Chapter 7

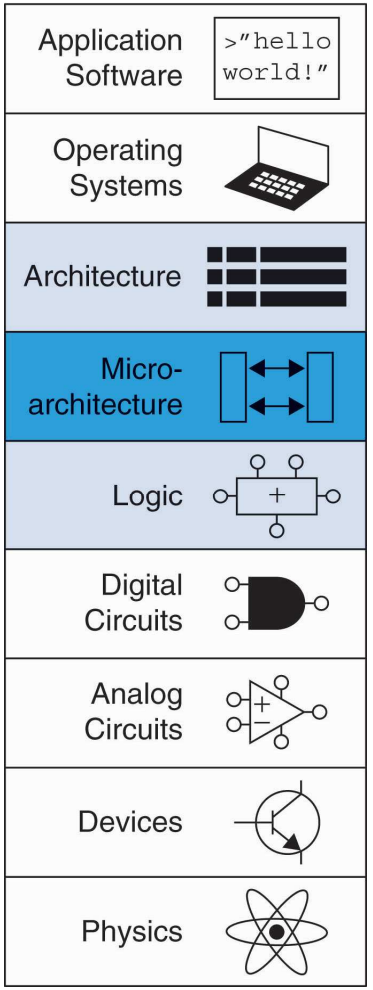
Digital Design and Computer Architecture: ARM® Edition

Sarah L. Harris and David Money Harris

ELSEVIER

Chapter 7 :: Topics

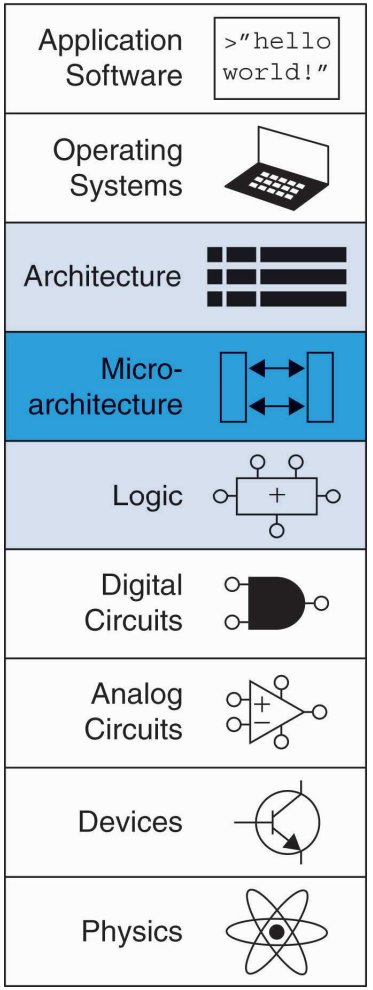
- **Introduction**
- **Performance Analysis**
- **Single-Cycle Processor**
- **Multicycle Processor**
- **Pipelined Processor**
- **Advanced Microarchitecture**



ELSEVIER

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

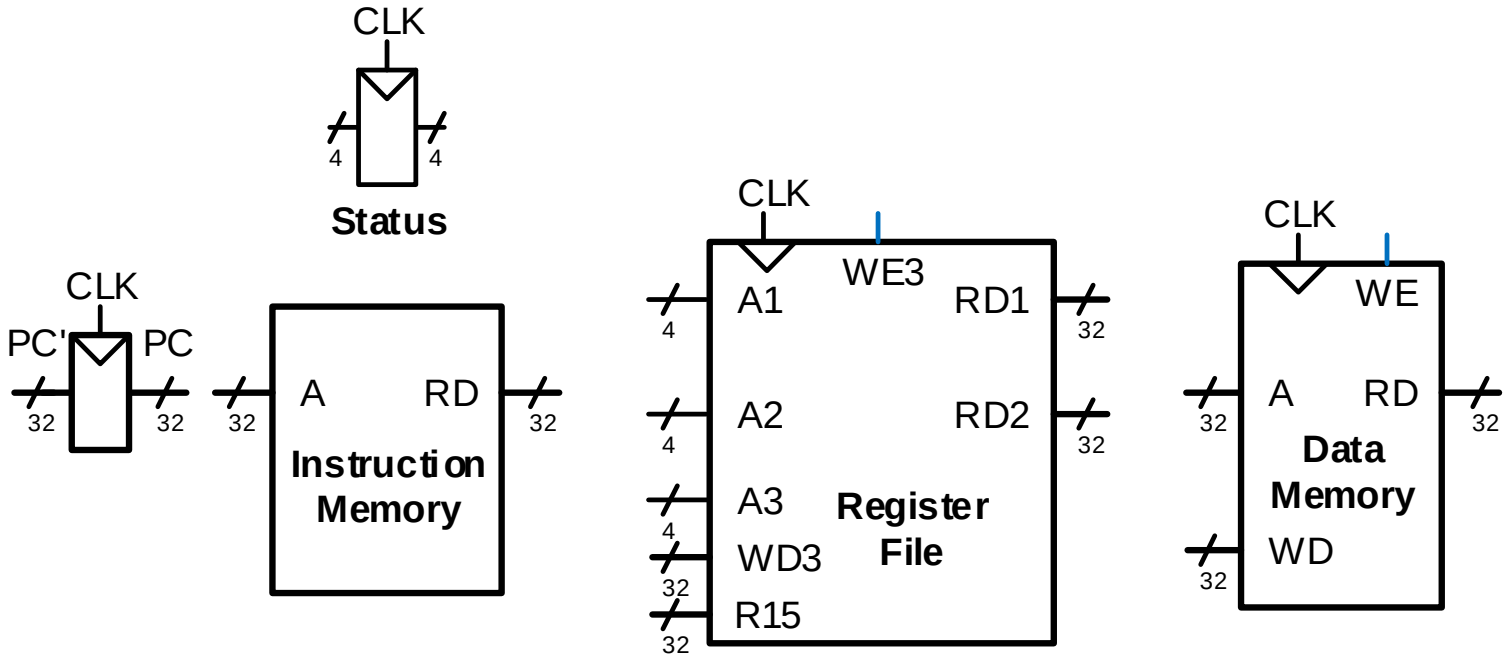
- Consider **subset** of ARM instructions:
  - **Data-processing instructions:**
    - **ADD, SUB, AND, ORR**
    - with register and immediate Src2, but **no shifts**
  - **Memory instructions:**
    - **LDR, STR**
    - with **positive immediate offset**
  - **Branch instructions:**
    - **B**

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

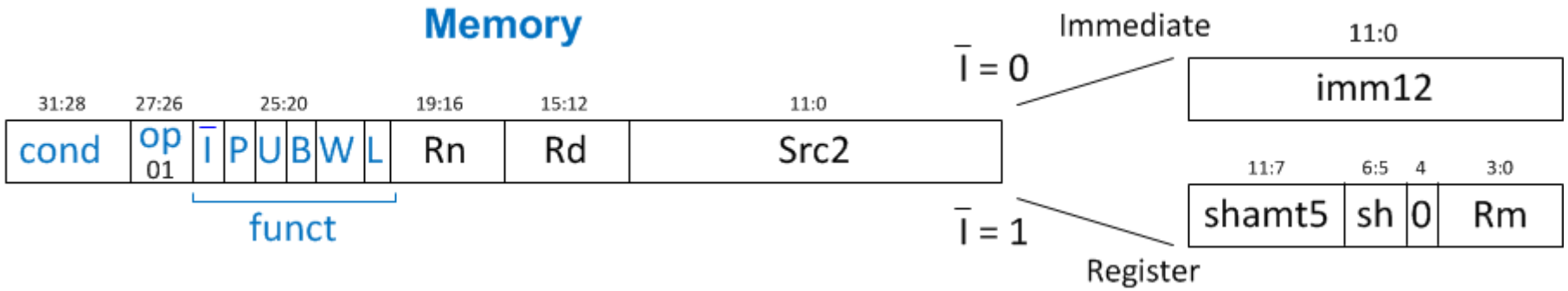
Single-Cycle ARM Processor

- **Datapath**
- Control

Single-Cycle ARM Processor

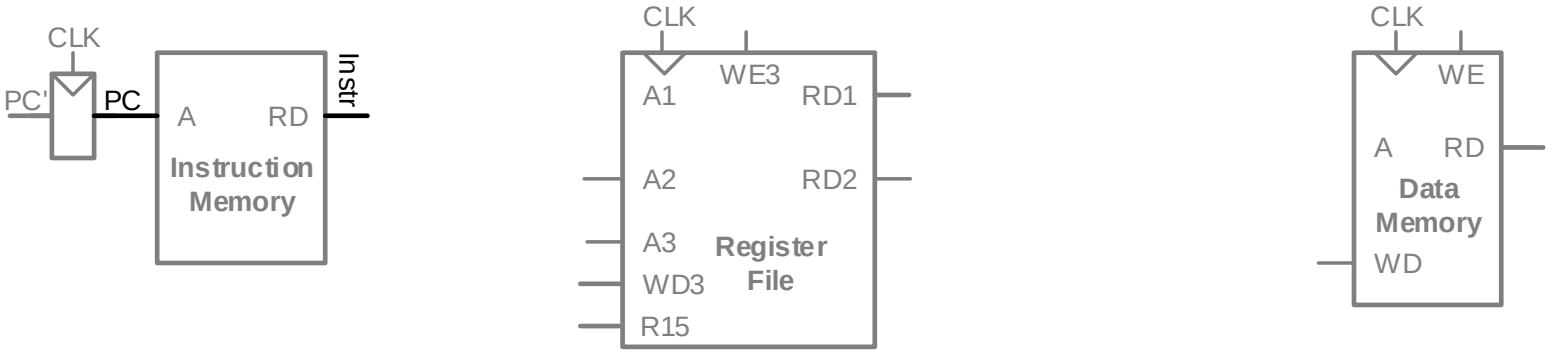
- **Datapath:** start with LDR instruction
- **Example:** `LDR R1, [R2, #5]`

  `LDR Rd, [Rn, imm12]`



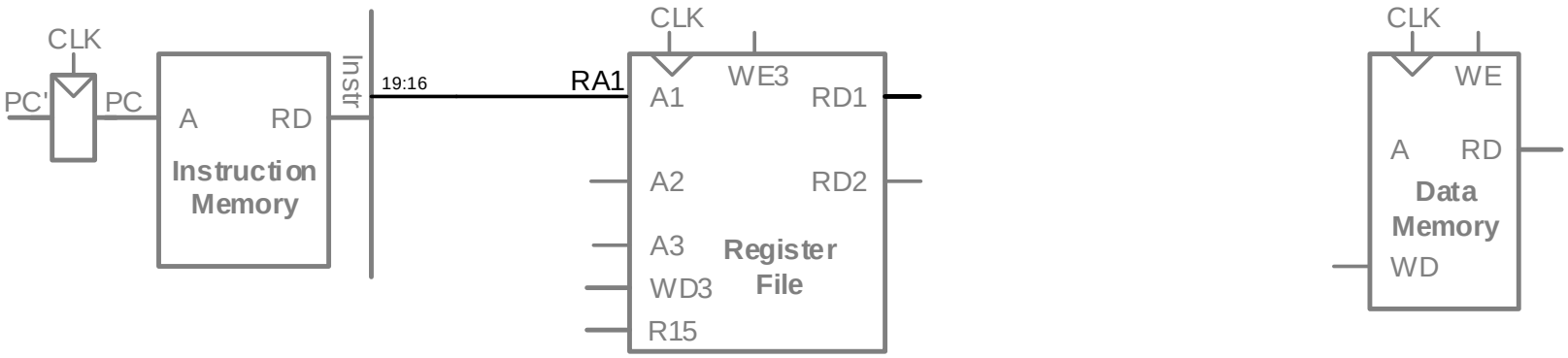
Single-Cycle Datapath: LDR fetch

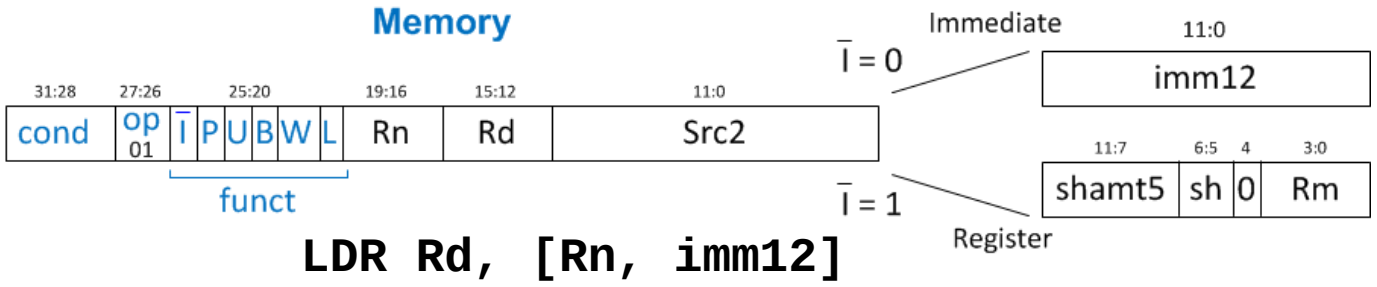
STEP 1: Fetch instruction



Single-Cycle Datapath: LDR Reg Read

STEP 2: Read source operands from RF

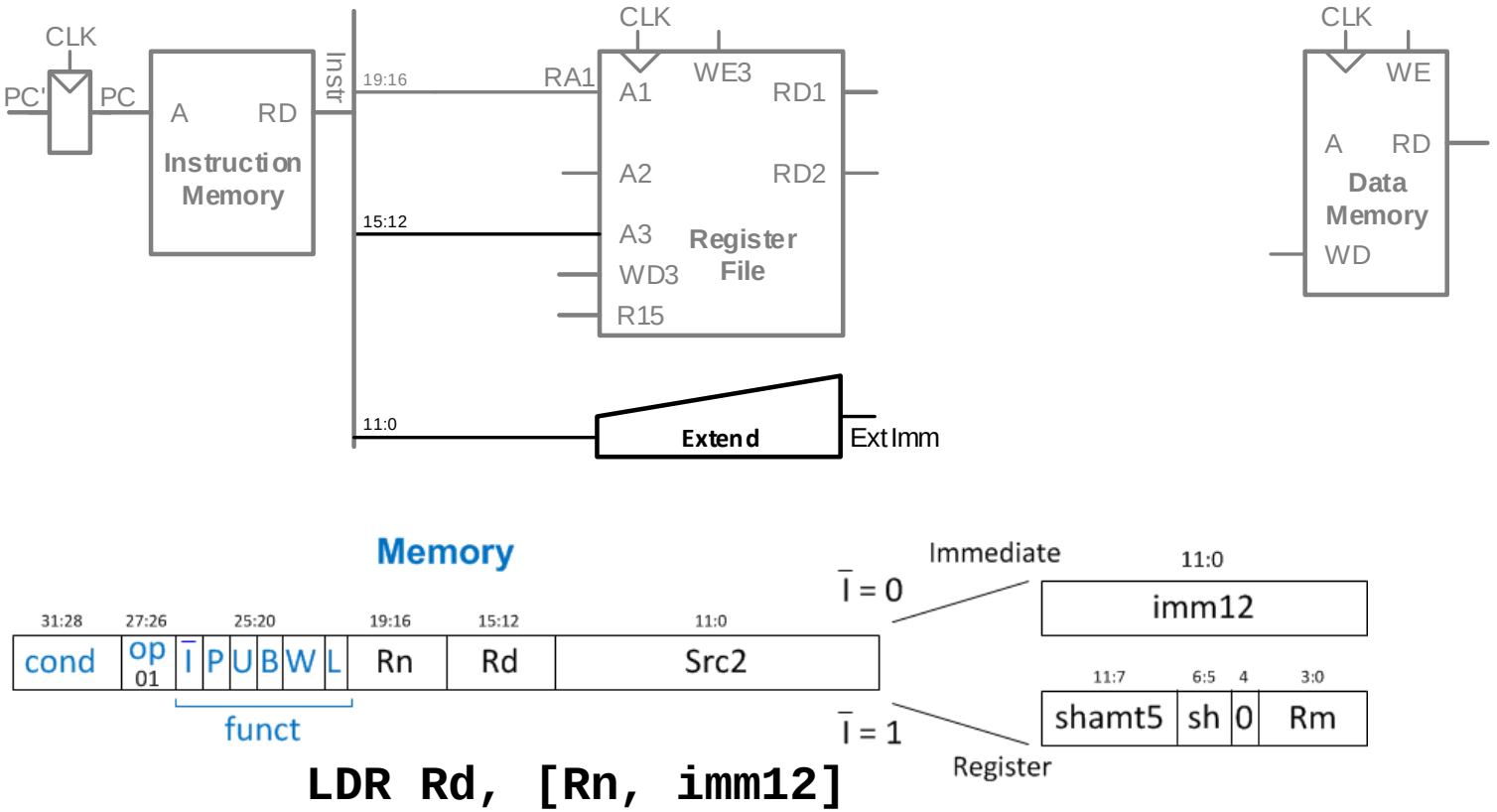




LDR Rd, [Rn, imm12]

Single-Cycle Datapath: LDR Immed.

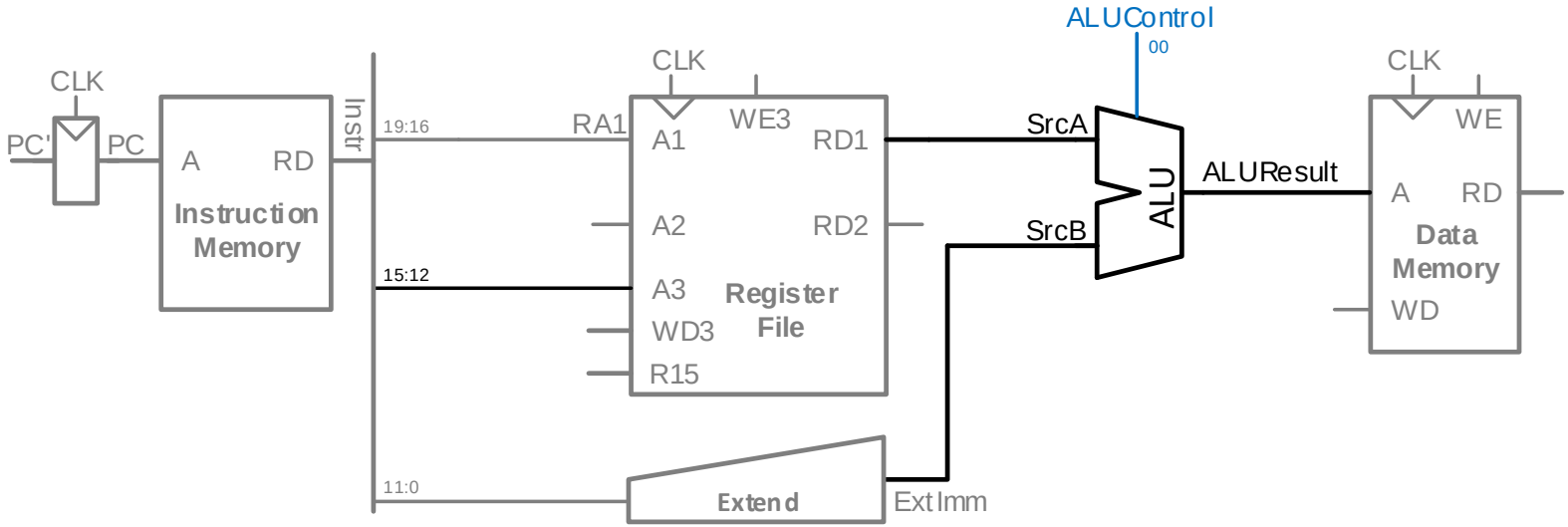
STEP 3: Extend the immediate

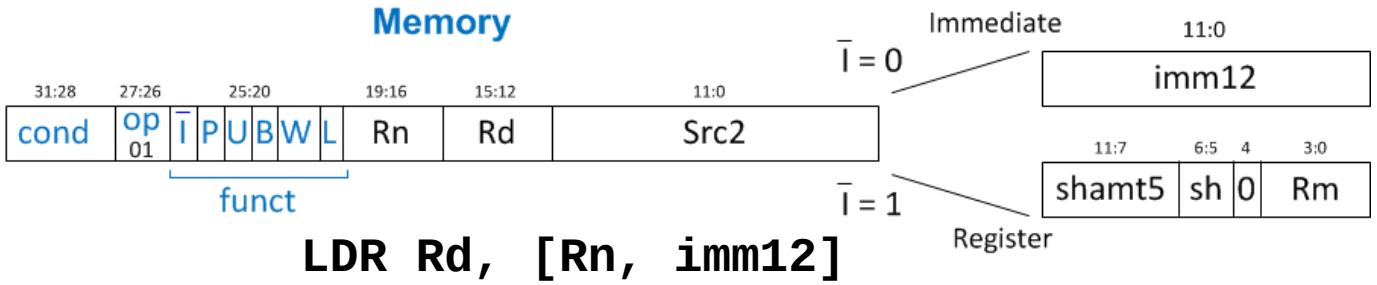


LDR Rd, [Rn, imm12]

Single-Cycle Datapath: LDR Address

STEP 4: Compute the memory address



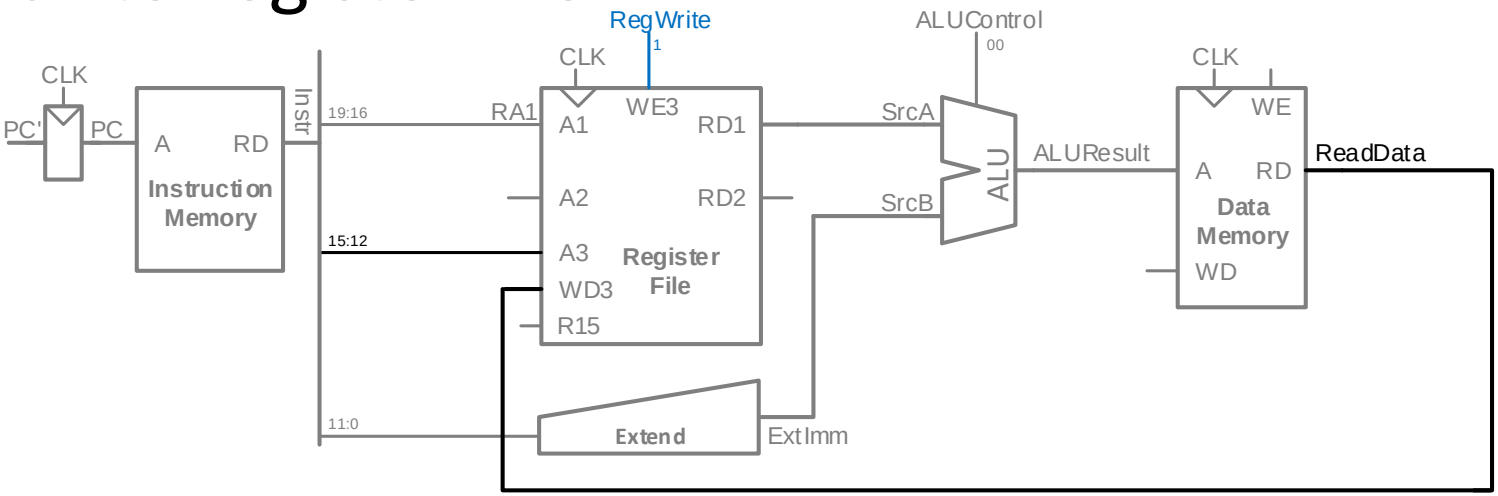


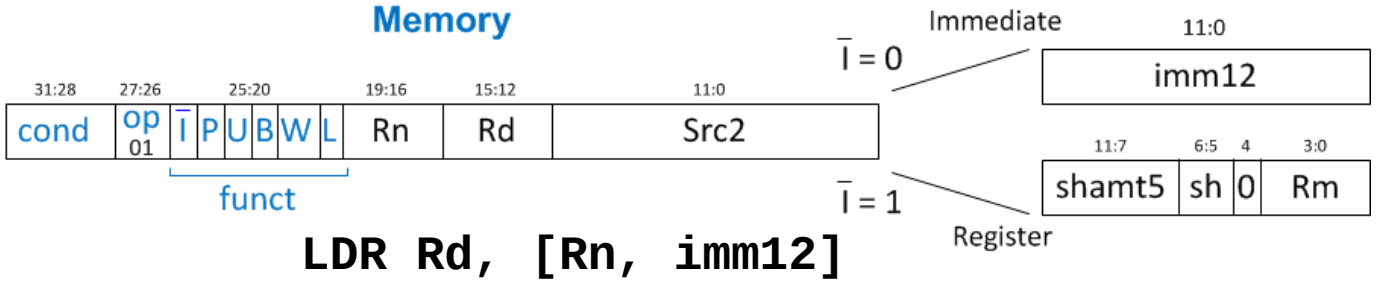
LDR Rd, [Rn, imm12]

ELSEVIER

Single-Cycle Datapath: LDR Mem Read

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



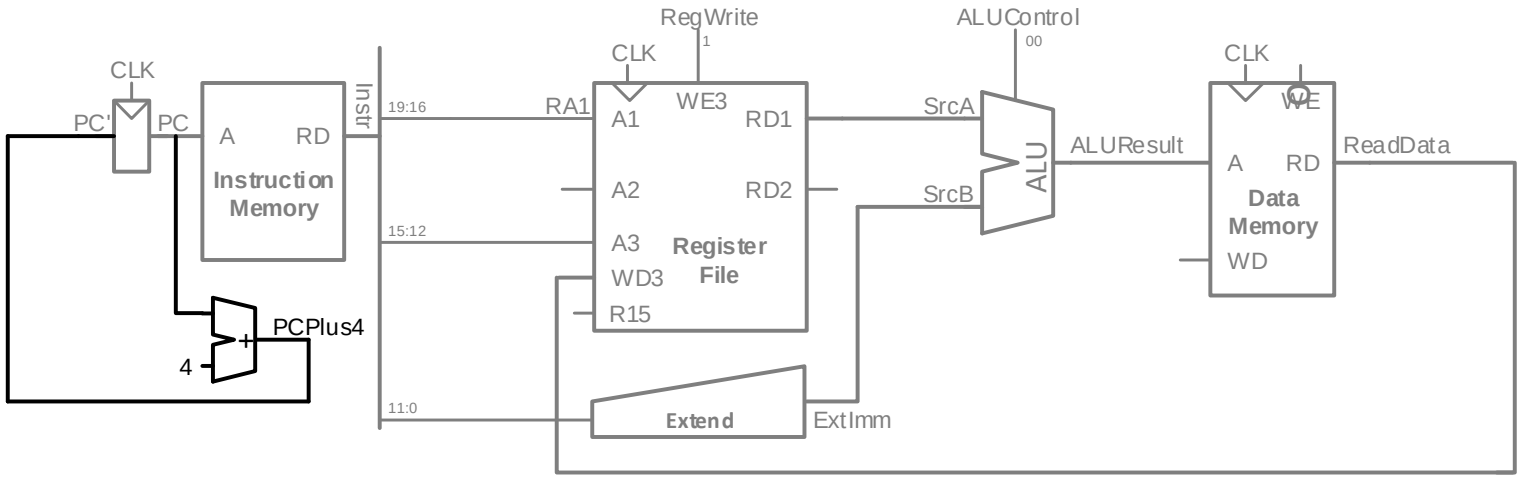


LDR Rd, [Rn, imm12]

ELSEVIER

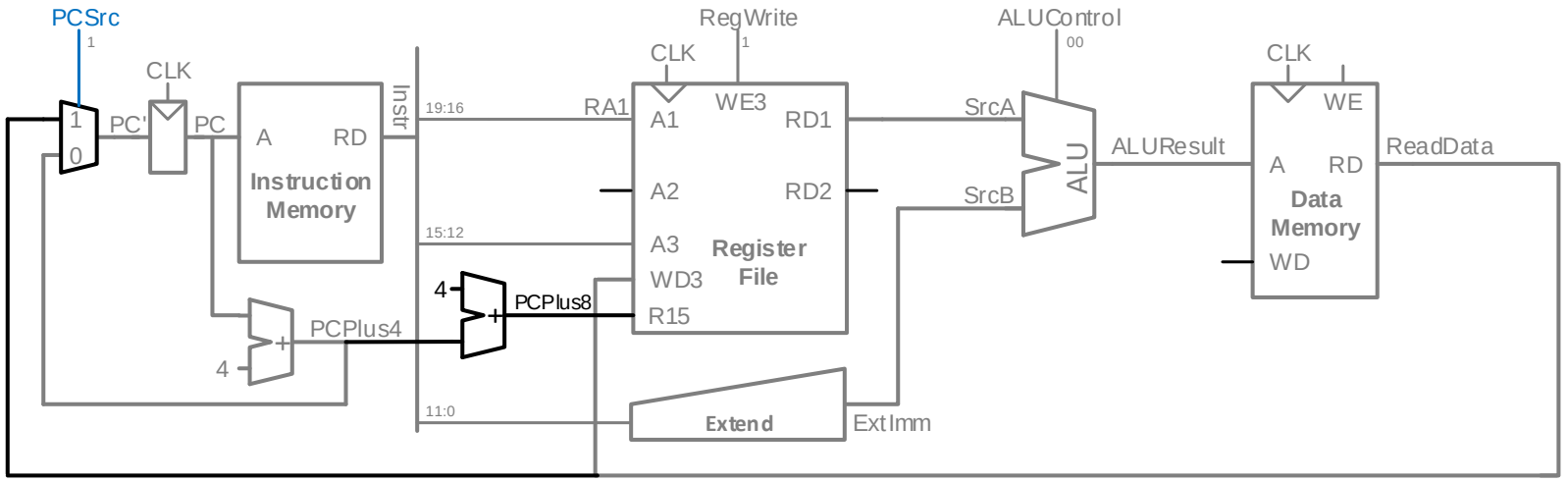
Single-Cycle Datapath: PC Increment

STEP 6: Determine address of next instruction



Single-Cycle Datapath: Access to PC

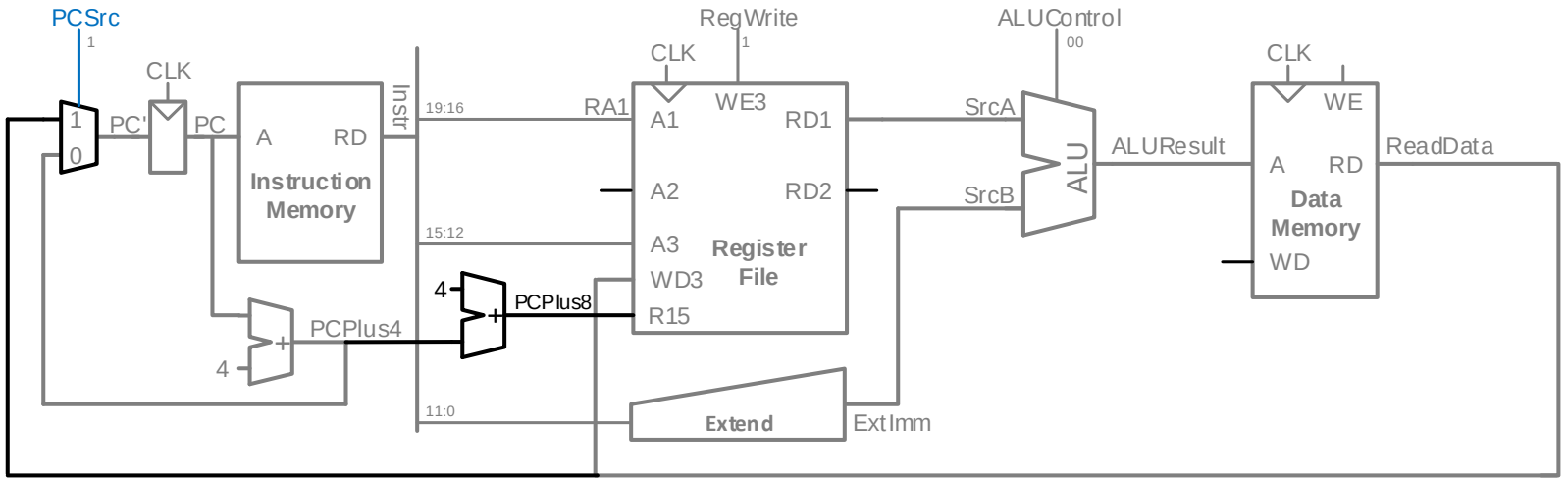
PC can be source/destination of instruction



Single-Cycle Datapath: Access to PC

PC can be source/destination of instruction

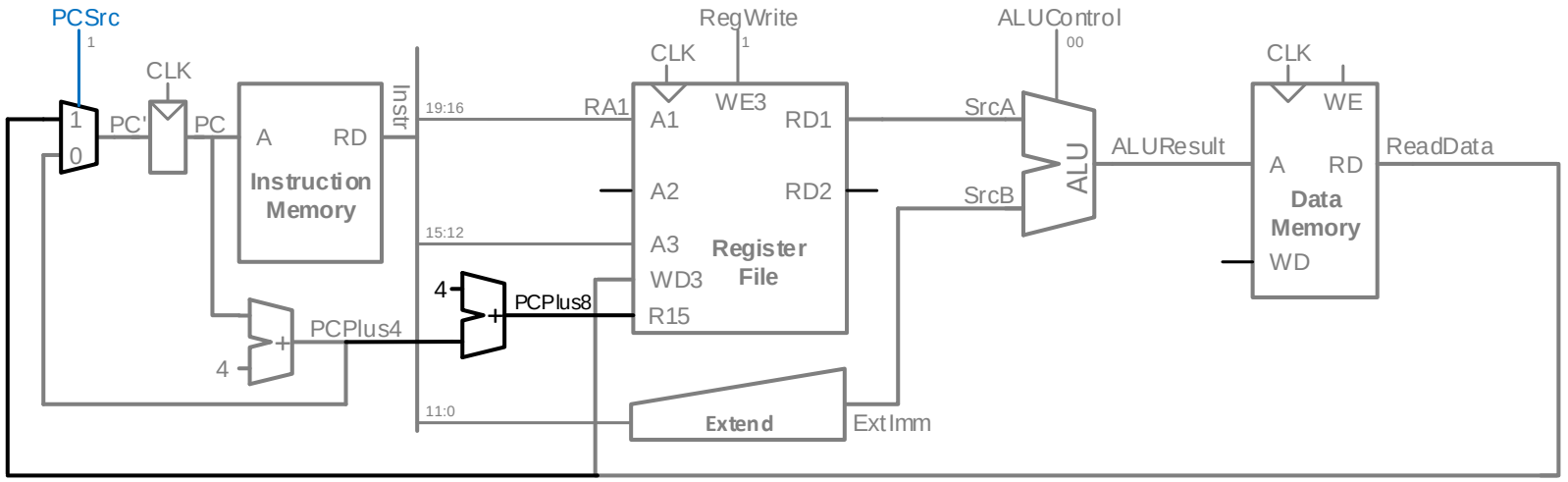
- **Source:** R15 must be available in Register File
  - **PC** is read as the current **PC plus 8**



Single-Cycle Datapath: Access to PC

PC can be source/destination of instruction

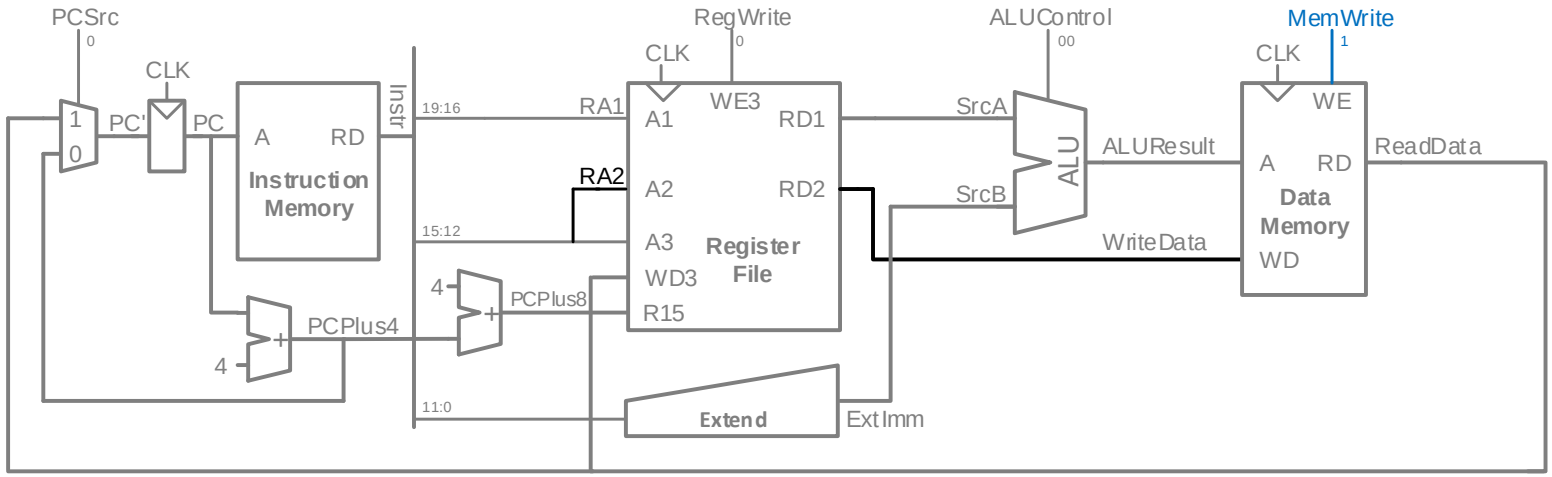
- **Source:** R15 must be available in Register File
  - **PC** is read as the current **PC plus 8**
- **Destination:** Be able to write result to PC

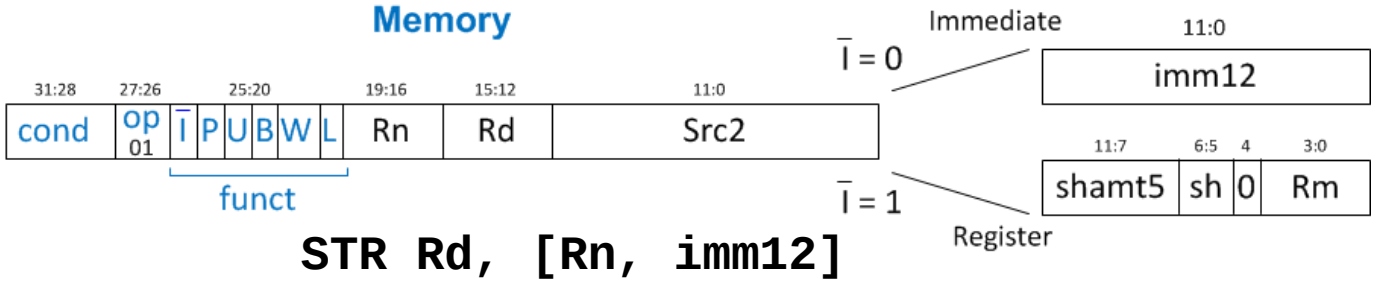


Single-Cycle Datapath: STR

Expand datapath to handle STR:

- Write data in Rd to memory





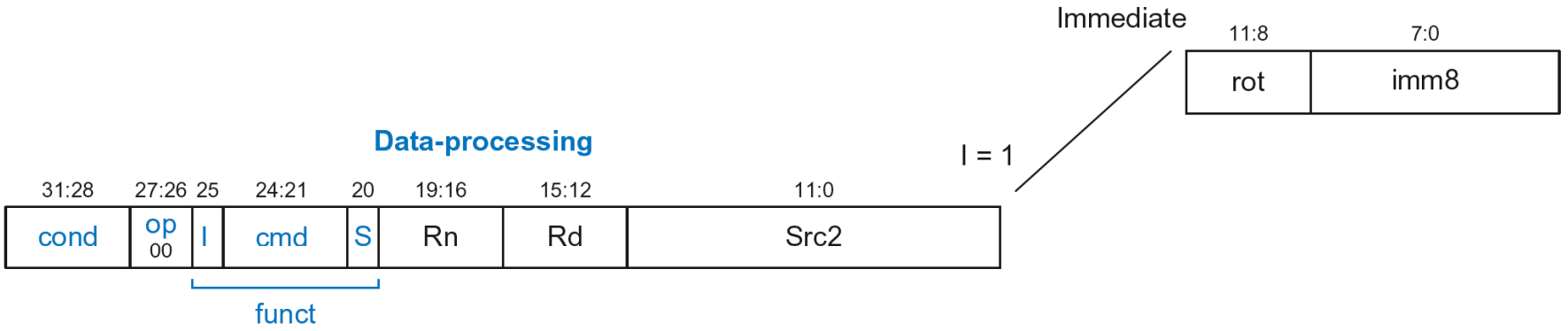
`STR Rd, [Rn, imm12]`

ELSEVIER

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

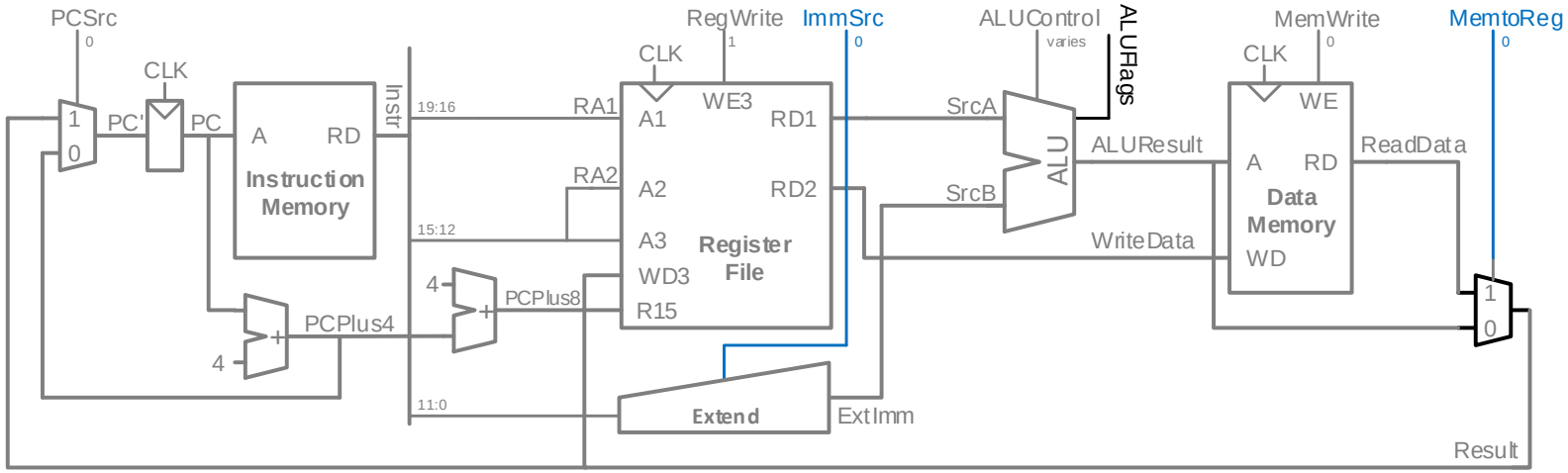


`ADD Rd, Rn, imm8`

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

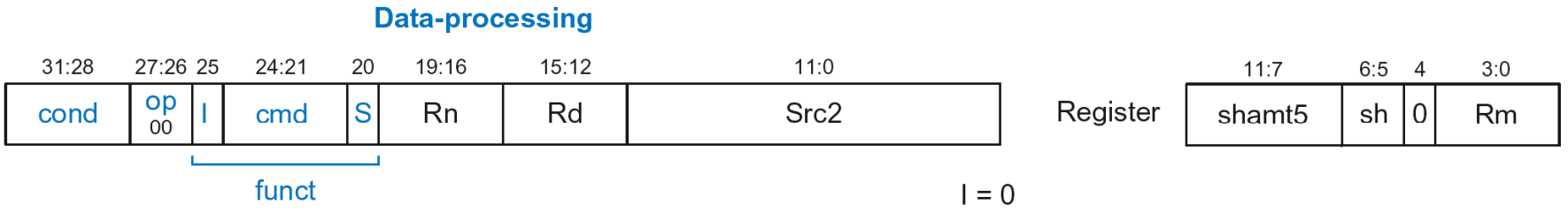


`ADD Rd, Rn, imm8`

Single-Cycle Datapath: Data-processing

With register Src2:

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



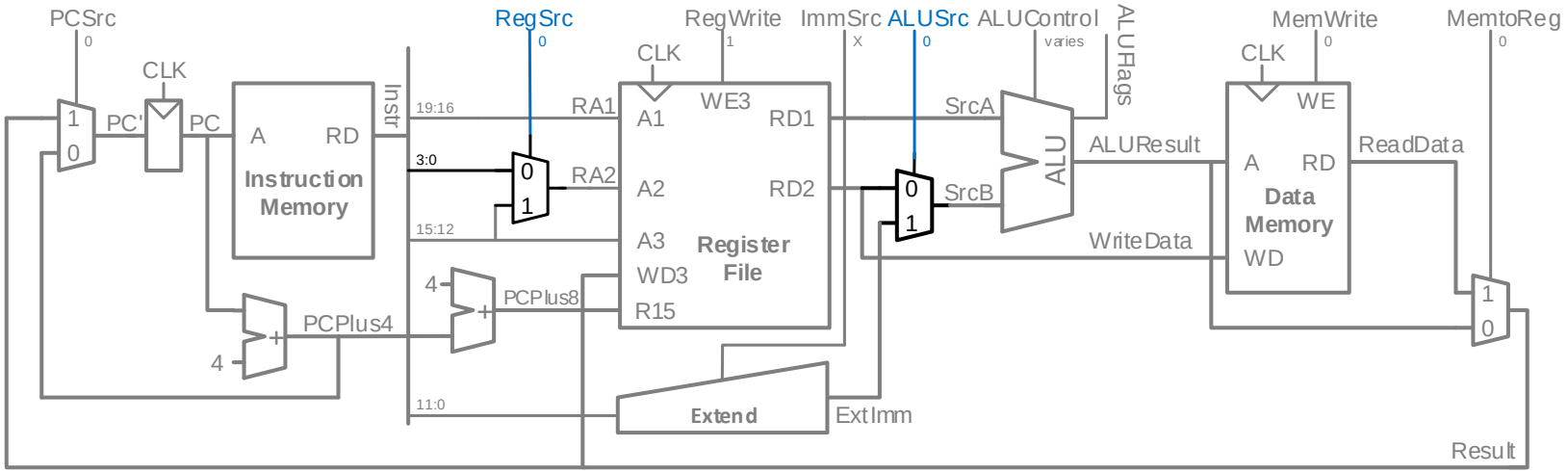
ADD Rd, Rn, Rm

ELSEVIER

Single-Cycle Datapath: Data-processing

With register Src2:

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



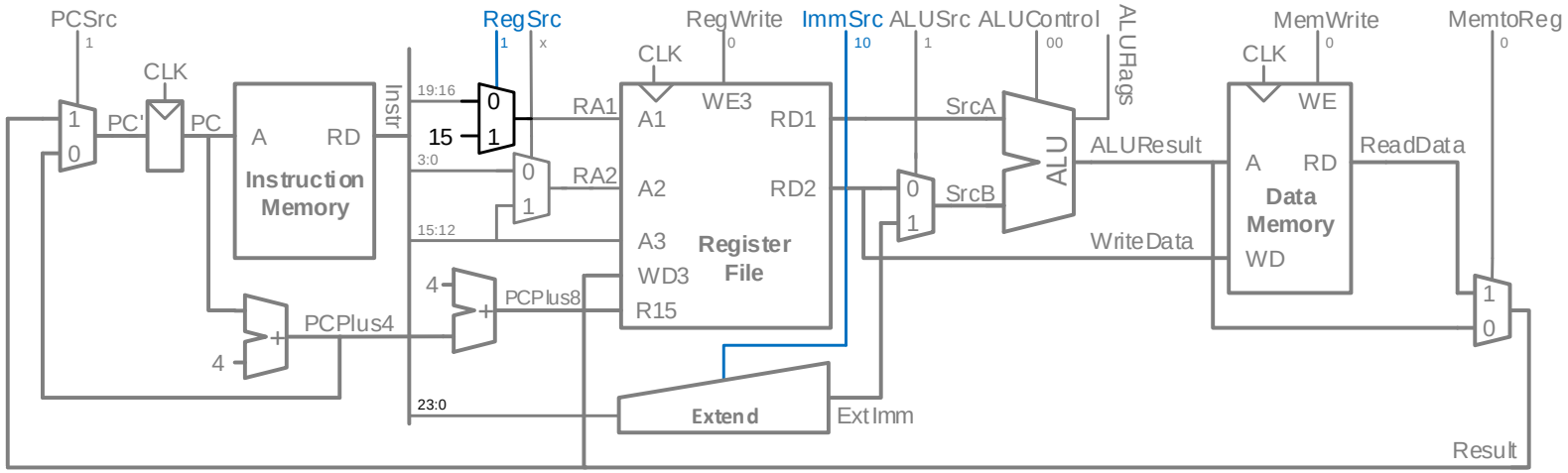
ADD Rd, Rn, Rm

Single-Cycle Datapath: B

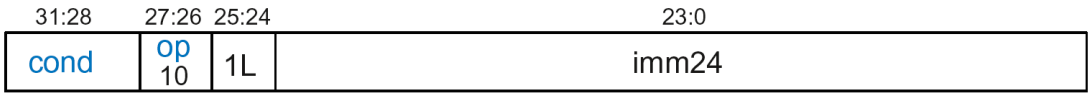
Calculate branch target address:

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

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

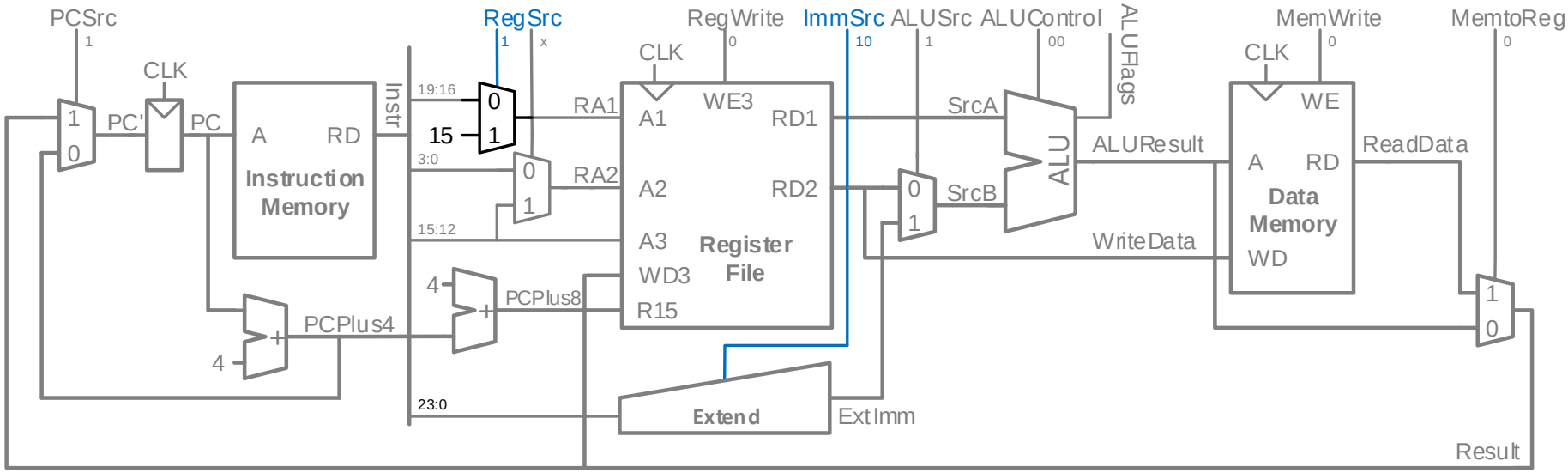


Branch



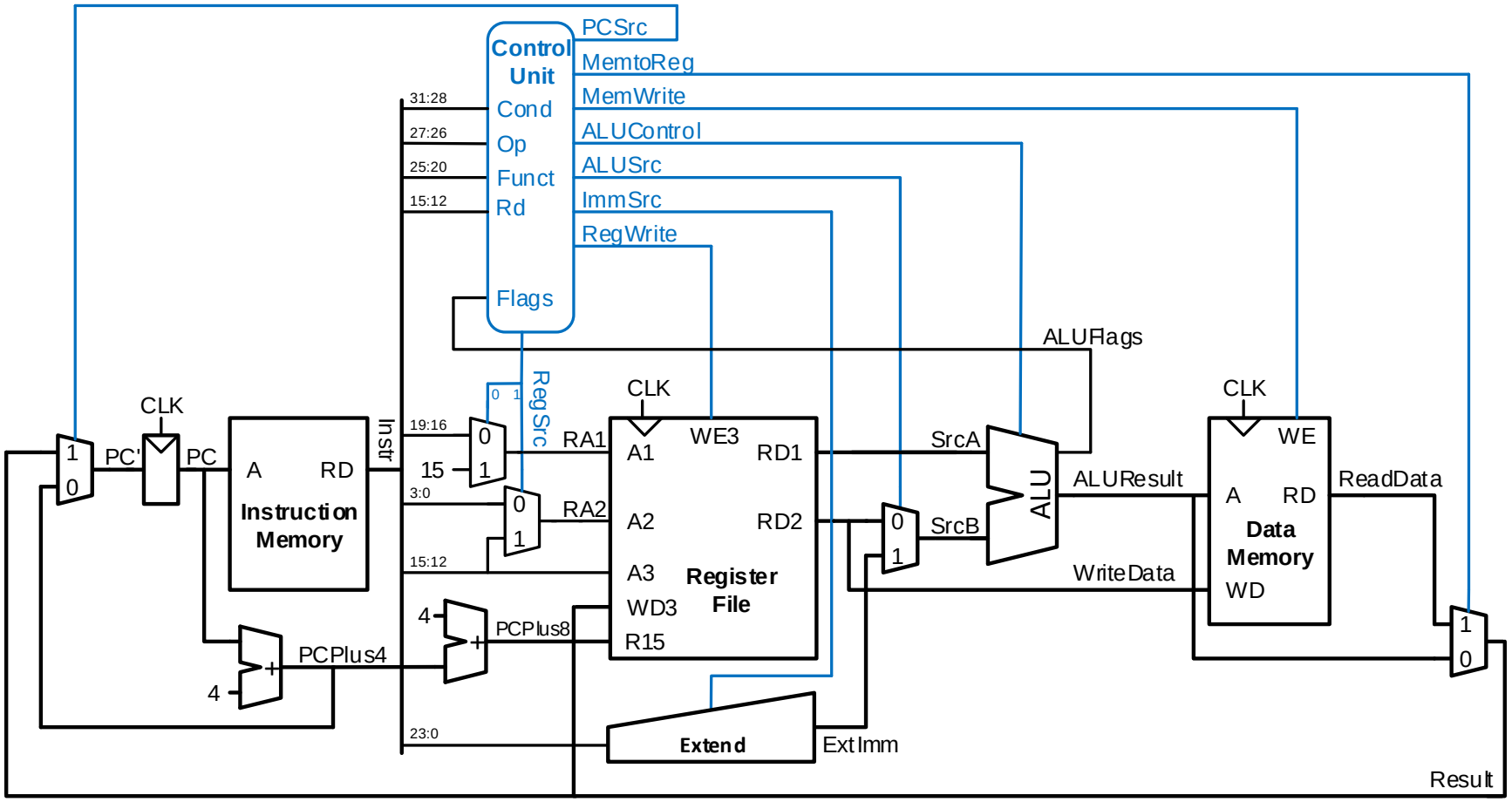
`B Label`

Single-Cycle Datapath: ExtImm

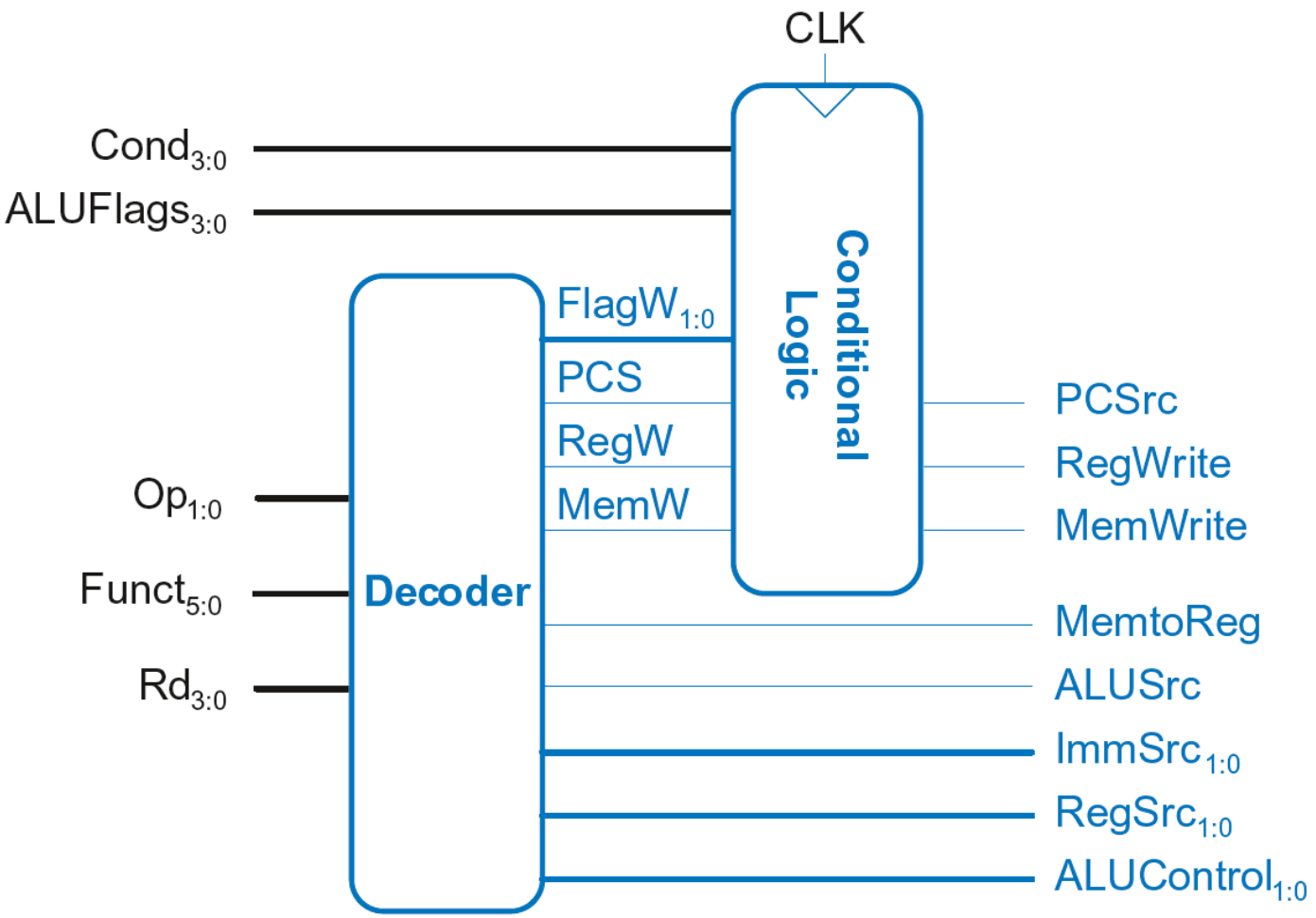


$ImmSrc_{1:0}$	ExtImm	Description
00	{24'b0, $Instr_{7:0}$}	Zero-extended *imm8*
01	{20'b0, $Instr_{11:0}$}	Zero-extended *imm12*
10	{6{$Instr_{23}$}, $Instr_{23:0}$}	Sign-extended *imm24*

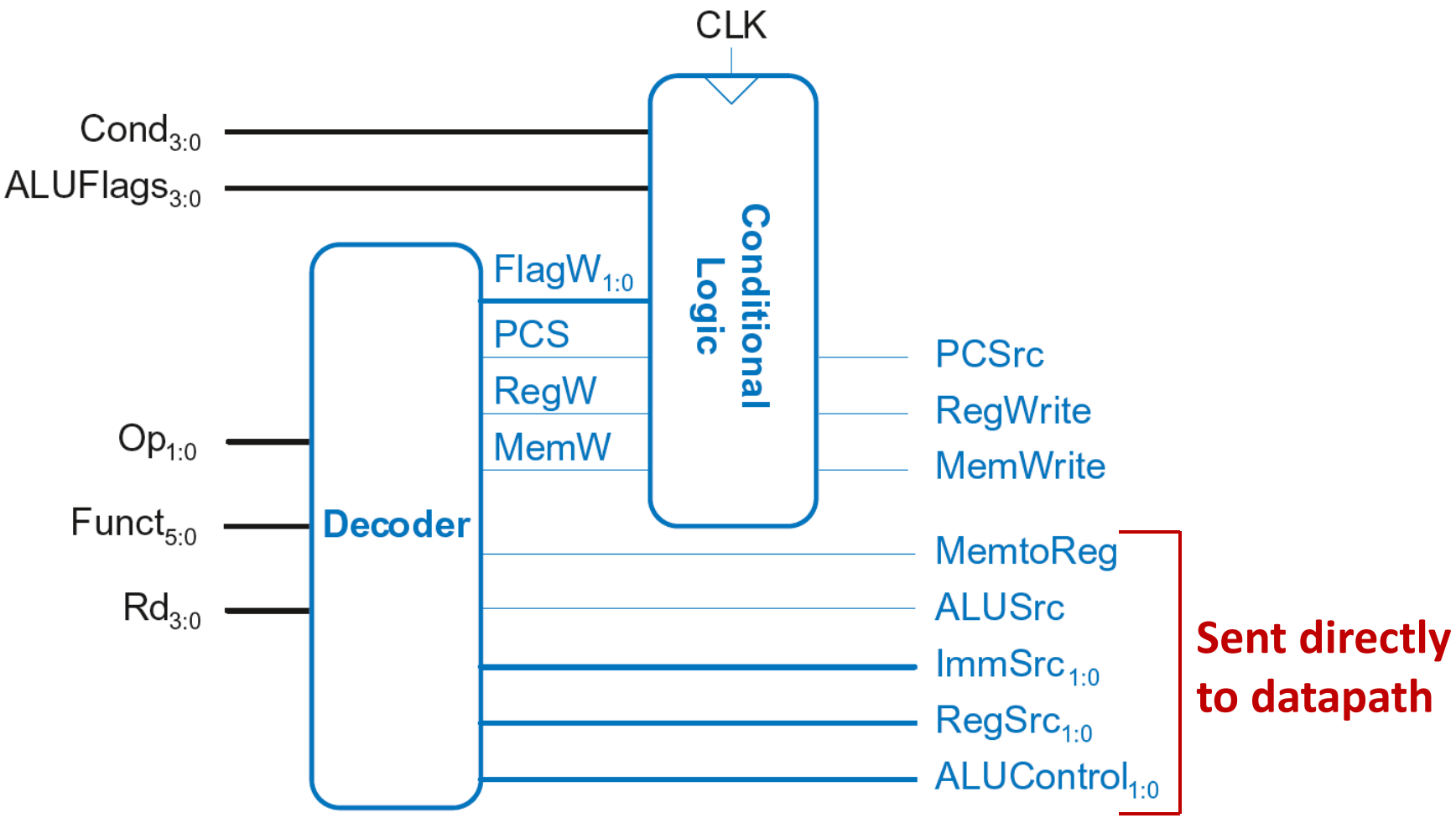
Single-Cycle ARM Processor



Single-Cycle Control

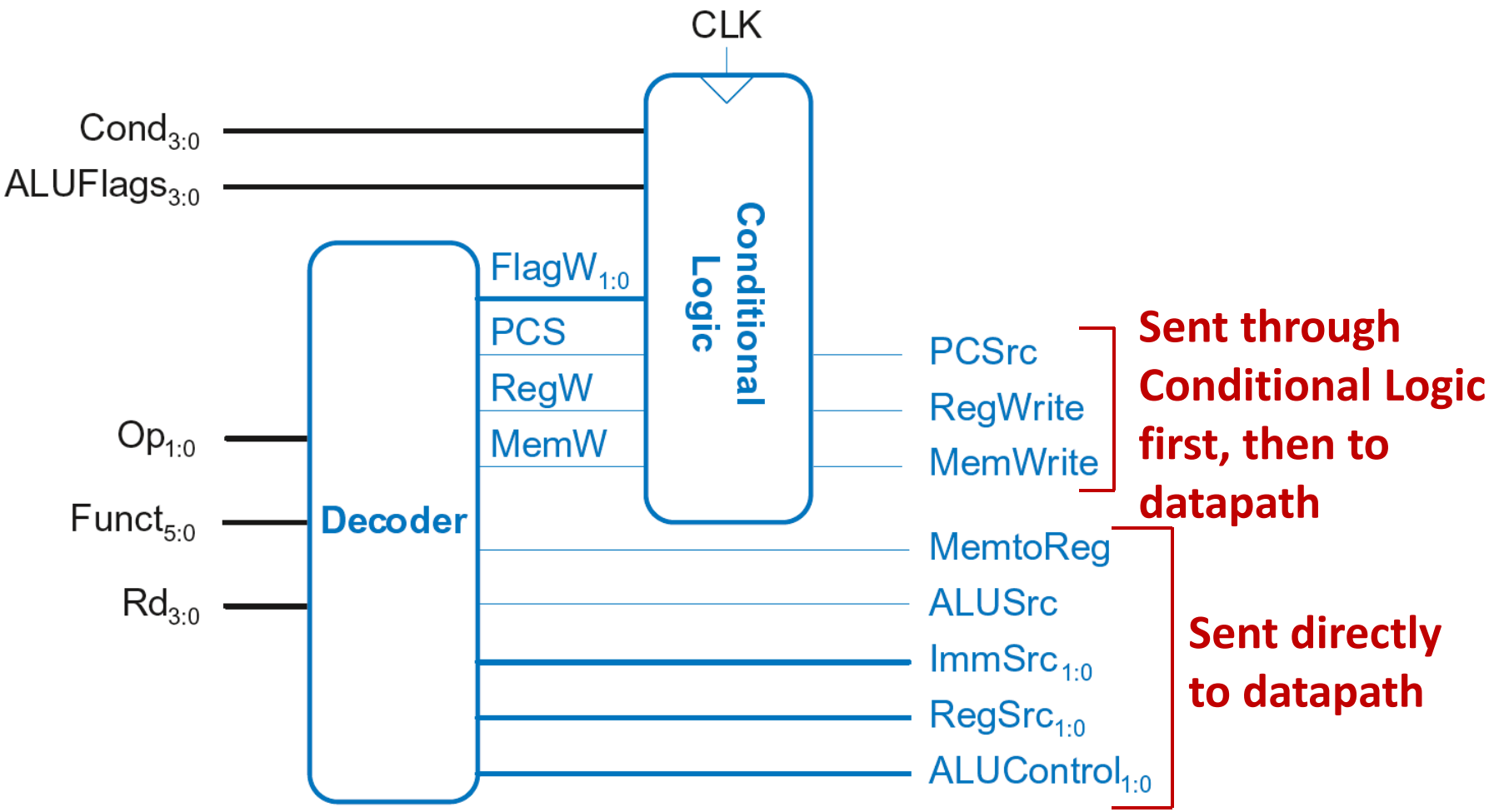


Single-Cycle Control



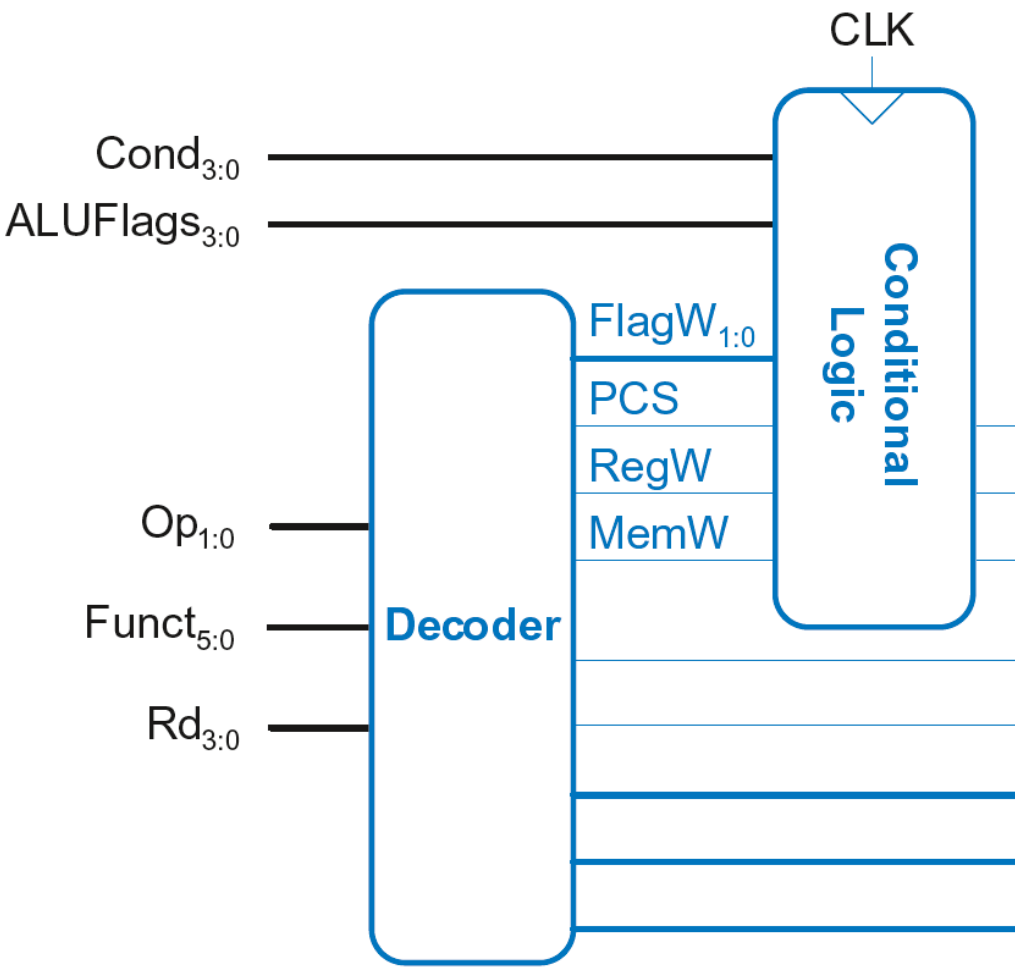
ELSEVIER

Single-Cycle Control



ELSEVIER

Single-Cycle Control

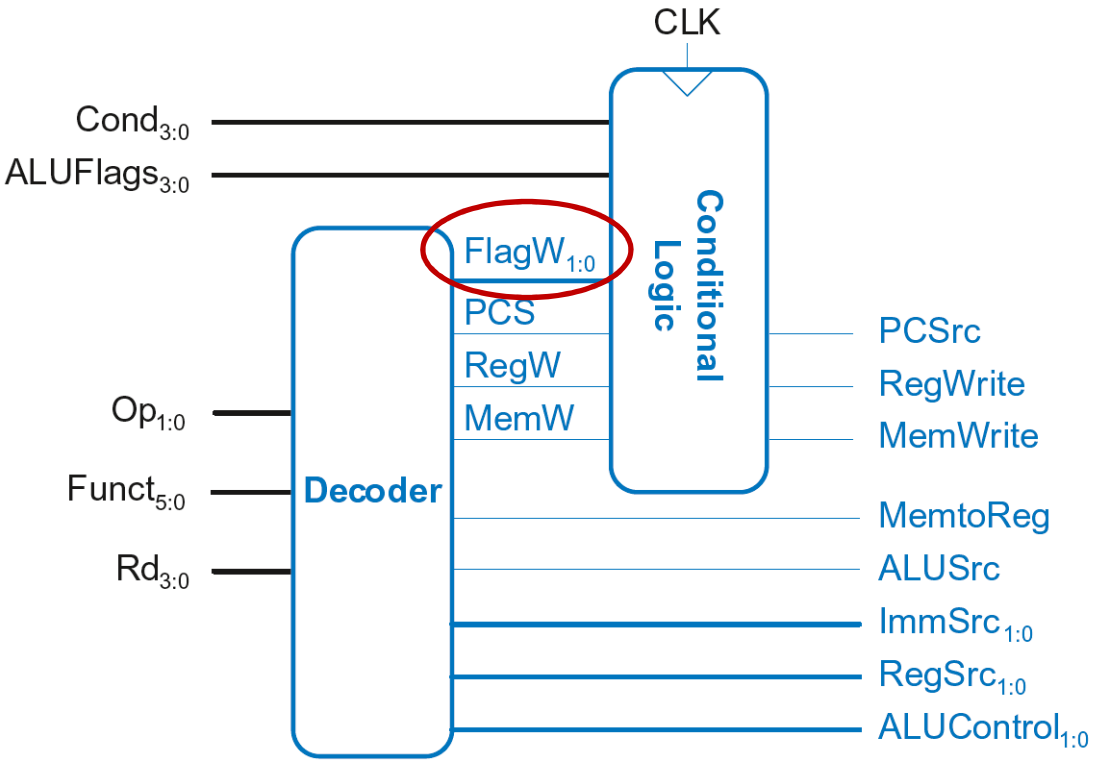


- These signals **change the state** (PC, RF, Memory)
- If instruction shouldn't execute, **forced to 0**

Sent through Conditional Logic first, then to datapath

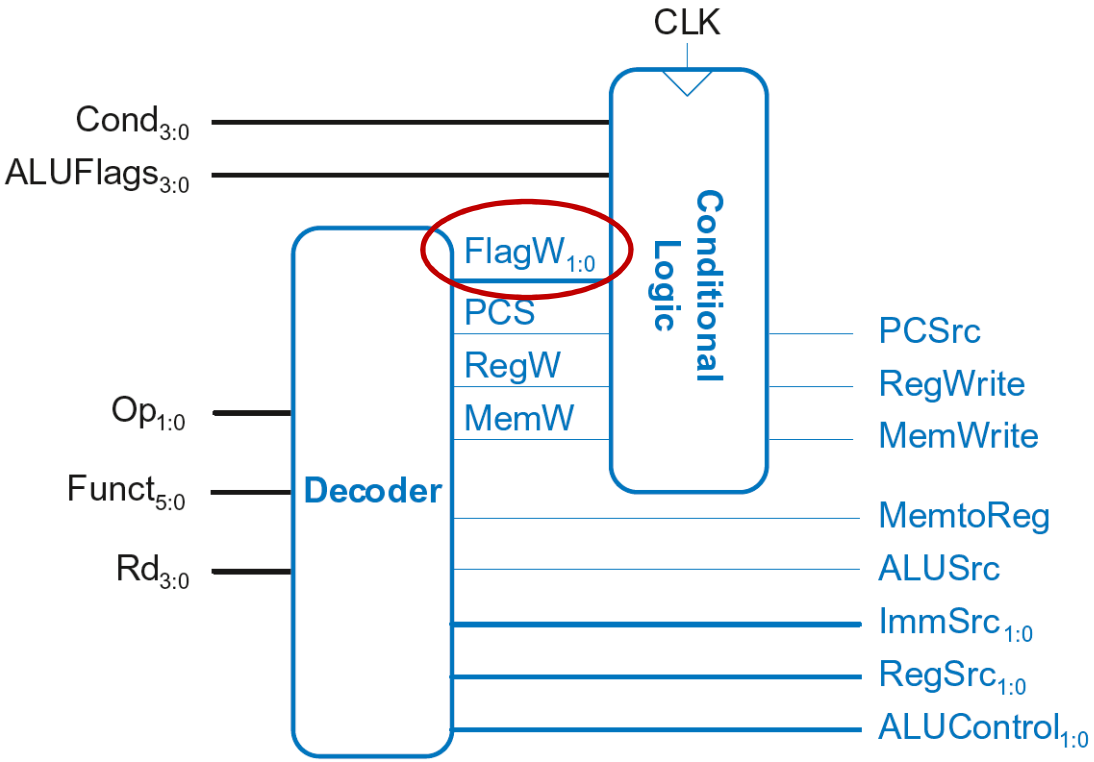
Sent directly to datapath

Single-Cycle Control



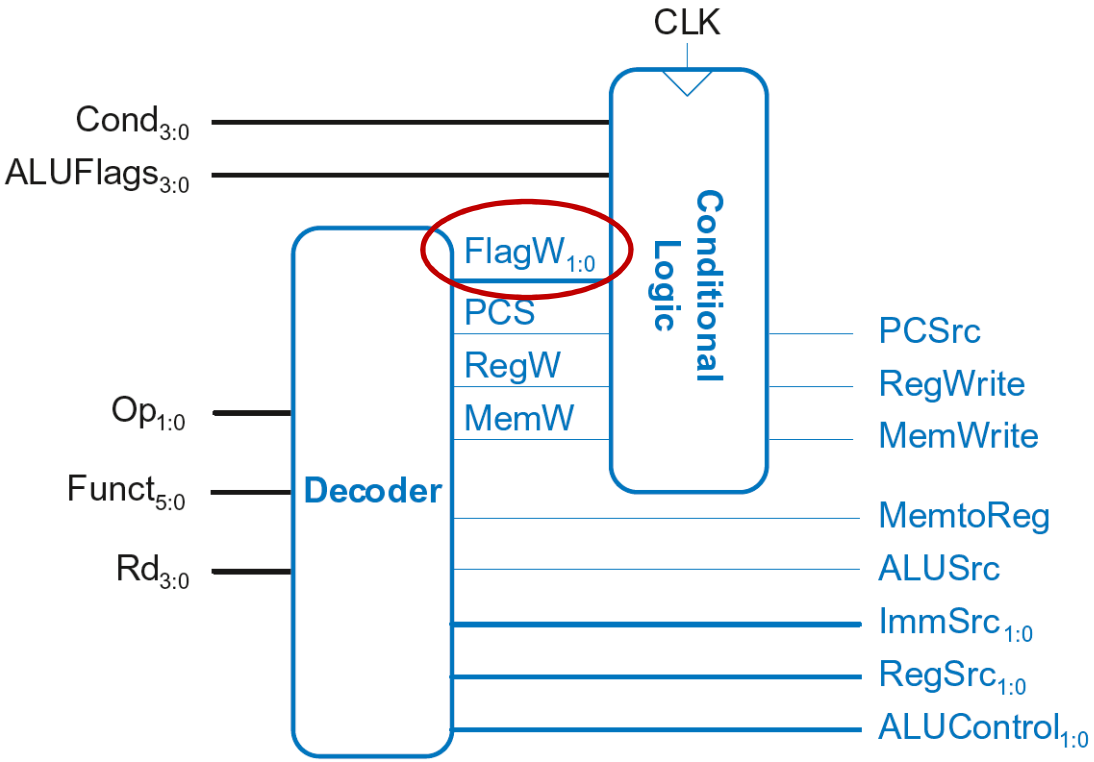
- ***FlagW*$_{1:0}$:** Flag Write signal, asserted when *ALUFlags* should be saved (i.e., on instruction with S=1)

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

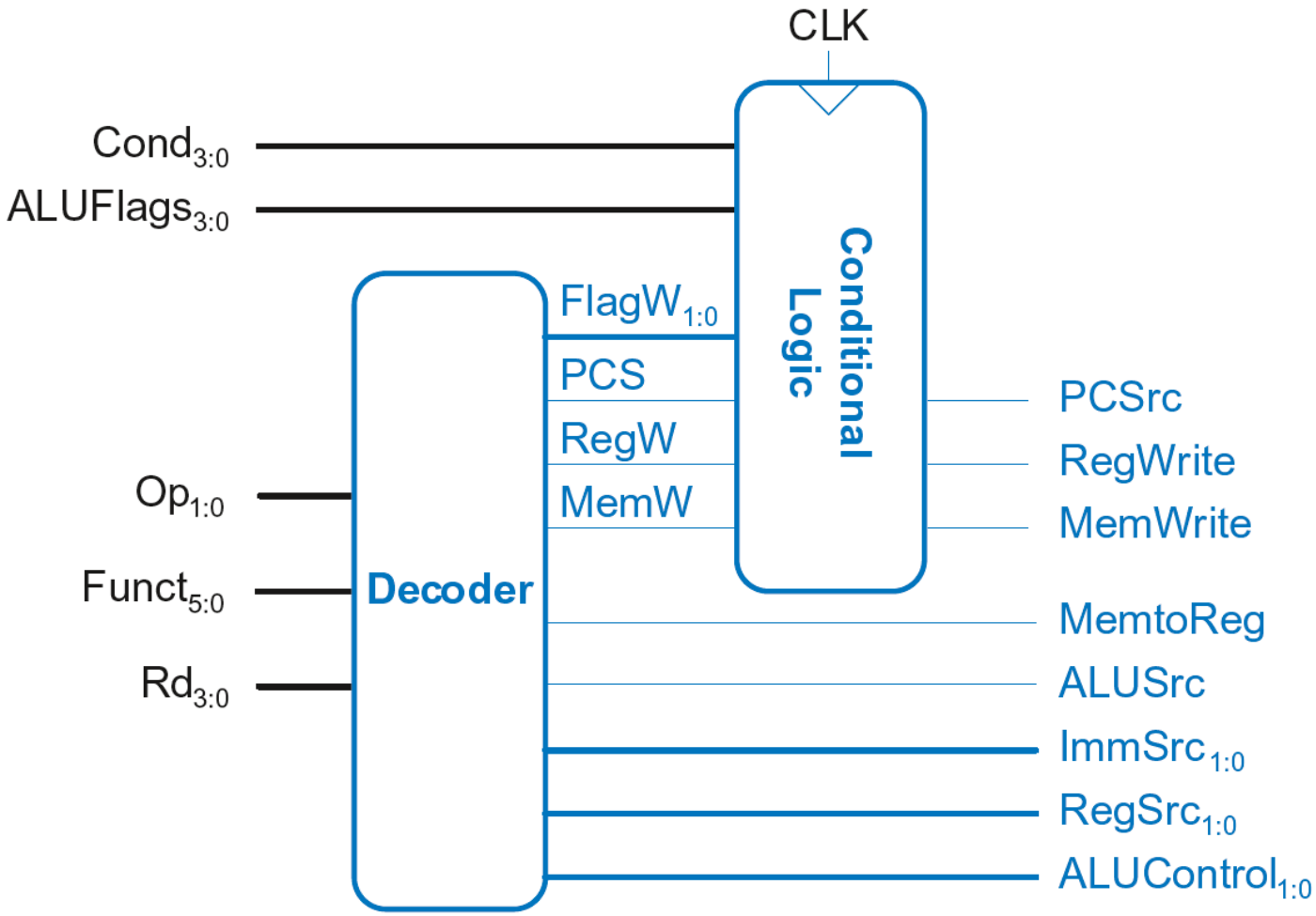


- ***FlagW*$_{1:0}$:** Flag Write signal, asserted when *ALUFlags* should be saved (i.e., on instruction with S=1)
- ADD, SUB update all flags (**NZCV**)
- AND, ORR only update **NZ** flags
- So, two bits needed:

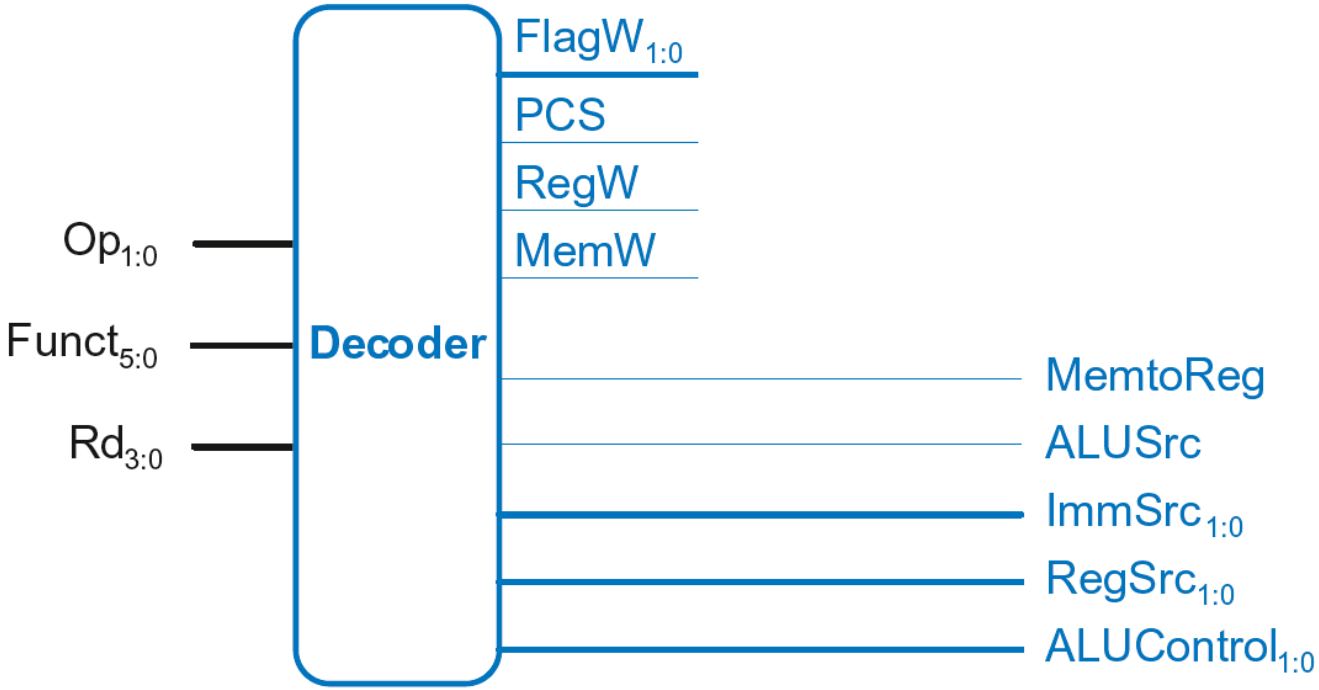
  ***FlagW*$_1$** = 1: *NZ* saved (*ALUFlags*$_{3:2}$ saved)

  ***FlagW*$_0$** = 1: *CV* saved (*ALUFlags*$_{1:0}$ saved)

Single-Cycle Control



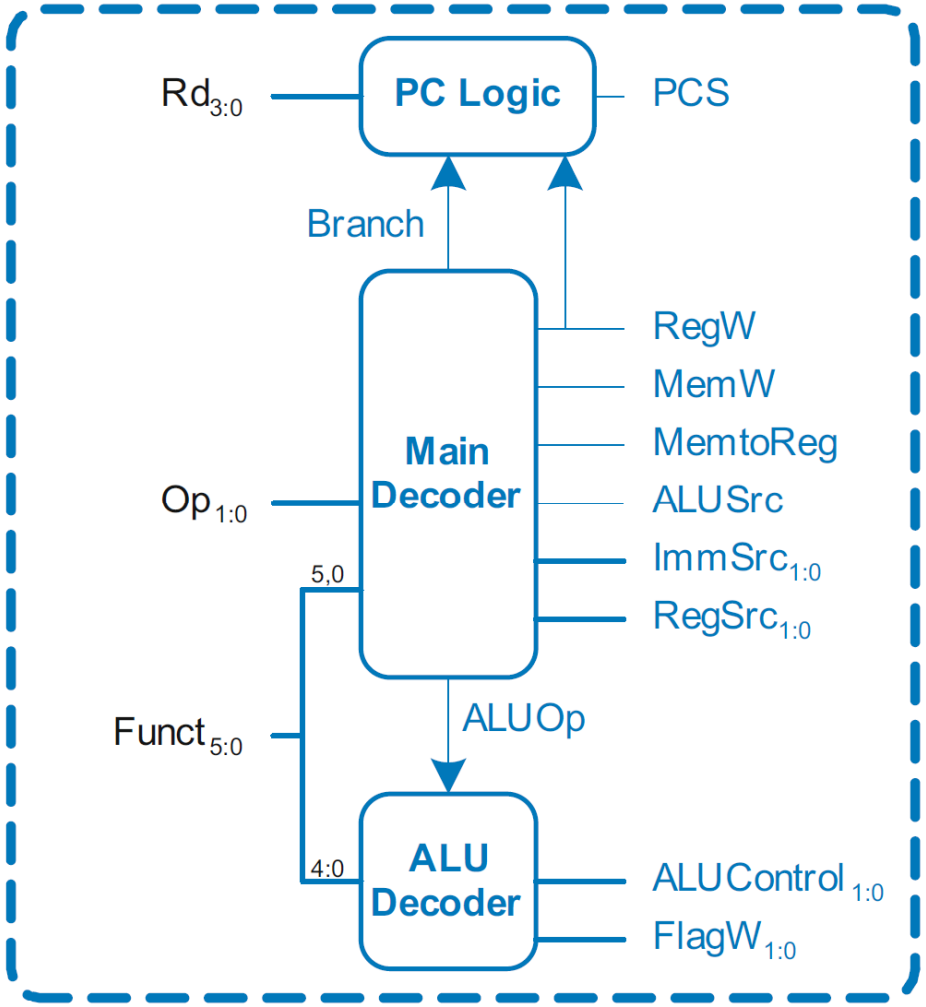
Single-Cycle Control: Decoder



Single-Cycle Control: Decoder

Submodules:

- Main Decoder
- ALU Decoder
- PC Logic

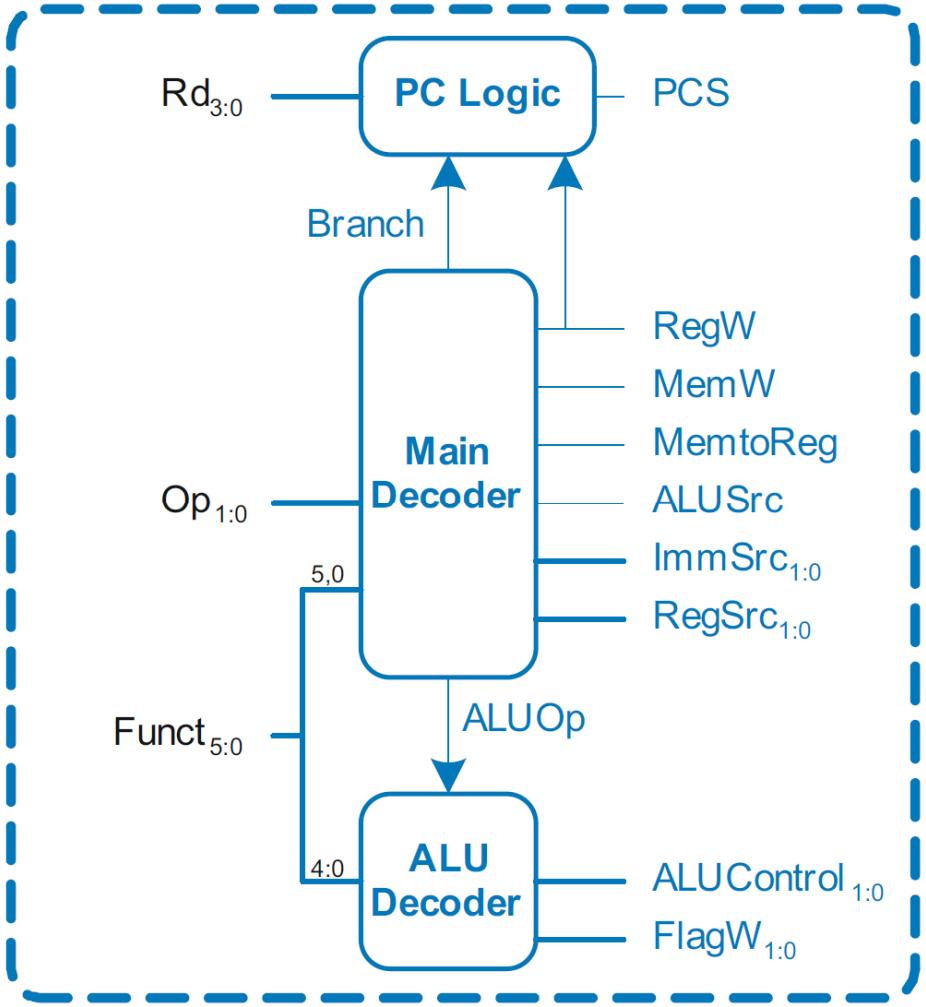


ELSEVIER

Single-Cycle Control: Decoder

Submodules:

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



ELSEVIER

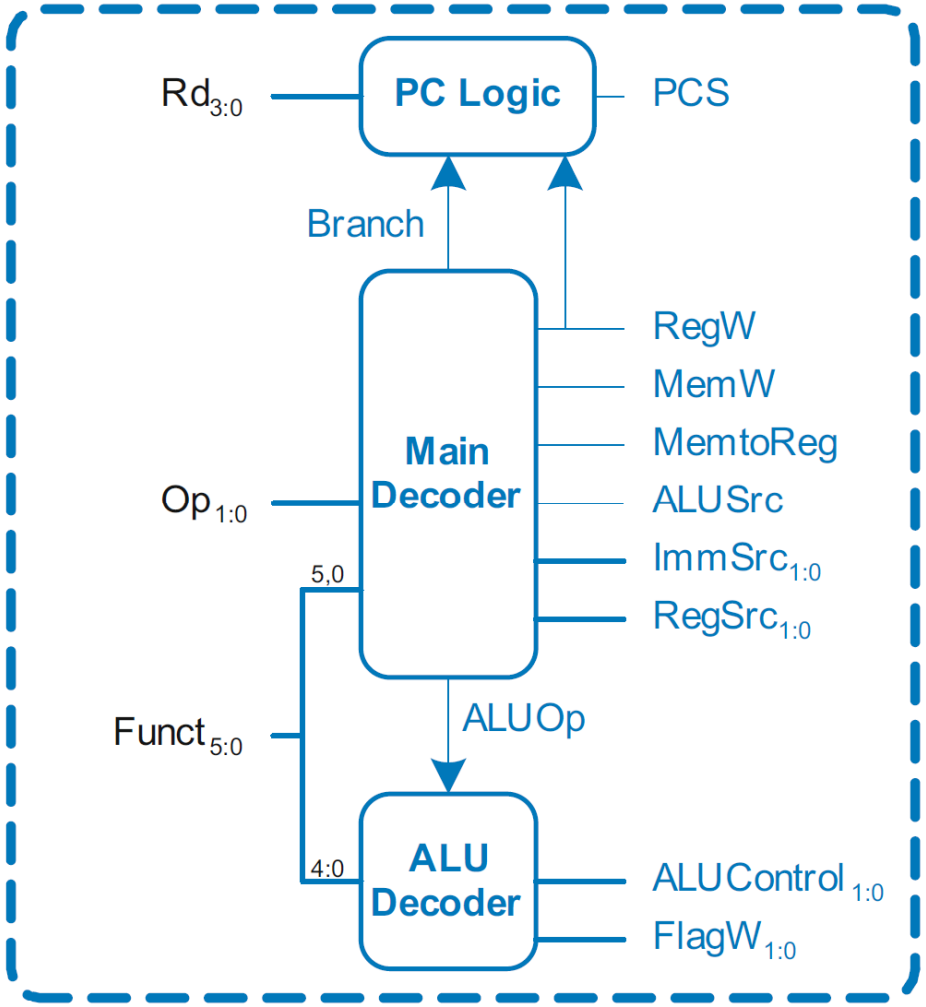
Control Unit: Main Decoder

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:

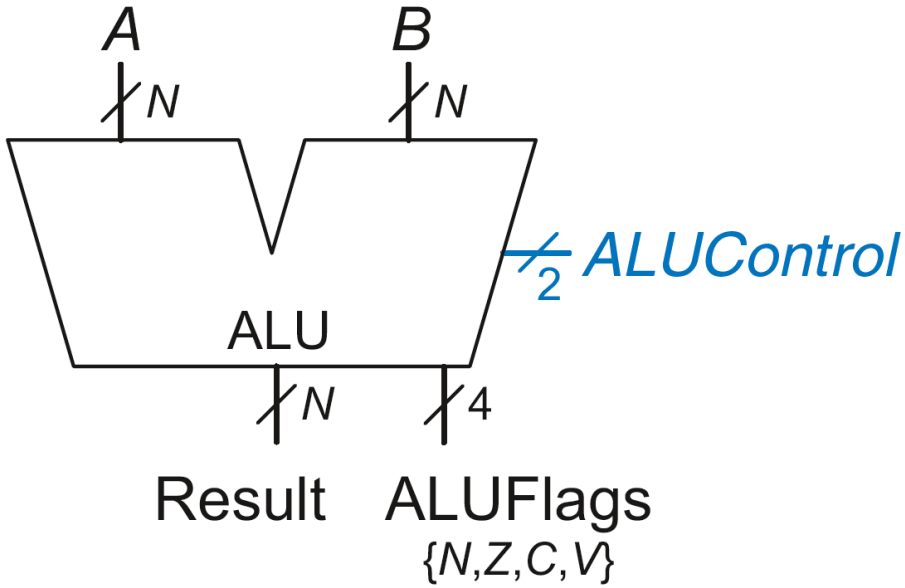
- Main Decoder
- **ALU Decoder**
- PC Logic



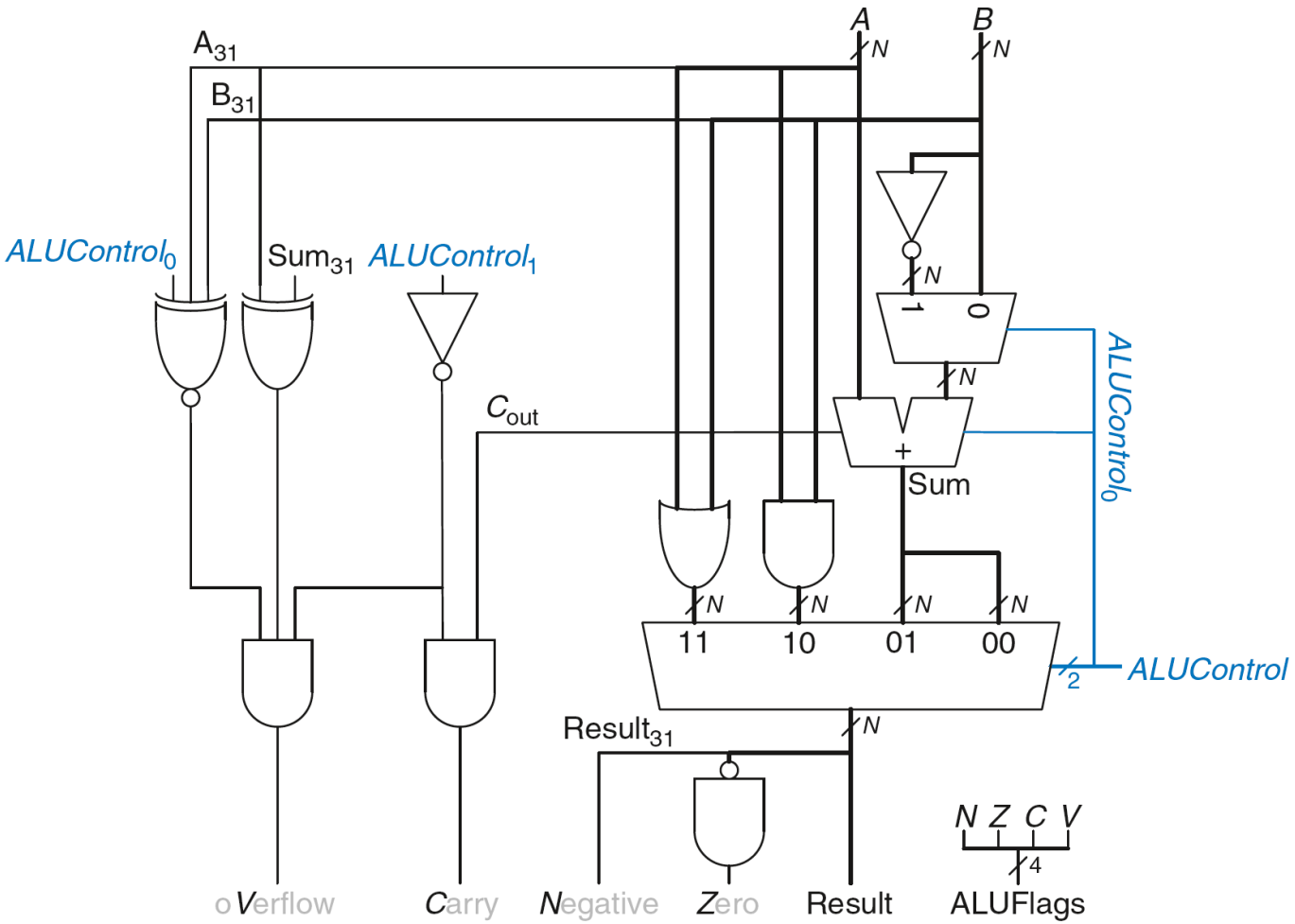
ELSEVIER

Review: ALU

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



Review: ALU

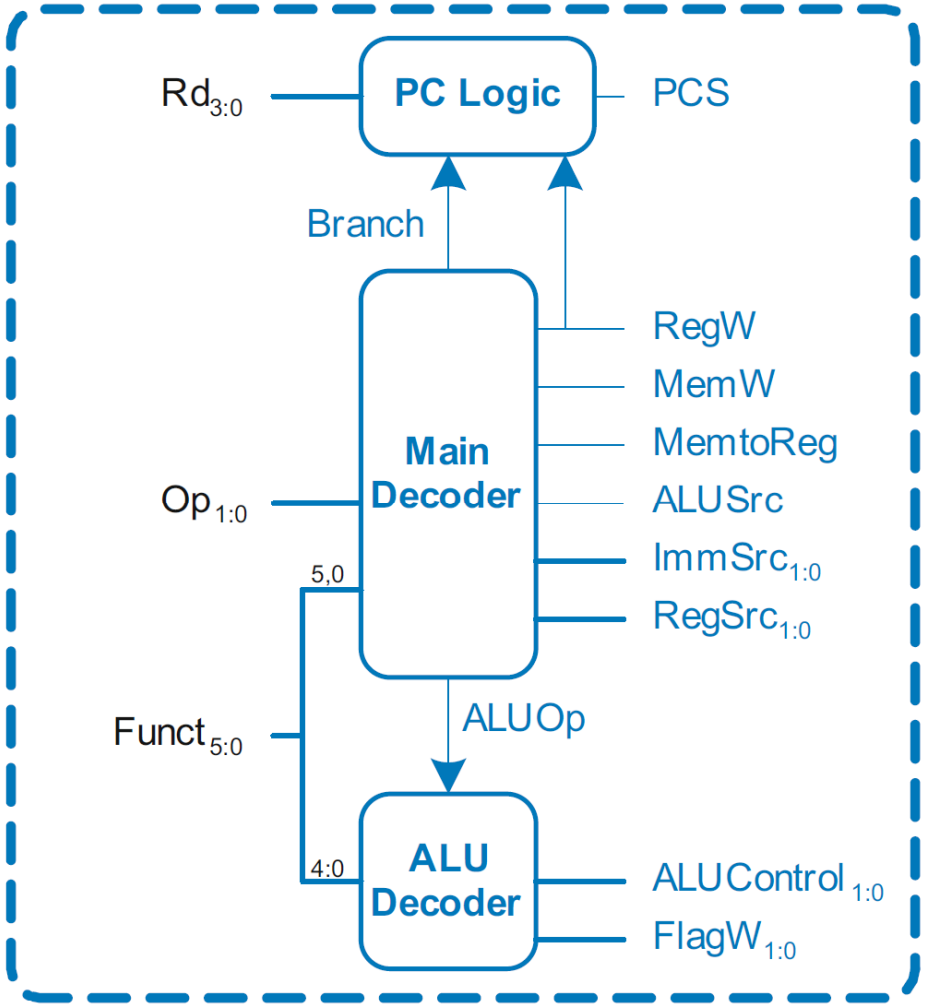


ELSEVIER

Single-Cycle Control: Decoder

Submodules:

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



ELSEVIER

Control Unit: ALU Decoder

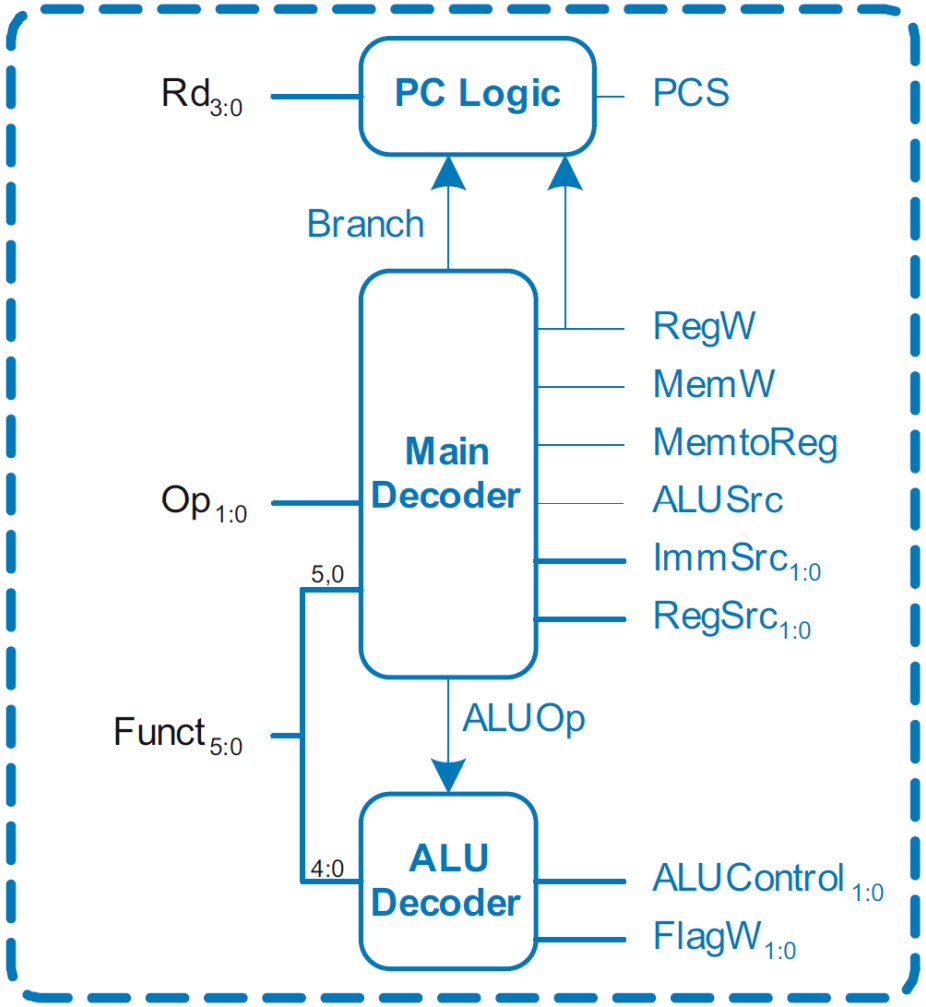
ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00	00
1	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10

- $FlagW_1$ = 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**

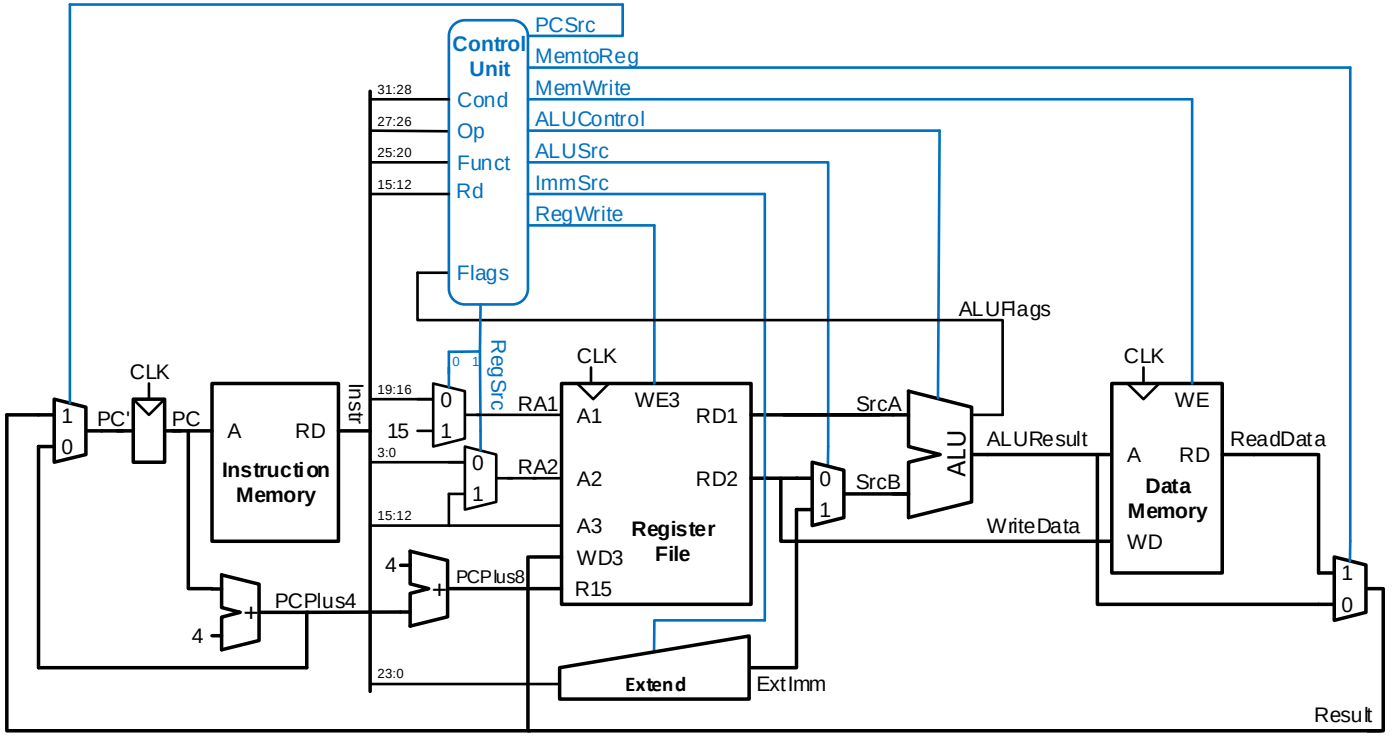


ELSEVIER

Single-Cycle Control: PC Logic

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

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

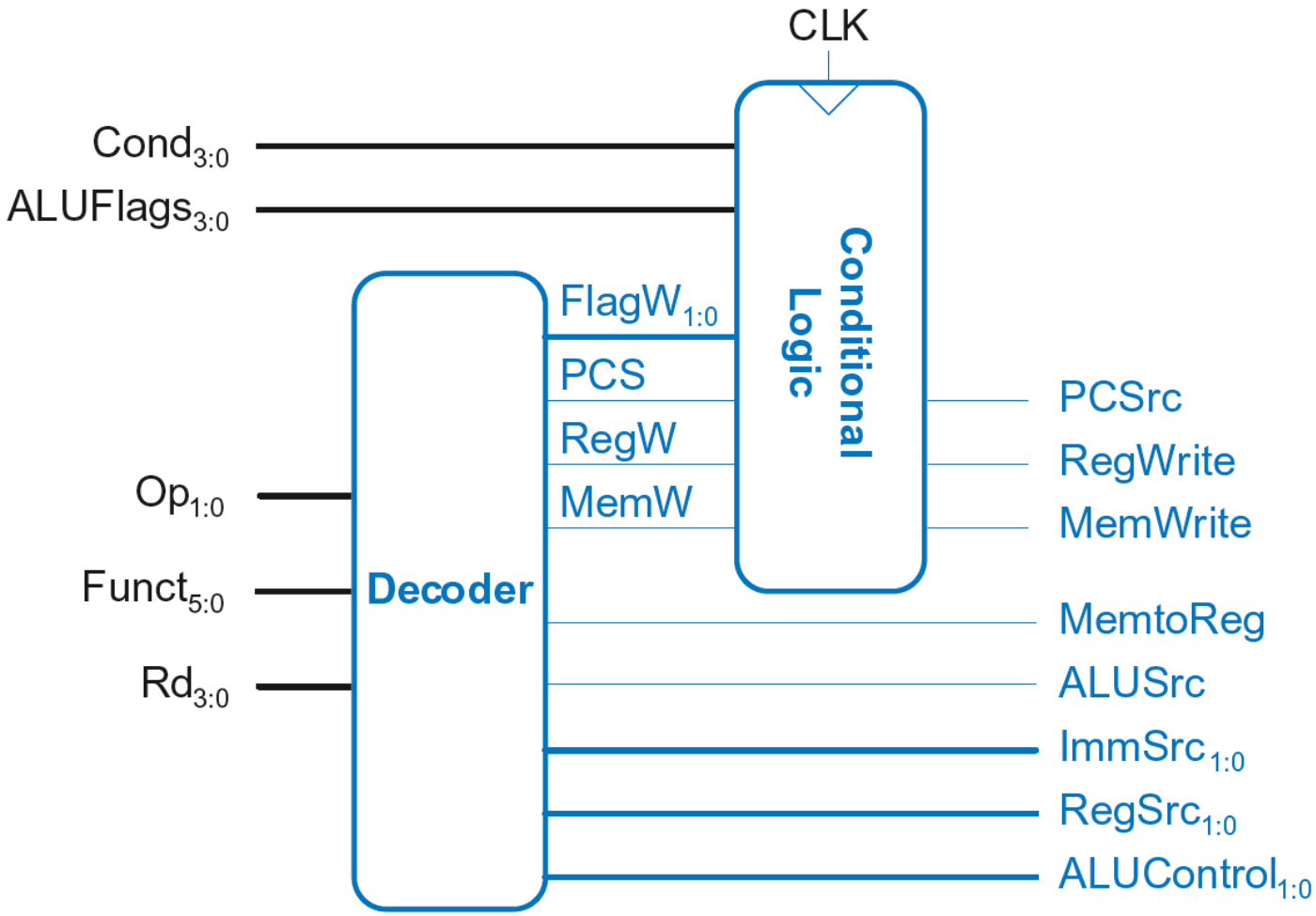


If instruction is executed: ***PCSrc* = *PCS***

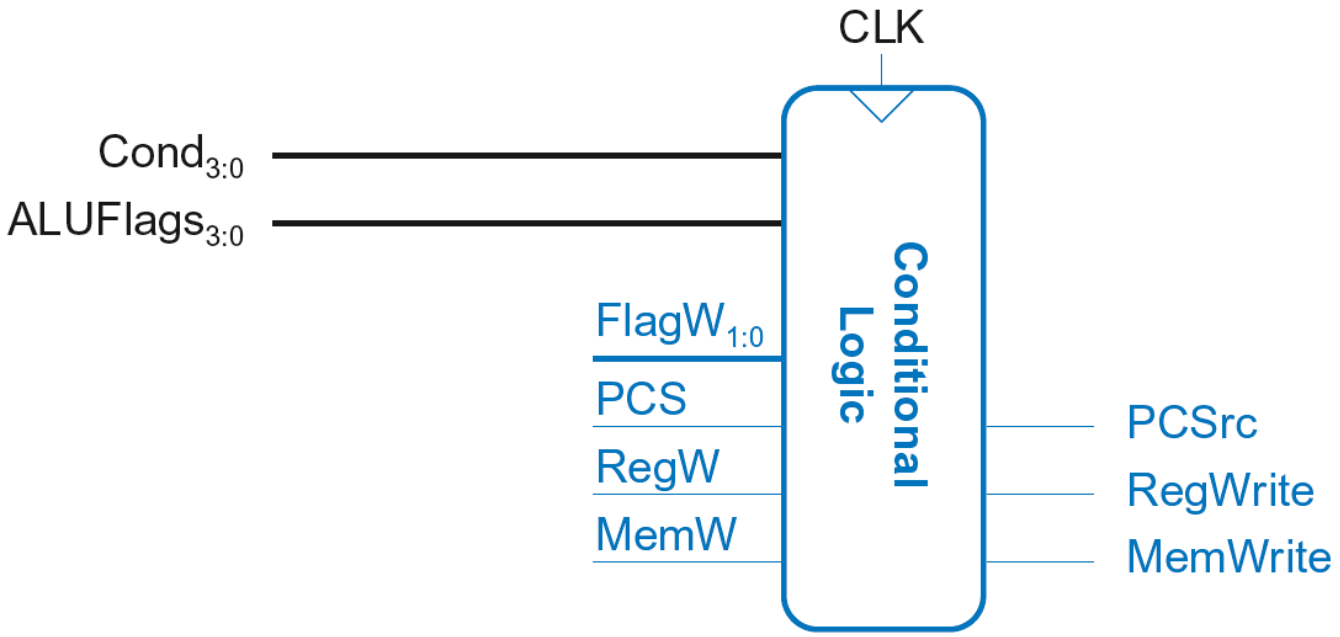
Else ***PCSrc* = 0 (i.e., PC = PC + 4)**

ELSEVIER

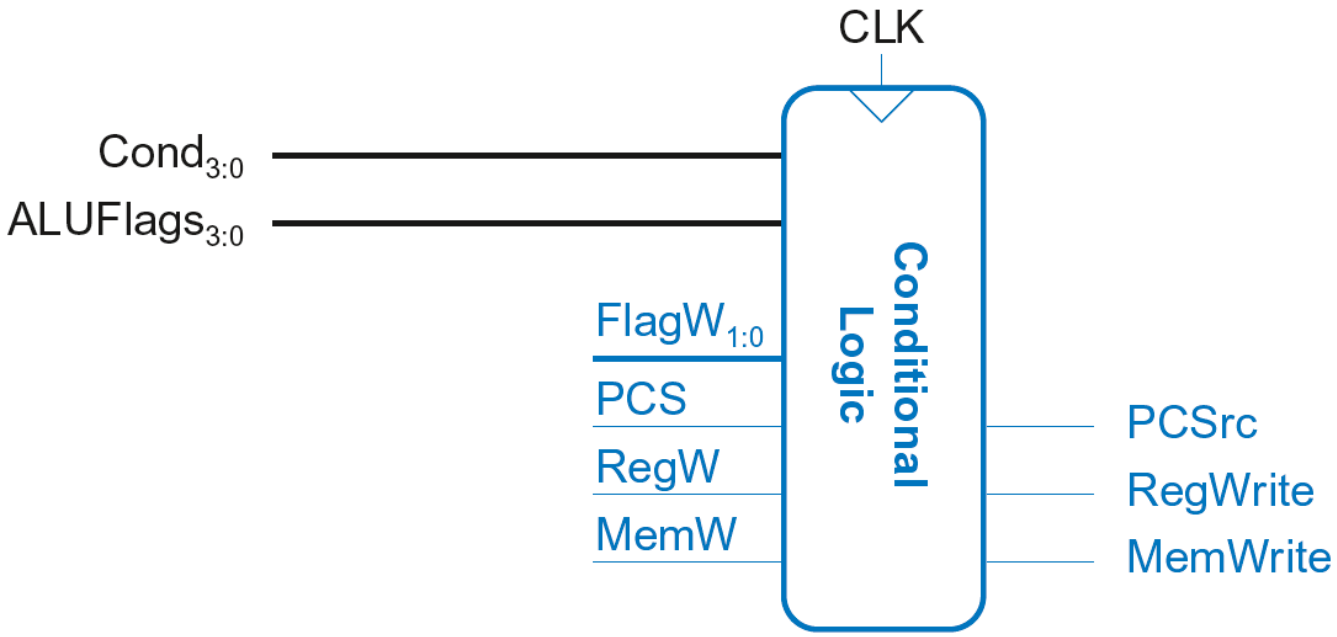
Single-Cycle Control



Single-Cycle Control: Cond. Logic



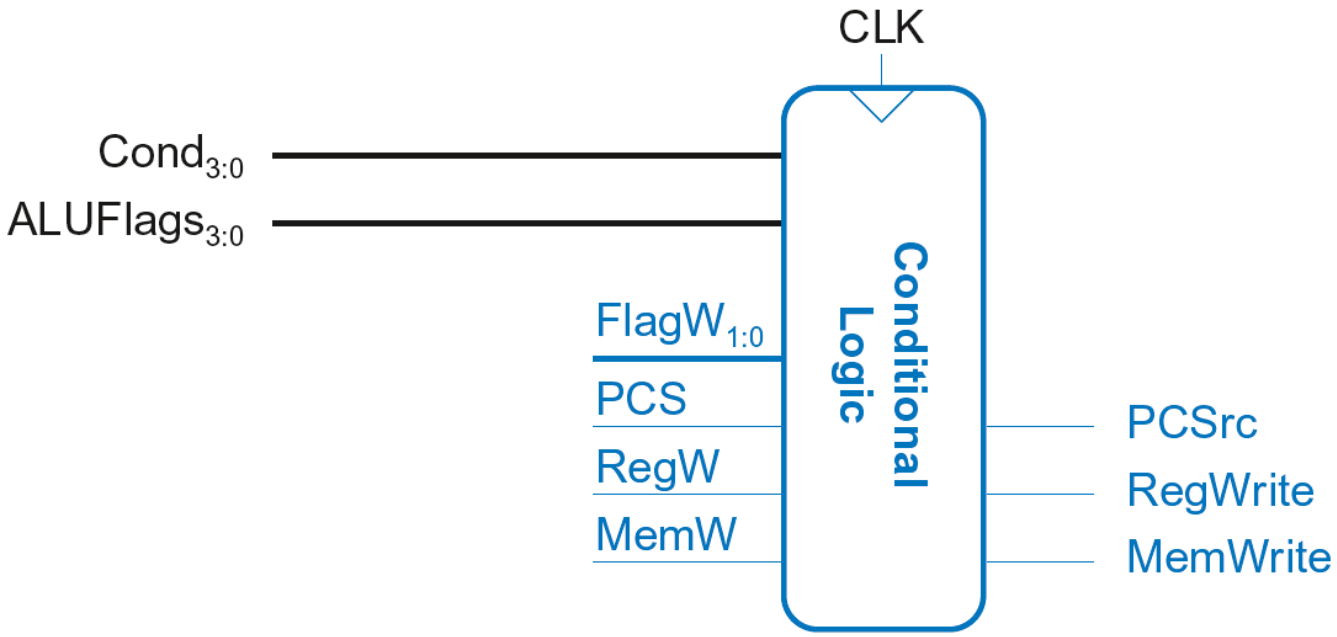
Conditional Logic



Function:

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

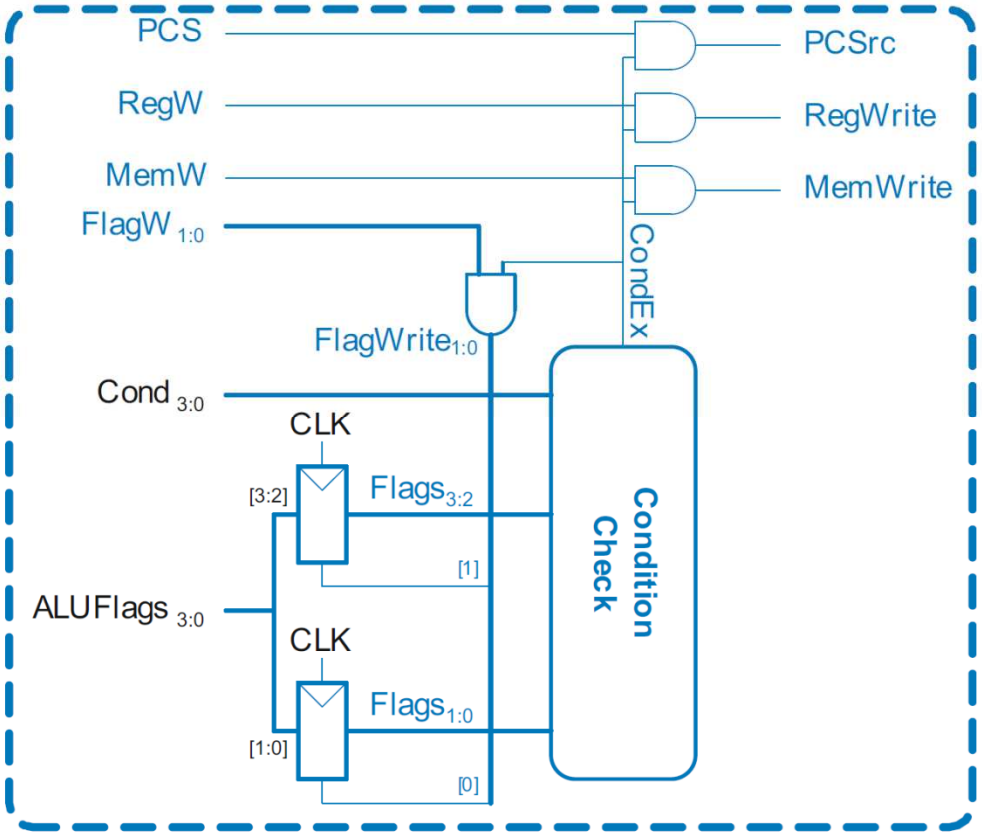
Conditional Logic



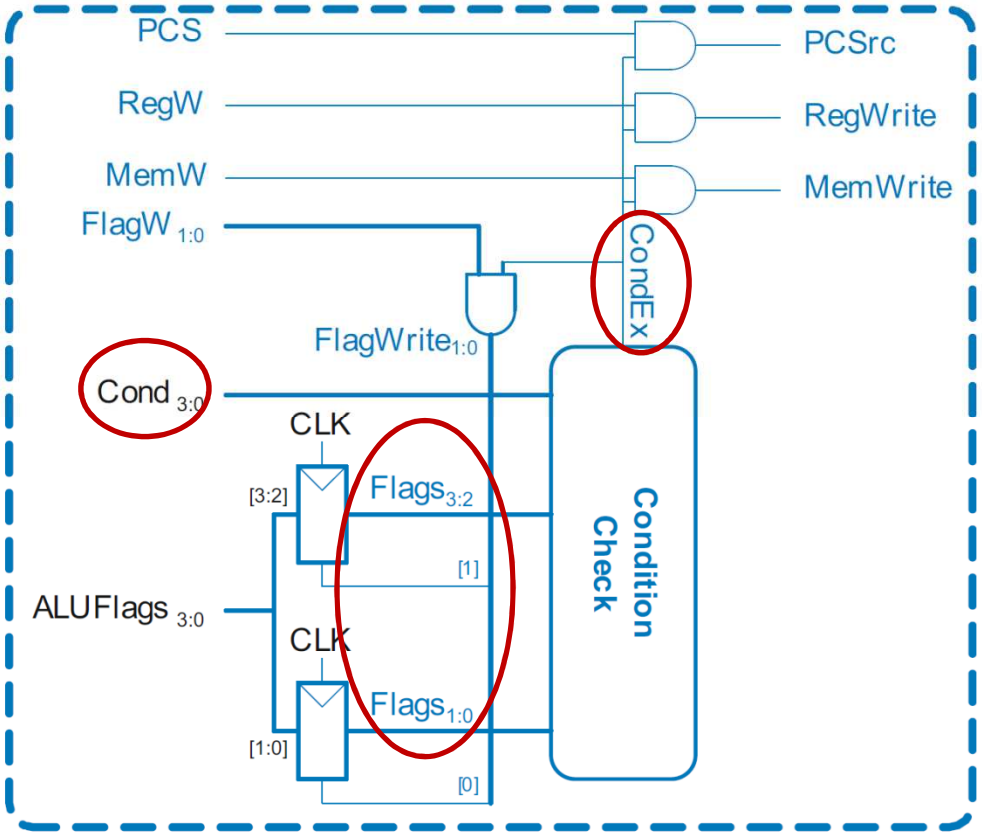
Function:

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

Single-Cycle Control: Conditional Logic



Conditional Logic: Conditional Execution

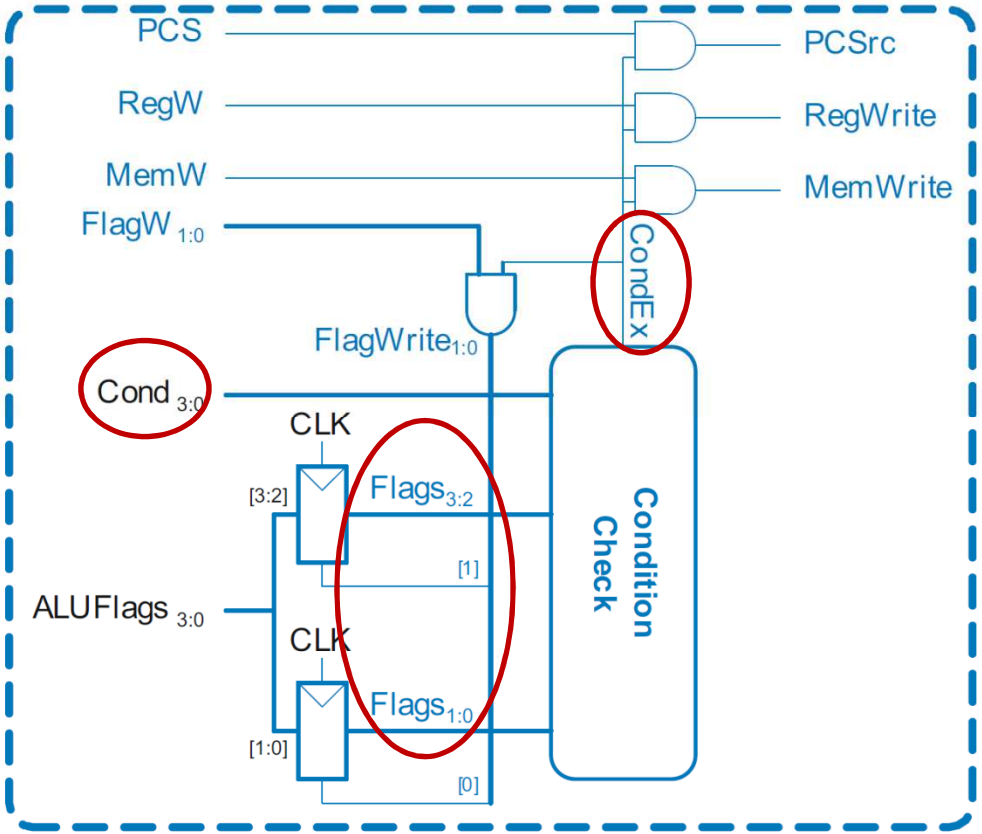


Depending on condition mnemonic (***Cond*$_{3:0}$**) and condition flags (***Flags*$_{3:0}$**) the instruction is executed (***CondEx*** = 1)

ELSEVIER

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)

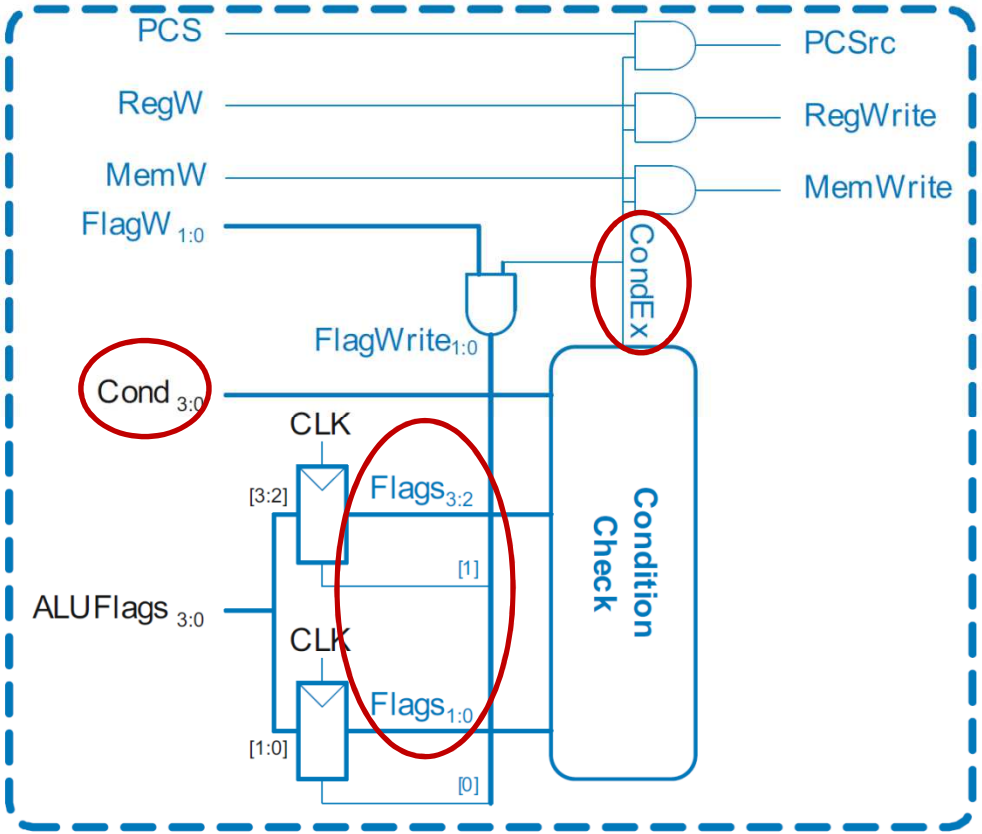
ELSEVIER

Review: Condition Mnemonics

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

Conditional Logic: Conditional Execution

$Flags_{3:0}$ = NZCV

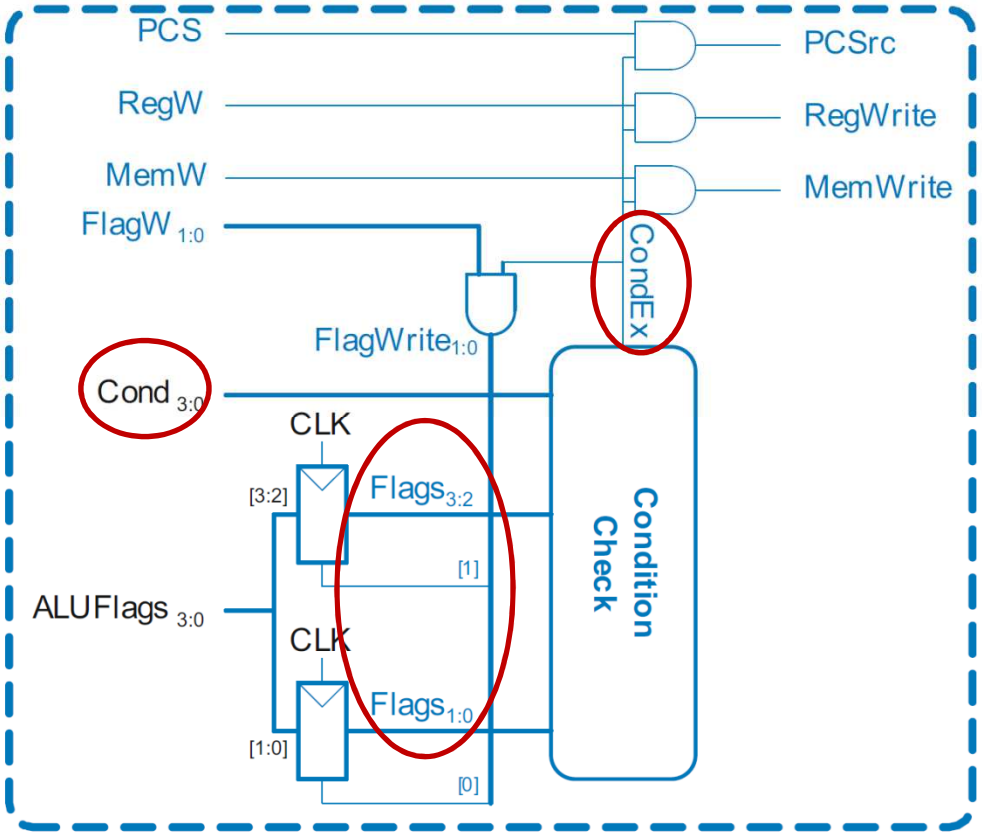


Example: `AND R1, R2, R3`

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

Conditional Logic: Conditional Execution

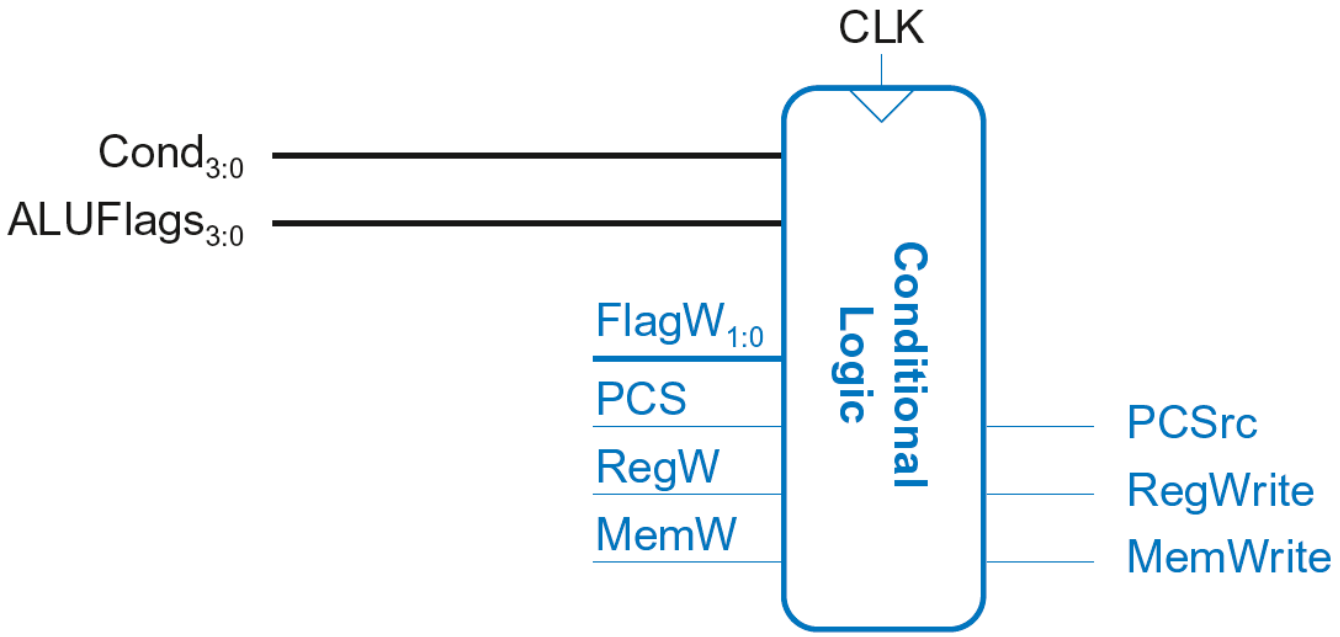
$Flags_{3:0}$ = NZCV



Example: EOREQ R5, R6, R7

$Cond_{3:0}$=0000 (EQ): if **$Flags_{3:2}$**=0100 => **CondEx** = 1

Conditional Logic

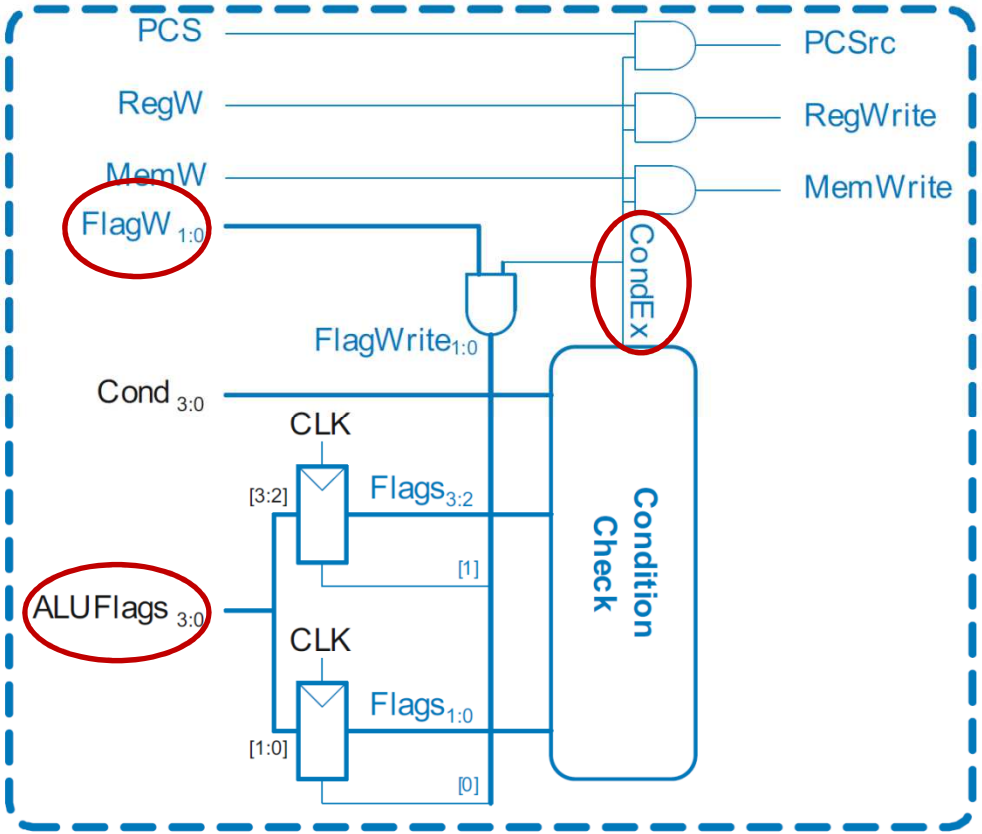


Function:

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

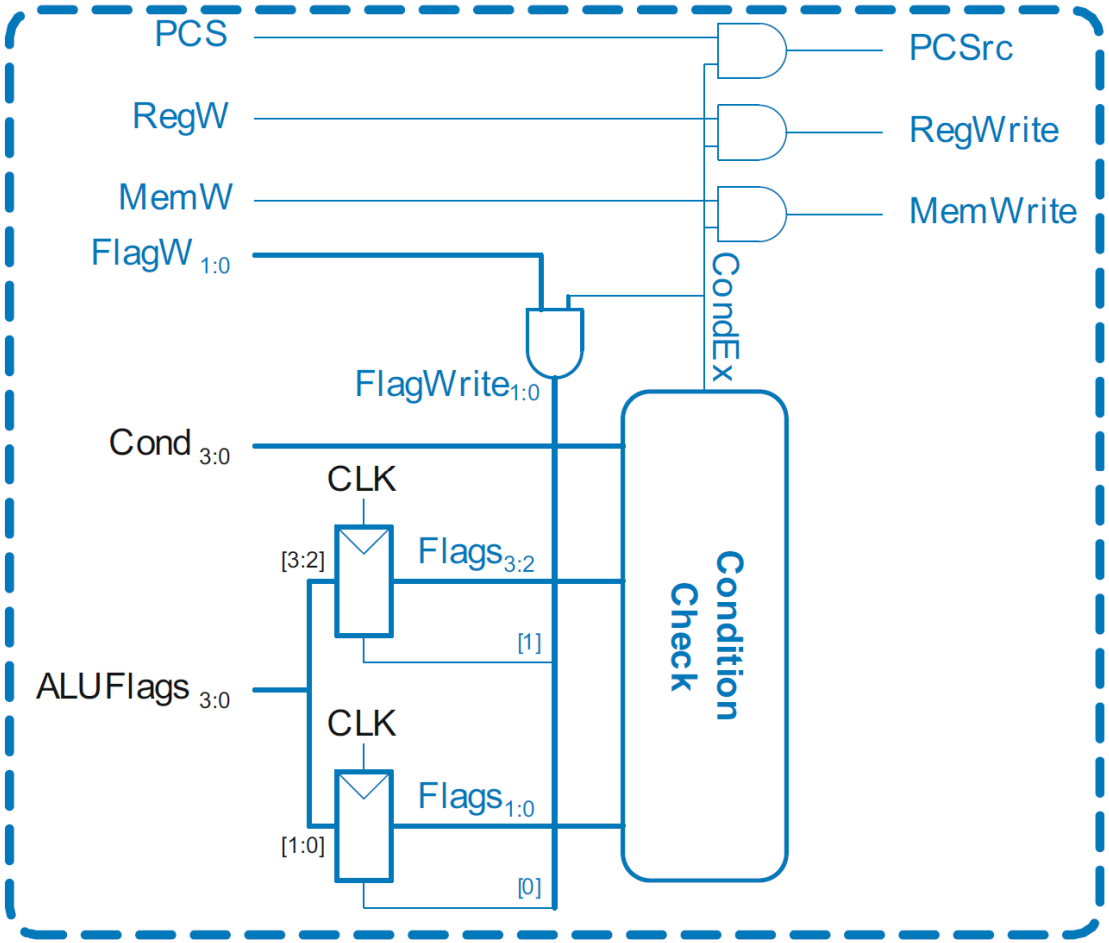
Flags$_{3:0}$ = NZCV



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

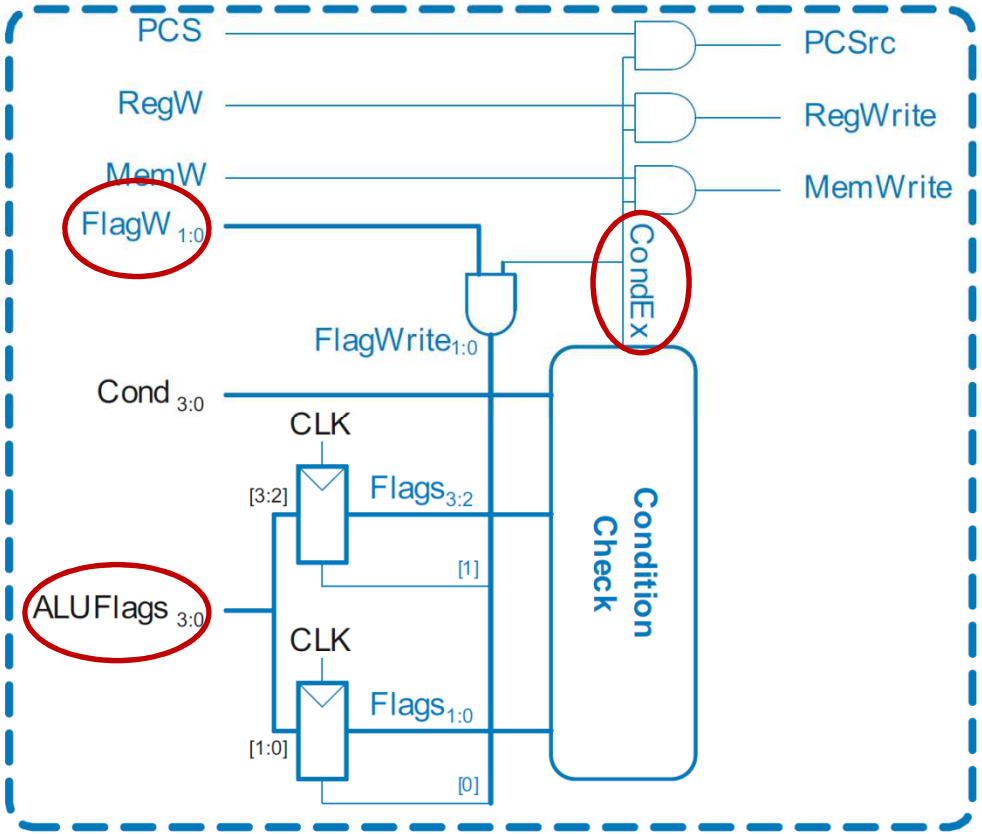
Review: ALU Decoder

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

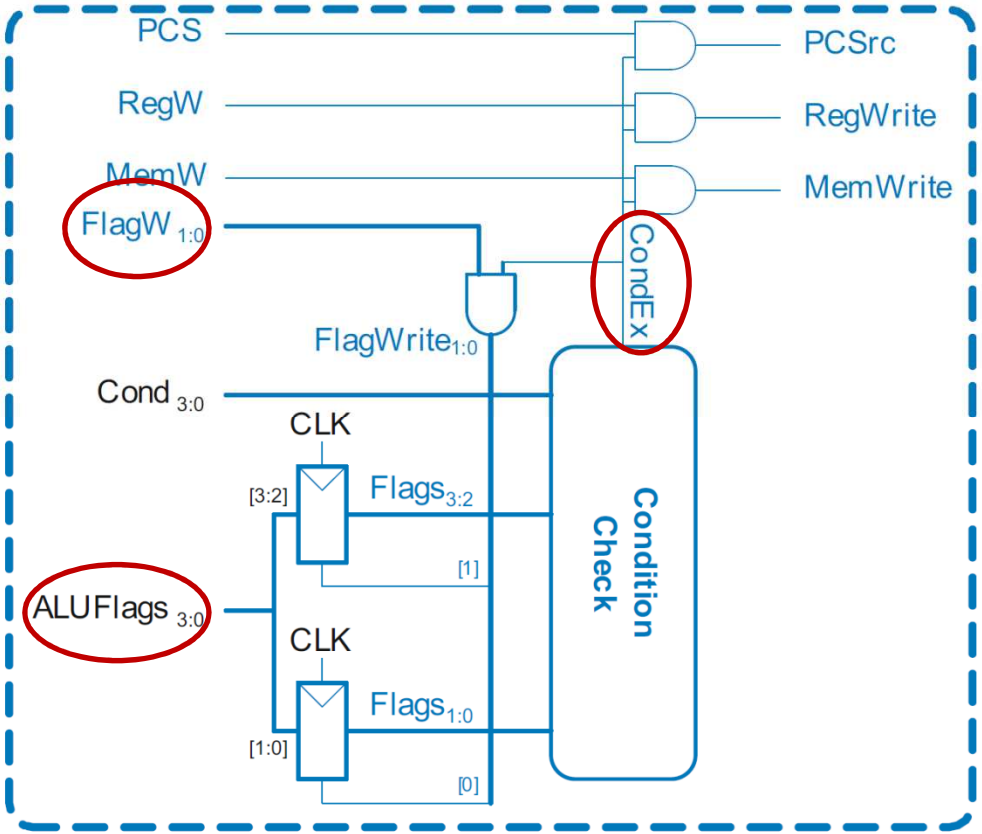
FlagW$_{1:0}$ = 11 AND **CondEx** = 1 (unconditional) => **FlagWrite$_{1:0}$** = 11

ELSEVIER

Conditional Logic: Update (Set) Flags

Flags$_{3:0}$ = NZCV

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



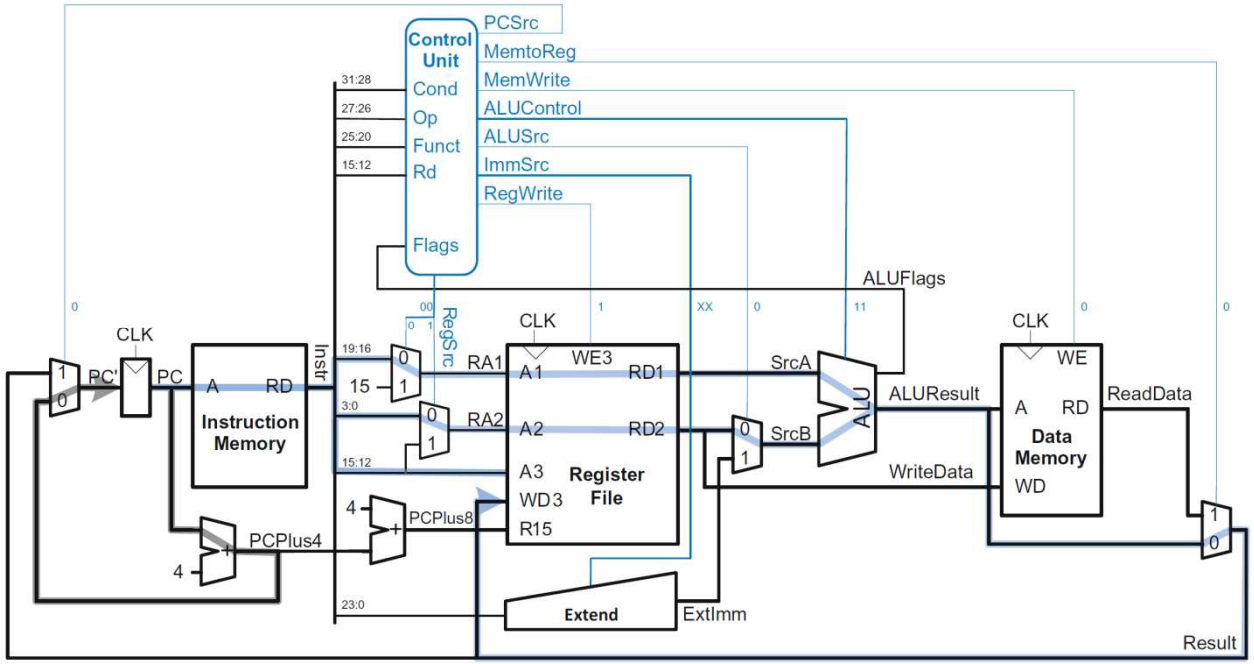
Example: `ANDS R7, R1, R3`

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

ELSEVIER

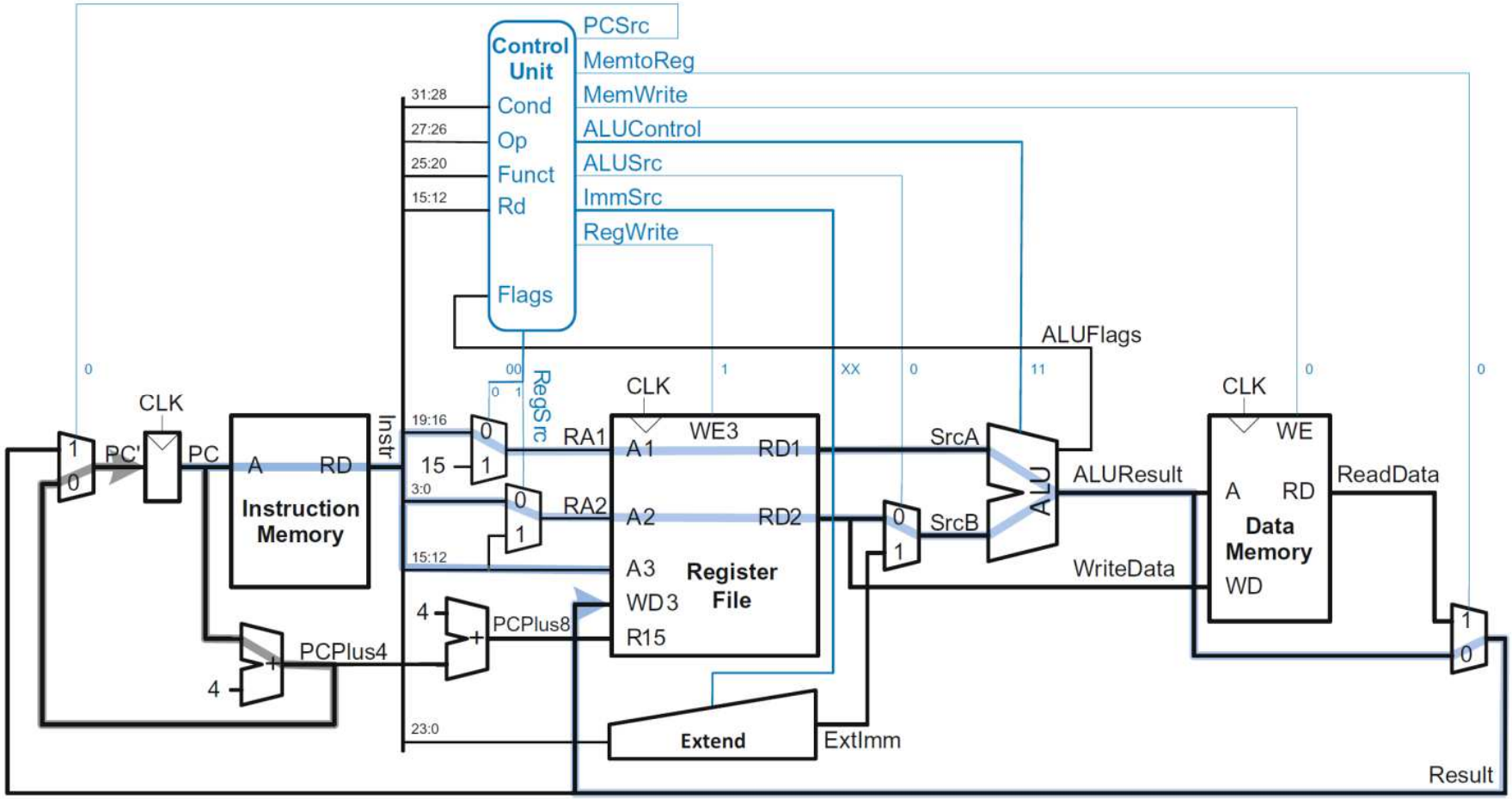
Example: ORR

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

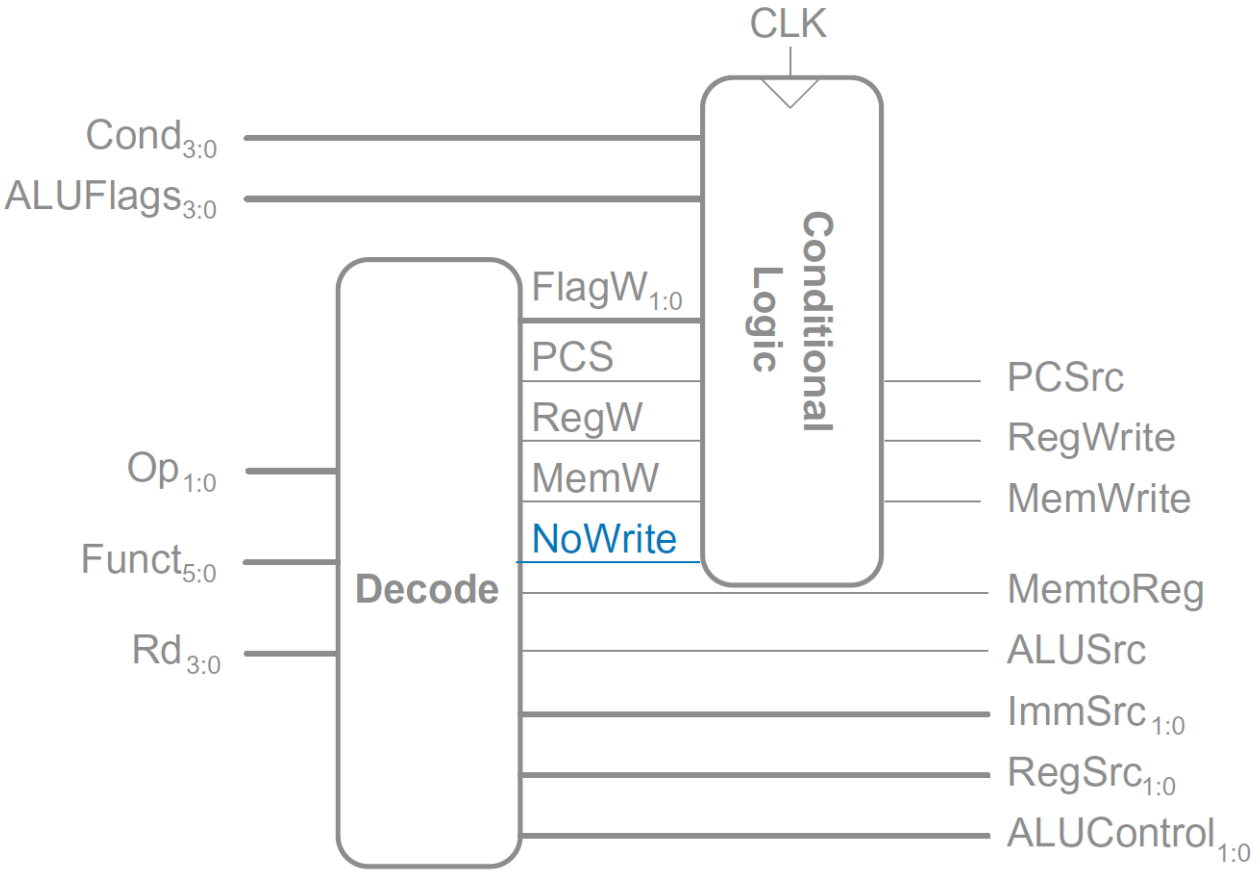


ELSEVIER

Example: ORR

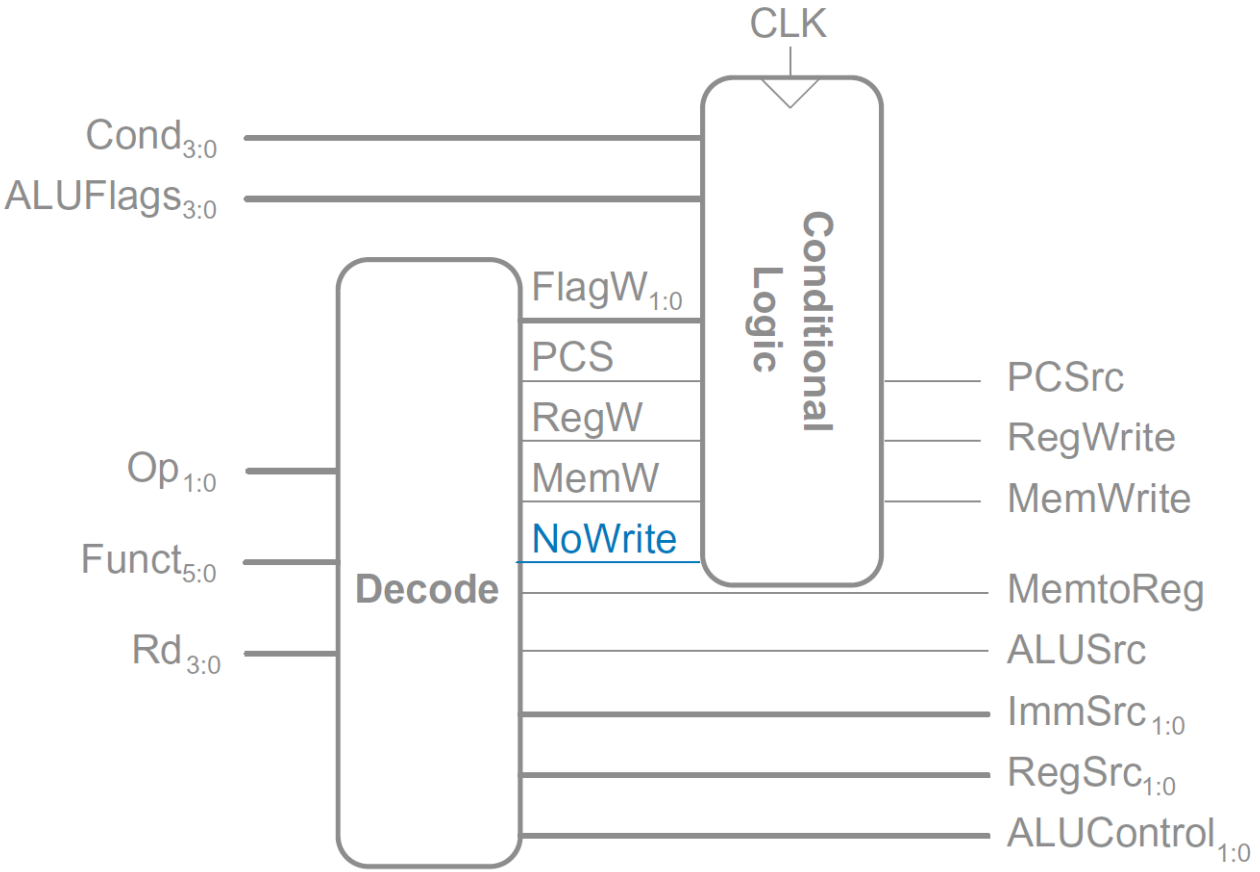


Extended Functionality: CMP



ELSEVIER

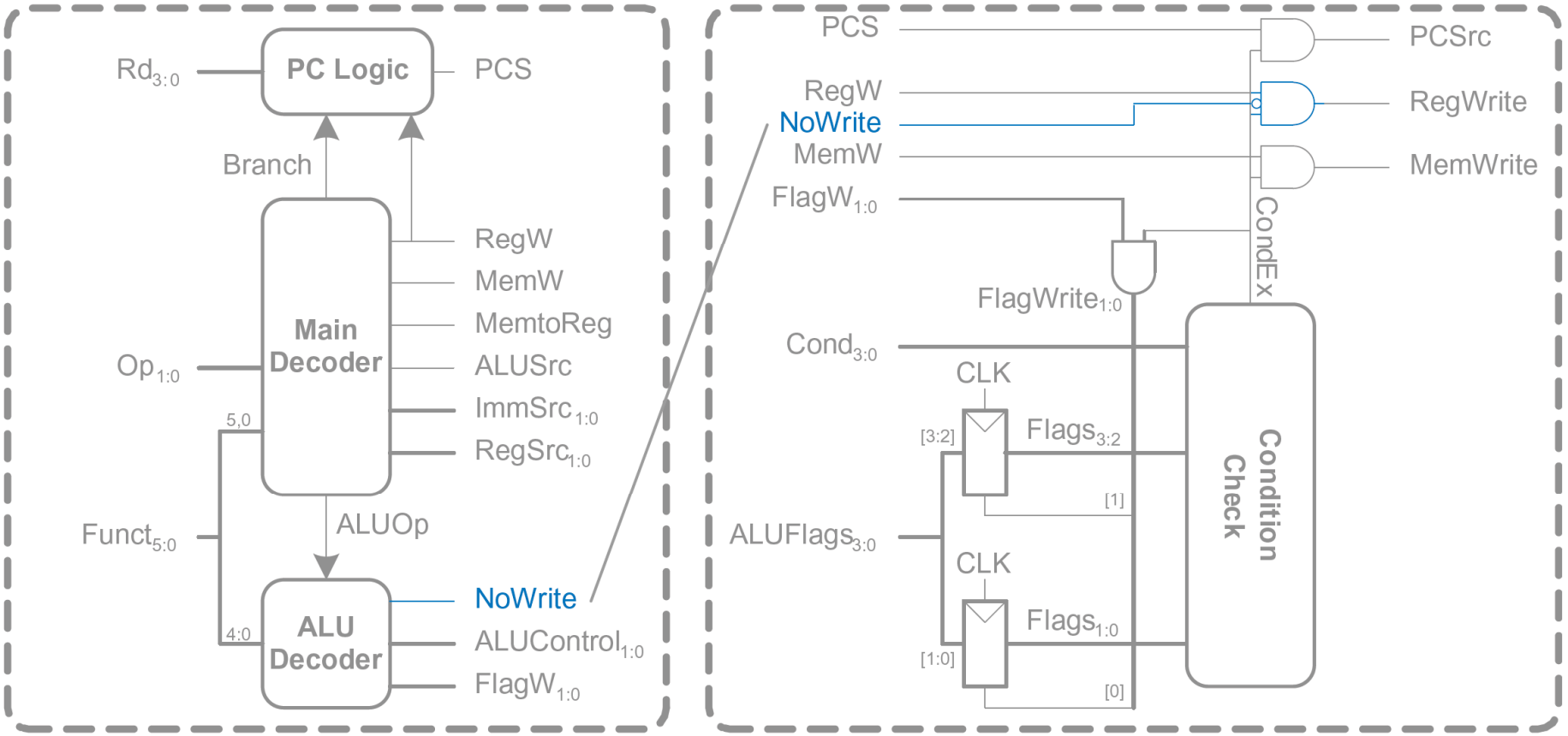
Extended Functionality: CMP



No change to datapath

ELSEVIER

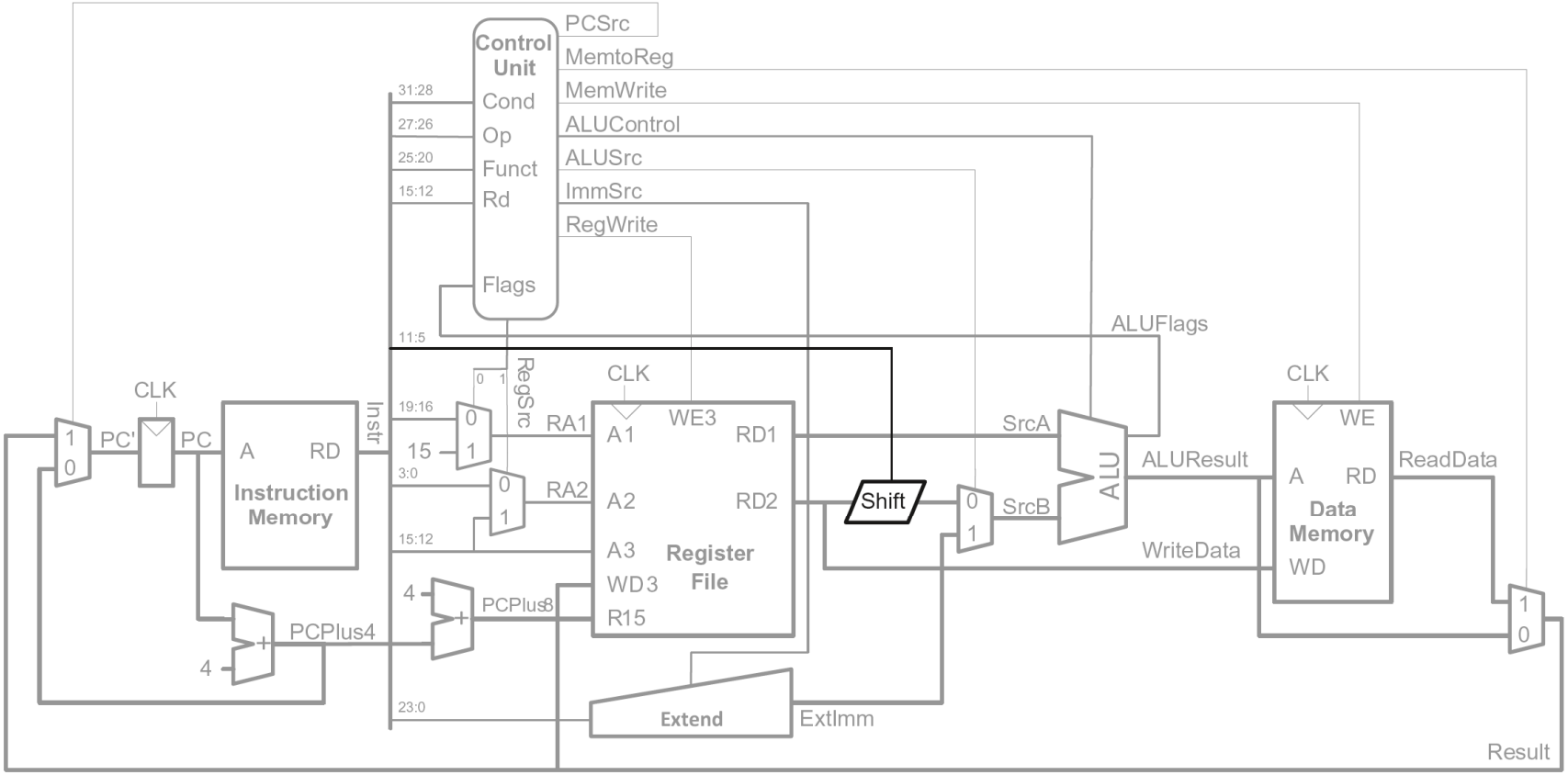
Extended Functionality: CMP



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

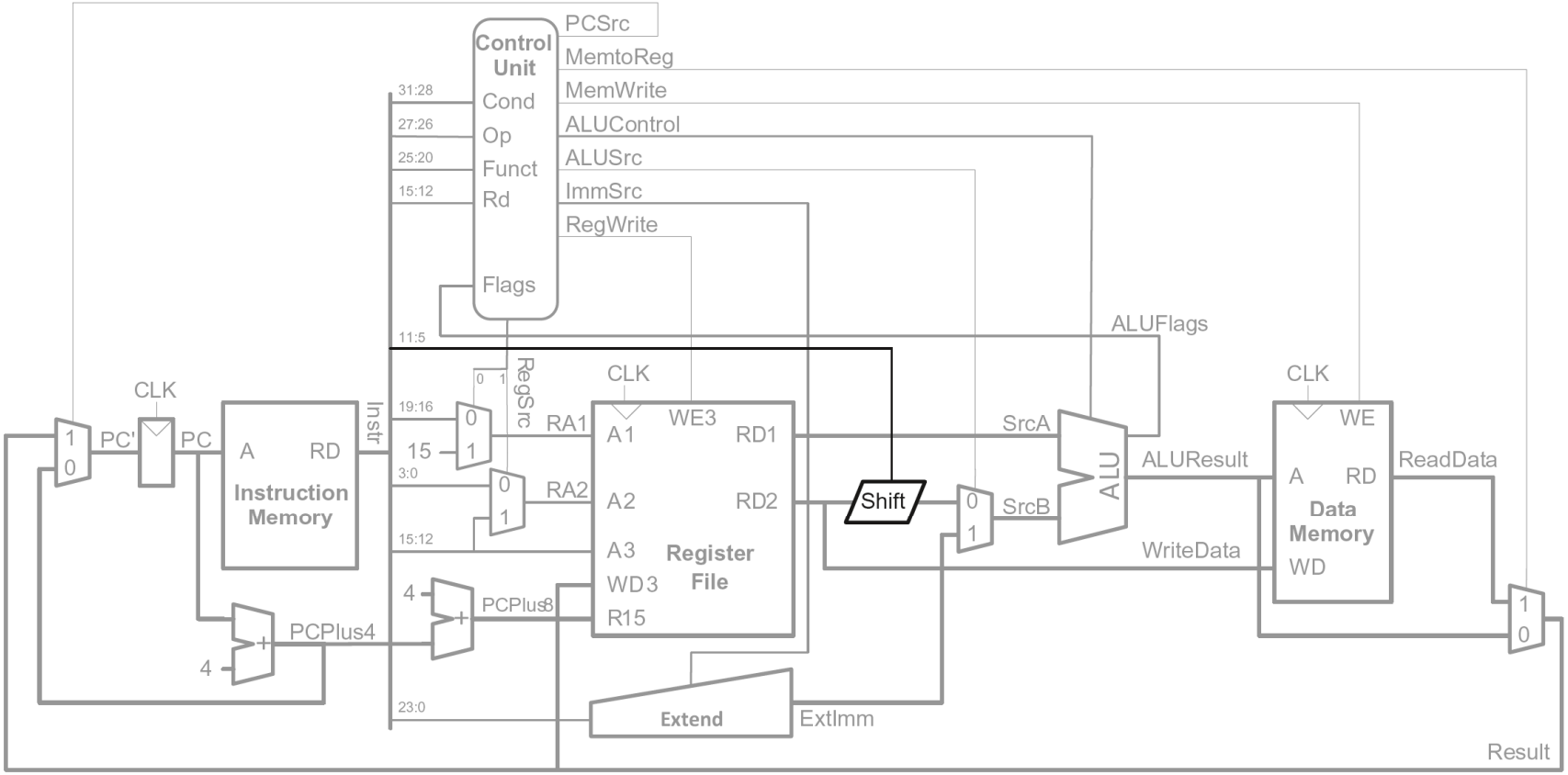


ADD R7, R2, R12, LSR #5

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

ELSEVIER

Extended Functionality: Shifted Register



No change to controller

ELSEVIER

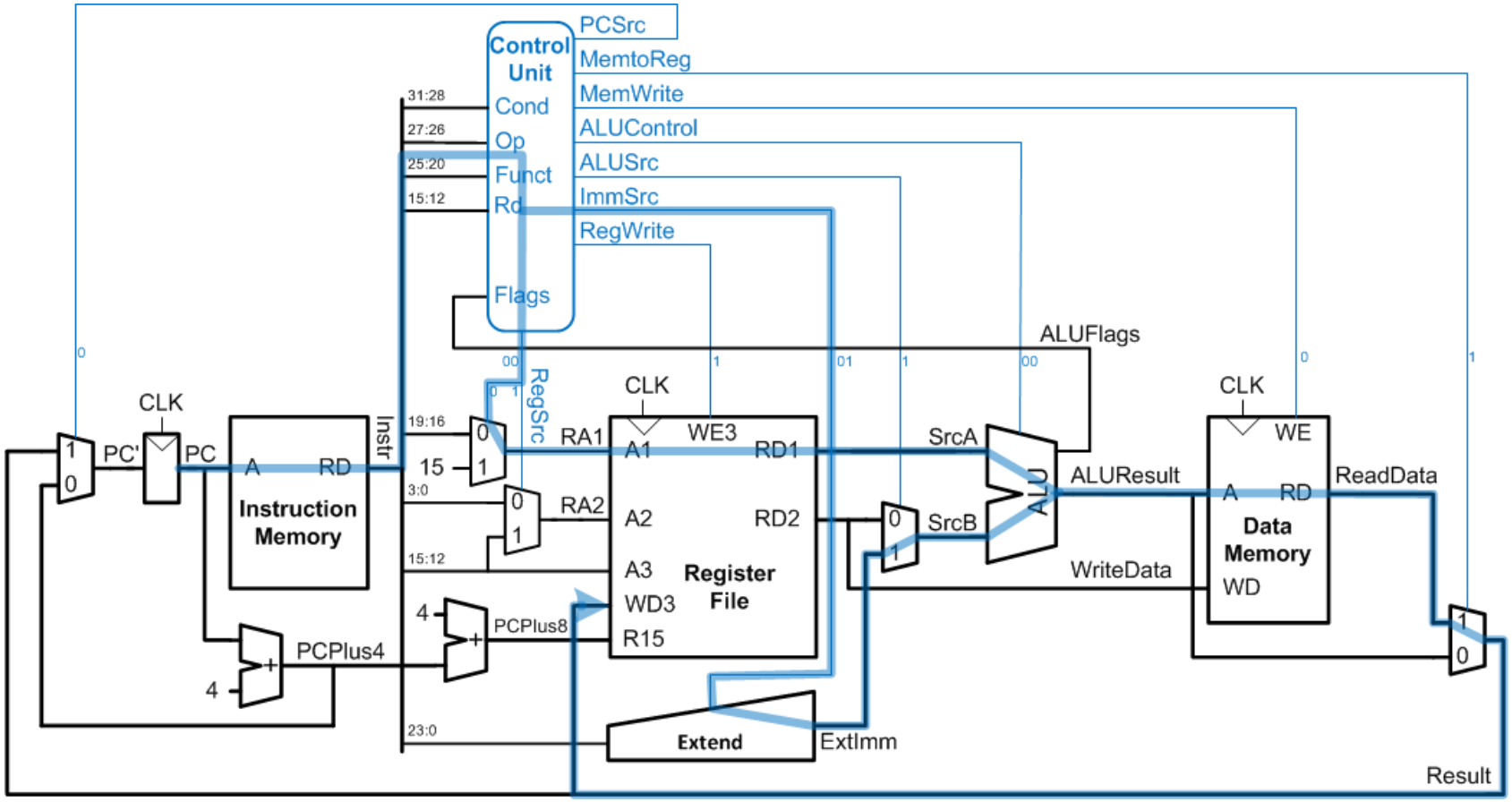
Review: Processor Performance

Program Execution Time

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

= # instructions x CPI x T_C

Single-Cycle Performance



T_c limited by critical path (LDR)

Single-Cycle Performance

- **Single-cycle critical path**:

$$T_{c1} = t_{pcq\_PC} + t_{\text{mem}} + t_{\text{dec}} + \max[t_{mux} + t_{RF\text{read}}, t_{sext} + t_{\text{mux}}] + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$$

- **Typically, limiting paths are:**
  - memory, ALU, register file
  - $T_{c1} = t_{pcq\_PC} + 2t_{\text{mem}} + t_{dec} + t_{RF\text{read}} + t_{\text{ALU}} + 2t_{\text{mux}} + t_{RF\text{setup}}$

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

$T_{c1} = ?$

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

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

= **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

Multicycle ARM Processor

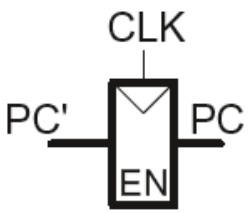
- **Single-cycle:**
  + simple
  - cycle time limited by longest instruction (LDR)
  - separate memories for instruction and data
  - 3 adders/ALUs
- **Multicycle:**
  + higher clock speed
  + simpler instructions run faster
  + reuse expensive hardware on multiple cycles
  - sequencing overhead paid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

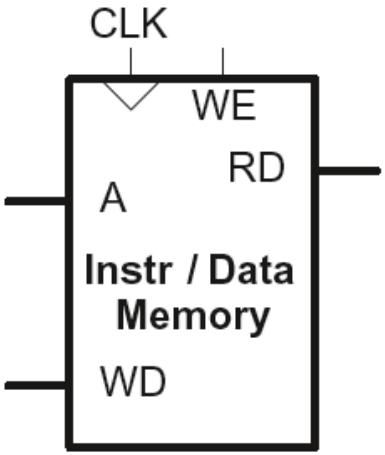
Same design steps as single-cycle:

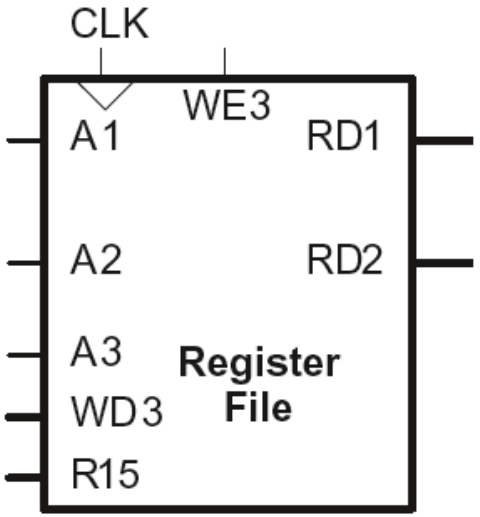
- **first datapath**
- **then control**

Multicycle State Elements

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

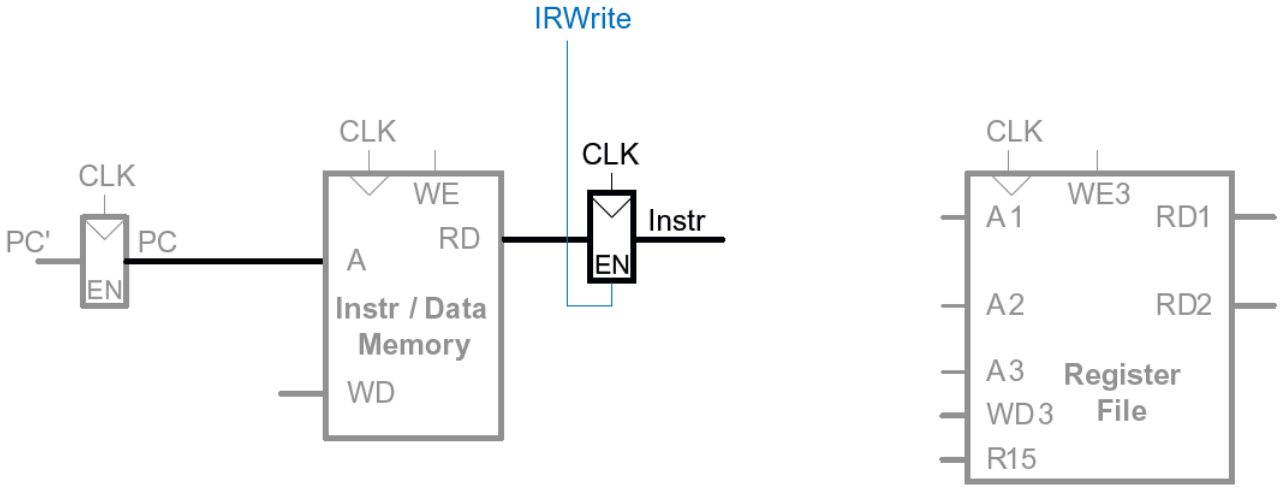


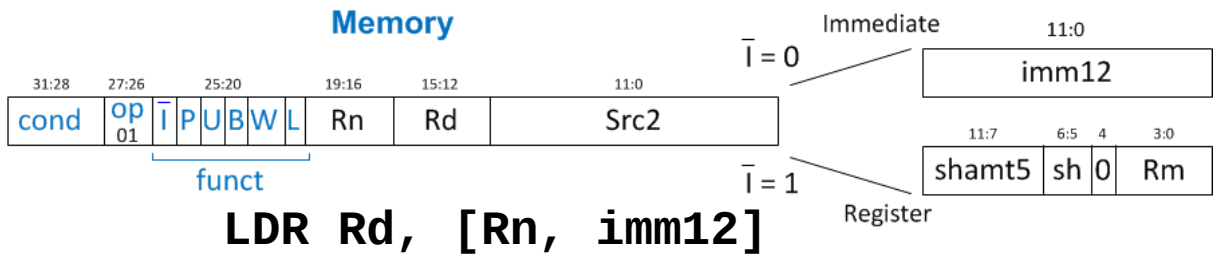




Multicycle Datapath: Instruction Fetch

STEP 1: Fetch instruction



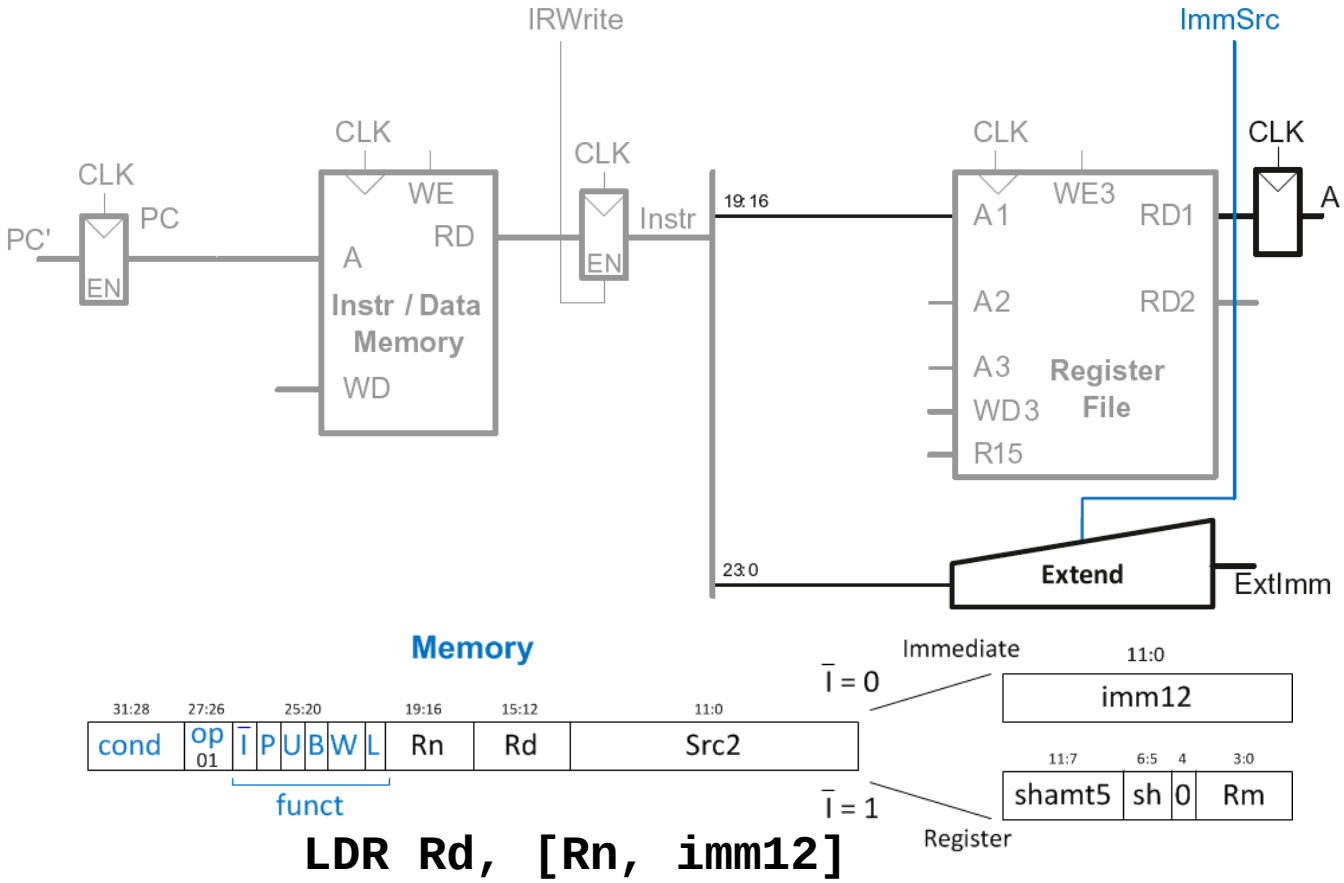


`LDR Rd, [Rn, imm12]`

ELSEVIER

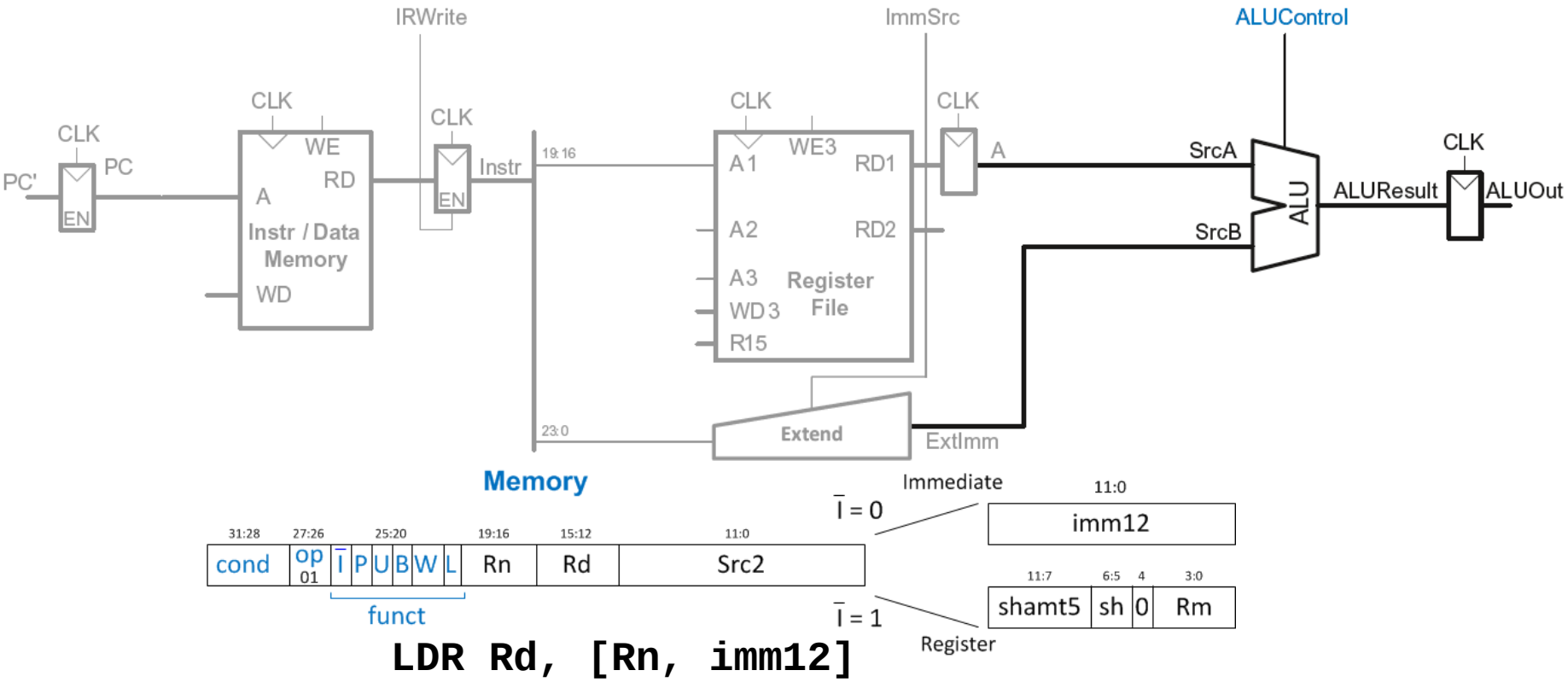
Multicycle Datapath: LDR Register Read

STEP 2: Read source operands from RF



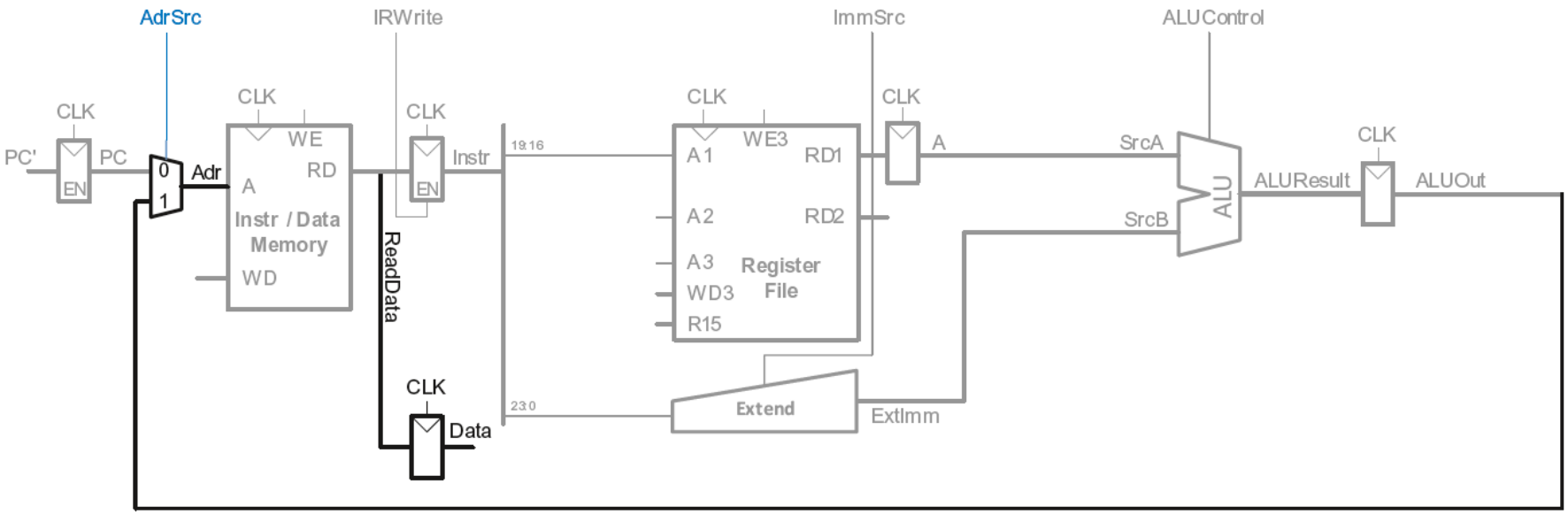
Multicycle Datapath: LDR Address

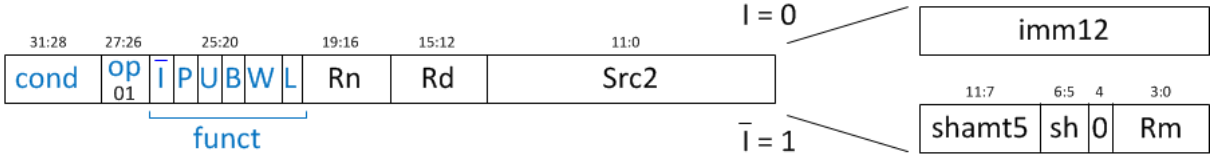
STEP 3: Compute the memory address



Multicycle Datapath: LDR Memory Read

STEP 4: Read data from memory

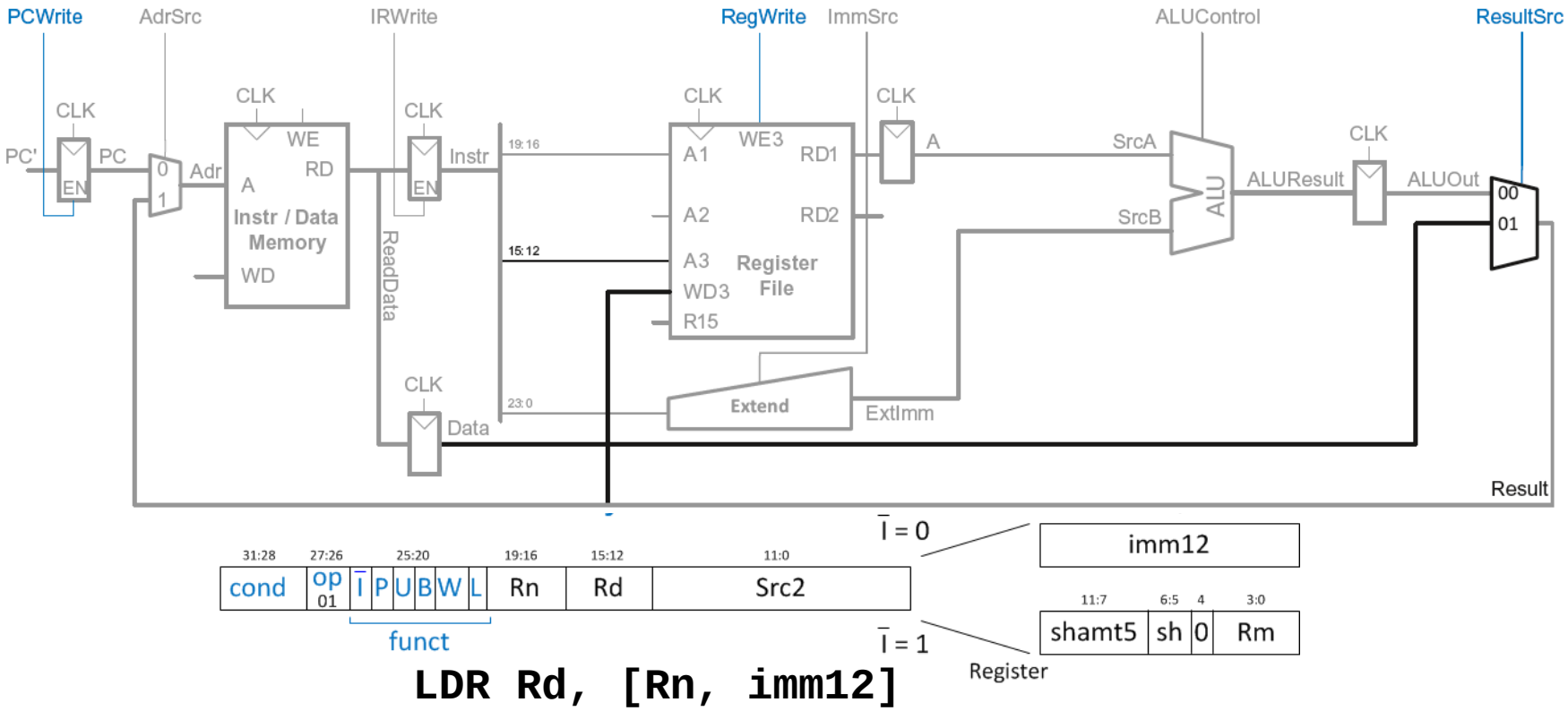




LDR Rd, [Rn, imm12]

Multicycle Datapath: LDR Write Register

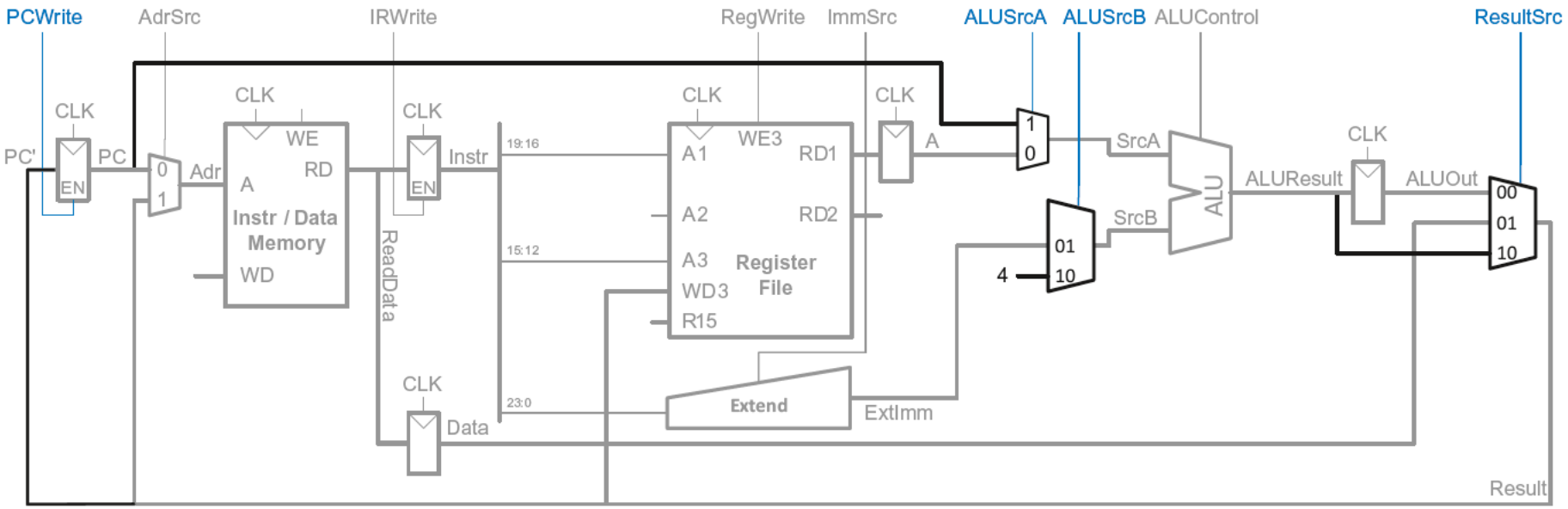
STEP 5: Write data back to register file



ELSEVIER

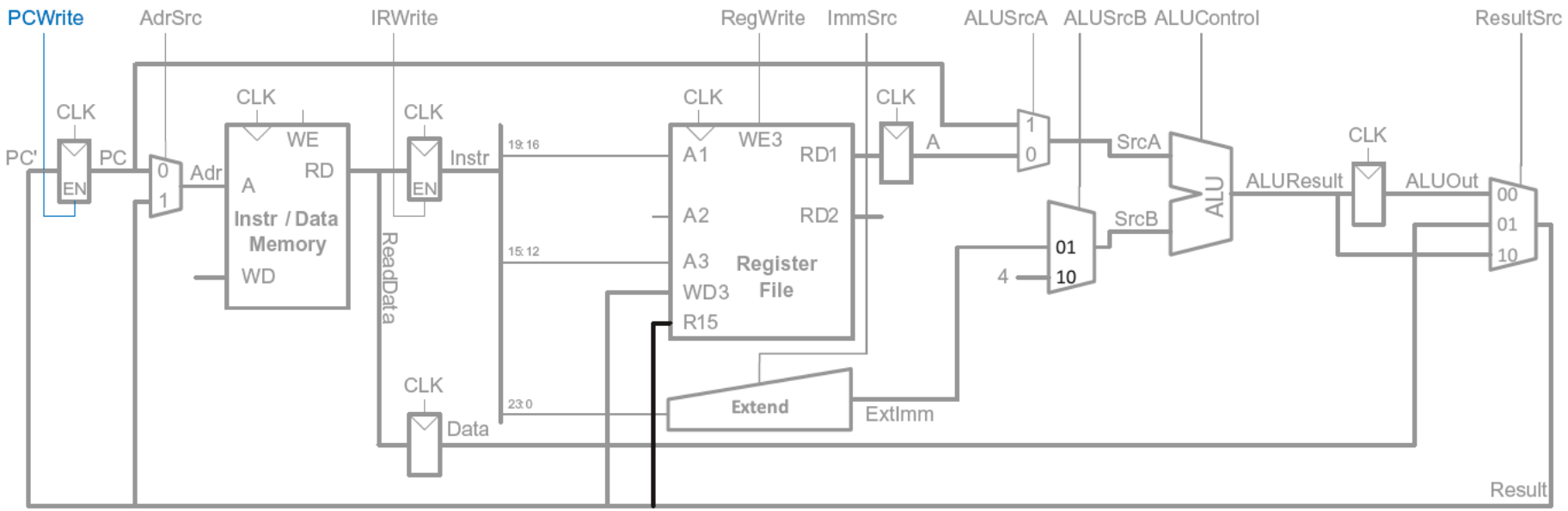
Multicycle Datapath: Increment PC

STEP 6: Increment PC



Multicycle Datapath: Access to PC

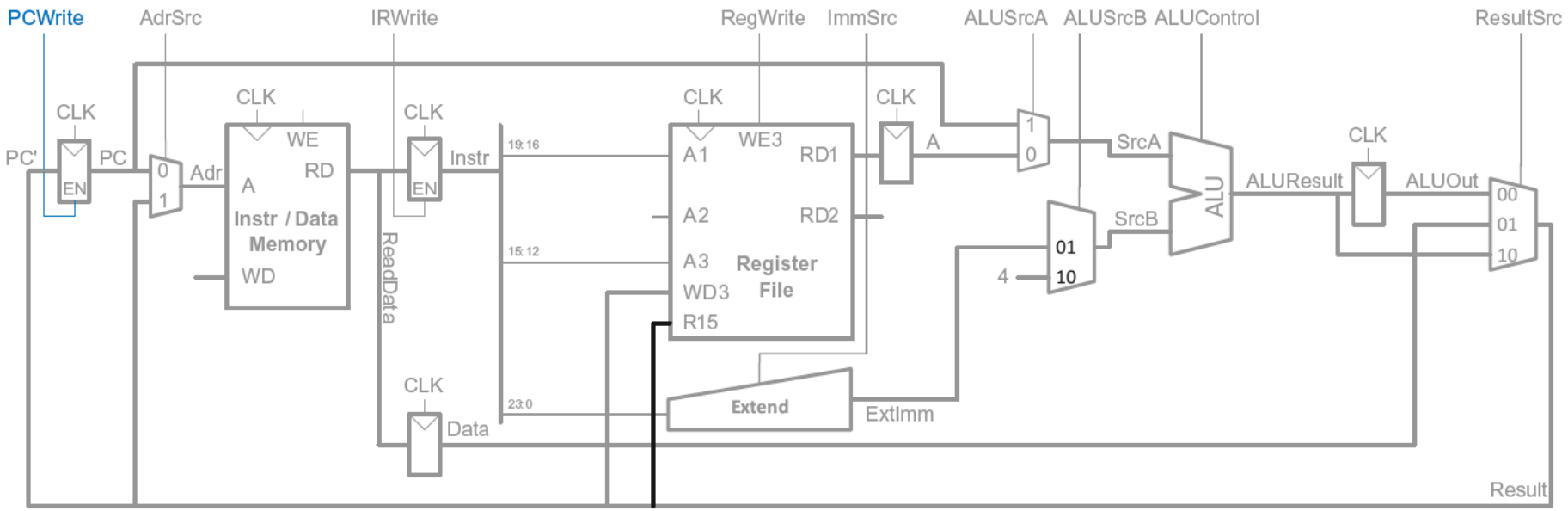
PC can be read/written by instruction



Multicycle Datapath: Access to PC

PC can be read/written by instruction

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



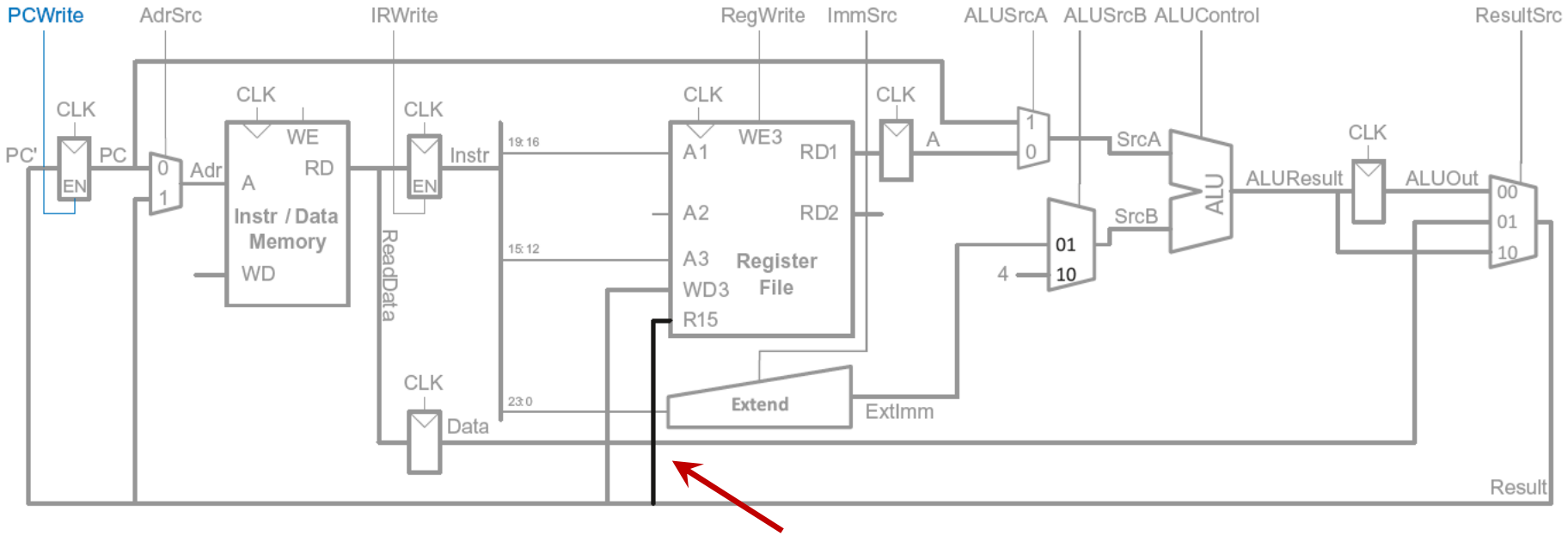
Multicycle Datapath: Read to PC (R15)

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)

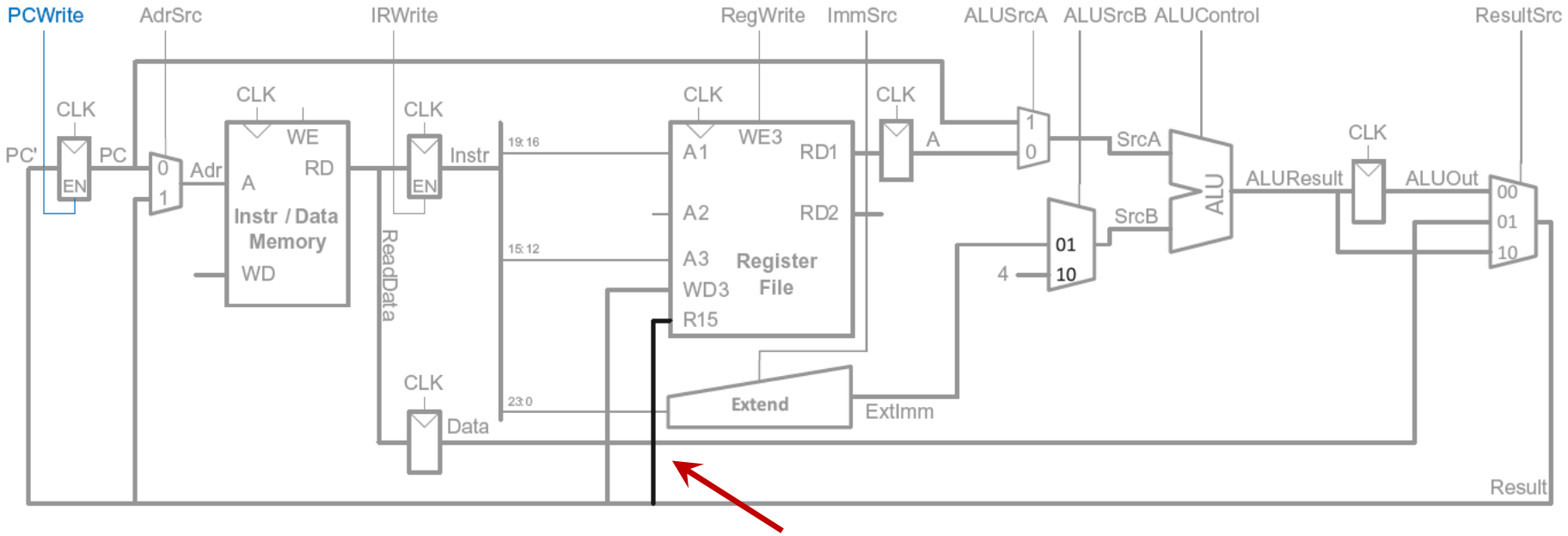
Example: `ADD R1, R15, R2`

- 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)

Multicycle Datapath: Read to PC (R15)

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

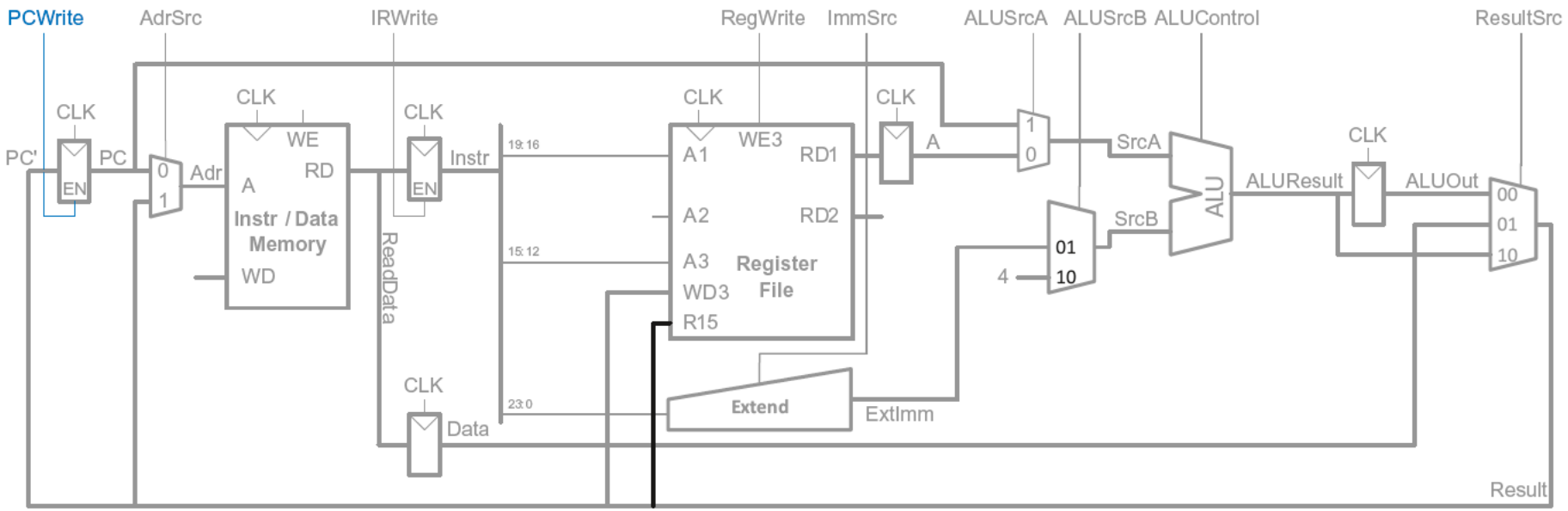
- R15 needs to be read as PC+8 from Register File (RF) in 2nd step
- So (also in 2nd step) PC + 8 is produced by ALU and routed to R15 input of RF



Multicycle Datapath: 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 Datapath: Write to PC (R15)

Example: SUB **R15**, R8, R3

Multicycle Datapath: Write to PC (R15)

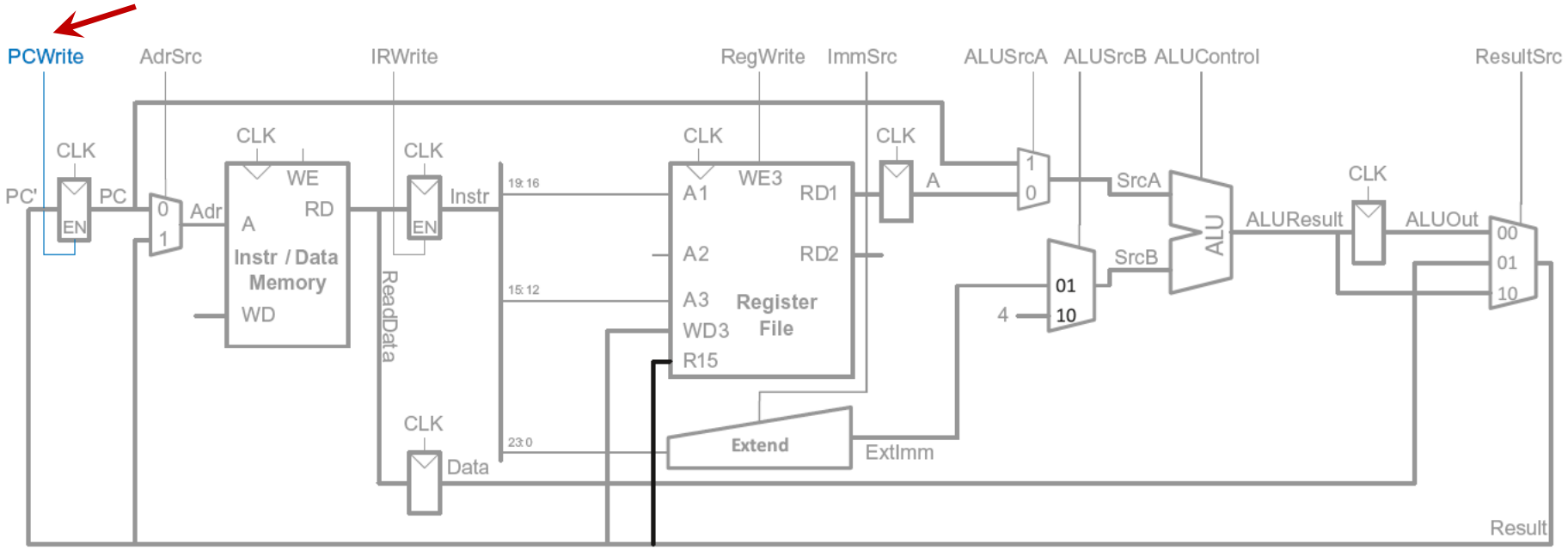
Example: `SUB **R15**, R8, R3`

- Result of instruction needs to be written to the PC register
- ALUResult already routed to the PC register, just assert PCWrite

Multicycle Datapath: Write to PC (R15)

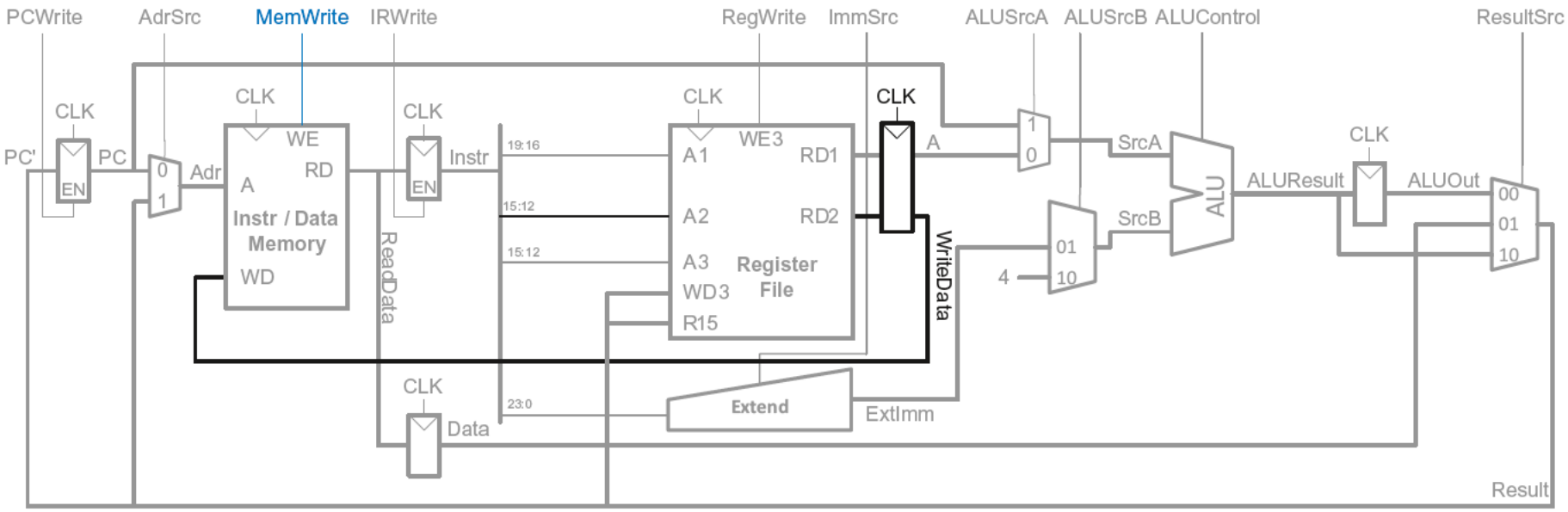
Example: SUB **R15**, R8, R3

- Result of instruction needs to be written to the PC register
- ALUResult already routed to the PC register, just assert PCWrite



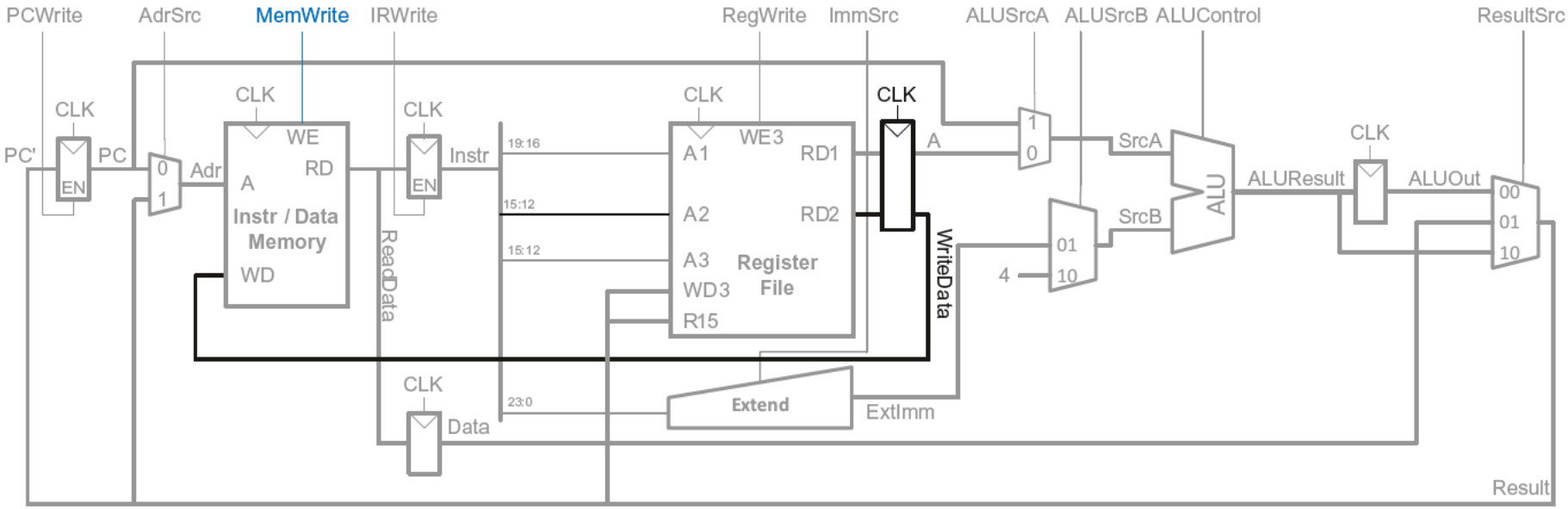
Multicycle Datapath: STR

Write data in Rn to memory



Multicycle Datapath: Data-processing

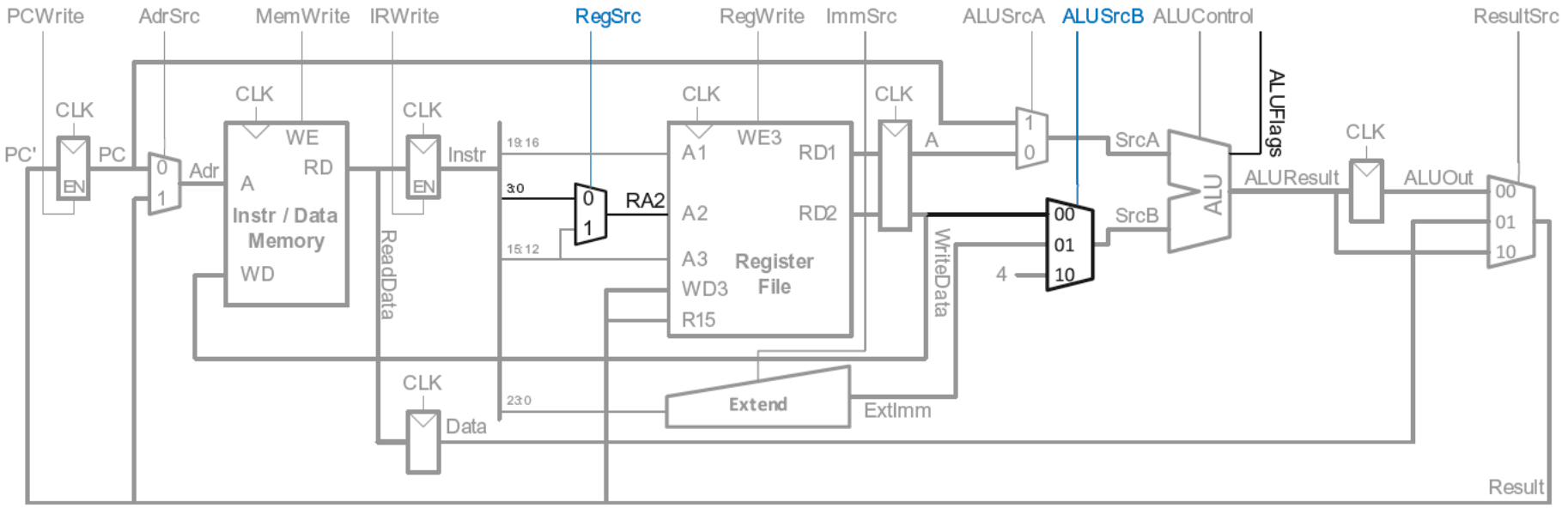
With immediate addressing (i.e., an immediate *Src2*), no additional changes needed for datapath



Multicycle Datapath: Data-processing

With register addressing (register *Src2*):

Read from Rn and Rm

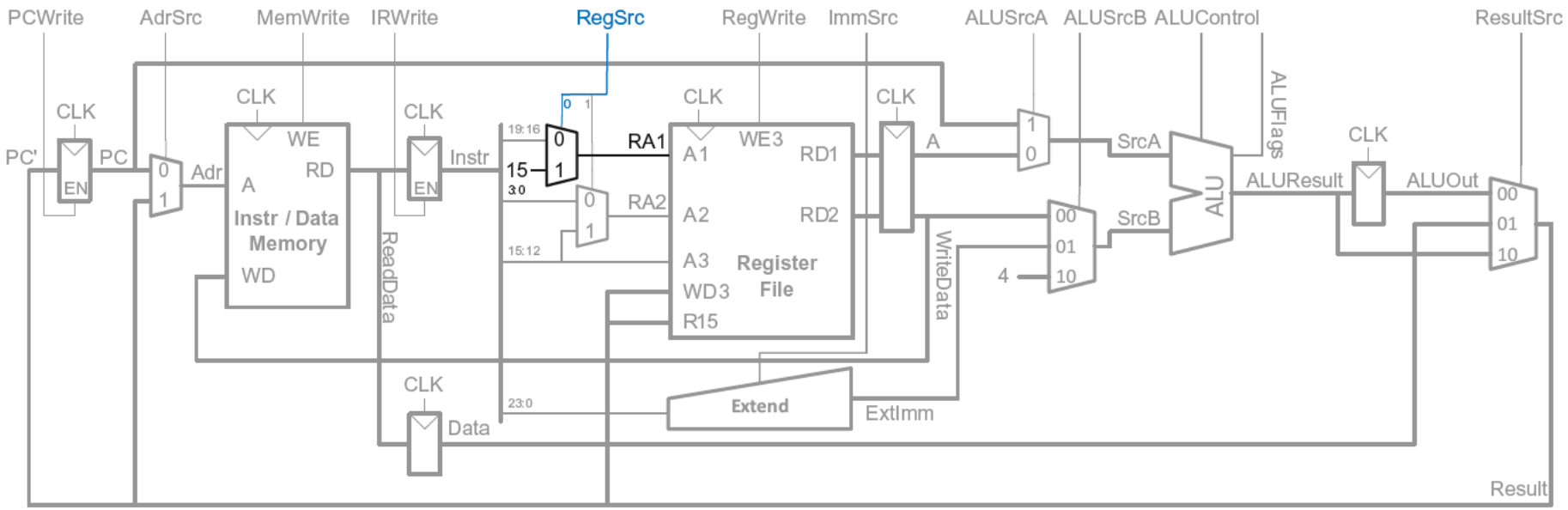


Multicycle Datapath: B

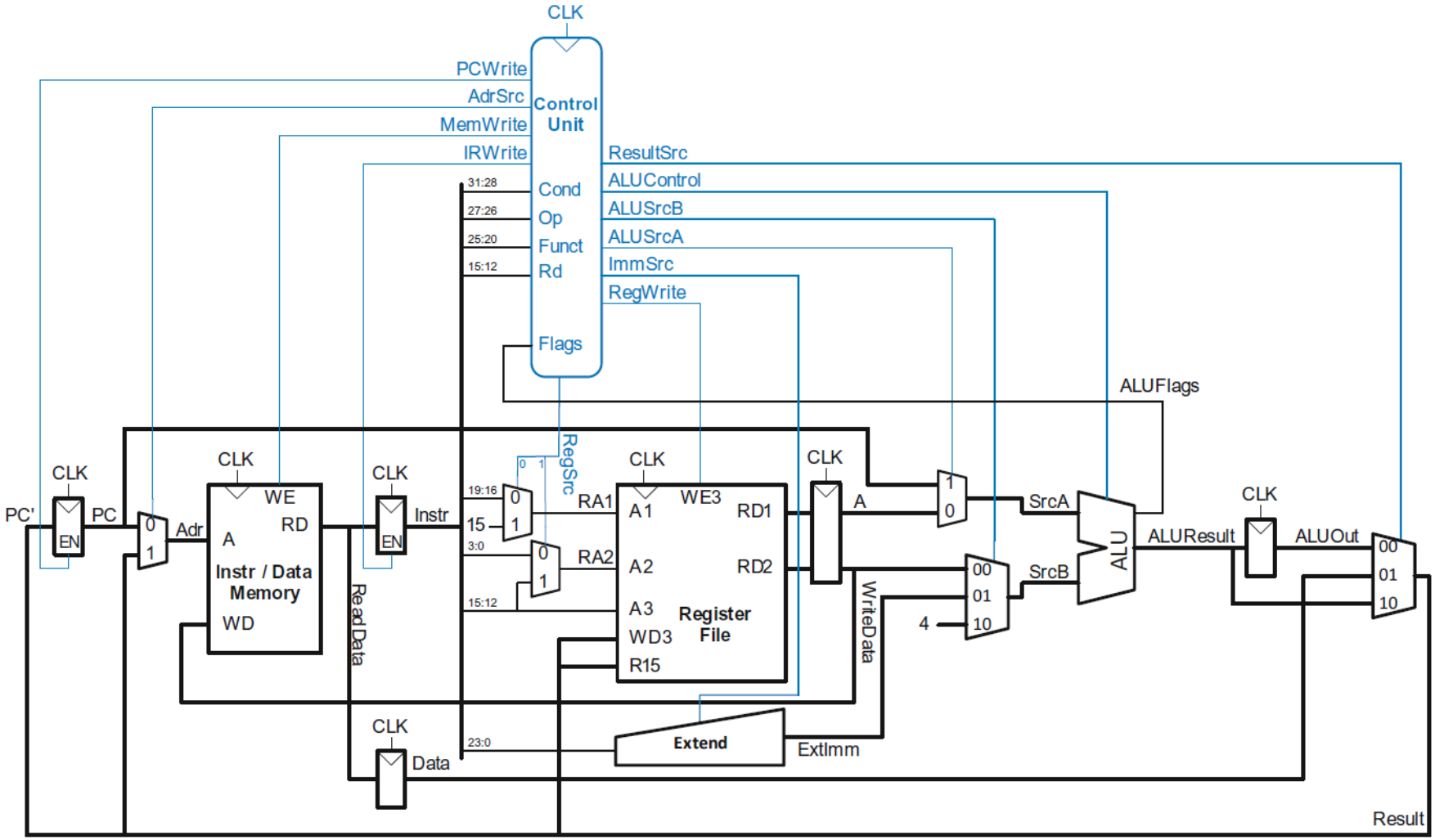
Calculate branch target address:

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

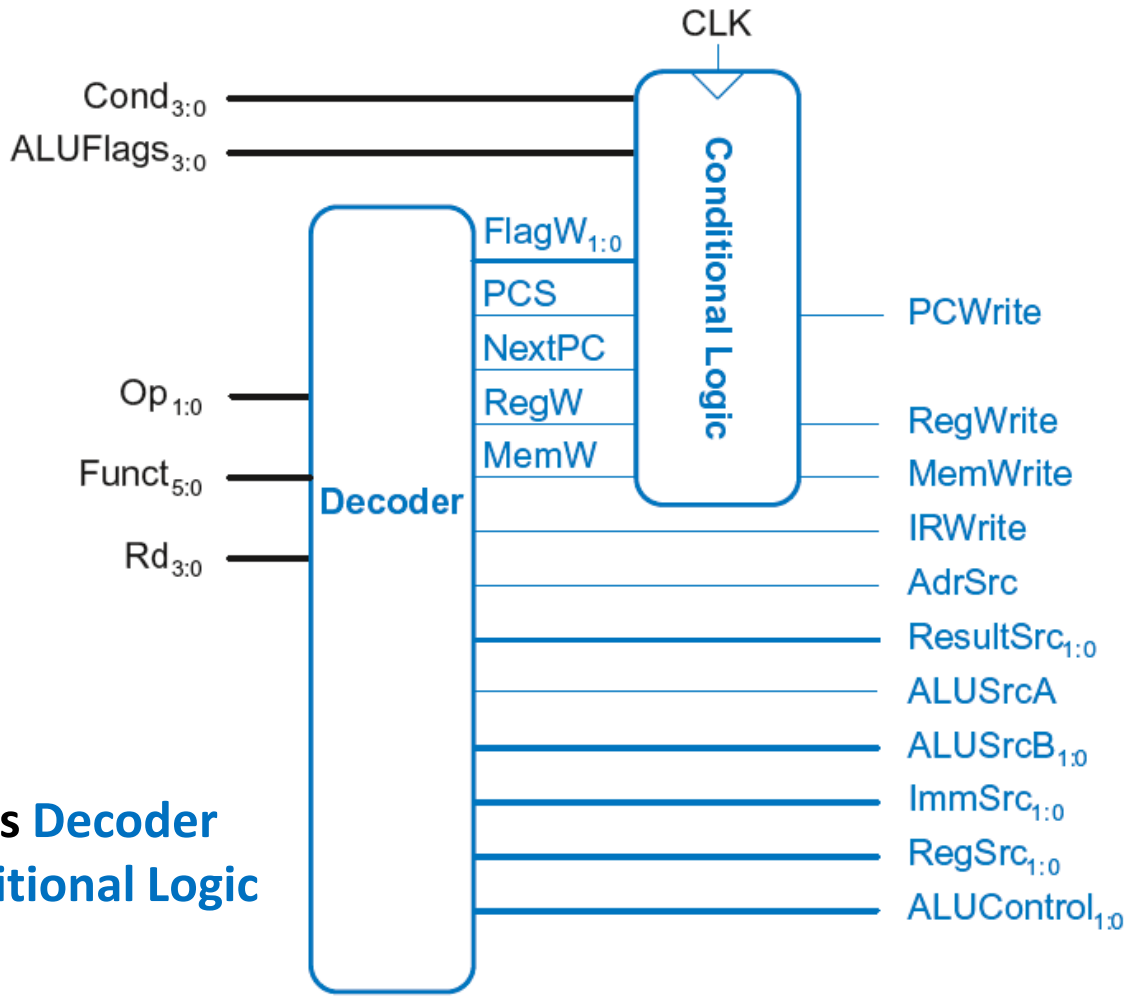
ExtImm = Imm24 << 2 and sign-extended



Multicycle ARM Processor

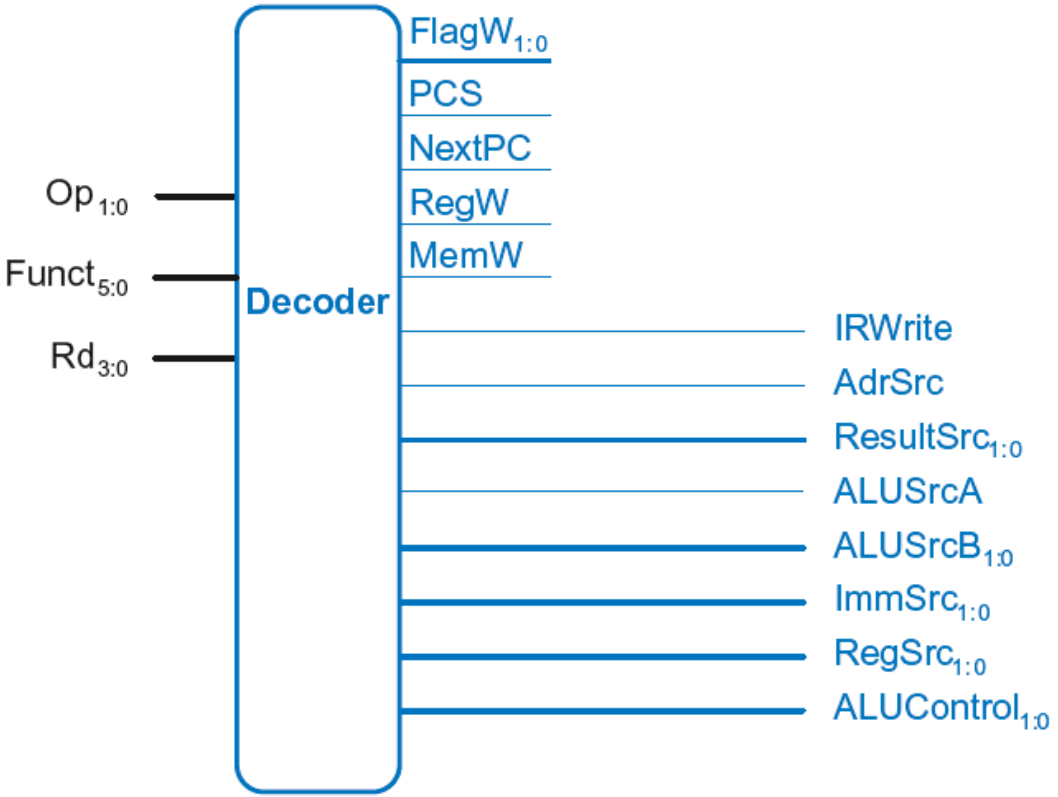


Multicycle Control

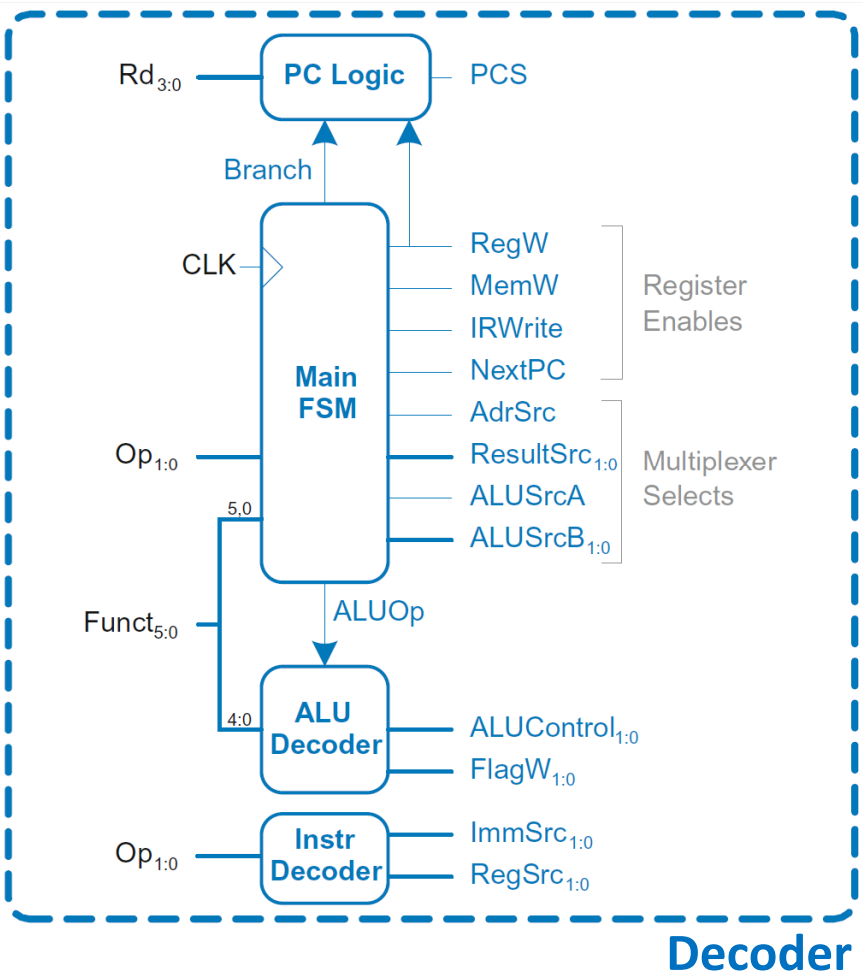


- **First, discuss Decoder**
- **Then, Conditional Logic**

Multicycle Control: Decoder

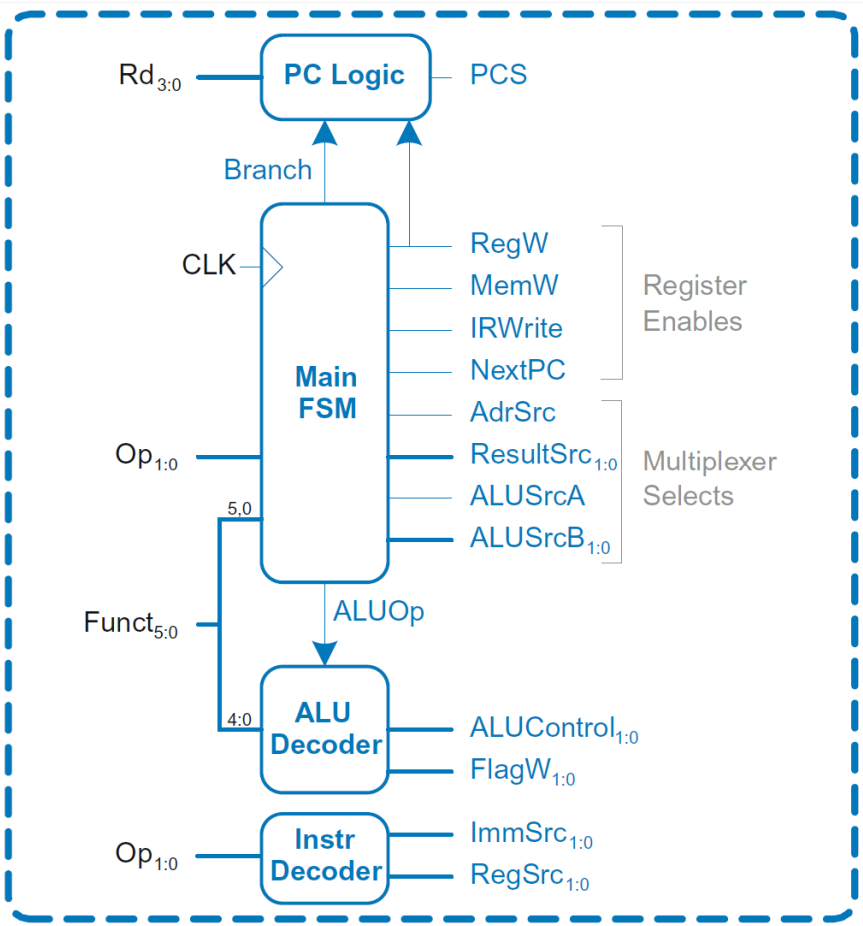


Multicycle Control: Decoder



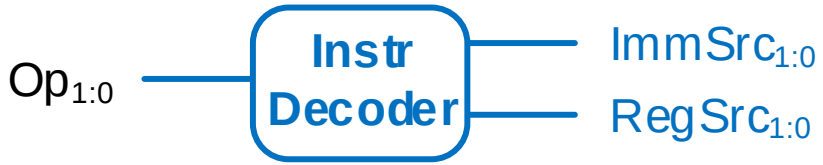
ELSEVIER

Multicycle Control: Decoder



ALU Decoder and PC Logic same as single-cycle

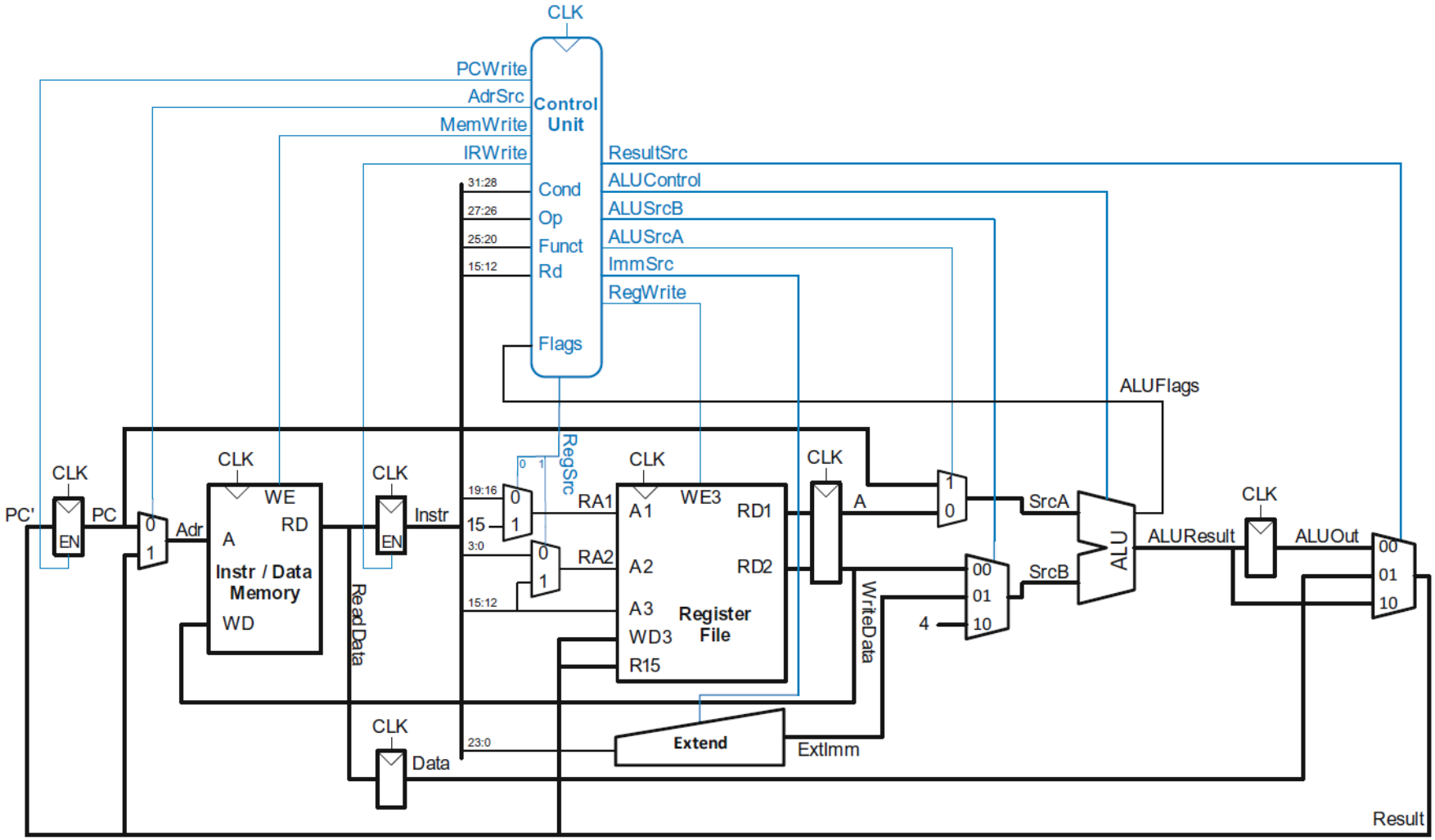
Multicycle Control: Instr Decoder



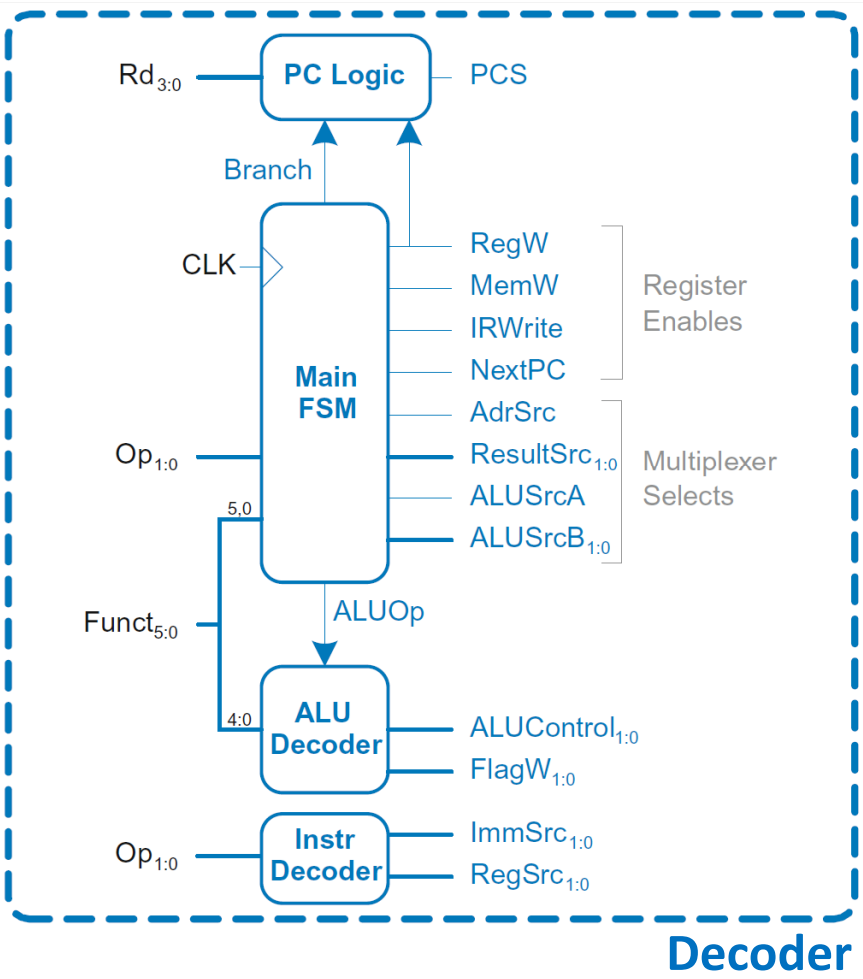
$$RegSrc_0 = (Op == 10_2)$$
$$RegSrc_1 = (Op == 01_2)$$
$$ImmSrc_{1:0} = Op$$

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

Multicycle ARM Processor

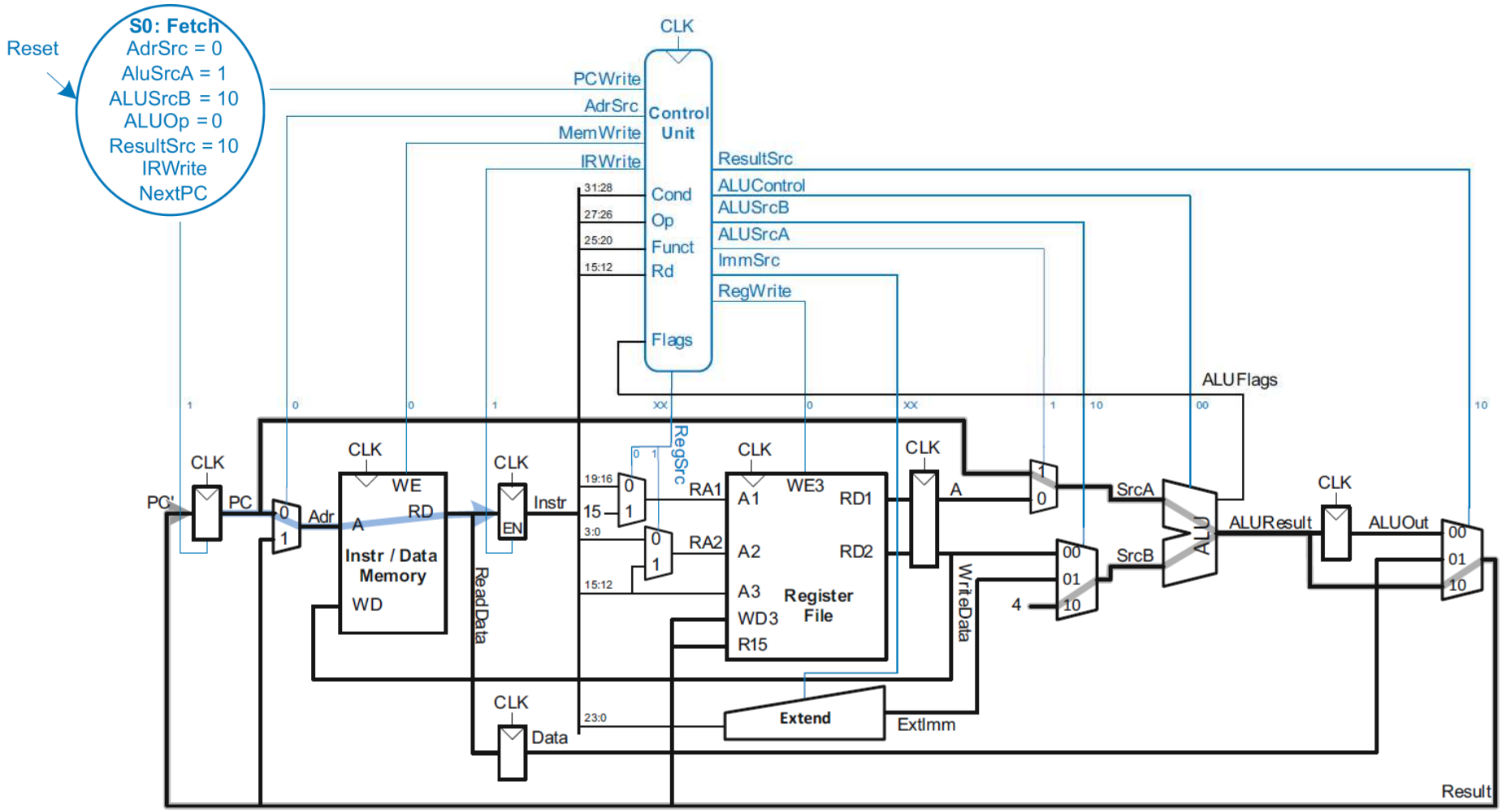


Multicycle Control: Main FSM

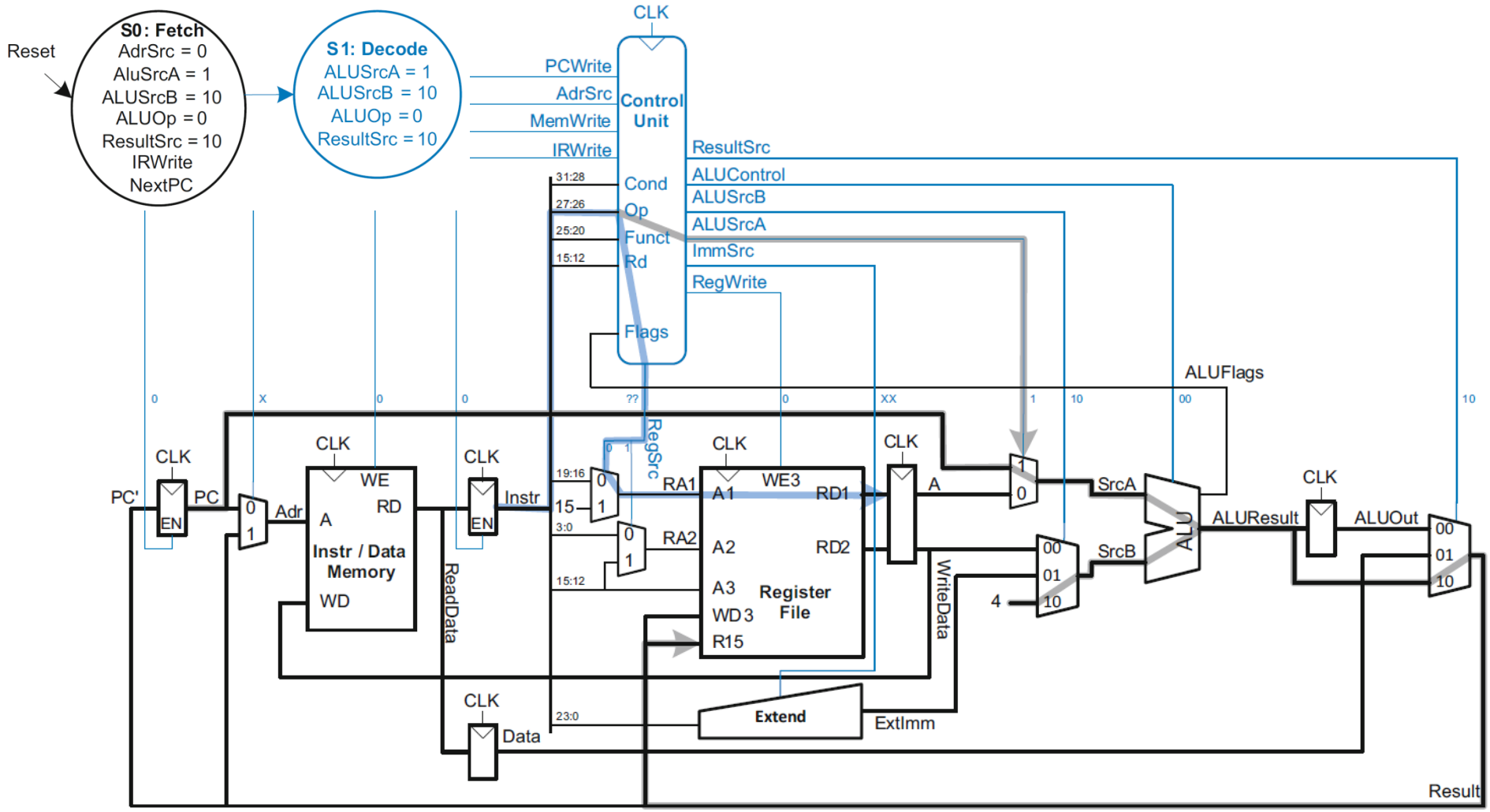


ELSEVIER

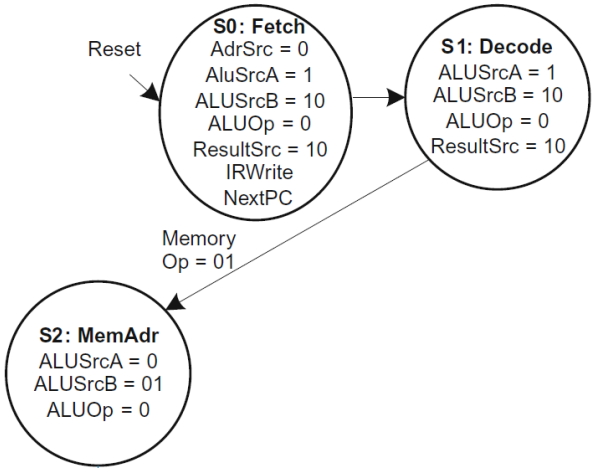
Main Controller FSM: Fetch

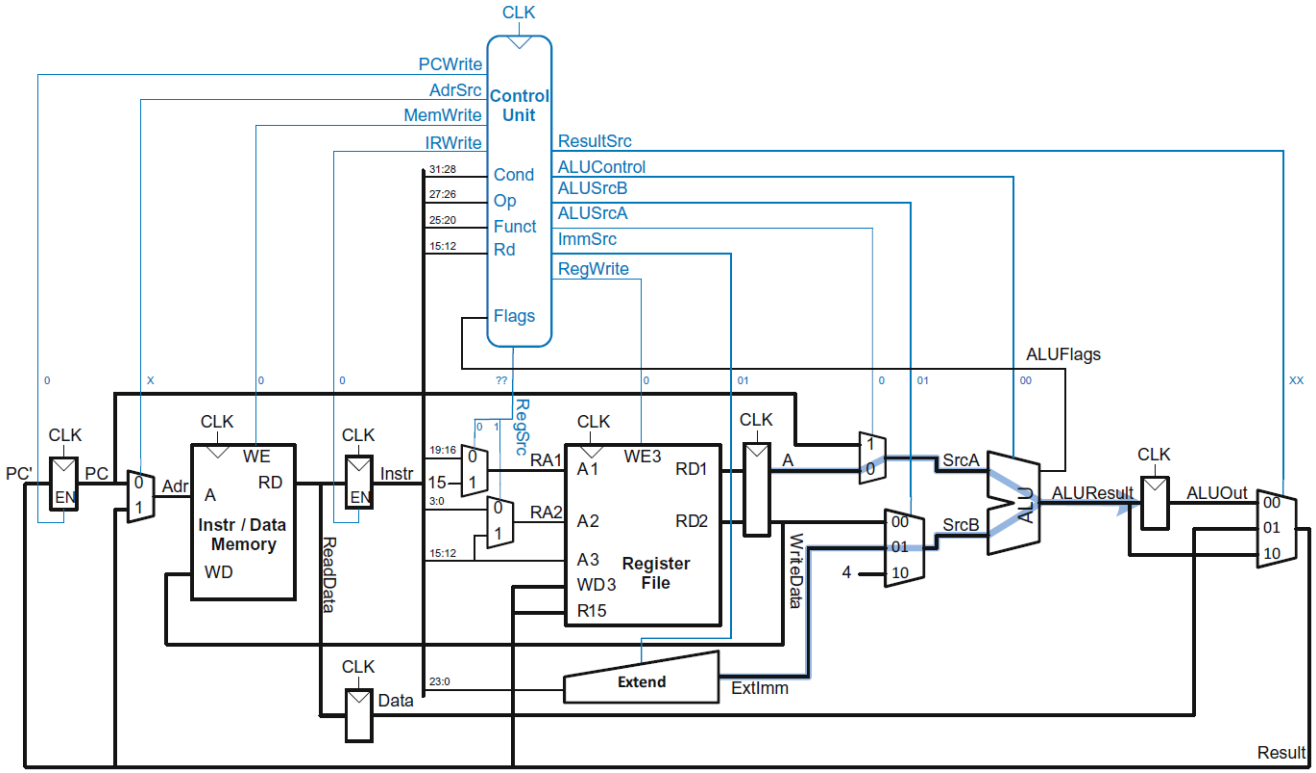


Main Controller FSM: Decode

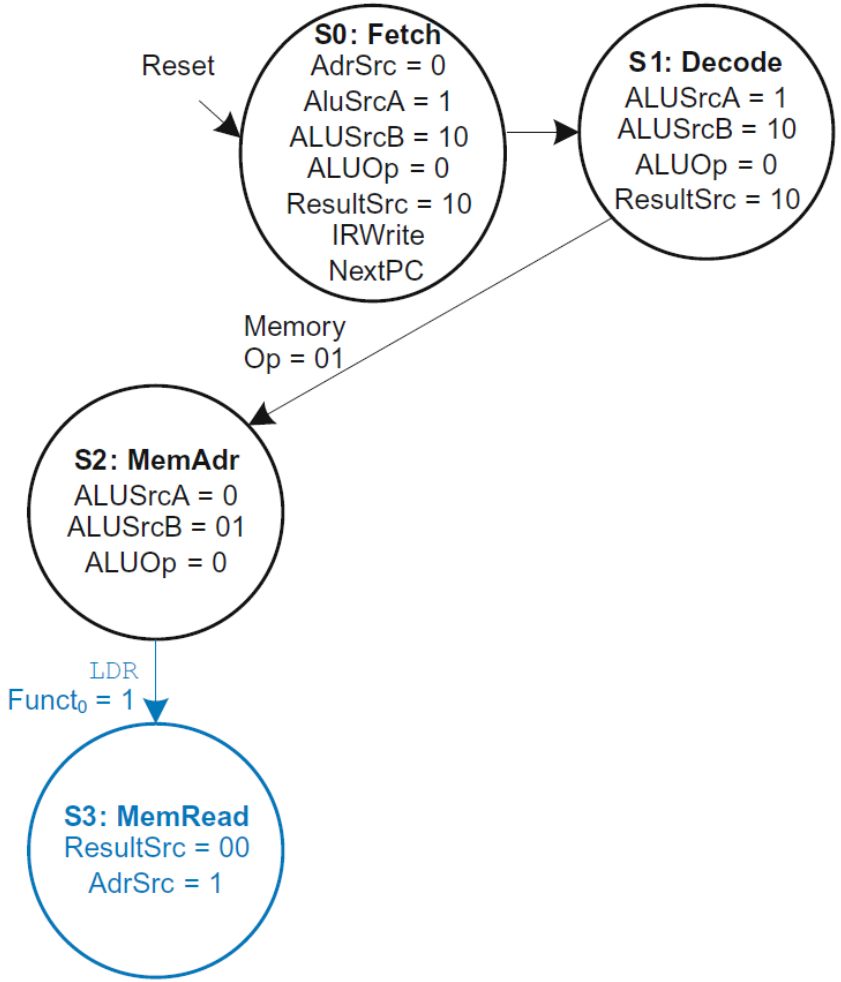


Main Controller FSM: Address

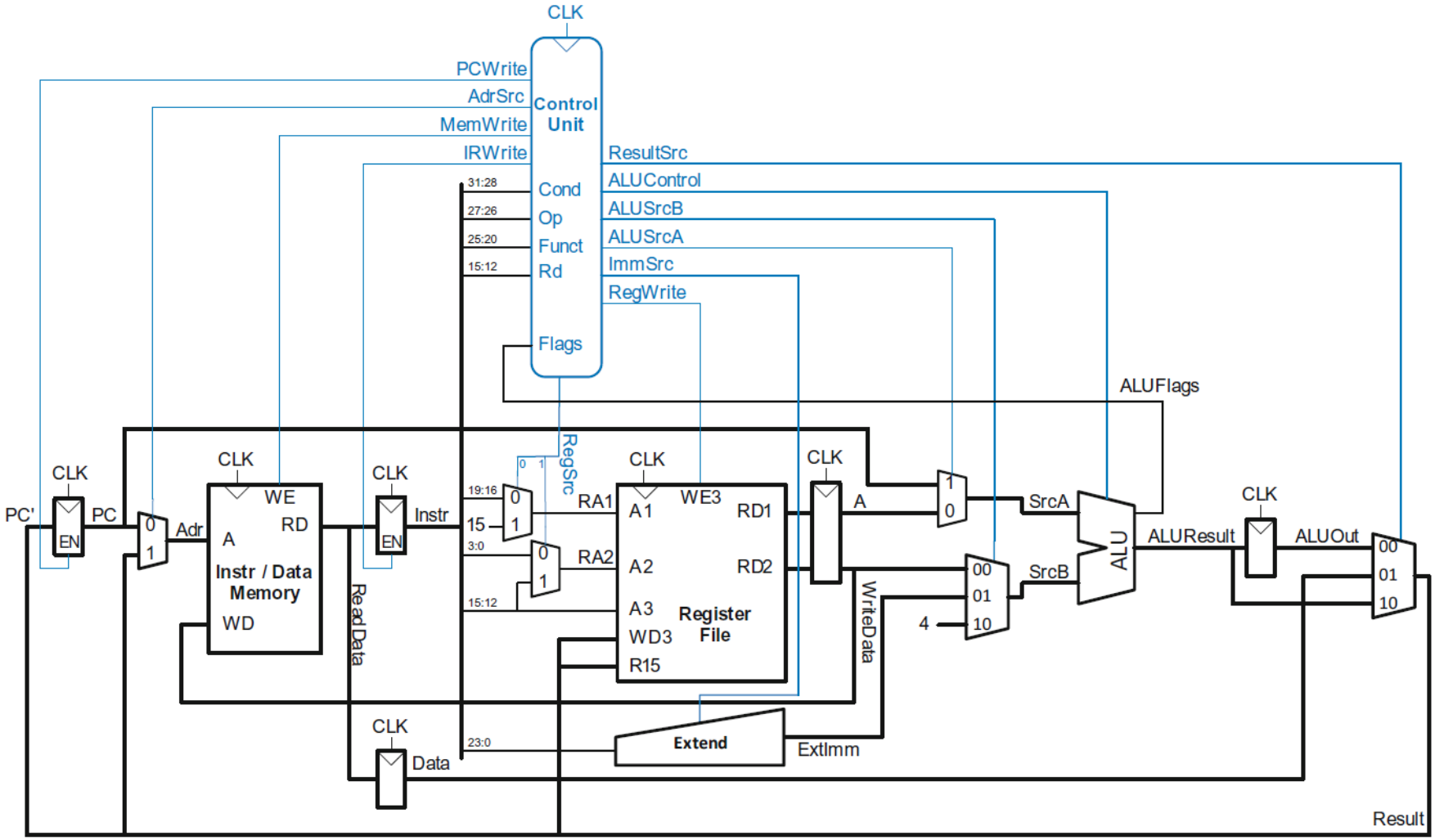




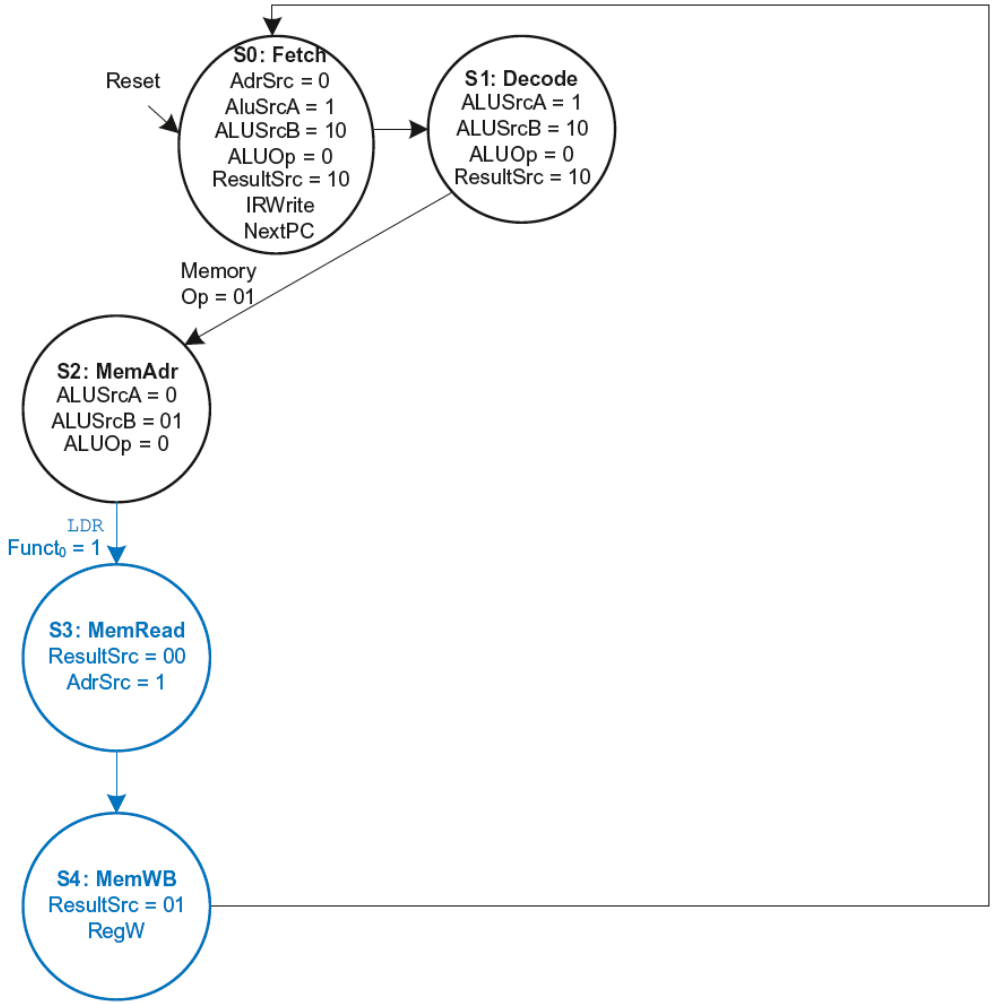
Main Controller FSM: Read Memory



Multicycle ARM Processor

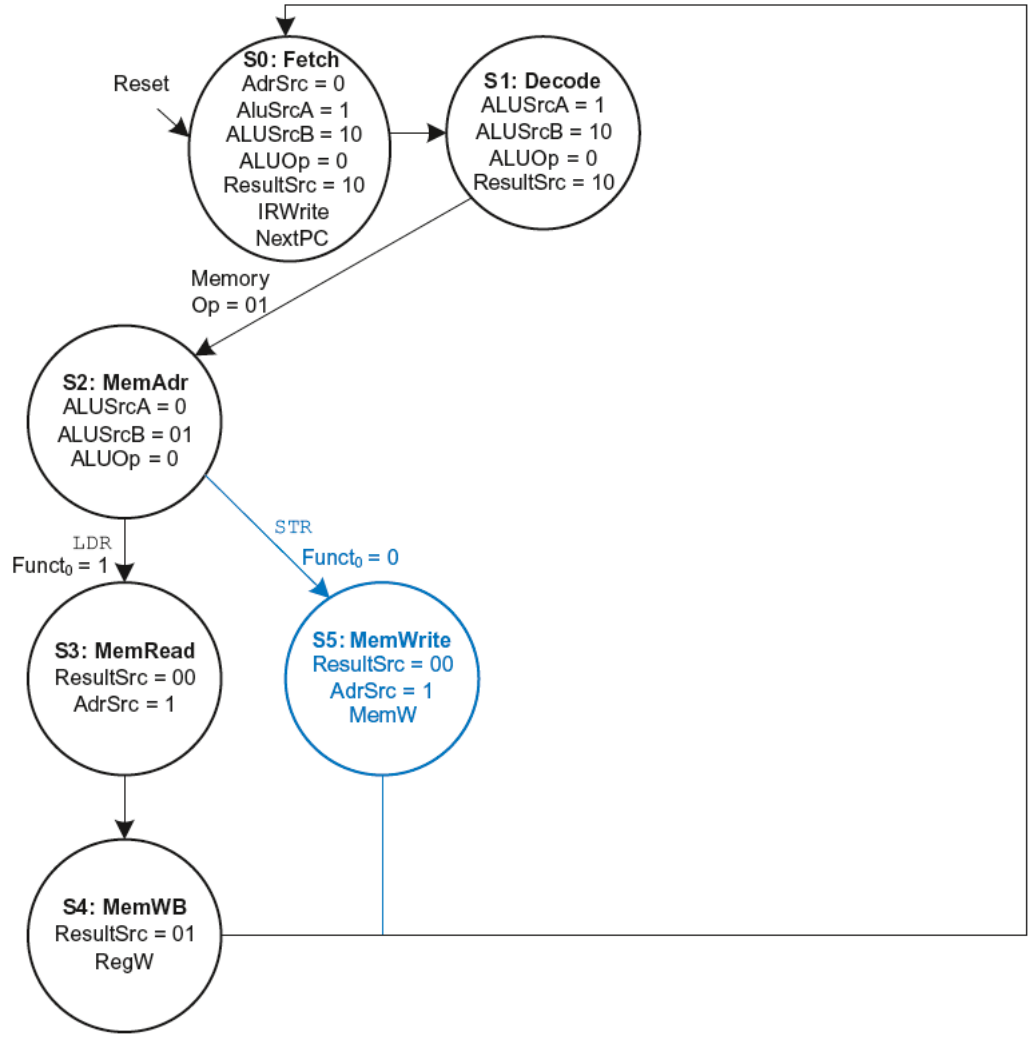


Main Controller FSM: LDR



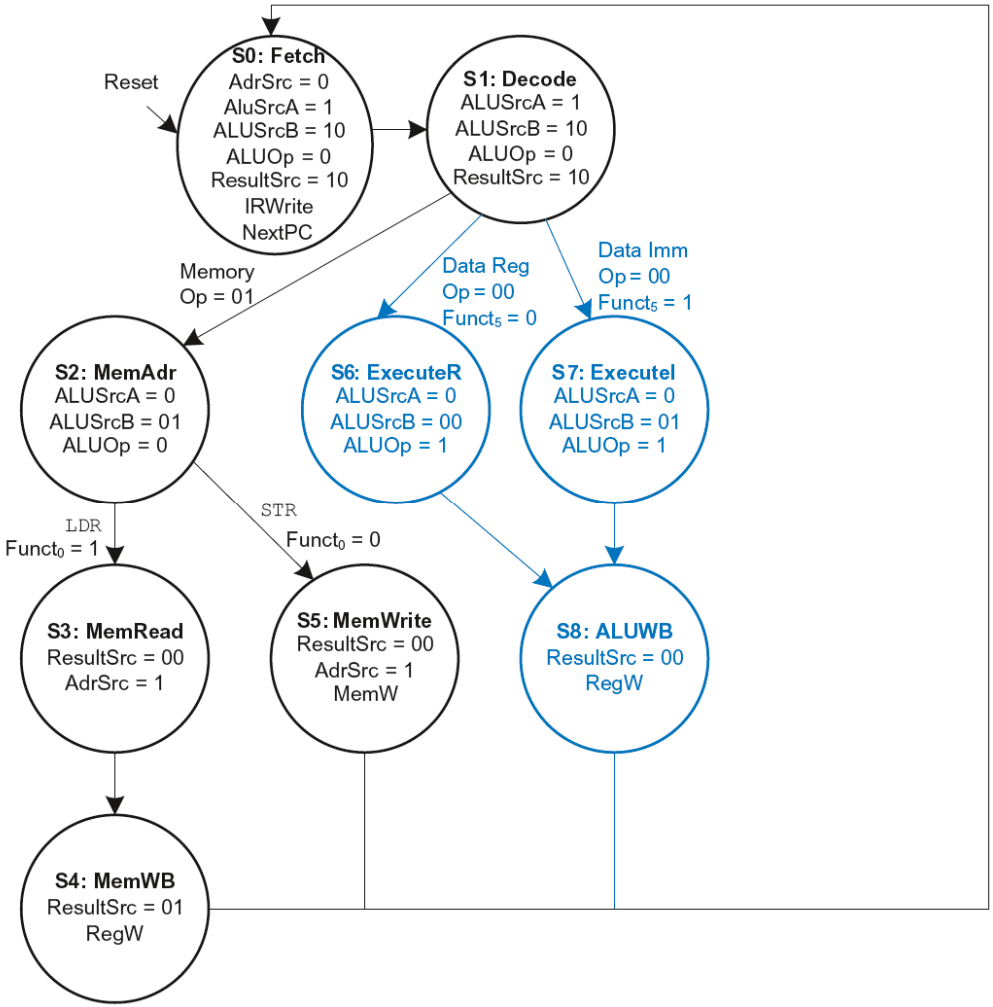
ELSEVIER

Main Controller FSM: STR

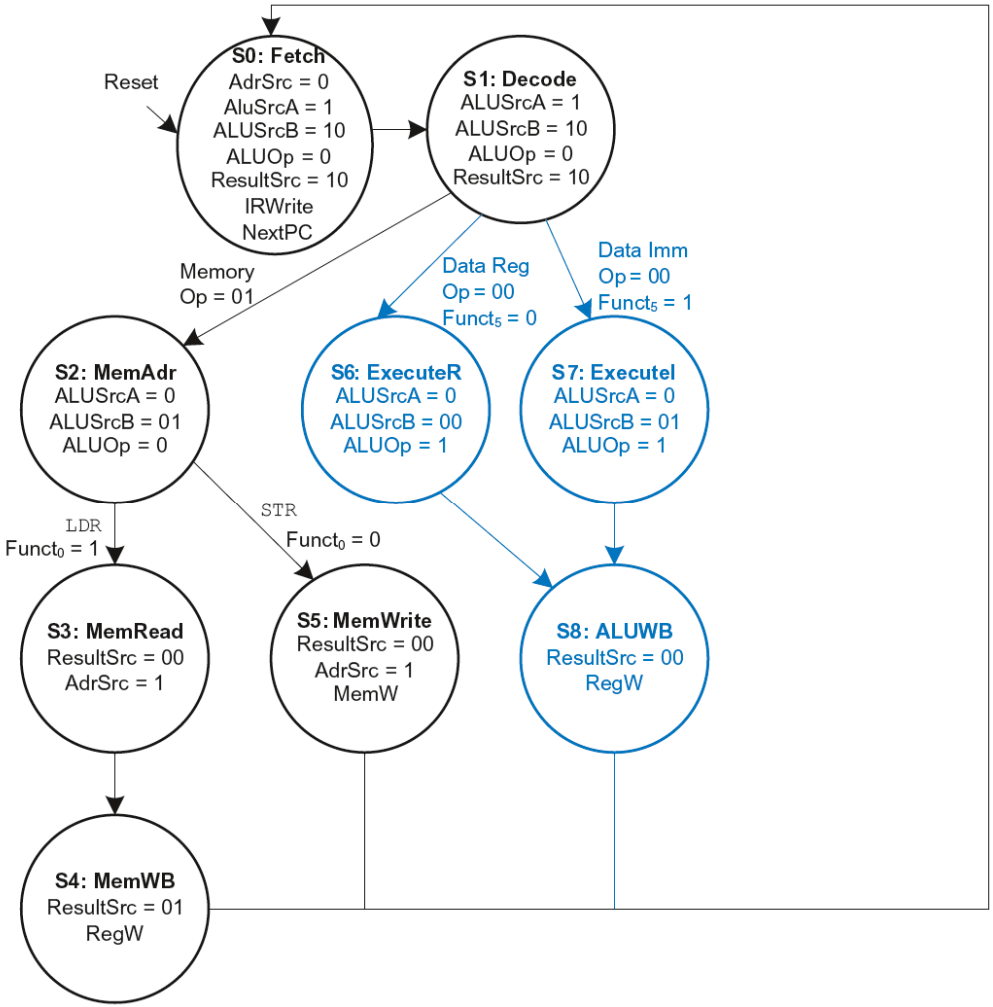


ELSEVIER

Main Controller FSM: Data-processing

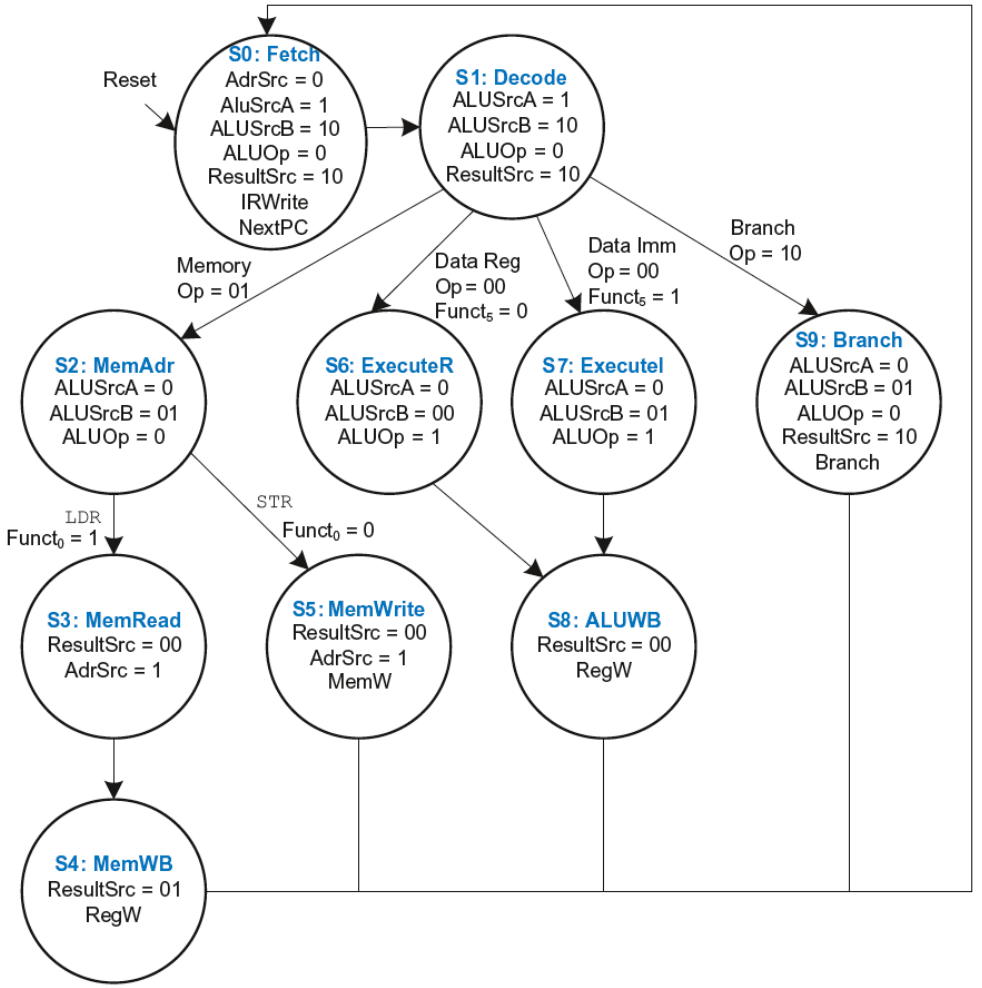


Main Controller FSM: Data-processing



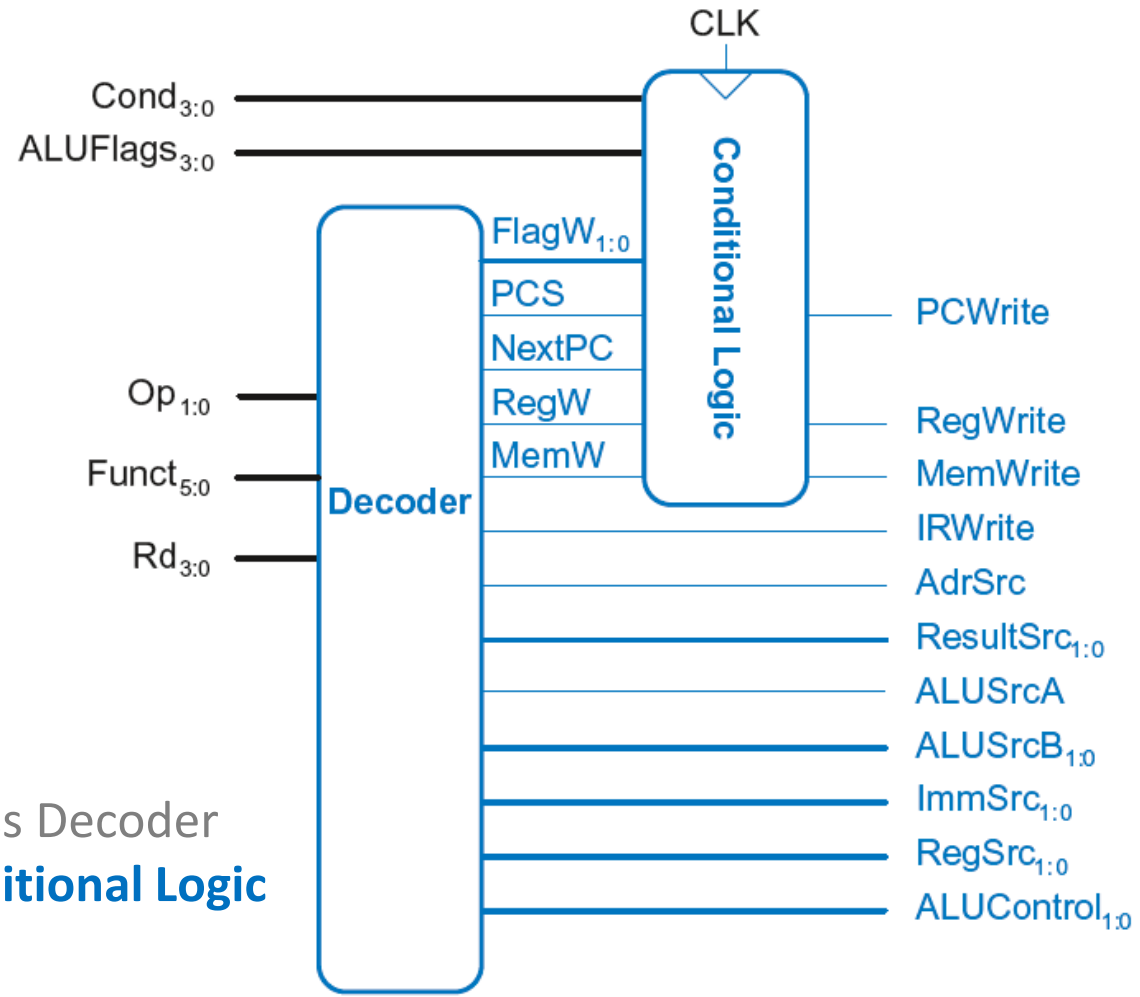
Multicycle Controller FSM

State	Datapath μOp
Fetch	Instr ← Mem[PC]; PC ← PC+4
Decode	ALUOut ← PC+4
MemAdr	ALUOut ← Rn + Imm
MemRead	Data ← Mem[ALUOut]
MemWB	Rd ← Data
MemWrite	Mem[ALUOut] ← Rd
ExecuteR	ALUOut ← Rn op Rm
ExecuteI	ALUOut ← Rn op Imm
ALUWB	Rd ← ALUOut
Branch	PC ← R15 + offset



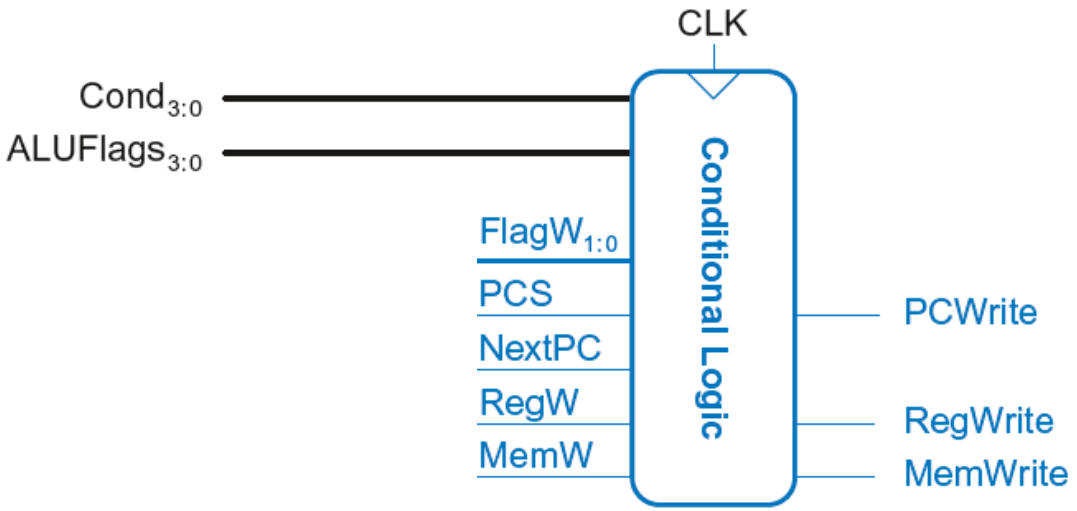
ELSEVIER

Multicycle Control

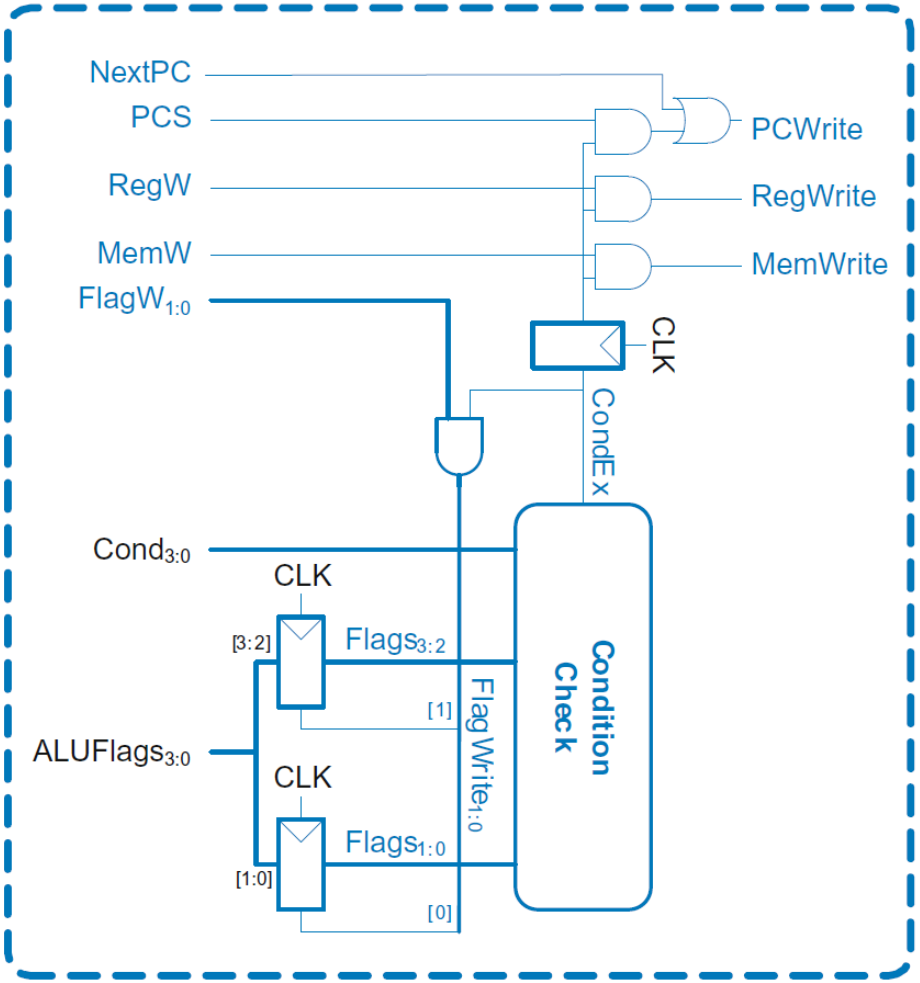


- First, discuss Decoder
- **Then, Conditional Logic**

Multicycle Control: Cond. Logic

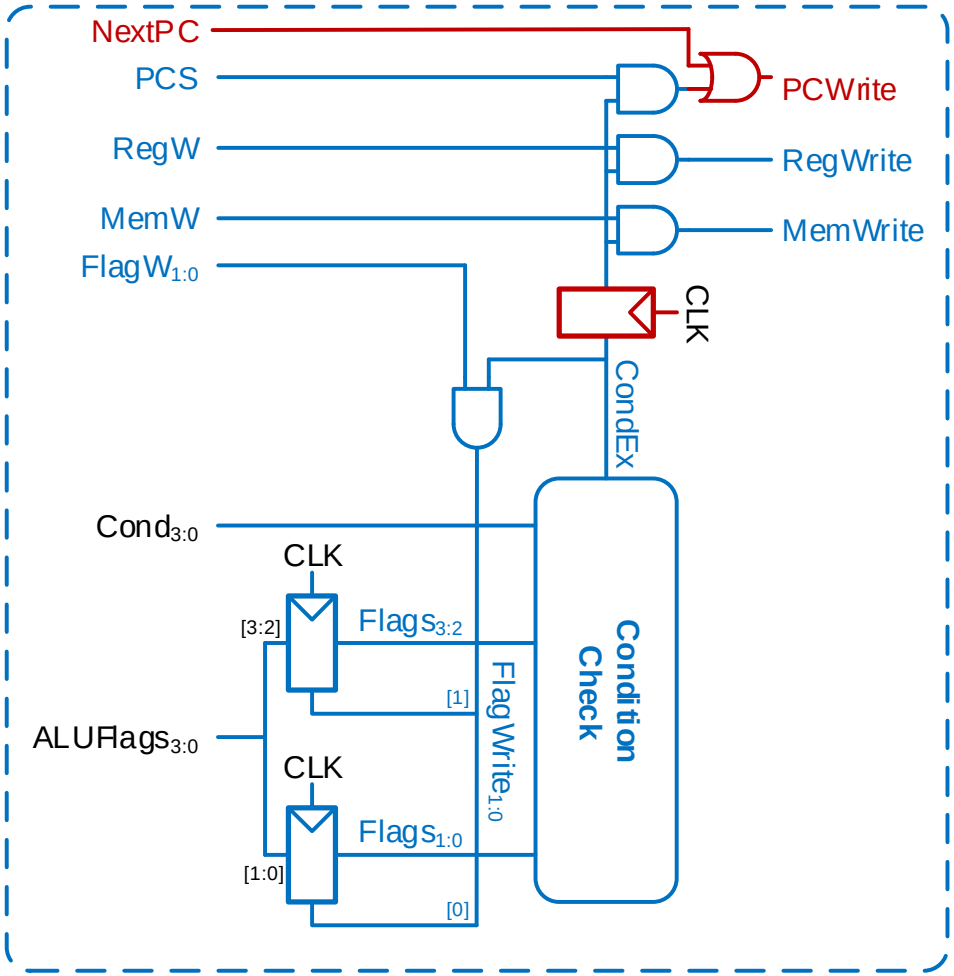


Single-Cycle Conditional Logic



ELSEVIER

Multicycle Conditional Logic



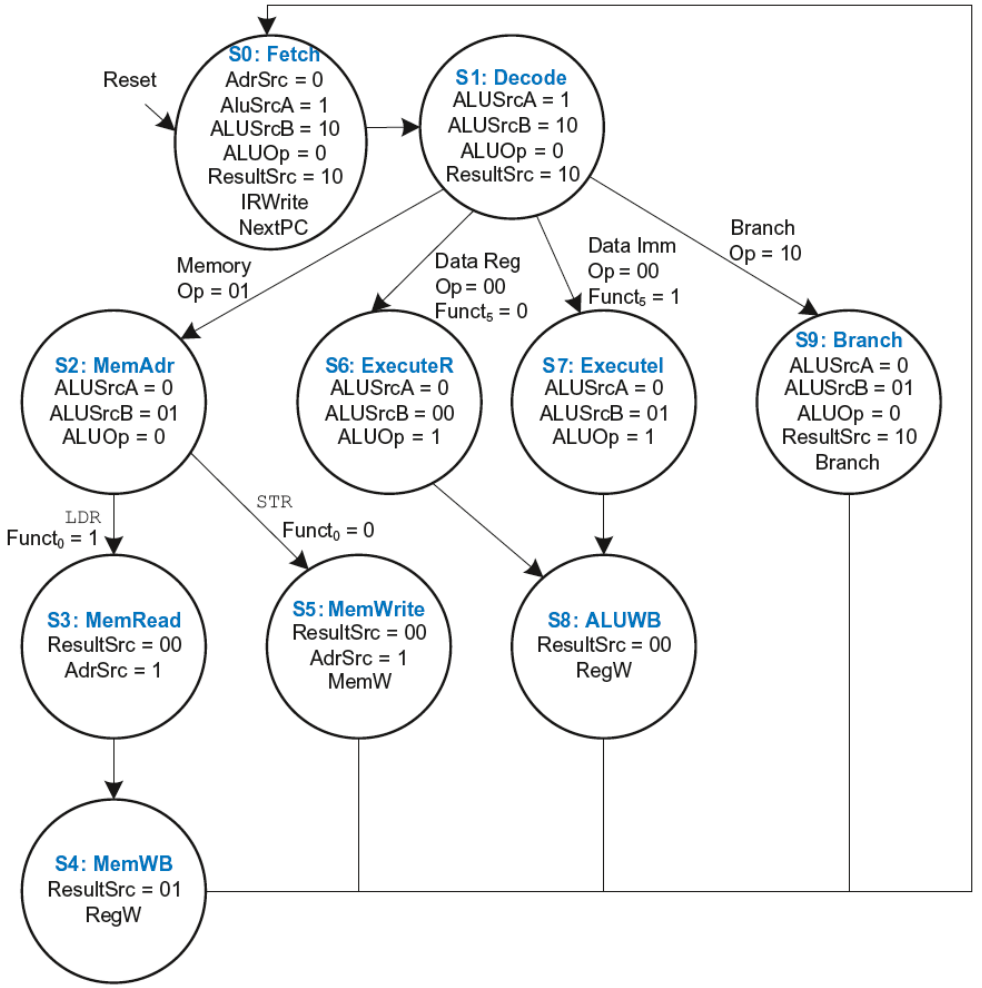
- **PCWrite** asserted in Fetch state
- **ExecuteI/ExecuteR** state:
  CondEx asserts
  ALUFlags generated
- **ALUWB** state:
  Flags updated
  CondEx changes
  PCWrite, ***RegWrite***, and ***MemWrite*** don't see change till new instruction (Fetch state)

Multicycle Processor Performance

- Instructions take different number of cycles.

Multicycle Controller FSM

State	Datapath μOp
Fetch	Instr ← Mem[PC]; PC ← PC+4
Decode	ALUOut ← PC+4
MemAdr	ALUOut ← Rn + Imm
MemRead	Data ← Mem[ALUOut]
MemWB	Rd ← Data
MemWrite	Mem[ALUOut] ← Rd
ExecuteR	ALUOut ← Rn op Rm
ExecuteI	ALUOut ← Rn op Imm
ALUWB	Rd ← ALUOut
Branch	PC ← R15 + offset



ELSEVIER

Multicycle Processor Performance

- Instructions take different number of cycles:
  - 3 cycles:
  - 4 cycles:
  - 5 cycles:

Multicycle Processor Performance

- Instructions take different number of cycles:
  - 3 cycles: B
  - 4 cycles: DP, STR
  - 5 cycles: LDR

Multicycle Processor Performance

- Instructions take different number of cycles:
  - 3 cycles: B
  - 4 cycles: DP, STR
  - 5 cycles: LDR
- CPI is weighted average
- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type

Multicycle Processor Performance

- Instructions take different number of cycles:
  - 3 cycles: B
  - 4 cycles: DP, STR
  - 5 cycles: LDR
- CPI is weighted average
- SPECINT2000 benchmark:
  - **25%** loads
  - **10%** stores
  - **13%** branches
  - **52%** R-type

Average CPI = (**0.13**)(3) + (**0.52 + 0.10**)(4) + (**0.25**)(5) = **4.12**

Multicycle Processor Performance

Multicycle critical path:

- Assumptions:
  - RF is faster than memory
  - writing memory is faster than reading memory

$$T_{c2} = t_{pcq} + 2t_{\mathbf{mux}} + \mathbf{max}(t_{\mathbf{ALU}} + t_{\mathbf{mux}}, t_{\mathbf{mem}}) + t_{\mathbf{setup}}$$

Multicycle 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

$T_{c2} = ?$

Multicycle 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

$$T_{c2} = t_{pcq} + 2t_{mux} + \max[t_{ALU} + t_{mux}, t_{mem}] + t_{setup}$$
$$= [40 + 2(25) + 200 + 50]\ \text{ps} = \mathbf{340\ ps}$$

Multicycle Performance Example

For a program with **100 billion** instructions executing on a **multicycle** ARM processor

- **CPI** = 4.12 cycles/instruction
- **Clock cycle time:** T_{c2} = 340 ps

Execution Time = ?

Multicycle Performance Example

For a program with **100 billion** instructions executing on a **multicycle** ARM processor

- **CPI** = 4.12 cycles/instruction
- **Clock cycle time:** T_{c2} = 340 ps

Execution Time = (# instructions) × CPI × T_c

$= (100 \times 10^9)(4.12)(340 \times 10^{-12})$

$=$ **140 seconds**

Multicycle Performance Example

For a program with **100 billion** instructions executing on a **multicycle** ARM processor

- **CPI** = 4.12 cycles/instruction
- **Clock cycle time:** T_{c2} = 340 ps

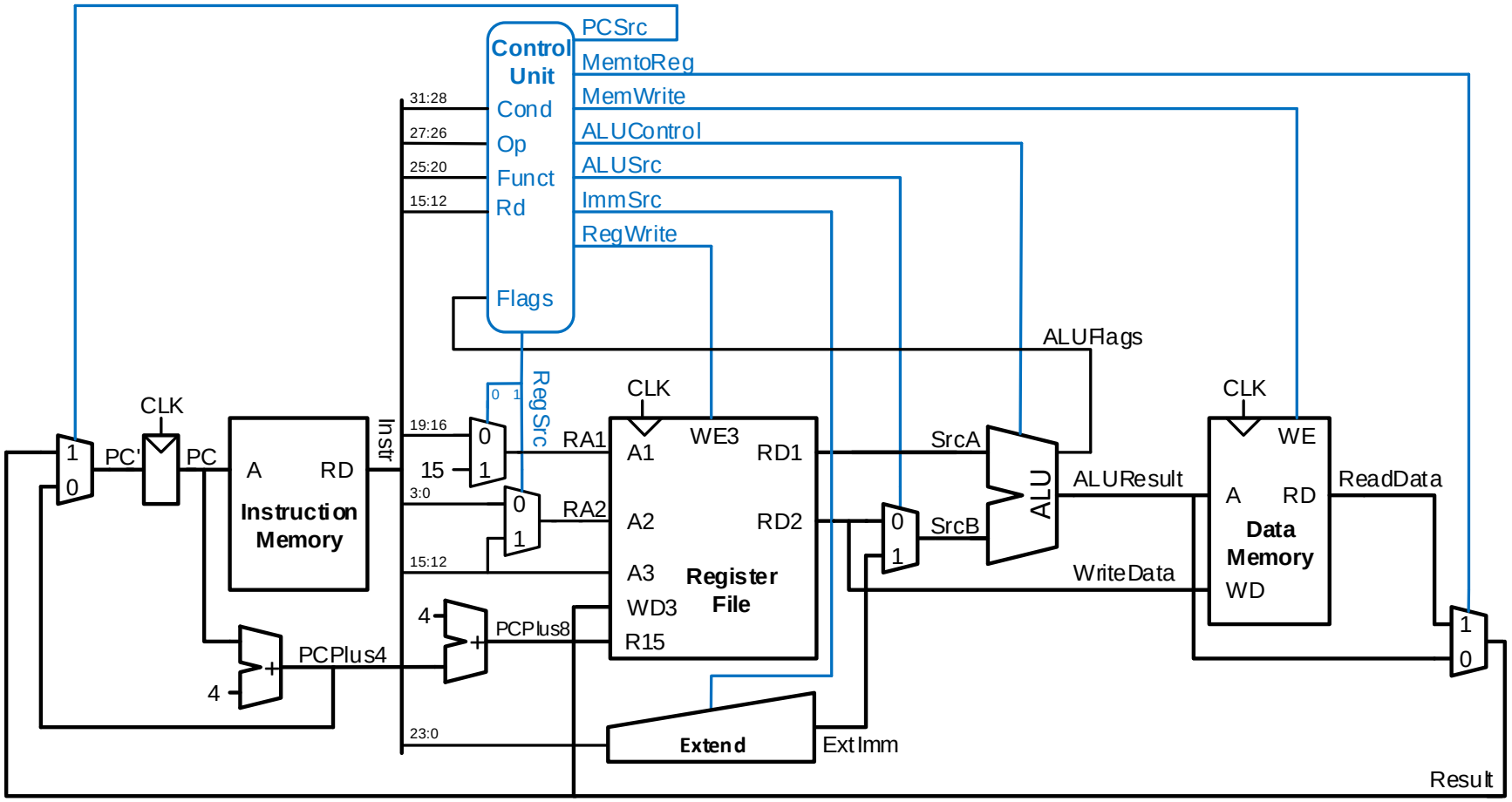
Execution Time = (# instructions) × CPI × T_c

$= (100 \times 10^9)(4.12)(340 \times 10^{-12})$

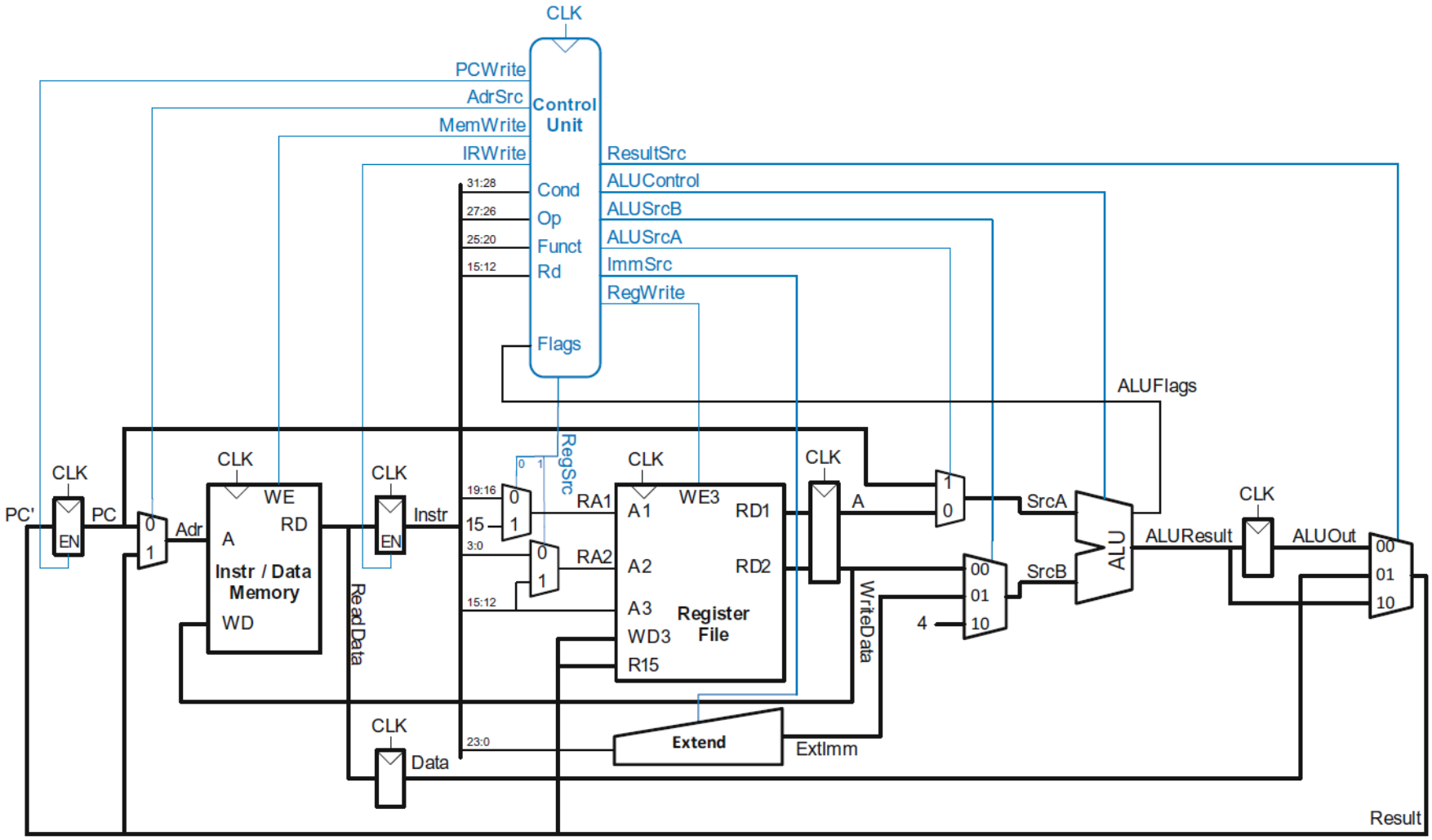
$=$ **140 seconds**

This is slower than the single-cycle processor (84 sec.)

Review: Single-Cycle ARM Processor



Review: Multicycle ARM Processor



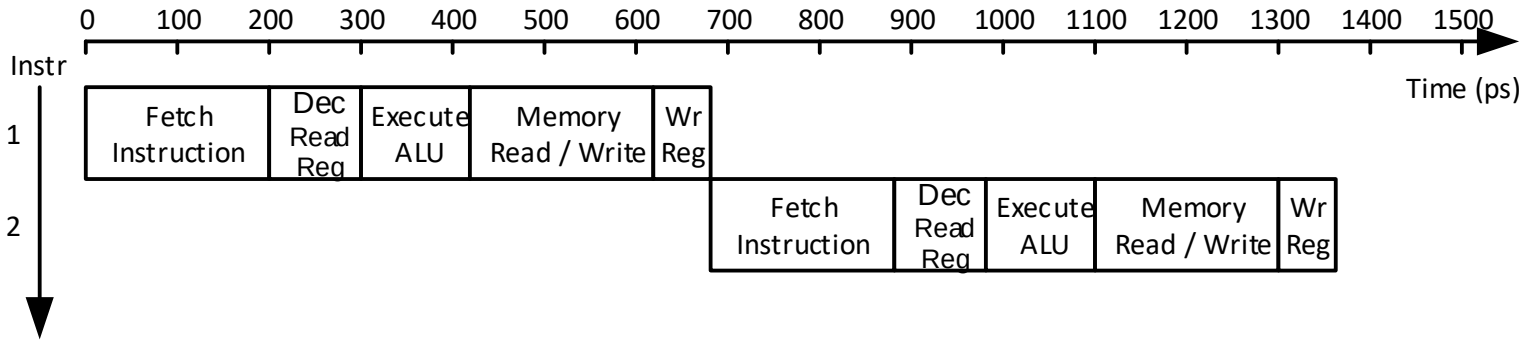
ELSEVIER

Pipelined ARM Processor

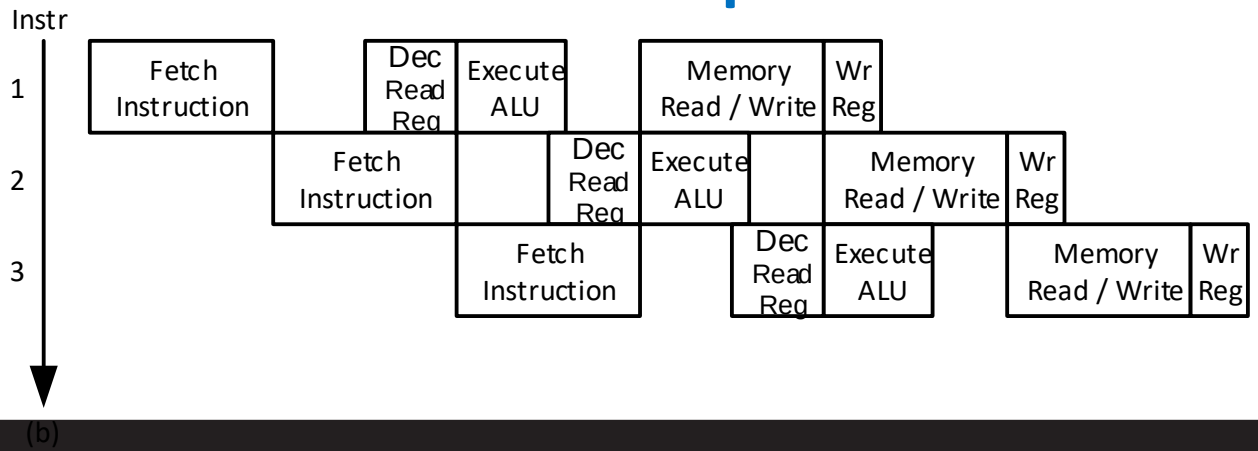
- Temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages

Single-Cycle vs. Pipelined

Single-Cycle



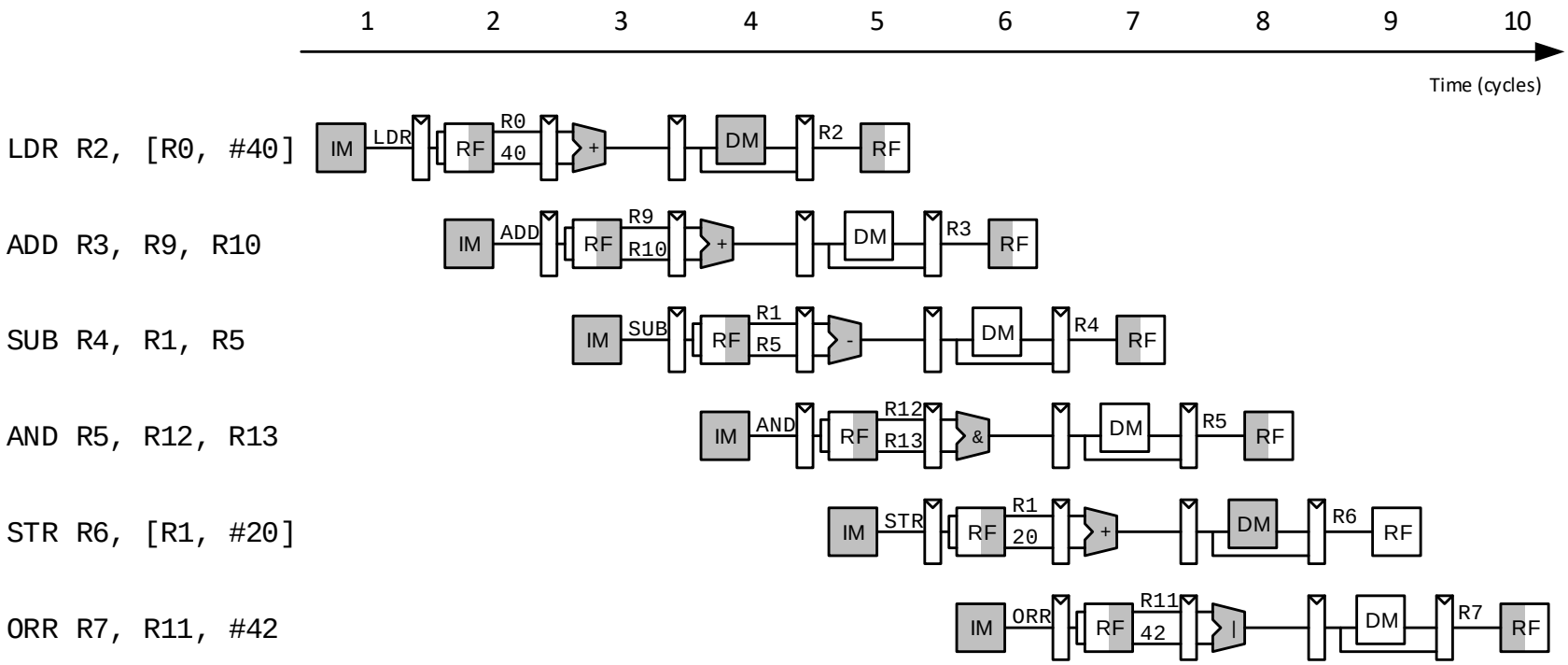
Pipelined



(b)

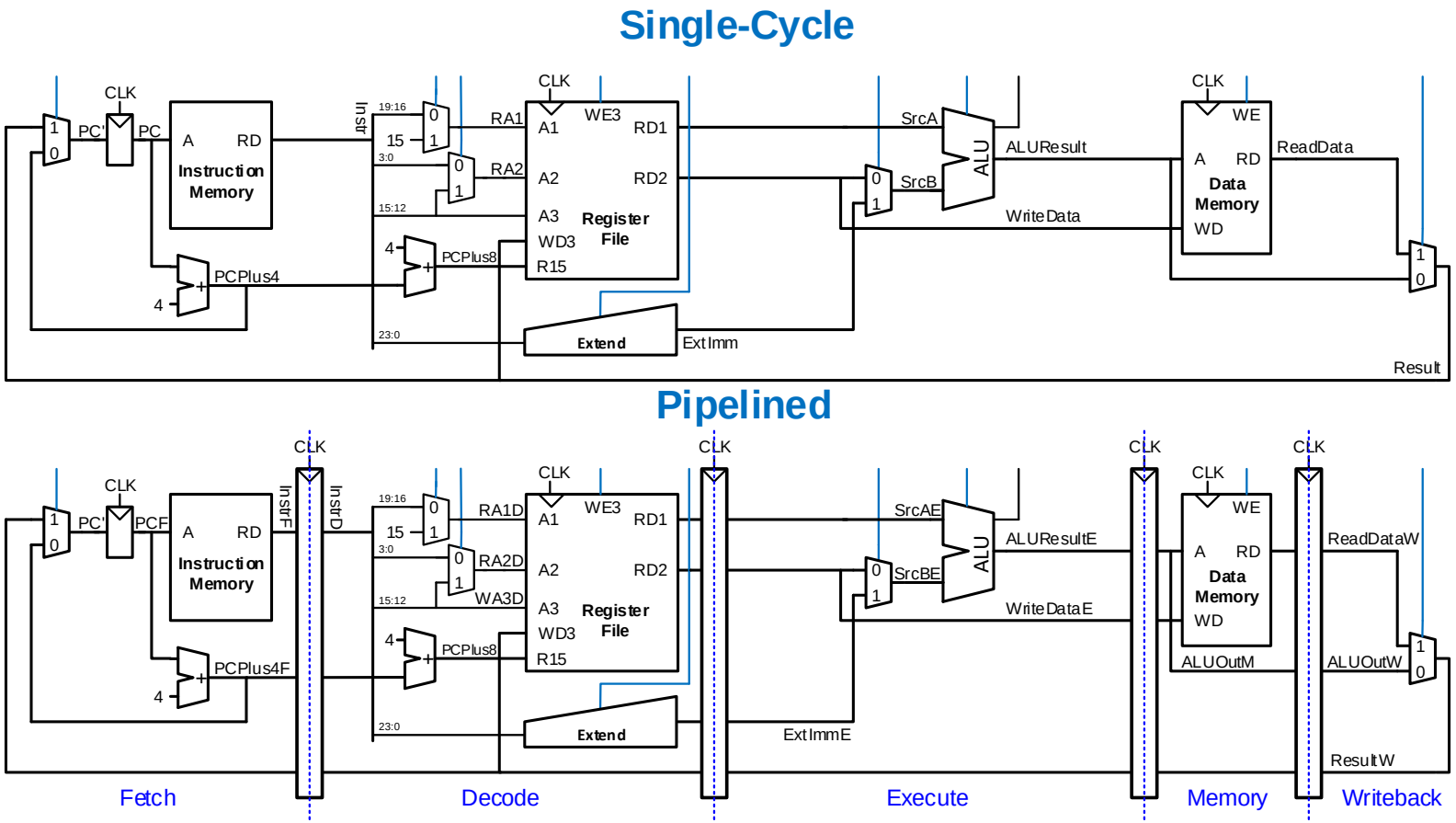
ELSEVIER

Pipelined Processor Abstraction

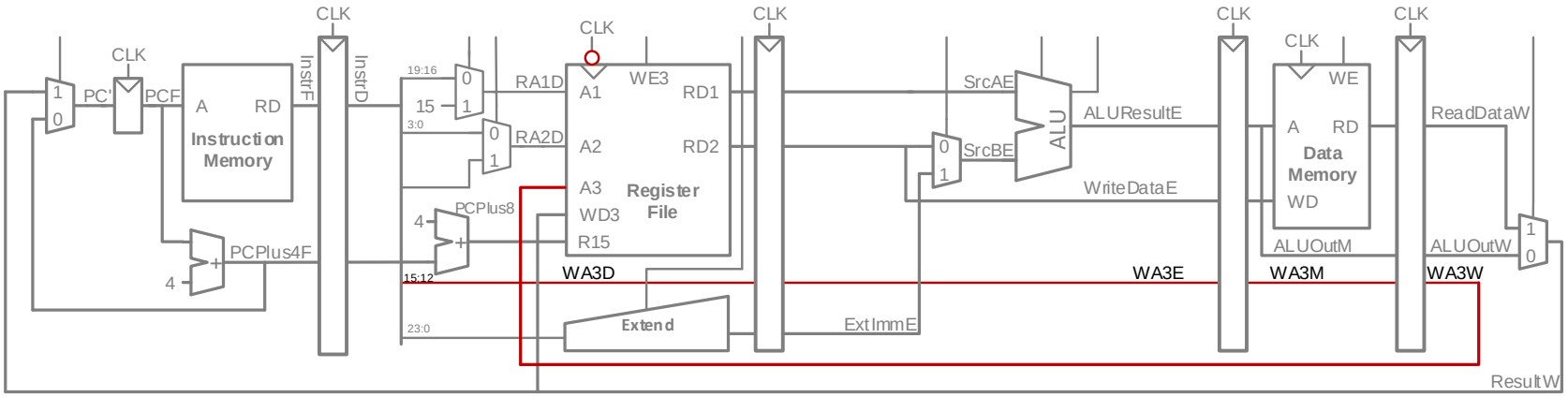


ELSEVIER

Single-Cycle & Pipelined Datapath

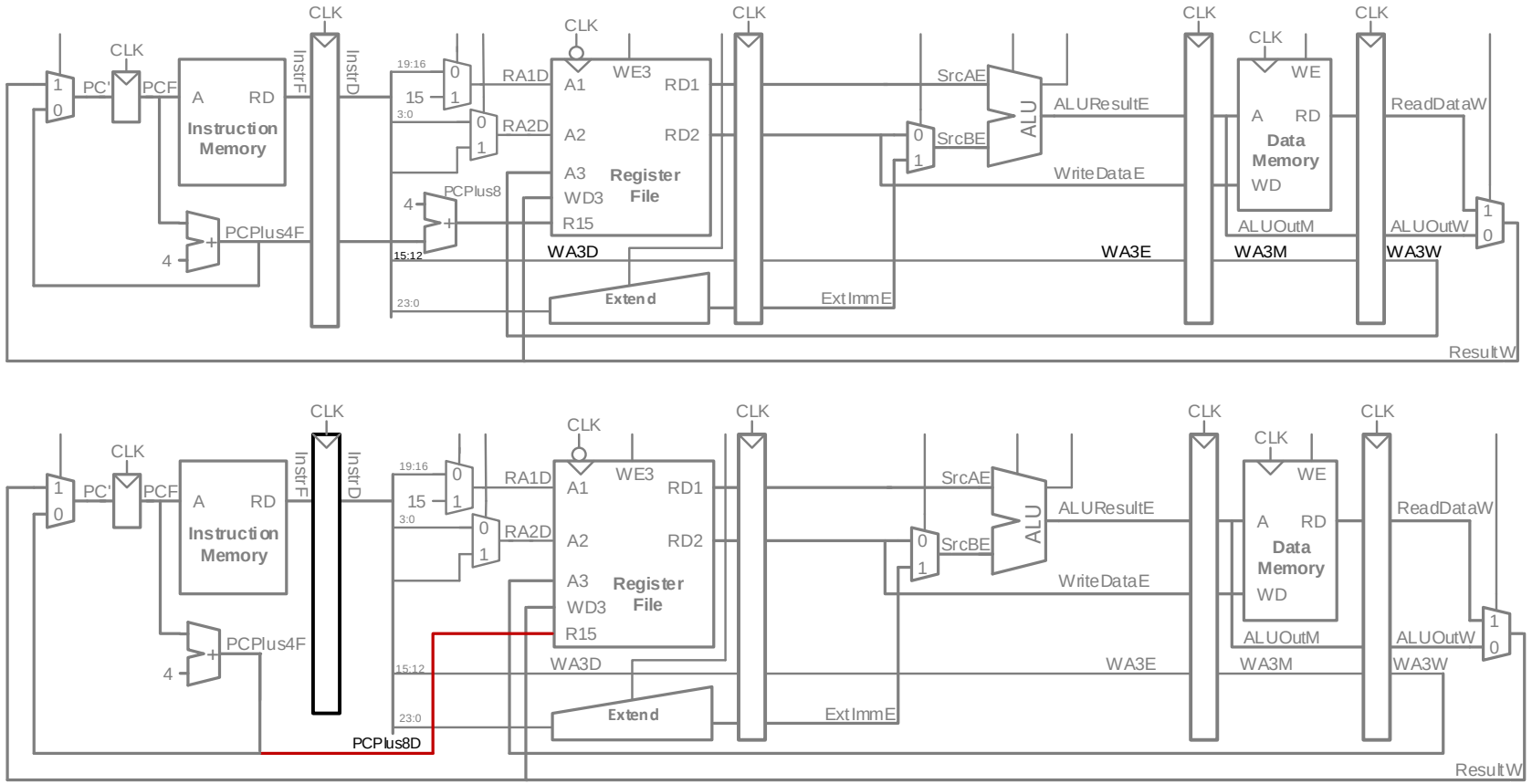


Corrected Pipelined Datapath



- ***WA3* must arrive at same time as *Result***
- **Register file written on falling edge of *CLK***

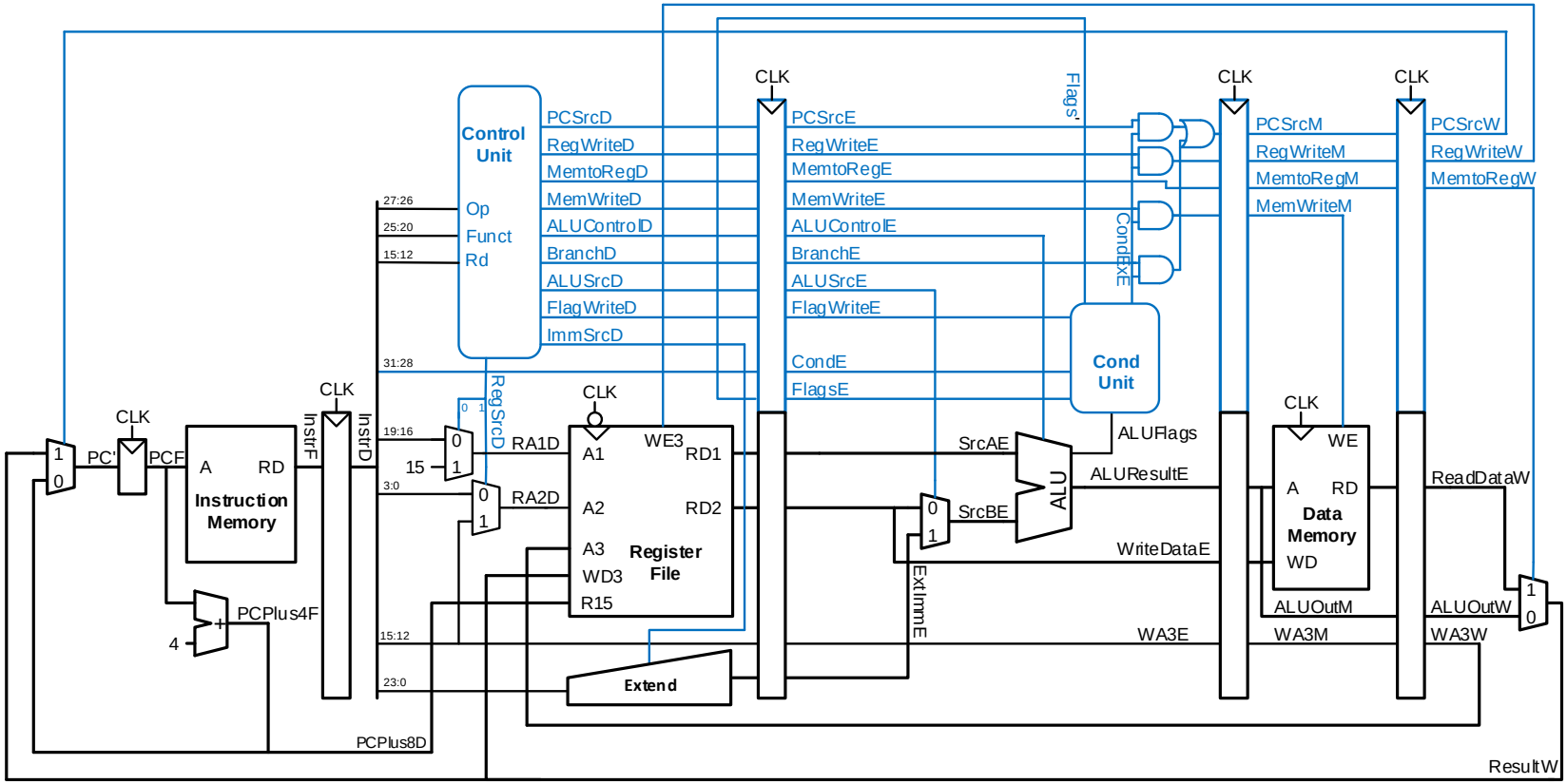
Optimized Pipelined Datapath



Remove adder by using *PCPlus4F* after *PC* has been updated to *PC*+4

ELSEVIER

Pipelined Processor Control



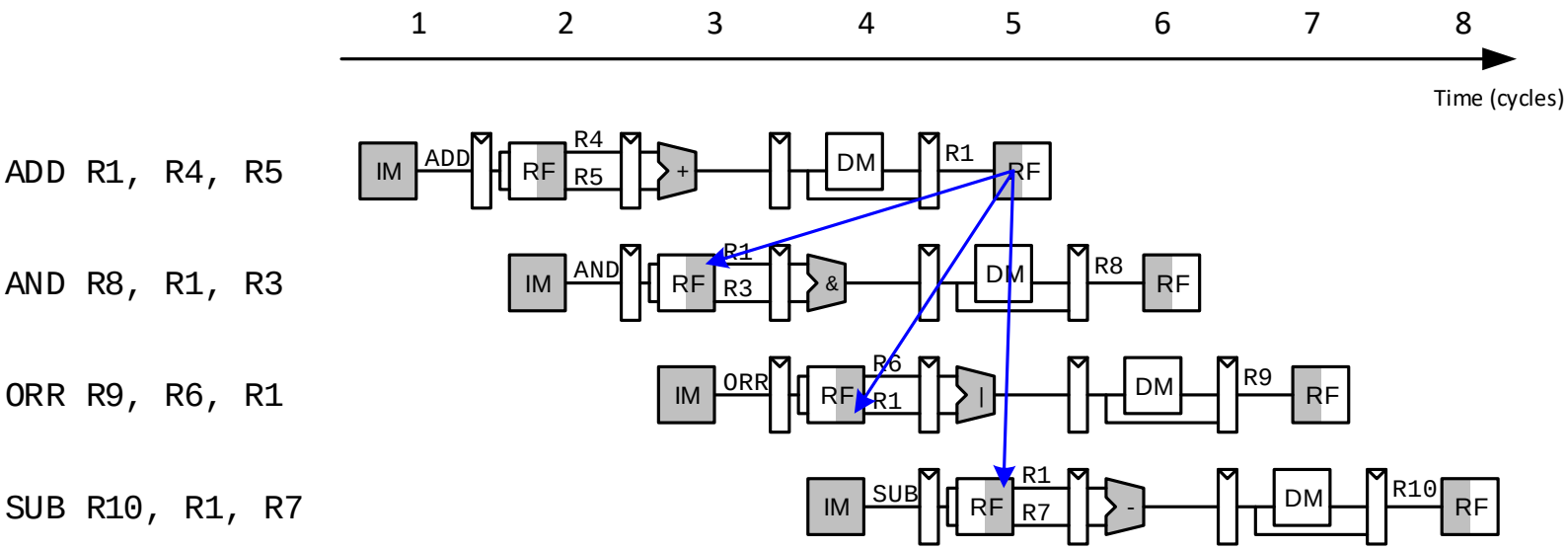
- **Same control unit as single-cycle processor**
- **Control delayed to proper pipeline stage**

ELSEVIER

Pipeline Hazards

- When an instruction depends on result from instruction that hasn't completed
- Types:
  - **Data hazard:** register value not yet written back to register file
  - **Control hazard:** next instruction not decided yet (caused by branch)

Data Hazard

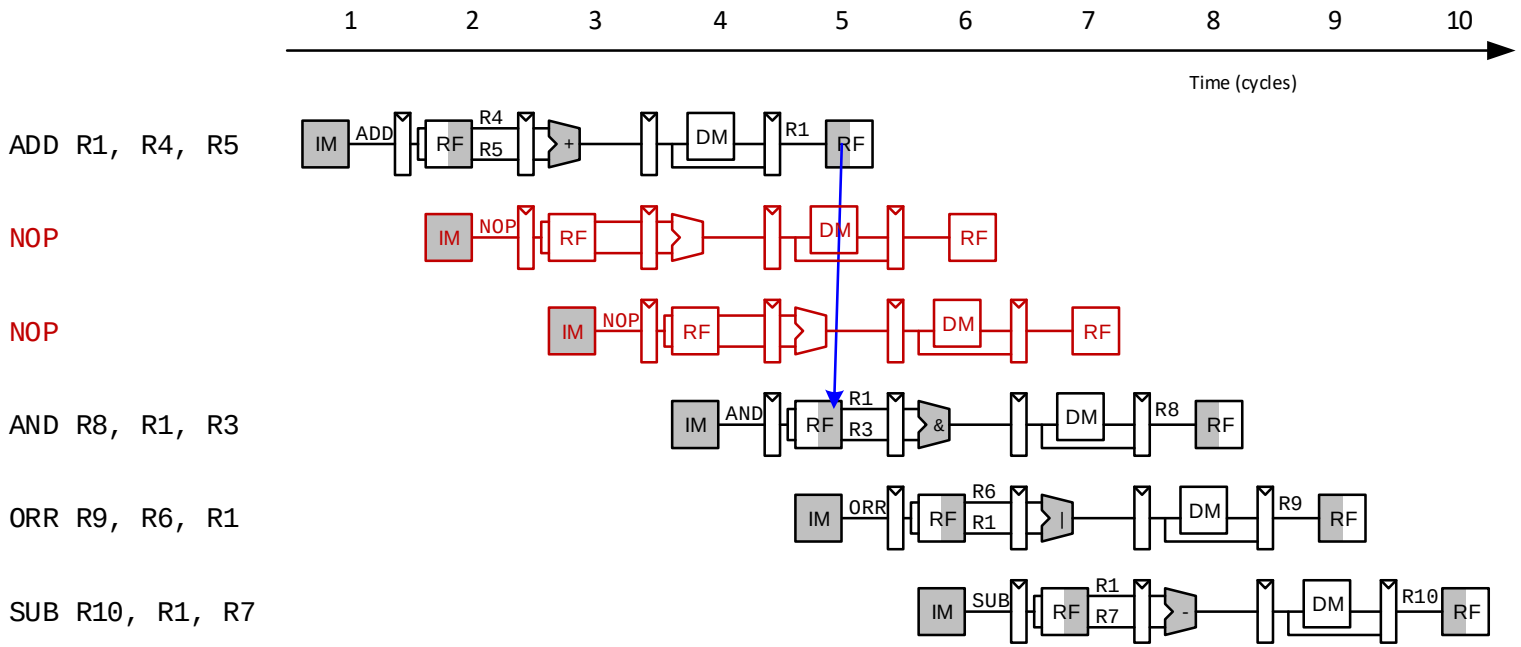

1
2
3
4
5
6
7
8
Time (cycles)
ADD R1, R4, R5
AND R8, R1, R3
ORR R9, R6, R1
SUB R10, R1, R7
IM
ADD
RF
R4
R5
+
DM
R1
RF
AND
R1
R3
&
R8
ORR
R6
R1
|
R9
SUB
R7
-
R10

Handling Data Hazards

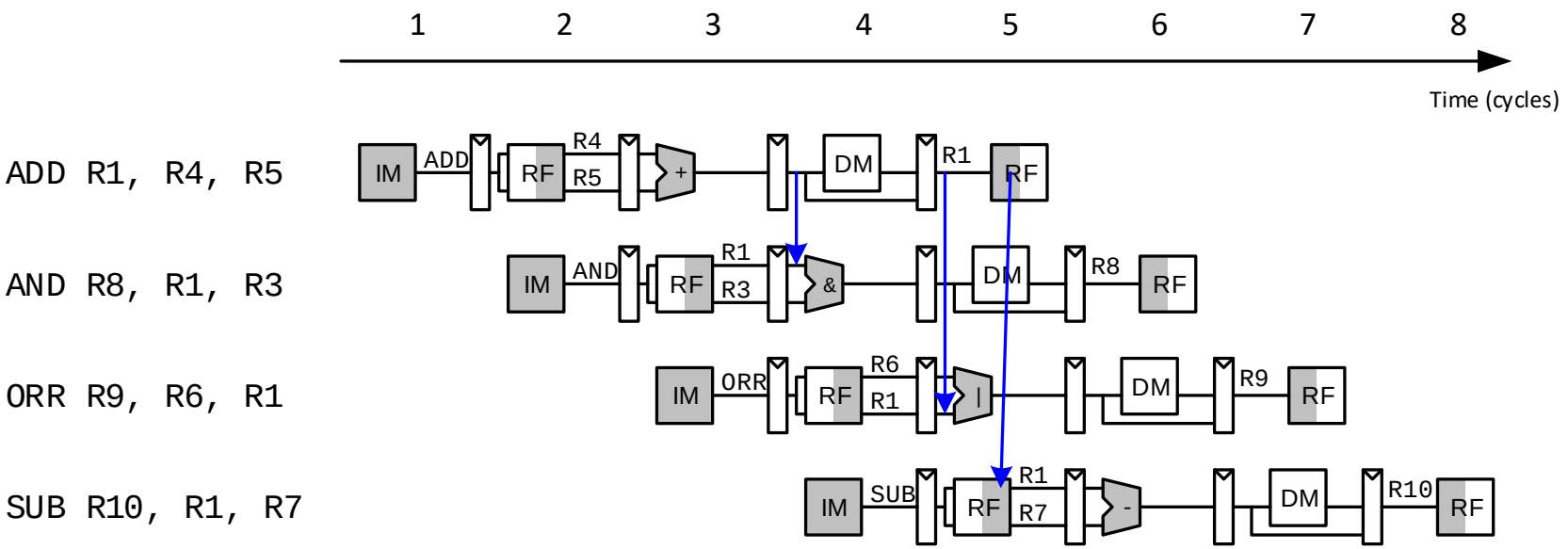
- Insert NOPs in code at compile time
- Rearrange code at compile time
- Forward data at run time
- Stall the processor at run time

Compile-Time Hazard Elimination

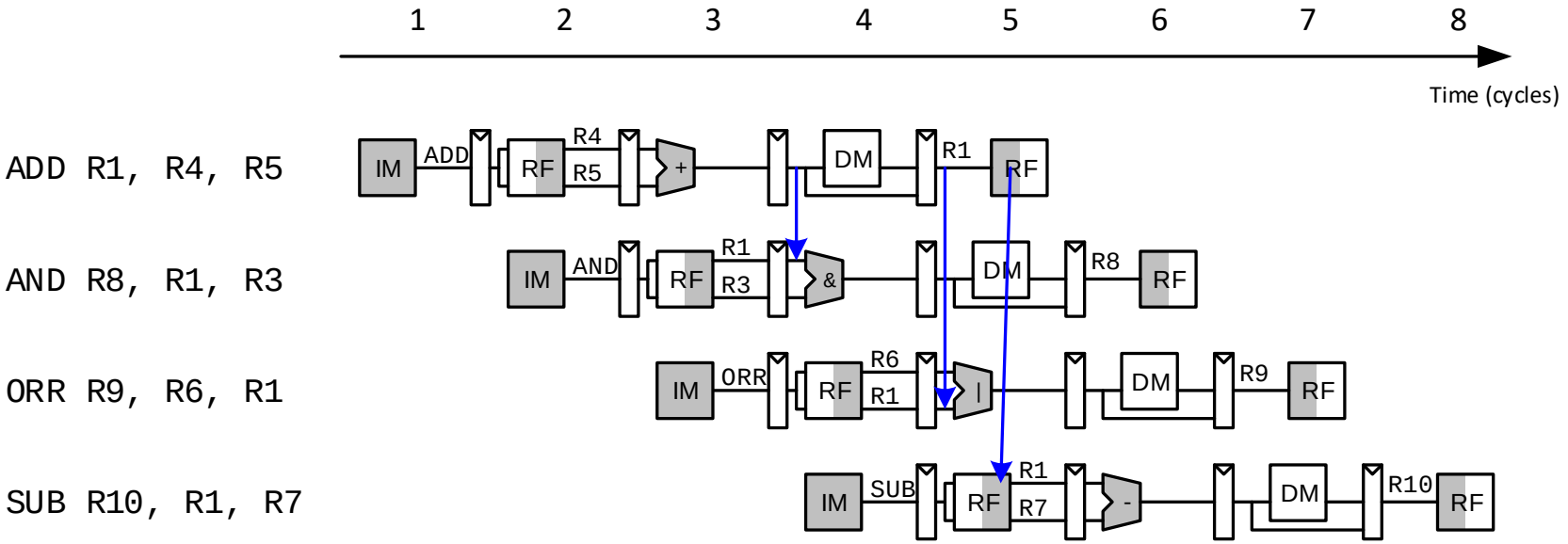
- Insert enough NOPs for result to be ready
- Or move independent useful instructions forward



Data Forwarding

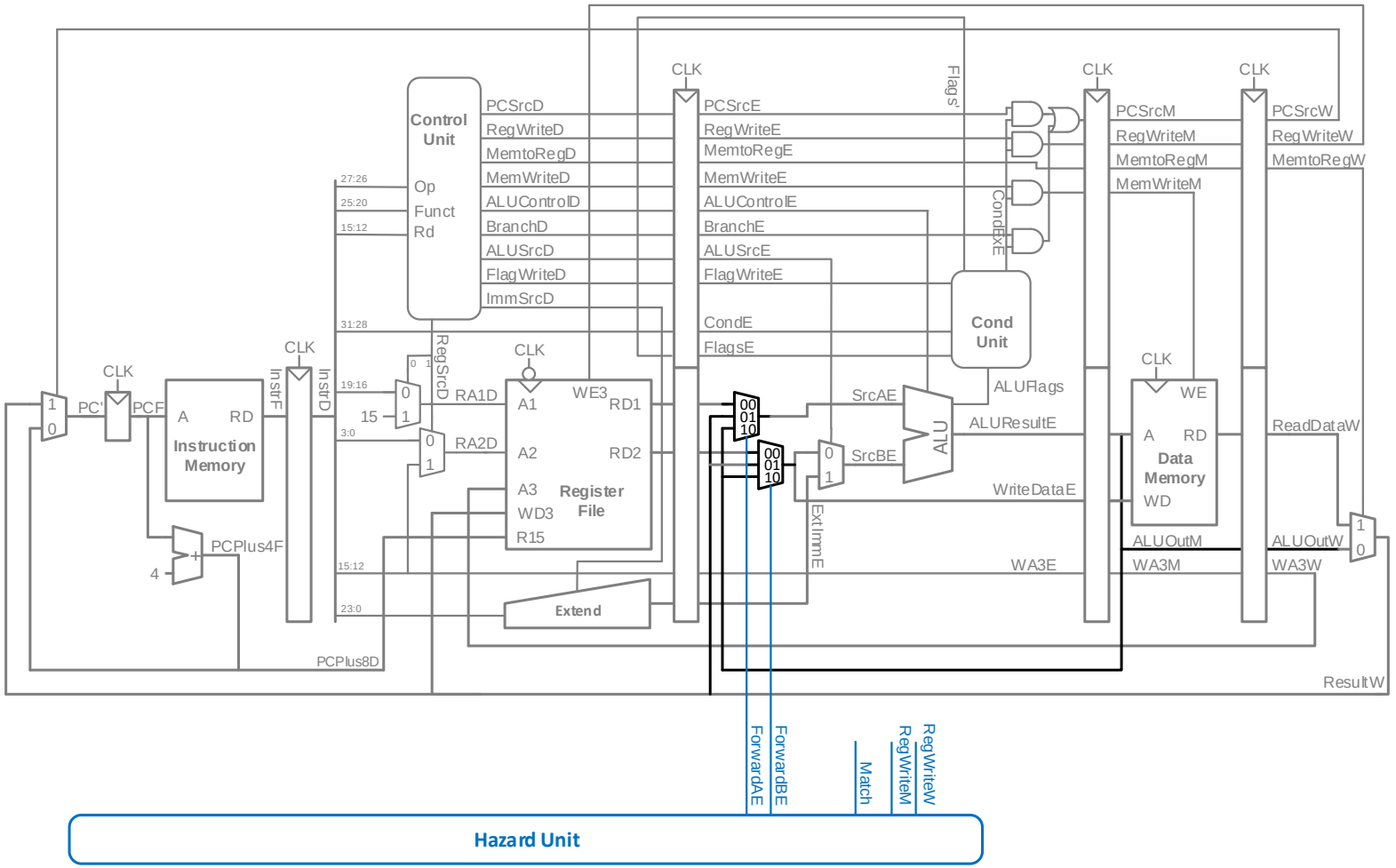


Data Forwarding



- Check if register read in Execute stage matches register written in Memory or Writeback stage
- If so, forward result

Data Forwarding



Data Forwarding

- **Execute** stage register matches **Memory** stage register?
  Match_1E_M = (RA1E == WA3M)
  Match_2E_M = (RA2E == WA3M)
- **Execute** stage register matches **Writeback** stage register?
  Match_1E_W = (RA1E == WA3W)
  Match_2E_W = (RA2E == WA3W)
- If it matches, forward result:

```
if      (Match_1E_M • RegWriteM)    ForwardAE = 10;
else if (Match_1E_W • RegWriteW)    ForwardAE = 01;
else                                ForwardAE = 00;
```

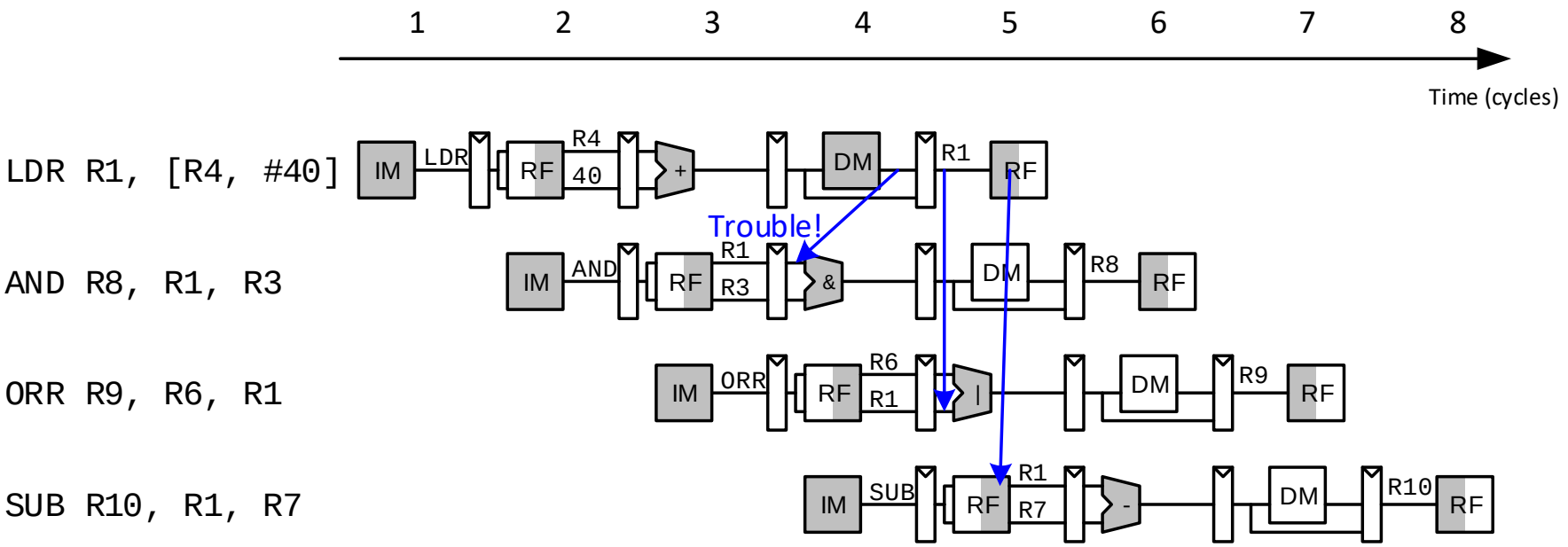
Data Forwarding

- **Execute** stage register matches **Memory** stage register?
  Match_1E_M = (RA1E == WA3M)
  Match_2E_M = (RA2E == WA3M)
- **Execute** stage register matches **Writeback** stage register?
  Match_1E_W = (RA1E == WA3W)
  Match_2E_W = (RA2E == WA3W)
- If it matches, forward result:

```
if      (Match_1E_M • RegWriteM)    ForwardAE = 10;
else if (Match_1E_W • RegWriteW)    ForwardAE = 01;
else                                ForwardAE = 00;
```

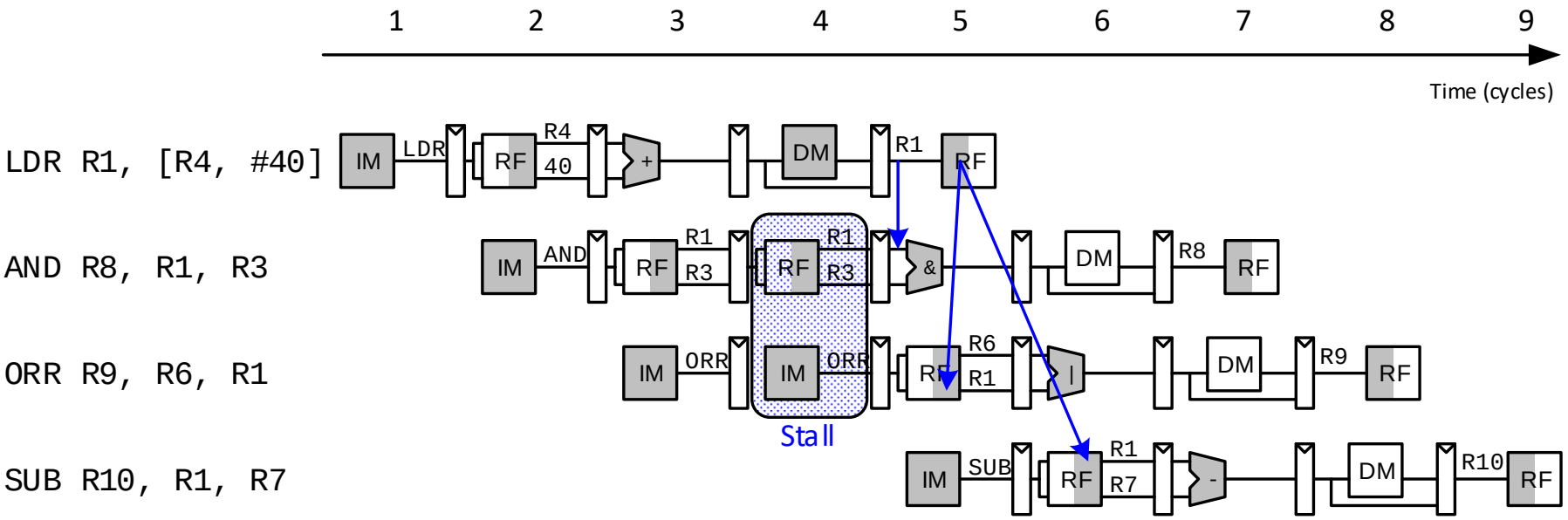
ForwardBE same but with Match2E

Stalling

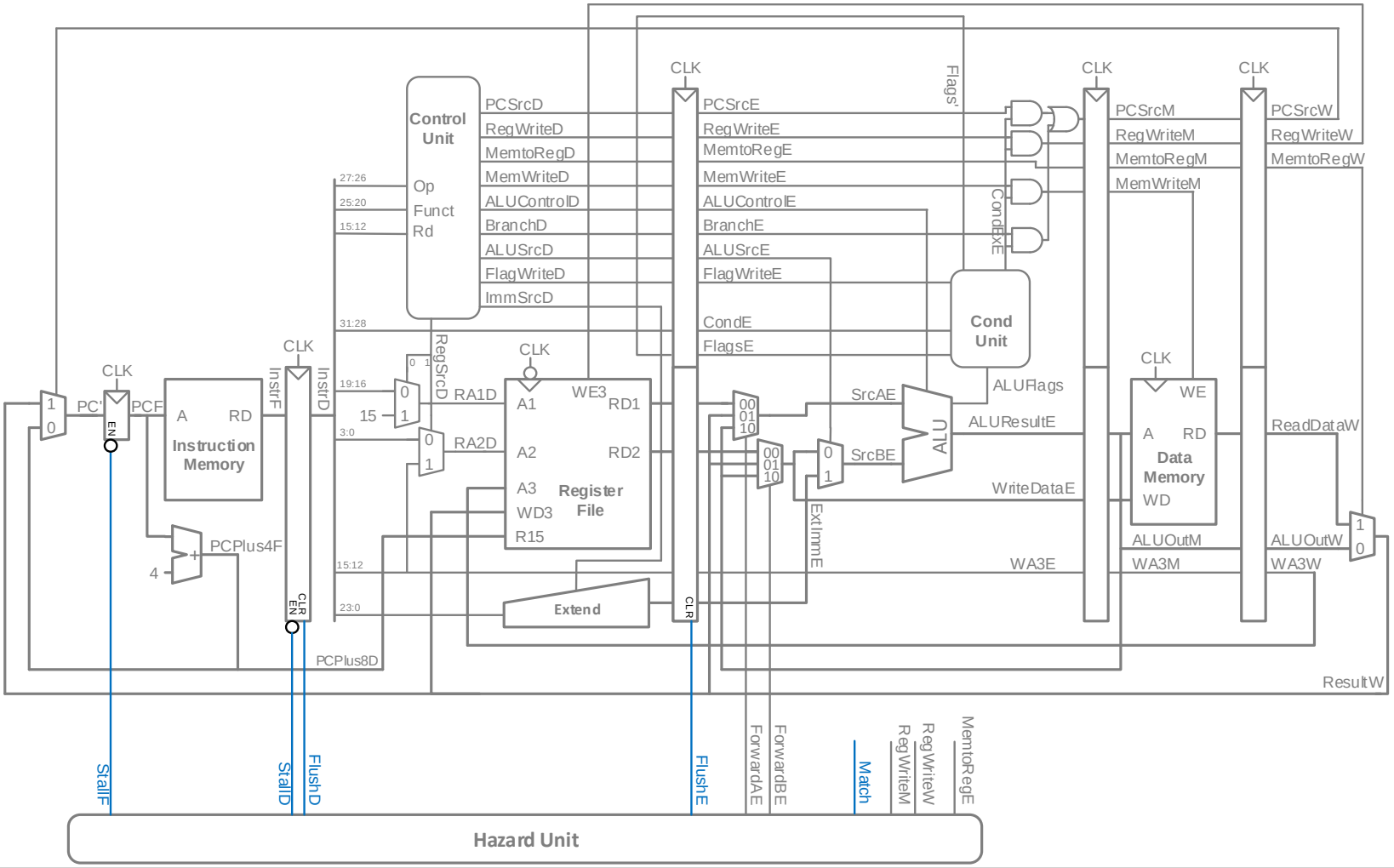


ELSEVIER

Stalling


1
2
3
4
5
6
7
8
9
Time (cycles)
LDR R1, [R4, #40]
AND R8, R1, R3
ORR R9, R6, R1
SUB R10, R1, R7
IM
LDR
RF
R4
40
+
DM
R1
AND
R3
&
R8
ORR
R6
|
R9
SUB
R7
-
R10
Stall

Stalling Hardware



Stalling Logic

- Is either source register in the Decode stage the same as the one being written in the Execute stage?

$$Match\_12D\_E = (RA1D == WA3E) + (RA2D == WA3E)$$

- Is a LDR in the Execute stage AND *Match_12D_E*?

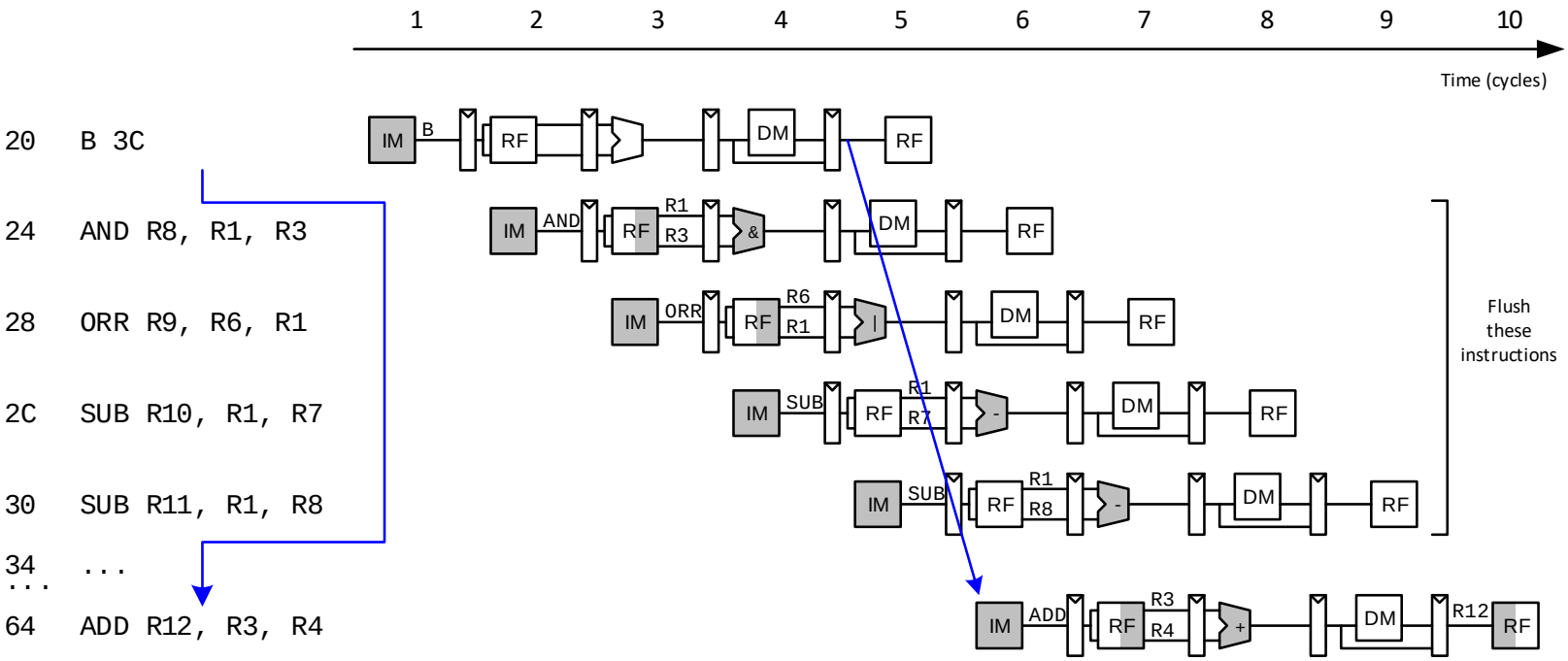
$$ldrstall = Match\_12D\_E \bullet MemtoRegE$$

$$StallF = StallD = FlushE = ldrstall$$

Control Hazards

- **B:**
  - branch not determined until the Writeback stage of pipeline
  - Instructions after branch fetched before branch occurs
  - These 4 instructions must be flushed if branch happens
- **Writes to PC (R15) similar**

Control Hazards



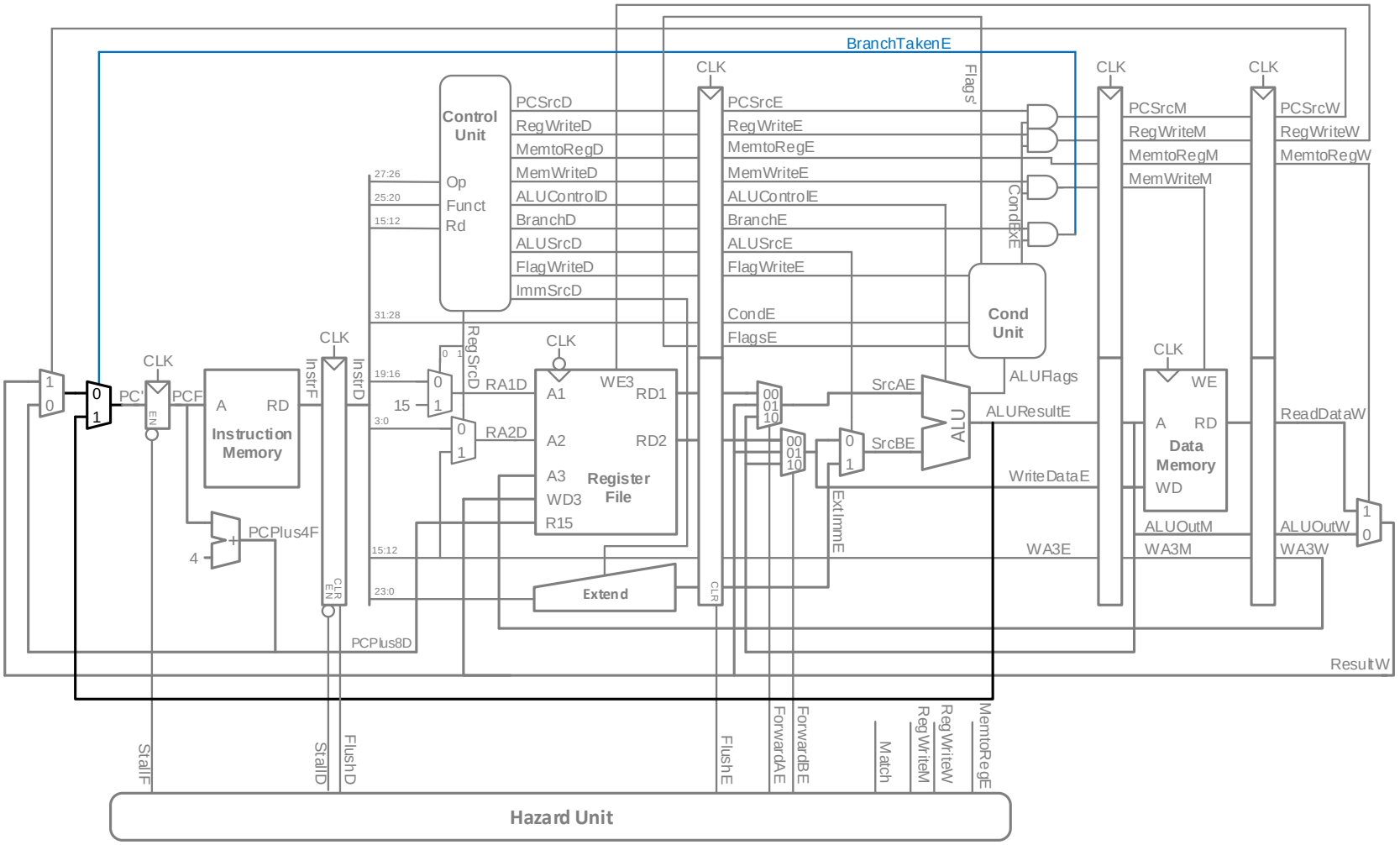
Branch misprediction penalty

- number of instruction flushed when branch is taken (4)
- May be reduced by determining BTA earlier

Early Branch Resolution

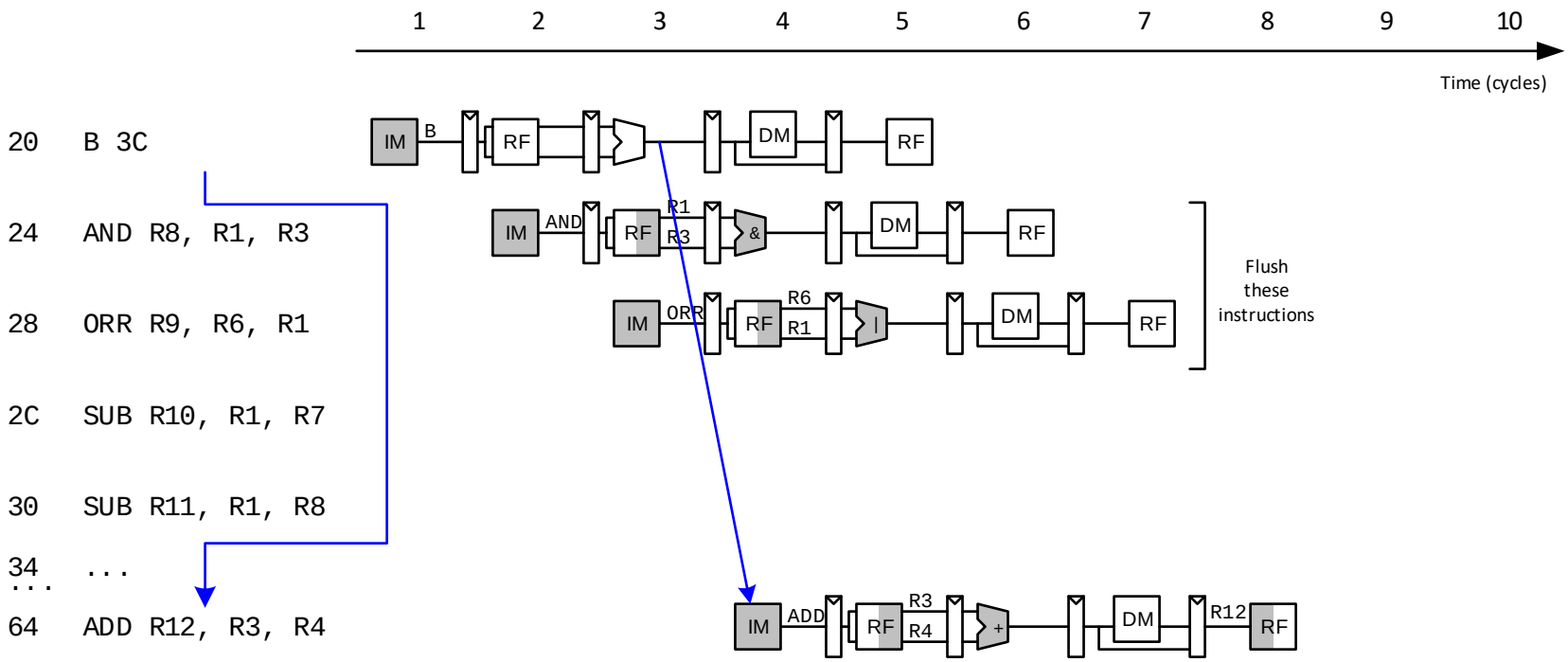
- **Determine BTA in Execute stage**
  - Branch misprediction penalty = 2 cycles
- **Hardware changes**
  - Add a branch multiplexer before *PC* register to select BTA from *ALUResultE*
  - Add *BranchTakenE* select signal for this multiplexer (only asserted if branch condition satisfied)
  - *PCSrcW* now only asserted for writes to *PC*

Pipelined processor with Early BTA



ELSEVIER

Control Hazards with Early BTA



Control Stalling Logic

- ***PCWrPendingF* = 1** if write to *PC* in Decode, Execute or Memory

  PCWrPendingF = PCSrcD + PCSrcE + PCSrcM

- **Stall Fetch** if *PCWrPendingF*

  StallF = ldrStallD + PCWrPendingF

- **Flush Decode** if *PCWrPendingF* OR *PC* is written in Writeback OR branch is taken

  FlushD = PCWrPendingF + PCSrcW + BranchTakenE

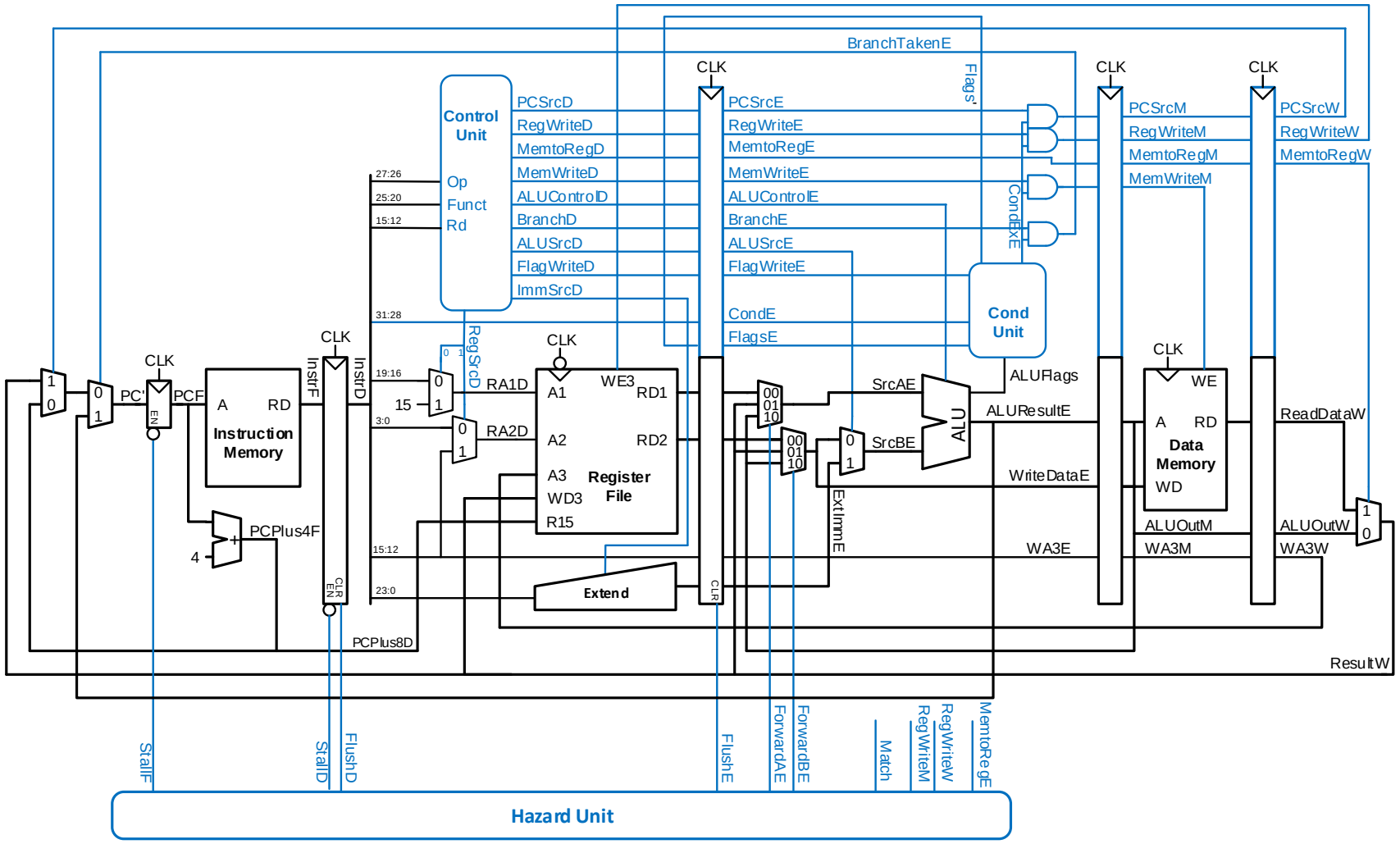
- **Flush Execute** if branch is taken

  FlushE = ldrStallD + BranchTakenE

- **Stall Decode** if *ldrStallD* (as before)

  StallD = ldrStallD

ARM Pipelined Processor with Hazard Unit



Pipelined Performance Example

- **SPECINT2000 benchmark:**
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type
- **Suppose:**
  - 40% of loads used by next instruction
  - 50% of branches mispredicted
- **What is the average CPI?**

Pipelined Performance Example

- **SPECINT2000 benchmark:**
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type
- **Suppose:**
  - 40% of loads used by next instruction
  - 50% of branches mispredicted
- **What is the average CPI?**
  - Load CPI = 1 when not stalling, 2 when stalling

    So, $\mathbf{CPI_{lw}} = 1(0.6) + 2(0.4) = 1.4$
  - Branch CPI = 1 when not stalling, 3 when stalling

    So, $\mathbf{CPI_{beq}} = 1(0.5) + 3(0.5) = 2$

Average CPI = $(0.25)(1.4) + (0.1)(1) + (0.13)(2) + (0.52)(1)$ **= 1.23**

Pipelined Performance

- Pipelined processor critical path:

$$T_{c3} = \max \left[\begin{array}{ll} t_{pcq} + t_{mem} + t_{setup} & \text{Fetch} \\ 2(t_{RFread} + t_{setup}) & \text{Decode} \\ t_{pcq} + 2t_{mux} + t_{ALU} + t_{setup} & \text{Execute} \\ t_{pcq} + t_{mem} + t_{setup} & \text{Memory} \\ 2(t_{pcq} + t_{mux} + t_{RFwrite}) & \text{Writeback} \end{array} \right]$$

Pipelined 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
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60
Register file write	$t_{RFwrite}$	70

Cycle time: $T_{c3} = ?$

Pipelined 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
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60
Register file write	$t_{RFwrite}$	70

Cycle time: $T_{c3} = 2(t_{RFread} + t_{setup})$

$= 2[100 + 50]$ ps **= 300 ps**

Pipelined Performance Example

Program with 100 billion instructions

Execution Time = (# instructions) × CPI × T_c

= $(100 \times 10^9)(1.23)(300 \times 10^{-12})$

= 36.9 seconds

Processor Performance Comparison

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
Single-cycle	84	1
Multicycle	140	0.6
Pipelined	36.9	2.28

Advanced Microarchitecture

- Deep Pipelining
- Micro-operations
- Branch Prediction
- Superscalar Processors
- Out of Order Processors
- Register Renaming
- SIMD
- Multithreading
- Multiprocessors

Deep Pipelining

- 10-20 stages typical
- Number of stages limited by:
  - Pipeline hazards
  - Sequencing overhead
  - Power
  - Cost

Micro-operations

- Decompose more complex instructions into a series of simple instructions called *micro-operations* (*micro-ops* or *μ-ops*)
- At run-time, complex instructions are decoded into one or more micro-ops
- Used heavily in CISC (complex instruction set computer) architectures (e.g., x86)
- Used for some ARM instructions, for example:

Complex Op	Micro-op Sequence
`LDR R1, [R2], #4`	`LDR R1, [R2]` `ADD R2, R2, #4`

Without u-ops, would need 2nd write port on the register file


I
DIGITAL
DESIGN