
Computer Organization and Design Processor

MIPS multicycle processor

[adapted from Mary Jane Irwin slides]

Reading assignment

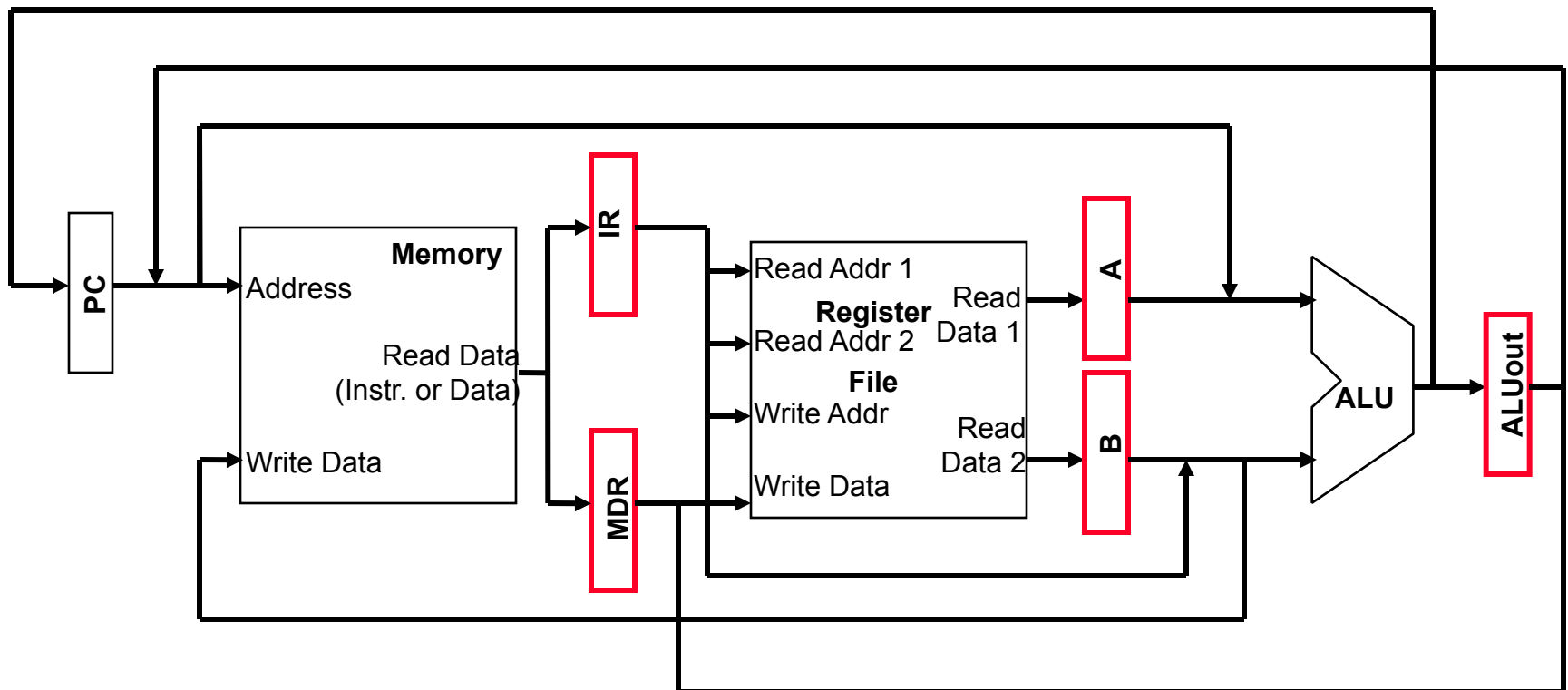
- ❑ Designing a MIPS multicycle processor
 - PH(3): 5.5, 5.7
 - Appendix C The Basics of Logic Design C.10
 - Appendix D Mapping Control to Hardware D.3-D.5

Multicycle Implementation Overview

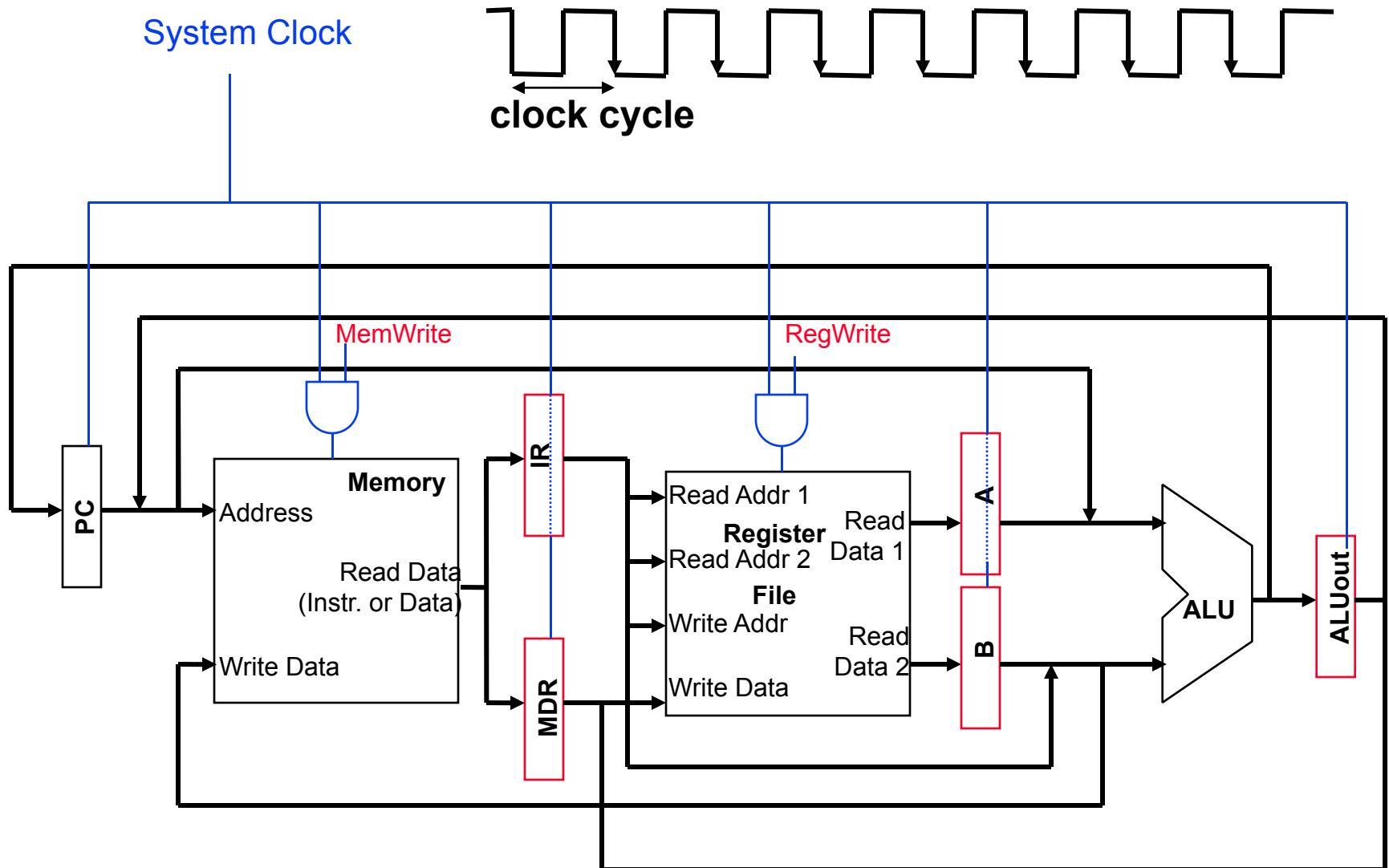
- ❑ Each instruction **step** takes 1 clock cycle
 - Therefore, an instruction takes **more** than 1 clock cycle to complete
- ❑ Not every instruction takes the **same** number of clock cycles to complete
- ❑ Multicycle implementations allow
 - faster clock rates
 - different instructions to take a different number of clock cycles
 - functional units to be used more than once per instruction as long as they are used on different clock cycles, as a result
 - ❑ only need one memory
 - ❑ only need one ALU/adder

The Multicycle Datapath – A High Level View

- ❑ Registers have to be added after every major functional unit to hold the output value until it is used in a subsequent clock cycle



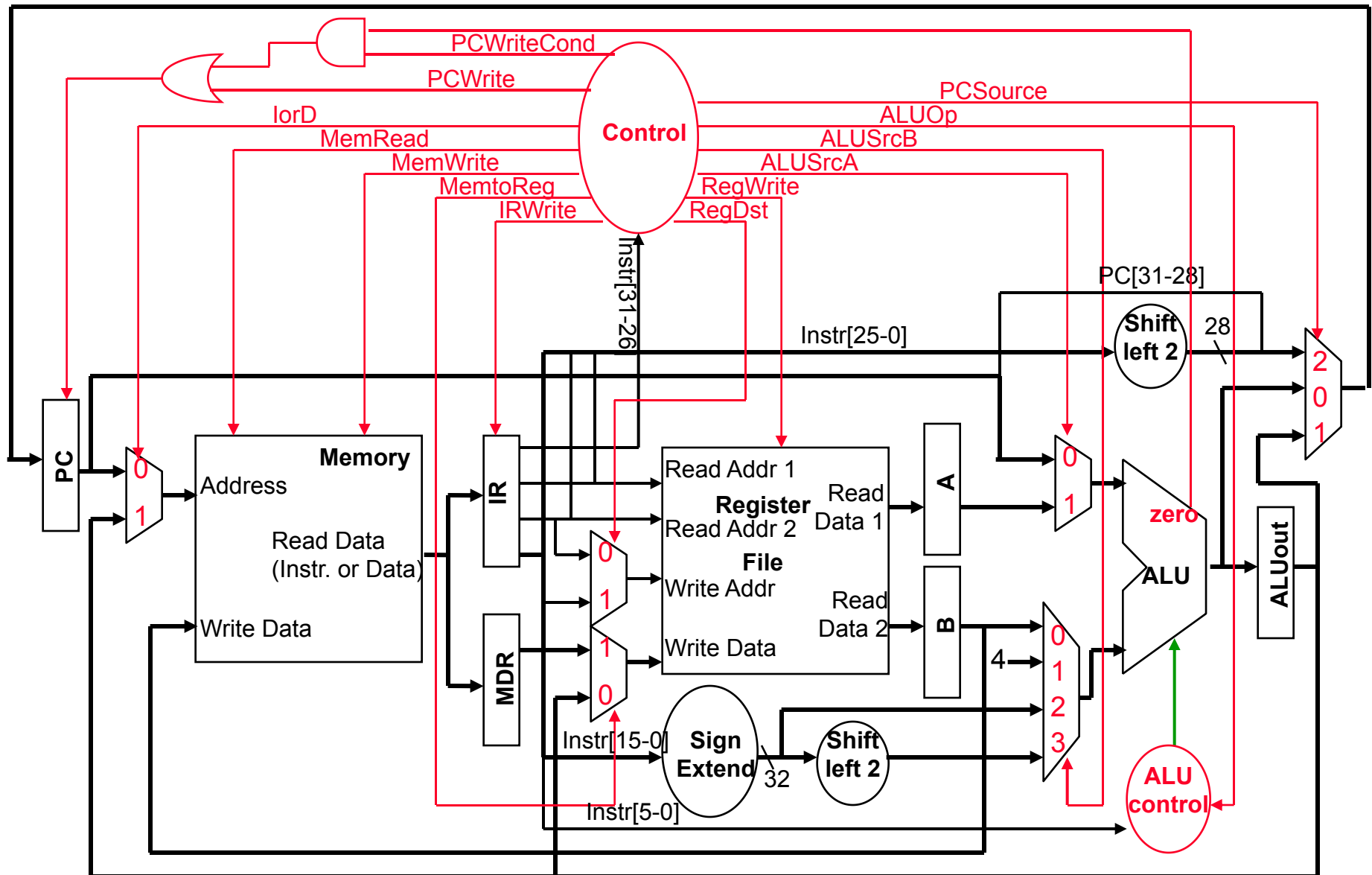
Clocking the Multicycle Datapath



Multicycle Approach

- ❑ Break up the instructions into steps where each step takes a clock cycle while trying to
 - balance the amount of work to be done in each step
 - use only one major functional unit per clock cycle
- ❑ At the end of a clock cycle
 - Store values needed in a later clock cycle by the **current** instruction in a state element (internal register: **not visible** to the programmer)
 - IR** – Instruction Register
 - MDR – Memory Data Register
 - A and B – Register File read data registers
 - ALUout – ALU output register
 - 📖 All (except **IR**) hold data only between a pair of adjacent clock cycles (so they don't need a write control signal)
 - Data used by subsequent instructions are stored in programmer **visible** state elements (i.e., Register File, PC, or Memory)

The Complete Multicycle Data with Control



Review: ALU Control

- Controlling the ALU uses of multiple decoding levels
 - main control unit generates the **ALUOp** bits
 - ALU control unit generates **ALUcontrol** bits

Instr op	funct	ALUOp	action	ALUcontrol
lw	xxxxxx	00	add	0110
sw	xxxxxx	00	add	0110
beq	xxxxxx	01	subtract	1110
add	100000	10	add	0110
sub	100010	10	subtract	1110
and	100100	10	and	0000
or	100101	10	or	0001
xor	100110	10	xor	0010
nor	100111	10	nor	0011
slt	101010	10	slt	1111

Multicycle Approach, con't

- ❑ Reading from or writing to any of the internal registers, Register File, or the PC occurs at the beginning (for read) or the end of a clock cycle (for write)
- ❑ Reading from the Register File takes ~50% of a clock cycle since it has additional control and access overhead (but reading can be done in parallel with decode)
- ❑ Had to add multiplexors in front of several of the functional unit input ports (e.g., Memory, ALU) because they are now shared by different clock cycles and/or do multiple jobs
- ❑ All operations occurring in one clock cycle occur in parallel
 - This limits us to one ALU operation, one Memory access, and one Register File access per clock cycle

Five Instruction Steps

1. Instruction Fetch
2. Instruction Decode and Register Fetch
3. R-type Instruction Execution, Memory Read/Write Address Computation, Branch Completion, or Jump Completion
4. Memory Read Access, Memory Write Completion or R-type Instruction Completion
5. Memory Read Completion (**Write Back**)

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

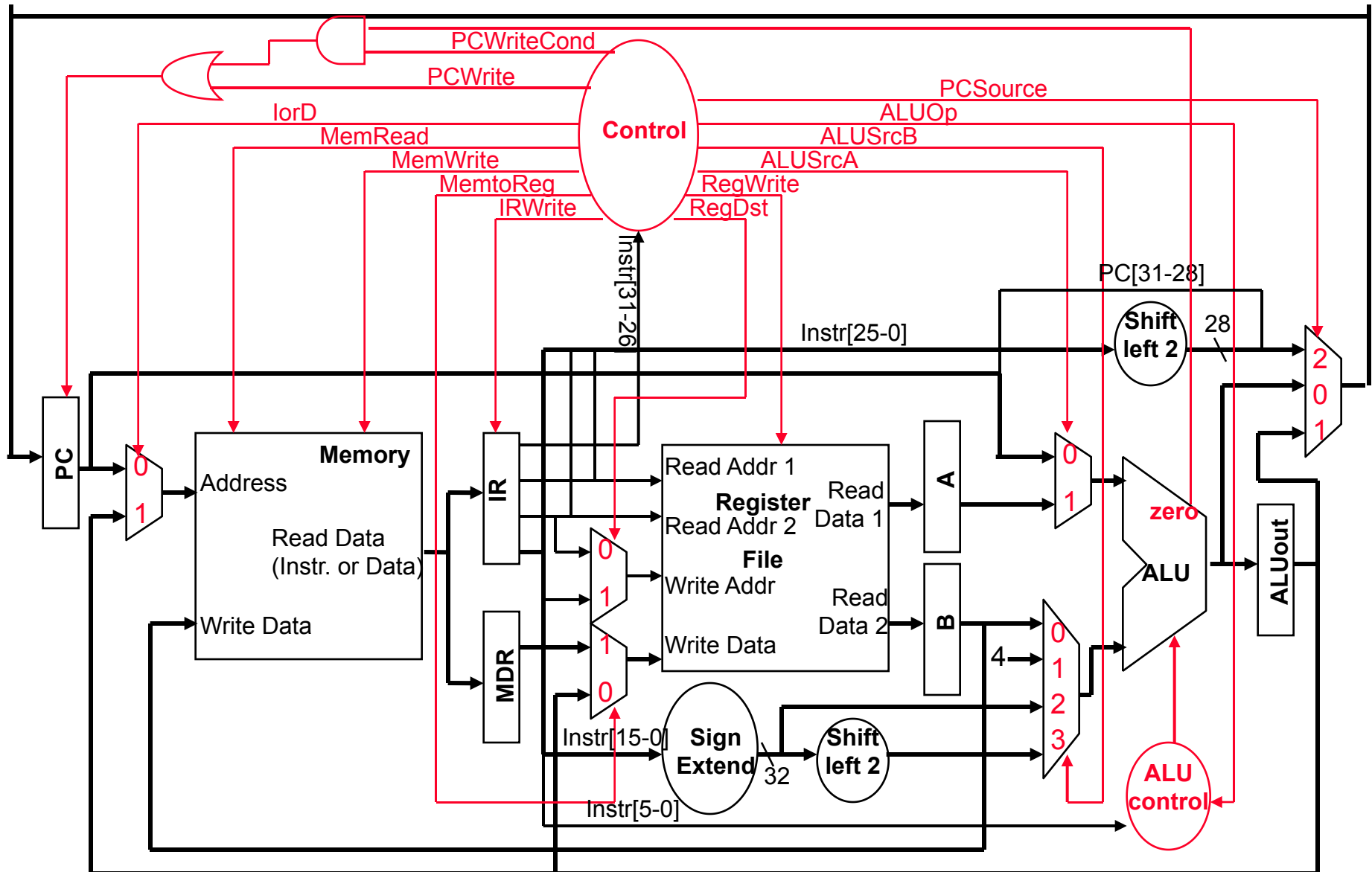
- ❑ Use PC to get instruction from the memory and put it in the Instruction Register
- ❑ Increment the PC by 4 and put the result back in PC
- ❑ Can be described succinctly using the RTL "Register-Transfer Language"

```
IR = Memory[PC];  
PC = PC + 4;
```

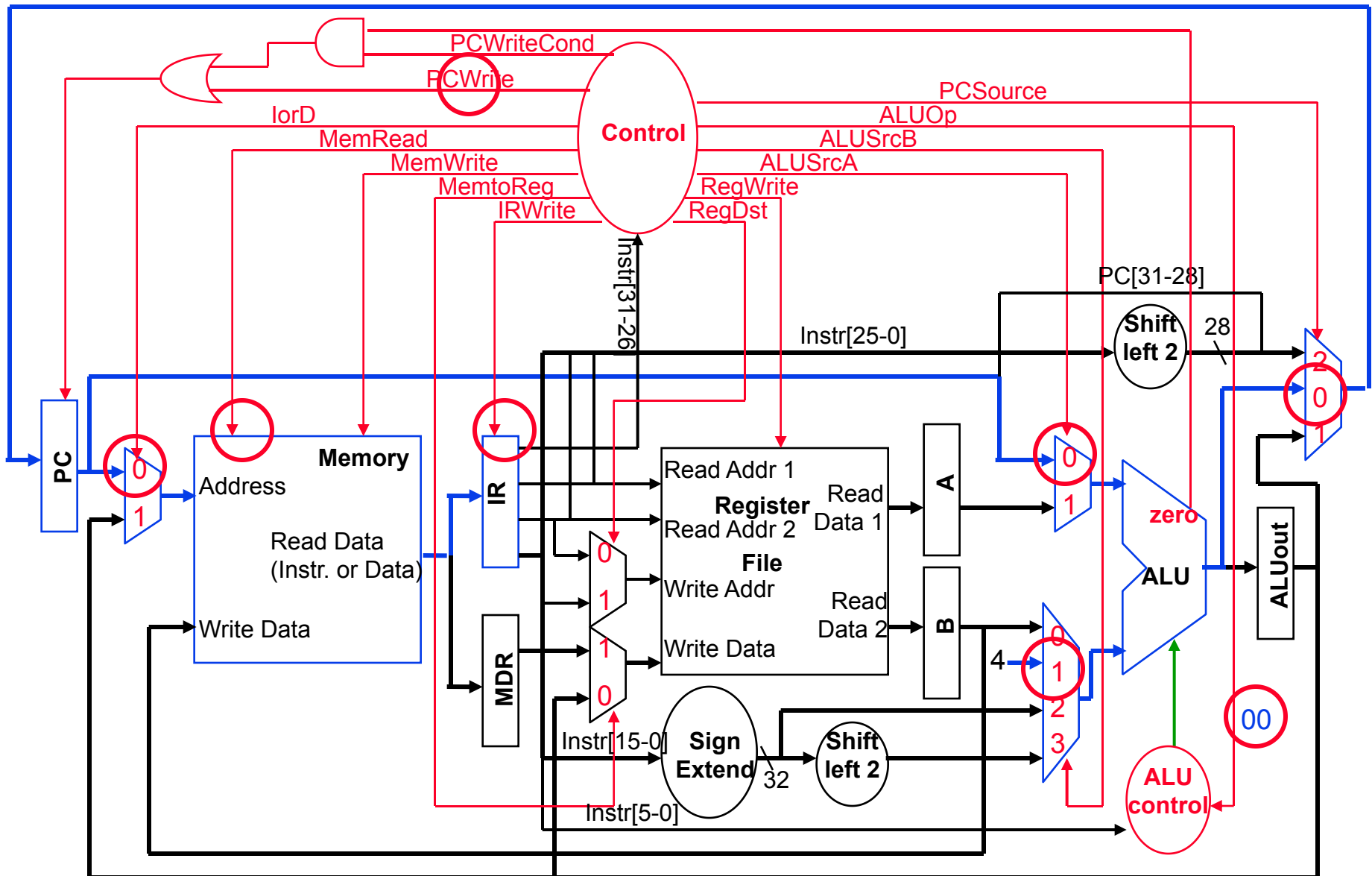
Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Datapath Activity During Instruction Fetch



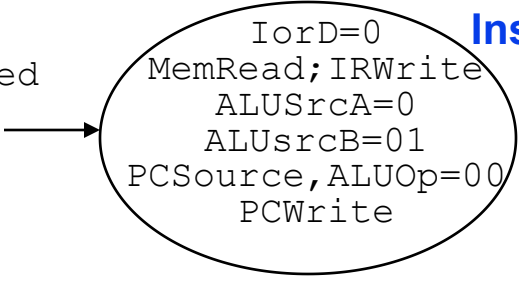
Datapath Activity During Instruction Fetch



Fetch Control Signals Settings

Unless otherwise assigned

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



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

Instr Fetch

Step 2: Instruction Decode and Register Fetch

- ❑ Don't know what the instruction is yet, so can only
 - Read registers rs and rt **in case** we need them
 - Compute the branch address **in case** the instruction is a branch
- ❑ The RTL:

`A = Reg[IR[25-21]];`

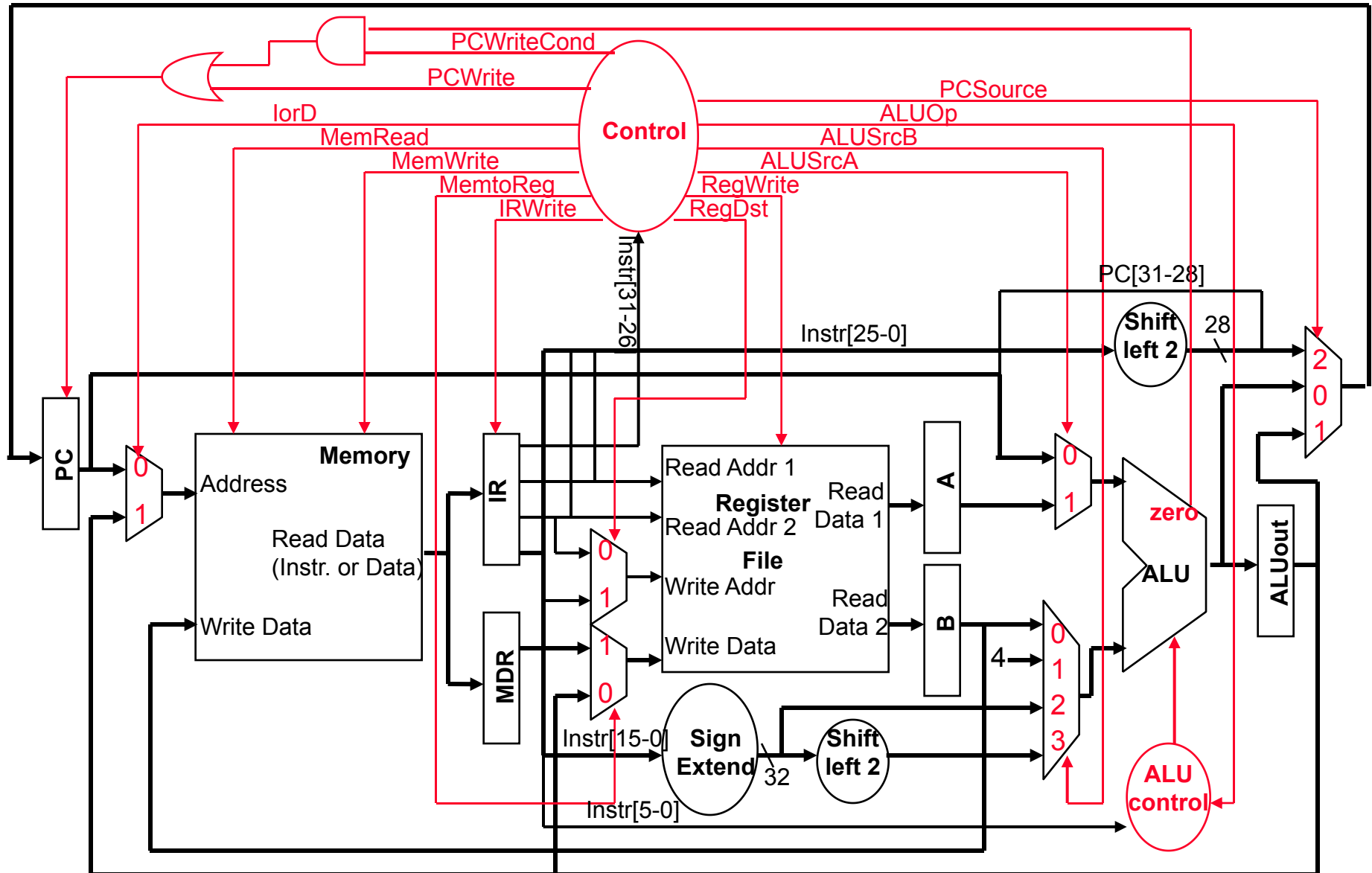
`B = Reg[IR[20-16]];`

`ALUOut = PC`

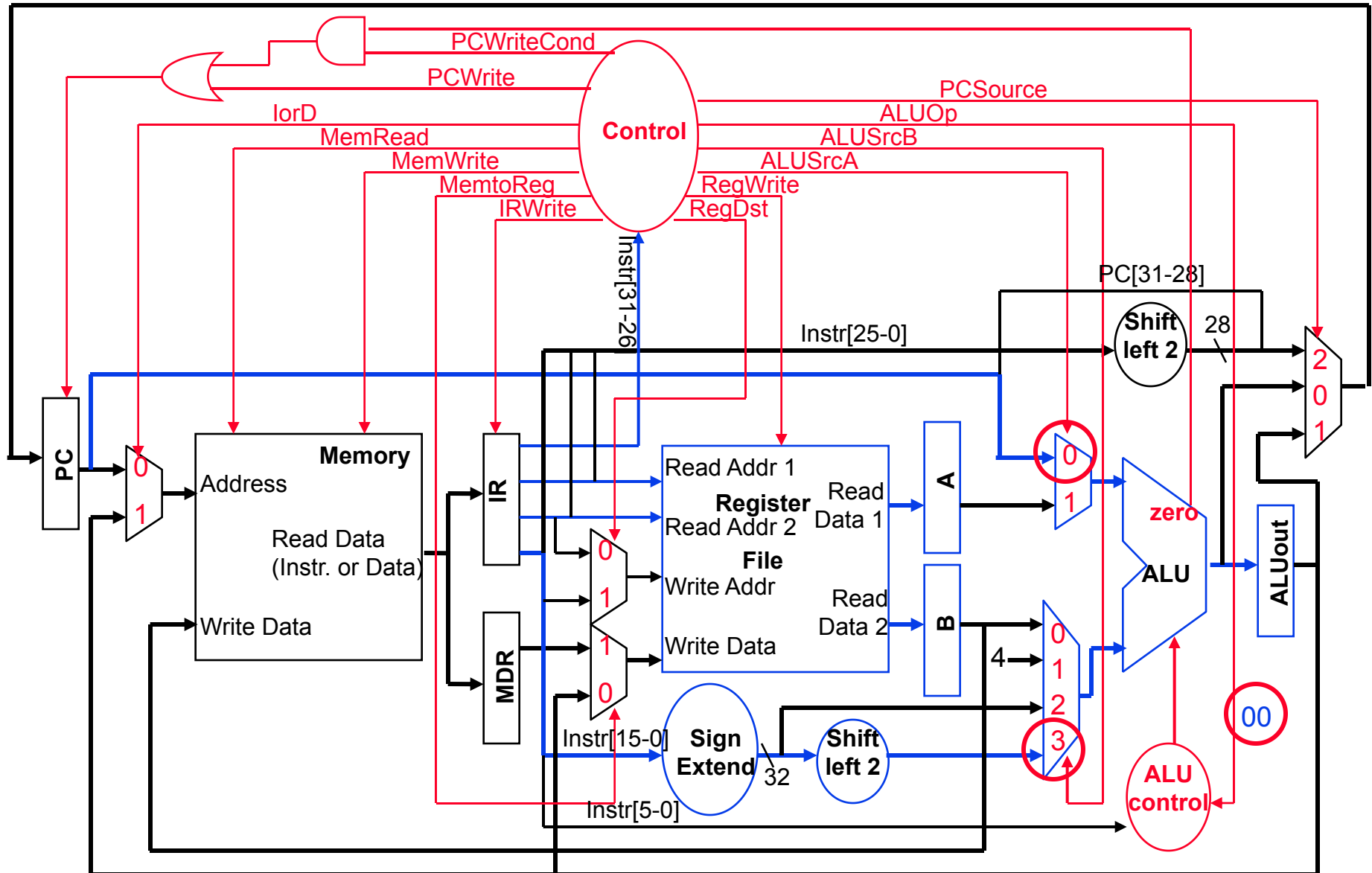
`+ (sign-extend(IR[15-0]) << 2);`

- ❑ Note we aren't setting any control lines based on the instruction (since we don't know what it is (the control logic is busy "**decoding**" the op code bits))

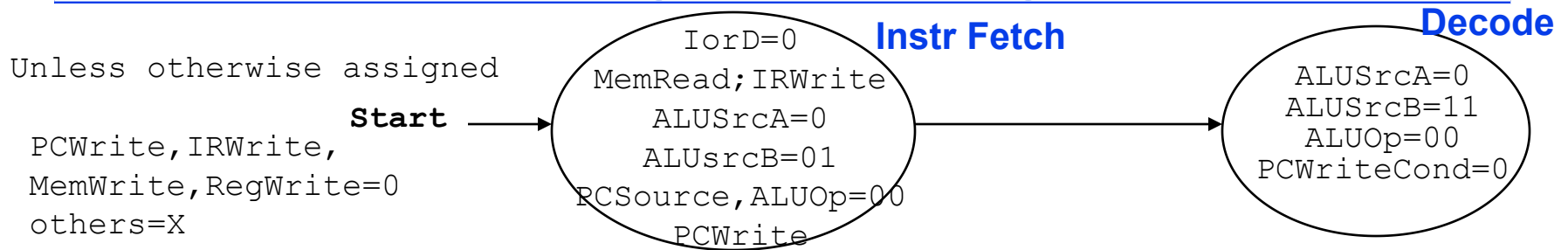
Datapath Activity During Instruction Decode



Datapath Activity During Instruction Decode



Decode Control Signals Settings



Step 3 (instruction dependent)

- ❑ ALU is performing one of four functions, based on instruction type

- ❑ Memory reference (`lw` and `sw`):

`ALUOut = A + sign-extend(IR[15-0]);`

- ❑ R-type:

`ALUOut = A op B;`

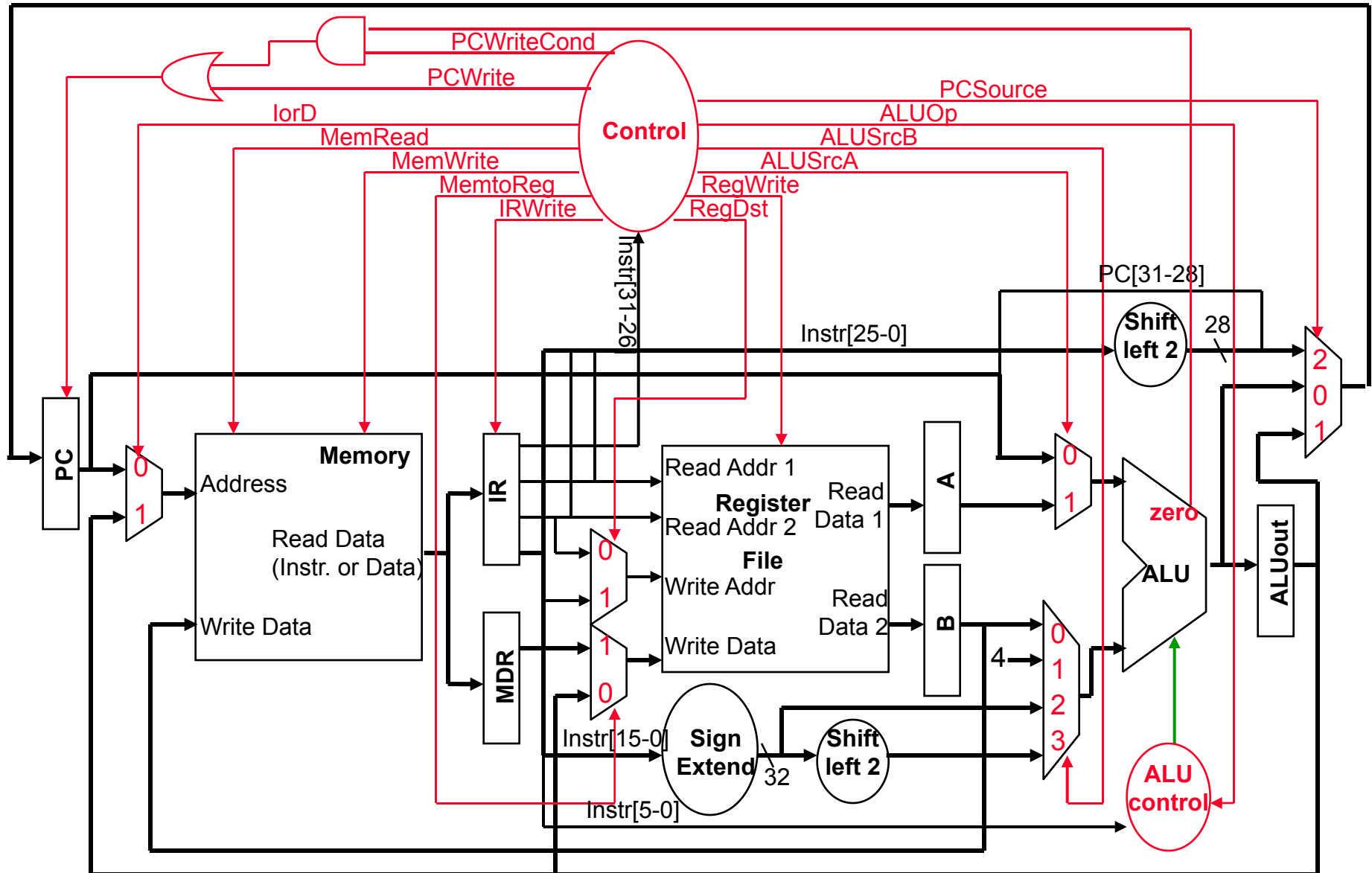
- ❑ Branch:

`if (A==B) PC = ALUOut;`

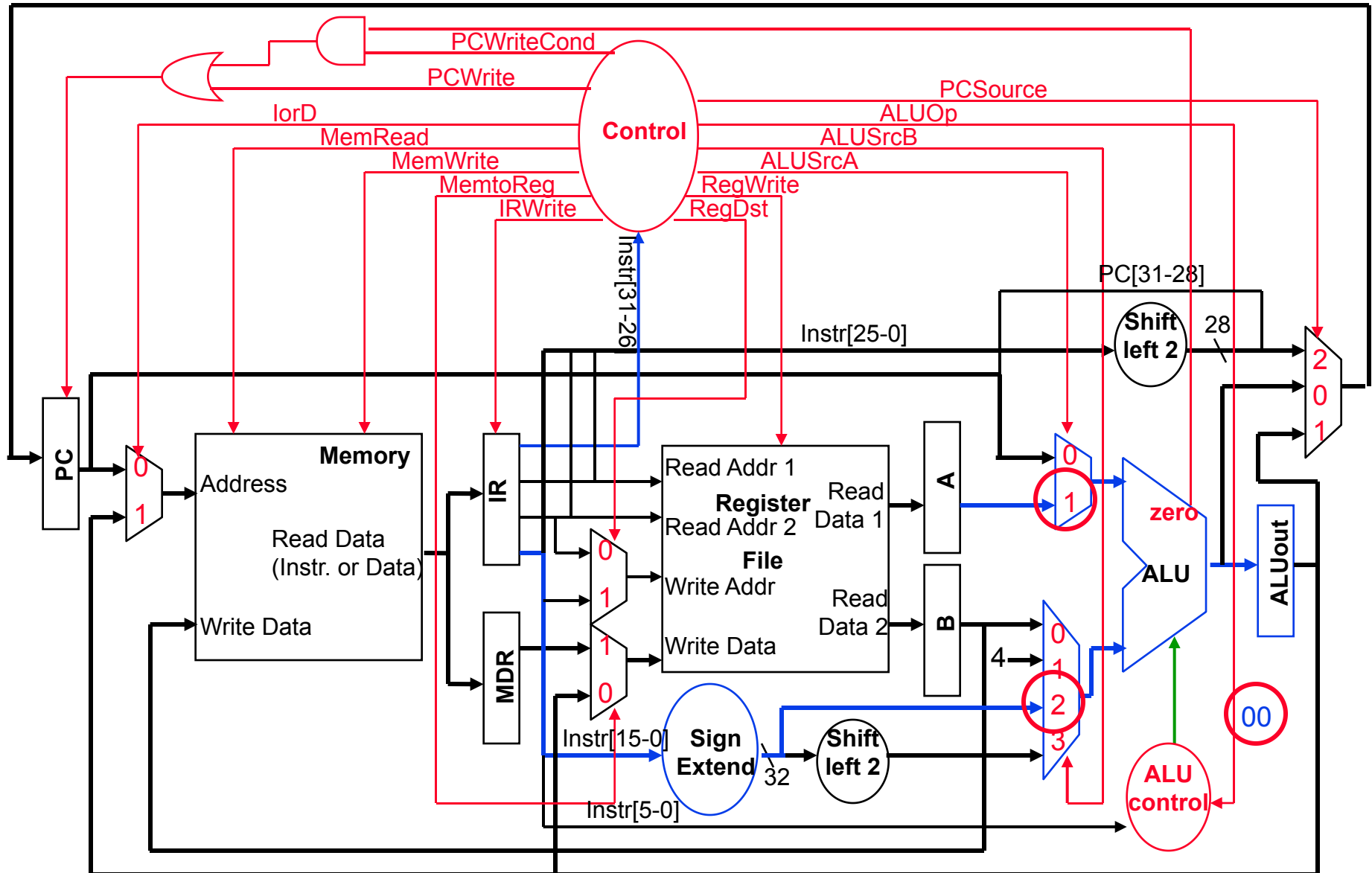
- ❑ Jump:

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

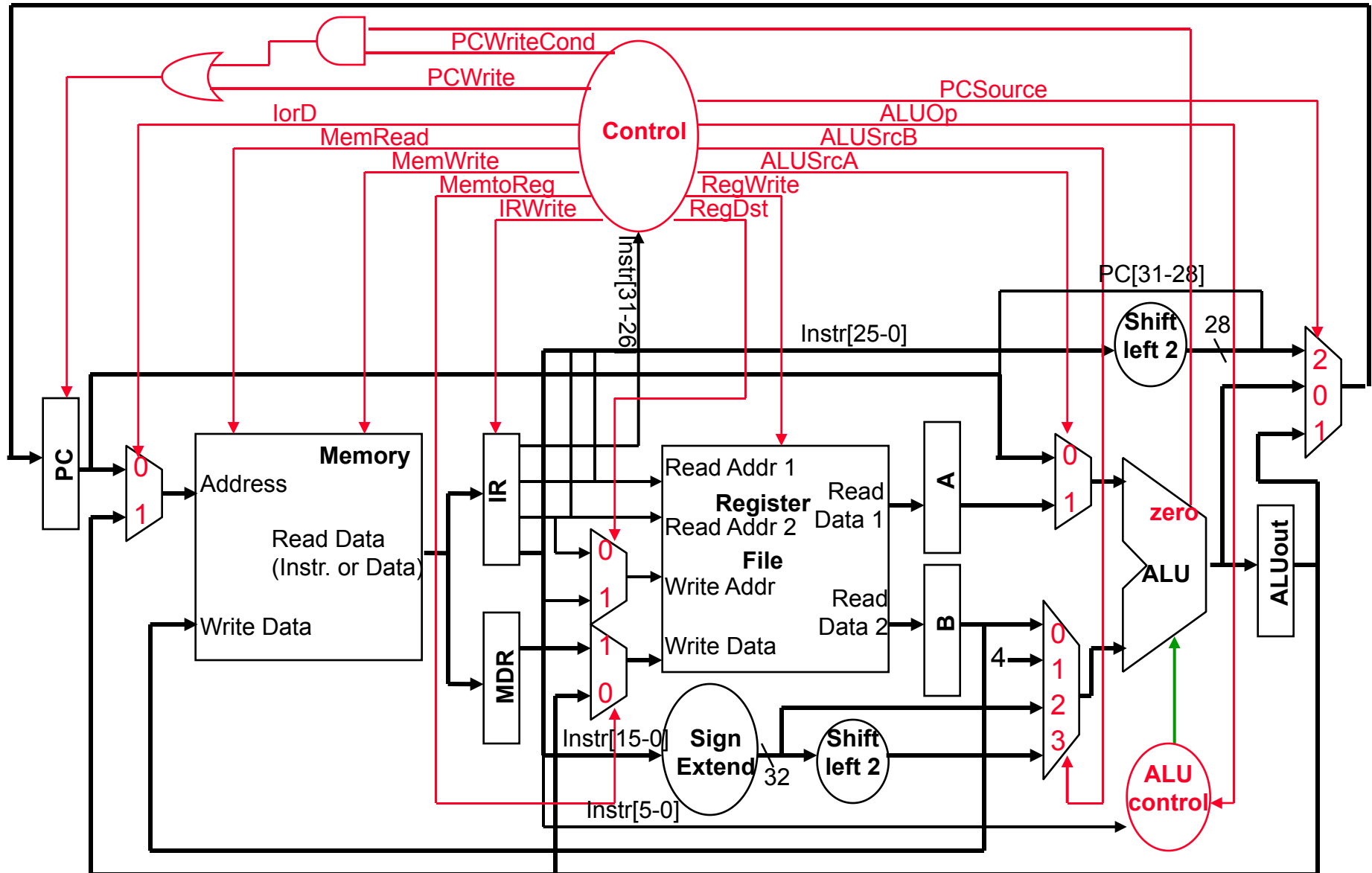
Datapath Activity During 1w & sw Execute



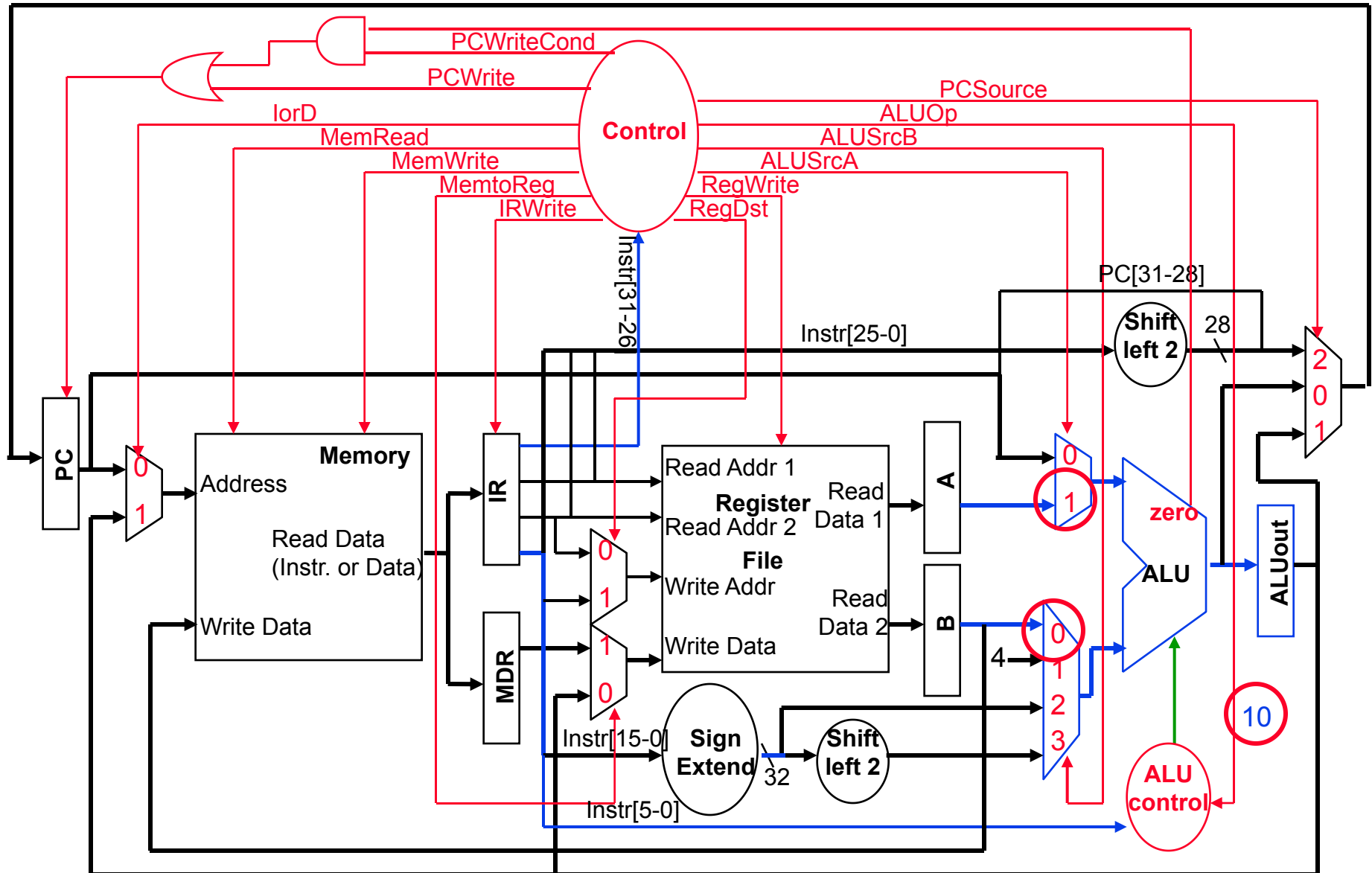
Datapath Activity During 1w & sw Execute



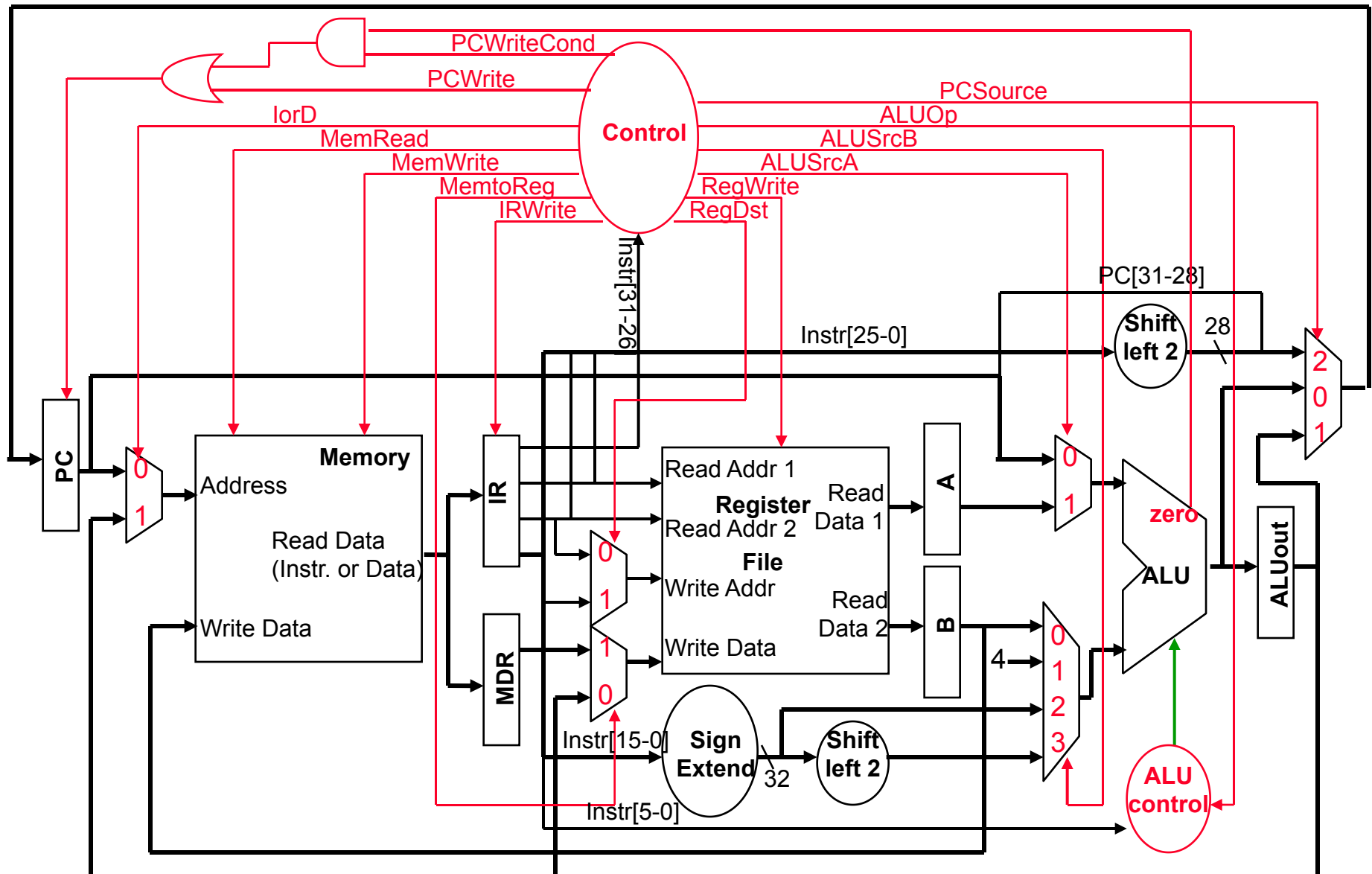
Datapath Activity During R-type Execute



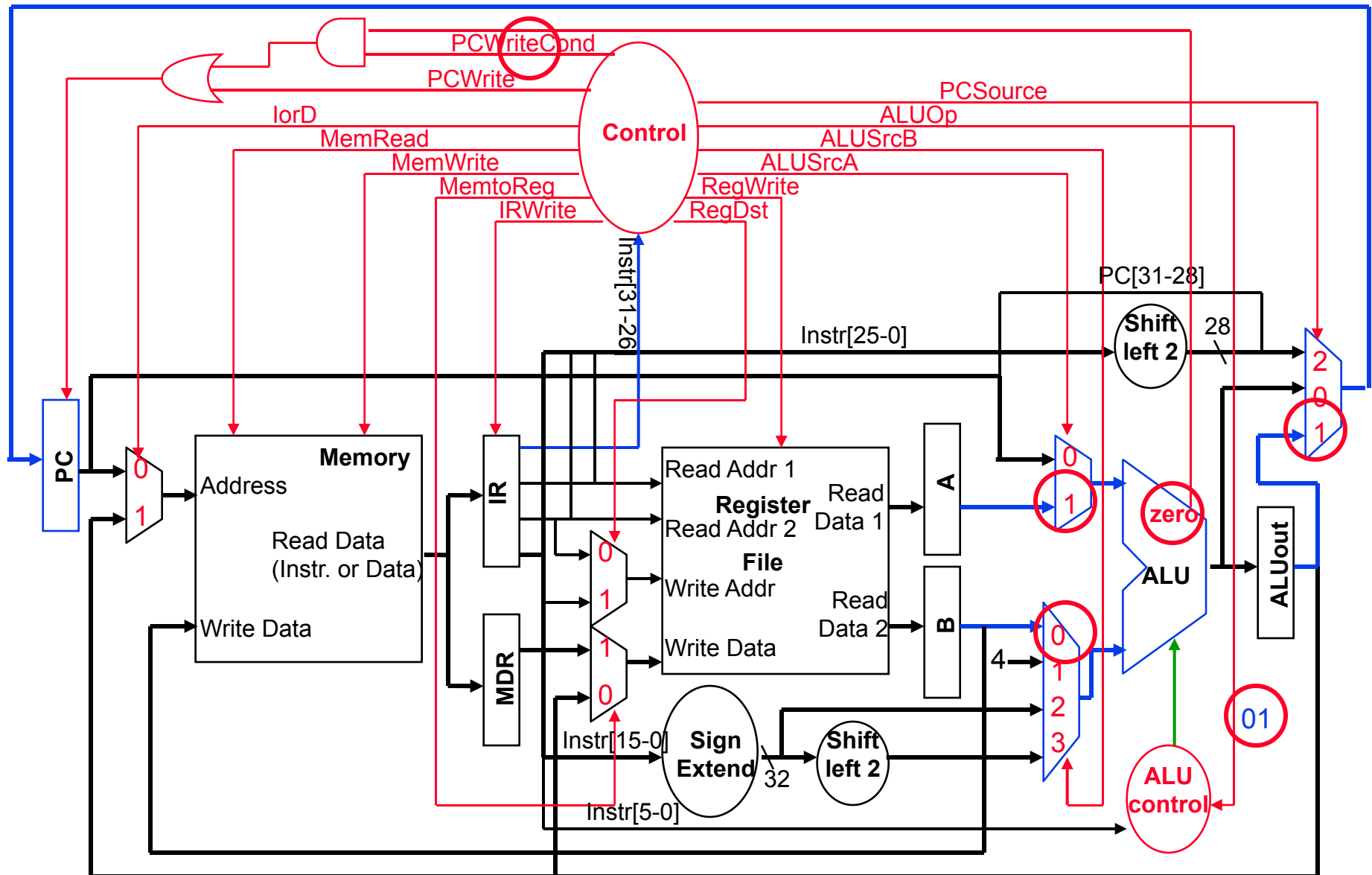
Datapath Activity During R-type Execute



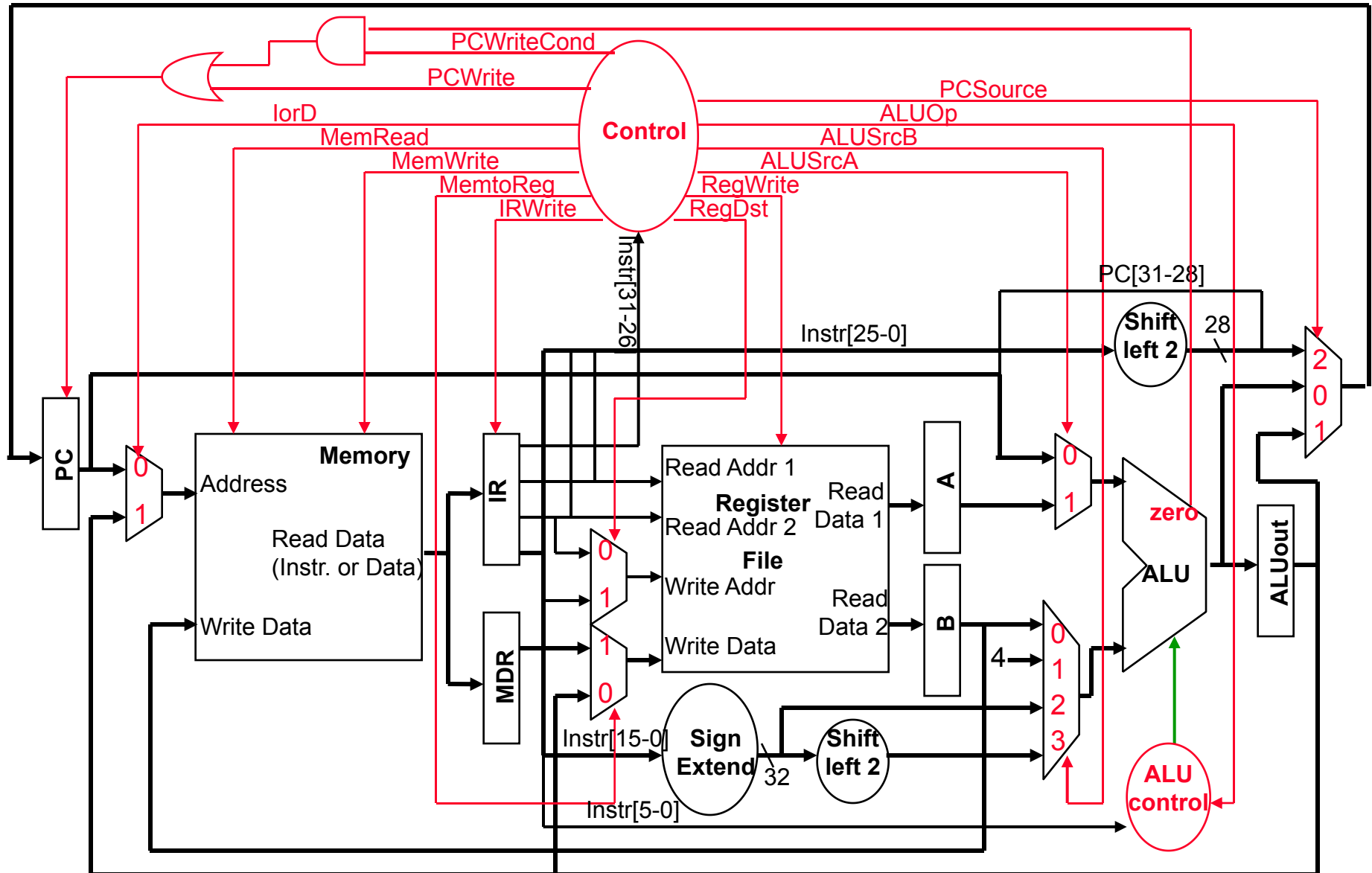
Datapath Activity During beq Execute



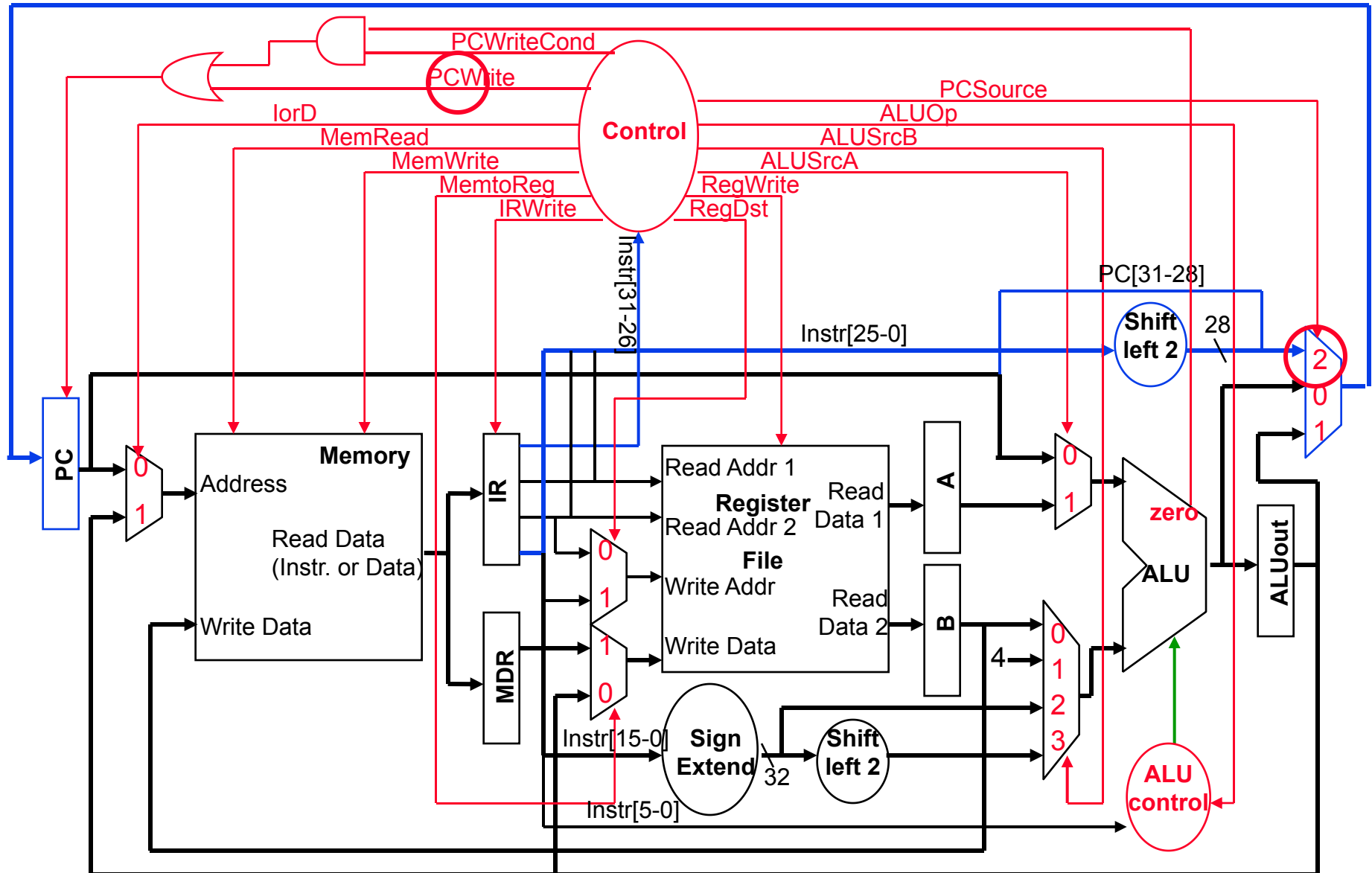
Datapath Activity During beq Execute



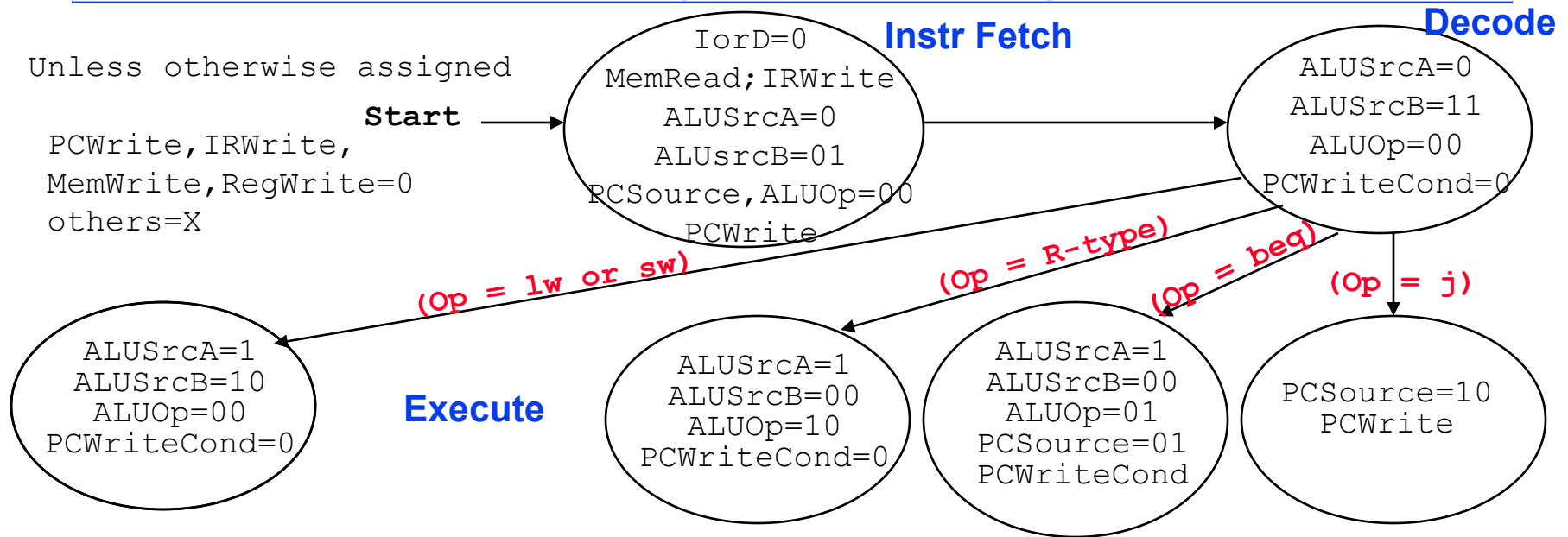
Datapath Activity During j Execute



Datapath Activity During j Execute



Execute Control Signals Settings



Step 4 (also instruction dependent)

❑ Memory reference:

`MDR = Memory[ALUOut];` `-- lw`

or

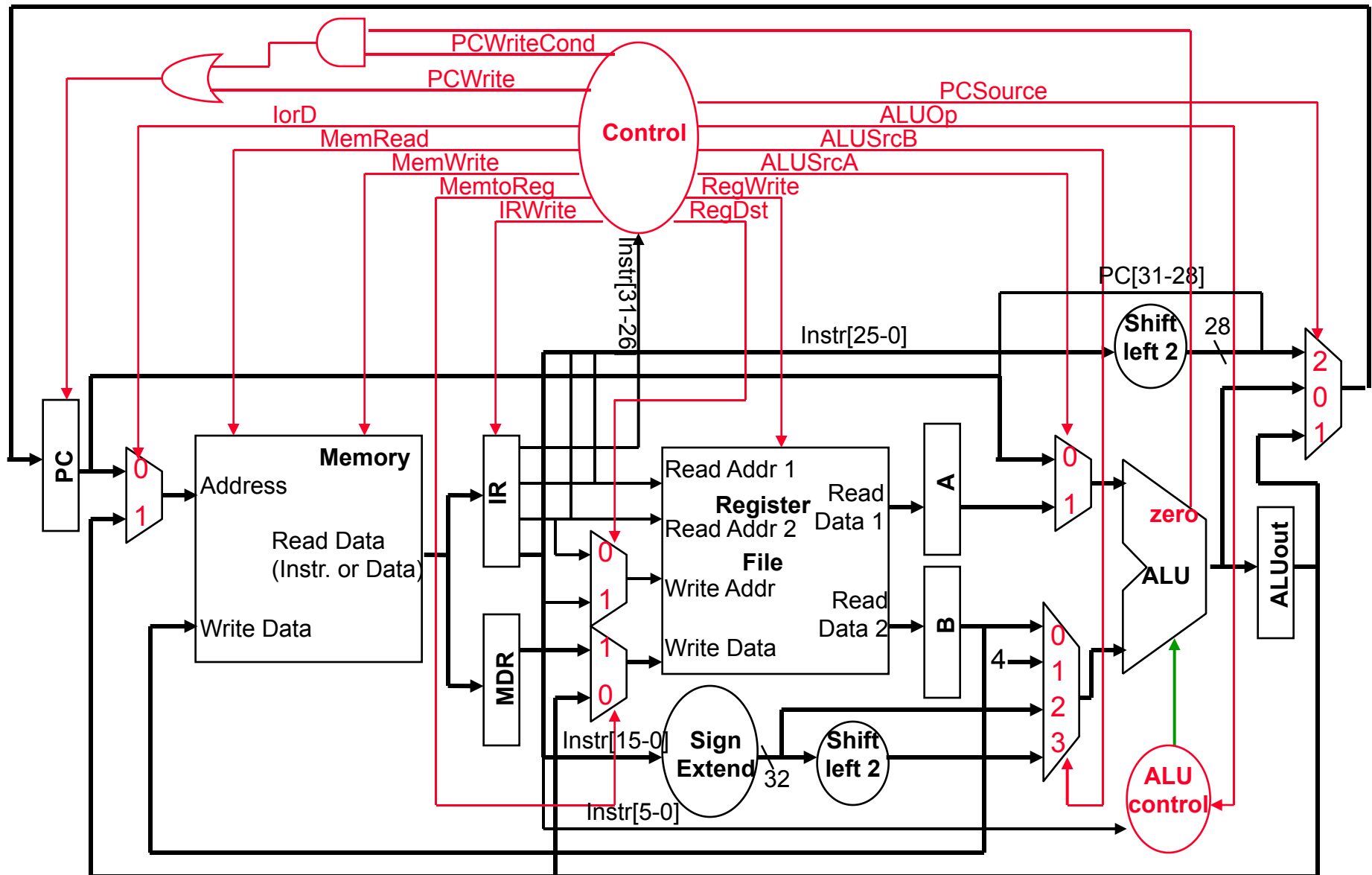
`Memory[ALUOut] = B;` `-- sw`

❑ R-type instruction completion

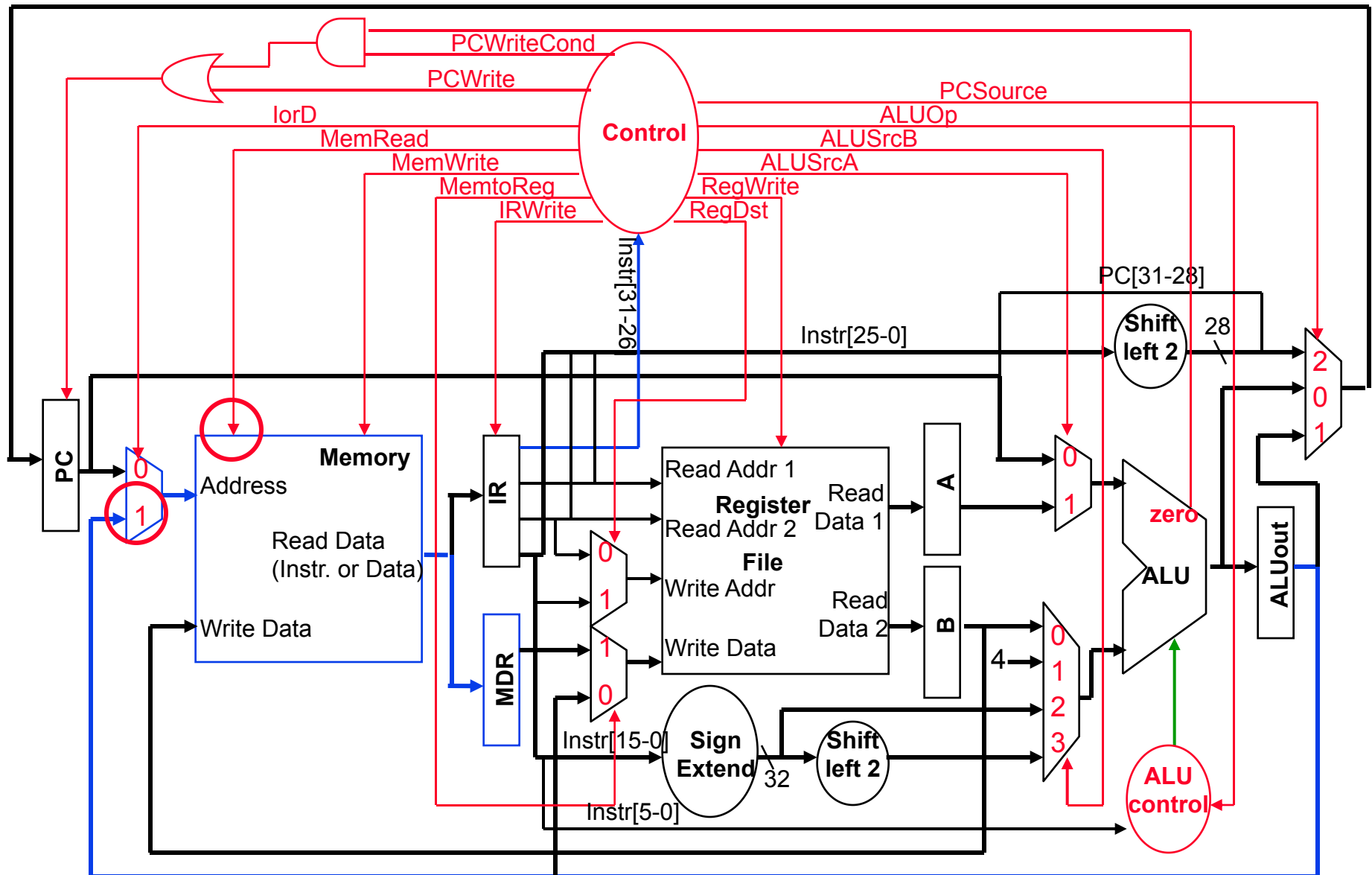
`Reg[IR[15-11]] = ALUOut;`

❑ Remember, the register write actually takes place at the **end** of the cycle on the clock edge

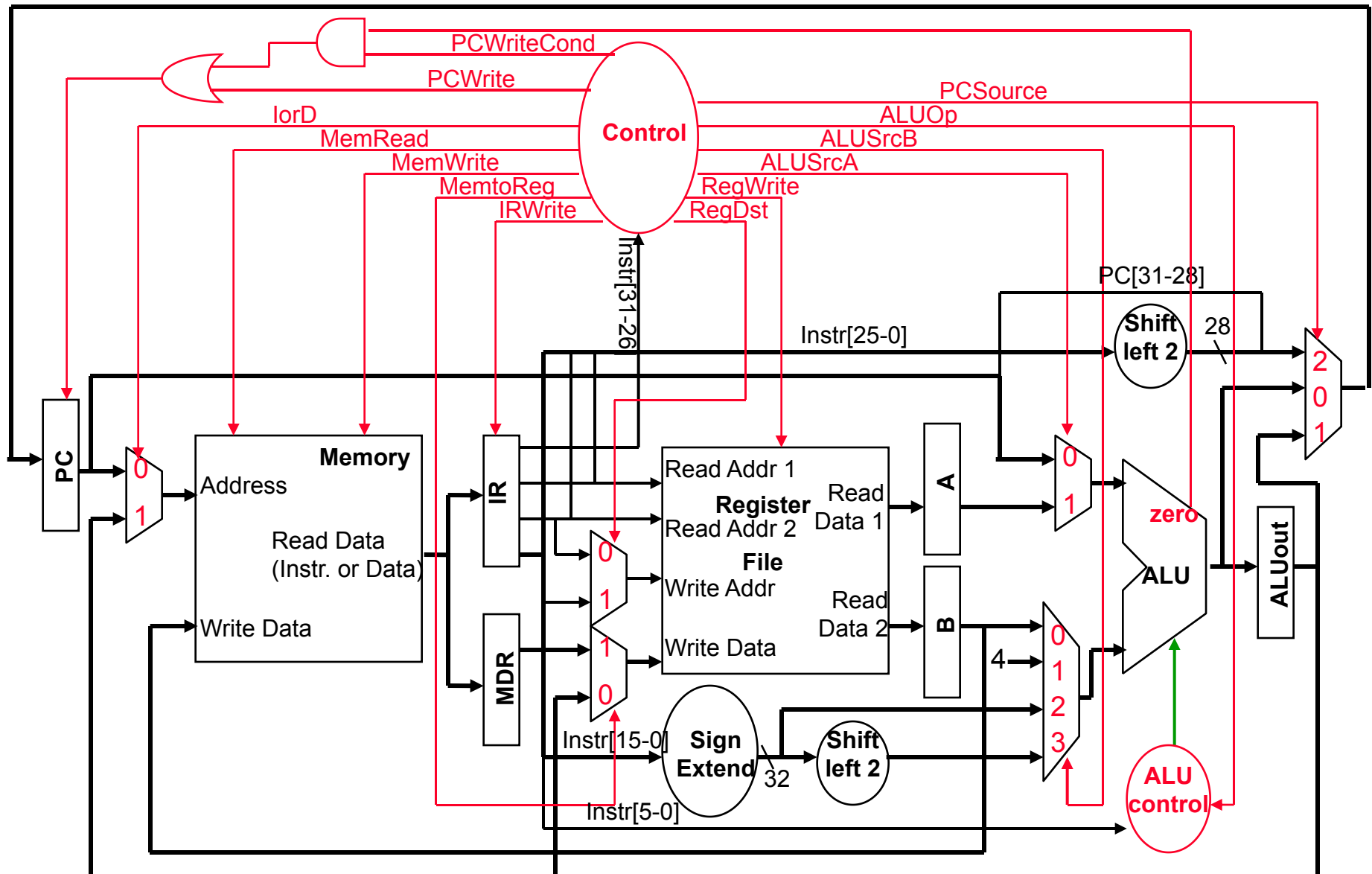
Datapath Activity During 1w Memory Access



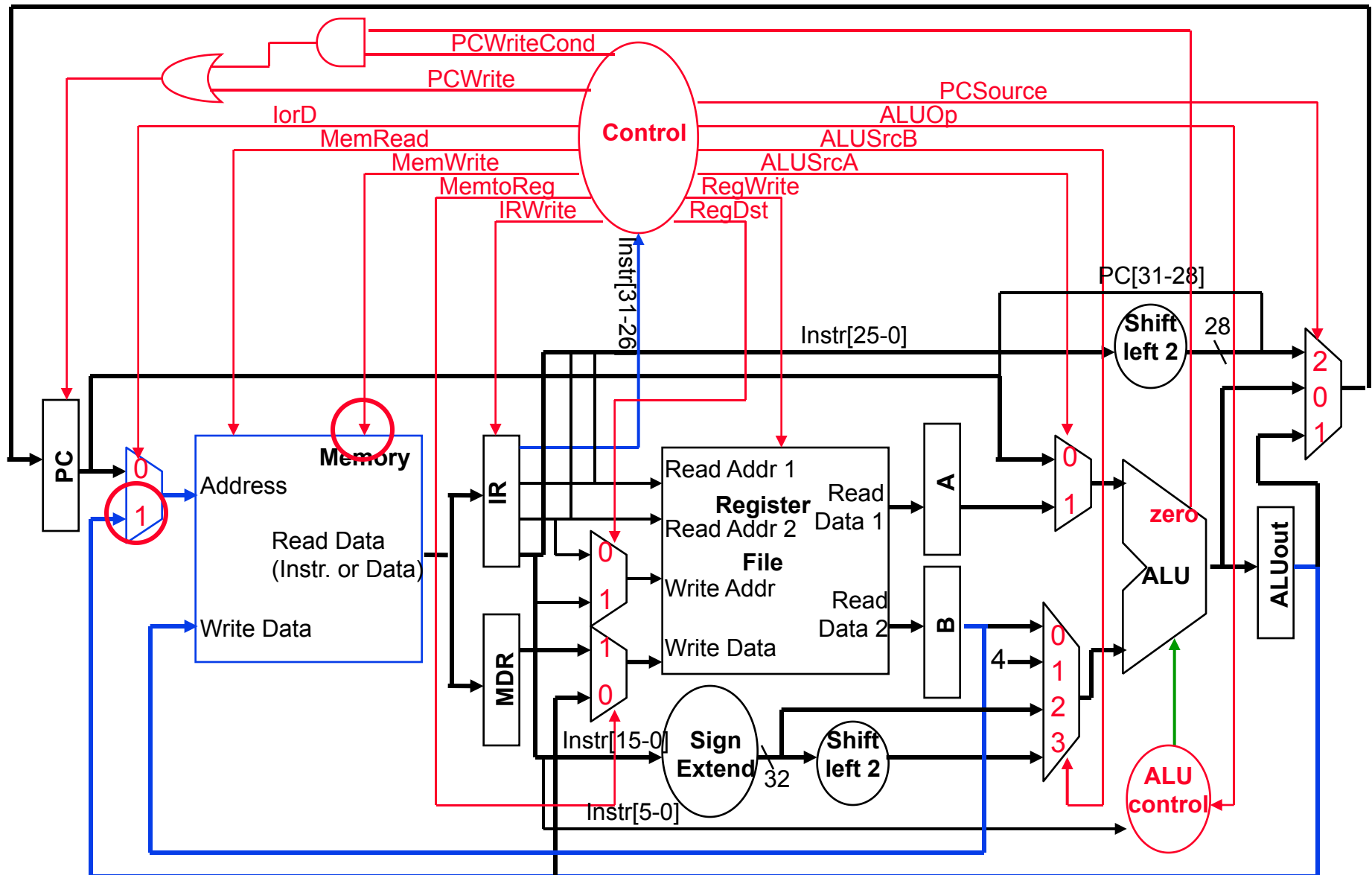
Datapath Activity During 1w Memory Access



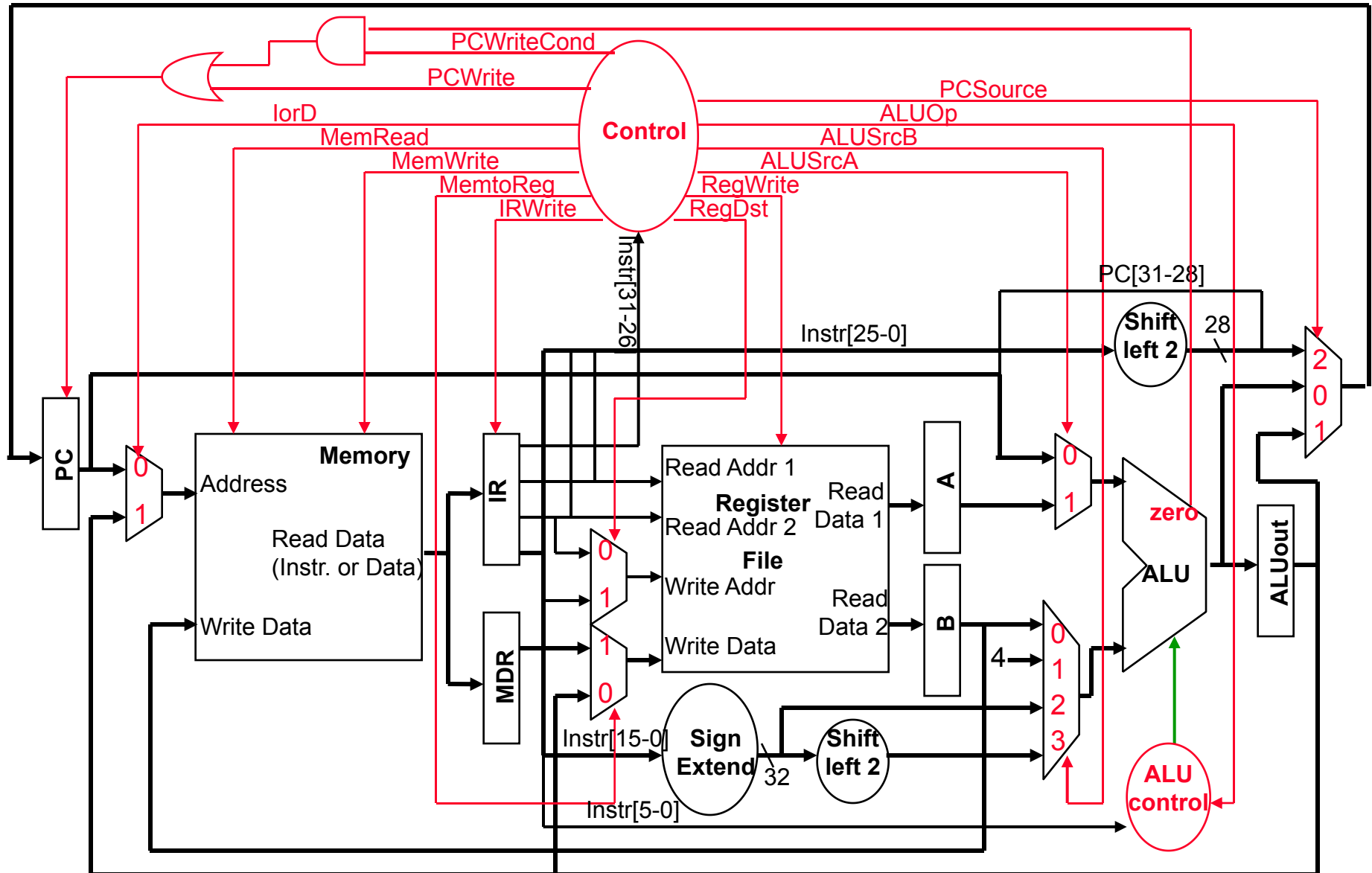
Datapath Activity During sw Memory Access



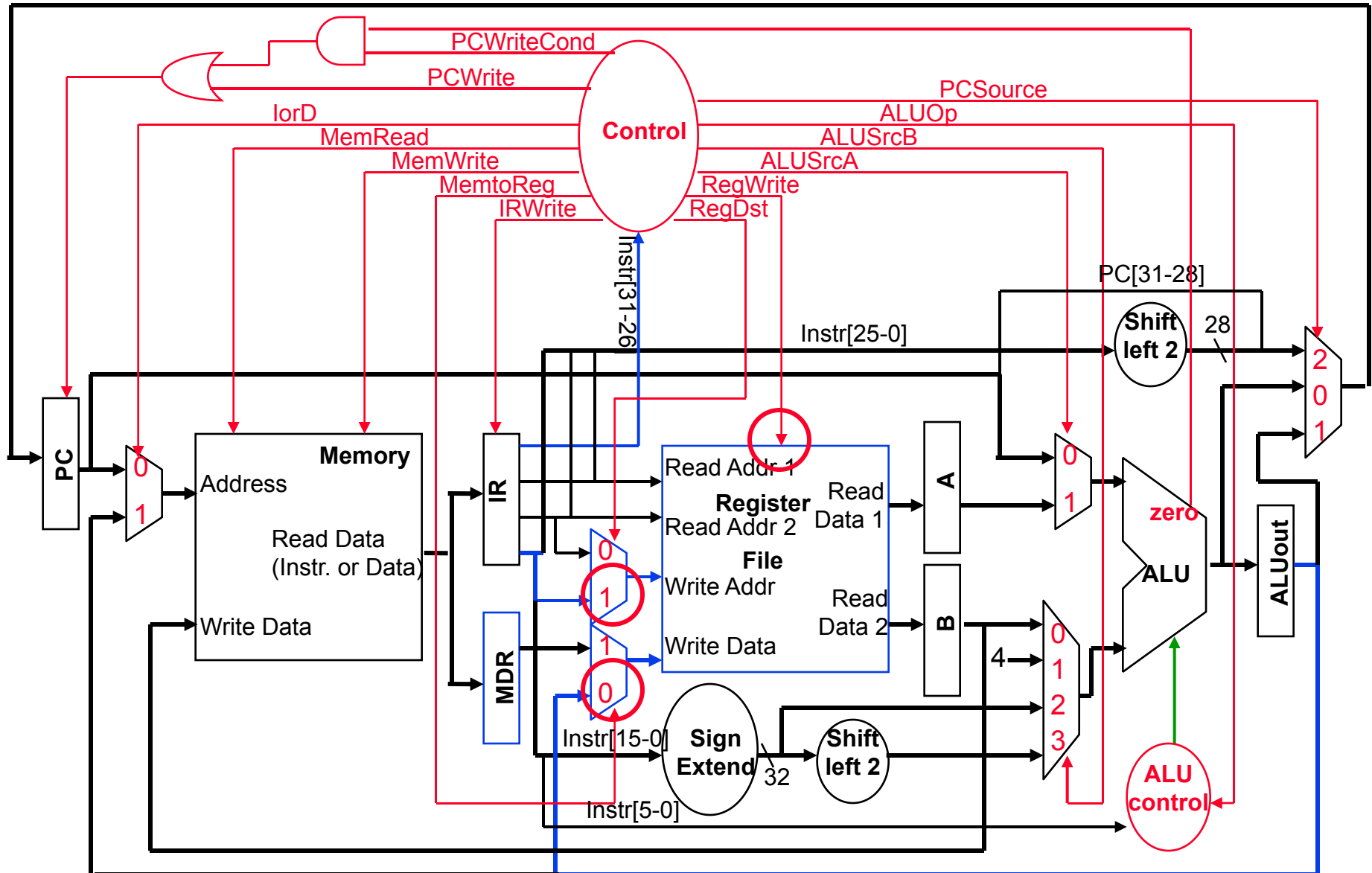
Datapath Activity During sw Memory Access



Datapath Activity During R-type Completion



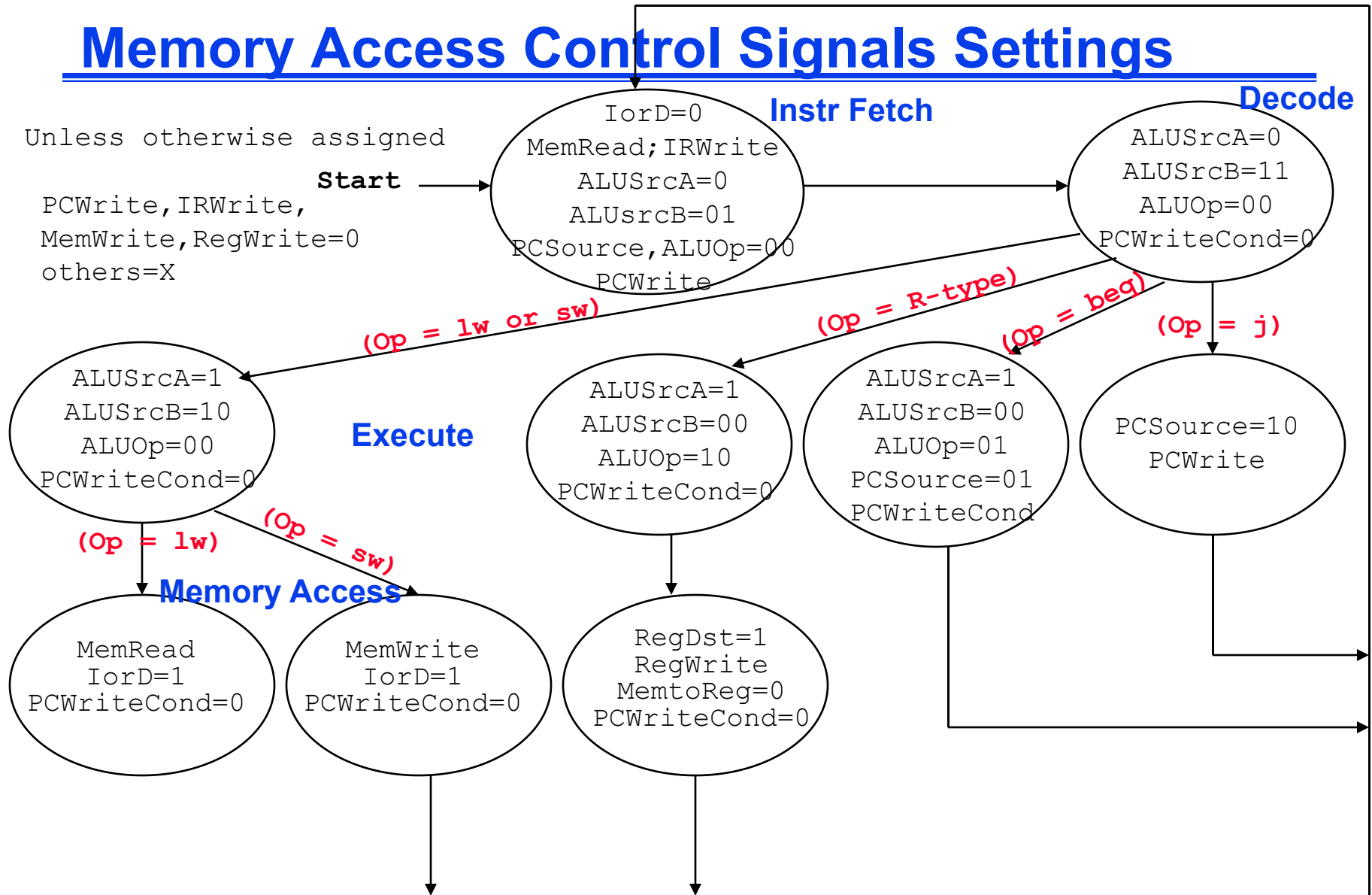
Datapath Activity During R-type Completion



Memory Access Control Signals Settings

Unless otherwise assigned

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



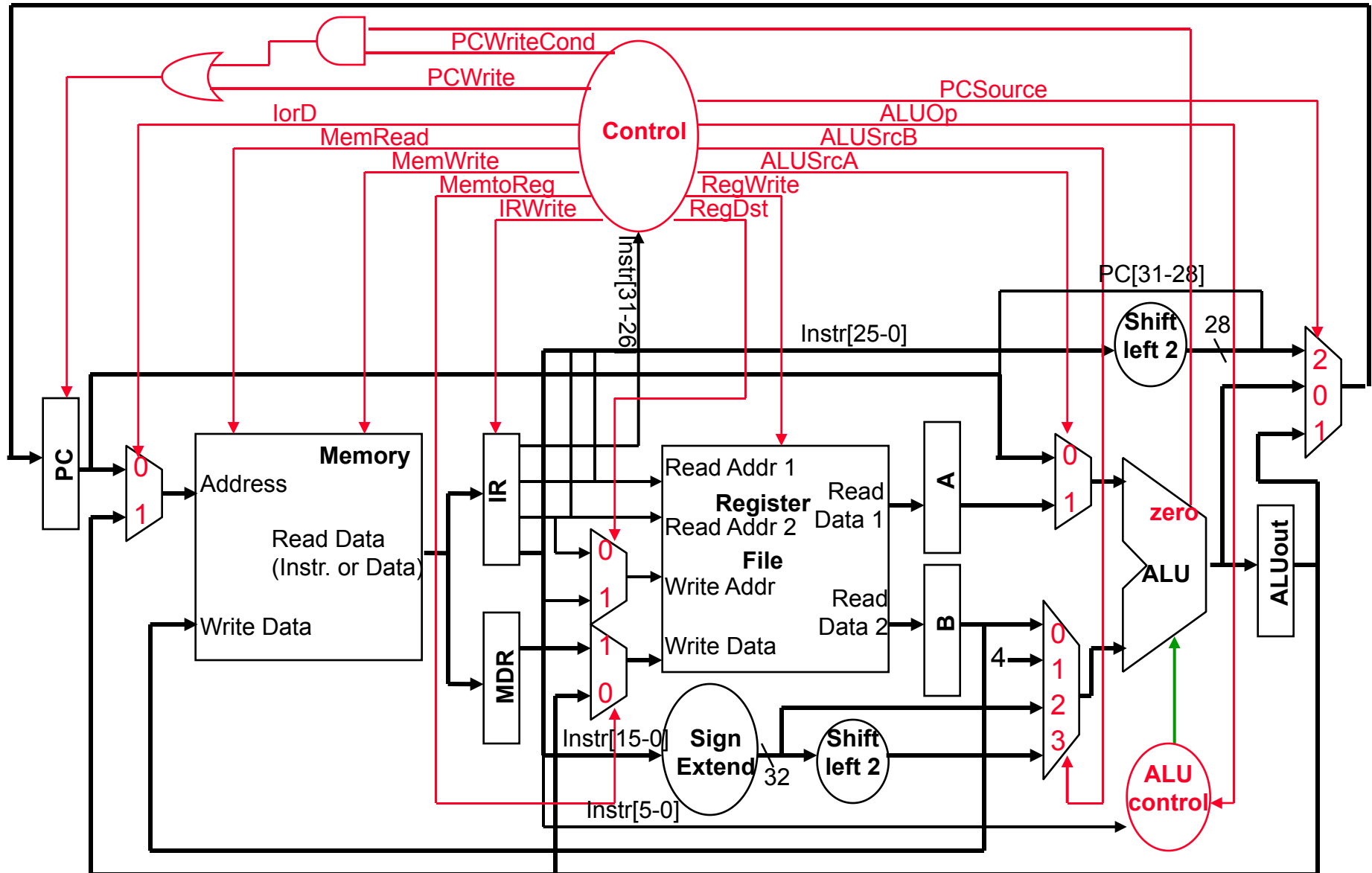
Step 5: Memory Read Completion (Write Back)

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

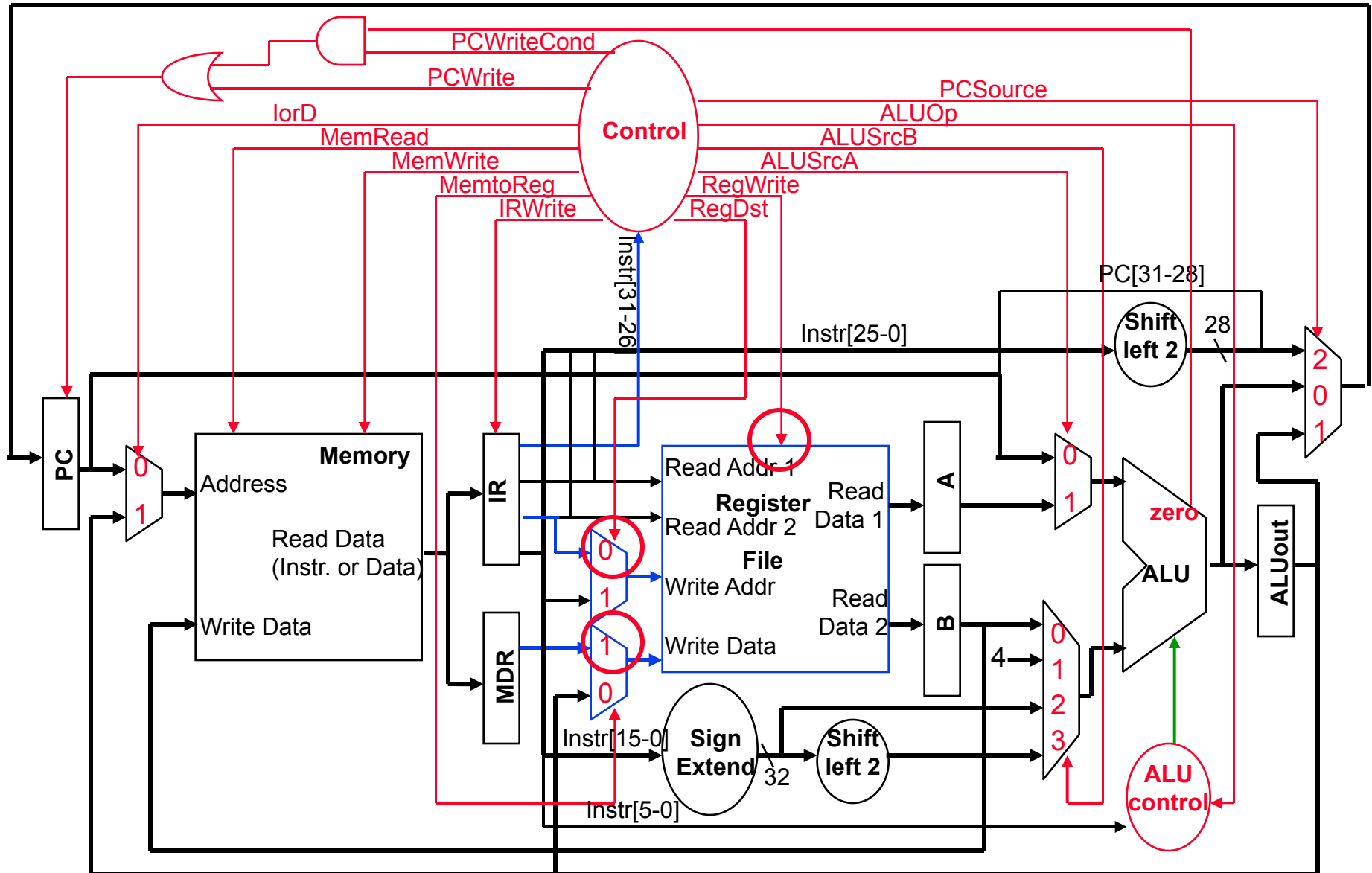
$\text{Reg}[\text{IR}[20-16]] = \text{MDR};$

What about all the other instructions?

Datapath Activity During 1w Write Back



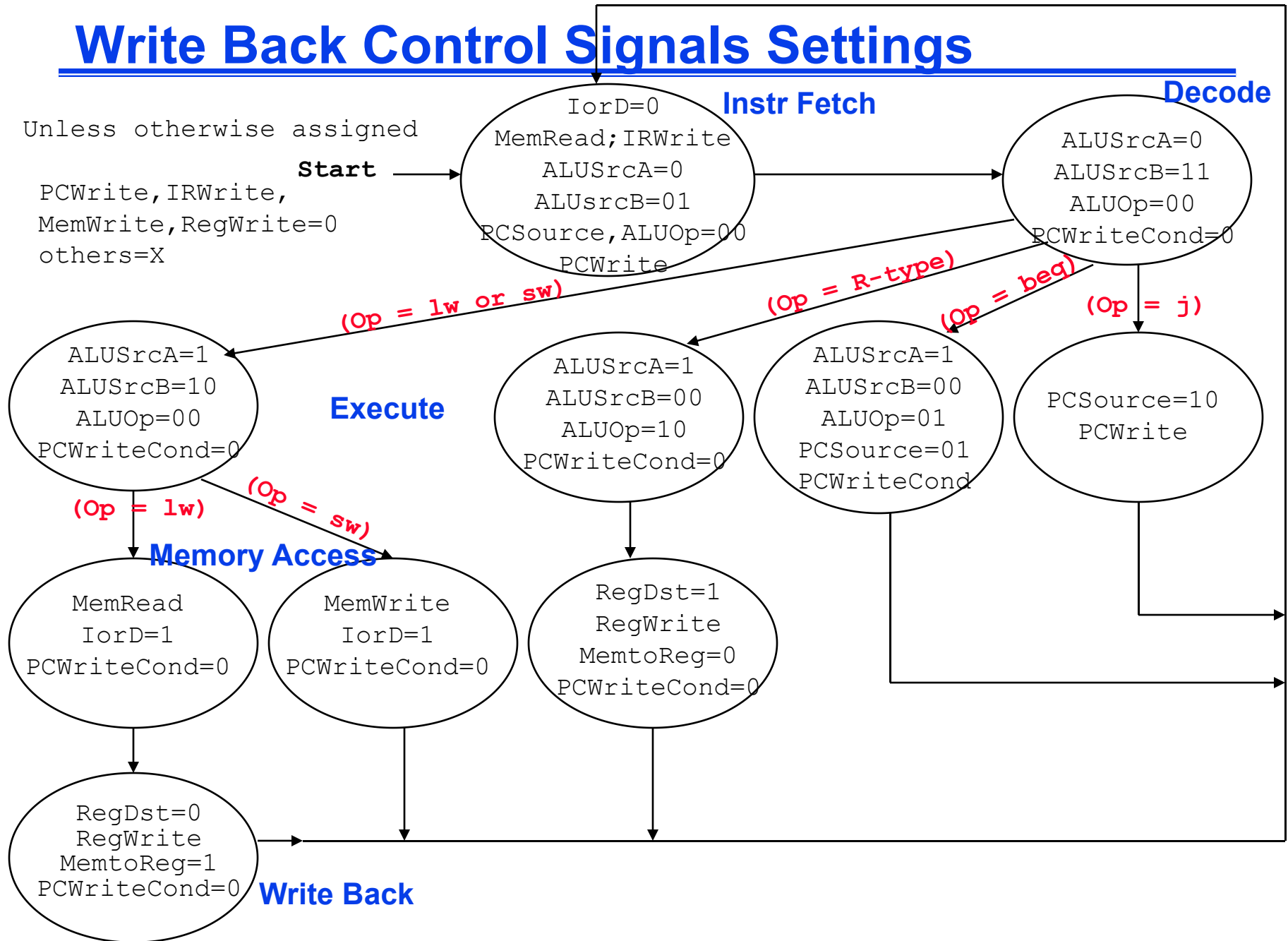
Datapath Activity During 1w Write Back



Write Back Control Signals Settings

Unless otherwise assigned

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



RTL Summary

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

Answering Simple Questions

- ❑ How many cycles will it take to execute this code?

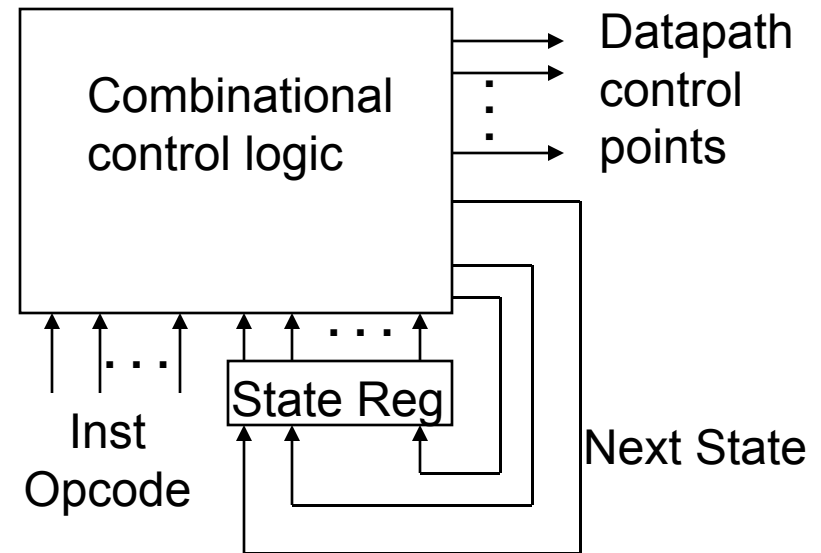
```
lw    $t2, 0($t3)      5
lw    $t3, 4($t3)      5
beq   $t2, $t3, Label  3
#assume not
add   $t5, $t2, $t3    4
sw    $t5, 8($t3)      4
Label: ...             =
                                21 cycles
address for second lw being calculated
```

- ❑ What is going on during the 8th cycle of execution?
- ❑ In what cycle does the actual addition of \$t2 and \$t3 takes place? 16th cycle 12th cycle
- ❑ In what cycle is the branch target address calculated?

Multicycle Control

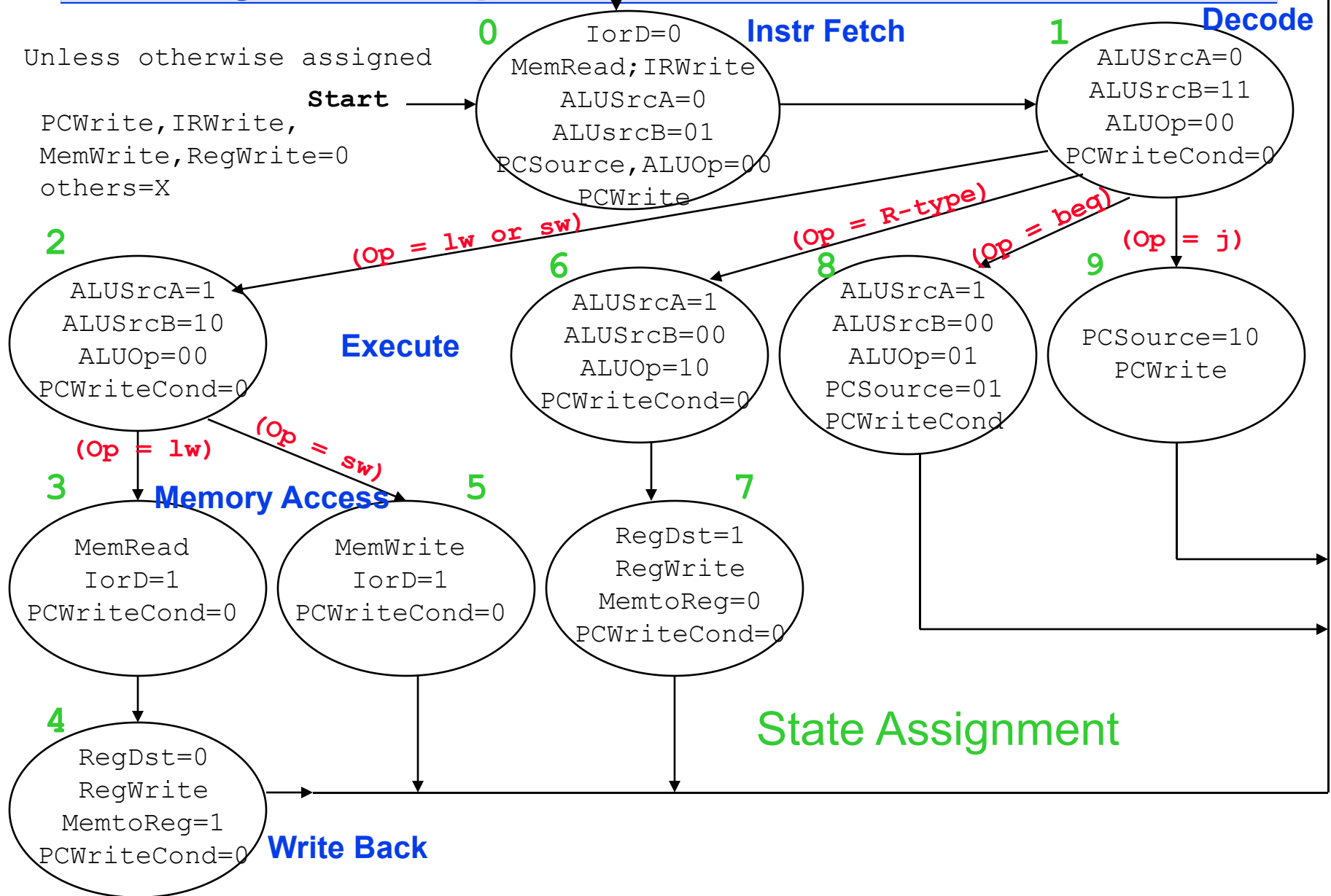
- ❑ Multicycle datapath control signals are not determined solely by the bits in the instruction
 - e.g., op code bits tell what operation the ALU should be doing, but *not* what instruction cycle is to be done next

- ❑ We can use a finite state machine for control
 - a set of states (current state stored in State Register)
 - next state function (determined by current state and the input)
 - output function (determined by current state)

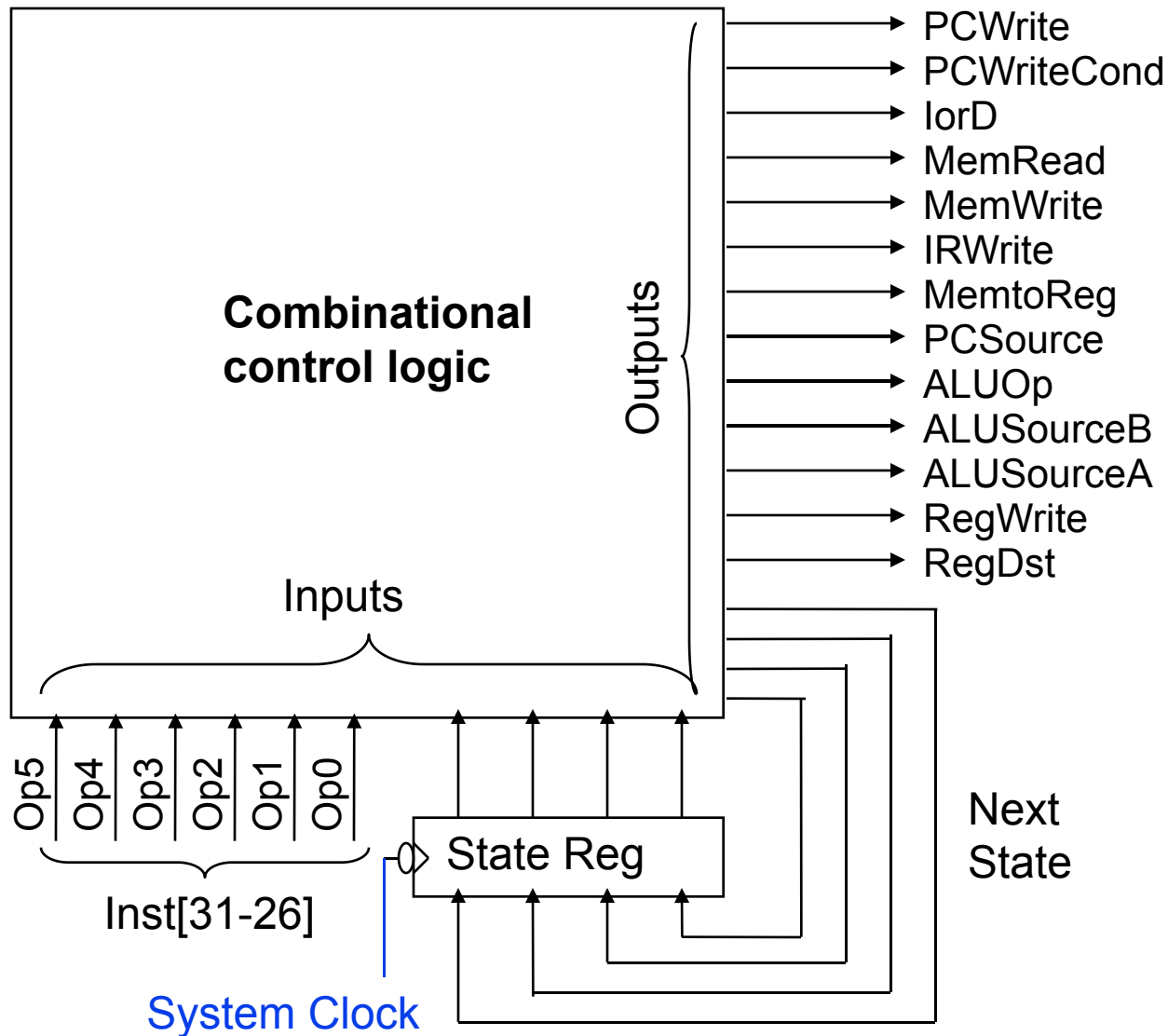


- ❑ So we are using a **Moore** machine (datapath control signals based only on current state)

Finite State Machine



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

输 出	当 前 状 态	操 作
PCWrite	state0 + state9	
PCWriteCond	state8	
lorD	state3 + state5	
MemRead	state0 + state3	
MemWrite	state5	
IRWrite	state0	
MemtoReg	state4	
PCSource1	state9	
PCSource0	state8	
ALUOp1	state6	
ALUOp0	state8	
ALUSrcB1	state1 + state2	
ALUSrcB0	state0 + state1	
ALUSrcA	state2 + state6 + state8	
RegWrite	state4 + state7	
RegDst	state7	

输 出	当 前 状 态	操 作
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

例题 下一状态输出的逻辑方程

给出下一状态最低位(NS0)的逻辑方程。

解 当下一个状态编码的 $NS0 = 1$ 时, 下一状态位 NS0 是有效的。这些状态有 NextState1, NextState3, NextState5, NextState7 和 NextState9。图 C-3-3 提供了这些状态有效的条件。下面分别给出这些下一状态的方程。第一个方程说明如果当前状态是状态 0, 则下一状态是状态 1; 右边的乘积式表明如果状态输入的每一位是 0, 则当前状态是状态 0。

$$\text{NextState1} = \text{State0} = \overline{S3} \cdot \overline{S2} \cdot \overline{S1} \cdot \overline{S0}$$

$$\begin{aligned}\text{NextState3} &= \text{State2} \cdot (\text{Op}[5-0] = lw) \\ &= \overline{S3} \cdot \overline{S2} \cdot S1 \cdot \overline{S0} \cdot \text{Op5} \cdot \overline{\text{Op4}} \cdot \overline{\text{Op3}} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \text{Op0}\end{aligned}$$

$$\begin{aligned}\text{NextState5} &= \text{State2} \cdot (\text{Op}[5-0] = sw) \\ &= \overline{S3} \cdot \overline{S2} \cdot S1 \cdot \overline{S0} \cdot \text{Op5} \cdot \overline{\text{Op4}} \cdot \text{Op3} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \text{Op0}\end{aligned}$$

$$\text{NextState7} = \text{State6} = \overline{S3} \cdot S2 \cdot S1 \cdot \overline{S0}$$

$$\begin{aligned}\text{NextState9} &= \text{State1} \cdot (\text{Op}[5-0] = jmp) \\ &= \overline{S3} \cdot \overline{S2} \cdot \overline{S1} \cdot S0 \cdot \overline{\text{Op5}} \cdot \overline{\text{Op4}} \cdot \overline{\text{Op3}} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \overline{\text{Op0}}\end{aligned}$$

NS0 是所有这些式子的逻辑和。

-
- ❑ 求NS1, NS2, NS3的逻辑表达式。
 - ❑ NS1有效: NextState2,或 NextState3,或
NextState6,或 NextState7---》 State1, 或State2,
或State6
 - ❑ NS2有效: ?
 - ❑ NS3有效: ?

16 个数据通路控制信号的真值表，它们的值仅取决于当前状态输入位

s3	s2	s1	s0
0	0	0	0
1	0	0	1

a) PCWrite 的真值表

s3	s2	s1	s0
1	0	0	0

b) PCWriteCond 的真值表

s3	s2	s1	s0
0	0	1	1
0	1	0	1

c) IorD 的真值表

s3	s2	s1	s0
0	0	0	0
0	0	1	1

d) MemRead 的真值表

s3	s2	s1	s0
0	1	0	1

e) MemWrite 的真值表

s3	s2	s1	s0
0	0	0	0

f) IRWrite 的真值表

s3	s2	s1	s0
0	1	0	0

g) MemtoReg 的真值表

s3	s2	s1	s0
1	0	0	1

h) PCSource1 的真值表

s3	s2	s1	s0
1	0	0	0

i) PCSource0 的真值表

s3	s2	s1	s0
0	1	1	0

j) ALUOp1 的真值表

s3	s2	s1	s0
1	0	0	0

k) ALUOp0 的真值表

s3	s2	s1	s0
0	0	0	1
0	0	1	0

l) ALUSrcB1 的真值表

s3	s2	s1	s0
0	0	0	0
0	0	0	1

m) ALUSrcB0 的真值表

s3	s2	s1	s0
0	0	1	0
0	1	1	0
1	0	0	0

n) ALUSrcA 的真值表

s3	s2	s1	s0
0	1	0	0
0	1	1	1

o) RegWrite 的真值表

s3	s2	s1	s0
0	1	1	1

p) RegDst 的真值表

下一状态的 4 个输出位(NS[3-0])的真值表

取决于操作码域 Op[5-0]的值和当前状态位 S[3-0]

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	0	1

a) NS3 输出的真值表, 在下一状态是 8 和 9 时有效。当前状态为 1 时该信号被激活。

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	1
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	0	1	1
X	X	X	X	X	X	0	1	1	0

b) NS2 输出的真值表, 在下一状态是 4, 5, 6 或 7 时有效。这种情况在当前状态为 1, 2, 3 或 6 时发生

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	0	1
1	0	1	0	1	1	0	0	0	1
1	0	0	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0

c) NS1 输出的真值表，在下一状态是 2, 3, 6 或 7 时有效。下一状态是 2, 3, 6 或 7 则当前状态是 1, 2 或 6

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
X	X	X	X	X	X	0	0	0	0
1	0	0	0	1	1	0	0	1	0
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0
0	0	0	0	1	0	0	0	0	1

d) NS0 输出的真值表，在下一状态是 1, 3, 5, 7 或 9 时有效。仅在当前状态是 0, 1, 2 或 6 时发生

1.使用ROM实现控制器

- ❑ 将真值表编码存放在ROM中：输入10位，包含6位OP，4位states；输出20位，16位控制信号（13类），4位下一状态NextState[3-0]；
- ❑ 真值表表项共 $2^{10}=1024$ 项（ROM全地址访问），每项大小20位：要求ROM空间至少 $1024*20=20Kb$ 。

[illegible]

地址低 4 位	控制字的 19~4 位
0000	1001010000001000
0001	0000000000011000
0010	0000000000010100
0011	0011000000000000
0100	0000001000000010
0101	0010100000000000
0110	0000000001000100
0111	0000000000000011
1000	0100000010100100
1001	1000000100000000

图 C-3-7 仅取决于状态输入的
ROM 高 16 位的内容

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

· 分为两个独立的表不仅仅是一种简化显示 ROM 内容的方法，它也是一种更有效地实现 ROM 的方法。大部分输出(16 位)仅取决于 10 个输入中的 4 个。用两个独立的 ROM 来实现控制所需的位数是 $2^4 \times 16 + 2^{10} \times 4 = 256 + 4096 = 4.3 \text{ Kbit}$ ，这是只用一块 ROM 的容量的 1/5，一块 ROM 本来将耗费 $2^{10} \times 20 = 20 \text{ Kbit}$ 。结构化逻辑块会有一些开销，但在这种情况下额外的一块 ROM 的开销要比分开一块 ROM 的节省少很多。

□ 两张表分别是：4位地址，16位输出，容量： $2^4 \times 16$ bit;
(把当前状态作为地址输入)

□ 10位地址，4位输出，容量： $2^{10} \times 4$ bit.
(把当前状态和OP作为地址输入)

2.使用PLA实现控制器

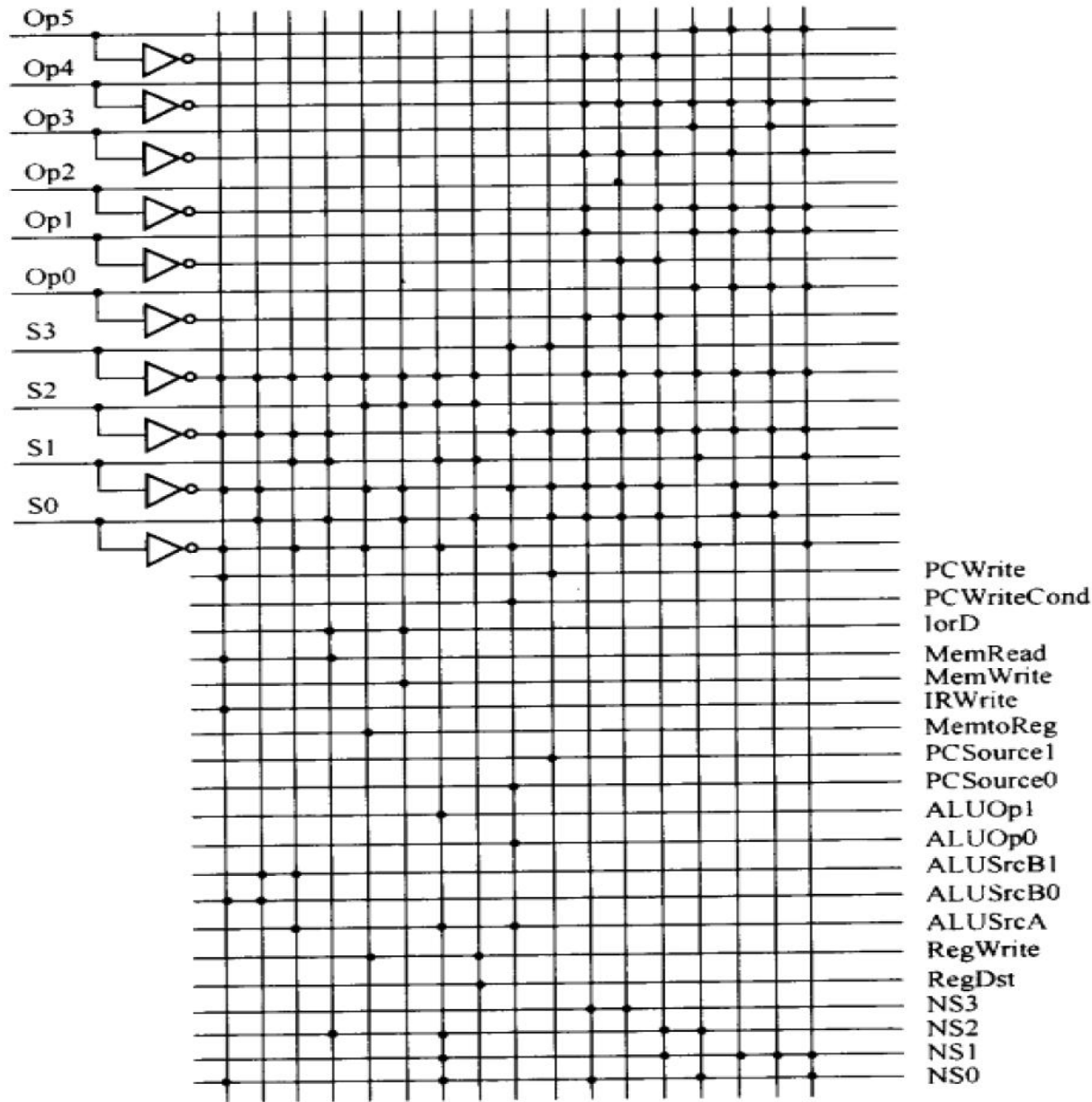


图 C-3-9 控制函数逻辑多周期的 PLA 实现

[左边是控制的输入，右边是控制的输出。图的上半部分是计算所有小项的与部分。小项由垂直的线传到或部分，每个黑点对应产生小项的信号。和式由这些小项计算得出，每个灰点代表和式中用到的一个小项。每个输出包含一个简单的和式]

从图 C-3-9 的 PLA 可以看到，有 17 个惟一的小项——其中 10 个仅取决于当前状态，另外 7 个取决于 Q_p 域和当前状态位。PLA 的大小和(输入数 \times 小项数) + (输出数 \times 小项数) 的值成正比，我们可以从图中看出这个关系。因此，图 C-3-9 中 PLA 的大小和 $(10 \times 17) + (20 \times 17) = 510$ 成正比。而整块 ROM 的大小是 20Kbit，即使用两块 ROM 也需要 4.3Kbit。由于 PLA 单元的大小只比 ROM 的存储单元稍大，所以 PLA 是实现控制单元更为高效的方法。

就像将 ROM 分为两部分，我们也可以将 PLA 分成两部分：一个包含 4 个输入和 10 个小项，产生 16 个控制输出；一个有 10 个输入和 7 个小项，产生 4 个下一状态输出。第一块 PLA 的大小和 $(4 \times 10) + (10 \times 16) = 200$ 成正比，第二块的大小和 $(10 \times 7) + (4 \times 7) = 98$ 成正比。这样整个 PLA 的大小就是 298 个 PLA 单元，是用一块 PLA 所需大小的 55%。这两块 PLA 的大小要比两块 ROM 的大小小很多。有关 PLA 的更多细节以及实现方法，包括逻辑设计的一些参考书目，请看 ■附录 B。

Simplifying the Control Unit Design

- ❑ For an implementation of the full MIPS ISA instr's can take from 3 clock cycles to 20+ clock cycles
 - resulting in finite state machines with hundreds to thousands of states with even *more* arcs (state sequences)
 - ❏ Such state machine representations become impossibly complex
- ❑ Instead, can represent the set of control signals that are asserted during a state as a low-level control “instruction” to be executed by the datapath

microinstructions vs states

- ❑ “Executing” the microinstruction is equivalent to asserting the control signals specified by the microinstruction


3. Microprogramming

- ❑ A microinstruction has to specify
 - what control signals should be asserted
 - what microinstruction should be executed next
- ❑ Each microinstruction corresponds to one state in the FSM and is assigned a state number (or “address”)
 1. **Sequential** behavior – increment the state (address) of the current microinstruction to get to the state (address) of the next
 2. **Jump** to the microinstruction that begins execution of the next MIPS instruction (state 0)
 3. **Branch** to a microinstruction based on control unit input using dispatch tables
 - ❑ need one for microinstructions following state 1
 - ❑ need another for microinstructions following state 2
- ❑ The set of microinstructions that define a MIPS assembly language instruction is its **microroutine**

Defining a Microinstruction Format

❑ **Format** – the fields of the microinstruction

- **control signals** specified by a field usually have functions that are related
- format is chosen to simplify the representation and to make it difficult to write inconsistent microinstructions

 i.e., that allow a given control signal be set to two different values

❑ Make each field of the microinstruction responsible for specifying a **nonoverlapping** set of control signals

- signals that are never asserted simultaneously may share the same field
- **seven fields** for our simple machine

 ALU control; SRC1; SRC2; Register control; Memory; PCWrite control; **Sequencing**

Our Microinstruction Format

Field	Value	Signal setting	Comments
ALU control (2bit)	Add	ALUOp = 00	Cause ALU to add
	Subt	ALUOp = 01	Cause ALU to subtract (compare op for beq)
	Func code	ALUOp = 10	Use IR function code to determine ALU control
SRC1 (1bit)	PC	ALUSrcA = 0	Use PC as top ALU input
	A	ALUSrcA = 1	Use reg A as top ALU input
SRC2 (2bit)	B	ALUSrcB = 00	Use reg B as bottom ALU input
	4	ALUSrcB = 01	Use 4 as bottom ALU input
	Extend	ALUSrcB = 10	Use sign ext output as bottom ALU input
	Extshft	ALUSrcB = 11	Use shift-by-two output as bottom ALU input
Register control (2bit)	Read		Read RegFile using rs and rt fields of IR as read addr's; put data into A and B
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write RegFile using rd field of IR as write addr and ALUOut as write data
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write RegFile using rt field of IR as write addr and MDR as write data

Our Microinstruction Format, con't

Field	Value	Signal setting	Comments
Memory (2bit)	Read PC	MemRead, lorD = 0,IRWrite	Read memory using PC as addr; write result into IR (and MDR)
	Read ALU	MemRead, lorD = 1	Read memory using ALUOut as addr; write results into MDR
	Write ALU	MemWrite, lorD = 1	Write memory using ALUOut as addr and B as write data
PC write control (2bit)	ALU	PCSource = 00 PCWrite	Write PC with output of ALU
	ALUOut-cond	PCSource = 01, PCWriteCond	If Zero output of ALU is true, write PC with the contents of ALUOut
	Jump address	PCSource = 10, PCWrite	Write PC with IR jump address after shift-by-two
Sequencing (2bit)	Seq	AddrCtl = 11	Choose next microinstruction sequentially
	Fetch	AddrCtl = 00	Jump to the first microinstruction (i.e., Fetch) to begin a new instruction
	Dispatch 1	AddrCtl = 01	Branch using PLA_1
	Dispatch 2	AddrCtl = 10	Branch using PLA_2

Dispatch (Branch) Logic

- ❑ Dispatch operations are implemented using special logic (PLAs)
- ❑ 由于ROM表采用全地址，每张表6位地址，有64项，每项大小4b,表容量为256b.

Microcode Dispatch PLA_1		
Opcode field	Opcode	Value (Addr)
000000	R-format	Rexec (6)
000010	jmp	Jump (9)
000100	beq	Beq (8)
100011	lw	Maddr (2)
101011	sw	Maddr (2)

Microcode Dispatch PLA_2		
Opcode field	Opcode	Value (Addr)
100011	lw	Memlw (3)
101011	sw	Memsw (5)

Creating the Microprogram

❑ Fetch microinstruction

Label (Addr)	ALU control	SRC1	SRC2	Reg control	Memory	PCWrite control	Seq'ing
Fetch (0)	Add	PC	4		Read PC	ALU	Seq

compute PC + 4

fetch instr
into IR

write ALU
output into
PC

go to μ instr 1

❑ Bit pattern: 微地址4位: 控制信号11位+顺序控制2位

0000: 00 0 10 00 00 00 11

❑ Label field represents the state (address) of the microinstruction

❑ Fetch microinstruction assigned state (address) 0

The Entire Control Microprogram

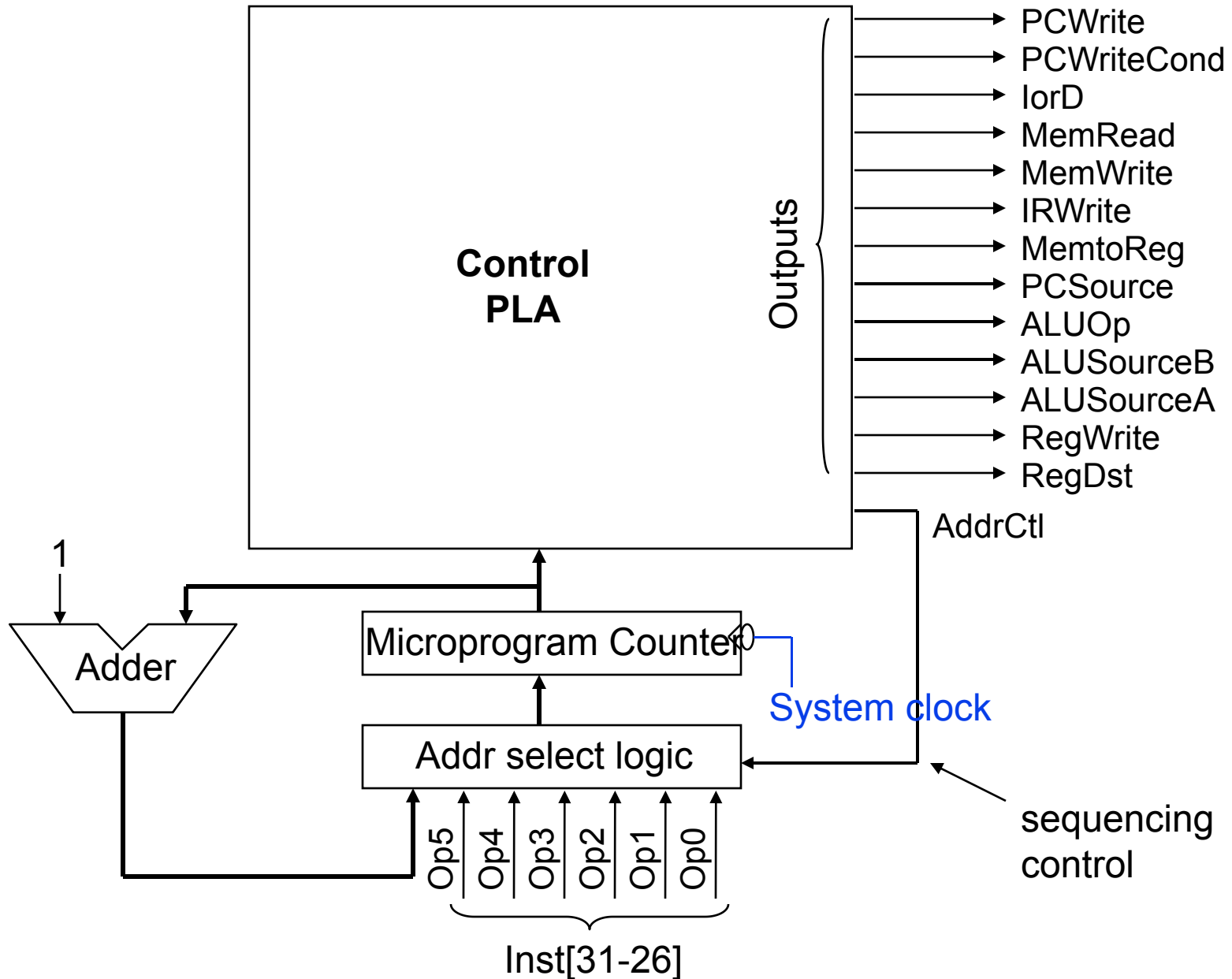
Addr	ALU control	SRC1	SRC2	Reg control	Memory	PCWrite control	Seq'ing
0	Add	PC	4		Read PC	ALU	Seq
1	Add	PC	Ext shft	Read			Disp 1
2	Add	A	Extend				Disp 2
3					Read ALU		Seq
4				Write MDR			Fetch
5					Write ALU		Fetch
6	Func code	A	B				Seq
7				Write ALU			Fetch
8	Subt	A	B			ALUOut-cond	Fetch
9						Jump address	Fetch

图 C-4-5 显示了整个控制 ROM 的内容。控制所需的存储容量是相当小的。共有 10 个控制字，每个 18 位宽，共 180 位。另外，两个调度表是 4 位宽，每个有 64 个表项，一共是 512 位。全部 692 位要比用两块 ROM 的实现并且其中一块 ROM 中存有下一状态编码(ROM 需要 4.3 Kbit)需要的存储空间少得多。 ---此处采用纯水平编码，每个信号占一位

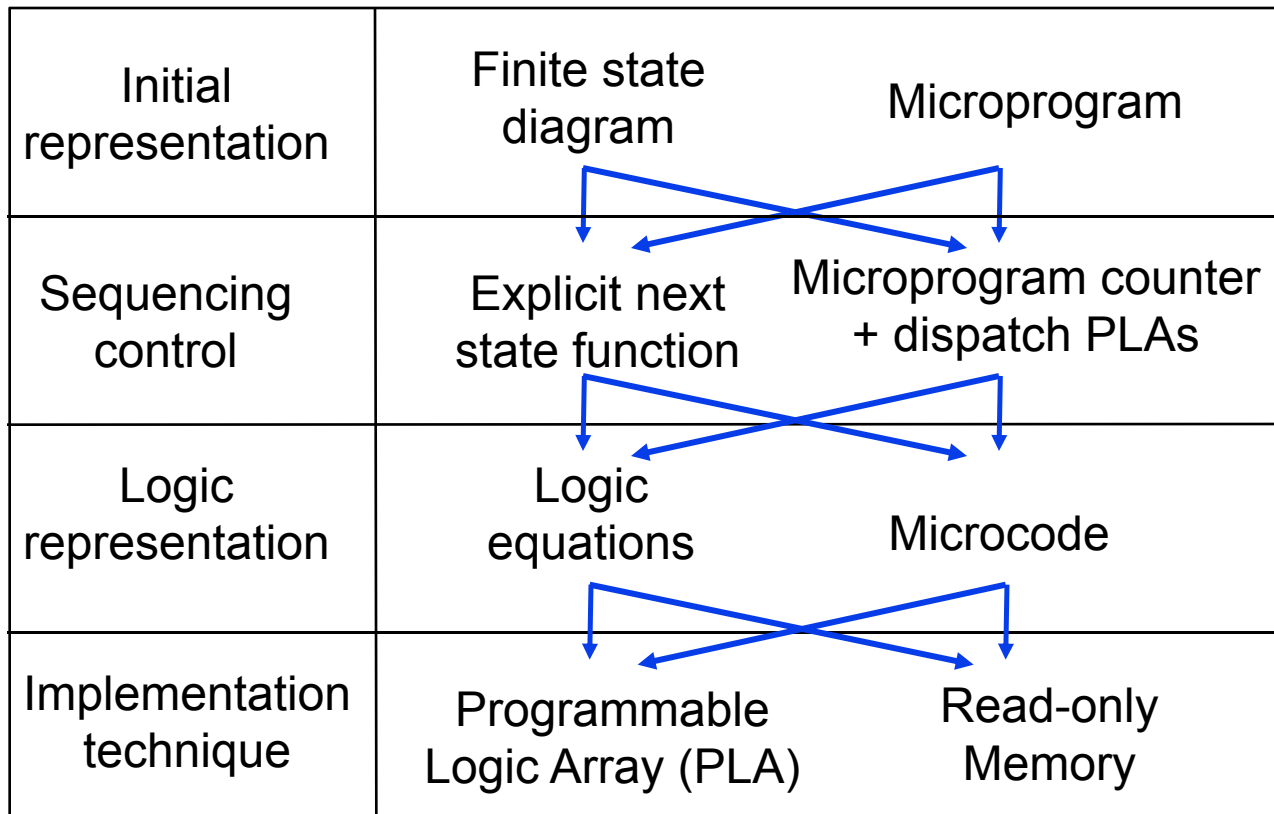
状 态 值	控制字 17~2 位	控制字 1~0 位
0	1001010000001000	11
1	0000000000011000	01
2	0000000000010100	10
3	0011000000000000	11
4	0000001000000010	00
5	0010100000000000	00
6	0000000001000100	11
7	0000000000000011	00
8	0100000010100100	00
9	1000000100000000	00

图 C-4-5 使用计数器的实现中控存的内容

Microcode Implementation



Control Path Design Alternatives



hardwired control microprogrammed control

❑ Microprogram representation advantages

- Easier to design, write, and debug

Multicycle Advantages & Disadvantages

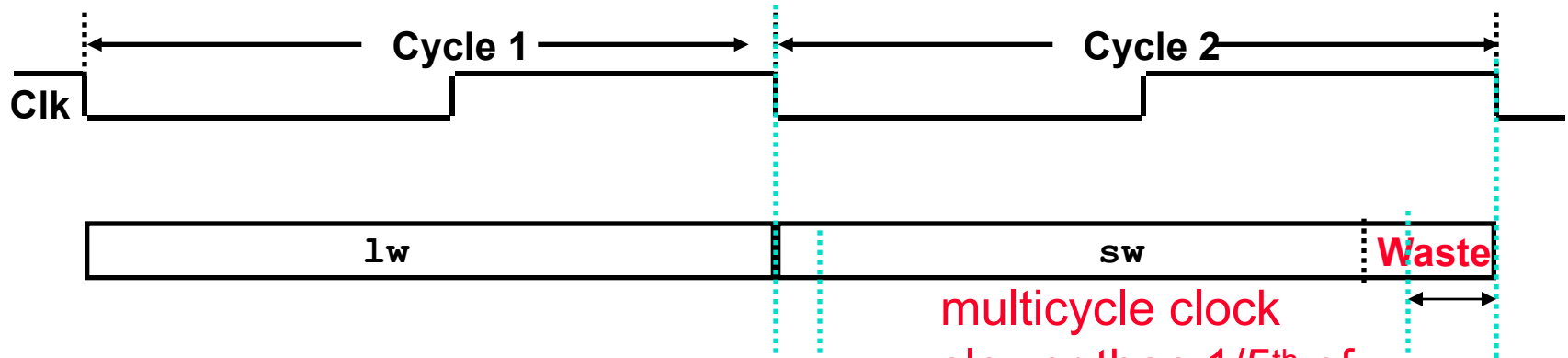
- ❑ Uses the clock cycle efficiently – the clock cycle is timed to accommodate the slowest instruction **step**
 - **balance** the amount of work to be done in each step
 - restrict each step to use only one major functional unit
- ❑ Multicycle implementations allow
 - faster clock rates
 - different instructions to take a different number of clock cycles
 - functional units to be used more than once per instruction as long as they are used on different clock cycles

but

- ❑ Requires additional internal state registers, muxes, and more complicated (FSM) control

Single Cycle vs. Multiple Cycle Timing

Single Cycle Implementation:



Multiple Cycle Implementation:

