[概念 2](#_Toc507759015)

[超流水线处理机 4](#_Toc507759016)

[开发并行性 7](#_Toc507759017)

[循环展开和指令调度 8](#_Toc507759018)

[指令级并行 10](#_Toc507759019)

[流水线 11](#_Toc507759020)

[CPU结构 13](#_Toc507759021)

[时序信号发生器 13](#_Toc507759022)

[功能部件 16](#_Toc507759023)

[指令系统 17](#_Toc507759024)

[指令集结构 17](#_Toc507759025)

[寻址方式 19](#_Toc507759026)

[功能设计 22](#_Toc507759027)

[控制指令 23](#_Toc507759028)

[操作数 25](#_Toc507759029)

[指令格式 26](#_Toc507759030)

[I/O系统 27](#_Toc507759031)

[总线 29](#_Toc507759032)

## 概念

|  |  |
| --- | --- |
| 多层次结构 | 按计算机语言从低级到高级：  微程序机器级、机器语言、操作系统虚拟机、汇编语言虚拟机、高级语言虚拟机、应用语言虚拟机 |
| 虚拟机 | 用于软件实现的机器 |
| 解释 | 从高一级机器上的程序中取下一条语句或指令，执行在低一级机器上对应的一段等效程序 |
| 翻译 | 先用转换程序把高一级机器上的程序转换为低一级机器上的等效程序，然后在低一级上运行，实现程序功能 |
| 计算机系统结构 | 机器语言程序员所看到的的计算机树形，即概念结构与功能特性 |
| 计算机组成 | 计算机系统结构的逻辑实现，包含物理机器级中的数据流和控制流的组成以及逻辑设计等 |
| 计算机实现 | 计算机组成的物理实现，包括处理机、主存部件的物理结构，期间的集成度和速度，模块、插件、底板的划分与连接，信号传输，电源、冷却、整机装配技术 |
| 透明 | 把本来存在的事物或属性，但从某种角度看又好像不存在的特性 |
| 系列机 | 由同一厂家生产的具有相同系统结构、但具有不同的组成和实现的一些列不同型号的计算机 |
| 最大并行度 | 计算机系统在单位时间内能够处理的最大二进制位数 |
| Amdahl定律 | 当对一个系统中的某个部件进行改进后，所能获得的整个系统性能的提高，受限于该部件的执行时间占总执行时间的百分比。 |
| CPI | 每条指令的平均执行的时间周期数 |
| 局部性原理 |  |
| CISC | 复杂指令集计算机 |
| RISC | 精简指令集计算机 |
| 软件兼容 | 一个软件可以不经过修改或者只需要很少量的修改就可以由一台计算机一直到另一台计算机上运行，差别只是执行时间的吧冉，则这两台计算机的软件就是兼容的 |
| 向上兼容 | 按某档计算机编制的程序，不佳修改就能运行与比它高档的计算机 |
| 向下兼容 | 按某档计算机编制的程序，不佳修改就能运行与比它低档的计算机 |
| 向前兼容 | 按某个时期投入市场的某种型号计算机编制的程序，不佳修改就能运行于之后投入市场的计算机 |
| 兼容机 | 由不同制造商生产的具有相同系统结构的计算机 |
| 摩尔定律 | 集成电路芯片上所继承的集体管数目每个18个月就翻一番 |
| 模拟 | 用软件方法在一台现有计算机(宿主)上实现另一台计算机(目标机)的指令系统 |
| 仿真 | 用一台现有计算机上的微程序(宿主)去解释实现另一台计算机(目标机)的指令系统 |
| 并行性 | 计算机系统在同一时刻或者同一时间间隔内进行多种运算或操作。是要在时间上相互重叠，就存在并行性，包括同时性与并发性两种含义 |
| 并发性 | 多个事件在同一时间间隔内发生 |
| 同时性 | 多个事件在同一时刻发生 |
| 时间重叠 | 多个处理过程在时间上相互错开、轮流、重叠地使用同一套硬件设备的各个部分，一加快硬件周转赢得速度 |
| 空间重叠 | 重复设置硬件资源，大幅提高计算机系统性能 |
| 资源共享 | 软件方法，是多个任务按一定时间顺序轮流使用同一套硬件设备 |
| 耦合度 | 反映多级系统中各计算机之间物理连接的紧密程度和交互作用能力的强弱 |
| 紧密耦合系统  (直接耦合系统) | 计算机之间的物理连接的带宽较大，一般通过总线或告诉开关互连，可以共享主存 |
| 松散耦合系统  (间接耦合系统) | 通过通道或通道线路实现计算机之间的互连，可以共享外存设备如磁盘磁带，各计算机之间的相互作用是在文件或数据集一级上进行的 |
| 异构型多处理机系统 | 多个不同类型、至少担负不同功能的处理机组成，按照作业要求的顺序，利用时间重叠原理，一次对多个任务进行加工各自完成规定的功能操作 |
| 同构型多处理机系统 | 多个不同类型或至少担负同等功能的处理机组成，同时处理同意作业中能并行执行的多个任务 |
|  |  |
| 指令集完整性 | 在一个有限的存储空间内，对于任何可接额问题，编制计算程序时，指令级所提供的指令足够使用 |
| 指令集规整性 | 没有或尽可能减少例外情况和特殊的应用，所欲运算都能对称、均匀地在存储器单元或寄存器单元之间进行 |
| 数据表示 | 计算机硬件结构能够识别，指令集可以直接调用 的数据类型 |
| PC相对地址 | 指令中提供一个偏移量 |

## 超流水线处理机

1.将每个流水段进一步细分，这样在一个时钟周期内能够分时流出多条指令。这种处理机称为超流水线处理机。

2.对于一台每个时钟周期能流出n条指令的超流水线计算机来说，这n条指令不是同时流出的，而是每隔1/n个时钟周期流出一条指令。

3.实际上该超流水线计算机的流水线周期为1/n个时钟周期。

一台每个时钟周期分时流出两条指令的超流水线计算机的时空图。

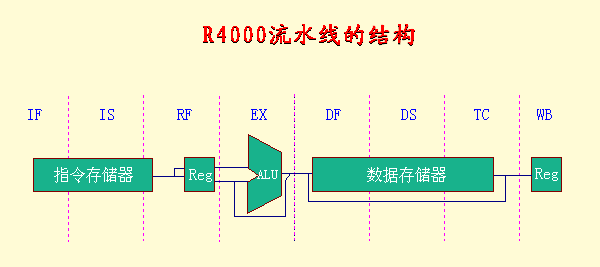


4.在有的资料上，把指令流水线级数为8或8以上的流水线处理机称为超流水线处理机。

典型的超流水线处理器：SGI公司的MIPS系列R4000

|  |  |
| --- | --- |
| 2个Cache | 指令Cache和数据Cache  容量都是8 KB  每个Cache的数据宽度为64 b |
| 核心处理部件  整数部件 | 一个32×32位的通用寄存器组  一个算术逻辑部件（ALU）  一个专用的乘法/除法部件 |
| 浮点部件 | 1.一个执行部件：  浮点乘法部件  浮点除法部件  浮点加法/转换/求平方根部件(它们可以并行工作)  2.一个16×64位的浮点通用寄存器组。浮点通用寄存器组也可以设置成32个32位的浮点寄存器。 |

8级流水线：



各级的功能

IF：取指令的前半步，根据PC值去启动对指令Cache的访问。

IS：取指令的后半步，在这一级完成对指令Cache的访问。

RF：指令译码，访问寄存器组读取操作数，冲突检测，并判断指令Cache是否命中。

EX：指令执行。包括有效地址计算，ALU操作，分支目标地址计算，条件码测试。

DF：取数据的前半步，启动对数据Cache的访问。

DS：取数据的后半步，在这一级完成对数据Cache的访问。

TC：标识比较，判断对数据Cache的访问是否命中。

WB：load指令或运算型指令把结果写回寄存器组。

MIPS R4000指令流水线时空图



载入延迟为两个时钟周期



## 开发并行性

指令之间存在的潜在并行性

在流水线思想的基础上，进一步扩展，开发出更多的指令级并行性

1.基于硬件的动态开发

2.基于软件的静态开发

CPI=CPI(理想)-结构冲突停顿-数据冲突停顿-控制冲突停顿

理想CPI是衡量流水线最高性能的一个指标

基本程序块：一串连续的代码，没有分支指令和转入点，即这段代码必定是从头开始执行到结尾统计指出，一般程序平均每5～7条指令就会有一个分支，因此在基本程序块中开发出的并行性很有限，需要跨越多个基本块开发ILP

|  |  |  |
| --- | --- | --- |
| 循环级  并行性 |  | 是最简单最常用的方法  使一个循环中的不同循环体并行执行  例如，考虑下述语句：  for （i=1； i<=500； i=i＋1）  a[i]=a[i]＋s；  每一次循环都可以与其他的循环重叠并行执行；  在每一次循环的内部，却没有任何的并行性 |
| 方法 | 循环展开 |
| 采用向量指令和向量数据表示 |

## 循环展开和指令调度

基本方法

1.充分开发指令之间存在的并行性，找出不相关的指令序列，让它们在流水线上重叠并行执行。

2.增加指令间并行性最简单和最常用的方法

开发循环级并行性——循环的不同迭代之间存在的并行性。

在把循环展开后，通过重命名和指令调度来开发更多的并行性。

3.编译器完成这种指令调度的能力受限于两个特性：

程序固有的指令级并行性；流水线功能部件的执行延迟。

4.本节中，我们使用的浮点流水线延迟为：

|  |  |  |
| --- | --- | --- |
| 产生结果的指令 | 使用结果的指令 | 延迟（时钟周期数） |
| 浮点计算 | 另一个浮点计算 | 3 |
| 浮点计算 | 浮点store（S.D） | 2 |
| 浮点load（L.D） | 浮点计算 | 1 |
| 浮点load（L.D） | 浮点store（S.D） | 0 |

假设采用第3章的5段整数流水线：

分支的延迟：1个时钟周期。

整数load指令的延迟：1个时钟周期。

整数运算部件是全流水或者重复设置了足够的份数。

循环展开和指令调度时要注意以下几个方面：

1.保证正确性：在循环展开和调度过程中尤其要注意两个地方的正确性：循环控制，操作数偏移量的修改。

2.注意有效性：只有能够找到不同循环体之间的无关性，才能有效地使用循环展开。

3.使用不同的寄存器：否则可能导致新的冲突

4.删除多余的测试指令和分支指令，并对循环结束代码和新的循环体代码进行相应的修正

5.注意对存储器数据的相关性分析：例如：对于load指令和store指令，如果它们在不同的循环迭代中访问的存储器地址是不同的，它们就是相互独立的，可以相互对调。

5.注意新的相关性：由于原循环不同次的迭代在展开后都到了同一次循环体中，因此可能带来新的相关性。

静态超标量处理机中的循环展开

## 指令级并行

|  |  |  |
| --- | --- | --- |
| 概念 | 实现多个指令并行执行  一个指令的执行过程分解为若干子过程(指令周期)，每个子过程由各自对应的独立部件完成执行；  某时刻，多个指令并行执行，每个指令占据一个功能部件 | |
| 方法 | 部件冗余 | 相同的部件重复设置 |
| 流水线 | 一个指令的执行过程分解为若干子过程(指令周期)，  每个子过程(流水段)由各自对应的独立部件完成执行  不同指令的子过程，若使用不同的部件，即可同时执行 |

|  |  |  |
| --- | --- | --- |
| 指令  调度 | 通过在编译时让编译器重新组织指令书序或通过硬件在执行时调整指令顺序来消除冲突 | |
| 静态 | 编译器调度指令的执行顺序，减少相关指令出现冲突  在编译期间进行代码的调度和优化 |
| 动态 | 在保持数据流和异常行为的情况下，通过硬件对指令执行顺序进行重新安排，减少数据相关导致的停顿 |

## 流水线

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 对象 | 运算流水线(部件级) | | | | 如浮点加法=求阶差+对阶+尾数相加+规格化 | |
| 指令流水线(处理机级) | | | | 如指令周期=取指+译码+执行 | |
| 宏流水线(处理机间) | | | | 多个处理机共同处理一个任务，每个处理机完成一部分 | |
| 深度 | 流水线深度=子过程个数=使用的功能部件个数  深度上限：流水线比非流水线更加复杂，有更多开销；增加深度一方面提高流水线性能，另一方面增加开销，深度达上限后，开销比重超过流水线带来的性能提升，流水线失去意义 | | | | | |
|  | 单功能 | | 只能完成一种固定功能，流水线各段之间的连接固定不变 | | | |
| 多功能 | | 流水线各段不同的连接方式实现不同的功能 | | | |
| 静态 | 某时刻，流水线只实现一种功能，处于该流水线的所有任务都实现该功能  要按另一种连接实现其他功能时，必须等按原来连接的所有任务都执行完毕后，才能改变连接 | | |
| 动态 | 某时刻，流水线可以执行不同功能 | | |
|  | 线性 | | 无反馈回路，执行过程中每个段最多只执行一次 | | | |
| 非线性 | | 有反馈回路，有些段执行多次，有些段被跳过不执行，用于：递归或多功能流水线 | | | |
|  | 顺序 | | 多个任务进入流水线，流出顺序与流入顺序相同 | | | |
|  | 乱序 | | 多个任务进入流水线，流出顺序与流入顺序可以不同 | | | |
| 方法 | 调度问题 | | 确定什么时候向流水线引进新的任务，才能使该任务不会与先前进入流水线的任务发生冲突(流水段争用) | | | |
| 静态开发 | | 基于软件 | | | |
| 动态开发 | | 基于硬件 | | | |
| 稳定状态 | 流水线满载，k段流水线在第k个时钟周期后，进入稳定状态 | | | | | |
| 通过时间 | 第一个任务进入流水线的时间，该时间段流水线不满载 | | | | | |
| 排空时间 | 最后一个任务流出的时间，该时间段流水线不满载 | | | | | |
| 吞吐率 | TP=指令数目/完成时间 | | | | TP(最大)=稳定状态下的吞吐率  TP(实际)=考虑建立时间和排空时间、以及其他相关因素 | |
| 加速比 | S=TP(实际)/TP(非流水线) | | | | | |
| 效率 | E=(++…)/深度，=第i段效率=第i段实际使用的时间/总时间 | | | | | |
| 瓶颈问题 | 举例 | 1(50ns)→2(50ns)→3(100ns)→4(200ns) | | | | |
| 瓶颈 | 执行时间较长的流水段成为瓶颈，如上例中的流水段3和4  时钟周期=最长流水段执行时间；较短流水段没有完全利用一个时钟周期，造成浪费 | | | | |
| 措施 | 细分瓶颈段 | | | |  |
| 重复设置瓶颈段 | | | |  |
| 流水寄存器延迟 | 建立时间 | | 在出发写操作的时钟信号到达之前寄存器输入必须保持稳定的时间 | | | |
| 传输延迟 | | 时钟信号到达后到寄存器输出可用的时间 | | | |
| 时钟偏移  开销 | 流水线中始终到达各个流水寄存器的最大差值时间 | | | | | |
| 冲突问题 | 流水线中各个任务之间可能存在关联 | | | | | |

在程序执行过程中，依靠专门的硬件对代码进行调度，减少数据相关导致的停顿

优点：

能够处理一些在编译时情况不明的相关（比如涉及存储器访问的相关），并简化了编译器；

能够使本来是面向某一流水线优化编译的代码在其他的流水线（动态调度）上也能高效地执行。

缺点：

硬件复杂

|  |  |  |  |
| --- | --- | --- | --- |
| 按指令输入流水线的顺序，先输入的是前驱指令，后输入的是后继指令  简单流水线局限性：指令按程序顺序执行，某条指令停顿，其后继指令全部停止，系统中的功能部件可能因为没有指令可处理而处于空闲，效率低下  解决办法：乱序执行，存在问题：指令冲突 | | | |
| 冲突 | | 相关 | 检测 |
| 前驱指令**某一段**必须在后继指令的**某一段**之前执行  前驱指令的执行不能收到任何后继指令的影响 | 相关的流水段执行顺序相反  实际实现与程序所指示的操作相违背 |
| 结构冲突 | | 结构相关：两指令使用同一功能部件  如运算器、存储器 | 数据和指令放在同一个存器  某时刻，指令A访问数据，指令B取指 |
| 数据冲突 | RAW  写后读 | 数据相关：两指令访问同一寄存器或存储单元  前驱指令写，后继指令读 | **先**后继指令读，**再**前驱指令写  那么后继指令读到的是错误值 |
| WAW  写后写 | 名相关-反相关：两指令访问同一寄存器或存储单元  前驱指令写，后继指令写 | **先**后继指令写，**再**前驱指令写  那么结果留下的是前驱指令的结果值 |
| WAR  读后写 | 名相关-输出相关：两指令访问同一寄存器或存储单元  前驱指令读，后继指令写 | **先**后继指令写，**再**前驱指令读  那么前驱指令读到错误值 |
| 控制冲突 | | 控制相关：相邻两指令，前驱指令是分支指令  后继指令需要根据前驱指令的分支结果来确定是否执行 | **先**后继指令执行，**再**分支之令执行 |
| 解决冲突的方法：  停顿气泡：后继指令中发生冲突的流水段推迟一个指令周期再执行  冗余部件：设置两个存储器分别存放数据和指令，避免结构冲突  指令预取：访存时间很短时，例如执行周期取数时间很短，执行指令时主存会有空闲，取出下一条指令，并将它暂存起来；  等执行部件空闲，将暂存的指令传给执行部件执行  只要指令队列空出，就可以取下一条指令，并放至空出的指令队列中，从而保证在执行第K条指令的同时对第K+1条指令进行译码，实现“执行K”与“分析K+1”重叠 | | | |

# CPU结构

## 时序信号发生器

同步时序控制

|  |  |
| --- | --- |
| 指令周期 | CPU取出并执行一条指令的全部时间  指令周期划分：取指周期，间址周期，执行周期，中断周期  取指周期和执行周期必须包含，另外两个随情况而定，  冯诺依曼计算机根据指令周期的不同阶段来区分从存储器取出的是指令还是数据，更准确地说：  计算机在不同的阶段对存储器进行读操作取出的代码，有不同的用处。  取指周期取出的是指令，执行周期取出的是数据  控制器可以区分存储单元中存放的是指令还是数据，存储器本身无法区分存储单元中存放的是指令还是数据，  程序是由指令组成的，程序的执行其实就是指令的顺序执行： |
| 机器周期 | 把一个指令划分成多个操作，机器周期是执行其中每一步完整操作所需要的时间  具体的划分方法则要根据该指令系统中的各个指令的执行步骤  目的是要使得机器周期作为所有指令执行过程中的一个基准时间，这样就可以用“x个机器周期”来描述一个指令的执行时间    以存取周期作为基准时间，即内存中读取一个指令字的最短时间作为机器周期。  把通过一次总线事物访问一次主存或I/O的时间定为一个机器周期  如：设定机器周期是取一次指令所用的时间，那么：  指令字长等于存储字长的2倍的指令，它的取指周期等于机器周期的2倍；  如果指令字长等于存储字长，取值周期等于机器周期 |
| 多级时序系统 | 机器周期、时钟周期组成了多级时序系统  一般来说，CPU的主频越快，机器的运行速度也越快。  在机器周期所含时钟周期数相同的前提下，两级平均指令执行速度比等于两机主频之比  如CPU主频是8MHZ，那么平均指令执行速度是0.8MIPS  若要得到平均指令执行速度是0.4MIPS的机器，则只需要用主频是8×0.4/0.8=4MHZ的CPU即可    实际上机器的速度不仅和主频有关，还与机器周期中所含的时钟周期数以及指令周期中所含的机器周期数有关  同样朱批判的机器，由于机器周期所含有的时钟周期数不同，运行速度也不同  机器周期所含时钟周期数少的机器，速度更快 |

CPI：执行一条指令平均需要的时钟周期数

单周期处理中即指所有指令的指令周期为一个时钟周期，那么每条指令的CPI=1

要考虑比较慢的指令，所以处理器的时钟频率较低

同步控制方式

任何一条指令或指令中任何一个微操作的执行都是事先确定的，并且都是受统一基准时标的时序信号所控制的方式

每个机器周期包含的时钟周期数是相同固定的

如果机器内的存储器存取周期不统一，那么只有把最长的存取周期作为机器周期，才能同步控制，否则取指令和取数时间不同，无法用同一的标准。

又如有些不访存的指令，执行周期的微操作很少，无须这个固定的节拍数所指定的那么多节拍

因此为了提高CPU效率，在同步控制中又有三种方案

|  |  |
| --- | --- |
| 定长机器周期 | 不论指令所对应的微操作序列由多长，也不管微操作的简繁，一律一最长的微操作序列和最复杂的微操作作为标准，  采取完全同一的，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各种不同的指令  这种方案对于微操作序列较短的指令来说，造成时间上的浪费 |
| 不定长机器周期 | 每个机器周期内的节拍数可以不相等  这种控制方式可以解决微操作执行时间不统一的问题。通常把大多数微操作安排在一个较短的机器周期内完成，而对某些复杂的微操作，采用延长机器周期或增加节拍的办法来解决 |
| 中央控制和局部控制相结合 | 将机器的大部分指令安排在统一的、较短的机器周期内完成，称为中央控制  将少数操作复杂的指令中的某些操作，如乘法和浮点运算等，采用局部控制方式  注意：  1.使局部控制的每一个节拍的宽度与中央控制的节拍宽度相同  2.将局部控制节拍作为中央控制中机器节拍的延续，插入到中央控制的执行周期内，是机器已同样的节奏 工作，保证了局部控制和中央控制的同步  的多少克根据情况而定，对于乘法，当操作数位数固定后，的个数也就确定了。而对于浮点运算的对接操作，由于移位次数不是一个固定值，因此的个数不能事先确定    以乘法指令为例  第一个机器周期采用中央控制的节拍控制取指令的操作，接着仍然用中央控制的、、节拍去完成将操作数从存储器总取出并送至寄存器的操作，然后转局部控制，用局部控制节拍完成重复加和移位的操作 |

假设机器采用同步控制，每个机器周期为3个节拍，

1.有些微操作的次序是不容改变的，故安排微操作节拍必须注意微操作的先后顺序

2.凡是被控制对象不同的微操作，若能在一个节拍内执行，应该尽可能地安排在同一个节拍内，以节省时间

3.若微操做所占时间不长，应该将它们安排在一个节拍内完成，并且允许这些微操作有先后次序

|  |  |
| --- | --- |
| 取值周期 | 根据原则2，节拍，安排两个微操作：PCàMAR，1àR  根据原则2，节拍，安排两个微操作：M(MAR)àMAR，(PC)+1àPC  节拍：安排MDRàIR，考虑到指令译码时间较短，根据原则3，可将指令译码OP(IR)àID也安排在节拍  实际上，(PC)+1àPC操作也可以安排在节拍，因为一旦PCàMAR后，PC的内容就可以修改 |
| 间址周期 | ：Ad(IR)àMAR，1àR  ：M(MAR)àMDR  ：MDRàAd(IR) |
| 执行周期 | 本例的执行周期只有一个微操作，按照同步的原则，这个微操作可以安排在的任意一个节拍，其余节拍为空  ：空  ：空  ：MDRàAd(IR) |
| 中断周期 | ：Ad(IR)àMAR，1àR  ：M(MAR)àMDR  ：MDRàAd(IR) |

其他时序控制方式

|  |  |
| --- | --- |
| 异步控制方式 | 不存在基准时标信号，没有固定的周期节拍和严格的时钟同步，执行每条指令和每个微操作需要多少时间就占用多少时间。  这种方式微操作的时序由专门的应答线路控制，即当CU发出执行某一微操作的控制信号后，等待执行部件完成了该操作后发送“回答”信号给CU，CU再开始新的微操作，  这样CPU就没有空闲状态  但因为需要采用各种应答电路，其结构复杂 |
| 联合控制方式 | 同步控制和异步控制相结合  对各种不同指令的微操作实行大部分同一，小部分区别对待的办法。  如：对每条指令都有的取值操作，采用同步方式控制  对那些时间难以确定的微操作，如I/O操作，则采用异步控制，以执行部件送回的“回答”信号作为本次微操作的结束 |
| 人工控制方式 | 为了调机和软件开发的需要，在机器面板或内部设置一些开关或按键，来达到人工控制的目的    复位键Reset  按下Reset键，是计算机处于初始状态。  当机器出现死锁状态或无法继续运行的时候，可按此键  若在机器运行的时候按此键，将会破坏机器内某些状态而引起错误，因此要慎用  有些微型计算机没有此按键，可采用停电再上电的方法    连续执行和单条执行，转换开关  由于调机的需要，有时需要观察执行完一条指令后的机器状态，有时有需要观察连续运行程序后的结果，设置该开关，提供两种选择    符合停机开关  这组开关指示存储器位置，当程序运行到与开关指示的地址相符时，机器便停止运行 |

## 功能部件

|  |  |
| --- | --- |
| ALU | 执行算术、逻辑、比较运算，并将运算结果存入RF  ALU的运算结果可以作为访问数据存储器DM的地址，将数据存储器单元内容读出送入RF，或者将RF的内容写入DM  ALU是一个组合逻辑电路，故其运算过程中必须保持两个输入端的内容不变。  单总线CPU中，为了得到两个不同的操作数，ALU的一个输入端与总线相连，另一个输入端需要通过一个寄存器与总线相连。此外，ALU的输出端也不能和内部总线相连，否则其输出又会通过总线反馈到输入端，影响运算结果，故输出端需要通过一个暂存器与总线相连 |

|  |  |  |
| --- | --- | --- |
| Add | 加法器 | 用于程序执行时，PC值的修正  PC与指令中地址偏移量的相加 |
| SigExit16/32 | 符号扩展单元 |  |
| MUX | 二路选择器 |  |

控制器

|  |  |
| --- | --- |
| 功能 | 产生一组定时信号，其中主时钟信号是最基本的定是信号，除此以外，可以对主时钟信号进行分频，产生其他不同层次的时钟信号  产生执行每条指令，所需要的全部控制信号  响应中断请求 |
| 组成 | 控制寄存器和译码器  时序信号产生器  控制单元 |
| 控制单元 | 有三种设计方法，区别在于如何产生微操作 (执行特定操作) 的控制信号 |

# 指令系统

## 指令集结构

根据CPU中用来存储操作数的存储单元的类型，分为：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 不同的指令集结构，操作数位置、个数、给出方式也会不同  显式给出：用指令字中的操作数字段给出  隐式给出：使用事先约定好的单元 | | | | |
|  | 操作数  位置 | 给出  方式 |  |  |
| 堆栈  结构 | 堆栈 | 隐式 | push A  push B  add  pop C | 指令字短，程序占用空间小  不能随机访问堆栈，难以生成有效代码  对栈顶的访问是个瓶颈。 |
| 累加器  结构 | 累加器 | 隐式 | load A  add B  store C | 指令字较短，程序占用空间较小  累加器作为唯一的中间结果暂存器所以需要频繁地访问存储器 |
| 存储器 | 显式 |
| 通用寄存器  结构RM | 通用寄存器  存储器 | 显式 | load R1,A  add R1,B  store R1,C | 1.寄存器访问速度比存储器快  2.对编译器而言，能更加容易、有效地分配和使用寄存器。在表达式求值方面，通用寄存器结构具有更大的灵活性和更高的效率。  例如，在一台通用寄存器结构的计算机上求表达式(A×B)-(C×D)-(E×F)，的值时  其中乘法运算可以按任意次序进行，操作数的存放也更加灵活，对流水处理也更合适，因而更高效  但是在堆栈型计算机中，该表达式的求职必须按从左到右的顺序进行。对操作数的存放也有较多的限制  3.寄存器可以用来存放变量。将变量分配给寄存器不但能减少对存储器的访问，加快程序的执行速度(因为寄存器比存储器快)，而且还可以用更少的地址位(相对于存储器地址来说)来对寄存器进行寻址，从而有效地减少程序的目标代码 |
| 通用寄存器  结构RR  (load-store) | 通用寄存器 | 显式 | load R1,A  load R2,B  add R3,R1,R2  store R3,C |

通用寄存器结构：

编译器设计者总是希望CPU内部的所有寄存器都是平等、通用的，

以前一些计算机将其一部分通用寄存器作为专用寄存器使用，导致了通用寄存器数量的减少

可以根据ALU指令操作数的两个特征来对通用寄存器型指令集结构进行进一步细分

一个特征是ALU指令的操作数个数，

3个操作数的指令：它包含两个源操作数和一个目的操作数，

2个操作数的指令：其中一个操作数即是源操作数，又是目的操作数

另一个特征：  
ALU指令中存储器操作数的个数，可以是0或1或2或3

由此总结：

|  |  |  |  |
| --- | --- | --- | --- |
| ALU指令中存储器操作数的个数 | ALU指令中操作数的最多个数 | 结构类型 | 计算机实例 |
| 0 | 3 | RR | MIPS、SPARC、Alpha  PowerPC、ARM |
| 1 | 2 | RM |  |
| 1 | 3 | RM | IBM 360/370 |
| 2 | 2 | MM | VAX |
| 3 | 3 | MM | VAX |
|  |  |  |  |

|  |  |  |
| --- | --- | --- |
| RR | 指令字长固定  指令结构简洁  是一种简单的代码生成模型  各种指令的执行时钟周期数相近 | 与RM、MM相比，指令条数多，目标代码不够紧凑，程序占用的空间大 |
| RM | 可以再ALU指令中直接对存储器操作数进行引用，而不必先用LOAD指令进行加载。  容易对指令进行编码  目标代码紧凑 | 由于有一个操作数的内容将被破坏，所以指令中两个操作数部队称。  在一条指令中同时对寄存器操作数和存储器操作数进行编码，有可能限制指令所能够表示的寄存器个数。指令的执行时钟周期数因操作数的来源(M或R)不同而差别比较大 |
| MM | 目标代码紧凑  不需要设置寄存器来保存变量 | 指令字长变化很大，特别是3操作数指令。而且每条指令完成的工作也差别很大。对存储器的频繁访问会是存储器称为瓶颈。这种类型的指令结构现在已经不用 |

一般来说，指令格式和指令字长越单一，编译器的工作就越简单，因为编译器所能做的选择比较少

如果指令集结构的指令格式和指令字长具有多样性，则可以有效地减少目标代码的大小。但是这种多样性也可能会增加编译器和CPU实现难度

另外，CPU中寄存器的个数也会影响指令的字长

通用寄存器型指令集结构比堆栈型指令结构更具优势，其中MM型已经淘汰，RR型备受青睐

## 寻址方式

|  |  |  |
| --- | --- | --- |
| 寻址：获取instruction或操作数的有效地址  形式地址A：指令字中的地址码  有效地址EA：形式地址结合寻址方式，计算出真实地址  地址A对应存储单元存储的数据：[A]  寄存器R存储的数据：[R] | | |
| Mode | A | EA |
| 隐含寻址 |  | 程序指定 |
| 立即寻址 | 操作数本身，补码表示 |  |
| 直接寻址 | 存储器地址或寄存器地址 | A |
| 相对寻址  偏移寻址 | 偏移量，可正可负，补码表示 | [PC]+A |
| 基址寻址 | 偏移量，可正可负，补码表示 | [BR]+A  BR：基址寄存器(专用或通用寄存器)  [BR]：操作系统确定，数值固定 |
| 变址寻址 | 基地址 | EA=A+[IX]  IX：变址寄存器(专用或通用寄存器)  [IX]：用户设定，数值可变 |
| 间接寻址 | 按照上述的某个寻址方式得到EA  查看EA的最高位，  EA=0，EA是有效地址  EA=1，EA是间接地址，使用(EA)作为地址码继续寻址 | |

对于偏移寻址，主要的问题是偏移量的取值范围，如果知道了该范围，就能确定指令中需要用多少位的字段来表示偏移量，这一选择直接影响指令字长

立即数寻址主要用于ALU指令、比较指令、用于给寄存器装入常数的指令等

设计指令集结构时，首先要确定是所有指令还是只有部分指令具有立即数寻址方式

需要知道立即数的取值范围，这个范围决定了指令字中立即数字段所需占用的位数，从而会影响指令的字长

至少要将立即数的大小设置为8~16位

|  |  |
| --- | --- |
| 隐含寻址 | 优点：缩短指令长度，执行速度快  缺点：需要增加存储操作数或隐含地址的硬件 |
| 立即寻址 | 优点：指令执行阶段不访问主存，执行速度快  缺点：操作数的位数限制了立即数的范围 |
| 直接寻址 | 优点：执行速度快：指令执行阶段仅访问一次主存或不访问主存，且不需要计算操作数的地址  寄存器数量少，对应的地址码位数少，指令长度短  支持向量/矩阵运算  缺点：A的位数决定了该指令数的寻址范围，操作数的地址不易修改  寄存器价格昂贵，计算机中的寄存器有限 |
| 相对寻址 | 优点：  编制浮动程序：操作数的地址不固定，随PC值的变化而变化，可以很方便地将程序装入到主存的任意区域  多道程序设计  广泛应用于转移命令 |
| 基址寻址 | 优点：扩大寻址范围  多道程序设计：用户不必考虑自己的程序存于主存的哪一空间区域  编制浮动程序 |
| 变址寻址 | 优点：扩大寻址范围  处理数组问题：设定A为数组首地址，不断改变变址寄存器IX的内容，形成数组中任意数据的地址，  编制循环程序 |
| 间接寻址 | 优点  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：执行速度慢：指令执行阶段要多次访存 |

按字寻址，程序计数器PC给出下一条指令字的访存地址(指令在内存中的地址)，故取决于存储器的字数；

指令寄存器IR用于接受取得的指令，故取决于指令字长

(PC)=(PC)+n

n=本指令占存储单元个数

|  |  |
| --- | --- |
| 顺序寻址方式 | 通过程序计数器加1，自动形成下一条指令的地址 |
| 跳跃寻址方式 | 通过转移类指令实现  下一条指令的地址码不是由程序计数器给出，而是由本条指令给出下条指令的地址  注意：是否跳跃可能受到状态寄存器和操作数的控制，  跳跃到的地址分为：  绝对地址：由标记符直接得到  相对地址：对于当前指令地址的偏移量  跳跃的结果是当前指令修改PC值，所以下一条指令仍然是通过程序计数器PC给出 |

转移指令是需要判断转移书否成功，若成功则PC修改位转移目标的地址，否则下一条指令仍然是PC自增后的地址

## 功能设计

设计一种指令集结构时，首先要考虑该指令集结构所应实现的基本功能(操作)，

由于指令集结构时系统结构的重要内容之一，所以他的功能设计实际上就是确定软硬件功能的分配，即确定哪些基本功鞥由硬件实现，哪些功能由软件实现比较合适。

硬件实现：考虑速度、成本、灵活性，特点是速度快、成本高、灵活性差

软件实现：速度慢，价格便宜，灵活性好

|  |  |  |  |
| --- | --- | --- | --- |
| 要求 | 完整性 | 功能齐全，使用方便  1.算术和逻辑运算  2.数据传输  3.程序控制  4.系统指令：在不同的指令集结构，支持程度有较大差异，但有一点事共同的，即必须对基本的系统功能调用提供指令支持  5.浮点运算  6.十进制  7.字符串  8.图形  前4类是基本指令，其中所有的指令结构一般都会拥有前3类  十进制和字符串指令在有的计算机中时以基本操作的形式出现，有的则是在编译时由编译器变换成由更简单的指令串来实现 | |
| 规整性 | 对称性 | 所有与指令集有关的存储单元的使用、操作码的设置等都是对称的，如：  使用存储单元时，所有通用寄存器同等对待  设置操作码时，如果设置了A-B的指令，那么也要设置B-A的指令 |
| 均匀性 | 对各种不同的操作数类型、字长、操作种类、数据存储单元，指令的设置都要同等对待，如  某计算机有5种数据表示，4种字长，两种存储单元，则要设置5\*4\*2=40种同一操作的指令  但这样太复杂，一般就是实现有线的规整性， |
| 高效率 | 使用频率高的指令速度快 | |
| 兼容性 |  | |

|  |  |  |  |
| --- | --- | --- | --- |
| CISC | 目标  程序 | 面向目标程序增强指令功能是提高计算机系统性能的最直接的方法，减少程序的执行时间，减少程序所占空间 | |
| 函数计算程序 | 专门的函数运算指令 |
| 向量和矩阵运算 | 成组传送数据的指令 |
| 循环 | 专门的循环控制指令：一条指令实现循环功能  循环通常需要3个指令：加法+比较+分支，循环体往往很短，因此循环指令在整个循环程序中占较大比例 |
| 高频度指令 | 高频度使用的指令，硬件执行  高频度使用的指令串，用一条新指令替代 |
| 高级语言 | 频度高、执行时间长的语句  增强功能，加快执行速度，或增加专门的指令 | |
| 一元赋值语句  属于数据传送指令，往往占比很大，因此提高数据传送指令功能 | |
| 减少系统结构中的各种例外情况 | |
| 减少编译器的翻译工作，缩小机器语言和高级语言的差距；  间接执行高级语言计算机：高级语言直接作为汇编语言  直接执行高级语言计算机：高级语言作为机器语言，直接由硬件对高级语言源程序的语句逐条进行解释进行，这时就不需要编译器了  对于微程序CPU，应增加微程序的解释工作 | |
| 操作系统 | 设置特权指令，支持操作系统(很大程度上取决于系统结构的支持)  用户不能使用，使用频率较低，但必不可少  处理机工作状态和访问方式的切换  进行的管理和切换  存储管理和信息保护  进程的同步与互斥，信号灯的管理等 | |
| 问题 | 增强指令功能，把越来越多的功能交由硬件实现，指令的数量很多  1.各种指令的使用频度相差悬殊，许多指令很少用到，而那些频度高的指令其实都是些简单的指令  2.指令集庞大(指令数≥200)，指令复杂(硬件复杂，成本高)  3.许多指令CPI很大，  4.由于指令功能复杂，规整性不好，不利于流水技术的发挥 | |
| RISC | 1.指令条数少而功能简单，确定指令集时，只选用频度高的指令，在此基础上补充一些最有用的指令，如支持操作系统和高级语言的指令  2.采用简单而又统一的指令格式，减少寻址方式，指令字长都为32位或64位  3.指令的执行在单个机器周期内完成(采用流水线机制后)  4.load-store结构：只有load/store指令才能访问存储器，其他指令都是在寄存器之间进行  5.大多数指令都采用硬件逻辑来实现  6.强调优化编译器的作用，为高级语言程序生成优化代码  7.充分利用流水线 | | |

## 控制指令

控制指令用来改变可能干支流

|  |  |
| --- | --- |
| conditional branch |  |
| jump |  |
| call |  |
| return |  |

大部分的控制指令是conditional branch，该指令的关键是如何表示分支条件

|  |  |  |  |
| --- | --- | --- | --- |
| 条件码CC | 检测有ALU操作设置的一些特殊的位，即CC位 | 可以自由设置分支条件 | 条件码是增设状态。  而且它限制了指令的执行顺序，因为他们要保证条件码能顺利地传送给分支指令 |
| 条件寄存器 | 比较指令把比较结果放入任何一个寄存器，检测室就检测该寄存器 | 简单 | 占用了一个寄存器 |
| 比较与分支 | 比较操作是分支指令的一部分，通常这种比较是受到一定限制的 | 用一条指令，而不是两条  就能实现分支 | 当采用流水线方式时，该指令的操作可能太多，在一拍内做不完 |

在控制指令中，必须给出转移的目标地址。

在绝大多数情况下，指令中都会显式地给出目标地址，只是return指令是个例外

因为在编译时还不知道其返回地址。

指定转移目标最常用的方法是在指令中提供一个偏移量，由该偏移量和PC的值相加得到目标地址，这种寻址方式称为PC相对寻址

1.因为转移目标地址通常是离当前指令的地址很近，用相对于当前PC值的偏移来确定目标地址可以有效减少表示该目标地址所需要的位数

2.position independence：可以使代码被装载到主存的任意位置执行，减少程序连接的工作量

使用PC相对寻址时需要知道偏移量的分布情况以确定偏移量字段的位数，统计数据表明4~8位即可

## 操作数

计算机处理的数据类型多种多样：图、表、数、阵列、队列、链表、堆栈、向量、字符串、实数、整数、字符

设计时，需要研究这些数据类型中，哪些用硬件实现，哪些用软件实现，并研究实现方法

|  |  |
| --- | --- |
| data representation | 硬件可以识别、指令集可以直接调用的数据类型  是最常用、相对比较简单的、用硬件实现容易的  如：定点数(整数)、逻辑数(bool)、浮点数、字符、字符串  有些计算机还包括十进制、向量、堆栈等data representation |
| data structure | 由软件进行处理和实现的数据类型  研究的是这些数据类型的逻辑结构和物理结构之间的关系，并给出相应的计算方法  参见“数据结构与算法” |

如何确定数据表示？

原理上：计算机只要有了最简单的数据表示，即定点数，就可以通过软件实现其他所有数据类型，但软件实现效率低下，但如果所有数据类型都由硬件实现，则成本高昂

因此要权衡

表示操作数类型的方法：

|  |  |
| --- | --- |
| 由指令中的操作码指定 | 由于是在操作码中指出，所以即使是同一种运算，对于不同的操作数类型也要设置不同的指令，如整数加，浮点加，无符号数加等 |
| 给数据加上标识tag | 数据本身给出操作数类型，硬件通过识别这些标识符就能得知操作数类型，并进行相应操作  优点：简化指令集、可由硬件自动实现一致性检查和类型转换、缩小机器语言与高级语言的语义差距、简化编译器等  缺点：由于需要在执行过程中动态检测标识符，动态开销大，一般不使用这种方法 |

操作数的大小：一般是字节bit，半字16bit，字32bit，双子64bit

字符一般用ASCII表示，一个字节大小

整数一般用二进制补码表示，大小可以是字节、半字、字

浮点操作数分为单精度(字)和双精度(双字)

## 指令格式

为保证操作码的译码速度，许多计算机采用固定长度的操作码

寻址方式的表示：

1.和操作码一起编码

2.设置专门的地址描述符，针对处理器有多重寻址方式，并且指令有多个操作数，很难和操作码一起编码，为每个操作数分配一个地址描述符

指令字长应该是整数个B，便于流水线处理

许多RISC计算机采用定长指令格式

|  |  |
| --- | --- |
| 变长编码格式 | 操作码+地址描述符1+地址码1+…地址描述符n+地址码n |
| 固定长度编码格式 | 操作码+地址码1+地址码2+地址码3 |
| 混合型编码格式 |  |

# I/O系统

包括：

I/O设备

I/O设备与处理机的连接

I/O系统是计算机系统中的一个重要组成部分

完成计算机与外界的信息交换

给计算机提供大容量的外部存储器

按照主要完成的工作进行分类：

存储I/O系统（本章内容）

通信I/O系统

人们对I/O系统的作用和性能没有给予足够的重视

人们更多地关注：CPU的性能

许多人甚至认为CPU的速度就是计算机的速度。

I/O设备通常被称为外围设备。

（外围的就似乎没那么重要了）

系统的响应时间(衡量计算机系统的一个更好的指标)

从用户输入命令开始，到得到结果所花费的时间。

由两部分构成：

I/O系统的响应时间

CPU的处理时间

误区：使用多进程技术可以忽略I/O性能对系统性能的影响

多进程技术只能够提高系统吞吐率，并不能够减

少系统响应时间。

进程切换时可能需要增加I/O操作。

可切换的进程数量有限，当I/O处理较慢时，仍

然会导致CPU处于空闲状态例6.1 假设一台计算机的I/O处理时间占响应时间的10%，当I/O性能保持不变，而对CPU的性能分别提高10倍和100倍时，该计算机系统的总体性能会发生什么样的变化？

解 假设改进前程序的执行时间为1个单位时间。

如果CPU的性能提高10倍，程序的执行时间（包含I/O处理时间）减少为：

(1−10%) / 10 + 10% = 0.19

即整机性能只能提高到原来的约5倍，约50%的CPU性能被浪费在I/O处理上。

如果CPU的性能提高100倍，程序的执行时间减少为：

(1 − 10%) / 100 + 10% = 0.109

这表示整机性能只能提高约10倍，约90%的性能被浪费在没有改进的I/O处理上。

评价I/O系统性能的参数主要有：

连接特性

（哪些I/O设备可以和计算机系统相连接）

I/O系统的容量

（I/O系统可以容纳的I/O设备数）

响应时间和吞吐率等

另一种衡量I/O系统性能的方法：

考虑I/O操作对CPU的打扰情况。

即考查某个进程在执行时，由于其他进程的I/O操

作，使得该进程的执行时间增加了多少。

6.2 I/O系统的可靠性、可用性和可信性

处理器性能已经很高，人们更加关注系统可靠性。

|  |  |
| --- | --- |
| 可靠性  (Reliability) | 系统从某个初始参考点开始一直连续提供服务的能力。  用平均无故障时间MTTF来衡量 |
| 可用性  (Availability) | 系统的可用性：系统正常工作的时间在连续两次正常服务间隔时间中所占的比率。 |
| 可信性  (Dependability) | 服务的质量。即在多大程度上可以合理地认为服务是可靠的。不可以度量 |
| 失效率 | 1/MTTF  如果系统中每个模块的生存期服从指数分布，则系统整体的失效率是各部件的失效率之和。 |
| 平均失效间隔时间 | MTBF=MTTF+MTTR |

例6.2 假设磁盘子系统的组成部件和它们的MTTF如下：

（1）磁盘子系统由10个磁盘构成，每个磁盘的MTTF为1000000小时；

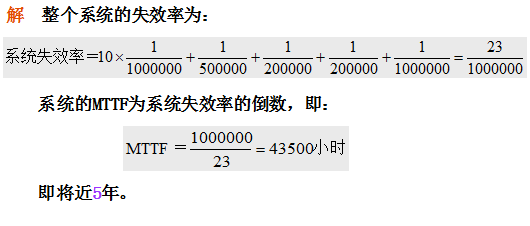
（2）1个SCSI控制器，其MTTF为500000小时；

（3）1个不间断电源，其MTTF为200000小时；

（4）1个风扇，其MTTF为200000小时；

（5）1根SCSI连线，其MTTF为1000000小时。

假定每个部件的生存期服从指数分布，同时假定各部件的故障是相互独立的，求整个系统的MTTF。



提高系统组成部件可靠性的方法

有效构建方法（valid construction）

在构建系统的过程中消除故障隐患，这样建立起来

的系统就不会出现故障。

纠错方法（error correction）

在系统构建中采用容错的方法。这样即使出现故障，

也可以通过容错信息保证系统正常工作。

|  |  |
| --- | --- |
| 分离事物总线 | 将总线事物分成请求和应答两部分  在请求和应答之间的空闲时间内，总线可以供给其他I/O使用 |
| 通道 | 专门负责整个计算机系统I/O工作的专用锤击，能执行有限的一组I/O指令 |
| 通道流量 | 一个通道在数据传送期间，单位时间内能够传送的数据量 |
| 虚拟DMA | 允许DA设备直接使用虚拟地址，并在DMA传送过程中由硬件将虚拟地址转换为物理地址 |
| 异步I/O | 允许进程在发出I/O请求后继续执行，直到该进程真正访问这些数据而它们又尚未就绪时，才被挂起 |

# 总线

在计算机系统中，各子系统之间可以通过总线互相连接。

优点：成本低、多样性

主要缺点：它是由不同的外设分时共享的，形成了信息交换的瓶颈，从而限制了系统中总的I/O吞吐量。

总线的设计

总线设计存在很多技术难点

一个重要原因：总线上信息传送的速度极大地受限于各种物理因素。

如总线的长度、设备的数目、信号的强度等，这些物理因素限制了总线性能的提高。

另外，我们一方面要求I/O操作响应快，另一方面又要求高吞吐量，这可能造成设计需求上的冲突。

设计总线时需要考虑的一些问题

|  |  |  |
| --- | --- | --- |
| 特性 | 高性能 | 低价格 |
| 总线宽度 | 独立的地址和数据总线 | 数据和地址分时  共用同一套总线 |
| 数据总线宽度 | 越宽越快（例如：64位） | 越窄越便宜（例如：8位） |
| 传输块大小 | 块越大总线开销越小 | 单字传送更简单 |
| 总线主设备 | 多个（需要仲裁） | 单个（无需仲裁） |
| 分离事务 | 采用——分离的请求包  和回答包能提高总线带宽 | 不采用——持续连接成本  更低，而且延迟更小 |
| 定时方式 | 同步 | 异步 |

分离事务总线

（又称：流水总线、悬挂总线、包交换总线）

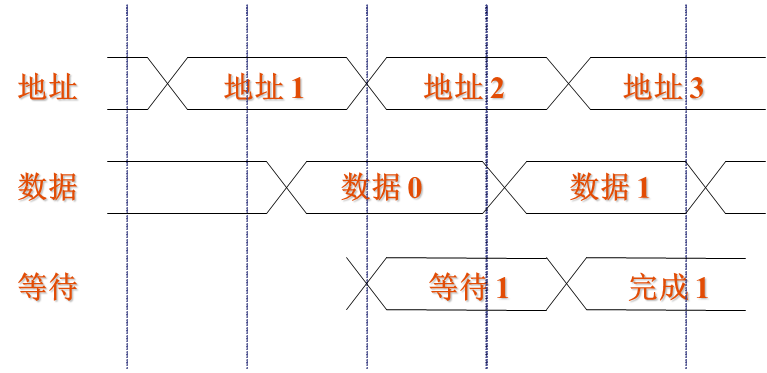
在有多个主设备时，可以通过打包技术来提高总线带宽。

基本思想

将总线事务分成请求和应答两部分。

在请求和应答之间的空闲时间内，总线可以供其他的I/O使用，这样就不必在整个I/O过程中都独占总线。

工作过程的示意图

分离事务总线有较高的带宽，但是它的数据传送延迟通常比独占总线方法大

同步总线

包含一个供总线上所有设备使用的时钟，并且这些设备是基于该时钟按照一个固定的协议来发送地址和数据的。

优点：速度快、成本低。

缺点：总线操作都必须以同样的时钟频率进行。

异步总线

没有统一的参考时钟，每个设备都有各自的定时方法。

采用握手协议

总线标准和实例

I/O总线标准：定义如何将设备与计算机进行连接的文档。

常见I/O总线的一些典型特征

几种常用并行I/O总线

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | IDE / Ultra ATA | SCSI | PCI | PCI-X |
| 数据宽度（b） | 16 | 8/16 | 32/64 | 32/64 |
| 时钟频率（MHz） | 100 | 10（Fast）  20（Ultra）  40（Ultra2）  80（Ultra3）  160（Ultra4） | 33/66 | 66/100/133 |
| 总线主设备数量 | 1个 | 多个 | 多个 | 多个 |
| 峰值带宽（MBps） | 200 | 320 | 533 | 1066 |
| 同步方式 | 异步 | 异步 | 同步 | 同步 |
| 标准 | 无 | ANSI X3.131 | 无 | 无 |

在嵌入式系统中使用较多的4种串行I/O总线的一些典型特征

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | I2C | 1-wire | RS-232 | SPI |
| 数据宽度（b） | 1 | 1 | 2 | 1 |
| 信号线数量 | 2 | 1 | 9/25 | 3 |
| 时钟频率（MHz） | 0.4～10 | 异步 | 0.04或异步 | 异步 |
| 总线主设备数量 | 多个 | 多个 | 多个 | 多个 |
| 峰值带宽（Mbps） | 0.4～3.4 | 0.014 | 0.192 | 1 |
| 同步方式 | 异步 | 异步 | 异步 | 异步 |
| 标准 | 无 | 无 | EIA, ITU-T V.21 | 无 |

在服务器系统中使用的CPU-存储器互连系统

|  |  |  |  |
| --- | --- | --- | --- |
|  | HP HyperPlane Crossbar | IBM SP | SUN Gigaplane-XB |
| 数据宽度（b） | 64 | 128 | 128 |
| 时钟频率（MHz） | 120 | 111 | 83.3 |
| 总线的主设备数 | 多个 | 多个 | 多个 |
| 每端口峰值带宽（MBps） | 960 | 1700 | 1300 |
| 总峰值带宽（MBps） | 7680 | 14200 | 10667 |
| 同步方式 | 同步 | 同步 | 同步 |
| 标准 | 无 | 无 | 无 |

与CPU的连接

I/O总线的物理连接方式有两种选择

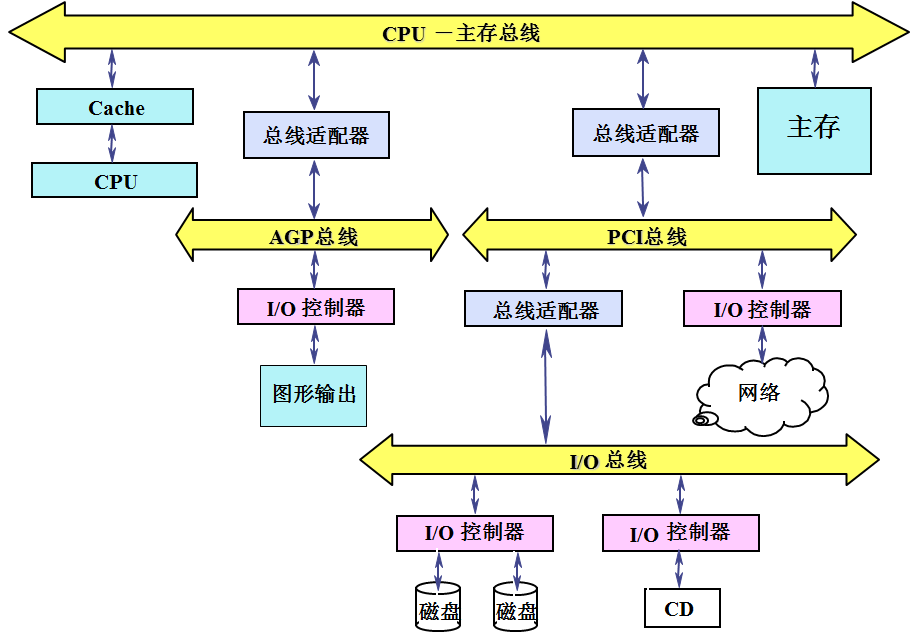
连接到存储器上

更常见

连接到Cache上

I/O总线连接到存储器总线上的方式

一种典型的组织结构



CPU对I/O设备的编址有两种方式

存储器映射I/O（也称为统一编址）

将一部分存储器地址空间分配给I/O设备，用load指令和store指令对这些地址进行读写将引起I/O设备的数据传输。

将一部分存储空间留出用于设备控制，对这一部分地址空间进行读写就是向设备发出控制命令。

给I/O设备独立编址

需要在CPU中设置专用的I/O指令来访问I/O设备。

CPU需要发出一个标志信号来表示所访问的地址是I/O设备的地址。

CPU与外部设备进行输入/输出的方式可分为4种

程序查询

中断

DMA

通道

通道处理机

通道处理机能够负担外围设备的大部分I/O工作。

通道处理机（简称通道）：专门负责整个计算机系统的输入/输出工作。通道处理机只能执行有限的一组输入/输出指令

通道的作用和功能

程序控制、中断和DMA方式管理外围设备会引起两个问题：

所有外设的输入/输出工作均由CPU承担，CPU的计算工作经常被打断而去处理输入/输出的事务，不能充分发挥CPU的计算能力。

大型计算机系统的外设虽然很多，但同时工作的机会不是很多。

解决上述问题的方法：采用通道处理机

一个典型的由CPU、通道、设备控制器、外设构成的4级层次结构的输入/输出系统。

通道的功能

接收CPU发来的I/O指令，并根据指令要求选择指定的外设与通道相连接。

执行通道程序

从主存中逐条取出通道指令，对通道指令进行译码，并根据需要向被选中的设备控制器发出各种操作命令。

给出外设中要进行读/写操作的数据所在的地址

如磁盘存储器的柱面号、磁头号、扇区号等。

给出主存缓冲区的首地址

该缓冲区存放从外设输入的数据或者将要输出到外设中去的数据。

控制外设与主存缓冲区之间的数据传送的长度

对传送的数据个数进行计数，并判断数据传送是否结束。

指定传送工作结束时要进行的操作

例如：将外设的中断请求及通道的中断请求送往CPU等。

检查外设的工作状态是否正常，并将该状态信息送往主存指定单元保存。

在数据传输过程中完成必要的格式变换

例如：把字拆分为字节，或者把字节装配成字等

通道的主要硬件

寄存器

数据缓冲寄存器

主存地址计数器

传输字节数计数器

通道命令字寄存器

通道状态字寄存器

控制逻辑

分时控制

地址分配

数据传送、装配和拆分等

通道对外设的控制通过输入/输出接口和设备控制器进行

通道与设备控制器之间一般采用标准的输入/输出接口来连接。

通道通过标准接口把操作命令送到设备控制器，设备控制器解释并执行这些通道命令，完成命令指定的操作。

设备控制器能够记录外设的状态，并把状态信息送往通道和CPU。

通道的工作过程

通道完成一次数据输入/输出的[工作过程](6-5-1通道完成一次数据传输的过程.exe)

在用户程序中使用访管指令进入管理程序，由管理程序生成一个通道程序，并启动通道。

用户在目标程序中设置一条广义指令，通过调用操作系统的管理程序来实现。

管理程序根据广义指令提供的参数来编制通道程序。

启动输入/输出设备指令是一条主要的输入/输出指令，属于特权指令。

工作流程





通道处理机执行通道程序，完成指定的数据输入/输出工作。

通道处理机执行通道程序与CPU执行用户程序是并行的。

通道程序结束后向CPU发中断请求。

[CPU执行程序和通道执行通道程序的时间关系](6-5-2通道程序、用户程序和管理程序之间的关系.exe)



6.5.3 通道的种类

根据信息传送方式的不同，将通道分为三种类型

字节多路通道

选择通道

数组多路通道

[三种类型的通道与CPU、设备控制器和外设的连接关系](6-5-3三种类型的通道与CPU、设备控制器和外围设备的连接关系.exe)



字节多路通道

为多台低速或中速的外设服务。

以字节交叉的方式分时轮流地为它们服务。

字节多路通道可以包含多个子通道，每个子通道连接一台设备控制器。

选择通道

为多台高速外围设备服务。

在一段时间内只为一台高速外设独占使用。

选择通道的硬件

5个寄存器

数据缓冲寄存器、设备地址寄存器、主存地址计数器、交换字节数计数器、设备状态/控制寄存器

格式变换部件

用于在主存和设备之间进行字与字节的拆分和装配

通道控制部件

数组多路通道

适用于高速设备。

每次选择一个高速设备后传送一个数据块，轮流为多台外围设备服务

数组多路通道之所以能够并行地为多台高速设备服务，是因为虽然其所连设备的传输速率很高，但寻址等辅助操作时间很长

通道中的数据传送过程与流量分析

通道流量

一个通道在数据传送期间，单位时间内能够传送的数据量。所用单位一般为Bps。

又称为通道吞吐率、通道数据传输率等。

通道最大流量

一个通道在满负荷工作状态下的流量 。

参数的定义

TS：设备选择时间。从通道响应设备发出的数据传送请求开始，到通道实际为这台设备传送数据所需要的时间。

TD：传送一个字节所用的时间。

p：在一个通道上连接的设备台数，且这些设备同时都在工作。

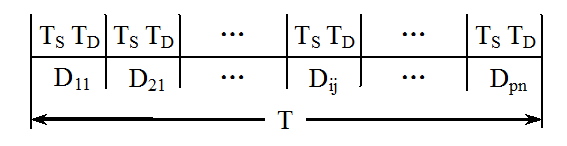
n：每台设备传送的字节数，这里假设每台设备传送的字节数都相同。

k：数组多路通道传输的一个数据块中包含的字节数。在一般情况下，k<n。对于磁盘、磁带等磁表面存储器，通常k=512。

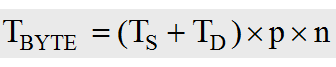
T：通道完成全部数据传送工作所需要的时间。

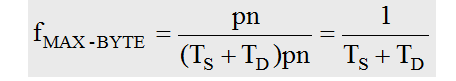
字节多路通道

[数据传送过程](6-5-4字节多路通道的数据传送过程.exe)

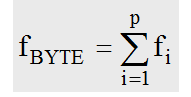


通道每连接一台个外设，只传送一个字节，然后又与另一台设备连接，并传送一个字节。

p台设备每台传送n个数据总共所需的时间为

最大流量

实际流量是连接在这个通道上的所有设备的数据传输率之和。

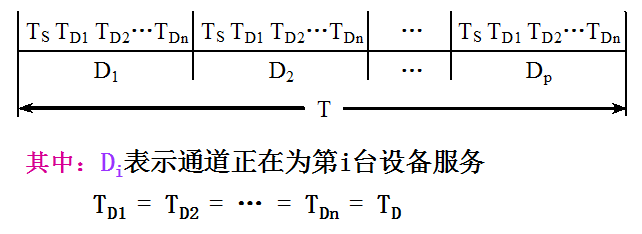


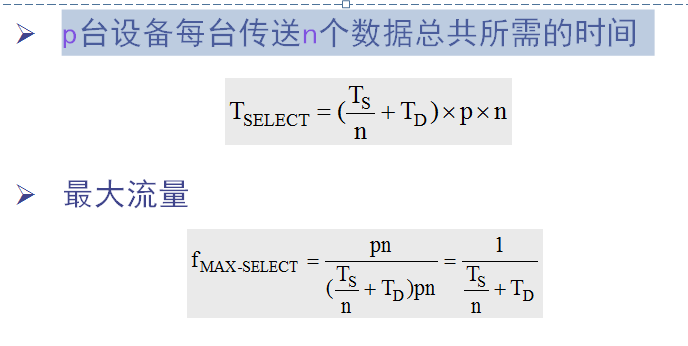
fi：第i台设备的实际数据传输率

选择通道

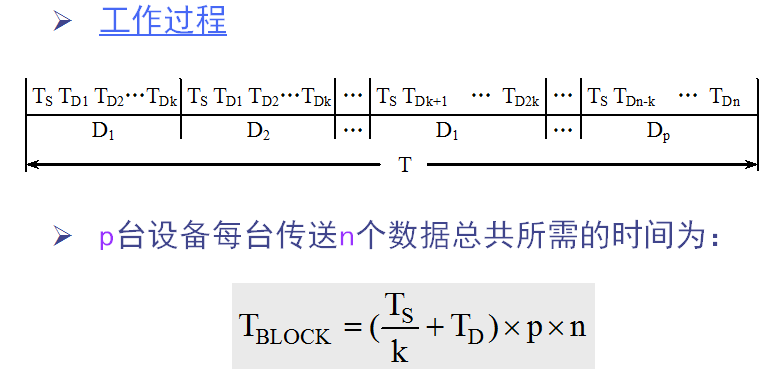
在一段时间内只能单独为一台高速外设服务，当这台设备的数据传送工作全部完成后，通道才能为另一台设备服务。

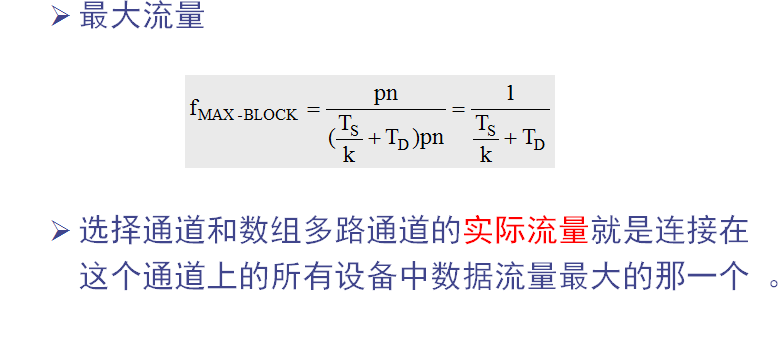
[工作过程](6-5-6选择通道的数据传送过程.exe)

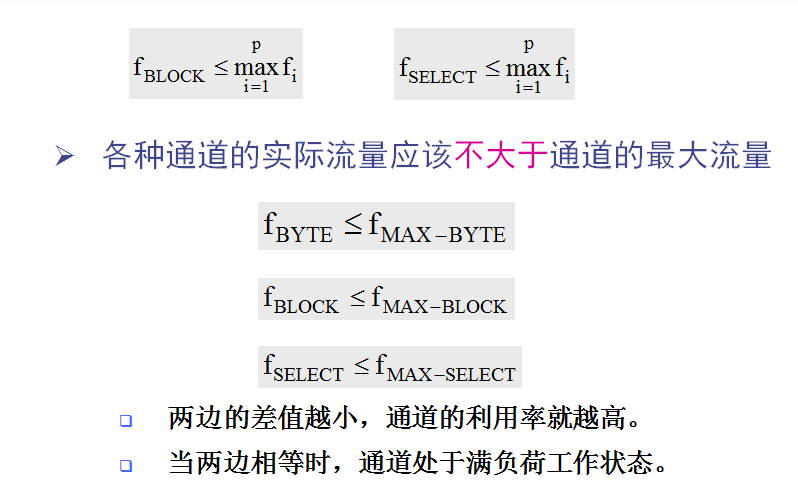




数组多路通道







I/O与操作系统

设计I/O系统需要注意操作系统的因素。

在用硬件实现的I/O技术中，哪些会实际被采用，是由操作系统来决定的。

I/O操作主要是在外设和存储器之间进行，所以操作系统必须保证这些I/O操作的安全性。

DMA和虚拟存储器

DMA是使用虚拟地址还是物理地址？

使用物理地址进行DMA传输，存在以下两个问题：

对于超过一页的数据缓冲区，由于缓冲区使用的页面在物理存储器中不一定是连续的，所以传输可能会发生问题。

如果DMA正在存储器和缓冲区之间传输数据时，操作系统从存储器中移出（或重定位）一些页面，那么，DMA将会在存储器中错误的物理页面上进行数据传输。

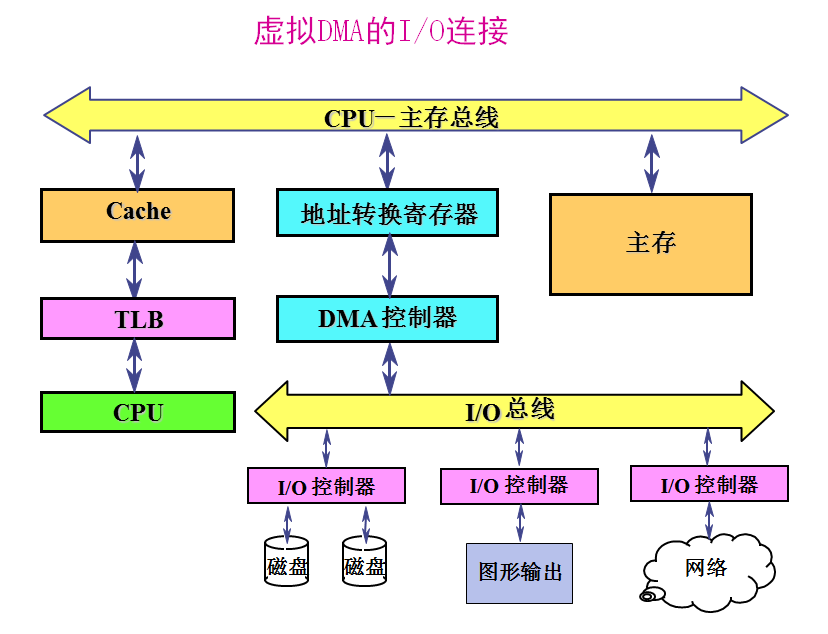
解决这些问题的方法

使操作系统在I/O的传输过程中确保DMA设备所访问的页面都位于物理存储器中，这些页面被称为是钉在了主存中。

“虚拟DMA”技术

允许DMA设备直接使用虚拟地址，并在DMA期间由硬件将虚拟地址转换为物理地址。

在采用虚拟DMA的情况下，如果进程在内存中被移动，操作系统应该能够及时地修改相应的DMA地址表。



I/O和Cache数据一致性

Cache会使一个数据出现两个副本：

一个在Cache中，另一个在主存中。

I/O设备可以修改存储器中的内容

把I/O连接到存储器上

会出现以下情况：

CPU修改了Cache的内容后，由于存储器的内容跟不上Cache内容的变化，I/O系统进行输出操作时所看到的数据是旧值。（写直达Cache没有这样的问题）

I/O系统进行输入操作后，存储器的内容发生了变化，但CPU在Cache中所看到的内容依然是旧值。

举例：假设Cache采用写回法，并且A’是A的副本，B’是B的副本。



把I/O直接连接到Cache上

不会产生由I/O导致的数据不一致的问题。

所有I/O设备和CPU都能在Cache中看到最新的数据。

I/O会跟CPU竞争访问Cache，在进行I/O时，会造成CPU的停顿。

I/O还可能会破坏Cache中CPU访问的内容，因为I/O操作可能导致一些新数据被加入Cache，而这些新数据可能在近期内并不会被CPU访问。

解决内容一致性问题的方法

（不管Cache是采用写直达法还是写回法）

软件的方法

设法保证I/O缓冲器中的所有各块都不在Cache中。

具体做法有两种

把I/O缓冲器的页面设置为不可进入Cache的，在进行输入操作时，操作系统总是把输入的数据放到该页面上。

在进行输入操作之前，操作系统先把Cache中与I/O缓冲器相关的数据“赶出”Cache，即把相应的数据块设置为“无效”状态。

硬件的方法

在进行输入操作时，检查相应的I/O地址（I/O缓冲器中的单元）是否在Cache中（即是否有数据副本）。

如果发现I/O地址在Cache中有匹配的项，就把相应的Cache块设置为“无效”。