第8章 中央处理器（一）

第一讲 中央处理器概述

第二讲 单周期数据通路的设计

第三讲 单周期控制器的设计

第四讲 多周期处理器的设计

第五讲 流水线处理器设计

第六讲 流水线冒险及其处理

第七讲 高级流水线技术

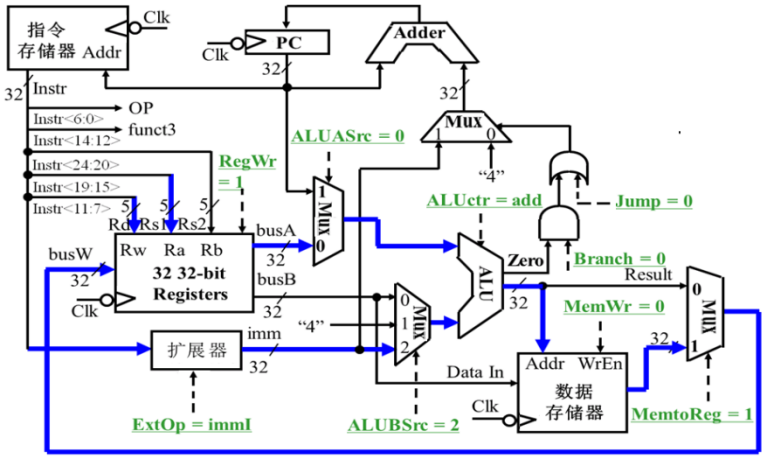
第四讲 多周期处理器的设计

主 要 内 容

- 多周期数据通路实现思想
- 单周期数据通路和多周期数据通路的差别
  - 通过简要分析LOAD指令分阶段执行过程，以加深理解单周期和多周期数据通路的差别
- 实现目标：一个简单指令系统
- 多周期数据通路设计
- 硬连线控制器和微程序控制器
- 带异常处理的处理器设计

单周期处理器时钟周期的特点

- 单周期处理器的CPI为1
- 所有指令执行时间都是1个时钟周期
- 但是，时钟周期必须以最长的load指令为准



- 因此，时钟周期远远大于其他指令实际所需的执行时间，效率低

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200	0	550 ps
Branch	200	50	100	0	0	350 ps
Jump	200	0	100	0	50	350 ps

- 单周期处理器的问题根源:
  - 时钟周期以最复杂指令所需时间为准，太长！

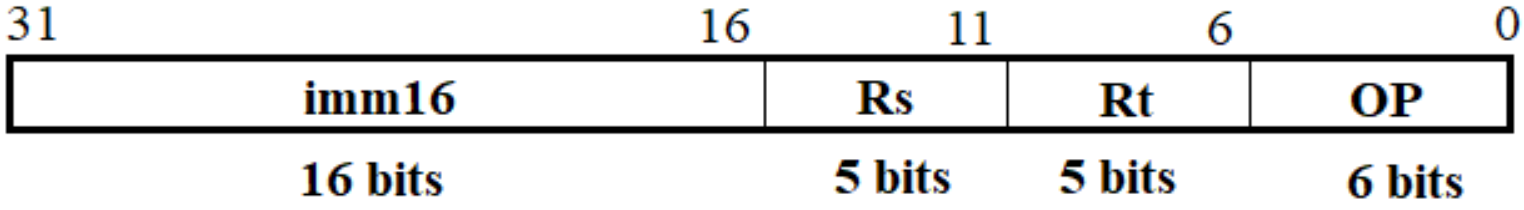
多周期处理器的实现思想

- 解决思路:
  - 把指令的执行分成多个阶段，每个阶段在一个时钟周期内完成
    - 时钟周期以最复杂阶段所花时间为准
    - 尽量分成大致相等的若干阶段
    - 规定每个阶段最多只能完成1次访存 或 寄存器堆读/写 或 ALU
  - 每步都设置状态单元，每步的执行结果都在下个时钟开始保存到相应单元
- 多周期处理器的好处:
  - 时钟周期短
  - 不同指令所用周期数可以不同（前两个周期每条指令都一样），如:
    - Load：4? 5? cycles
    - 其它： 2? 3? 4? cycles
  - 允许功能部件在一条指令执行过程中被重复使用。如:
    - Adder + ALU（多周期时只用一个ALU，在不同周期可重复使用）
    - Inst. / Data mem（多周期时合用，不同周期中重复使用）

以下以一个简单的指令系统为实现目标，来介绍多周期处理的设计

多周期处理器实现目标：一个简单指令系统

- 只有一种指令格式



- 可实现以下几类指令功能

① R-型：R[Rt]←R[Rs] op R[Rt]，两个寄存器内容运算，结果送寄存器。

② I-型运算：R[Rt]←R[Rs] op EXT[imm16]，寄存器内容和立即数运算。

③ Load：R[Rt]←M[R[Rs]+SEXT[imm16]]，地址为寄存器内容加立即数。

④ Store：M[R[Rs]+SEXT[imm16]]←R[Rt]，地址为寄存器内容加立即数。

⑤ Jump指令：PC←PC+SEXT[imm16]，跳转目标地址为PC内容加立即数。

EXT[imm16]：对16位立即数imm16进行扩展，转换为32位操作数。
对于逻辑运算和无符号整数算术运算，采用零扩展；对于带符号整数算术运算，则采用符号扩展。

SEXT[imm16]：符号扩展。

指令各阶段分析

单周期CPU不需要IR

- 取指令阶段
  - 执行一次存储器读操作
  - 读出的内容（指令）保存到寄存器IR（指令寄存器）中
  - IR的内容不是每个时钟都更新，所以IR必须加一个“写使能”控制
  - 在取指令阶段结束时，ALU的输出为PC+4，并送到PC的输入端，但不能在每个时钟到来时就更新PC，所以PC也要有“写使能”控制

第一个周期（取指周期）

- 译码/读寄存器堆阶段
  - 经过控制逻辑延迟后，控制信号更新为新值
  - 执行一次寄存器读操作，并同时进行译码
  - 期间ALU空闲，可以考虑“投机计算”地址

第二个周期
（译码取数周期）

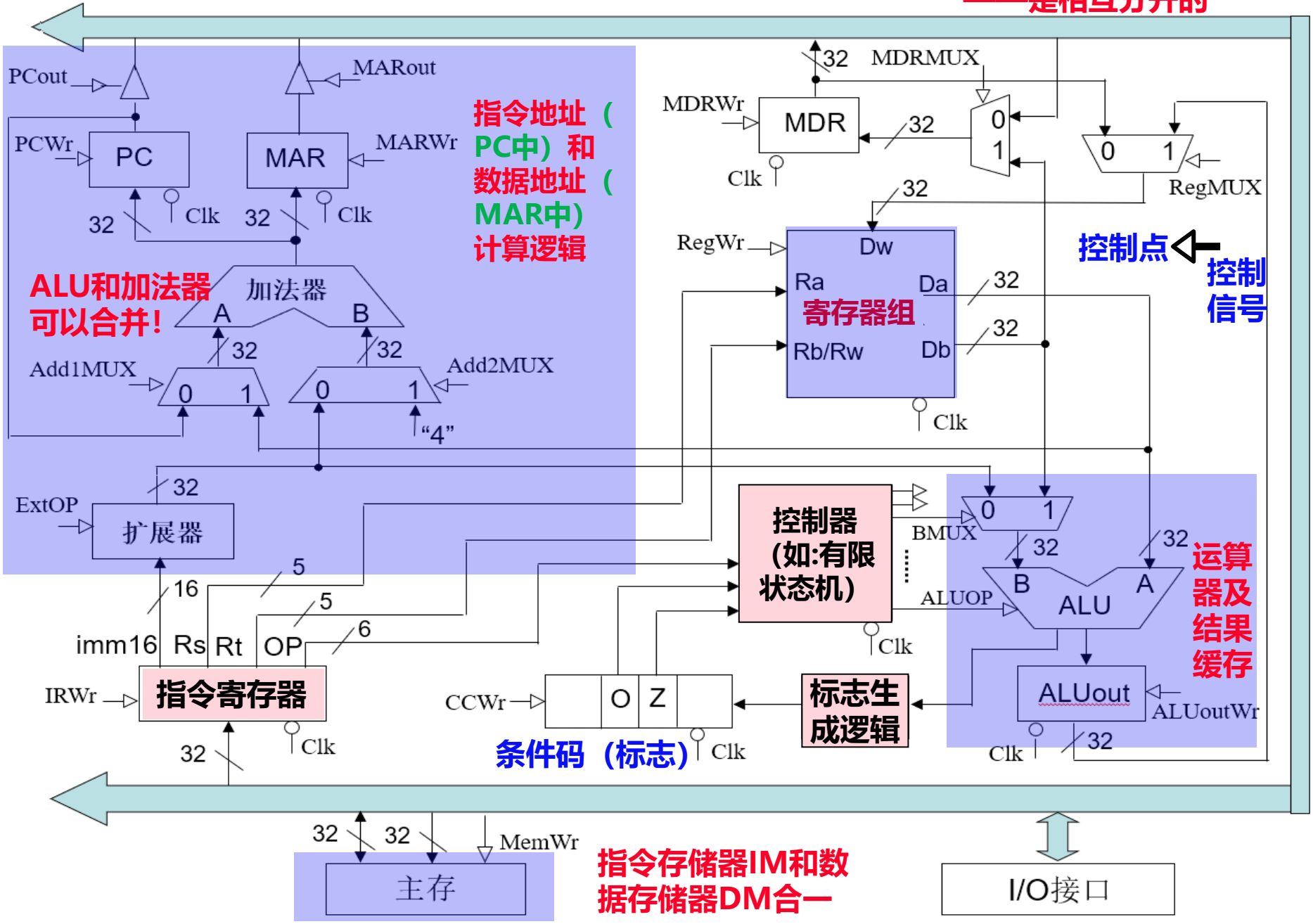
单周期CPU需要在一个时钟周期内完成整条指令的所有功能，ALU不可能空闲

- ALU运算阶段
  - ALU运算，输出结果一定要在下个时钟到达之前稳定
- 读存储器阶段
  - 由ALU运算结果作为地址访问存储器，读出数据
- 写结果到寄存器
  - 把之前的运算结果或读存储器结果写到寄存器堆中

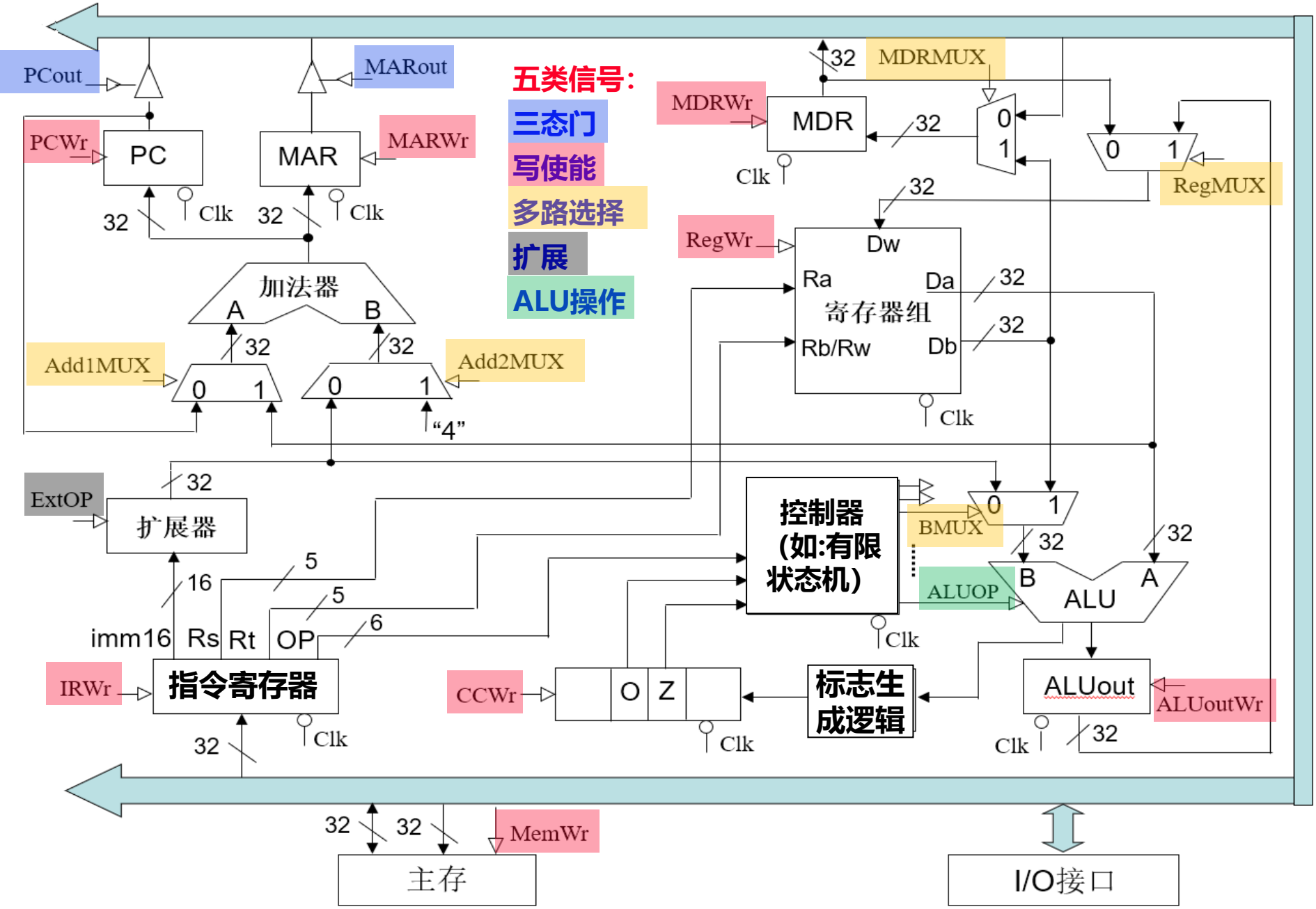
后面的周期（
各指令不同）

简单指令系统对应的多周期CPU

系统总线：AB+DB+CB
——是相互分开的

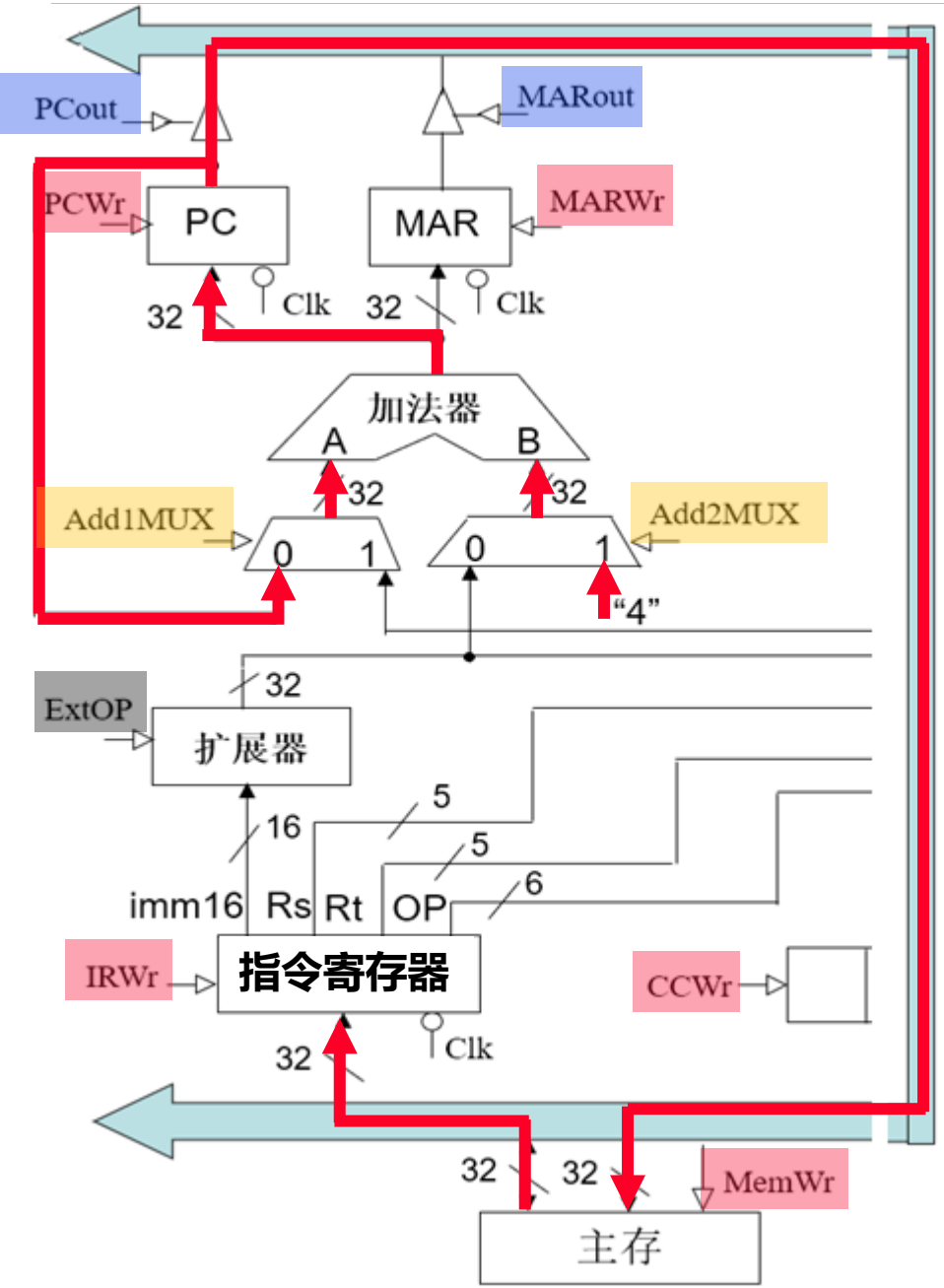


多周期CPU中的控制信号



各类指令执行过程分析

（1）取指令并计算下条指令地址（公共操作），记为 IFetch



°根据PC读指令并保存到IR，PC+4

°IR的内容不是每个时钟都更新，所以IR必须加一个“写使能”控制

°不是每个时钟到来时都更新PC，所以PC也要有“写使能”控制

°有效控制信号及其取值如下：

① R[IR] ← M[PC]：PCout=1、MARout=0、MemWr=0、IRWr=1

② PC ← PC+4：Add1MUX=0、Add2MUX=1、PCWr=1

③ 其他写使能信号（如MARWr、CCWr、MDRWr、ALUoutWr、RegWr）全部为0

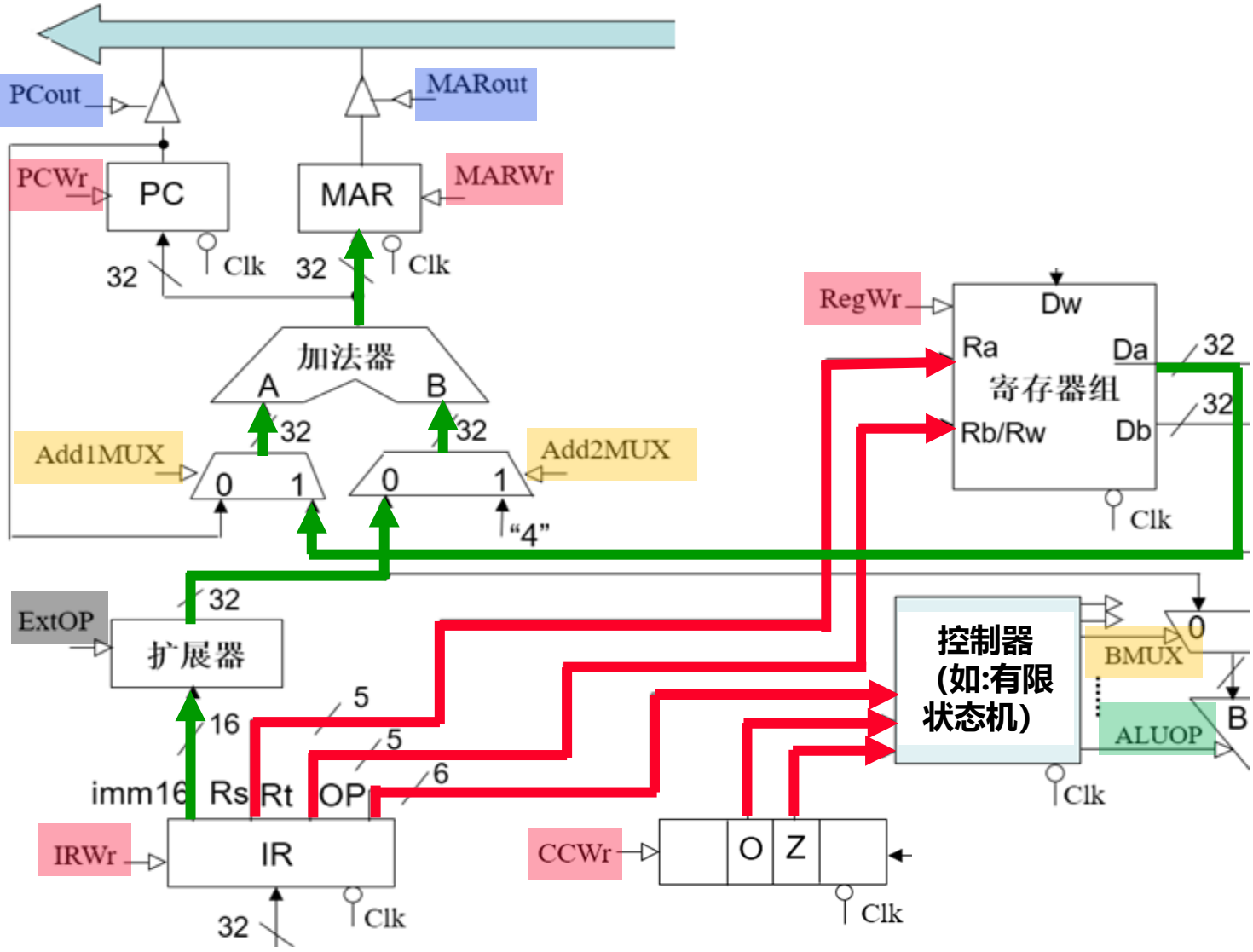
°当前时钟结束时，在IR和PC的输入端有新值，在IRWr和PCwr的控制下，下个时钟到来后的clk-to-Q时可用！

各类指令执行过程分析 （2）译码并取数（公共操作）记为 Rfetch/ID

°IR的OP及CC送控制器译码，并根据Rs和Rt读取寄存器中数据

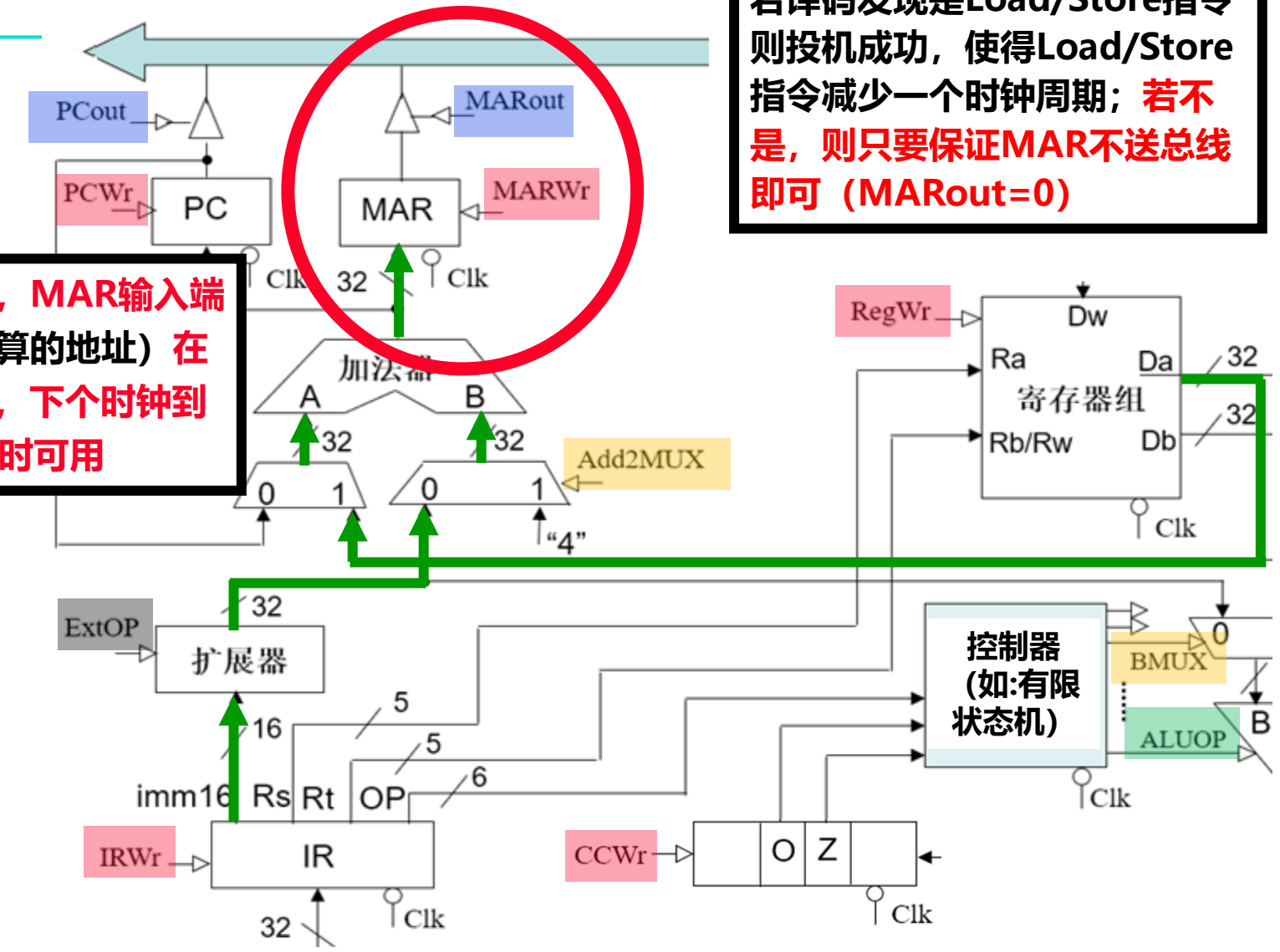
°加法器空闲，投机计算主存地址

有效控制信号及其取值为：
ExtOP=1(符扩)、
Add1MUX=1、
Add2MUX=0、
MARWr=1、
其他写使能信号全部为0



若译码发现是Load/Store指令则投机成功，使得Load/Store指令减少一个时钟周期；若不是，则只要保证MAR不送总线即可（MARout=0）

当前时钟结束时，MAR输入端有新值（投机计算的地址）在MARWr控制下，下个时钟到来后的clk-to-Q时可用



下个时钟开始，在控制器生成的控制信号控制下执行指令。

各类指令执行过程分析

（3）R-型指令的执行，需要两个时钟周期，记为 RExec、RFinish状态

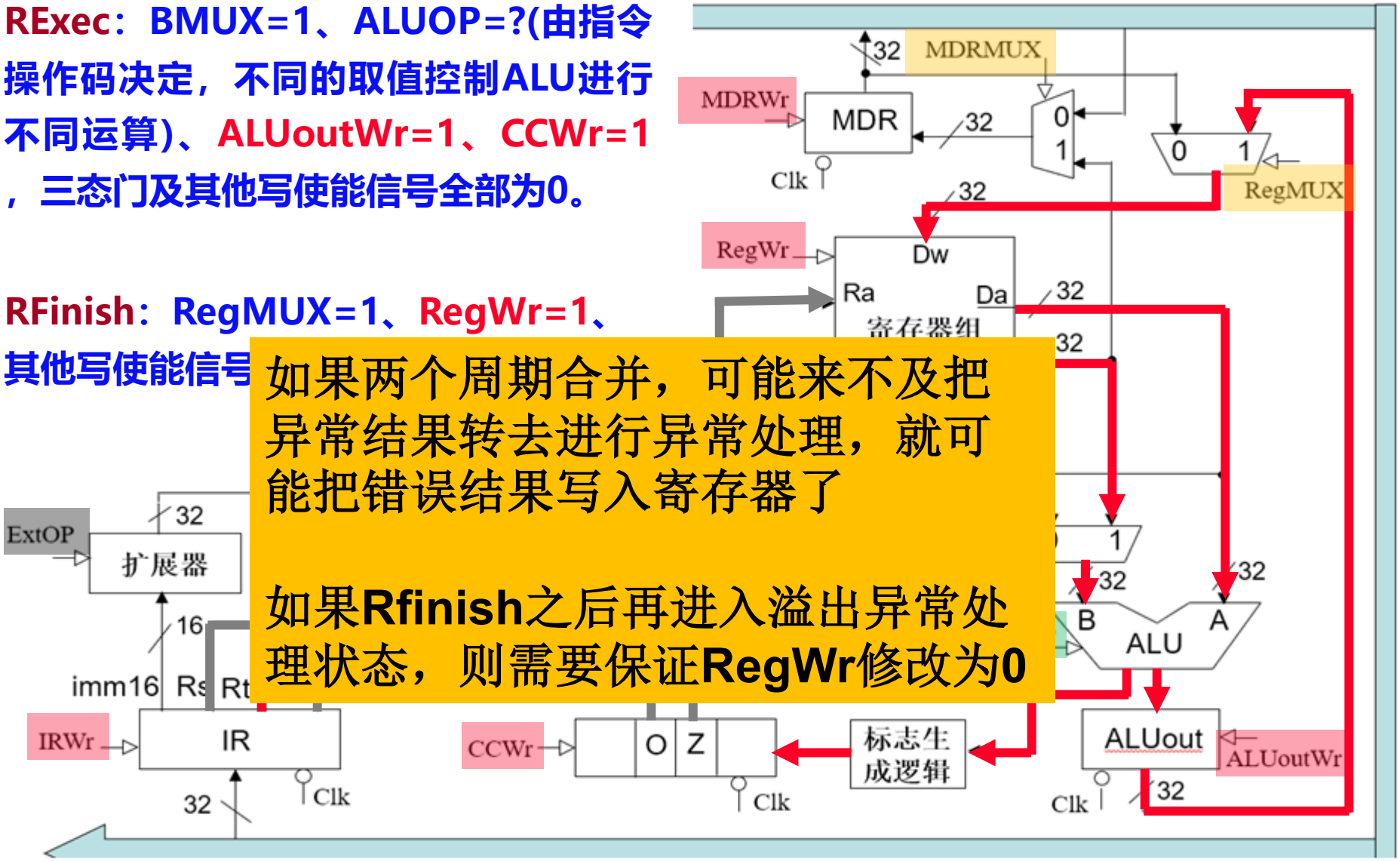
进行ALU运算并将结果存入ALUout，再将相应结果分别写入Rt和CC寄存器。

RExec：BMUX=1、ALUOP=?(由指令操作码决定，不同的取值控制ALU进行不同运算)、ALUoutWr=1、CCWr=1，三态门及其他写使能信号全部为0。

RFinish：RegMUX=1、RegWr=1、其他写使能信号

如果两个周期合并，可能来不及把异常结果转去进行异常处理，就可能把错误结果写入寄存器了

如果Rfinish之后再进入溢出异常处理状态，则需要保证RegWr修改为0



各类指令执行过程分析

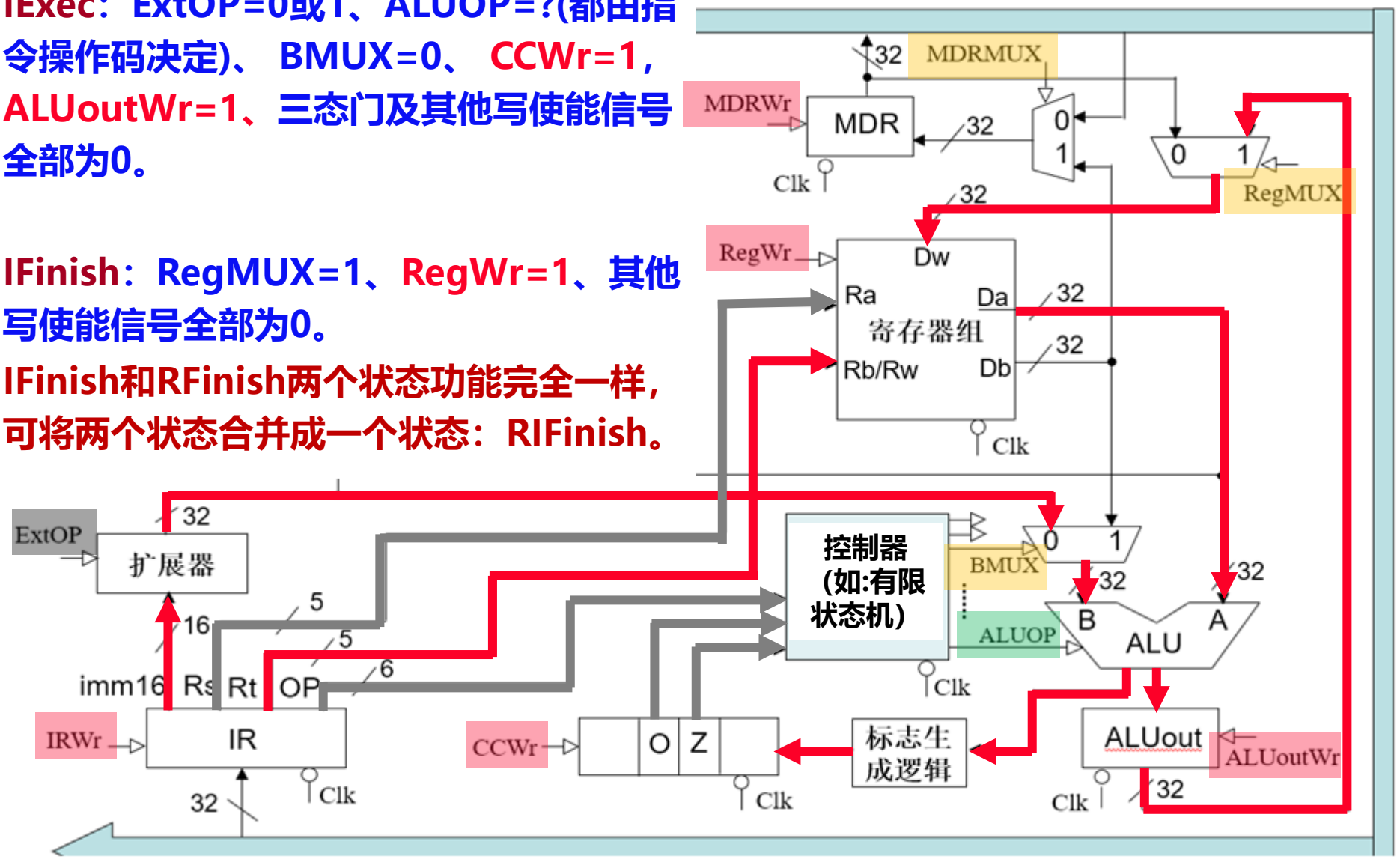
（4）I-型运算指令的执行，需要两个时钟周期，记为 IExec、IFinish状态

进行ALU运算并将结果存入ALUout，再将相应结果分别写入Rt和CC寄存器。

IExec：ExtOP=0或1、ALUOP=?(都由指令操作码决定)、 BMUX=0、 CCWr=1，ALUoutWr=1、三态门及其他写使能信号全部为0。

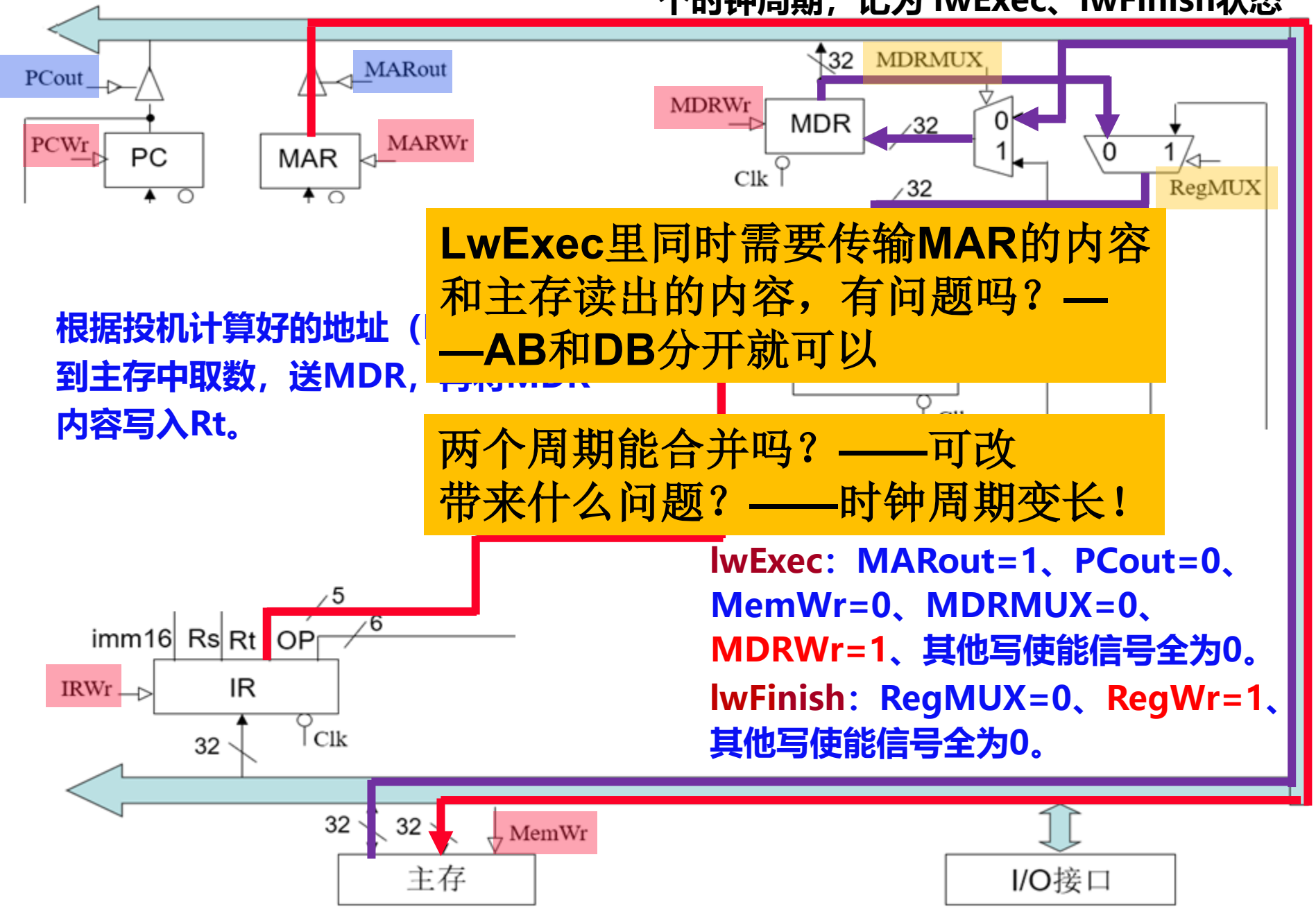
IFinish：RegMUX=1、RegWr=1、其他写使能信号全部为0。

IFinish和RFinish两个状态功能完全一样，可将两个状态合并成一个状态：RIFinish。



各类指令执行过程分析

（5）Load指令：地址已投机计算，还需两个时钟周期，记为 lwExec、lwFinish状态



根据投机计算好的地址（
到主存中取数，送MDR，
内容写入Rt。

LwExec里同时需要传输MAR的内容和主存读出的内容，有问题吗？——AB和DB分开就可以

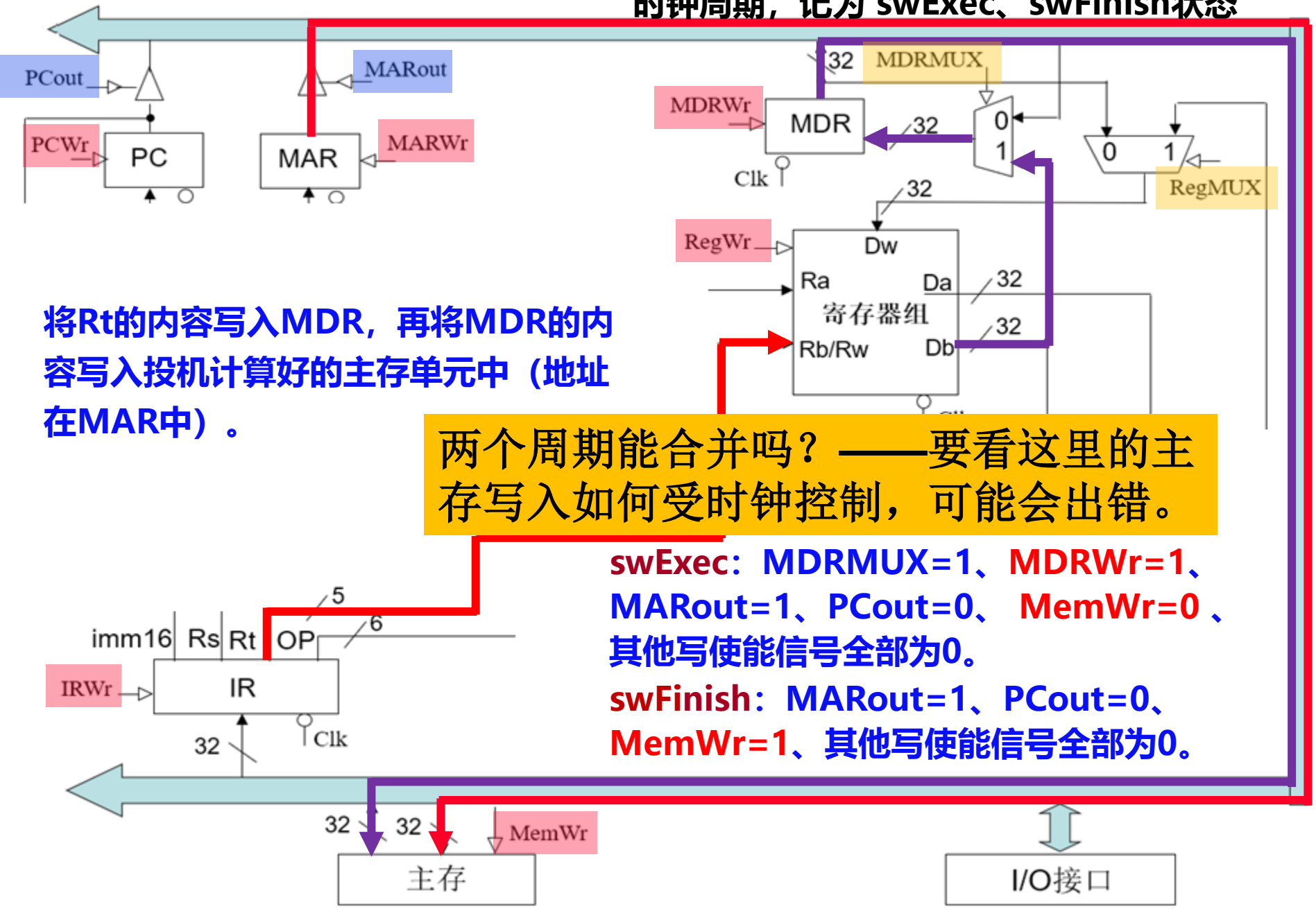
两个周期能合并吗？——可改
带来什么问题？——时钟周期变长！

lwExec：MARout=1、PCout=0、MemWr=0、MDRMUX=0、MDRWr=1、其他写使能信号全为0。

lwFinish：RegMUX=0、RegWr=1、其他写使能信号全为0。

各类指令执行过程分析

（6）Store指令：地址已投机计算，还需两个时钟周期，记为 swExec、swFinish状态



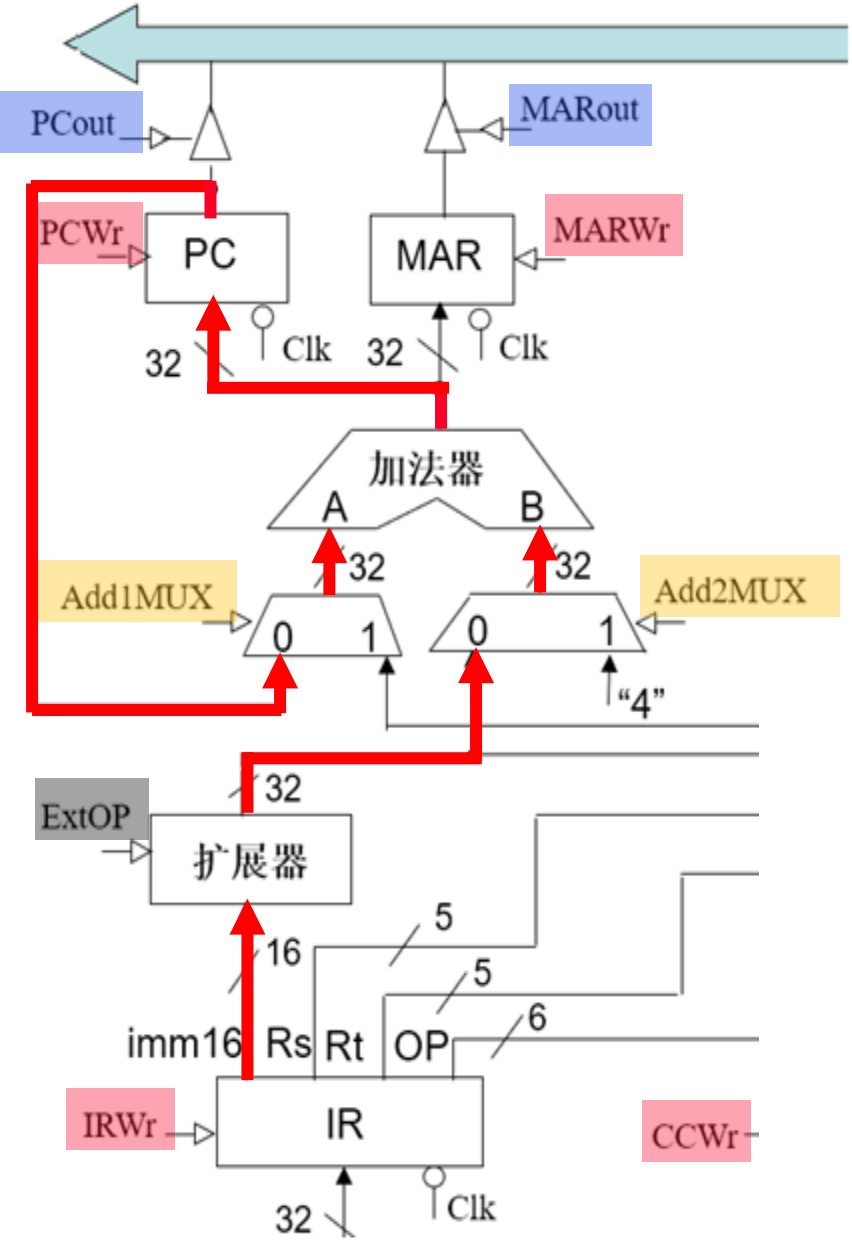
将Rt的内容写入MDR，再将MDR的内容写入投机计算好的主存单元中（地址在MAR中）。

两个周期能合并吗？——要看这里的主存写入如何受时钟控制，可能会出错。

swExec：MDRMUX=1、MDRWr=1、MARout=1、PCout=0、 MemWr=0 、其他写使能信号全部为0。

swFinish：MARout=1、PCout=0、MemWr=1、其他写使能信号全部为0。

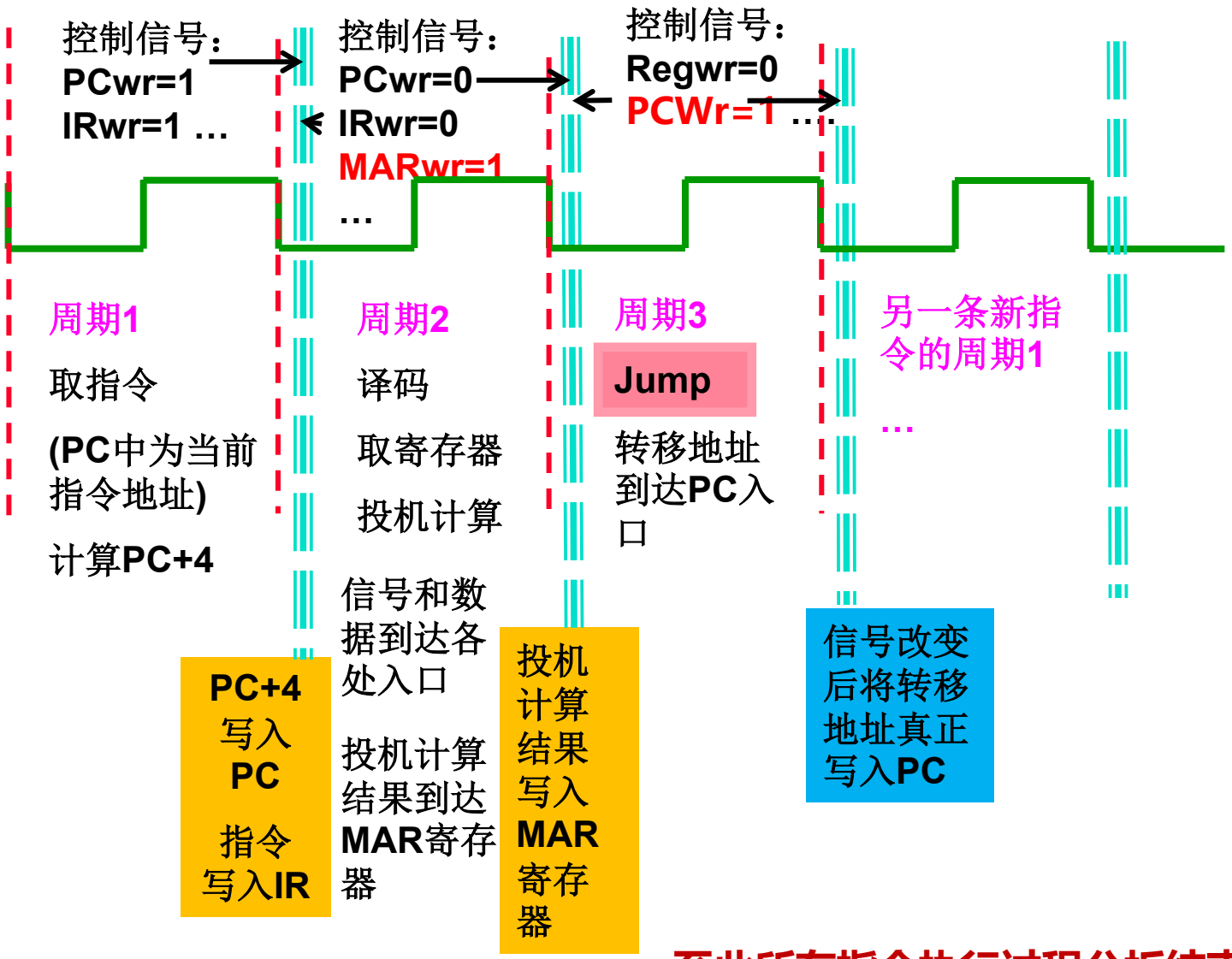
各类指令执行过程分析



(7) Jump指令：计算转移目标地址(PC+SEXT（imm6）)并送PC，需一个时钟周期，记为 JFinish状态

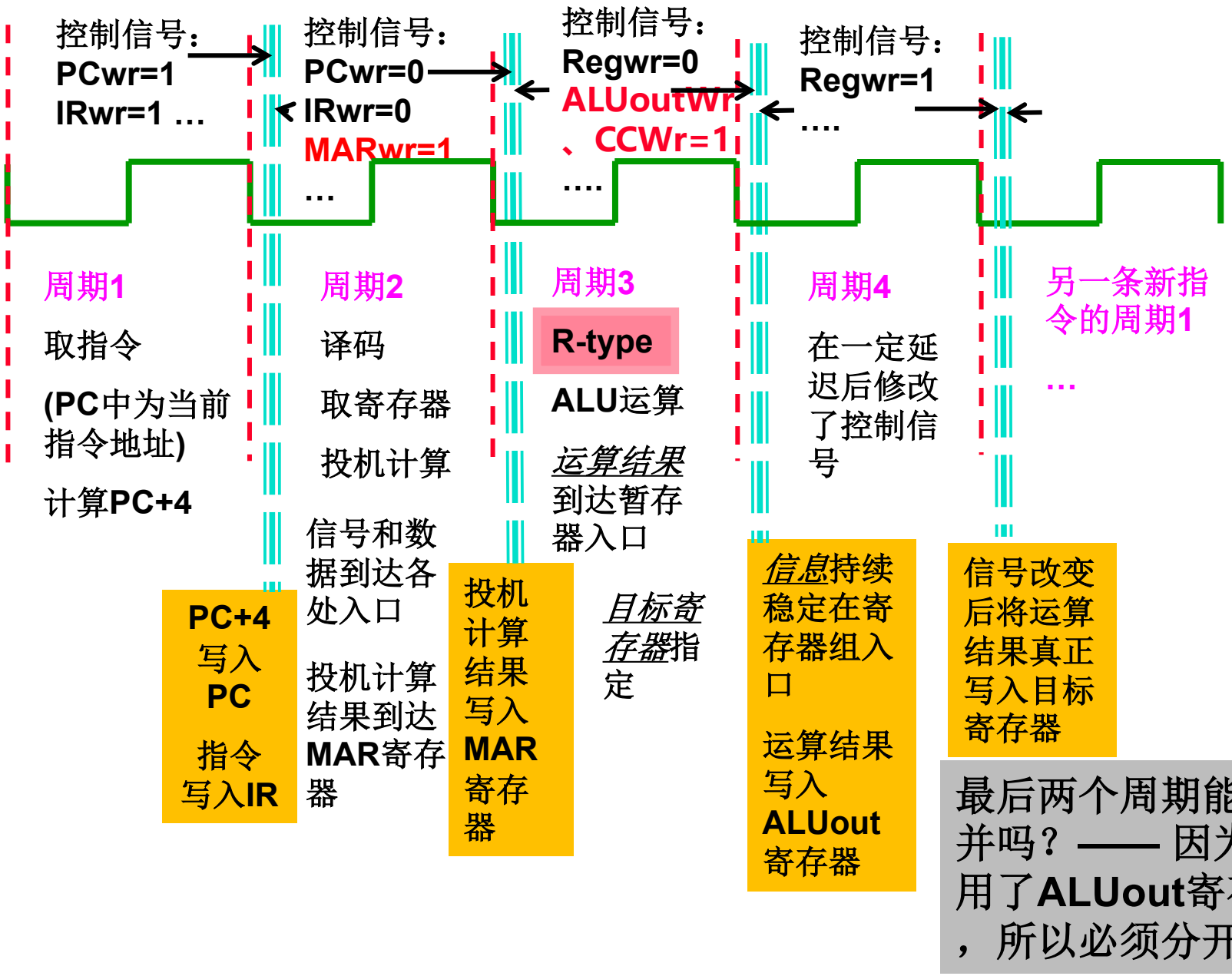
JFinish：ExtOP=1、Add1MUX=0、Add2MUX=0、PCWr=1、其他写使能信号全部为0。

状态元件内容和控制信号取值的改变：在时钟到来后的一定延时后发生



至此所有指令执行过程分析结束，下面设计控制部件，以支持能够完成上述指令功能！

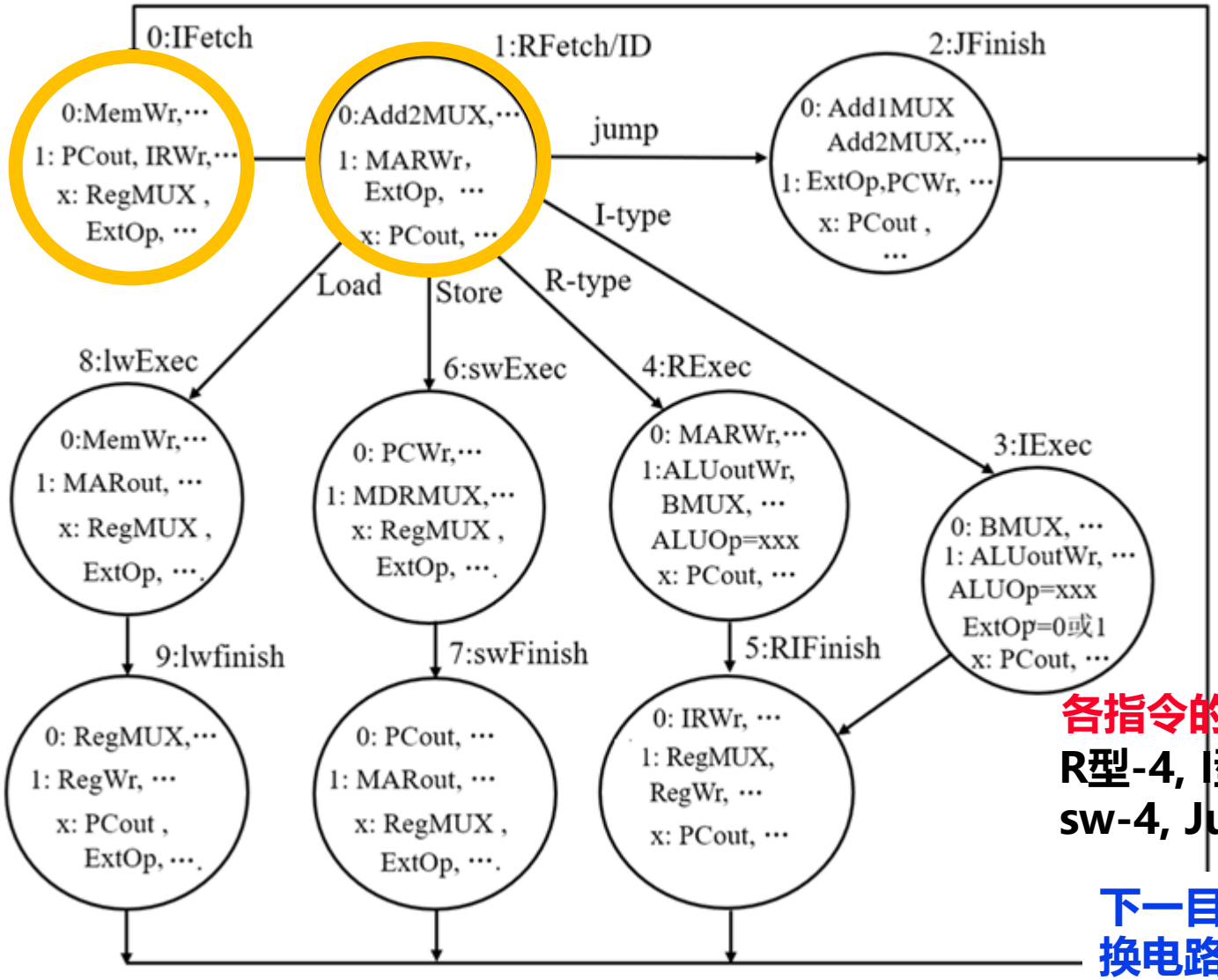
状态元件内容和控制信号取值的改变：在时钟到来后的一定延时后发生（R型指令，自行了解）



状态转换图

综合前面各指令执行过程，得到状态转换图

状态0和1是公共操作，状态1后译码得到不同的指令



每来一个时钟，进入下一个状态

各指令的时钟数多少？

R型-4, I型运算-4, lw-4, sw-4, Jump-3

下一目标：设计状态转换电路，即：控制器

多周期控制器的实现

回忆单周期控制器的实现：控制信号在整个指令执行过程中不变，用真值表能反映指令和控制信号的关系。根据真值表就能实现控制器！

多周期控制器能不能这样做?

——每个指令有多个周期

——每个周期控制信号取值不同！

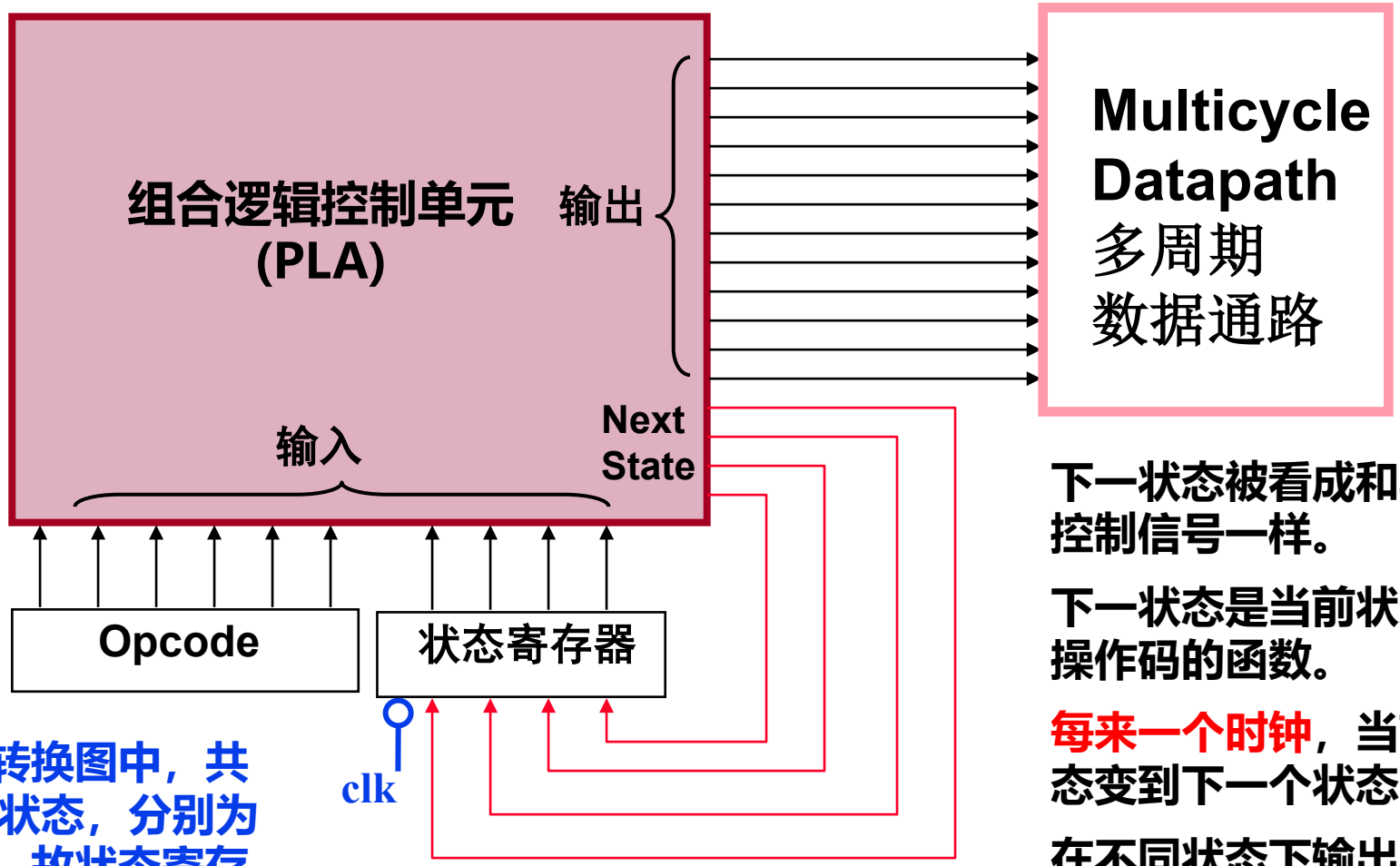
多周期控制器功能描述方式：

- **有限状态机：用硬连线路(PLA)实现**
- **微程序：用ROM存放微程序实现**

有限状态机（PLA）控制方式

由时钟、当前状态和操作码确定下一状态。不同状态输出不同控制信号值

控制逻辑采用“摩尔机”方式，即：输出函数仅依赖于当前状态



状态转换图中，共10个状态，分别为0~9，故状态寄存器至少需4位

下一状态被看成和其他控制信号一样。

下一状态是当前状态和操作码的函数。

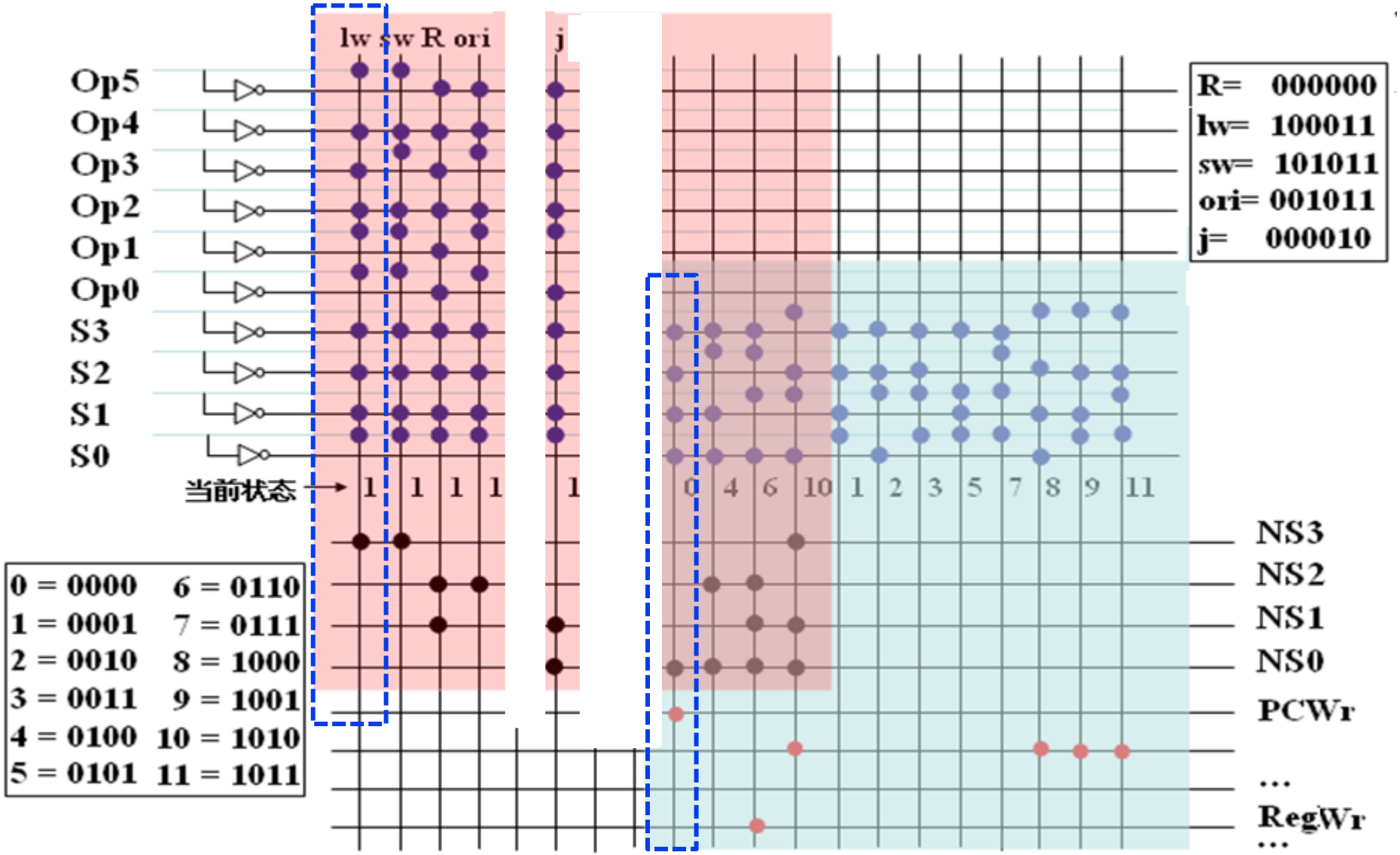
每来一个时钟，当前状态变到下一个状态

在不同状态下输出不同的控制信号。

状态转换表——最终由PLA电路来实现

当前状态 $S_3S_2S_1S_0$	指令操作码 $OP_5OP_4OP_3OP_2OP_1OP_0$	下一状态 $NS_3NS_2NS_1NS_0$
State2、5、7、9		0 0 0 0
State0 (IFetch)		0 0 0 1
State1 (RFetch/ID)	000010 (jump)	0 0 1 0
	001101 (ori)	0 0 1 1
	000000 (R-type)	0 1 0 0
State3 (IExec)		0 1 0 1
State4 (RExec)		0 1 0 1
State1 (RFetch/ID)	101011 (sw)	0 1 1 0
	100011 (lw)	1 0 0 0
State6 (swExec)		0 1 1 1
State8 (lwExec)		1 0 0 1

示意：用PLA电路实现控制单元（硬布线方式）



左上角：由操作码和当前状态确定下一状态的电路

右下角：由当前状态确定控制信号的电路

硬连线控制器的特点：

优点：速度快，适合于简单或规整的指令系统

缺点：它是一个多输入/多输出的巨大逻辑网络。对于复杂指令系统来说，结构庞杂，实现困难；修改、维护不易；灵活性差。甚至无法用有限状态机描述！

简化控制器设计的一个方法：微程序设计

微程序控制器设计

微程序控制器的基本思想：

- **仿照程序设计的方法，编制每个指令对应的微程序**
  - **每个微程序由若干条微指令构成，分别和各状态对应**
  - **每条微指令包含若干条微命令，分别和状态中的控制信号对应**

- **所有微程序放在只读存储器中（称为控制存储器Control Storage，简称控存CS ），都是0/1序列**

微程序设计的特点：具有规整性、可维性和灵活性，但速度慢。

微程序\微指令\微命令\微操作的关系

执行某条指令时

- 从CS中取出对应微程序
- 执行微程序，就是执行其中的各条微指令
- 对微指令译码就是产生对应的微命令——控制信号
- 由微命令控制数据通路的执行

控制程序执行要解决什么问题?

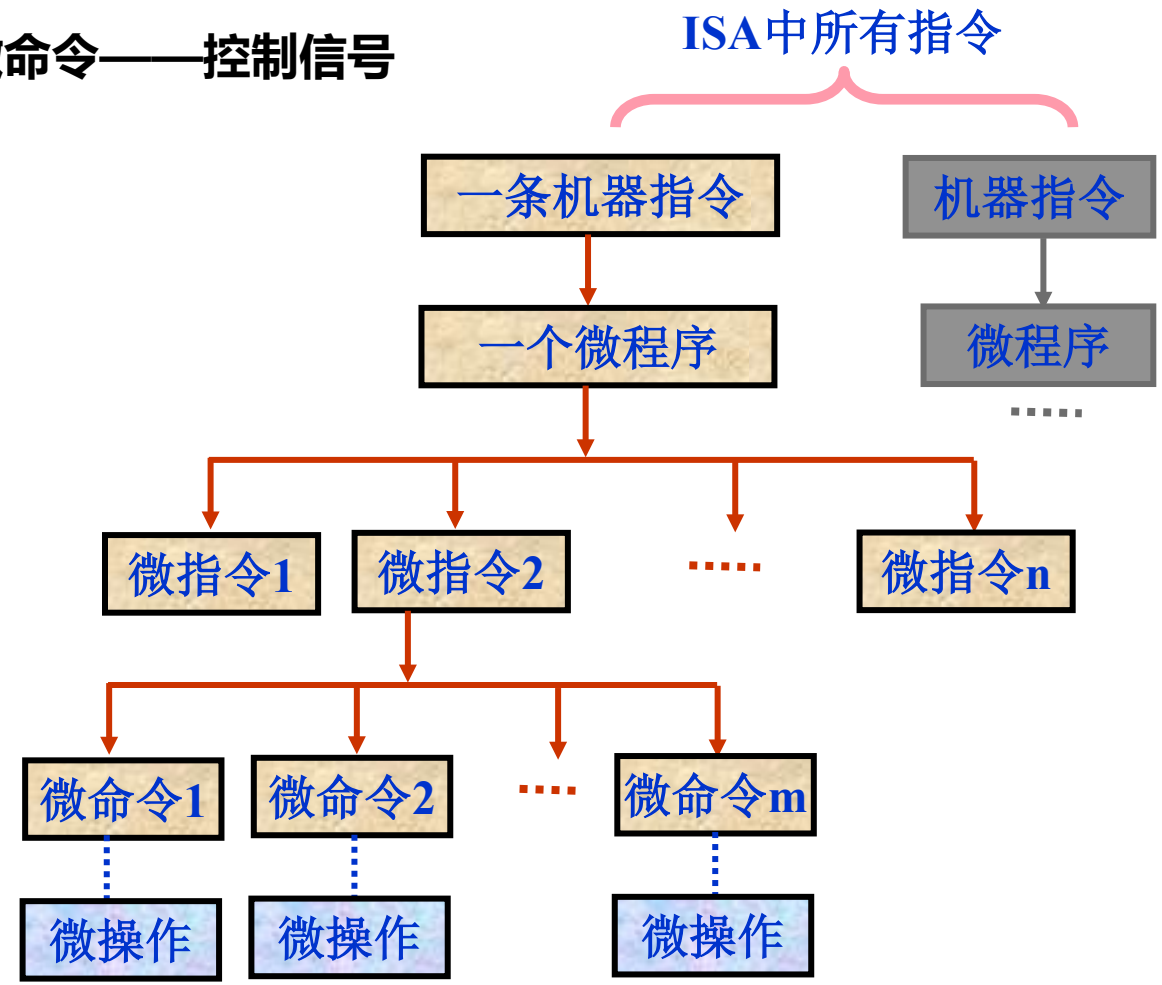
(1) 指令如何译码、执行

(2) 下条指令到哪里去取

微程序执行也要解决两个问题:

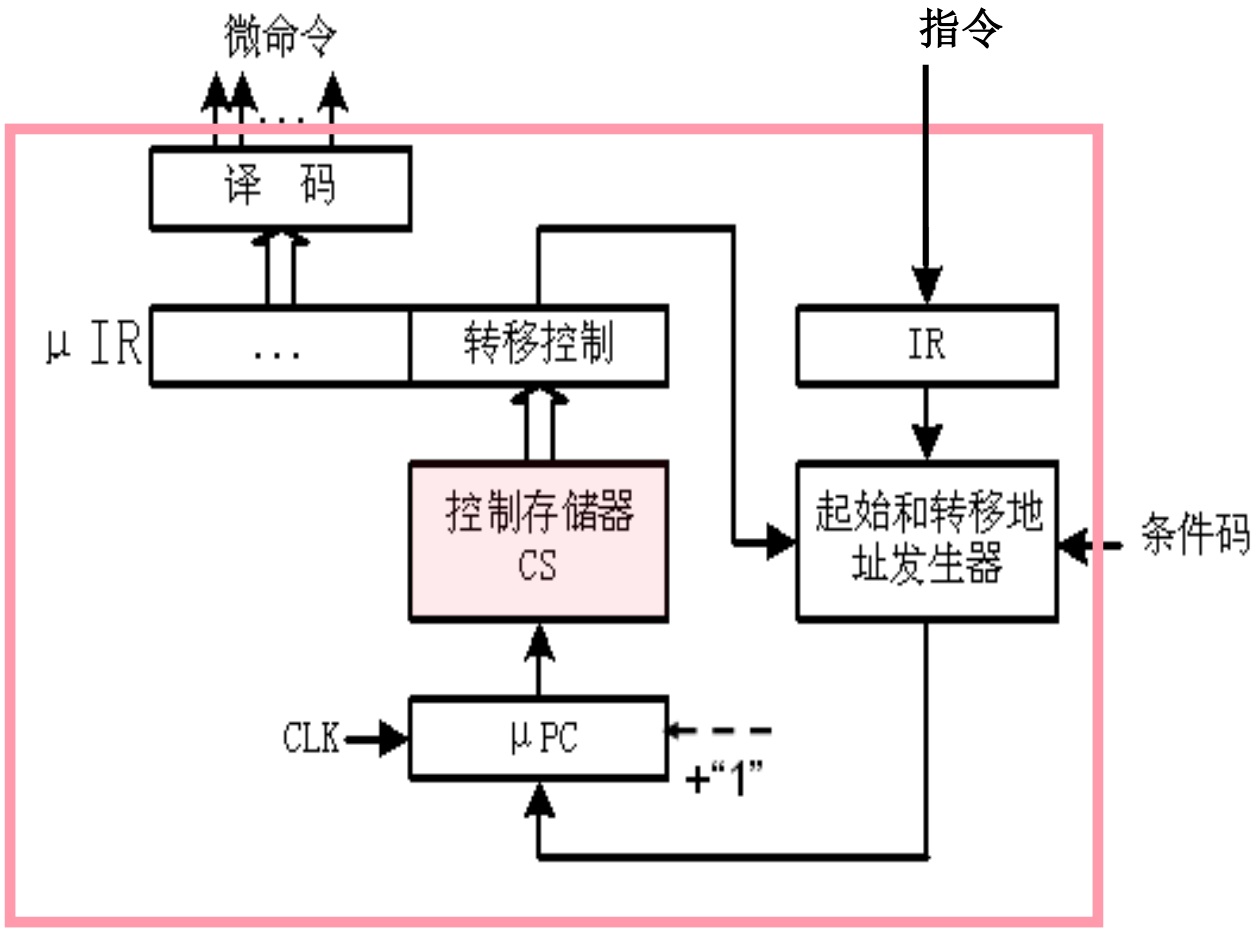
(1)微指令如何对微命令编码

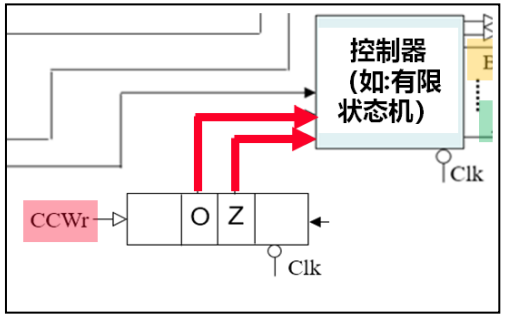
(2)下条微指令在哪里



微程序控制器的基本结构

- 输入：指令、条件码
- 输出：控制信号(微命令)
- 核心：控存CS
- μPC：指出将要执行的微指令在CS中的位置
- μIR： 正在执行的微指令
- 每个时钟执行一条微指令
- 微程序第一条微指令地址由起始地址发生器产生
- 顺序执行时， μPC+1
- 转移执行时，由转移控制字段指出对哪些条件码进行测试，转移地址发生器根据条件码修改μPC





第一个问题：微指令格式的设计

微指令中包含了：若干微命令、下条微指令地址（可选）、常数（可选）

微指令格式：

μOP	μAddr	常数

μOP: 微操作码字段，产生微命令（控制信号）；

μAddr（配合常数）：微地址码字段，产生下条微指令地址。

- 微指令格式设计风格取决于微操作码的编码方式
- 微操作码编码方式：

不译法（直接控制法）

字段直接编码（译）法

字段间接编码（译）法

} 水平型微指令风格 （微指令长，微程序短）

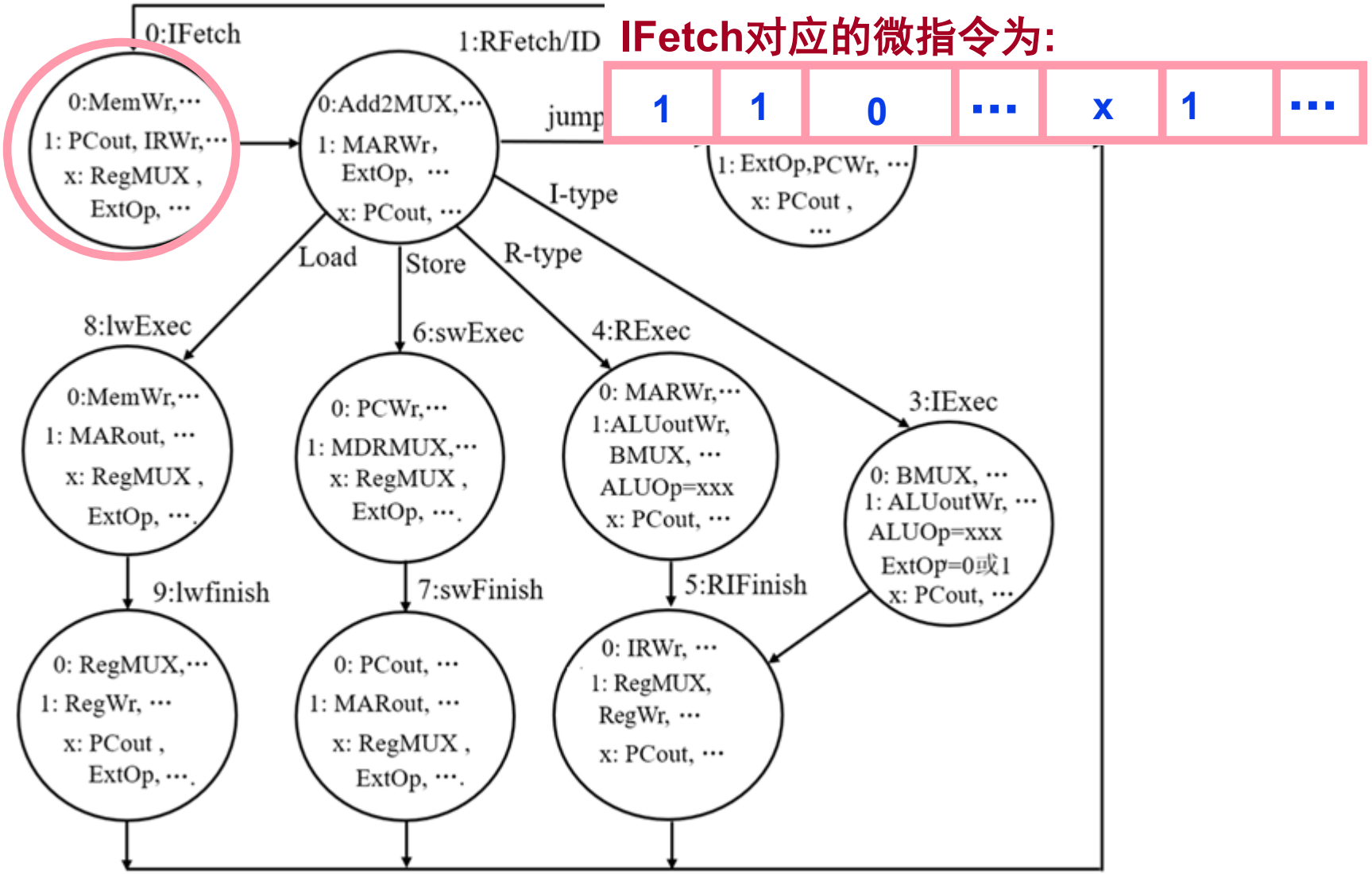
最小（最短、垂直）编码（译）法 —— 垂直型微指令风格（微指令短，微程序长）

采用不译法的微操作码格式为：

PCWr	IRWr	MemWr	……	ExtOp	PCout	……

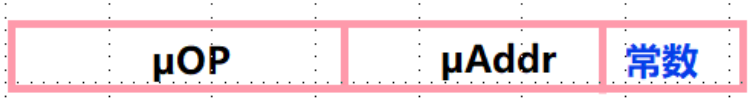
IFetch对应的微指令为:

1	1	0	…	x	1	…



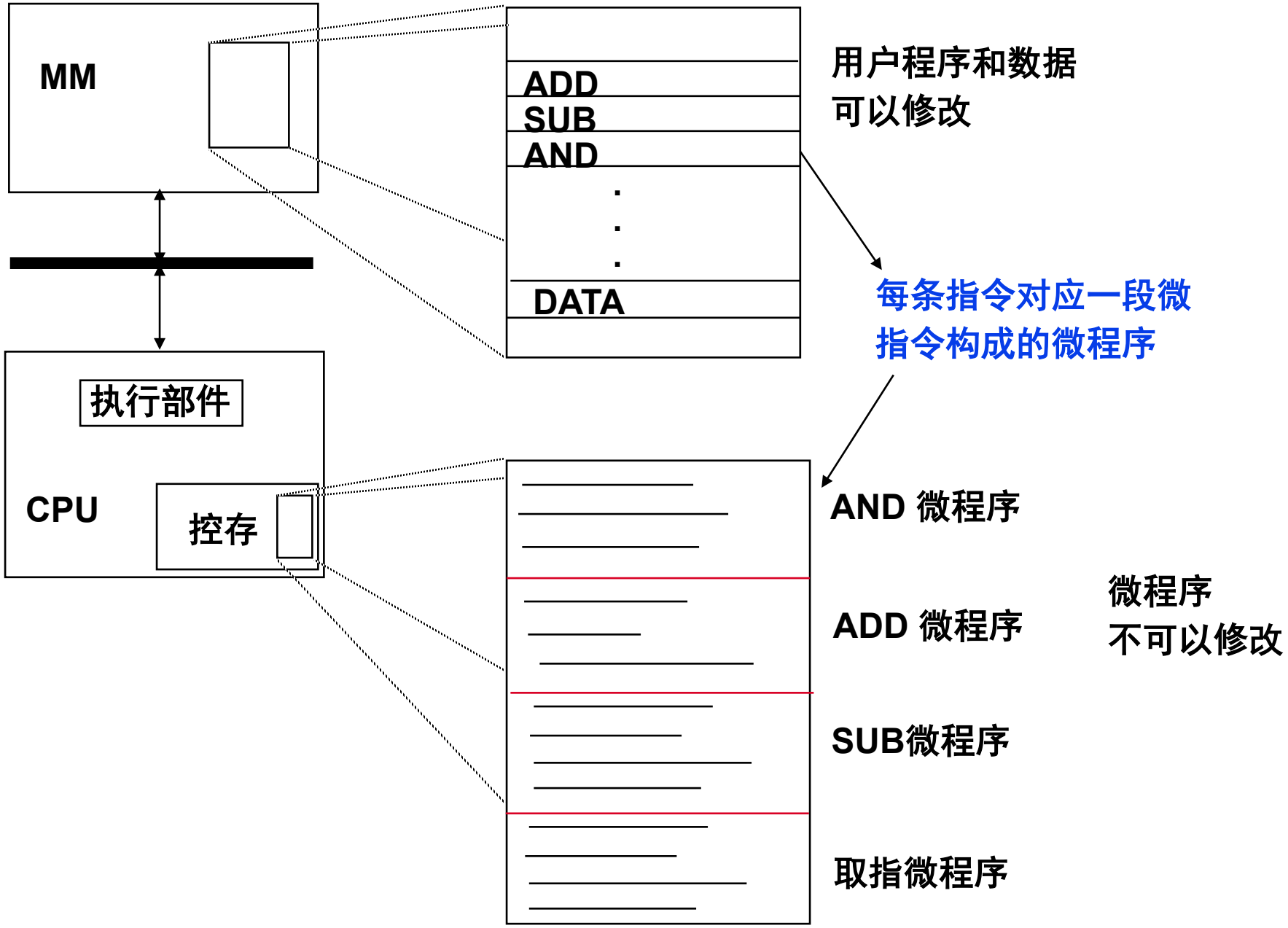
第二个问题：下条微地址的确定方式

- 什么是微程序执行顺序的控制问题?
  - 指在现行微指令执行完毕后，怎样控制产生下一条微指令的地址。
- 怎样控制微程序的执行顺序?



  - 通过在本条微指令中明显或隐含地指定下条微指令在控存中的地址来控制。
- 微指令地址的产生方法有两种：　问题：指令是隐含还是明显指出下条指令地址的?
  - 增量(计数器)法：下条微指令地址隐含在微程序计数器μPC中。
  - 断定(下址字段)法：在本条微指令中明显指定下条微指令地址。
- 选择下条要执行的微指令有以下四种情况：
  - 取指微程序首址：每条指令执行前，CPU先执行取指微程序。
  - 第一条微指令：每条指令取出后，必须转移到该指令对应的第一条微指令执行。
  - 顺序执行时：微程序执行过程中顺序取出下条微指令执行。
  - 分支执行时：在遇到按条件转移到不同微指令执行时，需要根据控制单元的输入来选择下条微指令。

微指令字的解释执行


MM
ADD
SUB
AND
DATA
用户程序和数据
可以修改
每条指令对应一段微
指令构成的微程序
执行部件
CPU
控存
AND 微程序
ADD 微程序
SUB微程序
取指微程序
微程序
不可以修改

回顾：异常和中断的处理

- 程序执行过程中CPU会遇到一些特殊情况，使正在执行的程序被“中断”
  - 此时，CPU中止原来正在执行的程序，转到处理异常情况或特殊事件的程序去执行，执行后再返回到原被中止的程序处（断点）继续执行
- 程序执行被 “中断” 的事件有两类
  - 内部“异常”：在CPU内部发生的意外事件或特殊事件，发现立刻处理

    按发生原因分为硬故障中断和程序性中断两类

    硬故障中断：如电源掉电、硬件线路故障等

    程序性中断：执行某条指令时发生的“例外(Exception)”，如溢出、缺页、越界、越权、非法指令、除数为0、堆栈溢出、访问超时、断点设置、单步、系统调用等
  - 外部“中断”：在CPU外部发生的特殊事件，通过“中断请求”信号向CPU请求处理。如实时钟、控制台、打印机缺纸、外设准备好、采样计时到、DMA传输结束等。——每个指令执行结束，CPU查询有没有中断请求，有则响应中断

处理器中的异常处理机制

检测到异常时，处理器必须进行以下基本处理：

① 关中断（“中断/异常允许”状态位清0）：使处理器处于“禁止中断”状态，以防止新异常(或中断)破坏断点、程序状态和现场（现场指通用寄存器的值）。

② 保护断点和程序状态：将断点和程序状态保存到堆栈或特殊寄存

PC→栈 或 专门存放断点的寄存器。

PSWR →栈 或 EPSWR （专门保存程序状态的寄存器）

③ 识别异常事件：

软件识别

硬件识别（向量中断方式）

识别异常事件：软件识别和硬件识别

（1）软件识别（RISC-V、MIPS等采用）

设置一个异常原因寄存器（如RISC-V和MIPS中的Cause寄存器），用于记录异常原因。操作系统使用一个统一的异常处理程序，该程序按优先级顺序查询异常状态寄存器，识别出异常事件。例如：

RISC-V中由某个CSR（如M-模式下的mtvec寄存器）所定义的地址作为异常处理程序的首地址；MIPS中位于内核地址0x8000 0180处有一个专门的异常处理程序。

该异常处理程序会检测Cause寄存器以查明异常的具体原因，然后转到内核中相应的异常处理程序段中进行具体的处理。

（2）硬件识别（向量中断）（80x86采用）

每个异常和中断都有一个异常/中断号，根据此号，到中断向量表（中断描述符表）中读取对应的具体的中断服务程序的入口地址。

RISC-V架构也支持通过硬件识别方式来识别外部中断源。

异常和中断机制是处理器设计中最具挑战性的任务之一

- RISC-V架构定义了一些控制和状态寄存器（Control and Status Register，CSR），用于配置或记录一些运行的状态。
- CSR寄存器是处理器核内部的寄存器，使用自己的地址编码空间，和存储器寻址的地址区间完全无关系。
- CSR寄存器的访问采用专用的CSR指令，包括CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI以及CSRRCI指令。

Register	Address
mstatus	0x300
misa	0x301
medeleg	0x302
mideleg	0x303
mie	0x304
mtvec	0x305
mcounterer	0x306
mverndorid	0xf11
marchid	0xf12
mimpid	0xf13
mhartid	0xf14
mscratch	0x340
mepc	0x341
mcause	0x342
mtval	0x343
mip	0x344
mcycle	0xb00
mcycleh	0xb80
minstret	0xb02
minstreth	0xb82

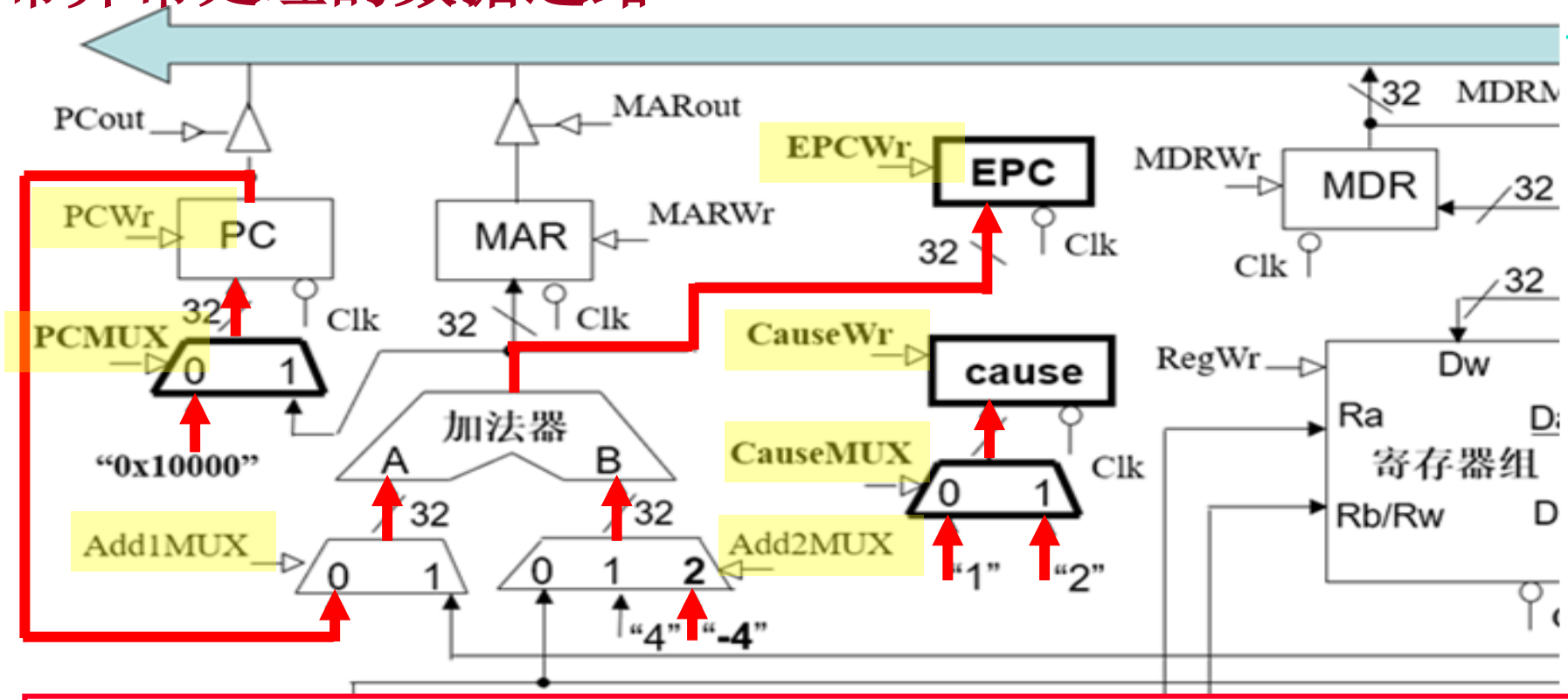
带异常处理的数据通路设计

以前述多周期处理器为例，简要说明带异常处理数据通路设计

- 假定其异常/中断机制中包括以下两个寄存器：
  - EPC：32位，用于存放断点（异常处理后返回到的指令的地址）。
    - 写入EPC的断点可能是正在执行的指令的地址（故障时）
    - 也可能是下条指令的地址（自陷和中断时）
    - 前者需要把PC的值减4后送到EPC，后者则直接送PC到EPC！
  - Cause：32位（有些位还没有用到），记录异常原因。
    - 假定处理的异常类型有以下两种：

      未定义指令（Cause=1）、溢出（Cause=2）
- 需要加入两个寄存器的“写使能”控制信号
  - EPCWr：在保存断点时该信号有效，使断点PC写入EPC。
  - CauseWr：在处理器发现异常（如：非法指令、溢出）时，该信号有效，使异常类型被写到Cause寄存器。
- 需要一个控制信号CauseMUX来选择正确的值写入到Cause中
- 需要将异常查询程序的入口地址（假设为0x10000）写入PC，可以在PC输入端增加一个MUX（控制信号PCMUX），其中一个输入为0x10000

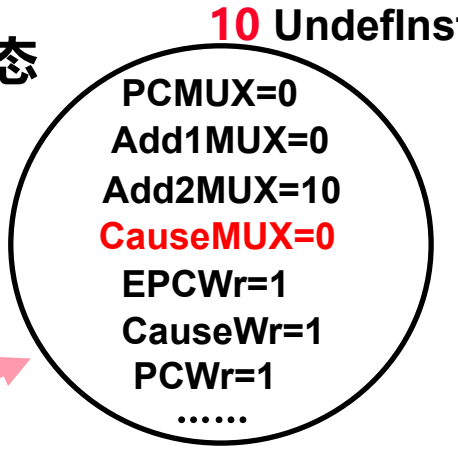
带异常处理的数据通路



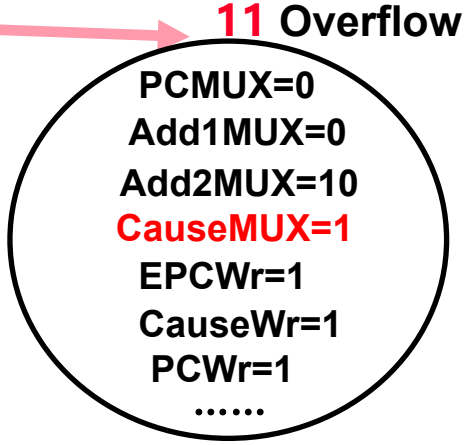
- 假设只要处理两种故障类异常：未定义指令（Cause=1）、溢出（Cause=2）
- 故障类异常的断点为：PC减4。因为状态0（IFetch）时，已执行PC加4
- Cause寄存器的两个输入端，分别是 1（未定义指令时）和 2（溢出时）
- PC中将设置异常处理程序的首地址（假设为0x10000）
- 若发生异常，则Add1MUX=0， Add2MUX=10，EPCWr=1, CauseWr=1, PCMUX=0, PCWr=1，其他写使能信号都为0，
- 发生未定义指令时CauseMUX=0，发生溢出时CauseMUX=1。

带异常处理的控制器设计

- 在有限状态机中增加异常处理的状态，每种异常占一个状态
- 每个异常处理状态中，需考虑以下基本控制
  - Cause寄存器的设置
  - 计算断点处的PC值（PC-4），并送EPC
  - 将异常查询程序的入口地址送PC
- 假设要控制的数据通路中有以下两种异常处理
  - 非法操作码（Cause=0）：状态10
  - 溢出（Cause=1）：状态11
- 在原来状态转换图基础上加入两个异常处理状态
  - 如何检测是否发生了这两种异常?
    - 未定义指令（非法操作码）：当指令译码器发现op字段是一个未定义的编码时
    - 溢出：当R-型或I-型运算类指令在ALU中执行后，在条件码寄存器CC中的标志O为1时

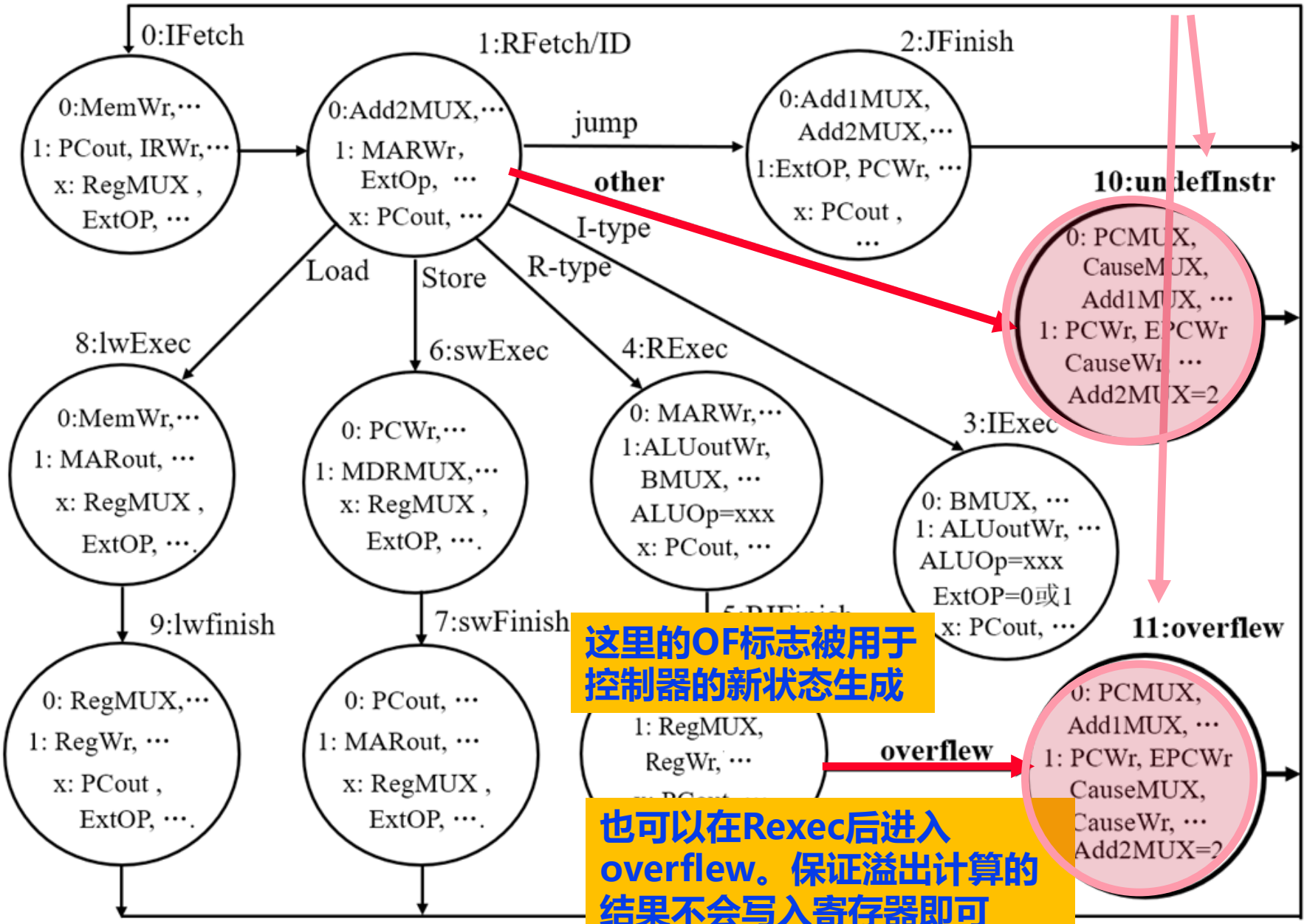


10 未定义指令异常状态



11 溢出异常状态

加入异常处理后的有限状态转换图



单周期和多周期的CPU比较

- 成本比较：
  - 单周期下功能部件不能重复使用；而多周期下可重复使用，比单周期省
  - 单周期指令执行结果直接保存在PC、Regfile和Memory；而多周期下需加一些临时寄存器保存中间结果，比单周期费
- 性能比较：
  - 单周期CPU的CPI为1，但时钟周期为最长的load指令执行时间
  - 多周期CPU的CPI是多少？时钟周期多长？

    假定程序中22%为Load，11%为Store，49%为R-Type，16%为I-Type，2%为Jump。每个状态需要一个时钟周期，CPI为多少？

    若每种指令所需的时钟周期数为：

    Load：4；Store：4；R-Type：4；I-Type：4；Jump：3

    则CPI计算如下：

    CPI=CPU时钟周期数/指令数 = Σ（指令数 x CPI）/指令数

    但是，如果多周期的时钟周期是单周期的1/3

    那么多周期总体时间就是3.98/3=1.33 > 1

    所以，多周期的性能不一定比单周期好！

    关键就在于划分阶段是否均匀。

    x1=1

    这

前四讲总结1

- CPU的主要功能
  - 周而复始执行指令
  - 执行指令过程中，若发现异常情况，则转异常处理
  - 每个指令结束，查询有没有中断请求，有则响应中断
- CPU的内部结构
  - 由数据通路(Datapath)和控制单元(Control unit)组成
    - 数据通路中包含组合逻辑单元和存储信息的状态单元
      - 组合逻辑单元用于对数据进行处理，如：加法器、运算器ALU、扩展器（0扩展或符号扩展）、多路选择器、以及状态单元的读操作线路等。
      - 状态单元包括触发器、寄存器、寄存器堆、数据/指令存储器等，用于对指令执行的中间状态或最终结果进行保存。
    - 控制单元对指令进行译码，与指令执行得到的条件码或当前机器的状态、时序信号（时钟）等组合，生成对数据通路进行控制的控制信号

前四讲总结2

- CPU中的寄存器
  - 用户可见寄存器（用户可使用）
    - 通用寄存器：用来存放地址或数据，需在指令中明显给出
    - 专用寄存器：用来存放特定的地址或数据，无需在指令中明显给出
    - 数据寄存器：专用于保存数据，可以是通用或专用寄存器
    - 地址寄存器：专用于保存地址，可以是通用或专用寄存器。如：段指针、变址器、基址器、堆栈指针、栈帧指针等。
    - 标志(条件码)寄存器、程序计数器PC：部分可见。由CPU根据指令执行结果设定，只能以隐含方式读出其中若干位，用户程序（非内核程序）不能改变
  - 控制和状态寄存器（用户不可使用）
    - 指令寄存器IR
    - 存储器地址寄存器MAR
    - 存储器缓冲(数据)寄存器 MBR / MDR
    - 程序状态字寄存器PSWR
    - 临时寄存器：用于存放指令执行过程中的临时信息
    - 其他寄存器：如，进程控制块指针、系统堆栈指针、页表指针等

前四讲总结3

- 指令执行过程
  - 取指、译码、取数、运算、存结果、查中断
  - 指令周期：取出并执行一条指令的时间，由若干个时钟周期组成
  - 时钟周期：CPU中用于信号同步的信号，是CPU最小的时间单位
- 数据通路的定时方式
  - 现代计算机都采用时钟信号进行定时
  - 一旦时钟有效信号到来，数据通路中的状态单元可以开始写入信息
  - 如果状态单元每个周期都更新信息，则无需加“写使能”控制信号，否则，需加“写使能”控制信号，以使必要时控制信息写入寄存器
- 数据通路中信息的流动过程
  - 每条指令在取指令阶段和指令译码阶段都一样
  - 每条指令的功能不同，故在数据通路中所经过的部件和路径可能不同
  - 数据在数据通路中的流动过程由控制信号确定
  - 控制信号由控制器根据指令代码来生成

前四讲总结4

- 单周期处理器的设计
  - 每条指令都在一个时钟周期内完成
  - 时钟周期以最长的Load指令所花时间为准
  - 无需加临时寄存器存放指令执行的中间结果
  - 同一个功能部件不能重复使用
  - 控制信号在整个指令执行过程中不变，所以控制器设计简单，只要写出指令和控制信号之间的真值表，就可以设计出控制器
- 多周期处理器的设计
  - 每条指令分成多个阶段，每个阶段在一个时钟内完成
  - 不同指令包含的时钟个数不同
  - 阶段的划分要均衡，每个阶段只能完成一个独立、简单的功能，如：
    - 一次ALU操作
    - 一次存储器访问
    - 一次寄存器存取
  - 需加临时寄存器存放指令执行的中间结果
  - 同一个功能部件能在不同的时钟中被重复使用
  - 可用有限状态机来表示指令执行流程，并以此设计控制器

前四讲总结5

本章作业：习题3、4、5、6、11、13、14、17、18

- 控制单元实现方式
  - 有限状态机描述方式
    - 每个时钟周期包含的控制信号的值的组合看成一个状态，每来一个时钟，控制信号会有一组新的取值，也就是一个新的状态
    - 所有指令的执行过程可用一个有限状态转换图来描述
    - 用一个组合逻辑电路（一般为PLA电路）来生成控制信号，用一个状态寄存器实现状态之间的转换
    - 实现的控制器称为硬布线控制器
  - 微程序描述方式
    - 每个时钟周期所包含的控制信号的值的组合看成是一个0/1序列，每个控制信号对应一个微命令，控制信号取不同的值，就发出不同的微命令
    - 若干微命令组合成一个微指令，每条指令所包含的动作就由若干条微指令来完成，每来一个时钟，执行一条微指令
    - 每条指令对应一个微程序，执行时，先找到对应的第一条微指令，然后按照特定的顺序取出后续的微指令执行
    - 实现的控制器称为微程序控制器