

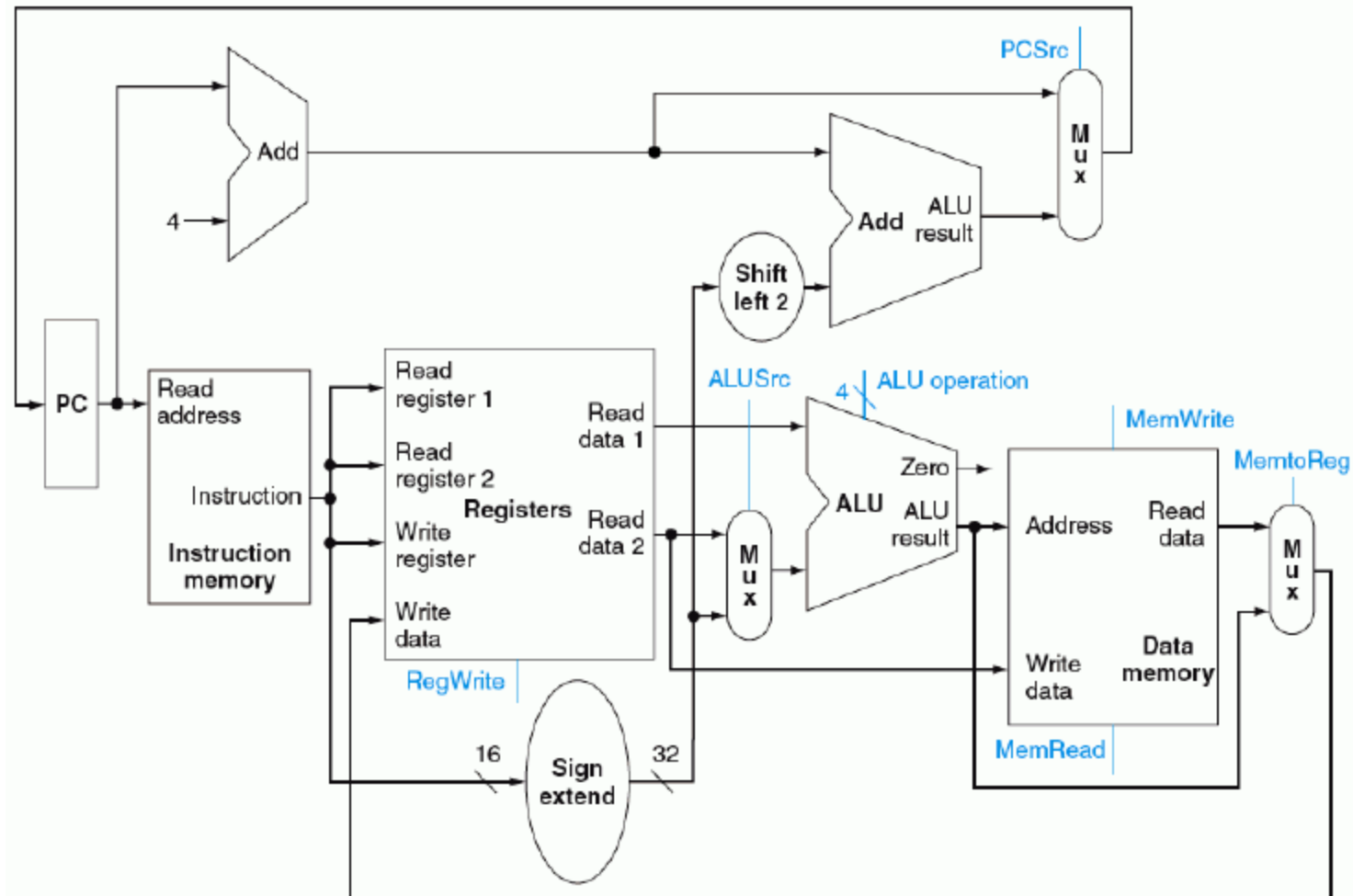
Computer Architecture

Designing a Multiple Cycle Processor

Outline of Today's Lecture

- **Introduction to the Concept of Multiple Cycle Processor**
- **Multiple Cycle Implementation of R-type Instructions**
- **What is a Multiple Cycle Delay Path?**
- **Multiple Cycle Implementation of Or Immediate**
- **Multiple Cycle Implementation of Load and Store**
- **Putting it all Together**

Single cycle Datapath



What's wrong with our CPI=1 processor?

Arithmetic & Logical



Load



← *Critical Path* →

Store



Branch



- Long Cycle Time
- All instructions take as much time as the slowest
- Real memory is not so nice as our idealized memory
 - cannot always get the job done in one (short) cycle

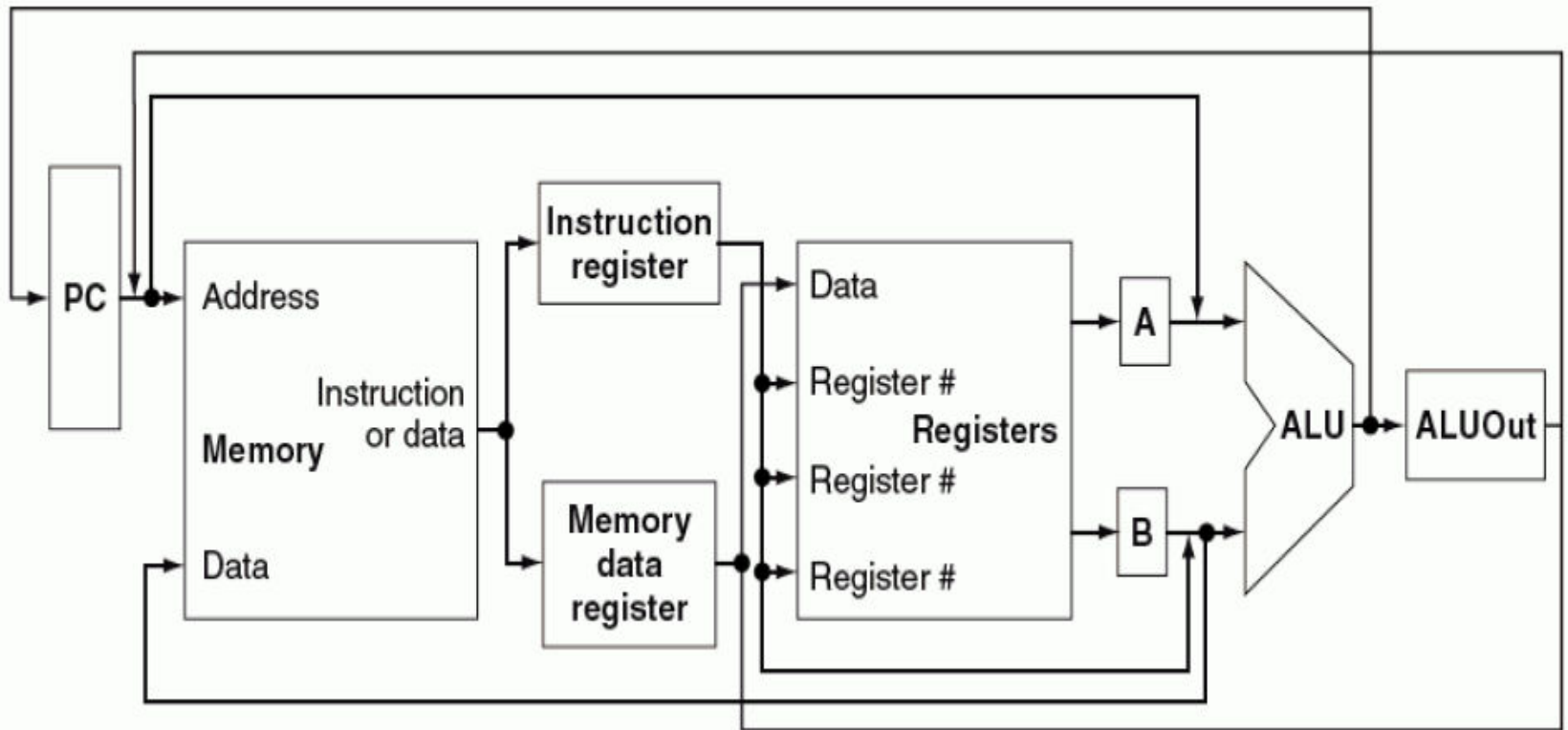
Drawbacks of this Single Cycle Processor

- Long cycle time:
 - Cycle time must be long enough for the load instruction:
 - PC's Clock -to-Q +
 - Instruction Memory Access Time +
 - Register File Access Time +
 - ALU Delay (address calculation) +
 - Data Memory Access Time +
 - Register File Setup Time +
 - Clock Skew
- Cycle time is much longer than needed for all other instructions.
Examples:
 - R-type instructions do not require data memory access
 - Jump does not require ALU operation nor data memory access

Overview of a Multiple Cycle Implementation

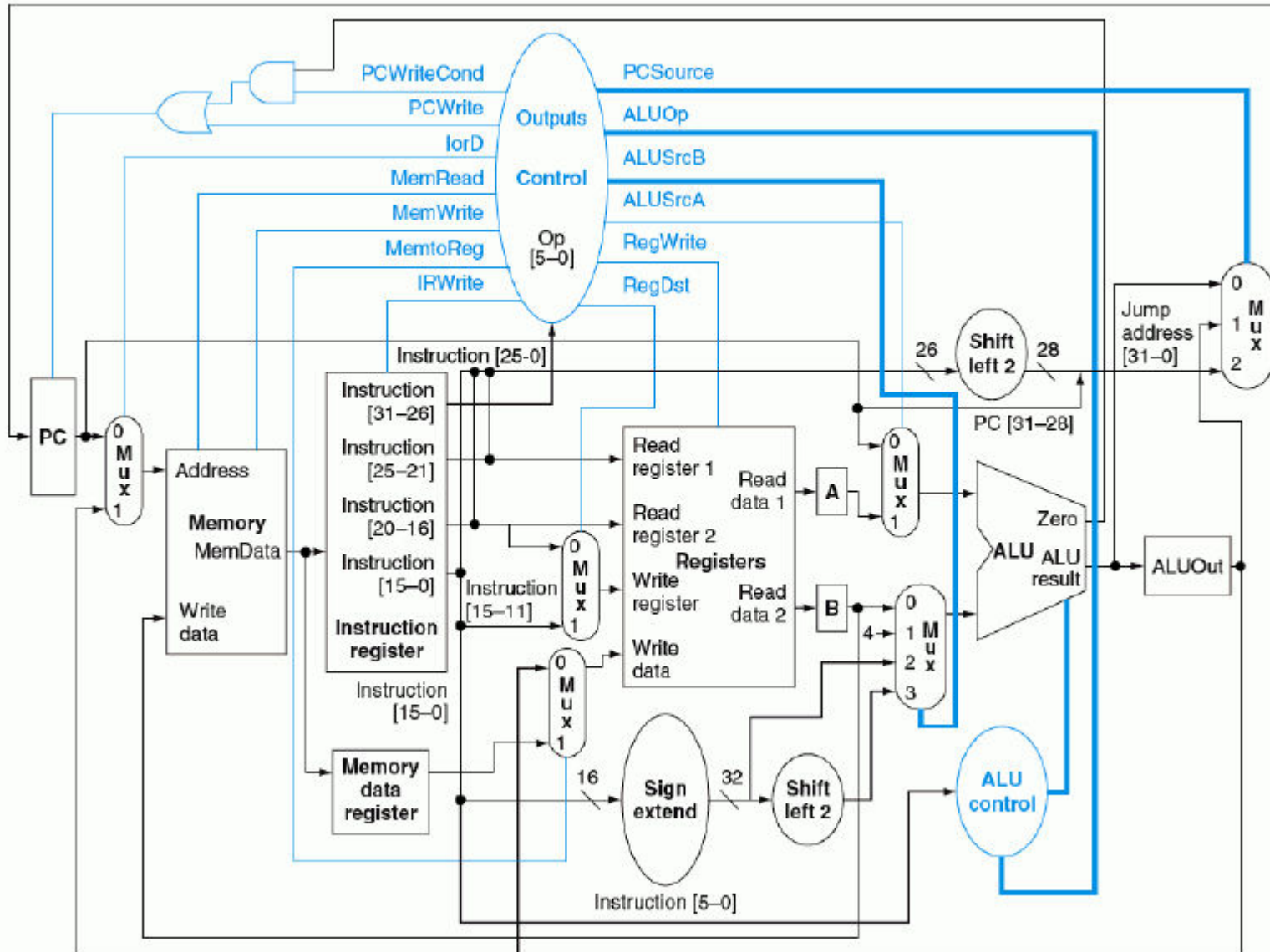
- The root of the single cycle processor's problems:
 - The cycle time has to be long enough for the slowest instruction
- Solution:
 - Break the instruction into smaller steps
 - Execute each step (instead of the entire instruction) in one cycle
 - Cycle time: time it takes to execute the longest step
 - Keep all the steps to have similar length
 - This is the essence of the multiple cycle processor
- The advantages of the multiple cycle processor:
 - Cycle time is much shorter
 - Different instructions take different number of cycles to complete
 - Load takes five cycles
 - Jump only takes three cycles
 - Allows a functional unit to be used more than once per instruction

Multi-cycle MIPS Datapath Design



Basic differences from single cycle datapath

- A single memory unit is used for both instructions and data.
- There is a single ALU, rather than an ALU and two adders.
- One or more registers are added after every major functional unit to hold the output of that unit until the value is used in a subsequent clock cycle.



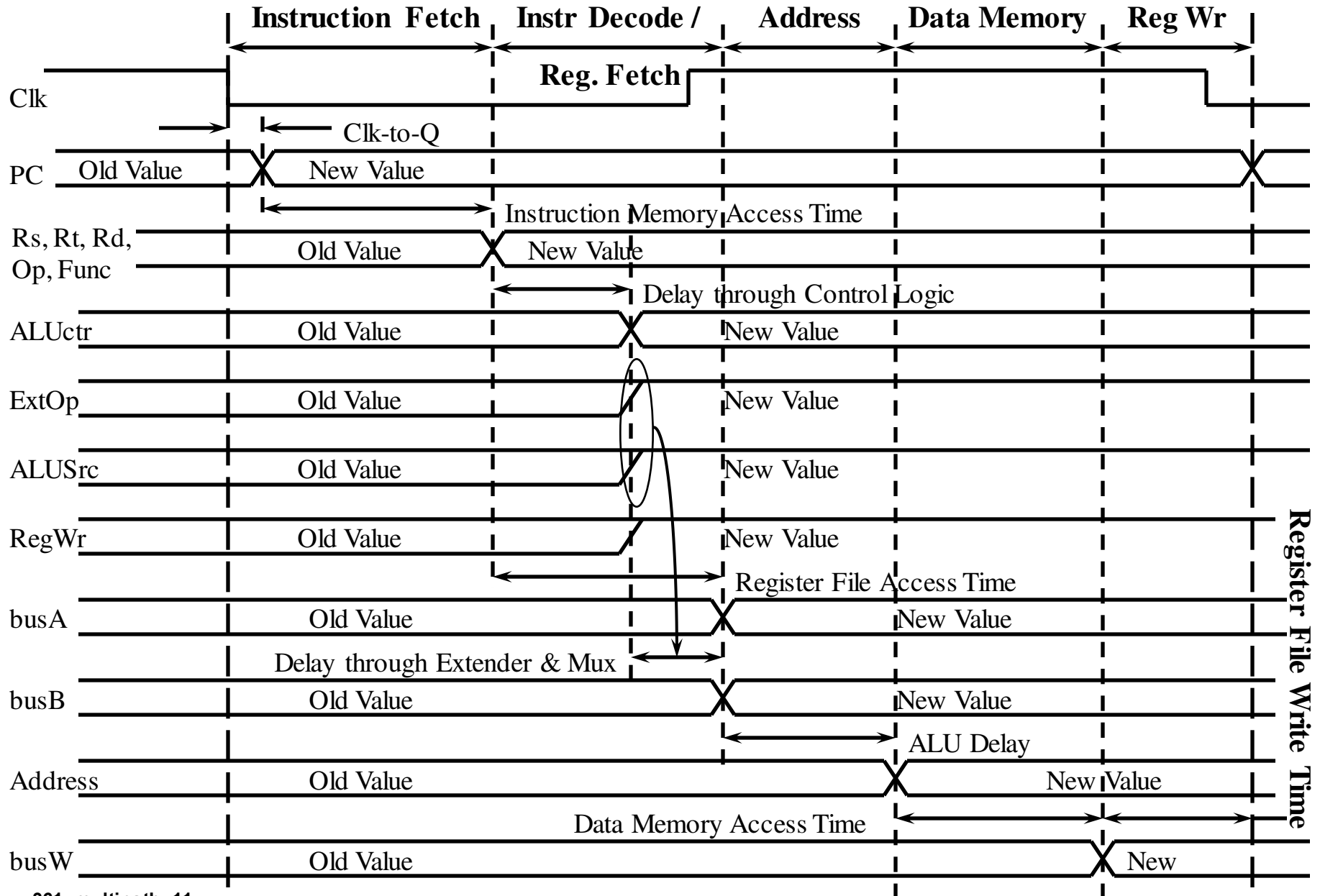
Actions of the 1-bit control signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
lrd	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSource.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

Actions of the 2-bit control signals

Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU ($PC + 4$) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25:0] shifted left 2 bits and concatenated with $PC + 4[31:28]$) is sent to the PC for writing.

The Five Steps of a Load Instruction



Dual-Port Ideal Memory

- **Dual Port Ideal Memory**

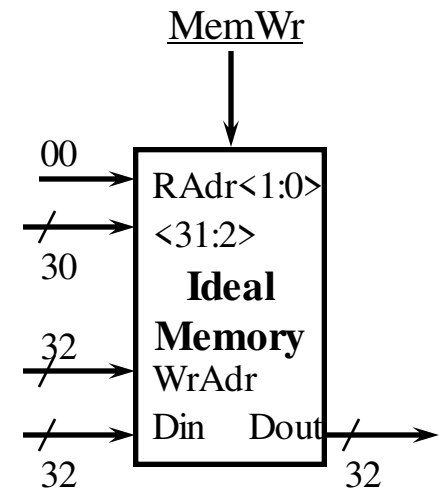
- Independent Read (RAdr, Dout) and Write (WAdr, Din) ports
- Read and write (to different location) can occur at the same cycle

- **Read Port is a combinational path:**

- Read Address Valid -->
- Memory Read Access Delay -->
- Data Out Valid

- **Write Port is also a combinational path:**

- MemWrite = 1 -->
- Memory Write Access Delay -->
- Data In is written into location[WrAdr]



MIPS Instruction Cycles

1. Instruction fetch step

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

2. Instruction decode and register fetch step

```
A <= Reg[IR[25:21]];  
B <= Reg[IR[20:16]];  
ALUOut <= PC + (sign-extend (IR[15:0]) << 2);
```

3. Execution, memory address computation, or branch completion

Memory reference:

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


Arithmetic-logical instruction (R-type):

```
ALUOut <= A op B;
```

Jump:

```
# {x, y} is the Verilog notation for concatenation of  
bit fields x and y  
PC <= {PC [31:28], (IR[25:0]), 2'b00});
```

Branch

PC  **ALUout**

4. Memory access or R-type instruction completion step

Memory reference:

MDR \leftarrow Memory [ALUOut];

or

Memory [ALUOut] \leftarrow B;

Arithmetic-logical instruction (R-type):

Reg[IR[15:11]] \leftarrow ALUOut;

5. Memory read completion step

Load:

Reg[IR[20:16]] \leftarrow MDR;

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	$IR \leftarrow \text{Memory}[PC]$ $PC \leftarrow PC + 4$			
Instruction decode/register fetch	$A \leftarrow \text{Reg}[IR[25:21]]$ $B \leftarrow \text{Reg}[IR[20:16]]$ $ALUOut \leftarrow PC + (\text{sign-extend}(IR[15:0]) \ll 2)$			
Execution, address computation, branch/jump completion	$ALUOut \leftarrow A \text{ op } B$	$ALUOut \leftarrow A + \text{sign-extend}(IR[15:0])$	if $(A == B)$ $PC \leftarrow ALUOut$	$PC \leftarrow \{PC[31:28], (IR[25:0], 2'b00)\}$
Memory access or R-type completion	$\text{Reg}[IR[15:11]] \leftarrow ALUOut$	Load: $MDR \leftarrow \text{Memory}[ALUOut]$ or Store: $\text{Memory}[ALUOut] \leftarrow B$		
Memory read completion		Load: $\text{Reg}[IR[20:16]] \leftarrow MDR$		

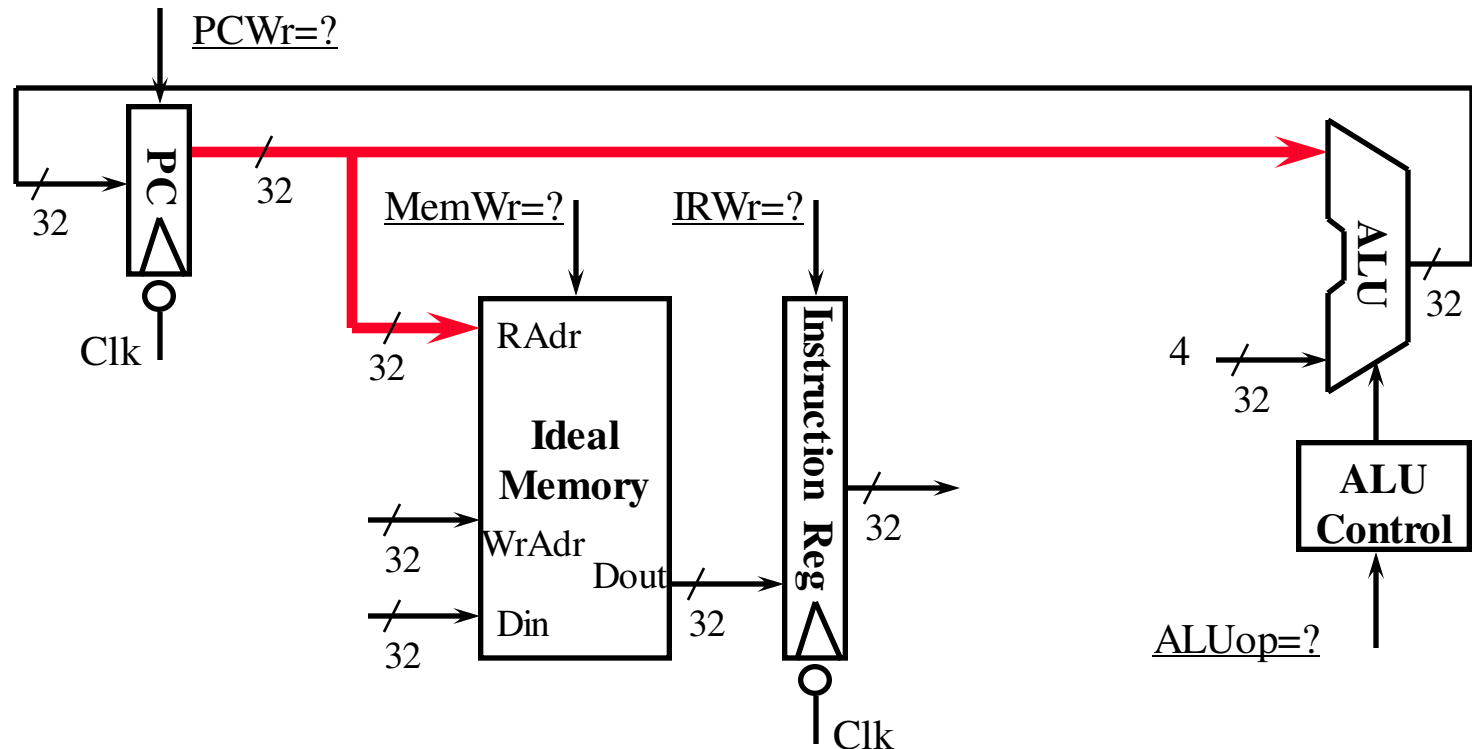
Instruction Fetch Cycle: In the Beginning

° Every cycle begins right **AFTER** the clock tick:

- $\text{mem}[\text{PC}] \quad \text{PC} \langle 31:0 \rangle + 4$



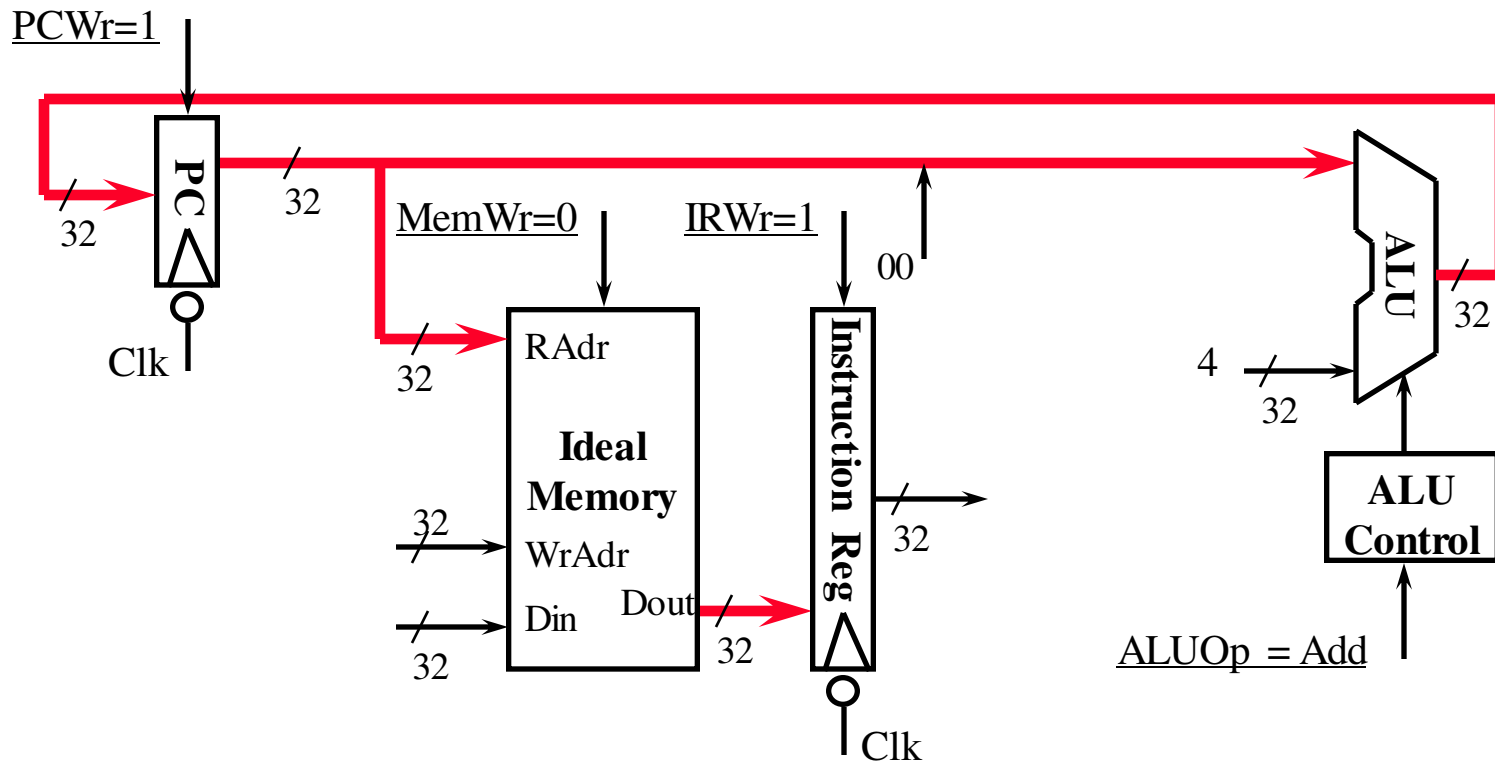
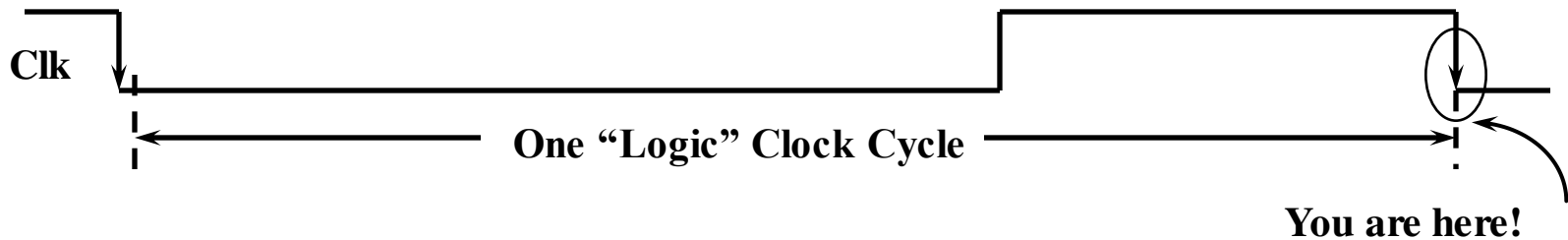
You are here!



Instruction Fetch Cycle: The End

° Every cycle ends AT the next clock tick (storage element updates):

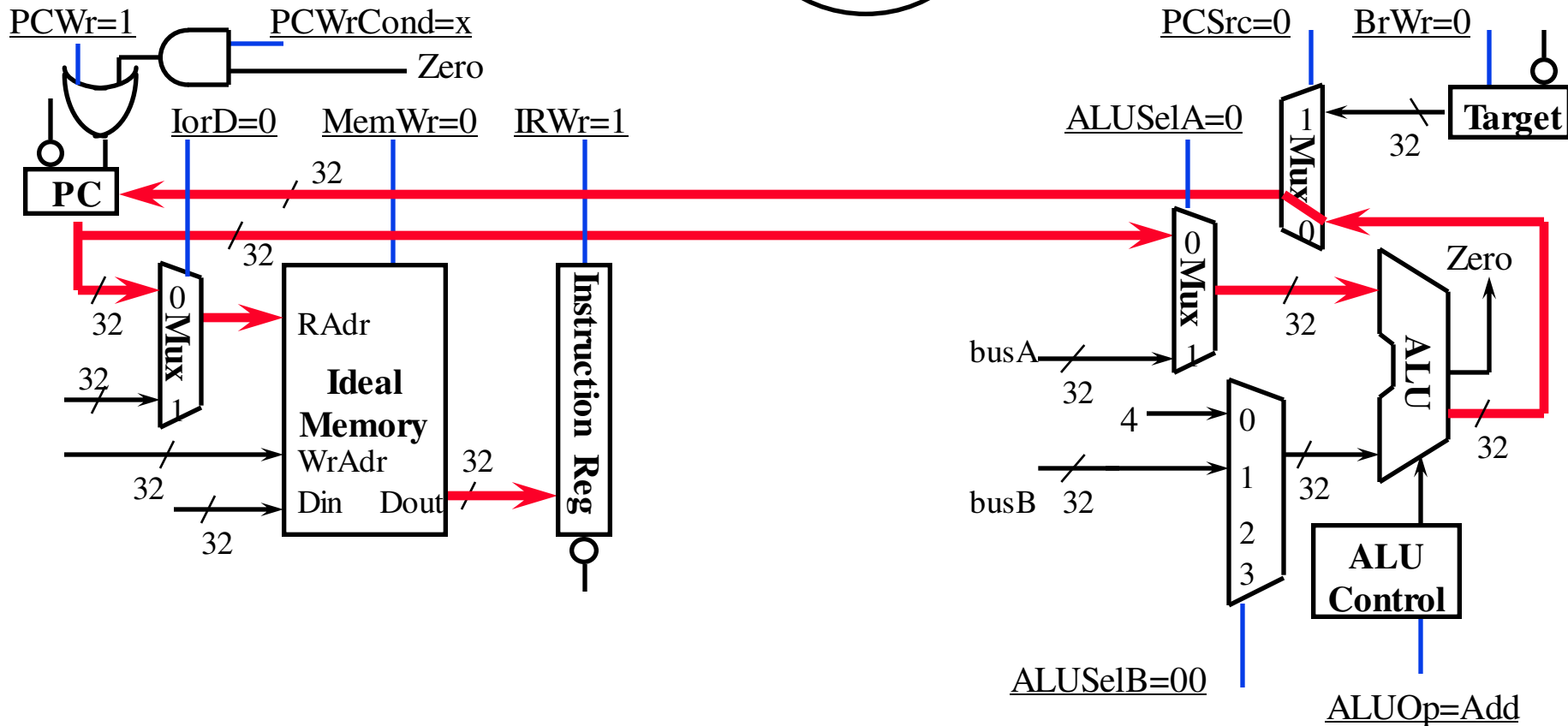
- $IR \leftarrow \text{mem}[PC]$ $PC_{<31:0>} \leftarrow PC_{<31:0>} + 4$



Instruction Fetch Cycle: Overall Picture

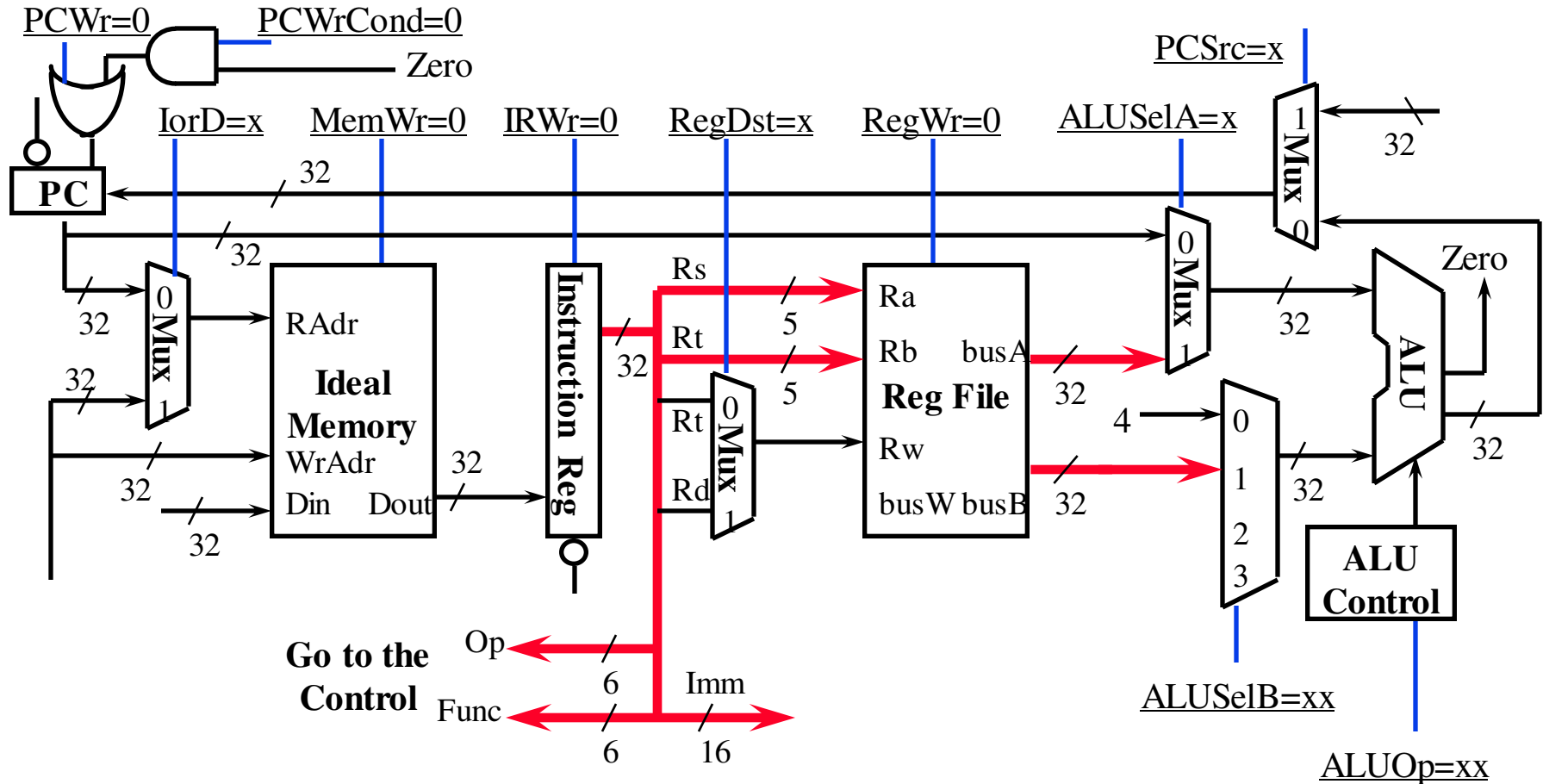
Ifetch

ALUOp=Add
1: PCWr, IRWr
x: PCWrCond
RegDst, Mem2R
Others: 0s



Register Fetch / Instruction Decode

- $\text{busA} \leftarrow \text{RegFile}[\text{rs}] ; \text{busB} \leftarrow \text{RegFile}[\text{rt}] ;$
- ALU is not being used: $\text{ALUctr} = \text{xx}$



Register Fetch / Instruction Decode (Continue)

- ° $\text{busA} \leftarrow \text{Reg}[\text{rs}] ; \text{busB} \leftarrow \text{Reg}[\text{rt}] ;$
- ° $\text{Target} \leftarrow \text{PC} + \text{SignExt}(\text{Imm16}) * 4$

Rfetch/Decode

ALUOp=Add

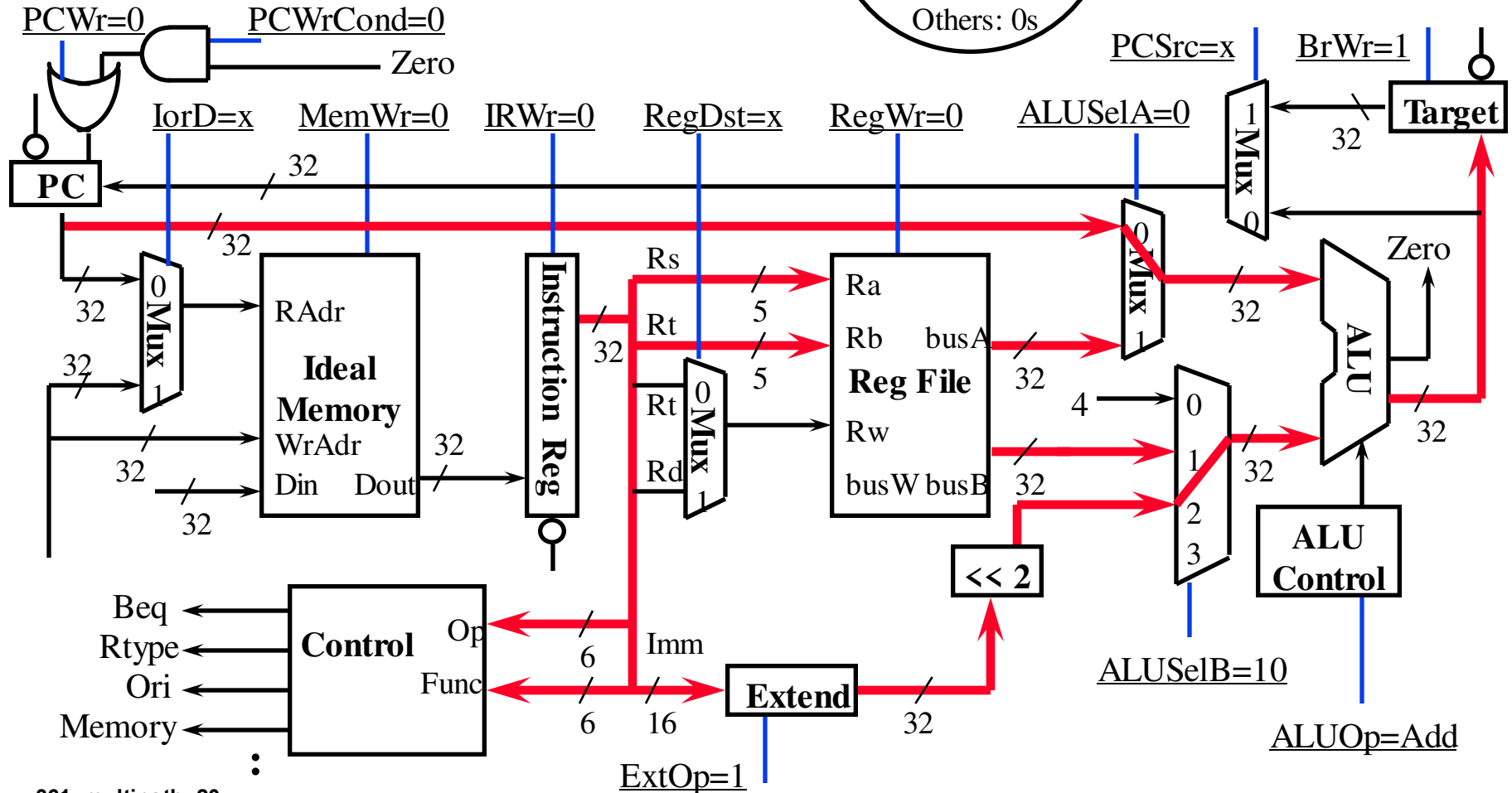
1: BrWr, ExtOp

ALUSelB=10

x: RegDst, PCSrc

IorD, MemtoReg

Others: 0s

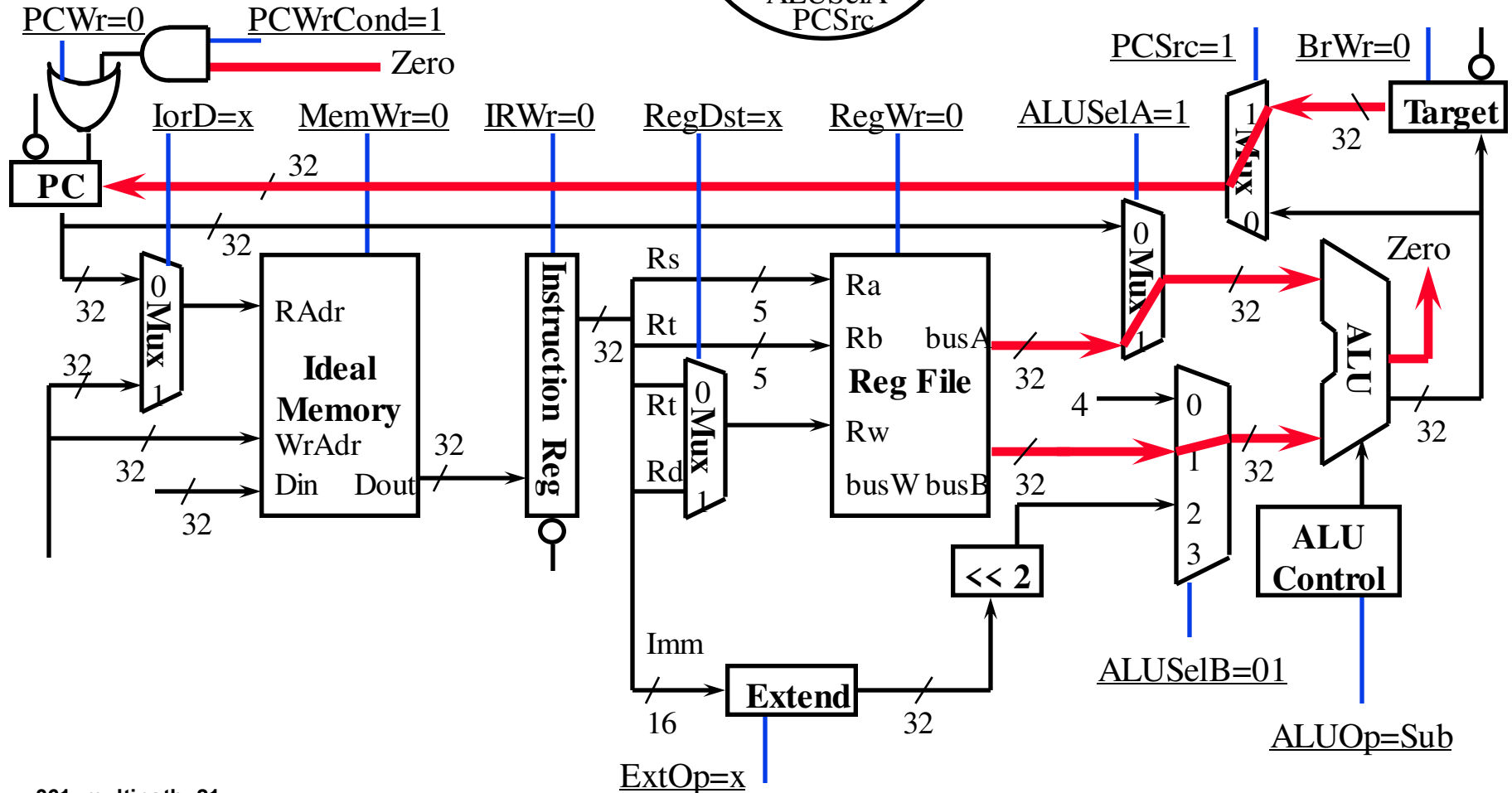


Branch Completion

- if (busA == busB)
 - PC <- Target

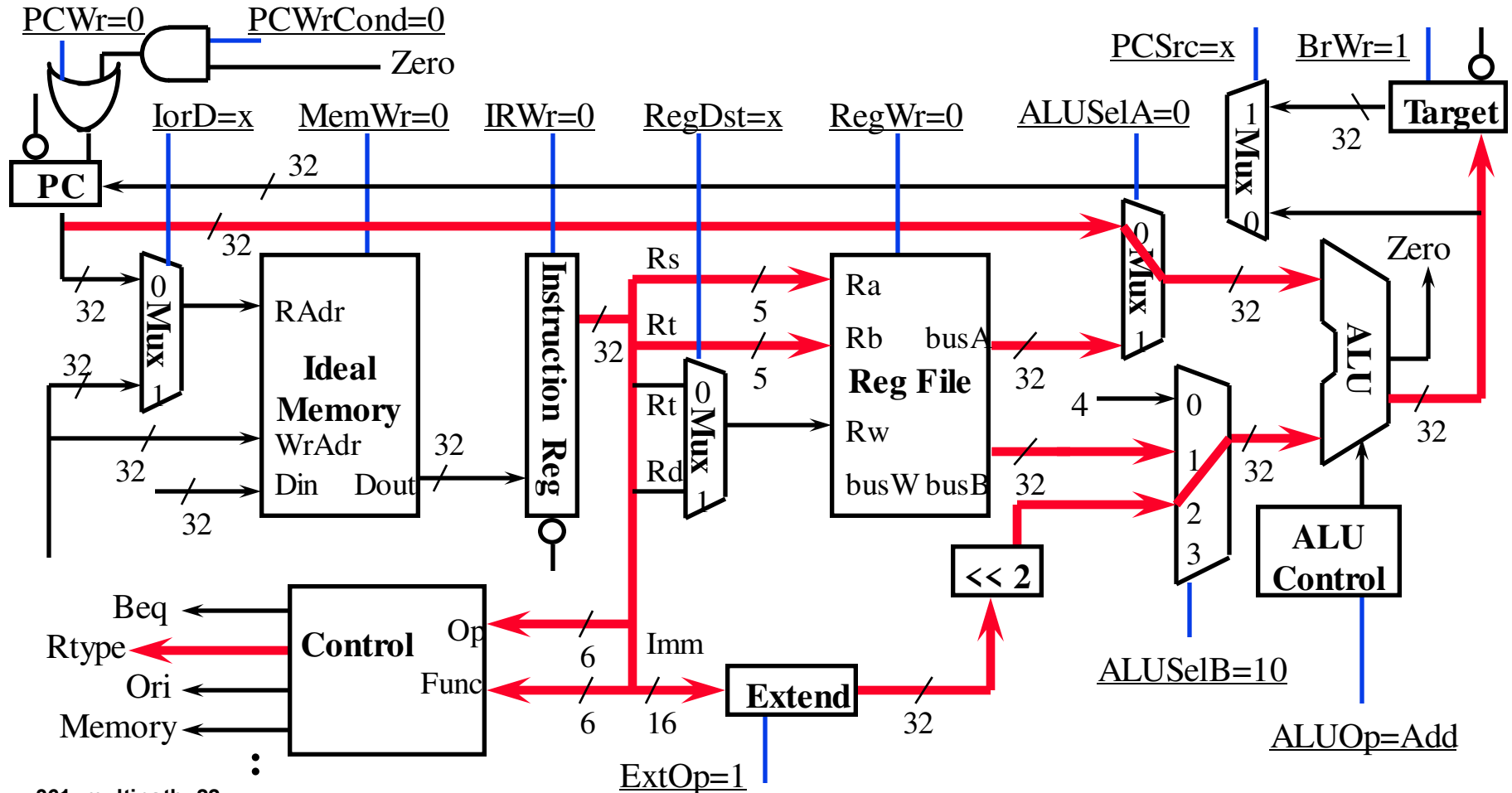
BrComplete

ALUOp=Sub
 ALUSelB=01
 x: IorD, Mem2Reg
 RegDst, ExtOp
 1: PCWrCond
 ALUSelA
 PCSrc



Instruction Decode: We have a R-type!

◦ Next Cycle: R-type Execution

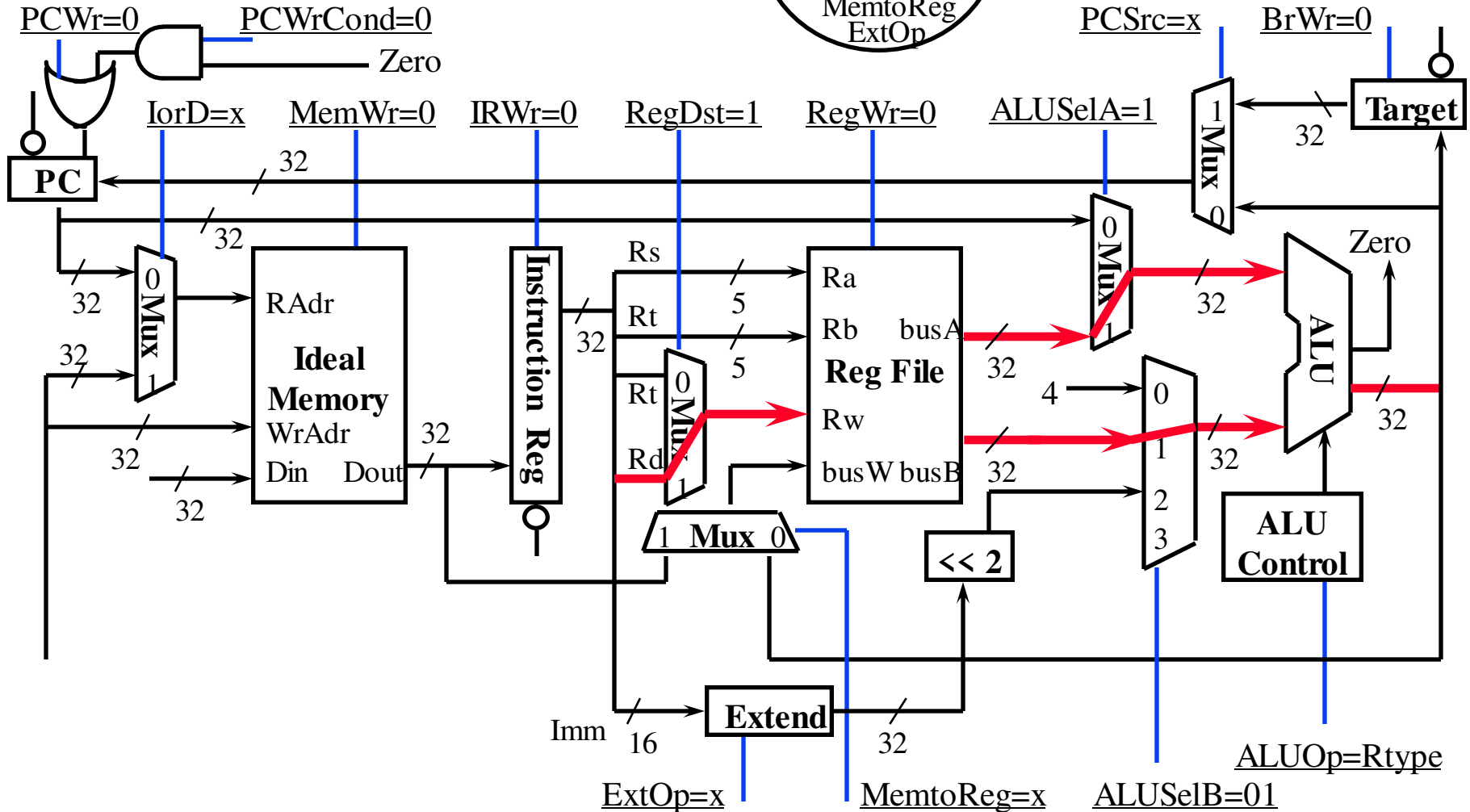


R-type Execution

° ALU Output <- busA op busB

RExec

1: RegDst
ALUSelA
ALUSelB=01
ALUOp=Rtype
x: PCSrc, IorD
MentoReg
ExtOp

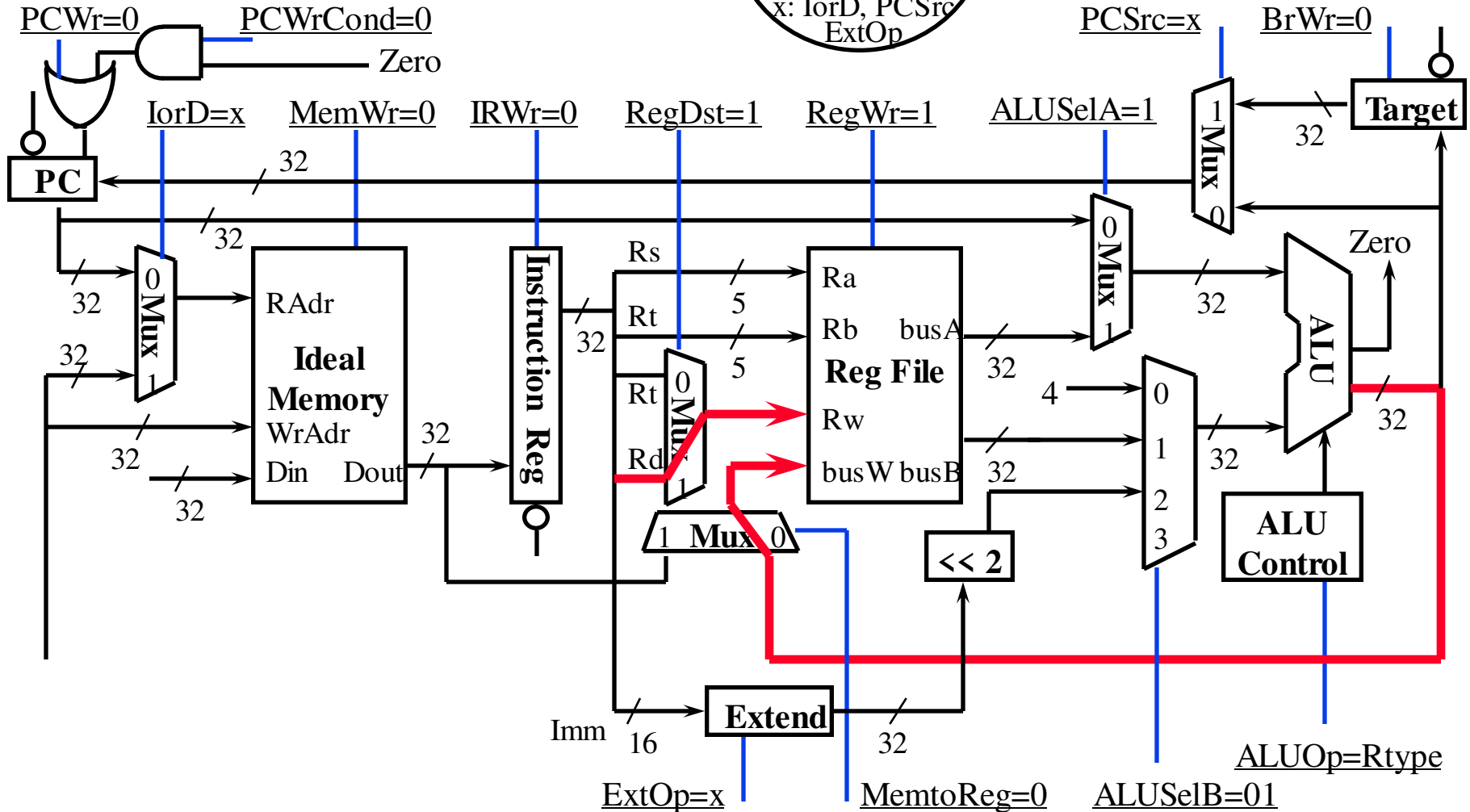


R-type Completion

° $R[rd] \leftarrow \text{ALU Output}$

Rfinish

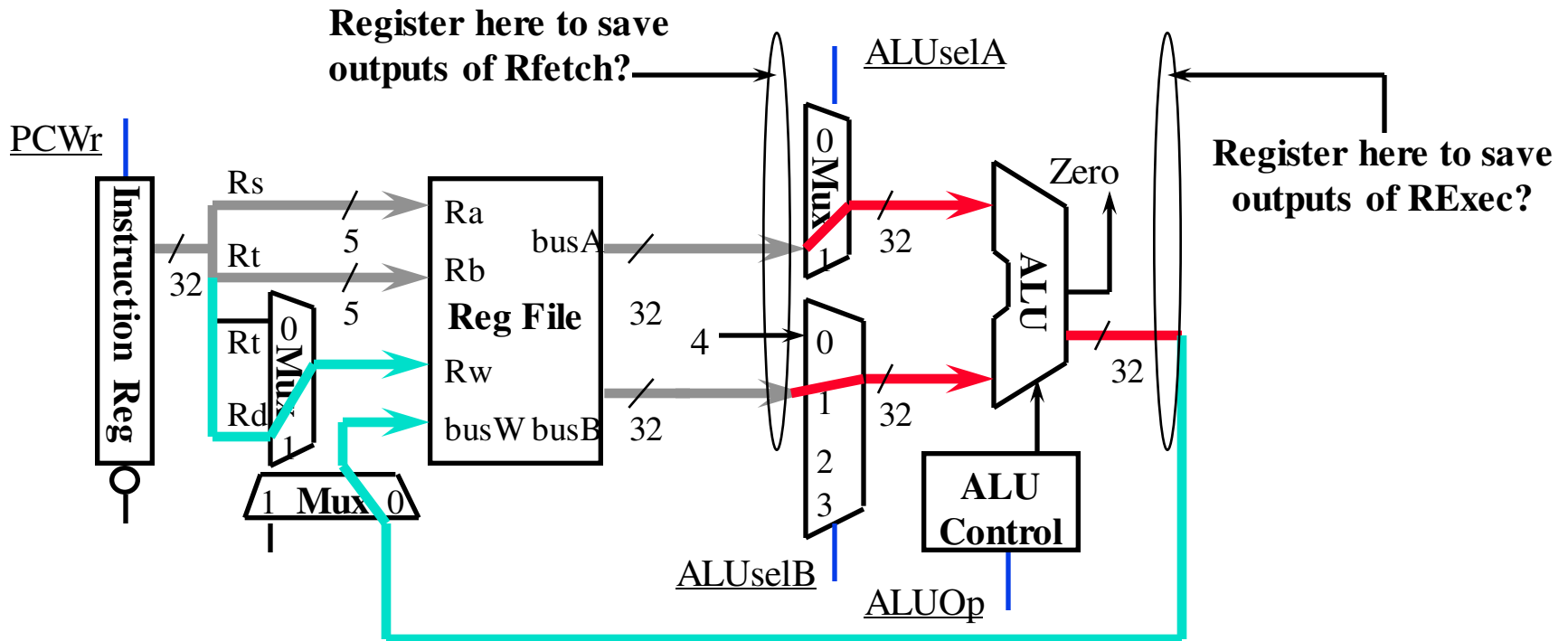
ALUOp=Rtype
1: RegDst, RegWr
ALUSelA
ALUSelB=01
x: IorD, PCSrc
ExtOp



A Multiple Cycle Delay Path

◦ There is no register to save the results between:

- Register Fetch: $\text{busA} \leftarrow \text{Reg}[\text{rs}] ; \text{busB} \leftarrow \text{Reg}[\text{rt}]$ —
- R-type Execution: $\text{ALU output} \leftarrow \text{busA op busB}$ —
- R-type Completion: $\text{Reg}[\text{rd}] \leftarrow \text{ALU output}$ —

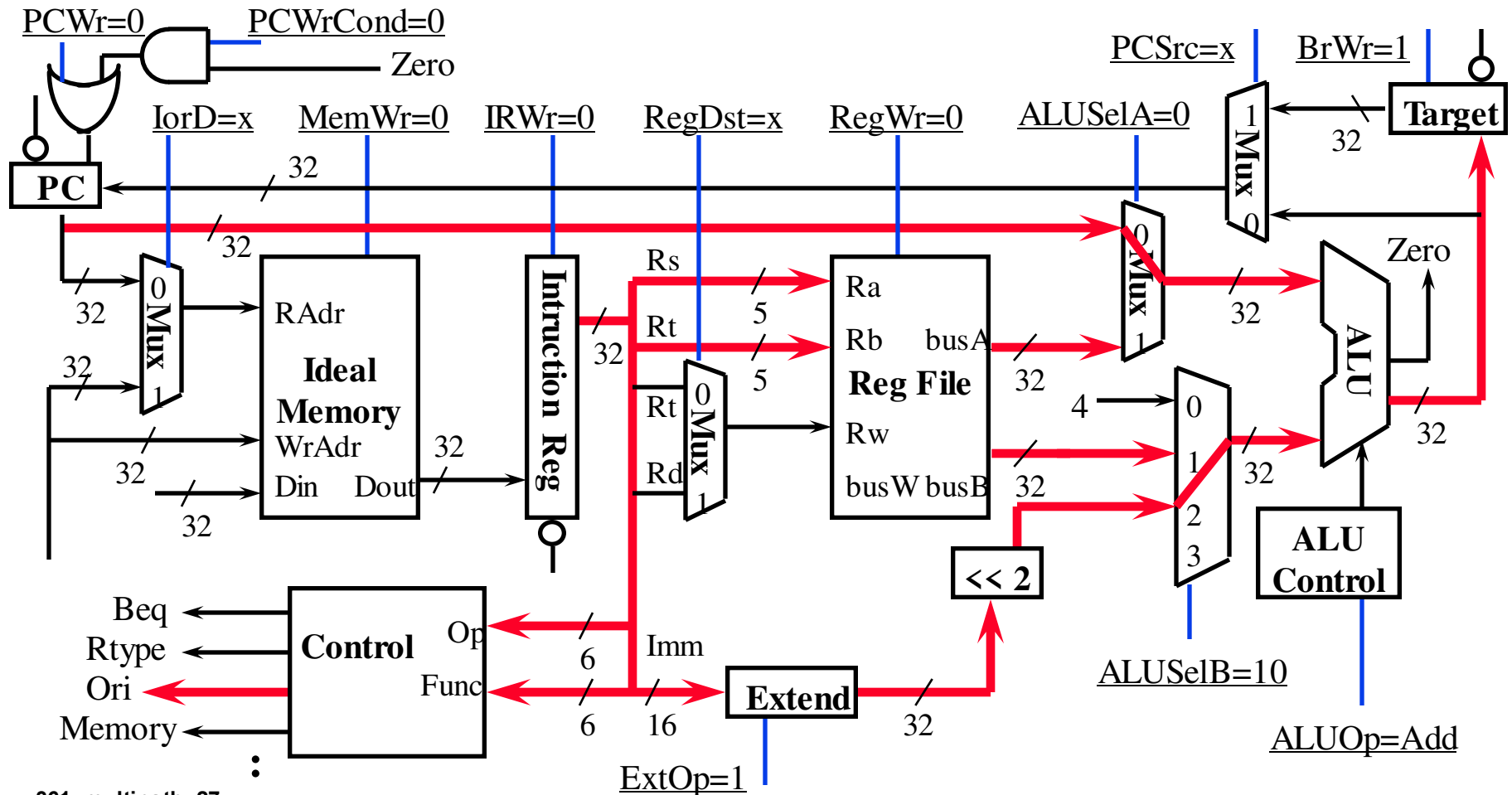


A Multiple Cycle Delay Path (Continue)

- Register is NOT needed to save the outputs of Register Fetch:
 - $IRWr = 0$: busA and busB will not change after Register Fetch
- Register is NOT needed to save the outputs of R-type Execution:
 - busA and busB will not change after Register Fetch
 - Control signals $ALUSelA$, $ALUSelB$, and $ALUOp$ will not change after R-type Execution
 - Consequently ALU output will not change after R-type Execution
- In theory (P. 316, P&H), you need a register to hold a signal value if:
 - The signal is computed in one clock cycle and used in another.

Instruction Decode: We have an Ori!

° Next Cycle: Ori Execution

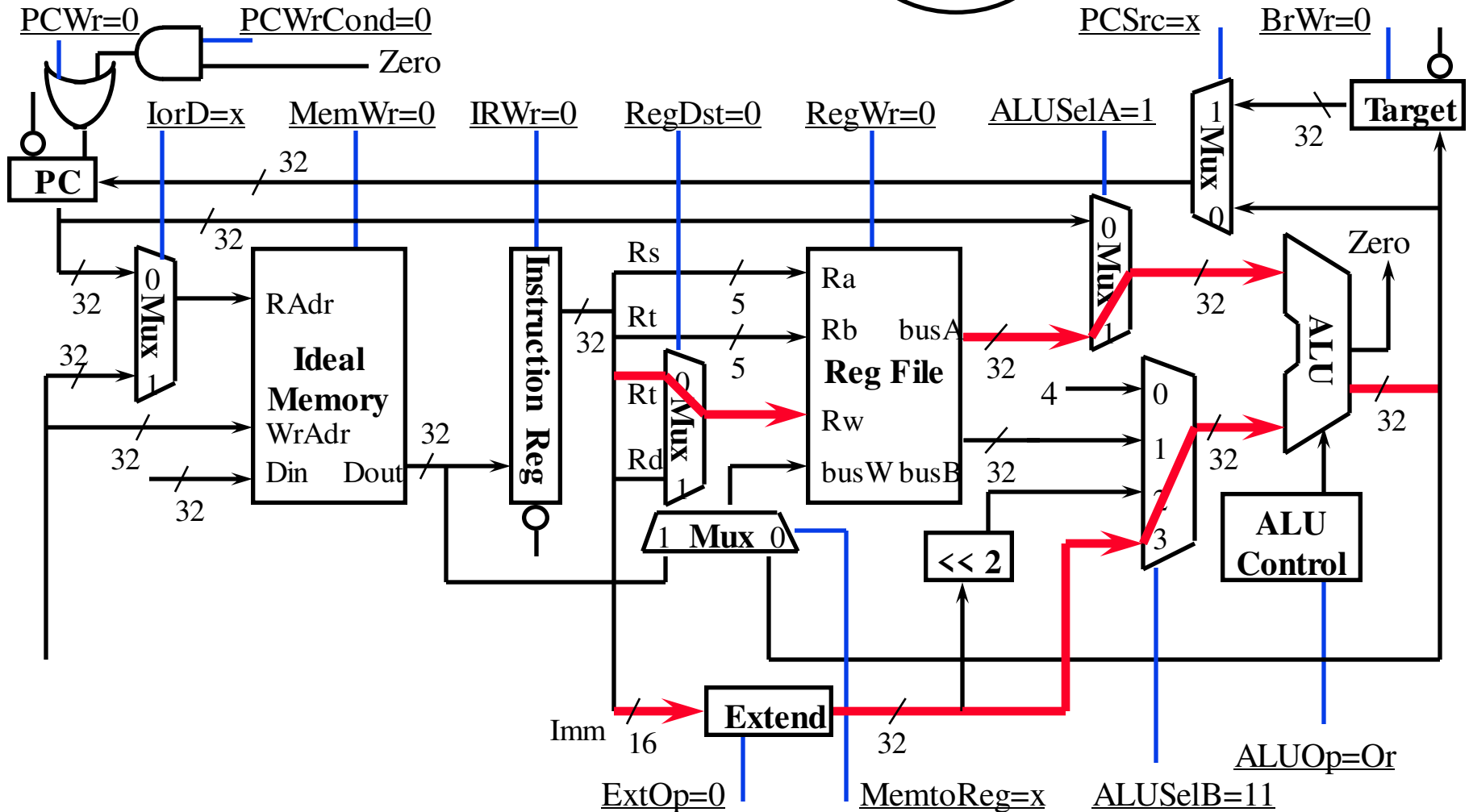


Ori Execution

- ° ALU output <- busA or ZeroExt[Imm16]

OriExec

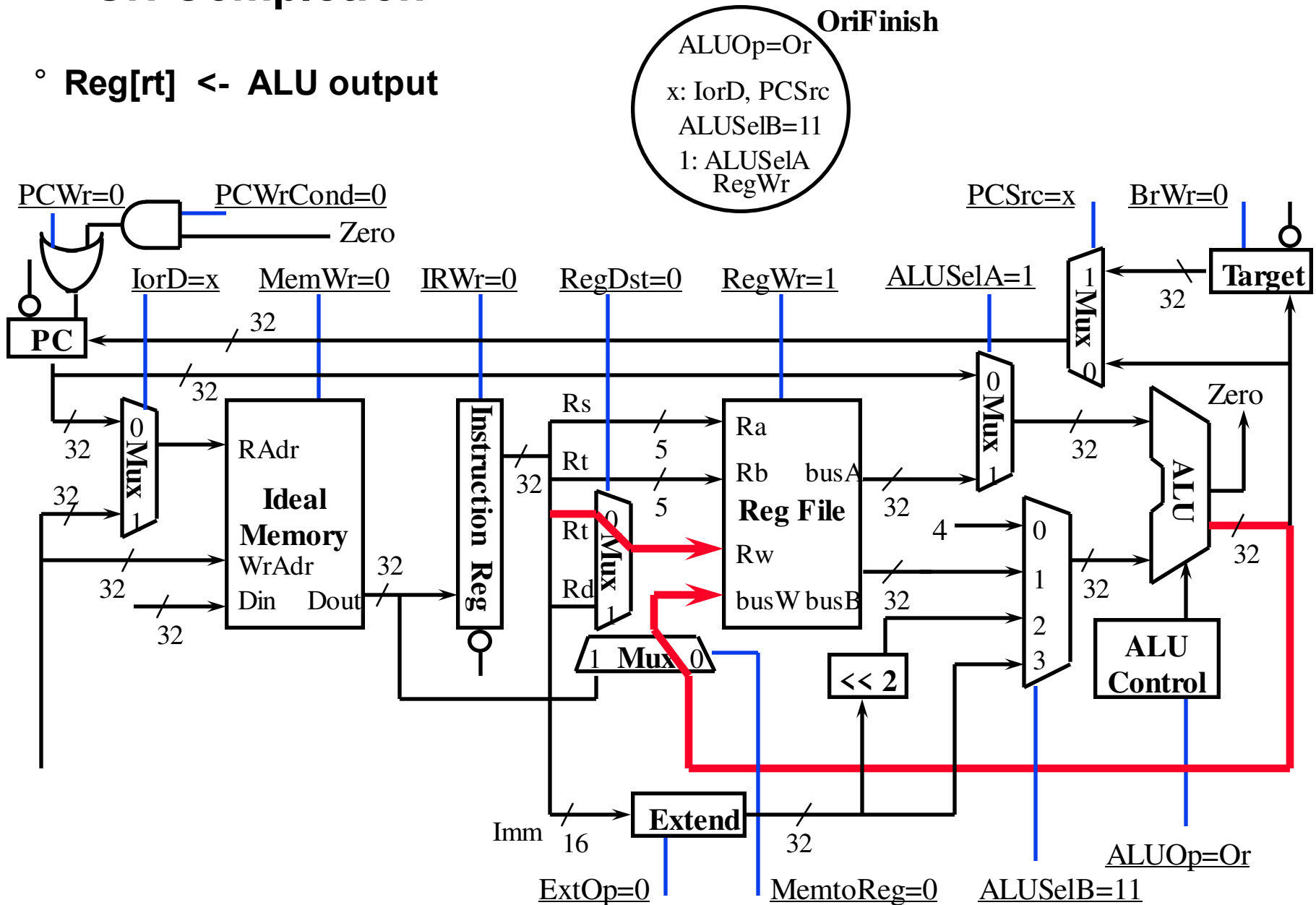
ALUOp=Or
1: ALUSelA
ALUSelB=11
x: MemtoReg
IorD, PCSrc



Ori Completion

° **Reg[rt] <- ALU output**

OriFinish

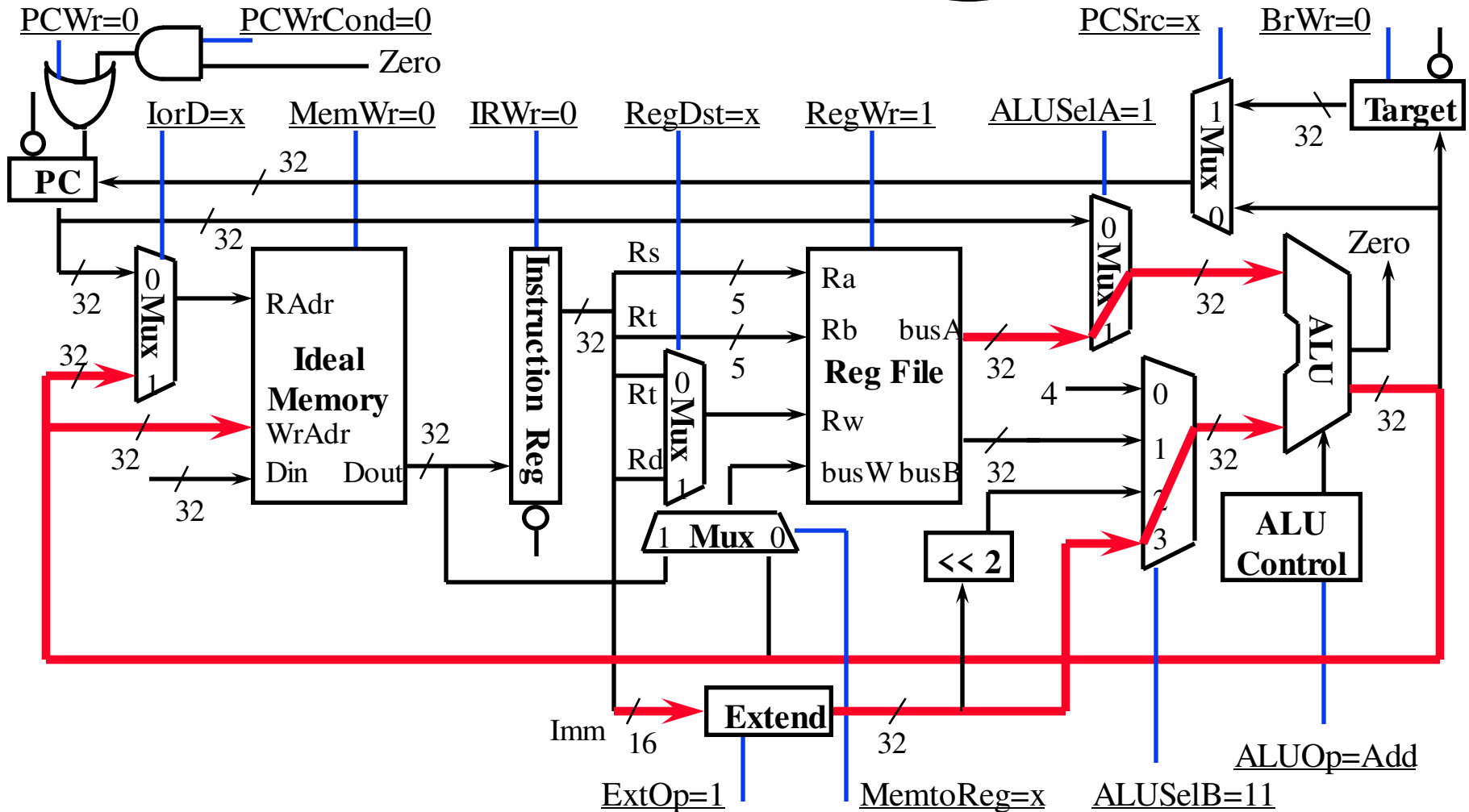


Memory Address Calculation

° ALU output <- busA + SignExt[Imm16]

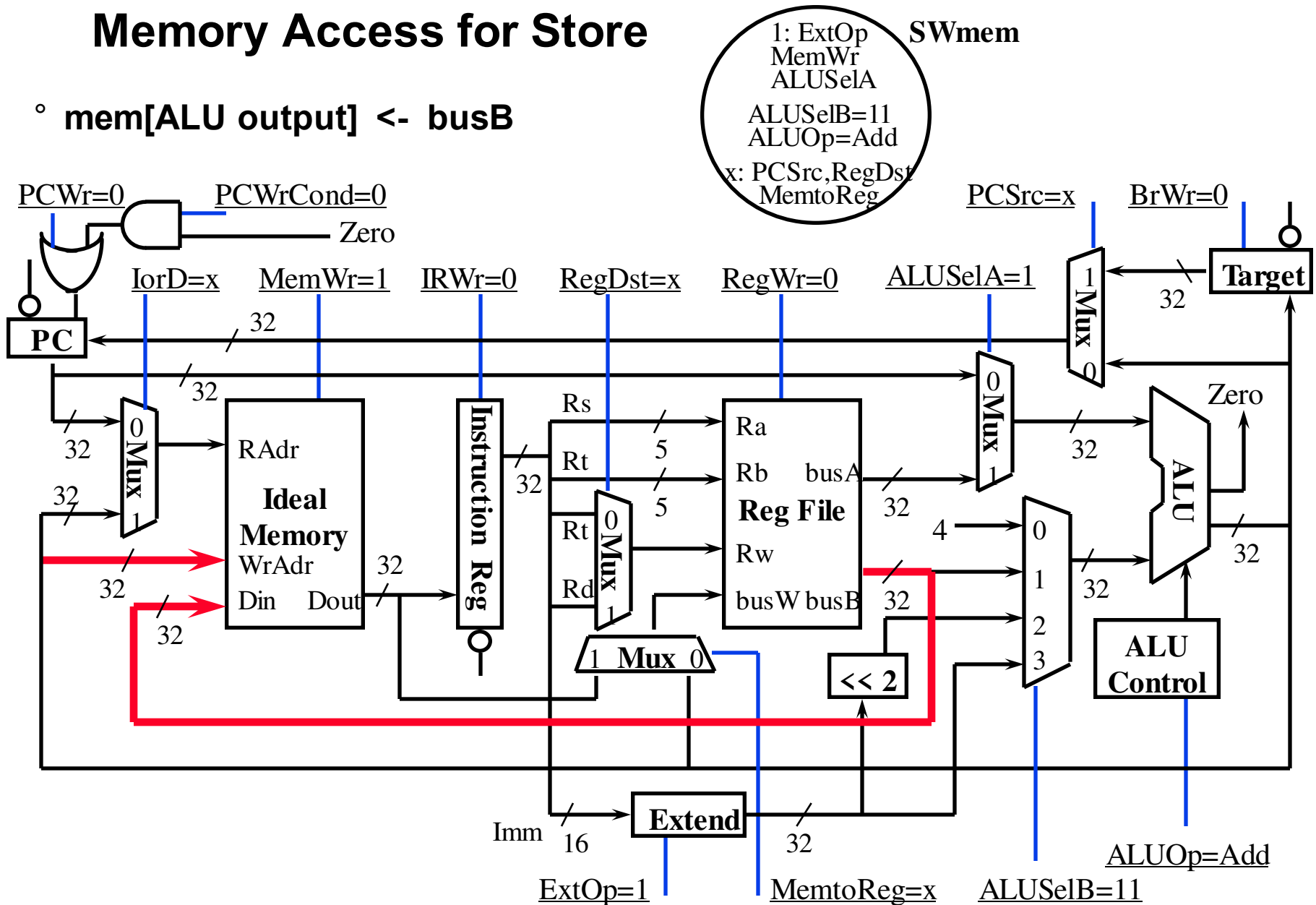
AdrCal

1: ExtOp
ALUSelA
ALUSelB=11
ALUOp=Add
x: MemtoReg
PCSrc



Memory Access for Store

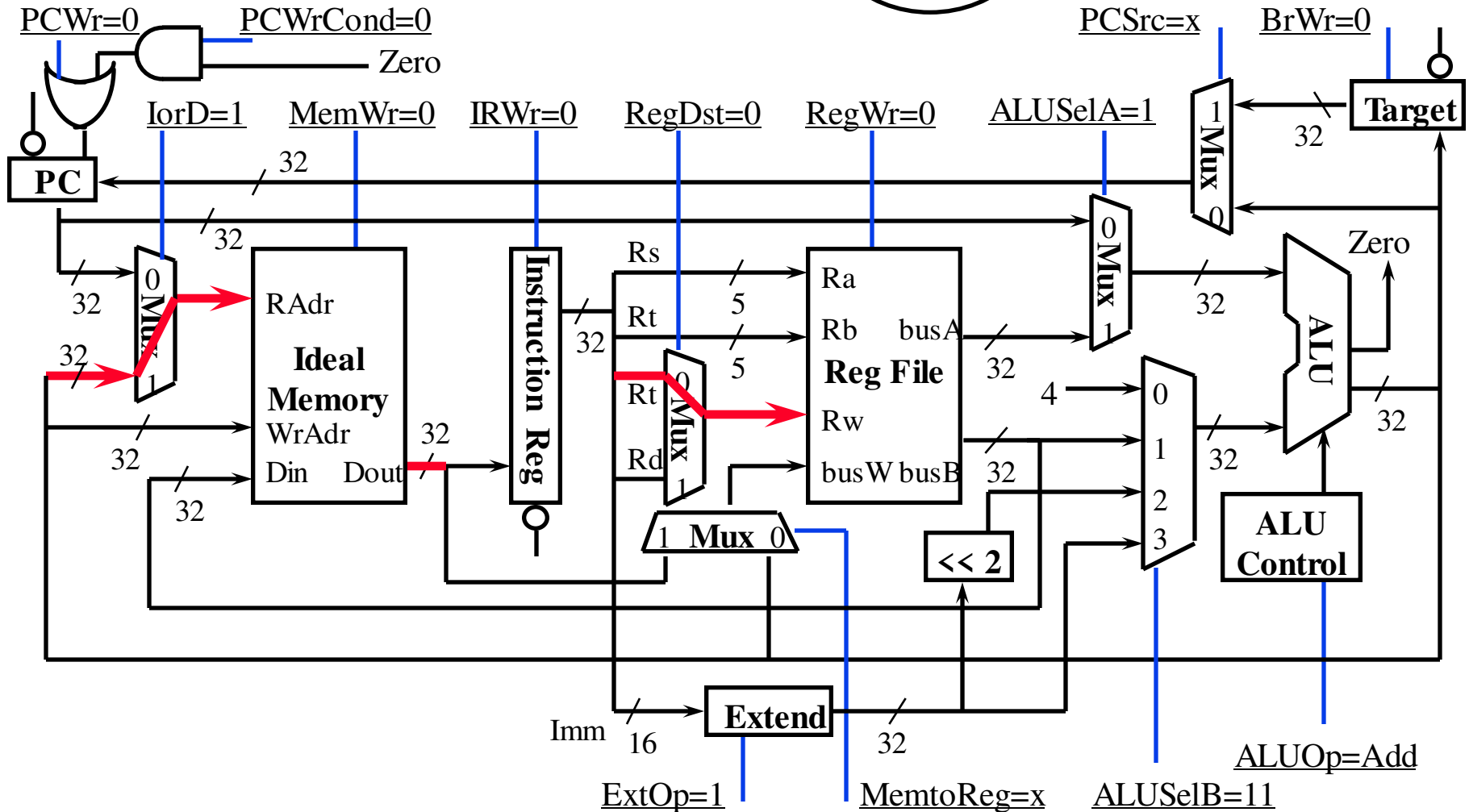
° $\text{mem}[\text{ALU output}] \leftarrow \text{busB}$



Memory Access for Load

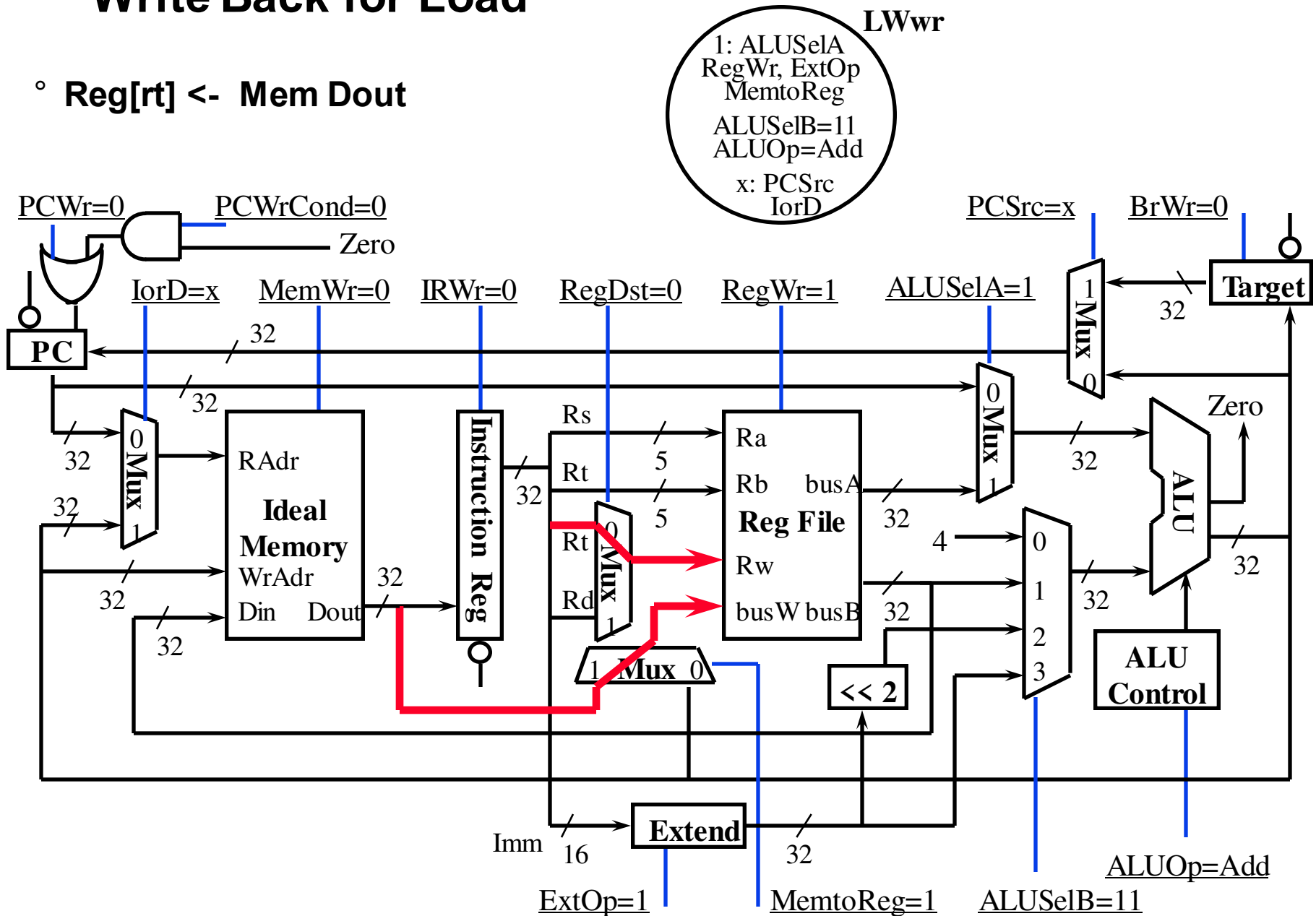
◦ **Mem Dout** <- mem[ALU output]

1: ExtOp
ALUSelA, IorD
ALUSelB=11
ALUOp=Add
x: MemtoReg
PCSrc

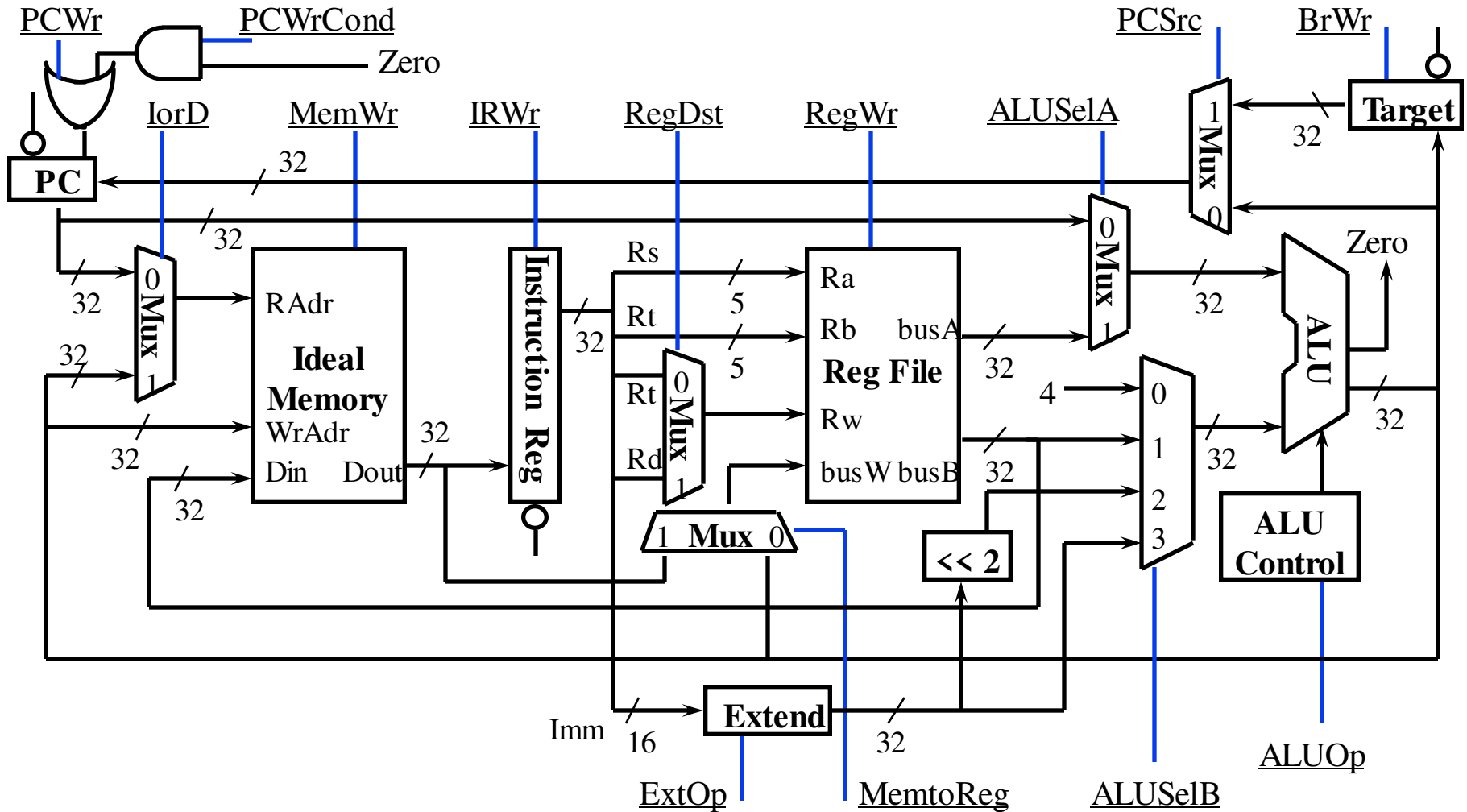


Write Back for Load

° $\text{Reg}[rt] \leftarrow \text{Mem Dout}$



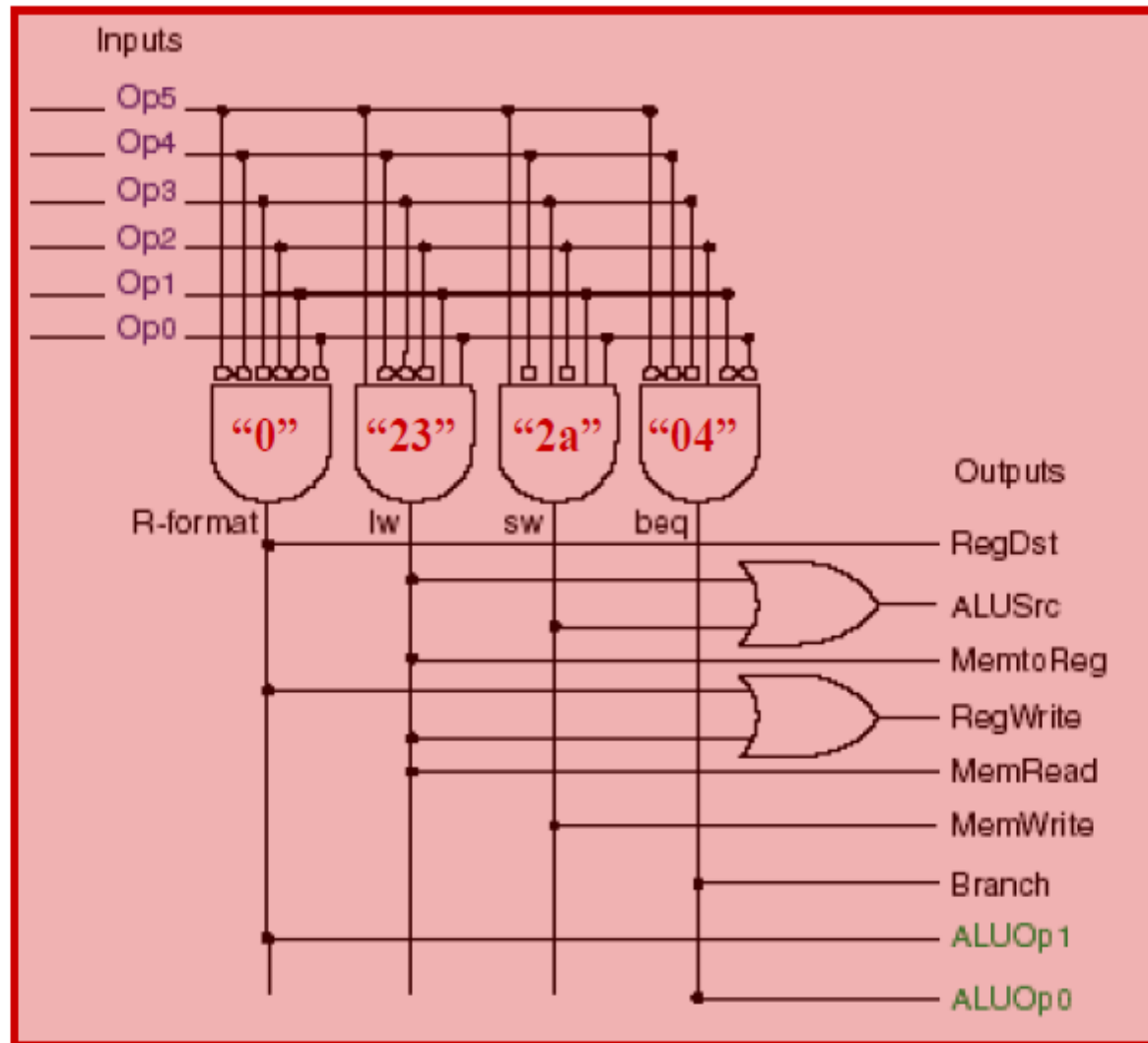
Putting it all together: Multiple Cycle Datapath



Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Op Code Control Block Circuitry

Instruction
bits 26-31



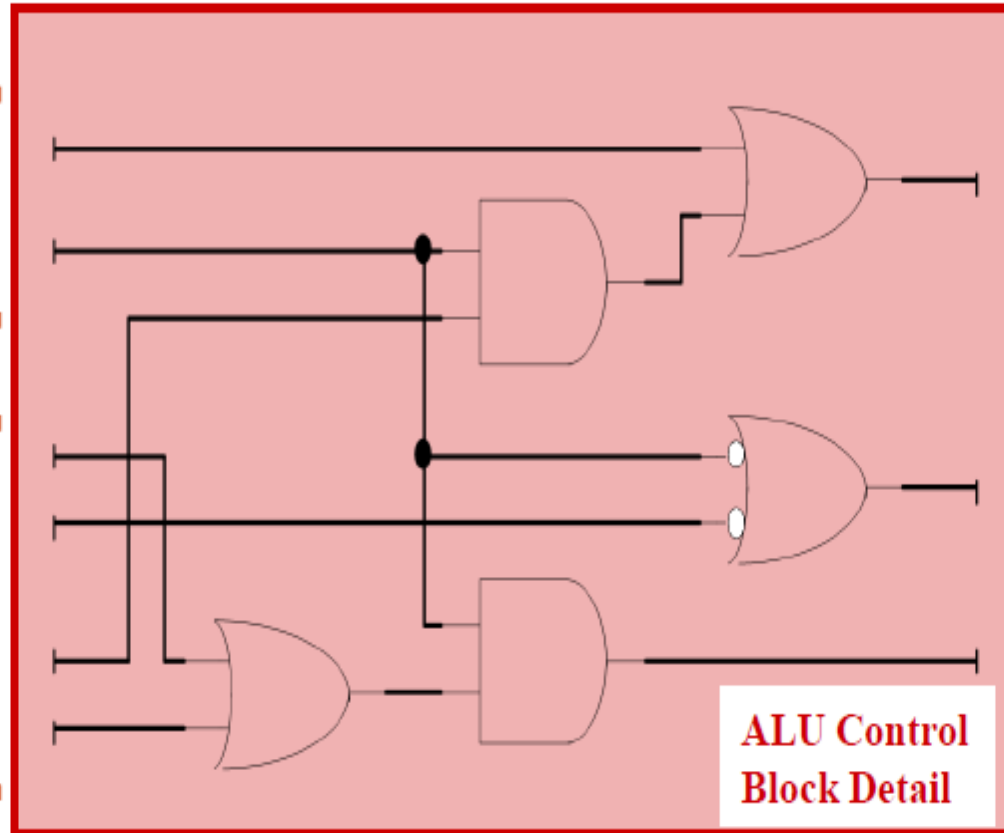
Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

ALU Control Block

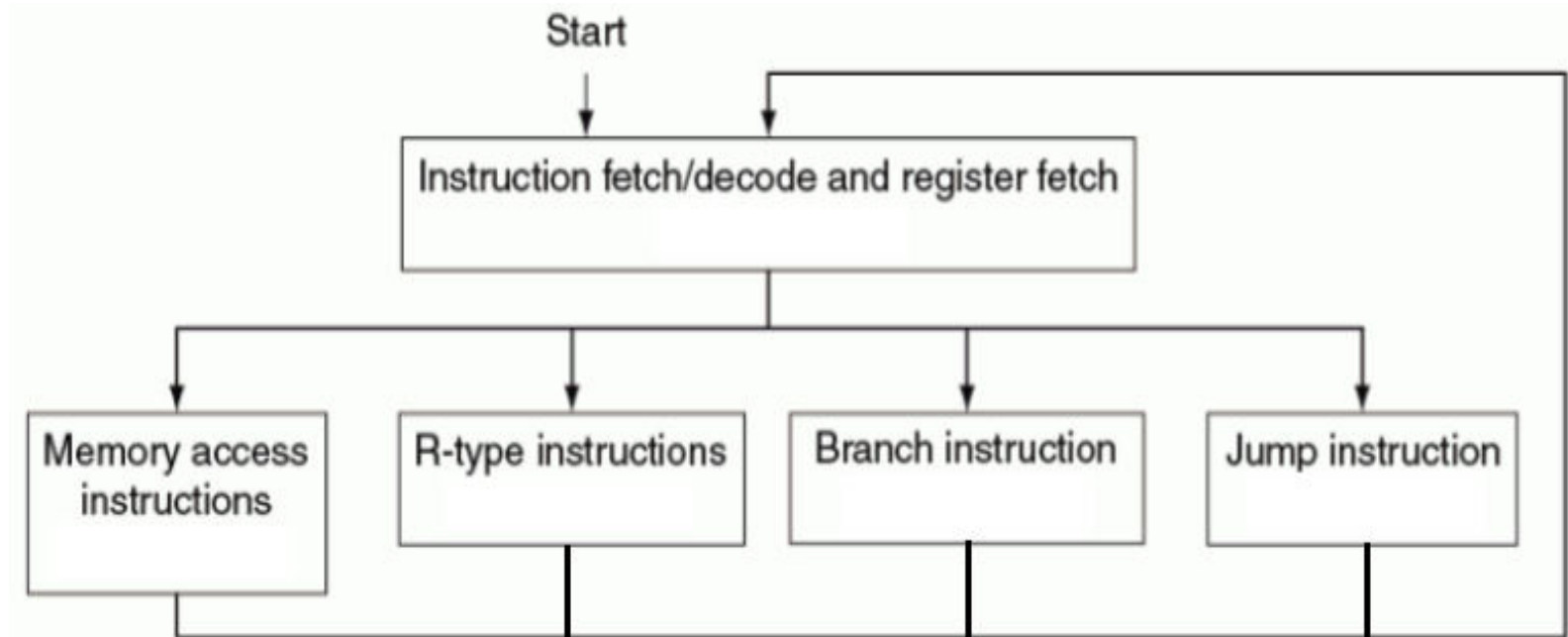
ALU operation
bits from op code
control unit

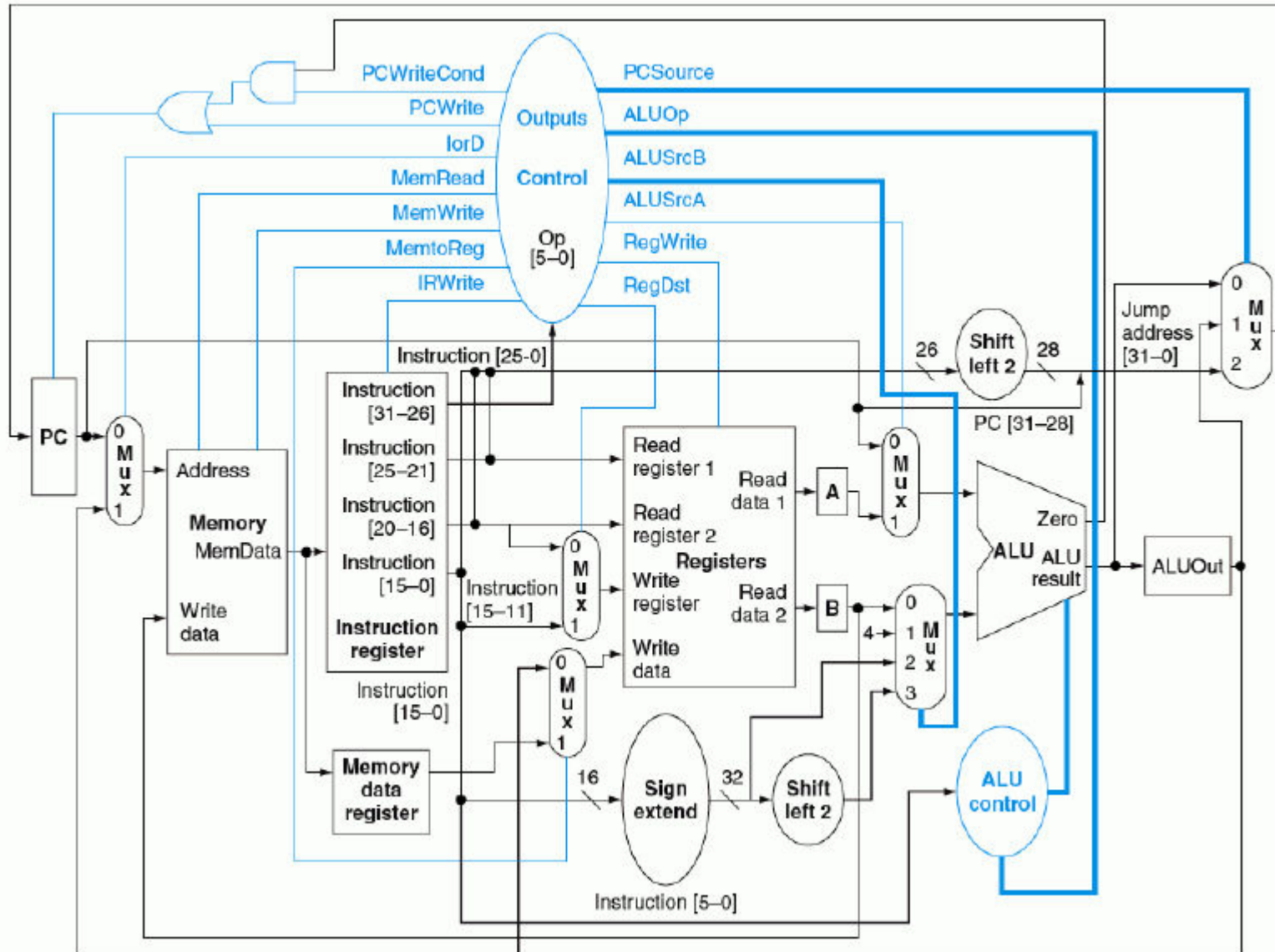
Function code
(instruction
bits 0-5)



3-bit ALU
control bus

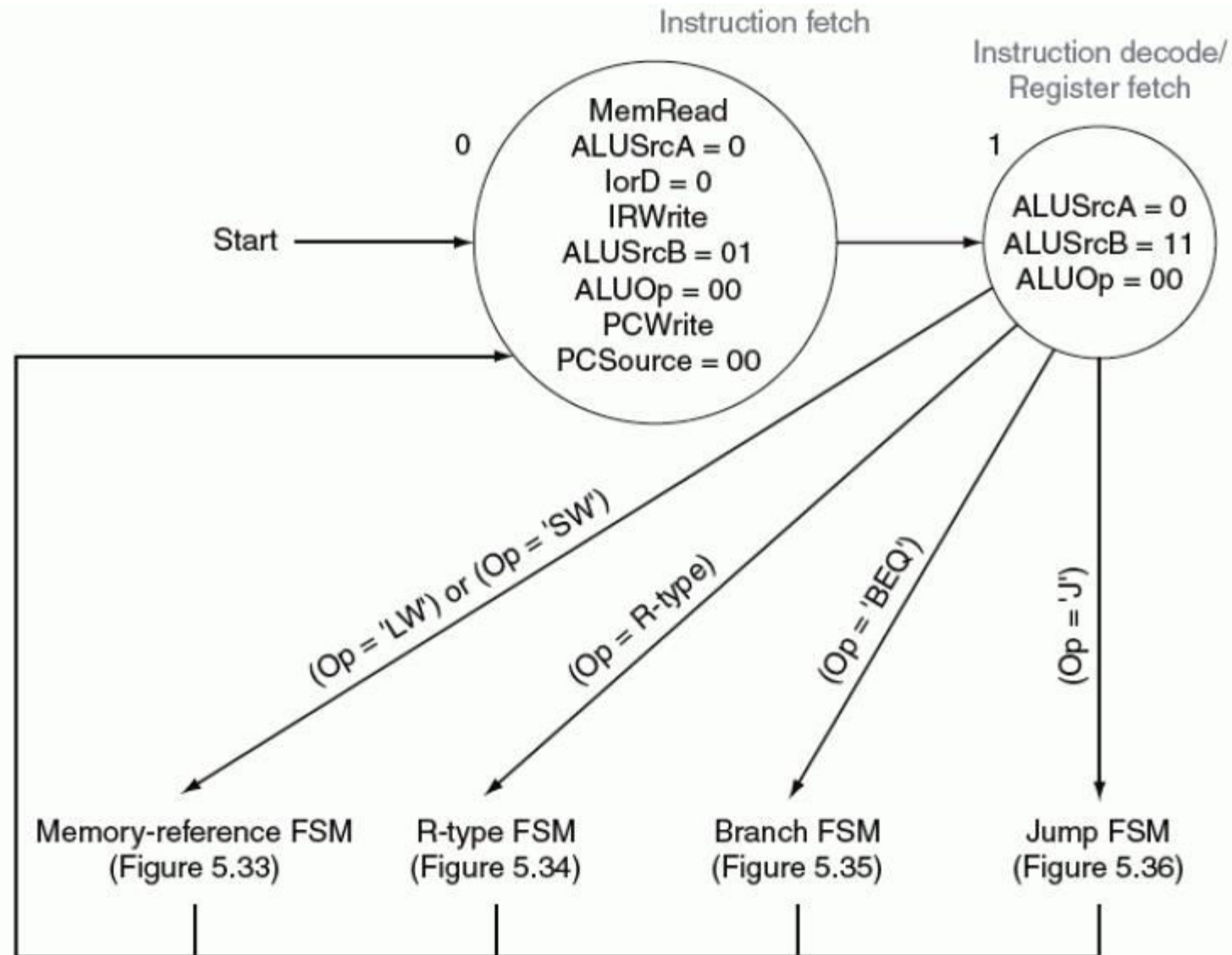
The high-level view of the finite state machine control



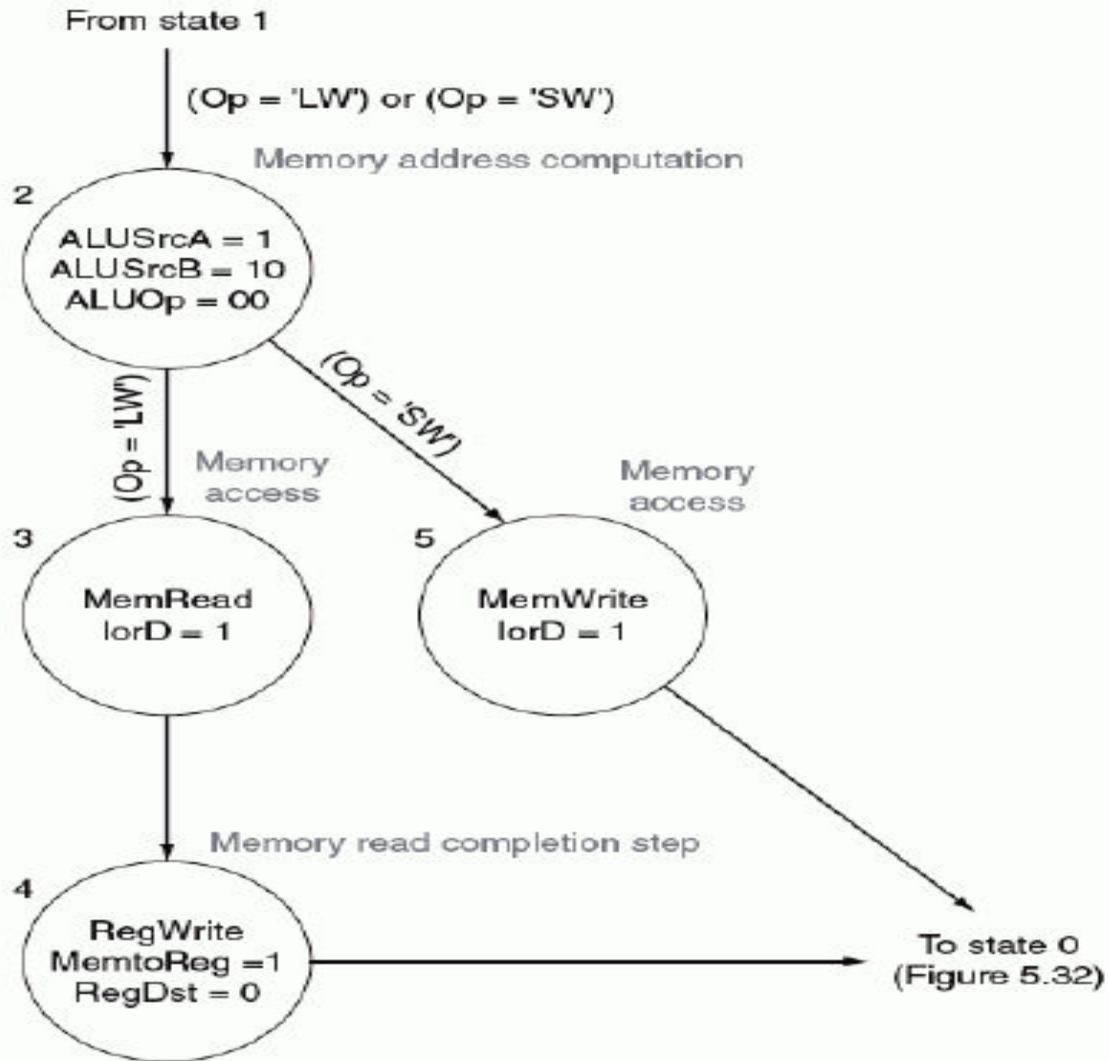


Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	$IR \leftarrow \text{Memory}[PC]$ $PC \leftarrow PC + 4$			
Instruction decode/register fetch	$A \leftarrow \text{Reg}[IR[25:21]]$ $B \leftarrow \text{Reg}[IR[20:16]]$ $ALUOut \leftarrow PC + (\text{sign-extend}(IR[15:0]) \ll 2)$			
Execution, address computation, branch/jump completion	$ALUOut \leftarrow A \text{ op } B$	$ALUOut \leftarrow A + \text{sign-extend}(IR[15:0])$	if $(A == B)$ $PC \leftarrow ALUOut$	$PC \leftarrow \{PC[31:28], (IR[25:0], 2'b00)\}$
Memory access or R-type completion	$\text{Reg}[IR[15:11]] \leftarrow ALUOut$	Load: $MDR \leftarrow \text{Memory}[ALUOut]$ or Store: $\text{Memory}[ALUOut] \leftarrow B$		
Memory read completion		Load: $\text{Reg}[IR[20:16]] \leftarrow MDR$		

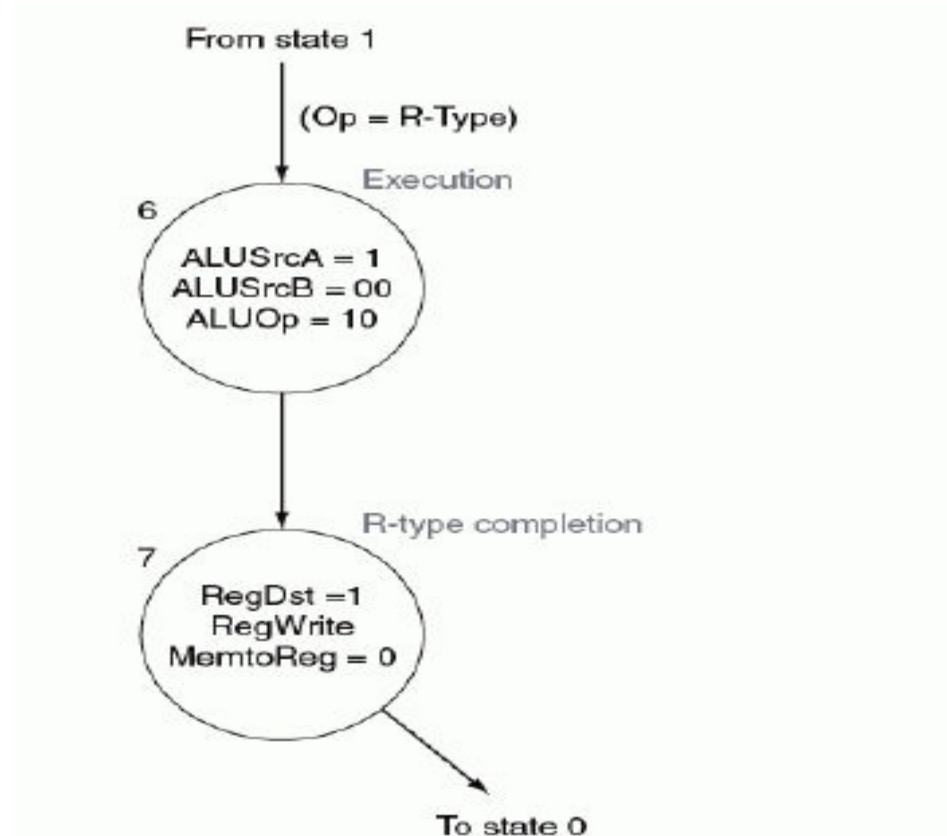
The instruction fetch and decode portion of every instruction



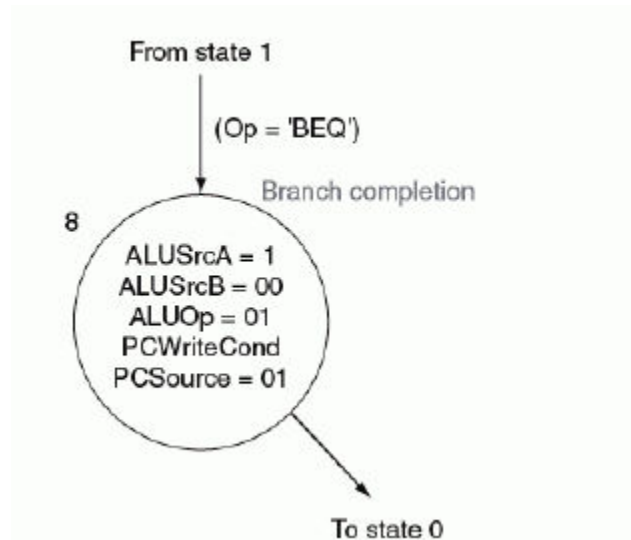
The finite state machine for controlling memory-reference instructions



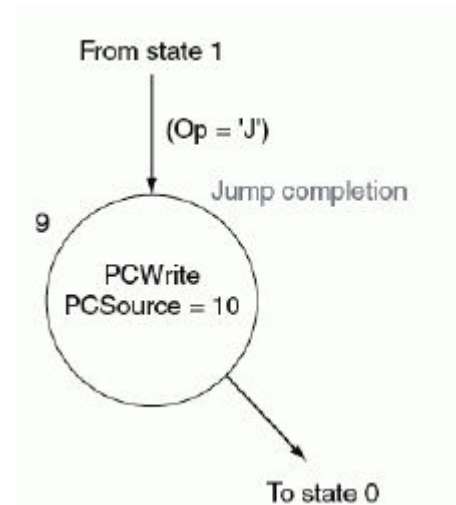
R-type instructions c



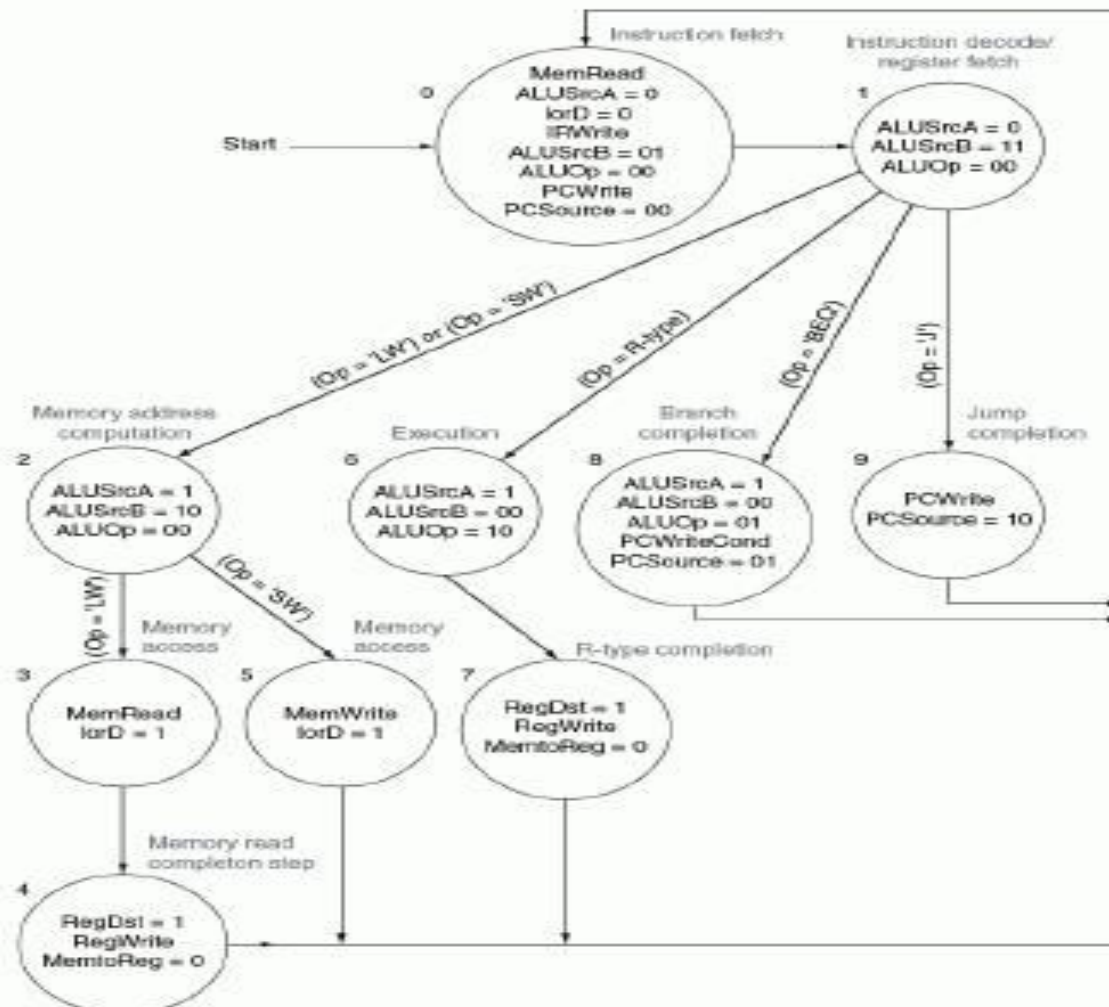
The branch instruction

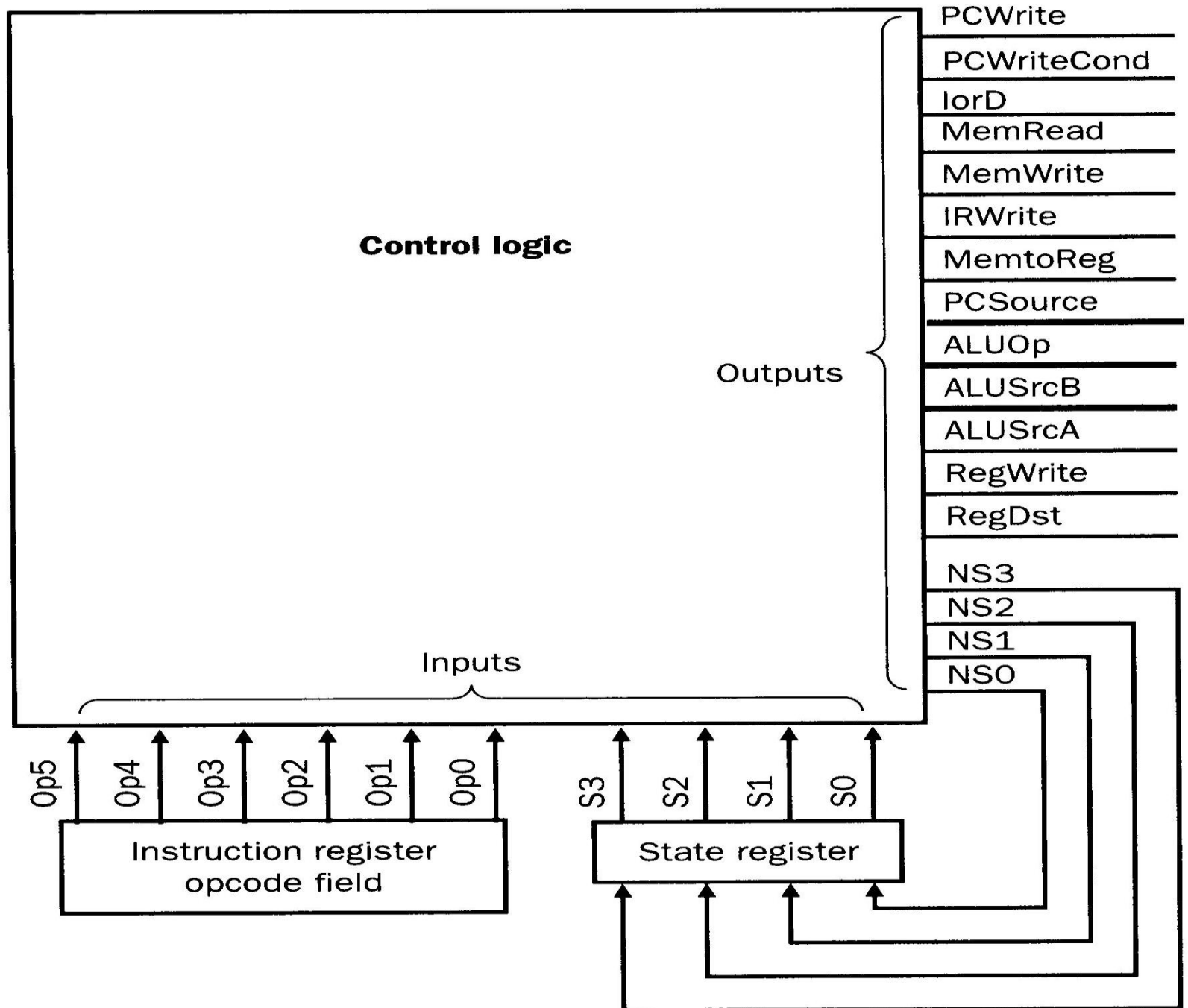


The jump instruction



The complete finite state machine control





Summary

- **Disadvantages of the Single Cycle Processor**
 - Long cycle time
 - Cycle time is too long for all instructions except the Load
- **Multiple Cycle Processor:**
 - Divide the instructions into smaller steps
 - Execute each step (instead of the entire instruction) in one cycle
- **Do NOT confuse Multiple Cycle Processor with Multiple Cycle Delay Path**
 - Multiple Cycle Processor executes each instruction in multiple clock cycles
 - Multiple Cycle Delay Path: a combinational logic path between two storage elements that takes more than one clock cycle to complete
- **It is possible (desirable) to build a MC Processor without MCDP:**
 - Use a register to save a signal's value whenever a signal is generated in one clock cycle and used in another cycle later