



西安电子科技大学
人工智能学院

计算机组成与体系结构

第6章 中央处理器 (CPU)

赵庆行、张骏鹏

本章主要介绍**CPU的结构及控制器**的设计方法。

- CPU结构和微操作
- 硬布线控制器设计
- 微程序控制器设计
- CPU性能的测量与提高
- CPU中的新技术
- 典型的CPU

本章第1次课重点

- CPU功能与结构
- 微操作



CPU的功能与结构



6.1.1 CPU功能

- **CPU** (**C**entral **P**rocessing **U**nit) 即中央处理单元，是计算机的控制与处理核心，决定着计算机系统的功能和性能。
- **CPU** 具有以下 4 个方面的基本功能
 - **指令控制**：确保计算机指令按程序的顺序执行。
 - **操作控制**：一条指令的功能通常有若干个操作信号（微操作）组合起来实现，**CPU** 控制这些微操作的产生、组合、传送和管理。
 - **时间控制**：使各种微操作和指令的执行严格按照时间序列进行。
 - **数据加工**：由运算器对数据进行算术运算和逻辑运算。

6.1.1 CPU功能

➤ CPU执行指令的过程

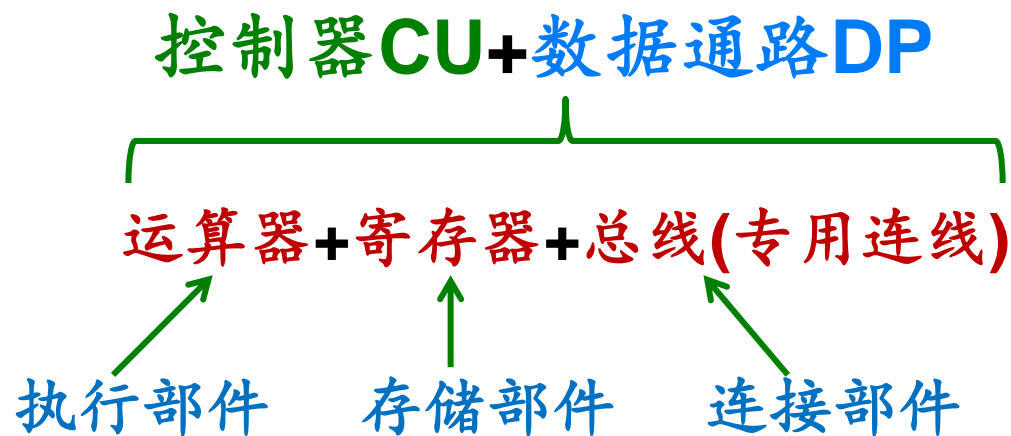
- ✓ 取得指令：CPU从主存中取得指令，并将其暂存在CPU内部的指令寄存器IR中；
- ✓ 译码指令：CPU对取得的指令进行译码；
- ✓ 执行指令：CPU根据指令译码结果执行指令。
- ✓ 确定下条指令执行顺序：除指令顺序被分支跳转类指令改变之外，CPU主要依据指令在主存中的存储顺序执行指令。

➤ CPU执行程序的过程

- ✓ 不断重复执行指令的过程

6.1.1 CPU功能

➤ CPU基本组成



➤ 控制器CU功能

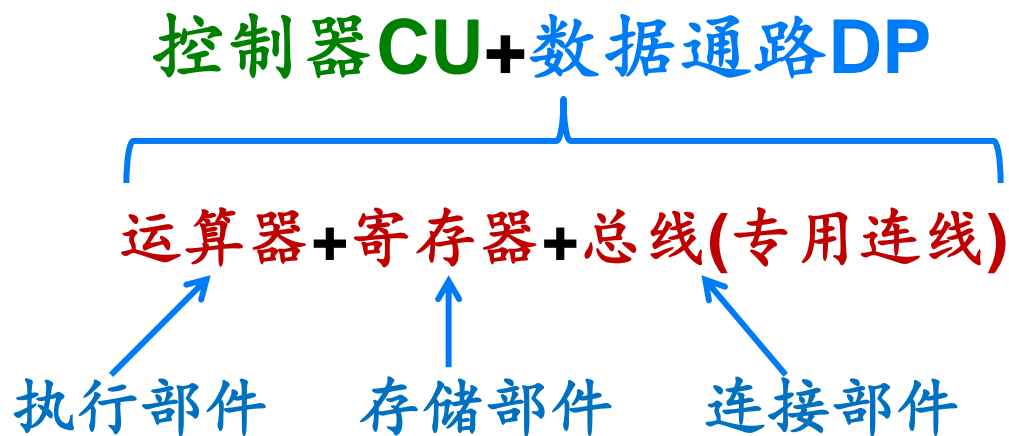
- ✓ 从存储器中取指令、对指令译码、产生控制信号并控制计算机系统各部件有序地执行，从而实现这条指令的功能。

➤ 控制器的组成

- ✓ 指令译码器、控制单元、时序信号产生器

6.1.1 CPU功能

➤ CPU基本组成



➤ 数据通路DP (datapath)

- ✓ 是一个利用内部**专用连线**或**总线**（具有将数据从一个地方移动到另一个地方的能力）将**存储部件**（**寄存器组**、**Cache**）和**算术逻辑部件ALU**（用于对数据完成各种操作）连接在一起的**网络**
- ✓ 是指令执行时数据经过的**路径**与路径上的**部件**的整体

6.1.1 CPU功能

➤ CPU执行一条指令所用的时间即为**指令周期**(instruction cycle)，在**指令周期**内CPU完成一组**操作**。

- ✓ 取指令子周期 (fetch cycle)
 - ✓ 执行指令子周期 (execute cycle)
 - ✓ 中断子周期 (interrupt cycle)
- 取数子周期
执行子周期
存数子周期

取指子周期 取数子周期 执行子周期 中断子周期

**CPU周期或
机器周期**

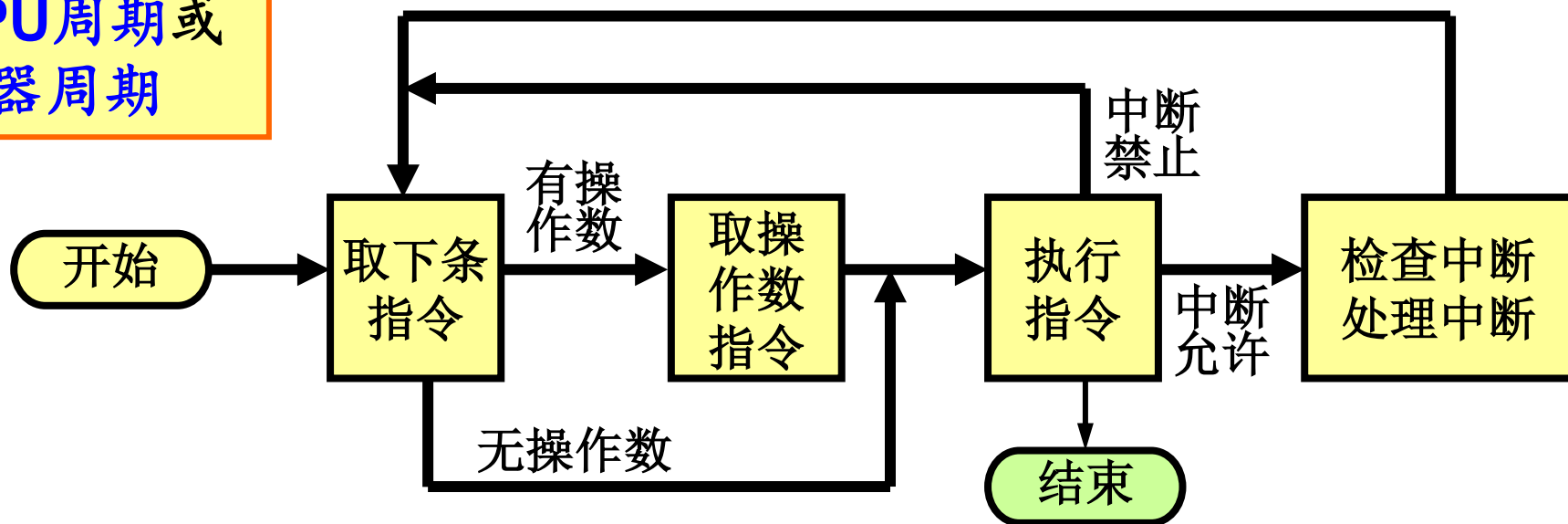


图6.1 指令周期及CPU操作

6.1.2 基础的RISC-V系统结构

➤ 基础RISC-V系统结构: 专用数据通路

- DP 包括 IM, DM, RF, ALU, Adder
- 支持 **ld**, **sd**, **add**, **sub**, **and**, **or**, **beq** 指令

RISC-V指令格式

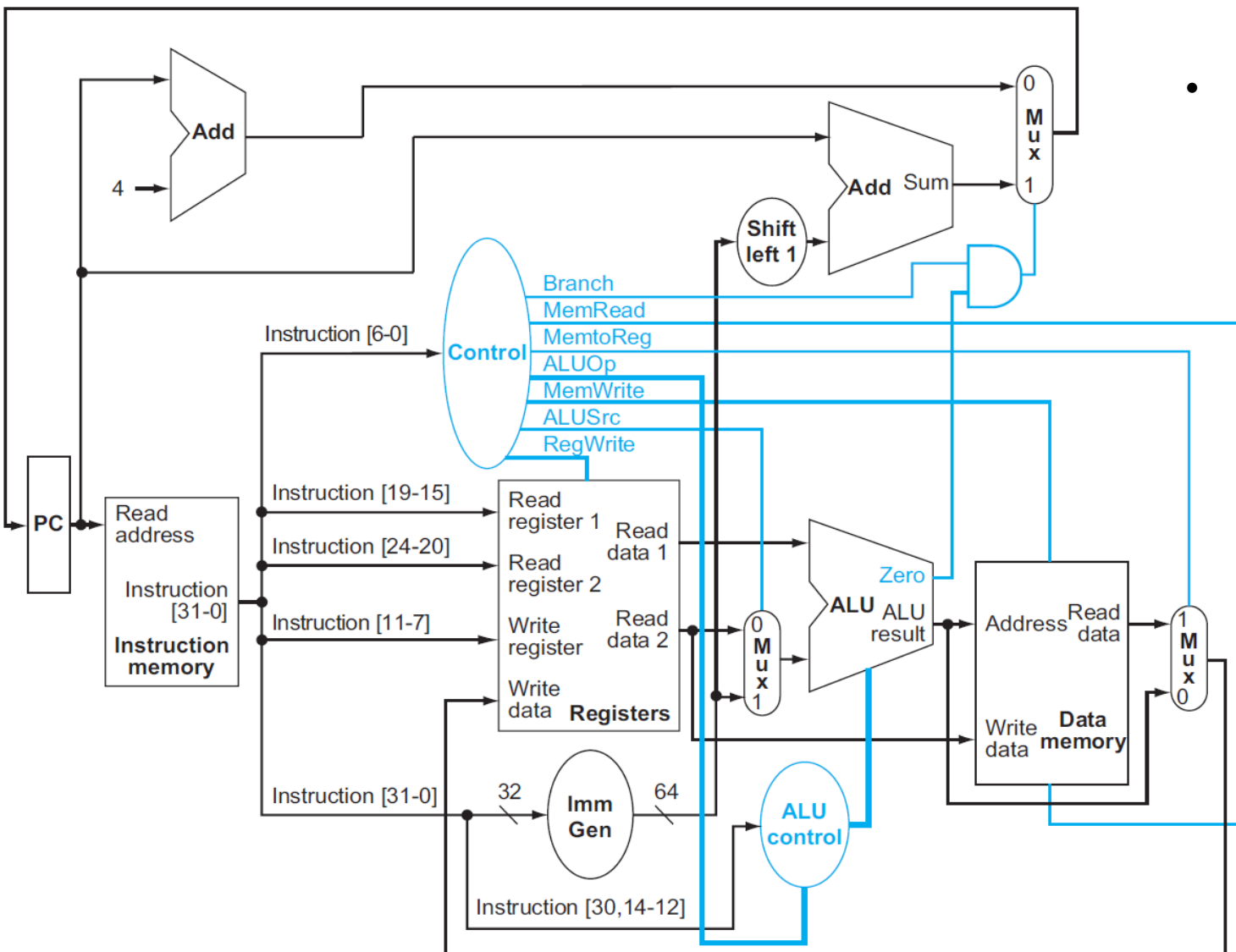


图6.2基础的RISC-V系统结构图

6.1.2 基础的RISC-V系统结构

➤ 回顾：RISC-V指令示例

	31	25	24	20	19	15	14	12	11	7	6	0
R型	funct7		rs2		rs1		funct3		rd		opcode	
I型	imm[11:0]				rs1		funct3		rd		opcode	
S型	imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode	
B型	imm[12,10:5]		rs2		rs1		funct3		imm[4:1,11]		opcode	
U型	imm[31:12]								rd		opcode	
J型	imm[20,10:1,11,19:12]								rd		opcode	



- ✓ **reg:0-31**之间的寄存器编号；**address: 12位地址或常量**；**funct7, funct3**充当附加的操作码字段

6.1.2 基础的RISC-V系统结构

➤ 控制信号及作用

- ✓ 支持访存指令（ld、sd）、算术逻辑指令（add、sub、and、or）和条件分支指令（beq）的实现

表 6.1 控制信号及作用

控制信号	作 用
ALUSrc	有效时，第二个 ALU 操作数是符号扩展的 12 位数据(在指令中)； 无效时，第二个 ALU 操作数来自第二个寄存器组的输出(Read data 2)
MemtoReg	有效时，写入寄存器的数据来自数据存储器； 无效时，写入寄存器的数据来自 ALU
RegWrite	有效时，写数据端输入的数据写入写寄存器端提供的寄存器中
MemRead	有效时，由地址端指定的数据存储单元内容被放在读数据端输出
MemWrite	有效时，由地址端指定的数据存储单元内容被替换为写数据端输入的数据
Branch	有效时，当前指令是分支指令

6.1.2 基础的RISC-V系统结构

【例6.1】根据图6.2，依据信息流次序，说明R型加法指令addx1, x2, x3的执行步骤。

RISC-V系统结构

		31	25	24	20	19	15	14	12	11	7	6	0		
指令	类型	funct7		rs2		rs1		funct3		rd		opcode		示例	
add	R	0000000		reg		reg		000		reg		0110011			
		0000000		00011		00010		000		00001		0110011		add x1,x2,x3	
sub	R	0100000		reg		reg		000		reg		0110011			
		0100000		00011		00010		000		00001		0110011		sub x1,x2,x3	

- ① IM[PC]→指令, (PC)+4→PC
- ② 读x2和x3, 控制单元→控制信号
- ③ 控制信号控制ALU: (x2)+(x3)
- ④ ALU结果→x1

RISC-V系统结构

		31	25	24	20	19	15	14	12	11	7	6	0		
指令	类型	immediate			rs1		funct3		rd		opcode		示例		
addi	I	constant			reg		000		reg		0010011				
		001111101000			00010		000		00001		0010011		addi x1,x2,1000		
ld	I	address			reg		011		reg		0000011				
		001111101000			00010		011		00001		0000011		ld x1, 1000(x2)		

- ① $IM[PC] \rightarrow \text{指令}, (PC)+4 \rightarrow PC$
- ② 读x2
- ③ 控制信号控制ALU: $(x2) + sext(offset) \rightarrow DM \text{地址}$
- ④ DM根据地址读取数据
- ⑤ 将数据写入x1

6.1.2 基础的RISC-V系统结构

【例6.3】根据图6.2，依据信息流次序，说明 B型相等分支指令 **beq x1, x2, offset** 的执行步骤。

RISC-V系统结构

		31	25	24	20	19	15	14	12	11	7	6	0		
指令	类型	immediate			rs2	rs1		funct3		immediate			opcode	示例	
beq	B	address[12,10:5]			reg	reg		000		address[4:1,11]			1100011		
		0, 111110			00010	00001		000		1000, 0			1100011	beq x1, x2, 2000	

- ① $IM[PC] \rightarrow$ 指令, $(PC)+4 \rightarrow PC$
- ② 读x1和x2
- ③ 控制信号控制ALU: $(x1)-(x2)$; 同时 $(PC)+sxt(offset)$
- ④ 根据ALU结果零状态位决定PC写入+4/+offset结果

6.1.3 简化的x86系统结构

ALU 仅依赖 CPU 内部寄存器中的数据工作进行，所以 CPU 内部总线用来在各寄存器与 ALU 之间传递数据，且某一时刻只传递一个数据。

- (1) 算术逻辑单元 ALU
- (2) 控制单元 CU
- (3) 状态寄存器 PSW
- (4) 通用寄存器组 R0~Rn
- (5) 堆栈指针寄存器 SP
- (6) 数据寄存器 DR
- (7) 地址寄存器 AR
- (8) 程序计数器 PC
- (9) 指令寄存器 IR
- (10) 暂存器 Y
- (11) 暂存器 Z

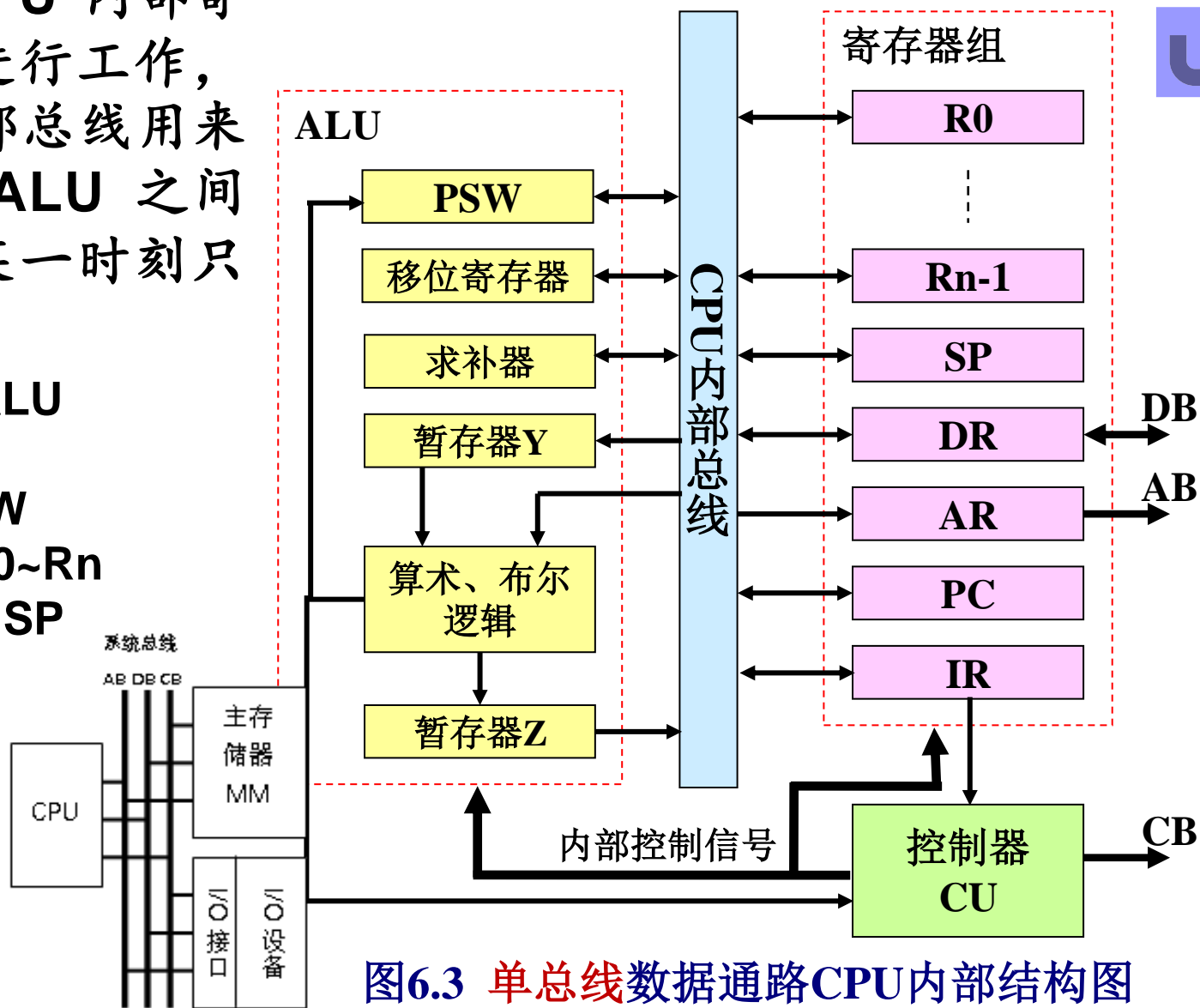


图6.3 单总线数据通路CPU内部结构图



指令执行的微操作



- 在 **CISC** 系统中，指令周期内的 **CPU** 行为常常被分解为一系列微操作(μop)。
- 程序执行分解：指令周期、**CPU**周期、微操作

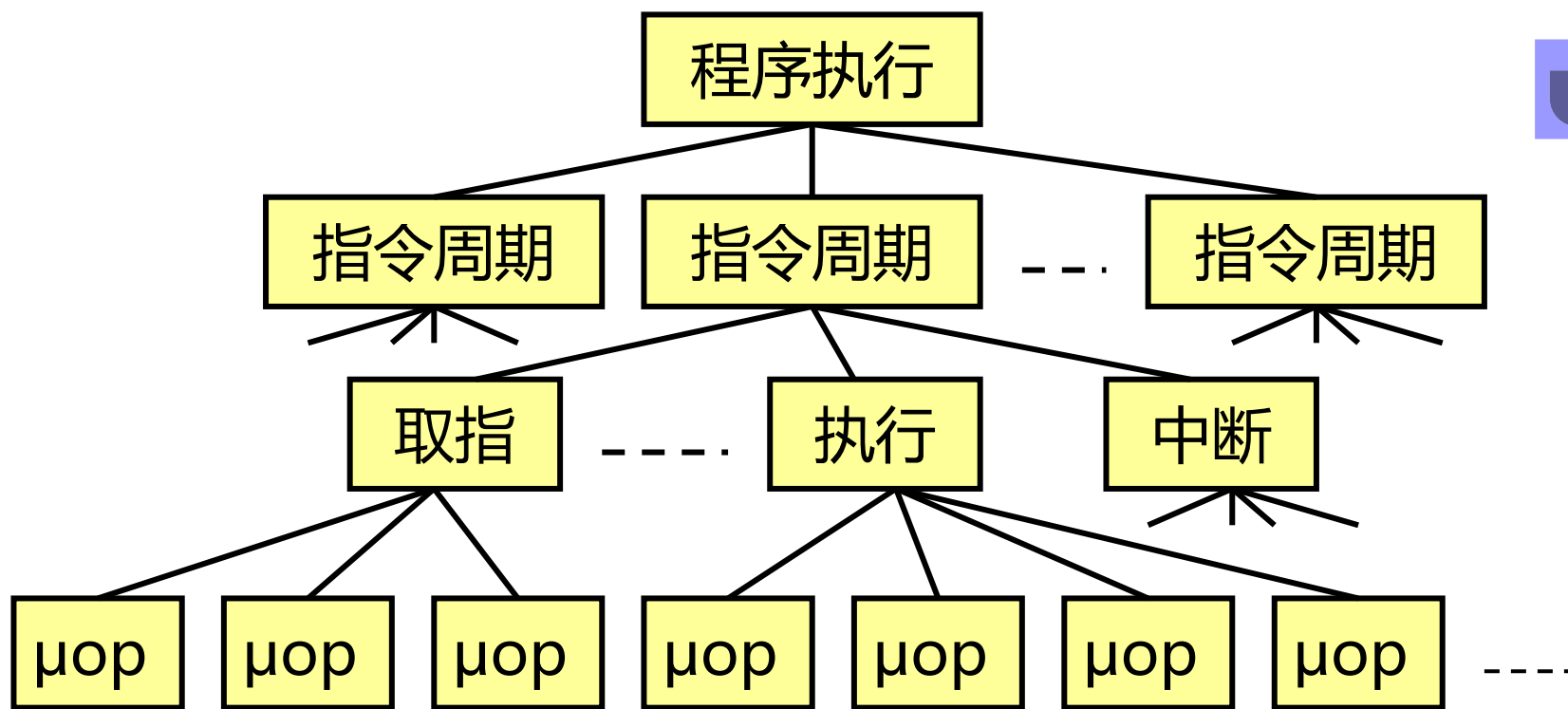


图6.5 程序执行示意图

6.1.4 微操作

一、微操作与微命令

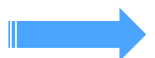
- **微操作**：处理器（CPU）的基本或原子操作。
 - ✓ CPU可以实现的、**不可分解**的操作动作
 - ✓ 以含有一个**寄存器传递**（移进、移出）操作为标志
- 每一个**微操作**是通过**控制器**将**控制信号**发送到相关部件上引起**部件动作**而完成的。
 - ✓ 这些控制微操作完成的**控制信号**称为**微命令**
 - ✓ **微命令**是由**控制器**产生的

$AR \leftarrow PC$; PC_{out}, AR_{in}

图6.3

微操作

微命令

- 执行一条指令  **有序执行一组**微操作


微操作流程(微操作序列)

微操作流程

- CPU执行微操作有严格的时间顺序性，所以CPU执行指令(微操作序列)需要三种时序信号：
 - 节拍周期：完成一个微操作所用的时间
 - CPU周期：完成一个子周期所用的时间
 - 指令周期：执行一条指令所用的时间
- 通常利用时序电路为控制器提供所需的时序信号。

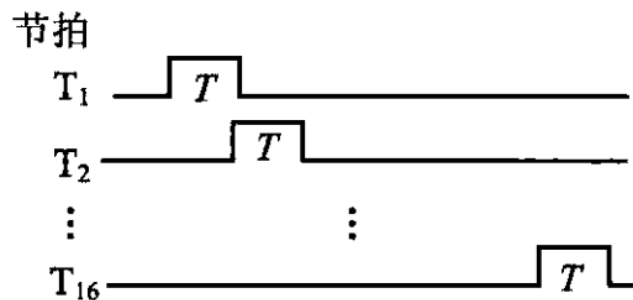
(1) 节拍周期

- 最基本的时序信号为节拍，它可由顺序脉冲发生器也称脉冲分配器或节拍脉冲发生器产生。
- 节拍周期 T ：完成各种CPU微操作所需时间的最大者，常作为定义CPU时钟周期 T_{clock} 或时钟频率 f_{clock} 的依据。

$$T = (1 \sim n) \times T_{clk} = (1 \sim n) / f_{clk}$$

- 节拍脉冲发生器分两类：

- 计数型=计数器+译码器
- 移位型=移位寄存器+译码器



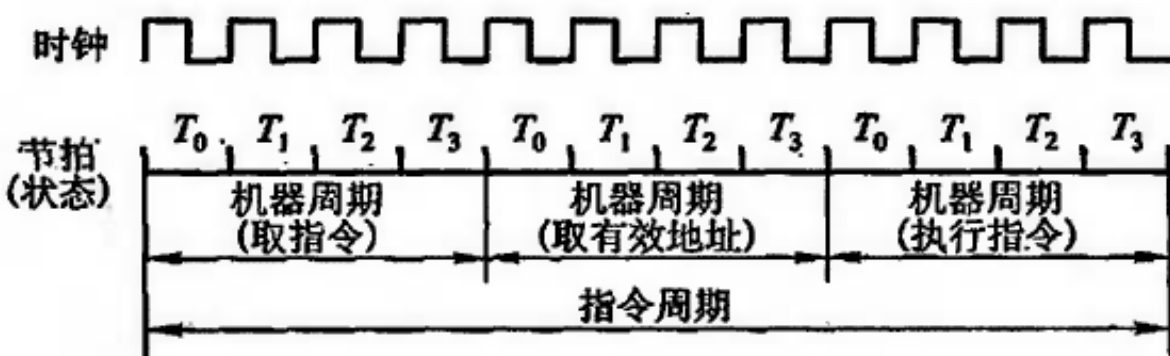
(2) CPU周期

- 完成一个**子周期**所用的时间。一般指**CPU**与内存交换一次信息所需要的时间。若干个节拍组成一个**CPU周期**。
- **CPU周期**可以设计为**定长 CPU周期**与**不定长CPU周期**两种。

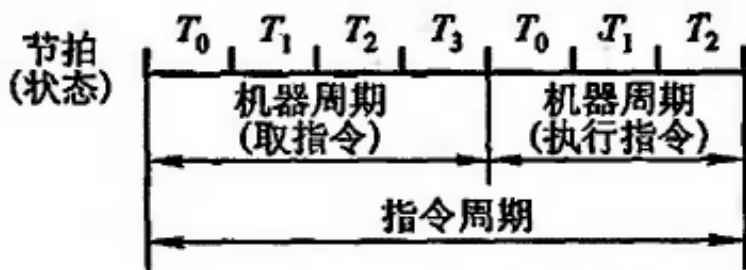
(3) 指令周期

- 执行一条指令所用的时间。**若干个CPU周期**组成一个指令周期，指令周期也可以设计为**定长指令周期**与**不定长指令周期**两种。

➤ 指令周期、CPU周期、节拍和时钟周期关系



(a) 定长的机器周期



(b) 不定长的机器周期

➤ 一个指令周期可以包含多个机器周期，每个指令周期内的机器周期数可能相等也可能不等

➤ 一个机器周期内也可以包含多个时钟周期(节拍)，每个机器周期内的时钟周期数可能相等也可能不等

- 当一条指令在该系统上执行时，可以被看作是一组微操作的执行。
- 每条指令对应的一组微操作，称为该指令的微操作流程或微操作序列。

图6.5

一条指令 \longleftrightarrow 一个完整微操作序列



- 以下分析或设计的微操作序列是基于图6.3和图6.4的硬件结构。

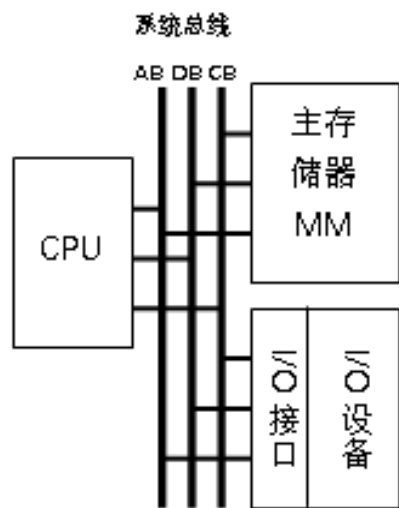


图 6.4 早期 x86 系统模型

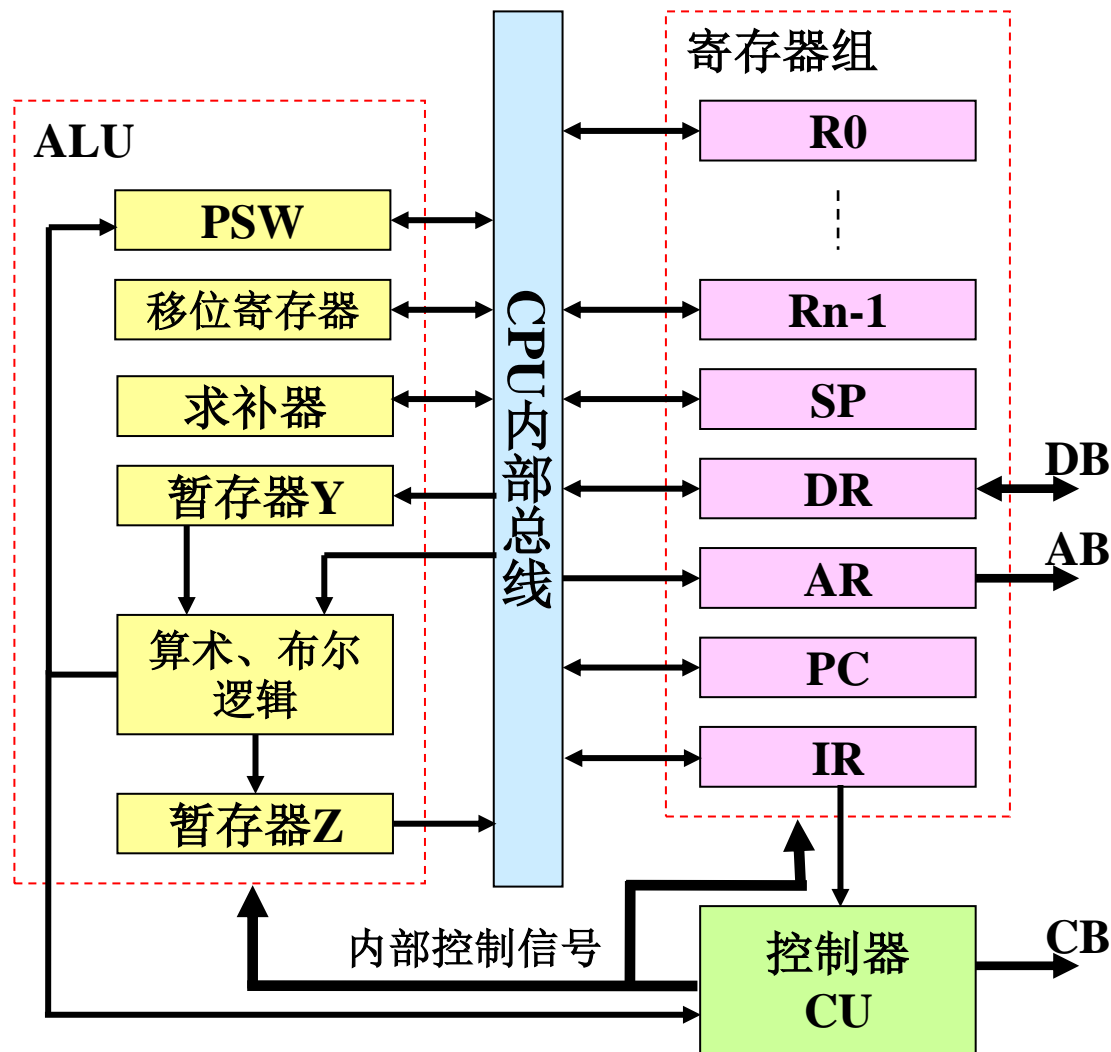


图6.3 单总线数据通路CPU内部结构图

图6.3

1. 取指周期

- 一个简单的取指周期可由3个节拍、4个微操作组成：

T1: $AR \leftarrow PC$;PC的内容传送到AR

T2: $DR \leftarrow \text{Memory}[AR], \text{Mread}$
;由AR规定的存储单元的内容(当前指令)传送到DR

$PC \leftarrow PC + I$

;PC内容加I形成下条指令地址, I为指令长度

T3: $IR \leftarrow DR$;DR的内容传送到IR

- 组合微操作：

T1: $AR \leftarrow PC$

T2: $DR \leftarrow \text{Memory}[AR], \text{Mread}$

T3: $PC \leftarrow PC + I$

$IR \leftarrow DR$

组合微操作的规则：

- 遵守操作发生的顺序
- 必须避免冲突

- 所有指令的取指操作相同，故取指周期被称作公操作。

2. 中断周期

- 在执行周期结束时有一个检测，用来确定被允许的中断是否已出现，若是，中断周期产生。
- 中断周期也是公操作。
- 中断周期微操作序列举例：

T1: DR←PC

;PC的内容传送到DR加以保护，以便实现从中断返回

T2: AR←Save_Address

;中断断点信息保护区的存储单元地址传送到AR

PC←Routine_Address ;中断服务程序首地址送入PC

T3: Memory[AR]←DR, Mwrite

;将老PC的内容保存于主存（如堆栈）中

3. 执行周期

(1) MOV R1, R0

实现将寄存器R0的内容传送至寄存器R1中。

执行周期的微操作序列:

T1: R1 ← R0 ;将R0中的数据传送到R1

(2) MOV R0, X

实现将存储单元X中的内容传送至寄存器R0中。

执行周期的微操作序列为:

T1: AR ← IR (地址字段)

;将指令中的存储器地址X传送到AR, IR (地址字段) = X

T2: DR ← Memory[AR], Mread

;从存储单元X中读出的数据传送到DR

T3: R0 ← DR ;DR的内容传送到R0

3. 执行周期

(3) MOV (R1), R0

将寄存器R0的内容传送至由寄存器R1间接寻址的存储单元中。

执行周期的微操作序列:

T1: AR ← R1 ;将R1中的存储单元地址传送到AR

T2: DR ← R0 ;R0中的数据传送到DR

T3: Memory[AR] ← DR, Mwrite
;将DR的内容写入指定的存储单元中

3. 执行周期

(4) ADD R1, R0

将寄存器R0的内容与寄存器R1的内容相加并将结果存入R1。

执行周期的微操作序列:

T1: $Y \leftarrow R0$;将R0中的数据传送到暂存器Y中

T2: $Z \leftarrow R1 + Y$

;R1中数据与Y中数据加载至ALU做加法, 结果暂存于Z中

T3: $R1 \leftarrow Z$;将暂存器Z的内容传送到R1中

(5) SUB R0, (X)

实现寄存器R0中的被减数减去存储器地址X间接寻址的存储单元中的减数、将差值传送至寄存器R0中。

执行周期的微操作序列:

图6.3

T1: $AR \leftarrow IR$ (地址字段)

;将指令中的存储器地址X传送到AR, IR (地址字段) = X

T2: $DR \leftarrow Memory[AR]$, Mread

;减数所在存储单元的地址传送到DR

T3: $AR \leftarrow DR$;DR的内容传送到AR

T4: $DR \leftarrow Memory[AR]$, Mread

;再次访问存储单元, 读出的减数传送到DR

T5: $Y \leftarrow R0$

;将R0中的被减数传送到暂存器Y, 假设ALU规定被减数在Y中

T6: $Z \leftarrow Y - DR$

;Y中被减数和DR中减数加载至ALU做减法, 结果暂存于Z

T7: $R0 \leftarrow Z$;将暂存器Z的内容传送到R0中

3. 执行周期

(6) IN R0, P

从I/O地址为P的I/O设备(接口)中输入数据并存入寄存器R0中。

执行周期的微操作序列:

T1: AR ← IR(地址字段)

;将指令中的I/O地址P传送到AR, IR(地址字段) = P

T2: DR ← IO[AR], IOread

;从I/O设备(接口)中输入的数据传送到DR

T3: R0 ← DR

;DR的内容传送到R0

3. 执行周期

(7) OUT P, R0

将寄存器R0中的数据输出到I/O地址为P的I/O设备(接口)中。

执行周期的微操作序列:

T1: $AR \leftarrow IR$ (地址字段)

;将指令中的I/O地址P传送到AR, IR (地址字段) = P

T2: $DR \leftarrow R0$;R0的内容传送到DR

T3: $IO[AR] \leftarrow DR$, IOwrite

;将DR的内容输出至指定的I/O设备(接口)中

3. 执行周期

(8) JUMP X

无条件跳转指令，实现将程序执行地址从当前跳转指令所在位置转移到存储器地址为X处。

执行周期的微操作序列：

T1: $PC \leftarrow IR(\text{地址字段})$

;将指令中的存储器地址X传送到PC, $IR(\text{地址字段}) = X$

3. 执行周期

(9) JZ offs

采用相对寻址的条件跳转指令。当条件为真（即零标志 $ZF=1$ ）时，程序发生跳转；条件为假（即零标志 $ZF=0$ ）时，程序顺序执行下条指令。跳转地址= $PC+offs$ ， $offs$ 为带符号地址偏移量。与该指令相应的执行周期的微操作序列为：

If ($ZF=1$) then

{

T1: $Y \leftarrow IR(\text{地址字段})$

；将指令中偏移地址 $offs$ 送入暂存器 Y ， $IR(\text{地址字段}) = offs$

T2: $Z \leftarrow PC + Y$

； PC 中当前地址与 Y 中偏移地址加载至 ALU ，相加，结果暂存于 Z

T3: $PC \leftarrow Z$ ；将暂存器 Z 中的跳转地址传送到 PC 中

}

3. 执行周期

(10) PUSH R0

实现将寄存器R0中的数据压入到堆栈中。

执行周期的微操作序列：

T1: $SP \leftarrow SP - n$;将SP指向新栈顶, n为一次压栈的字节数

$DR \leftarrow R0$

T2: $AR \leftarrow SP$

T3: Memory $[AR] \leftarrow DR$, Mwrite

;将R0的内容写入堆栈新栈顶处

3. 执行周期

(11) POP R0

实现将堆栈栈顶的数据弹出至寄存器R0中。

执行周期的微操作序列：

T1: $AR \leftarrow SP$

T2: $DR \leftarrow \text{Memory}[AR], \text{Mread}$

T3: $R0 \leftarrow DR$;堆栈栈顶处的内容传送到R0

$SP \leftarrow SP + n$;将SP指向新栈顶，n为一次弹出的字节数

3. 执行周期

(12) CALL (X)

子程序调用指令。将程序执行地址从当前调用指令所在位置转移到以**存储器地址X间接寻址**的存储单元处，并保存返回地址。

执行周期的**微操作序列**：

T1: $SP \leftarrow SP - n$;将**SP**指向新栈顶，**n**为**PC**的字节数

$DR \leftarrow PC$

T2: $AR \leftarrow SP$

T3: $Memory[AR] \leftarrow DR$, $Mwrite$

;将**PC**中的返回地址保存在堆栈新栈顶处

T4: $AR \leftarrow IR(\text{地址字段})$

;将指令中的存储器地址**X**传送到**AR**, $IR(\text{地址字段}) = X$

T5: $DR \leftarrow Memory[AR]$, $Mread$

T6: $PC \leftarrow DR$;从存储单元**X**中读出的子程序首地址传送到**PC**

3. 执行周期

(13) RET

子程序返回指令，实现从堆栈栈顶处获得子程序调用时保存的返回主程序的地址。

与该指令相应的执行周期的微操作序列为：

T1: $AR \leftarrow SP$

T2: $DR \leftarrow \text{Memory}[AR], \text{Mread}$

T3: $PC \leftarrow DR$;堆栈栈顶处的返回地址送入PC

$SP \leftarrow SP + n$;将SP指向新栈顶，n为PC的字节数