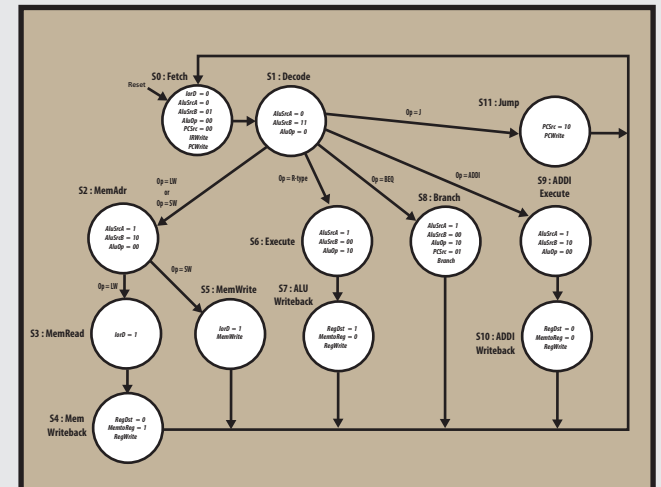


Computer Architecture

Paul Mellies

Lecture 15 : Multicycle microarchitectures



The multicycle microarchitecture

The multicycle microarchitecture

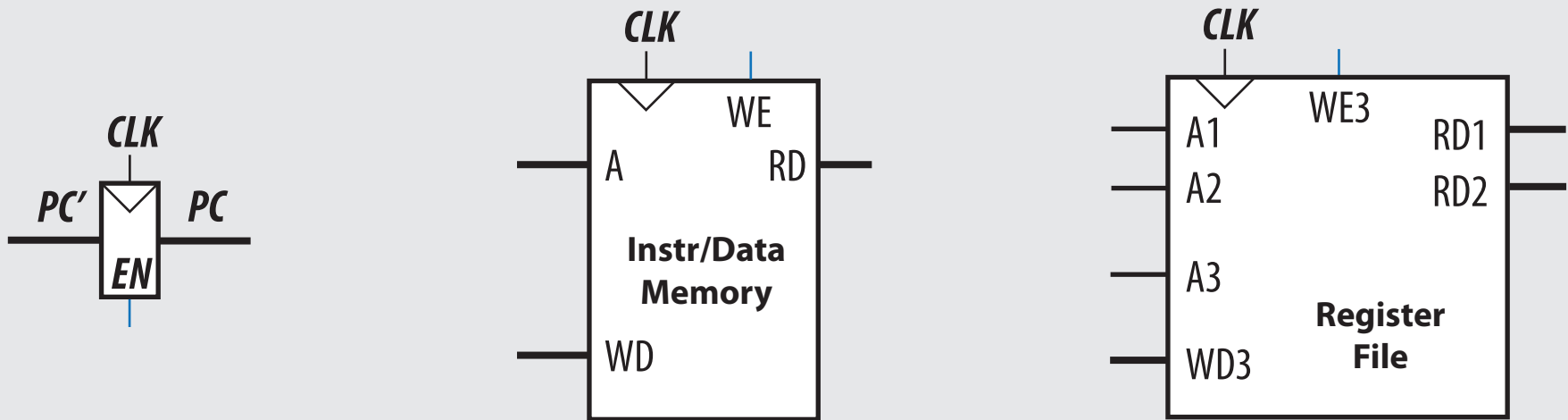
The single-cycle microarchitecture has three weaknesses :

- it requires a clock cycle long enough to support the slowest instruction `lw` even though most instructions are faster.
- it requires three adders: one in the ALU and two for the PC logic ; Adders are relatively expensive circuits, especially if they must be fast.
- it has separate instruction and data memories, which may not be realistic.

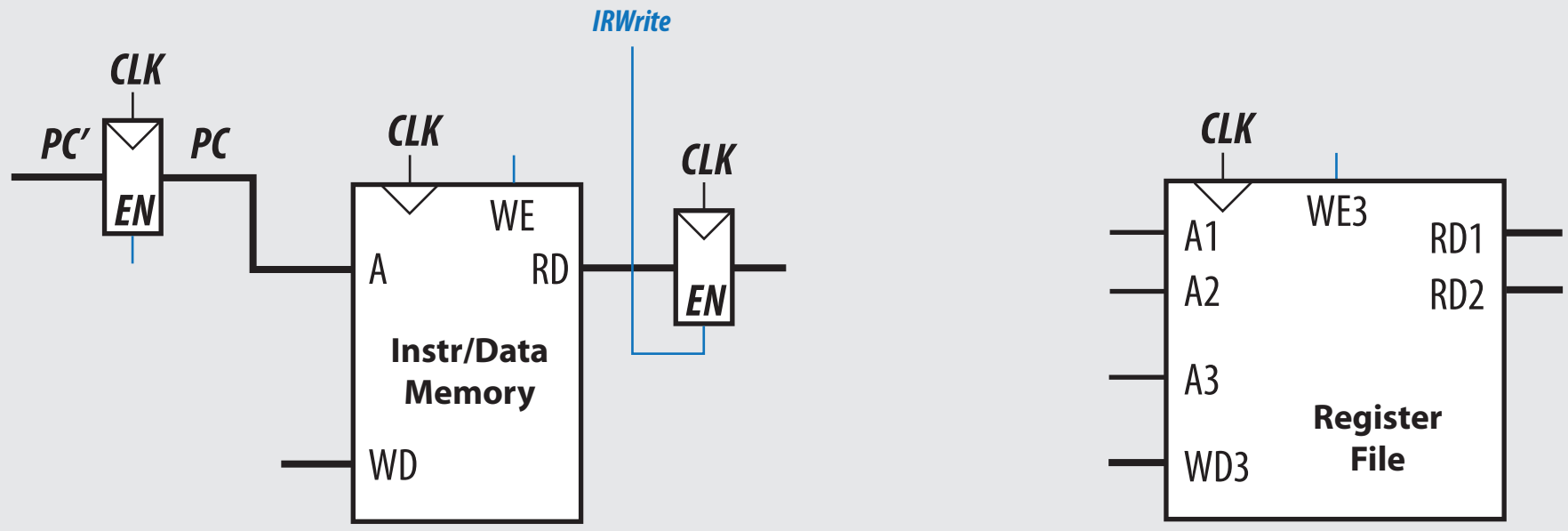
The multicycle processor addresses these three weaknesses by breaking an instruction into multiple shorter steps.

The key idea is to replace the controller of the single-cycle microarchitecture which is based on combinational logic by a finite state machine (FSM).

State elements with unified instruction/data memory

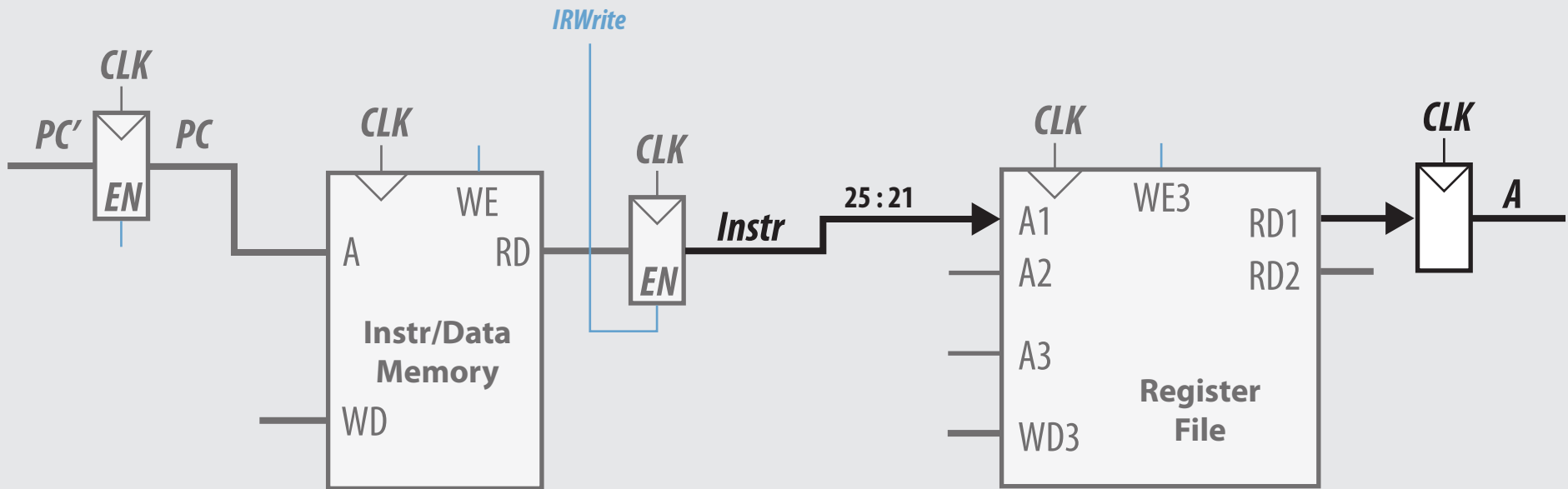


Step 1 : fetch instruction from memory



The instruction is read and stored in a new nonarchitectural Instruction Register so that it is available for future cycles. The Instruction Register receives an enable signal called *IRWrite* which is asserted when the Instruction Register should be updated with a new instruction.

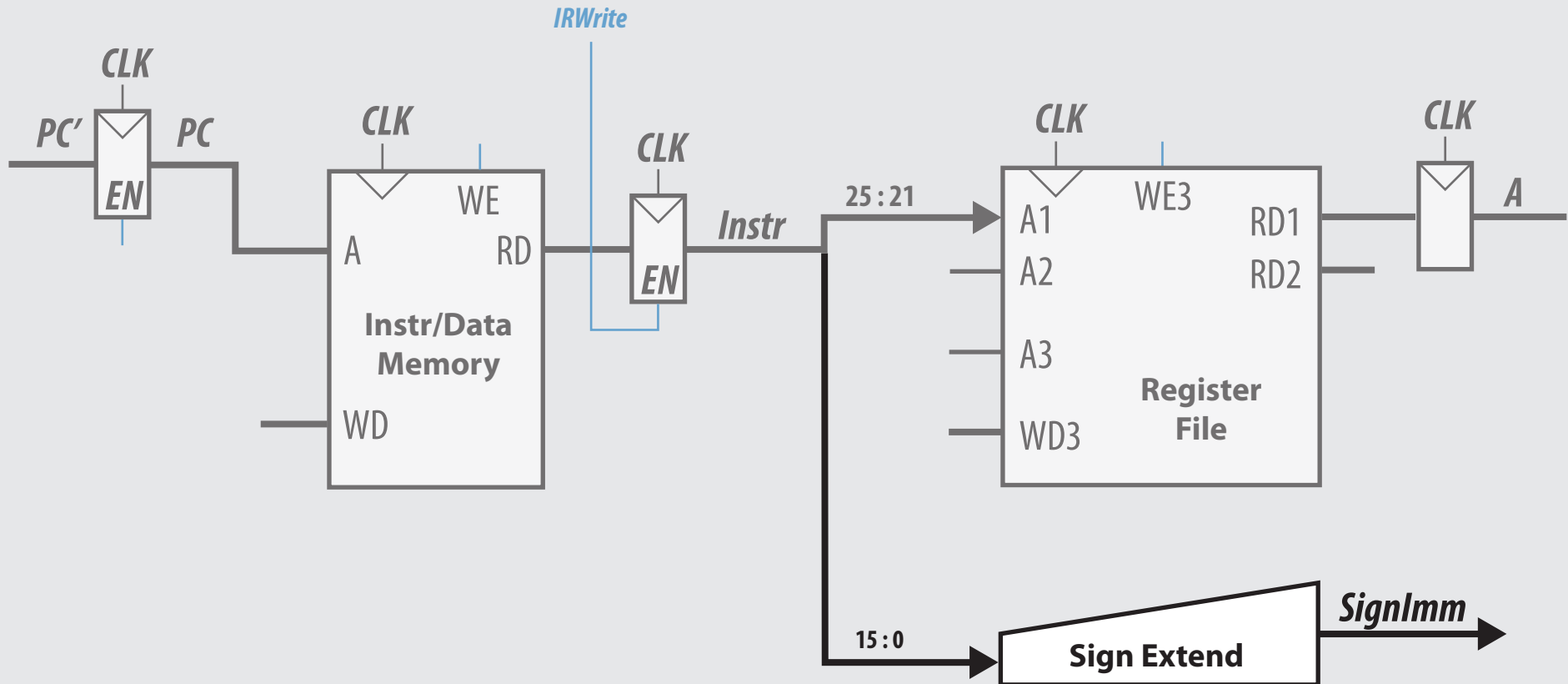
Step 2 : fetch instruction from memory



The next step is to read the source register containing the base address.

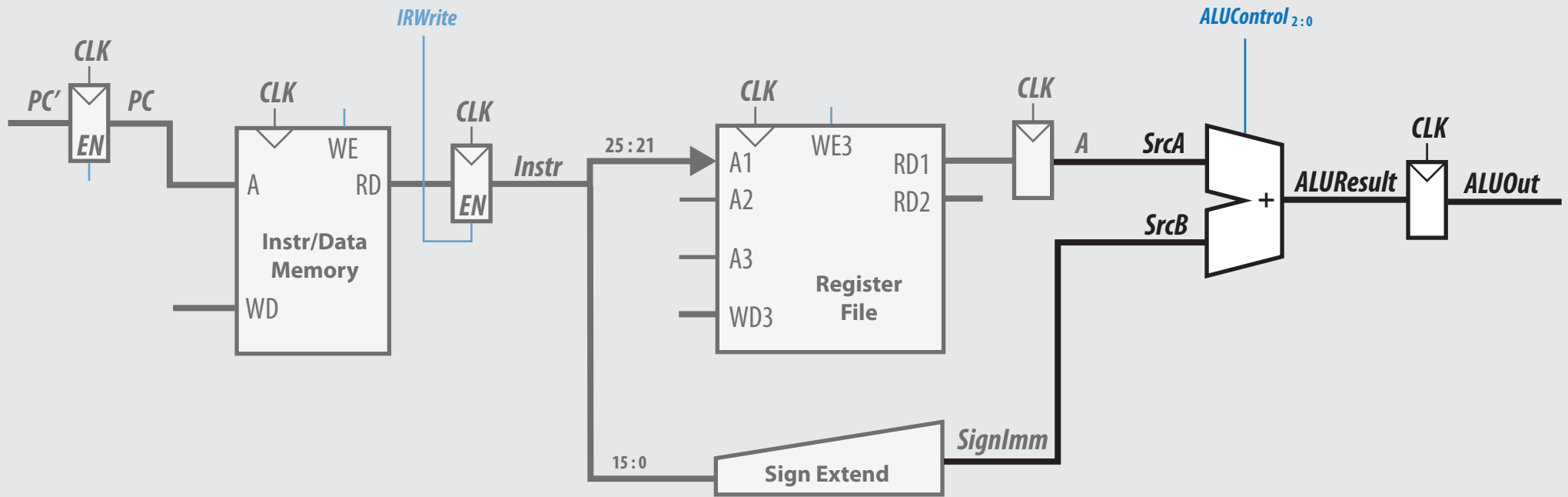
- The source register is specified in the `sr` field [25:21] of the instruction. So, just as in the single-cycle case, the 5 bits of the instruction are connected to the address input **A1** of the register file.

Step 3 : sign-extend the immediate

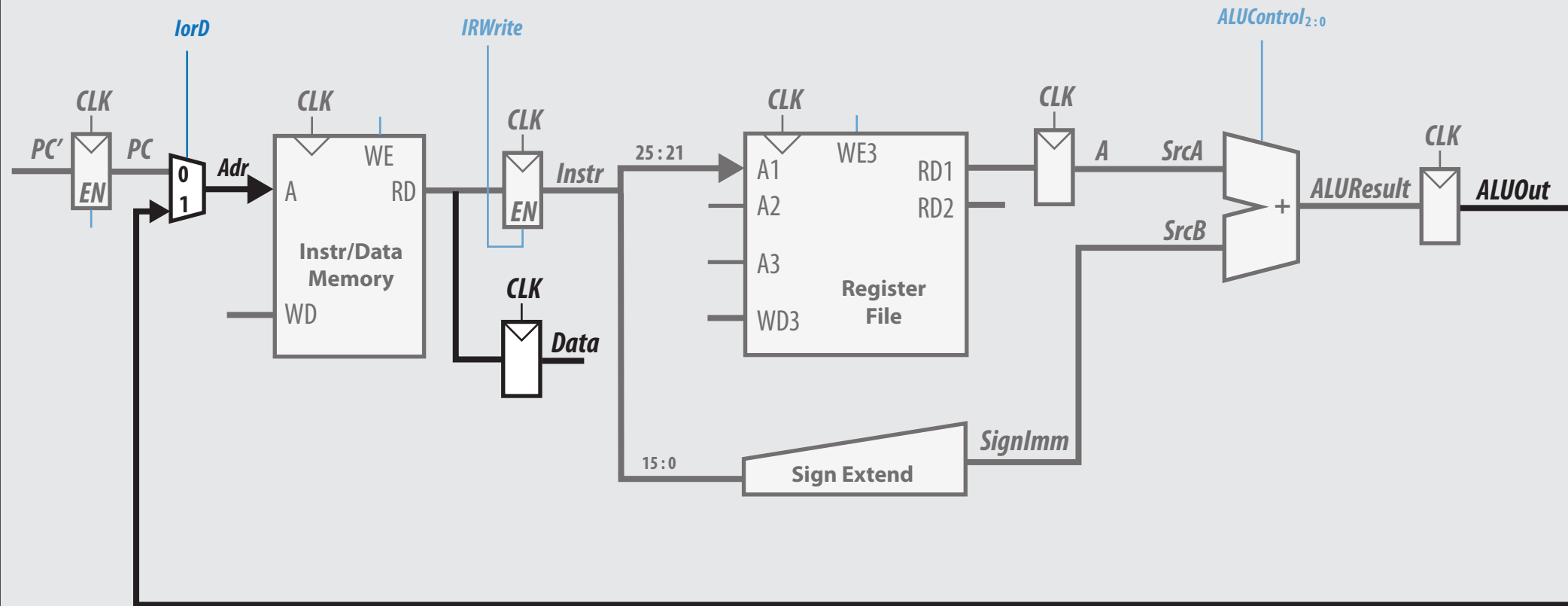


The `lw` instruction also requires an offset. The offset is stored in the immediate field of the instruction, `Instr` 15:0. Because the 16-bit immediate might be either positive or negative, it must be sign-extended to 32 bits. The 32-bit sign-extended value is called `SignImm`.

Step 4 : add base address to offset

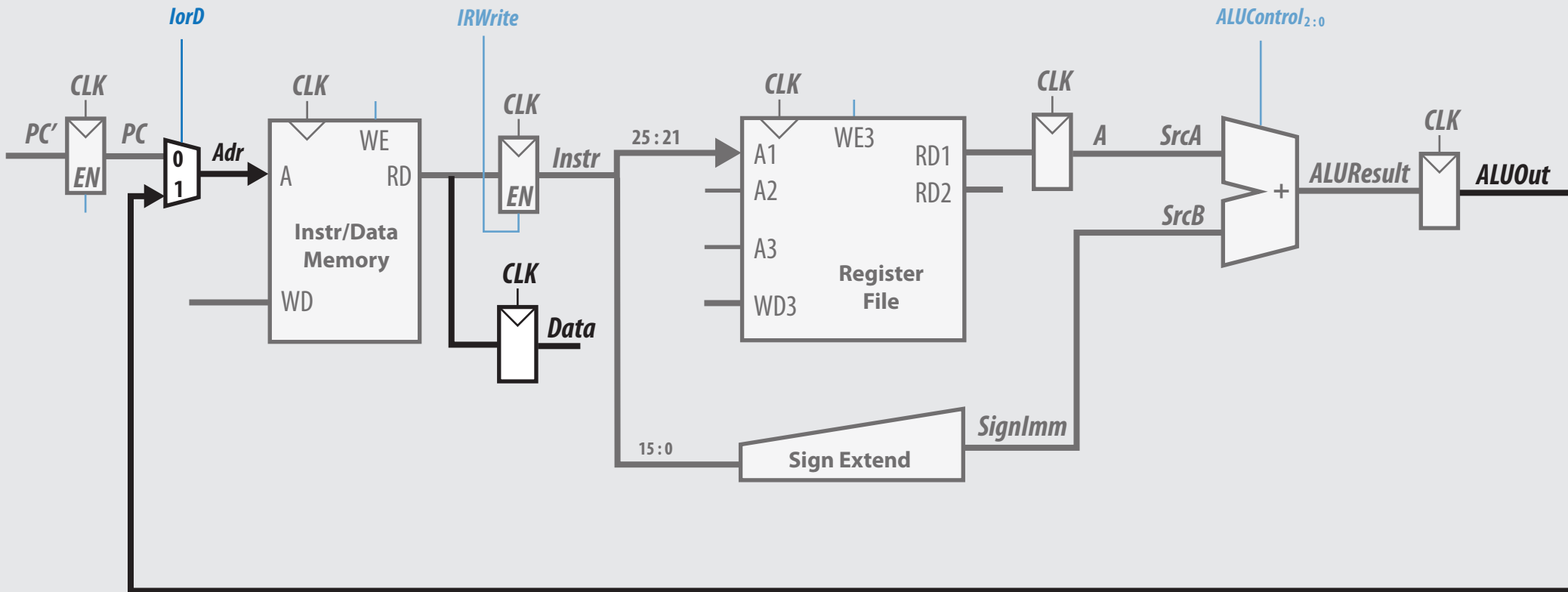


Step 6 : load data from memory



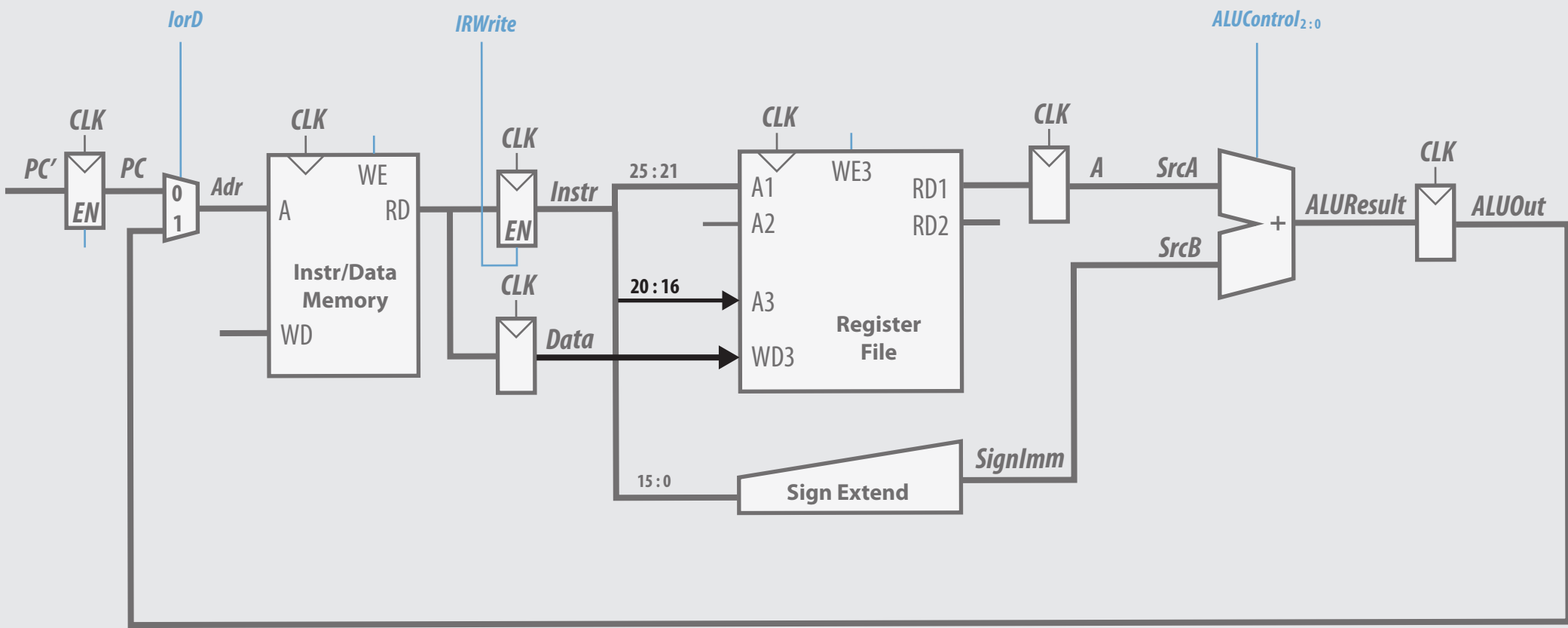
The next step is to load the data from the calculated address in the memory. We add a multiplexer in front of the memory to choose the memory address Adr from either the PC or $ALUOut$. The multiplexer select signal is called $lorD$ to indicate either an instruction or data address. The data read from the memory is stored in another nonarchitectural register, called $Data$.

Step 6 : load data from memory

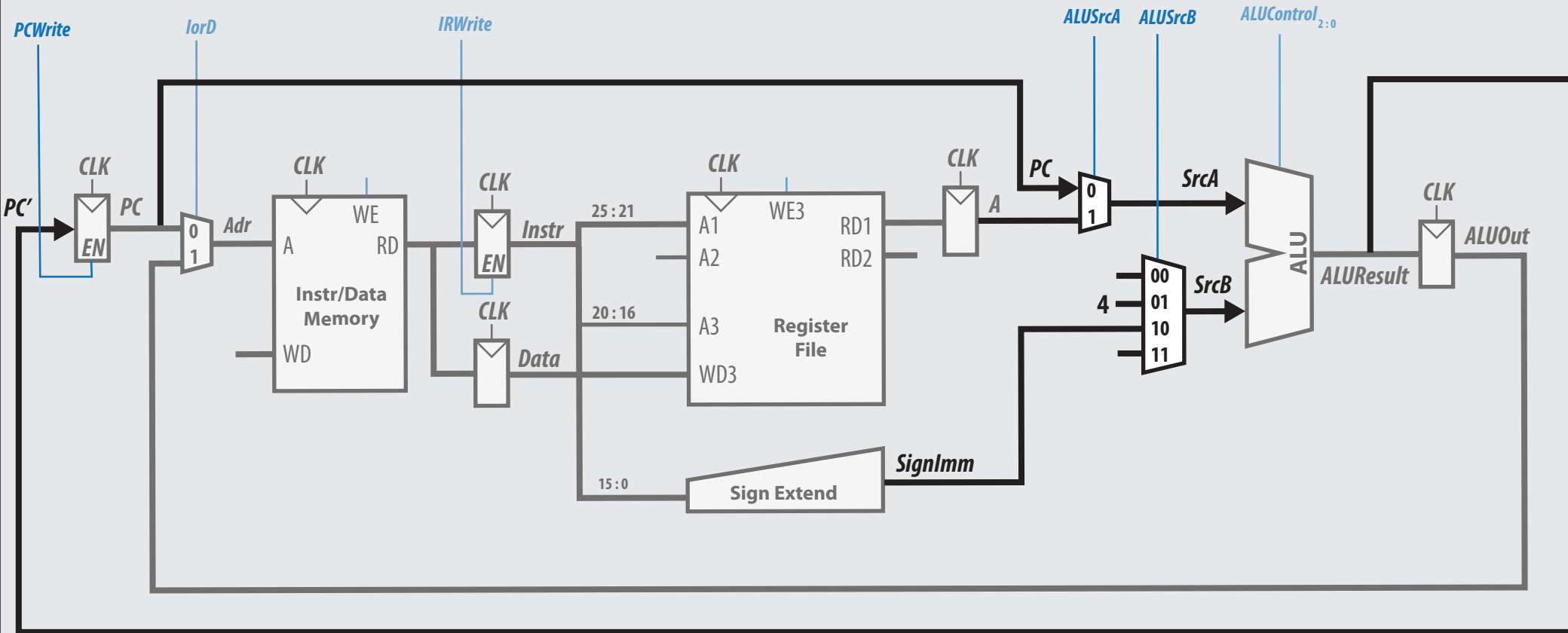


Notice that the address multiplexer enables us to reuse the memory during the `lw` instruction. On the first step, the address is taken from the `PC` to fetch the instruction. On the later step, the address is taken from `ALUOut` to load the data. Hence, the control signal `lorD` must have different values on these different steps.

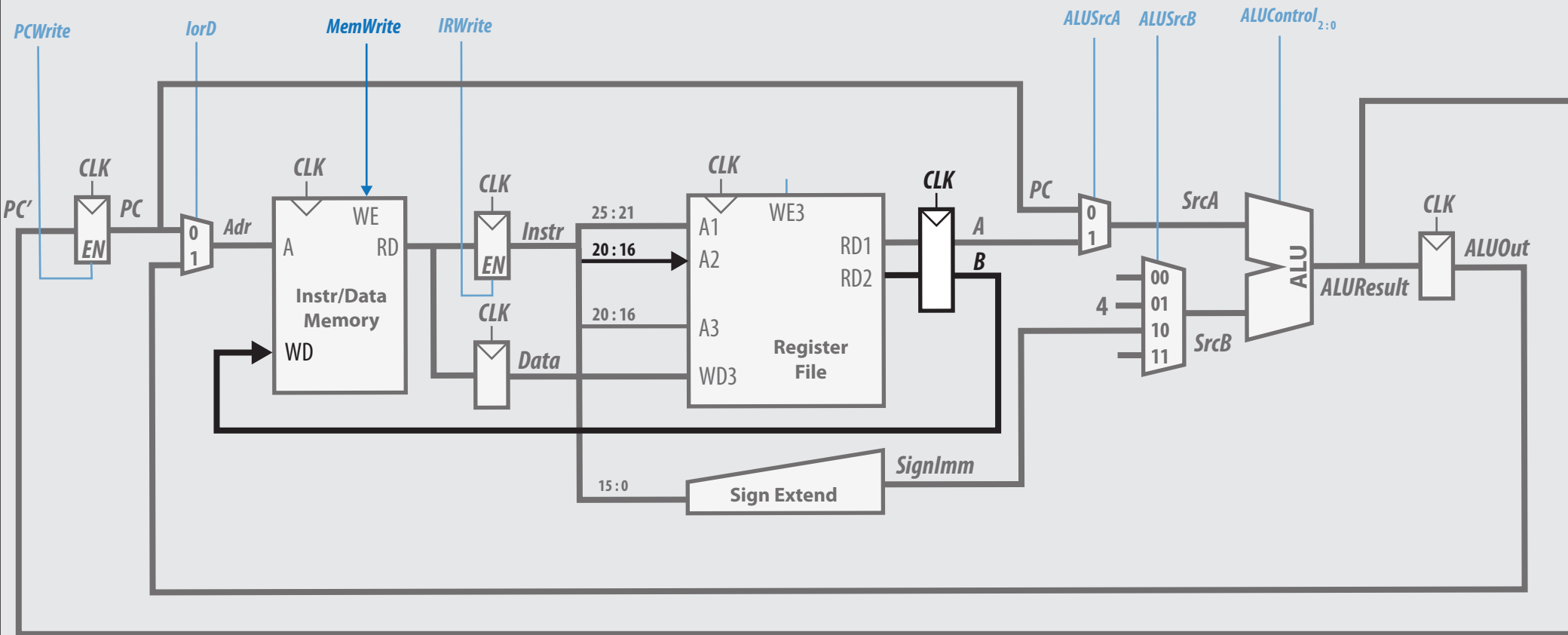
Step 7 : write back data to register file



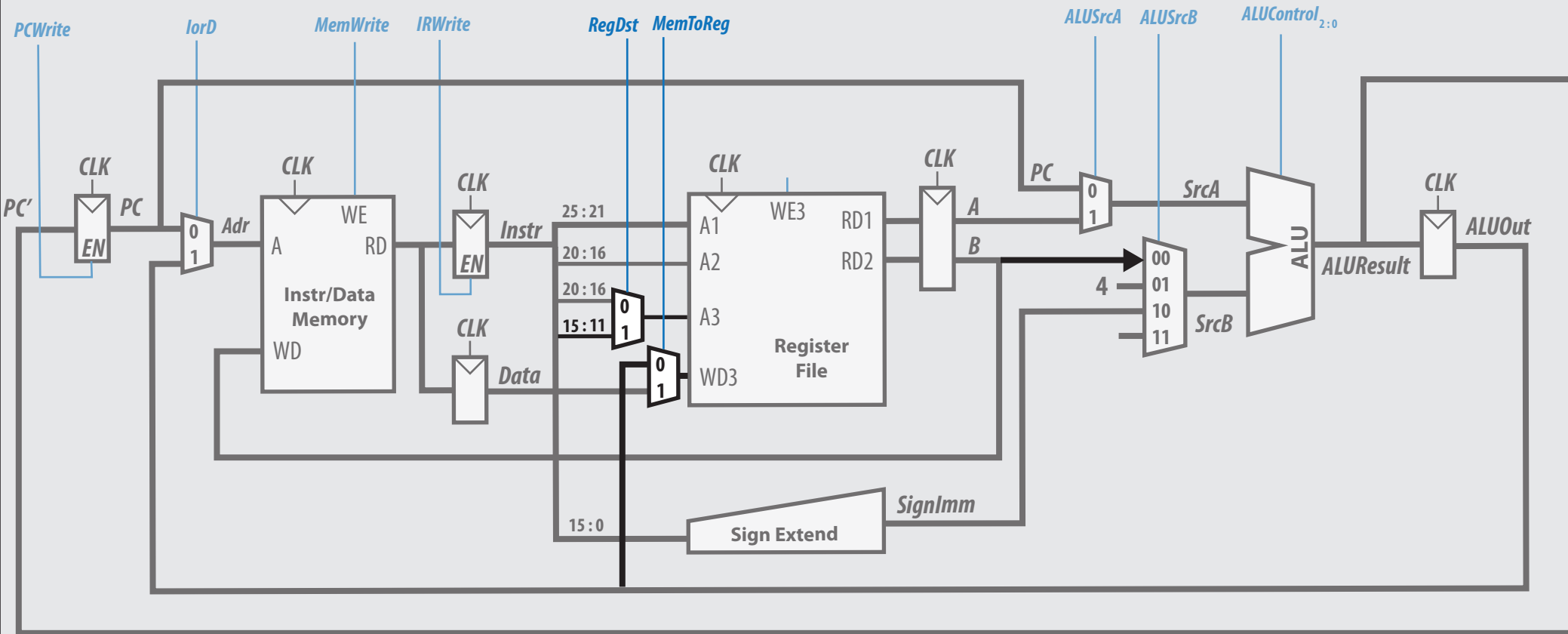
Step 8 : increment PC by 4



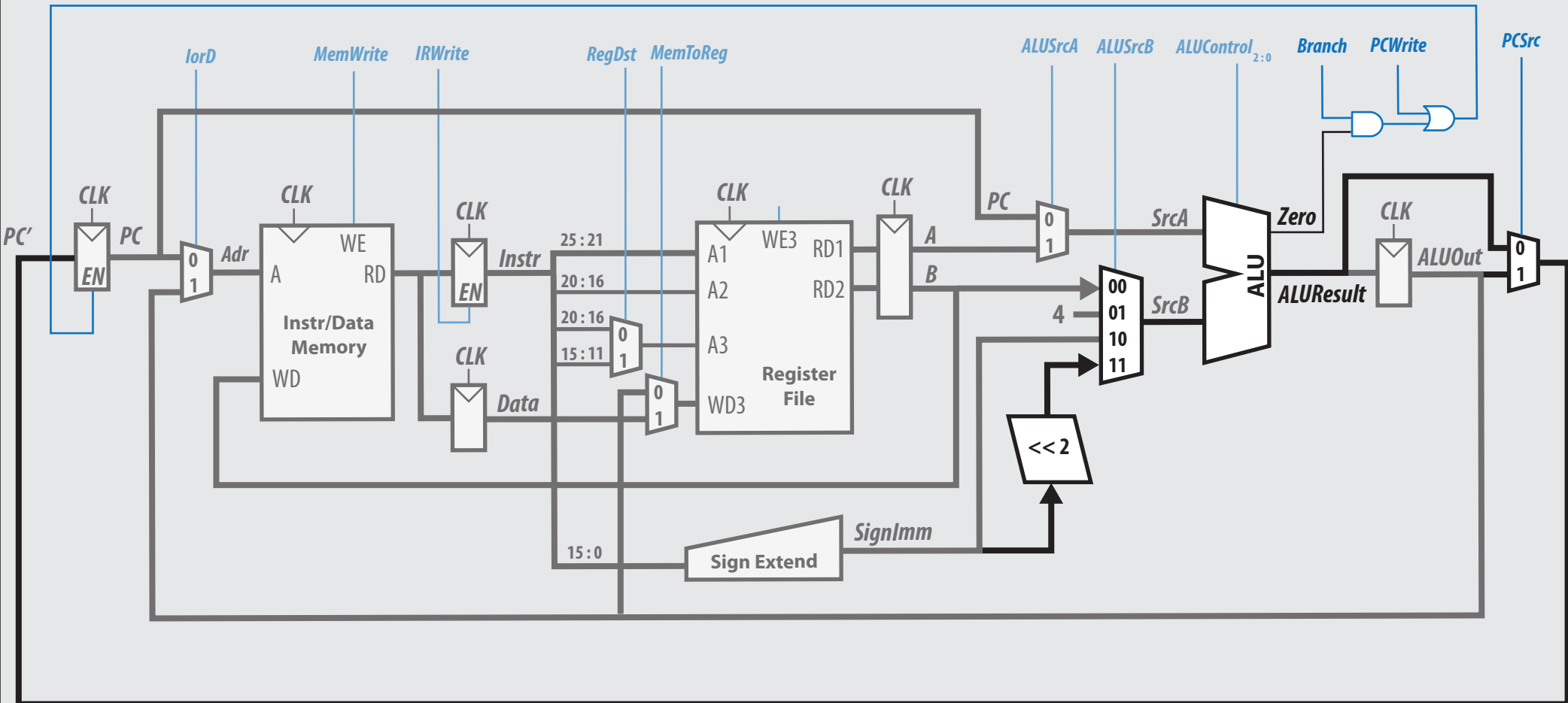
Enhanced data path for the *sw* instruction



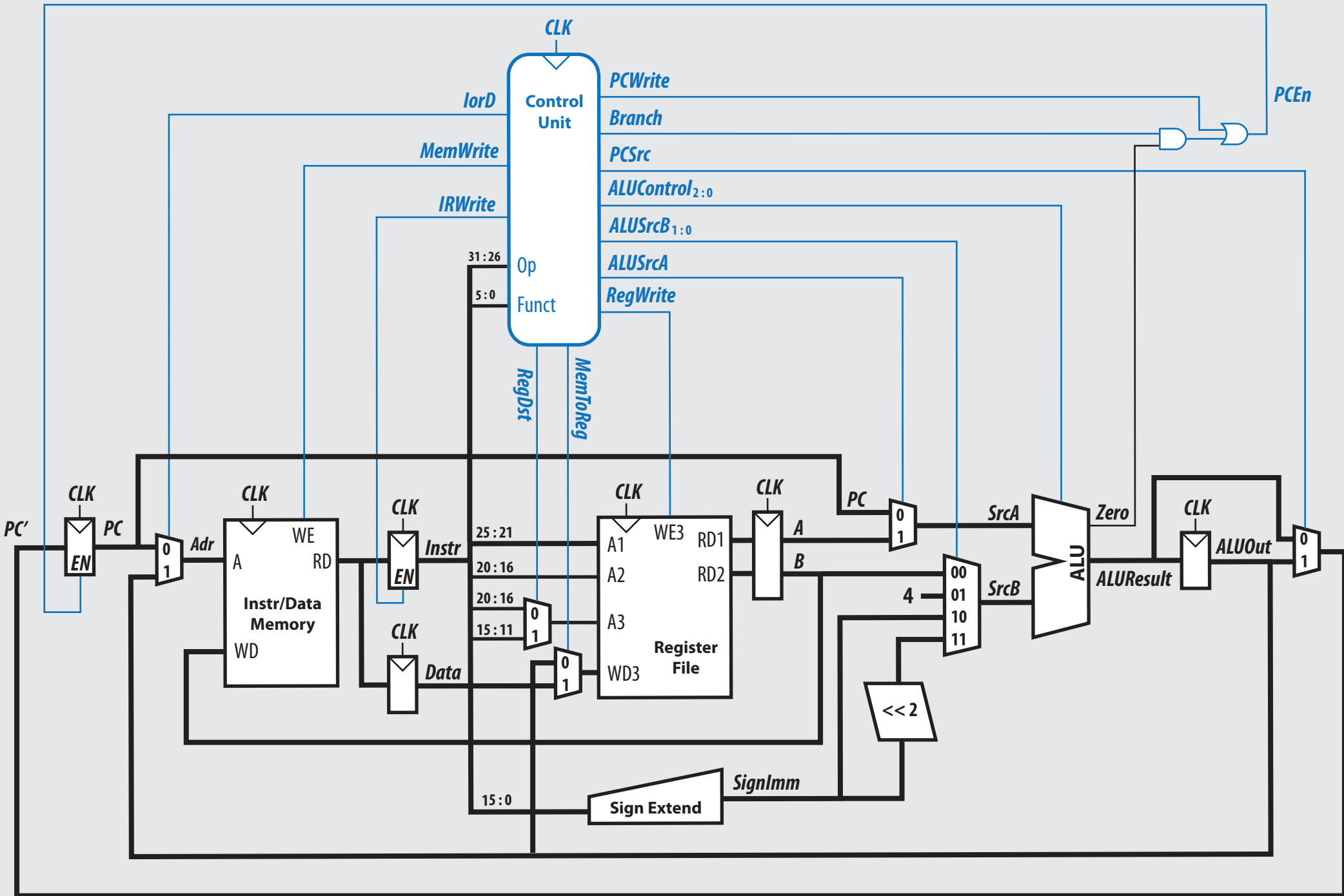
Enhanced data path for R-type instructions



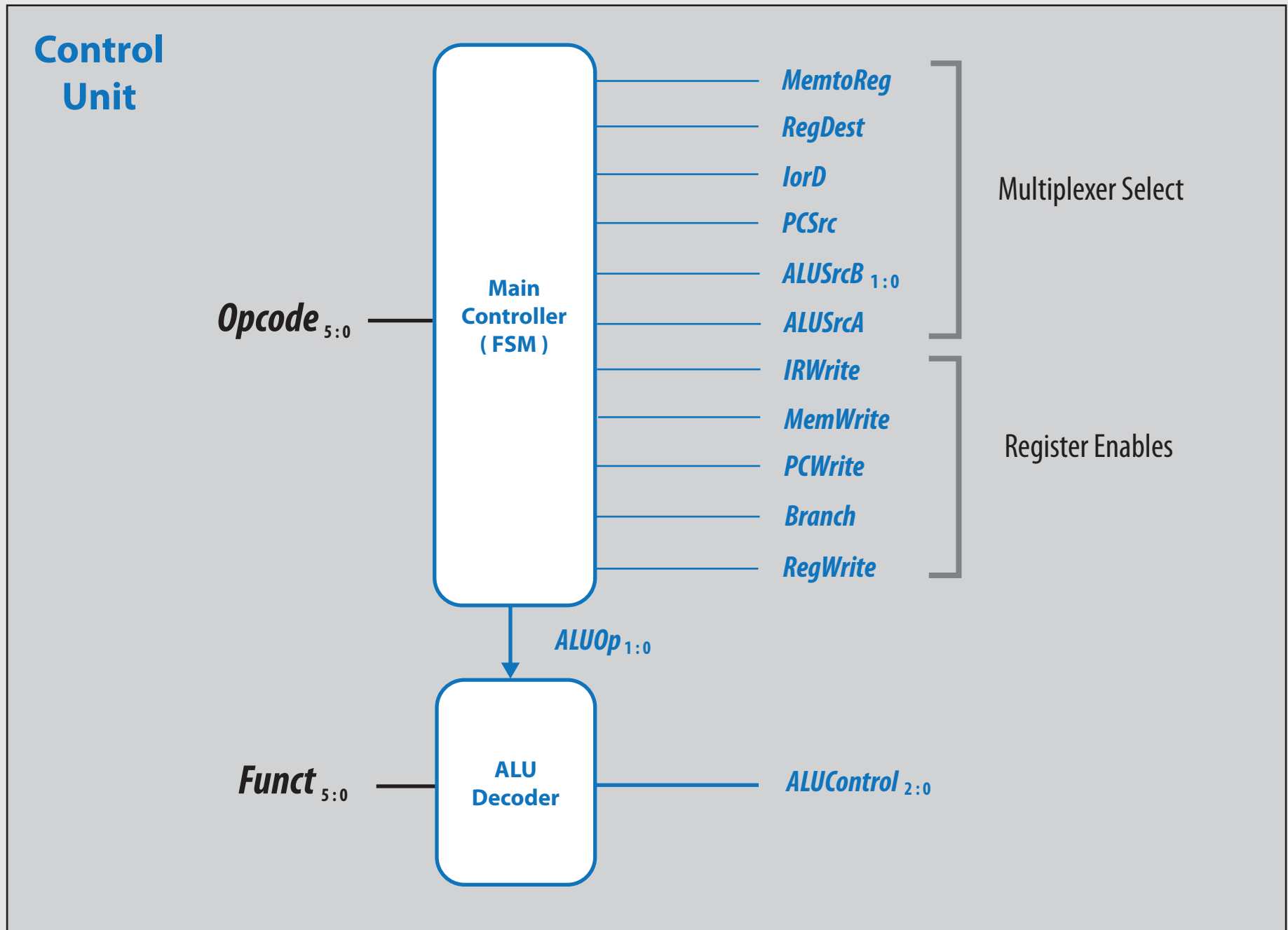
Enhanced data path for the `beq` instruction



Complete multicycle MIPS processor



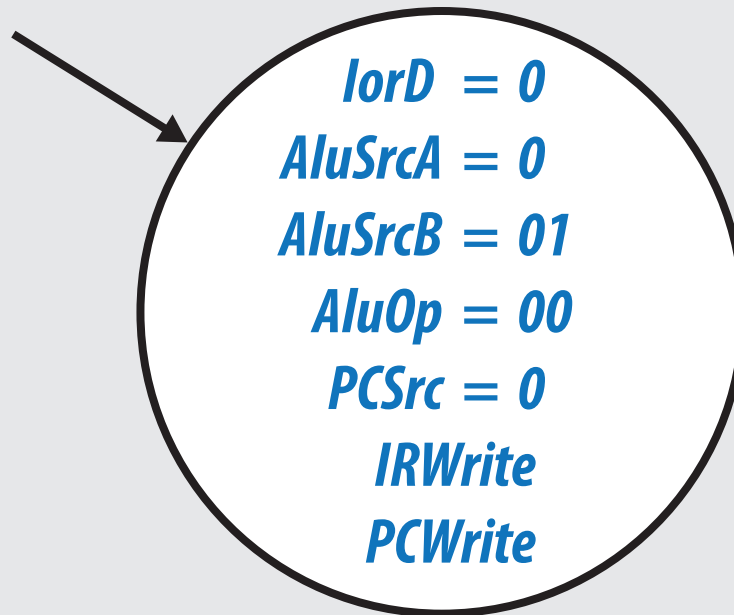
Control unit internal structure



Fetch

S0 : Fetch

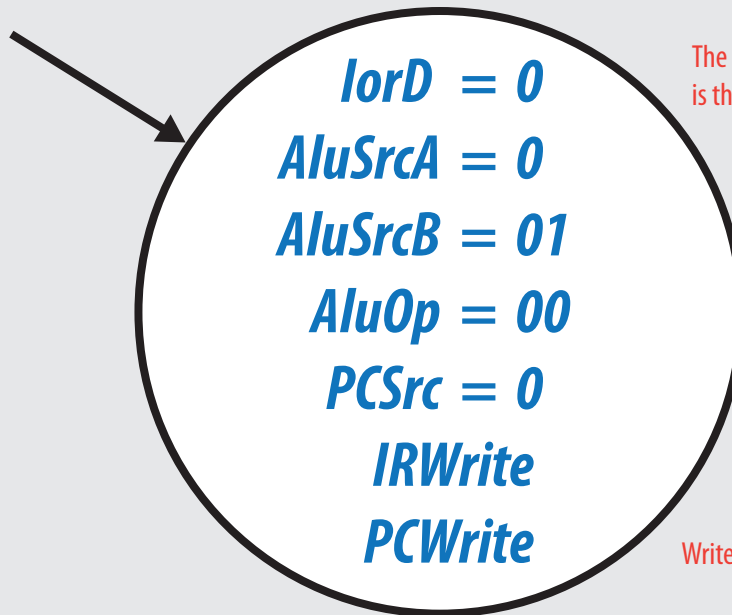
Reset



Fetch

S0 : Fetch

Reset



The address transmitted to the Instruction/Data Memory is the address of an Instruction

The SrcA of the ALU is equal to the current Program Counter (PC)

The SrcB of the ALU is equal to 4

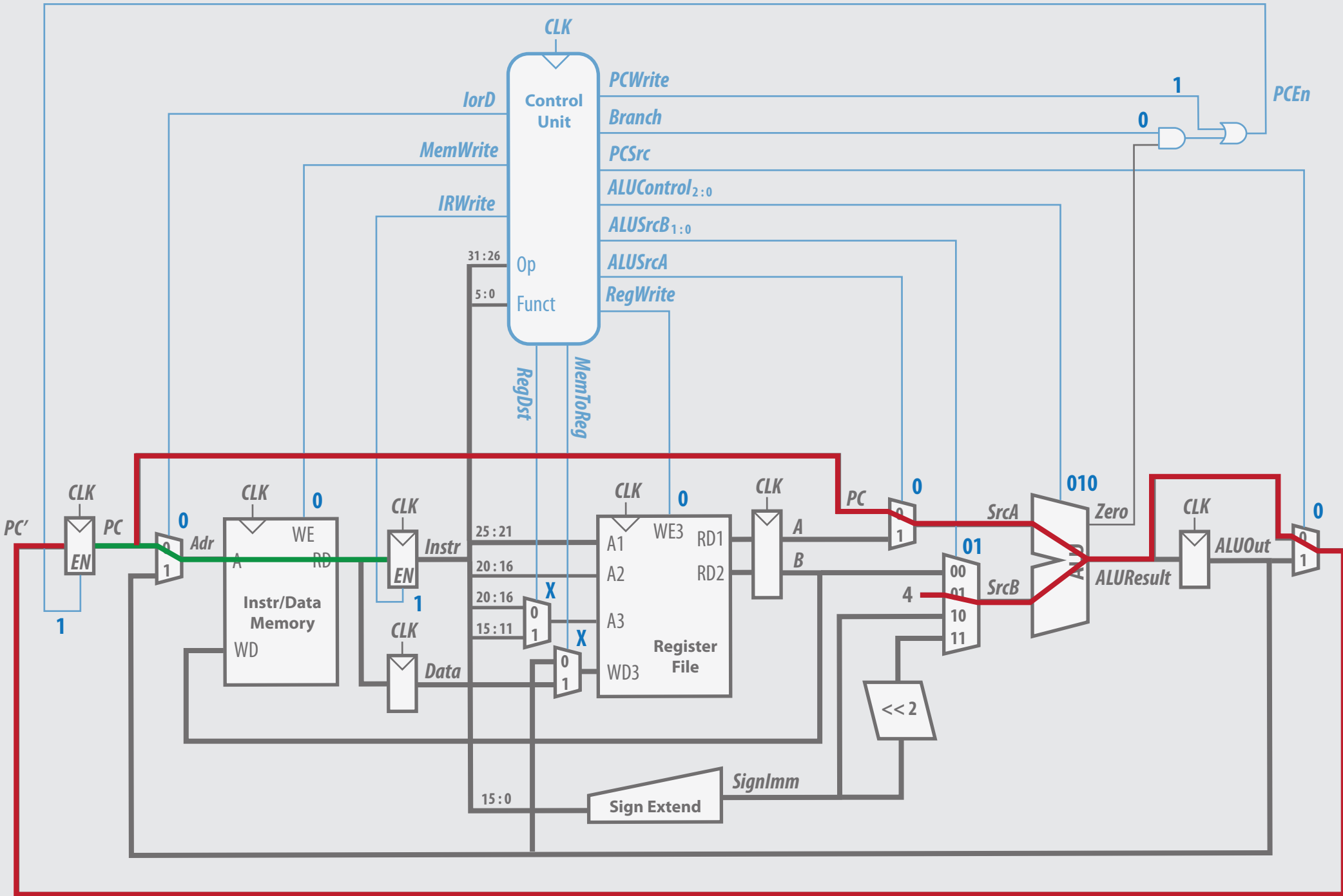
The ALU performs an addition $PC' = PC + 4$ to compute the next Program Counter (PC')

The next Program Counter (PC') is computed by the ALU

Writes the current instruction in the Instruction Register (IR)

Writes the next Program Counter ($PC' = PC + 4$) in the Program Counter (PC)

Data flow during the fetch step

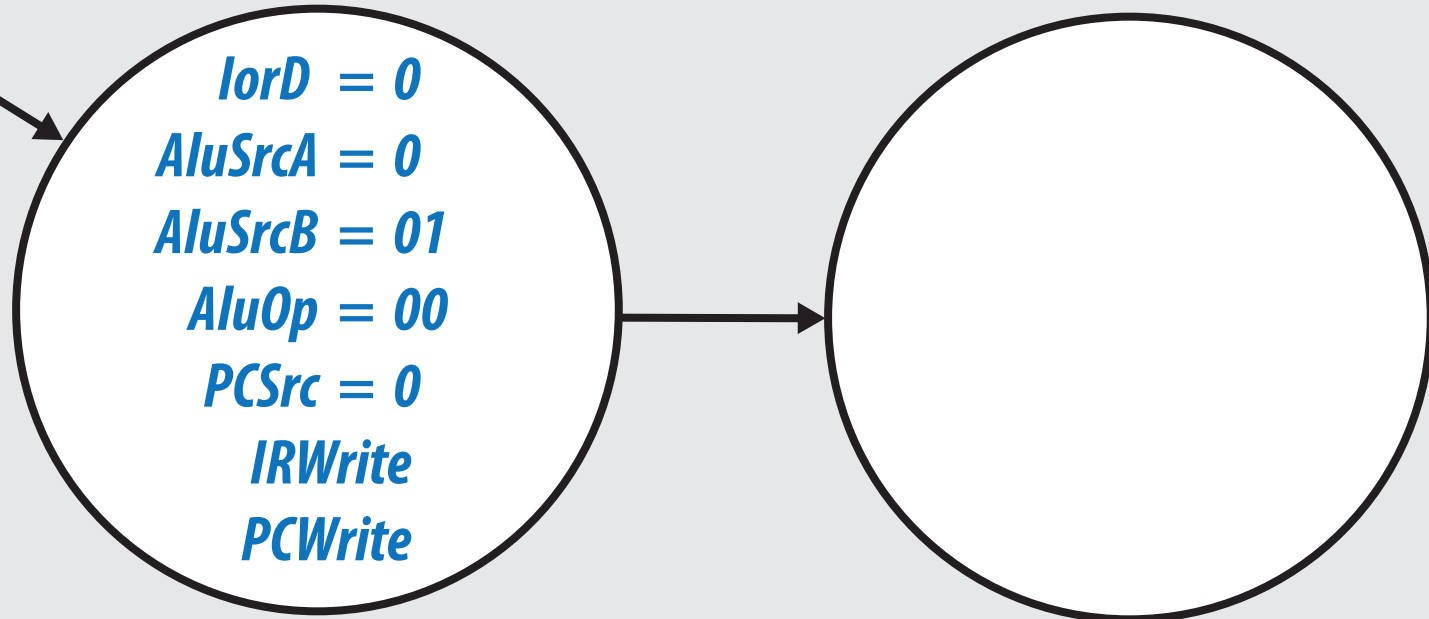


Decode

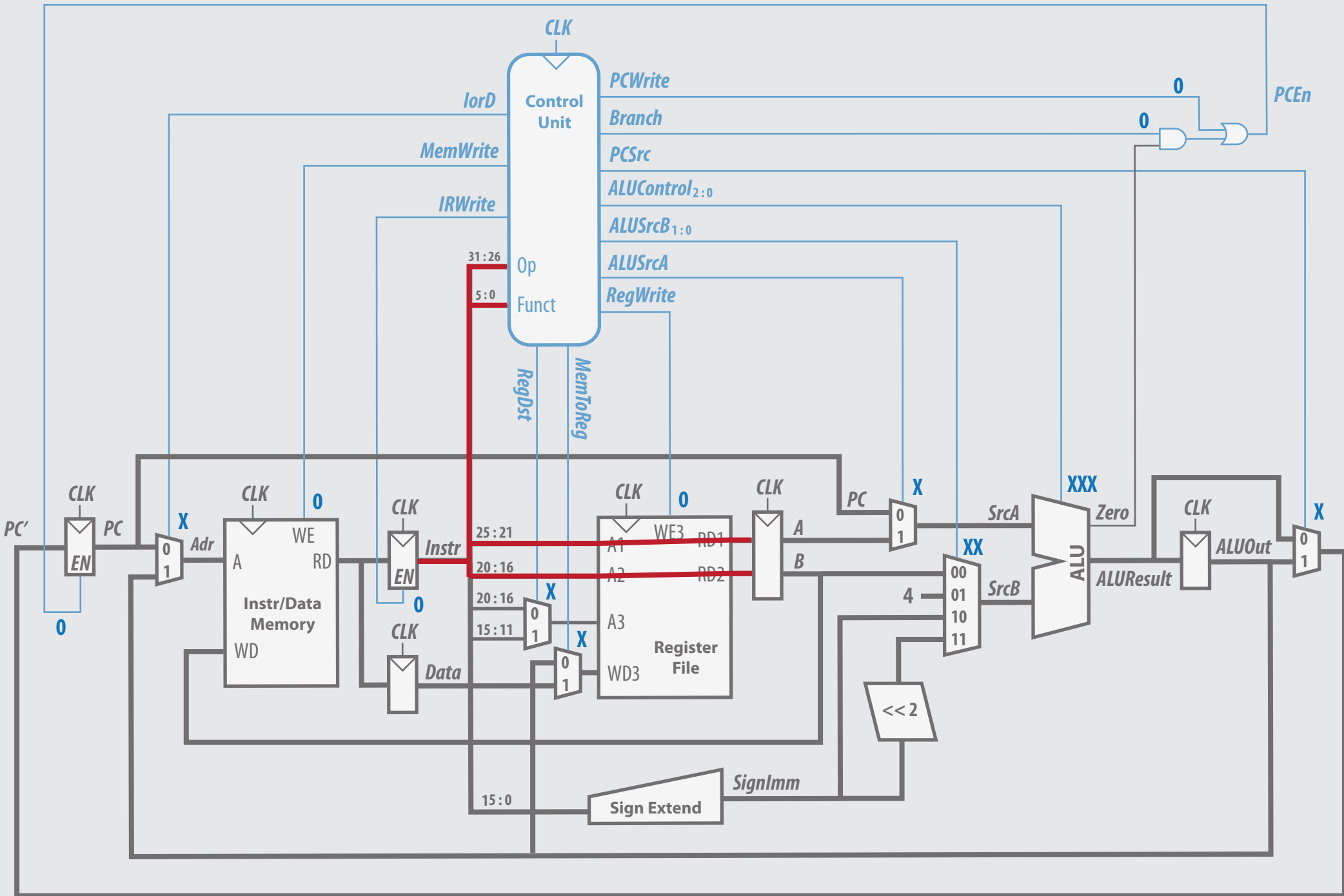
S0 : Fetch

S1 : Decode

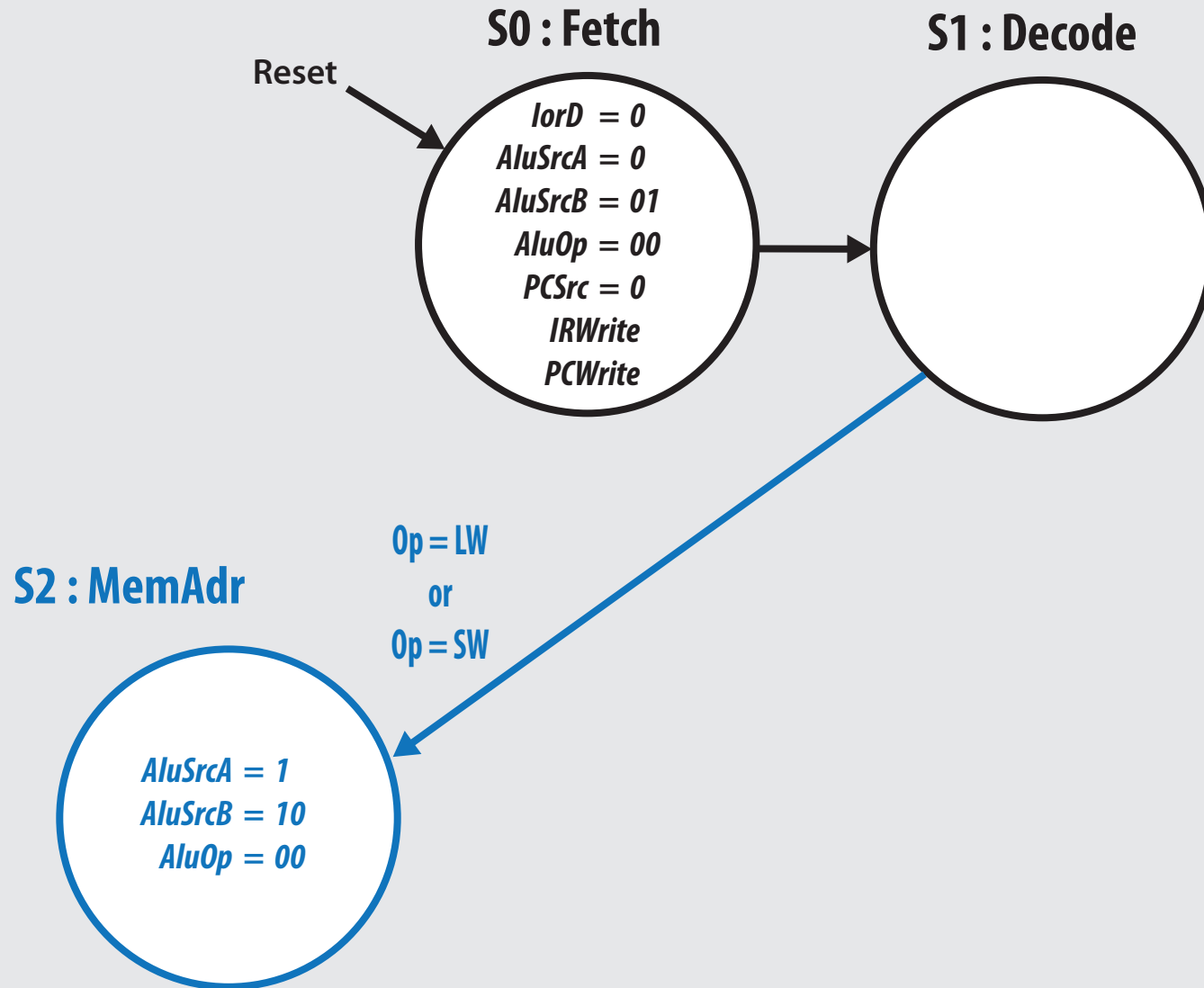
Reset



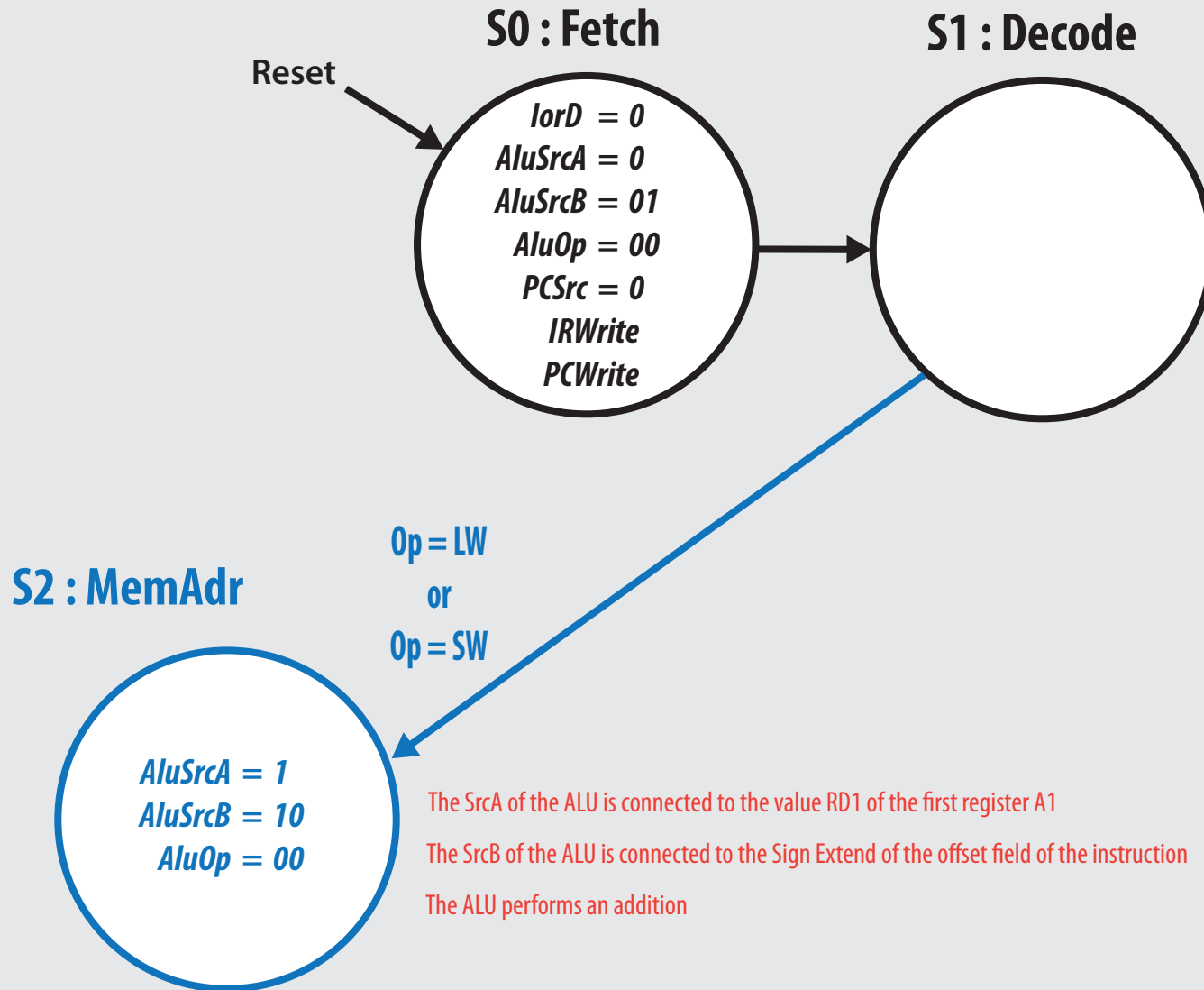
Data flow during the decode step



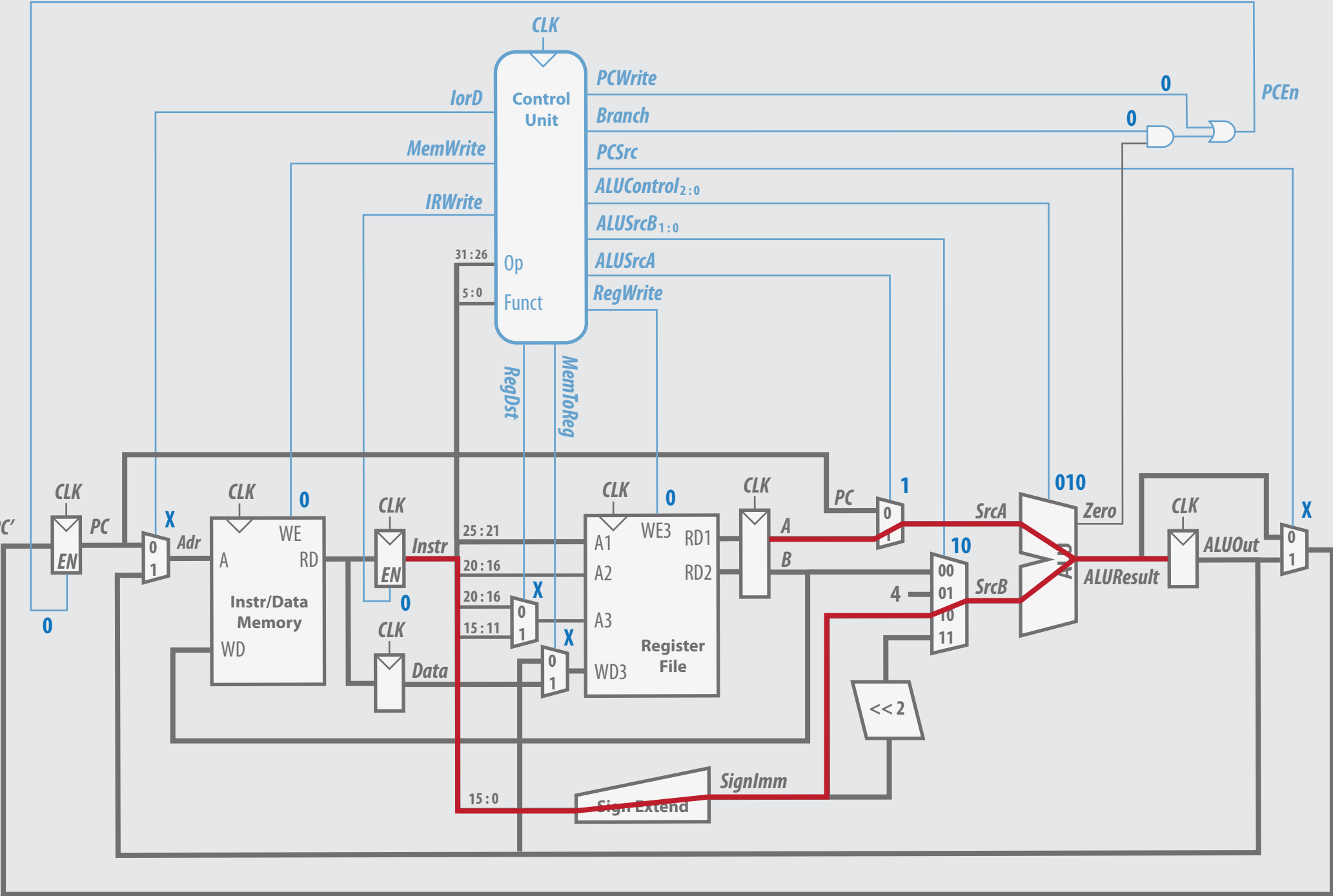
Memory Address Computation



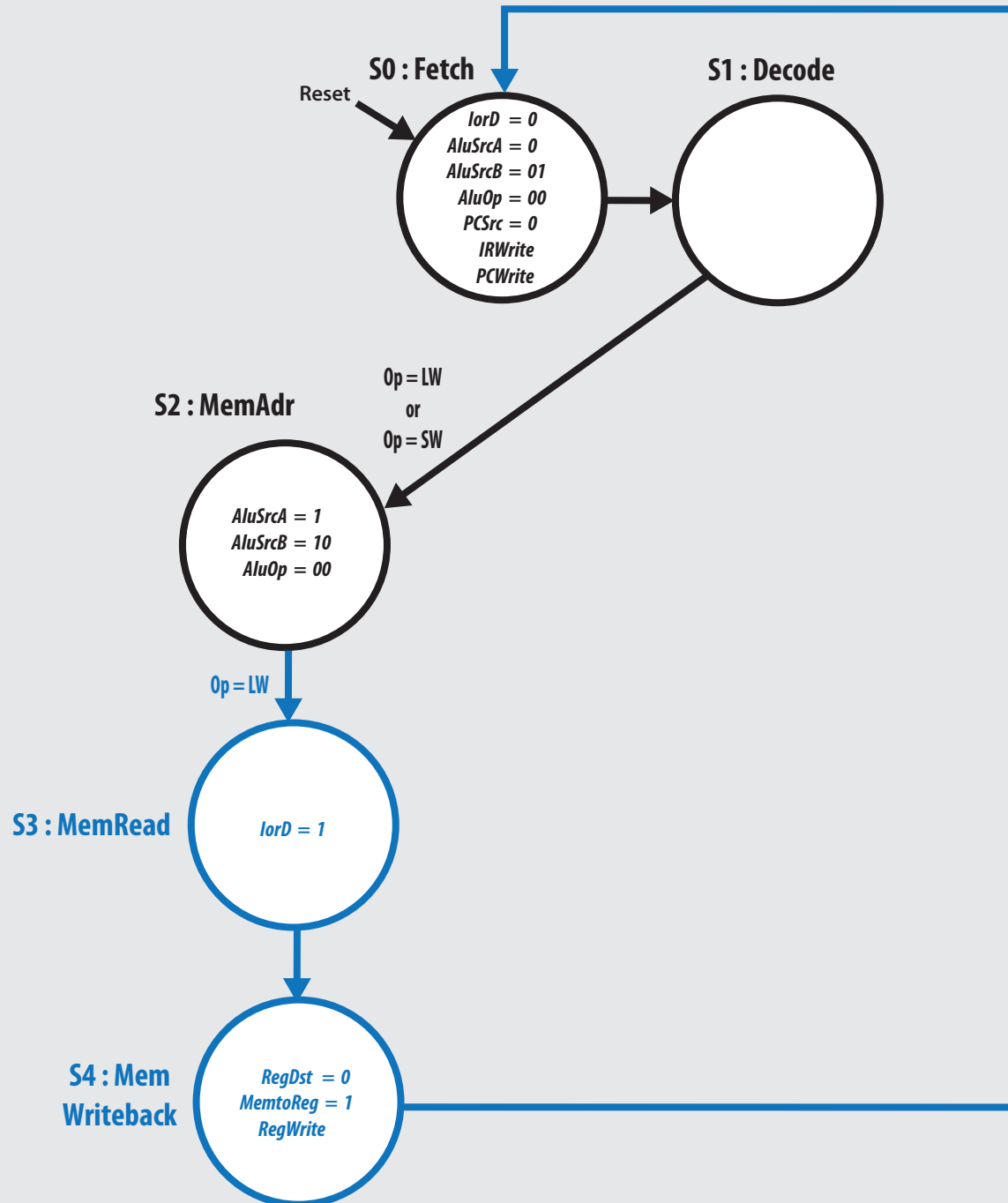
Memory Address Computation



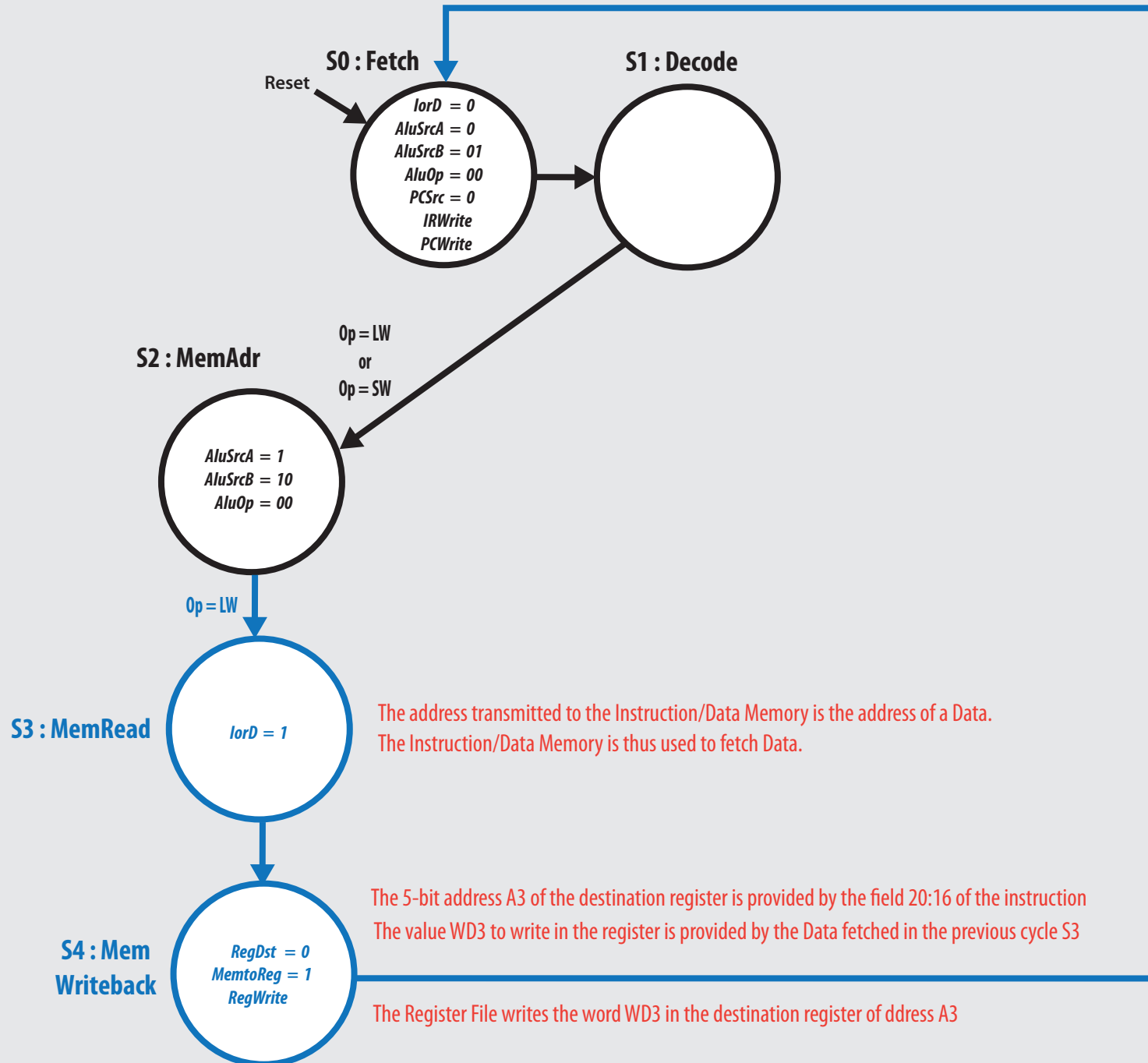
Data flow during memory address computation



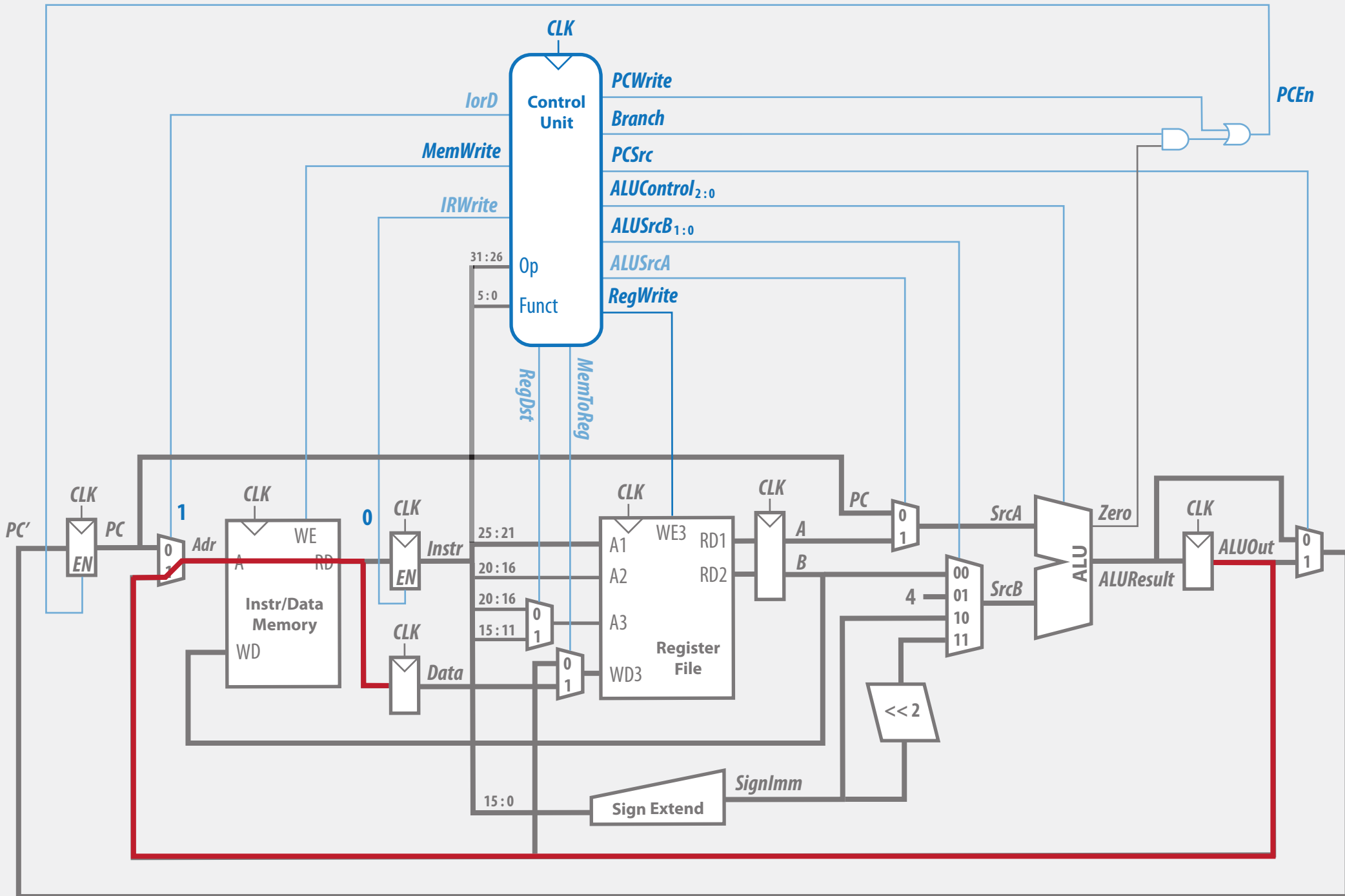
Memory Read



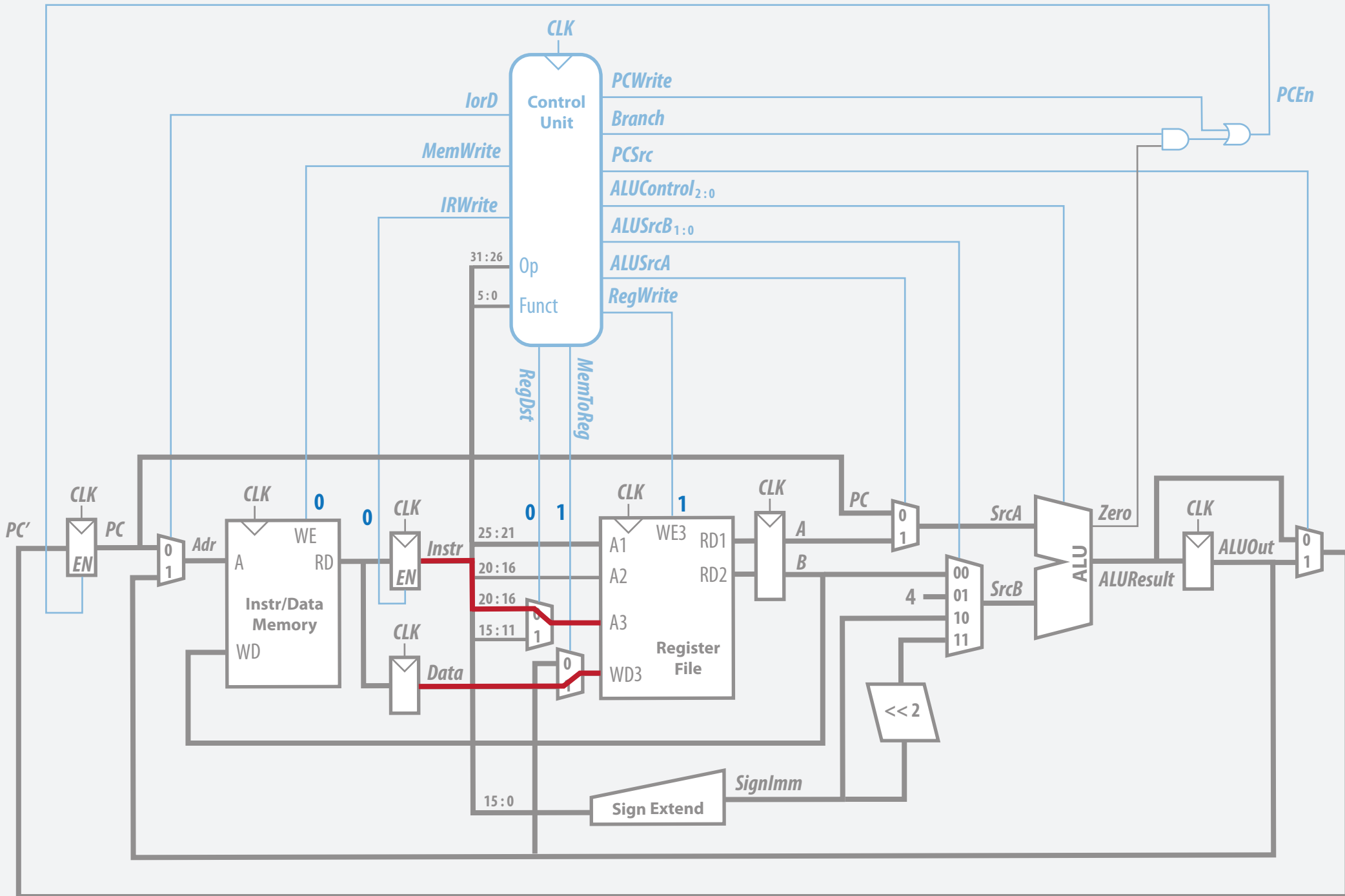
Memory Read



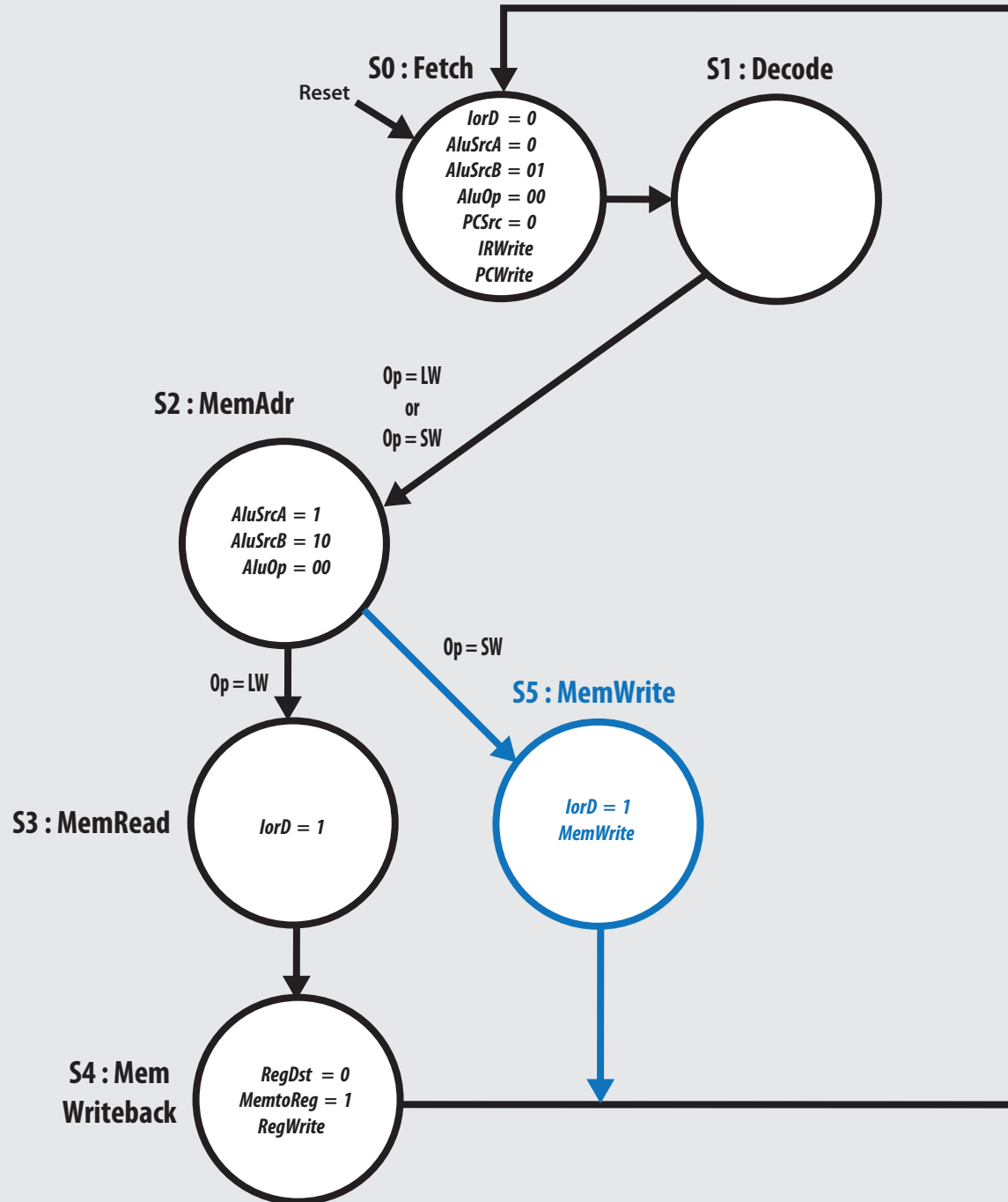
Data flow during the Memory Read



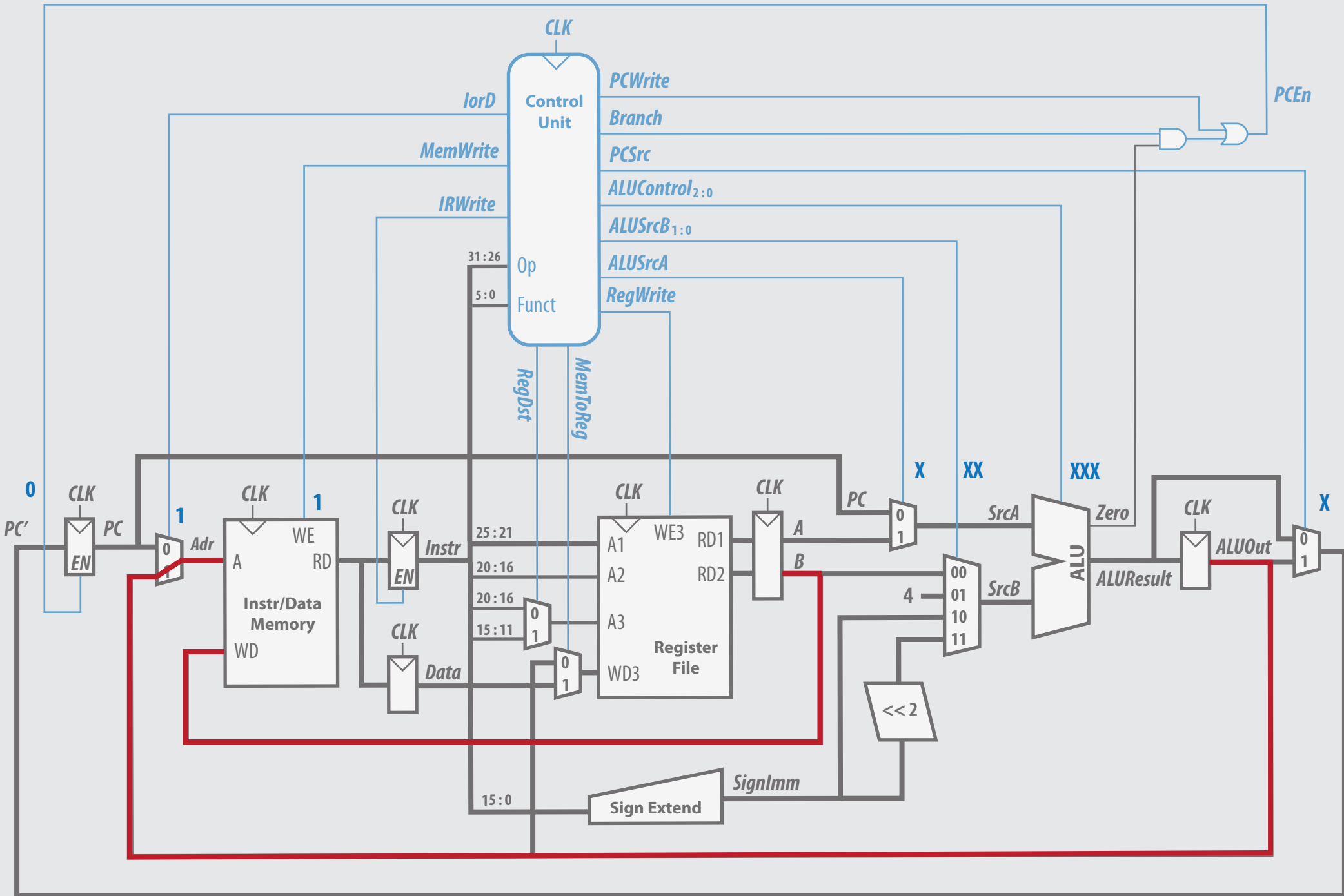
Data flow during the Writeback



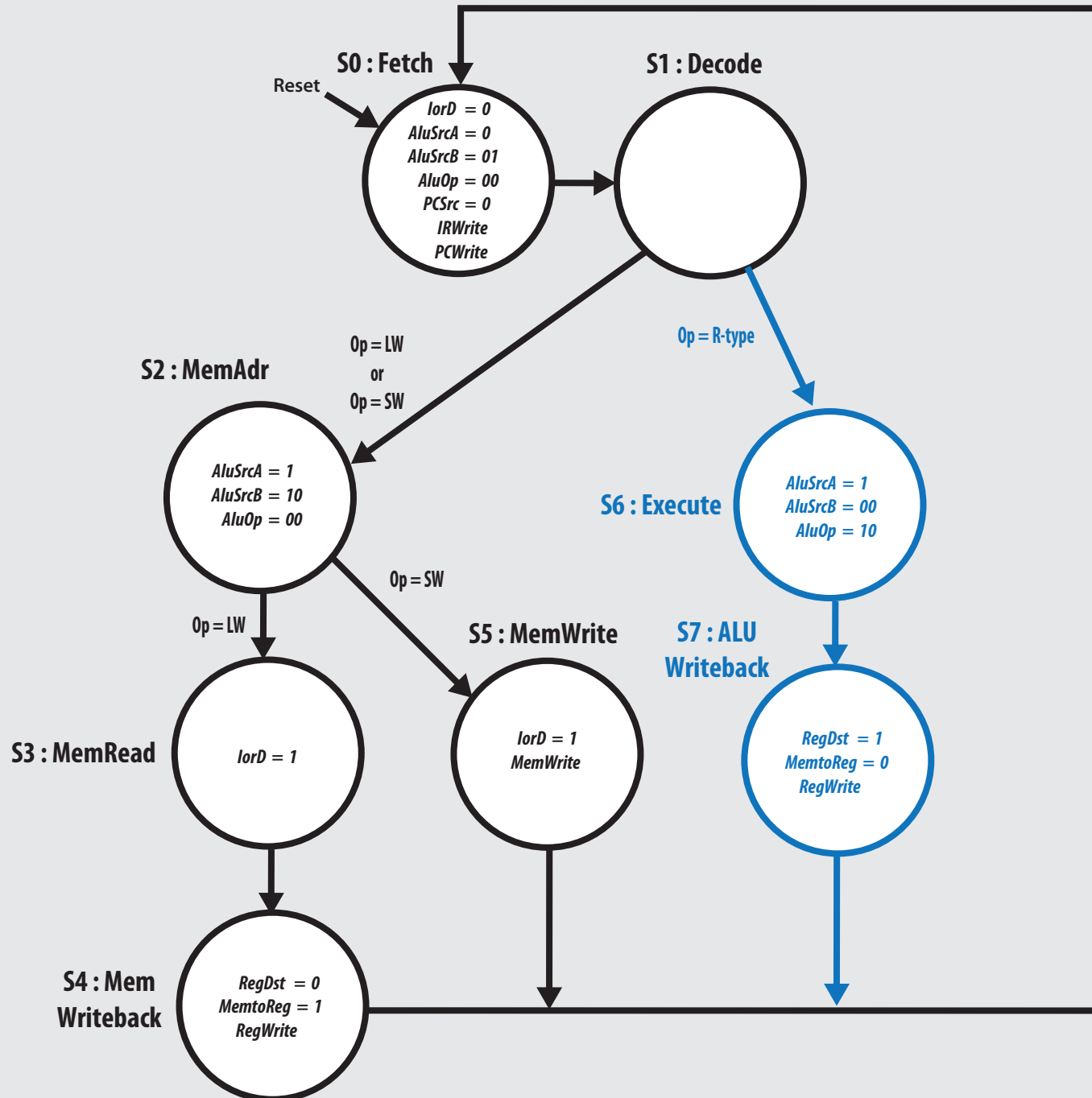
Memory Write



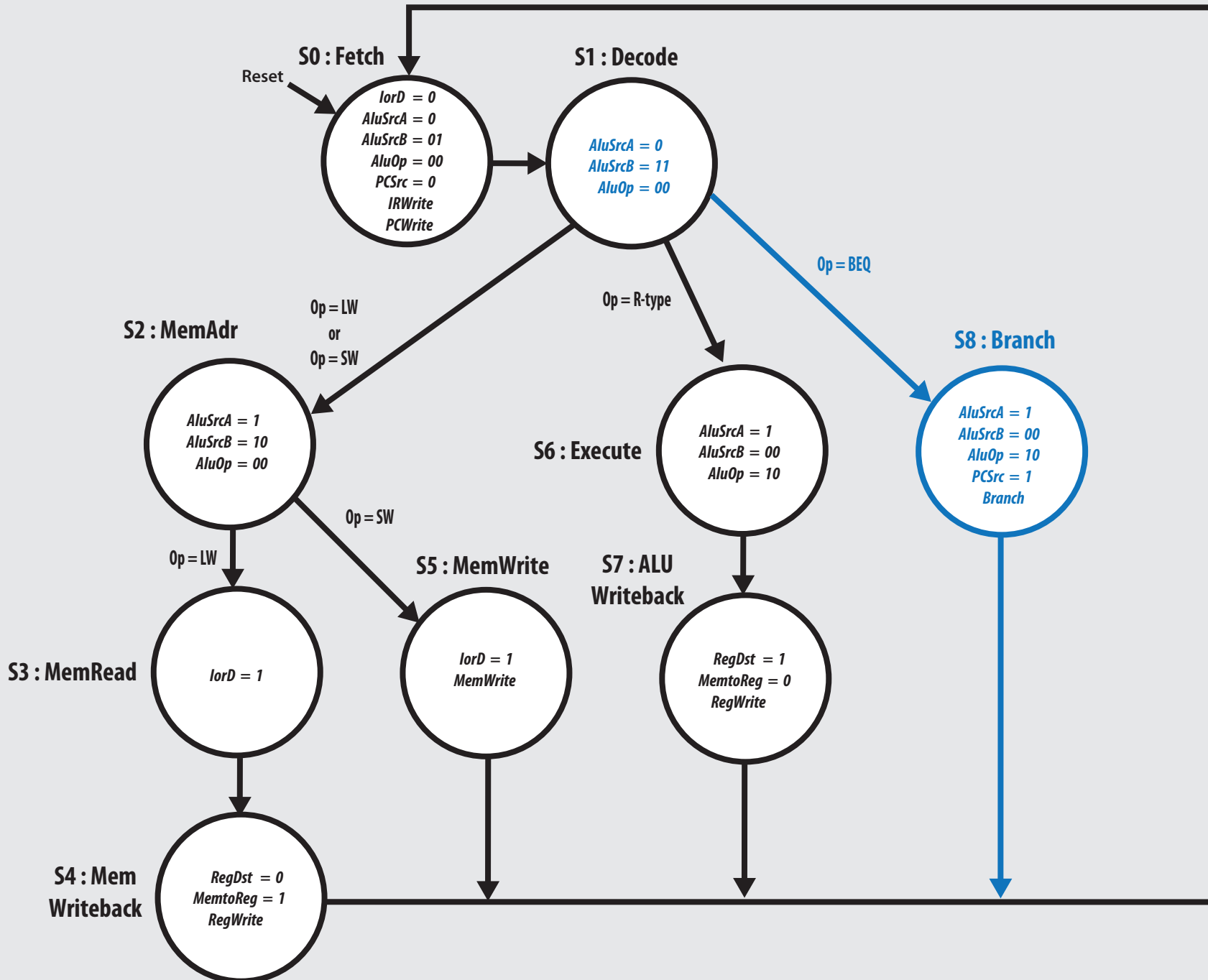
Data flow during the Memory Write step



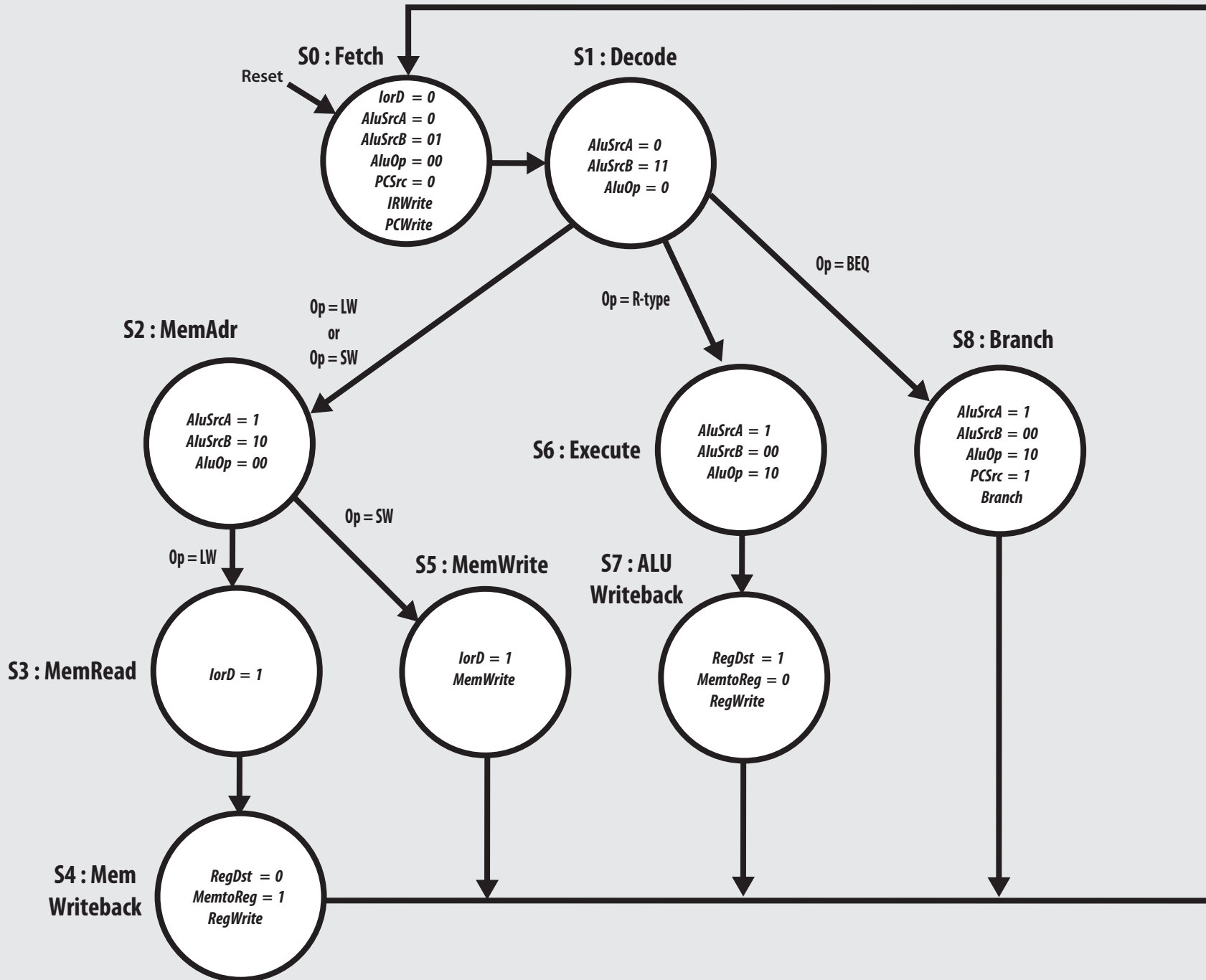
Execute R-type operation



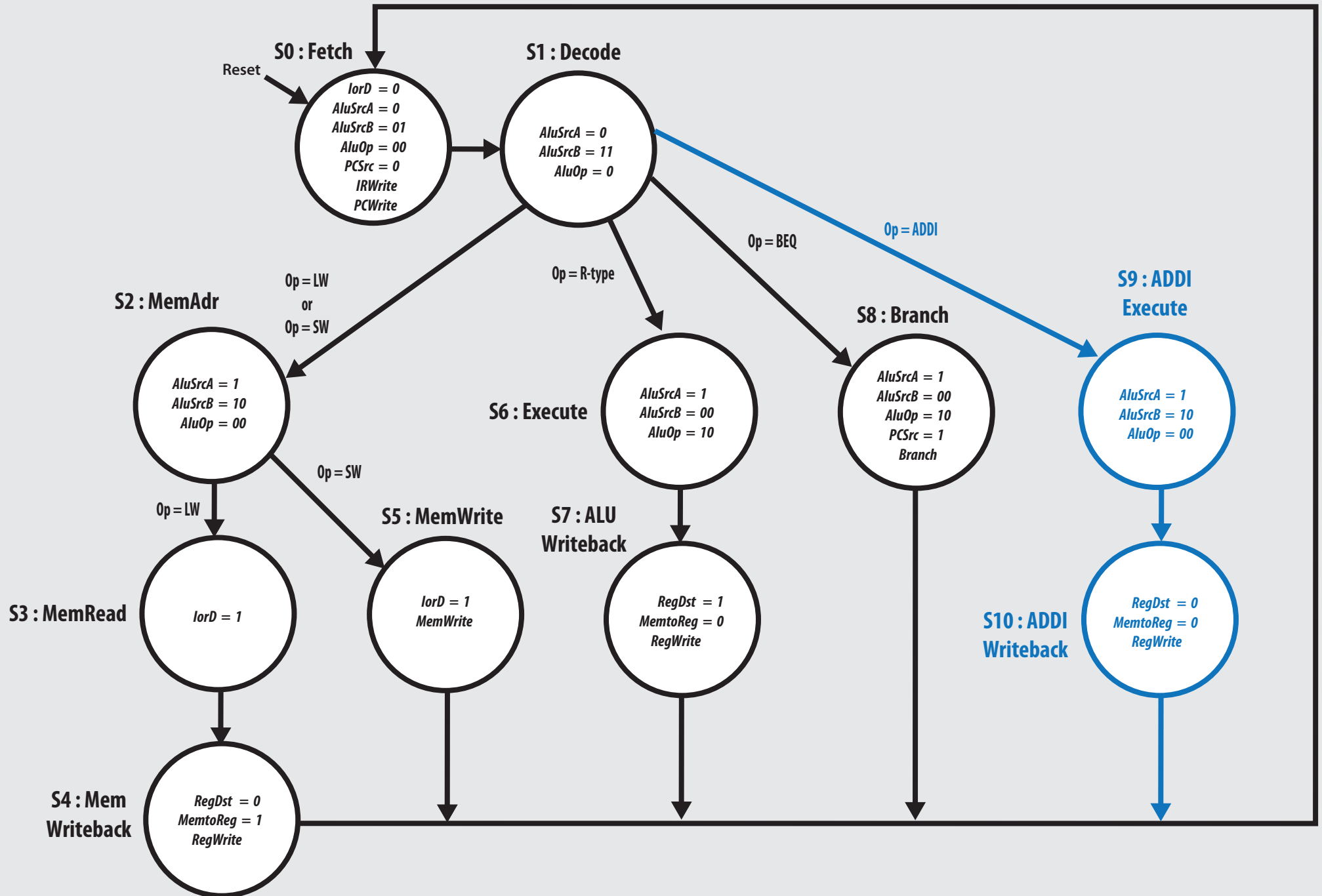
Branch



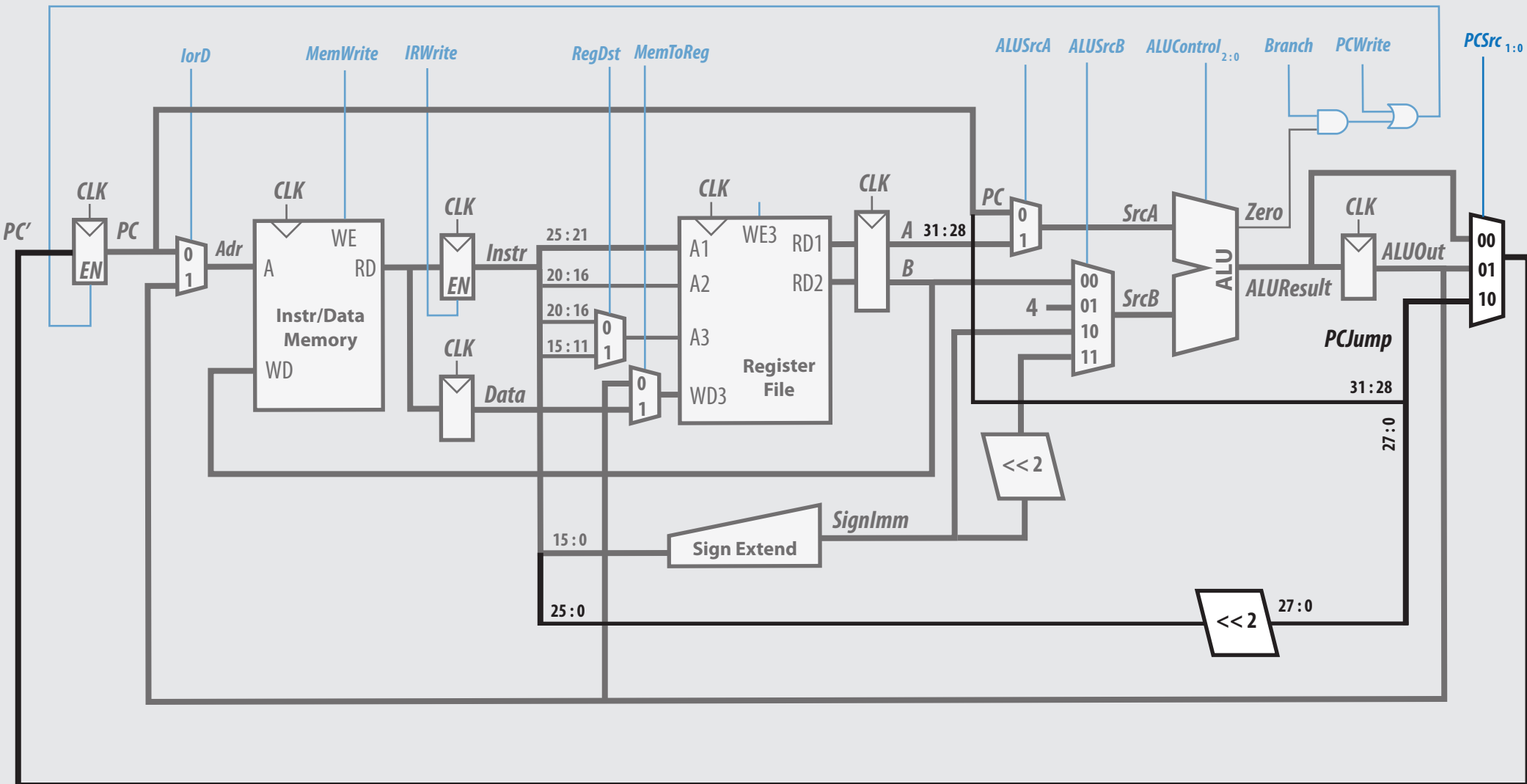
Complete multicycle control FSM



Main controller states for `addi`



Multicycle MIPS datapath enhanced to support the *j* instruction



Main controller state for j

