微机原理与嵌入式系统 1-3 章知识点总结

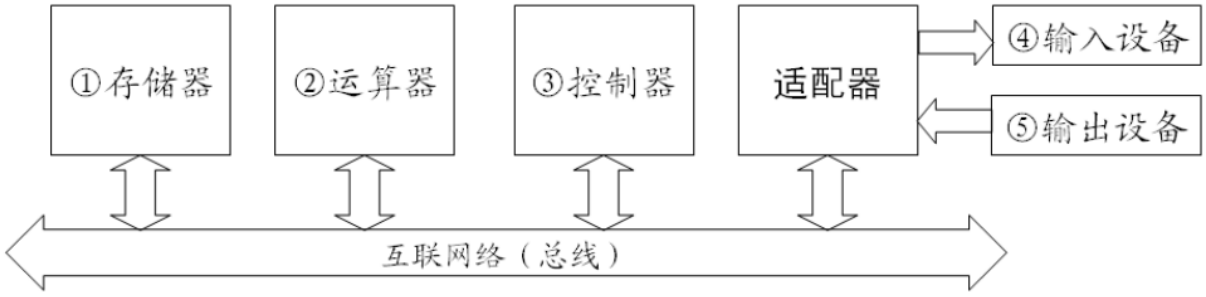
助教：杨冬　　PB18061362

第一章

一、冯.诺伊曼结构的组成

1、存储器　2、运算器 3、控制器 4、输入设备 5、输出设备

示意图：



冯.诺伊曼结构的组成部分

存储器：主存储器（半导体材料，掉电易失性），辅助存储器（主要指磁盘等外存）
运算器：算术逻辑单元 ALU + 寄存器阵列
控制器：指令寄存器IR + 指令译码器ID + 操作控制器OC

二、有符号数

1、数制

进制	英文全称	数值表达	备注
十进制(D)	Decimal	15 :15D	十进制表达时 D 可省略
二进制(B)	Binary	15: 1111B	
十六进制(H)	Hexadecimal	15:0fH	以 A~F 等字母开头的十六进制数，应在字母前加 0
八进制(O)	Octal	15:17O	

2、进制转换

十进制 → 二进制　　短除法取余，从下往上得到二进制

十进制 → 十六进制　1、用 16 除取余法 2、先转二进制，在每四位组合得到十六进制数

十进制 → 八进制　　1、用 8 除取余法 2、先转二进制，在每三位组合得到八进制数

3、有符号数的表示

有符号数需要将最高位来表示正负，正数为 0，负数为 1

假设使用n bit表达数据，则有符号数正数可以表达的范围有是 $0 < x \leq 2^{n-1} - 1$, 负数范围是$-(2^{n-1} - 1) \leq x < 0$，一共可表达$2^n - 1$个数

A. 原码表示

$$[x]_{原} = \begin{cases} x & 0 \leq x \leq 2^{n-1} - 1 \\ 2^{n-1} + |x| & -(2^{n-1} - 1) \leq x \leq 0 \end{cases}$$

B.反码表示

$$[x]_{反} = \begin{cases} x & 0 \leq x \leq 2^{n-1} - 1 \\ (2^n - 1) - |x| & -(2^{n-1} - 1) \leq x \leq 0 \end{cases}$$

C.补码表示

$$[x]_{补} = \begin{cases} x & 0 \leq x \leq 2^{n-1} - 1 \\ 2^n - |x| & -2^{n-1} \leq x \leq 0 \end{cases}$$

如用四位表达有符号数，可以表达的数值范围是-7~7

分别表达-6 和 6 三种表达如下

	-6	6
原码	1110B	0110B
反码	1001B	0110B

补码	1010B	0110B

4、补码判断结果溢出的方法

若记符号位向前进位为CP，次高位向前进位为CF，当且仅当 **CP ⊕CF =1 （异或运算）** 时，结果发生溢出。

如在 4bit 有符号表示下

1110(-2)+0111(7)

$$CP = 1, CF = 1, CP \oplus CF = 0$$

所以数据没有溢出，得到结果 0101(5)

三、BCD 编码

用四位的二进制数表达十进制数,如十进制数 9，用 1001 编码，如 99，则用 10011001 编码(有压缩)

注意

无压缩 BCD 编码：用一个字节表达一个十进制数，如 9，BCD 编码是 00001001，99 是 0000100100001001

有压缩 BCD 编码：用一个字节表达两个十进制数，如 96，10010110

第二章

一、微程序

1、一条指令可以分解为多个微操作

2、微操作由对应的一条微指令实现

3、多条微指令组成微程序实现指令功能

4、微程序存储在 ROM 中，执行时逐条读出完成微操作

二、存储器分体组织

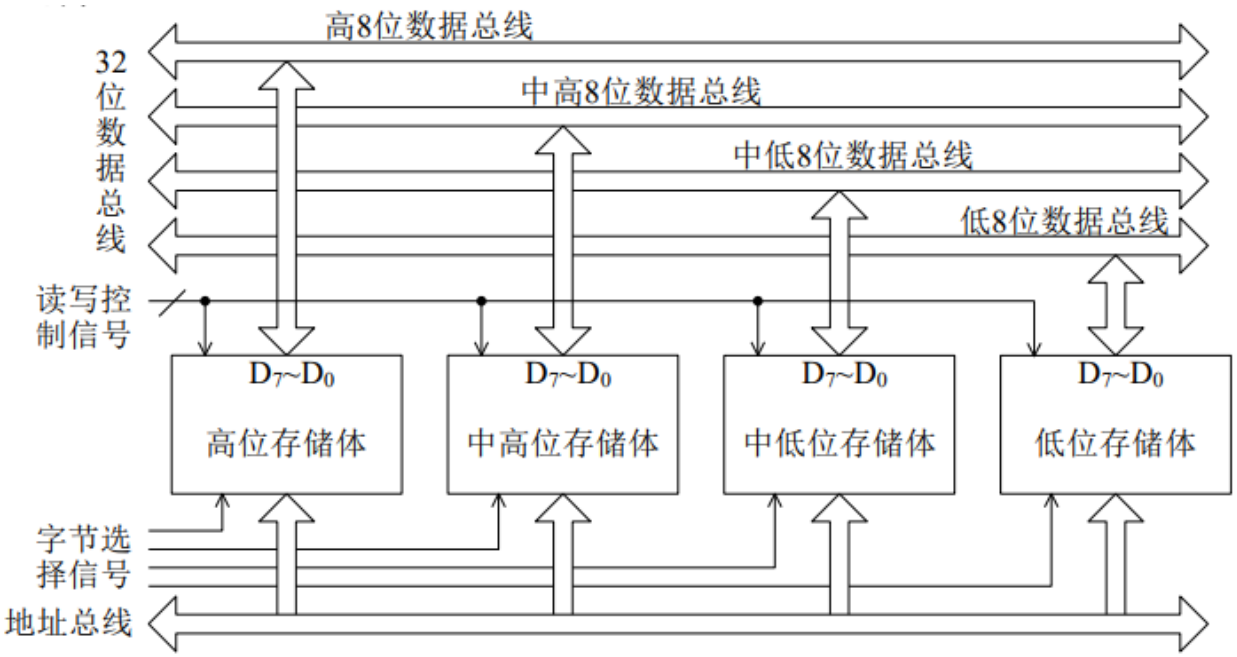
1、对数据总线 16/32/64 位时存储器采用分体编址方法，一次分别读 16/32/64 个bit数

2、编址不在连续，而是需要按照如下方式编址

A、16 位机的字起始地址应该是 2 的倍数，如 0、2、4...

B、32 位机的字起始地址应该是 4 的倍数，如 0、4、8 …

C、64 位机的字起始地址应该是 8 的倍数，如 0、8、16…



32 位数据总线存储器分体设计方式

三、小端存储和大端存储

大端存储：高位低地址，低位高地址

小端存储：高位高地址，低位低地址

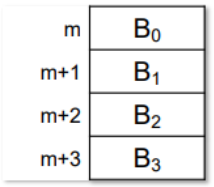
假设W由B_3、B_2、B_1和B_0 组成，B_3是最高字节，B_0是最低字节，存储W需使用4个地址连续的内存单元

对于地址依次为m、m+1、m+2和m+3的连续4个存储单元，m单元地址最小，称为尾部（Endian）；m+3单元地址最大，称为头部

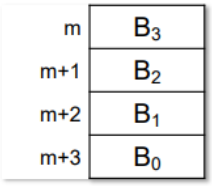
W有两种存放格式：

- Intel x86采用小尾或小端格式
- Motorola采用大尾或大端格式

如果采用对准存放，m是整个字的地址



小端格式
高位高地址
低位低地址



大端格式
高位低地址
低位高地址

大端和小端存储

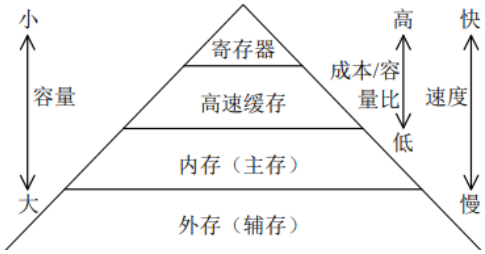
四、存储器分级

对存储器的要求：速度快、容量大、成本低

要同时满足这三个要求“太难了”

破局方法：分级存储体系结构

- 使用外存满足大容量、低成本和非易失的要求
- 使用DRAM型内存，兼顾容量、速度和成本
- 使用高速缓存，减少CPU访问内存的开销



高速缓存（Cache）

位于CPU与内存之间，SRAM型小容量快速存储器，用于存放CPU最近使用过或者可能要使用的指令和数据

存储器分级

五、模型机指令执行过程

主要对 RISC 的一条指令完成从取指，译码，执行三个主要过程的动作描述，对于第一条指令进行描述时基于如下假设：

1、模型机是 32 位，数据总线 32 位，地址总线 32 位

2、RISC 结构，定长指令设计，每条指令长度也是 32 位

3、待运行程序的首地址为 0x2000 0000

在教材上第一条指令二进制代码是为“E3 A0 06 FF”，汇编语句对应 MOV R0 #0x00FF000

则描述过程如下

① PC内容0x20000000送至地址缓冲器/驱动器，地址总线的输出经地址译码器译码，寻址内存单元

② PC值自动加4（假设PC内容以字节为单位），指向下一条指令的存放地址（何时修改PC有不同的策略）

③ OC发读信号，将“E3 A0 06 FF”读出到数据总线；

④ 由于是取指操作，数据总线上的数据被装入IR

⑤ ID对操作码译码，OC产生相应的控制信号

⑥ 第一条指令源操作数是立即数（取指时能从指令编码中立即得到），被装入R0寄存器，指令执行完毕

```
START:
  MOV    R0, #0x000FF000
  LDR    R1, [R3]
  ADD    R1, R0, R1
  JO     L2
  STR    R0, [R3+#0x80]
L2:
  HLT
```

对第二条汇编代码“LDR R1,[R3]“进行描述如下：

1、PC 内容 0x20000004 送至地址缓冲器/驱动器，地址总线的输出经地址译码器译码，寻址内存单元

2、PC 值自动加 4（假设 PC 内容以字节为单位），指向下一条指令的存放地址

3、OC 发读信号，将“LDR R1,[R3]”读出到数据总线

4、由于是取指操作，数据总线上的数据被装入 IR

5、ID 对操作码译码，OC 产生相应的控制信号

6、此条指令的源操作数是寄存器 R3 中的值，目标操作数是寄存器 R1

7、在操作控制器输出的控制信号作用下， R3 寄存器的内容经地址形成部件和地址驱

动器送到地址总线，再经地址译码后寻址到源操作数存放的内存单元

8、操作控制器发出读信号，将源操作数读出到数据总线，然后加载到 R1 寄存

器

总结：

在对指令进行动作分解描述时，取指，译码的步骤基本相同，也就是 1~6 步骤作为模板可以基本套用，在第七步根据该指令的具体的命令完成相应的功能

如 JO L2

可以描述为在操作控制器输出的控制信号作用下，PC 值改变为“L2“代表的地址值，并开始下一次取指过程。

六、RISC 和 CISC 的各自特性与区别

CISC

特点:

1、指令长度不一

2、非 Load/Store 体系

3、MOVE 操作 ：Move destination, source 的传送指令，可实现寄存器与寄存器之 间，以及寄存器与存储器之间的数据（复制）传送

4、两操作数 ：OPR(操作码) DST(目的操作数) , SRC(源操作数)

5、指令功能强大、寻址方式多样、程序简洁

RISC

RISC处理器的指令简单明了，易于理解，纯粹RISC风格的指令具有以下一些主要特点：

- 寻址方式简单，种类较少；
- 指令集中的指令数量较少；
- Load/Store体系结构；
- 每条指令长度一致，执行时间相同；
- 面向寄存器的编程思想；
- 算术和逻辑运算指令普遍支持三操作数；
- 只能对寄存器操作数进行算术和逻辑运算；
- 程序代码量较大，因为执行复杂操作需要使用较多的简单指令

七、流水线技术

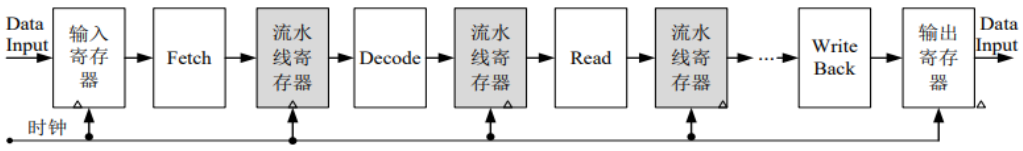
1、基本原理

将功能部件按指令操作步骤顺序进行排列部署，前后部件之间增加缓冲寄存器，构成指令处理流水线

前后两个部件经过缓冲寄存器隔离后，可以相对独立地并行工作

部件之间的工作交接（数据传递）将通过缓存寄存器进行

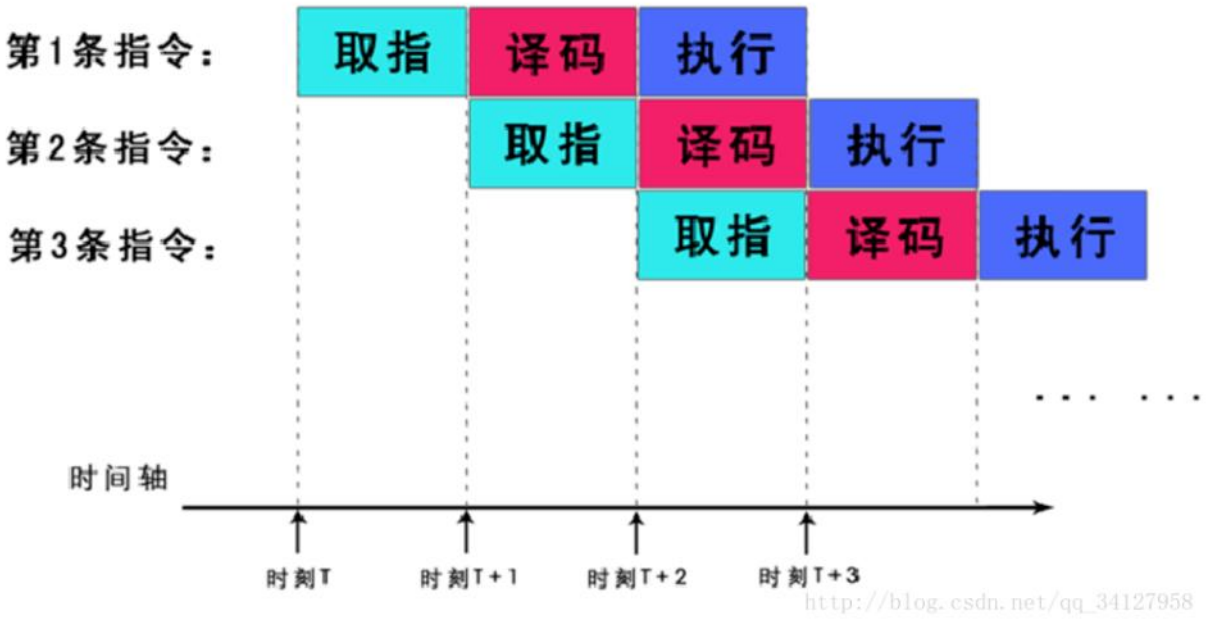
这种缓存寄存器被称为流水线寄存器。



一个指令需要 n 个步骤，称为 n 级流水线，插入 n-1 个流水线寄存器

2、3 级流水线

在 取指，译码，执行中插入两个流水线寄存器



三级流水线

3、5 级流水线

在 Fetch（取指）、Decode（译码）、Read（取操作数）、Execute（执行）和 Writeback（回写）5 个步骤插入四个流水线寄存器

周期	1	2	3	4	5	6	7	8	9	10
指令1	Fetch	Decode	Read	Execute	Write					
指令2						Fetch	Decode	Read	Execute	Write

（a）非流水线多级串行执行

周期	1	2	3	4	5	6	7	8	9	10
指令1	Fetch	Decode	Read	Execute	Write					
指令2		Fetch	Decode	Read	Execute	Write				
指令3			Fetch	Decode	Read	Execute	Write			
指令4				Fetch	Decode	Read	Execute	Write		
指令5					Fetch	Decode	Read	Execute	Write	
指令6						Fetch	Decode	Read	Execute	Write
指令7							Fetch	Decode	Read	Execute
指令8								Fetch	Decode	Read
指令9									Fetch	Decode
指令10										Fetch

5 级流水线

4、流水线三种相关冲突

1）资源相关，也称为结构相关

- 多条指令在同一个周期内争用同一个公用部件。例如：冯 诺依曼结构计算机的Fetch、Load和Store操作都使用公用总线接口访问同一个存储器，前一条指令的数据存取操作可能会影响后续指令的取指操作
- 解决方法：
  - ①后面一条指令等待一个节拍再启动。称作向流水线插入气泡（Bubble）或插入阻塞，这将造成流水线性能下降
  - ②采用哈佛结构。解除存取操作数与取指之间的资源相关

哈佛结构：指令和数据分别存储

2）数据相关

- 后一条指令执行需要使用前一条指令的结果。例如：流水线上前后执行的两条算逻指令：

```
SUB   R1, R2, R3    ; R2减R3结果写入R1
AND   R4, R1, R5    ; R1和R5逻辑与运算，结果写入R4
```

周期	1	2	3	4	5	6	7
SUB	Fetch	Decode	Read	Execute	Write		
ADD		Fetch	Decode	Read	Execute	Write	

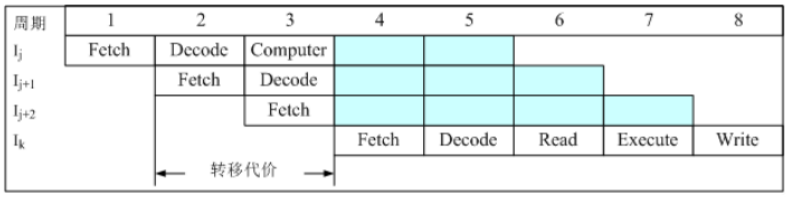
- SUB指令在第5个周期才将结果写回R1，但是AND指令在第4个周期就要读R1进行运算，而程序的本意是“写后读”（Read After Write，RAW）
- RAW是一种最为常见的数据相关

流水线还可能存在“写后写”（WAW）和“读后写”（WAR）两类数据相关

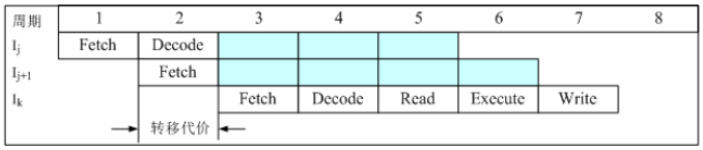
- WAW：I_{j+1}试图在指令I_j写数据之前写数据，这样最终结果将由I_j决定，而程序本意是保留I_{j+1}的结果
- WAR：I_{j+1}试图在指令I_j读一个数据之前写该数据，此时指令I_j读到的是被I_{j+1}“篡改”后的数据
- 插入气泡可消除数据相关，但将造成流水线性能下降
- 解决之道：
  - 定向推送，前一条指令执行结果通过专用通道直接推送给下一条，减少一个流水线周期，可减少数据相关
  - 优化编译器，对前后指令进行检查，调整执行顺序

3）控制相关

- 遇到转移指令时，后续已进入流水线的指令都应清空
- 以无条件转移（包括子程序调用）指令为例：
- 假设指令I_j是无条件转移指令，其执行步骤为：取指、译码、计算转移地址（Computer）并更新程序计算器PC。在第4个周期读取转移目标指令I_k。在此之前流水线上的指令I_{j+1}和I_{j+2}应清除，造成流水线断流。产生两个流水线周期延迟被称为转移代价（branch penalty）



- **减少转移代价的方法**
  - 对于无条件转移指令，增加电路，在译码阶段提前计算转移目标地址，在第3个周期读取转移目标指令I_k，将转移代价减少到一个流水线周期



  - 类似方法同样适用于少数条件转移指令
- **但是，大多数条件转移指令是否转移取决于状态标志位，而标志位在ALU运算后才更新，转移代价较大。流水线级数却多，代价越大。该如何解决呢？**

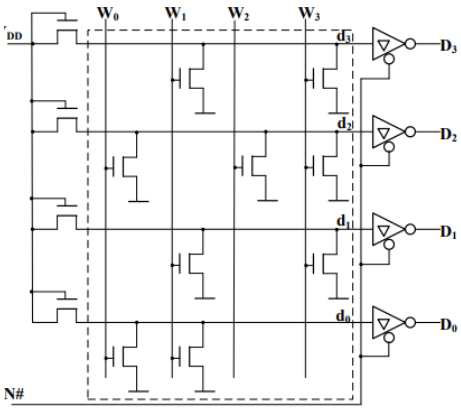
- **转移延迟槽（branch delay slot）：转移指令I_j后面的一个时间片。无论是否转移，位于转移延迟槽的指令总是会被执行。一个启示：**
  - 如果能对转移与否做出正确预测，则可根据预测结果选择合适的指令“装入”转移延迟槽。例如，假设预测结果是转移，位于转移延迟槽的恰好是转移目的指令I_k，转移没有任何代价
- **问题：如何对转移指令的行为进行预测？**
- **动态转移预测（dynamic branch prediction）**
  - 根据转移指令过去的行为进行预测
  - 对发生转移的可能性进行加权量化。例如，用2位二进制码表示发生转移可能性的权值，“11”是极有可能发生转移，“10”是有可能发生转移，“01”是有可能不转移，“00”是极有可能不转移。

- 动态“打分”：每发生一次转移，权值+1，加到11b为止；每发生一次不转移，-1，减到00b为止。
- 使用BTB（branch target buffer，转移目标缓冲器），收集和存储了近期所有转移指令的有关信息，并按照查找表的形式进行组织
- BTB不能太大，一般为1024个表项，其内容包括：
  - 转移指令I_j的地址（查找表索引）
  - I_j转移可能性的量化结果（2bit权值）
  - 转移目标指令I_k的地址
- 每条指令在取指时，处理器根据其地址在BTB中进行快速搜索，如果有记录则表明转移指令，再根据其“档案”记录进行相应处理，最后再根据实际行为修正权值

第三章 存储器系统

一、ROM 中的地址译码，字线、位线

- **Mask ROM（掩膜ROM）的存储数据由专门设计的掩模板决定，为固化数据，用户不能修改。**
- **ROM电路结构包括存储矩阵、地址译码器和输出缓冲器三个组成部分。**

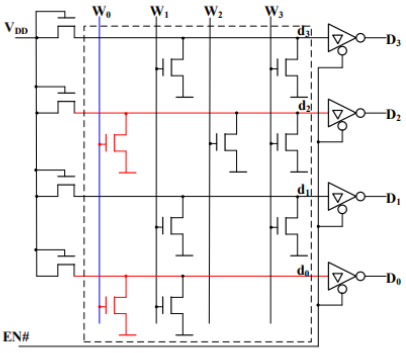


- **存储矩阵**由许多存储单元排列而成。存储单元可以用二极管构成，也可以用双极型三极管或MOS管构成。每个单元能存放一位二值代码（0或1）。每一个或一组存储单元有一个对应的地址代码。
- **地址译码器**的作用是将输入的地址代码译成相应的控制信号，利用这个控制信号从存储矩阵中将指定的单元选出，并把其中的数据送到输出缓冲器。
- **三态输出缓冲器**的作用有两个，一是能提高存储器的带负载能力，二是实现对输出状态的三态控制，以使与系统的总线连接与隔离。

根据字线和位线进行读数

行列交叉处有无MOS管分别表示了“0”和“1”

- 如4位数据输出，4×4位的MOS管，单译码结构。
- 地址线A1、A0，译码后输出4条选择线W_3~W_0，用于选中4个单元的某一个（每个单元4位输出）。
- 存储矩阵由MOS门组成，W_3~W_0任何一根线上给出高电平信号时，d_3~d_0会输出一个4位二值代码。
- 将每个输出代码称为一个“字”，并将W_3~W_0称为字线，将d_3~d_0称为位线(或数据线）。
- 输出端的缓冲器用来提高带负载能力，并将输出的高、低电平变换为标准的逻辑电平。
- 同时，通过给定EN#信号实现对输出的三态控制，将数据反相输出。



若地址线A_1A_0=00，则选中0号单元，即字线0为高电平，若有MOS管与其相连(如位线d_2和d_0），其相应的MOS管导通，位线输出为0，而位线1和3没有MOS管与字线相连，则输出为1。

二、存储器与 CPU 的连接

1、地址空间

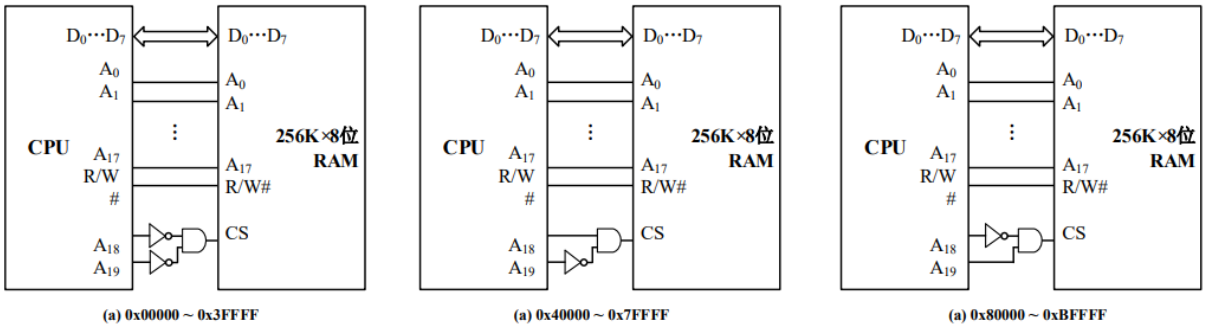
地址空间

- **计算机中地址总线AB的宽度决定了存储器空间的最大寻址范围，常把这个寻址范围称为地址空间。**
  - 例，16位地址总线可寻址空间为2^{16}=64K，可寻址64K个存储单元，存储单元地址范围：0x0000到0xFFFF。
- **按字编址、按字节编址**
  - 若CPU的地址线数目也是32，其所能寻址的空间大小为0~4G，若按照字节为单位进行编址（按字节编址的计算机），则可寻址4GB；若按照字为单位进行编址（按字编址的计算机），由于字长是4字节，故可寻址16GB。
  - 通常资料中分析的是按字节编址的计算机。

2、存储器与 CPU 的连接

存储器芯片和CPU的连接

- 连接存储器芯片和CPU时，要根据地址空间的划分设计存储器芯片CPU地址线的连接方式。
  - 例，用256K×8位的存储器芯片连接具有20根地址线的CPU



- 设计微机系统时，存储器应与地址、数据、控制总线正确连接，并应考虑如下问题：

① **CPU总线的负载能力**

CPU通过总线直接驱动负载的能力有限，应根据需要连接的存储器芯片参数，考虑在总线上增加缓冲器或驱动器，增大CPU的负载能力。

② **CPU时序与存储器存取速度间的配合**

CPU要对存储器频繁读/写，选芯片时要考虑其存取速度能否与CPU读/写时序匹配。

③ **存储器的地址分配和片选**

需要为存储器分配地址范围。由于每块芯片存储容量有限，一个存储器系统可能是由多块芯片组成，要重点考虑容量的扩充方案和片选信号的形成。

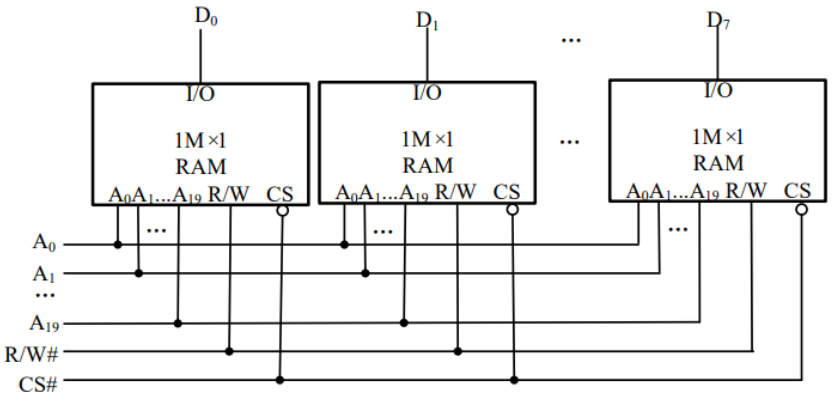
④ **控制信号的连接**

CPU提供的存储器控制信号，如CS#、OE#、WE#等，应与存储器的相关引脚正确连接，才能实现读/写等控制功能。

3、位扩展

存储器系统扩展方式：位扩展

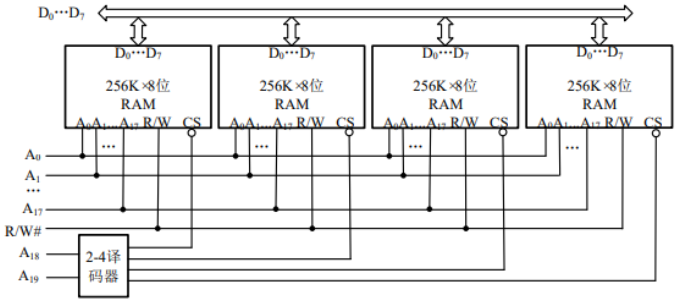
- 在存储器芯片字数不变的前提下，进行数据的位数扩展。
- 举例：1M×1位的芯片扩展为1M×8位的RAM并与CPU总线连接。



- 扩展设计方法
  - 每个芯片**数据线分别连接**数据总线D_7~D_0的不同位，以形成8位数据；
  - 各芯片的地址线A_{19}~A_0与CPU地址总线对应**地址线并联**；
  - 读写控制线R/W#与CPU读写**控制线分别并联**连接，CPU的片选控制线CS#与各芯片的CS#并联。

4、字扩展

- 在存储器芯片的位数满足的前提下，进行字数扩展。
  - 举例：存储器芯片为256K×8位，采用4片进行字扩展为1M×8位的RAM，并与CPU总线连接。



- 扩展设计方法
  - 每个芯片的各位**数据线**分别与数据总线D_7~D_0位**并联**；
  - 各芯片**低位地址线**A17~A0与CPU地址总线对应地址线**并联**；
  - **高位地址线**A19、A18**通过2线-4线译码器**分别产生不同的译码输出信号**控制**每个存储器芯片的**片选端CS#**；
  - 各芯片的读写**控制线**R/W#与CPU读写控制线分别**并联**连接。

三、Cache 基本工作原理及作用

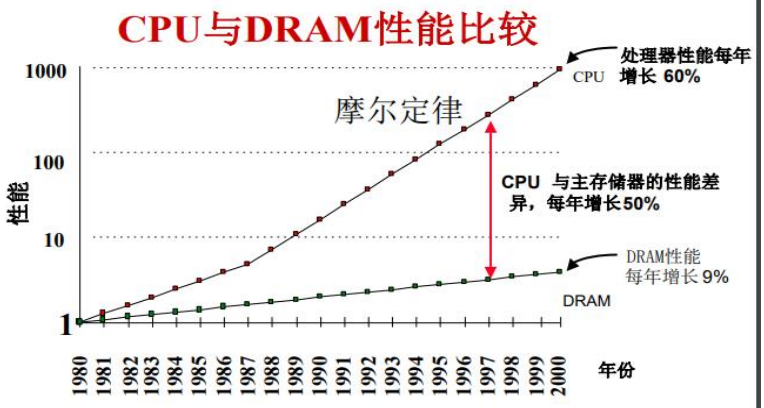
Cache原理
为什么采用高速缓冲存储器Cache？

- 在CPU的所有操作中，存储器的存取访问是最频繁的操作。
- **CPU速度比DRAM存储器的存取速度高得多。**
  - 高端CPU时钟频率已超5GHz，指令执行时间远小于1ns。
  - 内存访问速度虽达ns级，如SDRAM为6~10ns，但与CPU有明显差距。
- 存储器的访问速度低是制约计算机系统性能的关键因素。

Cache原理
CPU与DRAM的发展现状

CPU和主存储器的速度总是有差距，CPU的发展一直是提高速度为核心目标；主存的发展则一直以提高容量为核心目标。

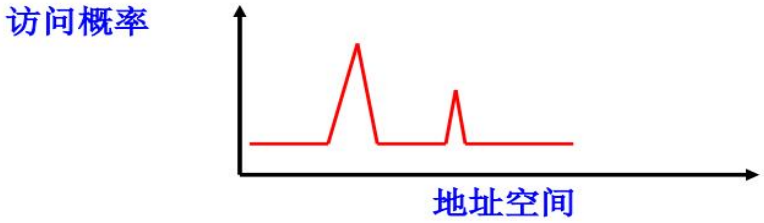
矛盾该如何解决？



- 这不是一个技术问题，而是一个经济问题。在技术方面，能够制造出多高速度的CPU，就能够制造出同样速度的存储器，只不过我们是否愿意付出相应的代价。**应如何在性能和代价之间进行权衡和折衷？**

Cache原理
程序访问的局部性原理

- 程序在一定时间段内通常只访问较小的地址空间



- 两种局部性：时间局部性和空间局部性
  - 时间局部性：最近访问的信息很可能再次被访问
  - 空间局部性：最近访问信息的邻近信息可能被访问

Cache原理
在主存和CPU之间设置Cache

- 思想：根据程序访问的时空局部性，把经常访问的代码和数据保存到高速缓冲存储器（Cache)中，把不常访问的代码和数据保存到大容量的相对低速DRAM中，尽量减少CPU访问DRAM的概率，在保证系统性能的前提下，降低存储器系统的实现代价。
- 实现：Cache设置在CPU与主存储器之间，通常采用存取速度快并且无需刷新的SRAM来实现。
  - 根据时间局部性：将最近被访问的信息项装入到Cache中；
  - 根据空间局部性：将最近被访问的信息项的邻近信息也装入到Cache中。
- 在主存和CPU之间设置了Cache之后，如果当前正在执行的程序和数据存放在Cache中，则当程序运行时不必再从主存储器读取指令和数据，而只需访问Cache即可。