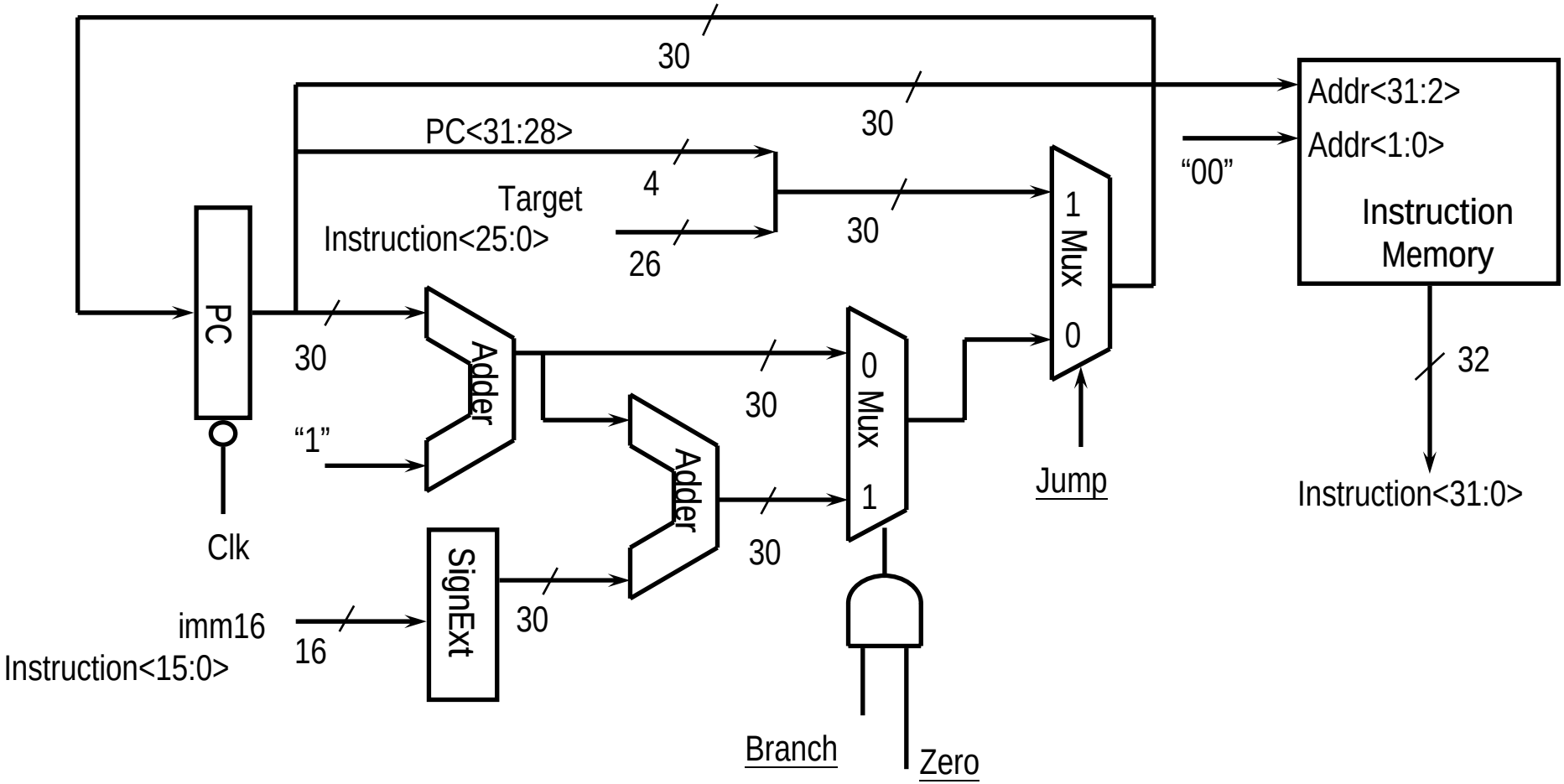
Computer Architecture

Ch. 5-3: Designing a Multiple Cycle Processor

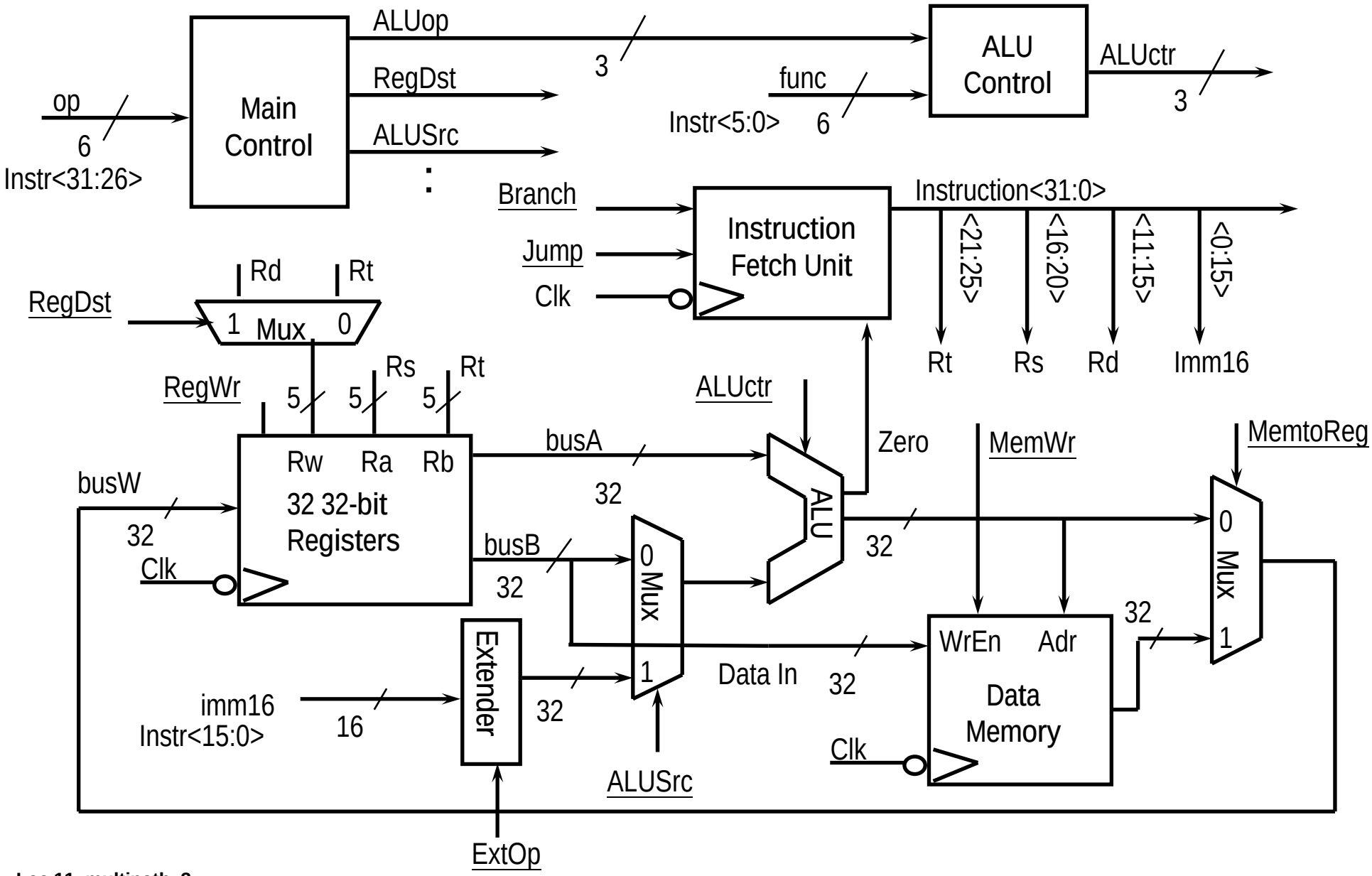
Spring, 2005

Sao-Jie Chen (csj@cc.ee.ntu.edu.tw)

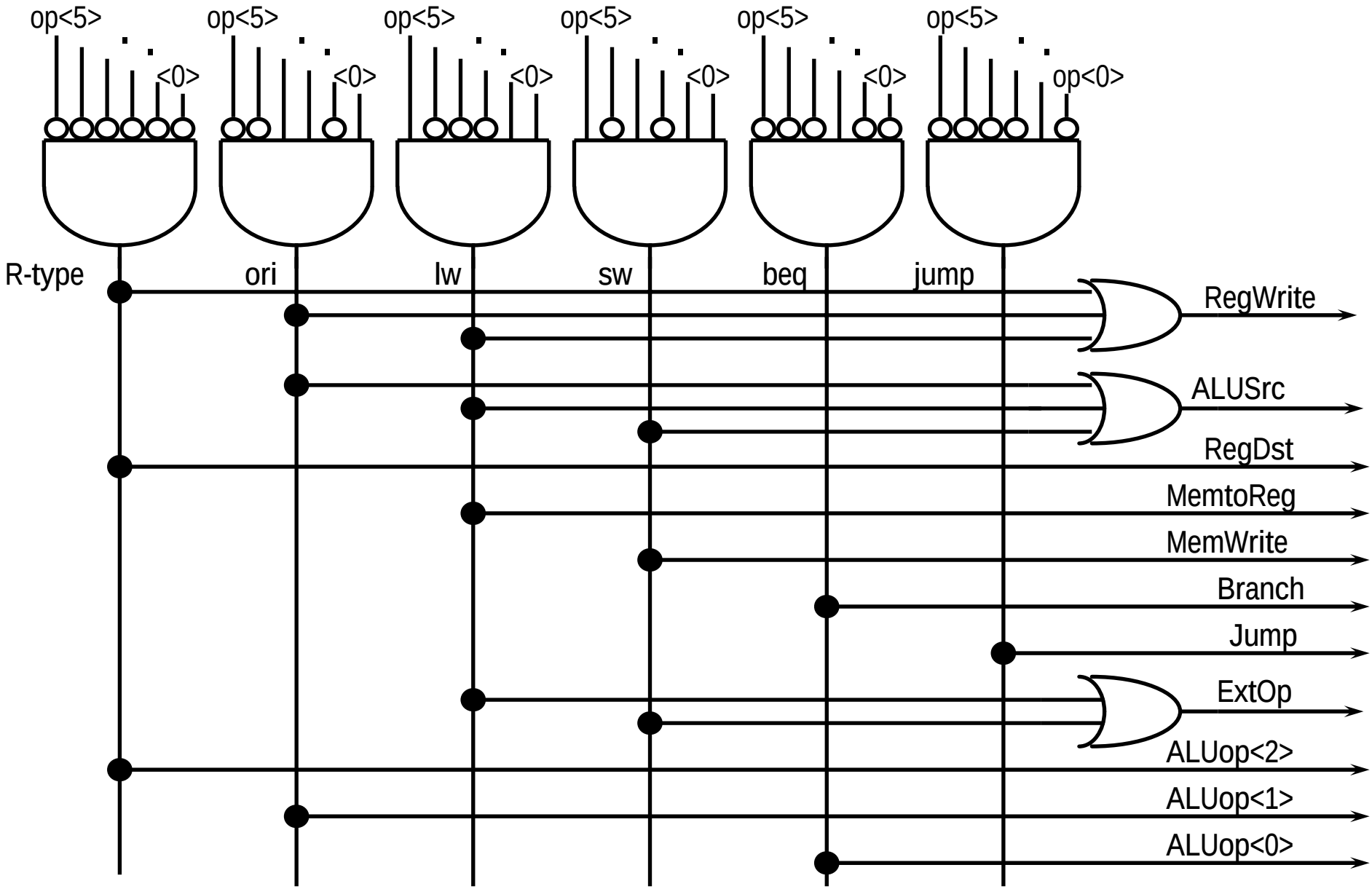
Recap: Instruction Fetch Unit



Recap: A Single Cycle Processor



Recap: The Main Control



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

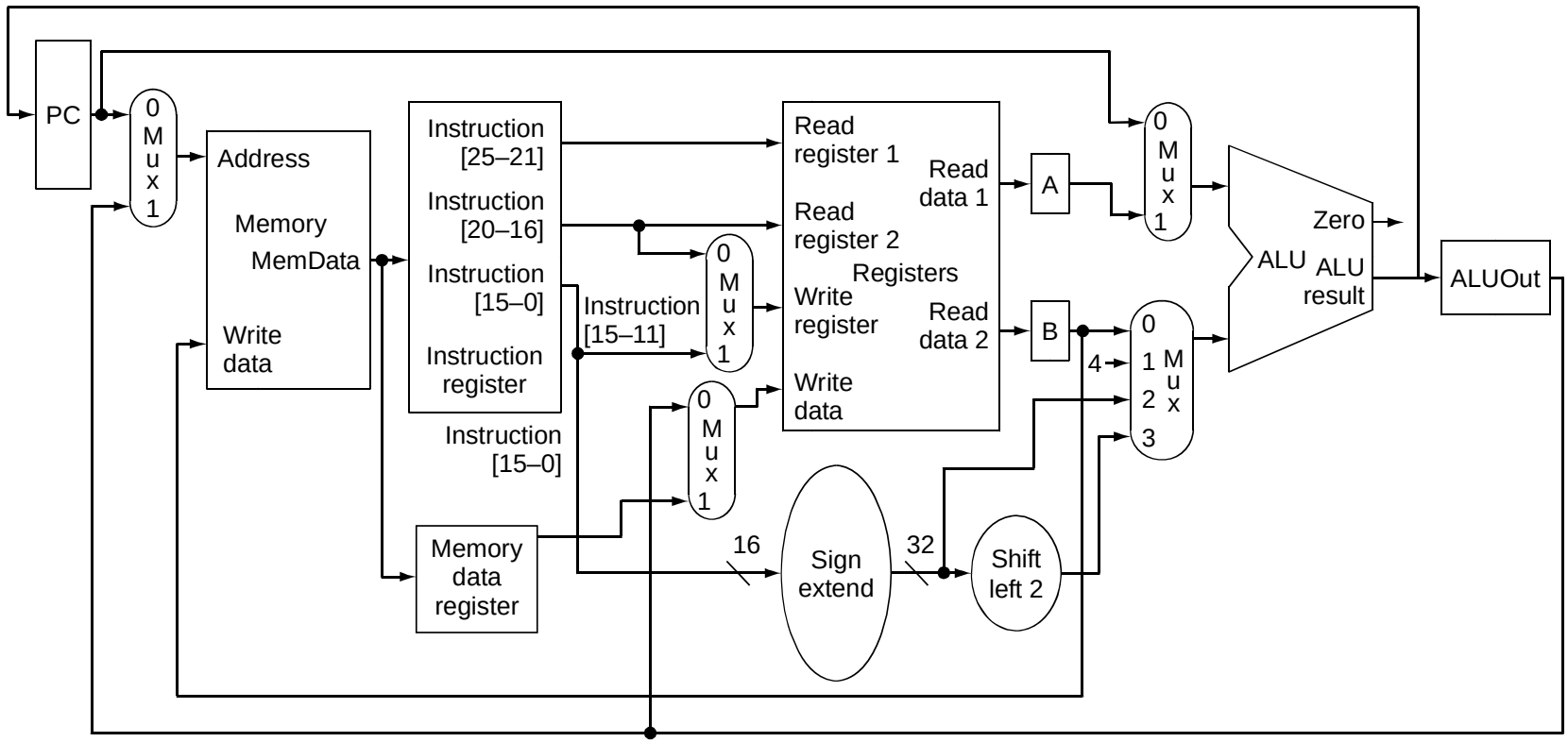
- The root of the single cycle processor's problems:
  - The cycle time has to be long enough for the slowest instruction
- Solution:
  - Break the instruction into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
    - ⇒ Cycle time: time it takes to execute the longest 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

Multicycle Approach

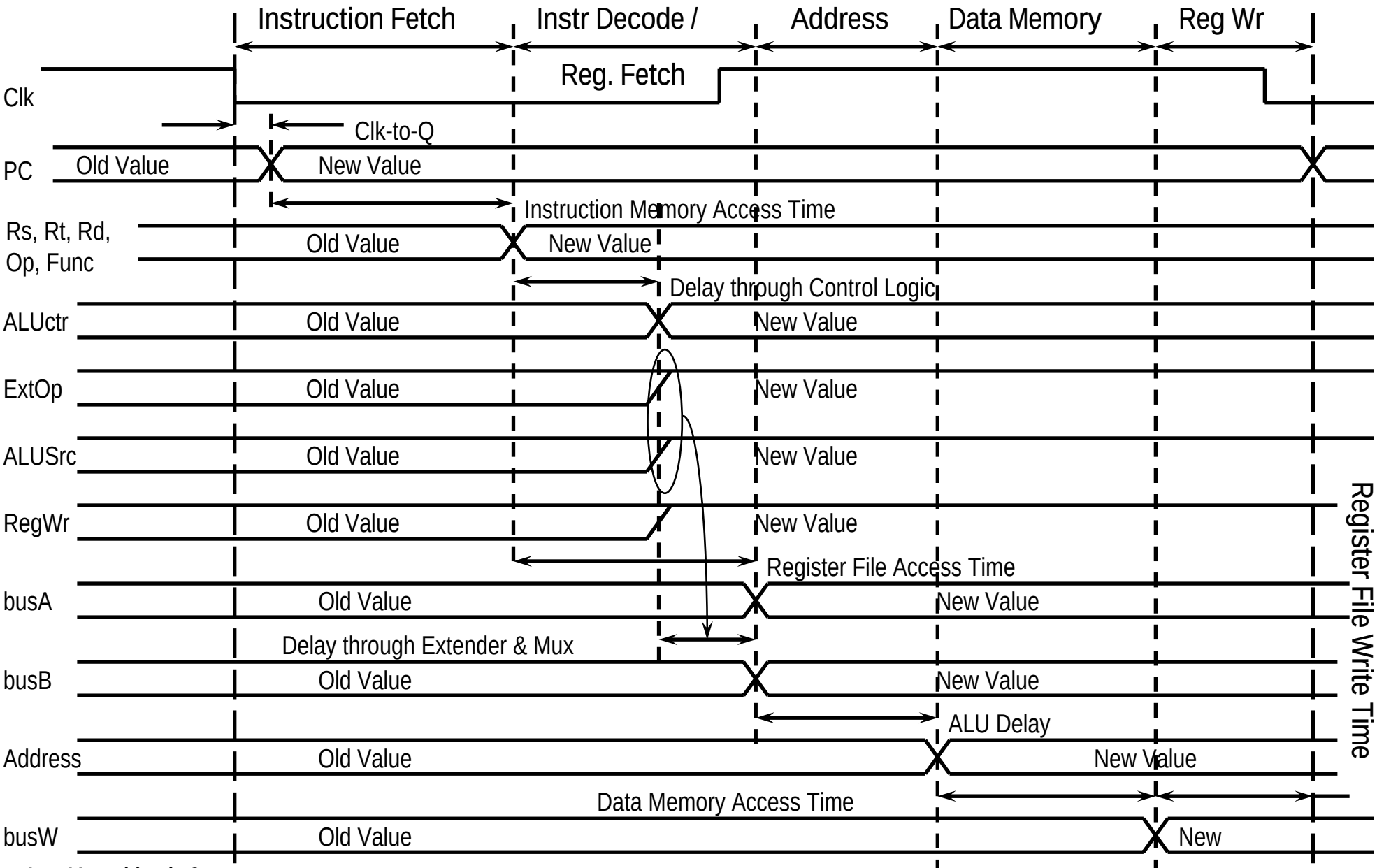
- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined solely by instruction
  - e.g., what should the ALU do for a *subtract* instruction?
- We'll use a finite state machine for control

Multicycle Approach

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional *internal* registers

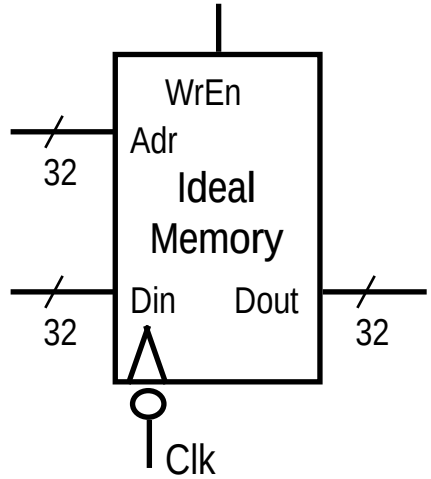


The Five Steps of a Load Instruction

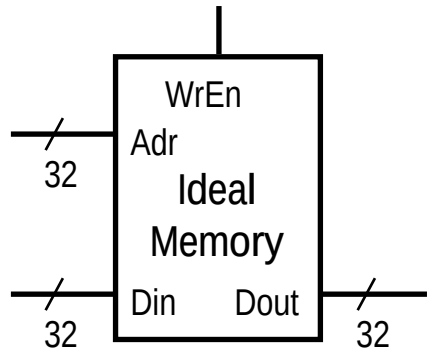


Register File & Memory Write Timing: Ideal vs. Reality

- In previous lectures, register file and memory are simplified:
  - Write happens at the clock tick
  - Address (Adr), input data (Din), and write enable (WrEn) must be stable one set-up time before the clock tick

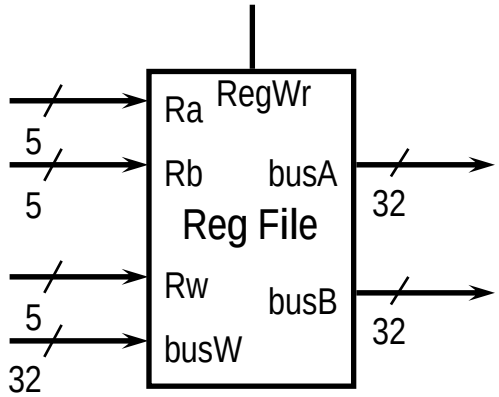


- In real life:
  - Neither register file nor ideal memory has the clock input
  - The write path is a combinational logic delay path:
    - ⇒ WrEn goes to 1 and Din (Data In) settles down
    - ⇒ Memory write access delay
    - ⇒ Din is written into mem[address]
  - Important: Address and Data must be stable BEFORE Write Enable goes to 1

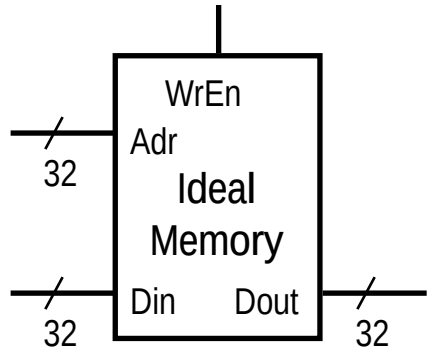


Race Condition Between Address and Write Enable

- This real (no clock input) register file may not work reliably in the single cycle processor because:
  - We cannot guarantee Rw will be stable BEFORE RegWr = 1
  - There is a race between Rw (address) and RegWr (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

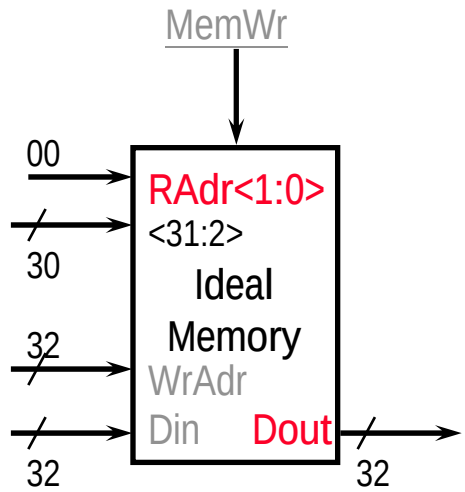


How to Avoid this Race Condition?

- **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)**
  - **We have to make sure that address cannot change until Write Enable is disasserted**

Dual-Port Ideal Memory

- **Dual Port Ideal Memory**
  - **Independent Read (RAdr, Dout) and Write (WrAdr, Din) ports**
  - **Read and write (to different location) can occur at the same cycle**
- **Read Port is a combinational path:**
  - **Read Address (RAdr) Valid -->**
  - **Memory Read Access Delay -->**
  - **Data Out (Dout) Valid**
- **Write Port is also a combinational path:**
  - **MemWrite MemWr = 1 -->**
  - **Memory Write Access Delay -->**
  - **Data In (Din) is written into location [WrAdr]**



Instructions from ISA perspective

- Consider each instruction from perspective of ISA.
- Example:
  - The add instruction changes a register.
  - Register specified by bits 15:11 of instruction.
  - Instruction specified by the PC.
  - New value is the sum ("op") of two registers.
  - Registers specified by bits 25:21 and 20:16 of the instruction

```
Reg[Memory[PC][15:11]] <=  Reg[Memory[PC][25:21]] op
                           Reg[Memory[PC][20:16]]
```

  - In order to accomplish this we must break up the instruction.
    (kind of like introducing variables when programming)

Breaking down an instruction

- ISA definition of arithmetic:

```
Reg[Memory[PC][15:11]] <= Reg[Memory[PC][25:21]] op
                          Reg[Memory[PC][20:16]]
```

- Could break down to:
  - `IR <= Memory[PC]`
  - `A <= Reg[IR[25:21]]`
  - `B <= Reg[IR[20:16]]`
  - `ALUOut <= A op B`
  - `Reg[IR[20:16]] <= ALUOut`

- We forgot an important part of the definition of arithmetic!
  - `PC <= PC + 4`

Idea behind multicycle approach

- We define each instruction from the ISA perspective (do this!)
- Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)
- Introduce new registers as needed (e.g, A, B, ALUOut, MDR, etc.)
- Finally try and pack as much work into each step
  (avoid unnecessary cycles)
  while also trying to share steps where possible
  (minimizes control, helps to simplify solution)
- Result: Our book's multicycle Implementation!

Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

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

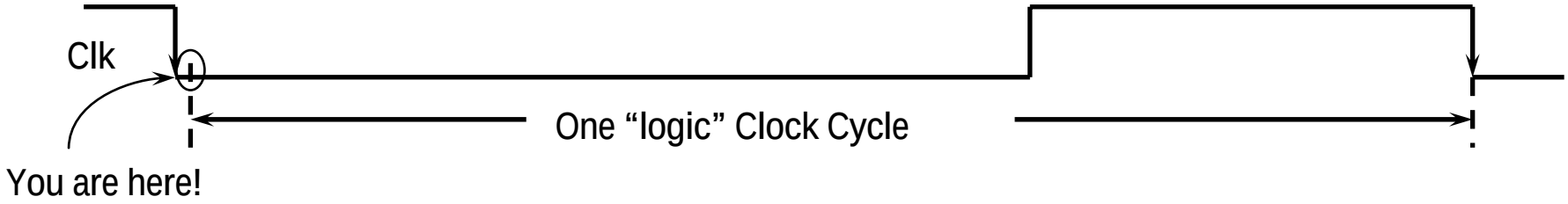
```
IR <= Memory[PC];
PC <= PC + 4;
```

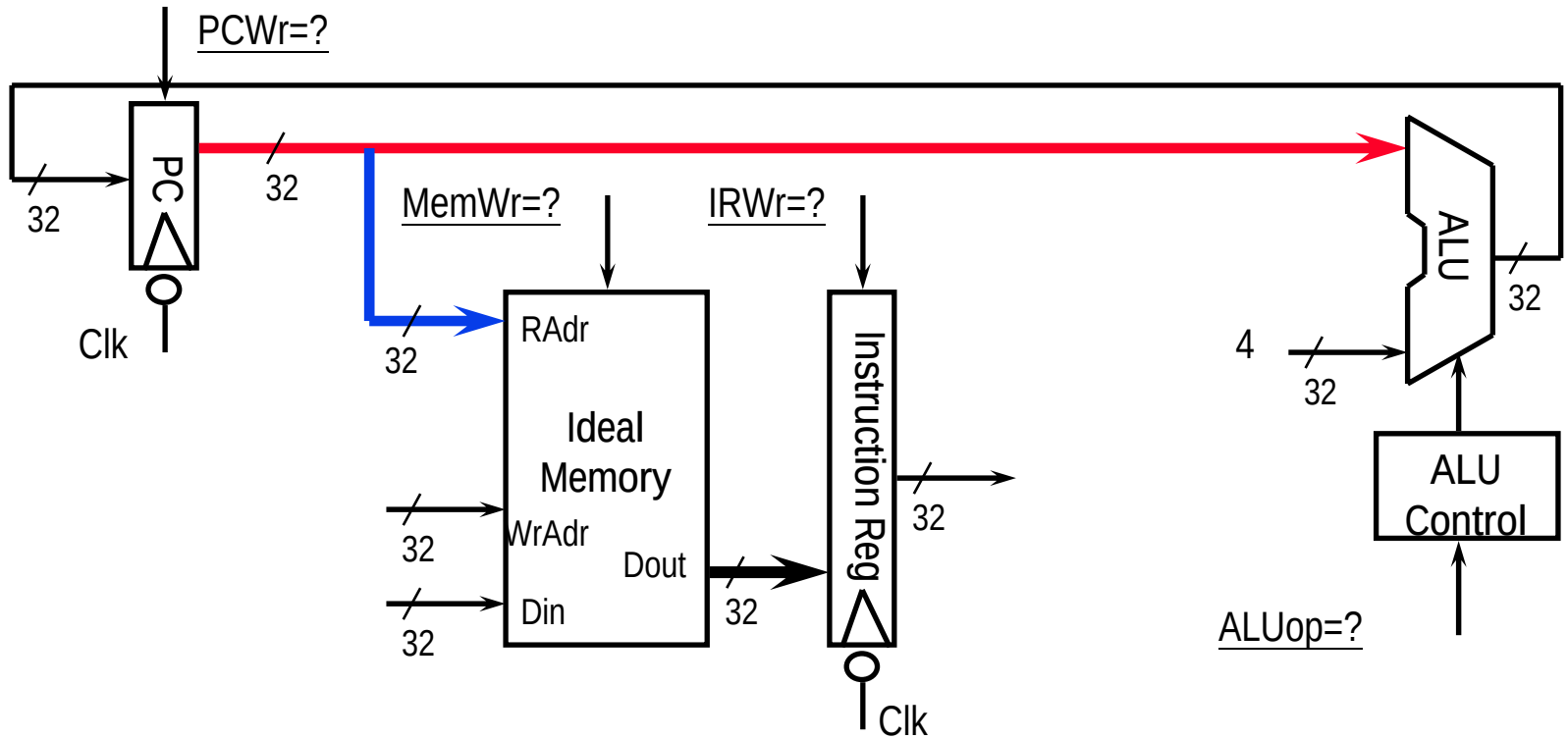
Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

1. Instruction Fetch Cycle: In the Beginning

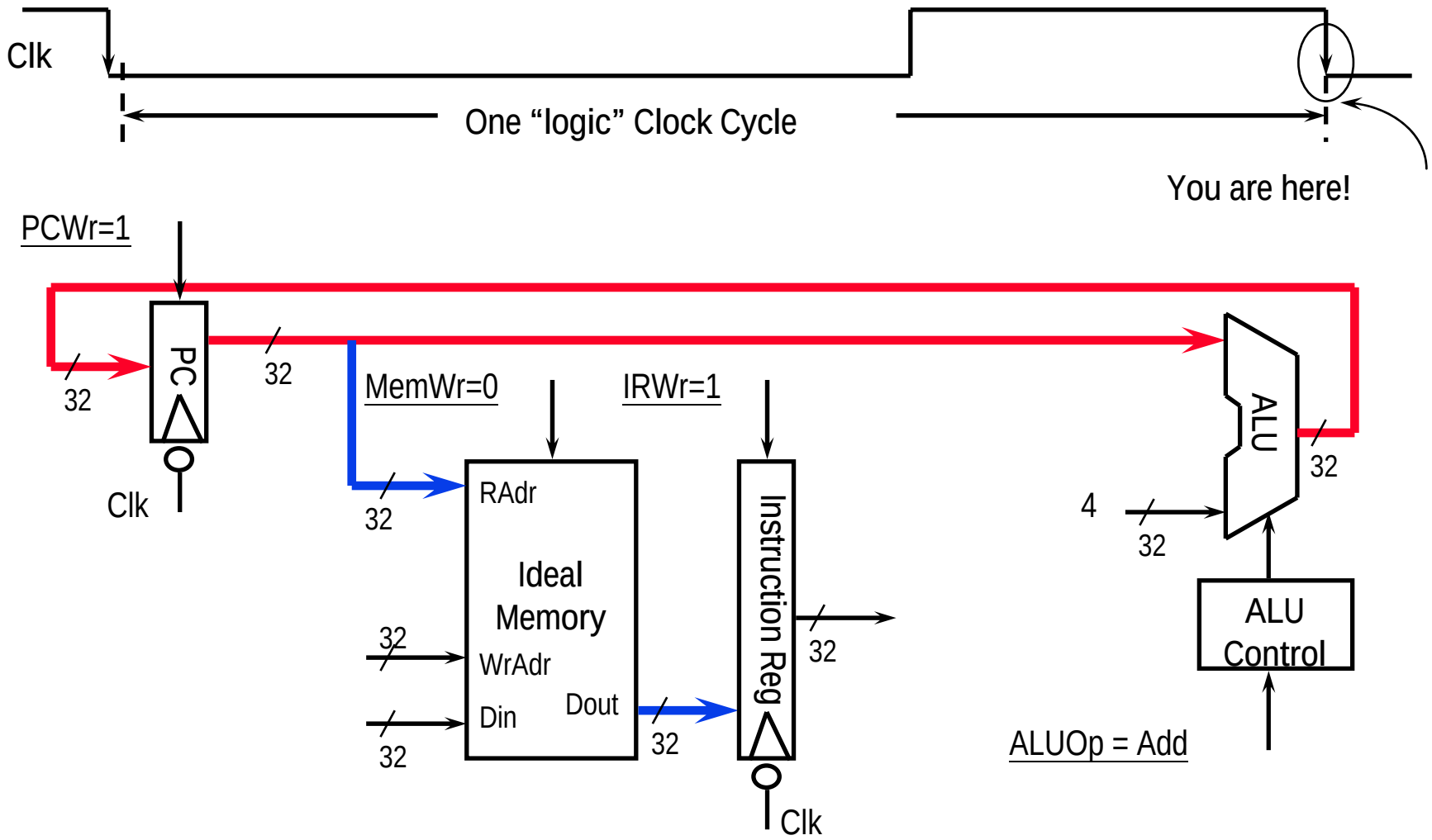
- **Every cycle begins right AFTER the clock tick:**
  - **mem[PC] PC<31:0> + 4**



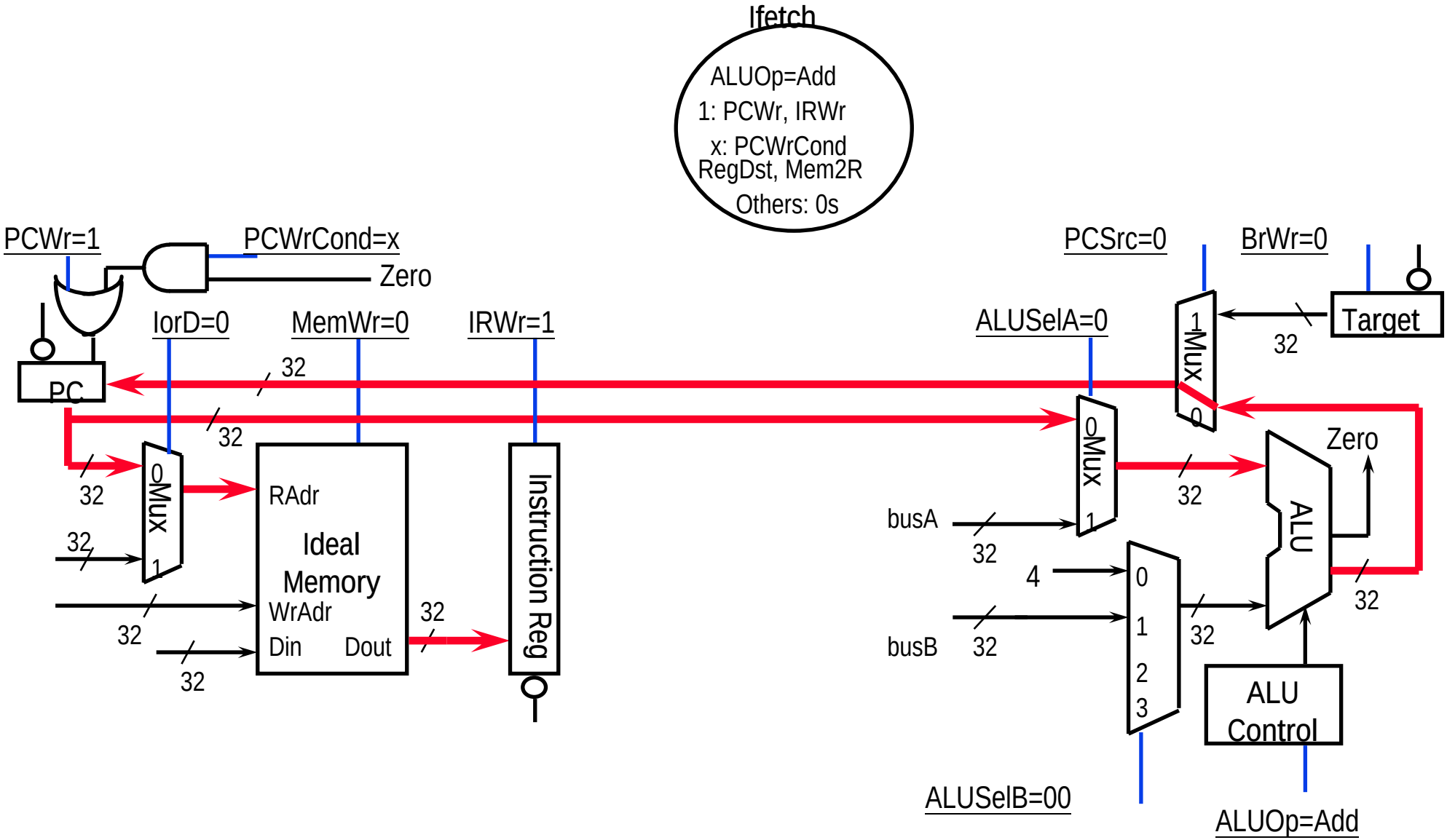


1. Instruction Fetch Cycle: The End

- **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



Step 2: Instruction Decode and Register Fetch

- **Read registers rs and rt in case we need them**
- **Compute the branch address in case the instruction is a branch**
- **RTL:**

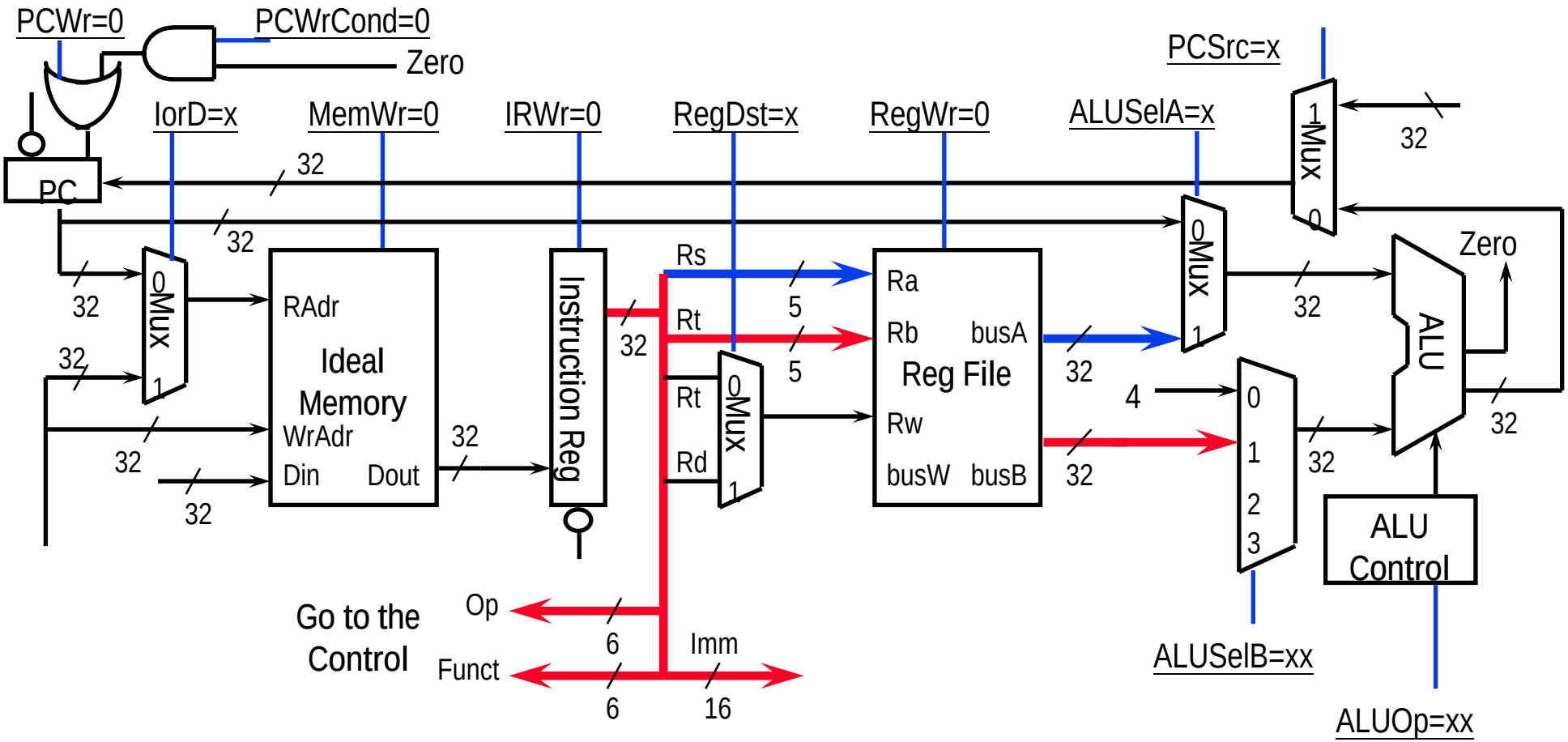
```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend(IR[15:0]) << 2);
```

- **We aren't setting any control lines based on the instruction type (we are busy "decoding" it in our control logic)**

2. Register Fetch / Instruction Decode

- busA <- RegFile[rs] ; busB <- RegFile[rt] ;

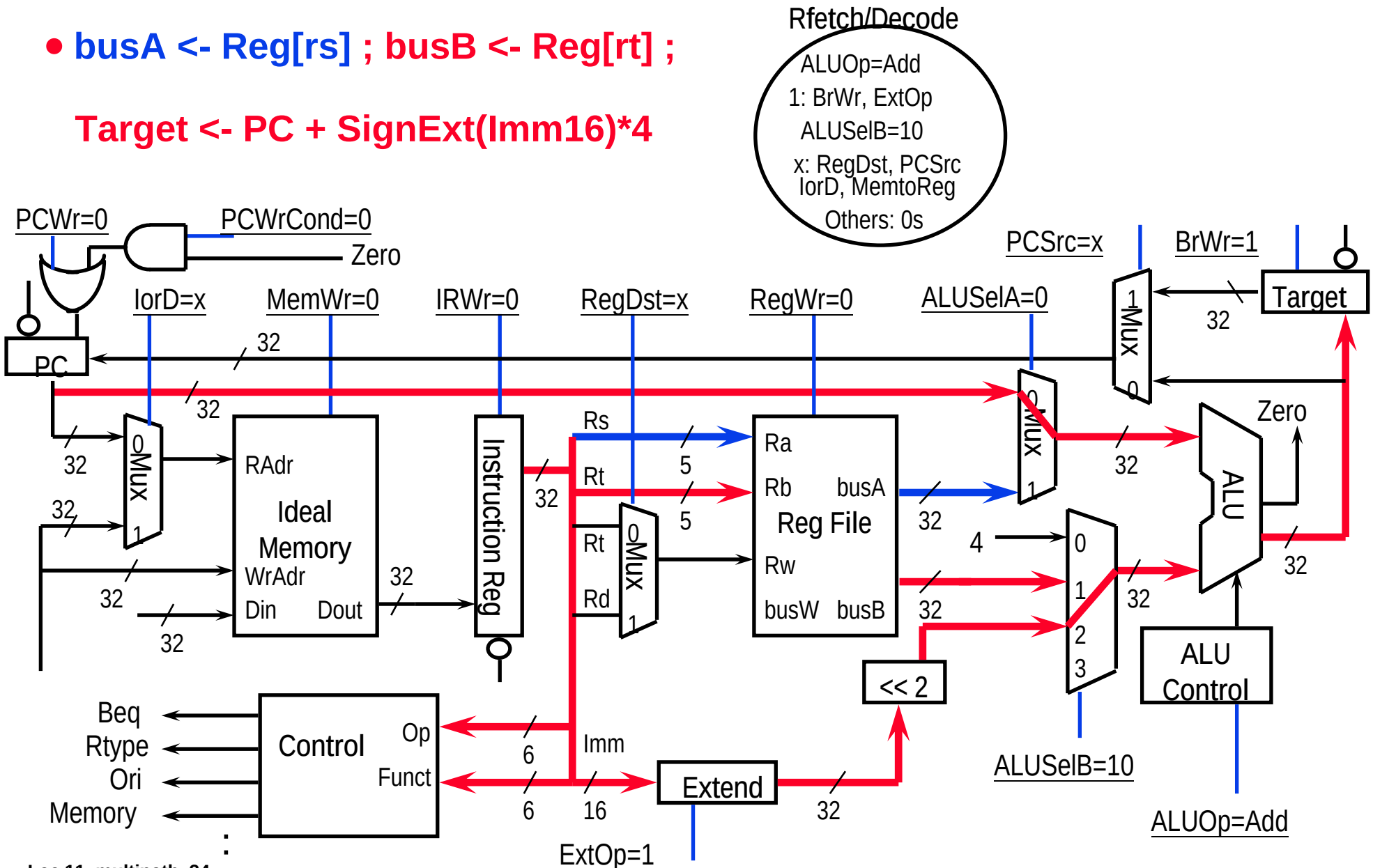
ALU is not being used: ALUctr <= xx



2. Register Fetch / Instruction Decode (Continued)

- busA <- Reg[rs] ; busB <- Reg[rt] ;

Target <- PC + SignExt(Imm16)*4



Step 3: (instruction dependent)

ALU is performing one of three functions, based on instruction type

- **Memory Reference:**

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

- **R-type:**

  ALUOut <= A op B;

- **Branch:**

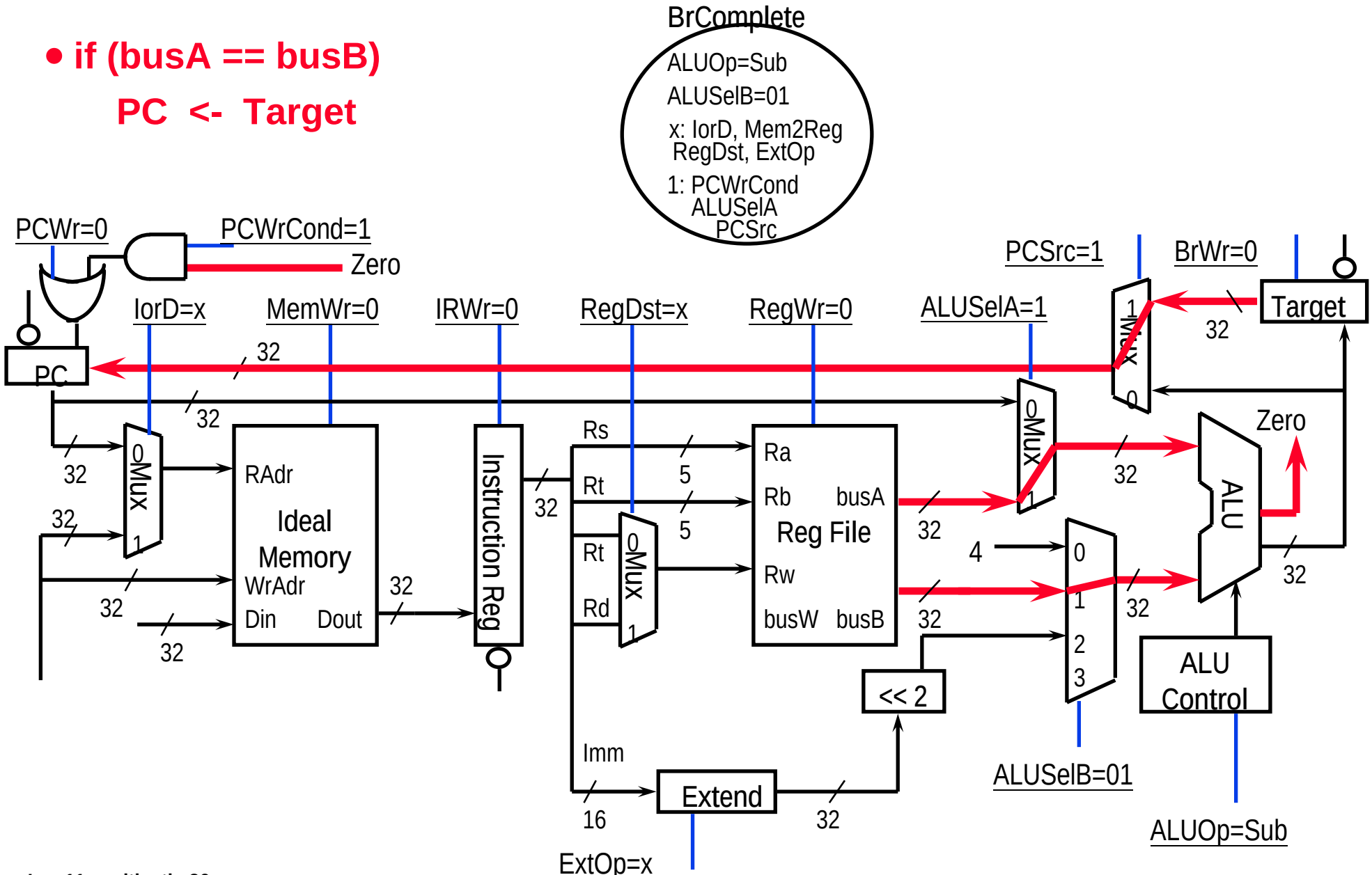
  if (A==B) PC <= ALUOut;

- **Jump (see Lec. 10-18)**

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

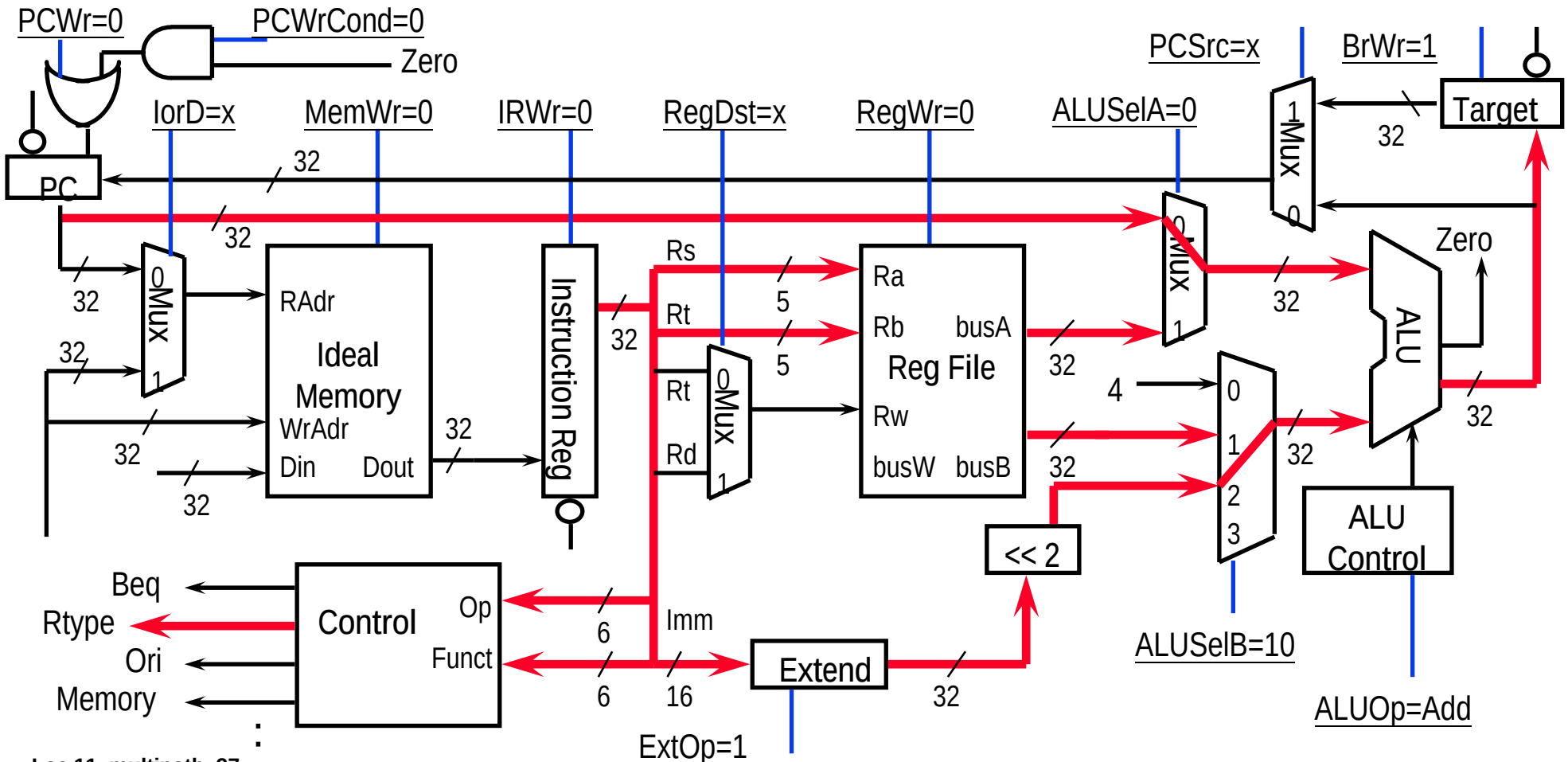
3. Branch Completion

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



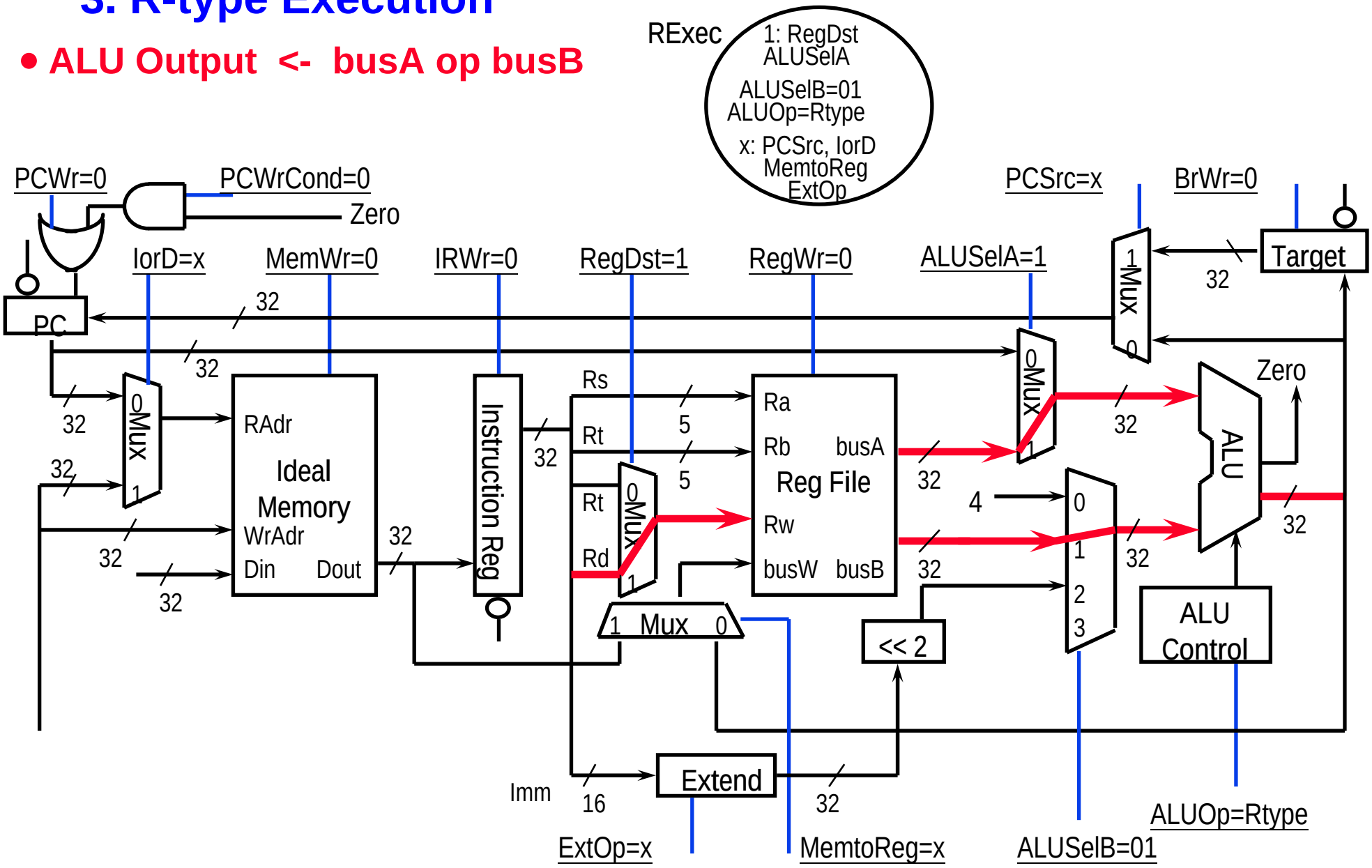
Instruction Decode: We have a R-type!

- **Next Cycle: R-type Execution**



3. R-type Execution

- **ALU Output <- busA op busB**



Step 4: R-type or memory-access

- **Loads and stores access memory**

```
MDR <= Memory[ALUOut];
        or
Memory[ALUOut] <= B;
```

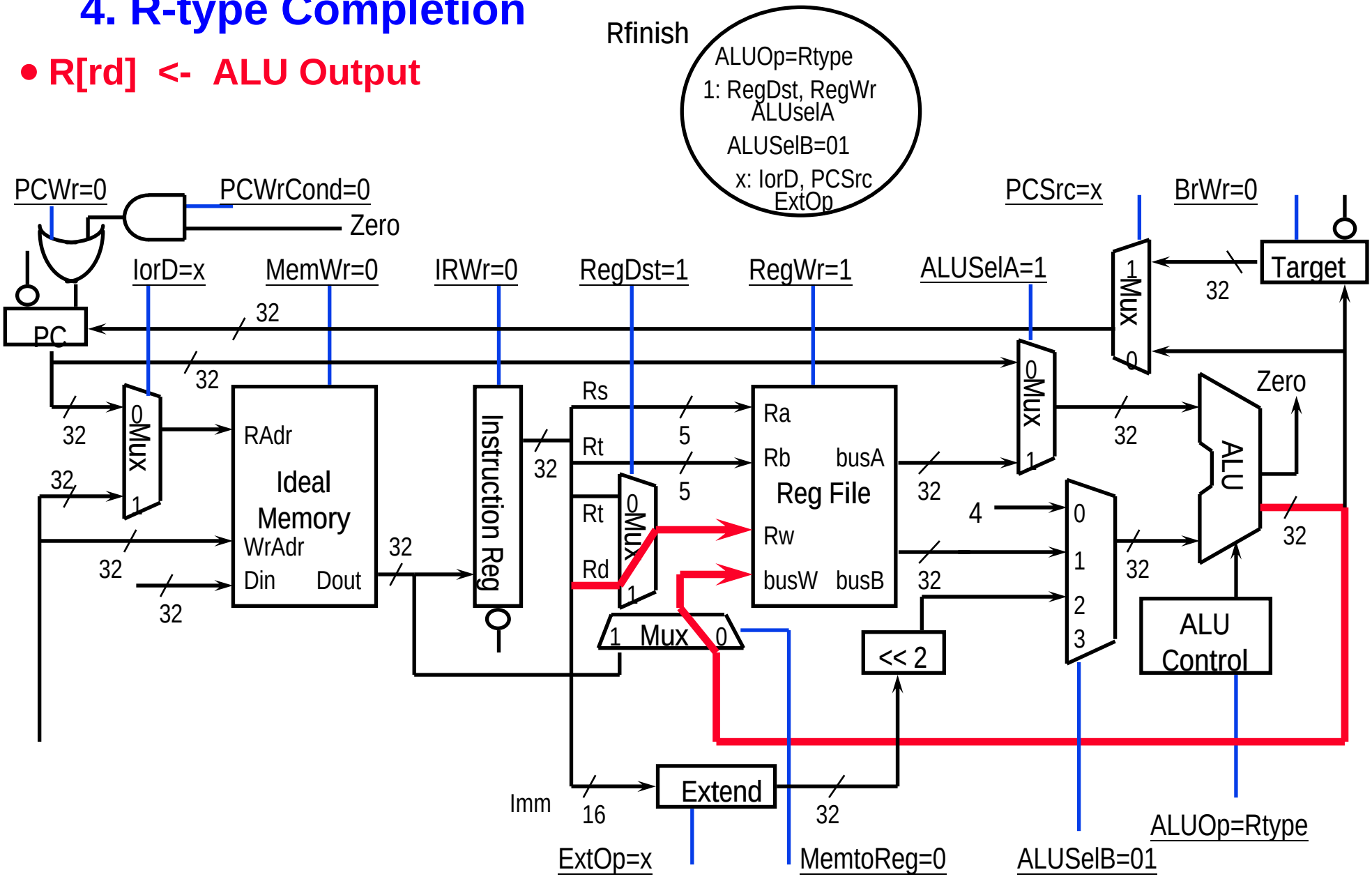
- **R-type instructions finish**

```
Reg[IR[15:11]] <= ALUOut;
```

The write actually takes place at the end of the cycle on the edge

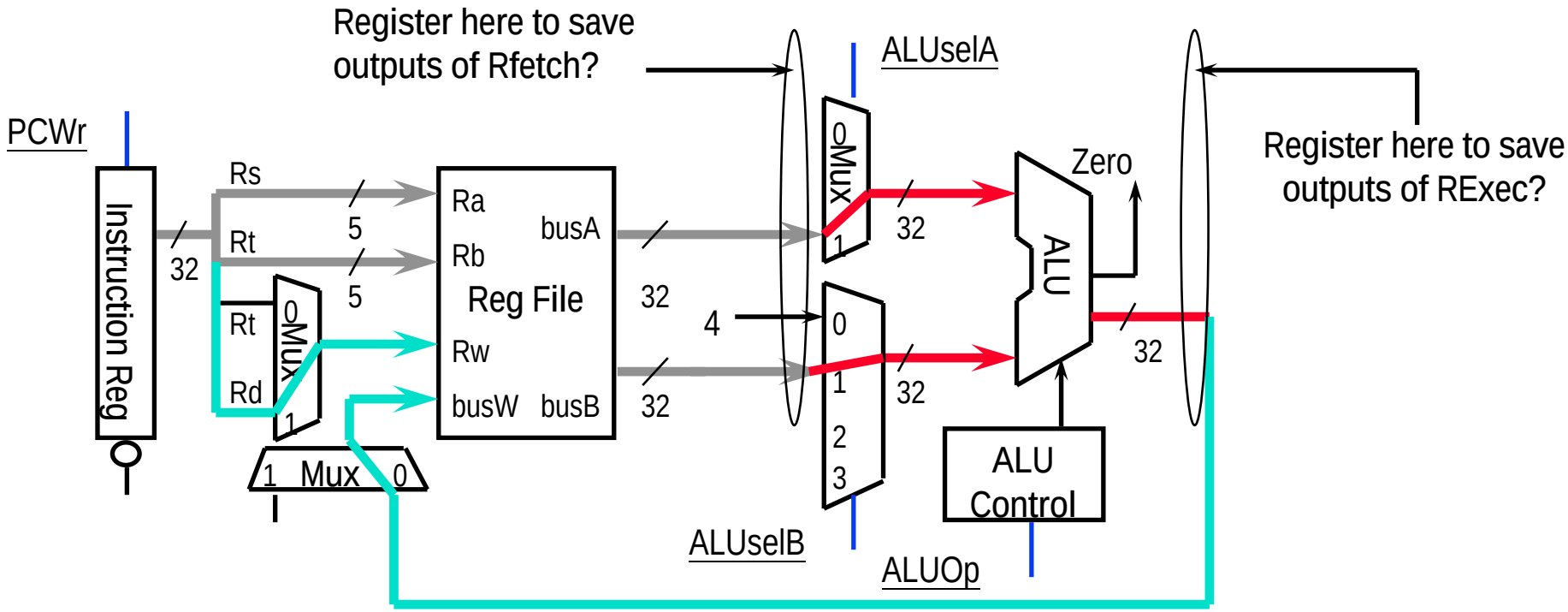
4. R-type Completion

- R[rd] <- ALU Output



A Multiple Cycle Delay Path

- There is no register to save the results between:
  - Register Fetch: busA <- Reg[rs] ; busB <- Reg[rt]
  - R-type Execution: ALU output <- busA op busB
  - R-type Completion: Reg[rd] <- ALU output

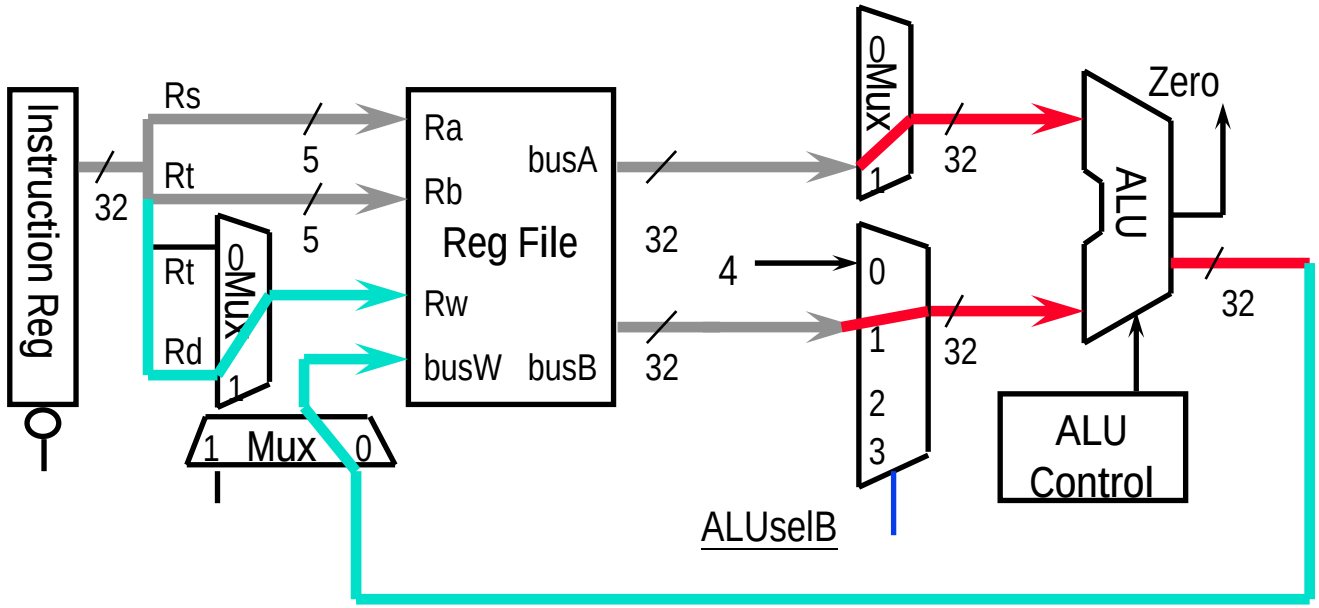


A Multiple Cycle Delay Path (Continued)

- 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, you need a register to hold a signal value if:
  - Cond.(1) The signal is computed in one clock cycle and used in another.
  - Cond.(2) AND the inputs to the functional block that computes this signal can change before the signal is written into a state element.
- You can save a register if Cond.(1) is true BUT Cond.(2) is false:
  - But in practice, this will introduce a multiple cycle delay path:
    - ⇒ A logic delay path that takes multiple cycles to propagate from one storage element to the next storage element

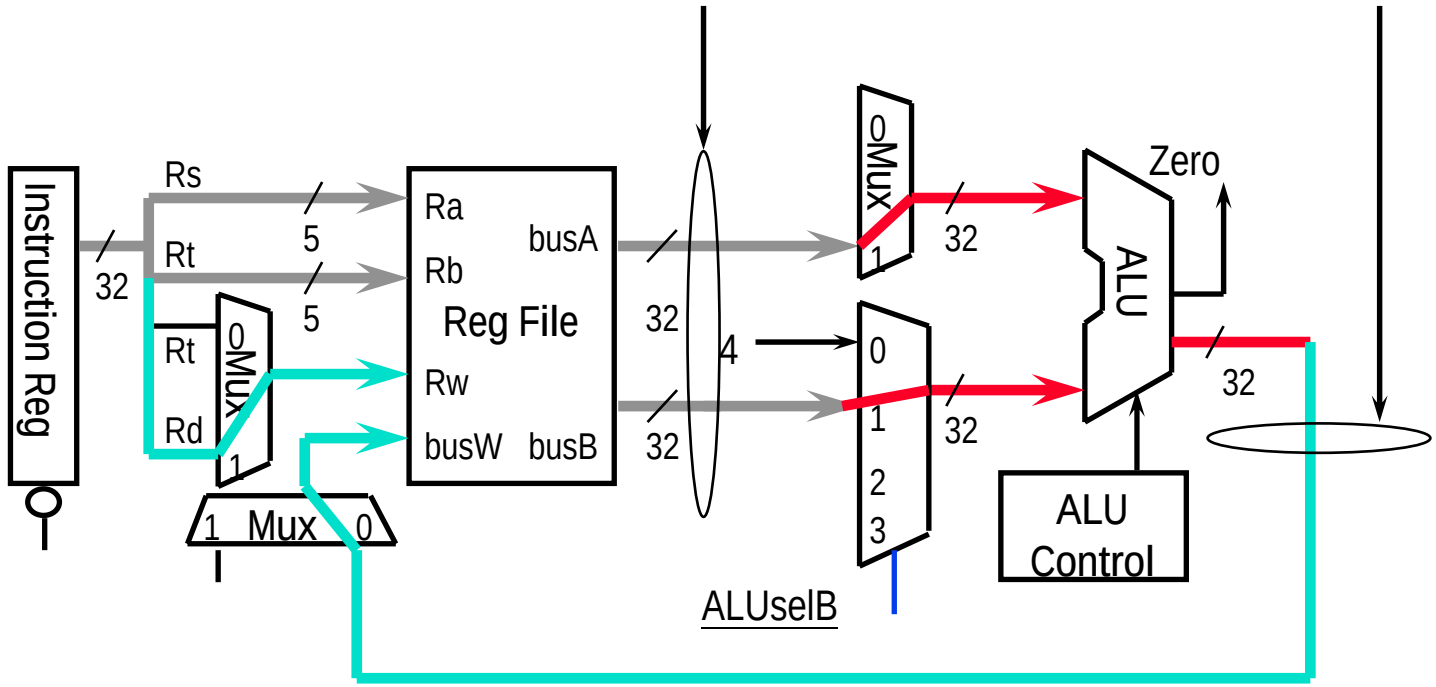
Pros and Cons of a Multiple Cycle Delay Path

- **A 3-cycle path example:**
  - **IR (storage) -> Reg File Read -> ALU -> Reg File Write (storage)**
- **Advantages:**
  - **Register savings**
  - **We can share time among cycles:**
    - ⇒ **If ALU takes longer than one cycle, still "a OK" as long as the entire path takes less than 3 cycles to finish**



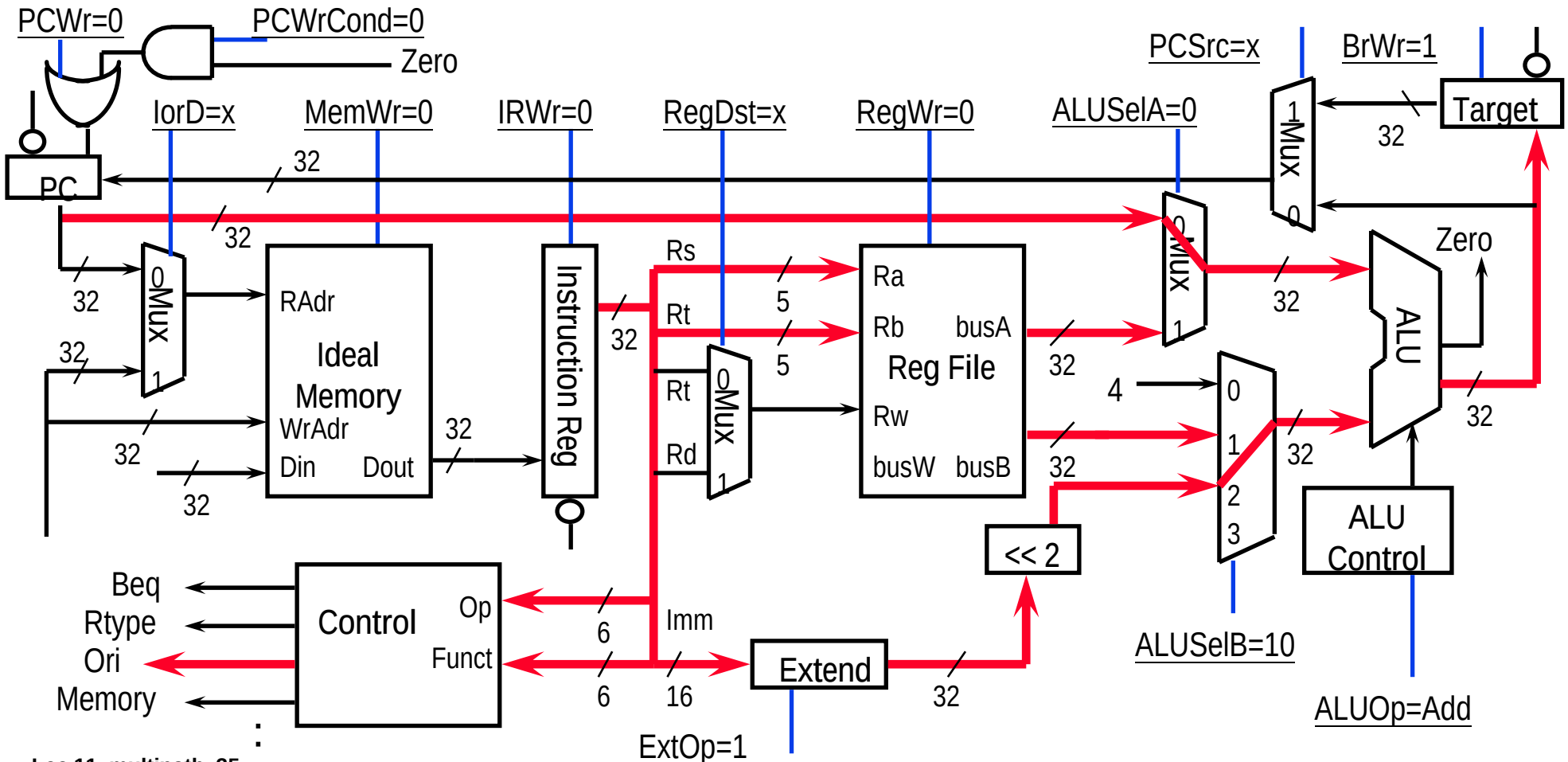
Pros and Cons of a Multiple Cycle Delay Path (Continued)

- **Disadvantage:**
  - **Static timing analyzer, which ONLY looks at delay between two storage elements, will report this as a timing violation**
  - **You have to ignore the static timing analyzer's warnings**
  - **But you may end up ignoring real timing violations**
  - **I always TRY to put in registers between cycles to avoid MCDP**



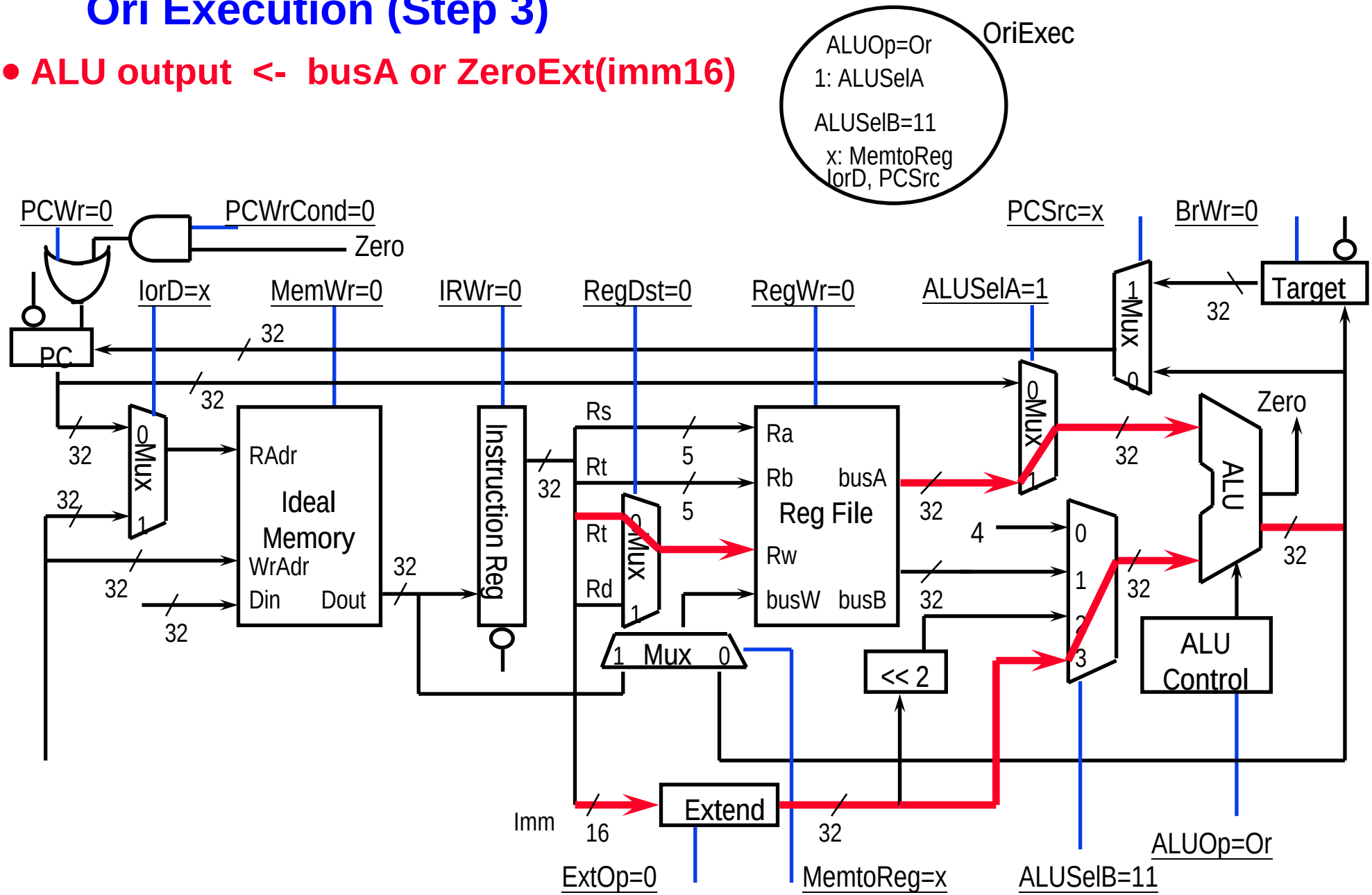
Instruction Decode: We have an Ori!

- **Next Cycle: Ori Execution**



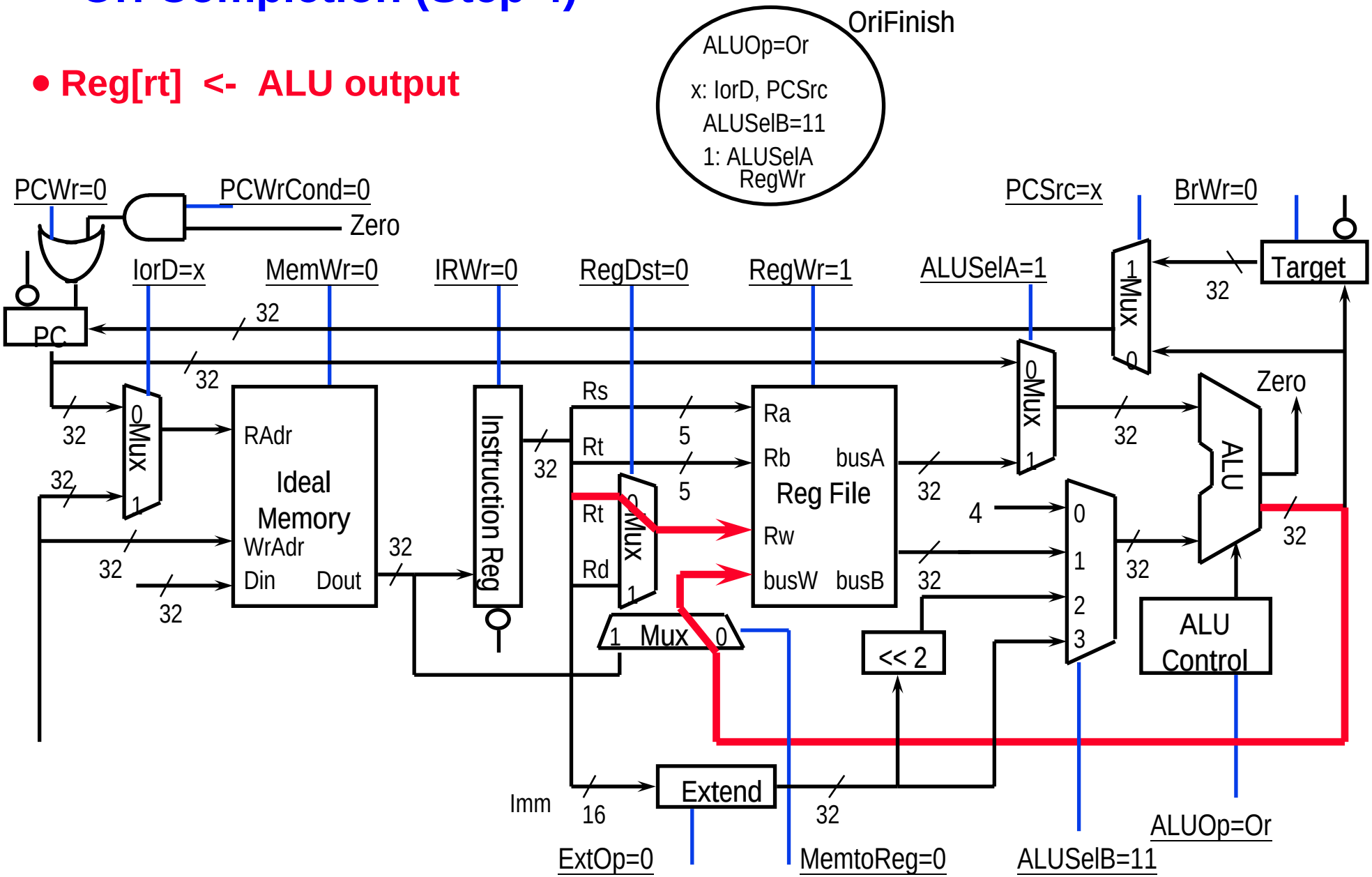
Ori Execution (Step 3)

- **ALU output <- busA or ZeroExt(imm16)**



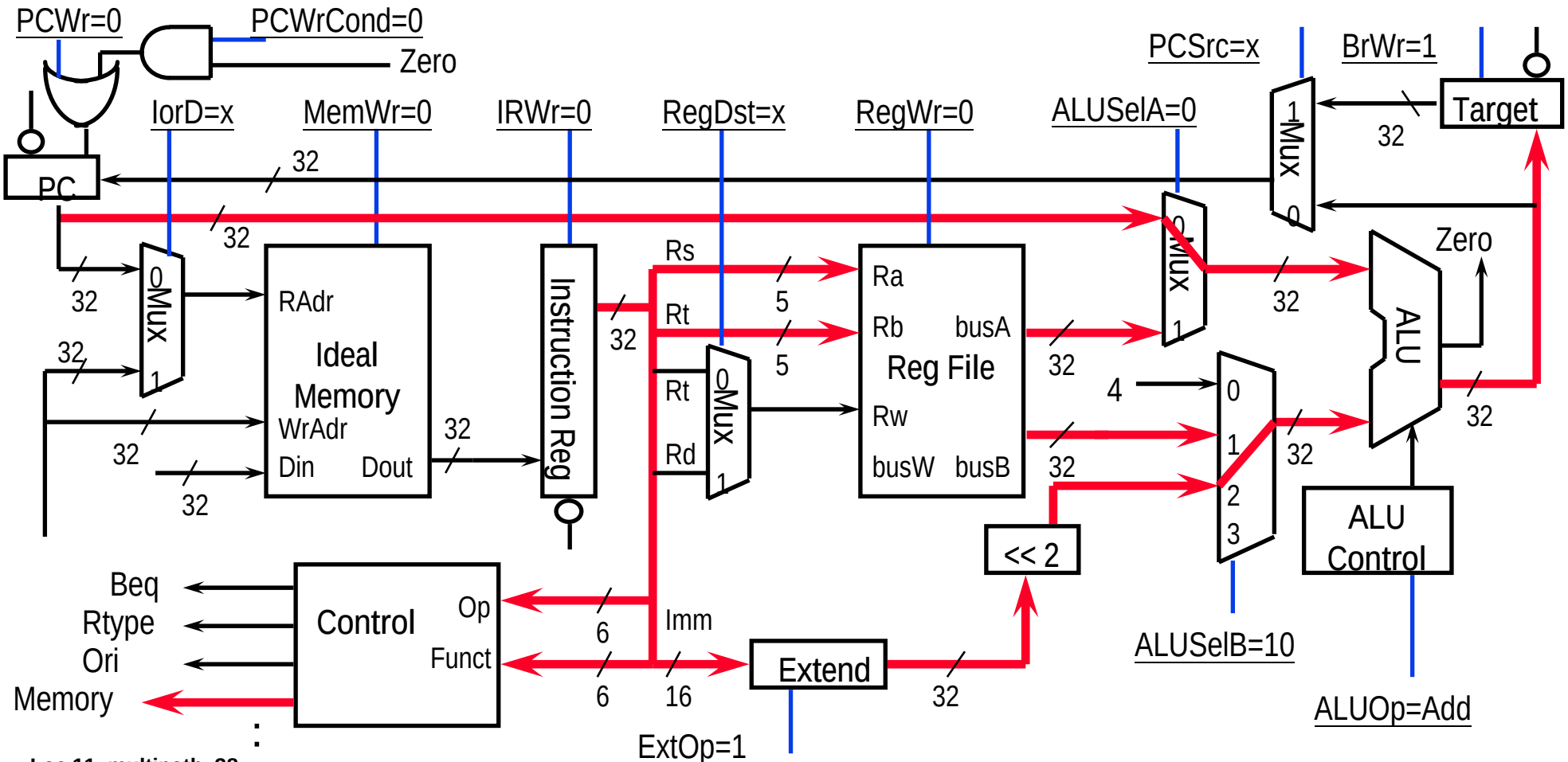
Ori Completion (Step 4)

- **Reg[rt] <- ALU output**



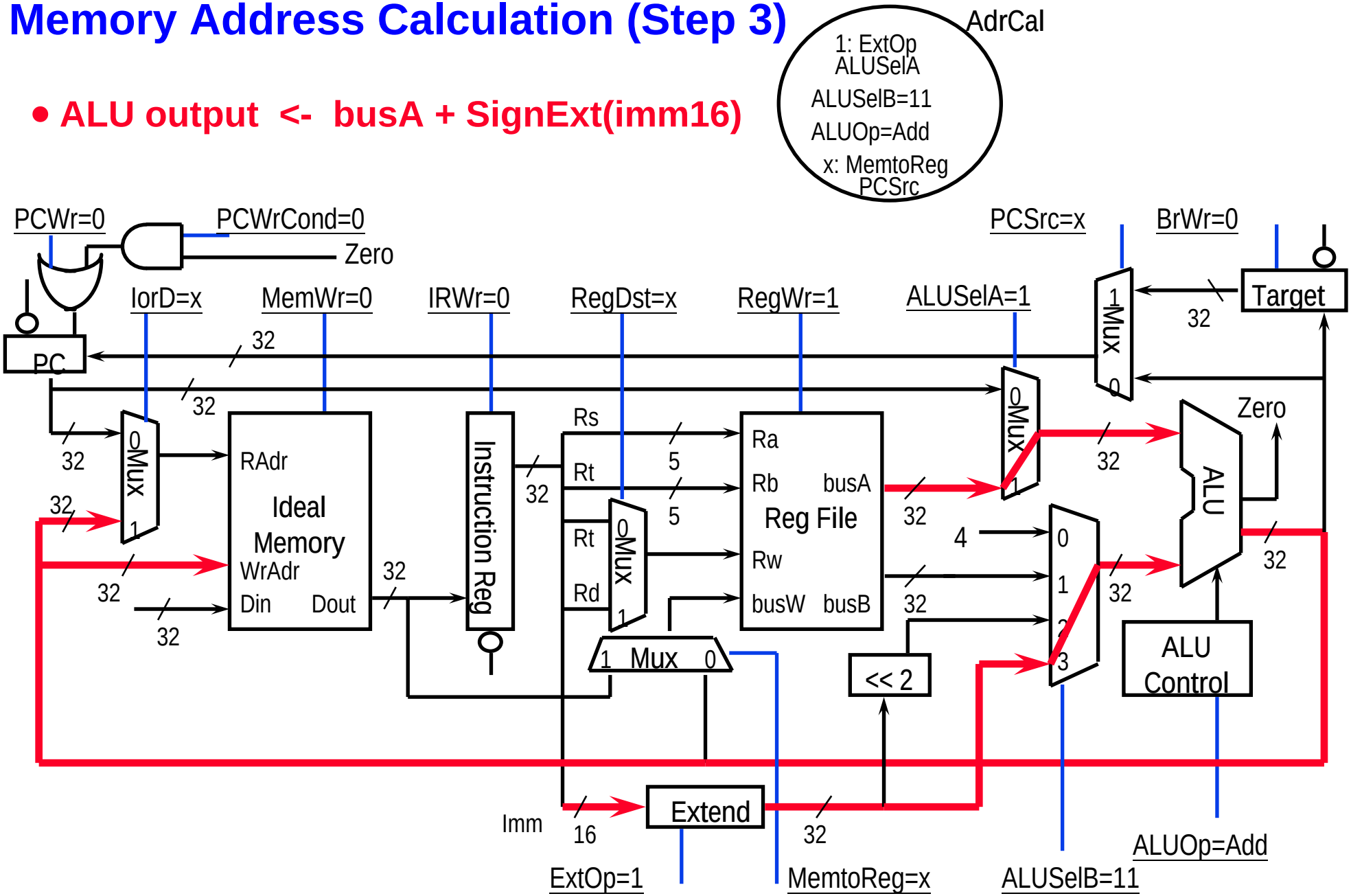
Instruction Decode: We have a Memory Access!

- **Next Cycle:** Memory Address Calculation



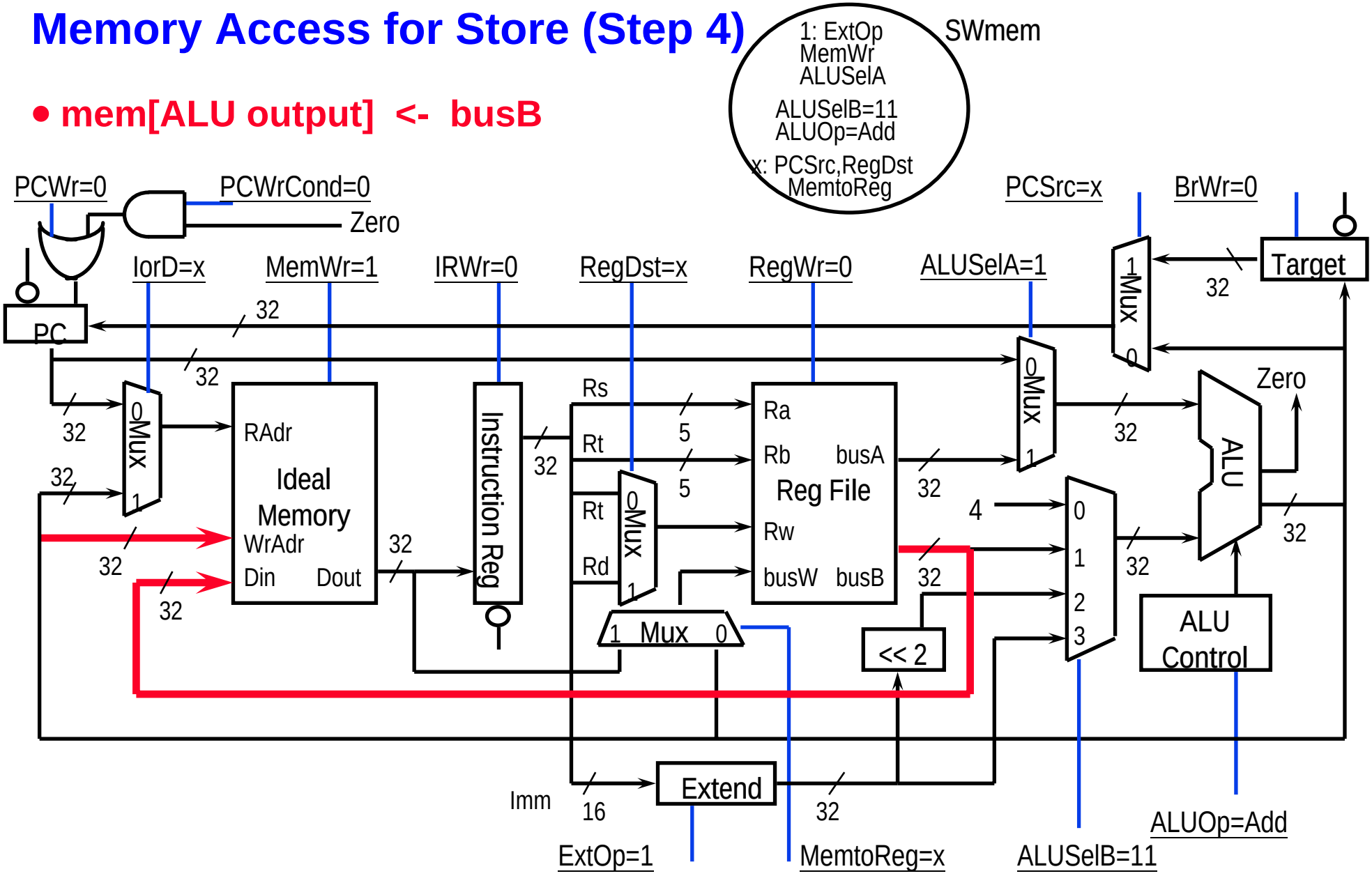
Memory Address Calculation (Step 3)

- **ALU output <- busA + SignExt(imm16)**



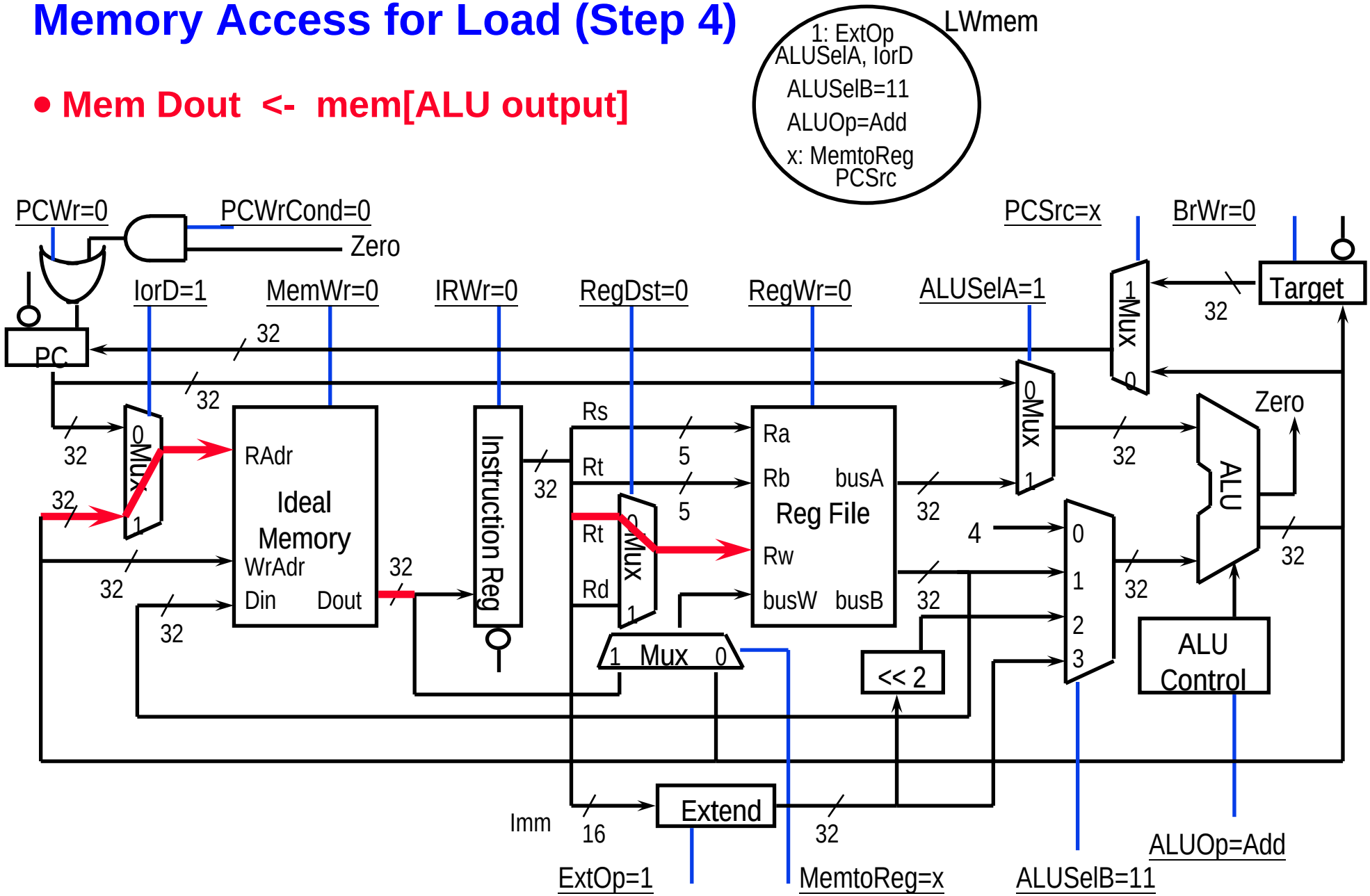
Memory Access for Store (Step 4)

- mem[ALU output] <- busB



Memory Access for Load (Step 4)

- Mem Dout <- mem[ALU output]



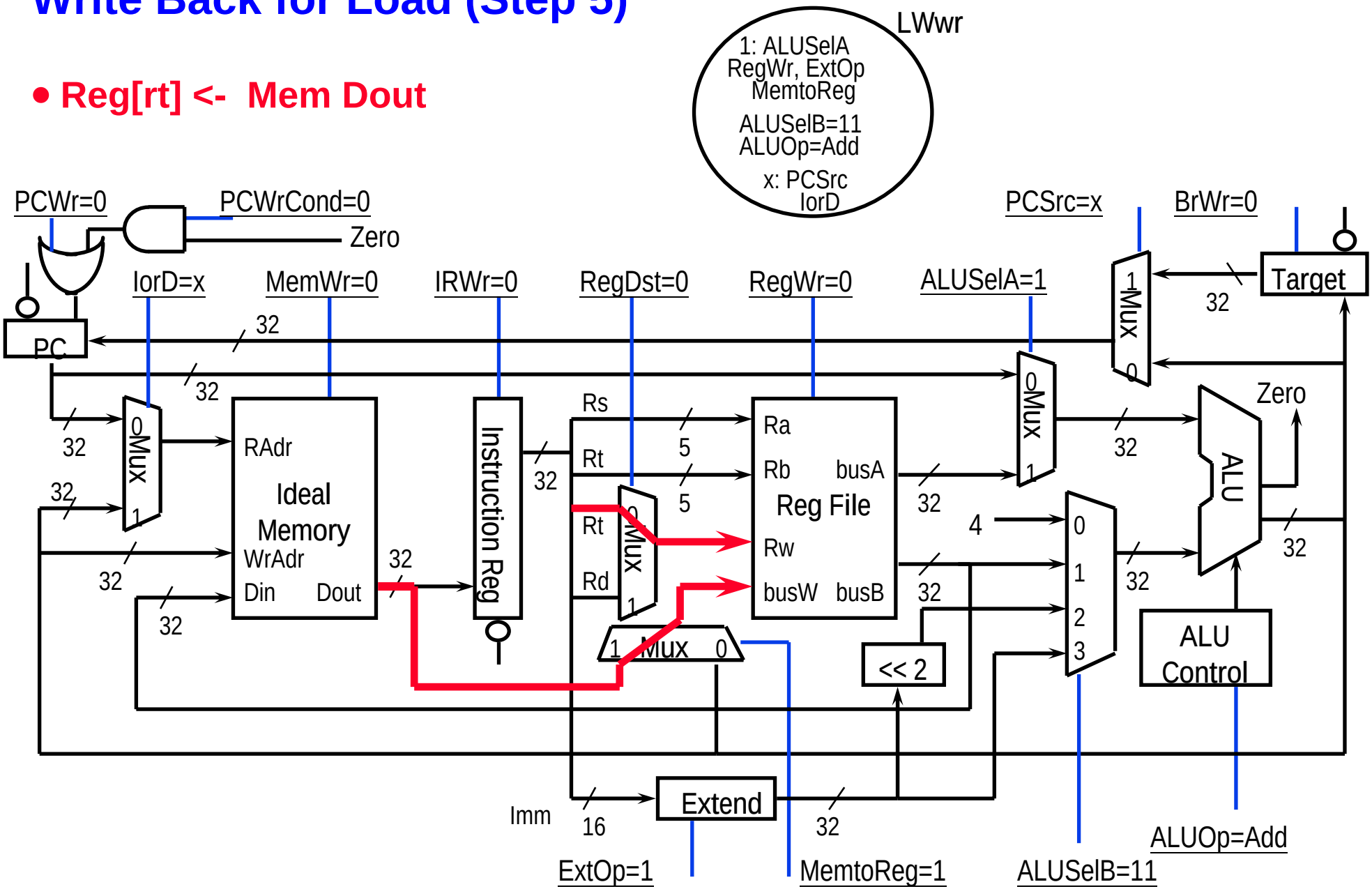
Step 5: Write-back for Load

- Reg[IR[20:16]]<= MDR;

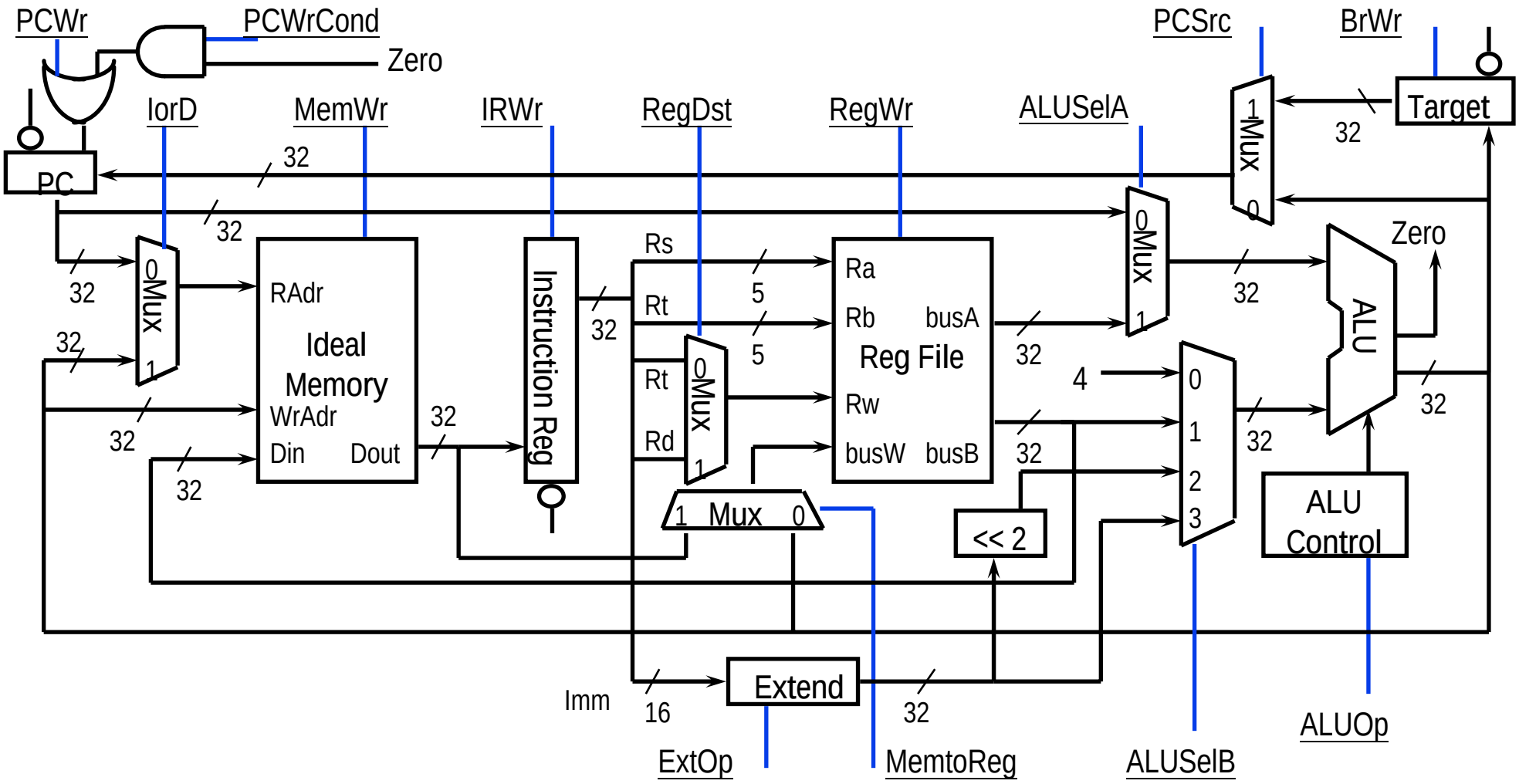
Which instruction needs this?

Write Back for Load (Step 5)

- **Reg[rt] <- Mem Dout**



Putting it all together: Multiple Cycle Datapath



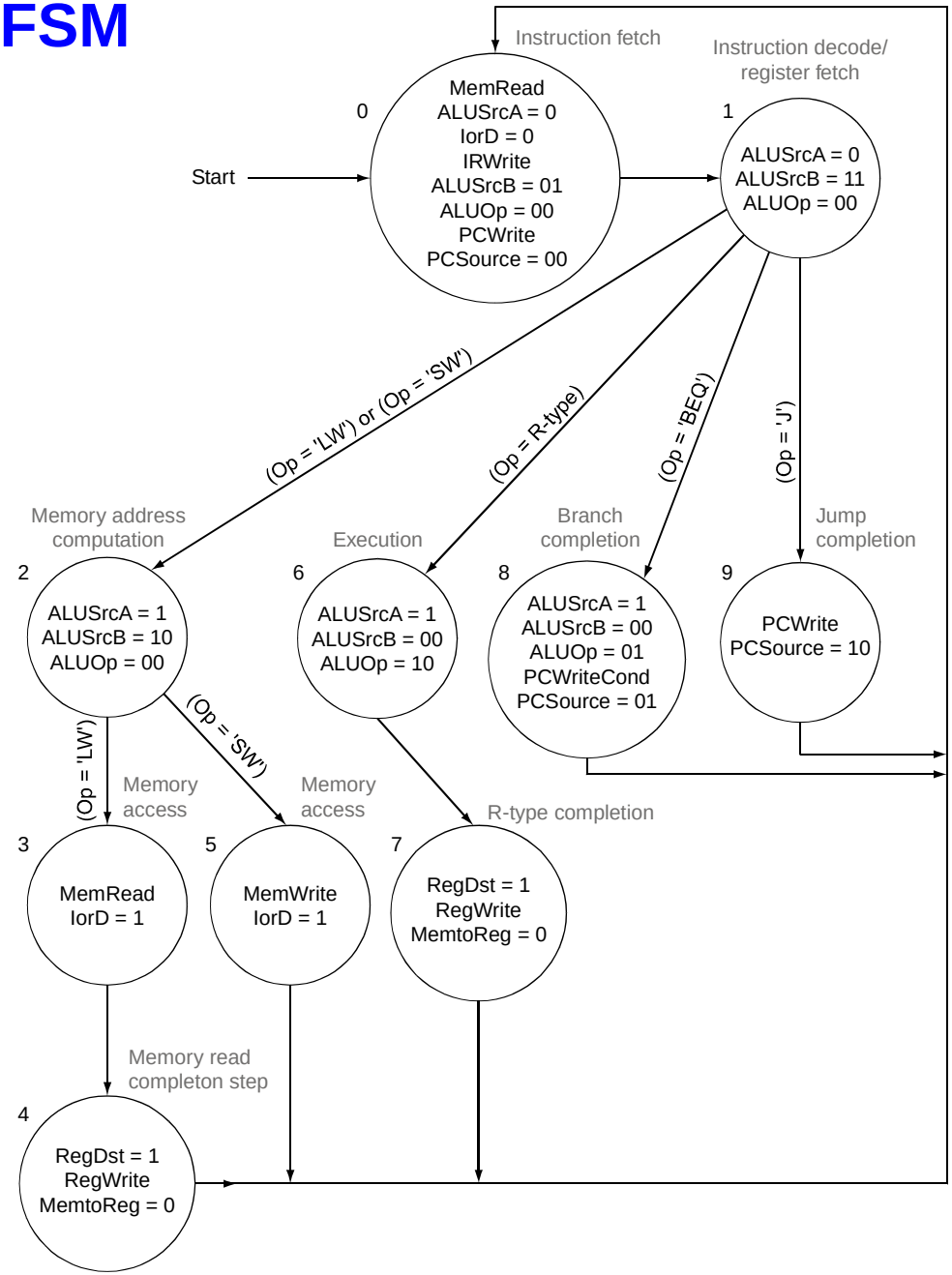
Summary:

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

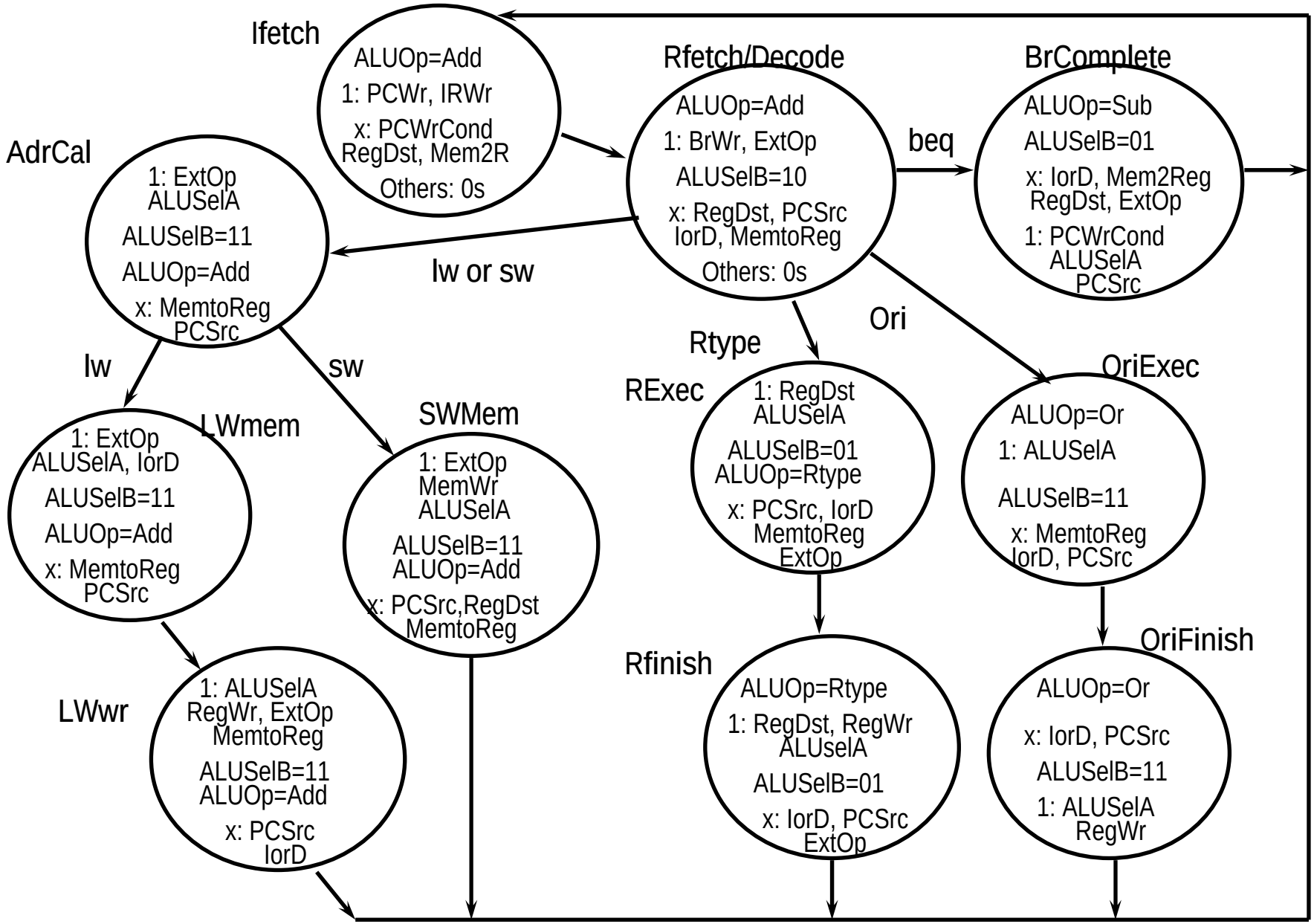
Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we've accumulated to specify a finite state machine (FSM)
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived from specification

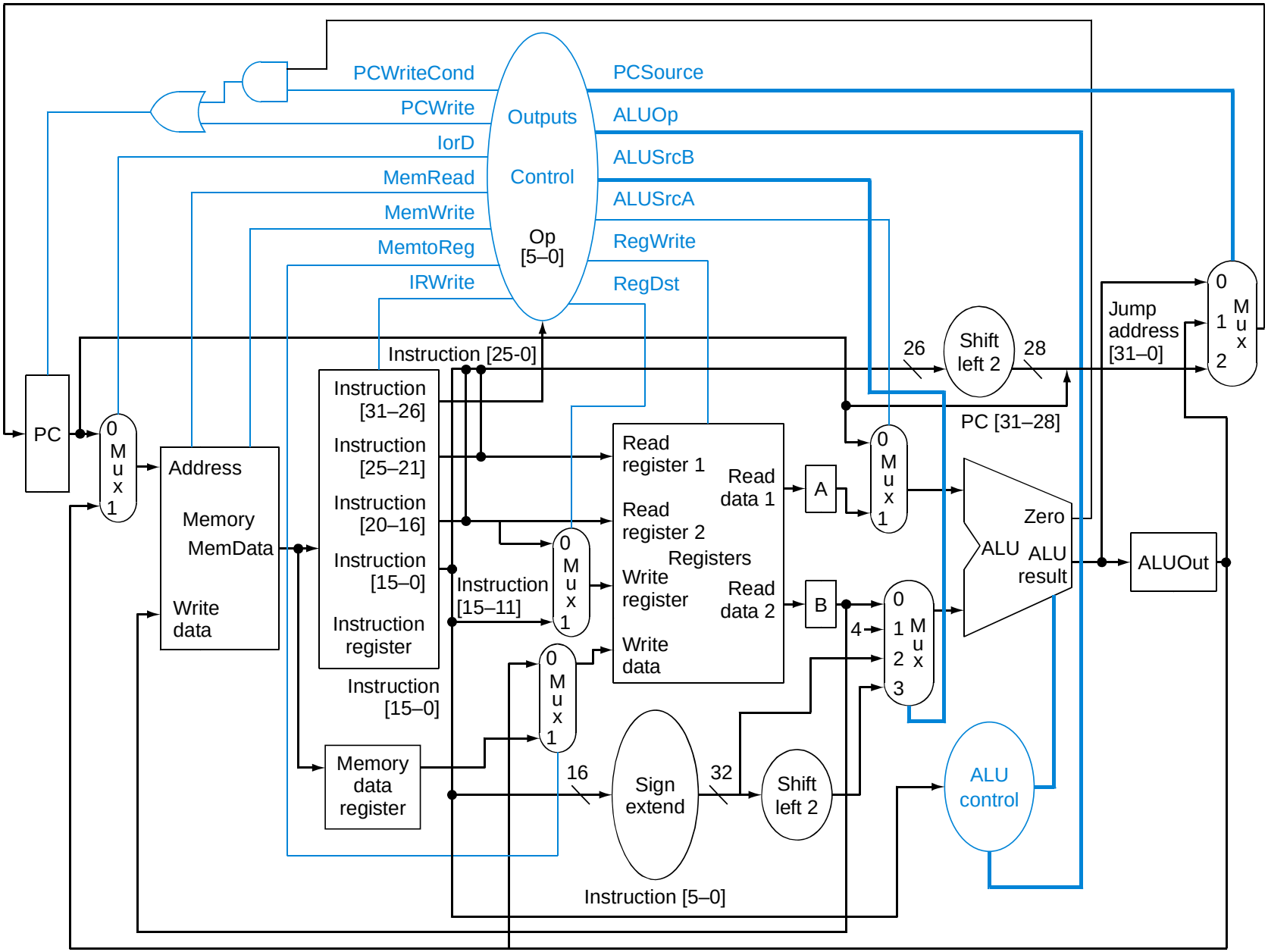
Graphical Specification of FSM



Putting it all together: Control State Diagram



The Final Control Path and Data Path



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

Where to get more information?

- **Next two lectures:**
  - **Multiple Cycle Controller: Appendix C of your text book.**
  - **Microprogramming: Section 5.7 (in CD) of your text book.**
- **D. Patterson, "Microprogramming," *Scientific America*, March 1983.**
- **D. Patterson and D. Ditzel, "The Case for the Reduced Instruction Set Computer," *Computer Architecture News* 8, 6 (October 15, 1980)**