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

Fall 2024, Prof. Brandon Lucia & Prof. Akshitha Sriraman

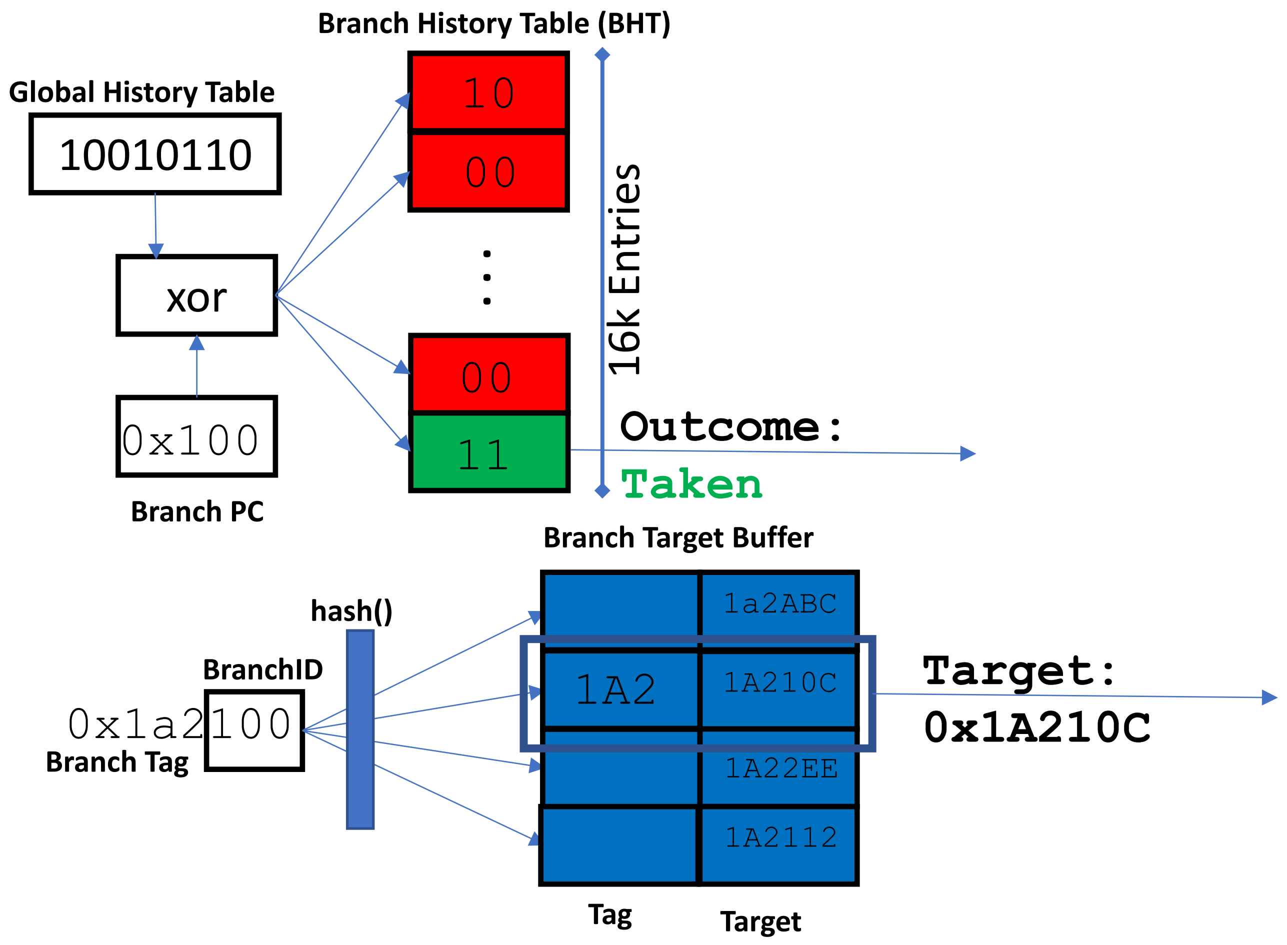
Recap: Design Space Exploration

- Defining the design space of a hardware or software system
- Pareto Frontiers and optimizing within a design space
- **Applied** Performance Evaluation
  - Finding the best performing design under constraints

Defining a **design space**

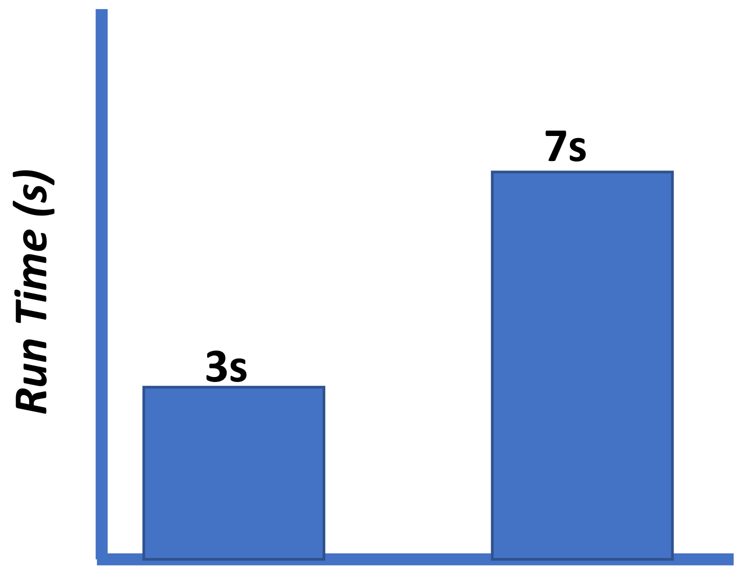
- **A design space is a set of possible incarnations of a system**
- **A design space is defined over a set of parameters**
- **A point in the design space is a concrete system with a concrete value for each of the design space's parameters**
- **Design spaces exist to allow systematic exploration of a collection of possible designs, like architectures.**

Example: Branch Predictor Design Space



$$\begin{bmatrix} 8 \\ 16k \\ \text{xor()} \\ 2 \\ \text{hash()} \\ 4 \\ 1 \end{bmatrix} \qquad \begin{bmatrix} Sg \\ Nb \\ Hp \\ Sb \\ Hb \\ Nt \\ At \end{bmatrix}$$

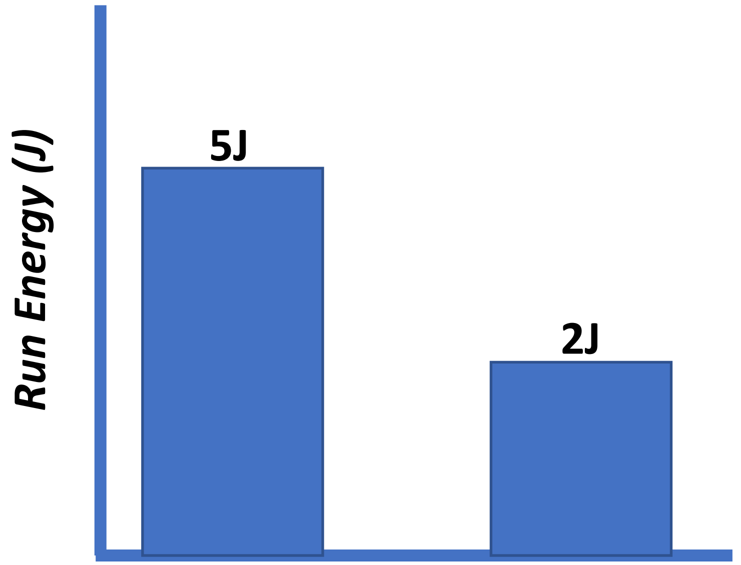
Is one of these **better**?



Faster design!

8
16k
xor()
2
hash()
4
1

16
1k
xor()
2
hash()
8
2



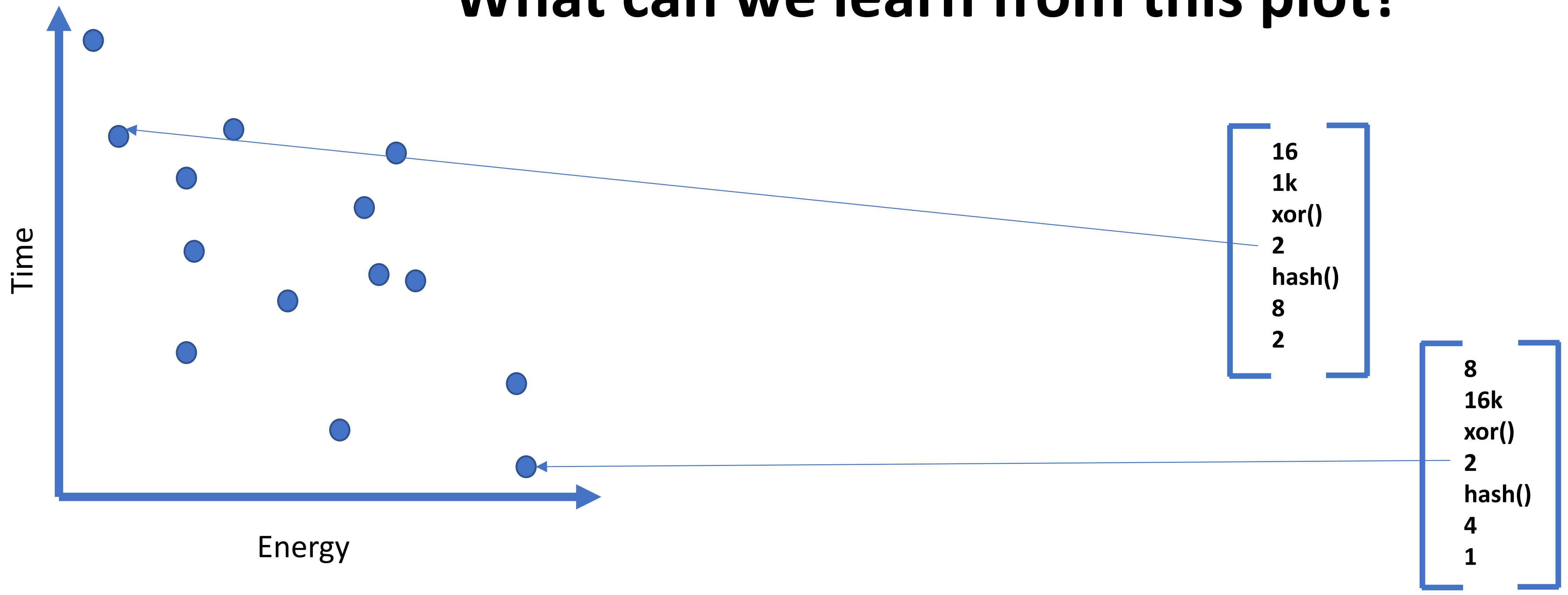
8
16k
xor()
4
1

Lower energy

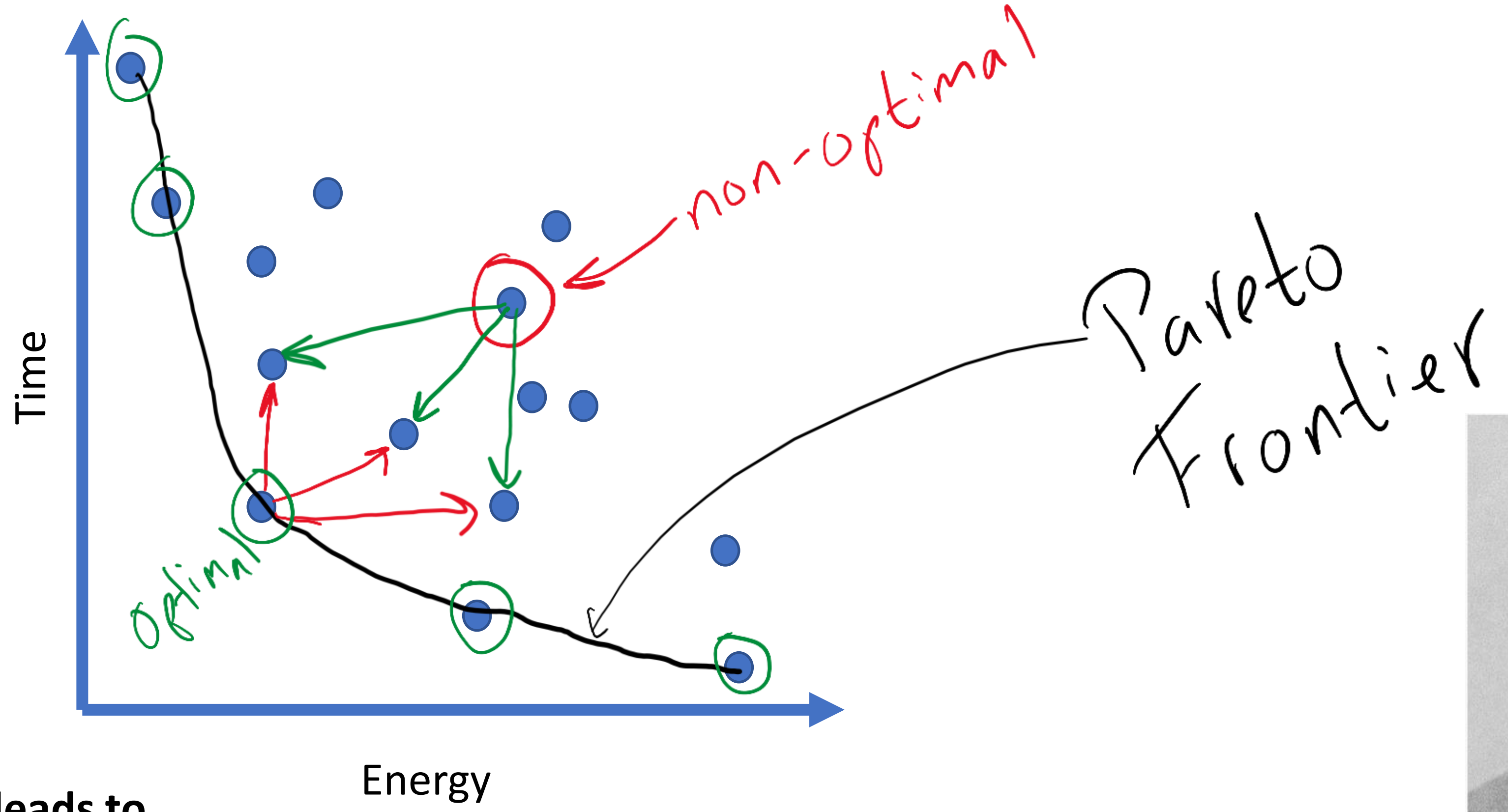
16
1k
xor()
2
hash()
8
2

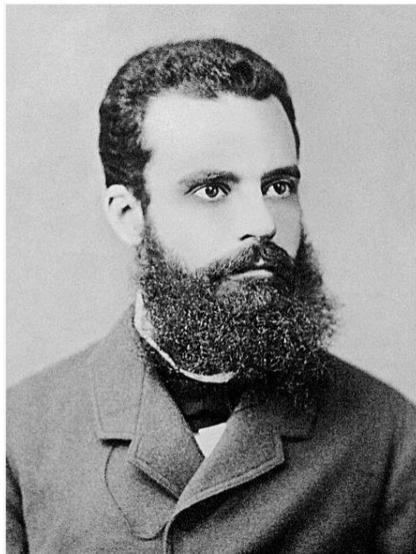
Plotting many designs to study a **tradeoff**

What can we learn from this plot?



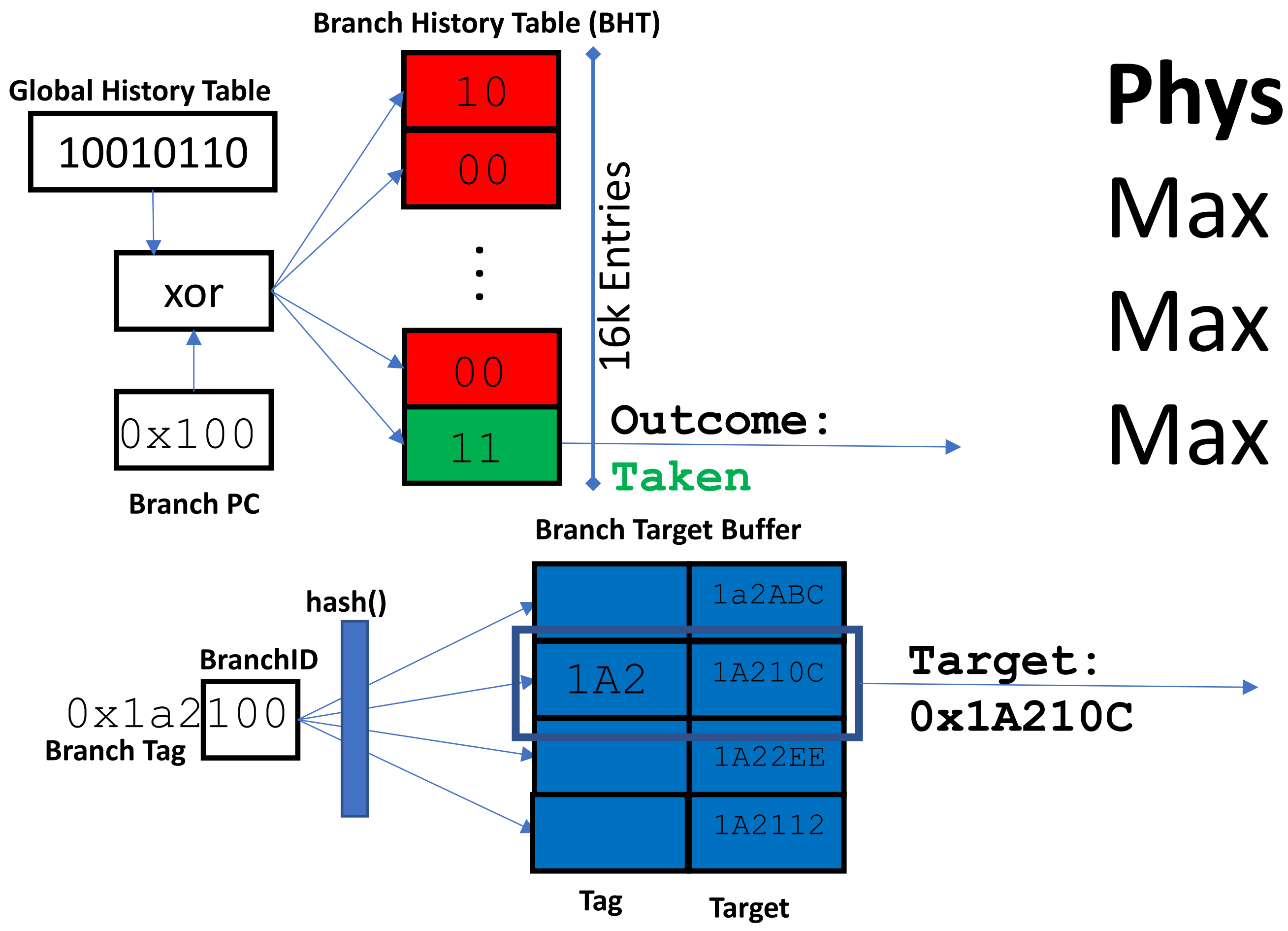
Pareto Optimality of Design Alternatives





Pareto Optimality:
A design is optimal if no change leads to improvement in one dimension without a loss in at least one other dimension

Constraining your design space



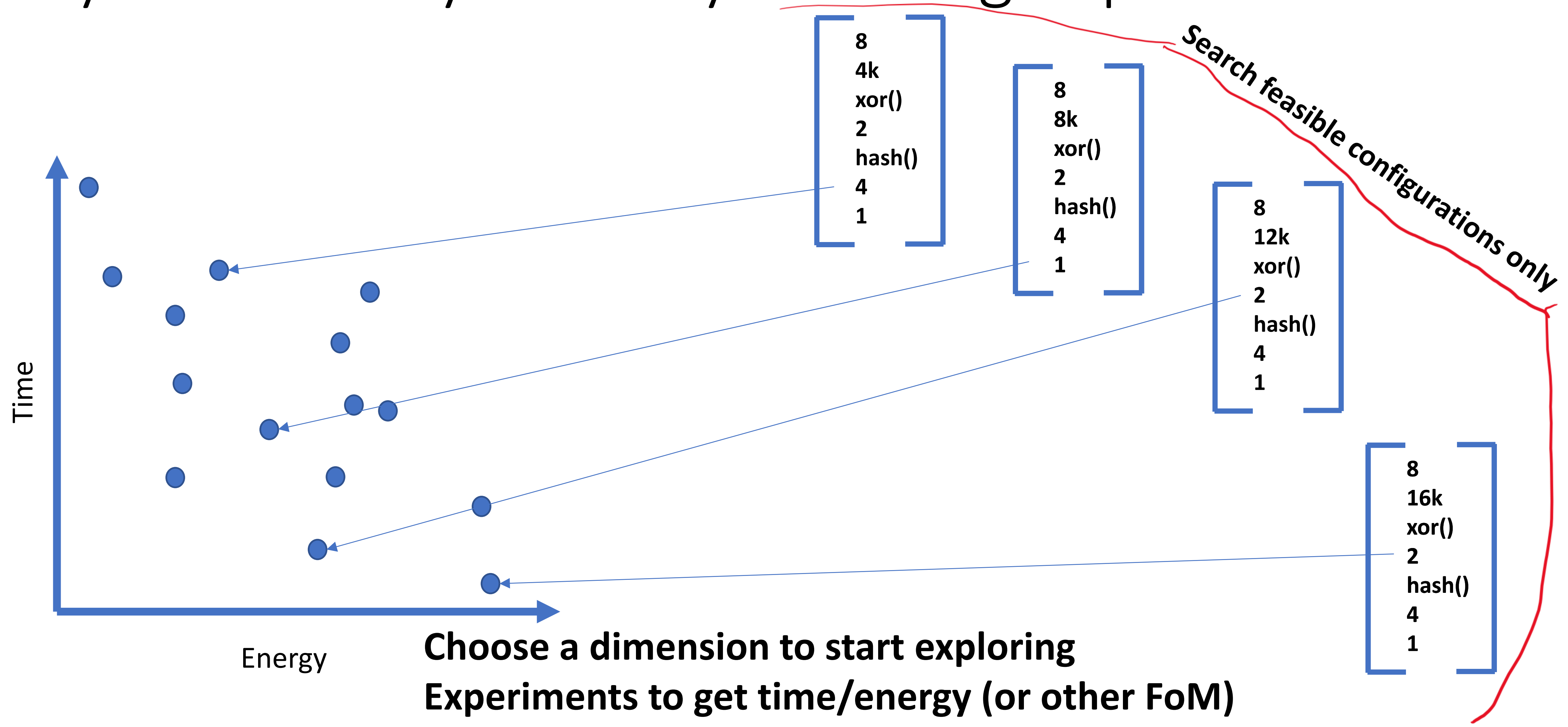
Physical design constraints

Max memory (BTB+BHT) = 20kB

Max BTB associativity = 2

Max BP power = 4mW

Systematically fill out your design space



Choose a dimension to start exploring

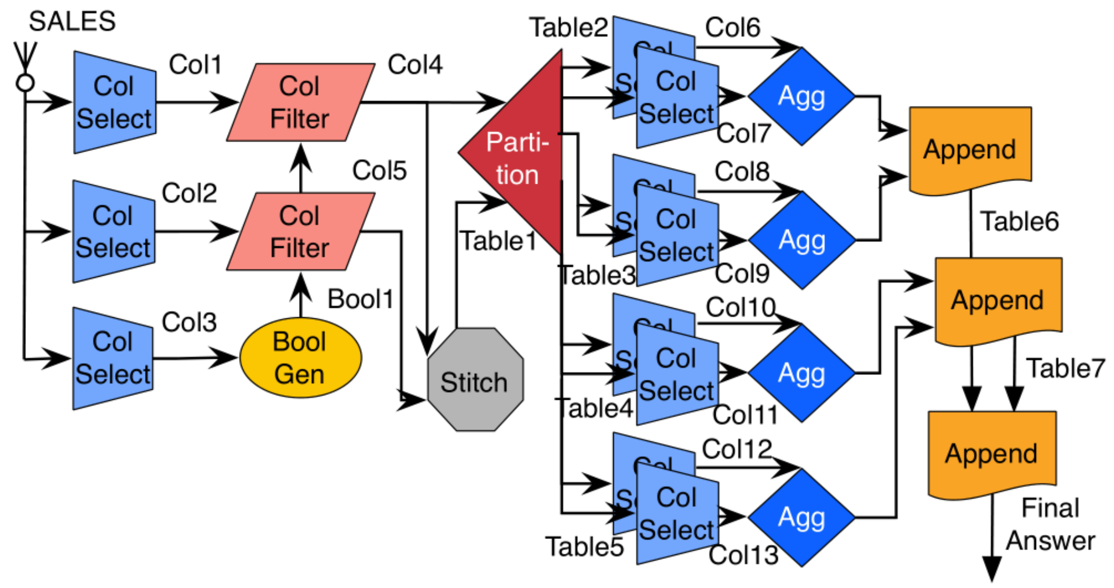
Experiments to get time/energy (or other FoM)

Change dimensions when way off frontier

Example of Design Space Optimization
The Q100 Database Acceleration Architecture

Q100: The Architecture and Design of a Database Processing Unit

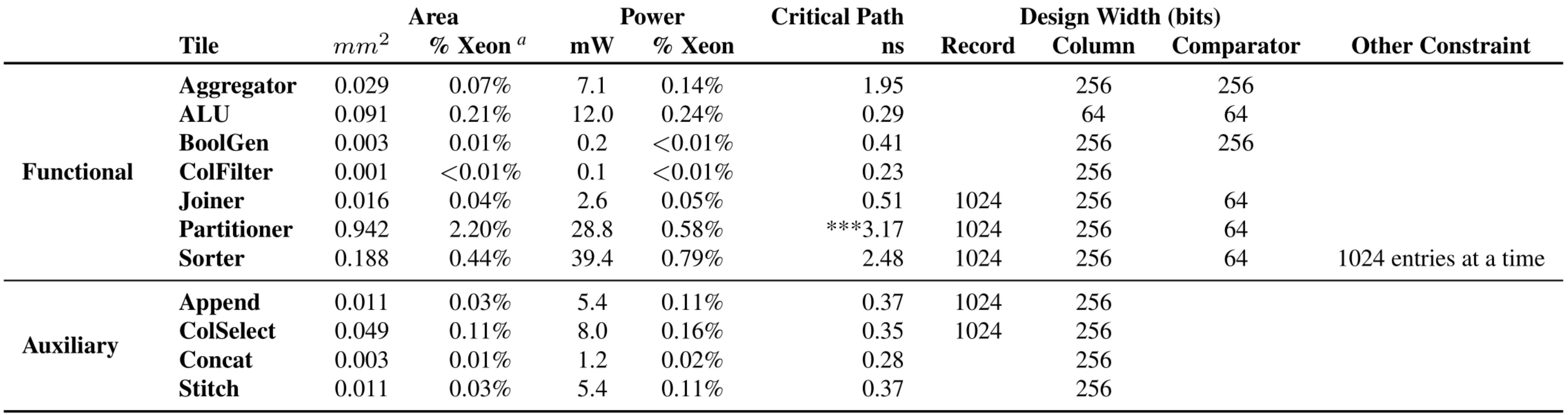
Lisa Wu Andrea Lottarini Timothy K. Paine Martha A. Kim Kenneth A. Ross
Columbia University, New York, NY



Cutting edge database query hardware accelerator

- “GPU for SQL & Database operations”
- Architecture built up of a collection of special computing tiles in hardware
- Each tile runs a particular kind of database operation
- Tiles connected by configurable wires that can be set up to make circuits to do a database query
- (Includes one of the best design space explorations I’ve encountered in a research paper)

Design Space Constraints
The Q100 Database Acceleration Architecture

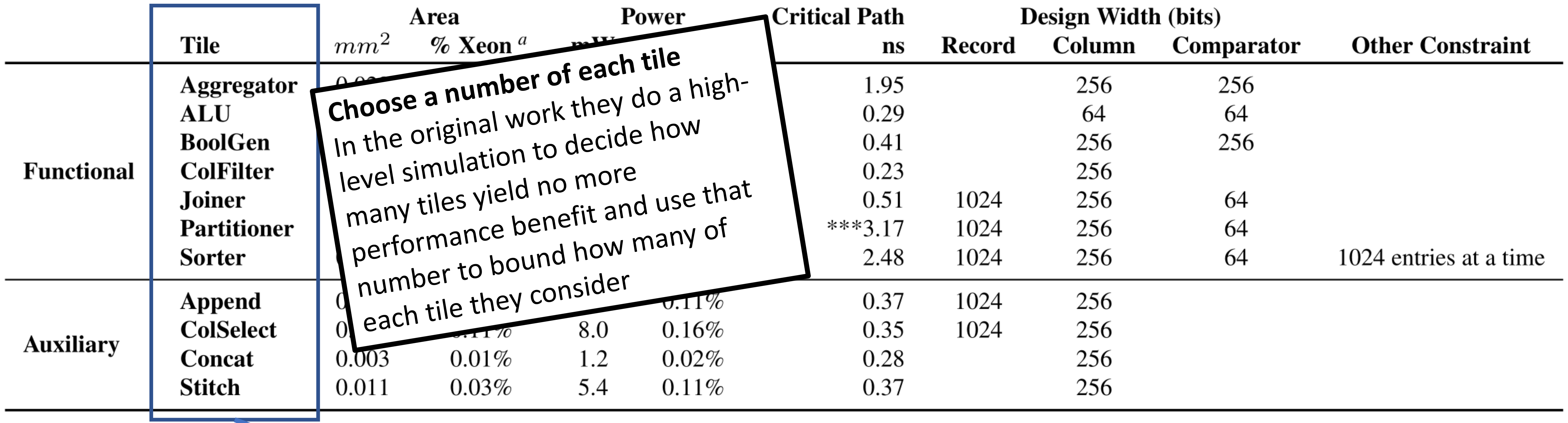


		Area		Power		Critical Path	Design Width (bits)			
	Tile	mm^2	% Xeon [a]	mW	% Xeon	ns	Record	Column	Comparator	Other Constraint
Functional	Aggregator	0.029	0.07%	7.1	0.14%	1.95		256	256	
	ALU	0.091	0.21%	12.0	0.24%	0.29		64	64	
	BoolGen	0.003	0.01%	0.2	<0.01%	0.41		256	256	
	ColFilter	0.001	<0.01%	0.1	<0.01%	0.23		256		
	Joiner	0.016	0.04%	2.6	0.05%	0.51	1024	256	64	
	Partitioner	0.942	2.20%	28.8	0.58%	***3.17	1024	256	64	
	Sorter	0.188	0.44%	39.4	0.79%	2.48	1024	256	64	1024 entries at a time
Auxiliary	Append	0.011	0.03%	5.4	0.11%	0.37	1024	256		
	ColSelect	0.049	0.11%	8.0	0.16%	0.35	1024	256		
	Concat	0.003	0.01%	1.2	0.02%	0.28		256		
	Stitch	0.011	0.03%	5.4	0.11%	0.37		256		

Design space optimization problem statement:

Choose the right mixture of tiles to have the best performance and power without using too much area or limiting frequency

Design Space Constraints
The Q100 Database Acceleration Architecture

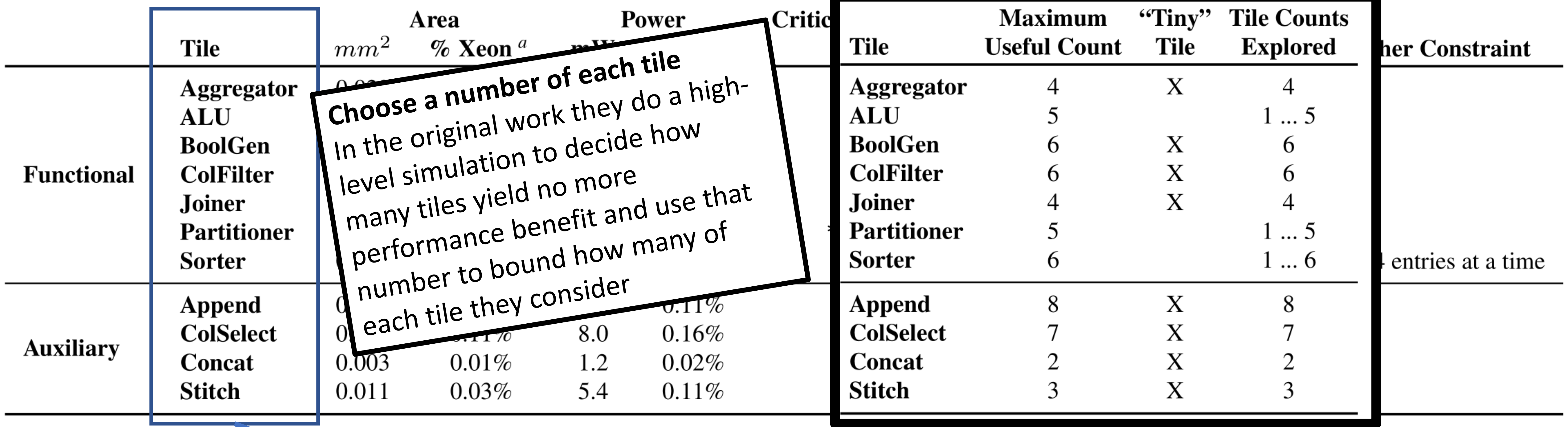


	Tile	Area mm^2	Area % Xeon [a]	Power mW	Power [illegible]	Critical Path ns	Design Width (bits) Record	Design Width (bits) Column	Design Width (bits) Comparator	Other Constraint
Functional	Aggregator	[illegible]	[illegible]	[illegible]	[illegible]	1.95		256	256	
	ALU	[illegible]	[illegible]	[illegible]	[illegible]	0.29		64	64	
	BoolGen	[illegible]	[illegible]	[illegible]	[illegible]	0.41		256	256	
	ColFilter	[illegible]	[illegible]	[illegible]	[illegible]	0.23		256		
	Joiner	[illegible]	[illegible]	[illegible]	[illegible]	0.51	1024	256	64	
	Partitioner	[illegible]	[illegible]	[illegible]	[illegible]	***3.17	1024	256	64	
	Sorter	[illegible]	[illegible]	[illegible]	[illegible]	2.48	1024	256	64	1024 entries at a time
Auxiliary	Append	[illegible]	[illegible]	[illegible]	0.11%	0.37	1024	256		
	ColSelect	[illegible]	[illegible]	8.0	0.16%	0.35	1024	256		
	Concat	0.003	0.01%	1.2	0.02%	0.28		256		
	Stitch	0.011	0.03%	5.4	0.11%	0.37		256		

Choose a number of each tile
In the original work they do a high-level simulation to decide how many tiles yield no more performance benefit and use that number to bound how many of each tile they consider

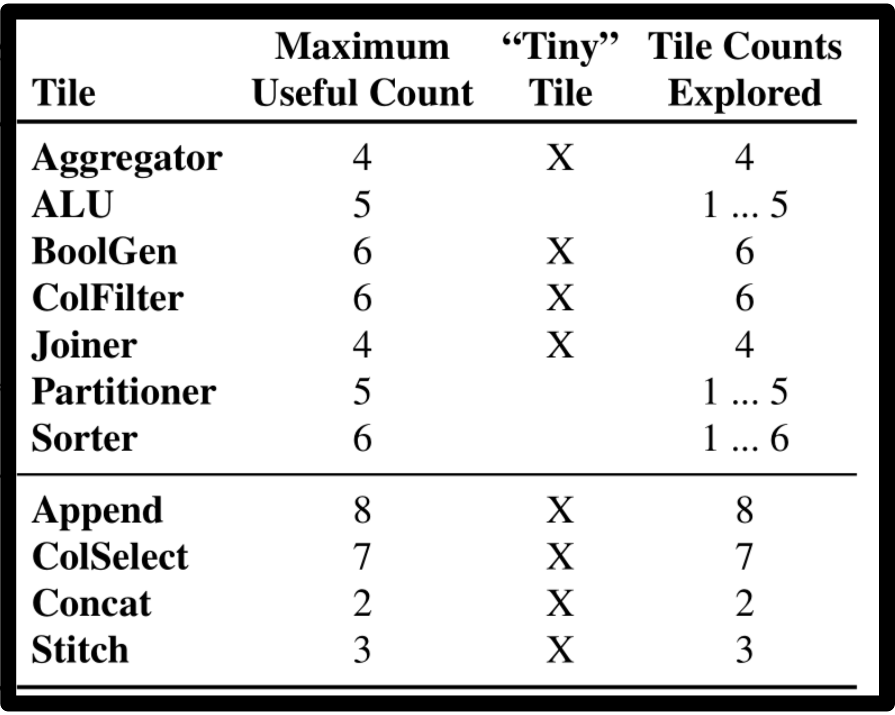
Design space optimization problem statement:
Choose the right mixture of tiles to have the best performance and power without using too much area or limiting frequency

Design Space Constraints
The Q100 Database Acceleration Architecture



	Tile	Area mm^2	Area % Xeon [a]	Power mW	Power	Critic	her Constraint
Functional	Aggregator						
	ALU						
	BoolGen						
	ColFilter						
	Joiner						
	Partitioner						
	Sorter						entries at a time
Auxiliary	Append	0			0.11%		
	ColSelect	0		8.0	0.16%		
	Concat	0.003	0.01%	1.2	0.02%		
	Stitch	0.011	0.03%	5.4	0.11%		

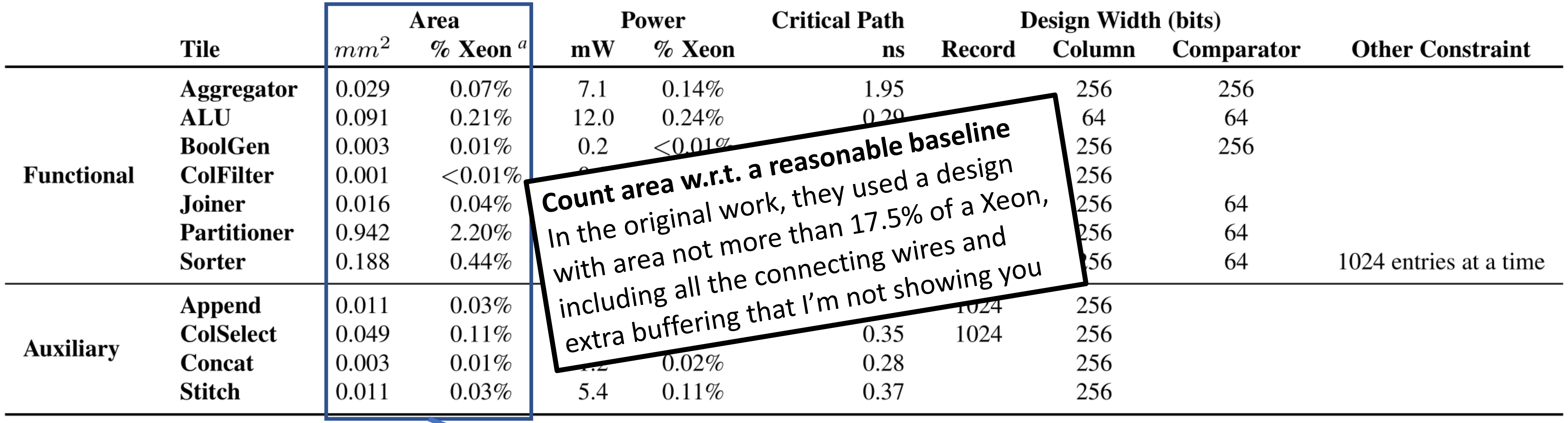
Choose a number of each tile
In the original work they do a high-level simulation to decide how many tiles yield no more performance benefit and use that number to bound how many of each tile they consider



Tile	Maximum Useful Count	"Tiny" Tile	Tile Counts Explored
Aggregator	4	X	4
ALU	5		1 ... 5
BoolGen	6	X	6
ColFilter	6	X	6
Joiner	4	X	4
Partitioner	5		1 ... 5
Sorter	6		1 ... 6
Append	8	X	8
ColSelect	7	X	7
Concat	2	X	2
Stitch	3	X	3

Design space optimization problem statement:
Choose the right mixture of tiles to have the best performance and power without using too much area or limiting frequency

Design Space Constraints
The Q100 Database Acceleration Architecture

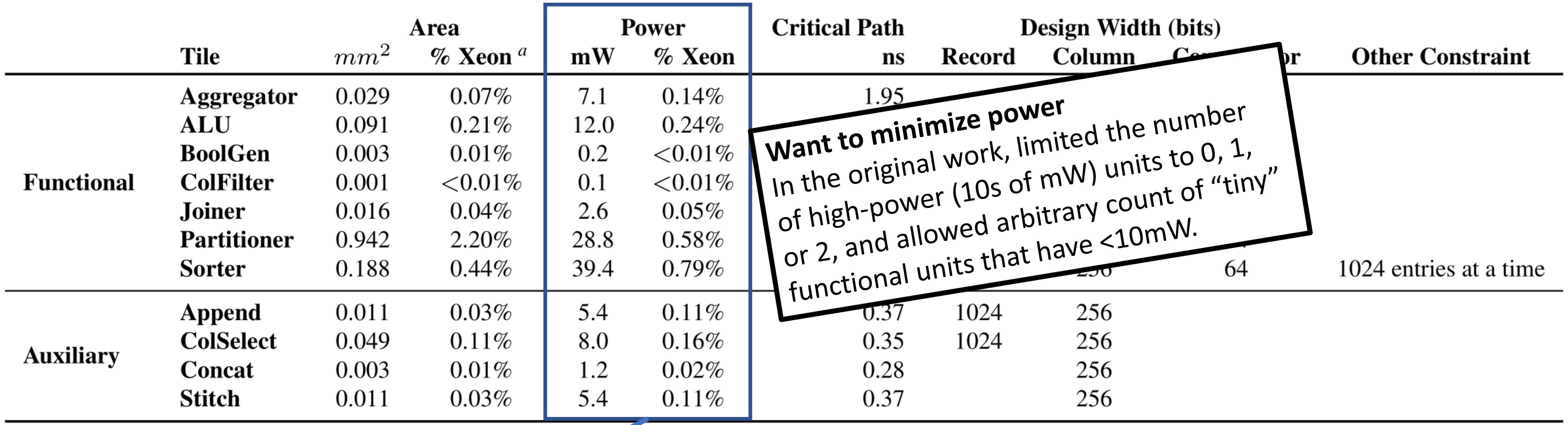


	Tile	Area		Power		Critical Path	Design Width (bits)			Other Constraint
		mm^2	% Xeon [a]	mW	% Xeon	ns	Record	Column	Comparator	
Functional	**Aggregator**	0.029	0.07%	7.1	0.14%	1.95		256	256	
	ALU	0.091	0.21%	12.0	0.24%	0.29		64	64	
	BoolGen	0.003	0.01%	0.2	<0.01%	[illegible]		256	256	
	ColFilter	0.001	<0.01%	[illegible]	[illegible]	[illegible]	[illegible]	256		
	Joiner	0.016	0.04%	[illegible]	[illegible]	[illegible]	[illegible]	256	64	
	Partitioner	0.942	2.20%	[illegible]	[illegible]	[illegible]	[illegible]	256	64	
	Sorter	0.188	0.44%	[illegible]	[illegible]	[illegible]	[illegible]	256	64	1024 entries at a time
Auxiliary	**Append**	0.011	0.03%	[illegible]	[illegible]	[illegible]	1024	256		
	ColSelect	0.049	0.11%	[illegible]	[illegible]	0.35	1024	256		
	Concat	0.003	0.01%	1.2	0.02%	0.28		256		
	Stitch	0.011	0.03%	5.4	0.11%	0.37		256		

Count area w.r.t. a reasonable baseline
In the original work, they used a design with area not more than 17.5% of a Xeon, including all the connecting wires and extra buffering that I'm not showing you

Design space optimization problem statement:
Choose the right mixture of tiles to have the best performance and power without using too much area or limiting frequency

Design Space Constraints
The Q100 Database Acceleration Architecture

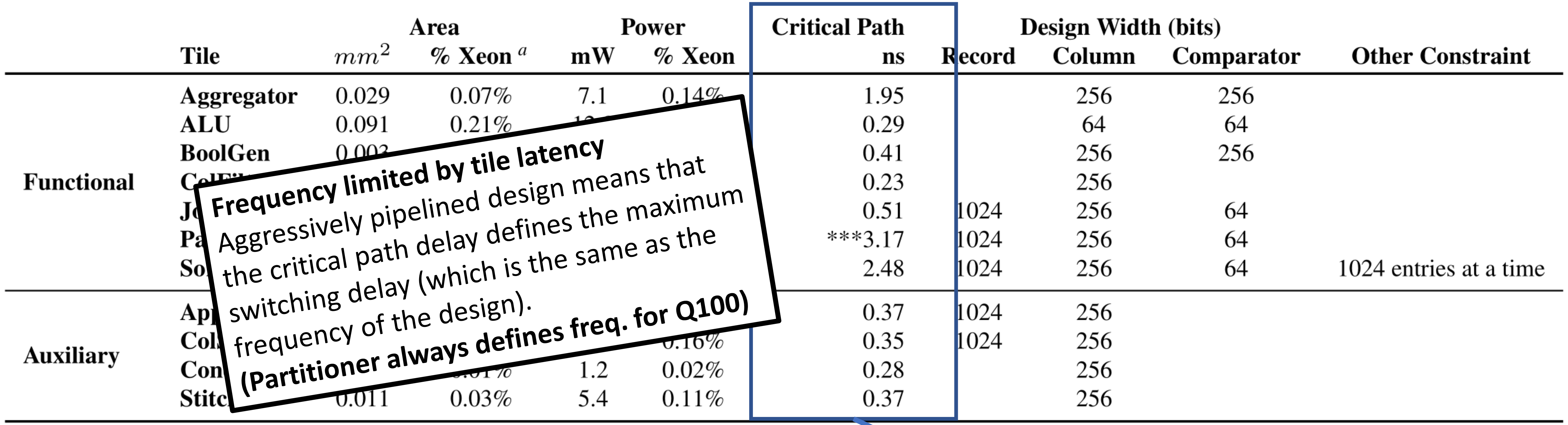


	Tile	Area mm^2	Area % Xeon [a]	Power mW	Power % Xeon	Critical Path ns	Design Width (bits) Record	Design Width (bits) Column	Design Width (bits) Co[illegible]or	Other Constraint
Functional	**Aggregator**	0.029	0.07%	7.1	0.14%	1.95				
	ALU	0.091	0.21%	12.0	0.24%					
	BoolGen	0.003	0.01%	0.2	<0.01%					
	ColFilter	0.001	<0.01%	0.1	<0.01%					
	Joiner	0.016	0.04%	2.6	0.05%					
	Partitioner	0.942	2.20%	28.8	0.58%					
	Sorter	0.188	0.44%	39.4	0.79%			256	64	1024 entries at a time
Auxiliary	**Append**	0.011	0.03%	5.4	0.11%	0.37	1024	256		
	ColSelect	0.049	0.11%	8.0	0.16%	0.35	1024	256		
	Concat	0.003	0.01%	1.2	0.02%	0.28		256		
	Stitch	0.011	0.03%	5.4	0.11%	0.37		256		

Want to minimize power
In the original work, limited the number of high-power (10s of mW) units to 0, 1, or 2, and allowed arbitrary count of "tiny" functional units that have <10mW.

Design space optimization problem statement:
Choose the right mixture of tiles to have the best performance and power without using too much area or limiting frequency

Design Space Constraints
The Q100 Database Acceleration Architecture

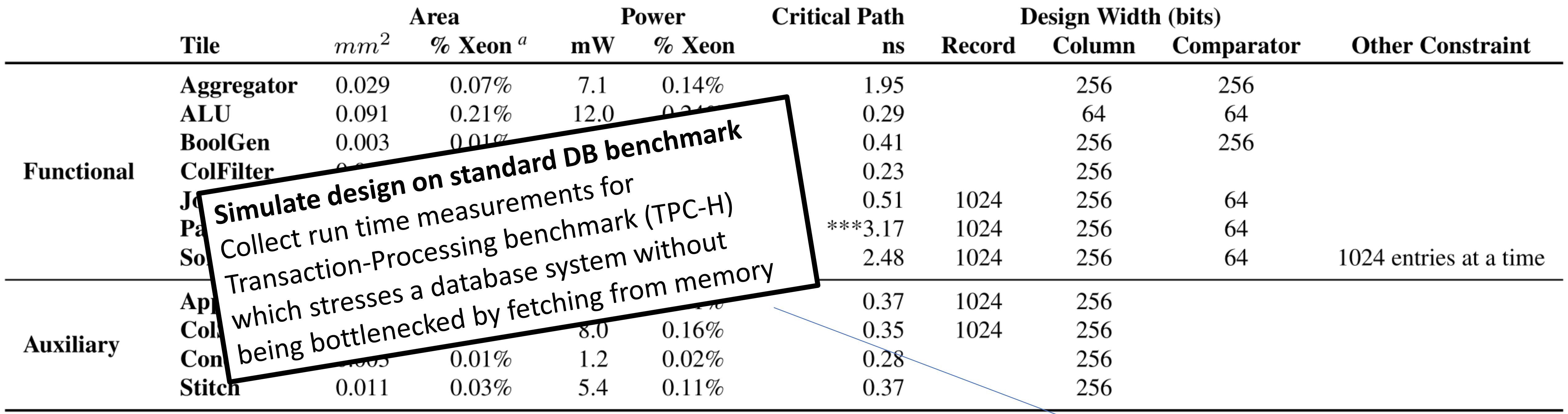


		Area		Power		Critical Path	Design Width (bits)			
	Tile	mm^2	**% Xeon** [a]	**mW**	**% Xeon**	**ns**	**Record**	**Column**	**Comparator**	**Other Constraint**
Functional	**Aggregator**	0.029	0.07%	7.1	0.14%	1.95		256	256	
	ALU	0.091	0.21%	[illegible]	[illegible]	0.29		64	64	
	BoolGen	0.003	[illegible]	[illegible]	[illegible]	0.41		256	256	
	ColFi	[illegible]	[illegible]	[illegible]	[illegible]	0.23		256		
	Jo	[illegible]	[illegible]	[illegible]	[illegible]	0.51	1024	256	64	
	Pa	[illegible]	[illegible]	[illegible]	[illegible]	***3.17	1024	256	64	
	So	[illegible]	[illegible]	[illegible]	[illegible]	2.48	1024	256	64	1024 entries at a time
Auxiliary	**Ap**	[illegible]	[illegible]	[illegible]	[illegible]	0.37	1024	256		
	Col	[illegible]	[illegible]	[illegible]	0.16%	0.35	1024	256		
	Con	[illegible]	0.01%	1.2	0.02%	0.28		256		
	Stitc	0.011	0.03%	5.4	0.11%	0.37		256		

Frequency limited by tile latency
Aggressively pipelined design means that the critical path delay defines the maximum switching delay (which is the same as the frequency of the design).
(Partitioner always defines freq. for Q100)

Design space optimization problem statement:
Choose the right mixture of tiles to have the best performance and power without using too much area or limiting frequency

Design Space Constraints
The Q100 Database Acceleration Architecture

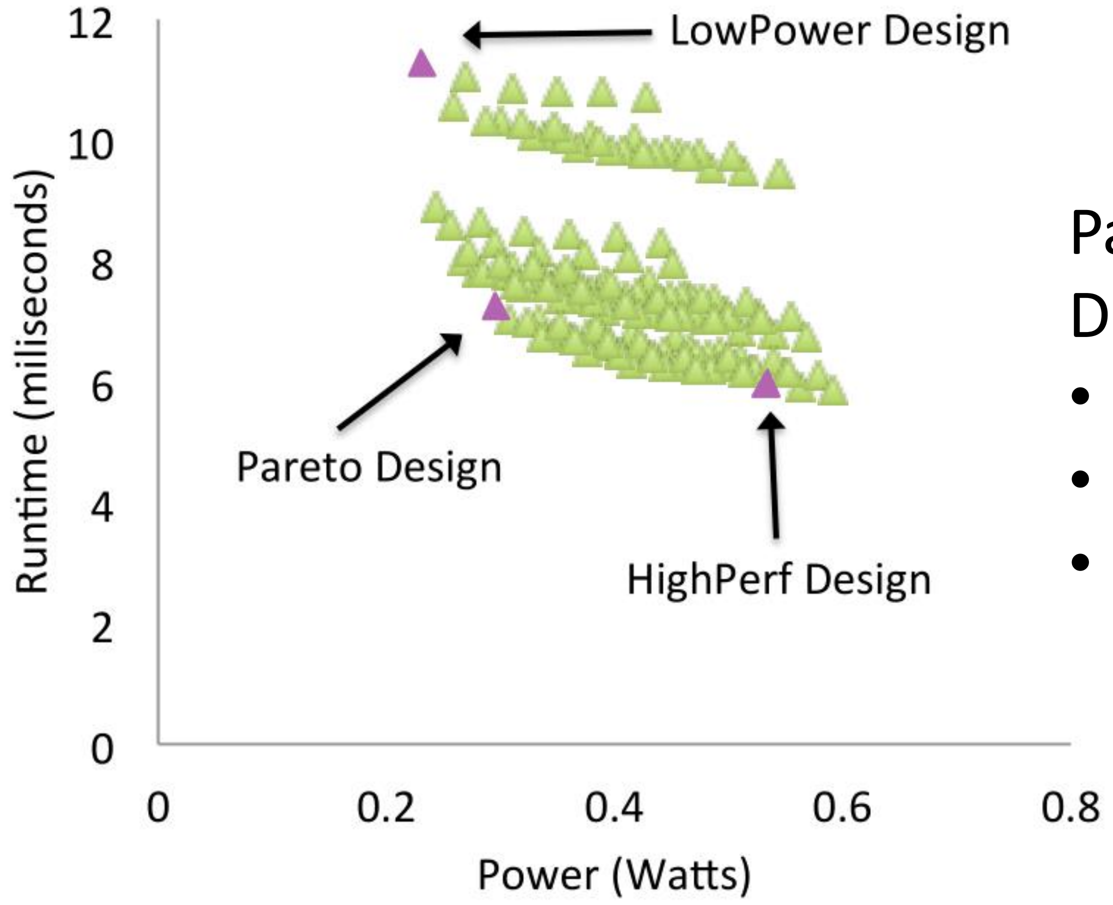


		Area		Power		Critical Path	Design Width (bits)			
	Tile	mm^2	**% Xeon** [a]	**mW**	**% Xeon**	**ns**	**Record**	**Column**	**Comparator**	**Other Constraint**
Functional	**Aggregator**	0.029	0.07%	7.1	0.14%	1.95		256	256	
	ALU	0.091	0.21%	12.0	[illegible]	0.29		64	64	
	BoolGen	0.003	[illegible]	[illegible]	[illegible]	0.41		256	256	
	ColFilter	[illegible]	[illegible]	[illegible]	[illegible]	0.23		256		
	Jo	[illegible]	[illegible]	[illegible]	[illegible]	0.51	1024	256	64	
	Pa	[illegible]	[illegible]	[illegible]	[illegible]	***3.17	1024	256	64	
	So	[illegible]	[illegible]	[illegible]	[illegible]	2.48	1024	256	64	1024 entries at a time
Auxiliary	**Ap**	[illegible]	[illegible]	[illegible]	[illegible]	0.37	1024	256		
	Col	[illegible]	[illegible]	8.0	0.16%	0.35	1024	256		
	Con	[illegible]	0.01%	1.2	0.02%	0.28		256		
	Stitch	0.011	0.03%	5.4	0.11%	0.37		256		

Simulate design on standard DB benchmark
Collect run time measurements for Transaction-Processing benchmark (TPC-H) which stresses a database system without being bottlenecked by fetching from memory

Design space optimization problem statement:
Choose the right mixture of tiles to have the best performance and power without using too much area or limiting frequency

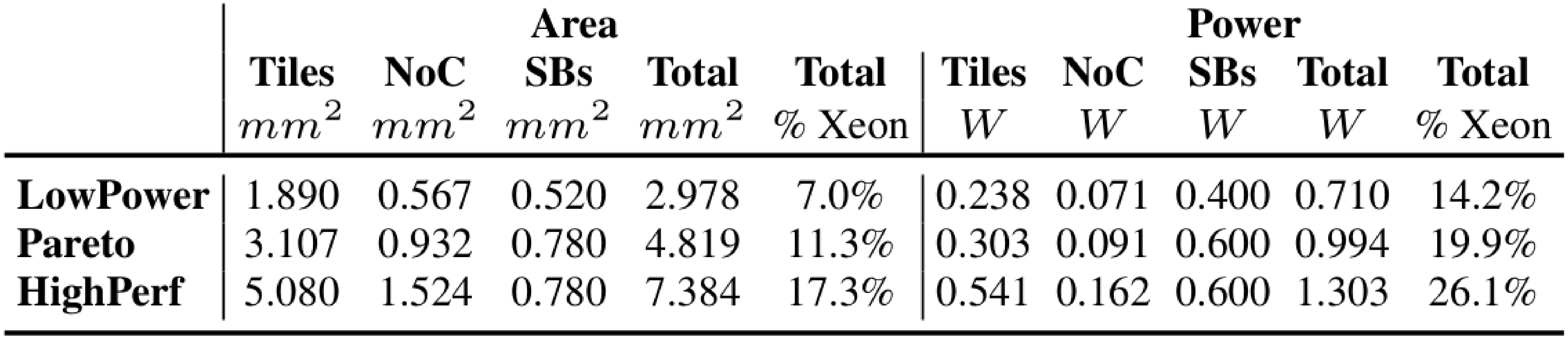
Q100 Pareto Frontier



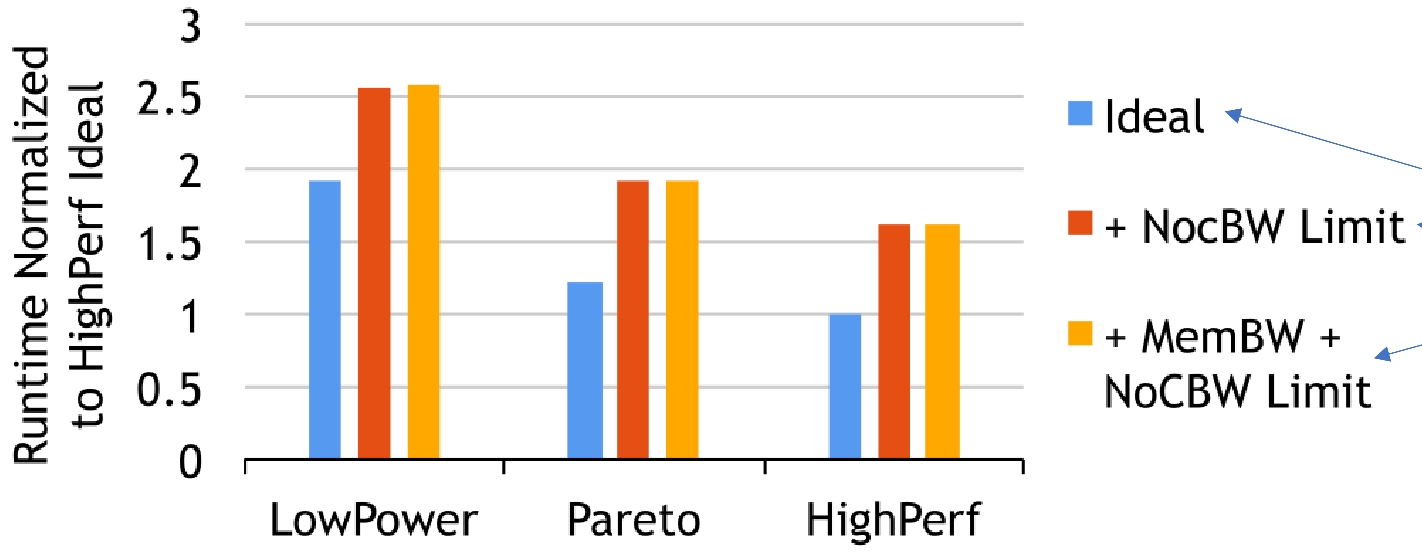
Pareto plot from a research paper on the Q100 Database accelerator by Wu et al, ASPLOS 2014

- How did they select magenta points?
- What other points might they have selected?
- What is the value in seeing all these points?

Results of Design Space Exploration



	Area					Power				
	Tiles	NoC	SBs	Total	Total	Tiles	NoC	SBs	Total	Total
	mm^2	mm^2	mm^2	mm^2	% Xeon	W	W	W	W	% Xeon
LowPower	1.890	0.567	0.520	2.978	7.0%	0.238	0.071	0.400	0.710	14.2%
Pareto	3.107	0.932	0.780	4.819	11.3%	0.303	0.091	0.600	0.994	19.9%
HighPerf	5.080	1.524	0.780	7.384	17.3%	0.541	0.162	0.600	1.303	26.1%

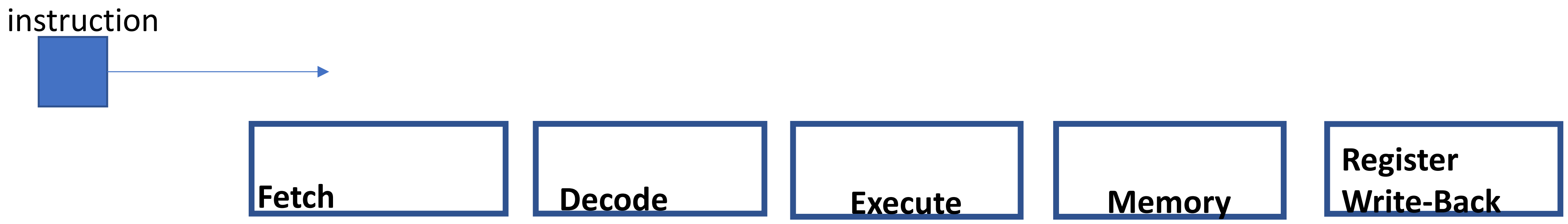


Final results show idealized design and results that include adding in costs related to the on-chip network and memory access bandwidth

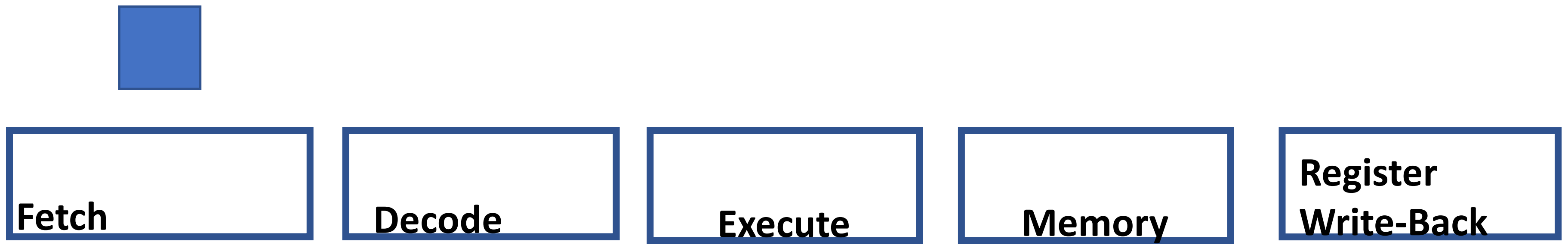
Today: Advanced Microarchitecture Techniques

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

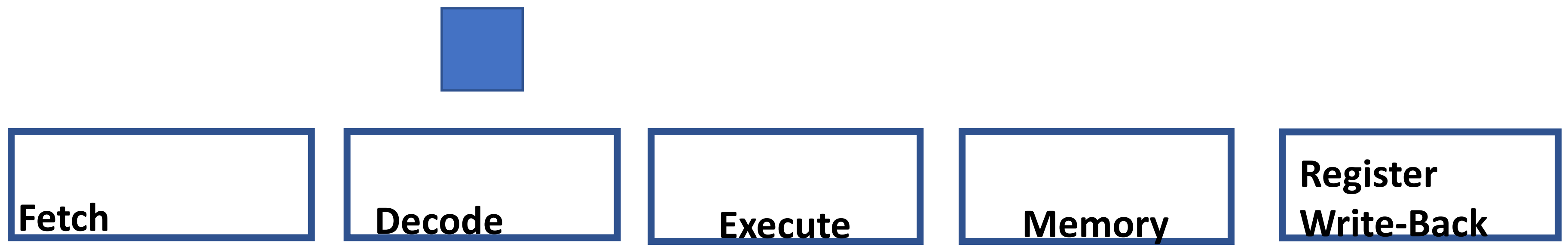
Pipelined scalar design



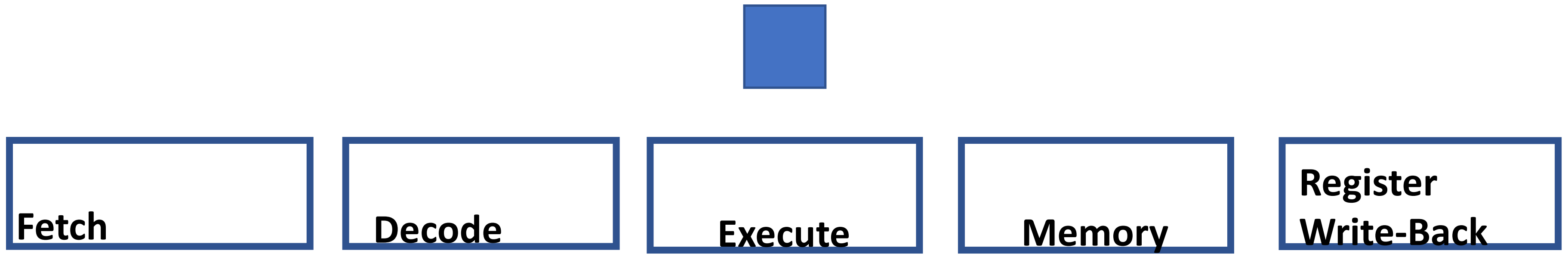
Pipelined scalar design



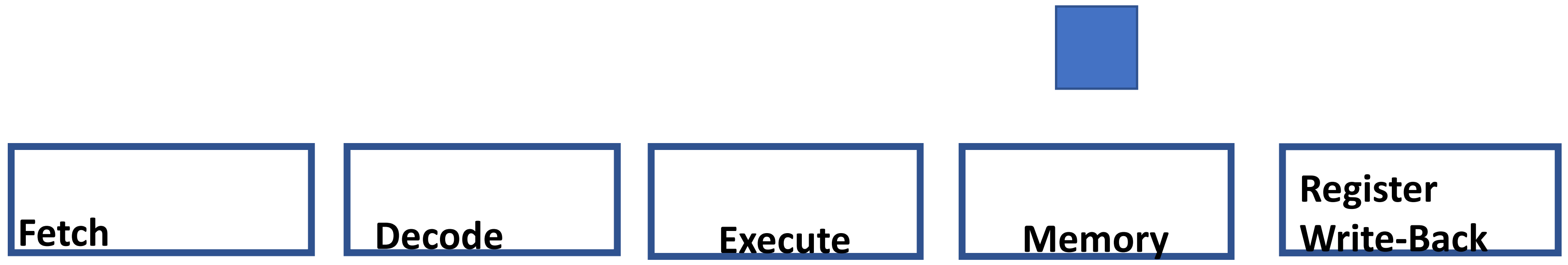
Pipelined scalar design



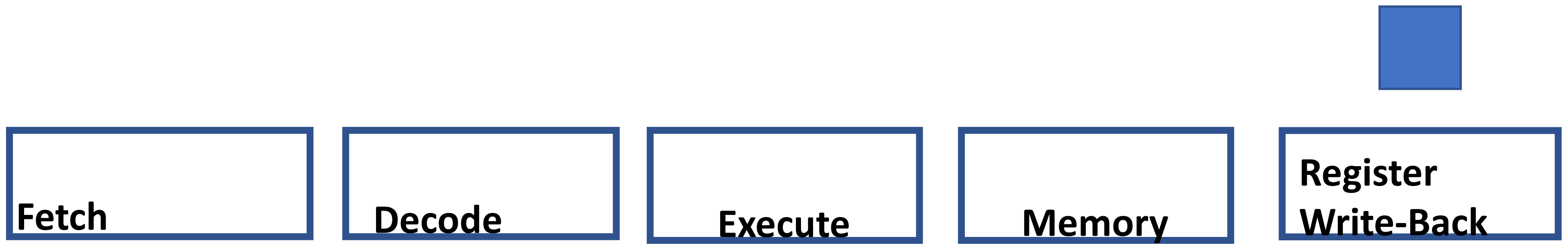
Pipelined scalar design



Pipelined scalar design

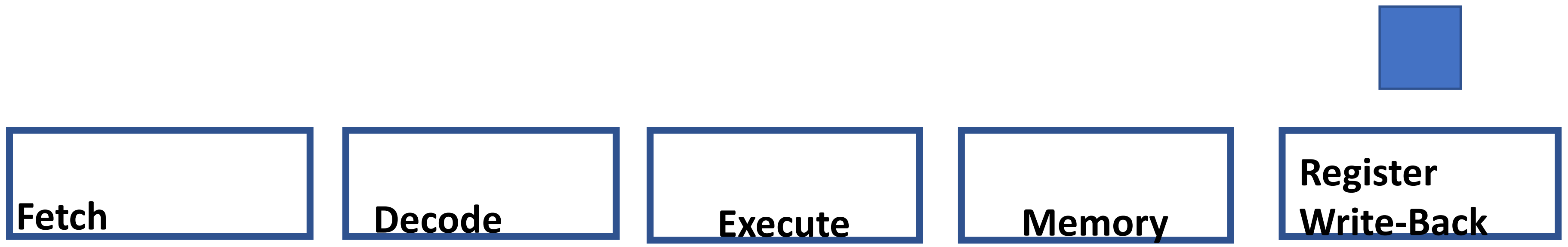


Pipelined scalar design



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

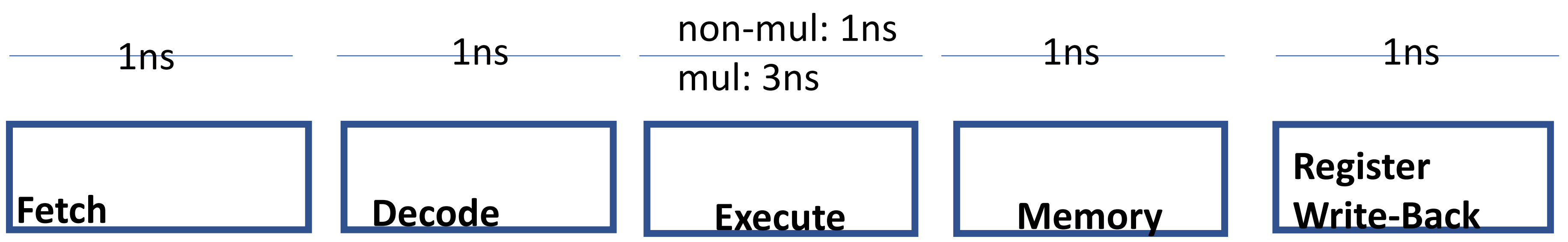


Iron Law of Processor Performance:

Instr / Prog x Cycles / Instr x Seconds / Cycle

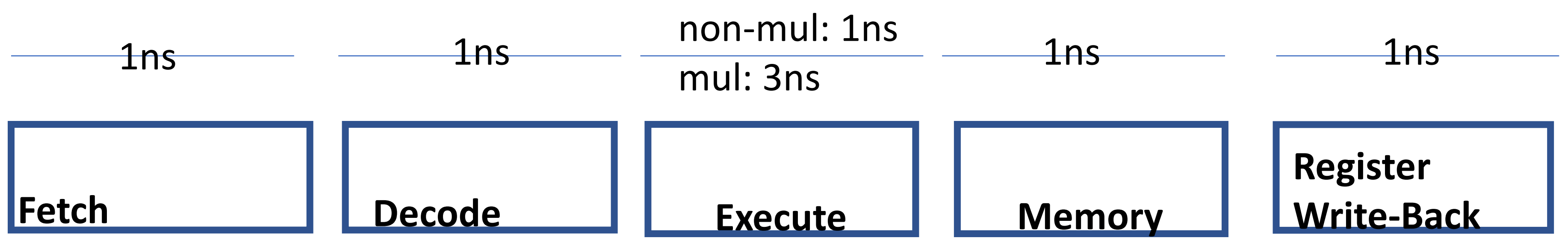
Fundamental limits to each of these terms in our current pipeline?

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?

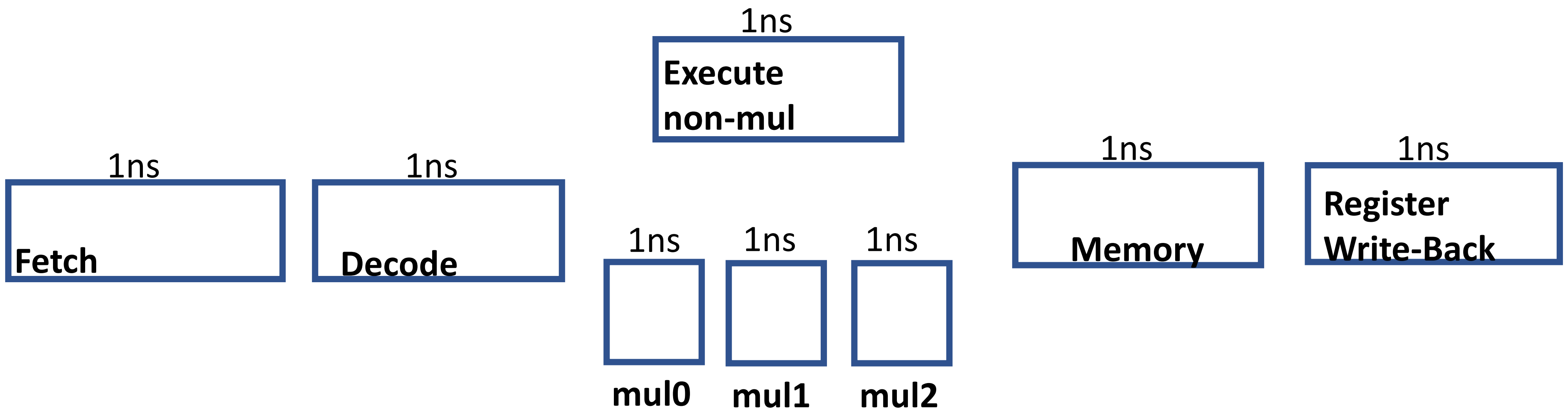
Thinking about latency (again) to optimize for cycle time



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

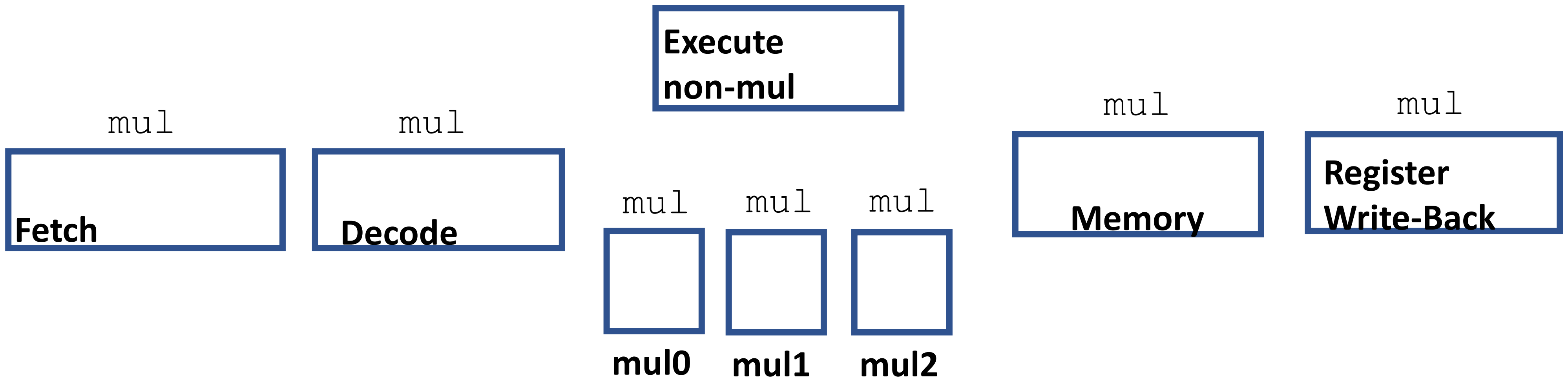
333MHz max clock frequency
(despite 1GHz being OK for non-mul operations)

What if we pipeline the multiplier independently?



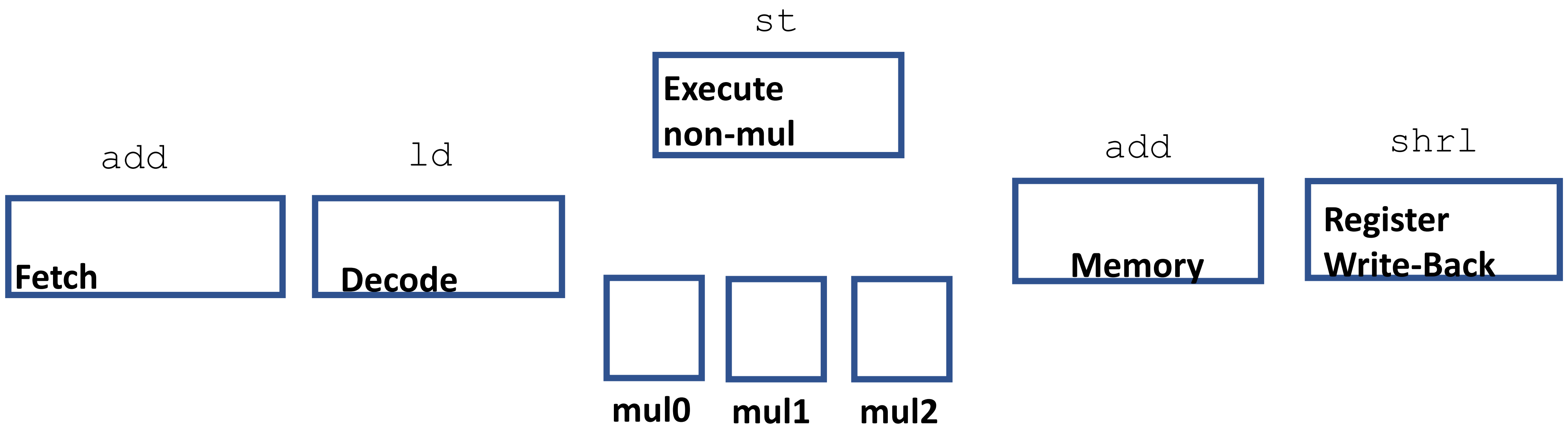
Break the multiply unit into 3 parts, each of which takes 1ns, equalizing all stages' latencies

What if we pipeline the multiplier independently?



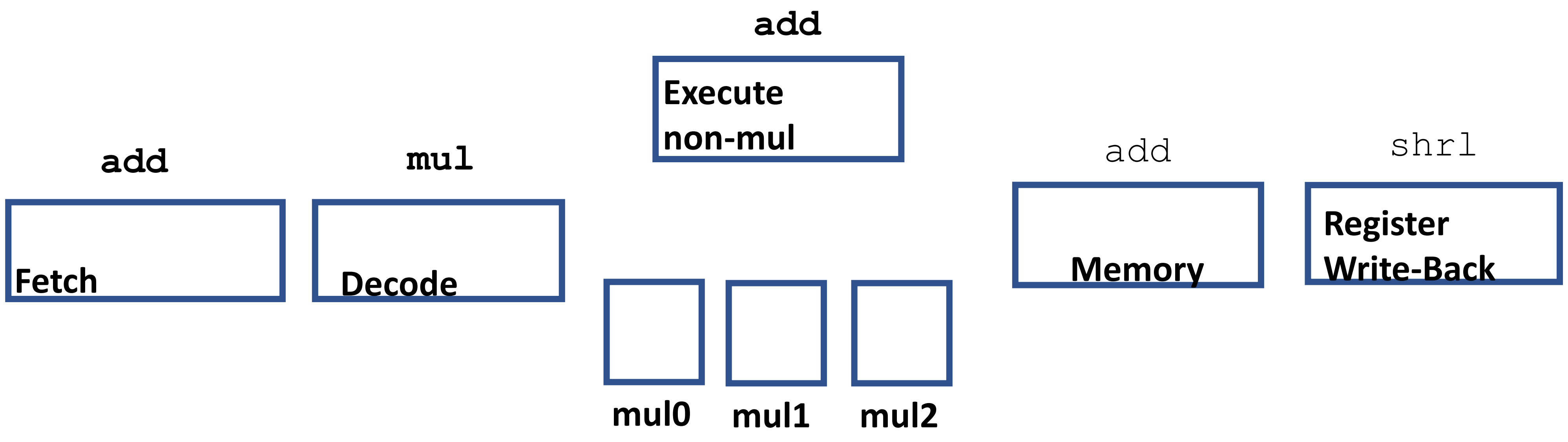
Back-to-back multiplies keep the mul pipe full, at 1GHz latency

What if we pipeline the multiplier independently?



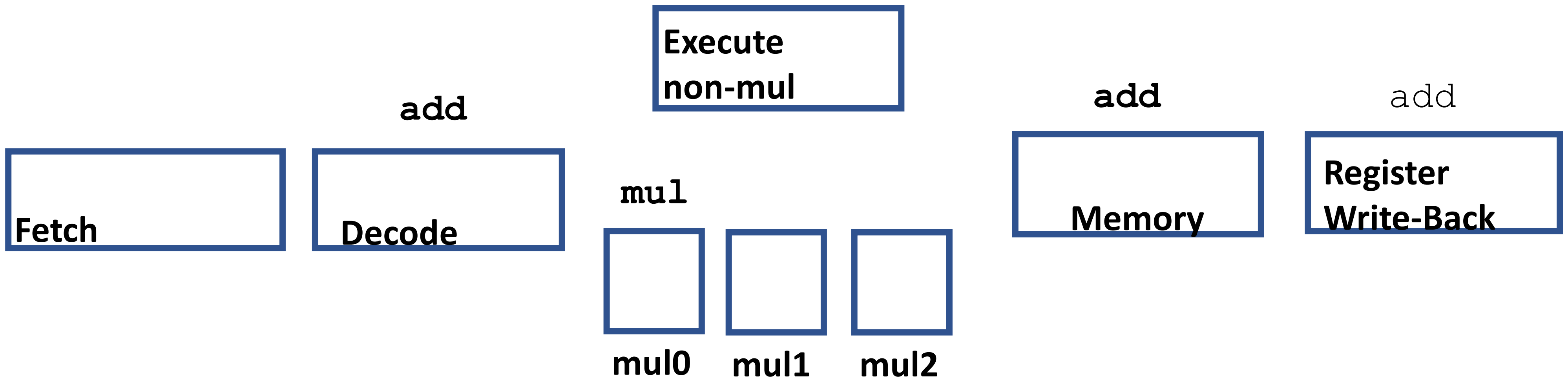
Back-to-back non-mul ops keep the pipe full, at 1GHz latency

What if we pipeline the multiplier independently?



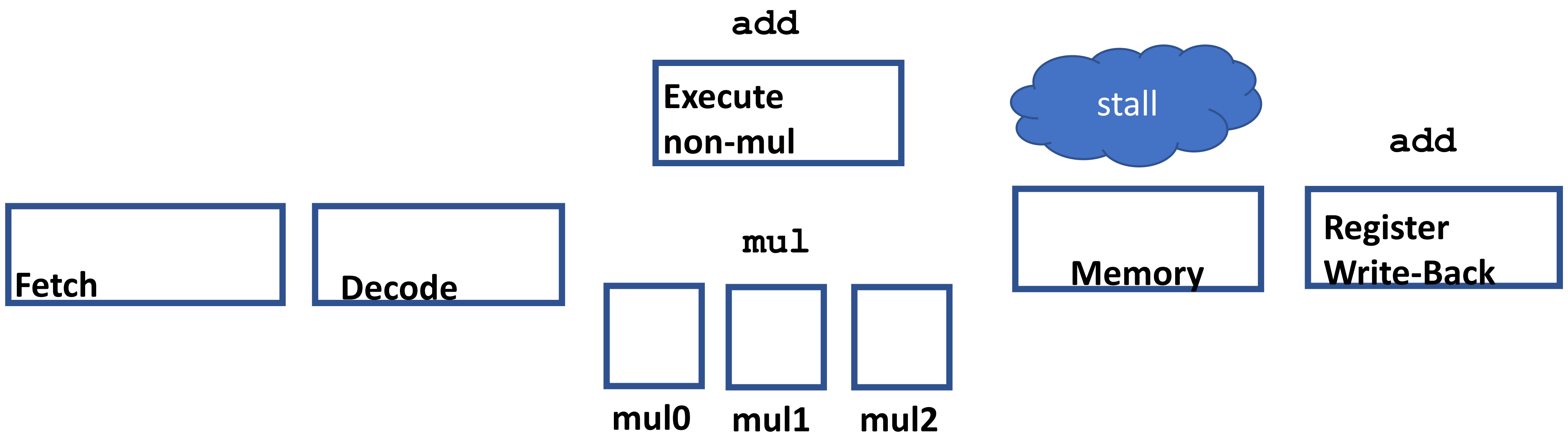
Question: What about add mul add mul?

What if we pipeline the multiplier independently?



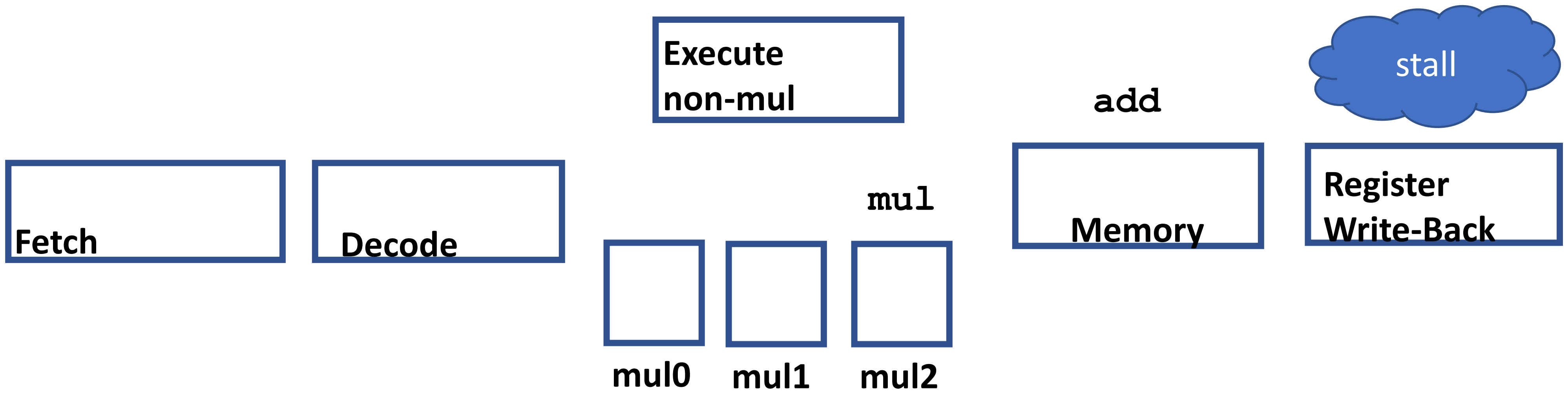
Question: What about add mul add mul?

What if we pipeline the multiplier independently?



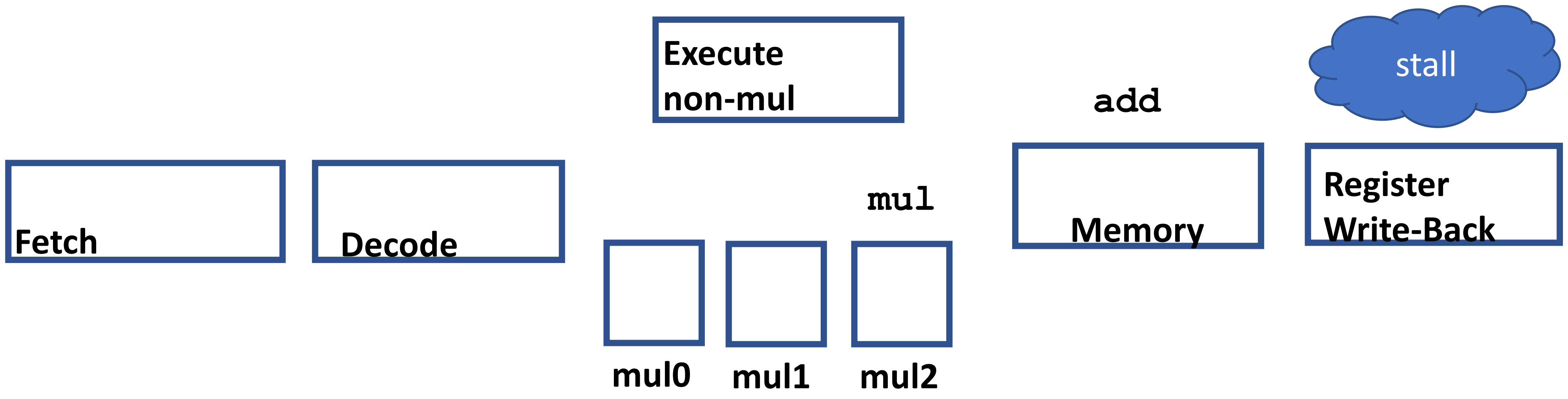
Question: What about add mul add mul?

What if we pipeline the multiplier independently?



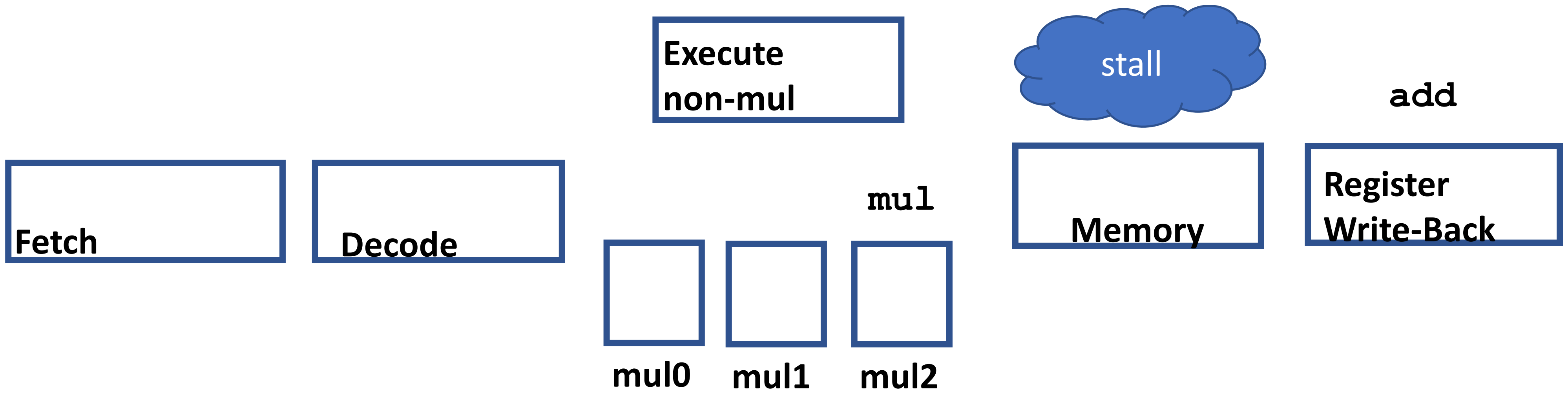
Problem?

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**!

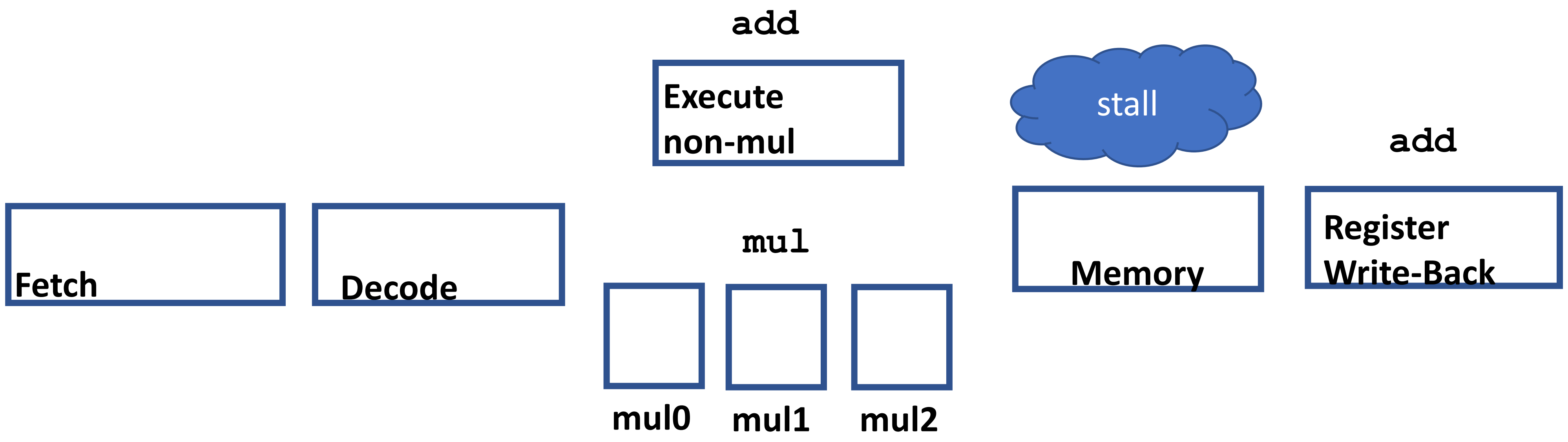
Avoiding out-of-order completion



Hard to avoid the *stall*…

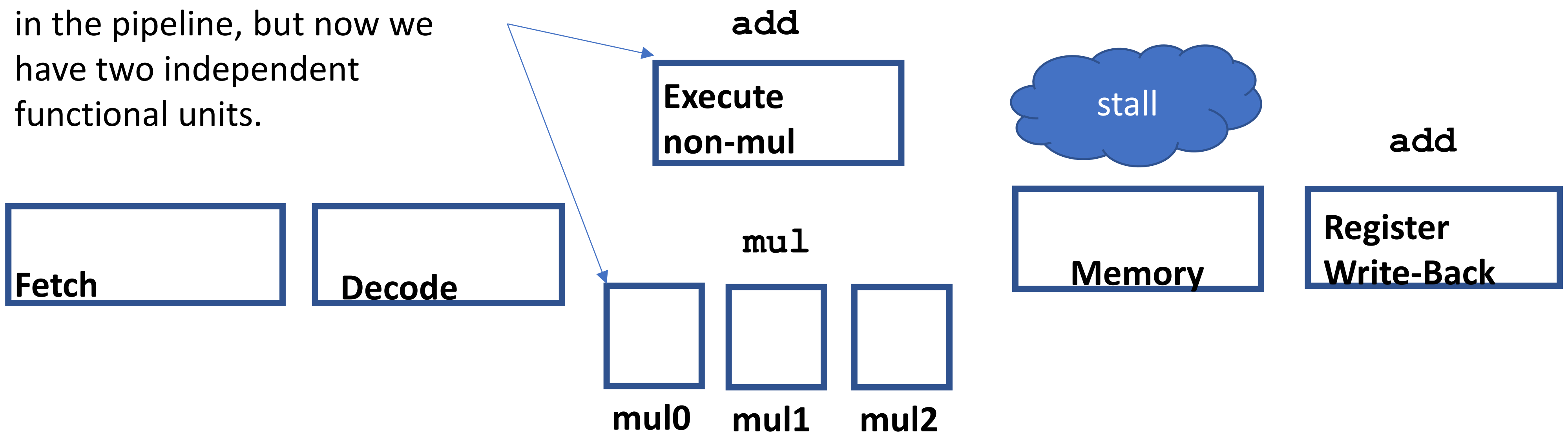
Can avoid the *ordering problem* with extra stall logic in **Ex**

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.

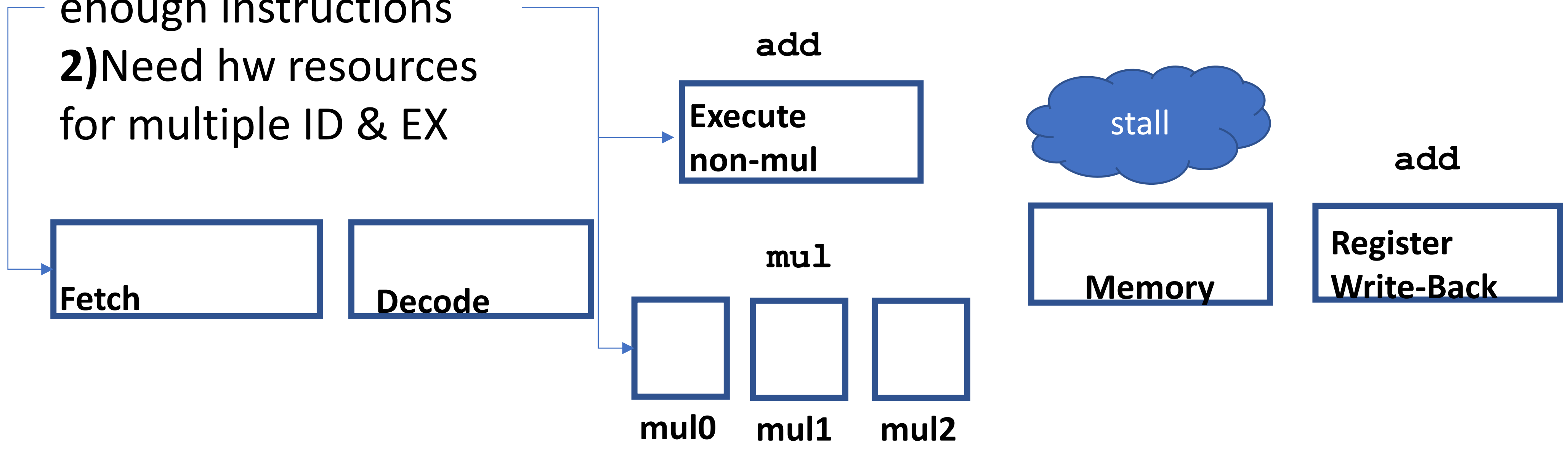


This pipeline is **Ex**ecuting multiple instructions at the same time on different functional units. **ILP begets IPC!**

Superscalar Out of Order Execution

A Superscalar Processor Executes Multiple Instructions at the Same Time

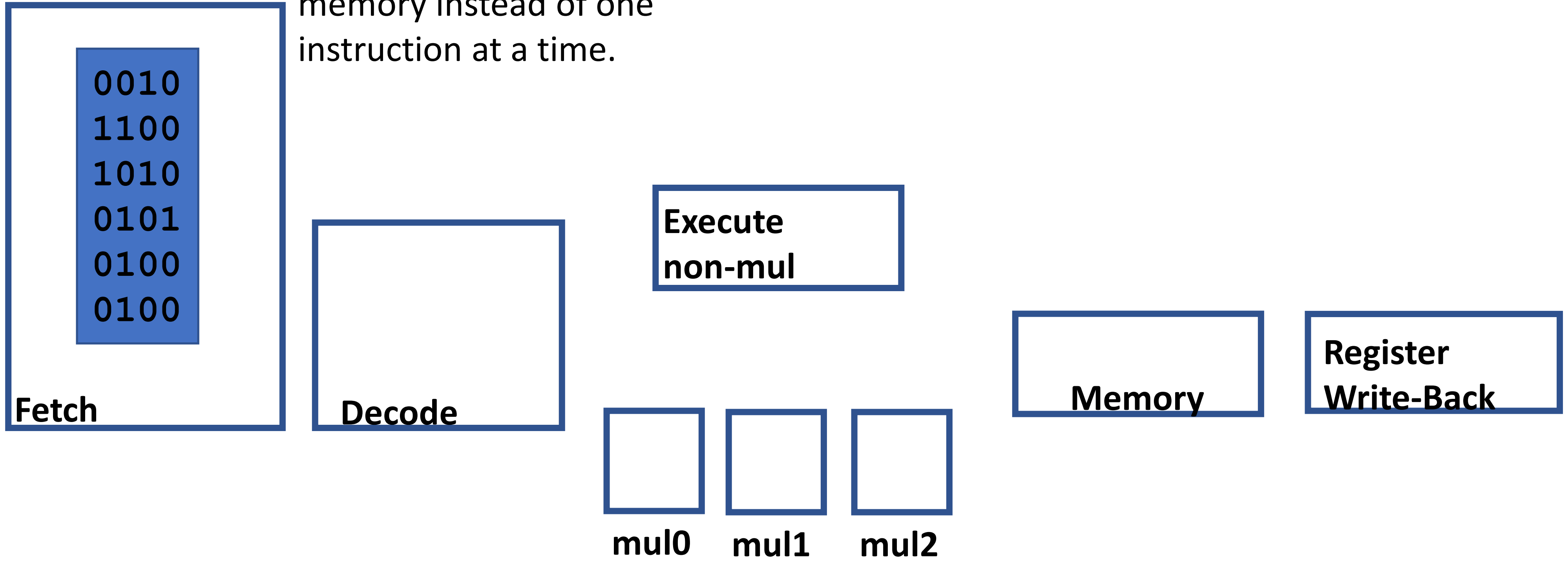
Front End Challenges:
1)Need to supply enough instructions
2)Need hw resources for multiple ID & EX



Scalar executes one instruction at a time
Superscalar executes multiple instructions at a time

Superscalar processors

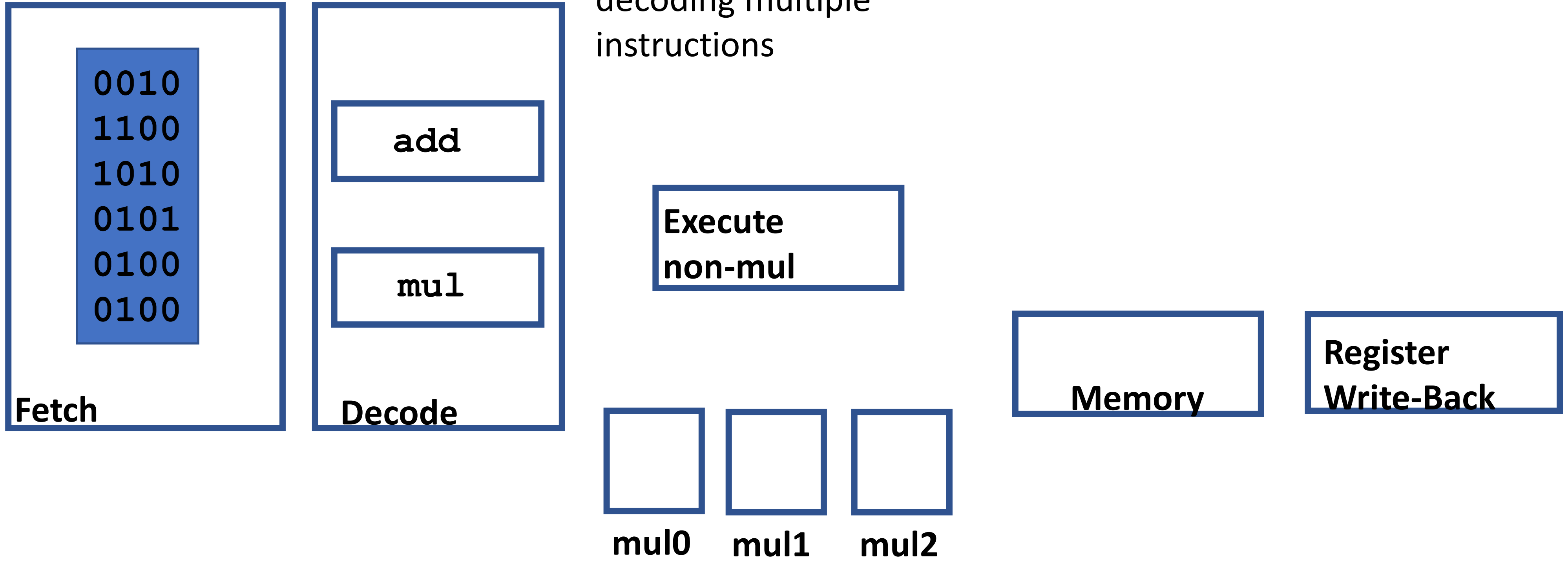
First idea: fetch a block of data from instruction memory instead of one instruction at a time.



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

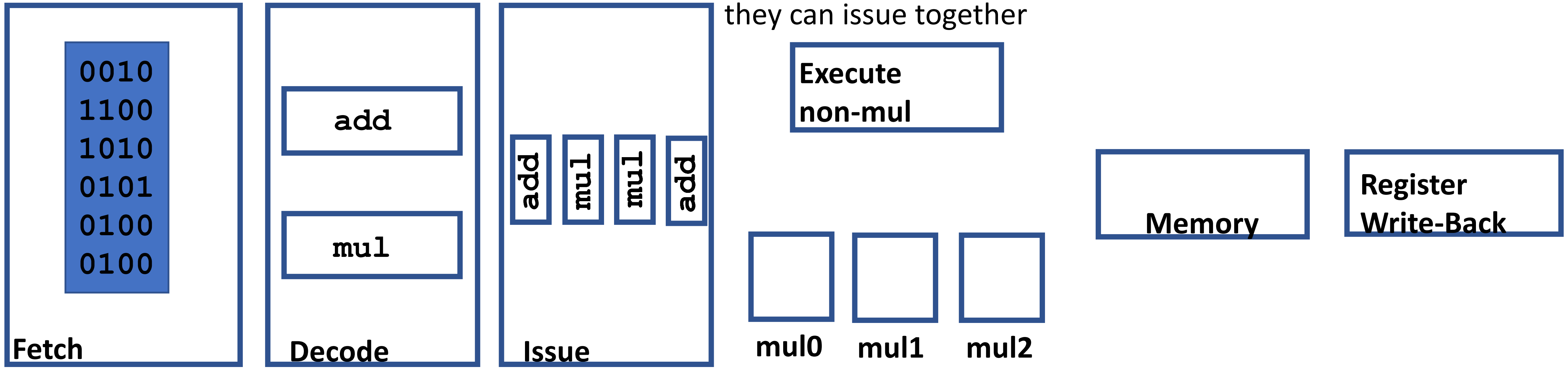
Superscalar processors

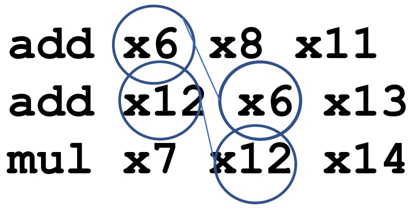
Second idea: Replicate decode logic to allow decoding multiple instructions



Superscalar processors

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





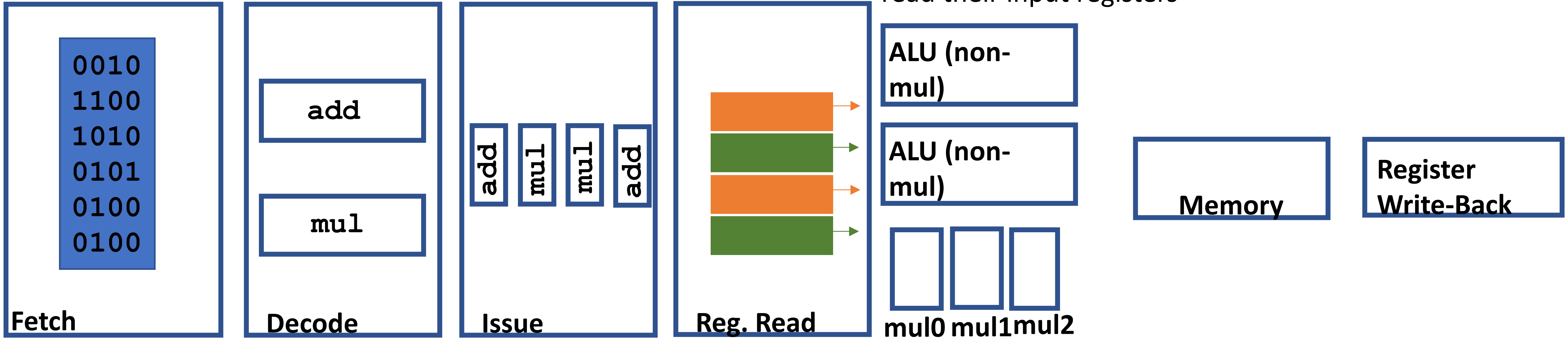
```
add  x6  x8  x11
add  x12 x6  x13
mul  x7  x12 x14
```

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

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

Superscalar processors

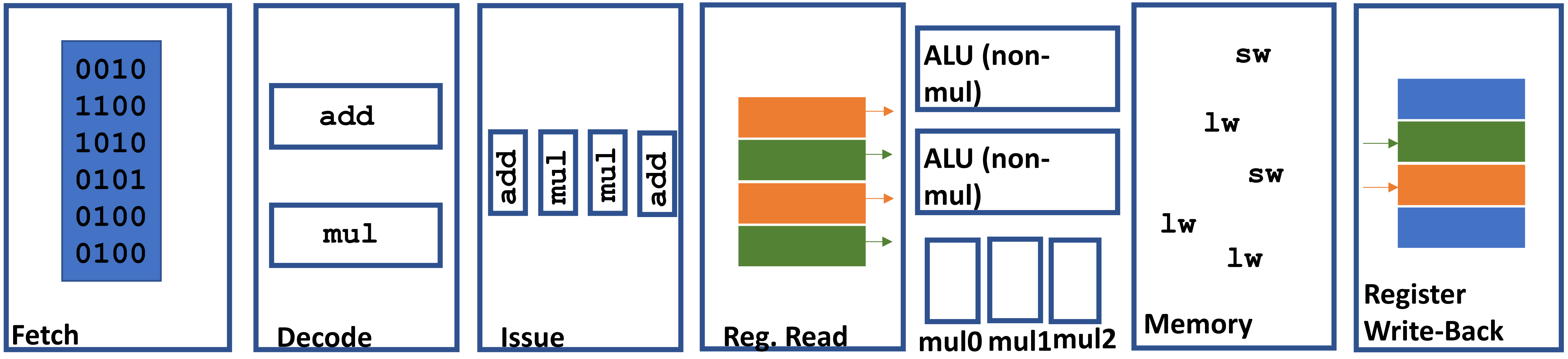
Fifth idea: Add ***multiple execute units*** to which to dispatch operations after they read their input registers



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

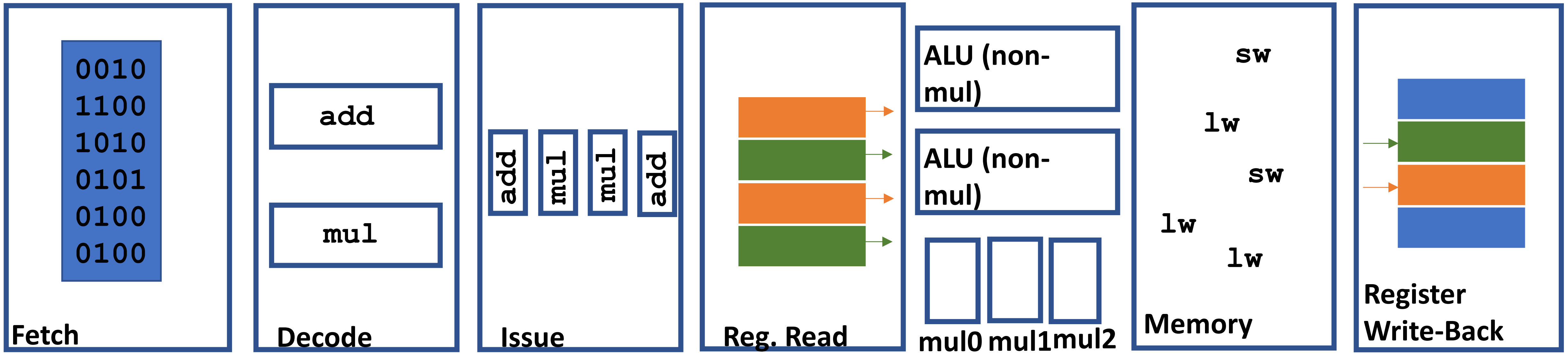
Superscalar processors

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



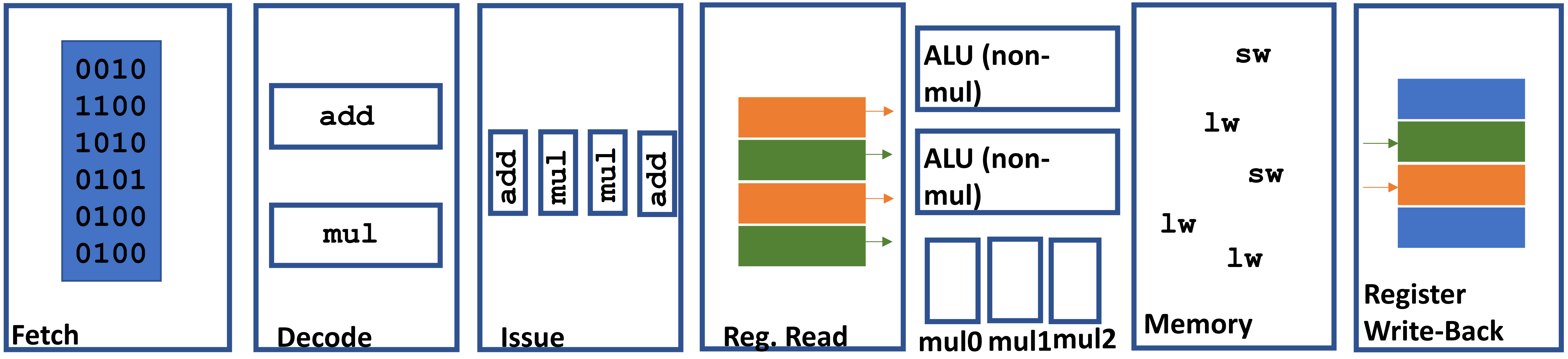
Sixth idea: Handle multiple outstanding memory operations in memory system (complex! we will mostly ignore this part)

Superscalar processors: Challenges & sources of complexity



Fetch:

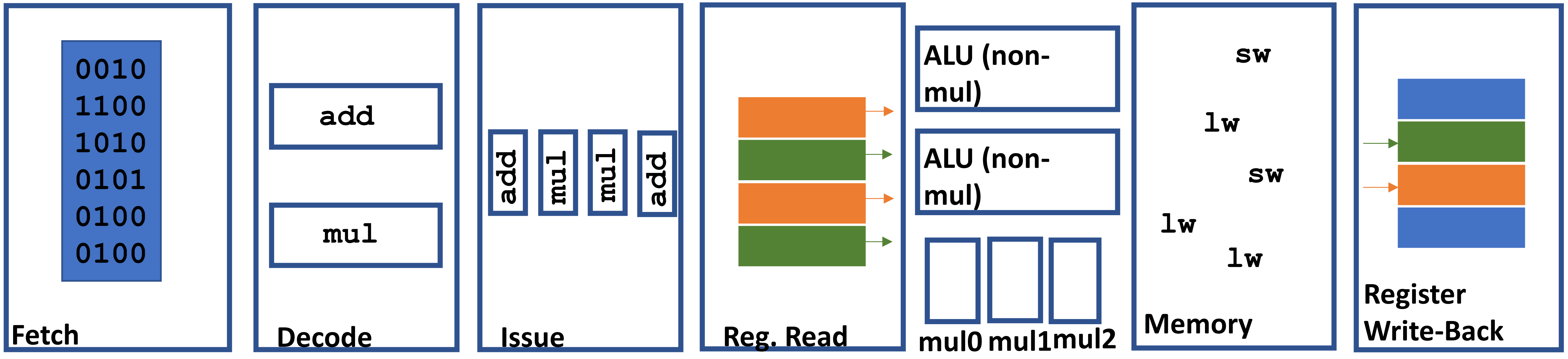
Superscalar processors: Challenges & sources of complexity



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

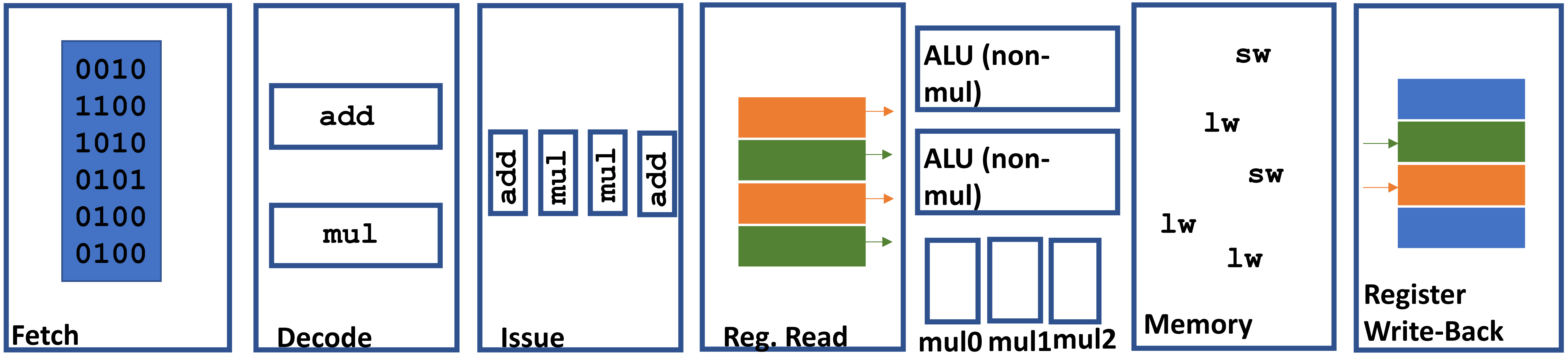
Decode:



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

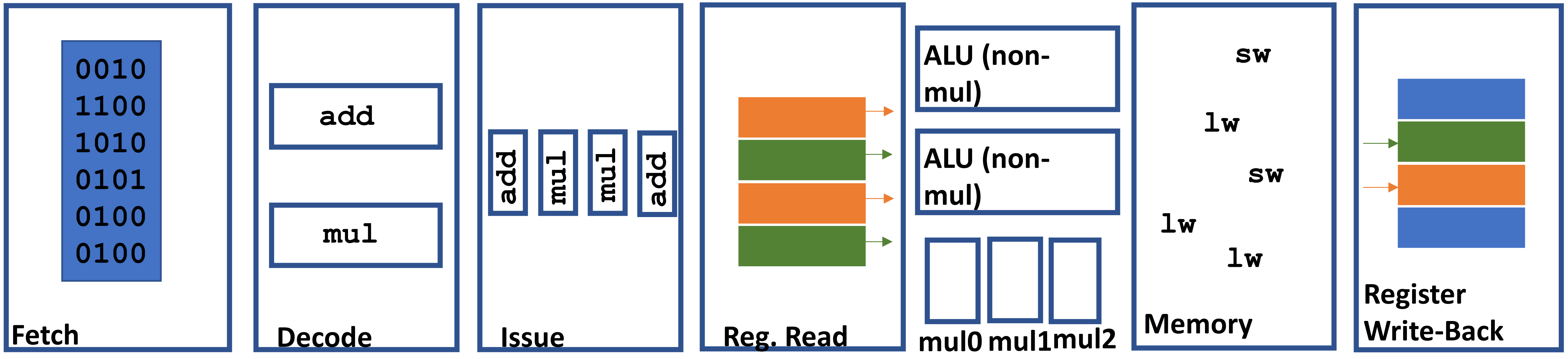
Decode: Not too bad, just replication of resources



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

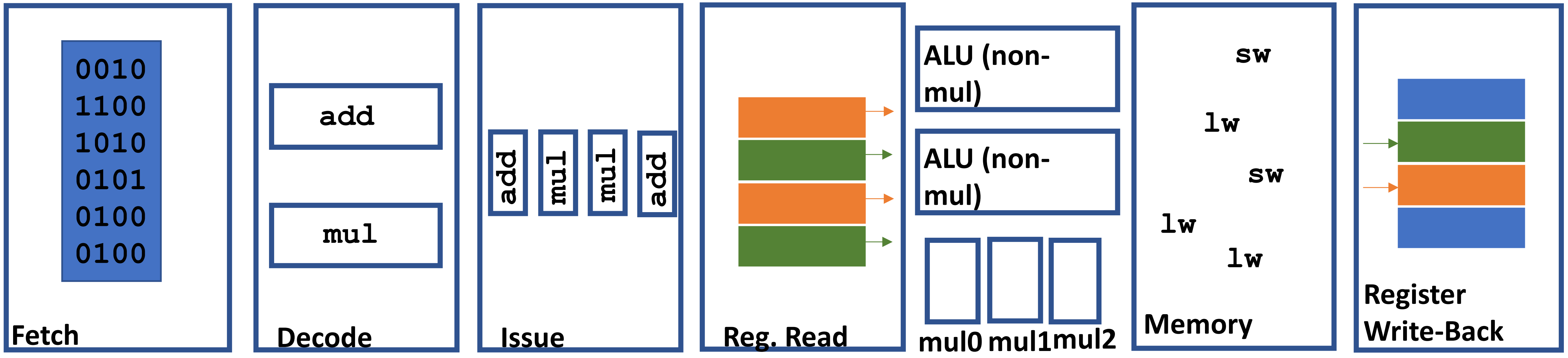


Fetch: Branch prediction more complex. Risk of *overfetch* because we're fetching a whole block? Must consider multiple, sequential fetches based on predictions

Issue:

Superscalar processors: Challenges & sources of complexity

Decode: Not too bad, just replication of resources



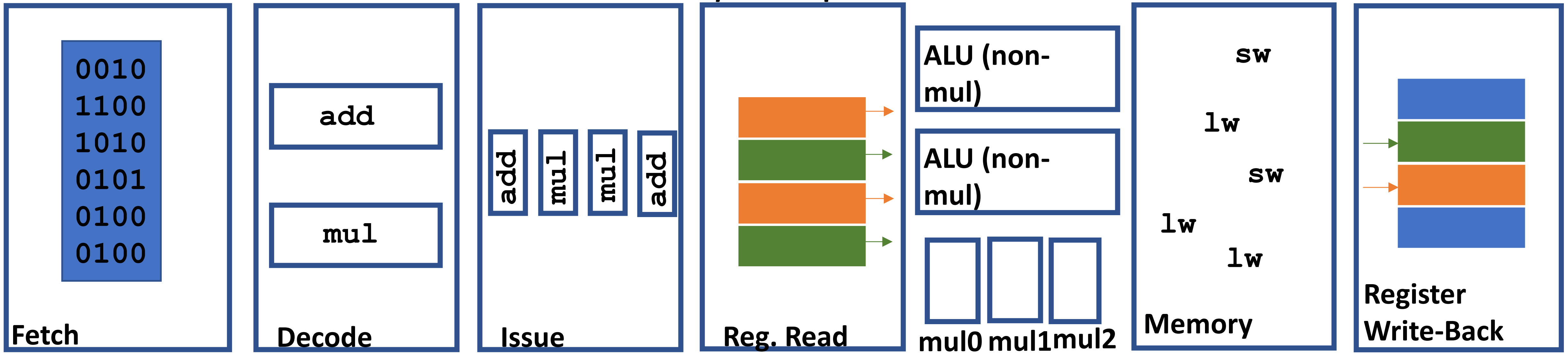
Fetch: Branch prediction more complex. Risk of *overfetch* because we're fetching a whole block? Must consider multiple, sequential fetches based on predictions

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



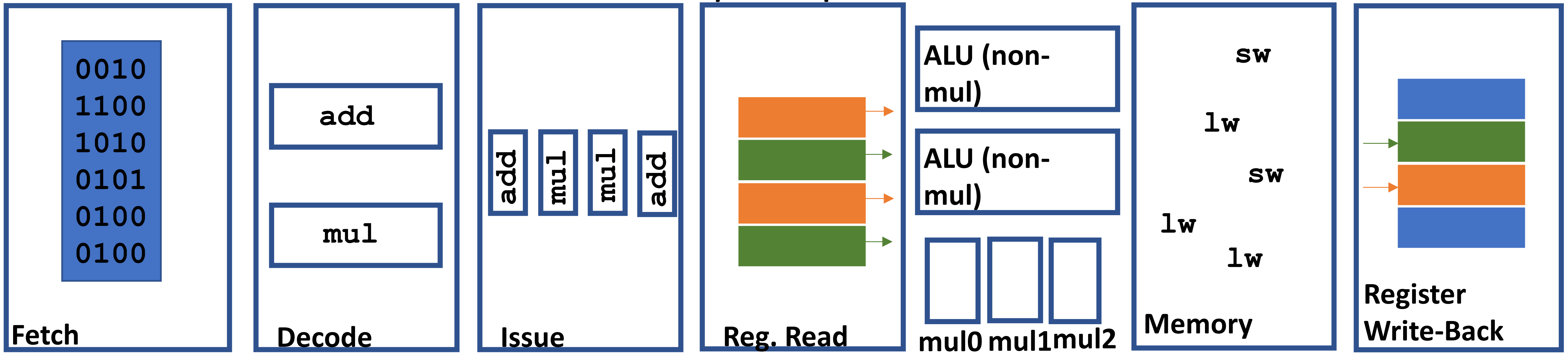
Fetch: Branch prediction more complex. Risk of *overfetch* because we're fetching a whole block? Must consider multiple, sequential fetches based on predictions

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

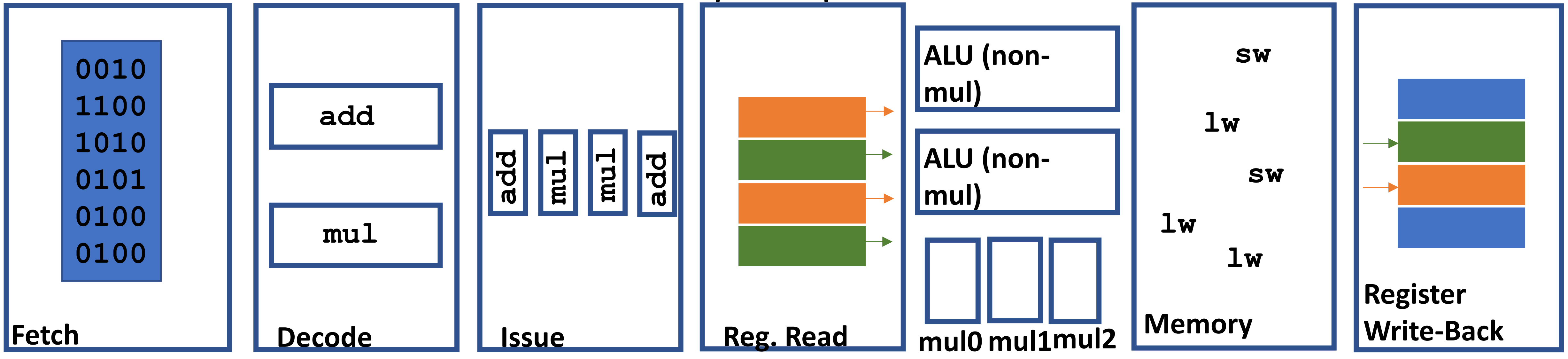
Issue: Dependence / hazard detection logic complexity. Need to detect dependences between all instructions in issue queue and some combinations of instructions cannot issue simultaneously

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

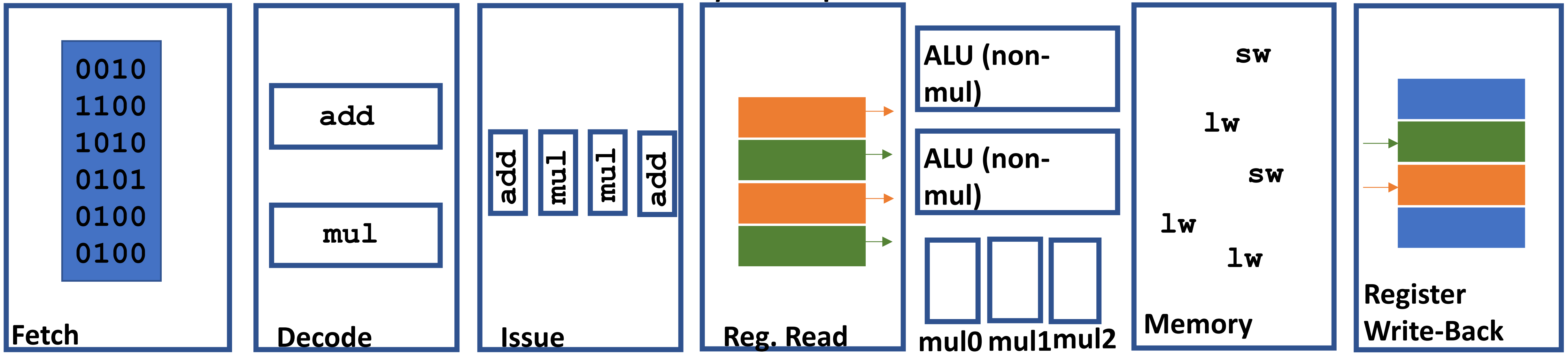
Execute / Memory: More execute units, more cache ports. Forwarding paths & input operand selection logic become very complicated.

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)

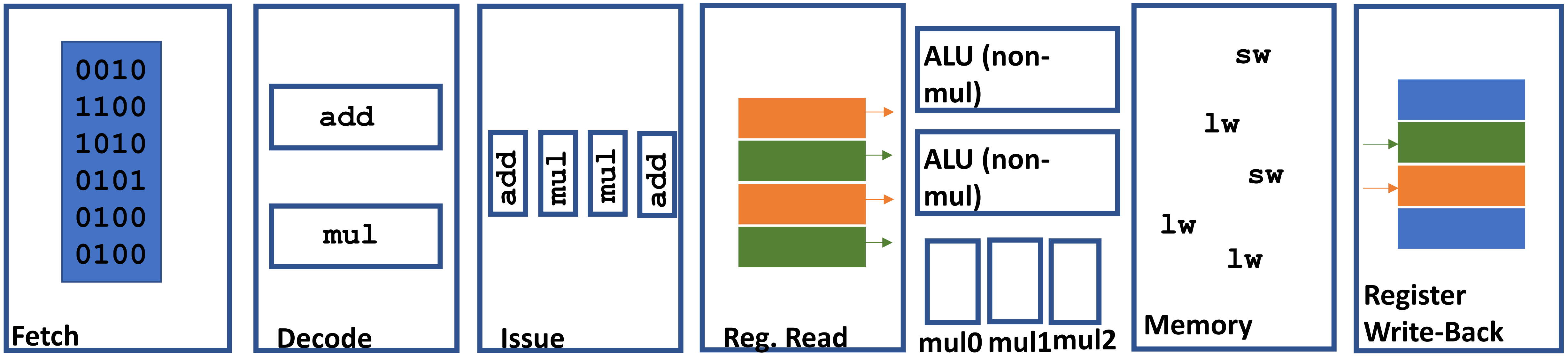


Fetch: Branch prediction more complex. Risk of *overfetch* because we're fetching a whole block? Must consider multiple, sequential fetches based on predictions

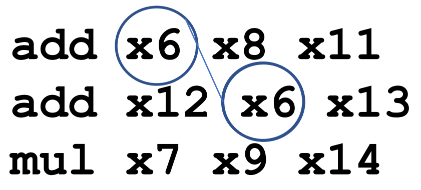
Issue: Dependence / hazard detection logic complexity. Need to detect dependences between all instructions in issue queue and some combinations of instructions cannot issue simultaneously

Execute / Memory: More execute units, more cache ports. Forwarding paths & input operand selection logic scale w/ square of insn window.

Remaining limits on performance of this processor?



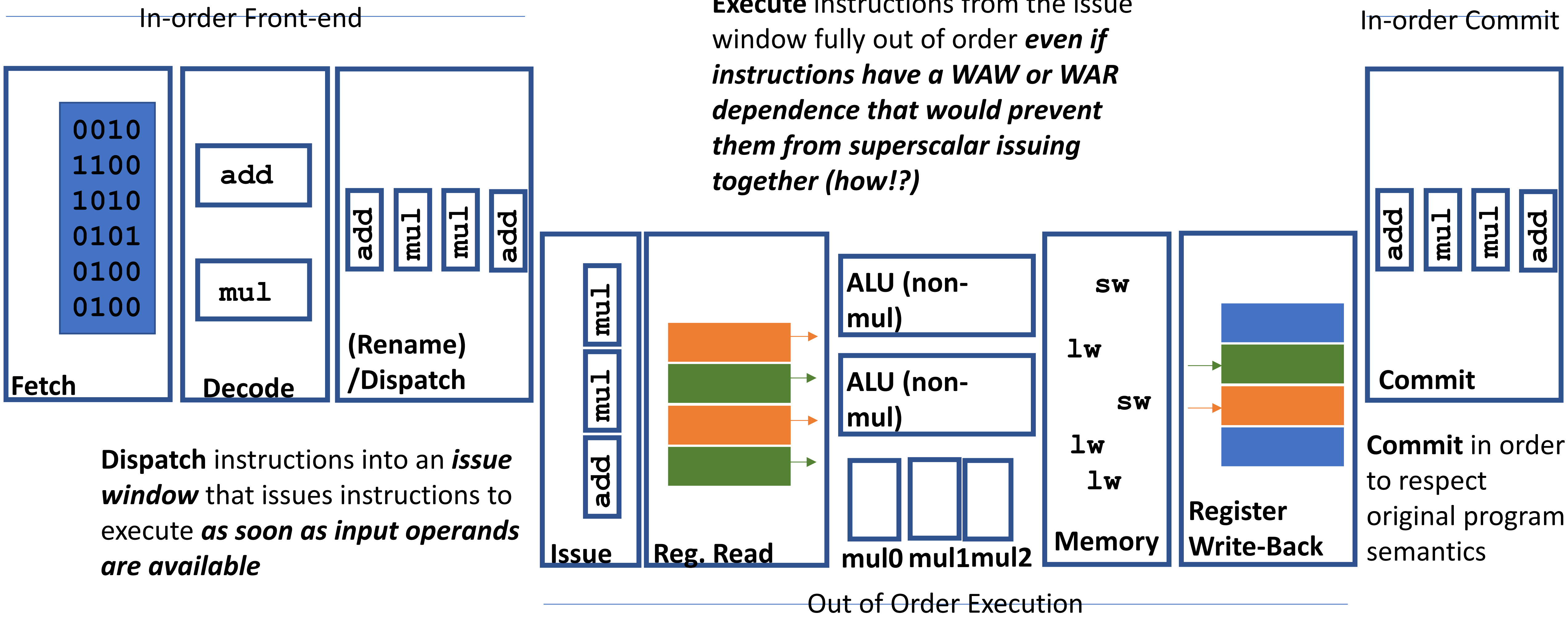
Application itself may not have ample ILP



```
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

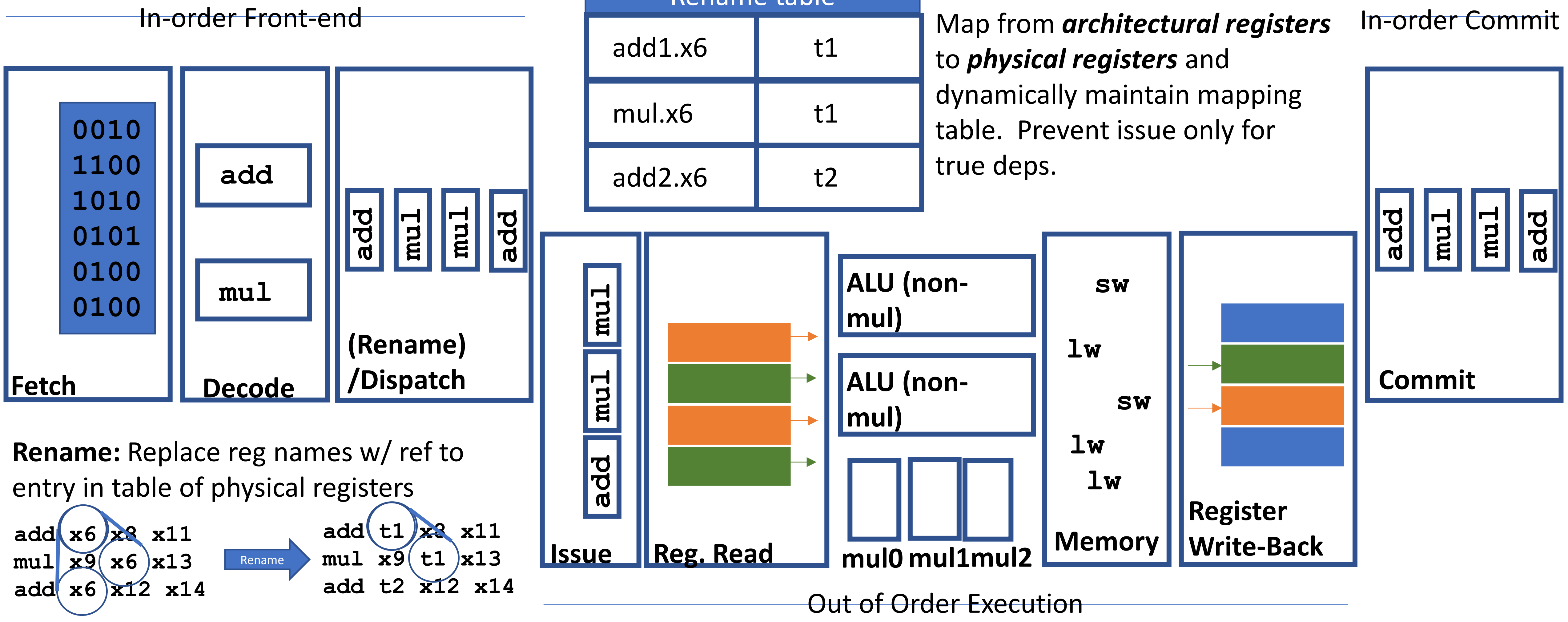


Execute instructions from the issue window fully out of order ***even if instructions have a WAW or WAR dependence that would prevent them from superscalar issuing together (how!?)***

Dispatch instructions into an ***issue window*** that issues instructions to execute ***as soon as input operands are available***

Commit in order to respect original program semantics

Register Renaming Resolves Dependences that Prevent Instructions from Executing Together



Rename table	
add1.x6	t1
mul.x6	t1
add2.x6	t2

Map from ***architectural registers*** to ***physical registers*** and dynamically maintain mapping table. Prevent issue only for true deps.

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

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

Rename

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

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

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

d

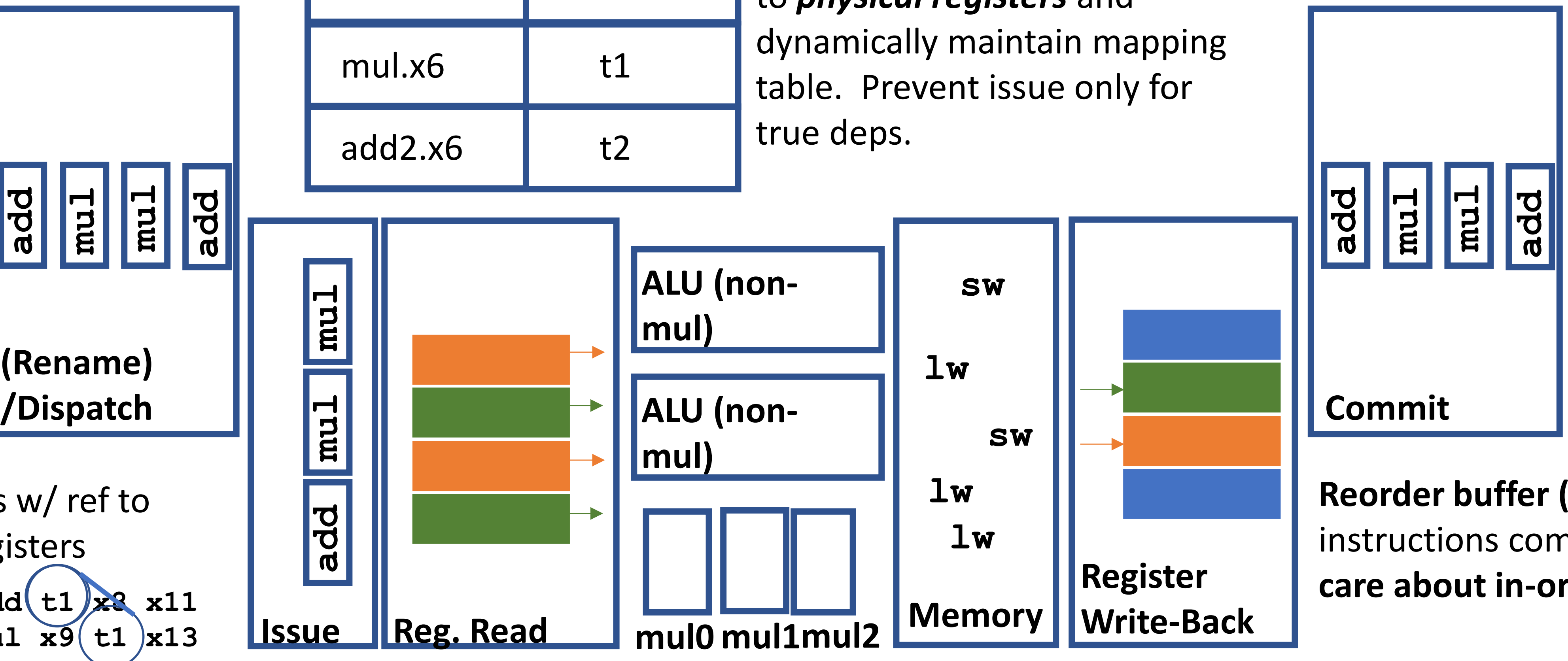
Rename table	
add1.x6	t1
mul.x6	t1
add2.x6	t2

Map from ***architectural registers*** to ***physical registers*** and dynamically maintain mapping table. Prevent issue only for true deps.

In-order Commit

Reorder Buffer		
add1	t1	17
mul	x9	???
add2	t2	245

add2 is complete, but waits to update t2 (i.e., x6) until mul is done



Reorder buffer (ROB) ensures instructions commit in order. **Why do we care about in-order commit?**

s w/ ref to
gisters

```
dd t1 x8 x11
ul x9 t1 x13
dd t2 x12 x14
```

Out of Order Execution

reserve RAW (why?)

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 x1
```

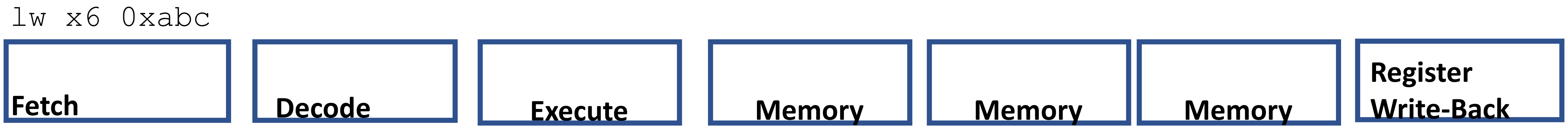
Write-After-Write (WAW)

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
```

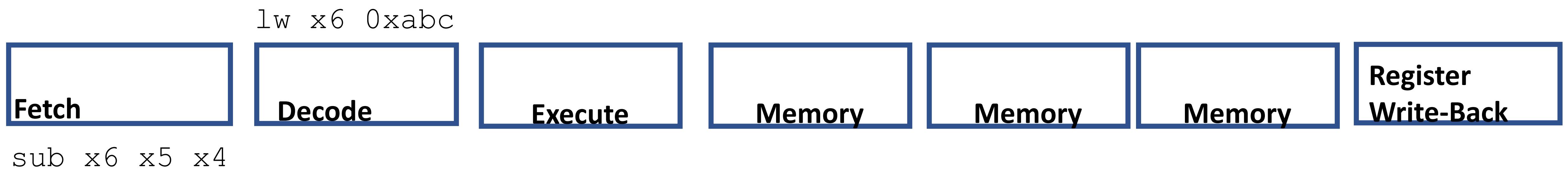
Write-After-Write (WAW)



Types of Data Hazards

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

Write-After-Write (WAW)



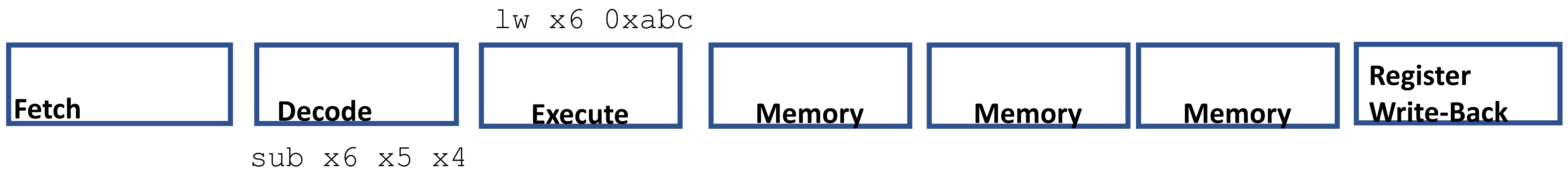
```
lw x6 0xabc
```

```
sub x6 x5 x4
```

Types of Data Hazards

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

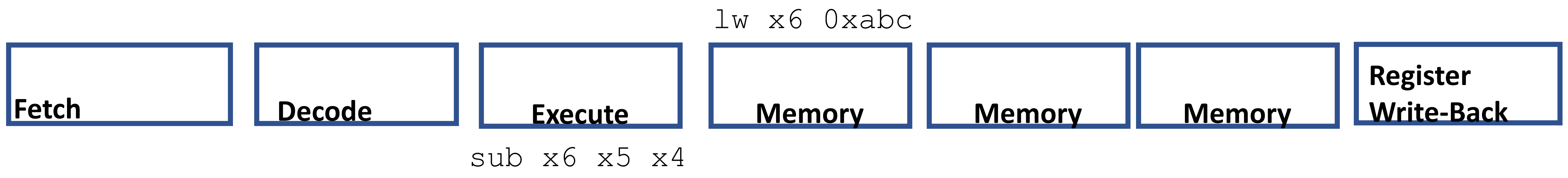
Write-After-Write (WAW)



Types of Data Hazards

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

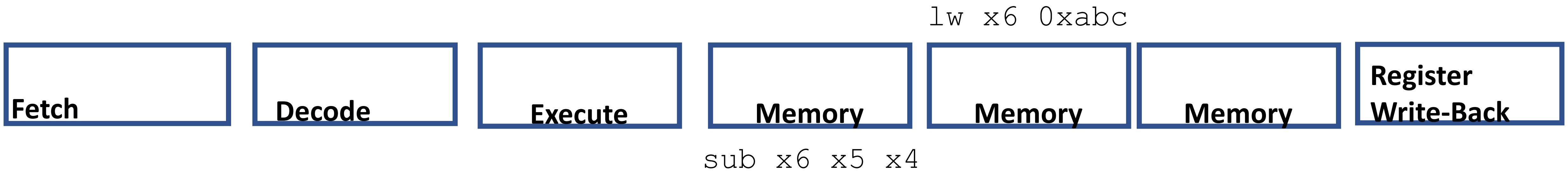
Write-After-Write (WAW)



Types of Data Hazards

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

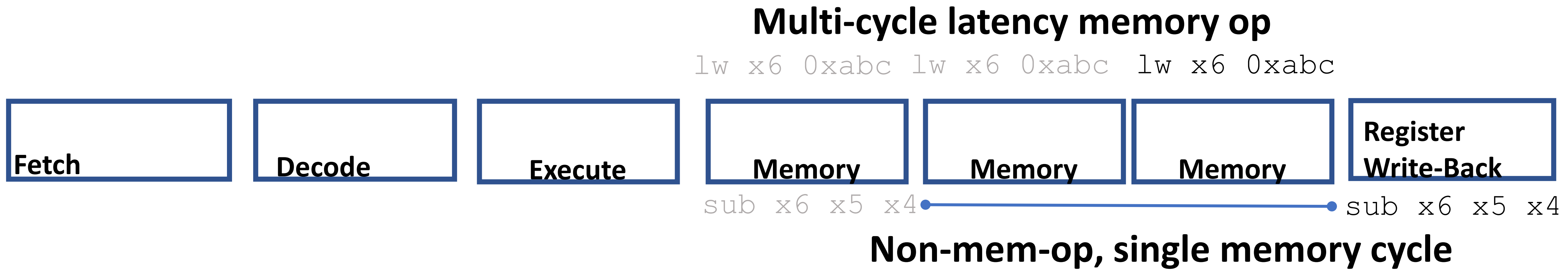
Write-After-Write (WAW)



Types of Data Hazards

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

Write-After-Write (WAW)



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

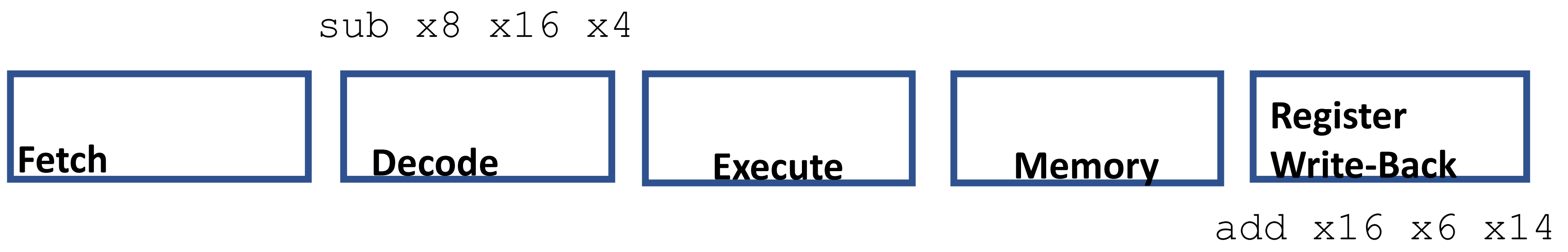
Explicitly handled with logic to maintain ordering in processors that allow this behavior (not our datapath)

Types of Data Hazards

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

Write-After-Read (WAR)

Stalled at decode/reg. read

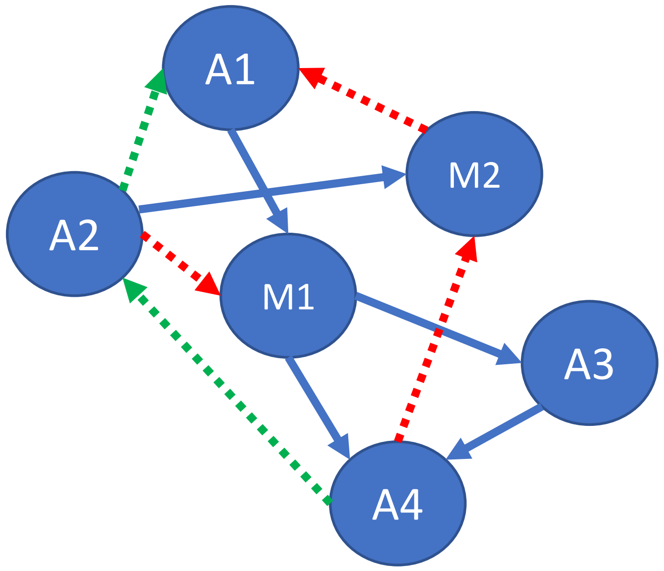


Completes quickly and writes reg.

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

Renaming Example

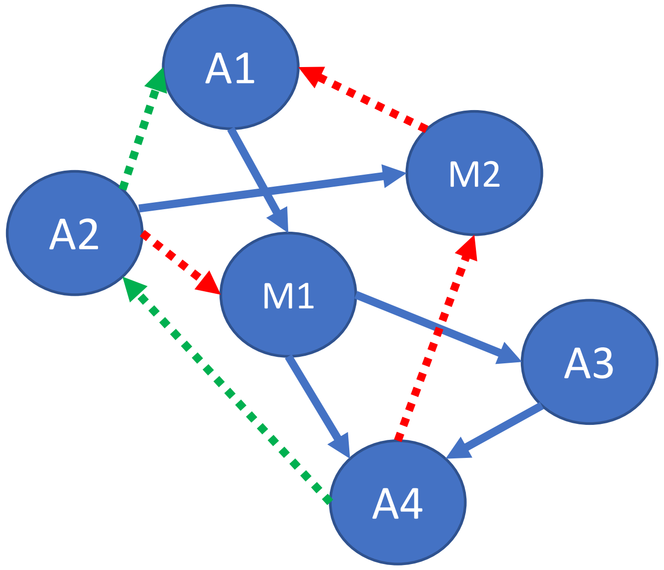
```
A1: add x6 x8 x11
M1: mul x9 x6 x13
A2: add x6 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```



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

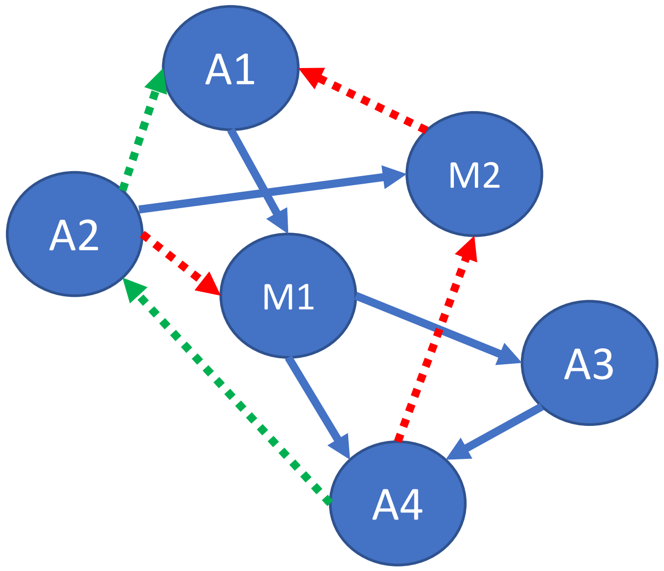
```
A1: add x6 x8 x11
M1: mul x9 x6 x13
A2: add x6 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```



```
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
```

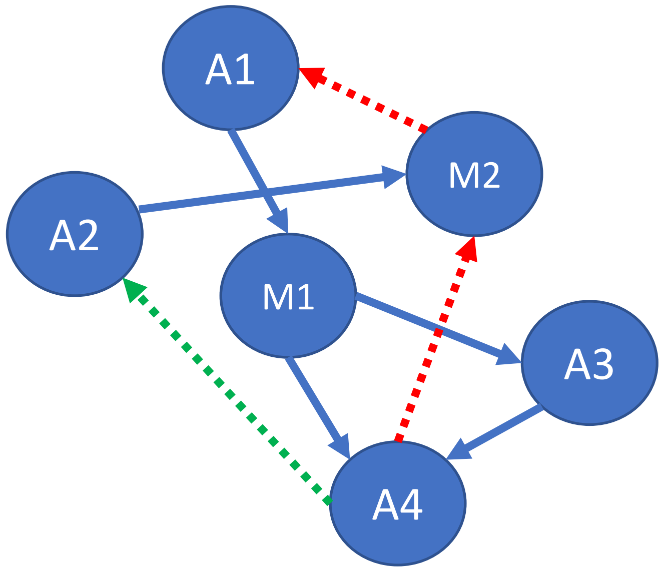


RAW dependence on x6
M1 waiting on result from A1 (r0)

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

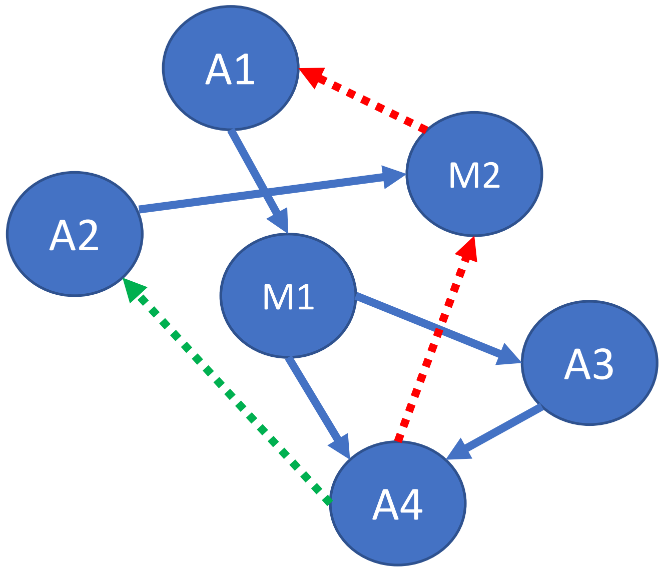


```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
A2.x6 -> r2
```

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
```



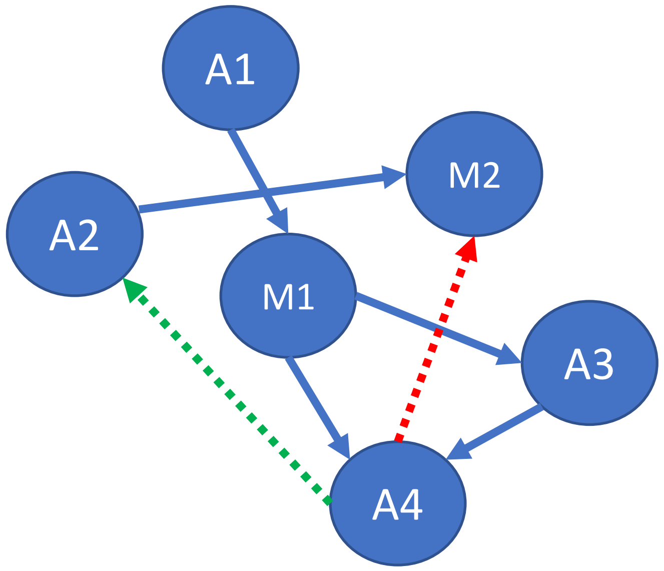
RAW dependence between M1 & A3
Cannot be resolved by renaming

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

Renaming Example

```
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
```

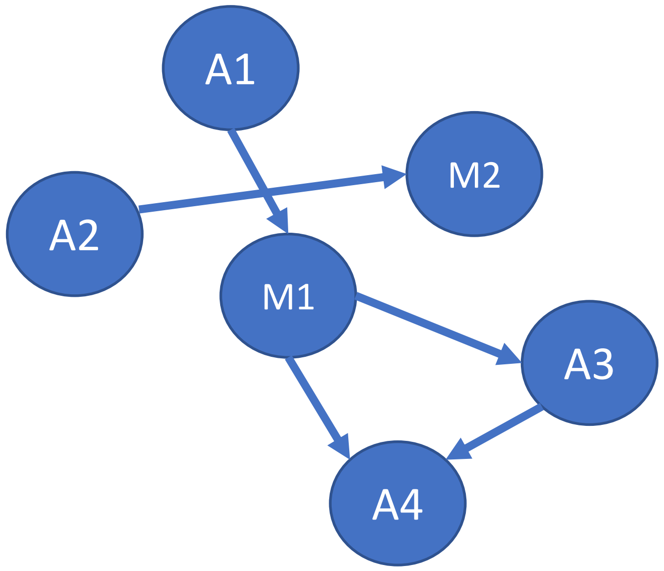
```
A1: add x6 x8 x11
M1: mul x9 x6 x13
A2: add x6 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```



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
```



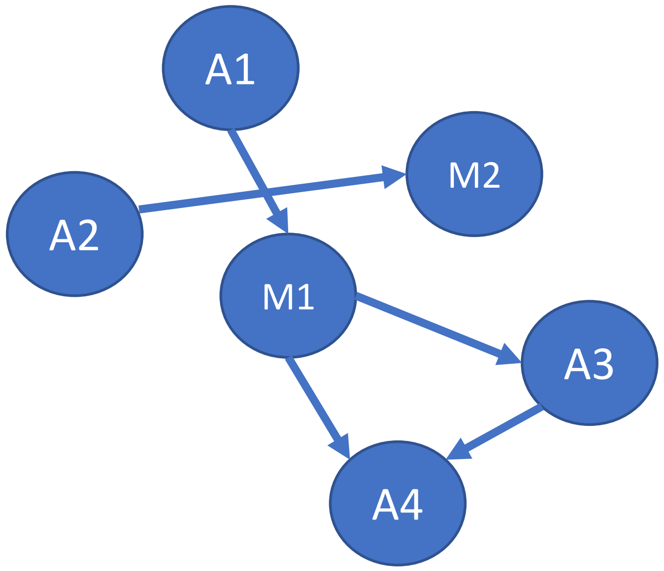
```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
A2.x6 -> r2
A3.x7 -> r3
A3.x9 <- r1
M2.x8 -> r4
M2.x6 <- r2
A4.x6 -> r5
A4.x7 <- r3
A4.x9 <- r1
```

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
```



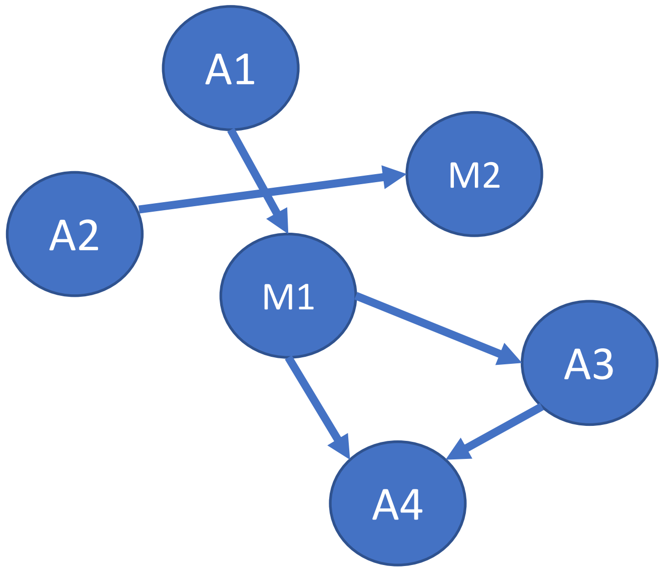
```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
A2.x6 -> r2
A3.x7 -> r3
A3.x9 <- r1
M2.x8 -> r4
M2.x6 <- r2
A4.x6 -> r5
A4.x7 <- r3
A4.x9 <- r1
```

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

Renaming Example

```
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
```

```
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
```



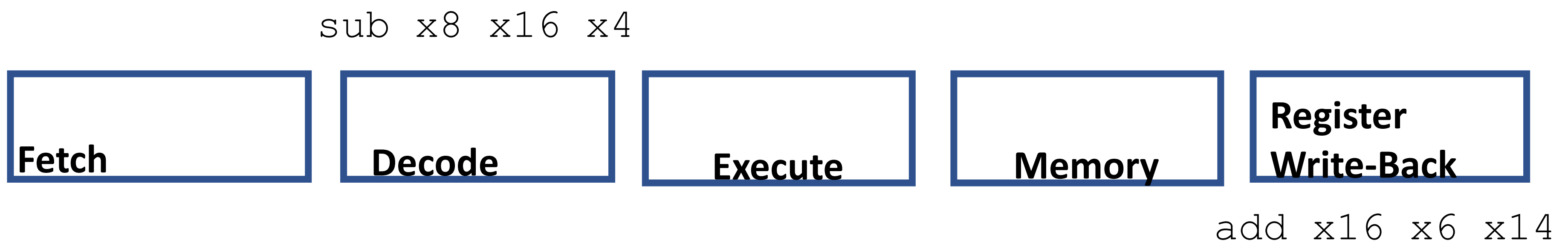
After register renaming, only RAW dependences (i.e., "True Dependences") remain in the execution

Renaming Avoids False Deps

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

Write-After-Read (WAR)

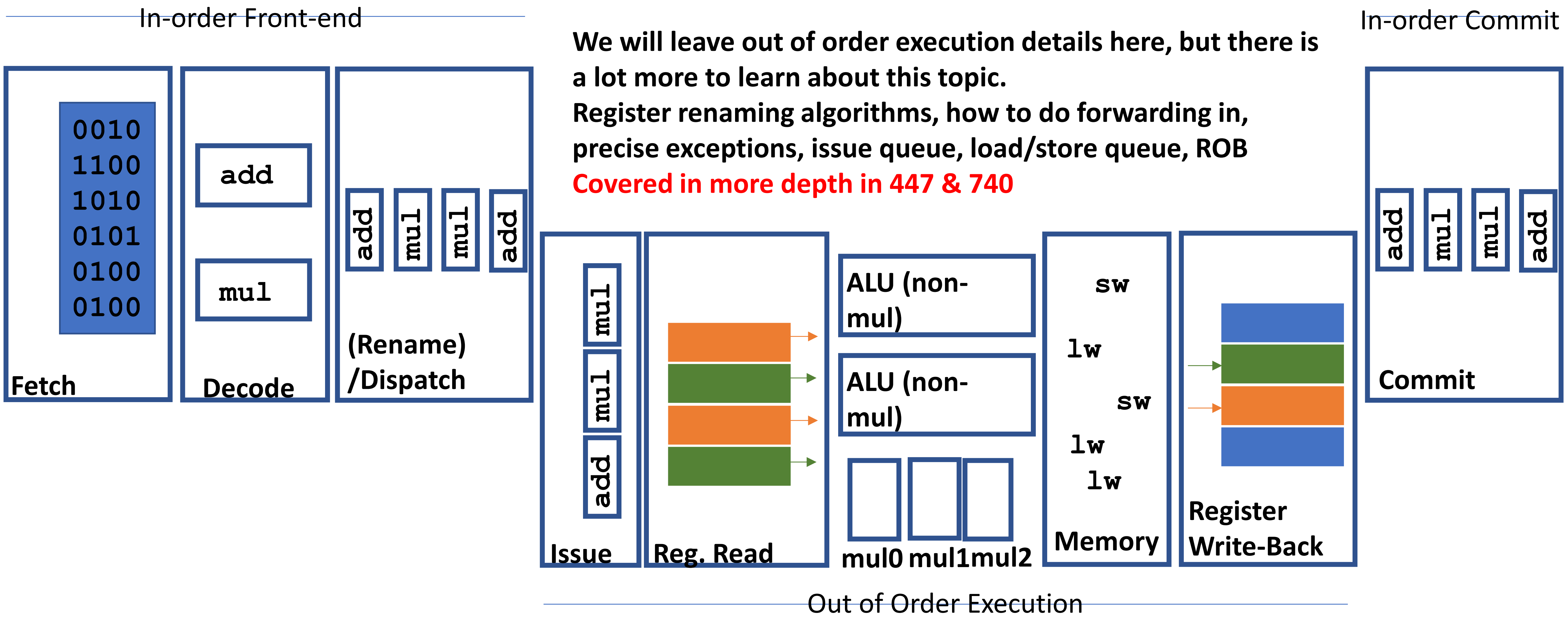
Stalled at decode/reg. read



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*



We will leave out of order execution details here, but there is a lot more to learn about this topic.
Register renaming algorithms, how to do forwarding in, precise exceptions, issue queue, load/store queue, ROB
Covered in more depth in 447 & 740

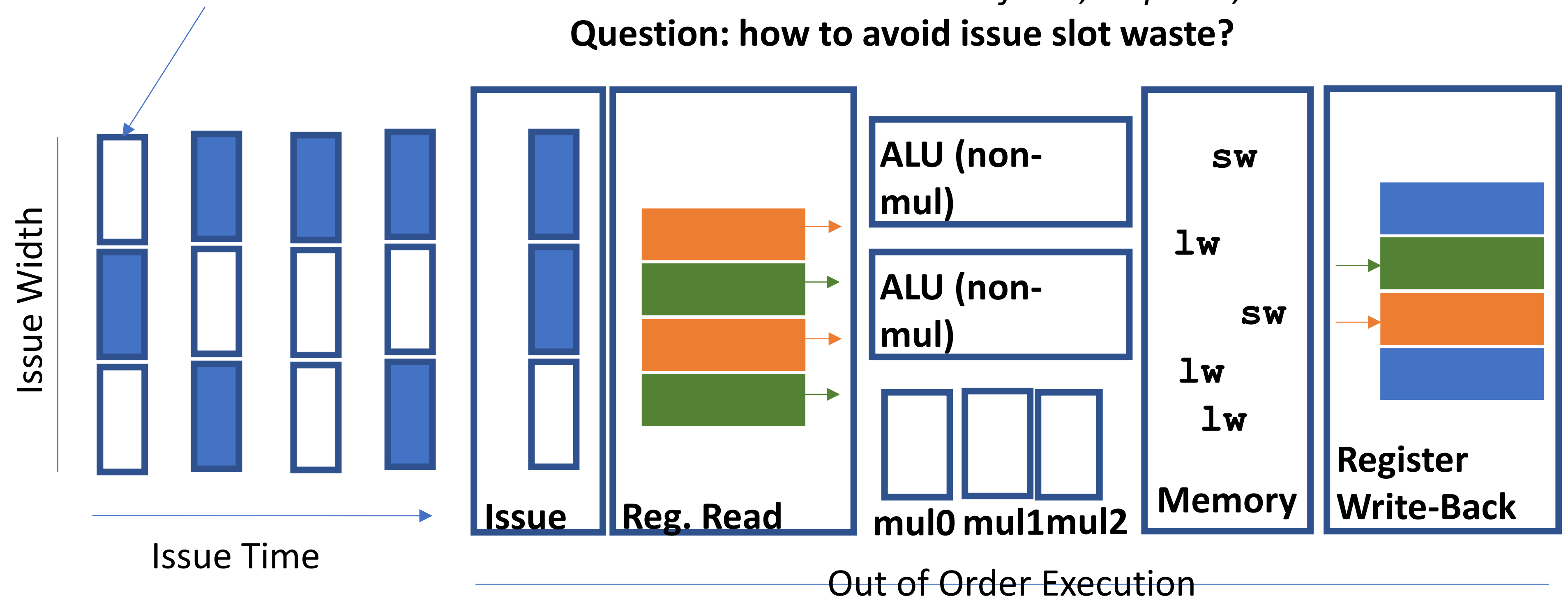
Scheduling Techniques to Maximize ILP

Superscalar execution exploits ILP to increase IPC

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?

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

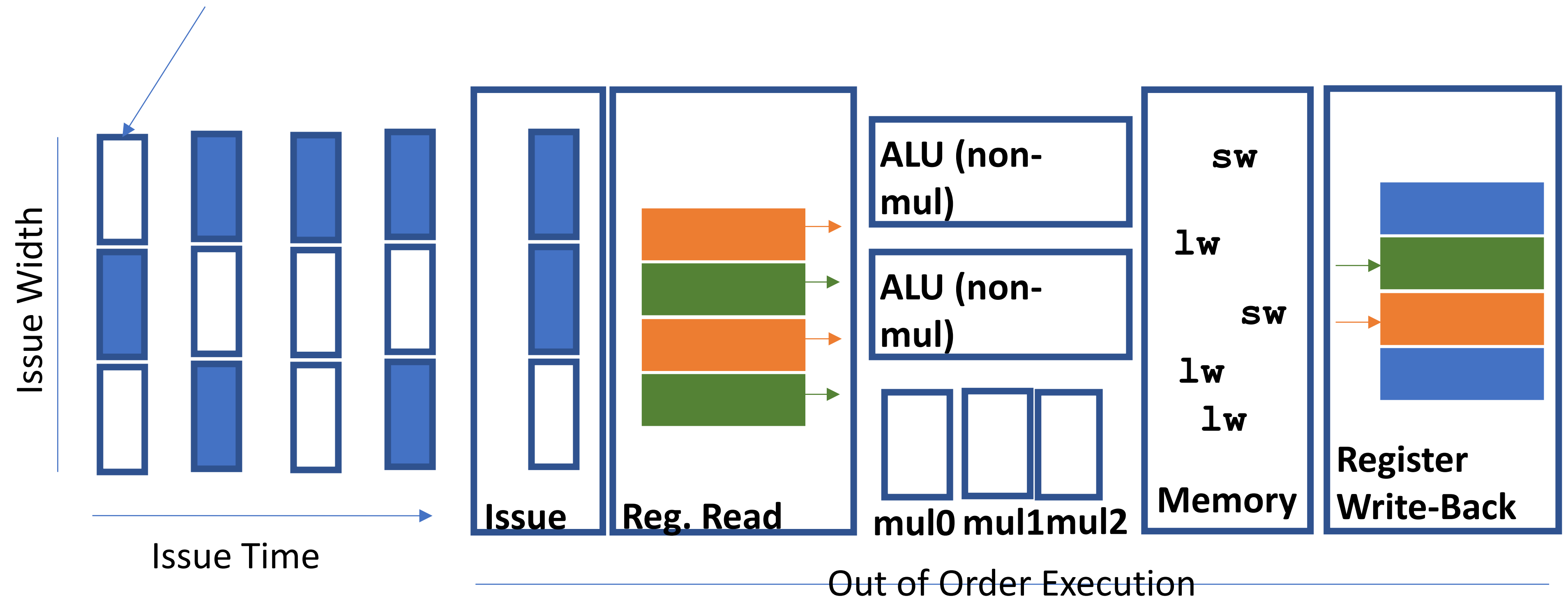


Superscalar execution exploits ILP to increase IPC

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.

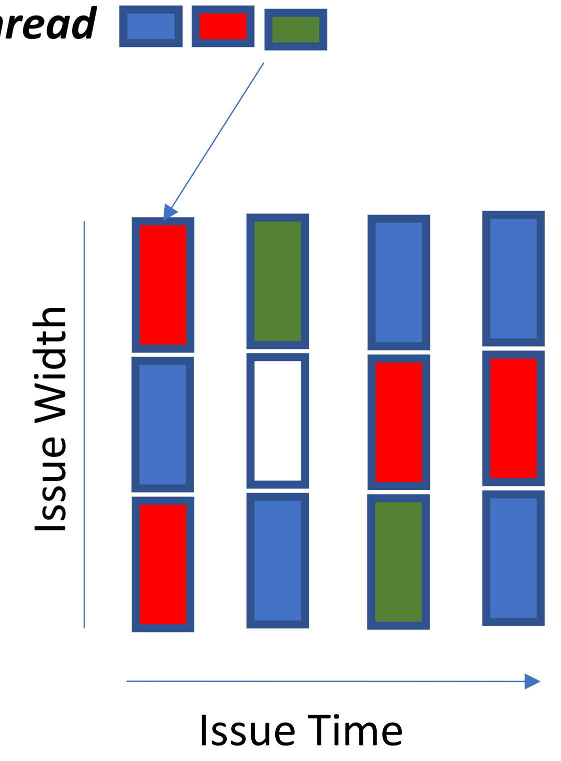
Susan Eggers, inventor of SMT, ca. 1980

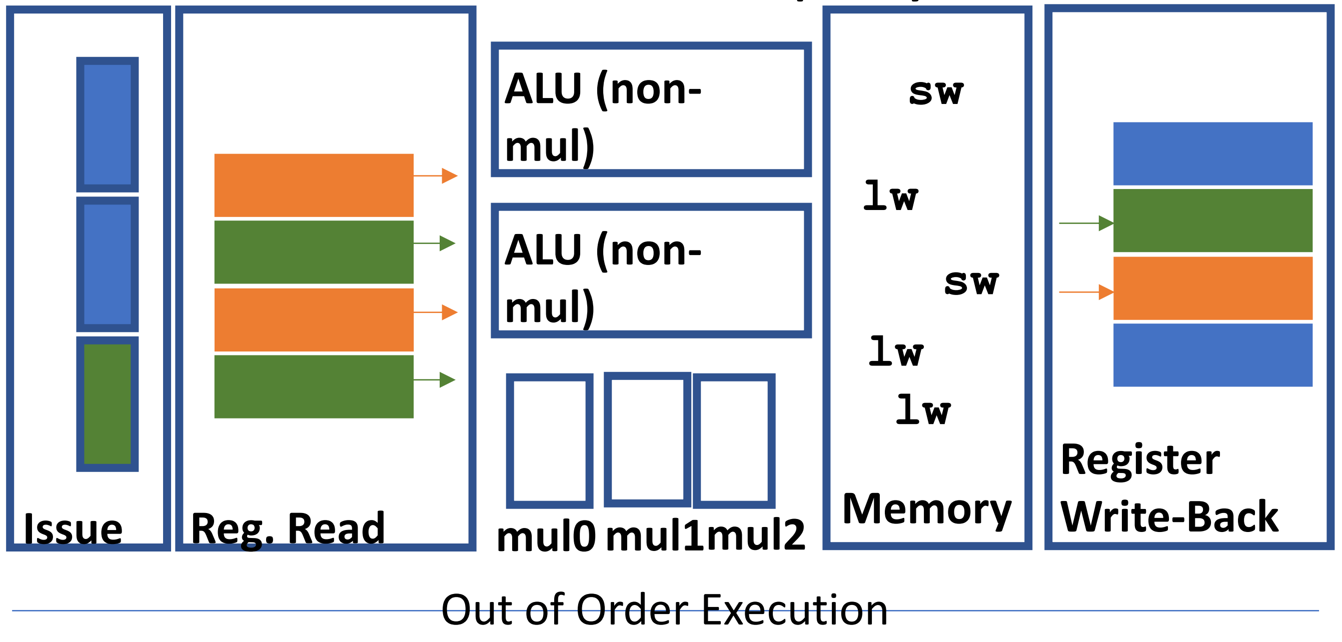
SMT exploits thread-level parallelism (TLP) instead of ILP to increase a machine’s useful IPC.

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

Question: Sources of hardware complexity for SMT?

Fill empty issue slots with instructions from another ***thread*** 🟦 🟥 🟩



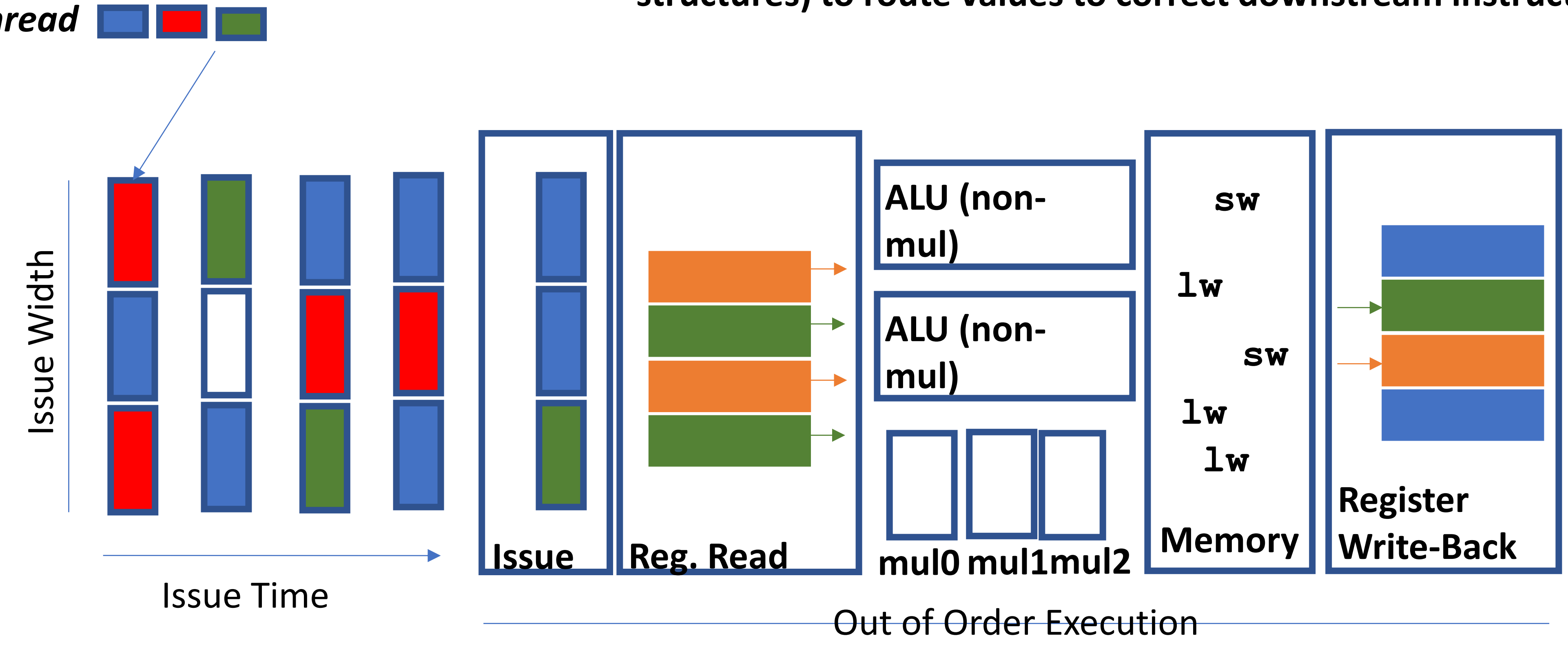


Simultaneous Multi-Threading (SMT)

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

Question: Sources of hardware complexity for SMT?

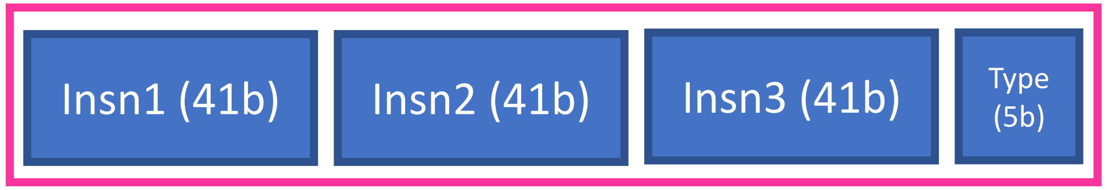
- **Need fetch to support multiple streams (including branch prediction logic…)**
- **Need to tag functional units, rename table entries, ROB entries (and other structures) to route values to correct downstream instructions**



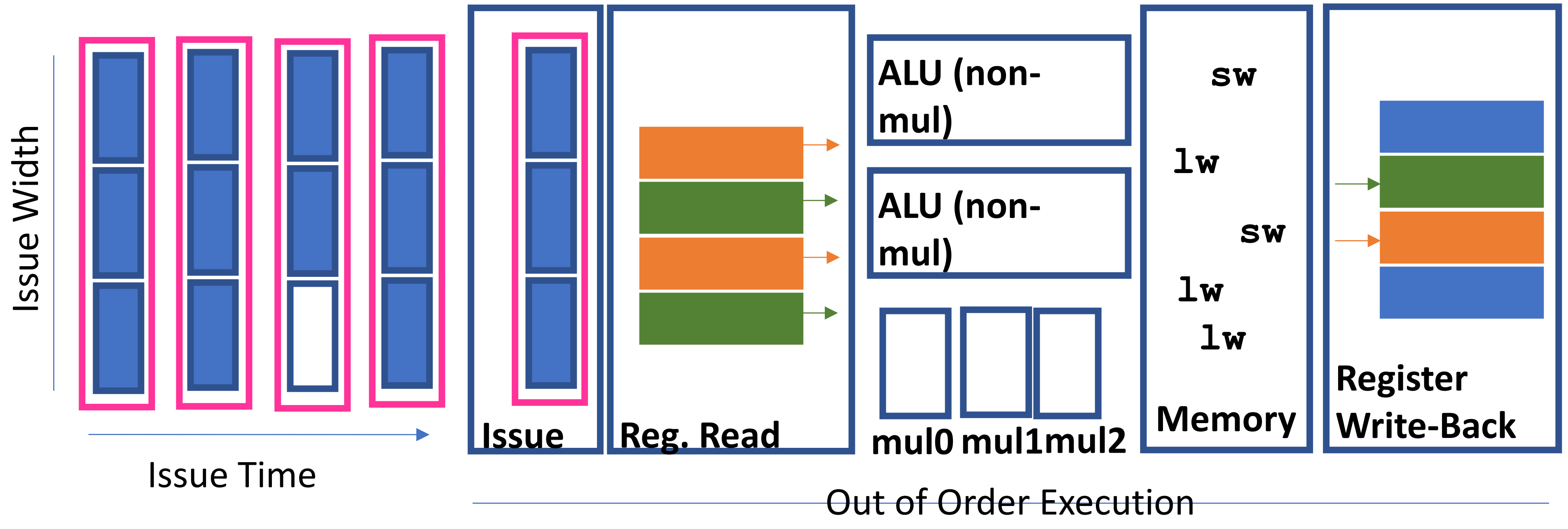
Very Large Instruction Word (VLIW) Architectures

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