COMP2300-COMP6300-ENGN2219
Computer Organization & Program Execution

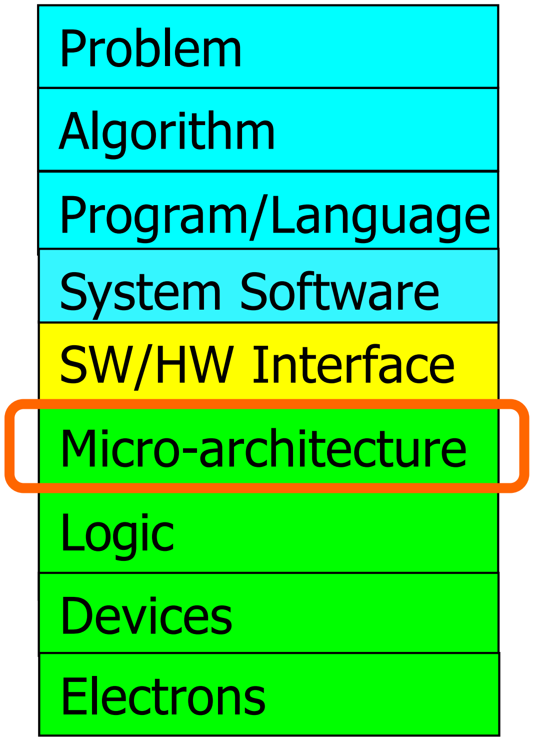
Convener: Shoaib Akram

shoaib.akram@anu.edu.au

Australian National University

Agenda for Remaining Lectures

- Multi-cycle microarchitecture
- Pipelining
  - Data and control hazards
  - State maintenance and interrupts
- Out-of-order execution
  - Key to high performance in modern processors



Readings

- Multi-cycle microarchitecture (Section 7.4)

Multi-Cycle Microarchitecture

Acknowledgement: *Selection of Slides from Digital Design and Computer Architecture, Onur Mutlu, ETH Zurich, Spring 2022*

Recall: Single-Cycle Microarchitecture (Very Basic)

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - *No intermediate, programmer-invisible state updates*

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

↓

Process instruction in **one clock cycle**

↓

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

Recall: The Instruction Processing "Cycle"

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

Recall: The Instruction Processing "Cycle"

- FETCH
- DECODE + Register Read
- EVALUATE ADDRESS/Execute
- FETCH OPERANDS (Mem Access)
- STORE RESULT (Writeback)

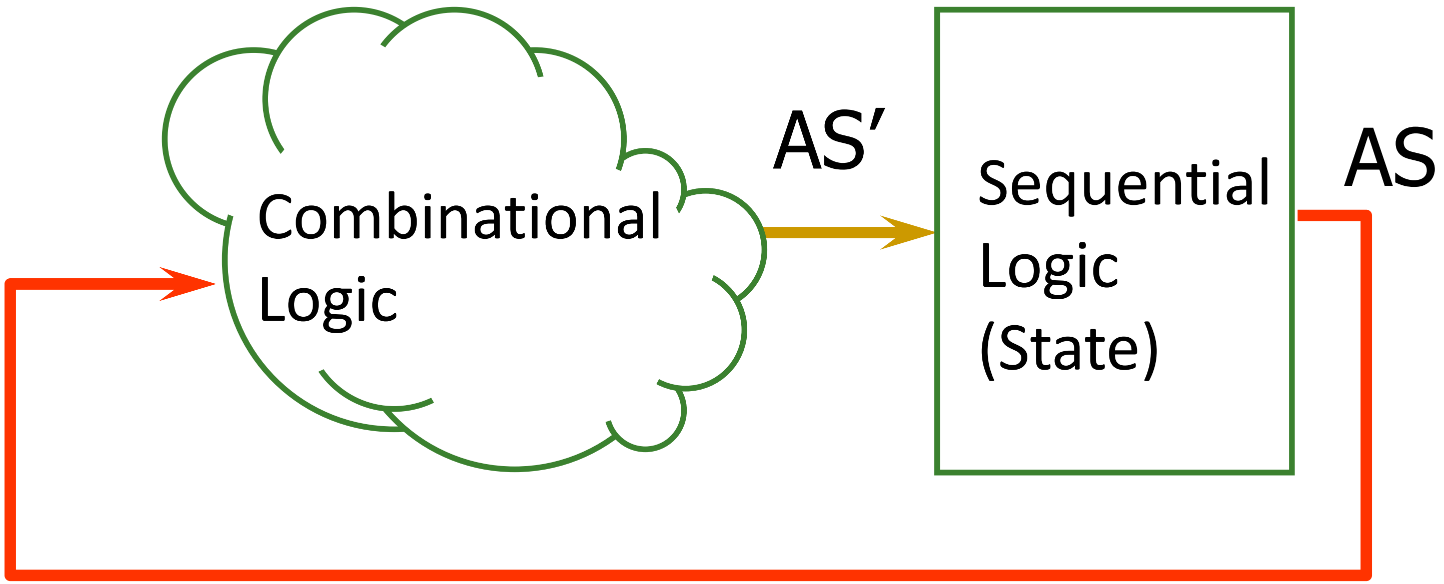
Instruction Processing "Cycle" vs. Machine Clock Cycle

- Single-cycle machine:
  - All phases of the instruction processing cycle take a *single machine clock cycle* to complete

- Multi-cycle machine:
  - Each phase of the instruction processing cycle can take *multiple machine clock cycles* to complete

Recall: Single-Cycle Machine

- Single-cycle machine



AS: Architectural State

Recall: Datapath and Control Logic

- An instruction processing engine consists of two components
  - Datapath: Consists of hardware elements that deal with and transform data signals
    - **functional units** that operate on data
    - **hardware structures** (e.g., wires, muxes, decoders, tri-state bufs) that enable the flow of data into the functional units and registers
    - **storage units** that store data (e.g., registers)
  - Control logic: Consists of hardware elements that determine control signals, i.e., signals that specify what the datapath elements should do to the data

A Single-Cycle Microarchitecture: Analysis

- Every instruction takes 1 cycle to execute
  - CPI (Cycles per instruction) is strictly 1

- How long each instruction takes is determined by how long the slowest instruction takes to execute
  - Even though many instructions do not need that long to execute

- Clock cycle time of the microarchitecture is determined by how long it takes to complete the slowest instruction
  - Critical path of the design is determined by the processing time of the slowest instruction

What is the Slowest Instruction to Process?

- Let's go back to the basics

- All phases of the instruction processing cycle take a *single machine clock cycle* to complete
  - Instruction Fetch (IF)
  - Instruction Decode and Register Read (ID/RF)
  - Execute (EX)
  - Memory Access (MEM)
  - Writeback (WB)

- Does every instruction take the same time (latency) to complete?

What is Really the Slowest Instruction to Process?

- Real world: **Memory is slow (not magic)**

- What if memory *sometimes* takes 150ns to access?

- Does it make sense to have a simple register to register add or jump to take {150ns + all else to perform a memory operation}?

- And, what if you need to access memory more than once to process an instruction?
  - Which instructions require this?
  - Do you provide multiple ports to memory?

Single-Cycle uArch: Complexity

- Contrived
  - All instructions run as slow as the slowest instruction
- Inefficient
  - All instructions run as slow as the slowest instruction
  - Must provide worst-case combinational resources in parallel as required by any instruction
  - Need to replicate a resource if it is needed more than once by an instruction during different parts of the instruction processing cycle
- Not necessarily the simplest way to implement an ISA
  - Tough for complex instructions, e.g., REP MOVS (x86) or INDEX (VAX)
- Not easy to optimize/improve performance
  - Optimizing the common case (frequent instructions) does not work
  - Need to optimize the worst case all the time

Multi-Cycle Microarchitectures

- Goal: Let each instruction take (close to) only as much time it really needs

- Idea
  - Determine clock cycle time independently of instruction processing time
  - Each instruction takes as many clock cycles as it needs to take
    - Multiple state transitions per instruction
    - The states followed by each instruction is different

Recall: The "Process Instruction" Step

- ISA specifies abstractly what AS' should be, given an instruction and AS
  - It defines an **abstract finite state machine** where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no "intermediate states" between AS and AS' during instruction execution
    - One state transition per instruction

- Microarchitecture implements how AS is transformed to AS'
  - There are many choices in implementation
  - We can have programmer-invisible state to optimize the speed of instruction execution: **multiple** state transitions per instruction
    - Choice 1: AS → AS' (transform AS to AS' in a single clock cycle)
    - Choice 2: AS → AS+MS1 → AS+MS2 → AS+MS3 → AS' (take multiple clock cycles to transform AS to AS')

Multi-Cycle Microarchitecture

AS = Architectural (programmer visible) state
at the beginning of an instruction

Step 1: Process part of instruction in one clock cycle

Step 2: Process part of instruction in the next clock cycle

…

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

Benefits of Multi-Cycle Design

- Critical path design
  - Can keep reducing the critical path independently of the worst-case processing time of any instruction
- Can optimize the common case
  - Can optimize the number of states it takes to execute "important" instructions that make up much of the execution time
- Efficient/balanced design
  - No need to provide more capability or resources than really needed
    - An instruction that needs resource X multiple times does **not** require multiple X's to be implemented
    - Leads to more efficient hardware: Can reuse hardware components needed multiple times for an instruction

Downsides of Multi-Cycle Design

- Need to store the intermediate results at the end of each clock cycle
  - Hardware overhead for microarchitectural registers
  - Register setup/hold overhead (i.e., sequencing overhead) is paid multiple times for an instruction

Remember: Performance Analysis

- Execution time of a single instruction
  - **{CPI} x {clock cycle time}**

CPI: Cycles Per Instruction

- Execution time of an entire program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - **{# of instructions} x {Average CPI} x {clock cycle time}**

- Single-cycle microarchitecture performance
  - CPI = 1
  - Clock cycle time = long
- Multi-cycle microarchitecture performance
  - CPI = different for each instruction
    - Average CPI → hopefully small
  - Clock cycle time = short

In multi-cycle, we have two degrees of freedom to optimize independently

Multi-Cycle Microarchitectures

- Key Idea for Realization
  - One can implement the "process instruction" step as a finite state machine that sequences between states and eventually returns back to the "fetch instruction" state
  - A state is defined by the control signals asserted in it
  - Control signals for the next state are determined in current state

A Basic Multi-Cycle Microarchitecture

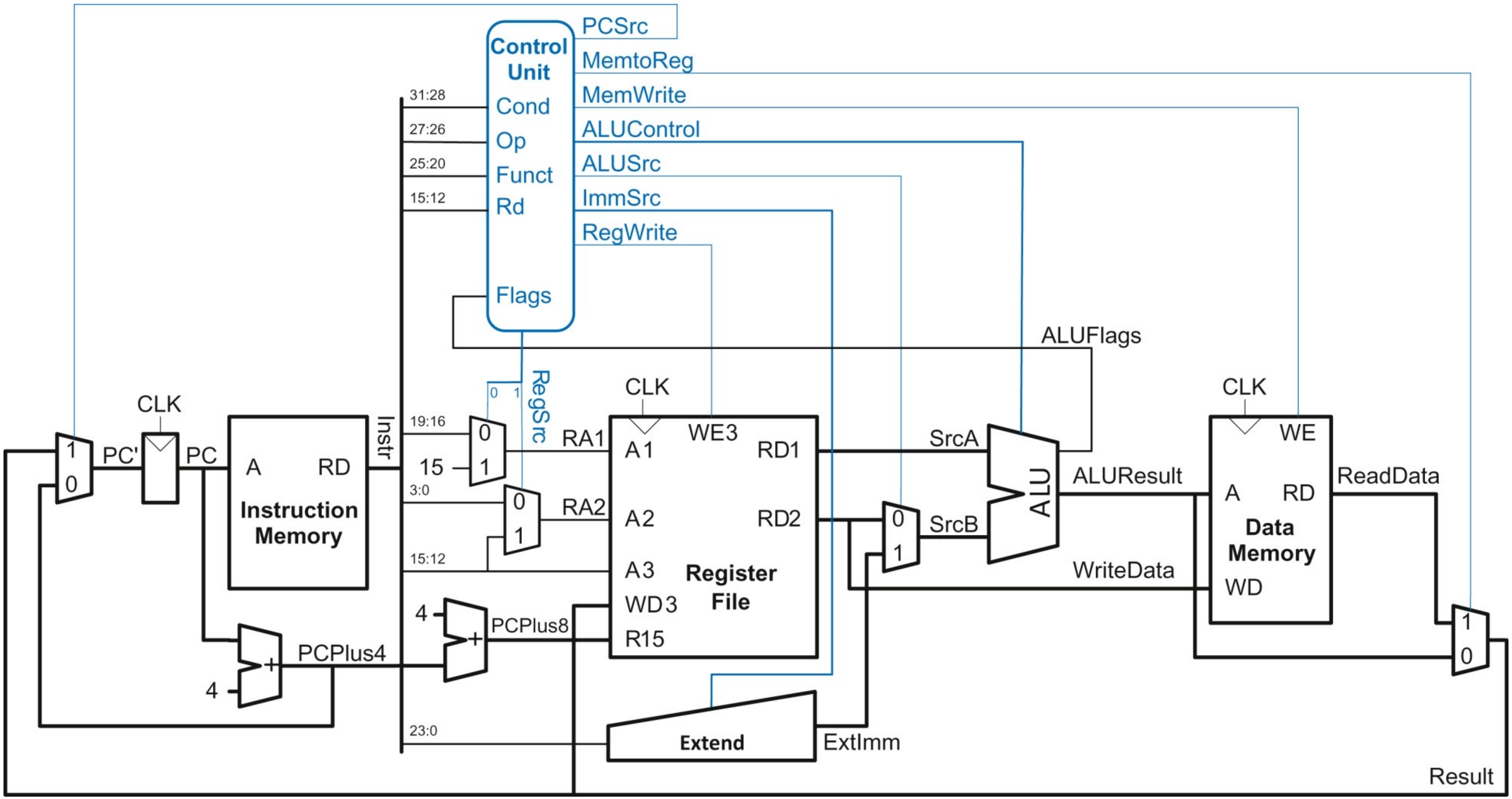
- Instruction processing cycle divided into "states"
  - A stage in the instruction processing cycle can take multiple states

- A multi-cycle microarchitecture sequences from state to state to process an instruction
  - The behavior of the machine in a state is completely determined by control signals in that state

- The behavior of the entire processor is specified fully by a *finite state machine*

- In a state (clock cycle), control signals control two things:
  - How the datapath should process the data
  - How to generate the control signals for the (next) clock cycle

Remember the Single-Cycle Uarch



Why do we Want Multi-Cycle?

- Single-cycle microarchitecture:
  - -- cycle time limited by longest instruction (`LDR`) → low clock frequency
  - -- three adders/ALUs and two memories → high hardware cost

- Multi-cycle microarchitecture:
  - \+ higher clock frequency
  - \+ simpler instructions take only a few clock cycles
  - \+ reuse expensive hardware across multiple cycles
  - -- hardware overhead for storing intermediate results
  - -- sequential logic overhead paid many times for each instruction

- Multi-cycle requires the same design steps as single cycle:
  - datapath
  - control logic

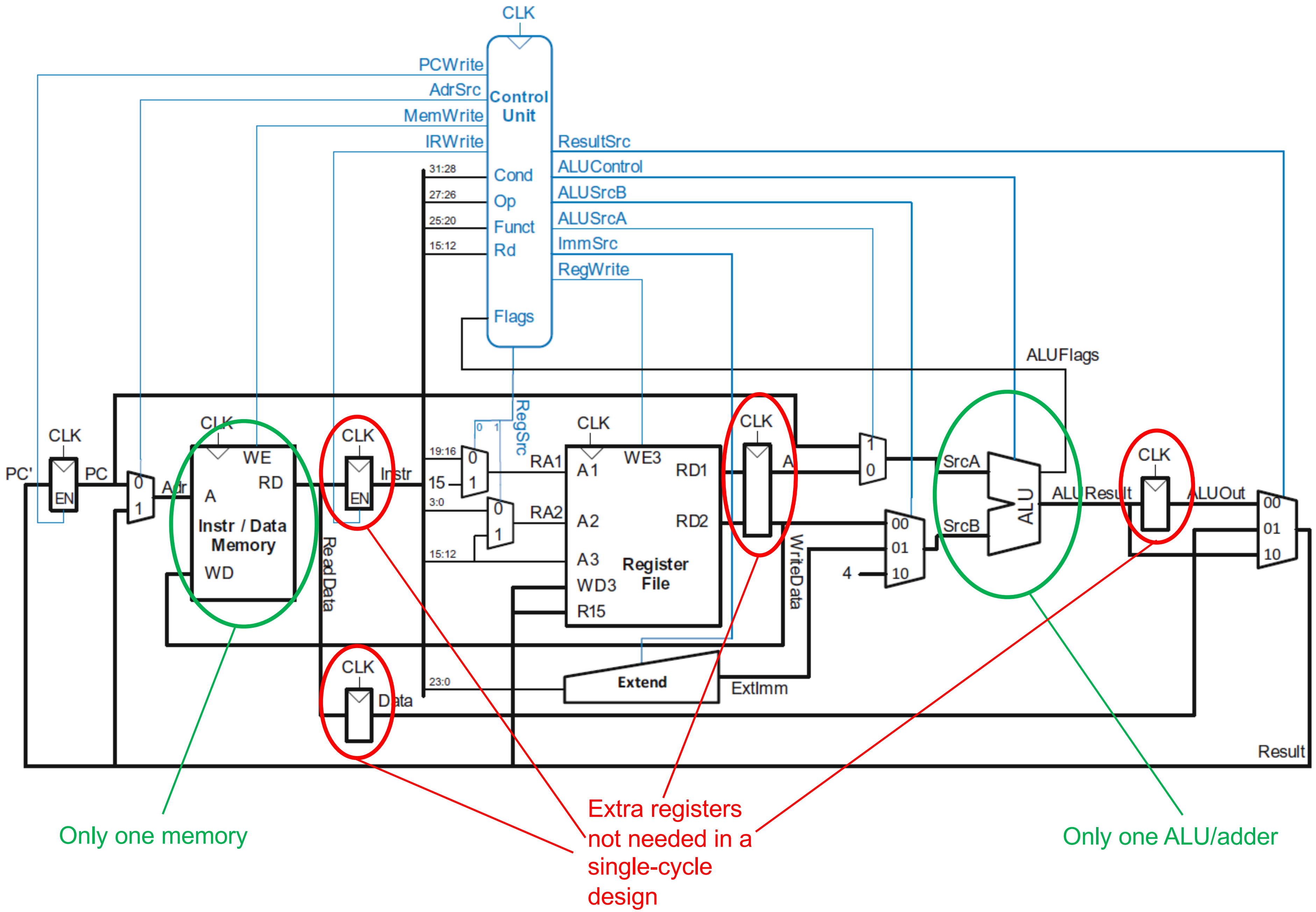
What Can We Optimize with Multi-Cycle

- Single-cycle microarchitecture uses two memories
  - One memory stores instructions, the other data
  - We want to use a single memory (lower cost)

- Single-cycle microarchitecture needs three adders
  - ALU, PC, Branch address calculation
  - We want to use only one ALU for all operations (lower cost)

- Single-cycle microarchitecture: each instruction takes one cycle
  - The slowest instruction slows down every single instruction
  - We want to determine clock cycle time independently of instruction processing time
    - Divide each instruction into multiple clock cycles
    - Simpler instructions can be very fast (compared to the slowest)

Overview: Multicycle ARM Processor



Let's Construct the Multi-Cycle Datapath for 32-bit ARM

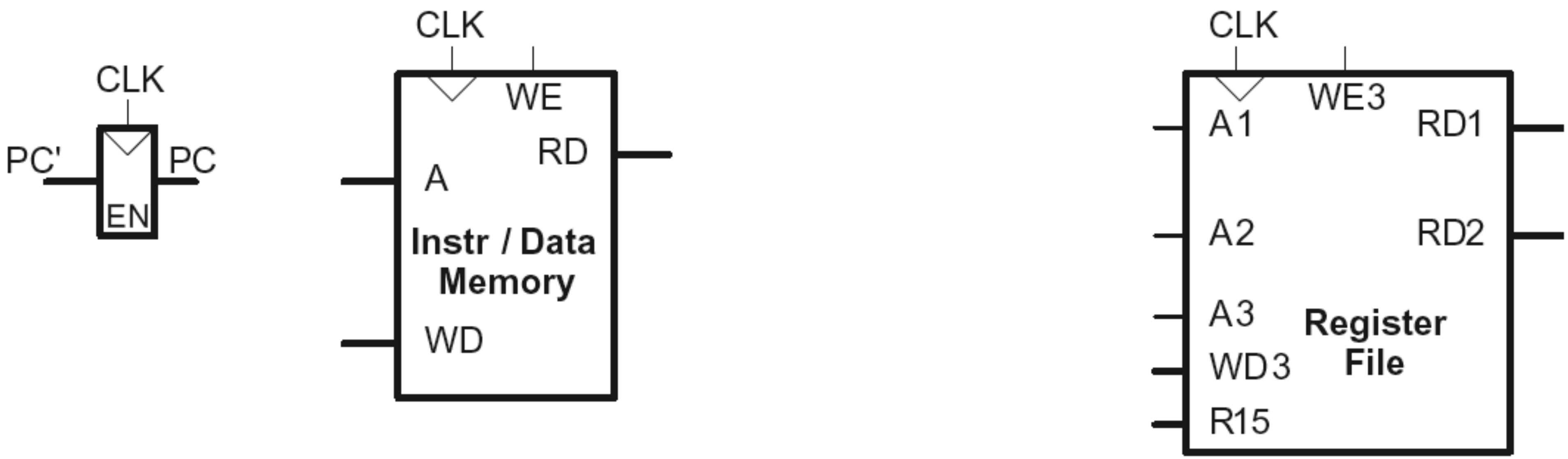
Consider the LDR Instruction

- `LDR R0, [R1, #32]`

- We need to:
  - Read the instruction from memory
  - Then read **R1** from the register file
  - Add the immediate value (`#32`) to calculate the memory address
  - Read the value at this memory address
  - Write to the register `R0` this value

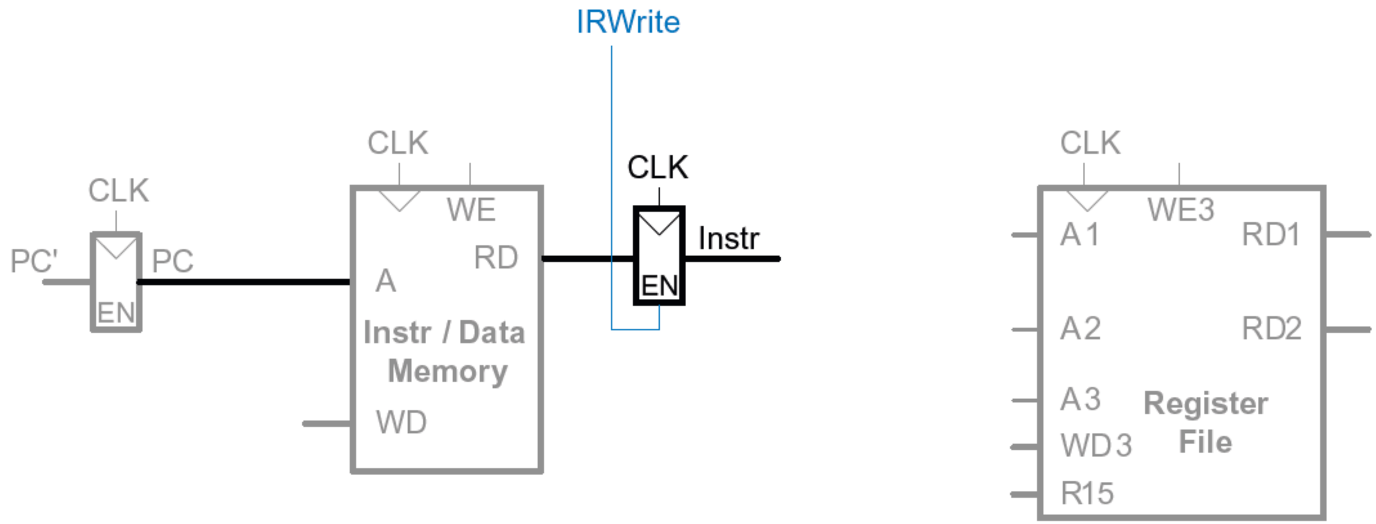
Multicycle State Elements

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



Multicycle Datapath: Instruction fetch

STEP 1: Fetch instruction

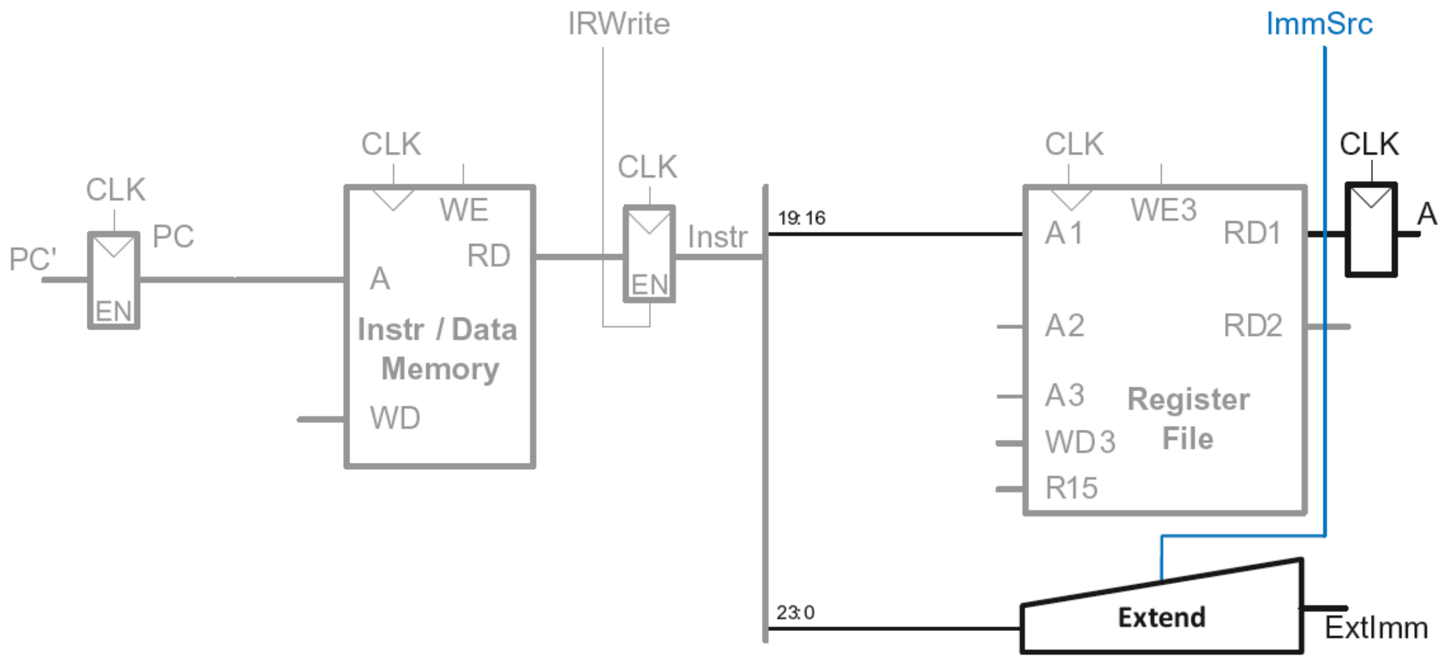


31:28	27:26	25:20						19:16	15:12	11:0
cond	01	1	1	1	0	0	L	Rn	Rd	imm12

LDR Rd, [Rn, imm12]

Multicycle Datapath: LDR Register Read

STEP 2: Read source operands from RF

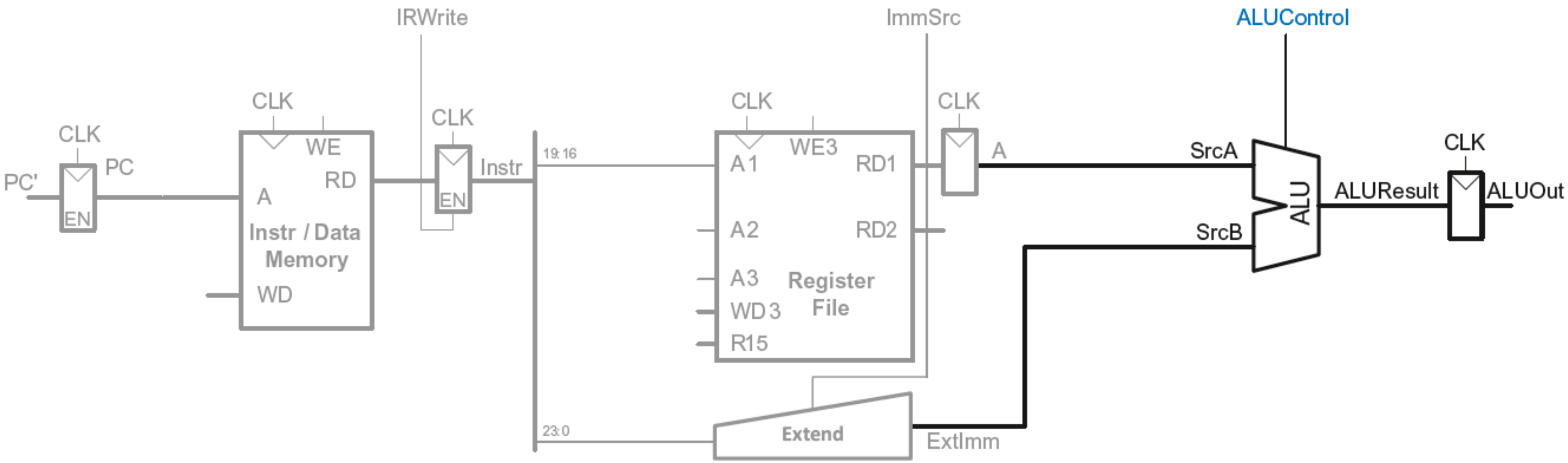


31:28	27:26	25:20						19:16	15:12	11:0
cond	01	1	1	1	0	0	L	Rn	Rd	imm12

`LDR Rd, [Rn, imm12]`

Multicycle Datapath: LDR Address

STEP 3: Compute the memory address

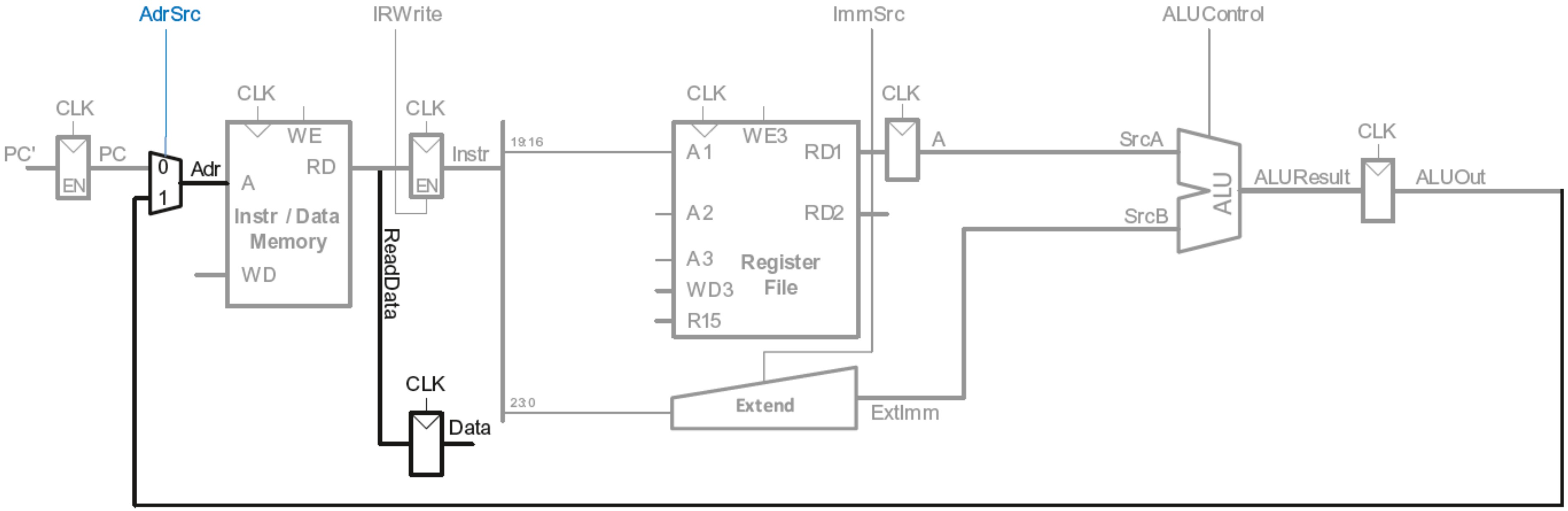


31:28	27:26	25:20						19:16	15:12	11:0
cond	01	1	1	1	0	0	L	Rn	Rd	imm12

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

Multicycle Datapath: LDR Memory Read

STEP 4: Read data from memory

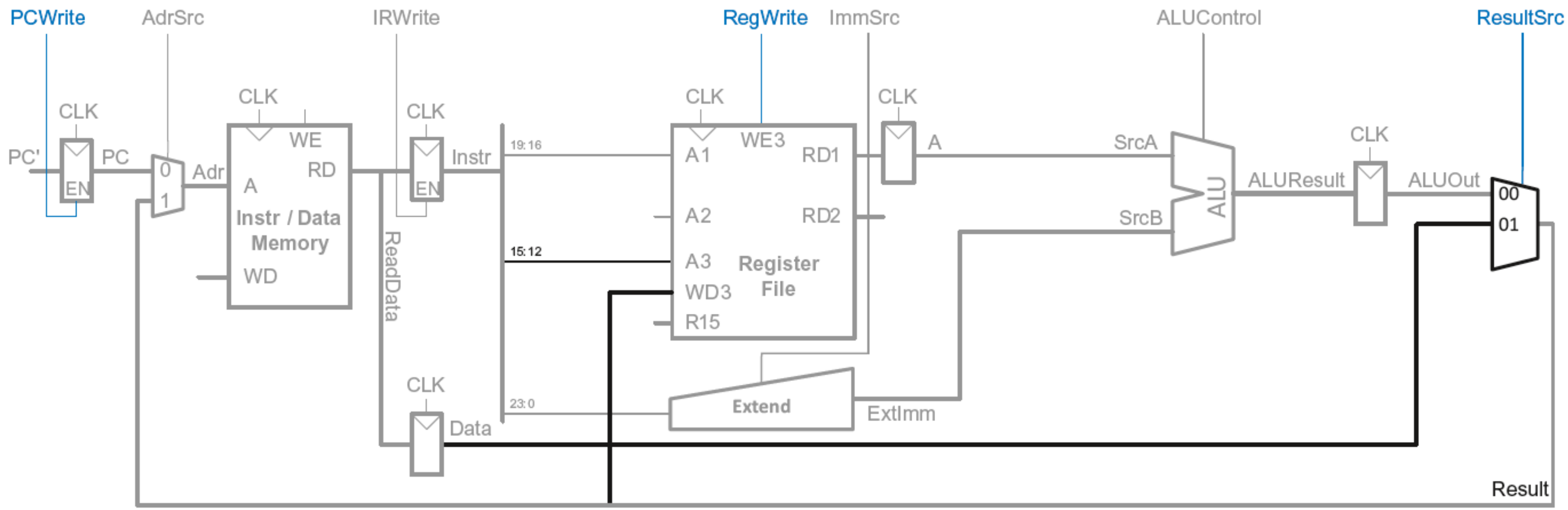


31:28	27:26	25:20						19:16	15:12	11:0
cond	01	1	1	1	0	0	L	Rn	Rd	imm12

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

Multicycle Datapath: LDR Write Register

STEP 5: Write data back to register file

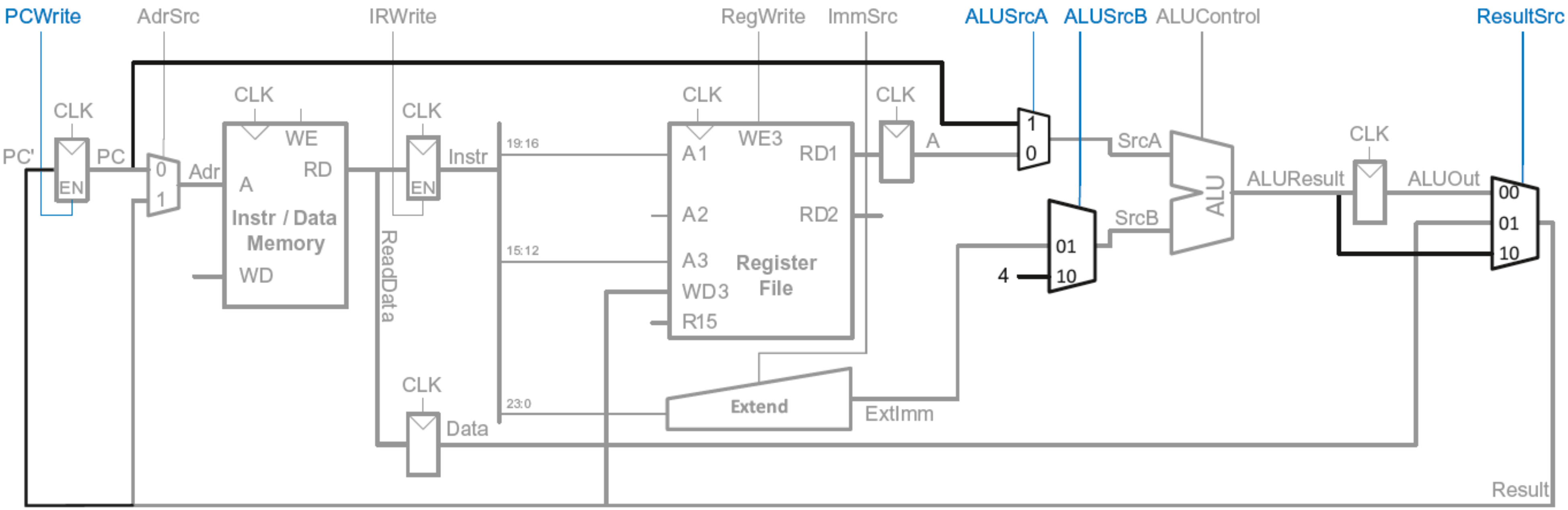


31:28	27:26	25:20						19:16	15:12	11:0
cond	01	1	1	1	0	0	L	Rn	Rd	imm12

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

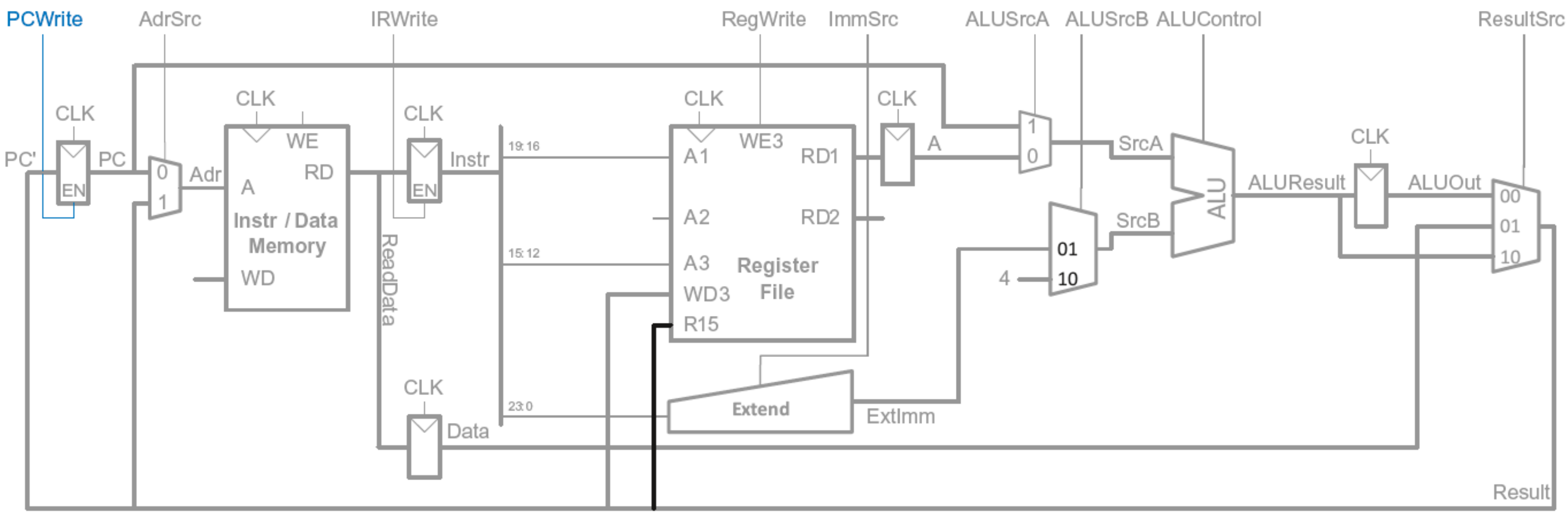
Multicycle Datapath: Increment PC

STEP 6: Increment PC



Multicycle Datapath: Access to PC

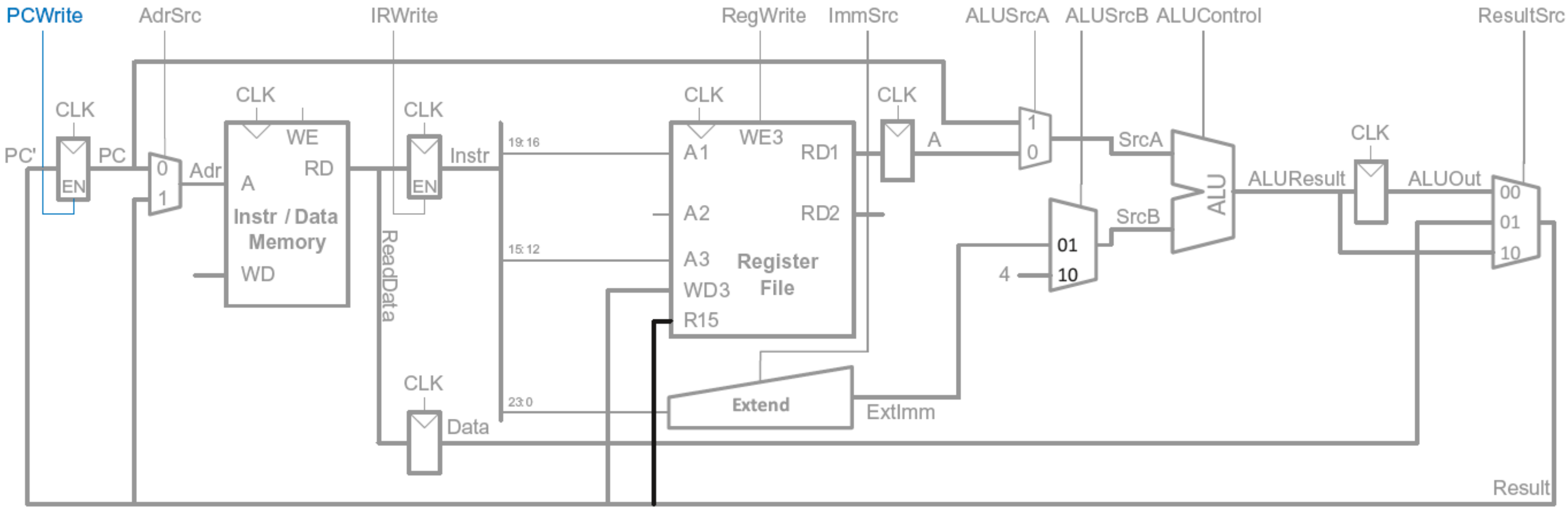
PC can be read/written by instruction



Multicycle Datapath: Access to PC

PC can be read/written by instruction

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



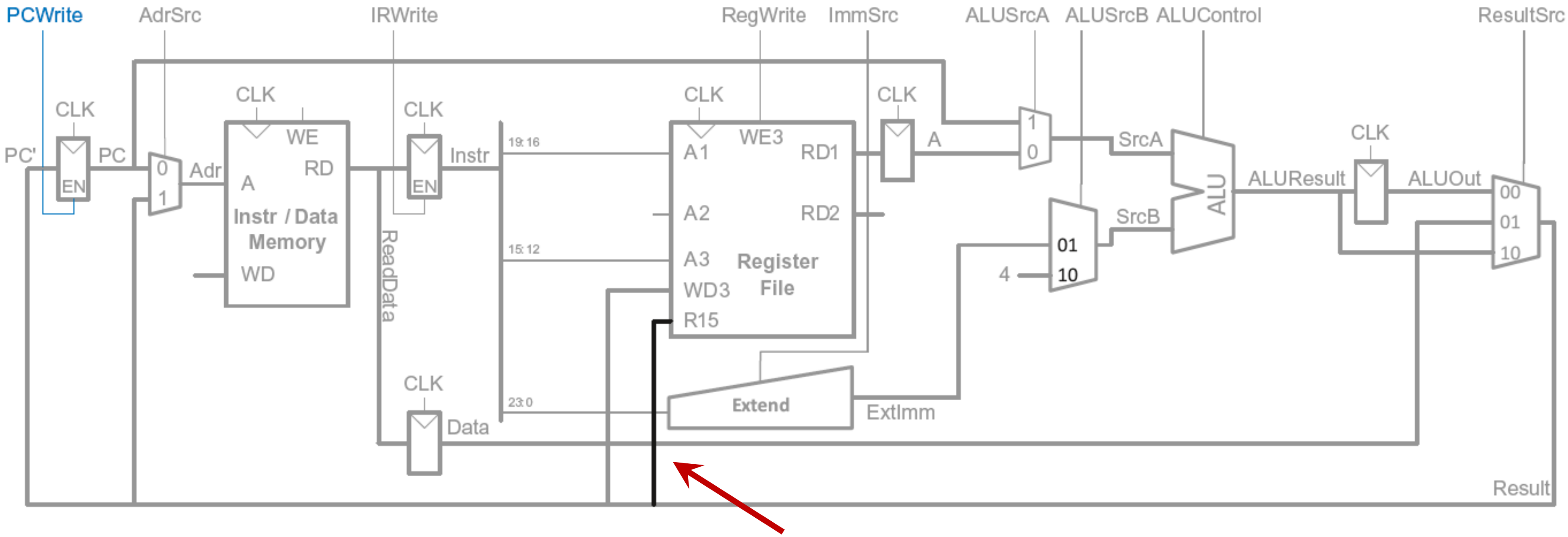
Multicycle Datapath: Read to PC (R15)

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)

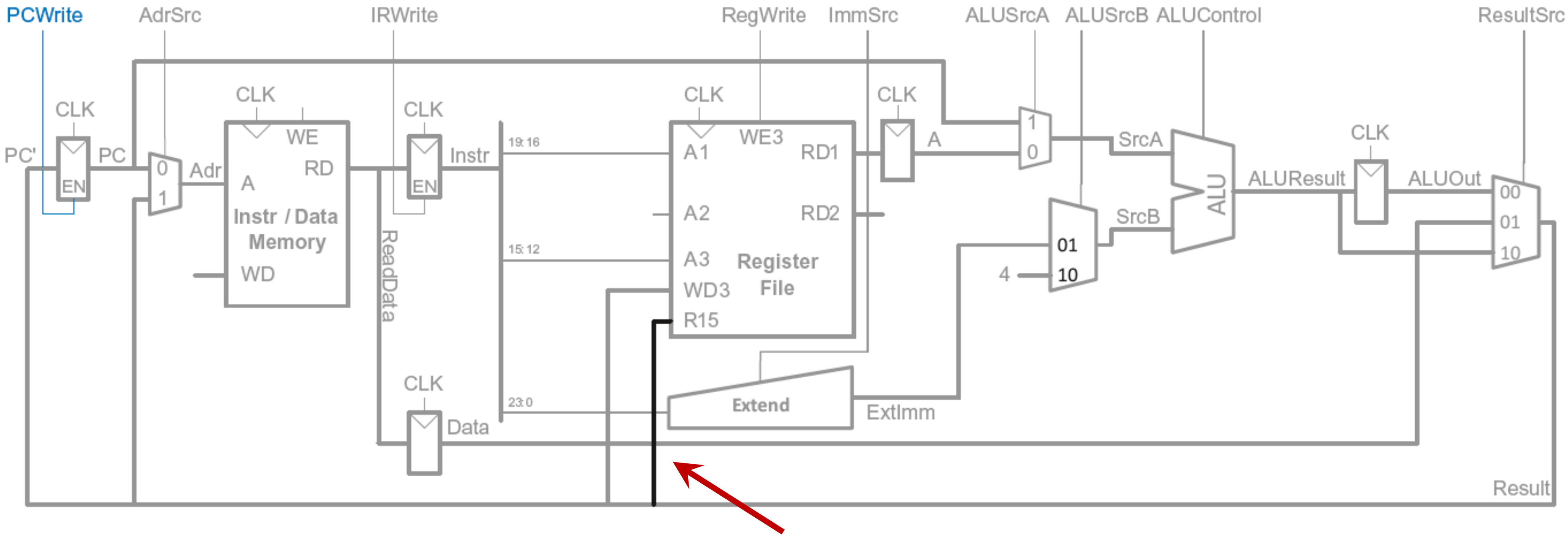
Example: `ADD R1, R15, R2`

- 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`

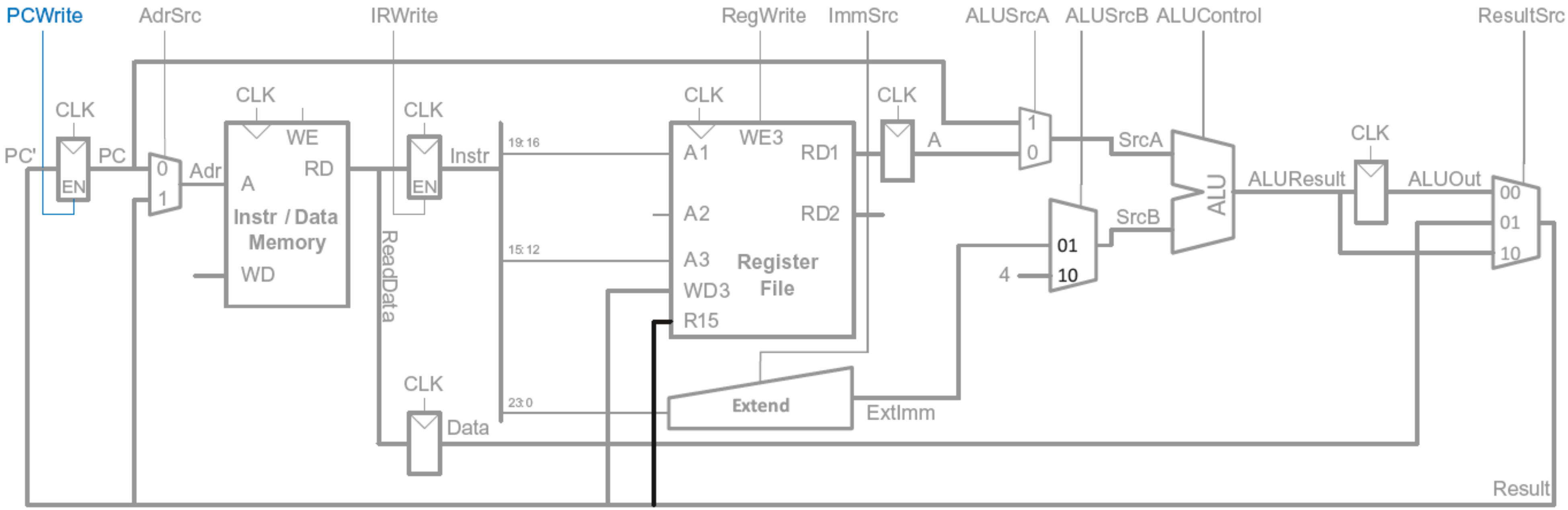
- R15 needs to be read as PC+8 from Register File (RF) in 2nd step
- So (also in 2nd step) PC + 8 is produced by ALU and routed to R15 input of RF



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)

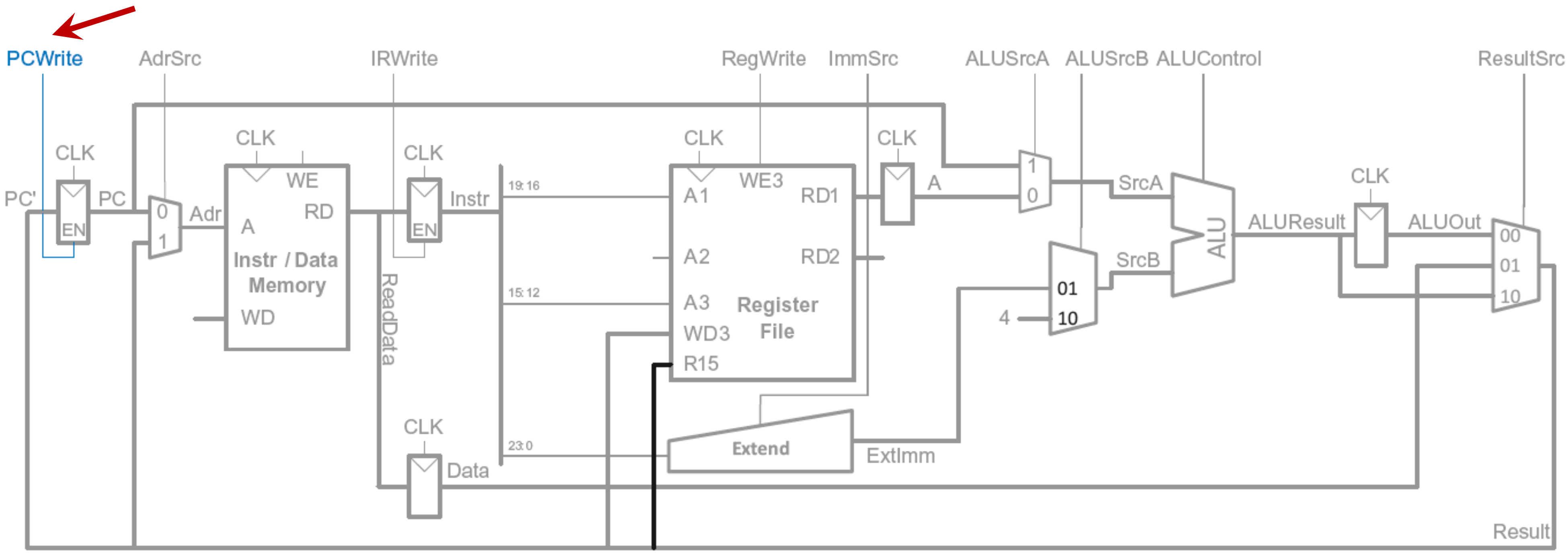
Example: `SUB` **`R15`**`, R8, R3`

- 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)

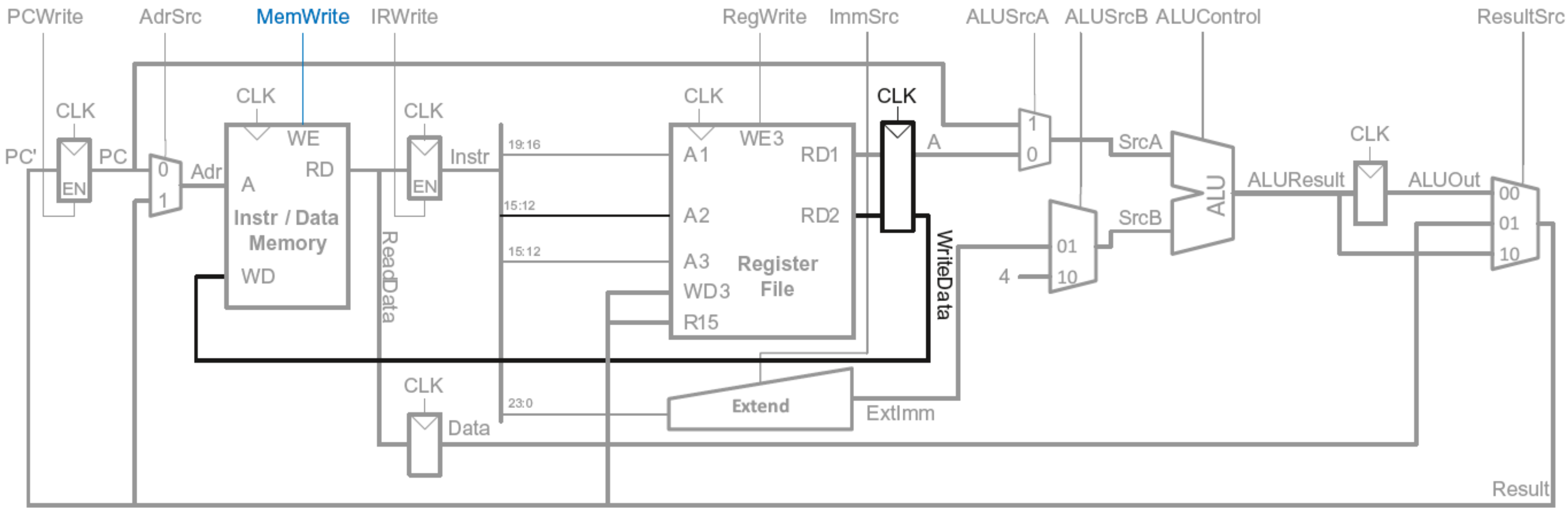
Example: `SUB **R15**, R8, R3`

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



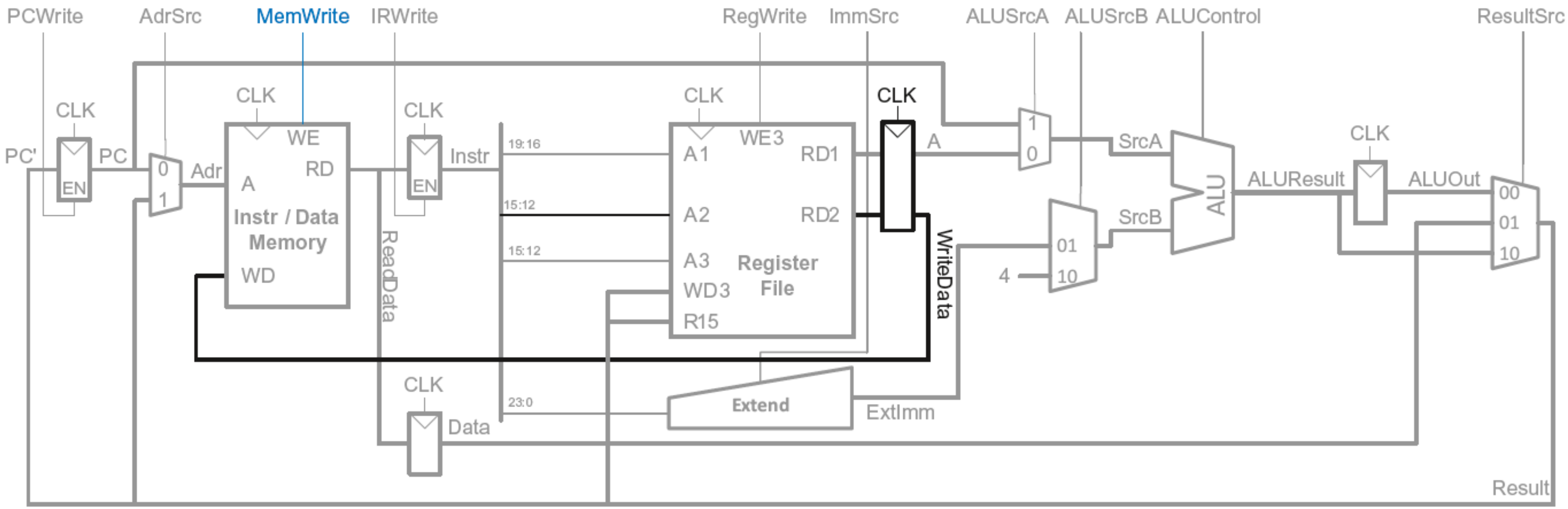
Multicycle Datapath: STR

Write data in `Rn` to memory



Multicycle Datapath: Data-Processing

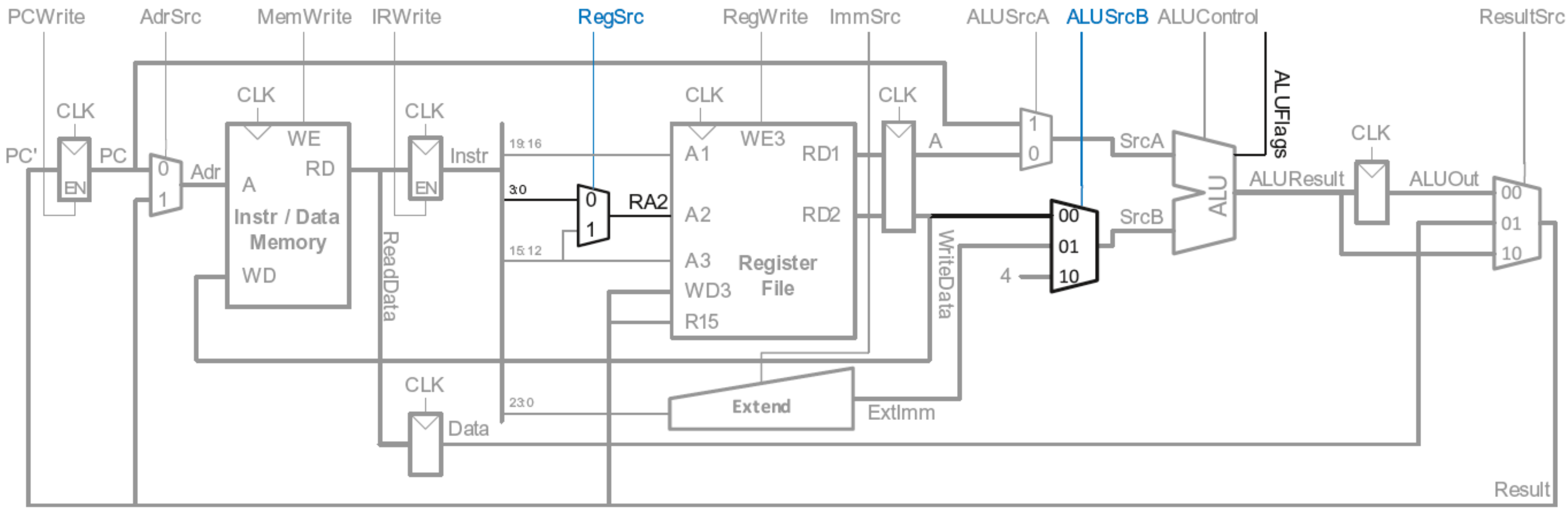
With immediate addressing (i.e., an immediate *Src2*), no additional changes needed for datapath



Multicycle Datapath: Data-Processing

With register addressing (register *Src2*):

Read from Rn and Rm

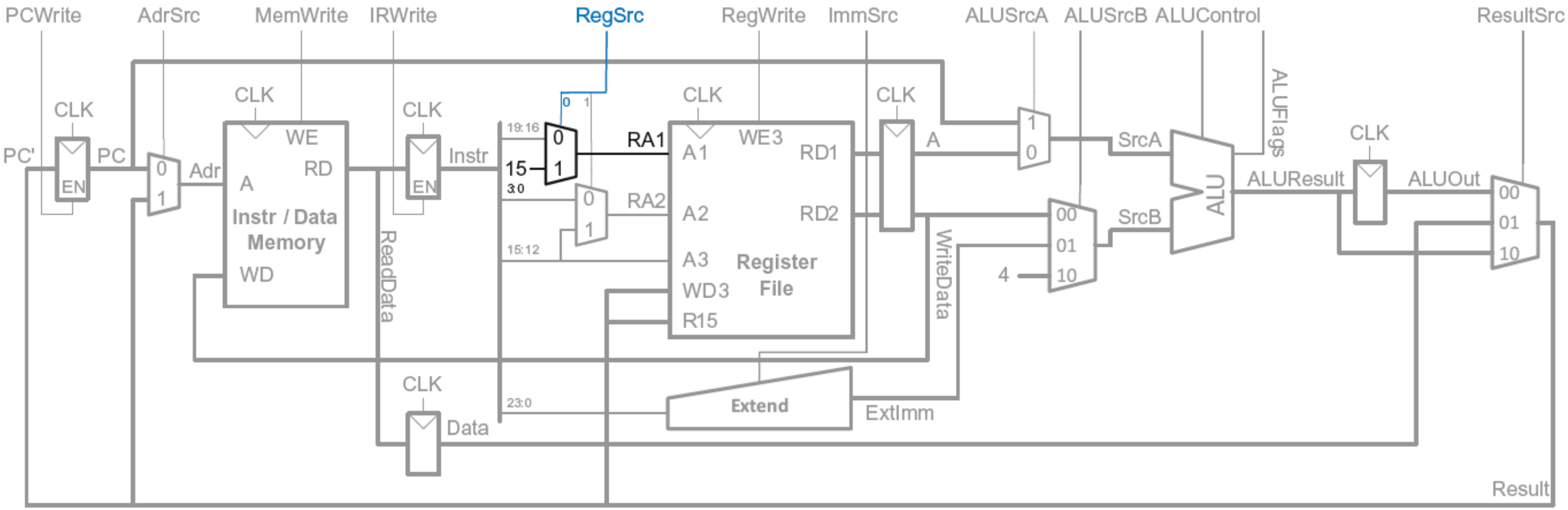


Multicycle Datapath: B

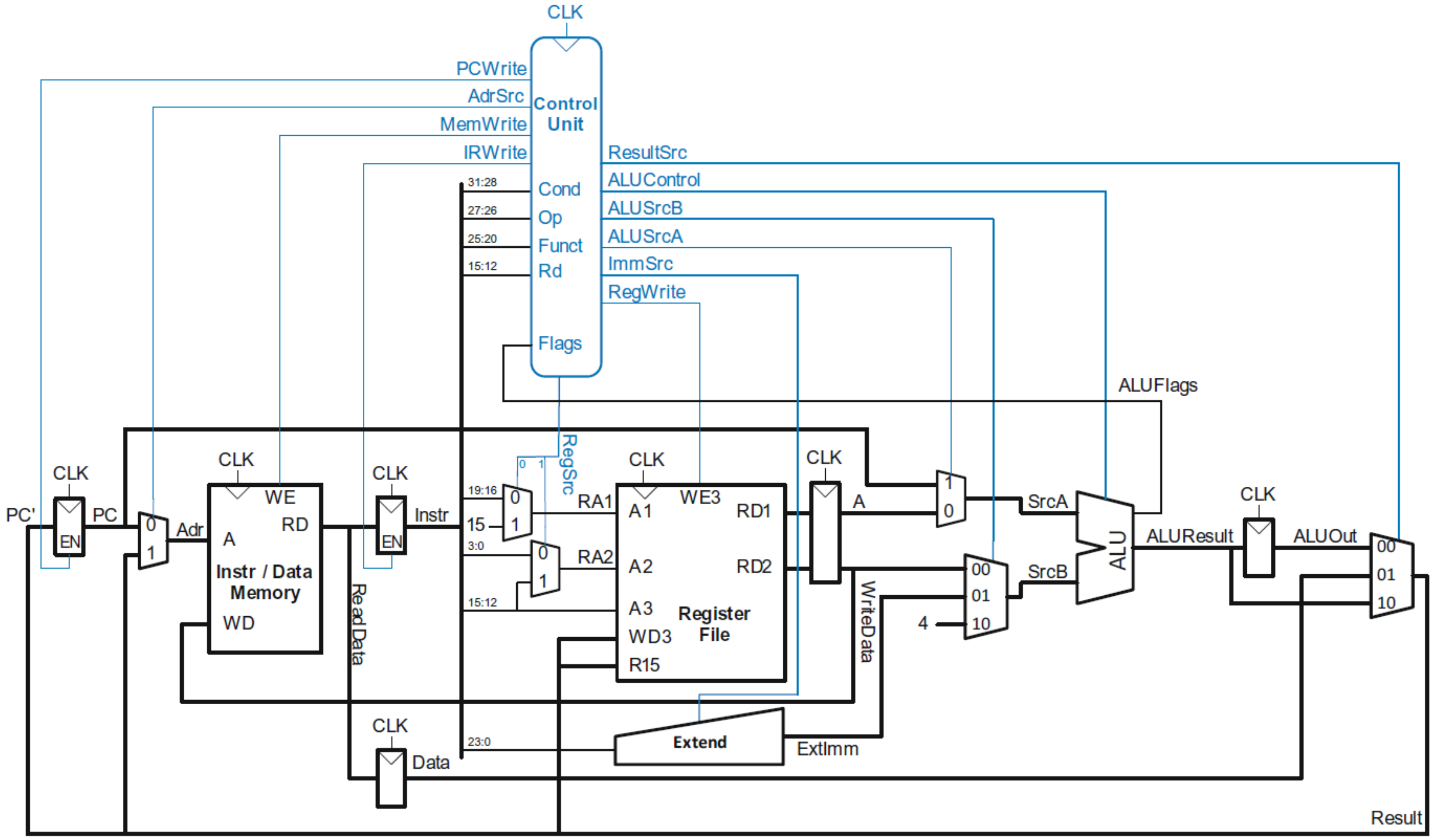
Calculate branch target address:

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

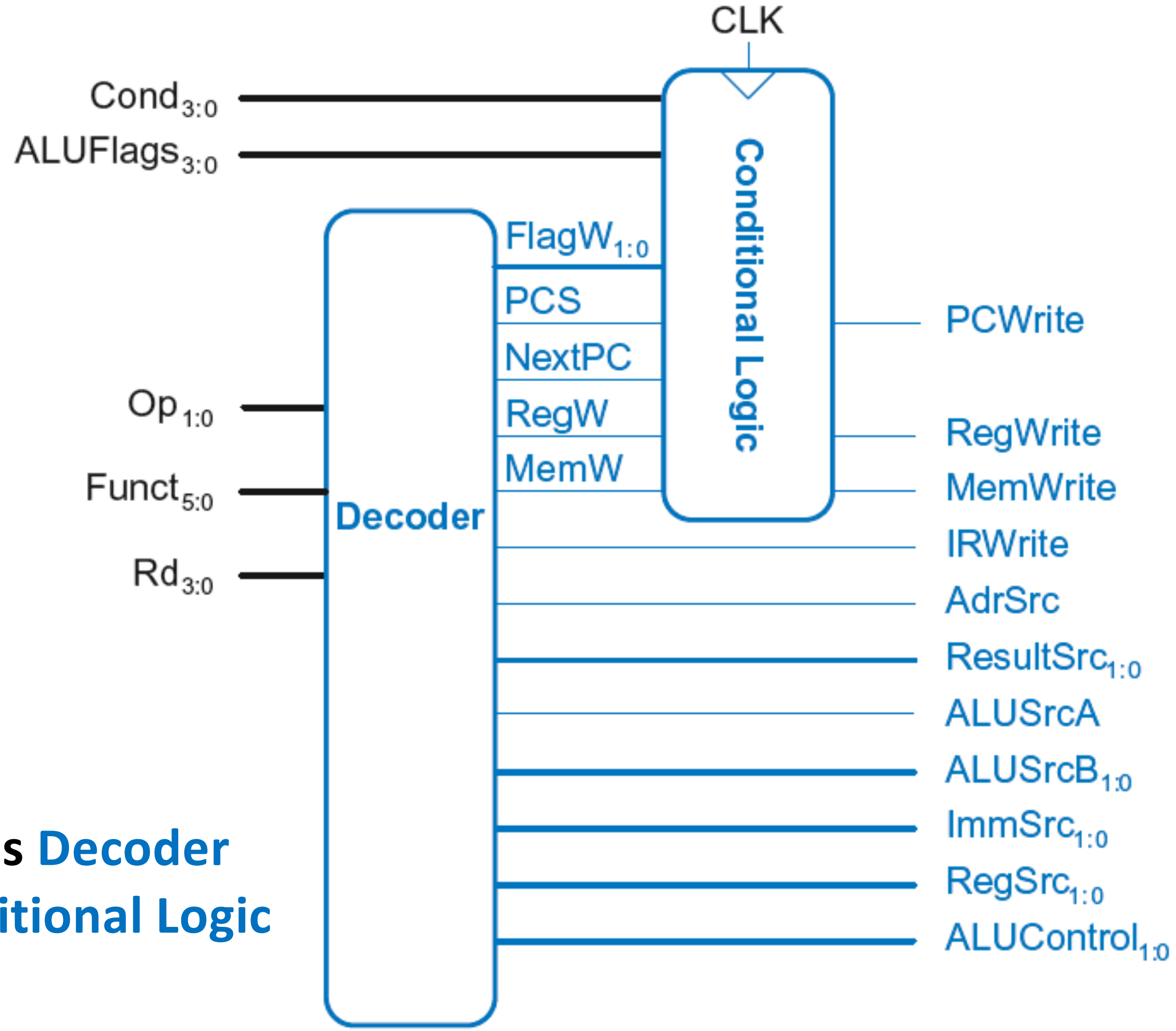
ExtImm = *Imm24* << 2 and sign-extended



Multicycle ARM Processor

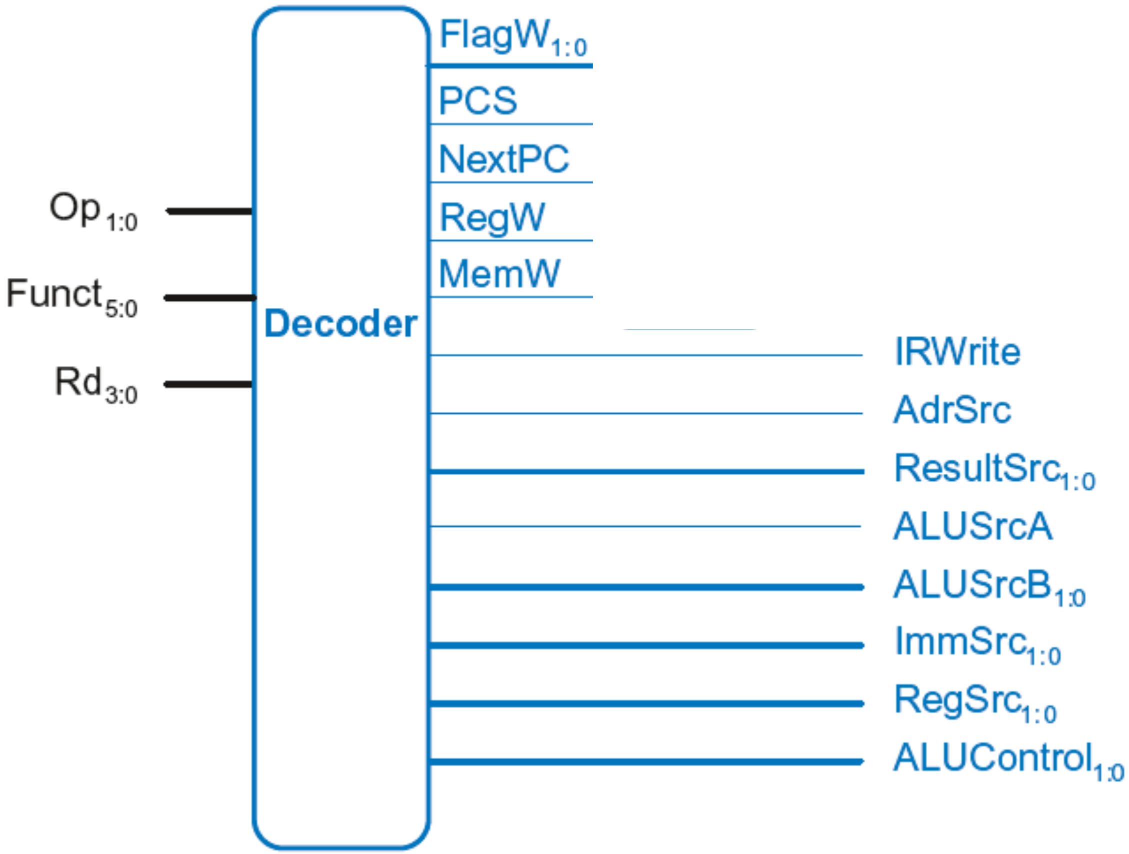


Multicycle Control

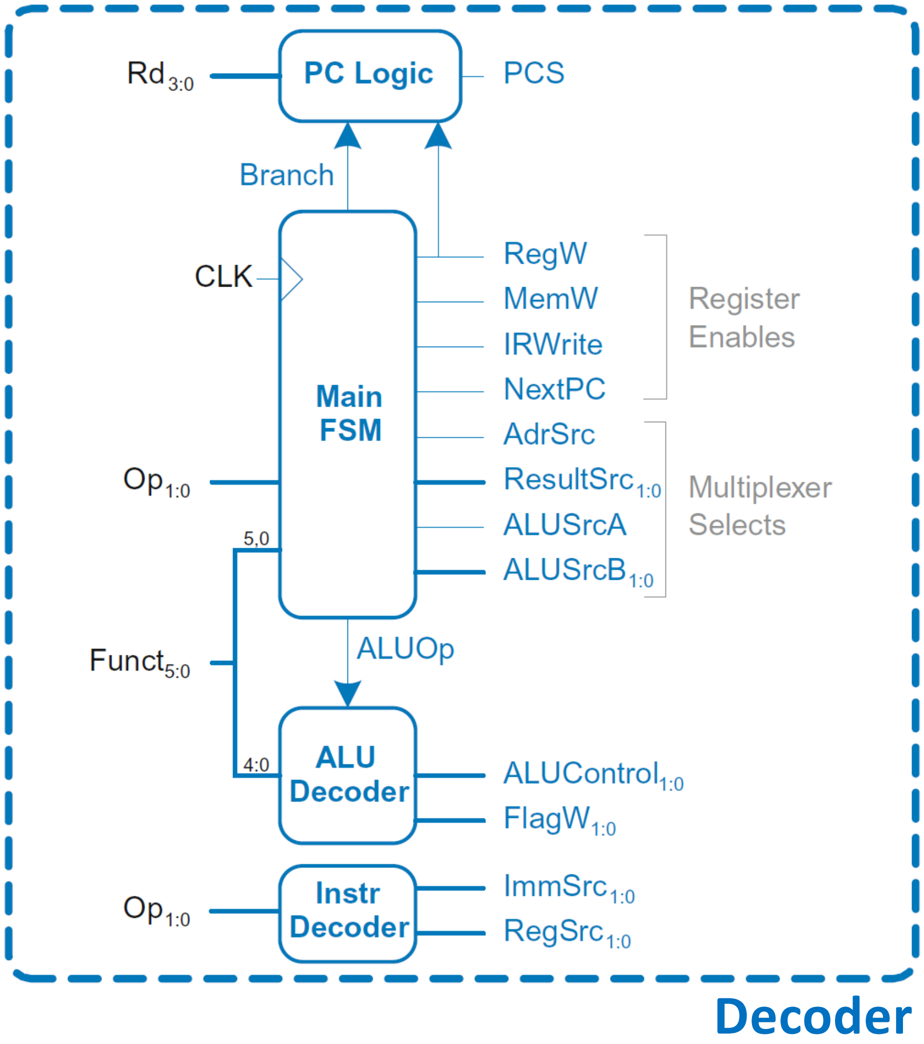


- **First, discuss Decoder**
- **Then, Conditional Logic**

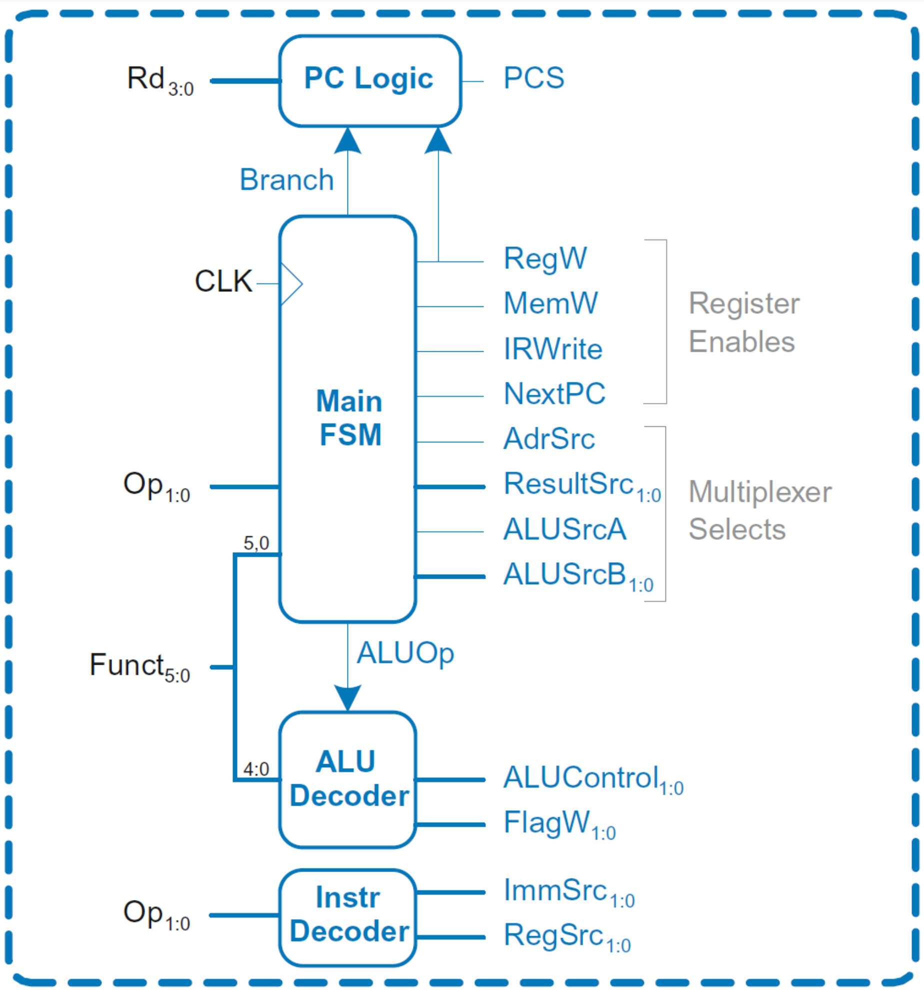
Multicycle Control: Decoder



Multicycle Control: Decoder

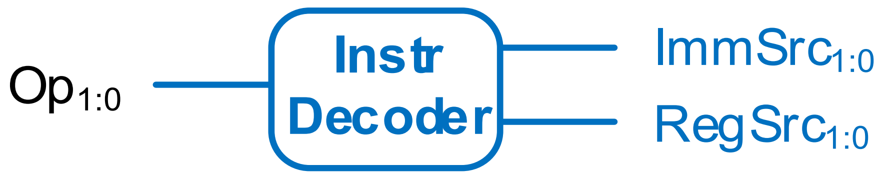


Multicycle Control: Decoder



ALU Decoder and PC Logic same as single-cycle

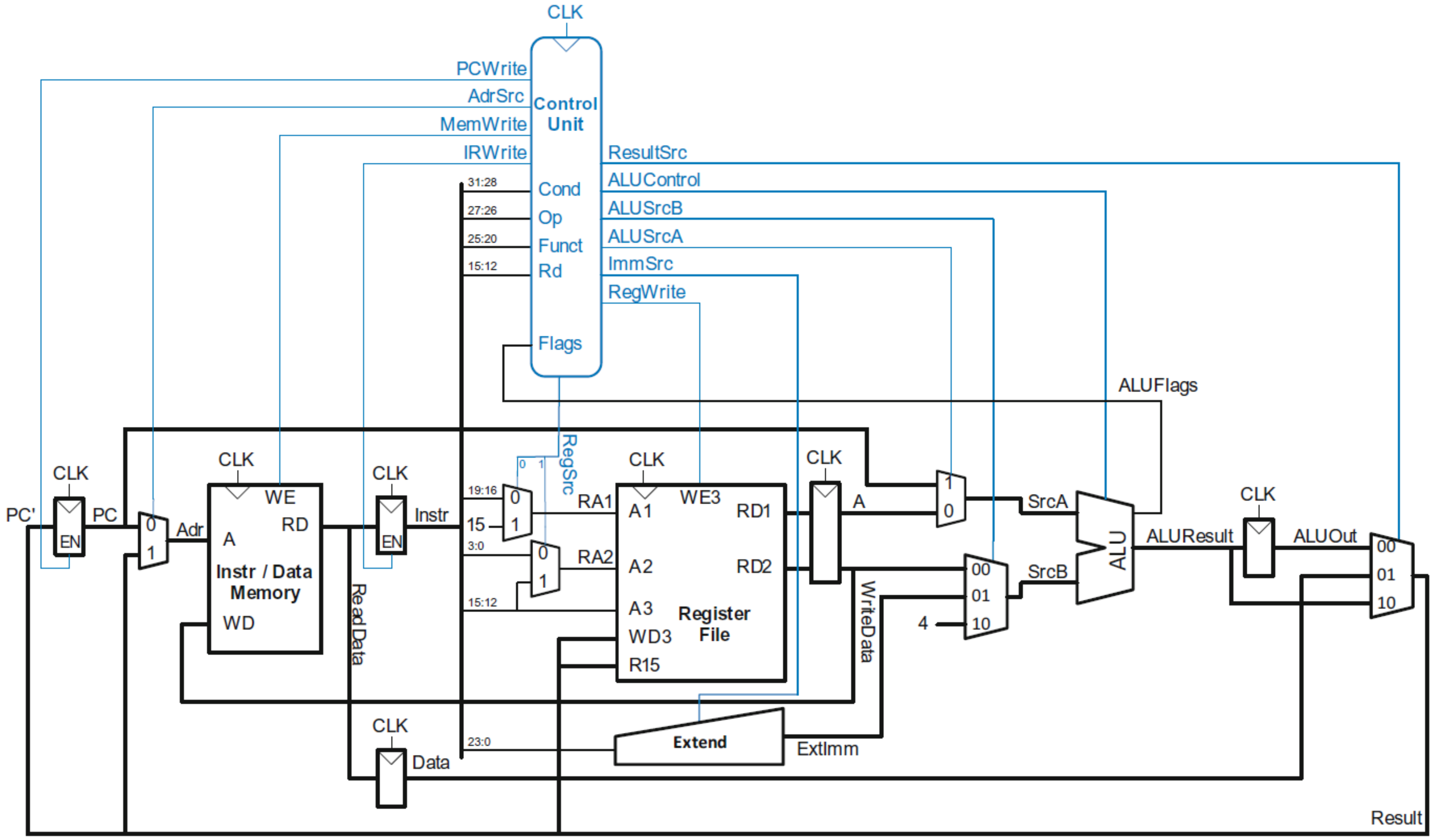
Multicycle Control: Instr Decoder



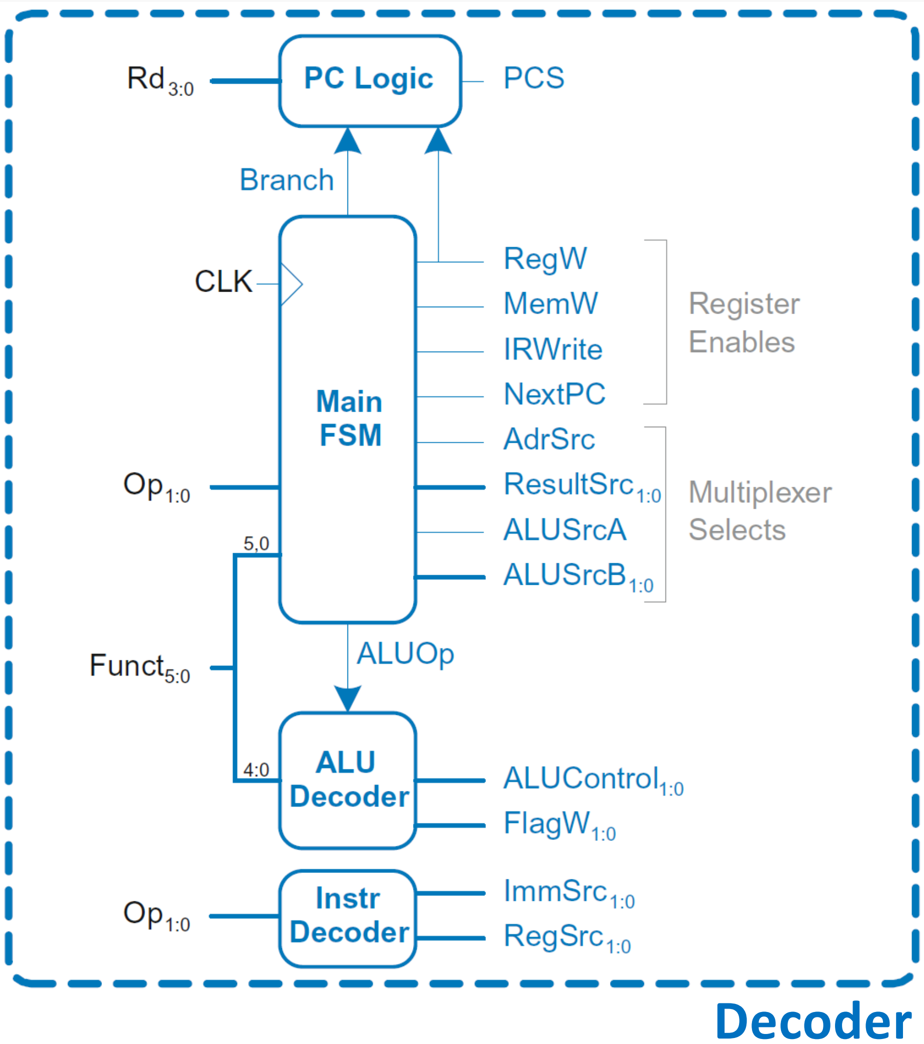
$$RegSrc_0 = (Op == 10_2)$$
$$RegSrc_1 = (Op == 01_2)$$
$$ImmSrc_{1:0} = Op$$

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

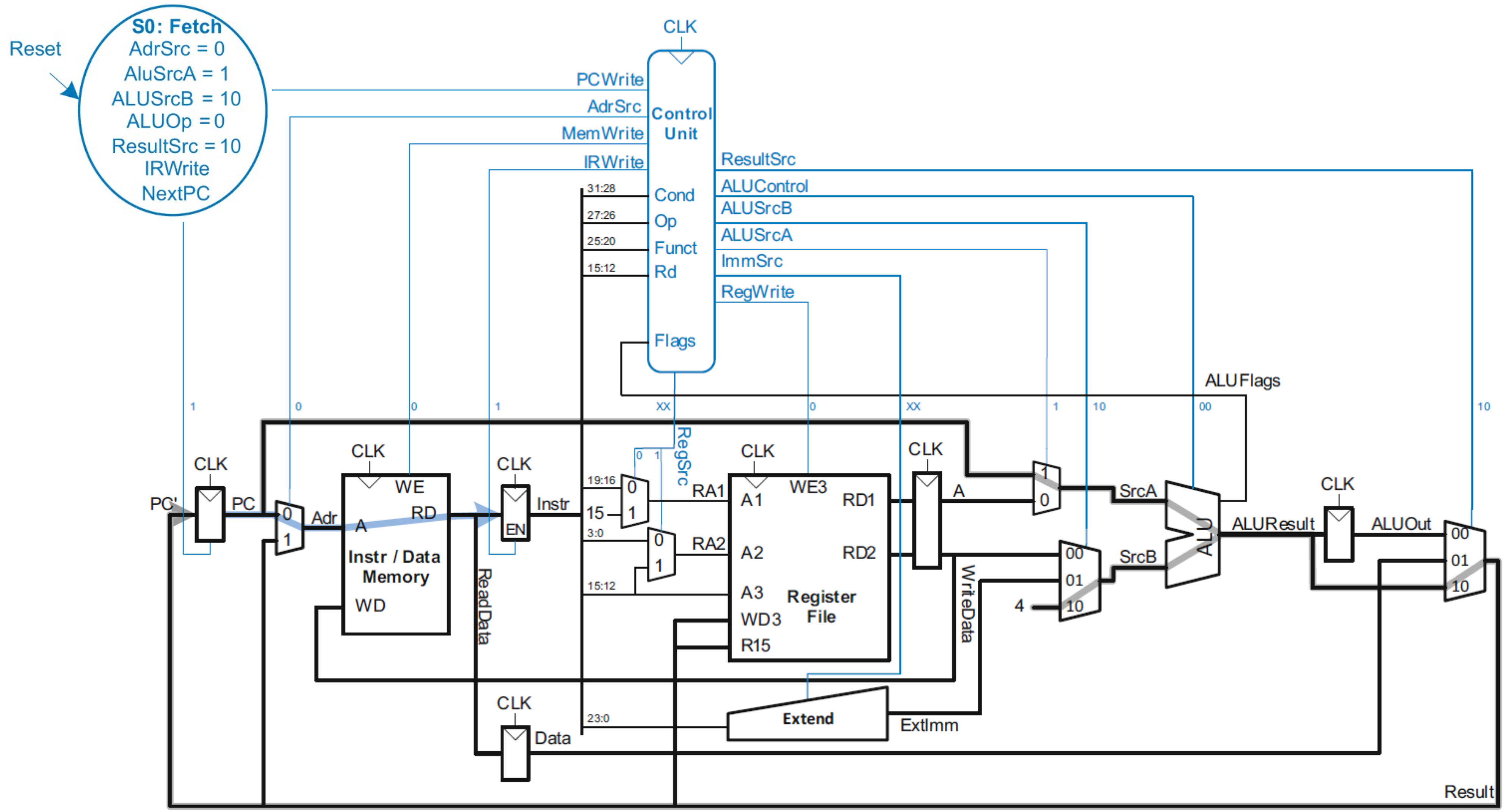
Multicycle ARM Processor



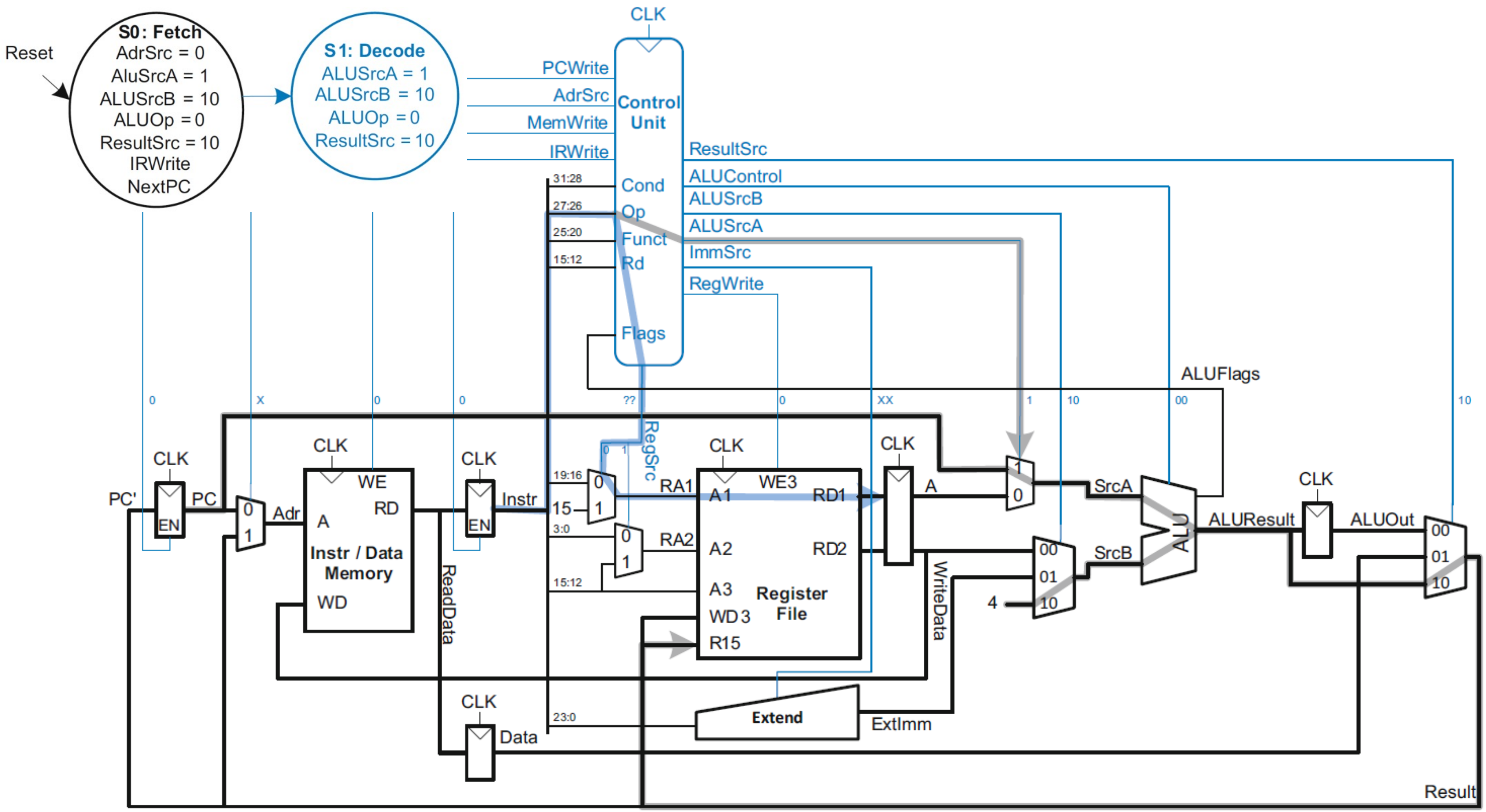
Multicycle Control: Main FSM



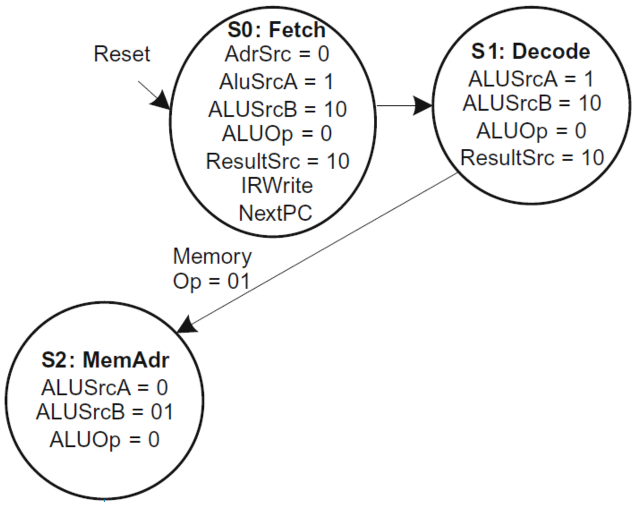
Main Controller FSM: Fetch

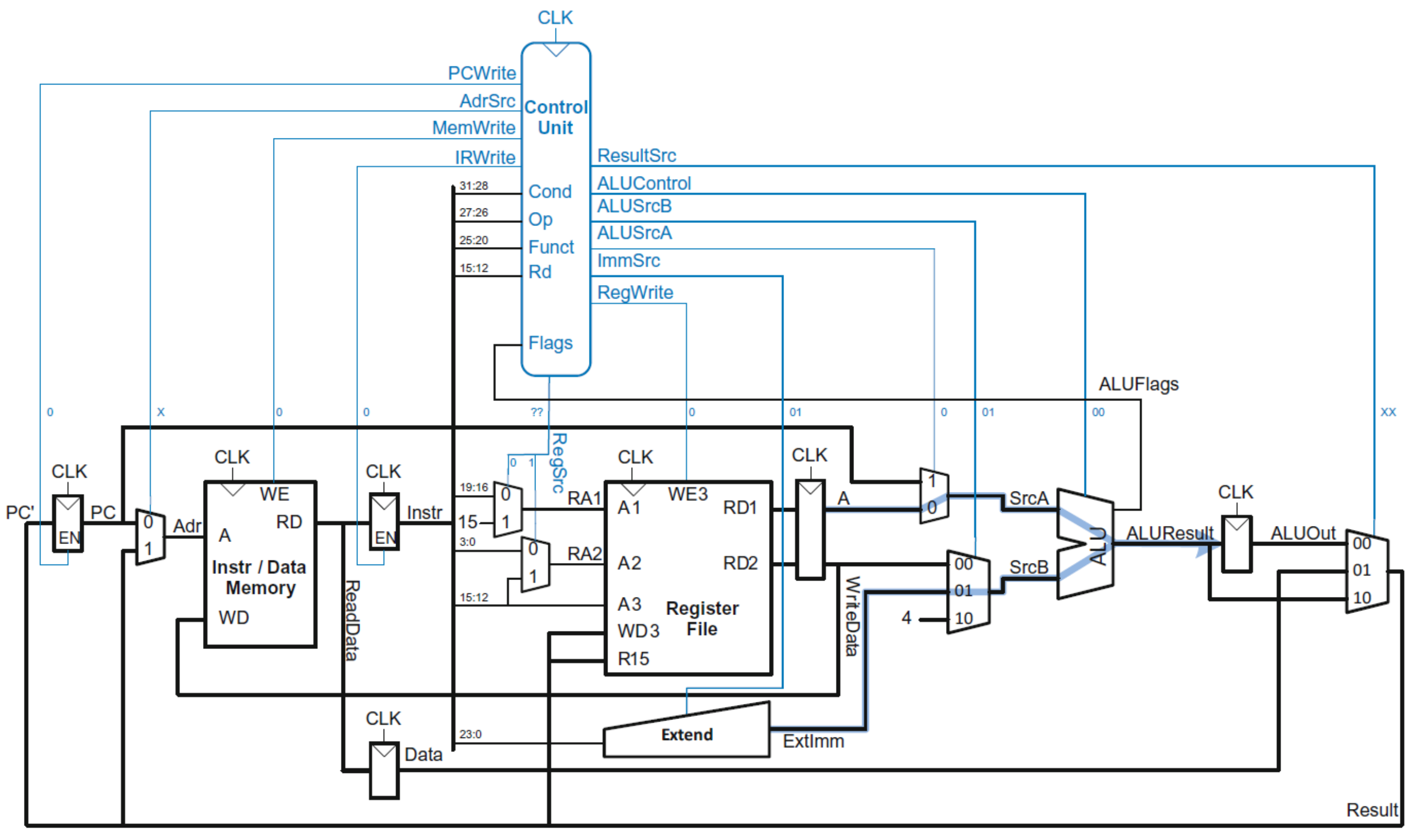


Main Controller FSM: Decode

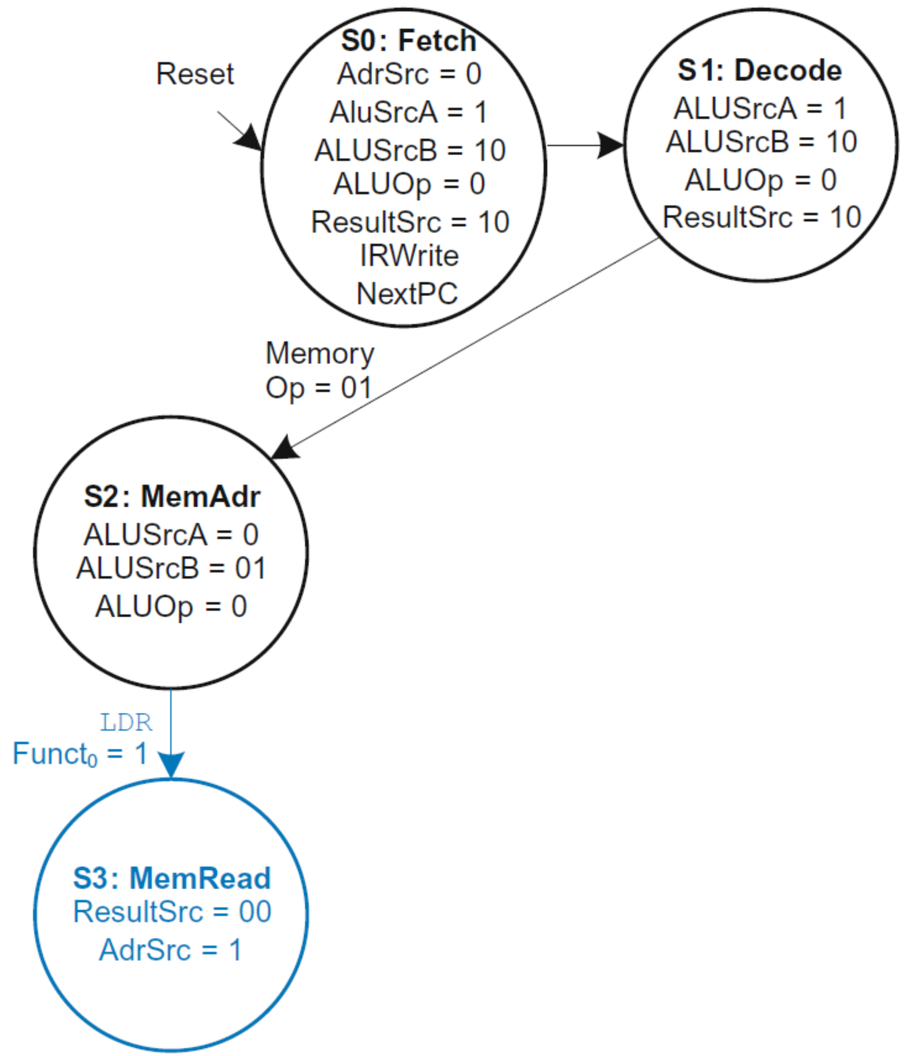


Main Controller FSM: Address

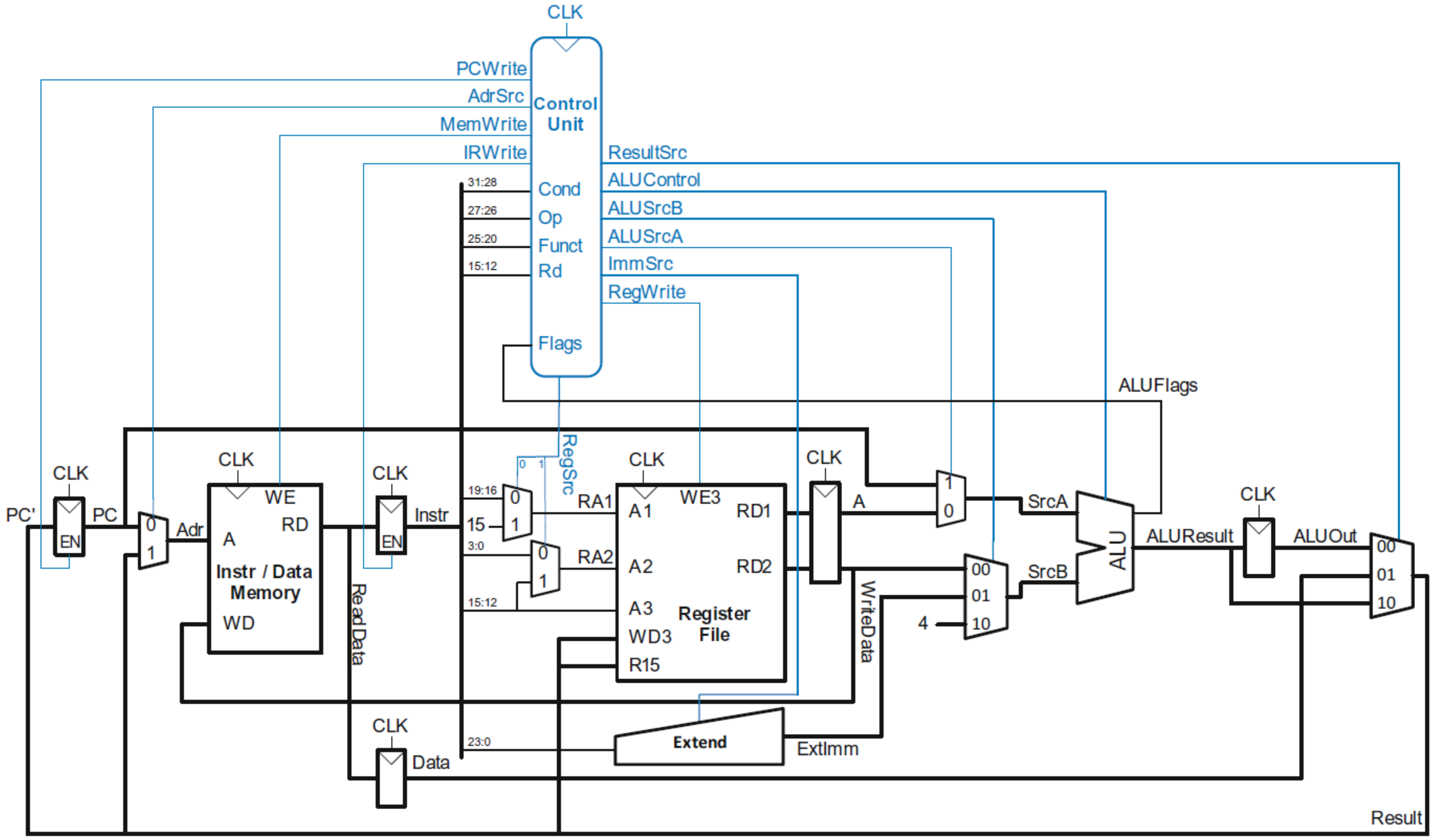




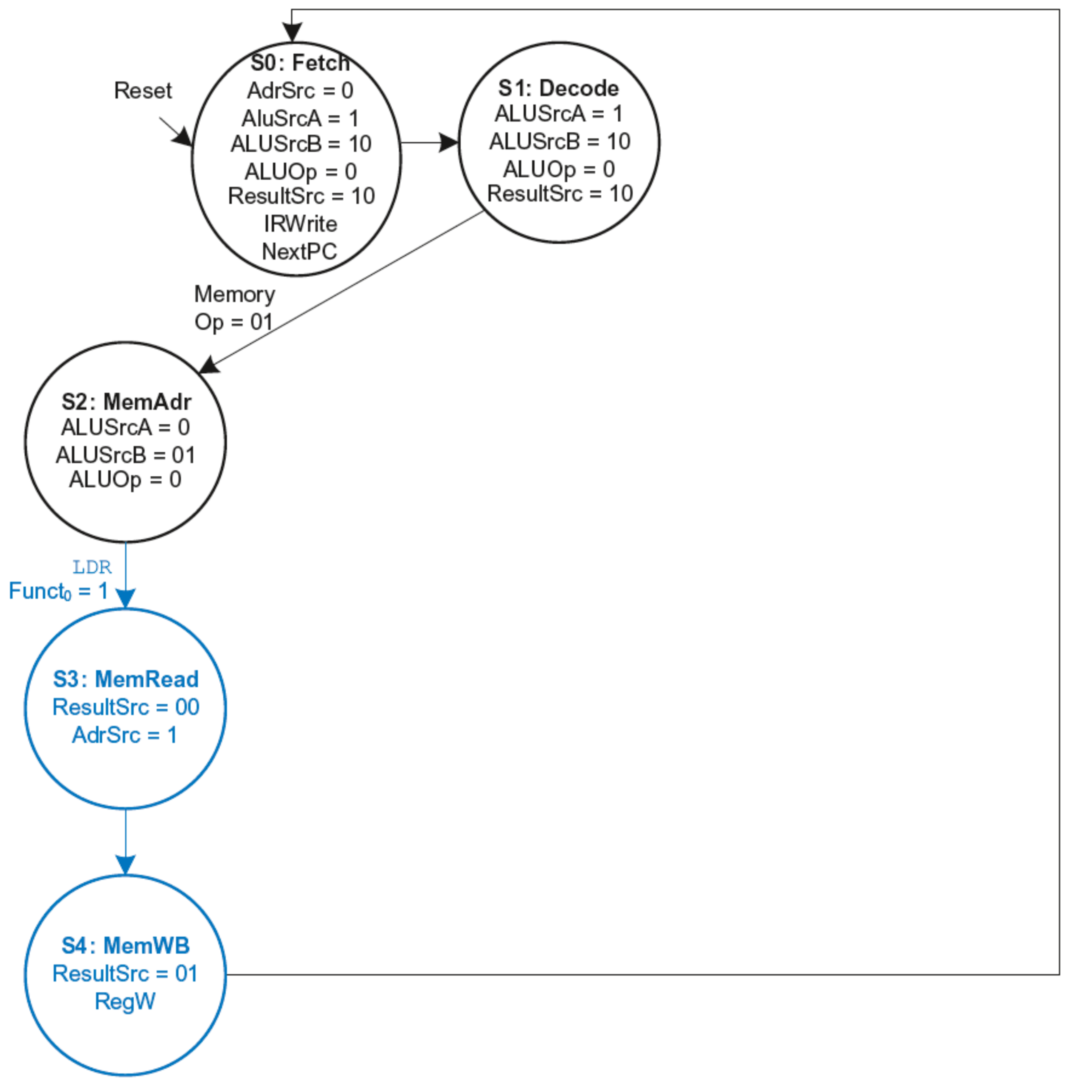
Main Controller FSM: Read Memory



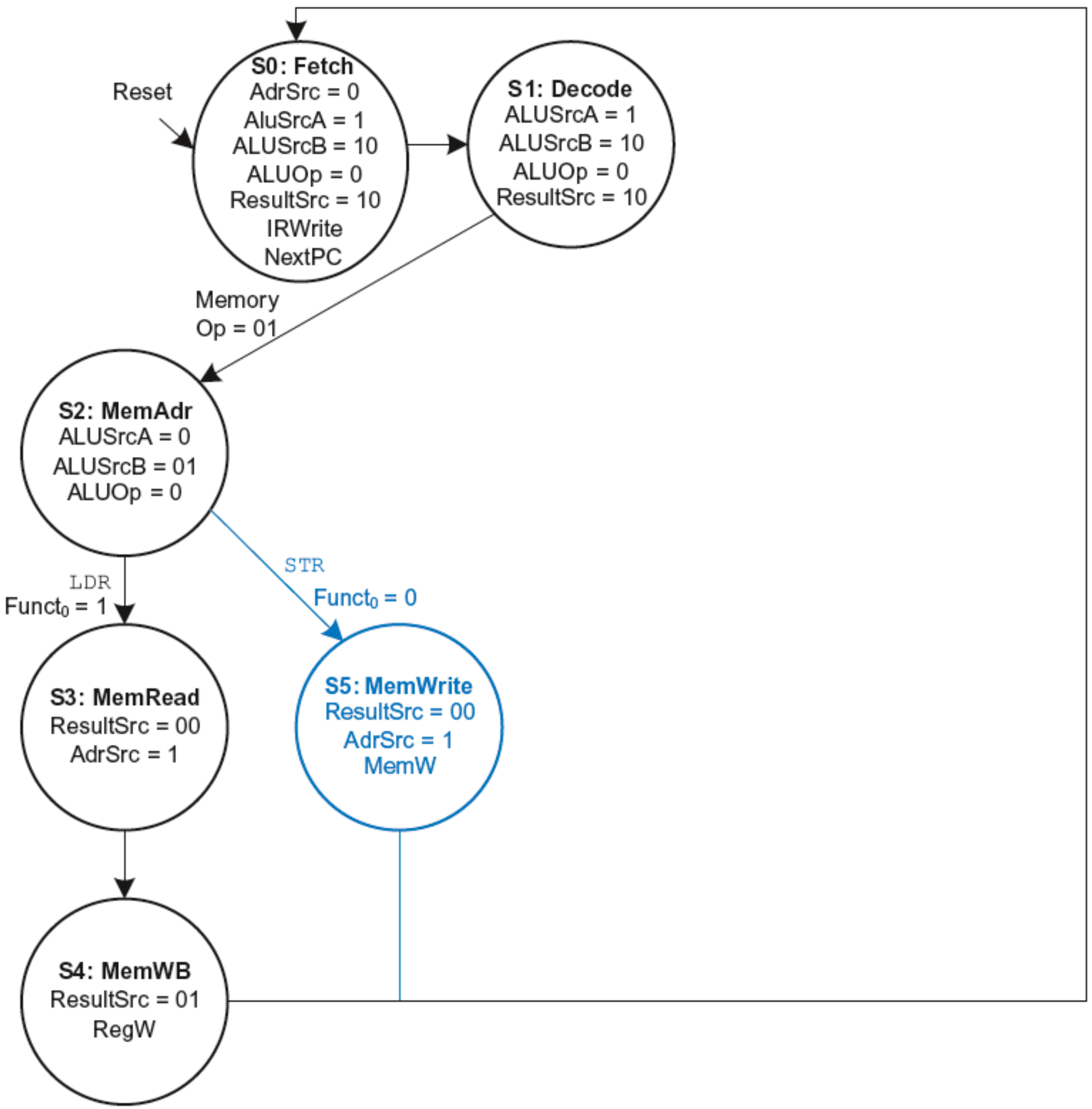
Multicycle ARM Processor



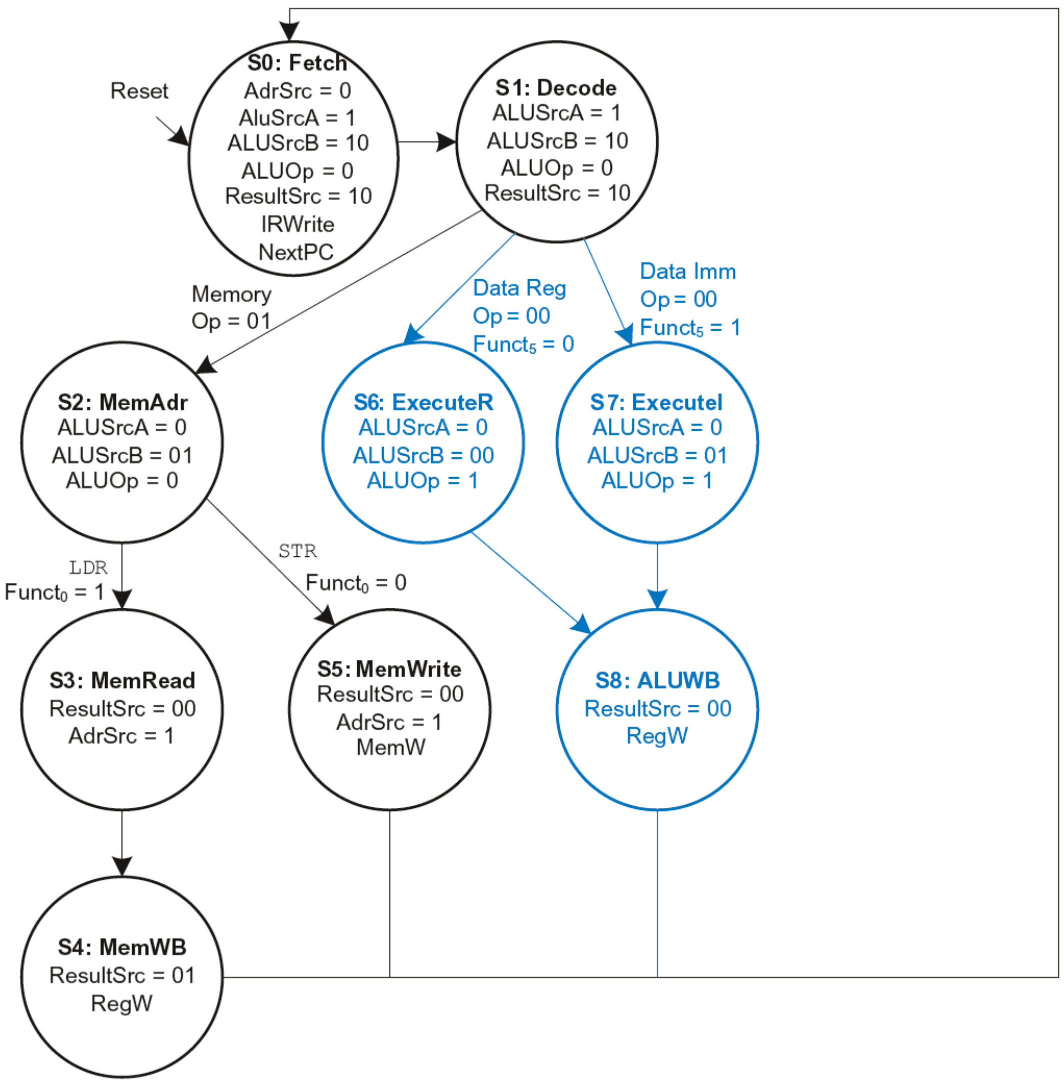
Main Controller FSM: LDR



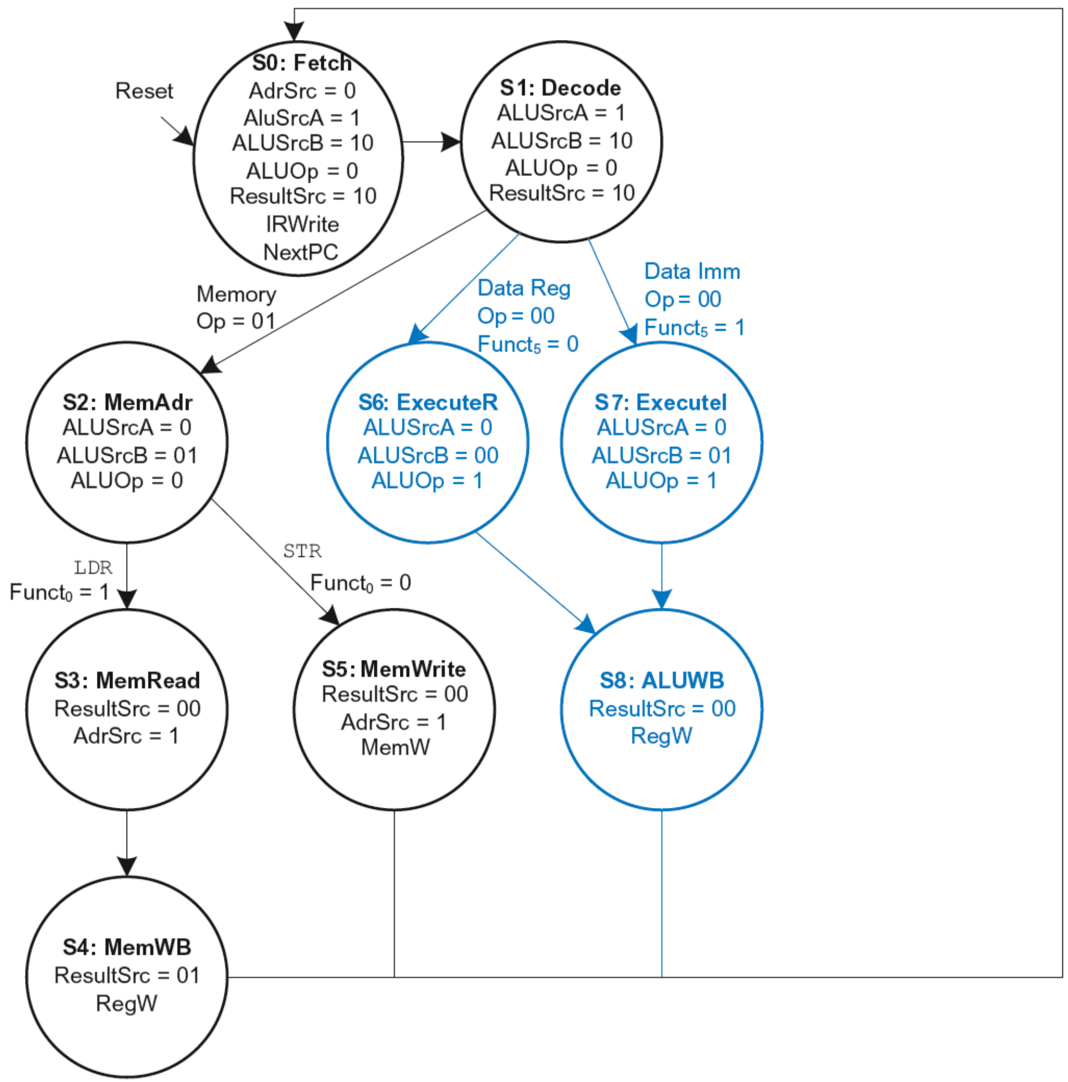
Main Controller FSM: STR



Main Controller FSM: Data-Processing

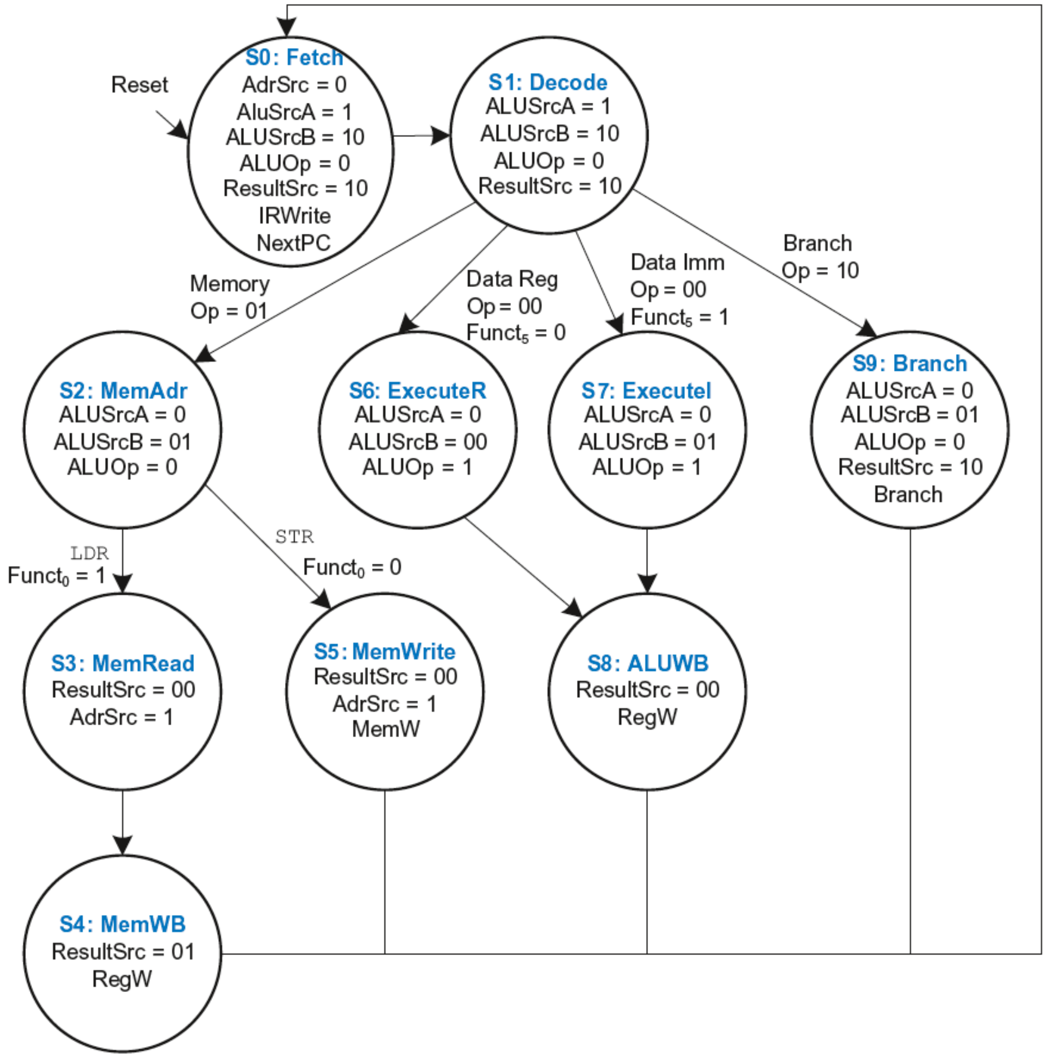


Main Controller FSM: Data-Processing

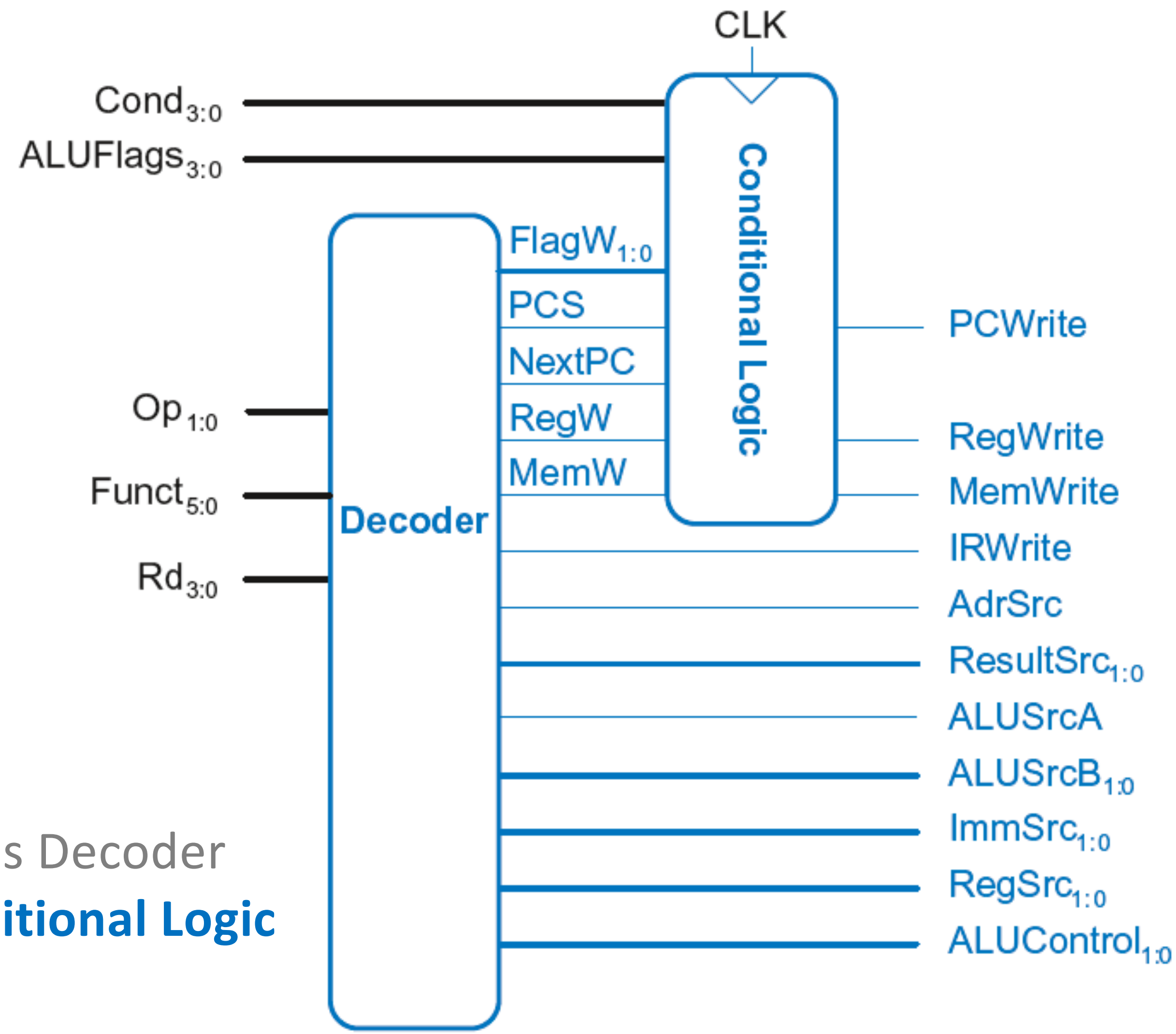


Main Controller FSM

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



Main Controller



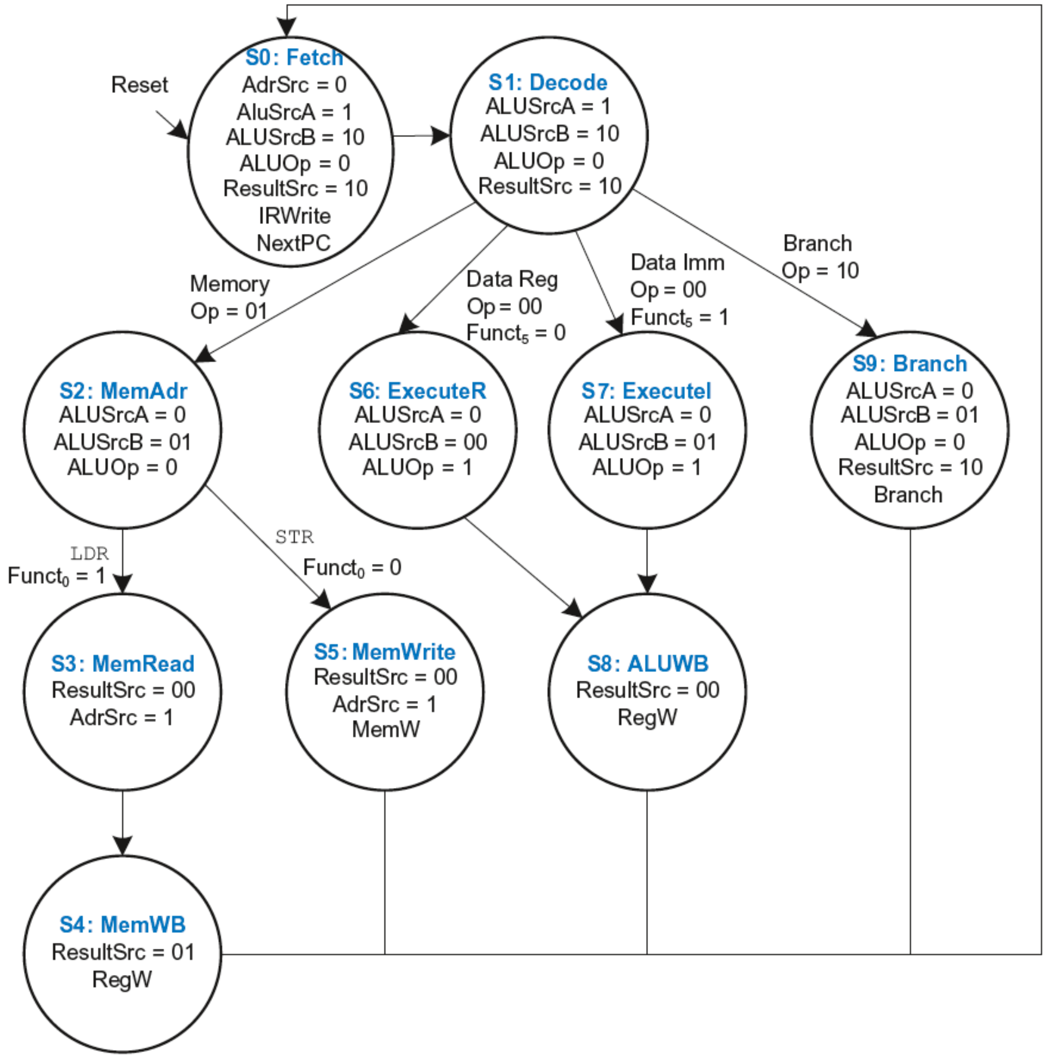
- First, discuss Decoder
- **Then, Conditional Logic**

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



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 suite:
  - 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

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

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	$t_{RF\mathrm{read}}$	100
Register file setup	$t_{RF\mathrm{setup}}$	60

$\boldsymbol{T_{c2}} = ?$

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

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

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

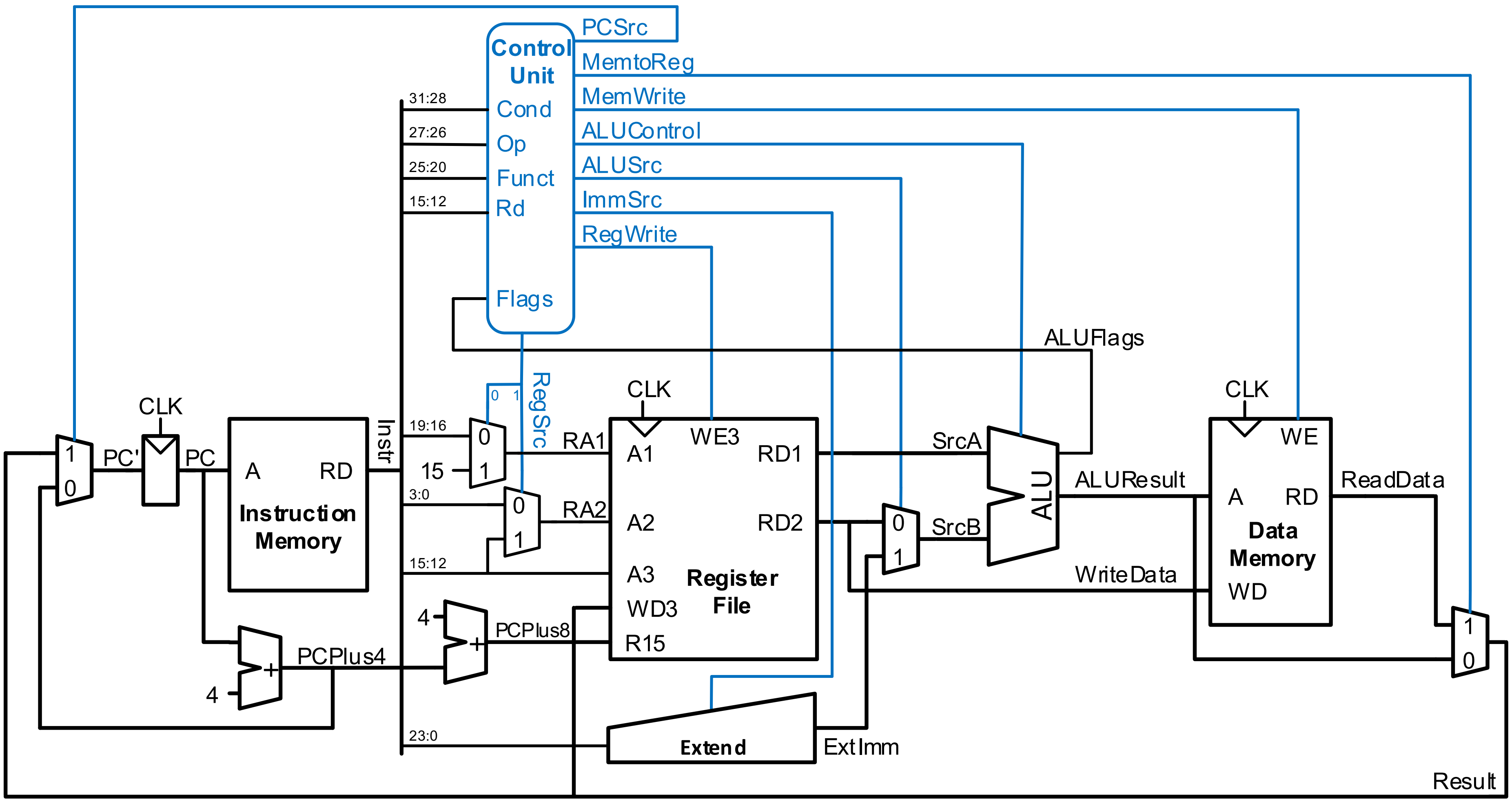
Execution Time = (# instructions) × CPI × T_c

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

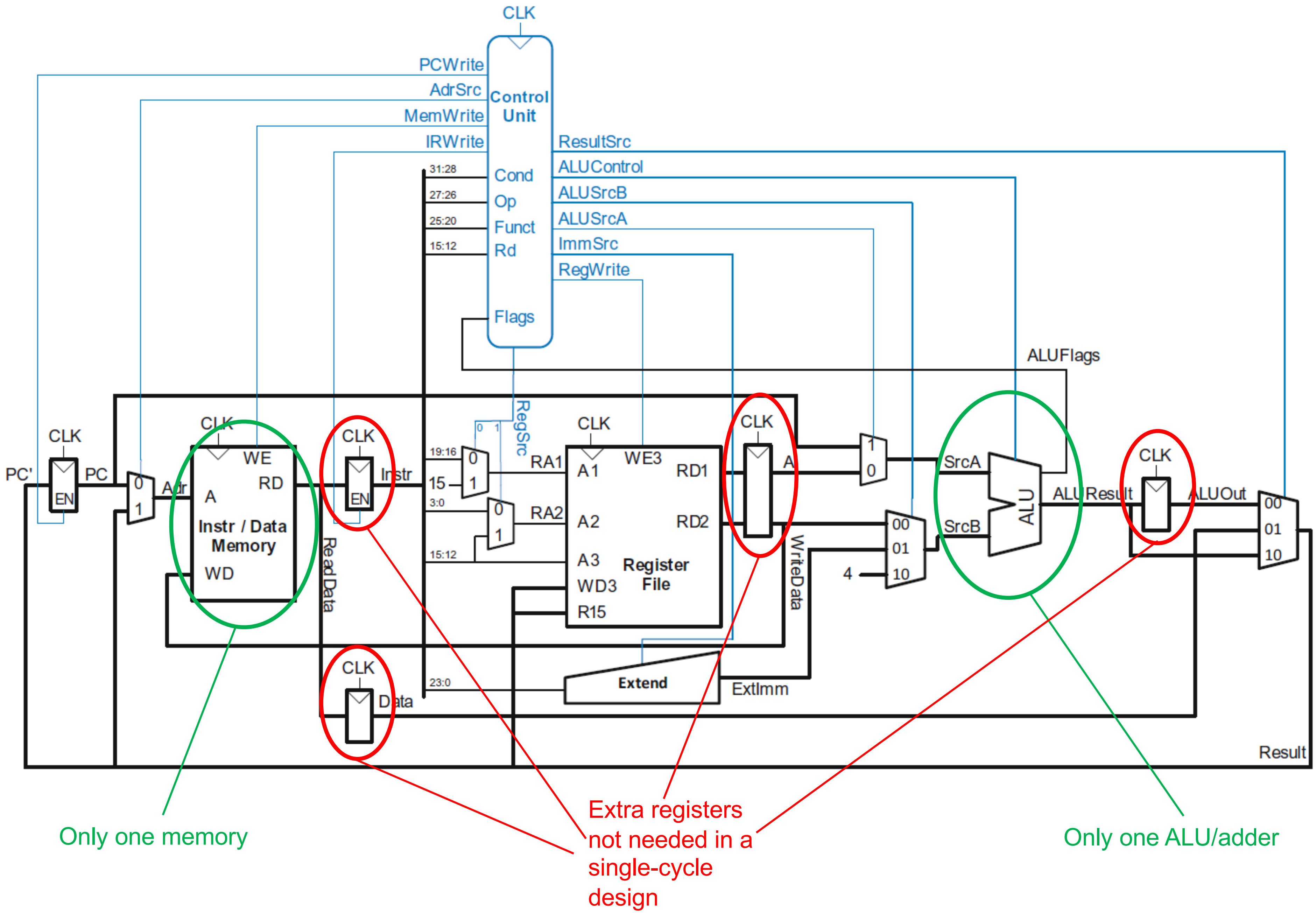
= 140 seconds

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

Review: Single-Cycle Processor



Review: Multicycle ARM Processor



Quiz: Multicycle ARM Processor

1. Why do we need each of the non-architectural registers?

2. Explain why do we need the path colored red (pick a path)?

3. Explain the ALUResult bypassing the 3-input mux?

4. Why are there two muxes in front of RF?

5. What is the purpose of PCWrite and IRWrite and MemWrite and RegWrite?

6. What is the purpose of AdrSrc (signal) and Adr (mux)?

7. Why do we store the instruction/data in a non-architectural register?

8. What if we don’t have a register at the output of RF?

9. Analyze the critical path of the multicycle processor (page 424 of book)
   - Hint: PC update, Memory read

Microprogramming

Microprogrammed Control

- Multi-cycle microarchitecture enables a key new abstraction called microprogramming

- Hardwired control
  - Physically connect the control lines to the actual machine instructions
  - Instructions are divided into fields, and bits in the field are connected to input lines that drive various digital logic components

- Microprogrammed control
  - Employs software consisting of microinstructions that carry out instruction's microoperations (each step in the instruction processing)
  - Microinstructions are stored in memory (control store)
  - Each microinstruction specifies the values of control signals

An Elegant Multi-Cycle Processor Design

- Maurice Wilkes, "The Best Way to Design an Automatic Calculating Machine," Manchester Univ. Computer Inaugural Conf., 1951.

THE BEST WAY TO DESIGN AN AUTOMATIC CALCULATING MACHINE

By M. V. Wilkes, M.A., Ph.D., F.R.A.S.

- An elegant implementation:
  - The concept of microcoded/microprogrammed machines

Microprogrammed Control Terminology

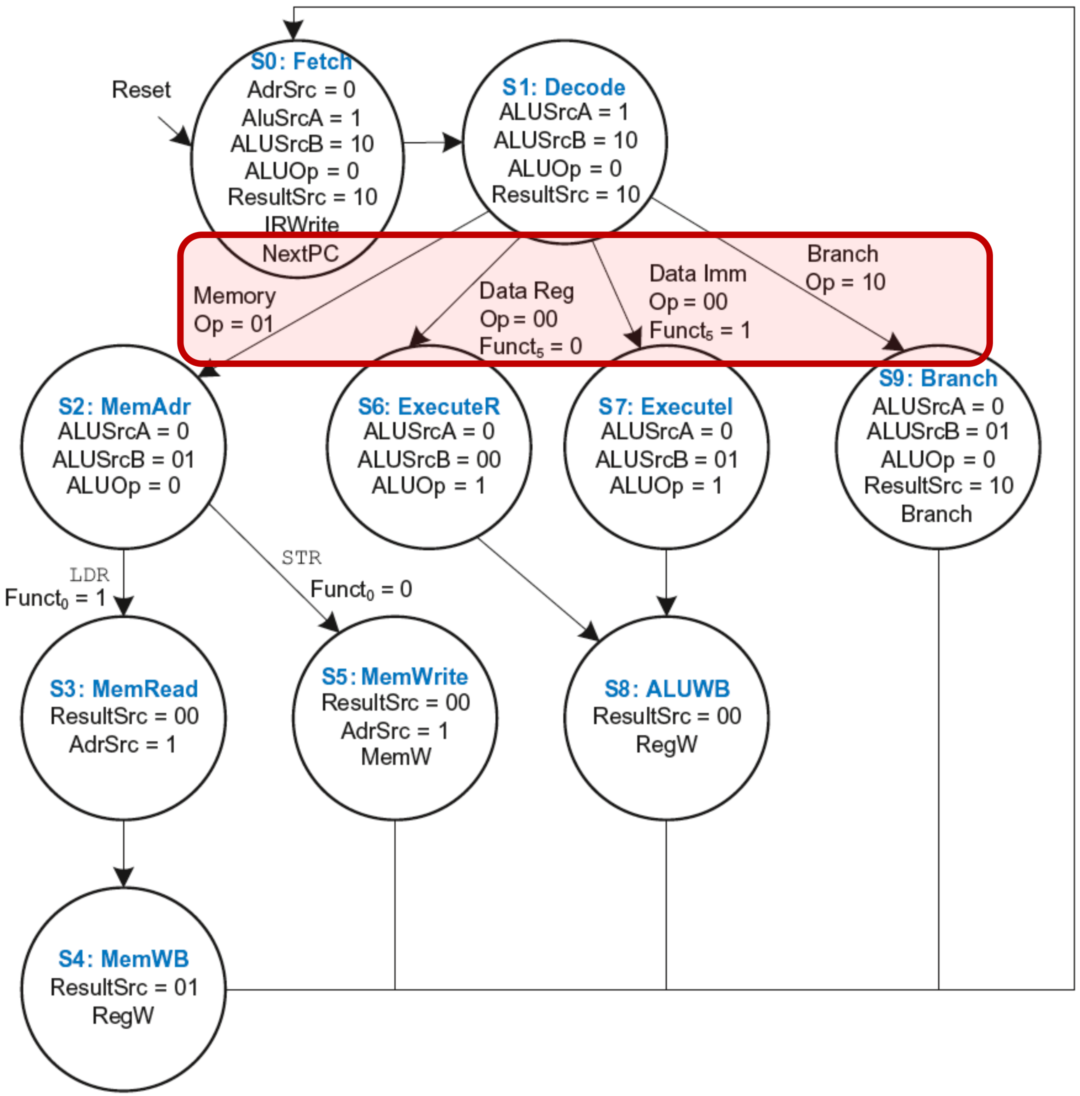
- Control signals associated with the current state
  - Microinstruction

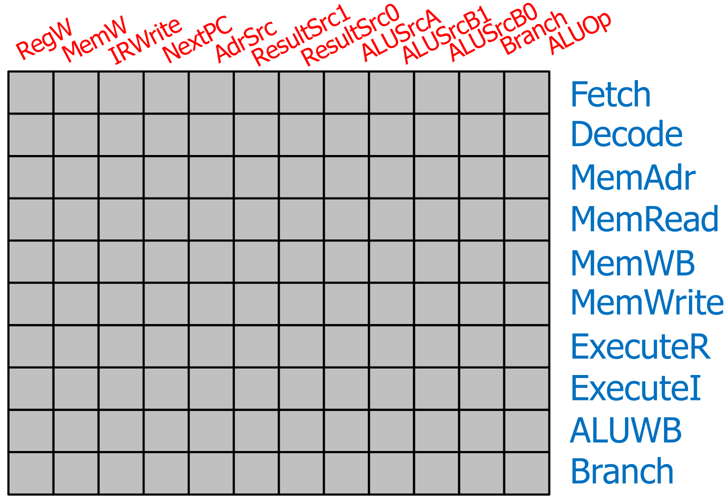
- Act of transitioning from one state to another
  - Determining the next state and the microinstruction for the next state
  - Microsequencing

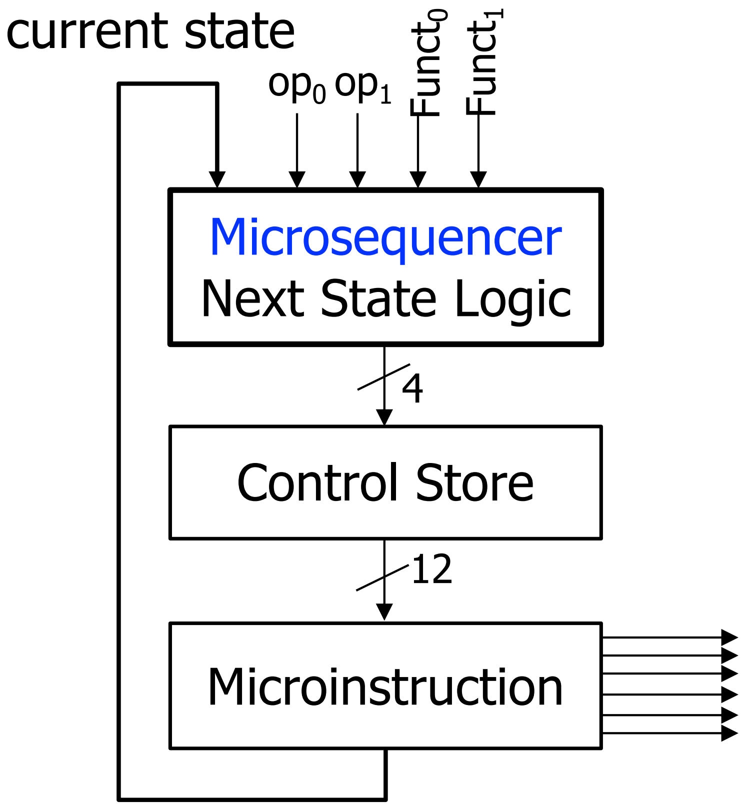
- Control store stores control signals for every possible state
  - Store for microinstructions for the entire FSM

- Microsequencer determines which set of control signals will be used in the next clock cycle (i.e., next state)

Microprogrammed ARM







What Happens In A Clock Cycle?

- The control signals (microinstruction) for the current state control two things:
  - Processing in the data path
  - Generation of control signals (microinstruction) for the next cycle

- Datapath and microsequencer operate concurrently

Microprogrammed Control Structure

- Three components: Microinstruction, Control store, Microsequencer

- Microinstruction: control signals that control the datapath (26 of them) and help determine the next state (9 of them)

- Each microinstruction is stored in a *unique location* in the control store (a special memory structure)
  - *Unique location*: address of the state corresponding to the microinstruction
  - Each state in the FSM corresponds to one microinstruction

- Microsequencer determines the address of the next microinstruction (i.e., next state)

Multicycle Microarchitecture:

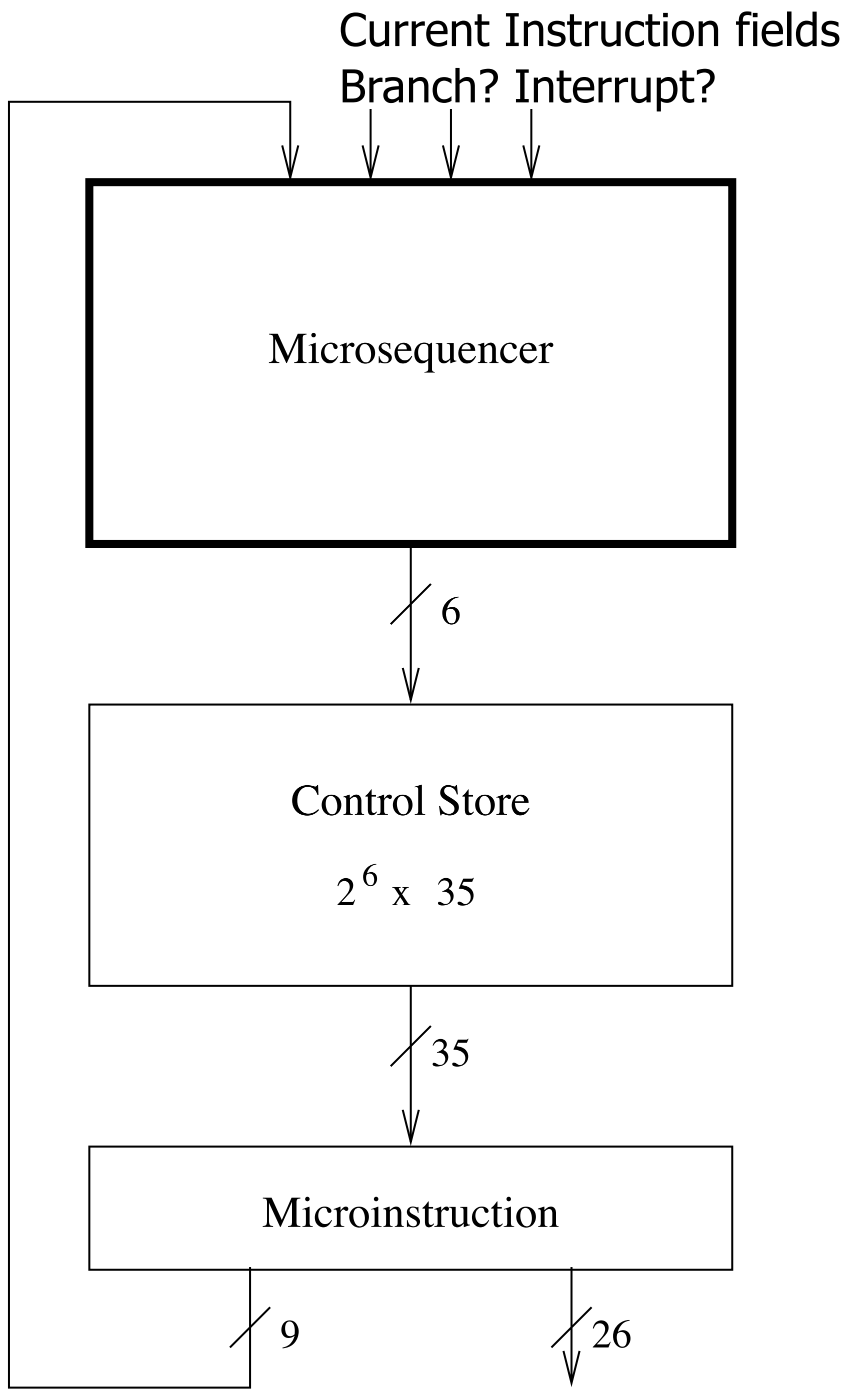
- Three components: Microinstruction, Control store, Microsequencer
- Microinstruction: control signals that control the datapath (26 of them) and help determine the next state (9 of them)
- Each microinstruction is stored in a *unique location* in the control store (a special memory structure)
  - *Unique location*: address of the state corresponding to the microinstruction
  - Each state in the FSM corresponds to one microinstruction
- Microsequencer determines the address of the next microinstruction (i.e., next state)

Microprogrammed Control Structure

A Simple Datapath Can Become
Very Powerful by Enabling a New Level of
Programmability Post-Fabrication

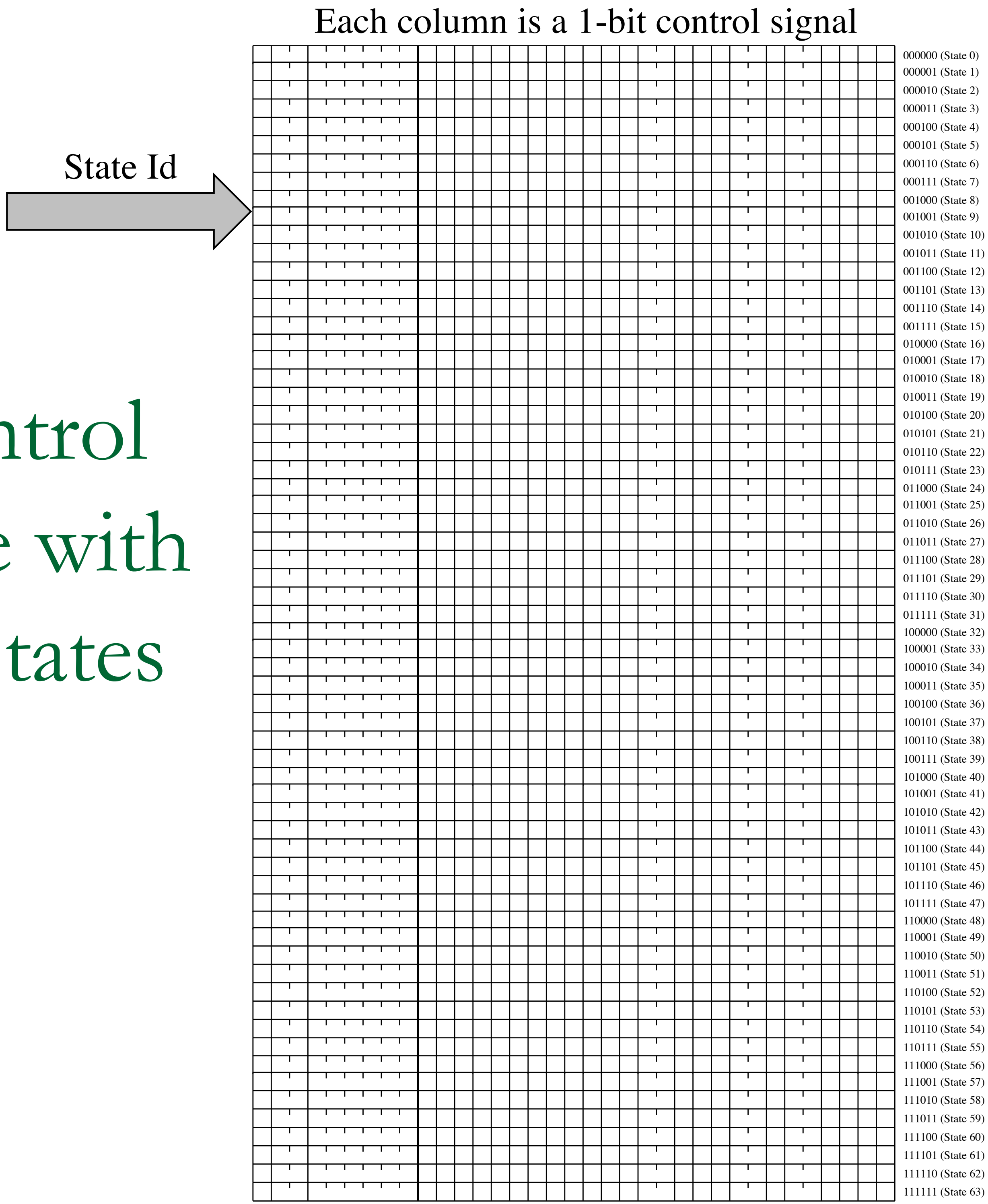
Example Control Structure

- 64 States
- **Microinstruction**
  - 26 signals to control datapath
  - 9 signals to help generate the next instruction
- Microseqeuncer takes as input some other signals from the datapath



Simple Design
of the Control Structure

Control Store with 64 States



Each entry in the control store is a microinstruction corresponding to the FSM state

FSM state number is used to address the control store to get the relevant microinstruction

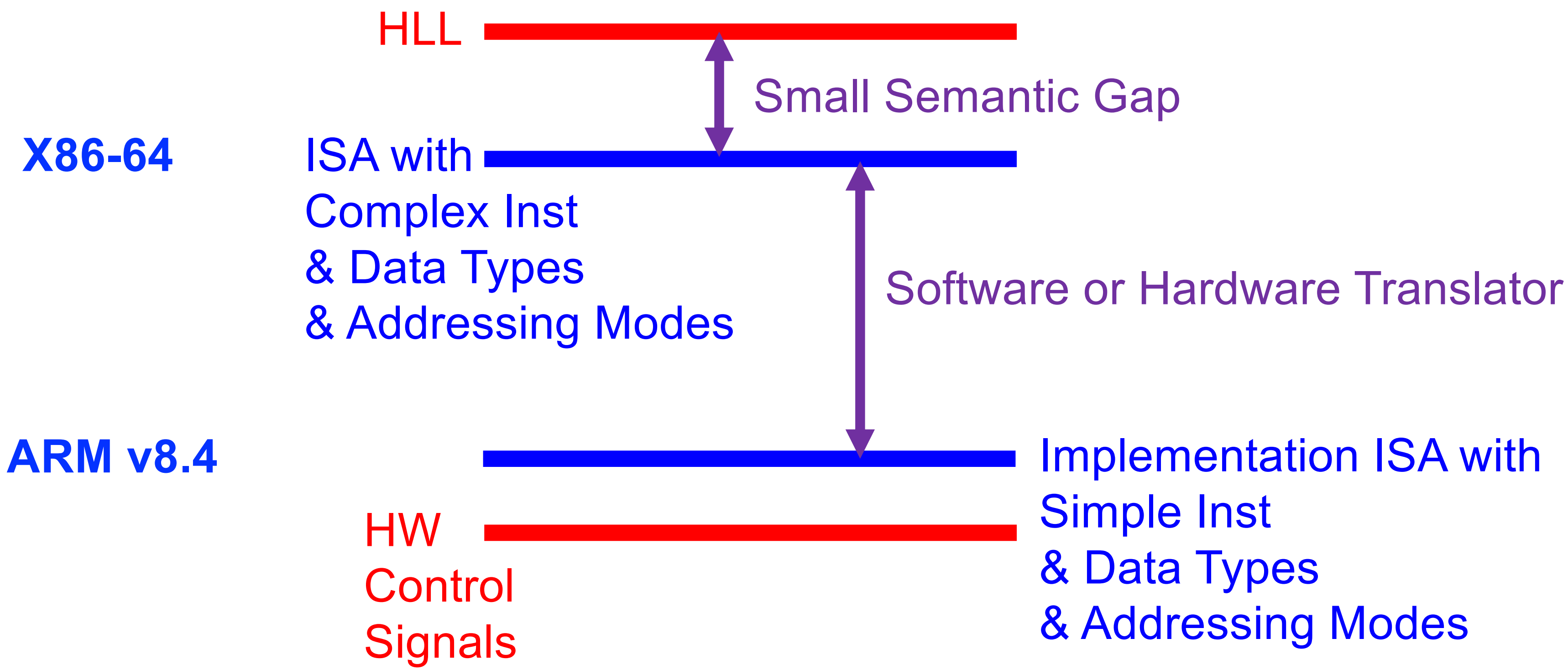
The Power of Abstraction

- The concept of a control store of microinstructions enables the hardware designer with a new abstraction: microprogramming

- The designer can translate any desired operation to a sequence of microinstructions
- All the designer needs to provide is
  - The sequence of microinstructions needed to implement the desired operation
  - The ability for the control logic to correctly sequence through the microinstructions
  - Any additional datapath elements and control signals needed (no need if the operation can be "translated" into existing control signals)

How to Change the Semantic Gap Tradeoffs?

- Translate from one ISA into a different "implementation" ISA



Recall: How to Change the Semantic Gap Tradeoffs

- An Example: Rosetta 2 Binary Translator ′ ISA

Rosetta 2 [edit]

In 2020, Apple announced Rosetta 2 would be bundled with macOS Big Sur, to aid in the Mac transition to Apple silicon. The software permits many applications compiled exclusively for execution on x86-64-based processors to be translated for execution on Apple silicon.[2][8]

In addition to the just-in-time (JIT) translation support, Rosetta 2 offers ahead-of-time compilation (AOT), with the x86-64 code fully translated, just once, when an application without a universal binary is installed on an Apple silicon Mac.[9]

Rosetta 2's performance has been praised greatly.[10][11] In some benchmarks, x86-64-only programs performed better under Rosetta 2 on a Mac with an Apple M1 SOC than natively on a Mac with an Intel x86-64 processor. One of the key reasons why Rosetta 2 provides such high level of translation efficiency is the support of x86-64 memory ordering in Apple M1 SOC.[12]

Mac transition to Apple silicon

Apple silicon · ARM architecture · Universal 2 binary · Rosetta 2 · Developer Transition Kit

V · T · E

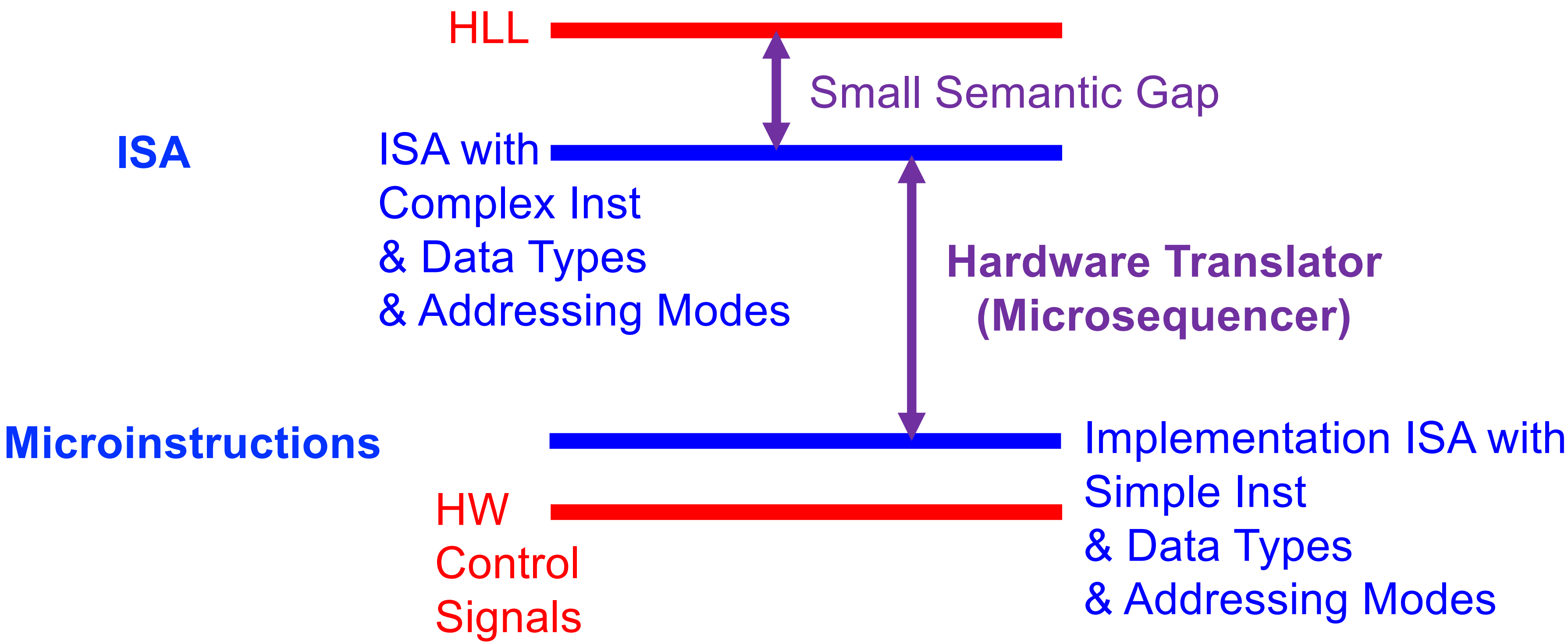
Although Rosetta 2 works for most software, some software doesn't work at all[13] or is reported to be "sluggish".[14] A lot of software can be made compatible with the new Macs by the vendor recompiling the software, often a simple task; while for some software (such as software that includes assembly language code, or that generates machine code), the changes to make them work aren't simple and cannot be automated.

Similar to the first version, Rosetta 2 does not normally require user intervention. When a user attempts to launch an x86-64-only application for the first time, macOS prompts them to install Rosetta 2 if it is not already available. Subsequent launches of x86-64 programs will execute via translation automatically. An option also exists to force a universal binary to run as x86-64 code through Rosetta 2, even on an ARM-based machine.[15]

https://en.wikipedia.org/wiki/Rosetta_(software)#Rosetta_2

How to Change the Semantic Gap Tradeoffs

- Translate from one ISA into a different "implementation" ISA



Advantages of Microprogrammed Control

- Allows a very simple design to do powerful computation by controlling the datapath (using a sequencer)
  - High-level ISA translated into microcode (sequence of u-instructions)
  - Microcode (u-code) enables a minimal datapath to emulate an ISA
  - Microinstructions can be thought of as a user-invisible ISA (u-ISA)

- Enables easy extensibility of the ISA
  - Can support a new instruction by changing the microcode
  - Can support complex instructions as a sequence of simple microinstructions (e.g., MultiDimensional Array Updates)

- Enables update of machine behavior
  - A buggy implementation of an instruction can be fixed by changing the microcode in the field
    - Easier if datapath provides ability to do the same thing in different ways

Update of Machine Behavior

- The ability to update/patch microcode in the field (after a processor is shipped) enables
  - Ability to add new instructions without changing the processor!
  - Ability to "fix" buggy hardware implementations

- Historical Examples
  - IBM 370 Model 145: microcode stored in main memory, can be updated after a reboot
  - IBM System z: Similar to 370/145.
    - Heller and Farrell, "Millicode in an IBM zSeries processor," IBM JR&D, May/Jul 2004.
  - B1700 microcode can be updated while the processor is running
    - User-microprogrammable machine!
    - Wilner, "Microprogramming environment on the Burroughs B1700", CompCon 1972.
  - Systems today use microcode patches to fix HW bugs/issues

Can We Do Better?

Can We Do Better?

- What limitations do you see with the multi-cycle design?

- Limited concurrency
  - Some hardware resources are idle during different phases of instruction processing cycle
  - "Fetch" logic is idle when an instruction is being "decoded" or "executed"
  - Most of the datapath is idle when a memory access is happening

Can We Use the Idle Hardware to Improve Concurrency?

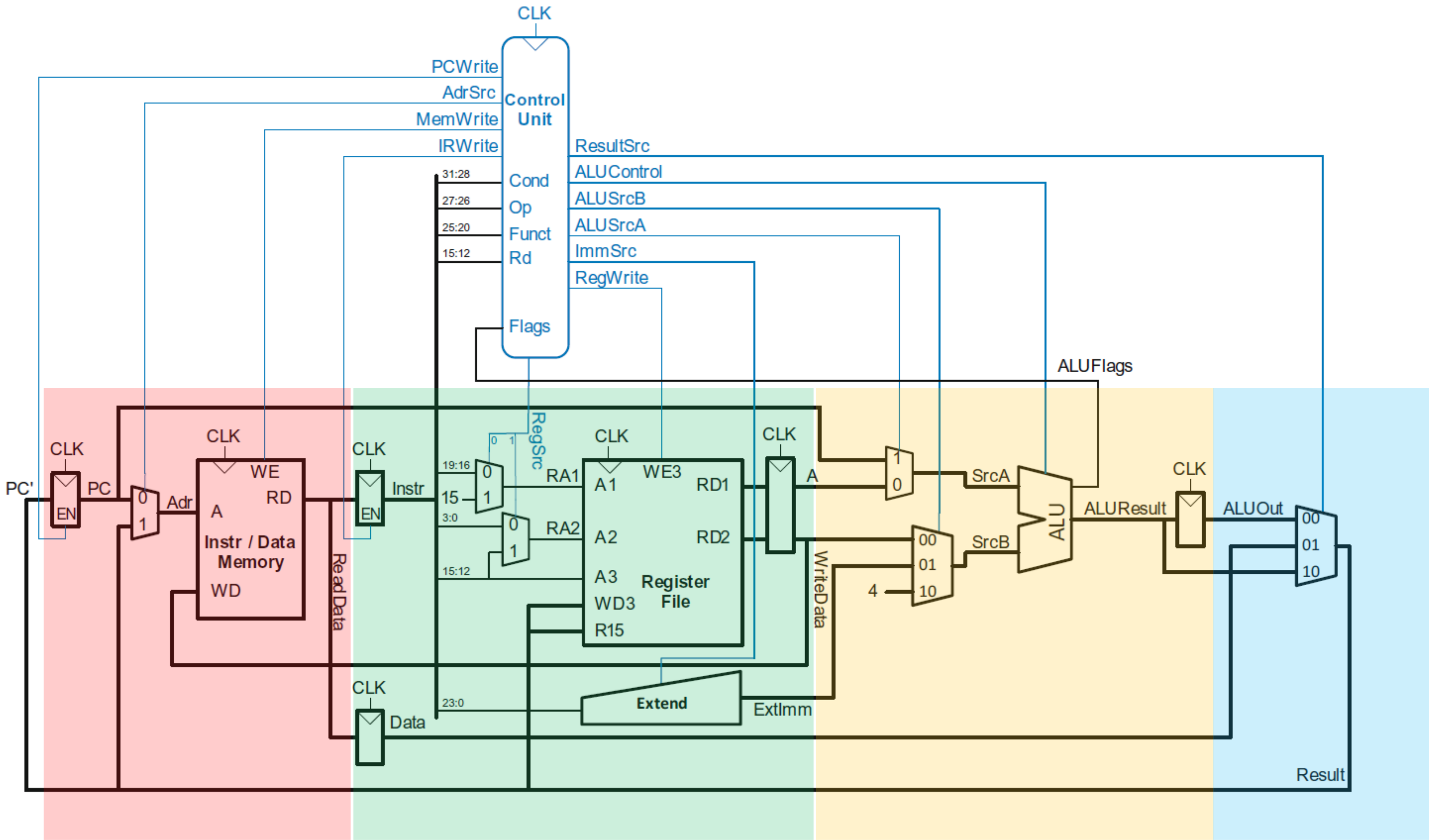
- Goal: More concurrency → Higher instruction throughput (i.e., more "work" completed in one cycle)

- Idea: When an instruction is using some resources in its processing phase, process other instructions on idle resources not needed by that instruction
  - E.g., when an instruction is being decoded, fetch the next instruction
  - E.g., when an instruction is being executed, decode another instruction
  - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
  - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction

Can Have Different Instructions in Different Stages

- Instruction Fetch (IF)
- Instruction Decode and Register Read (ID/RF)
- Execute (EX)
- Memory Access (MEM)
- Writeback (WB)

Can Have Different Instructions in Different Stages



Of course, we need to be more careful than this!

Pipelining

Pipelining: Basic Idea

- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: "Assembly line processing" of instructions
- Idea:
  - Divide the instruction processing cycle into distinct "stages" of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a **different** instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput (1/CPI)
- Downside: Start thinking about this…