《计组I》
第五章　处理器设计

计算机科学与技术学院

温故 —关于指令系统

- 简述MOV, LEA, MOV OFFSET的关系?
  - MOV AX, [BX + SI + 5]; 传输数据
  - LEA AX, [BX + SI + 5]; 取偏移地址
  - MOV AX, OFFSET [BX + SI + 5]; 等效LEA
- 汇编中实现循环的典型方案?
  - 计数器循环——CX+LOOP
  - 条件转移循环——JMP等
- 汇编中如何定义“变量”?
  - DB —— 定义字节
  - DW ——定义字
  - DD ——双字
- 简述procedure，如何调用procedure?
- 可将具有一定功能的程序段定义为一个过程，过程由伪指令PROC和ENDP来定义
- 调用方法：CALL、 JMP
- 简述marco，如何调用marco?
- 可将短的程序段定义为一个宏，支持参数
- 调用方法：宏名 参数 inline函数——

```
addmin macro a b
mov ax, a;
add ax, b;
addmin endm
addmin 1h 1h
```

回顾一下指令系统一章学了啥?

- **我们知道了CPU的逻辑结构:**
  - **CPU里面有通用寄存器，A/B/C/DX**
  - **有ALU**
  - **有段/偏移寄存器 CS:IP / DS:SI/DI/BX / SS:SP / ES**
  - **有标志位Flags (SF, CF, OF, IF, PF, ZF …)**
  - **CPU里面用总线方式连接这些部件…**

回顾一下指令系统一章学了啥?

- 我们还知道了:
  - 汇编其实是非常“简单”的语言…
  - 汇编/机器程序就是“流水帐”，数据与指令无区分
  - 运行时的程序在内存里，CPU从内存读指令执行
  - CPU运行哪里的代码?
    - CS:IP哪到哪就运行哪!
  - 学习+使用了一些常见的指令(mov, add, int, halt, ret…)
  - …

回顾一下指令系统一章学了啥?

- **我们还知道了:**
  - **多种寻址方式(为什么叫寻址?)**
    - **指令的寻址方式 (哪几种?)**
    - **数据的寻址方式 (哪几种?)**
    - **…**

回顾一下指令系统一章学了啥?

- 似乎，我们理解了…
  - CPU是基于寄存器器
  - CPU是基于地址运行的
  - CPU可以对一些指令正确响应&工作
  - 基于汇编语言的CPU操作方法
- 但是，
  - CPU到底是什么样?
  - 指令系统怎么设计的?
  - CPU是如何分解并执行指令的?
  - 以及最神秘的~“CPU内部的CPU”
    - CPU内部的微指令/操作如何实现

回顾一下指令系统一章学了啥?

- 但是,
  - CPU到底是什么样?
  - 指令系统怎么设计的?
  - CPU是如何分解并执行指令的?
  - 以及最神秘的~“CPU内部的CPU”
    - CPU内部的微指令/操作如何实现

- 第五章 CPU设计
  - 5.1 CPU结构
  - 5.2 CPU指令设计
  - 5.3 指令执行过程
  - 5.4 CPU部件设计
    - 5.4.2 控制器的设计
      - 组合逻辑控制器
      - 微指令控制器
        - 微指令编码设计

CPU的功能

指令控制 —— 程序的顺序控制

操作控制 —— 指令的操作信号

时间控制 —— 操作信号的时序

数据加工 —— 数据处理，CPU的根本任务

中断处理 —— 相应中断（异常）

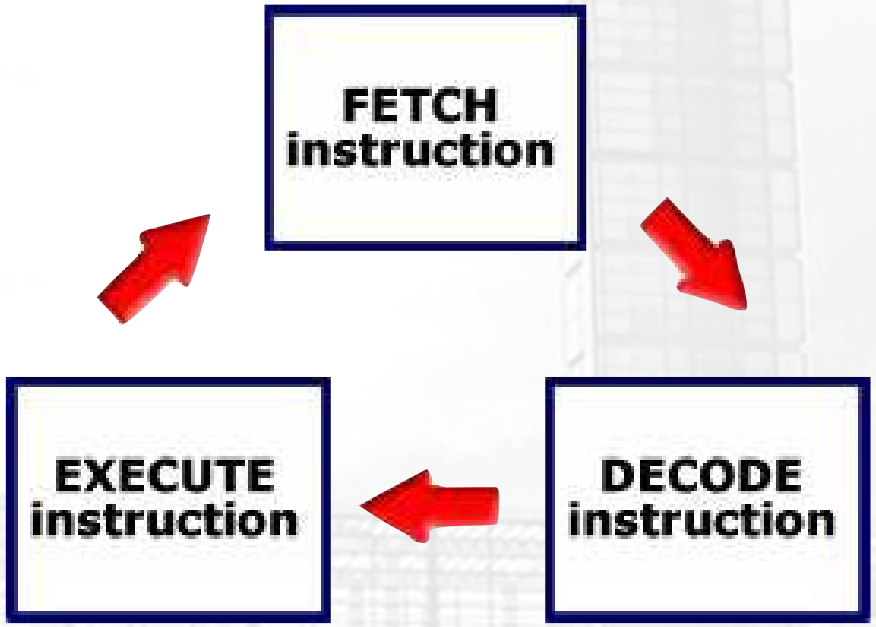
其它处理 —— 相应DMA、RESET等

本章内容

CPU（Central Processing Unit）主要负责获取程序的指令、译码所获取的指令、针对指令指定的数据完成指定顺序的操作。

本章主要介绍CPU的结构及控制器的设计方法。

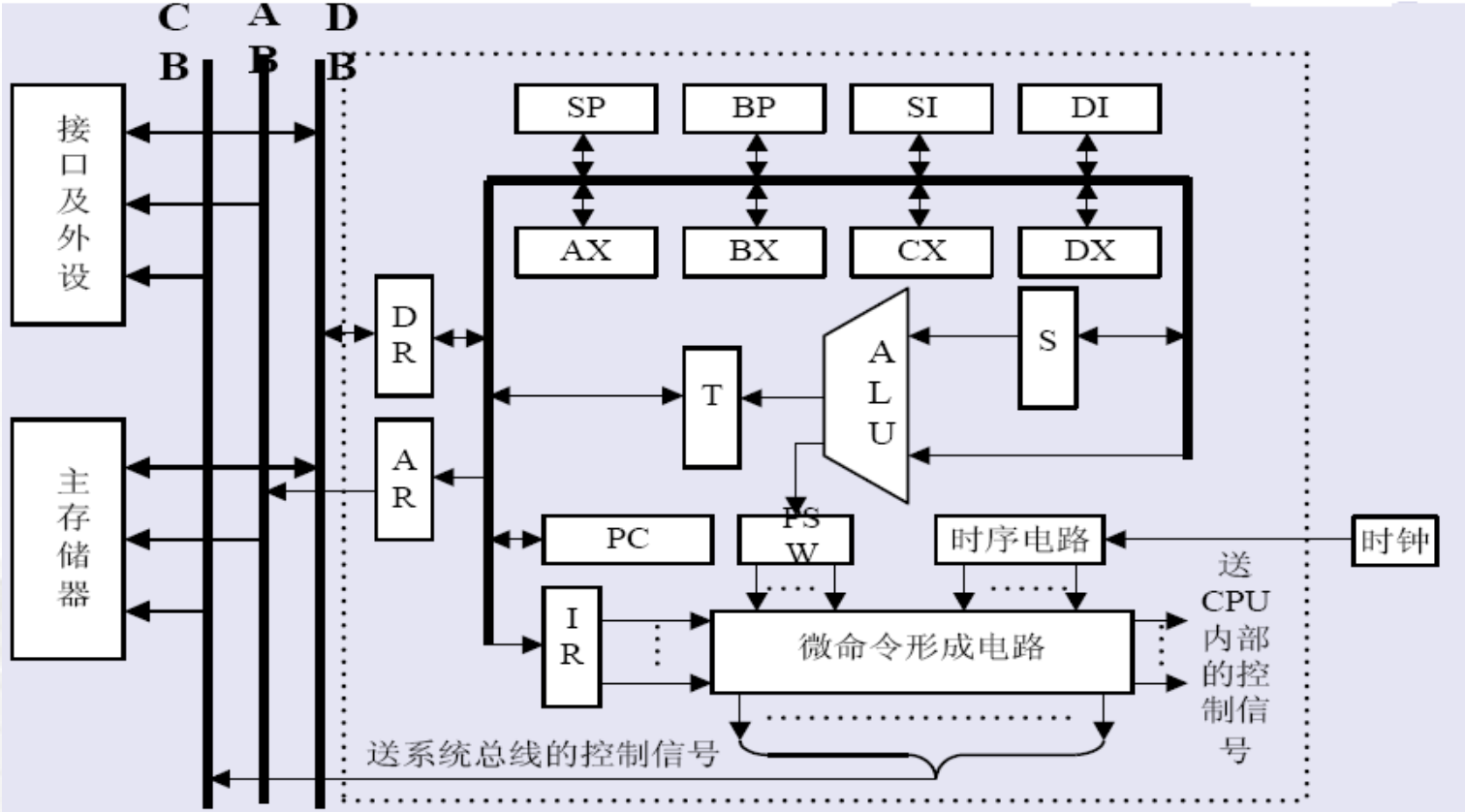
- CPU 的结构
- 指令系统设计
- CPU执行指令的过程
- CPU的部件设计



CPU基本组成

CPU基本组成

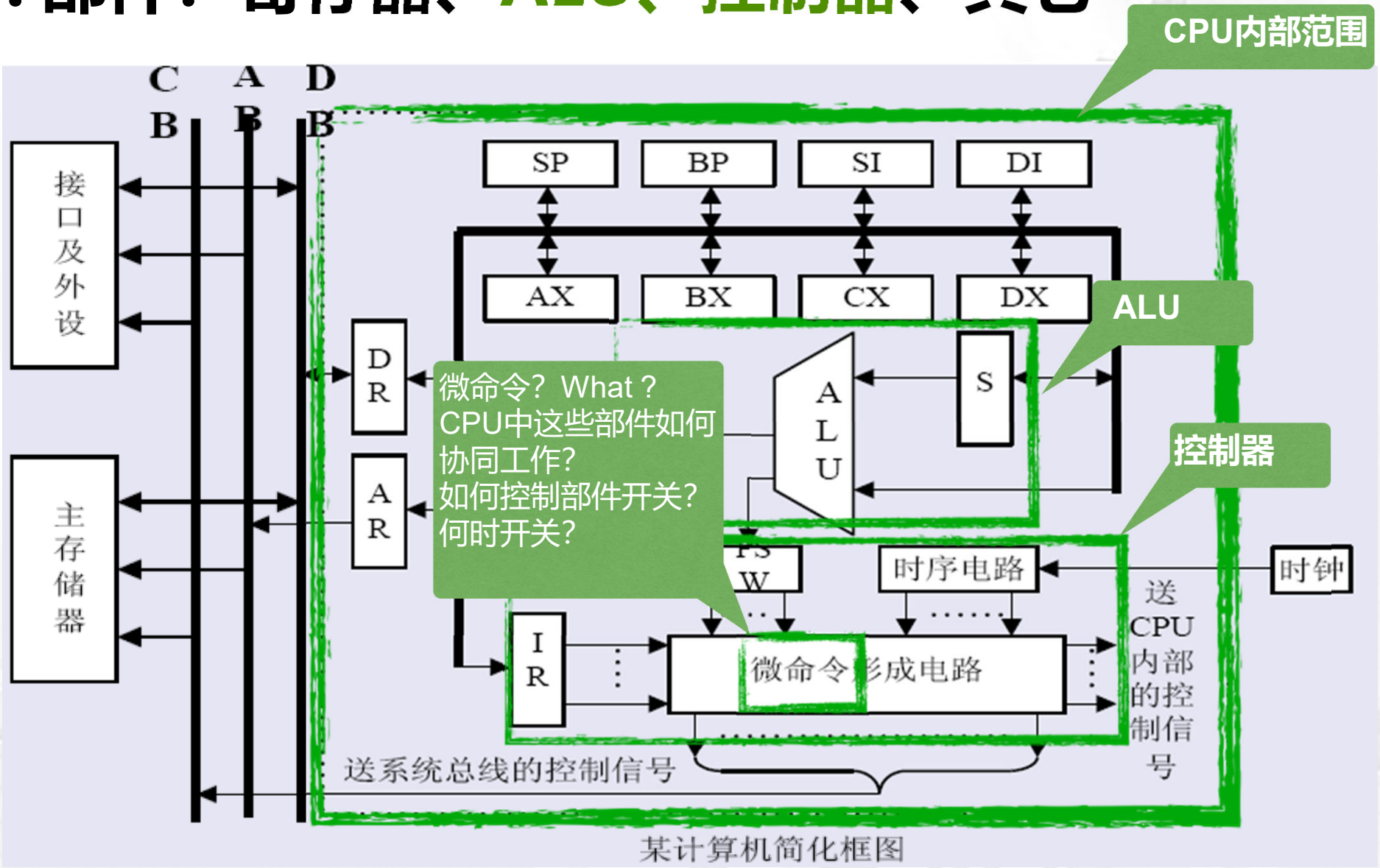
1. 部件：寄存器、ALU、控制器、其它
2. 数据通路(DataPath)：总线(BUS)、专用通路
3. 外部交互：外部总线



某计算机简化框图

CPU基本组成

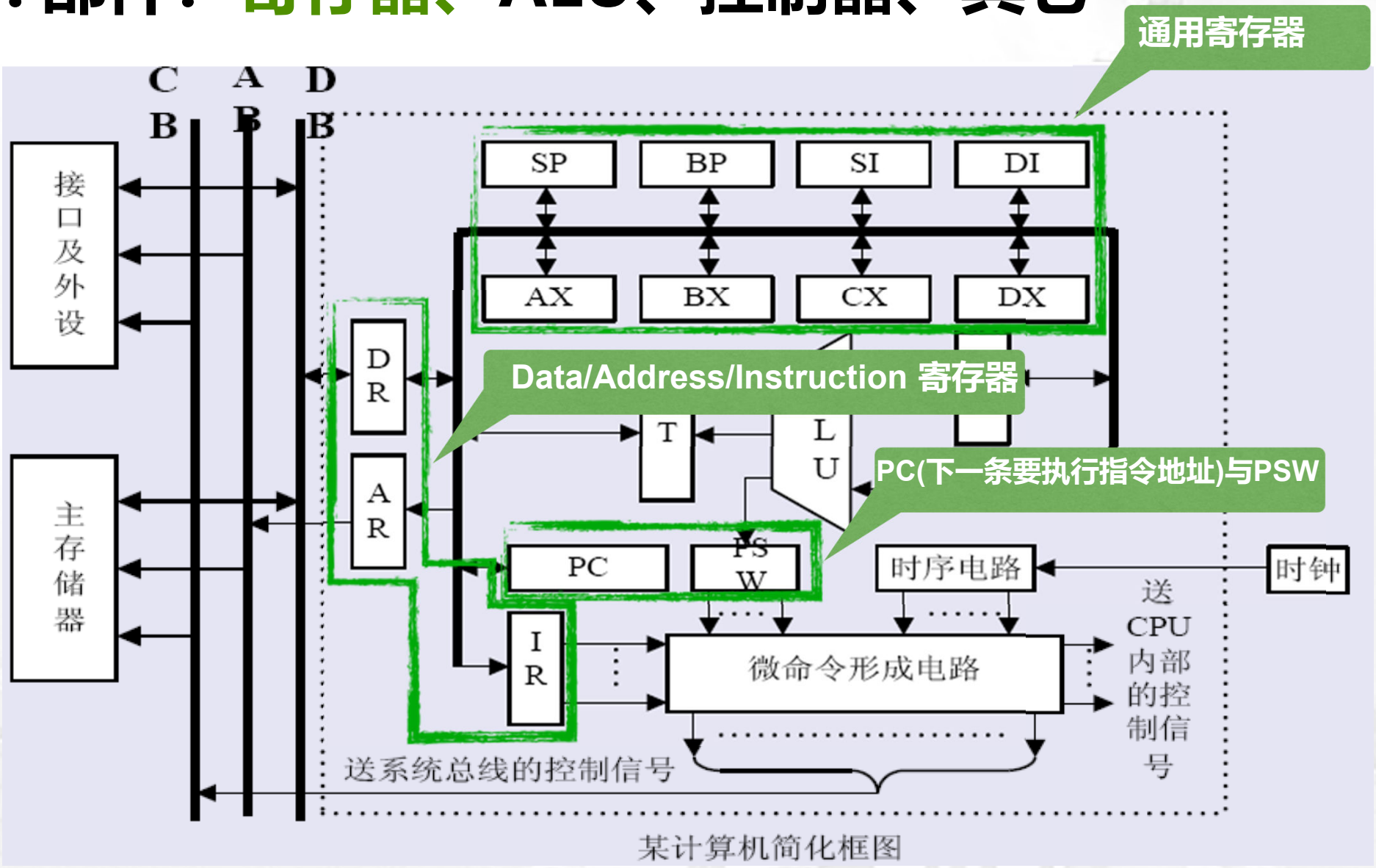
1. 部件：寄存器、ALU、控制器、其它



某计算机简化框图

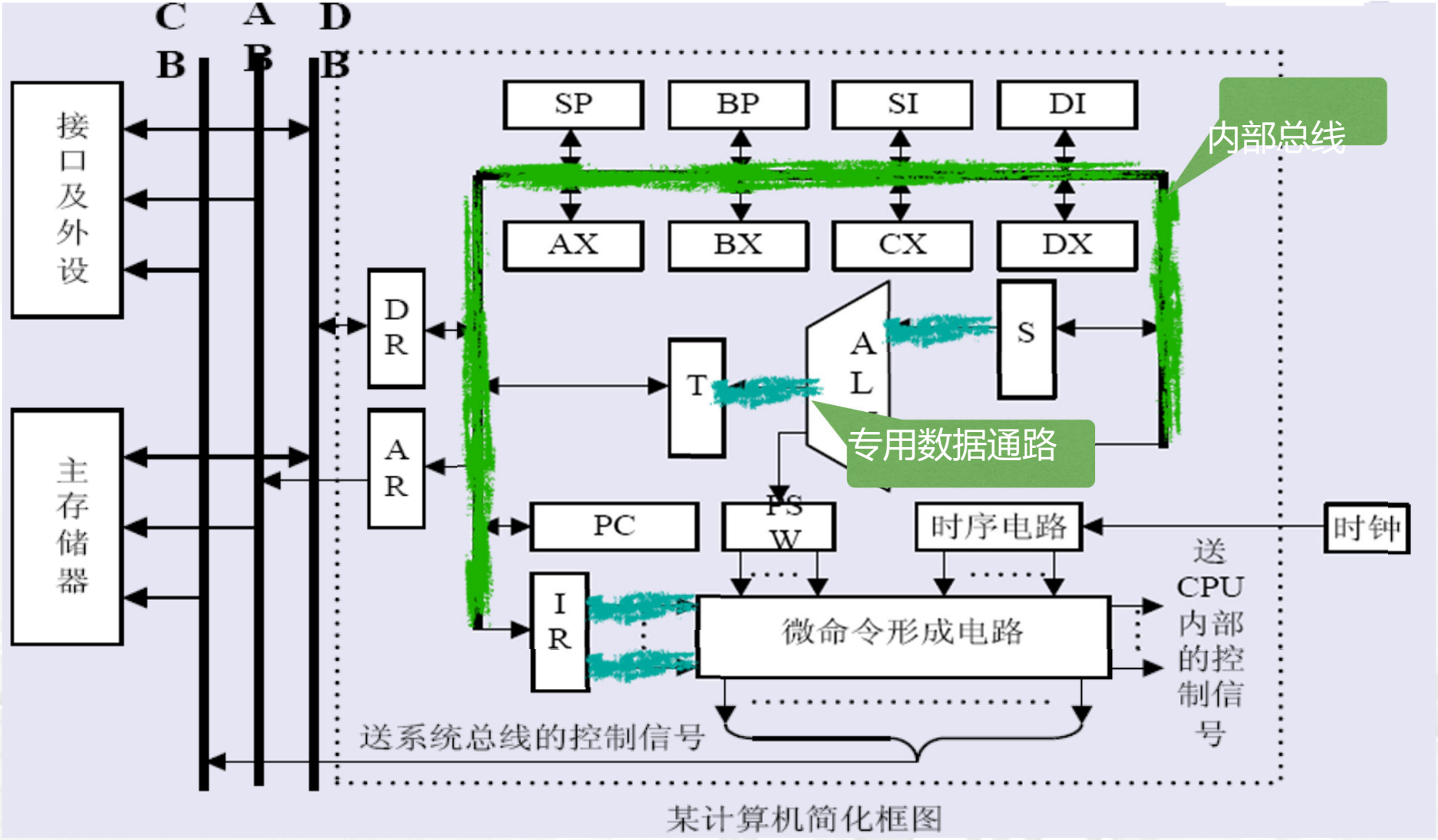
CPU基本组成

1. 部件：寄存器、ALU、控制器、其它



CPU基本组成

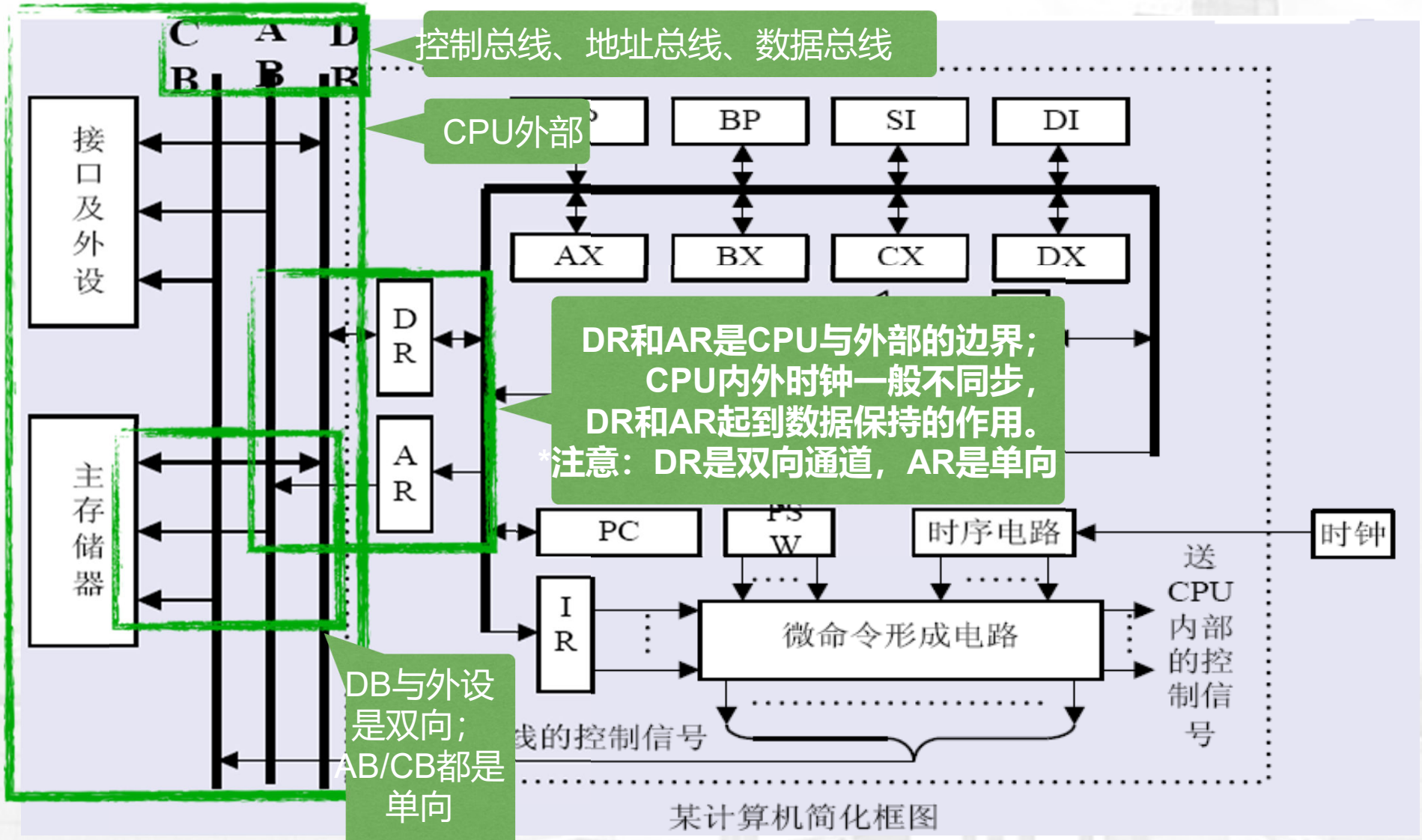
2. 数据通路(DataPath)：总线(BUS)、专用通



某计算机简化框图

CPU基本组成

3. 外部交互：外部总线



CPU基本组成

- 数据总线和CPU内部总线是双向的。
- 总线的条数称为数据总线宽度。比如，16位总线，指其数据总线为16根。
- 数据总线是三态的，是指0，1和高阻抗三个状态。由于总线是公共通道，在某一时刻，只允许接收某一设备的信号，其他一切设备都应和它断开。

CPU基本组成

- 控制总线上传送一个部件对另一个部件的控制信号。
- 主设备与从设备:在总线上所连接的各类设备，按其对总线有无控制功能可分为主设备和从设备。 主设备对总线有控制权，从设备只能响应。
- 三态？不一定~

CPU基本组成

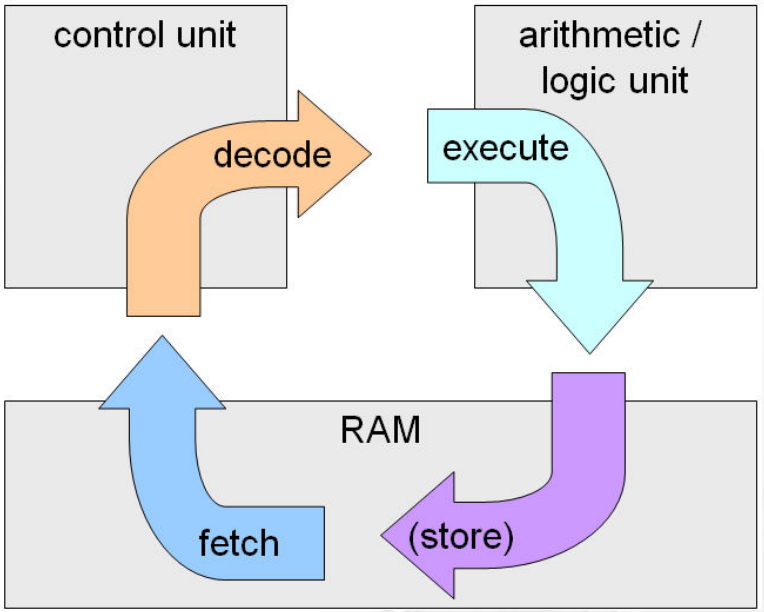
- 地址总线上传送地址信号，用来指定需要访问的地址/部件
- 总线上的所有部件接收到该地址信号，但只有经过译码电路选中的部件才接收控制信号。
- 地址总线是单向的
- 地址总线也是三态的

深入CPU指令执行过程

深入CPU指令执行过程

1、指令执行的基本过程

取指、执行
取指、取操作数、执行
取指、取操作数、执行、写结果



以指令add [2000h], ax为例，看看4个阶段的CPU内部执行流程

深入CPU指令执行过程

- 取指令Fetch
  - PC->AR
  - AR->AB
  - AB->MAR
  - Read (MM->MDR)
  - MDR->DB
  - DB->DR
  - DR->IR
  - PC++

CS:IP指向下一条指令的地址

内存里的AR

内存里的DR

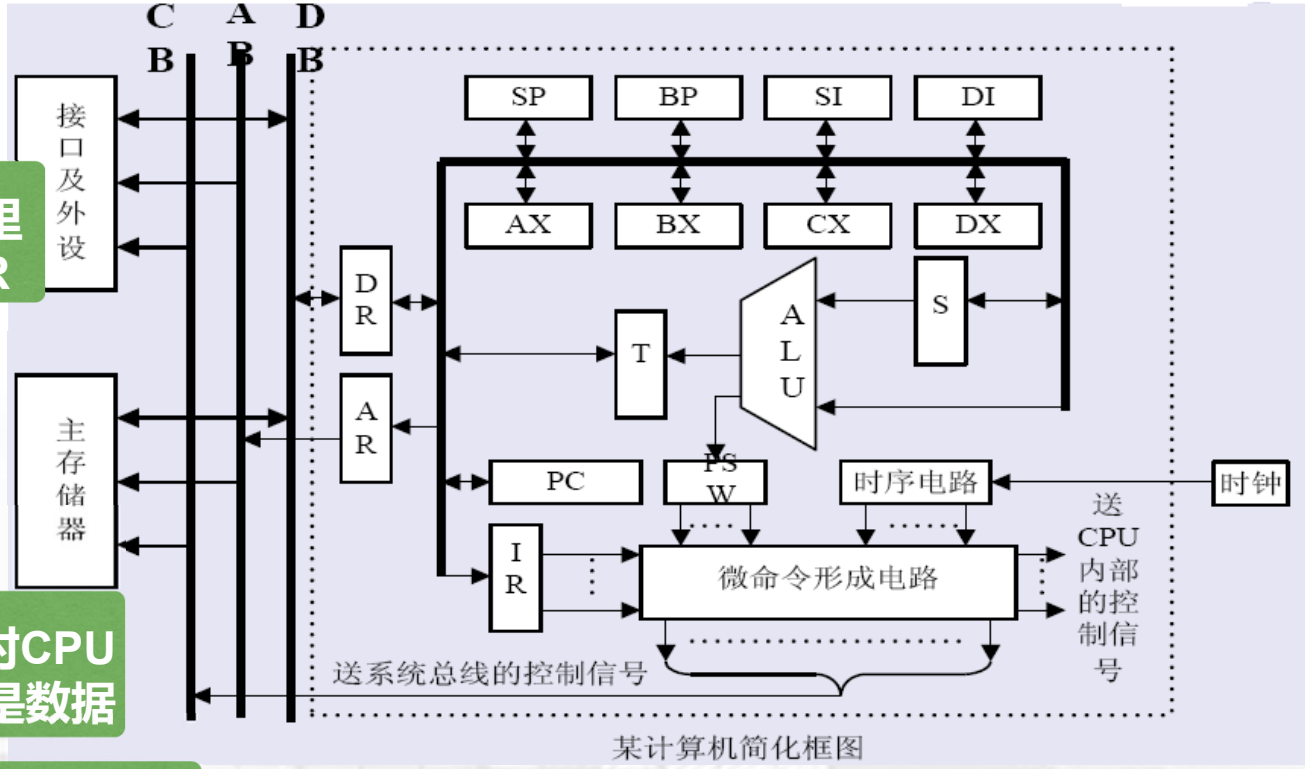
内存

取出的指令对CPU来说，就是数据

把“指令数据”导出IR才真的是看作指令

IP指针自加1

以指令add ax, 2000h为例



某计算机简化框图

深入CPU指令执行过程

以指令add ax, 2000h为例

- 取操作数Fetch Operands
  - PC->AR
  - AR->AB
  - AB->MAR
  - Read (MM->MDR)
  - MDR->DB
  - DB->DR
  - DR->S
  - PC++

CS:IP指向该指令的第二个字,2000h

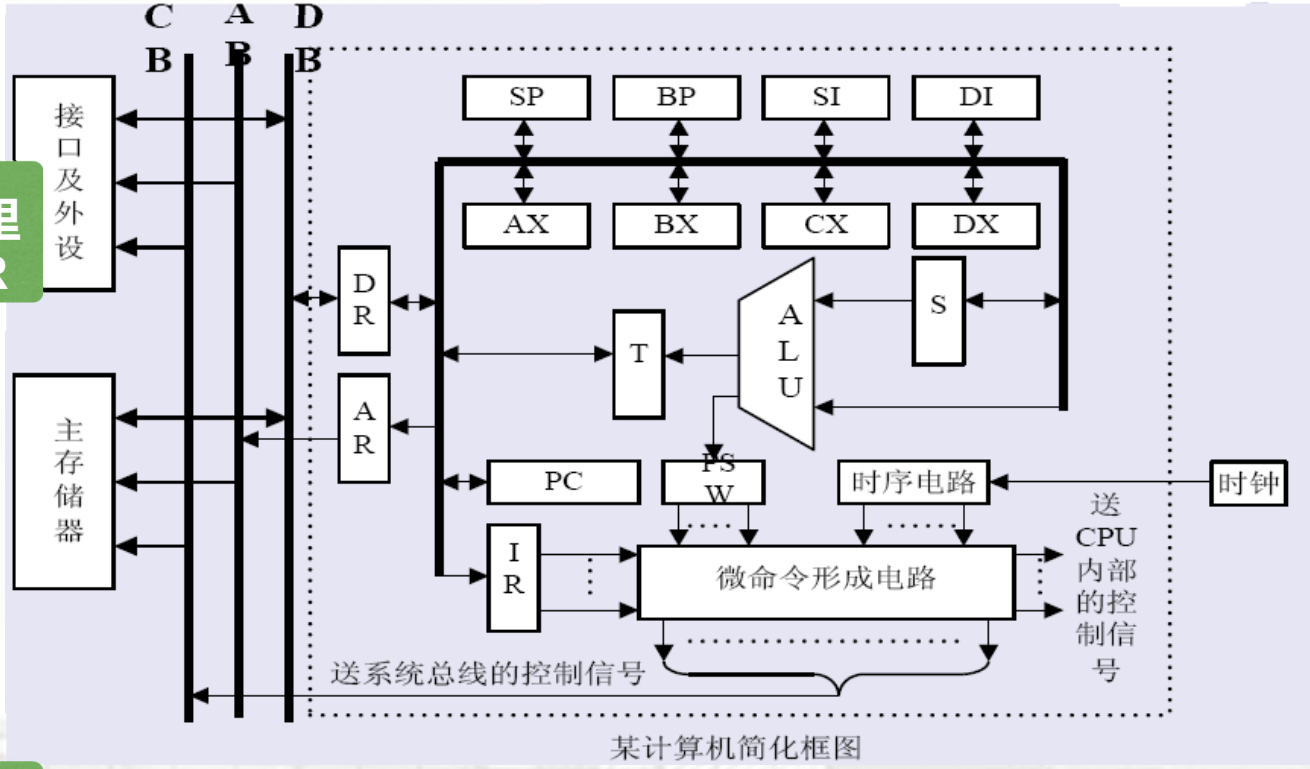
内存里的AR

内存里的DR

内存

把2000h的数直接放入S

IP指针自加1



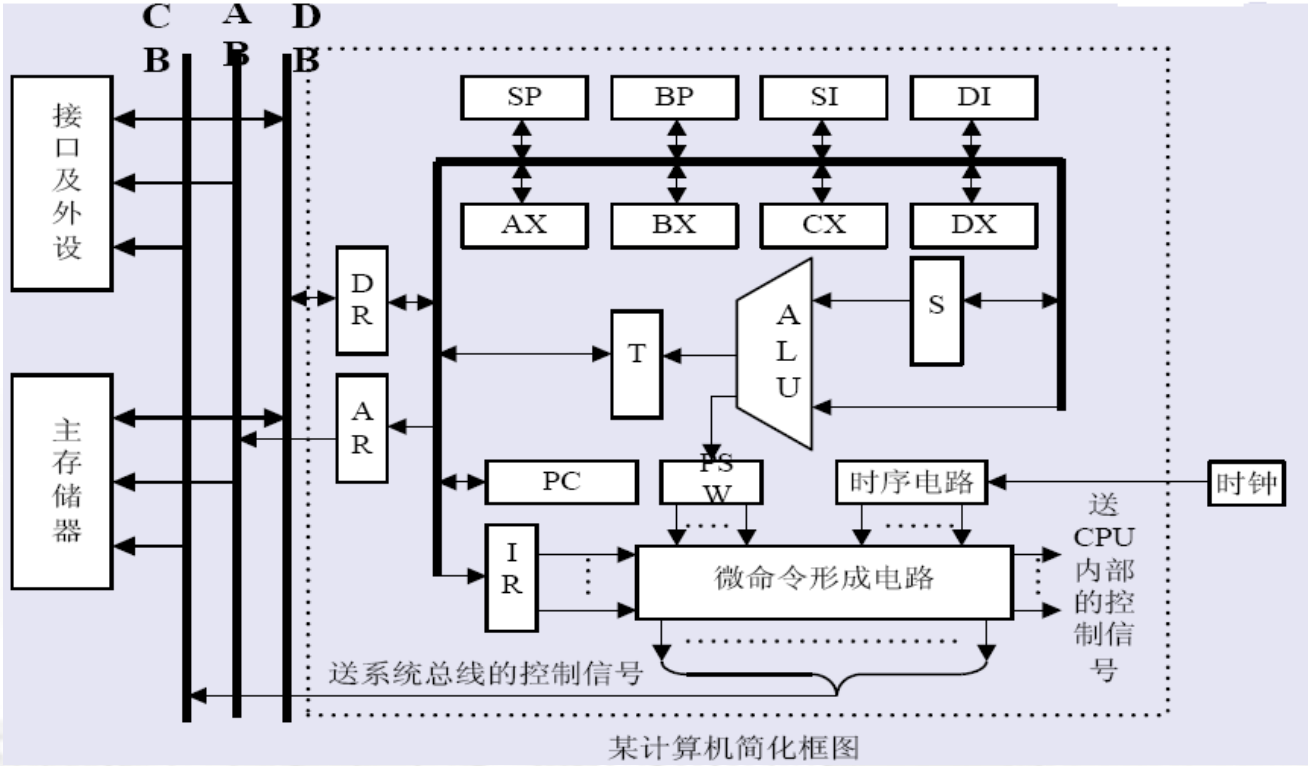
某计算机简化框图

深入CPU指令执行过程

以指令add ax, 2000h为例

- **执行Execute**
  - **AX->ALU**
  - **Add, ALU->T**

加法结果放入T



某计算机简化框图

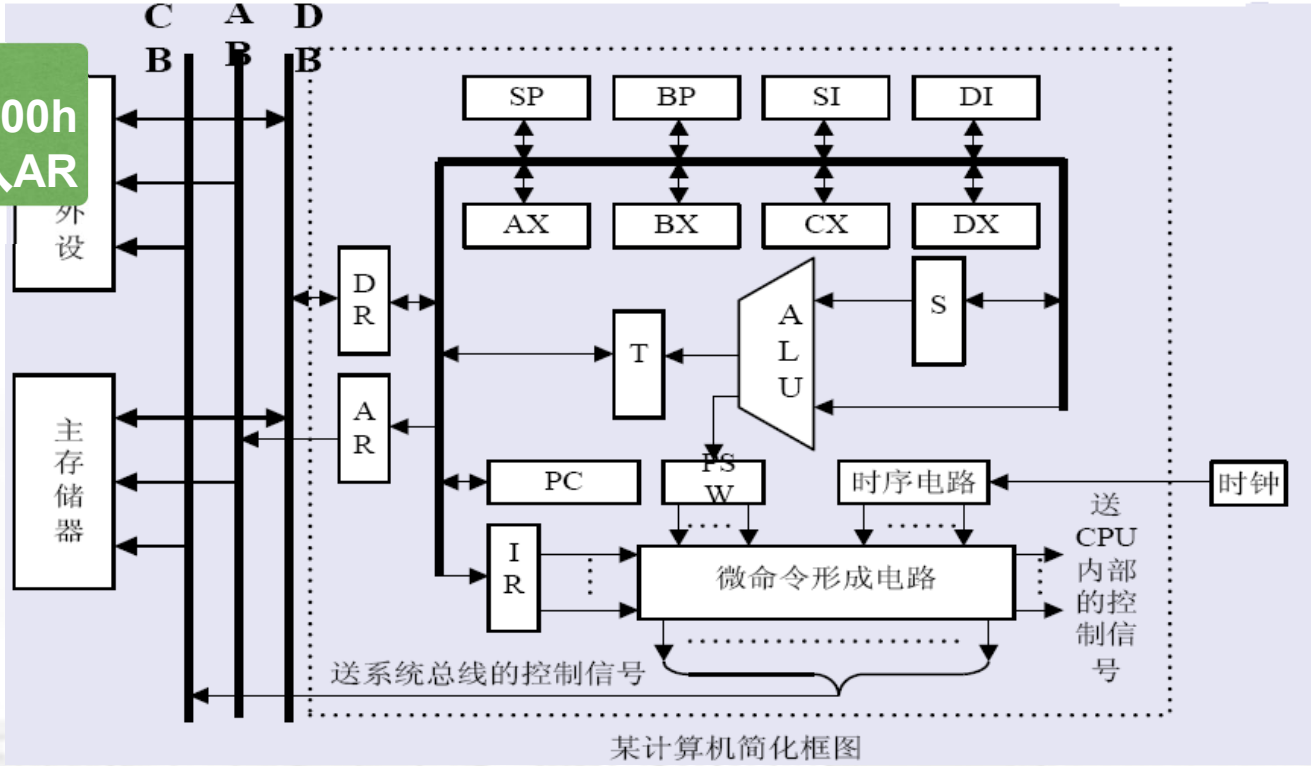
深入CPU指令执行过程

以指令add ax, 2000h为例

- **存储Store**
  - **T->AX**

结果T送入DR

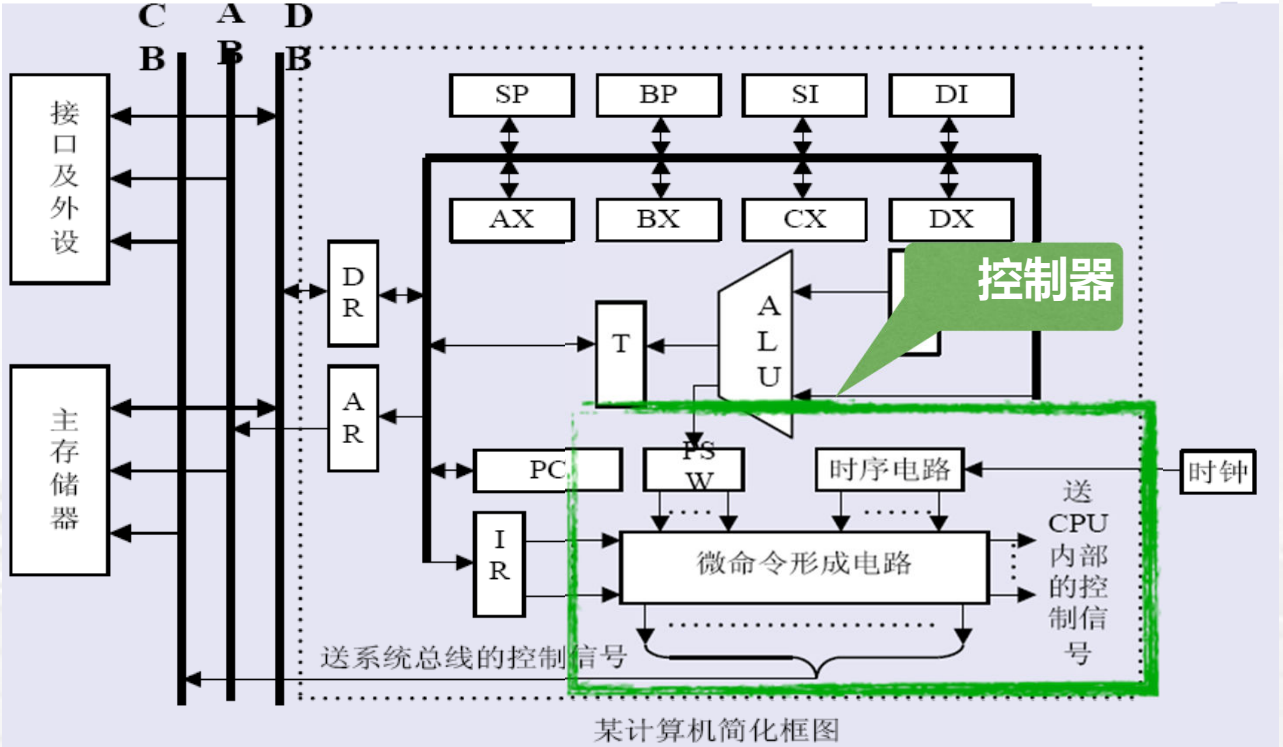
从IR中取出2000h送入AR



某计算机简化框图

深入CPU指令执行过程

- 这个例子给我们一些提示:
  - 一条机器指令，大约由10-30个更基础的器件级的操作构成，called微操作
  - 微操作需要按照顺序进行（微操作需要时序控制）
  - 微操作谁在控制?
    - CPU内的控制单元CU!

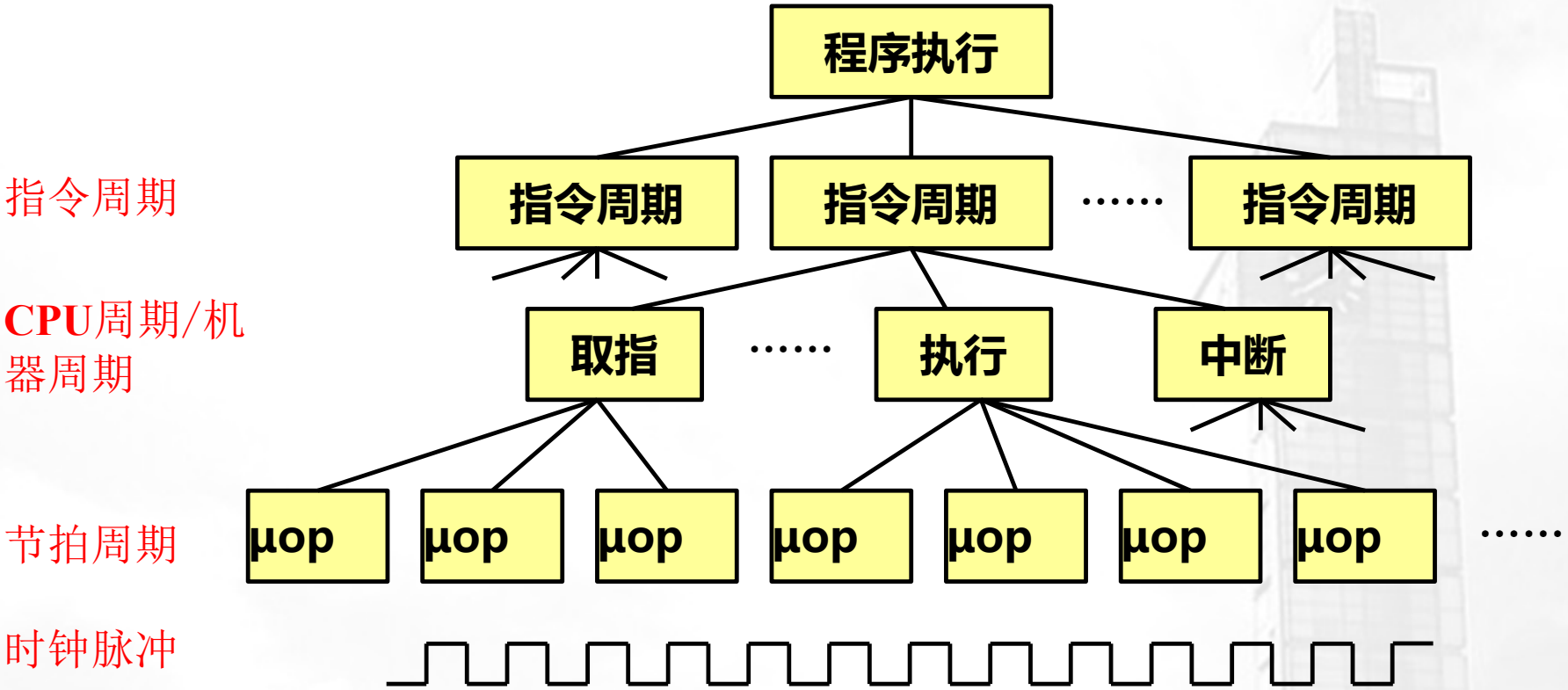


某计算机简化框图

关于时序与控制

- **时序系统的作用：将各种控制信号严格定时，在时间上相互配合完成某一功能。**
- **时序信号通常划分为4级：**
  - **指令周期**
  - **机器周期（CPU周期）**
  - **时钟周期（节拍周期）**
  - **[时钟脉冲、节拍脉冲]**

关于时序与控制

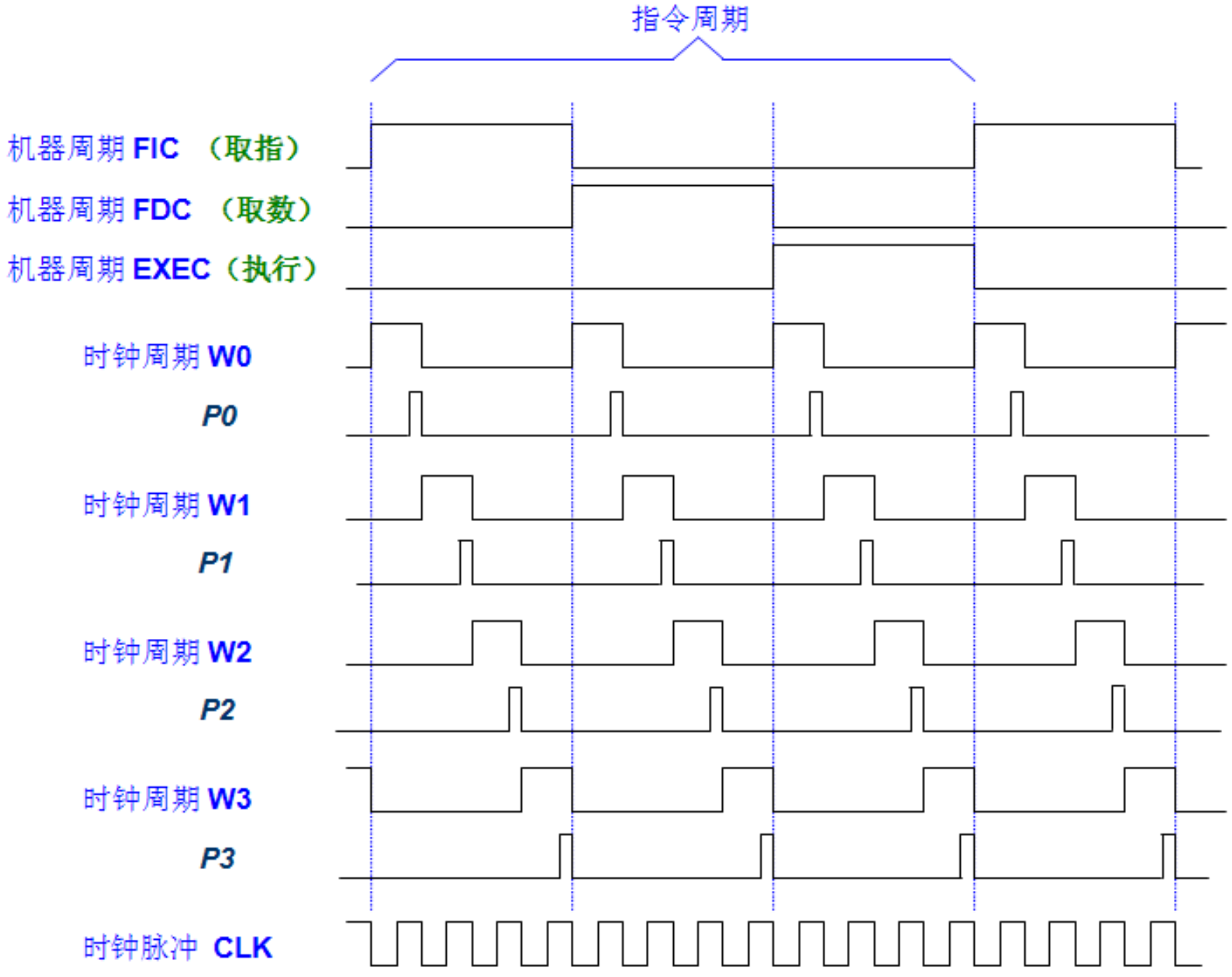


指令周期：指令执行时间，功能不同，指令周期不一定相同

CPU/机器周期：CPU对内存一次读写，包含的微操作个数不等，不一定相同

节拍周期：某种最复杂的微操作所需要的时间

关于时序与控制


指令周期
机器周期 FIC （取指）
机器周期 FDC （取数）
机器周期 EXEC（执行）
时钟周期 W0
P0
时钟周期 W1
P1
时钟周期 W2
P2
时钟周期 W3
P3
时钟脉冲 CLK

关于时序与控制

- 时序控制方式：
  - 同步、异步、联合
- 同步控制
  - 指令执行或指令中每个控制信号都由事先确定的统一的时序信号进行统一控制。
  - 可能有具体差异：
    - 定长的机器周期，定长的指令周期
    - 定长的机器周期，变长的指令周期
    - 变长的机器周期，变长的指令周期
    - 折中方案

关于时序与控制

- **异步控制**

当控制器发出某一操作控制信号后，等待执行部件完成操作后发回“回答(ACK)”信号，再开始新的操作。没有统一的时钟对信号进行同步。每条指令的指令周期可由多少不等的机器周期数组成

- **联合控制** **同步控制和异步控制相结合的方式。**

大部分微操作序列安排在固定的机器周期中，对某些时间难以确定的操作则采用“应答”方式。

如：CPU访问存储器时，依靠其送来的“READY”信号作为读 / 写周期的结束。

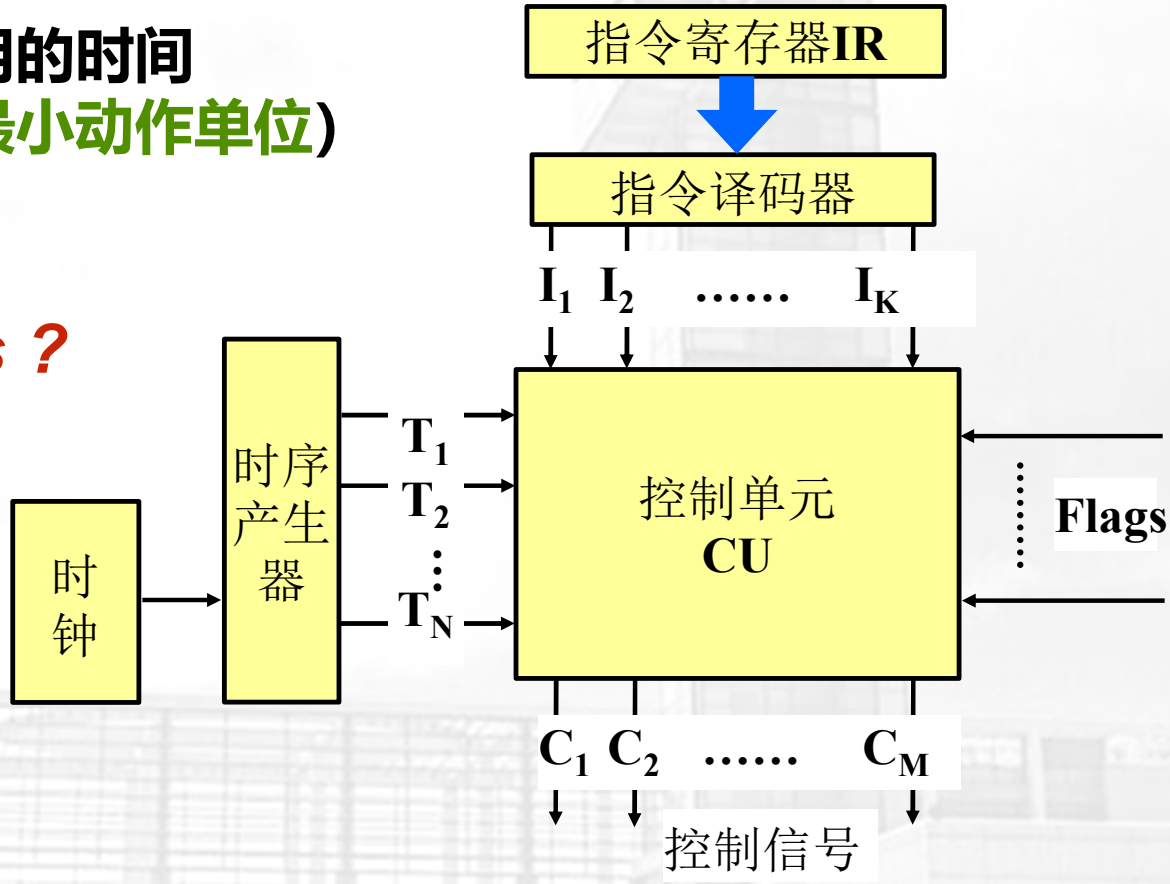
关于时序与控制

总结：指令执行过程涉及三种时序信号（或四种）

- **指令周期：执行一条指令所用的时间**
- **CPU周期：完成一个子周期（一组微操作）所用的时间**
- **节拍周期：完成一个微操作所用的时间**
- **（时钟脉冲：微操作过程中的最小动作单位）**

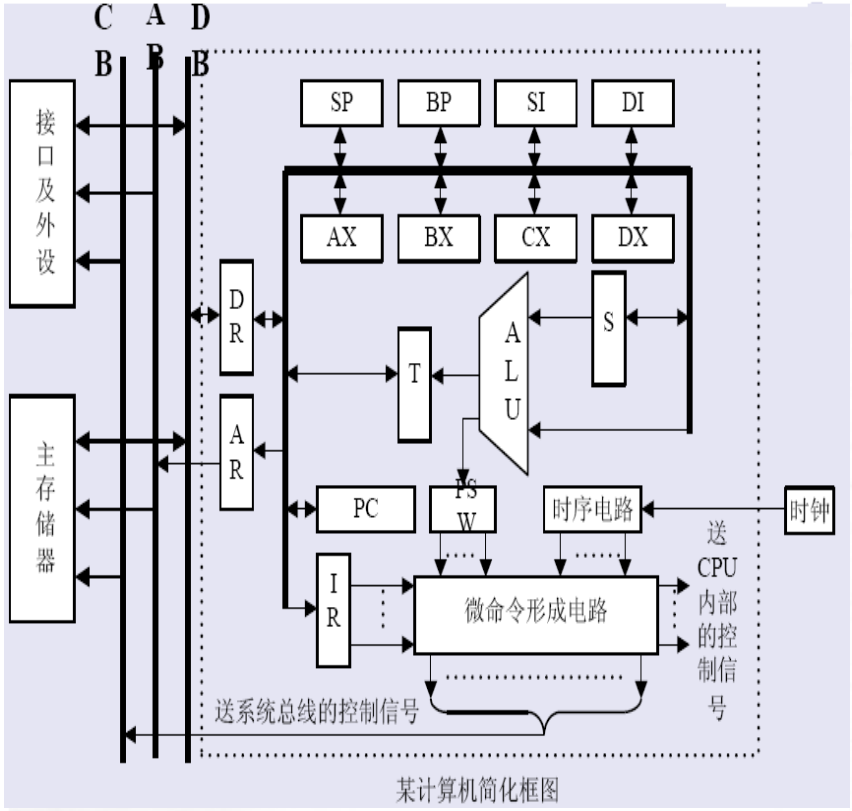
Then, why timing matters ?

CPU执行程序有严格的时间顺序性，通常利用时序电路为控制器提供所需的时序信号。最基本的时序信号为节拍，它由节拍脉冲发生器产生。



关于时序与控制

- 计算机简单框图



某计算机简化框图

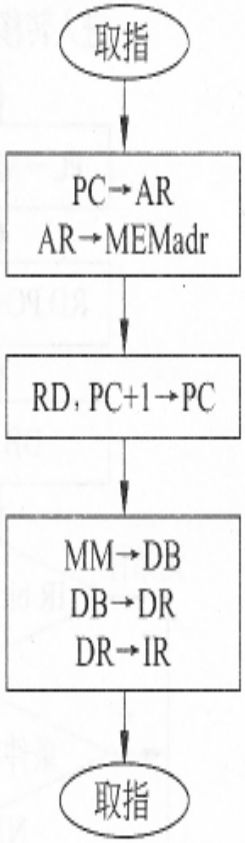


图 5-7 取指过程

Then, why Control Unit ?

We need CU, coz …

不同指令的操作序列需要给出…

不同部件的先/后顺序需要编排…

不同部件的功能/逻辑需要设定…

不同部件的数据I/O需要编排…

CU的主要作用：

发出满足一定时序关系的信号，实现指令系统所规定的各条指令的功能，并保证计算机系统正常运行。

关于时序与控制

- 总结一下，为什么需要控制器&控制器到底干什么事?
  - 一条汇编指令实现由一组微操作序列构成（时序+多个)
  - 一个微操作由不同部件配合完成(CPU内/外)
  - 部件之间也有时序关系(部件工作的先/后)
- 编排微操作时序和部件工作时序，是控制器的任务!

控制器内部根据时序信号和当前微操作命令对其它部件的输入、输出和控制信号进行控制

控制器的设计

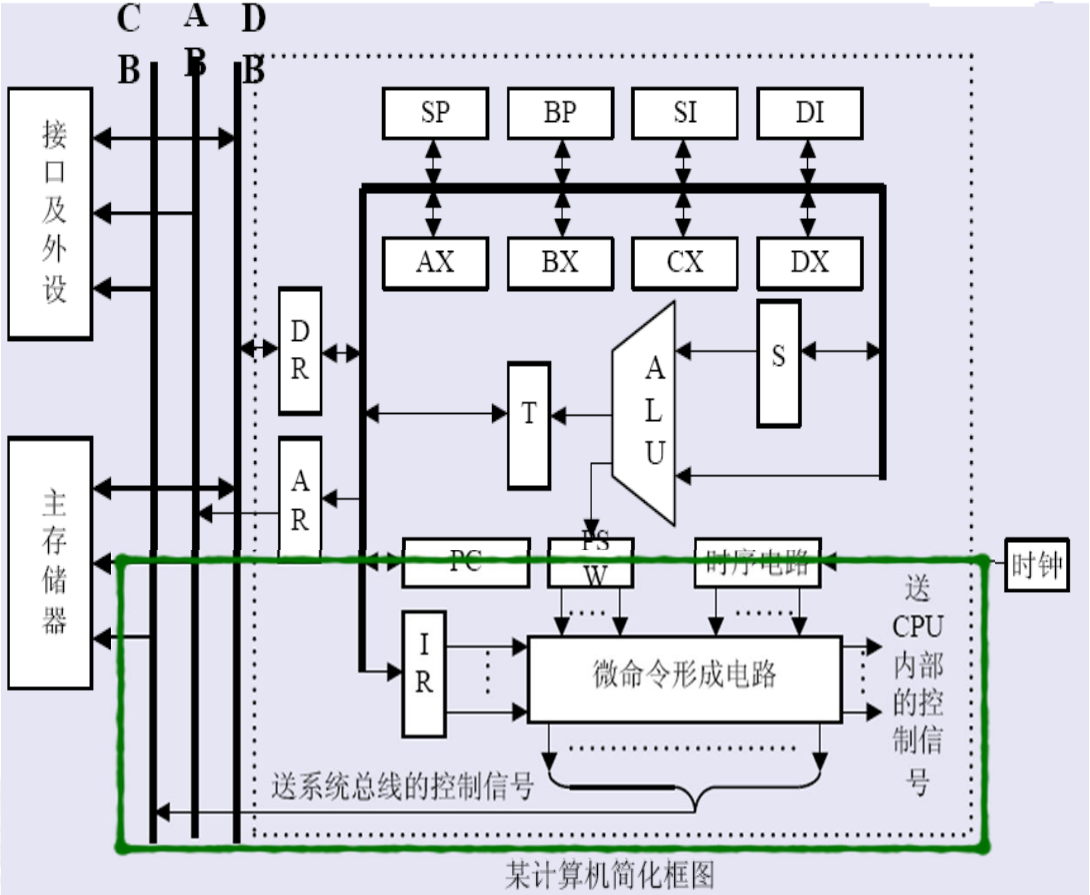
- 微操作：处理器（CPU）的基本或原子操作。
  - CPU可以实现的、不可分解的操作动作
  - 以含有一个寄存器传递（移进、移出）操作为标志
- 每一个微操作是通过控制器将控制信号发送到相关部件上引起部件动作而完成的。
  - 这些控制微操作完成的控制信号称为微命令
  - 微命令是由控制器产生的

$$\underbrace{AR \leftarrow PC;}_{\text{微操作}} \quad \underbrace{PC_{out},\ AR_{in}}_{\text{微命令}}$$

控制器的设计

- Then, How the Controller controls ?
- 相当多的一组并行的控制信号
  - 系统总结控制信号
  - CPU内部控制信号
  - 如：PC_{in}, PC_{out}, PC+1, IR_{in}, IR_{out} ...

- 计算机简单框图



某计算机简化框图

控制器的设计

- 8086处理器的控制信号：

  PC_{in}为程序计数器的锁存输入控制信号；

  PC_{out}为程序计数器的输出允许控制信号；

  PC+1为程序计数器的自动增量（如自动加1）控制信号；

  IR_{in}为指令寄存器的锁存输入控制信号；

  IR_{out}为指令寄存器的输出允许控制信号；

  SP_{in}为指令寄存器的锁存输入控制信号；

  SP_{out}为指令寄存器的输出允许控制信号；

  SP+1为堆栈指示器的自动增量（如自动加n）控制信号；

  SP-1为堆栈指示器的自动减量（如自动减n）控制信号；

  Ri_{in}为通用寄存器Ri（$0 \leqslant i \leqslant n-1$）的锁存输入控制信号；

  Ri_{out}为通用寄存器Ri（$0 \leqslant i \leqslant n-1$）的输出允许控制信号；

控制器的设计

- 8086处理器的控制信号：

  Y_{in}为暂存器Y的锁存输入控制信号；

  Z_{out}为暂存器Z的输出允许控制信号；

  AR_{in}为地址寄存器向CPU内部总线的锁存输入控制信号；

  AR_{out}为地址寄存器面向系统总线的输出允许控制信号；

  DRI_{in}为双端口数据寄存器面向CPU内部总线的锁存输入控制信号；

  DRI_{out}为双端口数据寄存器面向CPU内部总线的输出允许控制信号；

  DRS_{in}为双端口数据寄存器面向系统总线的锁存输入控制信号；

  DRS_{out}为双端口数据寄存器面向系统总线的输出允许控制信号；

  Mread为从主存储器读出信息的读控制信号；

  Mwrite为将信息写入到主存储器的写控制信号；

  IOread为从I/O设备输入信息的读控制信号；

  IOwrite为将信息写入到I/O设备的写控制信号；

控制器的设计

- 8086处理器的控制信号：

  ADD为加载至ALU的加法运算控制信号；

  SUB为加载至ALU的减法运算控制信号；

  AND为加载至ALU的逻辑与运算控制信号；

  OR为加载至ALU的逻辑或运算控制信号；

  SHL为加载至ALU的逻辑左移控制信号；

  SHR为加载至ALU的逻辑右移控制信号；

  ROL为加载至ALU的循环左移控制信号；

  ROR为加载至ALU的循环右移控制信号；

  （以上共31个控制信号）

  ……

控制器的设计

- 揭晓这两种设计之前，想几个问题：
  - 控制单元，它的输出是什么？
  - 众多（汇编）指令对应的微操作需要存储，如何存储？
  - 如何根据时序信号和操作码产生输出？
    - 带有时序的逻辑关系如何表达出来？

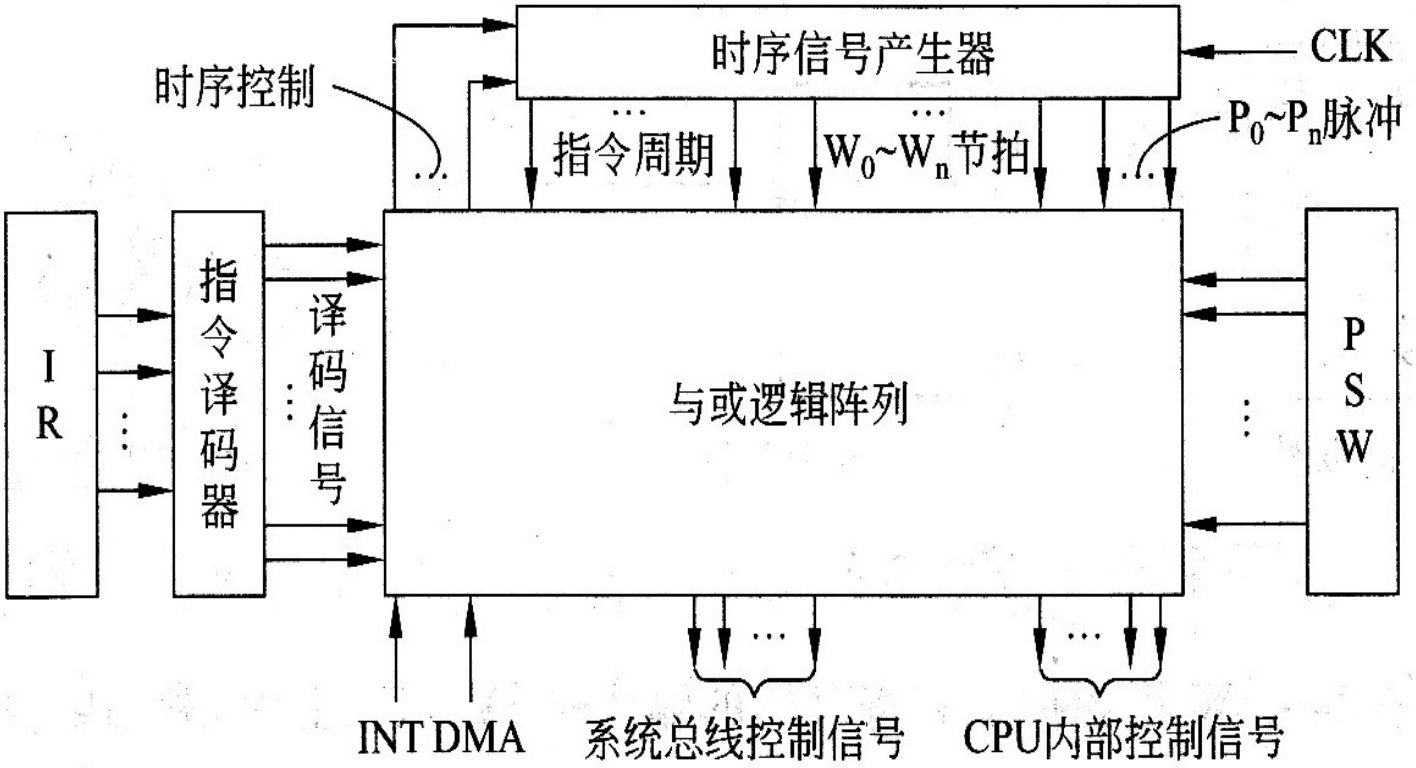
控制器的设计

- 控制器有两种实现方式
  - 组合逻辑控制器（硬布线/硬编码）
    - 硬，真的硬~
  - 微程序控制器

组合逻辑控制器

构成：

- **时序信号产生器**
  - **指令周期**
  - **节拍(W0, W1, W2 …)**
  - **触发脉冲(P0, P1, P2…)**
- **译码器**
- **PSW**
- **与或逻辑阵列**



逻辑阵列极端复杂：把各种指令的微指令序列/时序/控制/反馈都要硬编码在阵列里！

How ??~!

组合逻辑控制器

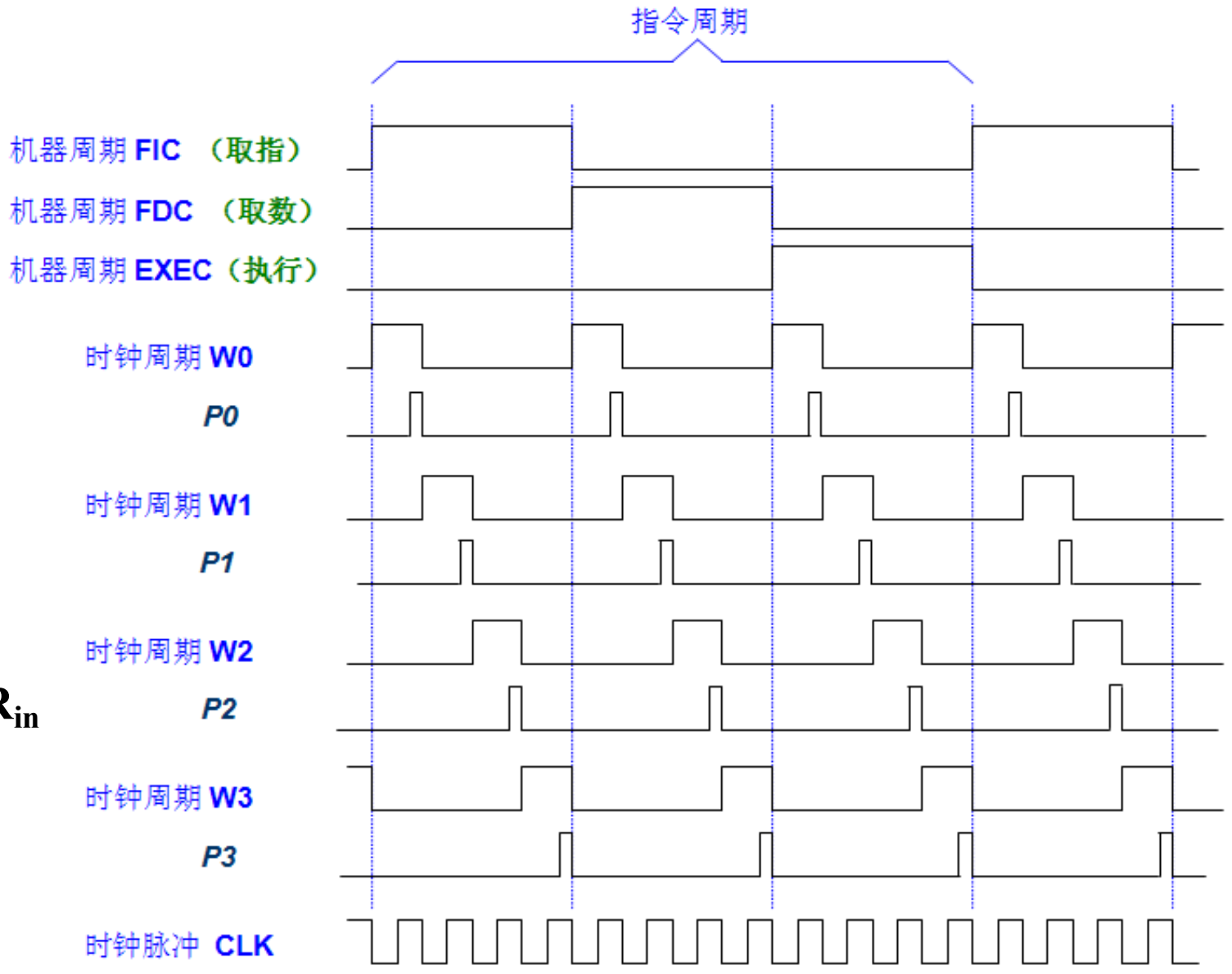
FIC 微操作命令

W_0　PC_{out}，AR_{in}

W_1　AR→AB，
RD，PC+1

W_2　MM→DB，DR_{in}

W_3　DR_{out}，IR_{in}



组合逻辑控制器

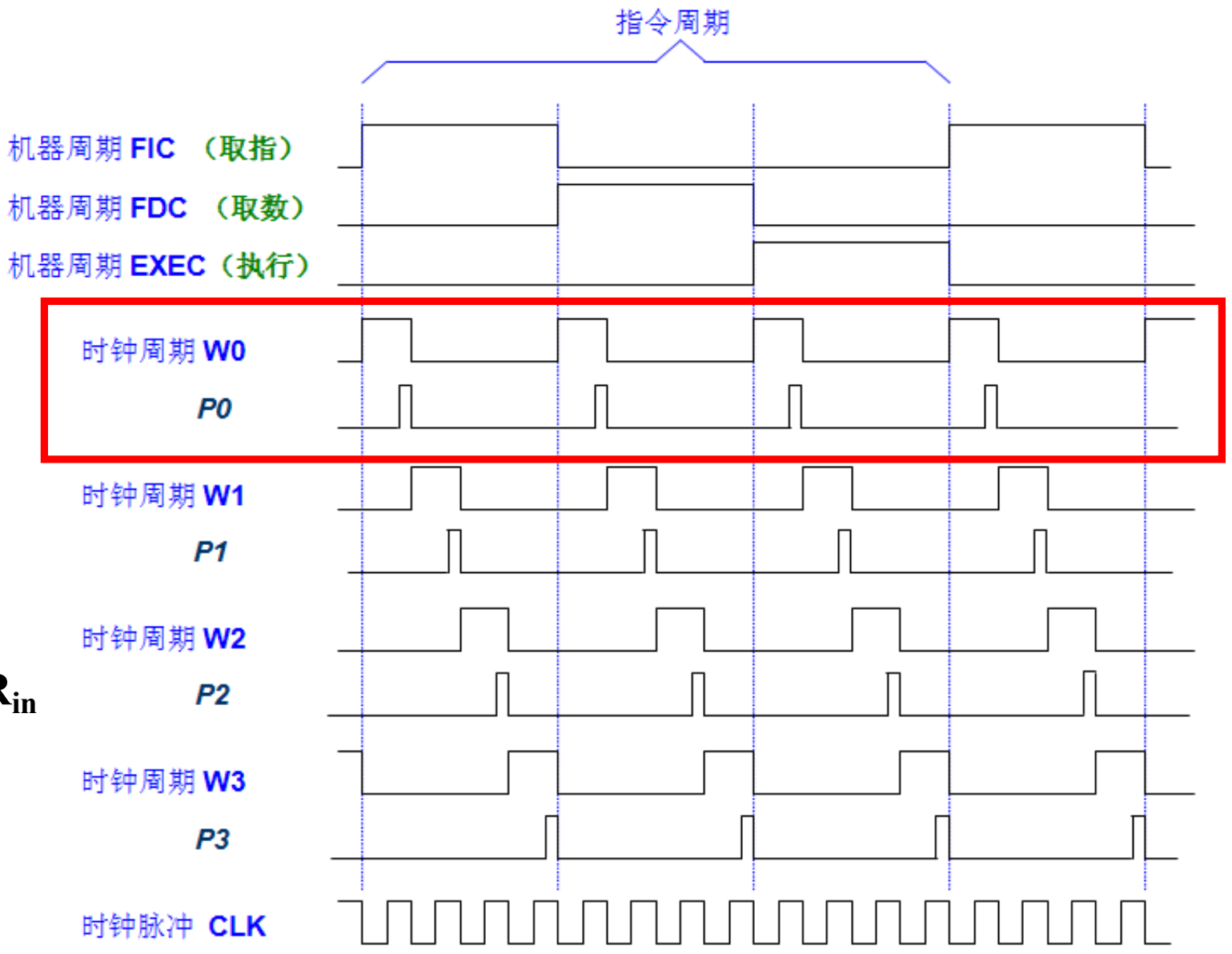
FIC 微操作命令

W_0　PC_{out}，AR_{in}

W_1　AR→AB，
RD，PC+1

W_2　MM→DB，DR_{in}

W_3　DR_{out}，IR_{in}



组合逻辑控制器

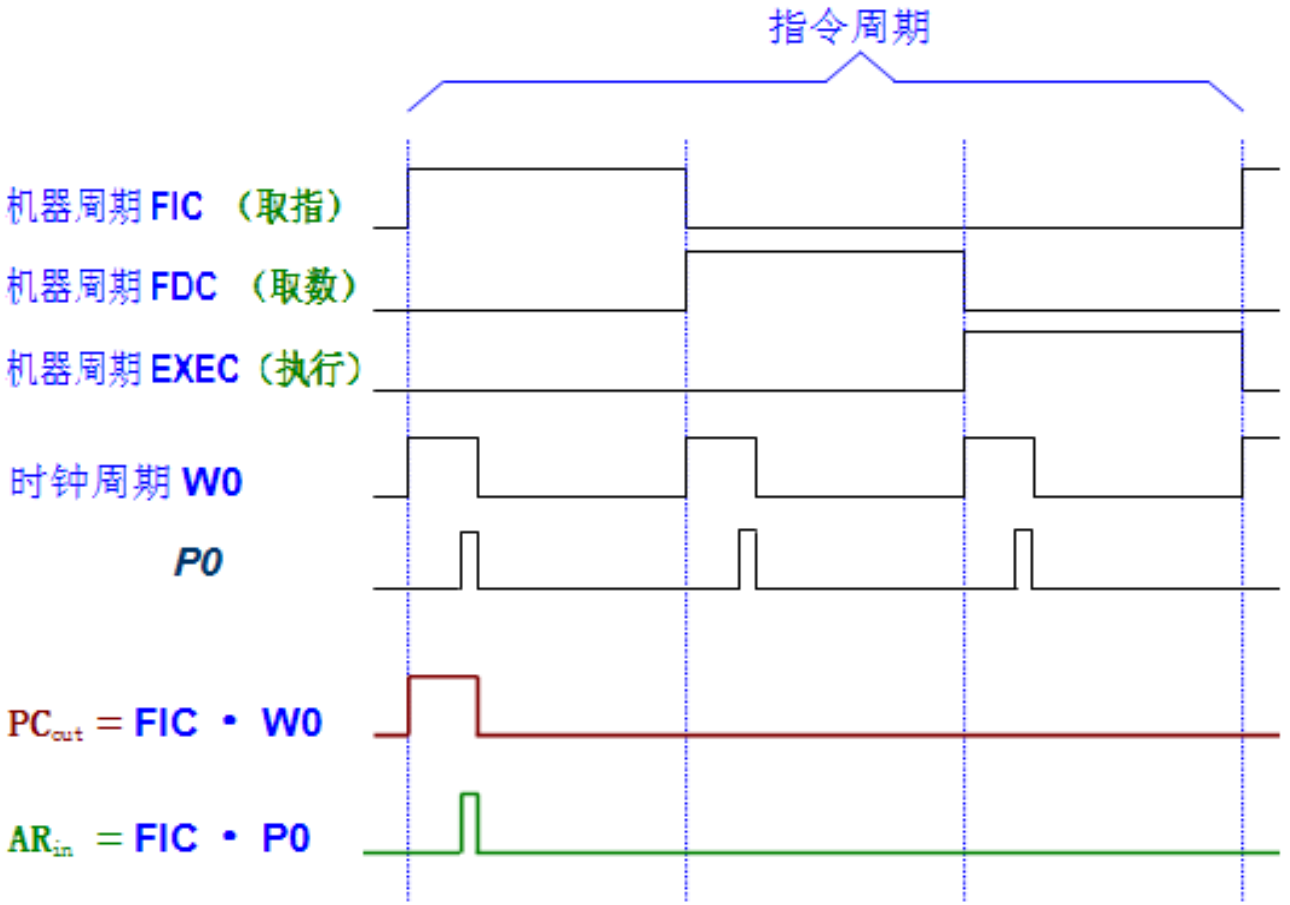
FIC 微操作命令

W_0　PC_{out}，AR_{in}

W_1　AR→AB，
RD，PC+1

W_2　MM→DB，DR_{in}

W_3　DR_{out}，IR_{in}



组合逻辑控制器

1. 取指周期中产生的控制信号相同。
2. 通常，一个控制信号在若干条指令的某些机器周期中都需要，应将它们组合起来。

   如：$PC_{out} = FIC \cdot W_0 + FDC \cdot$ 双字指令 $\cdot W_0 + \cdots\cdots$

   所以，控制信号‘ + ’可以用两级门电路产生

   —— 第一级为与门，第二级为或门。 “组合逻辑电路”
3. 逻辑式的化简以减少逻辑电路规模。
4. 构成与或逻辑阵列。

组合逻辑控制器

- PC_{out}出现在：
  - 取指周期的W0节拍
  - 指令JZ执行周期（假设为第2个CPU周期M2）的T1节拍
  - 指令CALL（X）执行周期的T4节拍
  - ……
- 生成PC_{out}的逻辑表达式为：

$$PC_{out} = FIC \cdot W0 + T1 \cdot JZ(\text{相对寻址}) \cdot (ZF=1) + T4 \cdot CALL(\text{间接寻址}) + \cdots\cdots$$

描述了“当aa周期的第y节拍或当bb指令的cc周期时…，发出PCout信号；

也就是说：组合逻辑控制器，是用非常机械的方式去描述其逻辑的：枚举所有微操作所需的发生的条件，将其描述为与或门逻辑。

组合逻辑控制器

■ AR_{in}出现在：

- 取指周期的T1节拍
- 指令 MOV R0,X 和指令 MOV (R1),R0 执行周期的T1节拍
- 指令 SUB R0,(X) 执行周期的T1和T3节拍
- 指令 IN R0,P 和指令 OUT P,R0 执行周期的T1节拍
- 指令 PUSH R0 执行周期的T2节拍
- 指令 POP R0 执行周期的T1节拍
- 指令 CALL (X) 执行周期的T2和T4节拍
- 指令 RET 执行周期的T1节拍
- ……

■ 生成AR_{in}的逻辑表达式为：

组合逻辑控制器

■ **生成AR_{in}的逻辑表达式为：**

AR_{in}＝T1＋T4·MOV(源操作数直接寻址**+**目的操作数寄存器间接寻址**)＋(T4+T6) ·SUB(**源操作数间接寻址**)＋T4·(IN(**直接寻址**)+OUT(**直接寻址**))＋T5·PUSH＋T4·POP＋(T5+T7)·CALL(**间接寻址**)＋T4·RET＋……**

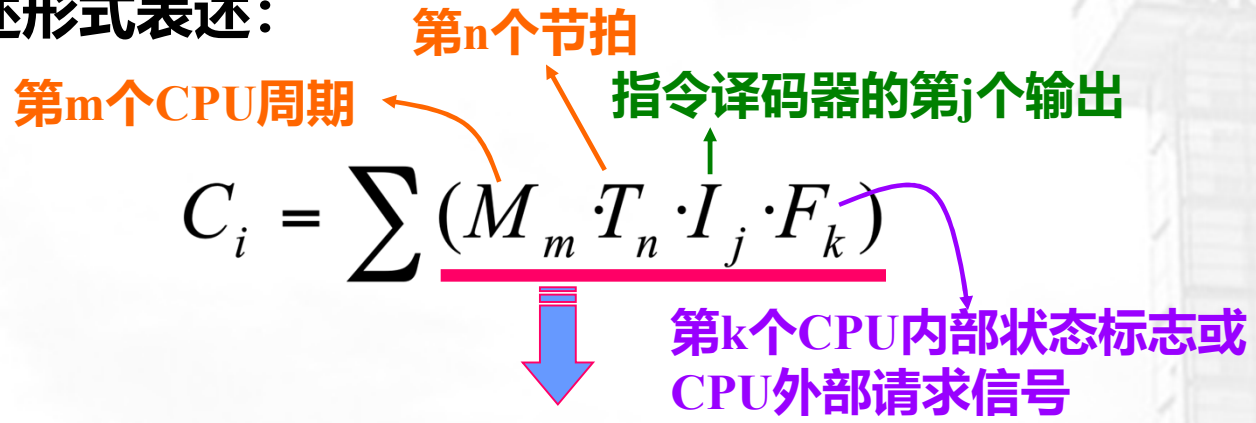
描述了“当aa周期的第y节拍或当bb指令的cc周期时…，发出PCout信号；

也就是说：组合逻辑控制器，是用非常机械的方式去描述其逻辑的：枚举所有微操作所需的发生的条件，将其描述为与或门逻辑。

组合逻辑控制器

- **组合逻辑控制器，它到底描述的是什么?**

由控制单元产生并加载到CPU内外的全部控制信号均可用下述形式表述：



在执行指令I_j时，若状态F_k满足要求，则在第m个机器周期M_m的第n个节拍T_n，控制单元发出C_i控制命令

所以称之为“基于组合逻辑的控制器”

组合逻辑控制器

- 组合逻辑控制单元，它的输出是什么?
  - 总线和内部的控制信号
- 众多（汇编）指令对应的微指令需要存储，如何存储?
  - 严格来说，组合逻辑控制单元并不单纯存储微程序;
- 如何根据时序信号和操作码产生输出?
  - 带有时序的逻辑关系如何表达出来?
    - 每一个控制信号的输出，由所有可能触发该信号的输入组合以“或”操作形式组合起来，每种输入组合内部是用“与”操作表达;
    - b.t.w“存储”了微程序和执行逻辑

组合逻辑控制器

小结：

- **每个控制信号的逻辑表达式就是一个与或逻辑方程式。**
- **将所有控制信号的与或逻辑电路组合在一起就构成了硬布线控制单元。**
- 时间信息、指令信息、状态信息是硬布线控制单元的输入，控制信号是硬布线控制单元的输出。

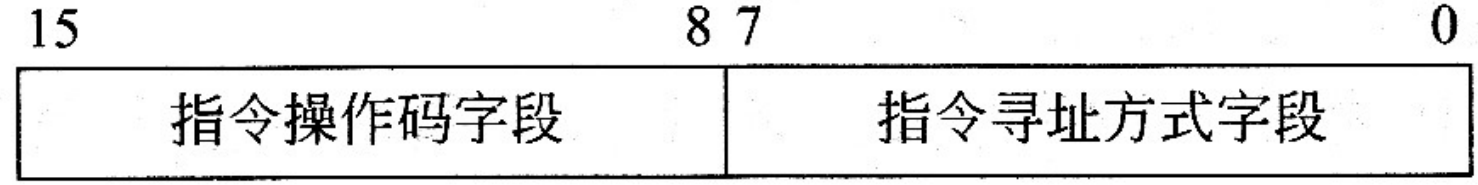
- **采用硬布线法设计控制器的特点：**
  - **一旦完成了控制器的设计，改变控制器行为的唯一方法就是重新设计控制单元 → 修改不灵活**
  - **使用PLD(PAL, PLA, GAL, FPGA)实现！**
  - **在现代复杂的处理器中，需要定义庞大的控制信号逻辑方程组 → 与或组合电路实现困难**
    - ➔ **微程序设计法**

CPU内部指令系统设计

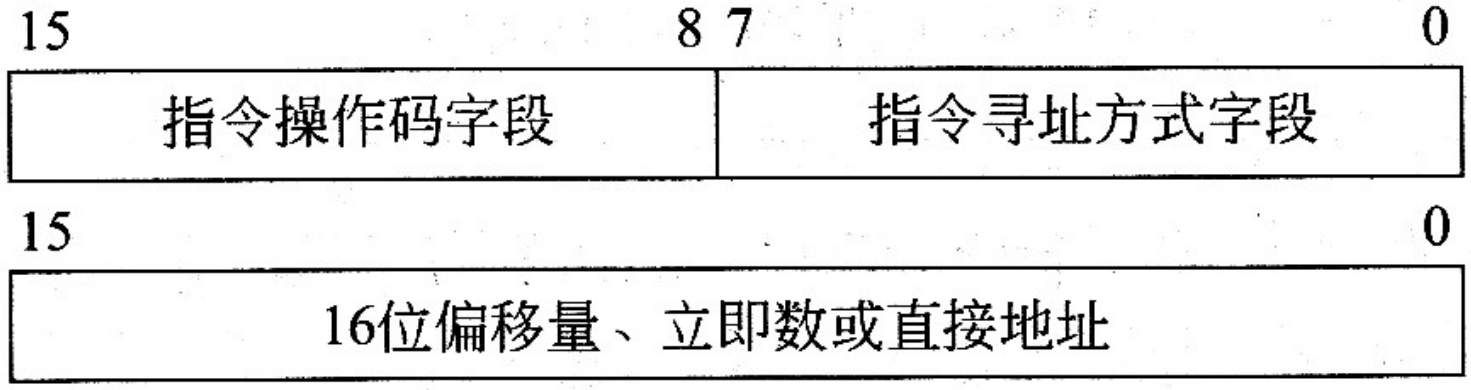
- 假设我们正在造一个CPU
  - 假设我们正在研发核心的指令系统….

- 两大步骤：
  - 给指令编码（设计编码规则）
  - 给指令解码（让机器按规则去识别编码）

Designing a dummy CPU~



(a) 单字指令



(b) 双字指令

图 5-3　指令码构成格式

Designing a dummy CPU~

操作码字段的编排

Designing a dummy CPU~

指令的设置

- 双操作数指令： 11条

  MOV、ADD、SUB、ADC、SBC、CMP、MUL、DIV、AND、OR、XOR

- 单操作数指令： 13条

  INC、DEC、PUSH、POP、NOT、SHL、SHR、SAR、ROL、ROR、RCL 、RCR、CALL

- 转移指令： 9条

  JMP、JZ、JNZ、JC、JNC、JG、JGE、JA、JAE

- 无操作数指令： 7条

  RET、IRET、NOP、CLI、STI、SWI、DAA

Designing a dummy CPU~

- 双操作数指令： 11条

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

0	0	0	0	X	X	X	X	双操作数指令
0	0	0	0	0	0	0	0	MOV
				0	0	0	1	ADD
				0	0	1	0	SUB
				0	0	1	1	ADC
				0	1	0	0	SBC
				0	1	0	1	CMP
				0	1	1	0	MUL
				0	1	1	1	DIV
				1	0	0	0	AND
				1	0	0	1	OR
				1	0	1	0	XOR
				1	0	1	1	

Designing a dummy CPU~

- 单操作数指令： 13条

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	X	X	X	X	双操作数指令							
0	0	0	0	1	1	1	0	X	X	X	X	单操作数指令			
0	0	0	0	1	1	1	0	0	0	0	0	CALL			
								0	0	0	1	INC			
								0	0	1	0	DEC			
								0	0	1	1	PUSH			
								0	1	0	0	POP			
								0	1	0	1	NOT			
								0	1	1	0	SHL			
								0	1	1	1	SHR			
								1	0	0	0	SAR			
								1	0	0	1	ROL			
								1	0	1	0	ROR			
								1	0	1	1	RCL			
								1	1	0	0	RCR			
								1	1	0	1				
								1	1	1	0				
								1	1	1	1				

Designing a dummy CPU~

- 转移指令： 9条

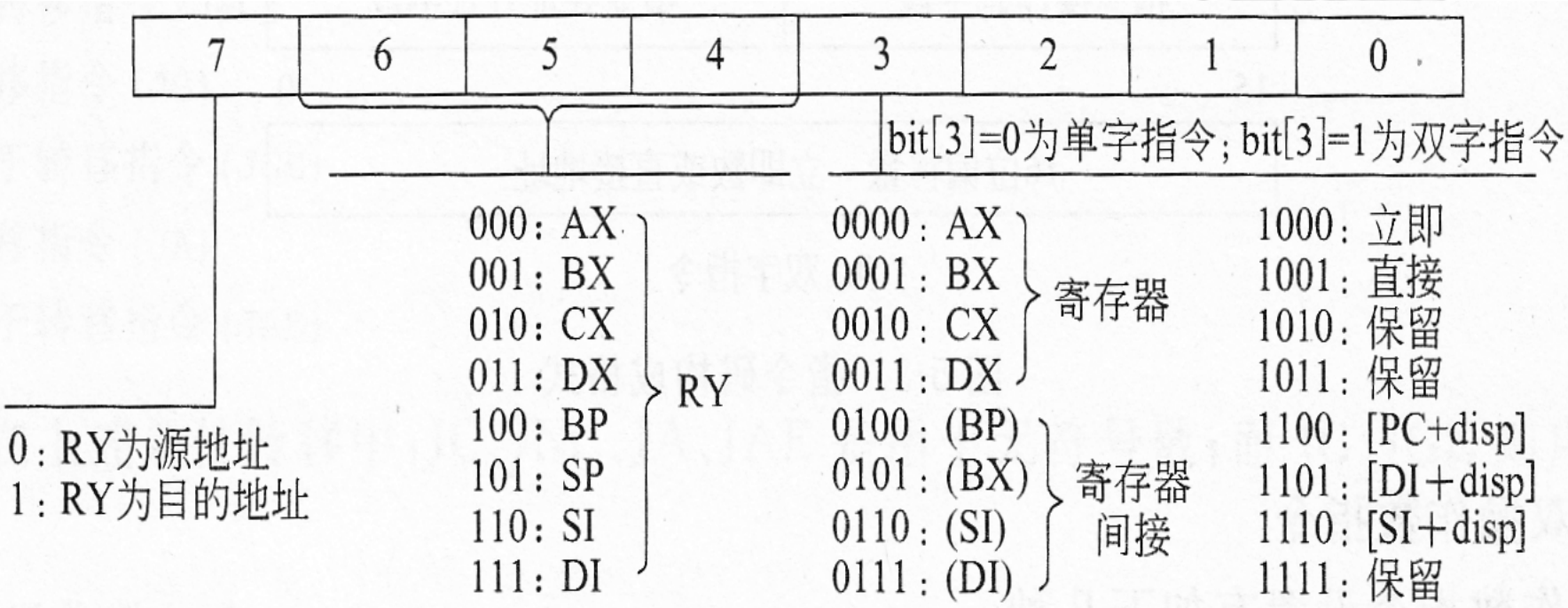
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	X	X	X	X									双操作数指令
0	0	0	0	1	1	1	0	X	X	X	X					单操作数指令
0	0	0	0	1	1	1	1	X	X	X	X					转移指令
0	0	0	0	1	1	1	1	0	0	0	0					JMP
								0	0	0	1					JZ
								0	0	1	0					JNZ
								0	0	1	1					JC
								0	1	0	0					JNC
								0	1	0	1					JG
								0	1	1	0					JGE
								0	1	1	1					JA
								1	0	0	0					JAE
								1	0	0	1					

Designing a dummy CPU~

- 无操作数指令： 7条

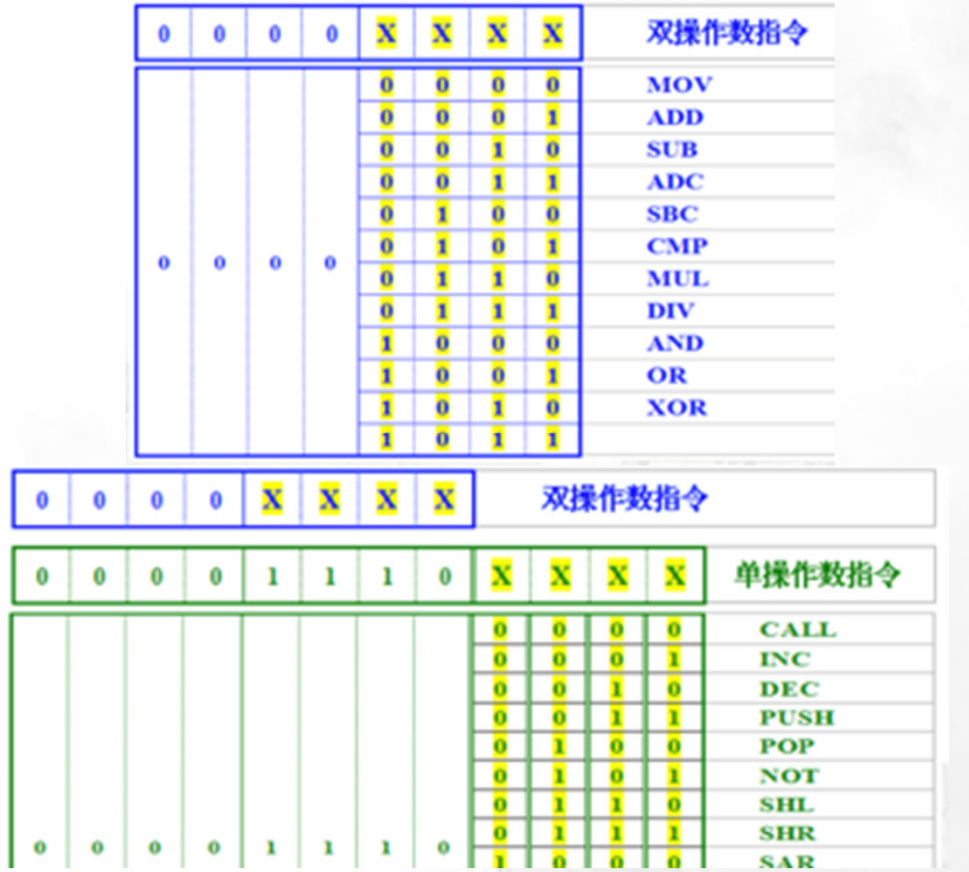
Designing a dummy CPU~

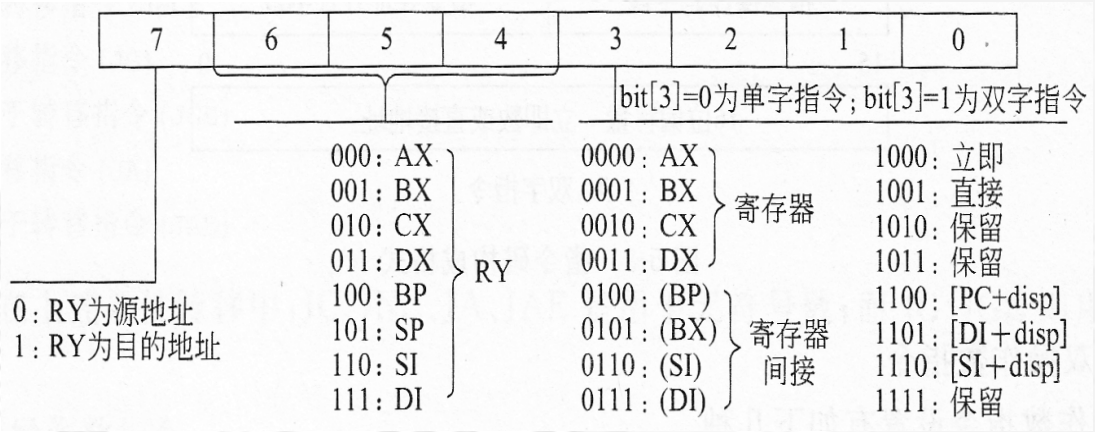
设计一个简单的寻址方式编码



Designing a dummy CPU~

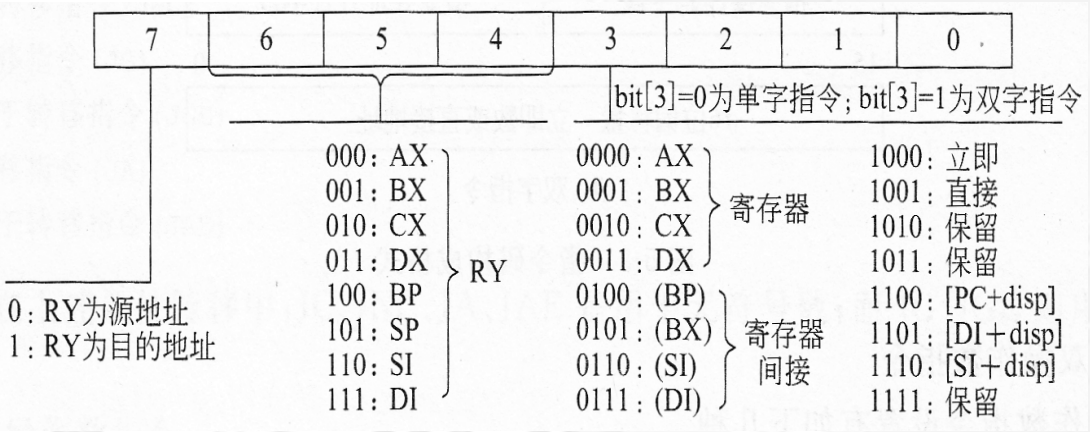
- 根据以上指令规范，我们可以为下面的指令给出操作码编码：
  - ADD AX [SI]
    - ?
  - Mov AX [2000H]
    - ?
  - INC [BX]
    - ?





Designing a dummy CPU~

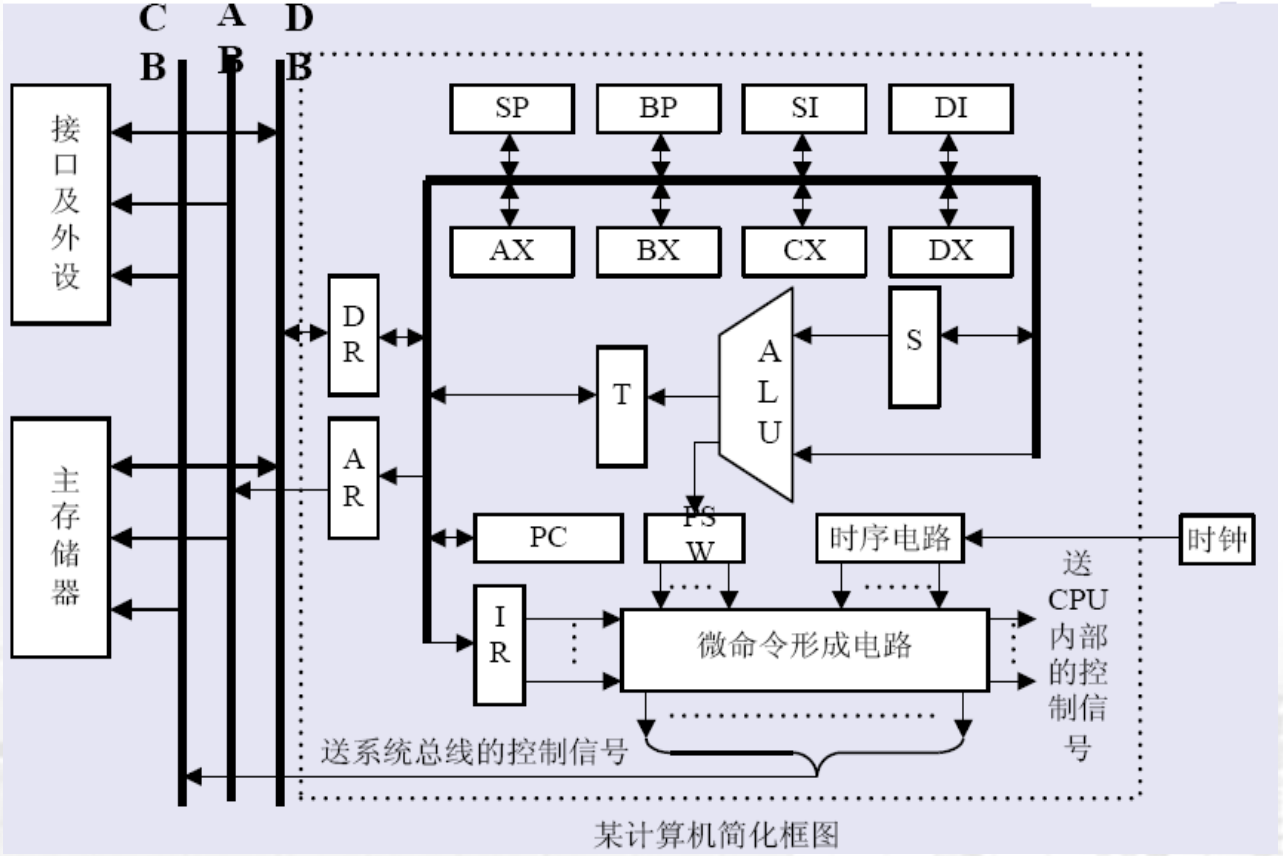
- 根据以上指令规范，我们可以为下面的指令给出操作码编码：
  - ADD AX [SI]
    - 01 86 (单字)
  - Mov AX [2000H]
    - 00 89 40 00 (双字)
  - INC [BX]
    - 0E 15 (单字)



Simulating a dummy CPU~

a) 取指令过程：

- 计算机简单框图



某计算机简化框图

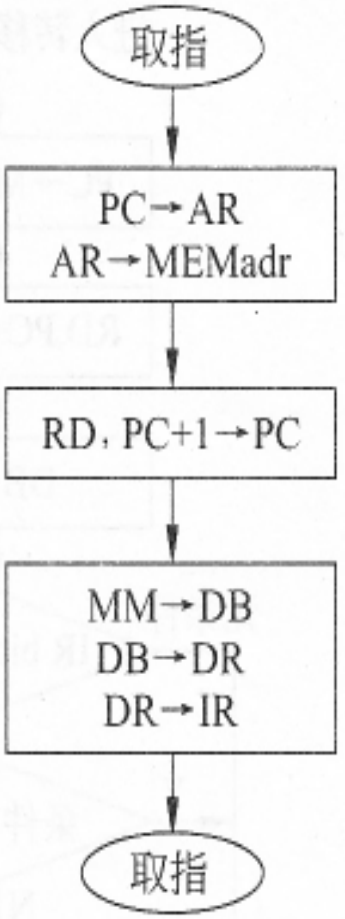
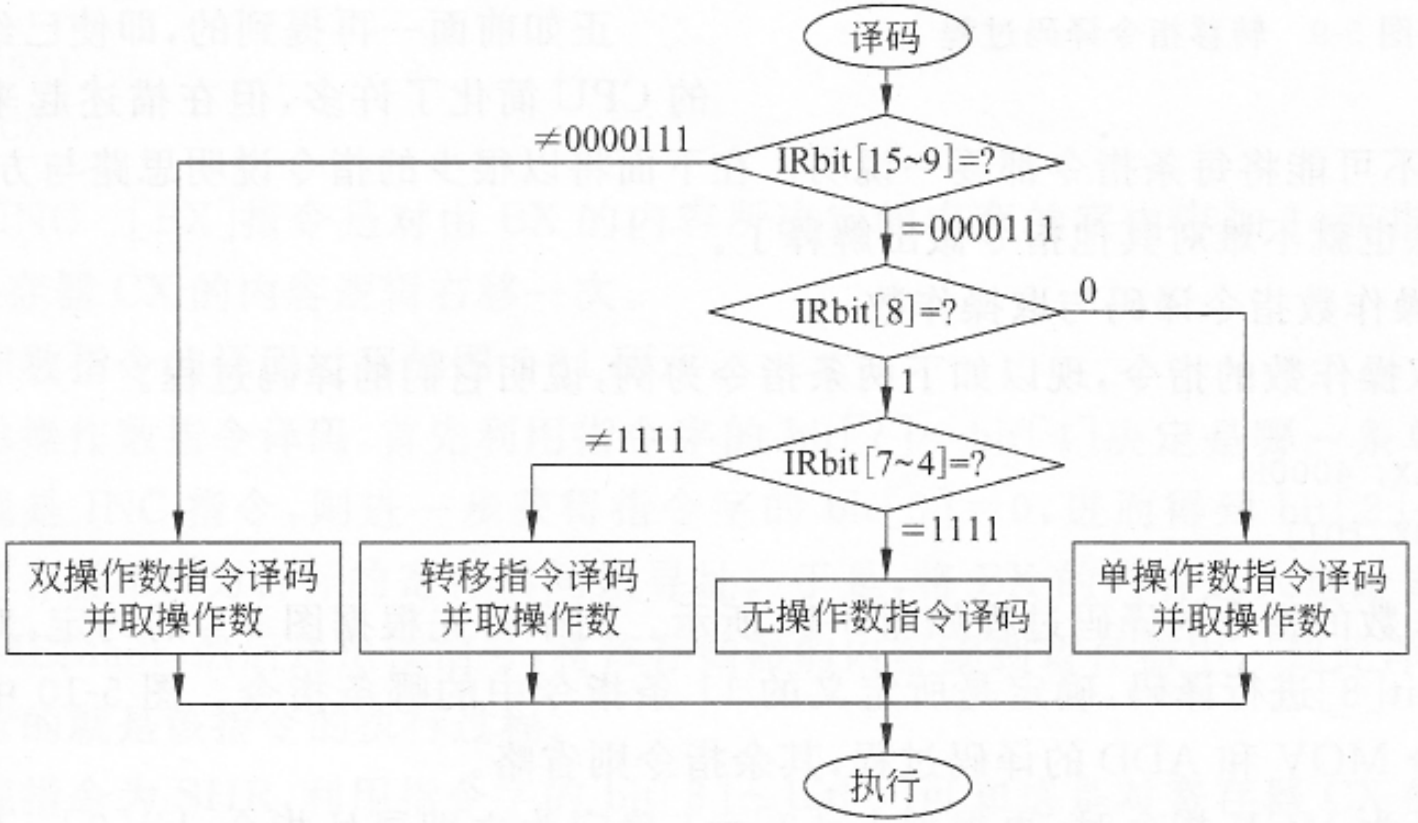


图 5-7 取指过程

Simulating a dummy CPU~

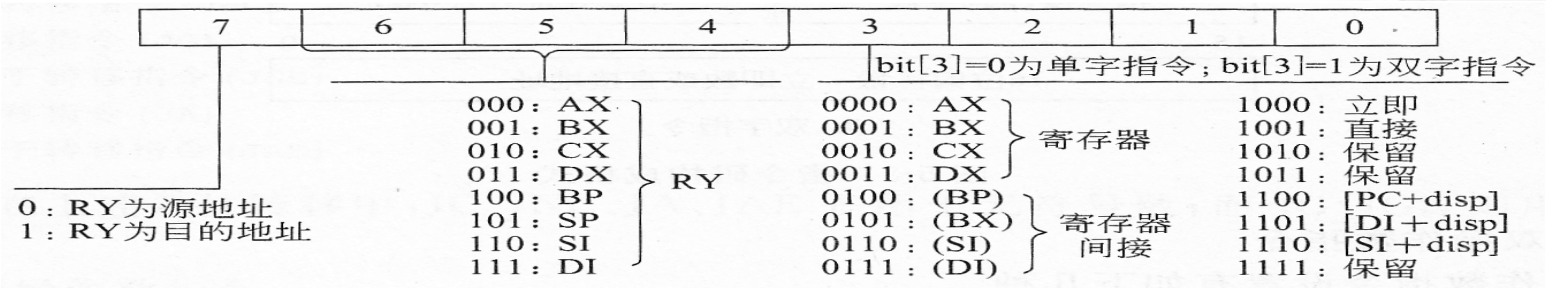
b) 译码取操作码过程：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	X	X	X	X	双操作数指令								
0	0	0	0	1	1	1	0	X	X	X	X	单操作数指令				
0	0	0	0	1	1	1	1	X	X	X	X	转移指令				
0	0	0	0	1	1	1	1	1	1	1	1	X	X	X	X	无操作数指令



Simulating a dummy CPU~

- 双操作数指令译码取操作数过程（续）：



- ADD AX [SI]
  - 01 86 (单字)
- Mov AX [2000H]
  - 00 89 40 00 (双字)
- INC [BX]
  - 0E 15 (单字)

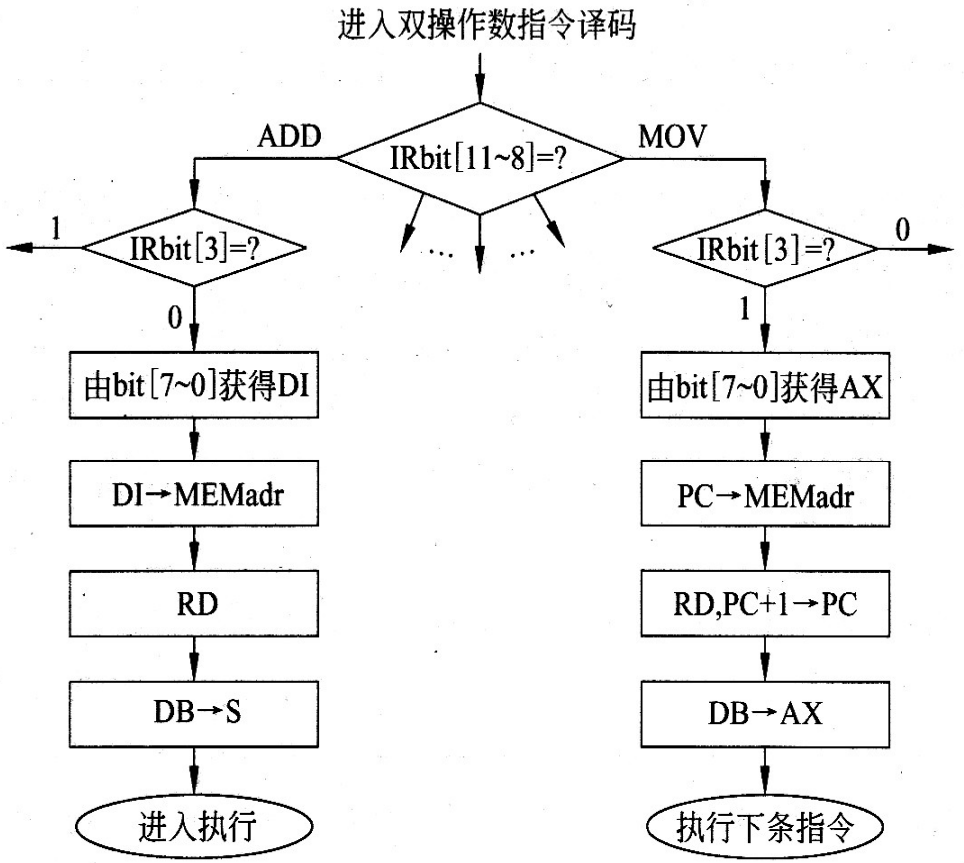


图 5-10　双操作数指令译码过程

Simulating a dummy CPU~

- 单操作数指令译码取操作数过程：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	1	1	1	0	X	X	X	X					单操作数指令
0	0	0	0	1	1	1	0	0	0	0	0					CALL
								0	0	0	1					INC
								0	0	1	0					DEC
								0	0	1	1					PUSH
								0	1	0	0					POP
								0	1	0	1					NOT
								0	1	1	0					SHL
								0	1	1	1					SHR
								1	0	0	0					SAR
								1	0	0	1					ROL
								1	0	1	0					ROR
								1	0	1	1					RCL
								1	1	0	0					RCR
								1	1	0	1					
								1	1	1	0					
								1	1	1	1					

- ADD AX [SI]
  - 01 86 (单字)
- Mov AX [2000H]
  - 00 89 40 00 (双字)
- INC [BX]
  - 0E 15 (单字)

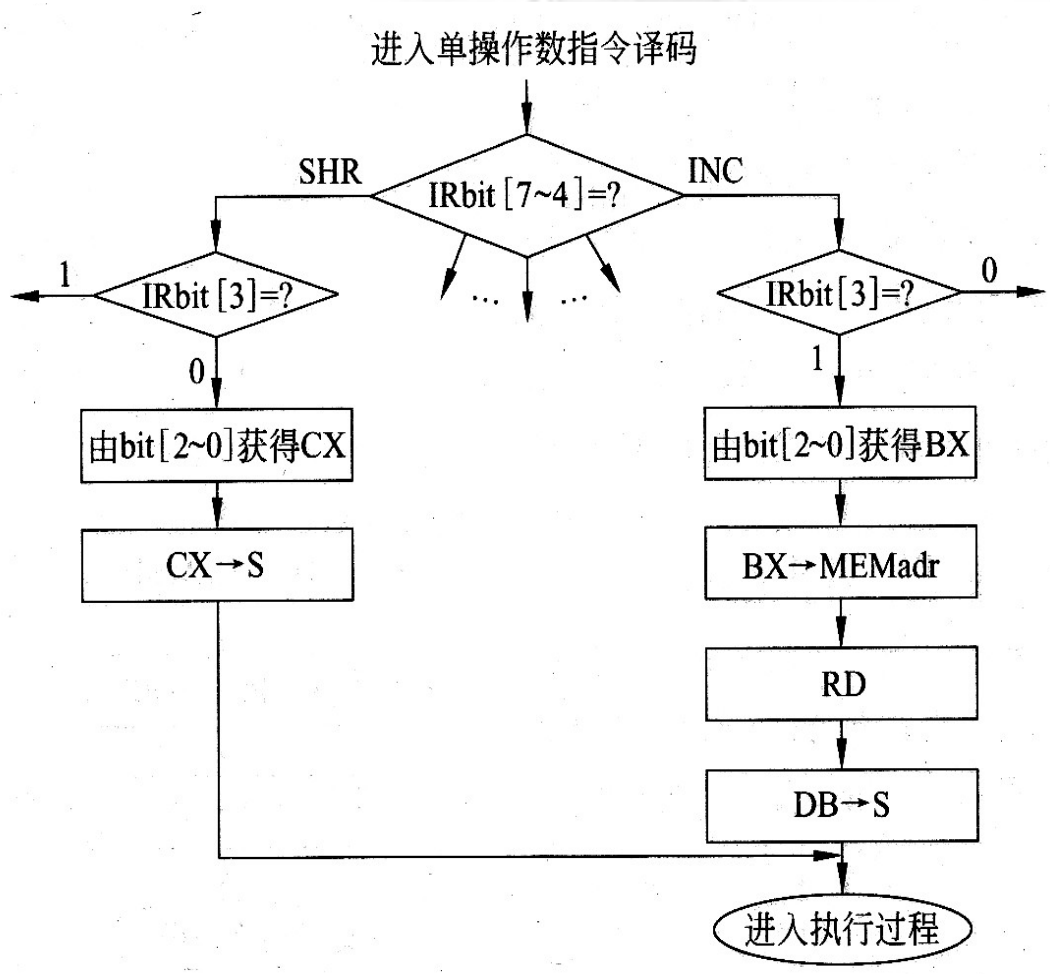
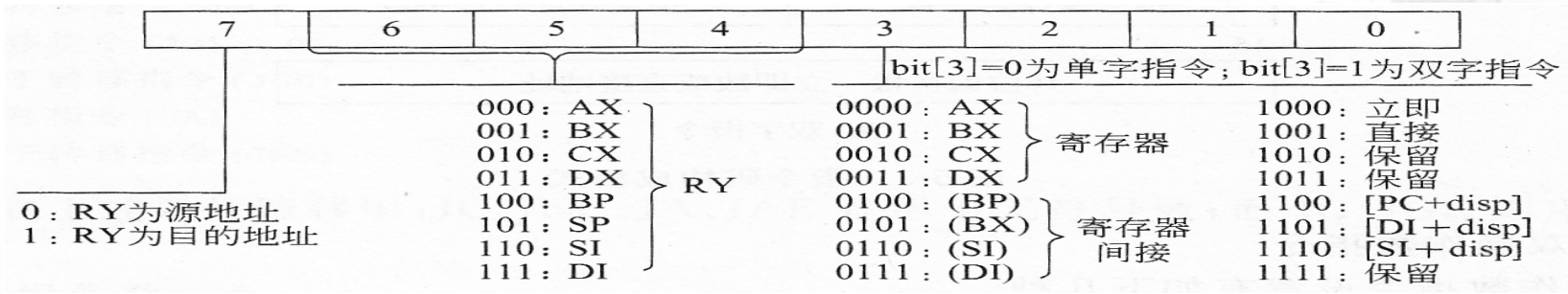


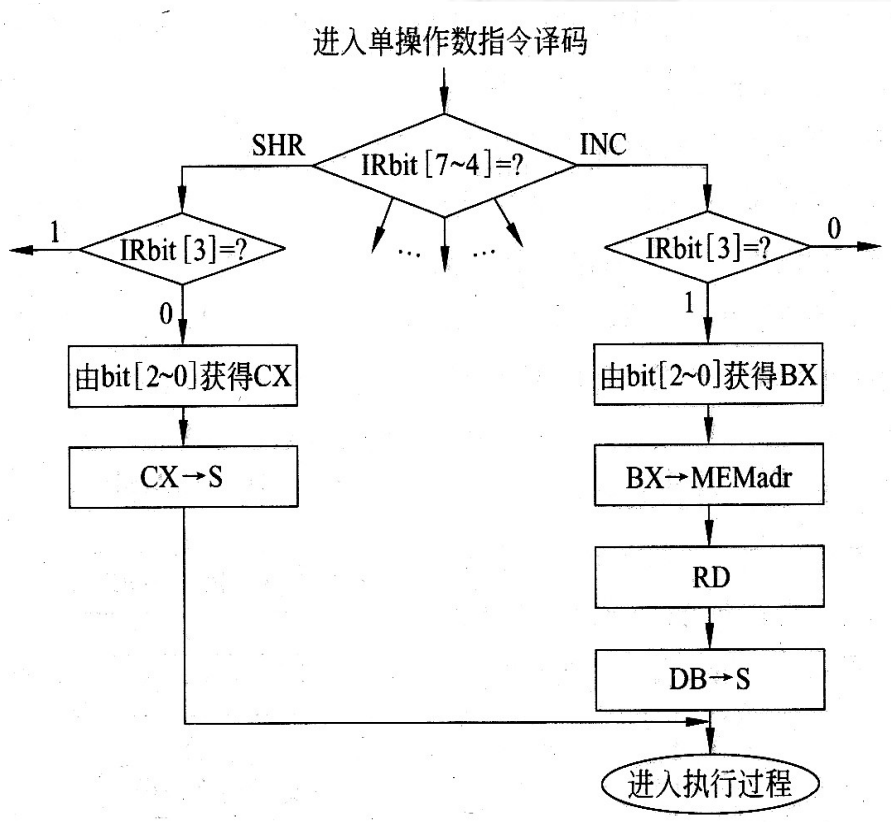
图 5-11　单操作数指令译码过程

Simulating a dummy CPU~

- 单操作数指令译码取操作数过程（续）：



- ADD AX [SI]
  - 01 86 (单字)
- Mov AX [2000H]
  - 01 89 40 00 (双字)
- INC [BX]
  - 0E 15 (单字)



Simulating a dummy CPU~

- 无操作数指令译码取操作数过程：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	1	1	1	1	1	1	1	1	X	X	X	X	无操作数指令
0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	RET
												0	0	0	1	IRET
												0	0	1	0	NOP
												0	0	1	1	CLI
												0	1	0	0	STI
												0	1	0	1	SWI
												0	1	1	0	DAA
												0	1	1	1	
												1	1	1	1	

IRET

0F F1

NOP

0F F2

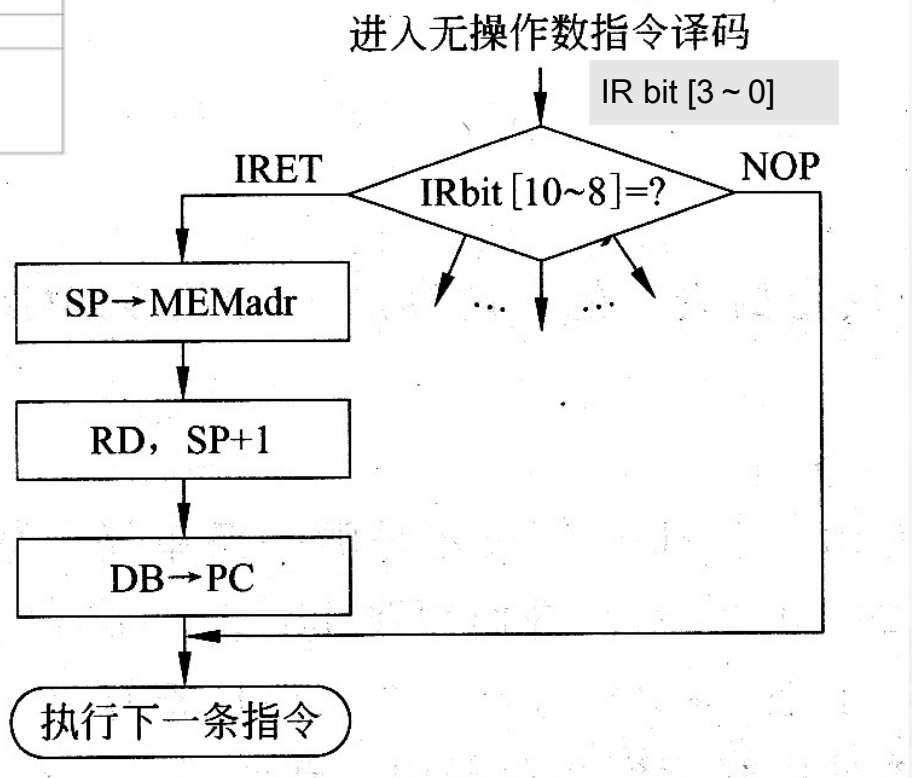
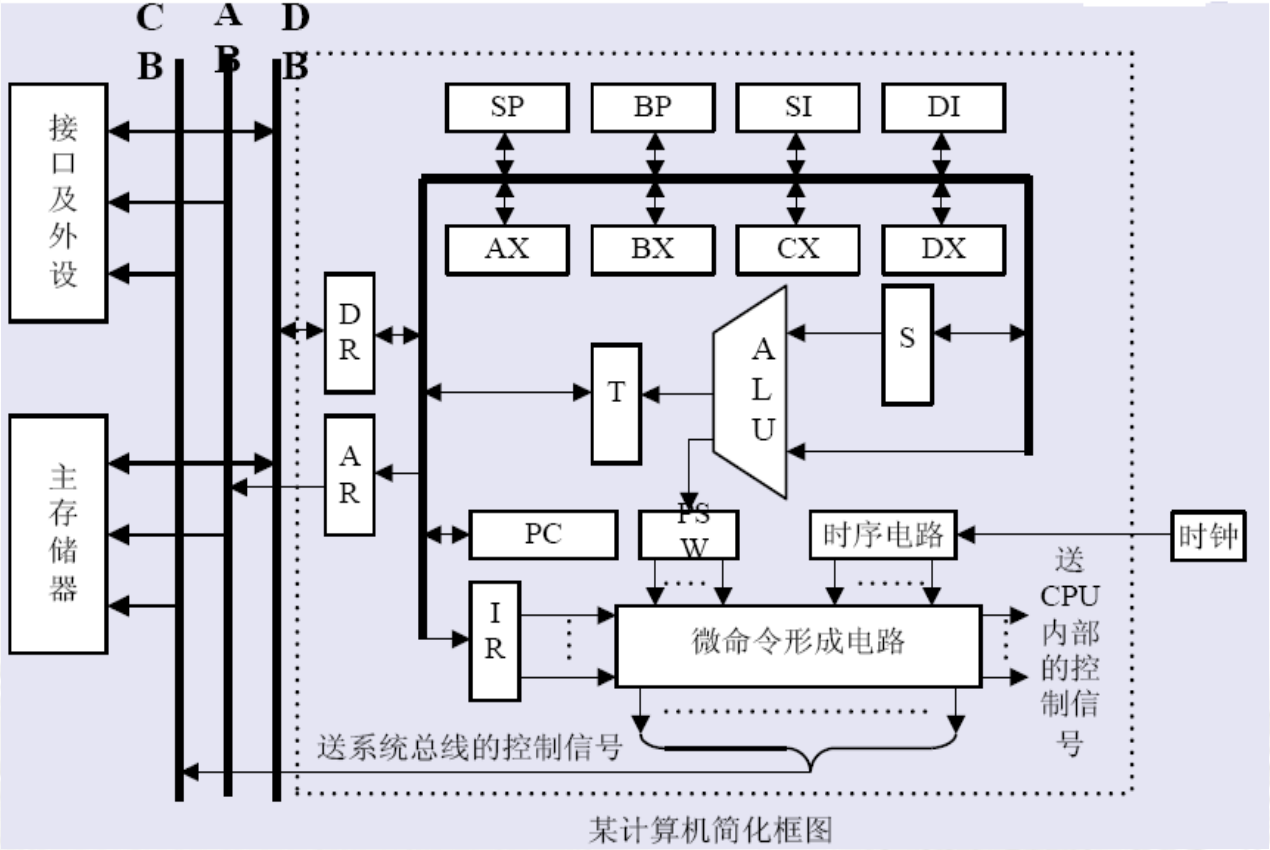


图 5-12　无操作数指令的译码过程

Simulating a dummy CPU~

c) 指令执行过程：

- 计算机简单框图



某计算机简化框图

JMP disp

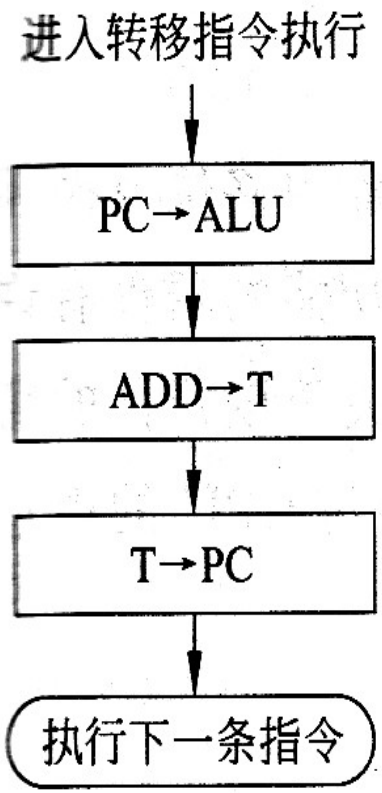


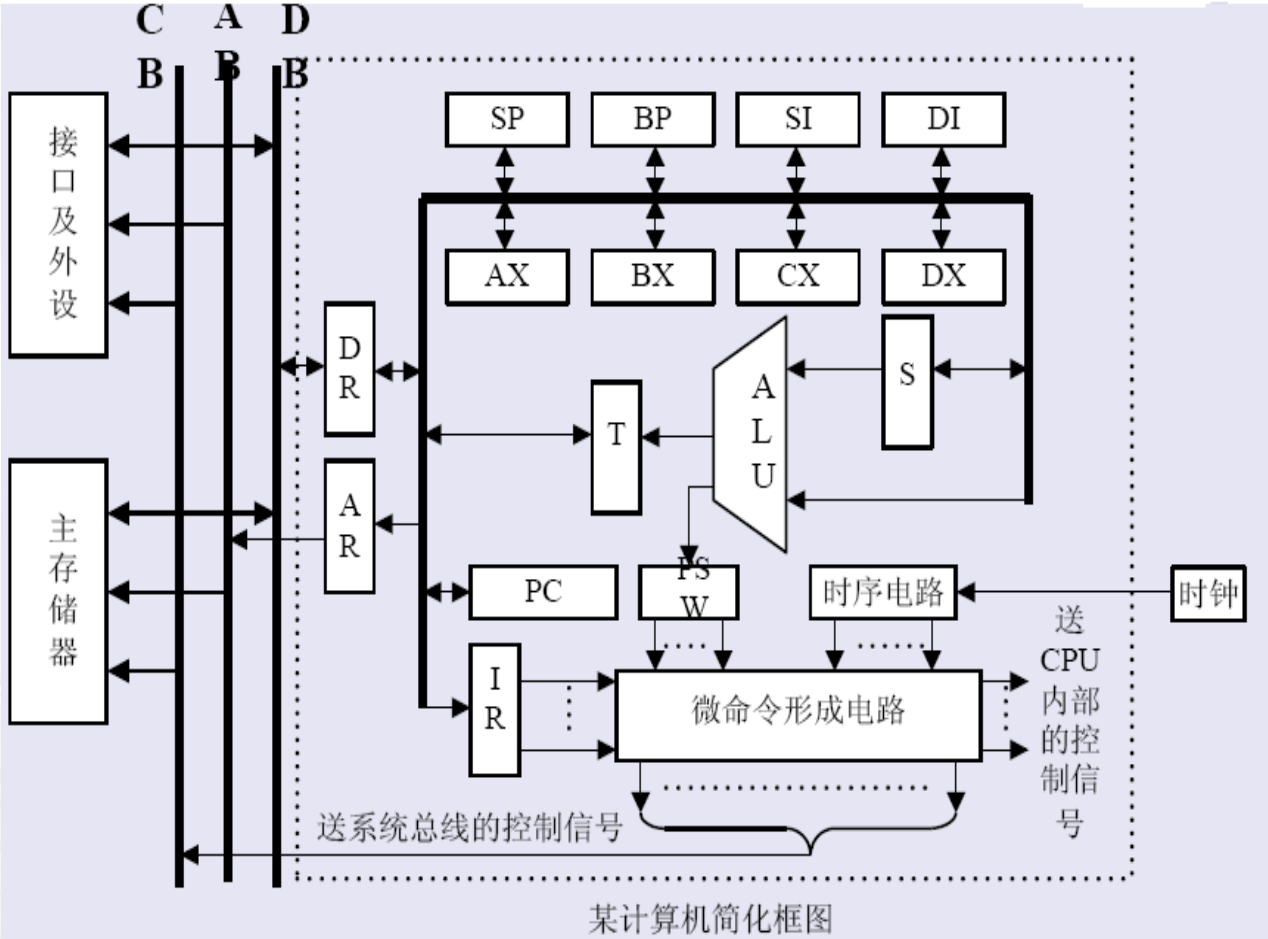
图 5-13 转移指令执行

Simulating a dummy CPU~

c) 指令执行过程(续):

ADD BX, [DI]

- 计算机简单框图



某计算机简化框图

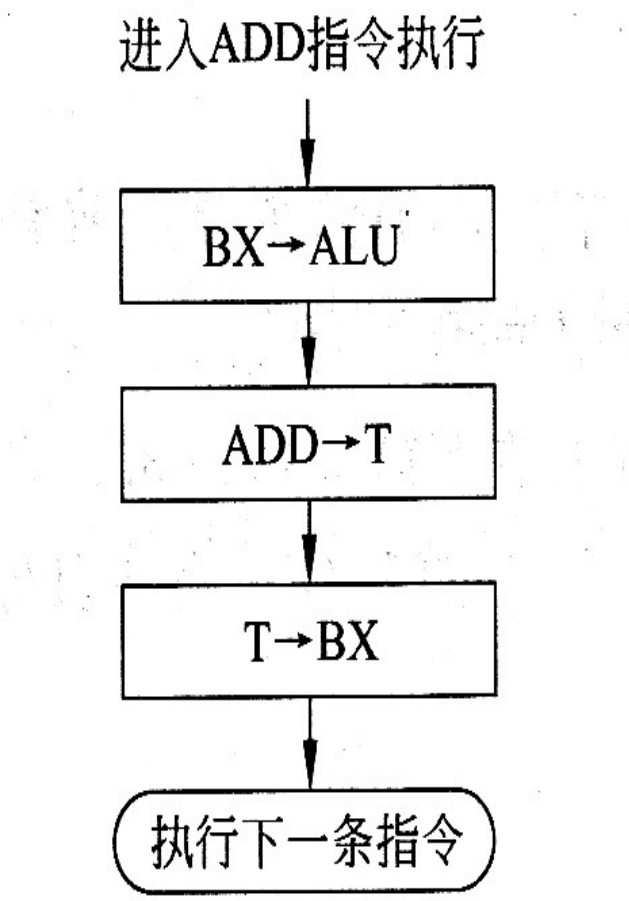
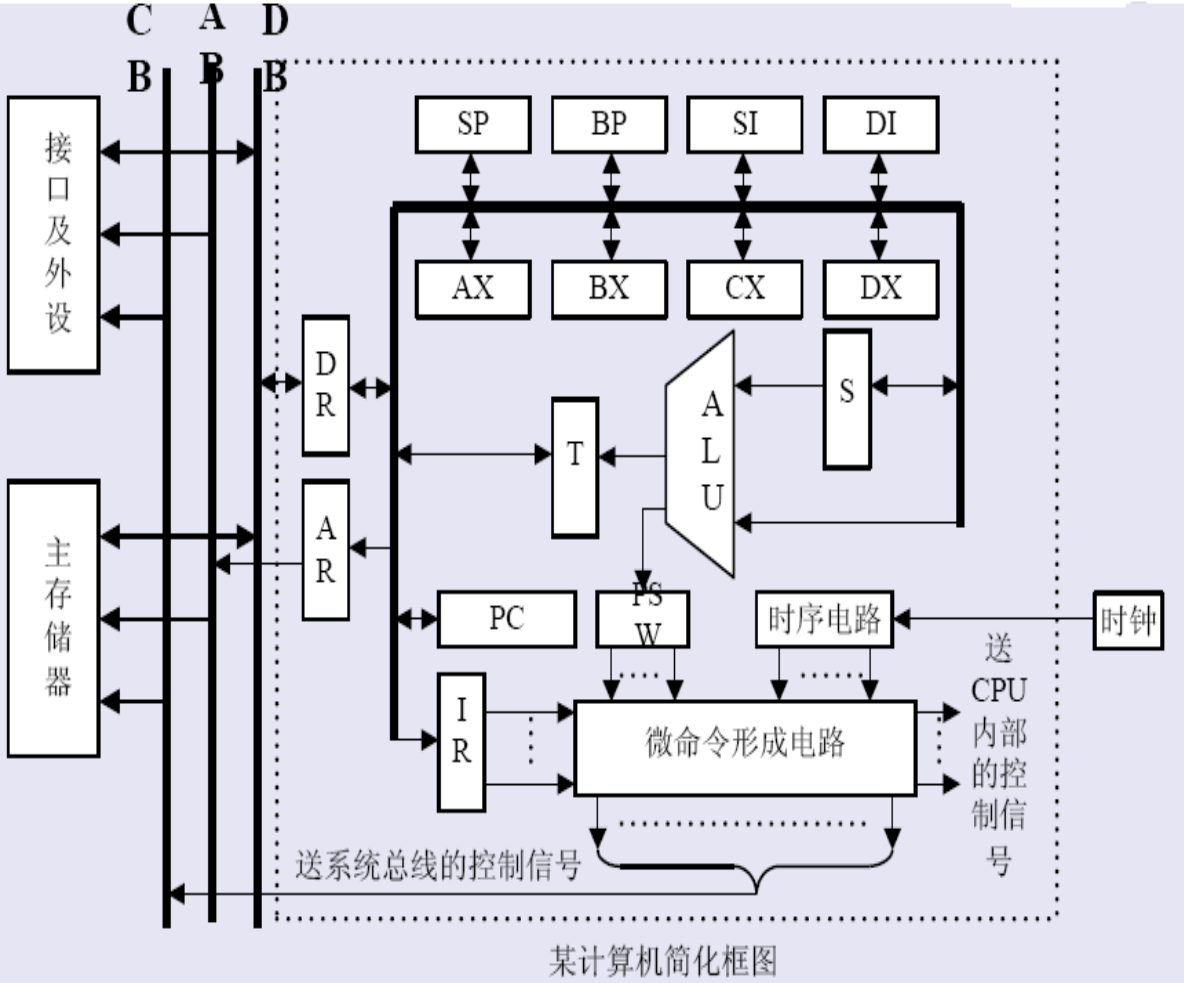


图 5-14 ADD 指令执行

Simulating a dummy CPU~

c) 指令执行过程(续):

- 计算机简单框图



某计算机简化框图

INC [BX]

SHR CX