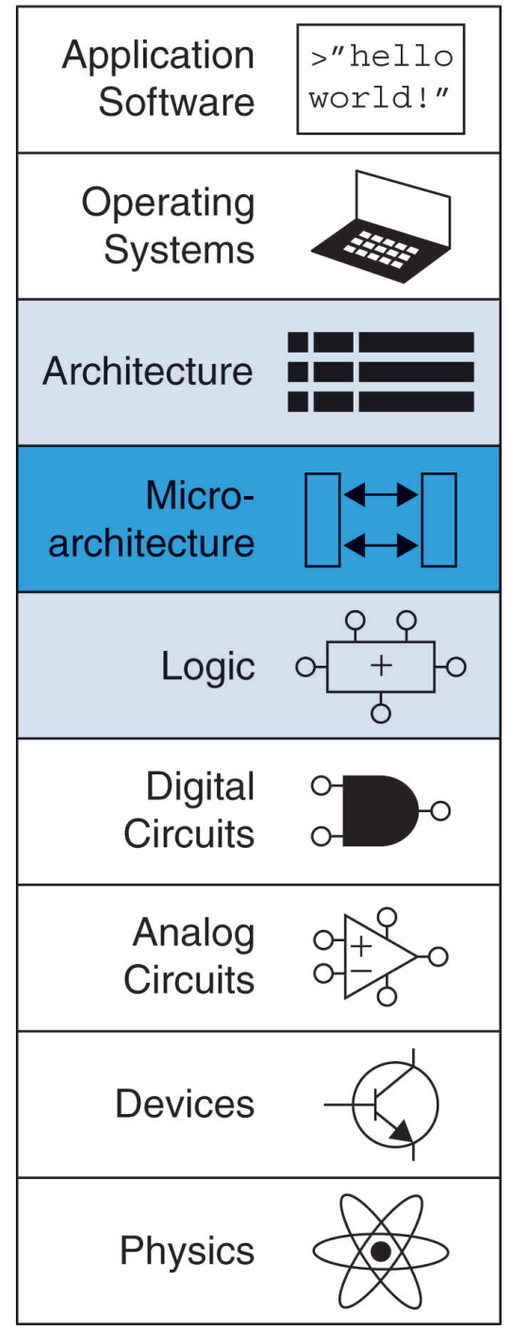
Microarchitecture

Digital Computer Design

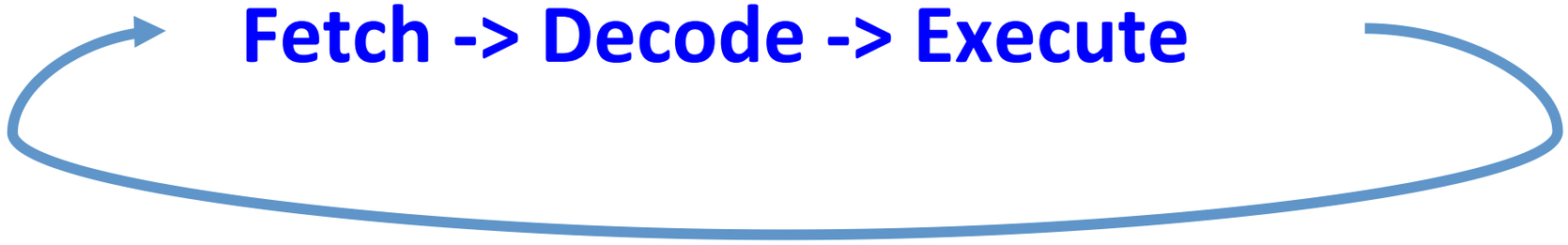
Introduction

- **Microarchitecture:** how to implement an architecture in hardware
- Processor:
  - **Datapath:** operates on words of data. It contains structures such as memories, registers, ALUs, and multiplexers.
  - **Control:** receives the current instruction from the datapath and tells the datapath how to execute that instruction.

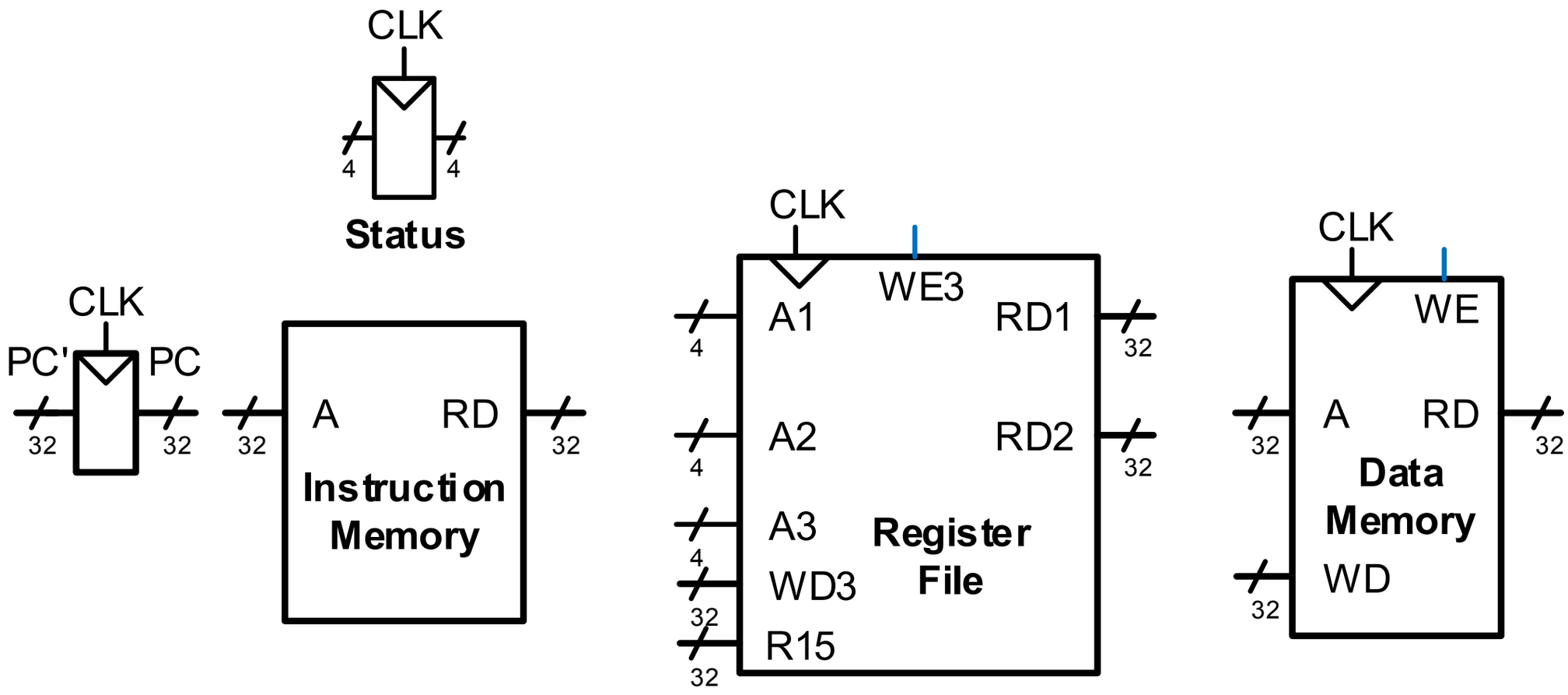


Single-Cycle Processor

- Executes instructions **in a single cycle**
  - **Datapath:** 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.



ARM Architectural State Elements



ARM Architectural State Elements

- **PC** points to the current instruction.
  - Its input, PC', indicates the address of the next instruction.
- The **instruction memory** has a single read port.
  - takes a 32-bit instruction address input, *A*
  - reads the 32-bit data (i.e., instruction) from that address onto the read data output, *RD*.
- The **data memory** has a single read/write port.
  - If its write enable, *WE*, is asserted, then it writes data *WD* into address *A* on the rising edge of the clock.
  - If its write enable is 0, then it reads address *A* onto *RD*.

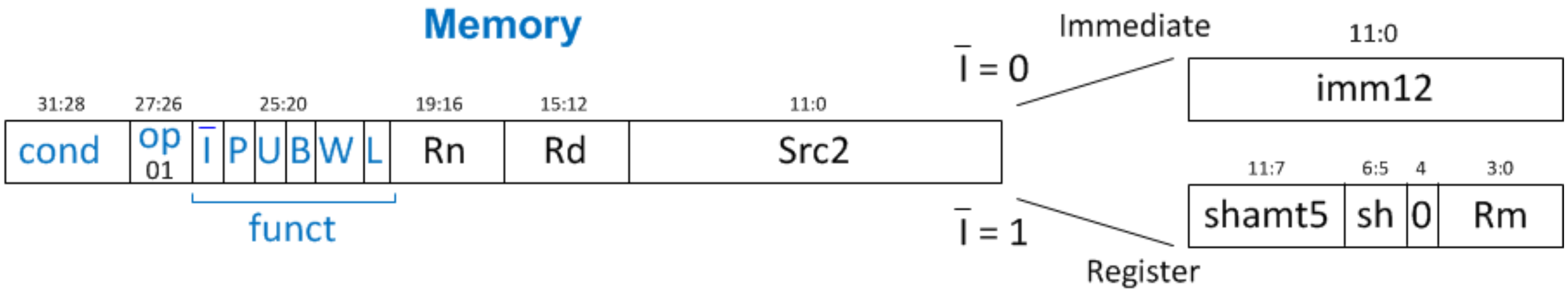
ARM Processor

- Consider **subset** of ARM instructions:
  - **Data-processing instructions:**
    - **`ADD, SUB, AND, ORR`**
    - with register and immediate Src2, but **no shifts**
  - **Memory instructions:**
    - **`LDR, STR`**
    - with **positive immediate offset**
  - **Branch instructions:**
    - **`B`**

Single-Cycle ARM Processor

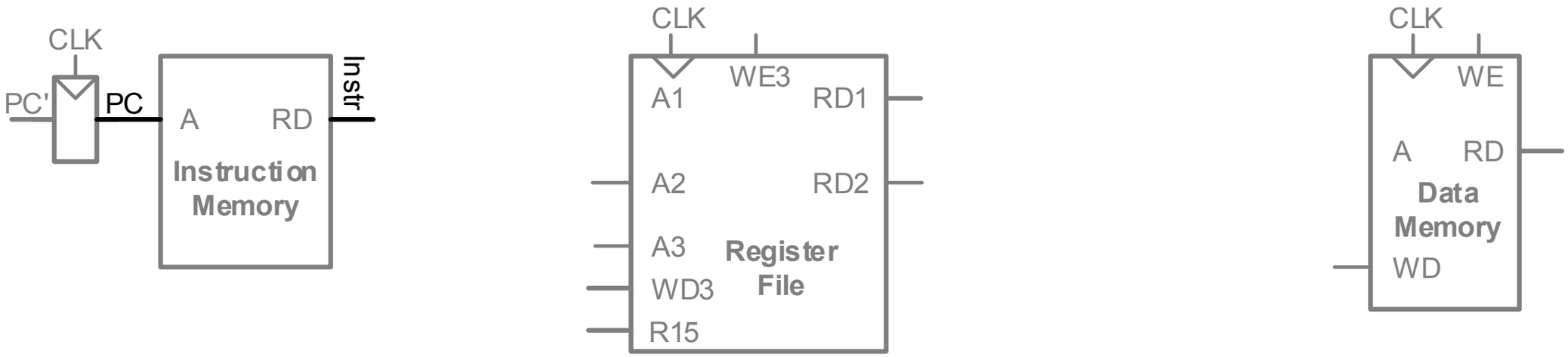
- **Example:** `LDR R1, [R2, #5]`

`LDR Rd, [Rn, imm12]`



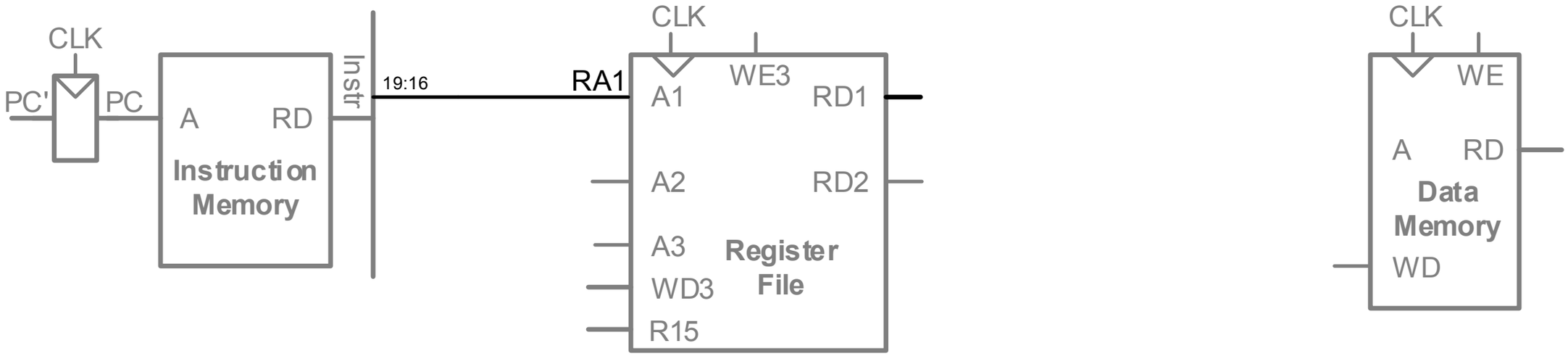
Single-Cycle Datapath: `LDR` fetch

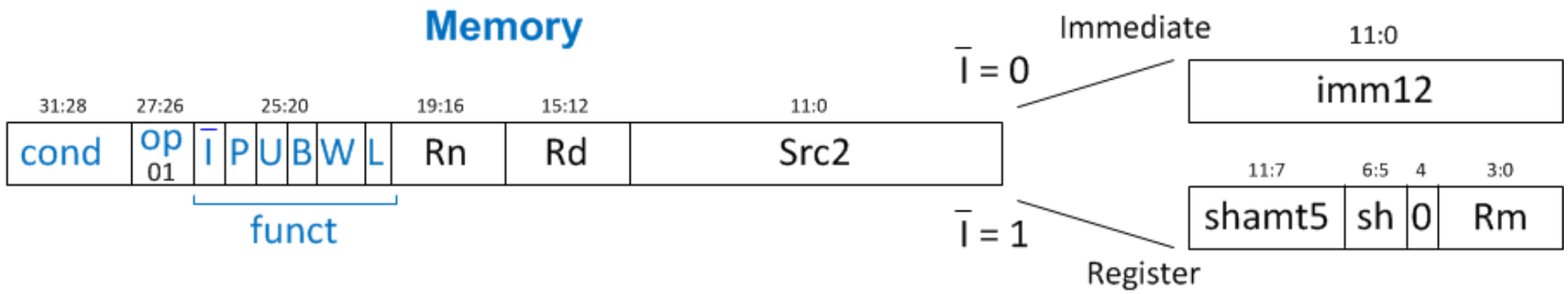
STEP 1: Fetch instruction



Single-Cycle Datapath: LDR Reg Read

STEP 2: Read source operands from Register File

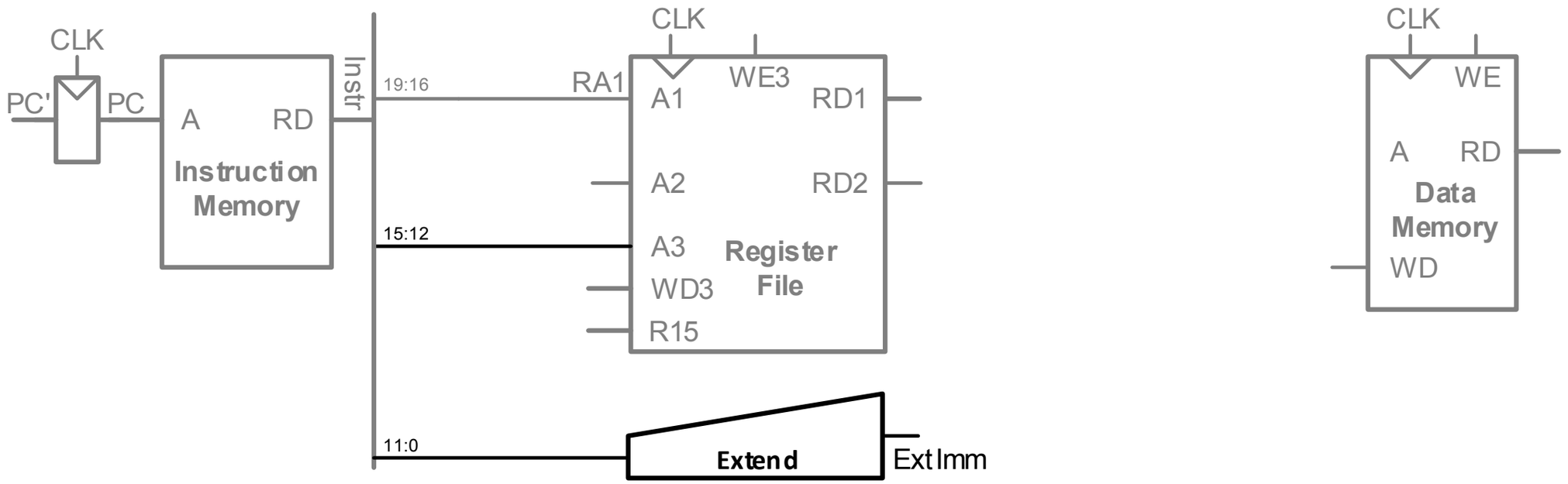


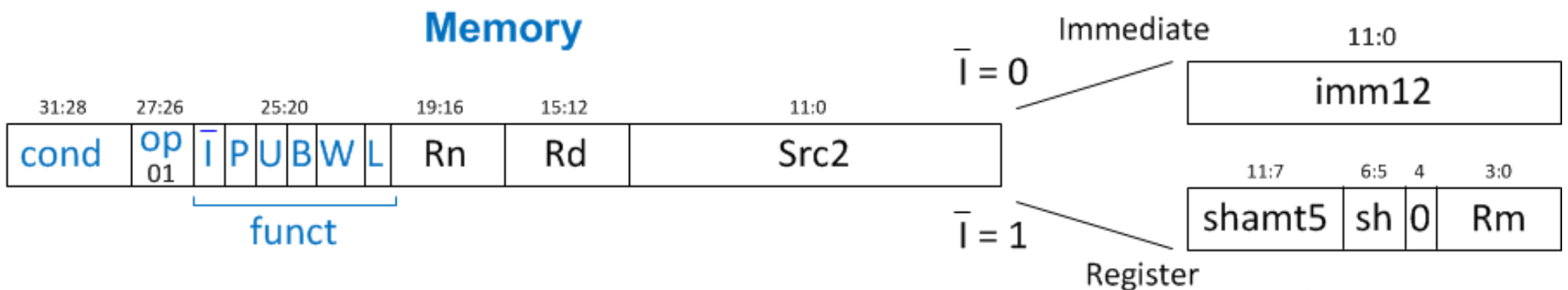


LDR Rd, [Rn, imm12]

Single-Cycle Datapath: `LDR` Immed.

STEP 3: Extend the immediate

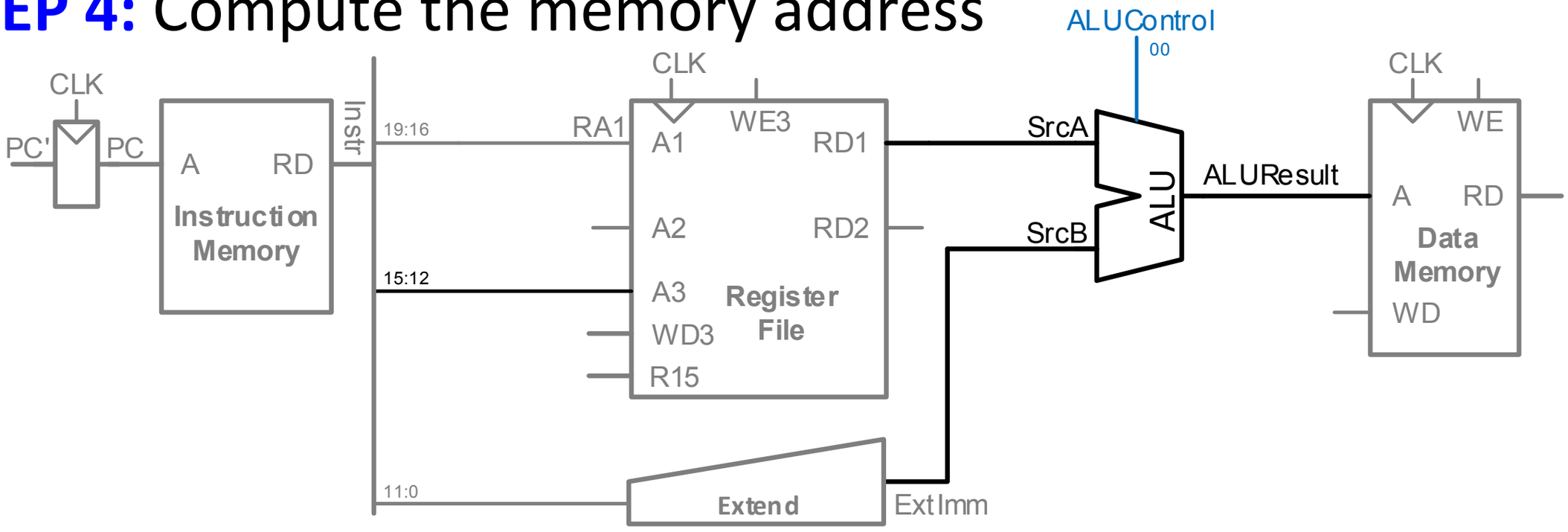


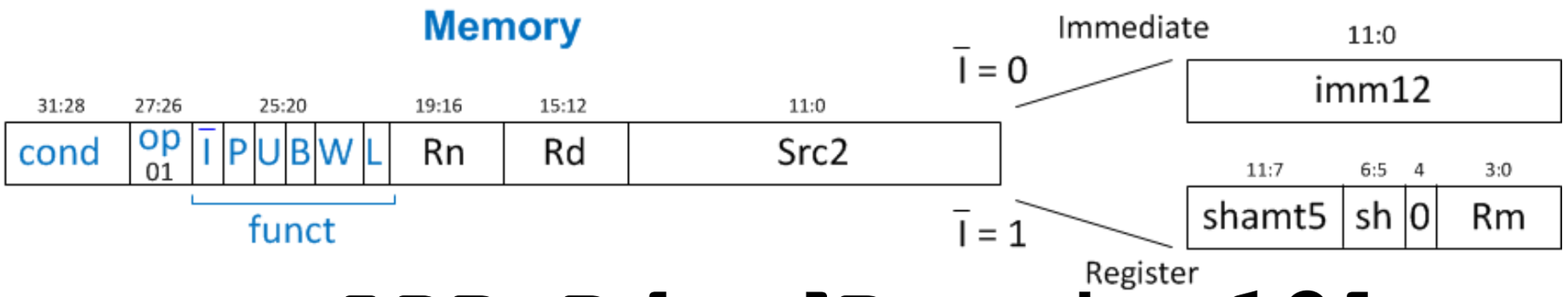


`LDR Rd, [Rn, imm12]`

Single-Cycle Datapath: `LDR` Address

STEP 4: Compute the memory address

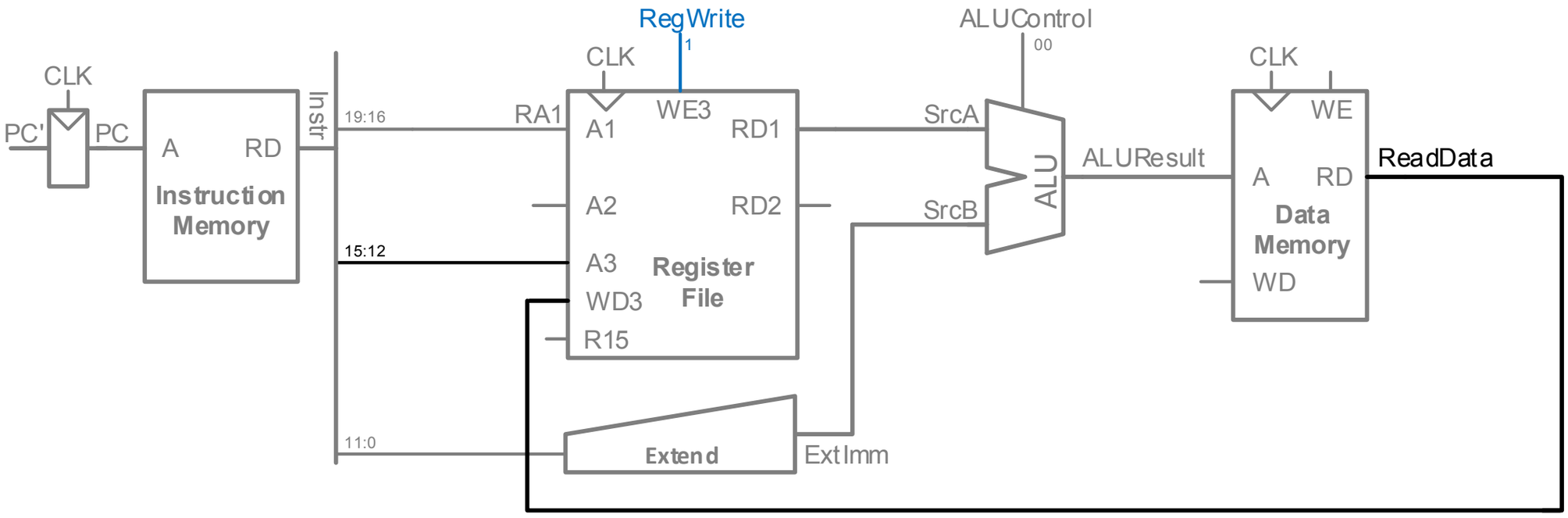


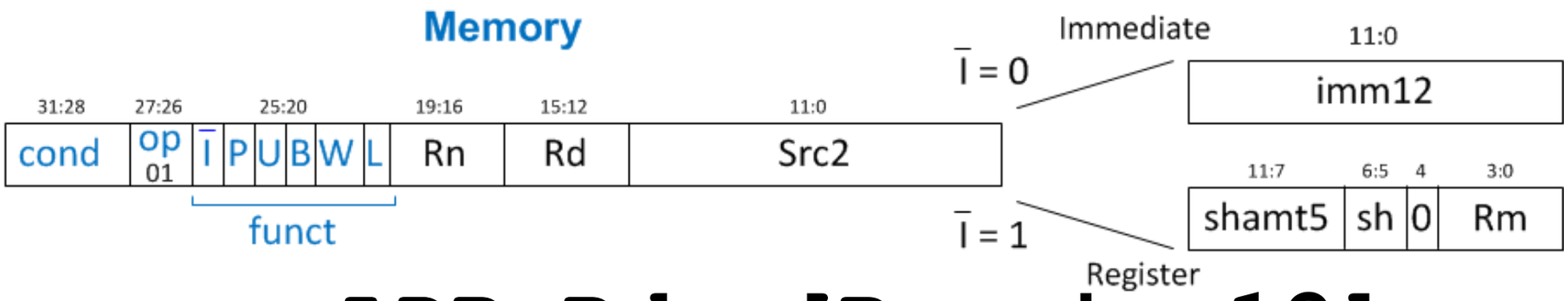


`LDR Rd, [Rn, imm12]`

Single-Cycle Datapath: LDR Mem Read

STEP 5: Read data from memory and write it back to register file

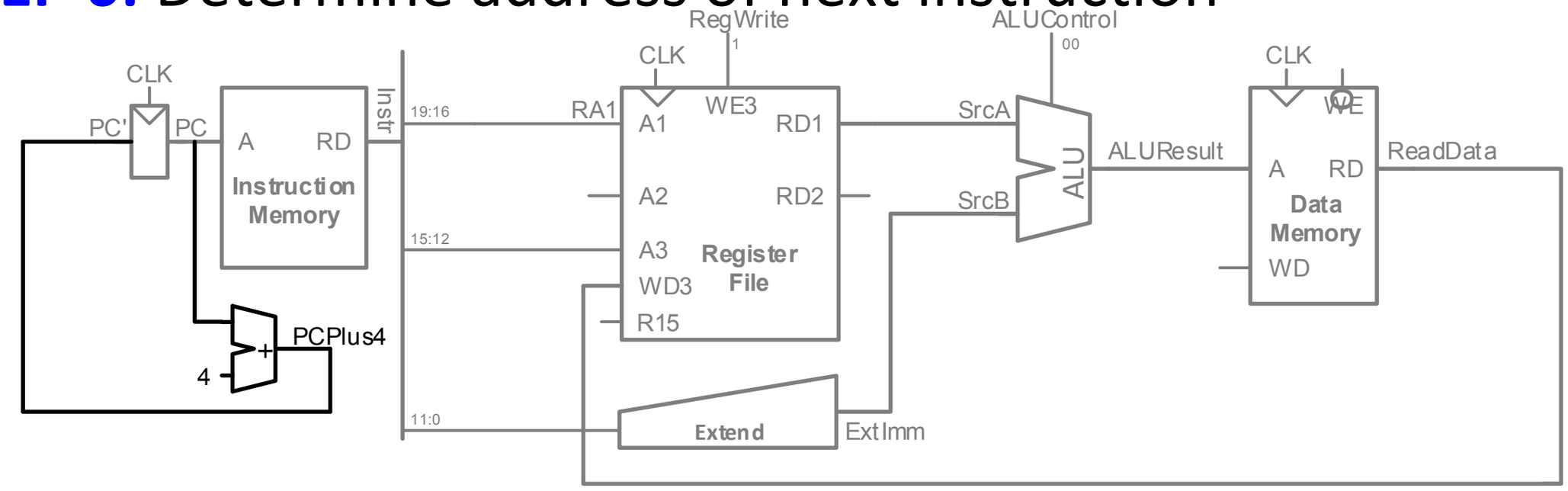




LDR Rd, [Rn, imm12]

Single-Cycle Datapath: PC Increment

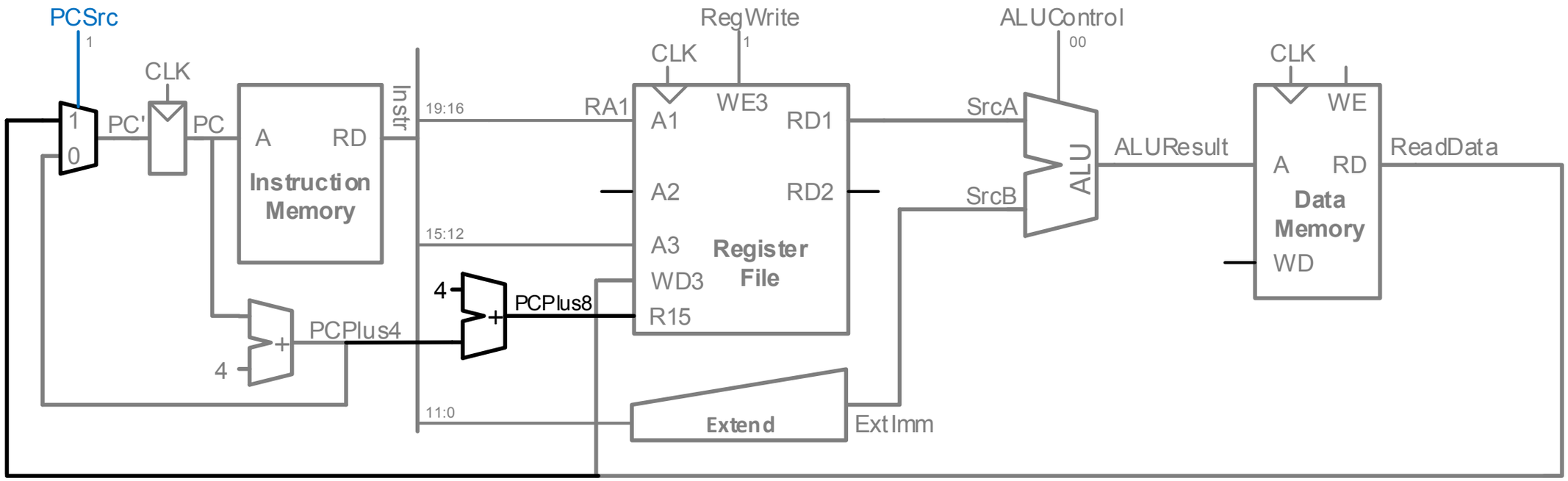
STEP 6: Determine address of next instruction



Single-Cycle Datapath: Access to PC

PC can be source/destination of instruction

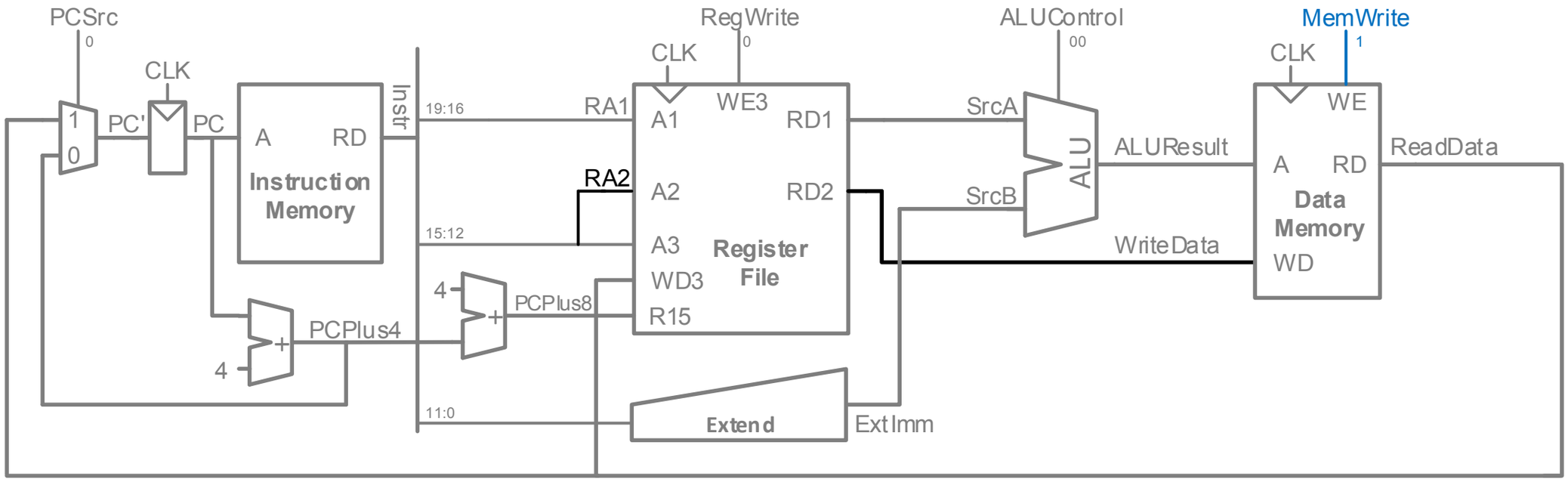
- **Source:** R15 must be available in Register File
  - in the ARM architecture, reading register R15 returns PC + 8.
- **Destination:** Be able to write result to PC

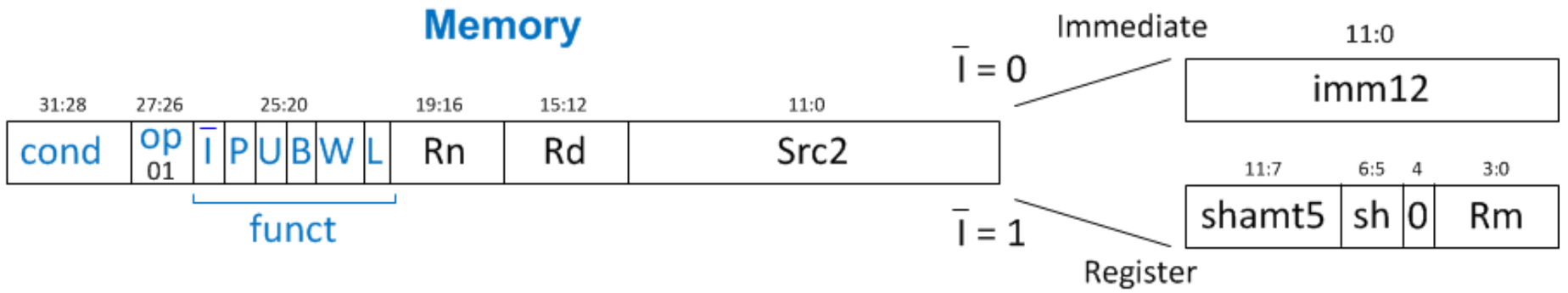


Single-Cycle Datapath: STR

Expand datapath to handle STR:

- Write data in Rd to memory



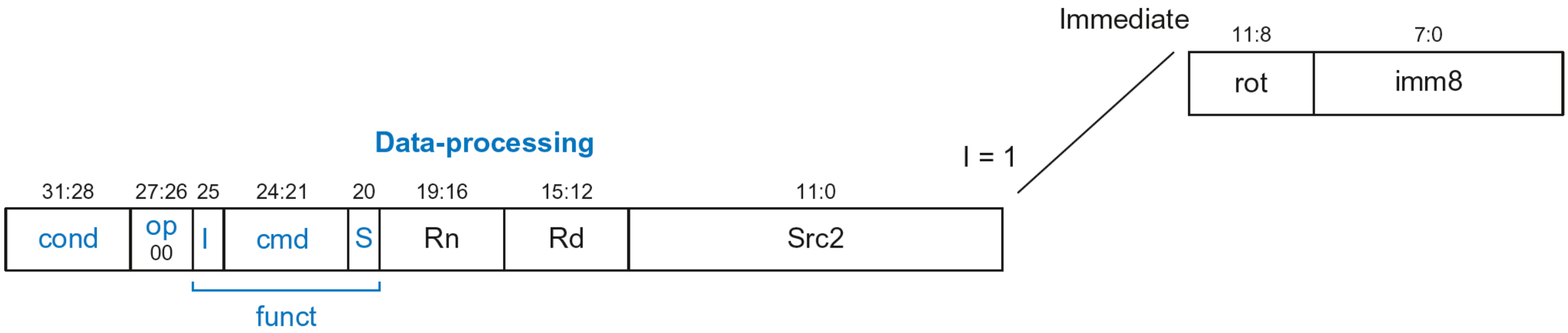


STR Rd, [Rn, imm12]

Single-Cycle Datapath: Data-processing

With immediate Src2:

- Read from `Rn` and `Imm8` (*ImmSrc* chooses the zero-extended `Imm8` instead of `Imm12`)
- Write *ALUResult* to register file
- Write to `Rd`

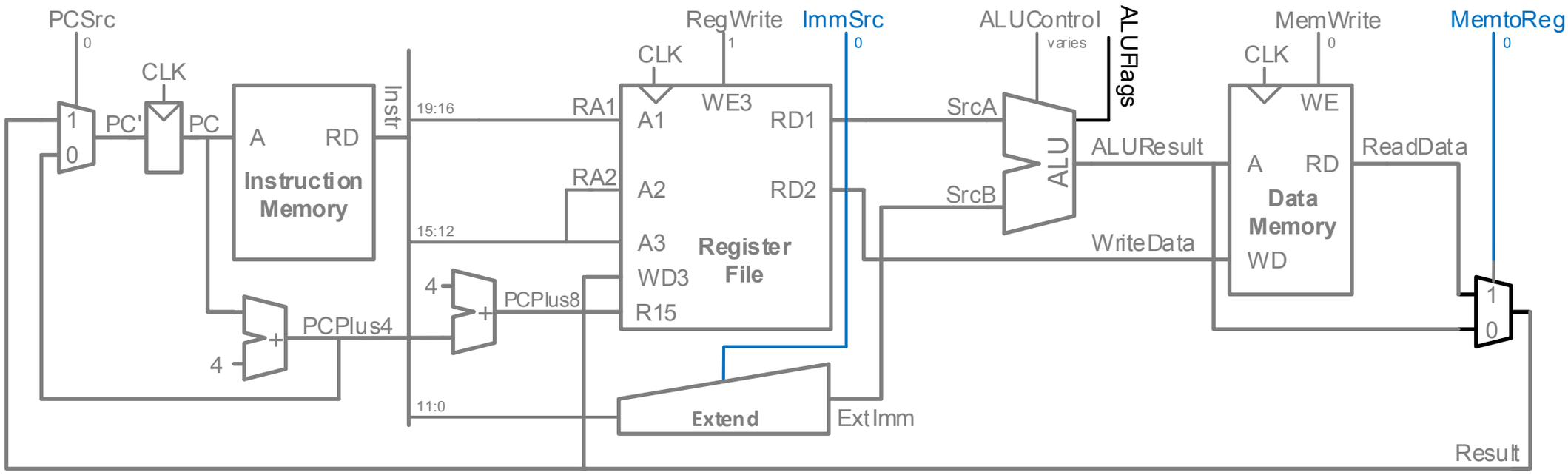


`ADD Rd, Rn, imm8`

Single-Cycle Datapath: Data-processing

With immediate Src2:

- Read from `Rn` and `Imm8` (*ImmSrc* chooses the zero-extended `Imm8` instead of `Imm12`)
- Write *ALUResult* to register file
- Write to `Rd`

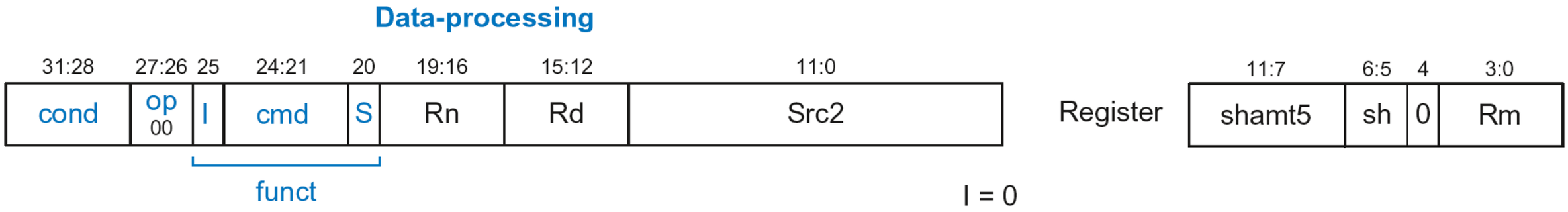


`ADD Rd, Rn, imm8`

Single-Cycle Datapath: Data-processing

With register Src2:

- Read from `Rn` and `Rm` (instead of `Imm8`)
- Write *ALUResult* to register file
- Write to `Rd`

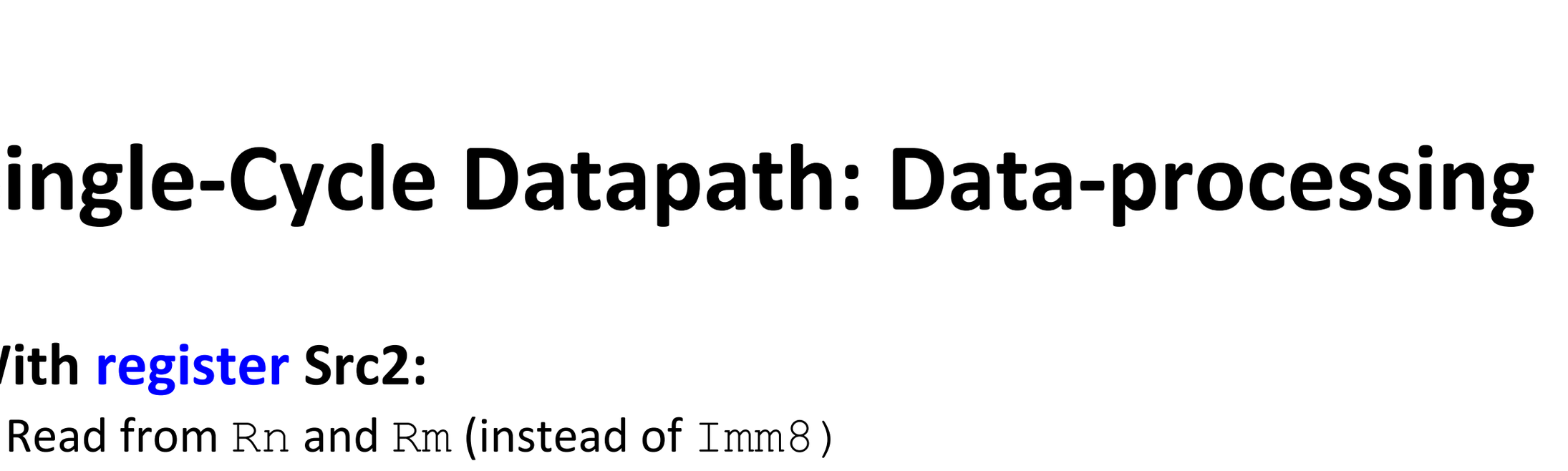


`ADD Rd, Rn, Rm`

Single-Cycle Datapath: Data-processing

With register Src2:

- Read from `Rn` and `Rm` (instead of `Imm8`)
- Write *ALUResult* to register file
- Write to `Rd`



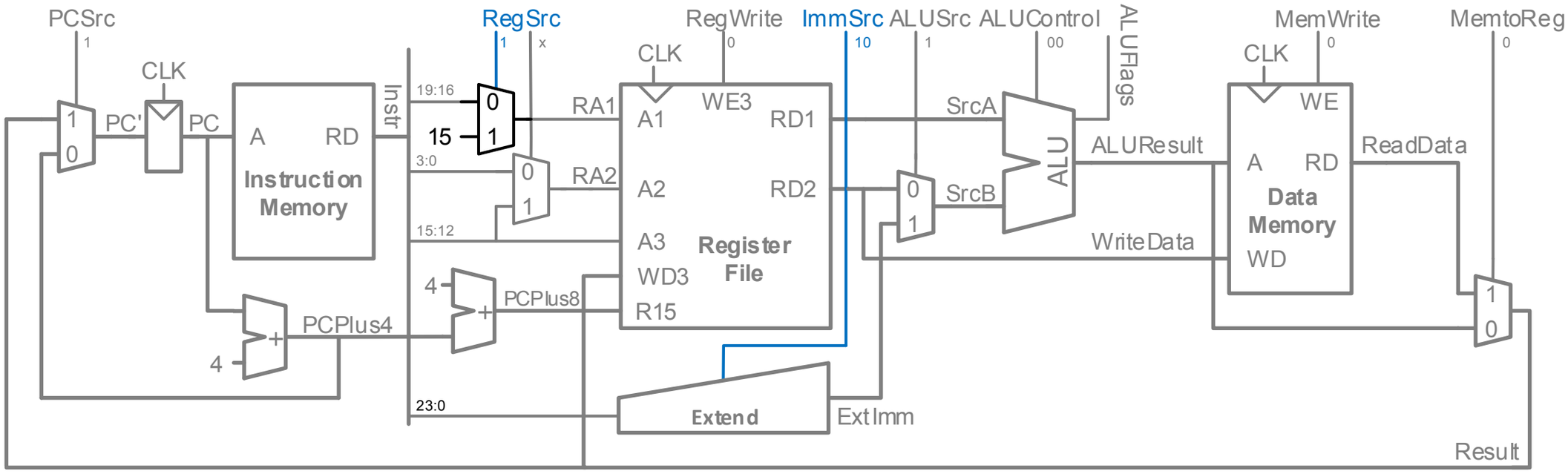
ADD Rd, Rn, Rm

Single-Cycle Datapath: B

Calculate branch target address:

BTA = (*ExtImm*) + (PC + 8)

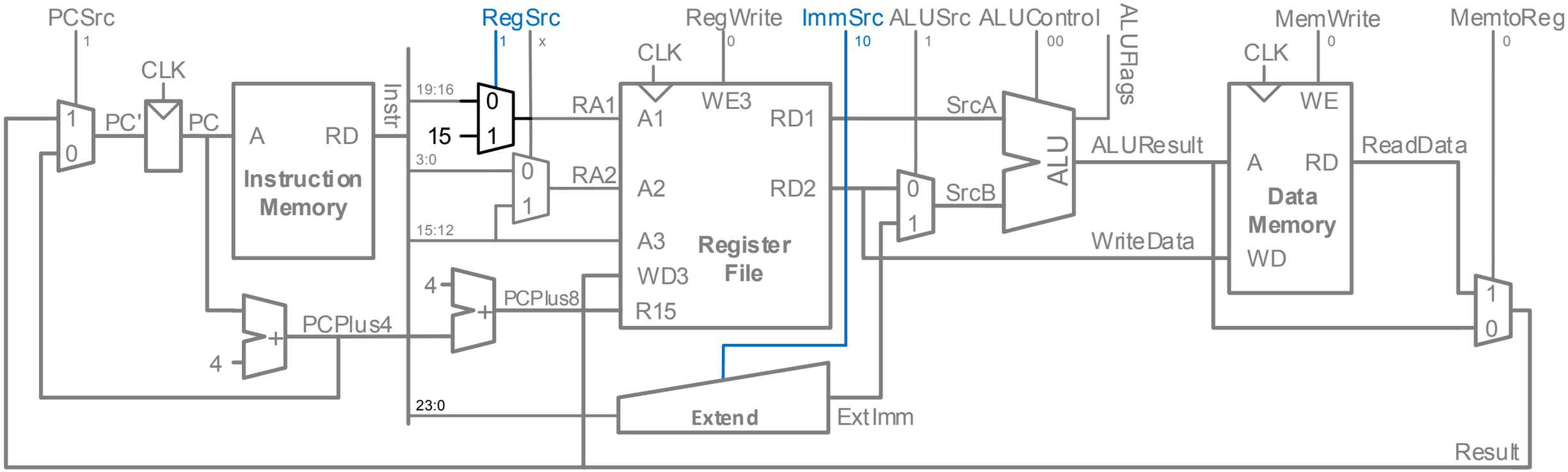
ExtImm = Imm24 << *2* and sign-extended



Branch

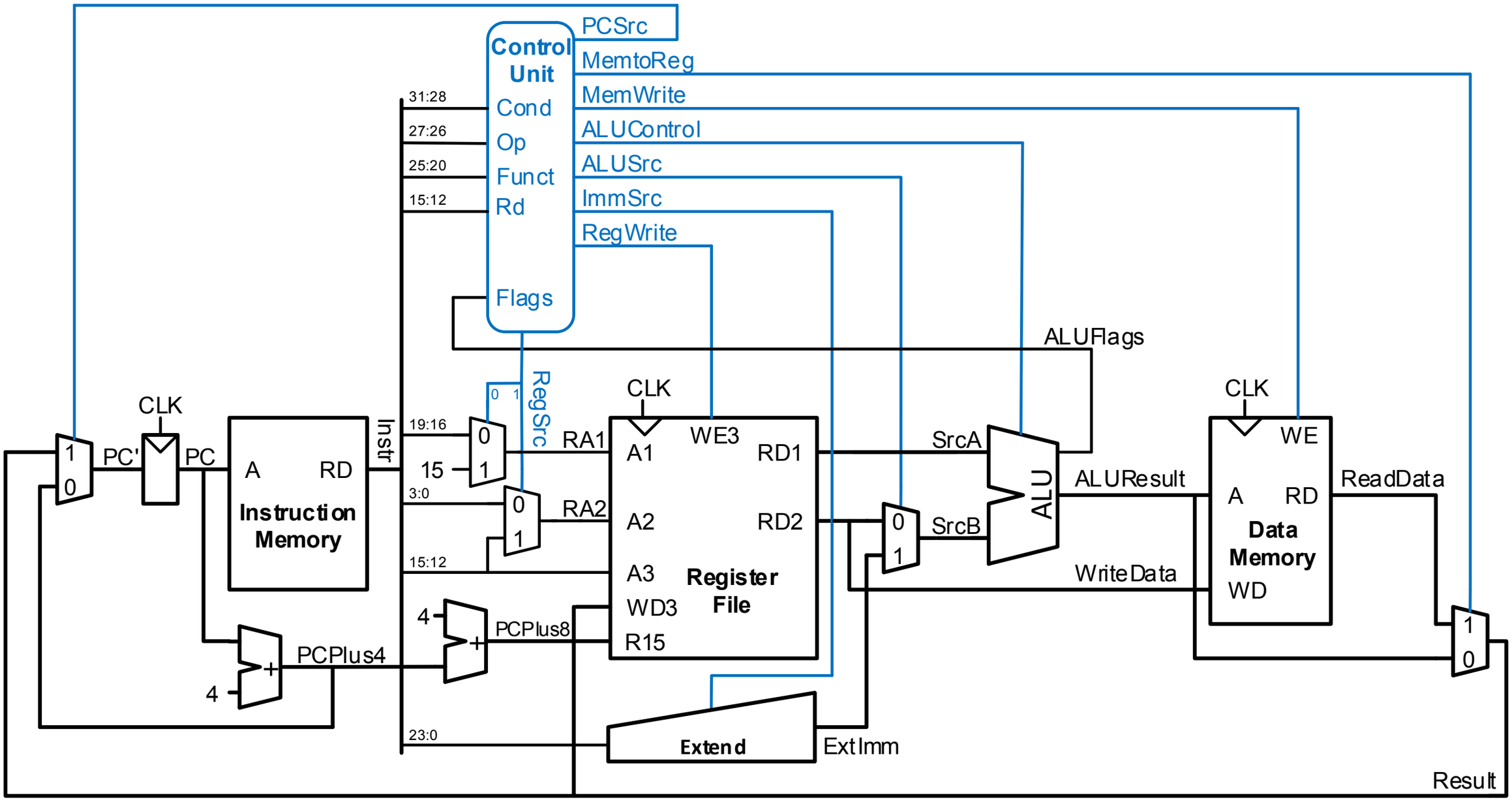
`B Label`

Single-Cycle Datapath: ExtImm

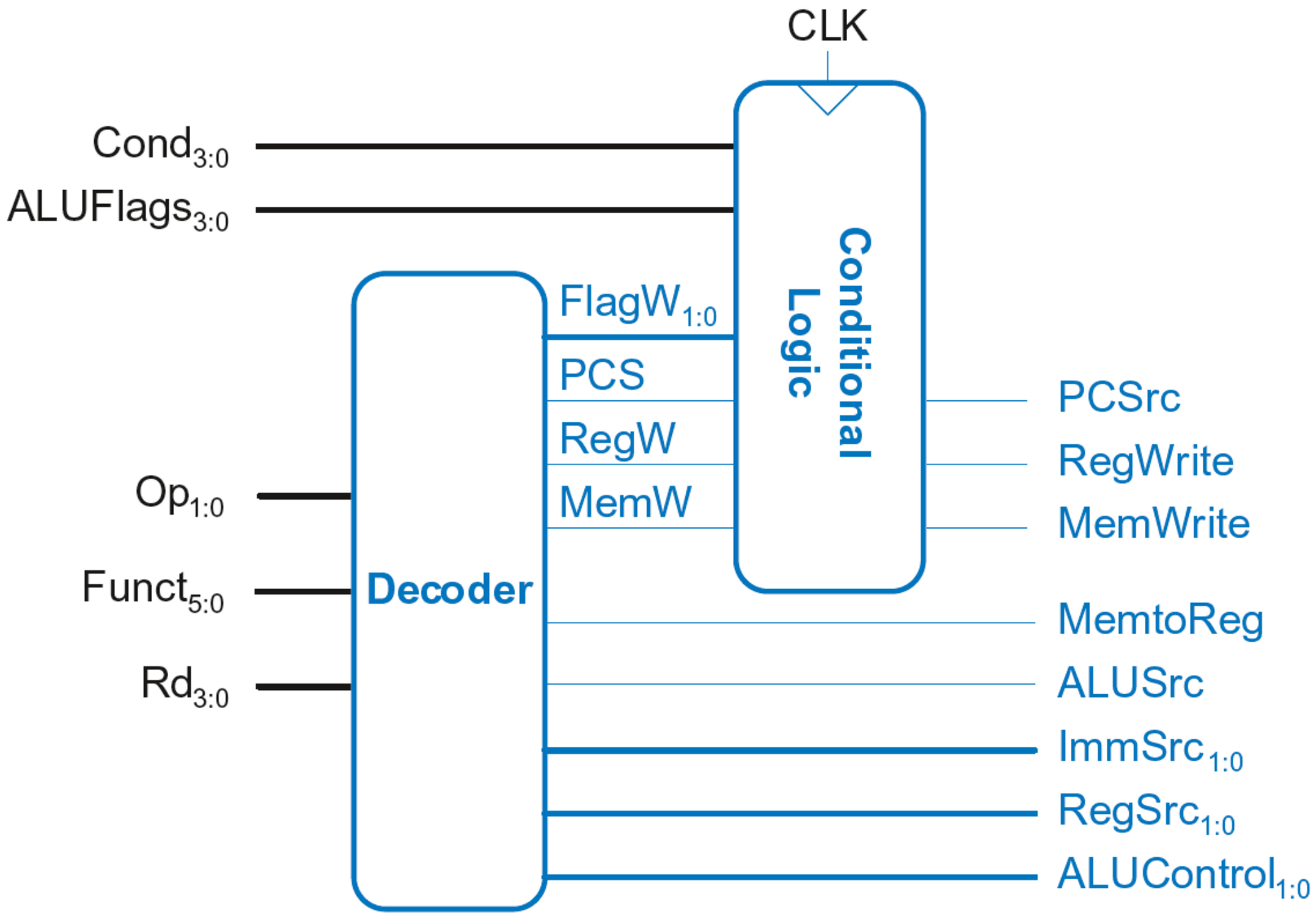


$ImmSrc_{1:0}$	ExtImm	Description
00	{24'b0, $Instr_{7:0}$}	Zero-extended *imm8*
01	{20'b0, $Instr_{11:0}$}	Zero-extended *imm12*
10	{6{$Instr_{23}$}, $Instr_{23:0}$}	Sign-extended *imm24*

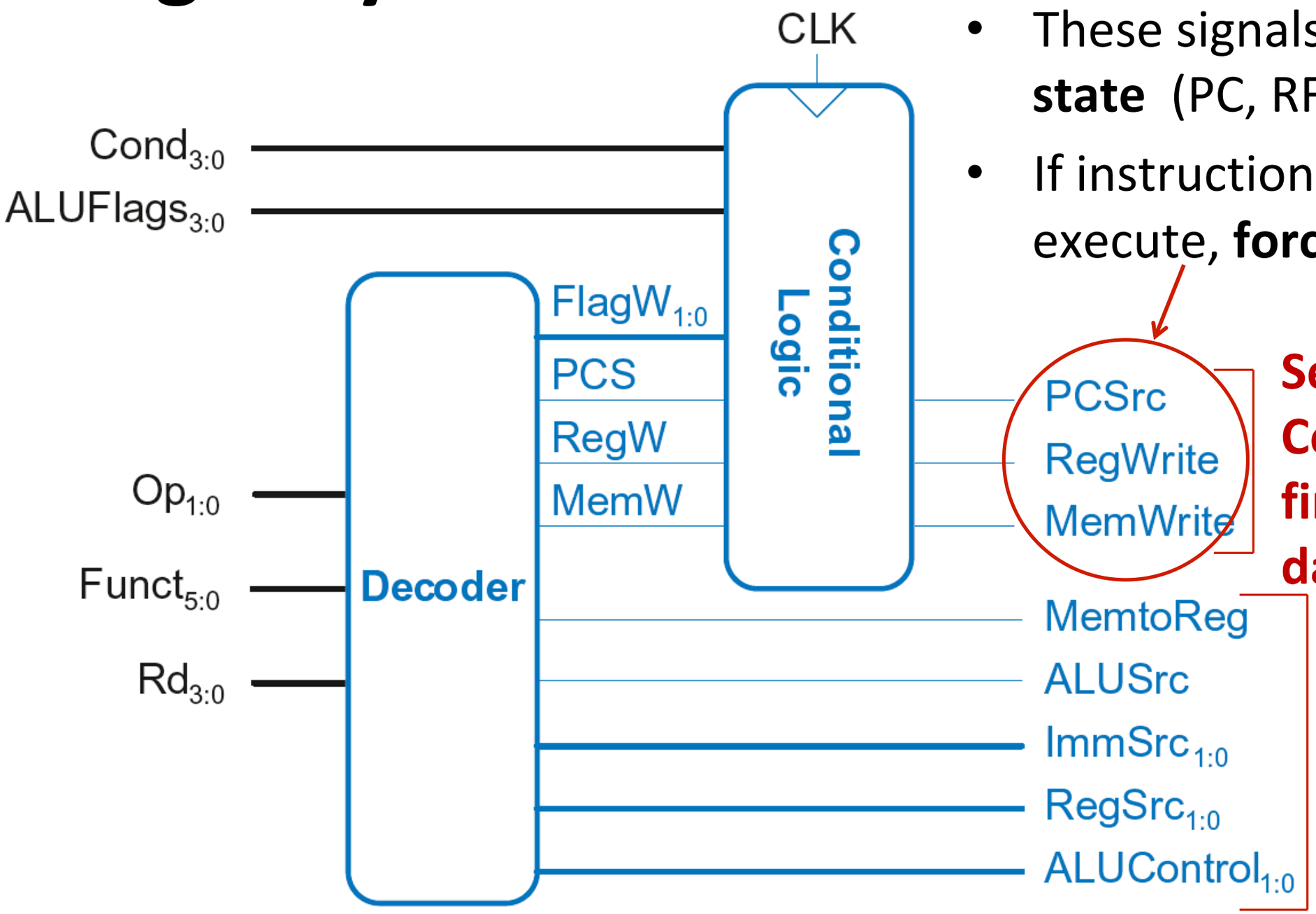
Single-Cycle ARM Processor



Single-Cycle Control



Single-Cycle Control

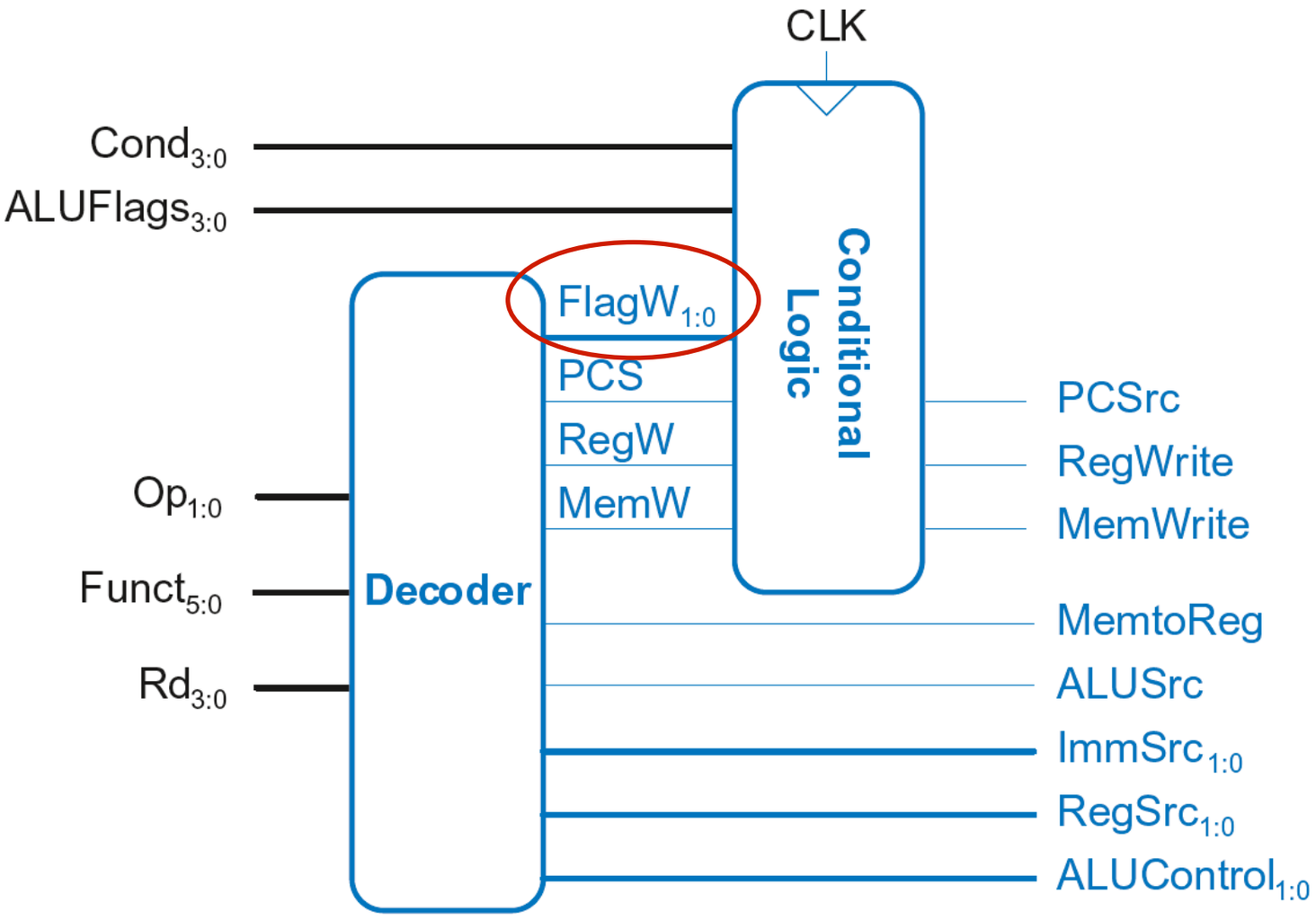


- These signals **change the state** (PC, RF, Memory)
- If instruction shouldn't execute, **forced to 0**

Sent through Conditional Logic first, then to datapath

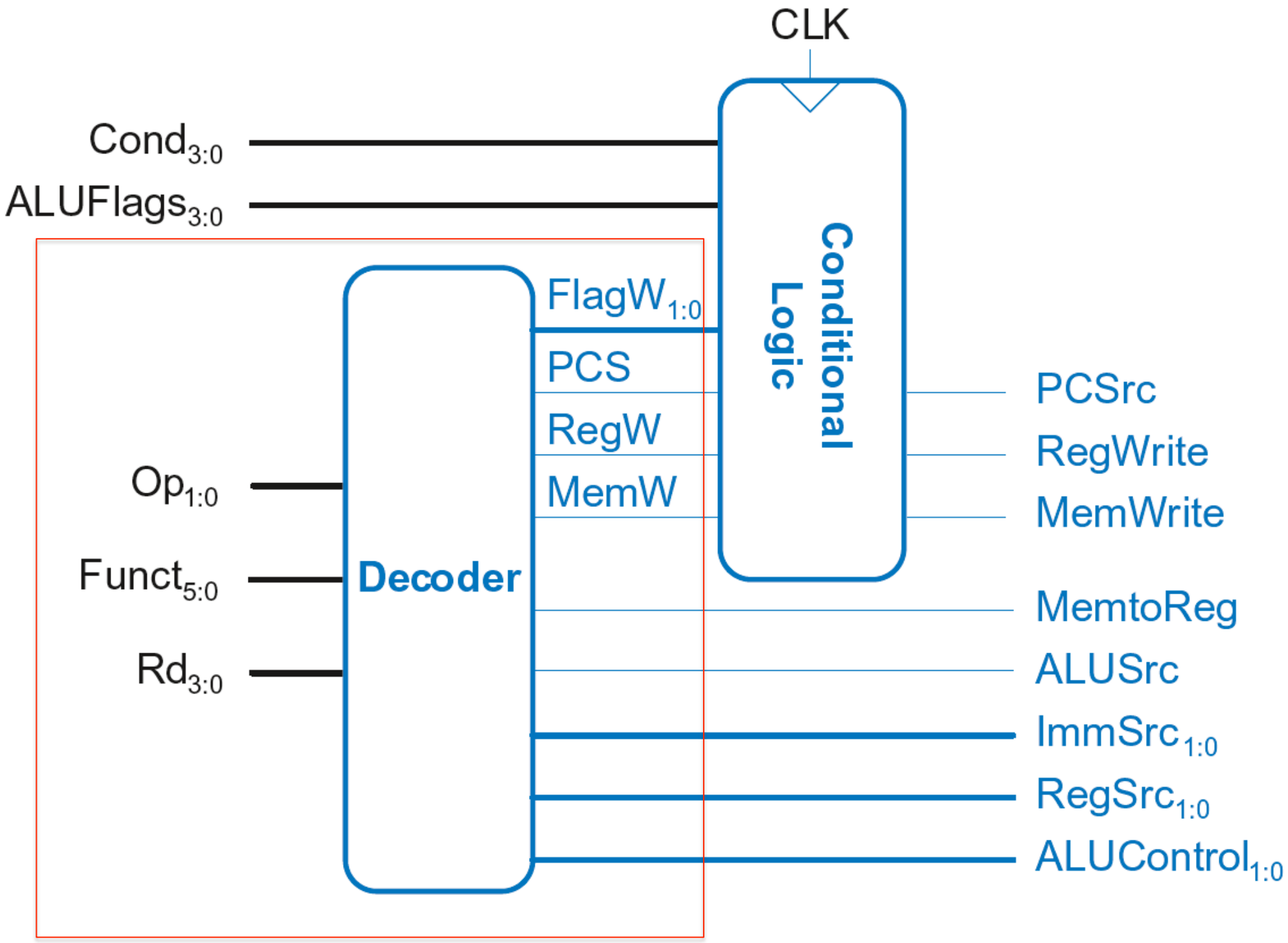
Sent directly to datapath

Single-Cycle Control



- ***FlagW*$_{1:0}$:** Flag Write signal, asserted when *ALUFlags* should be saved (i.e., on instruction with S=1)
- ADD, SUB update all flags (**NZCV**)
- AND, ORR only update **NZ** flags
- So, two bits needed:
  - ***FlagW*$_1$** = 1: *NZ* saved (*ALUFlags*$_{3:2}$ saved)
  - ***FlagW*$_0$** = 1: *CV* saved (*ALUFlags*$_{1:0}$ saved)

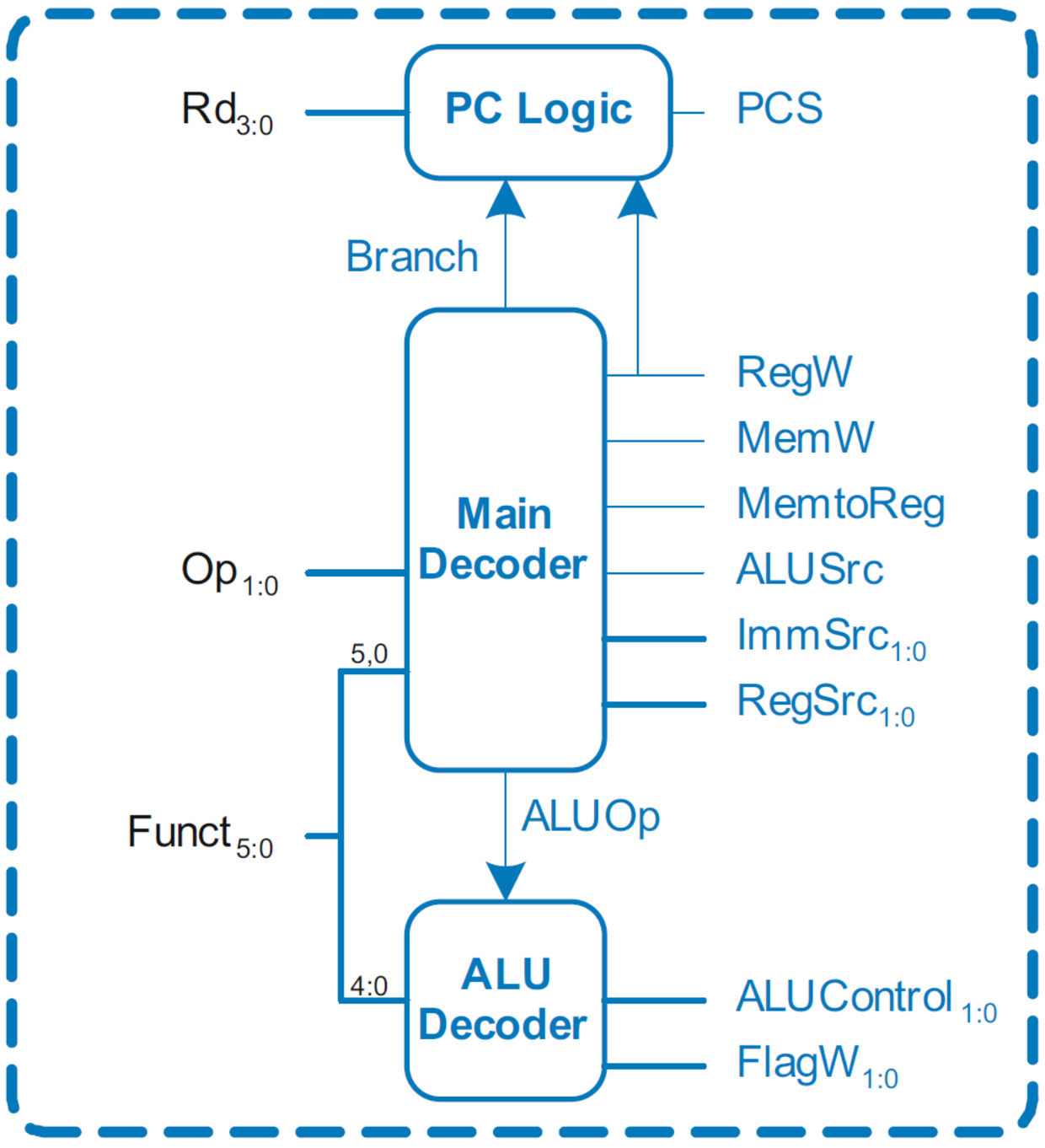
Single-Cycle Control: Decoder



Single-Cycle Control: Decoder

Submodules:

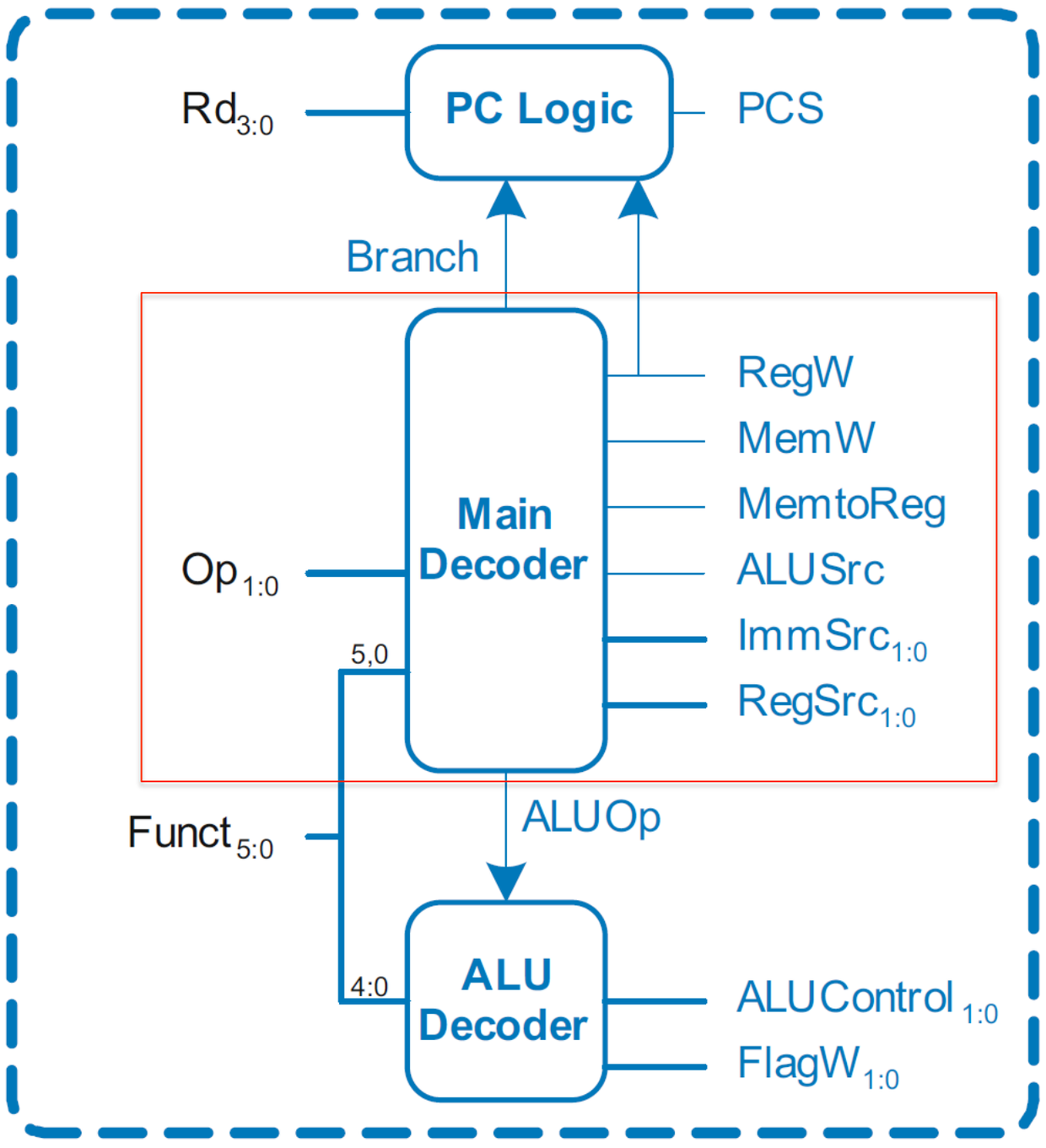
- Main Decoder
- ALU Decoder
- PC Logic



Single-Cycle Control: Decoder

Submodules:

- Main Decoder
- ALU Decoder
- PC Logic



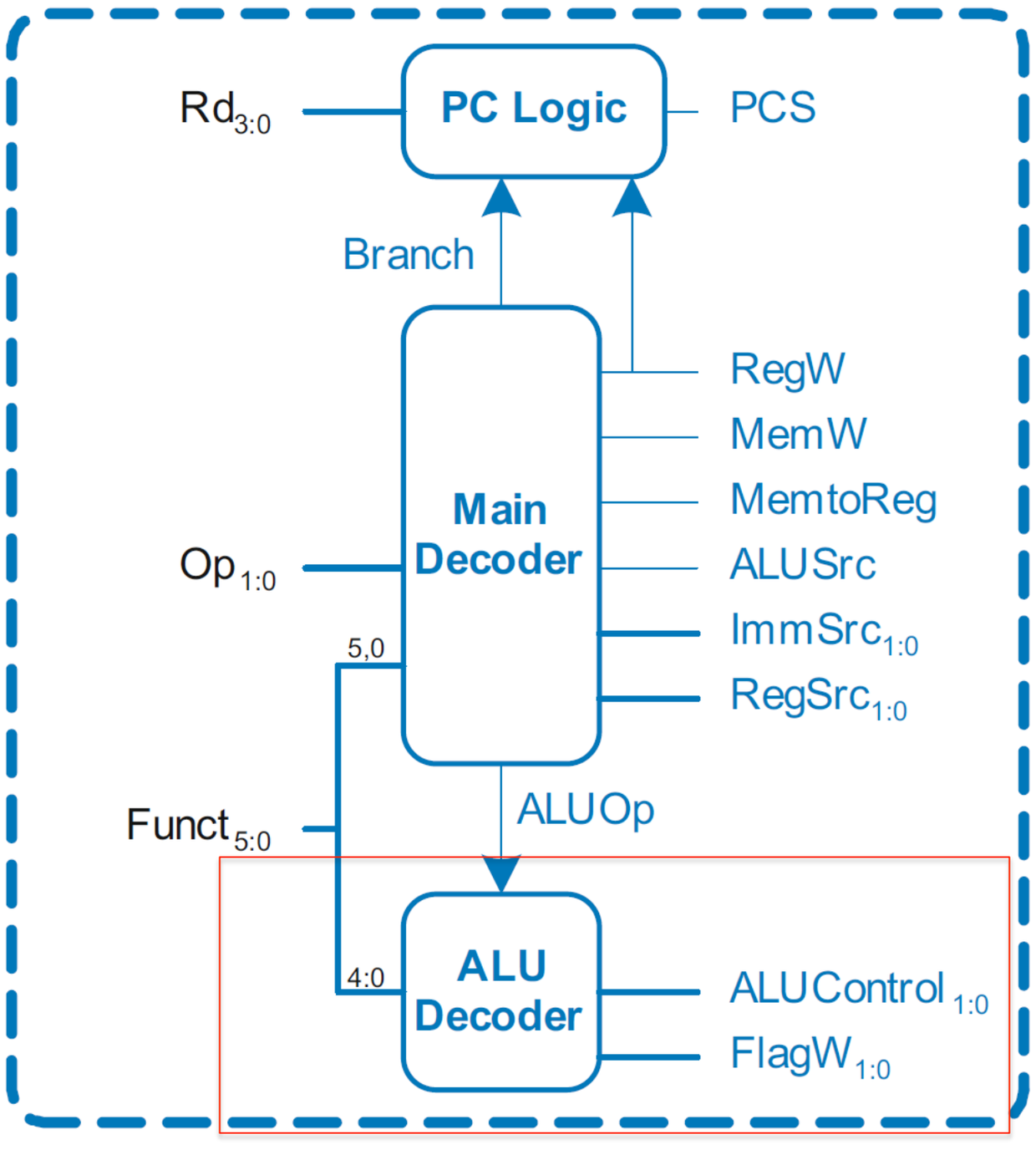
Control Unit: Main Decoder

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
11	X	X	B	1	0	0	1	10	0	X1	0

Single-Cycle Control: Decoder

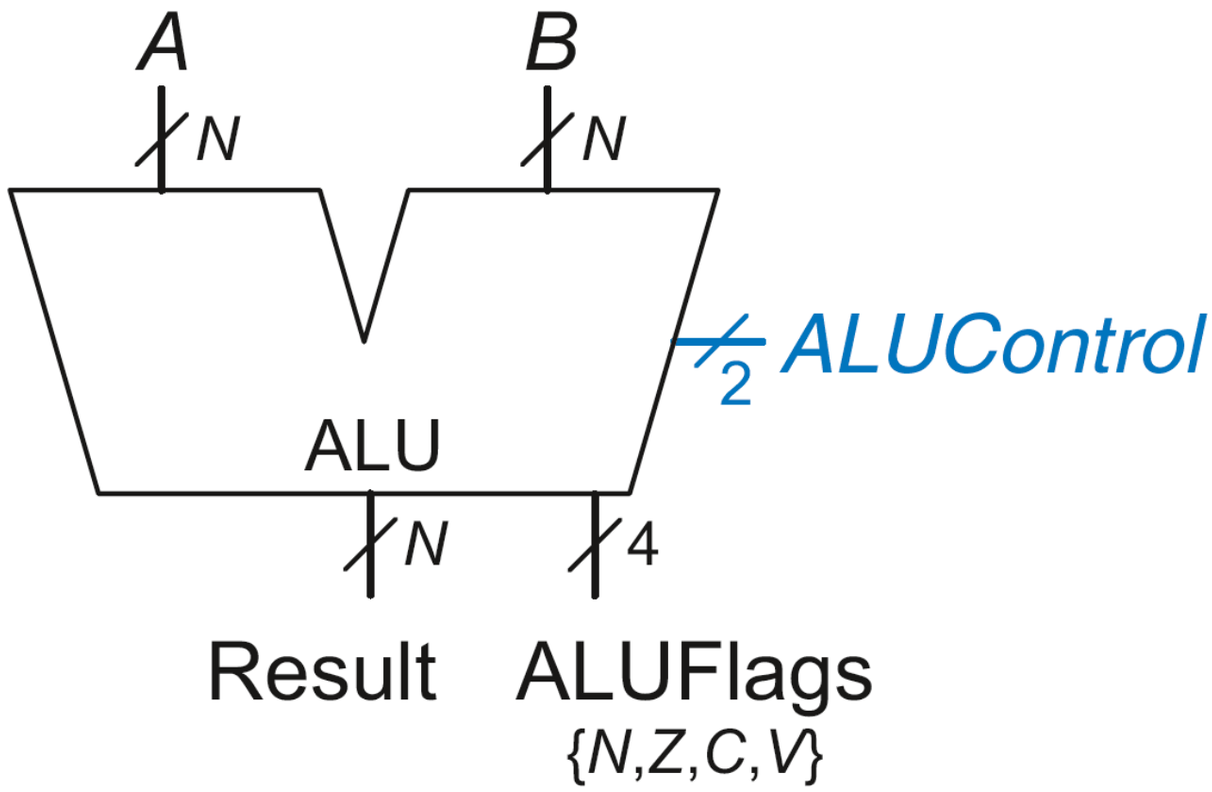
Submodules:

- Main Decoder
- ALU Decoder
- PC Logic

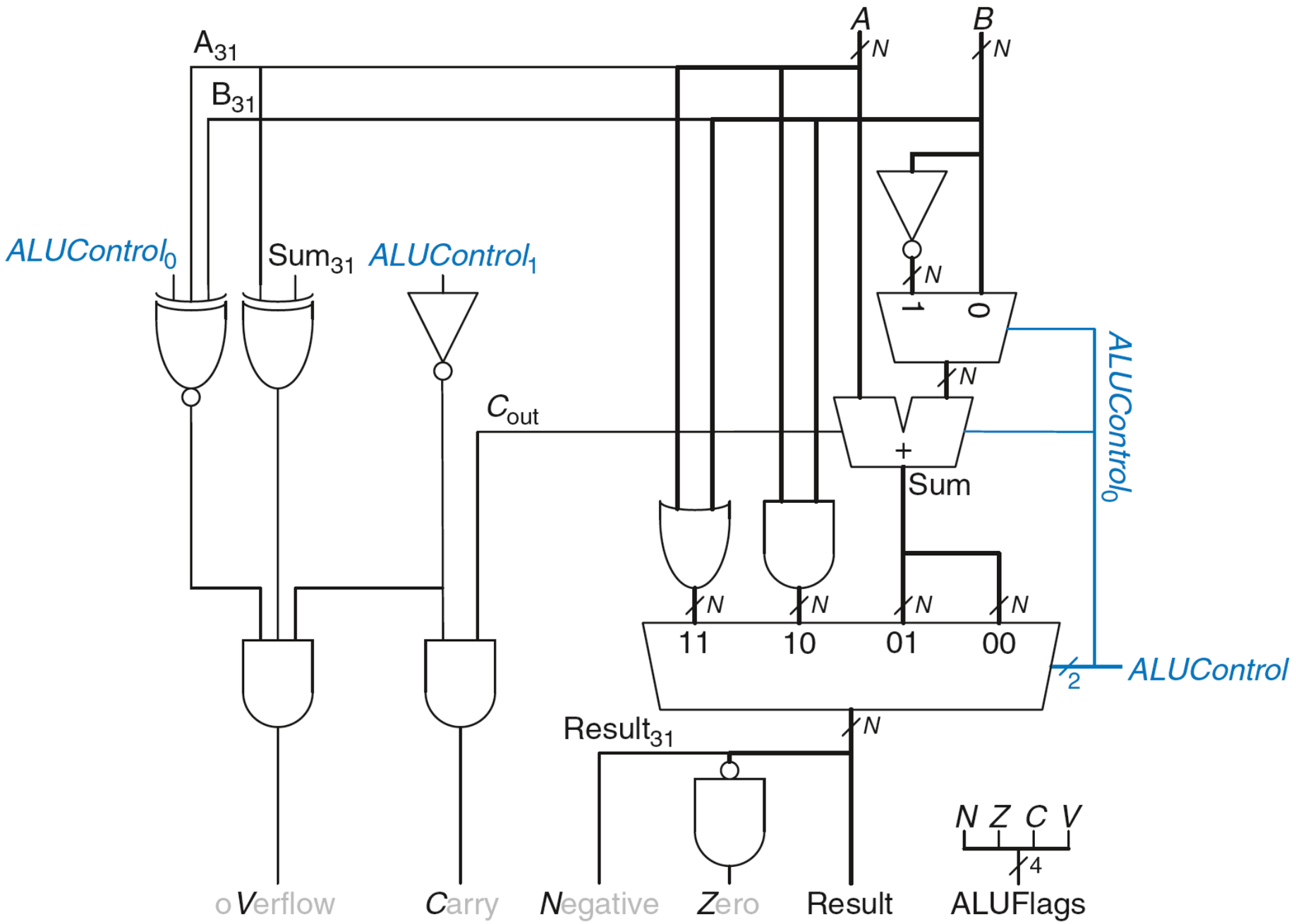


Review: ALU

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR



Review: ALU


A
B
N
N
A_{31}
B_{31}
$ALUControl_0$
Sum_{31}
$ALUControl_1$
C_{out}
N
1
0
N
+
Sum
$ALUControl_0$
N
N
N
N
11
10
01
00
2
ALUControl
$Result_{31}$
N
N Z C V
4
oVerflow
Carry
Negative
Zero
Result
ALUFlags

Control Unit: ALU Decoder

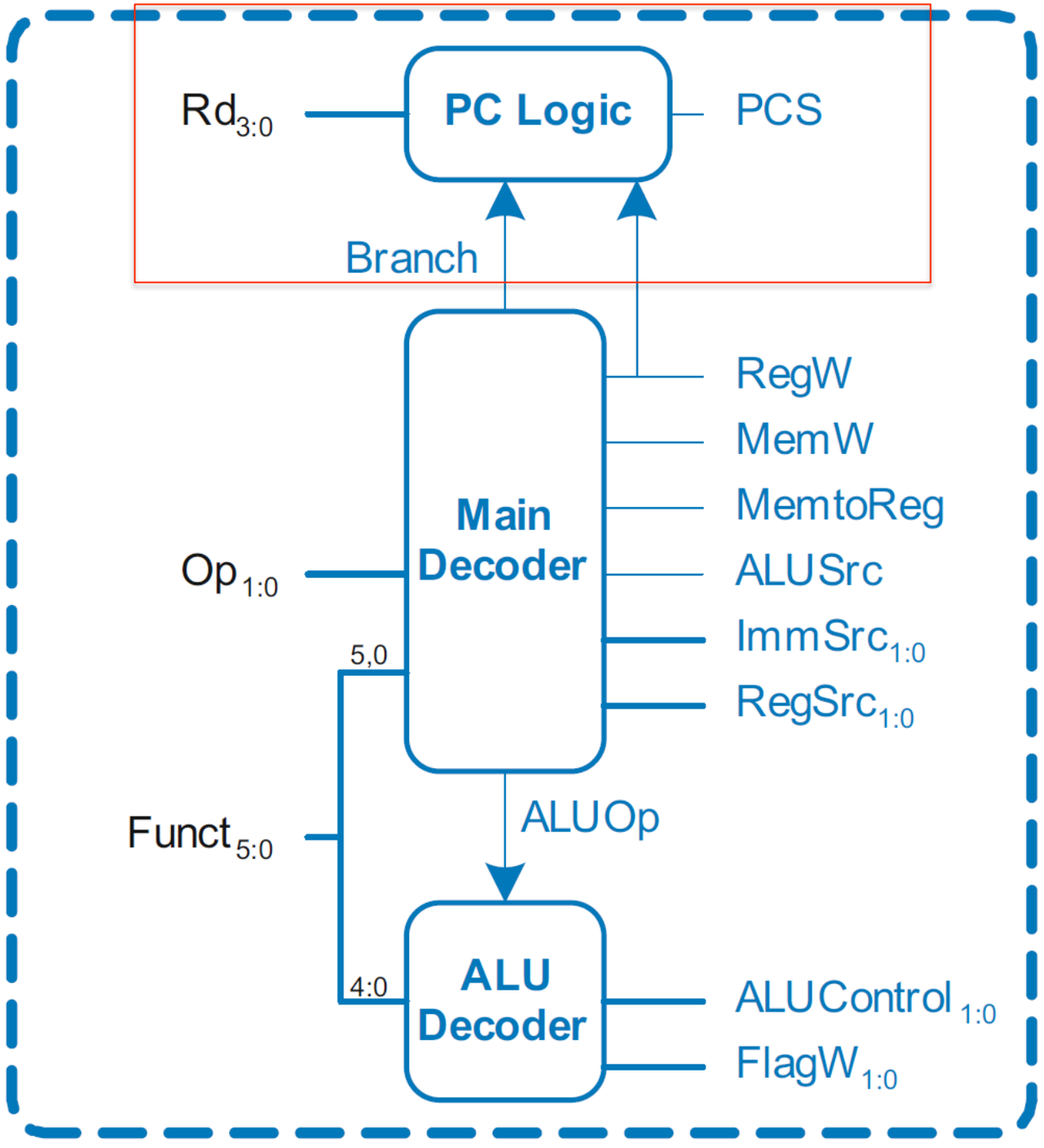
ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00	00
1	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10

- $FlagW_1$ = 1: *NZ* ($Flags_{3:2}$) should be saved
- $FlagW_0$ = 1: *CV* ($Flags_{1:0}$) should be saved

Single-Cycle Control: Decoder

Submodules:

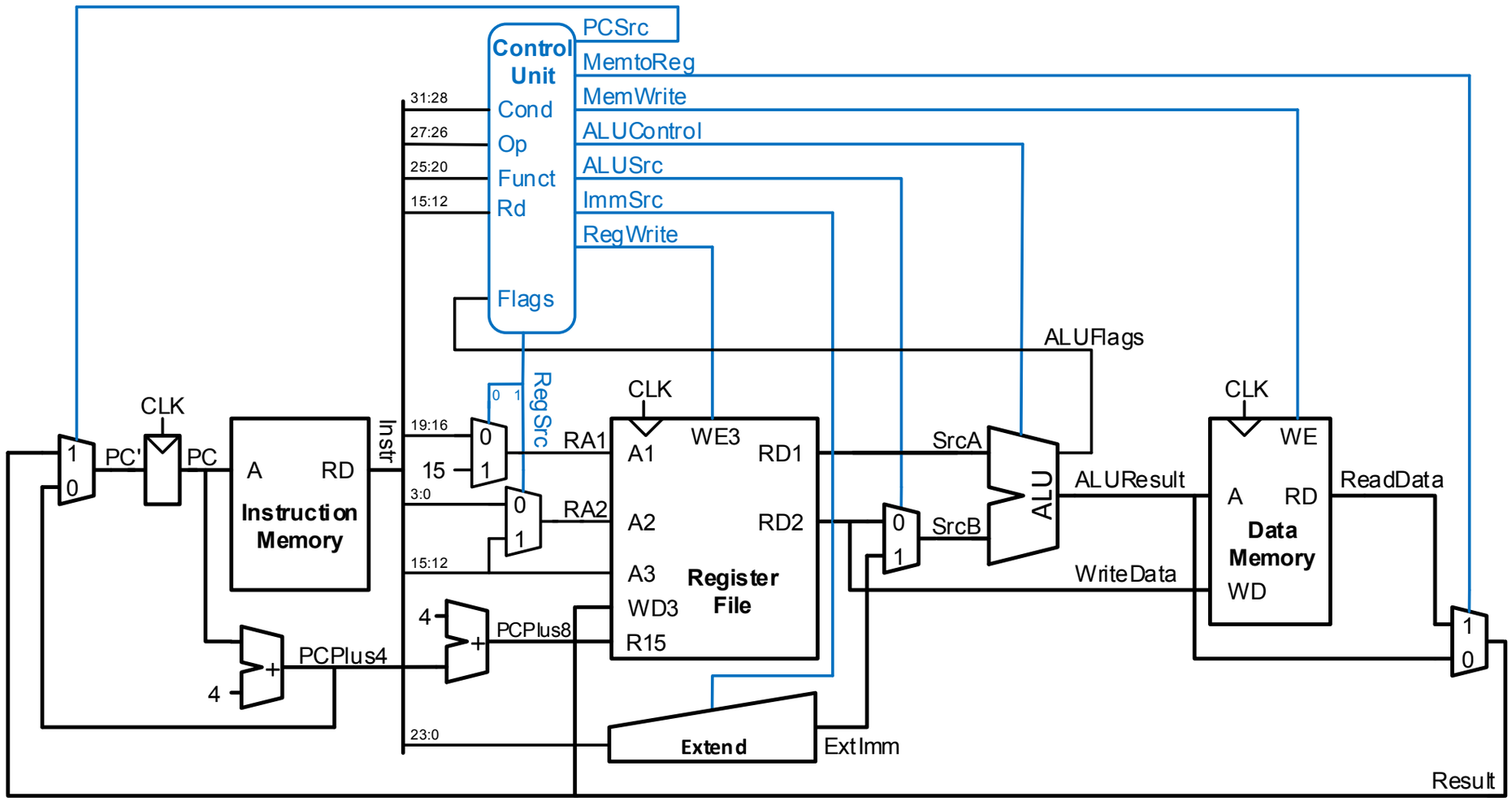
- Main Decoder
- ALU Decoder
- PC Logic



Single-Cycle Control: PC Logic

***PCS* = 1 if PC is written by an instruction or branch (B):**

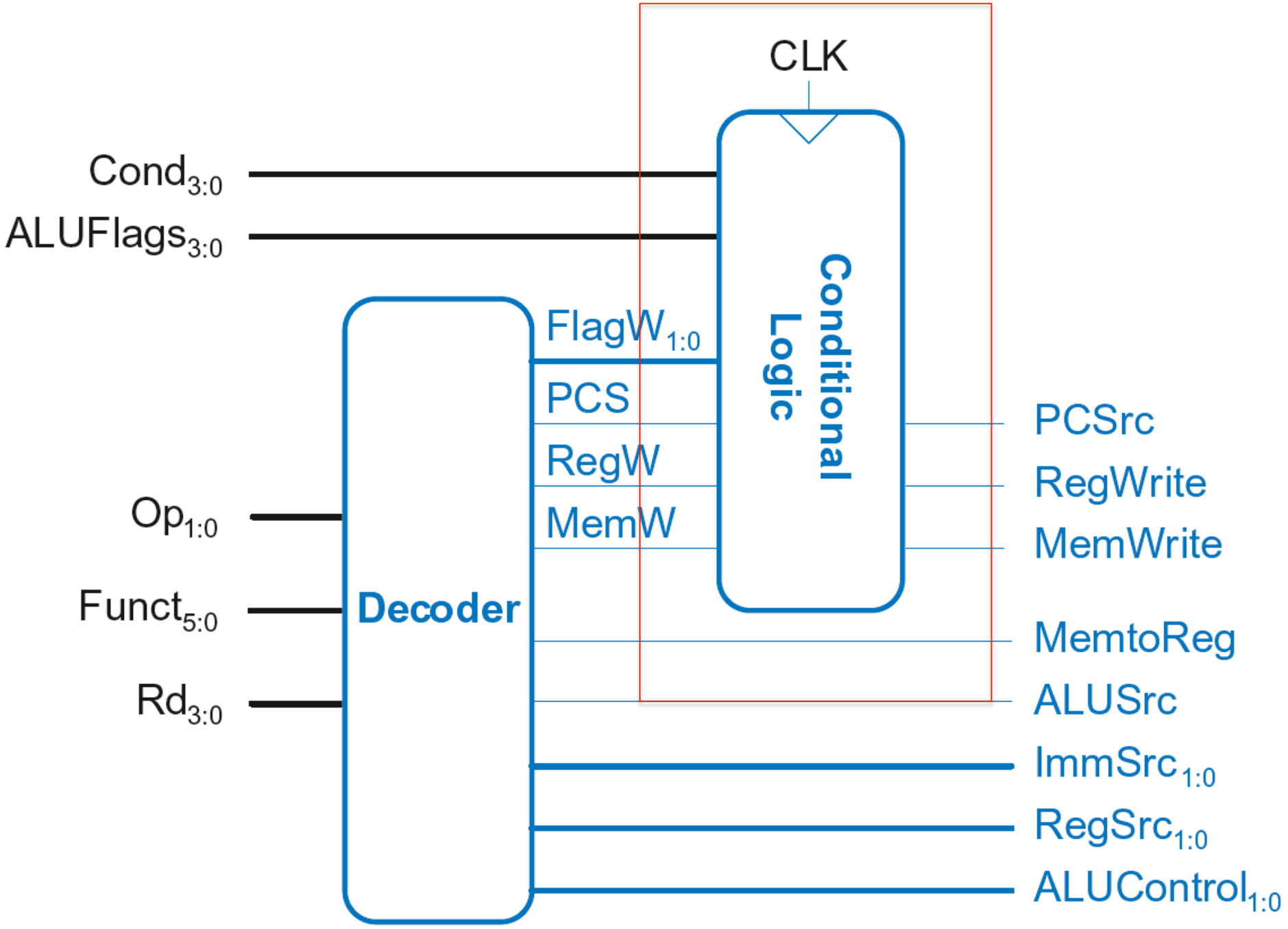
$$PCS = ((Rd == 15) \;\&\; RegW) \mid Branch$$



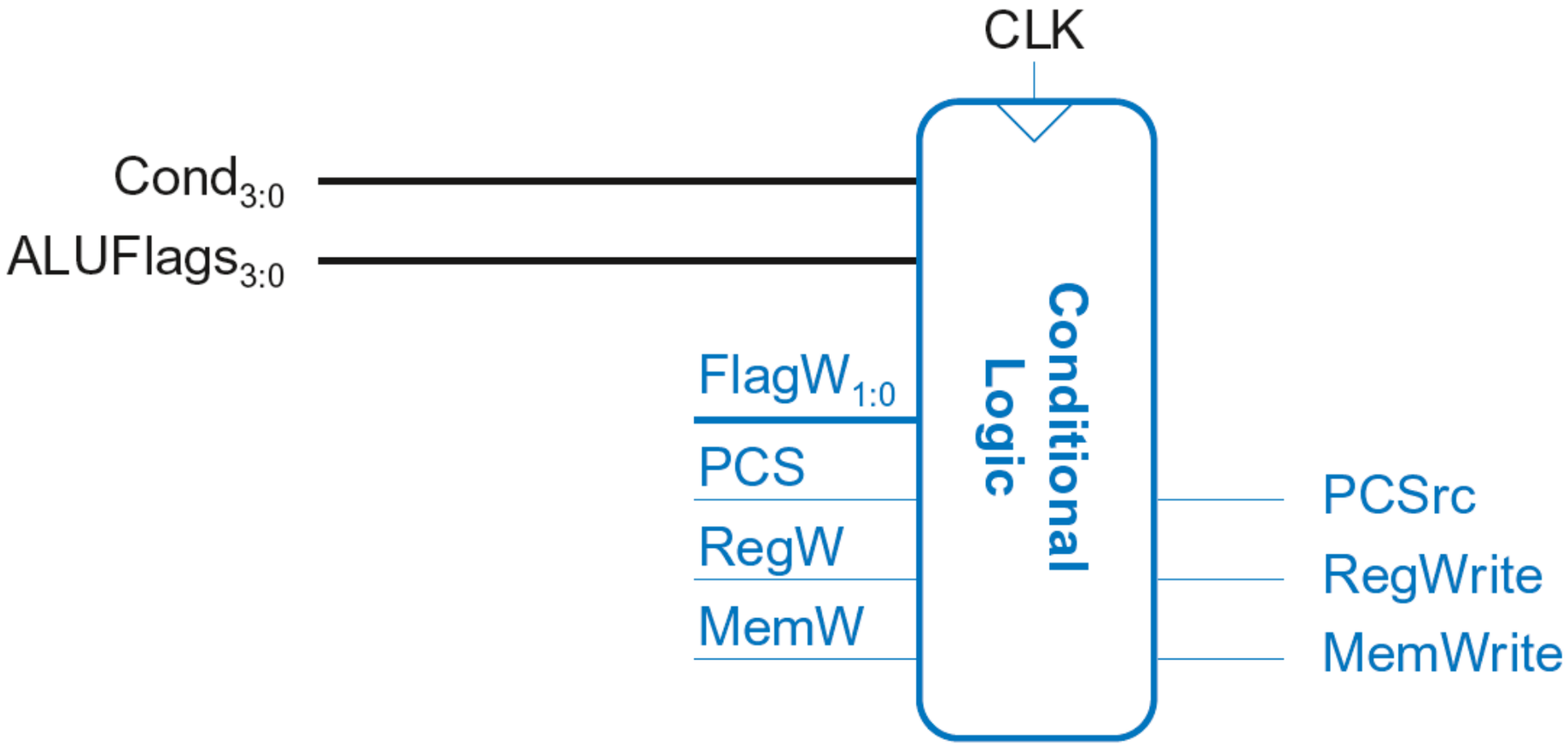
If instruction is executed: ***PCSrc* = *PCS***

Else ***PCSrc* = 0 (i.e., PC = PC + 4)**

Single-Cycle Control: Cond. Logic



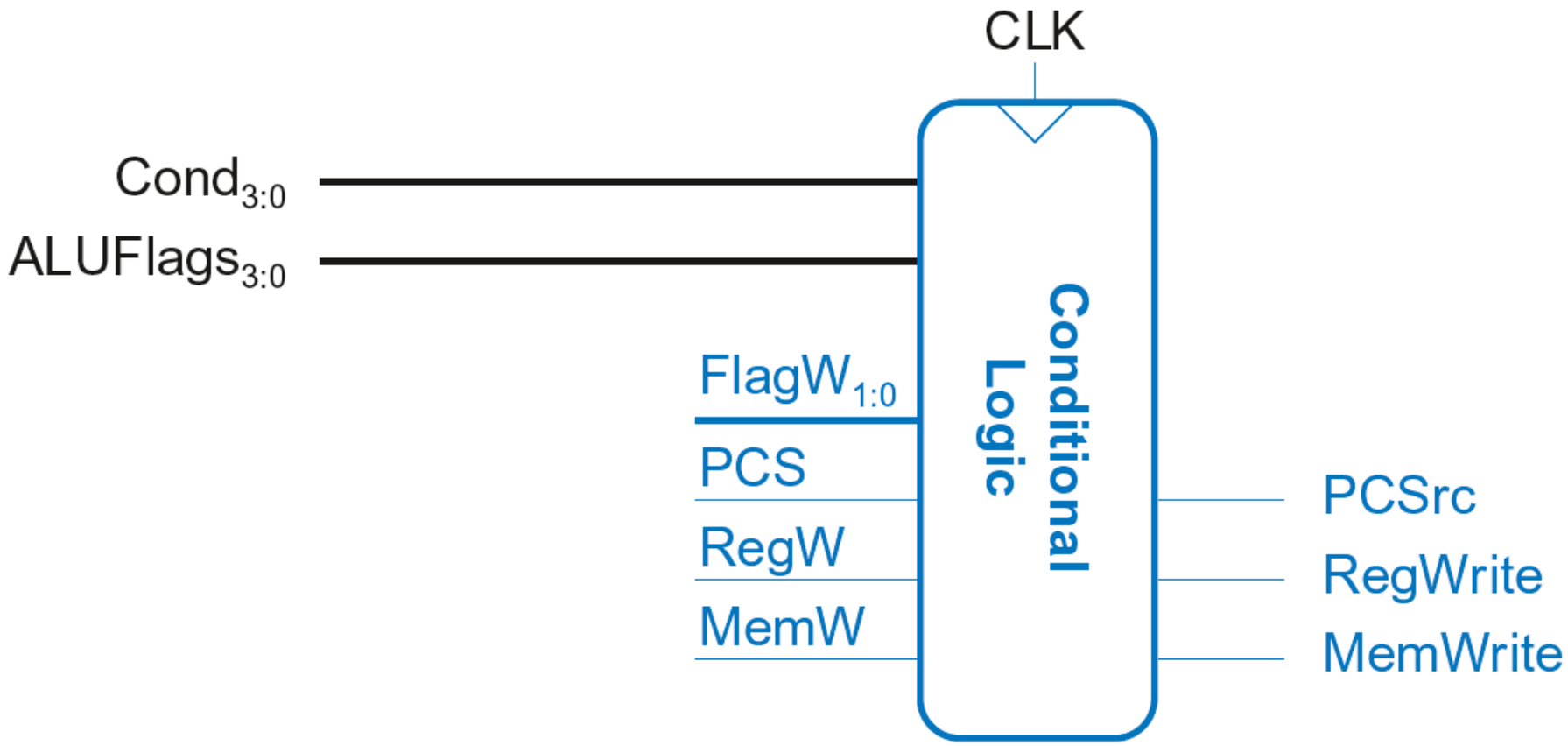
Conditional Logic



Function:

1. Check if instruction should execute (if not, force PCSrc, RegWrite, and MemWrite to 0)
2. Possibly update Status Register ($Flags_{3:0}$)

Conditional Logic

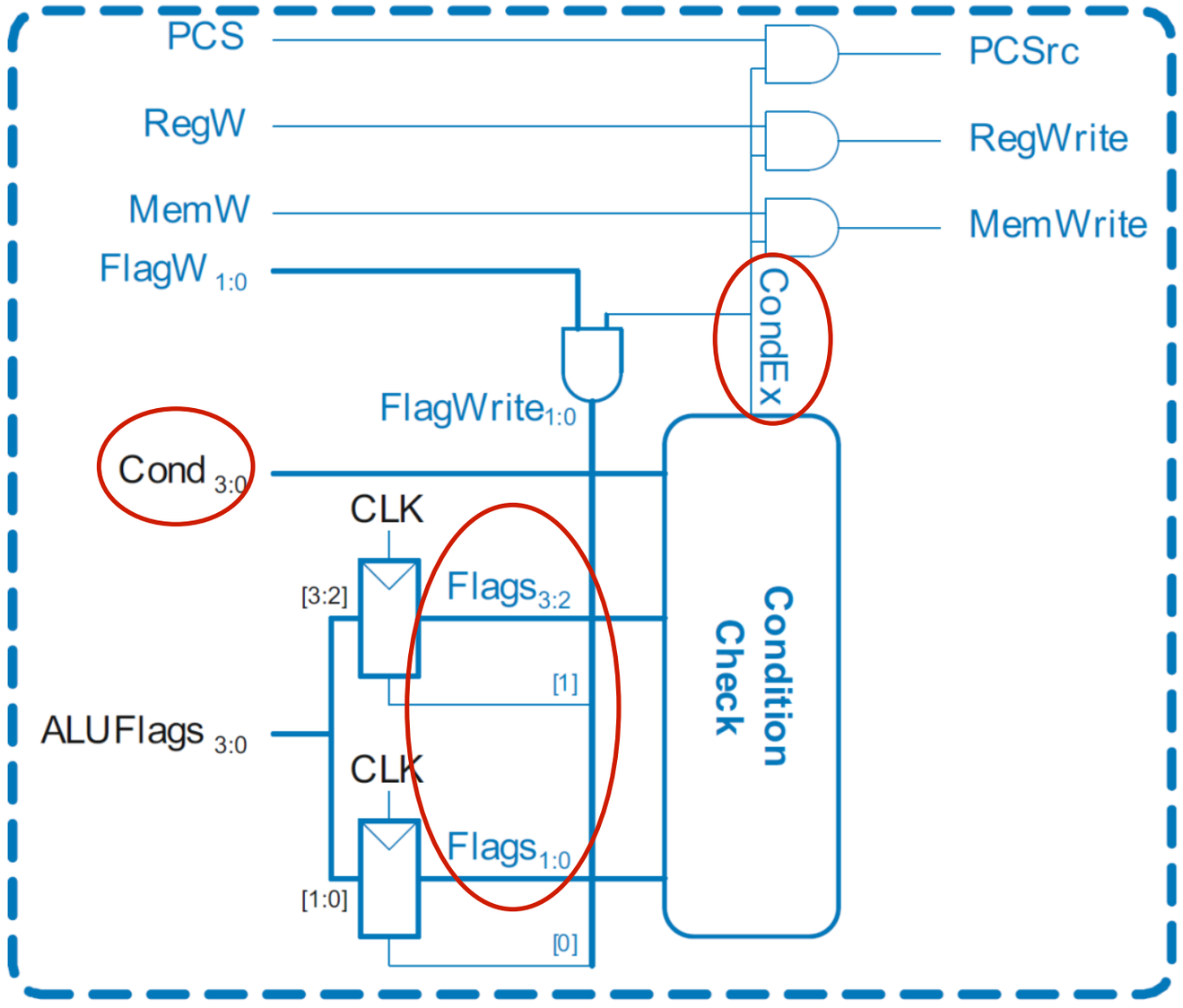


Function:

1. **Check if instruction should execute (if not, force PCSrc, RegWrite, and MemWrite to 0)**
2. Possibly update Status Register ($Flags_{3:0}$)

Conditional Logic: Conditional Execution

$Flags_{3:0}$ is the **status register**



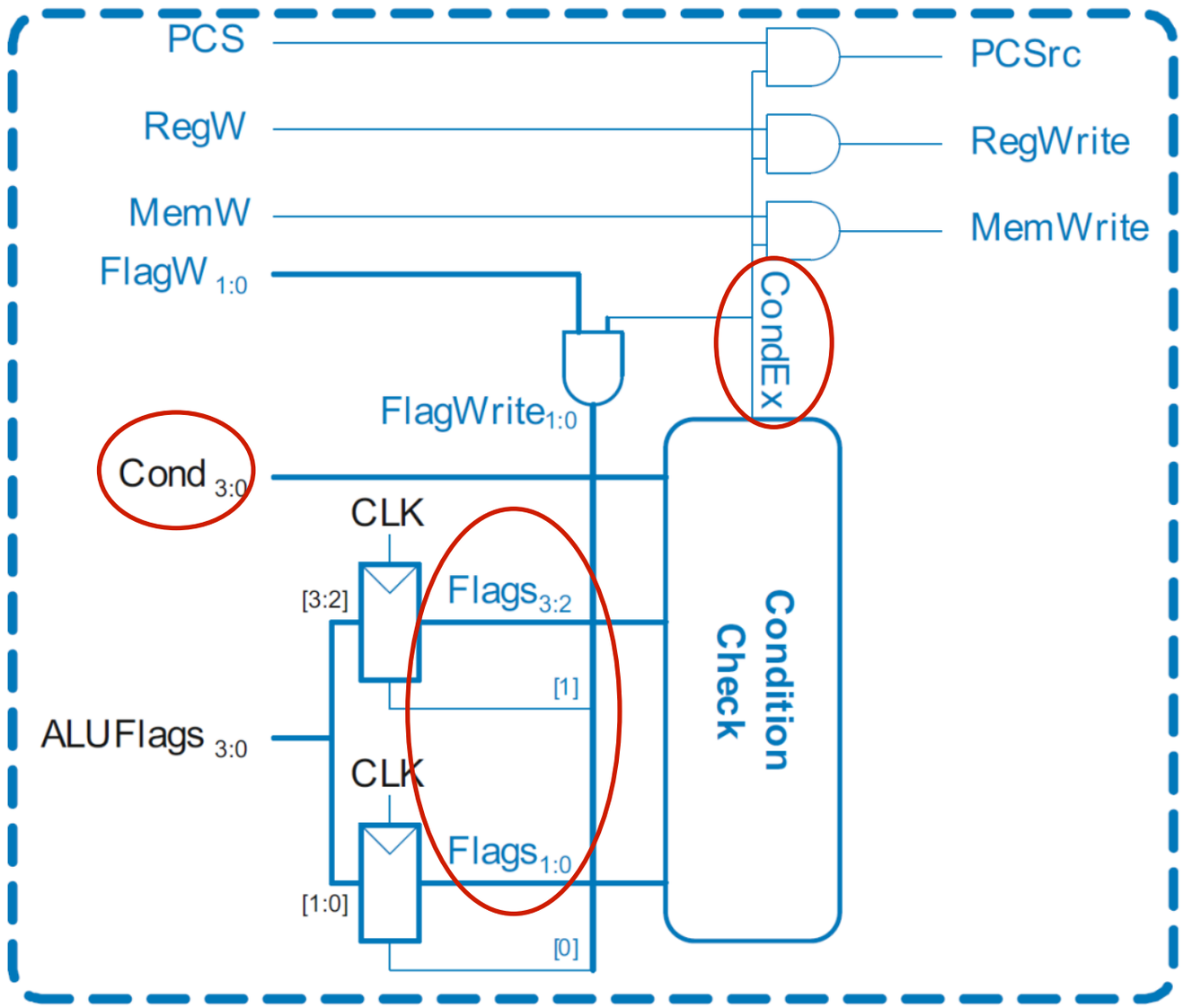
Depending on condition mnemonic (***$Cond_{3:0}$***) and condition flags (***$Flags_{3:0}$***) the instruction is executed (***CondEx*** = 1)

Conditional Logic: Conditional Execution

$Cond_{3:0}$	Mnemonic	Name	CondEx
0000	EQ	Equal	
0001	NE	Not equal	
0010	CS / HS	Carry set / Unsigned higher or same	
0011	CC / LO	Carry clear / Unsigned lower	
0100	MI	Minus / Negative	
0101	PL	Plus / Positive of zero	
0110	VS	Overflow / Overflow set	
0111	VC	No overflow / Overflow clear	
1000	HI	Unsigned higher	
1001	LS	Unsigned lower or same	
1010	GE	Signed greater than or equal	
1011	LT	Signed less than	
1100	GT	Signed greater than	
1101	LE	Signed less than or equal	
1110	AL (or none)	Always / unconditional	ignored

Conditional Logic: Conditional Execution

Flags$_{3:0}$ = NZCV

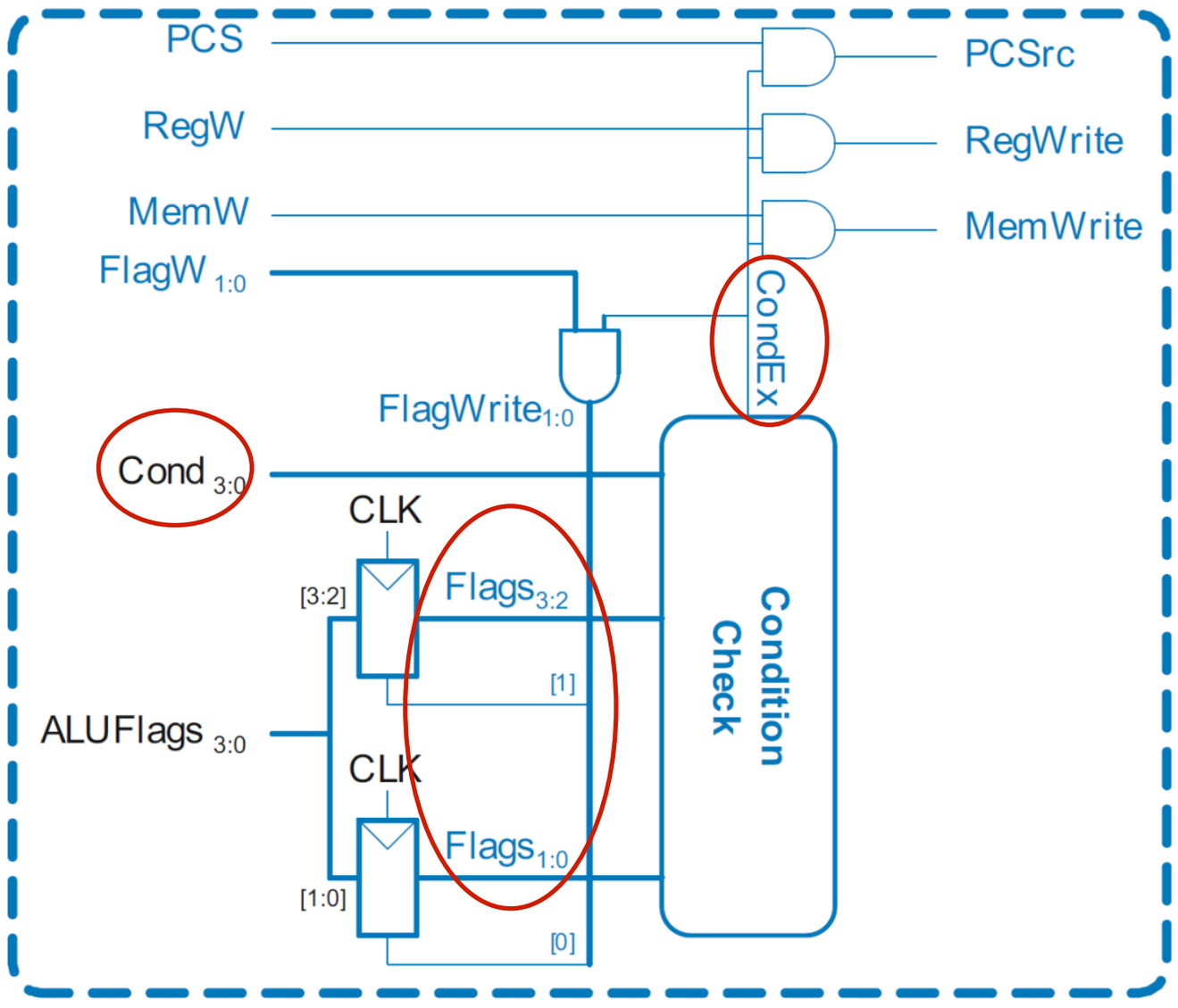


Example: AND R1, R2, R3

Cond$_{3:0}$=1110 (unconditional) => **CondEx** = 1

Conditional Logic: Conditional Execution

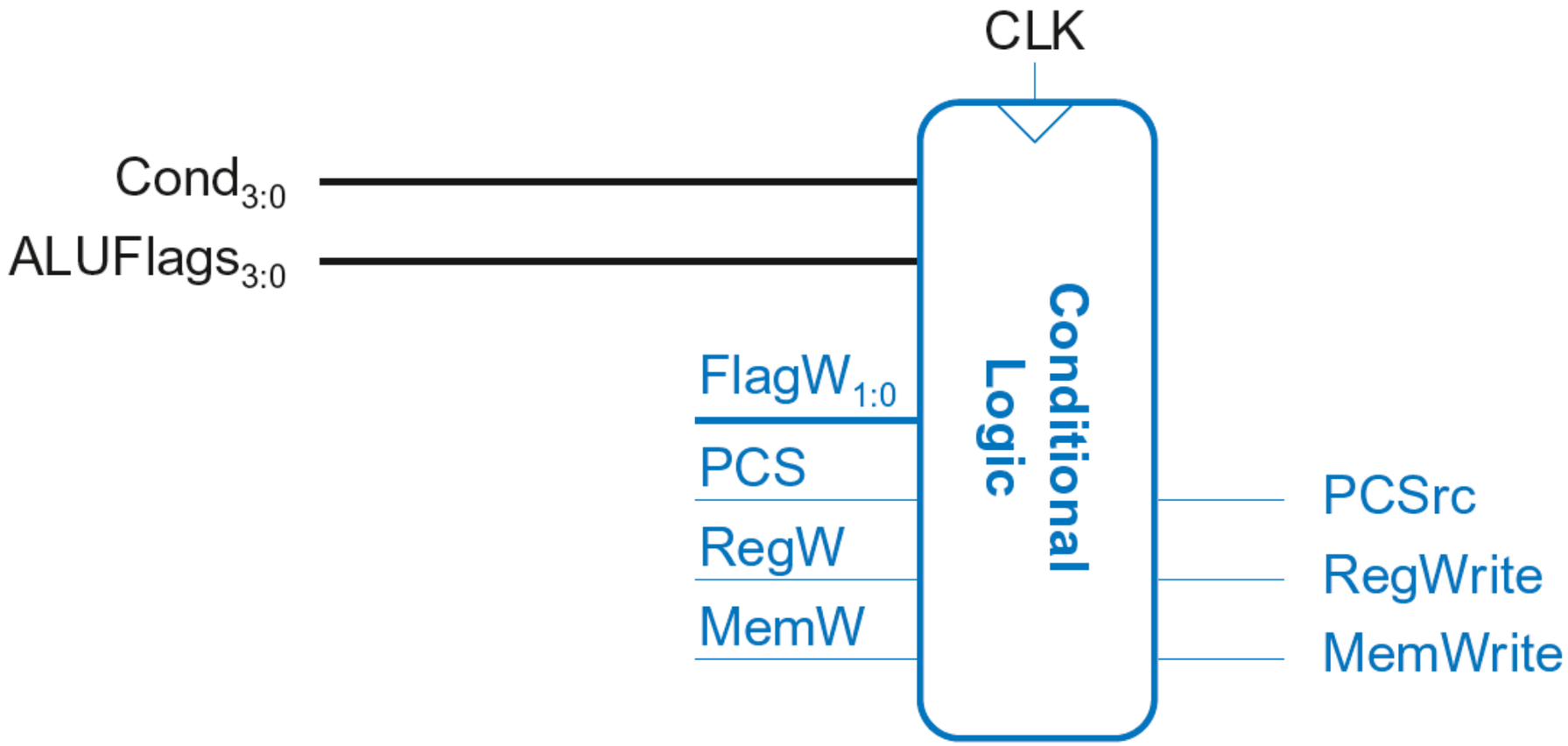
Flags$_{3:0}$ = NZCV



Example: EOREQ R5, R6, R7

Cond$_{3:0}$=0000 (EQ): if **Flags$_{3:2}$**=0100 => **CondEx** = 1

Conditional Logic

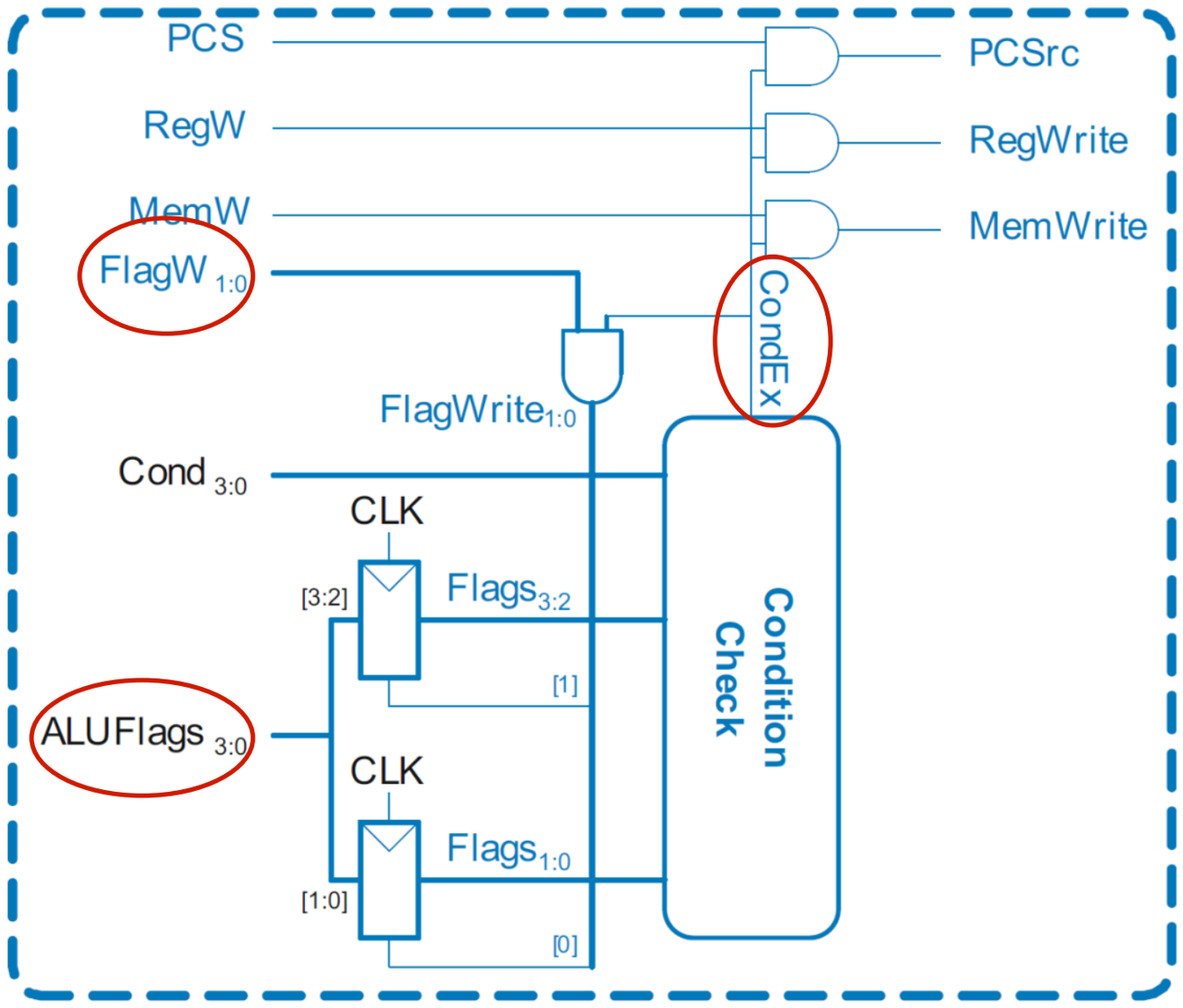


Function:

1. Check if instruction should execute (if not, force PCSrc, RegWrite, and MemWrite to 0)
2. **Possibly update Status Register ($Flags_{3:0}$)**

Conditional Logic: Update (Set) Flags

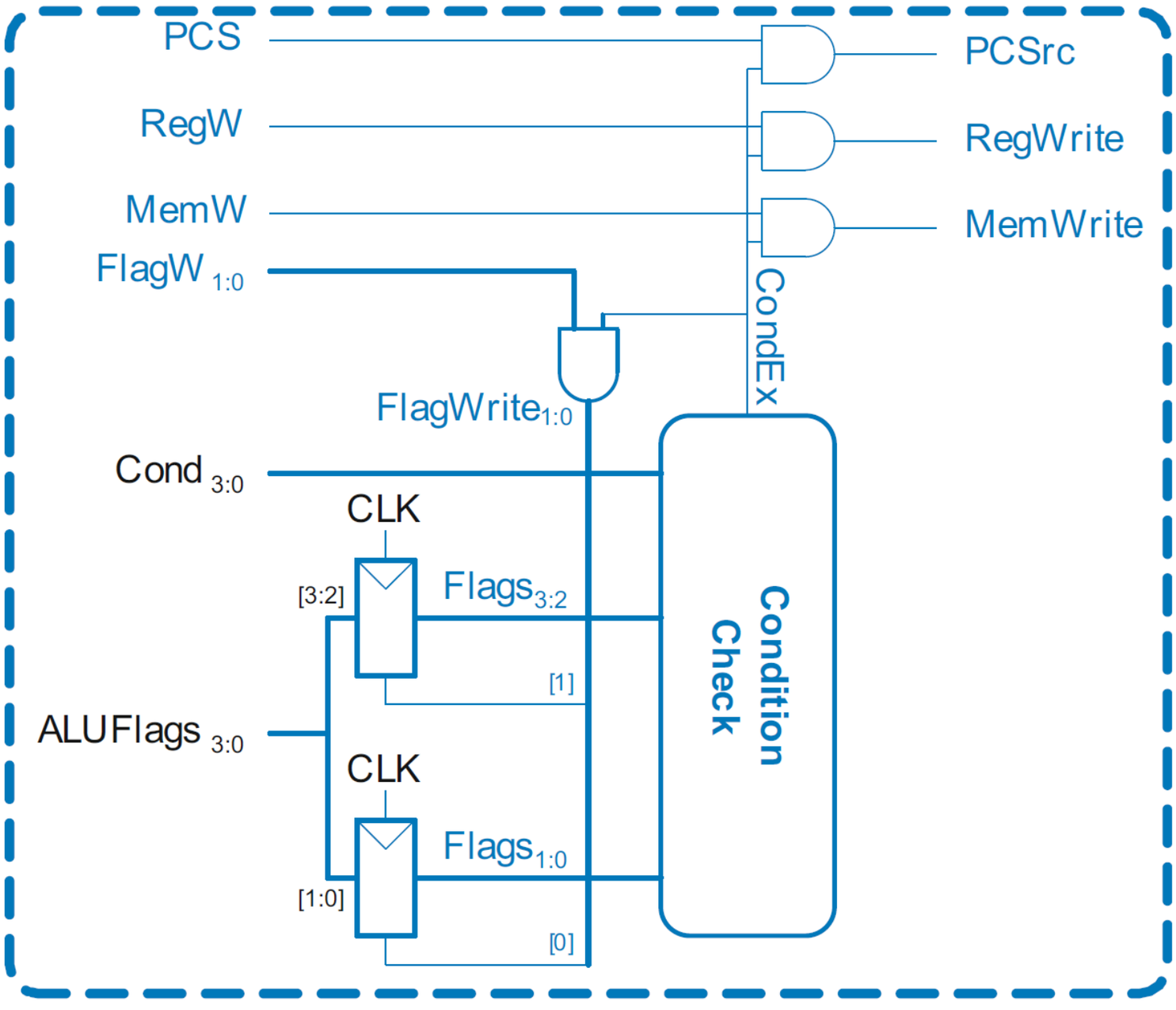
Flags$_{3:0}$ = NZCV



Flags$_{3:0}$ **updated** (with ALUFlags$_{3:0}$) if:

- **FlagW** is 1 (i.e., the instruction's S-bit is 1) AND
- **CondEx** is 1 (the instruction should be executed)

Conditional Logic: Update (Set) Flags



Recall:

- ADD, SUB update **all** Flags
- AND, OR update **NZ only**
- So Flags status register has two write enables: **$FlagW_{1:0}$**

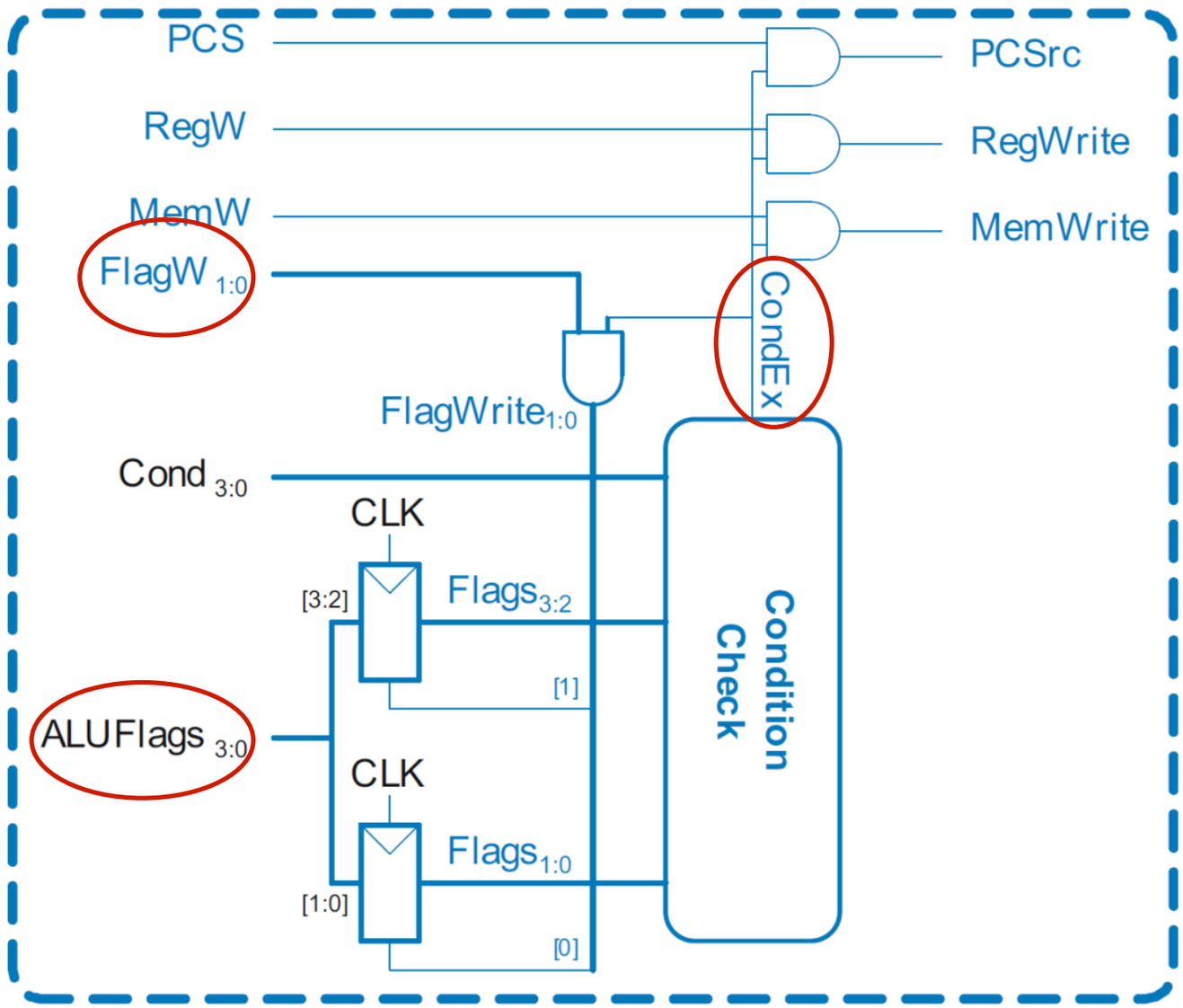
Review: ALU Decoder

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

- $FlagW_1$ = 1: *NZ* ($Flags_{3:2}$) should be saved
- $FlagW_0$ = 1: *CV* ($Flags_{1:0}$) should be saved

Conditional Logic: Update (Set) Flags

All Flags updated



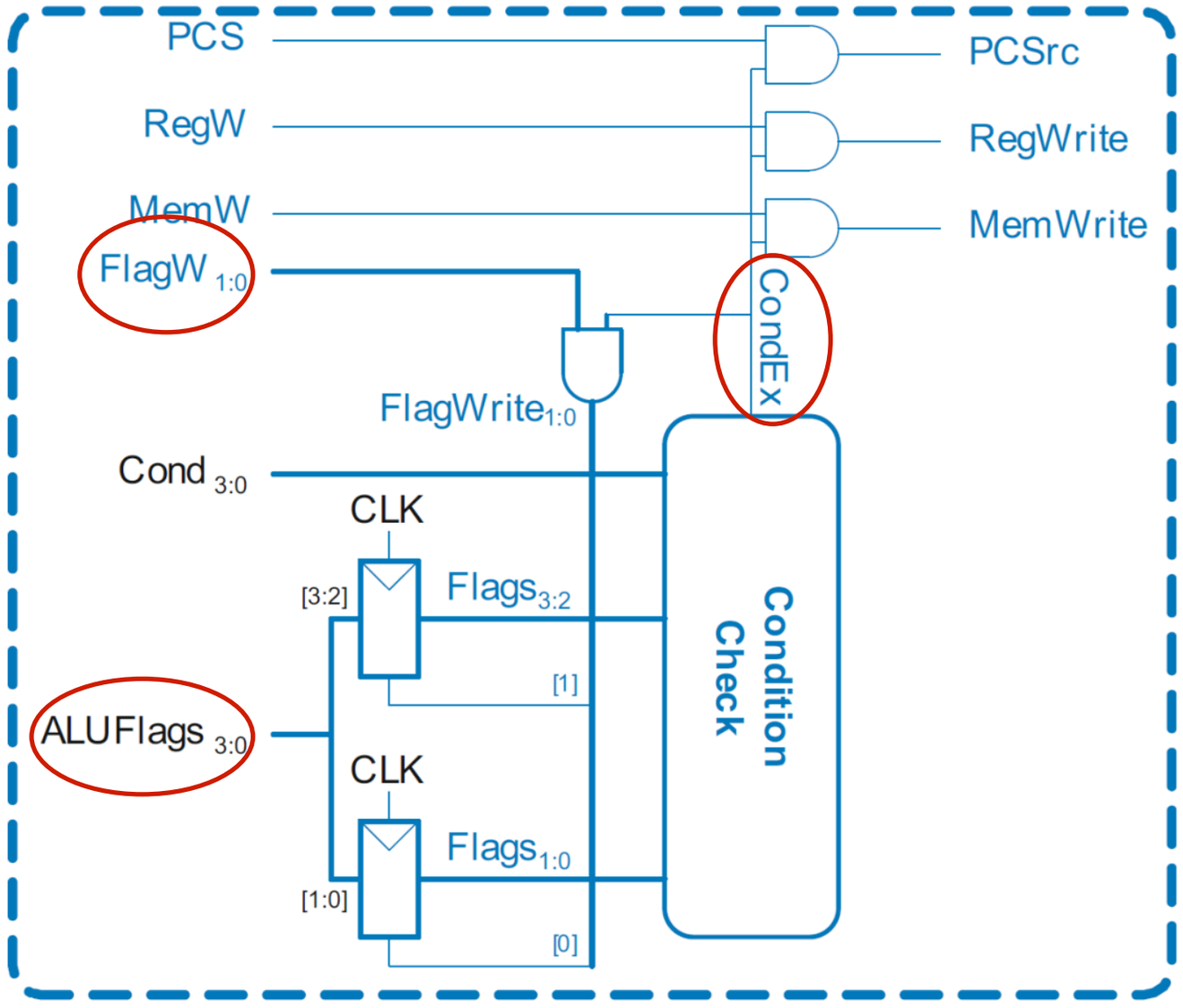
Example: SUBS R5, R6, R7

$FlagW_{1:0}$ = 11 AND **CondEx** = 1 (unconditional) => **$FlagWrite_{1:0}$** = 11

Conditional Logic: Update (Set) Flags

Flags$_{3:0}$ = NZCV

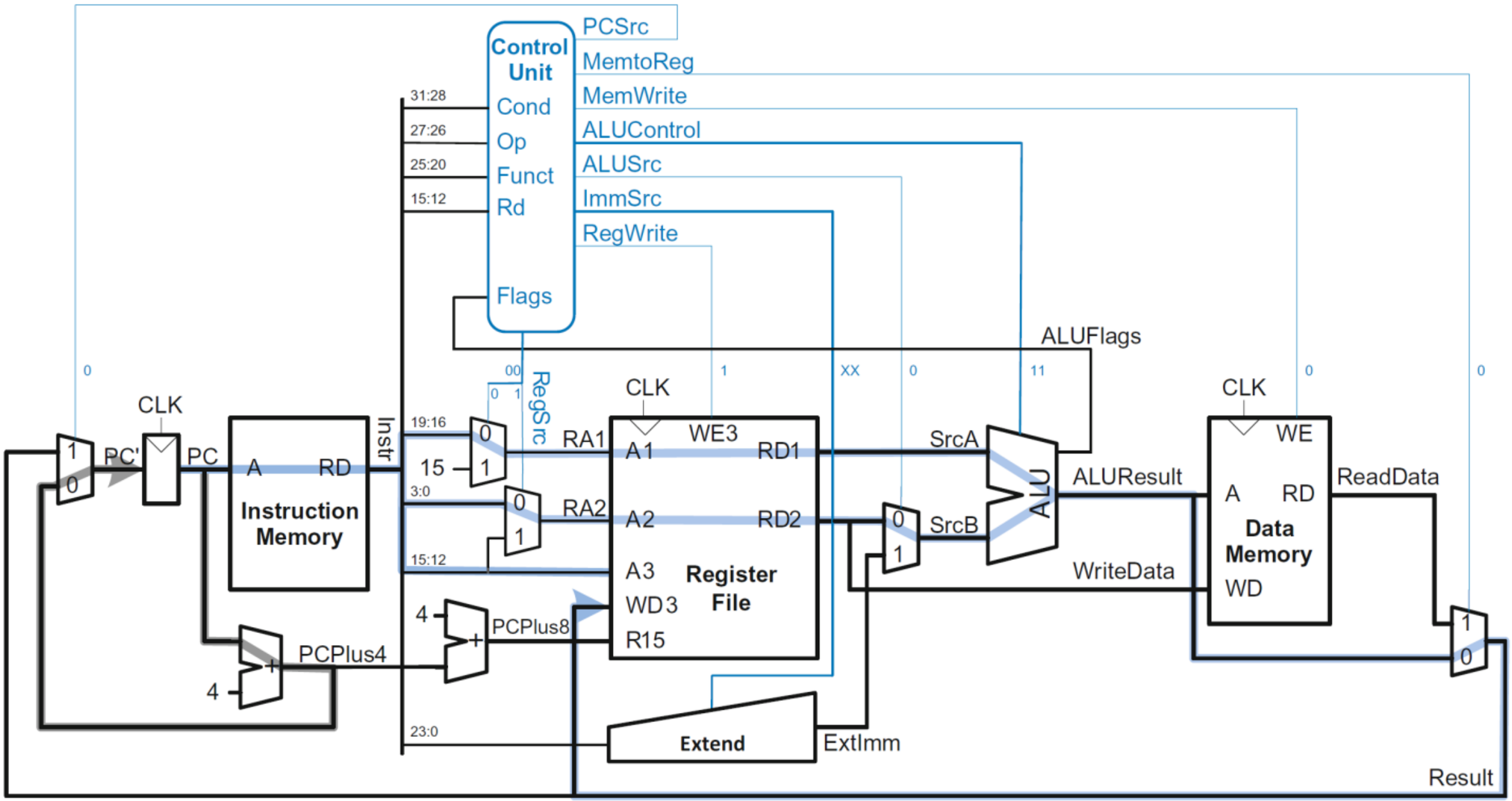
- Only **Flags$_{3:2}$** updated
- i.e., only **NZ** Flags updated



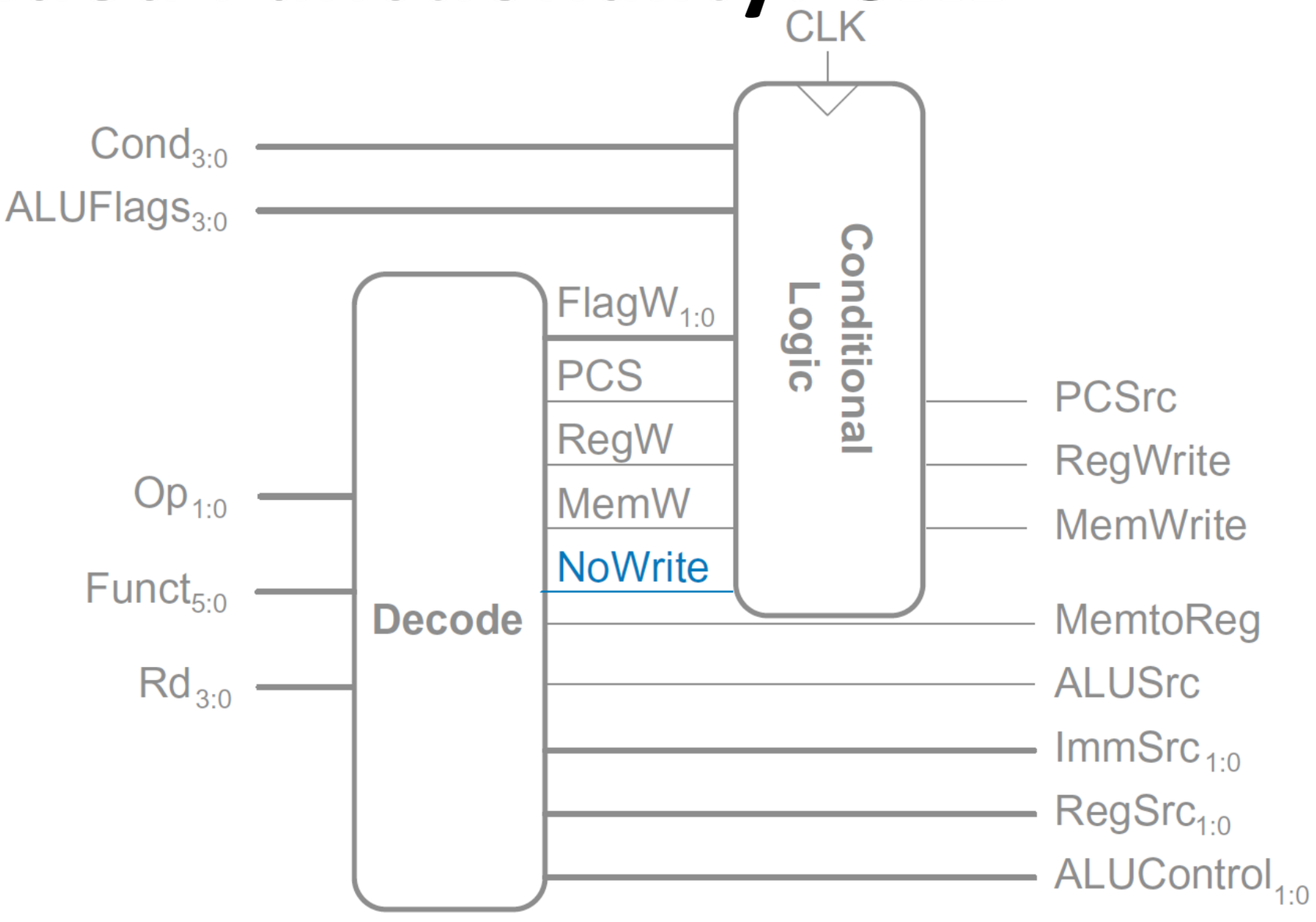
Example: ANDS R7, R1, R3

FlagW$_{1:0}$ = 10 AND **CondEx** = 1 (unconditional) => **FlagWrite$_{1:0}$** = 10

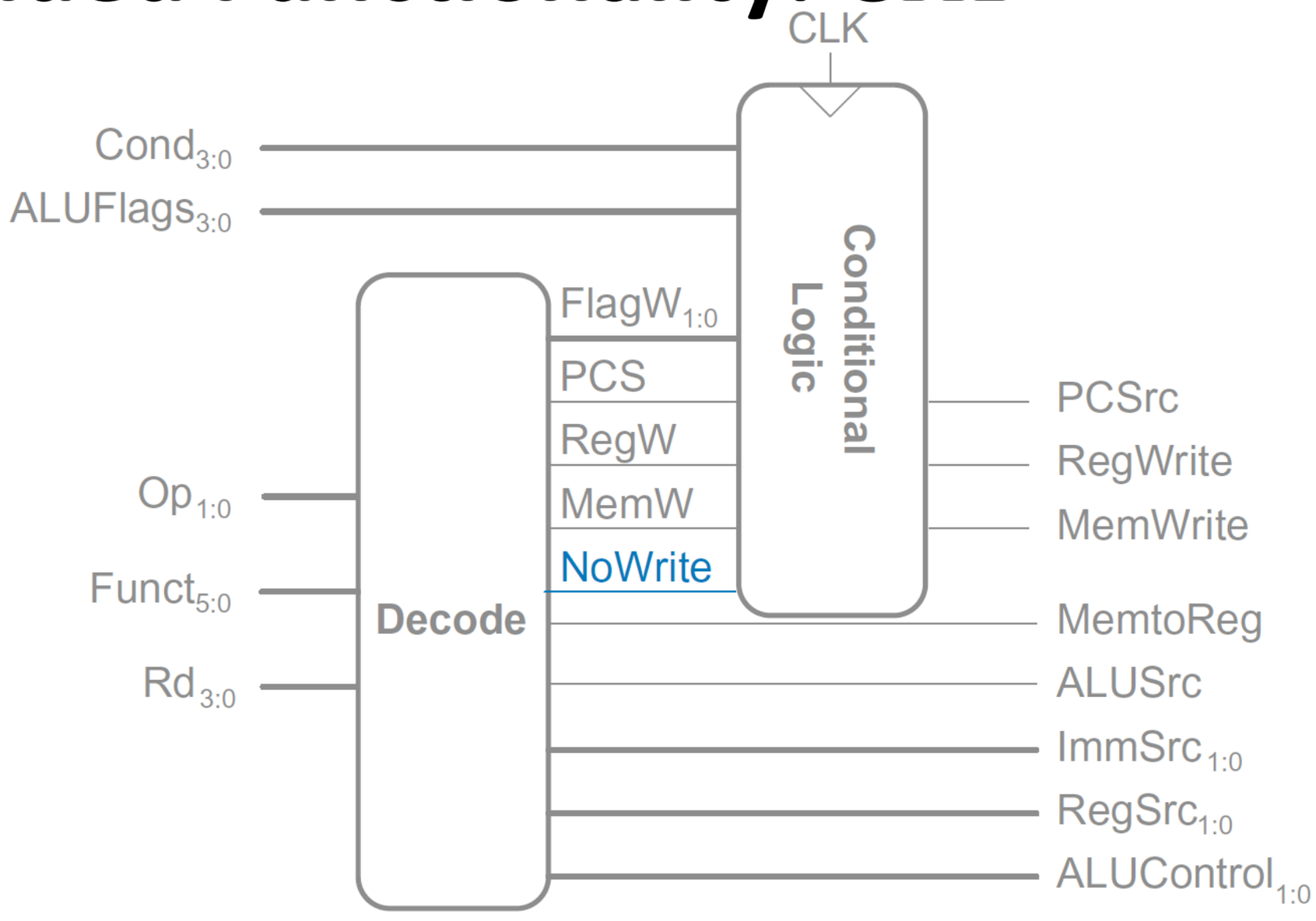
Example: ORR


PCSrc
Control Unit
MemtoReg
MemWrite
ALUControl
ALUSrc
ImmSrc
RegWrite
31:28
Cond
27:26
Op
25:20
Funct
15:12
Rd
Flags
ALUFlags
0
00
0 1
RegSrc
1
XX
0
11
0
0
CLK
CLK
CLK
1
0
PC'
PC
A
RD
Instruction Memory
Instr
19:16
15
0
1
3:0
0
1
15:12
RA1
RA2
A1
A2
A3
WD3
R15
WE3
RD1
RD2
Register File
SrcA
SrcB
0
1
ALU
ALUResult
WriteData
WE
A
RD
Data Memory
WD
ReadData
1
0
4
PCPlus8
4
PCPlus4
23:0
Extend
ExtImm
Result

Extended Functionality: CMP

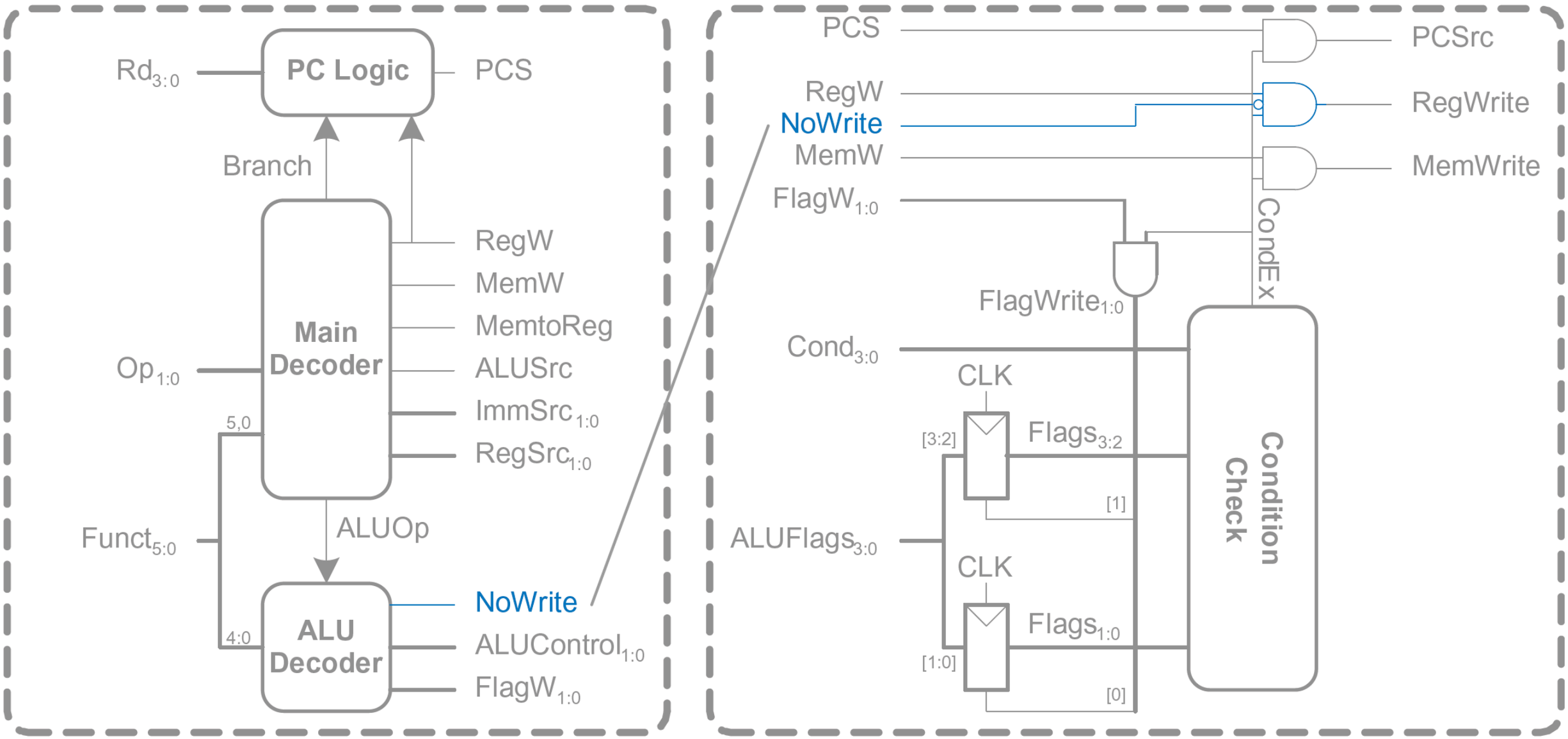


Extended Functionality: CMP



No change to datapath

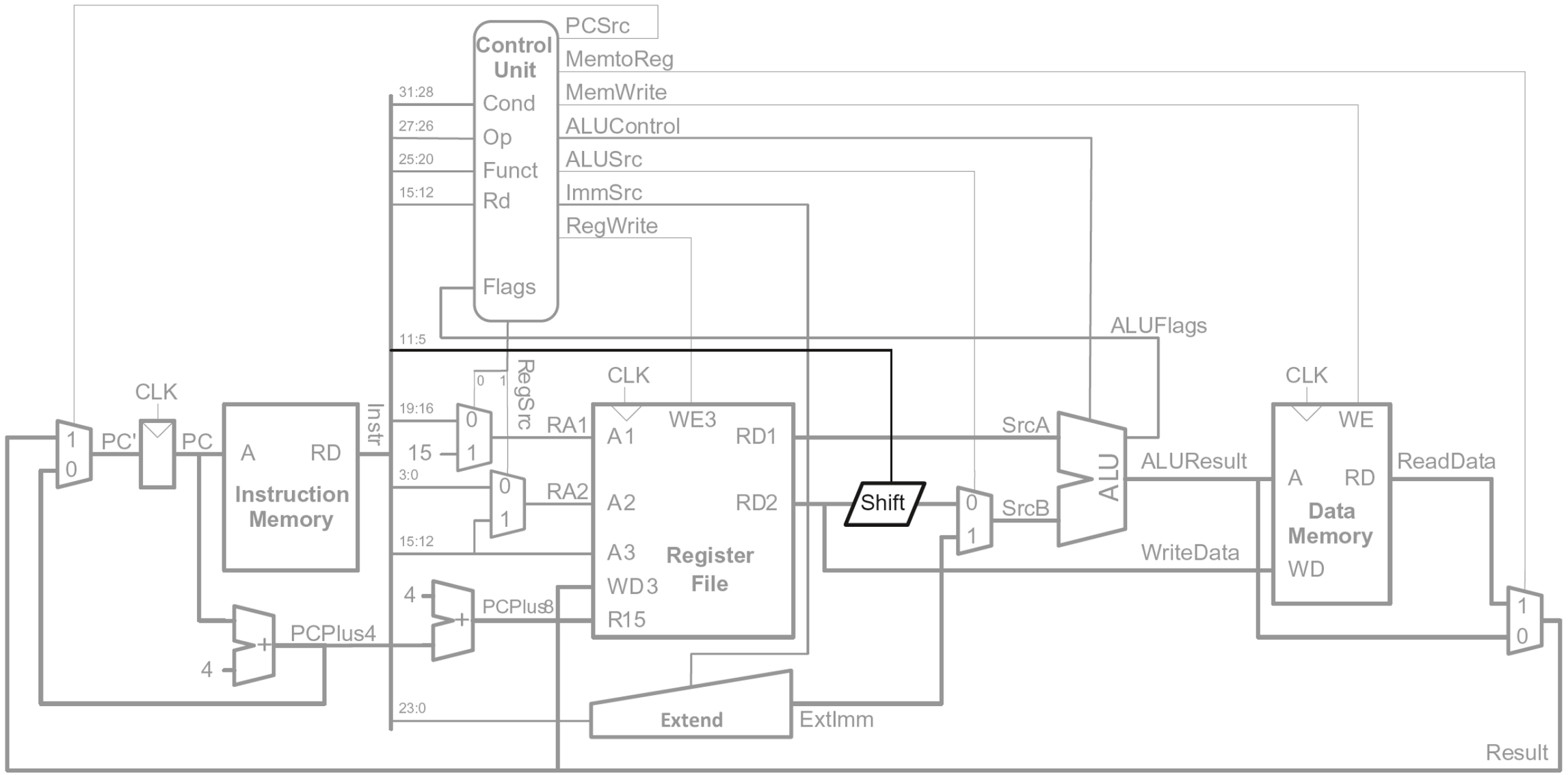
Extended Functionality: CMP



Extended Functionality: CMP

ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$	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**

Extended Functionality: Shifted Register



ADD R7, R2, R12, LSR #5

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
14	0	0	4	0	2	7	5	01_2	0	12
cond	op	I	cmd	S	rn	rd	shamt5	sh		rm

Processor Performance

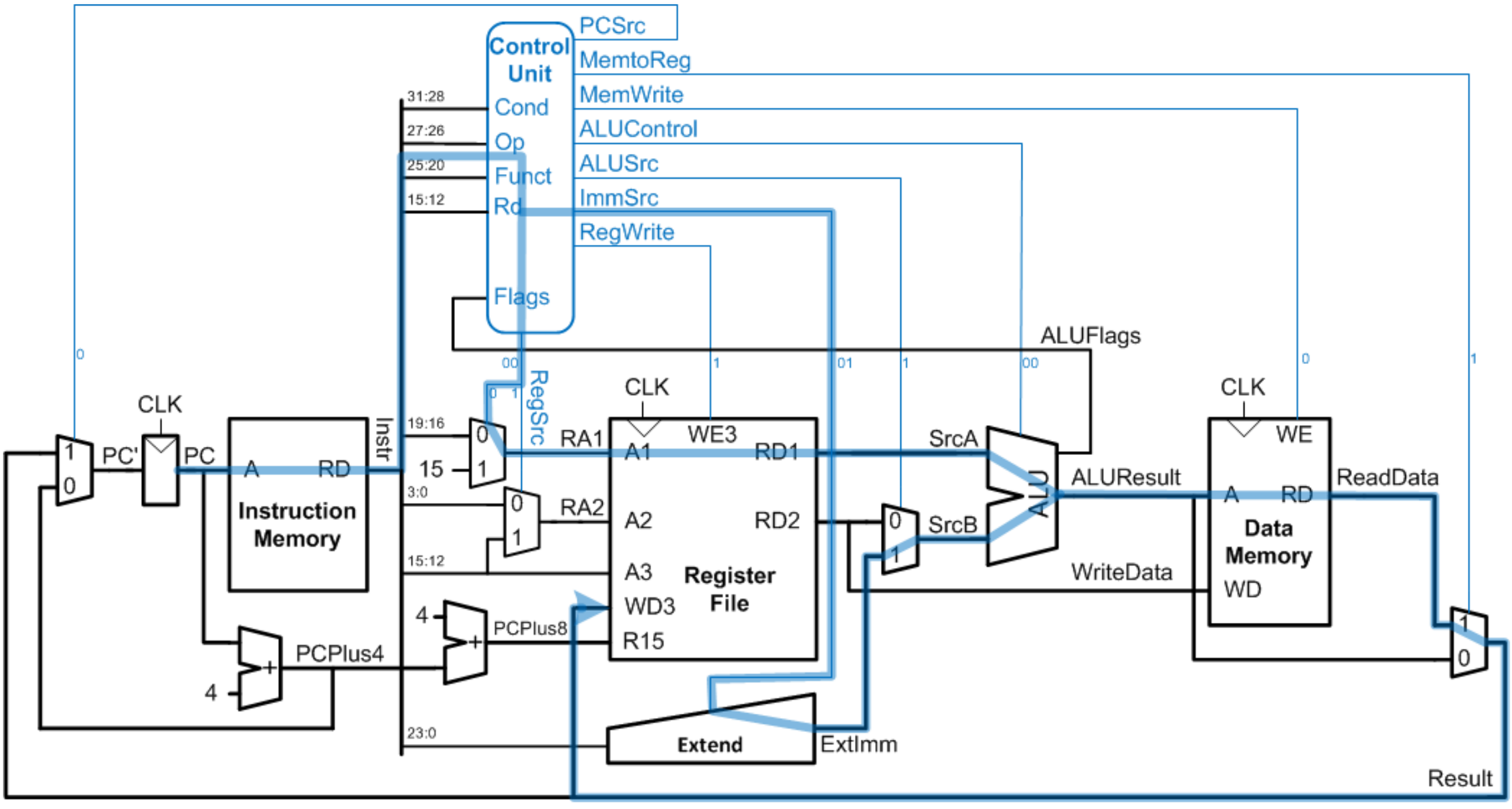
- **Program execution time**

Execution Time = (#instructions)(cycles/instruction)(seconds/cycle)

$$\#\ \text{instructions} \times \text{CPI} \times T_C$$

- **Definitions:**
  - CPI: cycles per instruction (Cycles/instruction)
  - clock period: seconds/cycle
  - IPC: instructions per cycle (instructions/cycle)
- **Challenge is to satisfy constraints of:**
  - Cost
  - Power
  - Performance

Single-Cycle Performance



T_C limited by critical path (`LDR`)

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Decoder	t_{dec}	70
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Decoder	t_{dec}	70
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

$$T_{c1} = t_{pcq\_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup}$$

$$= [50 + 2(200) + 70 + 100 + 120 + 2(25) + 60] \text{ ps}$$

$$= \mathbf{840 \text{ ps}}$$

Single-Cycle Performance Example

Program with 100 billion instructions:

Execution Time = # instructions x CPI x T_C

= $(100 \times 10^9)(1)(840 \times 10^{-12}\ s)$

= **84 seconds**

Multicycle ARM Processor

- **Single-cycle:**
  + simple
  - cycle time limited by longest instruction (`LDR`)
  - separate memories for instruction and data
  - 3 adders/ALUs
- **Multicycle processor addresses these issues by breaking instruction into shorter steps**
  - shorter instructions take fewer steps
  - can re-use hardware
  - cycle time is faster

Multicycle ARM Processor

- **Single-cycle:**
  + simple
  - cycle time limited by longest instruction (`LDR`)
  - separate memories for instruction and data
  - 3 adders/ALUs
- **Multicycle:**
  + higher clock speed
  + simpler instructions run faster
  + reuse expensive hardware on multiple cycles
  - sequencing overhead paid many times

Reading

- **Chapter 7**
  - **Sections 7.1, 7.2 and 7.3**