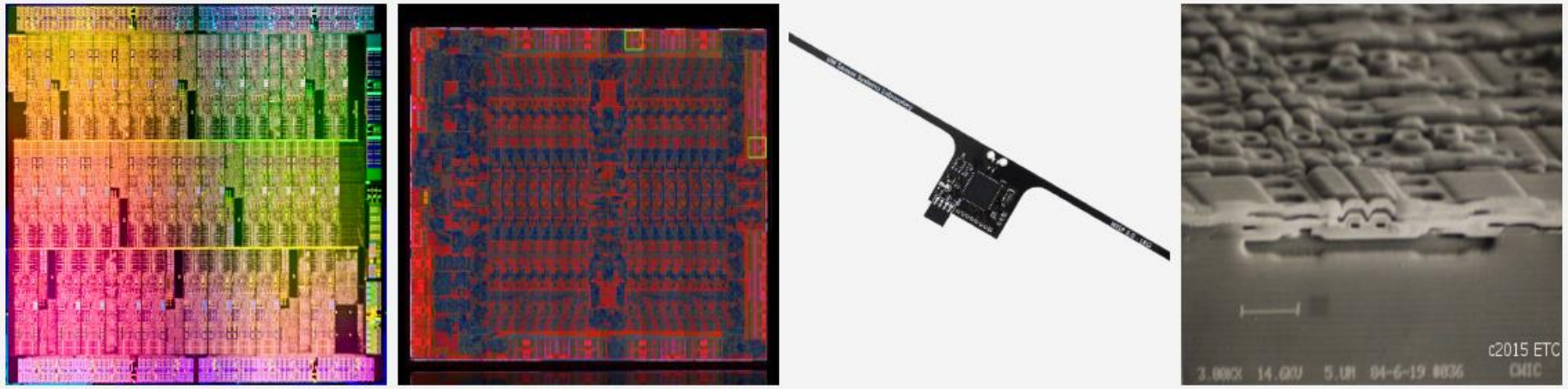
18-344: Computer Systems and the Hardware-Software Interface

Fall 2023



Course Description

Lecture 11: Advanced Architecture: Superscalar and Out of Order

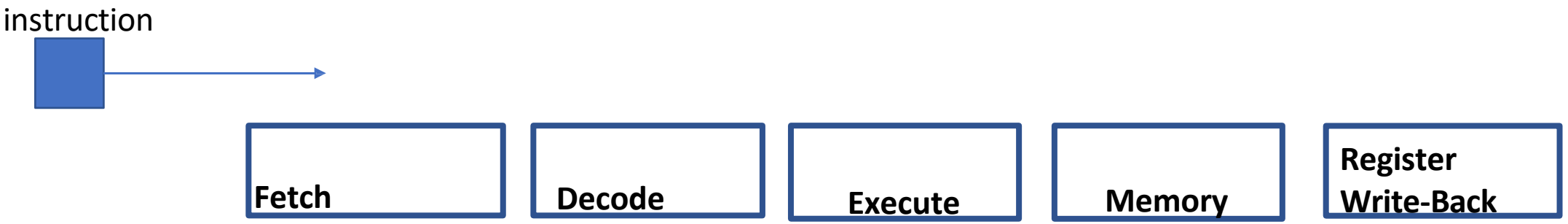
This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

Credit: Brandon Lucia

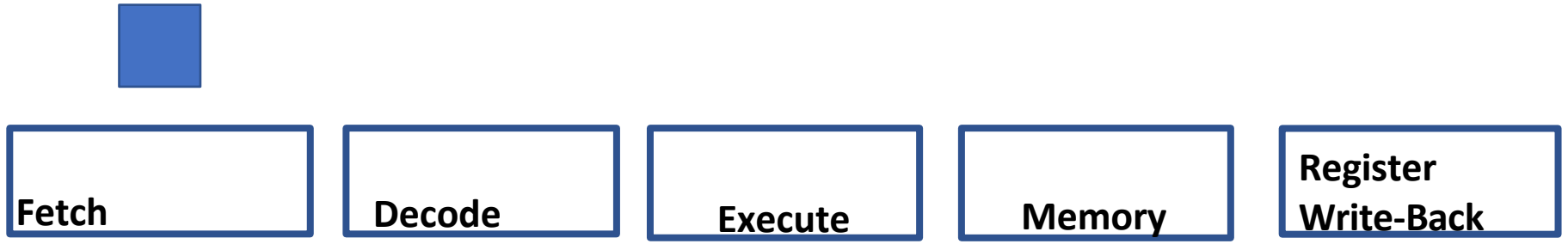
Today: Advanced Microarchitecture Techniques

- Advanced Instruction-Level Parallelism: Multiple Issue, Out of Order Execution, Register Renaming, SMT

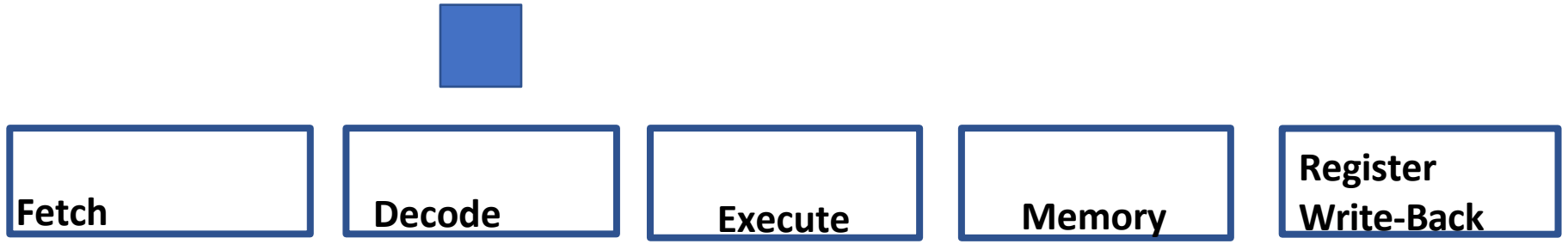
Pipelined scalar design



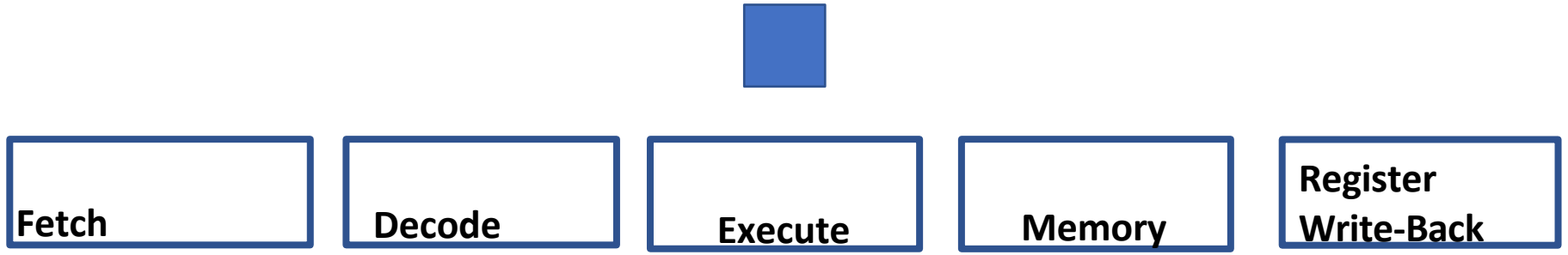
Pipelined scalar design



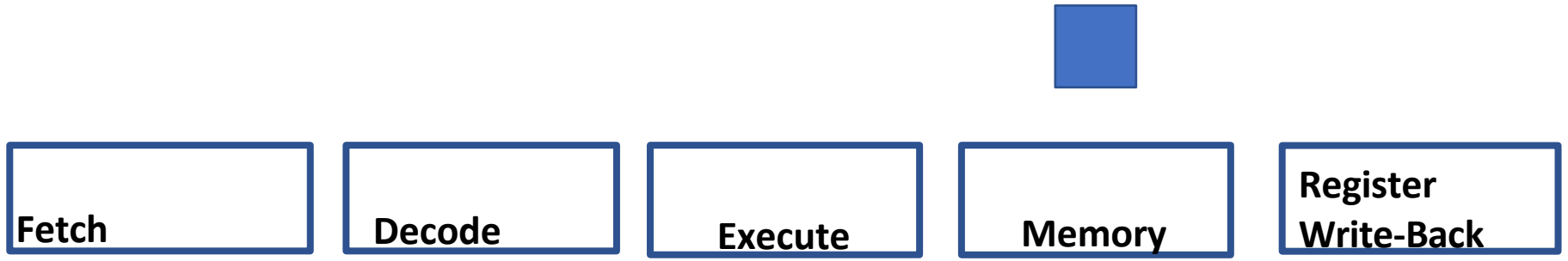
Pipelined scalar design



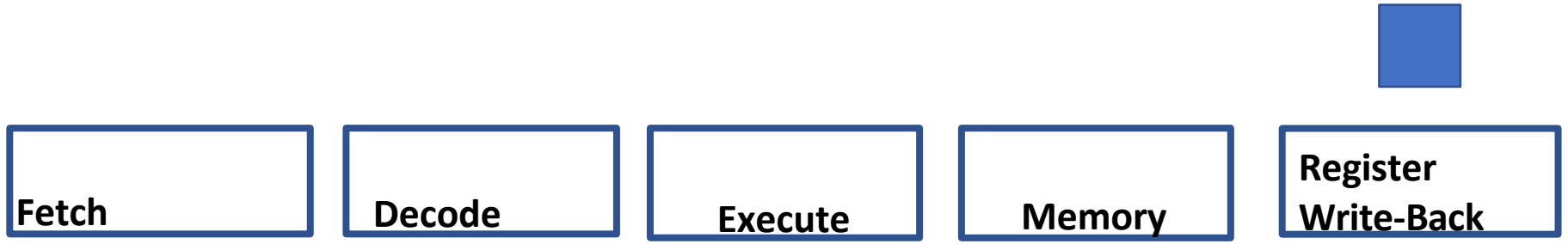
Pipelined scalar design



Pipelined scalar design

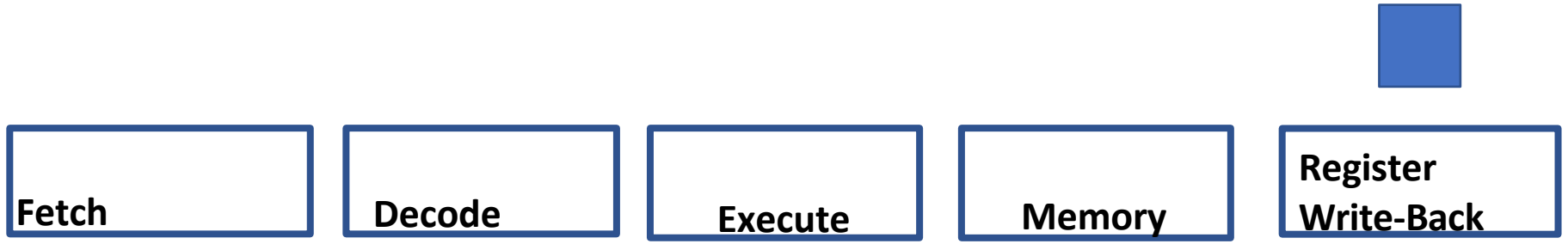


Pipelined scalar design



What is the best performance that we can ever get out of a pipeline like the one we have been studying?
(how do we answer this question?)

Pipelined scalar design

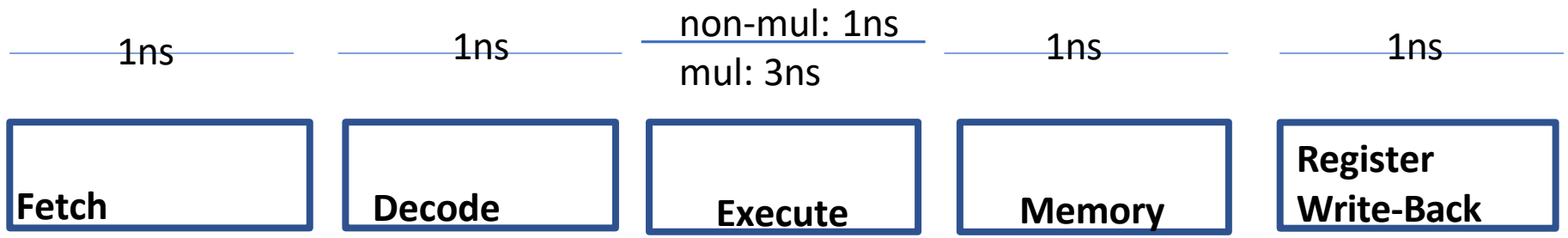


Iron Law of Processor Performance:

Instr / Prog x Cycles / Instr x Seconds / Cycle

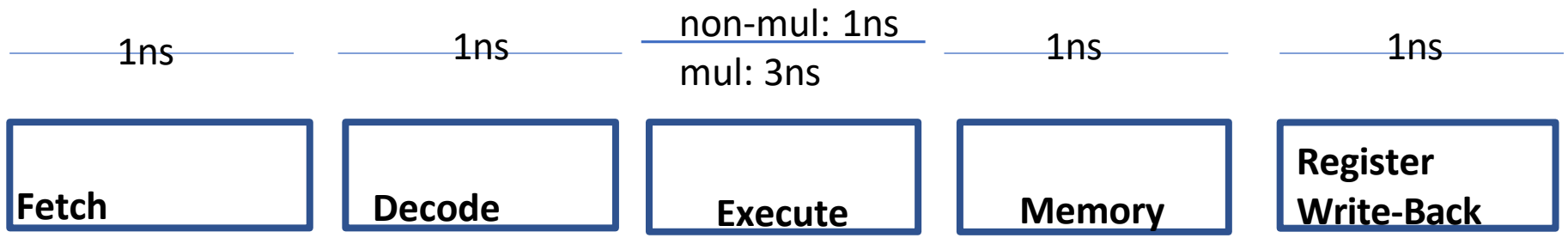
Fundamental limits to each of these terms in our current pipeline?

Thinking about latency (again) to optimize for cycle time



What is the implication of mul having a 3ns latency, compared to the latency of each of the other stages?

Thinking about latency (again) to optimize for cycle time

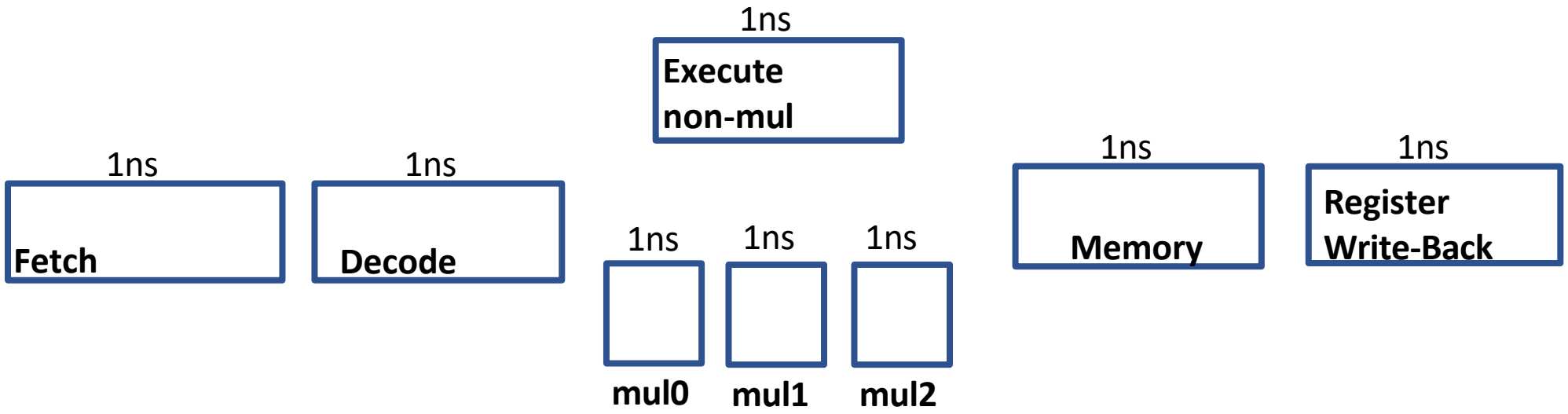


What is the implication of mul having a 3ns latency, compared to the latency of each of the other stages?

333MHz max clock frequency

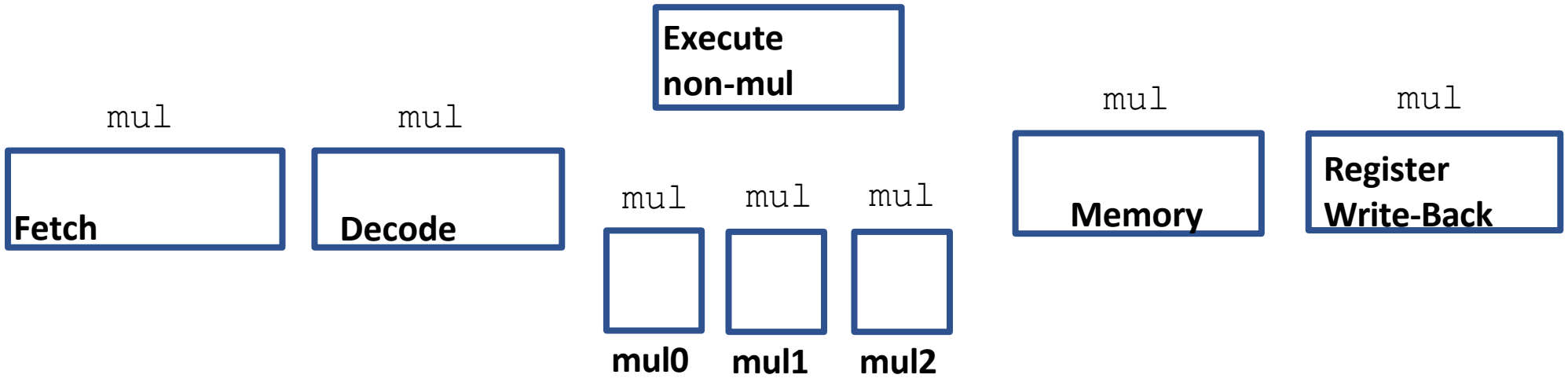
(despite 1GHz being OK for non-mul operations)

What if we pipeline the multiplier independently?



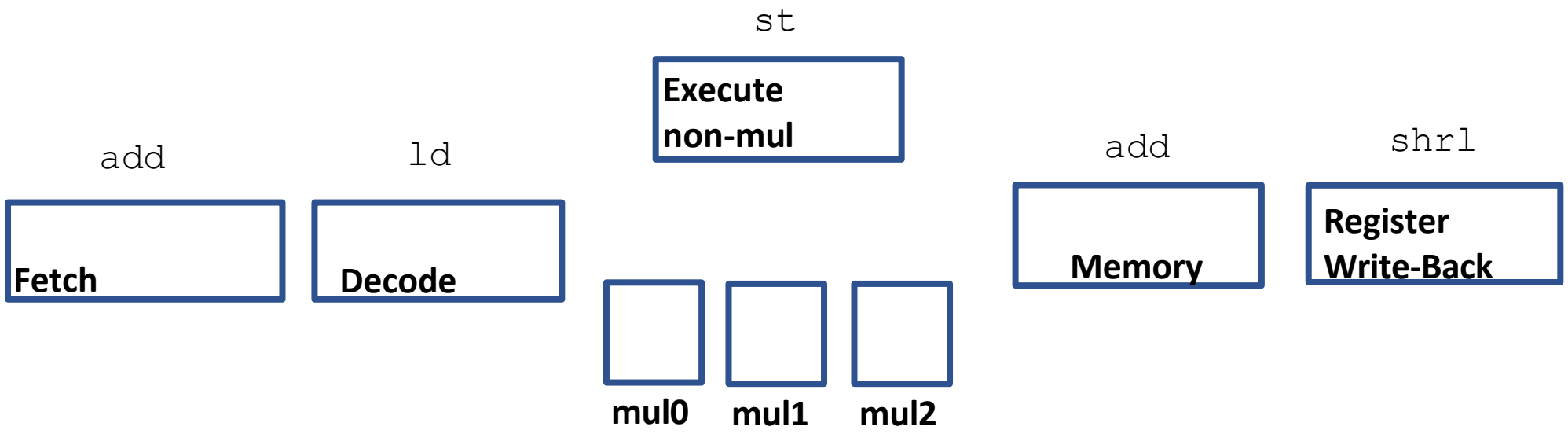
Break the multiply unit into 3 parts, each of which takes 1ns, equalizing all stages' latencies

What if we pipeline the multiplier independently?



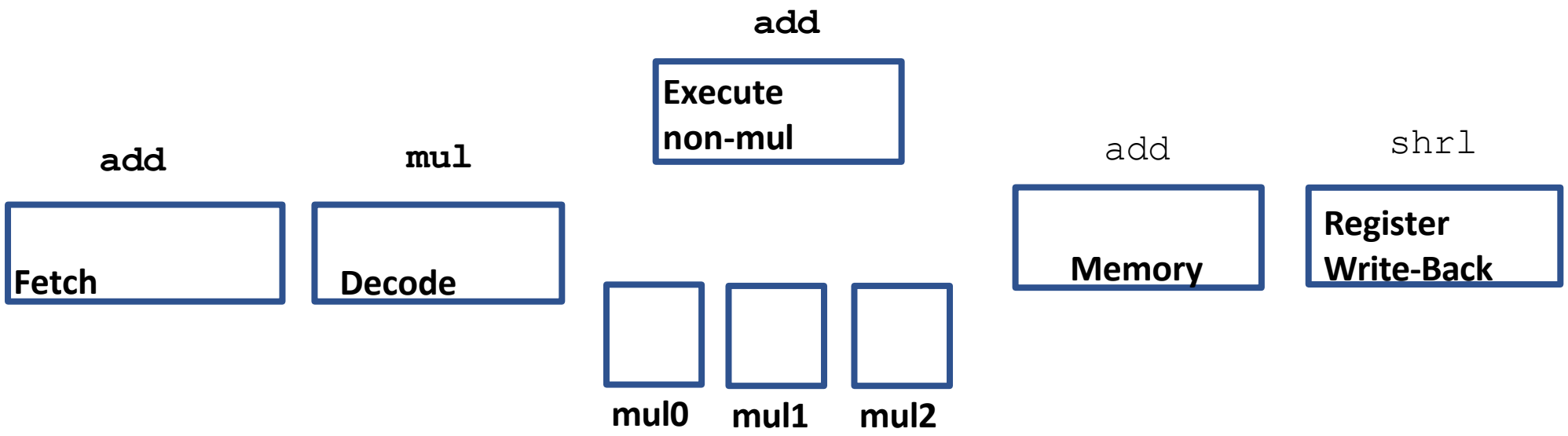
Back-to-back multiplies keep the mul pipe full, at 1GHz latency

What if we pipeline the multiplier independently?



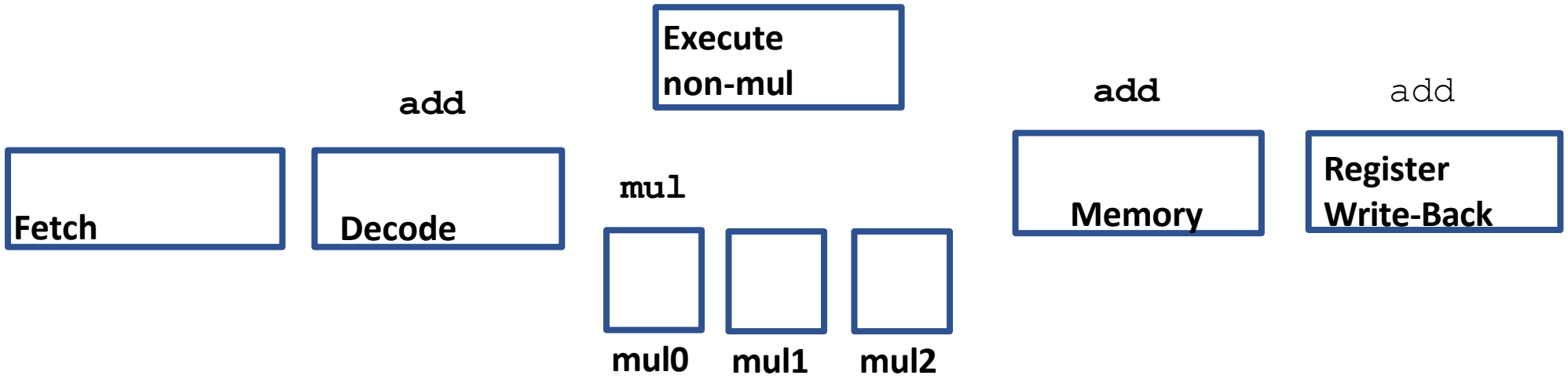
Back-to-back non-mul ops keep the pipe full, at 1GHz latency

What if we pipeline the multiplier independently?



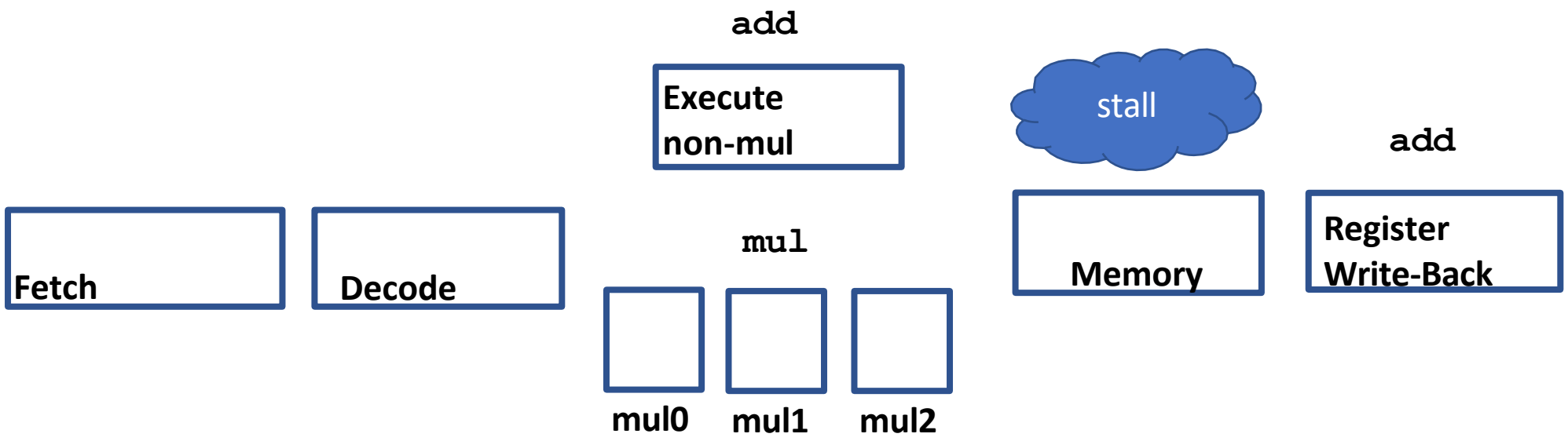
Question: What about add mul add mul?

What if we pipeline the multiplier independently?



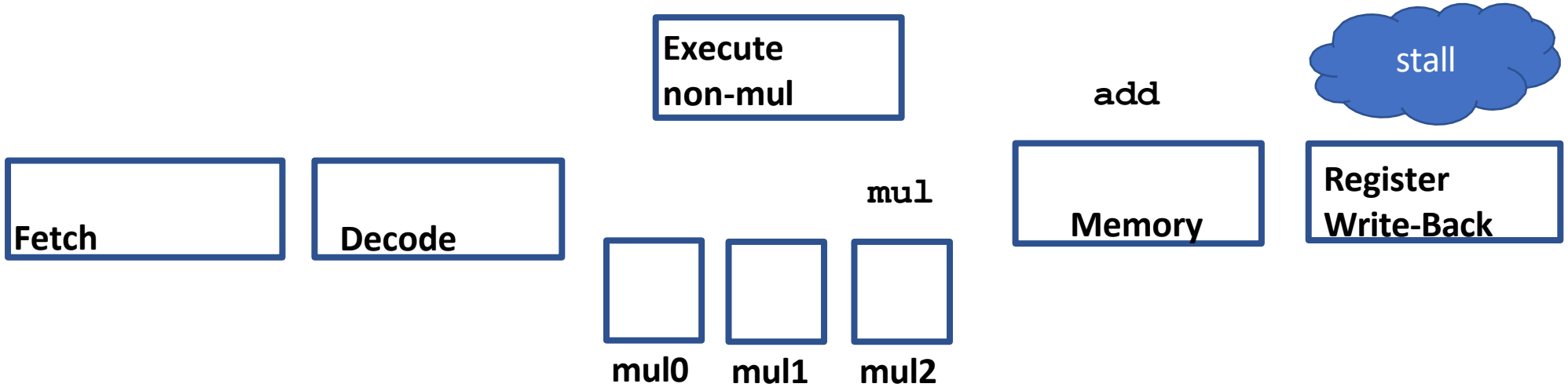
Question: What about add mul add mul?

What if we pipeline the multiplier independently?



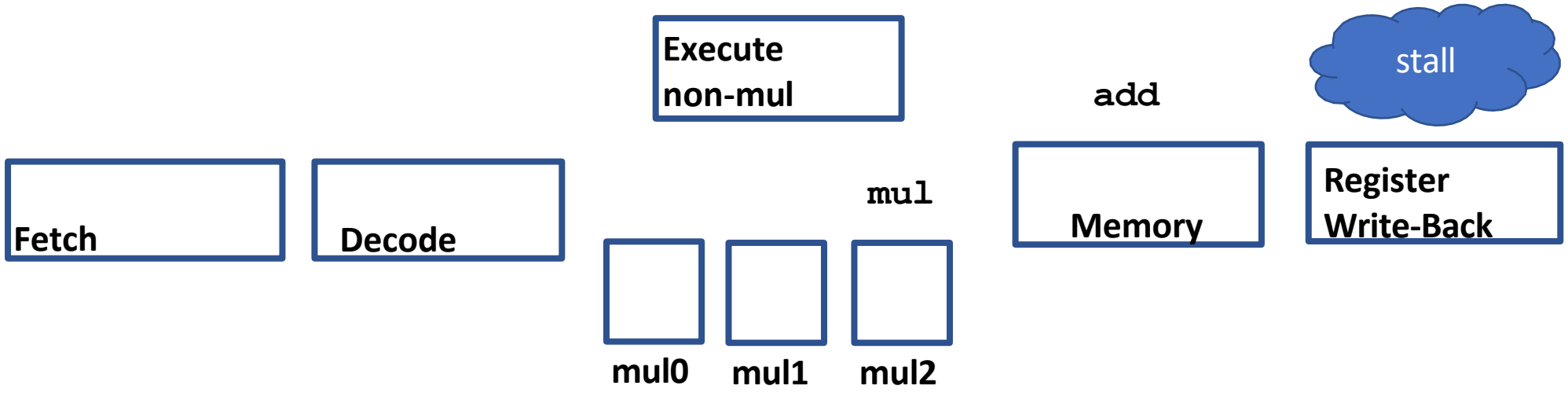
Question: What about add mul add mul?

What if we pipeline the multiplier independently?



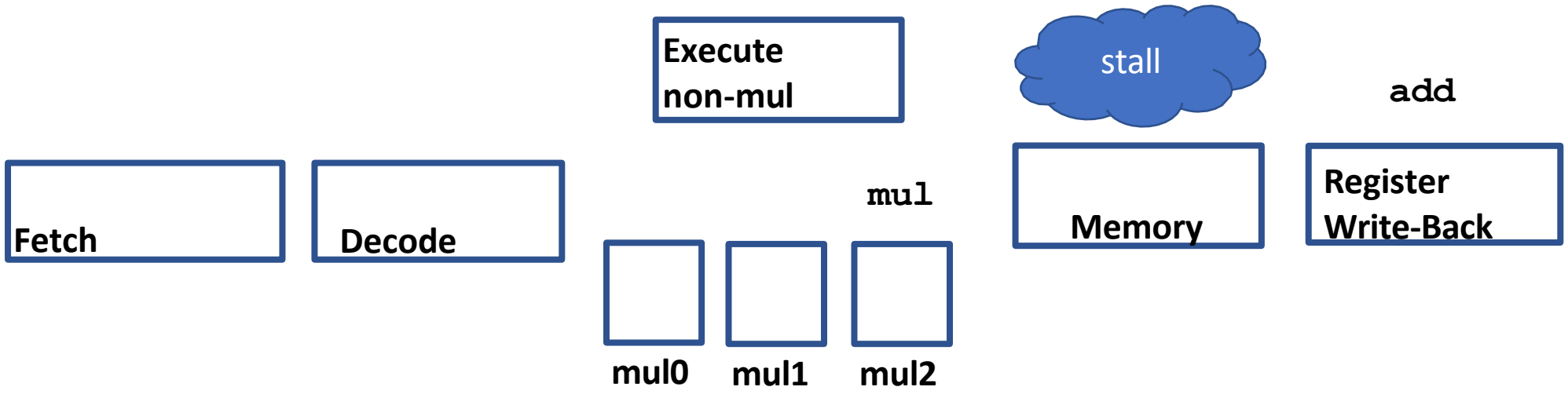
Problem?

Instructions might complete out of order if we are not careful!



In addition to the unfortunate **stall in the memory stage**, the add and the mul **execute in the wrong order**!

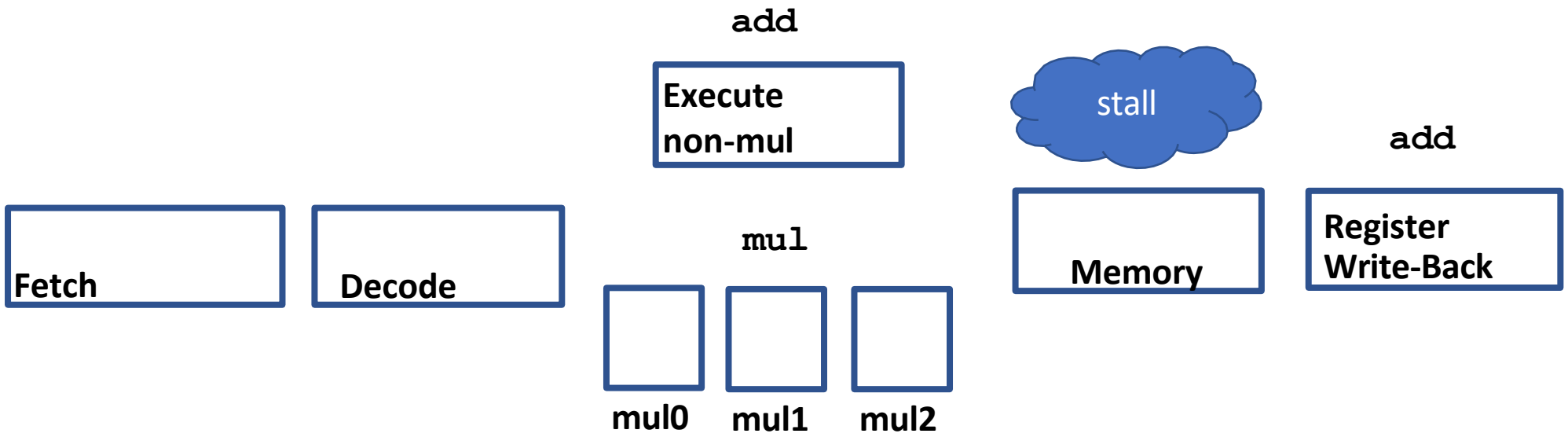
Avoiding out-of-order completion



Hard to avoid the *stall…*

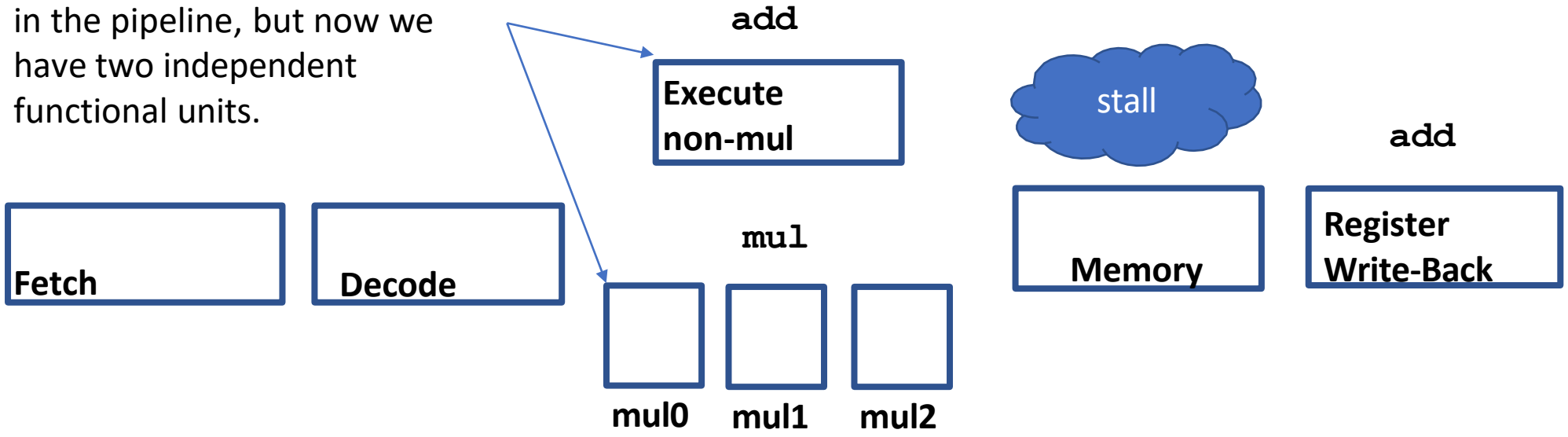
Can avoid the *ordering problem* with extra stall logic in **Ex**

Let's Rewind: Anything interesting about this snapshot in time?



Independent FUs allow us to optimize IPC directly by increasing ILP

Until now, we've considered a single ALU in a single **Ex stage** in the pipeline, but now we have two independent functional units.



This pipeline is **Ex**ecuting multiple instructions at the same time on different functional units. **ILP begets IPC!**

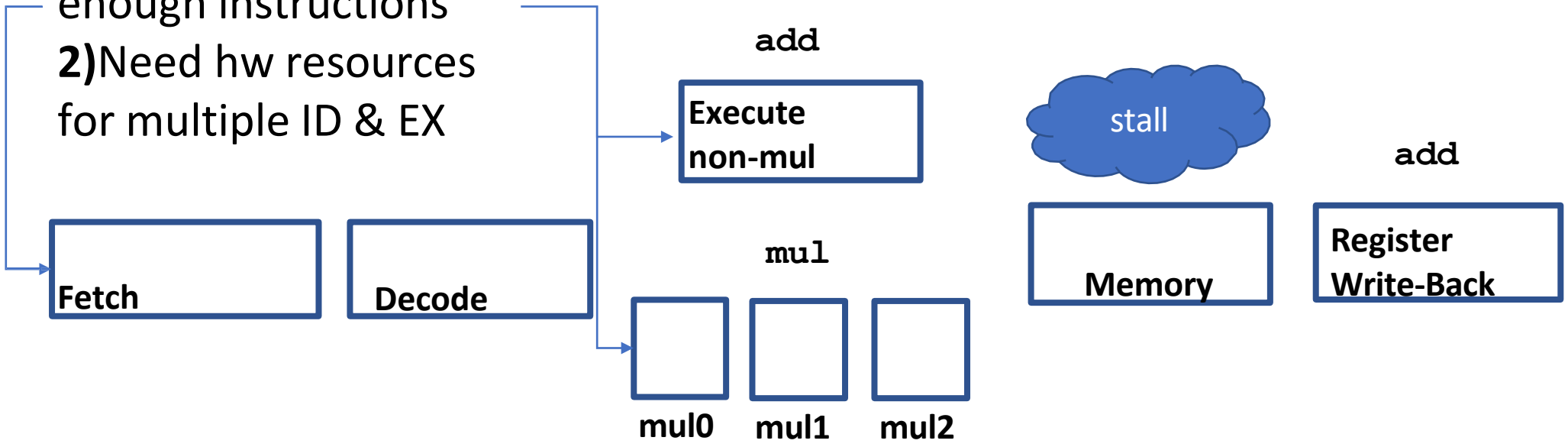
Superscalar Out of Order Execution

A Superscalar Processor Executes Multiple Instructions at the Same Time

Front End Challenges:

1)Need to supply enough instructions

2)Need hw resources for multiple ID & EX

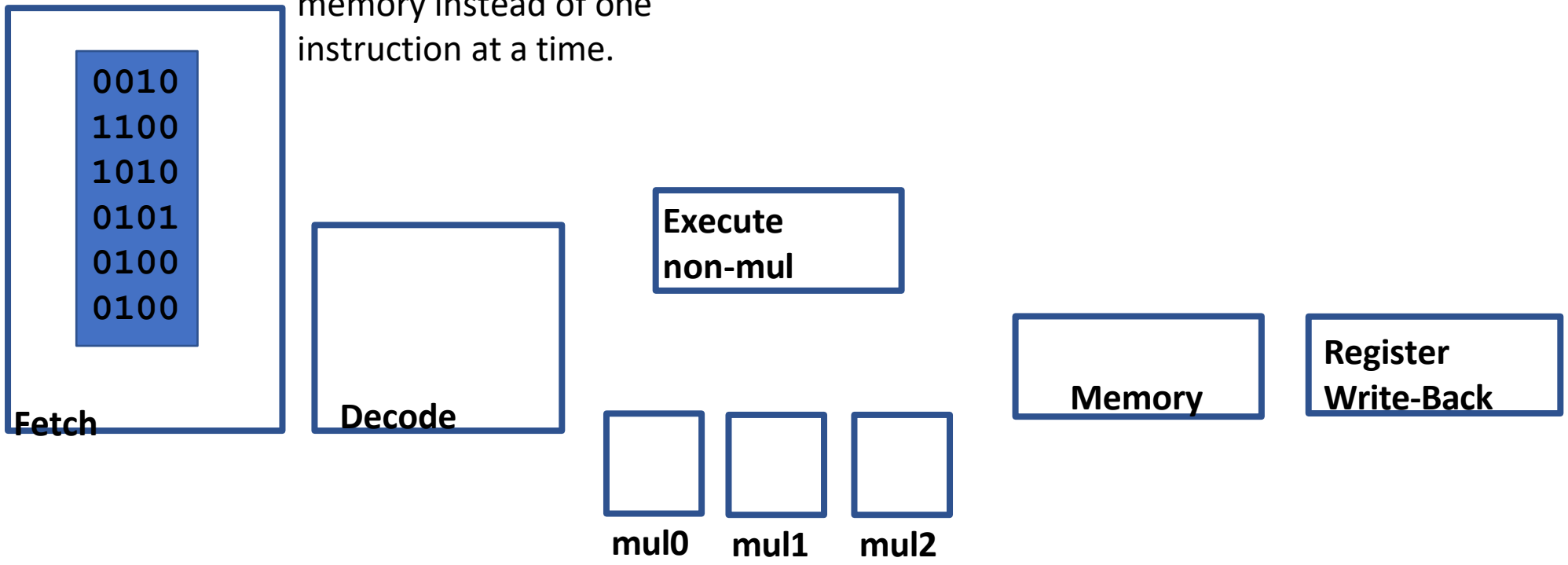


Scalar executes one instruction at a time

Superscalar executes multiple instructions at a time

Superscalar processors

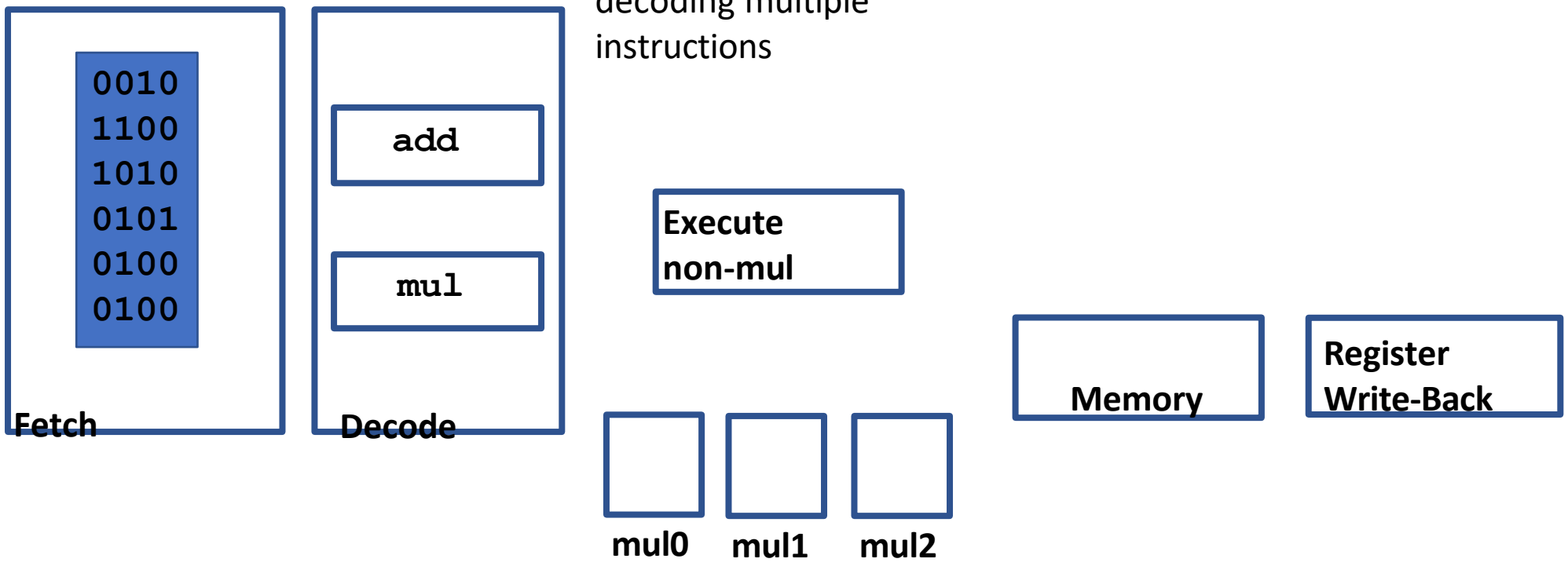
First idea: fetch a block of data from instruction memory instead of one instruction at a time.



(Here, we give up on the detailed pipeline diagram due to the increased complexity of the design.)

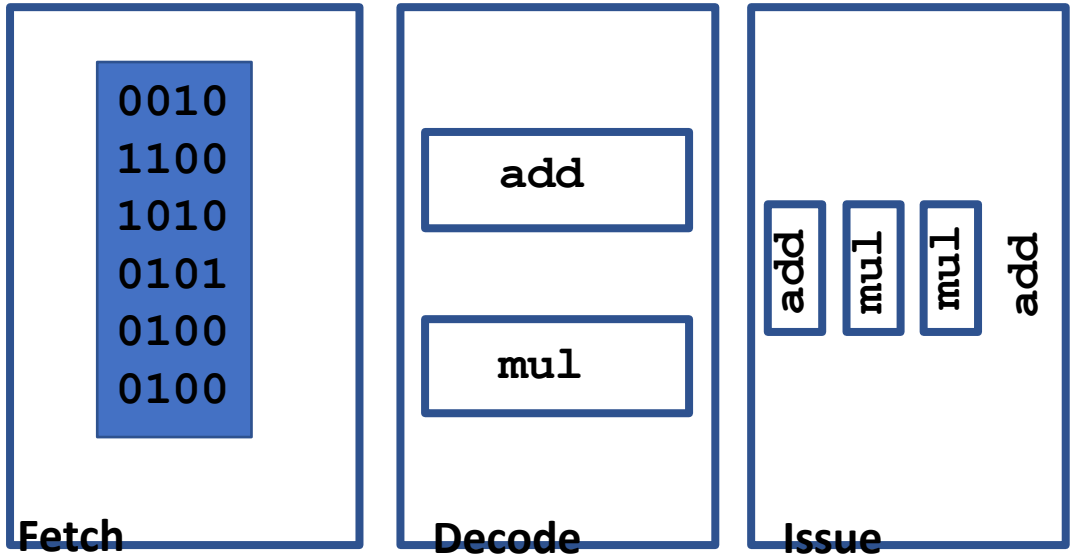
Superscalar processors

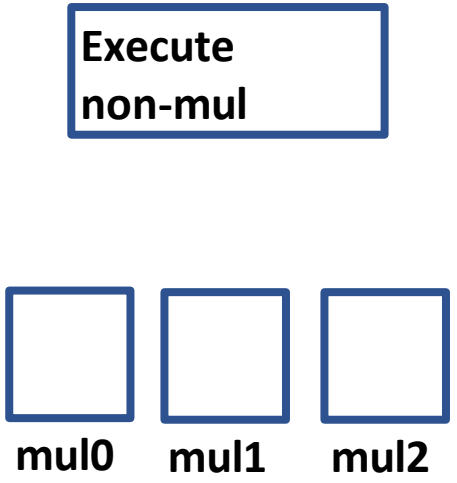
Second idea: Replicate decode logic to allow decoding multiple instructions

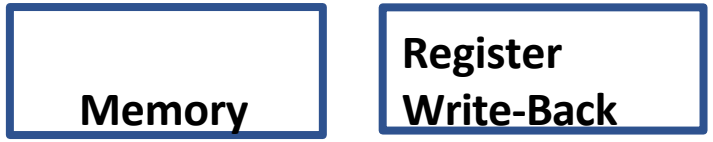


Superscalar processors

Third idea: Add ***issue queue*** of instructions ready to issue & logic to check whether they can issue together







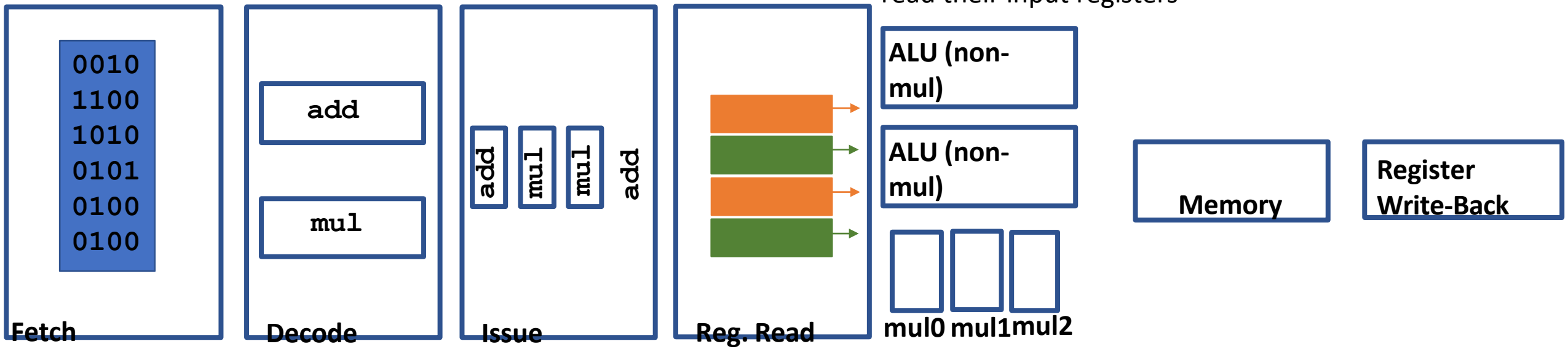

```
add x6  x8  x11
add x12 x6  x13
mul x7  x12 x14
```

These instructions **cannot** issue together (why? two reasons, actually!)

Question: how much checking required for n-wide issue?

Superscalar processors

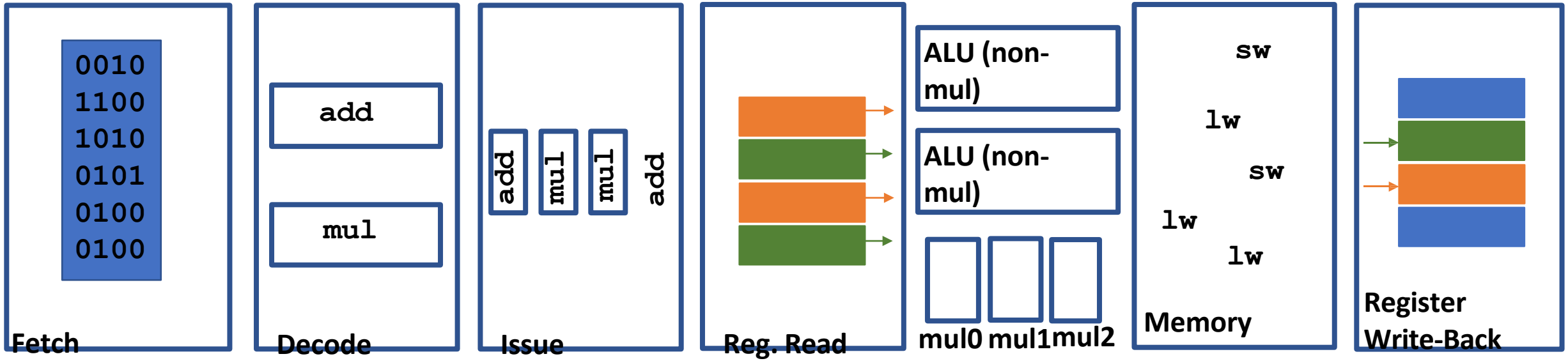
Fifth idea: Add ***multiple execute units*** to which to dispatch operations after they read their input registers



Fourth idea: Decouple ***register read*** from decode. Register read happens for ***issued*** instructions now

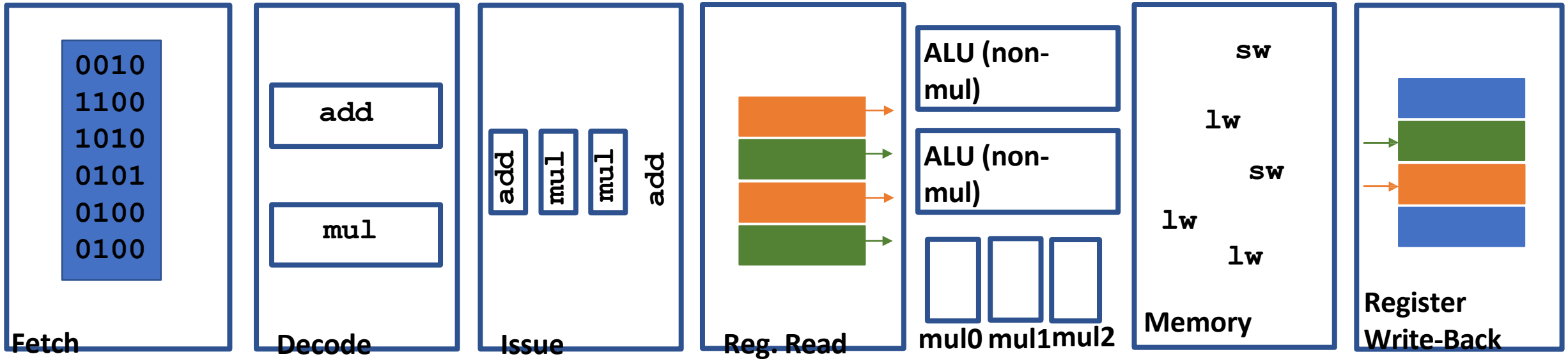
Superscalar processors

Seventh idea: Add ***multiple write ports*** to register file to allow simultaneous multiple register writebacks



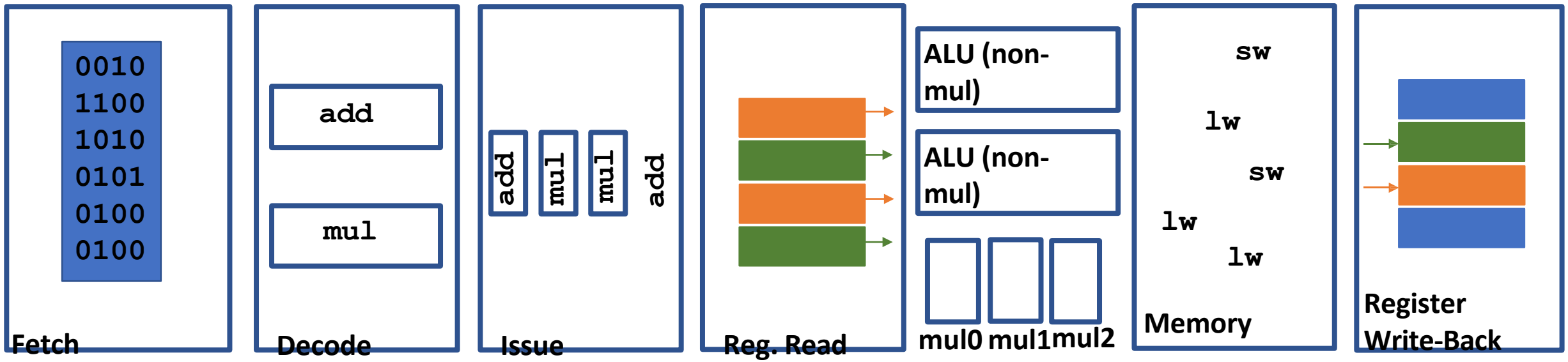
Sixth idea: Handle multiple outstanding memory operations in memory system (complex! we will mostly ignore this part)

Superscalar processors: Challenges & sources of complexity



Fetch:

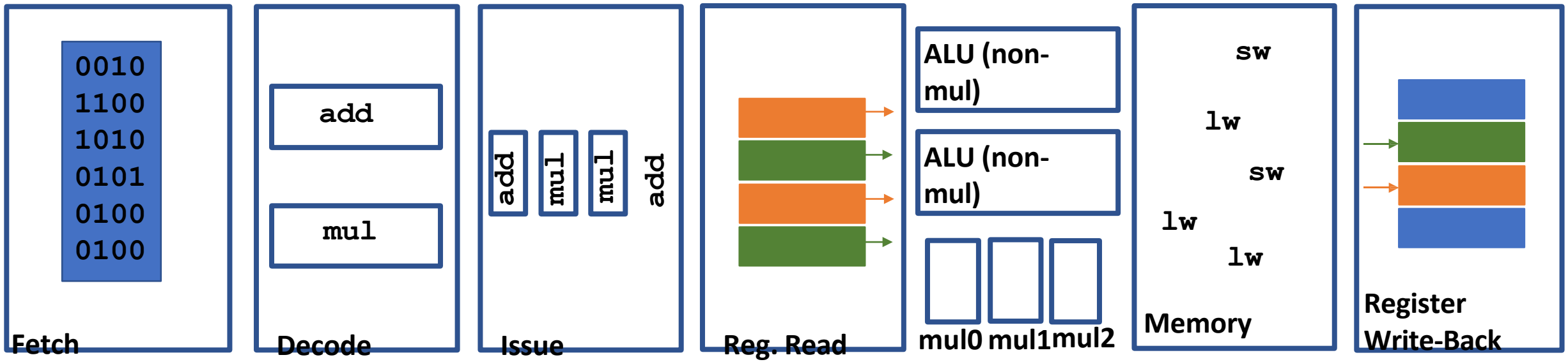
Superscalar processors: Challenges & sources of complexity



Fetch: Branch prediction more complex. Risk of *overfetch* because we're fetching a whole block? Must consider multiple, sequential fetches based on predictions

Superscalar processors: Challenges & sources of complexity

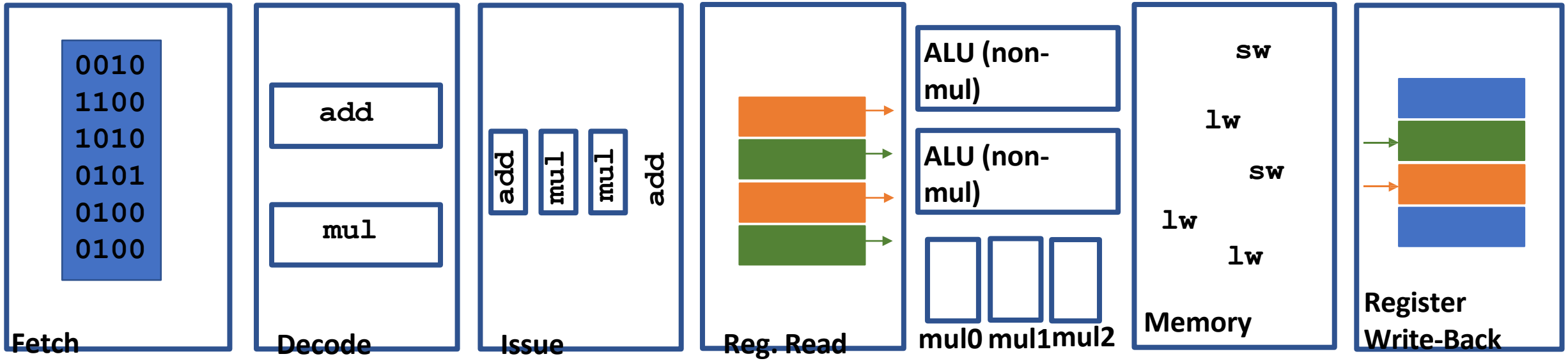
Decode:



Fetch: Branch prediction more complex. Risk of *overfetch* because we're fetching a whole block? Must consider multiple, sequential fetches based on predictions

Superscalar processors: Challenges & sources of complexity

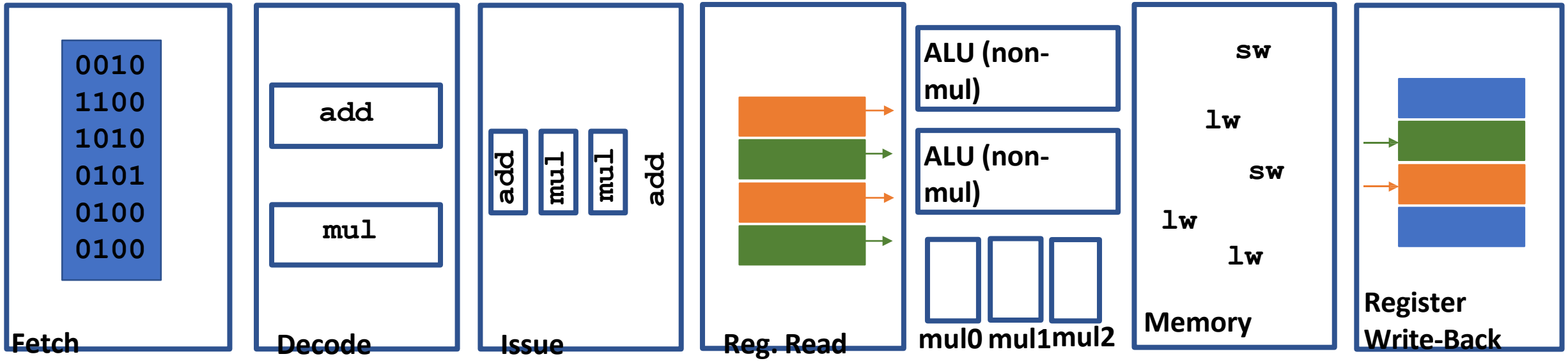
Decode: Not too bad, just replication of resources



Fetch: Branch prediction more complex. Risk of *overfetch* because we're fetching a whole block? Must consider multiple, sequential fetches based on predictions

Superscalar processors: Challenges & sources of complexity

Decode: Not too bad, just replication of resources

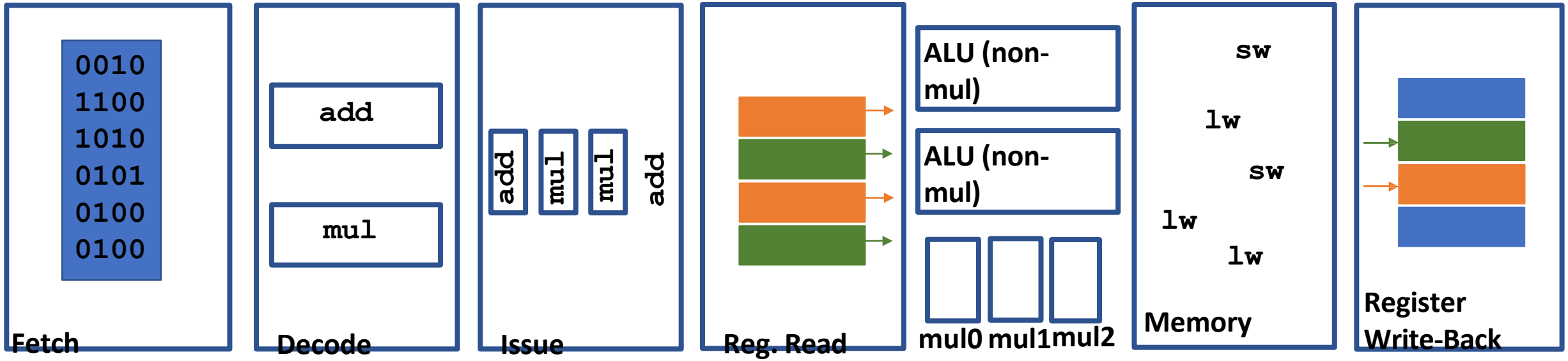


Fetch: Branch prediction more complex. Risk of *overfetch* because we're fetching a whole block? Must consider multiple, sequential fetches based on predictions

Issue:

Superscalar processors: Challenges & sources of complexity

Decode: Not too bad, just replication of resources



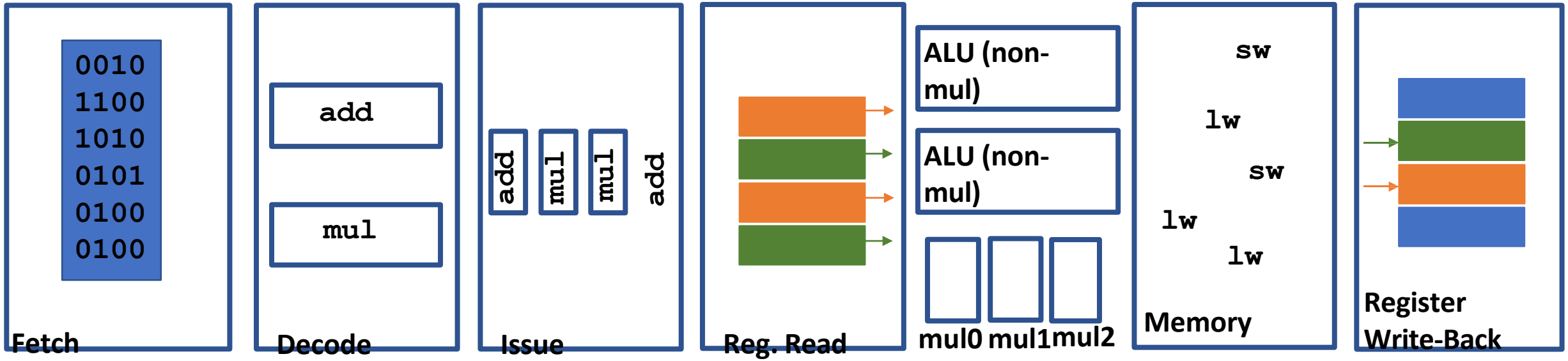
Fetch: Branch prediction more complex. Risk of *overfetch* because we're fetching a whole block? Must consider multiple, sequential fetches based on predictions

Issue: Dependence / hazard detection logic complexity. Need to detect dependences between all instructions in issue queue and some combinations of instructions cannot issue simultaneously

Superscalar processors: Challenges & sources of complexity

Decode: Not too bad, just replication of resources

Reg Read: Multi-porting register file has high cost (4-wide = 8 read ports) & area cost is proportional to *square* of port count



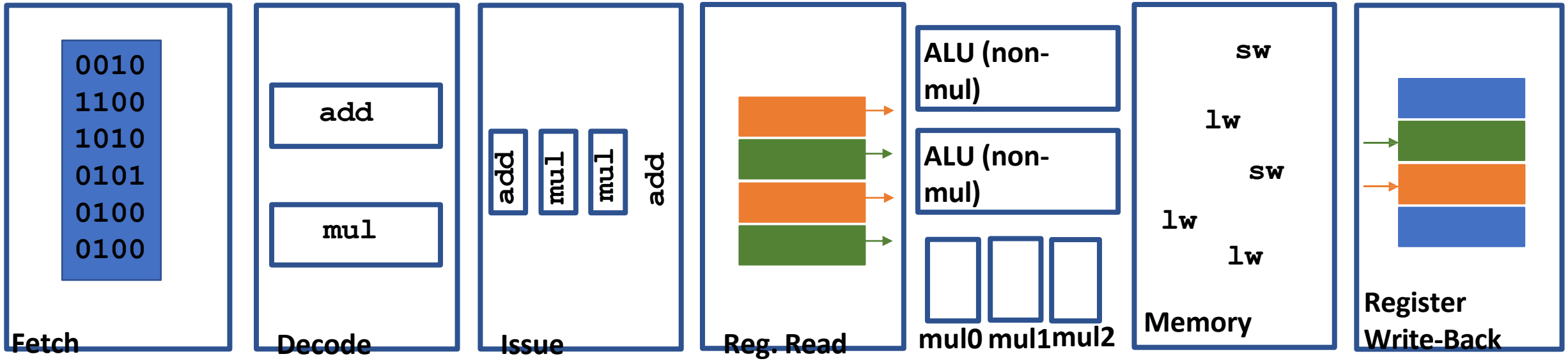
Fetch: Branch prediction more complex. Risk of *overfetch* because we're fetching a whole block? Must consider multiple, sequential fetches based on predictions

Issue: Dependence / hazard detection logic complexity. Need to detect dependences between all instructions in issue queue and some combinations of instructions cannot issue simultaneously

Superscalar processors: Challenges & sources of complexity

Decode: Not too bad, just replication of resources

Reg Read: Multi-porting register file has high cost (4-wide = 8 read ports) & area cost is proportional to *square* of port count



Fetch: Branch prediction more complex. Risk of *overfetch* because we're fetching a whole block? Must consider multiple, sequential fetches based on predictions

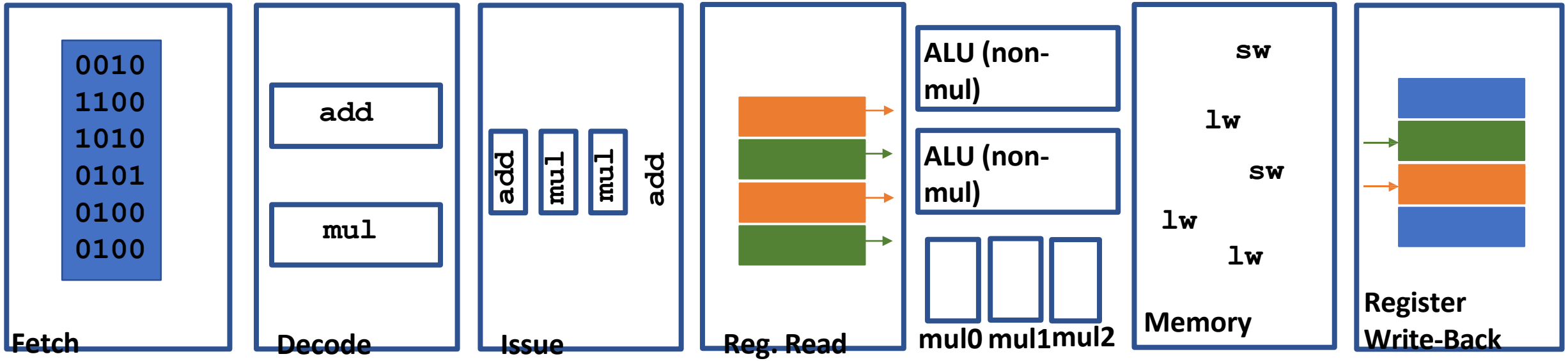
Issue: Dependence / hazard detection logic complexity. Need to detect dependences between all instructions in issue queue and some combinations of instructions cannot issue simultaneously

Execute / Memory:

Superscalar processors: Challenges & sources of complexity

Decode: Not too bad, just replication of resources

Reg Read: Multi-porting register file has high cost (4-wide = 8 read ports) & area cost is proportional to *square* of port count



Fetch: Branch prediction more complex. Risk of *overfetch* because we're fetching a whole block? Must consider multiple, sequential fetches based on predictions

Issue: Dependence / hazard detection logic complexity. Need to detect dependences between all instructions in issue queue and some combinations of instructions cannot issue simultaneously

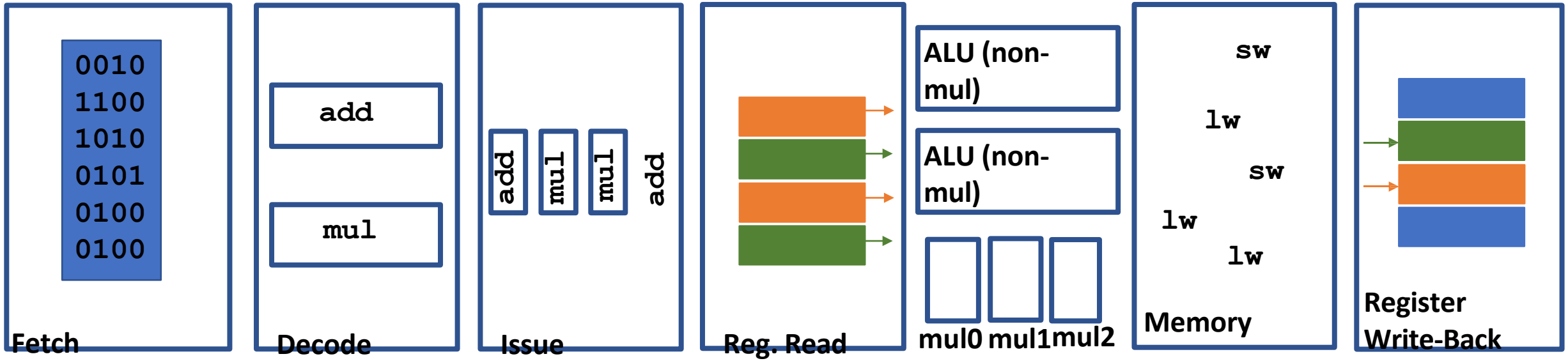
Execute / Memory: More execute units, more cache ports. Forwarding paths & input operand selection logic become very complicated.

Superscalar processors: Challenges & sources of complexity

Decode: Not too bad, just replication of resources

Reg Read: Multi-porting register file has high cost (4-wide = 8 read ports) & area cost is proportional to *square* of port count

Reg. WB: Write port per instruction that may complete that writes a register (4-wide = 4 write ports)

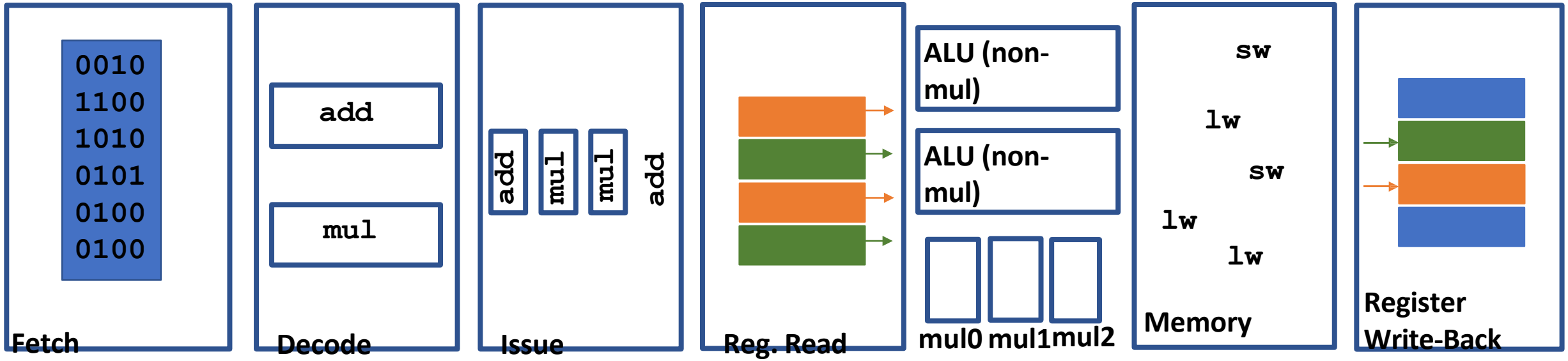


Fetch: Branch prediction more complex. Risk of *overfetch* because we're fetching a whole block? Must consider multiple, sequential fetches based on predictions

Issue: Dependence / hazard detection logic complexity. Need to detect dependences between all instructions in issue queue and some combinations of instructions cannot issue simultaneously

Execute / Memory: More execute units, more cache ports. Forwarding paths & input operand selection logic become very complicated.

Remaining limits on performance of this processor?

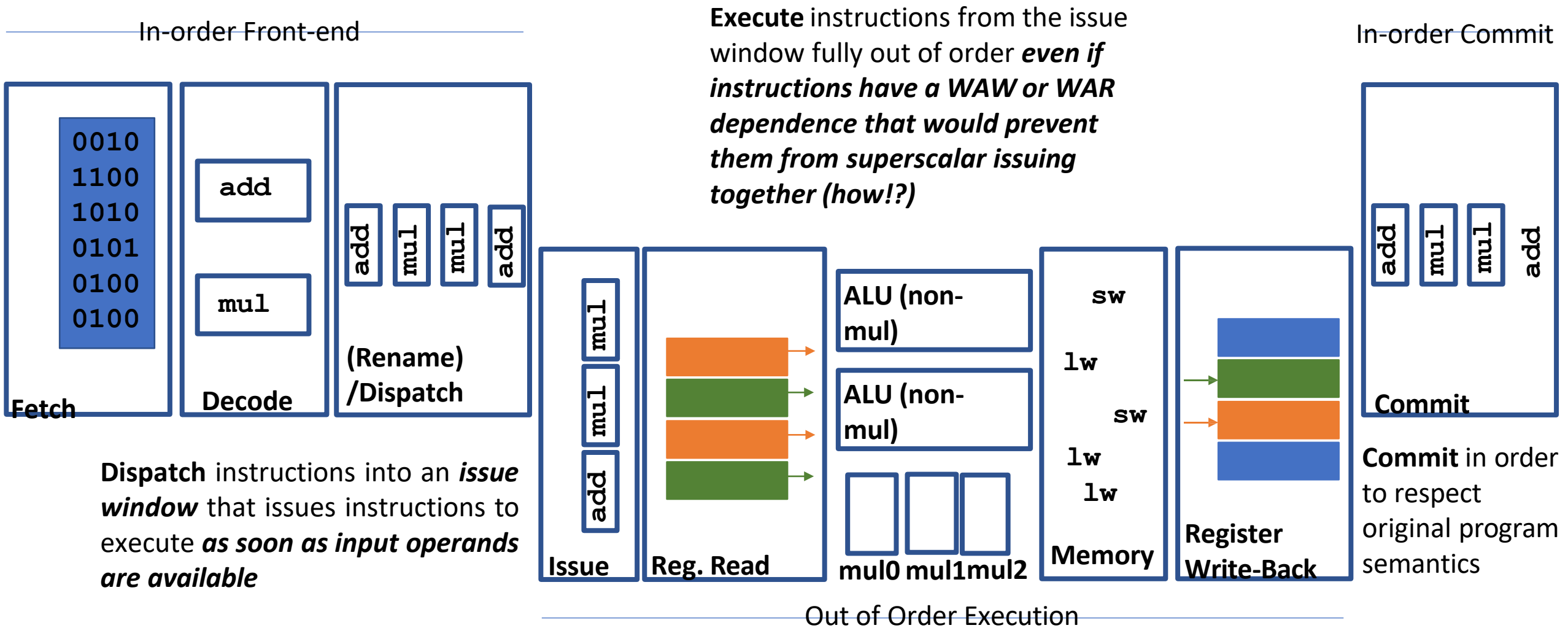


Application itself may not have ample ILP

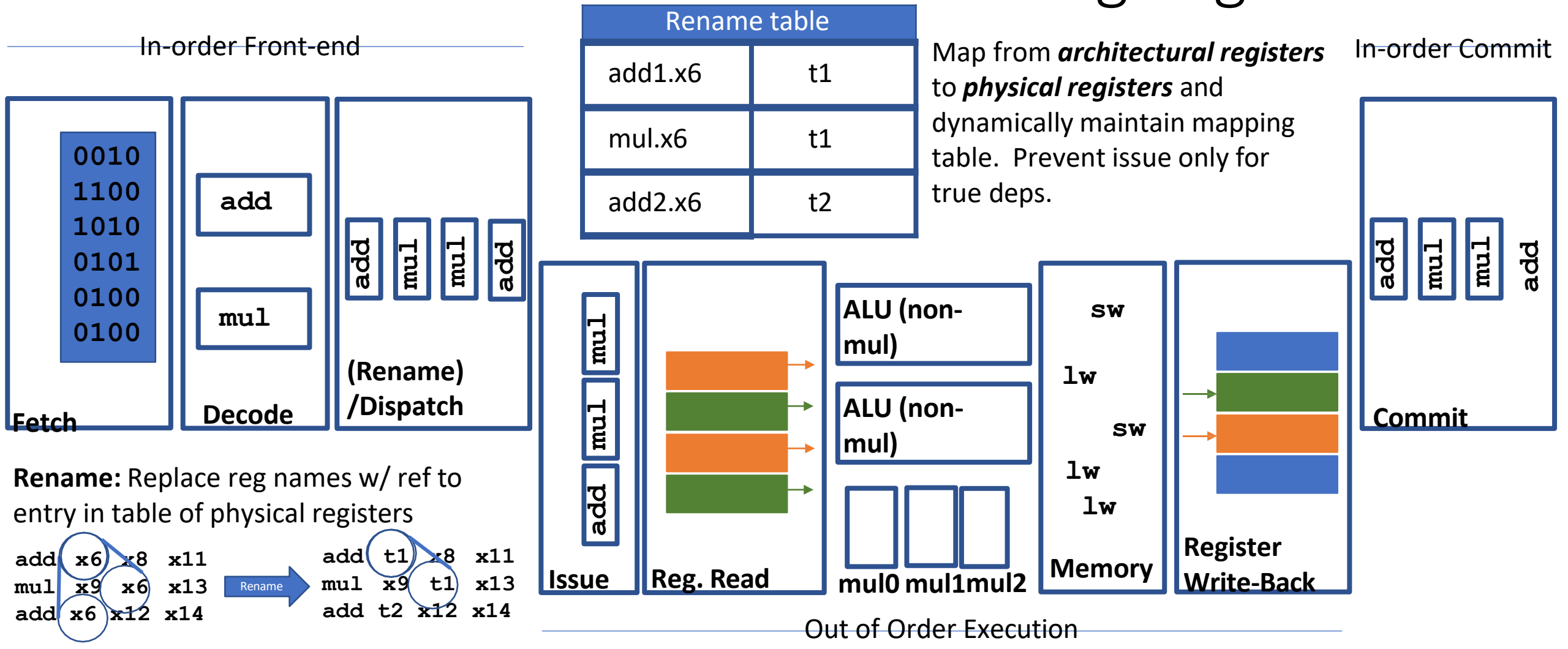
```
add x6  x8  x11
add x12 x6  x13
mul x7  x9  x14
```

In-order issue rule:
"Unlucky" sequence of instructions may prevent multiple issue. (e.g., the first add and the mul can issue together, but the second add prevents it.)

Out of Order Execution



Register Renaming Resolves Dependences that Prevent Instructions from Executing Together

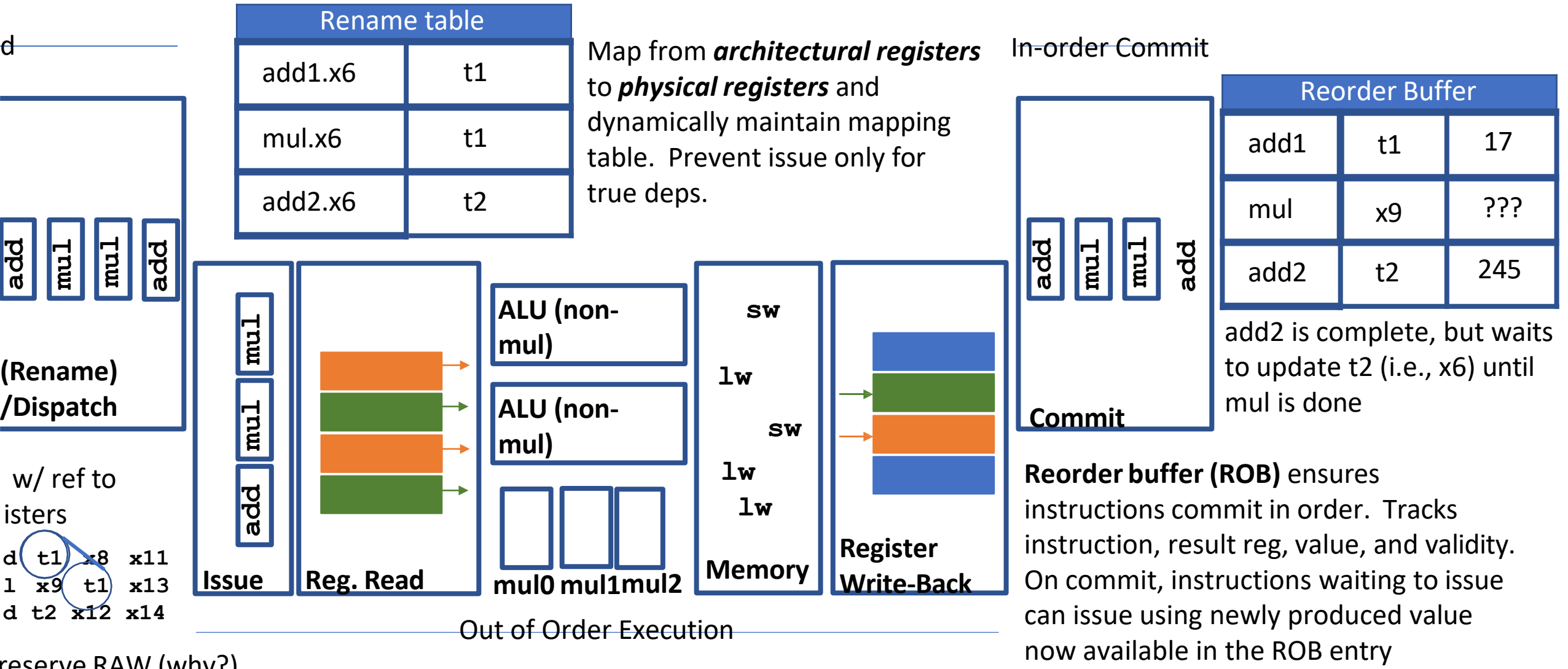


Rename: Replace reg names w/ ref to entry in table of physical registers

```
add x6  x8  x11          add t1  x8  x11
mul x9  x6  x13   Rename  mul x9  t1  x13
add x6  x12 x14          add t2  x12 x14
```

Eliminate WAW, WAR, and preserve RAW (why?)

In-order commit tracks instruction completion and ensures architectural state updates in order



All Types of Data Hazards Matter in OoO Execution

```
sub x6 x5 x4
lw  x16 0xabc
add x12 x6 x14
```

Read-After-Write (RAW)

```
sub x8 x16 x4
add x16 x6 x14
lw  x16 0xabc
```

Write-After-Read (WAR)

```
lw  x6 0xabc
sub x6 x5 x4
add x12 x6 x14
```

Write-After-Write (WAW)

Only Read-After-Write (RAW) hazards are possible in our simple pipeline

Types of Data Hazards

```
lw   x6 0xabc
sub  x6 x5 x4
add  x12 x6 x14
```

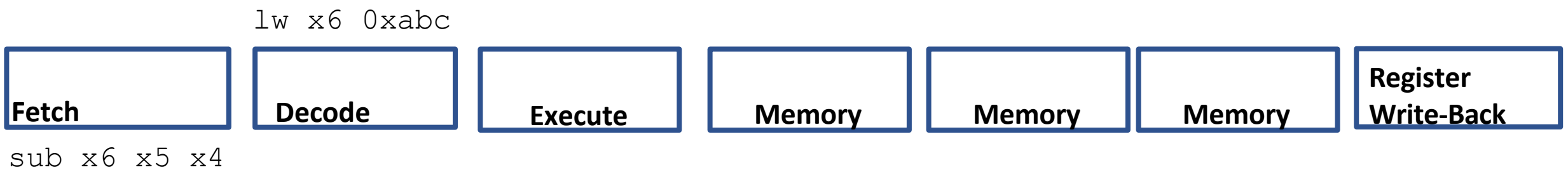
Write-After-Write (WAW)

`lw x6 0xabc`

Types of Data Hazards

```
lw   x6 0xabc
sub  x6 x5 x4
add  x12 x6 x14
```

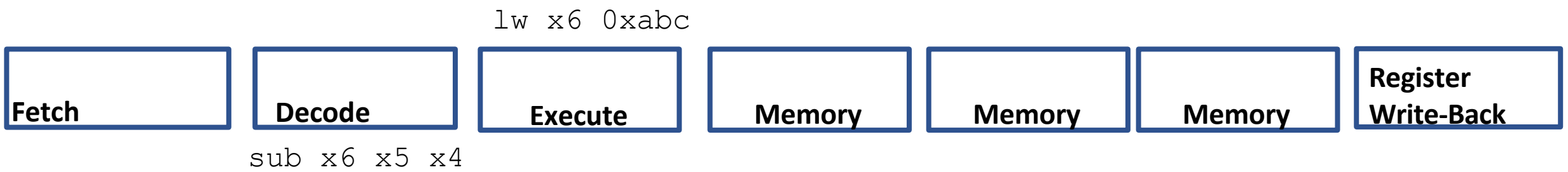
Write-After-Write (WAW)



Types of Data Hazards

```
lw   x6 0xabc
sub  x6 x5 x4
add  x12 x6 x14
```

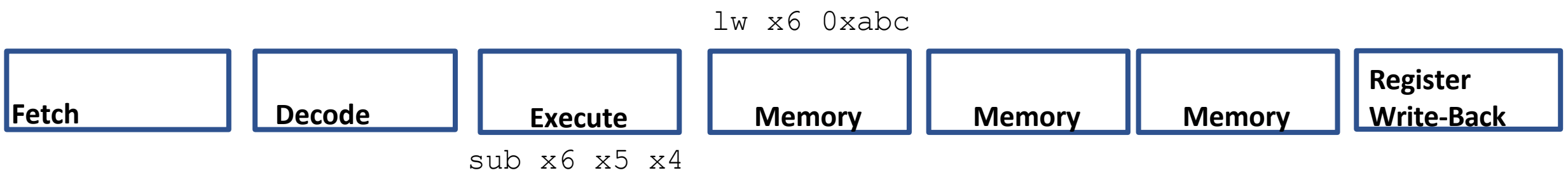
Write-After-Write (WAW)



Types of Data Hazards

```
lw   x6 0xabc
sub  x6 x5 x4
add  x12 x6 x14
```

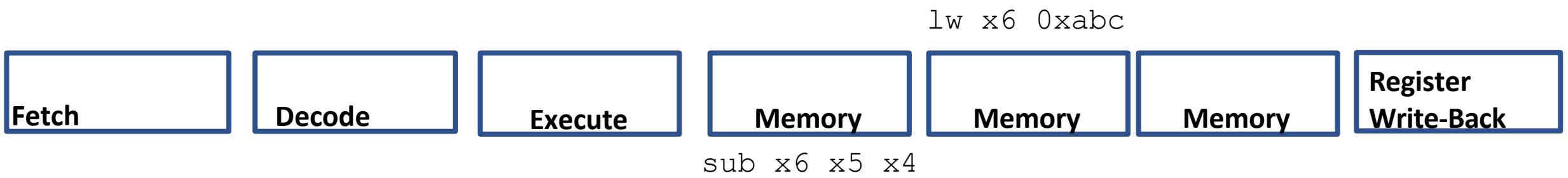
Write-After-Write (WAW)



Types of Data Hazards

```
lw   x6 0xabc
sub  x6 x5 x4
add  x12 x6 x14
```

Write-After-Write (WAW)

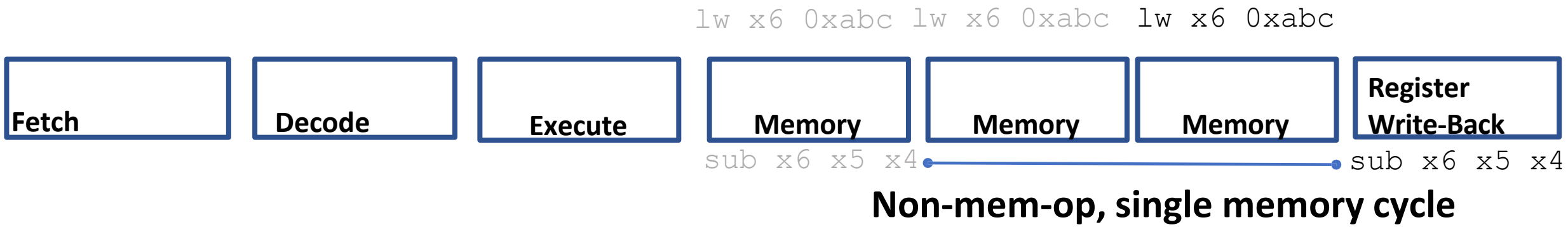


Types of Data Hazards

```
lw   x6 0xabc
sub  x6 x5 x4
add  x12 x6 x14
```

Write-After-Write (WAW)

Multi-cycle latency memory op



Non-mem-op, single memory cycle

Earlier `lw` instruction finishes after later `sub` instruction. Both write `x6`. Wrong final value in `x6`.

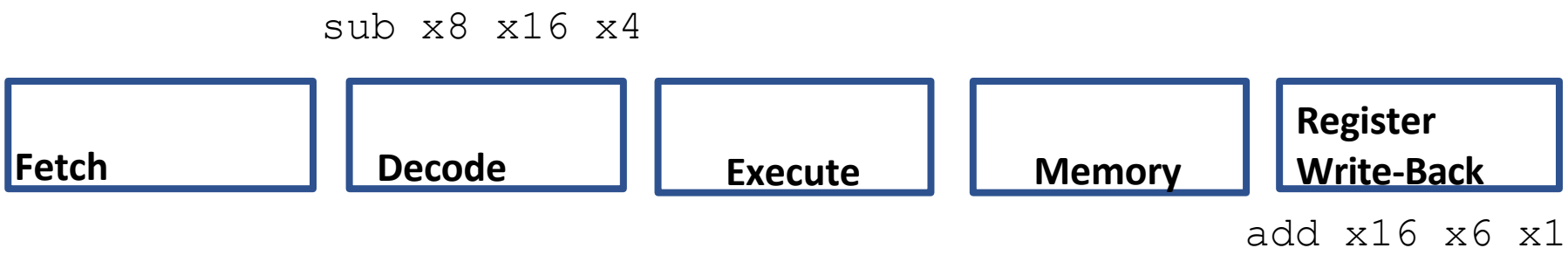
Explicitly handled with logic to maintain ordering in processors that allow this behavior (not our datapath)

Types of Data Hazards

```
sub  x8  x16  x4
add  x16 x6   x14
lw   x11 0xabc
```

Write-After-Read (WAR)

Stalled at decode/reg. read

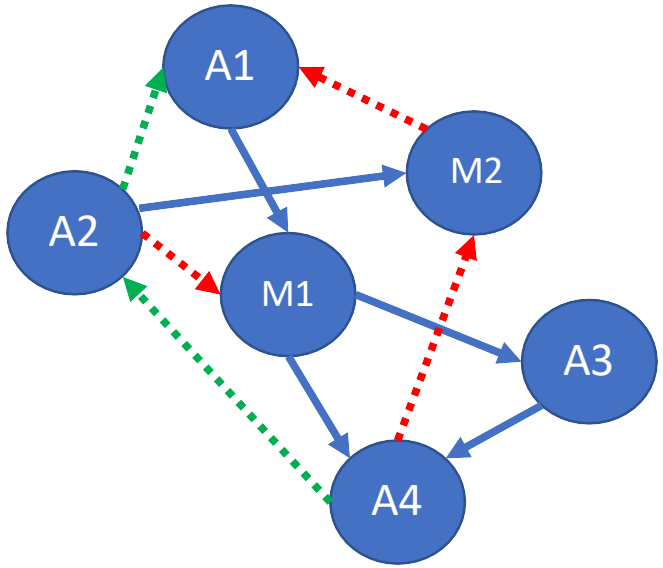


Completes quickly and writes reg.

Later `add` instruction writes `x16` before earlier `sub` instruction reads `x16`. `sub` sees wrong value!

Renaming Example

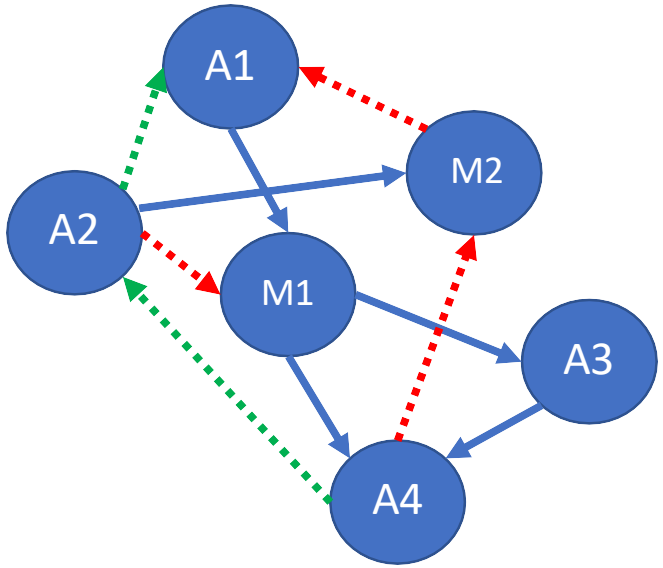
```
A1: add x6 x8 x11
M1: mul x9 x6 x13
A2: add x6 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```



Question: How can instructions issue to our out-of-order pipeline in which instructions may execute and complete out of order?
If WAW or WAR, can't just dispatch or OoO execution may read regs not yet updated

Renaming Example

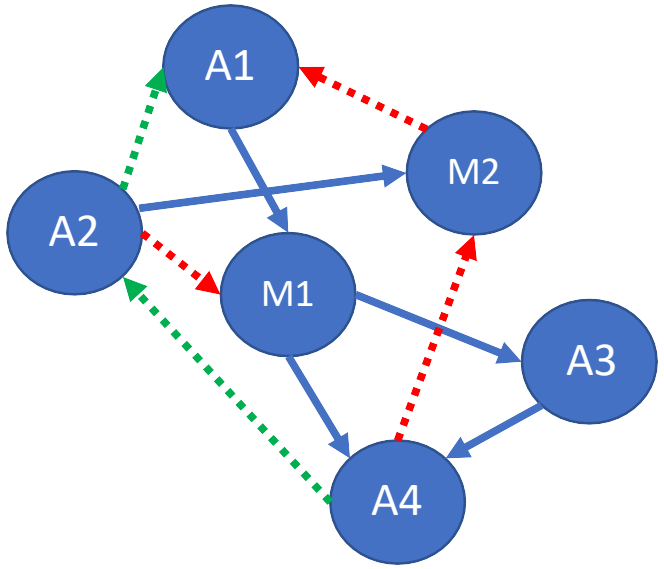
```
A1: add x6 x8 x11
M1: mul x9 x6 x13
A2: add x6 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```




```
Rename Table
A1.x6 -> r0
```

Renaming Example

```
A1: add x6 x8 x11
M1: mul x9 x6 x13
A2: add x6 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```

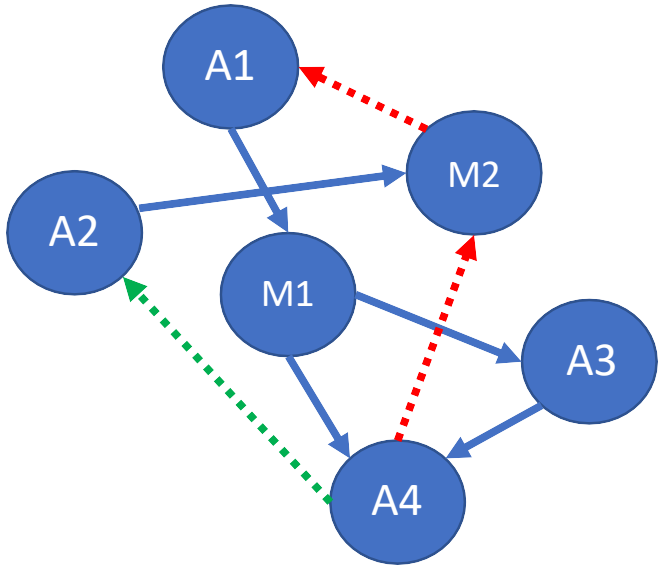



```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
```

RAW dependence on x6
M1 waiting on result from A1 (r0)

Renaming Example

```
A1: add x6 x8 x11
M1: mul x9 x6 x13
A2: add x6 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```



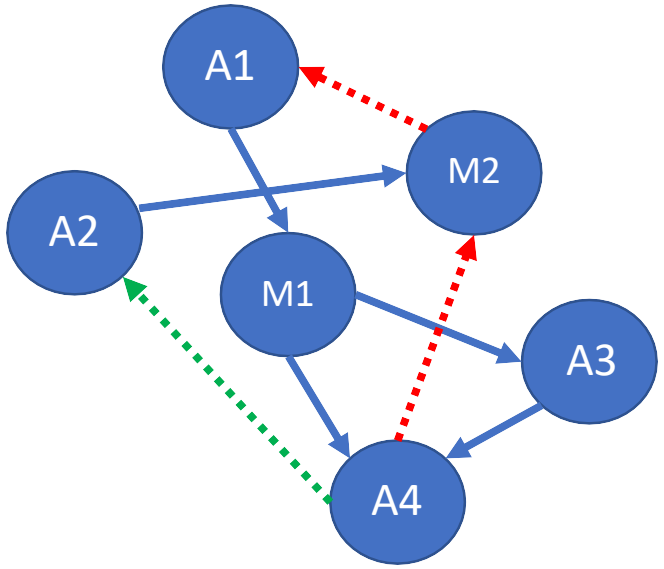

```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
A2.x6 -> r2
```

WAW dep b/w A1 & A2 & WAR dep w/ M1

Resolved by renaming output regs

Renaming Example

```
A1: add x6 x8 x11
M1: mul x9 x6 x13
A2: add x6 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```



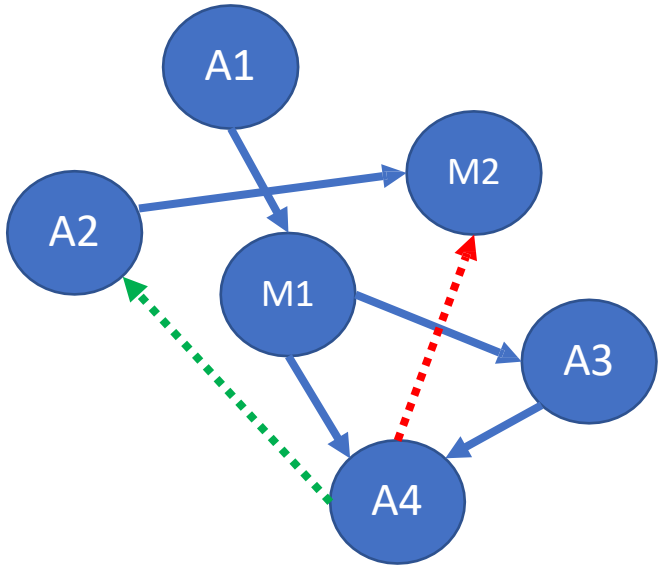

```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
A2.x6 -> r2
A3.x7 -> r3
A3.x9 <- r1
M2.x8 -> r4
```

RAW dependence between M1 & A3

Cannot be resolved by renaming

Renaming Example

```
A1: add x6 x8 x11
M1: mul x9 x6 x13
A2: add x6 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```



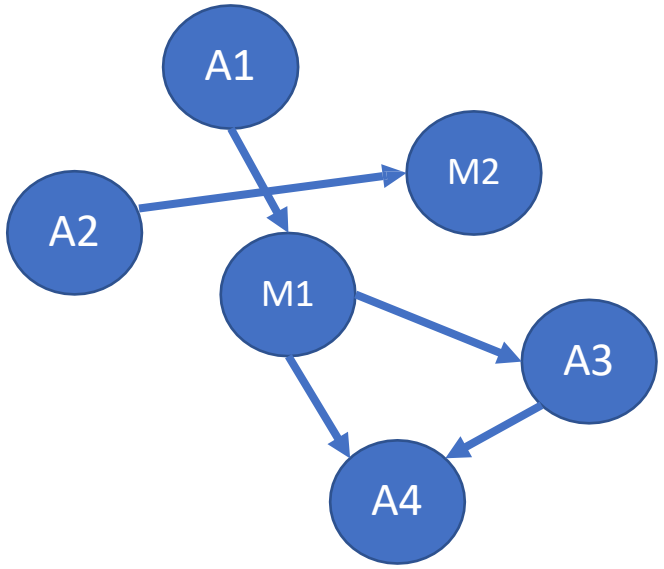

```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
A2.x6 -> r2
A3.x7 -> r3
A3.x9 <- r1
M2.x8 -> r4
M2.x6 <- r2
```

WAW dep w/ A1 resolved by renaming

True dep w/ A2 resolved by looking up renamed result of A2

Renaming Example

```
A1: add x6 x8 x11
M1: mul x9 x6 x13
A2: add x6 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```



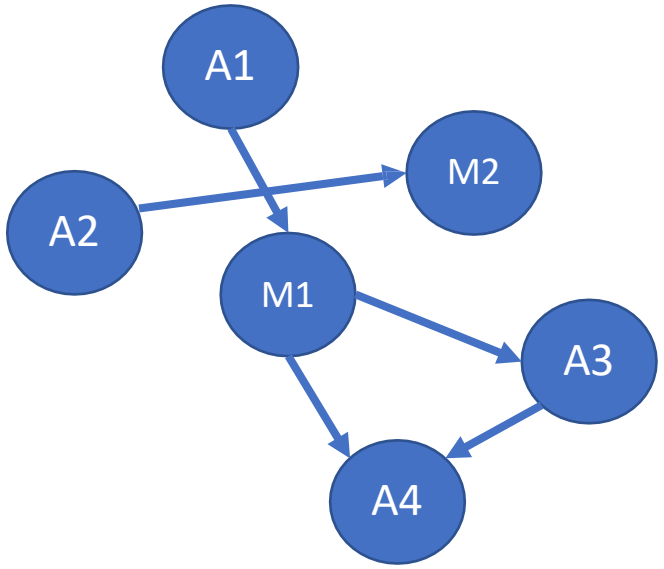

```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
A2.x6 -> r2
A3.x7 -> r3
A3.x9 <- r1
M2.x8 -> r4
M2.x6 <- r2
A4.x6 -> r5
A4.x7 <- r3
A4.x9 <- r1
```

WAR dep with M2 & WAW w/ A2 resolved by renaming

True deps w/ A3 and M1 resolved by looking up renamed regs in table

Renaming Example

```
A1: add x6 x8 x11
M1: mul x9 x6 x13
A2: add x6 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```

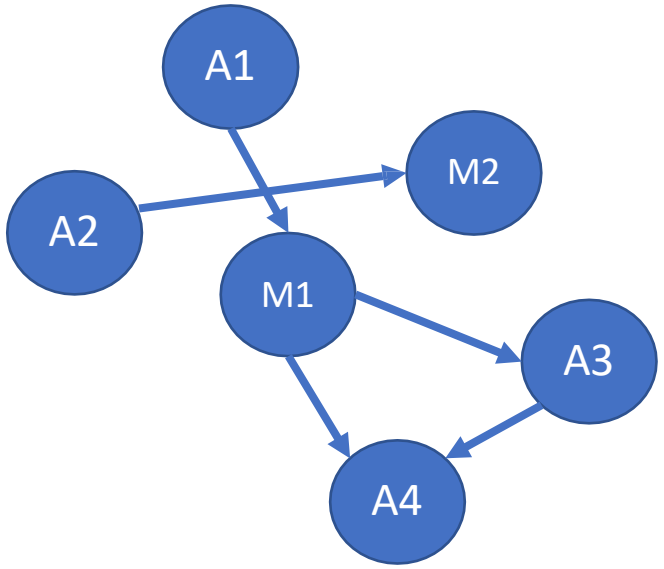



```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
A2.x6 -> r2
A3.x7 -> r3
A3.x9 <- r1
M2.x8 -> r4
M2.x6 <- r2
A4.x6 -> r5
A4.x7 <- r3
A4.x9 <- r1
```

After register renaming, only RAW dependences (i.e., “True Dependences”) remain in the execution

Renaming Example

```
A1: add r0 x8 x11
M1: mul r1 r0 x13
A2: add r2 x17 x30
A3: add r3 r1 x14
M2: add r4 x18 r2
A4: add r5 r3 r1
```




```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
A2.x6 -> r2
A3.x7 -> r3
A3.x9 <- r1
M2.x8 -> r4
M2.x6 <- r2
A4.x6 -> r5
A4.x7 <- r3
A4.x9 <- r1
```

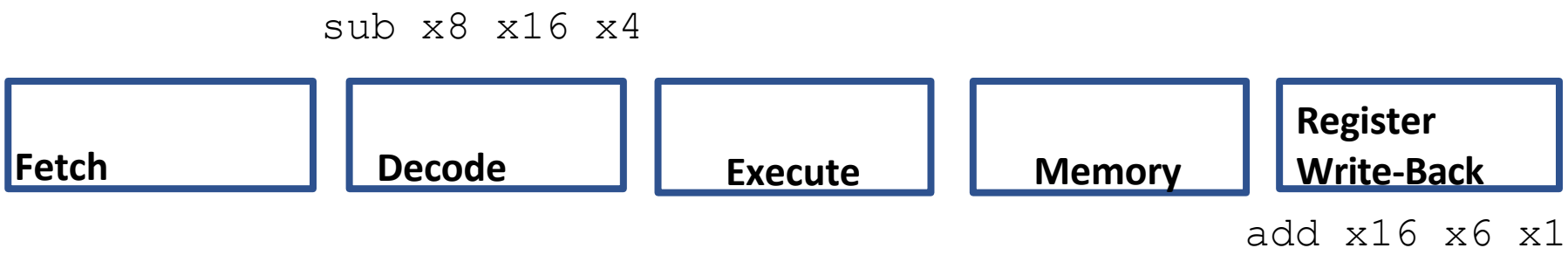
After register renaming, only RAW dependences (i.e., “True Dependences”) remain in the execution

Renaming Avoids False Deps

```
sub  x8  x16  x4
add  r1  x6   x14
lw   x11 0xabc
```

Write-After-Read (WAR)

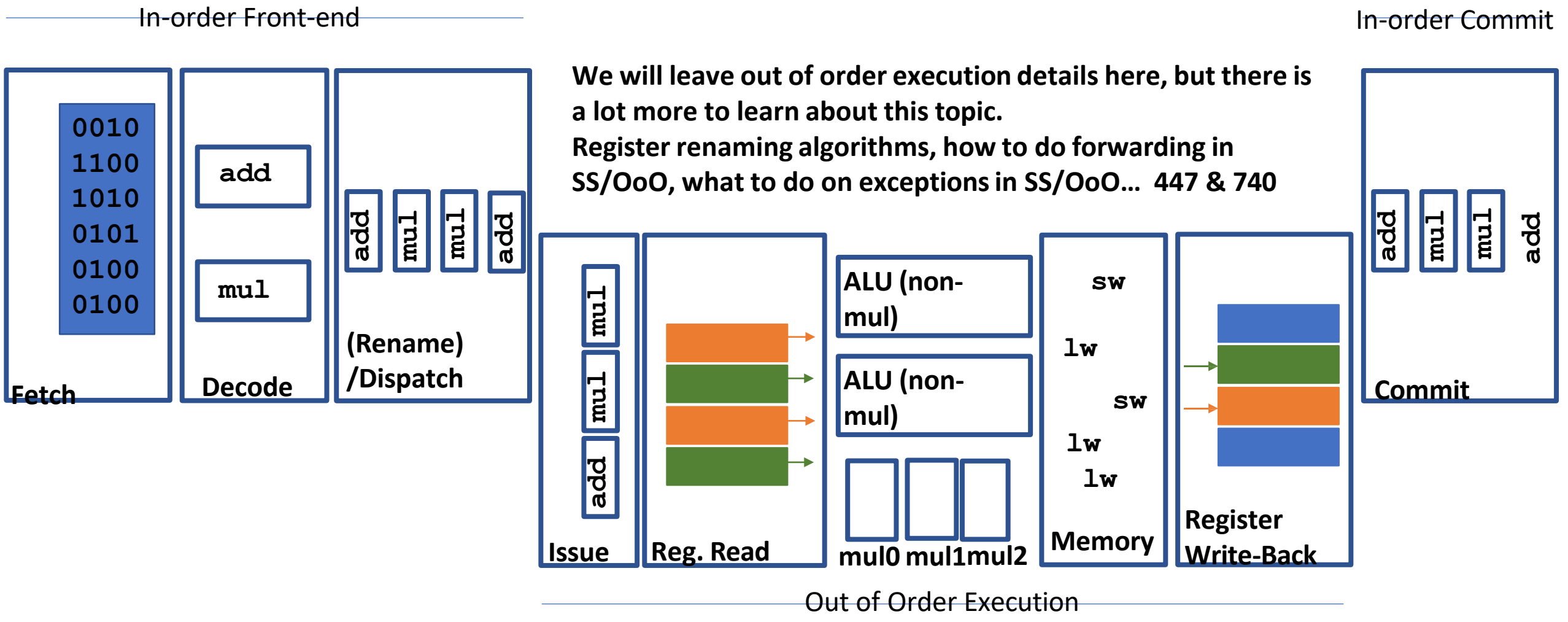
Stalled at decode/reg. read



Completes quickly and writes reg.

Later `add` instruction writes `r1` before earlier `sub` instruction reads `x16`, which is perfectly ok!

Superscalar Out of Order Execution is *extremely complex to implement*



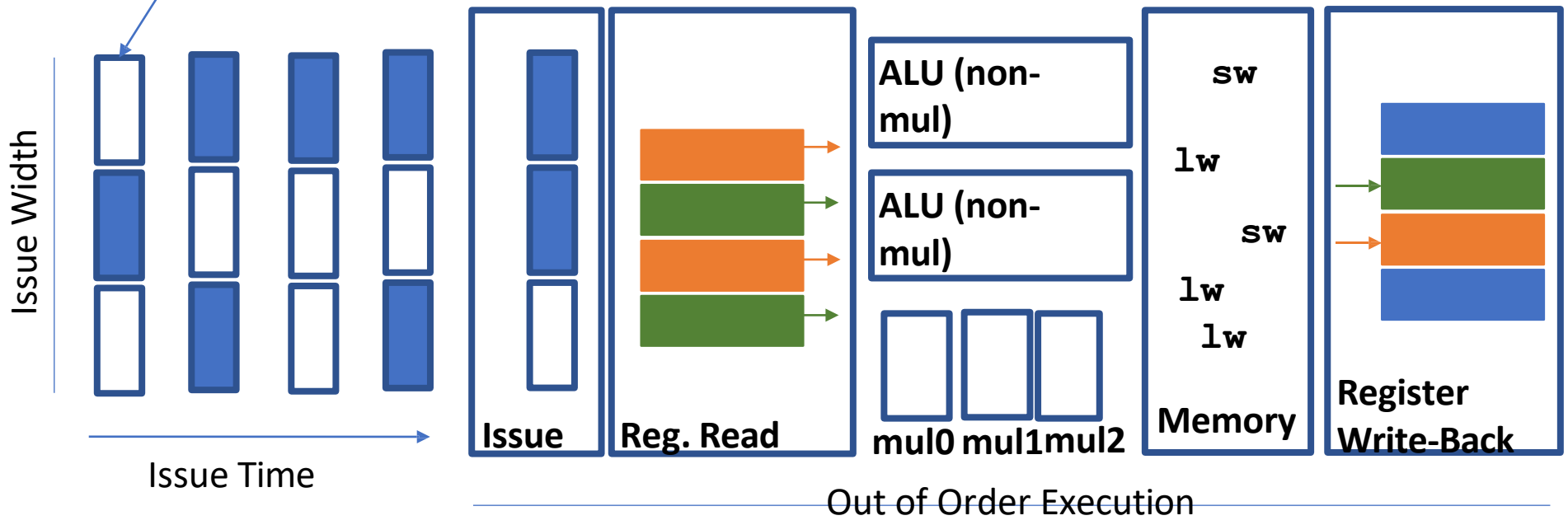
Scheduling Techniques to Maximize ILP

Superscalar execution exploits ILP to increase IPC

Empty issue slot represent wasted opportunity to do some work on a cycle

Performance in a superscalar processor depends on the existence of ILP in the program.

We need there to be parallelizable instructions in the instruction stream that we fetch, dispatch, and issue.
Question: how to avoid issue slot waste?

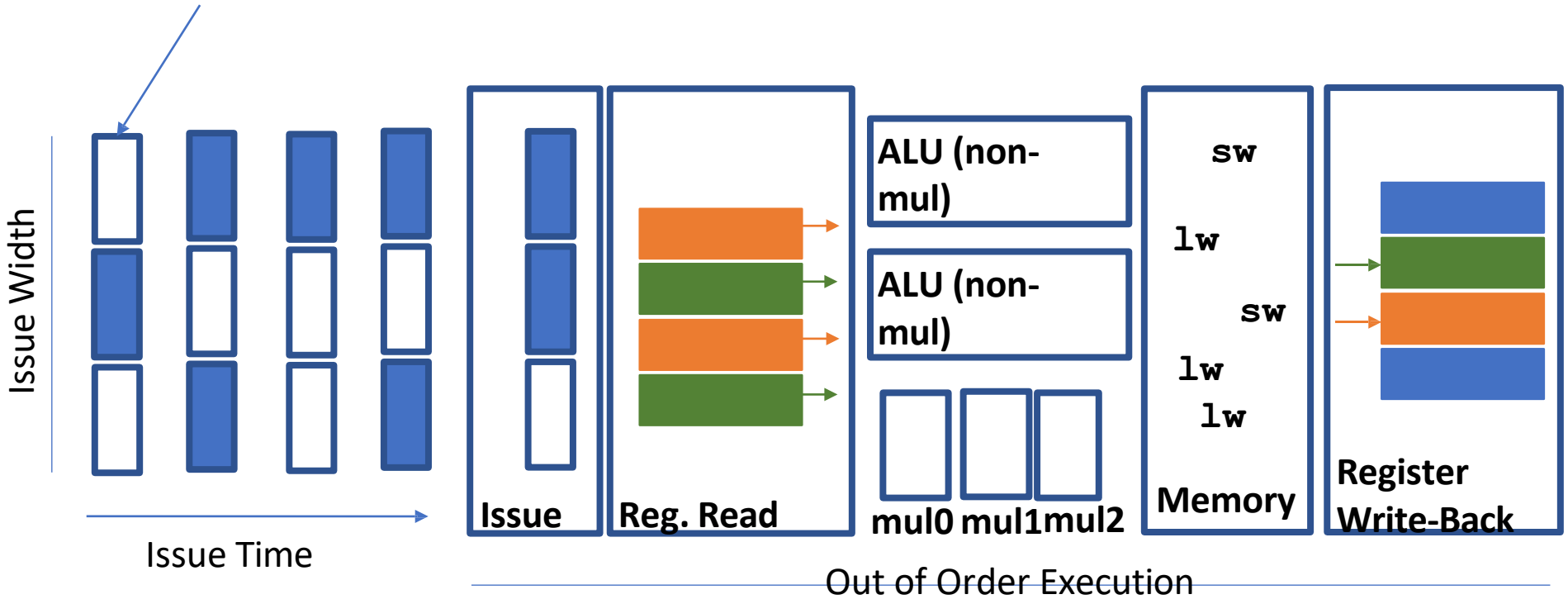


Superscalar execution exploits ILP to increase IPC

Question: how to avoid issue slot waste?

- *Schedule code in program to avoid dependences*
- *Schedule code in loops to align with fetch granularity*
- *Schedule code to avoid oversubscribing functional units (i.e., a sequence of consecutive multiplies can't issue together)*

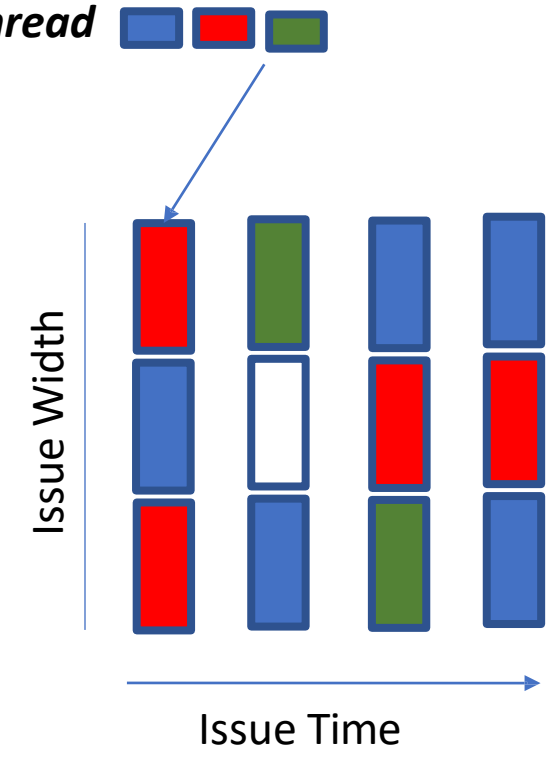
Empty issue slot represent wasted opportunity to do some work on a cycle



Simultaneous Multi-Threading (SMT)

Also known as “Hyper-threading” on Intel processors, used for decades now.

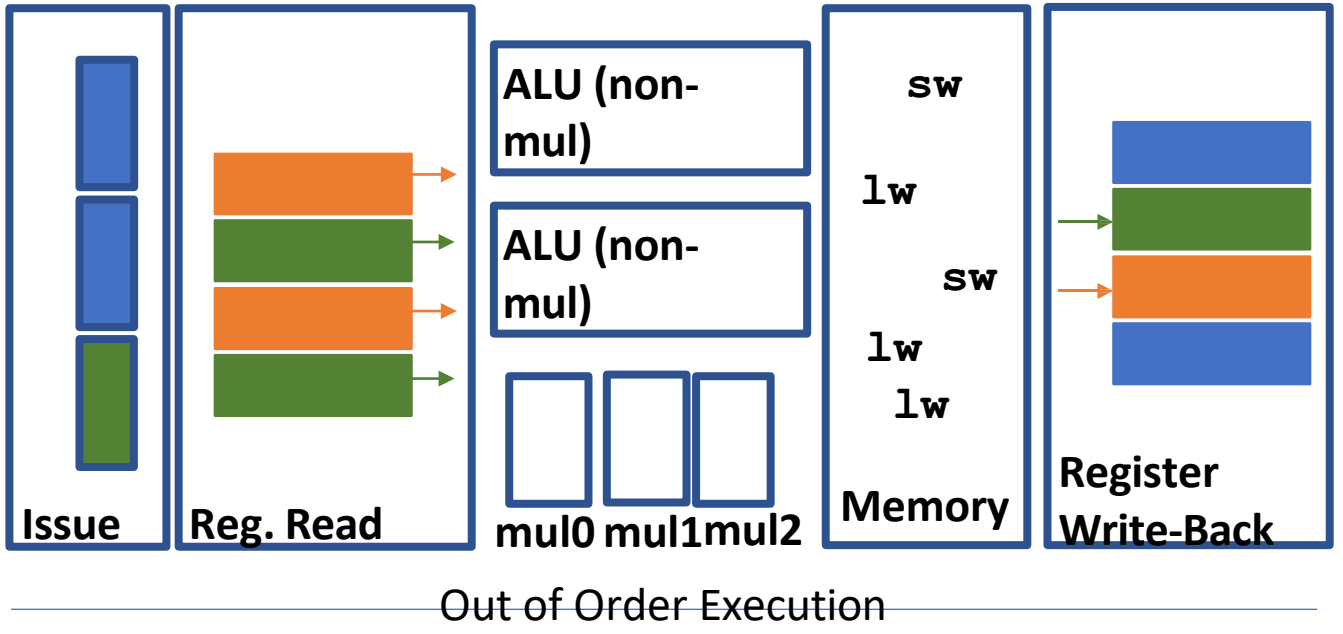
Fill empty issue slots with instructions from another ***thread***



SMT exploits thread-level parallelism (TLP) instead of ILP to increase a machine’s useful IPC.

If a program has multiple threads, issue from each thread.

Question: Sources of hardware complexity for SMT?

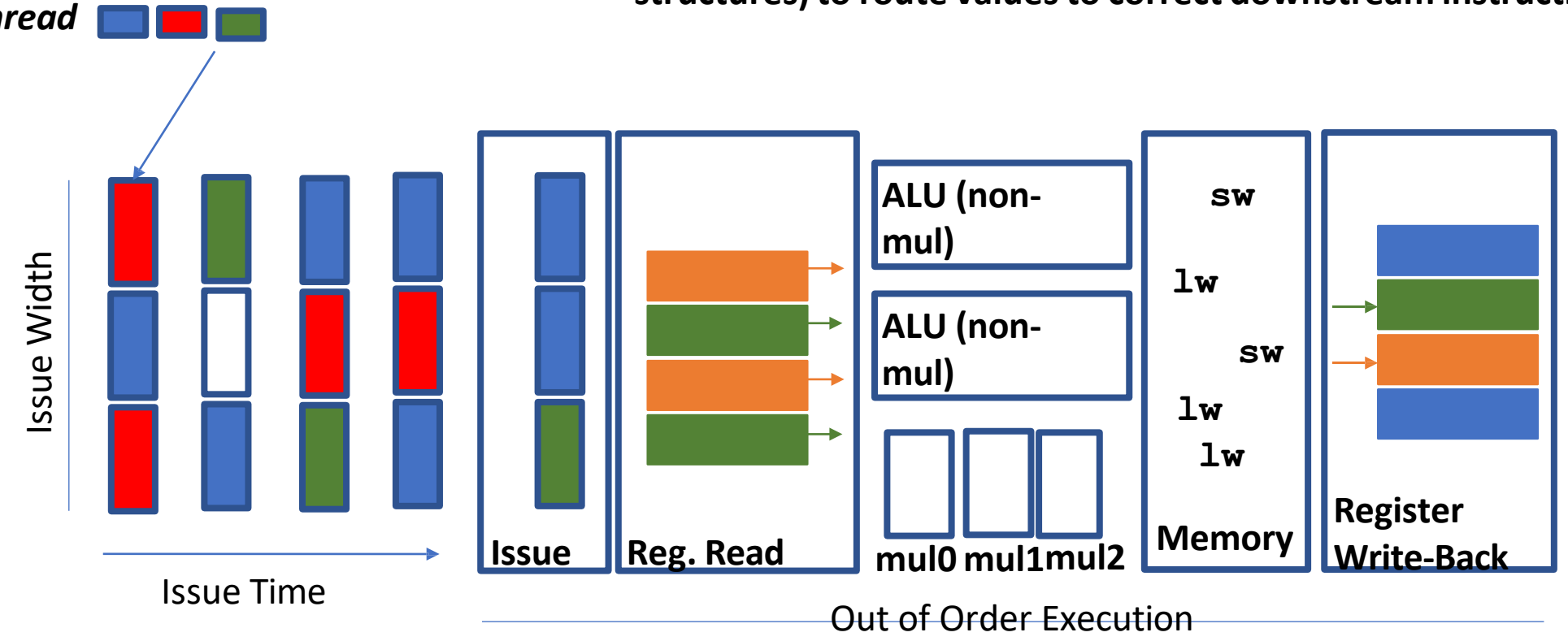


Simultaneous Multi-Threading (SMT)

Fill empty issue slots with instructions from another ***thread***

Question: Sources of hardware complexity for SMT?

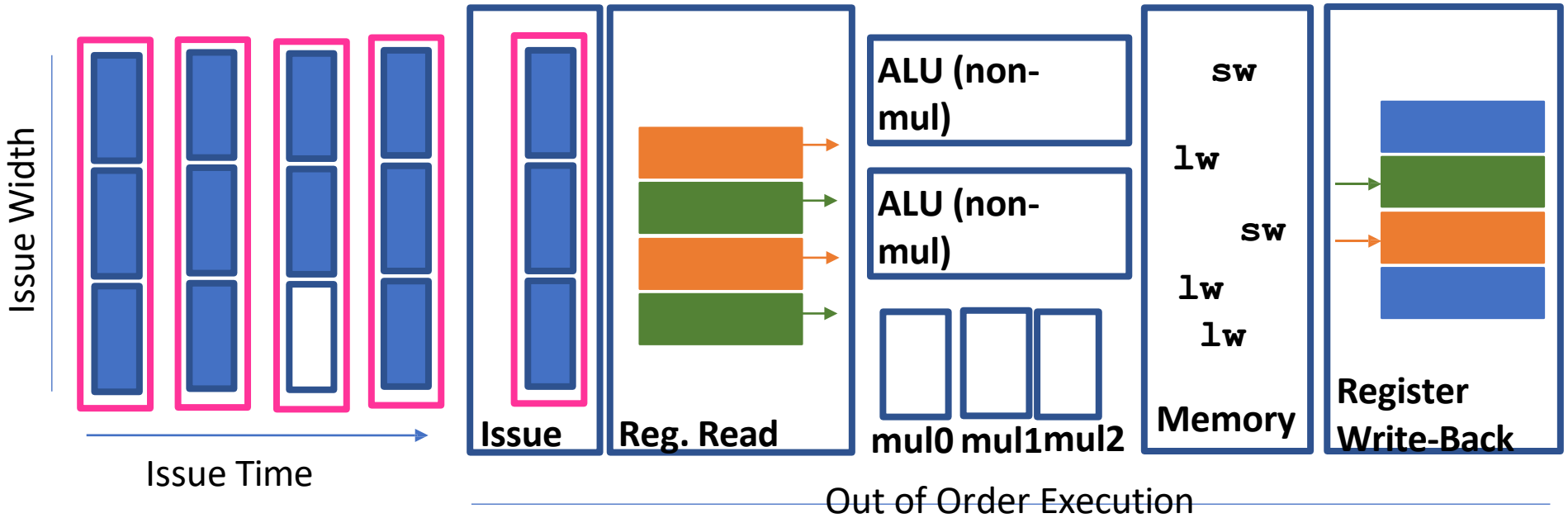
- **Need fetch to support multiple streams (including branch prediction logic…)**
- **Need to tag functional units, rename table entries, ROB entries (and other structures) to route values to correct downstream instructions**



Very Large Instruction Word (VLIW) Architectures

Change the *ISA*! In VLIW, the ISA exposes the issue width architecturally
Each fetch / issue is on a *packet* of instructions, hopefully independent

Intel IA-64 bundles up to 3 instructions with a *type* that says whether & how they're dependent or parallelizable



Type:
Mem, Float,
Int, Long Imm.
Branch
e.g.,
MMI, IIF, MMI
MM/I, M/MI

"/" indicates a "stop", break parallelism.

The compiler plays a crucial role

- We will pick up next time with more discussion of hardware/software interfaces that expose opportunities for parallelism
- We will study how the compiler exposes parallelism and exploits the opportunities for parallelism in the architecture
- More VLIW, Vector architectures
- Then we will look at some compiler fundamentals and see how all of these ideas converge in software