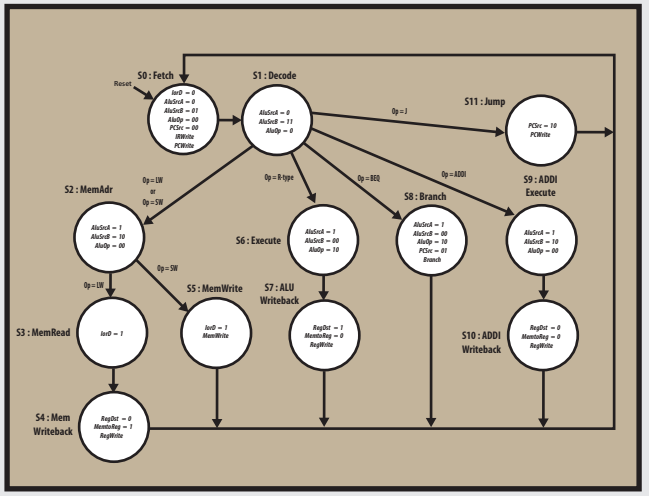
Computer Architecture

Paul Mellies

Lecture 15 : Multicycle microarchitectures



The multicycle microarchitecture

The multicycle microarchitecture

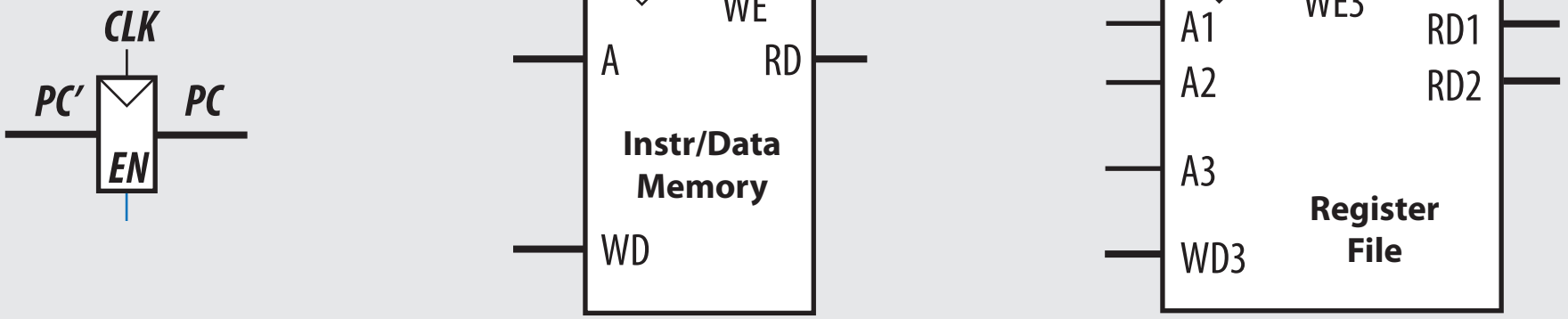
The single-cycle microarchitecture has three weakenesses :

- 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.

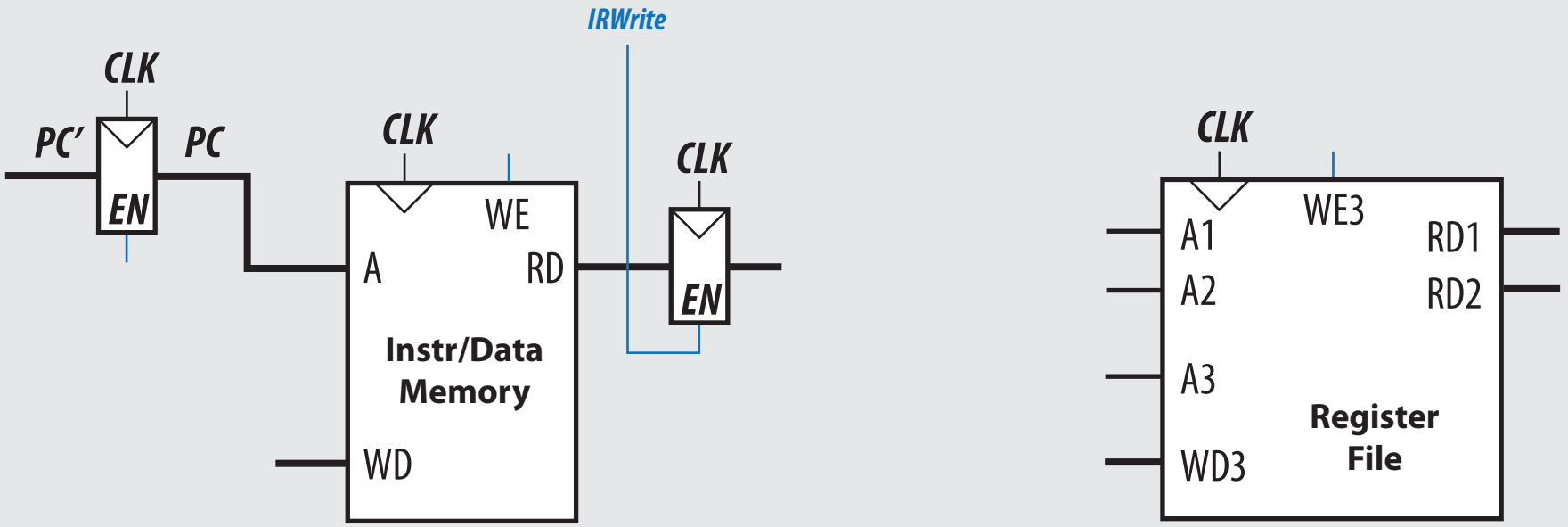
The multicycle processor addresses these three weaknesses by breaking an instruction into multiple shorter steps.

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

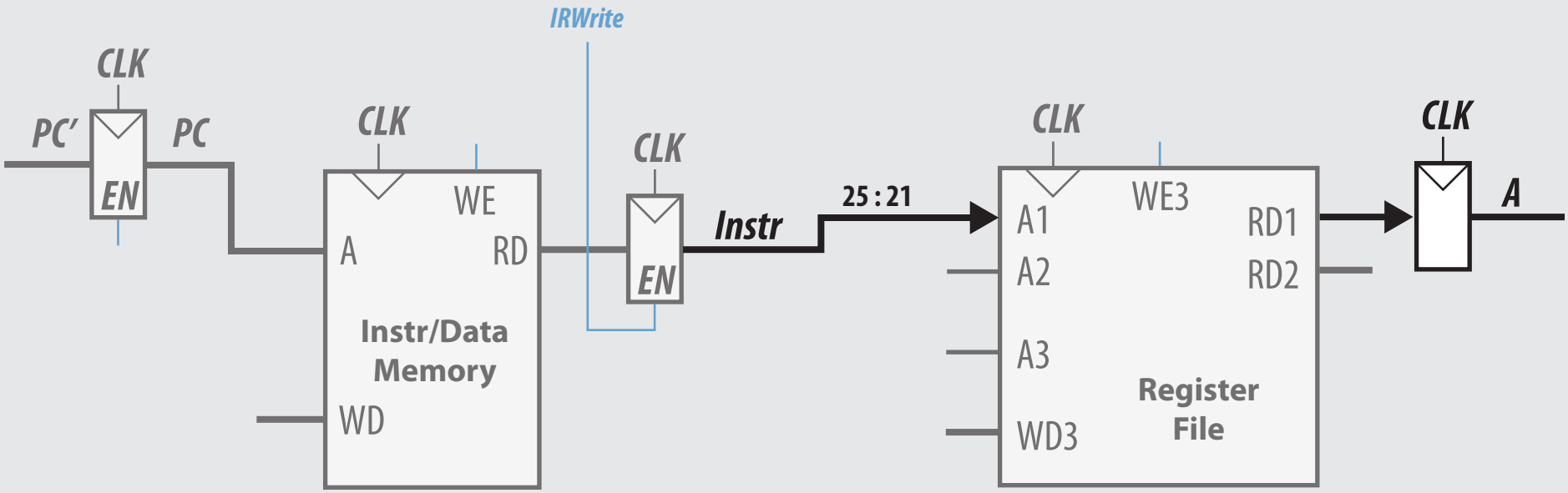


Step 0 : fetch the instruction from memory and store it in the Instruction Register



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 1 : read value of source register and store base address in Register A

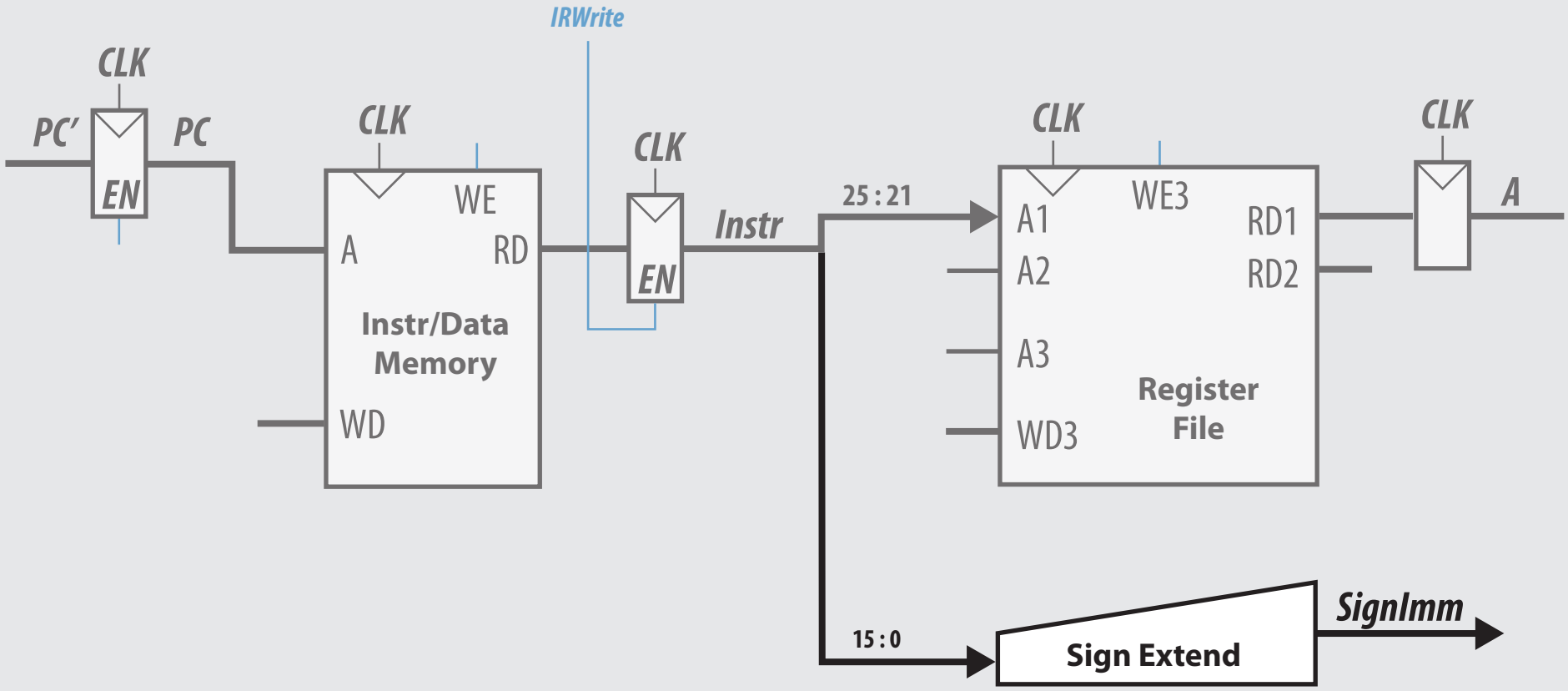


The next step is to read the source register containing the base address.

- The source register is specified in the `sr` field **[25 : 21]** of the instruction.

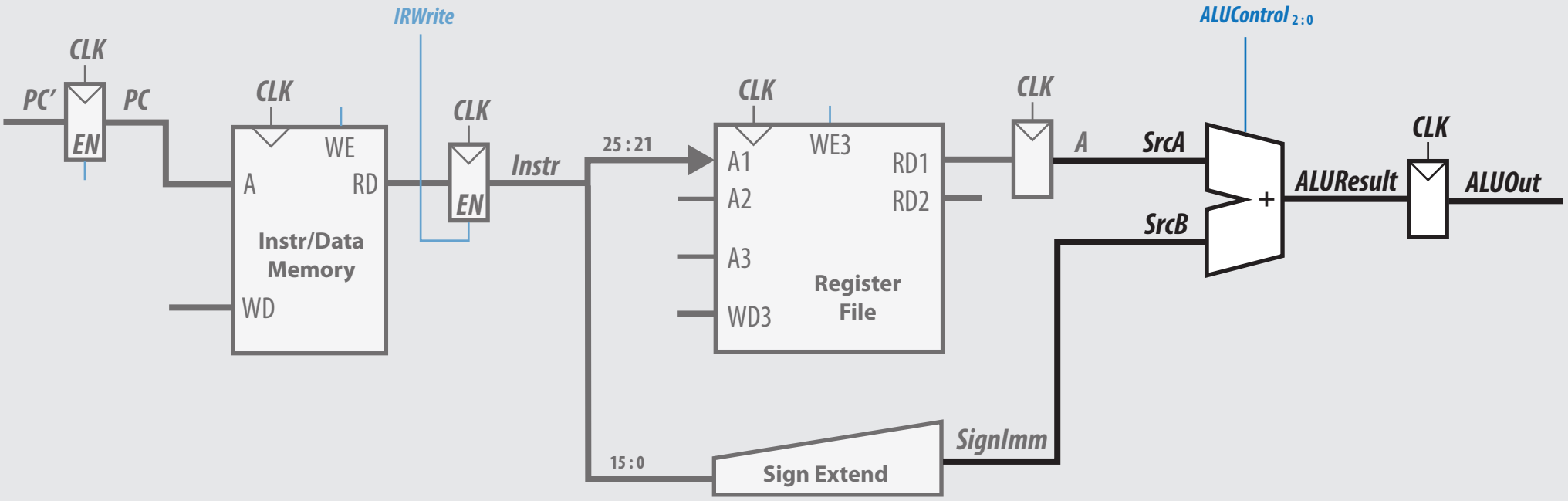
So, just as in the single-cycle case, the 5 bits of the instruction are connected to the address input **A1** of the register file.

Step 2 : sign-extend the immediate

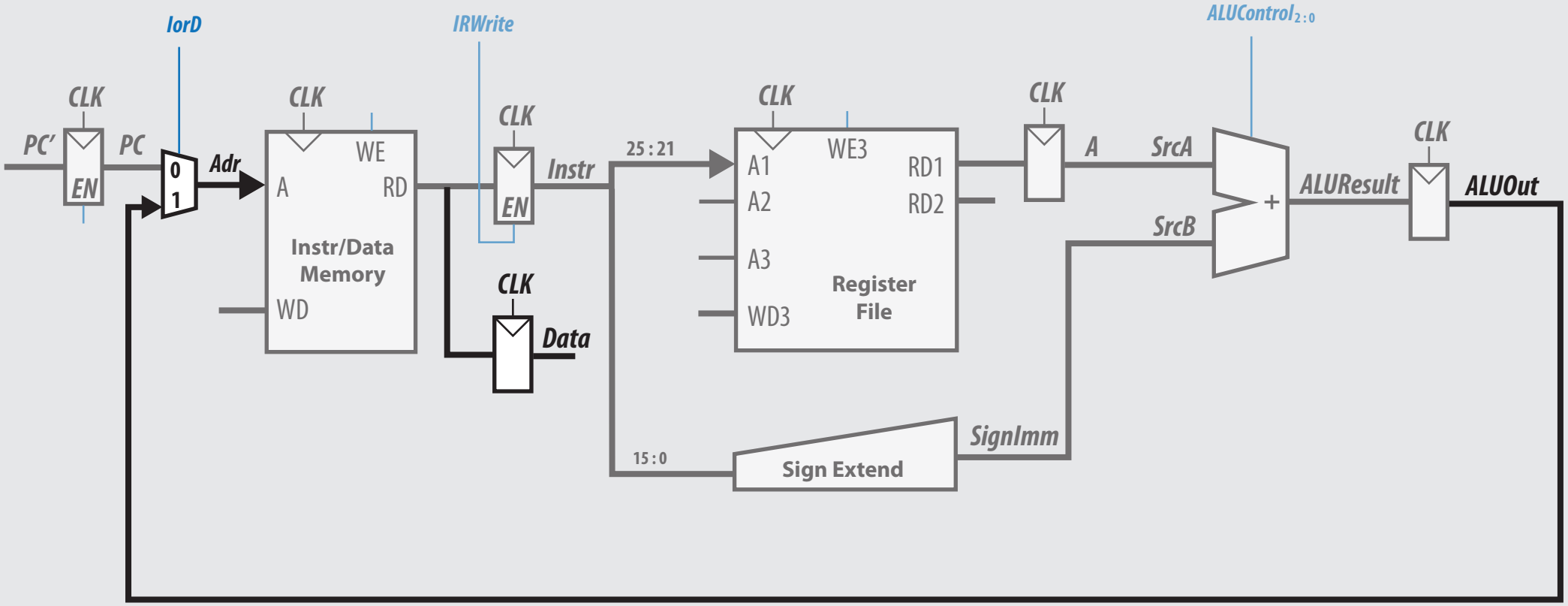


The `lw` instruction also requires an offset. The offset is stored in the immediate field of the instruction, ***Instr*** 15:0. Because the 16-bit immediate might be either positive or negative, it must be sign-extended to 32 bits. The 32-bit sign-extended value is called ***SignImm***.

Step 2 : add base address stored in Register A to the sign-extended offset

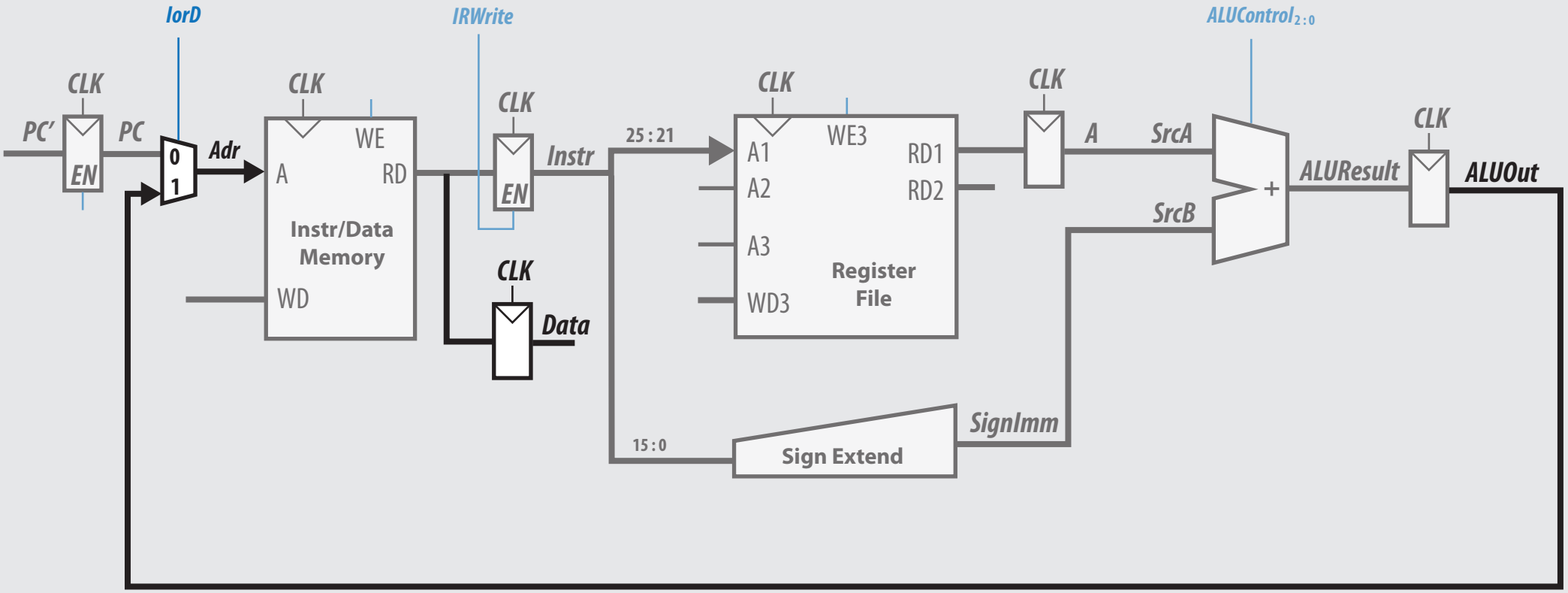


Step 3 : read data from memory and store it



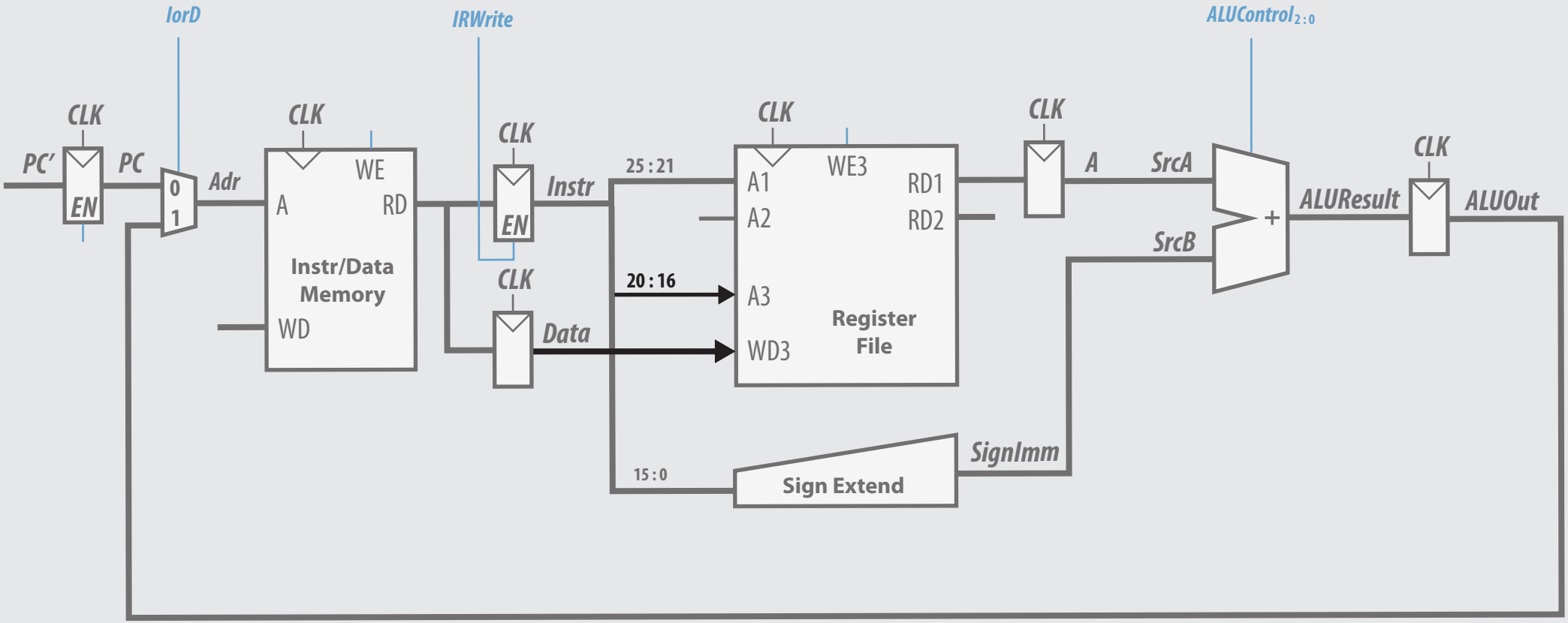
The next step is to load the data from the calculated address in the memory. We add a multiplexer in front of the memory to choose the memory address ***Adr*** from either the ***PC*** or ***ALUOut***. The multiplexer select signal is called ***IorD*** to indicate either an instruction or data address.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

Step 3 : read data from memory and store it

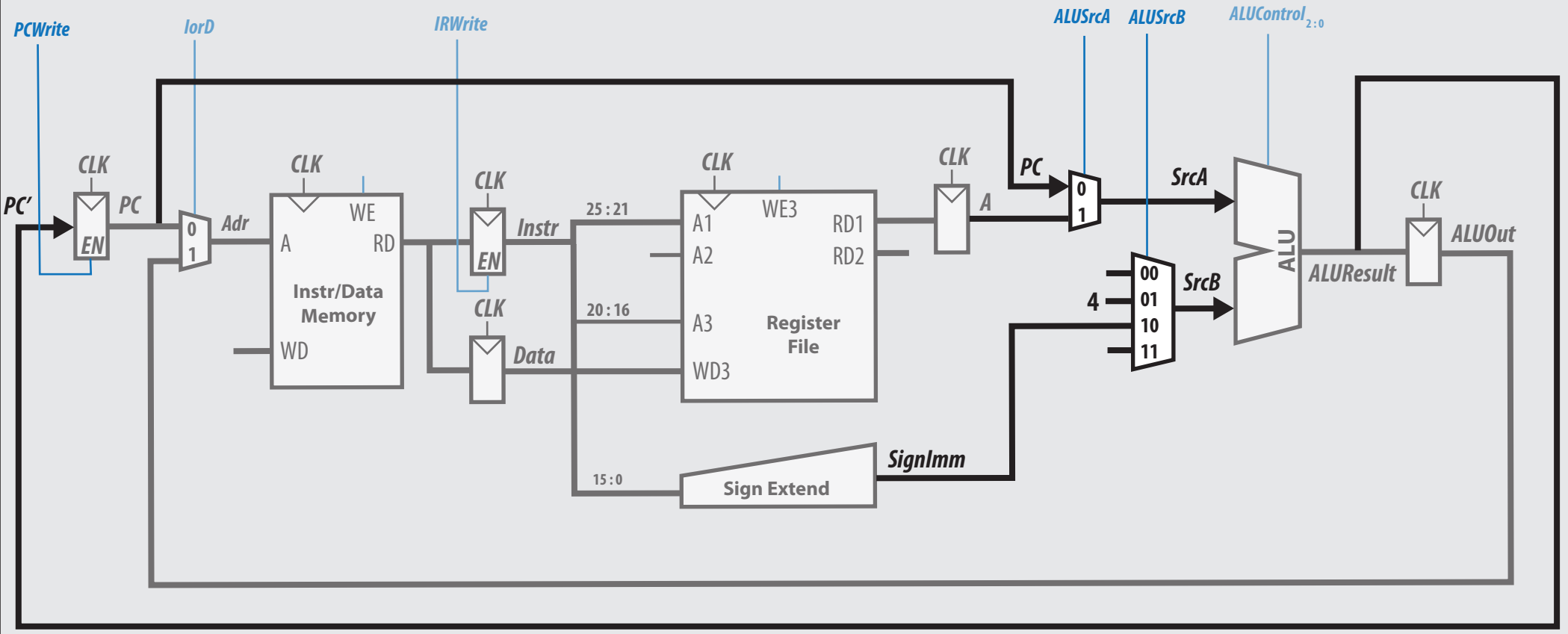


Notice that the address multiplexer enables us to reuse the memory during the `lw` instruction. On the first step, the address is taken from the ***PC*** to fetch the instruction. On the later step, the address is taken from ***ALUOut*** to read the data from memory. Hence, the control signal ***IorD***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

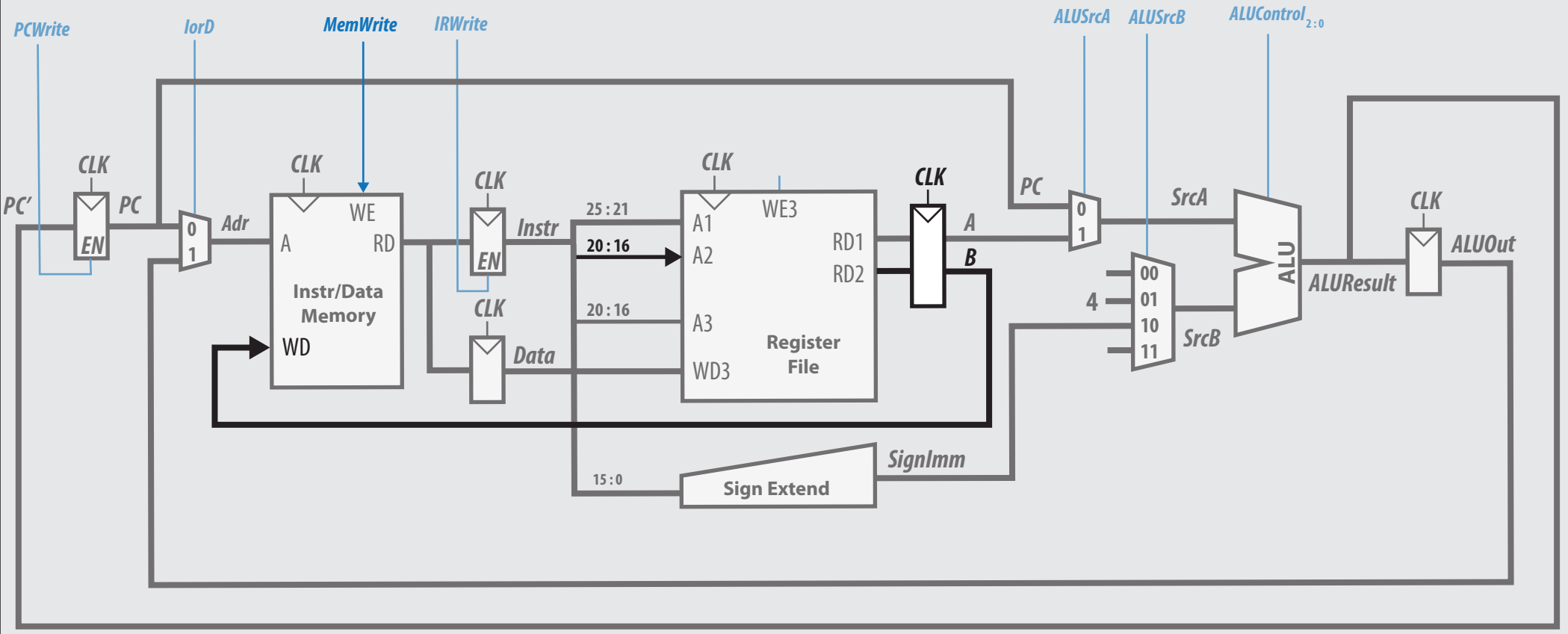
Step 4 : write back data to register file



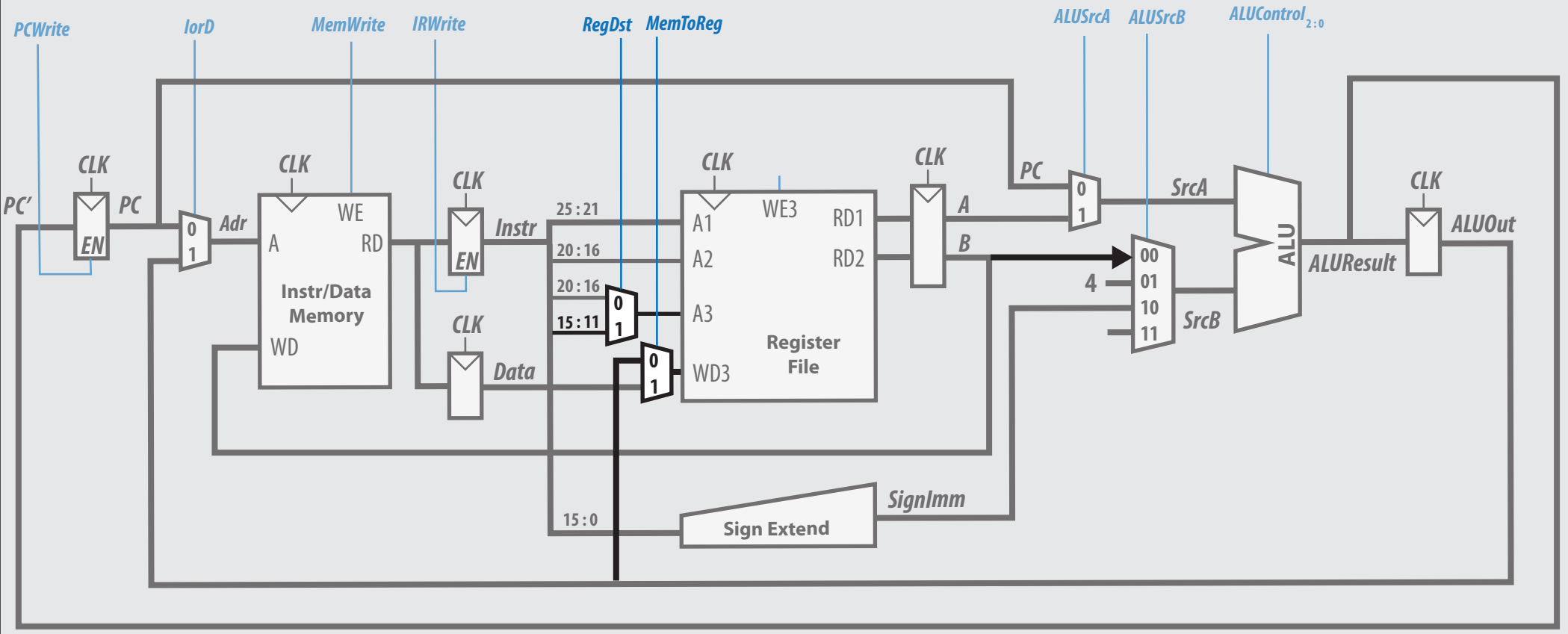
Should not be forgotten : increment PC by 4



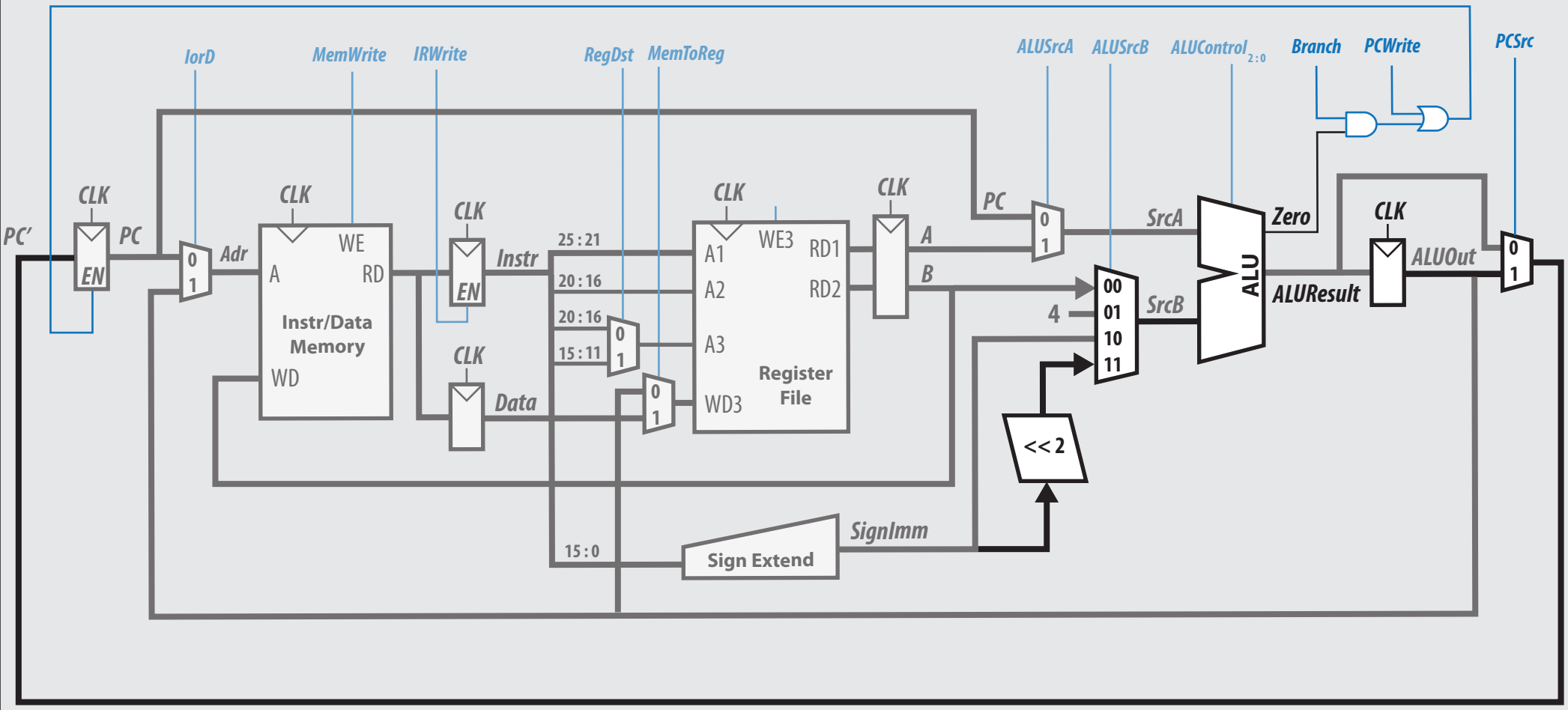
Enhanced data path for the sw instruction



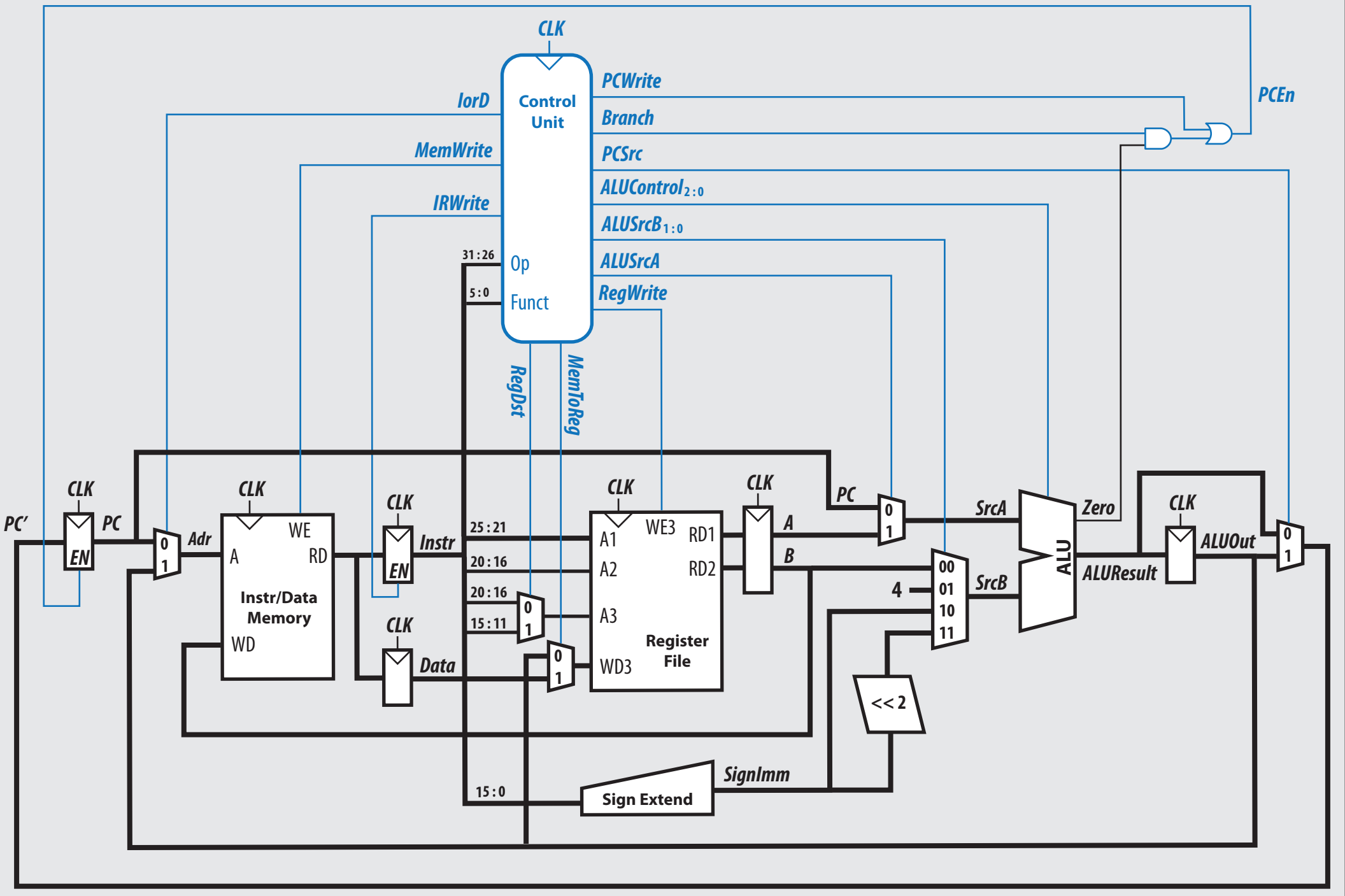
Enhanced data path for R-type instructions



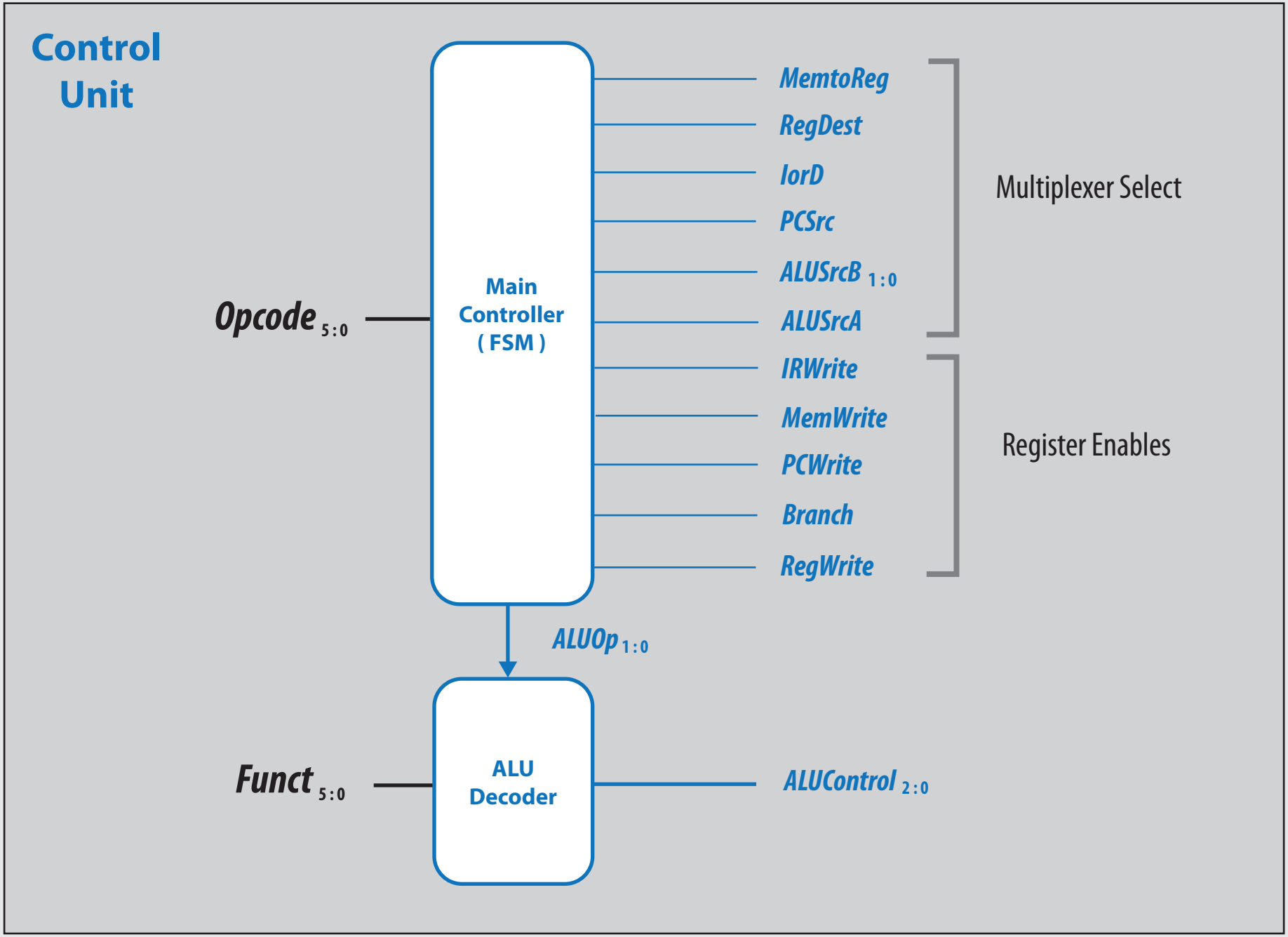
Enhanced data path for the beq instruction



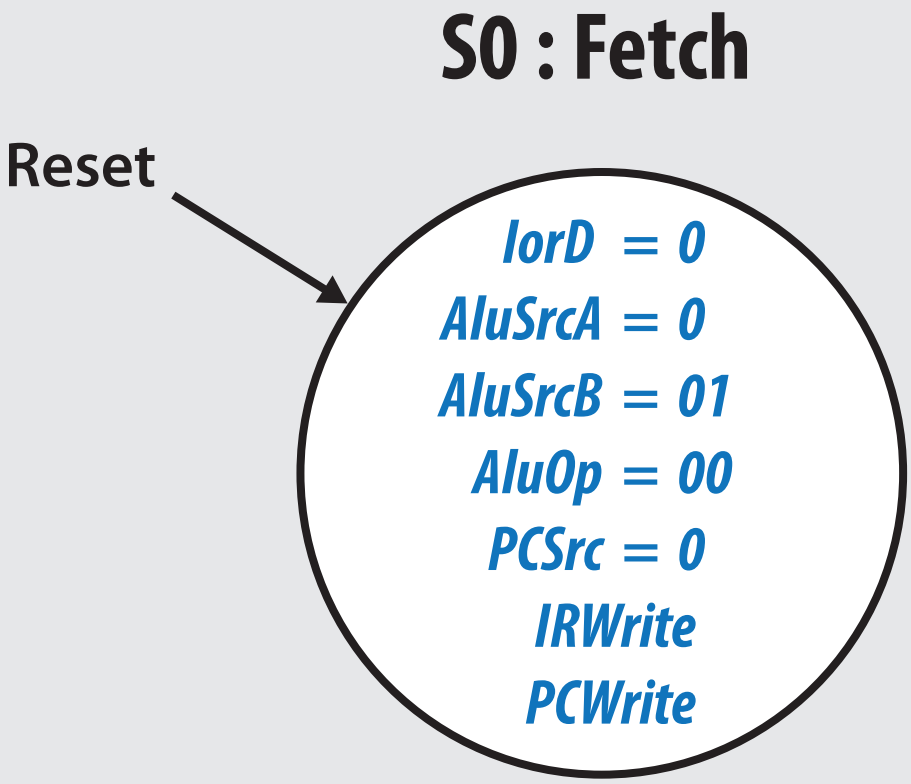
Complete multicycle MIPS processor



Control unit internal structure

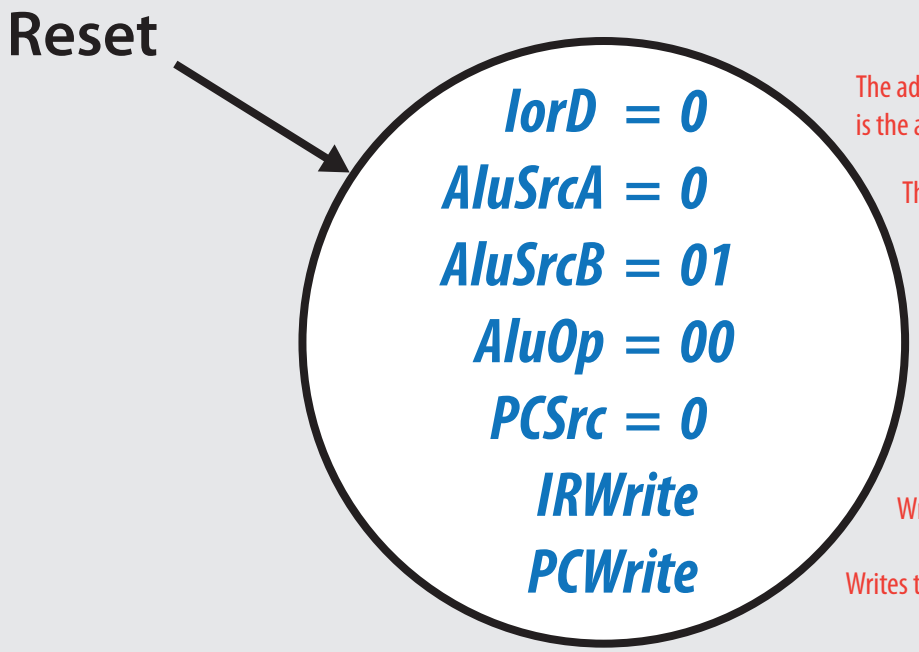


Fetch


S0 : Fetch
Reset
IorD = 0
AluSrcA = 0
AluSrcB = 01
AluOp = 00
PCSrc = 0
IRWrite
PCWrite

Fetch

S0 : Fetch



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

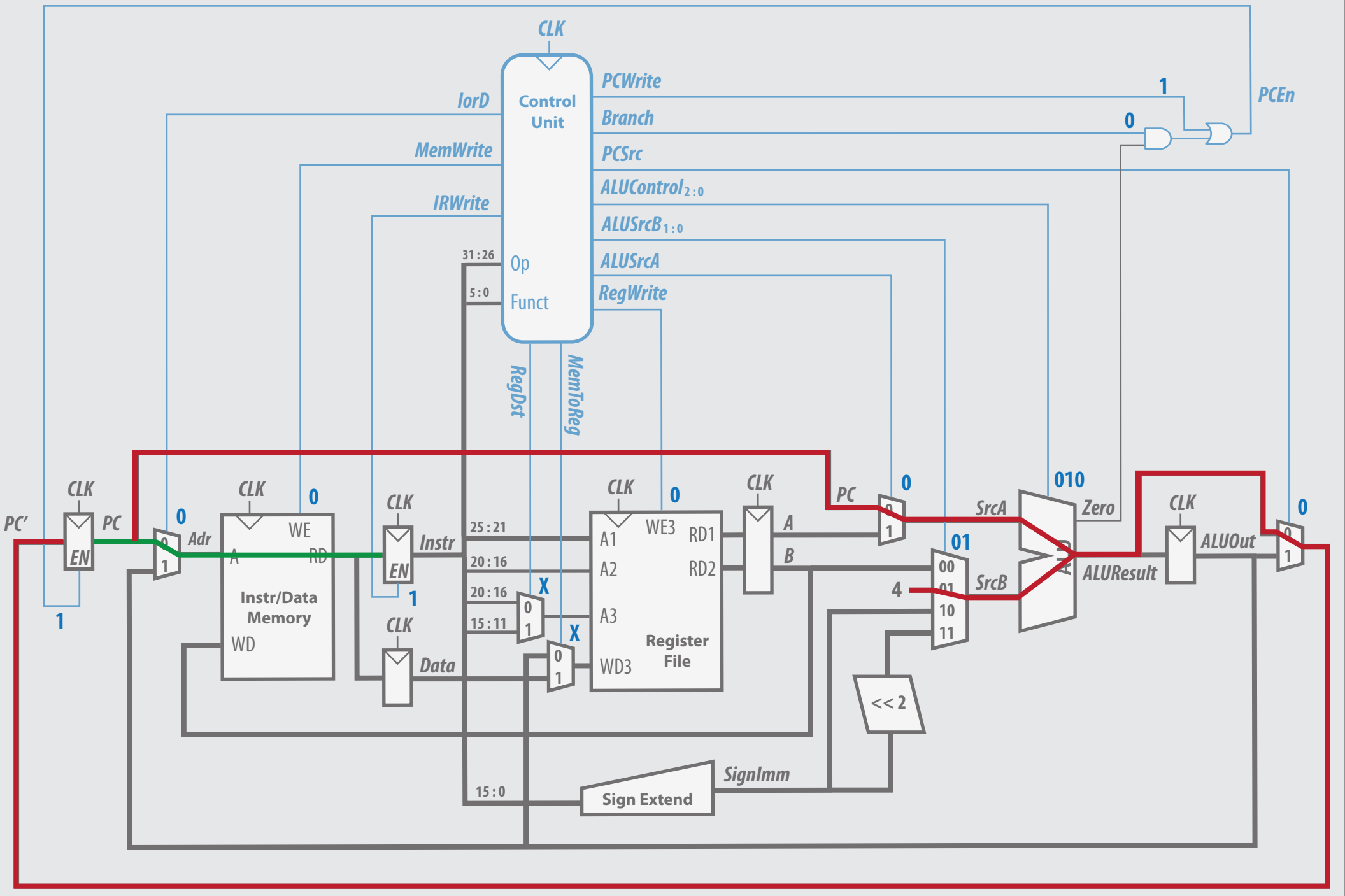
The ALU performs an addition PC′ = PC + 4 to compute the next Program Counter (PC′)

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

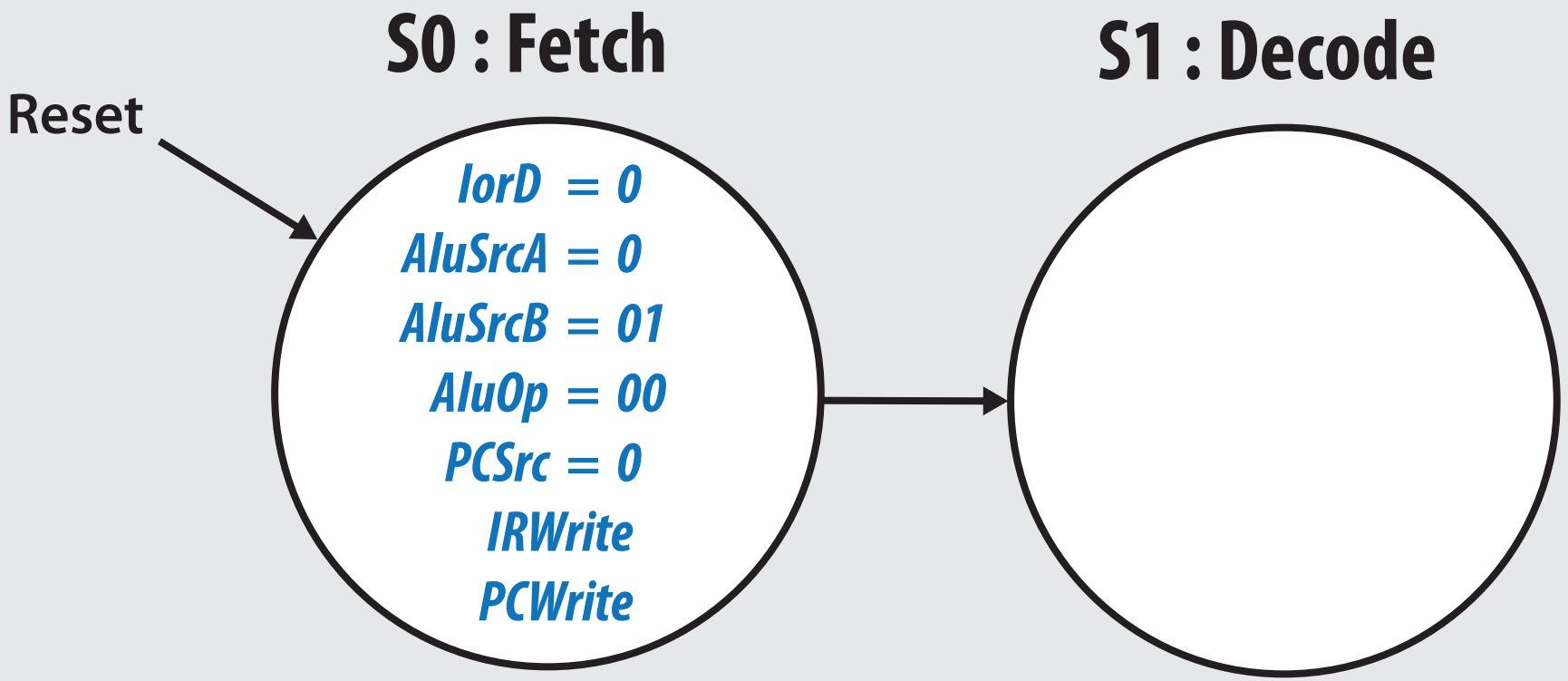
Writes the current instruction in the Instruction Register (IR)

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

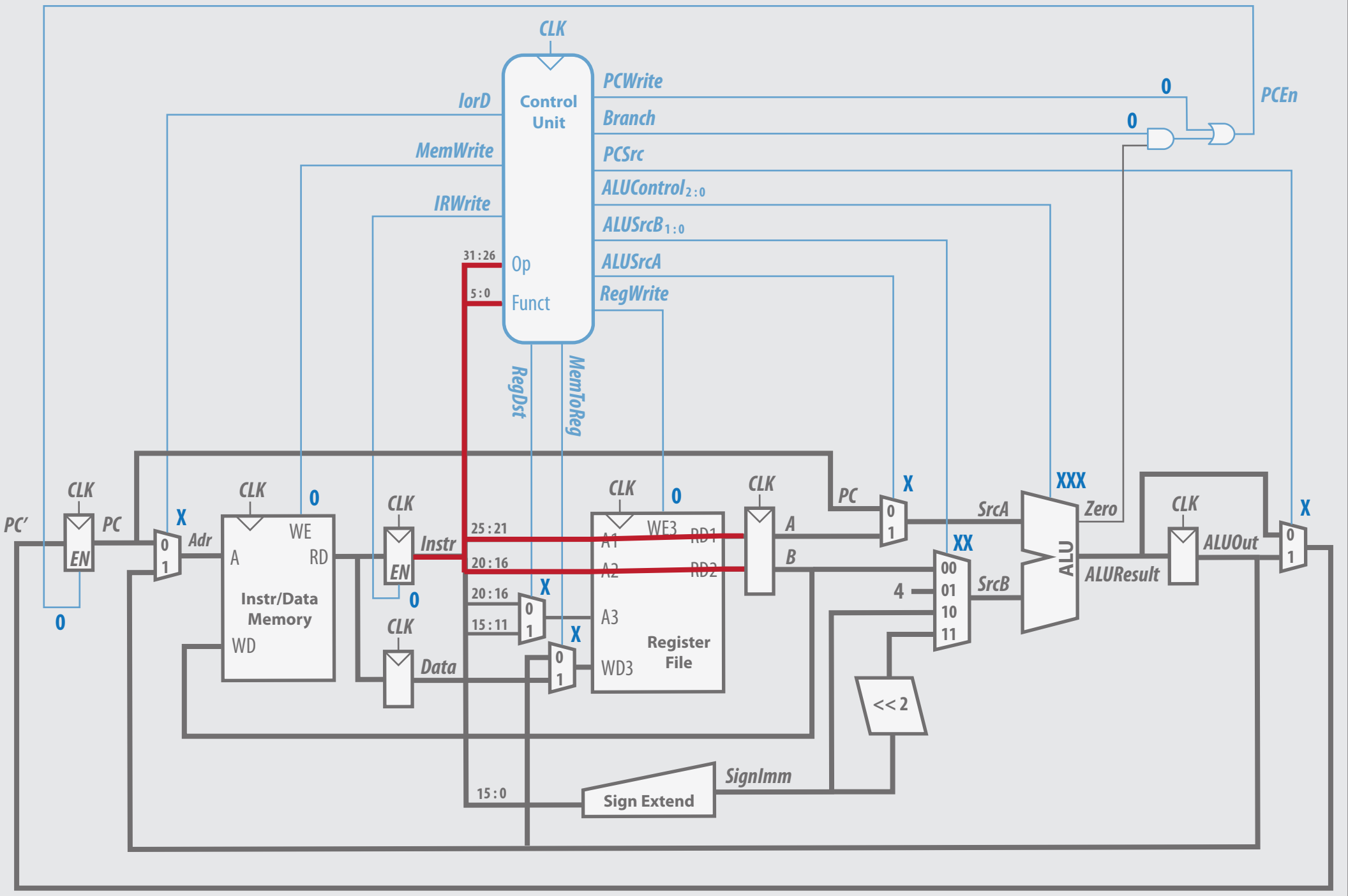
Data flow during the fetch step



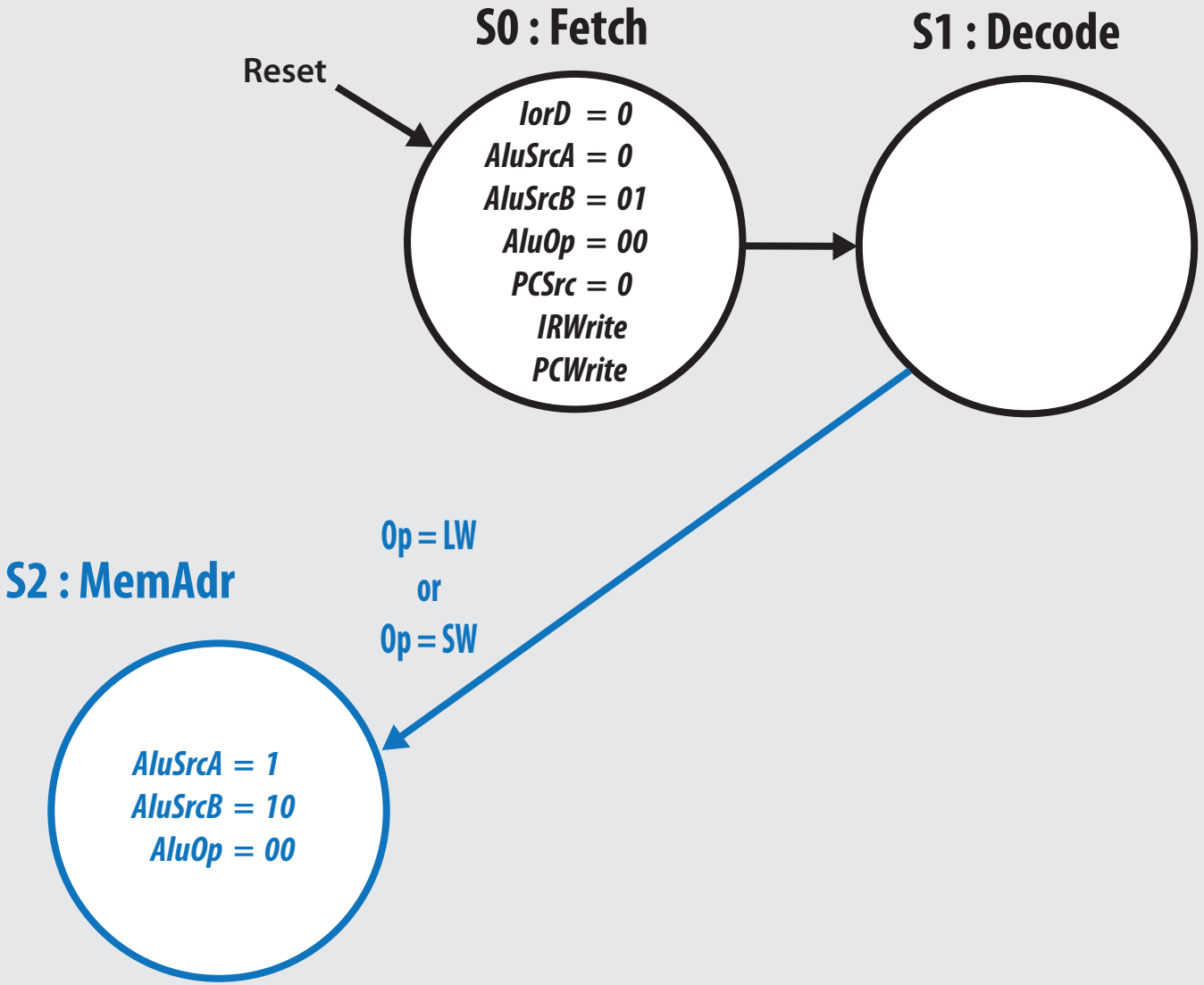
Decode


S0 : Fetch
S1 : Decode
Reset
IorD = 0
AluSrcA = 0
AluSrcB = 01
AluOp = 00
PCSrc = 0
IRWrite
PCWrite

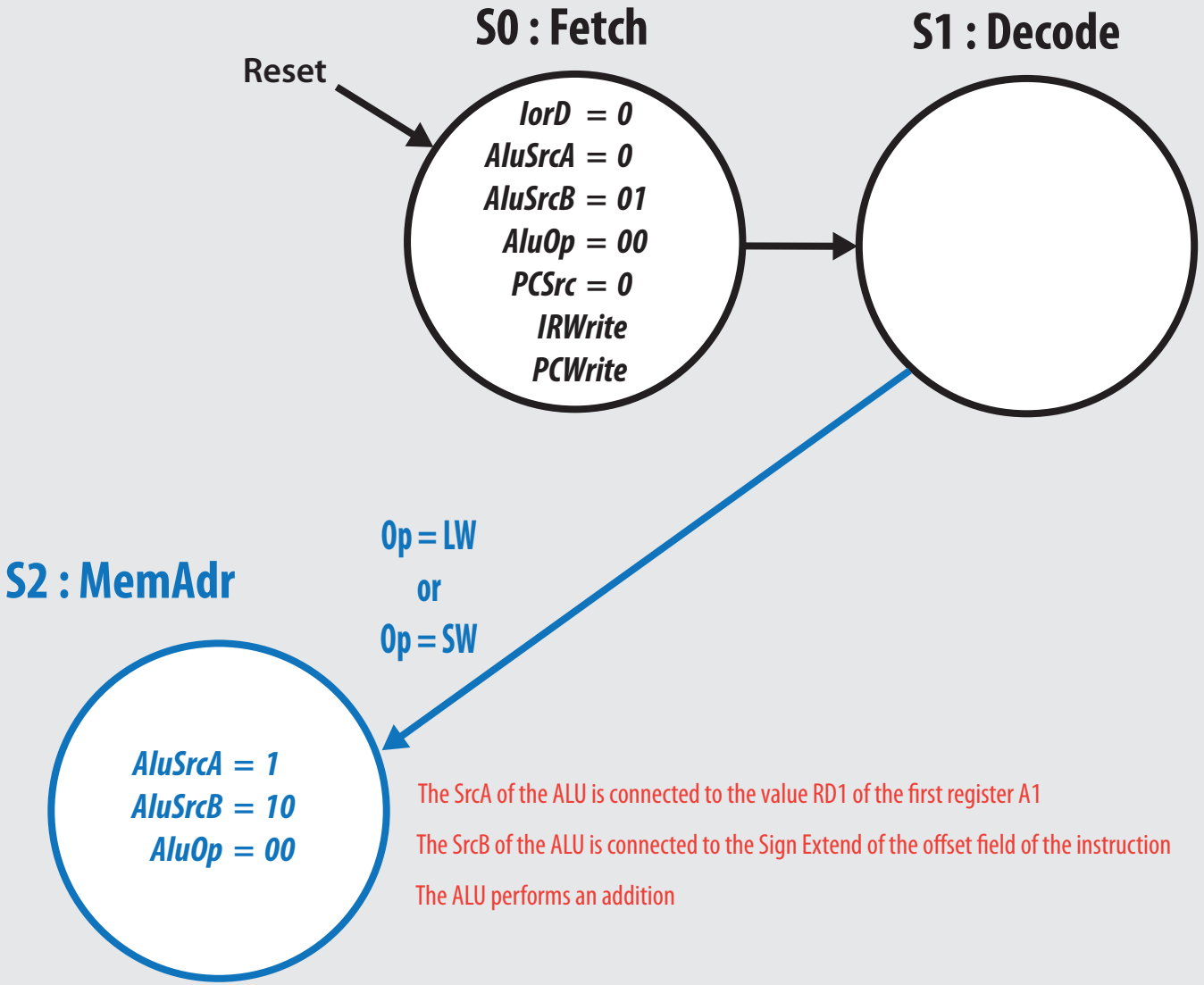
Data flow during the decode step



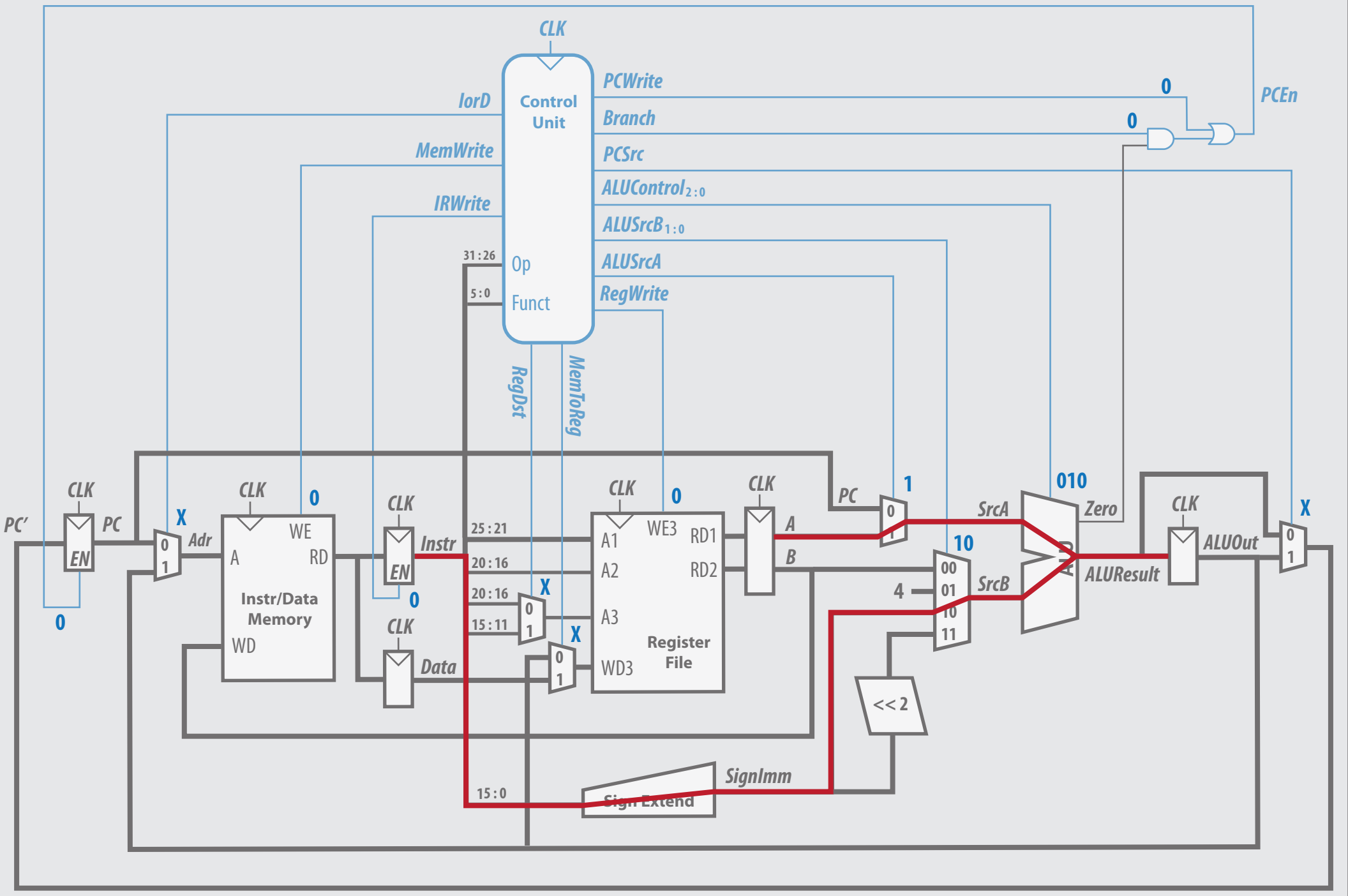
Memory Address Computation



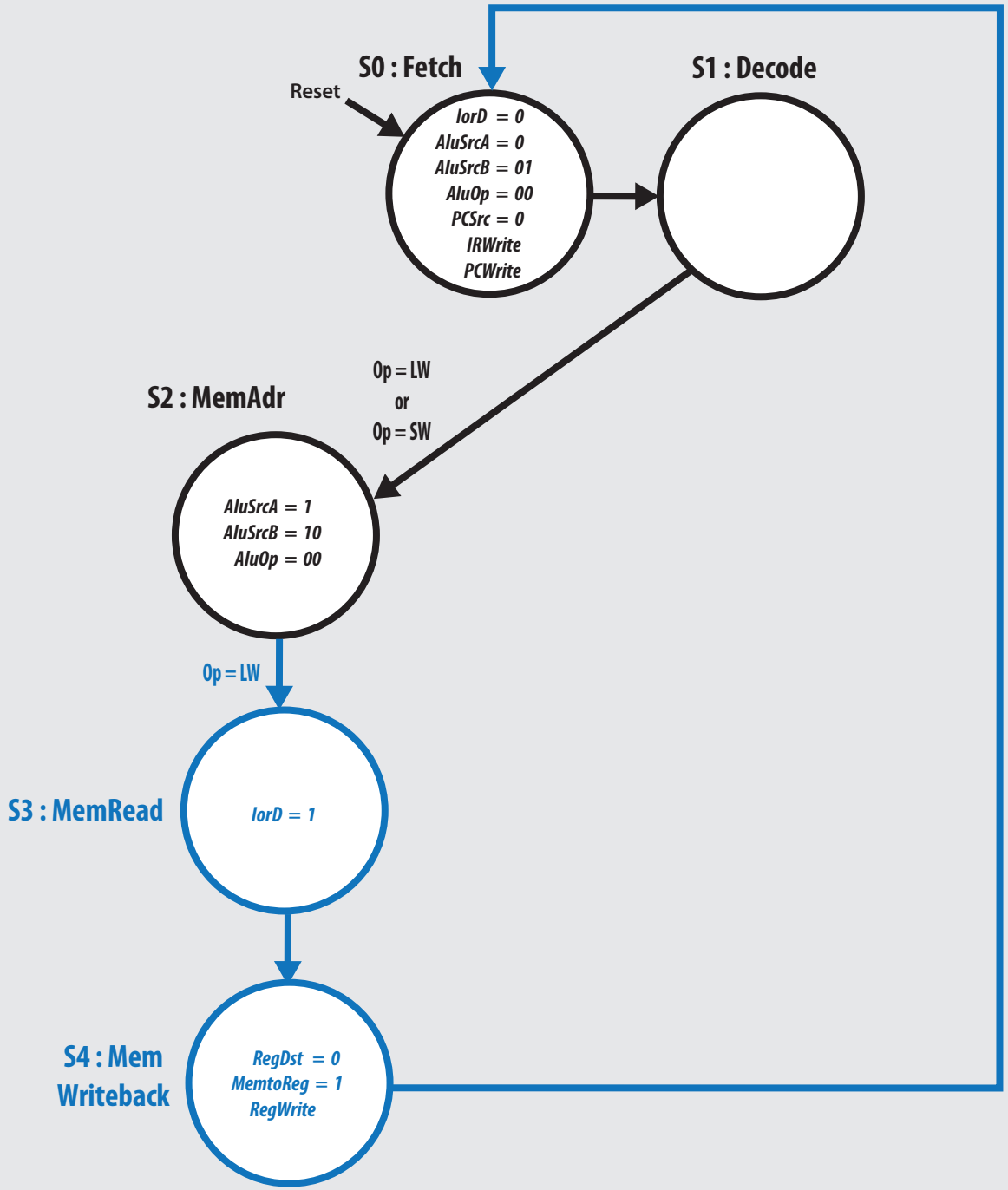
Memory Address Computation



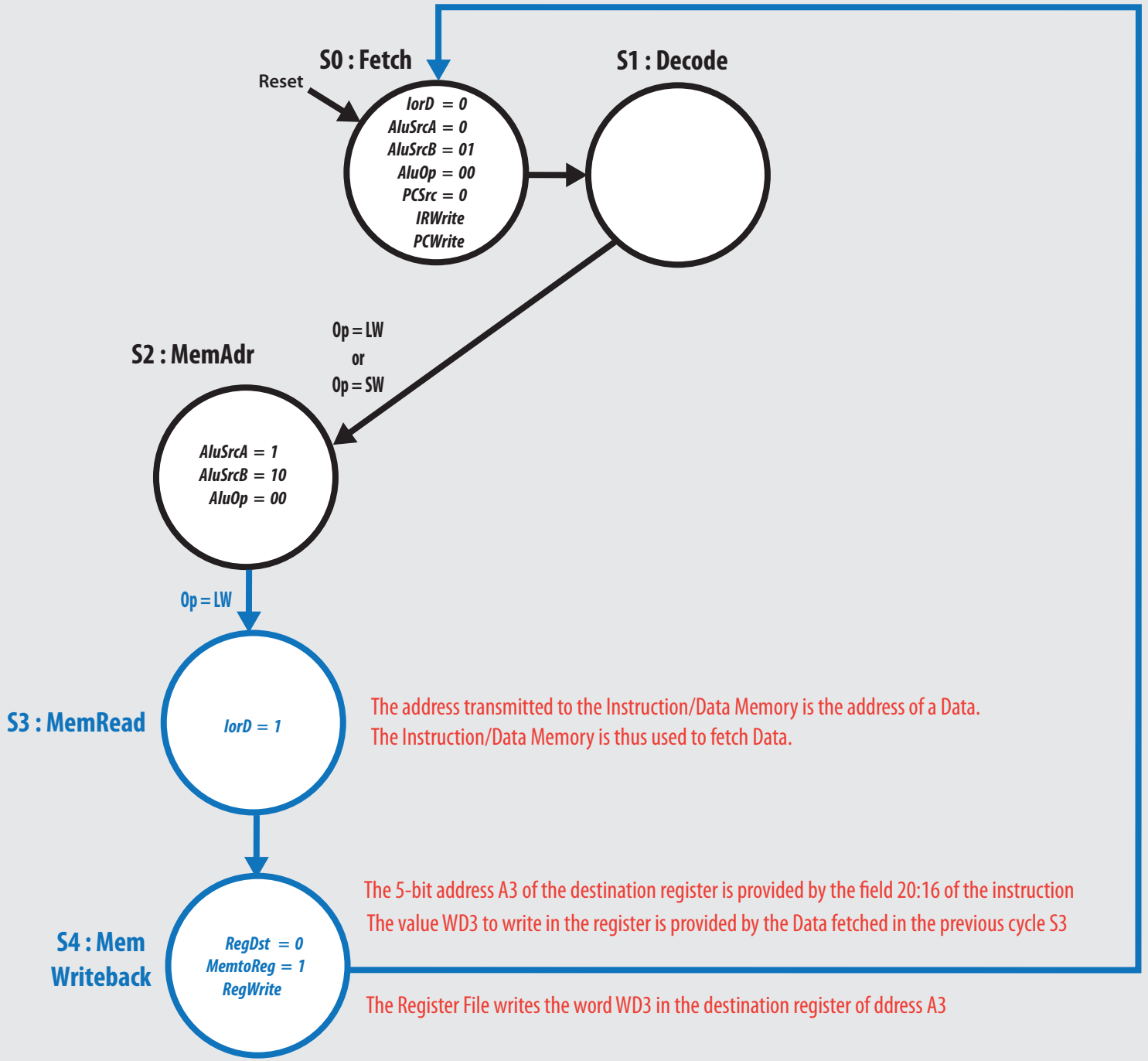
Data flow during memory address computation



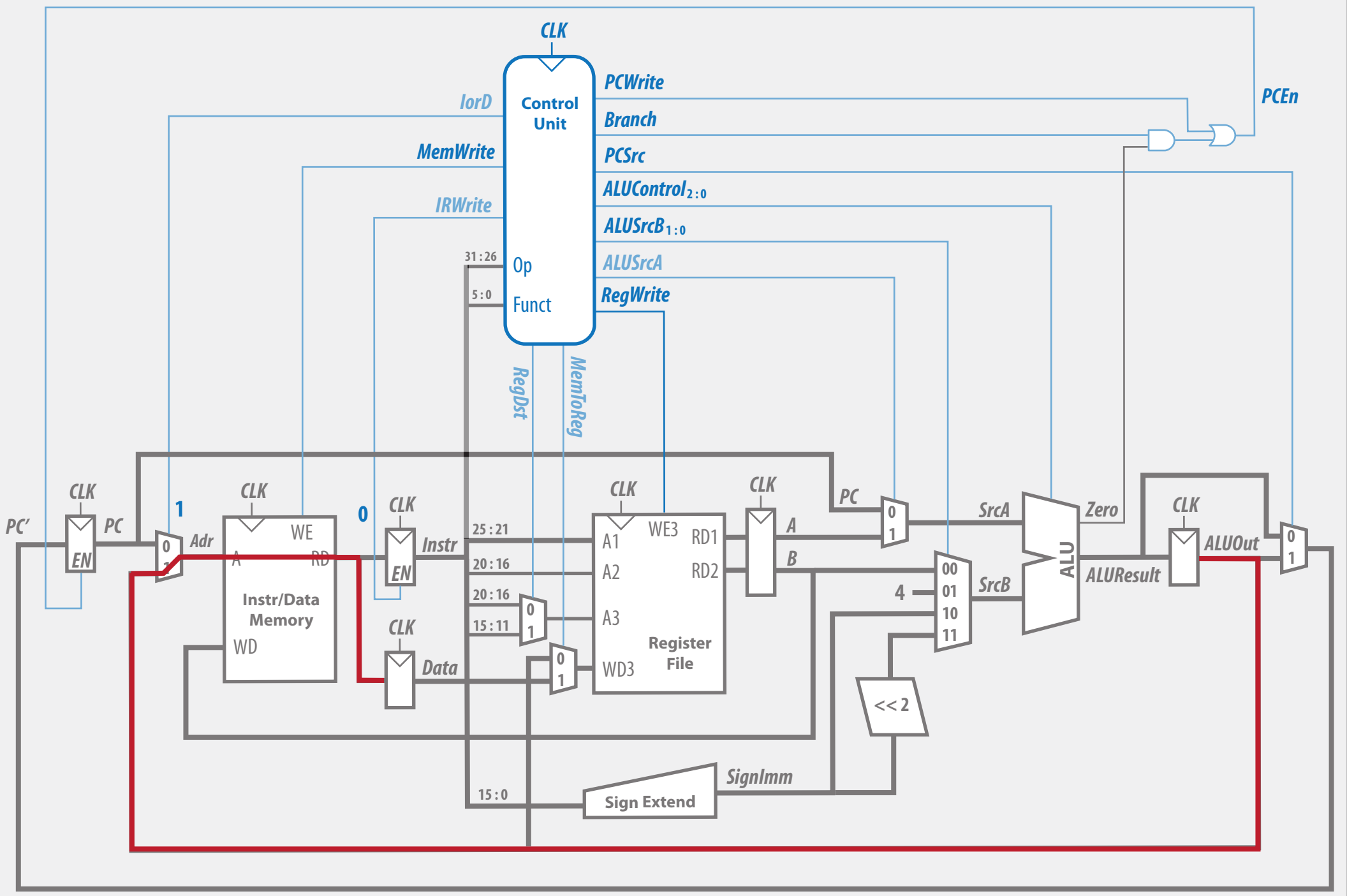
Memory Read


S0 : Fetch
Reset
IorD = 0
AluSrcA = 0
AluSrcB = 01
AluOp = 00
PCSrc = 0
IRWrite
PCWrite
S1 : Decode
Op = LW
or
Op = SW
S2 : MemAdr
AluSrcA = 1
AluSrcB = 10
AluOp = 00
Op = LW
S3 : MemRead
IorD = 1
S4 : Mem
Writeback
RegDst = 0
MemtoReg = 1
RegWrite

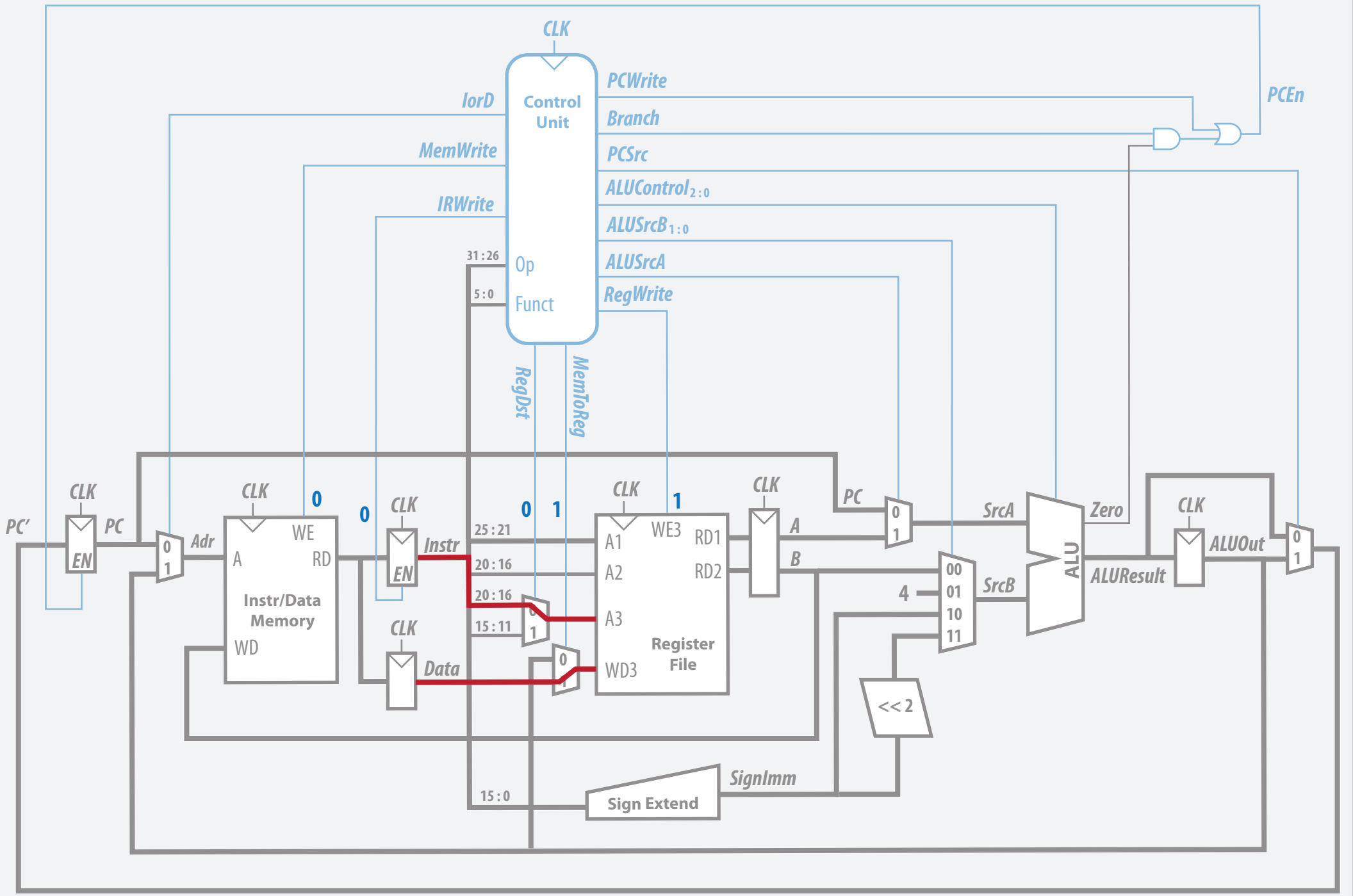
Memory Read


S0 : Fetch
Reset
IorD = 0
AluSrcA = 0
AluSrcB = 01
AluOp = 00
PCSrc = 0
IRWrite
PCWrite
S1 : Decode
Op = LW
or
Op = SW
S2 : MemAdr
AluSrcA = 1
AluSrcB = 10
AluOp = 00
Op = LW
S3 : MemRead
IorD = 1
The address transmitted to the Instruction/Data Memory is the address of a Data.
The Instruction/Data Memory is thus used to fetch Data.
S4 : Mem Writeback
RegDst = 0
MemtoReg = 1
RegWrite
The 5-bit address A3 of the destination register is provided by the field 20:16 of the instruction
The value WD3 to write in the register is provided by the Data fetched in the previous cycle S3
The Register File writes the word WD3 in the destination register of ddress A3

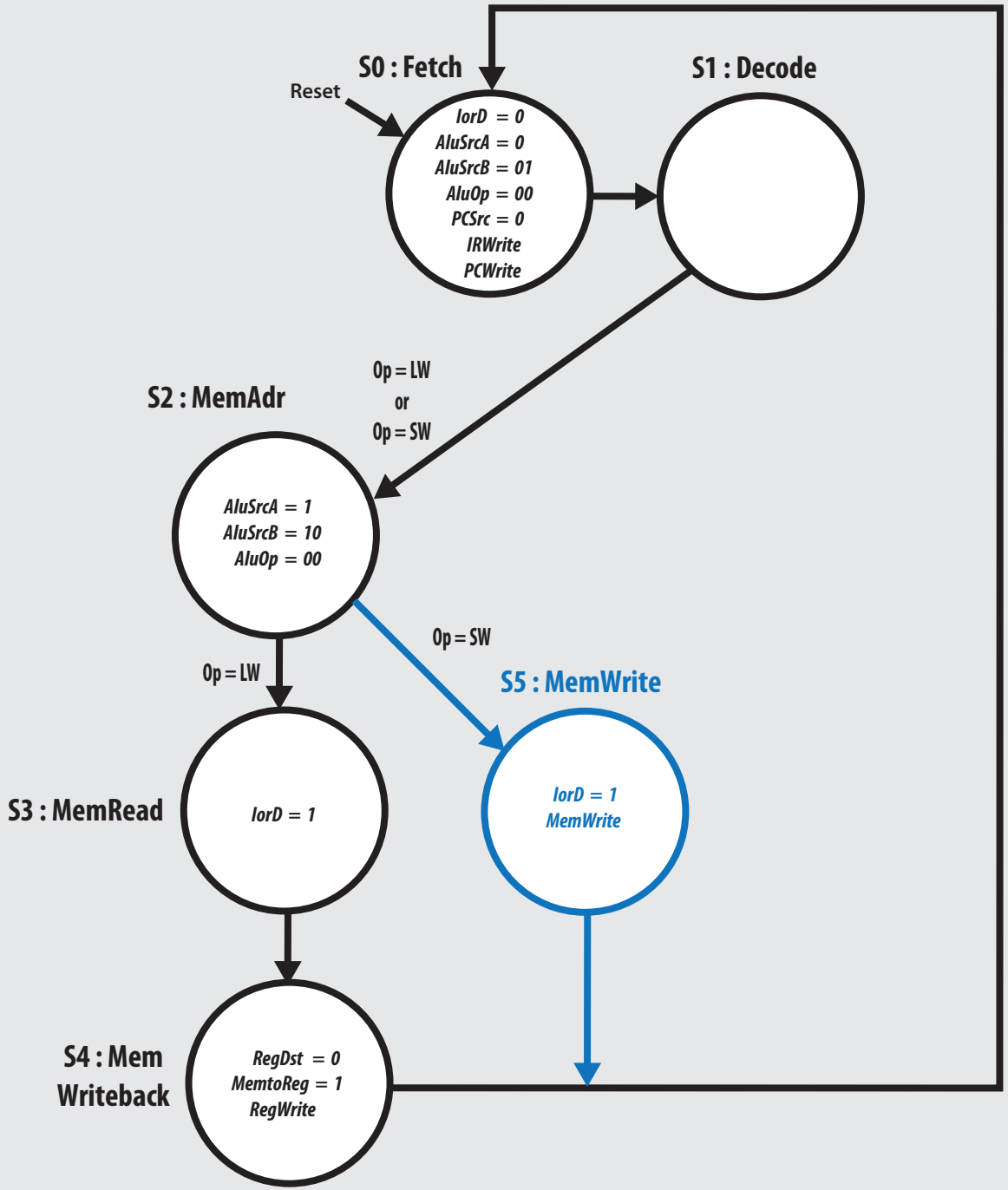
Data flow during the Memory Read



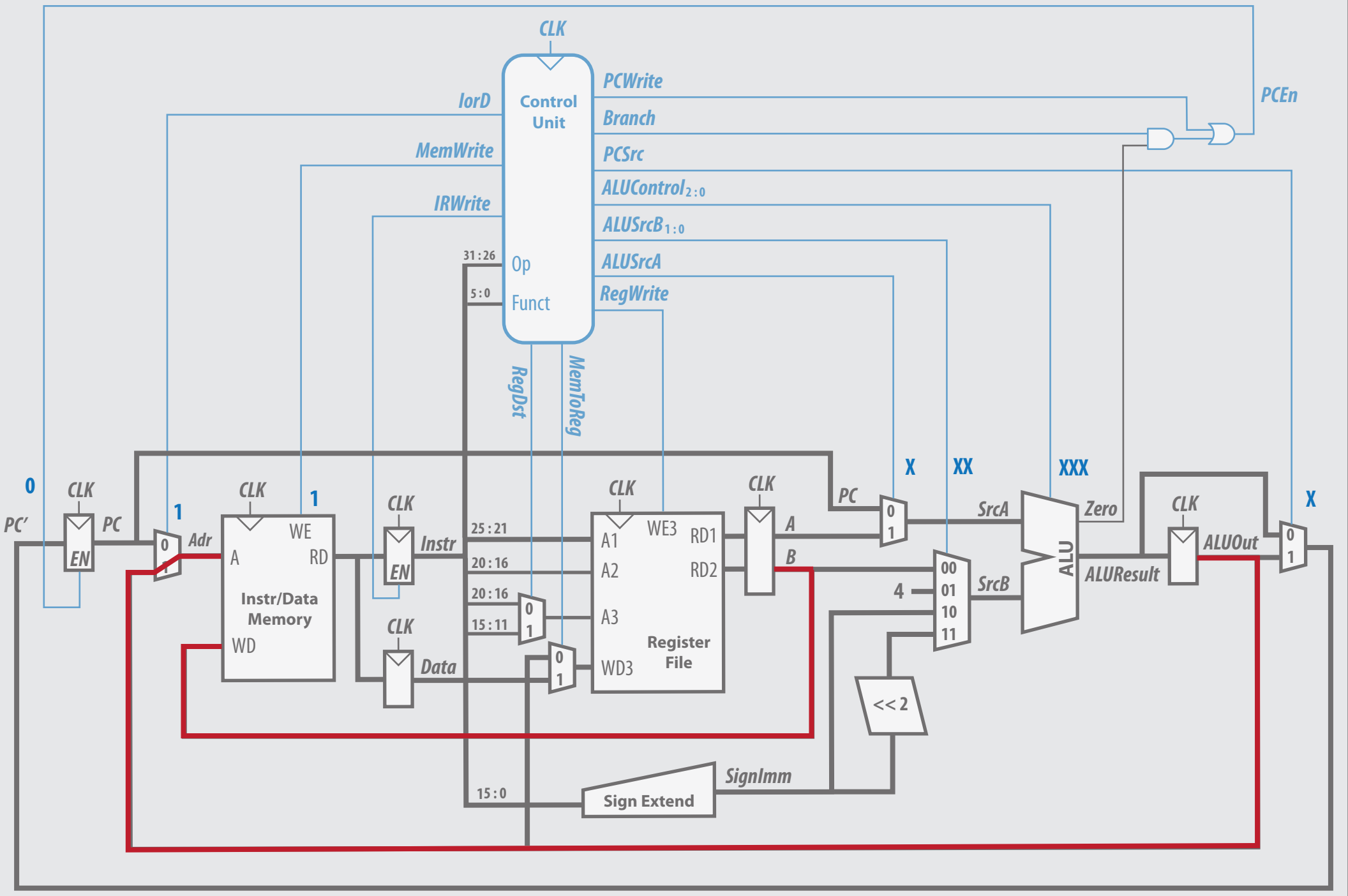
Data flow during the Writeback



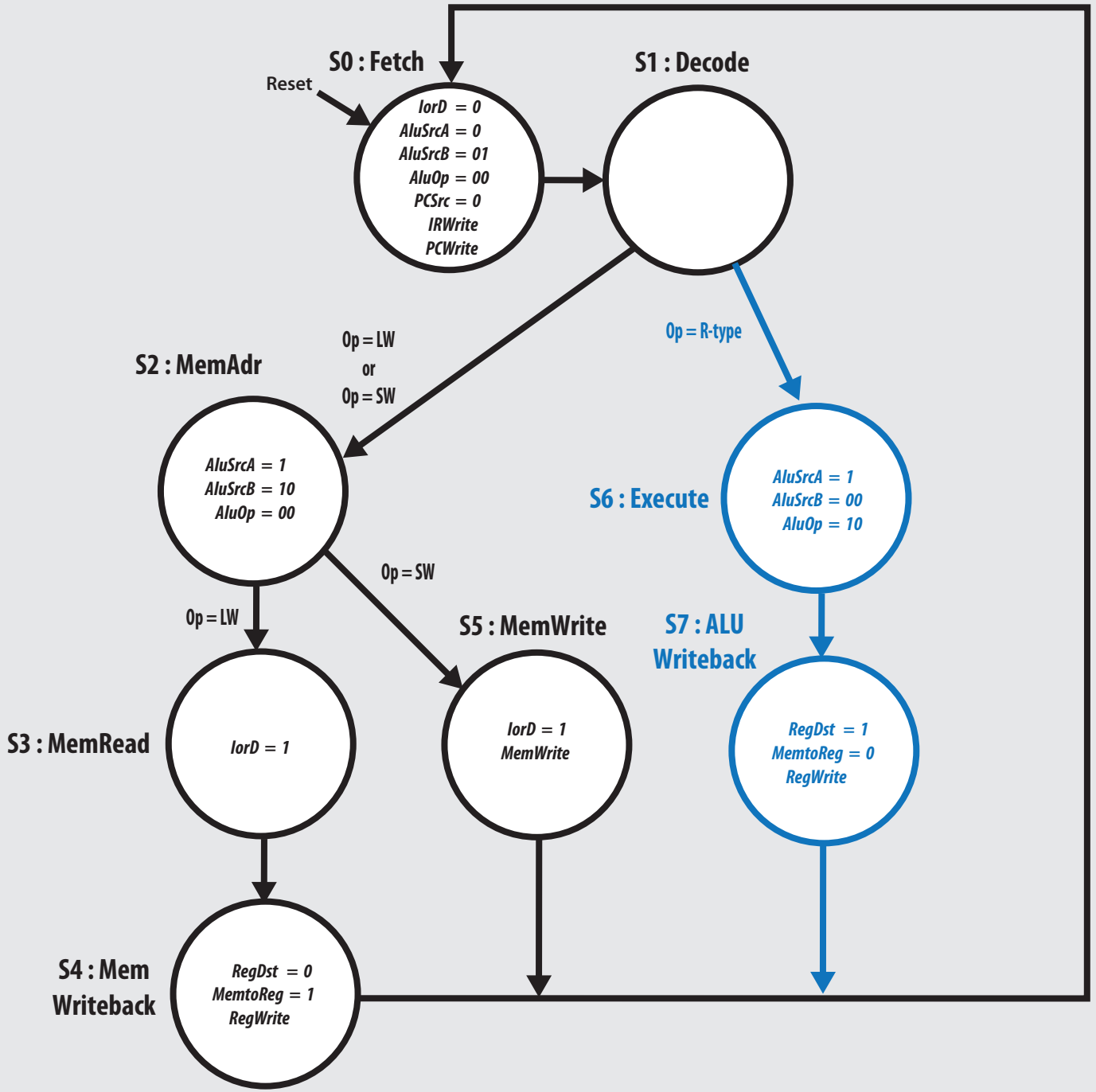
Memory Write



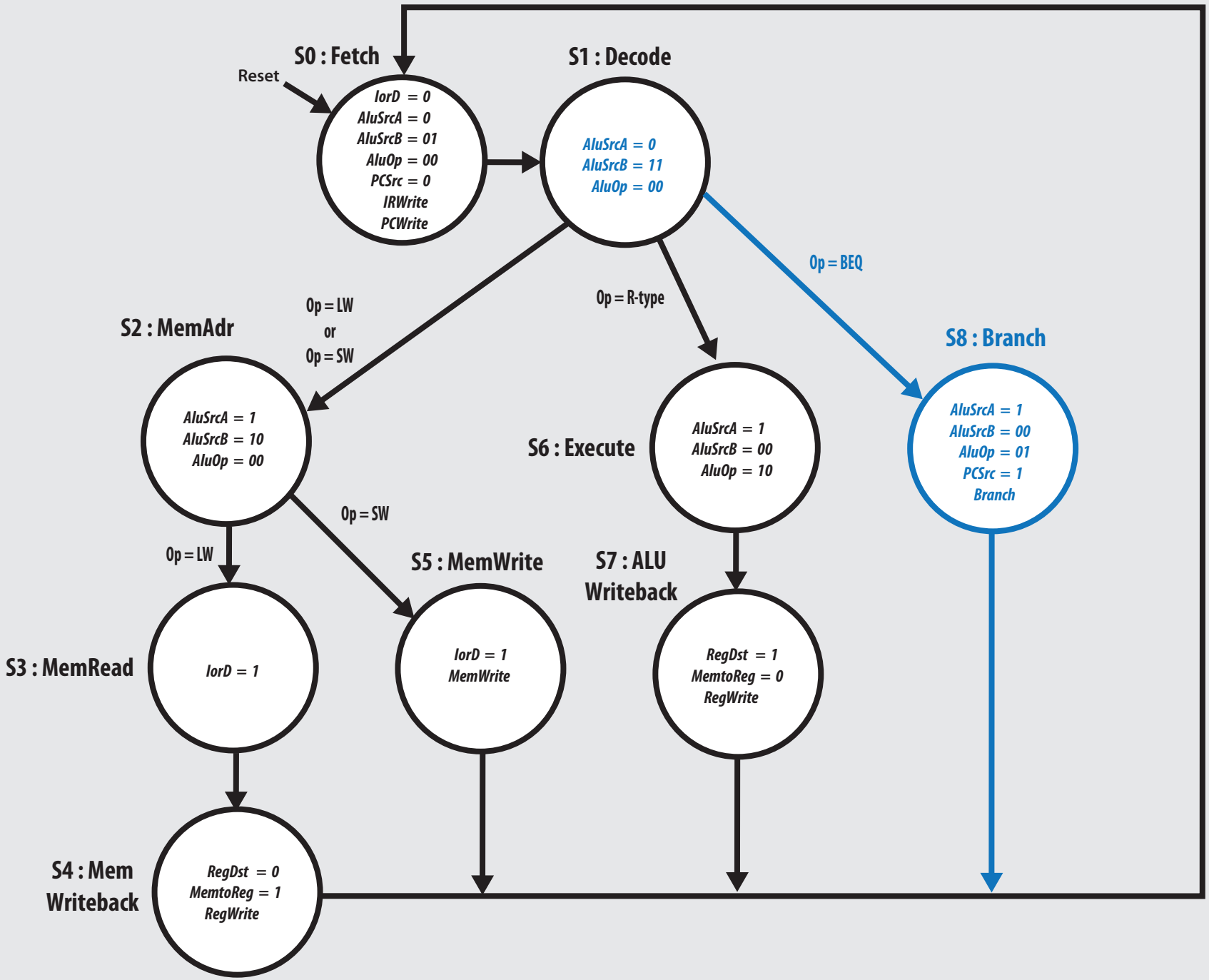
Data flow during the Memory Write step



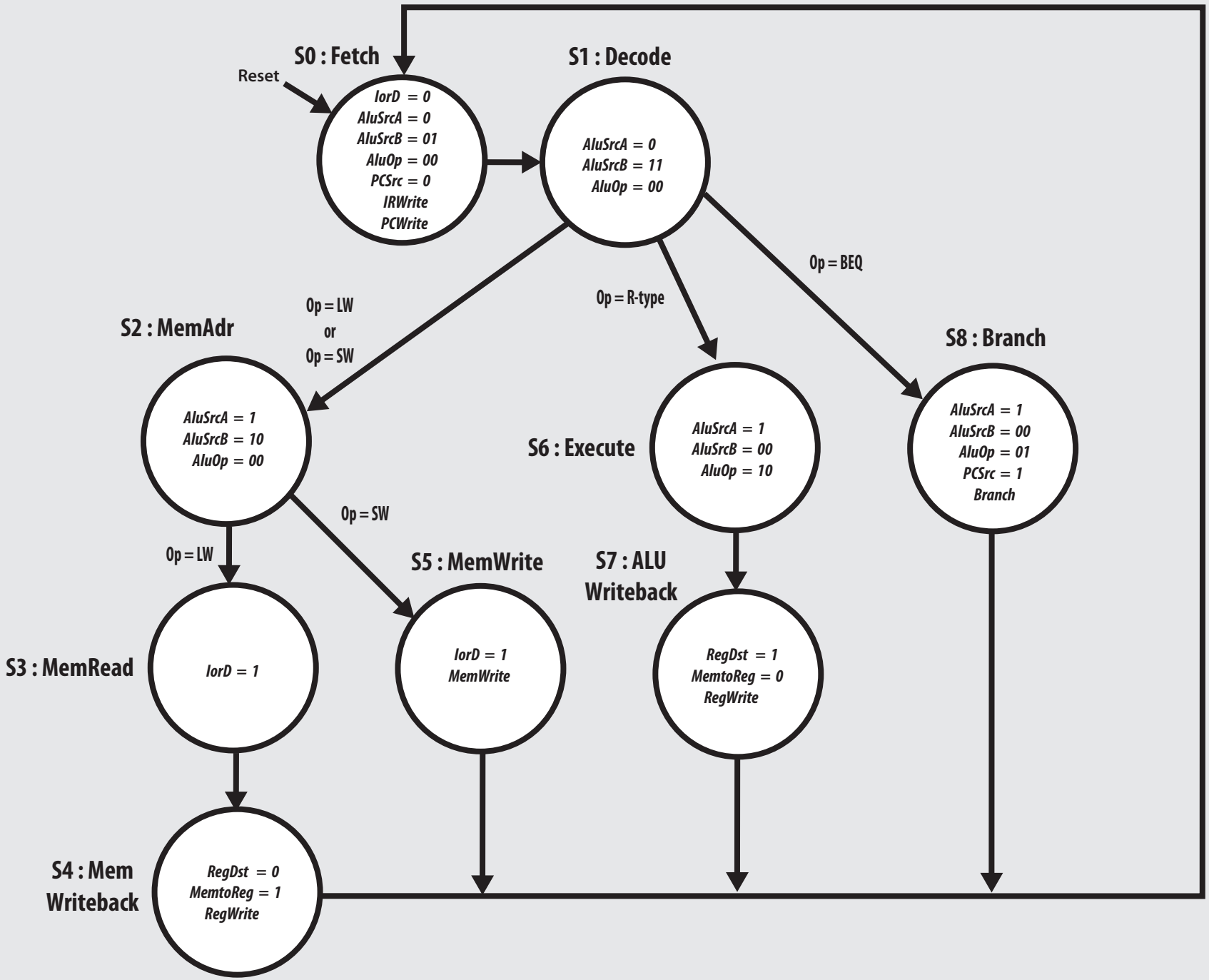
Execute R-type operation



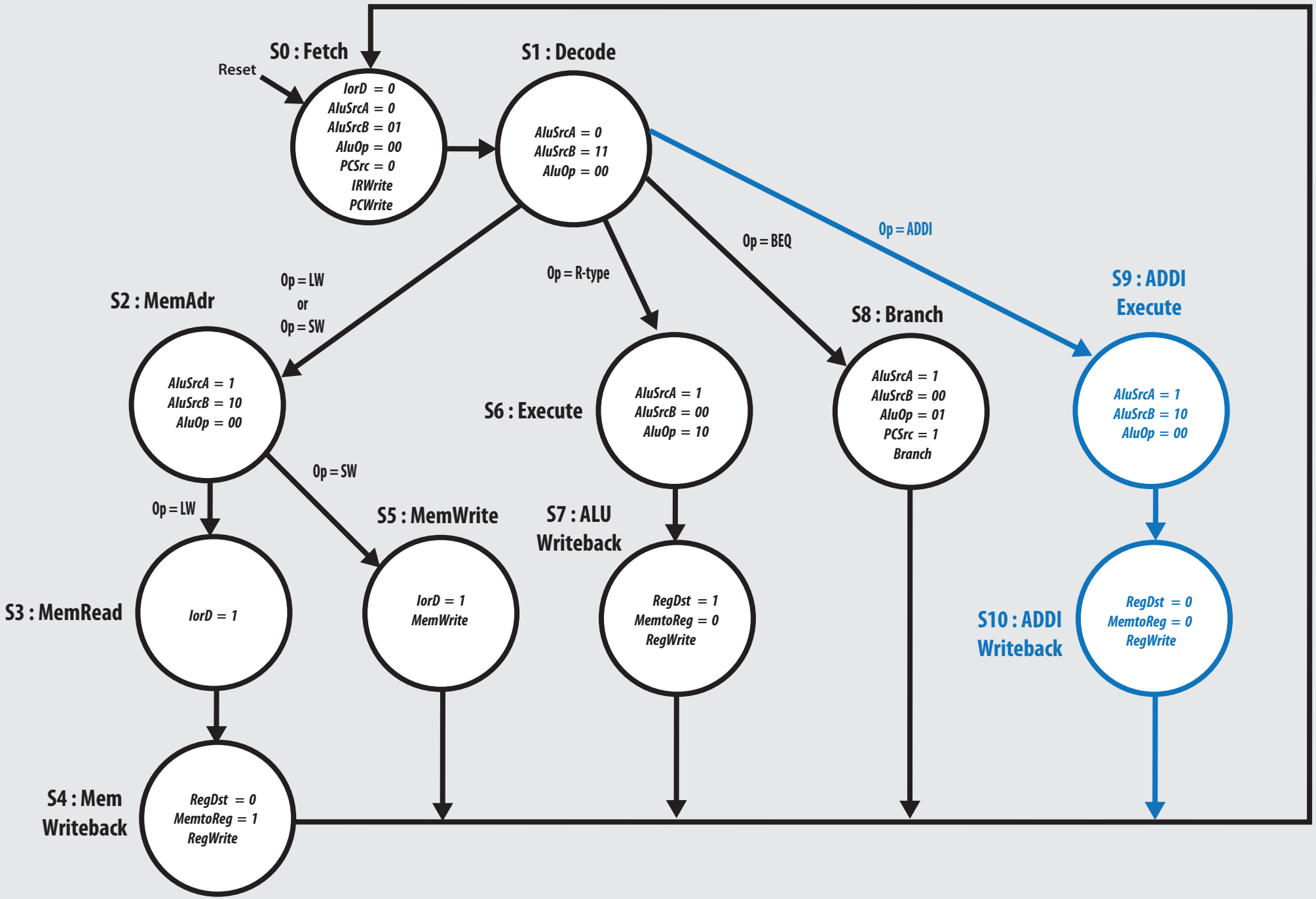
Branch


S0 : Fetch
Reset
IorD = 0
AluSrcA = 0
AluSrcB = 01
AluOp = 00
PCSrc = 0
IRWrite
PCWrite
S1 : Decode
AluSrcA = 0
AluSrcB = 11
AluOp = 00
Op = BEQ
Op = LW
or
Op = SW
Op = R-type
S2 : MemAdr
AluSrcA = 1
AluSrcB = 10
AluOp = 00
S8 : Branch
AluSrcA = 1
AluSrcB = 00
AluOp = 01
PCSrc = 1
Branch
S6 : Execute
AluSrcA = 1
AluSrcB = 00
AluOp = 10
Op = SW
Op = LW
S5 : MemWrite
IorD = 1
MemWrite
S7 : ALU
Writeback
RegDst = 1
MemtoReg = 0
RegWrite
S3 : MemRead
IorD = 1
S4 : Mem
Writeback
RegDst = 0
MemtoReg = 1
RegWrite

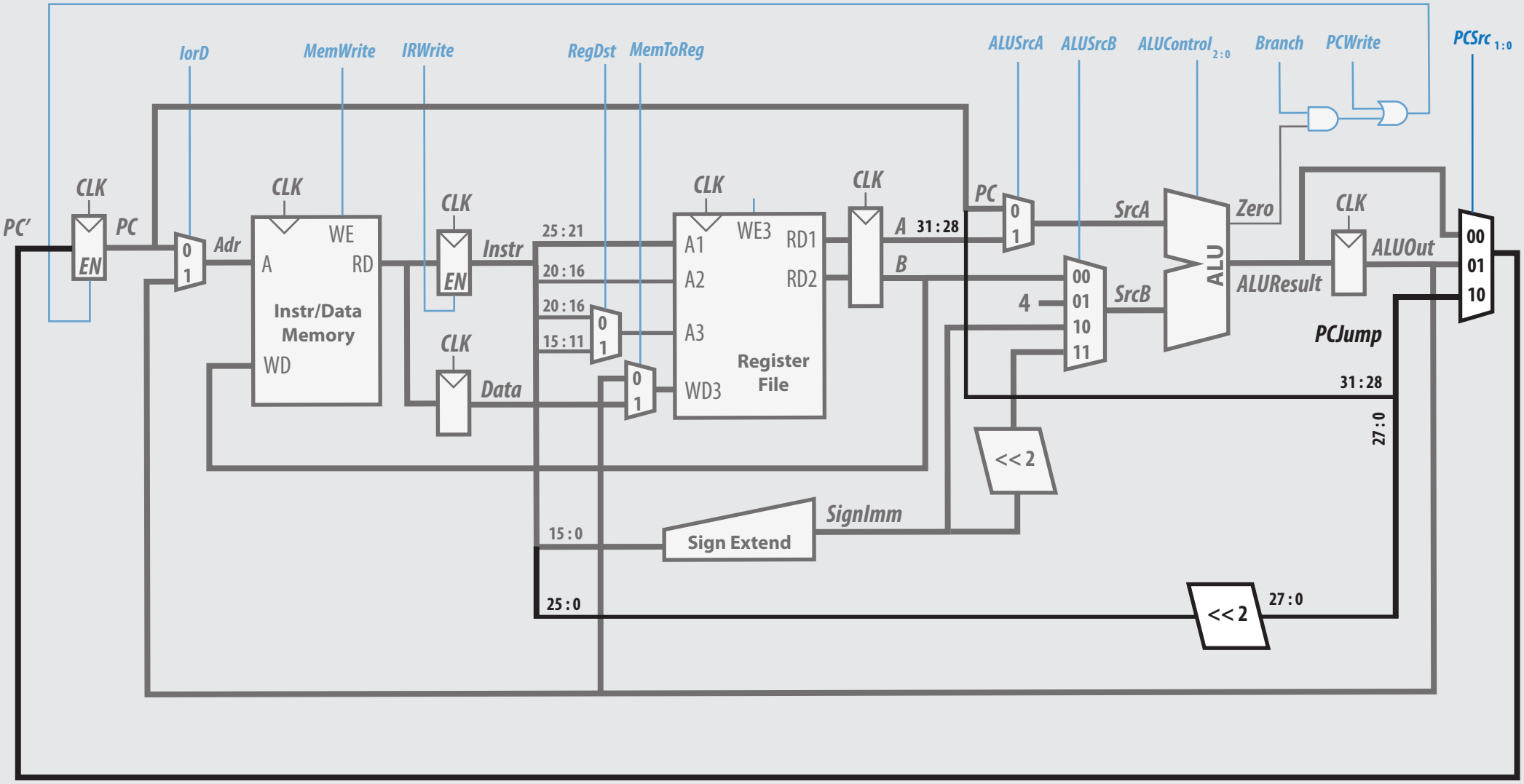
Complete multicycle control FSM



Main controller states for `addi`



Multicycle MIPS datapath enhanced to support the `j` instruction



Main controller state for j