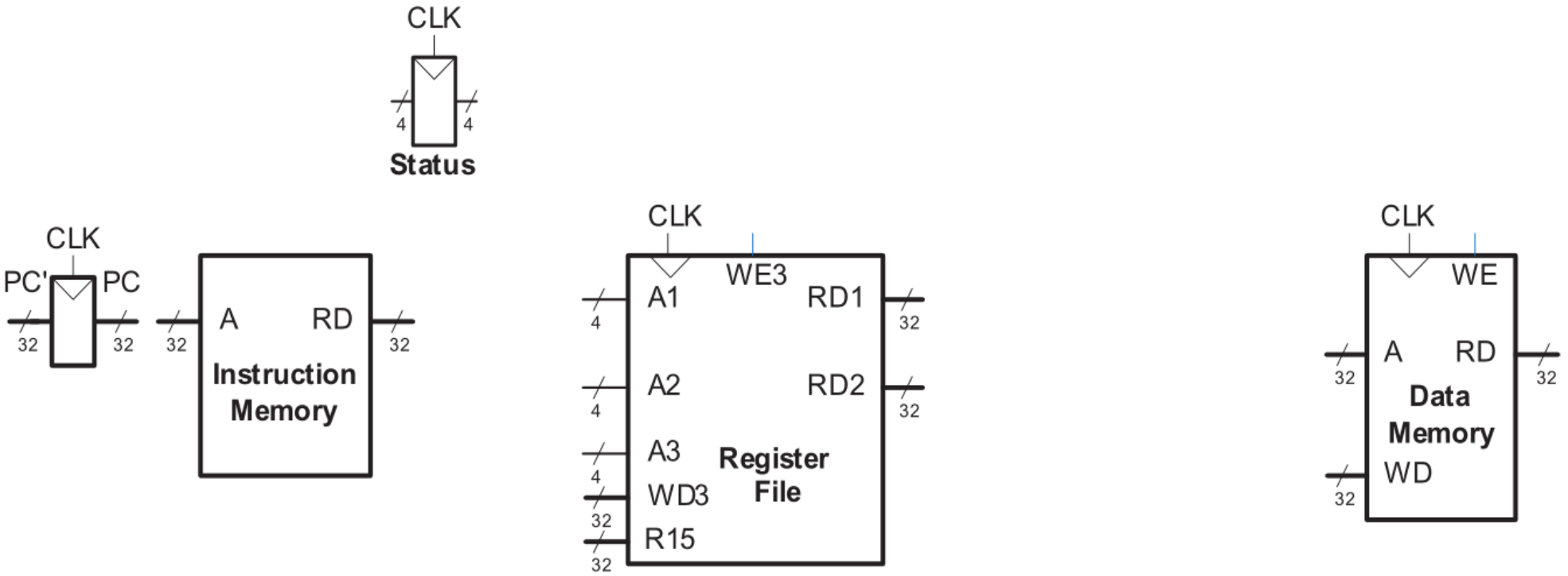
Computer organization and architecture

Lesson 14

Single-cycle processor

Single-cycle microarchitecture executes instructions in a single cycle.

We begin constructing the datapath by connecting the state elements with combinational logic that can execute the various instructions.



Control signals determine which specific instruction is performed by the datapath at any given time.

The control unit contains combinational logic that generates the appropriate control signals based on the current instruction.

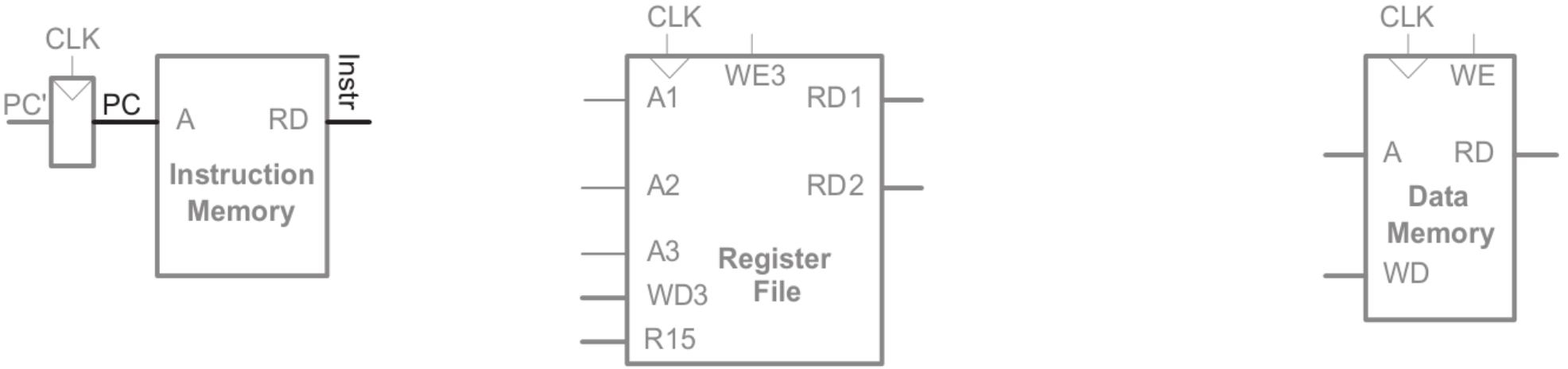
We will gradually develop the single-cycle datapath, adding one piece at a time to the state elements.

The new connections are emphasized in black (or blue, for new control signals), whereas the hardware that has already been studied is shown in gray.

The status register is part of the controller and will be omitted while we focus on the datapath.

The program counter contains the address of the instruction to execute.

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



The PC is connected to the address input of the instruction memory.

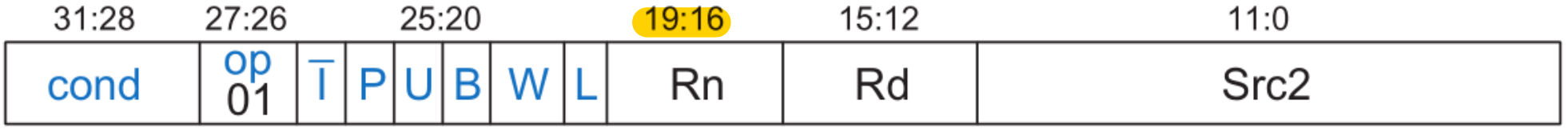
The instruction memory reads out, or **fetches**, the 32-bit instruction, labeled Instr.

The processor's actions depend on the specific instruction that was fetched.

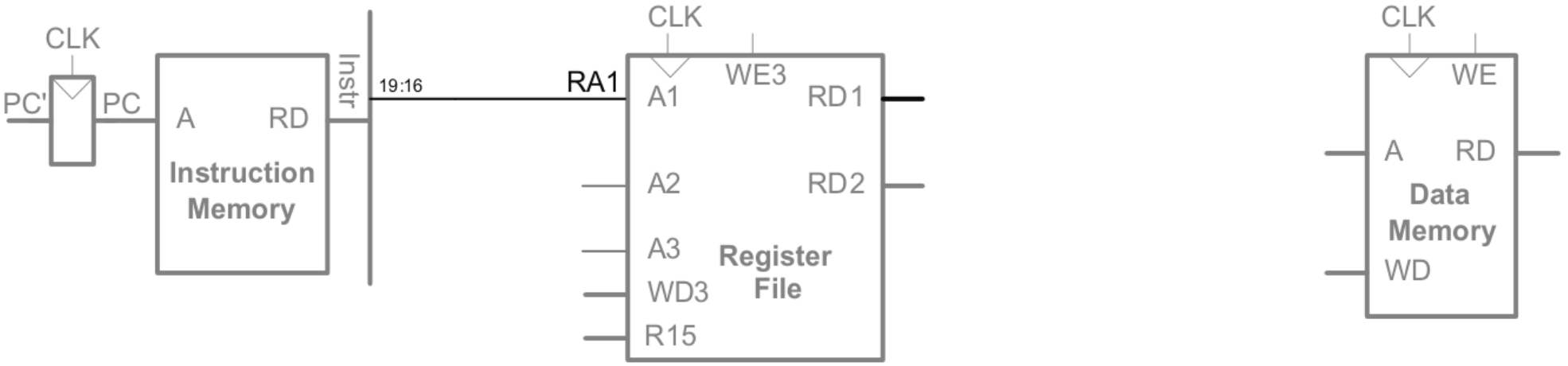
We start with LDR with positive immediate offset.

For the LDR instruction, the next step is to read the source register containing the base address.

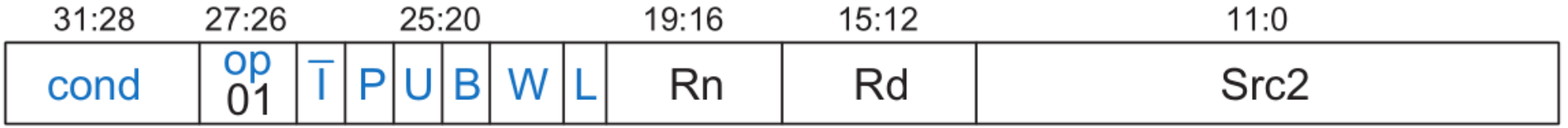
This register is specified in the Rn field of the instruction.



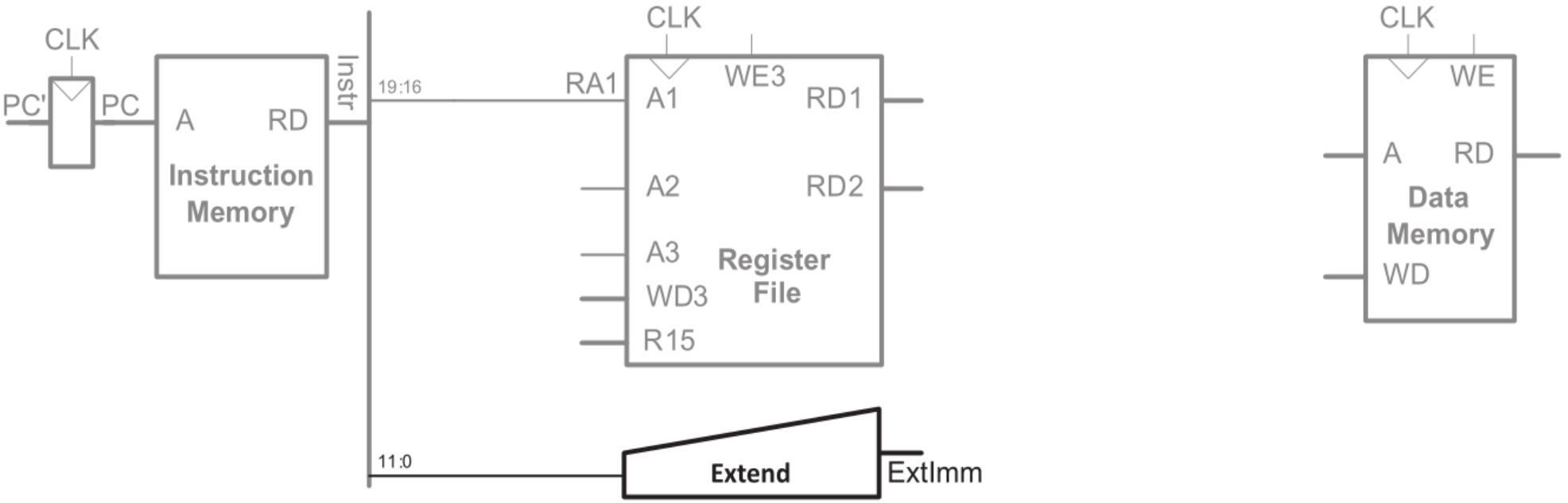
The bits 19:16 of the instruction are connected to the address input of one of the register file ports, A1.



The register file reads the register value onto RD1.



The offset, stored in the bits 11:0, is an unsigned value, so it must be zero-extended to 32 bits.

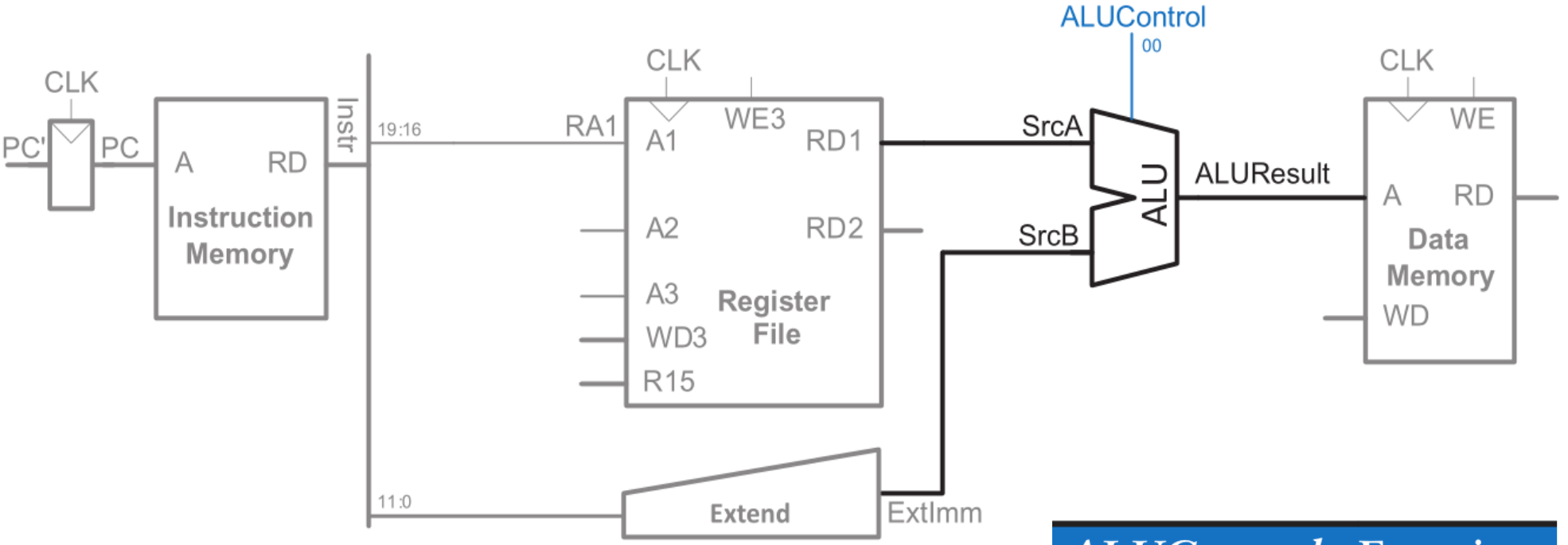


The 32-bit value is called ExtImm

$$ImmExt_{31:12} = 0$$

$$ImmExt_{11:0} = Instr_{11:0}$$

The processor must add the base address to the offset to find the address to read from memory.

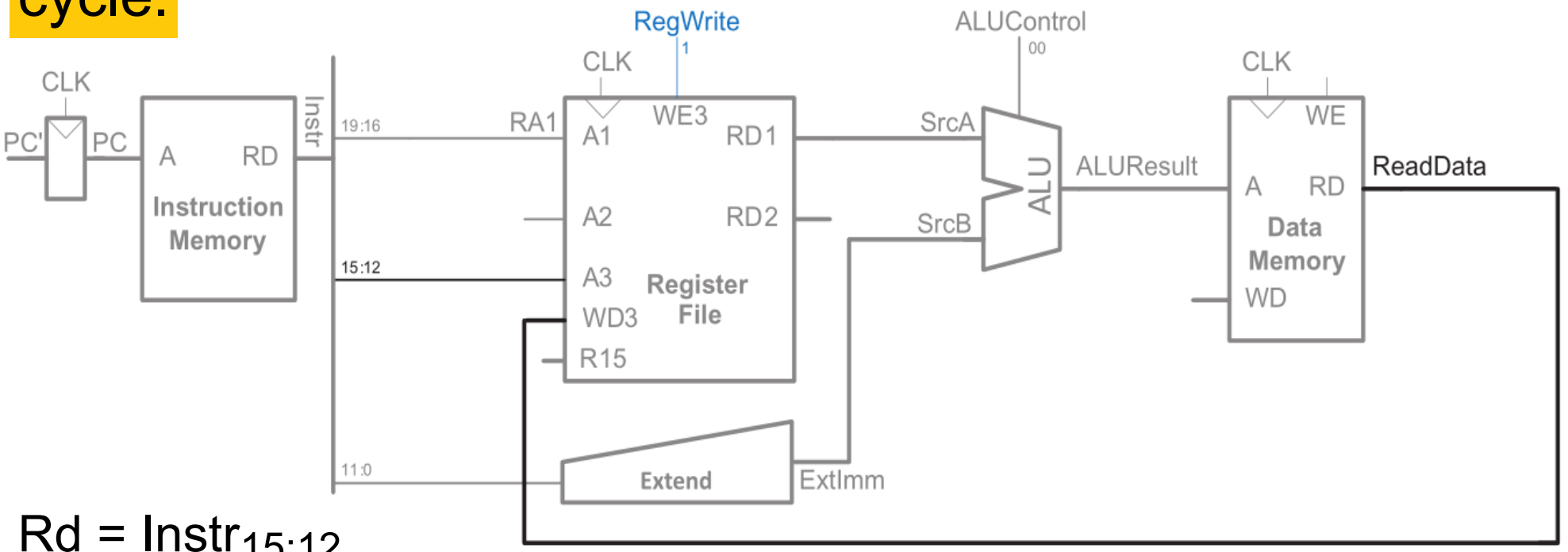


For an LDR instruction, ALUControl should be set to 00 to perform addition

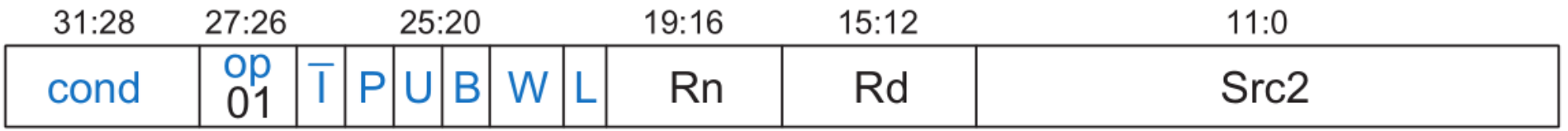
ALUResult is sent to the data memory as the address to read.

ALUControl	Function
00	Add
01	Subtract
10	AND
11	OR

The data is read from the data memory onto the ReadData bus and then written back to the destination register on the rising edge of the clock at the end of the cycle.

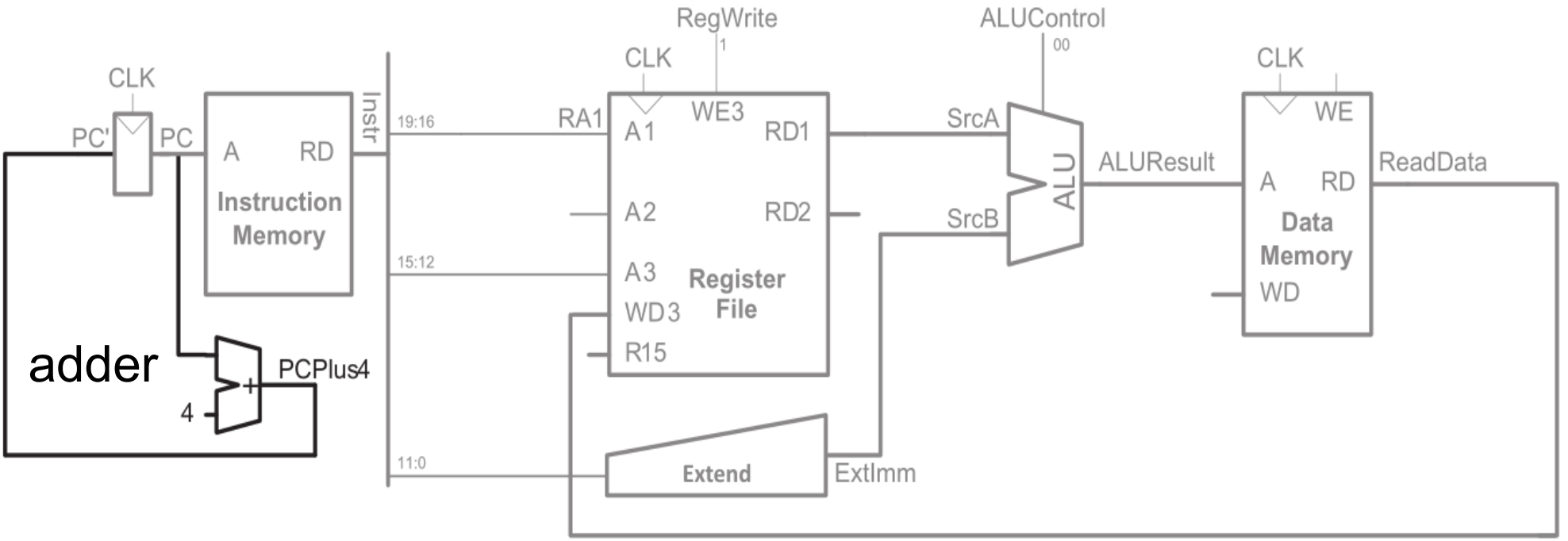


Rd = $Instr_{15:12}$



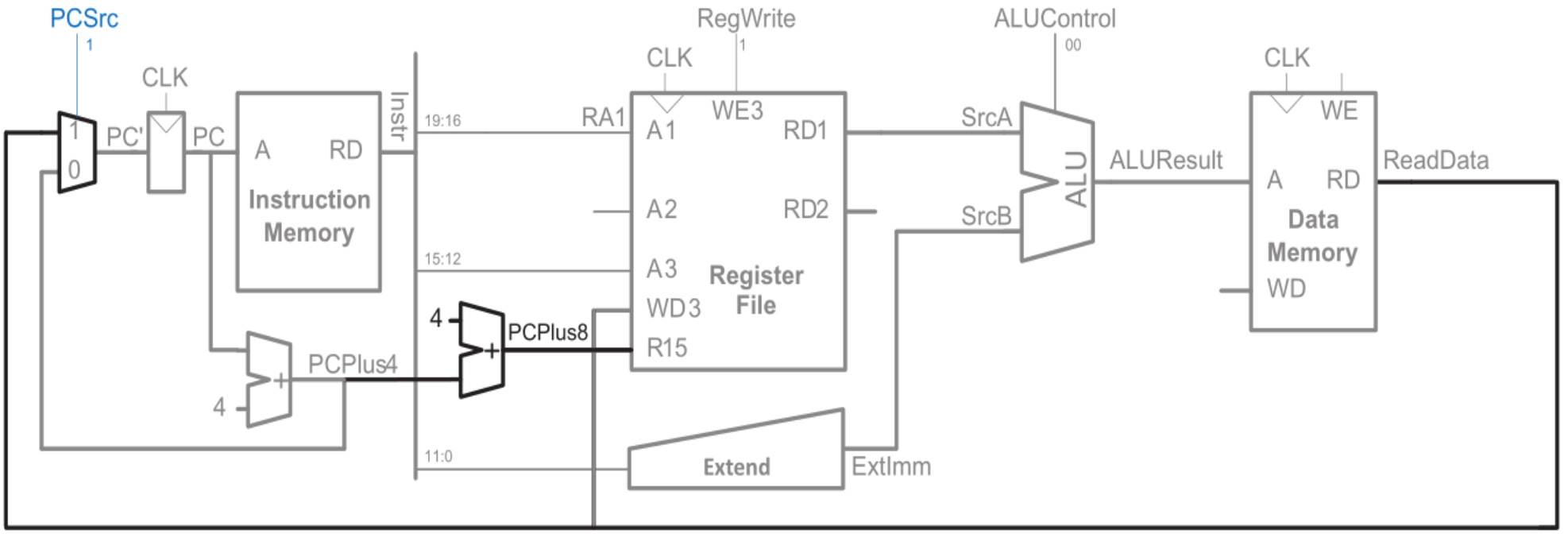
A control signal RegWrite is asserted so that the data value is written into the register file.

While the instruction is being executed, the processor must compute the address of the next instruction, PC'



The new address is written into the program counter on the next rising edge of the clock.

This completes the datapath for the LDR instruction, except for the case of the base or destination register being R15.



In the ARM architecture, reading register R15 returns PC+8

Another adder further increments the PC by 4.

Writing register R15 updates the PC.

PC' may come from the result of the instruction (ReadData) rather than PCPlus4.

The PCSrc control signal is set to 0 to choose PCPlus4 or 1 to choose ReadData.

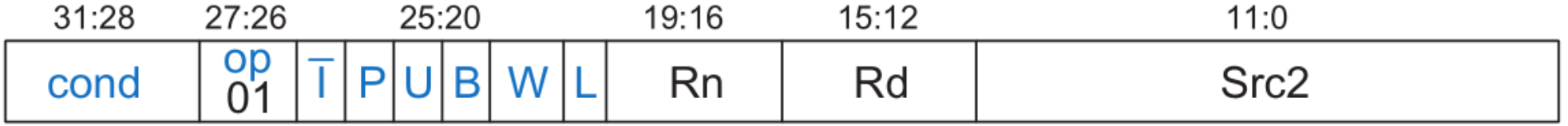
Extend the datapath to also handle the `STR` instruction.

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

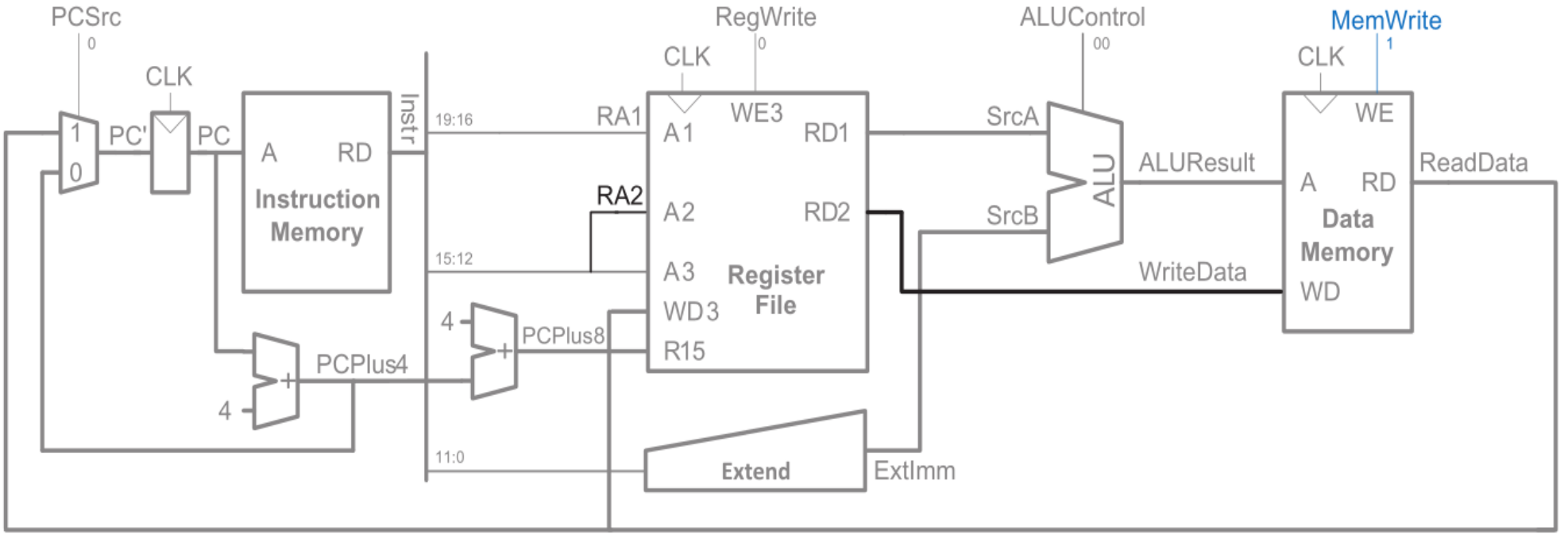
The ALU adds the base address to the immediate to find the memory address.

All of these functions are already supported in the datapath.

The STR instruction also reads a second register from the register file and writes it to the data memory.



The Rd value is read onto the RD2 port.



MemWrite=1 to write the data to memory; ALUControl=00 to add the base address and offset; RegWrite=0, because nothing should be written to the register file.

Extend the datapath to handle the data-processing instructions, `ADD` , `SUB` , `AND` , `ORR` , using the immediate addressing mode.

These instructions

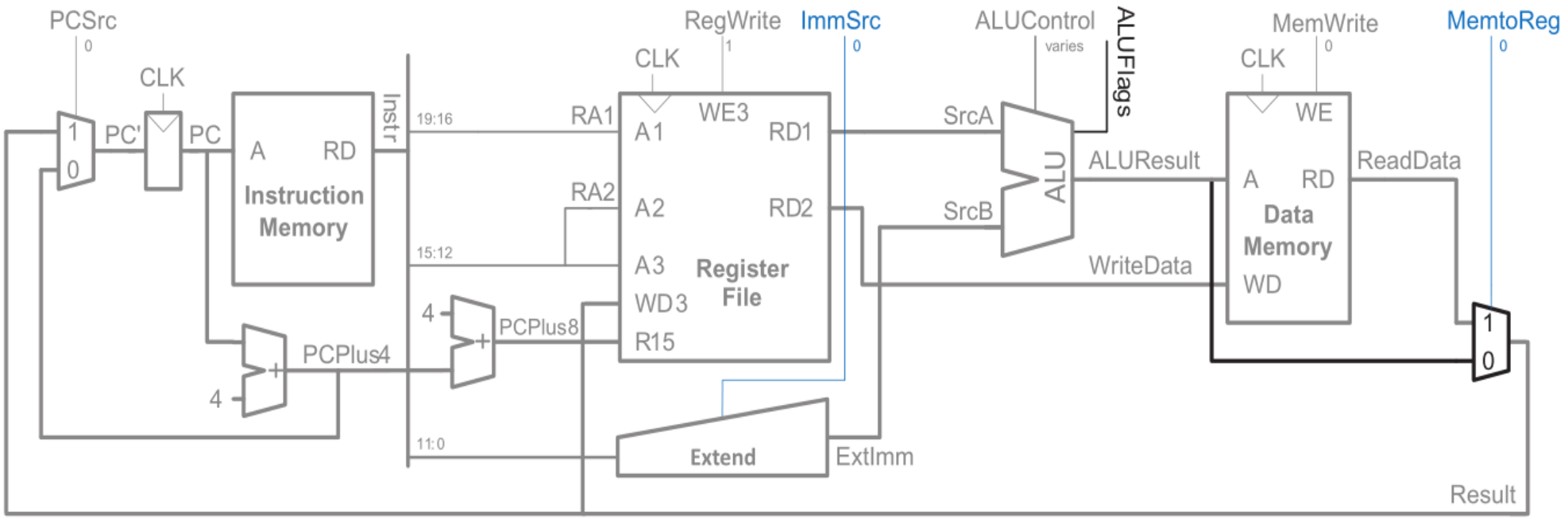
- read a source register from the register file
- read an immediate from the low bits of the instruction
- perform some ALU operation on them
- write the result back to a third register

They differ only in the specific ALU operation.

Hence, they can all be handled with the same hardware using different ALUControl signals.

The ALU also produces four flags, ALUFlags 3:0 (Zero, Negative, Carry, oVerflow), that are sent back to the controller.

Datapath enhancements for data-processing instructions with immediate addressing

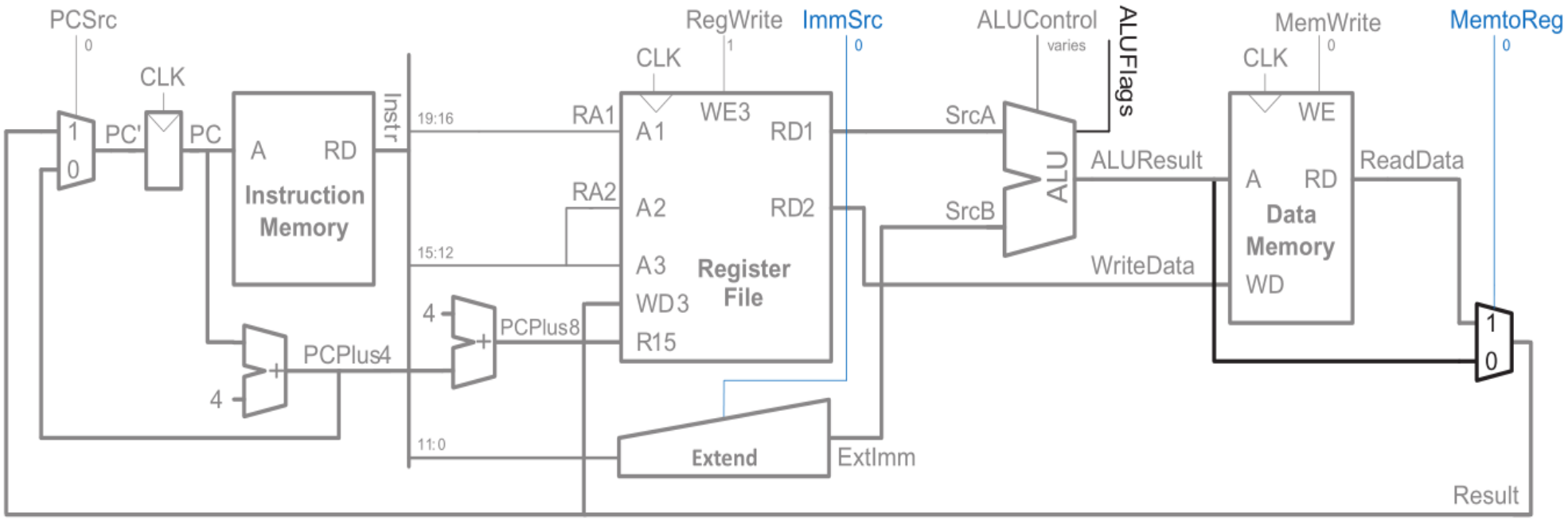


The 1st ALU source is read from port 1 of the register file and the immediate is extended from the low bits of Instr.

Data-processing instructions use only an 8-bit immediate.

When ImmSrc is 0, ExtImm is zero-extended from Instr 7:0 for data-processing instructions.

When ImmSrc is 1, ExtImm is zero-extended from Instr 11:0 for LDR or STR.



For LDR , the register file received its write data from the data memory.

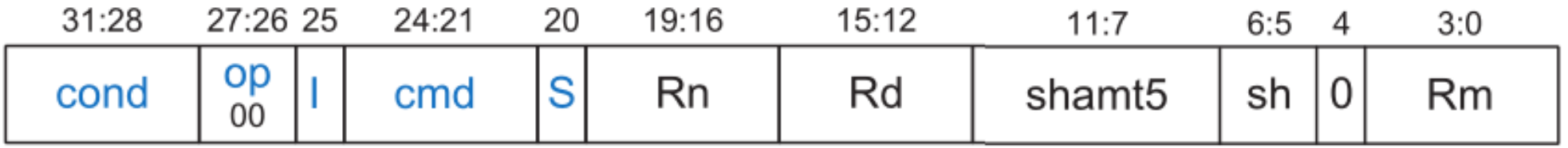
Data-processing instructions write ALUResult to the register file.

A multiplexer with a MemtoReg signal is added to choose between ReadData and ALUResult.

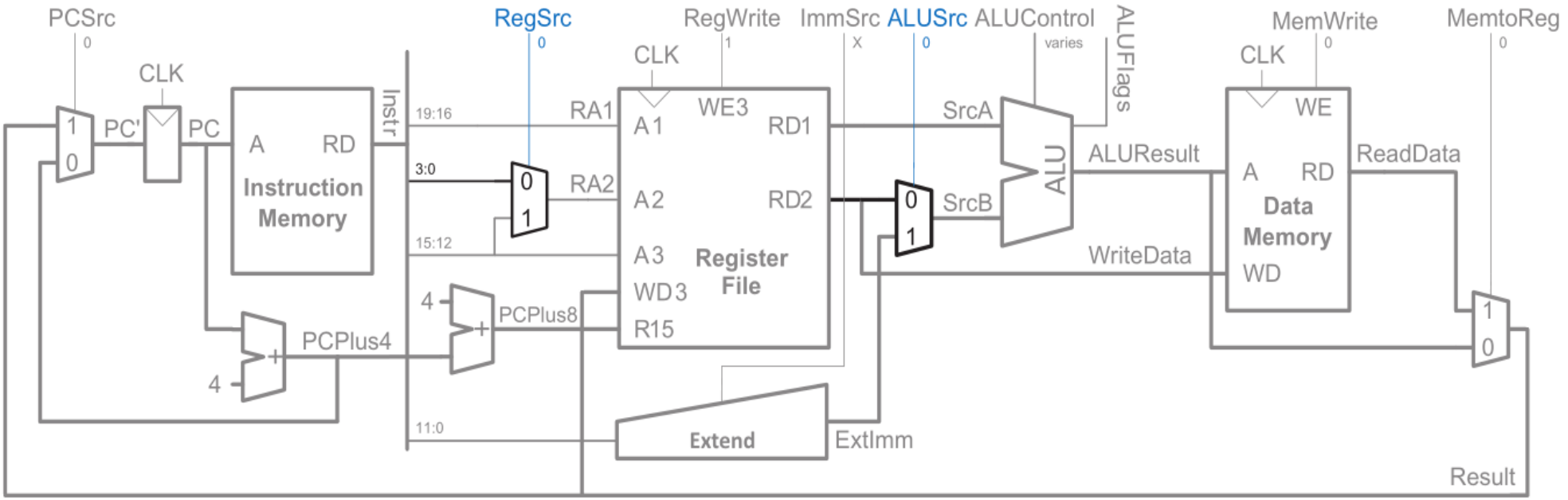
MemtoReg is 0 for data-processing instructions.

Let's add data-processing instructions with register addressing.

Data-processing instructions with register addressing receive their 2nd source from Rm, specified by $Instr_{3:0}$



We must add multiplexers on the inputs of the register file and ALU to select this second source register.



RA2 is chosen from the Rd field ($Instr_{15:12}$) for STR and the Rm field ($Instr_{3:0}$) for data-processing instructions with register addressing based on the RegSrc control signal.

Based on the ALUSrc control signal, the second source to the ALU is selected from ExtImm for instructions using immediates and from the register file for data-processing instructions with register addressing.

Extend the datapath to handle the B instruction

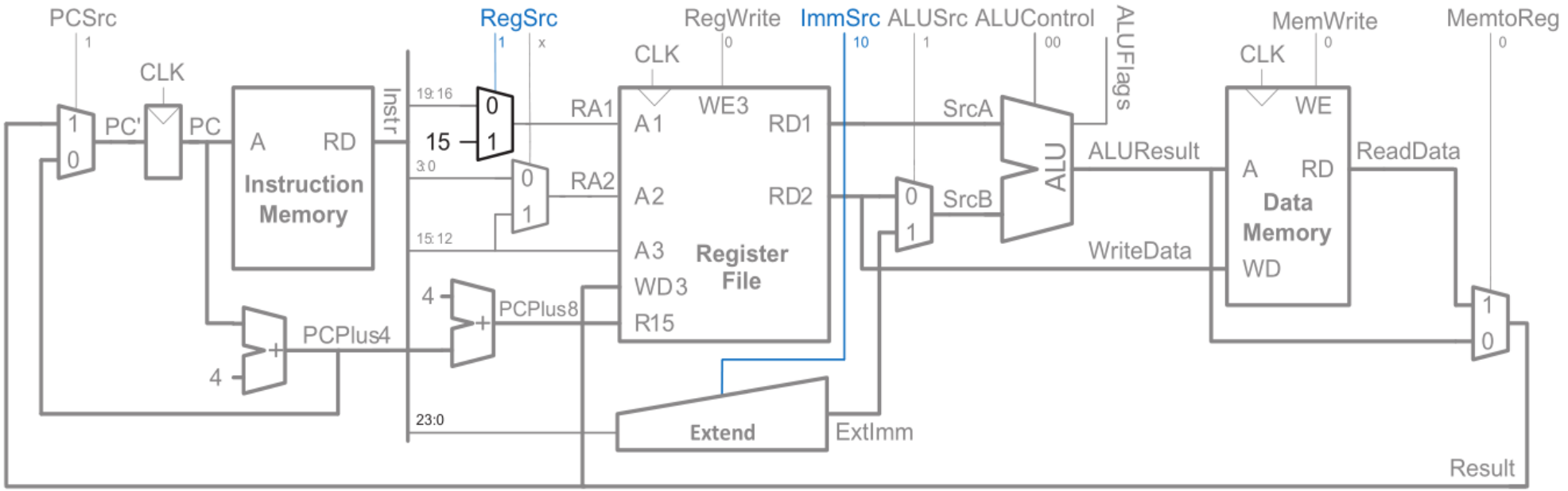
The branch instruction adds a 24-bit immediate to PC+8 and writes the result back to the PC.

The immediate is multiplied by 4 and sign extended.

Therefore, the Extend logic needs yet another mode.

ImmSrc is increased to 2 bits

ImmSrc	ExtImm	Description
00	{24 0s} $Instr_{7:0}$	8-bit unsigned immediate for data-processing
01	{20 0s} $Instr_{11:0}$	12-bit unsigned immediate for LDR/STR
10	{6 $Instr_{23}$} $Instr_{23:0}$ 00	24-bit signed immediate multiplied by 4 for B



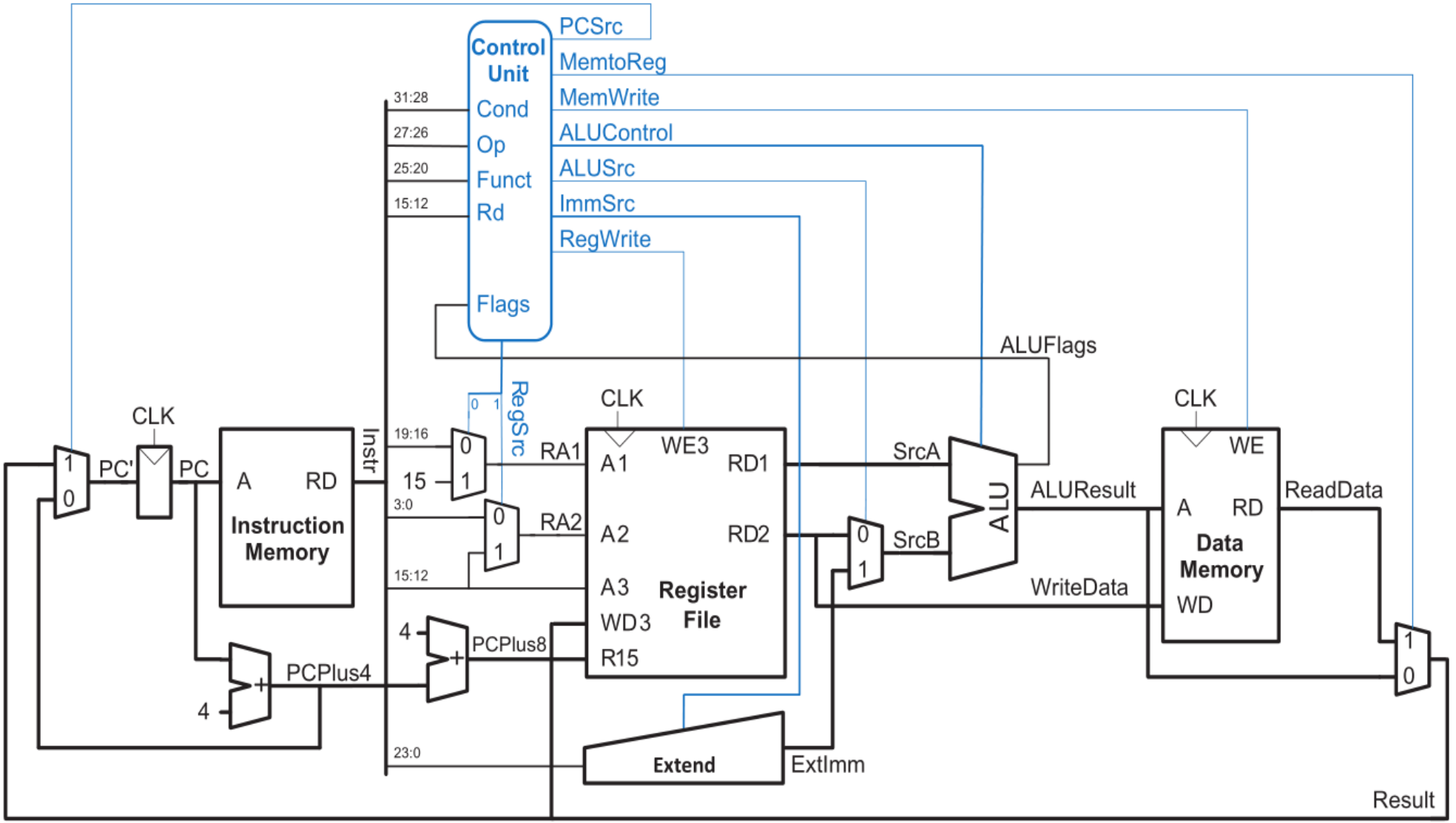
PC+8 is read from the first port of the register file.

Therefore, a multiplexer is needed to choose R15 as the RA1 input.

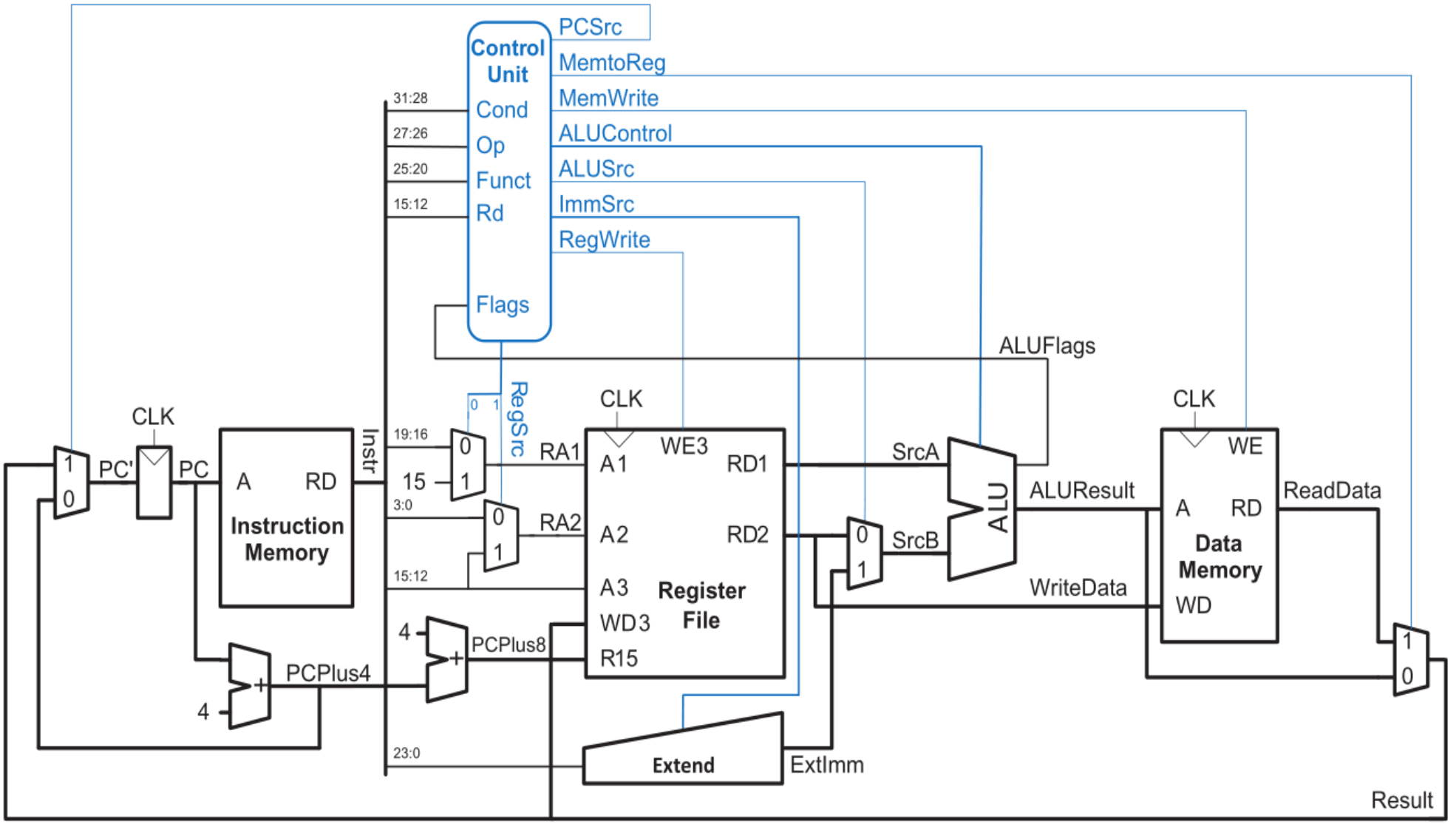
It is controlled by another bit of RegSrc, choosing $Instr_{19:16}$ for most instructions but 15 for B

MemtoReg is set to 0 and PCSrc is set to 1 to select the new PC from ALUResult for the branch

The control unit computes the control signals based on the cond, op, funct fields of the instruction ($Instr_{31:28}$, $Instr_{27:26}$, $Instr_{25:20}$) as well as the flags and whether the destination register is the PC.

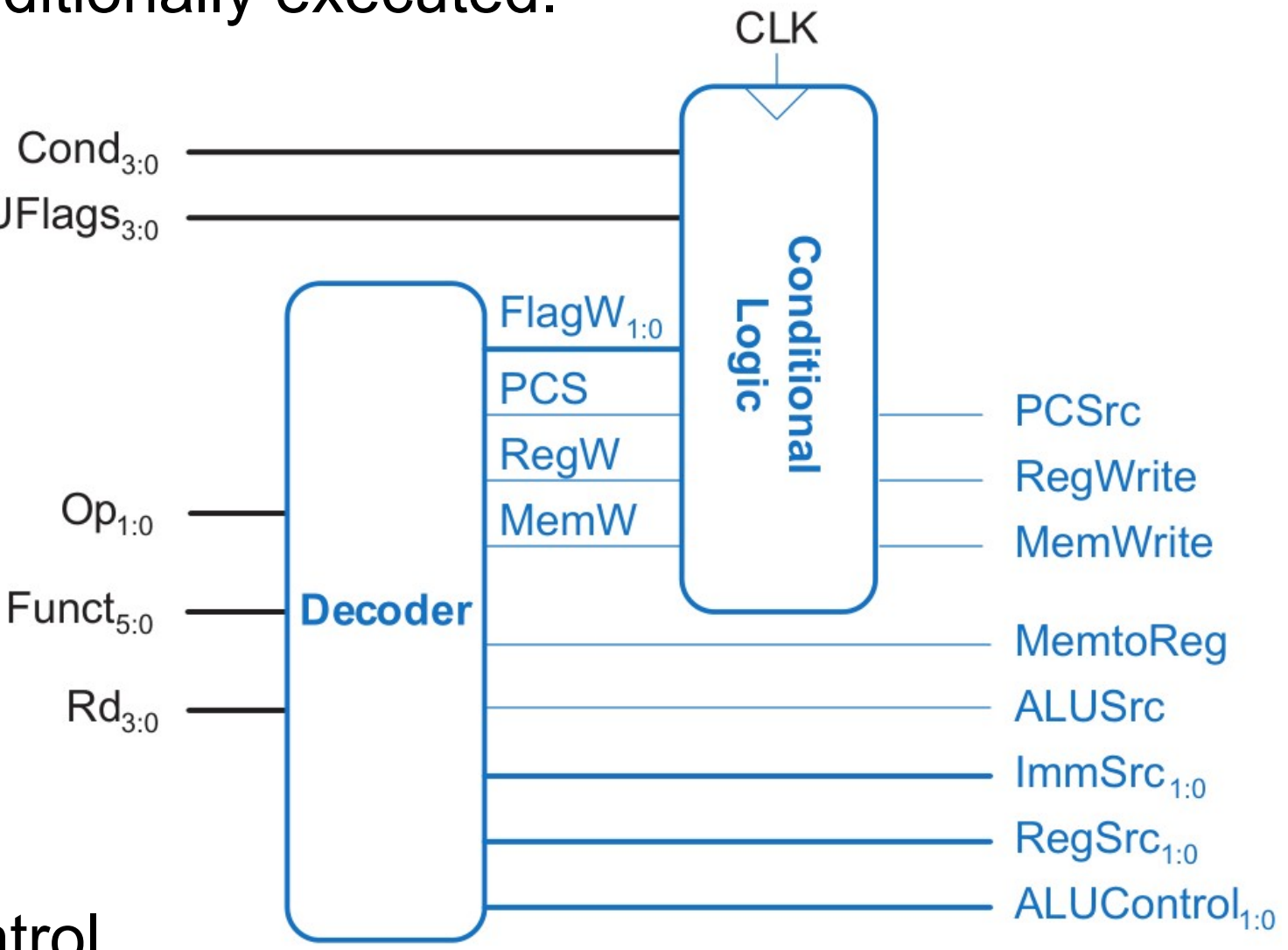


The controller also stores the current status flags and updates them appropriately.

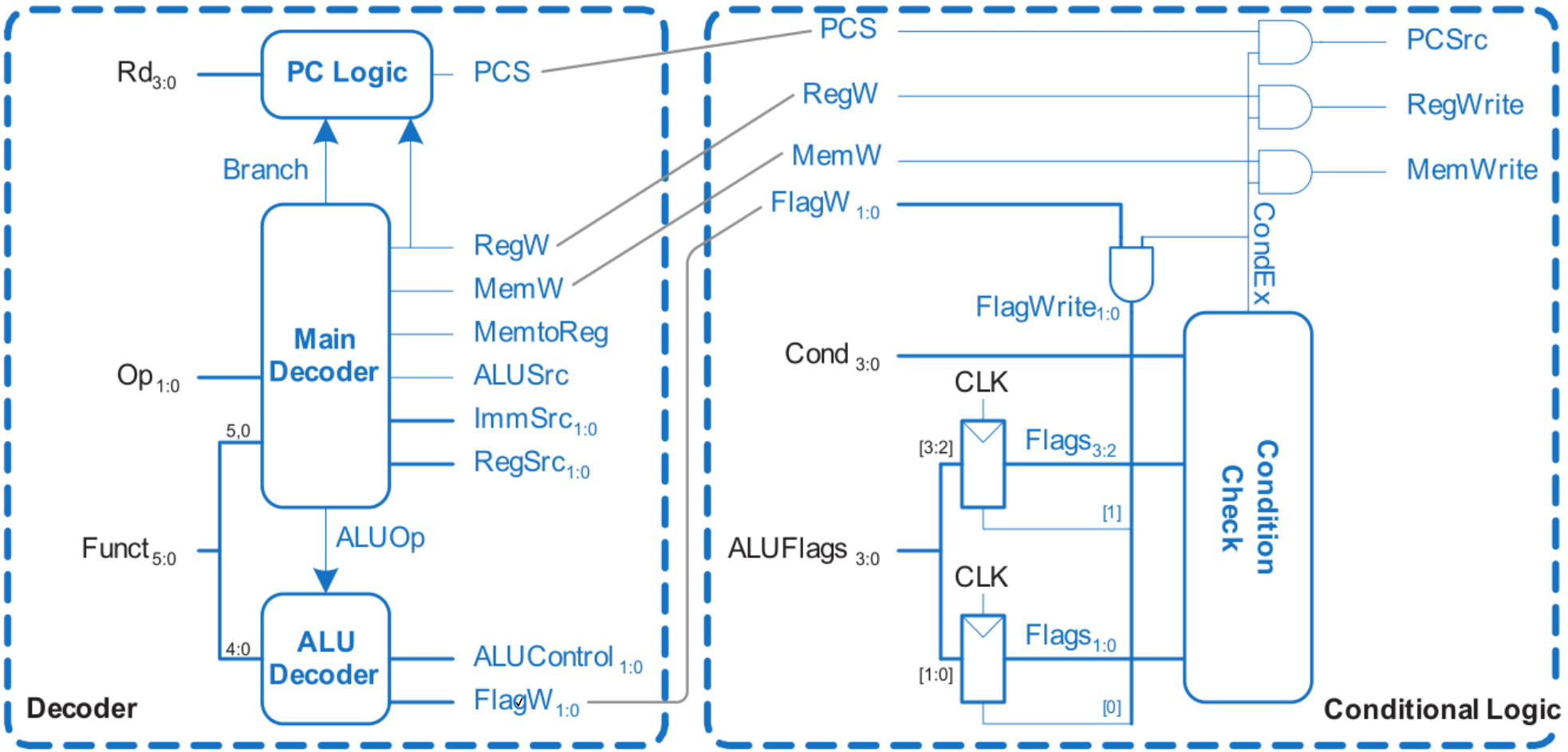


The **conditional logic** maintains the status flags and only enables updates to architectural state when the instruction should be conditionally executed.

Control unit



The **decoder** generates control signals based on Instr.



The **main decoder** produces most of the control signals.

The **ALU decoder** uses the Funct field to determine the type of data-processing instruction.

The **PC Logic** determines whether the PC needs updating due to a branch or a write to R15.

The truth table of the main decoder:

Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
10	X	X	B	1	0	0	1	10	0	X1	0

The main decoder determines the type of instruction: data-processing register, data-processing immediate, STR , LDR , B

It sends MemtoReg, ALUSrc, $ImmSrc_{1:0}$, and $RegSrc_{1:0}$ directly to the datapath.

The truth table of the main decoder:

Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
10	X	X	B	1	0	0	1	10	0	X1	0

The write enable signals MemW and RegW must pass through the conditional logic before becoming datapath signals MemWrite and RegWrite.

These write enables may be killed (reset to 0) by the conditional logic if the condition is not satisfied.

The main decoder also generates the Branch and ALUOp signals, which are used within the controller to indicate that the instruction is B or data-processing, respectively.

The truth table of the ALUDecoder:

$ALUOp$	$Funct_{4:1}$ (cmd)	$Funct_0$ (S)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00 (Add)	00
1	0100	0	ADD	00 (Add)	00
		1			11
	0010	0	SUB	01 (Sub)	00
		1			11
	0000	0	AND	10 (And)	00
		1			10
	1100	0	ORR	11 (Or)	00
		1			10

The ALUControl asserts FlagW to update the status flags when the S-bit is set.

`ADD` , `SUB` update all flags, whereas `AND` , `ORR` only update the N and Z flags.

The truth table of the ALUDecoder:

$ALUOp$	$Funct_{4:1}$ (cmd)	$Funct_0$ (S)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00 (Add)	00
1	0100	0	ADD	00 (Add)	00
		1			11
	0010	0	SUB	01 (Sub)	00
		1			11
	0000	0	AND	10 (And)	00
		1			10
	1100	0	ORR	11 (Or)	00
		1			10

$FlagW_1$ for updating N , Z ($Flags_{3:2}$)
$FlagW_0$ for updating C , V ($Flags_{1:0}$)
$FlagW_{1:0}$ is killed by the conditional logic when the condition is not satisfied (CondEx=0)

The PC Logic checks if the instruction is a write to R15 or a branch such that the PC should be updated

```
PCS = ((Rd == 15) & RegW) | Branch
```

PCS may be killed by the conditional logic before it is sent to the datapath as PCSrc.

The **conditional logic** determines whether the instruction should be executed (CondEx) based on the cond field and the current values of the N, Z, C, and V flags ($Flags_{3:0}$).

If the instruction should not be executed, the write enables and PCSrc are forced to 0 so that the instruction does not change the architectural state.

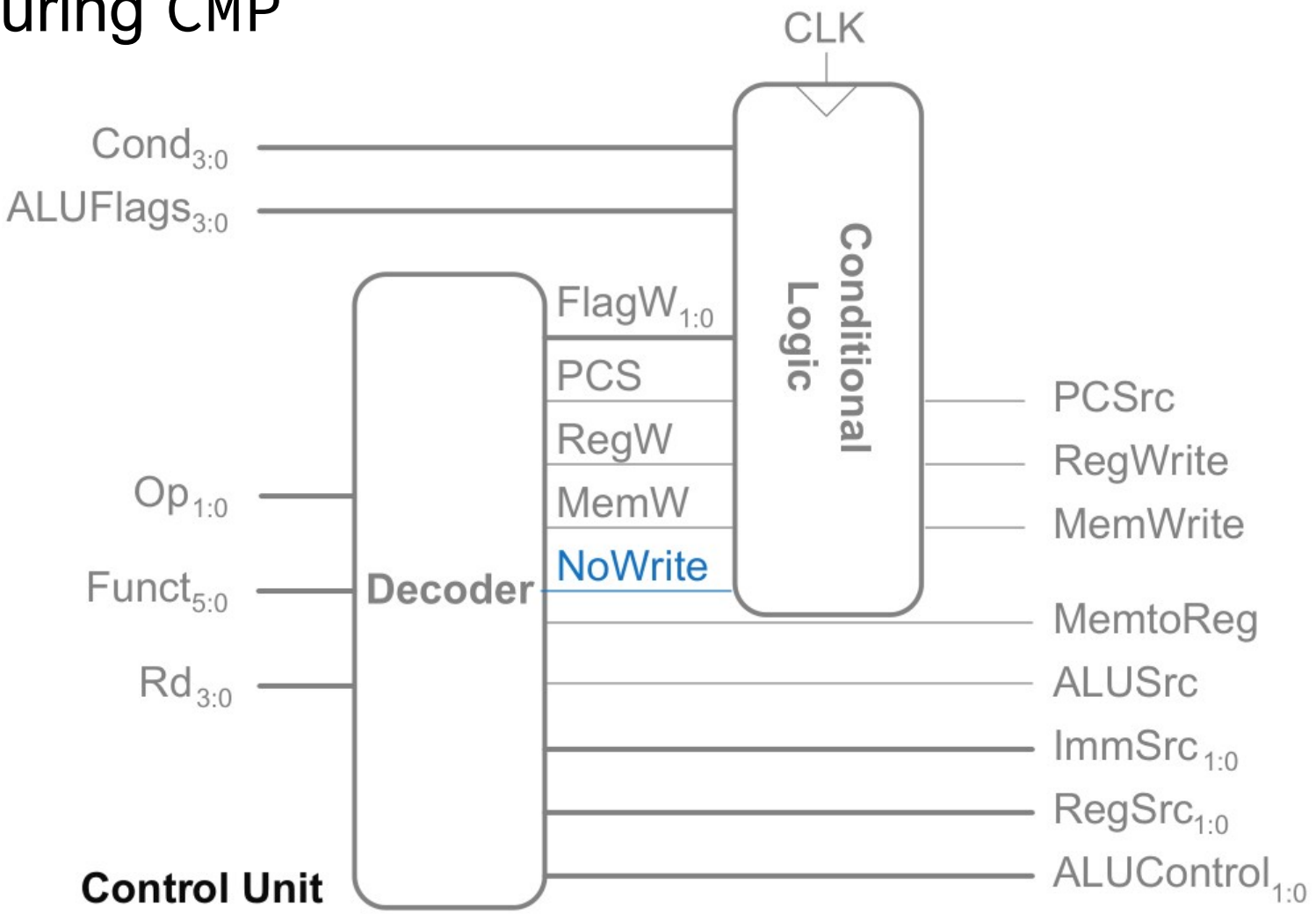
The conditional logic also updates some or all of the flags from the ALUFlags when FlagW is asserted by the ALU Decoder and the instruction's condition is satisfied (CondEx =1).

Let's add support for the `CMP` instruction.

`CMP` subtracts SrcB from SrcA and sets the flags but does not write the difference to a register.

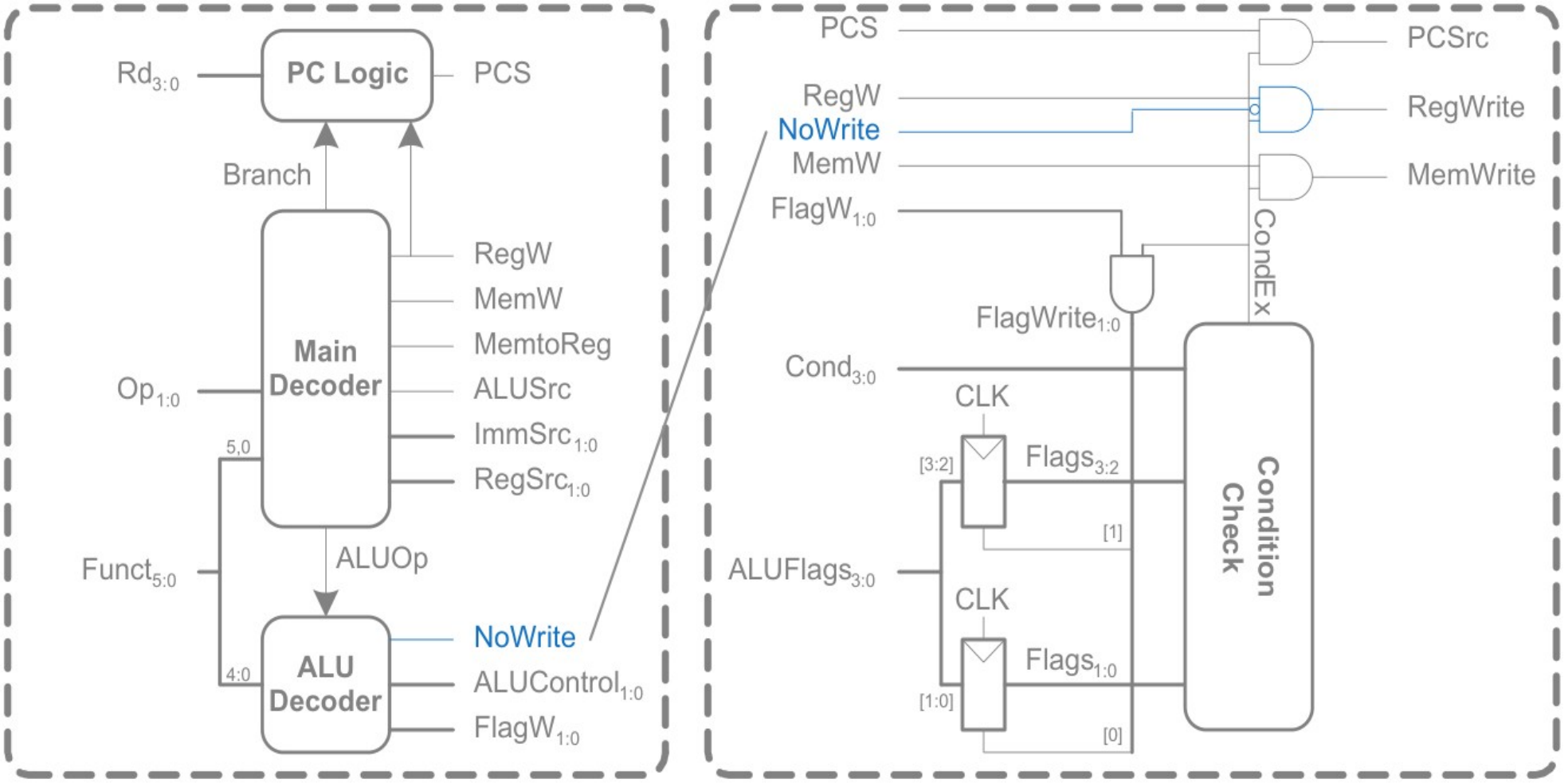
The datapath is already capable of this task.

Introduce a new control signal called NoWrite to prevent writing Rd during `CMP`



This signal would also be helpful for other instructions such as `TST` that do not write a register.

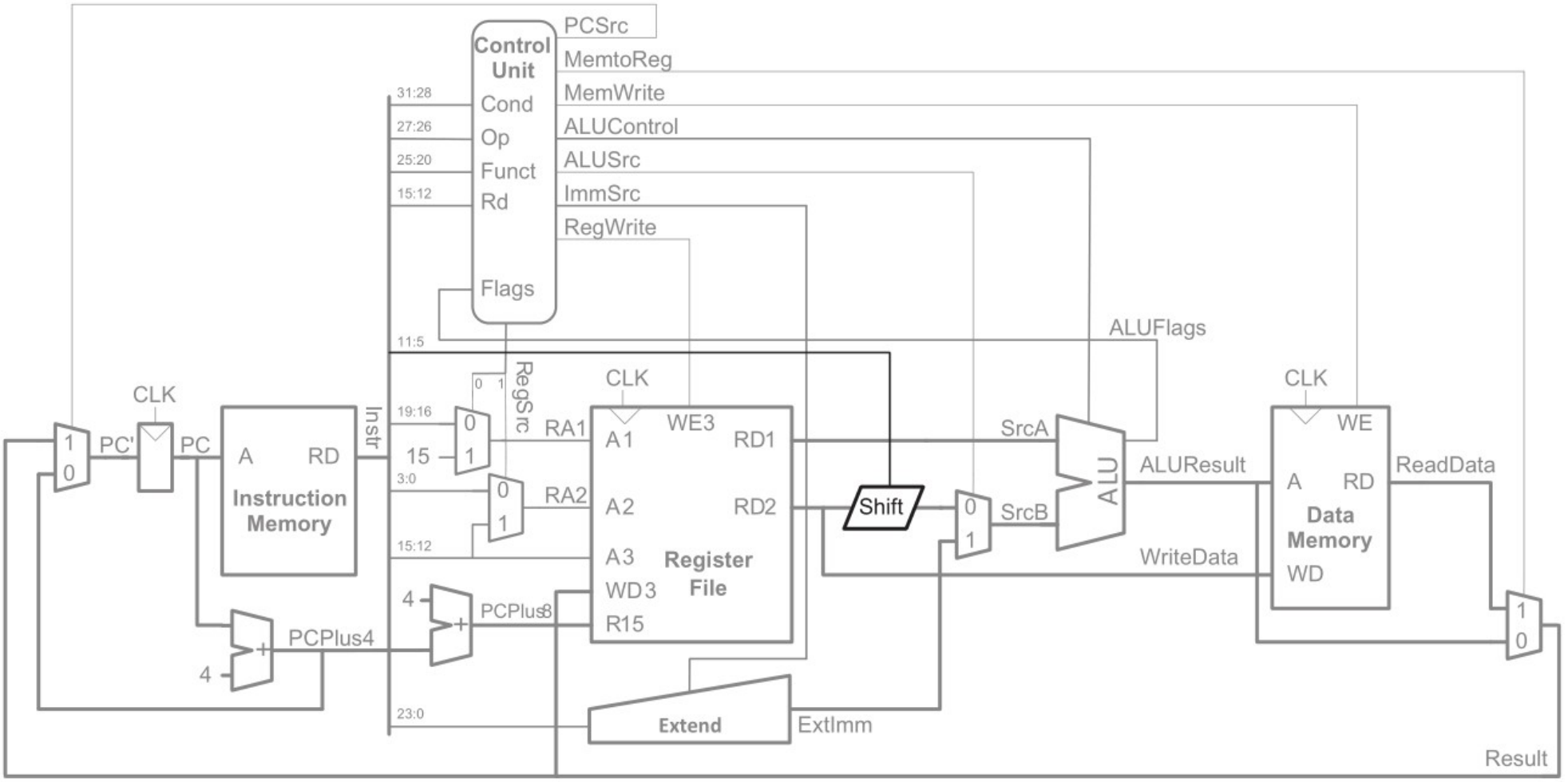
We extend the ALUDecoder to produce the NoWrite signal and the RegWrite logic to accept it.



ALU Decoder truth table enhanced for CMP:

ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Notes	$ALUControl_{1:0}$	$FlagW_{1:0}$	*NoWrite*
0	X	X	Not DP	00	00	0
1	0100	0	ADD	00	00	0
		1			11	0
	0010	0	SUB	01	00	0
		1			11	0
	0000	0	AND	10	00	0
		1			10	0
	1100	0	ORR	11	00	0
		1			10	0
	1010	1	CMP	01	11	1

Enhance the single-cycle processor to support a shift by an immediate: insert a shifter before the ALU



The shifter uses $Instr_{11:7}$ to specify the shift amount and $Instr_{6:5}$ to specify the shift type.

We see that adding some instructions simply requires enhancing the decoders, as in the case with `CMP`.

Supporting other instructions (like shifts) also requires new hardware in the datapath.

With enough effort, you can extend the single-cycle processor to handle every ARM instruction.

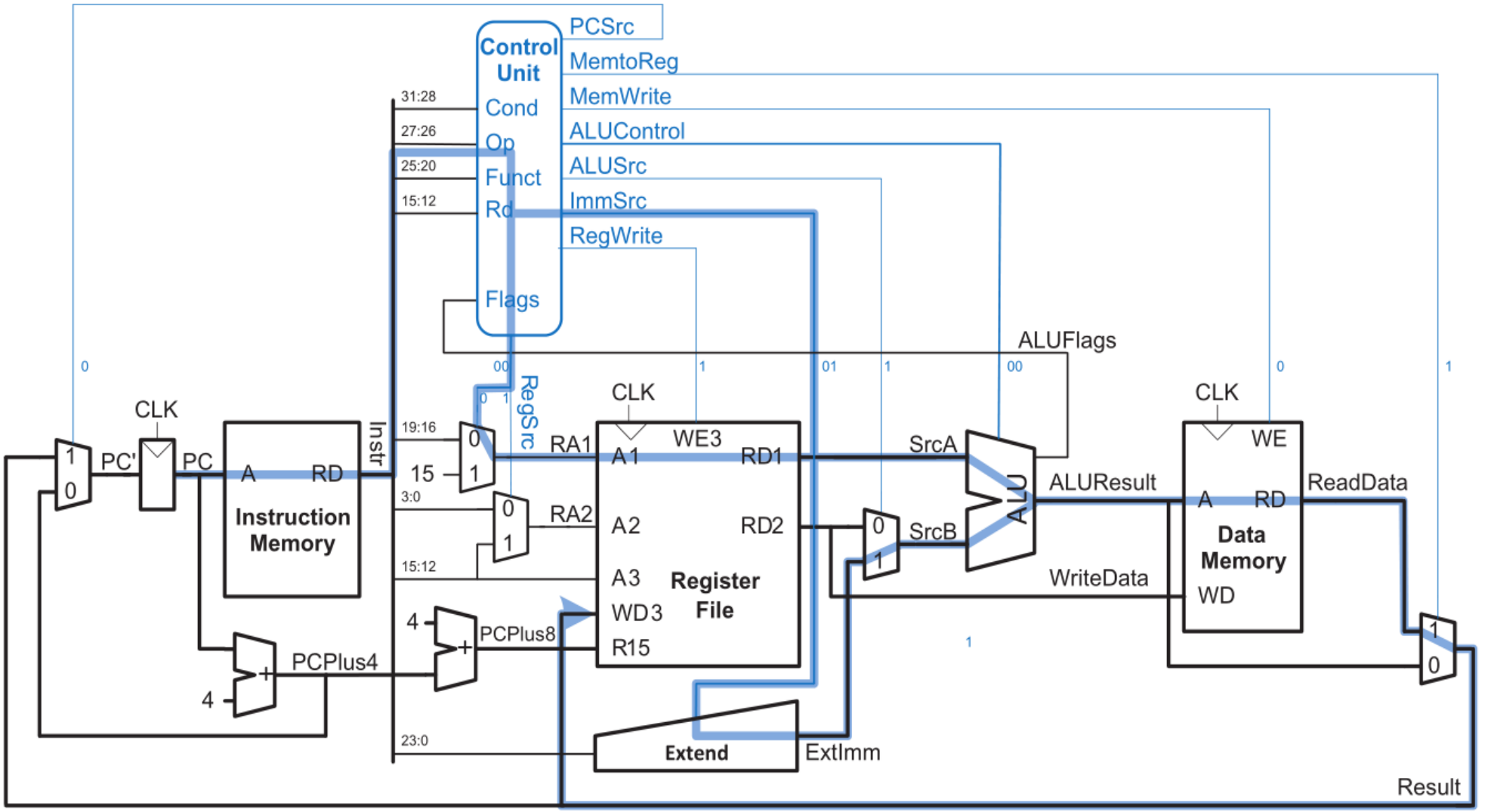
Performance analysis

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

Each instruction in the single-cycle processor takes one clock cycle:

$$CPI = 1$$

The critical paths for the LDR instruction:



It starts with the PC loading a new address on the rising edge of the clock.

The instruction memory reads the new instruction.

The main decoder computes $RegSrc_0$, which drives the multiplexer to choose $Instr_{19:16}$ as RA1, and the register file reads this register as SrcA.

While the register file is reading, the immediate field is zero-extended and selected at the ALUSrc multiplexer to determine SrcB.

The ALU adds SrcA and SrcB to find the effective address.

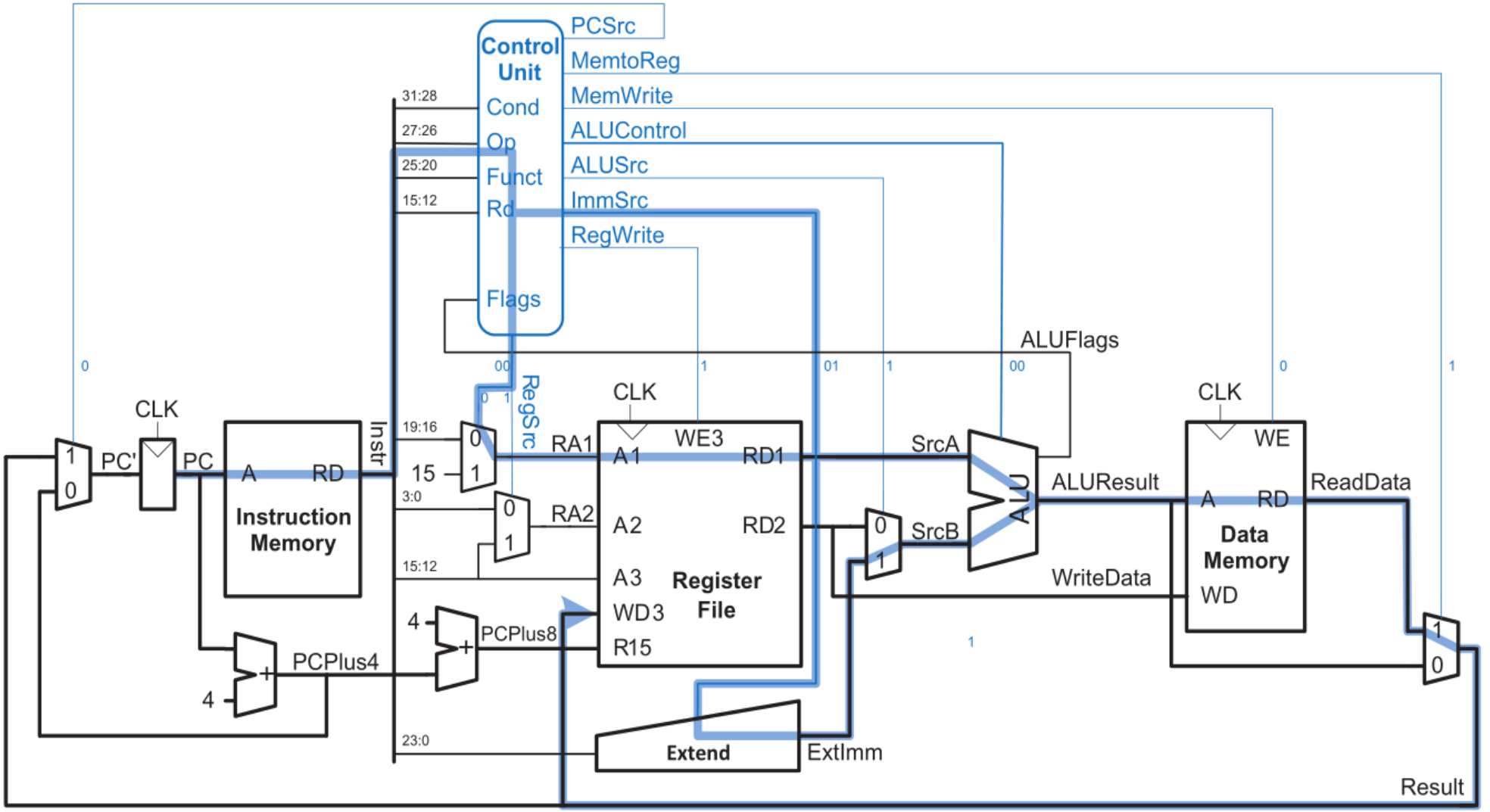
The data memory reads from this address.

The MemtoReg multiplexer selects ReadData.

Finally, Result must set up at the register file before the next rising clock edge so that it can be properly written.

The cycle time is:

$$T_{c1} = t_{pcqPC} + t_{mem} + t_{dec} + \max\left[t_{mux} + t_{RFread}, t_{ext} + t_{mux}\right] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$



In most implementation technologies, the ALU, memory, and register file are substantially slower than other combinational blocks.

$$T_{c1} = t_{pcqPC} + 2t_{mem} + t_{dec} + t_{ext} + 2t_{mux} + t_{ALU} + t_{RFsetup}$$

The numerical values of these times will depend on the specific implementation technology.

Other instructions have shorter critical paths.

The clock period is constant and must be long enough to accommodate the slowest instruction.

Exercise 14.1

Modify the single-cycle ARM processor to implement the following instructions:

a) `TST`

b) `LSL`

c) `CMN`

d) `ADC`

e) `EOR`

f) `LSR`

g) `TEQ`

h) `RSB`