Microarchitecture

Computer Architecture

CS3501 – 2024II
PROF.: JGONZALEZ@UTEC.EDU.PE
SRC: HARRIS, HARRIS - DIGITAL DESIGN AND COMPUTER ARCHITECTURE

Executive Summary

- Motivation: Computer Architecture studies the ISA and Microarchitecture design.
- Problem: We need to detail an efficient and simple implementation of the ISA that enables the processor, known as Microarchitecture.
- Overview:
    - Definitions of processor microarchitecture and comparison with ISA.
    - Define the ARM single-cycle microarchitecture.
    - Details of the processor datapath and control.
- Conclusion: We can build a processor using building blocks to implement an ISA, this defines the microarchitecture.

Outline

Introduction

Single Cycle Processor

Datapath

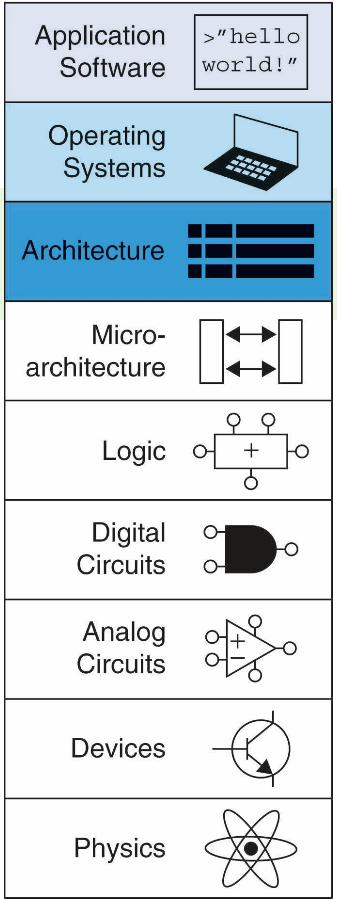
Control

Conclusions

Recall: ISA

- The **ISA** is the **interface between** what the **software** commands and what the **hardware** carries out
  - Specifies **how the programmer sees the instructions** to be executed

- **The ISA specifies**
  - **Memory organization**
    - Address space (ARM: 2^{32}, MIPS: 2^{32})
    - Addressability (ARM: 32 bits, MIPS: 32 bits)
    - Word- or Byte-addressable
  - **Register set**
    - R0 to R15 in ARM
    - 32 registers in MIPS
  - **Instruction set**
    - Opcodes
    - Data types
    - Addressing modes

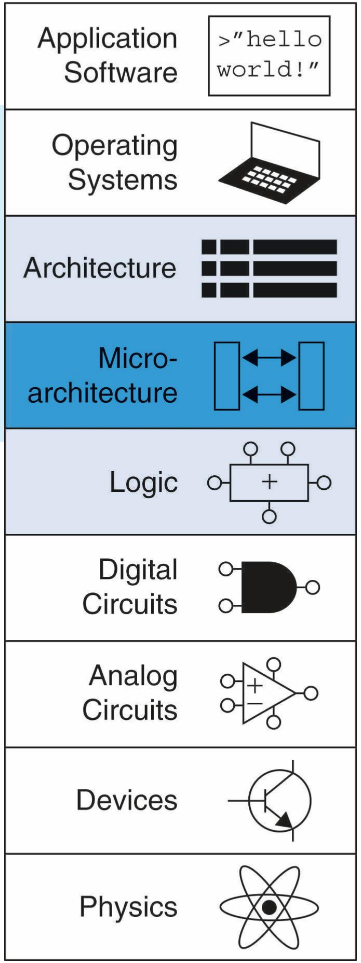


Microarchitecture Definition

- Microarchitecture: how to implement an architecture in hardware.
  - How the underlying implementation (invisible to software) actually executes instructions
    - Microarchitecture can execute instructions in any order **as long as it obeys the semantics specified by the ISA** when making the instruction results visible to software (to the programmer).

- **Two main parts in the processor:**
  - **Datapath:** functional blocks
  - **Control:** control signals

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



Computer Architecture

Computer Architecture = ISA + Microarchitecture

- All major **Instruction Set Architectures (ISAs)** today are based on the Von-Neumann model.
  - x86, ARM, MIPS, SPARC, Alpha, POWER, RISC-V, …

- **Microarchitecture (implementation)** **can be various as long as it satisfies** the specification (ISA)
  - **x86 ISA microarchitectures:** 286, 386, 486, Pentium, Pentium Pro, Pentium 4, Core, Kaby Lake, Coffee Lake, Cascade Lake, Tiger Lake, …

- **Microarchitecture** usually **changes faster** than **ISA**
  - **Few ISAs** but **many microarchitectures.**
  - In general **Computer Architecture** meets **functional, performance, energy consumption, cost, and other specific goals** (availability, reliability and correctness, Time to Market, Security, safety, predictability, etc.)

ISA vs Microarchitecture

ISA:

- Instructions
  - Opcodes, Addressing Modes, Data Types
  - Instruction Types and Formats
  - Registers, Condition Codes
- Memory
  - Address space, Addressability, Alignment
  - Virtual memory management
- Call, Interrupt/Exception Handling
- Access Control, Priority/Privilege
- I/O: memory-mapped vs. instr.
- Task/thread Management
- Power and Thermal Management
- Multi-threading support, Multiprocessor support

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Volume 1:
Basic Architecture

Microarchitecture:

- **Implementation of the ISA under specific design constraints and goals**
- **Anything done in hardware without exposure to software**
  - Pipelining
  - In-order versus out-of-order instruction execution
  - Memory access scheduling policy
  - Speculative execution
  - Superscalar processing (multiple instruction issue?)
  - Clock gating
  - Caching? Levels, size, associativity, replacement policy
  - Prefetching?
  - Voltage/frequency scaling?
  - Error correction?

Processor Performance

Different microarchitectures, different perfor

- **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

More about performance, later in lecture.

Outline

Introduction

Single Cycle Processor

Datapath

Control

Conclusions

Single Cycle Processor

- **Each instruction takes a single clock cycle to execute**
- **Execution with only combinational logic** is used to implement instruction execution
  - **No intermediate invisible** state updates.

AS = Architectural state

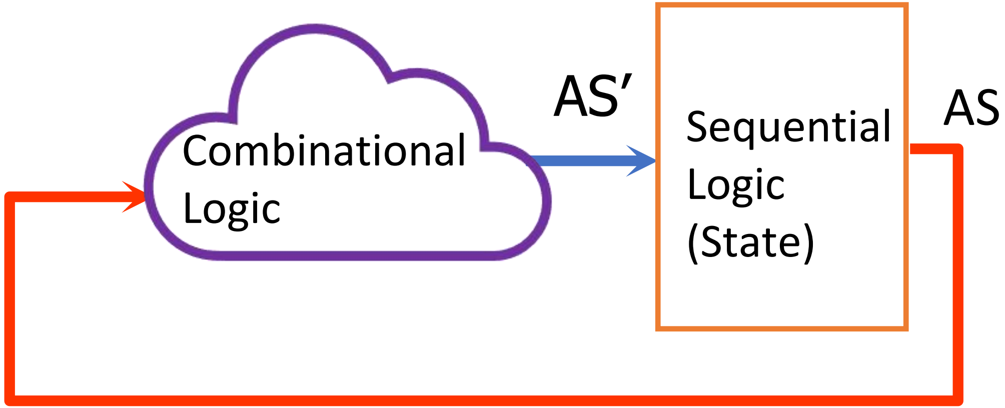
at the beginning of a clock cycle (programmer visible)

Process instruction in one clock cycle (programmer invisible)

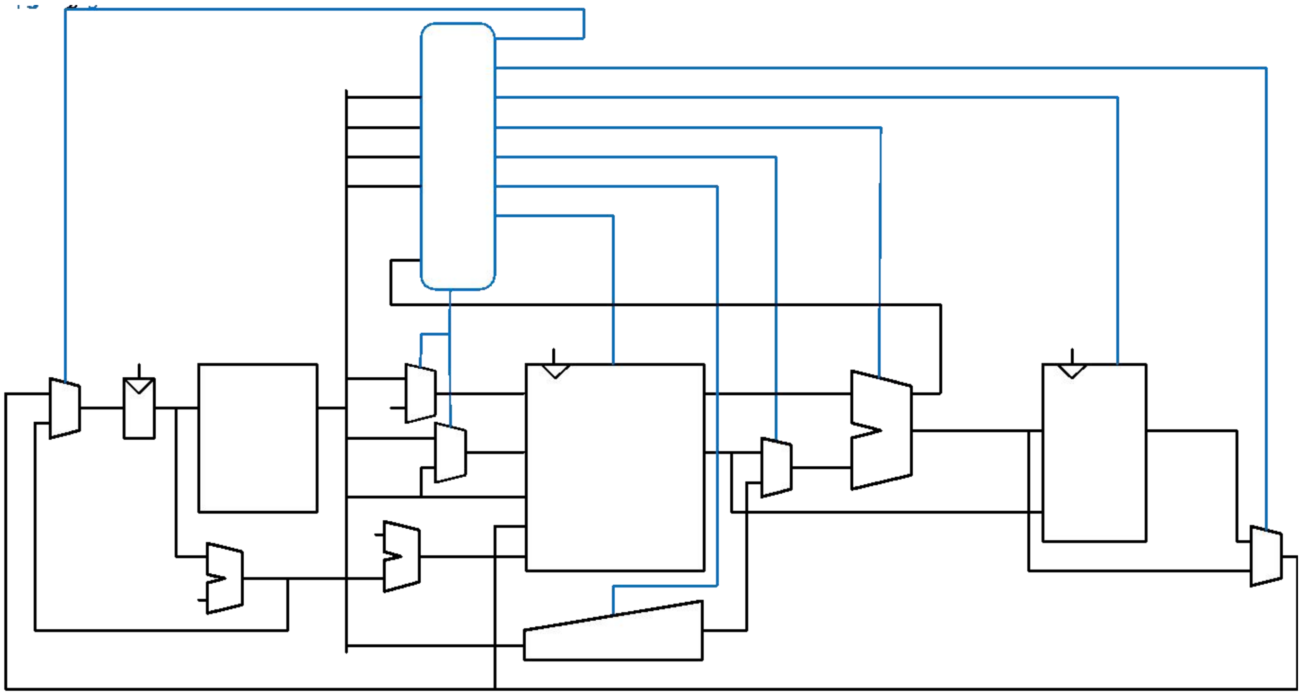
AS' = Architectural state

at the end of a clock cycle

(programmer visible)

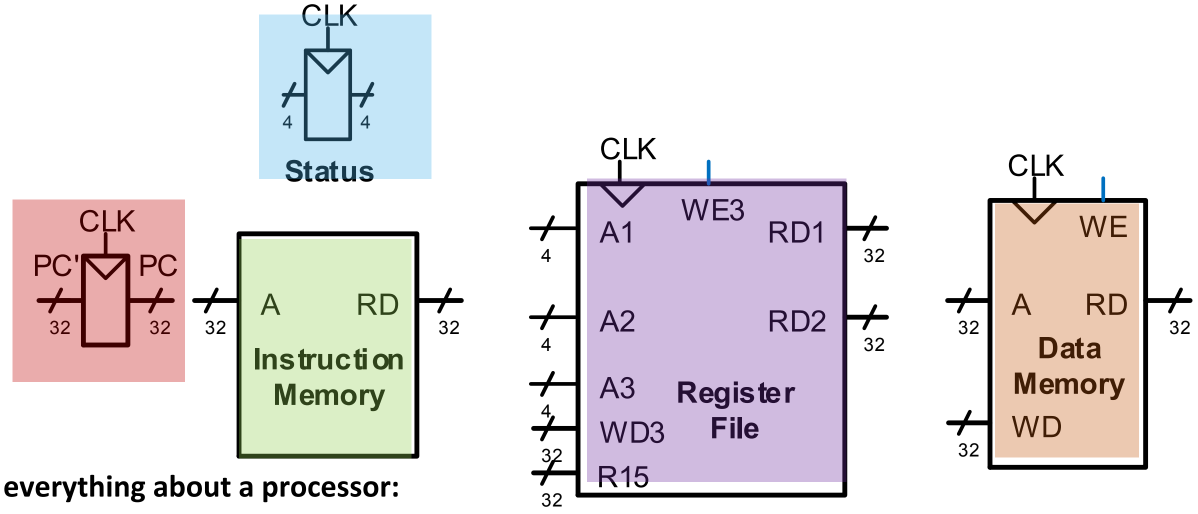


Single-Cycle ARM Processor



In this lecture, we analyze the single-cycle ARM processor.

Architectural State Elements



Determines everything about a processor:

- Program counter: 32-bit register
- Status register: 4-bit register. Also known as CPSR for flags: interrupt, overflow, carry, zero, negative, etc.
- Instruction memory: Takes input 32-bit address A and reads the 32-bit data (i.e., instruction) from that address to the read data output RD.
- Register file: The 16 register (including PC), 32-bit register file has 2 read ports and 1 write port
- Data memory: Has a single read/write port. If the write enable, WE, is 1, it writes data WD into address A on the rising edge of the clock. If the write enable is 0, it reads address A onto RD.

Outline

Introduction

Single Cycle Processor

Datapath

Control

Conclusions

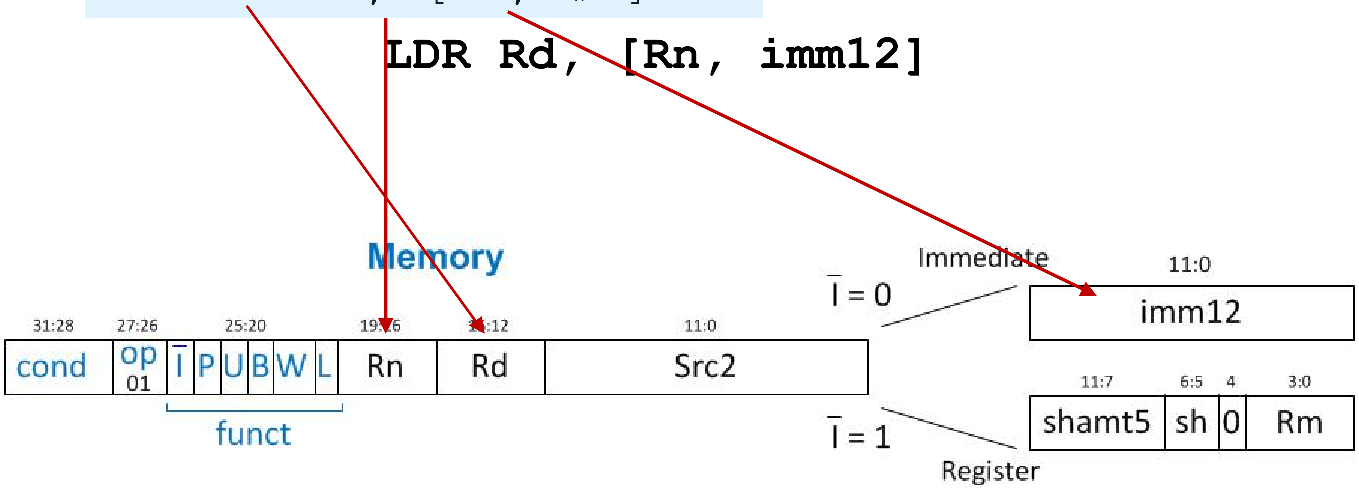
ARM Single-Cycle Processor Evaluation

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

In this lecture, we analyze the single-cycle ARM processor.

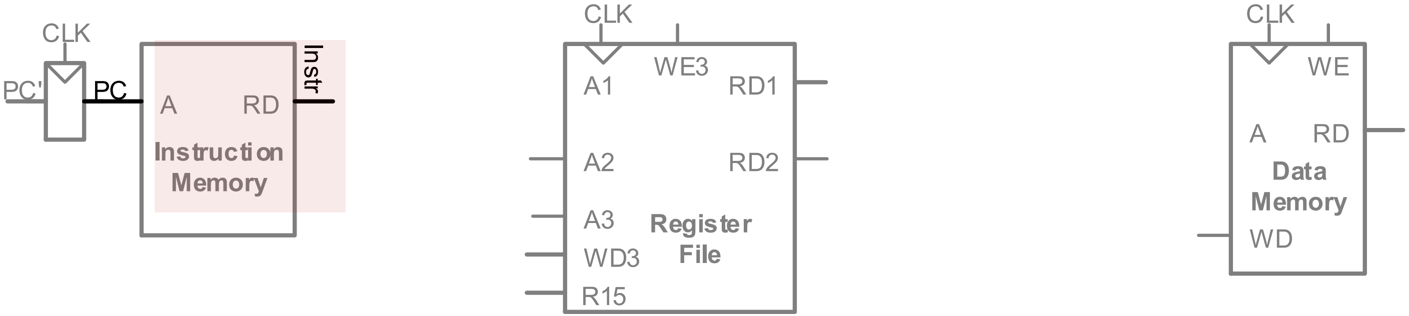
LDR Instruction

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

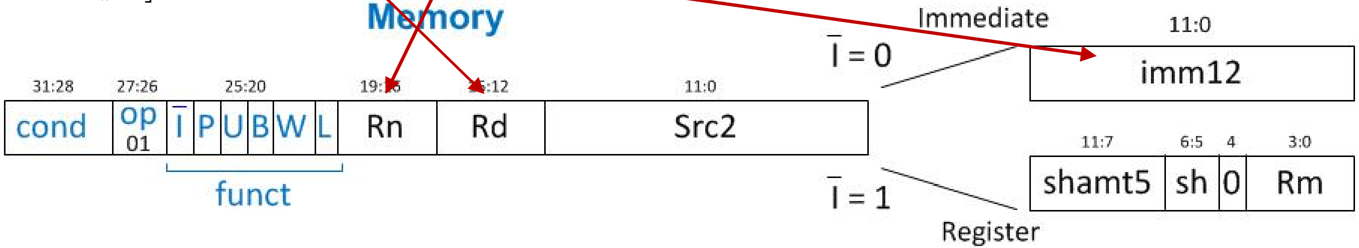


Single-Cycle Datapath: LDR fetch

STEP 1: Fetch instruction



```
LDR R1, [R2,
#5]
```



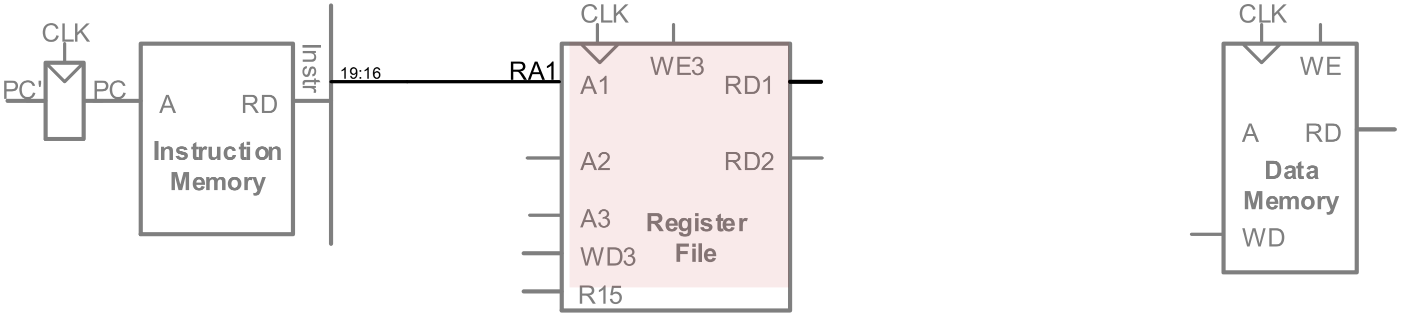
31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	$\bar{I}$	P	U	B	W	L	Rn	Rd	Src2

11:0
imm12

11:7	6:5	4	3:0
shamt5	sh	0	Rm

Single-Cycle Datapath: LDR Reg Read

STEP 2: Read source operands from RF



Memory

31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	Ī	P	U	B	W	L	Rn	Rd	Src2
		funct								

Ī = 0 Immediate

11:0
imm12

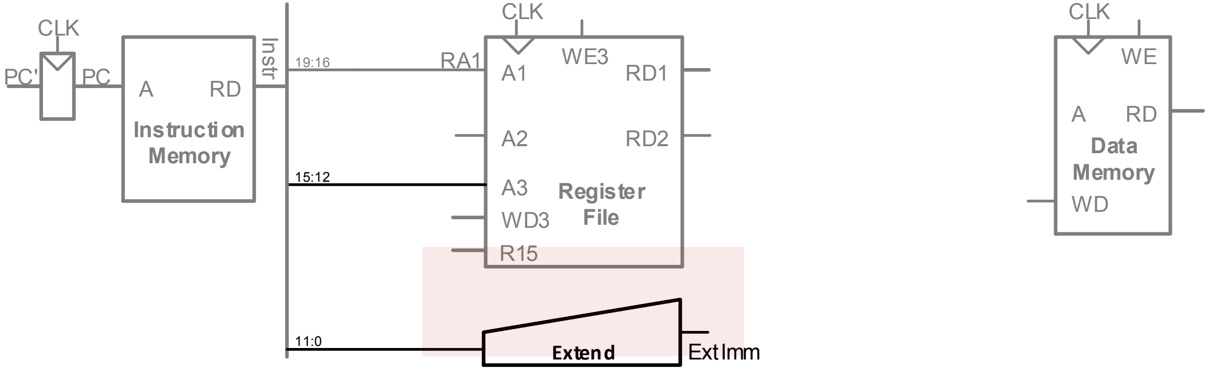
Ī = 1 Register

11:7	6:5	4	3:0
shamt5	sh	0	Rm

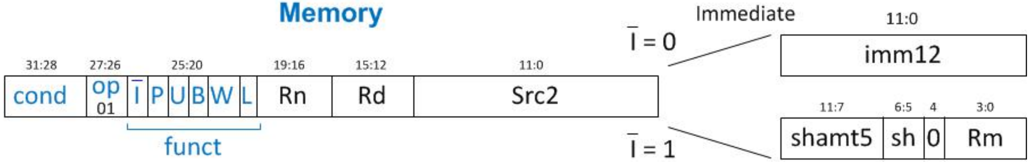
```
LDR Rd, [Rn, imm12]
```

Single-Cycle Datapath: LDR Immediate

STEP 3: Extend the immediate



Memory



31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	$\bar{I}$	P	U	B	W	L	Rn	Rd	Src2
		funct								

$\bar{I} = 0$ Immediate

11:0
imm12

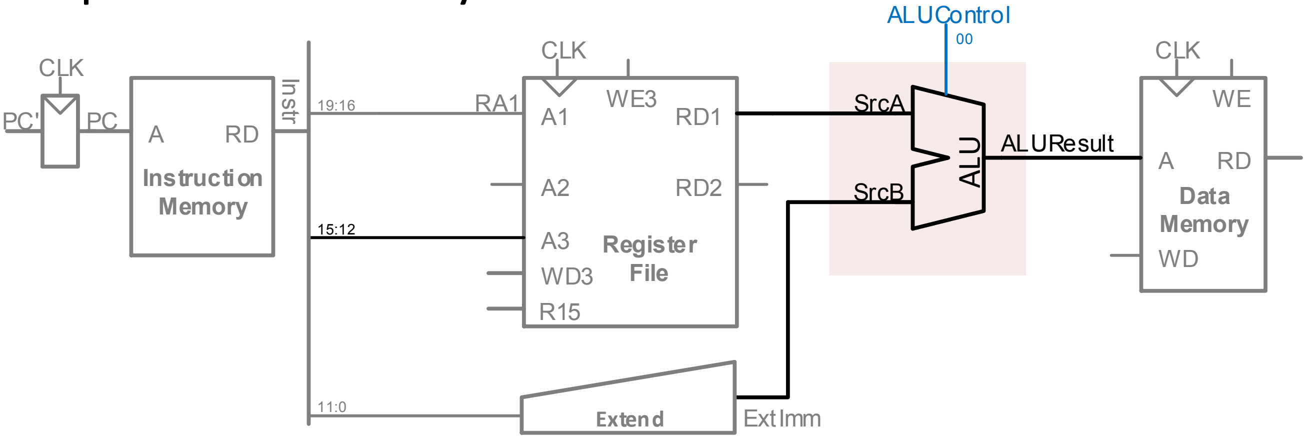
$\bar{I} = 1$ Register

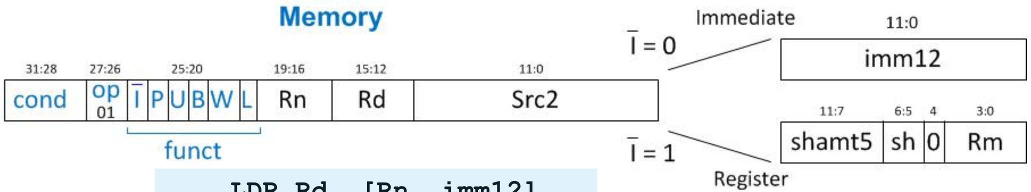
11:7	6:5	4	3:0
shamt5	sh	0	Rm

```
LDR Rd, [Rn, imm12]
```

Single-Cycle Datapath: LDR Address

STEP 4: Compute the memory address

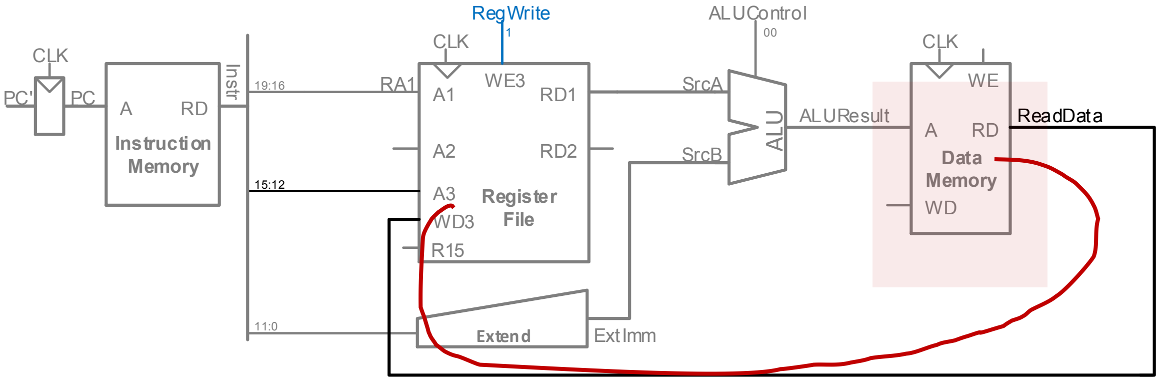


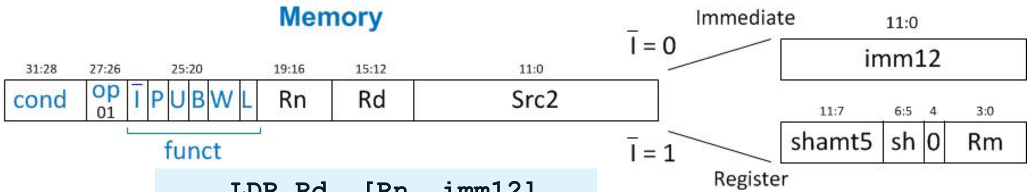


`LDR Rd, [Rn, imm12]`

Single-Cycle Datapath: LDR Mem Read

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

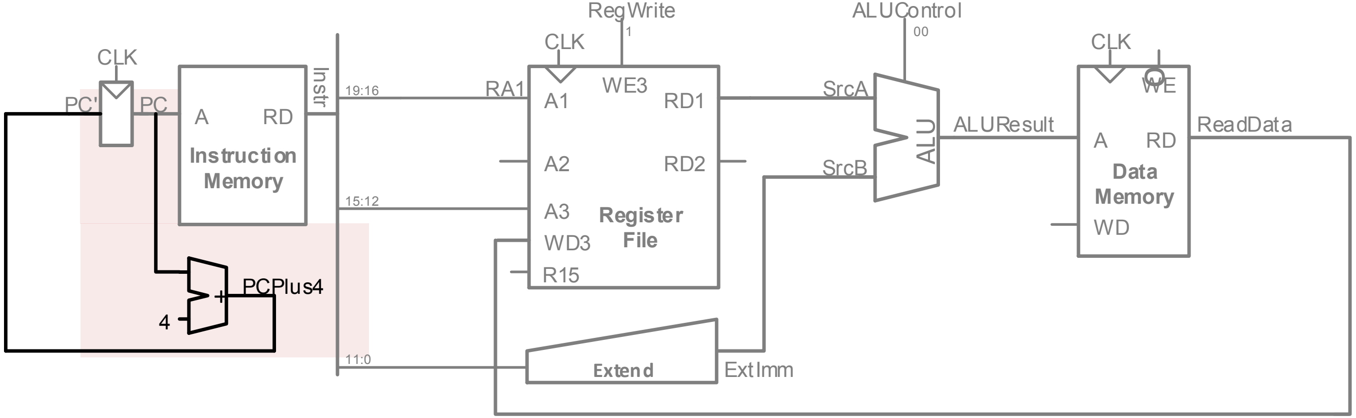




```
LDR Rd, [Rn, imm12]
```

Single-Cycle Datapath: PC Increment

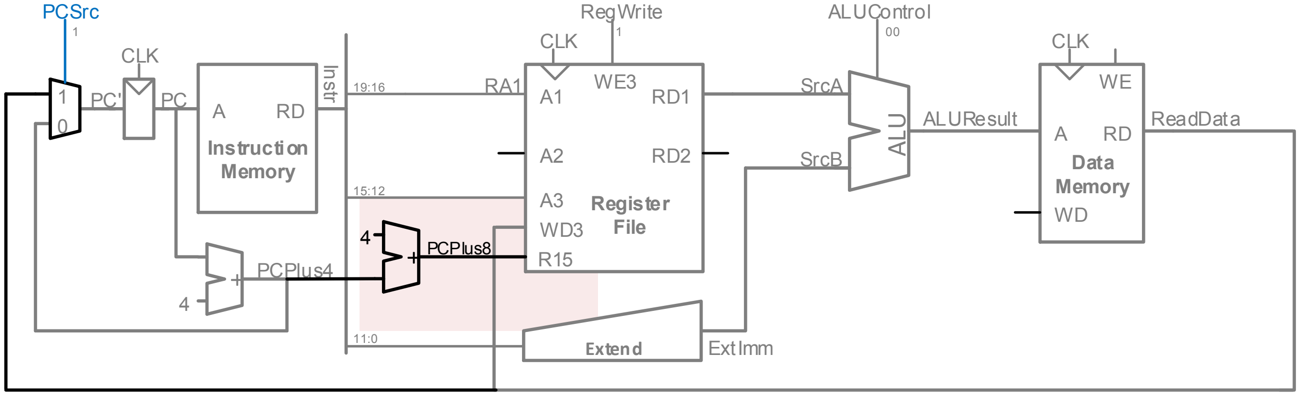
STEP 6: Determine address of next instruction



Single-Cycle Datapath: Access to PC

PC can be source/destination of instruction

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

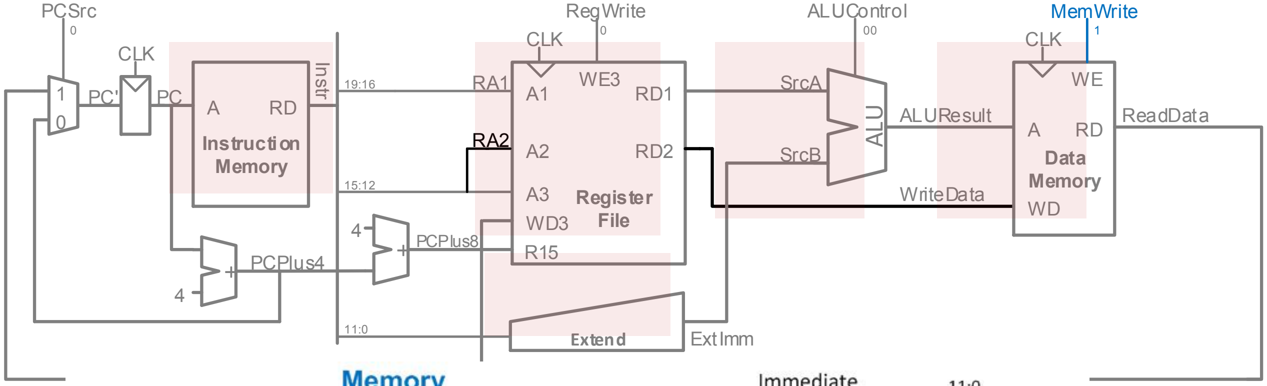


- This is because **older ARM processors always fetched two instructions ahead** of the currently executed instructions.
- The reason **ARM retains this definition is to ensure compatibility with earlier processors.**

Single-Cycle Datapath: `STR`

Expand datapath to handle `STR`:

- Write data in `Rd` to memory



Memory

31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	$\bar{I}$	P	U	B	W	L	Rn	Rd	Src2

funct

```
STR Rd, [Rn, imm12]
```

$\bar{I}$ = 0 Immediate

11:0
imm12

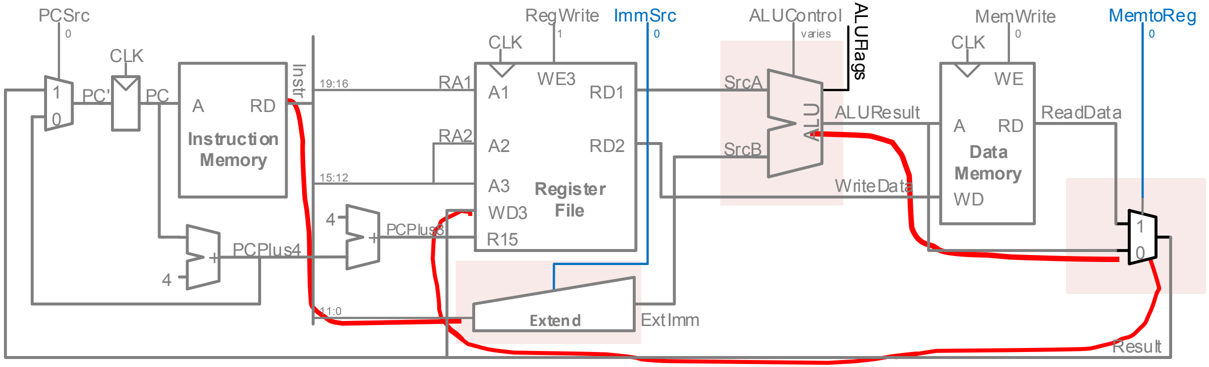
$\bar{I}$ = 1 Register

11:7	6:5	4	3:0
shamt5	sh	0	Rm

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



Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2

funct

I = 1

Immediate

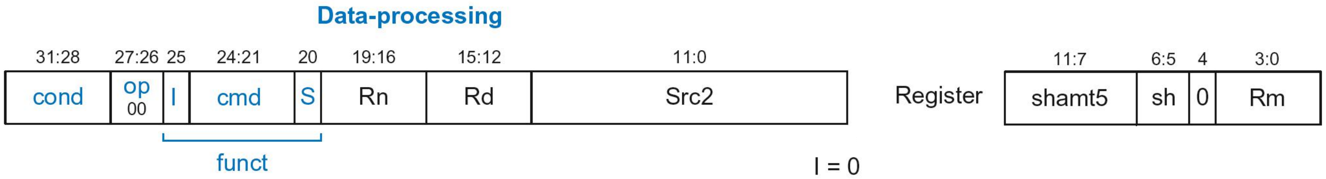
11:8	7:0
rot	imm8

```
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`



Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2

I = 0

Register

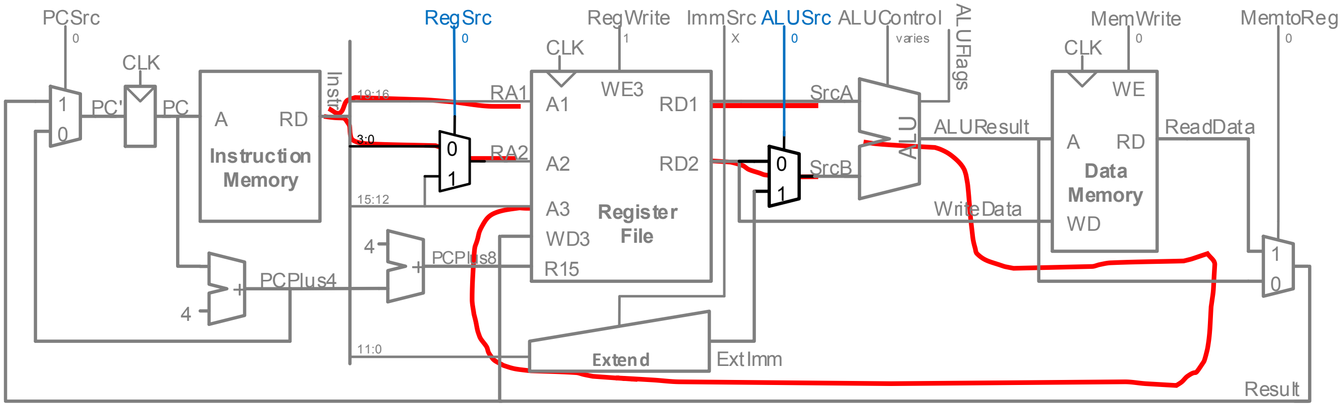
11:7	6:5	4	3:0
shamt5	sh	0	Rm

```
ADD Rd, Rn, Rm
```

Single-Cycle Datapath: Data-processing

With register Src2:

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



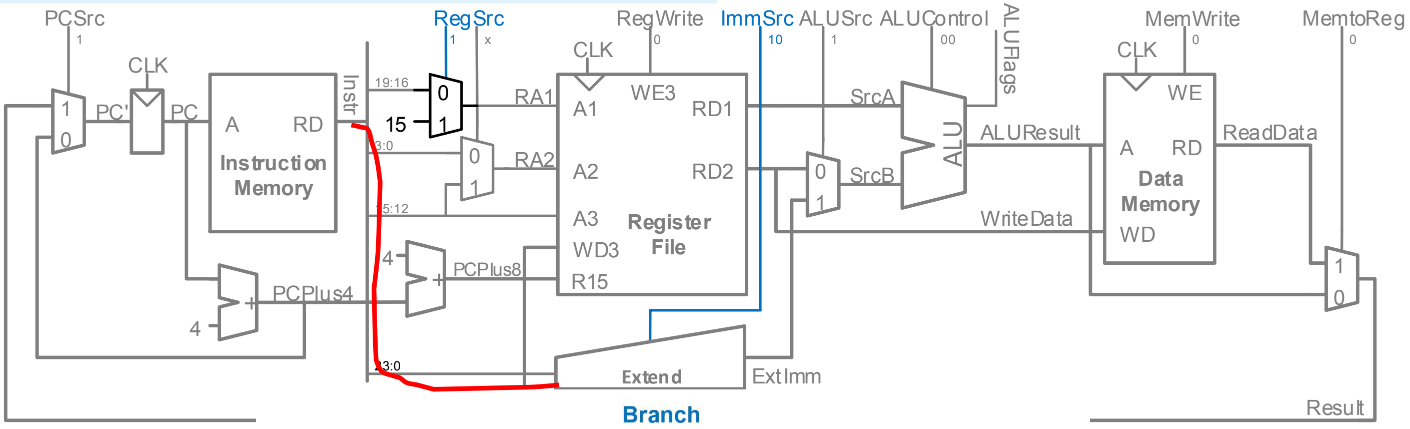
```
ADD Rd, Rn, Rm
```

Single-Cycle Datapath: B

Calculate branch target address:

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

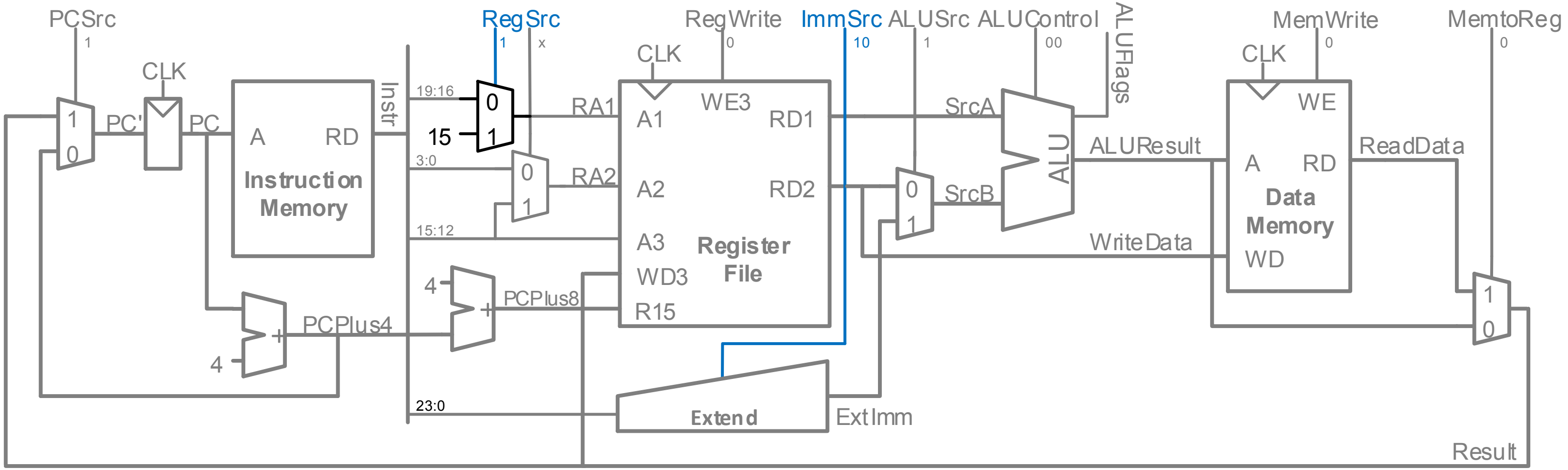
$$ExtImm = \text{Imm24} \ll 2 \text{ and sign-extended}$$



31:28	27:26	25:24	23:0
cond	op 10	1L	imm24
		funct	

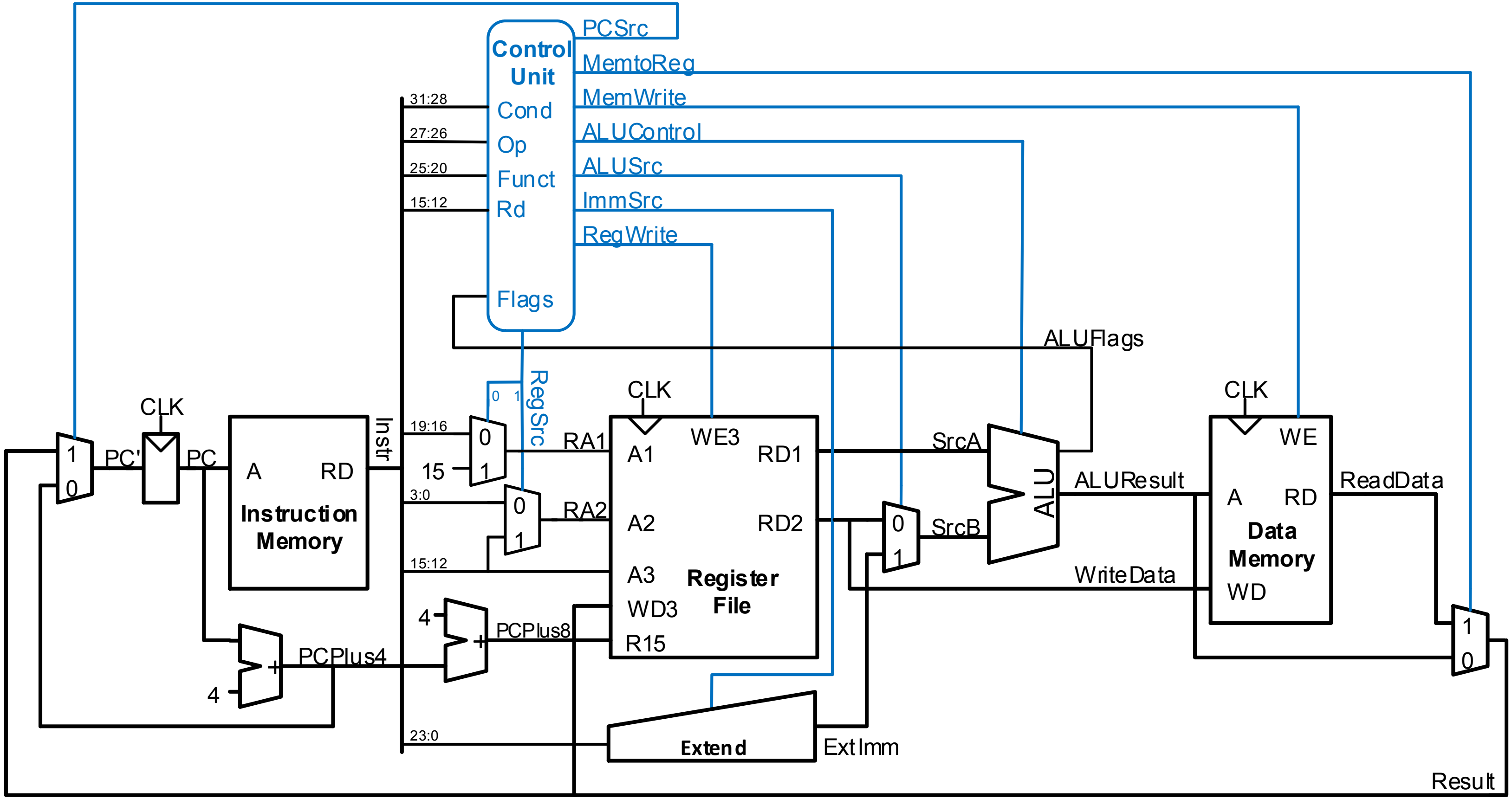
```
B Label
```

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



Outline

Introduction

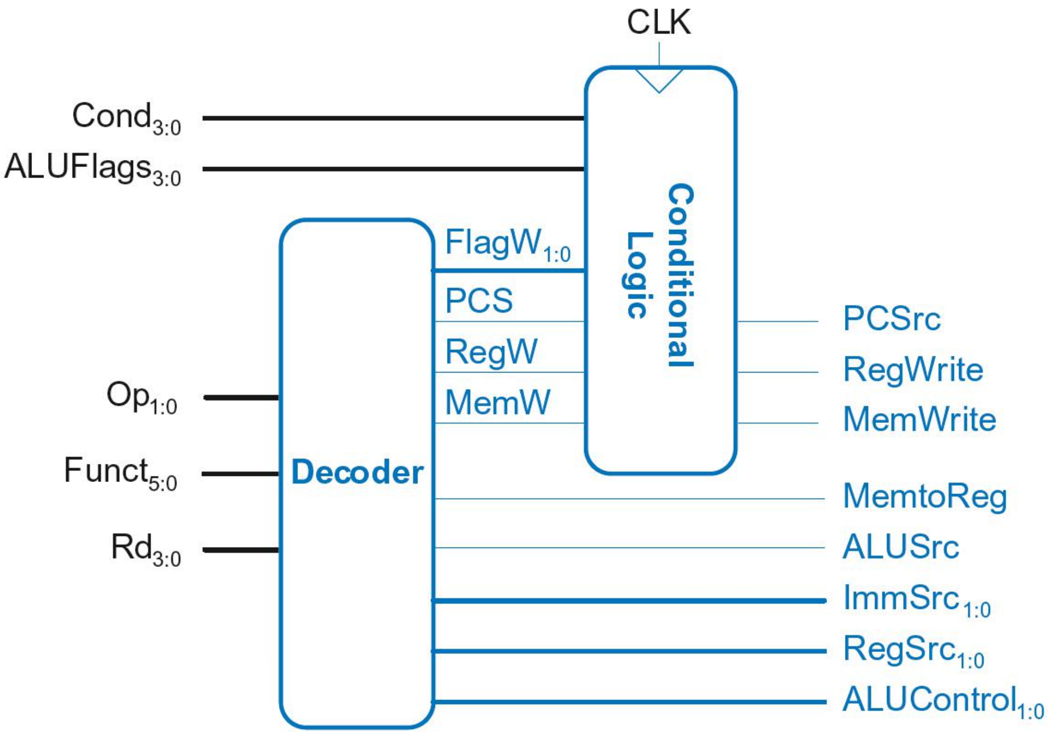
Single Cycle Processor

Datapath

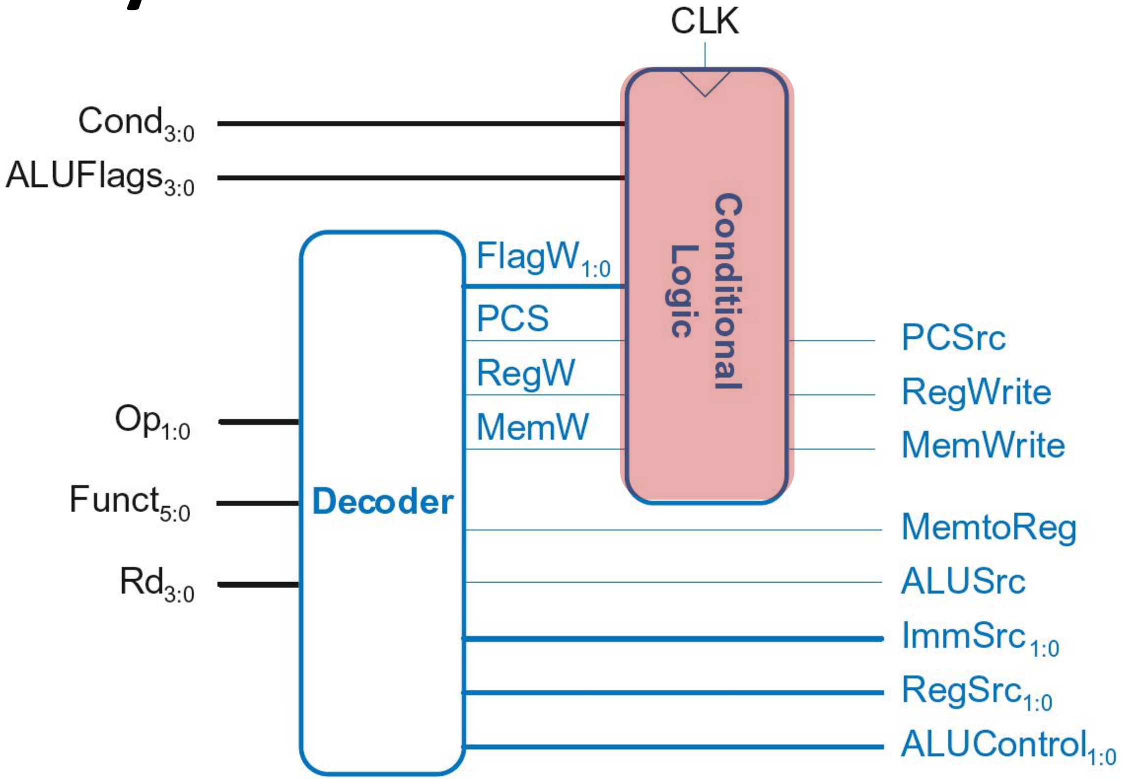
Control

Conclusions

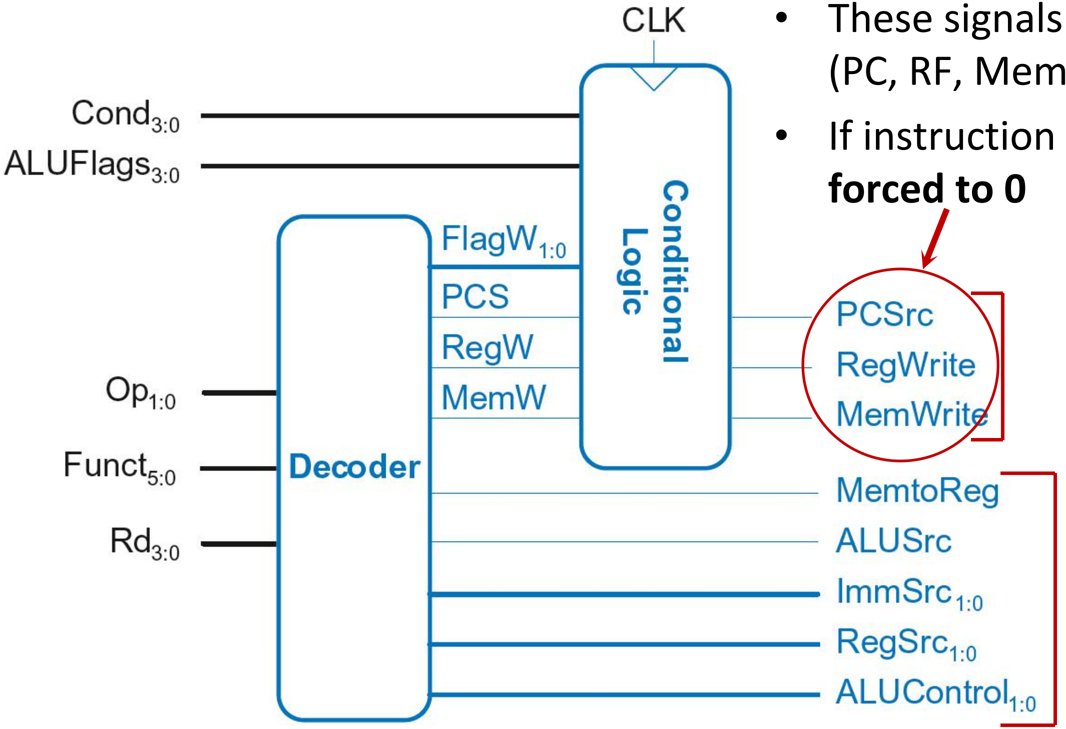
Single-Cycle Control



Single-Cycle ARM Processor



Single-Cycle Control: Signals

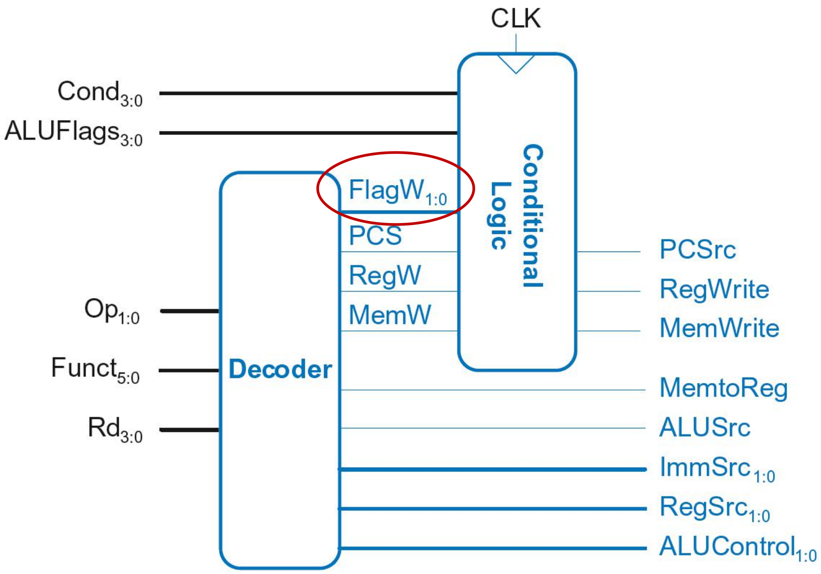


- These signals **change the state** (PC, RF, Memory)
- If instruction shouldn't execute, **forced to 0**

Sent through Conditional Logic first, then to datapath

Sent directly to datapath

Single-Cycle Control: Signals

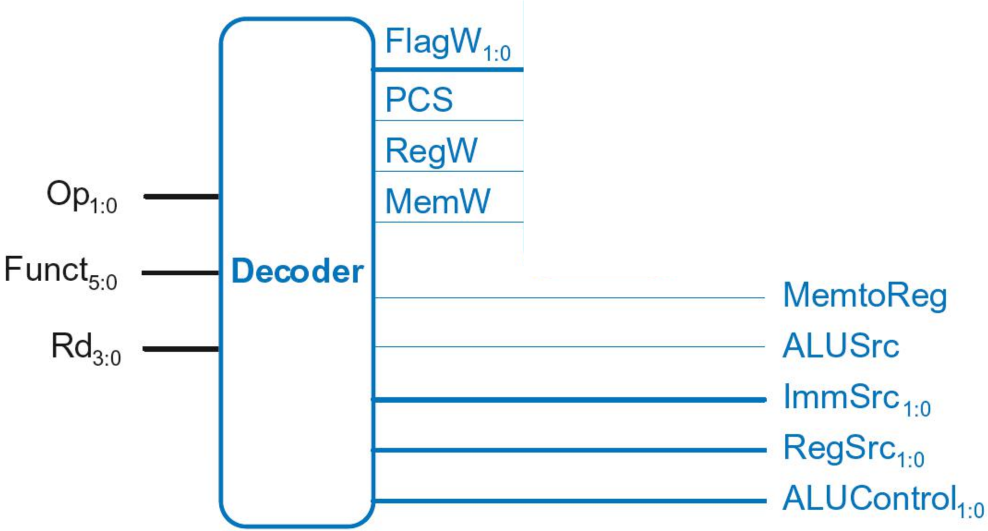


- ***$FlagW_{1:0}$***: Flag Write signal, asserted when *ALUFlags* should be saved (i.e., on instruction with S=1)
- ADD, SUB update all flags (**NZCV**)
- AND, ORR only update **NZ** flags
- So, two bits needed:

  $FlagW_1$ = 1: *NZ* saved (*$ALUFlags_{3:2}$* saved)

  $FlagW_0$ = 1: *CV* saved (*$ALUFlags_{1:0}$* saved)

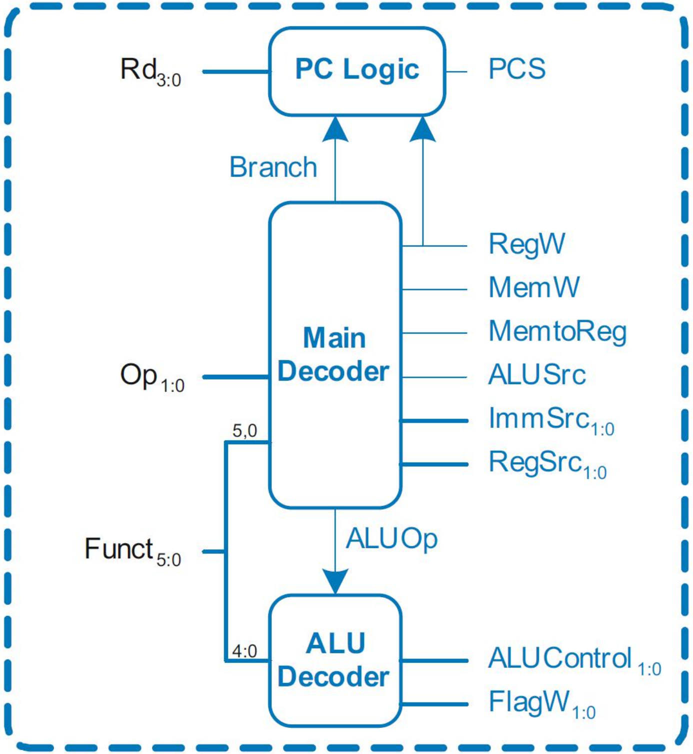
Single-Cycle Control: Decoder



Single-Cycle Control: Decoder

Submodules:

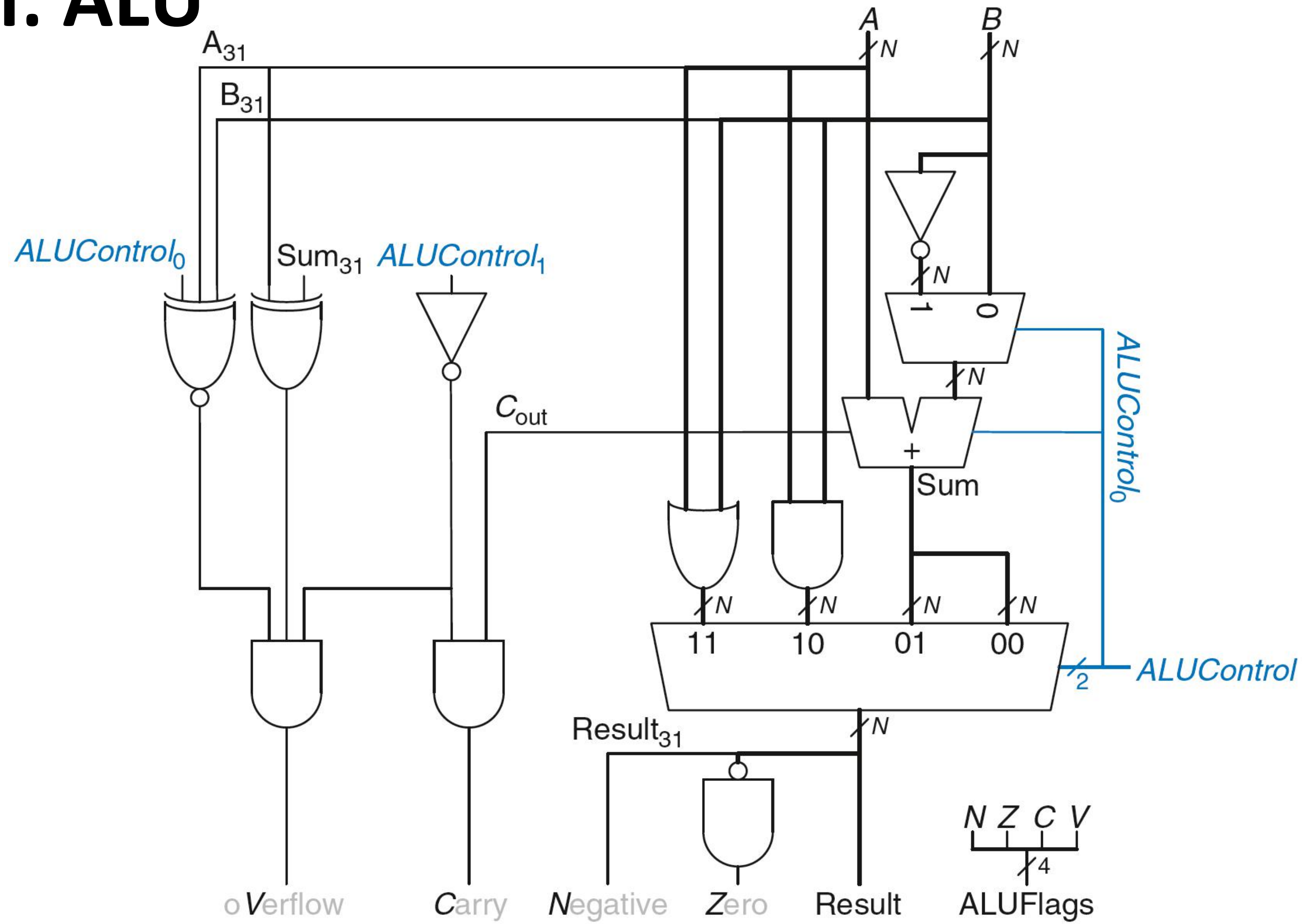
- Main Decoder
- ALU Decoder
- PC Logic



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

Recall: ALU


A
N
B
N
A_{31}
B_{31}
$ALUControl_0$
Sum_{31}
$ALUControl_1$
C_{out}
N
1
0
N
ALUControl₀
+
Sum
N
N
N
N
11
10
01
00
2
ALUControl
$Result_{31}$
N
N Z C V
4
oVerflow
Carry
Negative
Zero
Result
ALUFlags

Control Unit: ALU Decoder

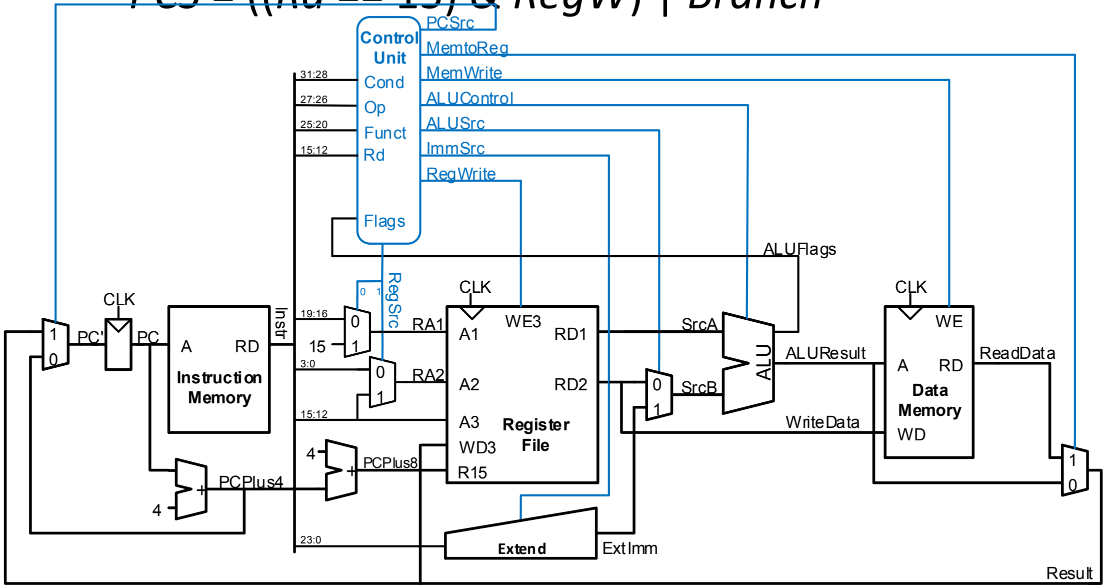
ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00	00
1	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10

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

Single-Cycle Control: PC Logic

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

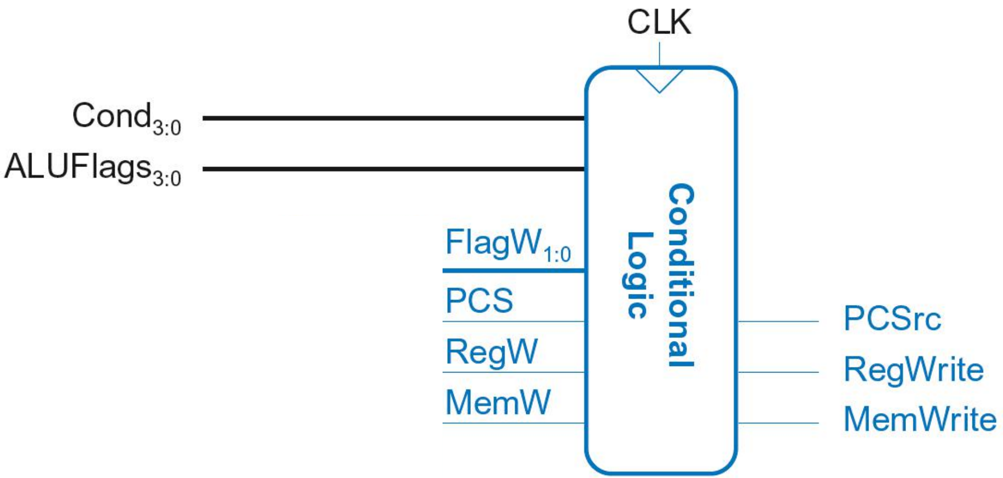
$$PCS = ((Rd == 15)\ \&\ RegW)\ |\ Branch$$



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

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

Conditional Logic

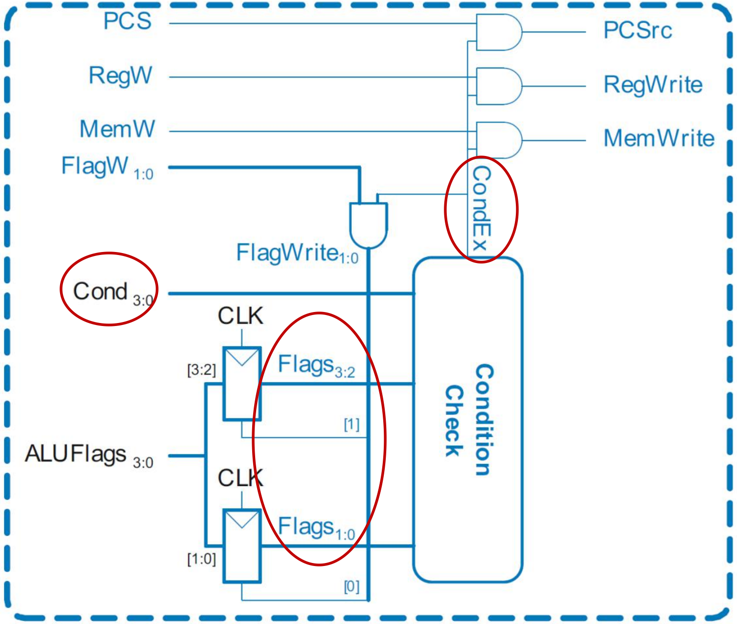


Function:

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

Conditional Logic: Conditional Execution

Flags$_{3:0}$ is the **status register**



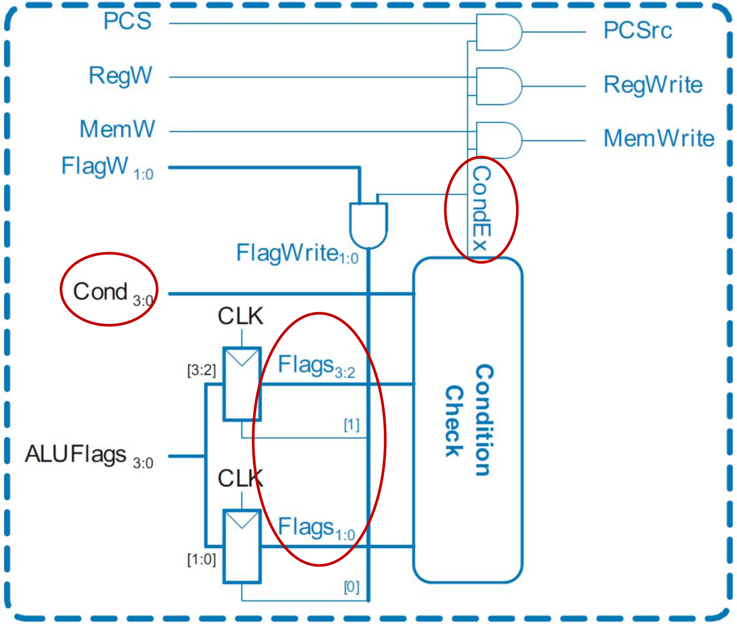
Depending on condition mnemonic (***Cond$_{3:0}$***) and condition flags (***Flags$_{3:0}$***) the instruction is executed (***CondEx*** = 1)

Recall: Condition Mnemonics

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

Example: Conditional Execution

Flags$_{3:0}$ = NZCV

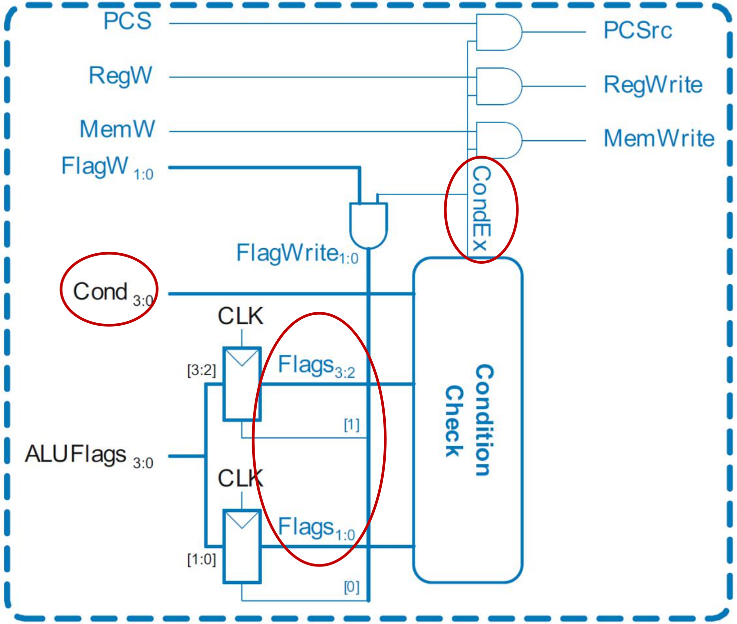


Example: `AND R1, R2, R3`

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

Example: Conditional Execution

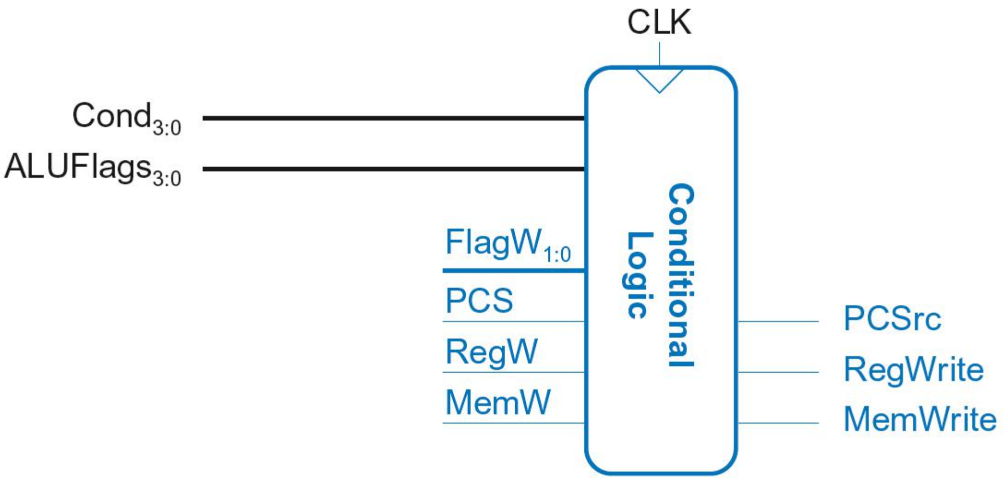
Flags$_{3:0}$ = NZCV



Example: `EOREQ R5, R6, R7`

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

Conditional Logic

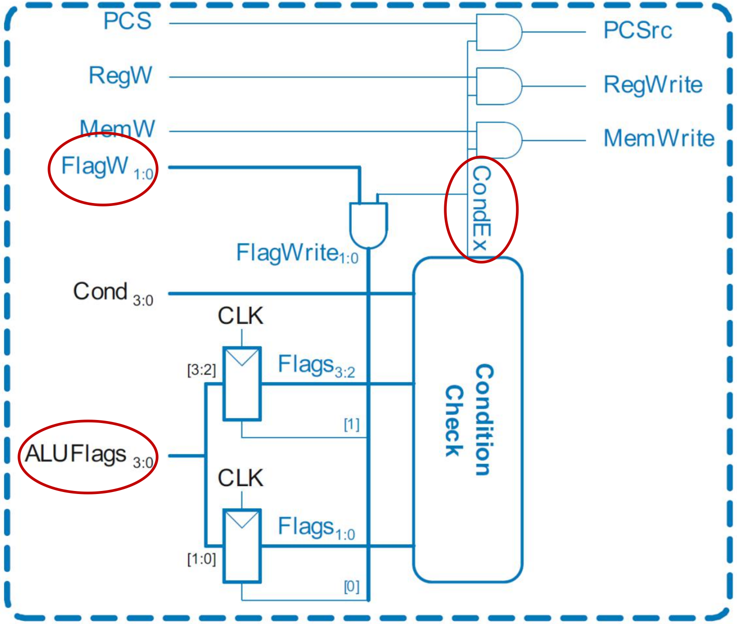


Function:

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

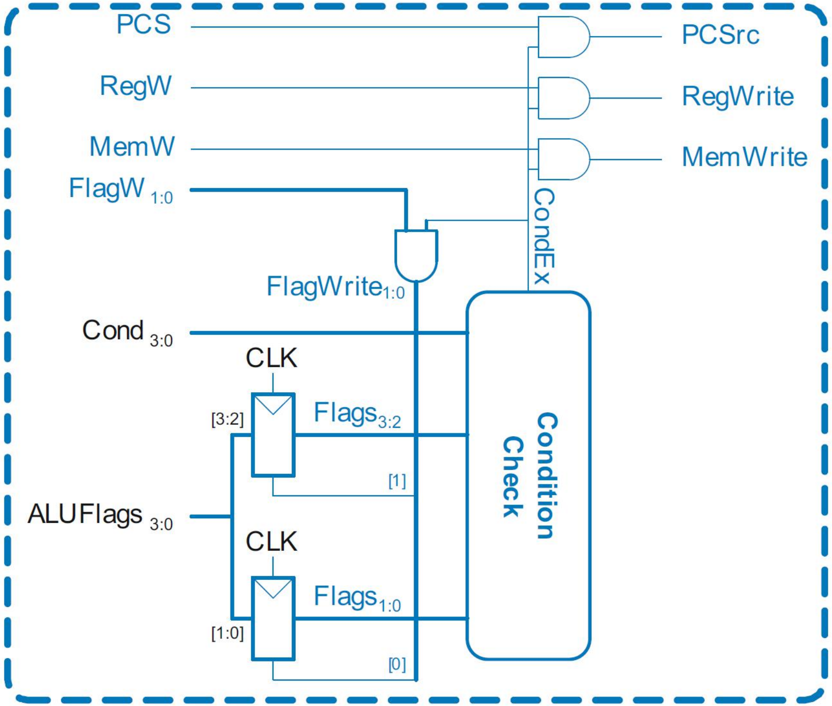
$\text{Flags}_{3:0}$ = NZCV



$\text{Flags}_{3:0}$ **updated** (with $\text{ALUFlags}_{3:0}$) 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: **$\text{FlagW}_{1:0}$**

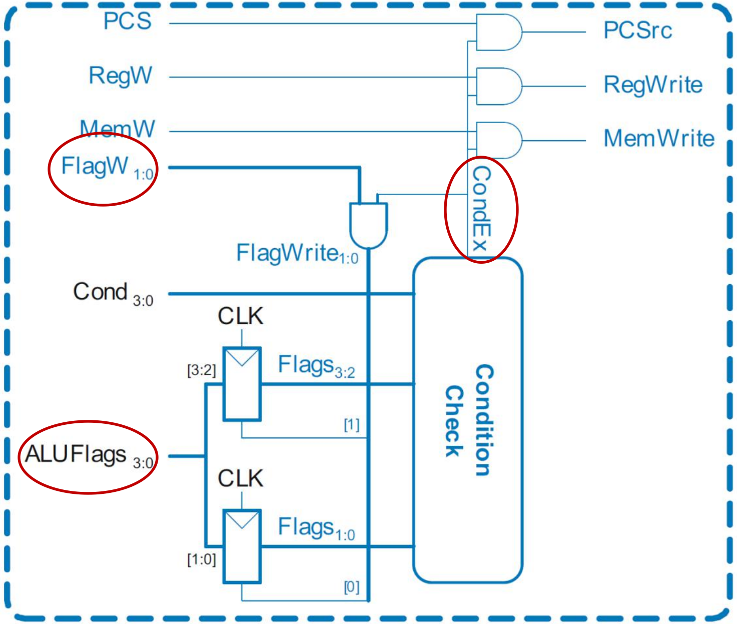
Recall: ALU Decoder

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

Example: Update (Set) Flags

All Flags updated



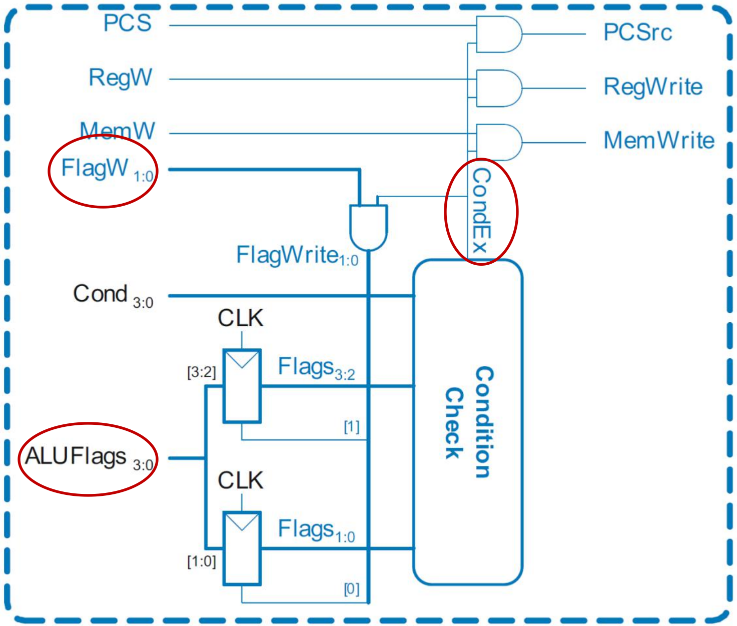
Example: `SUBS R5, R6, R7`

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

Example: Update (Set) Flags

Flags$_{3:0}$ = NZCV

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

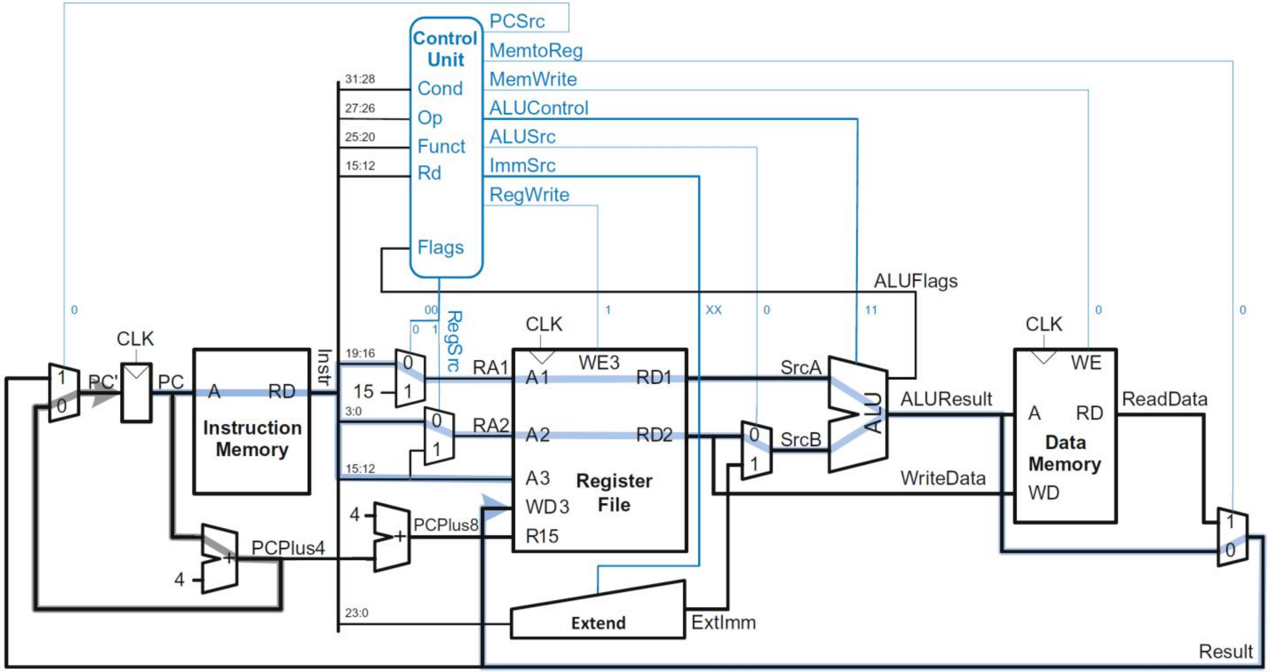


Example: `ANDS R7, R1, R3`

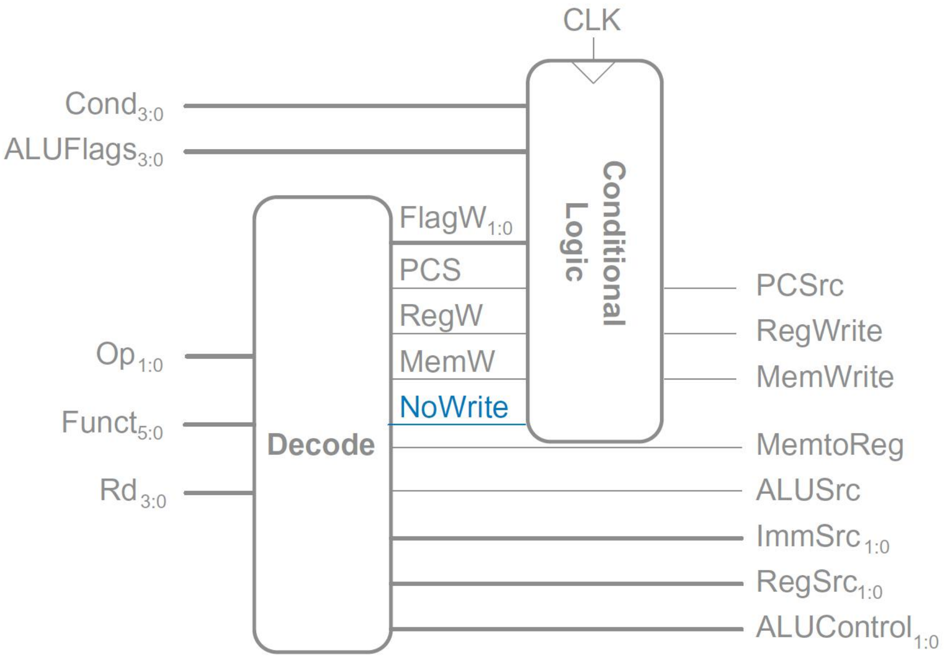
FlagW$_{1:0}$ = 10 AND **CondEx** = 1 (unconditional) => **FlagWrite$_{1:0}$** = 10

Example: ORR

Op	Funct5	Funct0	Type	B	Me	Me	ALUSrc	ImmSrc	Reg	RegSrc	ALUOp

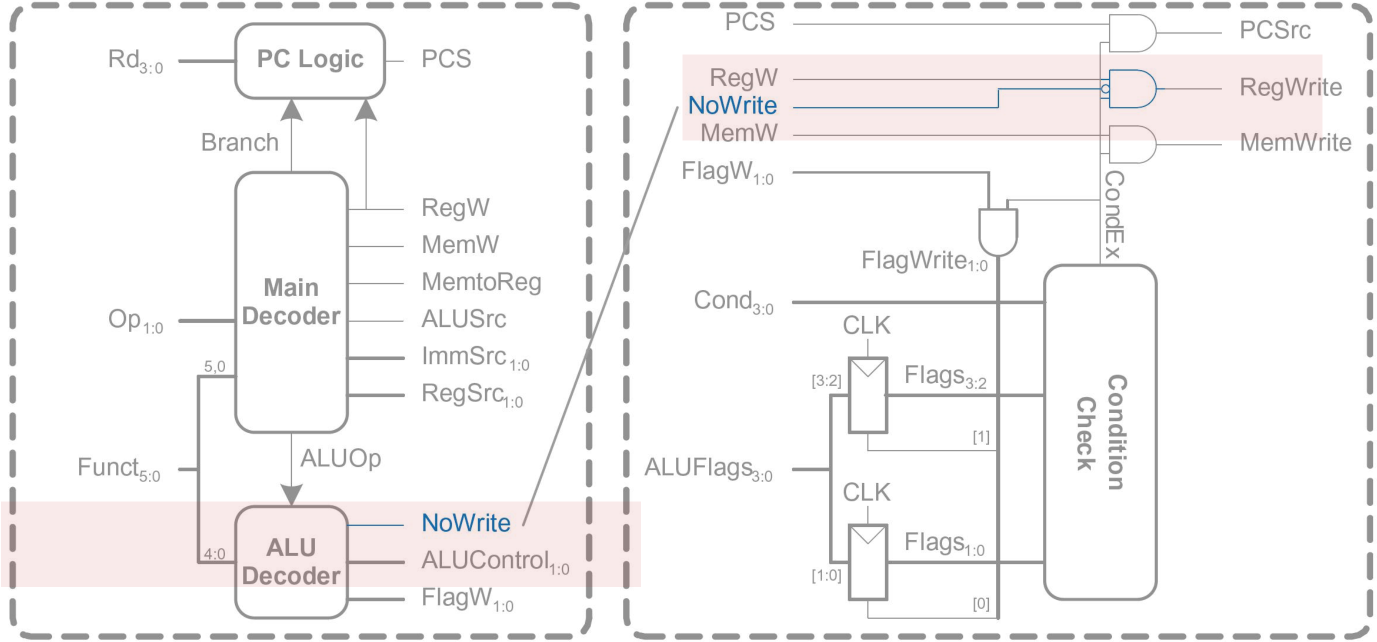


Extended Functionality: CMP



No change to datapath

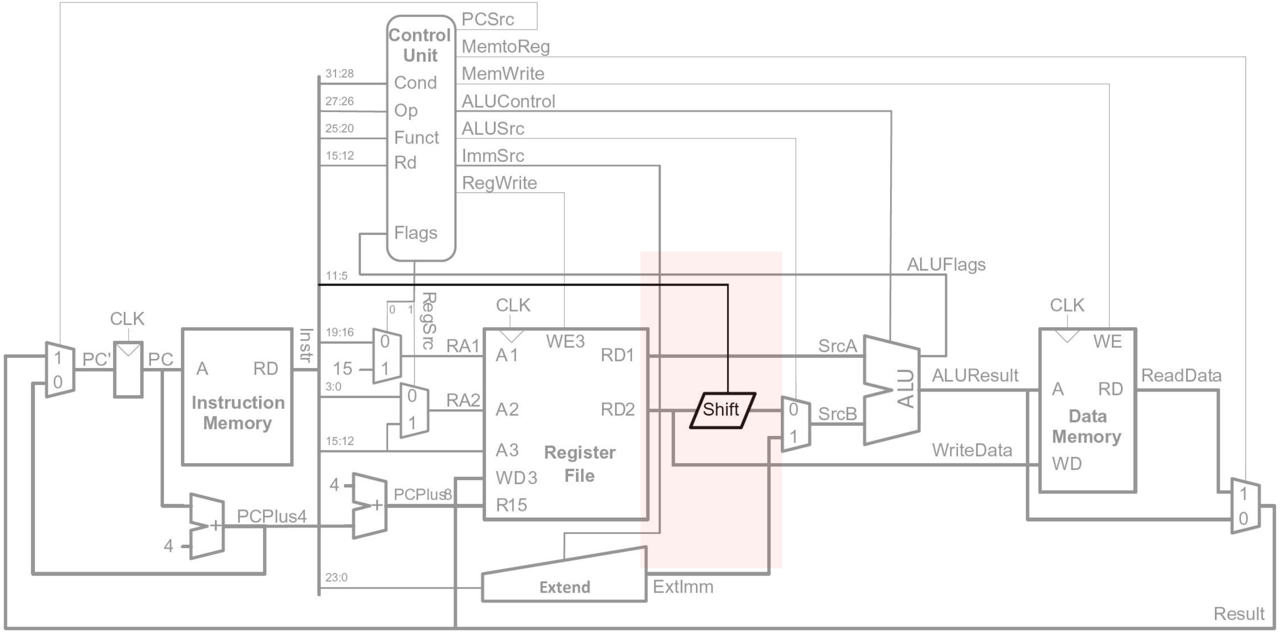
Extended Functionality: CMP



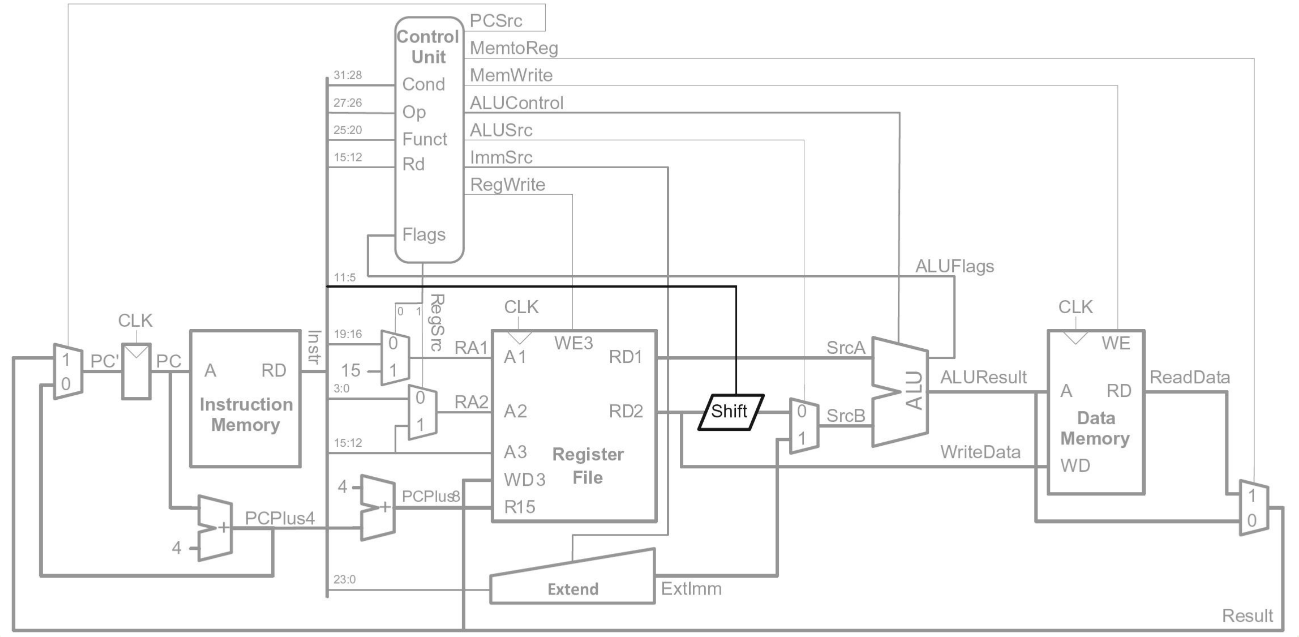
Extended Functionality: CMP

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



Extended Functionality: Shifted Register



No change to controller

Outline

Introduction

Single Cycle Processor

Datapath

Control

Conclusions

Conclusions

- **We detailed the processor microarchitecture.**
- **We analyzed the instruction operation and interaction with the processor datapath and control units.**
- We conclude that a **processor can have different implementations of the ISA based on logic blocks.**

Microarchitecture

Computer Architecture

CS3501 – 2024II

PROF.: JGONZALEZ@UTEC.EDU.PE