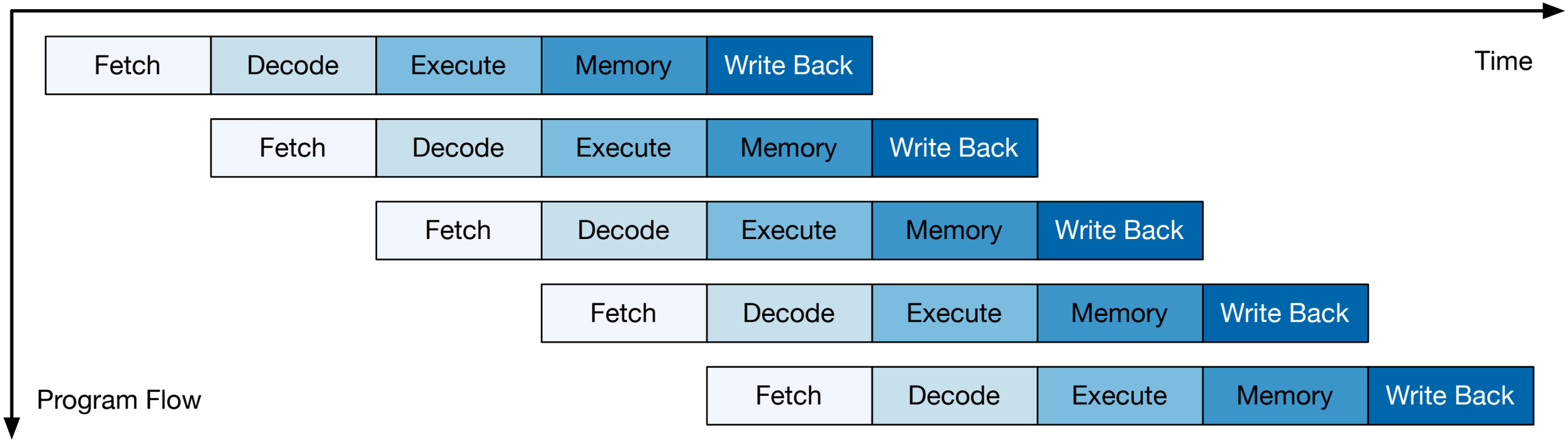
Lecture 16: Pipeline Controls

Spring 2024
Jason Tang

Topics

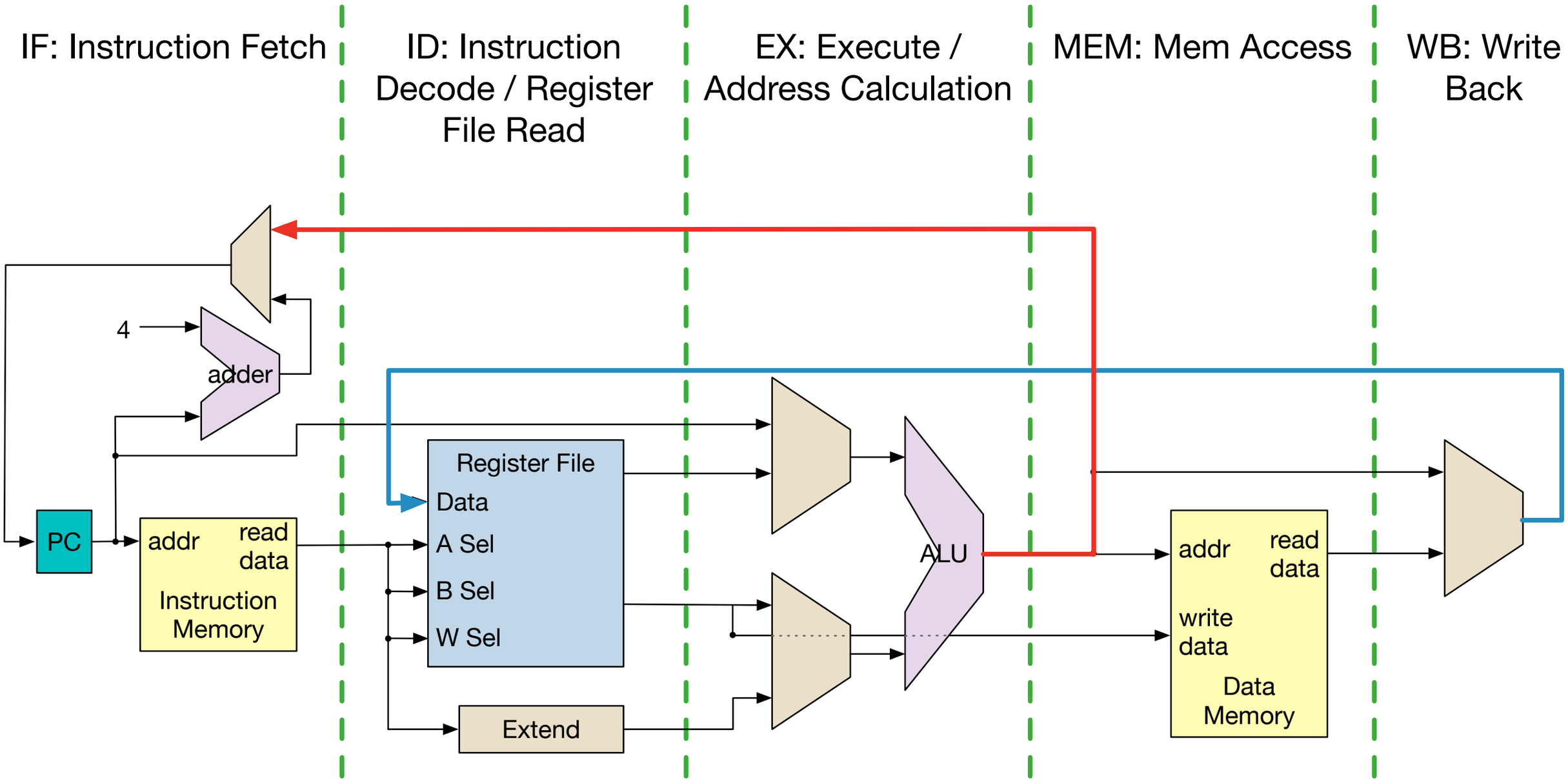
- Designing pipelined datapath
- Controlling pipeline operations

Instruction Pipelining

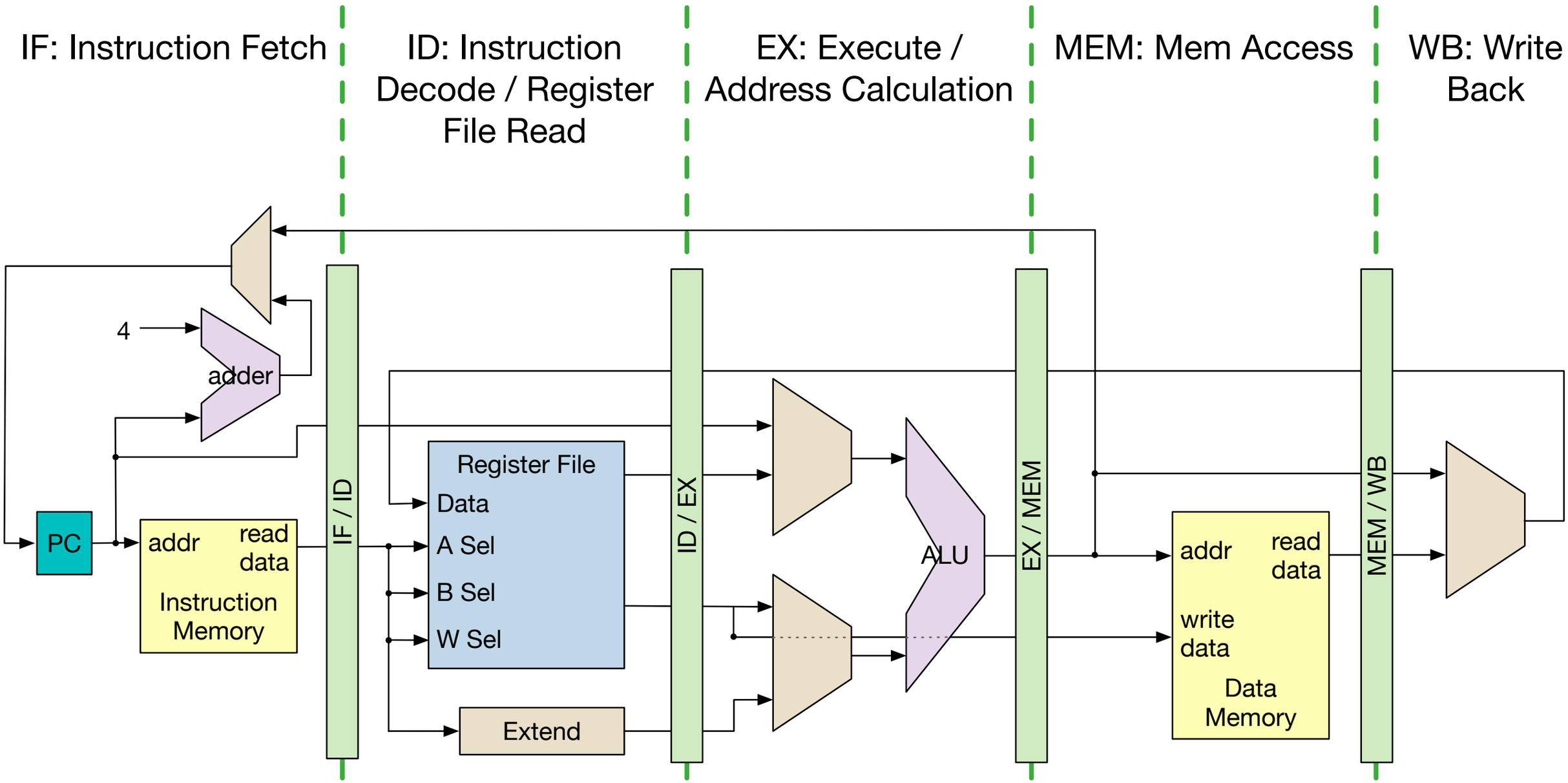


- Goal is to start executing next instruction on every clock cycle
- Pipelining only works when there are no resource contentions
  - If so, then either intentionally stall pipeline or duplicate those resources

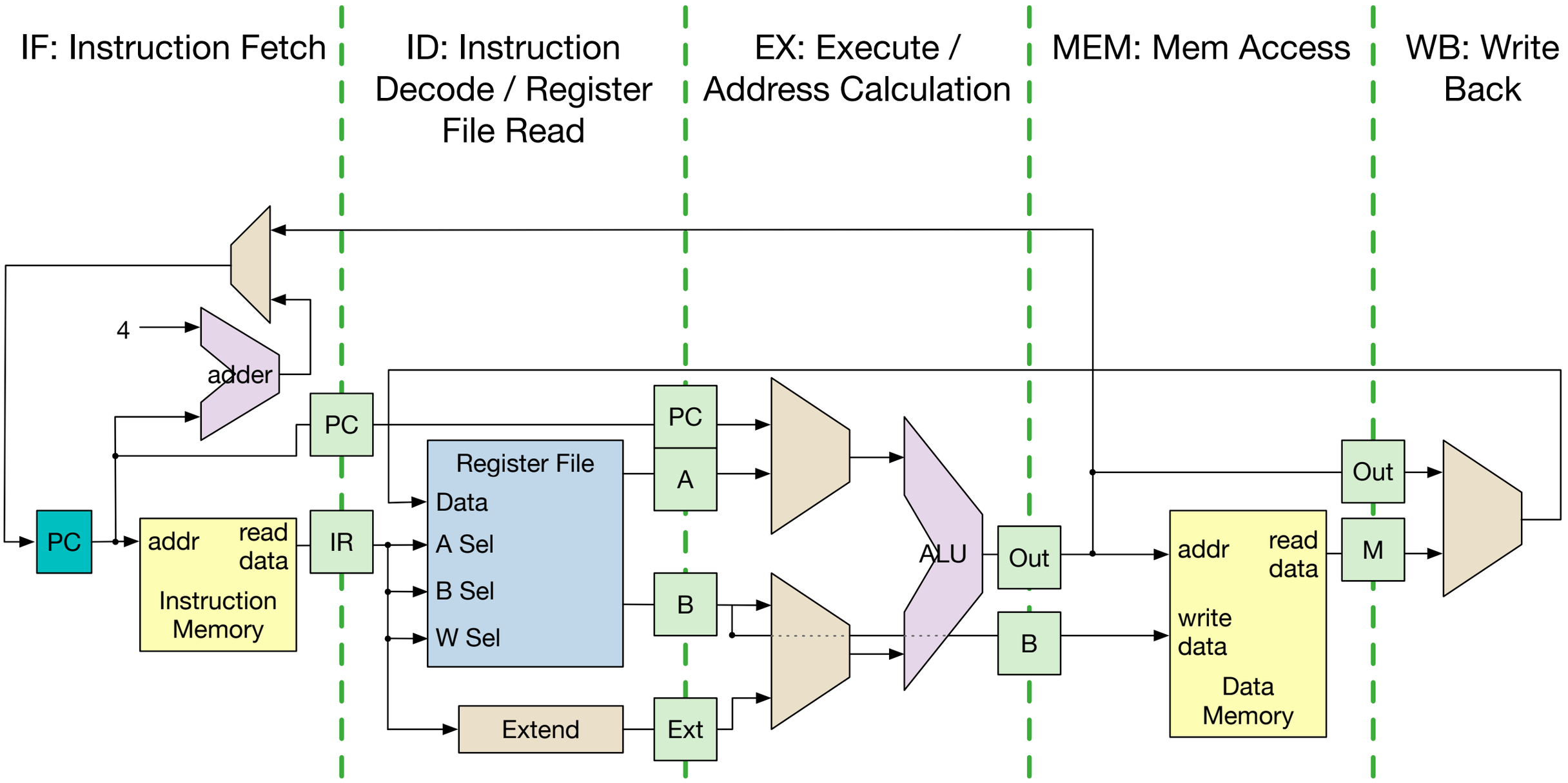
Multicycle Datapath



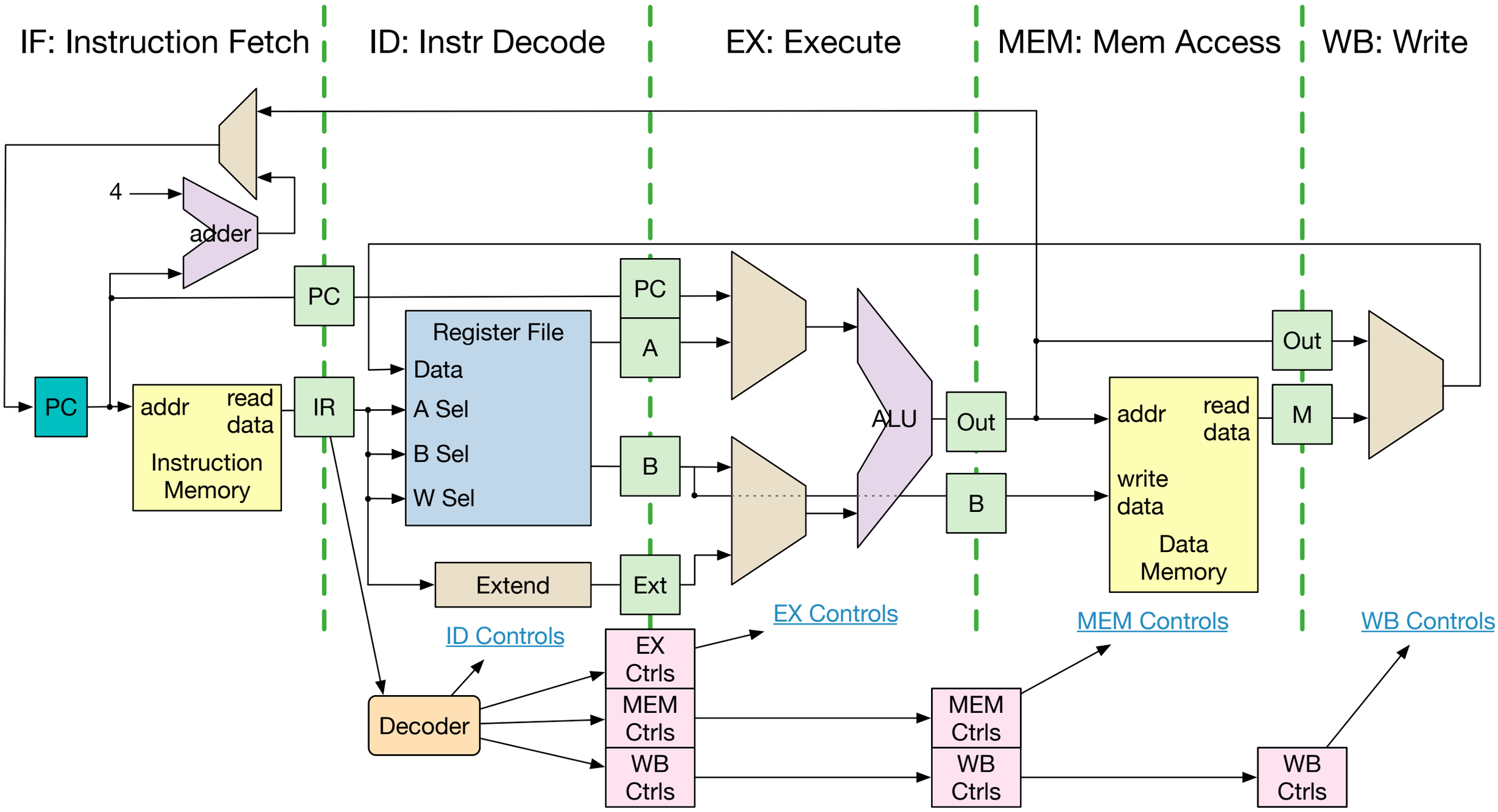
Pipelined Datapath



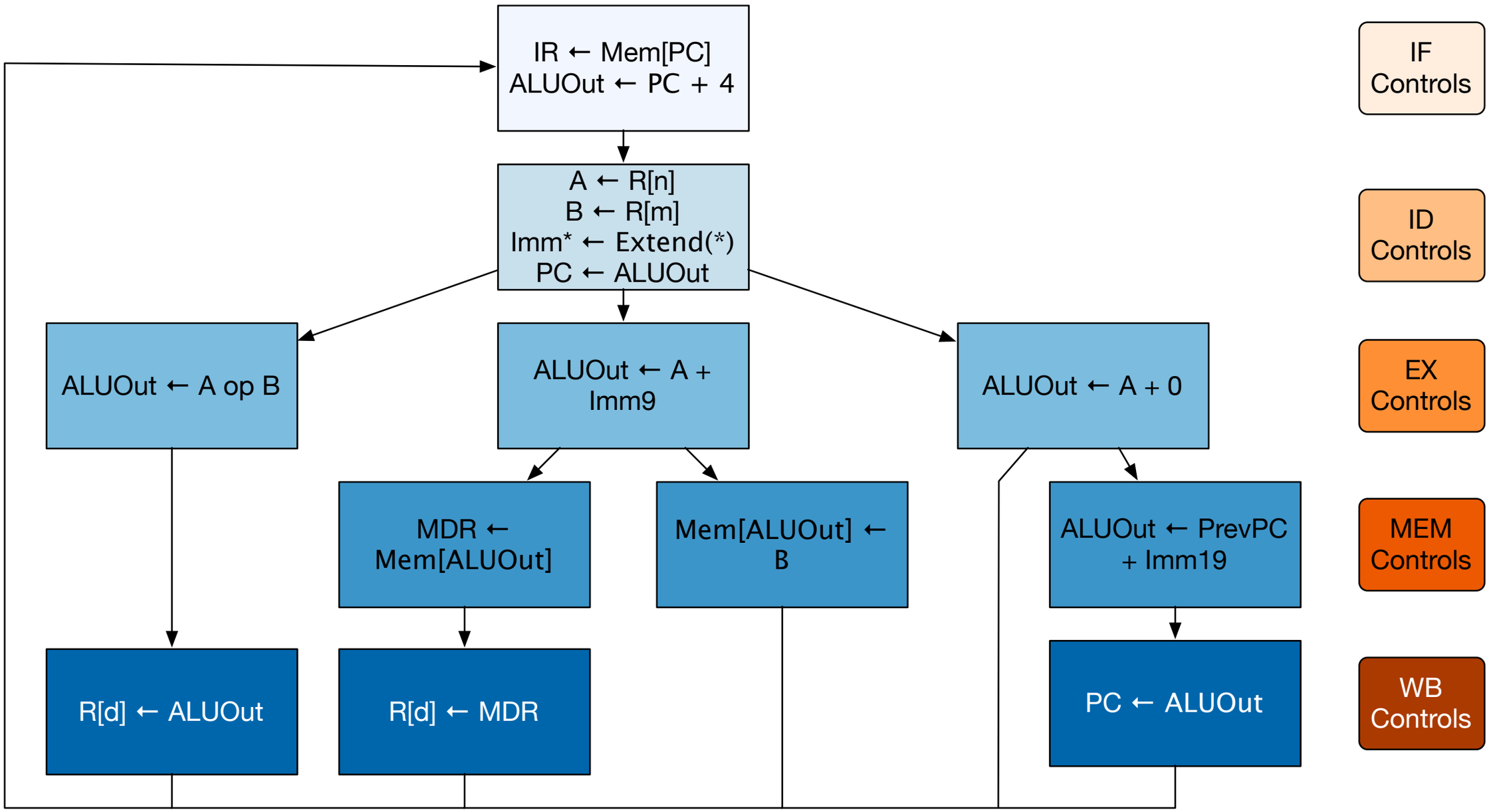
Pipeline Registers



Pipeline Controls



Controls and Datapath

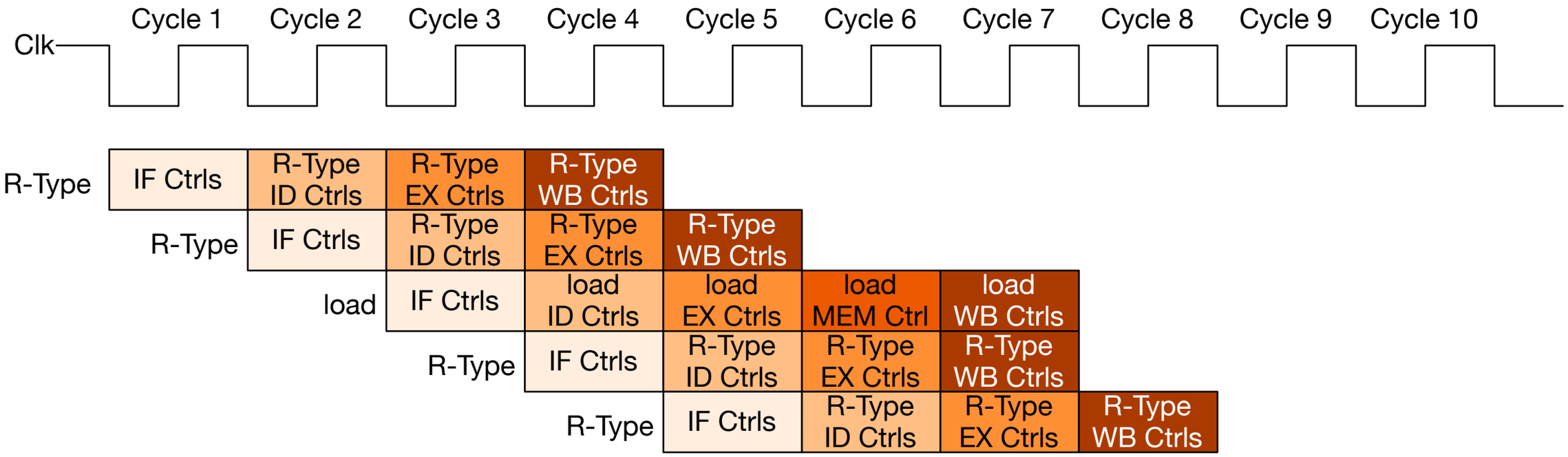


Pipeline Control Generation

- In single and multicycle datapaths, a single instruction decoder takes instruction register contents to generate control signals

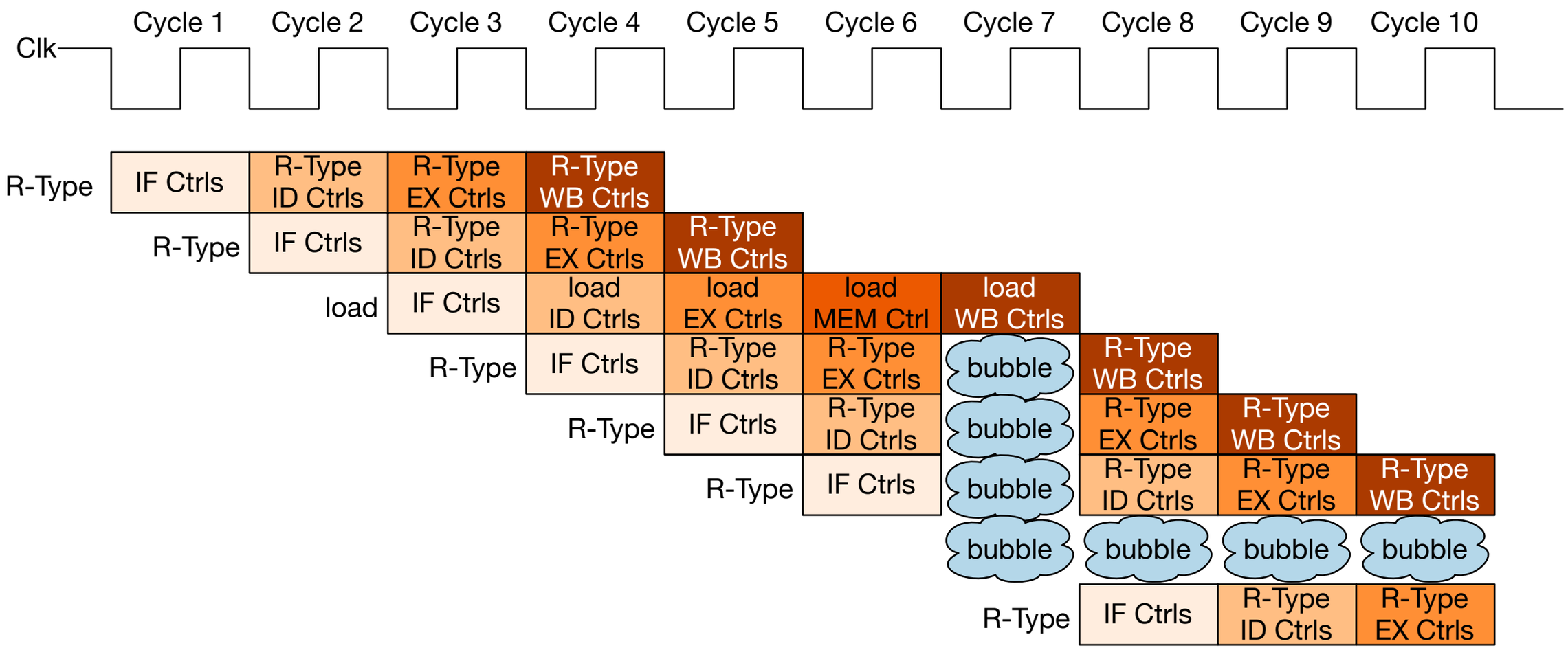
- In pipelined datapaths, either:
  - Single instruction decoder decodes upfront, then writes controls into data stationary registers to be used on subsequent cycles, or
  - Instruction register value itself is copied into data stationary registers, and then a local decoder generates signals for that portion of the datapath

Pipelining Instructions



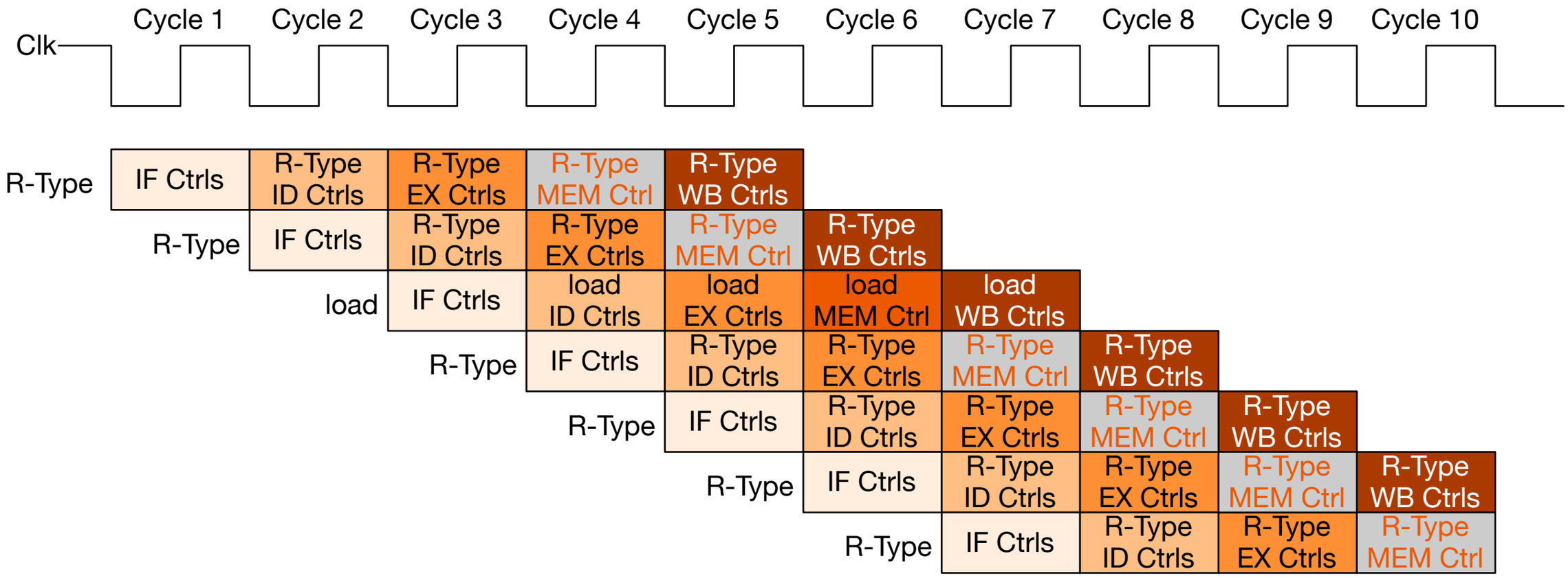
- Pipeline conflict (specifically, a structural hazard) when two instructions try to write to register file on same clock cycle
  - Register file can only store one new value at a time
  - Caused by uneven pipeline stages

Solution 1: Insert Bubble into Pipeline



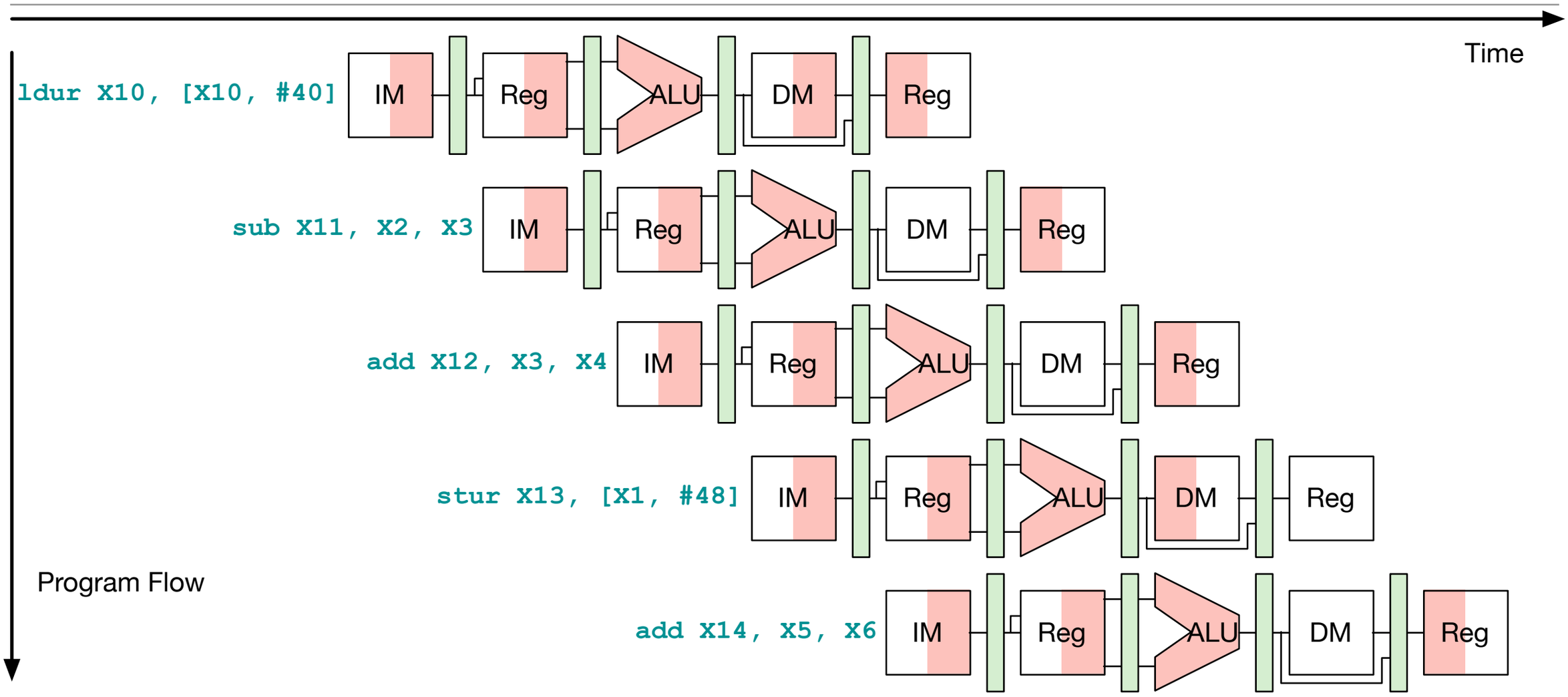
- Insert bubble into pipeline to prevent simultaneous writes
  - Control logic can be complex
  - No instruction started on Cycle 7

Solution 2: Delay Write by One Cycle



- Delay R-Type's write by one cycle
  - R-Type MEM Ctrl is a no-op; it generates no controls
- Now pipeline has same length for all instruction types

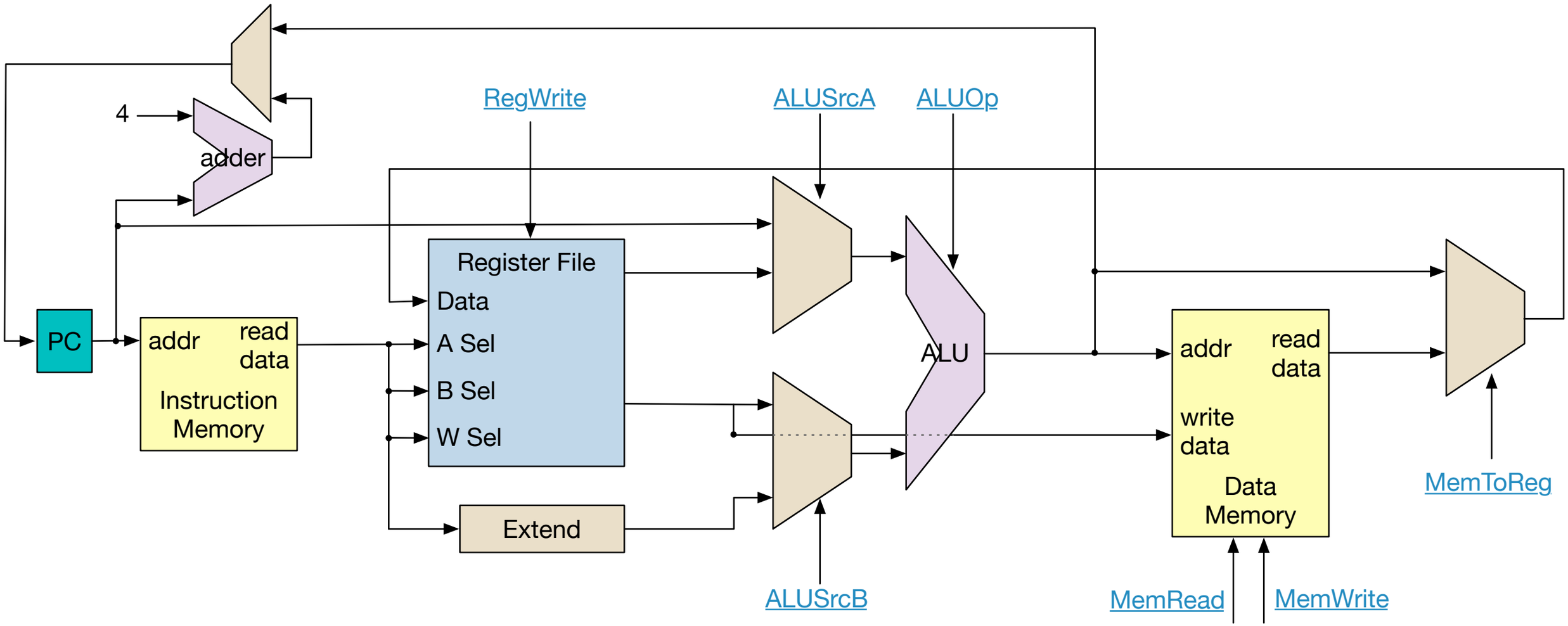
Pipelined Operations



- IM = Instruction Memory, Reg = Register File, DM = Data Memory

- Shaded on right side = read, Shaded on left = write

Pipeline Controls



- PC is updated every cycle, so no need for a control signal to control writing to PC

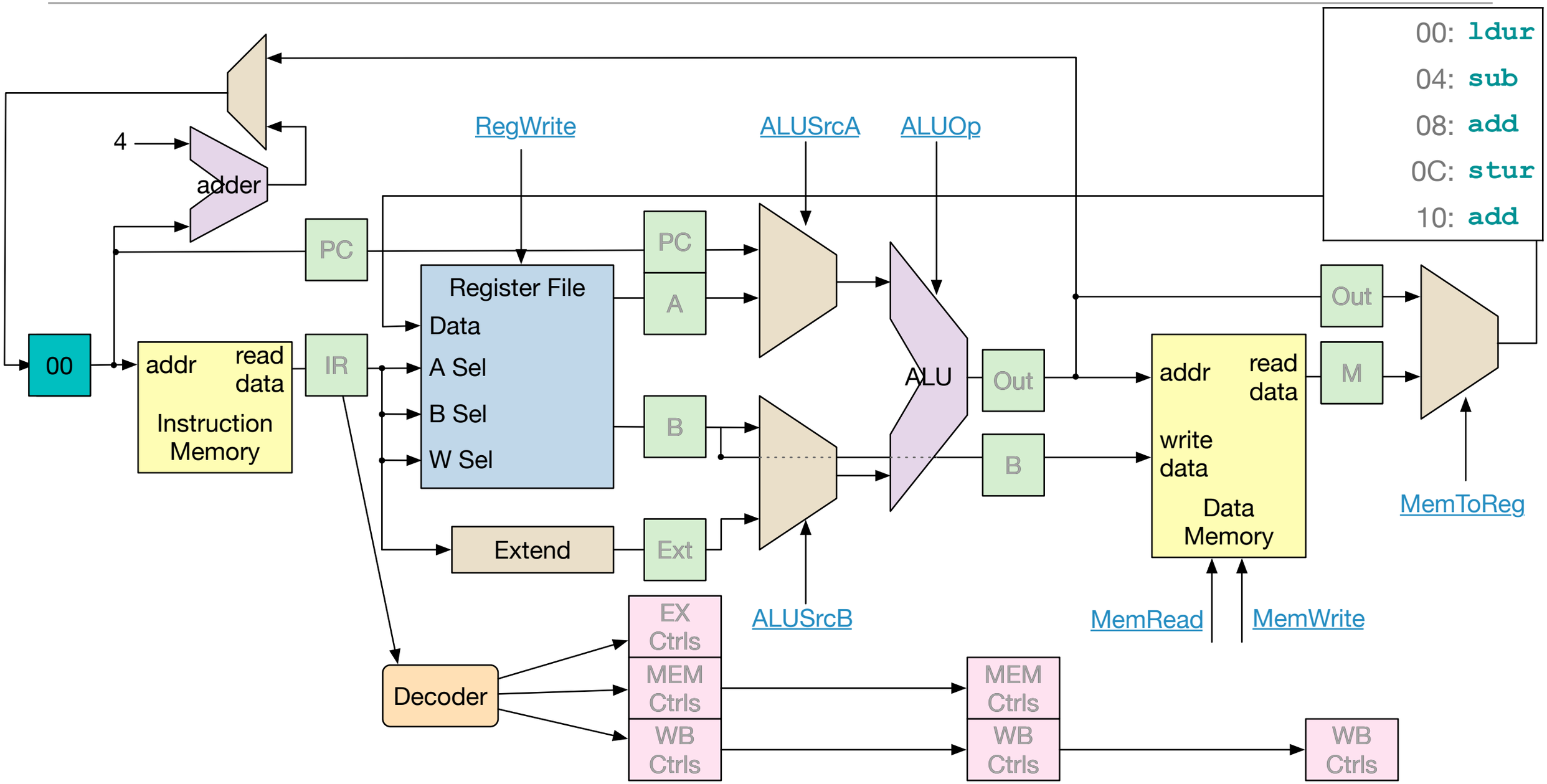
- Pipeline registers (green boxes) are also updated every cycle

Setting Pipeline Controls

Instruction	EX Ctrls			MEM Ctrls		WB Ctrls	
	ALUSrcA	ALUSrcB	ALUOp	MemRead	MemWrite	MemToReg	RegWrite
add	A	B	add	0	0	Out	1
sub	A	B	sub	0	0	Out	1
ldur	A	Ext (imm9)	add	1	0	M	1
stur	A	Ext (imm9)	add	0	1	X	0

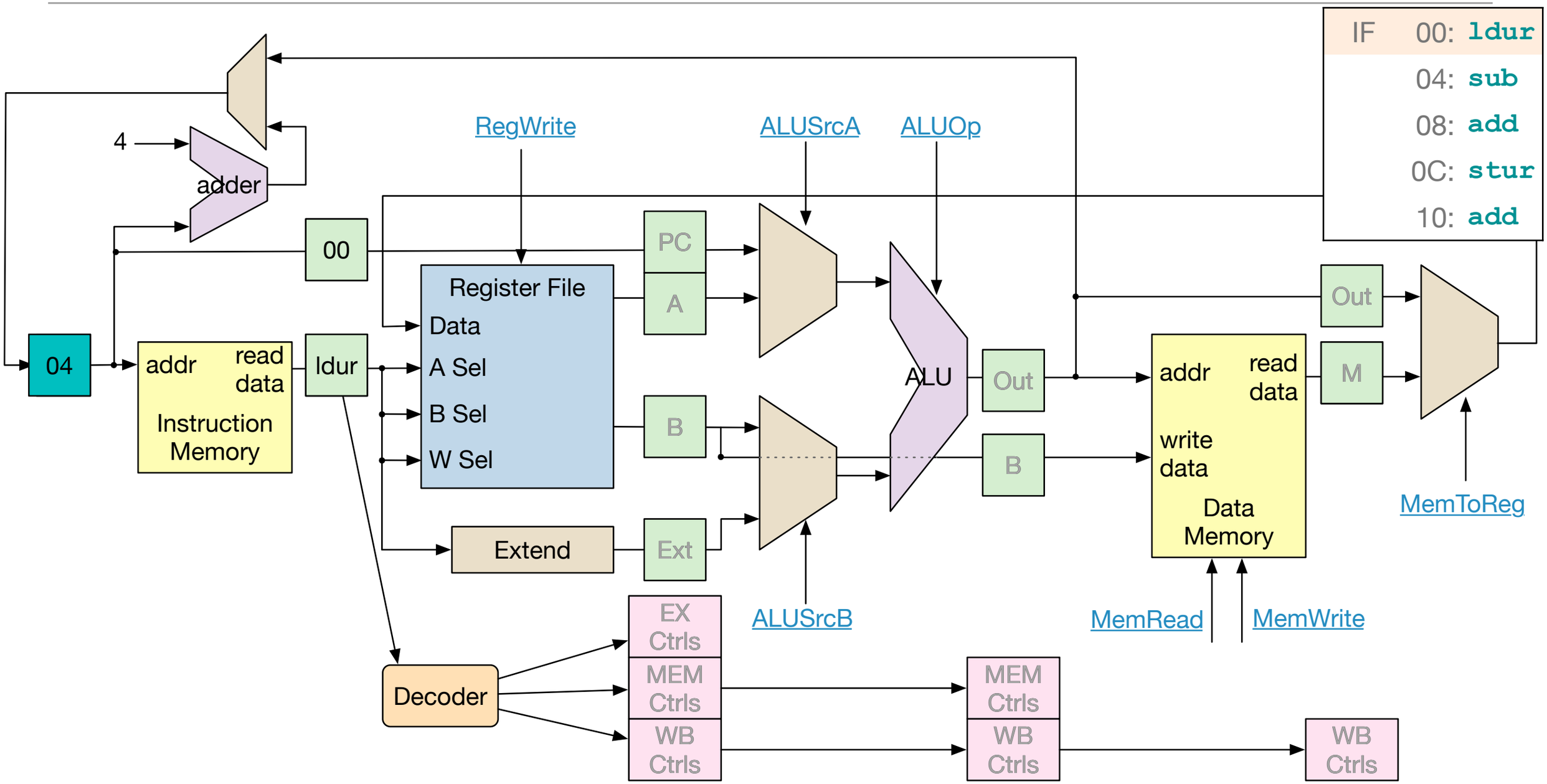
- Instruction decoder calculates control signal values given instruction register
  - Writes those values to pipeline registers, to be used by later cycles

Example of Pipeline: Cycle 0



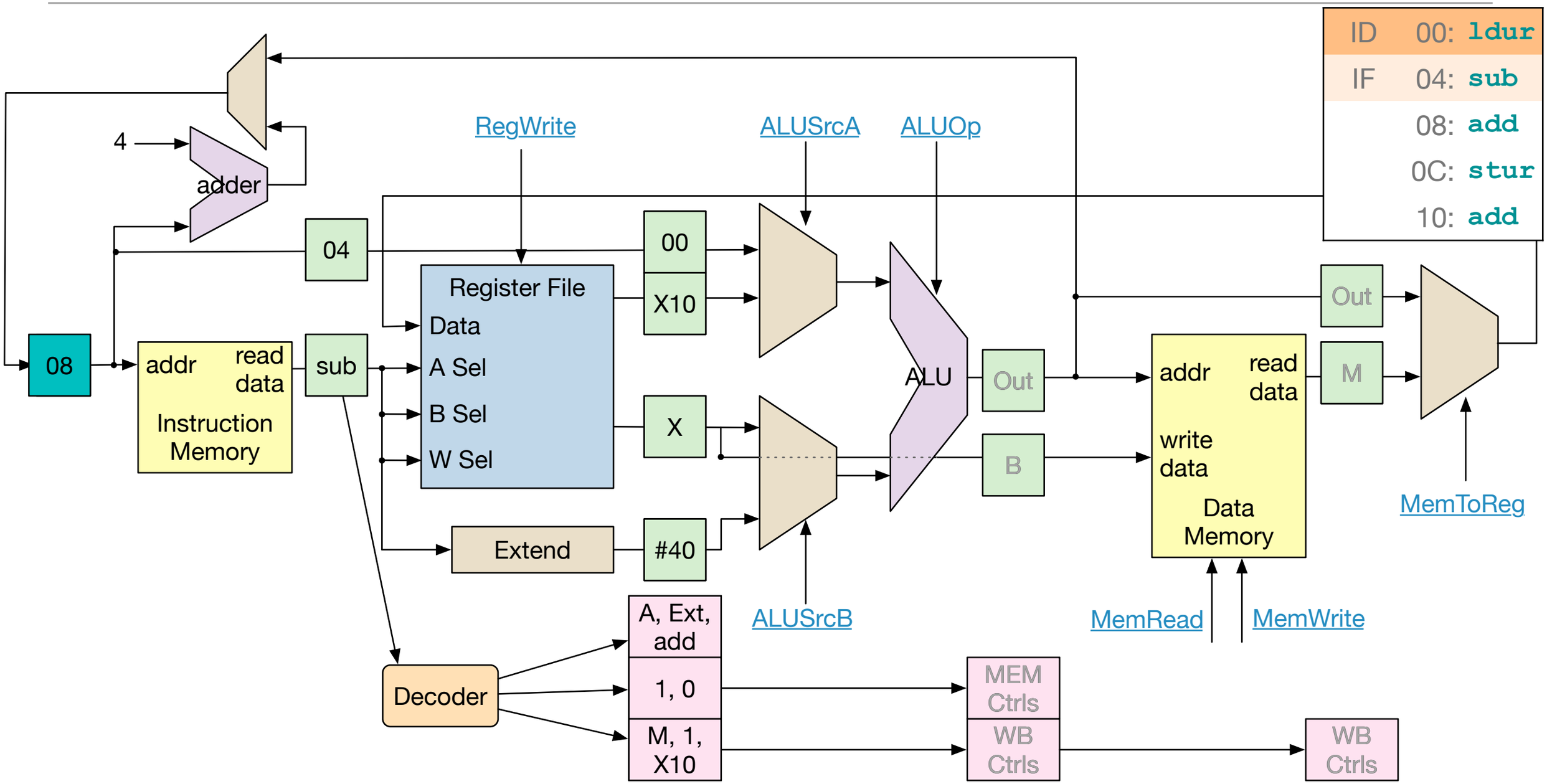
- Initial state (Cycle 0): PC = 00

Example of Pipeline: After Cycle 1



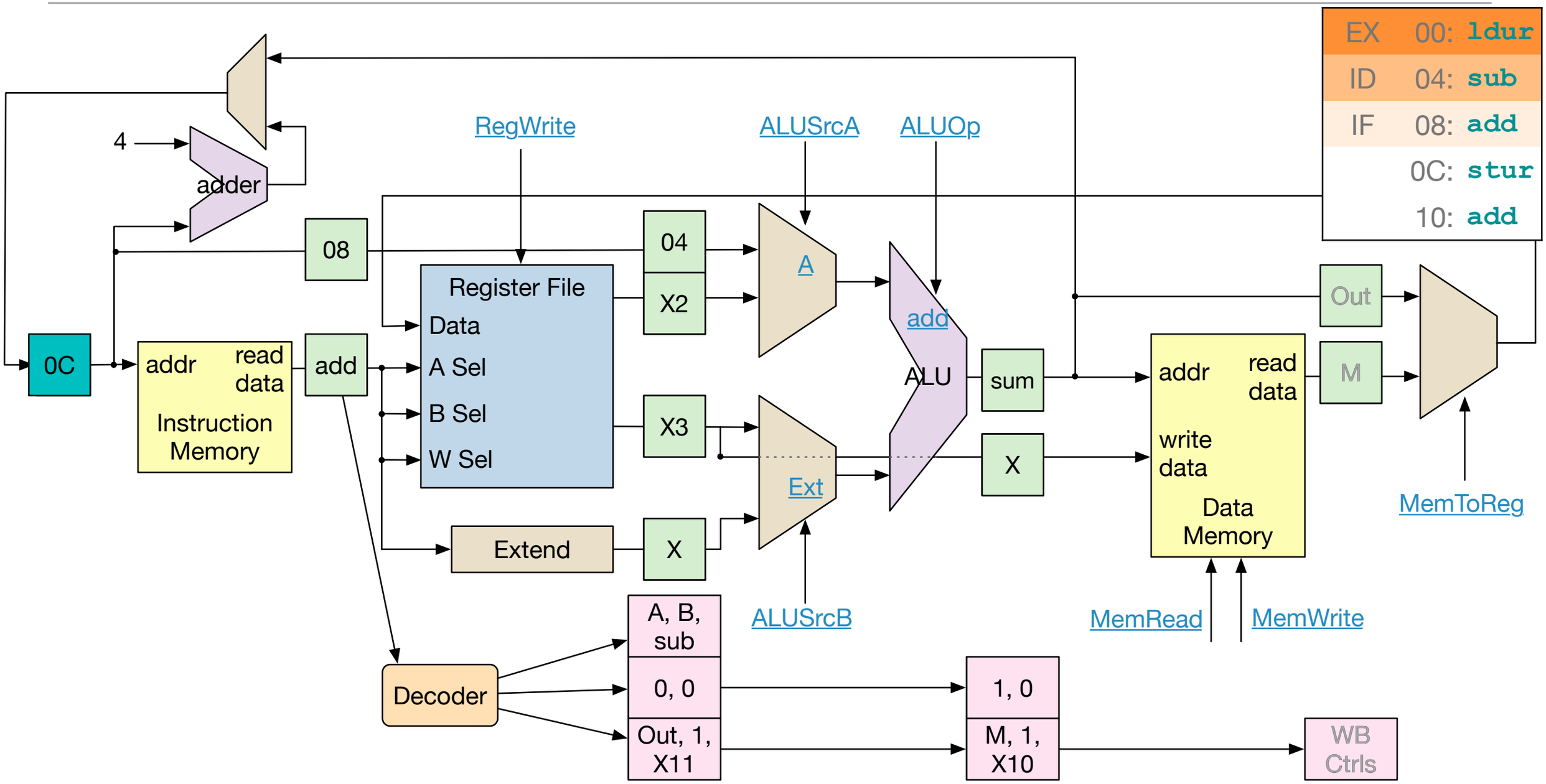
- Fetch from 00h, then increment PC

Example of Pipeline: After Cycle 2



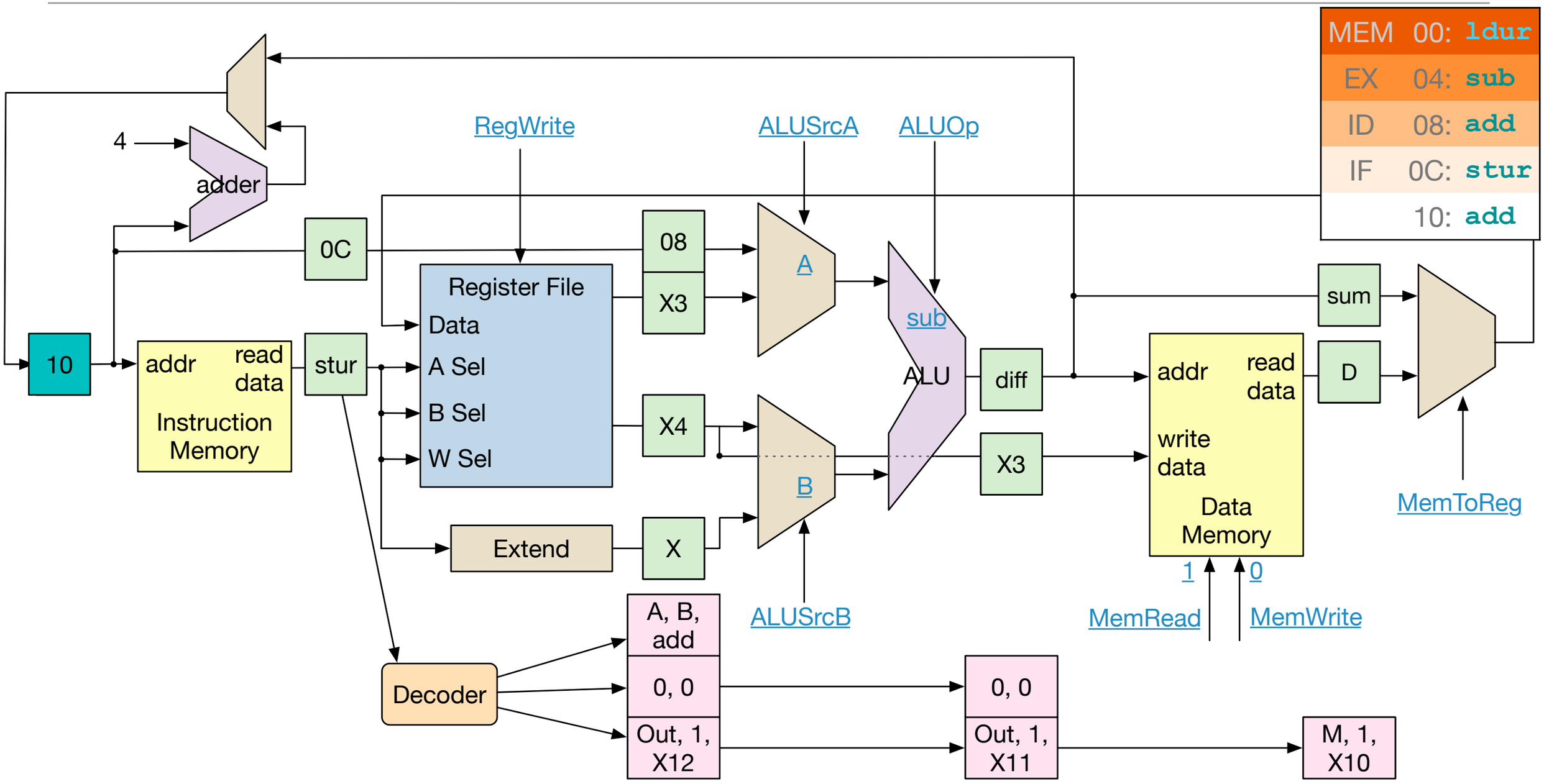
- Fetch from 04h; decode **`ldur X10, [X10, #40]`** and generate controls

Example of Pipeline: After Cycle 3



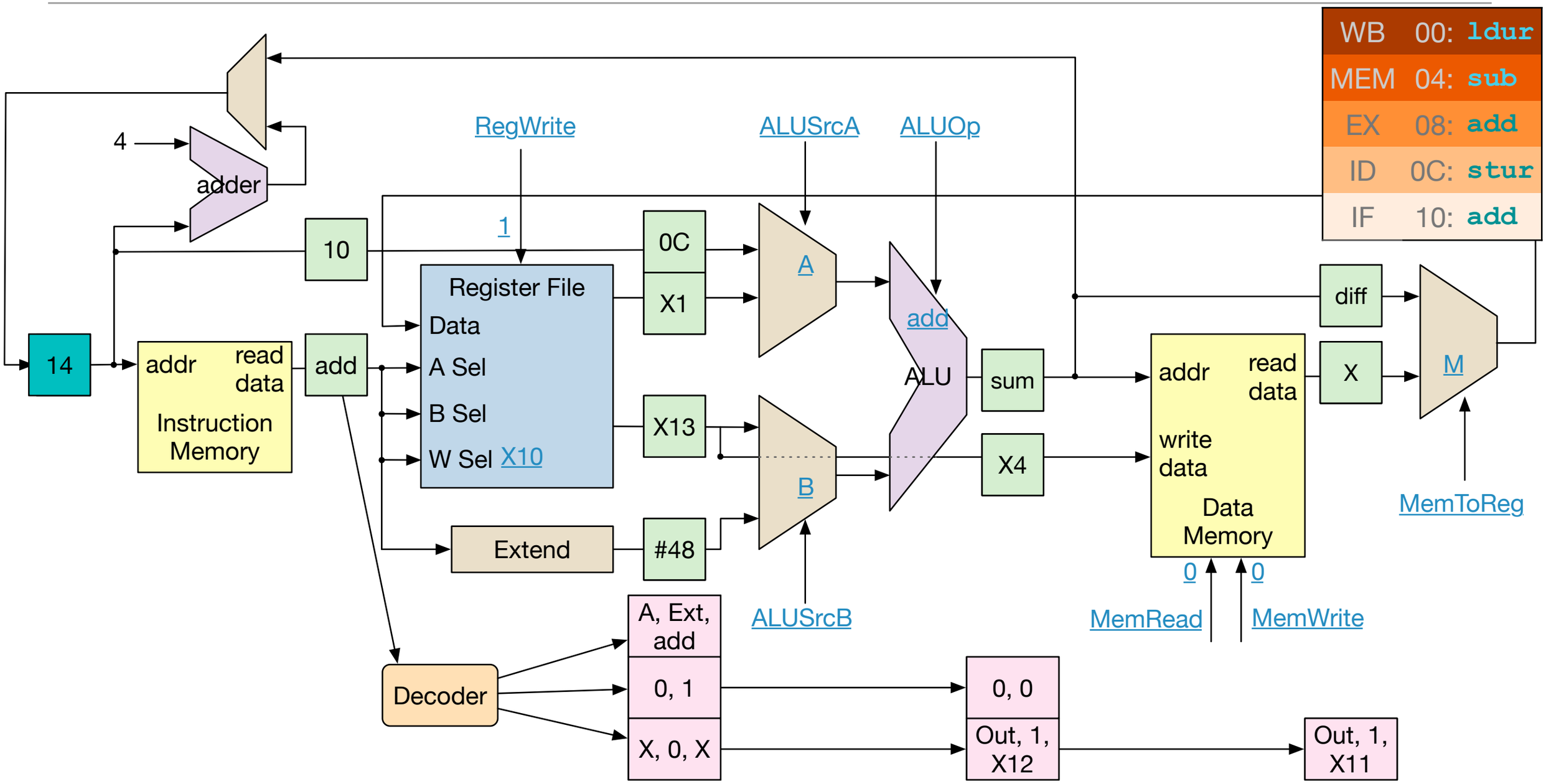
- Fetch from 08h; decode **`sub X11, X2, X3`**; execute **`ldur`**

Example of Pipeline: After Cycle 4



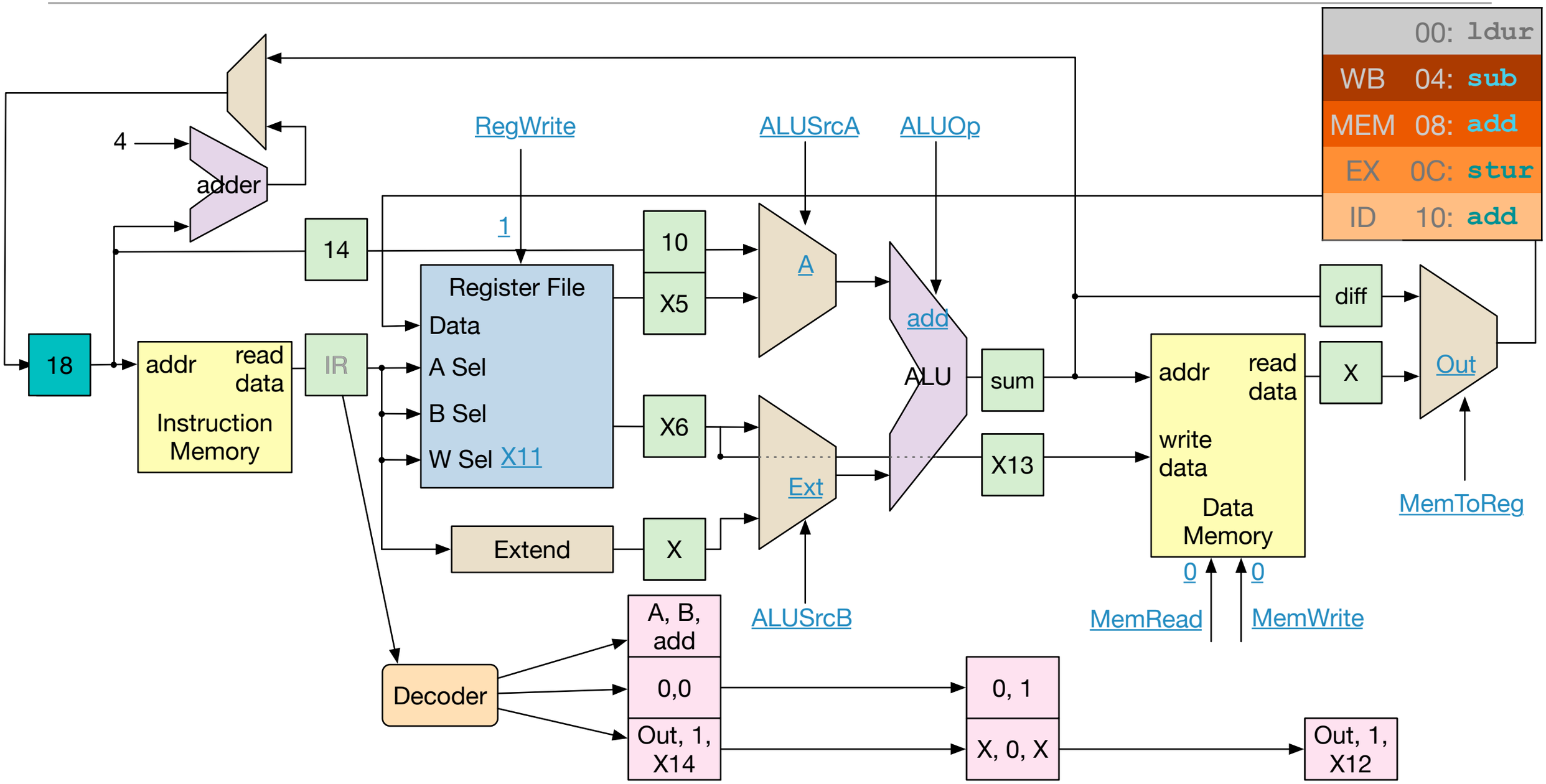
- Fetch 0Ah; decode **`add X12, X3, X4`**; execute **`sub`**; mem access **`ldur`**

Example of Pipeline: After Cycle 5



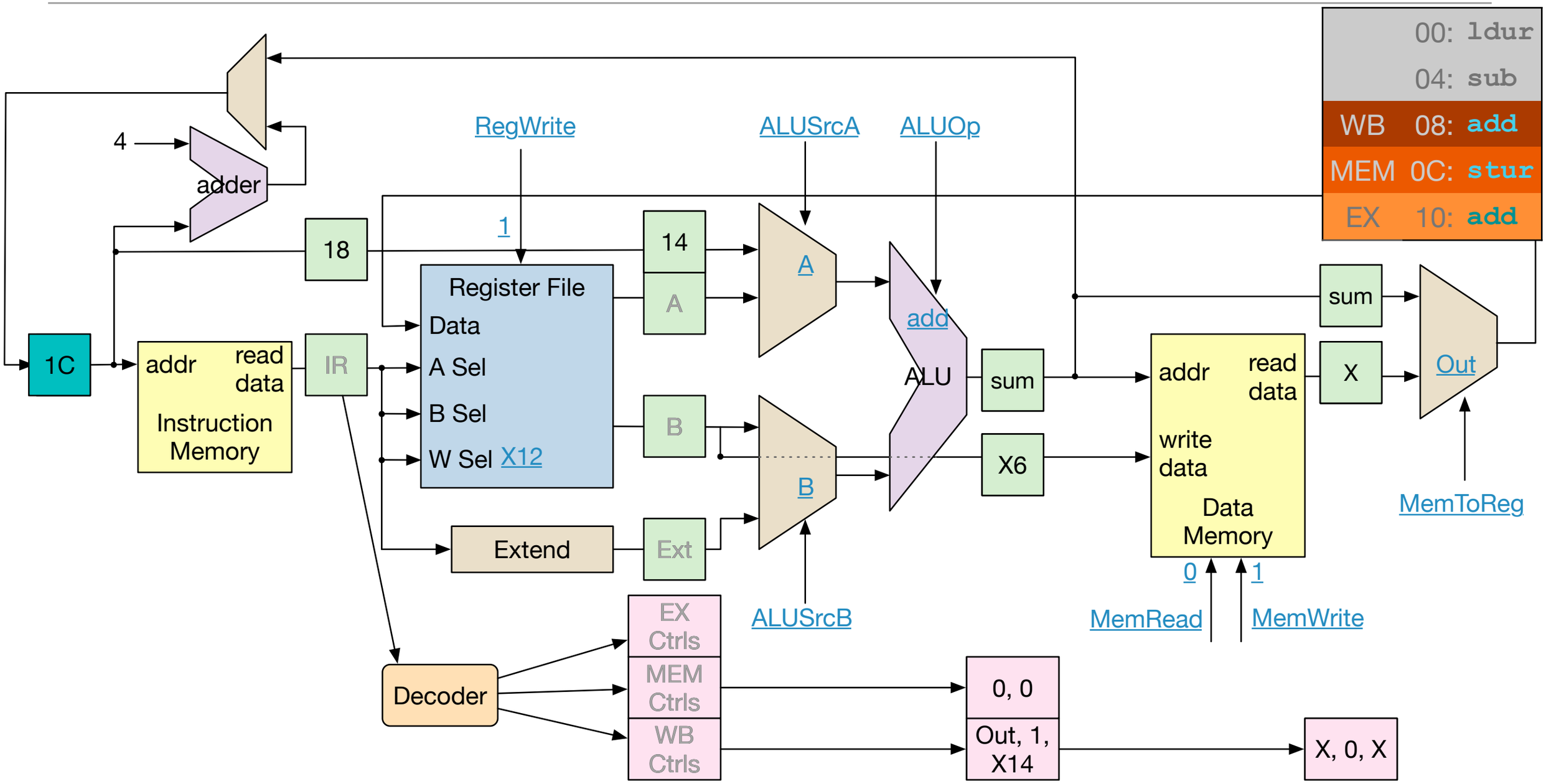
- Fetch 10h; decode **`stur X13, [X1, #48]`**; execute **`add`**; mem access **`sub`** (noop); write **`ldur`**

Example of Pipeline: After Cycle 6



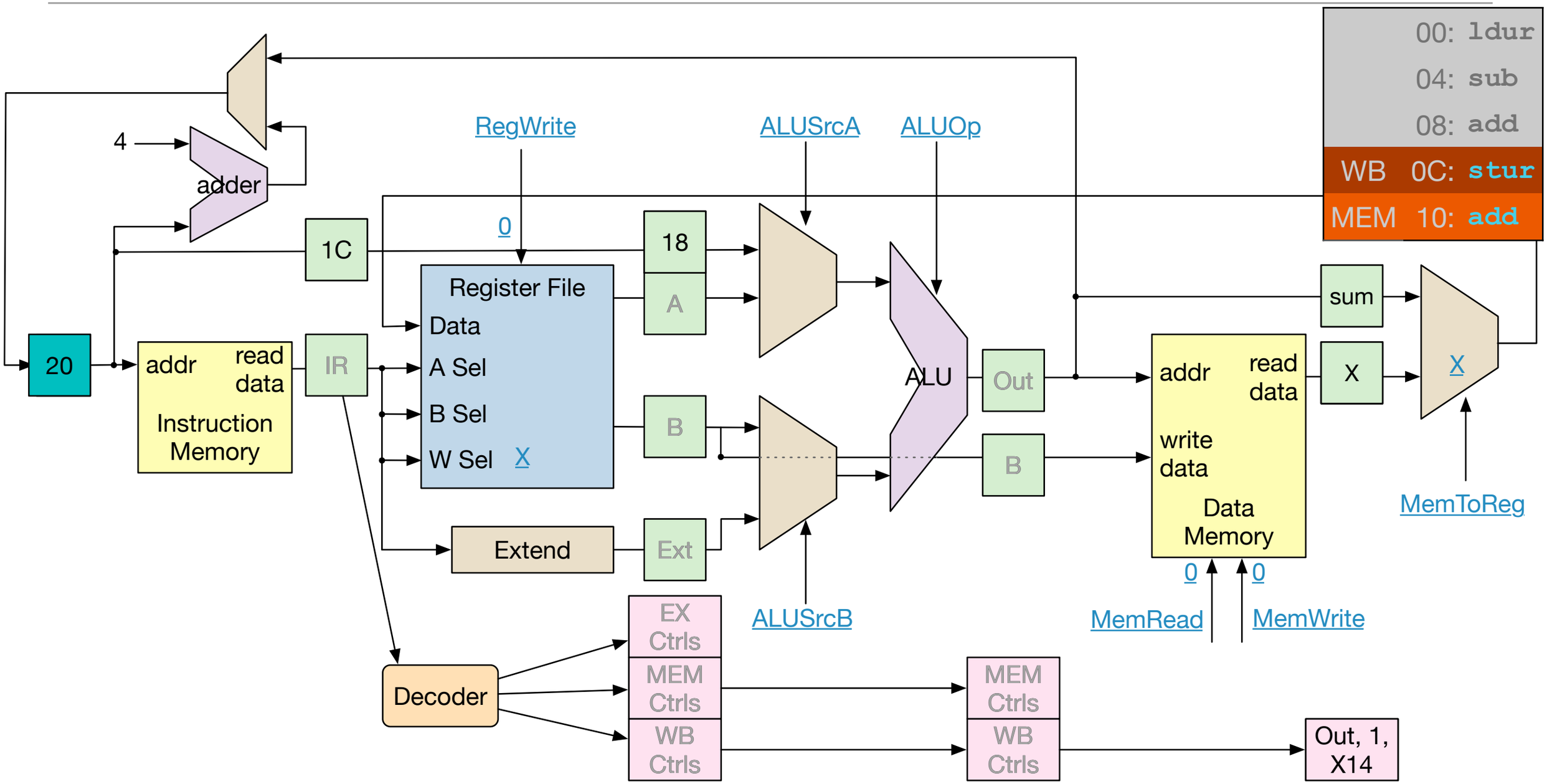
- Decode **`add X14, X5, X6`**; execute **`stur`**; mem access **`add`** (noop); write **`sub`**

Example of Pipeline: After Cycle 7



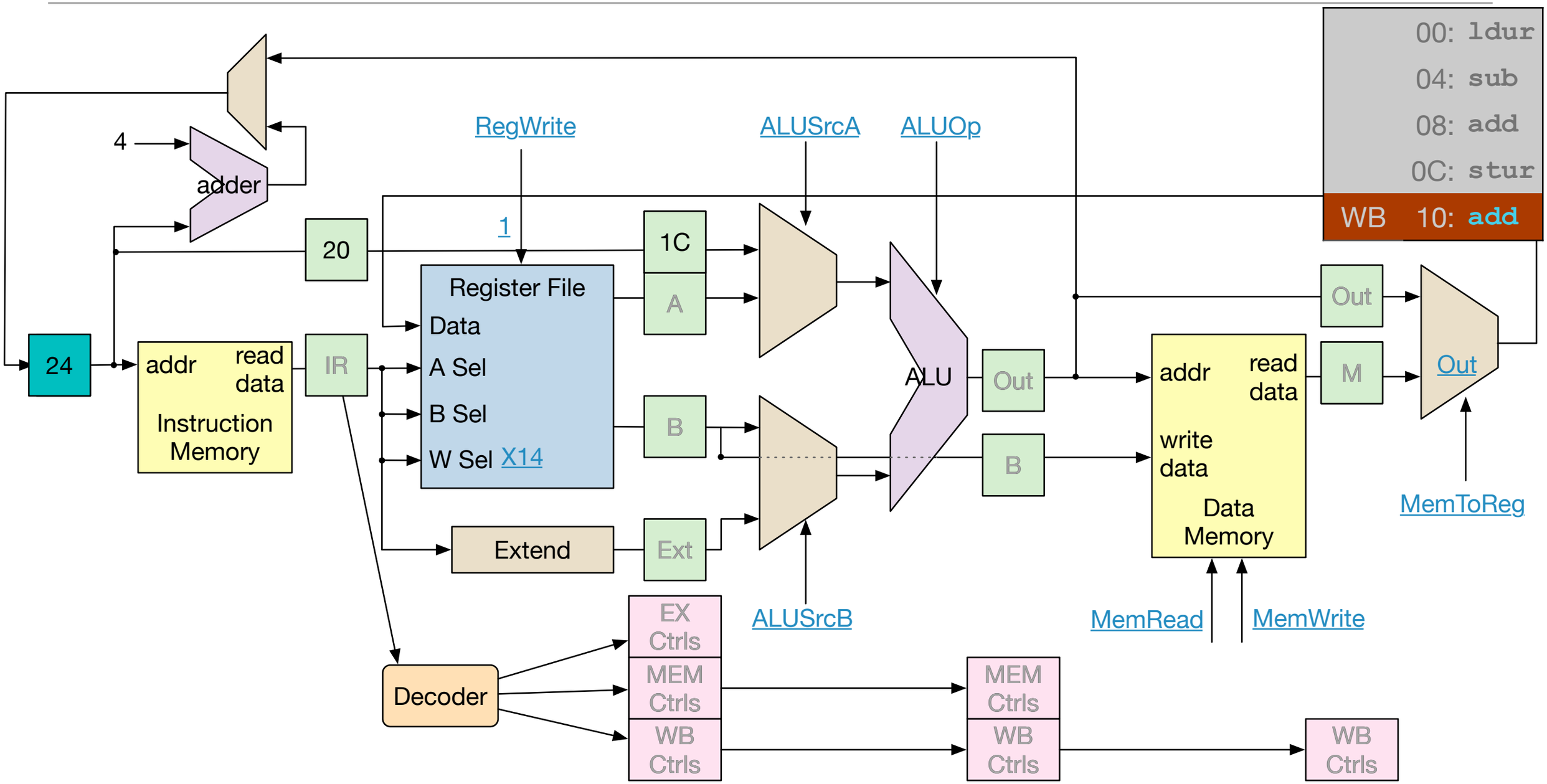
- Execute **`add`**; mem access **`stur`**; write **`add`**

Example of Pipeline: After Cycle 8



- Mem access **`add`** (noop); write **`stur`** (noop)

Example of Pipeline: After Cycle 9



- Write **add**