照这种方式计算：
    1. 字长若为$n+1$位，则ACC、MQ、X全部初始化为n位，将被乘数的绝对值放入X中，MQ放入乘数的绝对值，ACC初始化为全0。
    2. 将MQ的最右边的一位当做当前乘运算位，让其进行乘运算，运算规则是，若当前位是1，则ACC加上被乘数，即$ACC+=X$，若当前位是0，则ACC加上0（保持不变，跳过）。
    3. 将ACC和MQ的数据连接在一起，全部逻辑右移一位，ACC数据高位补0，ACC最后一个低位移到MQ的最高位。将MQ的最后一位抛弃。若是第 i 轮逻辑右移，则MQ的前i位是结果的后i个低位值。
    4. 从步骤二开始重复，字长若为$n+1$位，则重复n次，直到MQ的最后一位是符号位，则停止计算。此时ACC的全部和MQ的前n位都是结果。
    5. 定点小数的小数位隐藏在符号位后面第一位，定点正数的小数位隐藏在MQ符号位的前一位。
    6. 将两个符号位的异或结果赋值给积最高位。
- 原码一位乘法逻辑运算：
    1. 初始化，左边为部分积，即计算的部分结果，最开始为全0，右边为乘数的绝对值，最后边全部为丢失位。
    2. 根据丢失位前一位的值来判断加上什么，若是1则加上被乘数的绝对值，若是0则加上被乘数等长的全0。
    3. 右移部分积一位，高位补0，丢失位多一位。
    4. 继续计算，直到乘数全部被移出。字长为$n+1$位则需要移位计算n次。丢失位前的就是全部部分积。
    5. 将两个符号位的异或结果赋值给积最高位。

补码一位乘法：

- 对于补码的乘法运算的逻辑也跟原码的类似，补码的计算就是使用$Booth$算法实现。
- 辅助位其实就是在MQ最后再加上一位，辅助位初始为0。每次右移会使MQ的最低位顶替原本的辅助位（事实上MQ共$n+2$位）。
- 为了保证统一，所以ACC和X都会增加一位，变成$n+2$位，多出来的一位就可以实现双符号位补码运算，而MQ还是用原来的单符号位。
- 为了加快运算会有辅助电路实现$(-x)$的补码的运算。
- 最后一次不需要移位直接根据辅助位和MQ最后一位判断进行相加。从而让乘数的符号位也参数运算中来确定最后结果的符号。

- 补码一位乘法逻辑运算：
    1. 初始化，左边为部分积，即计算的部分结果，最开始为全0，右边为乘数，然后是一个辅助位，最后边全部为丢失位。
    2. 根据辅助位$-MQ$最低位的差值来判断加上什么，若是1则加上被乘数的补码，若是0则加与被乘数等长的全0，若是-1则加上被乘数的负数的补码。
    3. 算术右移部分积一位，正数高位补0，负数高位补1，丢失位多一位。
    4. 继续计算，直到乘数全部被移出。字长为$n+1$位则需要移位计算n次。丢失位前的就是全部部分积。
    5. 最后一次不需要移位，再加一次。

$Booth$算法的移位法则，其中y_n为MQ最低位，y_{n+1}为辅助位：

y_n（高位）	y_{n+1}（低位）	操作
0	0	部分积右移一位
0	1	部分积加$[X]_{补}$，右移一位
1	0	部分积加$[-X]_{补}$，右移一位
1	1	部分积右移一位

即辅助位减MQ最低位的值，若是1就加补码，若0则加0，若-1则加负数的补码。

两种乘法的区别：

	原码	补码
计算流程	n 轮加法、移位	n 轮加法、移位，最后进行一次加法
加法的值	+0、+x 的原码	+0、+x 的补码、+(-x)的补码
判断加值的根据	MQ 的最低位	辅助位-MQ 的最低位
判断关系	MQ 的最低位=1 时，ACC+x 的原码；MQ 的最低位=0 时，ACC 不变	辅助位-MQ 中最低位=1 时，(ACC)+x 的补码；辅助位-MQ 中最低位=0 时，ACC 不变；辅助位-MQ 中最低位=-1 时，ACC+(-x)的补码
移位类型	逻辑右移	算术右移
符号位	不参与运算	参与运算

定点数除法运算

二进制除法运算可以参考十进制的除法运算，将除数乘上一位数，使得乘积最大且不超过被除数，然后将被除数减去这个乘积并左移就得到下一轮运算的被除数。

```
      1.507
    --------
211 | 318
      211
    --------
      1070
      1055
    --------
        150
        000
    --------
        1500
        1477
    --------
          23
         ...
```

所以除法就是为了凑和被除数相近的商。

而使用二进制的一位位除法同理：

```
          0.1101
      ------------
01101 | 01011
        00000
      ------------
         10110
         01101
      ------------
          10010
          01101
      ------------
           01010
           01101
      ------------
            01010
            00000
      ------------
             10100
             01101
      ------------
              0111
```

- 进行除法操作时，都是为了找到一位能让商乘除数能最大即余数最小但大于0的值。若除数被除数都是小数，可以同时乘一个数变成整数再运算。

- 所以可以忽略小数点，每确定一位商进行一次减法，若机器字长为n位，则得到$n-1$位余数，在余数末尾补0，再确定下一位商0或1，直到确定n位商即可停止。
- 在运算器的组成时出现一个表格，说明在进行除运算时，ACC保存被除数和余数，MQ保存商，X保存除数。

所以同理可以推出恢复余数法，与源码一位法类似，都是计算绝对值最后判断符号。余数加0扩大余数就相当于左移一位。

原码恢复余数法：

物理上：

1. 字长若为$n+1$位，则ACC、MQ、X全部初始化为n位，将被除数的绝对值放入ACC中，X放入除数的绝对值，MQ初始化为全0。
2. 将MQ的最右边的一位当做当前除运算位，让其进行除运算，运算规则是，默认商1，$ACC-=X$，即X的补码要加上除数的绝对值的负值的补码（减法都由补码的加法实现），判断是否有误。若结果高位为0则无误，高位为1则有误，错误则商改为0，并恢复余数，ACC加上X中除数的补码。
3. 将ACC和MQ的数据连接在一起，全部逻辑左移一位，MQ数据低位补0，MQ最高位的0移到 ACC 的最低位。将ACC的最高一位抛弃。若是第i轮逻辑左移，则MQ的后i位是当前计算的商的结果。
4. 从步骤二开始重复，字长若为$n+1$位，则左移n次，上商$n+1$次，直到MQ中全部是计算结果，则停止计算。此时MQ中保存商，ACC中保存左移n位的余数值，真正的余数应该是结果再乘上2的$-n$次方。
5. 定点小数的小数位隐藏在符号位后面第一位，定点正数的小数位隐藏在最后一位后。
6. 将两个符号位的异或结果赋值给商最高位。

逻辑上：

1. 老余数减去除数的绝对值，得到新的余数。
2. 若新余数为负，即高位为1表示是负数，则商0，并重新加上除数的绝对值恢复为老余数。
3. 若新余数为正，则商1。
4. 余数逻辑左移，继续运算。

默认会商1，如果发现结果有问题就恢复余数（变成商0）。若字长为$n+1$，则只用左移n次，上商$n+1$次，最后一次上商余数不左移。

原码加减交替法：

- 因为恢复余数很麻烦，所以会考虑是否不用恢复余数，直接进行运算得到后面的结果。

- 假如令原始值为x，原始值加上$-y$绝对值的补码结果为a，y绝对值的补码为b，按恢复余数法，a这个余数是一个负值，所以要加上b即$a+b$变成原始值$a+b=x$，这时候商0，然后计算下一个商，余数$a+b$左移一位，即$(a+b)\times 2=2a+2b$，这时候商1看看结果是否正确，即$2a+2b$要减去 y 绝对值的补码（等价于加上$-y$绝对值的补码）$2a+2b-b=2a+b$。
- 所以如果得到了一个负的余数a，可以直接转换到$2a+b$这个结果，即直接左移一位余数再加上除数的补码就可以得到结果。
- 这就是加减交通法或不恢复余数法。
- 从而恢复余数法就是当余数为负时商0，并+|除数|，再左移，再-|除数|，而加减交替法是当余数为负时商0，并左移一位，再+[除数|，若余数为正时商1，并左移一位，再-|除数|。
- 值得注意的是，若在最后一步余数为负，需要商0，并加上除数的补码得到正确余数。

逻辑上：

1. 被除数减去除数的绝对值，得到新的余数。
2. 余数为负，则商0，左移，加上除数绝对值。
3. 余数为正，则商1，左移，减去除数绝对值。
4. 在最后一步余数为负，需要商0，并加上除数的补码得到正确余数。

注意：在定点小数运算时，商只能是小数而不能是整数，所以被除数一定要小于除数，机器判断标准是看第一步计算的商，若第一步计算的商是1则代表结果大于1，机器就会报错。

补码加减交替法：

- 符号位参与运算。
- 被除数÷余数、除数采用双符号位。
- 被除数和除数同号，则被除数减去除数，异号则被除数加上除数。
- 余数和除数同号，商1，余数左移一位减去除数;
- 余数和除数异号，商0，余数左移一位加上除数。
- 重复n次。
- 最后一次计算时末位商恒置为1，处理简单，而且精度误差也不会超过2^{-n}。

逻辑上：

1. 被除数和除数同号，则被除数减去除数，若异号则被除数加上除数。
2. 余数和除数异号，则商0，左移，加上除数。
3. 余数和除数同号，则商1，左移，减去除数。
4. 重复n次，末位商恒置为1。

除法类型	符号位参与运算	加减次数	移位方向	移位次数次数	上商和加减原则	说明
原码加减交替法	否	N+1 或 N+2	左	N	余数的正负	若最终余数为负，需恢复余数
补码加减交替法	是	N+1	左	N	余数和除数是否同号	商末位恒置 1

其实本质上原码和补码的上商和加减原则是一样的，只是因为除数被去掉符号为正值，所以余数是负就异号，是正就同号。

定点数强制类型转换

- 无符号数与有符号数：不改变数据内容，只改变解释方式。
- 长整数转短整数：高位截断，保留低位。
- 短整数转长整数：符号扩展。

1. 有符号数和无符号数之间的转换。例如，由$signed$型转换为等长$unsigned$型数据时，符号位成为数据的一部分，即负数转换为无符号数时，数值将发生变化。同理，由$unsigned$转换为$signed$时最高位作为符号位，也可能发生数值变化。
2. 数据的截取与保留。当一个浮点数转换为整数时，浮点数的小数部分全部舍去，并按整数形式存储。但浮点数的整数部分不能超过整型数允许的最大范围，否则溢出。
3. 数据转换中的精度丢失。四舍五入会丢失一些精度，截去小数也会丢失一些精度。此外，数据由$long$型转换为$float$型或$double$型时，有可能在存储时不能准确地表示该长整数的有效数字，精度也会受到影响。
4. 数据转换结果的不确定性。当较长的整数转换为较短的整数时，要将高位截去。例如，$long$型转换为$short$型，只将低16位送过去，这样就会产生很大的误差。浮点数降格时，如$double$型转换为$float$型，当数值超过$float$型的表示范围时，所得到的结果将是不确定的。

定点数数据存储与排列

- 数据最左边的高位就是最高有效字节MSB。
- 数据最右边的低位就是最低有效字节LSB。
- 大端模式：将MSB存到最低地址，LSB存在最高地址。便于人类阅读。
- 小端模式：将MSB存到最高地址，LSB存在最低地址。便于机器读取。
- 边界对齐：
  - 现代计算机通常是按字节编址即每个字节对应一个地址。也支持按字、按半字、按字节寻址。
  - 假设存储字长为32位，则1个字=$32bit$，半字= $16bit$。每次访存只能读/写1个字。

- 字地址转换为字节地址，只用逻辑左移两位就可以了，即乘以4，因为字长为32，而字节长8。
- 使用边界对齐方式会让每个数据都能一次性读完而不用跨行读取，多余的空间用0填充。

浮点数

指小数点的位置不固定，如使用科学计数法，如$9.694E2$。

浮点数表示

阶码与尾数

- 定点数能表示的数字范围有限，但我们不能无限制地增加数据的长度。
- 类似科学计数法，分为阶符、阶码、数符和尾数四个部分，阶符和阶码反映数值大小、表示范围、小数点实际位置，数符代表浮点数的符号，尾数反映精度。
- 对于二进制的浮点数，阶码是常用补码或移码表示的定点整数，而尾数是常用原码或补码表示的定点小数。
- 若阶码的真值为E，尾数的真值为M，则浮点数的真值为$N = r^E \times M$，其中r为阶码的底，即基数，一般为2。

尾数规格化

为了提高精度，充分利用尾数有效位数必须进行规格化，规定尾数必须是一个有效值。

- 左规：出现下溢需要左规，即若尾数的高位是无效值（即为0）则会丧失精度，所以我们需要尽可能将尾数多保存一些1，从而让最高位为1。所以需要让数值左移，让小数点右移，尾数算术左移n位，阶码减n，直到尾数最高位是有效值。可能会进行多次。
- 右规：出现上溢需要右规，规范要求小数点要在第一个非0的数据右边，如果小数点前有超过1个有效位，则需要将数值右移，小数点左移，尾数算术右移n位，阶码加n，直到小数点在尾数最高位的右边。只需要一次。

规格化浮点数的特点：

1. 用原码表示的尾数进行规格化，最高位数值一定为1：
   - 正数为$0.1xx\cdots x$的形式，其最大值表示为$0.11\cdots1$；最小值表示为$0.10\cdots0$。
   - 尾数的表示范围为$\frac{1}{2} \le M \le (1-2^{-n})$。
   - 负数为$1.1xx\cdots x$的形式，其最大值表示为$1.10\cdots0$;最小值表示为$1.11\cdots1$。
   - 尾数的表示范围为$-(1-2^{-n}) \le M \le -\frac{1}{2}$。

2. 用补码表示的尾数进行规格化，符号位与最高位数值一定相反：
   - 正数为$0.1xx\cdots x$的形式，其最大值表示为$0.11\cdots 1$；最小值表示为$0.10\cdots 0$。跟原码一致。
   - 尾数的表示范围为$\frac{1}{2} \le M \le (1-2^{-n})$。
   - 负数为$1.0xx\cdots x$的形式，其最大值表示为$1.01\cdots 1$；最小值表示为$1.00\cdots 0$。（负数的补码$1.0xx\cdots x$取反后就是$1.1xx\cdots x$）
   - 尾数的表示范围为$-1 \le M \le -\left(\frac{1}{2}+2^{-n}\right)$。（补码中强制规定$1.00\cdots 0$就代表$-1$）

当浮点数尾数的基数为2时，原码规格化数的尾数最高位一定是1，补码规格化数的尾数最高位一定与尾数符号位相反。

基数不同，浮点数的规格化形式也不同。当基数为4时，原码规格化形式的尾数正数最高两位不全为0，负数最高两位不全为1；当基数为8时，原码规格化形式的尾数正数最高3位不全为0，负数正数最高3位不全为1。

如若基数为8，则0.000111和1.111010都不是规格化数，而1.101010是规格化。

对于浮点数，上溢和下溢有正负之分：负上溢<负数区<负下溢<0<正下溢<正数区<正上溢。

定点浮点区别

假设定点数和浮点数的字长相同。

- 浮点数表示范围更大。
- 浮点数精度降低。（由于只取出一部分数据，还有部分长度表示阶码等，所以用于表示尾数的字长减少）
- 浮点数运算更复杂。（需要规格化，需要做尾数运算和阶码运算）
- 浮点数只有规格化后才能判断是否溢出。

IEEE 754 标准

IEEE 754标准就是浮点数标准，为了解决计算机中阶码、尾数使用什么码来表示，各取多少位的问题。

移码定义

- 移码的定义其实=真值+偏置值，一般取2^{n-1}，这时候移码才等于补码符号位取反，若移码采取其他方案则没有这个特点。
- 在*IEEE* 754标准中，规定移码的偏置值不再是128而是127，即$2^{n-1}-1$。所以这个标准下的移码与一般的移码不同。
- 从而真值-128的移码为$-1000\ 0000 + 0111\ 1111 = 1111\ 1111$，$-127$的移码为$0000\ 0000$，0的移码为$0111\ 1111$，127的移码为$1111\ 1110$。

IEEE 754 定义

- 分为数符（表示数值正负号）、阶码（用移码表示）、尾数（用原码表示，且默认最高位为1，实际尾数位都要加1）。
- 标准中会将全0的-127（非规格化数）和全1的-128（无穷大）做特殊的用途，所以短浮点数的真值正常范围是-126到127。

英文类型	中文类型	数符位	阶码位	尾数位	总位数	十六进制偏置值	十进制偏置值
float	短浮点数	1	8	23	32	7FH	127
double	长浮点数	1	11	52	64	3FFH	1023
long double	临时浮点数	1	15	64	80	3FFFH	16383

- 令数符为S，阶码为E，尾数为M。
- 规格化的短浮点数的真值为$(-1)^S \times 1.M \times 2^{E-127}$。
- 规格化的长浮点数的真值为$(-1)^S \times 1.M \times 2^{E-1023}$。
- 短浮点数和长浮点数都隐含一位尾数最高位1，即0.11为+1.11，1.11为-1.11，而临时浮点数没有隐含位。

IEEE 754 取值范围

格式	规格化的最小绝对值	规格化的最大绝对值
单精度	$E=1$，$M=0$：$1.0 \times 2^{1-127}=2^{-126}$	$E=254$，$M=.11\ldots1$：$1.11\ldots1 \times 2^{254-127}=2^{127}\times(2-2^{23})$
双精度	$E=1$，$M=0$：$1.0 \times 2^{1-1023}=2^{-1022}$	$E=2046$，$M=.11\ldots1$：$1.11\ldots1 \times 2^{2046-1023}=2^{1023}\times(2-2^{-52})$

- 对于规格化短浮点数，只有$1 \le E \le 254$时，$(-1)^S \times 1.M \times 2^{E-127}$。
- 当阶码E为全0，即应该为$-127D$，而尾数M不全为0时，表示非规格化小数$\pm(0.xx\ldots x)_2 \times 2^{-126}$。（此时最高位就不默认为1了，阶码固定设置为$-126$）。
- 当阶码E为全0，即应该为$-127D$，而尾数M全为0时，表示真值±0，正负看数符。
- 当阶码E为全1，即应该为$-128D$，而尾数M全为0时，表示无穷大$\pm\infty$，正负看数符。
- 当阶码E为全1，即应该为$-128D$，而尾数M不全为0时，表示非数值NaN（$Not\ a\ Number$）。如果非法操作如$0 \div 0$等就会使用到。

尾数运算

由于阶码由原码转换为移码，尾数加减都需要进行原码运算，所以需要两种实现方式。

- 转换为补码进行加减，再转回原码。
- 直接用原码加减，符号和数值分开。

浮点数运算

科学计数法加减运算

1. 对阶：阶数小的向阶数更大的对齐。
   - 因为计算机内部，尾数是定点小数，小数点位置不会变，改变的只是数据的相对位置。
   - 若是阶数大的向阶数小的对齐，则阶数大的尾数值会变大，需要对尾数进行算术左移，若内存不够大很可能会引起最高有效位丢失。
   - 若是阶数小的向阶数大的对齐，则阶数小的尾数值会变小，需要对尾数进行算术右移，若内存不够大很可能会引起最后几位丢失，即精度下降，影响较小。
2. 尾数加减：阶数不变，对尾数进行相加减。
3. 规格化，将数据变为整数部分为0到9的数据：
   - 当尾数加减结果的第一位为0时需要左规，直到第一位不为0。
   - 当结果的整数部分大于等于10需要右规，直到整数部分只有一位。
4. 舍入：计算机中由于尾数的比特位有限，所以需要舍弃尾数的低位。对阶和右规格化都会引起舍入。舍入方法有：
   - 直接去除。
   - 非0进1。
   - 四舍五入。
5. 判溢出：若运算后阶码超过规定范围则溢出。尾数的溢出未必会导致整体溢出，可以通过第三四步来修补，但是阶码溢出一定会整体溢出。

浮点数加减运算

浮点数的加减基本上与科学计数法的加减一致。基本上浮点数的运算不可能使用*IEEE* 754的标准，因为位数太长不好计算。所以加减运算一律使用补码。

舍入的方法：

- “0”舍“1”入法：类似于十进制数运算中的“四舍五入”法，即在尾数右移时，被移去的最高数值位为0，则舍去；被移去的最高数值位为1，则在尾数的末位加1。这样做可能会使尾数又溢出，此时需再做一次右规。
- 恒置“1”法：尾数右移时，不论丢掉的最高数值位是“1”还是“0”,都使右移后的尾数末位恒置“1”。这种方法同样有使尾数变大和变小的两种可能。

假如加减的结果为11100,10.110001011，因为尾数符号位为10代表溢出了，所以需要规格化。

使用0舍1入法，将尾数整体右移，并将符号位的低位移到尾数高位，阶码加移位数，符号位修改，从而变成了11101,11.011000101，溢出1，最高位为1，使用0舍1入时需要加1，从而变成11101,11.011000110。（假如尾数为全1，则加1后又会高位溢出，还需要一次右规）。恒置1法的答案是一样的。

浮点数强制类型转换

类型	16 位机器	32 位机器	64 位机器
char	8	8	8
short	16	16	16
int	16	32	32
long	32	32	64
long long	64	64	64
float	16	32	32
double	64	64	64

无损转换：

- $char \rightarrow int \rightarrow long \rightarrow double$。
- $float \rightarrow double$。

由于定点数和浮点数不同，浮点数使用阶码+尾数的存储方式存储，所以定点数数值精度看位长就可以了，而浮点数数值精度看尾数长度，按$IEEE\ 754$标准有一个隐含的高位1，所以$double$尾数长度为53位。对于32位机器$long$是32位，所以转换到$double$的53位没有损失，而对于64位机器$long$是64位，$double$还是53位，这时候转换就会产生损失了。

int与$float$转换

- int：表示整数，范围$[-2^{31}, 2^{31}-1]$，有效数字32位。
- $float$：表示整数及小数，范围$\pm[2^{-126}, 2^{127} \times (2-2^{-23})]$，有效数字$23+1=24$位。
- $int \rightarrow float$：可能损失精度。
- $float \rightarrow > int$：可能溢出及损失精度。

算术逻辑单元

即运算器中的ALU。

原理

ALU 结构

- 输入信号有一个操作数的输入口，输出信号有一个运算结果输出口，旁边还有一个控制单元CU发出的控制信号接口，会输入指令译码。
- 机器字长就是指ALU可以同时处理多长的数据。为了保存结果，寄存器的字长与机器字长相等。

逻辑运算

- 与可以用*表示，如$A * B$。
- 或可以用+表示，如$A + B$。
- 优先级上类比乘法加法，与优先于或。
- 具有类似的分配律结合律。
- 与非：先与后非。
- 或非：先或后非。
- 异或：相异为1，相同为0。
- 同或：相同为1，相异为0。

加法器实现

一位全加器

- 一位全加器FA中，令被加数为A，被加数从低到高的位数为A_i，令加数为B，被加数从低到高的位数为B_i，令来自低位$i-1$的进位为C_{i-1}，本位的和为S_i。$S_i = A_i + B_i + C_{i-1}$。
- 输入A_i、B_i、C_{i-1}，输出S_i、C_i。
- 输入中有奇数个1时为1（异或）：$S_i = A_i \oplus B_i \oplus C_{i-1}$。
- 输入中至少两个1才会高位进1，一种情况是两个本位都是1，另一种情况是只有一个是1且来自低位的进位是1：$C_i = A_iB_i + (A_i \oplus B_i)C_{i-1}$。

串行加法器

- 基于一个一位全加器，一位一位串行进入加法器运算。对比一位全加器，需要保存一个进位触发器用来保存进位位，进位触发器初始化值为0。
- 如果操作数长n位加法就要分n次进行，每次产生一位和，并且串行逐位地送回寄存器。

串行进位并行加法器

- 把n个全加器串接起来，就可进行两个n位数的相加。前一个全加器的进位输出将作为下一个全加器的输入。
- 串行进位又称为行波进位，每一级进位直接依赖于前一级的进位，即进位信号是逐级形成的。所以不可能比单纯的串行全加器快很多。
- 加快进产生和提高传递速度是关键。

并行进位并行加法器

- 由于$S_i = A_i \oplus B_i \oplus C_{i-1}$，所以计算的重点是计算$C_i$，而$C_i = A_iB_i + (A_i \oplus B_i)C_{i-1}$，通过递归可以不断展开：$C_i = A_iB_i + (A_i \oplus B_i)(A_{i-1}B_{i-1} + (A_{i-1} \oplus B_{i-1})C_{i-2})$一直可以递归到$C_0$，而$A_i$、$B_i$、$C_0$都是一开始可以知道的，所以第$i$位向更高位的进位$C_i$可根据被加数、加数的第1到$i$位,再结合$C_0$即可确定。
- 将$G_i = A_iB_i$，$P_i = A_i \oplus B_i$，所以式子得到化简：$C_i = A_iB_i + (A_i \oplus B_i)C_{i-1} = G_i + P_iC_{i-1}$。
- 所以$C_1 = G_1 + P_1C_0$，$C_2 = G_2 + P_2C_1 = G_2 + P_2G_1 + P_2P_1C_0$，$C_3 = G_3 + P_3C_2 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1C_0$……所以一直到最后面每个$C_i$都可以用对应的$G_i$和$P_i$计算，这时候$C_i$可以不用递归来计算，可以用一开始就知道的$P_i$、$G_i$和$C_0$计算得到，从而这时候每一位的计算都可以一开始就同时进行了而不用依赖前面的计算。
- 各级进位信号同时形成，又称为先行进位、同时进位。
- G_i称为进位产生函数，因为G_i是通过A_i和B_i相与，只有同时为1才会产生1。
- P_i称为进位传递函数，P_i是通过A_i和B_i异或，实际上P_i为1时，只有此时C_{i-1}才能为1，否则为0。

单级先行进位加法器

- 称为组内并行、组间串行进位方式。
- 由于逻辑表达式越长就代表电路实现越复杂，所以一般会最多用4个FA和一些新的线路、运算逻辑组成一个运算单元进行串联进位计算。
- 组内的信息可以并行同时得到，但是组件信息需要串行进位。这时虽然实现简单，但是效率对比并行进位并行加法器还是下降了。

多级先行进位加法器

- 多级先行进位方式，又称为组内并行、组间并行进位方式。
- 为了解决这个问题按照并行进位并行加法器的思路继续对每一组的数据进行计算。
- 令$G_1^* = G_4 + P_4G_3 + P_4P_3G_2 + P_4P_3P_2G_1$为组进位产生函数，$P_1^* = P_4P_3P_2P_1$为组进位传递函数。一开始输入就能确定，所以这些都是一开始就能确定的。
- 所以$C_4 = G_1^* + P_1^*C_0$，$C_8 = G_2^* + P_2^*C_4 = G_2^* + P_2^*G_1^* + P_2^*P_1^*C_0$，$C_{12} = G_3^* + P_3^*C_8 = G_3^* + P_3^*G_2^* + P_3^*P_2^*G_1^* + P_3^*P_2^*P_1^*C_0 \cdots$
- 所以4位$BCLA$（成组先行进位电路）只要输出G_i和P_i就可以在计算机中计算得到C_{4i}了。

存储系统

存储器被划分为若干个存储单元，每个存储单元都从0开始顺序编号。比如一个存储器有128个存储单元，则它的编号范围是0 ~ 127。

存储器概念

存储部件概念

存储主体

- 存储元：由电路控制的单个存储部件。
- 存储单元：由同一个电路控制的一组同时读写的存储部件集合，一般为一行存储元，一行存储元的个数就代表一次存储的字长。
- 存储体：由多个存储单元构成的，多个电路控制的存储集合。
- 存储字：存储单元通电后由电信号表示可以读写的一个存储单元信息集合。存储字的位数就是存储字长，单位是bit。

存储控制

存储控制即通过电路选择读写的存储单元。一般通过译码器来控制哪个存储单元用来读写。

- 高电平有效：接收到高电平时代表该电路是工作的，接收到低电平时代表该电路是关闭的。
- 低电平有效：接收到低电平时代表该电路是工作的，接收到高电平时代表该电路是关闭的。若代表线路或开关的字母上有横杠代表低电平有效。否则一般都是高电平有效。

译码器是将电路信号进行翻译，转换为片选信号的部件。由于同时对多个存储单元进行读写操作时，数据可能发生混乱。所以就需要用对应数据的所在地址来区分不同的数据，译码器将输入的二进制代码翻译为高低电平信号。

- 假如使用n位地址码，有一种方法是用1代表当前读取的逻辑地址，即$00\cdots01$就代表1，$00\cdots10$就代表2，这时候每一位就对应一位，就只能表示n个存储单元。这是基本电平信号控制。
- 为了提升效率表示更多地址，所以需要译码器将电平信号转换为逻辑信号，将普通的01信号变为逻辑的二进制信号，从而能表示的地址就变成了2^n个。

输入口为2个电路，输出口为$2^2=4$的译码器就是24译码器，输入口为3输出口为$2^3=8$的译码器就是38译码器。所以总容量=存储单元个数×存储字长。

通过译码器图可以表示对应线路以及输入口和输出口，用来说明整个CPU情况。

- 假如译码器的输入口为A、B……N等n个。默认是高电平有效。
- 若高电平有效，则输入口字母为A、B……N，输出口字母为$\overline{Y_i}$，其中$i=2^n-1$。
- 若是低电平有效，则输入口字母为$\overline{A}$、$\overline{B}$……$\overline{N}$，输出口字母为$\overline{Y_i}$，其中$i=2^n-1$。并在字母的线路直线的靠近译码器方框的末端加上空心圈。

译码器应该能被控制进行打开和关闭，所以译码器除了输入口和输出口还有使能端。

- 用于控制译码器的启动和关闭，若图上没有圈或字母上没有横杠则代表高电平有效，使能端输入1译码器就能开始工作。实际上会有多个使能端，只有高电平有效的使能端输入1，同时低电平有效的使能端输入0才能启动，只要输入有一个异常都不会工作，类似输入密码。

主存结构

主存读写

CPU要想实现数据的读写操作，就必须与外部器件（芯片）进行以下三类信息的交互：

1. 地址信息，即存储单元的地址。
2. 控制信息，即对存储器的存储器件（芯片）的选择——读或写。
3. 数据信息，即将要用于读或写的数据。

所以在主存结构中必须包含传输这三种信息的线路：地址总线、控制总线、数据总线。

1. 首先CPU通过地址总线把地址信息传递至存储器，对应地找到目标存储单元。
2. CPU又通过控制总线把控制信息（读操作或写操作）传递至存储器，找到对应读或写的芯片器件。
3. CPU最后通过数据总线把将要被读或被写的数据信息传递至目标存储单元，执行数据的读或写。

主存构成

- 存储矩阵：由大量相同位存储单元阵列。
- 译码驱动：将来自地址总线的地址信号翻译成对应存储单元的选通信号，该信号在读写电路的配合下未完成对被选中的单元的读写操作。
  - 译码器：将MAR输入的地址进行译码，选择选中的存储单元地址。
  - 驱动器：根据译码器提供的地址，通过驱动器获取对应存储单元。
- 读写电路：包括读出放大器和写入电路，用来完成读写操作。根据控制电路的读或写操作要求，将 MDR 传入的数据写入存储体中，或将存储体中的数据读出到MDR中。
- MAR：地址寄存器，保存从地址总线输入的地址。
- MDR：数据寄存器，保存读出或写入的数据。
- 地址线A_i：用来输入CPU要访问的主存地址，是单向的，位数与CPU芯片容量相关，一般与MAR位数相等。直接连接MAR。多少个存储单元就多少根地址线。地址线的多少决定 CPU 寻址能力。

- 数据线D_i：用来输入输出数据，是双向的，位数与读入写入数据位数相关。直接连接MDR。存储字长多少位就多少根数据线。数据线的多少决定 CPU 数据吞吐能力。
- 地址线和数据线位数同时决定的内存大小，假如地址线有N根，数据线有M根，则芯片容量为$2^N \times MB$。地址线位数代表存储体的行，数据线代表存储体的列。
- 片选线：是整个存储芯片的开关，用来确定哪一个存储芯片被选中，可用于容量扩充。
  - $\overline{CS}$：芯片选择信号，选择指定芯片，低电平有效。
  - $\overline{CE}$：芯片使能信号，打开指定芯片进行存储，低电平有效。
- 读写控制线：决定芯片进行何种操作。控制线的多少决定CPU可以控制的外部部件的数量。
  - 如果是一根就用$\overline{WE}$表示，低电平写，高电平读。如果是WE则反之。
  - 如果是两根，则$\overline{OE}$低电平表示允许读，$\overline{WE}$低电平表示允许写。如果是OE和WE则反之。

主存分配

- 指按照不同的长度切分存储单元。
- 若字长为$4B$，总容量为$1KB$，则按字节寻址是$1K$个单元，每个单元$1B$；按字寻址是256个单元，每个单元$4B$；按半字寻址是512个单元，每个单元$2B$；按双字寻址是128个单元，每个单元$8B$。

存储器的分类

作用

- 主存储器：主存或内存。
  - 存放计算机运行间的程序和数据。
  - CPU、$Cache$，能直接访问。
  - 容量小、速度快、价格高。
- 辅助存储器：辅存或外存。
  - 存放暂时不用的或永久的数据。
  - 不能与CPU直接交换信息。
  - 容量大、速度慢、成本低。
- 高速缓冲存储器$Cache$。
  - 存放正在执行的程序和数据。
  - 在CPU中。
  - 容量小、速度快、价格高。

存储介质

- 磁表面存储器：

    - 磁盘。
    - 磁带。
- 磁芯存储器。
    - MOS型存储器。
    - 双极型存储器。
- 光存储器：光盘。
- 半导体存储器。

存取方式

- 随机存取：
    - RAM：随机存取存储器：
        - $DRAM$：动态。
        - $SRAM$：静态。
    - ROM：只读存储器。
- 串行访问：
    - 直接存取：磁盘。
    - 顺序存取：磁带。

信息可保存性

- 断电后信息是否消失：
    - 易失性：RAM。
    - 非易失性：磁带、ROM。
- 破坏性，存取是否影响存储内存：
    - 破坏性读出：$DRAM$。
    - 非破坏性读出。

存储器性能指标

1. 存储容量：存储字数×字长（如$1M \times 8$位）。
2. 单位成本：每位价格=总成本÷总容量。
3. 存储速度：数据传输率=数据的宽度÷存储周期：
    - 存储周期=存取时间+恢复时间。
    - 存取时间（T_a）：存取时间是指从启动一次存储器操作到完成该操作所经历平均的时间，分为读出时间和写入时间。
    - 存取周期（T_m）：存取周期又称为读写周期或访问周期。它是指存储器进行一次完整的读写操作所需的金部时间，即连续两次独立地访问存储器操作（读或写操作）之间所需的最小时间间隔。
    - 主存带宽（B_m）：主存带宽又称数据传输率，表示每秒从主存进出信息的最大数量，单位为字/秒、字节/秒（B/s）或位/秒（b/s）。

存储器层次化结构

- CPU。
- $Cache$：为了解决CPU的高速与主存之间的低速速度不匹配的问题，由硬件自动完成。
- 主存。
- 辅存：为了解决主存容量不足的问题，由硬件和操作系统共同完成。
- 虚拟存储系统。

半导体随机存储器

主存由$DRAM$实现，$Cache$由$SRAM$实现。

SRAM 和 DRAM

随机存取存储器RAM分为$SRAM$和$DRAM$，即静态与动态。

SRAM 和 DRAM 的区别

类型	SRAM	DRAM
存储信息	双稳态触发器，分为 0 态和 1 态	电容，充电是 1，否则为 0
破坏性读出	非；读只用查看触发器状态；写只用改变触发器状态	是；读需要连接电容，检测电流变化，电流随着电路连通而溜走；写需要给电容充放电
需要刷新	不要，能保持两种稳定的状态	需要，因为电容上的电荷只能维持 2ms
送行列地址	同时送，因为地址分为行地址和列地址一同发送，需要一根片选线	分两次送，行地址和列地址分开，所以地址线可以复用，线路减少一半，需要一根行通选线和一根列通选线
运行速度	快	慢
集成度	低，6 个逻辑元件构成	高，1 个或 3 个逻辑元件构成
发热量	大	小
存储成本	高，常用于$Cache$	低，常用于主存

注意：$SRAM$不地址复用，而$DRAM$地址复用。（地址线只有原来的一半）

存储器芯片结构

- 存储体：由行选择线X和列选择线Y选择访问单元。

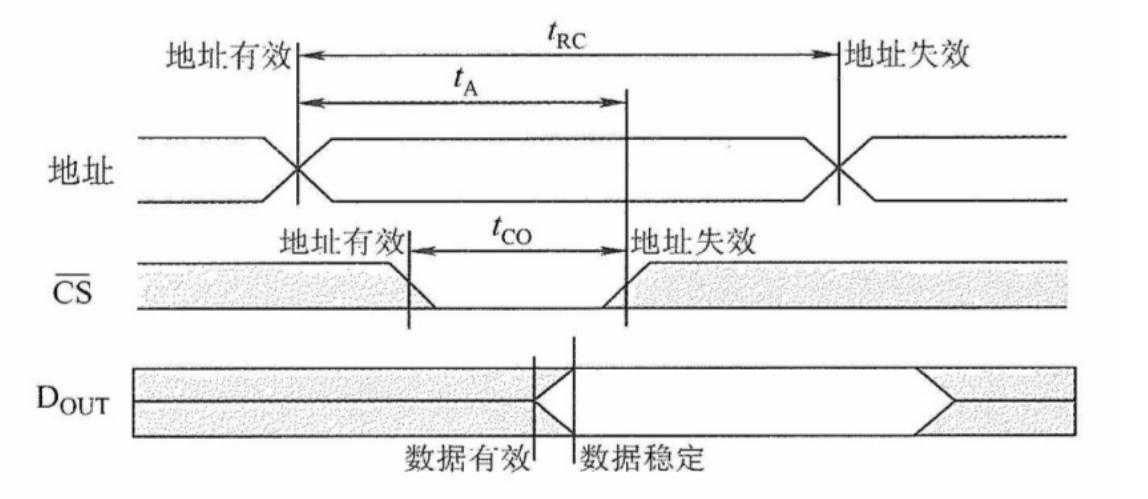
- 地址译码器：将地址转换为译码输出线上的高电平，以便驱动对应读写电路。
- I/O控制电路：控制选中单元读出写入，并放大信息。
- 片选控制信号：产生片选控制。
- 读/写控制信号：输入读或写命令。

DRAM 的刷新

- 刷新周期：一般为$2ms$。
- 刷新单元数：以行为单元，每次刷新一行存储单元。
  - 如果译码器有n位，则可以寻址2^n个，也就需要2^n与存储单元连接的线路，很难实现。
  - 将地址拆分为行列地址（$DRAM$行、列地址等长）。
  - $SRAM$需要2^n条线路，而$DRAM$需要$2^{\frac{n}{2}+1}$根线。
- 刷新方式：硬件支持，读出一行的信息后重新写入整个行，占用一个读写周期。
- 刷新时刻：假设$DRAM$内部结构排列为128×128的形式，读写周期为$0.5\mu s$，所以$2ms$一共4000个周期（注意针对刷新问题，读写时间不是重点，即无论是否读写或者读写多少行，都要在固定时间进行刷新所有行）：
  - 分散刷新：
    - 每读取完一行数据就刷新一次。
    - 如在每存取周期$1\mu s$中前0.5用于读写，后0.5用于刷新该行。
    - 没有死区，但是加长了系统存取周期，降低整机速度。
  - 集中刷新：
    - 有一段时间专门刷新，但是这时候就无法访问存储器，称为访存死区。该段时间为死时间。
    - 因为有128行，刷新需要128个周期的时间。所以一共需要专门刷新$128 \times 0.5 = 64\mu s$，则前面正常读写时间为$2000 - 64 = 1936\mu s$，读写需要3872个周期。
    - 读写时间不受刷新工作的影响，但是存在死区。
  - 异步刷新：
    - 隔一段时间刷新一次，一次要刷新所有的行，而如果将刷新设置在不需要访存的译码时间可以加大利用效率。
    - 将刷新周期除以行数，得到两次刷新操作之间的时间间隔t，利用逻辑电路每t时间产生一次刷新请求。
    - 因为每隔$2ms$要刷新128行即128次，所以平均每个时间周期为$2ms \div 128 = 15.6\mu s$，$15.6\mu s$中要读写数据并刷新一次即一行，所以每$15.6\mu s$中有$0.5\mu s$的死时间，其中前$15.6 - 0.5 = 15.1\mu s$用来读写。

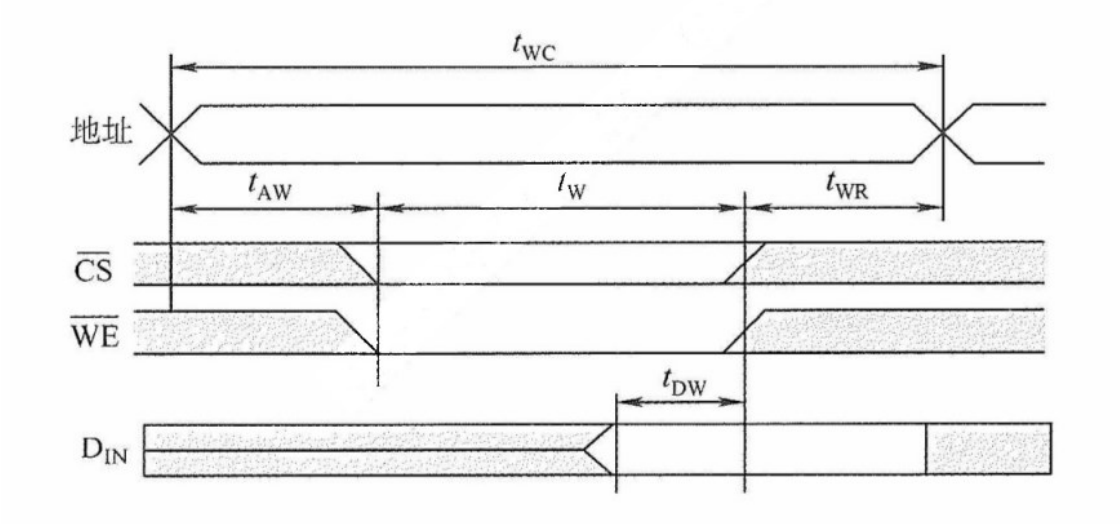
RAM 的读写周期

- 读周期：
  - 从给出有效地址开始，到读出所选中单元的内容并在外部数据总线上稳定地出现所需的时间，称为读出时间（t_A）。从数据稳定到数据有效之间存在一个时间缝隙，因为数据线上的信号速度是不一样的，所以需要这个缓冲。
  - 地址片选信号$\overline{CS}$必须保持到数据开始稳定输出，t_{CO}为片选的保持时间，即发出片选信号的从地址有效到地址失效的时间，在读周期中$\overline{WE}$为高电平。
  - 读周期与读出时间是两个不同的概念，读周期时间（t_{RC}）表示存储芯片进行两次连续读操做时必须间隔的时间，因为里面存在要等待数据稳定才能开始读的等待时间等其他时间，所以必然大于等于读出时间。



读周期

- 写周期：
  - 要实现写操作，要求片选信号$\overline{CS}$和写命令信号$\overline{WE}$必须都为低电平。
  - 为使数据总线上的信息能够可靠地写入存储器，要求$\overline{CS}$信号与$\overline{WE}$信号相“与”的宽度至少为t_{WC}。
  - 为了保证在地址变化期间不会发生错误写入而破坏存储器的内容，$\overline{WE}$信号在地址变化期间必须为高电平。
  - 为了保证有效数据的可靠写入，地址有效的时间至少应为$t_{WC} = t_{AW} + t_W + t_{WR}$。其中$t_{AW}$和$t_{WR}$为写入前和写入后必须的间隔时间，$t_W$为写入的时间。
  - 为了保证在$\overline{WE}$和$\overline{CS}$变为无效前能把数据可靠地写入，要求写入的数据必须在t_{DW}以前在数据总线上已经稳定。



写周期

ROM

只读存储器ROM即使断电也能保存数据，主存不能直接与CPU相连，所以一定会出现ROM来完成这个工作。

ROM写入速度不如RAM，所以一般还是用来保存信息而不用于大量的写。

ROM 的分类

- 掩膜式只读存储器（$MROM$）：存储内容由半导体制造厂按用户提出的要求在芯片的生产过程中直接写入，无法修改。
- 一次可编程只读存储器（$PROM$）：存储内容由用户用专门的设备（编程器）一次性写入，之后无法修改。
- 可擦除可编程只读存储器（$EPROM$）：先全部擦除数据然后编程。修改次数有限，写入时间长：
  - 紫外线擦除（$UVEPROM$）。
  - 电擦除（$EEPROM$）。
- 闪速存储器（$Flash\ Memory$）：如U盘，写入速度快。
- 固态硬盘（$Solid\ State\ Drives$）：控制单元+$FLASH$芯片。

主存与 CPU 连接

对CPU来讲，系统中所有物理存储器中的存储单元都处在一个统一的逻辑存储器中，这个逻辑存储器的容量大小受到 CPU 寻址能力的限制。如果一个CPU的地址总线宽度为10，则该CPU可以寻址的存储单元为$2^{10} = 1024$个，这1024个可寻到的存储单元就构成了这个CPU的内存地址空间，也叫做逻辑存储器。

所以对于CPU而言，它有一个固定的内存地址大小，对应的地址就是个逻辑地址，所以CPU读入的数据有限，一次性处理的能力有限；而内存（如内存条）可以扩充，

其实际地址就是物理地址，由操作系统给CPU的逻辑地址映射物理地址，还包括替换算法等一系列处理方案。

连接原理

CPU与内存通过总线连接。

- MDR和MAR虽然为寄存器，但是现在一般集成在CPU上。
- 数据总线直接连接在MDR上，可以写入也可以读出，是一个双向的。
- 地址总线直接连接在MAR上，将CPU的地址要求交给主存，是一个单向的。其位数决定可寻址最大内存空间。
- 控制总线向主存发送控制类型，如读写要求，是一个单向的。

主存容量扩展

为了获取更多的容量，所以需要对主存容量进行扩展。

位扩展法

- CPU的数据线数与存储芯片位数不一定相等，用多个存储器件对数据位数进行扩展（一次性输入输出数据数量）。
- 地址总线和片选线都是并联的，数据总线连接在每一块芯片上。
- 因为需要拓展位，所以一次性需要处理所有芯片的数据，从而需要对芯片同时进行片选线同步信号，所以所有芯片的$\overline{CS}$都可以连接在一起。
- 每个芯片各输入输出一部分数据。

字扩展法

- 增加存储器中字的数量（数据的地址大小即能保存的数据的数量），而位数不变，字扩展将芯片的地址线、数据线、读写控制线相应并联，与位扩展法连接方式一样。
- 但是如果每个芯片同时输入输出输数据则CPU无法区分到底是哪个芯片存储的数据，所以不能再将片选线连在一起同时控制。
- 需要用片选信号区分个芯片地址范围，即将每个芯片的片选线依次连接在CPU的地址线接口上，因为不能同时工作，所以片选线信号CS或$\overline{CS}$不会同时为1或0，而片选线的信号连接在CPU的地址线接口上就相当于将片选线的信号也作为芯片存储地址。
- 如一个CPU一共有16个地址线接口A_0到A_{15}与8个数据接口D_0到D_7以及一个读写控制线$\overline{WE}$，现在有两个$8K \times 8$位的存储芯片，首先按位扩展时将两个芯片的地址线A_0到A_{12}全部串联接到CPU的A_0到A_{12}接口上，芯片读写线WE串联接到CPU的WE接口上，数据线D_0到D_7借到CPU数据接口D_0到D_7上。如果这时CPU发出地址信号$0\,0000\,0000\,0000$，就无法识别这个地址是第一块还是第二位芯片的第一位。此时CPU的A_{13}到A_{15}三个接口还是空的，将芯片 1 的选片线CS接到CPU的A_{13}上，芯片 2 的选片线CS接到CPU的A_{14}上，此

时CPU的地址信号会变成 15 位。因为CS指高电平1有效，A_{13}为1代表选择芯片 1，A_{14}为1代表选择芯片2，所以010 0000 0000 0000代表选择芯片1，100 0000 0000 0000带包选择芯片2。若高位为110或000则冲突而浪费位数。

- 每个芯片各存储一部分数据。

字位同时扩展法

即增加存储字的数量又增加存储字长。各芯片连接地址线的方式相同，但是连接数据线的方式不同，需要通过片选信号$\overline{CS}$或采用译码器设计连接到对应芯片。

存储芯片片选

- 线选法：
  - 当某地址线信息为“0”时，就选中与之对应的存储芯片，只能一位有效。
  - 优点：不需要地址译码器，线路简单。
  - 缺点：地址空间不连续，选片的地址线必须分时为低电平（否则不能工作)，不能充分利用系统的存储器空间，造成地址资源的费。
- 译码片选法：
  - 由于译码器可以将n位映射到2^n位，所以通过地址译码芯片产生片选信号。如线选法如果三位编码只能选择三个芯片，而译码片选法三位编码可以选择八个芯片，即三位二进制编码。
  - 优点：地址空间可连续，可以增加逻辑设计。
  - 缺点：电路逻辑复杂。

双端口 RAM 和多模块存储器

由于CPU与主存相连，而CPU速度增长是指数级别的，而主存容量增长是线性的，所以双方速度不匹配，所以需要更快的访问速度。一方面可以更高性能存储器，一方面使用高速缓冲存储器。

存取周期由存取时间和恢复时间构成，所以缩短存取周期的方法一种是空间上并行（双端口存储器技术），一种是时间上并行（多模块存储器技术）。

双端口 RAM

左右有两个独立的端口，所以有两对独立的数据线、地址线、控制线可以同时对主存进行操作，如果不是一个位置不会发生异常。

两个端口对同一主存操作有以下4种情况：

1. 两个端口不同时对同一地址单元存取数据。
2. 两个端口同时对同一地址单元读出数据。
3. 两个端口同时对同一地址单元写入数据。会产生写入错误

4. 两个端口同时对同一地址单元，一个写入数据，另一个读出数据。会产生读出错误。

所以只用设置一个“忙”的标志位，若CPU发现该端口为忙，则等待一段时间再进行访问。

多模块存储器

单体多字存储器

由于CPU的速度远快于主存，所以同时从主存中拿出多个指令就能让CPU等待I/O时间变短，从而提高效率，单体多字存储器以此而实现。

普通存储器是每行为一个存储单元，而对于单体多字存储器来说，每个存储单元存储m个字，若总线宽度也为m个字，则一次并行就能读出m个字。但是只有数据和指令是连续存放在内存的才能这样操作，转移指令无效。

多体并行存储器

每个模块都有相同的容量和存取速度，以及独立的读写控制电路、地址寄存器和数据寄存器。地址分为体号和体内地址两个部分。

- 高位交叉编址的多体存储器：
  - 高位是体号，低位是体内地址。
  - 按列，模块里先编址，一个个模块进行分配。只是相当于扩容而已，对于速度没有改变。
  - 若连续取n个存储字，每次访问需要T的时间，则耗时nT。
- 低位交叉编址的多体存储器：
  - 低位是体号，高位是体内地址。
  - 按行，每一个单元先编址，一行行进行分配。
  - 由于每个存储体都是独立的，所以可以间隔一小段时间就能进行另一个的存储单元的访存而不用等待上一个单元的阶数。这就要求其模块数必须到达一个值，从而保证能流水线运行而不会卡住。
  - 设模块字长等于数据总线宽度，模块存取一个字的存取周期为T，总线传送周期为r，所以存储器交叉模块数应该大于等于$m = T/r$。这个值被称为交叉存取度。
  - 从而启动该模块后能保证经过$m \times r$的时间后再次使用该模块时上次存取操作已经完成。
  - 若连续取n个存储字，每次访问需要T的时间，启动间隔为τ，则耗时$T + (n-1)\tau$。

如高位交叉编址，体号+体内地址：

M0	M1	M2	M3

M0	M1	M2	M3
0000	0100	1000	1100
0001	0101	1001	1101
0010	0110	1010	1110
0011	0111	1011	1111

这时按顺序访问就是竖着的。

低位交叉编址，体内地址+体号：

M0	M1	M2	M3
0000	0001	0010	0011
0100	0101	0110	0111
1000	1001	1010	1011
1100	1101	1110	1111

此时顺序访问就是横着的，多个存储体可以共同输入输出数据，可以流水线操作。

对于交叉存储器的存取速度：

- 模块数为m，存储周期为T，字长为W，数据总线宽度为W，总线传输周期为r，连续存取n个字，求交叉存储器的带宽。（有m个存储体，存储周期为T，字长为W，每隔r时间启动下一个存储体，连续存取n个字，求交叉存储器的存取速率。）
- 连续存取n个字耗时为$T+(n-1)r$，但是需要$m \geq \frac{T}{r}$。因为如果模块数过少，则轮流到某一个存储体时这个存储体上一次的处理还没有完成就无法继续工作了。
- 所以带宽是$\frac{n \times W}{T+(n-1)r}$。
- 当n无限大时，带宽趋近于$\frac{W}{r}$，而单个存储体的带宽为$\frac{W}{T}$。

多端口存储器是对同一个存储体使用多套读写电路实现的，扩大存储容量的难度显然比多体结构的存储器要大，而且不能对多端口存储器的同一个存储单元同时执行多个写入操作，而多体结构的存储器则允许在同一个存储周期对几个存储体执行写入操作。

高速缓冲存储器

在操作系统中也谈论过$Cache$。

并注意访存指的是访问主存而不包括$Cache$。

高速缓冲存储器基本概念

局部性原理

- 时间局部性：程序所访问的数据在相邻时间也可能访问到。
- 空间局部性：程序所访问的数据的周围数据也可能访问到。

高速缓冲存储器地址

- 主存储器的地址包括主存块号和块内地址，而$Cache$的地址包含缓冲块号和块内地址，两个块内地址都是一样长的且完全相同的。
- $Cache$块又称为$Cache$行。长度为块长或$Cache$行长。
- $Cache$还有一个标记，用来说明$Cache$块与主存块的关系，等于此块在主存中的块号。

工作流程

1. CPU发出访问地址，从地址总线传输到$Cache$。
2. 通过$Cache$主存地址映射变换结构，将主存地址转换为$Cache$地址，在$Cache$中寻址对应数据。
3. 如果命中就访问$Cache$并取出指令通过数据总线返回CPU。
4. 如果不命中，就直接访问主存，取出信息通过数据总线返回CPU，并把这个信息存储在$Cache$中。
5. 需要检测$Cache$是否已满，如果不满就直接将新的主存块调入$Cache$进行存储。
6. 若已经满了则通过$Cache$替换结构，执行替换算法腾出空位再调入。

其中CPU和$Cache$之间数据交换以字为单位，而$Cache$和内存之间数据交换以块为单位。

命中与未命中

- 命中：主存块调入缓存，主存块与缓存块建立了对应关系，用标记记录与某缓存块建立了对应关系的主存块号。
- 未命中：主存块未调入缓存，主存块与缓存块未建立对应关系。
- 命中率：CPU欲访问的信息在$Cache$中的比率，设一个程序执行期间，$Cache$的总命中次数为N_c，访问主存的总次数为N_m，则命中率$H = \frac{N_c}{N_c+N_m}$。
- 命中率与$Cache$的容量与块长有关。一般每块可取4到8个字，块长取一个存取周期内从主存调出的信息长度。
- 访问效率=$Cache$访存时间÷平均访存时间。
- 相联存储器：并行比较标记，若有标记与当前将要访问的地址的标记相同，且有效位为 1（表示当前存储单元存储数据），则命中；若标记不同，则直接替换。

高速缓冲存储器的效率

- 效率e与命中率有关，e =访问$Cache$的时间/平均访问时间$\times 100\%$。
- 假如$Cache$和主存是同时被访问的，设$Cache$命中率为h，访问$Cache$的时间为t_c，访问主存的时间为t_m，则$e = \frac{t_c}{h \times t_c + (1-h) \times t_m} \times 100\%$。当$h = 0$时最小为$\frac{t_c}{t_m}$，当$h = 1$时最大为1。

地址映射

即将主存块调入$Cache$时应该将其放在哪里。

地址以字节为单位。

默认都需要一位有效位。

注意：$Cache$字块标记是指$Cache$有多少行，而不是有多少容量，往往以字为单位。

直接映射

存储方式：

- 对号入座，每一个主存块只能存放在唯一一个地方。
- 将主存储体按$Cache$存储体的长度划分为多个区，主存的每个区只能放在$Cache$的指定区域中。类似于模运算，将主存长度对$Cache$长度取模，地址相同余数的内存块放在同一个$Cache$块中。

地址组成：

- 地址=主存字块标记+$Cache$字块标记+字块内地址。
- $Cache$地址=$Cache$字块标记+字块内地址。
- 主存字块标记为主存容量除以$Cache$容量，表示要用多少位来区分主存地址。
- $Cache$字块标记，即$Cache$行号的位数为$Cache$的块数的二对数。
- 字块内地址就是块内地址，位数为每块容量的二对数。

附加位：

- 这种映射方式需要一位有效位标识是否有数据存储。
- 若发生冲突，不需要替换算法直接替换出去。所以不需要替换位。
- 但是与全相连映射不同的是，因为直接映射是根据模运算来存储的，所以行与行之间是顺序的关系，所以主存块号不需要全部存入标记项，把能区分相同模结果的地址前一部分位的数据存入即可。

特点：

- 优点：节省掉$Cache$字块标记，有效位存储的地址减少，实现难度降低。
- 缺点：空间利用率降低。

如$Cache$一共八行，则主存按八为一个周期存储，则如$Cache$的0号地址中一定存入主存块地址模八后余0的主存块，即地址都为为$xx\cdots000$，同理1号地址存的都是$xx\cdots001$，所以可以保存主存地址减三的前面位数就足够了。这里的000和001等类似于$Cache$的组号。若$Cache$的行数为2^n，主存块号地址为c位，则标记项只用保存$c-n$位就可以了。

全相联映射

存储方式：

- 空位随意放。
- 但是这时候还是不知$Cache$里的每一个存储单元存的是主存的哪一块数据，所以还需要将主存的主存块号保存到标记项中。
- 因为是乱序存放，所以一定要把整个主存块号都放入标记项中。

地址组成：

- 地址=主存字块标记+字块内地址。

附加位：

- 因为无法查看$Cache$里面每个存储块是否有数据，所以还需要一个有效位来表示里面是否有数据，将有效位放入标记项中。

特点：

- 优点：冲突概率低；空间利用率高，命中率高。
- 缺点：标记比较速度慢，实现成本高（相联存储器）。

组相联映射

存储方式：

- 按号分组，组内随意放，结合了上面二者的优点。
- 标记项也需要存入能区分数据块的部分地址。
- 将$Cache$的块分为几个组，主存不再按照$Cache$的行数进行模运算分组存入，而用$Cache$组的个数进行模运算，虽然这样能节省的位数变少，但是这样主存的某一块就可以在每一组内随机选一个存储，而不用只能存储在一个块内浪费其他块的空间。
- 每组有r个行，则称为r路组相联。

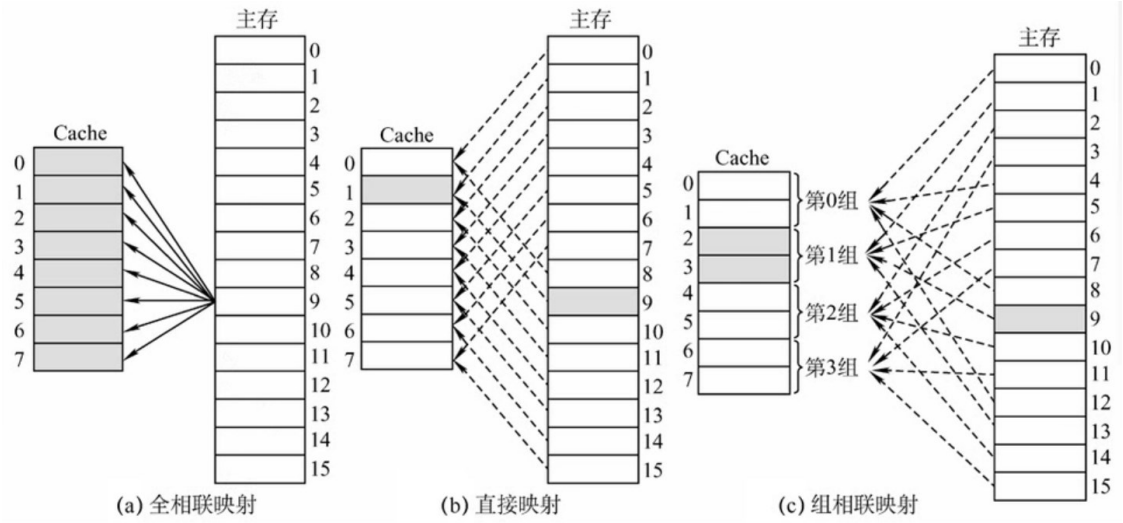
地址组成：

- 地址=主存字块标记+$Cache$组号+字块内地址（$Cache$每行长度）。
- 一般物理地址的中间部分直接映射为组号。

- 当$Cache$组地址位数长度变为表示$Cache$行数的二进制位，那么组号和行号一样就变成了直接映射，若组地址位数为1全部为一个组，则$Cache$不分组都随机存储即变成了全相连映射。

附加位：

- 需要有效位。



地址映射

标记整列

称为地址映射表，即标记项总体长度，即总$Cache$行数×标记项的长度。

标记项的长度=标记位长度+其他位长度（如有效位、替换控制位、一致控制位即脏位）。

- 有效位默认是一位。
- 替换控制位主要用于组相联映射方式，其他两种方式基本上没有这个。多少路组相联的二对数对应多少位替换控制位。如两路组相联对应标记项的一位替换控制位，物理地址的$Cache$组标记为三位对应八组，则一共有十六行$Cache$行。
- 脏位默认是一位。如果是写回法和写分配法则需要，全写法和非写分配法则不需要。

标记位长度=主存地址长度（即主存总容量位数而不是$Cache$容量长度）-其他地址长度（块内地址，即$Cache$行的数量的对数；组地址，即多少个组的对数）。

标记的主要计算在于主存块标记长度，即标记位长度，为什么是这个长度？为什么要如此计算？

我们要将主存的内容放到高速缓冲存储器中，所以我们要根据主存的地址找到$Cache$的地址，并能通过$Cache$的地址找到主存的地址。由于$Cache$的数量远小于主存的数量，所以我们使用不同的方式将主存映射到$Cache$中。我们通过替换映射算法能得到一部分主存和$Cache$的映射规则，但是这种规则只能得到不完整的地址信息，那么其他相关地址信息就是通过标记项来给出，通过标记项和$Cache$位置信息就能完整映射$Cache$到每一个主存地址。

因为是主存地址映射，所以我们自然要映射到每一个主存地址，所以主存地址长度是我们的目标，通过地址信息加和得到所有主存地址。由于主存与$Cache$之间的数据交换单位为块，所以只要给定一个$Cache$块在主存中的首地址，那么一个块内的其他数据的地址也是可以推算的，所以这部分的信息可以在地址总线传输过来的块内偏移量中得到，标记位中就可以不要这个信息，所以主存地址长度减去块内地址长度。通过映射方式不同，我们可以得到每一块或每一组$Cache$与主存之间的映射关系，确定映射关系后，地址总线传输块号或者组号，所以这部分内容也是可以减去的，即主存地址长度再减去块号长度或块组号长度。最后的位数就是标记长度，通过标记位能与块号/组号+偏移地址得到主存地址。

如某计算机的主存地址大小为$256MB$，按字节编址，有8个$Cache$行，行长$64B$，使用直接映射方式，求标记项长度。首先主存地址大小为$256MB = 2^{28}B$，则主存地址总长度为28位；然后行长$64B = 2^6B$，则块内偏移地址长度为6位，所以$28 - 6 = 22$，使用直接映射方式，一共8行，所以行号位长度为3位，所以$22 - 3 = 19$，标记位一共19位。

替换算法

即$Cache$中存储满了如何处理。对于直接映射法来说，固定替换出前一个数据存入此时的数据，所以替换算法只针对全相联映射和组相联映射。

随机算法

即$RAND$算法：随机地确定替换的$Cache$块。它的实现比较简单，但没有依据程序访问的局部性原理，故可能命中率较低。但是这种方式基本上不会使用。

先进先出算法

即$FIFO$算法：选择最早调入的行进行替换。它比较容易实现，可以使用堆栈，但也没有依据程序访问的局部性原理，可能会把一些需要经常使用的程序块（如循环程序）也作为最早进入$Cache$的块替换掉。

近期最少使用算法

即LRU算法：依据程序访问的局部性原理选择近期内长久未访问过的存储行作为替换的行，平均命中率要比$FIFO$要高，是堆栈类算法。是一种局部性策略。

LRU算法对每行设置一个计数器，$Cache$每命中一次，命中行计数器清0，而其他各行计数器均加1，需要替换时比较各特定行的计数值，将计数值最大的行换出。

最不经常使用算法

即LFU算法：将一段时间内被访问次数最少的存储行换出。每行也设置一个计数器，新行建立后从0开始计数，每访问一次，被访问的行计数器加1，需要替换时比较各特定行的计数值，将计数值最小的行换出。是一种全局性策略。

但是这种策略是无法实现的，因为计算机不可能全局考虑后面会出现什么，而只会考虑局部。

写策略

即$Cache$中内容修改后如何让主存于$Cache$修改的保持一致。分为当前命中和当前不命中的情况：

- 命中：全写法、写回法。
- 不命中：写分配法、非写分配法。

标记项结构：有效位+脏位+替换控制位+标记位。

全写法

也称为写直通法或$write-through$：当CPU对$Cache$写命中时，必须把数据同时写入$Cache$和主存。

由于写$Cache$要远快于写主存，所以一般使用写缓冲（$writebuffer$）暂存写入的数据，但是如果写的速度过快可能会出现溢出。

写回法

也称为$write-back$：当CPU对$Cache$写命中时，只修改$Cache$的内容，而不立即写入主存，只有当此块被换出时才写回主存（全部改完了再写回主存）。

需要在信息位中使用一个类似于有效位的1/0表示位，名为脏位，0代表未修改，1代表修改。

写分配法

也称为$write-allocate$：把主存中的块调入$Cache$，在$Cache$中修改。一般搭配写回法使用，即写完$Cache$之后再把$Cache$的值覆盖在主存的数据上，所以也需要设置一个脏位。

非写分配法

也称为$not-write-allocate$：只写入主存，不调入$Cache$。一般搭配全写法使用。

多级 Cache

使用两级$Cache$，在与CPU直接连接的第一层$Cache$使用全写法，在与主存直接连接的第二层$Cache$使用写回法。

虚拟存储器

$Cache$为了解决主存和CPU之间的问题，而虚拟存储器为了解决主存和辅存之间的问题。虚拟存储器包括主存和辅存，将主存和辅存联合在一起统一编址。

- 是一个逻辑模型。
- 用户给出一个地址，叫做虚地址或逻辑地址，虚拟存储器要给出该地址对应的数据。
- 有辅助硬件将虚地址映射到主存中某个单元，主存单元地址称为实地址或物理地址。虚地址远大于实地址。
- 基于局部性原理：在程序执行过程中，程序对主存的访问是不均匀的。
- 由操作系统和一部分硬件完成地址映射。

具体内存参考操作系统，主要考页式存储，段式非常少。

对于虚拟存储器的三个地址空间：

- 实地址=主存页号+页内字地址。
- 虚地址=虚存页号+页内字地址。
- 辅存地址=磁盘号+盘面号+磁道号+扇区号。

注意：这里地址空间基本上都是用字地址的，即以机器字长为单位进行编址，这是因为虚拟存储器面向程序员，所以以逻辑的每一个字为单位；而上一讲的高速缓冲存储器面向的机器本身，所以经常以字节为单位进行编址。

注意：虚拟系统中，找到的地址后，进行地址映射是靠操作系统完成，在操作系统的书中会更详细讲到。

页式虚拟寄存器

- 虚拟空间与主存空间都被划分成同样大小的页，主存的页称为实页，虚存的页称为虚页。
- 虚页地址分为虚页号和页内地址，实页地址分为实页号和页内地址，在虚页转换为实页时，页内地址是相同的，唯一要考虑的是虚页号如何转换为实页号。
- 做法就是将这种映射关系存在一张表中，这张表就是页表。页表存储的是实页号和装入位，用来标识虚拟地址是否装入主存地址。页表长期存储在主存中。

- 首先页内基表寄存器保存着页表起始地址，和虚页号进行计算就得到了页表项的地址，就可以找到页表项，把页表项中的实页地址取出并进行拼接就得到了页内地址。
- 若在$Cache$中存有则访问$Cache$，若没有再访问主存。
- 页表包括有效位（装入位）、脏位、引用位、物理页或磁盘地址。
- 优点：长度固定，页表简单，调入方便。
- 缺点：程序不可能总是页面的整数倍，所以最后一页的部分空间会浪费；页不是逻辑独立的实体，所以保护、处理、共享不便。

对于页表的大小选择需要适度。页面若很小，虚拟存储器中包含的页面数就会过多，使得页表的体积过大，导致页表本身占据的存储空间过大，同一个程序需要调用更多页表，使操作速度变慢；当页面很大时，虚拟存储器中的页面数会变少，由于主存的容量比虚拟存储器的容量小，主存中的页面数会更少，每次页面装入的时间会变长，每当需要装入新的页面时，速度会变慢。

段式虚拟寄存器

- 段式虚拟存储器中的段是按程序的逻辑结构划分的，各个段的长度因程序而异。
- 虚拟地址分为两部分：段号和段内地址。
- 段表：每一行记录了与某个段对应的段号、装入位、段起点和段长等信息。
- 由于段的长度可变，所以段表中要给出各段的起始地址与段的长度。
- 段表的段首址加上虚拟地址的段内地址就得到了物理地址。
- 优点：按逻辑划分，所以利于编译、管理、修改、保护、共享。
- 缺点：段长度可变，所以分配空间不便，容易造成碎片问题。

段页式寄存器

- 把程序按逻辑结构分段，每段再划分为固定大小的页，主存空间也划分为大小相等的页。
- 程序对主存的调入、调出仍以**页**为基本传送单位。每个程序对应一个段表，每段对应一个页表。
- 所以段长必须是页长的整数倍，段首址必须是某页的页首址。
- 虚拟地址：段号+段内页号+页内地址。
- CPU根据虚地址访存时，首先根据段号得到段表地址；然后从段表中取出该段的页表起始地址，与虚地址段内页号合成，得到页表地址；最后从页表中取出实页地址，与页内地址拼接形成主存实地址。

快表

- 页表、段表存放在主存中，收到虚拟地址后要先访问主存，查询页表、段表，进行虚实地址转换。放在主存中的页表称为慢表（$Page$）。
- 提高变换速度→用高速缓冲存储器$Cache$存放常用的页表项→快表（TLB）。

- TLB是$Page$的副本（快表是慢表的副本），而$Cache$为主存的副本。
- 慢表在主存中。快表是一个特殊的单独的$Cache$。
- 使用相联存储器，所以查找速度更快。也可以使用$SRAM$。（$DRAM$必须不断刷新不适合TLB和$Cache$）
- 快表地址计算与之前的地址计算类似，使用全相联或组相联的模式，若使用组相联也需要留出组号。

虚拟存储器与 Cache

1. $Cache$主要解决系统速度，而虚拟存储器却是为了解决主存容量。
2. $Cache$全由硬件实现，是硬件存储器，对所有程序员透明；而虚拟存储器由OS和硬件共同实现，是逻辑上的存储器，对系统程序员不透明，但对应用程序员透明。
3. 对于不命中性能影响，因为CPU的速度约为$Cache$的10倍，主存的速度为硬盘的100倍以上，因此虚拟存储器系统不命中时对系统性能影响更大。
4. CPU与$Cache$和主存都建立了直接访问的通路，而辅存与CPU没有直接通路。也就是说在$Cache$不命中时主存能和CPU直接通信，同时将数据调入$Cache$；而虚拟存储器系统不命中时，只能先由硬盘调入主存，而不能直接和CPU通信。

所以高速缓冲存储器连接主存实地址与$Cache$地址，虚拟存储连接主存实地址和虚拟存储逻辑地址，快表连接虚拟存储逻辑地址与快表地址。

指令系统

指令和数据是应用上的概念，同一个数据可以被解释为数据也可以被解释为指令。同一串二进制代码，可以是指令，也可以是数据，这决定于我们的程序设计。如1000100111011000被应用为数据时，它等于$89D8H$，H表示是十六进制。当被应用为指令时，它指的是$MOV\ AX,\ BX$。数据和指令的区分方式在第一章已经说明。

指令格式

指如何使用二进制代码表示指令。

操作码+寻址特征码（有多少种寻址方式）+地址码。

指令定义

- 指令（又称机器指令）：是指示计算机执行某种操作的命令，是计算机运行的最小功能单位。
- 指令系统是计算机软硬件的界面。指令系统指的是计算机执行的机器指令的集合；这里要注意的是微指令是微程序级命令，属于硬件范畴；伪指令是由若干的机器指令组成的指令序列，属于软件范畴；然而机器指令介于两者之

间，处于软硬件的交界面；而机器指令集又称为指令系统；所以回答是指令系统。

- 一台计算机的所有指令的集合构成该机的指令系统，也称为指令集。一台计算机只能执行自己指令系统中的指令，不能执行其他系统的指令。
- 一条指令通常要包括操作码字段（OP）和地址码字段（A）两部分。
- 操作码指出该指令要执行什么操作，地址码指出指令要操作的数据的地址。
- 指令地址由PC给出。
- 定长指令字结构：所有指令长度相等。执行速度块，控制简单。
- 变长指令字结构：指令长度随指令功能而异。

指令字长

- 固定：指令字长=存储字长。
- 可变：按字节的倍数变化。
- 指令字长取决于：
  1. 操作码长度。
  2. 操作数地址长度。
  3. 操作数地址个数。

地址码

- 一条指令的执行分为三步，如果是两个操作数则四次访存：
  1. 取指令。
  2. 取两个操作数。
  3. 放回结果。
- 访存是指访问内存，ACC在运算器中，访问ACC不是访问内存。如果数据在$Cache$中则不用访存。

指令类型：

- 四地址指令：操作码+操作数A_1地址+操作数A_2地址+结果地址A_3+下指令地址A_4。
  - $(A_1)OP(A_2) \rightarrow A_3$。
- 三地址指令：操作码+操作数A_1地址+操作数A_2地址+结果地址A_3。
  - 下一条指令的寻址靠程序计数器PC完成。
  - 一共访存四次。
  - $(A_1)OP(A_2) \rightarrow A_3$。
- 二地址指令：操作码+操作数A_1地址+操作数A_2地址。
  - 将结果存到操作数A_1地址或操作数A_2地址中。取指令，取数A_1和A_2，存放到A_1或A_2。一共访存四次。$(A_1)OP(A_2) \rightarrow A_1$。
  - 若使用累加器ACC暂存结果则只用访存三次，取指令，取数A_1和A_2，暂存到ACC。$(A_1)OP(A_2) \rightarrow ACC$。

- 一地址指令（单地址指令）：操作码+操作数地址：
  - 只需要一个操作数的指令操作，如加一、减一、取反、求补等。取出指令、取出数、操作、放回，只需要三次访存。$OP(A) \rightarrow A$。
  - 隐含约定的目的地址的双操作数指令，如目的地址为累加器ACC的地址。取指令和取数，只需要两次访存。$(ACC)OP(A) \rightarrow ACC$。
- 零地址指令：只有操作码：
  - 不需要操作数。如空操作、停机、关中断等。
  - 堆栈计算机，两个操作数隐含存放在栈顶和次栈顶，计算结果压回栈顶。
- 用硬件资源减少地址码字段的优势：
  - 扩大指令寻址范围。
  - 缩短指令字长。
  - 减少访存次数。

操作码

定长操作码

- 指令字的最高位部分分配固定的若干位表示操作码。
- 若有n位操作码，则有2^n条指令。
- 优点：能简化计算机硬件设计，提高指令译码和识别速度。
- 缺点：指令数量增加时会占用更多固定位，留给表示操作数地址的位数有限。
- 一般这种操作码用于指令字长较长的情况。

拓展操作码

- 拓展操作码是变成操作码实现的一种，让操作码长度随地址码减少而增加。
- 操作码的位数至少为当种指令总条数的二对数。如指令总长度为32位，二地址指令有27条，则二地址指令的操作码至少有$27 = 2^4 + 11 = 5$位，否则不能操作这么多地址。
- 优点：在指令字长有限的前提下能保持比较丰富的指令种类。
- 缺点：增加了指令译码和分析的难度，使控制器的设计复杂化。

操作码数	OP	A1	A2	A3
4	0000	A1	A2	A3
15 条	…	…	…	…
4	1110	A1	A2	A3
8	1111	0000	A2	A3
15 条	…	…	…	…
8	1111	1110	A2	A3
12	1111	1111	0000	A3

操作码数	OP	A1	A2	A3
15 条	…	…	…	…
12	1111	1111	1110	A3
16	1111	1111	1111	0000
16 条	…	…	…	…
16	1111	1111	1111	1111

- 假设指令字长为16位，前4位为基本操作码字段OP，另有3个4位长的地址字段A_1、A_2和A_3。4位基本操作码若全部用于三地址指令，则有16条。
- 但至少须将当前位的1111留作扩展操作码之用，即三地址指令为15条。所以目前就留下来了前4位操作数为1111的指令，这是只有四位操作码。
- 然后将留下来的指令的操作码位拓展为八位，将A_1的位也用作操作码，即将1111 0000 A_2 A_3到1111 1110 A_2 A_3作为二地址指令，二地址指也为15条。同样将1111 1111的指令作为下一个拓展操作码备用。
- 同样将留下来的指令的操作码拓展为十二位，将A_1A_2的位用作操作码，即将1111 1111 0000 A_3到1111 1111 1110 A_3作为一地址指令，一地址指令也为15条。同样将1111 1111 1111的指令作为下一个拓展操作码备用。
- 最后将操作码拓展为十六位，将$A_1A_2A_3$全部作为操作码，即将1111 1111 1111 0000到1111 1111 1111 1111作为零地址指令，零地址指令为16条。
- 在设计扩展操作码指令格式时，必须注意以下两点:
    1. 不允许短码是长码的前缀，即短操作码不能与长操个码的前面部分的代码相同，不然无法区分长码和短码。
    2. 各指令的操作码一定不能重复。
- 通常情况下，对使用频率较高的指令，分配较短的操作码，对使用频率较低的指令，分配较长的操作码，从而尽可能减少指令译码和分析的时间。

假设地址长度为n，上一层留出m种状态，则下一层可以拓展出$m \times 2^n$种状态。所以也可以形成任意种不同拓展方法。

操作类型

1. 数据传输：
    1. MOV：寄存器之间的传送。
    2. $LOAD$：把存储器的数据放到CPU寄存器中。
    3. $STORE$：把CPU寄存器的数据放到存储器中。
2. 算术逻辑：
    1. 算术：加ADD、减SUB、比较CMP、乘MUL、除DIV、自加一INC、自减一DEC、求补、浮点运算、十进制运算。
    2. 逻辑：与AND、或OR、非NOT、异或XOR、位操作、位测试、位清除、位求反。

3. 移位操作：
    1. 算术移位。
    2. 逻辑移位。
    3. 循环移位（带进位和不带进位）。
4. 转移操作：调用指令必须保存下一条指令的地址，当子程序结束时要返回主程序继续执行，而转移指令不用返回执行。
    1. 无条件转移：JMP。将地址码送入PC。
    2. 条件转移：$BRANCH$，如JZ：结果为0；JO：结果溢出；JC：结果进位。
    3. 调用$CALL$和返回RET。
    4. 陷阱$TRAP$和陷阱指令。（意外事故的中断）
5. 输入输出操作。

指令寻址方式

指令地址码不代表真实地址，这只是个形式地址A，只有结合寻址方式寻址才能得到真实地址EA。如$EA = (A)$指有效地址是形式地址A中保存的数值。

目的是：缩短指令字长、扩大寻址空间、提高编程灵活性。

寻址方式包括指令寻址和数据寻址。

操作数类型与存放方式

操作数类型

- 地址：无符号整数。
- 数字：定点数、浮点数、十进制数。
- 字符：$ASCII$。
- 逻辑数：逻辑运算。

存放方式

- 若一个操作数有多个内存地址对应，则存放地址为：
    - 大端方式：指令中给出的地址是操作数最高有效字节（MSB）所在的地址。字地址为高字节地址。如012345。
    - 小端方式：指令中给出的地址是操作数最低有效字节（LSB）所在的地址。字地址为低字节地址。如452301。
- 按字节地址寻址：给出个字节地址，可以取出长度为一个字节的数据。
- 按字地址寻址：给出个字地址，可以取出长度为一个字的数据。
- 按字节编址：每个字节存储单元都有一个地址编号。每个字中最小的字节地址就是字地址。

- 按字编址：每个字存储单元都有一个地址编号。但是按字编址就无法直接取出字节地址。所以一般只按字节编址。
- 三个字长：
  1. 机器字长：CPU一次能处理的二进制数据的位数。一般等于内部寄存器的位数。
  2. 指令字长：一个指令字中包含二进制代码的位数。若是单字长指令则指令字长等于机器字长，若是半字长指令等则不相等。
  3. 存储字长：一个存储单元存储二进制代码的长度。
- 从任意位置开始存储：
  - 优点：不浪费存储资源。
  - 缺点：除了访问一个字节之外，访阿其它任何类型的数据，都可能花费两个存储周期的时间。读写控制比较复杂。
- 从一个存储字的起始位置开始访问，其余位置置空或填充：
  - 优点：无论访问何种类型的数据，在一个周期内均可完成，读写控制简单。
  - 缺点：浪费了宝贵的存储资源。
- 边界对准方式，从地址的整数倍位置开始访问：
  - 由于不同的机器数据字长不同，每台机器处理的数据字长也不统一，为了便于硬件实现，通常要求多字节的数据在存储器的存放方式能满足“边界对准”的要求。
  - 数据存放的起始地址是数据长度（按照编址单位进行计算）的整数倍。
  - 本方案是前两个方案的折衷，在一个周期内可以完成存储访问，空间浪费也不太严重。

指令寻址

用于确定要执行的下一条指令的地址。始终由程序计数器PC给出。

- 顺序寻址：由PC自动加上某个数寻址下一个要操作的指令，注意：不是单纯加一。如果是按字编址就是直接加一，如果是按字节编址则是加一个指令字长。
- 跳跃寻址：由转移指令指出。如执行到操作码为JMP，则跳跃到保存的地址码所指向的指令地址进行执行。跳跃地址分为绝对地址（标记符直接得到）和相对地址（相对当前地址的偏移量）。跳跃的结构是当前指令修改PC值，所以下一条指令仍通过PC给出。可实现程序的条件或无条件转移。注意：程序跳跃后，按新的指令地址开始顺序执行。因此，程序计数器的内容也必须相应改变，以便及时跟踪新的指令地址。

数据寻址

- 用于确定本条指令的数据地址。
- 具体的数据寻址分为：

  - 主存寻址：数据都存储在主存中
    - 隐含寻址。
    - 立即寻址。
    - 直接寻址。
    - 间接寻址。
  - 寄存器寻址：数据都存储在寄存器中
    - 寄存器寻址。
    - 寄存器间接寻址。
  - 偏移寻址：都需要加一个偏移量
    - 相对寻址。
    - 基址寻址。
    - 变址寻址。
  - 堆栈寻址。
- 此时指令就需要由：操作码OP+寻址特征（寻址方式）+形式地址A组成。形式地址不再是实际的地址，而是指令字中的地址。
- 有效地址EA就是通过寻址特征和形式地址进行运算得到的。
- 无论是有多少个地址，都必须给出成对的寻址特征和形式地址。
- 主存地址不能为负。

由于访存慢于访问寄存器，所以寄存器访问快于直接访问。

如果采用变长指令码格式，由于要表示一定范围的立即数，立即数的指令通常需要较多的二进制位，取指时，可能需要不止一次的读内存来完成取指，因此采用变长指令码格式的时候，寄存器寻址方式的执行速度更快。但是如果采用定长指令码格式，那就是立即寻址更快了。

隐含寻址

- 不是明显地给出操作数的地址，而是在指令中隐含着操作数的地址。
- 如单地址指令，隐含约定另一个操作数的目的地址为累加器ACC的地址。
- 优点：有利于缩短指令字长。
- 缺点：需增加存储操作数或隐含地址的硬件。

立即寻址

- 形式地址指出的不是数据地址，而是数据本身，称为立即数，采用补码形式存储。
- 地址形式为：操作码OP + # +立即数。#即表示立即寻址特征。
- 立即寻址的指令执行：
  1. 取指令，访存1次。
  2. 执行指令，直接取指令操作数，访存0次。
  3. 暂不考虑如何存结果。

4. 共访存1次。

- 优点：指令执行阶段不访问主存，指令执行时间最短。
- 缺点：
  - 形式地址A的位数限制了立即数的范围。
  - 如A的位数为n，且立即数采用补码时，可表示的数据范围为$[-2^{n-1}, 2^{n-1}-1]$。

直接寻址

- 指令字中的形式地址A就是操作数的真实地址EA，即$EA = A$。
- 直接寻址的指令执行：
  1. 取指令，访存1次。
  2. 执行指令，取操作数，访存1次。
  3. 暂不考虑如何存结果。
  4. 共访存2次。
- 优点：简单，指令执行阶段仅访问一次主存，不需专门计算操作数的地址。
- 缺点：
  - 形式地址A的位数决定了该指令操作数的寻址范围。
  - 操作数的地址不易修改。

间接寻址

- 指令的地址字段给出的形式地址不是操作数的真正地址，而是操作数有效地址所在的存储单元的地址，也就是操作数地址的地址，即$EA = (A)$。
- 间接寻址的指令执行：
  1. 取指令，访存1次。
  2. 执行指令，取操作数地址，取操作数，访存2次。
  3. 暂不考虑如何存结果。
  4. 共访存3次。
- 若间址次数为n，则访问时间为$n + 2$。
- 优点：
  - 可扩大寻址范围（有效地址EA的位数大于形式地址A的位数）。
  - 便于编制程序（用间接寻址可以方便地完成子程序返回）。
- 缺点：指令在执行阶段要多次访存（一次间址需两次访存，多次寻址需根据存储字的最高位确定几次访存）。由于访问速度过慢，所以一般都不使用。

寄存器寻址

- 在指令字中直接给出操作数所在的寄存器编号，即$EA = R_i$，其操作数在由R_i所指的寄存器内。
- 类似于直接寻址，只是数据在主存中变为在寄存器中。寄存器编号就是形式地址。

- 寄存器寻址的指令执行：
  1. 取指令，访存1次。
  2. 执行指令，由于操作数在寄存器中，寄存器集成在CPU中，所以不需要访问主存，访存0次。
  3. 暂不考虑如何存结果。
  4. 共访存1次。
- 优点：
  - 指令在执行阶段不访问主存，只访问寄存器，指令字短且执行速度快。
  - 支持向量/矩阵运算。
- 缺点：
  - 寄存器价格昂贵。
  - 计算机中寄存器个数有限。

寄存器间接寻址

- 寄存器R_i中给出的不是一个操作数，而是操作数所在主存单元的地址，即$EA = (R_i)$。
- 寄存器间接寻址的指令执行：
  1. 取指令，访存1次。
  2. 执行指令，先访问寄存器获得地址，再根据地址在主存中访问，访存1次。
  3. 暂不考虑如何存结果。
  4. 共访存2次。
- 优点：与一般间接寻址相比速度更快。
- 缺点：指令的执行阶段需要访问主存（因为操作数在主存中）。

相对寻址

- 把程序计数器PC的内容加上指令格式中的形式地址A而形成操作数的有效地址，即$EA = (PC) + A$，其中A是相对于当前指令地址的位移量，可正可负，补码表示。
- 由于取指时PC会默认加上一个指令字长（$(PC) + 1 \rightarrow PC$），所以相对寻址所提供的相对地址是以下条指令在内存中首地址为基准位置的偏移量。
- 优点：
  - 操作数的地址不是固定的，它随着PC值的变化而变化，并且与指令地址之间总是相差一个固定值，因此便于程序浮动。
  - 相对寻址广泛应用于转移指令。
- 所以转移指令基本上使用相对寻址方式。无条件转移指令会在指令周期中对PC进行两次修改操作，一起是取指周期后自动加一，一个是执行周期后转移修改。有条件指令如果条件满足则修改两次，条件不满足则只用修改一次。

基址寻址

- 将CPU中基址寄存器（BR）的内容加上指令格式中的形式地址A，而形成操作数的有效地址，即$EA=(BR)+A$。
- 分为隐式基址寄存器（专用寄存器BR）和显式基址寄存器（需要指定某通用寄存器为基址寄存器）。
- 基址寄存器是面向操作系统的，其内容由操作系统或管理程序确定，用户不可改变。在程序执行过程中，基址寄存器的内容不变（作为基地址），形式地址可变（作为偏移量，使用补码表示）。
- 当采用通用寄存器作为基址寄存器时，可由用户决定哪个寄存器作为基址寄存器，但其内容仍由操作系统确定。
- 优点：
  - 可扩大寻址范围（基址寄存器的位数大于形式地址A的位数）。
  - 用户不必考虑自己的程序存于主存的哪一空间区域，故有利于多道程序设计，以及可用于编制浮动程序。

变址寻址

- 有效地址EA等于指令字中的形式地址A与变址寄存器IX的内容相加之和，即$EA=(IX)+A$。
- 其中IX为变址寄存器（专用），也可用通用寄存器作为变址寄存器。
- 与基址寄存器不同的是，变址寄存器是面向用户的，在程序执行过程中，变址寄存器的内容可由用户改变（作为偏移量），形式地址A不变（作为基地址）。
- 优点：
  - 可扩大寻址范围（变址寄存器的位数大于形式地址A的位数）。
  - 在数组处理过程中，可设定A为数组的首地址，不断改变变址寄存器IX的内容，便可很容易形成数组中任一数据的地址，特别适合编制循环程序。
- 变址寻址与基址寻址配合使用：$EA=A+(BR)+(IX)$。
- 变址寻址与间接寻址配合使用：
  - 先变址后间址，$EA=\big(A+(IX)\big)$。
  - 先间址后变址，$EA=(A)+(IX)$。

堆栈寻址

- 操作数存放在堆栈中，隐含使用堆栈指针（SP）作为操作数地址。
- 堆栈是存储器（或专用寄存器组）中一块特定的按“后进先出（$LIFO$）”原则管理的存储区，该存储区中被读/写单元的地址是用一个特定的寄存器给出的,该寄存器称为堆栈指针（SP）。
- 寄存器做堆栈就是硬堆栈，成本高，主存做堆栈就是软堆栈。

默认任何寻址方式都要加上一次取指访存一次：

寻址方式	有效地址	访存次数
隐含寻址	程序指定	0
立即寻址	A 即是操作数	0
直接寻址	EA=A	1
一次间接寻址	EA=(A)	2
寄存器寻址	EA=R	0
寄存器间接一次寻址	EA=(R)	1
相对寻址	EA=(PC)+A	1
基址寻址	EA=(BR)+A	1
变址寻址	EA=(IX)+A	1

- 立即寻址操作数获取便捷，通常用于给寄存器赋初值。
- 直接寻址相对于立即寻址，缩短了指令长度。
- 间接寻址扩大了寻址范围，便于编制程序，易于完成子程序返回。
- 寄存器寻址的指令字较短，指令执行速度较快。
- 寄存器间接寻址扩大了寻址范围。
- 基址寻址扩大了操作数寻址范围，适用于多道程序设计，常用于为程序或数据分配存储空间。
- 变址寻址主要用于处理数组问题，适合编制循环程序。
- 相对寻址用于控制程序的执行顺序、转移等。
- 基址寻址和变址寻址的区别：
  - 两种方式有效地址的形成都是寄存器内容＋偏移地址。
  - 在基址寻址中，程序员操作的是偏移地址，基址寄存器的内容由操作系统控制，在执行过程中是动态调整的。偏移量较短。
  - 在变址寻址中，程序员操作的是变址寄存器，偏移地址是固定不变的。偏移量较大。

指令集计算机

复杂指令集计算机

- 即$Complex\ Instruction\ Set\ Computer$，$CISC$。
- 一条指令完成一个复杂的基本功能。
- 代表：$x86$架构，主要用于笔记本和台式机。
- $80-20$规律：典型程序中80%的语句仅仅使用处理机20%的指令。

精简指令集计算机

- 即$Reduced\ Innstruction\ Set\ Computer$，$RISC$。
- 一条指令完成一个基本动作，多条指令组合完成一个复杂的基本功能。
- 代表：ARM架构，主要用于手机和平板。

对比项目	CISC	RISC
指令系统	复杂，庞大	简单，精简
指令数目	一般大于 200 条	一般小于 100 条
指令字长	不固定	定长
可访存指令	不加限制	只有 Load/Store 指令
各种指令执行时间	相差较大	绝大多数在一个周期内完成
各种指令使用频度	相差很大	大都比较常用
通用寄存器数量	较少	多
目标代码	难以用优化编译生成高效的目标代码程序	采用优化的编译程序，生成代码较为高效
控制方式	微程序控制	组合逻辑（硬布线）控制
利用率	低	高
指令流水线	可以通过一定方式实现	必须实现
兼容性	强	弱
速度	慢	快

中央处理器

即CPU，为最难的一章。

CPU 基本概念

CPU 功能

CPU 总功能

1. 指令控制。完成取指令、分析指令和执行指令的操作，即程序的顺序控制。
2. 操作控制。一条指令的功能往往是由若干操作信号的组合来实现的。CPU管理并产生由内存取出的每条指令的操作信号，把各种操作信号送往相应的部件，从而控制这些部件按指令的要求进行动作。
3. 时间控制。对各种操作加以时间上的控制。时间控制要为每条指令按时间顺序提供应有的控制信号。
4. 数据加工。对数据进行算术和逻辑运算。
5. 中断处理。对计算机运行过程中出现的异常情况和特殊请求进行处理。

运算器功能

- 对数据进行加工。
- 主要部件是ALU和ACC。

控制器功能

协调并控制计算机各部件执行程序的指令序列，基本功能包括取指令、分析指令、执行指令：

1. 取指令：
   - 自动形成指令地址。
   - 自动发出取指令的命令。
   - $(PC)+1 \rightarrow PC$。
2. 分析指令：
   - 操作码译码（操作码，分析本条指令要完成什么操作）。
   - 产生操作数的有效地址。
3. 执行指令：
   - 根据分析指令得到的“操作命令”和“操作数地址”。
   - 形成操作信号控制序列，控制运算器、存储器以及I/O设备完成相应的操作。
4. 中断处理：
   - 管理总线及输入输出。
   - 处理异常情况（如掉电）。
   - 特殊请求（如打印机请求打印一行字符）。

- 控制器部件向系统中的部件提供它们运行所需要的控制信号。
- 控制器部件从数据总线接收指令信息。
- 控制器部件从运算器部件接收指令转移地址。
- 控制器部件送出指令地址到地址总线。

CPU 结构

- ALU为首的组合逻辑。
- CU为首的时序逻辑。
- 寄存器。
- 中断系统。

内部寄存器：

- 用户可见可编程：
  - 通用寄存器组X。
  - 程序状态字寄存器PSW（标志寄存器）。
- 用户不可见不可编程：

- 存储器地址寄存器MAR。
- 存储器数据寄存器MDR。
- 指令寄存器IR。

运算器结构

1. 算术逻辑单元ALU：主要功能是进行算术/逻辑运算。
2. 通用寄存器组X：如AX、BX、CX、DX、SP等，用于存放操作数（包括源操作数、目的操作数及中间结果）和各种地址信息等。SP是堆栈指针，用于指示栈顶的地址。
3. 内部总线：
   - 专用数据通路方式：
     - 根据指令执行过程中的数据和地址的流动方向安排多条连接线路。
     - 如果直接用导线连接，相当于多个寄存器同时并且一直向ALU传输数据，解决方法：
       1. 使用多路选择器MUX根据控制信号选择一路输出解决方法。
       2. 使用三态门可以控制每一路是否输出，1允许，0不允许。
     - 性能较高，基本不存在数据冲突现象，但结构复杂，硬件量大，不易实现。
   - CPU内部单总线方式：
     - 将所有寄存器的输入端和输出端都连接到一条公共的通路上。
     - 结构简单，容易实现，但数据传输存在较多冲突的现象，性能较低。
     - 为了解决冲突用暂存寄存器解决。
4. 暂存寄存器R：用于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，否则会破坏其原有内容。
5. 累加寄存器ACC：它是一个通用寄存器，用于暂时存放ALU运算的结果信息，用于实现加法运算。
6. 程序状态字寄存器PSW：保留由算术逻辑运算指令或测试指令的结果而建立的各种状态信息，如溢出标志（OP）、符号标志（SF）、零标志（ZF）、进位标志（CF）等，PSW中的这些位参与并决定微操作的形成。
7. 移位器（移位寄存器）SR：对运算结果进行移位运算。拥有移位逻辑的寄存器。
8. 计数器（计数寄存器）CT：控制乘除运算的操作步数。拥有计数逻辑的寄存器。

控制器结构

1. 程序计数器PC：用于指出下一条指令在主存中的存放地址。CPU就是根据PC的内容去主存中取指令的。因程序中指令（通常）是顺序执行的，所以PC有自增功能。
2. 指令寄存器IR：用于保存当前正在执行的那条指令。
3. 指令译码器ID：仅对**操作码**字段进行译码，向控制器提供特定的操作信号。
4. 操作控制器OC：用来产生各种操作控制信号。
5. 微操作信号发生器：根据IR的内容（指令）、PSW的内容（状态信息）及时序信号，产生控制整个计算机系统所需的各种控制信号，其结构有组合逻辑型和存储逻辑型两种。
6. 时序系统（时序产生器）：用于产生各种时序信号,它们都是由统一时钟（$CLOCK$）分频得到。
7. 存储器地址寄存器MAR：用于存放所要访问的主存单元的地址。
8. 存储器数据寄存器MDR：用于存放向主存写入的信息或从主存中读出的信息。

指令执行

指令周期

- 指令周期：CPU从主存中每取出并执行一条指令所需的全部时间：
  - 取指周期：取指、分析。
  - 执行周期。
- 指令周期常常用若干机器周期来表示，机器周期又叫CPU周期。如取指令、取有效地址、执行指令这就是三个机器周期，是一个指令周期。
- 一个机器周期又包含若干时钟周期（也称为节拍、T周期或CPU时钟周期，它是CPU操作的最基本单位）。
- 每个指令周期内机器周期数可以不等，每个机器周期内的节拍数也可以不等。
- 指令周期流程：
  1. 进入取指周期。
  2. 判断是否有间址。
  3. 若有则进入间址周期，结束后进入执行周期。
  4. 若无则直接进入执行周期。
  5. 判断是否有中断。
  6. 若有则进入中断周期，结束后进入下一条指令的指令周期。
  7. 若无则直接进入下一条指令的指令周期。
- CPU访存的四种性质，可以设置四位二进制位作为标志触发器：
  - 取指令FE：取指周期。
  - 取地址IND：间址周期。
  - 存取操作数或结果EX：执行周期。

- 存程序断点INT：中断周期。

注意：中断周期中的进栈操作是将SP减1，这和传统意义上的进栈操作相反，原因是计算机的堆栈中都是向低地址增加，所以进栈操作是减1而不是加1。

四种周期：

- 指令周期：CPU从主存中每取出并执行一条指令所需的全部时间。指令周期可变。
- 时钟周期：通常称为节拍或T周期，它是CPU操作的最基本单位。时钟周期不变。
- CPU周期也称机器周期：一个机器周期包含若干时钟周期。是CPU进行一次操作的时间。由于CPU内部操作的速度较快，而CPU访问一次存储器的时间较长，因此机器周期通常由存取周期来确定，往往是通过一次总线事务访问一次主存或I/O的时间。机器周期可变。
- 存取周期：指存储器进行两次独立的存储器操作（连续两次读或写操作）所需的最小间隔时间。存取周期往往为固定值。

数据流

指令周期一共包含四个机器周期，但是指令不同可能只有部分周期，如零地址指令只有取指和执行周期。

取指周期

根据PC内容（无论是普通指令还是转移指令）从内存中取出指令代码并放入IR中。

注意：取指操作是控制器固化的自动执行的操作。

1. 当前指令地址送至存储器地址寄存器，记做：$(PC) \rightarrow MAR$。
2. MAR将地址码发送到地址总线。
3. 地址总线将地址发送给存储器，等待使用地址。
4. CU发出控制读信号给控制总线。
5. 控制总线将控制读的信号发送给存储器。启动存储器做读操作，记做：$1 \rightarrow R$。
6. 存储器根据地址总线传来的地址信息和控制总线传来的控制读信息来进行读操作，从中读出数据。并将地址所指的数据发送给数据总线。
7. 数据总线将数据送入MDR，记做：$M(MAR) \rightarrow MDR$。
8. 将MDR中数据（此时是指令内容）送入IR，记做：$(MDR) \rightarrow IR$。
9. CU发出控制信号，控制PC形成下一条指令地址，默认是加一，记做：$(PC) + 1 \rightarrow PC$。

间址周期

取操作数有效地址。

1. IR将指令的地址码送入MAR，记做：$Ad(IR) \rightarrow MAR$或$Ad(MDR) \rightarrow MAR$。
2. MAD将地址码发送到地址总线。
3. 地址总线将地址发送给存储器，等待使用地址。
4. CU发出控制读信号给控制总线。
5. 控制总线将控制读信息发送到存储器中。启动主存做读操作，记做：$1 \rightarrow R$。
6. 存储器根据地址总线传来的地址信息和控制总线传来的控制读信息来进行读操作，从中读出数据。并将地址所指的数据发送给数据总线。
7. 数据总线将数据送入MDR，记做：$M(MAR) \rightarrow MDR$。此时MDR保存的是操作数的地址而不是操作数本身。
8. 将有效地址送至指令的地址码字段，记做：$MDR \rightarrow Ad(IR)$。这一步可以没有。

执行周期

执行周期的任务是根据IR中的指令字的操作码和操作数通过ALU操作产生执行结果。不同指令的执行周期操作不同，因此没有统一的数据流向。

这里MDR访存取数得到操作数，此时MDR中的内容才是操作数。

中断周期

- 中断：暂停当前任务去完成其他任务。为了能够恢复当前任务，需要保存断点。
- 一般使用堆栈来保存断点，这里用SP表示栈顶地址，假设SP指向栈顶元素，进栈操作是先修改指针，后存入数据。

1. CU控制将SP减1，即将一个空元素进栈然后对其操作，并修改后的地址送入MAR，记做：$(SP) - 1 \rightarrow SP$，$(SP) \rightarrow MAR$。本质上是将断点存入某个存储单元，假设其地址为a，故可记做：$a \rightarrow MAR$。
2. MAR将地址码发送到地址总线。
3. 地址总线将地址发送给存储器，等待使用地址。
4. CU发出控制写信号给控制总线。
5. 控制总线将控制写信息发送到存储器中。启动主存做写操作，记做：$1 \rightarrow W$。
6. 将断点（PC内容）送入MDR，记做：$(PC) \rightarrow MDR$。
7. 将MDR内容传入数据总线。
8. 数据总结将内容发送存储器。存储器将内容写入其中，记为$(MDR) \rightarrow M$。
9. CU控制将中断服务程序的入口地更新（由向量地址形成部件产生）送入PC：向量地址$\rightarrow PC$。

指令执行方案

一个指令周期通常要包括多个时间段（执行步骤），每个步骤完成指令的一部分功能，几个依次执行的步骤完成这条指令的全部功能。

1. 单指令周期：
   - 对所有指令都选用相同的执行时间来完成。
   - 指令之间串行执行。
   - 指令周期取决于执行时间最长的指令的执行时间。
   - 缺点：对于那些本来可以在更短时间内完成的指令，要使用这个较长的周期来完成，会降低整个系统的运行速度。
   - 优点：实施简单，只用程序计数器PC就能实现。
2. 多指令周期：
   - 对不同类型的指令选用不同的执行步骤来完成。
   - 指令之间串行执行。
   - 可选用不同个数的时钟周期来完成不同指令的执行过程。
   - 缺点：需要更复杂的硬件设计。
   - 优点：系统运行速度更高，效率更高。
3. 流水线方案：
   - 在每一个时钟周期启动一条指令，尽量让多条指令同时运行，但各自处在不同的执行步骤中。

数据通路

- 数据通路就是数据在功能部件之间传送的路径。
- 由控制部件产生的控制信号建立数据通路。
- 数据通路的基本结构：
  - CPU内部单总线方式。
  - CPU内部多总线方式。
  - 专用数据通路方式。

CPU 内部单总线方式

- 内部总线是指同一部件，如CPU内部连接各寄存器及运算部件之间的总线。
- 系统总线是指同一台计算机系统的各部件，如CPU、内存、通道和各类I/O接口间互相连接的总线。
- 使用一根总线连接部件的输入和输出。
- 一个时钟内只允许一次操作。
- 实现简单。
- 容易冲突，且效率较低。

对于ALU这种是一个组合逻辑电路的部件，其运算过程中必须保持两个输入端的内容不变。如果使用内部单总线方式，因此为了得到两个不同的操作数，ALU的一个输入端与总线相连，另一个输入端需通过一个寄存器与总线相连，第一个传入值的输入端需要用寄存器保存数据等待第二个数的输入，避免数据发生变化。此外，ALU的唯一的输出端也不能直接与内部总线相连，否则其输出又会通过总线反馈到输入端，影响运算结果，因此输出端需通过一个暂存器（用来暂存结果的寄存器）或三态门（控制与总线的打开与关闭）与总线相连。

1. 寄存器之间数据传送，比如把PC内容送至MAR，实现传送操作的流程及控制信号为：
    1. $(PC) \rightarrow Bus$：$PCout$有效，PC内容送总线。
    2. $Bus \rightarrow MAR$：$MARin$有效，总线内容送MAR。
2. 主存与CPU之间的数据传送，比如CPU从主存读取指令，实现传送操作的流程及控制信号为：
    1. $(PC) \rightarrow Bus \rightarrow MAR$：$PCout$和$IMARin$有效，现行指令地址$\rightarrow MAR$。
    2. $1 \rightarrow R$：CU通过控制总线发出读命令。
    3. $MEM(MAR) \rightarrow MDR$：$MDRin$有效，根据MAR地址取值送到MDR。
    4. $MDR \rightarrow Bus \rightarrow IR$：$MDRout$和$IRin$有效，现行指令$\rightarrow IR$。
3. 执行算术或逻辑运算，比如一条加法指令，微操作序列及控制信号为：
    1. $Ad(IR) \rightarrow Bus \rightarrow MAR$：$MDRout$和$IMARin$有效。
    2. $1 \rightarrow R$：CU发读命令。
    3. $MEM(MAR)$ →数据线$\rightarrow MDR$：$MDRin$有效。
    4. $MDR \rightarrow Bus \rightarrow Y$：$MDRout$和$Yin$有效，操作数$\rightarrow Y$。
    5. $(ACC) + (Y) \rightarrow Z$：$ACCout$和$ALUin$有效，$CU$向$ALU$发送加命令。
    6. $Z \rightarrow ACC$：$Zout$和$ACCin$有效，结果$\rightarrow ACC$。

CPU 内部多总线方式

- 使用多根总线连接部件的输入和输出。
- 效率相对于单总线而言得到了提升。

专用数据通路方式

- 对于各种部件使用专用的通路进行连接。专用通路就是将总线分散到各个地方。
- 连接多，实现困难。

控制器

CU的设计包括硬布线和微程序两种，微程序比较重要，硬布线了解即可。

控制器输入输出

输入：

1. 指令寄存器：$OP(IR) \rightarrow CU$，控制信号的产生与操作码有关。
2. 时钟：一个时钟脉冲发一个操作命令或一组需要同时执行的操作命令。
3. 标志：如条件转移指令，根据相应的标志位决定下一步操作。
4. 外来信号：如中断请求信号$INTR$、总线请求信号HRQ。

CU的输入信号来源如下：

1. 经指令译码器译码产生的指令信息。
2. 时序系统产生的机器周期信号和节拍信号。
3. 来自执行单元的反馈信息即标志。

前两者是主要因素。

输出：

1. CPU内部的控制信号：寄存器之间的数据传输、PC的修改、控制ALU进行相应的运算。
2. 到控制总线的控制信号：
   - 到存储器:访存控制信号 MREQ、读命令 RD、写命令 WR。
   - 到 I/O 设备:访问 I/O 设备的控制信号 IO。
   - 中断响应信号 INTA。
   - 总线响应信号 HLDA。

硬布线

微操作控制信号由组合逻辑电路根据当前的指令码、状态和时序，即时产生。

设计步骤

1. 分析每个阶段的微操作序列。
2. 选择CPU的控制方式：产生不同微操作命令序列所用的时序控制方式：
   1. 同步控制方式：
      - 整个系统所有的控制信号均来自一个统一的时钟信号。
      - 通常以最长的微操作序列和最烦琐的微操作作为标准，采取完全统一的、具有相同时间间隔和相同数目的节拍作为机器周期来运行不同的指令。
      - 同步控制方式的优点是控制电路简单，缺点是运行速度慢。
   2. 异步控制方式：
      - 异步控制方式不存在基准时标信号。
      - 各部件按自身固有的速度工作，通过应答方式进行联络。
      - 异步控制方式的优点是运行速度快，缺点是控制电路比较复杂。
   3. 联合控制方式：

- 对各种不同的指令的微操作实行大部分采用同步控制、小部分采用异步控制的办法。

3. 安排微操作时序：
   - 原则：
     1. 微操作的先后顺序不得随意更改被控对象不同的微操作。
     2. 尽量安排在一个节拍内完成占用时间较短的微操作。
     3. 尽量安排在一个节拍内完成并允许有先后顺序。
4. 电路设计。
   1. 列出操作时间表。
   2. 写出微操作命令的最简表达式。
   3. 画出逻辑图。

指令类别

1. 非访存指令：
   1. CLA（$clear$）：ACC清零。
   2. COM（$complement$）：ACC取反。
   3. SHR（$shift$）：算术右移。
   4. CSL（$cyclic\ shift$）：循环左移。
   5. STP（$stop$）：停机。
2. 访存指令：
   1. ADD：加法指令，隐含ACC。
   2. STA：存数指令，隐含ACC。
   3. LDA：取数指令，隐含ACC。
3. 转移指令：
   1. JMP（$jump$）：无条件转移。
   2. BAN（$branch\ ACC\ Negative$）：条件转移。

基本微操作时序

首先对于取指周期，基本上的流程如下：

1. $PC \rightarrow MAR$：将PC地址交给MAR。
2. $1 \rightarrow R$：主存发出读命令。只需要存储器空闲就能发出。
3. $M(MAR) \rightarrow MDR$：将地址指向指令交给MDR。要MAR准备好，在1的后面。
4. $MDR \rightarrow IR$：将指令交给IR。把MDR有指令，在3后面。
5. $OP(IR) \rightarrow ID$：编译指令。IR中要有指令，在4后面。
6. $(PC) + 1 \rightarrow PC$：PC自加1。因为操作完就可以更新，所以在1后面就可以了。

所以没有依赖的可以尽量往前放，顺序可以变为123645。

因为12没有相互依赖，所以可以都安排在$T0$时间；36在1的后面，且36之间用到的设备不一样从而不会冲突，所以都安排在$T1$；而4和5时间都较短，所以可以都安排在$T2$。

然后是间址周期：

1. $Ad(IR) \rightarrow MAR$：获取IR指令中的地址交给MAR。
2. $1 \rightarrow R$：主存发出读命令。
3. $M(MAR) \rightarrow MDR$：取出对应的值放到MDR中。
4. $MDR \rightarrow Ad(IR)$：将MDR的地址值放入IR的指令中。

与取指周期一致，2可以跟1一起，而34都依赖于1，所以12是$T0$，3是$T1$，4是$T2$。

执行周期省略，最后是中断周期，假设中断时要保存的地址为a：

1. $a \rightarrow MAR$：将a保存到MAR中。
2. $1 \rightarrow W$：主存发出写命令。存储器空闲就可以。
3. $0 \rightarrow EINT$：硬件关中断。安排在第一个周期就可以。
4. $(PC) \rightarrow MDR$：将当前程序计数器保存的位置暂存到MDR，等待后期恢复。内部数据通路空闲就可以。
5. $MDR \rightarrow M(MAR)$：将MDR的数据保存到a这个地址。在4之后。
6. 向量地址$\rightarrow PC$：将PC送到中断服务地址。只用PC改好就可以，在4之后。

1、2、3都在$T0$，而456依次为$T1$、$T2$、$T3$。

这些操作由中断隐指令完成。中断隐指令不是一条指令，而是指一条指令的中断周期由硬件完成的一系列操作

微程序

事先把微操作控制信号存储在一个专门的存储器（控制存储器）中，将每一条机器指令编写成一个微程序，这些微程序可以存到一个控制存储器中，用寻址用户程序机器指令的办法来寻址每个微程序中的微指令。

- 在微程序控制器中，控制部件向执行部件发出的控制信号称为微命令。
- 微命令执行的操作称为微操作。
- 微指令则是若干微命令的集合。
- 若干微指令的有序集合称为微程序。

微程序构成

- 完成一条机器指令分为多个微操作命令，微操作命令即微命令，是微操作的控制信号。微操作是微命令的执行过程。
- 微命令分为：
  - 相容性微命令：可以同时产生、共同完成某些微操作的微命令。

  - 互斥性微命令：在机器中不允许同时出现的微命令。
- 对于相容性的微命令可以合并为一条微指令，而互斥性微命令只能单独为一条微指令。
- 微指令是若干微命令的集合。微周期通常指从控制存储器中读取一条微指令并执行相应的微操作所需的时间。
- 假设微指令的一个二进制位对应一个微操作命令，所有的微命令都合并为微指令，微指令再合并为一个微程序保存到ROM中，从而每一条机器指令对应一条微程序。
- 微指令基本格式：
  - 操作控制：微操作码：产生控制信号。
  - 顺序控制：微地址码：产生下一条指令地址。
- 每条指令取指周期的操作是相同的，所以将取指令操作的微命令统一编成一个微程序，而每条机器指令所对应的具体操作再单独编写一个对应的微程序。
- 执行公用的取指微程序从主存中取出机器指令后，由机器指令的操作码字段指出各个微程序的入口地址（初始微地址）。

微程序控制器结构

- 控制存储器CM：用于存放各指令对应的微程序，控制存储器可用只读存储器ROM构成。
- 微地址寄存器$CMAR$：接收微地址形成部件送来的微地址，为在CM中读取微指令作准备。
- 地址译码器：将地址码转换微存储单元控制信号。
- 微指令寄存器$CMDR$：用于存放从CM中取出的微指令，它的位数同微指令字长相等。
- 微地址形成部件：产生初始微地址和后继微地址，以保证微指令的连续执行。
- 顺序逻辑单元：为了保证指令联系执行，控制形成下一条微指令，拥有一个标志位标识顺序执行还是跳转和一个时钟信号CLK。
- 控制流程：
  1. 机器指令操作码OP送到微地址形成部件形成微地址。
  2. 将初始微地址送到顺序逻辑单元，判断顺序执行还是跳转。
  3. 送到$CMAR$保存微地址。
  4. 将微地址从$CMAR$送到地址译码器中译码。
  5. 形成控制单元后送到CM中取出微指令。
  6. 将微指令送到$CMDR$中。
  7. 根据微指令中的下一条指令地址送到顺序逻辑单元，判断顺序执行还是跳转。
  8. 返回结果后$CMDR$送到CPU内部和系统总线，产生控制信号。
- 微程序个数：
  - 默认一条机器指令对应一个微程序。

- 取指周期微程序默认是公共的（即单独拿出来公用），故如果某指令系统中有n条机器指令，则CM中微程序的个数至少是$n+1$个（加一个取指的公共微程序）。
- 间址周期微程序和中断周期微程序不一定是公共的。若公共的，如果这台计算机指令系统中有n条机滤指令，则CM中微程序的个数是$n+3$个。若题目中没有提到间址，就不考虑间址，所以是$n+2$个。

微指令格式

1. 水平型微指令：
   - 一次能定义并执行多个并行操作。
   - 基本格式是：操作控制+判别测试+后继地址。
   - 优点：微程序短，执行速度快。
   - 缺点：微指令长，编写微程序较麻烦。
2. 垂直型微指令：
   - 类似机器指令操作码的方式，由微操作码字段规定微指令的功能。一条指令对应一个操作。
   - 基本格式：微操作码+目的地址+源地址。
   - 优点：微指令短、简单、规整，便于编写微程序。
   - 缺点：微程序长，执行速度慢，工作效率低。
3. 混合型微指令：
   - 在垂直型的基础上增加一些不太复杂的并行操作。
   - 微指令较短，仍便于编写。
   - 微程序也不长，执行速度加快。

水平型微指令的求法。

- 操作控制：
  - 若是直接编码方式，则微命令数就是操作控制字段位数。
  - 若是字段直接编码方式，则会分n个组，每组n_i种状态，则总位数就是$\sum_{i=1}^{n}\log_2(n_i+1)$（全0空出）。如2,3,4，则一共需要$2+2+3=7$位。
- 判别测试：
  - 直接编码方式，有几个外部条件，就取几位。
  - 如果是字段直接编码，有N个外部条件就有n位，$2^n \geq N+1$，加1是因为还有无条件转移的情况。
- 后继地址：
  - 求出前面两项后，直接根据微指令字长减去。
  - 根据CM的容量的前一项确认，$2^N \times M$容量就是N位。

微指令编码

微指令的编码方式又称为微指令的控制方式，它是指如何对微指令的控制字段进行编码，以形成控制信号。编码的目标是在保证速度的情况下，尽量缩短微指令字长。

1. 直接编码（直接控制）方式：
   - 在微指令的操作控制字段中，每一位二进制位代表一个微操作命令。
   - 某位为“1”表示该控制信号有效，为1的指令同时执行；若“0”则代表此时该位指向的命令不运行。
   - 优点：简单、直观，执行速度快，操作并行性好。
   - 缺点：微指令字长过长，n个微命令就要求微指令的操作字段有n位，造成控存容量极大。
2. 字段直接编码方式：
   - 将微指令的控制字段分成若干“段”，每段经译码后发出控制信号。
   - 微命令字段分段的原则：
     1. 因为每组译码后可以通过每位10的不同来让每组中的操作互斥，而不同组则很难完成互斥，所以互斥性微命令分在同一段内；因为命令在不同的组可以同时执行，所以相容性微命令分在不同段内。
     2. 每个小段中包含的信息位不能太多，否则将增加译码线路的复杂性和译码时间。
     3. 一般每个小段还要留出一个状态，表示本字段不发出任何微命令。因此，当某字段的长度为3位时，最多只能表示7个互斥的微命令，通常用000表示不操作。
   - 优点：可以缩短微指令字长。
   - 缺点：要通过译码电路后再发出微命令，因此比直接编码方式慢。
3. 字段间接编码方式：
   - 一个字段的某些微命令需由另一个字段中的某些微命令来解释，由于不是靠字段直接译码发出的微命令，故称为字段间接编码，又称隐式编码。
   - 优点：可进一步缩短微指令字长。
   - 缺点：运行速度更慢；削弱了微指令的并行控制能力，故通常作为字段直接编码方式的辅助手段。

微指令地址

1. 微指令的下地址字段指出：微指令格式中设置一个下地址字段，由微指令的下地址字段直接指出后继微指令的地址，这种方式又称为断定方式。
2. 根据机器指令的操作码形成：当机器指令取至指令寄存器后，微指令的地址由操作码经微地址形成部件形成。
3. 由硬件产生微程序入口地址：

- 第一条微指令地址：专门硬件产生。
- 中断周期：硬件产生中断周期微程序首地址。

4. 增量计数器法：$(CMAR)+1\rightarrow CMAR$，适用于指令连续存放的情况。
5. 分支转移：指令分为三个字段：
    1. 操作控制字段：指明操作控制类型。
    2. 转移方式：指明判别条件。
    3. 转移地址：指明转移成功后的去向。
6. 通过测试网络：一个测试网络产生。

微程序控制单元设计

设计步骤：

1. 分析每个阶段的微操作序列。
2. 写出对应机器指令的微操作命令及节拍安排：
    1. 写出每个周期所需要的微操作（参照硬布线）。
    2. 补充微程序控制器特有的微操作：
        - 取指周期：$Ad(CMDR)\rightarrow CMAR$；$OP(IR)\rightarrow CMAR$。
        - 执行周期：$Ad(CMDR)\rightarrow CMAR$。
3. 确定微指令格式：
    - 根据微操作个数决定采用何种编码方式，以确定微指令的操作控制字段的位数。
    - 由微指令数确定微指令的顺序控制字段的位数。
    - 最后按操作控制字段位数和顺序控制字段位数就可确定微指令字长。
4. 编写微指令码点：根据操作控制字段每一位代表的微操作命令，编写每一条微指令的码点。

节拍安排	取指周期-硬布线控制器的节拍安排	取指周期-微程序控制器的节拍安排
T0	PC→MAR	PC→MAR
T0	1→R	1→R
T1	M(MAR)→MDR	M(MAR)→MDR
T1	(PC)+1→PC	(PC)+1→PC
T2	MDR→IR	MDR→IR
T2	OP(IR)→ID	OP(IR)→微地址形成部件

根据读出转入微指令的过程，变为：

节拍安排	取指周期-硬布线控制器的节拍安排	取指周期-微程序控制器的节拍安排
T0	PC→MAR	PC→MAR

节拍安排	取指周期-硬布线控制器的节拍安排	取指周期-微程序控制器的节拍安排
T0	1→R	1→R
T1	M(MAR)→MDR	Ad(CMDR)→CMAR
T1	(PC)+1→PC	
T2	MDR→IR	M(MAR)→MDR
T2	OP(IR)→ID	(PC)+1→PC
T3		Ad(CMDR)→CMAR
T4		MDR→IR
T4		OP(IR)→微地址形成部件

硬布线与微程序

对比项目	微程序控制器	硬布线控制器
工作原理	微操作控制信号以微程序的形式存放在控制存储器中，执行指令时读出即可	微操作控制信号由组合逻辑电路根据当前的指令码、状态和时序，即时产生
执行速度	慢（从 CM 中读取微指令）	快
规整性	较规整	烦琐、不规整
应用场合	CISC CPU	RISC CPU
易扩充性	易扩充修改	困难

微程序设计分类

- 静态微程序设计：无需改变，采用ROM。
- 动态微程序设计动态通过改变微指令和微程序改变机器指令，有利于仿真，采用$EPROM$。
- 毫微程序设计：微程序设计用微程序解种机器指令，毫微程序设计用毫微程序解释微程序。

指令流水线

流水线基本概念

指令流水线定义

一条指令大致分为取指、分析、执行三个阶段，设时间都为t，一共有n条指令：

1. 顺序执行：
   - 顺序执行所有指令。
   - 总耗时$3nt$。
   - 传统冯·诺依曼机采用顺序执行方式，又称串行执行方式。
   - 优点：控制简单，硬件代价小。
   - 缺点：执行指令的速度较慢，在任何时刻，处理机中只有一个指令在指令，各功能部件利用率都很低。
2. 一次重叠执行方式：
   - 在执行第k条指令的同时取第$k+1$条指令。
   - 总耗时$(1+2n)t$。
   - 优点：程序的执行时间缩短了$1/3$，各功能部件的利用率明患提高。
   - 缺点：需要付出硬件上较大开销的代价控制过程也比顺件执行复杂了。
3. 二次重叠执行方式：
   - 分析第k条指令的同时取$k+1$条指令，执行k条指令时分析$k+1$条指令与取$k+2$条指令。
   - 总耗时$(2+n)t$。
   - 与顺序执行方式相比，指令的执行时间缩短近$2/3$。
   - 这是一种理想的指令执行方式，在正常情况下,处理机中同时有3条指令在执行。
4. 指令流水方式：多次重叠执行方式就是流水线方式。

流水线表示

1. 指令执行过程图：
   - 横坐标为时间，纵坐标为指令序列。
   - 每一行就是一条指令序列。
   - 主要用于分析指令执行过程以及影响流水线的因素。
2. 时空图：
   - 横坐标为时间，纵坐标为空间，即不同的阶段所对应的不同硬件资源，如：取指、译码、执行、存结果。
   - 每一斜着的一列就是一条指令序列。
   - 主要用于分析流水线的性能。

指令集要求

1. 指令长度一致。
2. 指令格式规整，保证源寄存器位置相同。
3. 让$Load/Store$指令唯一访问存储器。
4. 数据和指令在存储器种对齐存放。

流水线影响因素

- 理想情况：各阶段花费时间相同，每个阶段结束后能立即进入下一阶段。
- 机器周期一般分为五段：取指IF、译码ID、执行EX、访存M、写回WB。
- 假如各部件实际耗时：$100ns$、$80ns$、$70ns$、$50ns$、$50ns$，为方便流水线的设计，将每个阶段的耗时取成一样，以最长耗时为准。即此处应将机器周期设置为$100ns$。
- 流水线每一个功能段部件后面都要有一个缓冲寄存器，或称为锁存器，其作用是保存本流水段的执行结果，提供给下一流水段使用。

结构相关（资源冲突）

- 由于多条指令在同一时刻争用同一资源而形成的冲突称为结构相关。
- 解决办法：
    1. 后一相关指令暂停一周期。
    2. 单独设置数据存储器和指令寄存器。

数据相关（数据冲突）

- 数据相关指在一个程序中，存在必须等前一条指令执行完才能执行后一条指令的情况，则这两条指令即为数据相关。
- 数据的基本操作：读、写。
- 冲突的基本类型：
    - 写后读RAW：没有写完就读取，按序发射，按序完成只会出现写后读的错误。
    - 读后写WAR：乱序发射，优化手段导致指令顺序不符合编写程序时预想的逻辑顺序。
    - 写后写WAW：存在多个功能部件时，后一条指令先于前一条指令完成。
- 解决办法:
    1. 把遇到数据相关的指令及其后续指令都暂停一至几个时钟周期，直到数据相关问题消失后再继续执行。可分为硬件阻塞（$stall$）和软件插入“NOP”（空指令）两种方法。
    2. 数据旁路技术：计算得到结果后不写回，直接当作输入使用。
    3. 编译优化：通过编译器调整指令顺序来解决数据相关。

控制相关（控制冲突）

- 当流水线遇到转移指令和其他改变PC值的指令而造成断流时，会引起控制相关。
- 主要指转移指令。
- 解决办法：
    1. 尽早判别转移是否发生，尽早生成转移目标地址。
    2. 预取转移成功和不成功两个控制流方向上的目标指令。

3. 加快和提前形成条件码。
4. 提高转移方向的猜准率。

流水线分类

- 根据流水线使用的级别的不同：
  - 部件功能级流水就是将复杂的算术逻辑运算组成流水线工作方式。例如，可将浮点加法操作分成求阶差、对阶、尾数相加以及结果规格化等 4 个子过程。
  - 处理机级流水是把一条指令解释过程分成多个子过程，如前面提到的取指、译码、执行、访存及写回5个子过程。
  - 处理机间流水是一种宏流水，其中每一个处理机完成某一专门任务，各个处理机所得到的结果需存放在与下一个处理机所共享的存储器中。
- 按流水线可以完成的功能：
  - 单功能流水线指只能实现一种固定的专门功能的流水线。
  - 多功能流水线指通过各段间的不同连接方式可以同时或不同时地实现多种功能的流水线。
- 按同一时间内各段之间的连接方式：
  - 静态流水线指在同一时间内，流水线的各段只能按同一种功能的连接方式工作。
  - 动态流水线指在同一时间内，当某些段正在实现某种运算时，另一些段却正在进行另一种运算。这样对提高流水线的效率很有好处，但会使流水线控制变得很复杂。
- 按流水线的各个功能段之间是否有反馈信息：
  - 线性流水线中，从输入到输出，每个功能段只允许经过一次，不存在反馈回路。
  - 非线性流水线存在反馈回路，从输入到输出过程中，某些功能段将数次通过流水线，这种流水线适合进行线性递归的运算。

流水线性能

1. 吞吐率：
   - 指在单位时间内流水线所完成的任务数量，或是输出结果的数量。
   - 设任务数为n，处理完成n个任务所用的时间为T_k，则计算流水线吞吐率TP的最基本的公式为$TP = \frac{n}{T_k}$。
   - 令每个指令分为k个阶段，每个阶段所需时间为Δt一般等于一个时钟周期，则理想状态下$T_k = (k + n - 1)\Delta t$，所以$TP = \frac{n}{(k+n-1)\Delta t}$。
   - 第一个指令执行阶段称为装入时间，最后一个指令执行的阶段称为排空时间。
2. 加速比：

- 完成同样一批任务，不使用流水线所用的时间与使用流水线所用的时间之比。
- 设T_0表示不使用流水线时的执行时间，即顺序执行所用的时间；T_k表示使用流水线时的执行时间，则计算流水线加速比S的基本公式为$S=\frac{T_0}{T_k}$。
- 单独完成一个任务需要$k\Delta t$，则顺序完成时间$T_0=nk\Delta t$。又$T_k=(k+n-1)\Delta t$，所以$S=\frac{nk\Delta t}{(k+n-1)\Delta t}=\frac{kn}{k+n-1}$。

3. 效率：
   - 流水线的设备利用率称为流水线的效率。
   - 在时空图上，流水线的效率定义为完成n个任务占用的时空区有效面积与n个任务所用的时间与 k 个流水段所围成的时空区总面积之比。（时空图中用到的类平行四边形面积除以围成矩形的面积）。
   - 流水线效率E的基本公式为$E=\frac{T_0}{kT_k}=\frac{n}{k+n-1}$。

流水线多发

1. 超标量技术：
   - 每个时钟周期内可并发多条独立指令。
   - 要配置多个功能部件。
   - 不能调整指令的执行顺序。
   - 可以结合动态调度技术优化。
2. 超流水技术：
   - 在一个时钟周期内再分段。
   - 在一个时钟周期内单个功能部件使用多次。
   - 不能调整指令的执行顺序，靠编译来优化。
3. 超长指令字：
   - 由编译程序挖掘出指令间潜在的并行性，将多条能并行操作的指令组合成一条具有多个操作码字段的超长指令字。
   - 需要更大的$Cache$。

总线

总线概述

总线基本定义

- 总线是一组能为多个部件分时共享的公共信息传送线路。是总线复用方式。
- 共享是指总线上可以挂接多个部件，各个部件之间互相交换的信息都可以通过这组线路分时共享。

- 分时是指同一时刻只允许有一个部件向总线发送信息，如果系统中有多个部件，则它们只能分时地向总线发送信息。
- 早期计算机外部设备少时大多采用分散连接方式，不易实现随时增减外部设备。为了更好地解决I/O设备和主机之间连接的灵活性问题，计算机的结构从分散连接发展为总线连接。
- 总线特性：
    1. 机械特性：尺寸、形状、管脚数、排列顺序。
    2. 电气特性：传输方向和有效的电平范围。
    3. 功能特性：每根传输线的功能（地址、数据、控制）。
    4. 时间特性：信号的时序关系

总线分类

- 数据传输格式：
    - 串行总线：
        - 优点：只需要一条传输线,成本低廉，广泛应用于长距离传输；应用于计算机内部时，可以节省布线空间。
        - 缺点：在数据发送和接收的时候要进行拆卸和装配，要考虑串行并行转换的问题。
    - 并行总线：
        - 优点：总线的逻辑时序比较简单，电路实现起来比较容易。
        - 缺点：信号线数量多，占用更多的布线空间;远距离传输成本高昂；由于工作频率较高时，并行的信号线之间会产生严重电磁影响导致无法使用，对每条线等长的要求也越来越高，所以无法持续提升工作频率。
- 总线功能（连接部件）：
    - 片内总线：片内总线是芯片内部的总线。 它是CPU芯片内部寄存器与寄存器之间、寄存器与ALU之间的公共连接线。
    - 系统总线：系统总线是计算机系统内各功能部件（CPU、主存、I/O接口）之间相互连接的总线。按系统总线传输信息内容的不同，又可分为：
        - 数据总线DB：传输各功能部件之间的数据信息，包括指令和操作数；位数（根数）与机器字长、存储字长有关；双向。
        - 地址总线AB：传输地址信息，包括主存单元或I/O端口的地址；位数（根数与主存地址大小及设备数量有关）；单向。
        - 控制总线CB：传输控制信息；一根控制线传输一个信号；有出：CPU送出的控制命令；有入：主存（或外设）返回CPU的反馈信号。

  - 通信总线：通信总线是用于计算机系统之间或计算机系统与其他系统（如远程通信设备、测试设备）之间信息传送的总线，通信总线也称为外部总线。
- 时序控制方式：
  - 同步总线。
  - 异步总线。

总线结构

- 单总线结构：
  - CPU、主存、I/O设备（通过I/O接口）都连接在一组总线上，允许I/O设备之间、I/O设备和CPU之间或I/O设备与主存之间直接交换信息。
  - 单总线并不是指只有一根信号线，系统总线按传送信息的不同可以细分为地址总线、数据总线和控制总线。
  - 优点：结构简单，成本低，易于接入新的设备。
  - 缺点：带宽低、负载重，多个部件只能争用唯一的总线，且不支持并发传送操作。
- 双总线结构：
  - 双总线结构有两条总线，一条是主存总线，用于CPU、主存和通道之间进行数据传送；另一条是I/O总线（因为I/O速度较慢），用于多个外部设备与通道之间进行数据传送。
  - 通道是具有特殊功能的处理器，能对I/O设备进行统一管理。通道程序放在主存中。
  - 支持突发（猝发）传送：送出一个地址，收到多个地址连续的数据。
  - 优点：将较低速的I/O设备从单总线上分离出来，实现存储器总线和I/O总线分离。
  - 缺点：需要增加通道等硬件设备。
- 三总线结构：
  - 三总线结构是在计算机系统各部件之间采用三条各自独立的总线来构成信息通路，分别为：
    - 主存总线：CPU和主存之间的连接。
    - I/O总线：CPU和I/O接口之间连接。
    - DMA总线：$Direct\ Memory\ Access$，直接内存访问，连接外设与主存。能直接连接主存的就是高速外设，如磁盘机，而无法直接连接主存的就是低速外设，如打印机和显示器。
  - 优点：提高了I/O设备的性能，使其更快地响应命令，提高系统吞吐量。
  - 缺点：系统工作效率较低。
- 四总线结构：

- 四总线结构一共四条总线：
  - CPU总线（局部总线）：CPU与$Cache$。
  - 系统总线：CPU和主存。
  - 高速总线：连接$SCSI$、图形、多媒体、局域网。
  - 扩充总线（扩展总线型）：连接传真机、扩展总线接口、调制解调器、串行接口。
- 桥接器：用于连接不同的总线，具有数据缓冲、转换和控制功能。
- 越靠近CPU总线速度越快，CPU总线>系统总线>高速总线>扩充总线。
- 每级总线的设计遵循总线标准。

性能指标

传输周期

也称为总线周期。一次总线操作所需的时间（包括申请阶段、寻址阶段、传输阶段和结束阶段），通常由若干个总线时钟周期构成。

时钟周期

即机器的时钟周期。计算机有一个统一的时钟，以控制整个计算机的各个部件，总线也要受此时钟的控制。

工作频率

总线上各种操作的频率，为总线周期的倒数。

若总线周期=N个时钟周期，则总线的工作频率=时钟频率$\div N$。

时钟频率

即机器的时钟频率，为时钟周期的倒数。若时钟周期为T，则时钟频率为$1/T$。

实际上指一秒内有多少个时钟周期。

总线宽度

又称为总线位宽，它是总线上同时能够传输的数据位数，通常是指数据总线的根数，如32根称为32位（bit）总线。

总线带宽

可理解为总线的数据传输率，即单位时间内总线上可传输数据的位数，通常用每秒钟传送信息的字节数来衡量，单位可用字节/秒（B/s）表示。

总线带宽=总线工作频率×总线宽度（bit/s）=总线工作频率×(总线宽度$\div 8$)（B/s）=总线宽度÷总线周期（bit/s）=总线宽度$\div 8 \div$总线周期（B/s）。

总线带宽是指总线本身所能达到的最高传输速率。

在计算实际的有效数据传输率时，要用实际传输的数据量除以耗时。

总线复用

总线复用是指一种信号线在不同的时间传输不同的信息。可以使用较少的线传输更多的信息，从而节省了空间和成本。

信号线数

地址总线、数据总线和控制总线三种总线数的总和称为信号线数。

总线仲裁

总线仲裁基本概念

- 解决多个设备争用总线的问题。
- 同一时刻只能有一个设备控制总线传输操作，可以有一个或多个设备从总线接收数据
- 将总线上所连接的各类设备按其对总线有无控制功能分为：
  - 主设备：获得总线控制权的设备。
  - 从设备：被主设备访问的设备，只能响应从主设备发来的各种总线命令。
- 总线仲裁原因：
  - 总线作为一种共享设备，不可避免地会出现同一时刻有多个主设备竞争总线控制权的问题。
- 总线仲裁的定义：多个主设备同时竞争总线控制权时，以某种方式选择一个主设备优先获得总线控制权称为总线仲裁。

总线控制流程

- 在总线控制中，申请使用总线的设备向总线控制器发出**总线请求**，由总线控制器进行裁决。
- 若经裁决允许该设备使用总线，就由总线控制器向该设备发出**总线允许**信号。
- 该设备收到信号后发出**总线忙**信号，通知其他设备总线已被占用。
- 该设备使用完总线时，将**总线忙**信号撤销，释放总线。

总线仲裁分类

- 集中仲裁方式：
  - “总线忙”信号的建立者是获得总线控制权的设备。
  - 工作流程：
    1. 主设备发出请求信号。

2. 若多个主设备同时要使用总线，则由总线控制器的判优、仲裁逻辑按一定的优先等级顺序确定哪个主设备能使用总线。
3. 获得总线使用权的主设备开始传送数据。

- 链式查询方式。
- 计数器定时查询方式。
- 独立请求方式。

- 分布仲裁方式

链式查询方式

- 总线与控制线：
  - 数据线。
  - 地址线。
  - 控制线：
    - BG：总线允许（响应），1条。
    - BR：总线请求，1条。
    - BS：总线忙，1条。
- 流程：
  - 总线上所有的部件共用一根总线请求线，当有部件请求使用总线时，需经此线发总线请求信号到总线控制器。
  - 由总线控制器检查总线是否忙，若总线不忙，则立即发总线响应信号，经总线响应线BG串行地从一个部件传送到下一个部件，依次查询。
  - 若响应信号到达的部件无总线请求，则该信号立即传送到下一个部件；若响应信号到达的部件有总线请求，则信号被截住，不再传下去。
- 优先级：离总线控制器越近的部件，其优先级越高，离总线控制器越远的部件，其优先级越低。
- 优点：链式查询方式优先级固定。只需很少几根控制线就能按一定优先次序实现总线控制，结构简单，扩充容易。
- 缺点：
  - 对硬件电路的故障敏感。
  - 优先级不能改变。
  - 容易发生饥饿，当优先级高的部件频繁请求使用总线时，会使优先级较低的部件长期不能使用总线。

计数器定时查询方式

- 总线与控制线：
  - 数据线。
  - 地址线。
  - 设备地址线。
  - 控制线：

        - 设备地址线：传输设备地址，$\lceil \log_2 n \rceil$条。
        - BR：总线请求，1条。
        - BS：总线忙，1条。
        - 若设备有n个，则需要$\lceil \log_2 n \rceil + 2$条控制线。（还有$BS$、$BR$）
- 结构特点：用一个计数器控制总线使用权，相对链式查询方式多了一组设备地址线，少了一根总线响应线BG；它仍共用一根总线请求线。**不需要总线允许**信号。
- 过程：
    - 当总线控制器收到总线请求信号，判断总线空闲时，计数器开始计数，计数值通过设备地址线发向各个部件。
    - 当地址线上的计数值与请求使用总线设备的地址一致时，该设备获得总线控制权。
    - 同时，中止计数器的计数及查询。
- 优点：
    - 计数初始值可以改变优次序：
        - 计数每次从“0”开始，设各的优先级就按顺序排列，固定不变。
        - 计数从上一次的终点开始，此时设备使用总线的优先级相等。
        - 计数器的初值还可以由程序设置。
    - 对电路的故障没有链式敏感。
- 缺点:
    - 增加了控制线数。
    - 控制相对比链式查询相对复杂。

独立请求方式

- 总线与控制线：
    - 数据线。
    - 地址线。
    - 控制线：
        - BG：总线允许，n条。
        - BR：总线请求，n条。
        - BS：总线忙，1条。
        - 若设备有n个，则需要$2n + 1$条控制线。其中$+1$为BS线，用于设备向总线控制部件反馈已经使用完毕总线。
- 结构特点：每一个设备均有一对总线请求线BR_i和总线允许线BG_i。由排队器控制优先级。
- 过程：
    - 当总线上的部件需要使用总线时，经各自的总线请求线发送总线请求信号，在总线控制器中排队。

  - 当总线控制器按一定的优先次序决定批准某个部件的请求时，则给该部件发送总线响应信号。
- 优点：
  - 响应速度快，总线允许信号BG直接从控制器发送到有关设备，不必在设备间传递或者查询。
  - 对优先次序的控制相当灵活。
- 缺点：
  - 控制线数量多。
  - 总线的控制逻辑更加复杂。

分布仲裁方式

- 特点：不需要中央仲裁器，每个潜在的主模块都有自己的仲裁器和仲裁号，多个仲裁器竞争使用总线。
- 过程：
  - 当设备有总线请求时，它们就把各自唯一的仲裁号发送到共享的仲裁总线上。
  - 每个仲裁器将从仲裁总线上得到的仲裁号与自己的仲裁号进行比较。
  - 如果仲裁总线上的号优先级高，则它的总线请求不予响应，并撤销它的仲裁号。
  - 最后，获胜者的仲裁号保留在仲裁总线上。

总线操作与定时

占用总线的一对设备进行数据传输的方法。

总线传输阶段

1. 中请分配阶段：由需要使用总线的主模块（或主设备）提出申请，经总线仲裁机构决定将下一传输周期的总线使用权授予某一申请者。也可将此阶段细分为传输请求和总线仲裁两个阶段。
2. 寻址阶段：获得使用权的主模块通过总线发出本次要访问的从模块的地址及有关命令，启动参与本次传输的从模块。
3. 传输阶段：主模块和从模块进行数据交换，可单向或双向进行数据传送。
4. 结束阶段：主模块的有关信息均从系统总线上撤除，让出总线使用权。

总线定时

总线定时是指总线在双方交换数据的过程中需要时间上配合关系的控制，这种控制称为总线定时，它的实质是一种协议或规则。

同步定时方式

- 也称为同步通信。
- 系统采用一个统一的时钟信号来协调发送和接收双方的传送定时关系。

- 时钟产生相等的时间间隔，每个间隔构成一个总线周期。
- 在一个总线周期中，发送方和接收方可进行一次数据传送。
- 因为采用统一的时钟，每个部件或设备发送或接收信息都在固定的总线传送周期中，一个总线的传送周期结束，下一个总线传送周期开始。
- 上升沿：数字电平从0变为1的一瞬间。
- 下升沿：数字电平从1变为0的一瞬间。
- 读命令过程：
    1. 取指：CPU在T_1时刻的上升沿给出地址信息。
    2. 间址：在T_2的上升沿给出读命令（低电平有效），与地址信息相符合的输入设备按命令进行一系列的内部操作，且必须在T_3的上升沿来之前将CPU所需的数据送到数据总线上。
    3. 执行：CPU在T_3时钟周期内，将数据线上的信息传送到其内部寄存器电中。
    4. 撤销：CPU在T_4的上升沿撤销读命令，输入设备不再向数据总线上传送数据，撤销它对数据总线的驱动。
- 优点：
    - 传送速度快，具有较高的传输速率。
    - 总线控制逻辑简单。
- 缺点：
    - 主从设备属于强制性同步。
    - 不能及时进行数据通信的有效性检验，可靠性较差。
- 同步通信适用于总线长度较短及总线所接部件的存取时间比较接近的系统。

异步定时方式

- 也称为异步通信。
- 在异步定时方式中，没有统一的时钟，也没有固定的时间间隔，完全依靠传送双方相互制约的“握手”信号来实现定时控制。
- 主设备提出交换信息的“请求”信号，经接口传送到从设备；从设备接到主设备的请求后，通过接口向主设备发出“回答”信号。
- 根据“请求”和“回答”信号的撤销足否互锁，分为：
    1. 不互锁方式：
        - 主设备发出“请求”信号后，不必等到接到从设备的“回答”信号，而是经过一段时间，便撤销“请求”信号。
        - 而从设备在接到“请求”信号后，发出“回答”信号，并经过段时间，自动撤销“回答”信号。
        - 双方的不存在互锁关系。
        - 速度快。
        - 可靠性低。

2. 半互锁方式：
    - 主设备发出“请求”信号后，必须待接到从设备的“回答”信号后，才撤销“请求”信号，有互锁的关系。
    - 而从设备在接到“请求”信号后，发出“回答”信号，但不必等待获知主设备的“请求”信号已经撤销，而是隔一段时间后自动撤销“回答”信号，不存在互锁关系。
    - 若从设备故障，则会一直停留。
3. 全互锁方式：
    - 主设备发出“请求”信号后，必须待从设备“回答”后，才撤销“请求”信号。
    - 从设备发出“回答”信号，必须待获知主设备“请求”货报销后，再撤销其“回答”信号。
    - 双方存在互锁关系。

- 优点：总线周期长度可变，能保证两个工作速度相差很大的部件或设备之间可靠地进行信息交换，自动适应时间的配合。
- 缺点：
    - 比同步控制方式稍复杂。
    - 速度比同步定时方式慢。

半同步通信

同步与异步的混合，统一时钟的基础上，增加一个“等待”响应信号$\overline{WAIT}$。

分离式通信

即分离事务通信：

- 上述三种通信的共同点：一个总线传输周期（以输入数据为例）：
    - 主模块发地址、命令，使用总线。
    - 从模块准备数据，不使用总线，总线空闲。
    - 从模块向主模块发数据，使用总线。
- 分离式通信的一个总线传输周期：
    - 子周期一：主模块申请占用总线，使用完后后放弃总线的使用权。
    - 子周期二：从模块申请占用总线，将各种信息发送至总线上。
- 特点：
    1. 各模块均有权申请占用总线。
    2. 采用同步方式通信，不等对方回答。
    3. 各模块准备数据时，不占用总线。
    4. 总线利用率提高。

总线标准

总线标准概念

- 总线标准是国际上公布或推荐的互连各个模块的标准，它是把各种不同的模块组成计算机系统时必须遵守的规范。
- 按总线标准设计的接口可视为通用接口，在接口的两端，任何一方只需根据总线标准的要求完成自身方面的功能要求，而无须了解对方接口的要求。
- 系统总线标准：ISA、$EISA$、$VESA$、PCI、$PCI-Express$等。
- 设备总线标准：IDE、AGP、$RS-232C$、USB、$SATA$、$SCSI$、$PCMCIA$等。
- 局部总线标准：在ISA总线和CPU总线之间增加的一级总线或管理层，如PCI、$PCI-E$、$VESA$、AGP等，可以节省系统的总带宽。
- 即插即用（$Plug-and-Play$）的作用是自动配置（低层）计算机中的板卡和其他设备，然后告诉对应的设备都做了什么。把物理设备和软件（设备驱动程序）相配合，并操作设备，在每个设备和它的驱动程序之间建立通信信道。
- 热插拔（$hot-plugging$或$Hot\ Swap$）即带电插拔，热插拔功能就是允许用户在不关闭系统，不切断电源的情况下取出和更换损坏的硬盘、电源或板卡等部件，从而提高了系统对灾难的及时恢复能力、扩展性和灵活性等，例如一些面向高端应用的磁盘镜像系统都可以提供磁盘的热插拔功能。

标准总览

这是重点，下面的具体介绍可以随便看看：

总线标准	全称	工作频率	数据线	最大速度	特点
ISA	Industry Standard Architecture	8MHz	8/16	8MB/s	系统总线
EISA	Extended ISA	8MHz	32	32MB/s	系统总线
PCI	Peripheral Component Interconnect	33/66MHz	32/64	133/528MB/s	局部总线
AGP	Accelerated Graphics Port			X1:266MB/s X8:2.1GB/s	局部总线
VESA	Video Electronics Standard Architecture	33MHz	32	132MB/s	局部总线
PCI-E	PCI-Express (3GI0)			10GB/s 以上	串行
USB	Universal Serial Bus			1280MB/s	设备总线、串行

总线标准	全称	工作频率	数据线	最大速度	特点
RS-232C	Recommended Standard			20Kbps	串行通信总线
IDE（ATA)	Integrated Drive Electronics			100MB/s	硬盘光驱接口
SATA	Serial Advanced Technology Attachment			600MB/s	串行硬盘接口
PCMCIA	Personal Computer Memory Card International Association			90Mbps	便携设备接口
SCSI	Small Computer System Interface			640MB/s	智能通用接口

系统总线标准

总线标准	全称	工作频率	数据线	最大速度	特点
ISA	Industry Standard Architecture	8MHz	8/16	8MB/s	系统总线
EISA	Extended ISA	8MHz	32	32MB/s	系统总线

- *ISA*和*EISA*都是并行系统总线。
- *ISA*数据传送需要*CPU*或*DMA*接口来管理，传输速率过低、*CPU*占用率高、占用硬件中断资源，不支持总线仲裁。
- *EISA*从*CPU*中分离出了总线控制权，支持多个总线主控器和突发传送。
- 被*PCI*淘汰了。

局部总线标准

PCI

总线标准	全称	工作频率	数据线	最大速度	特点
PCI	Peripheral Component Interconnect	33/66MHz	32/64	133/528MB/s	局部总线

*PCI*也是并行局部总线，可以将高速外部设备直接挂到*CPU*总线上。

特点：

1. 高性能:不依附于某个具体的处理器，支持突发传送。

2. 良好的兼容性。
3. 支持即插即用。
4. 支持多主设备。
5. 具有与处理器和存储器子系统完全并行操作的能力。
6. 提供数据和地址奇偶校验的能力。
7. 可扩充性好,可采用多层结构提高驱动能力。
8. 采用多路复用技术，减少了总线引脚个数。

AGP

总线标准	全称	工作频率	数据线	最大速度	特点
AGP	Accelerated Graphics Port			X1:266MB/s X8:2.1GB/s	局部总线

*AGP*即加速图形接口，是并行局部总线，是显卡专用的局部总线。

PCI-E

总线标准	全称	工作频率	数据线	最大速度	特点
PCI-E	PCI-Express (3GI0)			10GB/s 以上	串行

- 点对点串行。
- 支持双向传输模式，可以运行全双工模式。
- 支持热插拔。

VESA

总线标准	全称	工作频率	数据线	最大速度	特点
VESA	Video Electronics Standard Architecture	33MHz	32	132MB/s	局部总线

即视频局部总线，为了传输活动图形。

设备总线标准

USB

总线标准	全称	工作频率	数据线	最大速度	特点
USB	Universal Serial Bus			1280MB/s	设备总线、串行

*USB*属于设备总线，是设备和设备控制器之间的接口。

特点：

1. 可以热插拔、即插即用。

2. 具有很强的连接能力和很好的可扩充性。采用菊花链形式将众多外设连接起来，可使用*USB*集线器链式连接127个外设。
3. 标准统一。以前大家常见的是*IDE*接口的硬盘，串口的鼠标键盘，并口的打印机扫描仪，可是有了*USB*之后，这些应用外设统统可以用同样的标准与个人电脑连接，这时就有了*USB*硬盘、*USB*鼠标、*USB*打印机等等。
4. 高速传输。
5. 连接电缆轻巧，可为低压（5*V*）外设供电。

IS-232C

总线标准	全称	工作频率	数据线	最大速度	特点
RS-232C	Recommended Standard			20Kbps	串行通信总线

应用于串行二进制交换的数据终端设备（*DTE*）和数据通信设备（*DCE*）。

IDE（ATA）

总线标准	全称	工作频率	数据线	最大速度	特点
IDE（ATA)	Integrated Drive Electronics			100MB/s	硬盘光驱接口

硬盘、光驱都通过*IDE*接口与主板连接。

SATA

总线标准	全称	工作频率	数据线	最大速度	特点
SATA	Serial Advanced Technology Attachment			600MB/s	串行硬盘接口

PCMCIA

总线标准	全称	工作频率	数据线	最大速度	特点
PCMCIA	Personal Computer Memory Card International Association			90Mbps	便携设备接口

SCSI

总线标准	全称	工作频率	数据线	最大速度	特点
SCSI	Small Computer System Interface			640MB/s	智能通用接口

1. *IDE*工作需要*CPU*参与，而*SCSI*通过独立高速*SCSI*卡控制读写，*CPU*就不需要等待。

2. $SCSI$扩充性高。

视频线标准

VGA

即$Video\ Graphics\ Array$，也称为$D-sub$端口，传输模拟信号，所以要先将数字信号转换为模拟信号。

DIV

即$Digital\ Visual\ Interface$，传输数字信号，但是在分辨率$1024\times768$以下时与$VGA$差别不大。

HDML

即$High\ Definition\ Multimedia\ Interface$，可以传输音讯，源于$DVI$技术，分为三种类型：

- A型：高清电视，投影仪等。
- C型：平板电脑，$MP4$等。
- D型：智能手机，平板电脑等。

输入输出系统

输入输出基本概念

I/O 系统发展

1. 早期，CPU和I/O串行工作，分散连接：
   - 程序查询方式：由CPU通过程序不断查询I/O设备是否已做好准备，从而控制I/O设备与主机交换信息。
2. 接口模块和DMA阶段，CPU和I/O并行工作，总线连接：
   - 中断方式：只在I/O设备准备就绪并向CPU发出中断请求时才予以响应。
   - DMA方式：主存和/O 设备之间有一条直接数据通路，当主存和I/O设备交换信息时，无需调用中断服务程序。
3. 具有I/O通道结构，在系统中设有通道控制部件，每个通道都挂接若干外设，主机在执行I/O命令时，只需启动有关通道，通道将执行通道程序，从而完成I/O操作。
4. 具有I/O处理机。

I/O 系统组成

1. I/O软件：

- 包括驱动程序、用户程序、管理程序、升级补丁等。
- 通常采用I/O指令和通道指令实现CPU和I/O设备的信息交换：
- I/O指令：
  - CPU指令的一部分。
  - 机器指令的一部分。
  - 反映CPU与I/O设备交换信息的特点。
  - 分为操作码（表明识别I/O指令）+命令码（执行的操作）+设备码（操作的设备）。（与一般指令格式不同）
- 通道指令（通道程序）：
  - 通道自身的指令。
  - 指出数据的首地址、传送字数、操作作命令。
  - 通道指令放在**主存**中。
  - 通道指令由**通道**执行。由CPU执行启动I/O设备的指令，通道执行通道指令代替CPU对I/O设备进行管理。
  - 只有具备通道的I/O系统才能执行。

2. I/O硬件：
   - I/O接口。
   - 设备控制器：通过设备控制器，I/O设备与主板的系统总线相联。
   - 外设。

外部设备

外部设备也称外围设备，是除了主机以外的、能直接或间接与计算机交换信息的装置。

输入设备

用于向计算机系统输入命令和文本、数据等信息的部件。键盘和鼠标是最基本的输入设备。

键盘

- 键盘是最常用的输入设备，通过它可发出命令或输入数据。
- 键盘通常以矩阵的形式排列按键，每个键用符号标明它的含义和作用。
- 每个键相当于一个开关，当按下键时，电信号连通；当松开键时，弹簧把键弹起，电信号断开。
- 键盘输入信息可分为三个步骤：
  1. 查出按下的是哪个键。
  2. 将该键翻译成能被主机接收的编码，如$ASCII$码。
  3. 将编码传送给主机。

鼠标

- 鼠标是常用的定位输入设备，它把用户的操作与计算机屏幕上的位置信息相联系。
- 常用的鼠标有机械式和光电式两种。
- 工作原理：当鼠标在平面上移动时，其底部传感器把运动的方向和距离检测出来，从而控制光标做相应运动。

输出设备

用于将计算机系统中的信息输出到计算机外部进行显示、交换等的部件。显示器和打印机是最基本的输出设备。

显示器

按显示设备所用的显示器件分类：

- 阴极射线管（CRT）显示器：
  - CRT显示器主要由电子枪、偏转线圈、荫罩、高压石墨电极和荧光粉涂层及玻璃外壳5部分组成。
  - 具有可视角度大、无坏点、色彩还原度高、色度均匀、可调节的多分辨率模式、响应时间极短等目前LCD难以超过的优点。
- 液晶显示器（LCD）：
  - 利用液晶的电光效应，由图像信号电压直接控制薄膜晶体管，再间接控制液品分子的光学特性来实现图像的显示。
  - 体积小、重量轻、省电、无辐射、绿色环保、画面柔、不伤眼等。
- LED显示器：通过控制半导体发光二极管进行显示，用来显示文字、图形、图像等各种信息。
- LCD、LED是两种不同的显示技术，LCD是由液态晶体组成的显示屏，而LED则是由发光二极管组成的显示屏。与LCD相比，LED显示器在亮度、功耗、可视角度和刷新速率等方面都更具优势。

按所显示的信息内容分类：

- 字符显示器：
  - 显示字符的方法以点阵为基础。点阵是指由$m \times n$个点组成的阵列。点阵的多少取决于显示字符的质量和字符窗口的大小。字符窗口是指每个字符在屏幕上所占的点数，它包括字符显示点阵和字符间隔。
  - 将点阵存入由ROM构成的字符发生器中，在CRT进行光栅扫描的过程中，从字符发生器中依次读出某个字符的点阵，按照点阵中0和1代码不同控制扫描电子束的开或关，从而在屏幕上显示出字符。对应于每个字符窗口，所需显示字符的$ASCII$代码被存放在视频存储器$VRAM$中，以备刷新。

  - 按扫描方式不同可分为：光栅扫描显示器、随机扫描显示器。
- 图形显示器：
  - 将所显示图形的一组坐标点和绘图命令组成显示文件存放在缓冲存储器中，缓存中的显示文件传送给矢量（线段）产生器，产生相应的模拟电压，直接控制电子束在屏幕上的移动。为了在屏幕上保留持久稳定的图像，需要按一定的频率对屏幕进行反复刷新。
  - 这种显示器的优点是分辨率高且显示的曲线平滑。目前高质量的图形显示器采用这种随机扫描方式。
  - 缺点是当显示复杂图形时，会有闪烁感。
- 图像显示器。

参数：

- 屏幕大小：以对角线长度表示，常用的有$12 \sim 29$英寸等。
- 分辨率：所能表示的像素个数，屏幕上的每一个光点就是一个像素，以宽、高的像素的乘积表示，例如，800×600、1024×768和1280×1024等。
- 灰度级：指黑白显示器中所显示的像素点的亮暗差别，在彩色显示器中则表现为颜色的不同，灰度级越多，图像层次越清楚逼真，典型的有8位（256级）、16位等。n位可以表示2^n种不同的亮度或颜色。
- 刷新：光点只能保持极短的时间便会消失，为此必须在光点消失之前再重新扫描显示一遍，这个过程称为刷新。
- 刷新频率：单位时间内扫描整个屏幕内容的次数，按照人的视觉生理，刷新频率大于$30Hz$时才不会感到闪烁，通常显示器刷新频率在60到$120Hz$。
- 显示存储器（$VRAM$）：也称刷新存储器，为了不断提高刷新图像的信号，必须把一帧图像信息存储在刷新存储器中。其存储容量由图像分辨率和灰度级决定，分辨率越高，灰度级越多，刷新存储器容量越大。
  - $VRAM$容量=分辨率×灰度级位数。
  - $VRAM$带宽=分辨率×灰度级位数×帧频。

每个存储汉字数据的汉字内码占$2B$，表示汉字形体的字形码以16×16点阵表示，为$32B$大小。

打印机

打印机是计算机的输出设备之一，用于将计算机处理结果打印在相关介质上。

按印字原理不同可分为：

- 击打式打印机：
  - 利用机械动作使印字机构与色带和纸相撞而打印字符。
  - 优点：设备成本低；印字质量好。
  - 缺点：噪声大；速度慢。

- 非击打式打印机:
  - 采用电、磁、光、喷墨等物理、化学方法来印刷字符。
  - 优点：速度快；噪声小。
  - 缺点：成本高。

按打印机工作方式不同可分为：

- 串行打印机：
  - 逐字打印。
  - 速度慢。
- 行式打印机：
  - 逐行打印。
  - 速度快。

按工作方式可分为：

- 针式打印机：
  - 原理：在联机状态下，主机发出打印命令，经接口、检测和控制电路，间歇驱动纵向送纸和打印头横向移动，同时驱动打印机间歇冲击色带，在纸上打印出所需内容。
  - 特点：针式打印机擅长“多层复写打印”，实现各种票据或蜡纸等的打印。它工作原理简单，造价低廉，耗材（色带）便宜，但打印分辨率和打印速度不够。
- 喷墨式打印机：
  - 原理：带电的喷墨雾点经过电极偏转后，直接在纸上形成所需字形。彩色喷墨打印机基于三基色原理，即分别喷射3种颜色墨滴，按一定的比例混合出所要求的颜色。
  - 特点：打印噪声小，可实现高质量彩色打印，通常打印速度比针式打印机快；但防水性差，高质量打印需要专用打印纸。
- 激光打印机：
  - 原理：计算机输出的二进制信息，经过调制后的激光束扫描，在感光鼓上形成潜像，再经过显影、转印和定影，便在纸上得到所需的字符或图像。
  - 特点：打印质量高、速度快、噪声小、处理能力强；但耗材多、价格较贵、不能复写打印多份，且对纸张的要求高。激光打印机是将激光技术和电子显像技术相结合的产物。感光鼓（也称为硒鼓)是激光打印机的核心部件。

外存设备

- 是指除计算机内存及CPU缓存等以外的存储器。硬磁盘、光盘等是最基本的外存设备。

- 计算机的外存储器又称为辅助存储器，目前主要使用磁表面存储器。
- 所谓“磁表面存储”，是指把某些磁性材料薄薄地涂在金属铝或塑料表面上作为载磁体来存储信息。磁盘存储器、磁带存储器和磁鼓存储器均属于磁表面存储器。
- 磁表面存储器的优点:
    1. 存储容量大，位价格低。
    2. 记录介质可以重复使用。
    3. 记录信息可以长期保存而不丢失，甚至可以脱机存档。
    4. 非破坏性读出，读出时不需要再生。
- 磁表面存储器的缺点:
    1. 存取速度慢。
    2. 机械结构复杂。
    3. 对工作环境要求较高。
- 原理：当磁头和磁性记录介质有相对运动时，通过电磁转换完成读/写操作。
- 编码方法：按某种方案（规律），把一连串的二进制信息变换成存储介质磁层中一个磁化翻转状态的序列，并使读/写控制电路容易、可靠地实现转换。
- 磁记录方式：通常采用调频制（FM）和改进型调频制（MFM）的记录方式。

磁盘存储器

- 磁盘设备的组成：
    - 存储区域：一块硬盘含有若干个记录面，每个记录面划分为若干条磁道，而每条磁道又划分为若干个扇区。
    - **扇区**（也称块）是磁盘读写的最小单位，也就是说磁盘按块存取。
        - 磁头数（$Heads$）：即记录面数，表示硬盘总共有多少个磁头，磁头用于读取/写入盘片上记录面的信息，一个记录面对应一个磁头。
        - 柱面数（$Cylinders$）：表示硬盘每一面盘片上有多少条磁道，一个盘组中，不同记录面的相同编号（位置）的诸磁道构成一个圆柱面。
        - 扇区数（$Sectors$）：表示每一条磁道上有多少个扇区。
    - 硬盘存储器：
        - 硬盘存储器由磁盘驱动器、磁盘控制器和盘片组成。
        - 磁盘驱动器：核心部件是磁头组件和盘片组件，温彻斯特盘是一种可移动头固定盘片的硬盘存储器。
        - 磁盘控制器：是硬盘存储器和主机的接口，主流的标准有IDE、$SCSI$、$SATA$等。
- 磁盘的性能指标：
    - 磁盘容量：一个磁盘所能存储的字节总数称为磁盘容量。

        - 非格式化容量是指磁记录表面可以利用的磁化单元总数。
        - 格式化容量是指按照某种特定的记录格式所能存储信息的总量。
    - 记录密度：指盘片单位面积上记录的二进制的信息量。
        - 道密度：沿磁盘半径方向单位长度上的磁道数。
        - 位密度：磁道单位长度上能记录的二进制代码位数。
        - 面密度：位密度和道密度的乘积。
        - 磁盘最里面的位密度最大，最外面的位密度最低。注意虽然每一道的道密度不同，但是每一道所含的数据量是一样的。
    - 平均存取时间=寻道时间（磁头移动到目的磁道）+旋转延迟时间（磁头定位到所在扇区）+传输时间（传输数据所花费的时间）。
    - 数据传输率：磁盘存储器在单位时间内向主机传送数据的字节数，称为数据传输率。
- 磁盘地址：
    - 主机向磁盘控制器发送寻址信息，磁盘的地址=驱动器号+柱面（磁道）号+盘面号+扇区号。
    - 若系统中有4个驱动器，每个驱动器带一个磁盘，每个磁盘256个磁道、16个盘面，每个盘面划分为16个扇区，则每个扇区地址要18位二进制代码：驱动器号（$2bit$）+柱面（磁道）号（$8bit$）+盘面号（$4bit$）+扇区号（$4bit$）。
- 磁盘工作过程：
    - 硬盘的主要操作是寻址、读盘、写盘。每个操作都对应一个控制字，硬盘工作时，第一步是取控制字，第二步是执行控制字。
    - 硬盘属于机械式部件，其读写操作是串行的，不可能在同一时刻既读又写，也不可能在同一时刻读两组数据或写两组数据。

磁盘阵列

- $RAID$（廉价冗余磁盘阵列）是将多个独立的物理磁盘组成一个独立的逻辑盘，数据在多个物理盘上分割交叉存储、并行访问，具有更好的存储性能、可靠性和安全性。
- $RAID$的分级如下所示。在$RAID1$到$RAID5$的几种方案中，无论何时有磁盘损坏，都可以随时拔出受损的磁盘再插入好的磁盘，而数据不会损坏：
    - $RAID0$：无冗余和无校验的磁盘阵列。
    - $RAID1$：镜像磁盘阵列。
    - $RAID2$：采用纠错的海明码的磁盘阵列。
    - $RAID3$：位交叉奇偶校验的磁盘阵列。
    - $RAID4$：块交叉奇偶校验的磁盘阵列。
    - $RAID5$：无独立校验的奇偶校验磁盘阵列。

光盘存储器

- 光盘存储器是利用光学原理读/写信息的存储装置，它采用聚焦激光束对盘式介质以非接触的方式记录信息。
- 光盘存储系统：
  - 光盘片：
    - 透明的聚合物基片。
    - 铝合金反射层。
    - 漆膜保护层的固盘。
  - 光盘驱动器。
  - 光盘控制器。
  - 光盘驱动软件。
- 特点：
  - 存储密度高。
  - 携带方便。
  - 成本低。
  - 容量大。
  - 存储期限长。
  - 容易保存。
- 光盘的类型如下：
  - $CD-ROM$：只读型光盘，只能读出其中内容，不能写入或修改。
  - $CD-R$：只可写入一次信息，之后不可修改。
  - $CD-RW$：可读可写光盘,可以重复读写。
  - $DVD-ROM$：高容量的$CD-ROM$，DVD表示通用数字化多功能光盘。

固态硬盘

- 在微小型高档笔记本电脑中，采用高性能$Flash\ Memory$作为硬盘来记录数据，这种“硬盘”称固态硬盘。
- 固态硬盘除了需要$Flash\ Memory$外，还需要其他硬件和软件的支持。
- 闪存（$Flash\ Memory$）是在$E2PROM$的基础上发展起来的，本质上是只读存储器。

I/O 接口

即I/O控制器，是主机和外设之间的交接界面，通过接口可以实现主机和外设之间的信息交换。

接口基本概念

接口功能

1. 设备选址：设备选择电路、控制逻辑电路。
2. 传送命令：命令寄存器、命令译码器。
3. 传送数据：数据缓冲寄存器DBR（格式转换）。
4. 反映I/O设备的工作状态：设备状态标记：
   - 完成触发器D。
   - 工作触发器B。
   - 中断请求触发器$INTR$。
   - 屏蔽触发器$MASK$。

I/O指令实现的数据传送是在靠近CPU的**通用寄存器**和用来缓冲I/O数据的 **I/O 接口**之间。

接口类型

- 按数据传送方式：
  - 并行接口：一个字节或一个字所有位同时传送。
  - 串行接口：一位一位地传送。
  - 这里所说的数据传送方式指的是外设和接口一侧的传送方式，而在主机和接口一侧,数据总是并行传送的。接口要完成数据格式转换。
- 按主机访问I/O设备的控制方式：
  - 程序查询接口。
  - 中断接口。
  - DMA接口。
- 按功能选择的灵活性：
  - 可编程接口。
  - 不可编程接口。

接口结构

- 内部接口：内部接口与系统总线相连，实质上是与内存、CPU相连。数据的传输方式只能是并行传输。
- 外部接口：外部接通过接口电缆与外设相连,外部接口的数据传输可能是串行方式，因此I/O接口需具有串/并转换功能。
- CPU与I/O接口之间通过数据线、地址线、命令线、状态线连接。
- I/O接口与外部设备通过数据线相连，I/O接口向外设通过命令线传递命令，外设向I/O接口通过状态线传递状态。
- CPU同外设之间的信息传送实质是对接口中的某些寄存器（即端口）进行读或写。
- 传输数据过程（以控制外设输入为例）：

1. CPU通过地址线传输选择设备。
2. 根据设备选择电路判断是否设备。
3. 根据设备状态标记反馈设备状态，通过状态线传输回CPU。
4. 若设备已就绪，CPU根据命令线发送控制命令。
5. 命令寄存器暂存命令，通过命令译码器转换为外设的控制信号。
6. I/O接口通过数据线向外设发送控制信号。
7. 外设通过数据线向I/O接口输入数据。
8. 数据暂存到数据缓冲寄存器DBR中进行处理。
9. 外设数据输入完成后，将状态通过状态线输入设备状态标记中。
10. I/O接口通过状态线向CPU反馈输入完成，产生中断请求。
11. CPU收到中断请求后通过命令线反馈中断响应。
12. I/O接口再通过数据线向CPU传输中断类型号。
13. CPU通过数据线接收数据。

端口与地址

- I/O端口是指接口电路中可以被CPU直接访问的**寄存器**。
- 接口$Interface$：
  - 端口$Port$：实际上是寄存器：
    - 数据端口：可读可写。
    - 控制端口：只写。
    - 状态端口：只读。
- 控制逻辑。

I/O端口要想能够被CPU访问，必须要有端口地址，每一个端口都对应着一个端口地址。

统一编址

- 把I/O端口当做存储器的单元进行地址分配，用统一的访存指令就可以访问I/O端口，又称存储器映射方式。
- 靠不同的地址码区分内存I/O设备，I/O地址要求相对固定在地址的某部分。
- 如系统总线中地址线共10根，则可以访问的存储单元个数为1024个，假设要给10个I/O端口编址:
  - 0到9表示I/O地址，10到1023为主存单元地址。
  - 0到1013表示主存单元地址，1014到1023为I/O地址。
  - 10到19表示I/O地址，0到9、20到1023为主存单元地址。
- 优点：
  - 不需要专门的输入/输出指令，可使CPU访问I/O的**访存指令**，操作更灵活、更方便。
  - 使端口有较大的编址空间。

- 缺点：
  - 端口占用了存储器地址，使内存容量变小。
  - 利用存储器编址的I/O设备进行数据输入/输出操作，执行速度较慢。

独立编址

- I/O端口地址与存储器地址无关，独立编址CPU需要设置专门的输入/输出指令访问端口，又称I/O映射方式。
- 靠不同的指令区分内存和I/O设备。
- 优点：输入/输出指令与存储器指令有明显区别，程序编制清晰，便于理解。
- 缺点：
  - 需要专门的输入输出端口。
  - 输入/输出指令少，一般只能对端口进行传送操作。
  - 需要CPU提供存储器读/写、I/O设备读/两组控制信号，增加了控制的复杂性。

I/O 方式

即主机和I/O设备之间的数据传送的控制方式。

程序查询方式

- 过程：
  - CPU执行初始化程序，并预置传送参数：设置计数器、设置数据首地址。
  - 向I/O接口发送命令字，启动I/O设备。
  - CPU从接口读取设备状态信息。
  - CPU不断查询I/O设备状态，直到外设准备就绪。CPU一旦启动I/O，必须停止现行程序的运行，并在现行程序中插入一段程序。主要特点：CPU有“踏步”等待现象，CPU与I/O串行工作。
  - 状态就绪后传输一次数据，一般为一个字。
  - 修改地址和计数器参数。
  - 判断传送是否结束（一般计数器为0时结束）。如果没有结束则继续回到停止等待状态。
- 优点：接口设计简单、设备量少。
- 缺点：CPU在信息传送过程中要花费很多时间用于查询和等待，而且在一段时间内只能和一台外设交换信息，效率大大降低。

程序中断方式

1. 中断请求：中断源向CPU发送中断请求信号。
2. 中断响应：响应中断的条件。中断判优：多个中断源同时提出请求时通过中断判优逻辑响应一个中断源。

3. 中断处理：
   - 中断隐指令。
   - 中断服务程序。

中断和异常

操作系统中主要涉及。

- 程序中断是指在计算机执行现行程序的过程中，出现某些急需处理的异常情况或特殊请求，CPU暂时中止现行程序，而转去对这些异常情况或特殊请求进行处理，在处理完毕后CPU又自动返回到现行程序的断点处，继续执行原程序。
- 中断（广义中断）：
  - 内中断（异常、例外、陷入）CPU内部与当前执行的指令有关：
    - 自愿中断：指令中断（自陷）。
    - 强迫中断：硬件故障（终止）、软件中断（故障）。
  - 外中断（中断）CPU外部与当前执行的指令有关：
    - 外设请求。
    - 人工干预。
    - 非屏蔽中断：关中断（中断标志位$IF = 0$）时也会被响应。（IF为$Interrupt\ Flag$，在PSW中）。
    - 可屏蔽中断：关中断时不会被响应。
- 异常由指令在执行种产生，而中断不与指令相关，也不阻止指令的完成。
- 异常的检测由CPU完成，不需要外部信号通知。中断必须CPU通过总线获取中断源标记信息才能知道中断的类型。
- 异常是不可屏蔽的中断，而通过$INTR$信号线发出的中断是可屏蔽的中断。

中断请求

- 中断请求标记：
  - 每个中断源向CPU发出中断请求的时间是随机的。
  - 为了记录中断事件并区分不同的中断源，中断系统需对每个中断源设置中断请求标记触发器$INTR$，当其状态为“1“时，表示中断源有请求。
  - 这些触发器可组成中断请求标记寄存器，该寄存器可集中在CPU中，也可分散在各个中断源中。
  - 对于外中断，CPU是在统一的时刻即每条指令执行阶段结束前向接口发出中断查询信号，以获取I/O的中断请求，也就是说，CPU响应中断的时间是在每条指令执行阶段的结束时刻。
- 中断判优，即多个中断发生时处理顺序的训责：
  - 中断判优既可以用硬件实现，也可用软件实现。

  - 硬件实现是通过硬件排队器实现的，它既可以设置在CPU中，也可以分散在各个中断源中。
  - 软件实现是通过查询程序实现的。
- 中断优先级：
  1. 硬件故障中断属于最高级，其次是软件中断。
  2. 非屏蔽中断优于可屏蔽中断。
  3. DMA请求优于I/O设备传送的中断请求。
  4. 高速设备优于低速设备。
  5. 输入设备优于输出设备。
  6. 实时设备优于普通设备。

中断响应

- CPU响应中断必须满足以下三个条件：
  - 中断源有中断请求。
  - CPU允许中断即开中断。
  - 一条指令执行完毕，且没有更紧迫的任务。

中断隐指令指CPU在响应中断后在执行中断服务程序前的准备操作。由硬件实现，不是指令系统的真实指令，没有操作码，也不能被用户使用。

中断处理

1. 关中断。
2. 保存断点。
3. 识别中断源并引出中断服务程序。
4. 保存现场和中断屏蔽字。现场是用户可见的寄存器的内容，由软件实现。屏蔽字是CPU中断响应开始保存到栈或专门寄存器，由硬件实现。
5. 执行中断服务程序（中断事件处理）。
6. 恢复现场和屏蔽字。
7. 开中断。
8. 中断返回。

- 中断隐指令的主要任务（1到3）：
  1. 关中断：在中断服务程序中，为了保护中断现场（即CPU主要寄存器中的内容）期间不被新的中断所打断，必须关中断，从而保证被中断的程序在中断服务程序执行完毕之后能接着正确地执行下去。即通过硬件自动完成中断触发器置零。
  2. 保存断点：为了保证在中断服务程序执行完毕后能正确地返回到原来的程序，必须将原来程序的断点（即程序计数器PC的内容）保存起来。可以存入堆栈，也可以存入指定单元。

3. 引出中断服务程序：引出中断服务程序的实质就是取出中断服务程序的入口地址并传送给程序计数器PC。
    - 软件查询法。
    - 硬件向量法：每个中断有类型号，每个中断类型号对应一个中断服务程序，每个程序有一个入口地址，CPU需要找到这个地址，这就是中断向量。由中断向量地址形成硬件产生中断向量地址（中断类型号），再由向量地址找到入口地址。系统种保存中断向量的存储器就是中断向量表。（**中断向量**就是服务程序的入口地址，**中断向量地址**是入口地址的地址）

- 中断服务程序的任务（4到8）：
    1. 保护现场：
        - 保存程序断点PC，已由中断隐指令完成。
        - 二是保存通用寄存器和状态寄存器的内容，由中断服务程序完成。
        - 可以使用堆栈，也可以使用特定存储单元。
    2. 中断服务（设备服务）：主体部分，如通过程序控制需打印的字符代码送入打印机的缓冲存储器中。
    3. 恢复现场：通过出栈指令或取数指令把之前保存的信息送回寄存器中。
    4. 中断返回：通过中断返回指令回到原程序断点处。

多重中断技术

- 单重中断：执行中断服务程序时不响应新的中断请求。
- 多重中断：又称中断嵌套，执行中断服务程序时可响应新的中断请求。

	单重中断	多重中断
中断隐指令	关中断	关中断
	保存断点（PC)	保存断点（PC)
	送中断向量	送中断向量
中断服务程序	保护现场	保护现场和屏蔽字
		开中断，中断嵌套
	执行中断服务程序	执行中断服务程序
		关中断
	恢复现场	恢复现场和屏蔽字
	开中断	开中断
	中断返向	中断返回

- 中断屏蔽技术：主要用于多重中断，CPU要具备多重中断的功能，须满足下列条件：
    - 在中断服务程序中提前设置开中断指令。

  - 优先级别高的中断源有权中断优先级别低的中断源。
- 所有屏蔽触发器组合在一起，便构成一个屏蔽字寄存器，屏蔽字寄存器的内容称为屏蔽字。
- 中断屏蔽标志可以改变多个中断服务程序**执行完**的次序。但是不能改变开始执行的次序，因为中断请求响应顺序由排队器控制。
- 屏蔽字设置规律：
  - 1表示屏蔽该中断源的请求，0表示可以正常申请。
  - 每个中断源对应一个屏蔽字（在处理该中断源的中断服务程序时，屏蔽寄存器中的内容为该中断源对应的屏蔽字）。
  - 屏蔽字中1越多，优先级越高。每个屏蔽字中至少有一个1（至少要能屏蔽自身的中断）。
- 在保护现场关中断，在执行中断处理程序时开中断。

程序中断方式实现

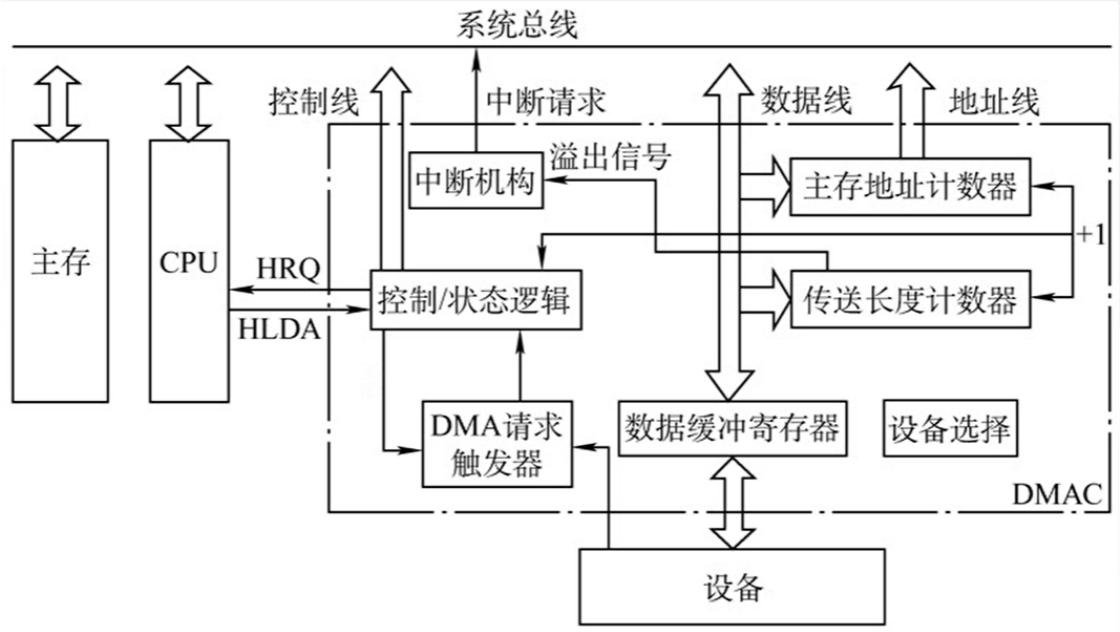
1. CPU遇到I/O指令，就去启动外设，然后CPU再回来执行现有程序，外部设备进行准备。
2. 外设准备完成后向CPU发出中断请求，CPU接收后进行响应，进行中断隐指令操作：关中断、保存断电、引出中断服务程序。CPU响应中断，运行中断服务程序：保护现场、中断服务程序控制数据传送。外设准备下一个请求。
3. CPU通过中断服务程序进行中断返回，若从 K 个程序开始中断，则返回到$K+1$个程序。

DMA 方式

DMA方式优先级高于中断方式，这样处理速度更快。

DMA 控制器

- 因为每一个数据都要中断一次，当有大量数据时CPU需要大量中断服务。
- 所以使用硬件——DMA控制器控制大量数据传输。
- 在DMA方式中，当I/O设备需要进行数据传送时，通过DMA控制器（DMA接口）向CPU提出DMA传送请求，CPU响应之后将让出系统总线，由DMA控制器接管总线进行数据传送。其主要功能有：
  1. 接受外设发出的DMA请求，并向CPU发出**总线**请求。
  2. 在每个**存储周期**（机器周期）后CPU检查是否存在DMA请求，若有则响应此总线请求，发出总线响应倍号，接管总线控制权，进入DMA操作周期。
  3. 确定传送数据的主存单元地址及长度，并能自动修改主存地址计数和传送长度计数。
  4. 规定数据在主存和外设间的传送方向，发出读写等控制信号，执行数据传送操作。
  5. 向CPU报告DMA操作的结束，发出中断。



DMA 控制器

DMA控制时，CPU对主存没有控制权，而DMA有。

- 控制/状态逻辑：由控制和时序电路及状态标志组成，用于指定传送方向，修改传送参数，并对DMA请求信号和CPU响应信号进行协调和同步。
- DMA请求触发器：每当I/O设备准备好数据后给出一个控制信号，使DMA请求触发器置位（为1）。
- 主存地址计数器AR：存放交换的数据的主存地址，能自动加减1。
- 传送长度计数器WC：用来记录传送数据的长度，计数溢出时，数据即传送完毕，自动发中断请求信号。
- 数据缓冲寄存器：用于暂存每次传送的数据。
- 中断机构：当一个数据块传送完毕后触发中断机构，向CPU提出中断请求。

DMA 传送过程

CPU只在开始和结尾参与控制，DMA控制整个传输过程：内存→数据总线→$DMAC$（DMA控制器）→外设。

- 预处理：
  - 主存起始地址→ AR。
  - I/O设备地址→ DAR。
  - 传送数据个数→ WC。
  - 启动I/O设备
- 数据传送：继续执行主程序同时完成一批数据的传送。以数据输入为例：
  - 设备将数据输入DR中。

- 写满后向DMA控制器发送DMA请求。
- DMA控制器向总线发送总线请求。
- CPU将总线控制权交给DMA控制器，DMA控制器接管总线。
- DMA控制器将地址从主存地址计数器上送到地址线上，将数据从数据缓冲寄存器送到数据线上。
- 修改地址和长度参数并返回传送长度计数器和主存地址计数器中。
- 若溢出（传输结束），则向中断机构发出中断。

- 后处理：中断服务程序做DMA结束处理。
- 继续执行主程序。

DMA 传送方式

- 主存和DMA控制器之间有一条数据通路，因此主存和I/O设备之间交换信息时，不通过CPU。
- 但当I/O设备和CPU同时访问主存时，可能发生冲突，为了有效地使用主存，DMA控制器与CPU通常采用以下三种方法使用主存：
  1. 停止CPU访问主存：
     - 控制简单。
     - CPU处于不工作状态或保持状态。
     - 未充分发挥CPU对主存的利用率。
  2. DMA和CPU交替访存：将一个CPU周期分为两个周期，一个给DMA使用一个给CPU使用：
     - 不需要总线使用权的申请、建立和归还过程。
     - 硬件逻辑更为复杂。
  3. 周期挪用（周期窃取）：
     - CPU此时不访存：不冲突。
     - CPU正在访存：存取周期结束让出总线给DMA。
     - CPU、DMA同时请求访存：DMA的I/O访存优先，挪用几个存取周期。传输一个字后立刻释放总线。

DMA 方式特点

- 主存和DMA接口之间有一条直接数据通路——DMA总线。
- 由于DMA方式传送数据不需要经过CPU，因此不必中断现行程序，I/O与主机并行工作，程序和传送并行工作。
- DMA方式具有下列特点:
  1. 它使主存与CPU的固定联系脱钩，主存既可被CPU访问，又可被外设访问。
  2. 在数据块传送时，主存地址的确定、传送数据的计数等都由硬件电路直接实现。
  3. 主存中要开辟专用缓冲区，及时供给和接收外设的数据。

4. *DMA*传送速度快，*CPU*和外设并行工作，提高了系统效率。
5. *DMA*在传送开始前要通过程序进行预处理，结束后要通过中断方式进行后处理。
6. 不用保护和恢复现场。
7. 不能立即响应，所以不适合键盘和鼠标这种情景。

I/O 方式对比

	CPU 与外设	传送与主程序
程序查询	串行	串行
程序中断	并行	串行
DMA	并行	并行

	中断	DMA
数据传送	软件控制，程序的切换→保存和恢复现场	硬件控制，CPU 只需进行预处理和后处理
请求资源	CPU 处理时间	总线使用权
响应	指令周期结束后	机器周期结束后，总线事务后
场景	CPU 控制，低速设备	DMA 控制器控制，高速设备
优先级	非屏蔽中断高于屏蔽中断	最高
异常处理	能处理异常事件	仅传送数据

汇编程序

概念

定义

在计算机组成原理中会涉及汇编语言的基本指令，需要对基本命令有基本的掌握。

机器语言是机器指令的集合。这些机器指令本质上就是由一组 0 和 1 组成的命令，是 CPU 唯一能解释执行的命令。

机器语言和高级语言之间的就是汇编语言，汇编语言是人类可以理解的语言。

汇编语言的主体是汇编指令，汇编指令和机器指令的差别在于指令的表示方法上，汇编指令是机器指令的助记符，汇编指令是更便于记忆的一种书写格式。它较为有

效地解决了机器指令编写程序难度大的问题，汇编语言与人类语言更接近，便于阅读和记忆。使用编译器，可以把汇编程序转译成机器指令程序。

```
1000100111011000
MOV AX,BX
```

汇编语言不区分大小写。

表示把寄存器 BX 的内容移动到寄存器AX中。

运行流程

这就需要对汇编程序进行编译，即将其翻译成由机器指令组成的机器码，如此计算机就能执行汇编程序上。本质上，计算执行的仍然是机器指令，而非汇编指令。对更多高级的编程语言（比如$C++$等），使用高级语言编写的程序同样需要先转译成汇编程序，再编译成机器码，从而进一步地在机器上运行。这个过程，即编译过程。

把机器码程序（机器指令程序）转译成汇编指令，即反编译过程。

汇编语言组成

1. 汇编指令：与机器指令一一对应，它是机器码的助记符，汇编后变成机器语言可以被计算机CPU执行。
2. 伪指令：由编译器识别执行，用于指示汇编程序如何汇编源程序，也称为命令语句，编译结束后伪指令失去作用，计算机CPU不执行。
3. 其它符号：如+、-等，由编译器识别执行，没有对应机器码。

汇编语言的核心是汇编指令，汇编指令决定了汇编程序的特性。

8086CPU

是一个由$Intel$于1978年所设计的16位微处理器芯片，是x86架构的鼻祖。

寄存器

8086CPU有14个16位寄存器，既能处理16位数据，也能处理8位数据，分别为：

通用寄存器，用来存放一般性数据：

- 数据寄存器：一般用于存放参与运算的操作数或运算结果。分为两个字节，高位和低位，其中高位为$\cdots H$，低位为$\cdots L$，如AX分为两个八位的AH和AL：
  - AX（$Accumulator$）：累加器。用该寄存器存放运算结果，可提高指令的执行速度。此外，所有的I/O指令都使用该寄存器与外设端口交换信息。

  - BX（$Base$）：基址寄存器。8086/8088CPU中有两个基址寄存器BX和BP。BX用来存放操作数在内存中数据段内的偏移地址，BP用来存放操作数在堆栈段内的偏移地址。
  - CX（$Counter$）：计数器。在设计循环程序时，使用该寄存器存放循环次数，可使程序指令简化，有利于提高程序的运行速度。
  - DX（$Data$）：数据寄存器。在寄存器间接寻址的I/O指令中存放I/O端口地址；在做双字长乘除法运算时，DX与AX一起存放一个双字长操作数，其中DX存放高16位数。
- 地址指针寄存器：
  - SP（$Stack\ Pointer$）：堆栈指针寄存器。在使用堆栈操作指令（$PUSH$或POP）对堆栈进行操作时，每执行一次进栈或出栈操作，系统会自动将SP的内容减2或加2，以使其始终指向栈顶。其中ESP（$ExtendedStackPointer$）为扩展栈指针寄存器，是32位寄存器，用于存放函数栈顶指针。
  - BP（$Base\ Pointer$）：基址寄存器。作为通用寄存器，它可以用来存放数据，但更经常更重要的用途是存放操作数在堆栈段内的偏移地址。EBP（$ExtendedBasePointer$），扩展基址指针寄存器，也被称为帧指针寄存器，是32位寄存器，用于存放函数栈底指针。
- 变址寄存器：
  - SI（$Source\ Index$）：源变址寄存器。存放源串在数据段内的偏移地址。
  - DI（$Destination\ Index$）：目的变址寄存器。存放目的串在附加数据段内的偏移地址。

段寄存器，为了对$1M$个存储单元进行管理，8086/8088CPU对存储器进行分段管理，即将程序代码或数据分别放在代码段、数据段、堆栈段或附加数据段中，每个段最多可达$2^{16} = 64K$个存储单元。段地址分别放在对应的段寄存器中，代码或数据在段内的偏移地址由有关寄存器或立即数给出：

- CS（$Code\ Segment$）：代码段寄存器。用来存储程序当前使用的代码段的段地址。CS的内容左移四位再加上指令指针寄存器IP的内容就是下一条要读取的指令在存储器中的物理地址。
- SS（$Stack\ Segment$）：堆栈段寄存器。用来存储程序当前使用的堆栈段的段地址。堆栈是存储器中开辟的按先进后出原则组织的一个特殊存储区，主要用于调用子程序或执行中断服务程序时保护断点和现场。
- DS（$Data\ Segment$）：数据段寄存器。用来存储程序当前使用的数据段的段地址。DS的内容左移四位再加上按指令中存储器寻址方式给出的偏移地址即得到对数据段指定单元进行读写的物理地址。
- ES（$Extra\ Segment$）：附加数据段寄存器。用来存储程序当前使用的附加数据段段的段地址。附加数据段用来存放字符串操作时的目的字符串。

段寄存器	提供段内偏移地址的寄存器
CS	IP
DS	BX、SI、DI 或一个 16 位数
SS	SP 或 BP
ES	DI（用于字符串操作指令）

控制寄存器：用于存放控制数据：

- IP（$Instruction\ Pointer$）：指令指针寄存器。用来存放下一条要读取的指令在代码段内的偏移地址。用户程序不能直接访问IP。
- $FLAGS$：标志寄存器，也称为程序状态寄存器（PSW）。它是一个16位的寄存器，但只用了其中的9位，包括6个状态标志位和3个控制标志位。

$FLAGS$标志位的索引从 0 到 15，则：

状态标志位，用于查看运算数据状态：

简写	英文名称	中文名称	索引	描述	用途
CF	Carry Flag	进位标志位	0	当进行加减运算时，若最高位发生进位或借位，则 CF 为 1，否则为 0	判断十六位无符号数运算结果是否超出了计算机所能表示的无符号数的范围
PF	Parity Flag	奇偶标志位	2	当指令执行结果的低 8 位中含有偶数个 1 时，PF 为 1，否则为 0	奇偶校验
AF	Auxiliary Flag	辅助进位标志位	4	当执行一条加法或减法运算指令时，若结果的低字节的低 4 位向低字节的高 4 位（0-3 向 4-7）有进位或借位，则 AF 为 1，否则为 0	判断四位无符号数运算结果是否超出了计算机所能表示的无符号数的范围，如 BCD 码
ZF	Zero Flag	零标志位	6	若当前的运算结果为 0，则 ZF 为 1，否则为 0	快速判断 0
SF	Sign Flag	符号标志位	7	若运算结果的最高位为 1，SF=1，否则为 0	快速获取符号
OF	Overflow Flag	溢出标志位	11	当运算结果超出了带符号数所能表示的数值范围，即溢出时，OF=1，否则为 0	该判断带符号数运算结果是否溢出

控制标志位有三个，用于控制 CPU 的操作，由程序设置或清除：

简写	英文名称	中文名称	索引	描述	用途
TF	Trap Flag	跟踪（陷阱）标志位	8	若将 TF 置 1，CPU 处于单步工作方式，每执行一条指令后产生一次单步中断	简化测试程序
IF	Interrupt Flag	中断允许标志位	9	若将 IF 置 1，表示允许 CPU 接受外部从 INTR 引脚上发来的可屏蔽中断请求；若用 CLI 指令将 IF 清 0，则禁止 CPU 接受可屏蔽中断请求信号	控制 CPU 是否可以响应 CPU 外部的可屏蔽状态中断请求
DF	Direction Flag	方向标志位	10	若将 DF 置为 1，串操作按减地址方式进行，也就是说，从高地址开始，每操作一次地址自动递减；否则按增地址方式进行	串操作时控制数据计算的方向

寻址

有16根数据线和20根地址线。

由于20根地址线，所以可寻址的内存空间为$2^{20} = 1MB$。但是由于数据线只有16位，所以CPU内部只能传输16位的地址。所以为了解决这个问题，8086CPU使用了两个十六位地址通过地址加法器合成一个二十位物理地址的方式。

其中定义第一个地址为段地址，第二个地址为偏移地址，即物理地址=段地址$\times 16 +$偏移地址。

如传入两个十六位地址，$1230H$和$00C8H$，然后$1230H \times 16 = 12300H$（乘其对应的进制数等价于左移一位），然后$12300H + 00C8 = 123C8$得到最终地址。

所以给定一个段地址，仅通过变化n位的偏移地址来寻址最多可以定位2^n个内存单元。

工作流程

在8086CPU加电启动或复位后（即CPU刚开始工作时）CS被设置为$FFFFH$，而IP被设置为$0000H$，即在8086PC机刚启动时，CPU使用地址加法器从内存$FFFF \times 10H + 0000H = FFFF0H$单元中读取指令执行，$FFFF0H$单元中的指令是8086$PC$机开机后执行的第一条指令。

- 在CPU中，程序员能够用指令读写的部件只有寄存器，程序员可以通过改变寄存器中的内容实现对CPU的控制。

- *CPU*从何处执行指令是由*CS*、*IP*中的内容决定的，程序员可以通过改变*CS*、*IP*中的内容来控制*CPU*执行目标指令。
- 但是不能直接改变*CS*、*IP*寄存器的值，所以8086*CPU*使用转移指令来进行设置 `JMP 段地址:偏移地址`。
- 其中段地址用来修改*CS*，偏移地址用来修改*IP*。如果仅修改*IP*内容，可以使用 `JMP 寄存器地址`，如 `JMP AX` 类似于 `MOV IP,AX`。

汇编指令

数据传输指令

通用数据传送指令：

- *MOV*：
  - 格式：`MOV 目的,源`。
  - 功能：把源操作数赋值传送给目的操作数，源操作数不变。
  - 要求：目的和源可以为数据也可以为地址。
  - 禁止操作：向段寄存器写入立即数（必须通过通用寄存器中转）；*CS*作为目标寄存器；目的地址与源地址同时为存储单元或段寄存器。
- *MOVSX*：
  - 格式：`MOVSX 目的寄存器,源操作数`。
  - 功能：把源操作数先进行符号拓展（为正数则高位补 0，负数则高位补 1），再赋值传送给目的操作数，源操作数不变。
  - 要求：源操作数字长要小于或等于目标寄存器字长。
- *MOVZX*：
  - 格式：`MOVZX 目的寄存器,源操作数`。
  - 功能：把源操作数先进行零拓展（高位补 0），再赋值传送给目的操作数，源操作数不变。
  - 要求：源操作数字长要小于或等于目标寄存器字长。
- *LEA*：
  - 格式：`LEA 目的寄存器,源操作数`。
  - 功能：计算内存单元的有效地址（不是其中的操作数）并送到目的寄存器中。
  - 说明：有效地址就是偏移地址，LEA 指令等效于 OFFSET 运算符，计算 DS 段地址加上偏移地址的地址。
- *XCHG*：
  - 格式：`XCHG 操作数 1,操作数 2`。
  - 功能：对两个操作数进行位置互换。
  - 要求：操作数类型需要一致；至少一个为寄存器。
  - 禁止操作：操作数为段寄存器或立即数或内存操作数。

- $CMPXCHG$：
  - 格式：CMPXCHG 操作数 1,操作数 2。
  - 功能：比较并对两个操作数进行位置互换。
  - 要求：第二个操作数必须为累加器$AL/AX/EAX$。
  - 禁止操作：操作数为段寄存器或立即数或内存操作数。

堆栈传输指令：

堆栈操作指令中的操作数类型必须是字操作数，即 16 位操作数。

- $PUSH$：
  - 格式：PUSH 源地址。
  - 功能：将字压入堆栈。
  - 说明：一个字进栈，系统自动完成两步操作：$SP \leftarrow SP-2$，$(SP) \leftarrow$操作数。
  - 说明：一个双字进栈，系统自动完成两步操作：$ESP \leftarrow ESP-4$，$(ESP) \leftarrow$操作数。
- POP：
  - 格式：POP 目的地址。
  - 功能：将字弹出堆栈。
  - 说明：弹出一个字，系统自动完成两步操作：操作数$\leftarrow (SP)$，$SP \leftarrow SP+2$。
  - 说明：弹出一个双字，系统自动完成两步操作：操作数$\leftarrow (ESP)$，$ESP \leftarrow SP+4$。
- $PUSHA$：
  - 格式：PUSHA。
  - 功能：把通用寄存器AX,CX,DX,BX,SP,BP,SI,DI依次全部压入堆栈。
  - 要求：为16位字。
- $POPA$：
  - 格式：POPA。
  - 功能：把通用寄存器DI,SI,BP,SP,BX,DX,CX,AX依次全部弹出堆栈。
  - 要求：为16位字。
- $PUSHAD$：
  - 格式：PUSHAD。
  - 功能：把拓展通用寄存器AX,CX,DX,BX,SP,BP,SI,DI依次全部压入堆栈。
  - 要求：为32位字。
- $POPAD$：
  - 格式：POPAD。

- 功能：把拓展通用寄存器$DI, SI, BP, SP, BX, DX, CX, AX$依次全部弹出堆栈。
- 要求：为32位字。

环境

下载

首先下载 DOSBox，下载后点击安装。DOSBox 即通过软件让在不同的系统如 Windows、Linux、MacOS 等运行 DOS 程序，直接操作硬件。

然后下载 MASM5.0，将压缩包解压并把对应文件夹放到指定位置。宏汇编程序$MASM$是具有宏加工功能的汇编程序。可以用它定义含参数的程序段，在使用的位置上调用它们，汇编时将进行宏指令展开，把宏定义所预先定义的指令目标代码插在该位置上。

挂载

完成后打开$DOSBox$的$DOSBox.exe$程序。

由于这个软件的起始目录是Z，并没有我们电脑上的任何本地目录，所以我们需要首先将我们自己的目录给挂载到$DOSBox$系统中。

需要将$MASM$目录挂载在$DOSBox$中，输入 `MOUNT` 挂载虚拟盘符 `MASM5.0` 安装地址（命令不区分大小写）。如我的$MASM$5.0安装在D盘的$MASM$目录下，要挂载到D盘中运行，就要在$DOSBox$控制台中输入 `MOUNT D: D://MASM`。如果挂载成功会显示 Drive\,D\,is\,mounted\,as\,local\,directory\,D://MASM\\

然后查看部署是否成功，`D:`、`dir`，如果显示列表中出现了$DEBUG$就代表 MASM 挂载成功，可以使用里面的$DEBUG$进行操作。

然后挂载成功后输入 `D:`进入挂载地址，输入 `DEBUG` 就可以开始使用$DEBUG$了（汇编程序执行器）。

此后每次使用$DOSBox$就必须挂载$MASM$，非常麻烦，可以在配置文件中配置挂载命令。

找到$DOSBox\ 0.74-3, Options.bat$，双击打开配置文件（注意不要直接修改 bat 文件），在末尾添加：

```
MOUNT D: D://MASM
D:
```

就相当于默认挂载，并转到虚拟盘符地址，点击$DOSBox.exe$可以直接使用相关工具了。

- $EDIT$编辑器。

- *MASM*编译器。
- *LINK*连接器。
- *DEBUG*调试器。

DEBUG

- *R*命令查看、改变*CPU*寄存器的内容。
  - R 查看所有寄存器内容。
  - R 寄存器名查看指定寄存器内容。然后下一行弹出一个:，后面可以添加值，从而能修改指定寄存器内容。
- *D*命令查看内存中的内容。其中左边为十六进制值，右边为对应 ASCII 值。
  - D 段地址:偏移地址：查看指定段地址与偏移地址所指向的内存后 128 个内存单元中内容。
  - D 段地址:偏移地址 数量：查看指定段地址与偏移地址所指向的内存后指定数量的内存单元中的内容。
- *E*命令改写内存中的内容。
  - E 段地址:偏移地址 数据列表：从指定段地址与偏移地址所指向的内存开始修改内容为数据列表。数据列表中间以空格分隔。
  - E 段地址:偏移地址：不表明数据列表，则控制台会主动给你提供从该地址开始的原数据，可以在后面输入新数据，输入空格后会继续弹出下个地址的原数据，回车结束输入执行修改。
  - E 段地址:偏移地址 "字符串"：从指定段地址与偏移地址所指向的内存开始保存字符串值，对应的 ASCII 码自动转换并保存到内容中。
- *U*命令将内存中的机器指令翻译成汇编指令。U 段地址:偏移地址。结果分别为地址、机器指令、汇编指令、说明。
- *A*命令以汇编指令的格式在内存中写入一条机器指令。A 段地址:偏移地址。从该地址开始输入汇编指令保存到对应内存。
- *T*命令执行一条机器指令。T 段地址:偏移地址。从该地址执行保存的汇编指令。

操作

数据操作

如要读取10000*H*单元的内容可以使用下面的汇编程序：

```
MOV BX,1000H
MOV DS,BX
MOV AL,[0]
```

首先我们要读取10000*H*单元的内容，则必须把这个内容移动到段寄存器中然后读取。但是段寄存器不能直接被赋值，必须通过通用寄存器。所以第一步就是将1000*H*移动到基址通用寄存器*BX*中，因为是基址所以将10000*H*右移四位变为

$1000H$为基址（$10000H$表示欸$1000:0$）。然后将$1000H$基址从基址寄存器中移动到数据段寄存器DS中，此时可以通过DS读取数据了。其中$[X]$表示内存单元，X表示相对于段寄存器地址的偏移量，所以$[0]$的计算后的基址就是$1000H \times 16 + 0 = 10000H$，然后通过$MOV$命令移动到$AL$累加器寄存器中。移动到$AL$而不是$AX$是因为一个内存单位为一个字节，为八比特，而$8086CPU$一个字为两个字节，所以只用累加器的低位就可以了，即AL即可。

所以针对数据传输必然是：数据→通用寄存器→段寄存器。

字操作

由于$8086CPU$数据线为16根，所以其字大小为十六位即两个字节。之前数据传输时只是传输一个字节，所以通用寄存器使用AL而不是AX，而这里要传输字，所以使用一整个通用寄存器。

假如内存情况为：

地址	数据
10000H	23
10001H	11
10002H	22
10003H	66

执行下面的汇编程序，请问执行后寄存器会变成什么样：

```
MOV AX,1000H
MOV DS,AX
MOV AX,[0]
MOV BX,[2]
MOV CX,[1]
ADD BX,[1]
ADD CX,[2]
```

计算机默认为小端模式，所以高位在高地址，低位在地地址，所以比如要读取10000H 指向的字，不能只读一个23，因为只有八位而$8086CPU$字长为十六位，所以还要向高位读一个字节，即为1123，高地址的在高位，所以11在23前面。

指令	寄存器内容变化	说明
MOV AX,1000H	AX=1000H	将基址 1000H 送到累加器中
MOV DS,AX	DS=1000H	将基址 1000H 赋值给数据段寄存器
MOV AX,[0]	AX=1123H	将偏移量为 0 的数据读取到 AX 中，多读一个[1]位置
MOV BX,[2]	BX=6622H	将偏移量为 2 的数据读取到 BX 中，多读一个[3]位置

指令	寄存器内容变化	说明
MOV CX,[1]	CX=2211H	将偏移量为 1 的数据读取到 CX 中，多读取一个[2]位置
ADD BX,[1]	BX=8833H	即将[1]位置的数据与 BX 数据相加，[1]位置的数据不能是半字，所以必须为十六位，即[2][1]=2211H，相加得到 6622H+2211H=8833H
ADD CX,[2]	CX=8833H	同理将[2]位置的数据扩展为[3][2]=6622H，相加 2211H+6622H=8833H

段操作

对于$8086CPU$可以将数据根据需要让一组内存单元定义为一个段，以后操作根据段来进行。

- 组的最大长度为$64K$，因为偏移量为$2^{16} = 64K$。
- 地址连续。
- 起始地址为16倍数。（即地址的最后一个数字为0，如$123B0H$）

将$123B0H \sim 123BAH$的内存单元定义为数据段，累加这个数据段中的前三个单元中的数据：

```
MOV AX,123BH
MOV DS,AX
MOV AL,0
ADD AL,[0]
ADD AL,[1]
ADD AL,[2]
```

栈操作

$8086CPU$的栈操作都是以字，即十六比特为单位进行。高地址单元放高八位，低地址单元放低八位。

其中SS指向栈顶的段地址，SP指向栈顶的偏移地址，所以$SS:SP$指向栈顶元素。

如将$10000H \sim 1000FH$作为栈，执行下面指令：

```
MOV AX,0123H
PUSH AX
MOV BX,2266H
PUSH BX
MOV CX,1122H
PUSH CX
POP AX
POP BX
POP CX
```

$10000H \sim 1000FH$作为栈，则栈底为高位$1000FH$，栈底为低位$10000H$，由于需要压入栈一共三个数据六个字节，所以只有$1000AH \sim 1000FH$被使用到。

指令	MOV AX,0123H	PUSH AX	MOV BX,2266H	PUSH BX	MOV CX,1122H	PUSH CX	POP AX	POP BX	POP CX
1000AH						22			
1000BH						11			
1000CH				66	66	66	66		
1000DH				22	22	22	22		
1000EH		23	23	23	23	23	23	23	
1000FH		01	01	01	01	01	01	01	

SP在入栈和出站是会加减二。

又将$10000H \sim 1000FH$这段空间当作栈，初始状态是空的。设置$AX = 001AH$，$BX = 001BH$。将AX、BX中的数据入栈。然后将AX、BX清零。最后从栈中恢复AX、BX原来的内容。

由于$10000H \sim 1000FH$为栈，栈底在高位，所以栈为空时栈顶应该初始化赋值为最大值$1000FH$的下一位$SP = 0010H$。

```
MOV AX,1000H
MOV SS,AX
MOV SP,0010H
MOV AX,001AH
MOV BX,001BH
PUSH AX
PUSH BX
SUB AX,AX
SUB BX,BX
POP BX
POP AX
```

由于$8086CPU$栈空间没有保护，所以可能存在栈顶越界。

栈段操作

可以将字段内存作为栈段，这是用户配置的，CPU 并不会把它当作栈，所以我们要让$SS{:}SP$指向自定义栈段，从而能让栈操作指令能访问我们自己定义的栈段。

由于栈段操作时只需要移动SP，所以SP的大小决定栈段的最大长度，所以最大长度为$2^{16} = 64K$。

如果将$10000H \sim 1FFFFH$这段空间作为栈段，初始空间为空，则$SS = 1000H$，求SP。

由于空间高位为$1FFFFH$作为栈底，所以如果为空，则栈顶应该在$1FFFFH$的更高一位$1FFFFH + 1 = 20000H$。但是给定了栈段空间$10000H \sim 1FFFFH$，所以不能取到$SS = 2000H$，否则越界。

所以向栈中填充一个元素，所以任意时刻，$SS:SP$指向栈顶，当栈中只有一个元素的时候，$SS = 1000H$，$SP = FFFEH$。

如果栈为空，就相当于栈中唯一的元素出栈，出栈后，$SP = SP + 2$。SP原来为$FFFEH$，SP=FFFEH+2=0，*所以，当栈为空的时候*，SS=1000H，SP=0$。

所以$CS:IP$指向哪里，CPU就将该段内存当作代码，$SS:IP$指向哪里，CPU就将该段内存当作栈。

程序定义

可以将汇编文件使用编译器编译为可执行文件。编写->编译->连接->执行。

如下面就是一个简单的汇编源程序：

```
ASSUME CS:codeseg
codeseg SEGMENT
START: MOV AX,0123H
       MOV BX,0456H
       ADD AX,BX
       ADD AX,AX
       MOV AX,4C00H
       INT 21H
codeseg ENDS
END START
```

定义段

- $SEGMENT$和$ENDS$是一对成对使用的伪指令，这是在写可被编译器编译的汇编程序时，必须要用到的一对伪指令。
- $SEGMENT$和$ENDS$的功能是定义一个段，$SEGMENT$说明一个段开始，$ENDS$说明一个段结束。（END可以视为$END\ SEGMENT$的简写）
- 一个段必须有一个名称来标识，使用格式为：段名 SEGMENT、段名 ENDS。
- 一个汇编程序是由多个段组成的，这些段被用来存放代码、数据或当作栈空间来使用。
- 一个有意义的汇编程序中至少要有一个段，这个段用来存放代码。
- 段名称最终会被编译、连接程序处理为一个段的段地址。

程序框架

*END*是一个汇编程序的结束标记，编译器在编译汇编程序的过程中，如果碰到了伪指令*END*，就结束对源程序的编译。

如果程序写完了，要在结尾处加上伪指令*END*。否则，编译器在编译程序时，无法知道程序在何处结束。

*END*除了通知编译器程序结束外，还可以通知编译器程序的入口在什么地方。由*START*后面加一个冒号，此后就是汇编指令，即基本的代码段。*ENDSTART*即指明入口在*START*位置。

假设

*ASSUME*表示为假设的伪代码，假设某一段寄存器与程序中的某个用*SEGMENT*和*ENDS*定义的段相关联，这样就不用再使用*MOV*指令将段通过通用寄存器移动到段寄存器，从而简化了代码。

如 `ASSUME CS:codeseg` 将*codeseg*代码段放到*CS*代码段寄存器中。

程序返回

我们的程序最先以汇编指令的形式存在源程序中，经编译、连接后转变为机器码，存储在可执行文件中。

而*DOS*是一个单任务操作系统，所以只能有一个程序运行，所以一个程序结束后，将*CPU*的控制枚交还给使它得以运行的程序，这个过程为程序返回。

返回应该在程序的末尾添加返回的程序段。

如代码的 `MOV AX,4C00H` 和 `INT 21H`。这两句话的意思是*DOS*系统功能调*INT*21*H*功能中的一种，表示带返回码结束用户程序。*INT*21*H*指令中，寄存器*AX*分为*AH*和*AL*两个部分，*AH*中存入指令码4*C*表示带返回码结束，*AL*为程序返回码。用这个指令返回是不需任何条件，还可顺便关闭打开后忘记关闭的文件，并返回寄存器*AL*的值。

AH	功能	调用参数
00	程序终止（同 INT 20H）	CS=程序段前缀。
01	键盘输入并回显	AL=输入字符。
02	显示输出	DL=输出字符。
03	异步通迅输入	AL=输入数据。
04	异步通迅输出	DL=输出数据

错误

- 语法错误：程序在编译时被编译器发现的错误，容易被发现。

- 逻辑错误：程序在编译时不能表现出来，在运行时发生的错误，不容易被发现。

编辑

打开 DOSBox，挂载后直接运行 `EDIT` 命令，打开编辑器，编写：

```
ASSUME CS:codeseg
codeseg SEGMENT
MOV AX,0123H
MOV BX,0456H
ADD AX,BX
ADD AX,AX
MOV AX,4C00H
INT 21H
codeseg ENDS
END
```

编写完成偶点击 File，选择 Save 然后输入文件名进行保存。然后点击 File 的 Exit 退出编辑。

编译

然后运行 `MASM`，然后需要输入源程序名。如果源程序文件不是以asm为扩展名的话，就要输入它的全名，比如$test.txt$。在输入源程序文件名的时候一定要指明它所在的路径，如 `D:\MASM\test.asm`（但是注意此时你已经挂载了$MASM$，所以必须使用 `dir` 查看当前文件路径，文件路径直接为$test.asm$）。如果文件就在当前路径下，只输入文件名就可以。

然后出现$Object\ filename$，默认是后面括号的内容，如果要修改就输入新名字。

然后是$Source\ listing$，表示列表文件，是编译器将源程序编译为目标文件的过程中产生的中间结果。

最后是$Cross-reference$，表示交叉引用文件，这个文件同列表文件一样，是编译器将源程序编译为目标文件过程中产生的中间结果。

也可以直接加上文件地址输入，如 `masm test.asm`。

连接

将生成的$TEST.OBJ$文件连接为可执行的$TEST.exe$文件。

控制台中输入$LINK$进行连接。如果目标文件不是以obj为扩展名的话，就要输入它的全名，同理要指明文件路径。也可以直接输入 `LINK TEST.OBJ`。

$Run\ file$即表示要生成的可执行文件的名字。$List\ file$映像文件是连接程序将目标文件连接为可执行文件过程中产生的中间结果。$Libraries$即库文件，包含了一些

可以调用的子程序，如果我们的程序中调用了某一个库文件中的子程序，就需要在连接的时候，将这个库文件和我们的自标文件连接到一起，生成可执行文件。

此时会提示一个警告：$waningL4021:nostacksegment$没有栈段，可以不用理会。得到一个 TEST.EXE。

运行

由于是可执行文件，所以可以直接在控制台中输入文件名进行执行。

操作系统是由多个功能模块组成的庞大、复杂的软件系统。任何通用的操作系统，都要提供一个称为$shell$外壳）的程序，用户（操作人员）使用这个程序来操作计算机系统工作。

DOS中有一个程序$command.com$，这个程序在DOS中称为命令解释器，也就是DOS系统的$shell$。

所以就是这个运行的$command$将汇编文件的程序加载进入内存，并设置CPU的$CS:IP$指向程序的第一条指令，即程序入口，从而使得程序得以运行。程序运行结束后返回$command$中，CPU继续运行$command$。

可以使用 DEBUG 程序名的方式对程序进行监控。使用 R 指令对内存进行查看。可以看到$CX=000F$，即汇编程序一共十五个字节。

高级使用

[BX]

类似于[0]表示内存单元的偏移地址，所以$[BX]$也表示一个内存单元，但是偏移地址在BX中。

(X)

(X)用于表示一个地址或寄存器中的内容。如$(AX)=0010H$，即寄存器AX中的内容为$0010H$。如$MOVAX,[2]$等价于$(AX)=((DS)*16+2)$，$ADDAX,2$等价于$(AX)=(AX)+2$。

其中定义$idata$表示常量值的占位，如$MOVAX,[idata]$代表$MOVAX,$[常量]。

LOOP

为循环指令，格式：LOOP 标号。CPU执行该指令时会进行两步操作：

1. $(CX)=(CX)-1$。
2. 判断CX中的值，里面保存循环次数，不为零则转至标号处执行程序，如果为零打破循环向下执行。

计算2^{12}：

```
ASSUME CS:CODE
CODE SEGMENT
MOV AX,2
MOV CX,11
S: ADD AX,AX
LOOP S
MOV AX,4C00H
INT 21H
CODE ENDS
END
```

这里的标号S就指向的是 ADD AX,AX 指令。循环执行的语句要在标号和$LOOP$指令之间。

在实际编程中，经常会遇到，用同一种方法处理地址连续的内存单元中的数据的问题。 我们需要用循环来解决这类问题，同时我们必须能够在每次循环的时候按照同一种方法来改变要访问的内存单元的地址。这时，我们就不能用常量来给出内存单元的地址（比如[0]、[1]是常量），而应用变量。MOV AL,[BX]中的BX就可以看作一个代表内存单元地址的变量，我们可以不写新的指令，仅通过改变 bx 中的数值，改变指令访问的内存单元。

默认地址的段前缀都是放在DS中，可以设置不同的段寄存器从而设置不同的段前缀来简化。

多段程序

一个程序不可以只有一个代码段，所以必然存在多个段进行不同的数据操作。

由于END和$START$是一组关键字，END可以指明程序。

```
ASSUME 段寄存器名:段名
段名 SEGMENT
……
定义数据
……
START:
……
数据代码
……
段名 ENDS
END START
```