

ECE-332:437
DIGITAL SYSTEMS DESIGN (DSD)

Fall 2016 – Lecture 11
Micro Architecture (ARM)

Nagi Naganathan
November 17, 2016

Announcements – November 17, 2016

- Final Quiz today
- Final HW to be assigned today
- Mid Term 2 – December 1, 2016
- Chapters 5, 6, 7, 8
 - Architecture – MIPS (Chapter 6), ARM (Lecture Slides)
 - Lecture Slides 8,10
 - Microarchitecture – MIPS (Chapter 7), ARM (Lecture Slides) –
 - Lecture Slides 9,11
 - Memory and I/O Sub-Systems – MIPS (Chapter 8) – Sections 8.1, 8.2 & 8.3
 - Lecture Slides 12
- Last class on Dec 8
 - Chapter 8 – Sections 8.4, 8.5, 8.6, 8.7, 8.8
 - ARM Cache

Topics to cover today – November 17, 2016

- Lecture 11 – ARM Microarchitecture
- Lecture 12 – Memory (Cache)
 - Chapter 8 from the text book
 - Sections 8.1, 8.2 & 8.3

Chapter 7

Digital Design and Computer Architecture: ARM® Edition

Sarah L. Harris and David Money Harris



Quick Recap – ARM Architecture

- The ARM is a 32-bit architecture.
- When used in relation to the ARM:
 - **Byte** means 8 bits
 - **Halfword** means 16 bits (two bytes)
 - **Word** means 32 bits (four bytes)
- Most ARM's implement two instruction sets
 - 32-bit ARM Instruction Set
 - 16-bit Thumb Instruction Set
- Jazelle cores can also execute Java bytecode

ARM Register Set

| Name | Use |
|-----------------|--|
| R0 | Argument / return value / temporary variable |
| R1-R3 | Argument / temporary variables |
| R4-R11 | Saved variables |
| R12 | Temporary variable |
| R13 (SP) | Stack Pointer |
| R14 (LR) | Link Register |
| R15 (PC) | Program Counter |



Operands: Registers

- **Registers:**
 - R before number, all capitals
 - Example: “R0” or “register zero” or “register R0”



Operands: Registers

- **Registers used for specific purposes:**
 - **Saved registers:** R4-R11 hold variables
 - **Temporary registers:** R0-R3 and R12, hold intermediate values
 - Discuss others later



Generating Constants

Generating small constants using move (MOV):

C Code

```
//int: 32-bit signed word  
int a = 23;  
int b = 0x45;
```

ARM Assembly Code

```
; R0 = a, R1 = b  
MOV R0, #23  
MOV R1, #0x45
```



Reading Memory

- Memory read called *load*
- **Mnemonic:** *load register* (LDR)
- **Format:**

LDR R0, [R1, #12]

Address calculation:

- add *base address* (R1) to the *offset* (12)
- address = (R1 + 12)

Result:

- R0 holds the data at memory address (R1 + 12)



Writing Memory

- Memory write are called **stores**
- **Mnemonic:** *store register* (STR)



Writing Memory

- **Example:** Store the value held in R7 into memory word 21.



Writing Memory

- **Example:** Store the value held in R7 into memory word 21.
- Memory address = $4 \times 21 = 84 = 0x54$

ARM assembly code

```
MOV R5, #0
STR R7, [R5, #0x54]
```

| Word address | Data | Word number |
|--------------|-----------------|-------------|
| ⋮ | ⋮ | ⋮ |
| 00000010 | C D 1 9 A 6 5 B | Word 4 |
| 0000000C | 4 0 F 3 0 7 8 8 | Word 3 |
| 00000008 | 0 1 E E 2 8 4 2 | Word 2 |
| 00000004 | F 2 F 1 A C 0 7 | Word 1 |
| 00000000 | A B C D E F 7 8 | Word 0 |

Width = 4 bytes



Logical Instructions

- AND
- ORR
- EOR (**XOR**)
- BIC (**Bit Clear**)
- MVN (**MoVe and NOT**)



Logical Instructions: Examples

Source registers

| | | | | |
|----|-----------|-----------|-----------|-----------|
| R1 | 0100 0110 | 1010 0001 | 1111 0001 | 1011 0111 |
| R2 | 1111 1111 | 1111 1111 | 0000 0000 | 0000 0000 |

Assembly code

AND R3, R1, R2

ORR R4, R1, R2

EOR R5, R1, R2

BIC R6, R1, R2

MVN R7, R2

Result

| | | | | |
|----|-----------|-----------|-----------|-----------|
| R3 | 0100 0110 | 1010 0001 | 0000 0000 | 0000 0000 |
| R4 | 1111 1111 | 1111 1111 | 1111 0001 | 1011 0111 |
| R5 | 1011 1001 | 0101 1110 | 1111 0001 | 1011 0111 |
| R6 | 0000 0000 | 0000 0000 | 1111 0001 | 1011 0111 |
| R7 | 0000 0000 | 0000 0000 | 1111 1111 | 1111 1111 |



Logical Instructions: Uses

- AND or BIC: useful for **masking** bits

Example: Masking all but the least significant byte of a value

$0xF234012F \text{ AND } 0x000000FF = 0x0000002F$

$0xF234012F \text{ BIC } 0xFFFFFFFF00 = 0x0000002F$



Logical Instructions: Uses

- AND or BIC: useful for **masking** bits

Example: Masking all but the least significant byte of a value

$0xF234012F \text{ AND } 0x000000FF = 0x0000002F$

$0xF234012F \text{ BIC } 0xFFFFFFFF00 = 0x0000002F$

- ORR: useful for **combining** bit fields

Example: Combine $0xF2340000$ with $0x000012BC$:

$0xF2340000 \text{ ORR } 0x000012BC = 0xF23412BC$



ARM Condition Flags

| Flag | Name | Description |
|----------|------------------|--|
| <i>N</i> | N egative | Instruction result is negative |
| <i>Z</i> | Z ero | Instruction results in zero |
| <i>C</i> | C arry | Instruction causes an unsigned carry out |
| <i>V</i> | oV erflow | Instruction causes an overflow |



Condition Mnemonics

- Instruction may be *conditionally executed* based on the condition flags
- Condition of execution is encoded as a *condition mnemonic* appended to the instruction mnemonic

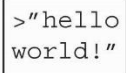


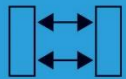
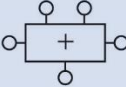
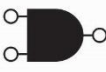
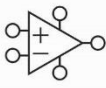


Example: `CMP R1, R2`
 `SUBNE R3, R5, R8`

- **NE:** condition mnemonic
- SUB will only execute if $R1 \neq R2$
(i.e., $Z = 0$)



Chapter 7 :: Topics

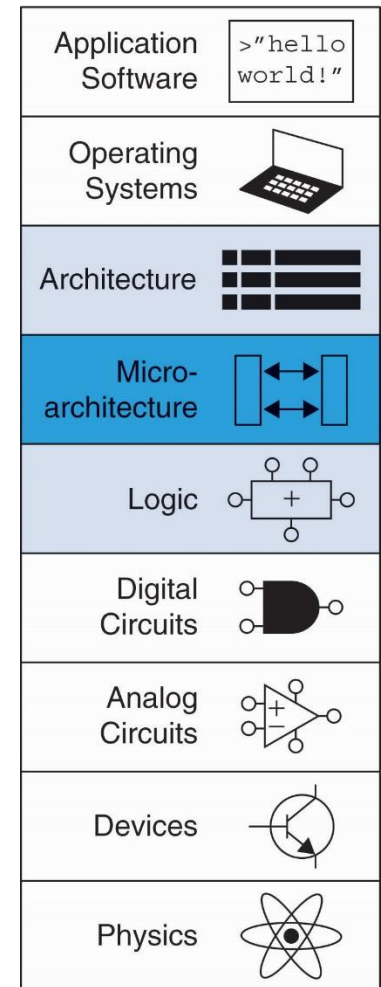
- Introduction
- Performance Analysis
- Single-Cycle Processor
- Multicycle Processor
- Pipelined Processor
- Advanced Microarchitecture

| | |
|----------------------|--|
| Application Software |  |
| Operating Systems |  |
| Architecture |  |
| Micro-architecture |  |
| Logic |  |
| Digital Circuits |  |
| Analog Circuits |  |
| Devices |  |
| Physics |  |



Introduction

- **Microarchitecture:** how to implement an architecture in hardware
- **Processor:**
 - **Datapath:** functional blocks
 - **Control:** control signals



Microarchitecture

- Multiple implementations for a single architecture:
 - **Single-cycle:** Each instruction executes in a single cycle
 - **Multicycle:** Each instruction is broken up into series of shorter steps
 - **Pipelined:** Each instruction broken up into series of steps & multiple instructions execute at once



Processor Performance

- Program execution time

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

- Definitions:
 - CPI: Cycles/instruction
 - clock period: seconds/cycle
 - IPC: instructions/cycle = IPC
- Challenge is to satisfy constraints of:
 - Cost
 - Power
 - Performance



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

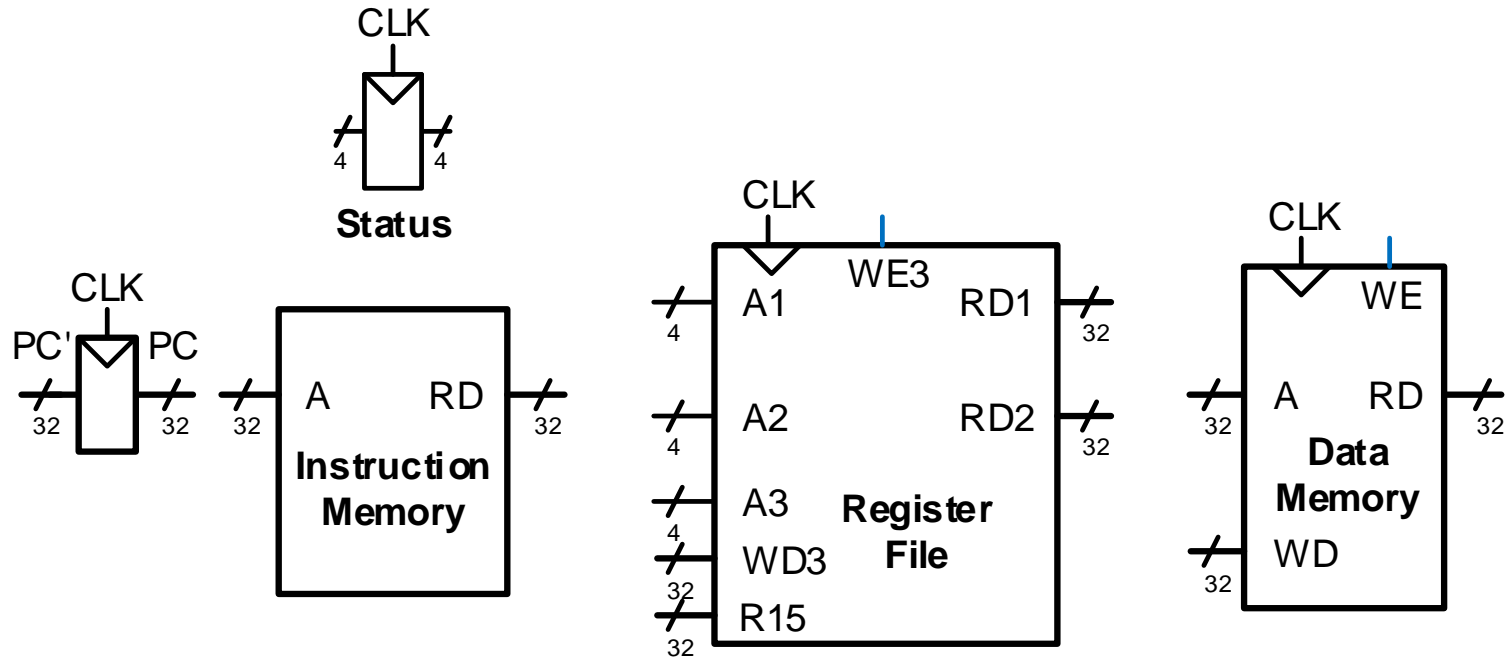


Architectural State Elements

- Determines everything about a processor:
 - Architectural state:
 - 16 registers (including PC)
 - Status register
 - Memory



ARM Architectural State Elements



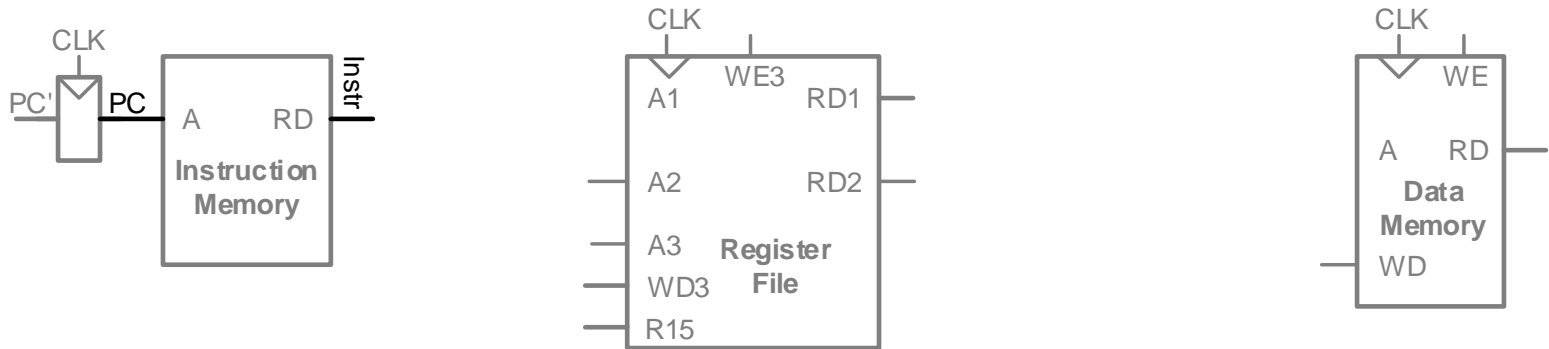
Single-Cycle ARM Processor

- Datapath
- Control



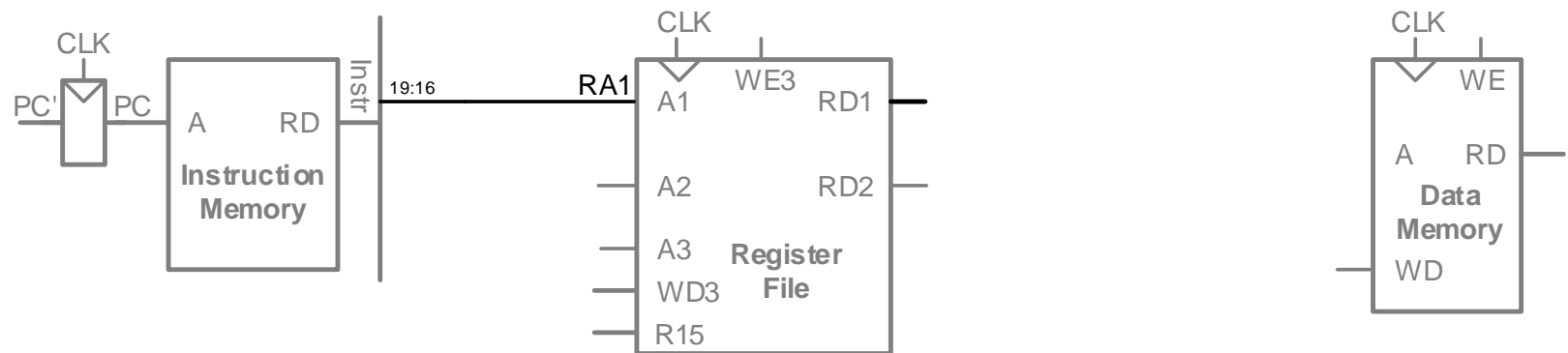
Single-Cycle Datapath: LDR fetch

STEP 1: Fetch instruction



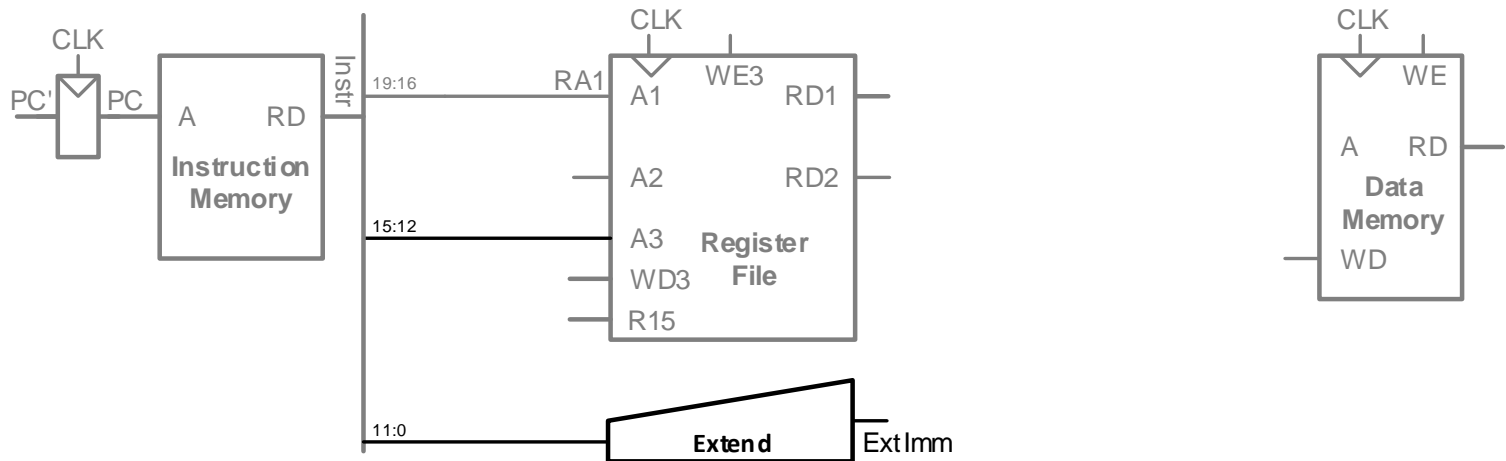
Single-Cycle Datapath: LDR Reg Read

STEP 2: Read source operands from RF



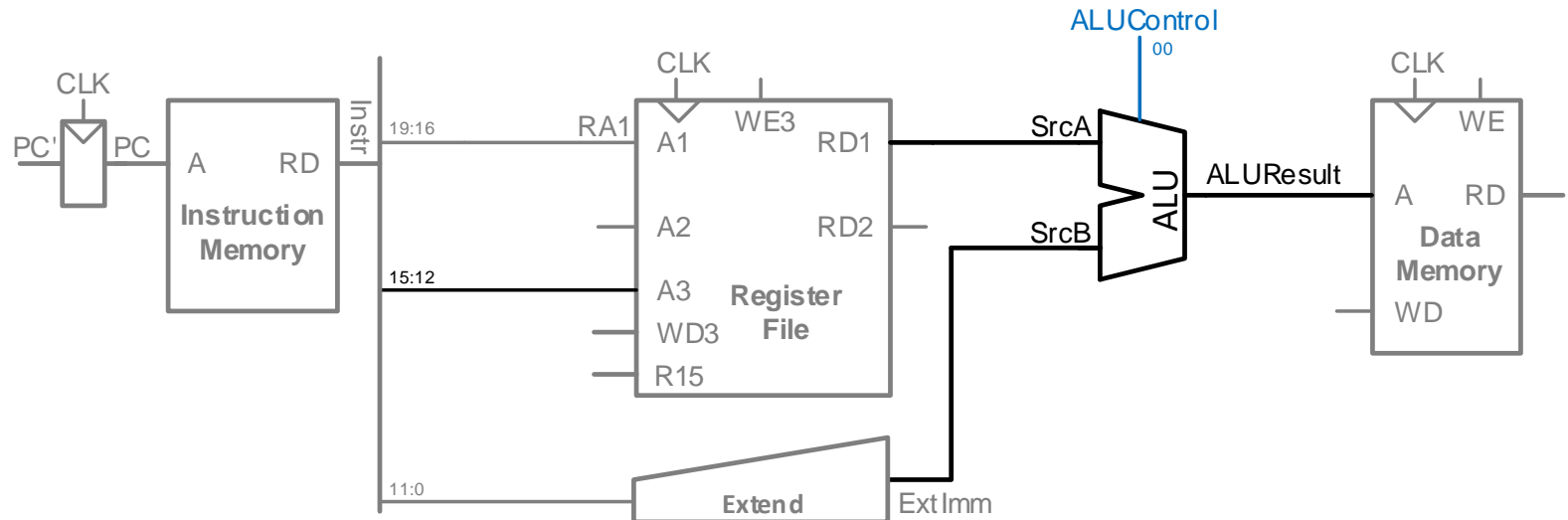
Single-Cycle Datapath: LDR Immed.

STEP 3: Extend the immediate



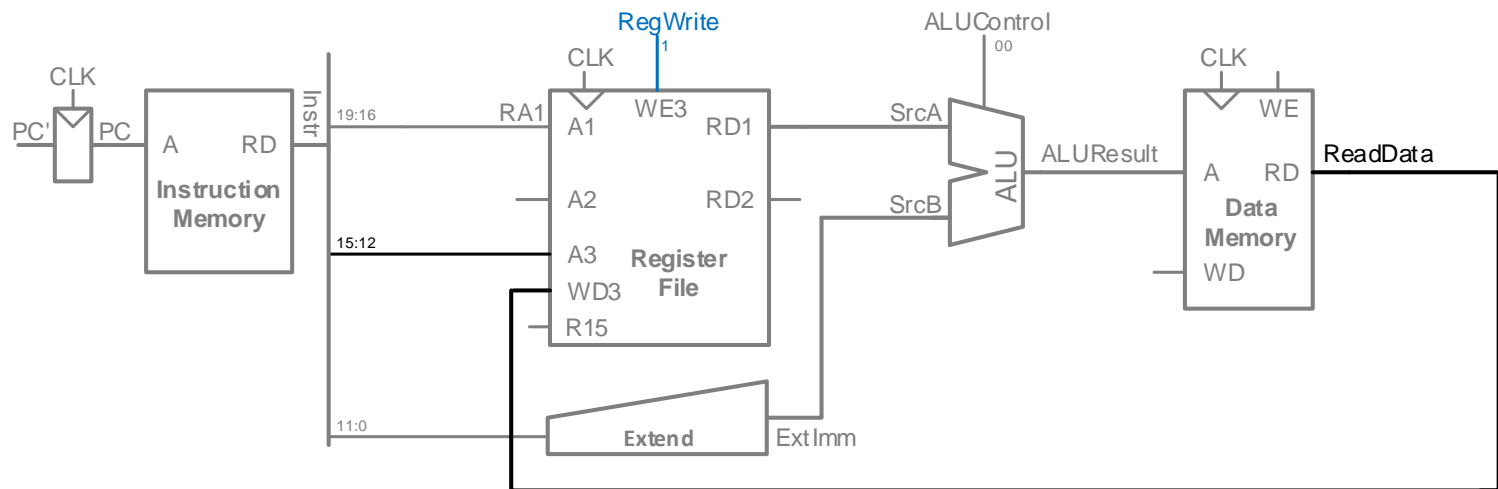
Single-Cycle Datapath: LDR Address

STEP 4: Compute the memory address



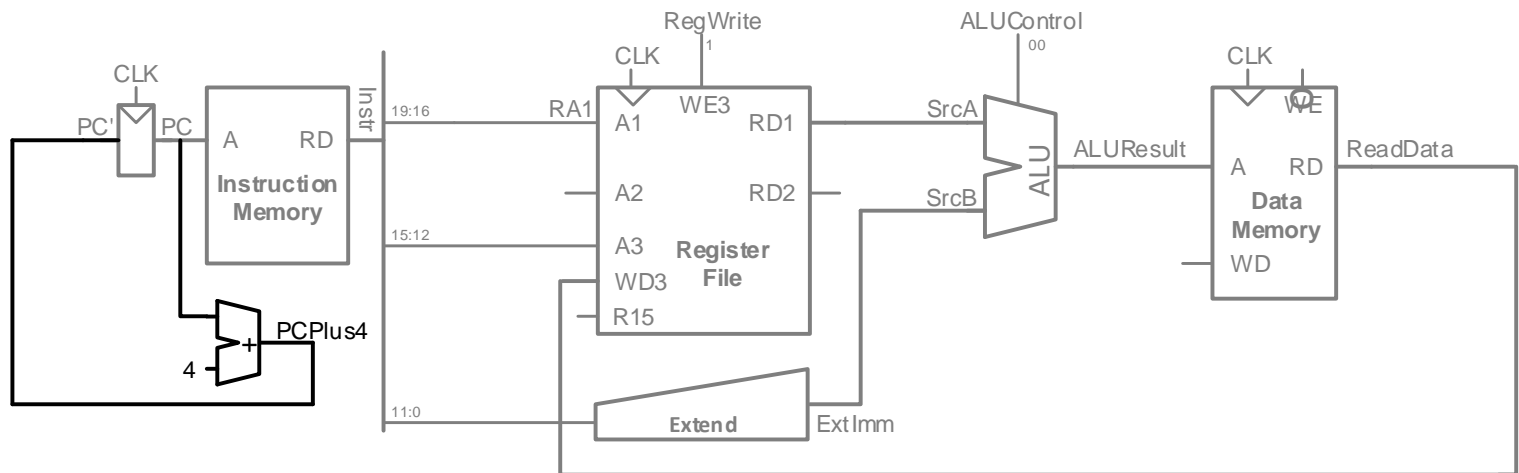
Single-Cycle Datapath: LDR Mem Read

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



Single-Cycle Datapath: PC Increment

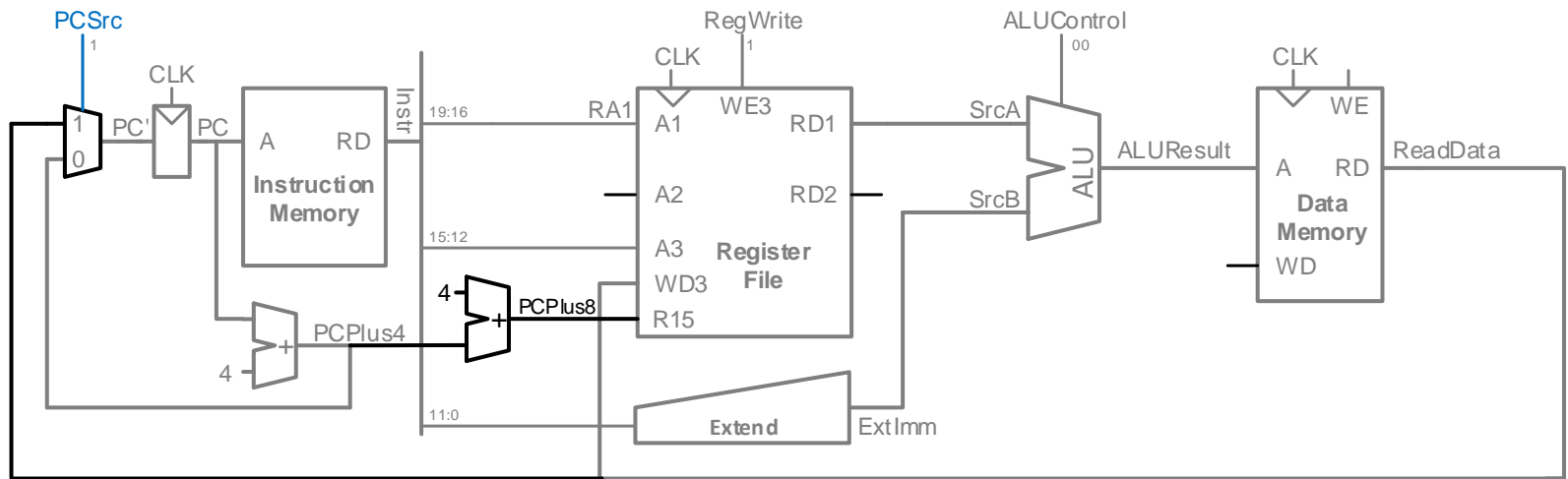
STEP 6: Determine address of next instruction



Single-Cycle Datapath: Access to PC

PC can be source/destination of instruction

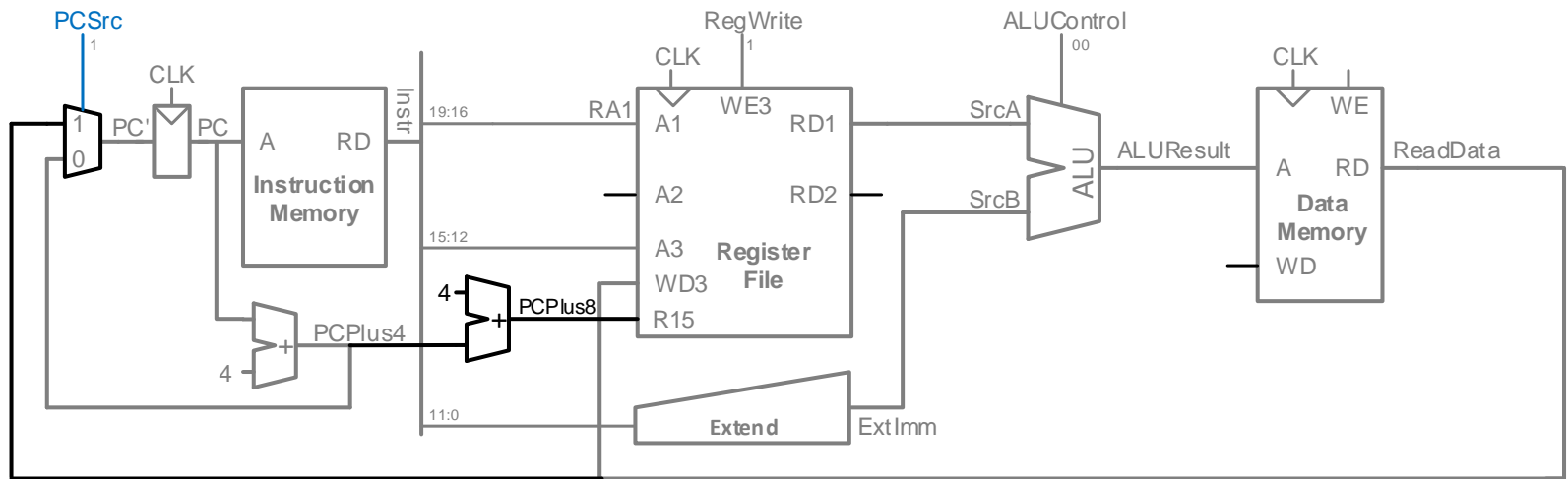
- **Source:** R15 (PC+8) available in Register File



Single-Cycle Datapath: Access to PC

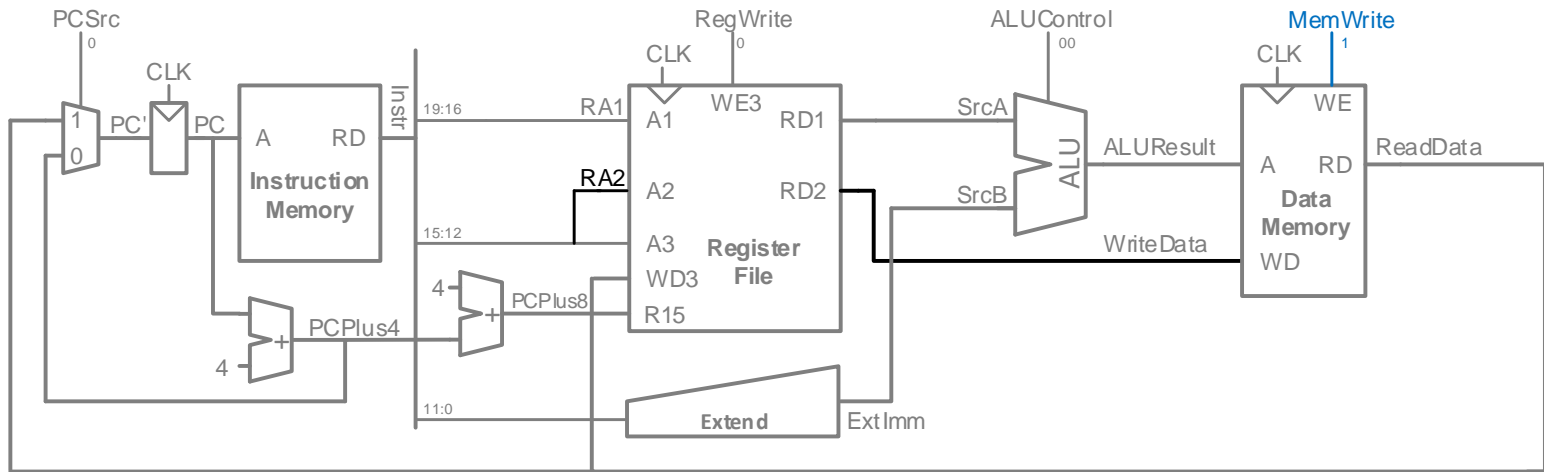
PC can be source/destination of instruction

- **Source:** R15 (PC+8) available in Register File
- **Destination:** Be able to write result to PC



Single-Cycle Datapath: STR

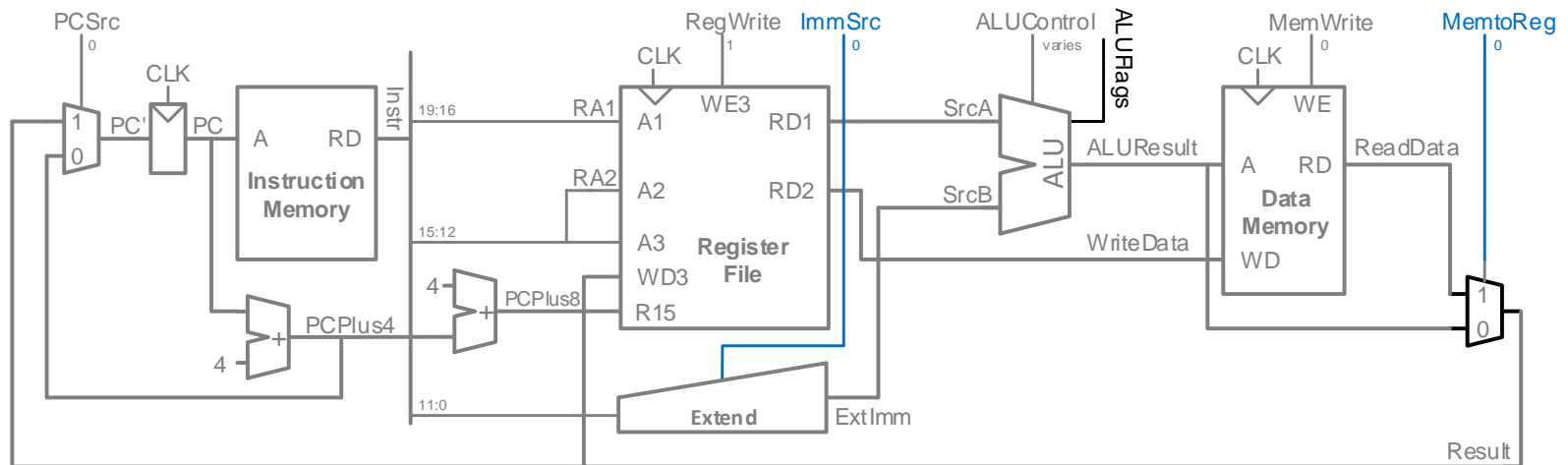
Write data in RD to memory



Single-Cycle Datapath: Data-processing

With immediate Src2:

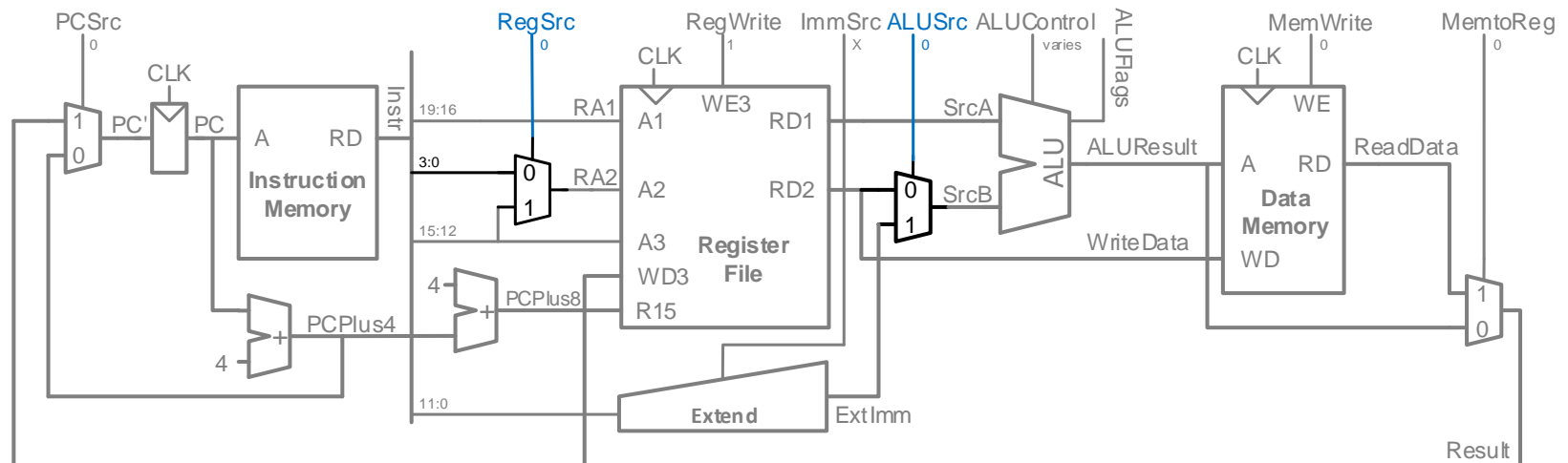
- Read from R_n and $Imm8$ ($ImmSrc$ chooses the zero-extended $Imm8$ instead of $Imm12$)
- Write $ALUResult$ to register file
- Write to R_d



Single-Cycle Datapath: Data-processing

With register Src2:

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

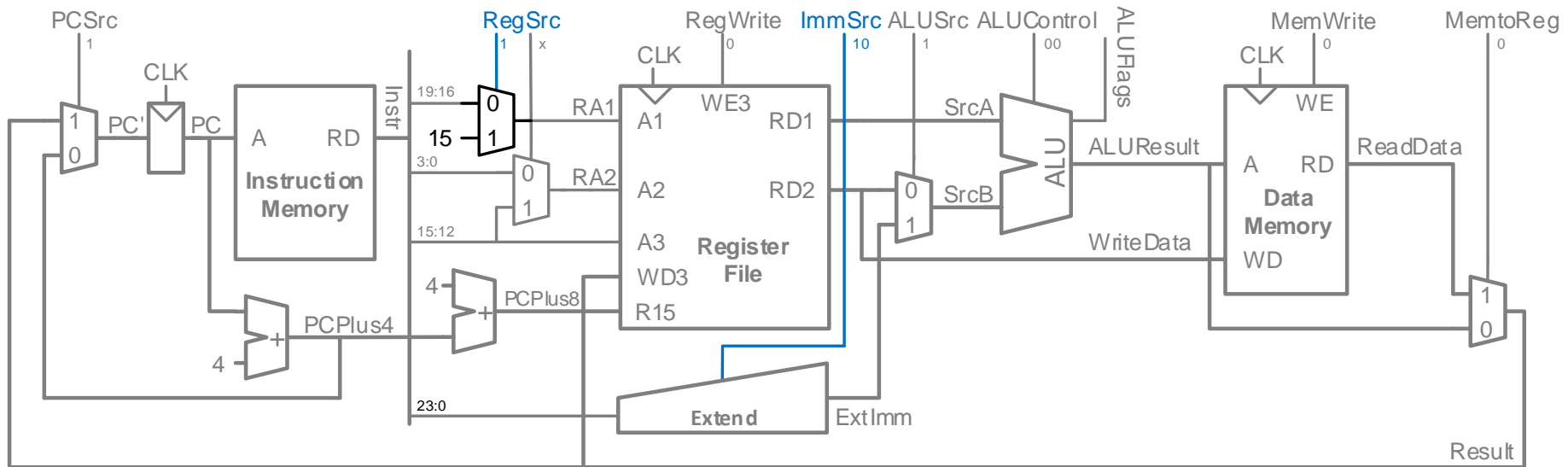


Single-Cycle Datapath: B

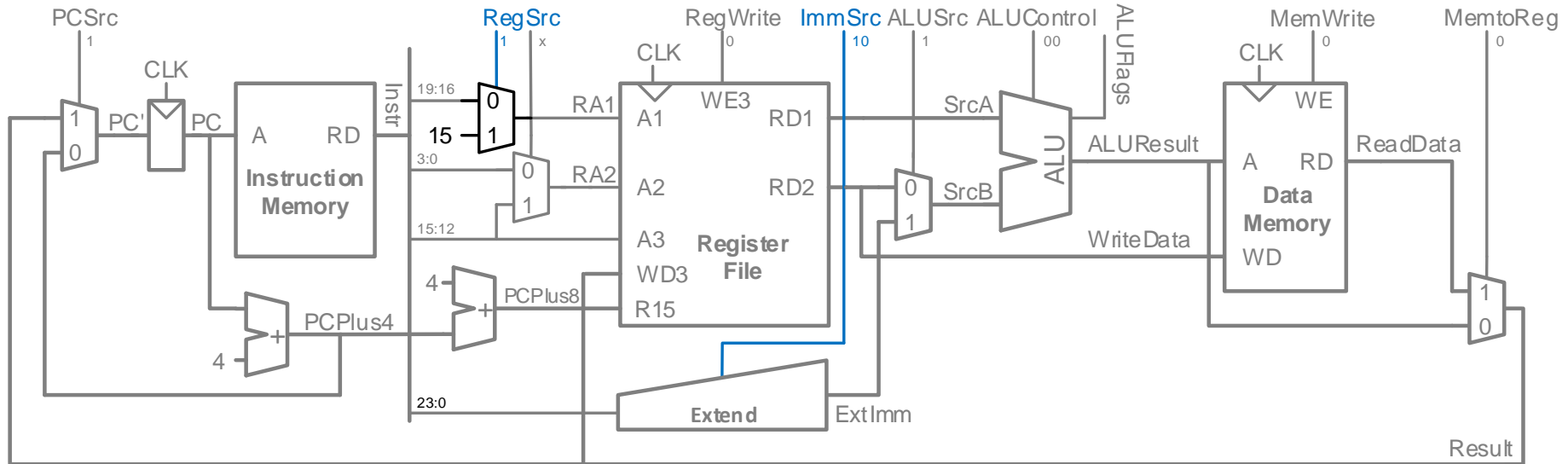
Calculate branch target address:

$$\text{BTA} = (\text{ExtImm}) + (\text{PC} + 8)$$

$\text{ExtImm} = \text{Imm24} \ll 2$ and sign-extended



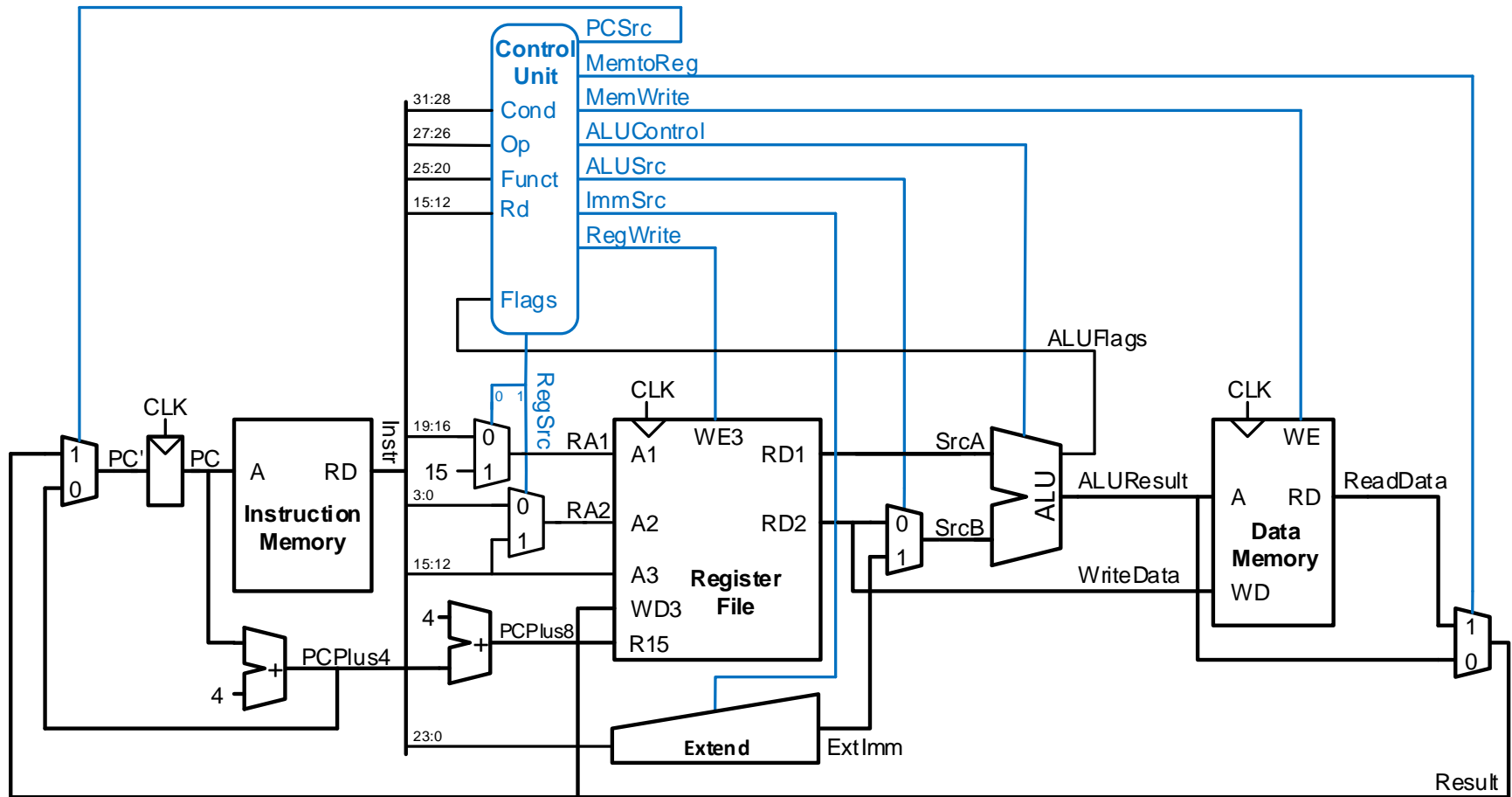
Single-Cycle Datapath: ExtImm



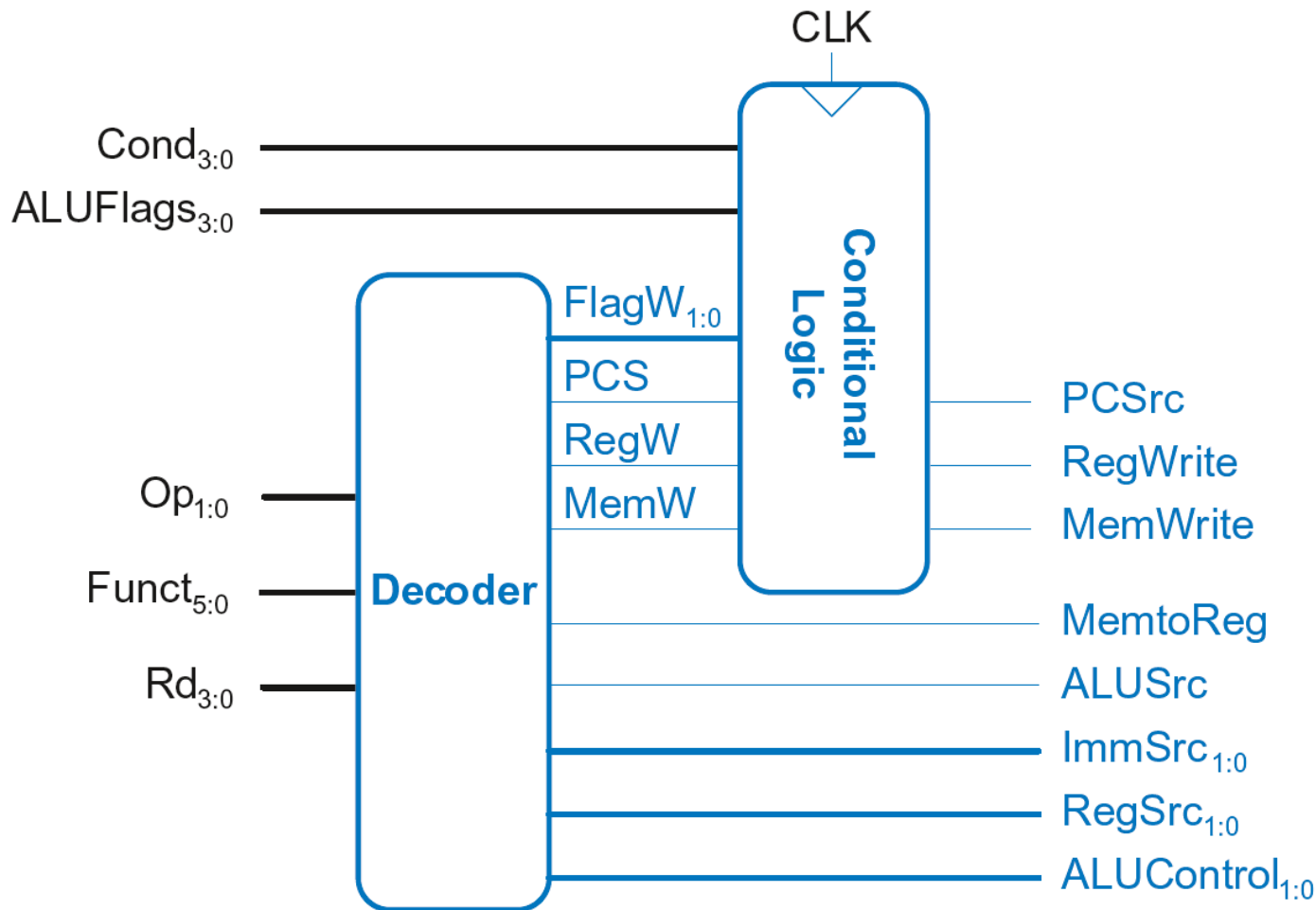
| ImmSrc _{1:0} | ExtImm | Description |
|-----------------------|---|----------------------------|
| 00 | {24'b0, Instr _{7:0} } | Zero-extended <i>imm8</i> |
| 01 | {20'b0, Instr _{11:0} } | Zero-extended <i>imm12</i> |
| 10 | {6{Instr ₂₃ }, Instr _{23:0} } | Sign-extended <i>imm24</i> |



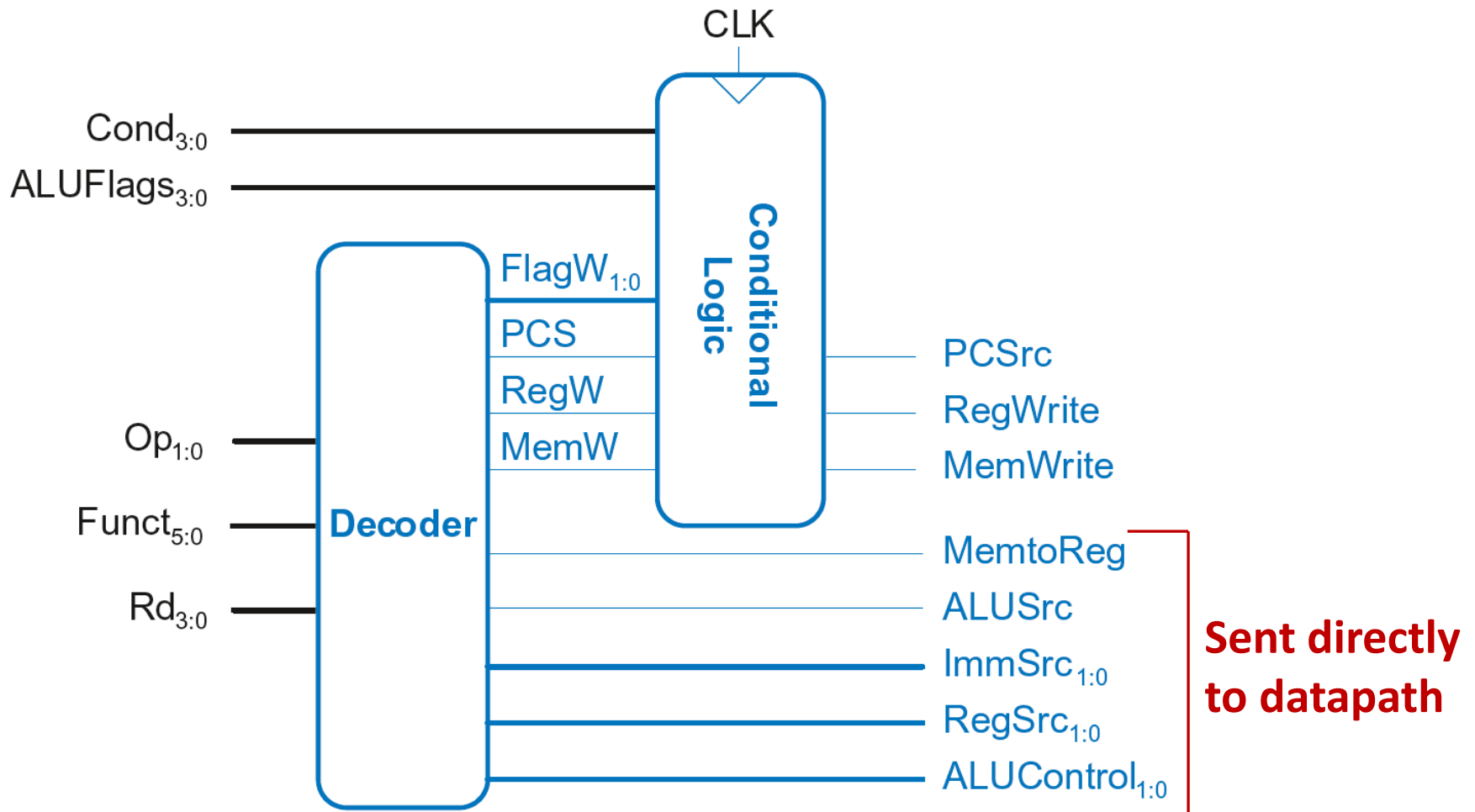
Single-Cycle ARM Processor



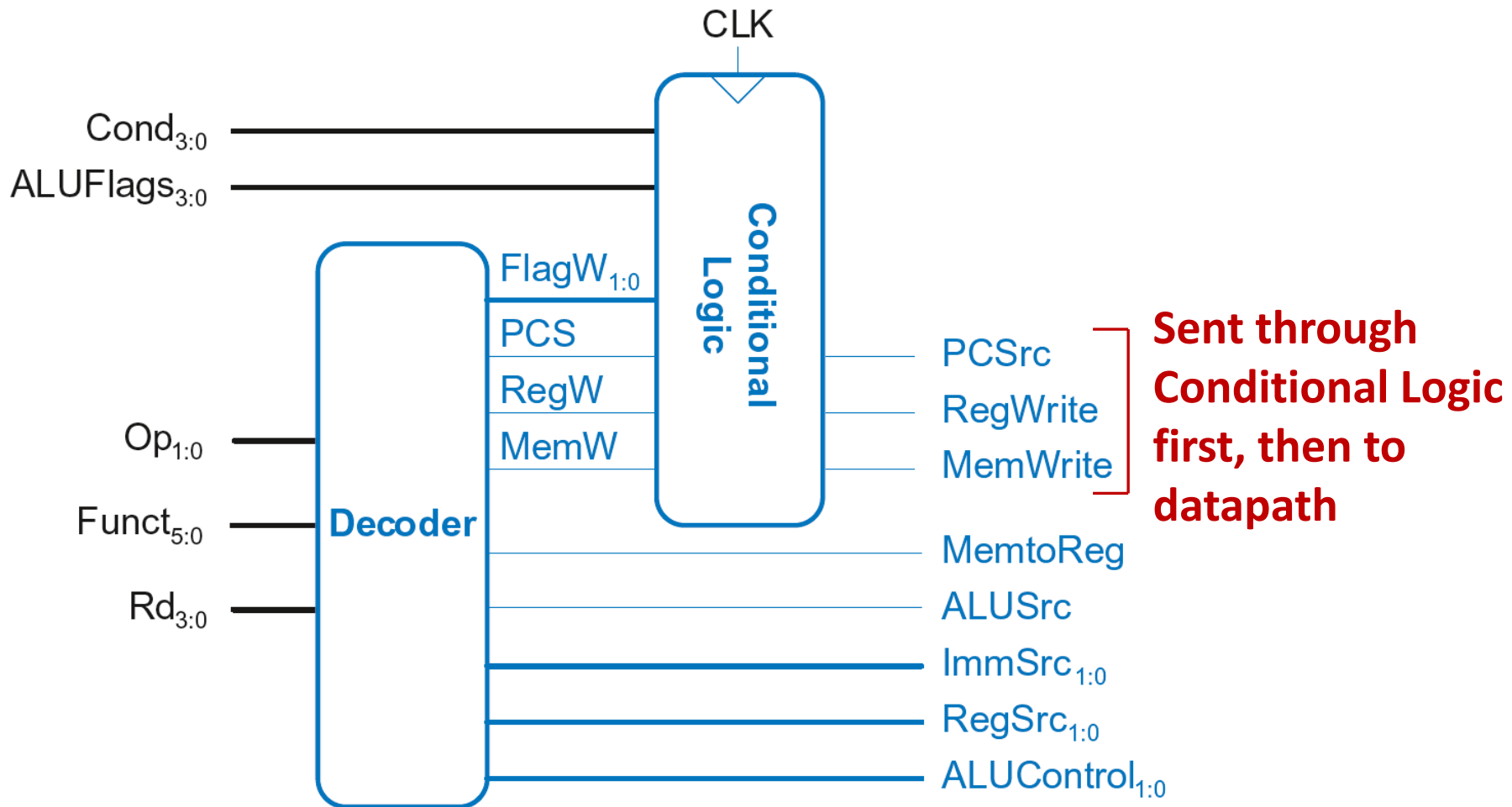
Single-Cycle Control



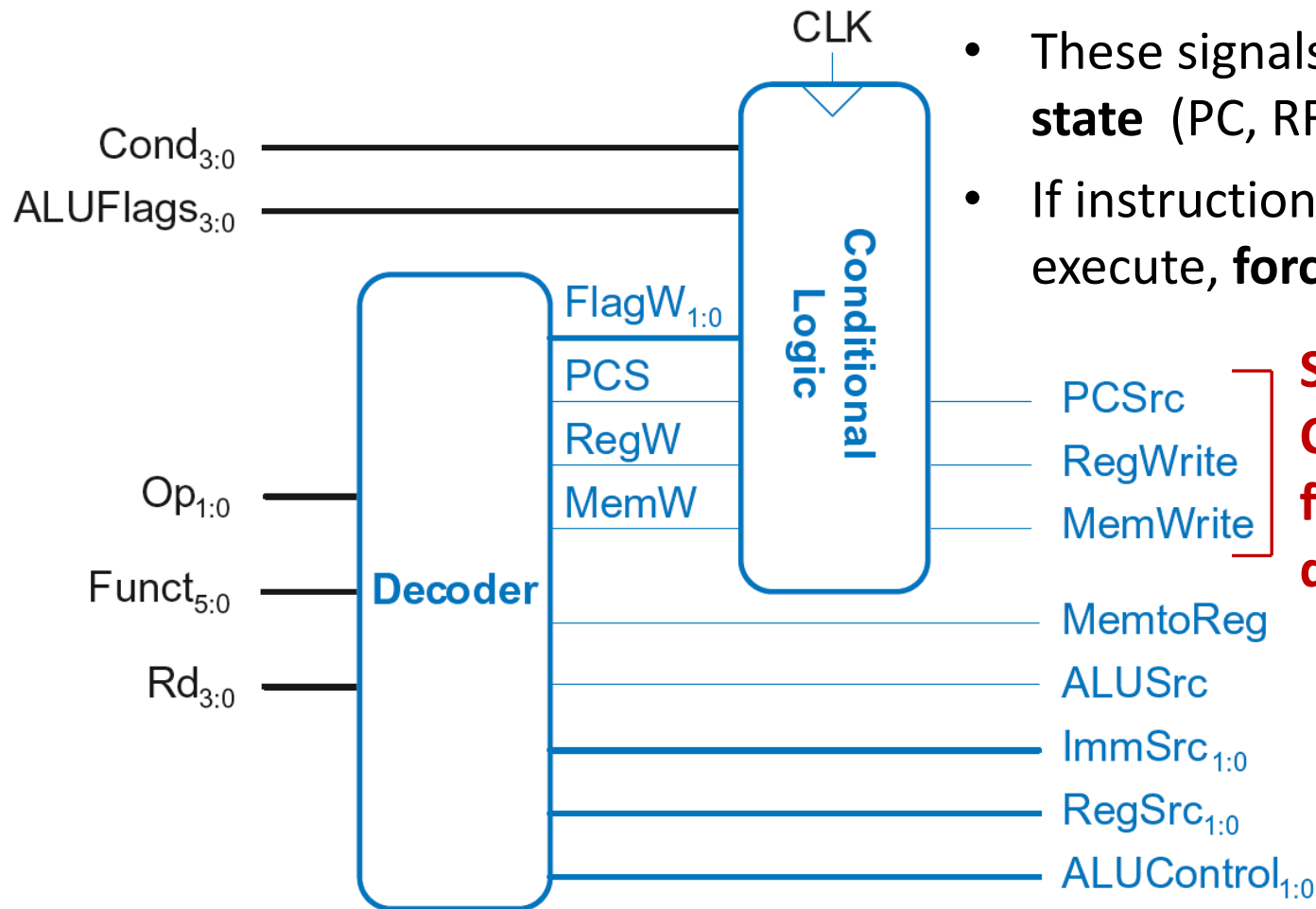
Single-Cycle Control



Single-Cycle Control



Single-Cycle Control

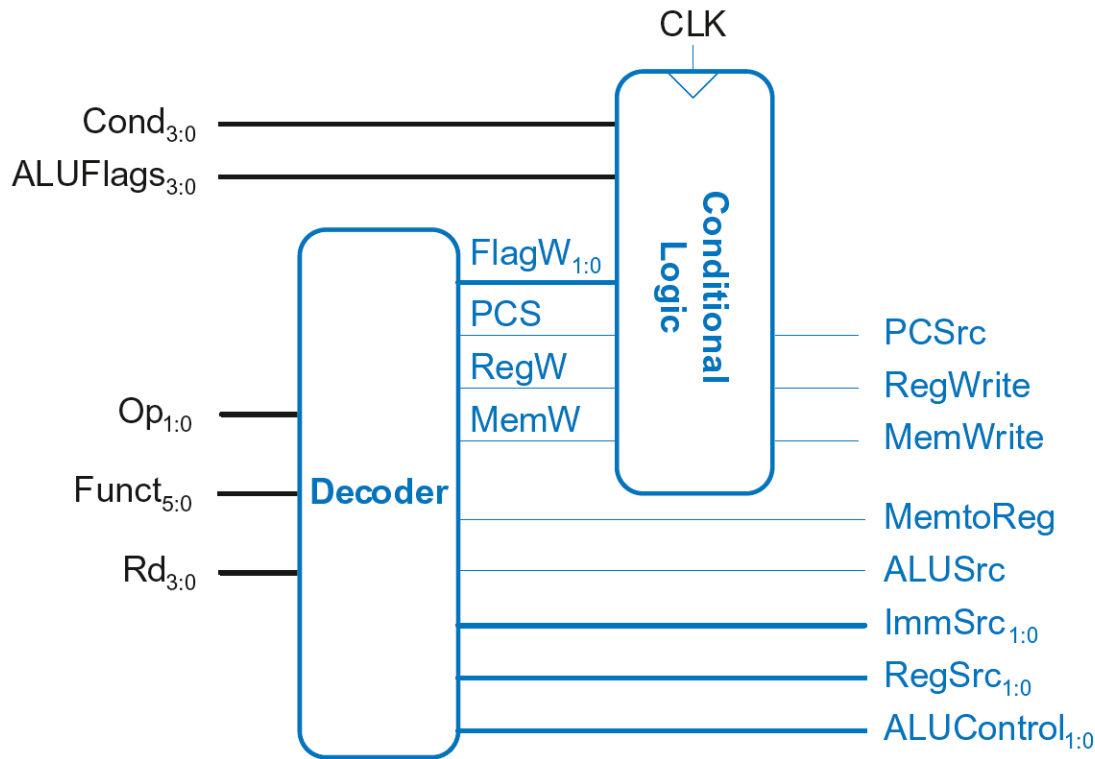


- These signals **change the state** (PC, RF, Memory)
- If instruction shouldn't execute, **forced to 0**

Sent through Conditional Logic first, then to datapath



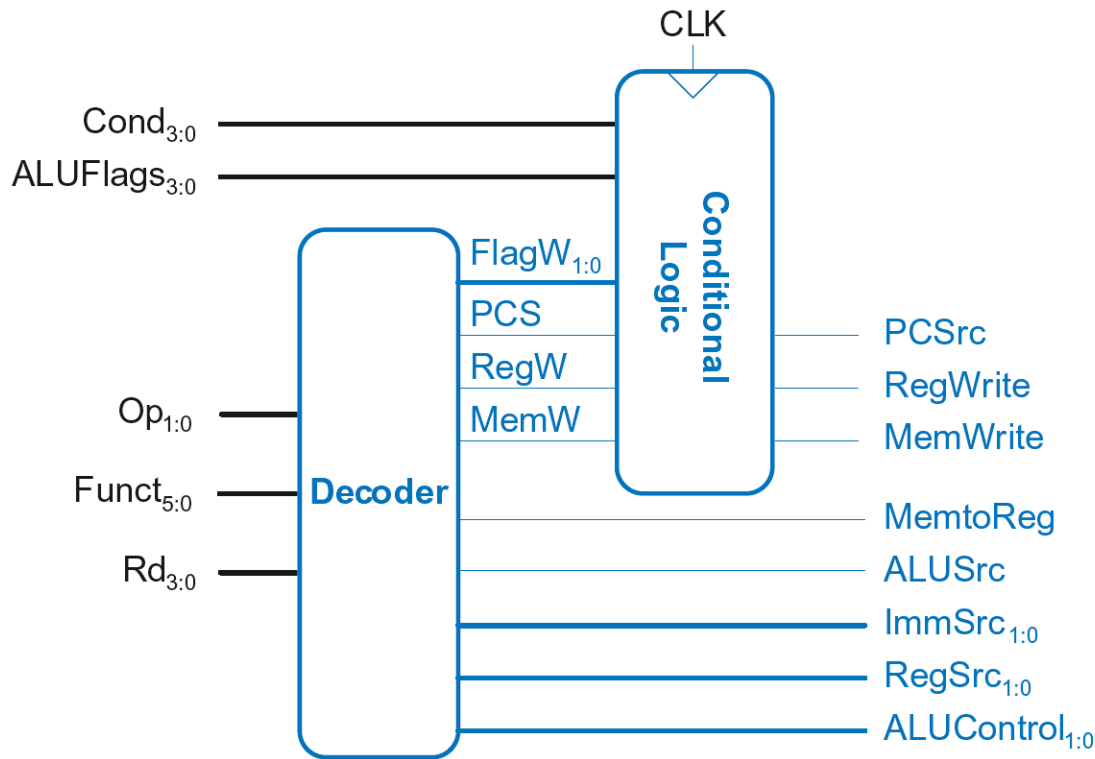
Single-Cycle Control



- **$FlagW_{1:0}$:** Flag Write signal, asserted when *ALUFlags* should be saved (i.e., on instruction with $S=1$)



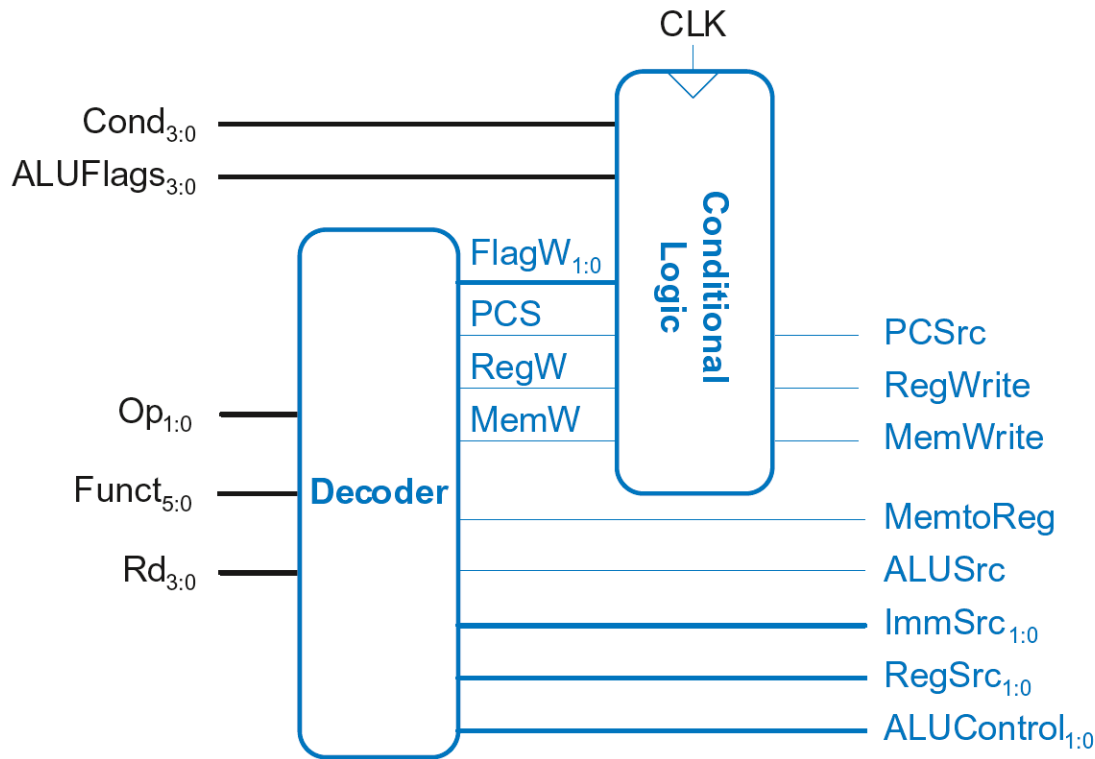
Single-Cycle Control



- **$FlagW_{1:0}$** : 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



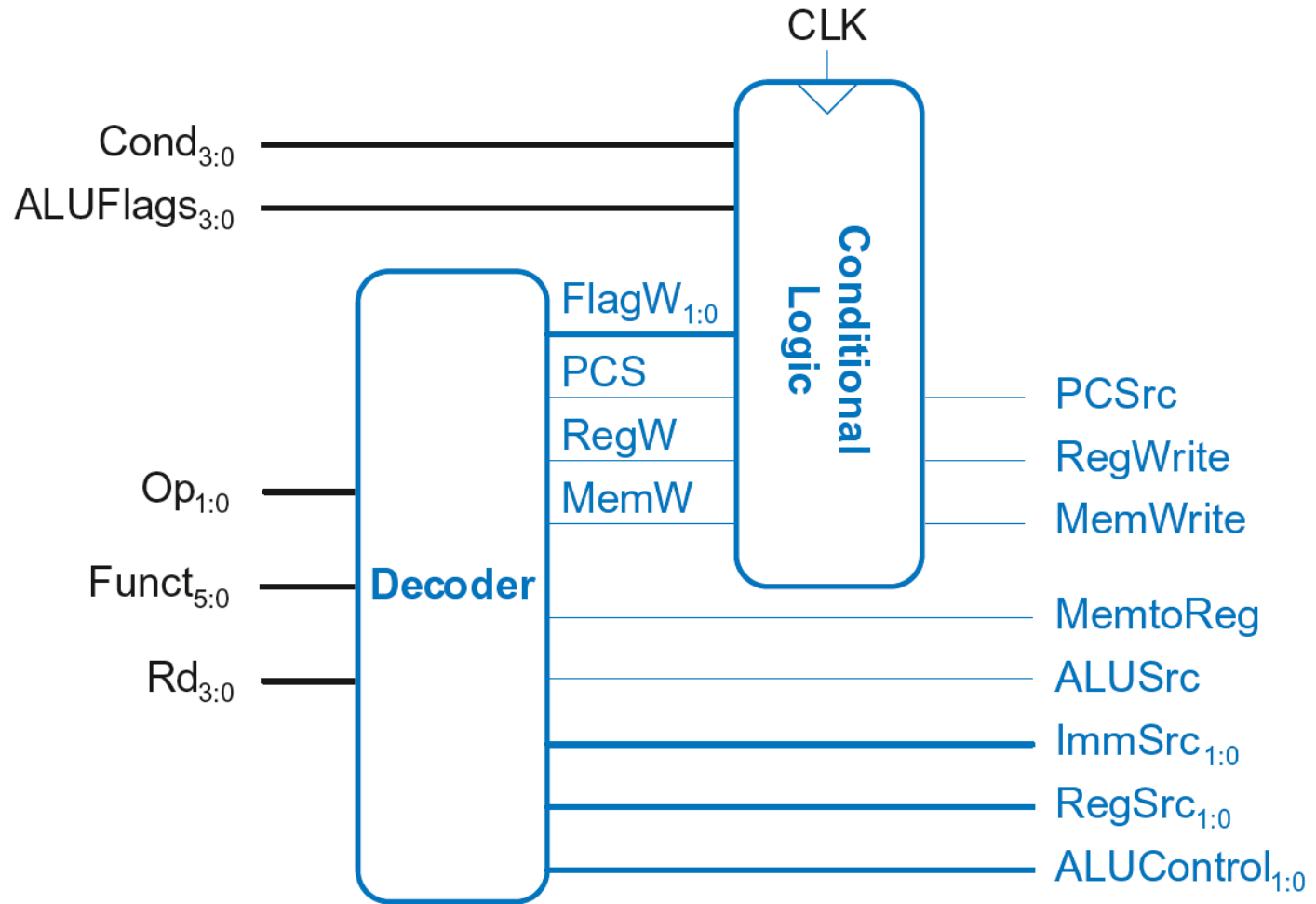
Single-Cycle Control



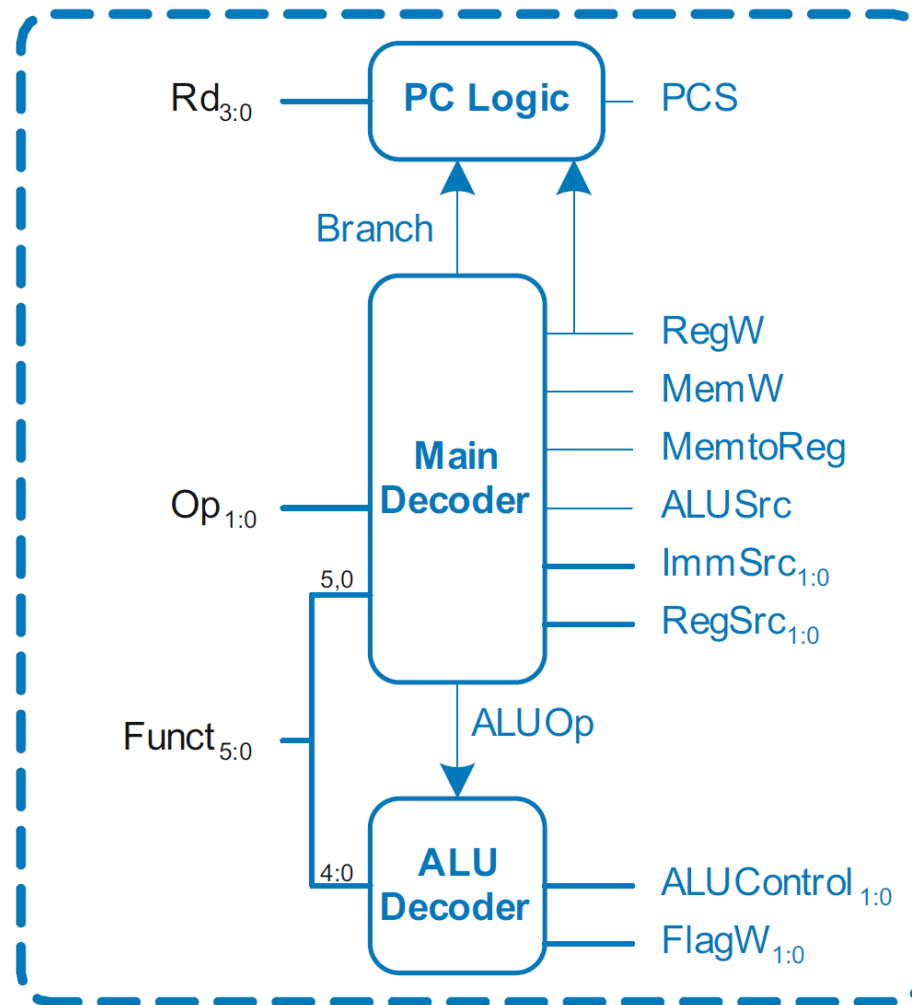
- **$FlagW_{1:0}$** : Flag Write signal, asserted when *ALUFlags* should be saved (i.e., on instruction with $S=1$)
- ADD, SUB update all flags (**NZCV**)
- AND, ORR only update **NZ** flags
- So, two bits needed:
 - $FlagW_1 = 1$** : NZ saved (*ALUFlags*_{3:2} saved)
 - $FlagW_0 = 1$** : CV saved (*ALUFlags*_{1:0} saved)



Single-Cycle Control: Decoder



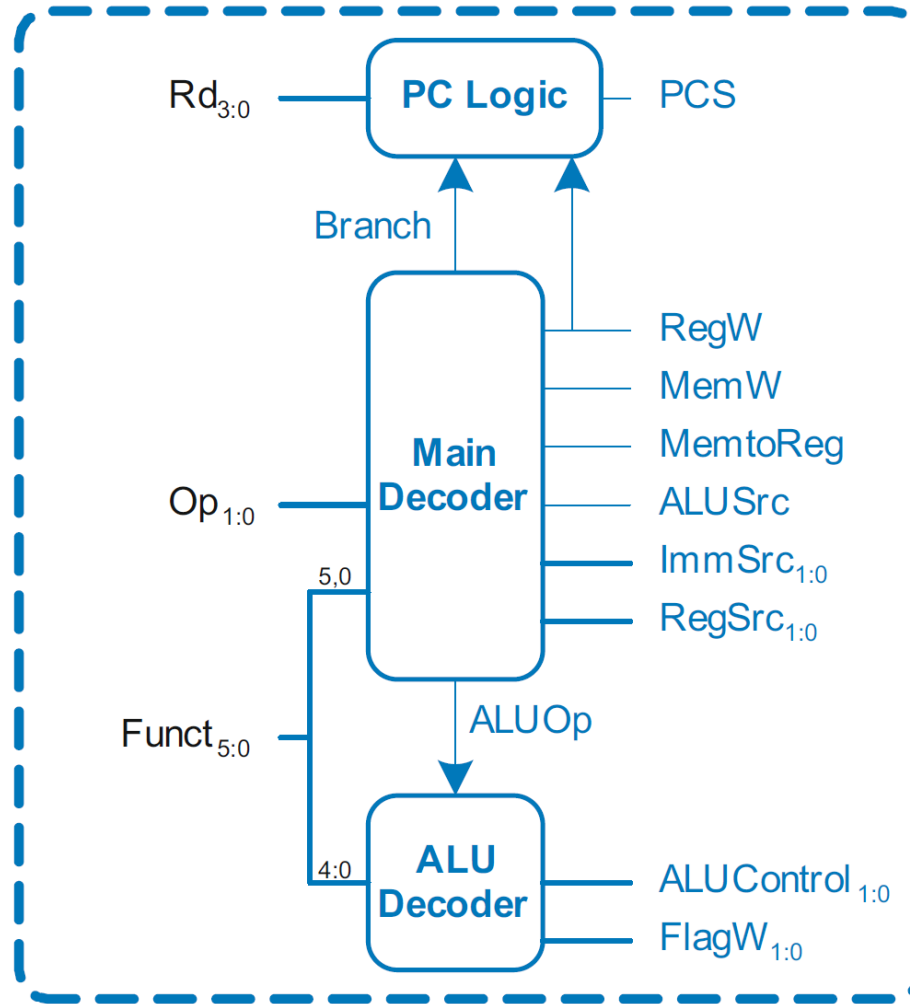
Single-Cycle Control: Decoder



Single-Cycle Control: Decoder

Submodules:

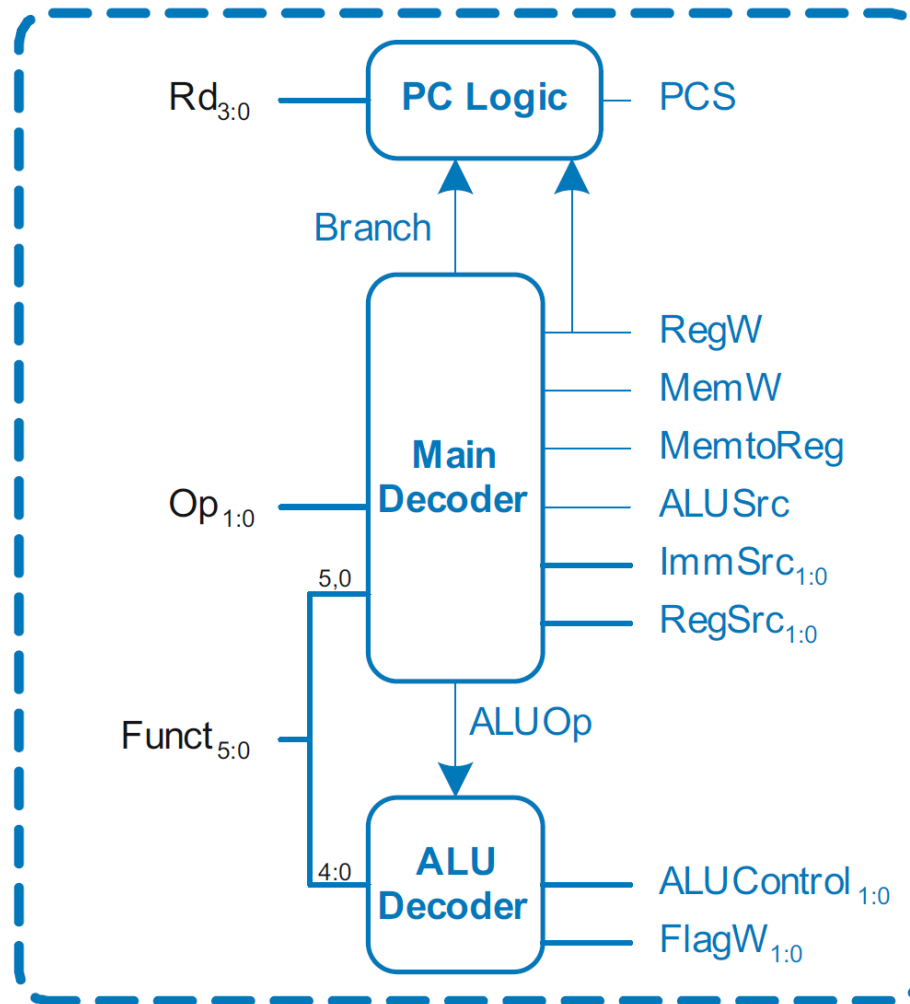
- Main Decoder
- ALU Decoder
- PC Logic



Single-Cycle Control: Decoder

Submodules:

- **Main Decoder**
- ALU Decoder
- PC Logic



Control Unit: Main Decoder

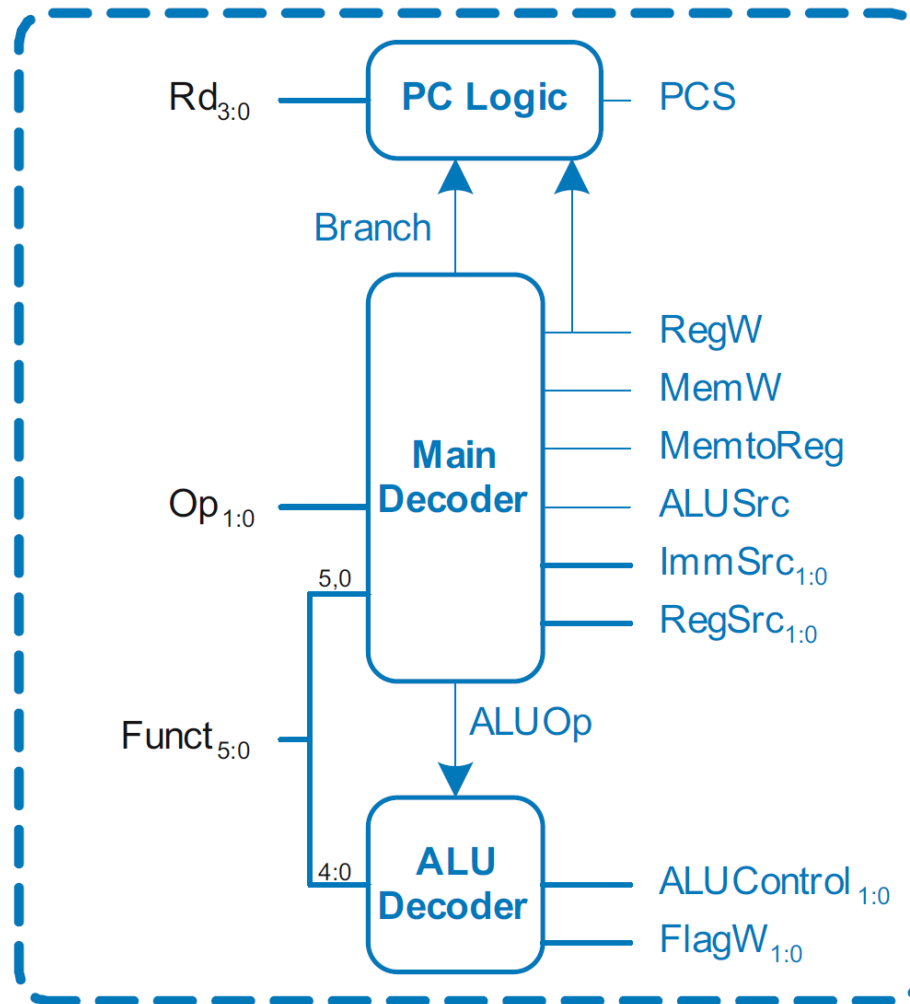
| Op | Funct ₅ | Funct ₀ | Type | Branch | MentoReg | MemW | ALUSrc | ImmSrc | RegW | RegSrc | ALUOp |
|----|--------------------|--------------------|--------|--------|----------|------|--------|--------|------|--------|-------|
| 00 | 0 | X | DP Reg | 0 | 0 | 0 | 0 | XX | 1 | 00 | 1 |
| 00 | 1 | X | 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

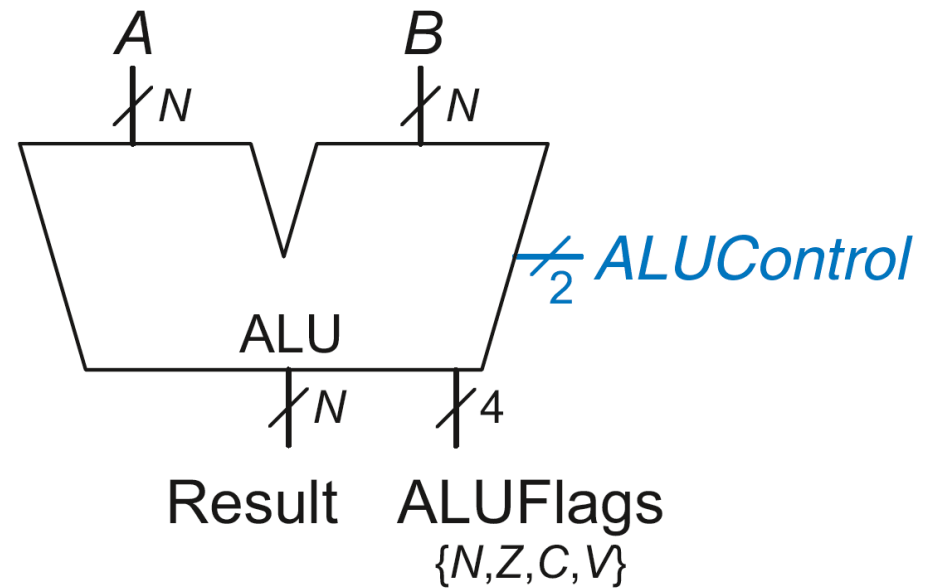
Submodules:

- Main Decoder
- **ALU Decoder**
- PC Logic

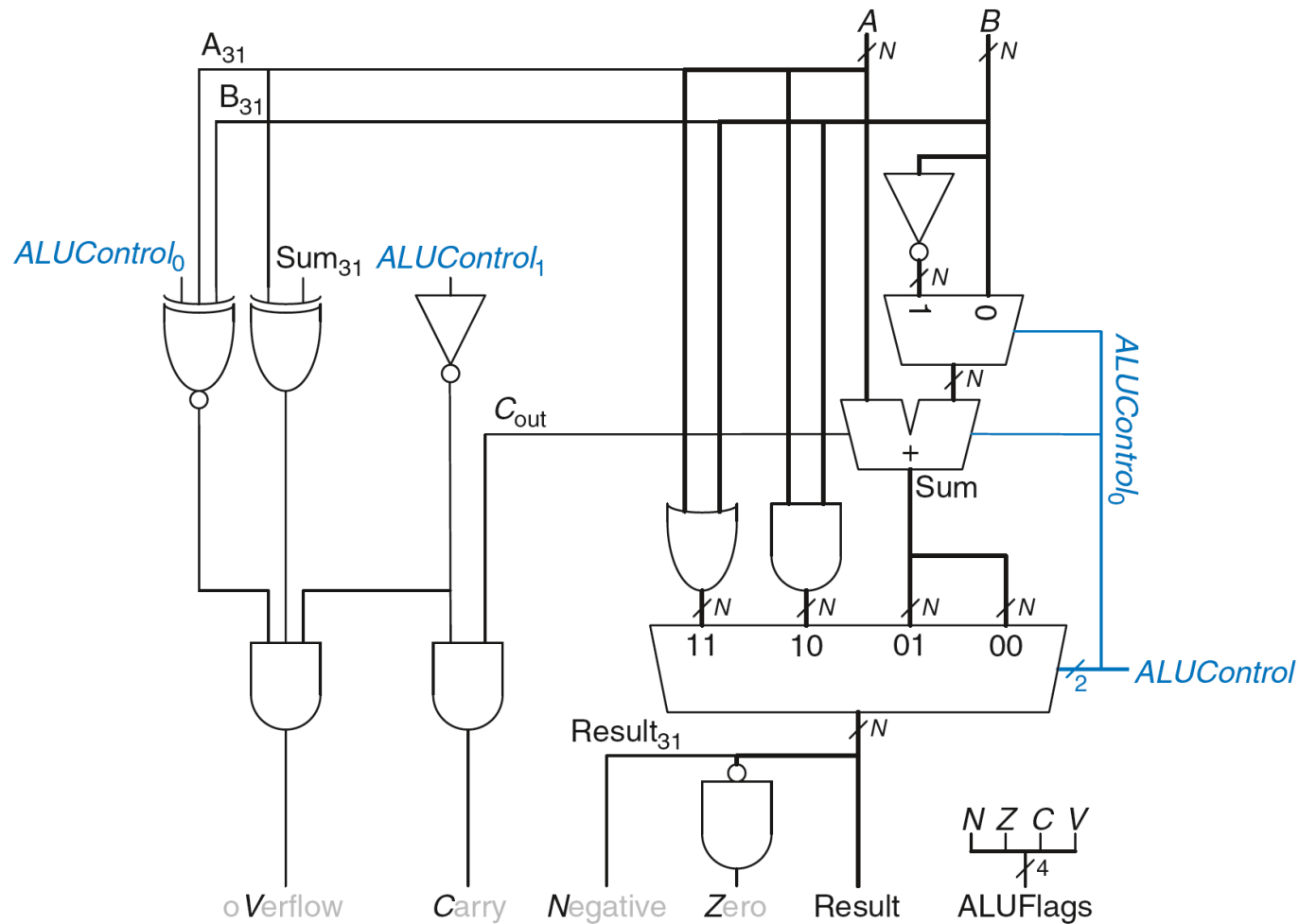


Review: ALU

| ALUControl _{1:0} | Function |
|---------------------------|----------|
| 00 | Add |
| 01 | Subtract |
| 10 | AND |
| 11 | OR |



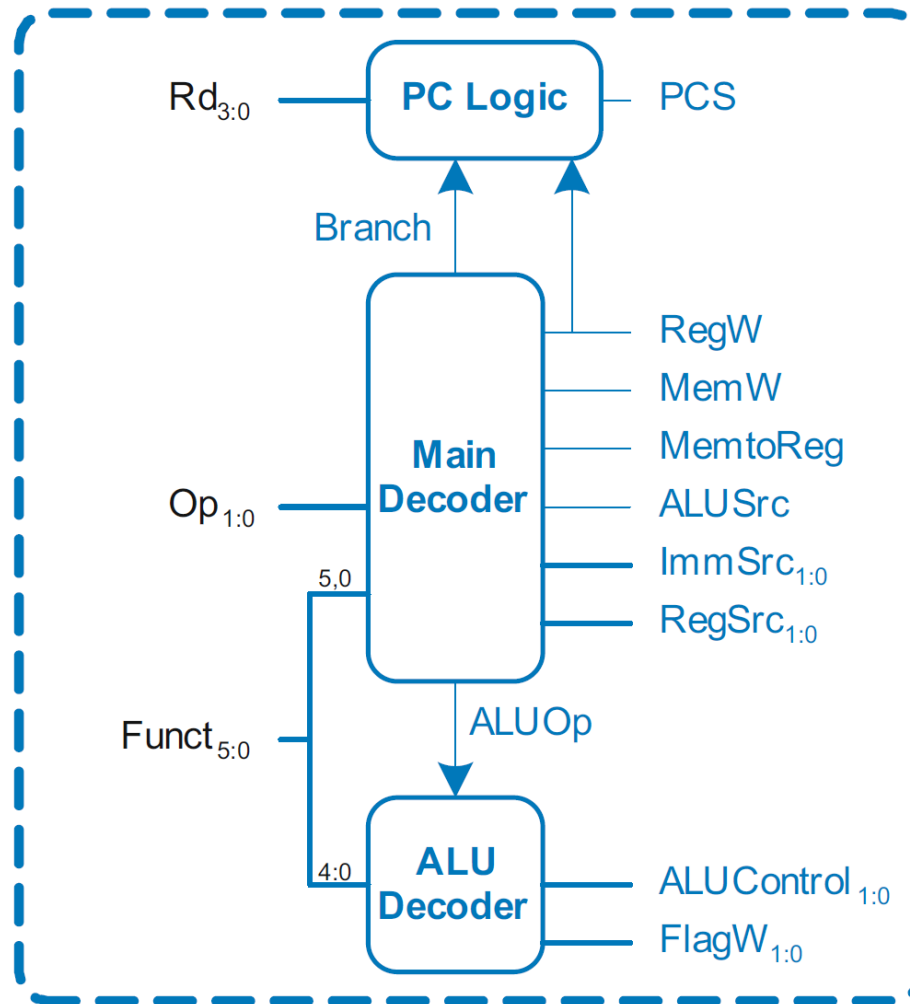
Review: ALU



Single-Cycle Control: Decoder

Submodules:

- Main Decoder
- **ALU Decoder**
- PC Logic



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 |

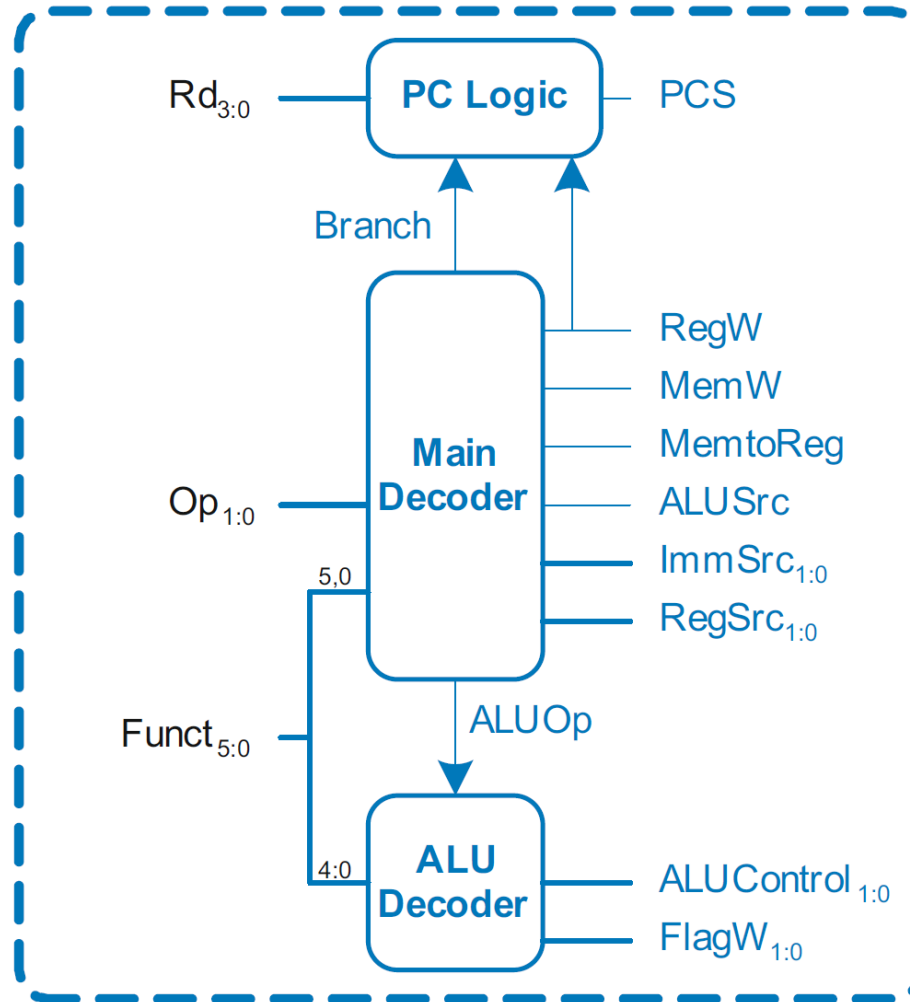
- $FlagW_1 = 1$: NZ ($Flags_{3:2}$) should be saved
- $FlagW_0 = 1$: CV ($Flags_{1:0}$) should be saved



Single-Cycle Control: Decoder

Submodules:

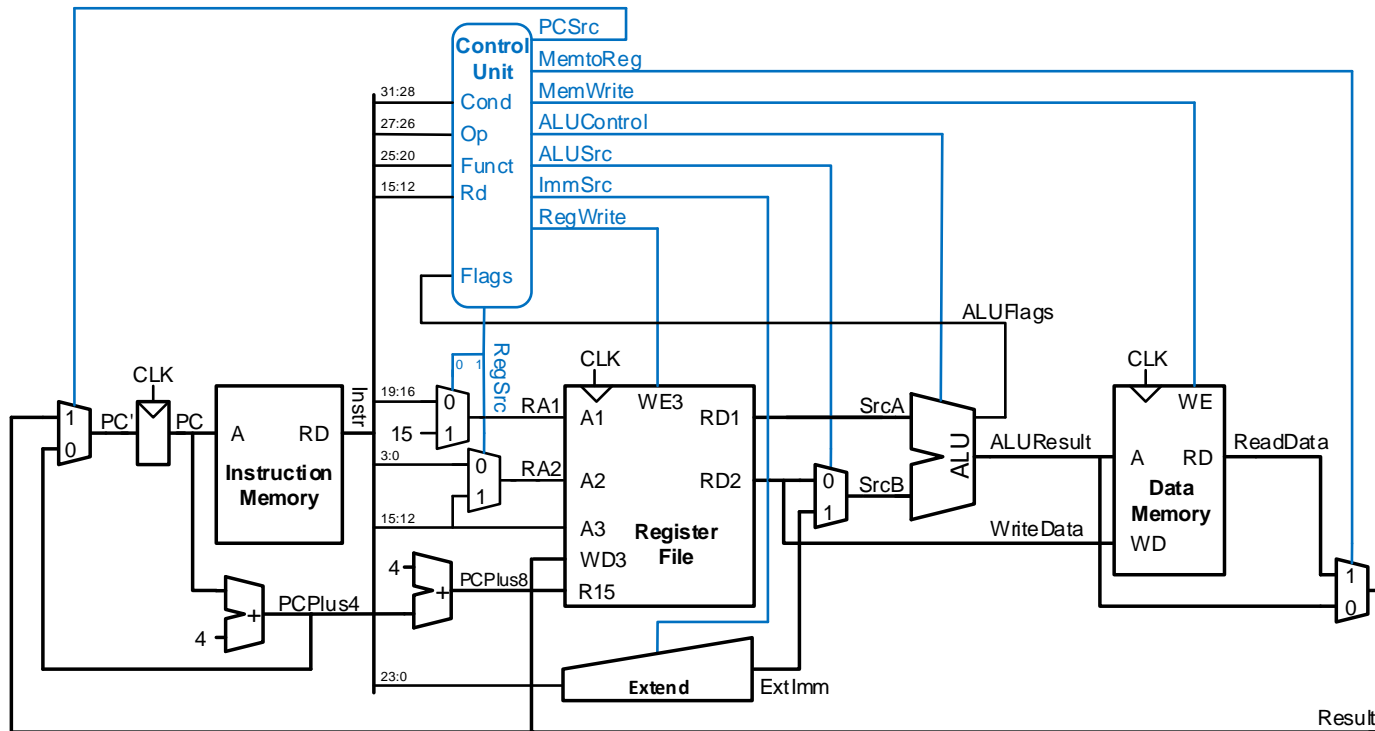
- Main Decoder
- ALU Decoder
- **PC Logic**



Single-Cycle Control: PC Logic

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

$$PCSrc = ((Rd == 15) \& RegW) \mid Branch$$



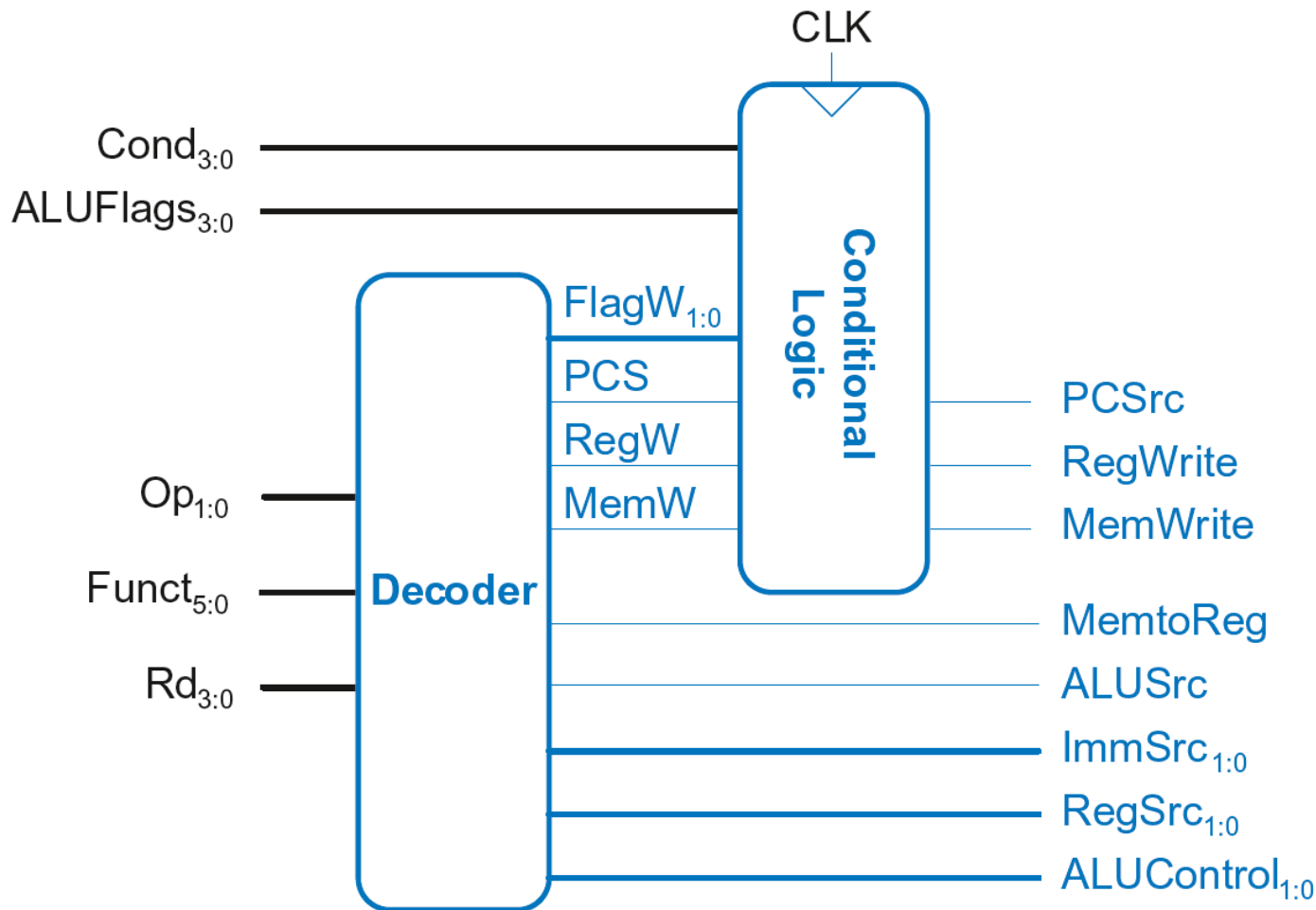
If instruction is executed: *PCSrc = PCS*

Else

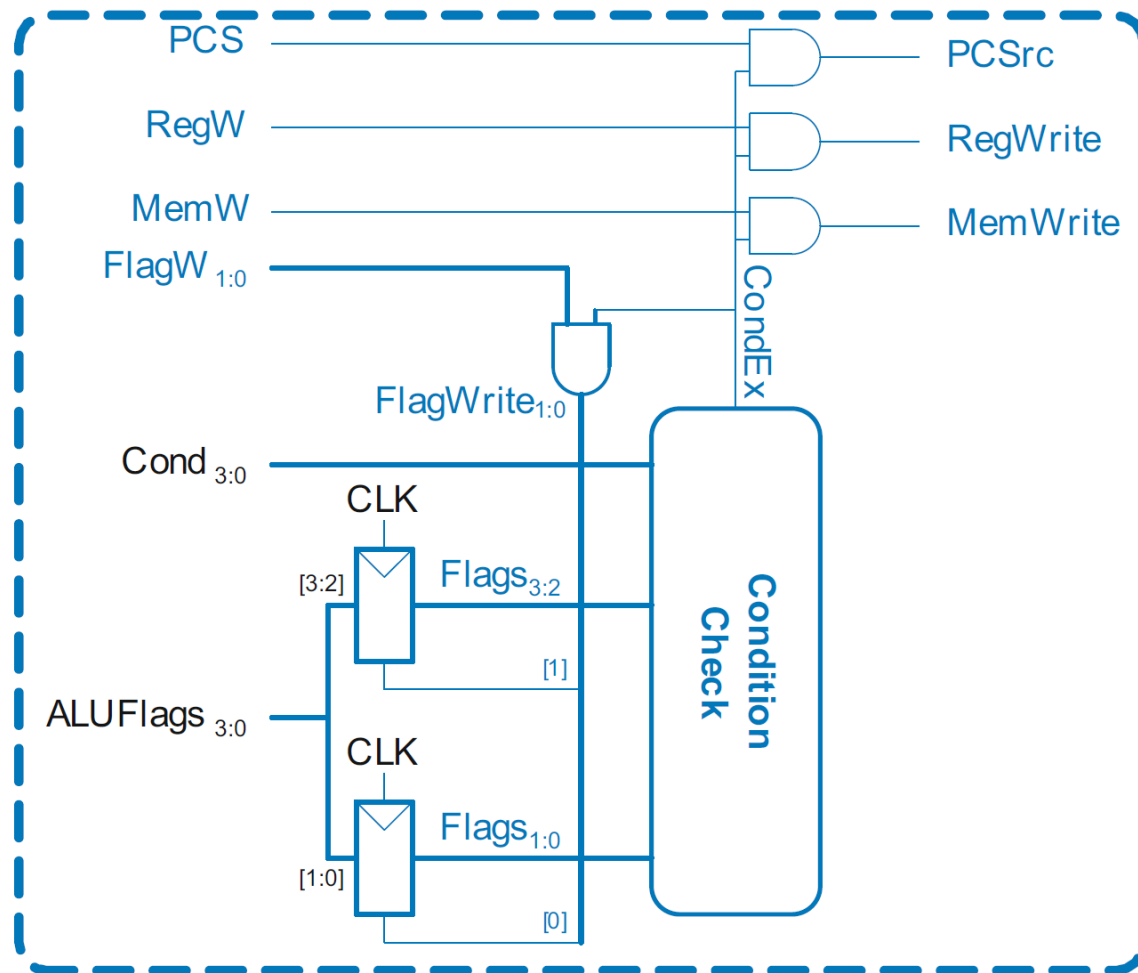
***PCSrc = 0* (i.e., $PC = PC + 4$)**



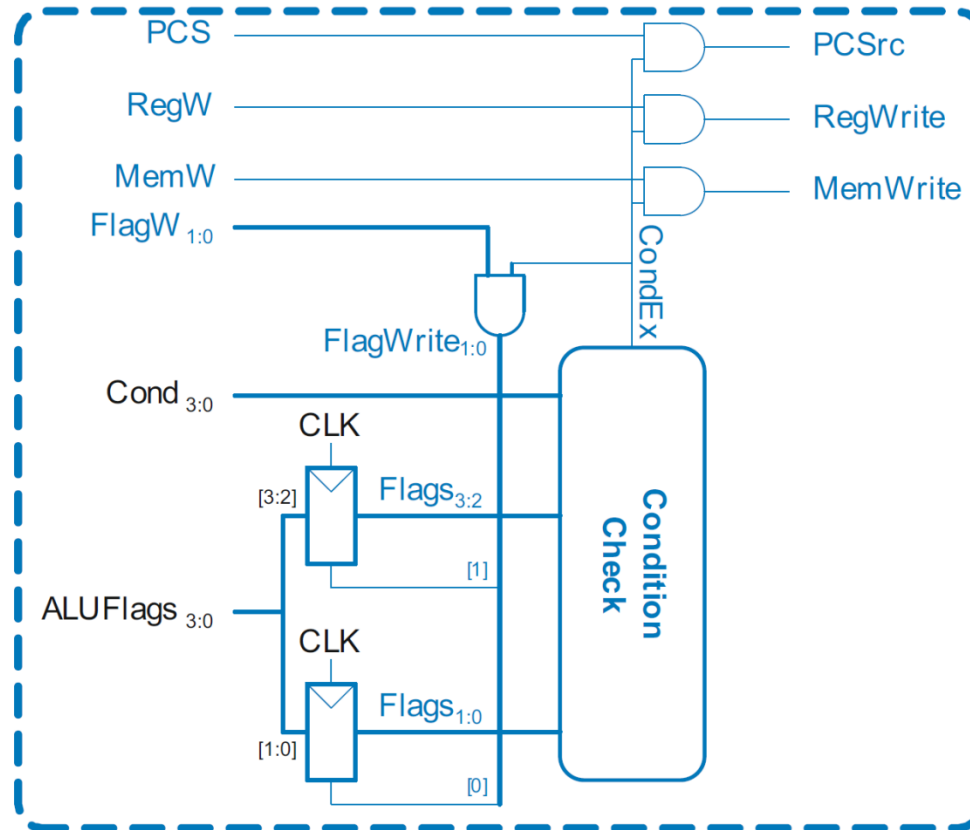
Single-Cycle Control



Single-Cycle Control: Conditional Logic



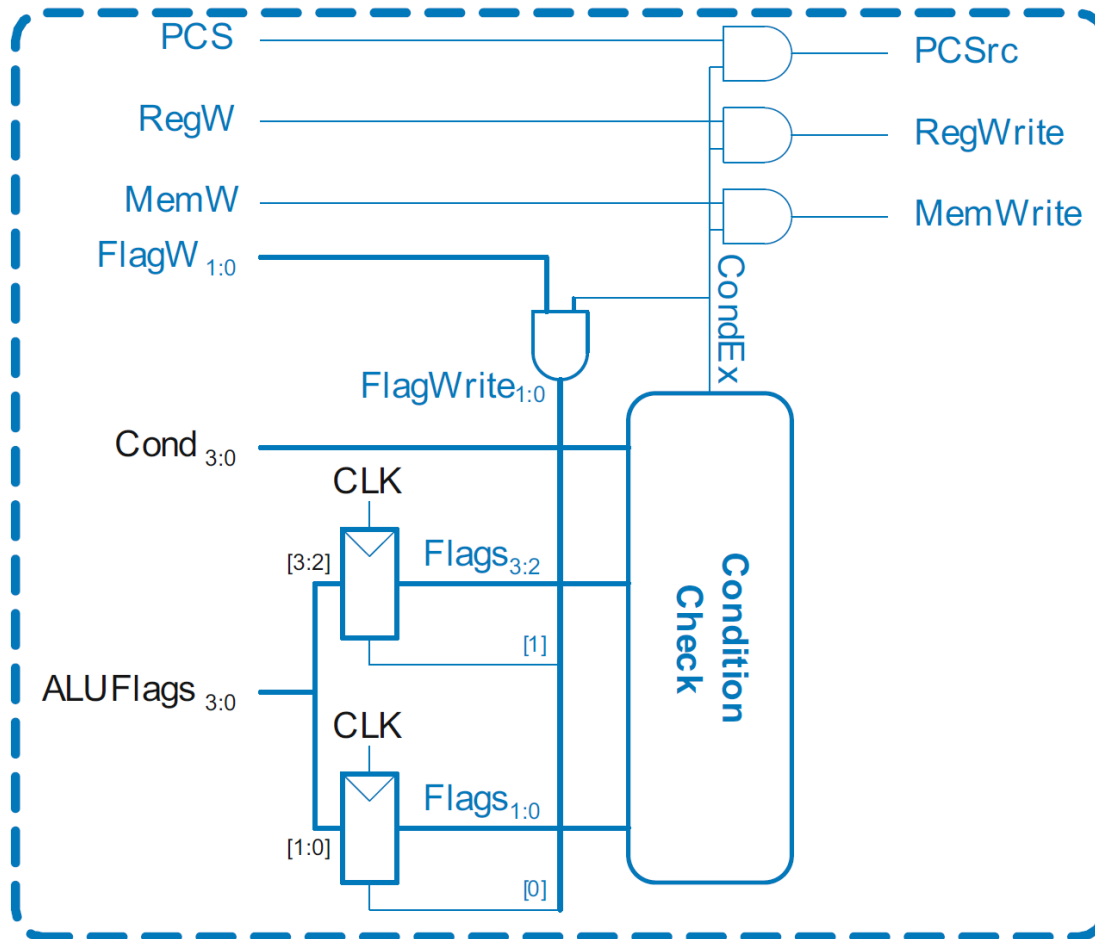
Single-Cycle Control: Conditional Logic



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



Single-Cycle Control: Conditional Logic



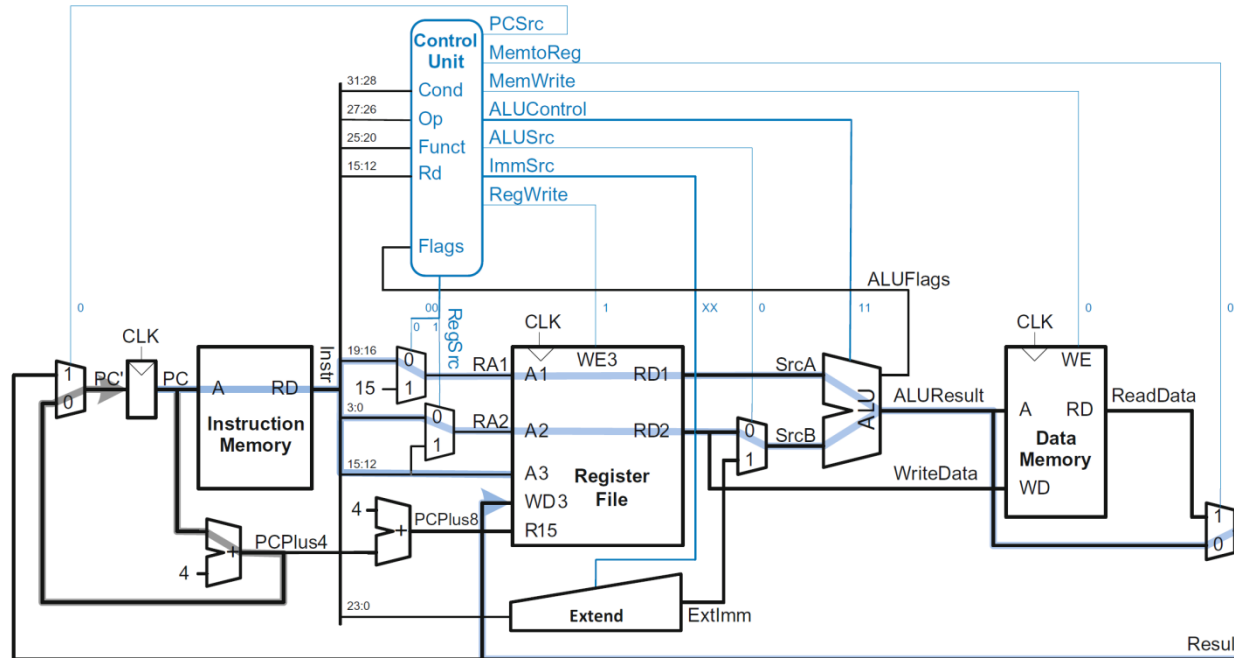
Recall:

- ADD, SUB update all Flags
- AND, OR update NZ only
- So Flags status register has two write enables
- $Flags_{3:0} = \{N, Z, C, V\}$

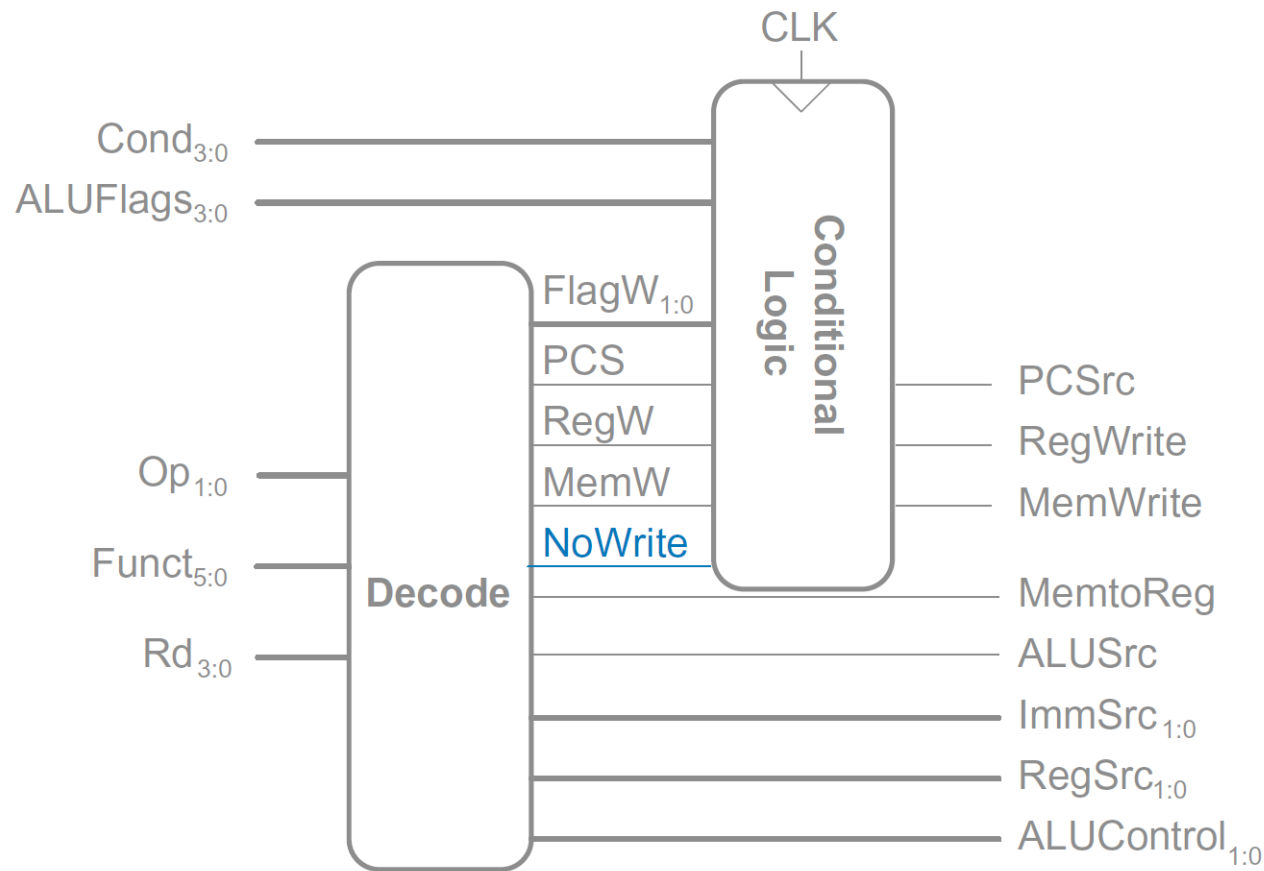


Example: ORR

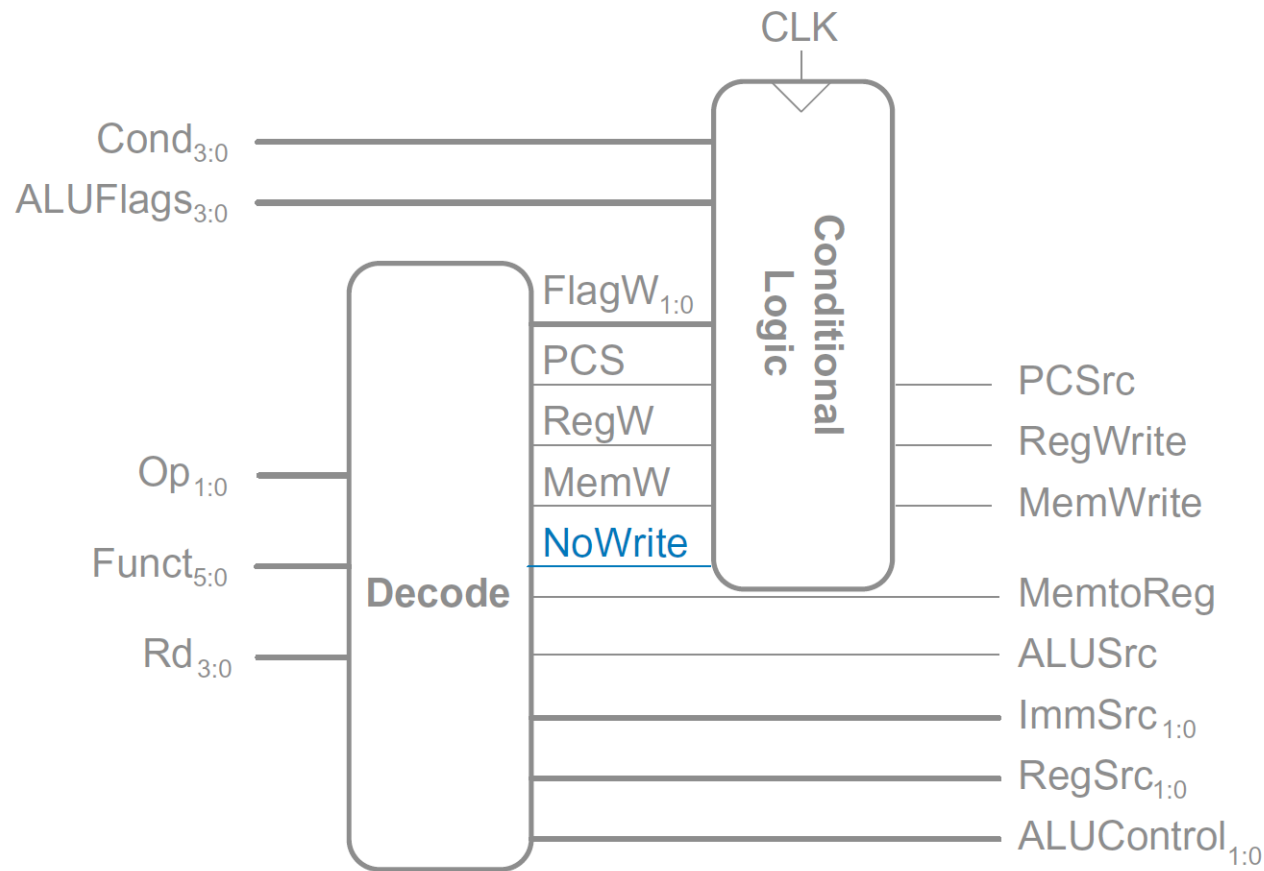
| Op | Funct ₅ | Funct ₀ | Type | Branch | MemoReg | MemW | ALUSrc | ImmSrc | RegW | RegSrc | ALUOp |
|----|--------------------|--------------------|--------|--------|---------|------|--------|--------|------|--------|-------|
| 00 | 0 | X | DP Reg | 0 | 0 | 0 | 0 | XX | 1 | 00 | 1 |



Extended Functionality: CMP



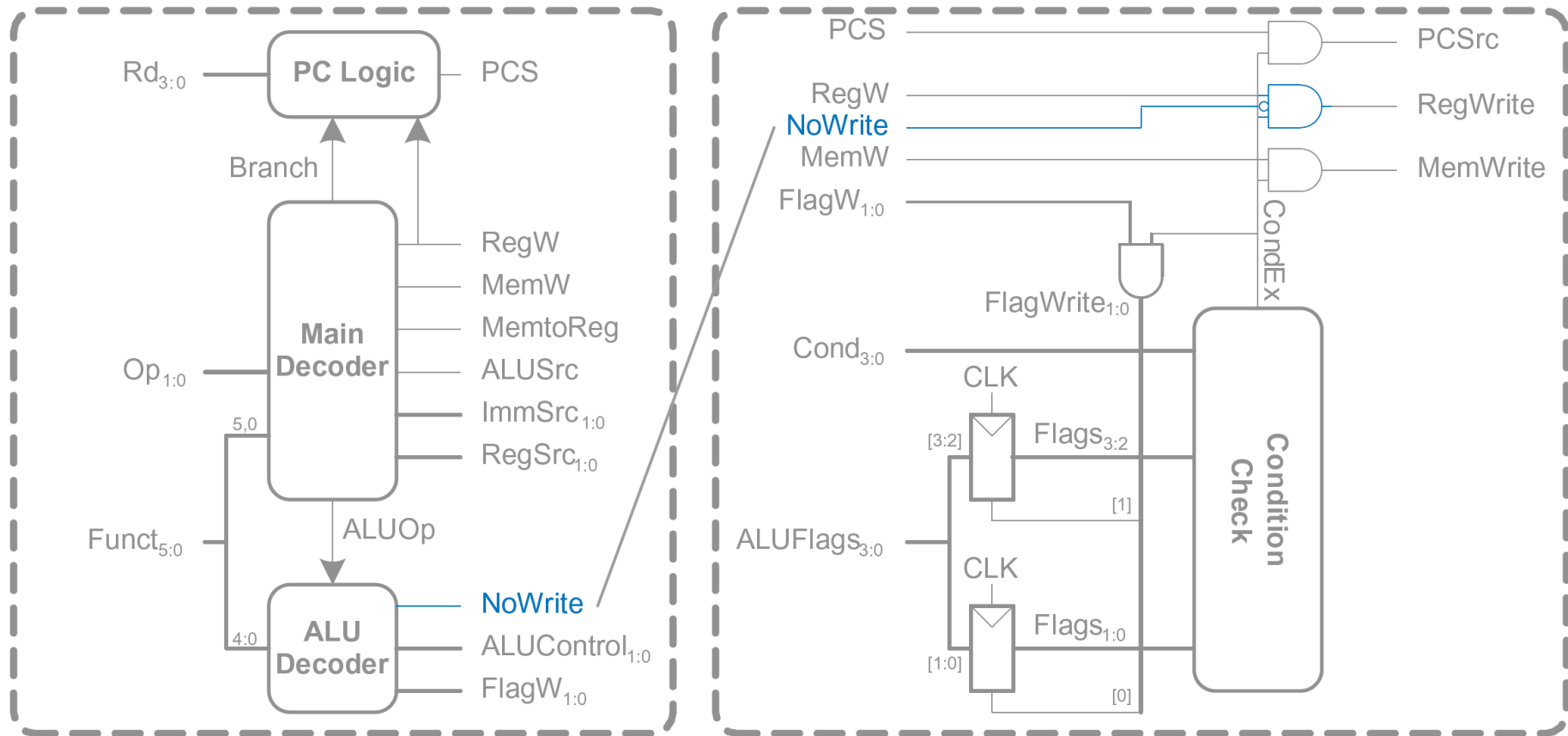
Extended Functionality: CMP



No change to datapath



Extended Functionality: CMP

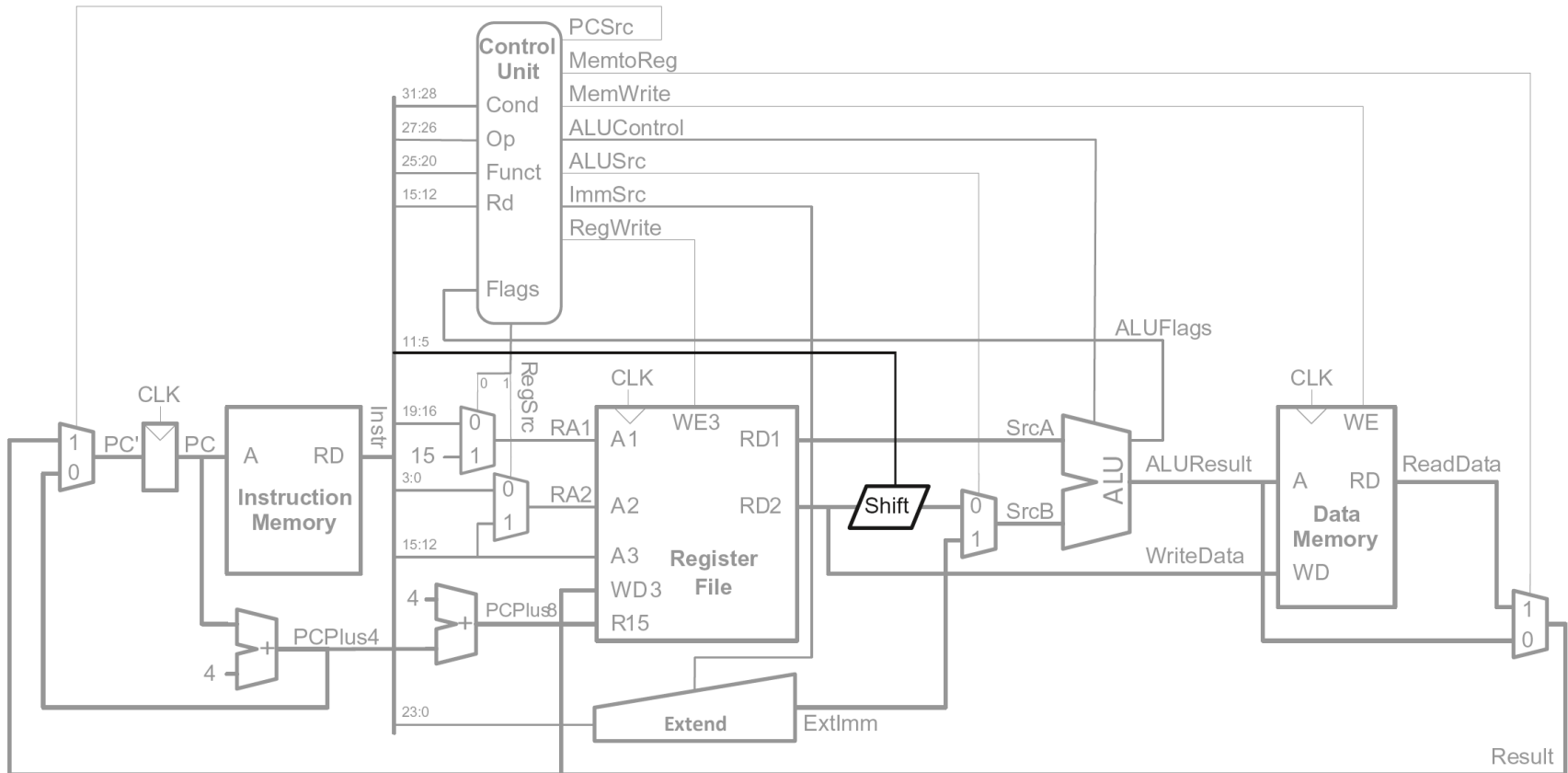


Extended Functionality: CMP

| 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 |



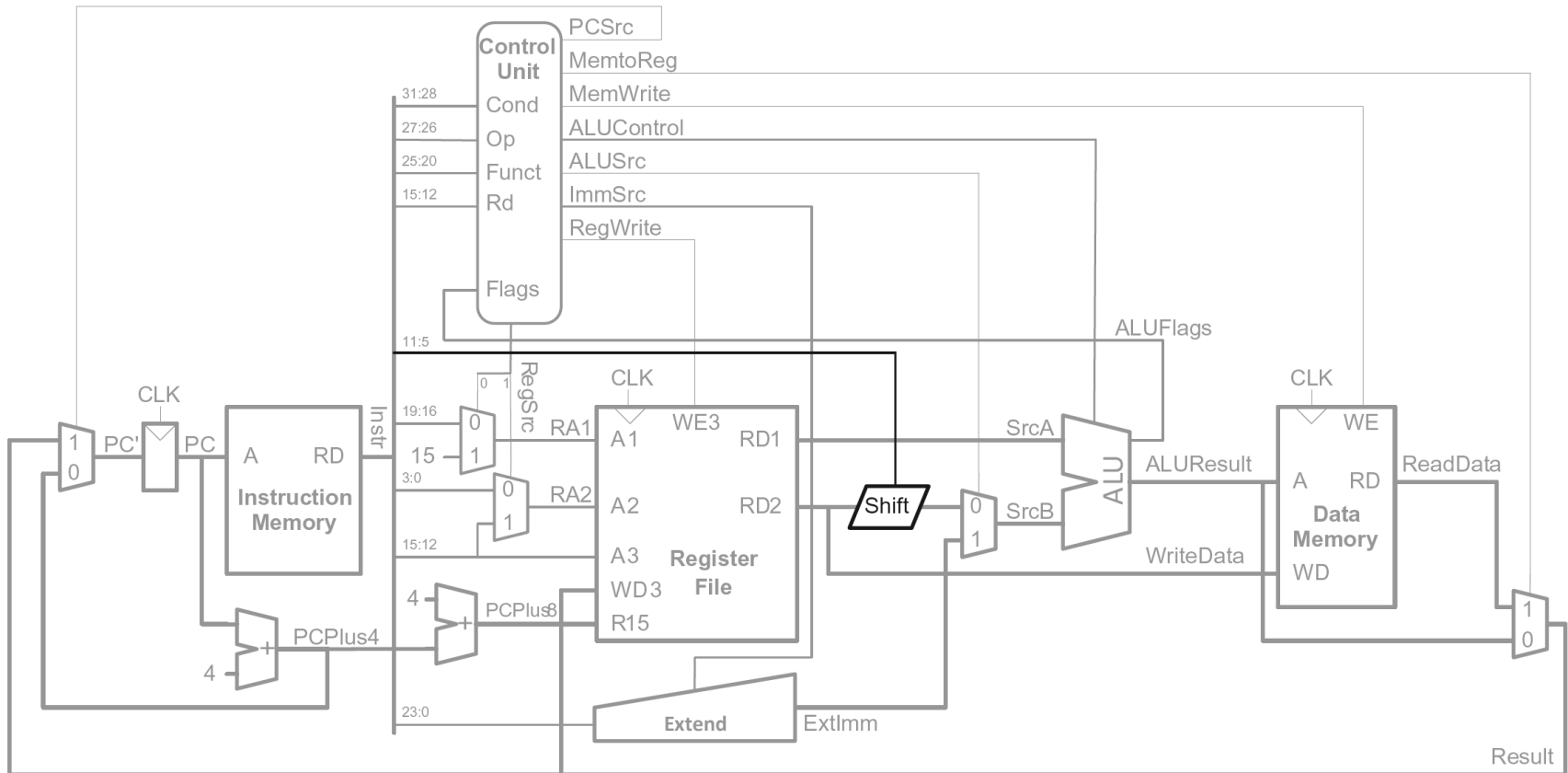
Extended Functionality: Shifted Register



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 |

Extended Functionality: Shifted Register



No change to controller



Review: Processor Performance

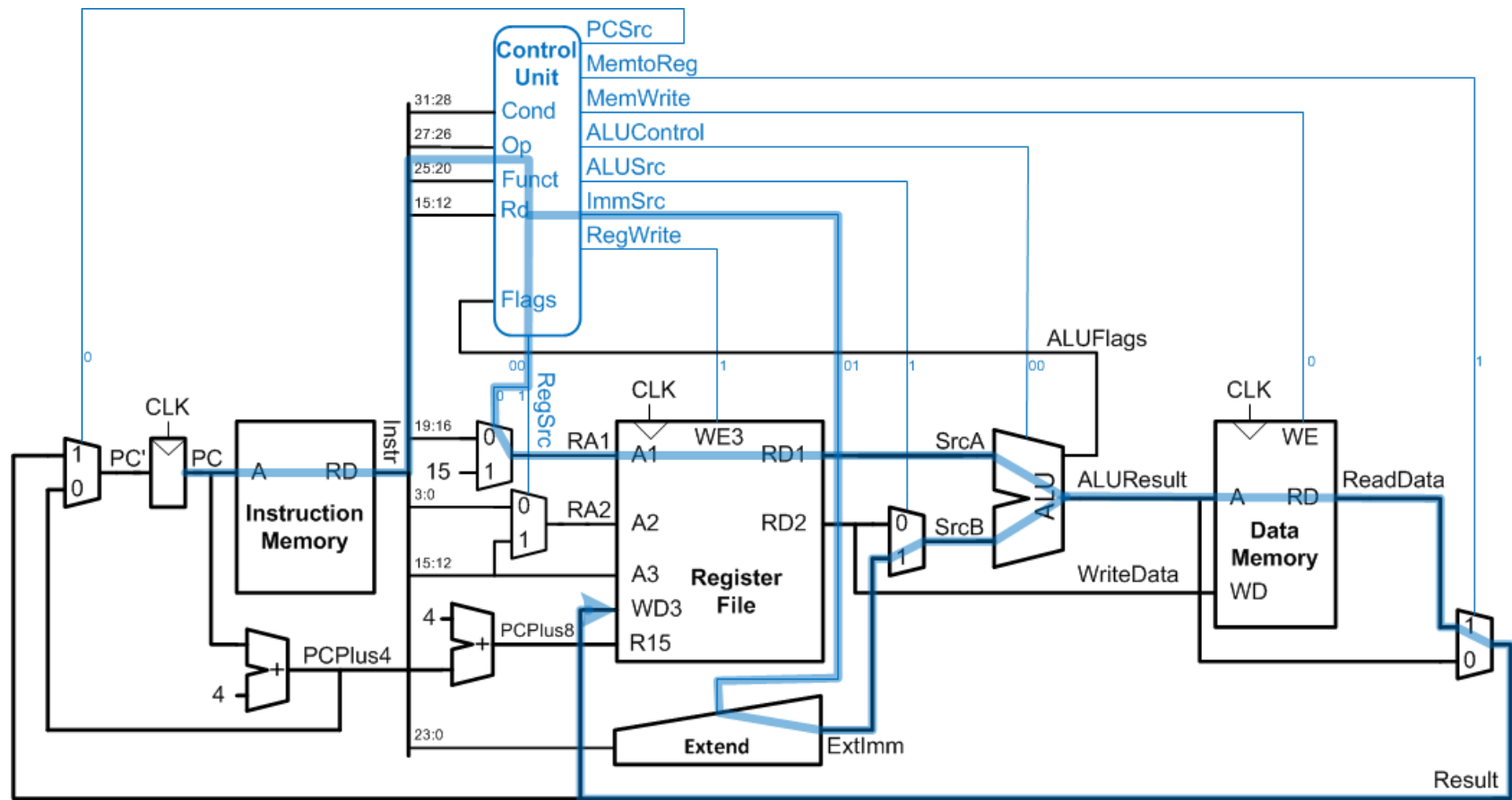
Program Execution Time

$$= (\# \text{instructions})(\text{cycles/instruction})(\text{seconds/cycle})$$

$$= \# \text{ instructions} \times \text{CPI} \times T_c$$



Single-Cycle Performance



T_c limited by critical path (LDR)



Single-Cycle Performance

- Single-cycle critical path:

$$T_{cl} = t_{pcq_PC} + t_{mem} + t_{dec} + \max[t_{mux} + t_{RFread}, t_{sxt} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

- Typically, limiting paths are:

- memory, ALU, register file

- $T_{cl} = t_{pcq_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup}$



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 |

$$T_{cl} = ?$$



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 |

$$\begin{aligned}T_{cl} &= t_{pcq_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup} \\&= [50 + 2(200) + 70 + 100 + 120 + 2(25) + 60] \text{ ps} \\&= \mathbf{840 \text{ ps}}\end{aligned}$$



Single-Cycle Performance Example

Program with 100 billion instructions:

$$\begin{aligned}\text{Execution Time} &= \# \text{ instructions} \times \text{CPI} \times T_c \\ &= (100 \times 10^9)(1)(840 \times 10^{-12} \text{ s}) \\ &= \mathbf{84 \text{ seconds}}\end{aligned}$$



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



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

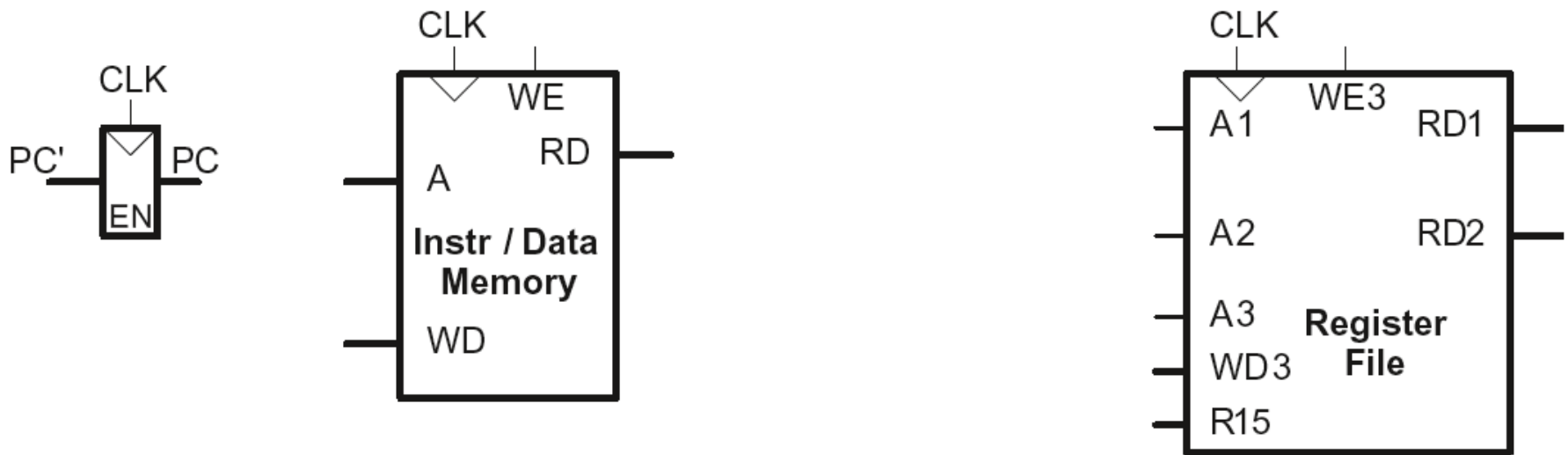
**Same design steps
as single-cycle:**

- **first datapath**
- **then control**



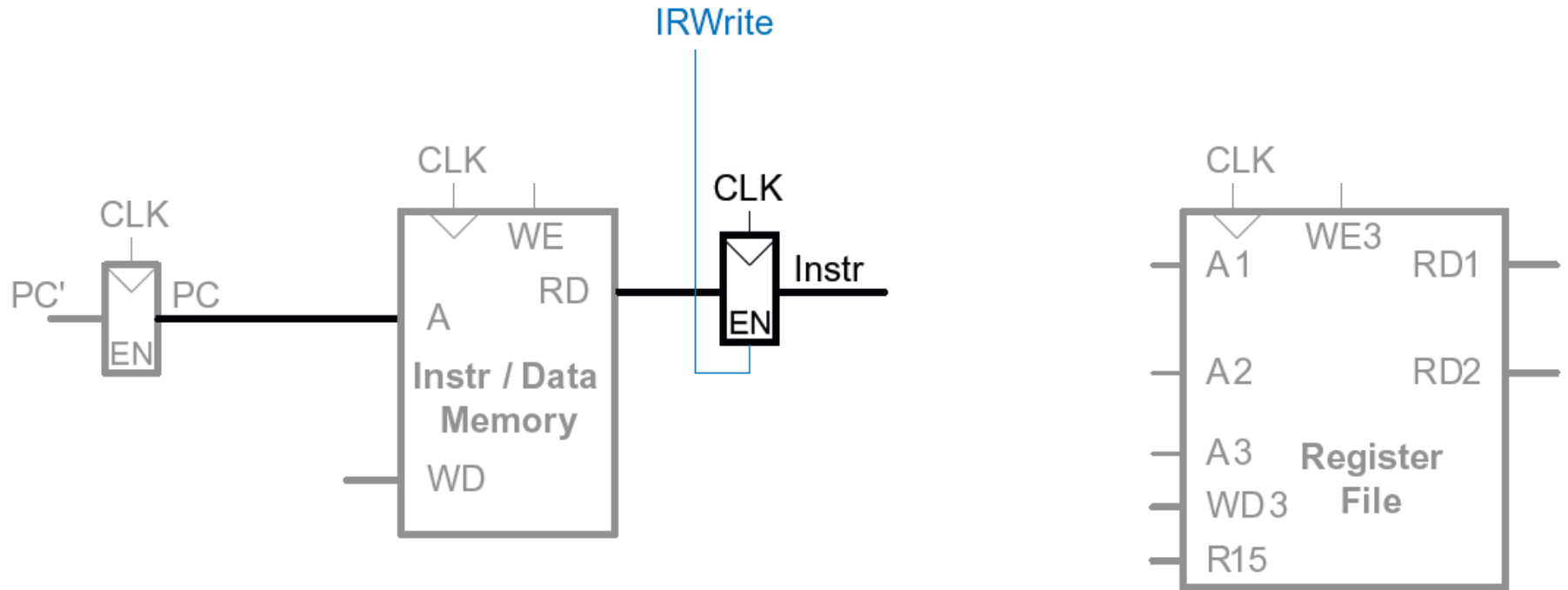
Multicycle State Elements

Replace Instruction and Data memories with a single unified memory – more realistic



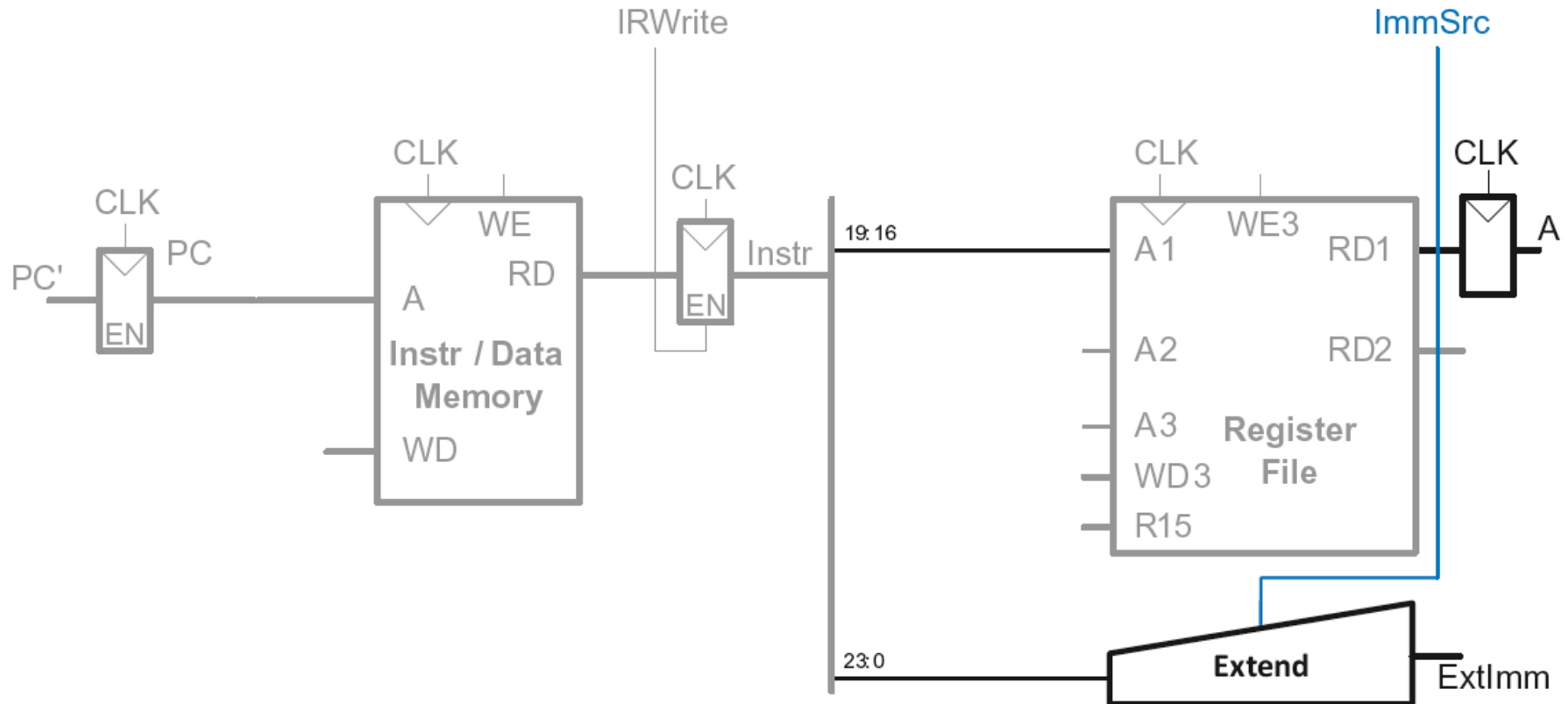
Multicycle Datapath: Instruction Fetch

STEP 1: Fetch instruction



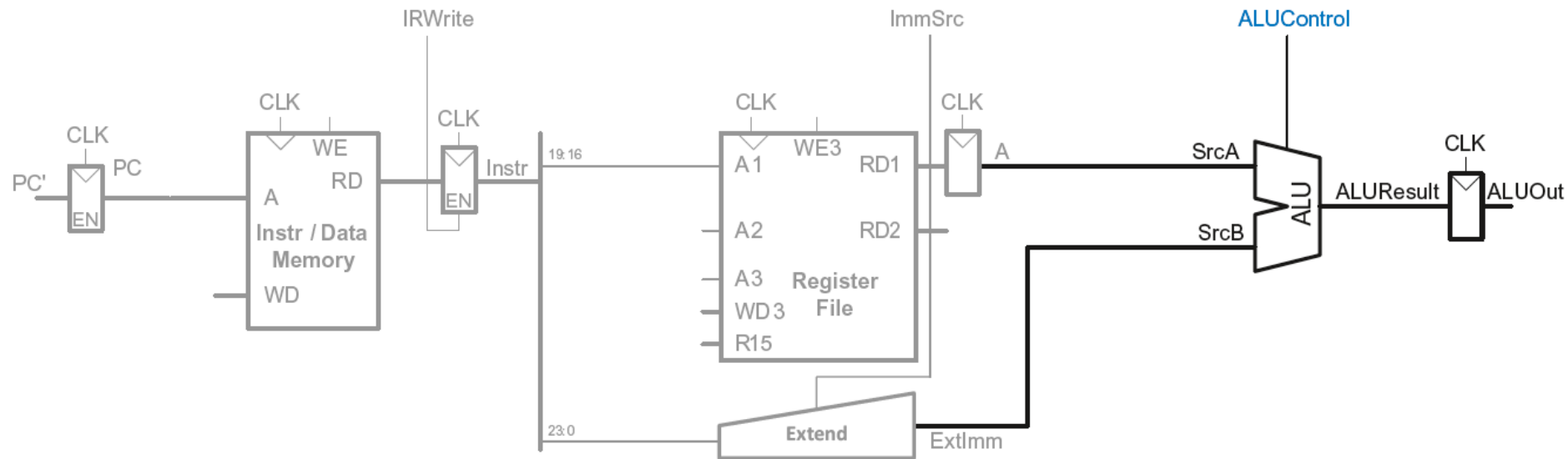
Multicycle Datapath: LDR Register Read

STEP 2: Read source operands from RF



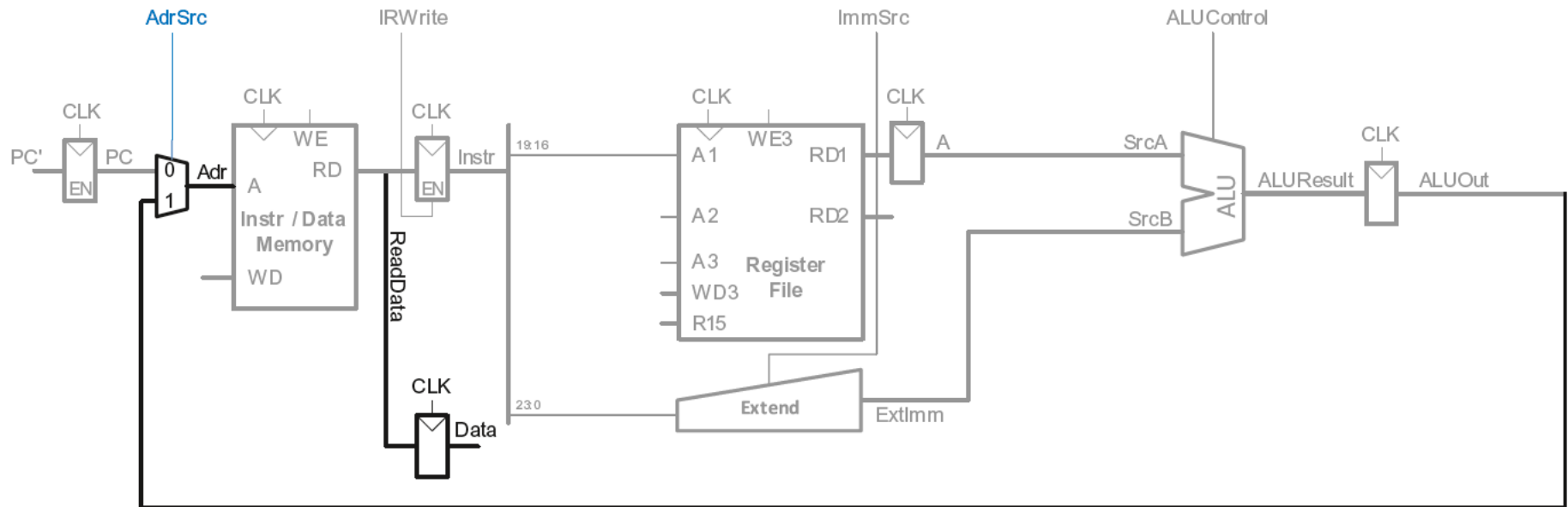
Multicycle Datapath: LDR Address

STEP 3: Compute the memory address



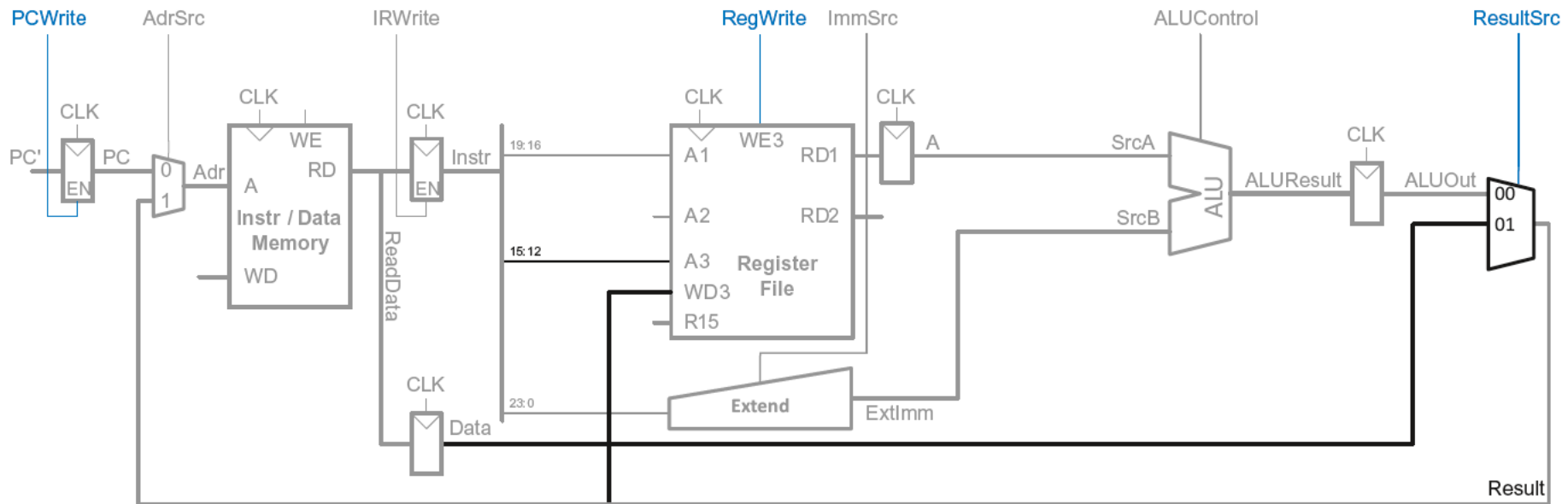
Multicycle Datapath: LDR Memory Read

STEP 4: Read data from memory



Multicycle Datapath: LDR Write Register

STEP 5: Write data back to register file



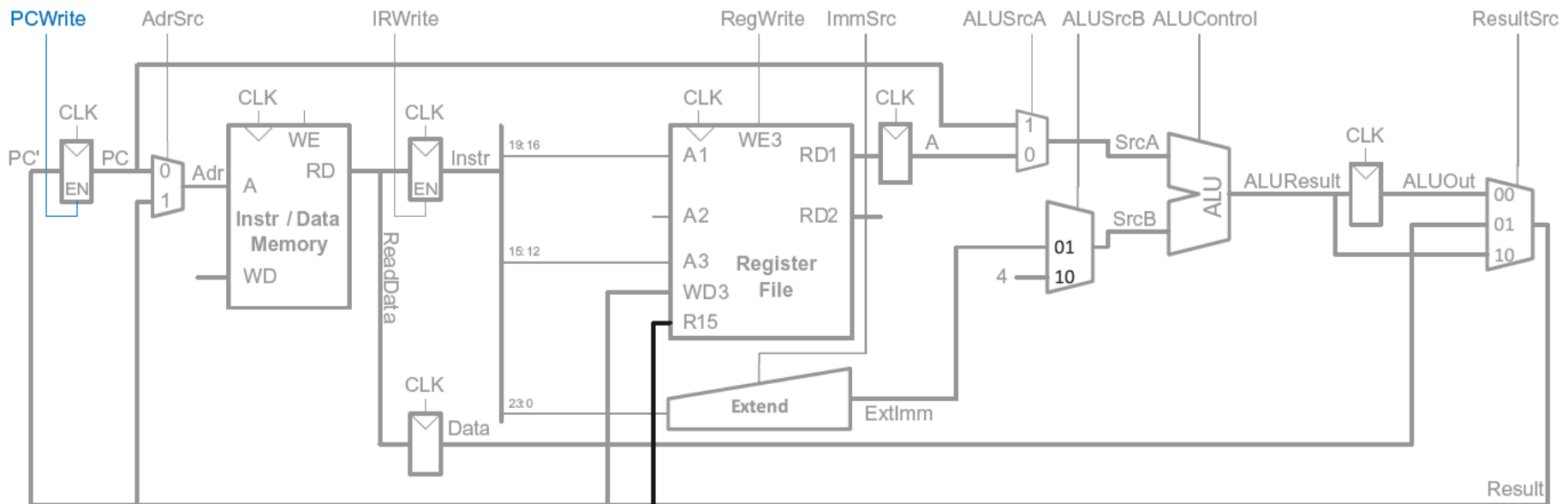
STEP 6: Increment PC



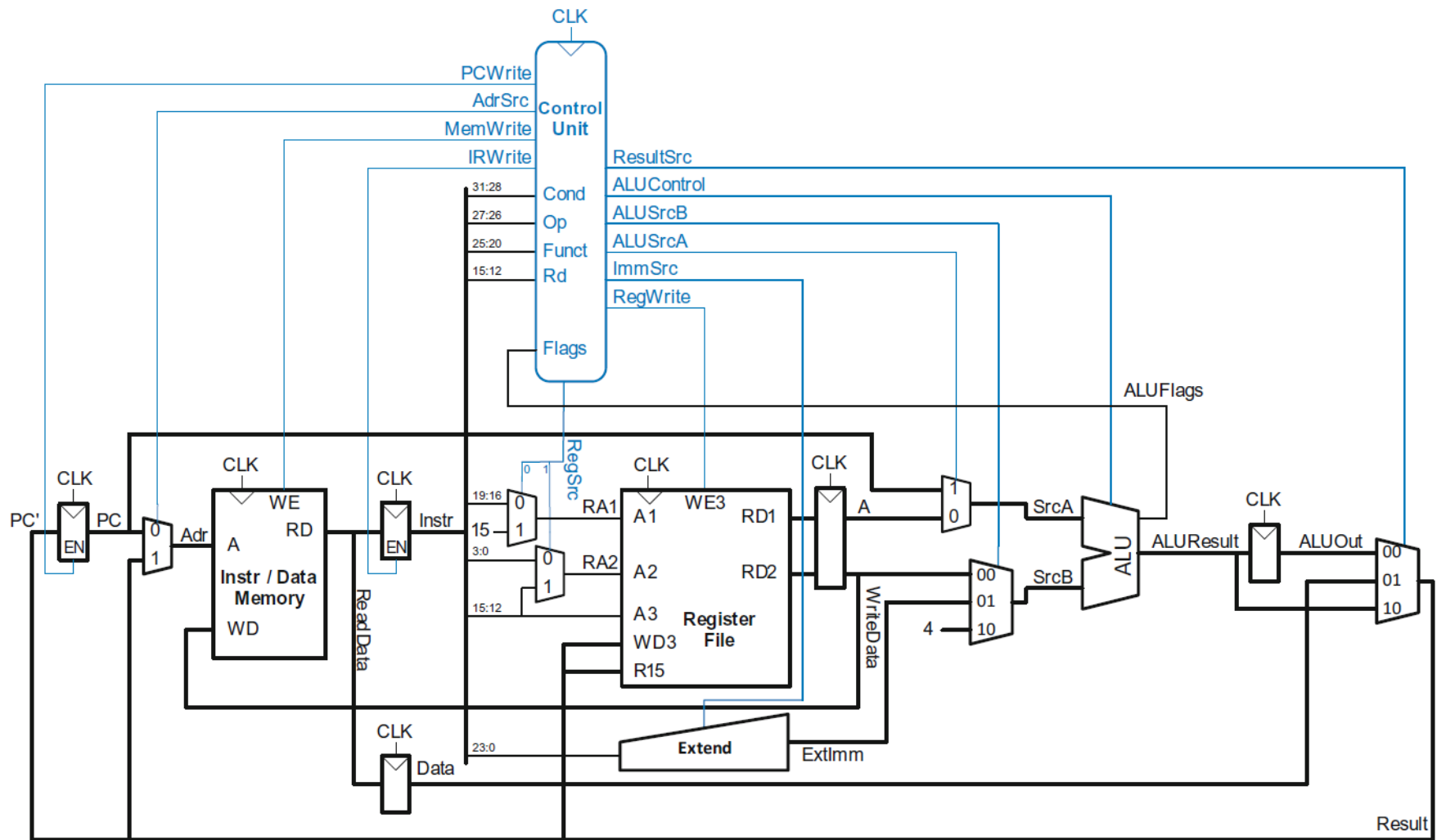
Multicycle Datapath: Access to PC

PC can be read/written by instruction

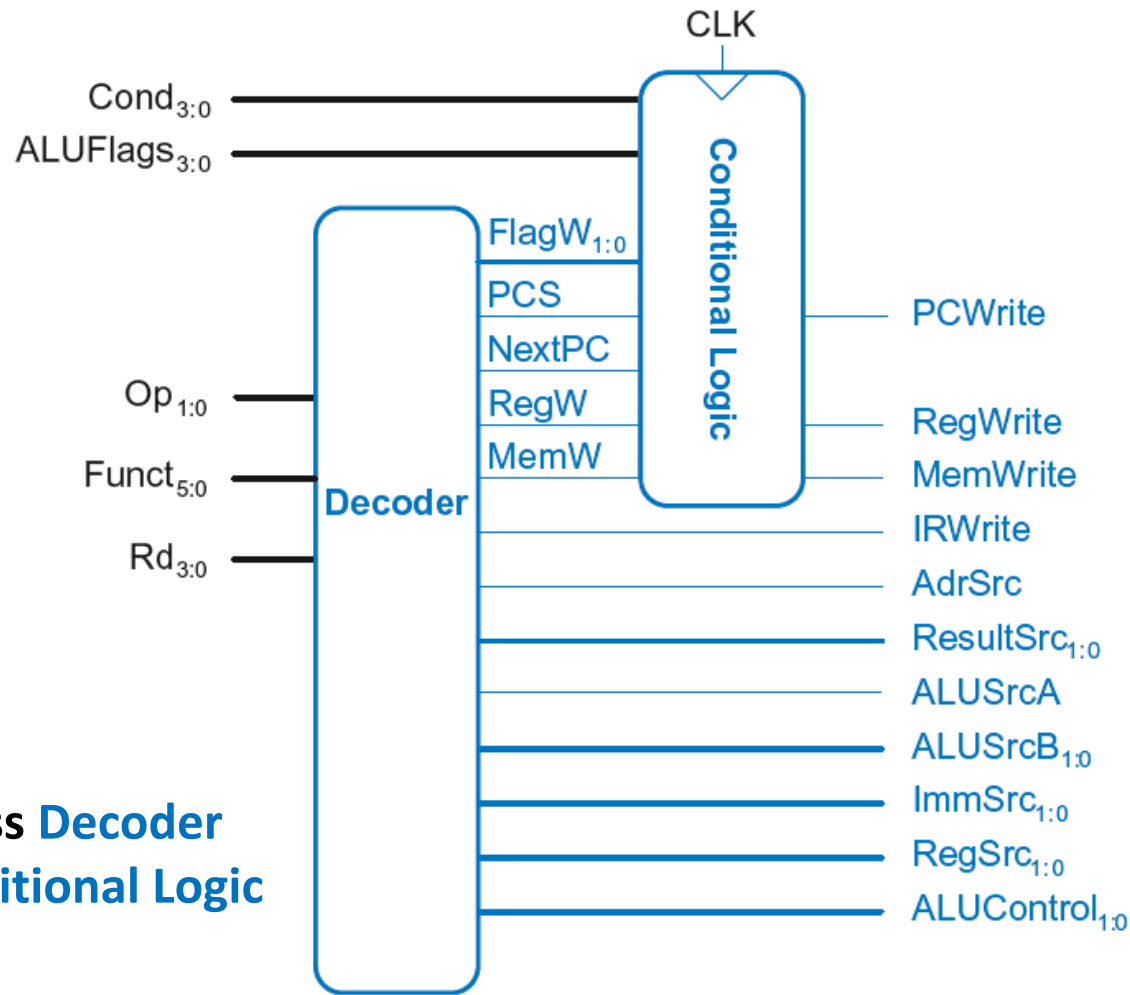
- **Read:** R15 (PC+8) available in Register File
- **Write:** Be able to write result of instruction to PC



Multicycle ARM Processor



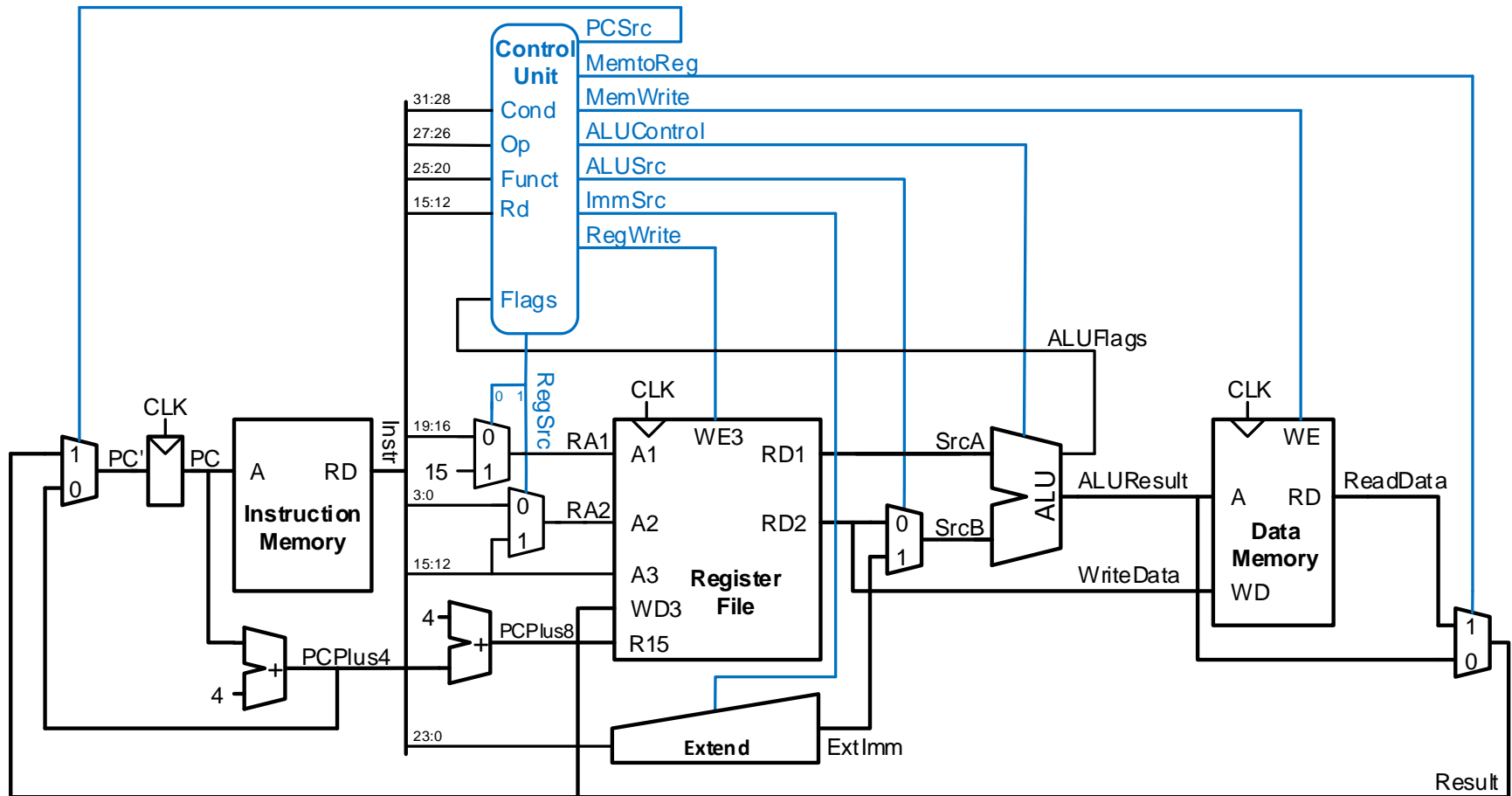
Multicycle Control



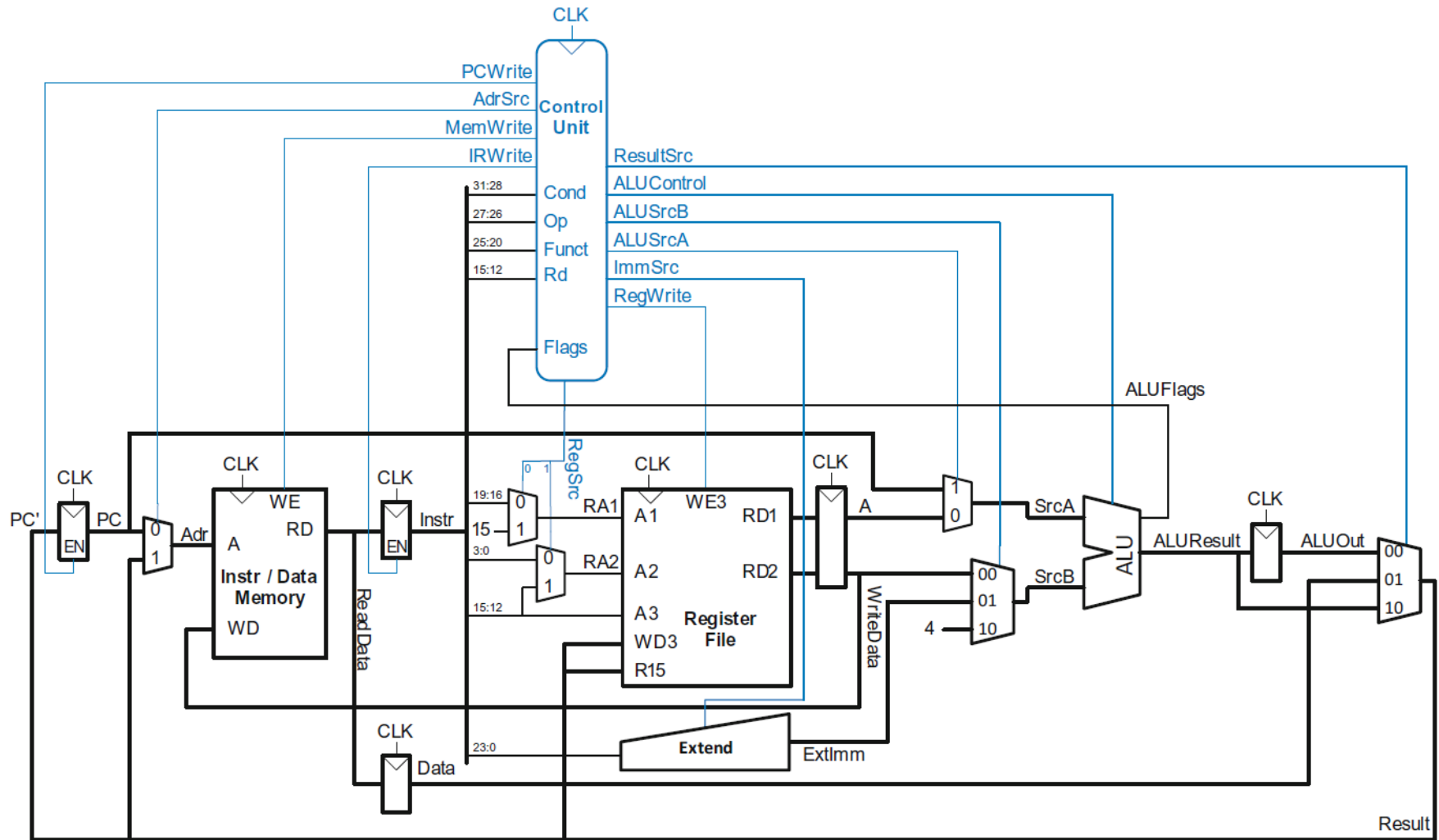
- First, discuss **Decoder**
- Then, **Conditional Logic**



Review: Single-Cycle ARM Processor



Review: Multicycle ARM Processor



Backup Material

Multicycle Datapath: Read to PC (R15)

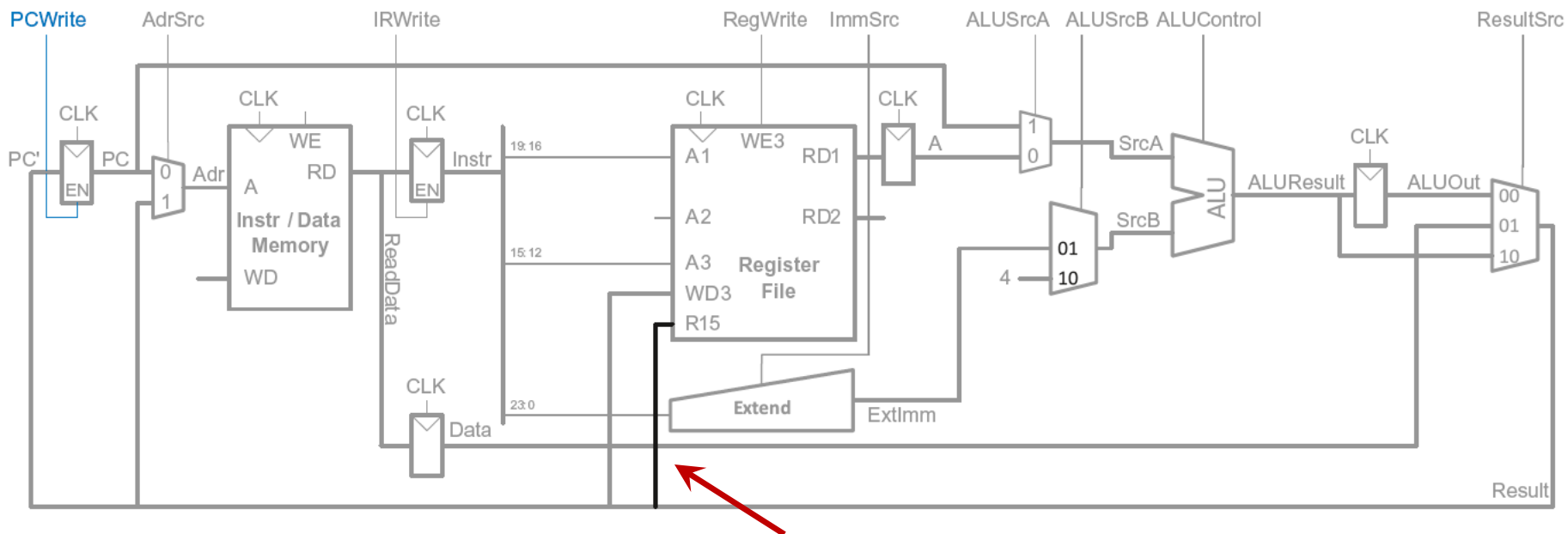
Example: ADD R1, **R15**, R2



Multicycle Datapath: Read to PC (R15)

Example: ADD R1, R15, R2

- R15 needs to be read as PC+8 from Register File (RF) in 2nd step
- So, also in 2nd step, PC + 8 is produced by ALU and routed to R15 input of RF



Multicycle Datapath: Read to PC (R15)

Example: `ADD R1, R15, R2`

- R15 needs to be read as PC+8 from Register File (RF) in 2nd step
- So, also in 2nd step, PC + 8 is produced by ALU and routed to R15 input of RF
 - $SrcA = PC$ (which was already updated in step 1 to PC+4)
 - $SrcB = 4$
 - $ALUResult = PC + 8$
- ALUResult is fed to R15 input port of RF in 2nd step (which is then routed to RD1 output of RF)

