408计算机组成原理导学和概论

导学

2023年更新

增加部分

- 计算机性能指标增加了EFLOPS和ZFLOPS
- 微指令格式

改动部分

- “微指令编码方式”改为”微命令的编码方式”

删除部分

- 计算机发展历程，BCD码，校验码
- 总线仲裁
- 集中仲裁方式，分布仲裁方式
- 光盘存储器

课程内容

1. 计算机系统概论
2. 数据的表示和运算
3. 存储器层次结构
4. 指令系统
5. 中央处理器
6. 总线系统
7. 输入输出系统

重点：2，3，4，5
次重点：6，7

概论 > 基本组成

计算机系统由软硬件共同构成。

计算机的4个基本功能包括：数据的加工、保存、传送，操作控制。

电子计算机分为：电子模拟计算机（数值由连续量表示，计算过程连续），电子数字计算机（用数目字表示数量大小，按位运算，不连续地跳动计算）。

数字计算机分为：专用计算机，通用计算机。

通用计算机分类，体积、功耗、性能、数据存储量、指令系统、价格、**复杂性**由低到高：
单片机，微型机，工作站，服务器，大型机，超级计算机

早期冯诺依曼机

冯诺依曼提出了“**存储程序**”的概念。
将指令实现以二进制代码输入主存储器，随后在存储器中的首地址执行第一条，规定程序按地址顺序地执行其它指令，直到程序执行结束。

冯诺依曼机特点：

1. 计算机由**运算器、控制器、存储器、输入、输出**组成
2. 指令和数据以同等地位放入存储器，可按地址寻访
3. 指令和数据用二进制数表示
4. 指令由操作码和地址码组成，操作码用来表示操作性质，地址码用来表示操作数在存储器重点位置
5. 指令在存储器内按顺序存放
6. 机器以运算器为中心，输入输出设备与存储器间数据通过运算器完成。

现代计算机的组织结构

控制器ALU、控制器CU构成CPU，主存、辅存构成存储器，
CPU、主存构成主机，主机、辅存、I/O设备共同组成一台计算机硬件。

计算机的功能组件

I/O设备

（输入设备、输出设备）让计算机能够与外界联系

存储器

存放程序和数据，分为主存储器（简称主存，包括缓存、内存）和辅助存储器（辅存，也称外存）。

CPU能够直接访问主存，辅存必须讲信息调入主存，才能被CPU访问。

存储器特点：

- 主存包括存储体、地址MAR、数据MDR，数据在存储体内存地址存储，MAR位数反应存储单元个数，MDR位数=存储字长
  - MAR = 4位：总共有2^4个存储单元
  - MDR = 16位：每个存储单元可存放16bit
  - 1个字word = 16bit
  - 1个字节Byte = 1B = 8bit = 8b
- 由存储体、存储单元、存储元件(0/1)组成（大楼-房间-床位(无人/有人)）
- 存储单元存放一串二进制代码
- 存储字是存储单元的二进制代码组合
- 存储字长是存储单元中二进制代码的位数，每个存储单元赋予一个地址号
- 按地址寻访

2^12	2^11	2^10	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1
4096	2048	1024	512	256	128	64	32	16	8	4	2

存储容量：

1K = 2^10, 1M = 2^20, 1G = 2^30
1TB = 1024GB
1PB = 1024TB
EB，ZB，YB，BB依此类推1024数量级换算

运算器ALU

运算器的核心是算术逻辑存储单元，用于暂存操作数和中间结果，**累加器ACC、乘商寄存器MQ、操作数寄存器X**，变址寄存器IX、基址寄存器BR等，前三个寄存器是必须的

程序状态寄存器PSW（标志寄存器），存放ALU运算得到的一些标志信息或处理及的状态信息，包括结果是否溢出、有无产生仅为或错位、结果是否为负等。

X -> ALU <-> ACC <-> MQ

1. ACC：累加器，用于存放操作数，或运算结果
2. MQ：乘商寄存器，乘除运算时，存放操作数和运算结果
3. X：通用的操作数寄存器，用于存放操作数
4. ALU：算术逻辑单元，通过内部复杂电路实现算数运算、逻辑运算

_	**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、乘积高位	商
X	加数	减数	被乘数	余数

控制器

向计算机提供每一时刻协同运行所需的控制信号：

1. 正确分析与执行每条指令：取指令 -> 分析指令 -> 执行指令（指令周期）
2. 保证指令按规定序列自动连续地执行
3. 对各种异常情况和请求及时响应和处理

控制器由PC、IR、CU组成：
PC存放当前欲执行行指令的地址，具有计数功能：(PC) + 1 -> PC；
IR存放当前欲执行指令；
CU执行指令。

概论 > 计算机软件

软件分为**系统软件**（操作系统、标准程序库、语言处理、服务程序、数据库管理系统、软件）、**应用软件**（面向用户根据特殊需求编制的应用程序）。

软件与硬件具有**逻辑等价性。**任何操作、指令都可以由软件或硬件实现。

计算机的工作过程

1. 把程序和数据装入主存储器；

2. 将源程序转换成可执行文件；
3. 从可执行文件的首地址开始逐条执行指令。

计算机系统层次结构

软件:
虚拟机器M4（高级语言机器，用编译程序返程汇编语言程序）
↓
虚拟机M3（汇编语言及其，用汇编程序翻译成机器语言程序）
↓
虚拟机M2（用机器语言解释操作系统）
↓
硬件:
实际机器M1（用机器语言的机器，用微指令解释机器指令）
↓
微程序机器M0（由硬件直接执行微指令）

计算机性能指标

存储器：总容量 = 存储单元个数 × 存储字长bit
= 存储单元个数 × 存储字长/8bit
1Byte = 8bit

存储器容量：存储器中所有存储单元的总数目，单位：KB、MB、GB、TB

吞吐量：计算机在某一时间间隔内能够处理的信息量，单位：字节/秒（B/S）

响应时间：有效输入到系统产生响应之间的时间，单位：秒

利用率：给定时间间隔内，系统被实际使用的事件所占比率，用%表示

处理机字长：处理机运算器中一次能完成的二进制位数，字长越长，精度越高

总线宽度：CPU中运算器与存储器间进行互联的总线二进制位数

存储器带宽：单位时间内从存储器读出的二进制数信息量，单位：字节/秒（B/S）

主频：主时钟的频率f，主时钟不断产生固定频率的时钟，CPU工作节拍受主时钟控制，单位：MHz、GHz

时钟周期：主频的倒数是时钟周期T，`T = 1/f`，单位：微秒，纳秒

CPU执行时间：CPU执行一段程序所占用CPU的时间 `tcpu`，
`CPU执行时间 = CPU时钟周期数 × CPU时钟周期`
`tcpu = Nc × T`

CPI：每条指令的平均周期数，执行一条指令所需的平均始终周期数，
`CPI = Nc / In` // In是指令的数量
`执行一条指令的耗时 = CPI × CPU时钟周期`
`CPU执行时间 = CPU时钟周期数 / 主频 = (指令条数 × CPI) / 主频`

IPS：每秒指令条数，`IPS = 主频 / 平均CPI`

FLOPS：每秒执行的浮点运算次数

MIPS：每秒执行百万条的指令数，`MIPS = 指令条数 / (程序执行时间 × 10^6)`

MFLOPS：每秒百万次浮点操作次数，`MFLOPS = 程序中浮点操作次数 / (程序执行时间 × 10^6)`

补充知识点

速度比较：寄存器 > Cache > 内存

MAR的位数决定了地址码长度，MDR位数（存储单元的二进制位数）决定了存储字长

8位的计算机系统以16位表示地址，意味着处理机字长8位、MAR有16位，它有`2^16 = 65536`个地址空间

CPU是依据指令周期的不同阶段，来区分存储器中的指令和数据

计算机硬件能够直接执行的只有机器语言
汇编语言还需要翻译为机器语言才能被硬件执行
硬件描述语言用于设计大规模电路

指令和数据都用二进制表示，形式上无差别，所以需要指令周期来区分。数据不一定在指令中直接给出。

位数一定与机器字长相同的是ALU、通用寄存器

1G = 10^9, 1M = 10^6

数据的表示与运算

考纲要求：

1. 数制与编码：
    1. 进位计数制机器数据之间的相互转换
    2. 定点数的编码表示
2. 运算方法和运算电路
    1. 基本运算部件：加法器，算数逻辑部件ALU
    2. 加减法运算：补码加/减运算器，标志位的生成
3. 整数表示和运算
    1. 无符号整数的表示和运算
    2. 带符号整数的表示和运算
4. 浮点数的表示和运算
    1. 浮点数的表示：IEEE754标准
    2. 浮点数的加减运算

进位计数制

二进制B、八进制Q（或O）、十进制D、十六进制H（或0x）

r进制数D转为十进制数N：
`N = d(n) × r^(n-1) + d(n-1) × r^(n-2) + ... + d(1) × r^0 + d(-1) × r^(-1) + ... + d(-m) × r^(-m)`,
d(n)表示D对应位数的数值

二进制转八进制、十六进制：
每3位二进制，相当于1位八进制；
每4位二进制，相当于1位十六进制；
三十二进制、六十四进制依此类推

十进制转二进制（通用方法）：
整数部分÷2取余，余数逆序（从下往上）位数递减；
小数部分×2取整，整数顺序（从上往下）位数递减

十进制转二进制（拼凑法）：
根据`2^n`依次减去对应的二进制数的十进制数，余下的差依此类推，各个n则为1所在的二进制位数

真值和机器数

真值：±和某进制数绝对值的形式成为真值；
机器数：符号数码化的数为机器数，例如0/1表示+/-

计算机使用的数据可分为符号数据（ASCII码、汉字、图形等）、数值数据（定点、浮点）。

数据格式

定点数:
约定机器中所有数据的小数点位置固定不变，由于约定在固定位置，小数点不再用“.”表示。
定点表示数的范围受字长限制，且精度有限

1. 定点纯小数：符号+整数部分+小数部分
2. 定点纯整数：符号+数值

浮点数:
小数点的位置随阶码的不同而浮动，表示方式：`N=R^E·M`，例如233*10^3，`(1.75)10 = 1.11*2^0 = 0.111*2^1 = 0.0111*2^2`
IEEE754规定了单精度float（32）和双精度double（64），由
`【32】数符s(1位)+阶码E（8位,移码）+尾数M(23位,原码)`或
`【64】数符s(1位)+阶码E（11位,移码）+尾数M(52位,原码)`
构成， 它们由二进制转为十进制的公式为：
`【32】D = (-1)^s × (1.M) × 2^(E-127)`
`【64】D = (-1)^s × (1.M) × 2^(E-1023)`
例如`(41360000)16`求十进制的过程为：

1. 转为二进制数：`0 1000 0010 011 0110 0000 0000 0000 0000`;
2. e = 阶码-127 = 100000010 - 01111111 = 00000011 = (3)10;
3. 包含隐藏位1的尾数：`1.M = 1.01101100000000000000000 = 1.011011`;
4. 由2、3得：(-1)^s × 1.M × 2^e = +(1.011011)×2^3 = +1.011.011 = (11.375)10

数的机器码表示

源码表示法:
用第一位表示±，其余部分为二进制表示的数值本身;
简单易表示，乘除运算简单，加减运算麻烦。
定点小数`x0.x1x2x3...xn`，例如：
x = +0.11011，x源码 = 0.11011
y = -0.11001，y源码 = 1.11001
定点整数`x0x1x2x3...xn`，例如：
x = +11011，x源码 = 011011
y = -11001，y源码 = 111001

补码表示法:
正数的补码和源码相同，负数的补码是正数的反码每一位取反 并在末位加1;
将加法运算转换为加法运算：`[x]补 - [y]补 = [x]补 + [-y]补`;
没有正零和负零之分。
定点小数`x0.x1x2x3...xn`以`2`为模
定点整数`x0x1x2x3...xn`以`2^(n+2)`为模
【注：计算时最高1位若超过字长，需丢掉】
变形补码:
双符号补码，为了防止溢出

反码表示法:
正数的反码和源码相同，负数的反码是对应正数的源码每一位取反;
电路容易实现，触发器的输出有正负之分。

移码表示法:
通常用在阶码种，定点整数定义为`[x]移 = 2^n + x, 2^n > x ≥ -2^n`;
只能用于整数。

字符和字符串（非数值）的表示方式

编码的各个字符通常以`00H`（也就是`\0`）作为结尾

ASCII码： 用一个字节表示，低七位用于编码(128)，最高位为校验位。
字符串占用主存中连续多个字节，每个字节存储一个字符。

- 0：48，30H
- A：65，41H
- a：97，61H

汉字编码:
GB2312-80：汉字+各种符号共7445个。

C语言中的整数类型转换

有符号数和无符号数的转换:
强制类型转换的结果保持位置不变，仅改变解释这些位的方式

```
int mian(){

   /*有符号转无符号*/
   short x = -4321;
   unsigned short y = (unsigned short)x;
   printf("x=%d, y=%u\n", x, y);   //x=-4321, y=61215

   /*无符号转有符号*/
   unsigned short x = 65535;
   short y = (short)x;
   printf("x=%u, y=%d\n", x, y);    //x=65535, y=1

}
```

不同字长的整数转换:

```
int main(){

   /*长转换为短*/
   int x = 165537, u = -34991;
   short y = (short)x, v = (short)u;
   printf("x=%d, y=%d\n", x, y);
   printf("u=%d, v=%d\n", u, v);
   //x=165537, y=-31071
   //u=-34991, v=30545
```

```
    /*短转换为长*/
    short x=-4321;
    int y=x;
    unsigned short u = (unsigned short)x;
    unsigned int v = u;
    printf("x=%d, y=%d\n", x, y);
    printf("u=%u, v=%u\n", u, v);
    //x=-4321(0xed1f), y=-4321(0xffffef1f)
    //u=61215(0xef1f), v=61215(0x0000ef1f)

}
```

32位系统下的各数据类型长度

数据类型	占据内存大小(byte)
short	2
int	4
long	4
float	4
double	8
char	1
	从大字长向小字长强制类型转换时，会把多余的高位字长截断，低位直接赋值。
	短字长整数到长字长整数转换，不仅要使相应的位值相等，高位部分还会扩展为原数字的符号位
	char类型转为8位ASCII码整数int时，在高位部分补0即可

短数据转为长数据:
正整数整数在高位扩展，小数在低位扩展；

负数的短数据转为长数据:

	负整数	小数
源码	原最高位挪到转换后的最高位，其余高位补0	低位补0
反码	高位补1	低位补1
补码	高位补1	低位补0

数据的存储和排列

小端方式 和 大端方式 存储:
较低的有效字节存放在较低的存储器地址（地址偏移越多的越低），较高的字节存放在较高的存储器地址。

大端存储反之。
例如存储(0x12345678)到int，OP0表示32位数据最高字节MSB，OP3表示32位数据最低字节LSB：

地址偏移	大端模式	小端模式
0x00	12(OP0)	78(OP3)
0x01	34(OP1)	56(OP2)
0x02	56(OP2)	34(OP1)
0x03	78(OP3)	12(OP0)

边界对齐 存储:
通常32位的计算机，可按照字节、半字和字寻址。
数据以边界对齐的方式存储，半字地址一定是2的整数倍，字地址一定是4的整数倍。
这样数据可以一次访存取出。
对于不满足的数据则填充空白字节。

定点加减法运算

先判断两数符号，同号相加、异号相减，绝对值大的减去小的，最后确定符号位

补码加减法

补码加法:
[x+y]补 = [x]补 + [y]补 mod 2 或 mod 2^(n+1)。
补码减法:
[x-y]补 = [x]补 - [y]补 = [x]补 + [-y]补。

从[y]补求[-y]补:
[-y]补 = ¬[y]补 + 2^(-n)，
¬表示对[y]补作包括符号位内的求反操作，
2^(-n)表示最末位的1。
负数求补码口诀:
从右向左，第一个1和第一个0保持不变，其它按位取反。

溢出:
定点小数中，数的范围是|x| < 1，如果大于1则为“溢出”，在定点机种通常不允许。
例如：俩正数相加变负数、俩负数相加变正数。
检测方法:
两个符号位看作数码参加运算；两数以4 或 2^(n+2)为模的加法，最高位符号位产生的进位需要舍弃；采用变形补码后，两数相加其结果符号位出现01或10组合是，表示溢出。
电路加法器中，当Cn = C(n-1)，运算无溢出；当Cn ≠ C(n-1)，运算溢出。

进位标志的生成：条件码:
标志寄存器由16位的存放条件标志、控制标志寄存器，用于反映处理器状态和ALU运算结果的某些特征以及控制指令的执行。
条件码:

- 溢出标志 OverflowFlag OF：==1时溢出

- 符号标志 SignFlag SF：==0表示正数，==1表示负数
- 零标志 ZeroFlag ZF：==1表示结果为0
- 进位/错位标志 CarryFlag CF：==1表示无符号数的加减法发生了错位/进位，即将发生溢出

浮点运算

浮点数加减

存在两个浮点数`x=2^(Ex)Mx`和`y=2^(Ey)My`，Ex、Ey是阶码，Mx、My是尾数，则它们相加减的规则是：`x±y = (Mx·2^(Ex-Ey)±My)·2^(Ey)`（设Ex≤Ey）

运算步骤

1. 对阶：求阶差的绝对值`▲E = |Ex - Ey|`，当其不等于0时，小阶向大阶看齐——阶码较小的尾数右移▲E位，其阶码值+▲E（每右移一位都要+1）。对原码尾数，符号位不参加唯一，尾数高位补0；对补码尾数，尾数高位补符号位：
    - 逻辑左/右移：直接移动，低位/高位补0，高位/低位丢弃。
    - 算数左/右移：高位不变，其余位同逻辑移动。
2. 尾数加减：完成对阶后，对尾数求和/差；
3. 规格化：对原码，最高数值位为1；对补码尾数，必须是`00.1xxx...`或`11.0xxx...`。补码规格化规则：
    - 若尾数符号位不同（尾数计算结果溢出），应当使结果右移一位，并使阶码值+1（向右规格化，简称右归）。
    - 若尾数双符号位相同（不溢出），且最高位数值位与符号位相同，表示不满足规格化规则，此时应重复使尾数左移、阶-1，直到出现最高位数值与符号位数值不同为止（向左规格化，简称左归）。
4. 舍入：要求有舍有入，尽量使舍和入的机会均等（防止误差的积累）。常用方法是"0舍1入"法：
    - 就近摄入：类似四舍五入，丢弃最高位为1，进1。
    - 朝0舍入：截尾。
    - 朝+∞舍入：正数多余位不全为0，进1；负数截尾。
    - 朝-∞舍入：负数多余位不全为0，进1；正数截尾。
5. 判溢出：浮点数的溢出以**阶码溢出**来表现出来的：
    - 若阶码正常，加/减法运算正常结束。
    - 若阶码下溢，要置运算结果为浮点形式的机器0。
    - 若阶码上溢，则置溢出标志。

补充

综合题01

按字节编址的计算器存储器，用小端方式存储。编译器规定int为32位，short为16位，数据按边界对齐存储，有一个从语言段如下：

```
struct{
   int a;
   char b;
   short c;
}record;
record.a=273;
```

若record变量的首地址位0xC008，则其中的内容的地址为0x11，record.c的地址为0xC00E

a	a	a	a
b	-	c	c

补码计算

字长8位计算机，`x补 = 1 111 0100`，`y补 = 1 011 0000`，求`z=2x+y/2`，则直接对补码进行计算：`2x = 1101000，y/2=11011000`，相加即可得到机器码`1100000`

存储器层次结构

考纲要求

1. 存储的分类
2. 层次化存储器的层次化结构
3. 班都提随机存取存储器
    1. SRAM存储器
    2. DRAM存储器
    3. Flash存储器
4. 主存储器
    1. DRAM芯片和内存条
    2. 多模块存储器
    3. 主存和CPU之间的连接
5. 外部存储器
    1. 磁盘存储器
    2. 固态硬盘SSD
6. 高速缓冲存储器Cache
    1. 基本原理
    2. 和主存之间的映射方式
    3. 主存块的替换算法
    4. 写策略

存储器分类

1. 按存储介质分类

    1. 半导体存储器（易失）：TTL、MOS
    2. 磁表面存储器（非易失）：
        1. 磁表面存储器：磁头、磁载体
        2. 磁芯存储器：硬磁材料、环状原件
        3. 光盘存储器：激光、磁光材料

2. 按存取方式分类

    1. 存期时间与物理地址无关（随机访问）
        - 随机存储器：程序执行过程中可读可写
        - 只读存储器：程序执行过程中只读
    2. 存取时间与物理地址有关（串行访问）
        - 顺序存取存储器（磁带）
        - 直接存取存储器（磁盘）

3. 按读写功能分类

    - ROM
        - MROM，PROM，EPROM，EEPROM
    - RAM

        - 静态RAM，动态RAM

4. 按信息的可保存性分类
    - 永久性，非永久性
5. 按存储器系统种的作用分类
    - 主存，辅存，缓存，控

存储器分级

速度快的，价格贵、容量小；反之便宜、容量大。

CPU <-> 缓存 <=> 主存 <=> 辅存

主存储器技术指标

- 存放一个机器字的存储单元，是字存储单元，对应的单元地址叫字地址
- 存放一个字节的存储单元，是字节存储单元，对应的单元地址叫字节地址
- 可编址的最小单位：按字寻址、按字节寻址
- 存储容量：一个存储器种可容纳的存储单元数量
- 存取时间（访问时间）：一次读操作命令发出到该操作完成，并将数据读出到数据总线上所经历的时间。通常取写操作时间等于读操作时间，称为存储器存取时间
- 存取周期：连续两次读操作所需的最小间隔时间。通常存取周期略大于存取时间
- 存储器带宽：单位时间内存储器能够存取的信息量

主存储器SRAM和DRAM比较：

类型特点	**SRAM**	**DRAM**
存储信息	触发器	电容
破坏性读出	非	是
读出后是否需要重写	是	否
运行速度	快	慢
集成度	低	高
发热量	低	高
存储成本	高	低
易失/非易失	易失（断电后信息消失）	易失（断电后信息消失）
是否需要刷新	不需要	需要
送行列地址	同时送	分两次送
-	常用于Cache	常用于主存

SRAM存储器

静态读写存储器SRAM的存取速度块。而DRAM则是存储容量大。
SRAM用于cache，DRAM用于主存。

任何一个内部存储器都有三组信号线：ADC（A：地址线，D：数据线，C：控制线）。
地址线A -> | 译码驱动 存储矩阵 读写电路 | <-> 数据线D
控制线C负责片选CS和读写R/W

n位地址对应2^n个存储单元。

SRAM逻辑结构：
大部分SRAM采用双译码方式，采用二级译码：将地址分为x向、y向。

DRAM存储器

DRAM用于主存。每个存储单元都能够实现写1、写0、读出1、刷新存储位元的1。

为了实现行列分时传送，需要行/列地址锁存器。

读/写周期

读/写周期的定义是从行选通信号RAS下降沿开始，到下一个RAS信号的下降沿为止（即连续两个读周期的时间间隔）——为了控制方便，读、写周期的时间相等。

刷新周期

在电荷漏掉前充电，保证存储信息不被破坏——这一充电过程称为再生或刷新。
DRAM一般≤2ms会刷新一次，采用"读出"方式、按行刷新，包括有：

1. 集中刷新
    - 读/写或维持结束后，再集中执行刷新，存取周期0.5μs；
    - 以128x128矩阵为例：
        - 死区：`0.5μs × 128 = 64μs`；
        - 死亡时间率：`128/4000 × 100% = 3.2%`。
2. 分散刷新
    - 存取周期为1μs，无死区；
    - 以128x128矩阵为例：`tc = tm + tr`，即0.5μs+0.5μs。
3. 异步刷新
    - 分散刷新与集中刷新结合；
    - 对于128x128的存储芯片，存取周期为0.5μs，则每间隔15.625μs刷新一行（通常取15.5μs）；
    - 每行间隔2ms刷新一次，死区0.5μs；
    - 若将刷新安排在指令译码阶段，则不会出现死区。

存储器容量库充

位扩展、字扩展

字长位数扩展（位扩展）：
用多片给定芯片扩展字长位数。
地址线和控制线公用，而数据线D单独分开连接。
所需芯片数 = 设计要求的存储容量 ÷ 选择芯片存储器容量

字存储容量扩展（字扩展）：
三组信号组中，给定芯片的地址总线和数据总线公用，控制总线C中R/W公用，使能端CS不公用，它由地址总线的高位段译码来决定片选信号。
所需芯片数 = 设计要求的存储器容量 ÷ 选择瞎拍存储器容量

字位扩展：
实际应用中往往需要同时扩展。
若存储器容量为M×N位，使用L×K位存储器芯片扩充，则总共需要：
(M/L) × (N/K)个L×K位存储器芯片。可以先位扩展，再字扩展。

只读存储器ROM和闪存存储器

ROM和RAM都支持随机存取，SRAM和DRAM均为易失性半导体存储器。ROM则即使断电也不会丢失。ROM结构简单，位密度高，非易失性，可靠性高。

ROM

1. 掩模式只读存储器MROM：
    - 存储内容固定的ROM：行列选择交叉处有MOS管为1，无MOS管为0。
2. 一次可编程只读存储器PROM：
    - 一次性编程：单个原件中，熔丝断为0、熔丝未断为1。
3. 可擦除可编程只读存储器EPROM：
    - 光擦除可编程可读存储器，多次性编程；
    - 需要更新市江源存储内容抹去，允许多次重写；
    - 但写入时间有限，不能对个别单元单独擦除或重写，擦写时间较长。
4. EEPROM:
    - 电擦除可编程存储器；
    - 出厂时存储内容全为1，使用时可根据需求吧某些存储写0。

FLASH

Flash存储元是在EPROM和EEPROM存储元的基础上发展的。

1. 闪速存储器FlashMemory：
    - 高密度非易失性的读写存储器，存储容量大；
    - 三种操作:
        1. 编程操作：实际上是写操作；
        2. 读取操作；
        3. 擦除操作：使全部存储单元变为1状态；
2. 固态硬盘SSD：
    - 由控制单元+存储单元(Flash)构成；
    - 可进行多次快速擦除重写；

- 速度快、功耗低、价格高，目前逐渐取代机械硬盘；
- 组成：
    1. 闪存翻译层：翻译逻辑块号，找到对应页（page）；
    2. 存储介质：多个闪存芯片，每个芯片包含多个块，每个块包含多个页（page）。
- 读写特性：
    - 以页为单位读写——相当于磁盘的“扇区”
    - 以块为单位擦除
    - 支持随机访问
    - 读取快，写入较慢；如果要写的页有数据，则无法写入，需要将块内其他页全部复制到新的块中，再写入新的页
- 与机械硬盘相比：
    - 读写速度快，随机访问性能高，无需通过移动磁旋臂控制到访问位置
    - 安静无噪音，耐摔抗震，耗能低，造价贵
    - 多次擦写一个块可能会损坏
- 磨损均衡技术：
    - 将擦除的任务均分在每个块上，提升使用寿命
    - 动态磨损均衡：写入时有限选择累计擦除次数少的新的块
    - 静态磨损均衡：SSD检测并自动进行数据分配、迁移，让老旧闪存块承担以读为主的存储任务，让较新闪存块承担更多写任务

并行存储器

为了提高CPU和主存间的数据传输率，除了缓存外，还可以使用并行存储器。
包括有：

- 空间并行技术：双端口存储器
- 时间并行技术：多体交叉存储器

双端口存储器

同一个存储器用两组相互独立的读写控制电路。

两个端口对同一主存操作有以下情况：

1. 同时对不同地址单元存储数据。
2. 对同一地址单元读出数据。
3. 同时对同一地址单元写入数据（有冲突）。
4. 同时对同一地址单元，一个写入、另一个读出（有冲突）。

解决方法：

设置BUSY标志，置BUSY信号为0，由判断逻辑界定暂时关闭一个端口（被延时），未被关闭的端口正常访问，被关闭端口延长一个很短的时间段后再访问。
有冲突读写控制判断方法：

1. CE判断：若地址匹配再CE之前有效，片上的控制逻辑在CEL和CER之间判断选择端口。
2. 地址有效判断：若CE再地址匹配前遍地，片上的控制逻辑在左、右地址之间进行判断来选择端口。

多模块交叉模拟器

主存被分为多个相互独立、容量相同的模块M0、M1、M2、M3...，每个模块都有自己的读写控制电路、地址寄存器、数据寄存器，个自己等同方式与CPU传送信息。
理想状态下（程序段或数据块都是连续地在贮存中存取）将大幅提高主存访问速度。

高位选模块，低位选块内地址：
当某一模块故障，其他模块可以照常工作。但由于各个模块串行工作，带宽受限。

高位选块内地址，低位选模块：
连续地址分布在相邻的不同模块内（同一模块内的地址都不连续）。对连续字的成块传送可实现多模块流水式并行存取，提高带宽。

Cache

Cache采用高速SRAM组成，解决CPU和主存的速度不匹配的问题，全由硬件调度，对用户透明。

CPU与Cache之间的数据传送与字为单位，主存与Cache之间则是块为单位。
CPU读取主存使，把地址同时传送给Cache和主存：若判断地址在Cache中，则直接传送给CPU；否则用主存读周期把字送到CPU。同时把整个数据块从主存送到Cache。

为什么需要Cache

程序访问局部性：
对局部范围存储器地址频繁访问，而对此范围以外的地址则较少访问的现象称为“程序访问的局部性”。
空间局部性：在较近的未来要用道德信息（指令、数据），很可能与正在使用的信息在存储空间上是临近的。
时间局部性：在较近的未来要用到的信息很可能是现在正在使用的信息。

Cache性能指标

命中率h：
在性能上使主存的平均读出时间尽可能接近Cache的读出时间，
Nc为Cache中的访问时间，Nm为在主存中的访问时间 `h = Nc / (Nc + Nm)`

Cache-主存系统的平均访问时间ta：
tc为命中时Cache访问时间，tm为命中时主存访问时间，则：
`ta = h*tc + (1-h)tm`

效率e：
效率和命中率有关，
`e = 访问Cache / 平均访问时间 * 100%`
h为Cache命中率，tc为Cache访问时间，则：
`e = tc / (h*tc+(1-h)*tm) * 100%`

主存与Cache的地址映射

Cache容量很小，保存的只是主存中的一个子集。Cache与主存的数据交换以块为单位。
地址映射则是将主存地址定位于Cache中。
映射时需要将主存和Cache划分围同样大小的块。

全相连的映射方式：
主存块放在Cache的任意位置，是一种最灵活但成本最高的方式。

Cache完全满了才会替换，需要在全局选择替换对应块。

直接映射方式：
映射方式一对多：`i = j mod m`。
若对应位置非空，则毫无选择地直接替换。

组相连映射方式：
全相连和直接映射的结合：`q = j mod u`，主存第j块的内容靠背到Cache的q组某行。
分组内满了才需要替换，需要在分组内选择替换对应块。

Cache替换策略

从主存中调入新的字块，若其位置一杯其它字块占有，则需要替换旧字块。
最优情况是使被替换的字块是下一段时间内估计最少使用的。

常见替换算法有：先进先出FIFO，近期最少使用LRU，最不经常使用LFU，随机替换。

先进先出FIFO：
把最先调入Cache的字块替换，不记录各字块使用情况，实现容易，开销小。

近期最少使用LRU：
被访问的行数计数器置0，其它行的计数器+1。近期最少使用的字块被替换（计数器值最大的行）。符合Cache工作原理。

最不经常使用LFU：
被访问的行进行计数器+1，同一时间访问次数最少的数据被替换。不能访问近期Cache的访问情况

随机替换：
随机选取一行换出。

Cache写操作策略

- **写命中**
  - *全写法*
    - 数据必须同时写入Cache和主存，一般使用写缓冲。访问次数增加，速度慢，但更保证数据一致。
  - *写回法*
    - 只修改Cache内容，不立即写入主存，只有被换出才写回主存。减少了访问次数，但存在数据不一致的隐患。
- **写不命中**
  - *写分配法*
    - 把主存快调入Cache，在Cache中修改。通常搭配写回法。
  - *非写分配法*
    - 只写入主存，不调入Cache。通常搭配全写法，只有读未命中时调入Cache。

虚拟存储器

将主存和辅存的地址空间统一编址，合成一个更大的地址空间。

虚存的替换算法和Cache类似，也有先进先出FIFO，近期最少使用LRU，最不经常使用LFU，随机替换等。

- 虚地址：用户编程时用的地址称为虚拟地址和逻辑地址，对应的存储空间称为虚存空间或逻辑地址空间。
- 实地址：物理内存访问的地址，其对应存储空间是物理存储空间或主存空间。
- 程序的再定位：程序进行虚地址到实地址转换的过程。
- 虚存的访问过程：由地址变换机构依据当时分配给程序的实地址空间，把程序的一部分调入实存。每次访问时都将判断虚地址对应部分是否在实存中：是，则地址转换并用实地址访问主存；否，则按指定算法将辅存中的部分程序调度进内存，再按同样方式访问主存。
- Cache与虚存的异同：
    - 主存-辅存的访问机制和Cache-主存访问机制类似。这是由Cache、主存、辅存构成的三级存储体系中的两个层次。
    - Cache-主存构成了系统内存；主存-辅存依靠软硬件的支持构成了虚拟存储器

页式虚拟存储器

页式虚存地址映射：
虚地址空间和主存空间分别分为等长大小的页（逻辑页和物理页）。
虚地址分为两个字段：高字段为逻辑页号，低字段为页内地址（偏移量）；
实存地址分为两个字段：高字段为物理页号，地字段为页内地址。
通过页表把虚地址转为物理地址。

页式虚拟存储器的地址映射过程：
每个进程对应一个页表，表项的内容包含该虚存页面所在的主存页面的物理页号。页表的基地址存在寄存器，页表本身放在主存。

快/慢表：
页表通常在主存，即使逻辑页在主存，也至少访问两次物理存储器才实现一次访存。
可对页表本身实行二级缓存，把页表中最活跃的部分放在高速存储器，组成**快表**。
专用于页表缓存的高速存储部件通常称为转换后援缓冲器TLB。存在主存里完整的页表则是**慢表**。

段式虚拟存储器

段是按照程序的自然分界划分的长度可以动态改变的区域。
通常把子程序、操作数、常数等不同类型数据分到不同的段，每个程序可以有多个相同类型的段。

虚地址由段号和段内地址（偏移量）组成。虚地址到实地址主存地址的变换通过段表实现。

每个程序设置一个段表，每一个表项对应一个段。每个表项包含：有效位，段起止，段长。

段页式虚拟存储器

页式和段式的结合。
实存被分为页，每个程序先按逻辑结构分段，每段再按实存的页大小来分页，程序按页进行调入调出操作，但可按段进行编程、保护、共享。

错题补充

- 不能采取随机存取的存储器：CD-ROM

- 存取周期为200ns的64K×32位的四体并行低位交叉存储器，在200ns内，存储器能够向CPU提供128位的二进制信息（每个模块提供32位）

- 对于四体低位交叉存储器，读取6个连续地址单元中的存储字，需要2个完整的存储周期（同理读取3个连续地址单元的，需要1个完整的周期）

- 4M×8位的DRAM芯片，其地址引脚数是11、数据引脚数是8。
  DRAM的特点是地址复用，所以在计算地址引脚数时需要÷2（对于4M，有`log2(4*1024*1024) = 22`，所以地址引脚数是11；
  数据引脚数就是该芯片的位数。

- 使用四体交叉编制存储器，存储器总线上出现的主存地址序列为：8005、8006、8007、8008、8001、8002、8003、8004、8000。可能发生访问冲突的是8000和8004。
  对各个地址做取余处理得到下表：

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

- 计算机主存按字节编址，由4个64M×8位的DRAM芯片采用交叉编址方式构成，并与宽度为32位的存储器总线相连，主存每次最多读写32位数据。若double型变量x的主存地址为804001AH，则读取x需要的存储周期数是3。
  x的起始位是A(1010)，所以对于该四体交叉编制存储器，x从第3个开始；而double占据8个字节。所以综上存储周期为3。

- 对于一个`8192*8192*8位`的DRAM芯片，芯片内缓冲有8192*8位。

- 用若干2K×4位的芯片组成一个8K×8位的存储器，则地址0BIFH所在的芯片最小地址是什么？
  8K = 2^13，所以要13根地址线；单个芯片2K需要11根地址线；
  所以前2位代表4组选片地址，后11位代表片内地址；
  所以对于`0BIDH = 000(0 1)(011 0001 1111)`，其所在的最小地址为`0000 1000 0000 0000 = 0800H`。

控制器

指令

指令（机器指令）是计算机执行某种操作的命令。
一条指令由**操作码OP**和**地址码A**构成。
一台计算机所有的指令集合构成该计算机的指令系统（指令集）。

字长：

- 指令字长：一条指令的总长度（可能会变）
  - 指令字长会直接影响取指令的时间
  - 按字长分类有：
    - 定长指令字结构
    - 变长指令字结构
- 机器字长：CPU进行一次整数运算所能处理的二进制数据的位数（通常和ALU直接相关）
- 存储字长：一个存储单元中的二进制代码位数（通常和MDR位数相同）

按操作码长度分类：

- 定长操作码：n位 -> 2^n条指令
  - 控制器的译码电路设计简单，但灵活性低
- 可变长操作码
  - 控制器的译码电路设计复杂，但灵活性高
- 定长指令字解构+可变长操作码 -> 扩展操作码指令格式

按操作类型分类：

- 数据传送
  - LOAD：把存储器的数据放到寄存器
  - STORE：把计算器的数据放到存储器
- 算数逻辑操作
  - 算数：加减乘除、±1、求补、福鼎运算、十进制运算
  - 逻辑：与或非、亦或、未操作、微测试、未清除、位求反
- 转移操作
  - 无条件转移：JMP
  - 条件转移：JZ（结果为0），JO（结果溢出），JC（结果有进位）
  - 调用和返回：CALL，RETURN
  - 陷阱Trap和陷阱指令
- 输入输出操作
  - CPU寄存器与IO端口之间的数据传送（端口即IO接口中低寄存器）

零地址指令

| OP |

- 不要操作数的指令（空操作、停机、关中断等）
- 堆栈计算（基于后缀表达式，两个操作数隐含存放在栈顶和次栈顶，计算结果压回栈顶）

一地址指令

| OP | A1 |

- 只需要单操作数（±1、取反、求补等）
  - 指令含义`OP(A1) -> A1`
  - 完成一次指令需要3次访存：取址 -> 读A1 -> 写A1
- 需要两个操作数，但其中一个操作数隐含在某个寄存器（ACC）
  - 指令含义`(ACC)OP(A1) -> ACC`

注：*A1指某个主存地址（类似指针），(A1)表示A1所指向地址中的内容（类似指针所指位置的内容）*

二地址指令

| OP | A1（目的操作数） | A2（源操作数） |

- 常用于两操作数的算术运算、逻辑运算
  - 指令含义：`(A1)OP(A2) -> A1`
  - 完成一条指令需要访存4次：取址 -> 读A1 -> 读A1 -> 写A1

三地址指令

| OP | A1 | A2 | A3（结果） |

- 常用于需要两个操作数的算数运算、逻辑运算
  - 指令含义：`(A1)OP(A2) -> A3`
  - 完成一条指令需要访存4次：取址 -> 读A1 -> 读A1 -> 写A3

四地址指令

| OP | A1 | A2 | A3（结果） | A4（下址） |

- 指令含义：`(A1)OP(A2) -> A3, A4 = 下一条将要指令指令的地址`
- 完成一条指令需要访存4次：取址 -> 读A1 -> 读A1 -> 写A3

通常取指令后`PC+1`，指向下一条指令。
对于四地址指令，执行后PC直接修改为A4所指地址。
`n位地址码的直接寻址范围 = 2^n`

拓展操作码

定长指令字结构 + 可变长操作码 = 扩展操作码

通常对使用频率高的指令，分配较短操作码（类似哈夫曼编码），尽可能减少指令译码和分析的时间。
设计扩展码时需要注意：

1. 不允许短码是长码的前缀：即操作码不能与长操作码的前面部分的代码相同。
2. 各指令的操作码一定不能重复。

E.G. 设指令字长固定16位，设计一套指令系统满足下表左侧：

-	-	-	-	-	-

-	-	-	-	-	-
有15条三地址指令		0000-1110	A1	A2	A3
有12条二地址指令	1111 XXXX XXXX XXXX	1111	0000-1011	A1	A2
有62条一地址指令	1111 11XX XXXX XXXX	1111	1100-1110,1111	0000-1111,0000-1101	A1
有32条零地址指令	1111 1111 111X XXXX	1111	1110-1111	1110-1111	0000-1111

设地址长度为n，上一层留出m种状态，下一层可扩展出m*2^n中状态

指令寻址

确定下一条欲执行指令的存放地址（始终由程序计数器PC给出）。
分为顺序寻址、跳跃寻址。

顺序寻址

对于定长指令字解构：PC按照指令地址的间隔d，在执行完成指令后(PC)+d -> PC

对于变长指令字结构：根据操作码判断指令的总字节数n，修改PC的值(PC)+n -> PC;
根据指令类型，CPU还要进行多次访存，每次读入一个字。

跳跃寻址

若执行到JMP指令，将会改变程序执行流，执行转移指令，直接修改PC值。

数据寻址

确定本条指令的地址码指明的真实地址。

例如对于指令JMP n，根据实际情况，它可以有以下情形：

- 直接跳转到地址为n的指令
- 跳转到相对起始地址+n的指令
- 跳转到相对于当前PC+n的指令(PC)+n -> PC

所以在原有的指令解构上增加若干比特位，用于表示数据寻址的特征（简写为EA）
| 操作码OP | 寻址特征 | 形式地址A |

二地址指令则有
| 操作码OP | 寻址特征 | 形式地址A1 | 寻址特征 | 形式地址A2

寻址方式	**有效地址**	**访存次数（指令执行期间）**
隐含寻址	程序指定	0
立即寻址	A即是操作数	0

寻址方式	有效地址	访存次数（指令执行期间）
直接寻址	EA=A	1
一次间接寻址	EA=(A)	2
寄存器寻址	EA=Ri	0
寄存器间接一次寻址	EA=(Ri)	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
堆栈寻址	入栈/出栈时EA的确定方式不同	硬堆栈不访存，软堆栈访存一次

直接寻址

指令中的形式地址A就是操作数的真实地址EA，即`EA = A`

简单，且指令执行阶段仅访问一次主存，不用撰文计算操作数的地址。
但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（灵活性较差）。

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地址，也就是操作数地址的地址。
即`EA = (A)`

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）。
（对于多次寻址）便于编制程序（用间接寻址可以方便地完成子程序的返回）。

但指令在执行阶段需要多次访存（一次间接寻址需要两次访存，多次寻址需要根据存储字的最高位确定访存次数）。

寄存器寻址

在指令自重直接给出操作数所在的寄存器编号，即`EA = Ri`， 其操作数在由Ri所指的寄存器内。

这样只需要在取指令时访存一次，执行指令时只要访问寄存器，而不需要缓存。

指令在执行阶段不访问驻村，只访问寄存器，指令字短且执行速度快，执行向量/矩阵运算。
但寄存器昂贵，计算机的寄存器个数有限。

寄存器间接寻址

寄存器Ri中给出的是操作数所在贮存的单元地址，即`EA = (Ri)`。

这样取指令和执行指令各需访存一次（共2次）。

与一般间接寻址相比速度快。
但指令的执行阶段需要访问主存（因为操作数在主存中）。

隐含寻址

在指令中隐含着操作数的地址（不明显给出操作数的地址）。

有助于缩短指令字长。
但需要增加存储操作数或隐含地址的硬件。

立即寻址

形式地址A就是操作数本身（又称为立即数），一般用补码形式，#表示立即寻址特征。

取指令时访存一次（执行指令不需要访问）。

指令执行阶段不访问驻村，指令执行时间最短。
但A的位数限制了立即数的范围。比如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`。

偏移寻址（数据寻址的一种）

根据形式地址作为偏移量来得到实际地址。主要有如下几种：
堆栈选址
基址寻址`EA = (BR) + A`，用于转移指令，BR是基址寄存器
变址寻址`EA = (IX) + A`，用于多道程序，IX是变址寻址器
相对选址`EA = (PC) + A`，用于循环程序、数组，PC是地址计数器

基址寻址

将CPU中的基址寄存器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`EA = (BR) + A`

有时会使用通用寄存器作为基址寄存器

基址寄存器面向操作系统，其内容有操作系统或管理程序确定（程序员无法接触到）。

便于程序“浮动”，方便多道程序并发运行。

变址寻址

有效地址EA等与指令字中的形式地址A与变址寄存器IX的内容相加之和`EA = (IX) + A`，IX可以是变址寄存器(专用)，也可以用通用寄存器作为变址寄存器。

变址寄存器面向用户，IX的内容可由用户改变，形式地址A不变。

实际使用中往往会需要多种寻址方式符合使用，例如先基址后变址寻址：`EA = (IX) + ((BR)+A)`

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`，
A是相对于PC所指地址（正在执行指令的下一条指令的位移量），补码表示，可正可负。

操作数的地址会随着PC的变化而变化，与指令地址间总是相差一个固定值，便于程序浮动（一段代码内部的活动）。
广泛应用于转移指令。

堆栈寻址

操作数存放在对战中，隐含使用堆栈指针SP作为操作数地址。

对战是存储器（或专用寄存器组）中的一块特定的按LIFO的原则管理的存储区，区中悲读/写单元的地址是用一个特定的计算器给出，该寄存器称为堆栈指针SP。

有寄存器中的硬堆栈（成本高），和主存中的软堆栈

对战可用于函数调用时保存当前函数相关信息。

机器级代码

机器语言由二进制代码构成，汇编语言由助记符构成，高级语言翻译成汇编后再翻译为机器语言。

指令的作用：

- 改变程序执行流
- 处理数据
- 指令格式：操作码+地址码
    - 操作码（负责如何处理）：
        - 算术运算：加减乘除、取负数、自增自减
        - 逻辑运算：与或非、异或、左移、右移
        - 其他
    - 地址码（负责寻找数据）：
        - 寄存器
            - 在指令中给出“寄存器名”
            - 通用寄存器：eax, ebx, ecx, edx
                - X=未知，E=Extended=32bit，
                - 去除E则是使用低16bit（变址和堆栈寄存器不行）
            - 变址寄存器：esi, edi
                - I=Inex，S=Sourse，D=Destination
            - 堆栈寄存器：ebp、esp
                - BP=BasePointer，SP=StackPointer
        - 主存
            - 在指令中给出度写长度、主存地址
            - dword ptr——双字，32bit
            - word ptr——单字，16bit
            - byte ptr——字节，8bit
        - 指令
            - 直接在指令中给出要操作的数，“立即寻址”
            - 直接给出常量

以mov指令为例：

`mod 目的操作数d，源操作数s` # 将s复制到d的位置

```
mov eax, ebx     # 将寄存器ebx的数值复制到寄存器eax
mov eax, 5       # 将立即数5复制到寄存器eax
mov eax, dword ptr[af996h]  # 将内存地址af996h所指的32bit值复制到寄存器eax
mov byte ptr[af996h], 5     # 将立即数5复制到内存地址af996h所指的一字节中
move eax, dword ptr[ebx]    # 将ebx所指主存地址的32bit复制到eax寄存器
mov dword ptr[ebx], eax     # 将eax的内容复制到ebx所指主存地址的32bit
mov eax, bytr ptr[ebx]      # 将ebx所指的主存地址的8bit复制到eax
mov eax, [ebx]              # 若未指明读写长度，默认32bit
mov [af996h], eax           # 将eax的内容复制到ad996h所指的地址（未指明长度默认
32bit)
mov eax, dword ptr[ebx+8]   # 将ebx+8所指的主存地址的32bit复制到eax寄存器中
mov eax, dword ptr[af996-12h]   # 将af996-12所指的主存地址的32bit复制到eax寄存器中
```

考试要求

- 只关注x86汇编语言（其他架构会详细注释）
- 结合C语言看懂汇编语言的关键语句
  - 常见指令
  - 选择结构
  - 循环结构
  - 函数调用
- 结合汇编语言分析机器语言指令的格式、寻址方式
- （不会考C翻译为汇编或机器语言）

常见指令

算术运算指令

功能	汇编指令	注释
加	add d, s	add，`d=d+s`
减	sub d, s	substract，`d=d-s`
乘	mul d, s imul d, s	multiply，integer。mul无符号数`d=d*s`；imul有符号数乘法
除	div s idiv s	divide，integer。div无符号除法`edx:eax/s，商存入eax，余数存入edx`；idiv有符号除法
取负数	neg d	negative，`d=-d`
自增++	inc d	increase，`d++`
自减--	dec c	decrease，`d--`

逻辑运算指令

功能	汇编指令	注释
与	and d, s	and，将d、s逐位相与，结果放回d
或	or d, s	or，将d、s逐位相或，结果放回d
非	not d	not，将d逐位取反，结果放回d
异或	xor d, s	exclusive or，将d、s逐位异或，结果放回d
左移	shl d, s	shift left，将d逻辑左移s位，结果放回d
右移	shr d, s	shift right，将d逻辑左移s位，结果放回d

其他指令

实现分支结构、循环结构:
cmp、test、jmp、jxxx

实现函数调用:
push、pop、call、ret

实现数据转移:
mov

AT&T格式汇编语言

通常用在Unix、Linux系统

Intel格式是Windows常用格式

-	AT&T格式	Intel格式
目的操作数d、源操作数s	op s, d	op d, s
寄存器的表示	mov %ebx, %eax	mov eax, ebx
立即数的表示	mov $985, %eax	mov eax, 985
主地址的表示	mov %eax, (af996h)	mov [af996h], eax
读写长度表示	movb $5, (af996) movw $5, (af996h) movl $5, (af996h) addb $4, (af996)	mov byte ptr[af996h], 5 mov word ptr[af996h], 5 mov dword ptr [af996h], 5 add byte ptr [af996h], 4
主存地址偏移量的表示	movl -8(%ebx), %eax movl 4(%ebx, %ecx, 32), %eax *偏移量(基址,变址,比例因子)*	mov eax, [ebx-8] mov eax, [ebx+ecx*32+4] *[基址+变址*比例因子+偏移量]*

实现选择语句

程序计数器PC在Intel x86处理器通常被称作IP

从jmp指令引入

无条件转移指令：`jmp <地址> # PC无条件转移至<地址>`

```
mov eax, 7
mov ebx, 6
jmp NEXT    # 用“标号”锚定位置，名字可自取。类似goto
mov ecx, ebx
NEXT:
mov ecx, eax
```

条件转移指令：jxxx

两数比较：`cmp a, b`

```
cmp a, b  # 两数比较
je <地址>     # jump when equal, a==b时跳转
jne <地址>    # jump when not equal, a!=b时跳转
jg <地址>     # jump when greater then, a>b时跳转
jge <地址>    # jump when greater then or equal, a>=b时跳转
jl <地址>     # jump when less then, a<b时跳转
jle <地址>    # jump when less then or equal, a<=b时跳转

"""配合标号的跳转例子"""
cmp eax, ebx
jg NEXT

"""将c翻译成汇编的例子"""
'''c
if(a>b){
  c = a;
} else {
  c = b;
}
'''
'''翻译后'''
cmp eax, ebx
jg NEXT       # 若满足a>b，跳转到NEXT。否则进入else（如下内容）
mov ecx, ebx
jmp END
NEXT:
mov ecx, eax
END:
```

用条件转移指令实现循环

```
"""将一个c语言的循环语句翻译为汇编"""
'''c
int result = 0;
for(int i=1; i<=100; i++){
  result += i;
}
'''
'''翻译后'''
move eax, 0     # eax保存result
mov edx, 1      # edx保存i
cmp edx, 100
jg L2           # 若i>100，跳转到L2
L1:             # 循环主体
add eax, edx    # 实现result += i
inc edx         # 自增指令，实现i++
cmp edx, 100
```

```
jle L1          # 若i<=100，跳转到L1
L2:             # 跳出循环
```

用loop实现循环

```
"""将一个c语言的循环语句翻译为汇编"""
'''c
for(int i=500; i>0; i--){
  <执行操作>
}
'''
'''翻译后'''
mov ecx, 500
Looptop:        # 循环开始
<执行操作>
loop Looptop    # ecx--，若exc!=0，跳转到Looptop
'''
loop Looptop等价于：
dec ecx
cmp ecx, 0
jne Looptop
'''
```

补充：loopnz、loopz

loopnz：loop not zero，当exc!=0且ZF==0时，继续循环

loopz：loop zero，当exc!=0且ZF==1时，继续循环

实现函数调用

函数调用指令：`call <函数名>`
函数返回指令：`ret`

```
caller:
... ...
call add    # 调用add
... ...
leave
ret

add:        # add的功能
... ...
leave
ret         # 返回call并继续运行
```

访问栈帧数据：push、pop指令

```
push eax        # 将寄存器eax的值压栈
push 986        # 将立即数985压栈
push [ebp+8]    # 将主地址[ebp+8]里的数据压栈

pop eax         # 栈顶元素出栈，写入寄存器eax
pop [ebp+8]     # 栈顶元素出栈，写入主存地址[ebp+8]
```

访问栈帧数据：mov指令

通过add和sub调整当前函数的栈帧范围，
用mov指令结合esp和ebp访问栈帧数据。

```
sub esp, 12         # 顶栈指针-12
mov [esp+8], eax    # 将eax的值复制到主存[esp+8]
mov [esp+4], 985    # 将985复制到主存[esp+4]
mov eax, [ebp+8]    # 将主存[ebp+8]的值复制到eax
mov [esp], eax      # 将eax的值复制到主存[esp]
add esp, 8          # 顶栈指针+8
```

切换栈帧

call指令的作用：

1. 将IP旧值压栈保存（相当于`push IP`）
2. 设置IP新值，无条件转移至被调用函数的第一条指令（效果相当于`jmp add`）

当一个函数return之前，只需要如下指令，即可让esp和ebp重新只会上一层函数的栈帧（**效果等价于leave**）：

```
mov esp, ebp    # 让esp指向当前栈顶的底部
pop ebp         # 将esp所指元素出栈，写入寄存器ebp
# 效果等价于leave
```

传递参数和返回值

将*局部变量*集中存储在栈帧*底部*区域；
将*调用参数*集中存储在栈帧*顶部*区域。

栈帧最*底部*一定是*上一层栈帧基址*（ebp旧值）；
栈帧最*顶部*一定是*返回地址*（当前函数的栈帧除外）。

汇编中调用函数的情况：

- **调用者**：
  - 保存必要的寄存器
    - eax、edx、ecx等
  - 将调用参数写入当前栈帧的顶部区域

        - push或mov可实现
    - 执行call指令
        - 返回之地亚茹顶栈、并跳转到被调用函数的第一条指令
    - 使用返回值
        - 通过eax寄存器
    - 回复必要的寄存器
- **被调用者**:
    - 保存上一层函数的栈帧，设置当前函数的栈顶
        - push ebp
          mov ebp, esp
        - 或enter指令
    - 初始化局部变量
        - [ebp-4]、[ebp-8]...
    - 一系列逻辑处理
    - 向上层函数传递返回值
        - 通过eax寄存器
    - 恢复上一层函数的栈帧
        - mov esp, ebp
          pop ebp
        - 或leave指令
    - 执行ret指令
        - 从栈顶找到返回地址，出栈并恢复IP值

CISC、RISC

CISC和RISC是指令系统的两种设计方向

CISC = Complex Instruction Set Computer复杂指令集计算机系统，
一条指令可以完成一个复杂的基本功能，
主要用于x86架构。

RISC = Reduced Instruction Set Computer精简指令集计算机系统，
一条指令完成一个基本操作，多条指令组成完成一个复杂的基本功能，
主要用于ARM架构。

CPU

CPU由
运算器（对数据加工：累加器ACC、乘商寄存器MQ、通用操作数寄存器X、算术逻辑单元ALU）和
控制器（协调控制计算机各部件执行程序的指令序列：控制单元CU、指令寄存器IR、程序计数器PC）组成。

控制器的基本功能包括：取指令、分析指令、执行指令、中断处理

一条指令会经过：取指周期FE，间址周期IND，执行周期EX，中断周期INT

CPU的功能有：
指令控制（程序顺序的控制），
操作控制（管理并产生由内存取出的每条指令的操作信号，把并送往对应部件并控制其按指令要求运行），
时间控制（为每条指令按时间允许提供应有的控制信号），
数据加工（读数据进行算数和逻辑运算），
中断处理（对异常清理和特殊请求进行处理）。

运算器的基本结构

算术逻辑单元：进行算术/逻辑运算。

通用寄存器组（AX、BX、CX、，R0、R1，SP等）：存放操作数。SP是堆栈指针，用于指示栈顶地址。

暂存寄存器：暂存从主存读来的数据，此数据不存放在通用寄存器（否则会破坏原有内容）。

程序状态字寄存器PSW：保留由算数逻辑运算指令或测试指令的结果而建立的各种状态信息。

移位器：对运算结果进行移位计算。

计数器：控制乘除法运算。

寄存器输入/输出到ALU

`ALU <= [R0 R1 R2 R3]`，如果直接用导线连接，相当于多个寄存器同时一直向ALU传输数据。解决方法：

1. 使用多路选择器：`ALU <=MUX= [R0 R1 R2 R3]`，控制信号选择一路输出。
2. 使用三态门：`ALU <=◁= [R0 R1 R2 R3]`，控制每一路是否输出。
3. CPU内部总线：`ALU <=总线= [R0 R1 R2 R3]`，将所有寄存器的IO端接到一条公共通路。并引入暂存寄存器、累加寄存器、PSW。

1、2方案性能高，寄存不存在数据冲突，但结构复杂，硬件量大，不易实现；
3方案解构简单、易实现，数据传输存在较多冲突，性能低。

控制器的基本结构

程序计数器PC：指明下条指令在主存中存放的地址，有自增功能。

指令寄存器IR：保存当前正在执行指令。

指令译码器：对操作码字段进行译码，想控制器提供特定的操作信号。

微操作信号发生器：根据IR内容（指令）、PSW内容（状态信息），产生控制整个计算机系统所需的各种控制信号。

时钟系统：产生时序信号，由统一时钟CLOCK分频得到。

存储器地址寄存器MAR：存放所要访问的主存单元的地址。

存储器数据寄存器MDR：存放向驻村写入的信息或从主存中读出的信息。

指令执行过程

指令周期、数据流、指令执行方案。

指令周期

指令周期是CPU从主存中每取出并执行一条指令的全部时间。
通常用若干机器周期（CPU周期）来表示。
一个机器周期包含若干时钟周期（CPU的最基本单位）。

一个指令周期的完成内容：
开始-》取指令`PC+1`-》对指令译码-》执行指令-》取下条指令`PC+1`
------<=======取指周期=======><=执行周期=>------------

指令周期内的机器周期数可以不等，每个机器内的节拍数也可以不等。

指令周期的流程：

- *取指周期*
- 有间接地址吗?
  - 是：间址周期
  - 否：执行周期
- 有中断吗?
  - 是：中断周期
  - 否：继续回到*取指周期*

指令周期的数据流

取指周期：

1. 当前指令地址送至存储器地址寄存器：`(PC)->MAR`
2. CU发出信号，经控制总线传到主存（读信号）：`1->R`
3. 将MAR所致主存中的内容经数据总线送入MDR：`M(MAR)->MDR`
4. 将MDR中的内容（指令）送入IR：`(MDR)->IR`
5. CU发出控制信号，形成下一条指令地址：`(PC)+"1"->PC`

间址周期：

1. 指令的地址码送入MAR：`Ad(IR)->MAR`或`Ad(MDR)->MAR`
2. CU发出控制信号，启动主存做读操作：`1->R`
3. MAR所指主存中的内容经数据总线送入MDR：`M(MAR)->MDR`
4. 将有效地址送到指令的地址码字段：`(MDR)->Ad(IR)`

执行周期:
不同指令的执行周期操作不同，没有统一的数据流向。

中断周期:
暂停当前任务去完成其他任务。为了可以回复当前任务，需要保存断点。一般用堆栈来保存断点，SP只想站定元素，进栈操作先修改指针，再存入数据。

1. CU控制将SP减1，修改后的地址送入MAR：`(SP)-1 -> SP, (SP)->MAR`
2. CU发出控制信号，启动主存做写操作：`1->W`
3. 将断点（PC内容）送入MDR：`(PC)->MDR`
4. CU控制将中断服务程序的入口地址（由向量地址形成部件产生）送入PC：`向量地址->PC`

指令执行方案

单指令周期:
所有指令的执行时间是相同的，指令周期取决于执行时间最长的指令的执行时间；
指令间串行运行。
*但是*本可以较短时间内完成的指令需要用最长指令周期的时间完成，降低整个系统的运行速度。

多指令周期:
不同指令用不同步骤执行，可以选用不同个数的时钟周期来完成不同指令间的执行过程；
指令见串行执行。
*但是*需要更复杂的硬件设计。

流水线方案:
每个始终周期启动一条指令，尽可能让多条指令同时运行，但各自在不同的执行步骤；
指令之间并行执行。

数据通路

数据通路是数据在功能部件之间传送的路径。其基本结构包括有：

1. CPU内部单总线方式
2. CPU内部多总线方式
3. 专用数据通路方式

CPU内部单主线

内部总线是一个部件内部连接个寄存器以及运算部件之间的总线（例如CPU）；
系统总线是一台计算机系统各部件和各类I/O接口间互相联通的总吸纳。

寄存器之间数据传送:
e.g. 将PC送至MAR，实现传送操作的流程以及控制信号为：

```
(PC) -> Bus    # PCout有效，PC内容送总线
Bus -> MAR     # MARin有效，总线内容送MAR
# 可以写为：(PC) -> Bus -> MAR
```

主存与CPU之间的数据传送:
eg. CPU从主存读取指令，实现传送操作的历程以及控制信号为:

```
(PC) -> Bust -> MAR   # PCout和MARin有效，现行指令地址->MAR
1->R                  # CU发出读命令
MEM(MAR) -> MDR       # MDRin有效
MDR -> Bus -> IR      # MDRout和IRin有效，现行指令->IR
```

算术运算或逻辑运算的数据传送:
eg. 一条假发指令，未操作序列以及控制信号为:

```
Ad(IR) -> Bus -> MAR    # MDRout和MARin有效
1 -> R                  # CU发读命令
MEM(MAR) -> 数据线 -> MDR   # MDRin有效
MDR -> Bus -> Y     # MDRout和Yin有效，操作数->Y
(ACC) + (Y) -> Z    # ACCout和ALUin有效，CU向ALU发送加命令
Z -> ACC            # Zout和ACCin有效，结果->ACC
```

例题：以指令`ADD (R0), R1`的流程和控制信号为例

功能：`((R0))+(R1) -> (R0)`,
取指周期、间址周期、执行周期

各阶段指令流程:

取指周期：公共操作

时序	微操作	有效控制信号
1	(PC)->MAR	PCout, MARin
2	M(MAR)->MDR	MemR, MARout, MDRinE
3	(MDR)->IR	MDRout, IRin
4	指令译码	-
5	(PC)+1 -> PC	-

间址周期：完成取数操作，被加数在主存中，加数已经在寄存器R1中

时序	微操作	有效控制信号
1	(R0)->MAR	R0out, MARin
2	M(MAR)->MDR	MemR, MARout, MDrinE
3	(MDR)->Y	MDRout, Yin

执行周期：完成取数操作，被加数在主存中，加数已经放在寄存器R1中

时序	微操作	有效控制信号
1	(R1)+(Y) -> Z	R1out, ALUin, CU向ALU发ADD控制信号
2	(Z)->MDR	Zout, MDRin
3	(MDR)->M(MAR)	MemW, MDRoutE, MARout

专用通路结构

效率更多，成本更高，借助多路选择器与三态门。

取指周期:

```
(PC) -> MAR     # PCout, MARin
(MAR) -> 主存   # MARout
1 -> R          # 控制单元箱主存发送读命令
M(MAR) -> MDR   # MDRin
(MDR) -> IR     # MDRout, IRin
(PC)+1 -> PC
# (如果有控制信号): Op(IR) -> CU    # CUin
```

控制器设计

-	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形式存储在控制存储器中， 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	快
规整性	较规整	繁琐、不规整
应用场合	CISC	RISC
易扩充性	易扩充修改	困难

硬布线控制器

根据指令操作码、目前的机器周期、节拍信号、机器状态条件，即可确定现在这个接拍下应该发出哪些“微命令”。

控制单元CU会接收来自节拍发生器的信号T，并且可以用与门来设计电路，当T和指令信号同时为True时，则会发出对应微命令C。那么对于微操作`(PC) -> MAR, C = FE·T`。

设计步骤:

1. 分析每个阶段的微操作序列（取值、间址、执行、中断）
2. 选择CPU控制方式
    - 选择定长或不定长
3. 安排微操作时序
    - 先后顺序不得随意更改
    - 对于不同的被控对象，其微操作尽量在一个节拍完成
    - 时间较短的微操作，一个节拍内完成，并允许有先后顺序
4. 设计电路
    - 列出操作时间表
    - 写出微操作命令的最简表达式
    - 画出逻辑图

特点:
指令越多，设计和实现越复杂，因此一般用于RISC；
若扩充一条新指令，则控制器设计就要大改，扩充指令较困难。
由于使用纯硬件实现控制，执行速度快。未操作控制信号由组合逻辑电路即时产生。

微程序控制器

在微程序控制器中，微程序由微指令序列组成，*每种指令对应一个微程序*。

指令是对程序执行步骤的描述；
微指令是对指令执行步骤的描述。

若干微指令构成指令；若干指令构成程序。

CU结构:

1. 微地址形成部件
    - 微地址即微指令在CM中存放地址
    - 通过指令操作码形成对应为程序的第一条微指令的存放地址
2. 顺序逻辑
    - 根据机器标志和时序信息确定下一条微指令的存放地址
3. CMAR（μPC）
    - 明确接下来要执行的微指令的存放地址
4. 地址译码器
    - 将CMAR内的地址信息译码为电信号，控制CM读出微指令
5. 控制存储器CM
    - 存放所有机器指令对应的微程序（微指令序列）
    - 用ROM实现，按地址寻访。通常在CPU出厂时就把所有微程序写入。
6. CMDR（μIR）
    - 微指令寄存器，用于存放当前要执行的微指令。`CM(μPC)->μIR`

微程序控制器的工作原理:

- 指令周期 = 取指周期 -> 间址周期 -> 执行周期 -> 中断周期
- 取值周期、间址周期、中断周期的微指令序列通常是公用的；执行周期的微指令序列各不相同
- 取指周期的微指令序列固定从#开始存放；执行后期的微指令序列的存放根据指令操作码确定

微指令控制器解构：

- IR
- ->微地址形成部件
- ->顺序逻辑
    - <-标志
    - <-CLK
- ->CMAR
- ->地址译码
- ->控制存储器CM
- ->CMDR
    - 下地址->顺序逻辑
- ->CPU内部和系统总线的控制信号

微指令的设计

微命令按并行执行与否可分为：相容性微指令，互斥性微指令

有三种格式的微指令：

1. **水平型**：一条微指令定义多个可并行微指令
    - |<-操作控制->|<-顺序控制->|
    - 微程序短，执行速度快；
    - 但微指令长，编写微程序较麻烦
2. **垂直型**：一条微指令只定义一个微命令，由操作码字段规定具体功能
    - |<-微操作码->|<-目的地址->|<-源地址->|
    - 微指令短，简单规整，便于编写；
    - 但微程序长，执行速度慢，工作效率低
3. **混合型**：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，便于编写
    - 微程序补偿，执行速度快

编码方式：

1. **直接编码**：在微指令的操作控制字段中，每一位代表一个微操作命令。
    - 简单直观速度快，操作并行性好；
    - 但微指令字长过长，n个微指令就要微指令的操作字段由n位，控存容量极大。
2. **字段直接编码**：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。
    - 互斥性微指令分在同一段内，相容性的分在不同段；
    - 每个小段信息为不能太多；
    - 每个小段还要留出一个状态，用000表示不操作。
    - 可以缩短微指令字长；但要通过译码电路后在发出微命令，比直接编码慢。
3. **字段间接编码**：一个字段中某些微命令由另一个字段中的某些微命令来解释。
    - 可进一步缩短微命令字长；
    - 但削弱了微指令的并行控制能力，故通常作为字段直接编码的方式的一种辅助手段。

微指令地址形成方式：

1. 微指令的下地址字段指出：微指令格式中设置下一个地址字段，由微指令的下地址字段直接指出后续为指令单地址，这种方式又被称为**断定法**
2. 根据机器指令的操作码形成

3. 增量**计数器法**(CMAR)+1 -> CMAR
4. 分支转移：指明判别条件（转移方式），指明转移成功后的去向（转移地址）
5. 通过测试网络（顺序逻辑）
6. 由硬件产生微程序入口地址

微程序控制单元的设计

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以及节拍安排
    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：
        1. 取指周期：
           Ad(CMDR -> CMAR)
           OP(IR) -> 微地址形成部件 -> CMAR
        2. 执行周期：
           Ad(CMDR) -> 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。
    - 根据CM中存储的微指令总数，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以确定微指令字长。
4. 编写微指令码点
    - 根据操作控制字段的每一位代表的微操作命令，编写每一条微指令的码点。

微程序设计的分类：

1. 静态微程序设计
    - 程序无需改变，采用ROM存储
2. 动态微程序设计
    - 通过改变微指令和微程序改变机器指令，利于仿真，采用EPROM

CPU

指令流水线

一条指令的执行过程可以分为多个阶段或过程。

1. 顺序执行方式
    - 传统冯诺依曼机采用顺序执行（也称串行执行方式）。总耗时$T = n\times3t = 3nt$
    - 控制简单，硬件代价小；
    - 但执行指令的速度慢，任何时刻只要有一条指令在执行，各功能部件的利用率低。
2. 一次重叠执行方式
    - 总耗时$T = 3t+(n-1)\times2t = (1+2n)t$
    - 程序执行时间缩短了$1/3$，各功能部件利用率明显提高；
    - 但硬件开销较大，控制过程比顺序执行复杂。
3. 二次重叠方式
    - 总耗时$T = 3t+(n-1)\times t = (2+n)t$
    - 指令执行时间缩短近$2/3$。一种理想的指令执行方式，正常情况下，处理机同时又2条指令在执行。
    - （还有三次、四次或更多折叠方式）

流水线的性能指标

1. **吞吐率**：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。若任务数为n，处理完成n个任务所用的时间Tk，则计算流水线吞吐率TP的基本公式为：$TP = n/Tk$。
    - 当连续输入任务$n->\infty$时，最大吞吐量$TPmax = 1/\triangle t$。
    - 一条指令的执行分为k个阶段，每个阶段耗时$\triangle t$，一般取$\triangle t$=一个始终周
    - 流水线实际吞吐率为$TP = n/((k+n-1)\triangle t)$
    - 流水线时空图在头尾有装入时间和排空时间。
2. **加速率**：完成同样任务，不使用流水线与否的时间之比。
    - 不使用流水线的执行时间T0（顺序执行所用时间），使用流水线的执行时间Tk，流水线加速比$S = T0/Tk$
    - 当连续输入任务$n->\infty$时，最大加速比$Smax = k$
    - 顺序完成n个任务耗时$T0 = nk\triangle t$，流水线完成n个任务耗时$Tk = (k+n-1)\triangle t$，实际加速比$S = kn\triangle t/(k+n-1)\triangle t = kn/(k+n-1)$
3. **效率**：流水线的设备利用率。
    - 流水线效率$E = 完成n个任务占用的时空区有效面积/n个任务所用时间与k个流水段所谓成的时空区总面积 = T0/(kTk)$

指令流水线的影响因素

MIPS架构的五段流水线：取指令IF，指令译码ID，指令执行EX，访存M，写回通用寄存器WB

为了方便流水线设计，会将每阶段耗时取一样（以最长的为准），则流水线每个部件后面都要有一个缓冲寄存器（锁存器）来保存本流水段的执行结果，供下一段流水使用。

1. **结构相关**（资源冲突）
    - 多条指令在同一时刻争用同一资源而形成的冲突
    - 解决方法：
        1. 可以让后一相关指令暂停一周期
        2. 或资源重复配置：数据存储器+指令存储器
2. ***数据相关***（数据冲突）
    - 一个程序中，存在必须等前一条指令执行完后才能执行后一条指令的情况
    - 解决方法：
        1. 相关指令以及后续指令都暂停1~数个时钟周期，直到数据相关问题消失后继续。可用*硬件阻塞stall（纯硬件阻塞）或软件插入NOP（空指令阻塞）*
        2. 数据旁路技术：处理后的数据除了写回原定寄存器外，还额外送入下一步需要进行操作的部件
        3. 编译优化：编译器调整指令顺序来解决
3. **控制相关**（控制冲突）
    - 放流水线遇到转移指令和其他改变PC的指令而造成断流
    - 解决方法：
        1. 转移指令分支预测。有*简单预测（永远猜TrueOrFalse）*、*动态预测（根据历史情况动态调整）*
        2. 预取转移成功和不成功的两个控制流方向上的目标命令
        3. 加快和提前形成的条件码
        4. 提高转移方向的猜准率

流水线的分类

1. **部件功能级、处理机级、处理机间级流水线**：根据流水线的级别不同而分类
    1. 部件功能级流水：将复杂的算术逻辑运算组成流水线。例如将浮点加减操作分为求接差、对阶、尾数相加、结果规格化等过程
    2. 处理机级流水：一条指令解释过程分为多个子过程。例如取址、译码、执行、访存、写回
    3. 处理机间流水：宏流水，每个处理机完成某一专门任务，各处理机所得到的结果许存放在与下一个处理机所共享的存储器中
2. **单功能流水线、多功能流水线**：根据流水线可完成的功能分类
    1. 单功能流水线：只能实现一种固定的专门功能的流水线
    2. 多功能流水线：通过各段间的不同连接方式可以实现多种功能的流水线
3. **动态流水线、静态流水线**：根据同一时间内各段之间的连接方式分类
    1. 静态流水线：同一时间内，流水线各段只能按某一功能的连接方式工作

2. 动态流水线：同一时间内，当某些段正在实现某种运算，另一些段正在进行另一种运算。高效，但流水线控制复杂

4. **线性流水线、非线性流水线**：根据各个功能段之间是否有反馈信号分类

    1. 线性流水线：从输入到输出，各功能段只允许经过一次，不存在反馈回路
    2. 非线性流水线：存在反馈回路。适合进行线性递归运算

流水线的多发技术

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*
    - 通过编译优化技术，把可并行的执行指令搭配

2. **超流水技术**：

    - *在一个时钟周期*内*再分段*（3段）
    - 在一个时钟周期内，*一个功能部件使用多次*（3次）
    - *不能调整*指令的*执行顺序*
    - 靠*编译程序*解决*优化问题*

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*（可达几百位）
    - 采用*多个处理部件*

五段式指令流水线

MIPS架构的五段流水线：取指令IF，指令译码ID，指令执行EX，访存M，写回通用寄存器WB

常见五类指令：运算类、LOAD、STORE、条件转移、无条件转移

运算类：
IF：根据PC从指令Cache取指令到IF锁存器;
ID：去除操作数到ID段锁存器；
EX：运算，将结果存入EX段锁存器；
M：空段 WB：将运算结果写回指定寄存器

LOAD：
IF：根据PC从指令Cache取指令到IF锁存器；
ID：将基址寄存器的值放到锁存器A，将偏移量的值放到Imm；
EX：运算，得到有效地址；
M：从数据Cache中取数并放入锁存器；
WB：将取出的数写回寄存器

STORE:
IF：根据PC从指令Cache取指令到IF段锁存器；
ID：将基址寄存器的值放到锁存器A，将偏移量的值放到Imm。将要存的数放到B；
EX：运算得到有效地址。并将锁存器B的内容放到锁存器Store；
M：写入数据Cache；
WB：空段

条件转移:
IF：根据PC从指令Cache取指令到IF段的锁存器；
ID：进行比较的两个数放入锁存器A、B，偏移量放入Imm；
EX：运算，比较两数；
M：将目标PC值写回PC WB：空段

无条件转移:
IF：根据PC从指令Cache取指令到IF段锁存器；
ID：偏移量放入Imm；
EX：将目标PC值写回PC；
M：空段；
WB：空段

多处理器

- 多处理器
    - 多处理器multiprocessor：两个或以上处理器的计算机系统
    - 任务级并行task-level parallelism或进程级并行process-level parallelism：同时运行独立程序来利用多处理器
    - 并行处理程序：同时运行在多个处理器上的单一程序
    - 集群cluster：局域网连接的一组计算机，等同于一个大型多处理器
    - 多核处理器multicore microprocessor：单一集成电路上包含多个处理器的微处理器
    - 共享内存处理器shared memory processor(SMP)：共享一个物理地址空间的并行处理器（等同于多核处理器，只是命名角度不同）

- 处理器和向量处理器
    - SISD：单指令流单数据流的单处理器
        - 一个处理器+一个主存储器
        - 各指令序列只能并发、不能并行、每条指令处理一两个数据
        - 不是数据级并行技术
        - 若采用指令流水线，需要设置多个功能部件，采用多模块交叉存储器
    - SIMD：单指令流多数据流（同样的指令在多个数据流上操作，和向量处理器中一样）。
        - 一个CU+多个处理单元或执行单元+多个局部存储器+一个主存储器
        - 各指令序列只能并发、不能并行，但每条指令可以同时处理多个具有相同特征的数据
        - 数据级并行技术
        - 有三种变体：向量体系结构、多媒体SMD指令集扩展、图形处理单元
    - MIMD：多指令流多数据流的多处理器（共享存储多处理器系统）
        - 个指令并行执行，分别处理多个不同数据
        - 一种线程级并行、甚至是线程级以上的并行技术
        - 细分：

        - 多处理器系统：多个处理器共享单一的物理地址空间；
          各处理器间通过LOAD/STORE访问同一个主存储器，通过主存相互传送数据
        - 多计算机系统：多台计算机之间只能通过"消息传递"相互传送数据；
          每台计算机拥有各自的私有存储器，物理地址空间相互独立
    - SPMD：单程序多数据流（一种传统MIMD编程模型，其中一个程序运行在所有处理器之上）
    - 数据级并行：不同数据执行相同操作所获得的并行
    - 陈列处理器：SIMD的一种，并行处理机
    - 向量机（向量处理器）：SIMD的一种
        - 多个处理单元，多组向量寄存器
        - 一条指令的处理对象是"向量"，擅长对向量型数据并行计算、浮点数运算，常用于超算
        - 主存采用"多个端口同时读取"的交叉多模块处理器
    - 标量体系结构：常规指令集体系结构

- 硬件多线程

    - 硬件多线程：线程阻塞时处理器可切换到另一线程（MIMD相关）。有三种种实现方法：
        - 细粒度多线程
        - 粗粒度多线程
        - 同时多线程SMT
    - 进程process：一个进程包含≥1个线程、地址空间、操作系统。进程切换通常需要操作系统介入
    - 线程thread：一个线程包含计数器、寄存器状态、内存栈，一个轻量级进程，多个线程共享一个地址空间（进程不是）

-	**细粒度多线程**	**粗粒度多线程**	**同时多线程SMT**
指令发射	各个时钟周期，轮流发射多个线程指令	连续几个时钟周期都发射同一线程的指令序列； 流水线阻塞时切换到另一个线程	一个时钟周期内，同时发射多个线程指令
线程切换频率	每个时钟周期	当流水线阻塞时	NULL
线程切换代价	低	高，需要重载流水线	NULL
并行性	指令级并行，线程间不并行	指令级并行，线程间不并行	指令级并行，线程级并行

- 多核处理器

    - CPU架构由 控制单元、运算单元、存储单元 这三个模块，由CPU内部总线连接。
      多核CPU则是多个核组织（多个控制单元和多个运算单元）共用存储单元。
      多核结构则在CPU内布置多个执行核（执行单元），如ALU、FPU、L2缓存，其它部分由多个核共享。
      由于布置了多个核，其指令级并行是真正并行，而非超线程结构半并行。

- 共享内存多处理器SMP

    - 共享内存多处理器SMP（共享地址多处理器）：
        - 硬件为所有处理器提供单一物理地址空间

        - 用锁来同步共享变量
            - 同步：对可能运行不同处理器上的两个或更多进程的行为进行协调
            - 锁：一个时刻进允许一个处理器访问数据的同步装置
    - SMP体系结构
        - 采用相关技术以确定和同步对资源的相关占用声明，确保进程不会从队列中丢失。但由于必须确保两个处理器不会选择同一个进程，增加了操作系统的复杂性。
    - SMP组织结构
        - 有多个处理器，每个都有自己的控制单元、算术逻辑单元、存储器
        - 每个处理器可以通过某种形式的互联机制访问一个共享内存和IO设备
        - 共享总线是一个通用方法

- 多处理器操作系统的关键涉及问题：

    - 同时对并发进程或线程
    - 调度
    - 同步
    - 存储管理
    - 可靠性和容错

总线

总线是一组能为多个部件分时共享的公共信息传输线路。包含以下特性：

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

通过总线进行信号传输，包括地址总线、数据总线、控制总线。

按数据传输格式分类：
串行总线：一条传输线，成本低，广泛应用长距离传输，节省布线空间。但在数据传输和接收、拆卸和装配，要考虑串并行转换的问题。
并行总线：逻辑是需简单，电路实现容易。但信号线数量多，占用更多布线空间，远距离传输成本高。

按总线功能分类：
片内总线：芯片内部总线。CPU内部的寄存器和寄存器间、寄存器和ALU间的公共连接线。
系统总线：计算机系统内各功能组件（CPU、主存、IO接口）间相互连接的总线。按传输信息内容的不同，可细分为地址总线、数据总线、控制总线。
通信总线：用于计算机系统之间或计算机系统与其他系统（远程通信设备、测试设备等）之间信息传送的总线（例如网线），也叫外部总线。

系统总线的结构

单总线结构：
各功能组件都接在一组总线。
结构简单成本低，易于接入新设备；
但带宽低负载重，多个部件争用唯一总线，不支持并发传送操作。

双总线结构：
两条总线：主存总线（CPU、主存、通道之间进行数据传送）、IO总线（多个外部设备与通道之间进行数据传送）。
将较低苏IO设备从蛋总线上分离，实现存储器总线和IO总线分离；
但需要增加通道等硬件设备。

三总线结构：
三条总线：主存总线、IO总线、DMA总线（直接内存访问）。
提高了IO设备性能，使其更快相应命令，提高系统吞吐量；
但系统工作效率较低。

四总线结构对于现代计算机中更常用。

总线的性能指标

传输周期（总线周期）：
一次总线操作所需时间（申请、寻址、传输、结束），由若干总线时钟周期构成。

时钟周期：
即机器的时钟周期，计算机有一个统一时钟以控制整个计算机各部件，总线也要受此控制。

工作频率：
总线周期的倒数，反映一秒内传送几次数据。

时钟频率：
时钟周期的倒数，反映一秒内时钟周期的数量。

总线宽度（总线位宽）：
总线上能同时传输的数据位数，通常指数据总线的根数（eg 32根称为32位(bit)总线。

总线带宽：
总线的数据传输率，单位时间内总线上可传输的数据位数，通常用美妙传送信息的字节数来衡量，单位字节/秒（B/s）。
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/8）（B/s）

注：总线带宽指总线本身所能达到的最高传输速率，计算实际有效数据传输率时，要用实际传输的数据量除以耗时。

总线复用：
一种信号线在不同时间传输不同信息，用较少线传输更多信息，节省空间和成本。

信号线数：
地址总线、数据总线、控制总线的线数量总和。

总线操作和定时

调度占用总线的一对设备进行数据传输。

总线定时是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，实质是一种协议或规则。包括有：

- **同步通信**（同步定时方式）：由统一时钟控制数据传送
  - 由若干时钟产生若干相等时间间隔并构成一个总线周期，期间发收双方都可进行一次数据传送。一个总线传送周期结束，下一总线传送周期开始。
  - 传送速度快，总控逻辑简单；
    但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，可靠性差。
  - 适用于总线长度较短以及总线所接部件存取时间比较接近的系统。
- **异步通信**（异步定时方式）：采用应答方式，没有公共始时钟标准
  - 主设备提出交换信息的“请求”信号，经接口传送到从设备；从设备接到主设备请求后，通过接口向主设备发出“回答”信号。
    请求和应答的信号是否互锁会分为以下类型：
    1. 不互锁方式
    2. 半互锁方式
    3. 全互锁方式
  - 总线周期长度可变，保证两个工作速度相差很大的部件间可靠地进行信息交换，自动适应时间的配合；
    但比同步控制方式复杂，速度较慢。
- **半同步通信**：同步异步结合
  - 在统一时钟的基础上，增加一个等待响应信号WAIT

- **分离式通信**：充分挖掘系统总线每瞬间的潜力。将总线的传送周期分为两个：
    1. 子周期1：*主模块*申请占用总线，使用完后放弃总线的使用权
    2. 子周期2：*从模块*申请占用总线，将各信息送至总线

输入输出

I/O

I/O接口：I/O控制器、设备控制器，负责协调主机与外部设备之间的数据传输。
I/O控制器种类多，用于控制USB设备、SATA设备的IO接口等（I/O控制器是悲集成在主板上的芯片）

以下将I/O简写为IO

IO控制方式

数据流：输入设备 -> IO接口的数据寄存器 -> 数据总线 -> CPU某寄存器 -> 主存(变量的对应位置)

1. 程序查询方式：CPU不断轮询检查IO控制器中低"状态寄存器"，检测到状态为"已完成"后，再从数据寄存器取出输入数据。
2. 程序中断方式：等待键盘IO时，CPU可以先去执行其他程序，键盘IO完成后，IO控制器向CPU发出中断请求，CPU相应中断请求并取走输入数据。

DMA控制方式

DMA接口即是DMA控制器，也是一种特殊的IO控制器。

主存与高速IO设备间有一条直接数据通路（DMA总线）。CPU向DMA接口发出"读写"命令，病知名主存地址、磁盘地址、读写数据量等参数。
DMA控制器自动控制控制磁盘与主存的数据读写，每完成一整块数据读写（如1KB为一整块），才向CPU发出一次中断请求。

通道控制方式

通道是具有特殊功能的处理器，能对IO设备进行统一管理。可理解为性能并不很强的CPU，可以识别并执行一系列通道指令，通道指令种类、功能通常更单一。

IO软件

通常用IO指令和通道指令实现主机和IO设备的信息交换

1. IO指令：CPU指令的一部分。操作码|命令码|设备码
2. 通道指令：通道能识别的指令，通道程序提前编制好放在贮存中

外部设备

外部设备（外围设备）是除主机外，能直接或间接与计算机交换信息的装置。包括有：
输入设备，输出设备，外存设备

输入设备：鼠标，键盘，等

输出设备：
显示器 [屏幕大小，分辨率，灰度级（常见于黑白显示器），刷新率，显示存储器（VRAM，刷新存储器，把一帧图像信息存储在刷新存储器中，其存储容量VARM容量 = 分辨率 × 灰度级位数，带宽 = 分辨率 × 灰度级

位数 × 帧频）];
打印机 [打击式和非打击式，串行和行式，针式、喷墨式、激光];

IO接口

IO接口（IO控制器、设备控制器）协调主机与外部设备之间的数据传输。

IO接口可以通过数据缓冲寄存器让主机和外设工作速度匹配；通过状态寄存器反馈设备的各种错误、状态信息，供CPU查用；接收从控制总线发来的控制信号、时钟信号；串并、并串等格式转换；实现主机-IO接口-IO设备间的通信。

内部接口：内部接口与系统总线相连，实质上是内存与CPU相连。

外部接口：外部接口通过接口电缆与外部相连，外部接口的数据传输可以是串并相互转换的。

- 接口Interface
    - 端口Port（寄存器）
        - 数据端口：读&写
        - 控制端口：写
        - 状态端口：读
    - 逻辑控制
- IO端口是指接口电路中可以被CPU直接访问的寄存器

对端口编址

有同一编址和独立编址两种方式。

统一编址：IO端口的地址和内存地址是同一套（存储器映射方式），靠不同的地址码区分内存和IO设备，
程序设计灵活性高，读写控制逻辑电路简单，
但端口占用了主存地址空间，外设寻址时间长。

独立编址：IO端口的地址和内存地址不是同一套，需要专门的IO指令访问IO端口，
使用专用IO指令，程序编制清晰，端口地址为树梢，地址译码速度高，不占用主存空间，
但IO指令类型少，一般只对端口进行传送操作，程序设计灵活性差，需要CPU提供存储器读写、IO设备读写两组信号，增加逻辑电路的复杂性。

IO接口工作原理

1. 发命令：发送命令字到IO控制寄存器，箱设备发送命令（需要驱动程序的协助）
2. 读状态：从状态寄存器读取状态字，会哦的设备或IO控制器的状态信息
    - 控制寄存器、状态寄存器在使用时间上是错开的，所以有的IO接口会将二者合一
    - IO控制器中的各种寄存器称为IO端口
3. 读写数据：从数据缓冲寄存器发送或读取数据，完成主机与外设的数据交换

IO方式

程序查询方式

|<-CPU执行现行程序(启动IO)->|<-CPU查询等待(IO准备)->|<-CPU控制数据传送(数据传送)->|<-CPU执行现行程序|

- 预设值传送参数；启动外设
- 取外设状态
- 外设准备就绪?
    - 否：回到取外设状态
    - 是：继续
- 传送一次数据；修改传送参数
- 传送完否?
    - 否：回到取外设状态
    - 是：继续
- 结束

接口设计简单、设备量少；
但CPU在信息传送过程中要发飞很多时间用于查询和等待，而且一段时间内只能和一台外设交换信息，效率大大降低。

e.g. 每个查询要100个时钟周期，CPU时钟频率=50MHz，CPU每秒对鼠标进行30次查询，硬盘以32位字长单位传输数据（每32位被CPU查询一次），传输率=2×2^20B/s

解：一个时钟周期=1/50Mhz=20ns，一个查询操作耗时100×20ns=2000ns；
鼠标每秒查询耗时=30×2000ns=60000ns，
查询鼠标的时间比率=60000ns/1s=0.006%；
硬盘每秒查询=(2×2^20B)/4B=2^19次，
（1B = 8bit，8位，32位即为4B）
查询硬盘耗时=2^19×2000ns=512×1024×2000ns≈1.05×10^9ns，
查询硬盘花费时间比率=(1.05×10^9ns)/1s=1.05%

程序中断方式

中断是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暂时终止现行程序，转去对异常情况或特殊请求进行处理，处理后CPU自动返回现行程序的断点处继续执行。
工作流程如下：

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

中断的类别：

- 中断（广义的中断）
    - 内中断（衣长、例外、陷入）
        - 资源中断——指令中断
        - 强迫中断
            - 硬件故障
            - 软件中断
    - 外中断（狭义的中断）
        - 外设请求

        - 人工干预
- 外中断可细分为：
    - 非屏蔽中断：关中断时也会被响应（例如掉电）
    - 可屏蔽中断：关中断时不会被响应

CPU会通过关中断指令进入原子操作。
(原子操作：CPU执行某一操作时不受其它指令影响)

中断判优（当有多个中断信号时判定哪个优先处理）：
可用硬件实现、软件实现。
硬件实现通过硬件排队器实现，它皆可以设置在CPU中，也可以分散在各个中断源中；
软件实现通过查询程序实现。

中断判优的优先级排序（靠左的优先）：

1. 硬件故障，软件中断
2. 非屏蔽中断，屏蔽中断
3. DMA请求，IO设备传送的中断请求
4. 高速设备，低速设备
5. 输入设备，输出设备
6. 实时设备，普通设备

中断隐指令的主要任务：

1. 关中断：保护中断现场
2. 保存断点：保存原程序的断电（程序计数器PC的内容），可以放入堆栈或存入指定单元
3. 引出中断服务程序：取出中断服务程序的入口地址并传送给程序计数器PC

由硬件产生向量地址，再由向量地址找到入口地址。

中断服务程序的主要任务：

1. 保护现场：保存通用寄存器和状态寄存器的内容，以便返回原程序后恢复CPU环境。可以用堆栈或特定存储单元
2. 中断服务（设备服务）：中体部分，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
3. 恢复现场：通过栈指令或取数指令吧之前保存的信息送回寄存器中

中断处理的过程：

- 取指令；执行指令
- 是否中断?
    - 否：返回取指令
    - 是：继续
- 中断隐指令（中断周期）
    - 中断响应，程序断点进站，关中断，向量地址->PC
- 中断服务程序
    - 保护现场；设备服务；恢复现场
    - 开中断，中断返回
        - 返回取指令

多重中断

执行中断程序时响应新的中断请求。CPU需要满足下列条件才能具备多重中断的功能：

1. 在中断服务程序中提前设置开中断指令
2. 优先级别高的中断源有权中断优先级别低的中断源

每个中断源都有一个屏蔽触发器，1表示屏蔽该中断源的请求，0表示可以正常申请，所有屏蔽触发器组合一起，构成一个屏蔽字寄存器，屏蔽字寄存器的内容称为屏蔽字。

e.g.：四个中断源A、B、C、D的硬件排队次序为A>B>C>D，中断服务程序时间为20us，CPU在5us开始运行B、10us运行D、35us运行A、60us运行C。要求终端次序改为D>A>C>B，写出中断源对应屏蔽字，并按时间轴画出CPU执行程序轨迹。

解：

中断源	屏蔽字(ABCD)
A	1110
B	0100
C	0110
D	1111

程序执行轨迹：
|5-B-10-D-30-B-35-A-55-B-60-C-80-B-85|

DMA方式

DMA方式可以控制数据传输的方式，DMA控制器与主存每次传送完一整块数据后才向CPU发出中断请求。通常用于控制块设备（例如磁盘）

CPU向DMA控制器致命要输入还是输出，要传送多少个数据，数据在主存、外设中的地址（1、2传送前，3、4传送时，5传送后）：

1. 接受外设发出的DMA请求（外设传送一个字的请求），并向CPU发出总线请求。
2. CPU响应此总线请求，发出总线响应信号，接管总线控制权，进入DMA操作周期。
3. 确定传送数据的主存储单元地址和长度，并能自动修改主存地址计数器和传送长度计数。
4. 规定数据在主存和外设间的传送方向，发出读写控制信号，执行数据传送操作。
5. 向CPU报告DMA操作的结束。

传送过程：

- 预处理：
  - 主存起始地址 -> AR
  - IO设备起始地址 -> DAR
  - 传送数据个数 -> WC
  - 启动IO设备
- 数据传送：
  - 继续执行主程序
  - 同时完成一批数据的传送
- 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IO设备之间交换信息时，不通过CPU。dna当IO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控制器和CPU通常采用以下方式使用主存:

1. 停止CPU访问主存
    - 控制简单
    - 但CPU处于不工作状态或保持状态，未充分发挥对主存的利用率
2. DMA与CPU交替访存
    - 不需要总线的使用权申请、建立和归还过程
    - 但硬件逻辑更为复杂
3. 周期挪用（周期窃取）
    - DMA访问主存有三种可能:
        1. CPU此时不访存（不冲突）
        2. CPU正在访存（存取周期结束让出总线）
        3. CPU与DMA同时请求访存（IO访存优先）

DMA方式和中断方式对比

-	**中断**	**DMA**
数据传送	程序控制 程序切换 -> 保存和恢复现场	硬件控制 CPU只需进行预处理和后处理
中断请求	传送数据	后处理
响应	指令执行周期结束后响应中断	每个机器周期结束均可，总线空闲时即可相应DMA请求
场景	CPU控制，低速设备	DMA控制器控制，高速设备
优先级	优先级低于DMA	优先级高于中断
异常处理	能处理异常事件	仅传送数据