



CPU/内存基本工作原理

来源：北京大学计算概论 B 教学小组

修改：甘锐



本讲内容

- 计算机的数学理论模型 – 图灵机
- CPU的内部结构和工作原理
- 指令系统
- 主存储器及其与CPU之间的信息传输



计算机的数学理论模型－图灵机



抽象的“计算”概念

- “计算”是一种过程
 - 从一些初始符号开始，通过操作得到一些符号结果
 - 只包含有穷种不同的基本操作 (指令)
 - 每条指令的规则清晰明确，可以机械地精确执行
 - 指令的每次执行总能得到正确、唯一的结果
 - 例如，十进制加法规则、珠算口诀等
 - 操作过程经过有限步骤后终止
 - 原则上可以由人单独采用纸笔完成、而不依靠其它辅助



可计算性

对于一个问题，

如果存在一个机械的过程，

当给定一个输入，这个过程能够在有限步内终止并给出正确答案，

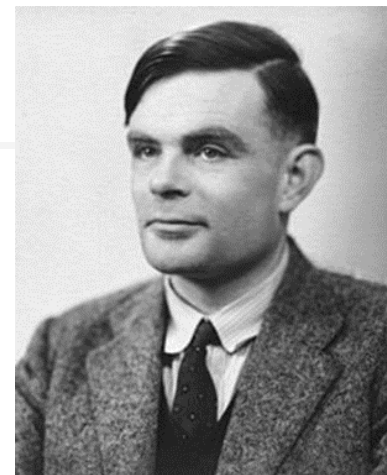
那么，这个问题就称为是(能行 feasible) 可计算的/具有可计算性。



计算机理论的发展历史

- 图灵 研究了 可计算性
 - 提出了 图灵机 和 图灵机能解决的问题类
 - 证明了 存在着图灵机无法解决的问题类
- 冯·诺伊曼 给出了 现代计算机的设计蓝图
 - 提出了 数字计算机的组成原理和体系结构
 - 对指令、指令周期、指令系统和存储式程序控制原理都给出了明确的方案
- 库克 (Stephen A. Cook) 研究了 计算复杂性
 - 有一些问题，虽然可计算，但随着问题规模的增加，就连最快的计算机用几百年也不能结束计算

图灵机 (Turing Machine)



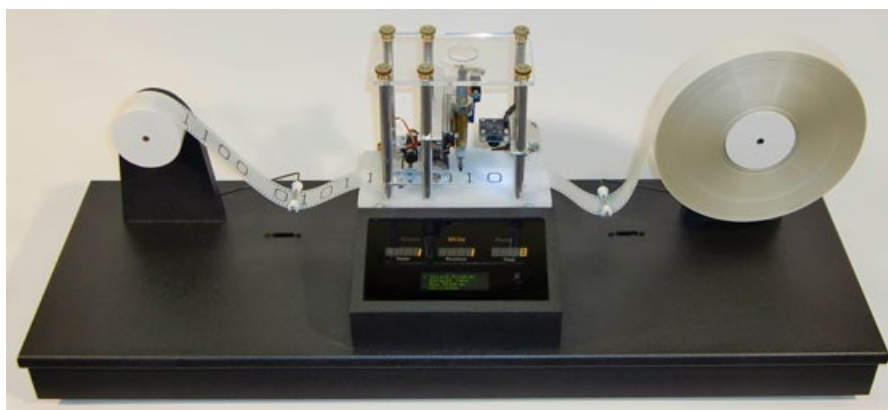
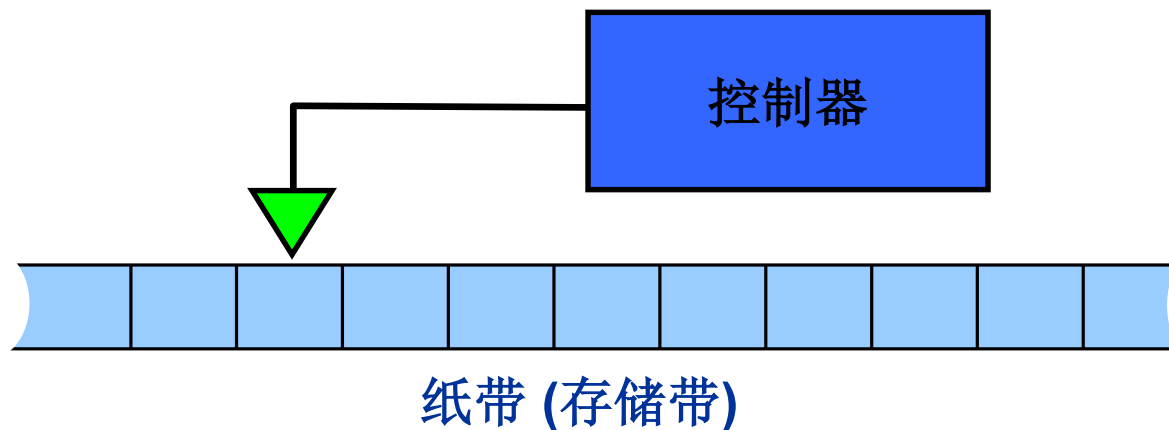
- 1936年由 英国数学家 阿兰·图灵 提出
 - 一种抽象的计算模型
 - 现代电子计算机的理论基础
- 简单理解：用机器来模拟人类用纸和笔进行数学运算的过程
 - 人用纸和笔进行数学运算的两种简单动作：
 - 在纸上写下或擦除某个符号
 - 把注意力从纸的一个位置移动到另一个位置
 - 同时，人的下一步动作依赖于两个因素：
 - 此人当前所关注的纸上某个位置的符号
 - 此人当前的思维状态

图灵机 (Turing Machine)

- 组成

- 纸带
- 读写头
- 控制器

- 控制规则表



A 'real' turing machine

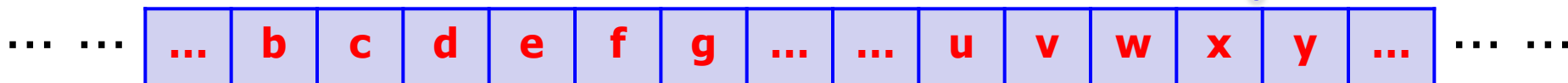
图灵机的构成成分 (1)

1. 一条无限长的纸带 TAPE

- 纸带被划分为一个接一个小方格
- 每个方格存储一个来自一个有限符号集合的符号
- 纸带的两端可以无限延伸

图灵机的
符号表

TAPE



图灵机的构成成分 (2)

2. 一个读写头 HEAD

- 能 读出当前位置的方格里的符号
- 能 在当前位置的方格里写入一个符号
- 能 左右移动
 - 一次移动一个方格的宽度



图灵机的构成成分 (3)

3. 一个控制器 CONTROL

- 一个状态寄存器 REG

- 记录了图灵机的当前状态

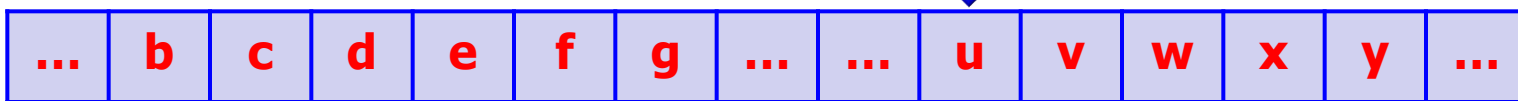
- 一个图灵机具有有限数量的可能状态

有限状态控制器

- 一个控制规则表 TABLE

- 有限个控制规则，规定了图灵机如何在不同的状态之间进行迁移/转换

CONTROL



图灵机的运作方式

- 图灵机的每一步动作取决于四个因素
 - 控制器中的当前状态 q_i
 - 读写头的当前位置（在哪个方格上）
 - 当前位置的方格内存储的符号 s_i
 - 控制规则表中的规则

控制器 根据 q_i 、 s_i 、以及控制规则，

决定：1. 向当前方格内写入的符号

2. 读写头的移动方向（左移, 右移, 不动）

3. 控制器新的当前状态（START, \dots , HALT）

起始
状态

停机
状态

控制规则表的结构

当前状态	当前方格中的符号	写入方格的符号	读写头移动方向	新状态
START
...
q_i	s_i	s_{i+1}	左移	q_{i+1}
...
...	HALT

每一行是个五元组，对应一条控制规则

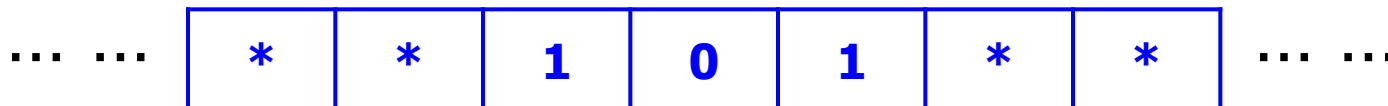
图灵机—实例1

请观察这个图灵机的功能是什么

- 符号表：{0, 1, *}
 - 状态集合：
 - {START/开始, ADD/相加, CARRY/进位, OVERFLOW/溢出, RETURN/返回, HALT/停机}
- 输入：纸带上的初始符号序列
- 结果：停机后纸带上的符号序列

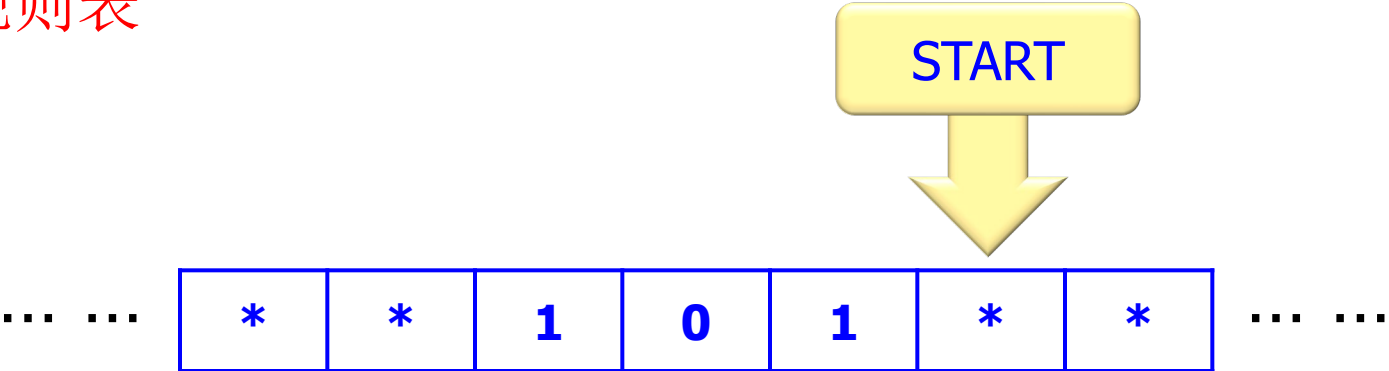
纸带初始布局 and 图灵机初始状态

START



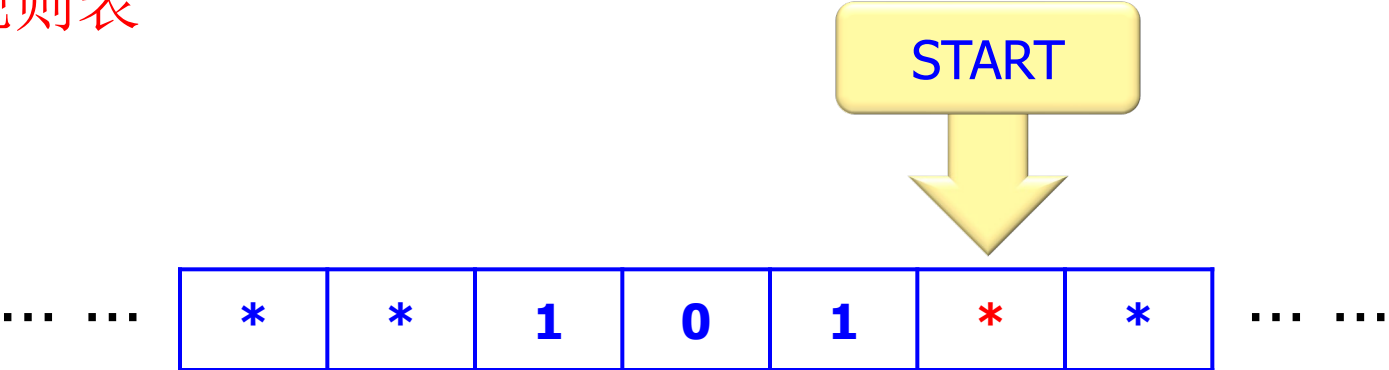
ID	当前状态	当前符号	写入符号	移动方向	新的状态
01	START	*	*	←左移	ADD
02	ADD	0	1	→右移	RETURN
03	ADD	1	0	←左移	CARRY
04	ADD	*	*	→右移	HALT
05	CARRY	0	1	→右移	RETURN
06	CARRY	1	0	←左移	CARRY
07	CARRY	*	1	←左移	OVERFLOW
08	OVERFLOW	*	*	→右移	RETURN
09	RETURN	0	0	→右移	RETURN
10	RETURN	1	1	→右移	RETURN
11	RETURN	*	*	↓不动	HALT

控制规则表



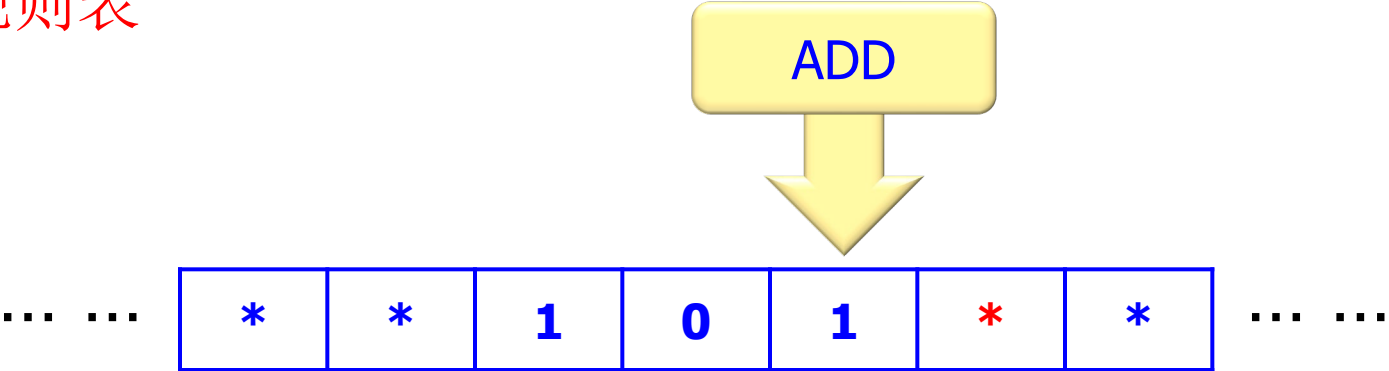
ID	当前状态	当前符号	写入符号	移动方向	新的状态
01	START	*	*	←左移	ADD
02	ADD	0	1	→右移	RETURN
03	ADD	1	0	←左移	CARRY
04	ADD	*	*	→右移	HALT
05	CARRY	0	1	→右移	RETURN
06	CARRY	1	0	←左移	CARRY
07	CARRY	*	1	←左移	OVERFLOW
08	OVERFLOW	*	*	→右移	RETURN
09	RETURN	0	0	→右移	RETURN
10	RETURN	1	1	→右移	RETURN
11	RETURN	*	*	↓不动	HALT

控制规则表



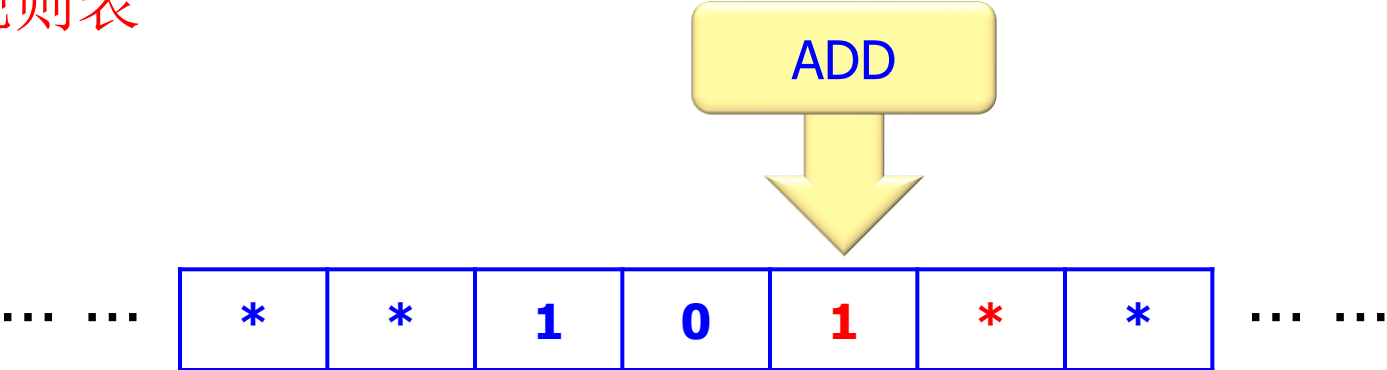
ID	当前状态	当前符号	写入符号	移动方向	新的状态
01	START	*	*	←左移	ADD
02	ADD	0	1	→右移	RETURN
03	ADD	1	0	←左移	CARRY
04	ADD	*	*	→右移	HALT
05	CARRY	0	1	→右移	RETURN
06	CARRY	1	0	←左移	CARRY
07	CARRY	*	1	←左移	OVERFLOW
08	OVERFLOW	*	*	→右移	RETURN
09	RETURN	0	0	→右移	RETURN
10	RETURN	1	1	→右移	RETURN
11	RETURN	*	*	↓不动	HALT

控制规则表



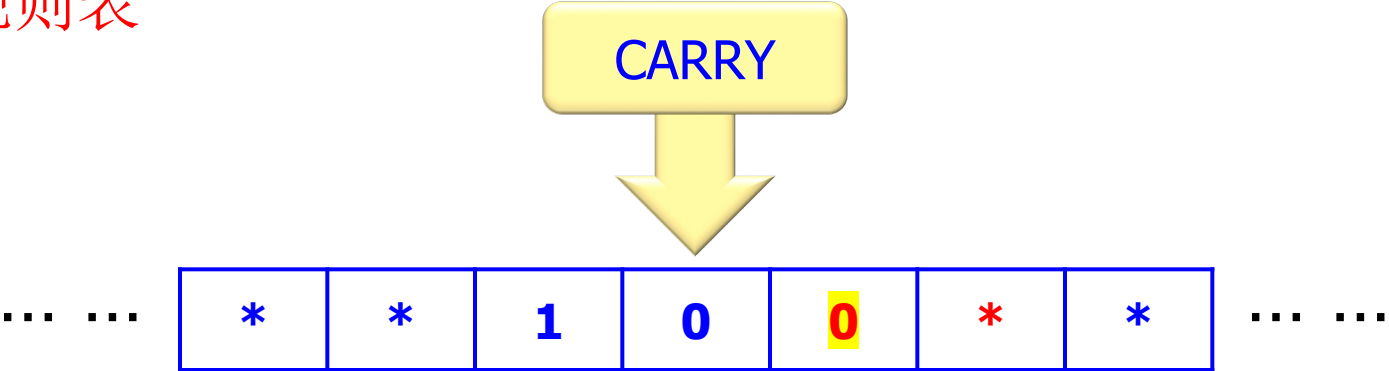
ID	当前状态	当前符号	写入符号	移动方向	新的状态
01	START	*	*	←左移	ADD
02	ADD	0	1	→右移	RETURN
03	ADD	1	0	←左移	CARRY
04	ADD	*	*	→右移	HALT
05	CARRY	0	1	→右移	RETURN
06	CARRY	1	0	←左移	CARRY
07	CARRY	*	1	←左移	OVERFLOW
08	OVERFLOW	*	*	→右移	RETURN
09	RETURN	0	0	→右移	RETURN
10	RETURN	1	1	→右移	RETURN
11	RETURN	*	*	↓不动	HALT

控制规则表



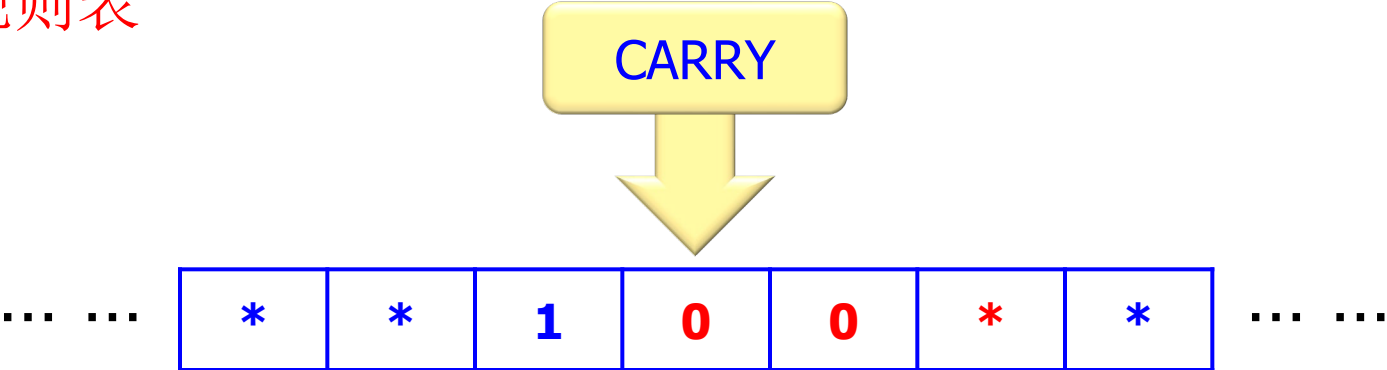
ID	当前状态	当前符号	写入符号	移动方向	新的状态
01	START	*	*	←左移	ADD
02	ADD	0	1	→右移	RETURN
03	ADD	1	0	←左移	CARRY
04	ADD	*	*	→右移	HALT
05	CARRY	0	1	→右移	RETURN
06	CARRY	1	0	←左移	CARRY
07	CARRY	*	1	←左移	OVERFLOW
08	OVERFLOW	*	*	→右移	RETURN
09	RETURN	0	0	→右移	RETURN
10	RETURN	1	1	→右移	RETURN
11	RETURN	*	*	↓不动	HALT

控制规则表



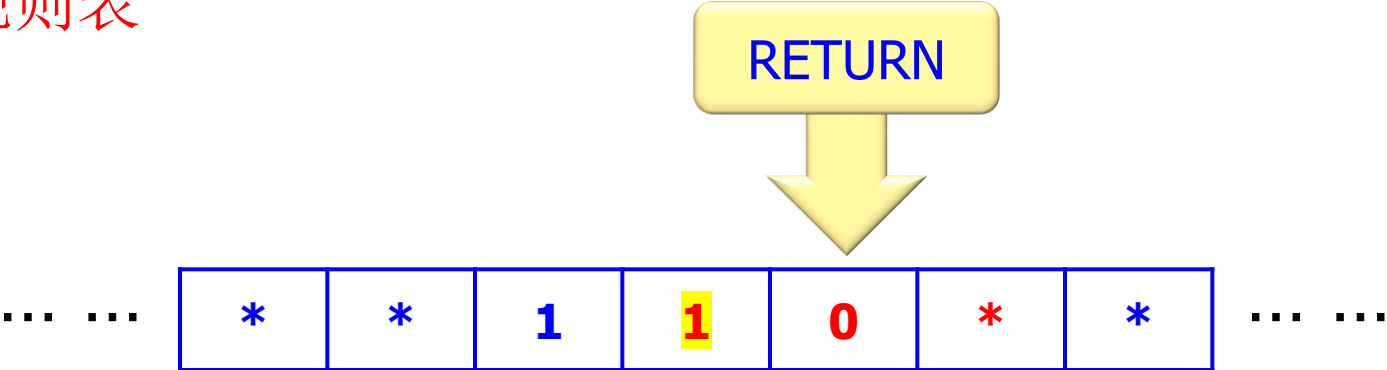
ID	当前状态	当前符号	写入符号	移动方向	新的状态
01	START	*	*	←左移	ADD
02	ADD	0	1	→右移	RETURN
03	ADD	1	0	←左移	CARRY
04	ADD	*	*	→右移	HALT
05	CARRY	0	1	→右移	RETURN
06	CARRY	1	0	←左移	CARRY
07	CARRY	*	1	←左移	OVERFLOW
08	OVERFLOW	*	*	→右移	RETURN
09	RETURN	0	0	→右移	RETURN
10	RETURN	1	1	→右移	RETURN
11	RETURN	*	*	↓不动	HALT

控制规则表



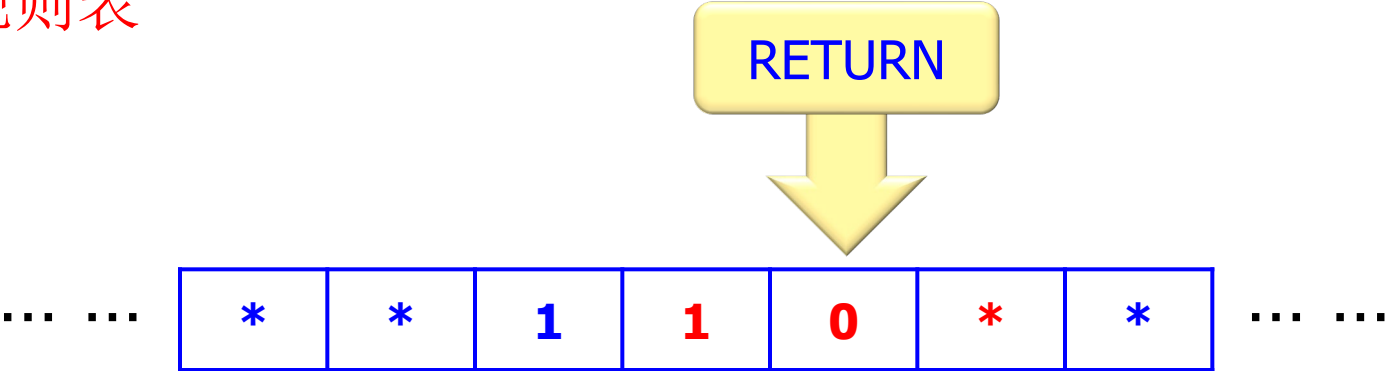
ID	当前状态	当前符号	写入符号	移动方向	新的状态
01	START	*	*	←左移	ADD
02	ADD	0	1	→右移	RETURN
03	ADD	1	0	←左移	CARRY
04	ADD	*	*	→右移	HALT
05	CARRY	0	1	→右移	RETURN
06	CARRY	1	0	←左移	CARRY
07	CARRY	*	1	←左移	OVERFLOW
08	OVERFLOW	*	*	→右移	RETURN
09	RETURN	0	0	→右移	RETURN
10	RETURN	1	1	→右移	RETURN
11	RETURN	*	*	↓不动	HALT

控制规则表



ID	当前状态	当前符号	写入符号	移动方向	新的状态
01	START	*	*	←左移	ADD
02	ADD	0	1	→右移	RETURN
03	ADD	1	0	←左移	CARRY
04	ADD	*	*	→右移	HALT
05	CARRY	0	1	→右移	RETURN
06	CARRY	1	0	←左移	CARRY
07	CARRY	*	1	←左移	OVERFLOW
08	OVERFLOW	*	*	→右移	RETURN
09	RETURN	0	0	→右移	RETURN
10	RETURN	1	1	→右移	RETURN
11	RETURN	*	*	↓不动	HALT

控制规则表



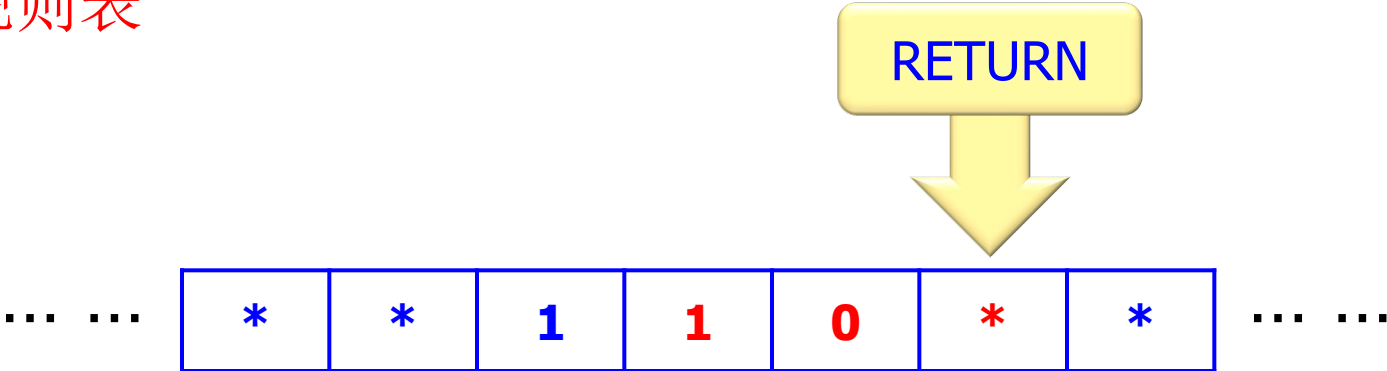
ID	当前状态	当前符号	写入符号	移动方向	新的状态
01	START	*	*	←左移	ADD
02	ADD	0	1	→右移	RETURN
03	ADD	1	0	←左移	CARRY
04	ADD	*	*	→右移	HALT
05	CARRY	0	1	→右移	RETURN
06	CARRY	1	0	←左移	CARRY
07	CARRY	*	1	←左移	OVERFLOW
08	OVERFLOW	*	*	→右移	RETURN
09	RETURN	0	0	→右移	RETURN
10	RETURN	1	1	→右移	RETURN
11	RETURN	*	*	↓不动	HALT

控制规则表



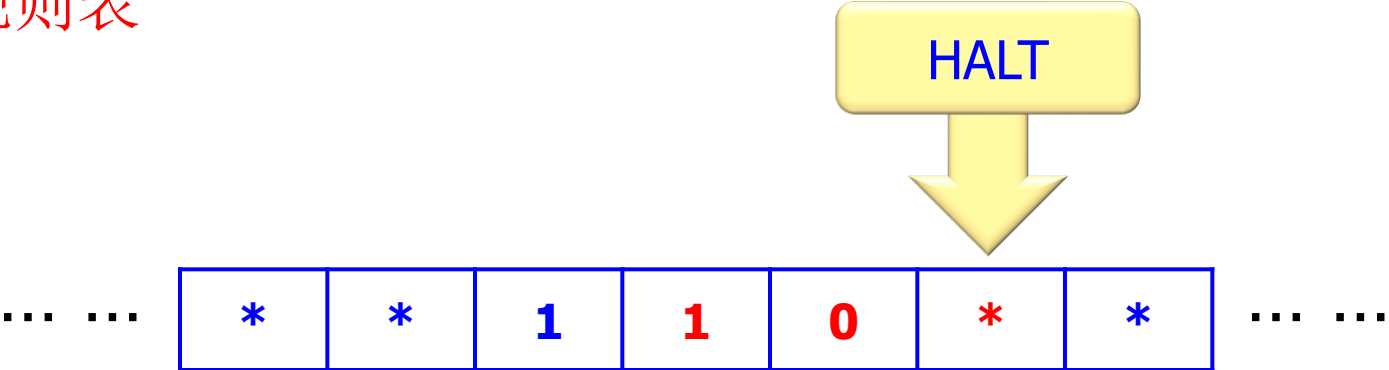
ID	当前状态	当前符号	写入符号	移动方向	新的状态
01	START	*	*	←左移	ADD
02	ADD	0	1	→右移	RETURN
03	ADD	1	0	←左移	CARRY
04	ADD	*	*	→右移	HALT
05	CARRY	0	1	→右移	RETURN
06	CARRY	1	0	←左移	CARRY
07	CARRY	*	1	←左移	OVERFLOW
08	OVERFLOW	*	*	→右移	RETURN
09	RETURN	0	0	→右移	RETURN
10	RETURN	1	1	→右移	RETURN
11	RETURN	*	*	↓不动	HALT

控制规则表



ID	当前状态	当前符号	写入符号	移动方向	新的状态
01	START	*	*	←左移	ADD
02	ADD	0	1	→右移	RETURN
03	ADD	1	0	←左移	CARRY
04	ADD	*	*	→右移	HALT
05	CARRY	0	1	→右移	RETURN
06	CARRY	1	0	←左移	CARRY
07	CARRY	*	1	←左移	OVERFLOW
08	OVERFLOW	*	*	→右移	RETURN
09	RETURN	0	0	→右移	RETURN
10	RETURN	1	1	→右移	RETURN
11	RETURN	*	*	↓不动	HALT

控制规则表





图灵机—实例1

- 这个图灵机的功能是什么？

$$f(x) = x + 1$$

- 启示：理解计算机程序执行的顺序性

ID	当前状态	当前符号	写入符号	移动方向	新的状态
01	START	*	*	←左移	ADD
02	ADD	0	1	↓不动	HALT
03	ADD	1	0	←左移	CARRY
04	ADD	*	*	→右移	HALT
05	CARRY	0	1	↓不动	HALT
06	CARRY	1	0	←左移	CARRY
07	CARRY	*	1	←左移	OVERFLOW
08	OVERFLOW	*	*	↓不动	HALT
09	RETURN	0	0	↓不动	HALT
10	RETURN	1	1	↓不动	HALT
11	RETURN	*	*	↓不动	HALT

$f(x)=x+1$ 的另一种控制规则表示

通用图灵机

- 通用图灵机：能够模拟任意一个图灵机对任意一个输入的计算
 - 可以解释任意图灵机的行为
- 图灵停机悖论：是否存在一个程序 P ，能够判断出任意一个程序 X 是否会在输入数据 Y 的情况下停机？
- 图灵测试：如果一台机器能够与人类展开对话（通过电传设备）而不能被辨别出其机器身份，那么称这台机器具有智能。



图灵测试示意图



CPU的内部结构和工作原理

CPU — 中央处理器

- Central Processing Unit

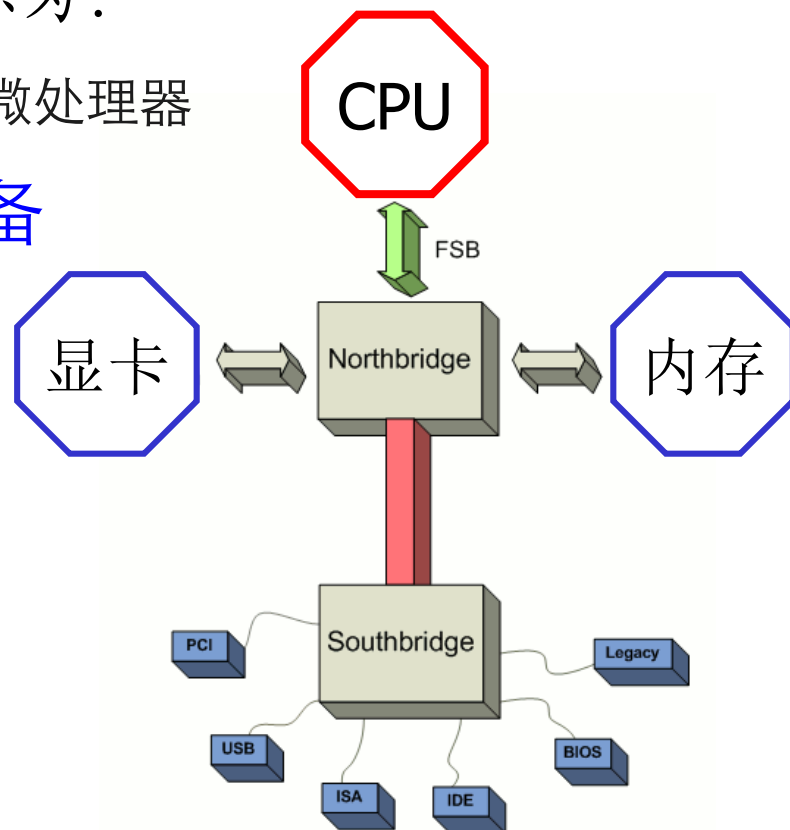
- 微型/个人计算机的CPU又被称为：

- MPU (Micro Processor Unit) 微处理器

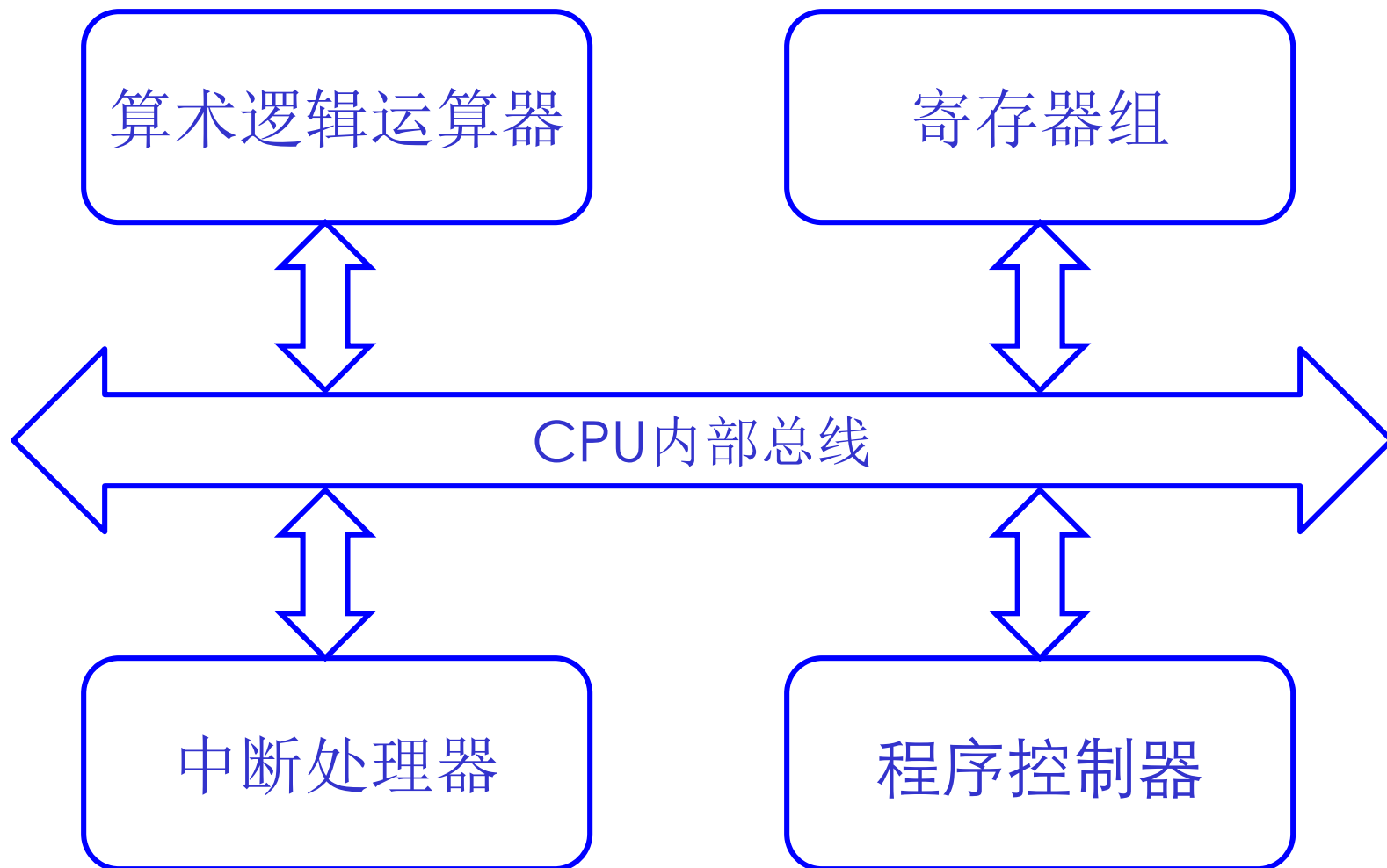
- 计算机系统核心硬件设备

- 主要功能：执行程序

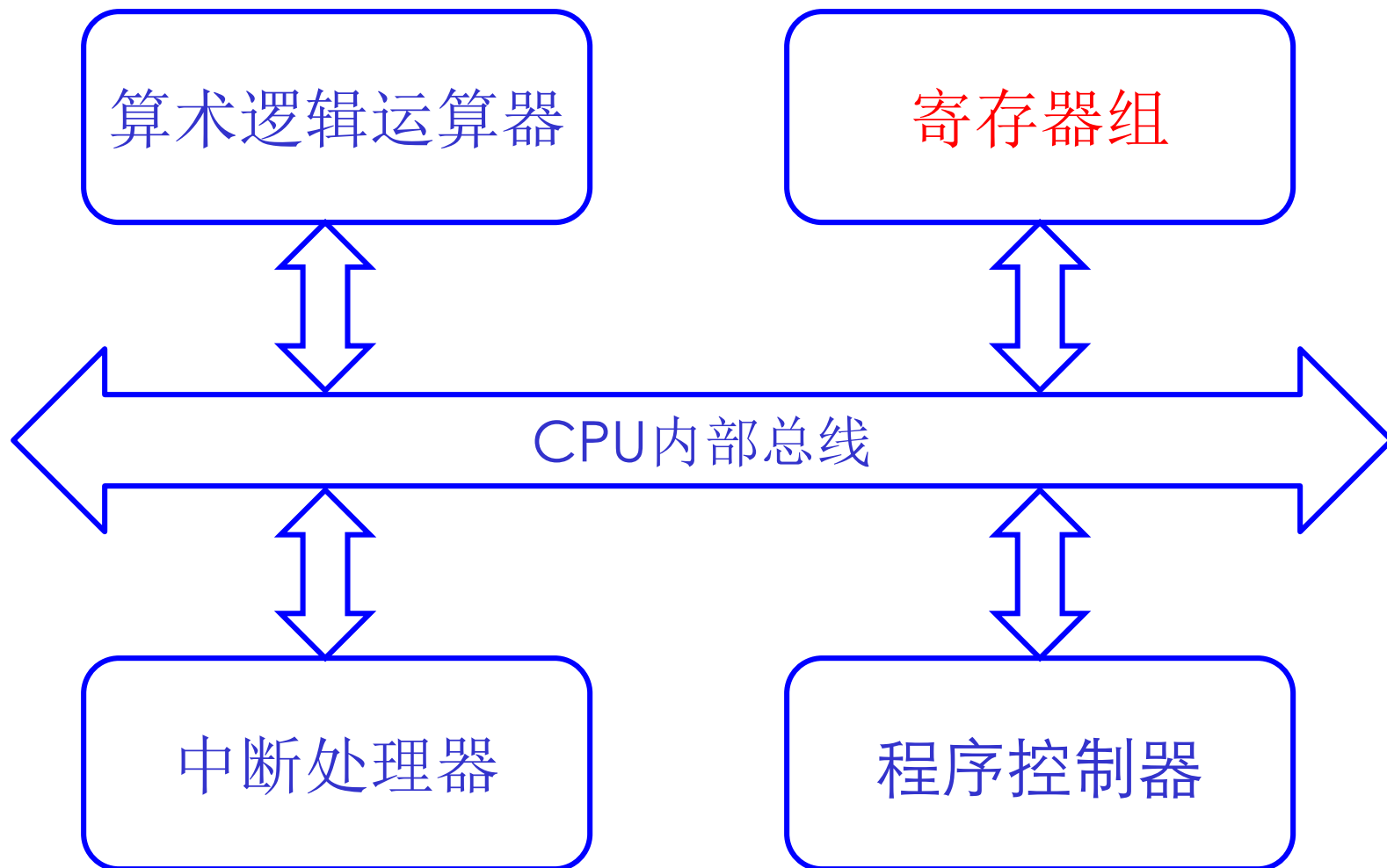
- 与其它部件协同工作



CPU的内部结构

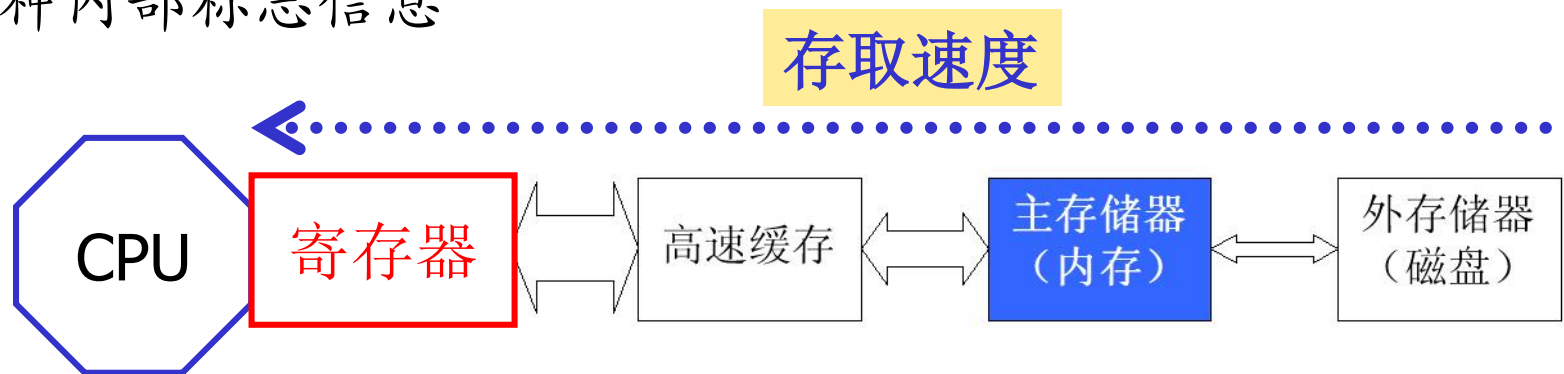


CPU的内部结构

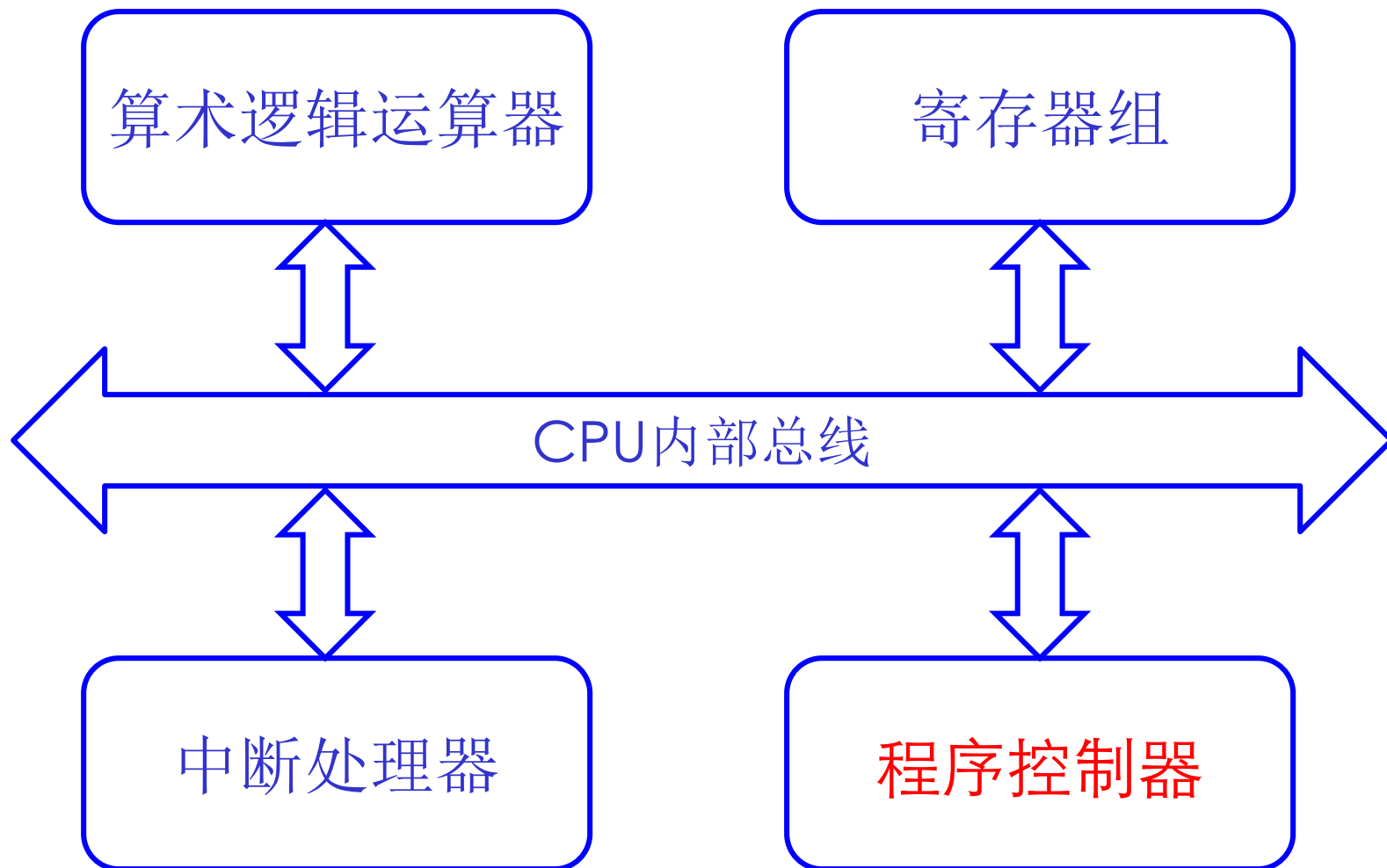


CPU的内部结构 — 寄存器组

- 由 一组寄存器 组成的高速存储单元
- 用于 暂时存放 运算数据 或 其它信息 (通用/专用)
 - 整数类型的操作数或运算结果
 - 浮点数类型的操作数或运算结果
 - 指令
 - 指令地址
 - 各种内部标志信息



CPU的内部结构

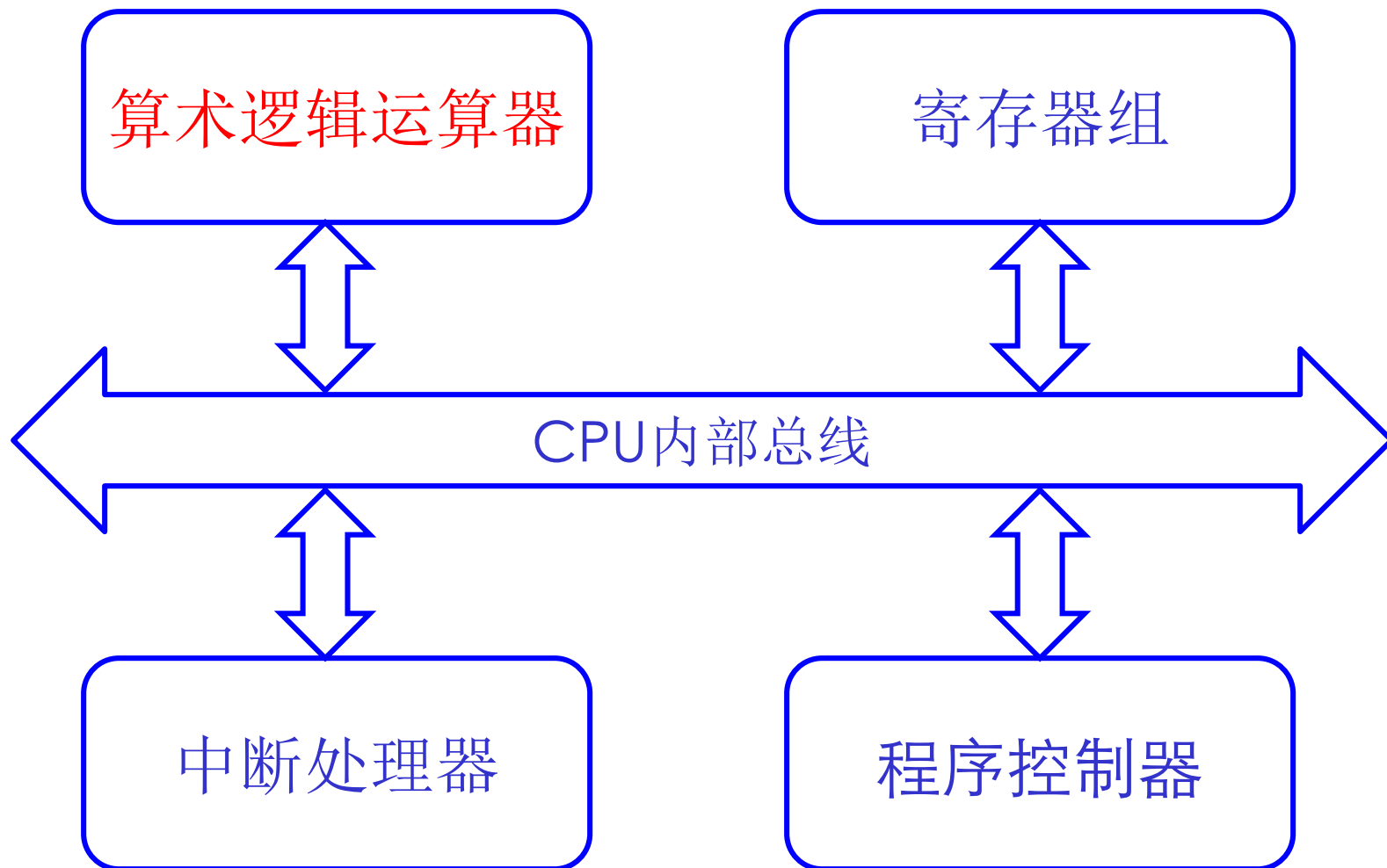


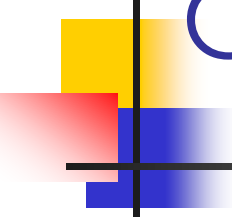


CPU的内部结构 — 程序控制器

- Program Control Unit, CPU的控制中心
 - 分析/解释 指令
 - 根据 分析/解释 结果 向 其它部件 发出命令
 - 控制CPU的工作进度和工作方式
- 具体而言，当一条指令进入CPU后，程序控制器：
 1. 分析/解释该指令的编码内容；
 2. 确定为执行该指令应该完成的动作；
 3. 确定指令相关的参数；例如：对于一个“加法指令”，需要确定两个被加数的地址
 4. 将所需的数据从主存储器读取到CPU的寄存器中；
 5. 要求算术逻辑运算器进行相关的运算动作；
 6. 指示算术逻辑运算器将运算结果放入寄存器或主存储器中。

CPU的内部结构

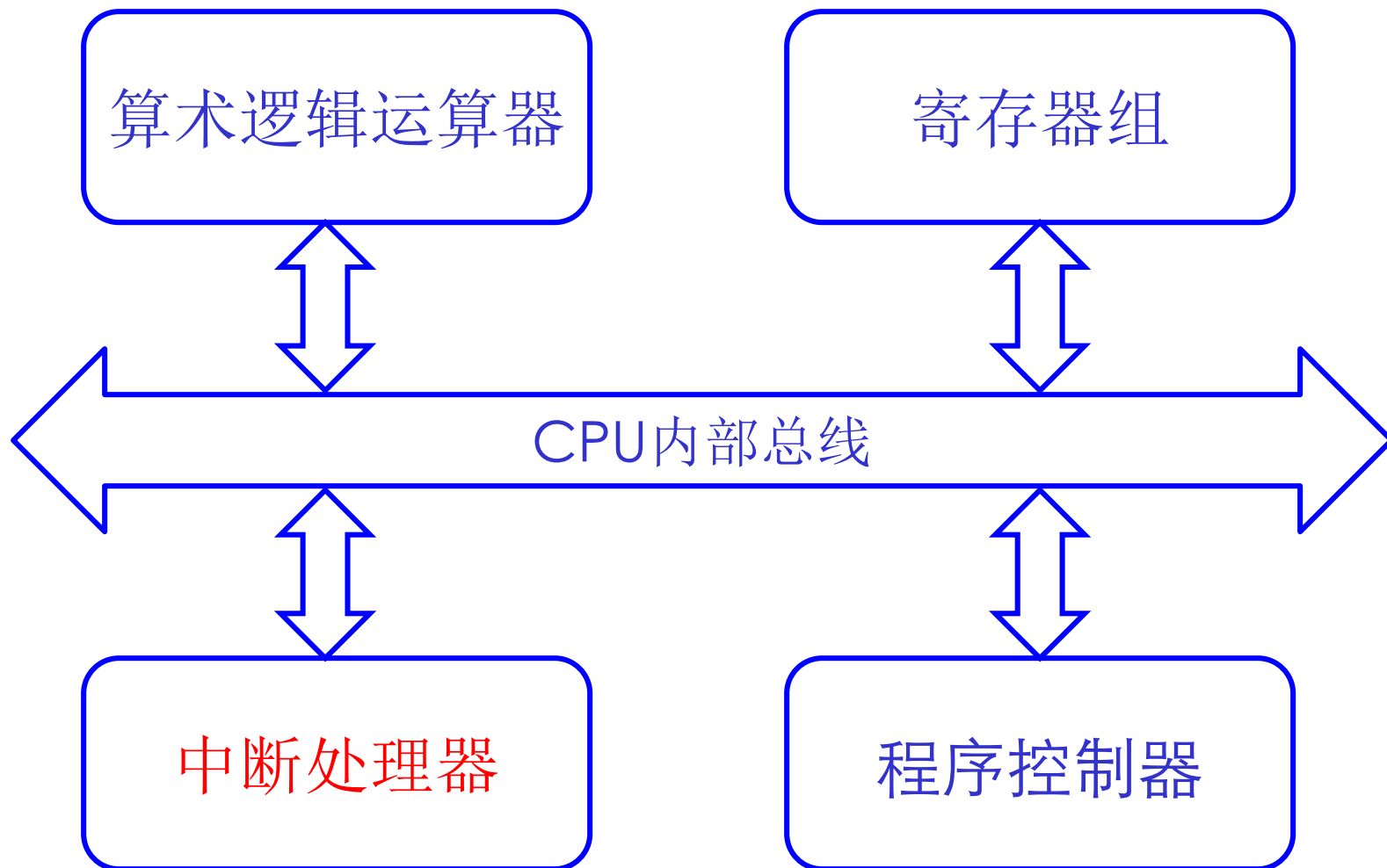




CPU的内部结构 — 算术逻辑运算器

- Arithmetic Logical Unit (ALU), 主要进行算术运算和逻辑运算
- 加法指令的例子
 1. 一条加法指令（其中包含了两个被加数/操作数的地址）进入CPU;
 2. 程序控制器 分析该指令，判断两个操作数是在寄存器内，还是在主存内;
 3. 如果在主存内，程序控制器 从主存内读入操作数;
 4. 程序控制器 将加法运算 提交给 ALU;
 5. ALU 进行加法运算;
 6. ALU 根据程序控制器的指示，将运算结果存放 到寄存器或主存中。

CPU的内部结构





什么是中断？

- 中断 (interrupt): 正常过程中出现的紧急/意外情况
- 可以预知的：预案处理
 - 例子1：消防预案
 - 例子2：手术过程中出现大出血，处理大出血，继续手术
- 不可预知的：临机决策/无可奈何
 - 例子1：2019年新冠病毒的爆发
 - 例子2：到达机场后发现航班延误或者临时取消



CPU的内部结构 — 中断处理器

- 问题背景：
 - 在CPU执行一般程序运算的过程中，如何处理紧急出现的事件？比如：鼠标移动事件
- 发生一个紧急事件 → 触发一个中断信号
- 中断信号的处理：
 - 当发现中断信号后，程序控制器暂停正在运行的程序，保存该程序的运行现场（CPU内的各种状态信息）；
 - 程序控制器根据中断信号的编码，从特定位置启动中断处理程序（由操作系统提供）；
 - 中断处理程序运行完毕后，程序控制器恢复被暂停的程序。



CPU的内部结构 — 中断处理器

- 中断信号的产生：
 - 各种软硬件，比如：鼠标、键盘、其它外设…
- 中断信号的接收：
 - 中断处理器 负责 中断信号的接收，并将中断信号的编码、中断处理程序的起始地址 传给 程序控制器
- 中断信号的检测
 - 程序控制器 在每条指令执行完毕后，都会检测 是否出现了新的中断信号



中断的例子

中断向量号	异常事件	Linux的处理程序
0	除法错误	Divide_error
1	调试异常	Debug
2	NMI中断	Nmi
3	单字节, int 3	Int3
4	溢出	Overflow
5	边界监测中断	Bounds
6	无效操作码	Invalid_op
7	设备不可用	Device_not_available
8	双重故障	Double_fault

9	协处理器段溢出	Coprocessor_segment_overnun
10	无效TSS	Incalid_tss
11	缺段中断	Segment_not_present
12	堆栈异常	Stack_segment
13	一般保护异常	General_protection
14	页异常	Page_fault
15	(intel保留)	Spurious_interrupt_bug
16	协处理器出错	Coprocessor_error
17	对齐检查中断	Alignment_check

Linux 系统软件中断



CPU的主要性能指标

- 工作主频：
 - CPU内部的时钟频率；
- 运算字长：
 - CPU一次能够处理的二进制位数； 32位/64位
- 运算速度：
 - 每秒钟执行的指令数；例如：1000MIPS (Million Instructions Per Second)
 - IPC (instruction per cycle)
- 多核CPU：
 - 核心数量； 每个核心线程数



指令系统

指令系统

- 指令 (program instruction)
 - 组成程序的基本单位
- 每一条指令：
 - 规定了CPU执行指令应该完成的工作：运算、或其它控制动作等
 - 控制CPU的相关部件执行微操作，从而完成指令所规定的功能



早期计算机



指令系统

- 每一条指令用若干字节的二进制编码表示
 - 包括所要完成的**动作**及其**相关的参数**
- 指令的分类：
 - 存储访问指令
 - 算术运算指令
 - 逻辑运算指令
 - 条件判断和分支转移指令
 - 输入输出指令
 - 其他用于系统控制的指令

8086 INSTRUCTION SET

OPCODE	DESCRIPTION				
AAA		ASCII adjust addition	JNAE	label	Jump if not above or equal
AAD		ASCII adjust division	JNB	label	Jump if not below
AAM		ASCII adjust multiply	JNBE	label	Jump if below or equal
AAS		ASCII adjust subtraction	JNC	label	Jump if no carry
ADC	dt,sc	Add with carry	JNE	label	Jump if not equal
ADD	dt,sc	Add	JNG	label	Jump if not greater
AND	dt,sc	Logical AND	JNGE	label	Jump if not greater or equal
CALL	proc	Call a procedure	JNL	label	Jump if not less
CBW		Convert byte to word	JNLE	label	Jump if not less or equal
CLC		Clear carry flag	JNZ	label	Jump if not zero
CDI		Clear direction flag	JNO	label	Jump if not overflow
CLI		Clear interrupt flag	JNP	label	Jump if not parity
CMC		Complement carry flag	JNS	label	Jump if not sign
CMP	dt,sc	Compare	JO	label	Jump if overflow
CMPS	[dt,sc]	Compare string	JPO	label	Jump if parity odd
CMPSB	"	" bytes	JP	label	Jump if parity
CMPSW	"	" words	JPE	label	Jump if parity even
CWD		Convert word to double word	JS	label	Jump if sign
DAA		Decimal adjust addition	JZ	label	Jump if zero
DAS		Decimal adjust subtraction	LAHF		Load AH from flags
DEC	dt	Decrement	LDS	dt,sc	Load pointer using DS
DIV	sc	Unsigned divide	LEA	dt,sc	Load effective address
ESC	code,sc	Escape	LES	dt,sc	Load pointer using ES
HLT		Halt	LOCK		Lock bus
IDIV	sc	Integer divide	LODS	[sc]	Load string
IMUL	sc	Integer multiply	LODSB	"	" bytes
IN	ac,port	Input from port	LODSW	"	" words
INC	dt	Increment	LOOP	label	Loop
INT	type	Interrupt	LOOPE	label	Loop if equal
INTO		Interrupt if overflow	LOOPZ	label	Loop if zero
IRET		Return from interrupt	LOOPNE	label	Loop if not equal
JA	label	Jump if above	LOOPNZ	label	Loop if not zero
JAE	label	Jump if above or equal	MOV	dt,sc	Move
JB	label	Jump if below	MOVS	[dt,sc]	Move string
JBE	label	Jump if below or equal	MOVSB	"	" bytes
JC	label	Jump if carry	MOVSW	"	" words
JCXZ	label	Jump if CX is zero	MUL	sc	Unsigned multiply
JE	label	Jump if equal	NEG	dt	Negate
JG	label	Jump if greater	NOP		No operation
JGE	label	Jump if greater or equal	NOT	dt	Logical NOT
JL	label	Jump if less	OR	dt,sc	Logical OR
JLE	label	Jump if less or equal	OUT	port,ac	output to port
JMP	label	Jump	POP	dt	Pop word off stack
JNA	label	Jump if not above	POPF		Pop flags off stack
			PUSH	sc	Push word onto stack
			PUSHF		Push flags onto stack
			RCL	dt,cnt	Rotate left through carry
			RCR	dt,cnt	Rotate right through carry
			REP		Repeat string operation
			REPE		Repeat while equal
			REPZ		Repeat while zero
			REPNE		Repeat while not equal
			REPNZ		Repeat while not zero
			RET	[pop]	Return from procedure
			ROL	dt,cnt	Rotate left
			ROR	dt,cnt	Rotate right
			SAHF		Store AH into flags
			SAL	dt,cnt	Shift arithmetic left
			SHL	dt,cnt	Shift logical left
			SAR	dt,cnt	Shift arithmetic right
			SBB	dt,sc	Subtract with borrow
			SCAS	[dt]	Scan string
			SCASB	"	" byte
			SCASW	"	" word
			SHR	dt,cnt	Shift logical right
			STC		Set carry flag
			STD		Set direction flag
			STI		Set interrupt flag
			STOS	[dt]	Store string
			STOSB	"	" byte
			STOSW	"	" word
			SUB	dt,sc	Subtraction
			TEST	dt,sc	Test (logical AND)
			WAIT		Wait for 8087
			XCHG	dt,sc	Exchange
			XLAT	table	Translate
			XLATB	"	"
			XOR	dt,sc	Logical exclusive OR

Notes:

dt - destination

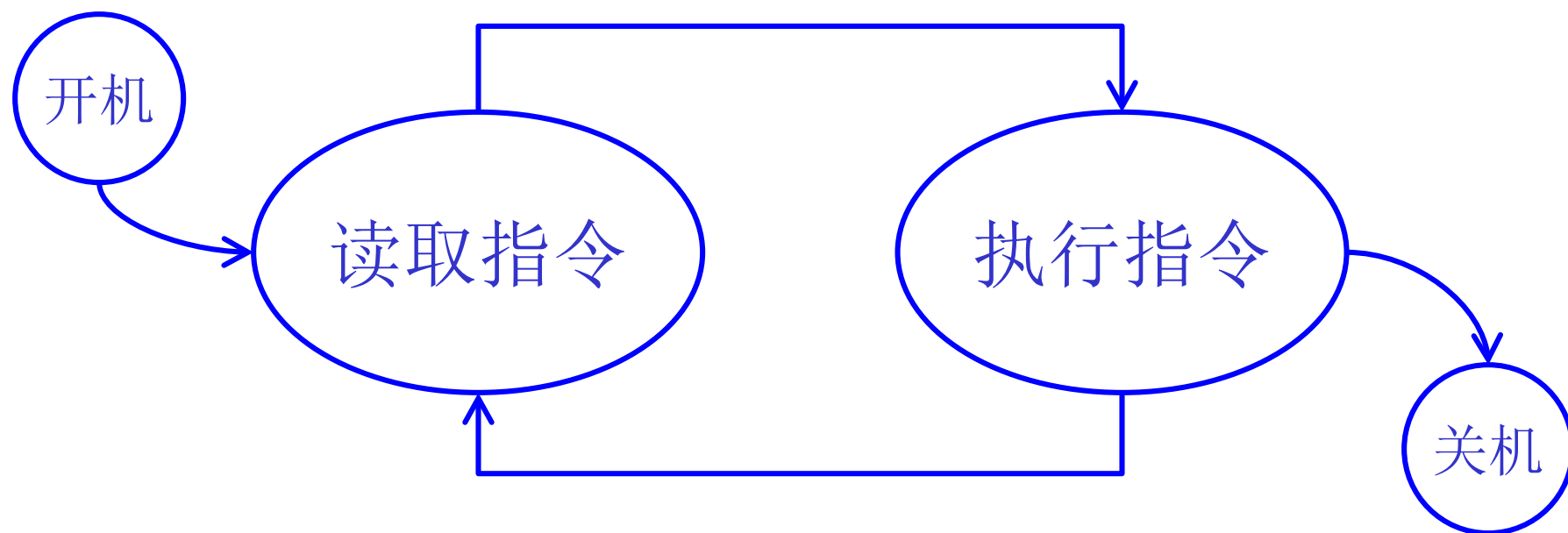
sc - source

label - may be near or far address

label - near address

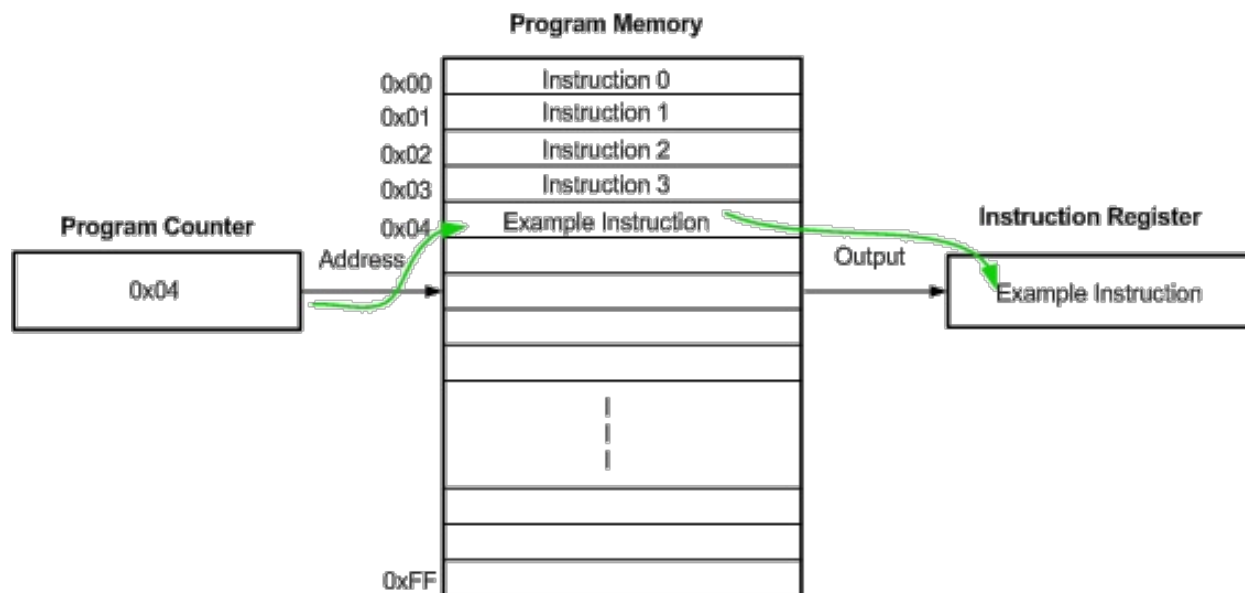
指令工作周期

- 程序控制器 按照 “读取指令—执行指令” 的周期循环地工作



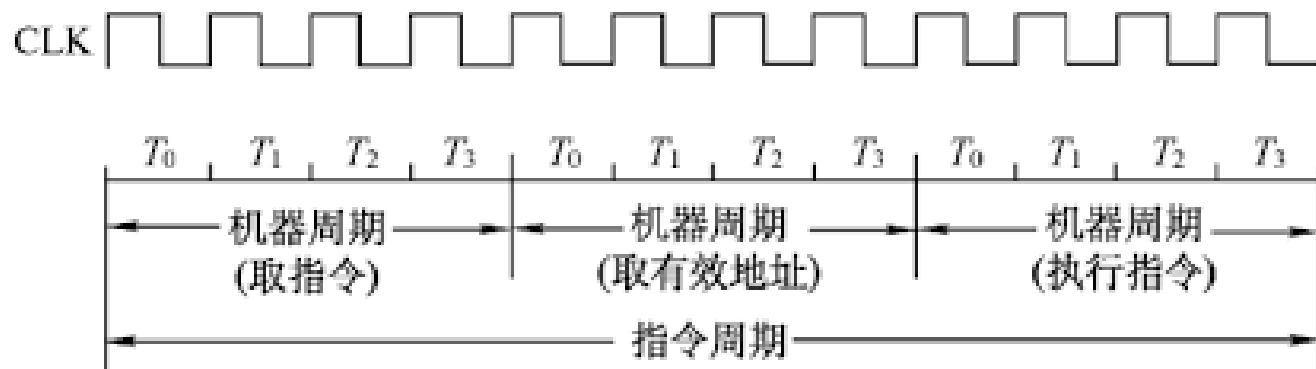
如何确定下一条要执行的指令？

- CPU中有一个特殊的寄存器 — **指令计数器**
 - 其中保存了**下一条**要执行指令在内存中的**地址**
- 程序指令在内存中顺序存放
 - 每条指令的长度都确定 (但不同指令的长度不一定相同)



指令工作周期

- 一个指令周期 一般需要 占用 多个CPU时钟周期/时钟节拍
- CPU时钟周期/时钟节拍
 - CPU完成一个原子动作的基本时间单位
 - 如果 一个CPU的时钟频率是 y GHz，那么这个CPU的时钟周期/时钟节拍 是 $1/y \cdot 10^{-9}$ 秒





指令工作周期

■ 问题：

- 一个CPU能够在 4 个 时钟周期/时钟节拍 内完成 一个指令周期
- 如果这个 CPU 的时钟频率/主频是 2GHz，问这个 CPU 在一秒之内能执行多少条指令？这个 CPU 的运算速度是多少？

■ 答：

- 一秒中内可以执行的指令个数为 $2 * 10^9 \text{ Hz} * 1 \text{ 秒} / 4 = 5 * 10^8$ 个
- 运算速度为 500 MIPS

主存储器

及其与CPU之间的信息传输



主存储器/内存

- 由基于大规模集成电路的存储芯片组装而成，存储CPU可直接访问的数据和程序
- 主存储器的工作速度和容量对计算机系统整体性能影响极大
 - 一次读写大约在几十纳秒 (ns) 左右，1纳秒 = 10^{-9} 秒
- 主存储器容量的基本计量单位为字节 (Byte)
 - 目前常见的计算机标配内存容量为 GB 级
- 主要功能：存储数据和读写数据

存储空间的管理

- 为了更有效地进行管理，通常以 8 个比特（一个字节）为一个**存储单元**
- 每个存储单元都有其特定且唯一的地址——**存储地址**
 - 存储地址为整数编码，可表示为**二进制整数**
- **地址空间**：由主存储器的所有存储单元的地址构成的集合
- **地址宽度**：表示地址空间所需的二进制位数
 - 内存容量越大，地址空间也就越大，地址宽度也必须相应加大

地址	存储单元								
0 (0000)	1	0	0	1	0	1	0	1	→ 1Byte
1 (0001)	1	1	0	0	1	0	1	0	
2 (0010)	0	1	0	0	1	0	1	1	4Byte
3 (0011)	0	1	1	0	0	1	0	0	
4 (0100)	1	0	1	0	1	0	1	1	
	1	1	1	0	0	1	0	1	
	1	1	0	1	0	1	1	0	
	1	0	0	0	1	0	0	1	
	1	1	0	1	0	1	1	0	8Byte
	1	0	0	1	0	1	1	1	
	0	0	0	0	1	0	0	0	
	1	0	1	0	1	1	1	1	
	0	0	0	1	1	1	0	1	
	0	0	1	1	1	0	0	0	
14 (1110)	1	0	0	0	0	0	1	0	
15 (1111)	0	1	0	1	1	1	0	1	

主存储器的访问方式

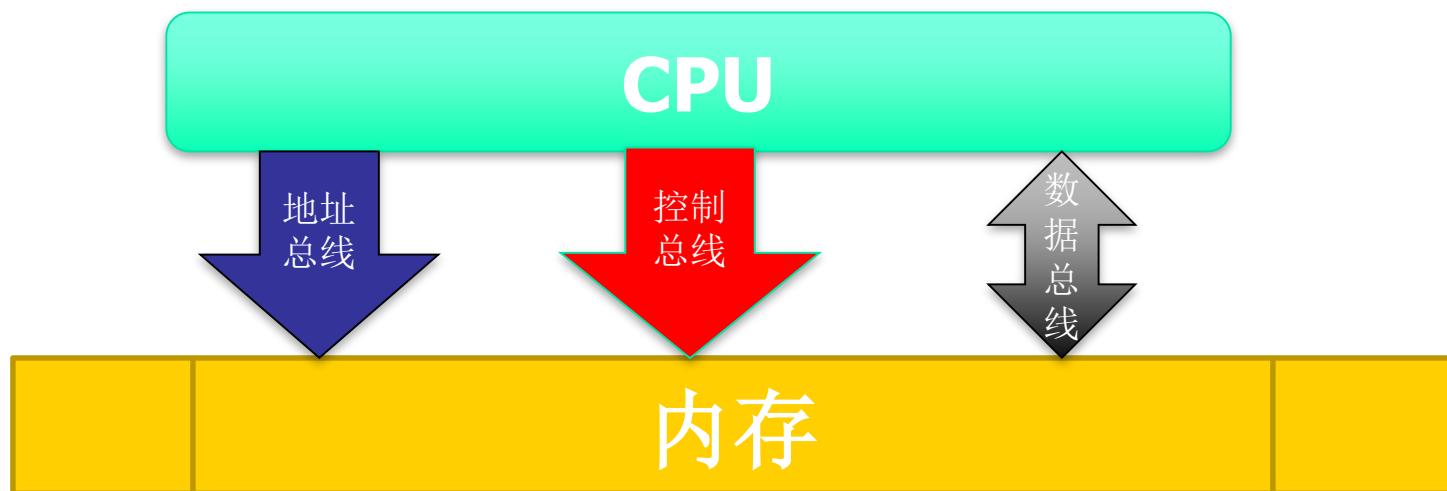
这真的对我
有用么？



- 地址访问方式
 - 给出想要访问的存储单元的地址
 - 从该地址的存储单元中读取数据，或向其中写入数据
- 随机访问存储器（RAM — Random Access Memory）：访问时间不随访问地址的不同而不同，也即，读写任意地址的存储单元，其所需时间是一样的
 - （相对概念：顺序访问存储器）
- CPU的字长一般是4个字节 或 8个字节，因此
 - CPU读写内存数据的方式是每次4个或8个字节
- 在一段程序中，变量 和 存储单元 相对应
 - 变量名字 对应于 存储单元地址
 - 变量内容 对应于 存储单元中的数据
 - 指针型变量：专门存放 存储单元地址 的 变量

主存和CPU之间的信息传输

- 通过 **存储总线** 进行信息传输
- 存储总线由三组总线构成
 - **数据总线**：用于传输数据
 - **地址总线**：用于传输存储单元地址
 - **控制总线**：用于各种控制信息的传递（读、写等）





主存和CPU之间的信息传输

- CPU 从主存中读取数据的过程
 1. CPU 把 存储单元地址 写入 地址总线
 2. CPU 通过 控制总线 发出一个“读”信号
 3. 主存 收到“读”信号，根据 地址总线 上的 地址信息，把连续几个存储单元的数据读出，送到 数据总线；
(需要一定的时间)
 4. 在等待一段时间后，CPU 从 数据总线 上 获得数据



主存和CPU之间的信息传输

- CPU 向主存写入数据的过程
 1. CPU 把 存储单元地址 写入 地址总线
 2. CPU 把 数据 写入 数据总线
 3. CPU 通过 控制总线 发出一个“写”信号
 4. 主存 收到“写”信号，根据 地址总线 上的 地址信息，把 数据总线 上的 数据 写入到 相应的 存储单元 中（需要一定的时间）



总线的宽度

- 一条总线上 一次可传输的二进制位数
- 数据总线的宽度 一般 和 CPU 的 字长 相同
 - 目前 CPU 一般采用 32 位或 64 位的数据总线
 - 数据总线宽度决定了一次传送数据量的大小
- CPU 的 地址总线宽度 决定主存储器地址 (寻址) 空间的大小
 - 16位地址 (64K)
 - 32位地址 (4G)
 - 64位地址 (天文数字)



小结：CPU与内存的工作原理

- 计算机的数学理论模型 – 图灵机
- CPU的内部结构和工作原理
- 指令系统
- 主存储器及其与CPU之间的信息传输

思考题：考虑任意由 0、1 构成的序列 c ，请定义一个图灵机 T ，如果 c 形如 $0\cdots 01\cdots 1$ ，且其中 0、1 的个数相同，则 T 关于 c 的计算结束（即停机），否则不结束。定义图灵机时，可以自由选择合适的方式，比如形式化定义方法（即给出状态集合、字符集合、转换函数），状态转换图等。