Computer organization and architecture

Lesson 15

Multicycle processor

Three weaknesses of the single-cycle processor:

1) It requires separate memories for instructions and data

Most processors only have a single external memory

2) It requires a clock cycle long enough to support the slowest instruction

Even though most instructions could be faster.

3) It requires three adders (one in the ALU and two for the PC logic)

Adders are expensive, especially if they must be fast.

The **multicycle processor** breaks an instruction into multiple shorter steps.

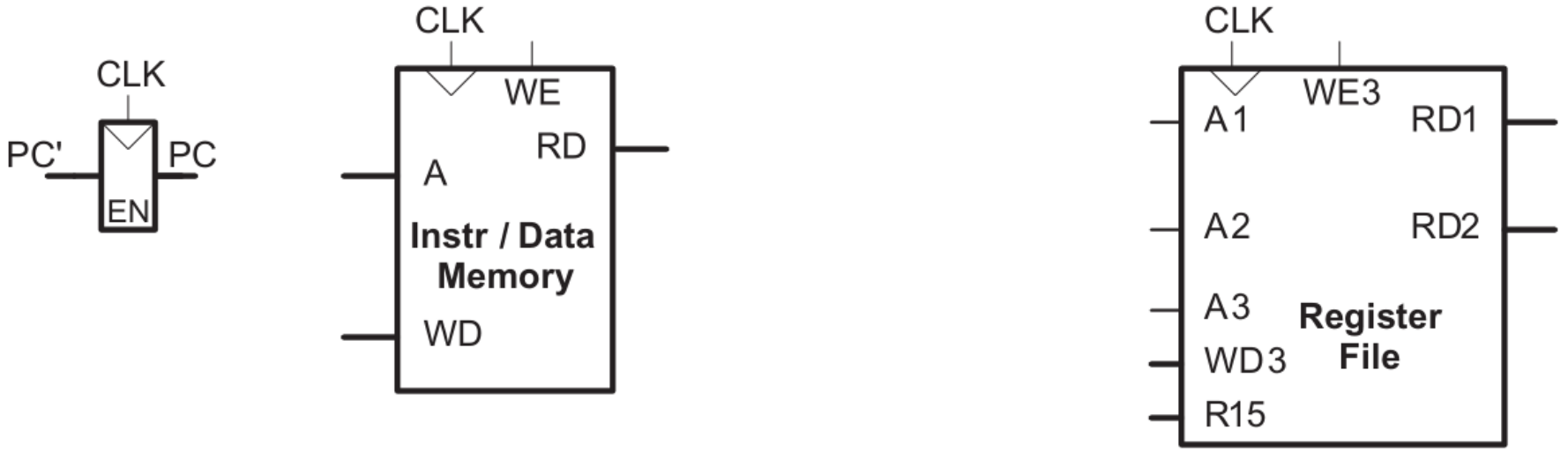
Nonarchitectural state elements are added to hold intermediate results between the steps.

The instruction is read in one step and data can be read or written in a later step, so the processor can use a single memory for both.

Different instructions use different numbers of steps, so simpler instructions can complete faster than more complex ones.

The processor needs only one adder, which is reused for different purposes on different steps.

We start with the memory and architectural state of the processor.

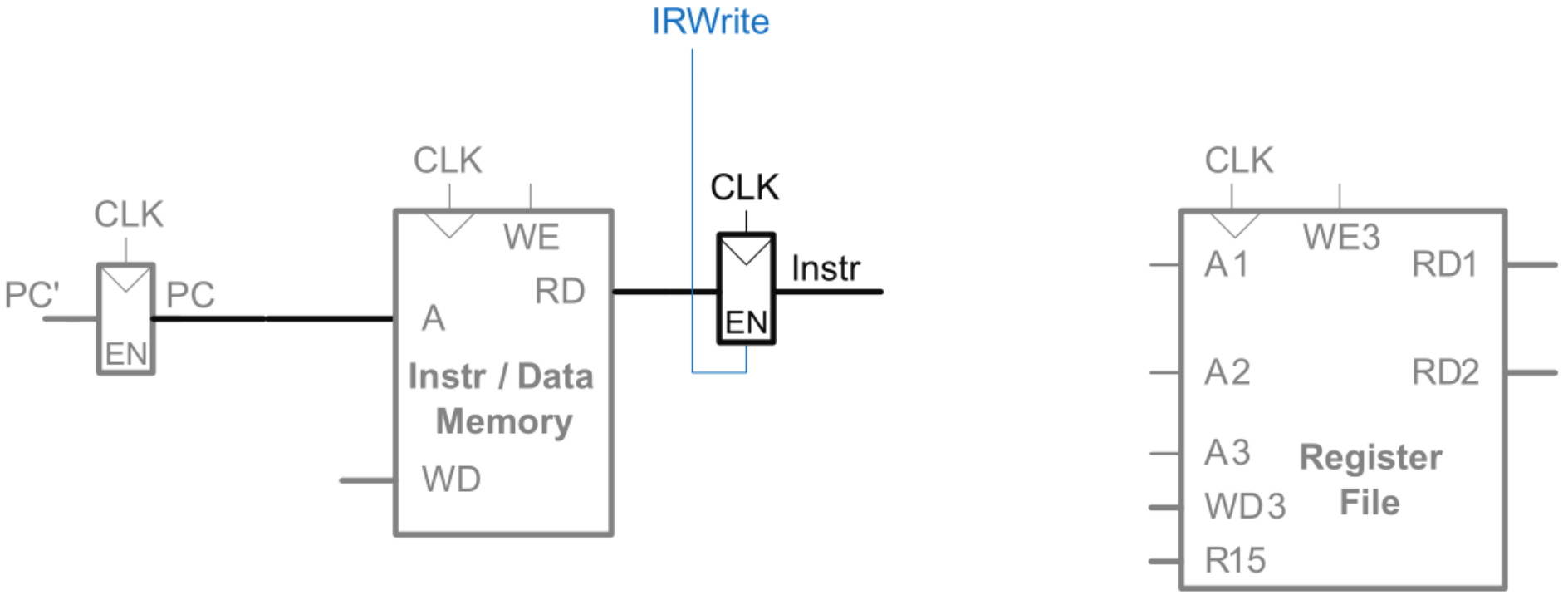


Use a combined memory for both instructions and data.

The PC and register file are the same as for the single-cycle processor.

The PC contains the address of the instruction to execute.

The first step is to read this instruction from instruction memory.

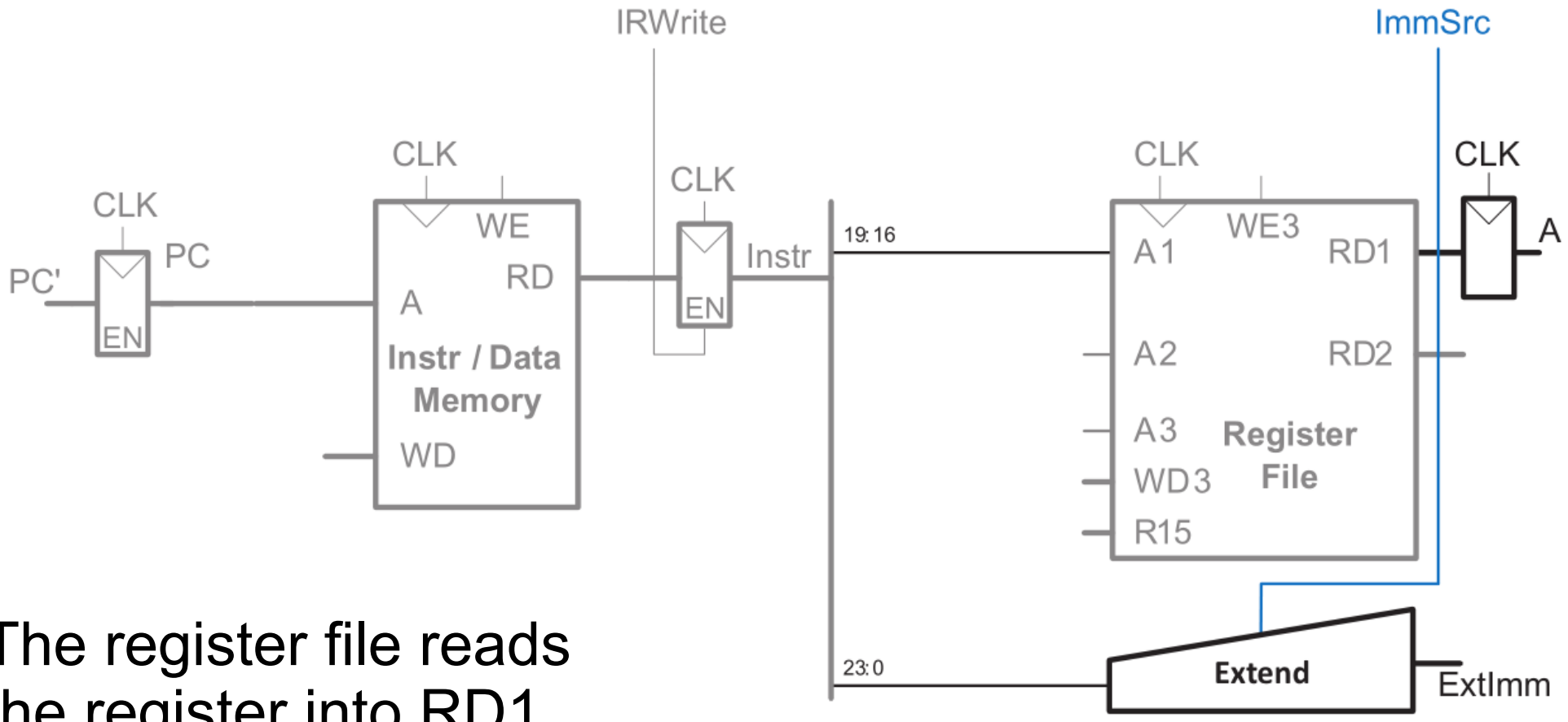


The instruction is read and stored in a nonarchitectural **instruction register** (**IR**).

The IR receives an enable signal, IRWrite

We first build the datapath for the LDR instruction.

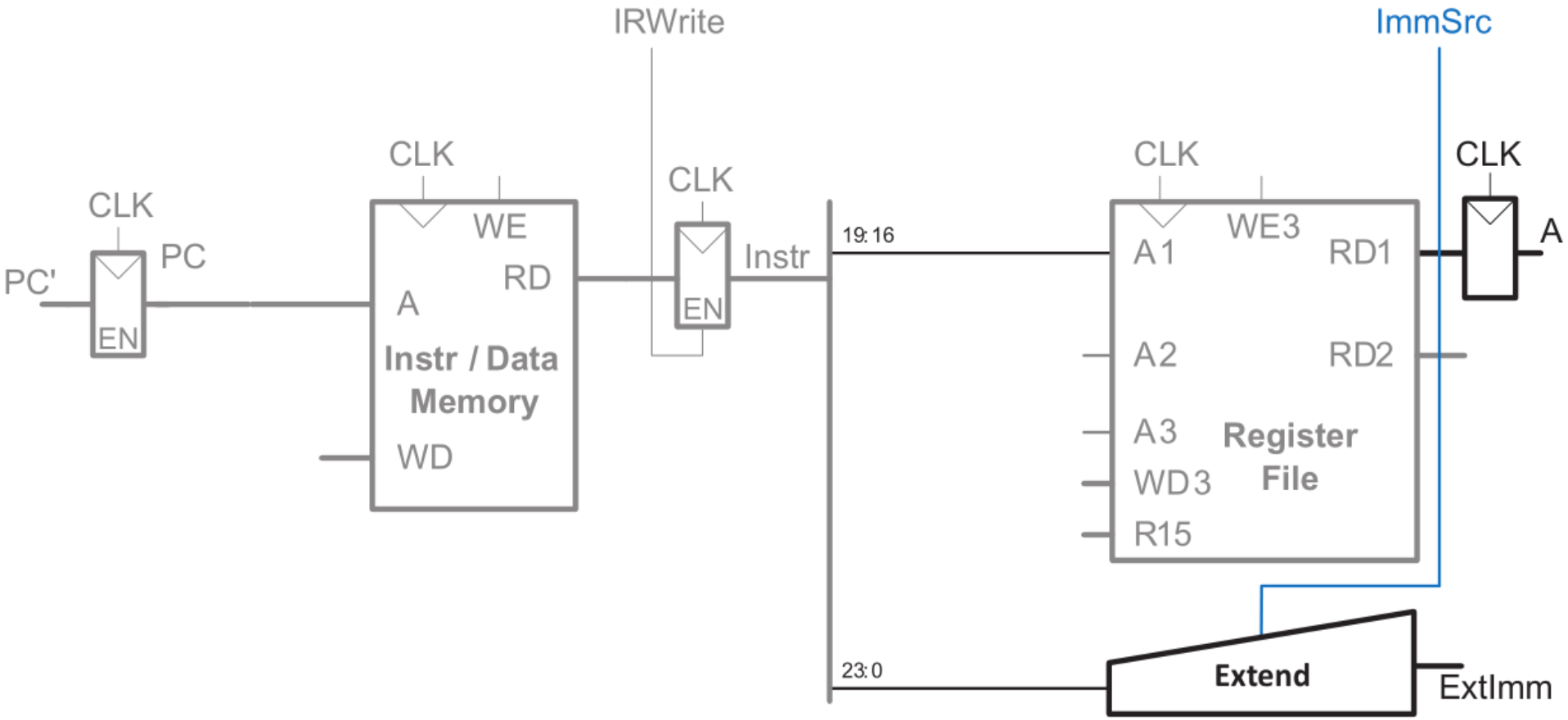
The base address is in the register specified in the Rn field, $Instr_{19:16}$



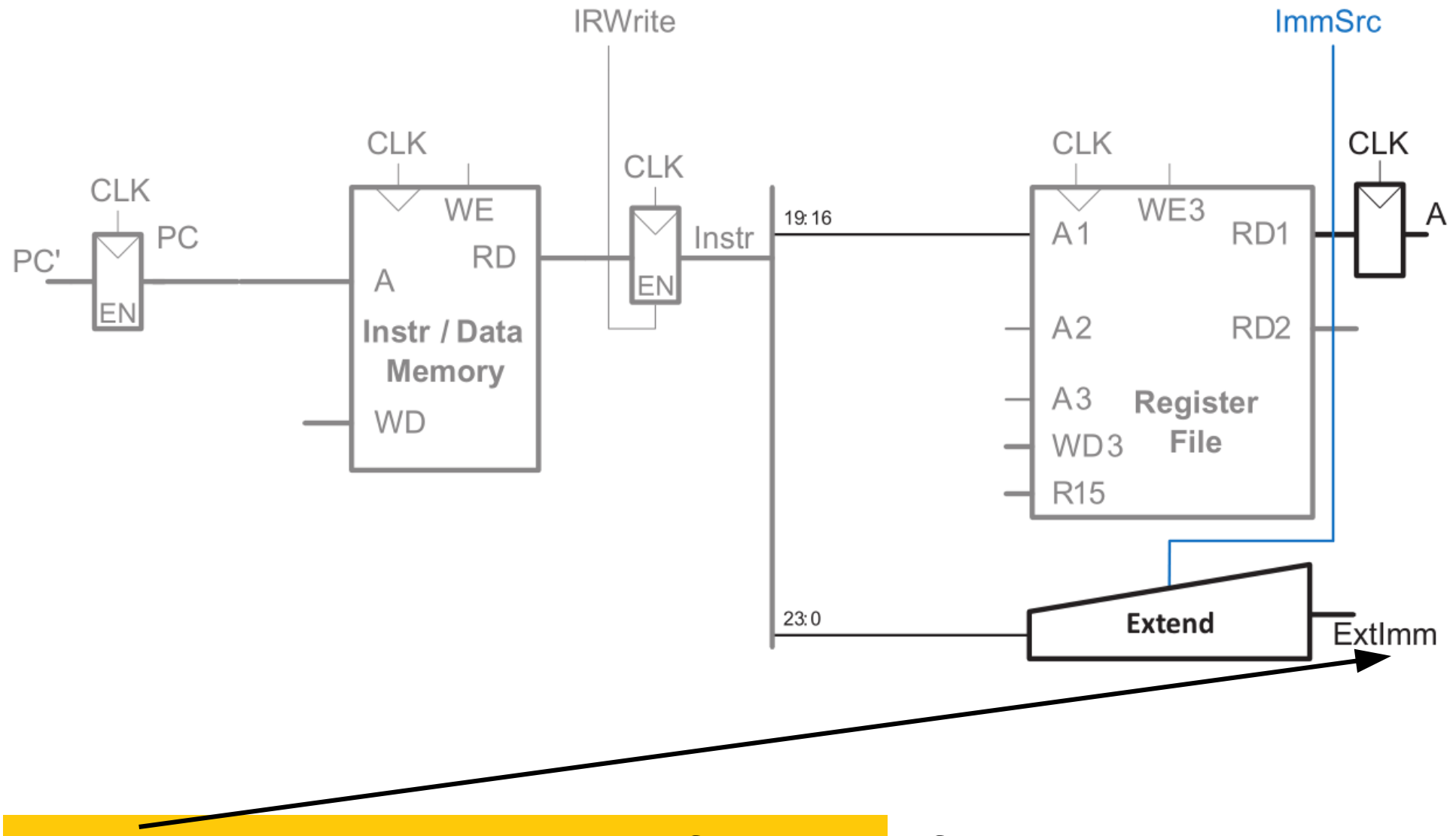
The register file reads the register into RD1

This value is stored in a nonarchitectural register, A.

The LDR instruction also requires a 12-bit offset, found in the immediate field of the instruction, $Instr_{11:0}$, which must be zero-extended to 32 bits.



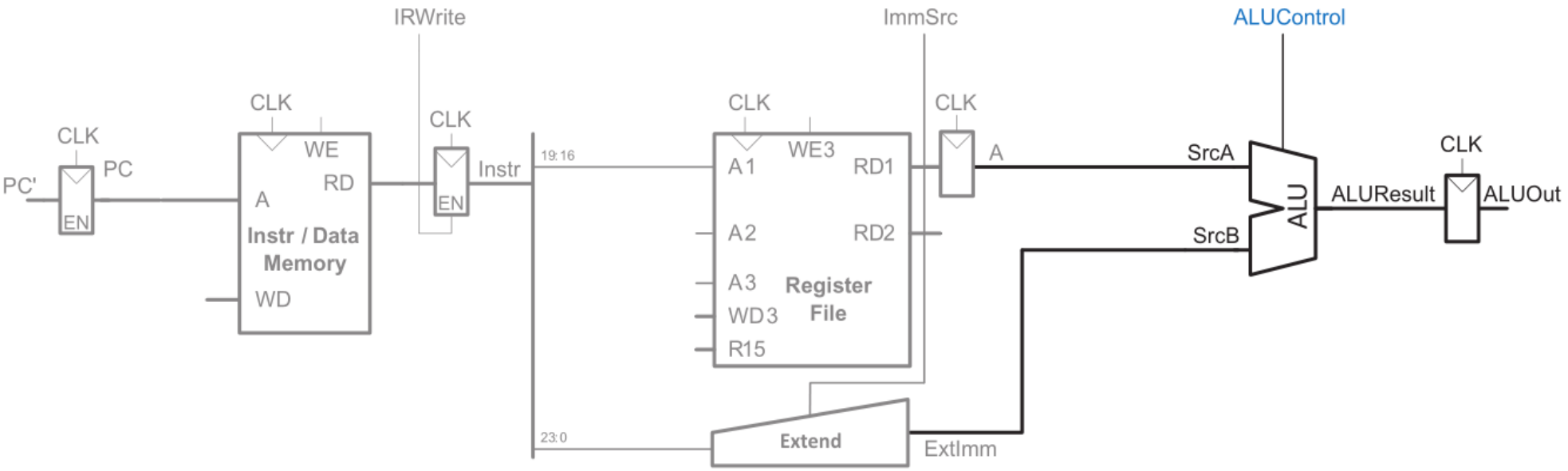
As in the single-cycle processor, the Extend block takes an ImmSrc control signal to specify an 8-, 12-, or 24-bit immediate to extend for various types of instructions.



ExtImm is a combinational function of Instr and will not change while the current instruction is being processed, so there is no need to dedicate a register to hold the constant value.

The address of the load = base address + offset

Use an ALU to compute this sum.

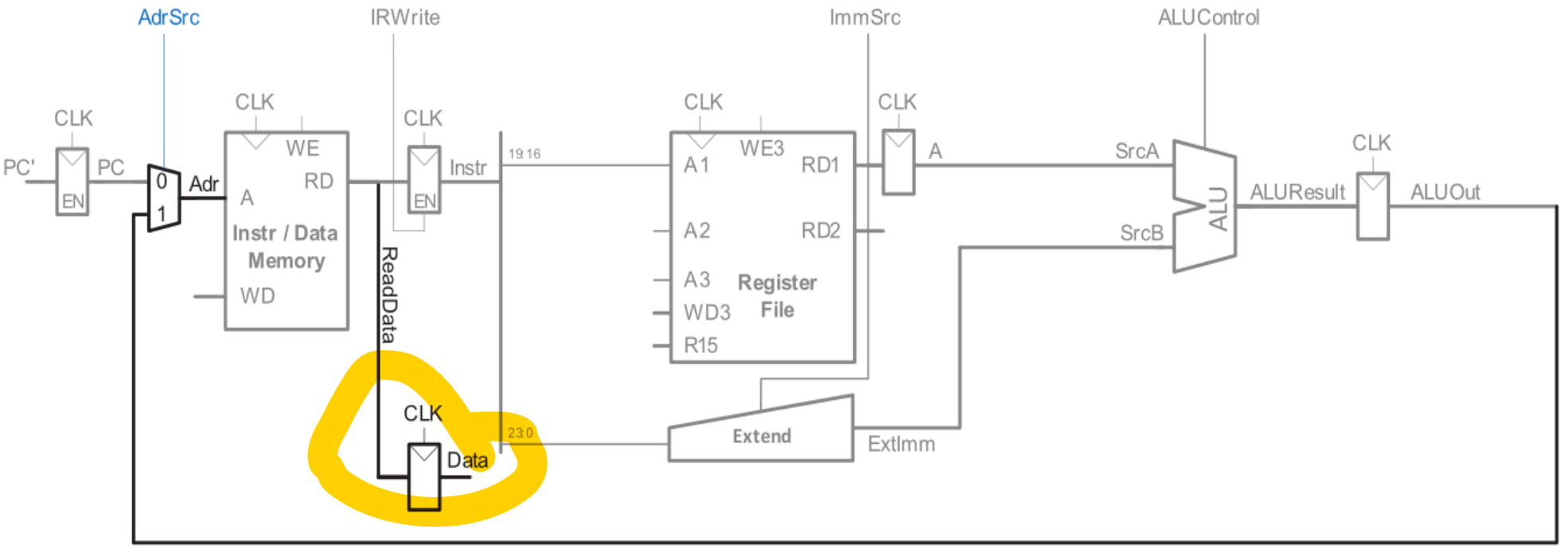


ALUControl should be set to 00 to perform the addition.

ALUResult is stored in a non-architectural register called ALUOut.

The next step is to load the data from the calculated address in the memory.

Add a multiplexer in front of the memory to choose the memory address, Adr, from either the PC or ALUOut based on the AdrSrc select.



The data read from memory is stored in another nonarchitectural register, Data.

The address multiplexer permits us to reuse the memory during the `LDR` instruction.

On a first step, the address is taken from the PC to fetch the instruction.

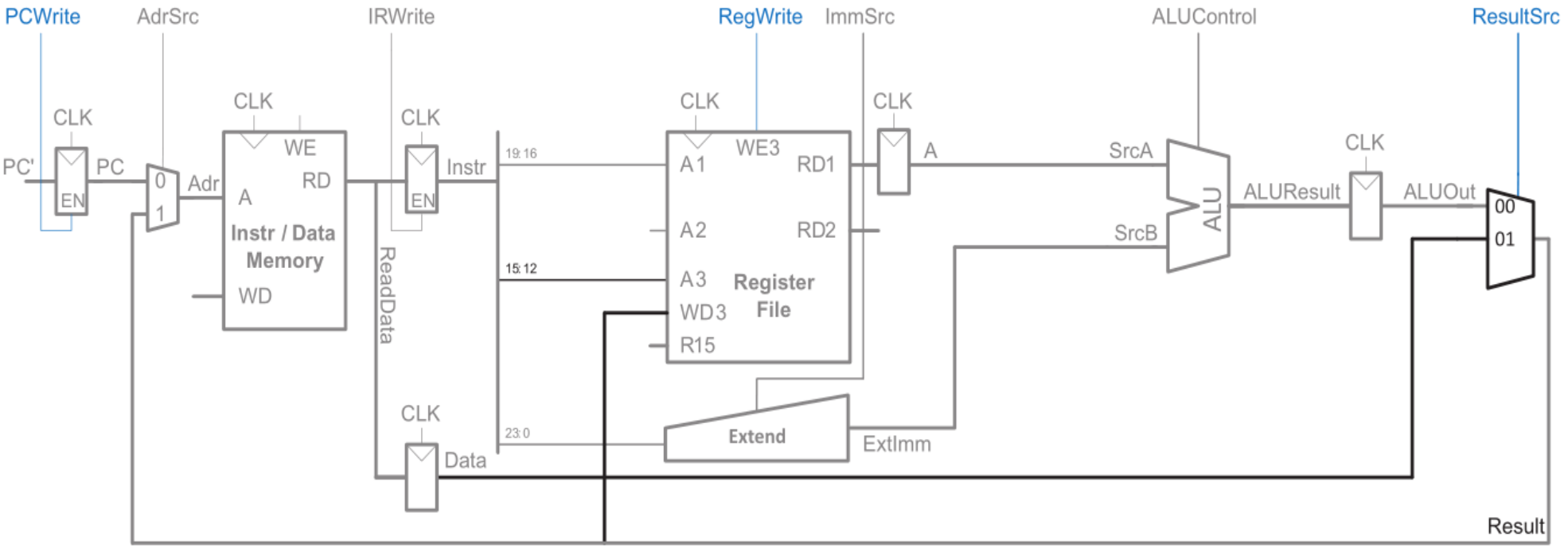
On a later step, the address is taken from ALUOut to load the data.

Hence, AdrSrc must have different values on different steps.

The control unit has to generate sequences of control signals.

Finally, the data is written back to the register file.

The destination register is specified by the Rd field of the instruction, $Instr_{15:12}$



The result comes from the Data register.

A multiplexer on the Result bus chooses either ALUOut or Data before feeding Result to the register file write port.

RegWrite = 1 to update the register file.

Note that we write the result to the register file through the multiplexer instead of connecting the Data register directly to the register file WD3 write port.

This is because other instructions will need to write a result from the ALU.

The processor must update the program counter by adding 4 to the old PC.

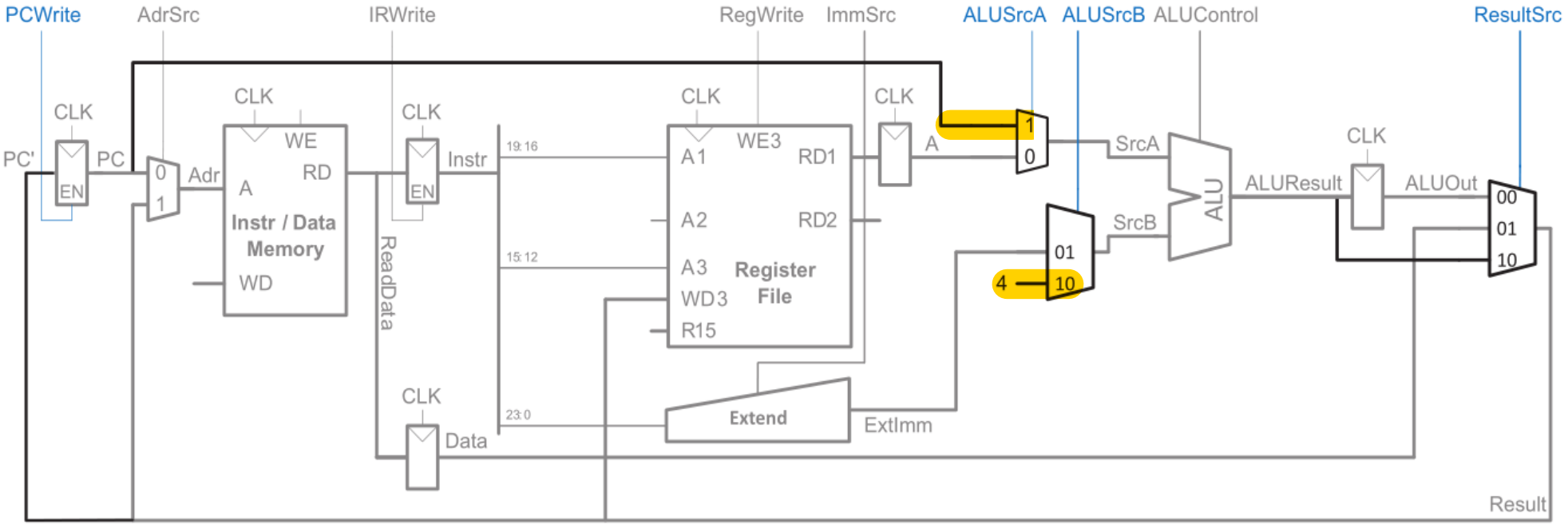
In the single-cycle processor, a separate adder was needed.

In the multicycle processor, we can use the existing ALU during the fetch step because it is not busy.

Insert source multiplexers to choose PC and the constant 4 as ALU inputs.

A multiplexer controlled by ALUSrcA chooses either PC or register A as SrcA.

Another multiplexer chooses either 4 or ExtImm as SrcB.



The ResultSrc multiplexer chooses this sum from ALUResult rather than ALUOut; this requires a third input.

The PCWrite enables the PC to be written only on certain cycles.

In the ARM architecture, reading R15 returns PC +8 and writing R15 updates the PC.

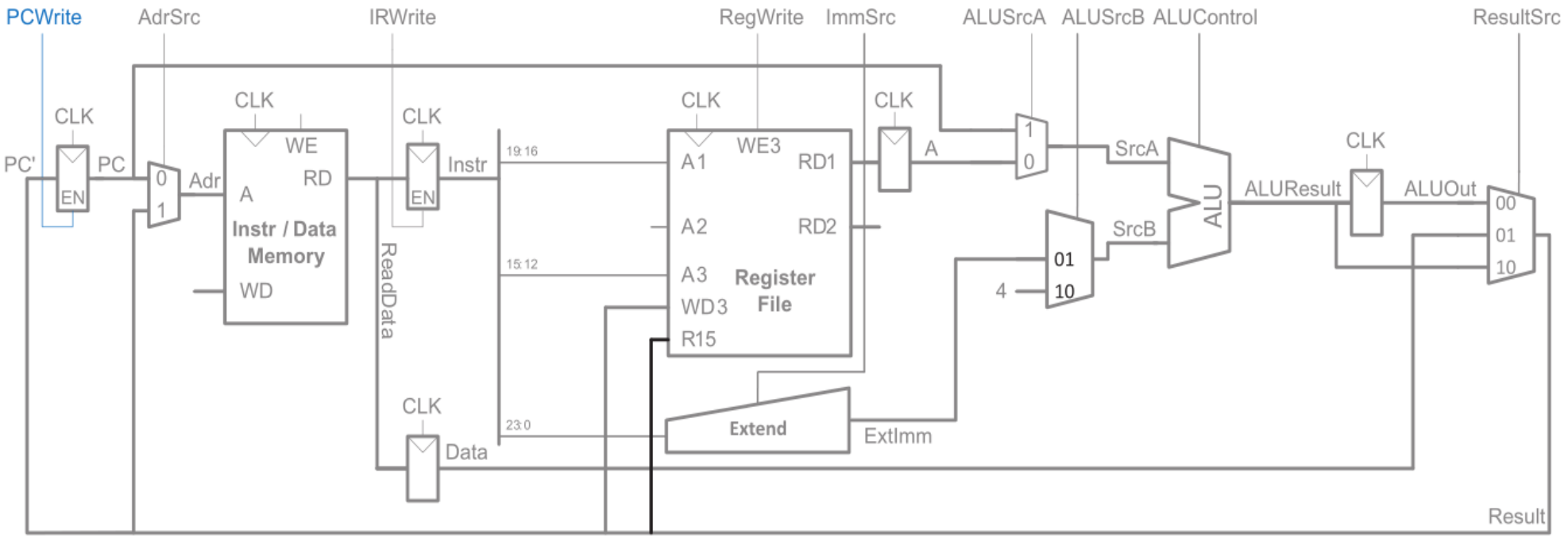
First, consider R15 reads.

We already computed PC +4 during the fetch step.

The sum is available in the PC register.

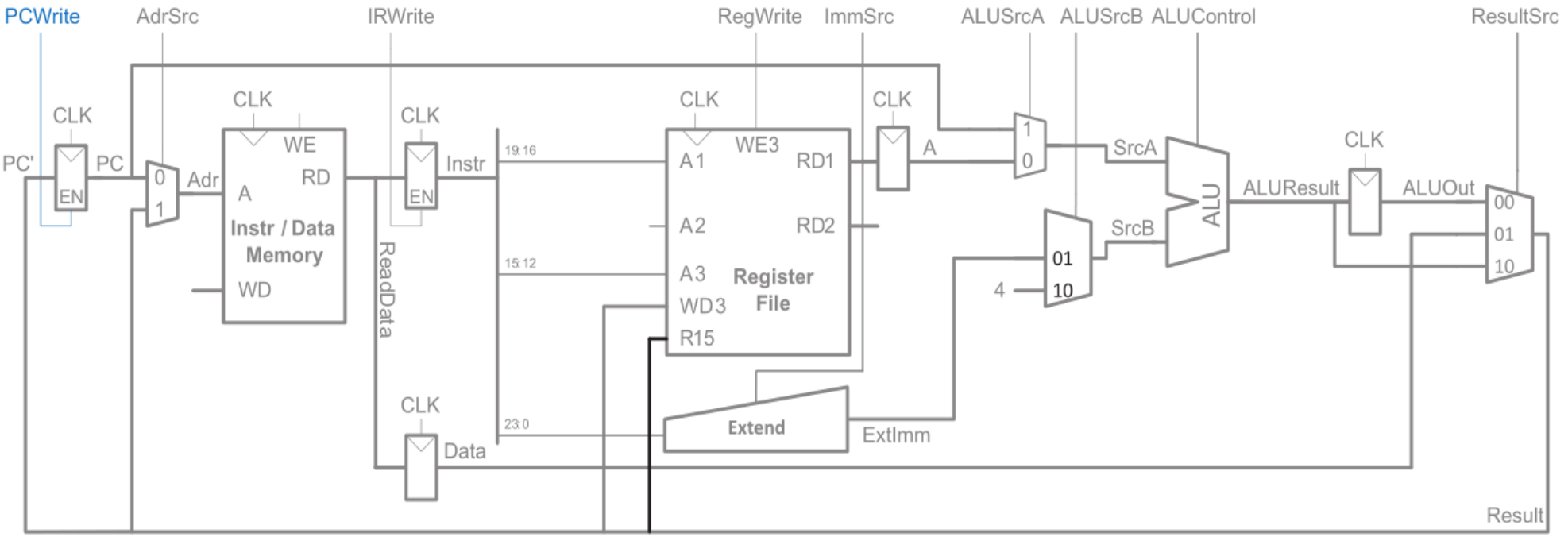
During the second step, we obtain PC +8 by adding 4 to the updated PC using the ALU.

ALUResult is selected as the Result and fed to the R15 input port of the register file.



A read of R15, which also occurs during the second step, produces the value PC +8 on the read data output of the register file.

Writes to R15 require writing the PC register instead of the register file.



Thus, in the final step of the instruction, Result must be routed to the PC register (instead of to the register file) and PCWrite must be asserted (instead of RegWrite).

The datapath already accommodates this, so no datapath changes are required.

Let us extend the datapath to handle the `STR` instruction.

Like `LDR` , `STR` reads a base address from port 1 of the register file and extends the immediate.

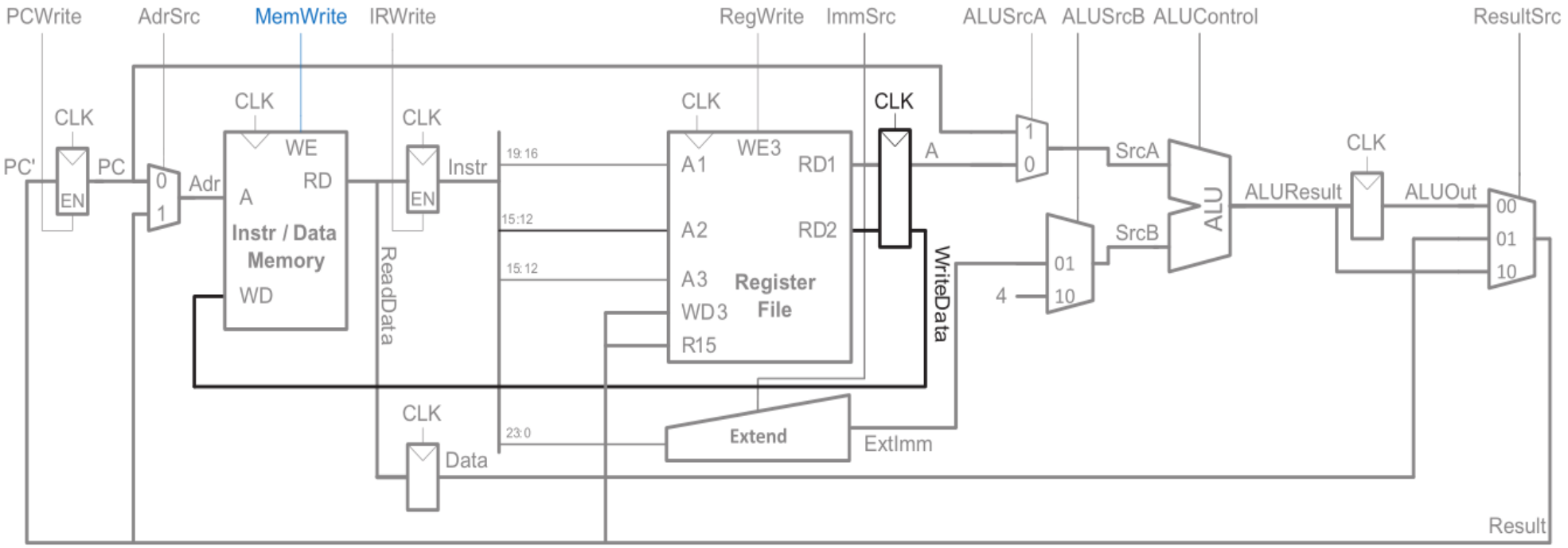
The ALU adds the base address to the immediate to find the memory address.

All of these functions are already supported by existing hardware in the datapath.

The only new feature of STR – read a second register from the register file and write it into the memory

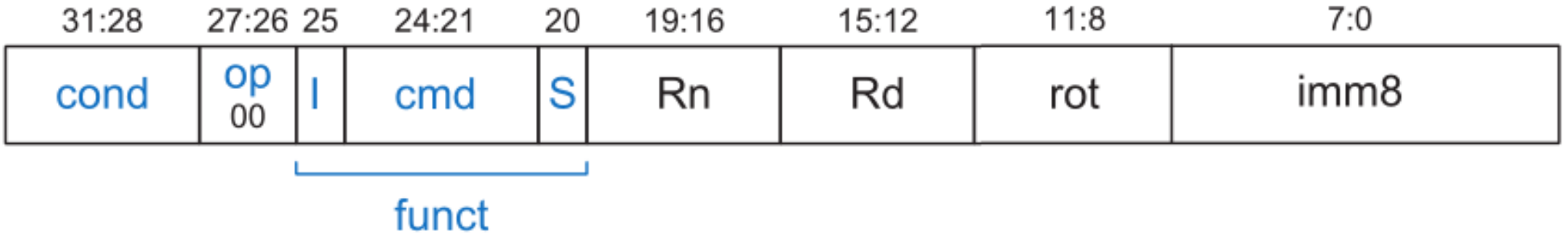
The register is specified in $Instr_{15:12}$, which is connected to the port A2 of the register file.

When the register is read, it is stored in a nonarchitectural register, WriteData.



On the next step, it is sent to the write data port (WD) of the data memory to be written.

Data-processing instructions with immediate addressing read the first source from Rn and extend the second source from an 8-bit immediate.



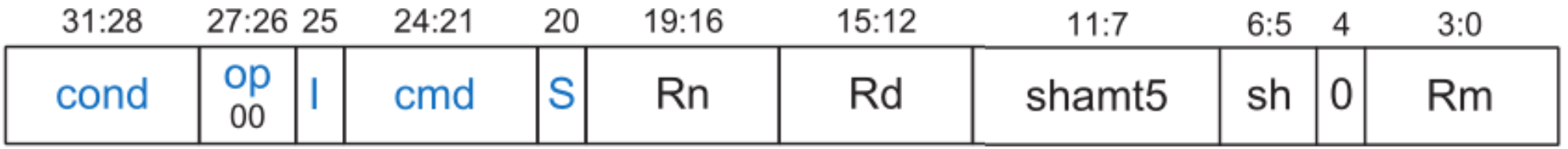
They operate on these two sources and then write the result back to the register file.

The datapath already contains all the connections necessary for these steps.

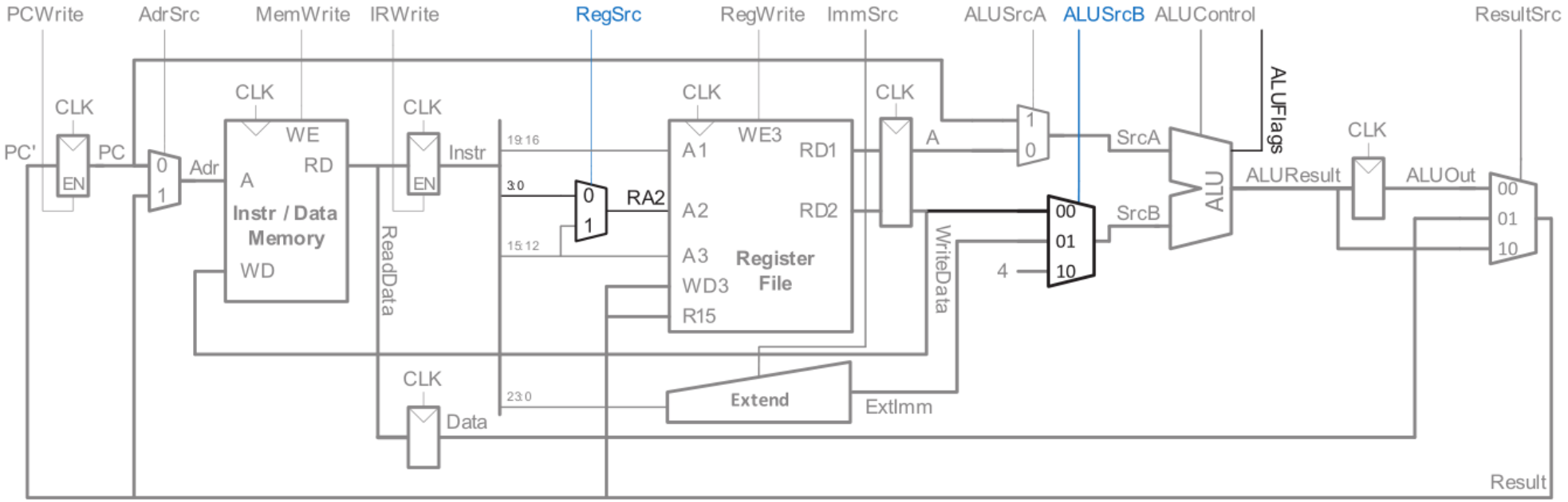
The ALU uses the ALUControl signal to determine the type of data-processing instruction to execute.

The ALUFlags are sent back to the controller to update the Status register.

Data-processing instructions with register addressing select the second source from the register file.



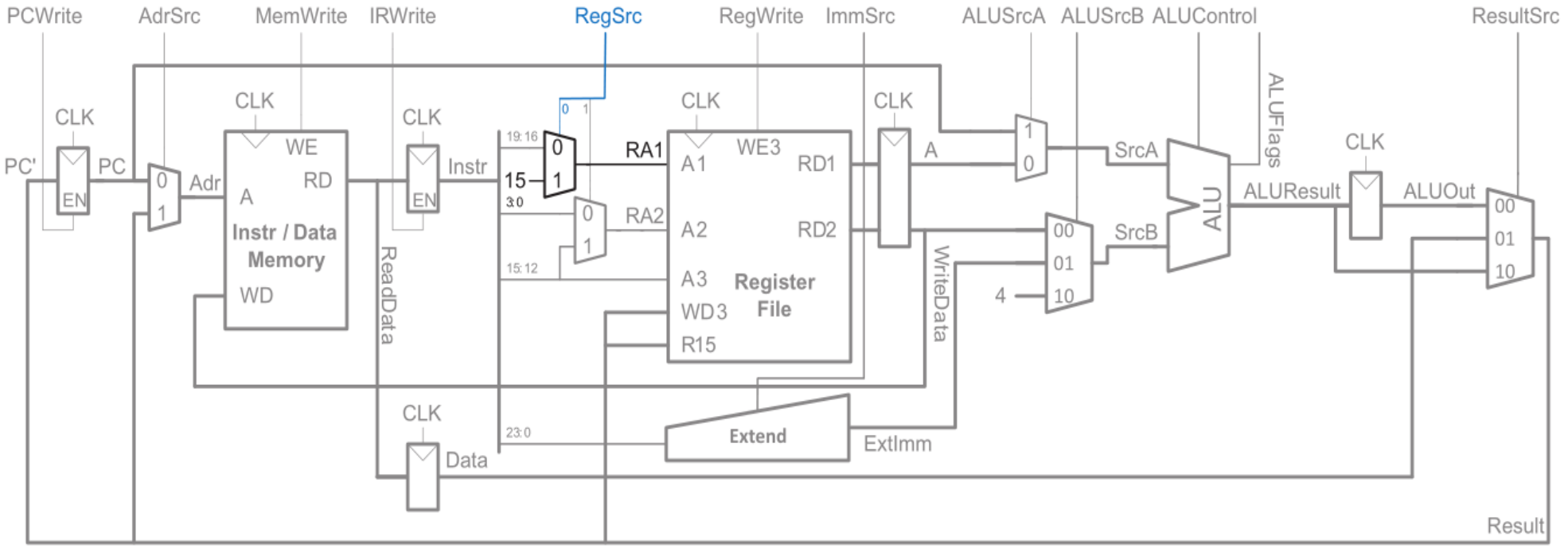
Insert a mux to choose $Instr_{3:0}$ as RA2 for the register file.



Also extend the SrcB multiplexer to accept the value read from the register file.

The branch instruction B reads PC +8 and a 24-bit immediate, sums them, and adds the result to the PC.

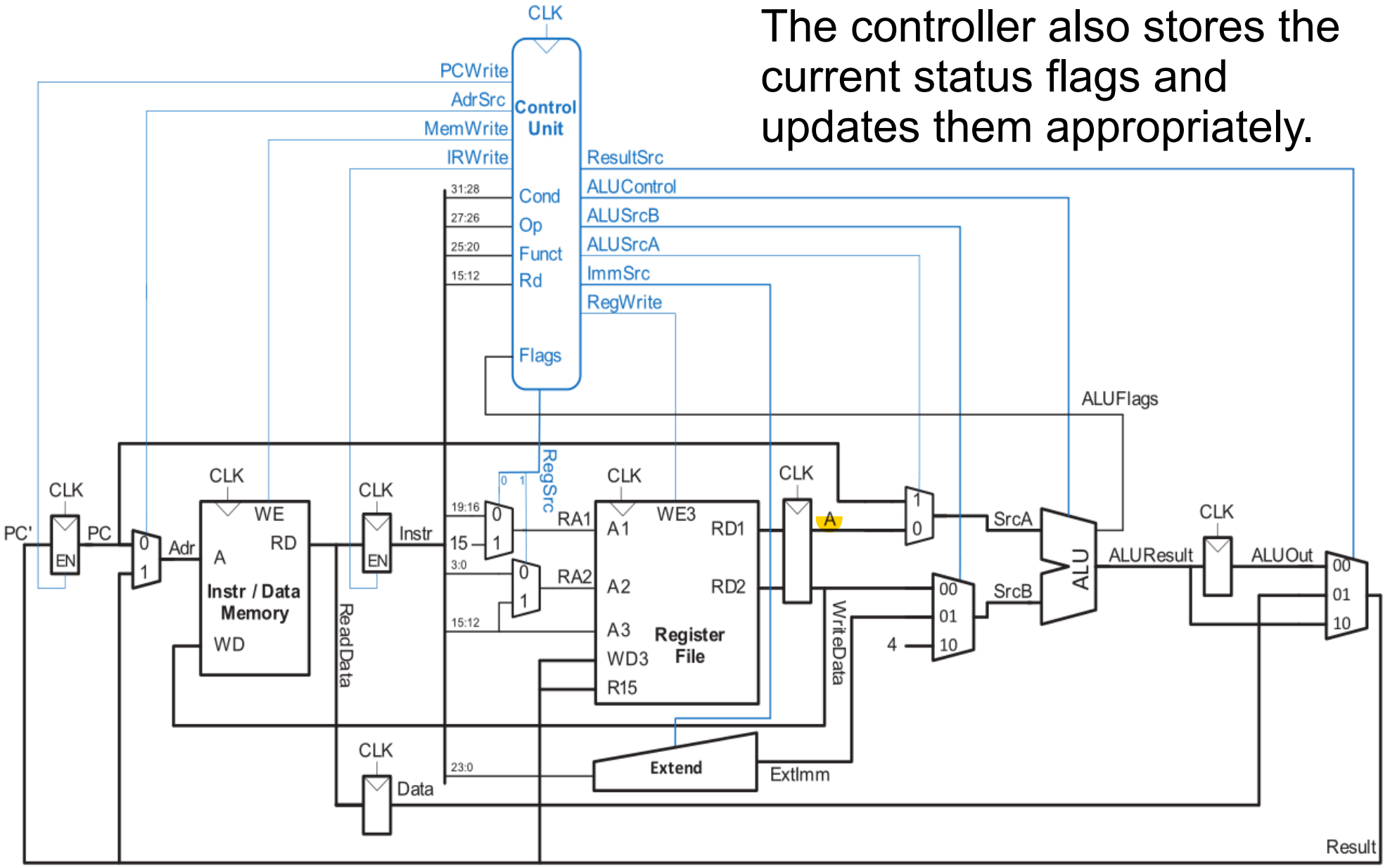
A read to R15 returns PC+ 8, so we add a multiplexer to choose R15 as RA1 for the register file



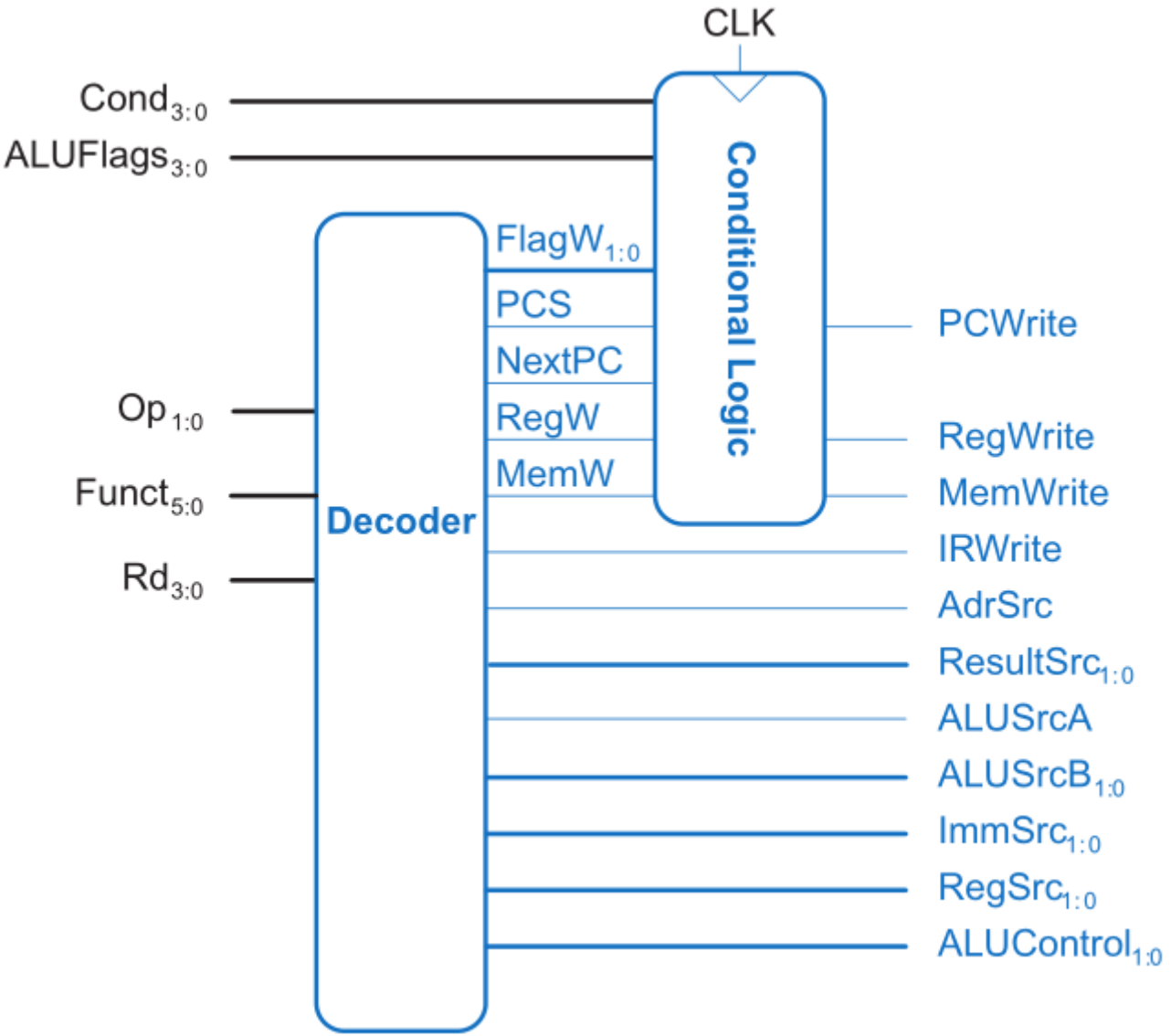
This completes the design of the multicycle datapath.

The control unit computes the control signals based on the cond, op, and funct fields of the instruction as well as the flags and whether the destination register is the PC.

The controller also stores the current status flags and updates them appropriately.

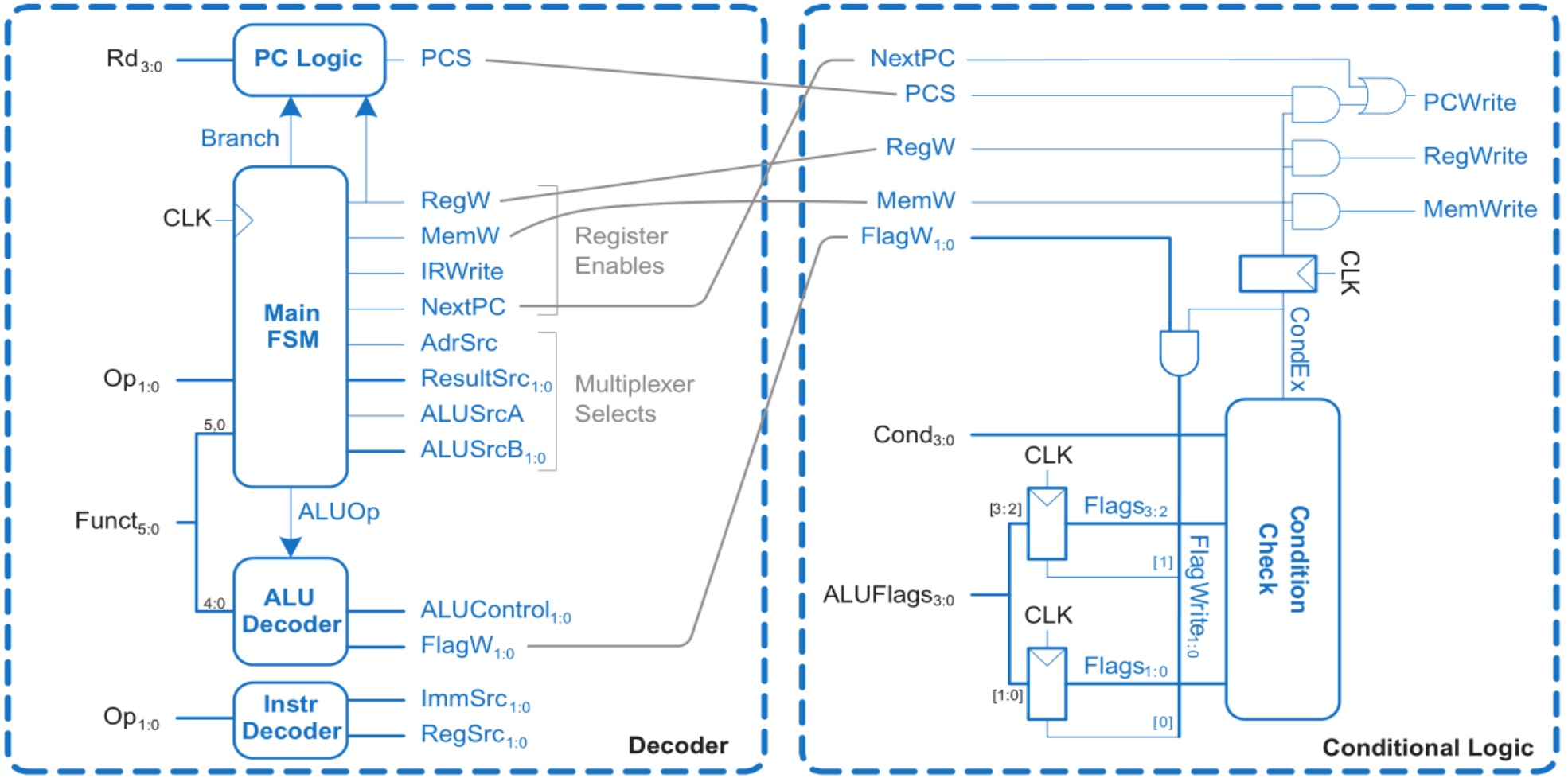


The control unit is partitioned into Decoder and Conditional Logic blocks.



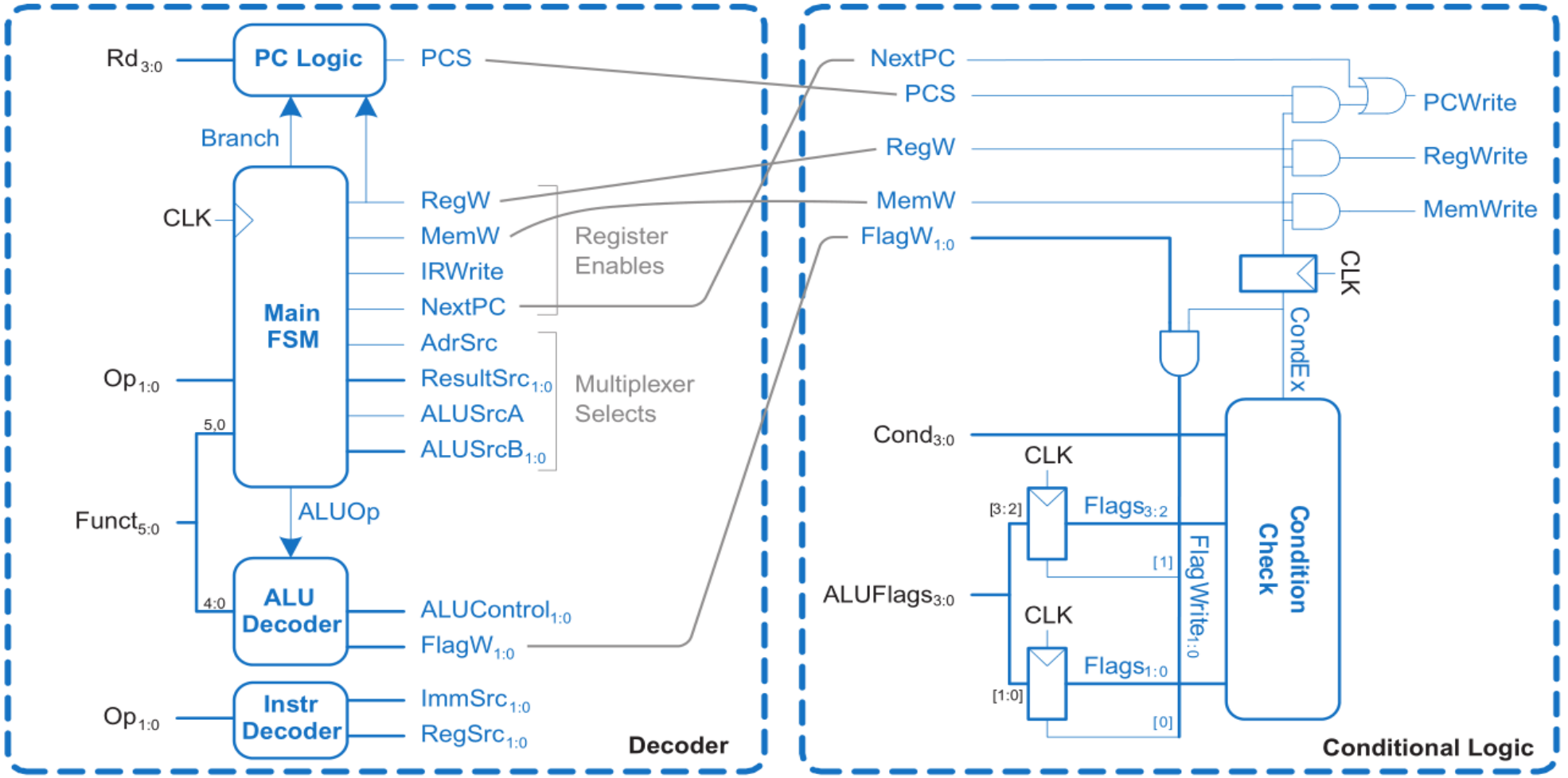
The combinational Main Decoder of the single-cycle processor is replaced with a **Main FSM** in the multicycle processor to produce a sequence of control signals on the appropriate cycles.

The Main FSM is a Moore machine.



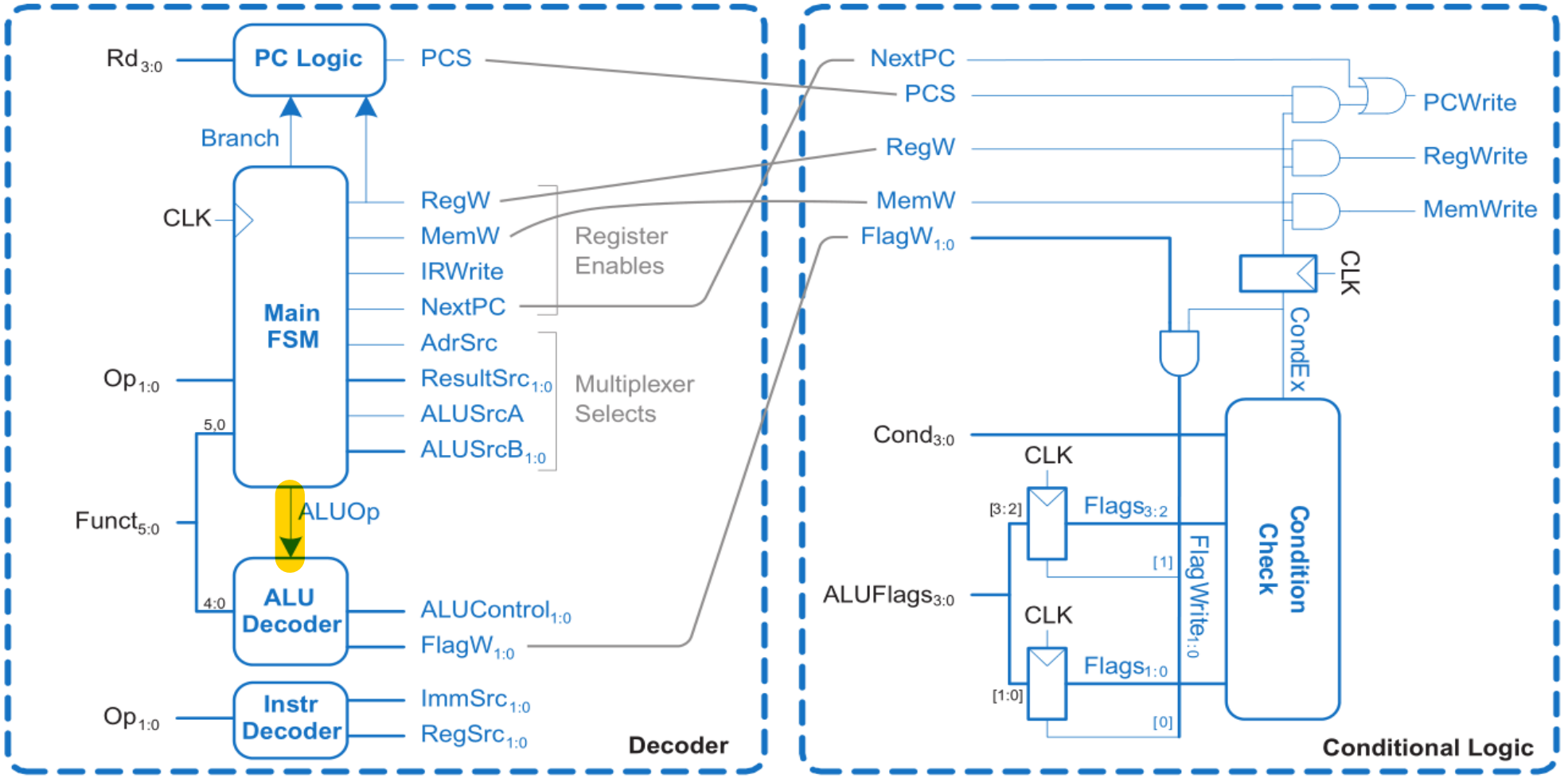
ImmSrc and RegSrc are a function of Op rather than the current state, so we also use an **Instruction Decoder** to compute these signals

The ALU Decoder and PC Logic are identical to those in the single-cycle processor.



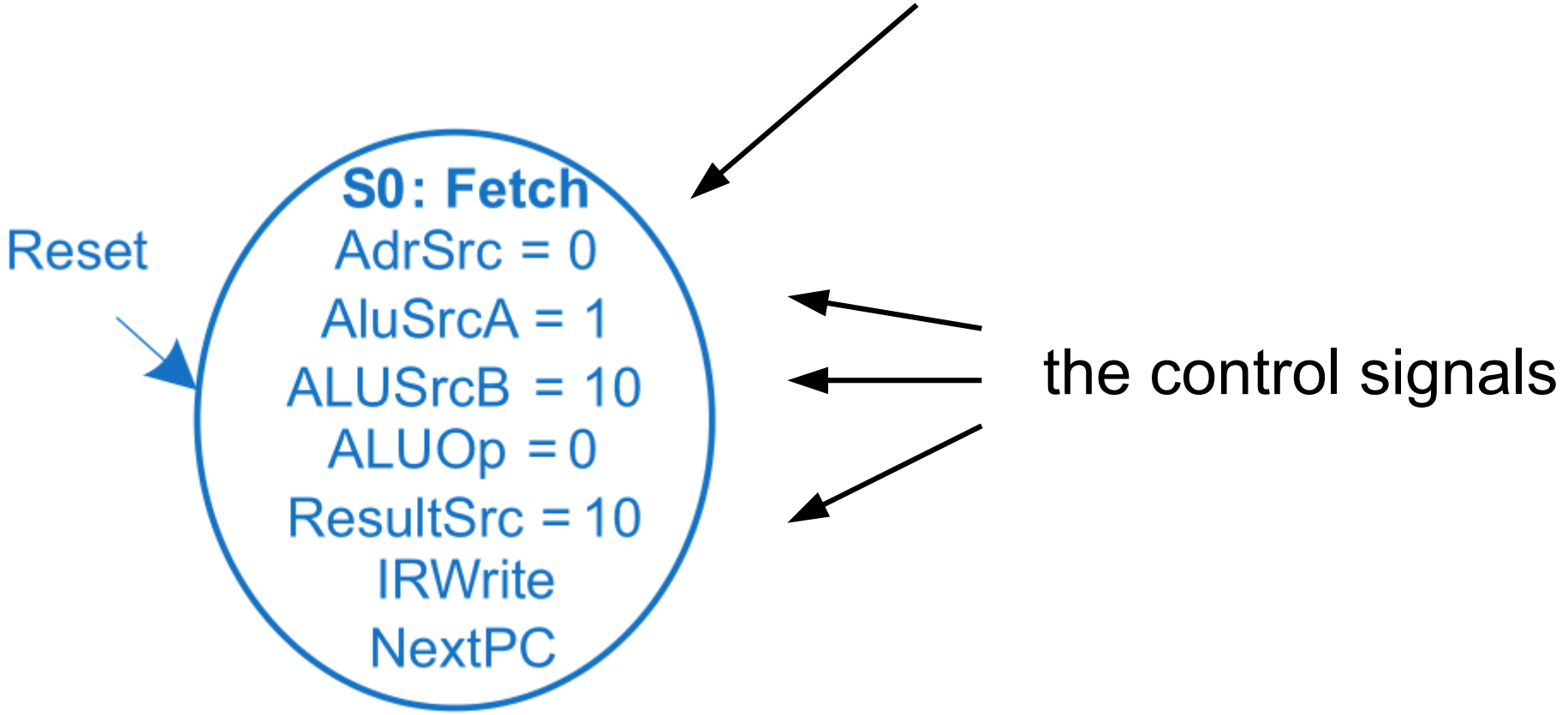
The Conditional Logic is almost identical to that of the single-cycle processor, but we add a NextPC signal to force a write to the PC when we compute PC +4.

We also delay CondEx by one cycle before sending it to PCWrite, RegWrite, and MemWrite so that updated condition flags are not seen until the end of an instruction.

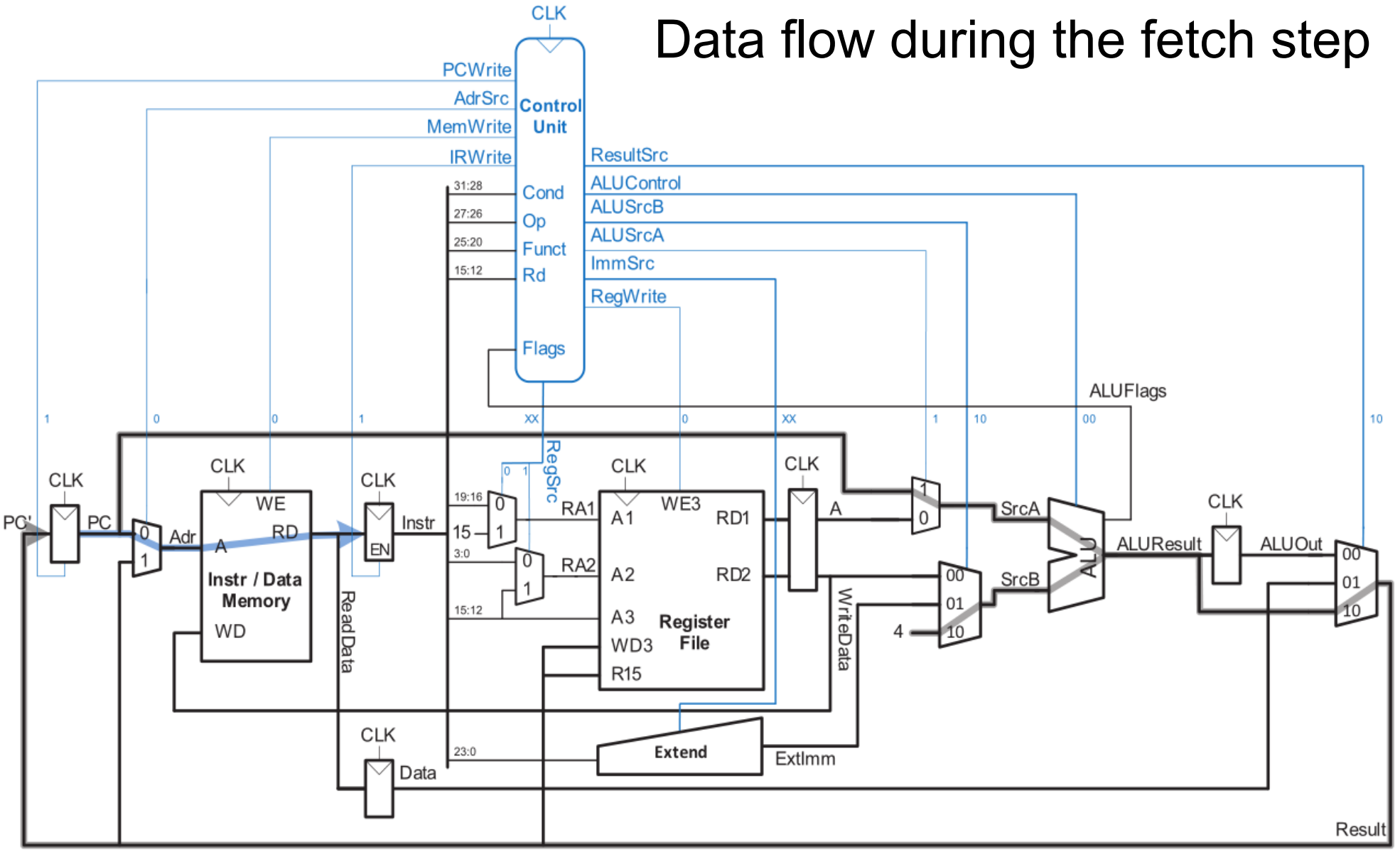


The first step for any instruction is to fetch the instruction from memory at the address held in the PC and to increment the PC to the next instruction.

The Main FSM enters this Fetch state on reset.



Data flow during the fetch step



Because the ALU is not being used for anything else, the processor can use it to compute PC+4 at the same time that it fetches the instruction.

The second step is to read the register file and/or immediate and decode the instructions.

The registers and immediate are selected based on RegSrc and ImmSrc, which are computed by the Instr Decoder based on Instr.

Instr Decoder logic for RegSrc and ImmSrc:

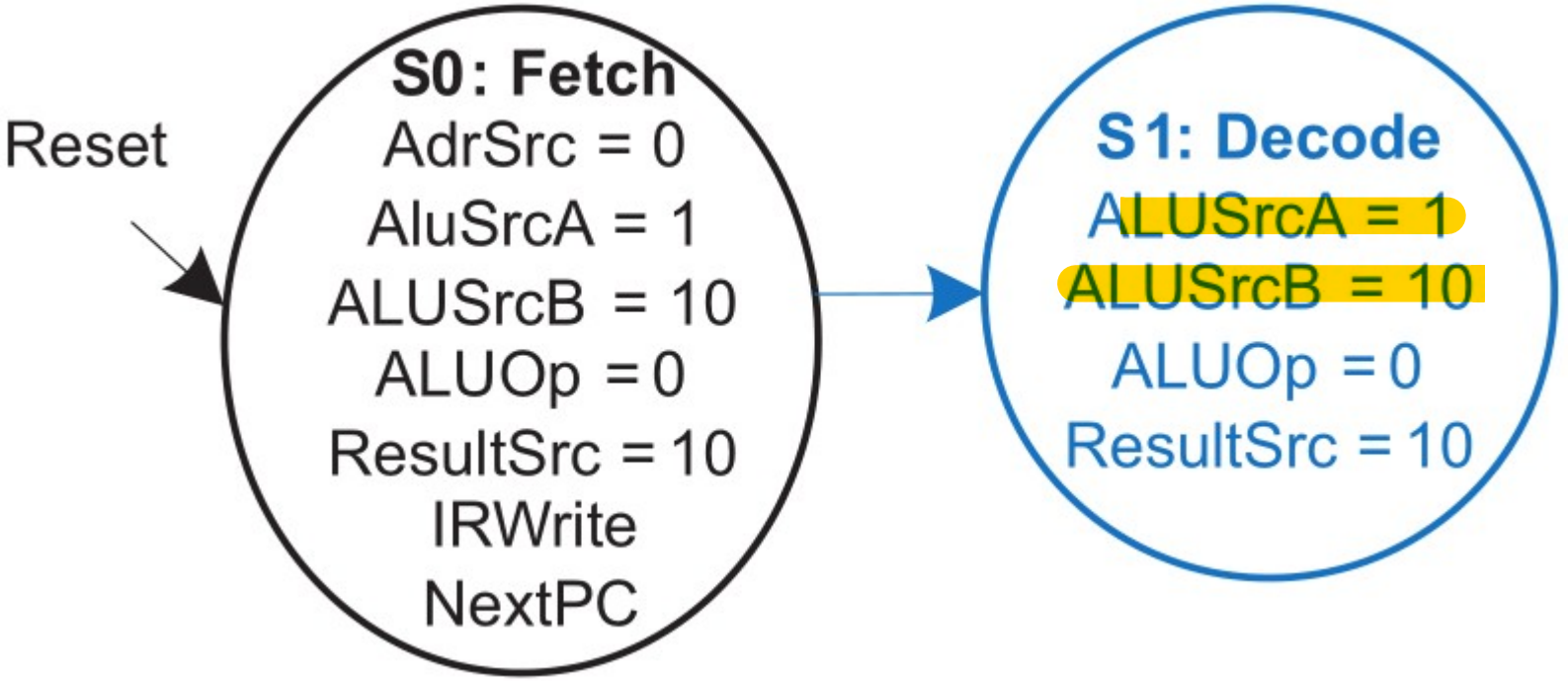
Instruction	Op	$Funct_5$	$Funct_0$	$RegSrc_1$	$RegSrc_0$	$ImmSrc_{1:0}$
LDR	01	X	1	X	0	01
STR	01	X	0	1	0	01
DP immediate	00	1	X	X	0	00
DP register	00	0	X	0	0	00
B	10	X	X	X	1	10

$$RegSrc_1 = (Op == 01) \quad RegSrc_0 = (Op == 10) \quad ImmSrc = Op$$

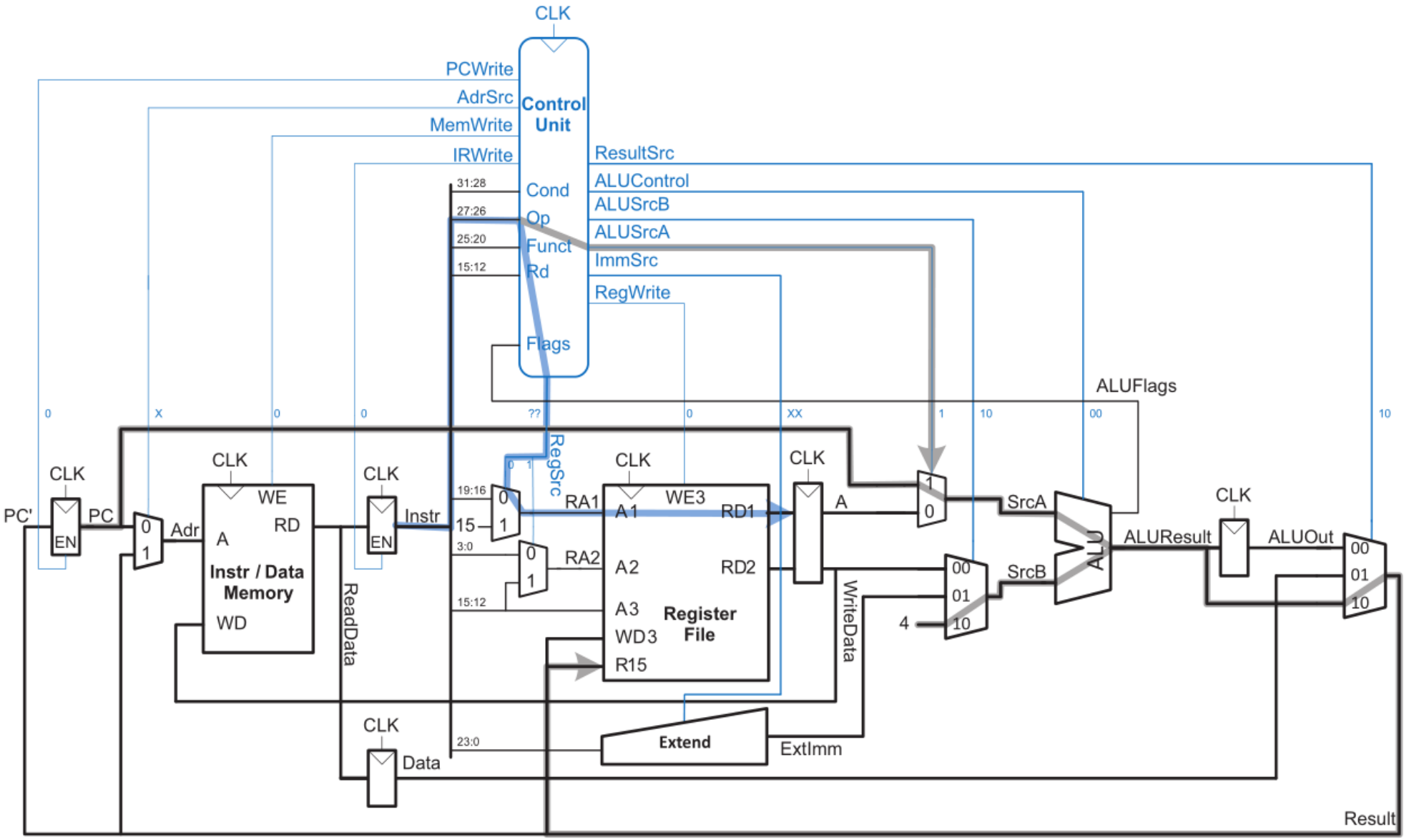
Meanwhile, the ALU is reused to compute PC+8 by adding 4 more to the PC that was incremented in the Fetch step.

The sum is selected as the Result (ResultSrc =10) and provided to the R15 input of the register file so that R15 reads as PC+ 8.

The FSM Decode step:



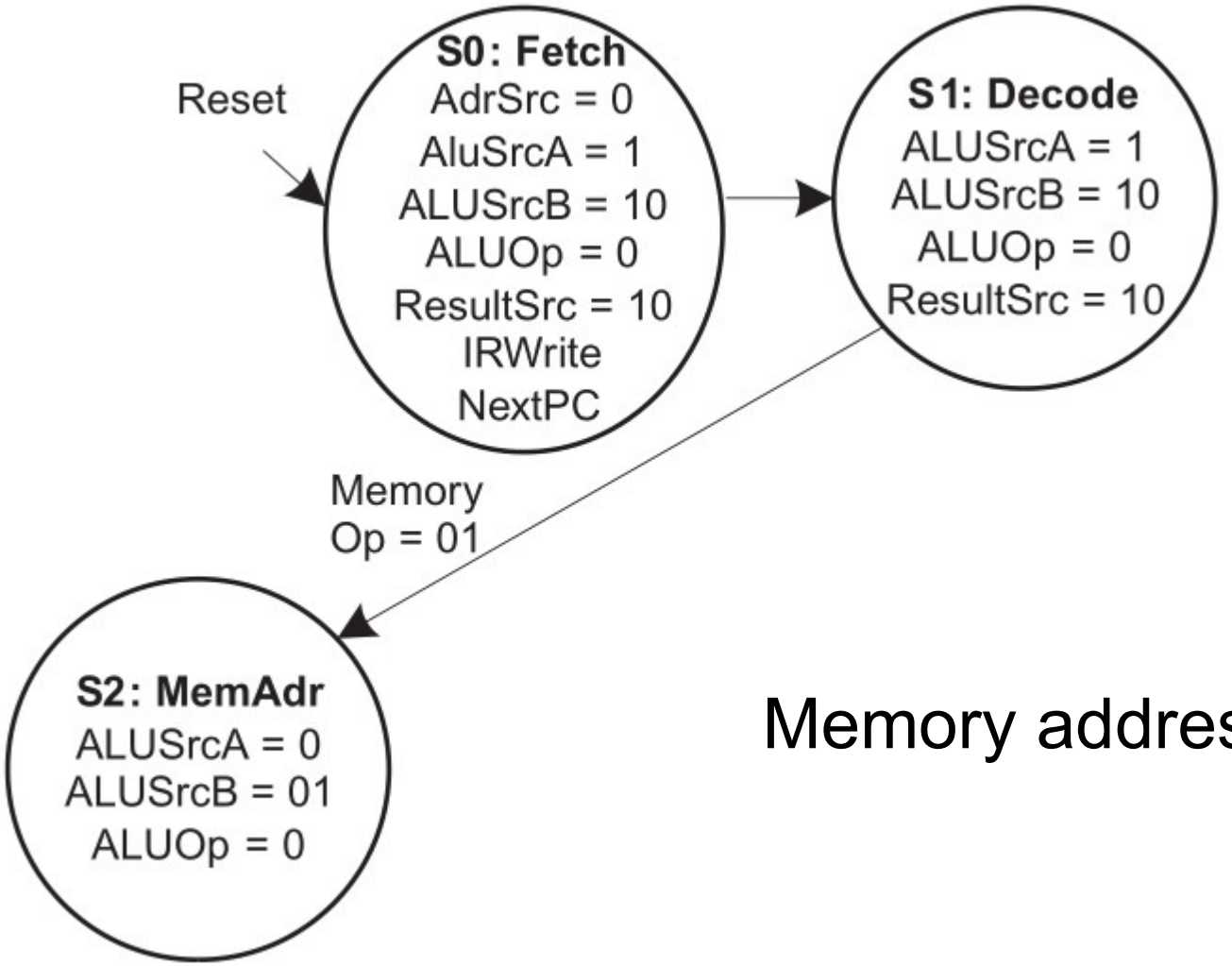
Data flow during the Decode step



Now the FSM proceeds to one of several possible states, depending on Op and Funct that are examined during the Decode step.

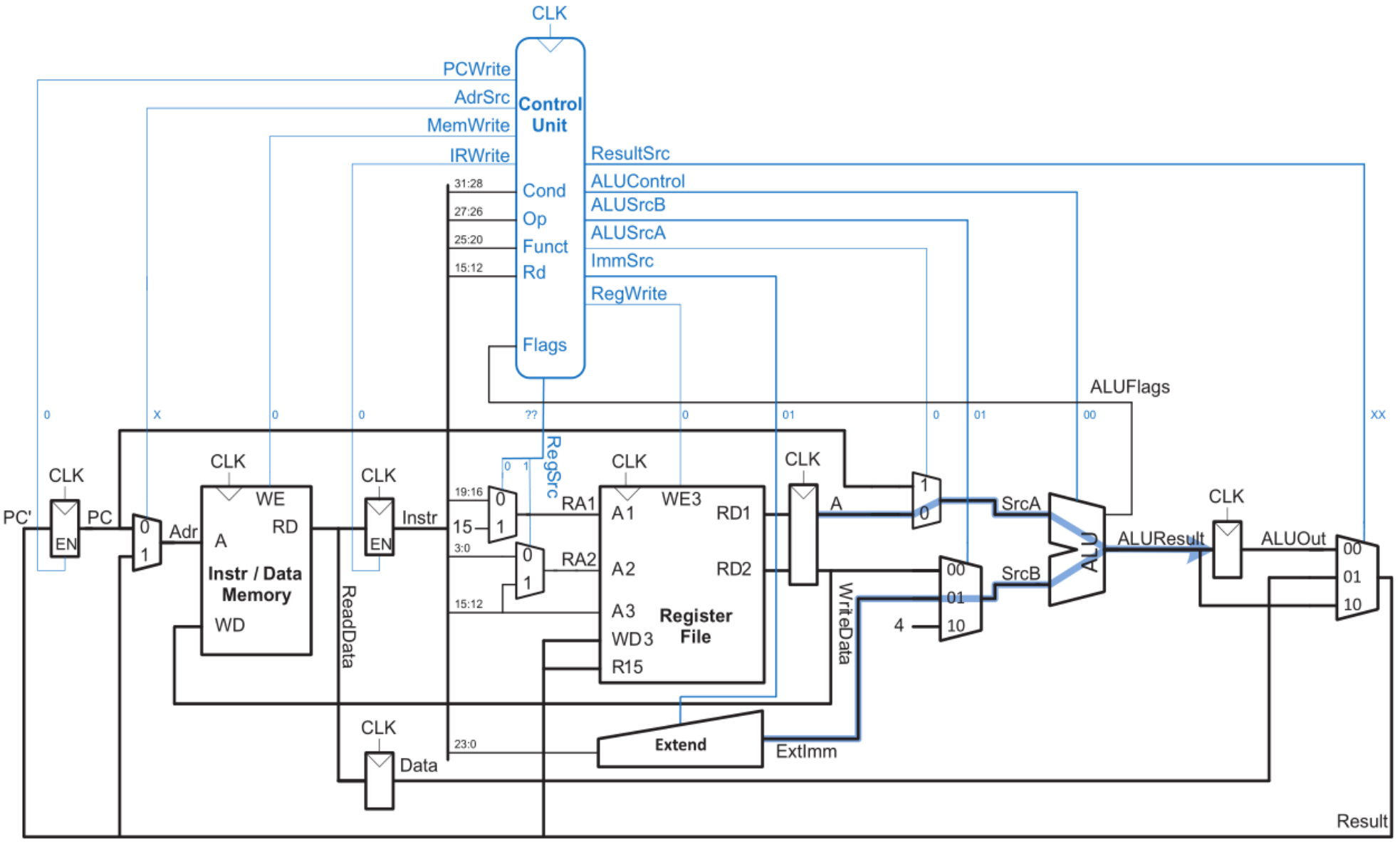
If the instruction is a memory load or store, then the processor computes the address by adding the base address to the zero-extended offset.

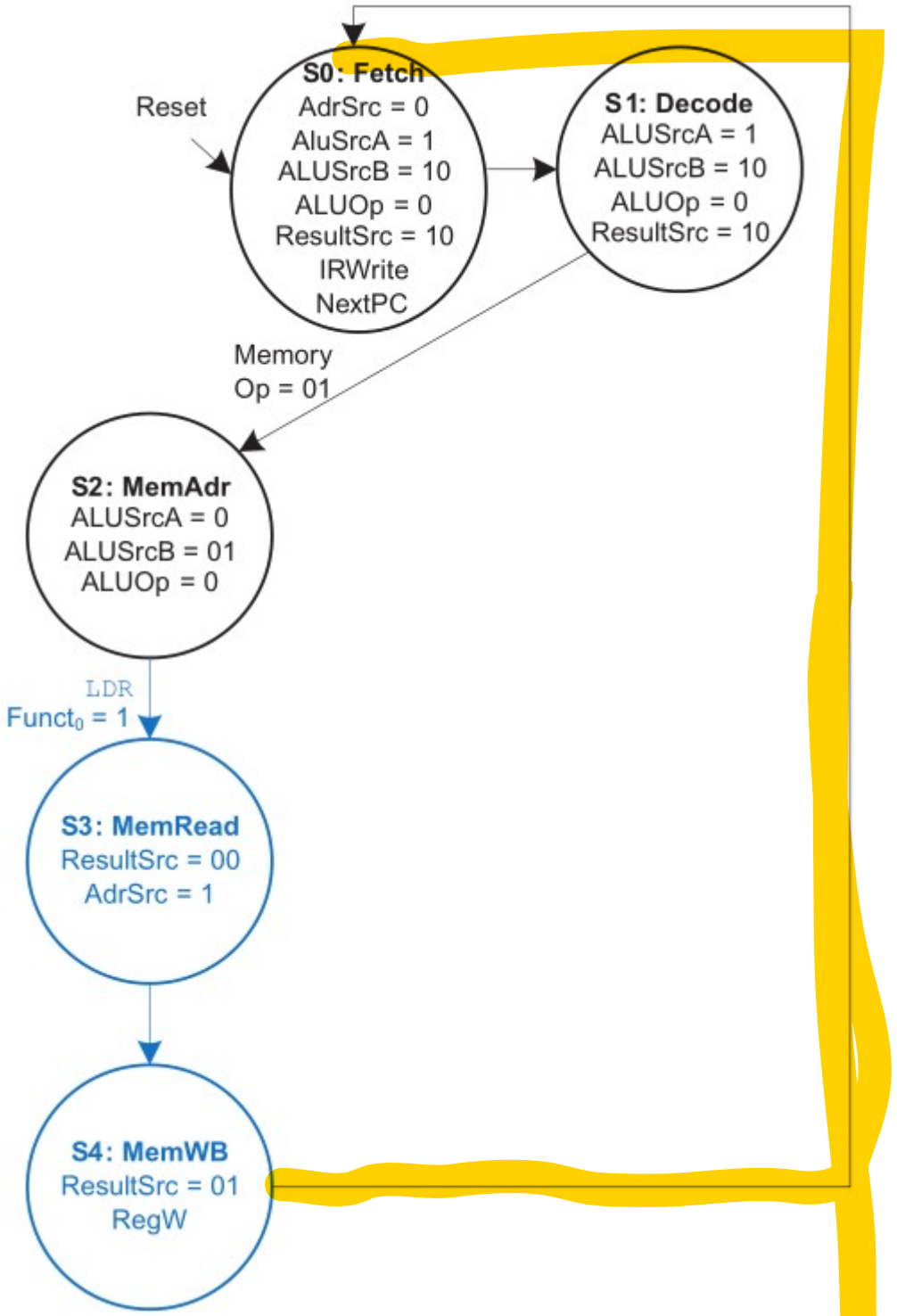
The effective address is stored in the ALUOut register for use on the next step.



Memory address computation

Data flow during memory address computation



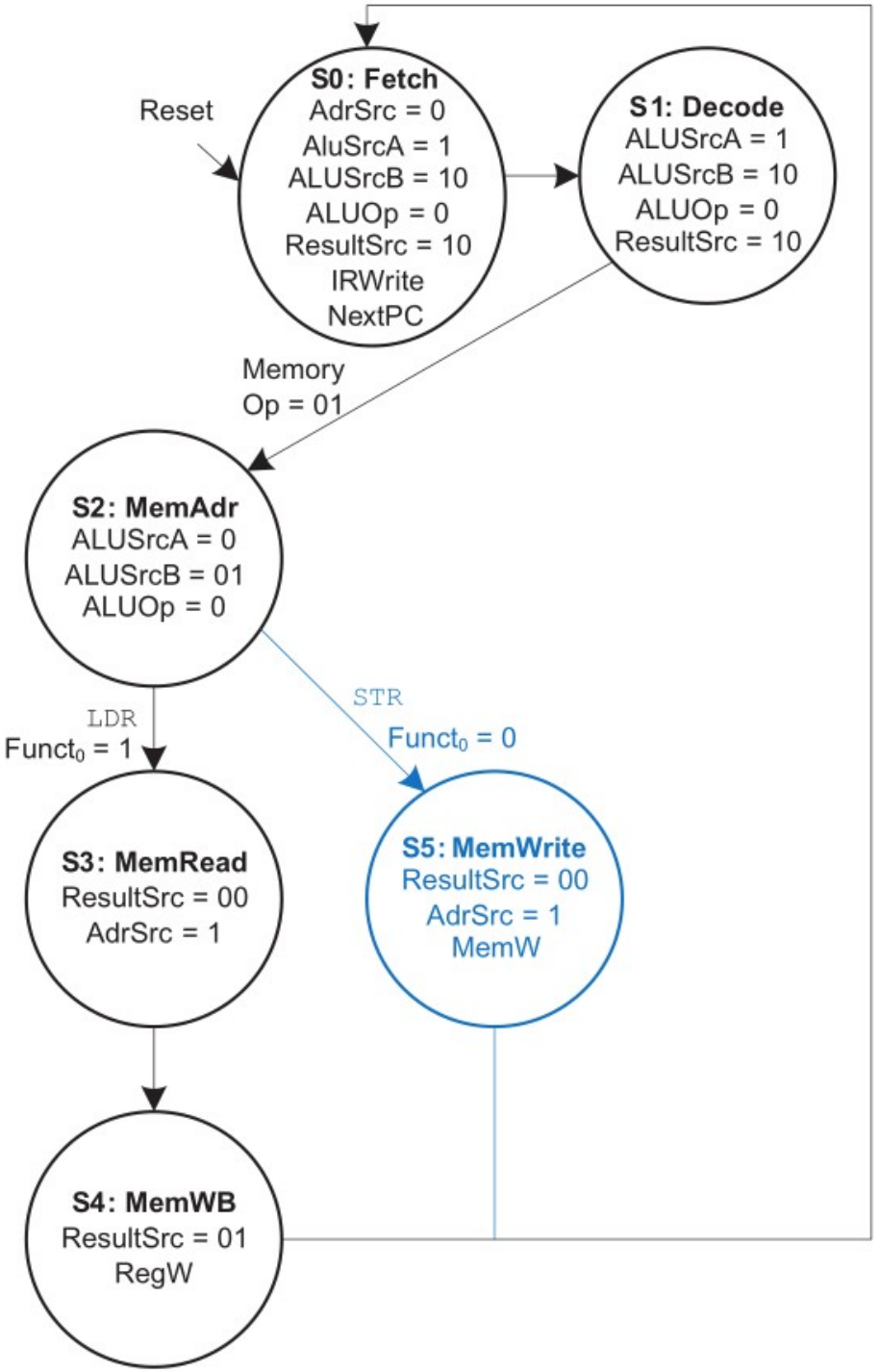


If the instruction is LDR, then the processor must next
1) read data from the memory and
2) write it to the register file.

The value in memory is read and saved in the Data register during the MemRead step.

In the memory writeback step MemWB, Data is written to the register file.

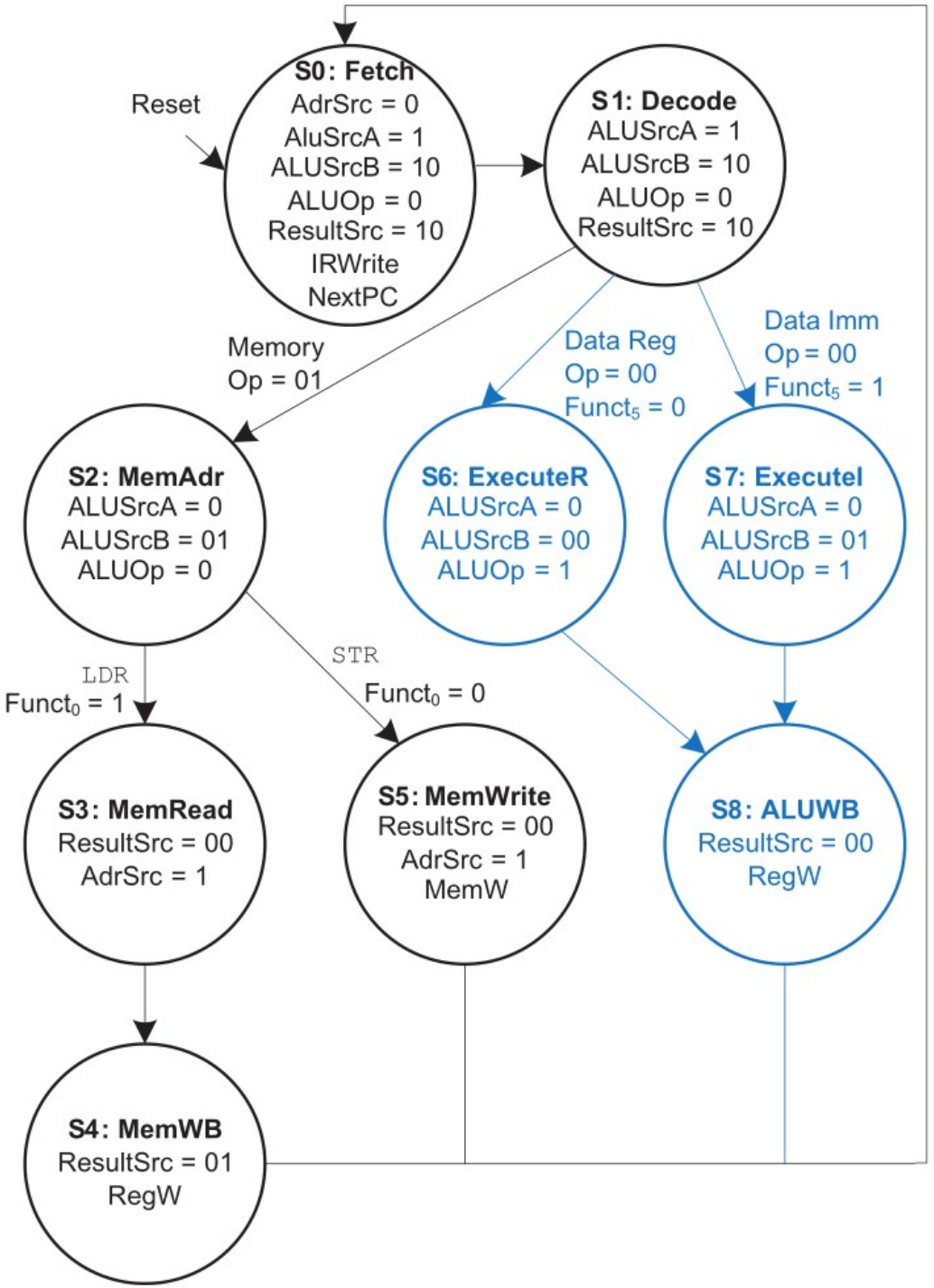
Finally, the FSM returns to the Fetch state to start the next instruction.



The STR instruction:

From the MemAdr state, the data read from the second port of the register file is simply written to memory.

After that, the FSM returns to the Fetch state.

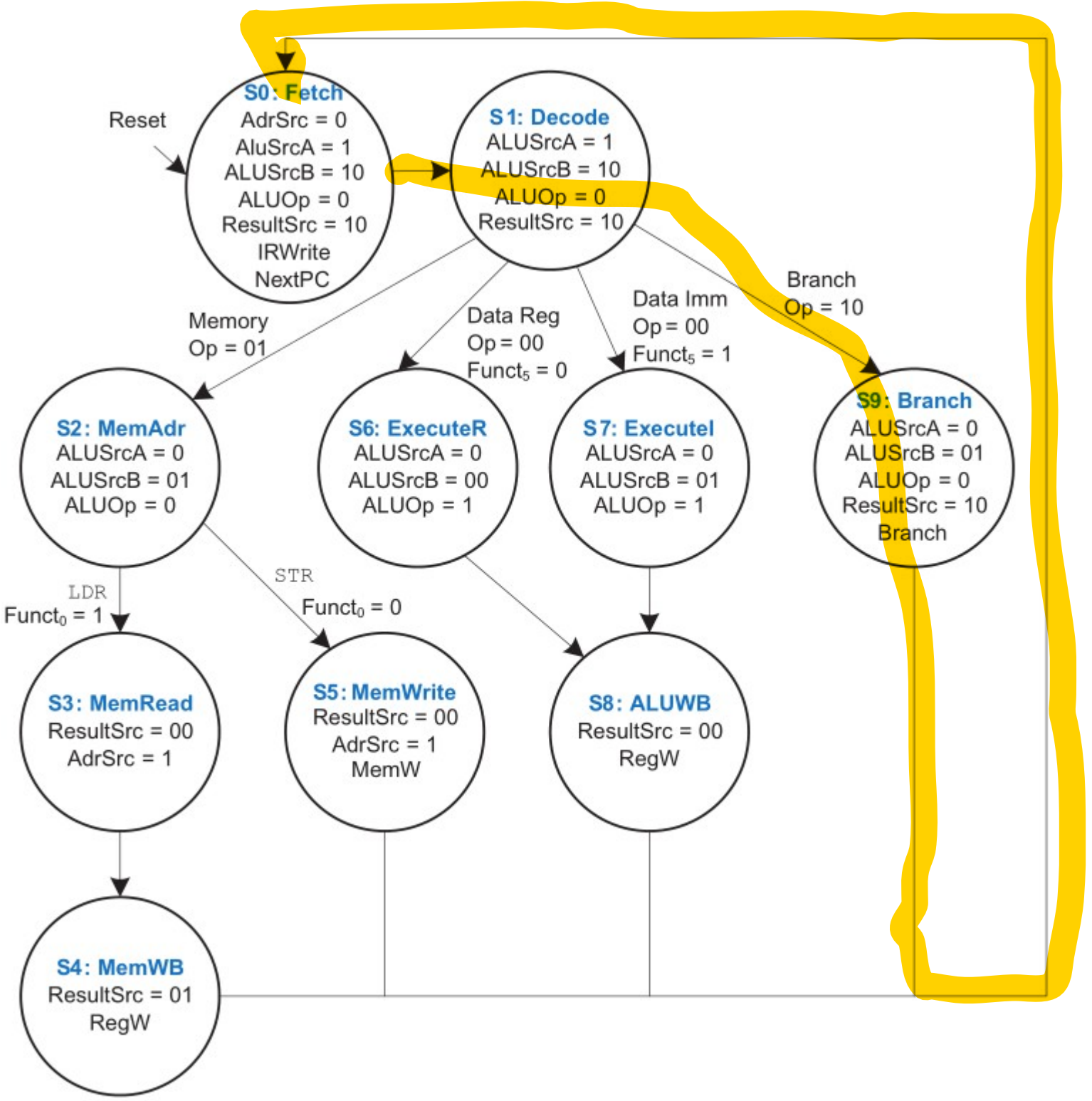


For data-processing instructions, the multicycle processor must

1) calculate the result using the ALU and

2)write that result to the register file.

In the ALU Writeback state (ALUWB), the result is selected from ALUOut and written to the register file.



Complete multicycle control FSM

Performance analysis

$$Execution\ Time = \left(\#instructions\right)\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right)$$

The multicycle processor uses varying numbers of cycles for the various instructions but it does less work in a single cycle and, thus, has a shorter cycle time.

The multicycle processor requires three cycles for branches, four for data-processing instructions and stores, and five for loads.

The CPI depends on the relative likelihood that each instruction is used.

Exercise 15. 1

Visualize the data flows for the MemRead, and the memory writeback steps of the `LDR` instruction.

Exercise 15. 2

Visualize the data flow for the MemWrite step of the `STR` instruction.

Exercise 15. 3

Visualize the data flows for the ExecuteR, ExecuteI, and the ALU writeback steps of the data processing instructions.