Performance of Single-Cycle Machine

Arithmetic & Logical

Load

Critical Path

Store

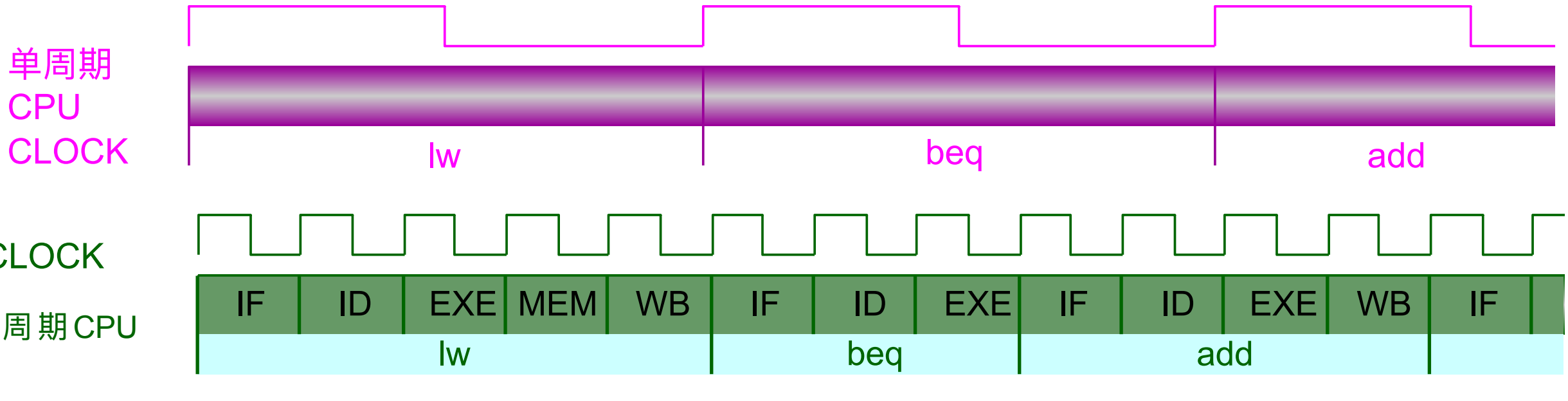
Branch

What's wrong with our CPI=1 processor?

An implementation in which every instruction operates in 1 clock cycle of a fixed length.

单周期数据通路的缺点：回顾 Load 指令执行

- 单周期处理器的CPI为1，时钟周期以最长的load指令为准，时钟周期远远大于其他指令实际所需的执行时间，效率极低



多周期处理器的实现思想

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

每步都设置存储元件，每步执行结果都在下个时钟开始保存到相应单元

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

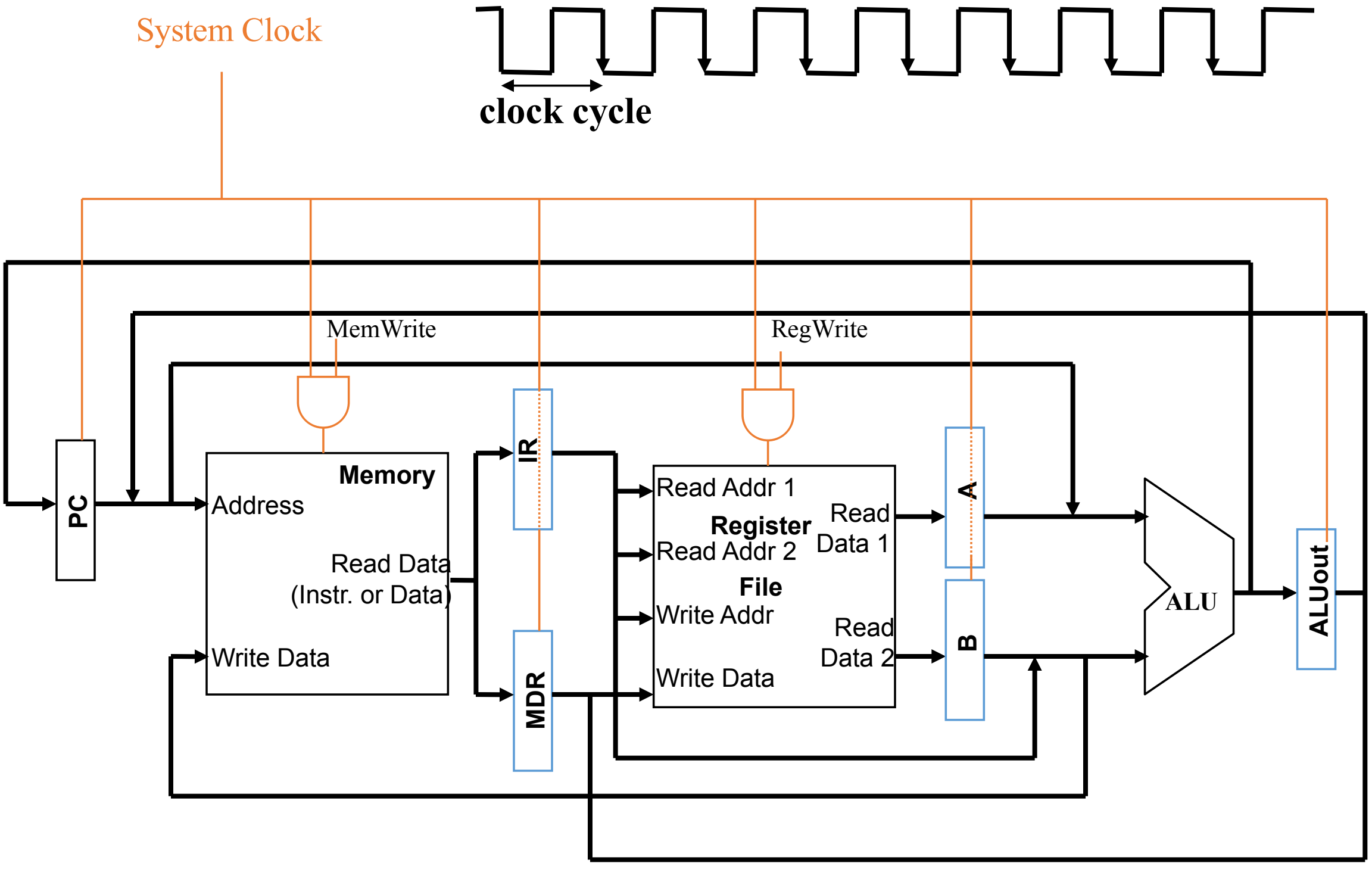
多周期处理器的实现思想

- 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，可参考：

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

- 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）

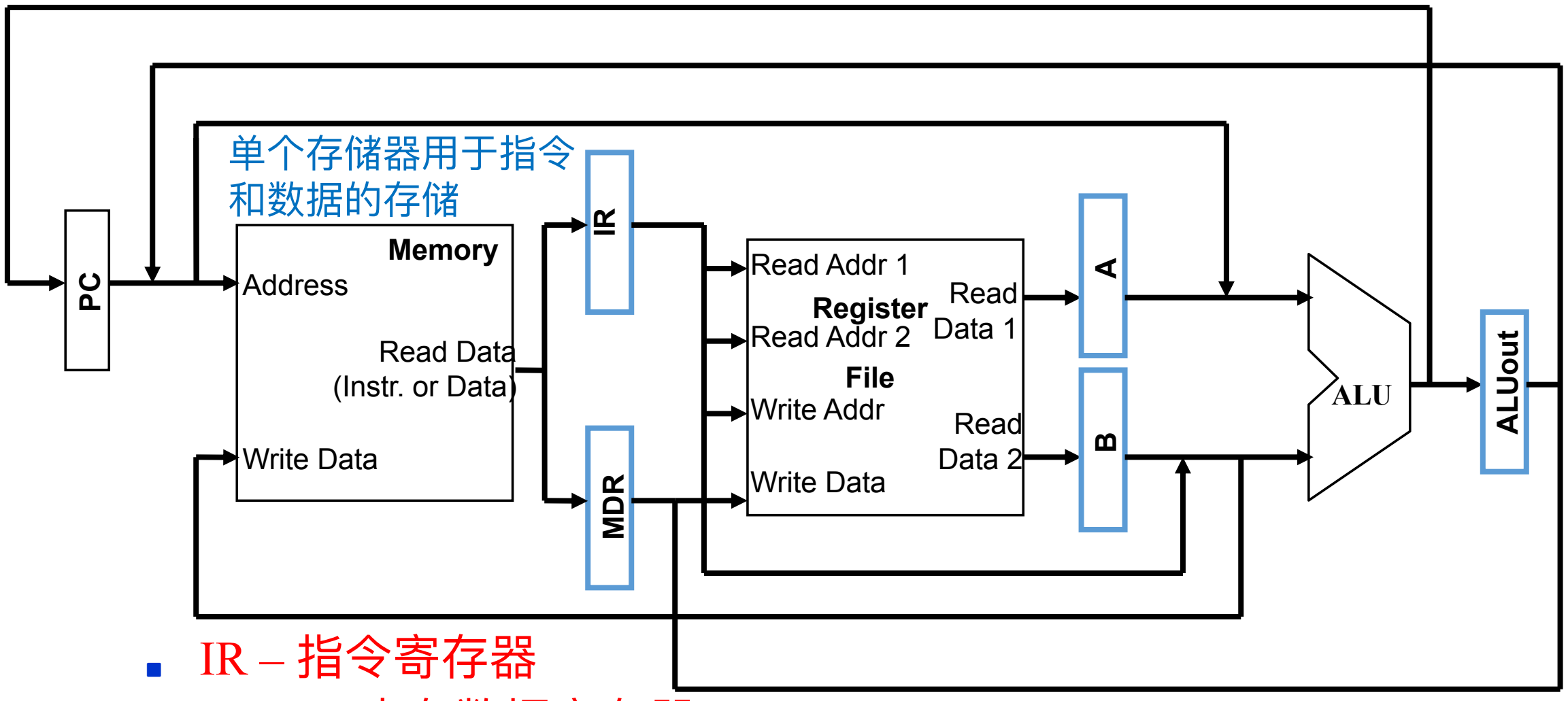
Clocking the Multicycle Datapath



extra registers added – IR, MDR, A, B, and ALUout

The Multicycle Datapath – A High Level View

- 必须在每个主要功能单元之后添加寄存器以保存输出值，在随后的时钟周期中使用它.



- IR – 指令寄存器
- MDR – 内存数据寄存器
- A and B – 寄存器堆读端口两个寄存器
- ALUout – ALU 输出寄存器
- 除了IR指令寄存器，所有的寄存器只保存到它的下一个周期，它们不需要写使能。

指令各阶段分析

- 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
- 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”分支地址
- 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
  - 如果是分支branch指令，该阶段需决定是否将分支地址写入PC
- 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

第一个周期（取值周期）

第二个周期
（译码取数周期）

第三、四、五个周期（各指令不同）

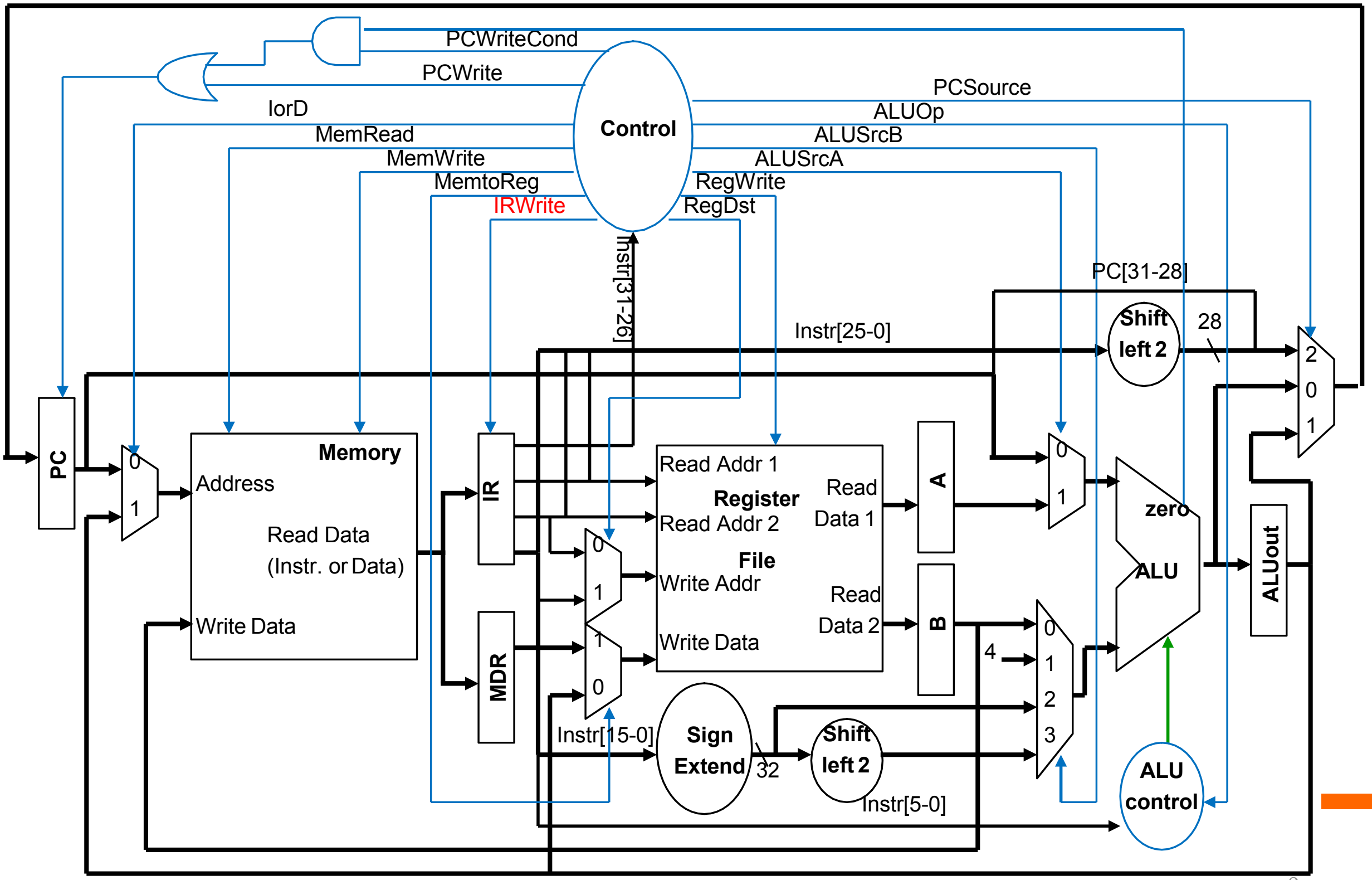
Multicycle Implementation Overview

- 每条指令每一步用一个时钟周期
  - 因此每条指令花费多个时钟周期
- 不是每条指令都花费同样的时钟周期数
- 多周期CPU
  - 更快的时钟频率
  - 不同的指令用不同的周期数
  - 单个ALU用于所有计算
  - 单个存储器用于指令和数据的存储
  - 需要增加寄存器暂存数据用于跨时钟周期

Our Multicycle Approach

- **状态器件的读写：**寄存器堆或PC每个时钟开始后进行读操作，时钟结束进行写操作。
- 寄存器堆读花差不多~50% 的时钟周期时间，控制器与其并行进行。
- 在功能部件前面增加选择器，因为不同的时钟周期共享同一个部件。
- 所有的操作在一个时钟周期内并行进行。
  - 与单周期不同的是PC+4，条件跳转地址计算都在ALU上进行，指令存储器和数据存储器使用同一个存储器，每个时钟周期都进行寄存器堆的读操作。

The Complete Multicycle Data with Control



Five Execution Steps

1. Instruction Fetch
2. Instruction Decode and Register Fetch
3. Execution, Memory Address Computation, or Branch Completion
4. Memory Access or R-type instruction completion
5. Write-Back Step

INSTRUCTIONS TAKE FROM 3 - 5 STEPS/CYCLES!

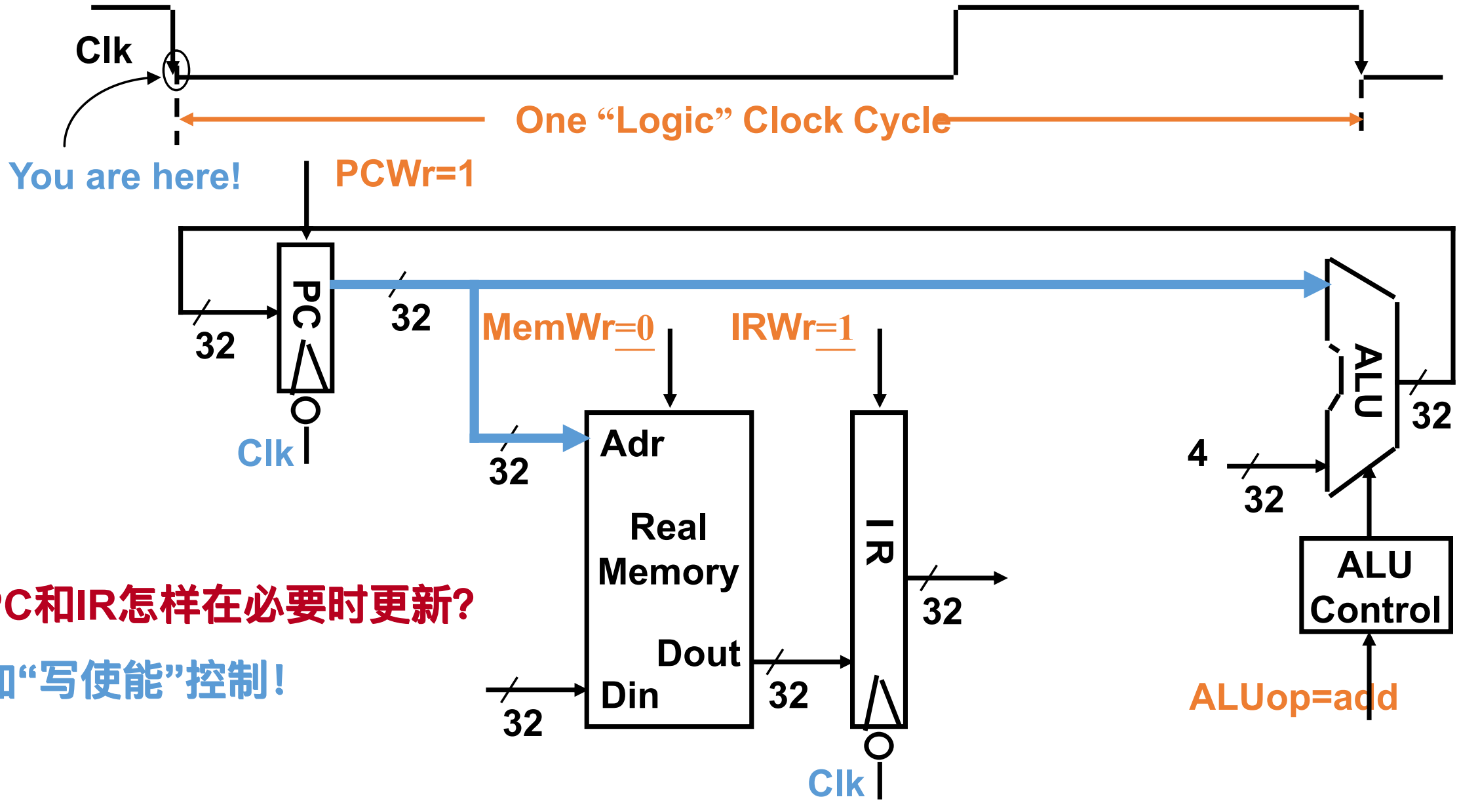
RTL Summary

Step	R-type	Mem Ref	Branch	Jump
Instr fetch	IR = Memory[PC]; PC = PC + 4			
Decode	A = Reg[IR[25-21]]; B = Reg[IR[20-16]]; ALUOut = PC +(sign-extend(IR[15-0])<< 2);			
Execute	ALUOut = A op B;	ALUOut = A + sign-extend (IR[15-0]);	if (A==B) PC = ALUOut;	PC = PC[31-28] ||(IR[25-0] << 2);
Memory access	Reg[IR[15-11]] = ALUOut;	MDR = Memory[ALUOut]; or Memory[ALUOut] = B;		
Write-back		Reg[IR[20-16]] = MDR;		

Instruction Common Steps

Step 1: Instruction Fetch

- 在一个时钟到来的下降沿开始取指令周期的任务：
  - M[PC]； PC←PC + 4

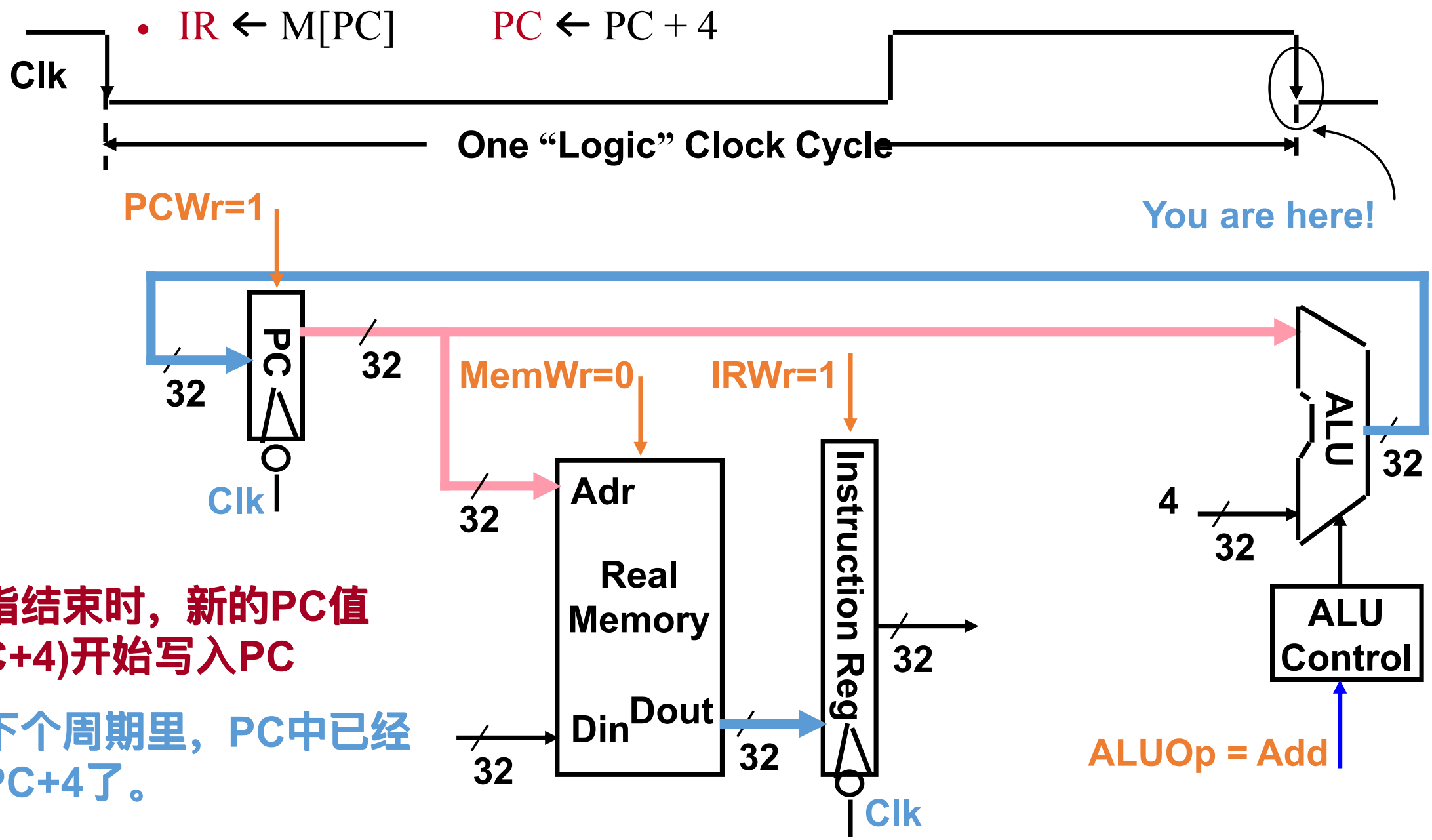


PC和IR怎样在必要时更新?

加“写使能”控制!

取指周期结束时

- 每一个周期都在下一个时钟到来时结束 (此时，受时钟控制的存储元件被更新。不受时钟约束的存储原件则没有此特点):

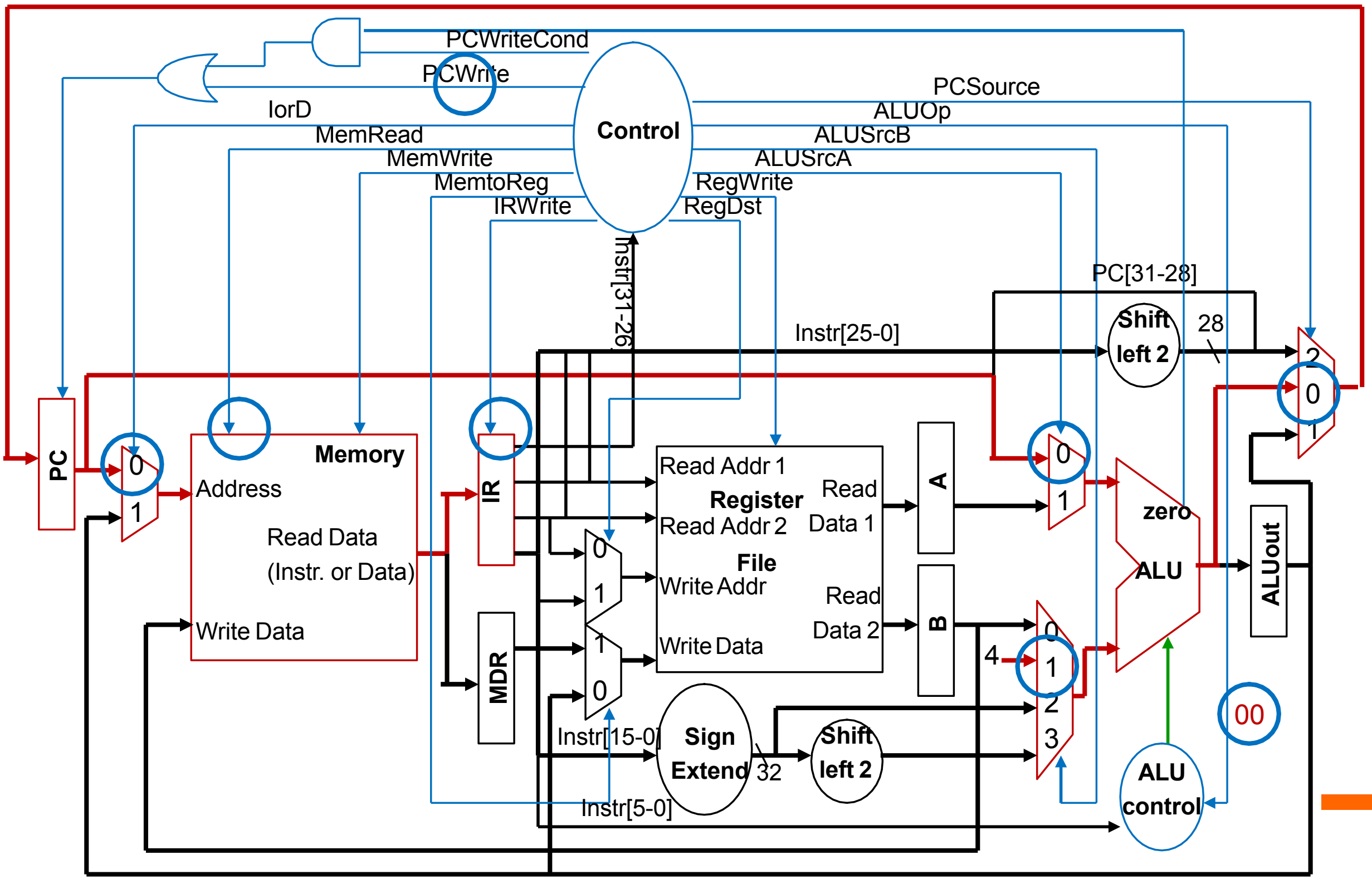


取指结束时，新的PC值(PC+4)开始写入PC

即下个周期里，PC中已经是PC+4了。

取指结束时，当前指令开始写入IR！为保证本指令期间IR中指令不变，后面周期中IRWr应该为0

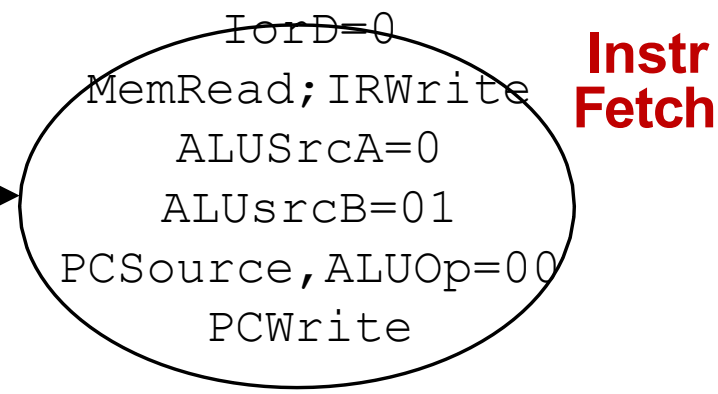
Datapath Activity During Instruction Fetch



Fetch Control Signals Settings

Unless otherwise assigned

PCWrite,IRWrite,
MemWrite,RegWrite=0
others=X



Step 2: Instruction Decode and Register Fetch

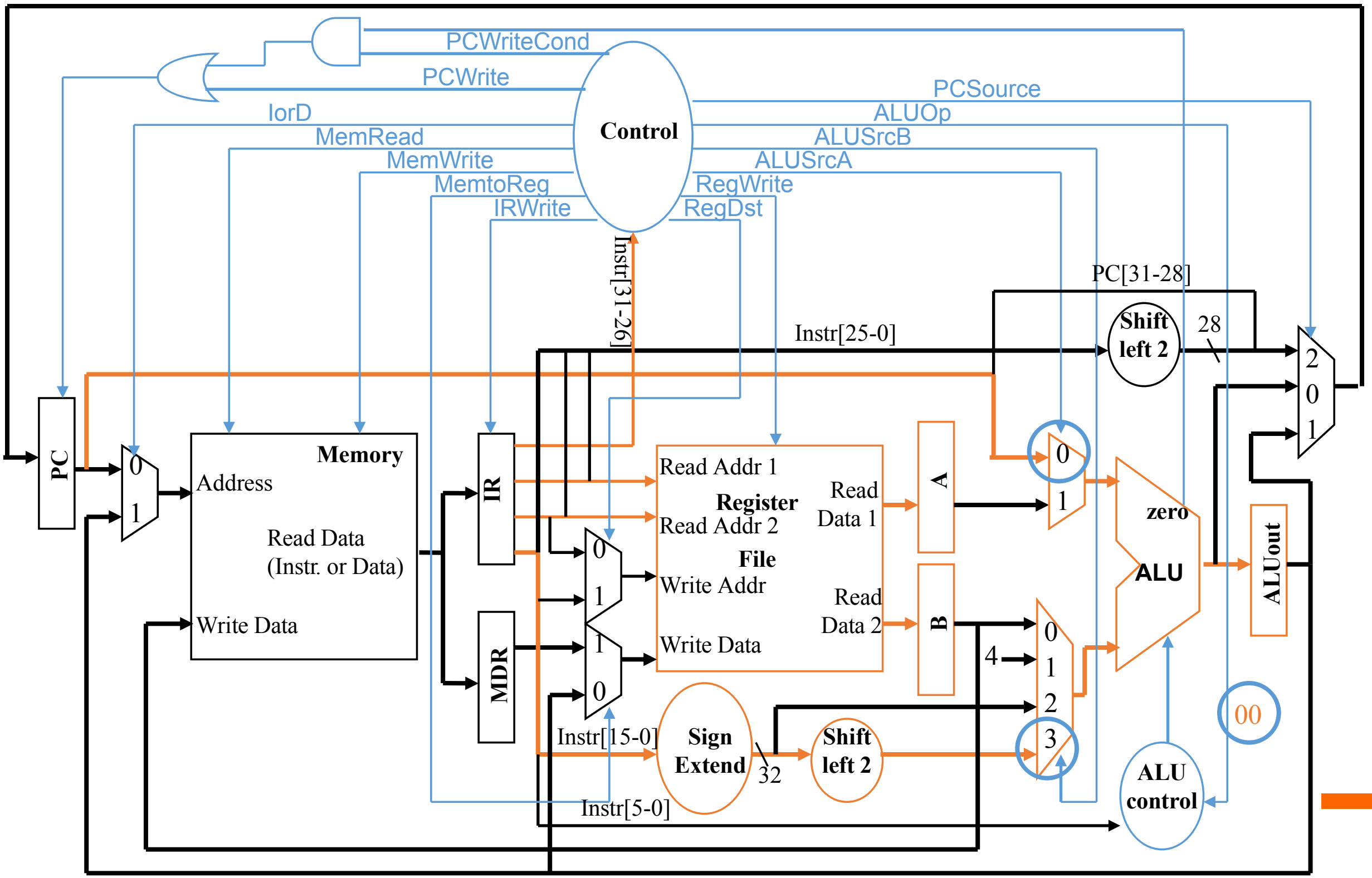
- 指令未译码，故只执行公共操作，ALU空闲，可用ALU“投机计算”转移地址
- **`ALUOut = PC+(sign-extend(IR[15-0])<< 2);`**
- 读出寄存器堆的两个寄存器内容

The RTL:

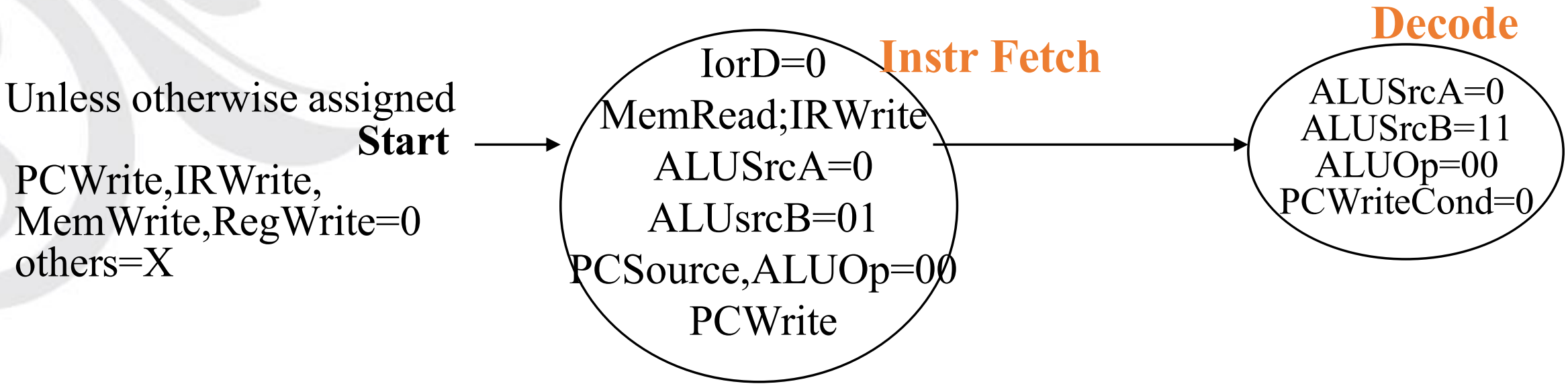
- `A = Reg[IR[25-21]];`
- `B = Reg[IR[20-16]];`

请注意，我们没有根据指令设置任何控制线（因为控制逻辑正忙于“译码”操作码位），还不知道它是什么）

Datapath Activity During Instruction Decode



Decode Control Signals Settings



指令未译码，故只执行公共操作

ALU空闲，可用ALU“投机计算”转移地址！

- busA ← Reg[rs] ; busB ← Reg[rt] ;
- Decoder ← Op and Func;
- **投机：** Target ← PC + SignExt(Imm16)*4

（为什么不是PC +4+ SignExt(Imm16)*4**？）PC中已是下条顺序指令的地址**

Step 3 Instruction Dependent Operations

- ALU is performing one of four functions, based on instruction type
- Memory reference (lwand sw):

  ALUOut = A + sign-extend(IR[15-0]);
- R-type:

  ALUOut = A op B;
- Branch:

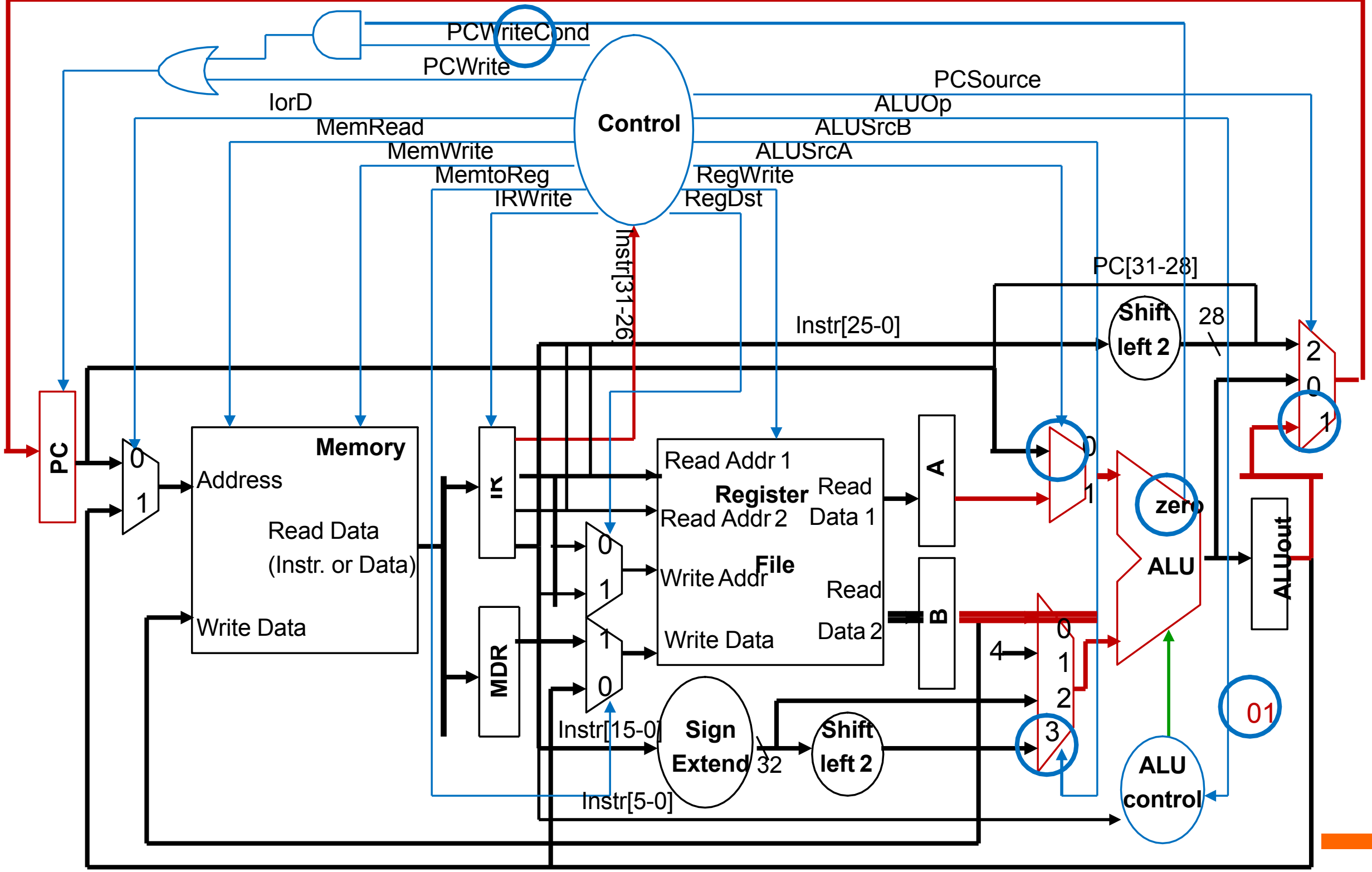
  if (A==B) PC = ALUOut;
- Jump:

  PC = PC[31-28] || (IR[25-0] << 2);

如果指令译码输出为：Beq

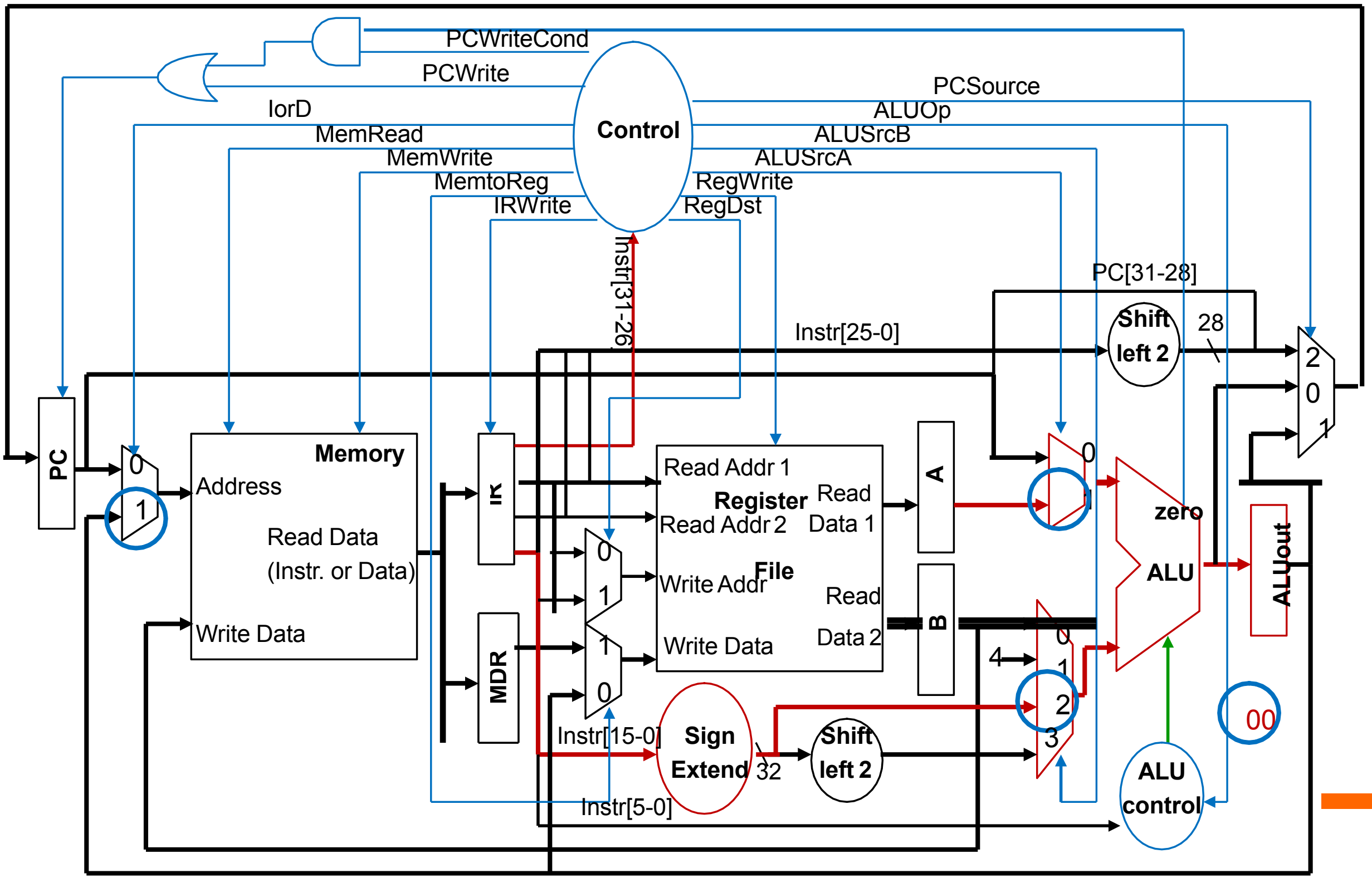
下面第三个周期就是Beq指令的第一个执行周期！

Datapath Activity During beq Execute

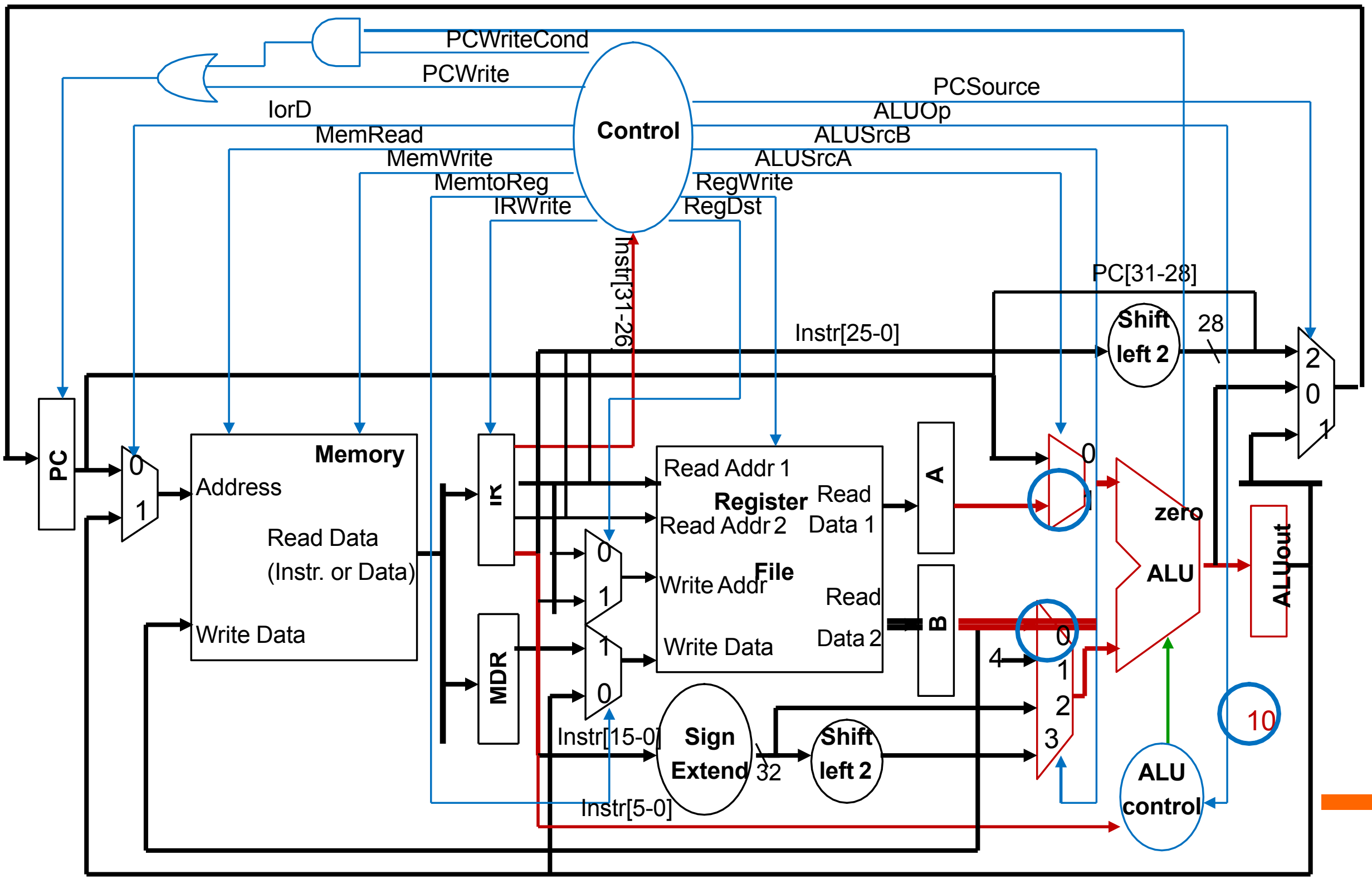


zero信号到达IFU，Aluout寄存器内容到达PC，虽然PCwrite=0，但如果满足Beq跳转条件，zero信号与PCWriteContion相与为1，则写入跳转地址到PC。

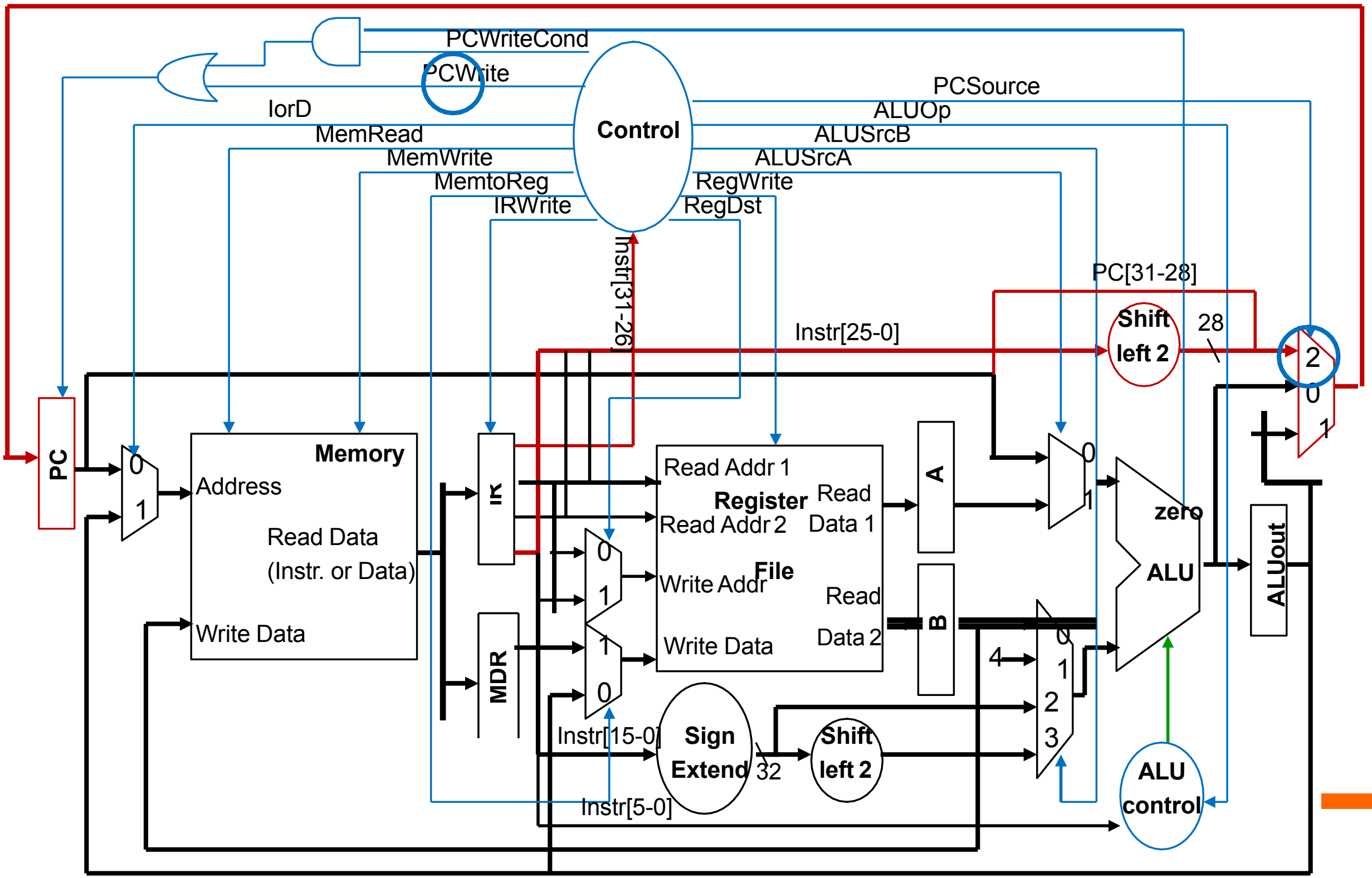
Datapath Activity During `lw` & `sw` Execute



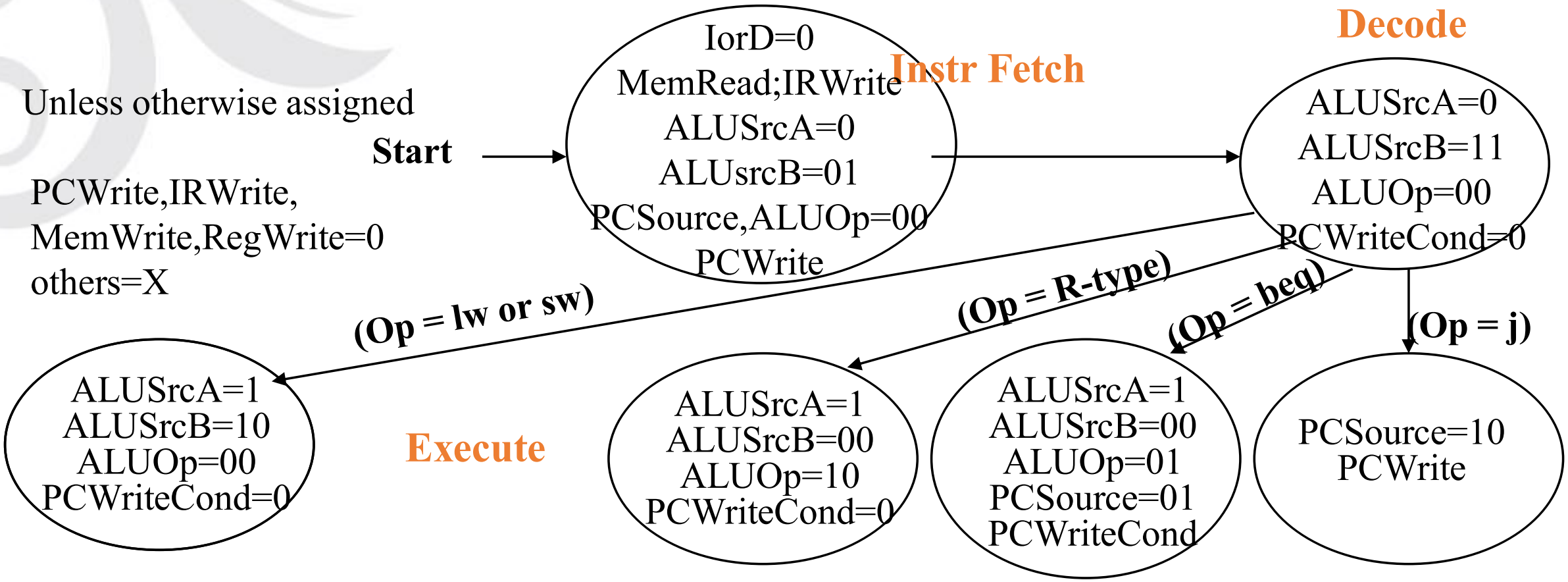
Datapath Activity During R-type Execute



Datapath Activity During j Execute



Execute Control Signals Settings



Step 4 (also instruction dependent)

- LW指令，从内存读出的数据存入内存数据寄存器ＭＤＲ，ＳW指令把端口B读出的内容写入内存:

  MDR = Memory[ALUOut]; -- lw

  Or

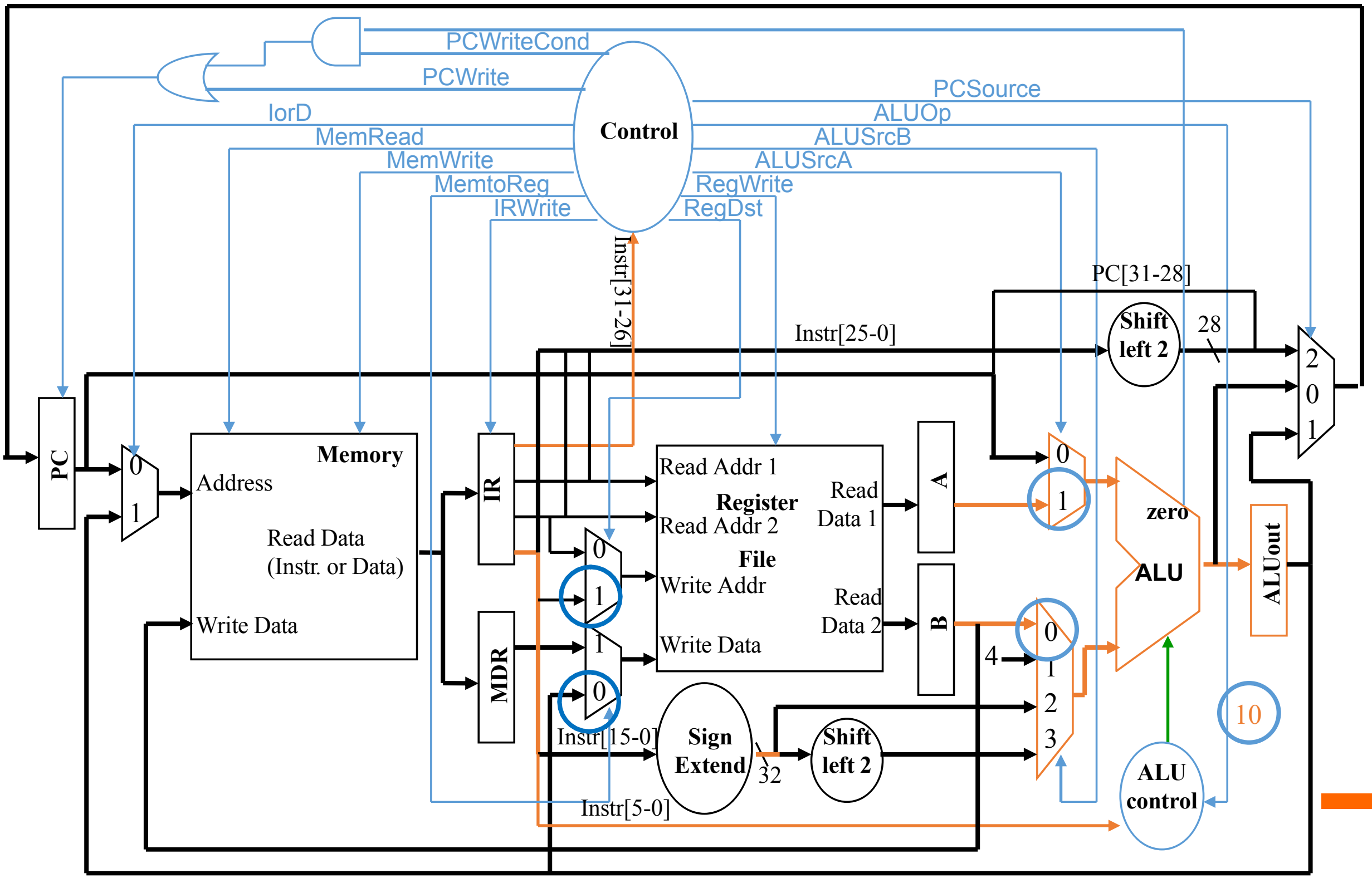
  Memory[ALUOut] = B; -- sw

- R-type 计算结果写入目标寄存器

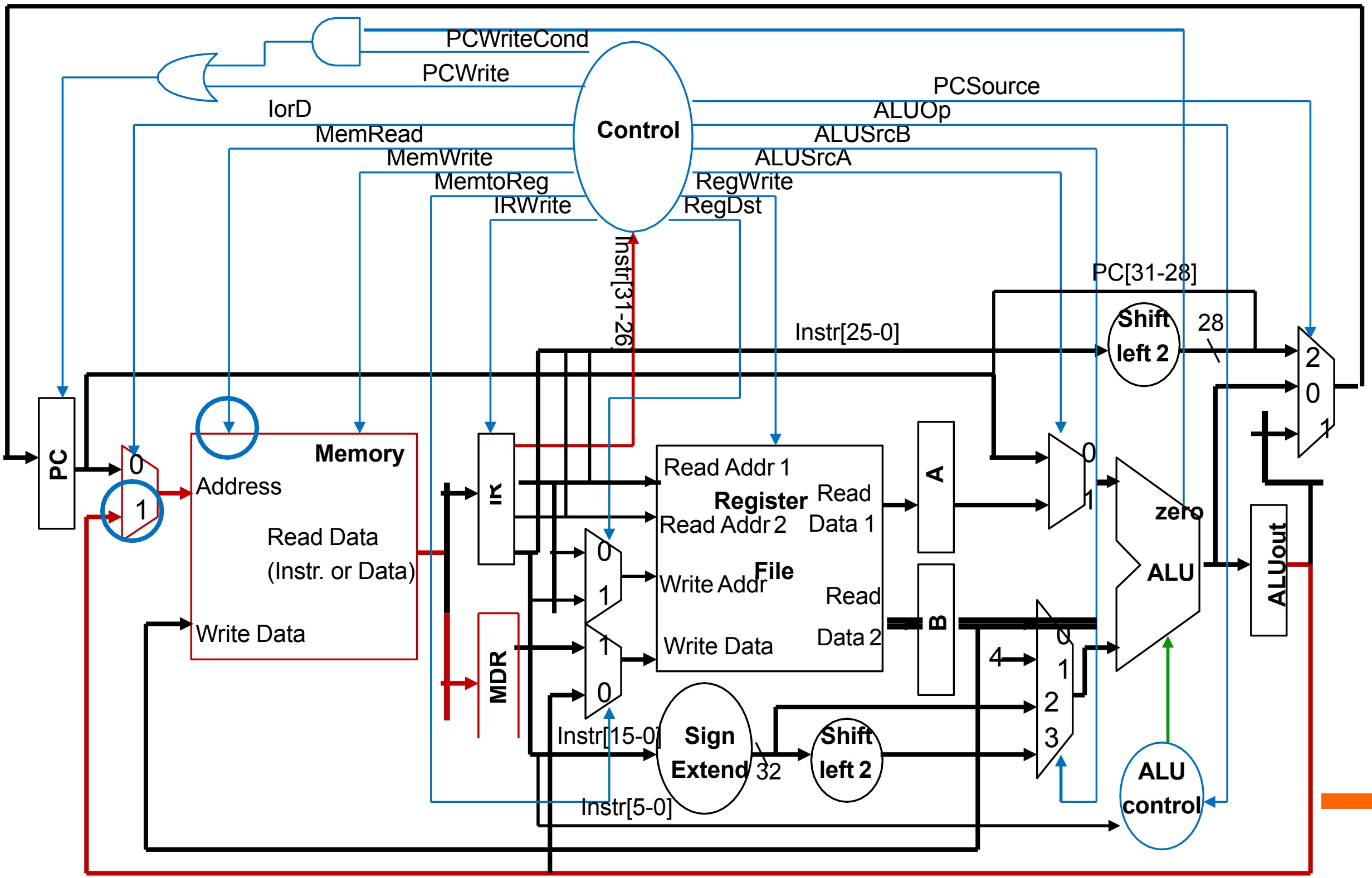
  Reg[IR[15-11]] = ALUOut;

- 请记住，寄存器写入实际上发生在时钟周期结束时的边沿

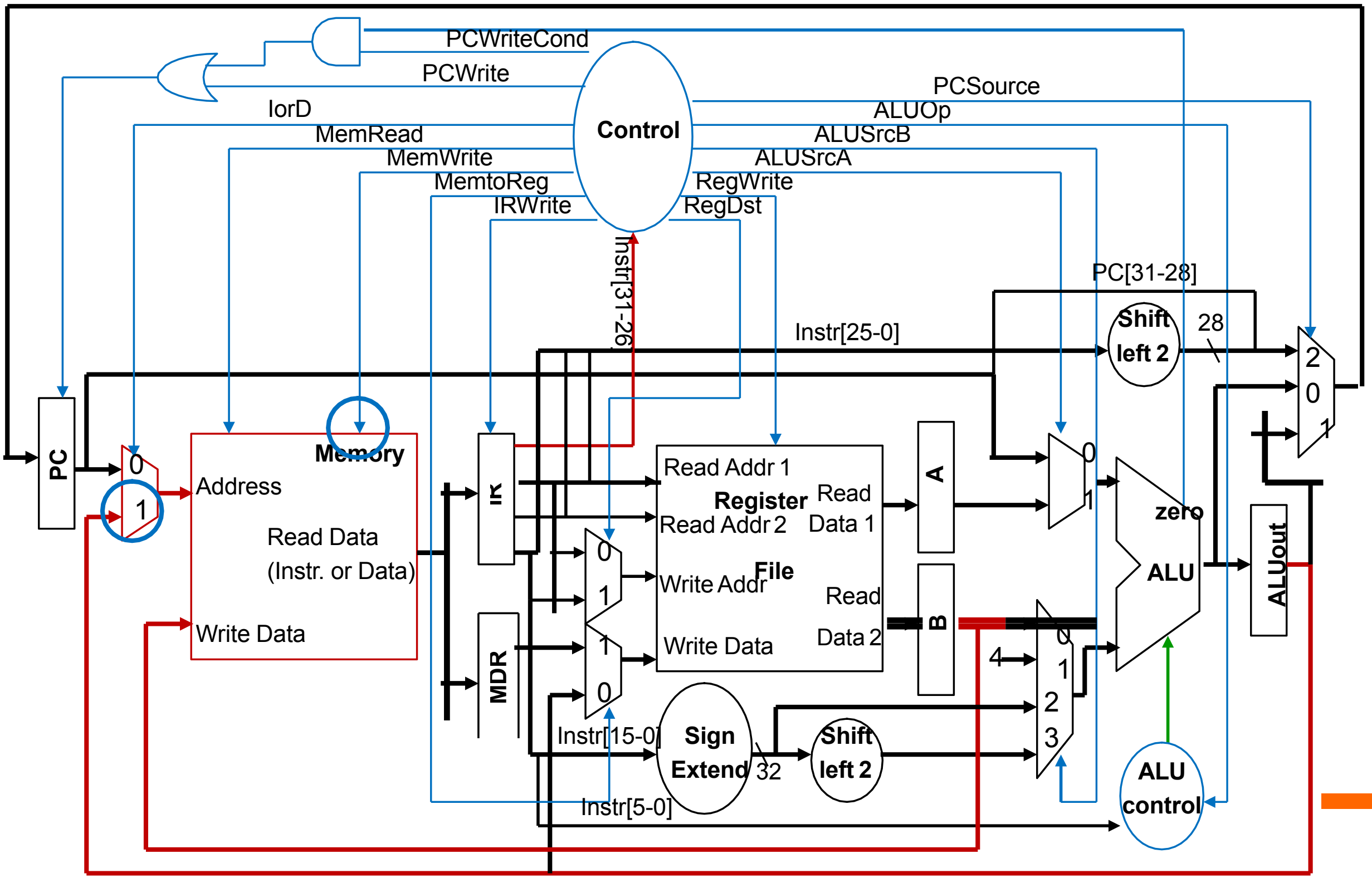
Datapath Activity During R-type Execute



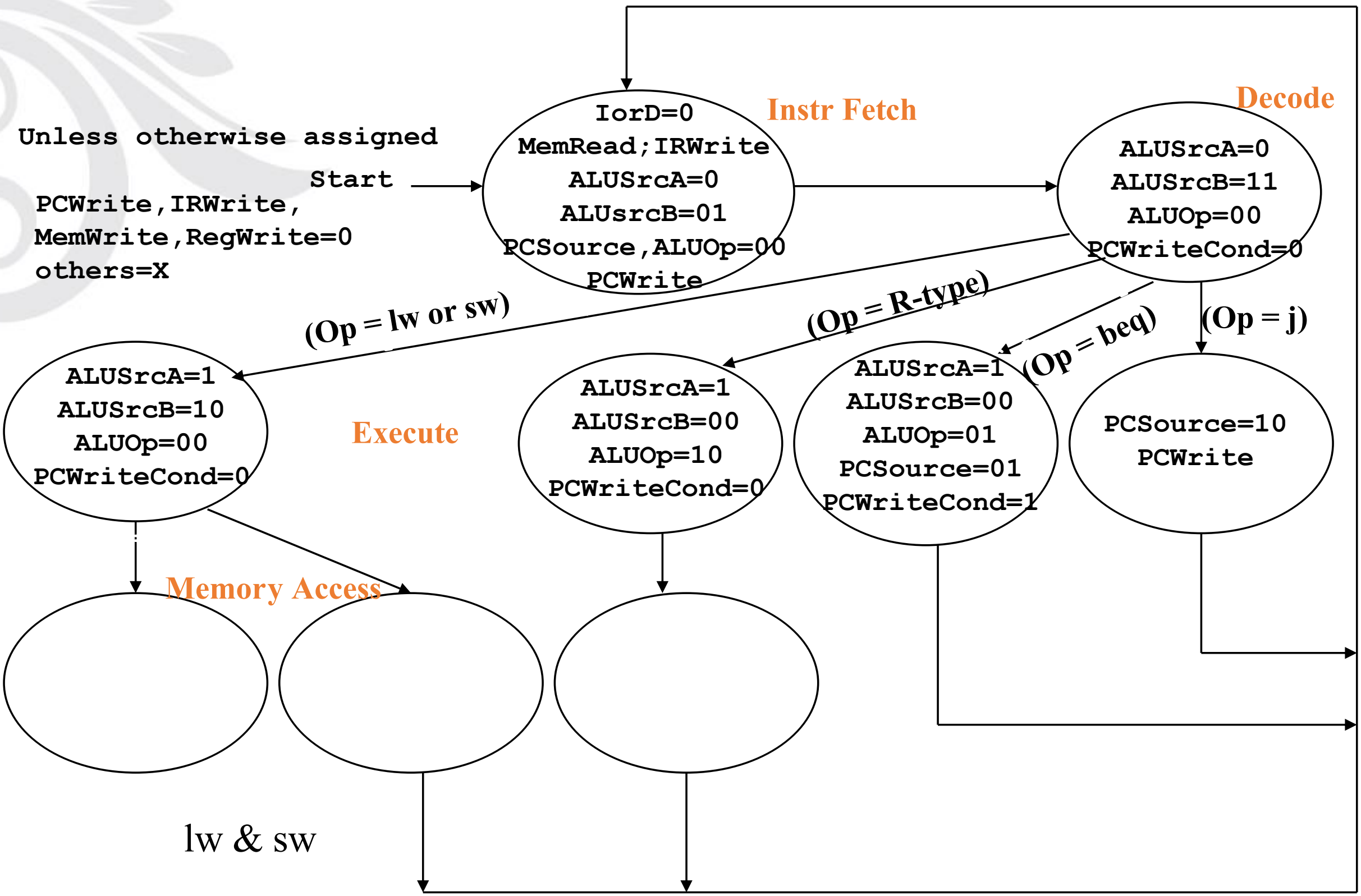
Datapath Activity During `lw` Memory Access



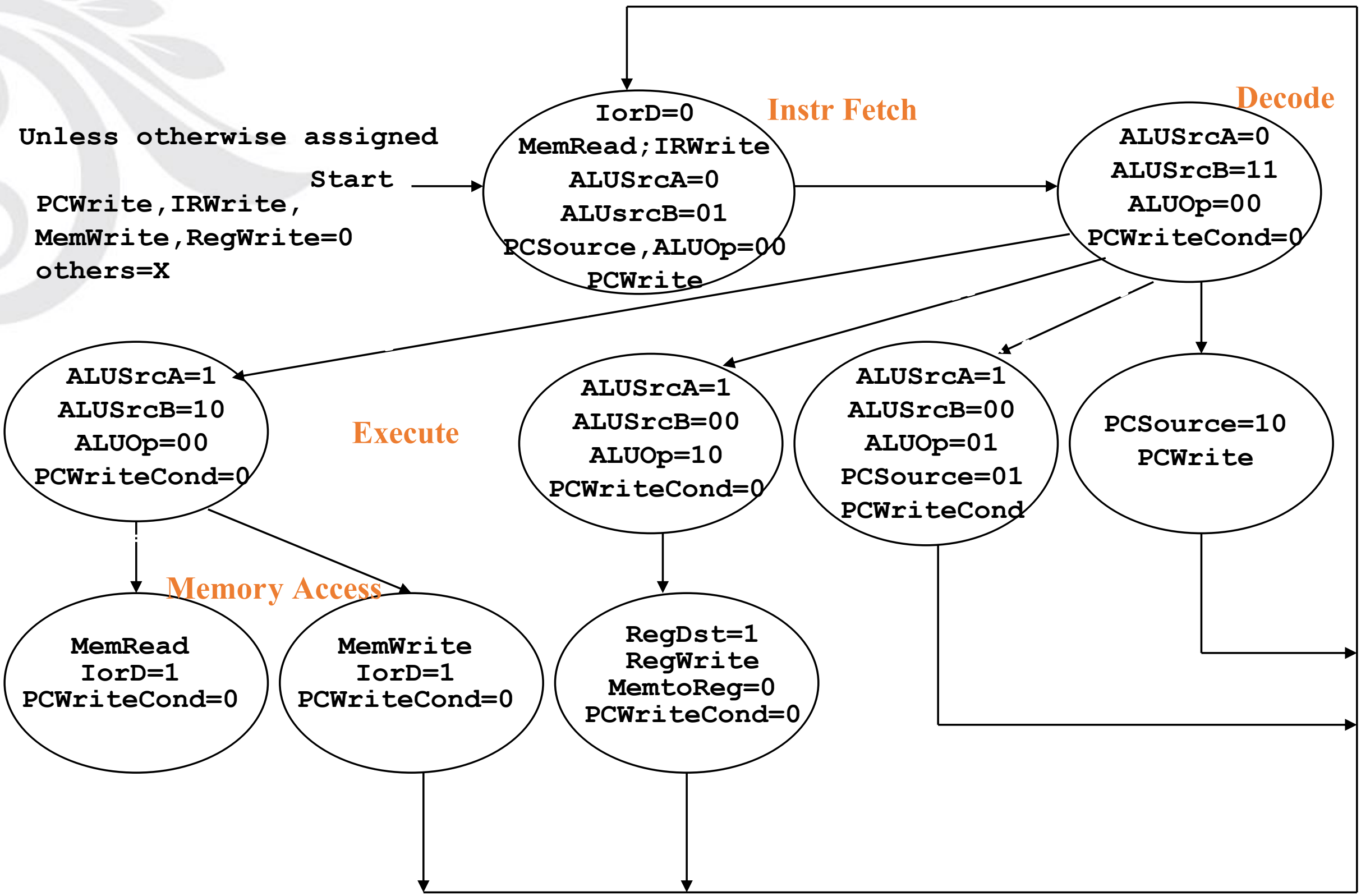
Datapath Activity During `sw` Memory Access



Memory Access Control Signals Settings



Memory Access Control Signals Settings



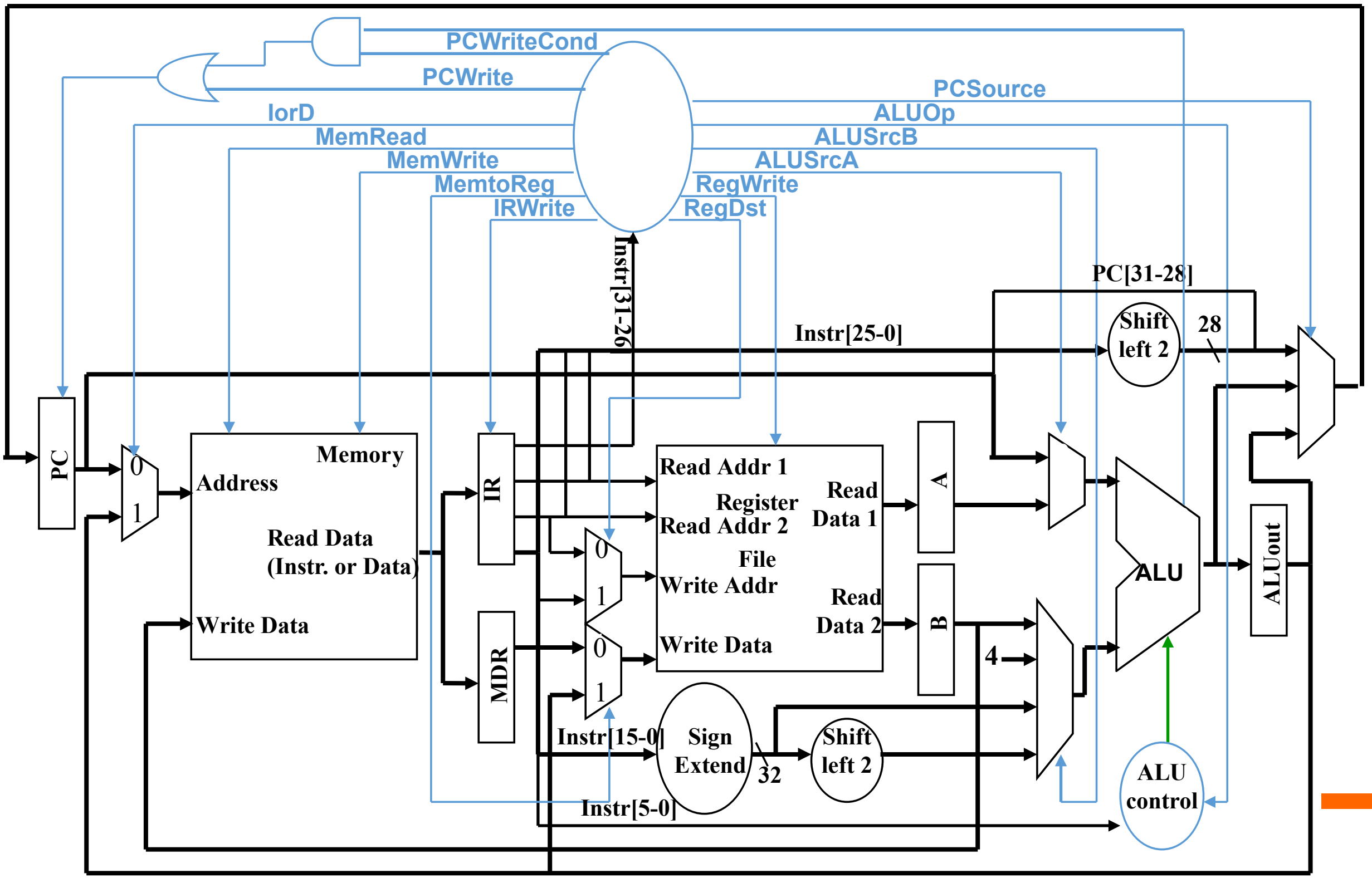
Step 5: Memory Read Completion (Write Back)

- All we have left is the write back into the register file the data just read from memory for lw instruction

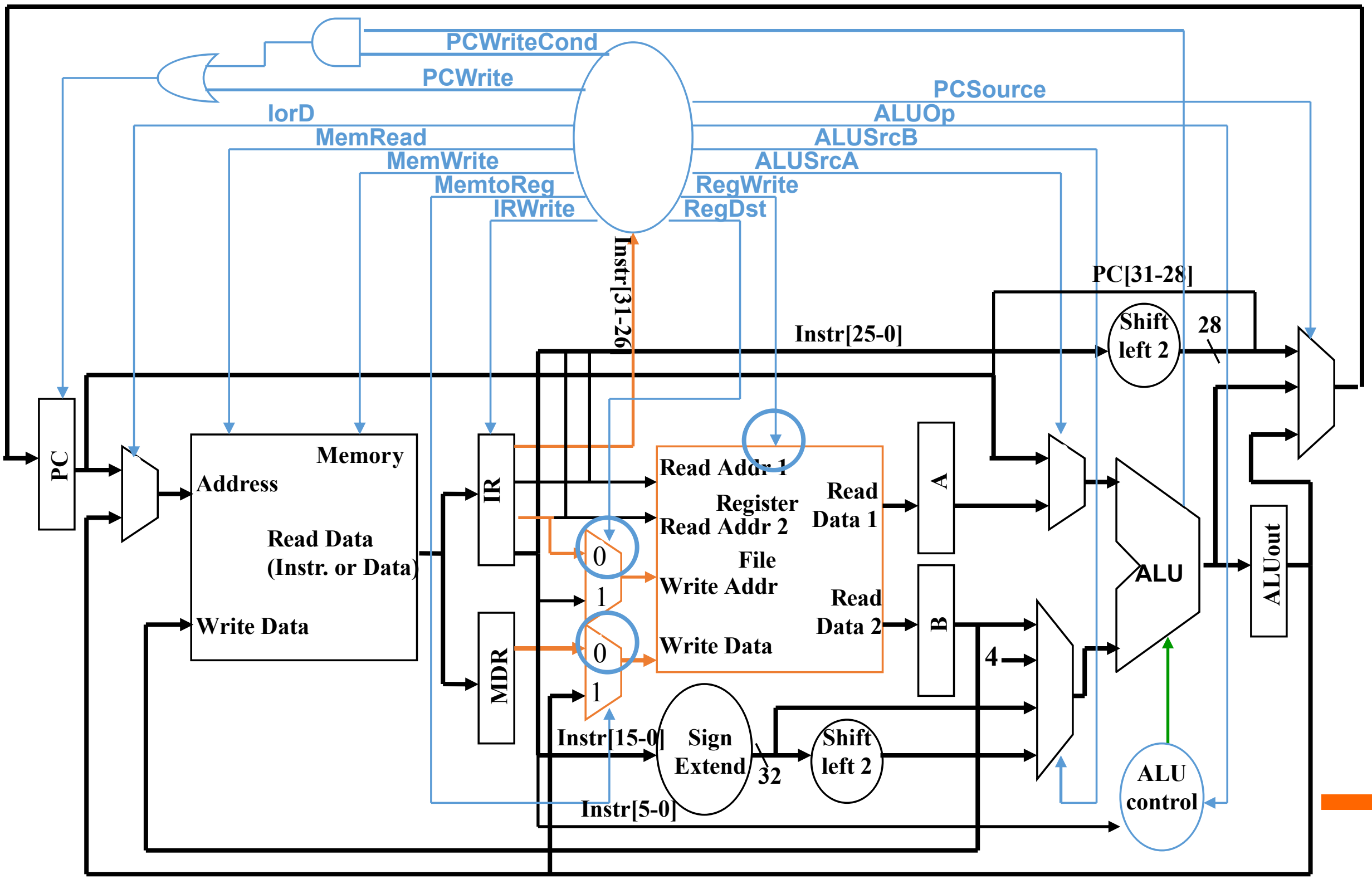
```
Reg[IR[20-16]]= MDR;
```

What about all the other instructions?

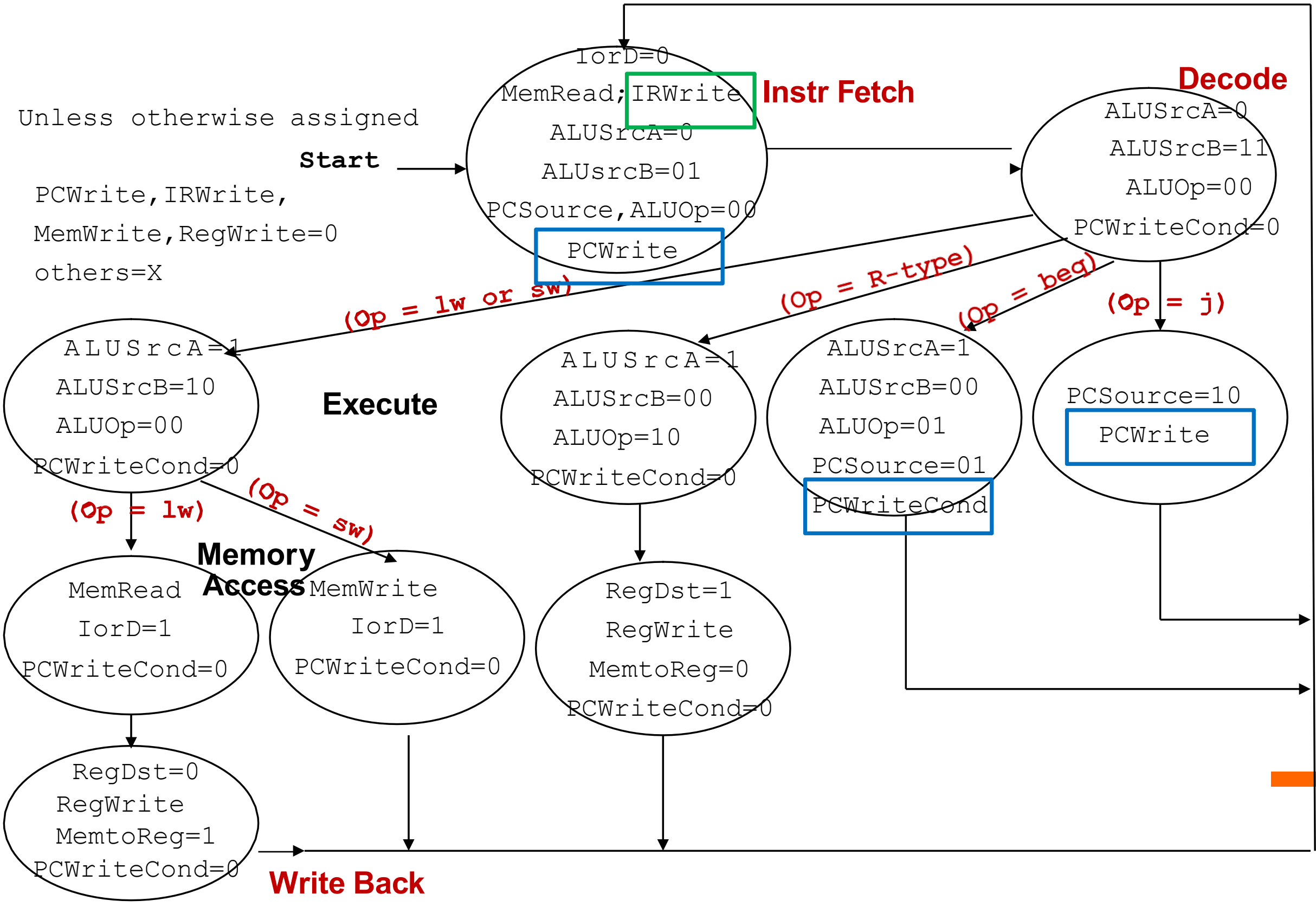
Datapath Activity During `lw` Write Back



Datapath Activity During `lw` Write Back



Write Back Control Signals Settings



RTL Summary

Step	R-type	Mem Ref	Branch	Jump
Instr fetch	IR = Memory[PC]; PC = PC + 4;			
Decode	A = Reg[IR[25-21]]; B = Reg[IR[20-16]]; ALUOut = PC +(sign-extend(IR[15-0])<< 2);			
Execute	ALUOut = A op B;	ALUOut = A + sign-extend (IR[15-0]);	if (A==B) PC = ALUOut;	PC = PC[31-28] ||(IR[25-0] << 2);
Memory access		MDR = Memory[ALUOut]; or Memory[ALUOut] = B;		
Write-back	Reg[IR[15-11]] = ALUOut;	Reg[IR[20-16]] = MDR;		

多周期控制器的实现

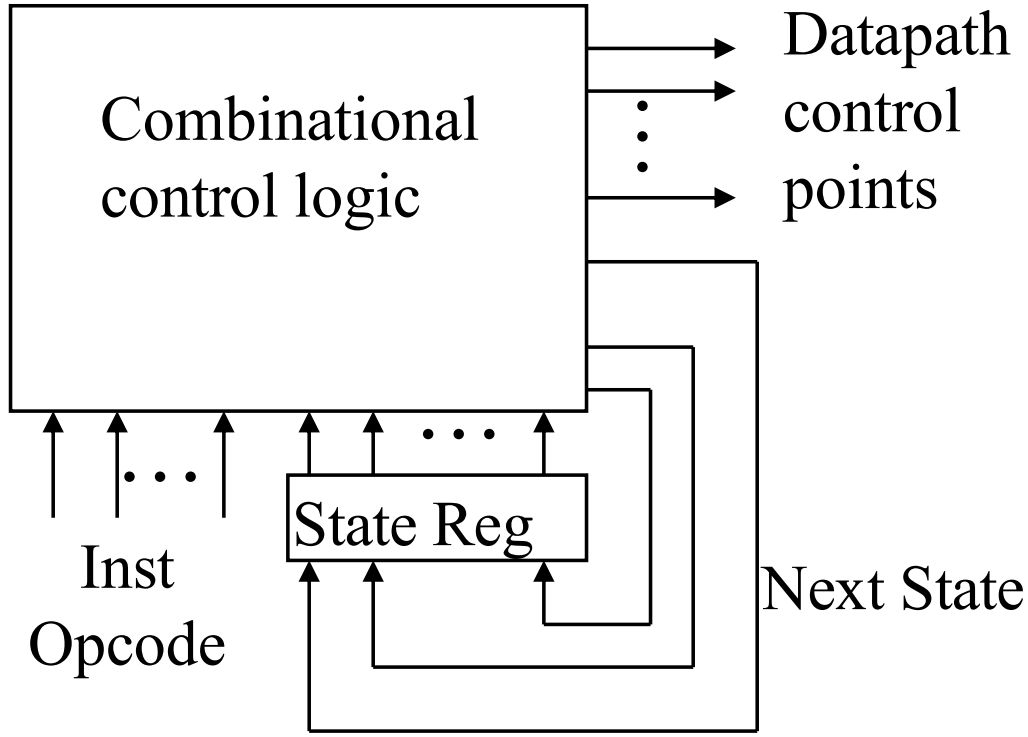
回忆单周期控制器的实现：控制信号在整个指令执行过程中不变，用真值表能反映指令和控制信号的关系。根据真值表就能实现控制器！

多周期控制器能不能这样做?

不可以，因为：
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：
  - 采用组合逻辑设计
  - 用硬连线路(PLA)实现
- 微程序：
  - 用ROM存放微程序实现



Multicycle Datapath Finite State Machine

Unless otherwise assigned

PCWrite,IRWrite,
MemWrite,RegWrite=0
others=X

Start

0
IorD=0
MemRead;IRWrite
ALUSrcA=0
ALUsrcB=01
PCSource,ALUOp=00
PCWrite

Instr Fetch

1
Decode
ALUSrcA=0
ALUSrcB=11
ALUOp=00
PCWriteCond=0

(Op = lw or sw)

(Op = R-type)

(Op = beq)

(Op = j)

2
ALUSrcA=1
ALUSrcB=10
ALUOp=00
PCWriteCond=0

Execute

6
ALUSrcA=1
ALUSrcB=00
ALUOp=10
PCWriteCond=0

8
ALUSrcA=1
ALUSrcB=00
ALUOp=01
PCSource=01
PCWriteCond

9
PCSource=10
PCWrite

(Op = lw)

(Op = sw)

3
MemRead
IorD=1
PCWriteCond=0

Memory Access

5
MemWrite
IorD=1
PCWriteCond=0

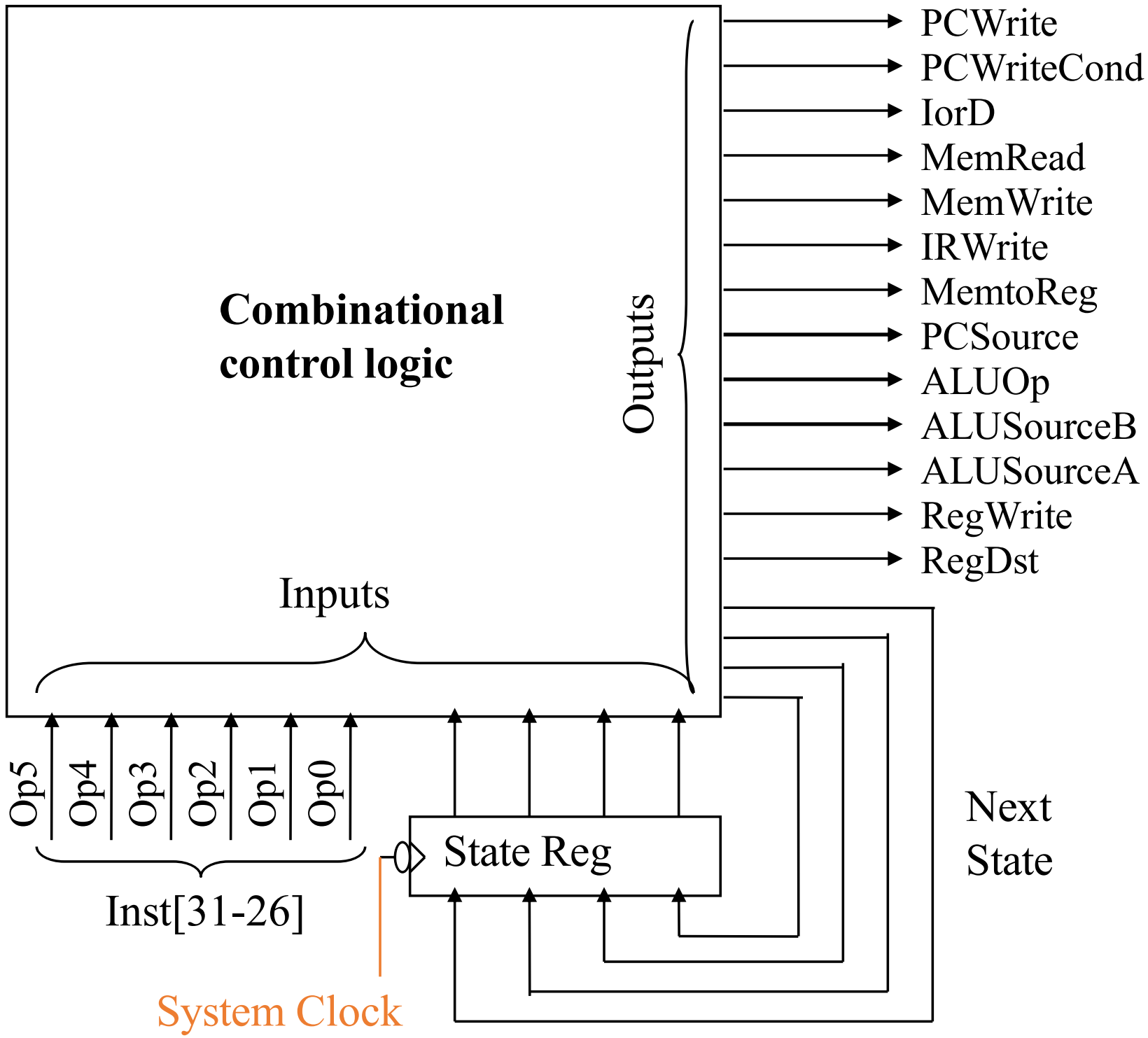
7
RegDst=1
RegWrite
MemtoReg=0
PCWriteCond=0

4
RegDst=0
RegWrite
MemtoReg=1
PCWriteCond=0

Write Back

State Assignment

Finite State Machine Implementation



Datapath Control Outputs Truth Table

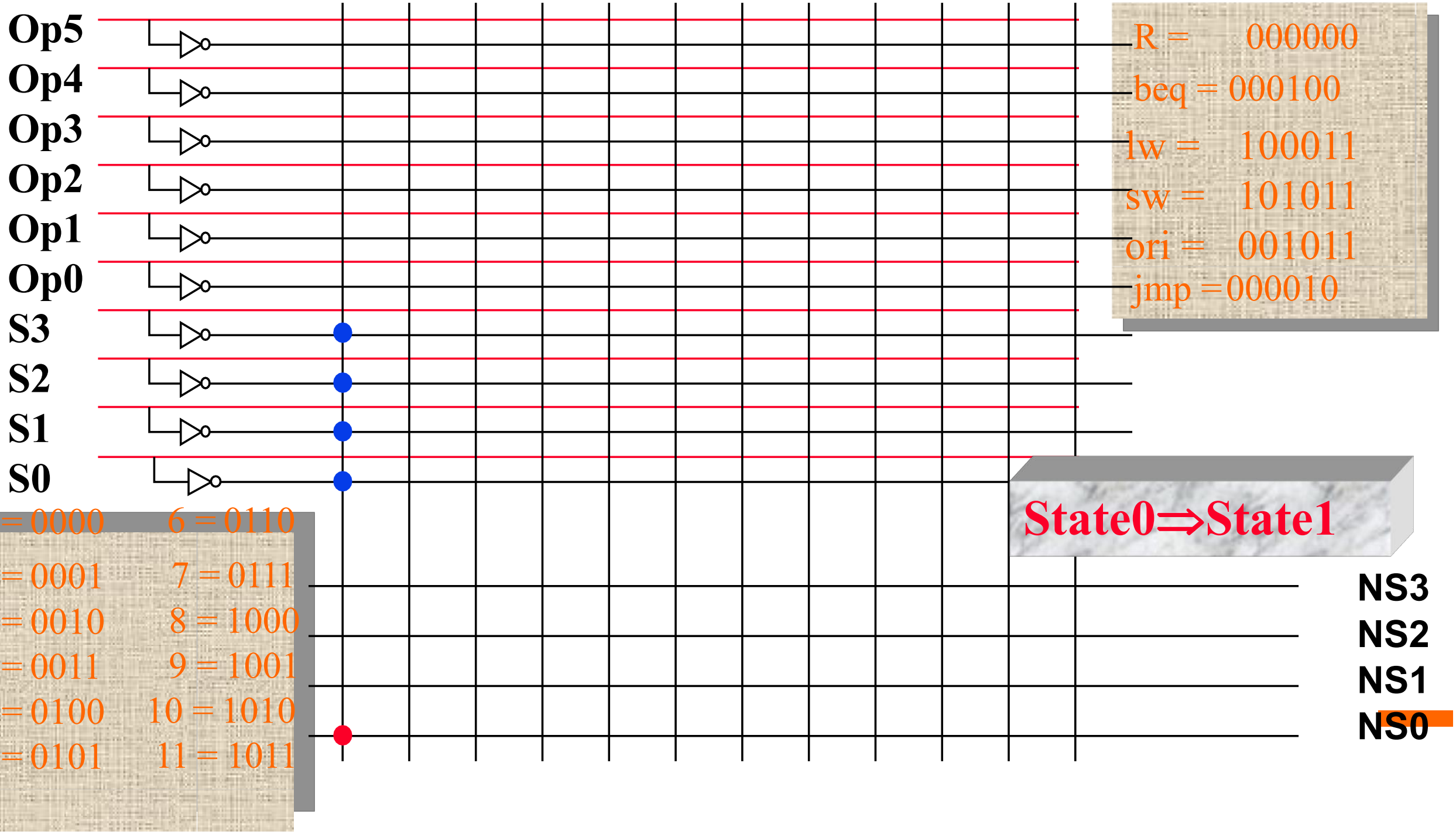
Outputs	Input Values (Current State[3-0])									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	X	0	0	0	0	0	0	0	1	X
IorD	0	X	X	1	X	1	X	X	X	X
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	X	X	X	X	1	X	X	0	X	X
PCSource	00	XX	XX	XX	XX	XX	XX	XX	01	10
ALUOp	00	00	00	XX	XX	XX	10	XX	01	XX
ALUSrcB	01	11	10	XX	XX	XX	00	XX	00	XX
ALUSrcA	0	0	1	X	X	X	1	X	1	X
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	X	X	X	X	0	X	X	1	X	X

Next State Truth Table

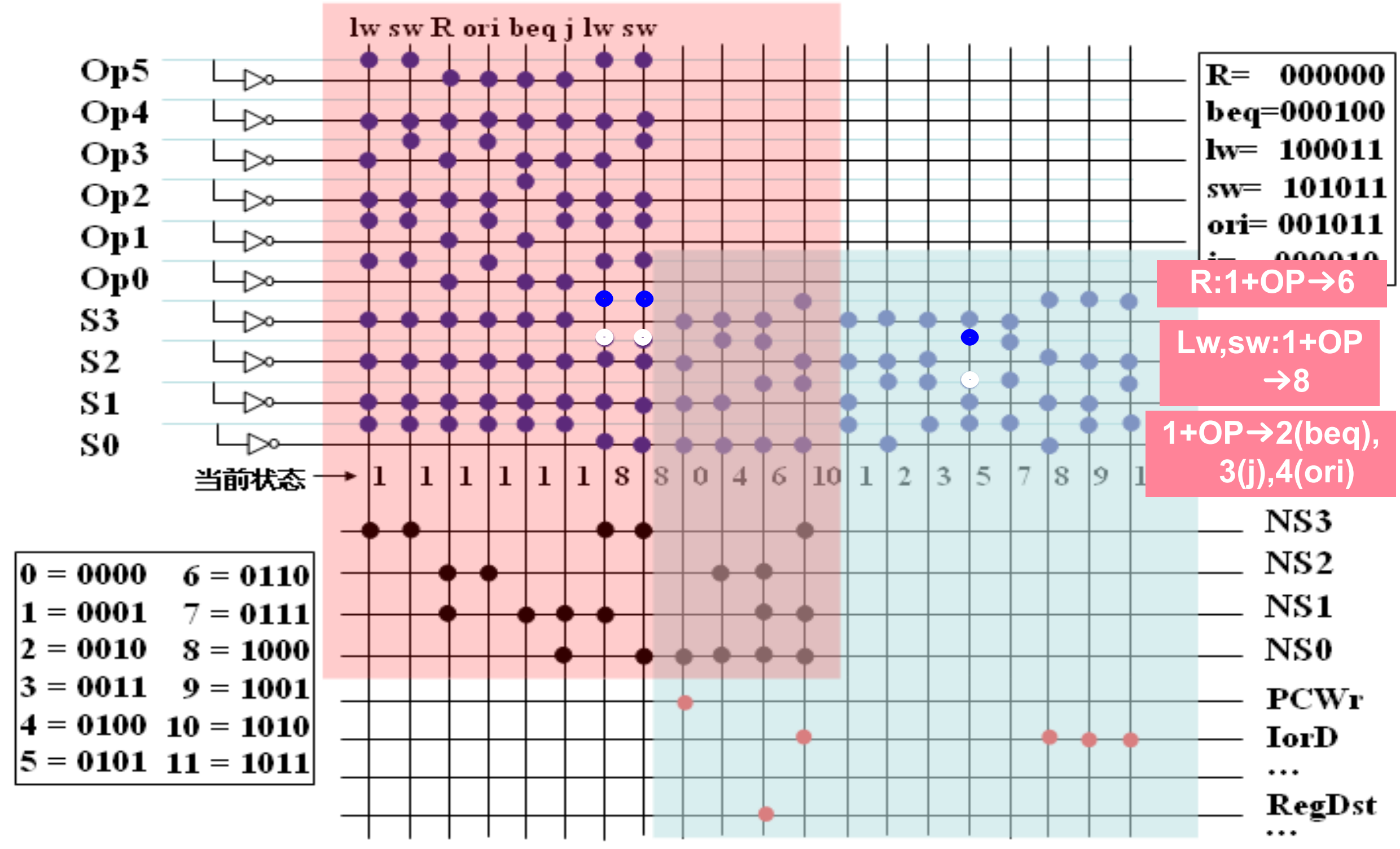
Current State [3-0]	Inst[31-26] (Op[5-0])					
	000000 (R-type)	000010 (jmp)	000100 (beq)	100011 (lw)	101011 (sw)	Any other
0000	0001	0001	0001	0001	0001	0001
0001	0110	1001	1000	0010	0010	illegal
0010	XXXX	XXXX	XXXX	0011	0101	illegal
0011	XXXX	XXXX	XXXX	0100	XXXX	illegal
0100	XXXX	XXXX	XXXX	0000	XXXX	illegal
0101	XXXX	XXXX	XXXX	XXXX	0000	illegal
0110	0111	XXXX	XXXX	XXXX	XXXX	illegal
0111	0000	XXXX	XXXX	XXXX	XXXX	illegal
1000	XXXX	XXXX	0000	XXXX	XXXX	illegal
1001	XXXX	0000	XXXX	XXXX	XXXX	illegal

实现技术: PLA(Programmed Logic Arrays)

- 每个输出线：输入信号或其反向信号 的 逻辑与 的 逻辑或: AND minterms 在上部
AND模板（plane）指定，OR sums 在底部 OR模板（plane）指定



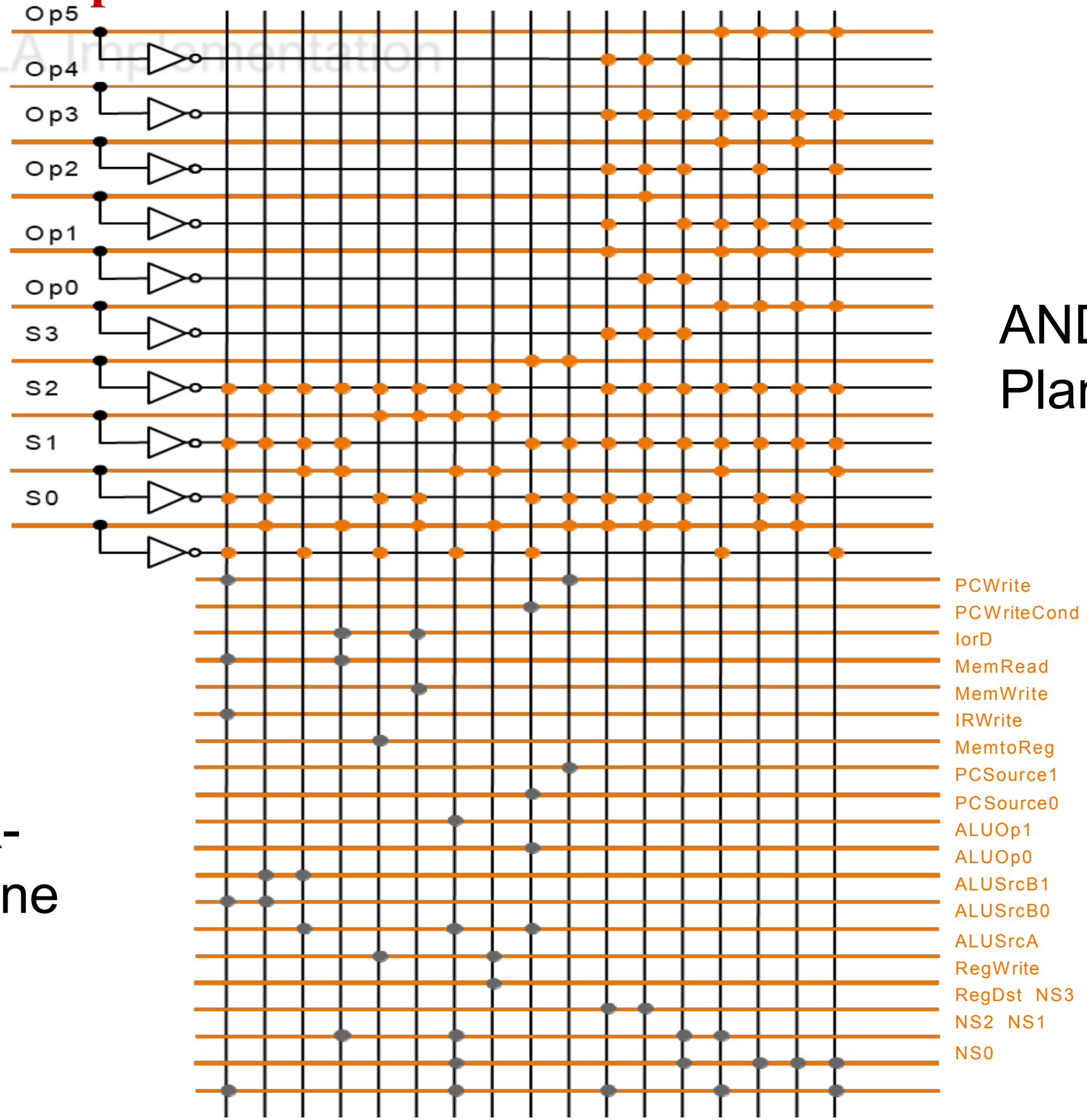
用PLA电路实现的组合逻辑控制单元（硬布线方式）



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

你能找出图中的错误吗?
有三个点的位置不对!

PLA Implementation



AND-Plane

OR-Plane

硬连线路设计和微程序设计

硬连线路设计的特点：

优点：速度快，适合于简单或规整的指令系统，例如，MIPS指令集。

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**

微程序设计的特点：具有规整性、可维性和灵活性，但速度慢

微程序设计

°控制是处理器设计的一个难点，早期计算机设计人员面临的最大挑战是正确地控制电路

•数据通路具有较好的规整性和很好的组织

•存储器具有很高的规整性

•控制不规整，并且涉及全局

微程序设计:

一种特殊的实现处理器控制部件的策略，它在寄存器传输操作的级别进行“编程”

微体系结构（Microarchitecture）：

微程序编程人员所看到的硬件的逻辑结构和功能特性

注释:

•IBM 360系列首次把体系结构和组成的概念区别开来

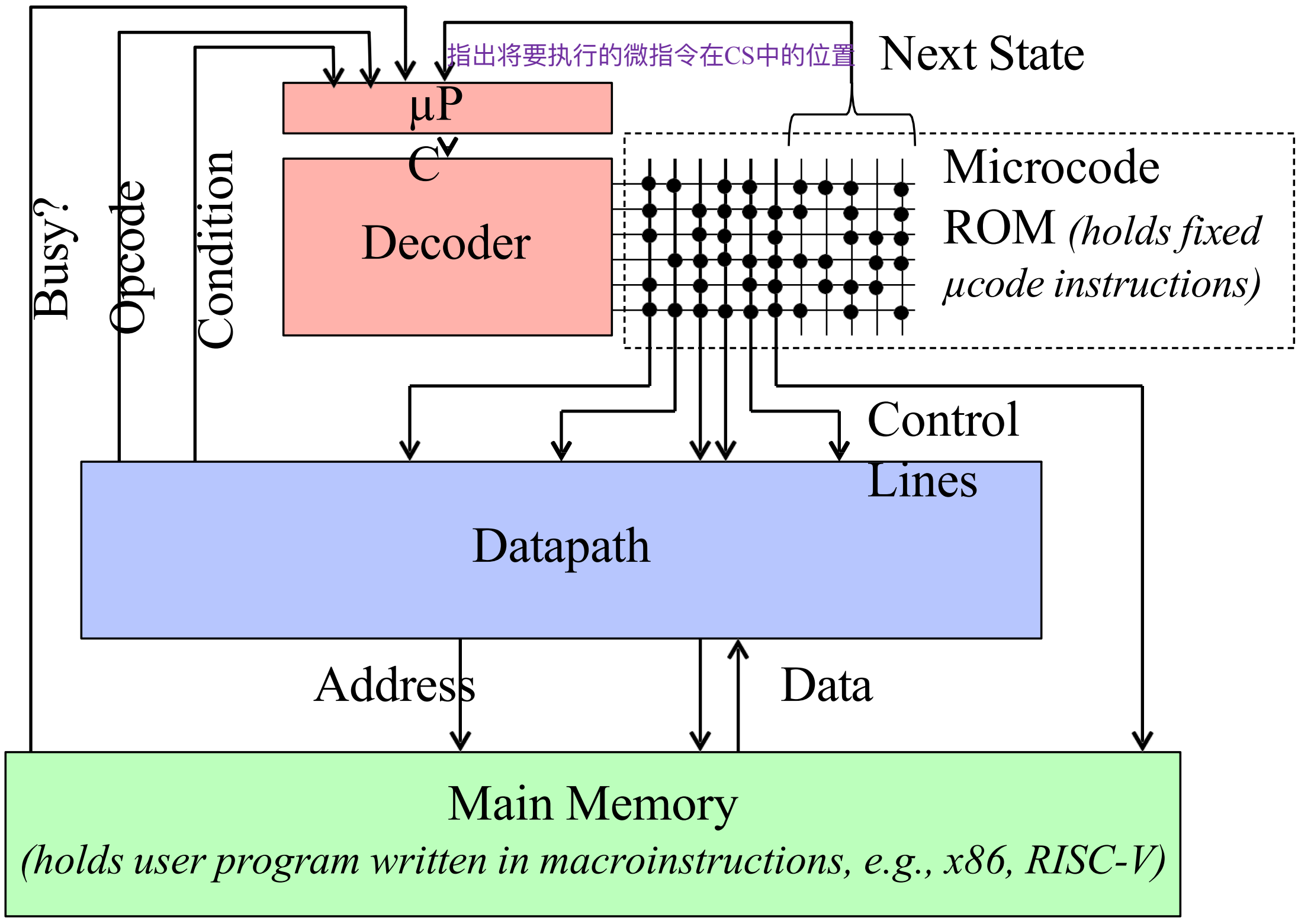
•同一指令系统在很大范围内采用不同的实现技术来实现，使其具有不同的性能价格比

控制存储器（Control Memory）

- 控制存储器概念：
  - 用于存放微程序的存储器，简称控存。
  - 执行一条指令实际上就是执行一段存放在控制存储器中的微程序。
- 控制存储器与主存储器的区别

	控制存储器	主存储器
位置	CPU内	CPU外
器件	ROM	RAM和ROM
内容	微程序、微指令	程序、指令和数据

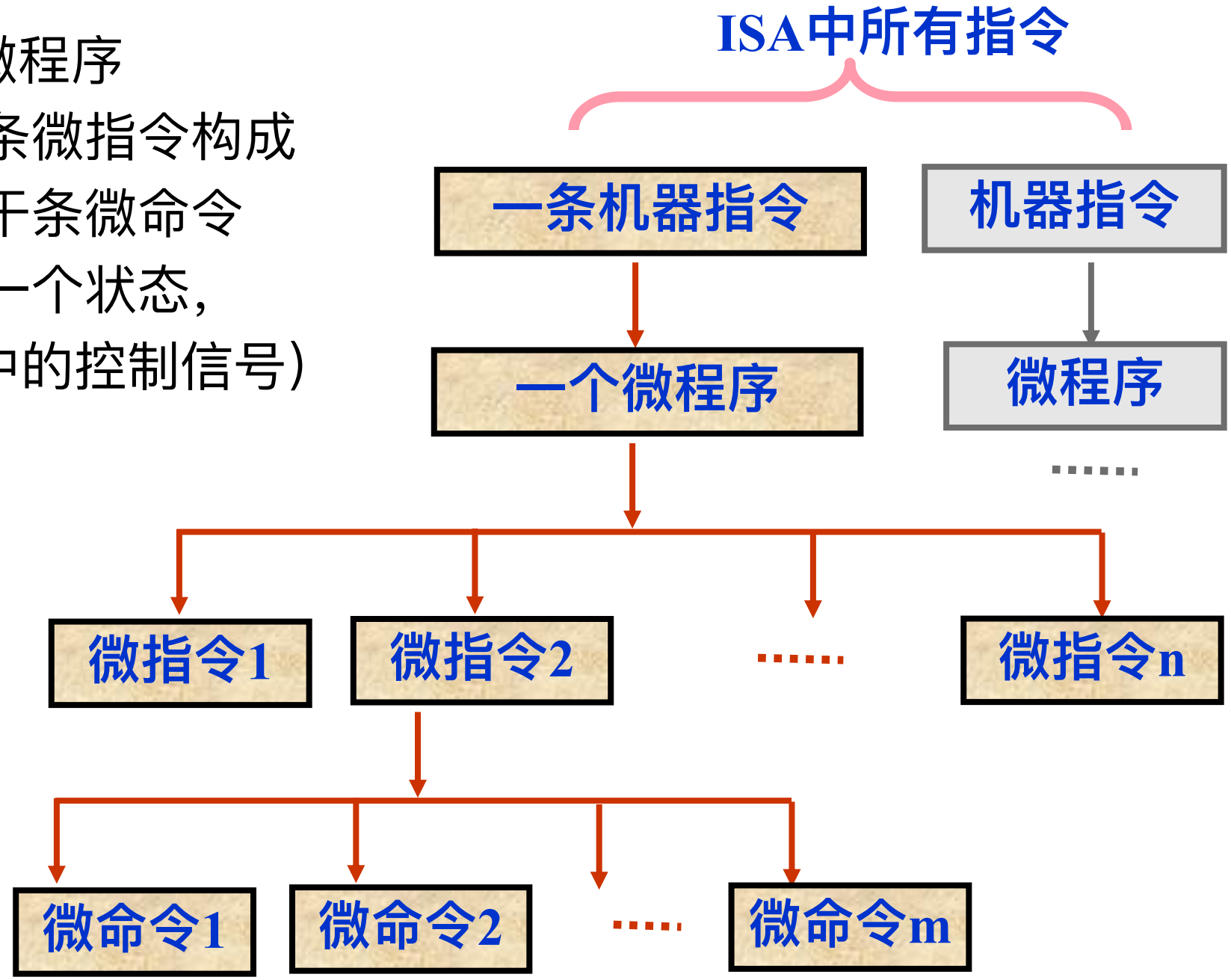
微程序 CPU


指出将要执行的微指令在CS中的位置
Next State
μP
C
Decoder
Microcode
ROM (holds fixed
μcode instructions)
Busy?
Opcode
Condition
Control
Lines
Datapath
Address
Data
Main Memory
(holds user program written in macroinstructions, e.g., x86, RISC-V)

微程序设计思想

- 每个指令对应一个微程序
- 每个微程序由若干条微指令构成
- 每个微指令包含若干条微命令

(一条微指令相当于一个状态，一个微命令就是状态中的控制信号)

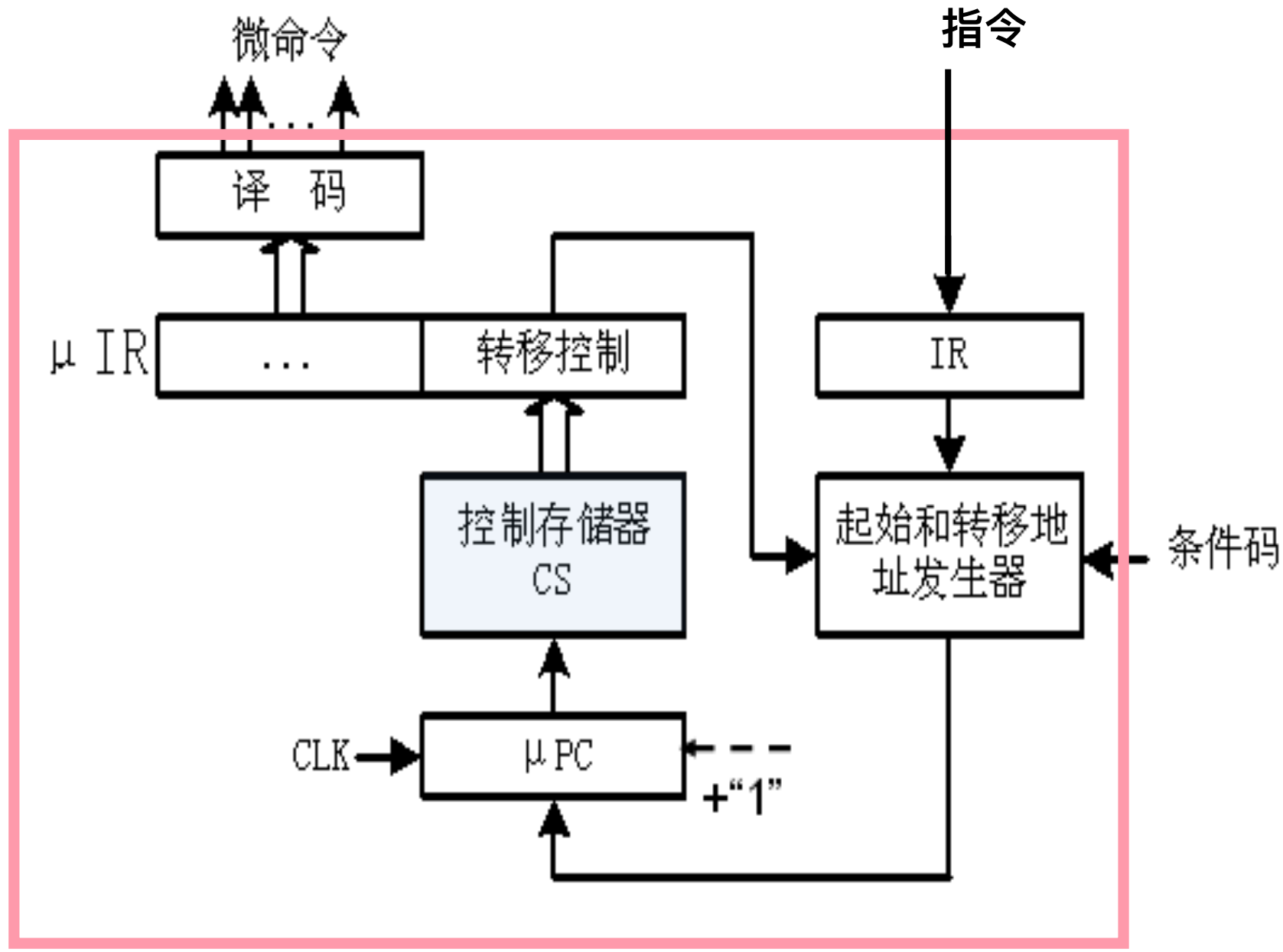


所有指令对应的微程序放在只读存储器(控制存储器Control Storage，简称控存CS)

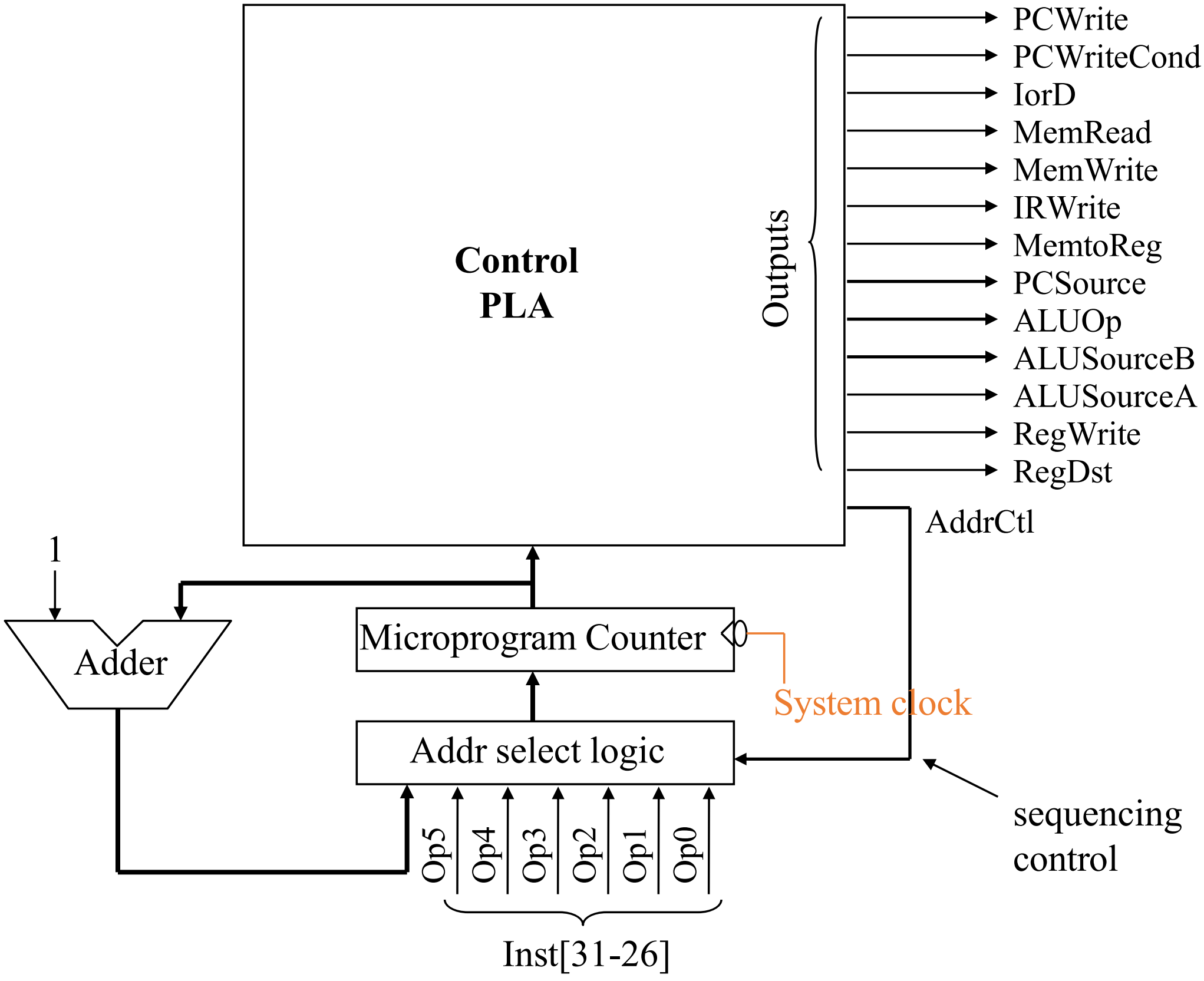
执行某条指令时，取出对应微程序中的各条微指令，对微指令译码产生微命令(控制信号)。

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，由控制转移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



Microcode Implementation



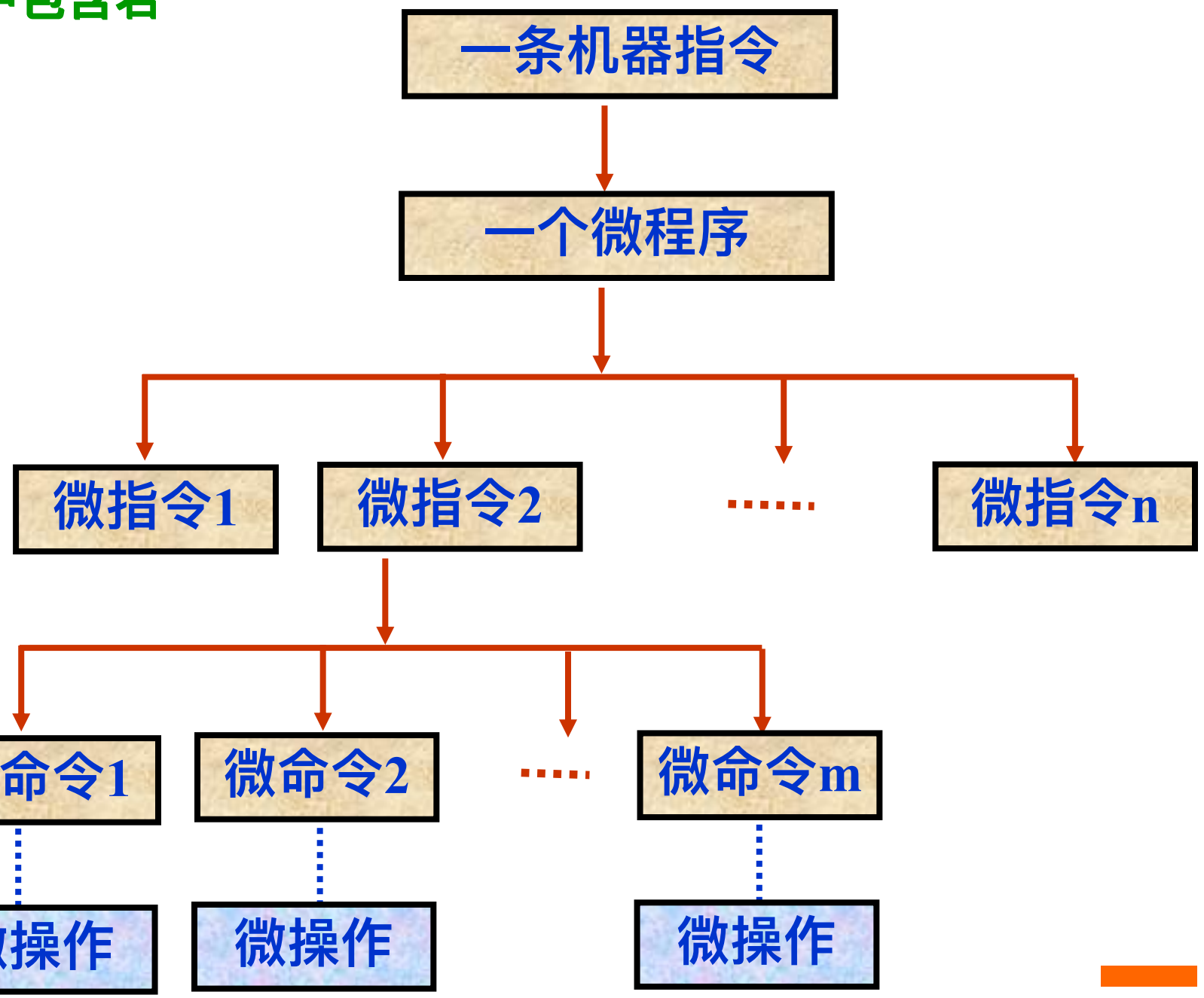
微程序\微指令\微命令\微操作的关系

将指令的执行转换为微程序的执行

微程序是一个微指令序列

每条微指令是一个0/1序列，其中包含若干个微命令（即：控制信号）

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

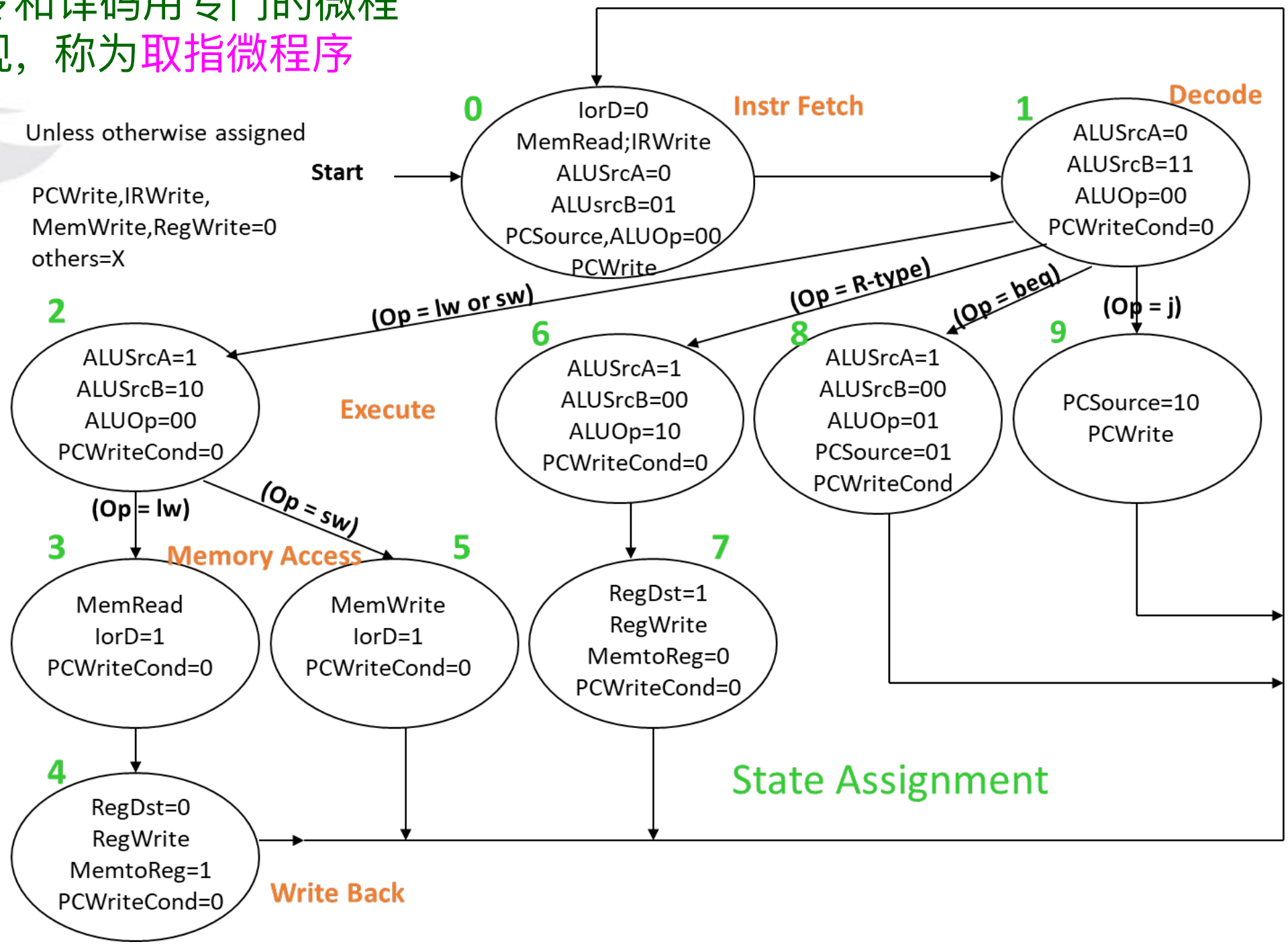
(2)下条微指令在哪里

每条指令用一个微程序实现

下图需要用几个微程序？ 6个+1个

取指令和译码用专门的微程序实现，称为取指微程序

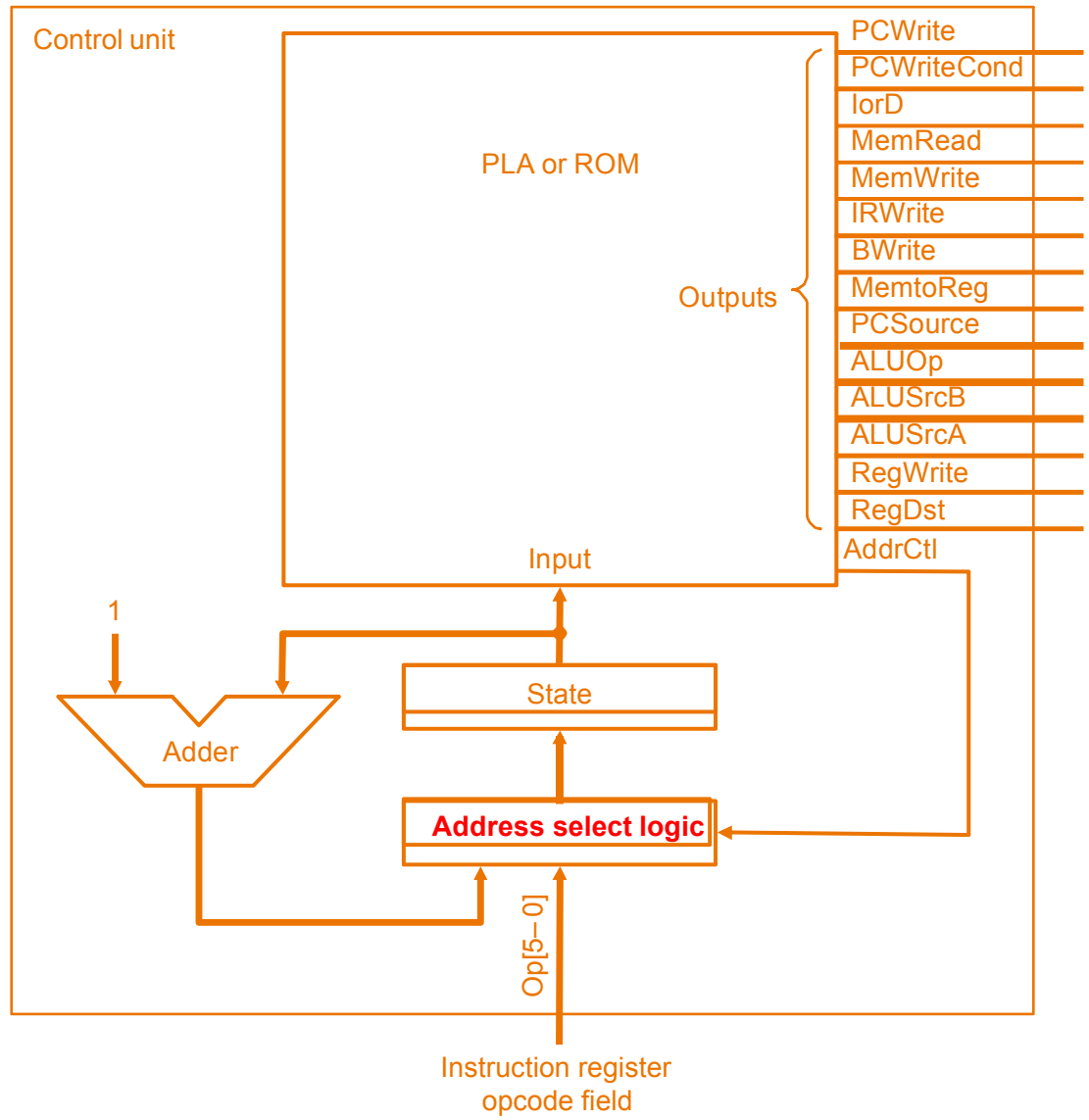
每个微程序由若干微指令组成，每个状态可以恰好对应一条微指令(也可以对应多条)



上述取指微程序包含几条微指令？ 2条 lw指令有几条微指令？ 3条

微程序

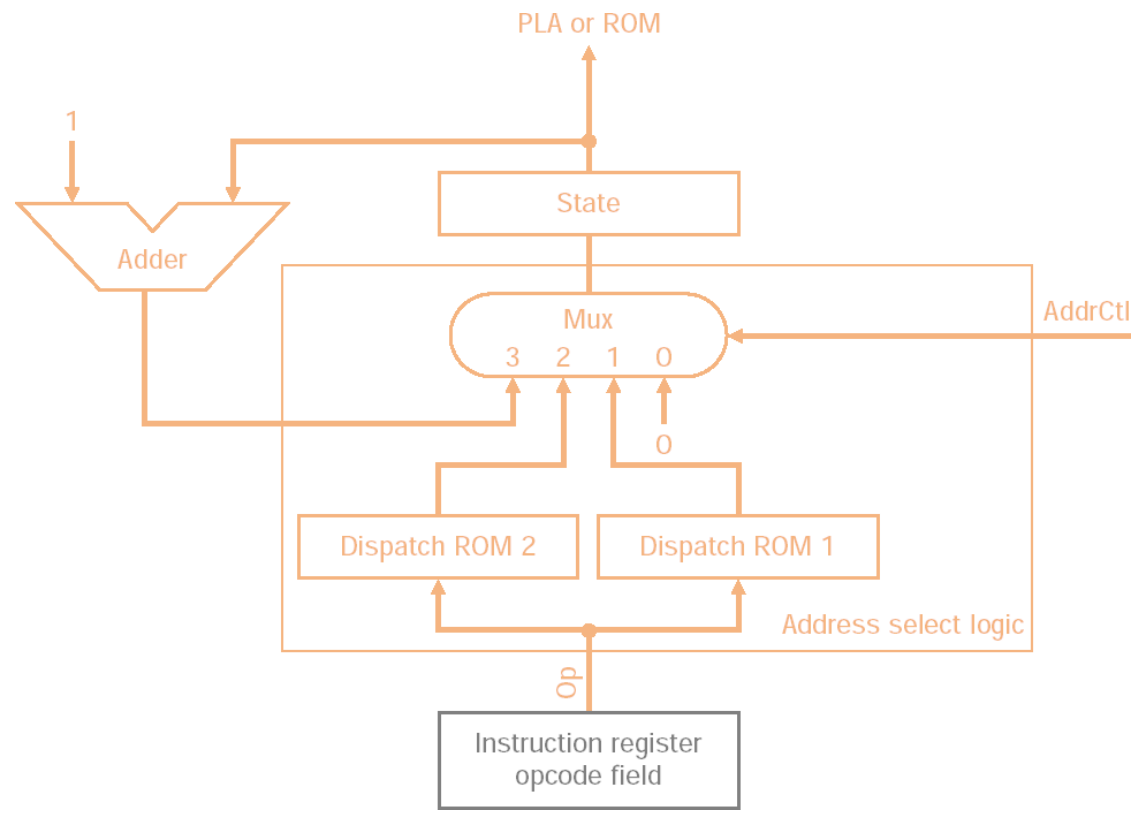
- “下一个状态” 一般是当前状态+1
- “下一条指令” 也常常是当前指令地址 + 1



微指令地址选择逻辑

Dispatch ROM 1		
Op	**Opcode name**	**State Value**
000000	R-format	0110
000010	`jmp`	1001
000100	`beq`	1000
100011	`lw`	0010
101011	`sw`	0010

Dispatch ROM 2		
Op	**Opcode name**	**State Value**
100011	`lw`	0011
101011	`sw`	0101



State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

微指令

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

每个状态是一条微指令

每条微指令是一个0/1序列，其中包含若干个微命令（即：控制信号）

微命令控制数据通路的执行

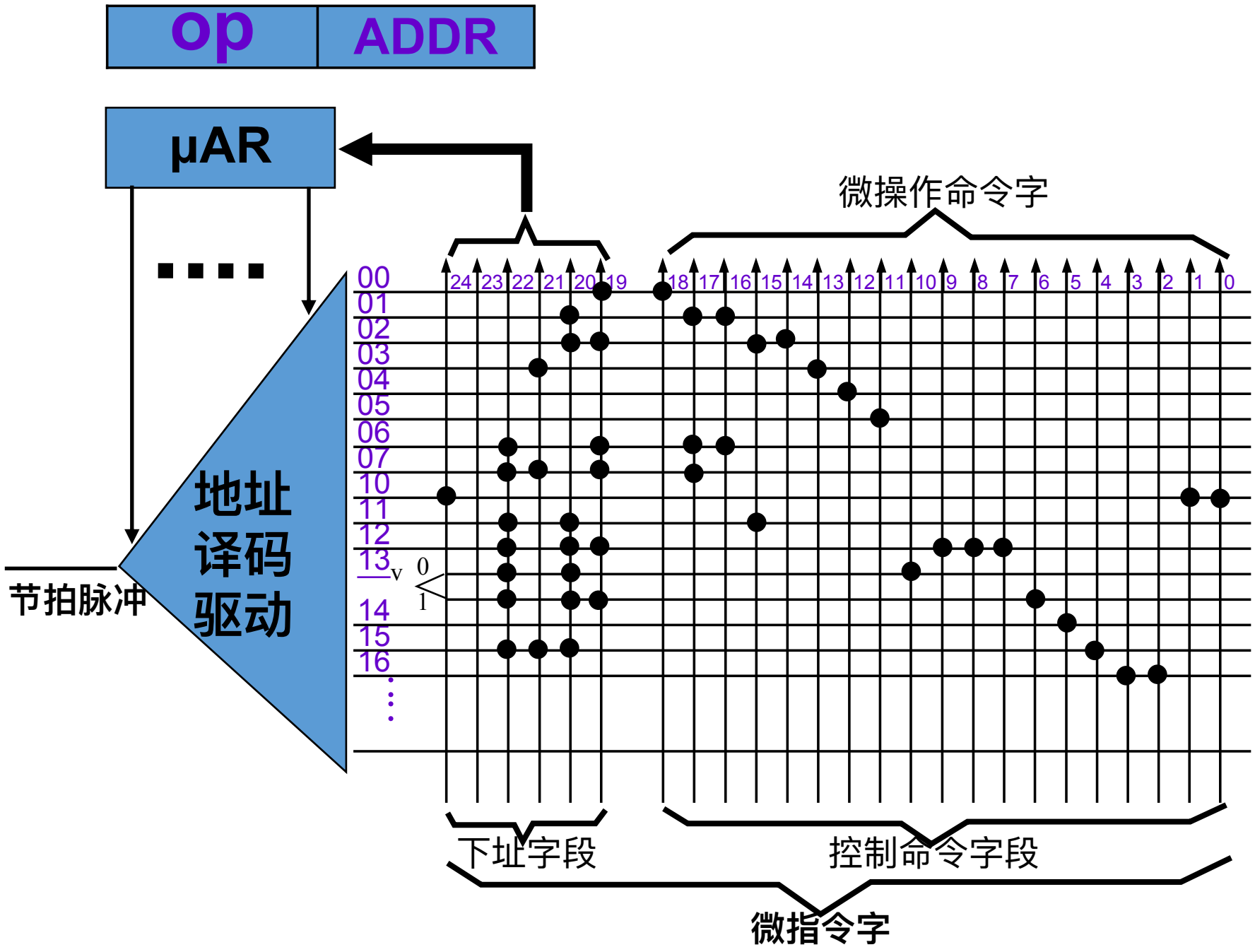
微指令的格式

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, IorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, IorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, IorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

微程序控制器设计的一般步骤

- 根据指令系统，列出微操作序列
- 微指令编码
- 控制微程序流
- 确定指令格式
- 微程序写入控制存储器

写入存储器示意图


op
ADDR
μAR
微操作命令字
24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
00
01
02
03
04
05
06
07
10
11
12
13
14
15
16
地址
译码
驱动
节拍脉冲
v 0
1
下址字段
控制命令字段
微指令字

意外和中断(Exceptions and Interrupts)

°控制是设计中的难点

°控制部分最难的是意外和中断

- 除了转移和跳转之外,可以改变指令执行的正常流动的事件
- 意外是来自处理器内部的不可预知的事件,例如,算术溢出
- 中断 是来自处理器外部的不可预知的事件,例如, I/O

°MIPS的约定: 意外(exception)意味着任何改变控制流的不可预知的事件,它不区分外部和内部;
当外部原因导致的事件出现时,才使用术语“中断(interrupt)”

事件类型	*来自哪里?*	*MIPS的术语*
I/O device request	External	Interrupt
Invoke OS from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or Interrupt

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

    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、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。

    (外部中断是一种I/O方式)

异常和中断的处理

内部“异常” 按处理方式分为故障、自陷和终止三类

故障(fault)： 执行指令引起的异常事件，如溢出、缺页、堆栈溢出、访问超时等。 **“断点”为发生故障指令的地址**

自陷(Trap)： 预先安排的事件，如单步跟踪、系统调用(执行访管指令)等。是一种自愿中断。 **“断点”为发生下条指令的地址**

终止(Abort)： 硬故障事件，此时机器将“终止”，调出中断服务程序来重启操作系统。 **“断点”是什么？ 随便！**

思考1：自陷处理完成后回到哪条指令执行？ **回到下条指令！**

思考2：哪些故障补救后可继续执行，哪些只好终止当前进程？

缺页、TLB缺失等：补救后可继续，回到发生故障的指令重新执行；

溢出、除数为0、非法操作、内存保护错等：终止当前进程。

- **不同体系结构和教科书对“异常”和“中断”定义的内涵不同，在看书时要注意！**

本章主要介绍如何在数据通路中增加“程序性异常”的检测和处理逻辑。

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器中。即：

PC→堆栈 或 EPC（专门存放断点的寄存器）

PSWR →堆栈 或 EPSWR （专门保存程序状态的寄存器）

PSW（Program Status Word）：程序状态字，包括条件码、中断码、状态位等。

PSWR（PSW寄存器）：用于存放程序状态字的寄存器。如，X86的FLAGS）。

③ 识别异常事件：有软件识别和硬件识别（向量中断方式）两种不同的方式。

处理器中的异常处理机制

有两种不同的识别方式：

（1）软件识别（MIPS采用）

设置一个异常状态寄存器（MIPS中为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：MIPS中位于内核地址0x8000 0180处有一个专门的异常处理程序，用于检测异常的具体原因，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）

用专门的硬件查询电路按优先级顺序识别异常，得到“中断类型号”，根据此号，到中断向量表中读取对应的具体的中断服务程序的入口地址。

举例-8086/8088中断系统（采用向量中断方式）

统称为“中断”：内中断（内部异常）和外中断（外部中断）

- 内中断：CPU自己产生而不通过中断请求线请求，皆为不可屏蔽中断。
  - 中断指令引起的异常：CPU执行预置的指令后在特定的情况下发生的异常。

    INTO 溢出：执行算术指令后，若发生溢出，则产生类型4中断。

    INT n 用户定义：指令的第二字节给出一个类型号(n=0～255)。

    其中n=3 (INT 3)时为断点设置，该指令执行后，自动产生类型3中断。
  - 处理器检测异常：CPU执行指令时产生的异常，如：除法错、无效操作码、缺页、单步跟踪调试等。如：

    除法错：除数为0或商溢出，则产生类型0中断。

    单步跟踪：当自陷位TF=1且处在开中断状态(即IF=1)时，每条指令执行完就自动产生类型1中断。
- 外中断：通过中断请求线INTR和NMI来实现。
  - INTR：可屏蔽中断 (外设中断源引起的中断)。
  - NMI：不可屏蔽中断 (重要或紧急的硬件故障)，属于类型2中断。

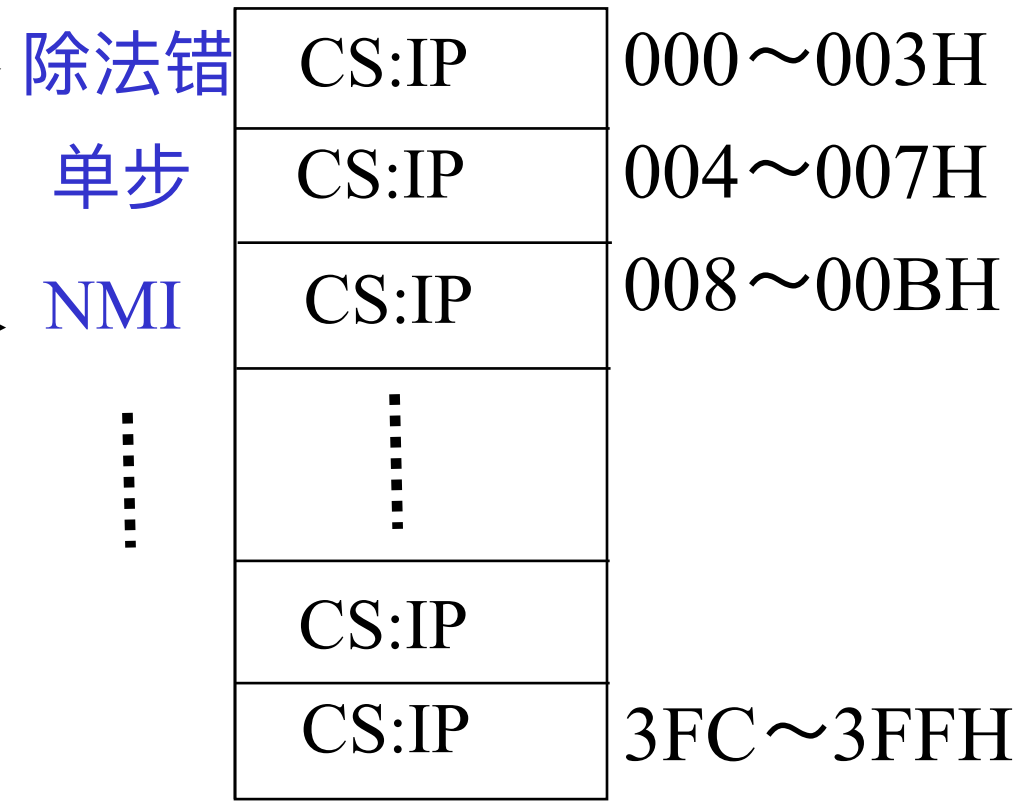
所有事件都被分配一个“中断类型号”，每个中断都有相应的“中断服务程序”，可根据中断类型号找到中断服务程序的入口地址

8086/8088的中断向量表

中断向量表也称中断入口地址表（或异常表），位于0000H～03FFH。
共256组，每组占四个字节 CS:IP 。向量地址=中断类型号 x 4

例1：除法错的中断类型号为0，故其向量地址为：0x4=0

例2：NMI的中断类型号为2，故其向量地址为：2x4=8



- 中断向量表（异常表）中每一项是对应中断服务程序的入口地址，被称为中断向量(Interrupt Vector)。
- 中断向量表的起始地址存放在一个异常表基址寄存器中。

这是最简单的向量中断方式，用以说明其基本原理，x86的实际情况要复杂得多，后面相关课程会有说明。

MIPS带异常处理的数据通路设计

- MIPS采用软件（操作系统提供的一个特定的异常查询程序）识别中断源
- 数据通路中需增加以下两个寄存器：
  - 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，记录异常原因。供异常查询程序用
    - 假定处理的异常类型有以下两种：

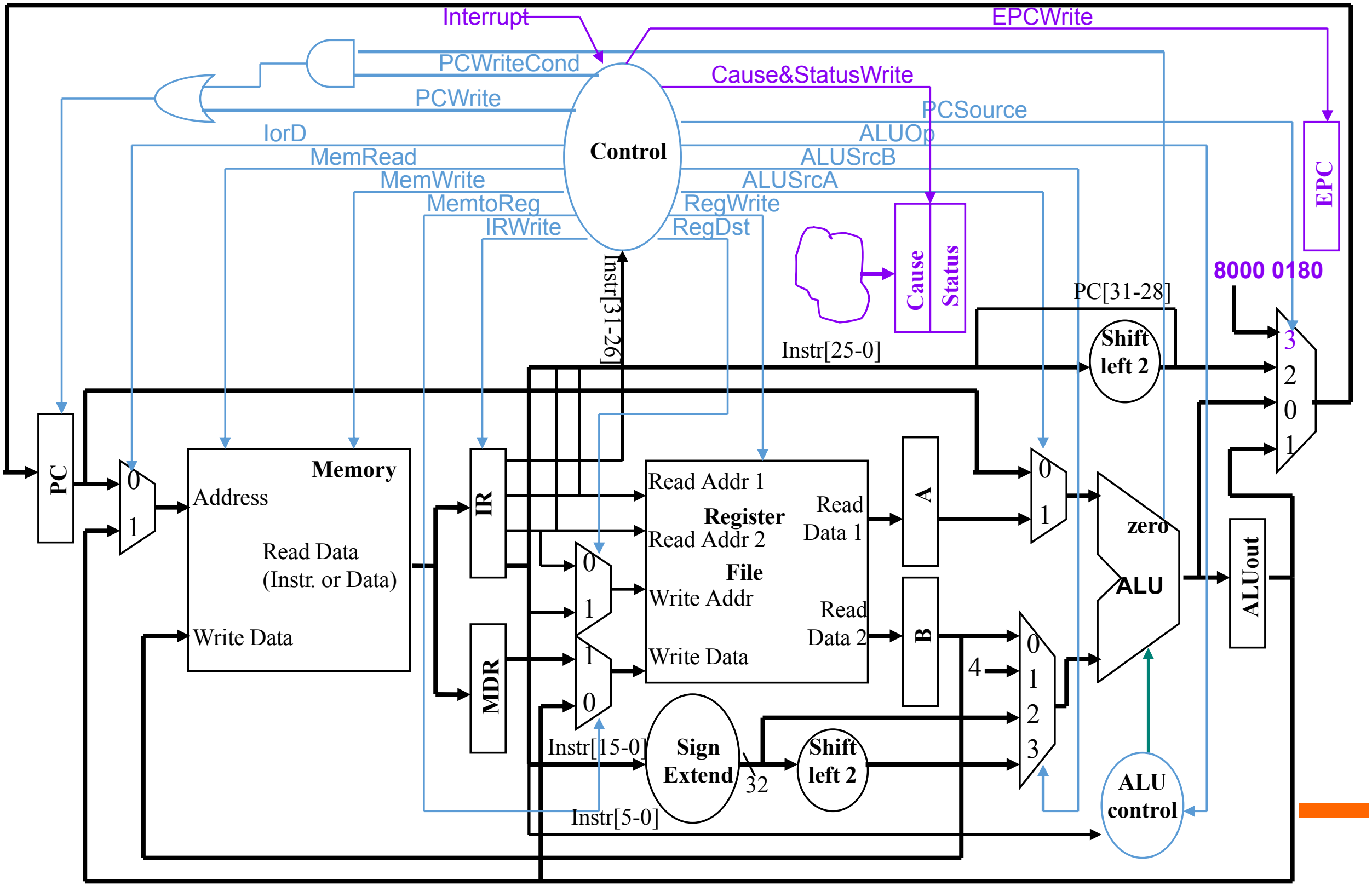
      未定义指令（Cause=0）、数据溢出（Cause=1）

- 需加入两个寄存器的“写使能”控制信号

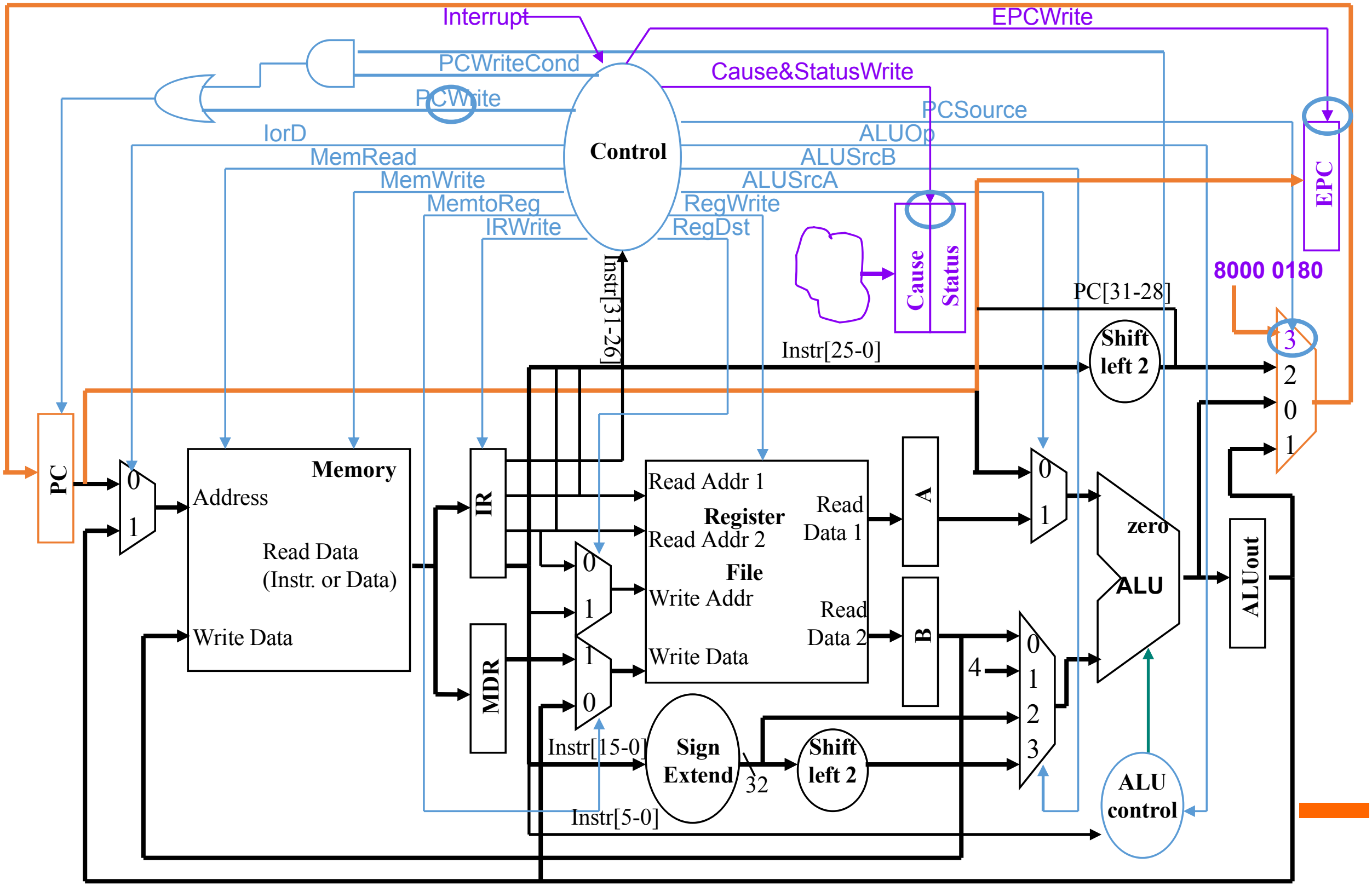
  EPCWr：保存断点时该信号有效，使断点PC写入EPC

  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
- 需一个控制信号IntCause来选择正确的值写入到Cause中
- 需将异常查询程序入口地址（MIPS为0x8000 0180）写入PC， 可在原PCSource控制的多路器中再增加一路，其输入为 0x8000 0180。

Interrupt Modified Multicycle Datapath

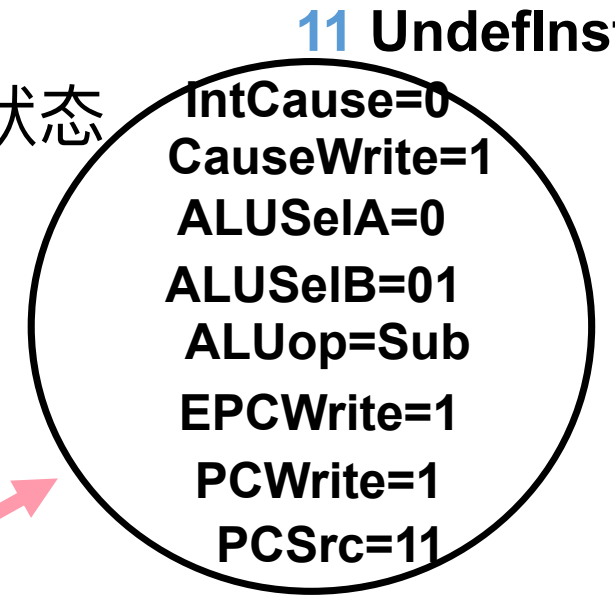


Interrupt Modified Multicycle Datapath

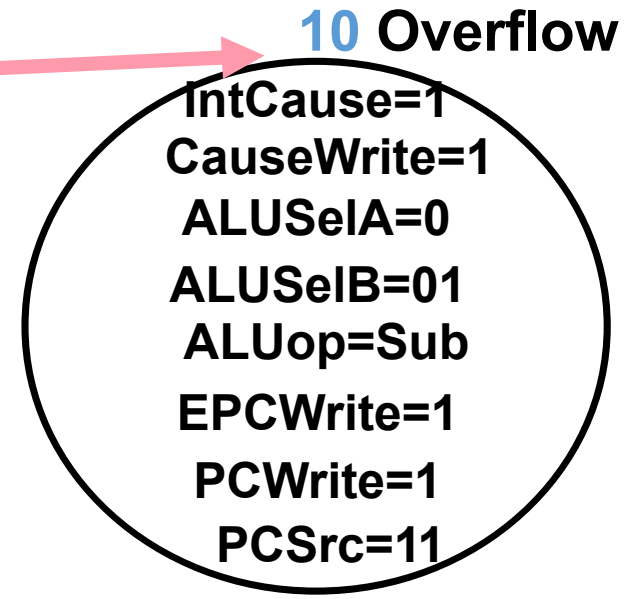


带异常处理的控制器设计

- 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
  - 将中断允许位清0（关中断）
- 假设要控制的数据通路中有以下两种异常处理
  - 未定义指令（Cause=0）：状态11
  - 数据溢出（Cause=1）：状态10



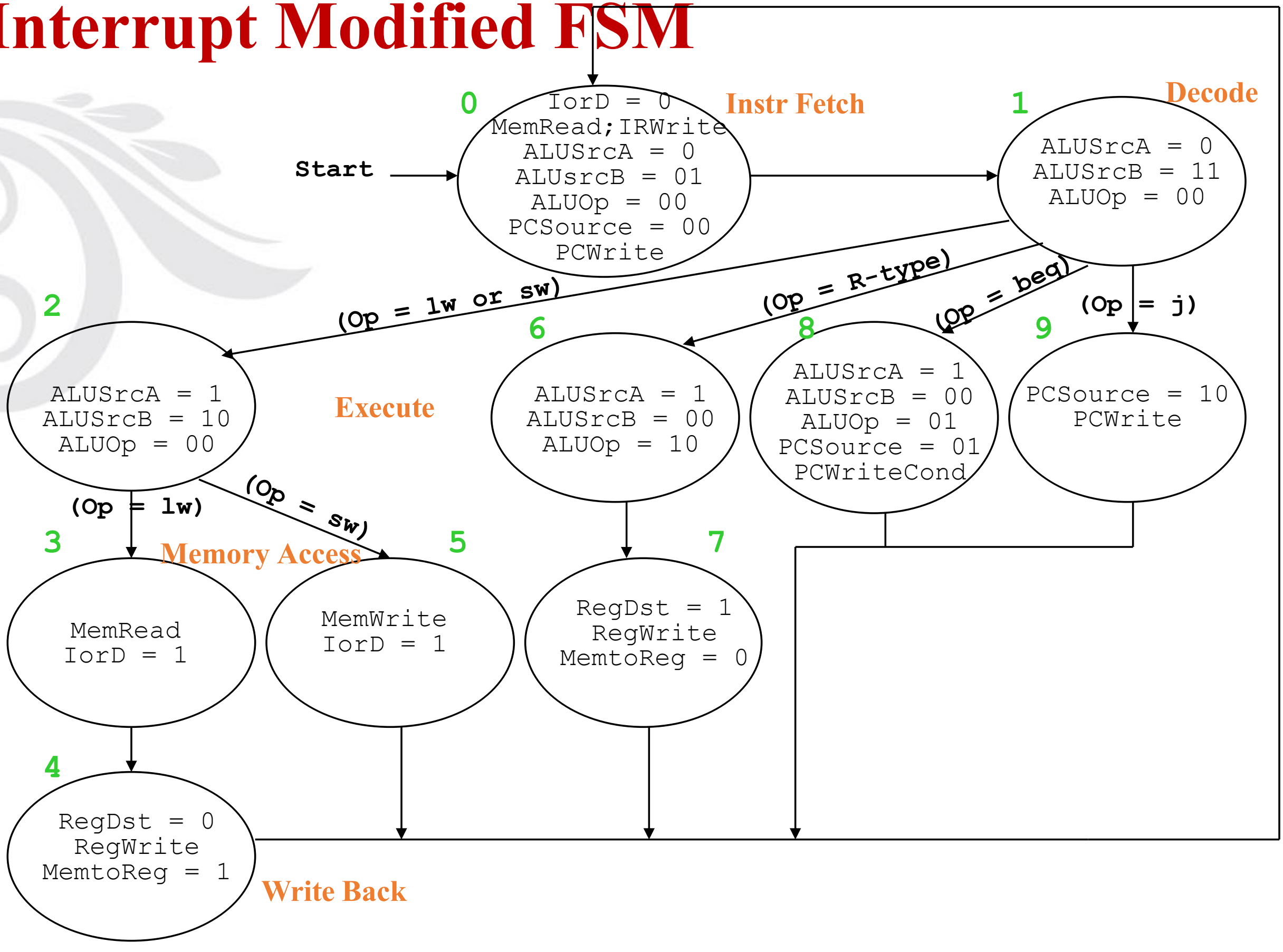
11未定义指令异常状态



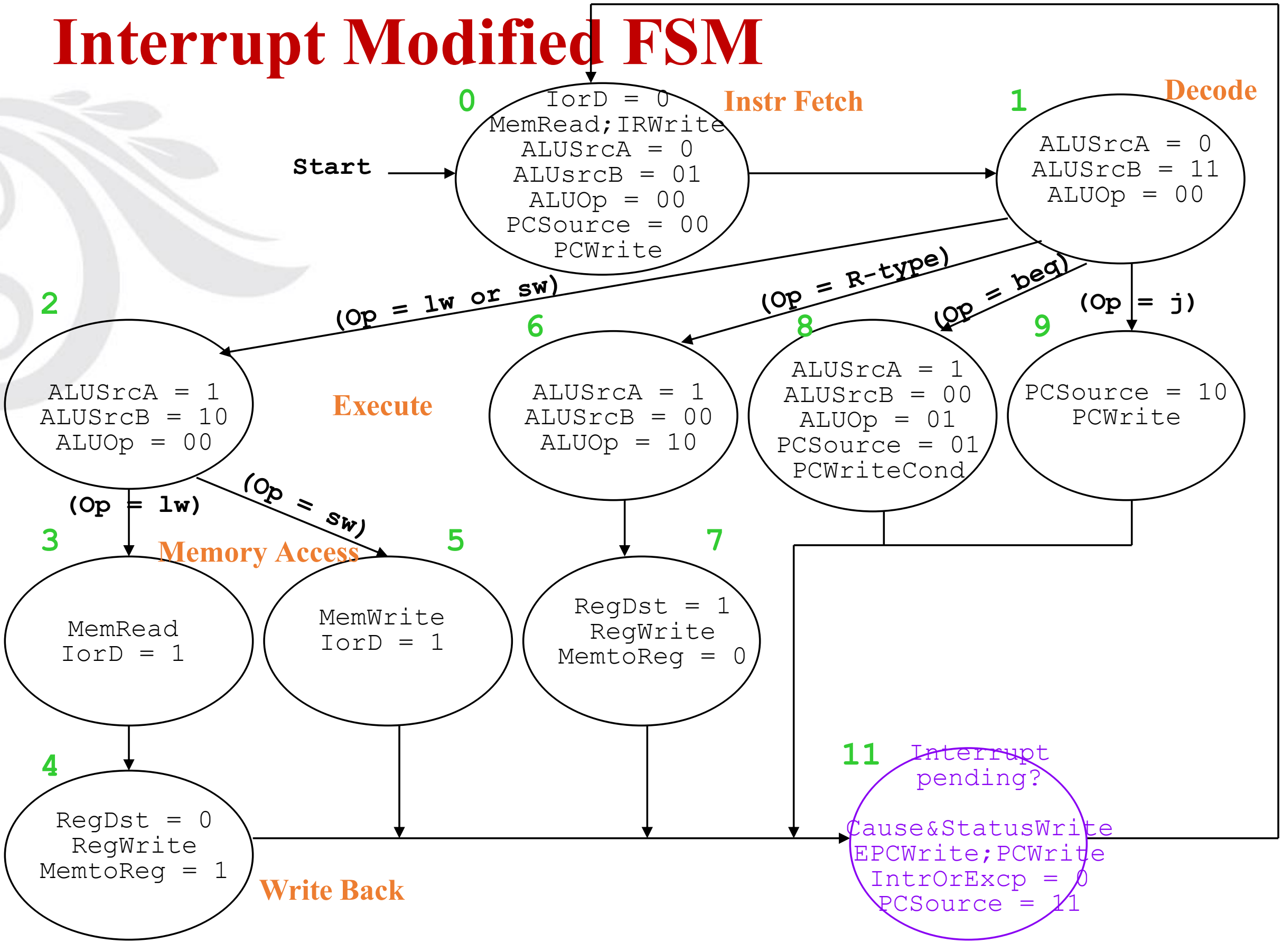
10 数据溢出异常状态

- 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：当指令译码器发现op字段是一个未定义的编码时
    - 数据溢出：当R-Type指令执行后在ALU输出端的Overflow为1时

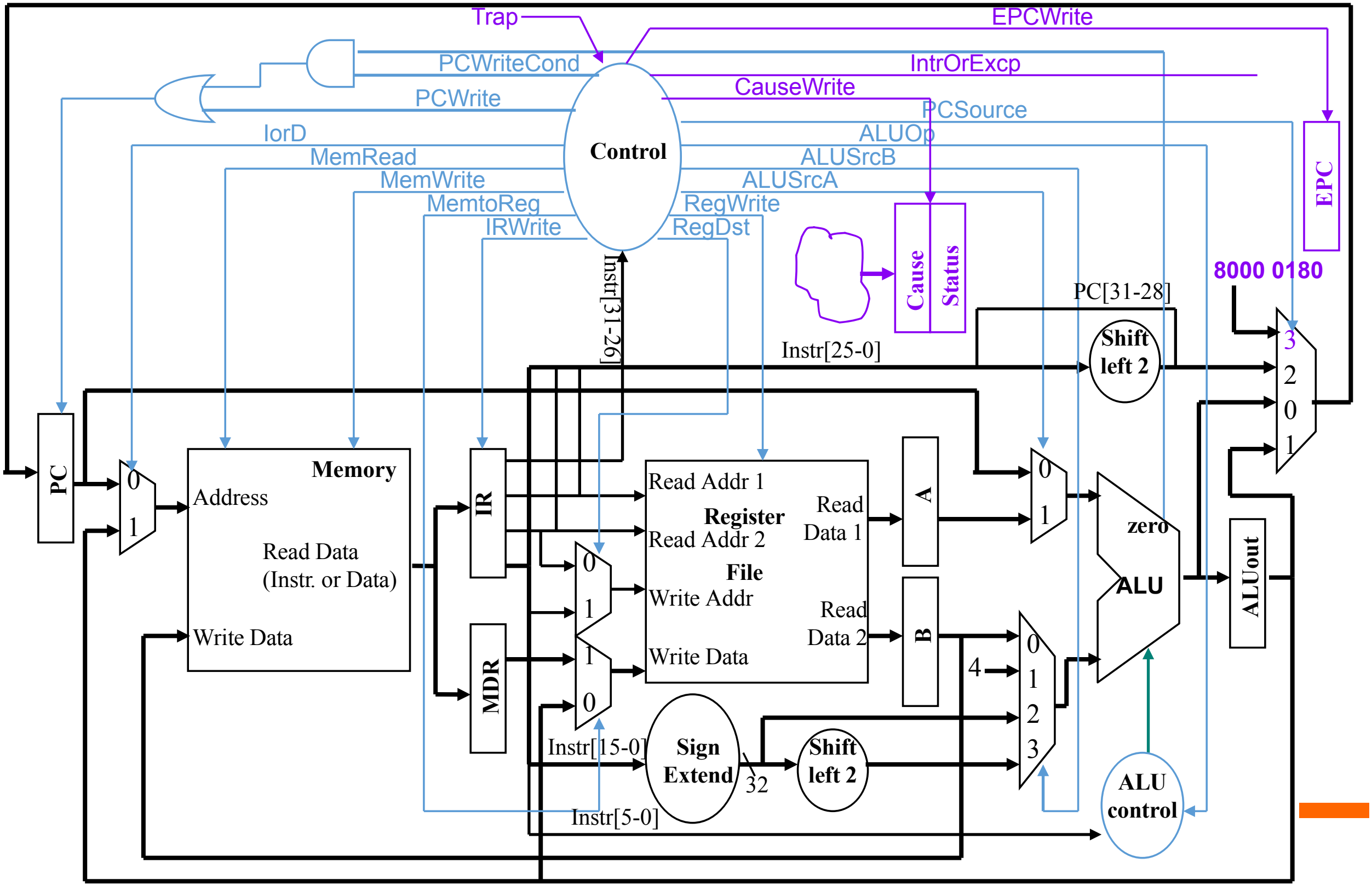
Interrupt Modified FSM



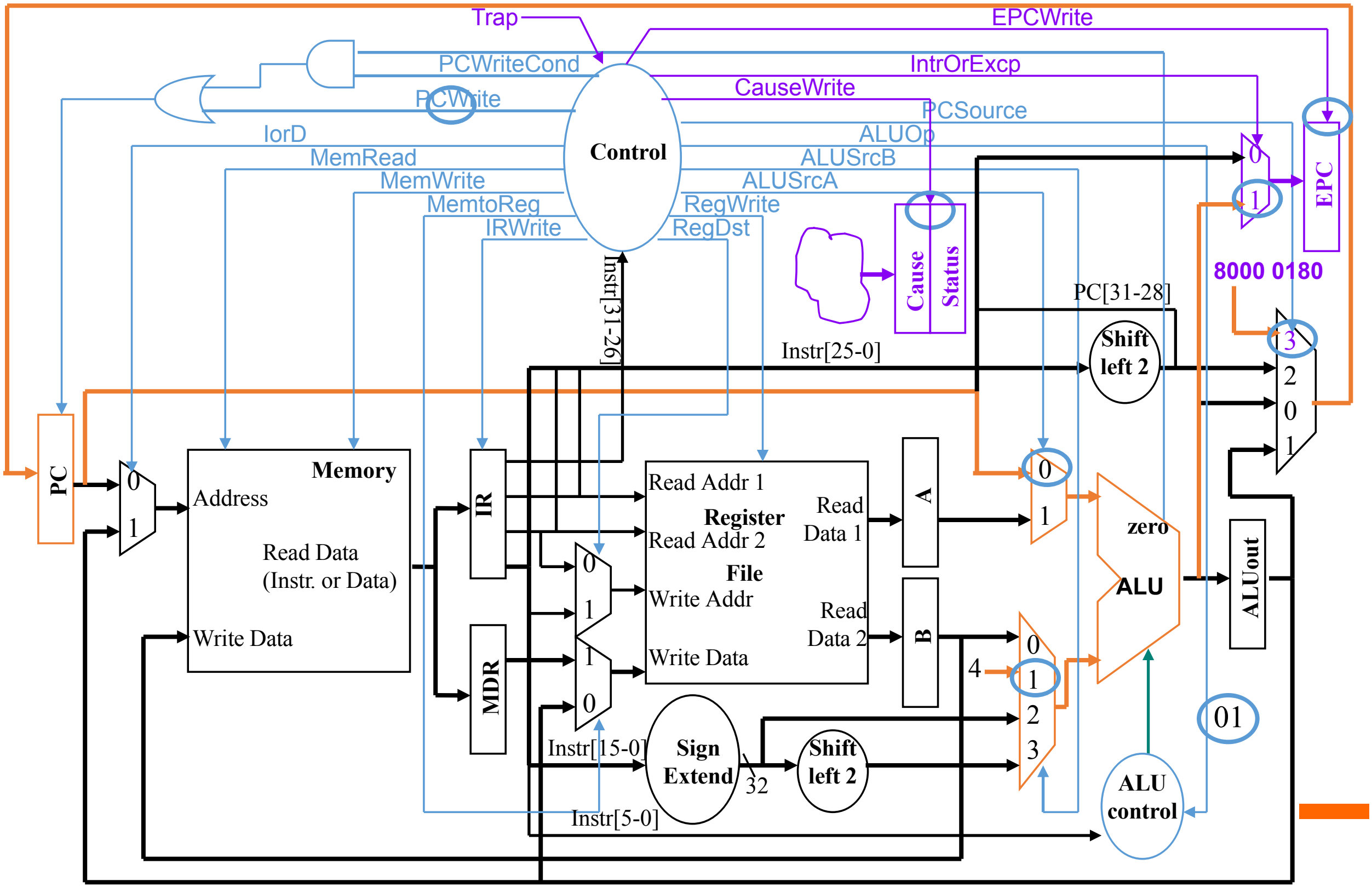
Interrupt Modified FSM



Trap Modified Multicycle Datapath



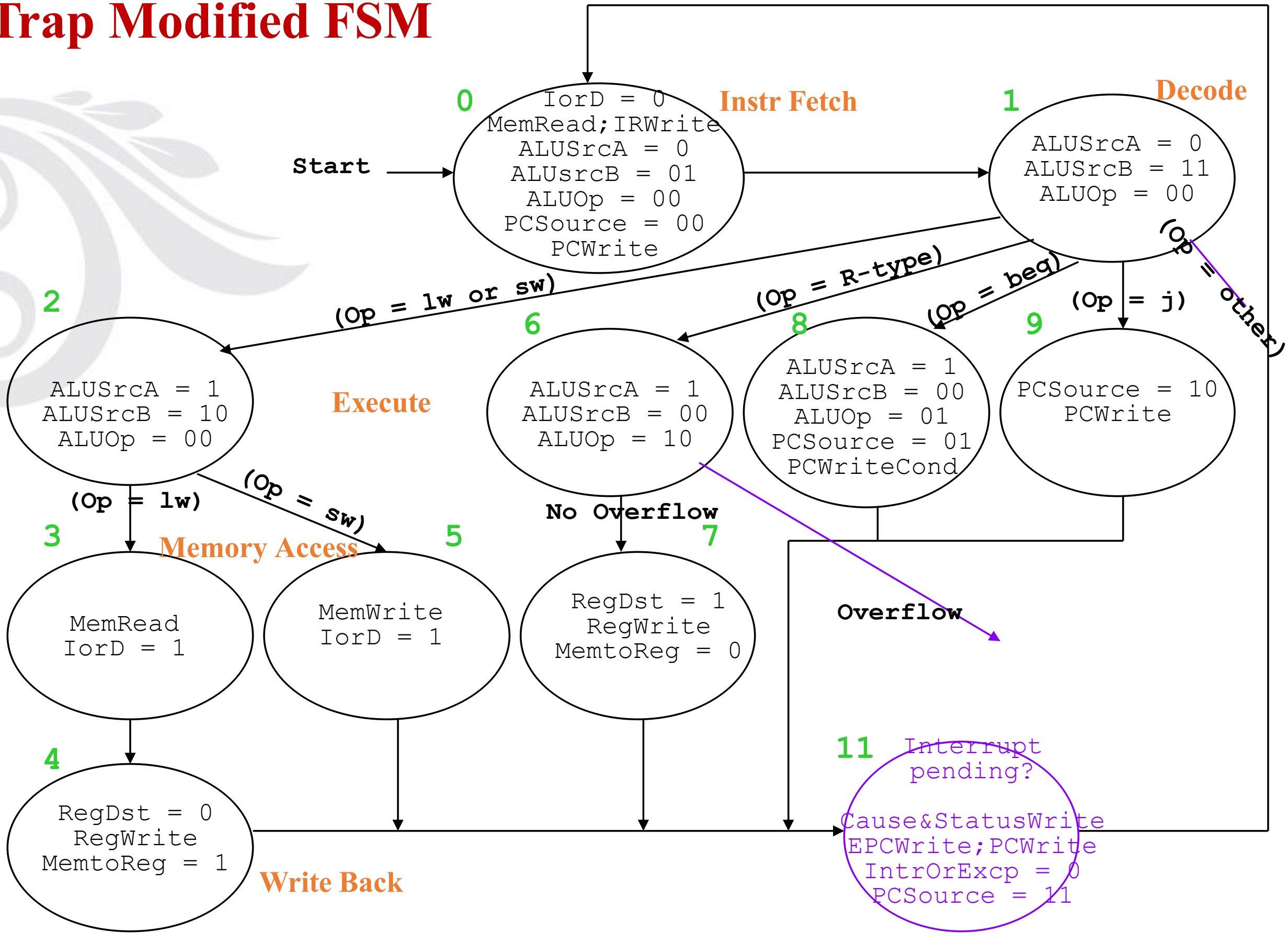
Trap Modified Multicycle Datapath



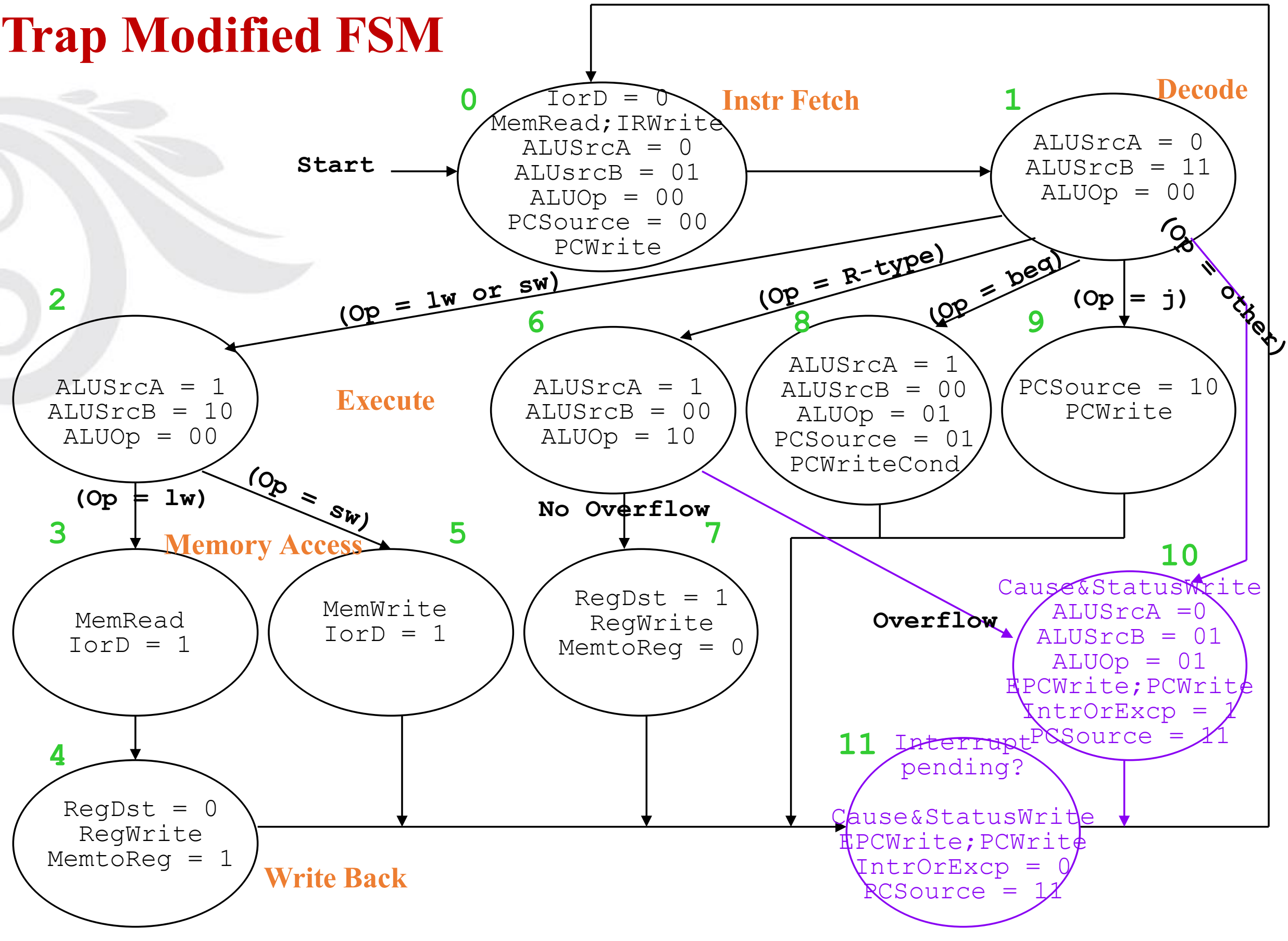
How Control Detects Two Traps

- 未定义指令(RI) – 状态1检测到下一个状态没有定义，既没有相应的操作码
  - 所有未定义指令的下一个状态为新状态 10
- 算数运算溢出(Ov) – ALU溢出信号在状态6出现 (即不再有结果写入寄存器操作)
- 需要修正状态图
  - 挑战在于处理指令和引发异常的事件之间的交互，以使控制逻辑保持小而快
    - 复杂的交互使控制单元成为硬件设计中最具挑战性的事情，尤其是在流水线处理器中

Trap Modified FSM



Trap Modified FSM



多周期性能分析

- 与单周期CPU对比，分成了多个阶段
  - + 时钟周期减少
  - – **High CPI每个指令所需周期数增加了**
- CPI 取决于指令
  - Branches / Jumps: 3 cycles
  - ALU: 4 cycles
  - Stores: 4 cycles
  - Loads: 5 cycles
  - CPI 算平均CPI，取决于权重
- 比如:
  - 20% loads, 15% stores, 20% branches, 45% ALU

CPI= 0.20 * 5 + 0.15 * 4 + 0.20 * 3 + 0.45 * 4 = **4.0**

多周期性能分析

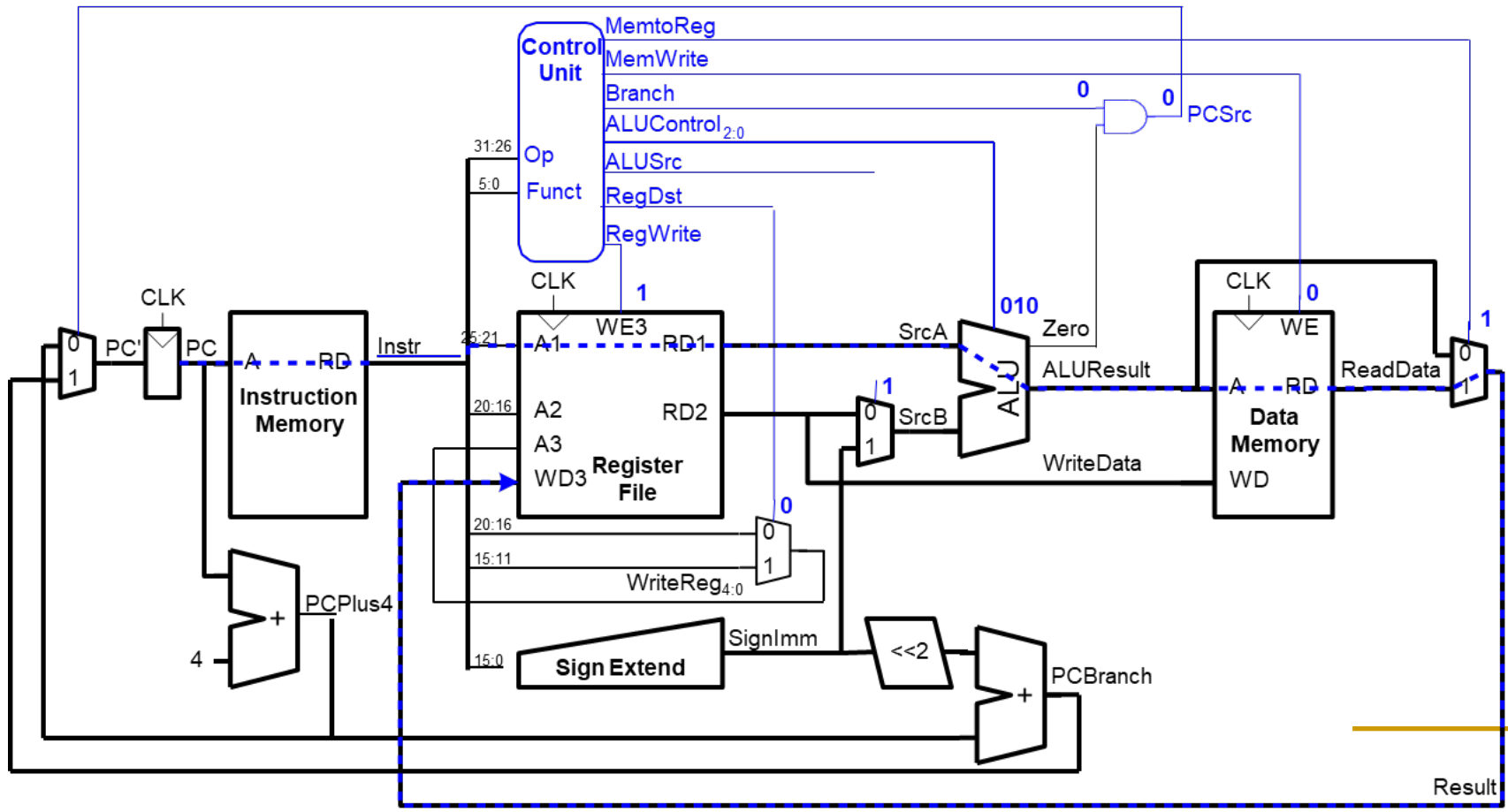
理论上，单周期完成整个五个阶段作为一个周期，应该是5倍的多周期的时钟周期，但假设主要功能部件的操作时间如下表，多周期设计**时钟周期取步骤中最长的200ps，**而单周期CPU是各个阶段所需时间**之和（200+100+100+200+50=650≠800），加上增加的寄存器开销，效率提高有限。**

各类指令执行时间

步骤	R型指令	Lw指令	Sw指令	Beq指令	J指令	执行时间
取指令	IR ← M[PC]，PC ← PC + 4					200ps
读寄存器/译码	A ← R[IR[25:21]]， B ← R[IR[20:16]] ALUOut ← PC + Signext[IR[15:0]]<<2					100ps
计算	ALUOut ← A op B	ALUOut ← A + Signext(IR[15:0])		If (A-B==0) then PC ← ALUout	PC ← PC[31:28] || IR[25:0]<<2	100ps
R型完成/访问内存	R[IR[15:11]] ← ALUOut	DR ← M[ALUOut]	M[ALUOut] ← B			200ps
写寄存器		R[IR[20:16]] ← DR				50ps

单周期CPU性能分析

- 单周期CPU关键路径:
  - $T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$
- 大部分实现中，忽略PC时间，主要考虑:
  - memory, ALU, register file.
  - $T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$



单周期CPU性能分析举例

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$$T_c = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}}$$
$$= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps}$$
$$= 925 \text{ ps}$$

单周期CPU性能分析举例

- Example:

  一个程序有 100 billion 条指令，在单周期CPU上执行:

Execution Time = # instructions x CPI x T_c

= (100 × 10^9)(1)(925 × 10^{-12} s)

= 92.5 seconds

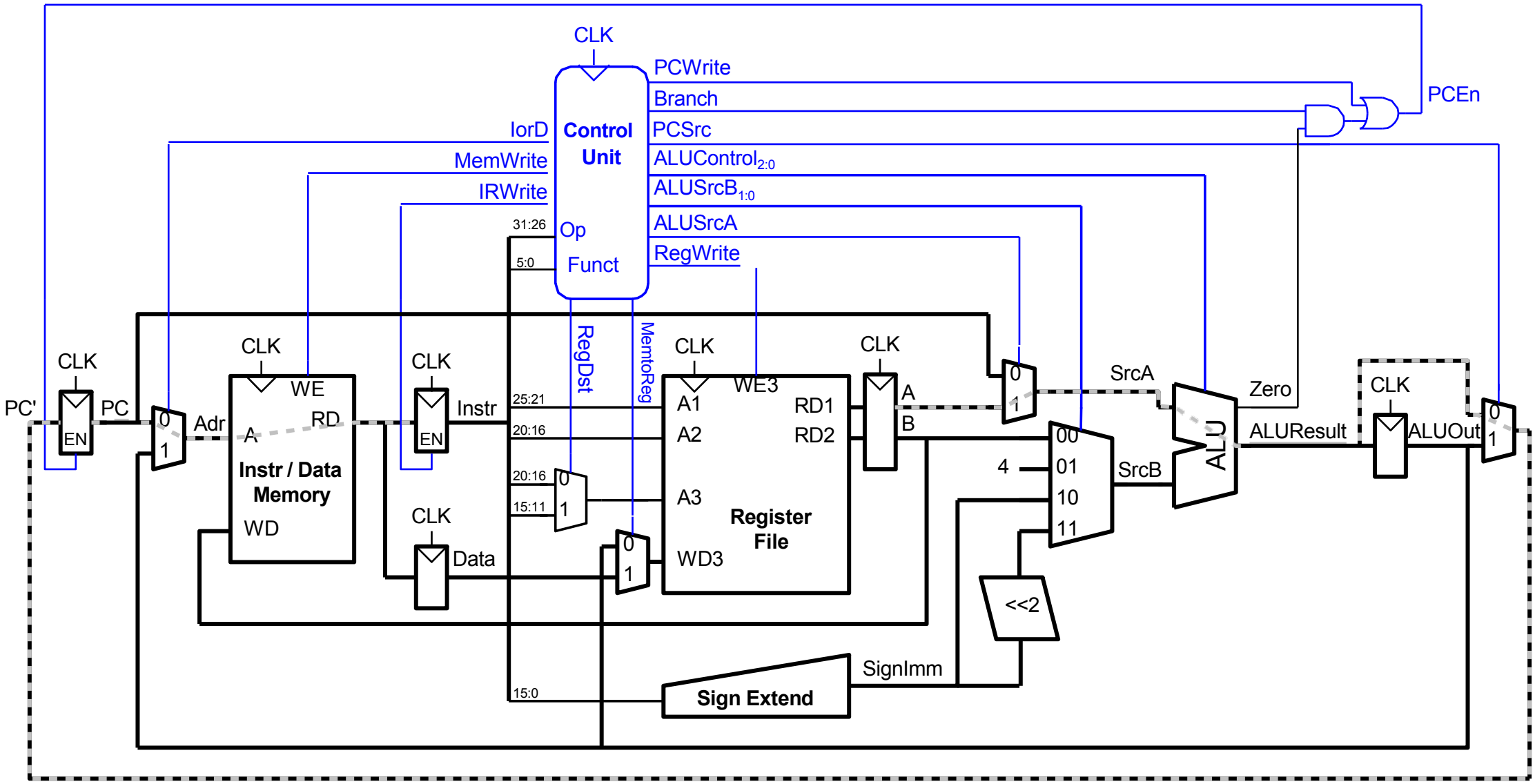
多周期CPU性能分析: CPI

- 不同的指令所需周期数不同:
  - 3 cycles: beq, j
  - 4 cycles: R-Type, sw, addi
  - 5 cycles: `lw` **Realistic?**
- CPI 按权重算平均数, e.g. SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
- *Average CPI* = (0.11 + 0.02) 3 +(0.52 + 0.10) 4 +(0.25) 5
  = 4.12

多周期CPU的时钟周期: Cycle Time

- Multi-cycle critical path:

$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$



多周期CPU举例，确定时钟周期

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$$T_c = t_{pcq\_PC} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$

$$= [30 + 25 + 250 + 20]\ \text{ps}$$

$$= 325\ \text{ps}$$

多周期CPU执行时间

- 程序100 billion 条指令在多周期处理器上运行
  - CPI = 4.12
  - T_c = 325 ps

- *Execution Time* = (# instructions) × CPI × T_c
  = $(100 \times 10^9)(4.12)(325 \times 10^{-12})$
  = 133.9 seconds

- 比单周期上还慢 (单CPU 92.5 seconds). Why?
- 为了平衡选最慢的部件确定时钟周期，而且增加了寄存器开销
- **-----流水线CPU**

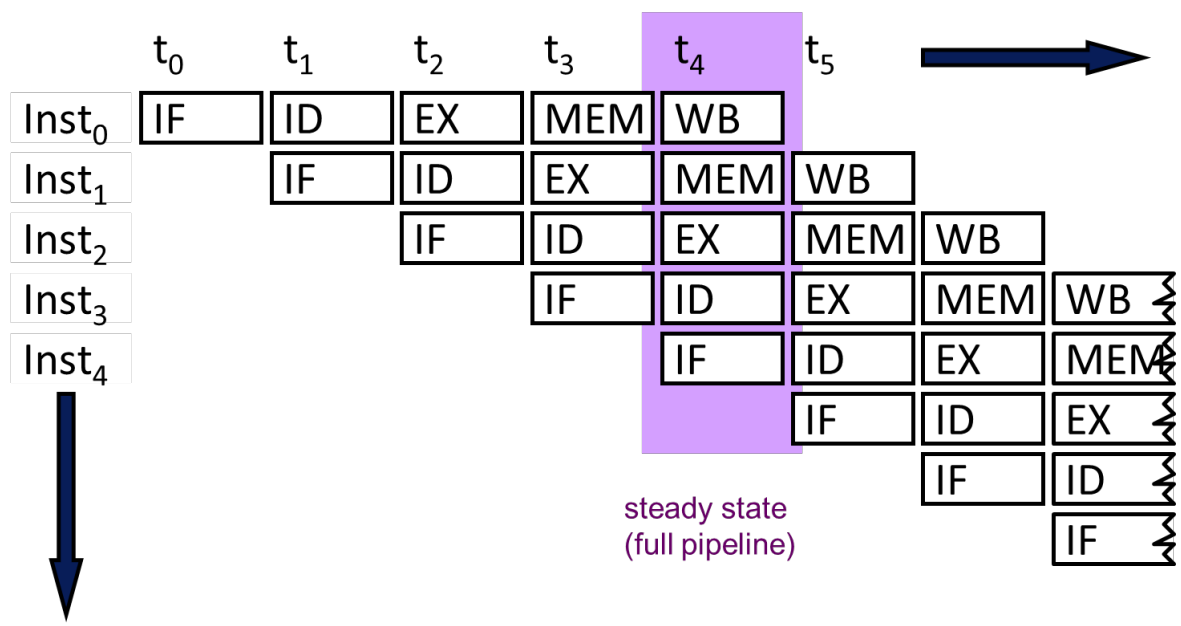
我们可以使用空闲硬件来提高并发性吗?

目标：提高并发性

更高的指令**吞吐量**（即一个周期内完成更多“工作”）

想法：当一条指令在其处理阶段使用某些资源时，在该指令不需要的空闲资源上处理其他指令

- 当一条指令被译码时，获取下一条指令
- 当一条指令被执行时，译码另一条指令
- 当一条指令正在访问数据存储器（lw/sw）时，执行下一条指令
- 当一条指令将其结果写入寄存器文件时，为下一条指令访问数据存储器



```
module CtrlUnit
(
		input Clk,
		input [5:0] op, func,
		output reg[3:0] StatShow,
		output reg [3:0] ALUop,
		output Cin, symbol,
		output reg [1:0] ALUseleB, PCSrc,
		output reg ALUseleA, IorD, PCWr, PCWrCond, IRWr,
MemWr, RegWr, BrWr, EXTOp, MemtoReg, RegDst, move, dir
);
		reg [3:0] State;
		parameter [3:0]
		IFetch=4'b0000,
		RFetch_ID=4'b0001,
		BrFin=4'b0010,
		JumpFin=4'b0011,
		IExe=4'b0100,
		IFin=4'b0101,
		RExe=4'b0110,
		RFin=4'b0111,
		MemAdr=4'b1000,
		swFin=4'b1001,
		MemFetch=4'b1010,
		lwFin=4'b1011;
assign Cin=ALUop[0];
		assign symbol=ALUop[1];
		always @(negedge Clk) begin
  case (State)
	IFetch:
		begin
StatShow=4'b0000;
			IorD=1'b0;
			ALUseleA=1'b0;
			ALUseleB=2'b01;
			PCSrc=2'b01;
			PCWr=1'b1;
			IRWr=1'b1;
			MemWr=1'b0;
			RegWr=1'b0;
			BrWr=1'b0;

State <= RFetch_ID;
			ALUop <= 4'b0000;
			move=1'b0;
		end
	RFetch_ID:
		begin
			ALUop <= 4'b0000;
StatShow=4'b0001;
			EXTOp=1'b1;
			ALUseleA=1'b0;
			ALUseleB=2'b11;
			BrWr=1'b1;
			PCWr=1'b0;
			PCWrCond=1'b0;
			IRWr=1'b0;
			MemWr=1'b0;
			RegWr=1'b0;
			move=1'b0;
```

case(op)

6'b000110://blez
begin
State <= BrFin;
end
6'b000010://j
begin
State <= JumpFin;
end
6'b000000://R
begin
State <= RExe;
end
6'b011100://R
begin
State <= RExe;
end
6'b100011://lw
begin
State <= MemAdr;
end
6'b100010://lwl
begin
State <= MemAdr;
end
6'b100110://lwr
begin
State <= MemAdr;
end
6'b101011://sw
begin
State <= MemAdr;
end
6'b001000://I
begin
State <= IExe;
end
6'b001001://I
begin
State <= IExe;
end
6'b001010://I
begin
State <= IExe;
end
6'b001011://I
begin
State <= IExe;
end
6'b001110://I
begin
State <= IExe;
end
endcase
end

BrFin:
begin
StatShow=4'b0010;
ALUseleA=1'b1;
ALUseleB=2'b00;
PCSrc=2'b10;
PCWrCond=1'b1;
RegWr=1'b0;
PCWr=1'b0;
IRWr=1'b0;
MemWr=1'b0;
BrWr=1'b0;
State <= IFetch;
ALUop <= 4'b0001;
move=1'b0;
end

JumpFin:

```
begin
        StatShow=4'b0011;
        PCSrc=2'b00;
        PCWr=1'b1;
        RegWr=1'b0;
        IRWr=1'b0;
        MemWr=1'b0;
        BrWr=1'b0;
        State <= IFetch;
        move=1'b0;
end
```

IExe:

```
begin
        StatShow=4'b0100;
        ALUseleA=1'b1;
        ALUseleB=2'b10;
        MemtoReg=1'b0;
        RegDst=1'b0;
        RegWr=1'b0;
        PCWr=1'b0;
        PCWrCond=1'b0;
        IRWr=1'b0;
        MemWr=1'b0;
        BrWr=1'b0;
        State <= IFin;
        move=1'b0;
```

case(op)

```
                6'b001110: begin ALUop <= 4'b1001;
EXTOp <= 1'b0; end//xori
                6'b001000: begin ALUop <= 4'b0000;
EXTOp <= 1'b1; end//addi
                6'b001001: begin ALUop <= 4'b1010;
EXTOp <= 1'b1; end//addiu
                6'b001010: begin ALUop <= 4'b0101;
EXTOp <= 1'b1; end//slti
                6'b001011: begin ALUop <= 4'b0111;
EXTOp <= 1'b1; end//sltiu
 endcase
        end
....
```