

Microarchitecture

Computer Architecture



CS3501 - 2023I
PROF.: JGONZALEZ@UTEC.EDU.PE
SRC: HARRIS, HARRIS - DIGITAL DESIGN AND COMPUTER ARCHITECTURE



Executive Summary

2

- **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

3

Introduction

Single Cycle Processor

Datapath

Control

Conclusions

Outline

4

Introduction

Single Cycle Processor

Datapath

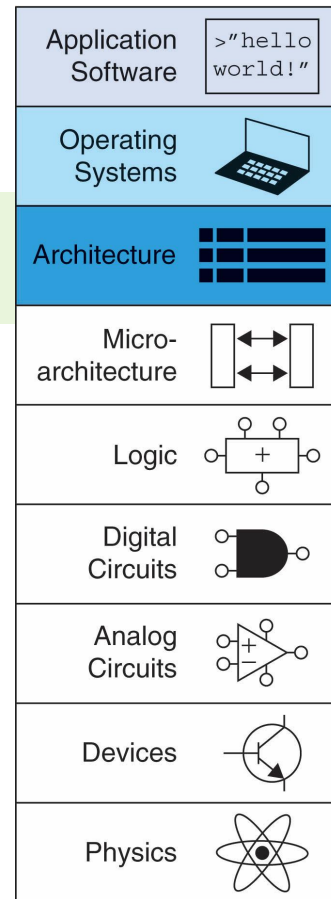
Control

Conclusions

Recall: ISA

5

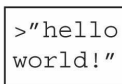


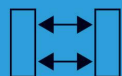
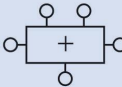

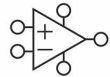


- The **ISA** is the **interface between** what the **software** commands and what the **hardware** carries out
 - Specifies **how the programmer sees the instructions** to be executed
- The ISA specifies
 - **Memory organization**
 - Address space (ARM: 2^{32} , MIPS: 2^{32})
 - Addressability (ARM: 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



Microarchitecture Definition

6

- **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

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

Computer Architecture

$$\text{Computer Architecture} = \text{ISA} + \text{Microarchitecture}$$

- 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

8

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 performance?

- Program execution time

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

- 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

10

Introduction

Single Cycle Processor

Datapath

Control

Conclusions

Single Cycle Processor

II

- 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

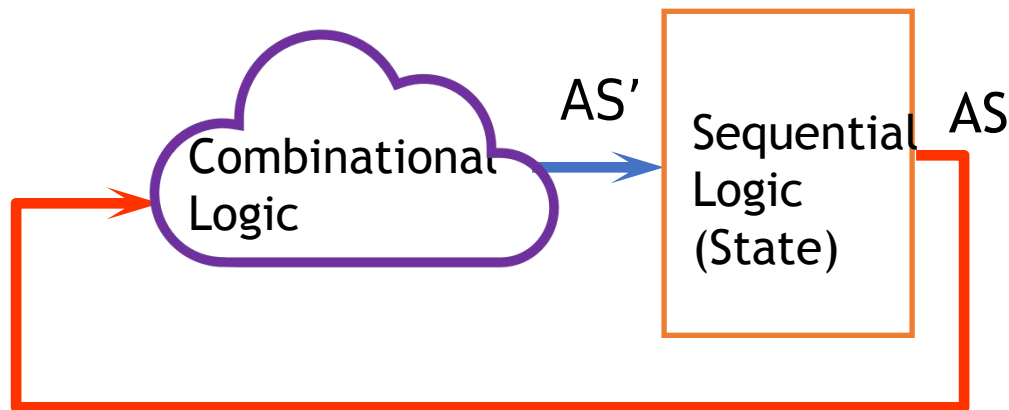
at the beginning of a clock cycle (programmer visible)



Process instruction in one clock cycle (programmer invisible)

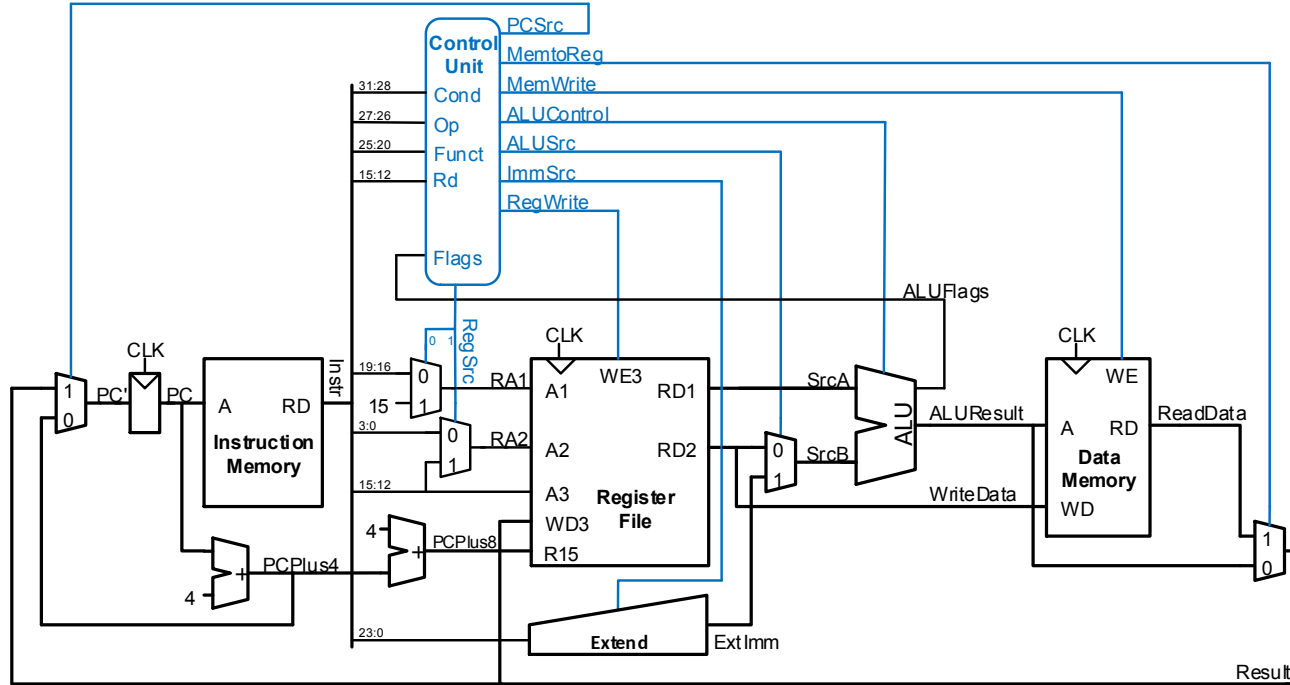


AS' = Architectural state
at the end of a clock cycle
(programmer visible)



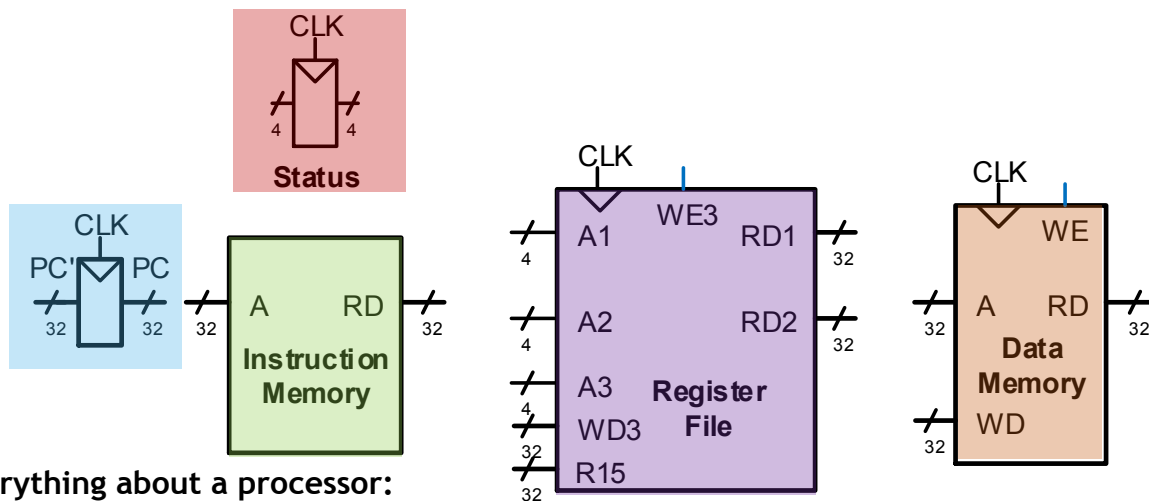
Single-Cycle ARM Processor

12



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

14

Introduction

Single Cycle Processor

Datapath

Control

Conclusions

ARM Single-Cycle Processor Evaluation

15

- 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**

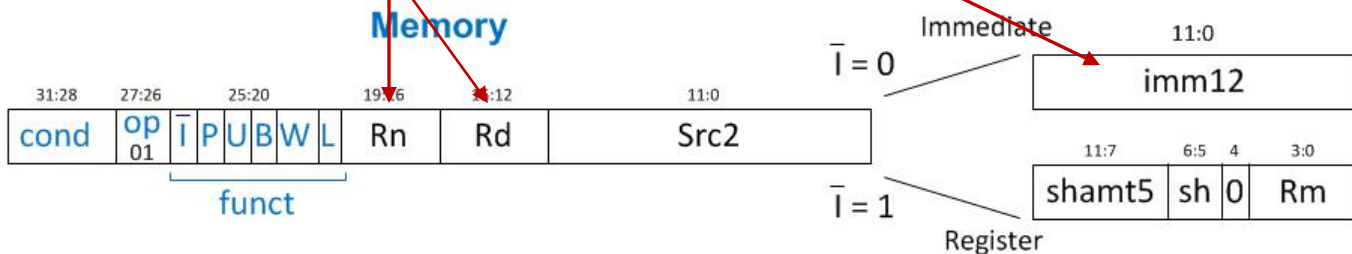
In this lecture, we analyze the single-cycle ARM processor.

LDR Instruction

16

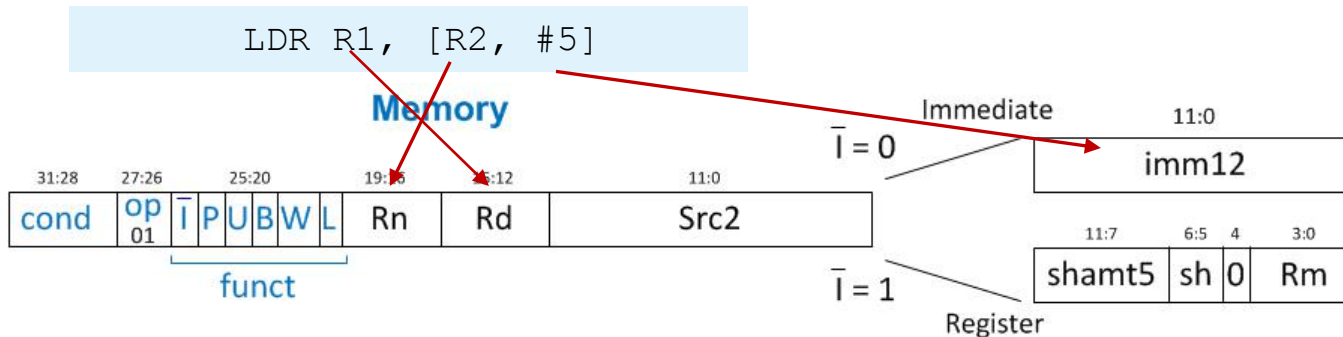
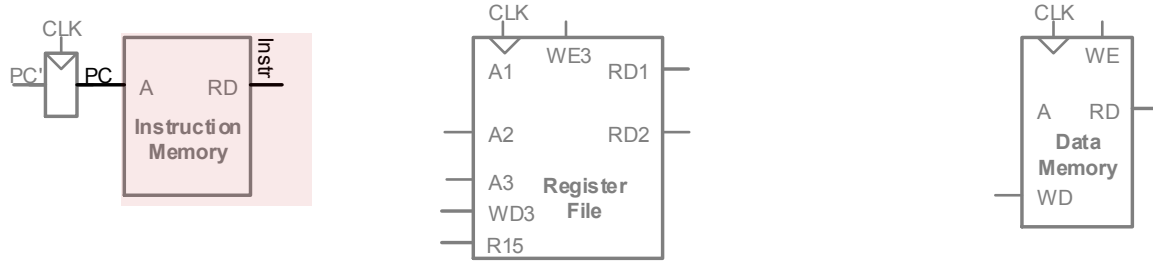
- Datapath: start with LDR instruction

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



Single-Cycle Datapath: LDR fetch

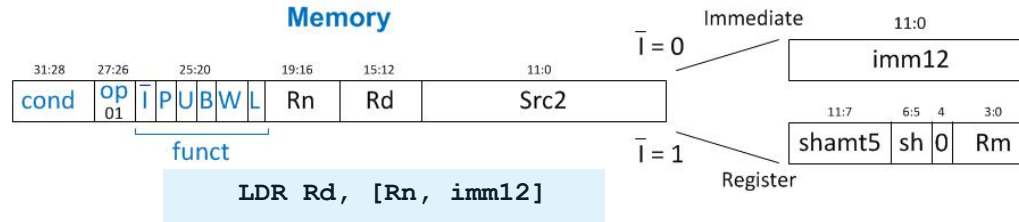
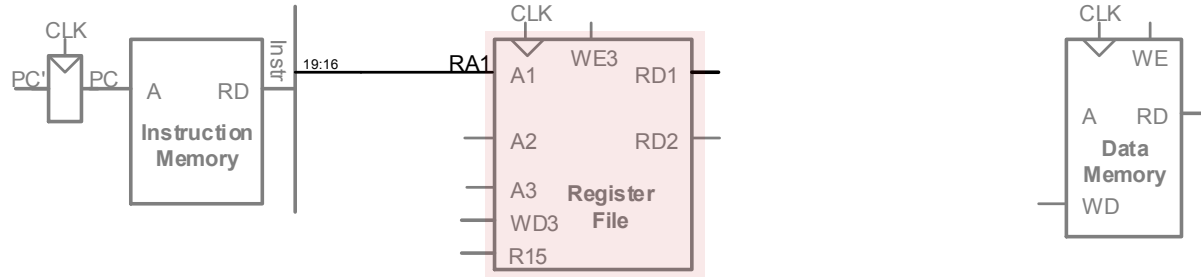
STEP 1: Fetch instruction



Single-Cycle Datapath: LDR Reg Read

18

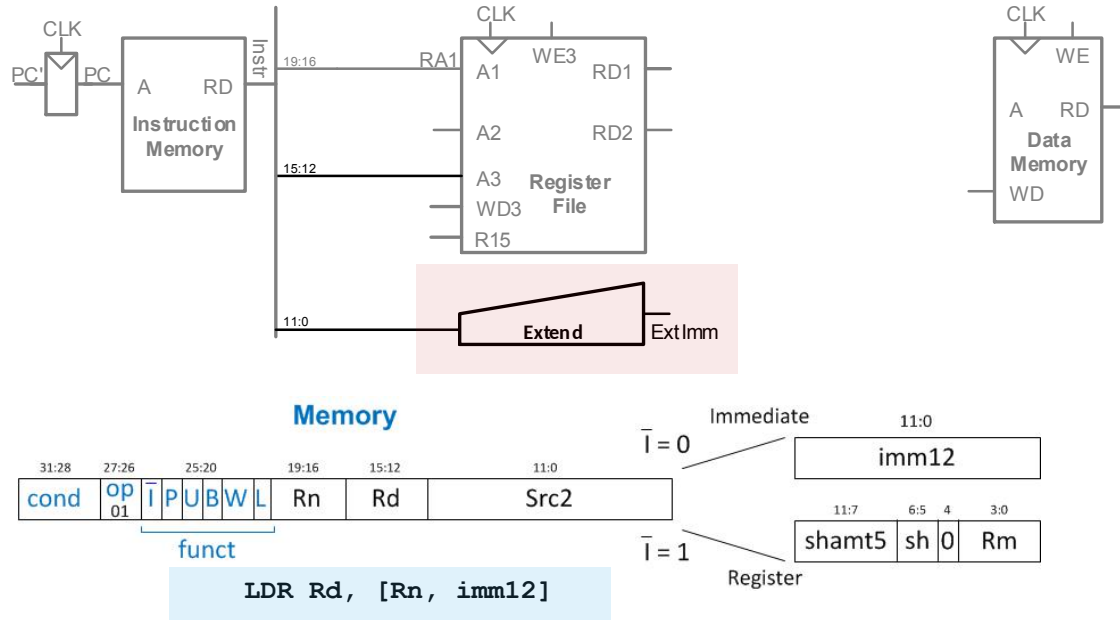
STEP 2: Read source operands from RF



Single-Cycle Datapath: LDR Immediate

19

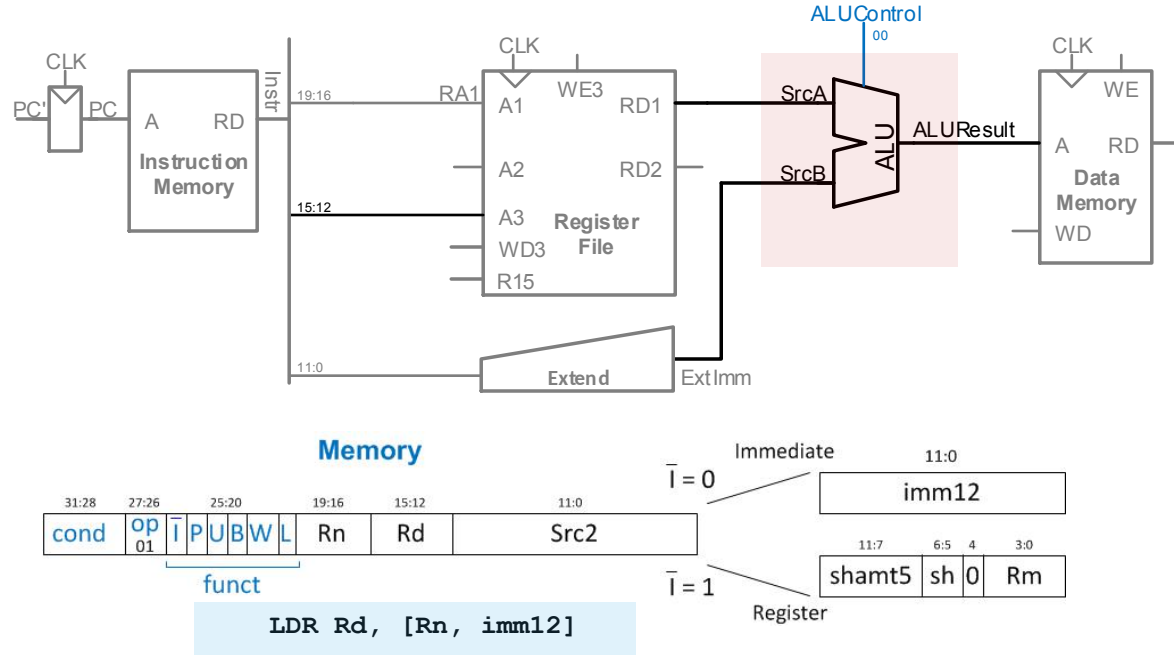
STEP 3: Extend the immediate



Single-Cycle Datapath: LDR Address

20

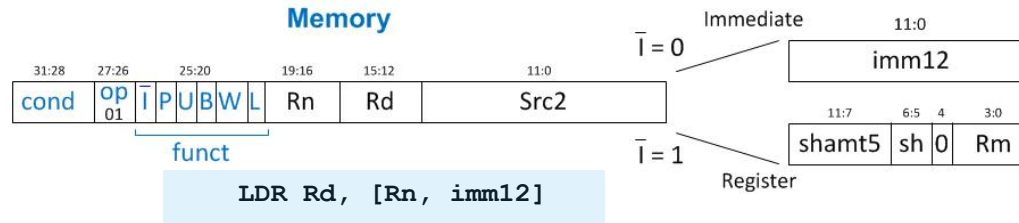
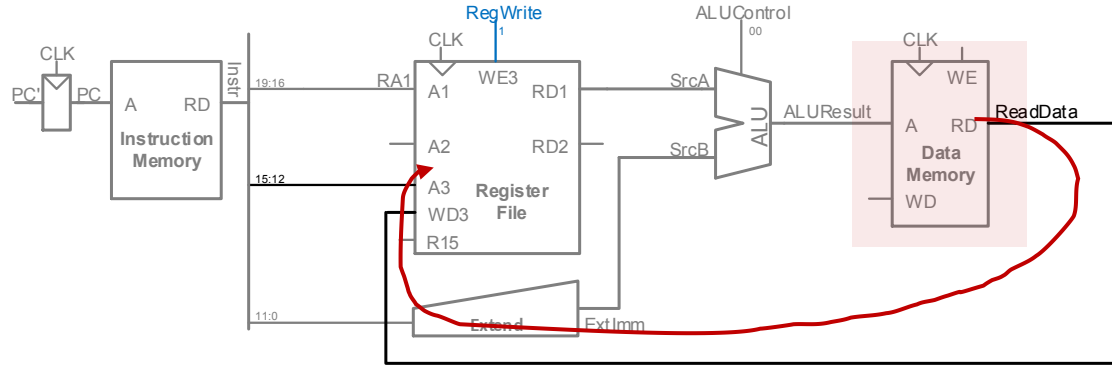
STEP 4: Compute the memory address



Single-Cycle Datapath: LDR Mem Read

21

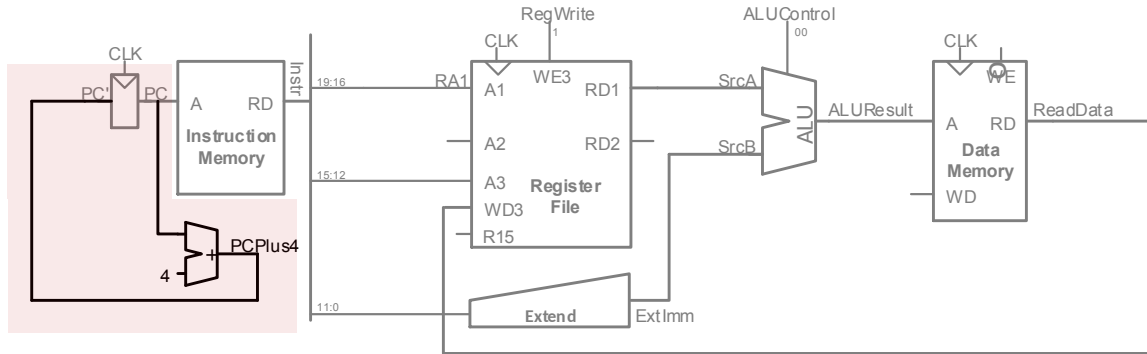
STEP 5: Read data from memory and write it back to register file



Single-Cycle Datapath: PC Increment

22

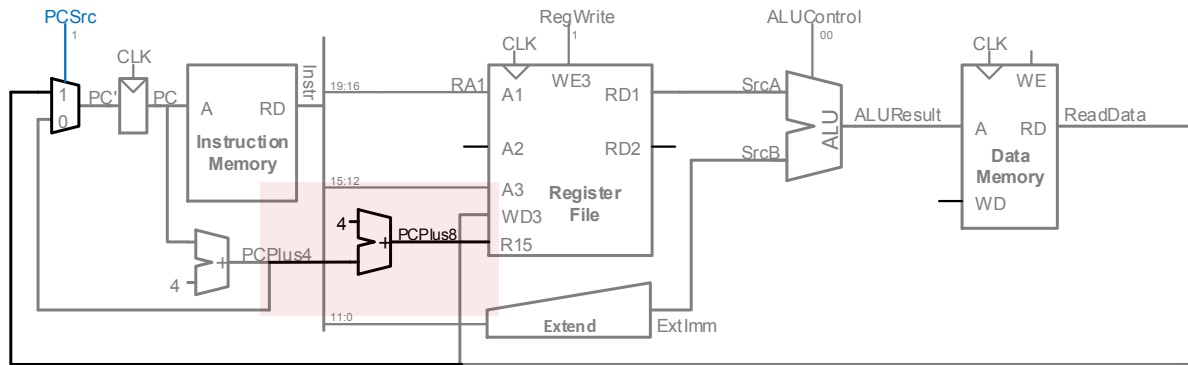
STEP 6: Determine address of next instruction



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



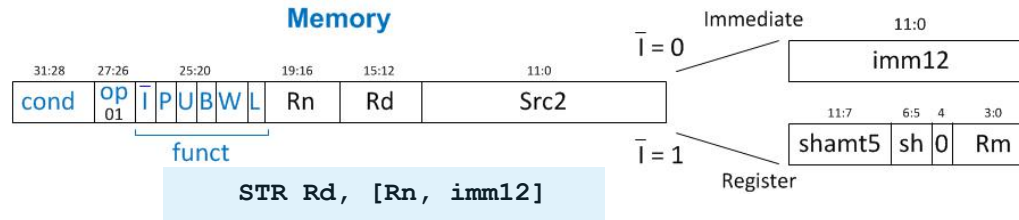
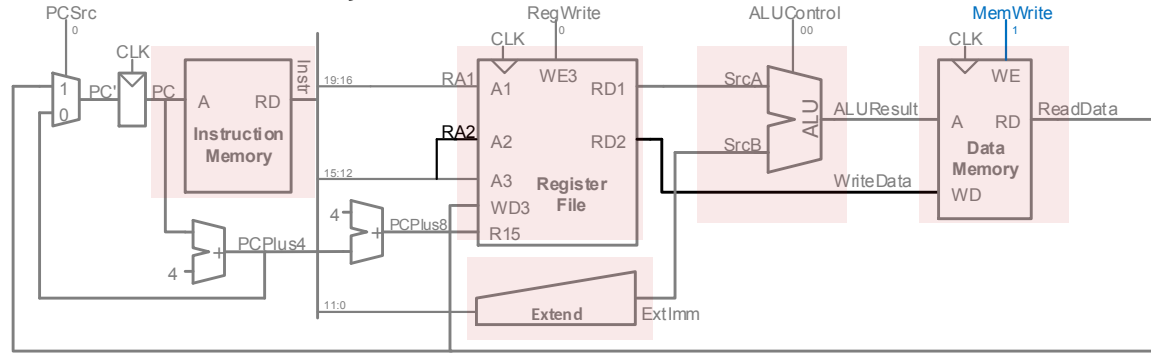
- This is because **older ARM processors always fetched two instructions ahead** of the currently executed instructions.
- The reason **ARM retains this definition is to ensure compatibility with earlier processors.**

Single-Cycle Datapath: STR

24

Expand datapath to handle STR:

- Write data in R_d to memory

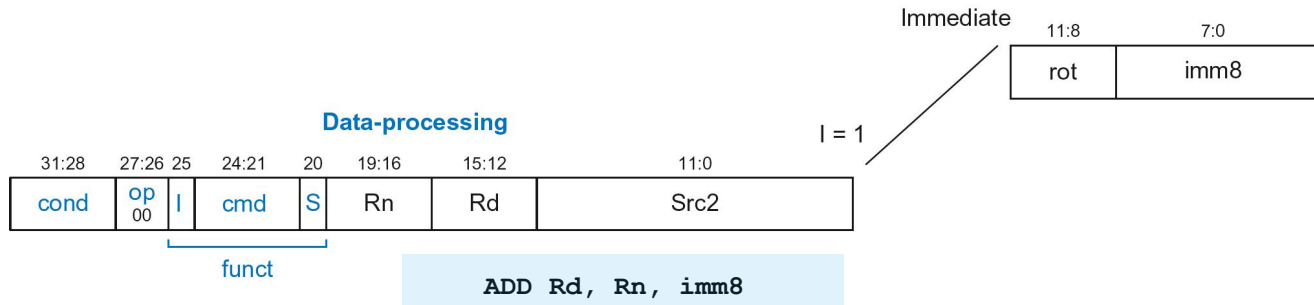
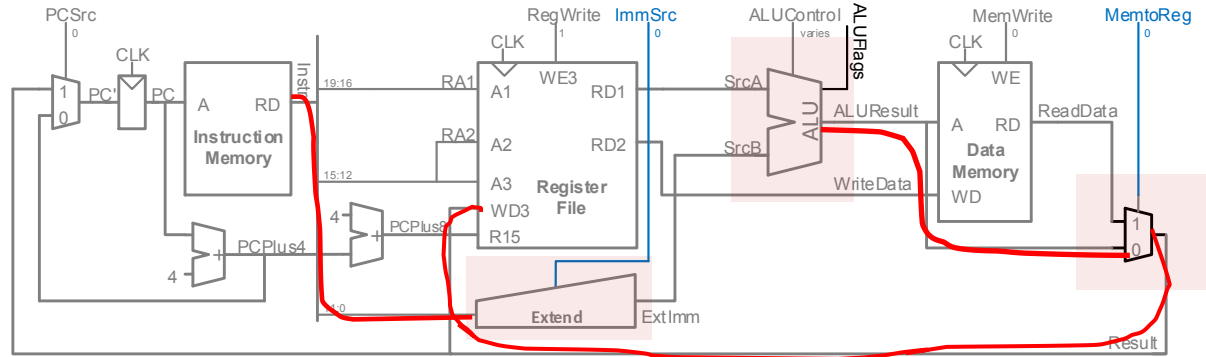


Single-Cycle Datapath: Data-processing

25

With **immediate** Src2:

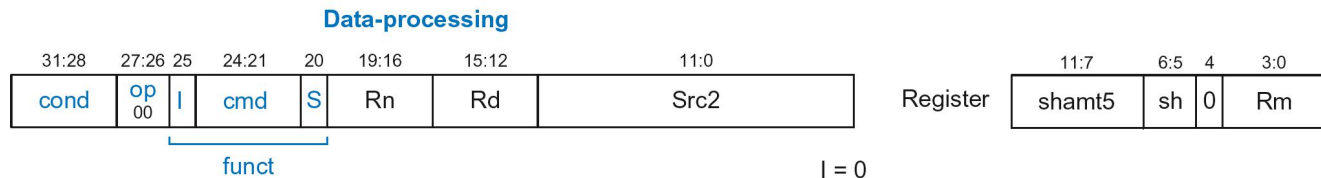
- Read from R_n and $Imm8$ ($ImmSrc$ chooses the zero-extended $Imm8$ instead of $Imm12$)
- Write $ALUResult$ to register file
- Write to R_d



Single-Cycle Datapath: Data-processing

With **register** Src2:

- Read from R_n and R_m (instead of $Imm8$)
- Write *ALUResult* to register file
- Write to R_d



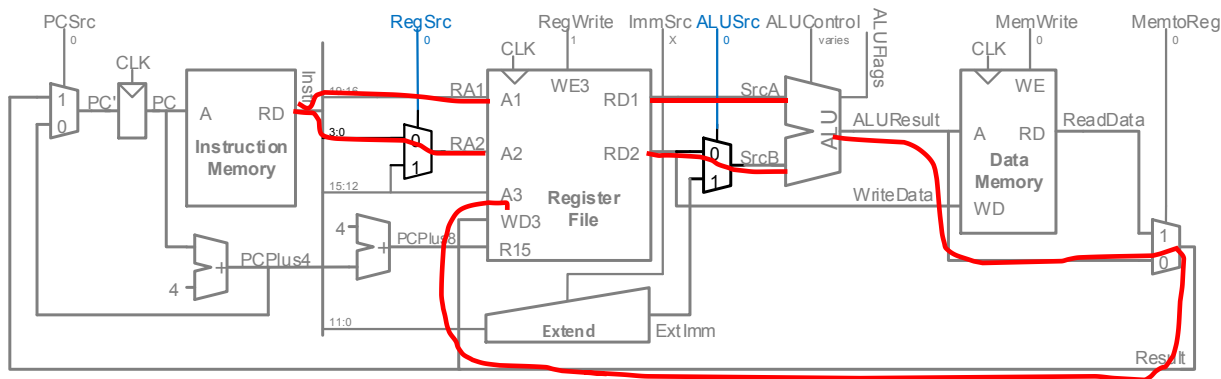
ADD R_d, R_n, R_m

Single-Cycle Datapath: Data-processing

27

With **register Src2**:

- Read from R_n and R_m (instead of $Imm8$)
- Write $ALUResult$ to register file
- Write to R_d



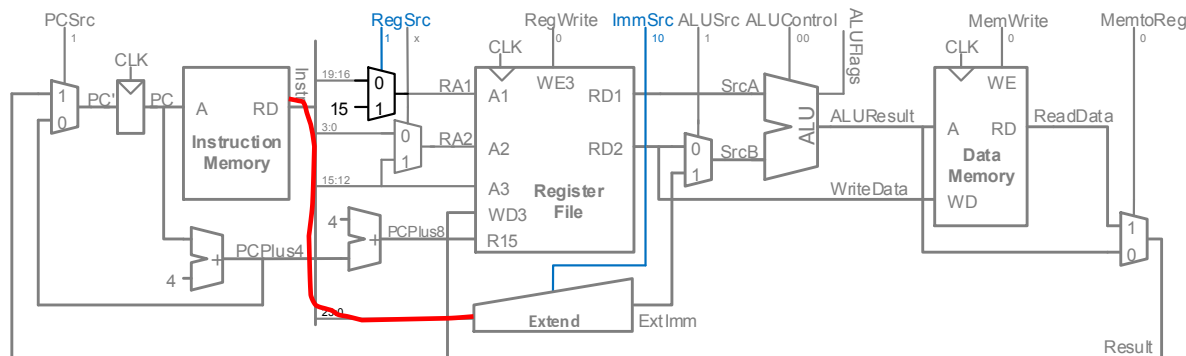
ADD Rd, Rn, Rm

Single-Cycle Datapath: B

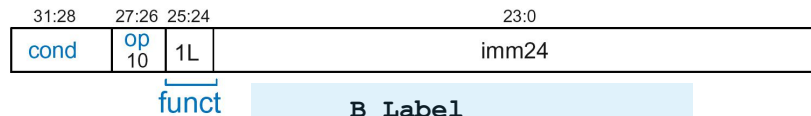
Calculate branch target address:

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

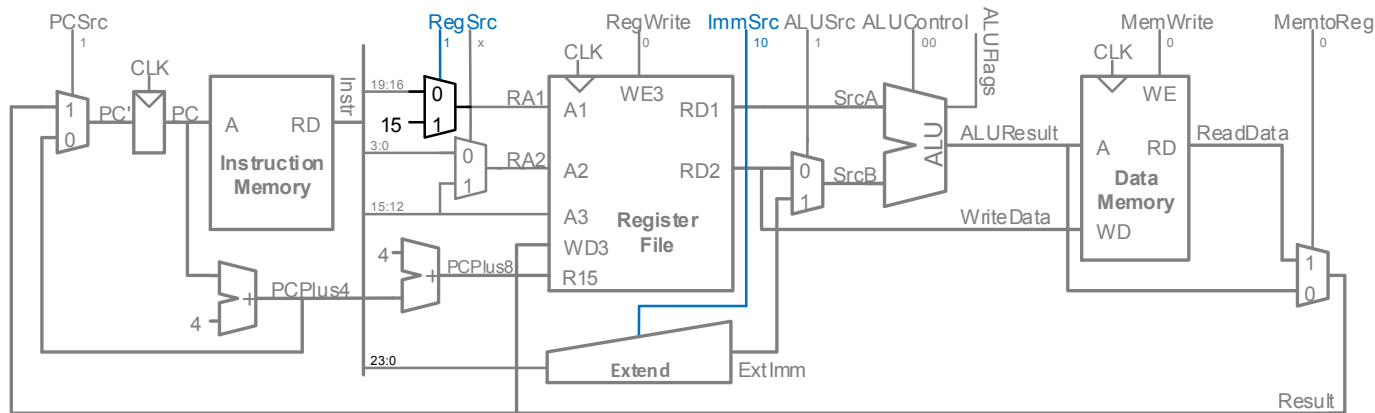
$ExtImm = Imm24 \ll 2$ and sign-extended



Branch



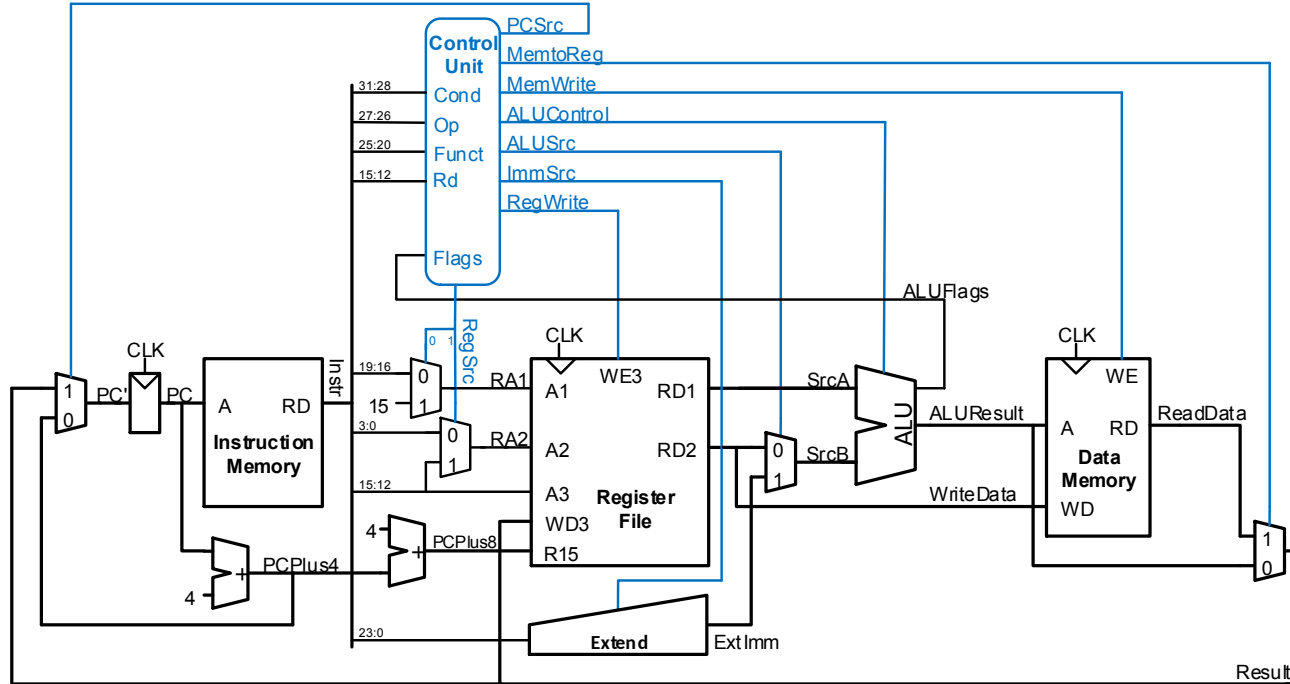
Single-Cycle Datapath: ExtImm



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

Single-Cycle ARM Processor

30



Outline

31

Introduction

Single Cycle Processor

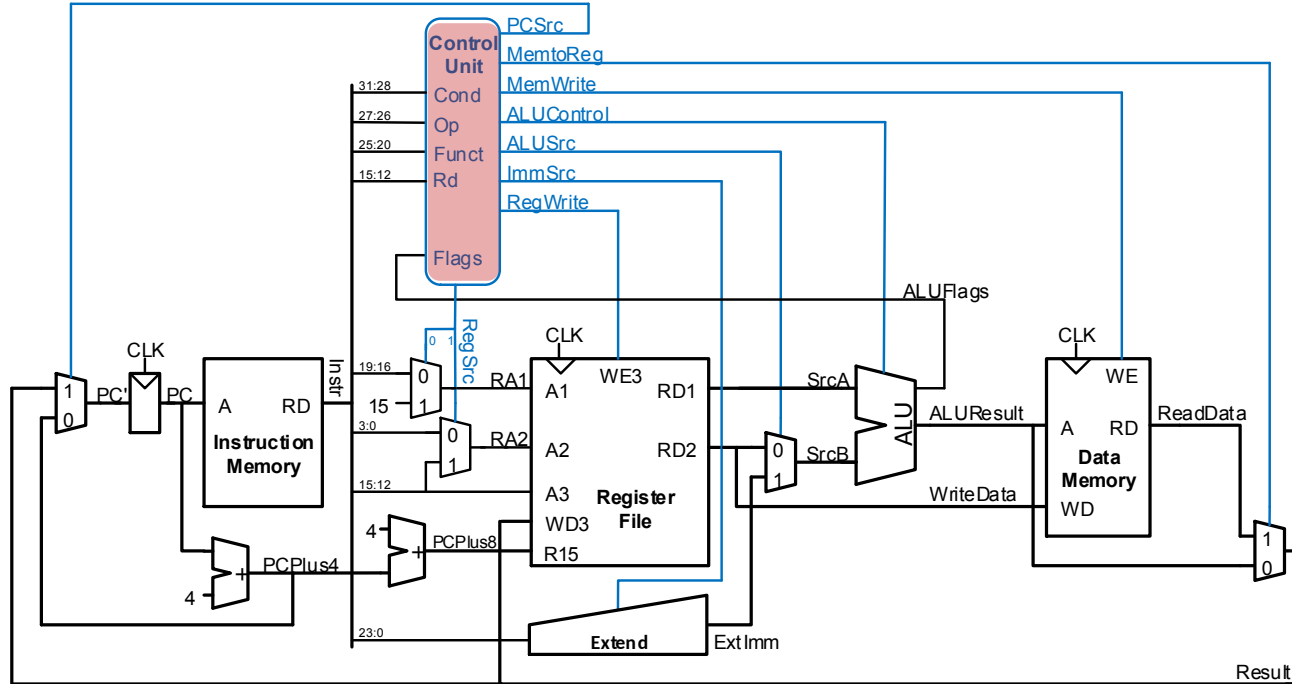
Datapath

Control

Conclusions

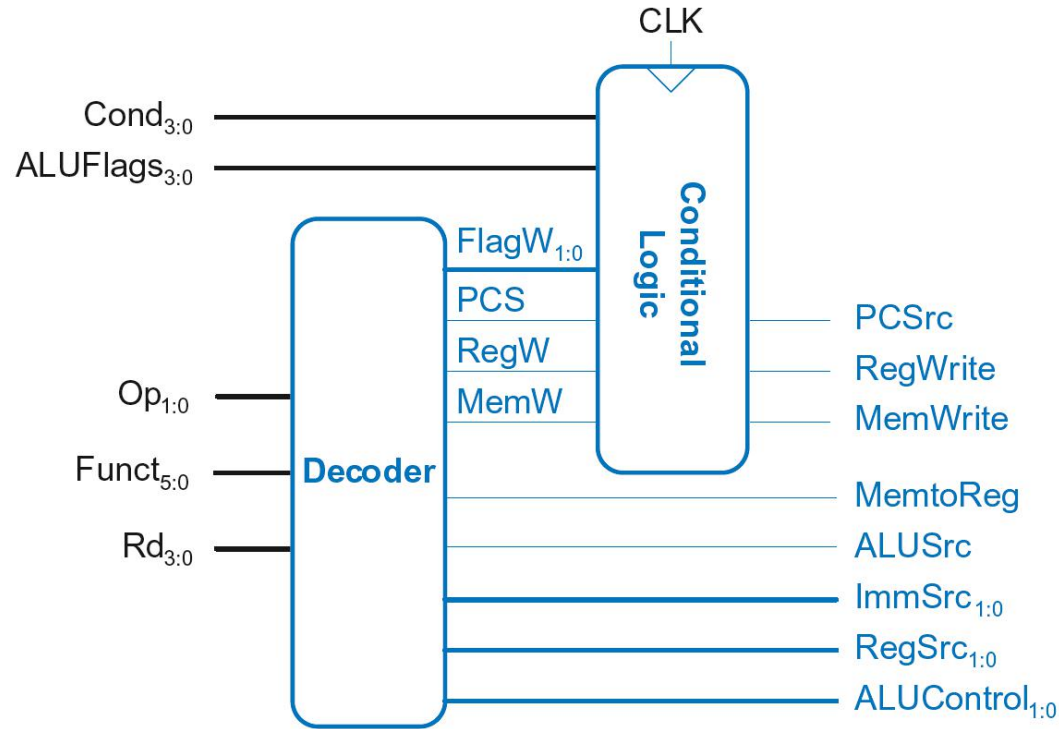
Single-Cycle ARM Processor

32

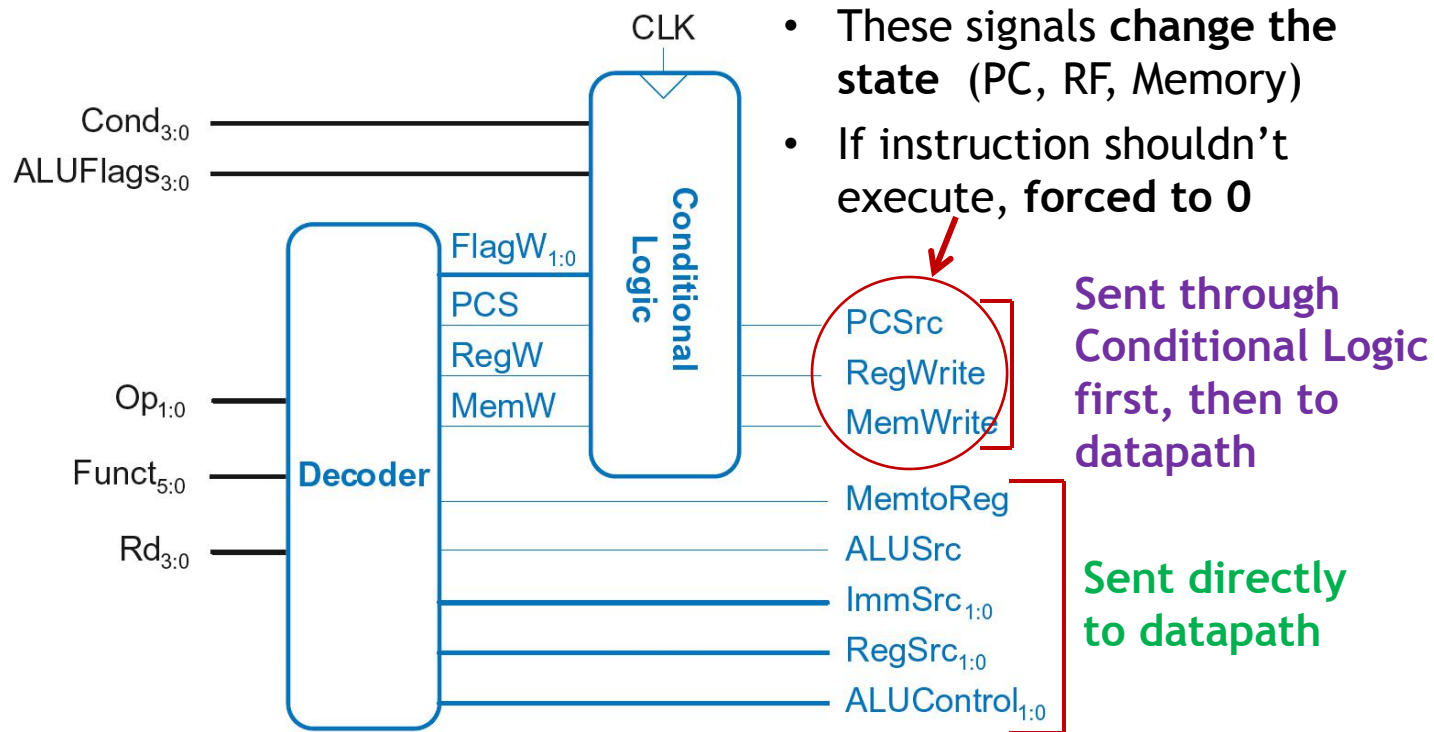


Single-Cycle Control

33

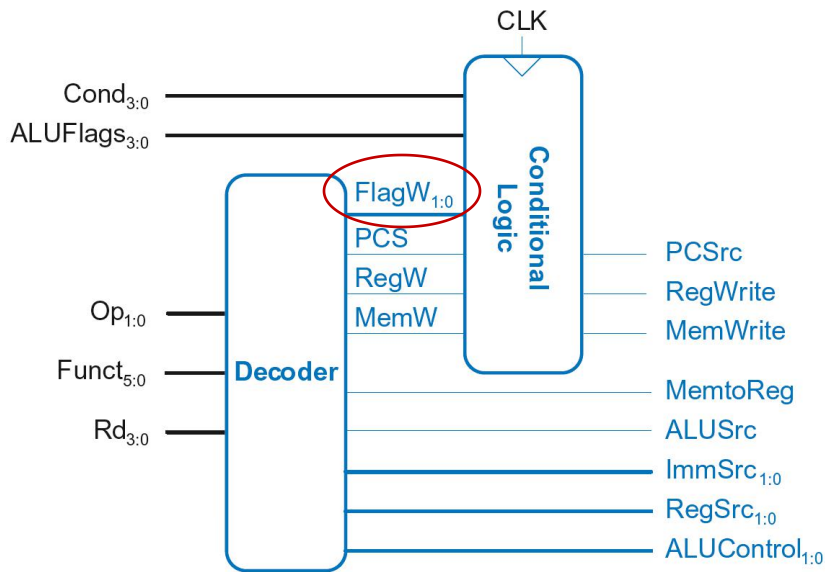


Single-Cycle Control: Signals



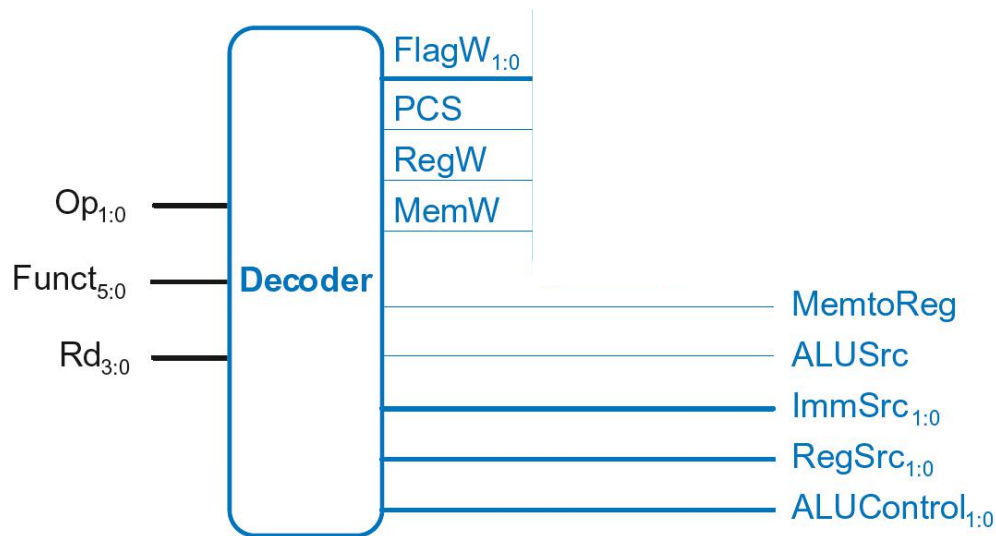
- These signals change the state (PC, RF, Memory)
- If instruction shouldn't execute, forced to 0

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)

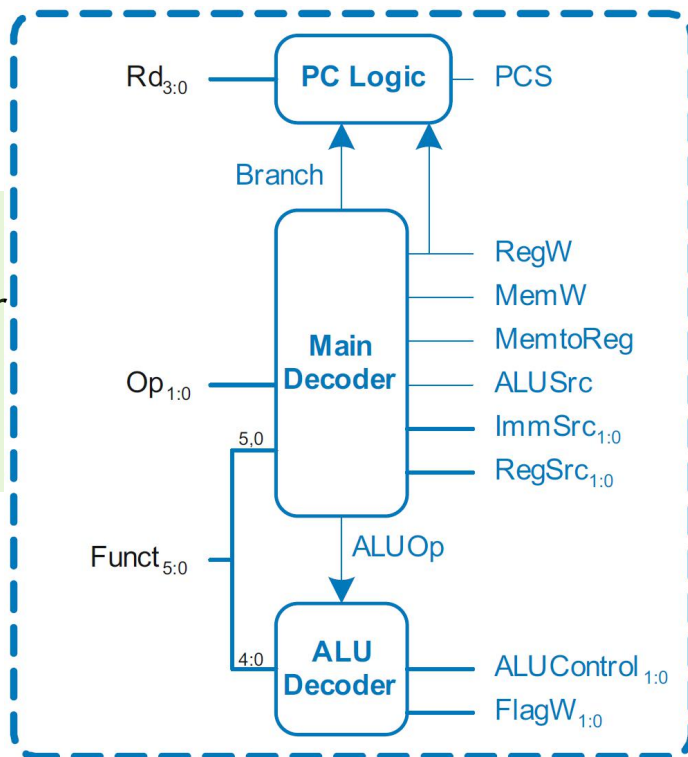
Single-Cycle Control: Decoder



Single-Cycle Control: Decoder

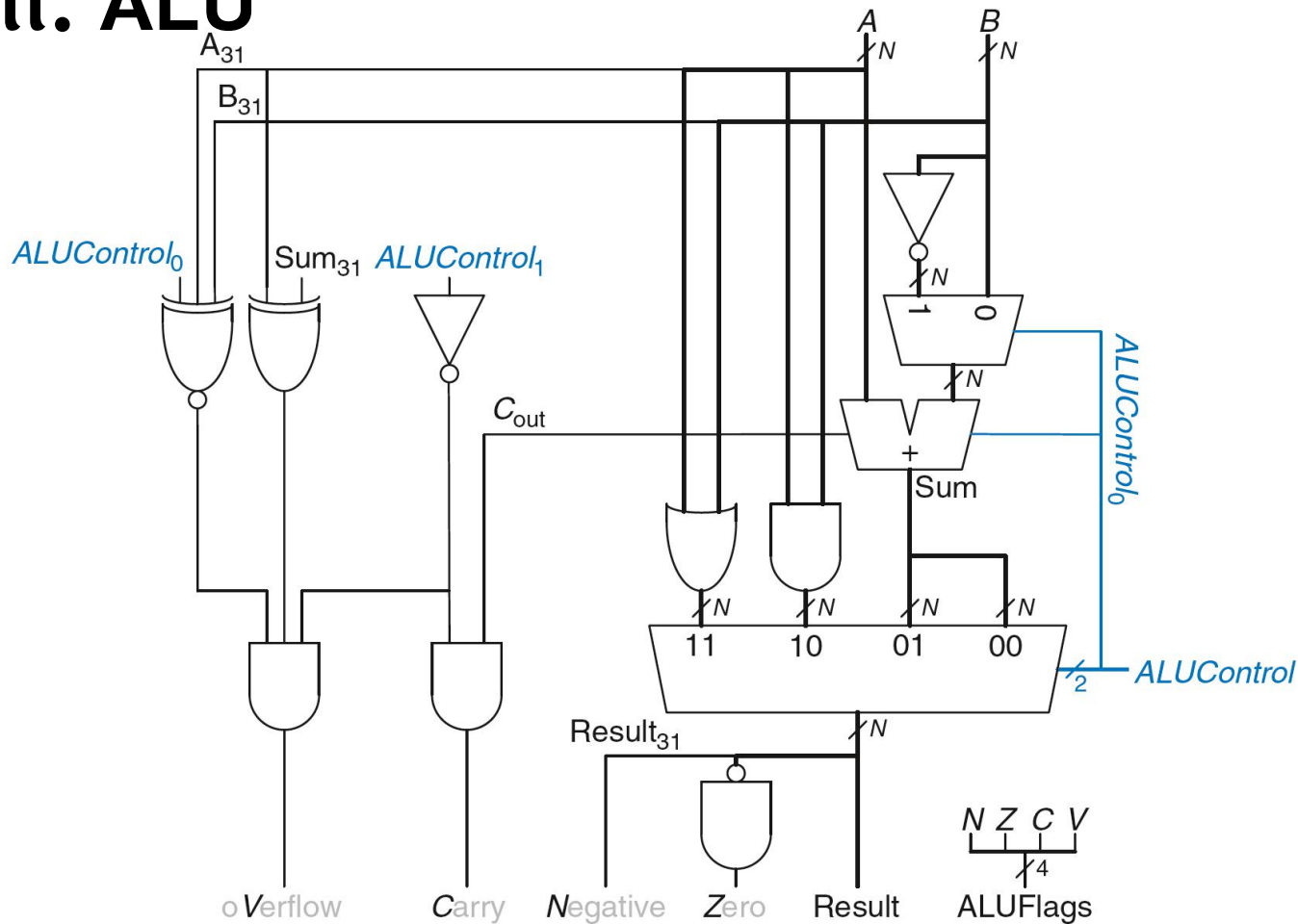
Submodules:

- Main Decoder
- ALU Decoder
- PC Logic



Control Unit: Main Decoder

Op	Funct ₅	Funct ₀	Type	Branch	MementoReg	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



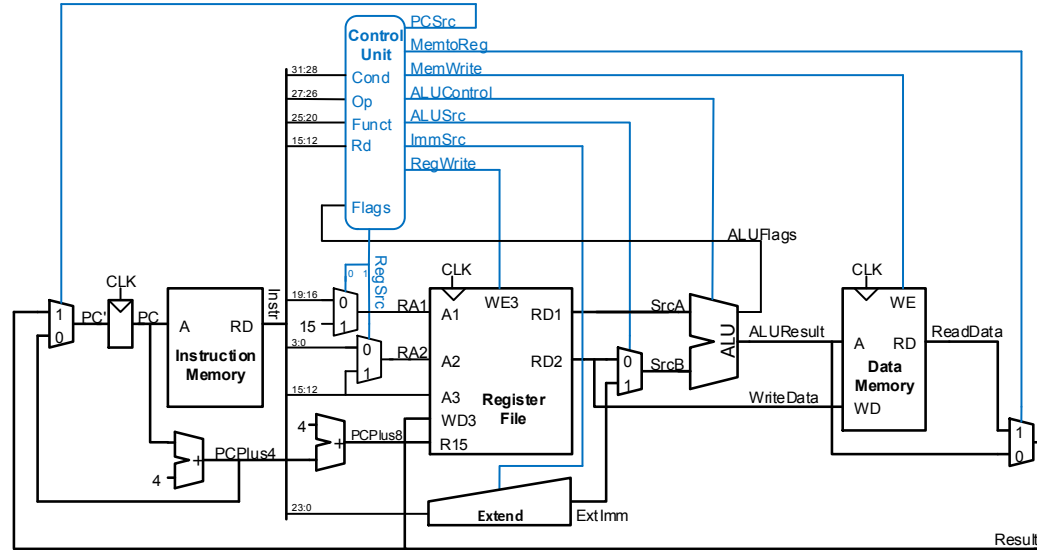
Control Unit: ALU Decoder

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

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

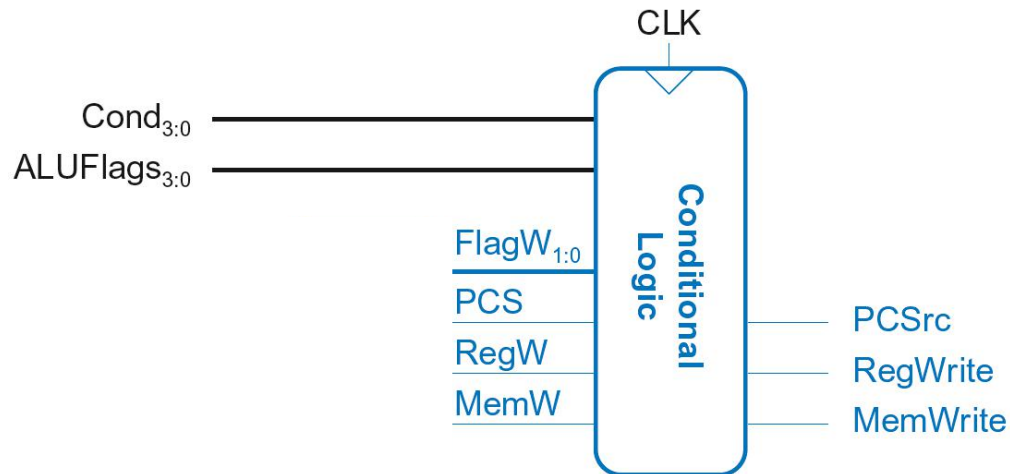
$$PCS = ((Rd == 15) \& RegW) \mid Branch$$



If instruction is executed: $PCSrc = PCS$

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

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})

Logic diagram of the Condition Check block. The block has inputs: PCS, RegW, MemW, FlagW_{1:0}, Cond_{3:0}, ALUFlags_{3:0}, and CLK. It has outputs: PCSrc, RegWrite, MemWrite, and CondEx. The logic involves AND gates combining PCS, RegW, and MemW to produce PCSrc, RegWrite, and MemWrite. The FlagW_{1:0} input is ANDed with the Cond_{3:0} input to produce FlagWrite_{1:0}. The ALUFlags_{3:0} input is split into two 2-bit buses, each with a 2-to-1 multiplexer controlled by CLK. The outputs of these multiplexers are Flags_{3:2} and Flags_{1:0}. The CondEx output is the AND of FlagWrite_{1:0} and Flags_{1:0}. The Condition Check block is a large rectangle that takes Flags_{3:2}, Flags_{1:0}, and Cond_{3:0} as inputs.

CS3501-COMPUTER ARCHITECTURE

Recall: Condition Mnemonics

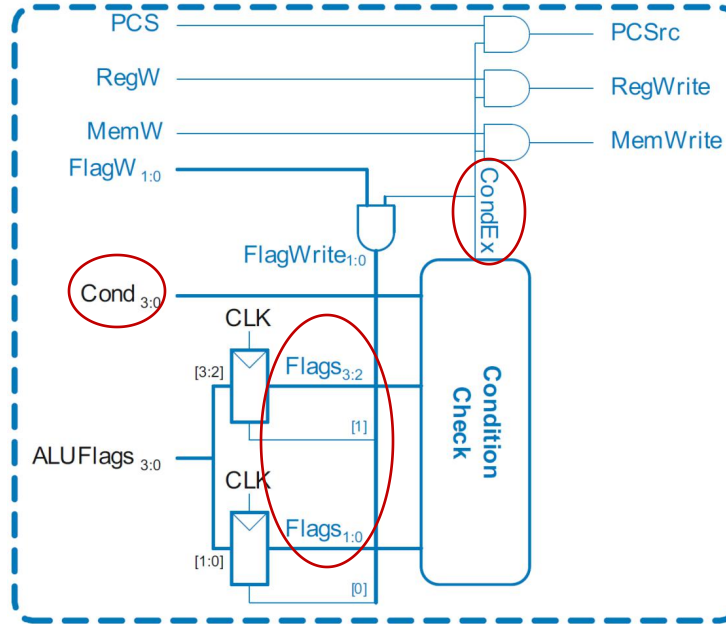
44

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

Example: Conditional Execution

45

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



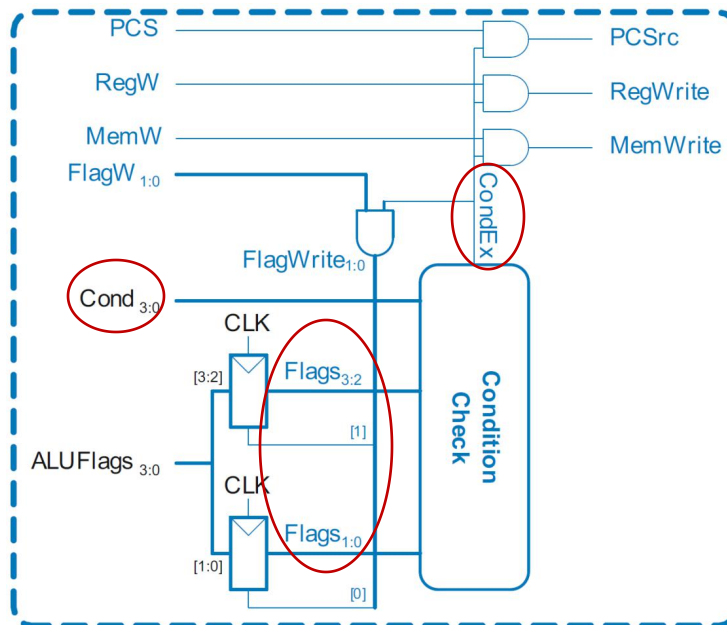
Example:

AND R1, R2, R3

$\text{Cond}_{3:0} = 1110$ (unconditional) $\Rightarrow \text{CondEx} = 1$

Example: Conditional Execution

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

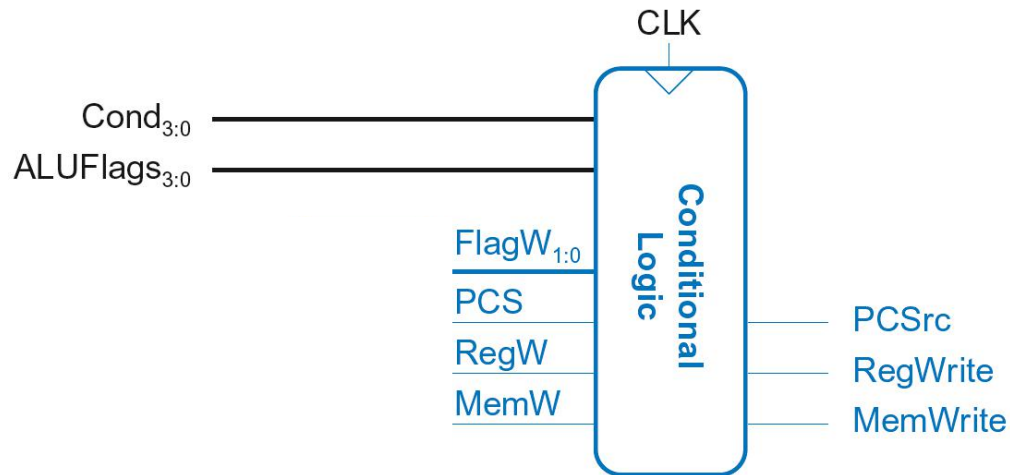


Example:

EOREQ R5, R6, R7

$\text{Cond}_{3:0} = 0000$ (EQ): if $\text{Flags}_{3:2} = 0100 \Rightarrow \text{CondEx} = 1$

Conditional Logic



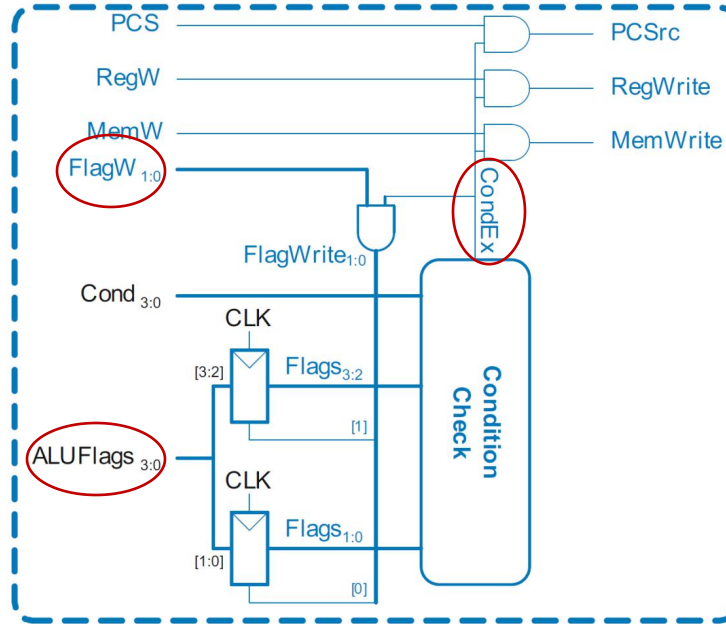
Function:

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

48

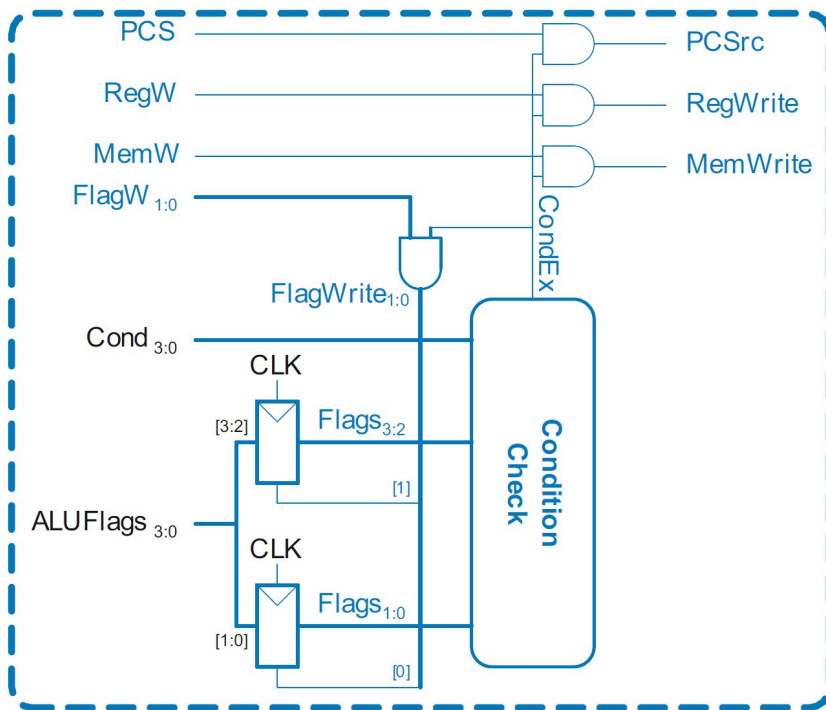
$\text{Flags}_{3:0} = \text{NZCV}$



$\text{Flags}_{3:0}$ 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:
FlagW_{1:0}

Recall: ALU Decoder

50

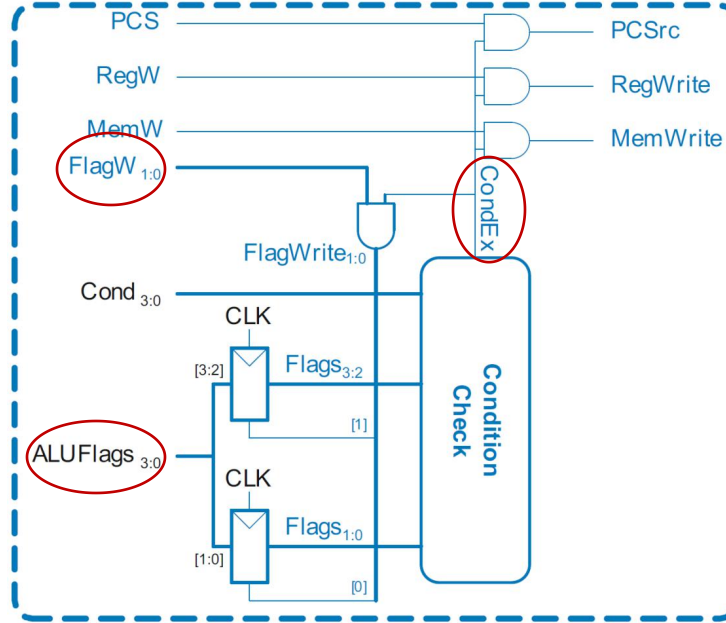
ALUOp	Funct _{4:1} (cmd)	Funct ₀ (S)	Type	ALUControl _{1:0}	FlagW _{1:0}
0	X	X	Not DP	00	00
1	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10

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

Example: Update (Set) Flags

51

All Flags
updated



Example: SUBS R5, R6, R7

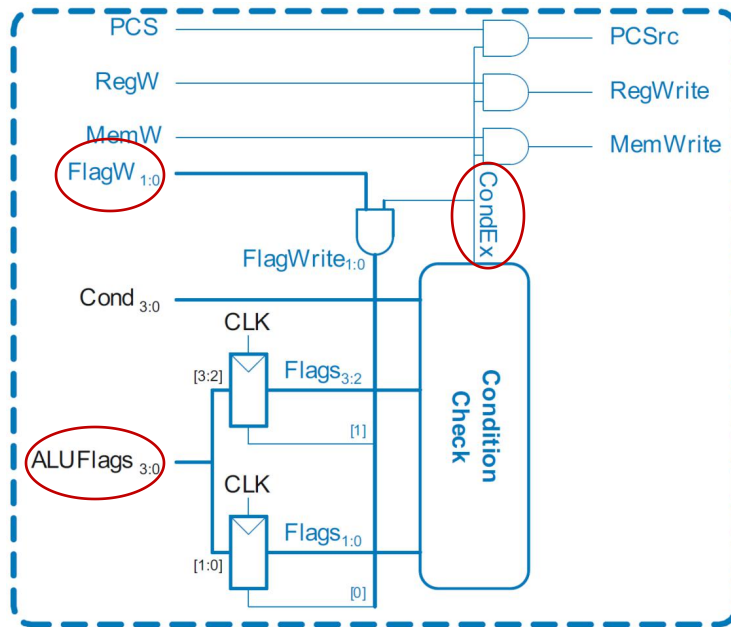
FlagW_{1:0} = 11 AND CondEx = 1 (unconditional) => FlagWrite_{1:0} = 11

Example: Update (Set) Flags

52

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

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



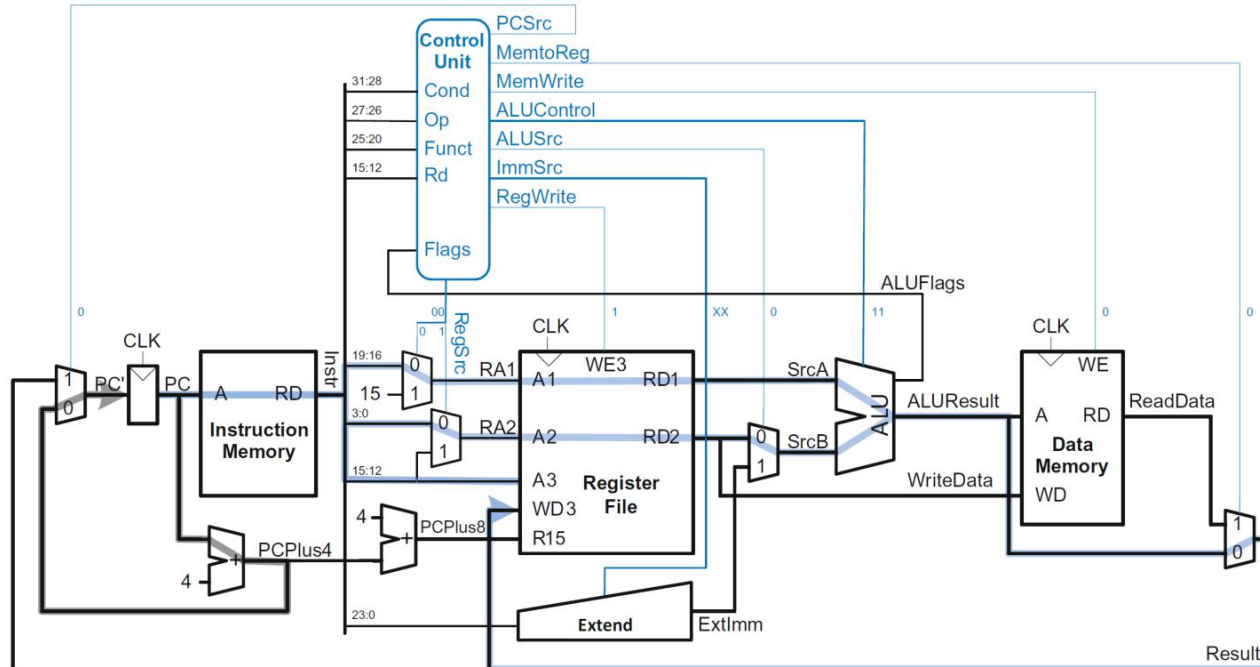
Example: `ANDS R7, R1, R3`

$\text{FlagW}_{1:0} = 10$ AND $\text{CondEx} = 1$ (unconditional) $\Rightarrow \text{FlagWrite}_{1:0} = 10$

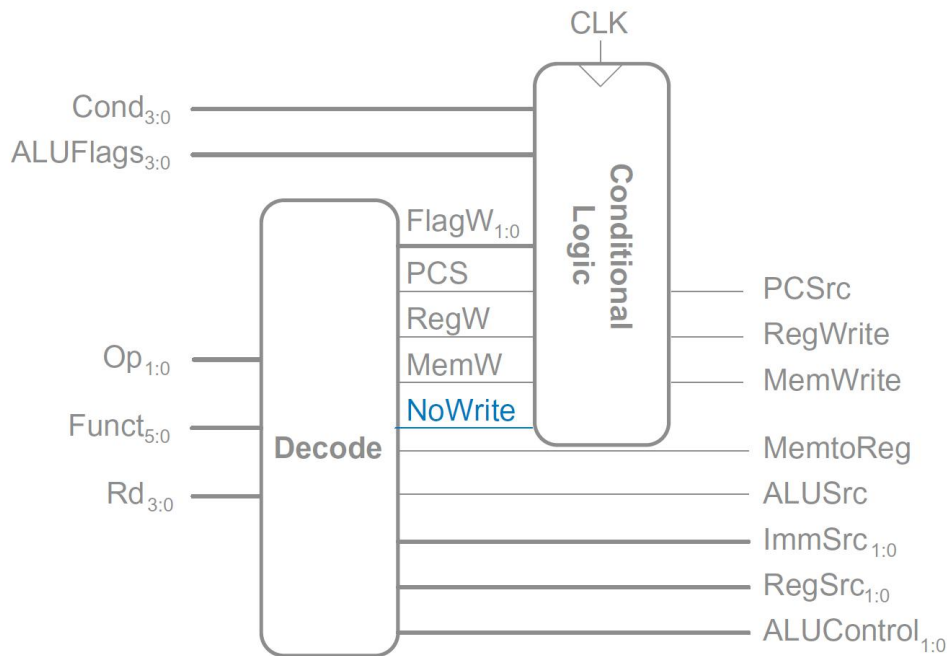
Example: ORR

53

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



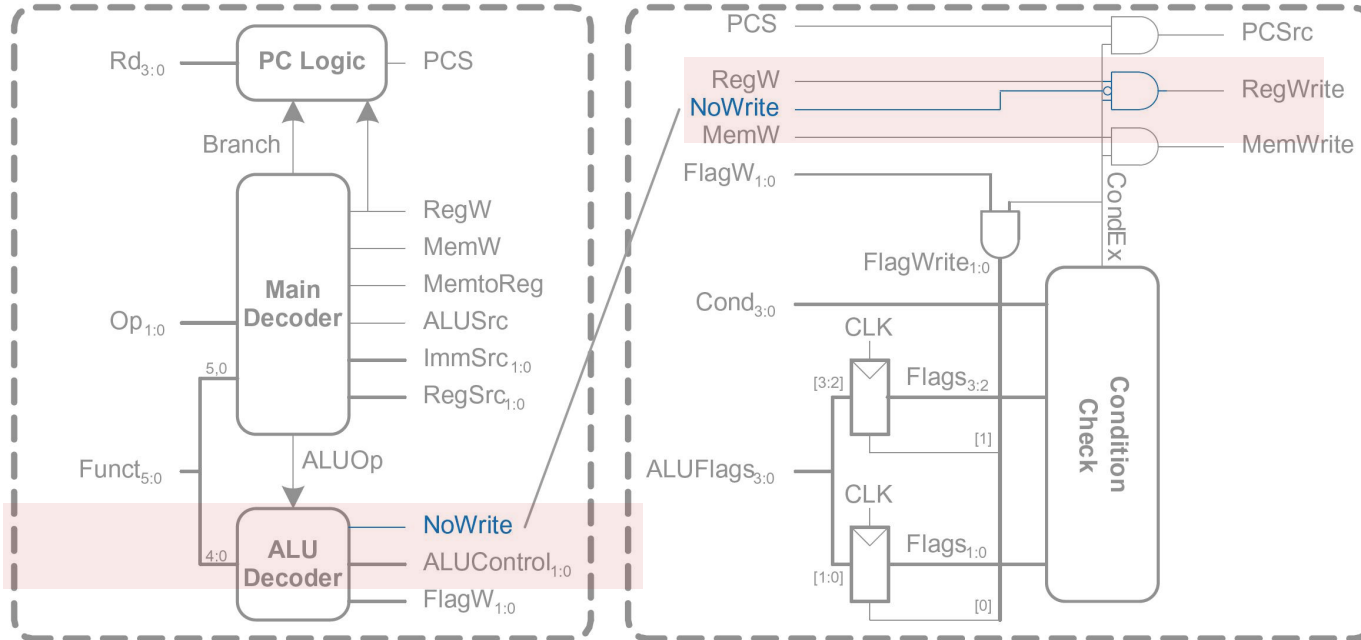
Extended Functionality: CMP



No change to datapath

Extended Functionality: CMP

55

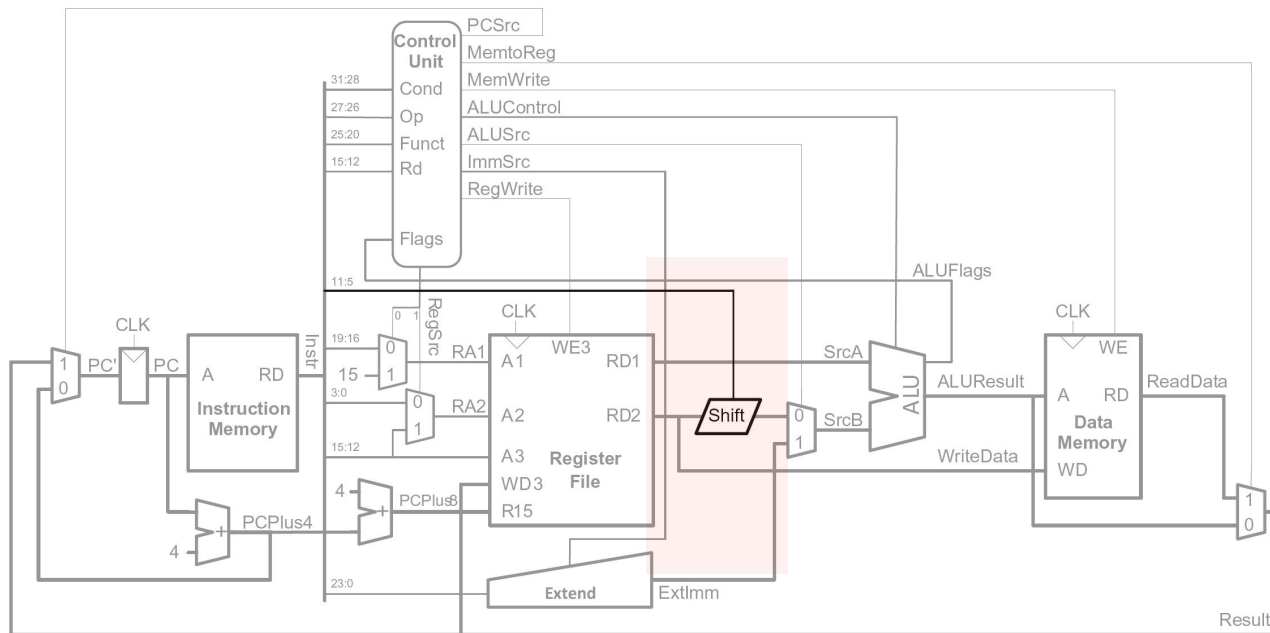


Extended Functionality: CMP

56

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

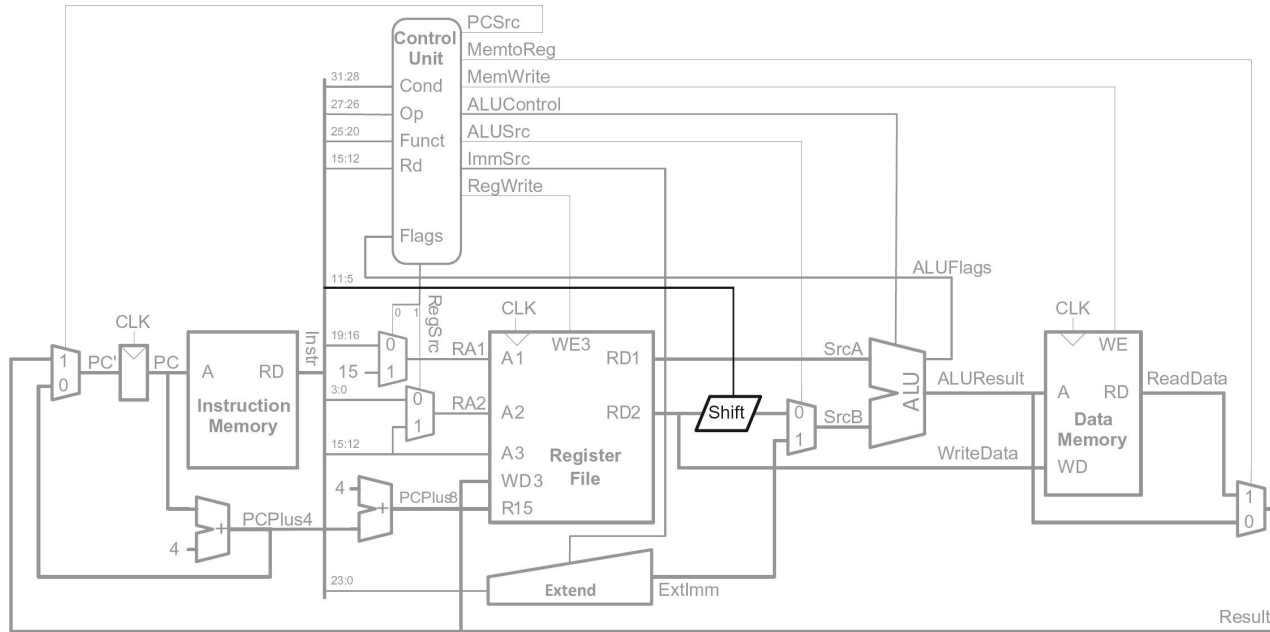


ADD R7, R2, R12, LSR #5

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	12
cond	op	I	cmd	S	rn	rd	shamt5	sh		rm

Extended Functionality: Shifted Register

58



No change to controller

Outline

59

Introduction

Single Cycle Processor

Datapath

Control

Conclusions

Conclusions

60

- 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 - 2023I

PROF.: JGONZALEZ@UTEC.EDU.PE

