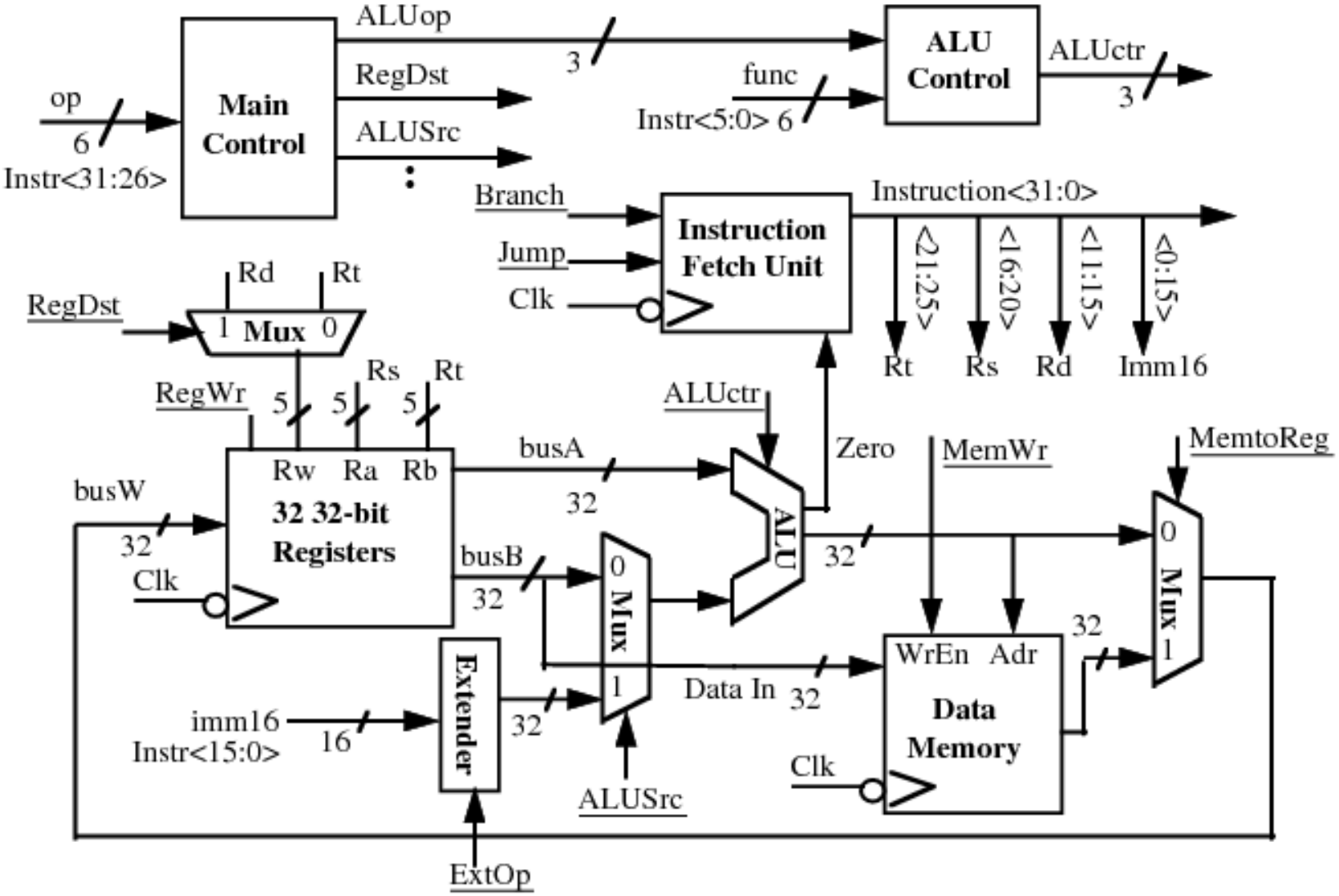
Acknowledgment: Almost all of these slides are based on Dave Patterson's CS152 Lecture Slides at UC, Berkeley.

COMPUTER SYSTEMS ORGANIZATION

Multi Cycle CPU Design -- Spring 2010 -- IIIT-H -- Suresh Purini

A Single Cycle Processor



Drawbacks of Single Cycle Processor

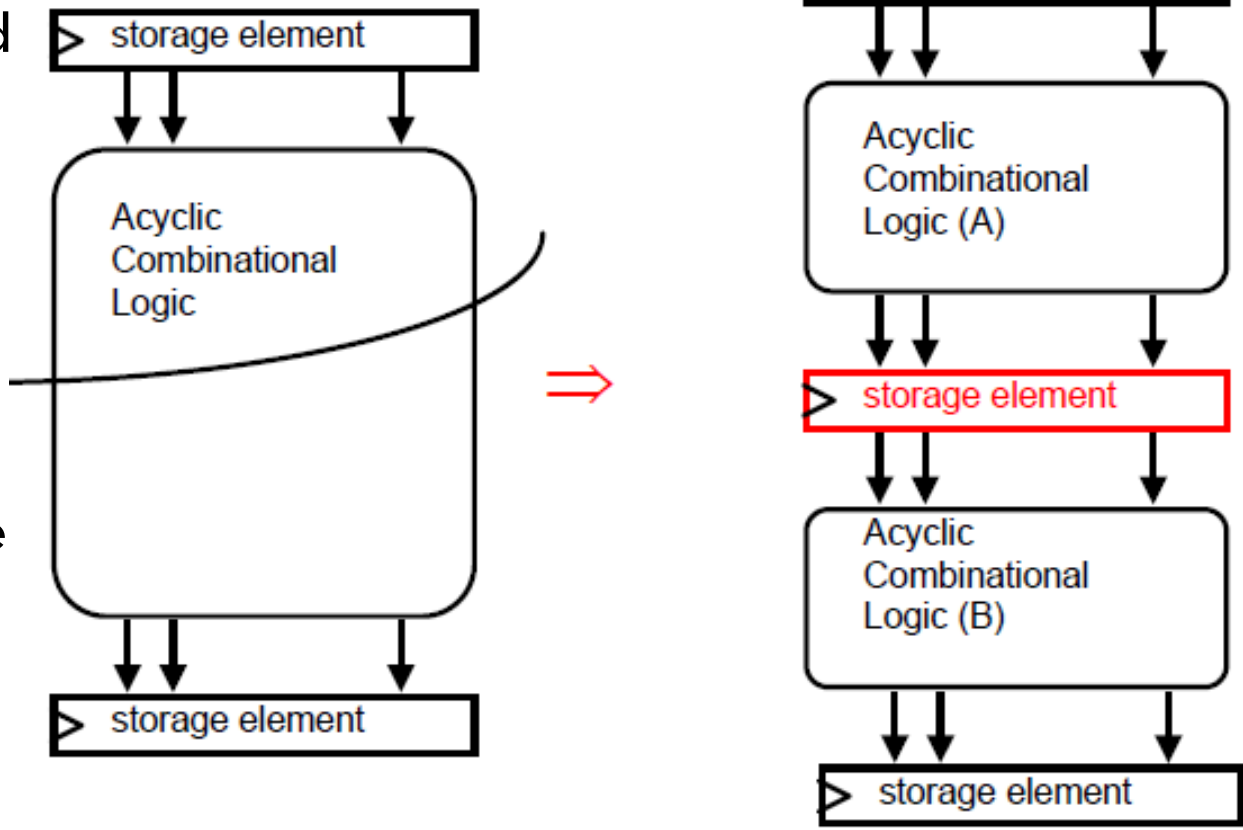
- Long cycle time: Cycle time must be long enough for the load instruction:
  - PC's Clock -to-Q +
  - Instruction Memory Access Time +
  - Register File Access Time +
  - ALU Delay (address calculation) +
  - Data Memory Access Time +
  - Register File Setup Time
- Cycle time is much longer than needed for all other instructions.
  - 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

Reducing Cycle Time

- Cut combinational dependency graph and insert register / latch
- Do same work in two fast cycles, rather than one slow one
- May be able to short-circuit path and remove some components for some instructions!



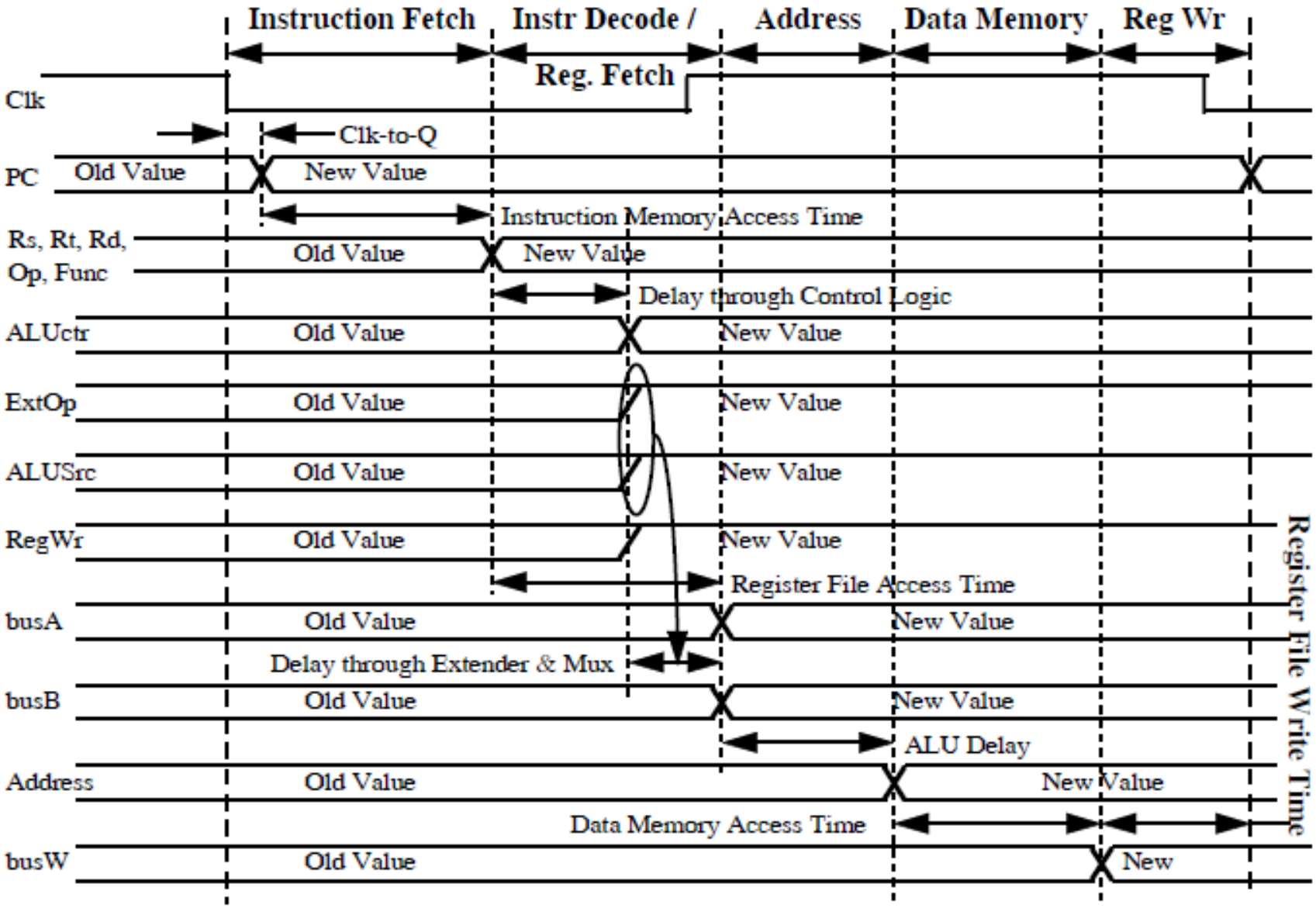
Advantages of Multiple Cycle Processor

- 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

The Big Picture: Performance Perspective

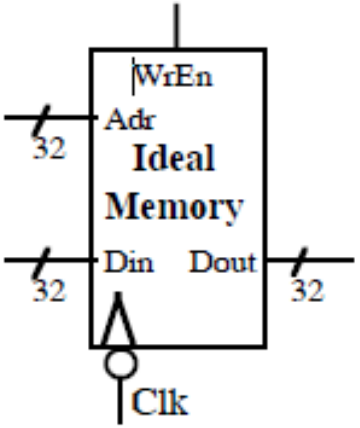
- Performance of a machine was determined by
  - Instruction Count
  - Clock cycle Time
  - Clock cycles per instruction
- Processor Design (data path and control) will determine
  - Clock cycle time
  - Clock cycles per instruction

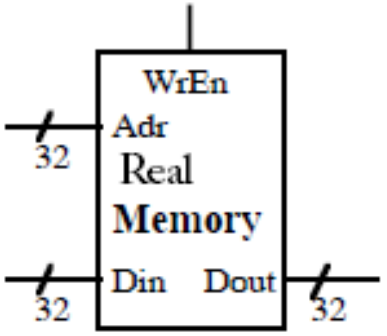
The Five Steps of a Load Instruction



Memory Write Timing: Ideal Vs Reality

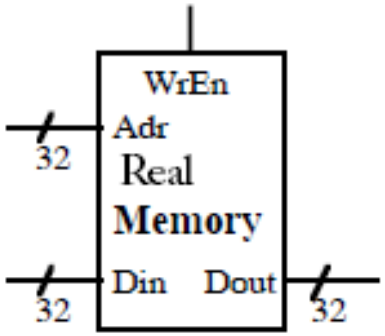
- In the Single Cycle Processor memory module is simplified
  - Write happens at the clock tick
  - Address, data, and write enable must be stable one "set-up" time before the clock tick
- In real life memory module has no clock input
  - The write path is a combinational logic delay path:
  - Write enable goes to 1 and Din settles down
  - Memory write access delay
  - Din is written into mem[address]
- Important: Address and Data must be stable BEFORE Write Enable goes to 1





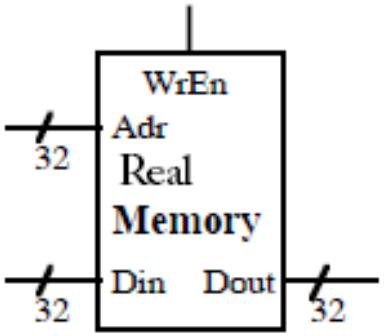
Race Condition Between Address and Write Enable

- The “real” (no clock input) memory may not work reliably in the single cycle processor because:
  - We cannot guarantee Address will be stable BEFORE WrEn = 1
  - There is a race between Adr and WrEn



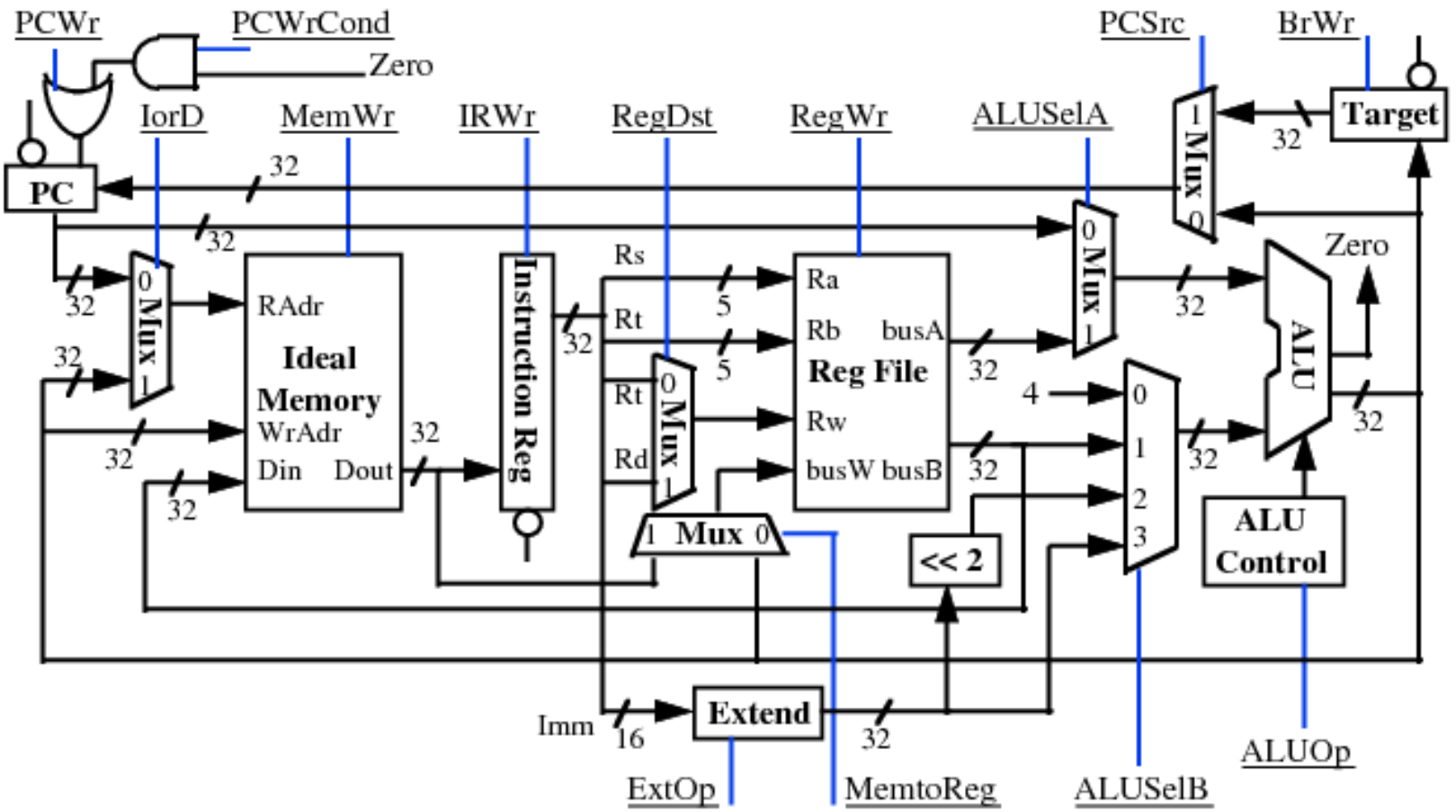
How to Avoid this Race Condition?

- Solution for the multiple cycle implementation:
  - Make sure Address is stable by the end of Cycle N
  - Assert Write Enable signal ONE cycle later at Cycle (N + 1)
  - Address cannot change until Write Enable is deasserted.

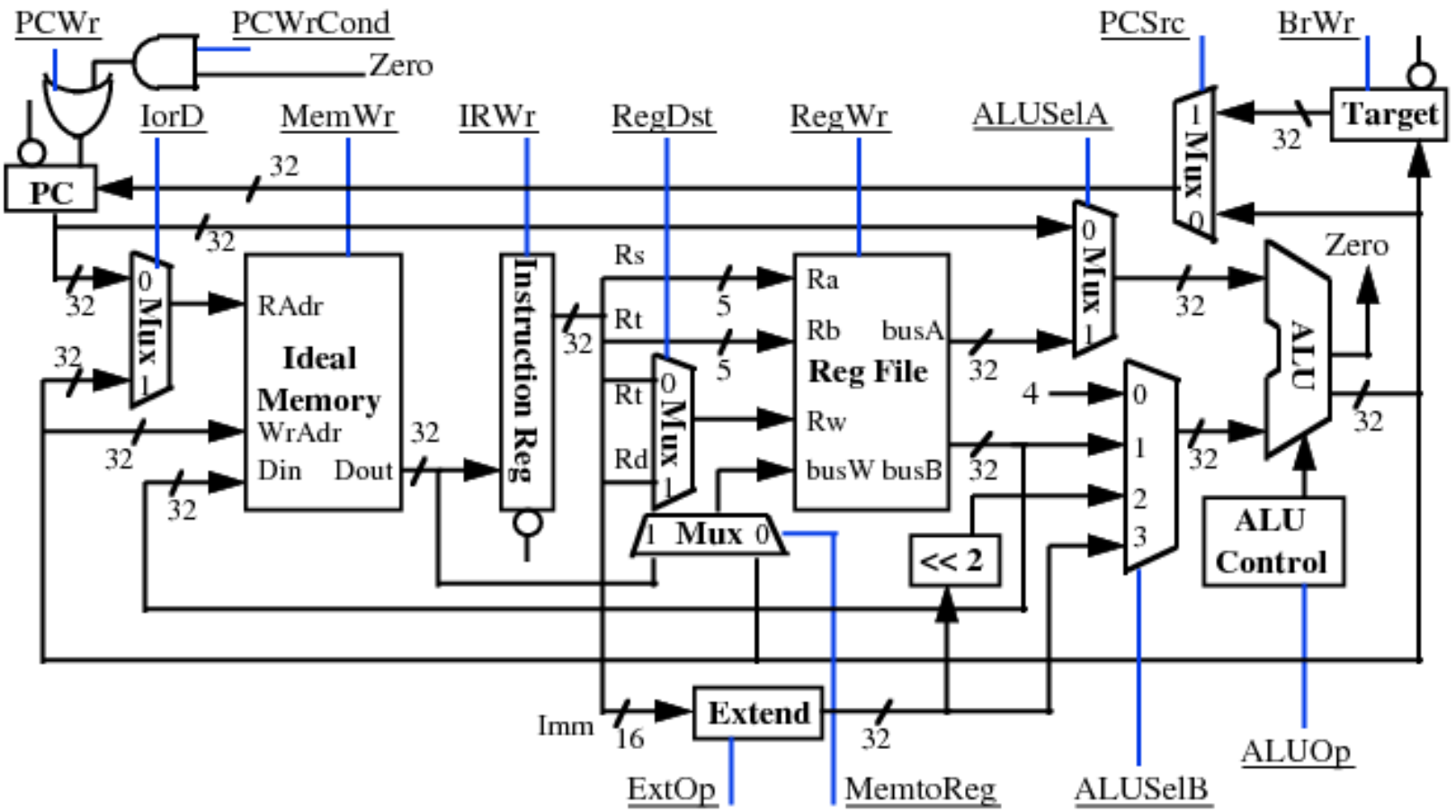


We can make use the same logic if we are using unclocked Register File to build our multiple cycle processor.

Multiple Cycle Data Path



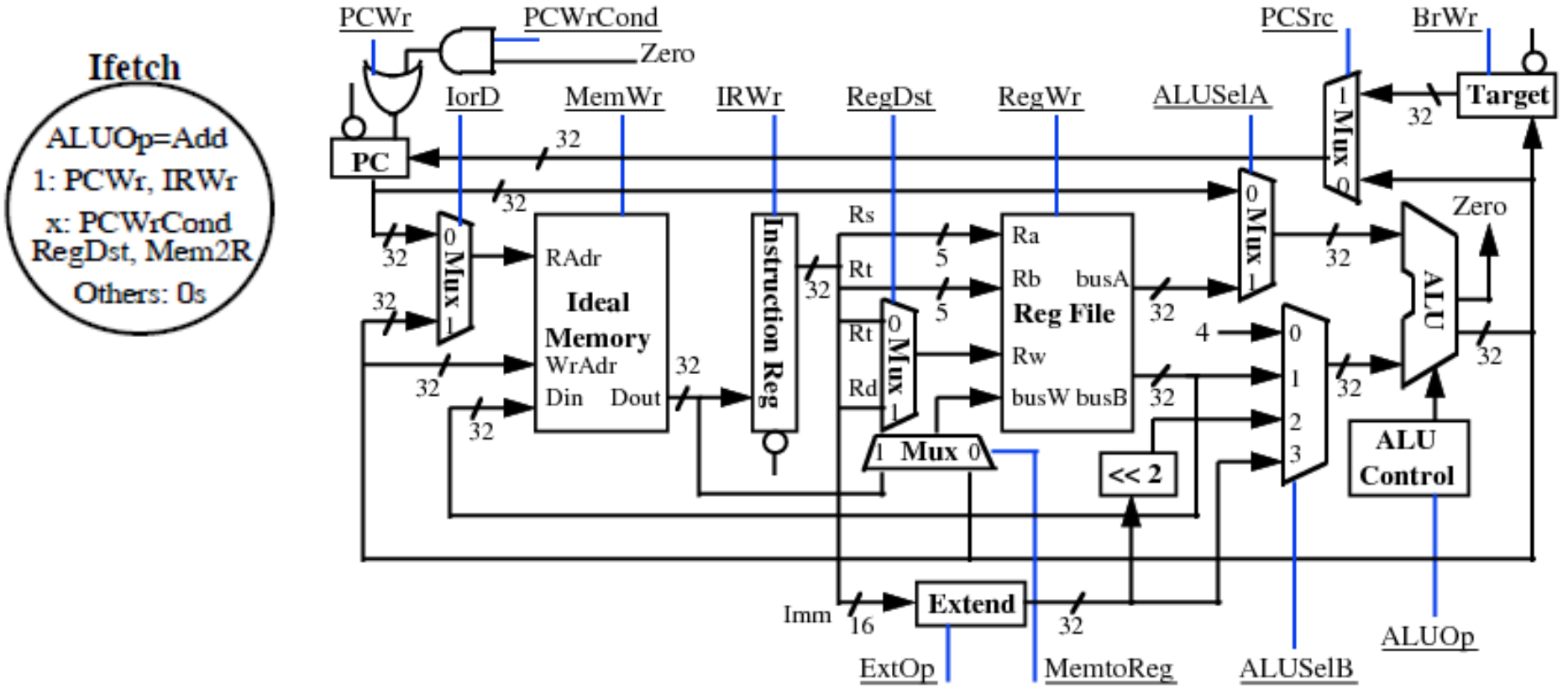
Multiple Cycle Data Path



Instruction Fetch Phase

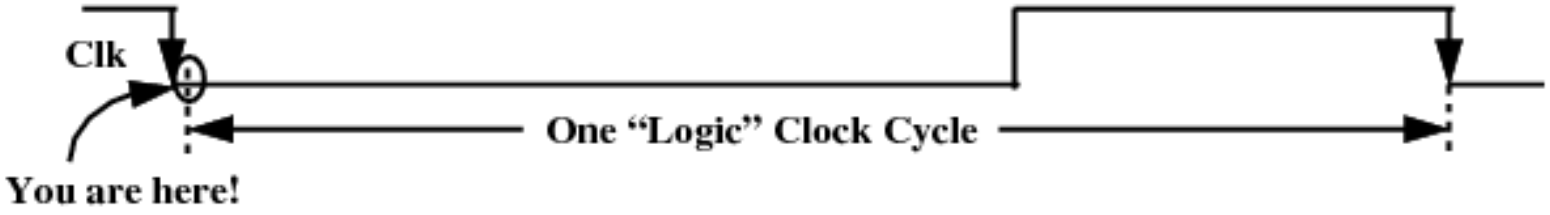
- IR = mem[PC]
- PC = PC + 4

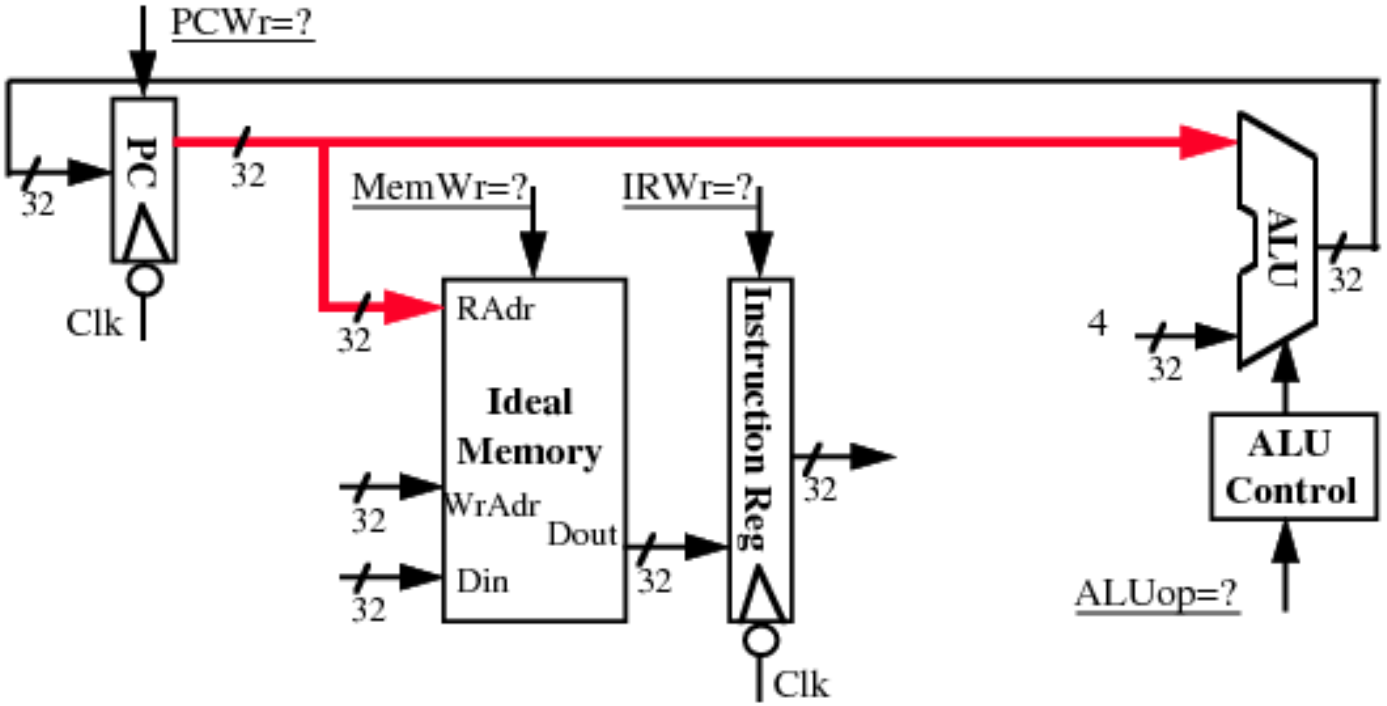
PCWr	IorD	MemWr	IrWr	RegDst	RegWr	ALUSelA
1	0	0	1	x	0	0
PCWrCond	**PCSrc**	**BrWr**	**ExtOp**	**MemtoReg**	**ALUOp**	**ALUSelB**
x	0	0	x	x	ADD	0



Instruction Fetch Cycle: In the Beginning

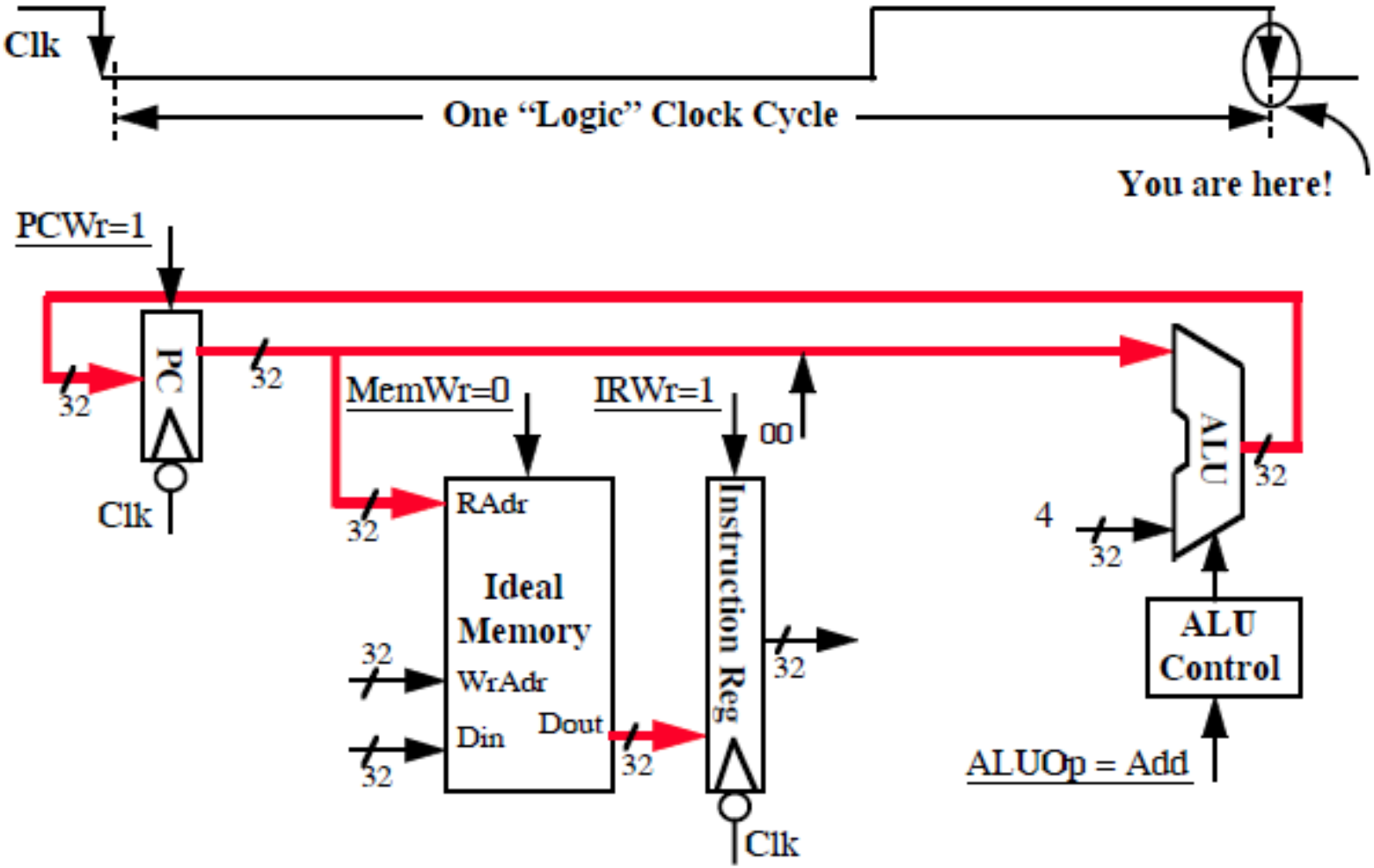
- Every cycle begins right AFTER the clock tick:
  - mem[PC] PC<31:0> + 4



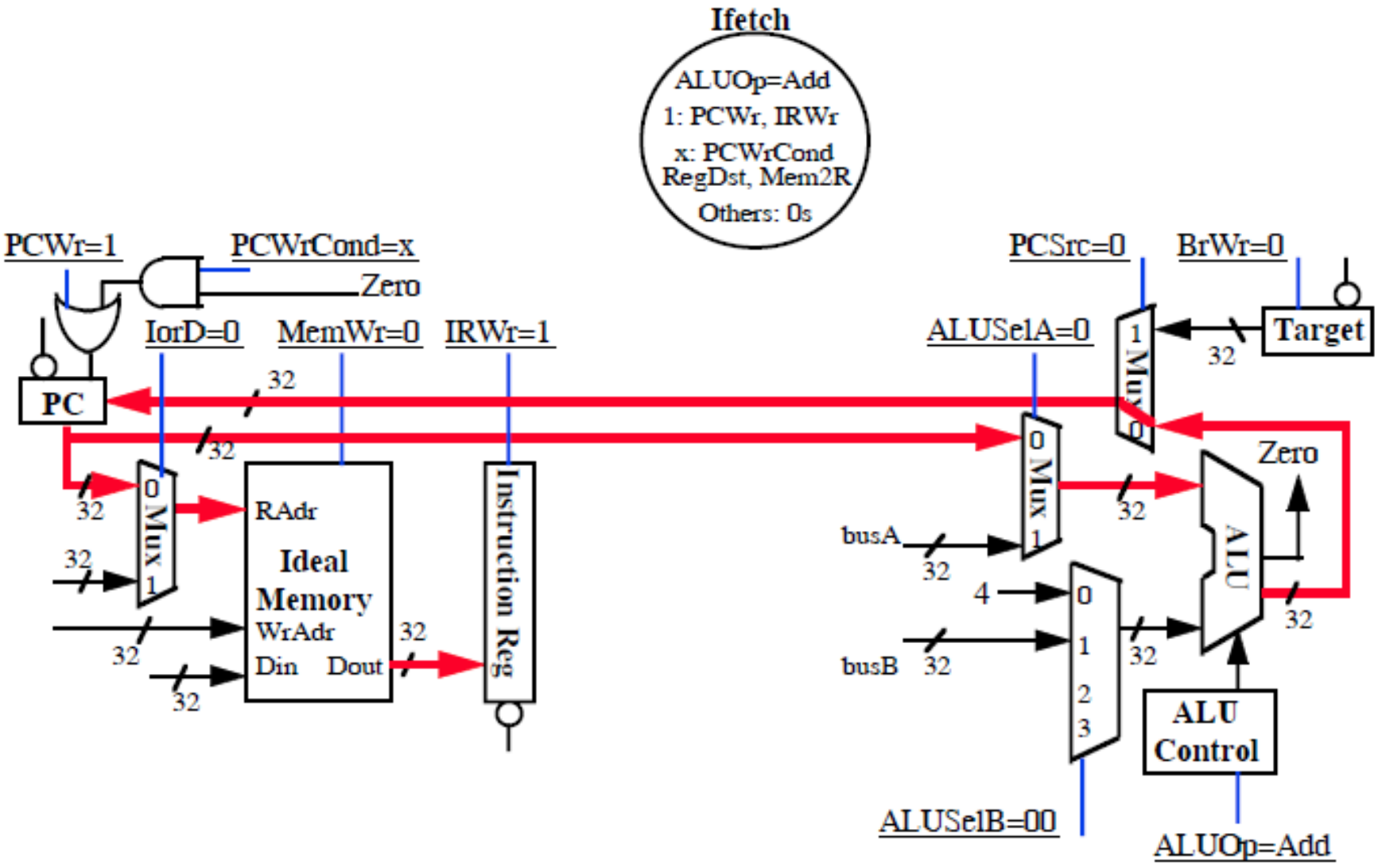


Instruction Fetch Cycle: The End

- Every cycle ends AT the next clock tick (storage element updates):
  - IR <-- mem[PC] PC<31:0> <-- PC<31:0> + 4



Instruction Fetch Cycle: Overall Picture



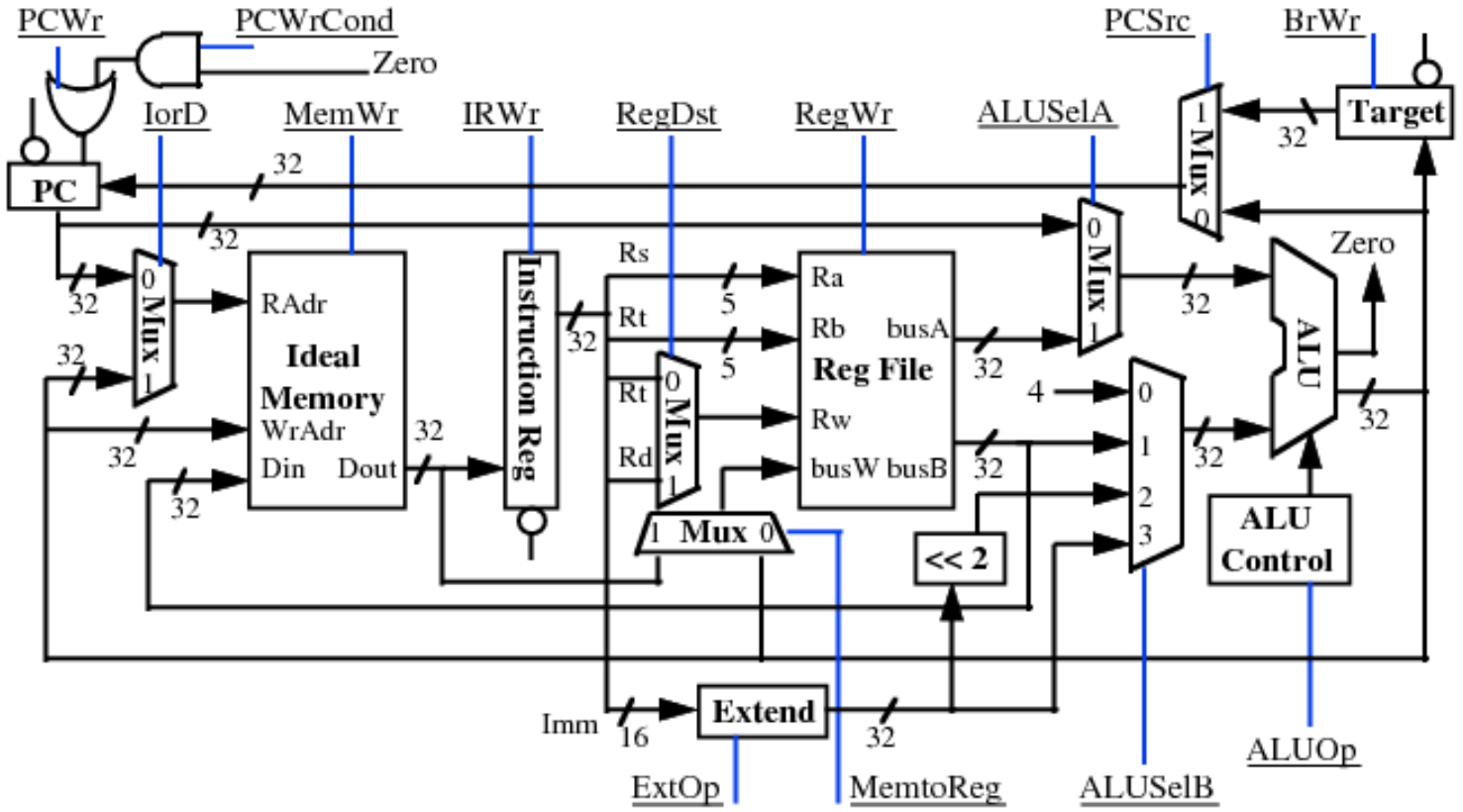
Register Fetch/Instruction Decode Phase

- IR = mem[PC]
- PC = PC + 4

PCWr	IorD	MemWr	IrWr	RegDst	RegWr	ALUSelA
1	0	0	1	x	0	0
PCWrCond	**PCSrc**	**BrWr**	**ExtOp**	**MemtoReg**	**ALUOp**	**ALUSelB**
x	0	0	x	x	ADD	0

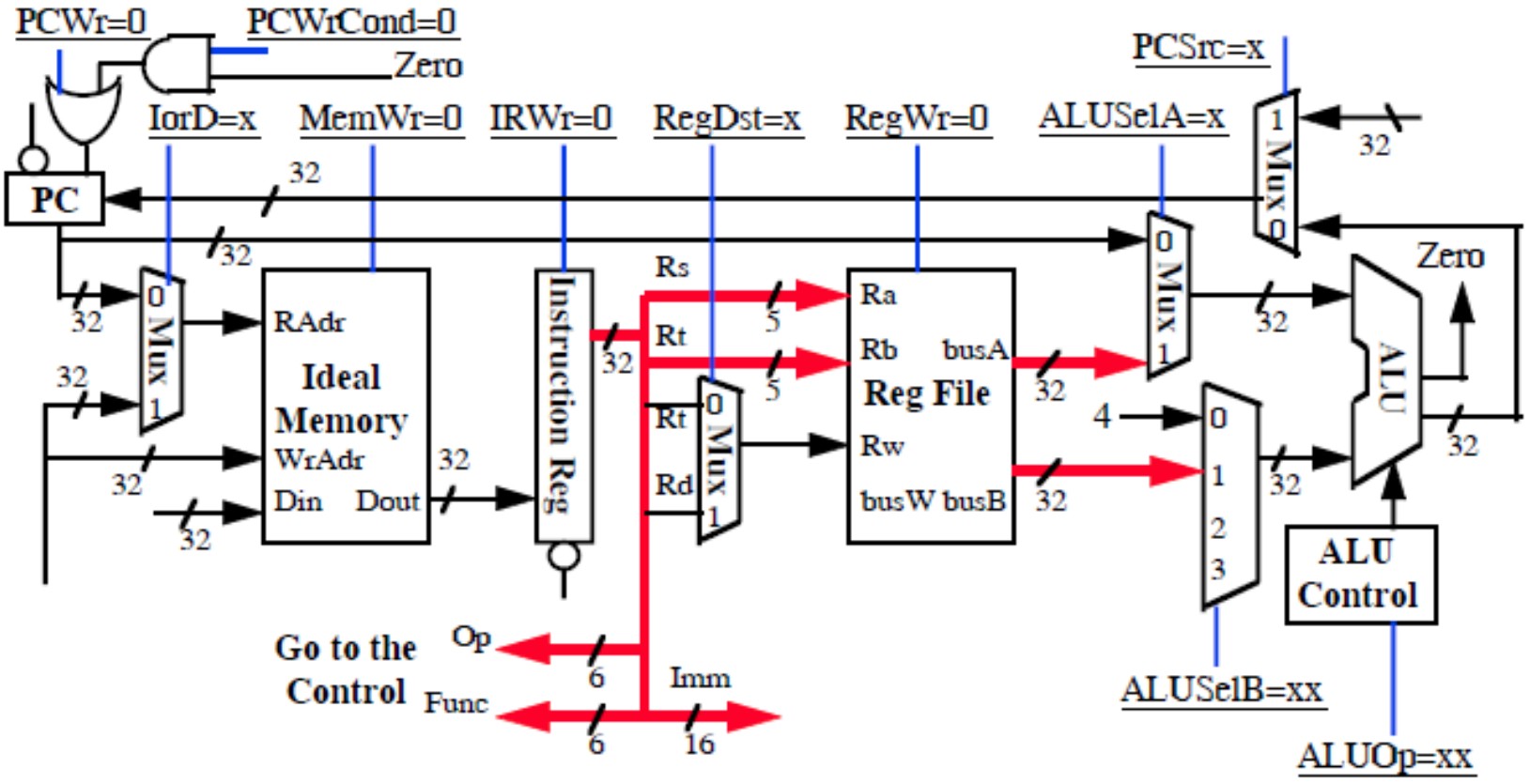
Ifetch

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



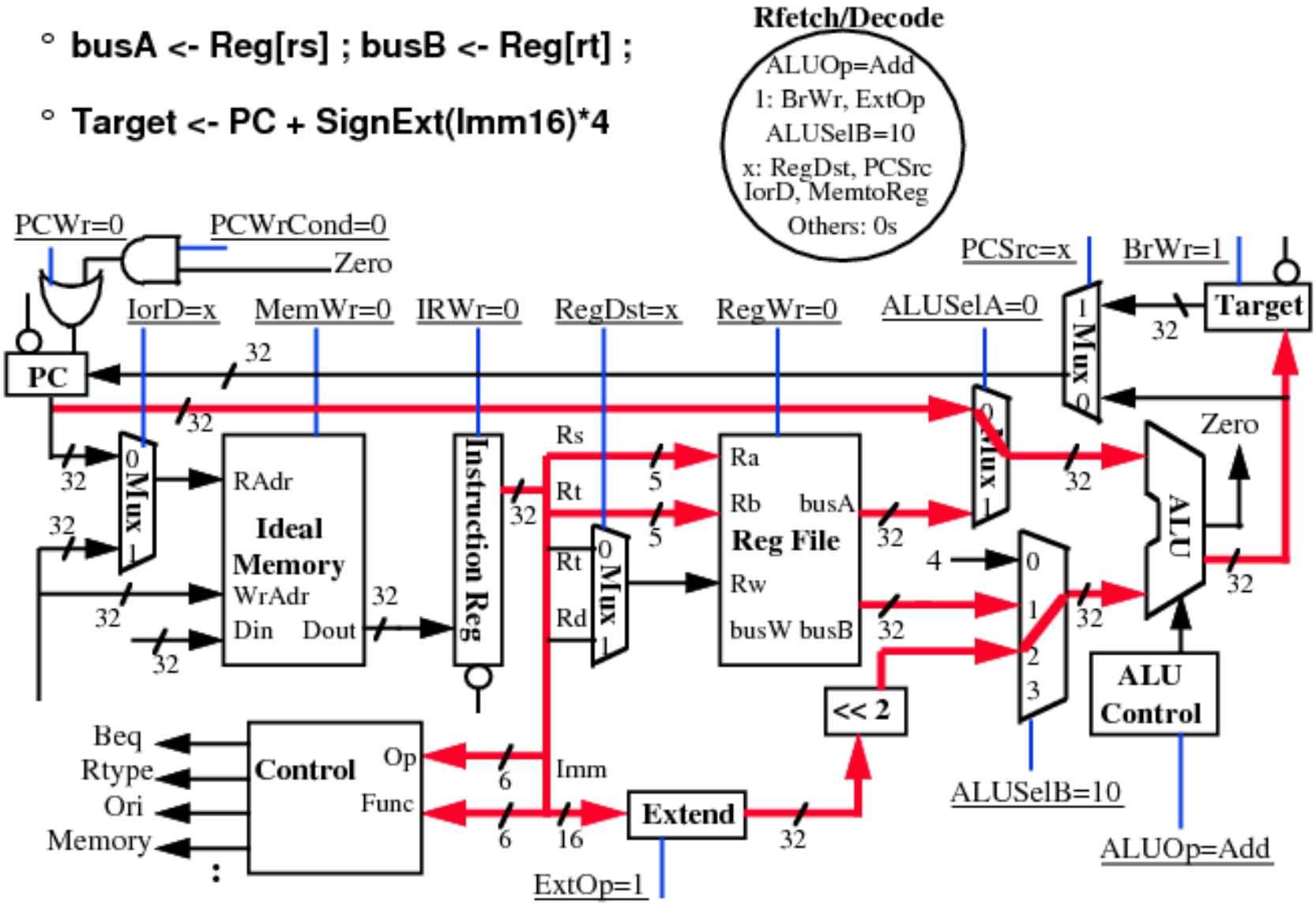
Register Fetch / Instruction Decode

- busA <- RegFile[rs] ; busB <- RegFile[rt] ;
- ALU is not being used: ALUctr = xx



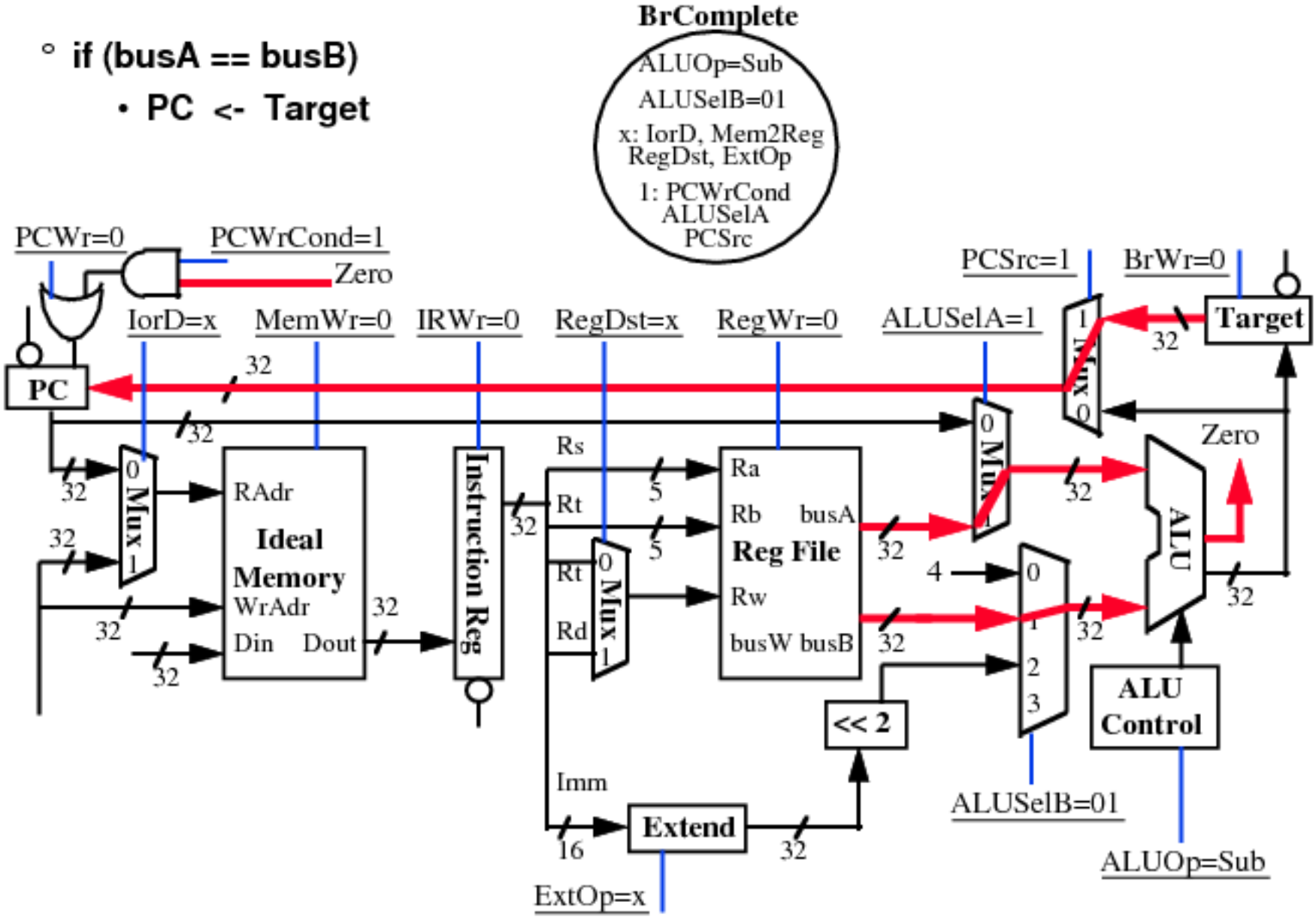
Register Fetch / Instruction Decode (Continue)

- busA <- Reg[rs] ; busB <- Reg[rt] ;
- Target <- PC + SignExt(Imm16)*4



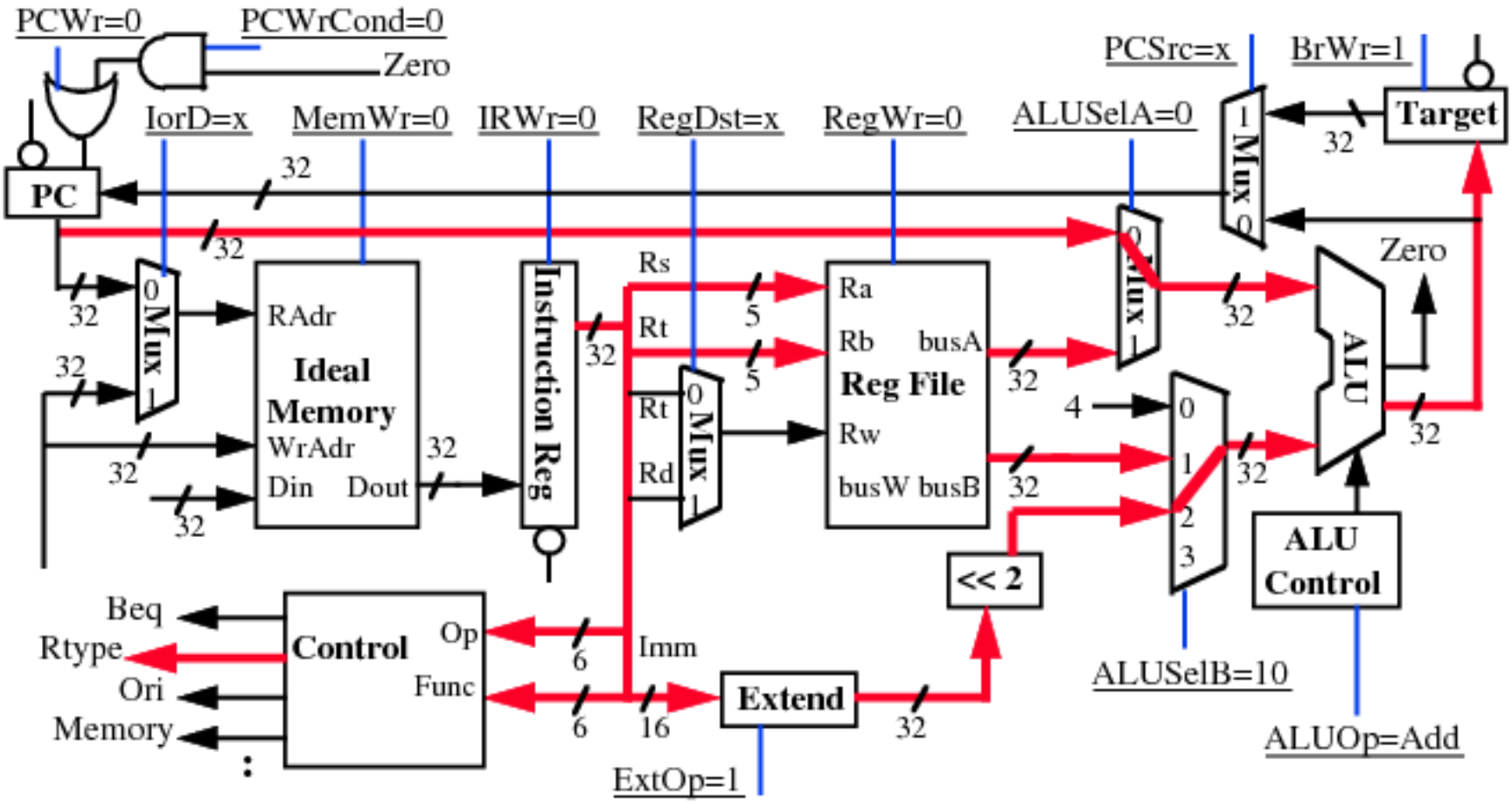
Branch Completion

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



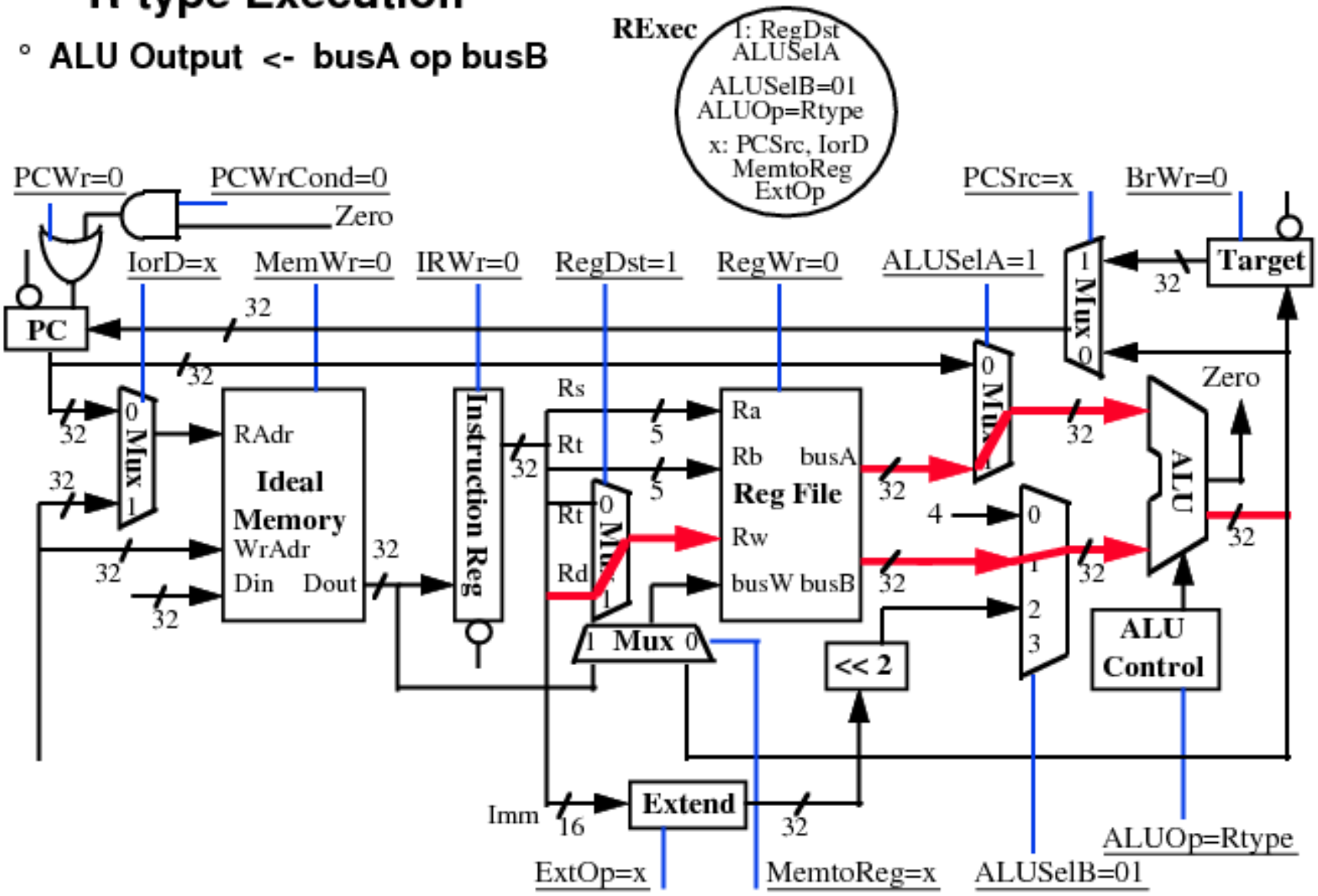
Instruction Decode: We have a R-type!

- Next Cycle: R-type Execution



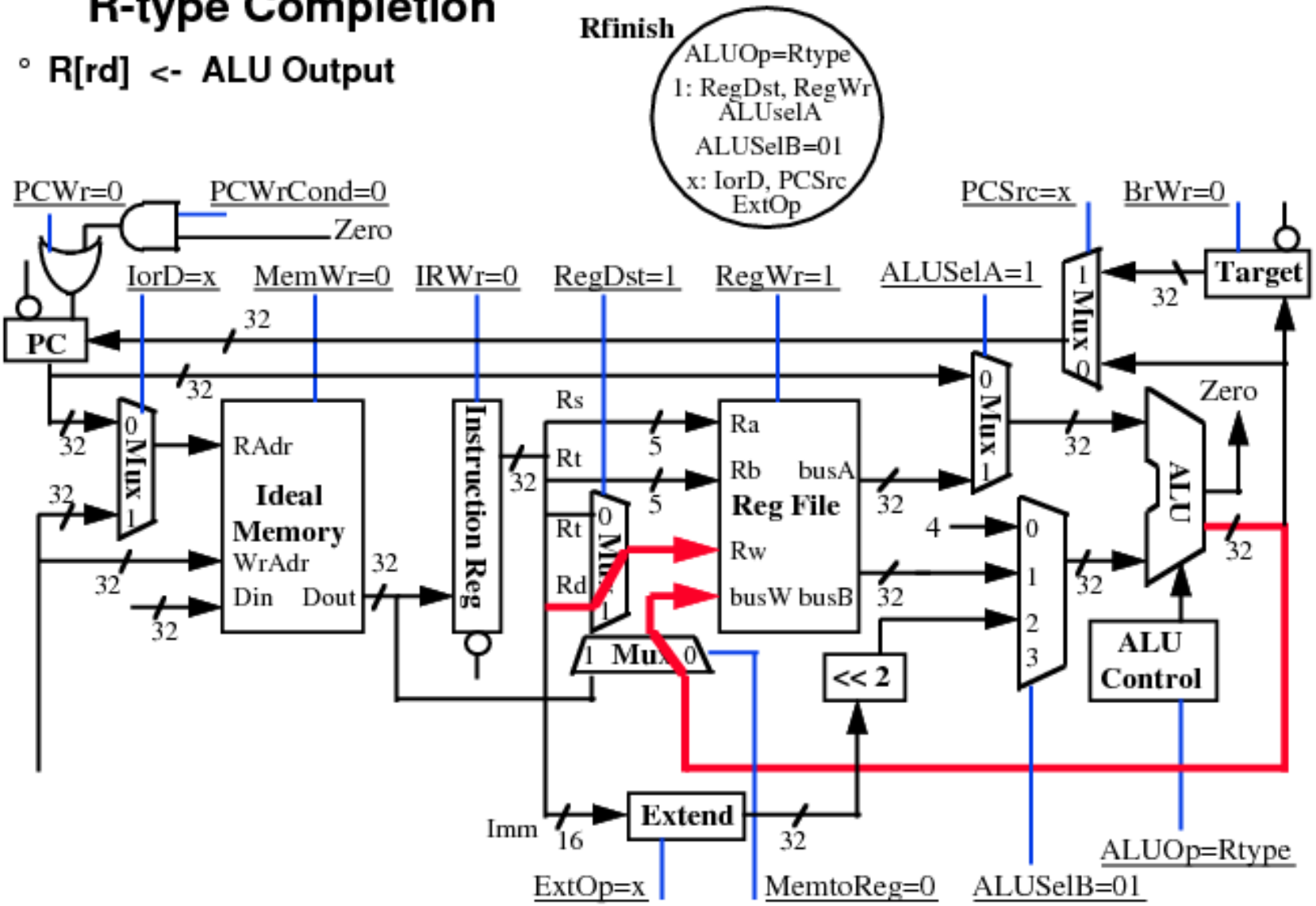
R-type Execution

° ALU Output <- busA op busB



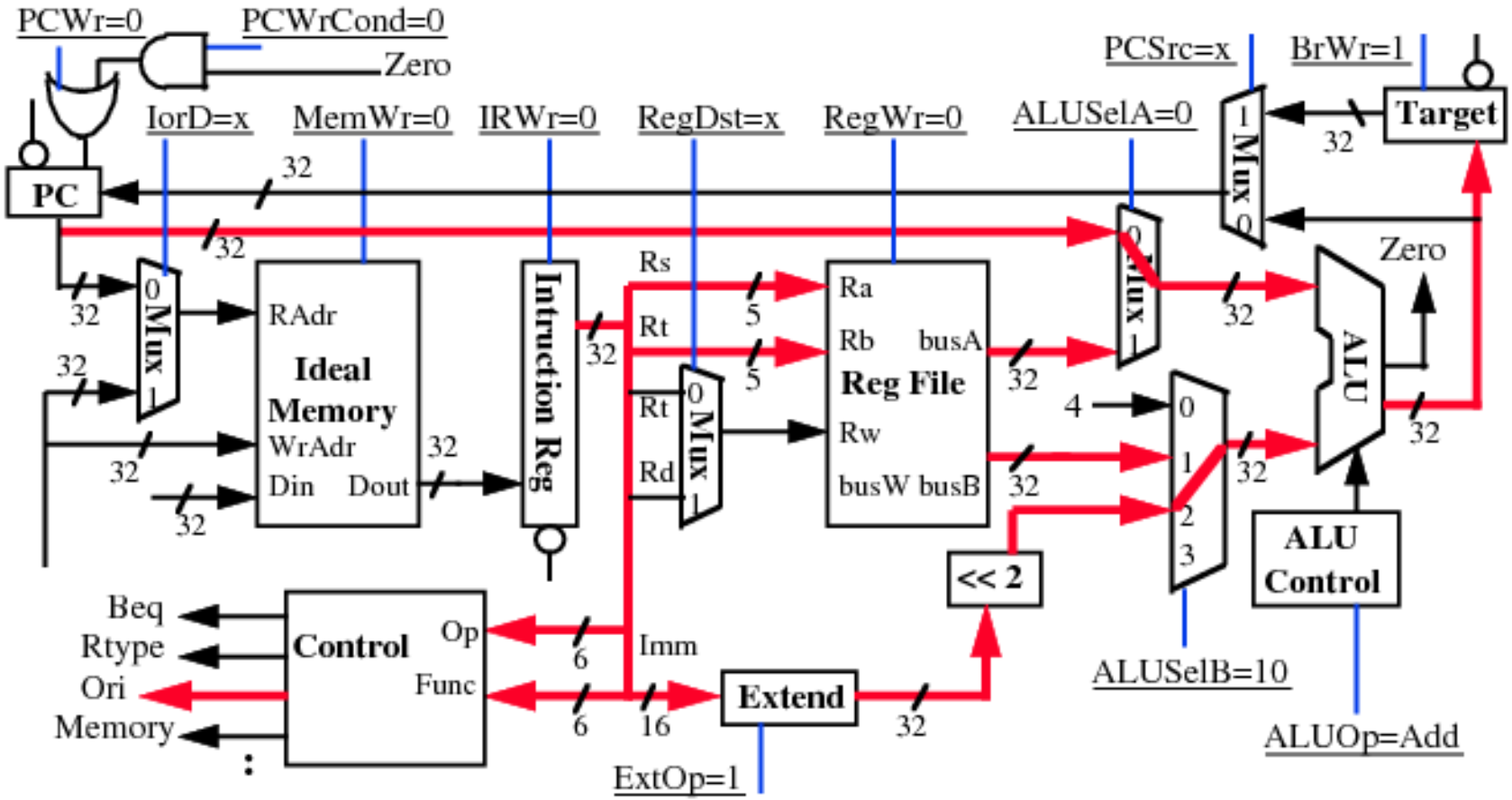
R-type Completion

° R[rd] <- ALU Output



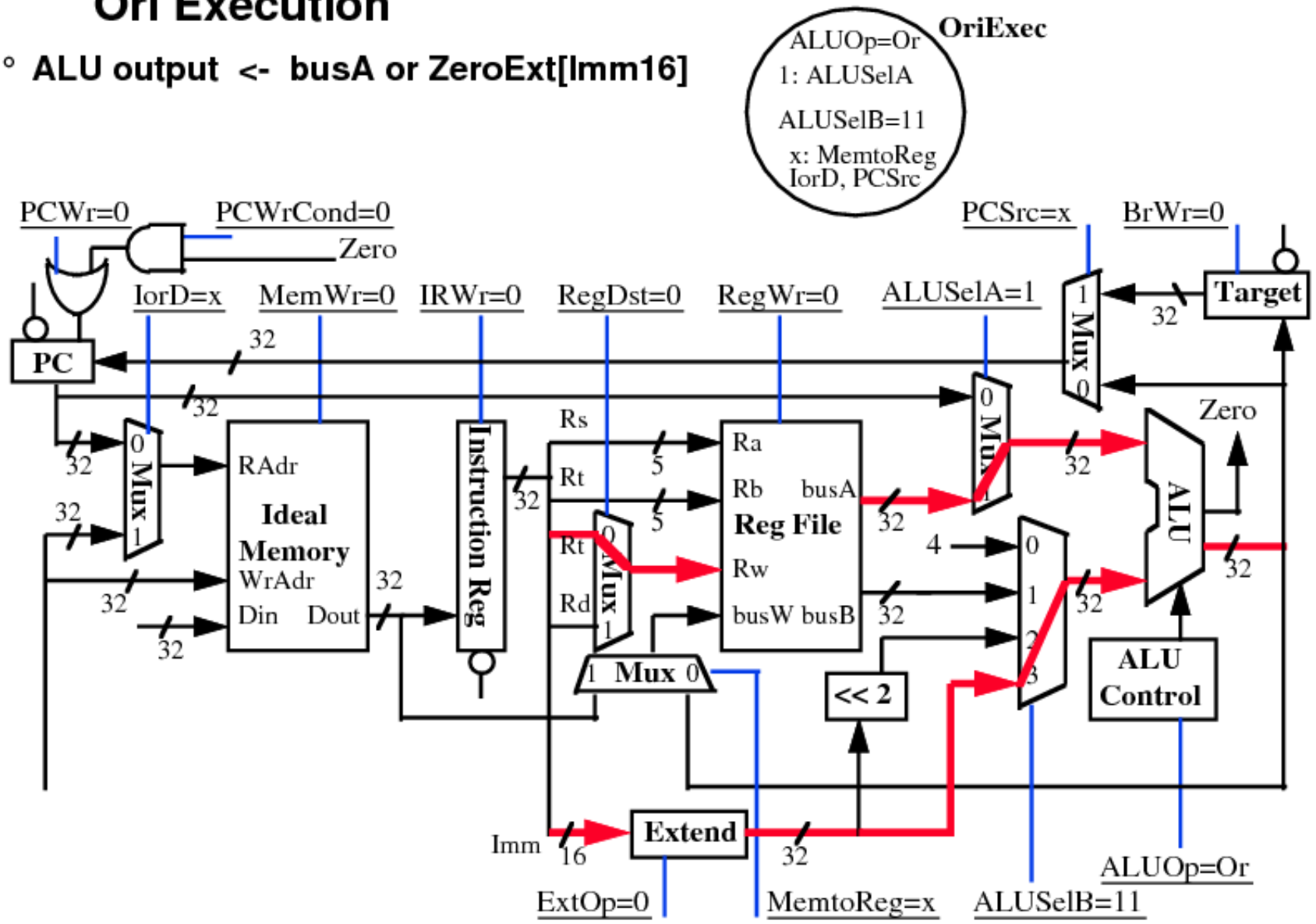
Instruction Decode: We have an Ori!

- Next Cycle: Ori Execution



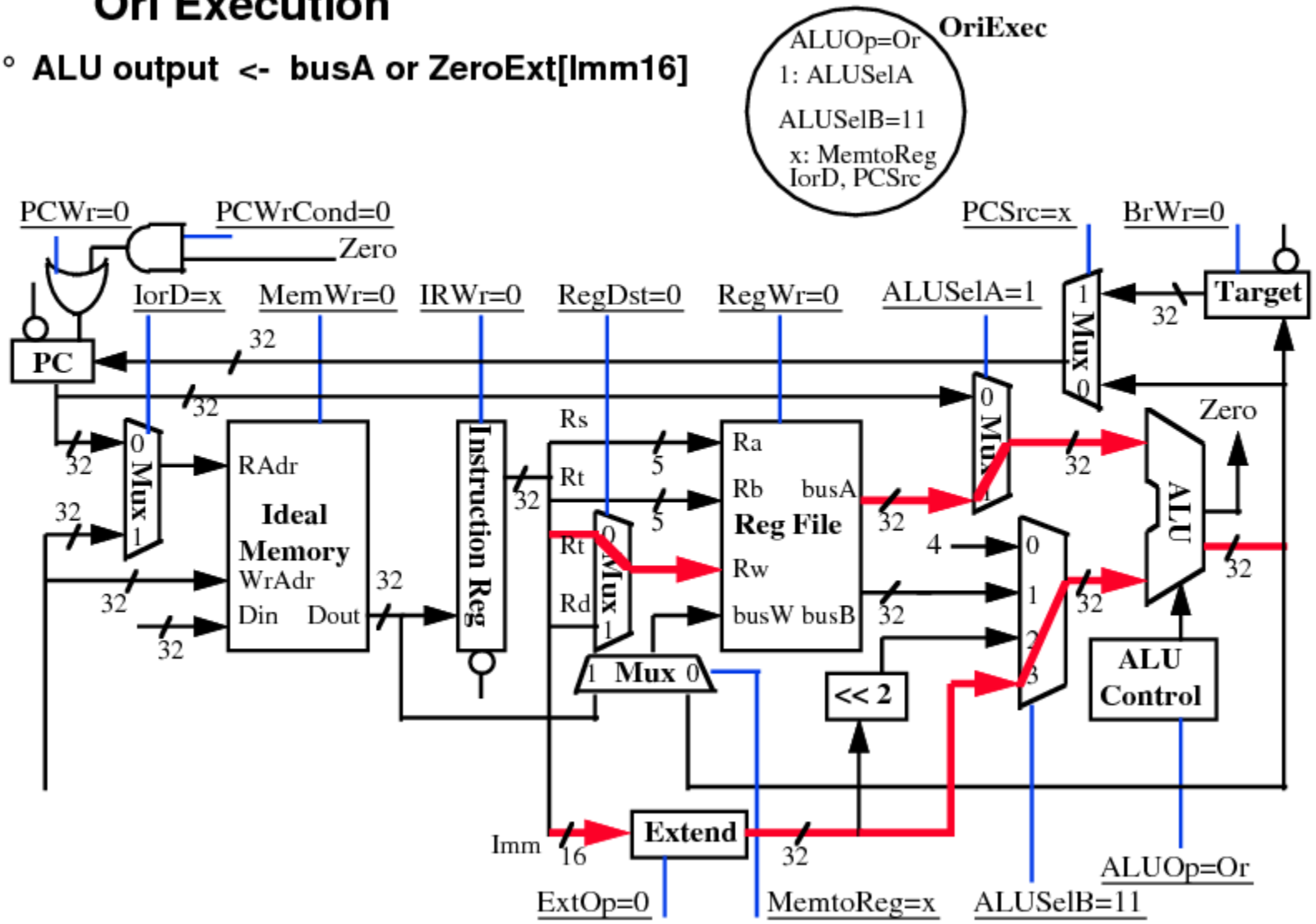
Ori Execution

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



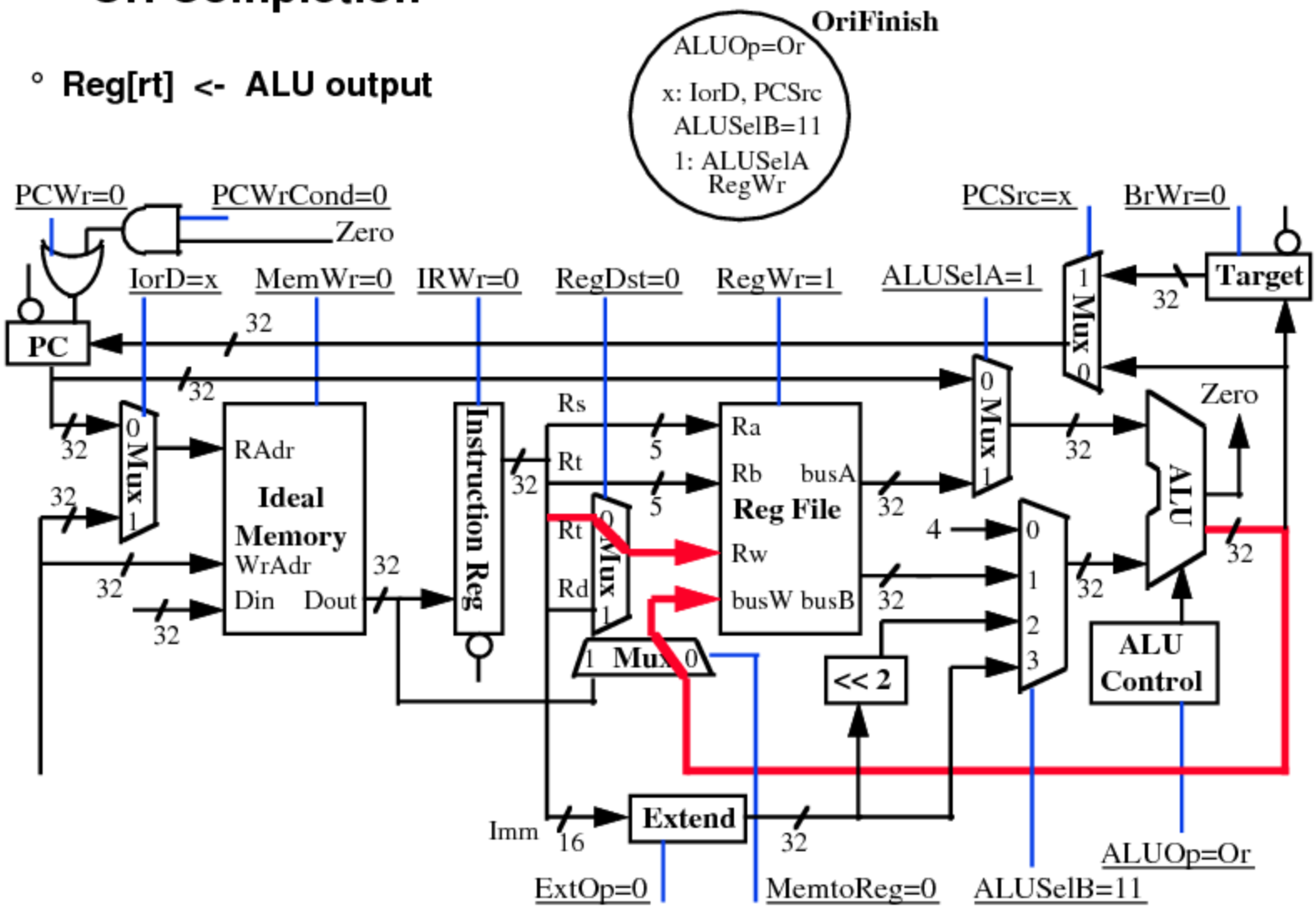
Ori Execution

° ALU output <- busA or ZeroExt[Imm16]



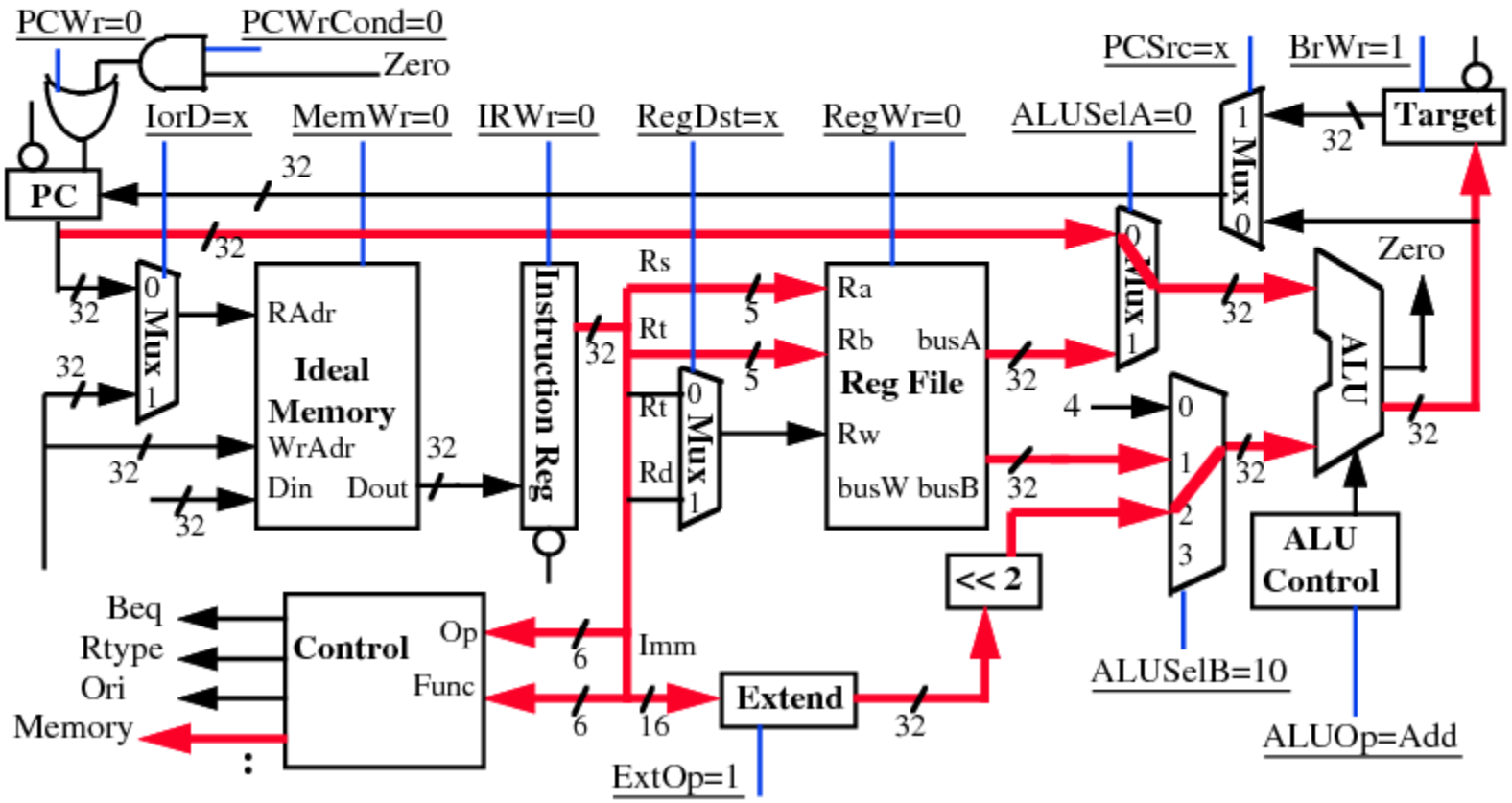
Ori Completion

- Reg[rt] <- ALU output



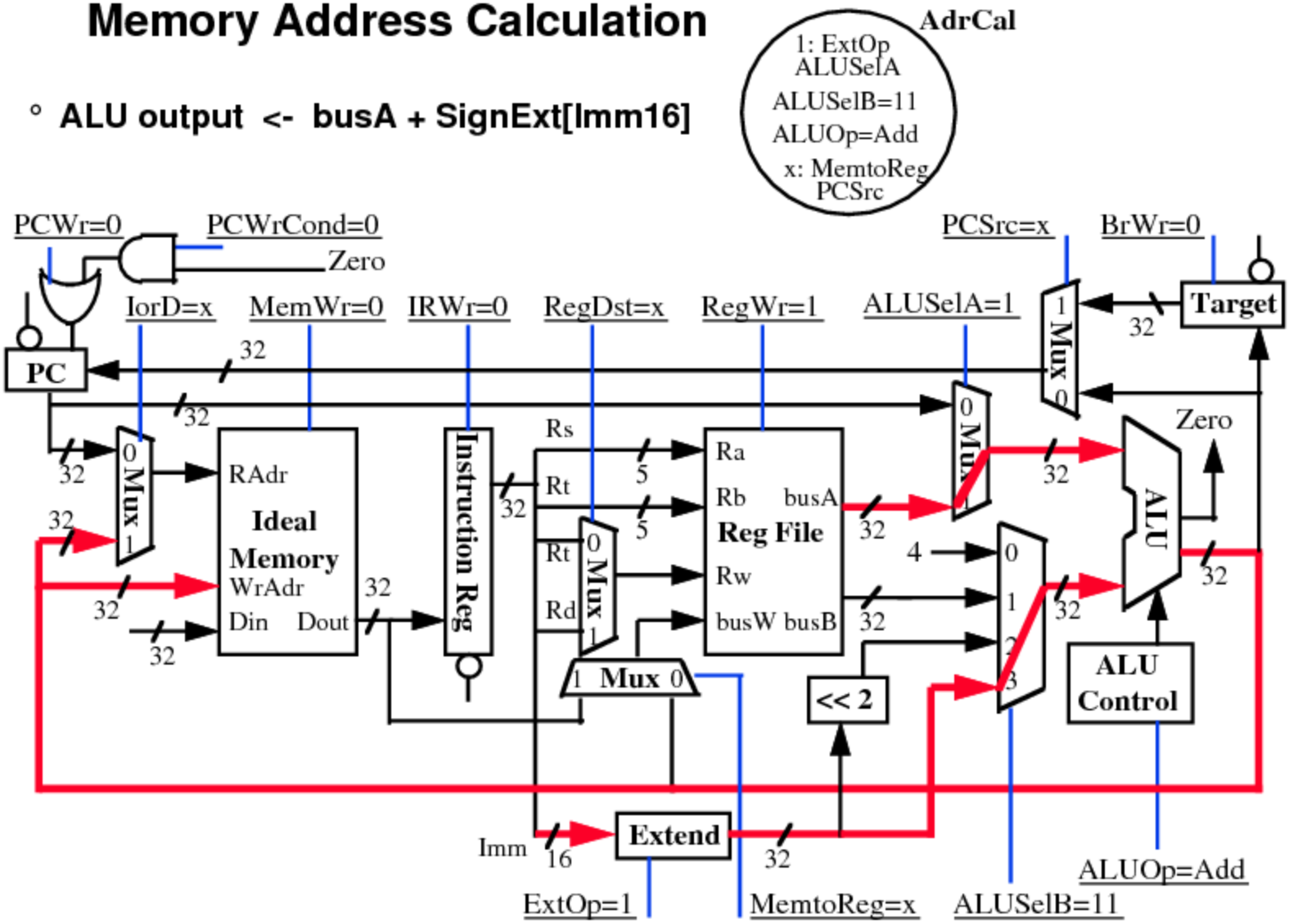
Instruction Decode: We have a Memory Access!

- **Next Cycle: Memory Address Calculation**



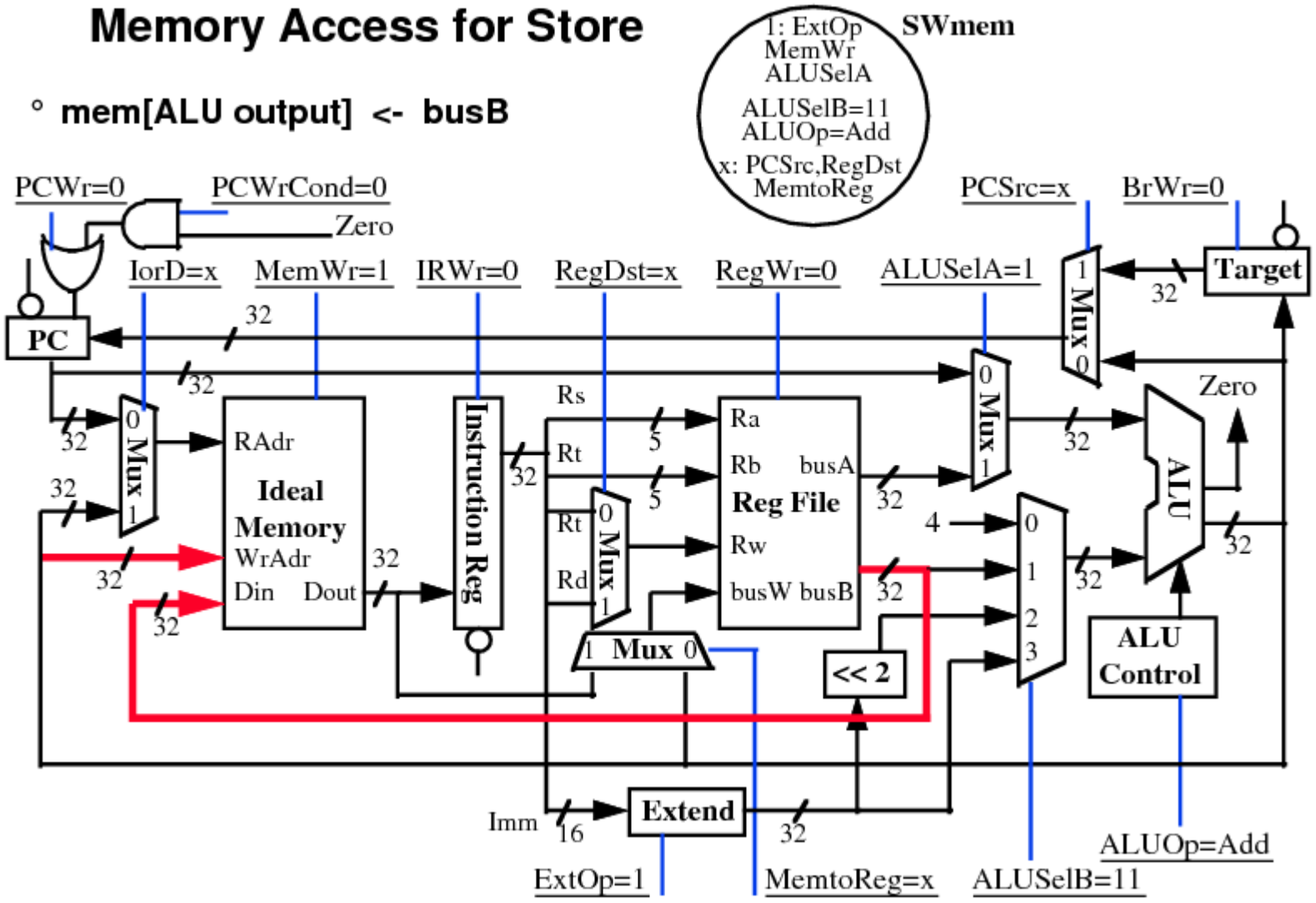
Memory Address Calculation

- **ALU output <- busA + SignExt[Imm16]**



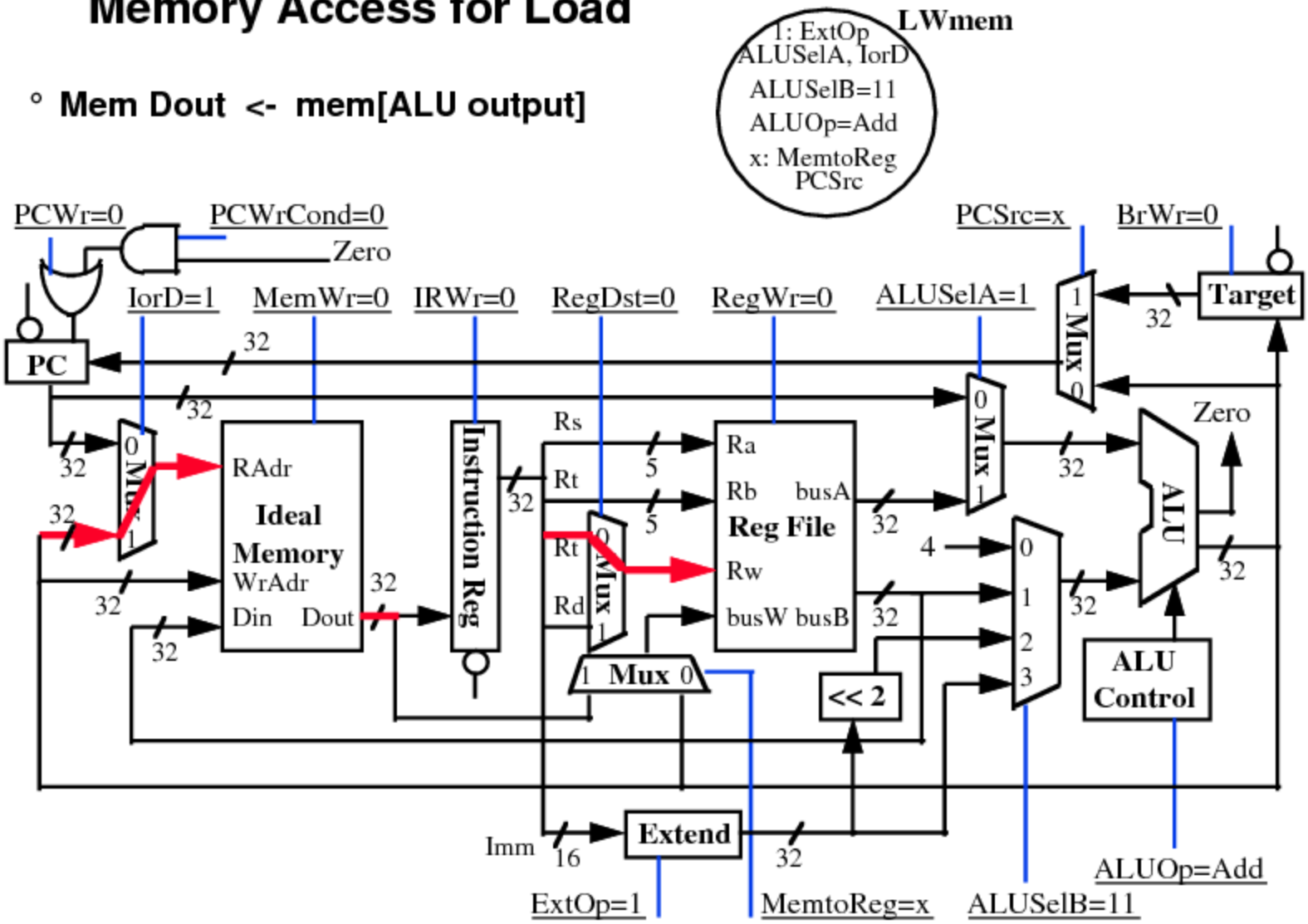
Memory Access for Store

° mem[ALU output] <- busB



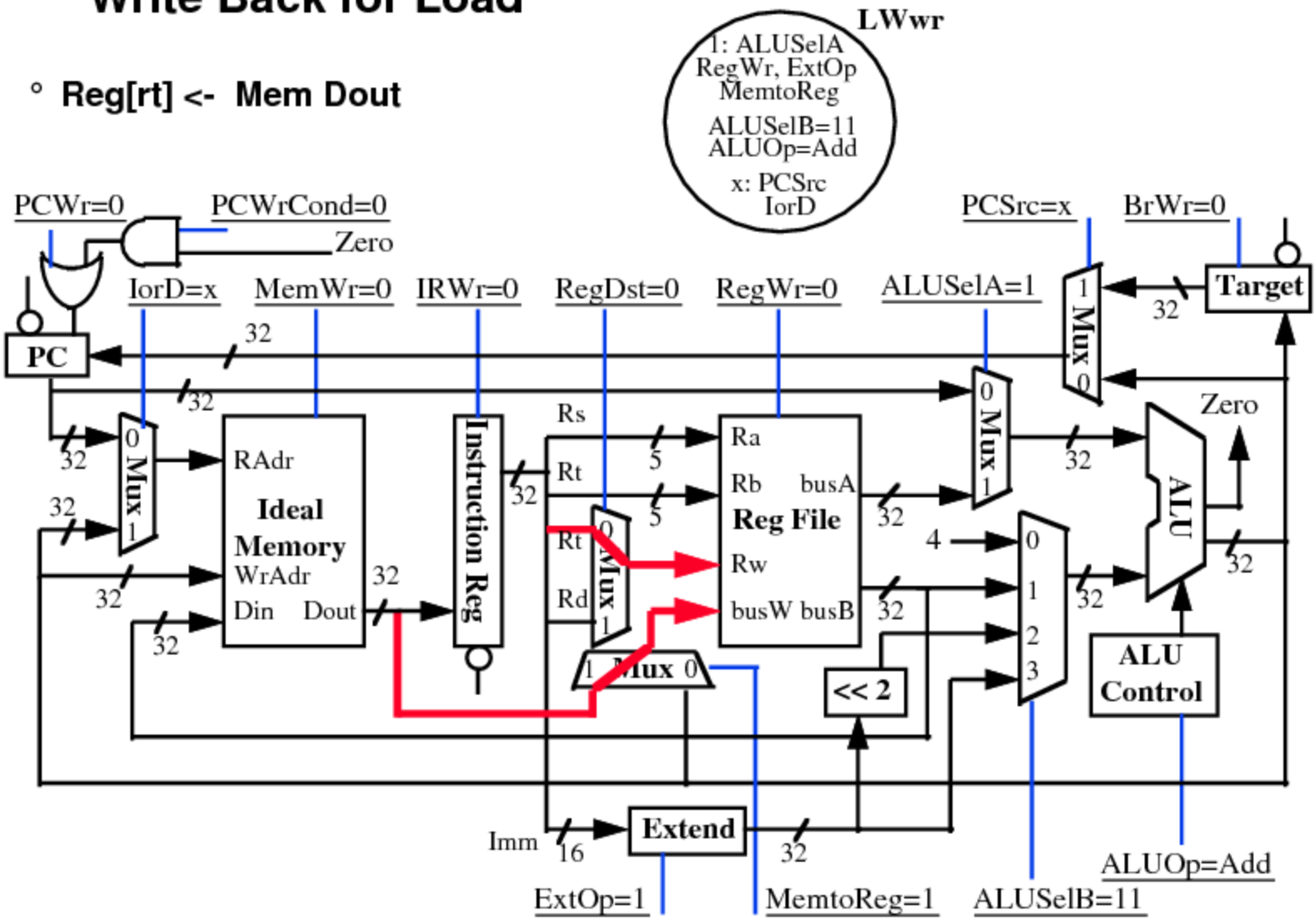
Memory Access for Load

° Mem Dout <- mem[ALU output]

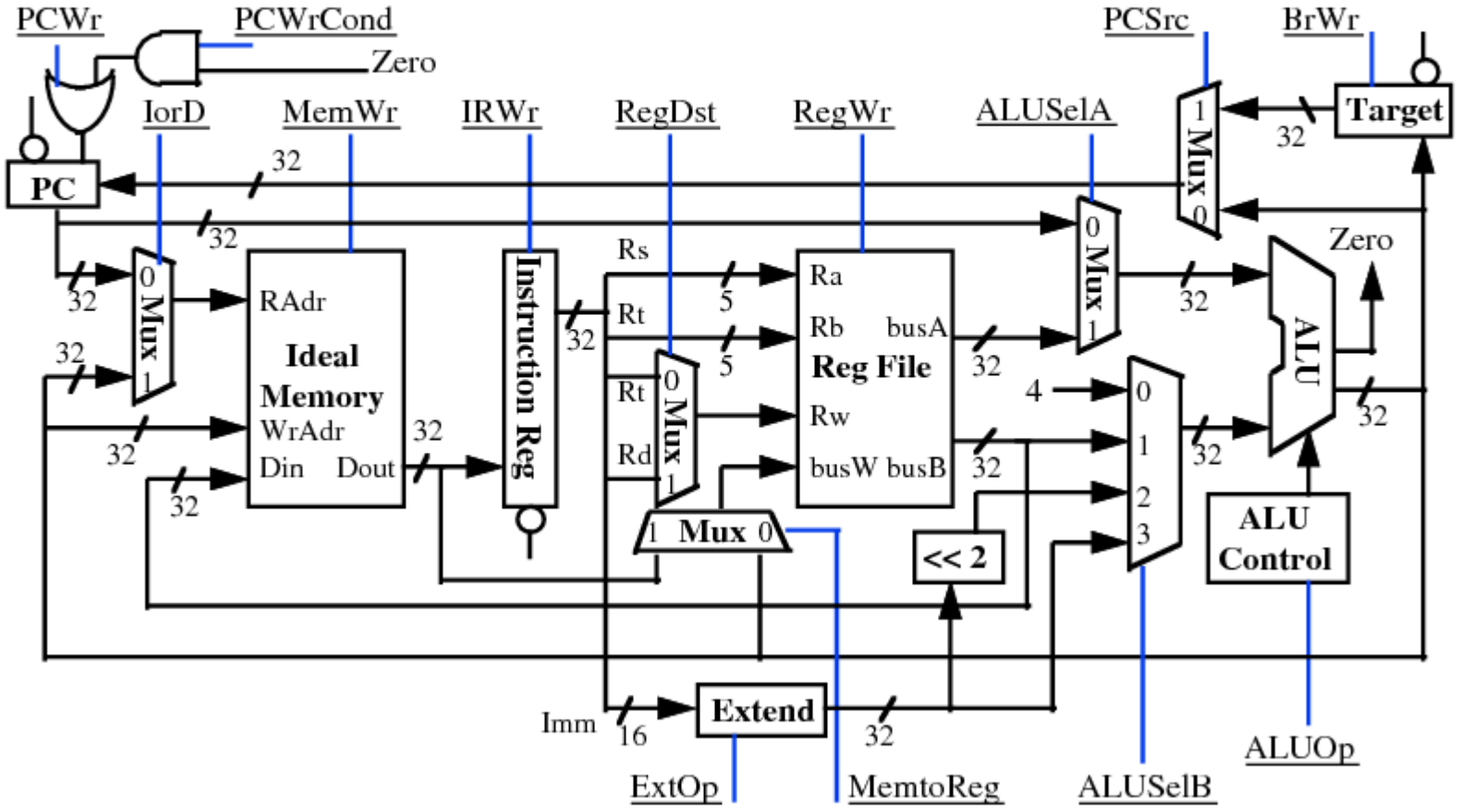


Write Back for Load

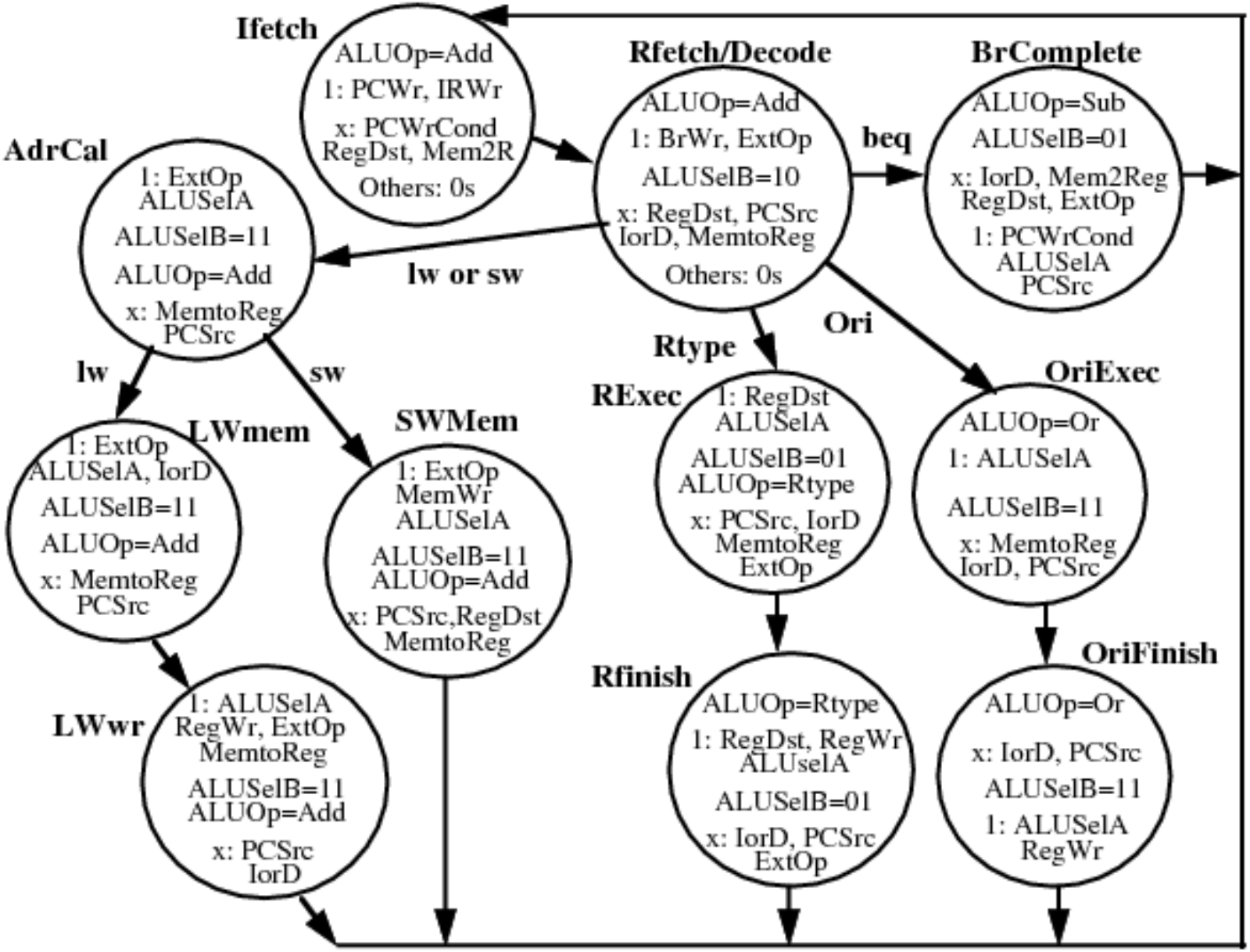
- Reg[rt] <- Mem Dout



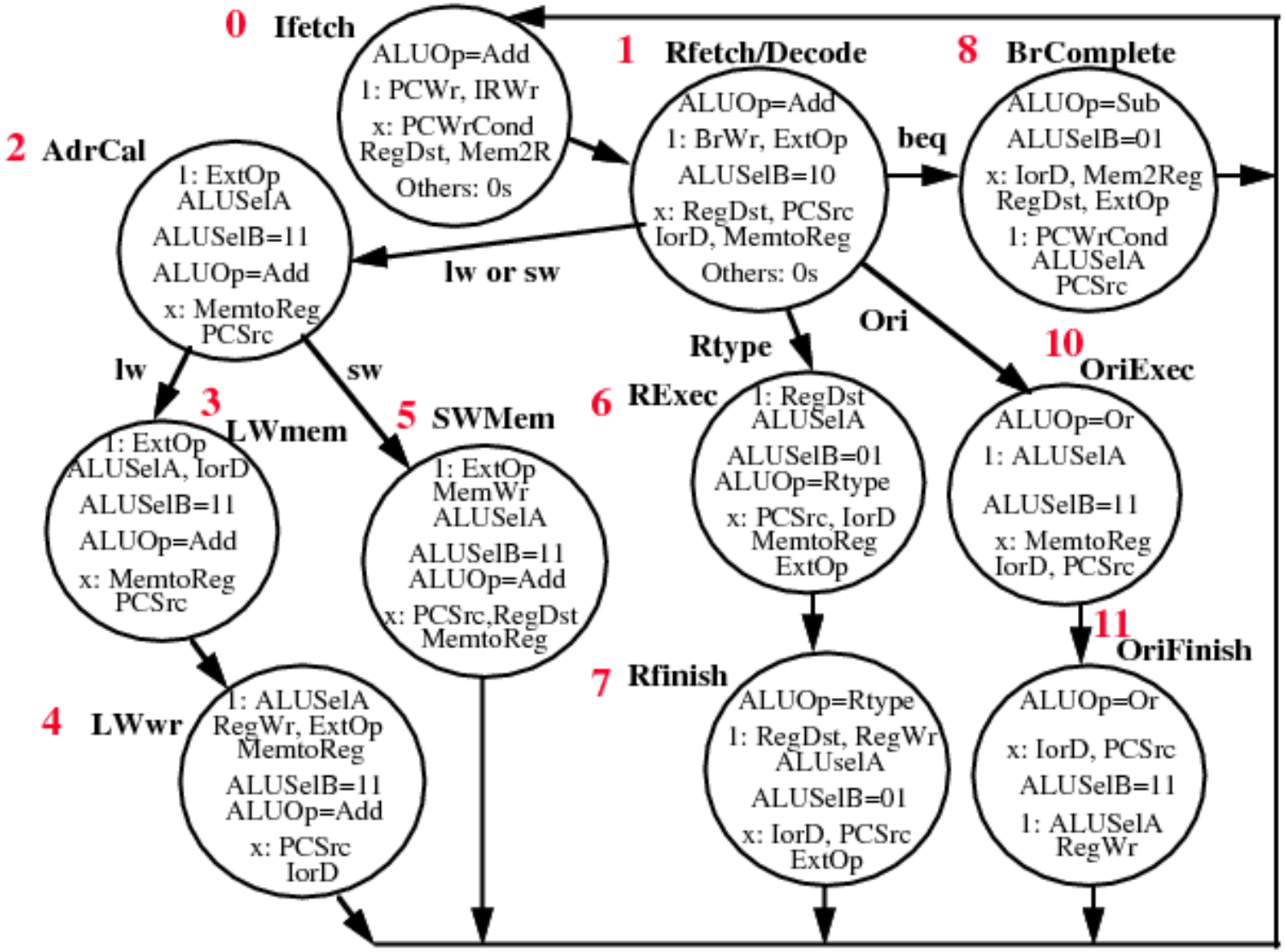
Putting it all together: Multiple Cycle Datapath



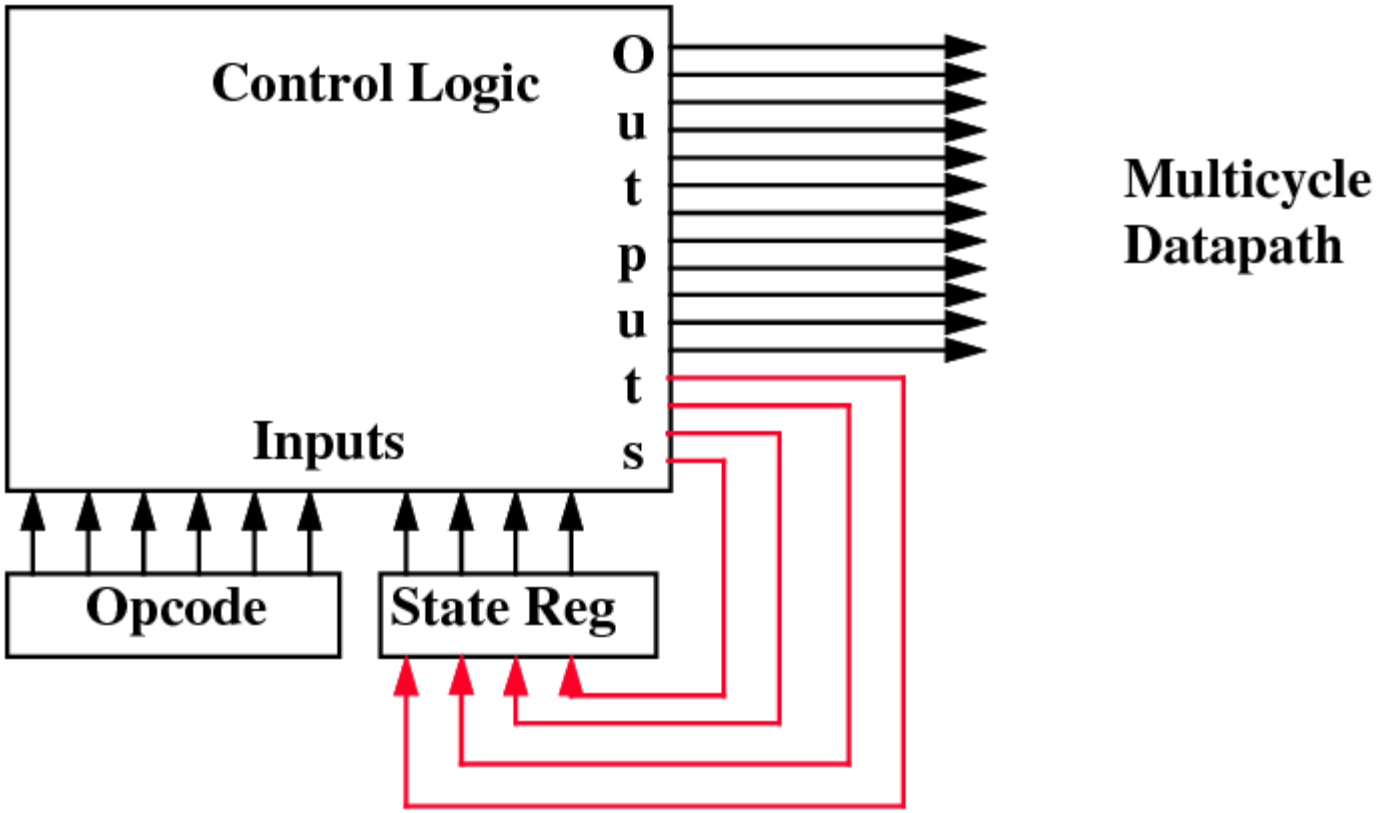
Putting it all together: Control State Diagram



Control Logic in the Form of Finite State Diagram



Sequencing Control: Explicit Next State Function



Summary

- Disadvantages of the Single Cycle Processor
  - Long cycle time
  - Cycle time is too long for all instructions except load
- Multiple cycle processor
  - Divide the instruction into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle