

VII - Pipeline Basic

ICS 26Fall

Junqi Xu 2026.10.21

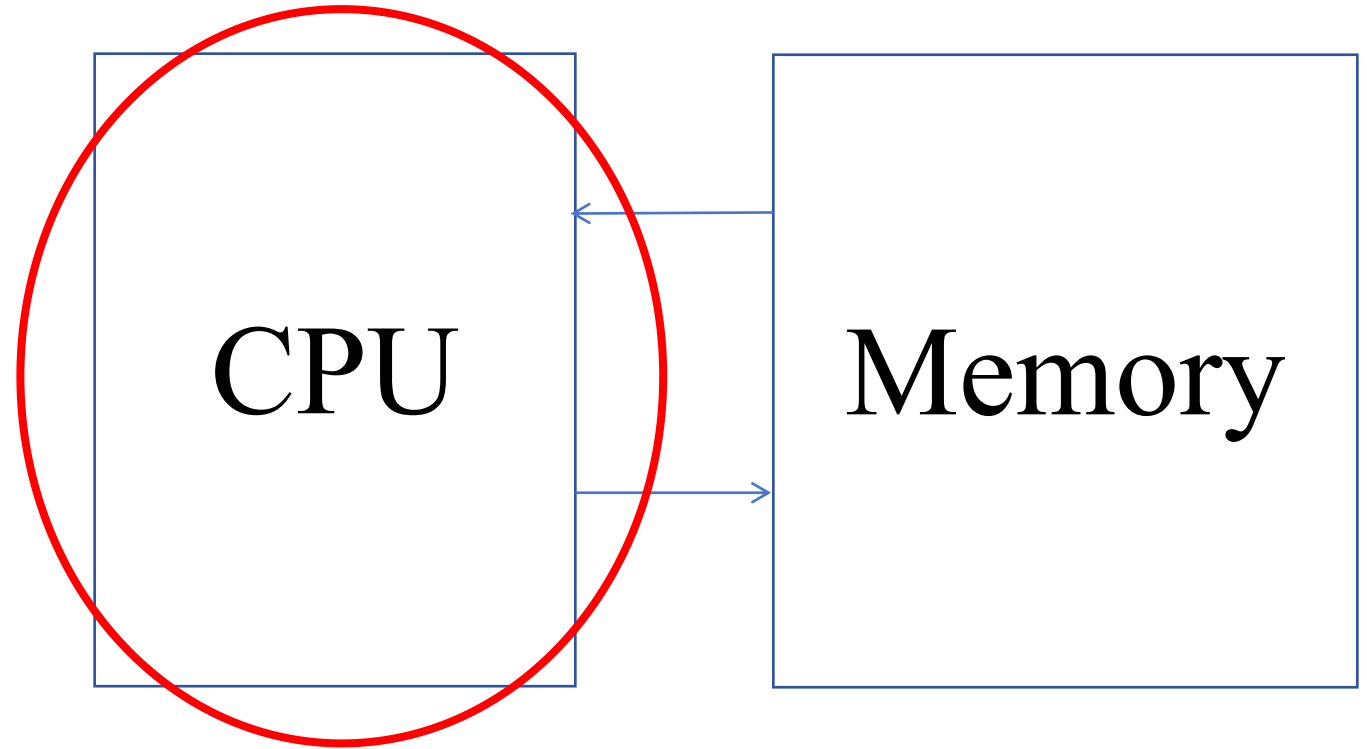
Recall

Concepts:

Instruction Encodings

RISC vs. CISC

Sequential Implementation



Topics

- Intro: Pipelining
- Dependancy
- Error Handling

Pipelining - Intro

SEQ Runtime

1: `movq (%1), %2`

2: `movq (%3), %4`

3: `addq %2, %4`

4: `movq %4, (%5)`

Cycle: 1

Read `%rip`

1: `movq (%1), %2`

Fetch

Decode

Execute

Memory

Write Back

Pipelining - Intro

SEQ Runtime

1: `movq (%1), %2`

2: `movq (%3), %4`

3: `addq %2, %4`

4: `movq %4, (%5)`

Cycle: 2

Read %1, %2

1: `movq (%1), %2`

Fetch

Decode

Execute

Memory

Write Back

Pipelining - Intro

SEQ Runtime

1: `movq (%1), %2`

2: `movq (%3), %4`

3: `addq %2, %4`

4: `movq %4, (%5)`

Cycle: 3

None

1: `movq (%1), %2`

Fetch

Decode

Execute

Memory

Write Back

Pipelining - Intro

SEQ Runtime

1: `movq (%1), %2`

2: `movq (%3), %4`

3: `addq %2, %4`

4: `movq %4, (%5)`

Cycle: 4

None

1: `movq (%1), %2`

Fetch

Decode

Execute

Memory

Write Back

Pipelining - Intro

SEQ Runtime

1: `movq (%1), %2`

2: `movq (%3), %4`

3: `addq %2, %4`

4: `movq %4, (%5)`

Cycle: 5

Write %2

Write %rip

1: `movq (%1), %2`

Fetch

Decode

Execute

Memory

Write Back

Pipelining - Intro

SEQ Runtime

1: `movq %1, %2`

2: `movq %3, %4`

3: `addq %2, %4`

4: `movq %4, (%5)`

Cycle: 6

2: `movq %3, %4`

Fetch

Decode

Execute

Memory

Write Back

Pipelining - Intro

SEQ Runtime

1: `movq (%1), %2`

2: `movq (%3), %4`

3: `addq %2, %4`

4: `movq %4, (%5)`

Cycle: 7

2: `movq (%3), %4`

Fetch

Decode

Execute

Memory

Write Back

Pipelining - Intro

SEQ Runtime

1: `movq (%1), %2`

2: `movq (%3), %4`

3: `addq %2, %4`

4: `movq %4, (%5)`

Cycle: 20

Fetch

Decode

Execute

Memory

Write Back

4: `movq %4, (%5)`

Pipelining - Intro

Multi-Cycle Runtime

1: `mrmovq (%1), %2`

2: `mrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`

Cycle: 18 (5+5+4+4)

Fetch

Decode

Execute

Memory

Write Back

Pipelining - Intro

Pipeline Runtime

1: mrmovq (%1), %2

2: mrmovq (%3), %4

3: addq %2, %4

4: rmmovq %4, (%5)

Cycle: 1

1: Read %rip

1: Write %rip

1: mrmovq (%1), %2

Fetch

Decode

Execute

Memory

Write Back

Pipelining - Intro

Pipeline Runtime

1: `movq %1, %2`

2: `movq %3, %4`

3: `addq %2, %4`

4: `movq %4, (%5)`

Cycle: 2

2: Read `%rip`

2: Write `%rip`

1: Read `%1, %2`

2: `movq (%3), %4`

1: `movq (%1), %2`

Fetch

Decode

Execute

Memory

Write Back

Pipelining - Intro

Pipeline Runtime

1: `movq (%1), %2`

2: `movq (%3), %4`

3: `addq %2, %4`

4: `movq %4, (%5)`

3: `addq %2, %4`

2: `movq (%3), %4`

1: `movq (%1), %2`

Cycle: 3

3: Read `%rip`

3: Write `%rip`

2: Read `%3, %4`

Fetch

Decode

Execute

Memory

Write Back

Pipelining - Intro

Pipeline Runtime

1: `movq (%1), %2`

2: `movq (%3), %4`

3: `addq %2, %4`

4: `movq %4, (%5)`

Cycle: 4

4: Read `%rip`

4: Write `%rip`

3: Read `%2, %4`

4: `movq %4, (%5)`

3: `addq %2, %4`

2: `movq (%3), %4`

1: `movq (%1), %2`

Fetch

Decode

Execute

Memory

Write Back

Pipelining - Intro

Pipeline Runtime

1: `movq (%1), %2`

2: `movq (%3), %4`

3: `addq %2, %4`

4: `movq %4, (%5)`

4: `movq %4, (%5)`

3: `addq %2, %4`

2: `movq (%3), %4`

1: `movq (%1), %2`

Cycle: 5

4: Read %4, %5

1: Write %2

Fetch

Decode

Execute

Memory

Write Back

Pipelining - Intro

Pipeline Runtime

1: `mrmovq (%1), %2`

2: `mrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`

4: Read %4

1: Write %2

Cycle: 5

Wait, is there anything wrong?

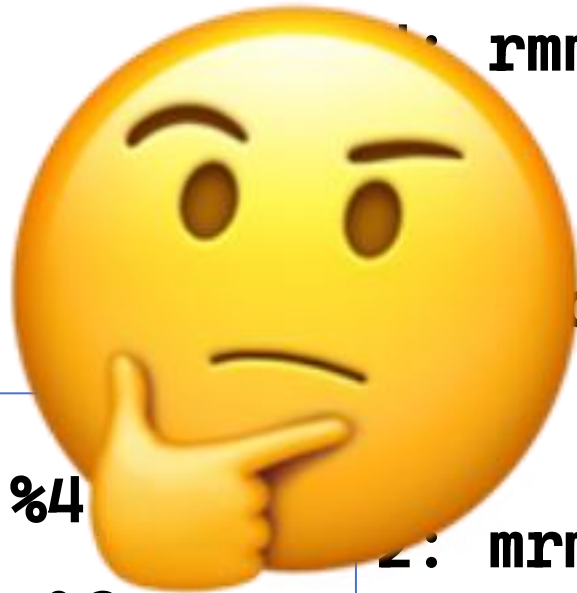
Fetch

Decode

Execute

Memory

Write Back



Pipelining - Intro

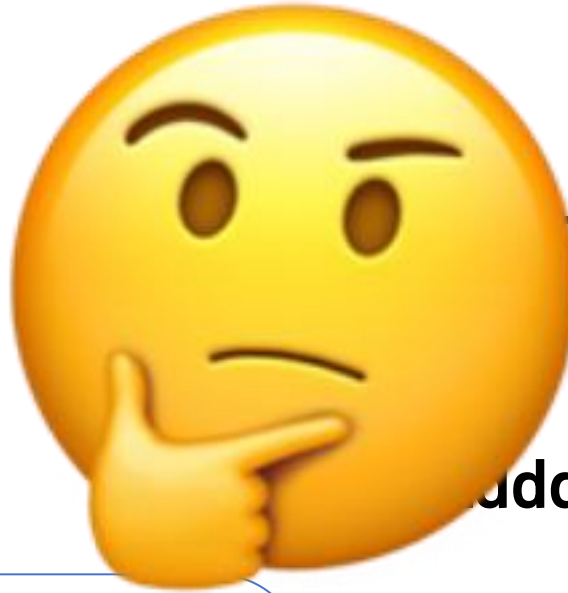
Pipeline Runtime

1: `movq (%1), %2`

2: `movq %2, %4`

3: `addq %2, %4`

4: `movq %4, (%5)`



4: `movq (%5), %4`
Date flows from inst. 1 to inst. 3.

Cycle: 5 1: Write %2

1: `movq (%1), %2`

Fetch

Decode

Execute

Memory

Write Back

Pipelining - Intro

Pipeline Runtime

1: `movq (%1), %2`
2: `movq (%2), %4`
3: `addq %2, %4`
4: `movq %4, (%5)`

Cycle: 5

4: Read %4, %5
1: Write %2

4: `movq %4, (%5)`

3: `addq`

2: `movq`

1: `movq`

Inst. 1 writes %2 now.

Fetch

Decode

Execute

Memory

Write Back



Pipelining - Intro

Pipeline Runtime

1: `movq (%1), %2`
2: `movq (%2), %4`
3: `addq %2, %4`
4: `movq %4, (%5)`

Cycle: 4

4: Read %rip
3: Read %2, %4

4: `movq %4, (%5)`

3: `addq`

2: `movq`

1: **Inst. 3 reads %2 1 cycle ago.**

Fetch

Decode

Execute

Memory

Write Back



Pipelining - Intro

Pipeline Runtime

1: `movq (%1), %2`
2: `movq (%2), %4`
3: `addq %2, %4`
4: `movq %4, (%5)`

Cycle: 4

4: Read %rip
3: Read %2, %4

4: `movq %4, (%5)`

3: `addq %2, %4`
2: `movq (%2), %4`



So we have proved that $5 < 4$!

Fetch

Decode

Execute

Memory

Write Back

Dependancy

It is not trivial at all to implement a pipeline!

When an instruction is issued, instructions before it has not accomplished yet!

Unfortunately instructions are not independant.

Their dependancies are described by...

Execute	1: rrmovq (%1), %2
Decode	2: rrmovq (%3), %4
Fetch	3: addq %2, %4
	4: rmmovq %4, (%5)

Dependancy

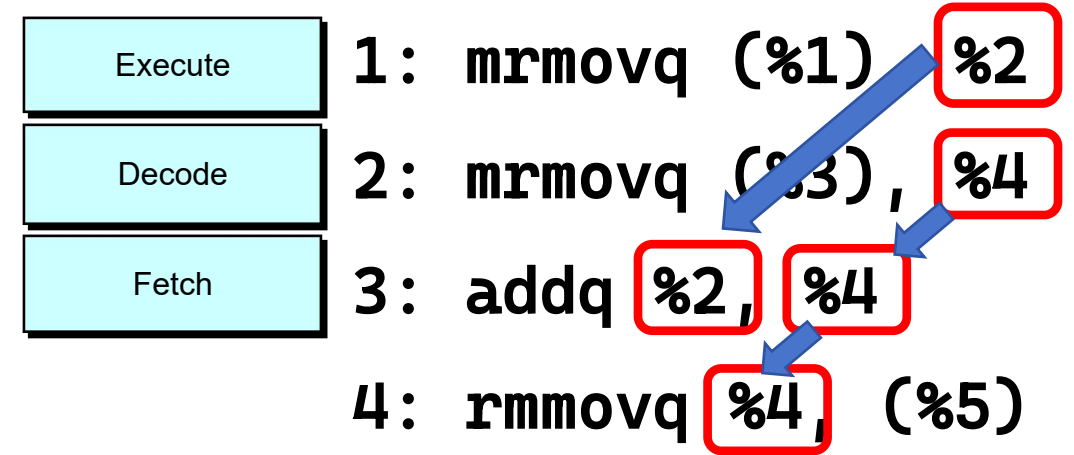
It is not trivial at all to implement a pipeline!

When an instruction is issued, instructions before it has not accomplished yet!

Unfortunately instructions are not independant.

Their dependancies are described by...

Date Flow



Dependancy - Synchronization

Let's make the problem clear:

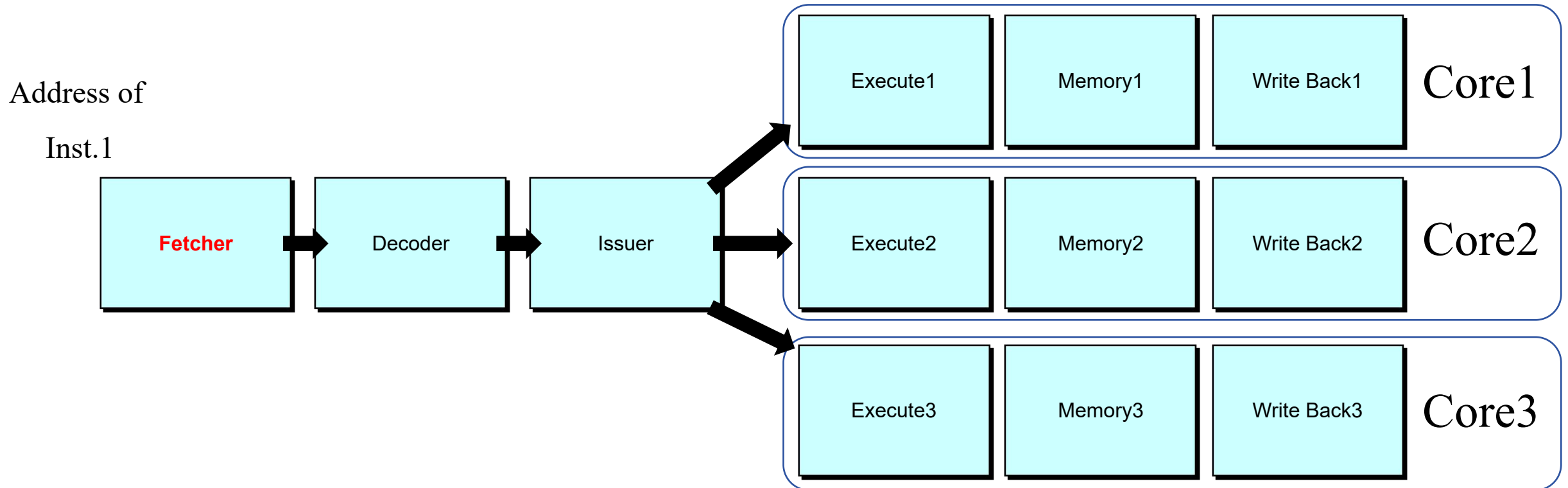
Assume that instructions are executed **concurrently**.

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`



Dependancy - Synchronization

Let's make the problem clear:

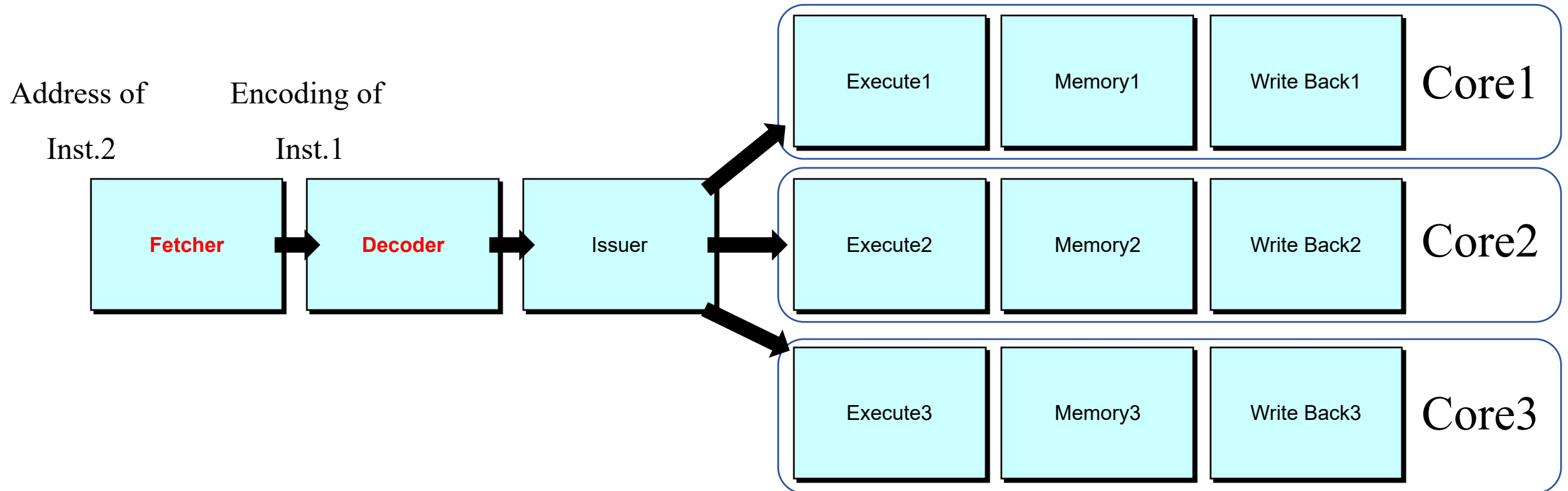
Assume that instructions are executed **concurrently**.

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`



Dependancy - Synchronization

Let's make the problem clear:

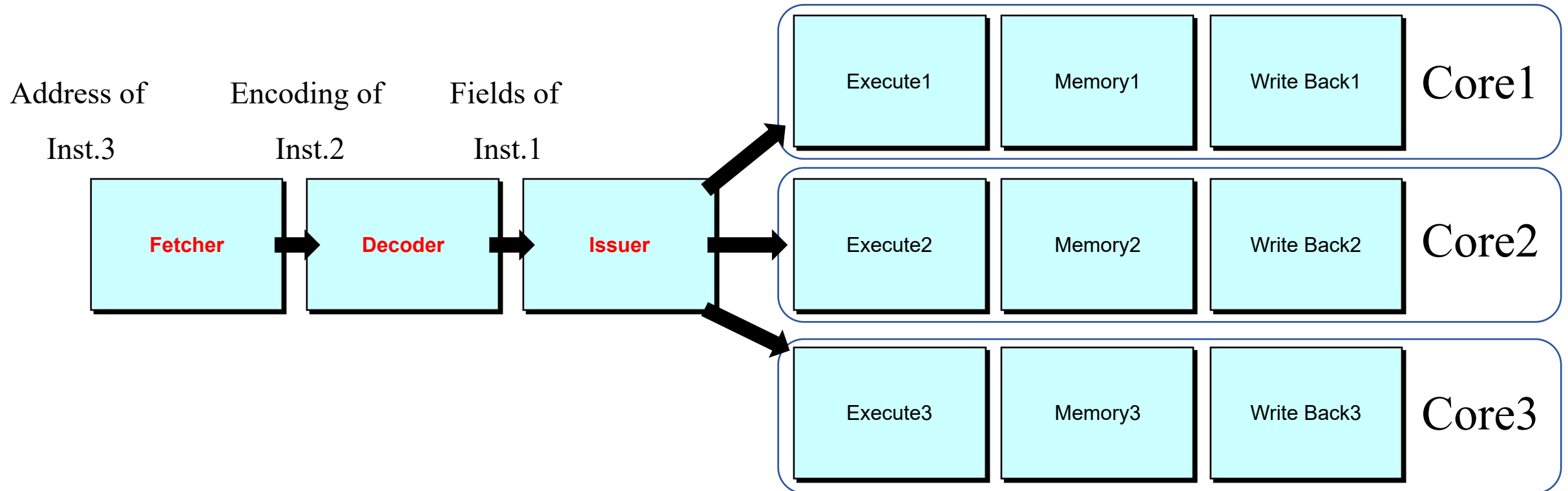
Assume that instructions are executed **concurrently**.

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`



Dependancy - Synchronization

Let's make the problem clear:

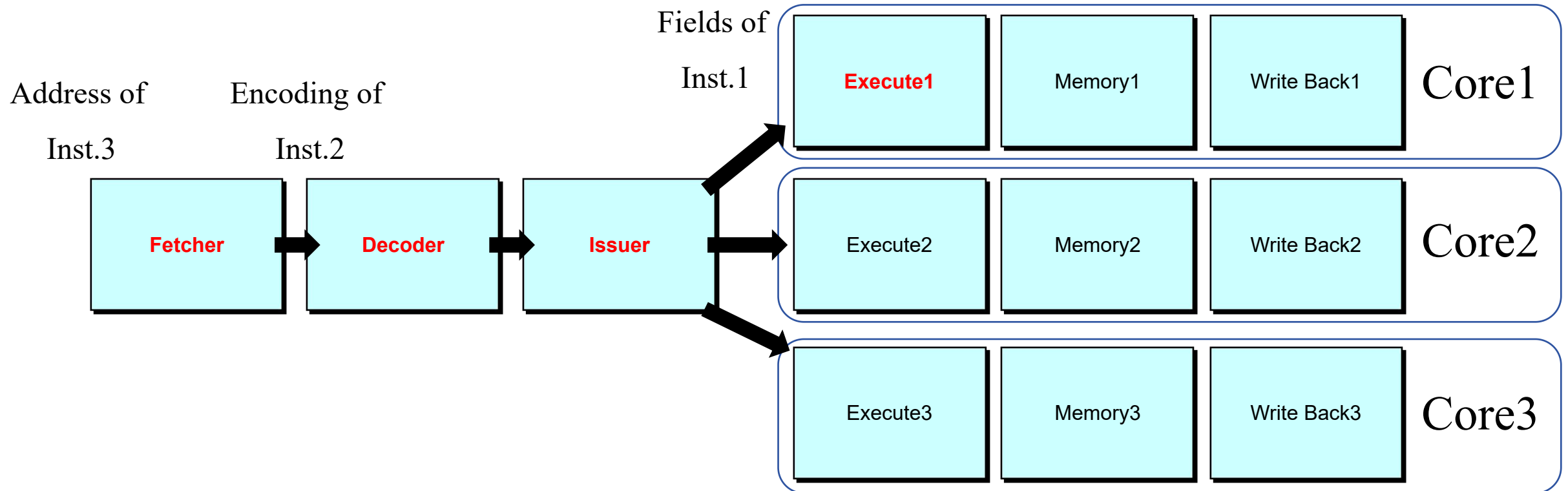
Assume that instructions are executed **concurrently**.

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`



Dependancy - Synchronization

Let's make the problem clear:

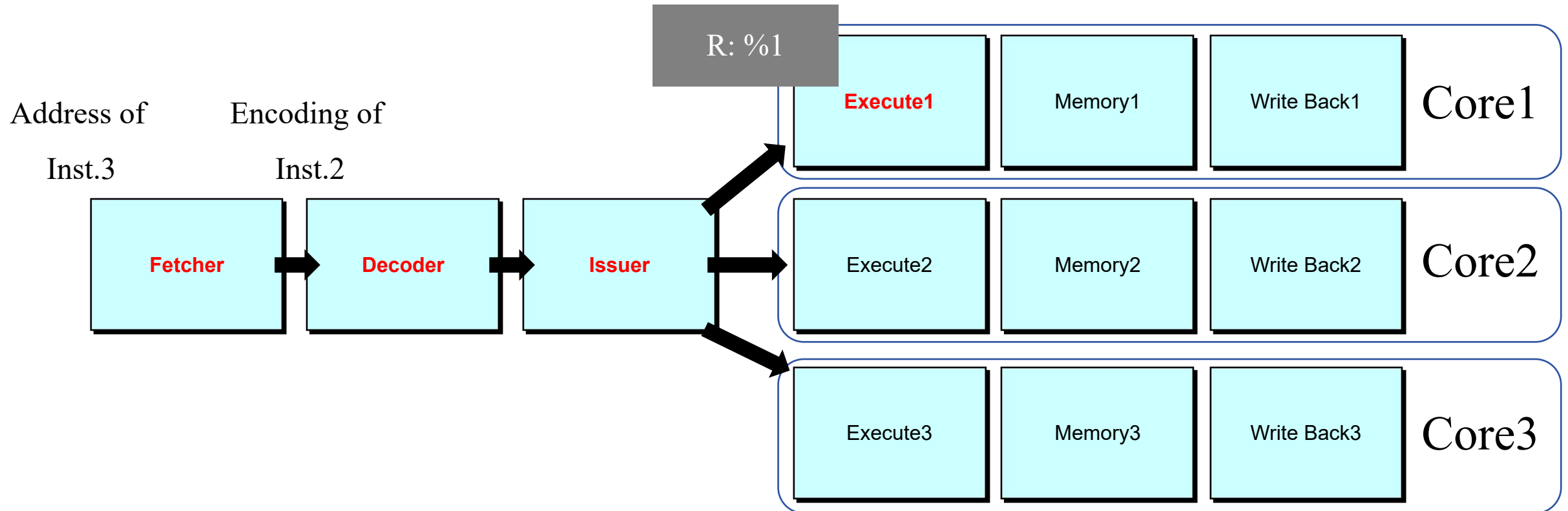
Assume that instructions are executed **concurrently**.

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`

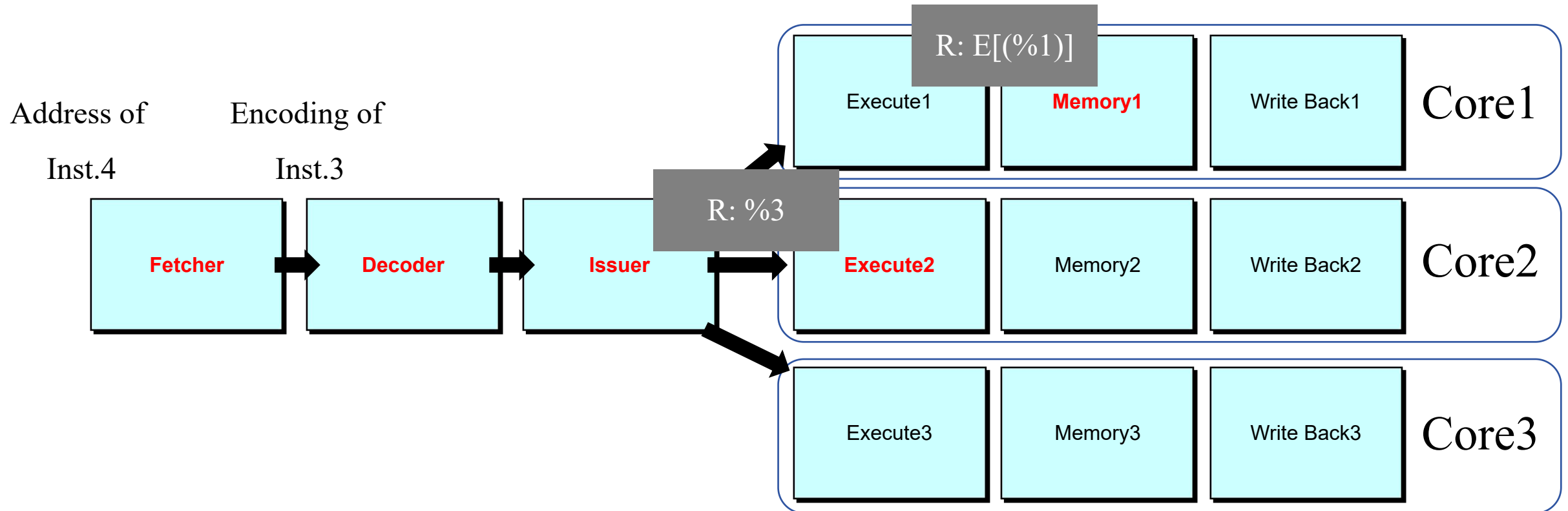


Dependancy - Synchronization

Let's make the problem clear:

Assume that instructions are executed **concurrently**.

```
1: mrmovq (%1), %2
2: mrmovq (%3), %4
3: addq %2, %4
4: rmmovq %4, (%5)
```



Dependancy - Synchronization

Let's make the problem clear:

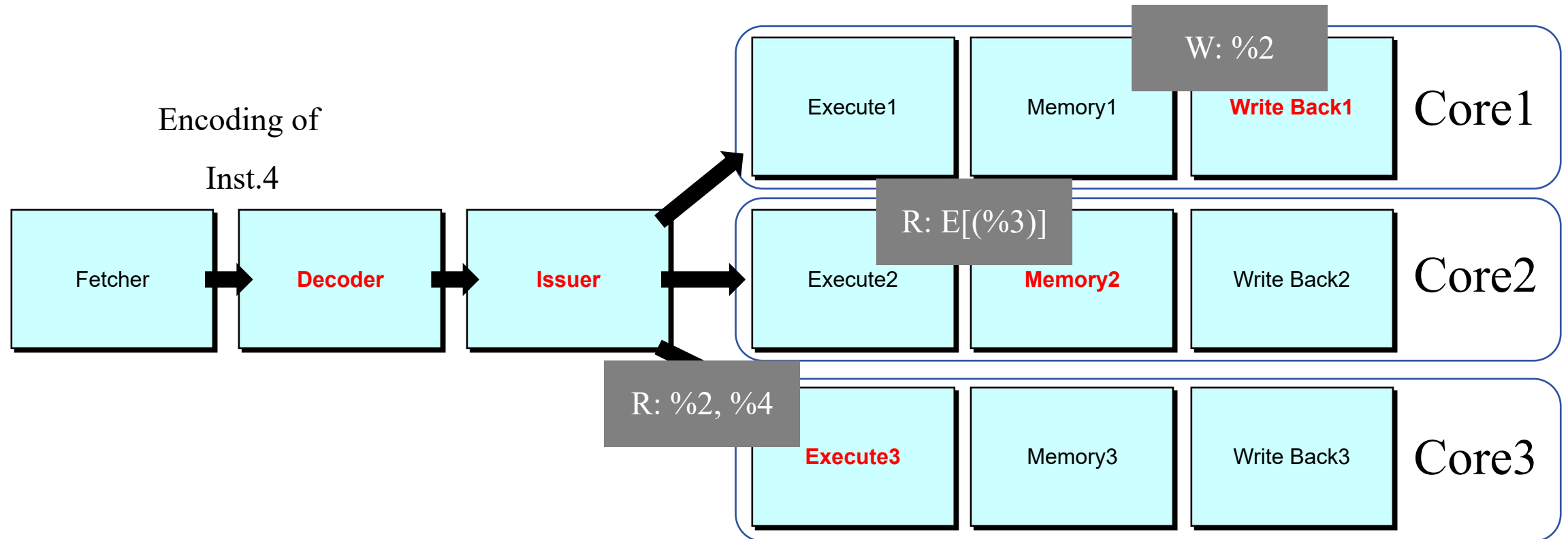
Assume that instructions are executed **concurrently**.

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`

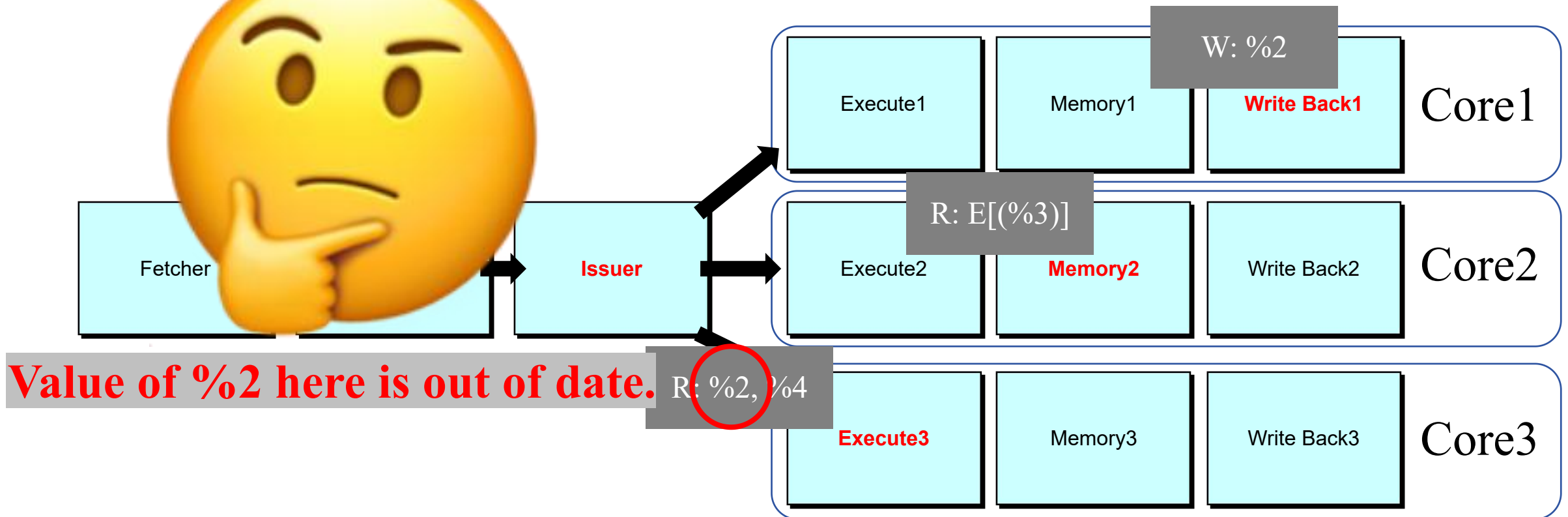


Dependancy - Synchronization

Let's make the problem clear:

Assume that instructions are executed **concurrently**.

```
1: mrmovq (%1), %2
2: mrmovq (%3), %4
3: addq %2, %4
4: rmmovq %4, (%5)
```

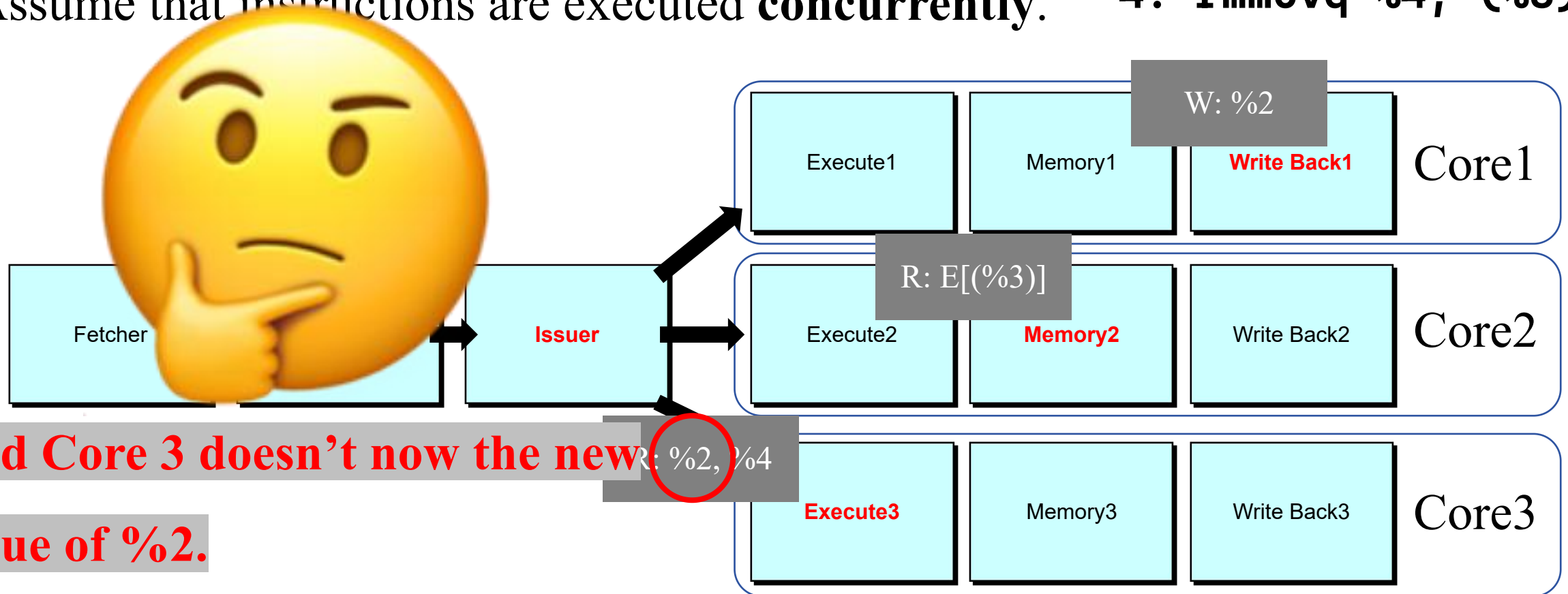


Dependancy - Synchronization

Let's make the problem clear:

Assume that instructions are executed **concurrently**.

```
1: mrmovq (%1), %2
2: mrmovq (%3), %4
3: addq %2, %4
4: rmmovq %4, (%5)
```

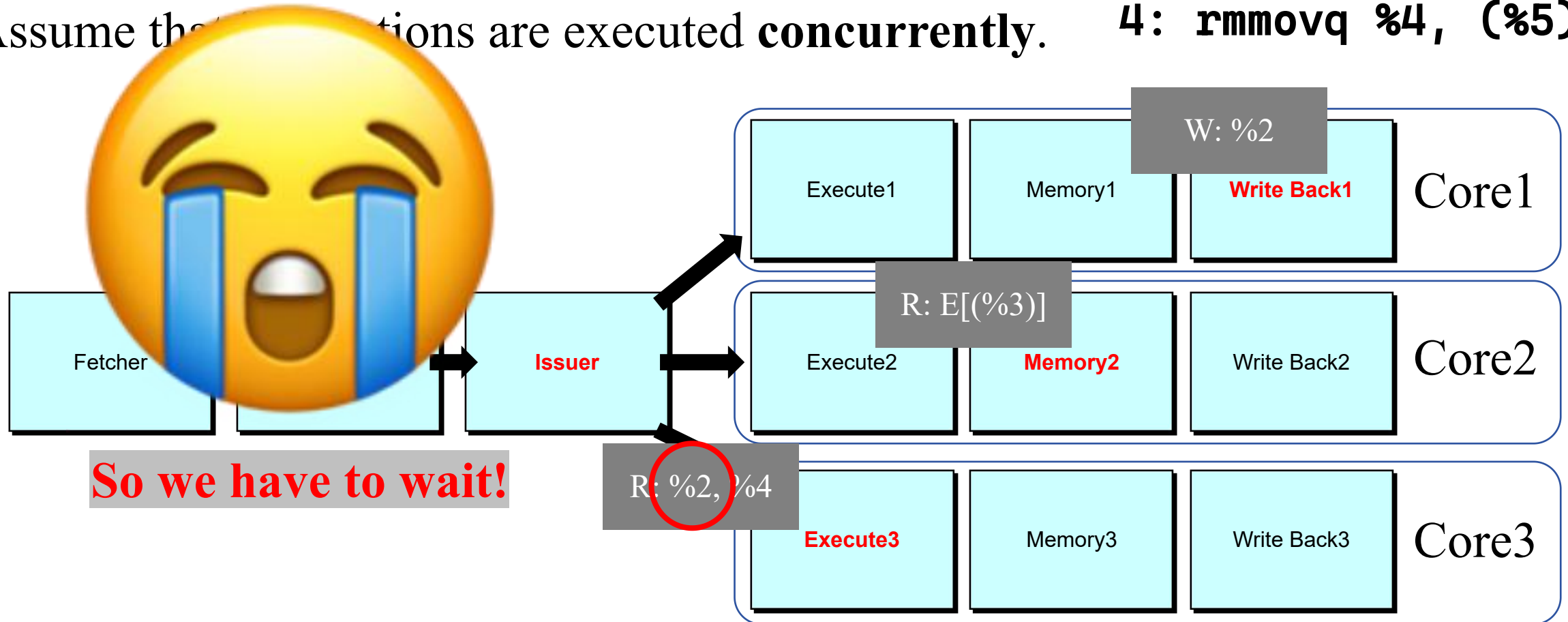


Dependancy - Synchronization

Let's make the problem clear:

Assume the instructions are executed **concurrently**.

```
1: mrmovq (%1), %2
2: mrmovq (%3), %4
3: addq %2, %4
4: rmmovq %4, (%5)
```



Dependancy - Synchronization

Let's make the problem clear:

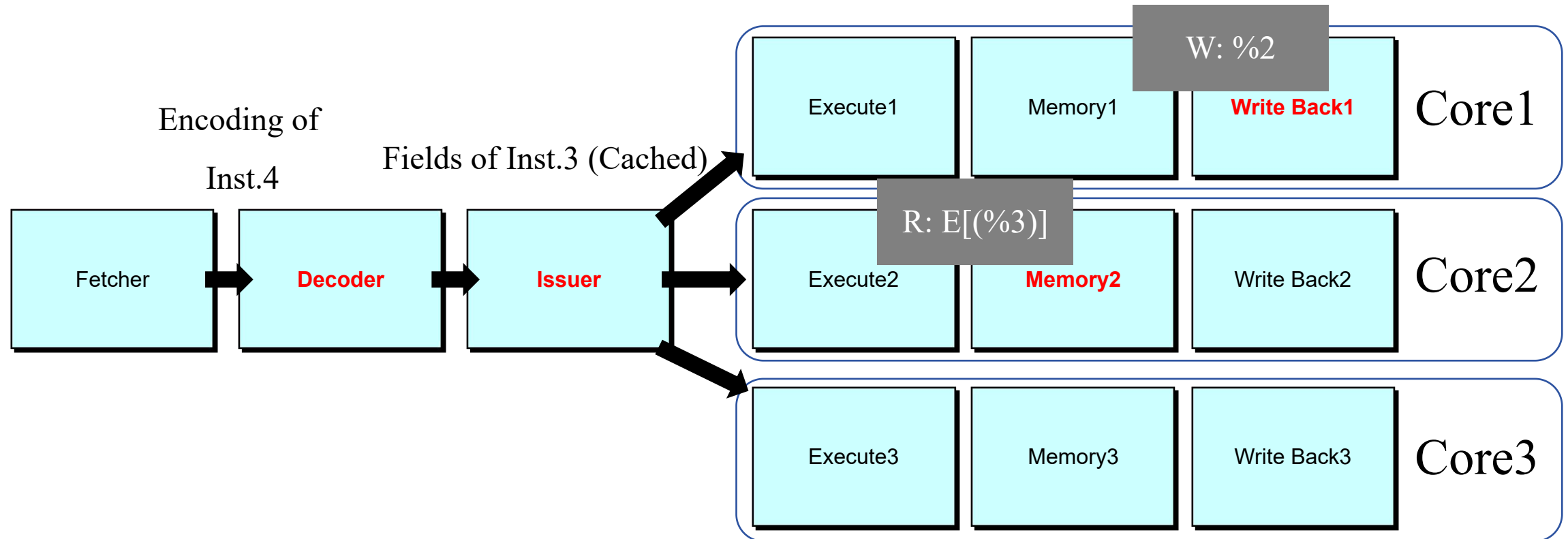
Assume that instructions are executed **concurrently**.

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`



Dependancy - Synchronization

Let's make the problem clear:

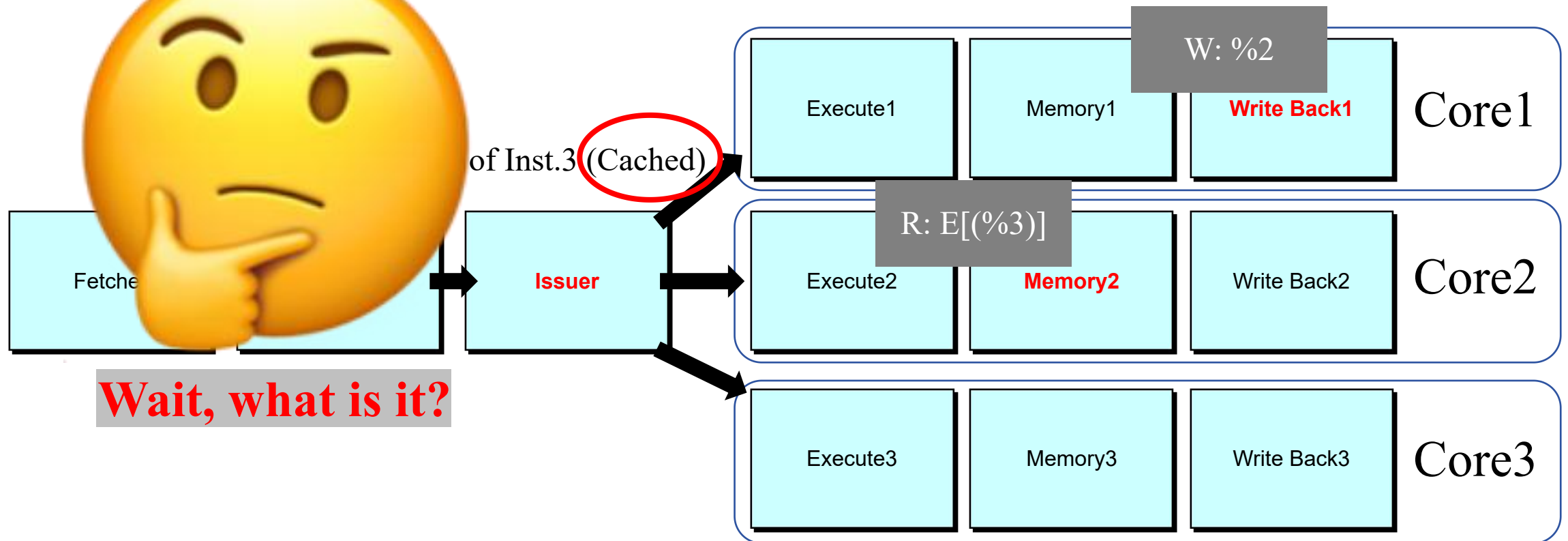
Assume that instructions are executed **concurrently**.

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`



Breaking Into - Cache

A kind of **storage** unit.

vs. **Registers**

Slower

Larger capacity size

vs. **Memory**

Faster

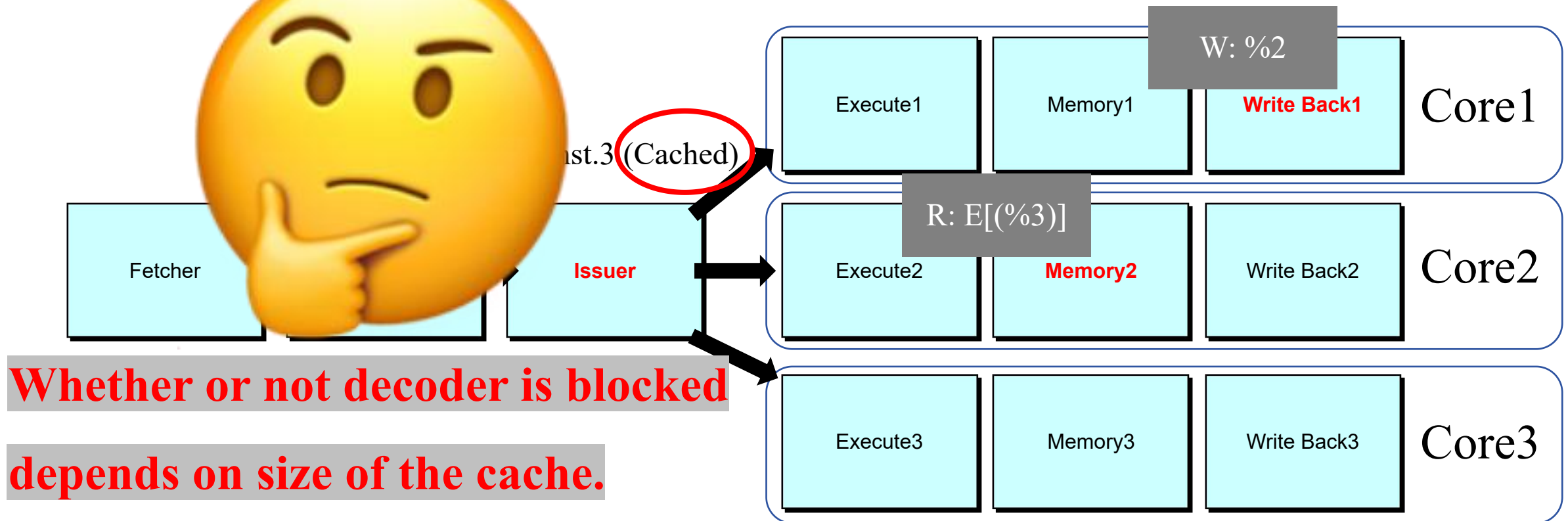
Less capacity size

Dependancy - Synchronization

Let's make the problem clear:

Assume that instructions are executed **concurrently**.

```
1: mrmovq (%1), %2
2: mrmovq (%3), %4
3: addq %2, %4
4: rmmovq %4, (%5)
```

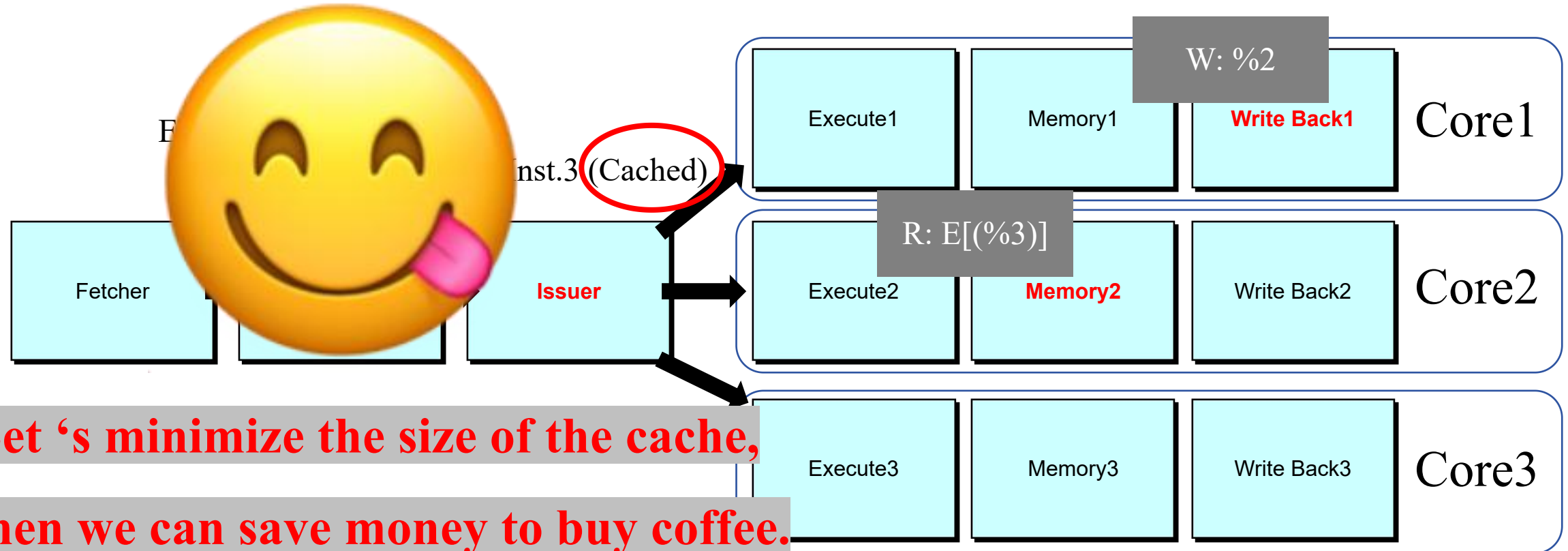


Dependancy - Synchronization

Let's make the problem clear:

Assume that instructions are executed **concurrently**.

```
1: mrmovq (%1), %2
2: mrmovq (%3), %4
3: addq %2, %4
4: rmmovq %4, (%5)
```



Dependancy - Synchronization

Let's make the problem clear:

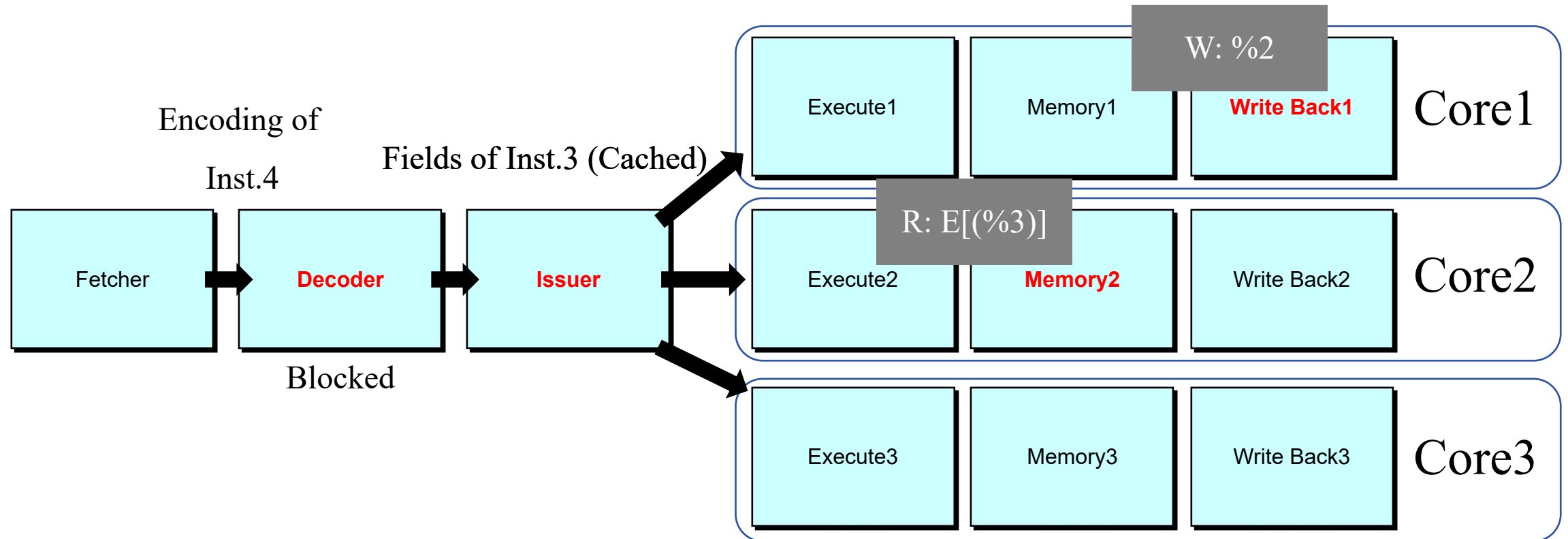
Assume that instructions are executed **concurrently**.

```
1: mrmovq (%1), %2
```

```
2: mrmovq (%3), %4
```

```
3: addq %2, %4
```

```
4: rmmovq %4, (%5)
```

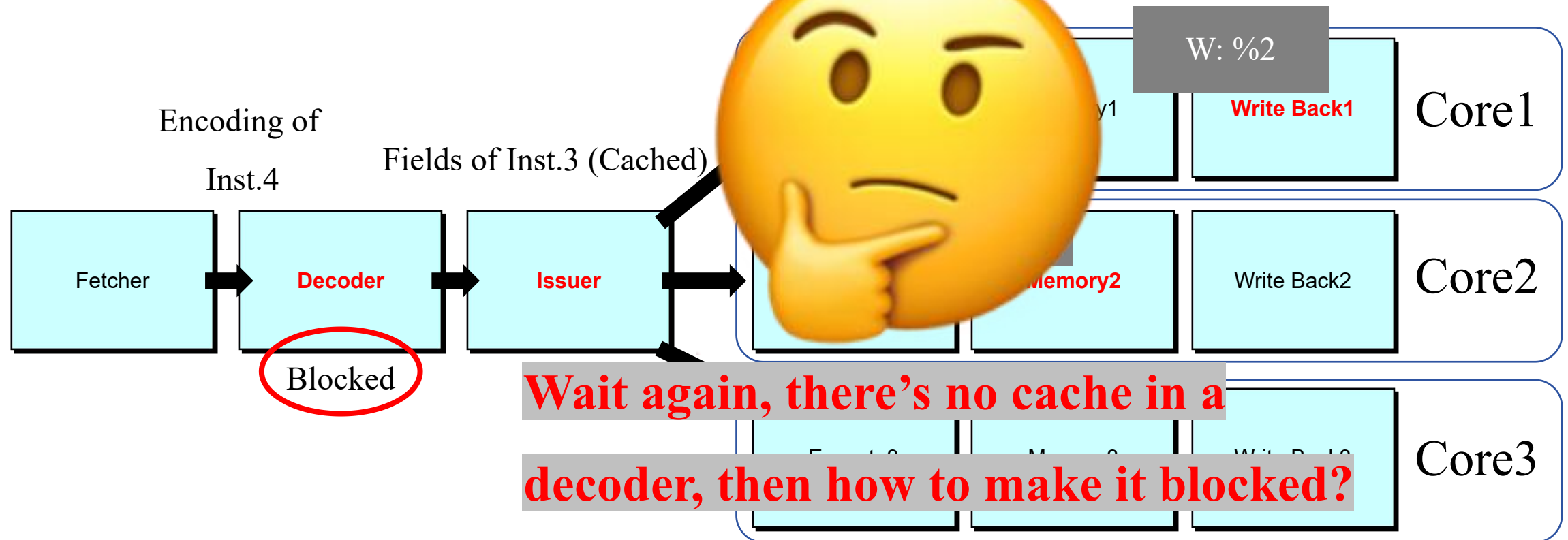


Dependancy - Synchronization

Let's make the problem clear:

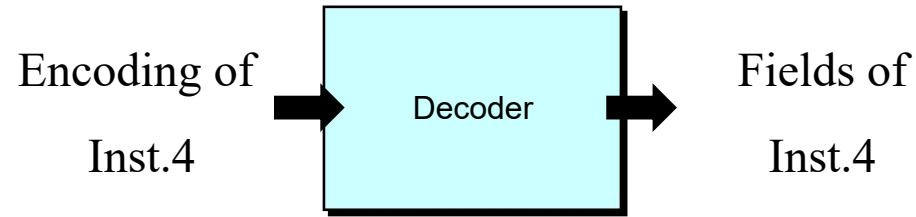
Assume that instructions are executed **concurrently**.

```
1: mrmovq (%1), %2
2: mrmovq (%3), %4
3: addq %2, %4
4: rmmovq %4, (%5)
```



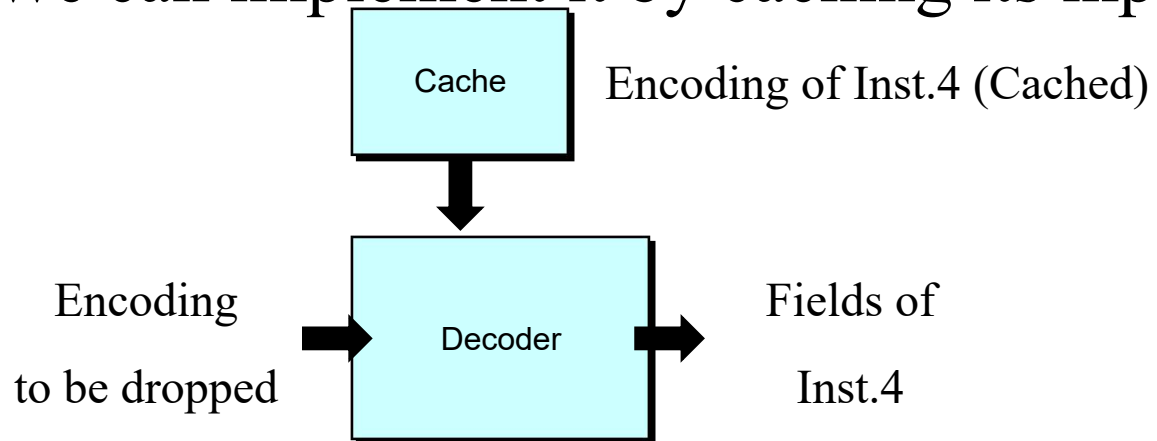
Breaking Into - How to Implement Blocking

We can view each unit as a **function**:



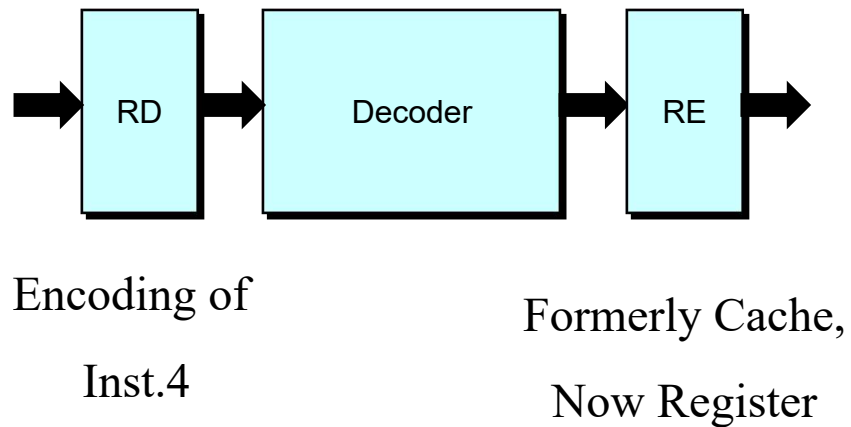
A unit that is blocked means that its output keeps unchanged.

We can implement it by caching its input:



Breaking Into - How to Implement Blocking

Since once a unit is blocked, we only need to cache exactly one input, we don't need to use a cache: use a **register** would be better!

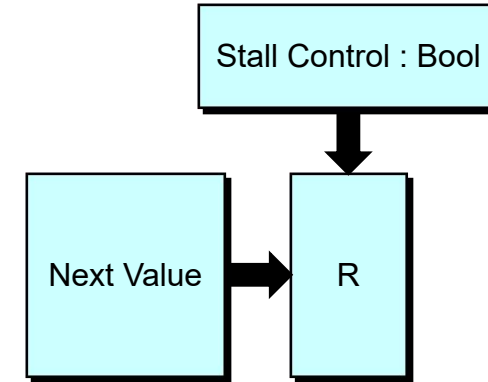


Then block a unit is to make its input register's value keep unchanged.

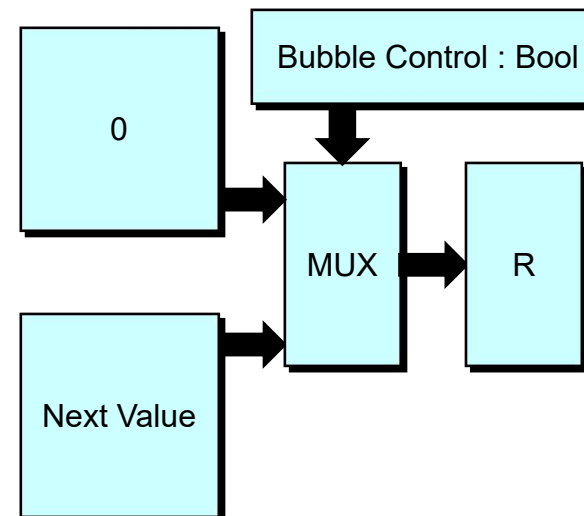
We should also flush its output to avoid side effects.

Breaking Into - How to Implement Blocking

Stall: Make a register's value unchanged.



Bubble: Make a register's value none.

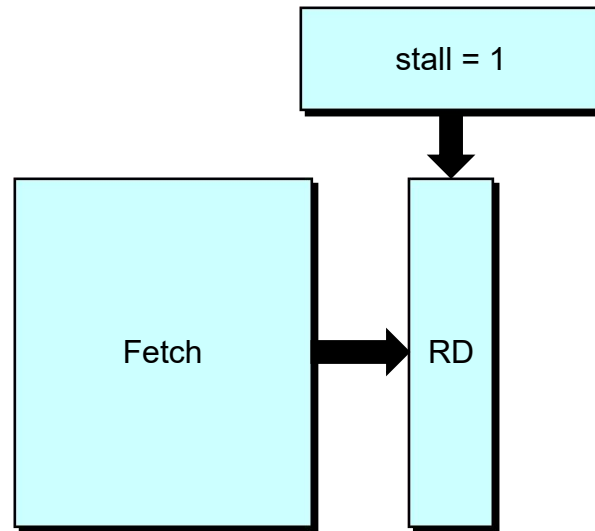


Breaking Into - How to Implement Blocking

To implement correct stall, we have to add RF.

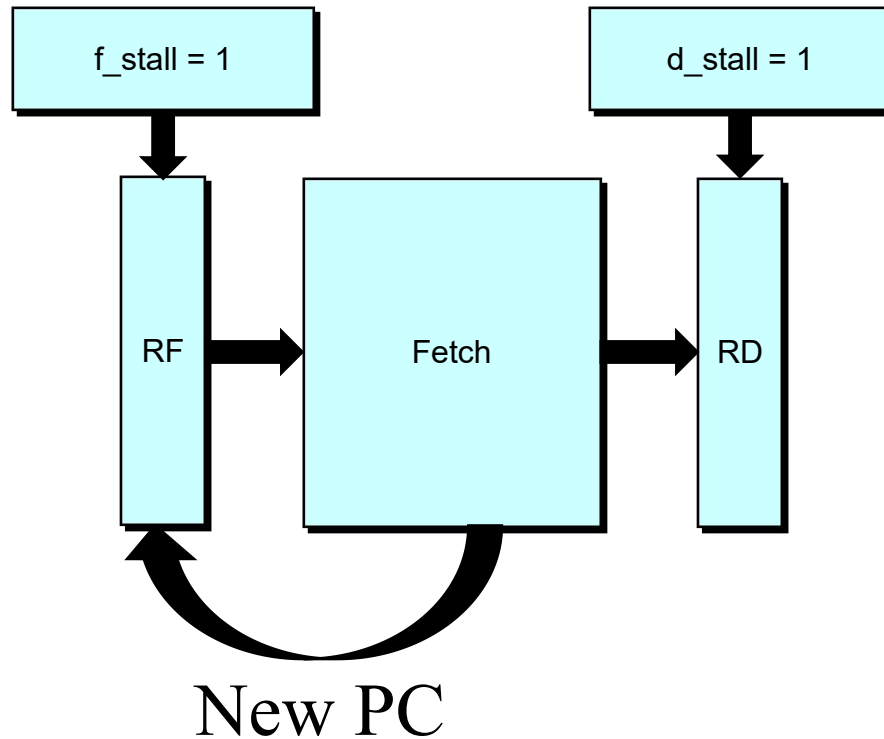
Otherwise:

PC keeps increasing!



Breaking Into - How to Implement Blocking

To implement correct stall, we have to add RF.

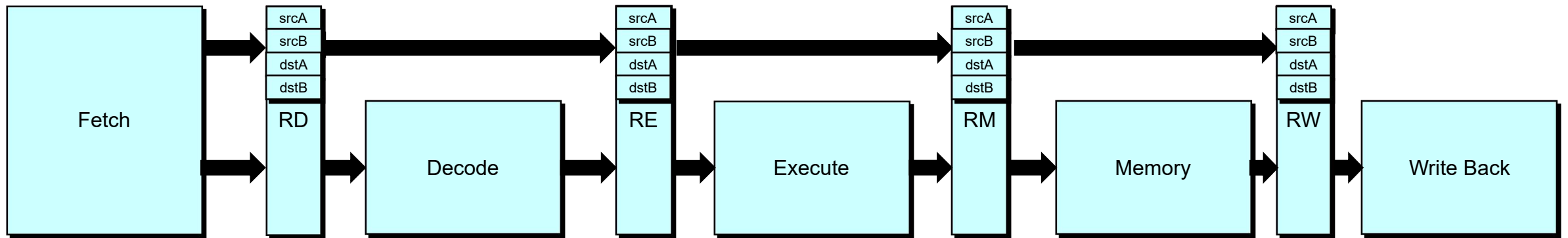


Breaking Into - When to Block

Next problem is how to generate control value.

Let's keep the message that which register is going to be read/written.

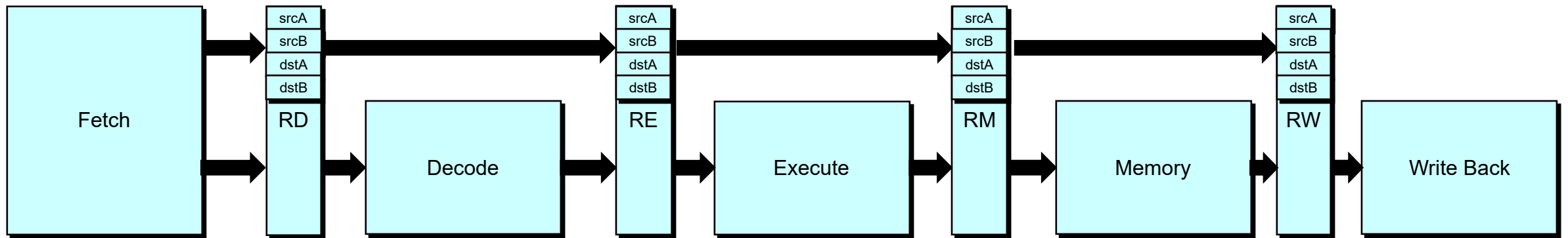
It is very easy to implement because IDs of register are encoded in a instruction encoding.



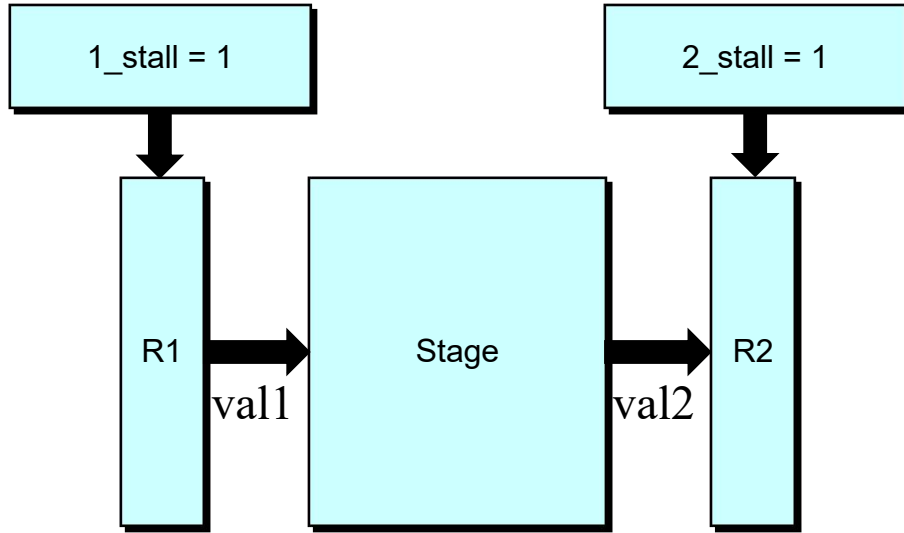
Breaking Into - When to Block

Then we can use these value to generate control value.

e.g. $D_srcA == E_dstA$ means rA is going to be written, so we should stall RD, and bubble RE. That is, $D_stall = 1, E_bubble = 1$.



Breaking Into - Case Study: Stall-Stall $\Leftrightarrow$ Stall-Bubble



$\text{val2} = \text{Stage}(\text{val1}) = \text{Stage}(\text{R1}) = \text{R2}$

R2's value could be dropped.

Dependancy - Synchronization

Let's make the problem clear:

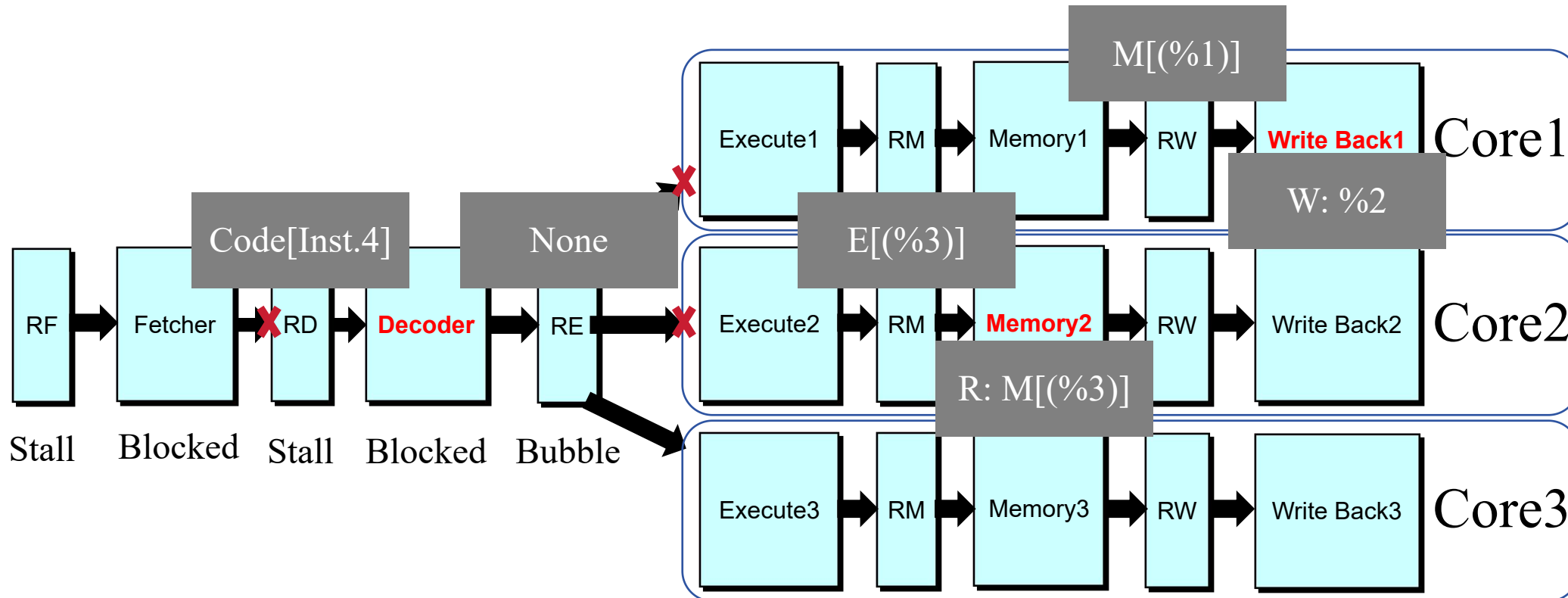
Assume that instructions are executed **concurrently**.

1: `mrmovq (%1), %2`

2: `mrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`



Dependancy - Synchronization

Let's make the problem clear:

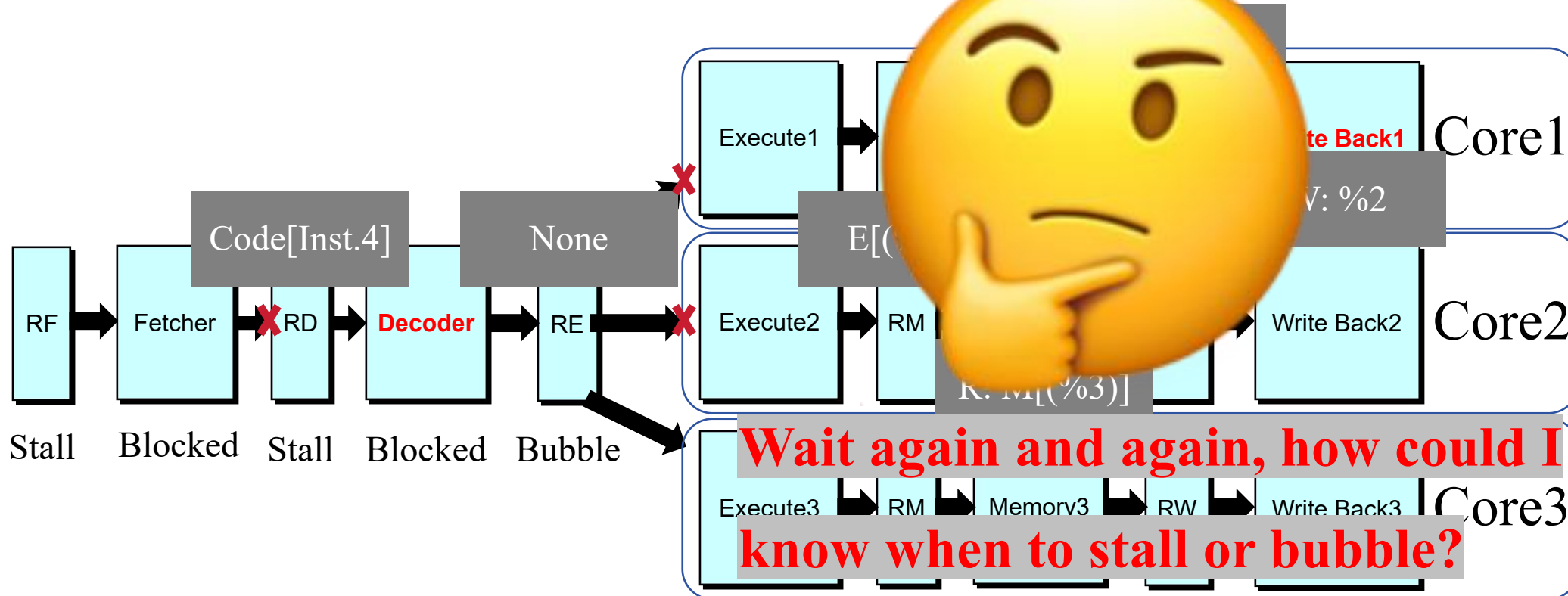
Assume that instructions are executed **concurrently**.

```
1: mrmovq (%1), %2
```

```
2: mrmovq (%3), %4
```

```
3: addq %2, %4
```

```
4: rmmovq %4, (%5)
```



Dependancy - Synchronization

Let's make it clear:

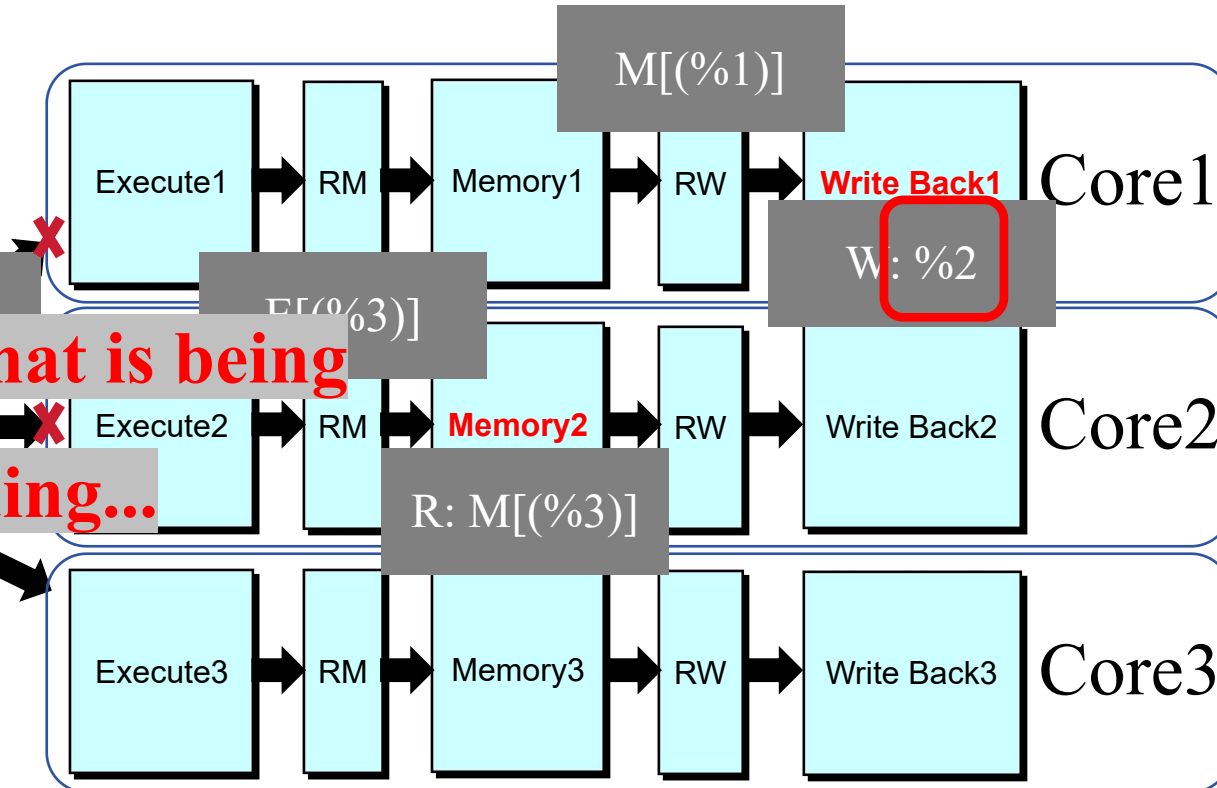
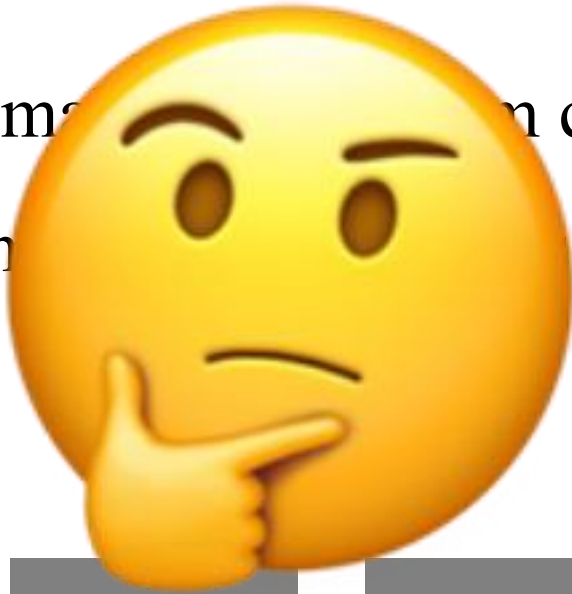
Assume instructions are executed **concurrently**.

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

3: `addq %2, %4`

4: `rrmovq %4, (%5)`



If I am decoding a register that is being written, I have to stall decoding...

Stall Blocked Stall Blocked Bubble

Dependancy - Synchronization

Let's make the problem clear:

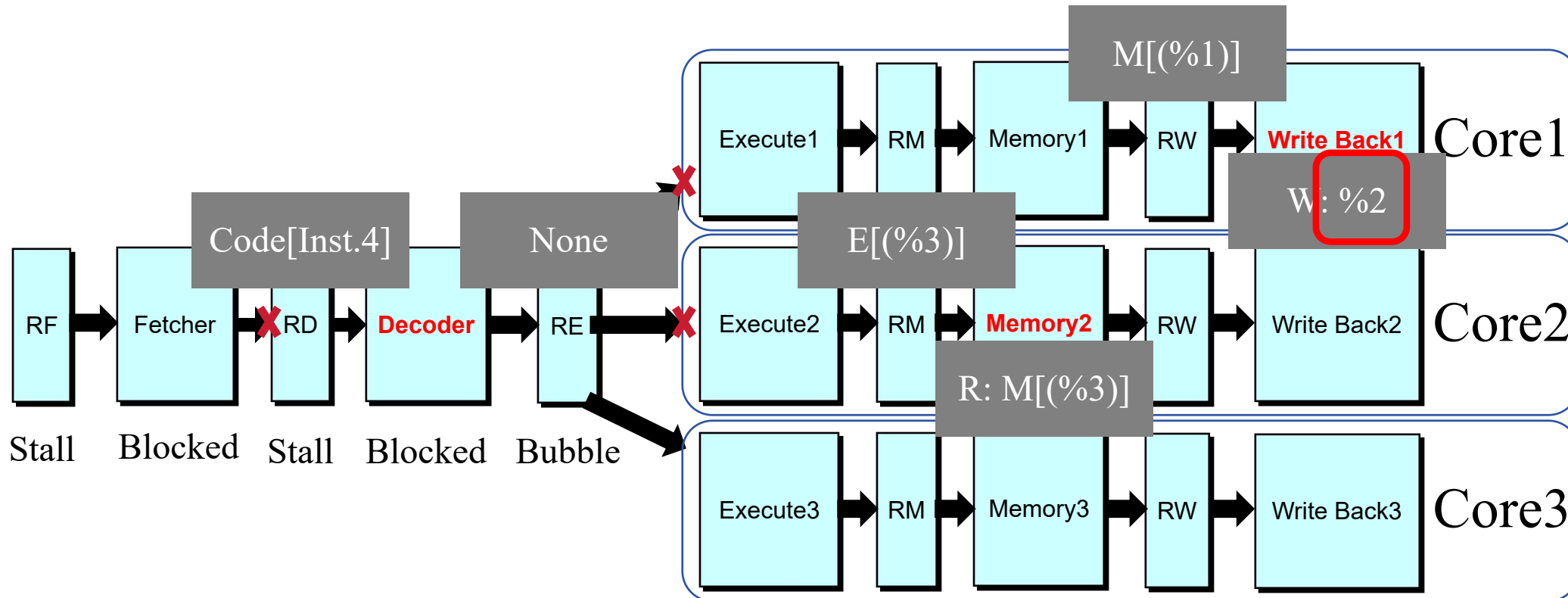
Assume that instructions are executed **concurrently**.

1: `mrmovq (%1), %2`

2: `mrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`



Dependancy - Synchronization

Let's make the problem clear:

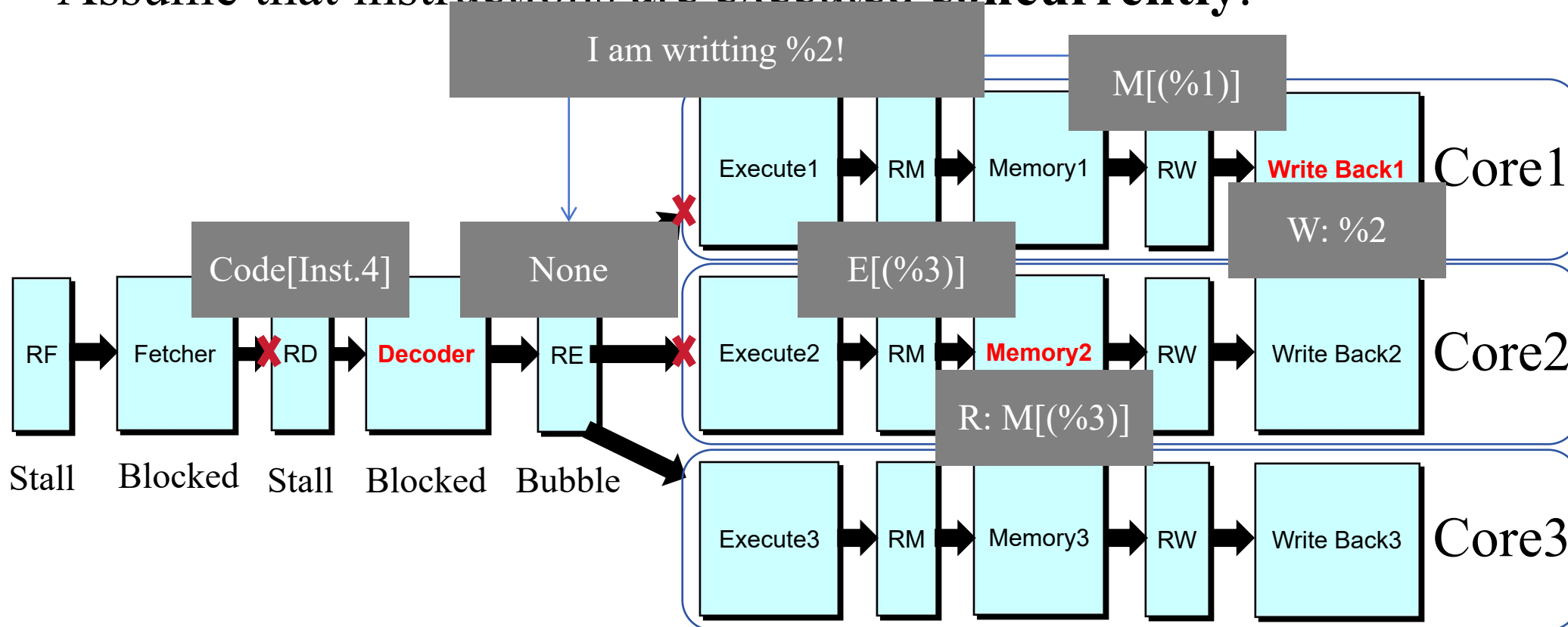
Assume that instructions are executed **concurrently**.

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`



Dependancy - Synchronization

Let's make the problem clear:

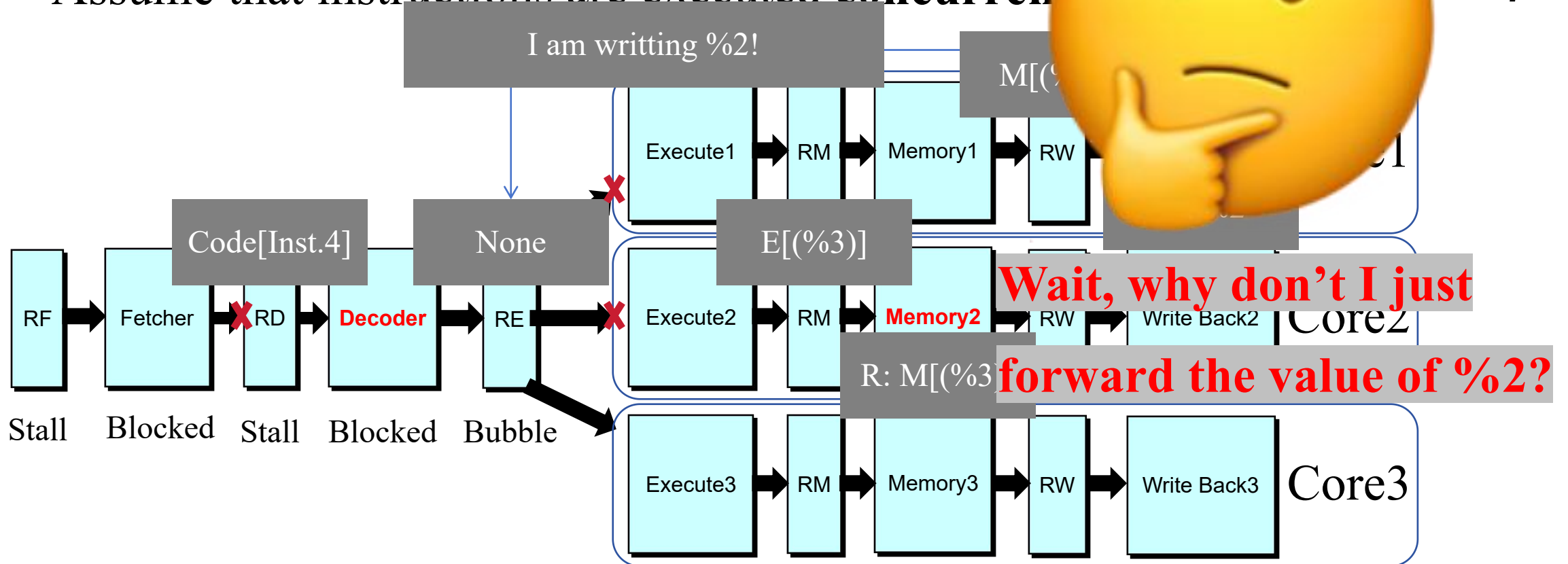
Assume that instructions are executed **concurrent**

1: `mrmovq (%1), %2`

2: `mrmovq (%3), %4`

`%2, %4`

`%4, (%5)`



Dependancy - Synchronization

Let's make the problem clear:

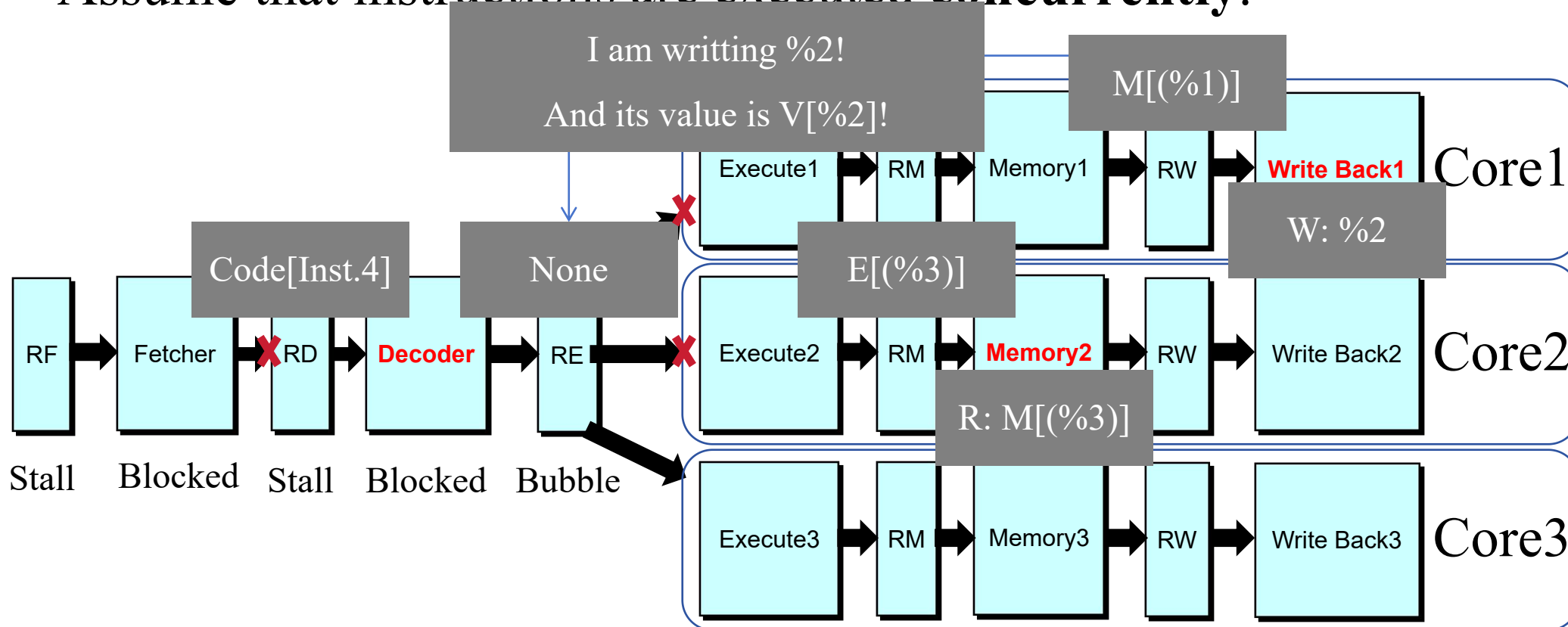
Assume that instructions are executed **concurrently**.

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`



Dependancy - Synchronization

1: mrmovq (%1), %2

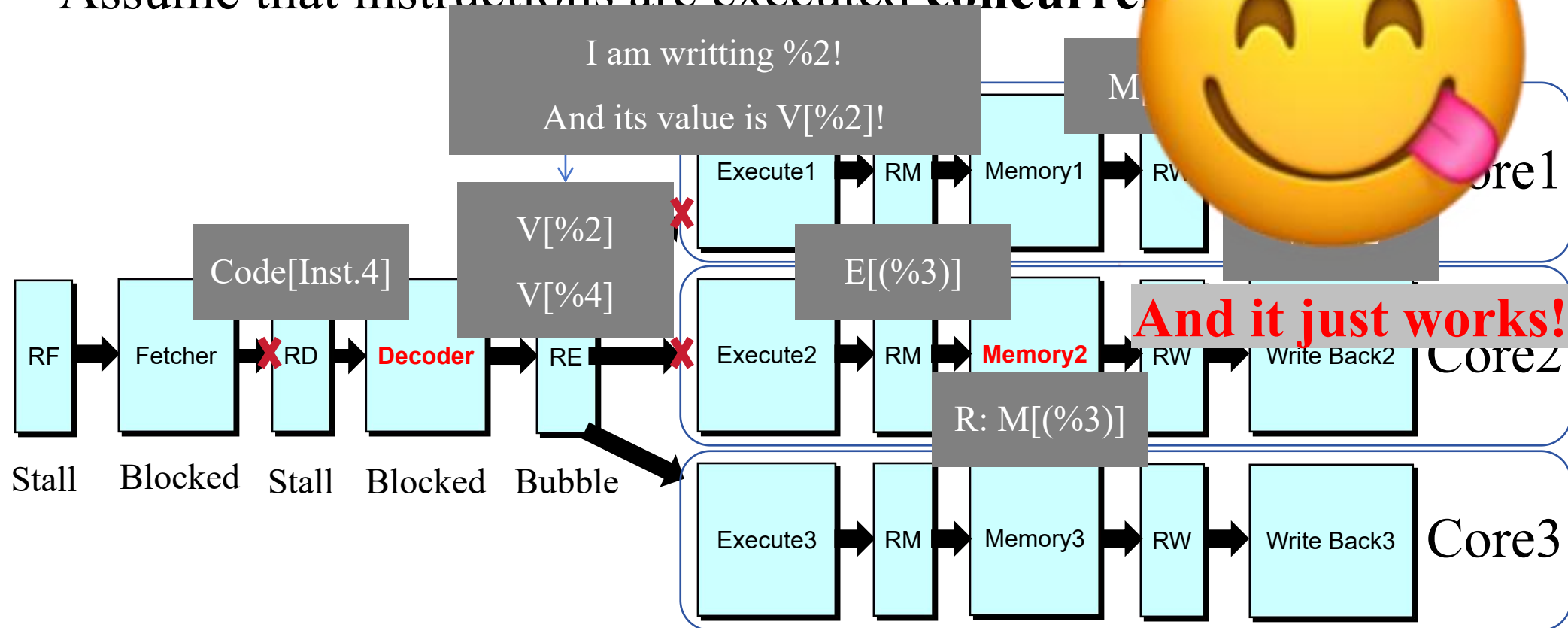
2: mrmovq (%3), %4

addq %2, %4

mrmovq %4, (%5)

Let's make the problem clear:

Assume that instructions are executed **concurrently**



Dependancy - Synchronization

Let's make the problem clear:

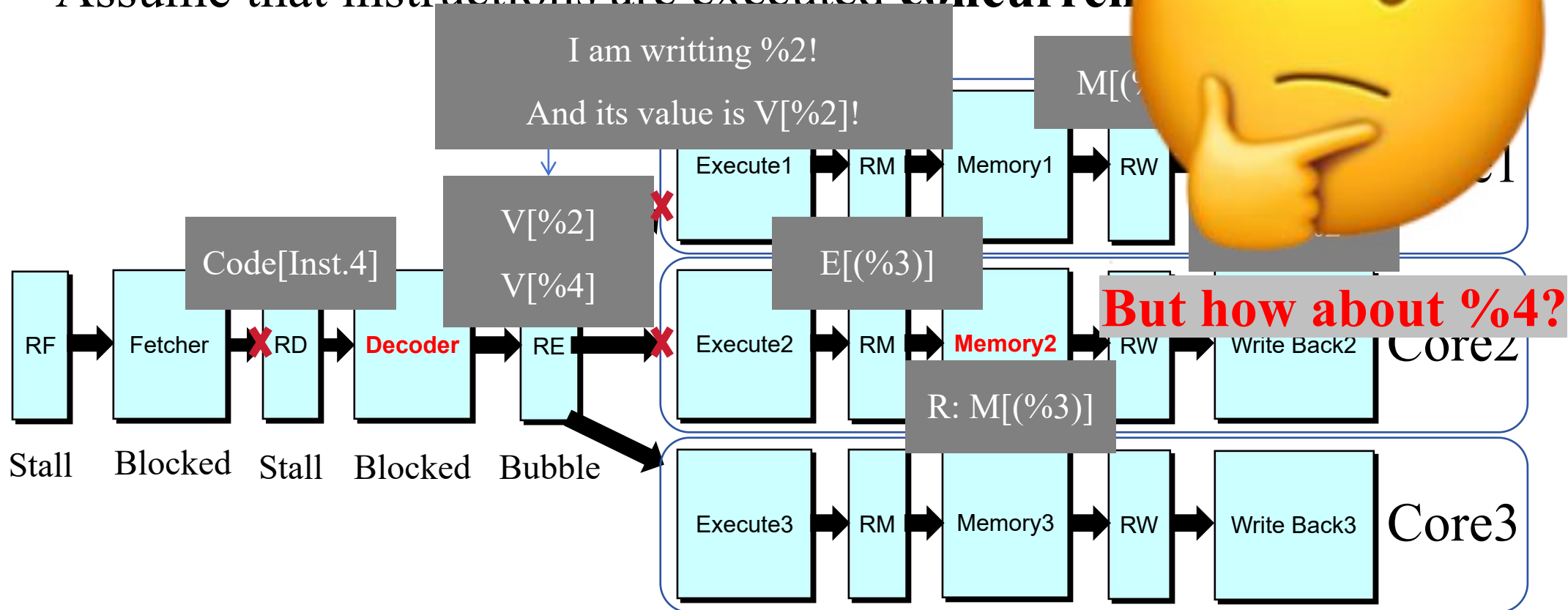
Assume that instructions are executed **concurrent**

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

`%2, %4`

`%4, (%5)`



Dependancy - Synchronization

Let's make the problem clear:

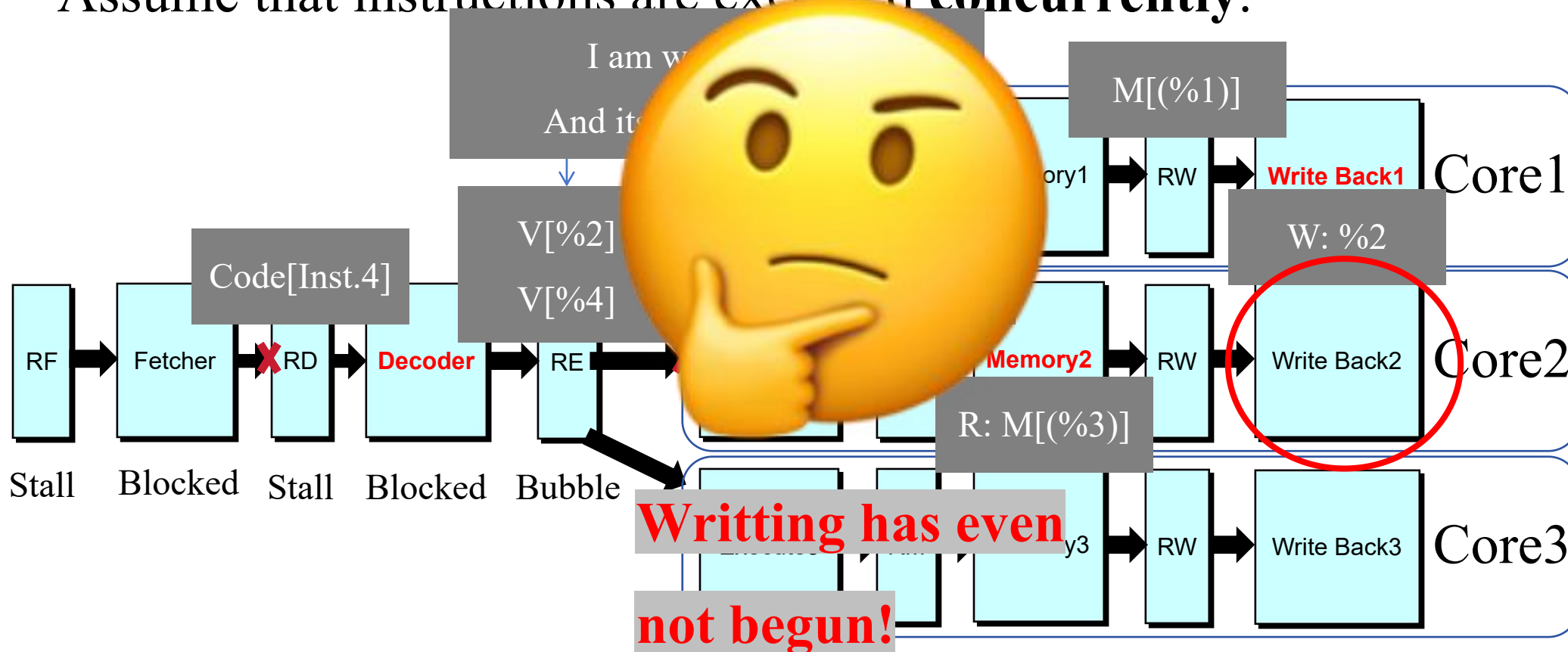
Assume that instructions are executed **concurrently**.

```
1: mrmovq (%1), %2
```

```
2: mrmovq (%3), %4
```

```
3: addq %2, %4
```

```
4: rmmovq %4, (%5)
```



Dependancy - Synchronization

Let's make the problem clear:

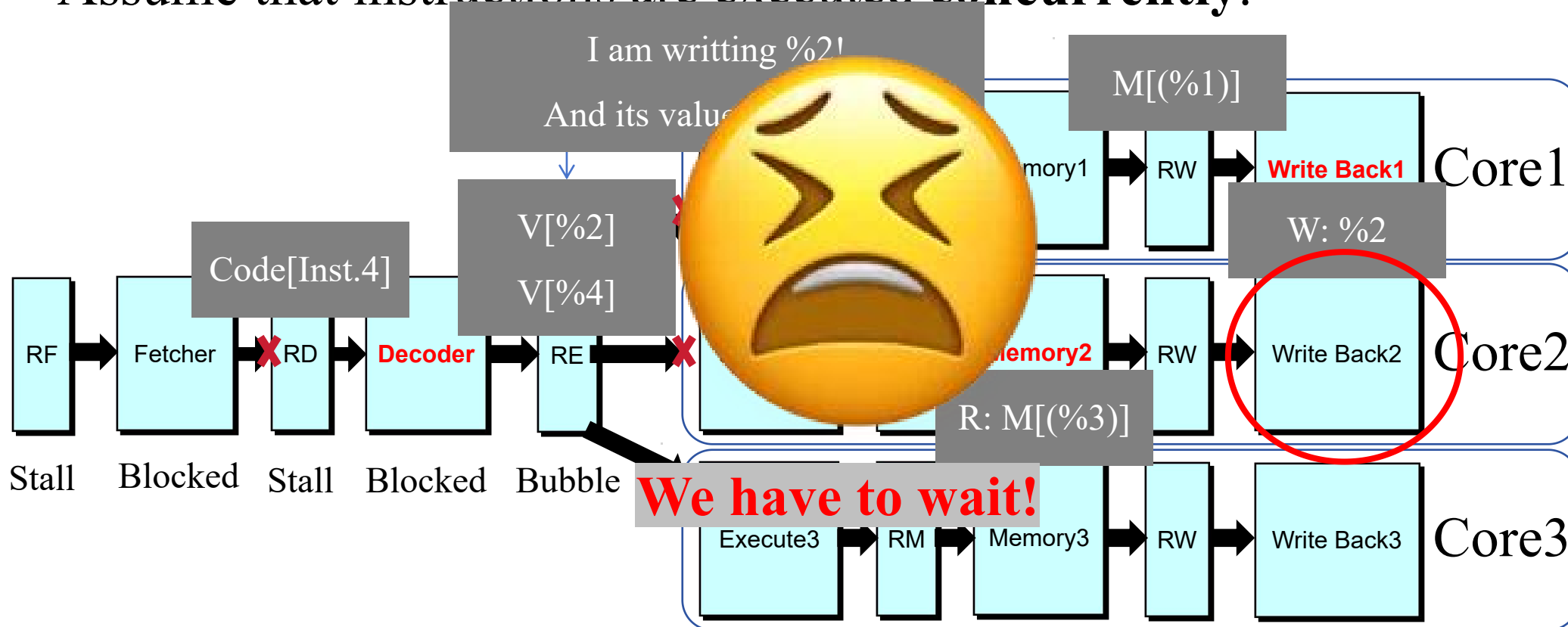
Assume that instructions are executed **concurrently**.

```
1: mrmovq (%1), %2
```

```
2: mrmovq (%3), %4
```

```
3: addq %2, %4
```

```
4: rmmovq %4, (%5)
```



Dependancy - Synchronization

Let's make the problem clear:

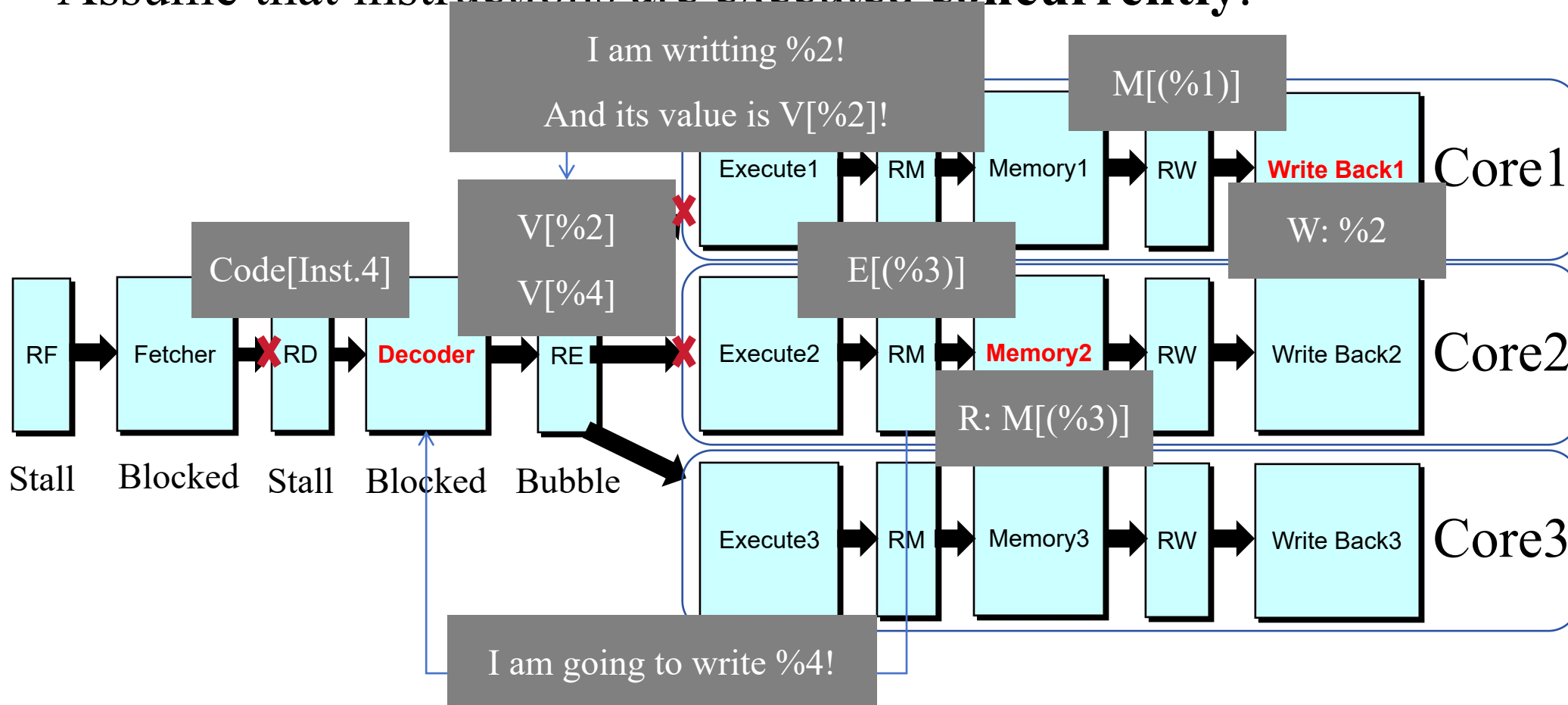
Assume that instructions are executed **concurrently**.

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

3: `addq %2, %4`

4: `rmmovq %4, (%5)`



Dependancy - Synchronization

Let's make the problem clear:

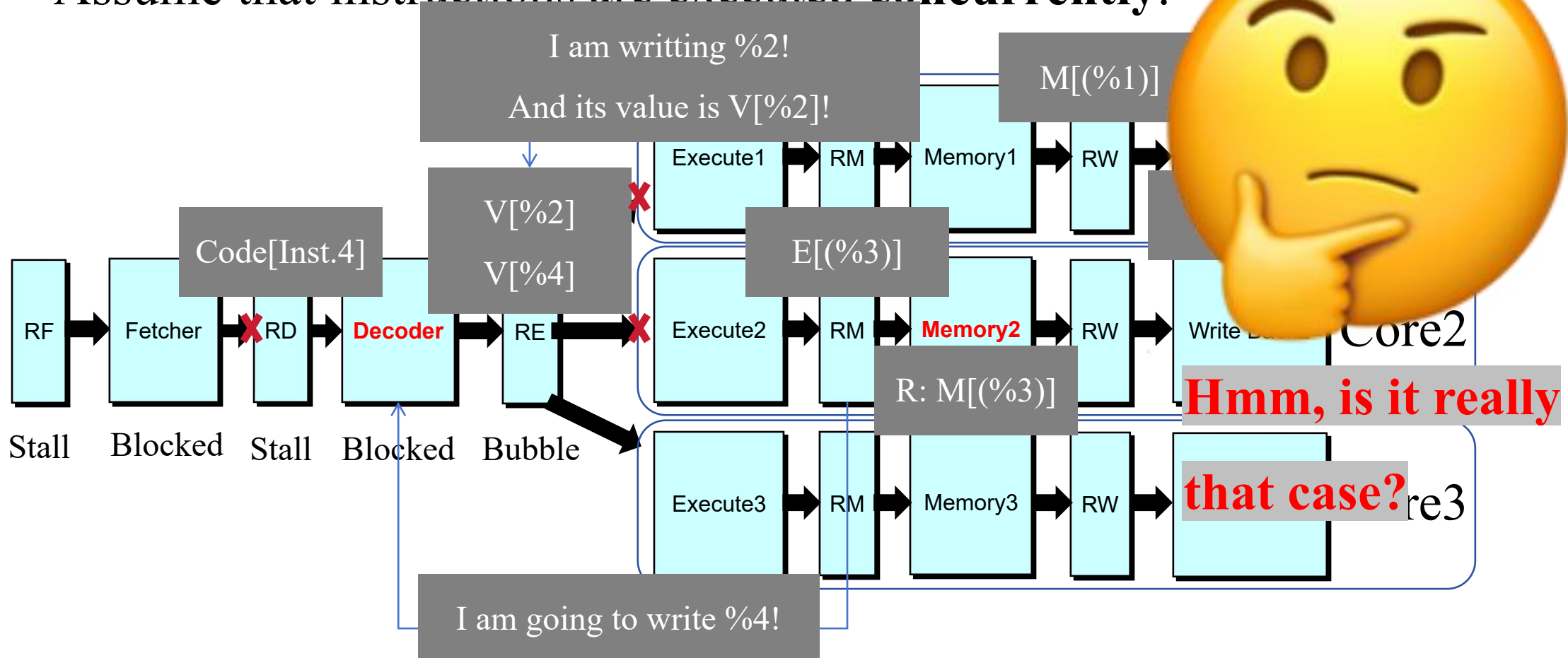
Assume that instructions are executed **concurrently**.

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

3: `addq %2, %4`

4: `addq %4, (%5)`



Dependancy - Synchronization

Let's make the problem clear:

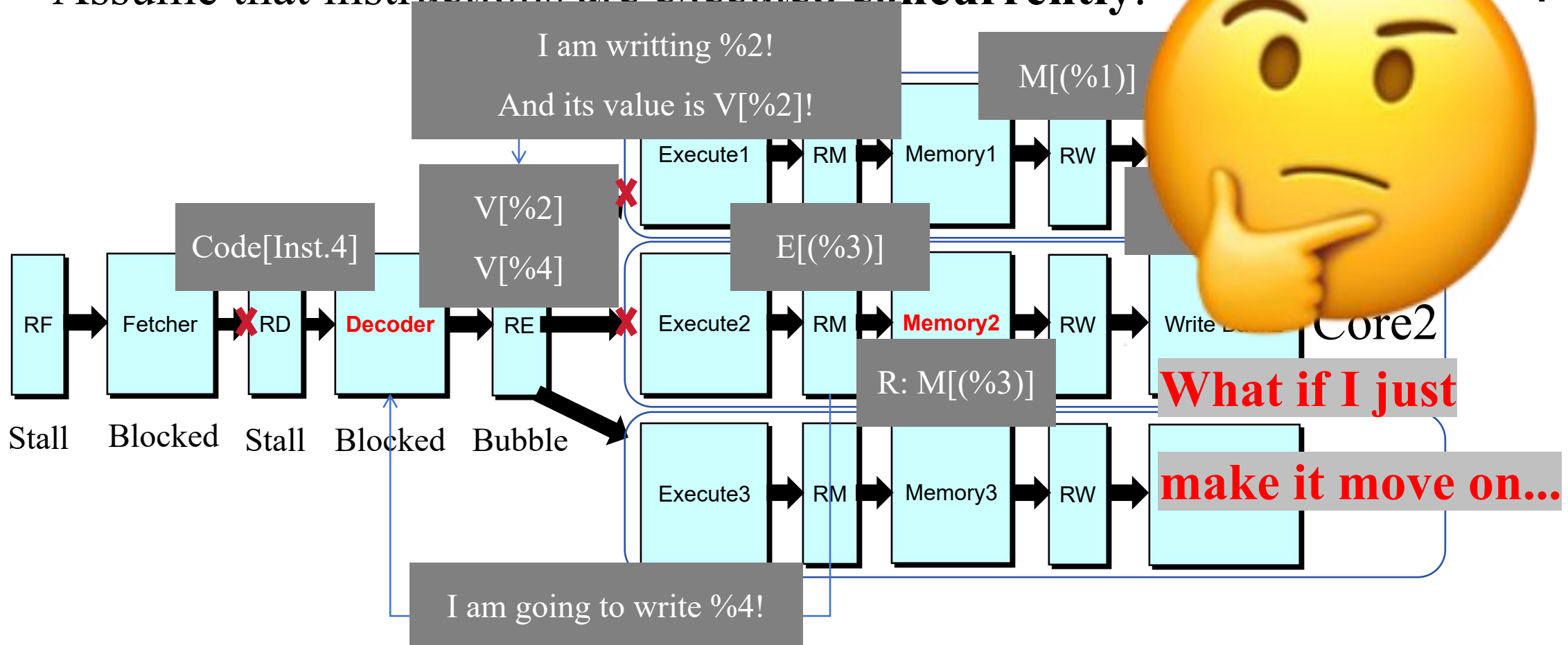
Assume that instructions are executed **concurrently**.

```
1: mrmovq (%1), %2
```

```
2: mrmovq (%3), %4
```

```
3: addq %2, %4
```

%4, (%5)



Dependancy - Synchronization

Let's make the problem clear:

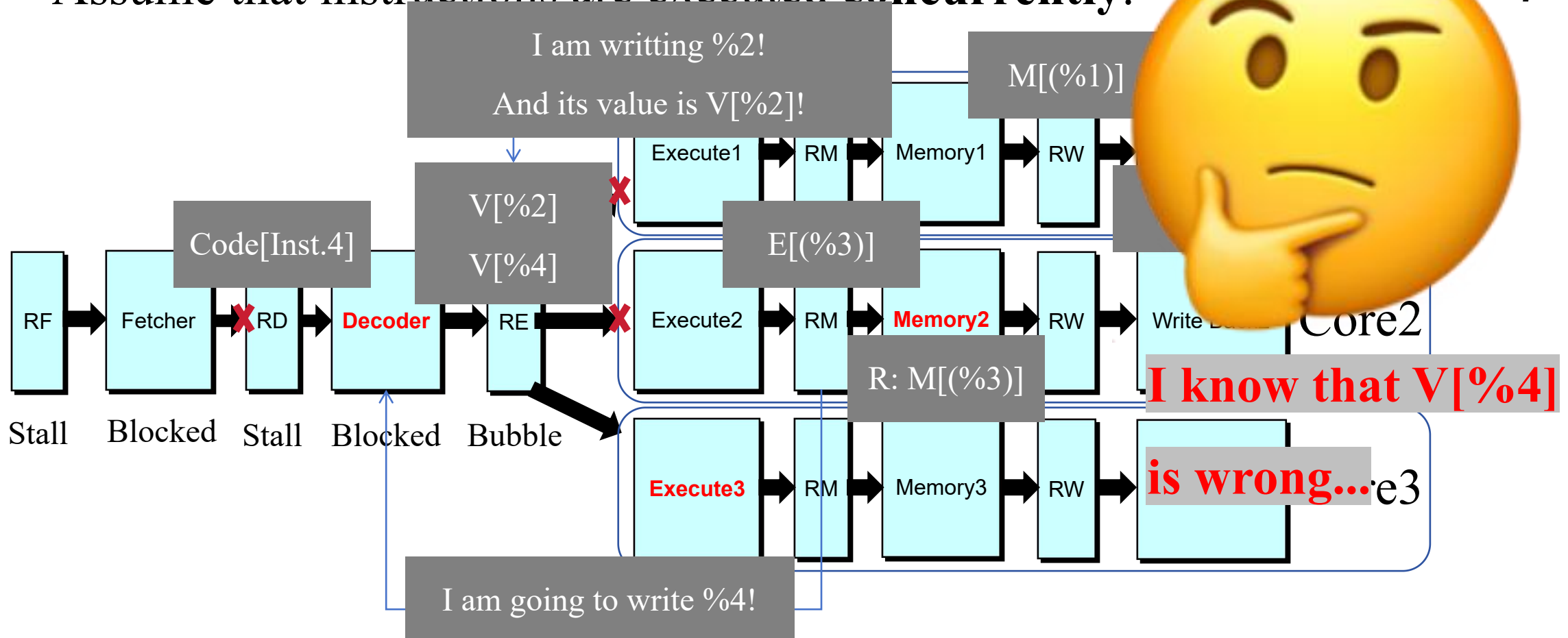
Assume that instructions are executed **concurrently**.

1: `rrmovq (%1), %2`

2: `rrmovq (%3), %4`

3: `addq %2, %4`

4: `addq %4, (%5)`



Dependancy - Synchronization

Let's make the problem clear:

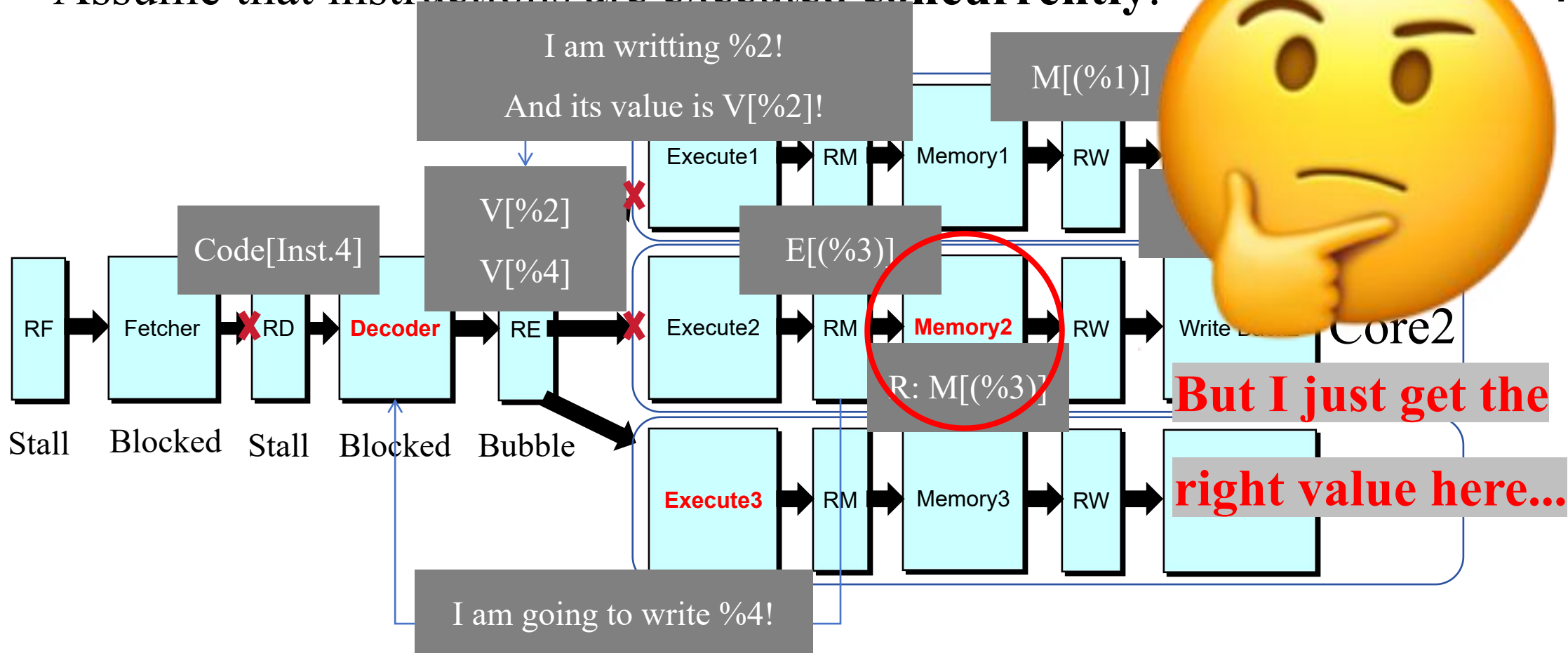
Assume that instructions are executed **concurrently**.

1: `mrmovq (%1), %2`

2: `mrmovq (%3), %4`

3: `addq %2, %4`

4: `addq %4, (%5)`



Dependancy - Synchronization

Let's make the problem clear:

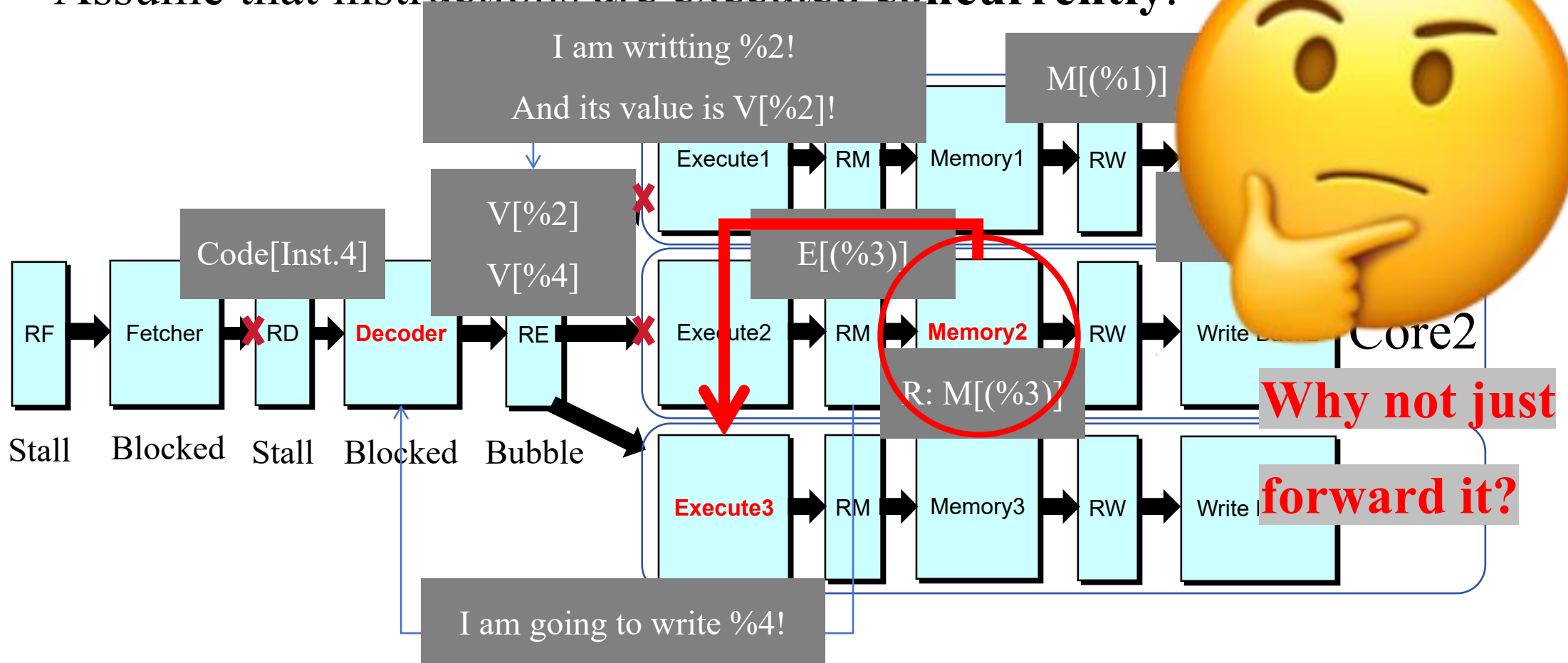
Assume that instructions are executed **concurrently**.

1: `mrmovq (%1), %2`

2: `mrmovq (%3), %4`

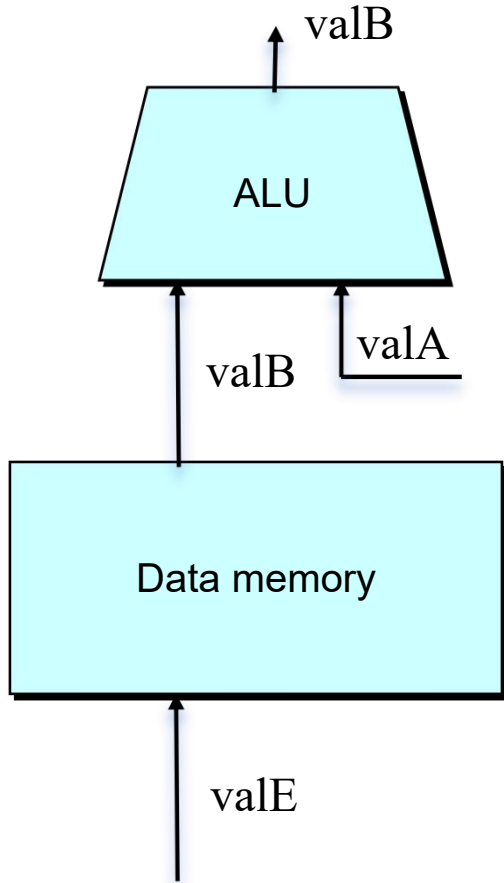
3: `addq %2, %4`

4: `addq %4, (%5)`



Dependancy - Forwarding Is Constrained

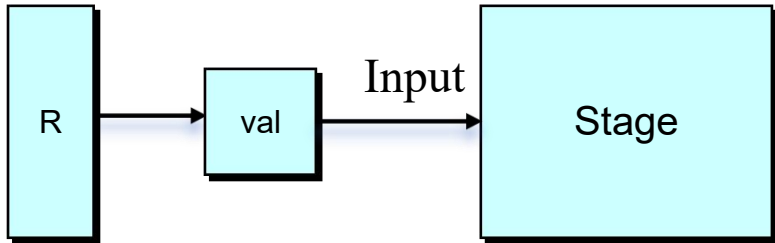
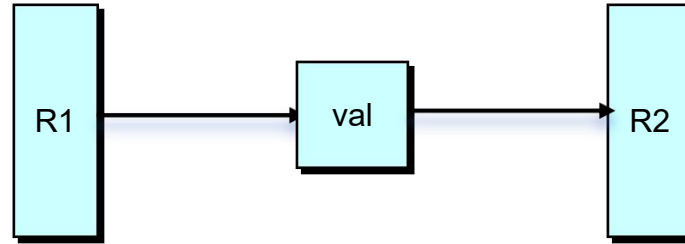
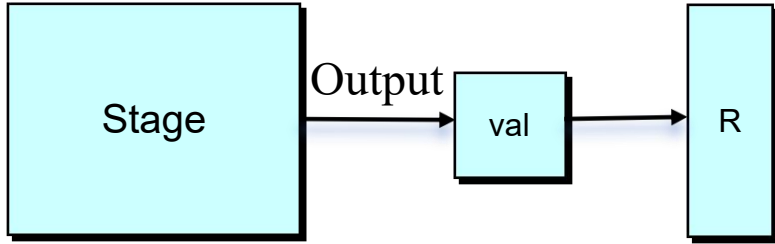
Actually you are doing this...



You are trying to complete **Execute** and **Memory** in a single cycle!

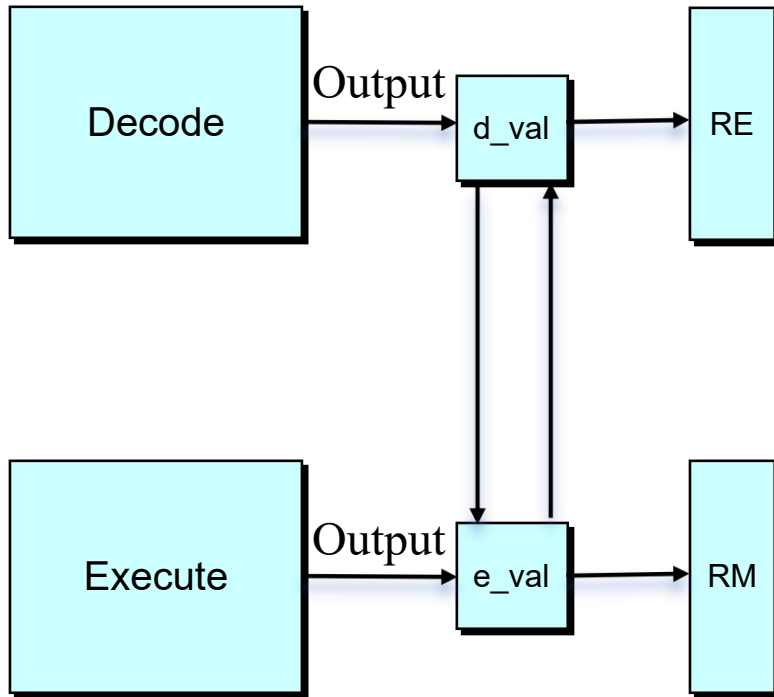
Dependancy - Fowarding Is Constrained

Valid forwading seems like:



Dependancy - Forwarding Is Acyclic

Forwarding below seems valid...

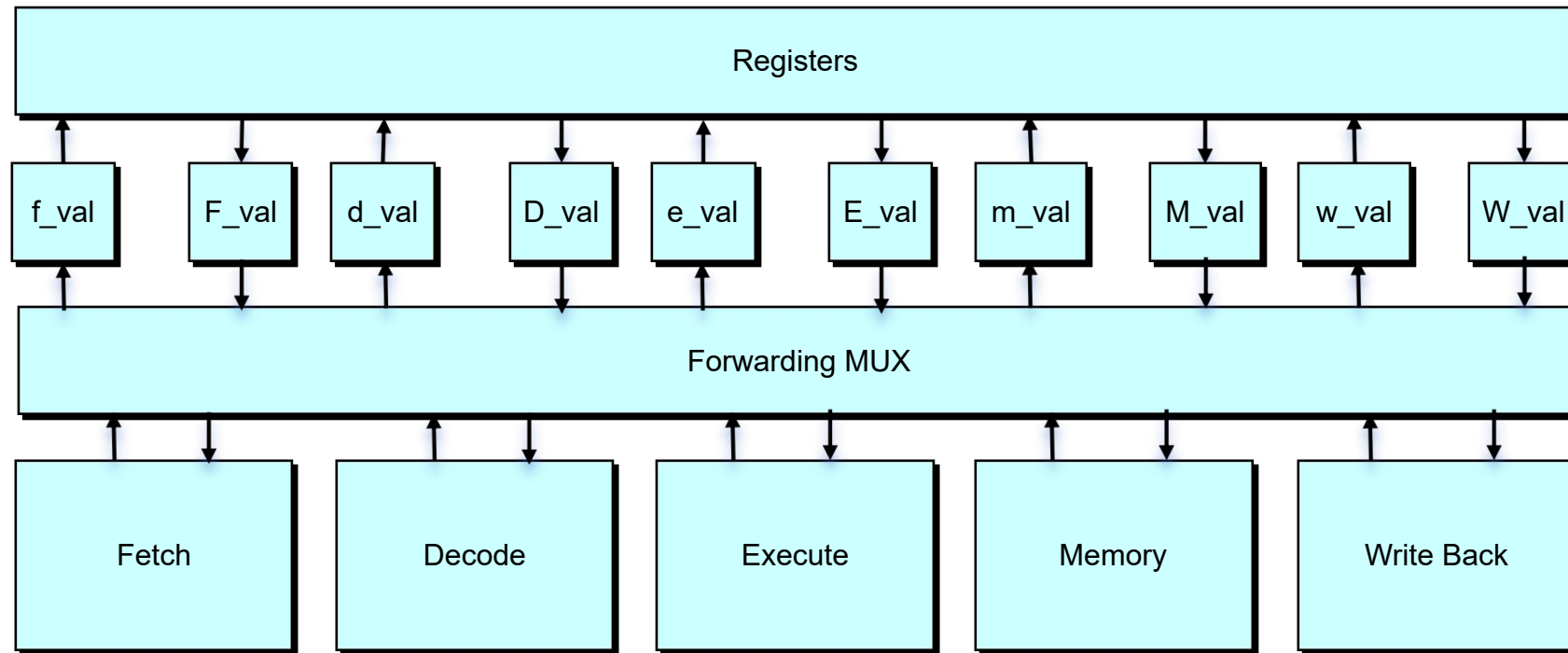


But 'forward' a future message to the past doesn't make sense!

That's why it calls **forwarding**.

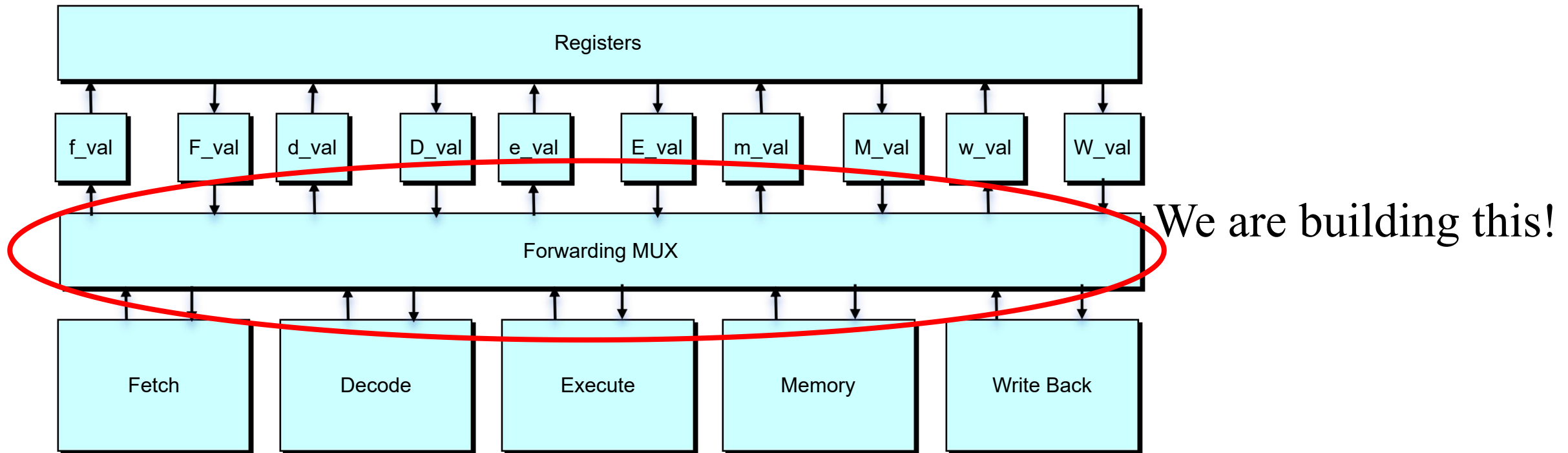
Dependancy - Pipeline with Forwarding

Pipeline with forwarding actually works as:



Dependancy - Pipeline with Forwarding

Pipeline with forwarding actually works as:



Dependancy - Pipeline with Forwarding

What can forwarding do?

Handle RAW (Read after Write) problem.

Inst.4 reads %4.

Actually reads %2 + %4 in Inst.3

Actually reads M[(%1)] + M[(%3)]

To read %2 all these work have to be done.

1: mrmovq (%1), %2

2: mrmovq (%3), %4

3: addq %2, %4

4: rmmovq %4, (%5)

Dependancy - Pipeline with Forwarding

Writes happen on Write Back Stage.

But the value to write is computed on Execute Stage or Memory Stage, so we just forward them.

1: mrmovq (%1), %2

2: mrmovq (%3), %4

3: addq %2, %4

4: rmmovq %4, (%5)

(Black-Board Writting here.)

(TODO: Explain why only Decode Stage needs forwarding.)

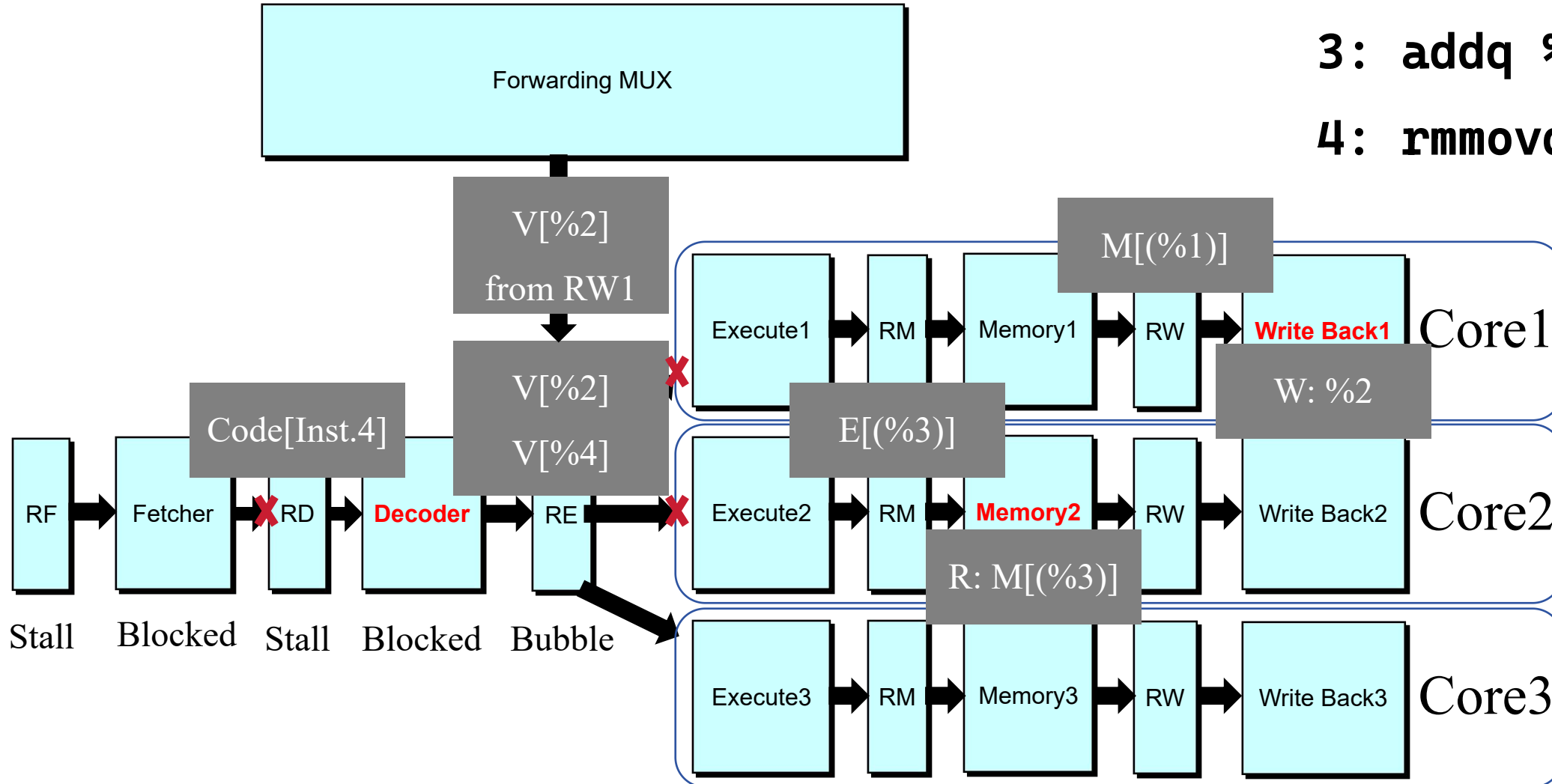
Dependancy - Synchronization

```
1: mrmovq (%1), %2
```

```
2: mrmovq (%3), %4
```

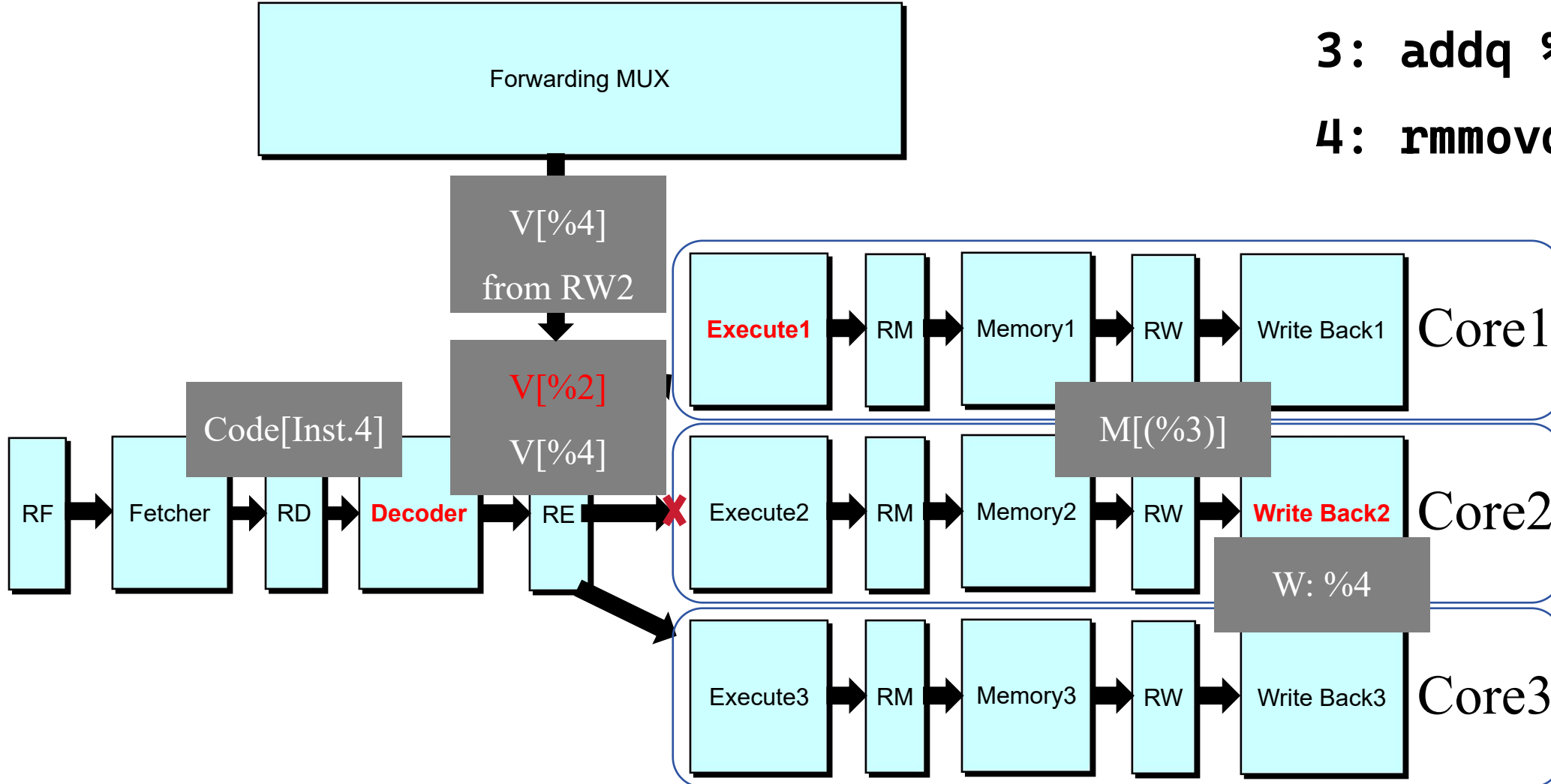
```
3: addq %2, %4
```

```
4: rmmovq %4, (%5)
```



Dependancy - Synchronization

1: `mrmovq (%1), %2`
2: `mrmovq (%3), %4`
3: `addq %2, %4`
4: `rmmovq %4, (%5)`



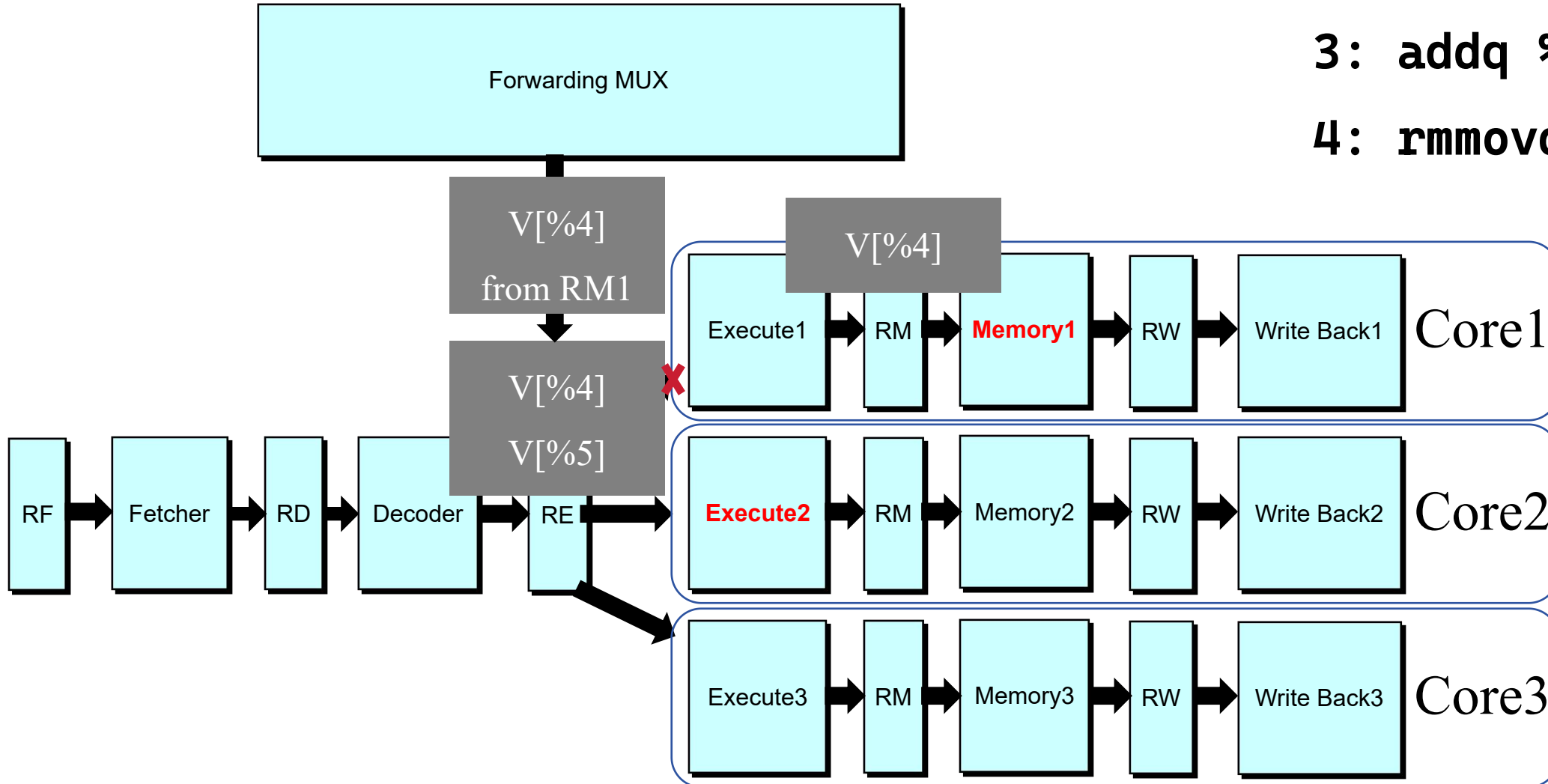
Dependancy - Synchronization

```
1: mrmovq (%1), %2
```

```
2: mrmovq (%3), %4
```

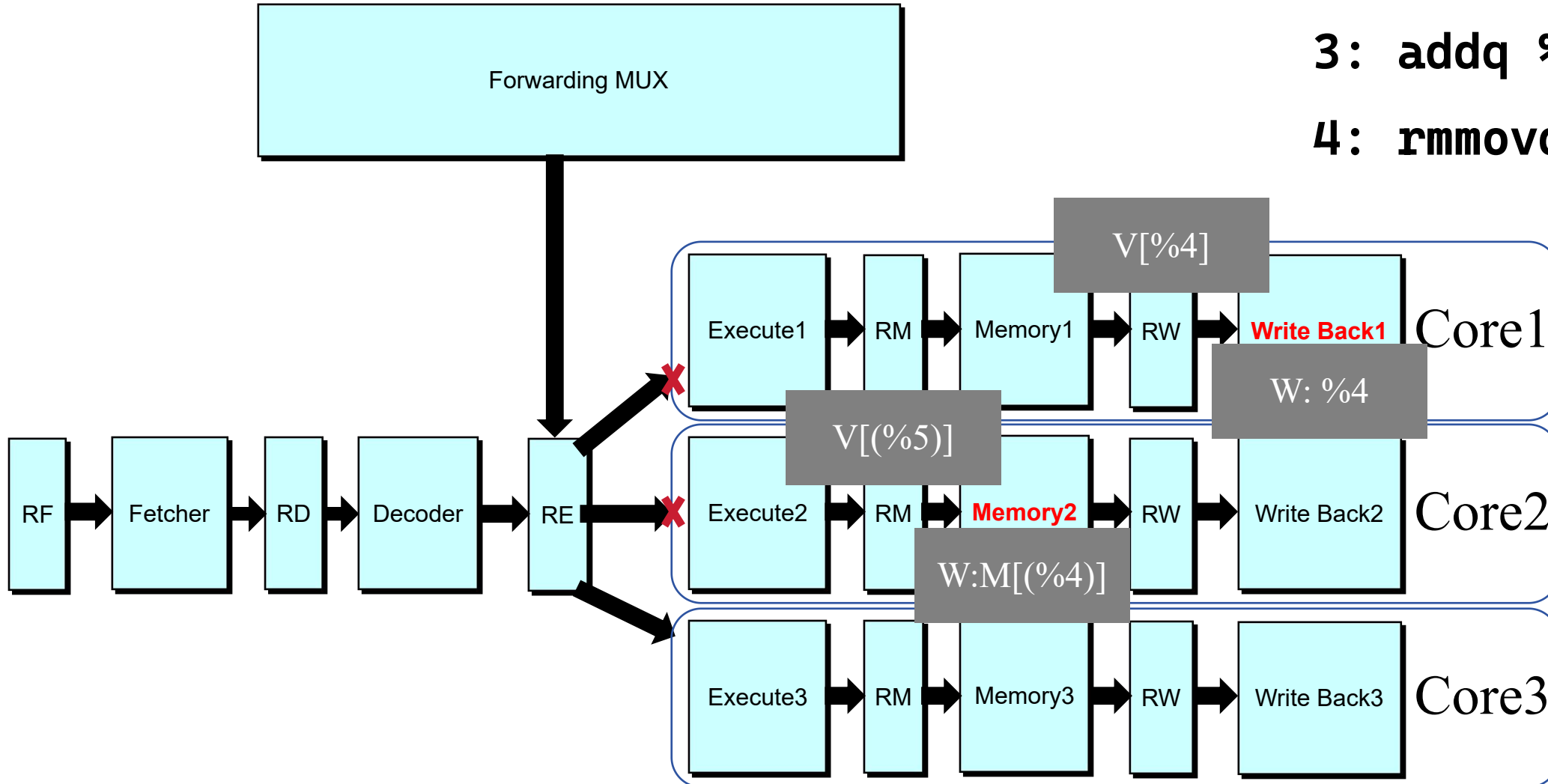
```
3: addq %2, %4
```

```
4: rmmovq %4, (%5)
```



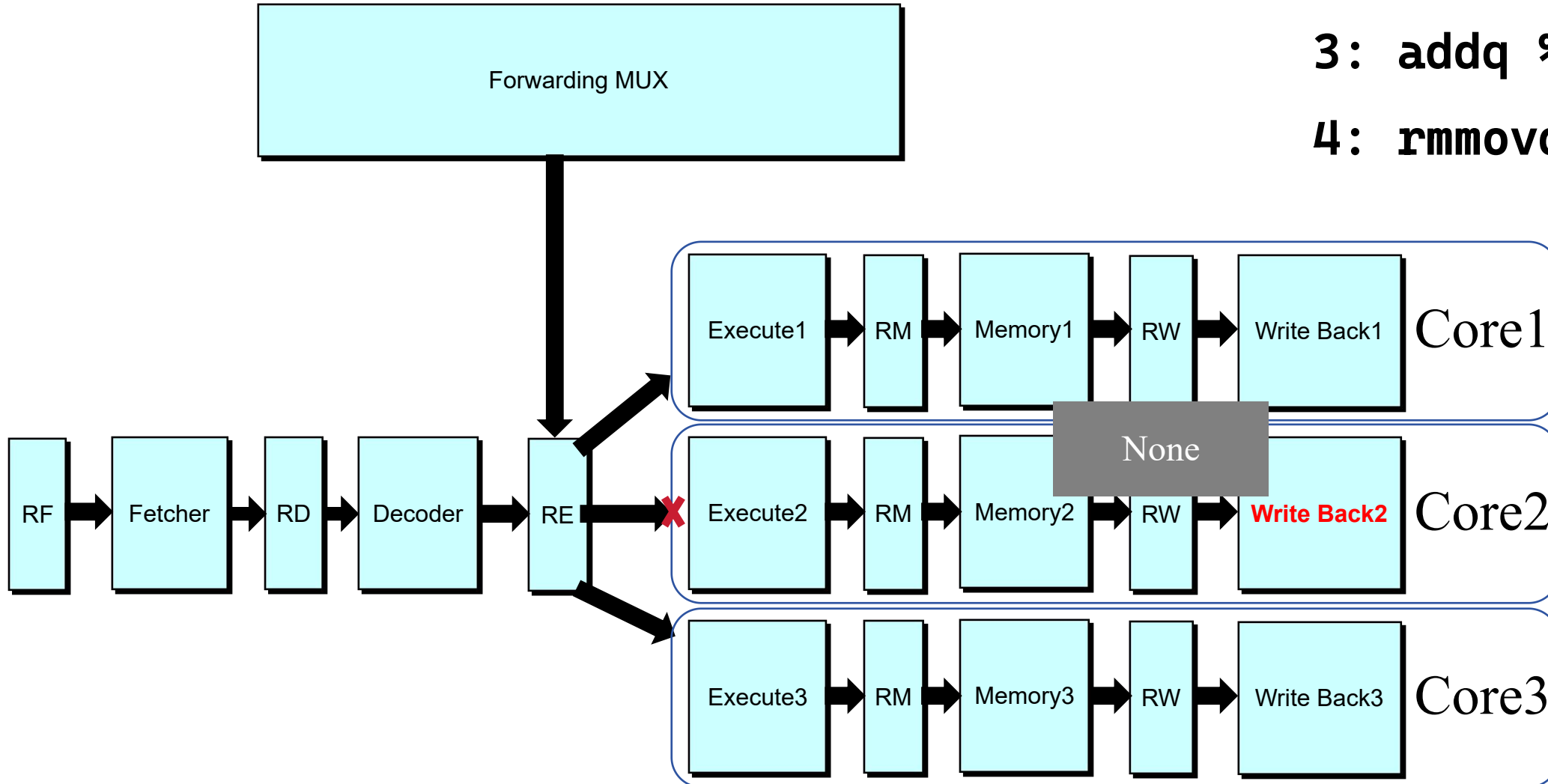
Dependancy - Synchronization

1: `rrmovq (%1), %2`
2: `rrmovq (%3), %4`
3: `addq %2, %4`
4: `rmmovq %4, (%5)`



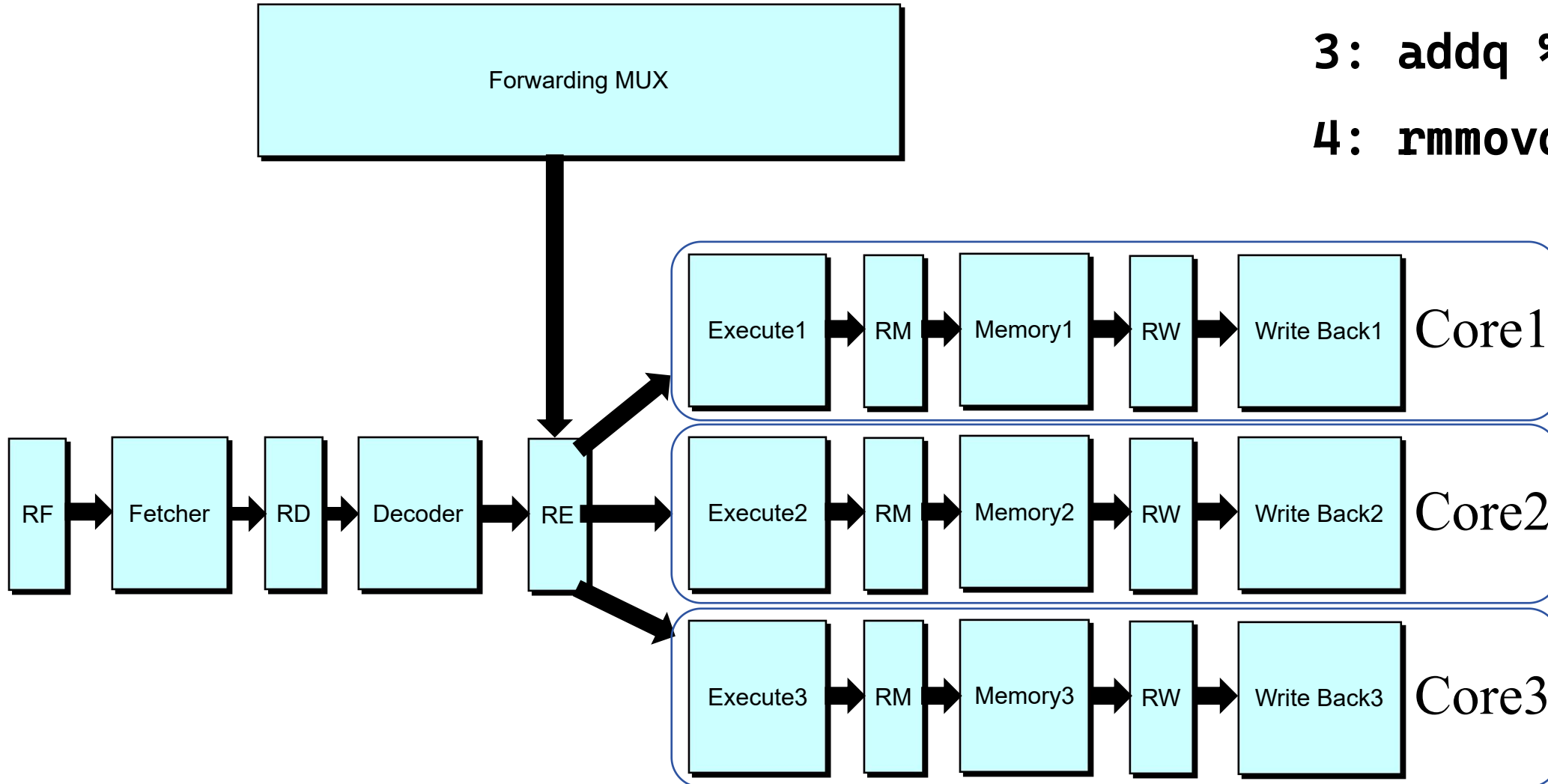
Dependancy - Synchronization

1: `rrmovq (%1), %2`
2: `rrmovq (%3), %4`
3: `addq %2, %4`
4: `rmmovq %4, (%5)`



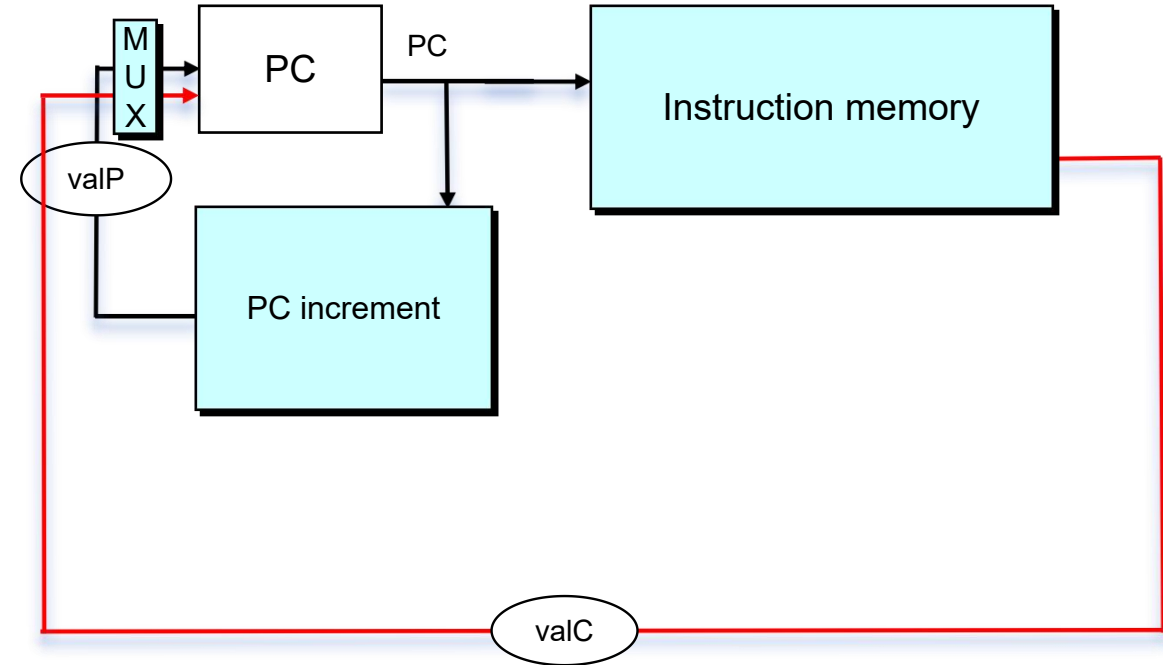
Dependancy - Synchronization

1: `rrmovq (%1), %2`
2: `rrmovq (%3), %4`
3: `addq %2, %4`
4: `rmmovq %4, (%5)`



Breaking Into - Jmp

Jmp could be finished in Fetch Stage.

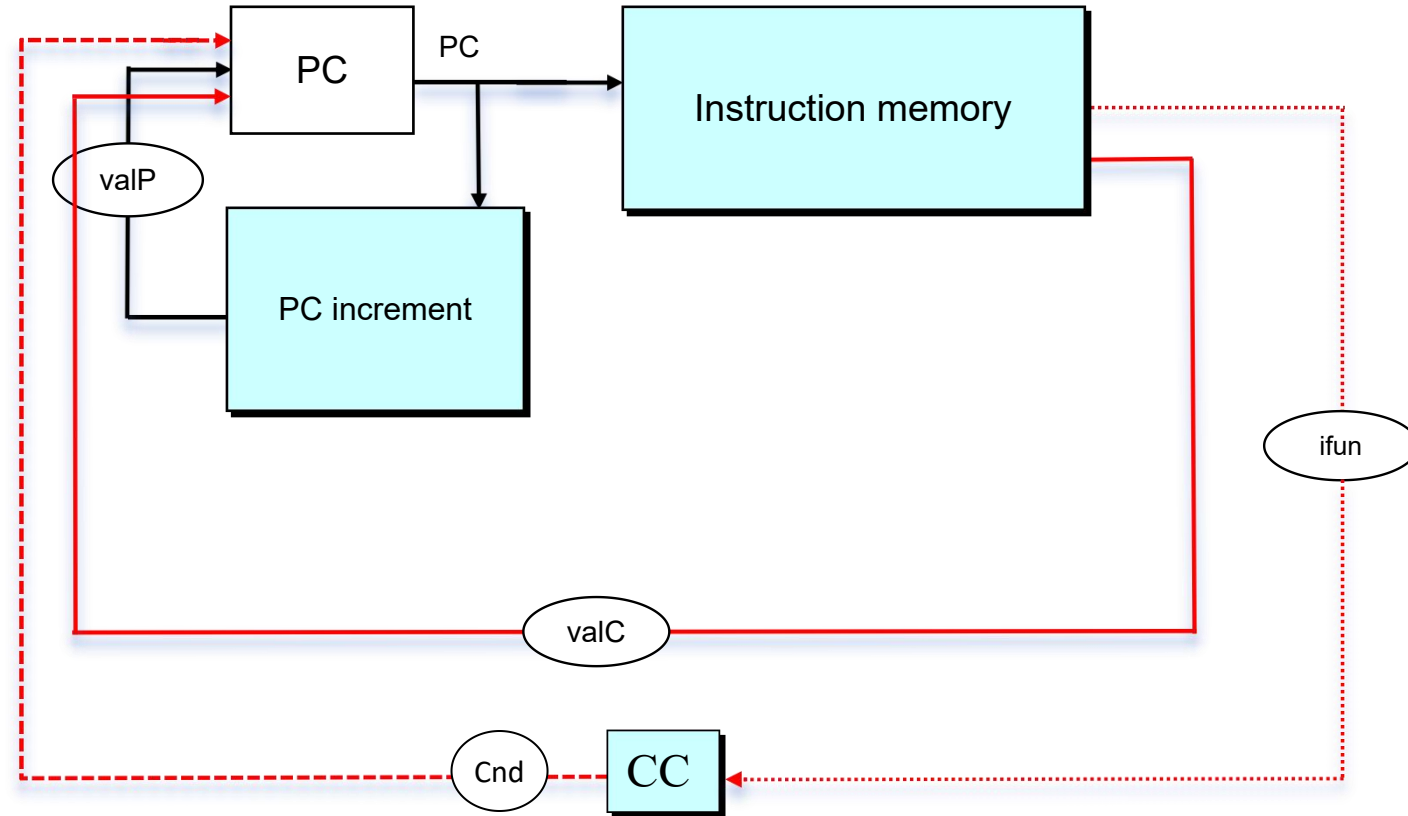


Breaking Into - Jne

Jne needs value of CC...

Ok we can forward it.

But is it that easy?



Breaking Into - Jne

Inst.1 writes CC.

Inst.2 reads CC.

A RAW problem of CC!

```
1: subq %1, %2
2: jne .L1
3. rrmovq %2, %1
.L1:
4. addq %1, %1
```

Just like normal RAW problem, we could stall and wait the CC's new value.

Breaking Into - Jne

Different from memory read, CC's result is simple.

Memory Read Result: 64 bits data, 2^{64} possible values.

CC Read Result: 1 bit data, 2 possible values.

```
1: subq %1, %2
2: jne .L1
3. rrmovq %2, %1
.L1:
4. addq %1, %1
```

If we randomly guess a result of CC, we have 50% possibility to be right!

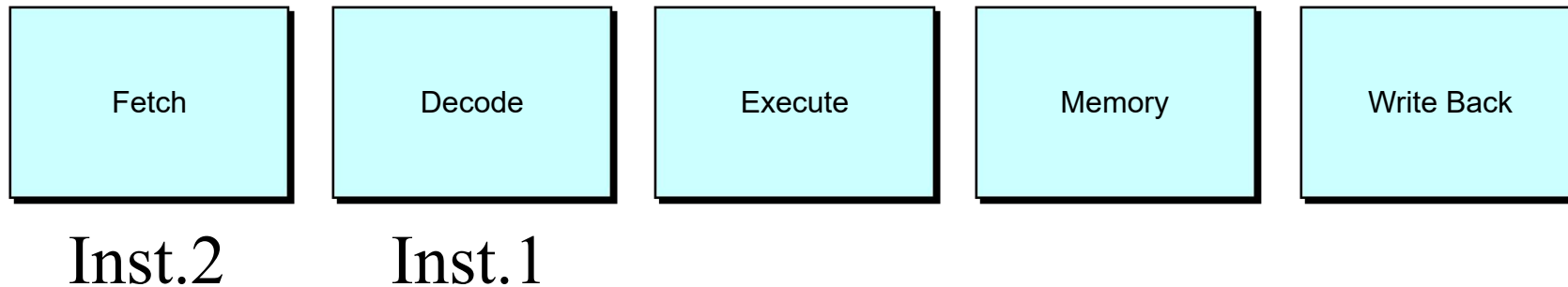
And if we are right, we don't need to stall our pipeline at all!

Breaking Into - Jne

What if we are wrong?

Assume that we guess Inst.2 doesn't jump,
but actually it should jump.

```
1: subq %1, %2  
2: jne .L1  
3: rrmovq %2, %1  
.L1:  
4: addq %1, %1
```



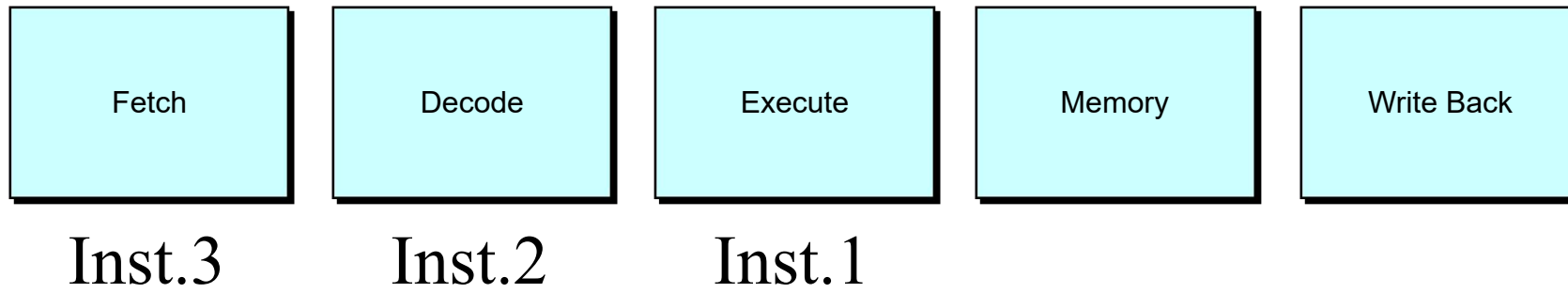
Breaking Into - Jne

Here we get CC's real value and realize that we are wrong.

At the same time, Inst.3 is wrongly fetched.

We get an **error**!

```
1: subq %1, %2
2: jne .L1
3. rrmovq %2, %1
.L1:
4. addq %1, %1
```

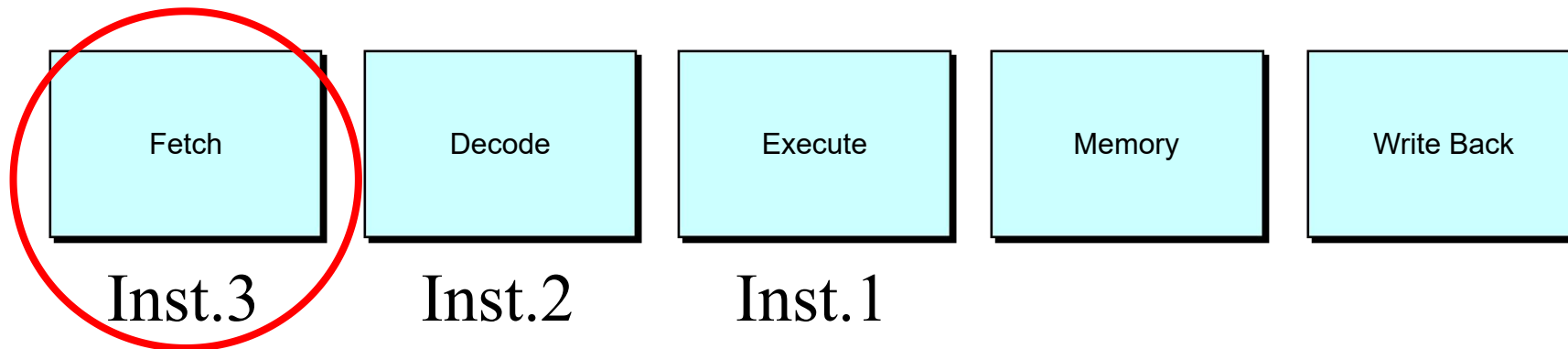


Error Handling

If an error is detected, we have to handle it to protect the correctness of our pipeline!

We firstly try to **recover** the error, which is to make the error (seem to be) not happen at all.

If our recovering failed, we have to **shut down** our pipeline. (Fatal Error)



Try to recover it!

Error Handling

To recover an instruction that shouldn't have been executed, we should avoid it **changing the world**.

Remember and remember, only **writes** change the world.

Fetch writes **PC**.

Decode writes nothing.

Execute writes **CC**.

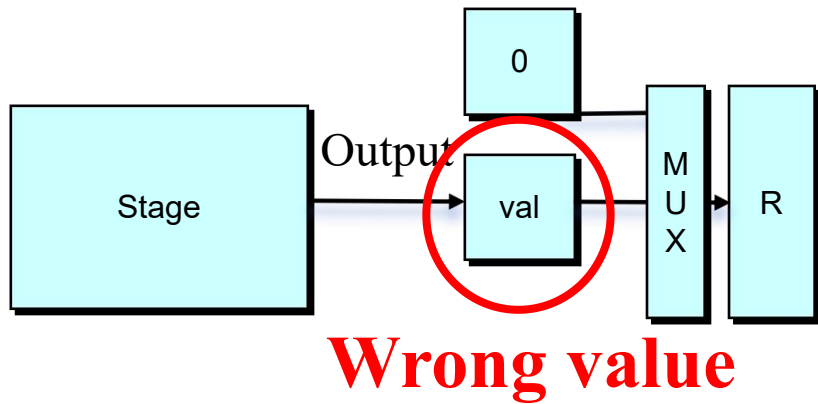
Memory writes **memory**.

Write Back writes **registers**.

All stages write **pipeline registers**.

Error Handling - Recover Pipeline Registers

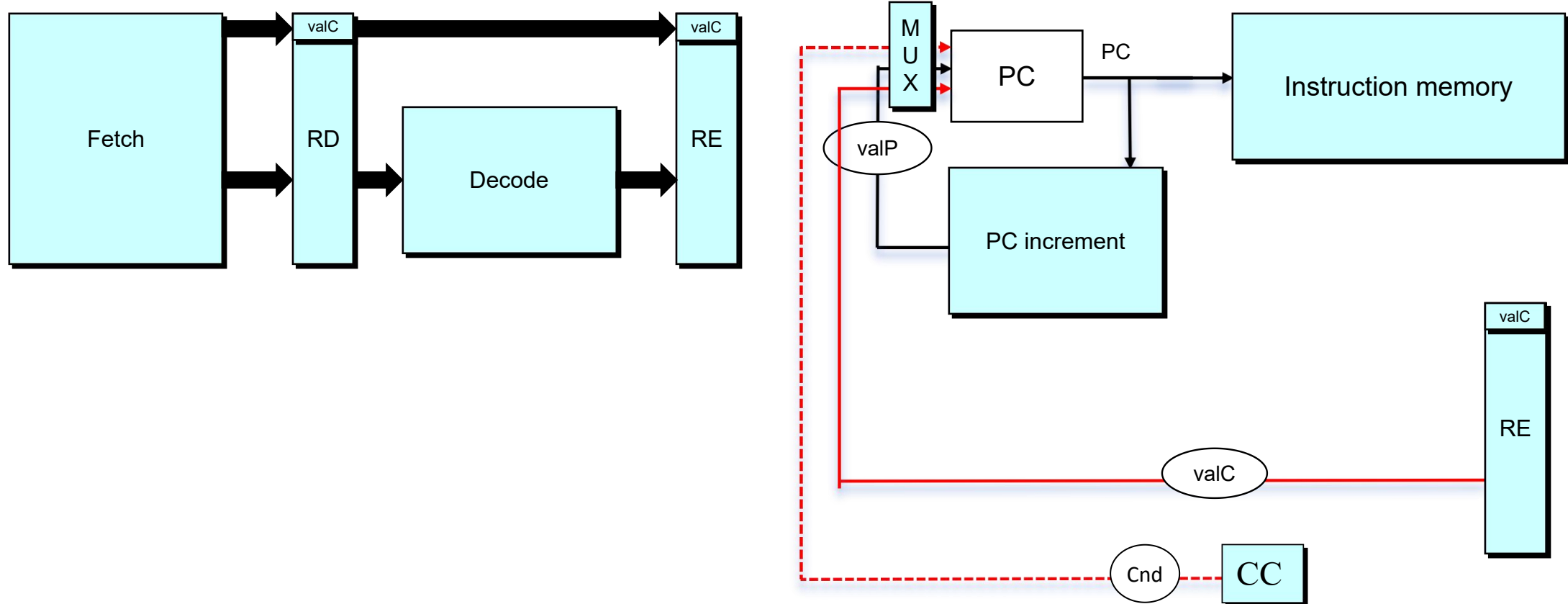
Just bubble it.



Error Handling - Recover Program Counter

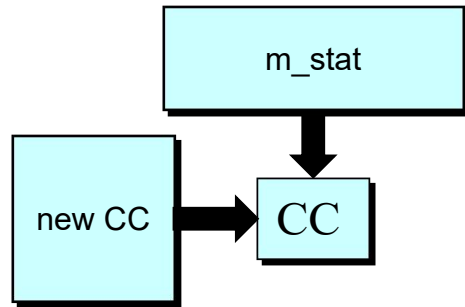
Give it a right value.

So we should keep the possible right value just like how we keep **dstA**.



Error Handling - Recover CC

Y86 pipeline uses a **suppression** implementation to avoid wrong CC being written.



It could only recover error detected at Memory Stage.

Write Back error is fatal.

Error Handling - Recover Memory

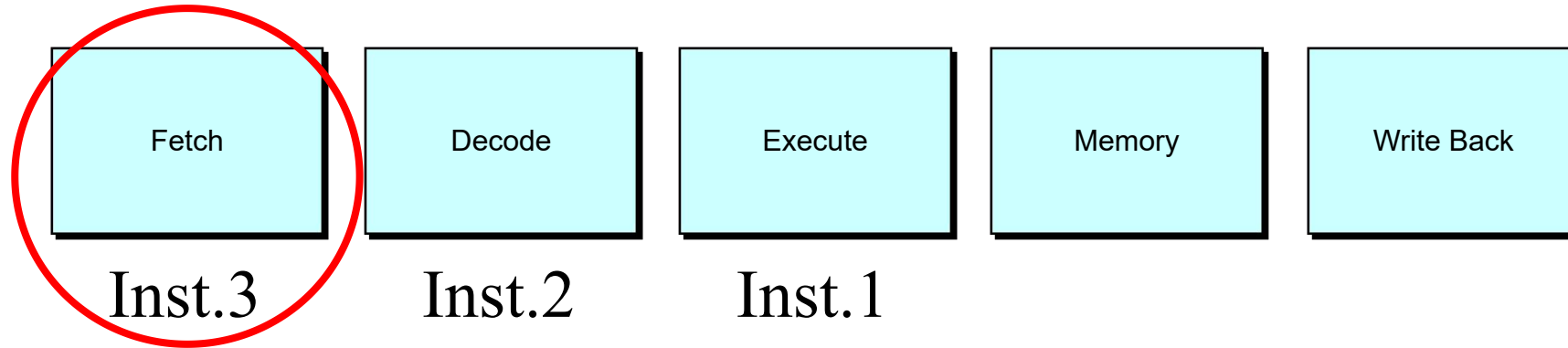
No way to recover, unless we have a copy of the memory.

Write Back error is fatal.

Error Handling - Recover Registers

No need to recover, because Write Back Stage is the last stage.

Error Handling - Case Study: Recover Jne



Try to recover it!

We need to recover RD, by bubbling it.

We need to recover PC, by giving the right value to it.

Time Penalty: 1 Cycle

Error Handling

Since wrong prediction of branch has time penalty...

We want to enhance the prediction accuracy.

It is called, Branch Prediction.

Takeaway

Concepts:

Cache

RAW Problem

Stall, Bubble

Forwarding

Error Handling

Branch Prediction