

VI - Arch Basic & SEQ

ICS 26Fall

Junqi Xu 2026.10.14

Recall

Concepts:

Caller-Saved Registers

Callee-Saved Registers

%r10, %r11

RFLAGS, Syscall

Virtual Registers

Basic Blocks, CFG, CG

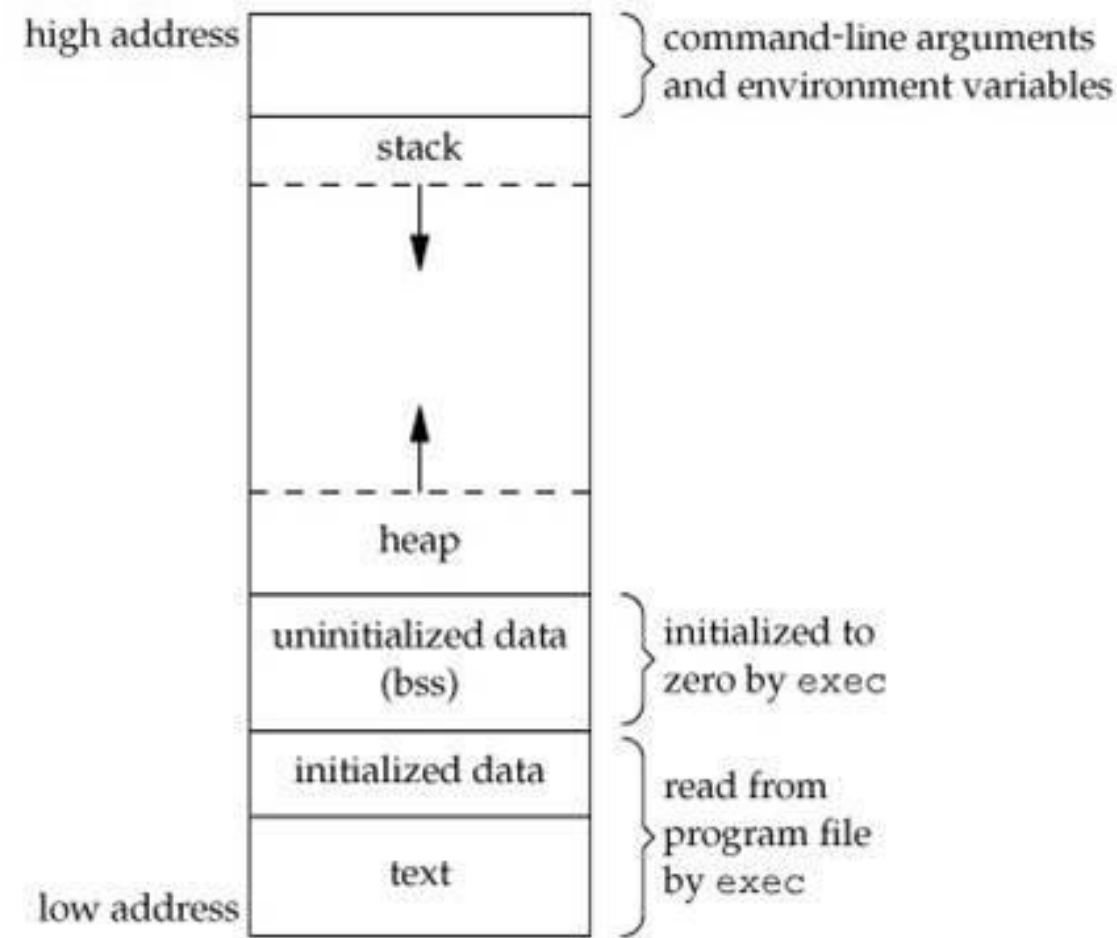


Fig.1^[1]

[1] Stevens, W. R., & Rago, S. A. (2013). Advanced Programming in the UNIX Environment (3rd ed.). Addison-Wesley.

Recall

Concepts:

Caller-Saved Registers

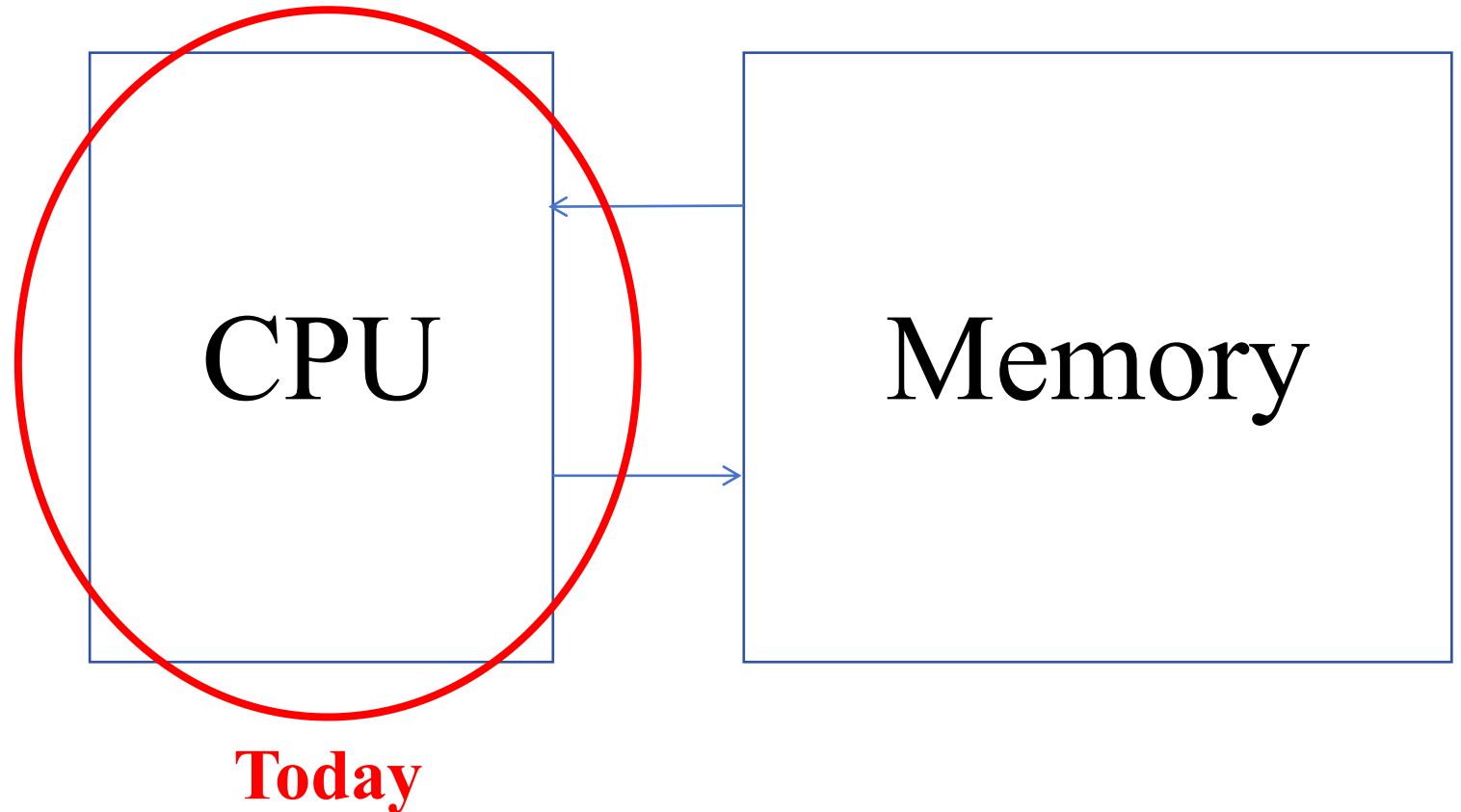
Callee-Saved Registers

%r10, %r11

RFLAGS, Syscall

Virtual Registers

Basic Blocks, CFG, CG



Topics

- Intro - How programs change the world?
- Computer Architecture Basic
- Sequential Implementation

Intro - How programs change the world?

Programs must be executed: Static programs can't do anything!

What do programs do? (In Von Neumann Architecture)

- Programs “control” the operation of processors.
- Programs contain a set of **instructions** to be executed.
- Programs define the **order** to execute the instructions.

Programs are guides!

Intro - How programs change the world?

Programs change the world through **instructions**:

An instruction may read, write or compute something...

```
rrmovq %1, %2  
rmmovq %1, 0(%2)  
addq %1, %2  
je .L1
```

Only **writes** do make a change.

Intro - How programs change the world?

Not all writes make sense:

- Write registers that would never be read.
- Write memory spaces whose address would never be accessed.

If we could assert that a write doesn't make sense, we could eliminate it:

- Easy for registers writes (registers are local).
- Hard for memory writes (memory are global, value of a pointer is arbitrary).

Intro - How programs change the world?

Local writes can't change the world!

- As programmers, we don't care and actually can't know value in registers or memory...
- Value in memory and registers would be refreshed when we enter a new progress.
- We use **print** to get the value. (System I/O, which actually writes a file.)
- **Syscalls** make it possible.
- Memory is volatile, while disc (files) is non-volatile.

Intro - How programs change the world?

Syscalls and storages are interesting, but not for today.

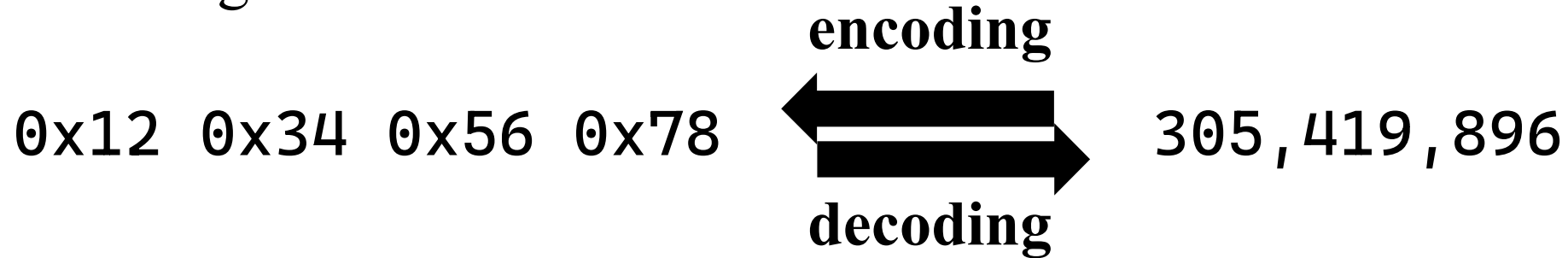
An instruction may read, write or compute something.

To be exact, an instruction **guide** the **processor** to read, write or compute something.

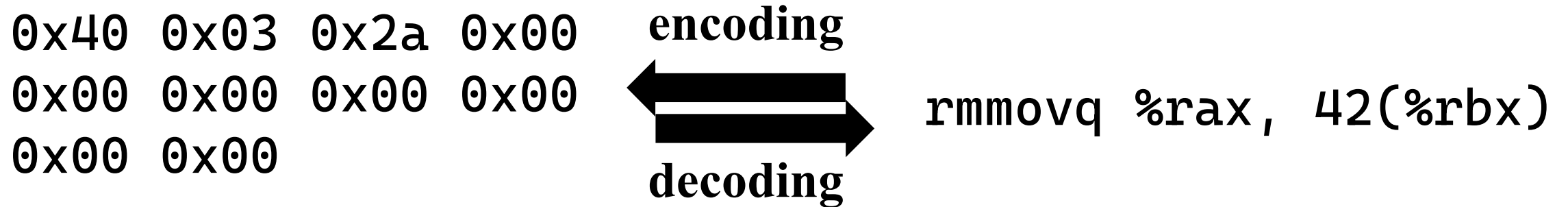
We are talking about how a processor **execute** an instruction.

Computer Architecture Basic - Instruction Encodings

Encoding is not a new technical term:



All information has to be encoded into bytes, including instructions:



Check CSAPP for details.

Computer Architecture Basic - Instruction Encodings

icode: determine rules processors use to decode instructions.

- length, field offsets, ...

ifun: denote extra information.

Like the structure, an instruction has many fields. Instructions with different **icodes** have different **layouts**.

Bytes ordering only works inside a field, just like the structure.

Computer Architecture Basic - RISC

Reduced Instruction Set Computer

- Instructions in RISC are more “atomic”.
- y86 is (more like) a RISC ISA

RISC is the mainstream now.

RISC-V is the most popular RISC ISA now.

If you want to learn RISC-V, take Prof. Zhang’s architecture course!

Computer Architecture Basic - RISC

Reduced Instruction Set Computer

- Instructions in RISC are more “atomic”.
- y86 is (more like) a RISC ISA

RISC is the mainstream now.

RISC-V is the most popular RISC ISA now.

If you want to learn RISC-V, take Prof. Zhang's architecture course!



Computer Architecture Basic - R&W&C

Read&Writes (Loads & Stores) are the most basic operations of a processor.

Computes are based on R&Ws.

ALU provides implementations of computes, we just treat them as miracles.

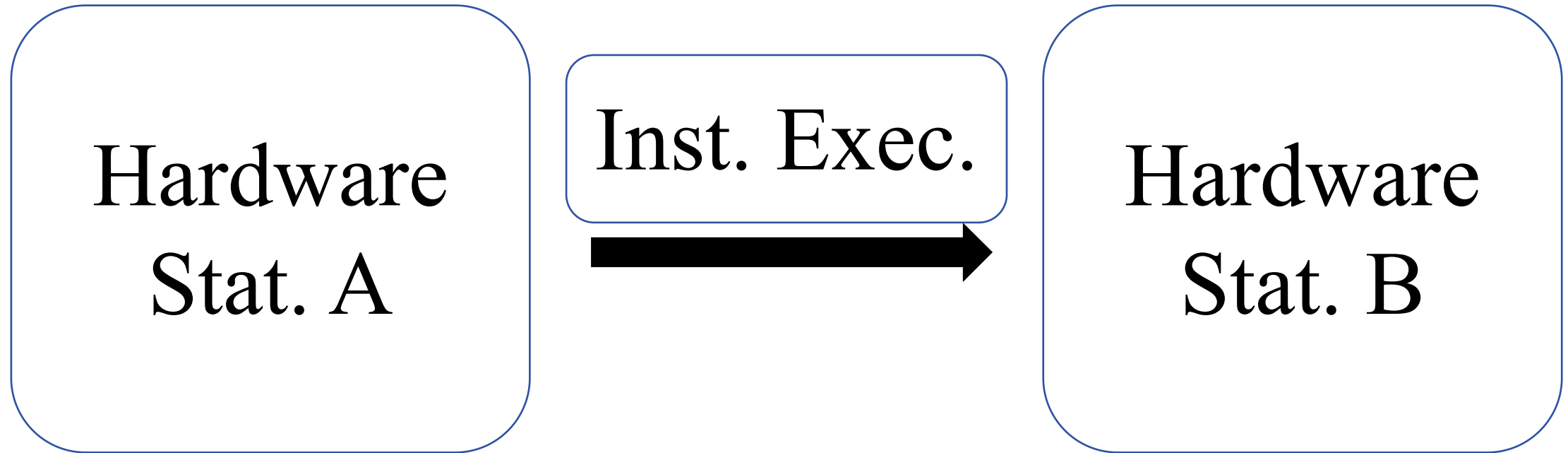
rmmovq %1, 42(%2)

R = { %1, %2 } $\cup$ { %rip, M[%rip] }

C = { (+, 42, R[%2]) } $\cup$ { (+, size(instr), R[%rip]) }

W = { M[(+, 42, R[%2])] } $\cup$ { %rip }

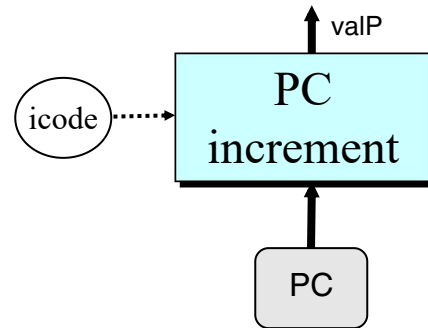
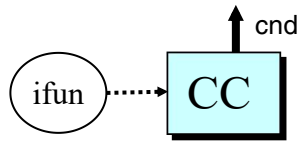
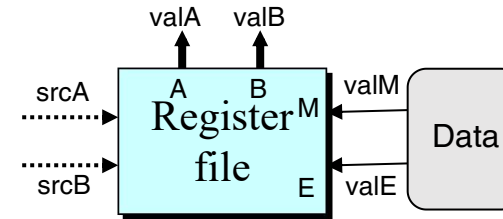
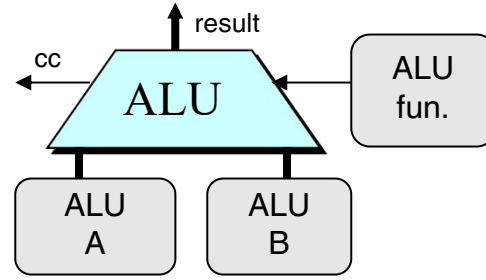
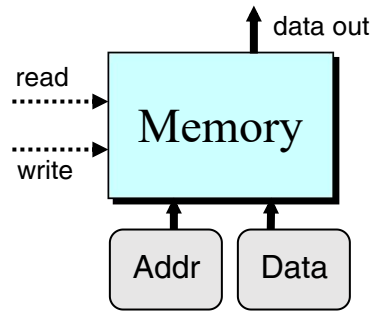
Computer Architecture Basic - Instruction Abstraction



Again, remember, only **writes** change the world.

An instruction writing nothing is equal to NOP.

Computer Architecture Basic - Hardware Units



Sequential Implementation - Motivation

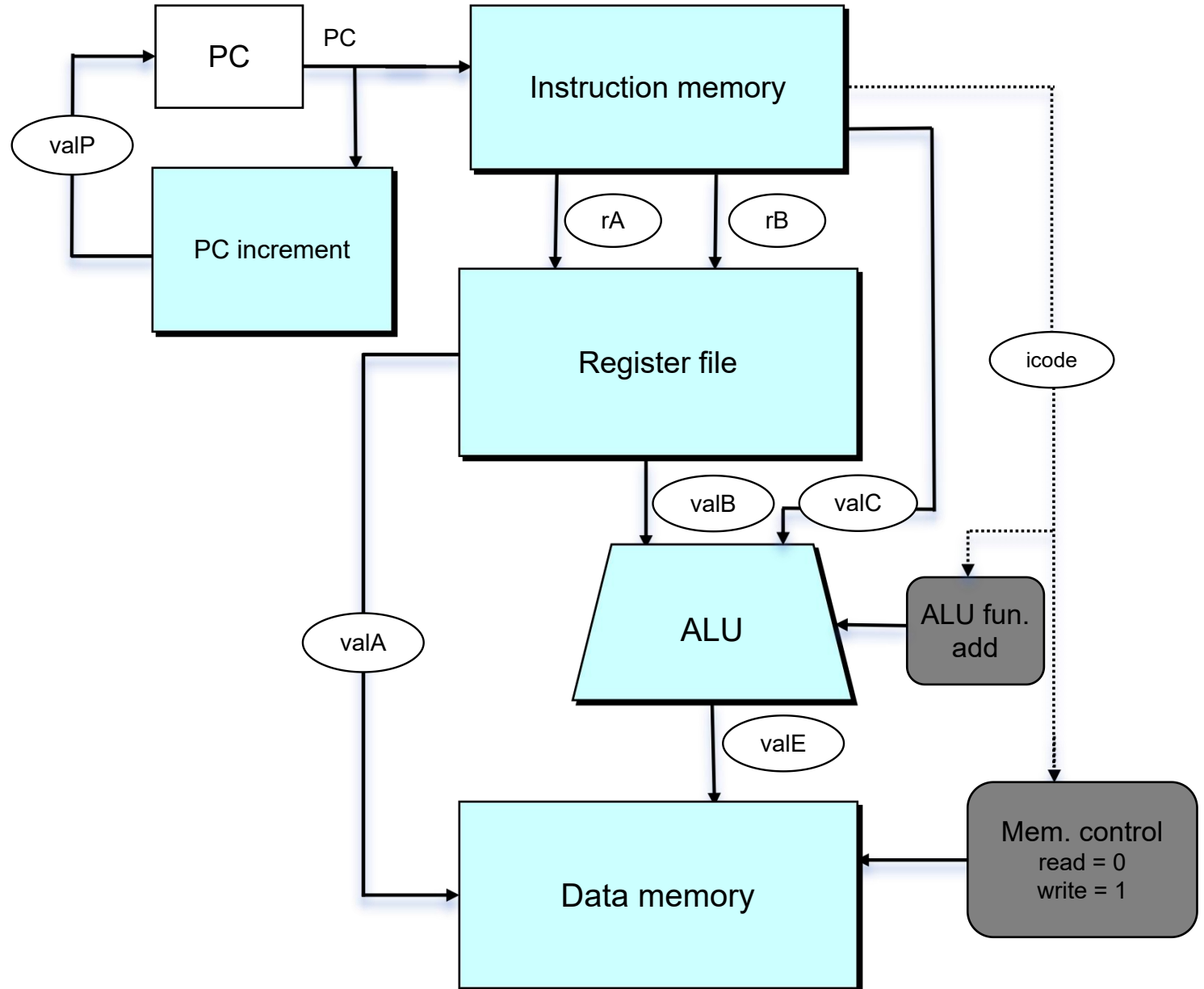
A trivial idea to handle ISA is implementing each single inst. and distributing different inst. to different implementation.

We need to:

1. Get the address of the instruction to be executed. (PC)
2. Access the address and get the encoding. (Fetch)
3. Decode the inst. encoding to figure out which instruction it is. (Decode)
4. Send the decoding results to according implementation.

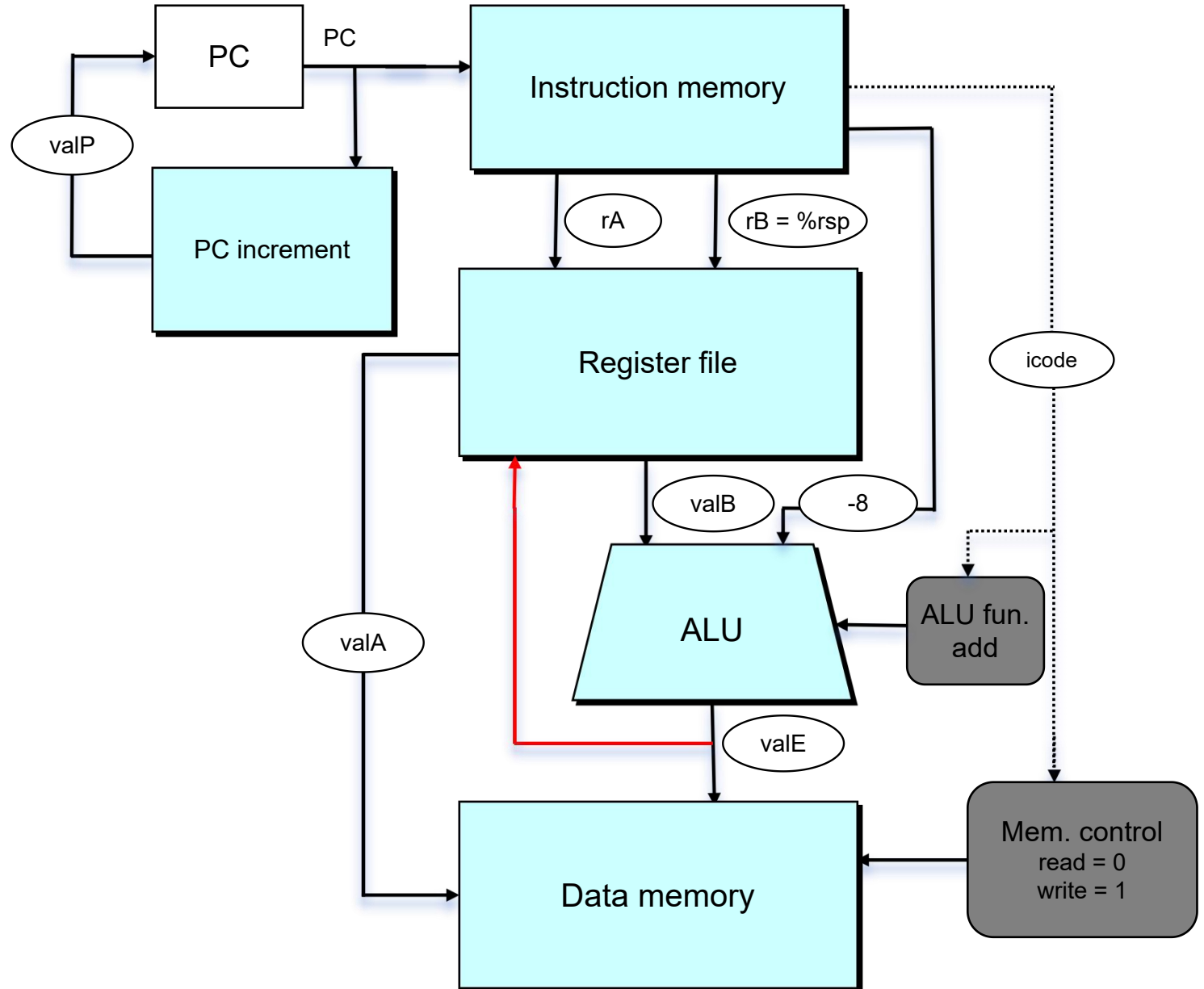
Sequential Implementation - ~~r~~mmovq Implementation

0	icode 4 bit · 0100 (0x4)	ifun 4 bit · 0000 (0x0)
1	rA 4 bit	rB 4 bit
2	D[7:0] · 8 bit	
3	D[15:8] · 8 bit	
4	D[23:16] · 8 bit	
5	D[31:24] · 8 bit	
6	D[39:32] · 8 bit	
7	D[47:40] · 8 bit	
8	D[55:48] · 8 bit	
9	D[63:56] · 8 bit	



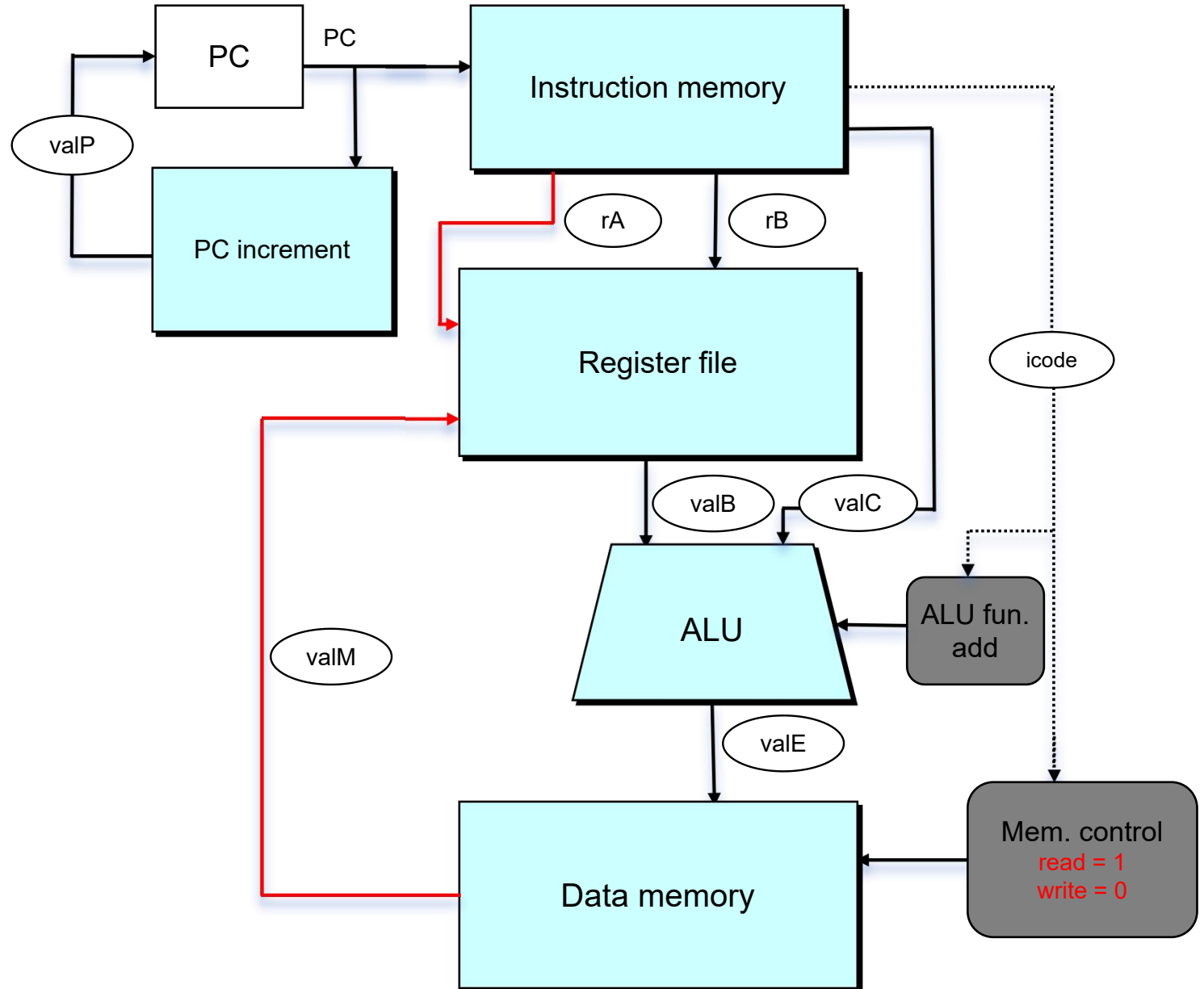
Sequential Implementation - pushq Implementation

0	icode 4 bit · 1010 (0xA)	ifun 4 bit · 0000 (0x0)
1	rA 4 bit	None 4 bit · 1111



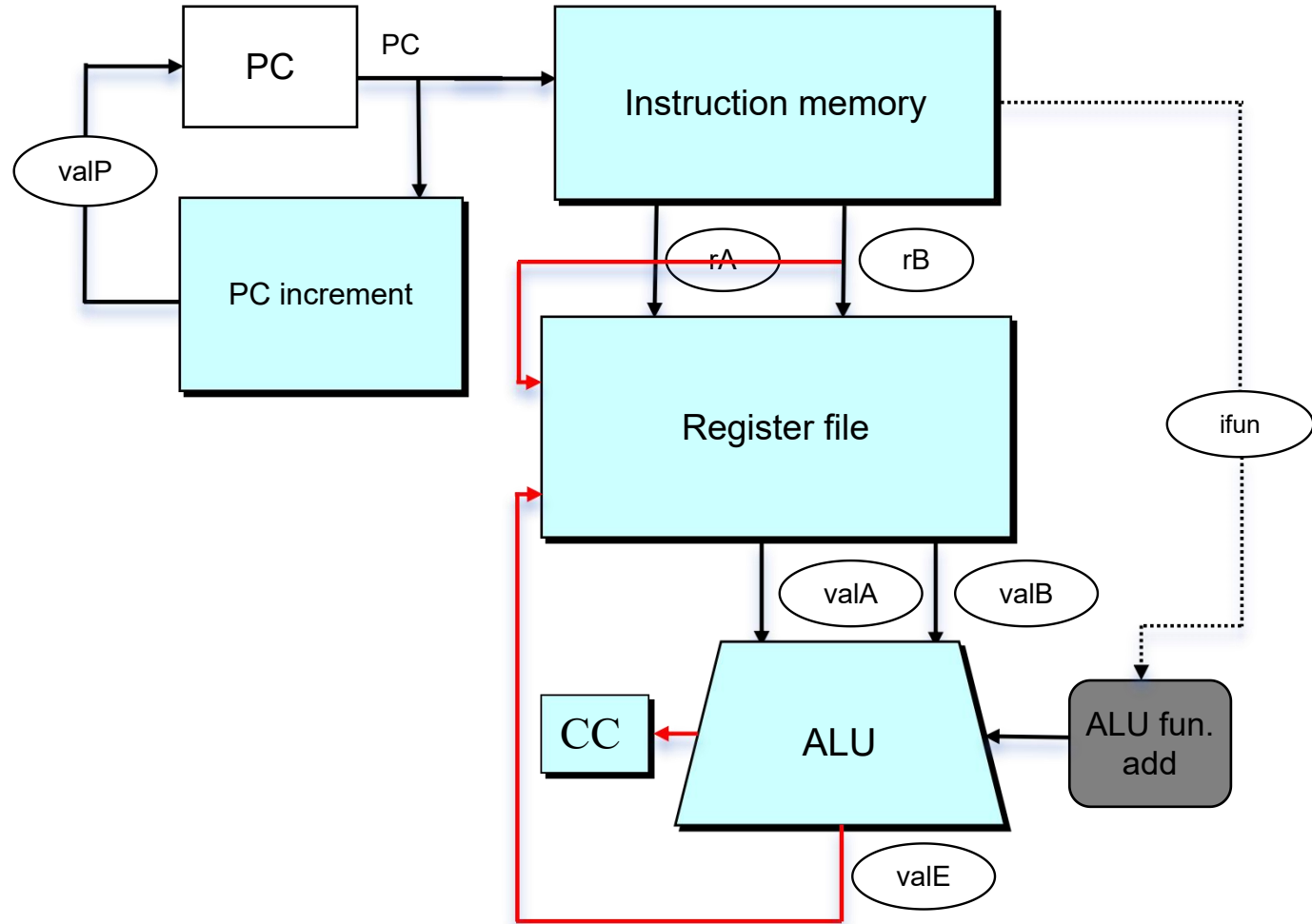
Sequential Implementation - `rrmovq` Implementation

0	icode 4 bit · 0101 (0x5)	ifun 4 bit · 0000 (0x0)
1	rA 4 bit	rB 4 bit
2	D[7:0] · 8 bit	
3	D[15:8] · 8 bit	
4	D[23:16] · 8 bit	
5	D[31:24] · 8 bit	
6	D[39:32] · 8 bit	
7	D[47:40] · 8 bit	
8	D[55:48] · 8 bit	
9	D[63:56] · 8 bit	



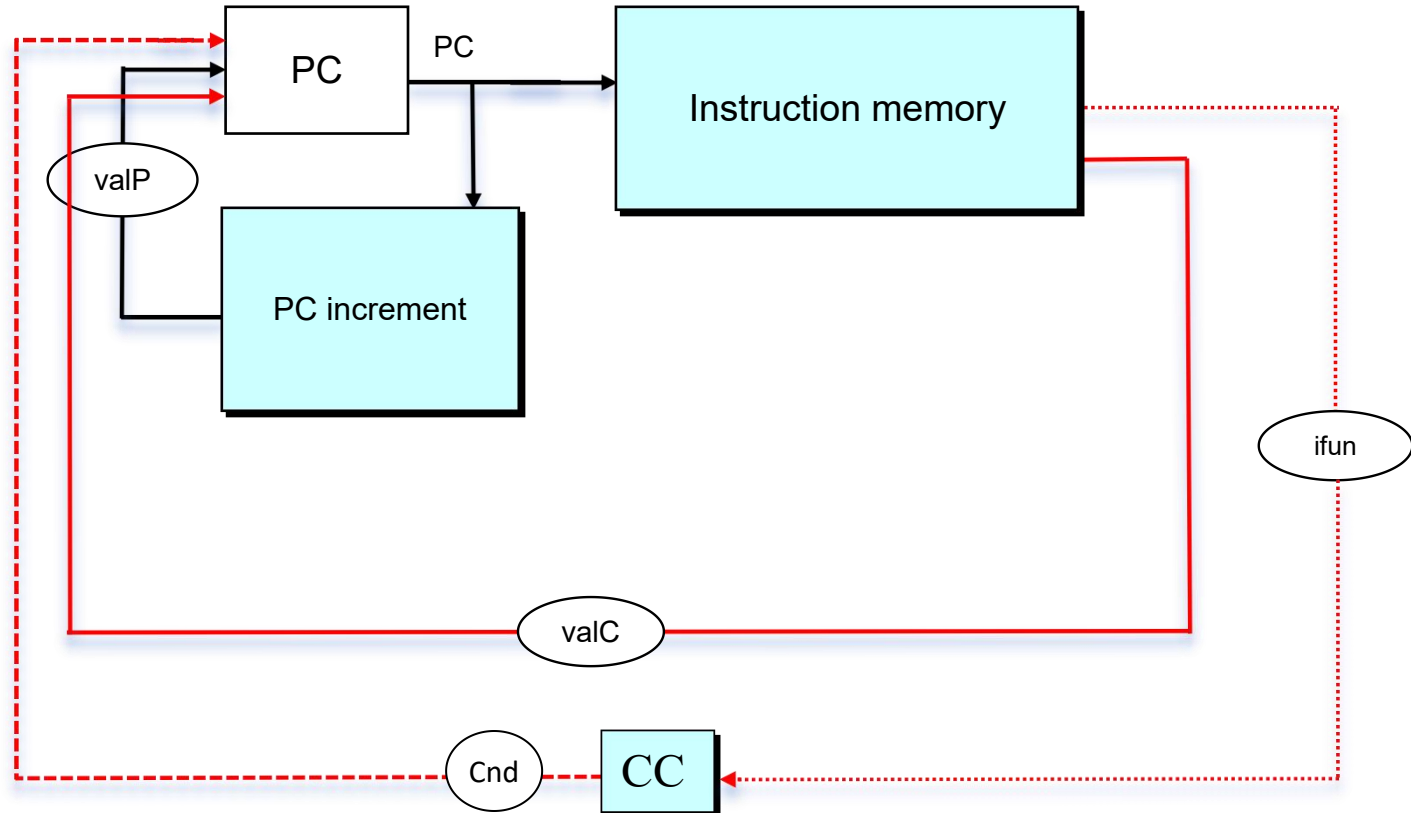
Sequential Implementation - addq Implementation

0	icode 4 bit · 0110 (0x6)	ifun 4 bit · 0000 (0x0)
1	rA 4 bit	rB 4 bit

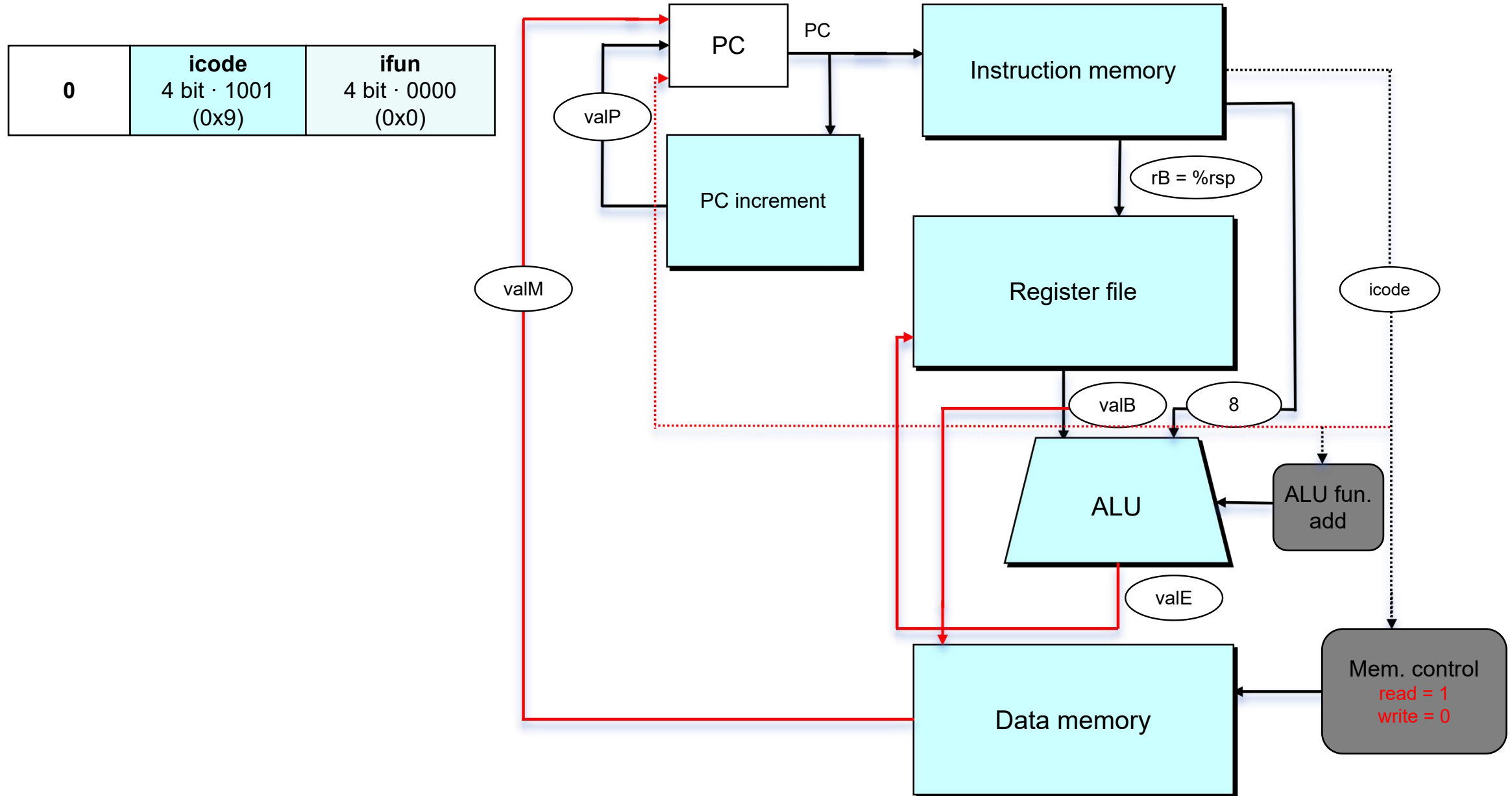


Sequential Implementation - jne Implementation

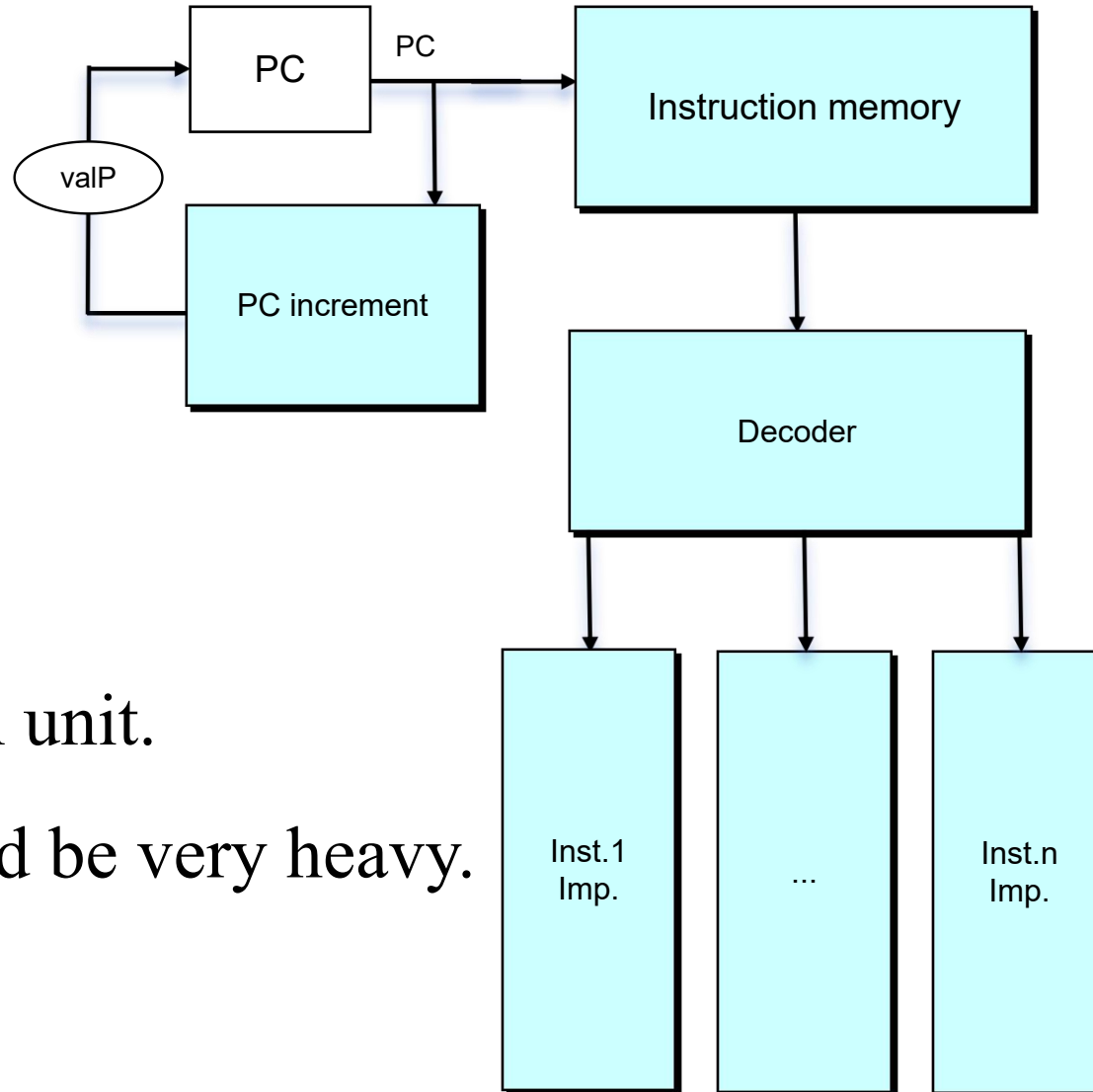
0	icode 4 bit · 0111 (0x7)	ifun 4 bit · 0100 (0x4)
1	D[7:0] · 8 bit	
2	D[15:8] · 8 bit	
3	D[23:16] · 8 bit	
4	D[31:24] · 8 bit	
5	D[39:32] · 8 bit	
6	D[47:40] · 8 bit	
7	D[55:48] · 8 bit	
8	D[63:56] · 8 bit	



Sequential Implementation - **ret** Implementation



Sequential Implementation - SEQ- Implementation

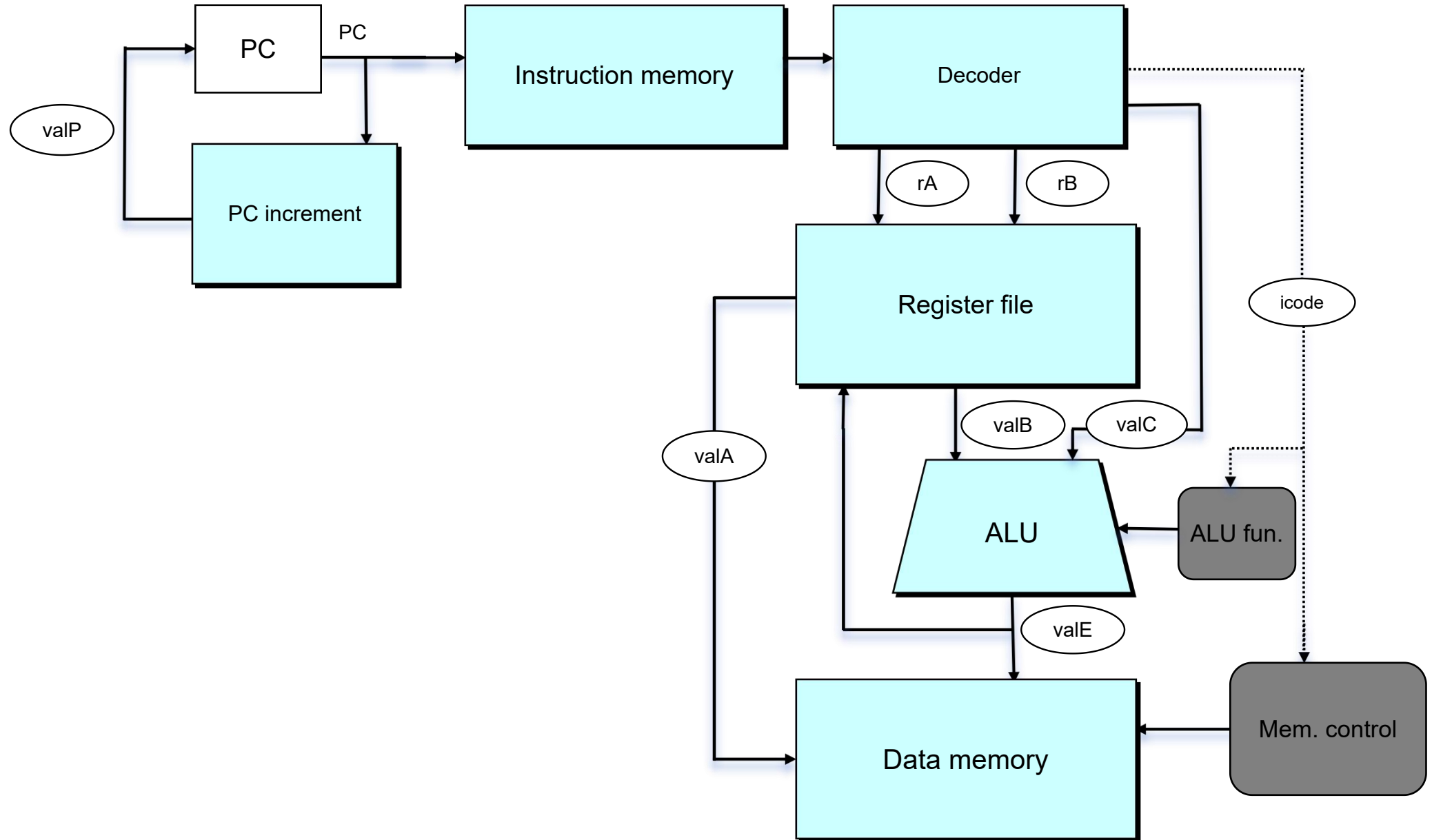


Cons:

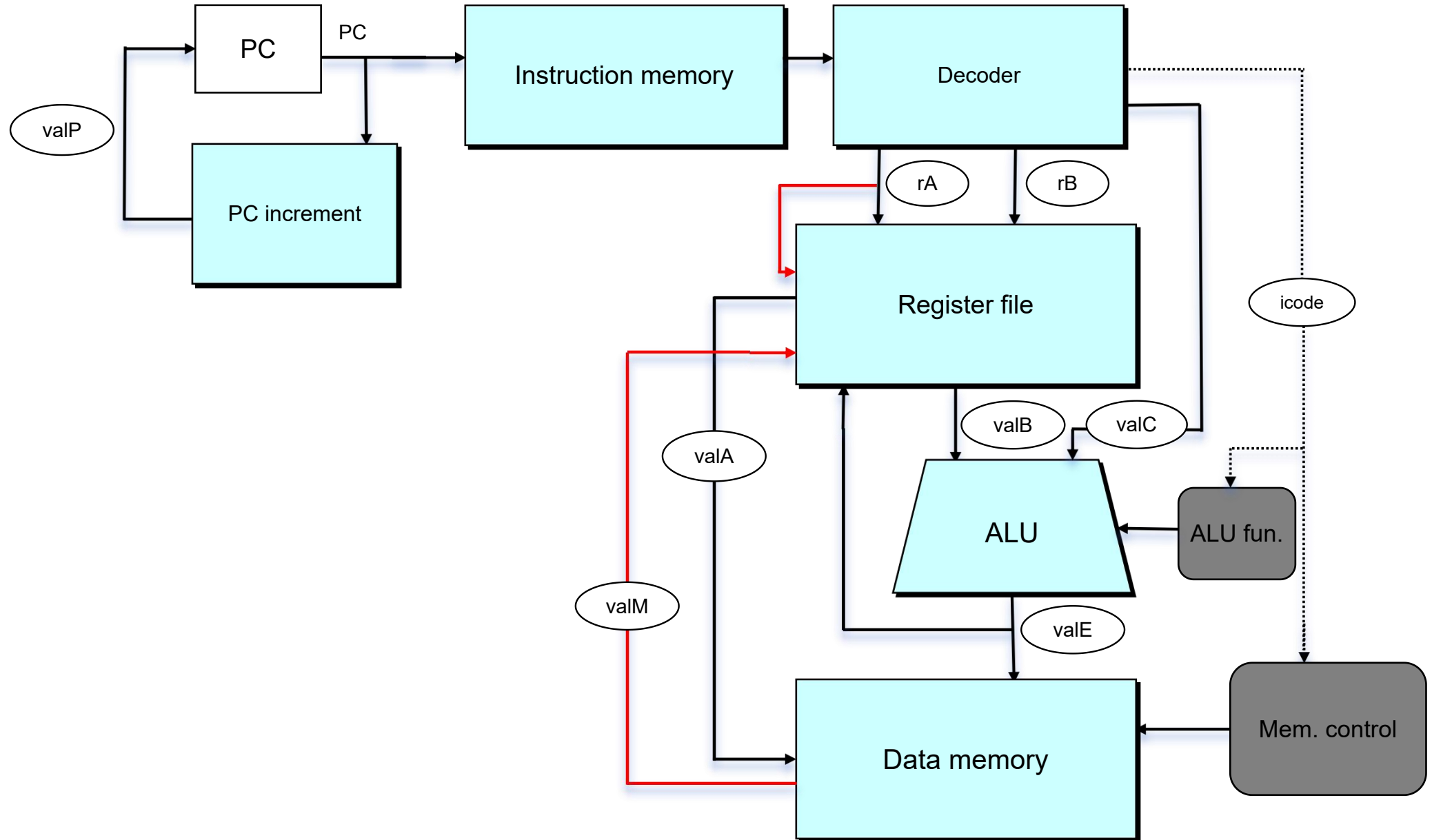
Need about n copies of each unit.

If n is large, processor would be very heavy.

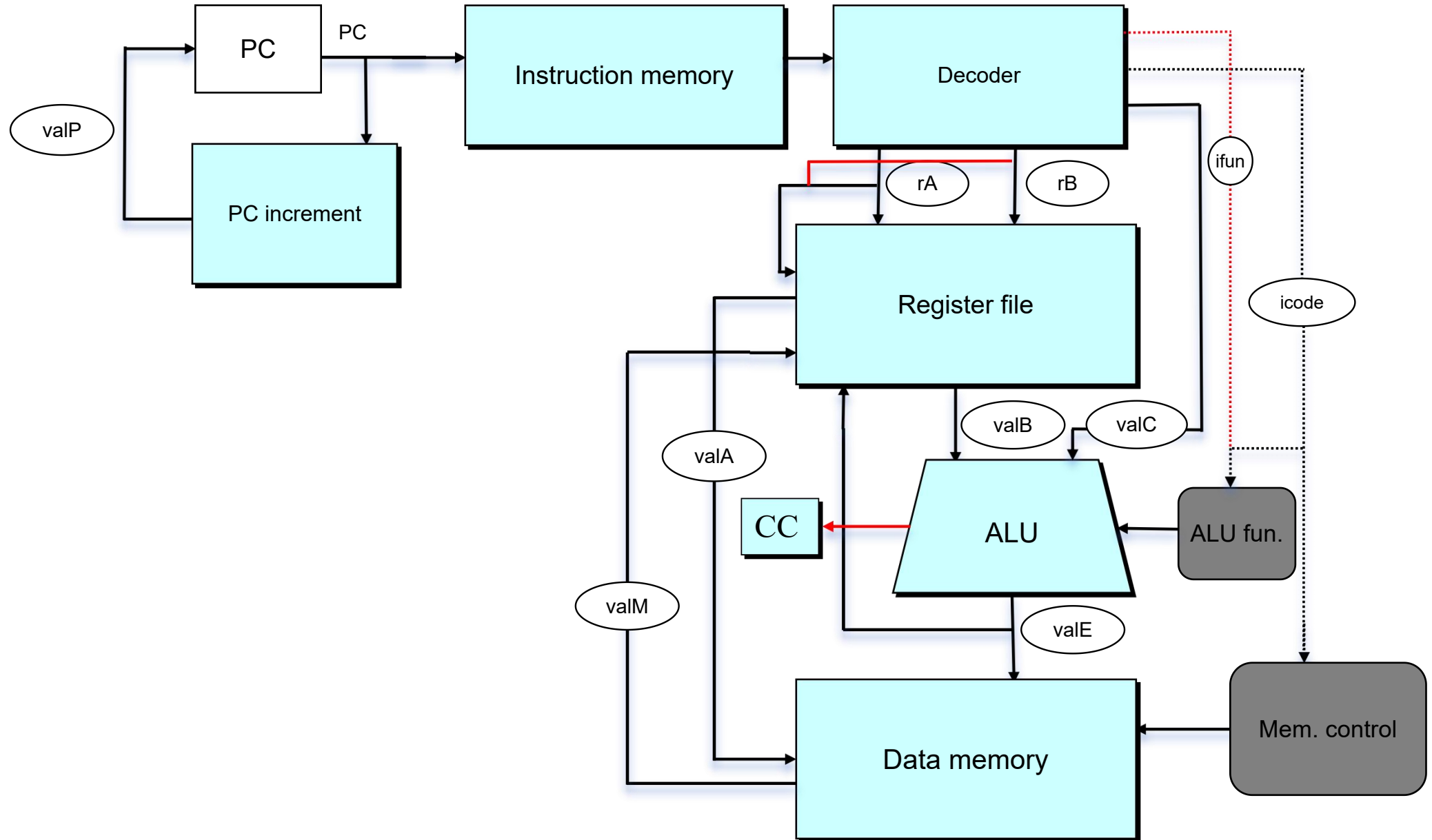
Sequential Implementation - Sharing: **rmmovq + pushq**



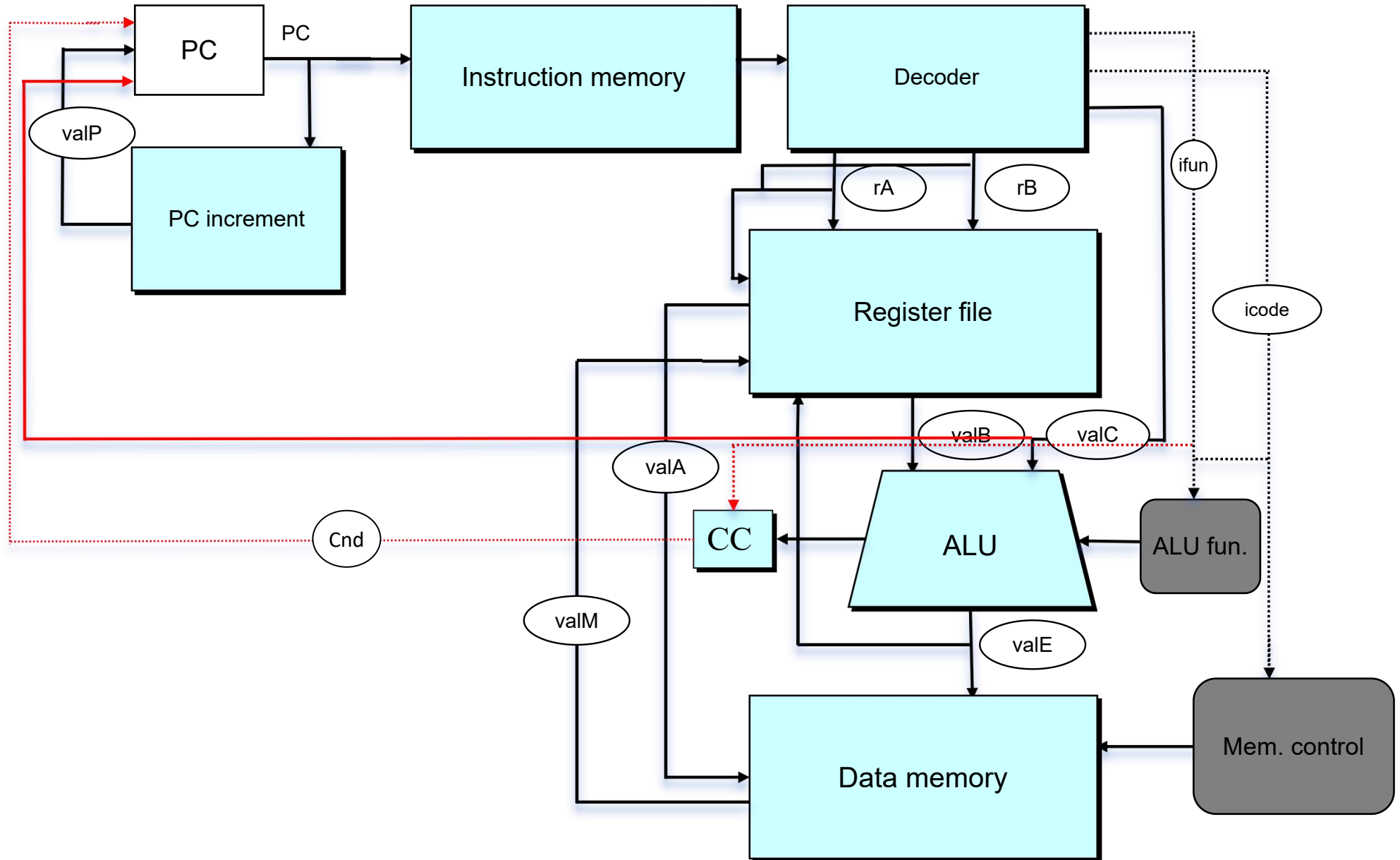
Sequential Implementation - Sharing: + mrmovq



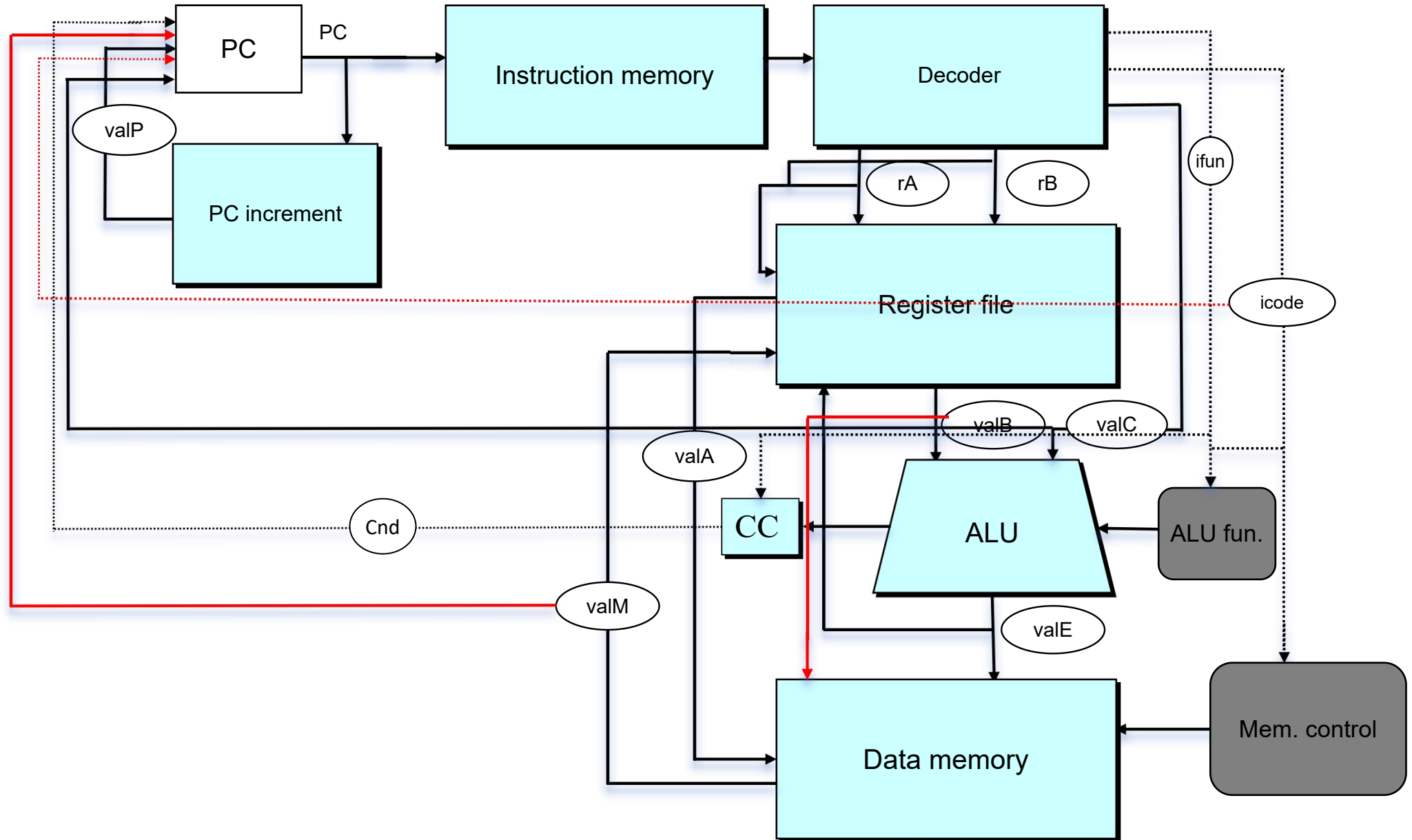
Sequential Implementation - Sharing: + addq



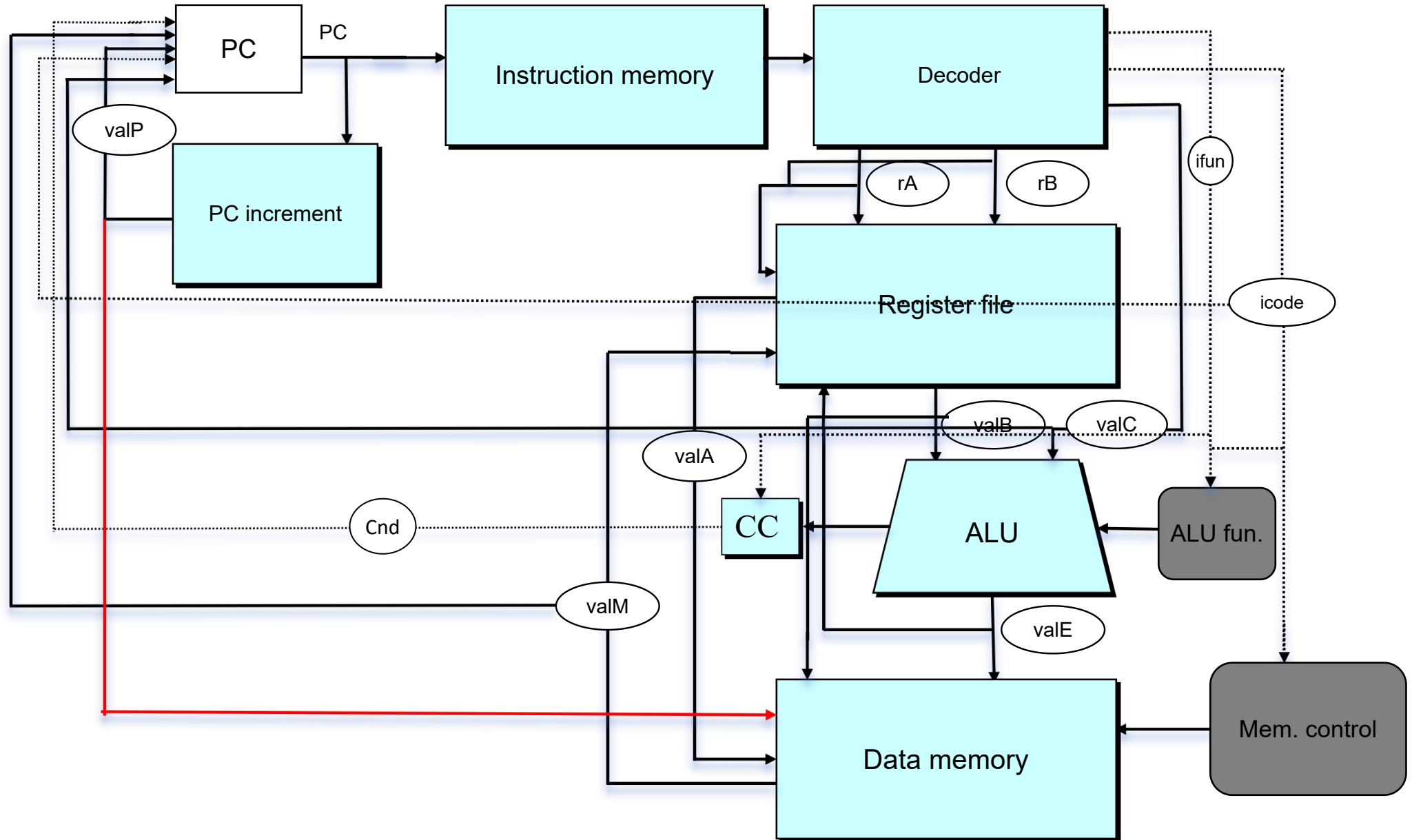
Sequential Implementation - Sharing: + jne



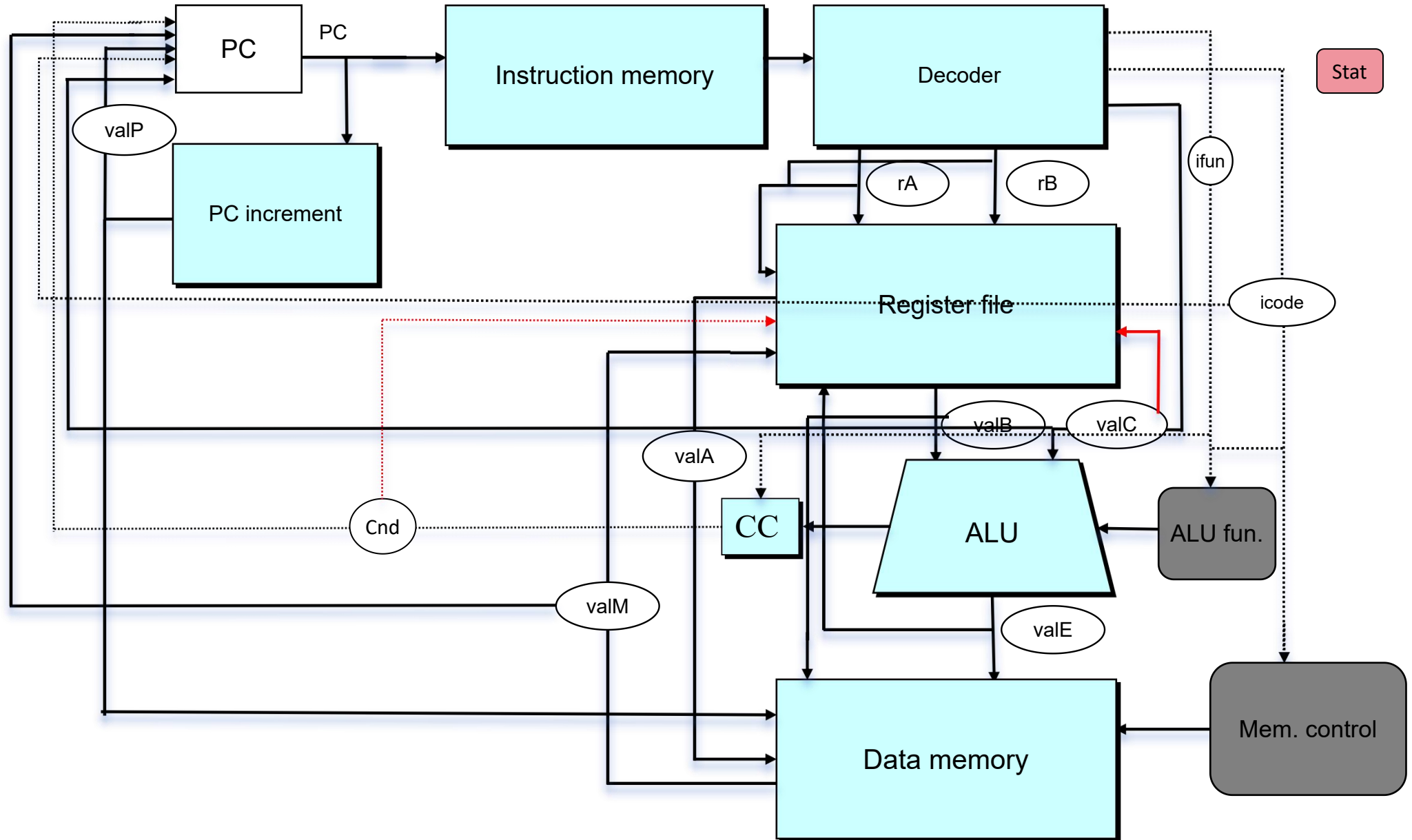
Sequential Implementation - Sharing: + ret



Sequential Implementation - Sharing: + call



Sequential Implementation - SEQ+ Implementation



Sequential Implementation - SEQ+ Implementation

Pros:

Reuse units.

Cons:

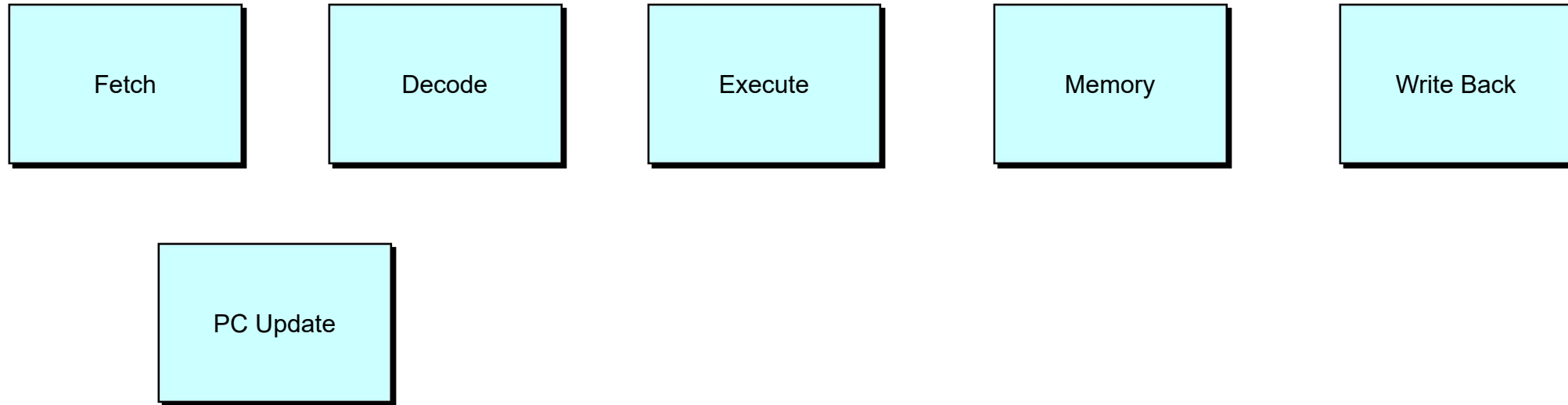
Literally, different inst. has different time to process.

Bottleneck becomes the slowest inst..

When a simple inst. is being processed, most units are on strike...

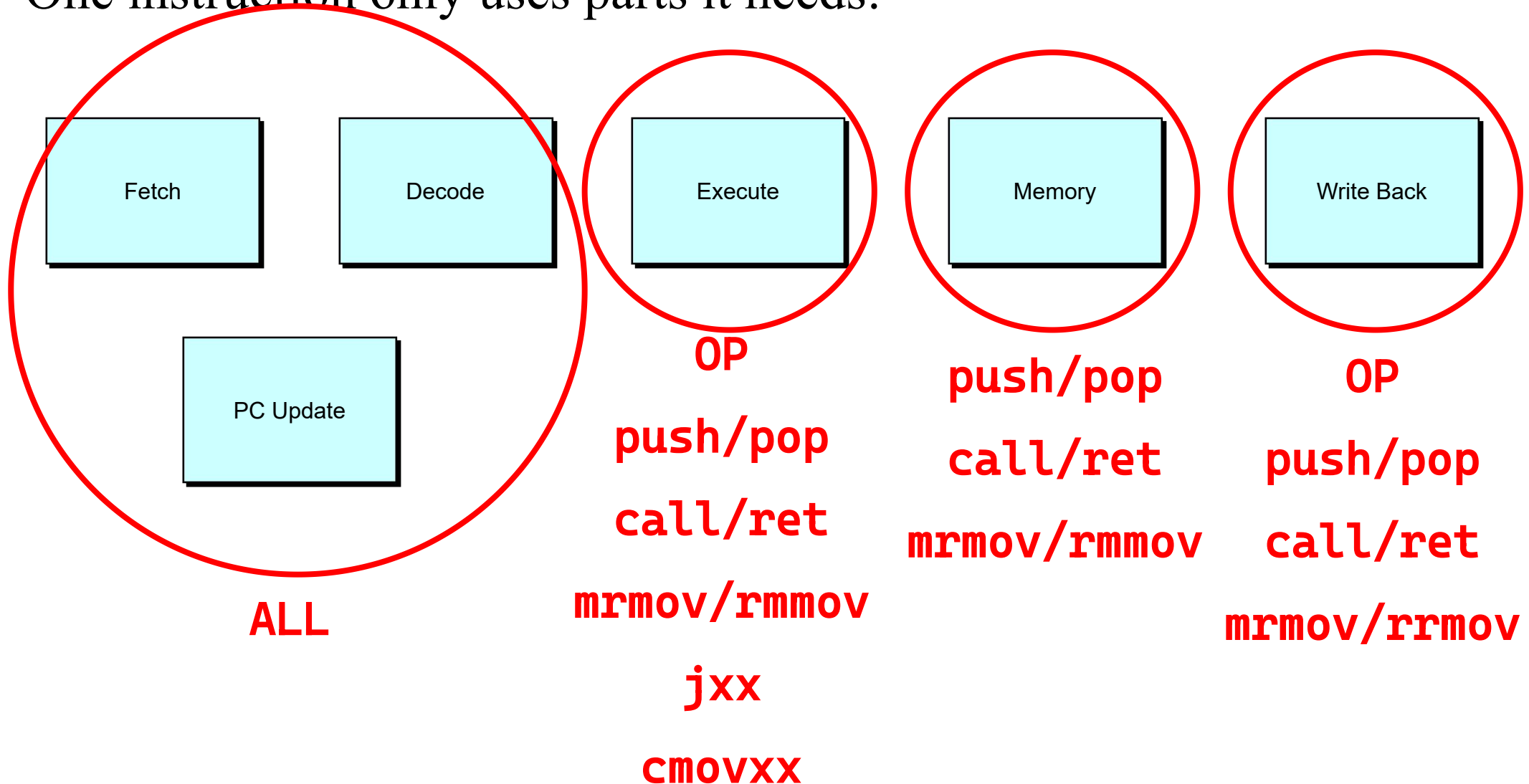
Sequential Implementation - Multi-Cycle

We can divide units into different parts:



Sequential Implementation - Multi-Cycle

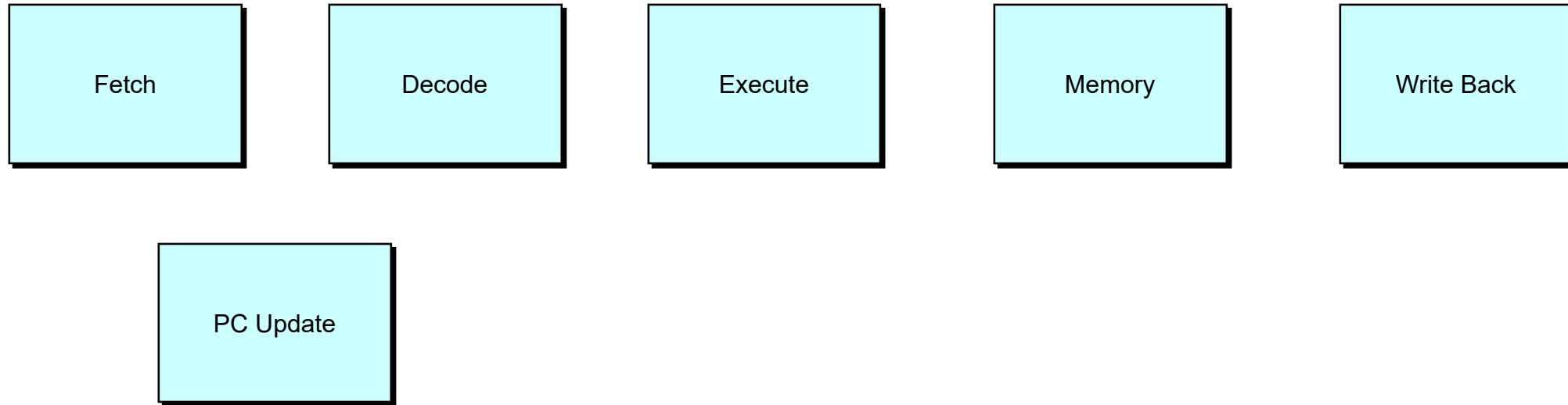
One instruction only uses parts it needs:



Sequential Implementation - Multi-Cycle

Cons: When we decode an inst., units in other parts are on strike...

Solution: Pipeline



Takeaway

Concepts:

Instruction Encodings

RISC vs. CISC

Sequential Implementation