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

Recap: Advanced Microarchitecture Techniques

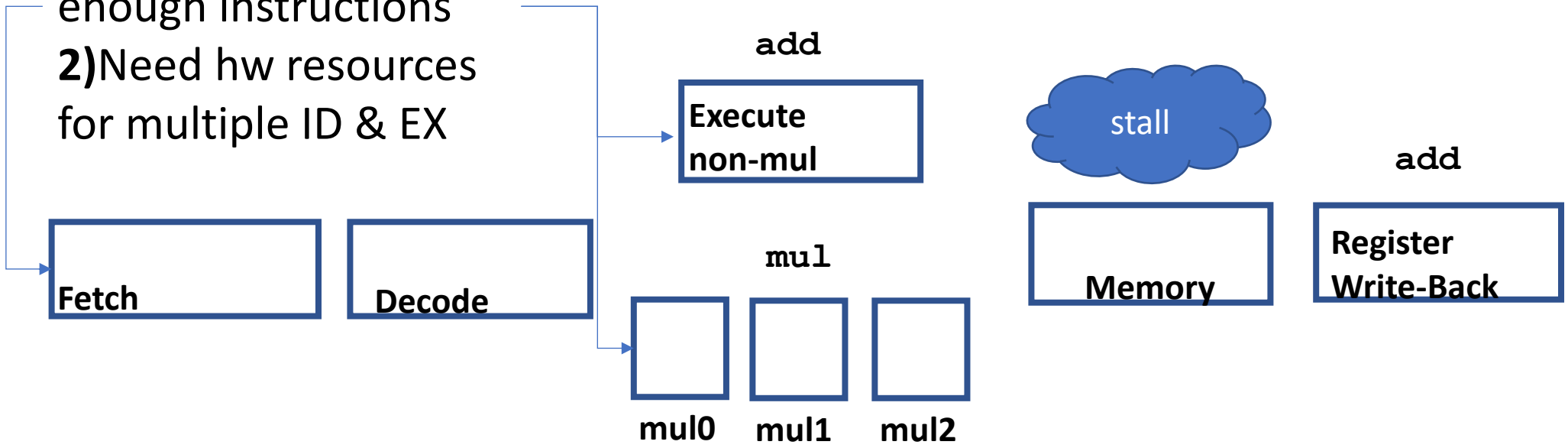
- Advanced Instruction-Level Parallelism: Deep pipelining, Multiple Issue, and Out of Order Execution
- Vector / SIMD processors

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

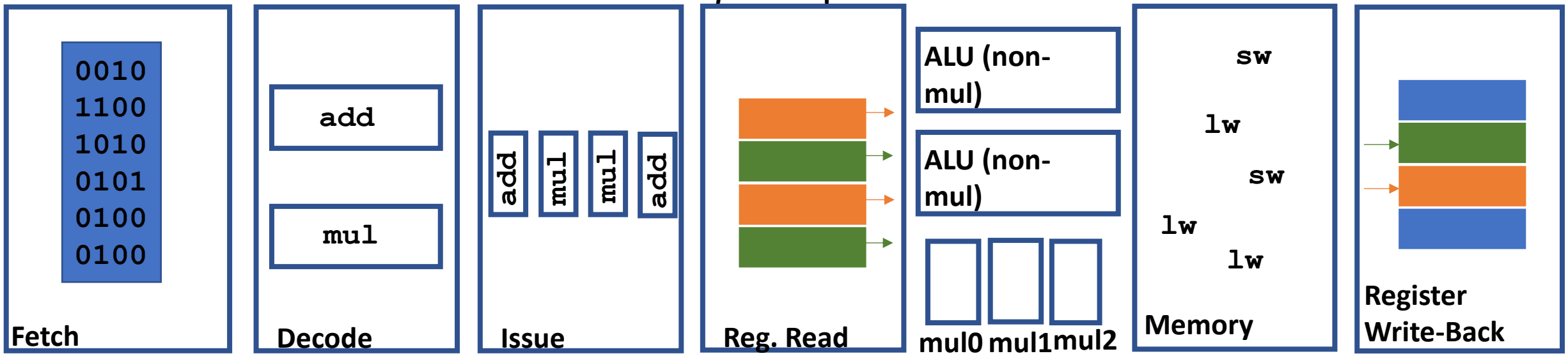
Superscalar executes multiple instructions at a time

Superscalar processors: 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)

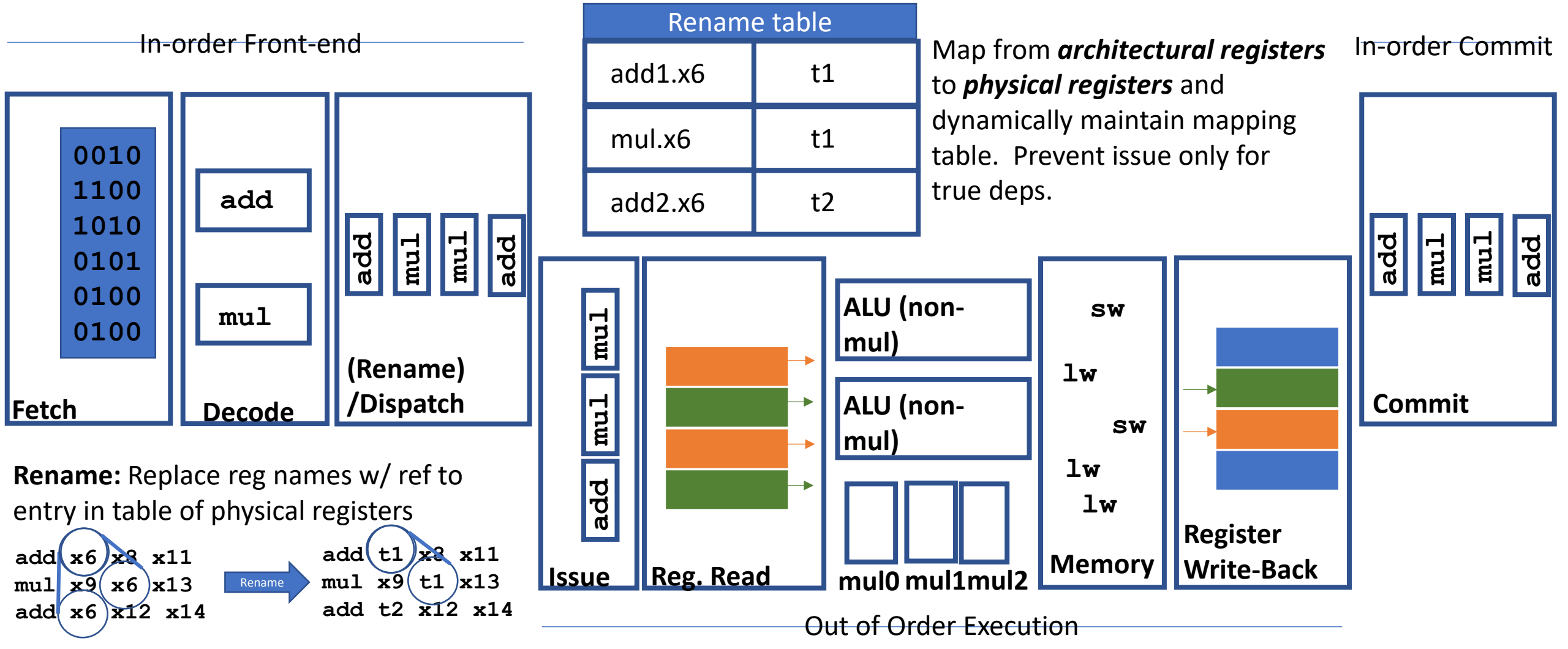


Fetch: Branch prediction more complex. Risk of *overfetch* because we're fetching a whole block? Must consider multiple, sequential fetches based on predictions

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.

Out-of-Order Execution: Register Renaming Eliminates Dependences that Prevent Simultaneous Dispatch



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?)

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

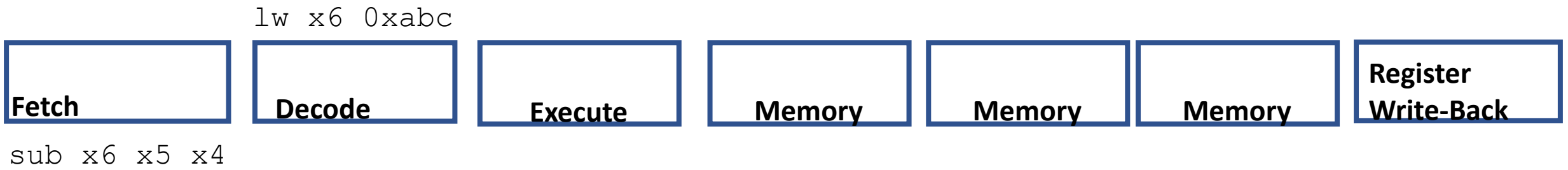
```
lw   x6 0xabc
sub  x6 x5 x4
add  x12 x6 x14
```

Write-After-Write (WAW)

Types of Data Hazards

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

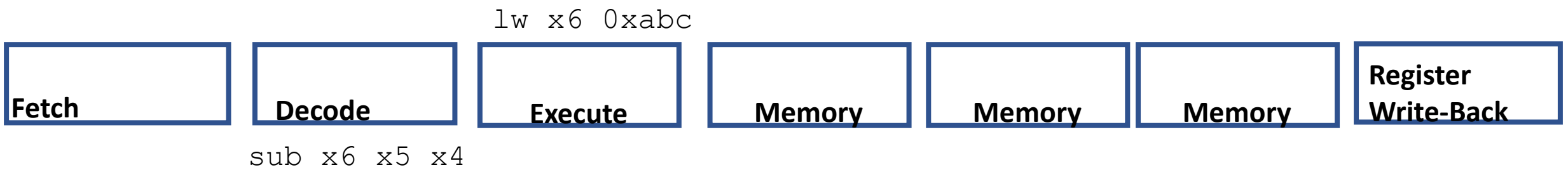
Write-After-Write (WAW)



Types of Data Hazards

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

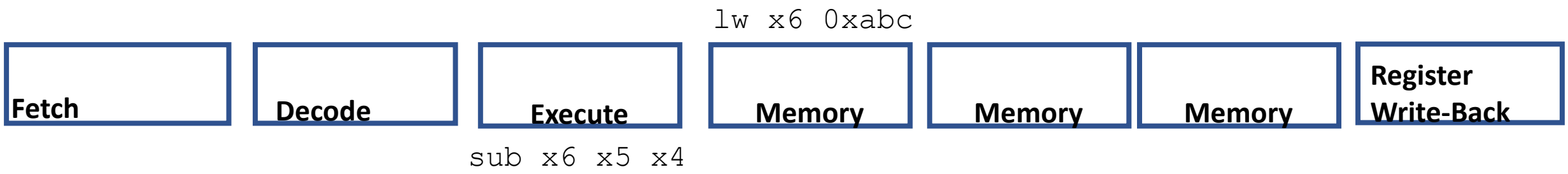
Write-After-Write (WAW)



Types of Data Hazards

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

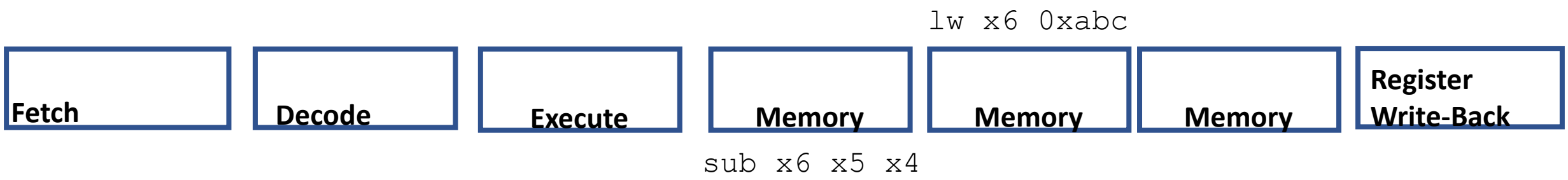
Write-After-Write (WAW)



Types of Data Hazards

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

Write-After-Write (WAW)

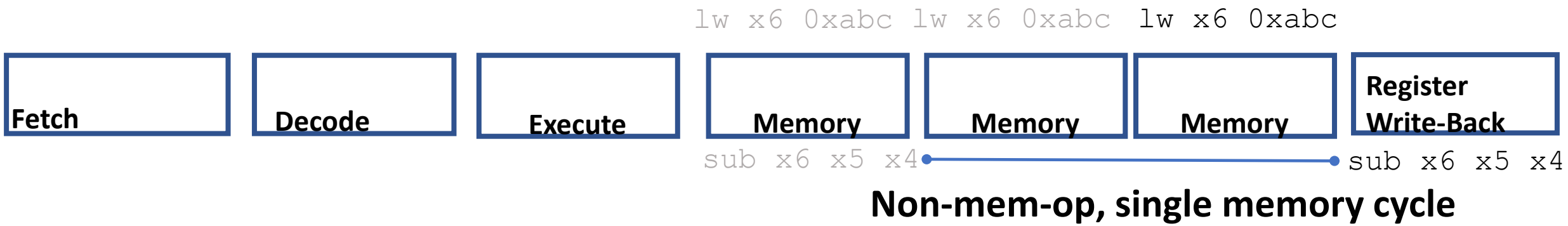


Types of Data Hazards

```
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`.

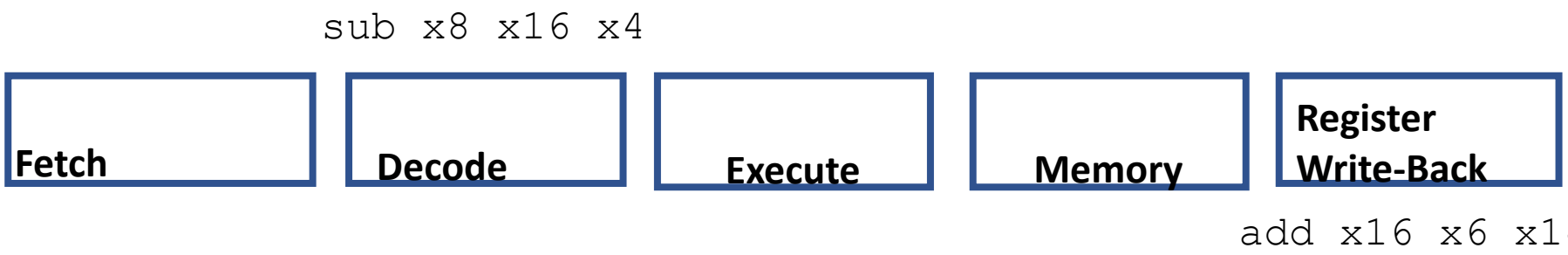
Explicitly handled with logic to maintain ordering in processors that allow this behavior (not our datapath)

Types of Data Hazards

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

Write-After-Read (WAR)

Stalled at decode/reg. read

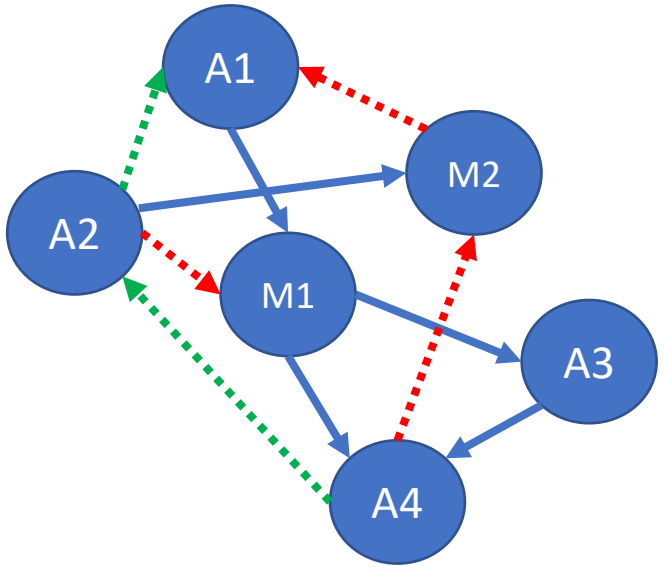


Completes quickly and writes reg.

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

Renaming Example

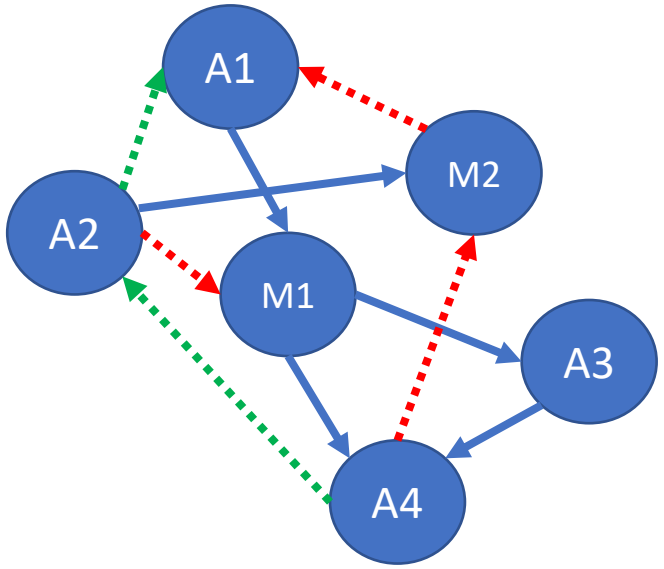
```
A1: add x6 x8 x11
M1: mul x9 x6 x13
A2: add x6 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```



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

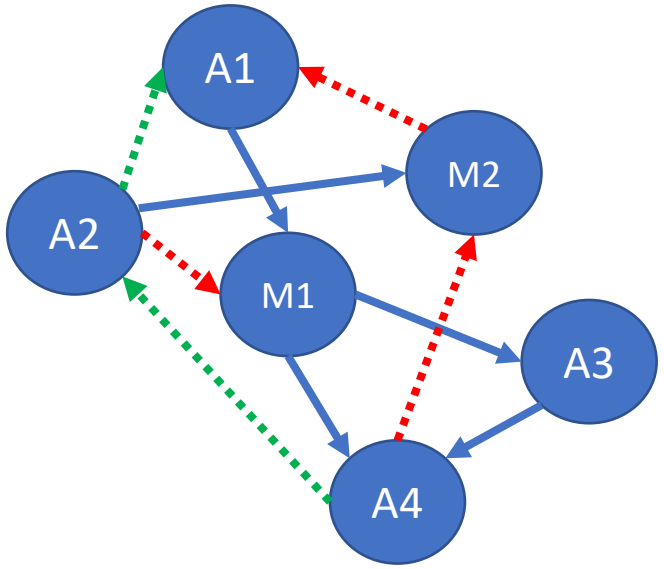
```
A1: add r0 x8 x11
M1: mul x9 x6 x13
A2: add x6 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```




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

Renaming Example

```
A1: add r0 x8 x11
M1: mul r1 r0 x13
A2: add x6 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```

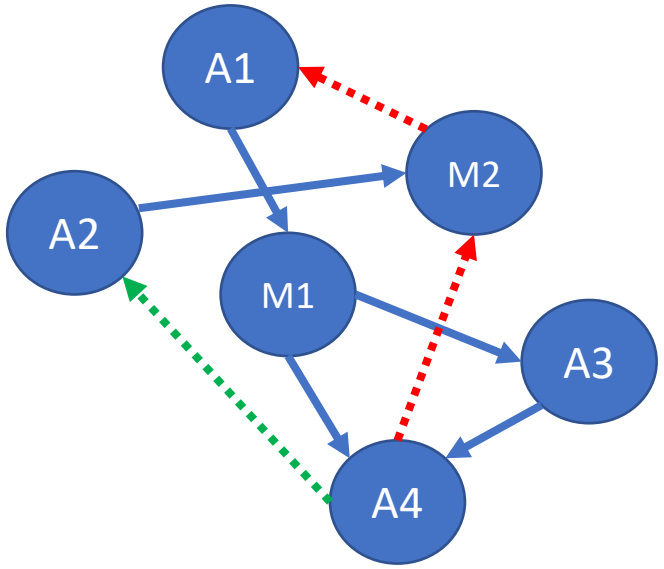



```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
```

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

Renaming Example

```
A1: add r0 x8 x11
M1: mul r1 r0 x13
A2: add r2 x17 x30
A3: add x7 x9 x14
M2: add x8 x18 x6
A4: add x6 x7 x9
```



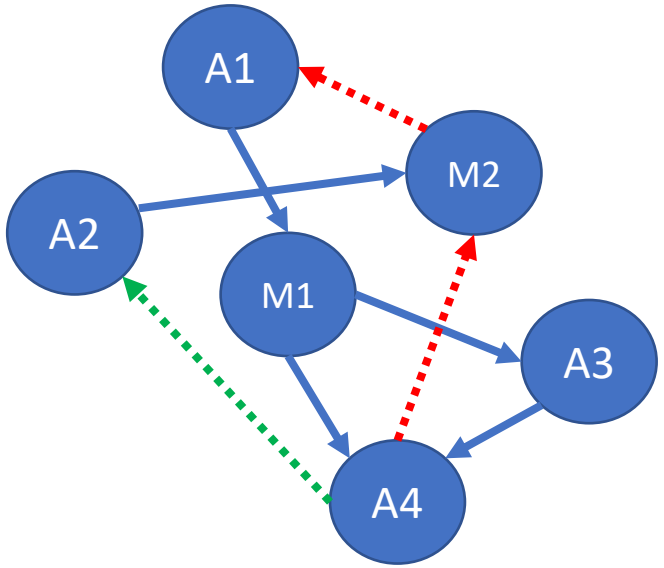

```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
A2.x6 -> r2
```

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

Resolved by renaming output regs

Renaming Example

```
A1: add 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 x6 x7 x9
```



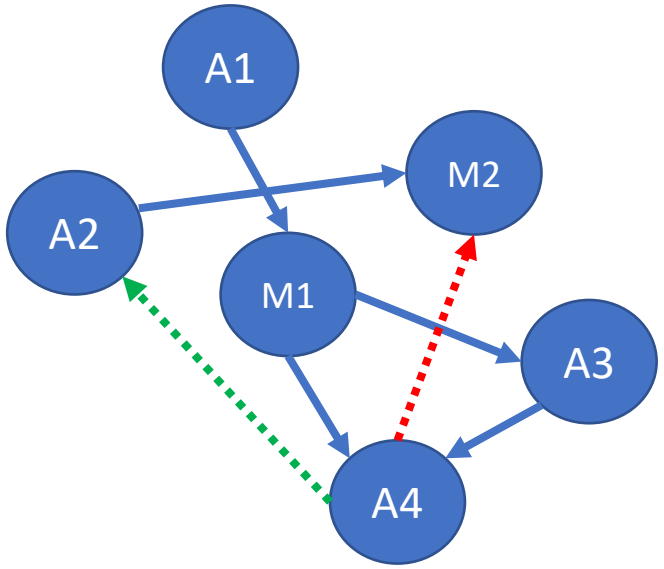

```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
A2.x6 -> r2
A3.x7 -> r3
A3.x9 <- r1
M2.x8 -> r4
M2.x6 <- r2
```

RAW dependence between M1 & A3

Cannot be resolved by renaming

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 x6 x7 x9
```



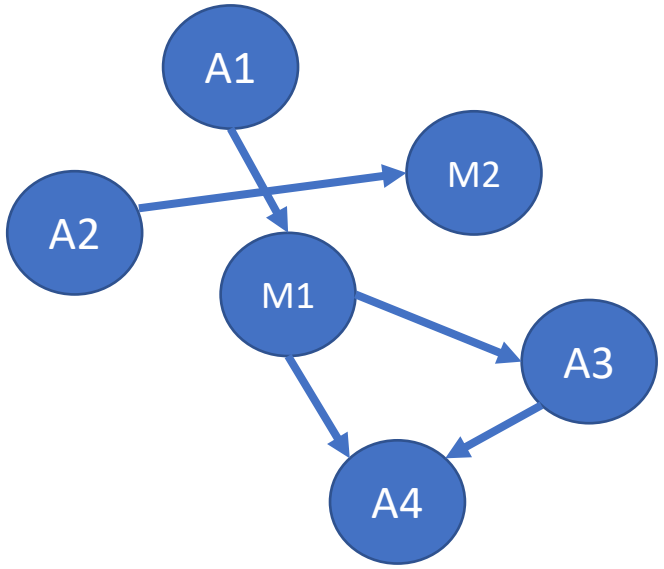

```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
A2.x6 -> r2
A3.x7 -> r3
A3.x9 <- r1
M2.x8 -> r4
M2.x6 <- r2
```

WAW dep w/ A1 resolved by renaming

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

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
```



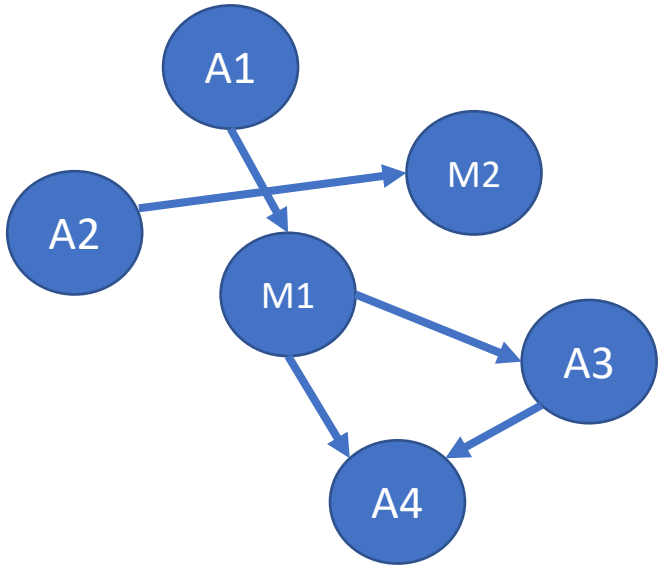

```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
A2.x6 -> r2
A3.x7 -> r3
A3.x9 <- r1
M2.x8 -> r4
M2.x6 <- r2
A4.x6 -> r5
A4.x7 <- r3
A4.x9 <- r1
```

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
```

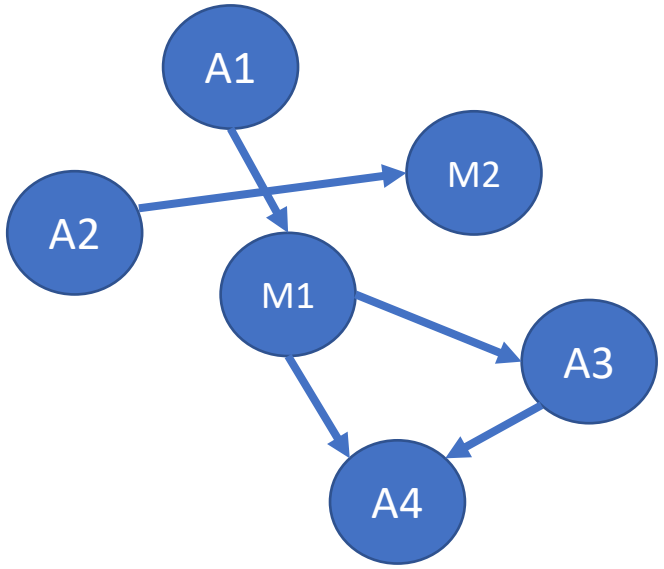



```
Rename Table
A1.x6 -> r0
M1.x9 -> r1
M1.x6 <- r0
A2.x6 -> r2
A3.x7 -> r3
A3.x9 <- r1
M2.x8 -> r4
M2.x6 <- r2
A4.x6 -> r5
A4.x7 <- r3
A4.x9 <- r1
```

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
```

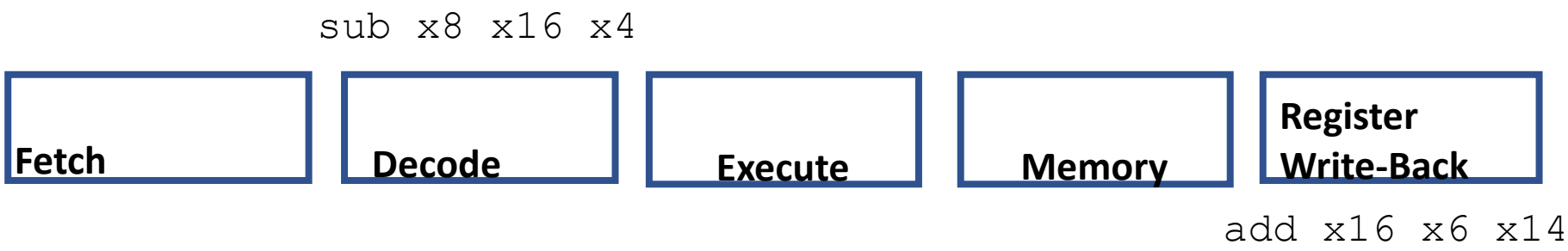
After register renaming, only RAW dependences (i.e., “True Dependences”) remain in the execution

Renaming Avoids False Deps

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

Write-After-Read (WAR)

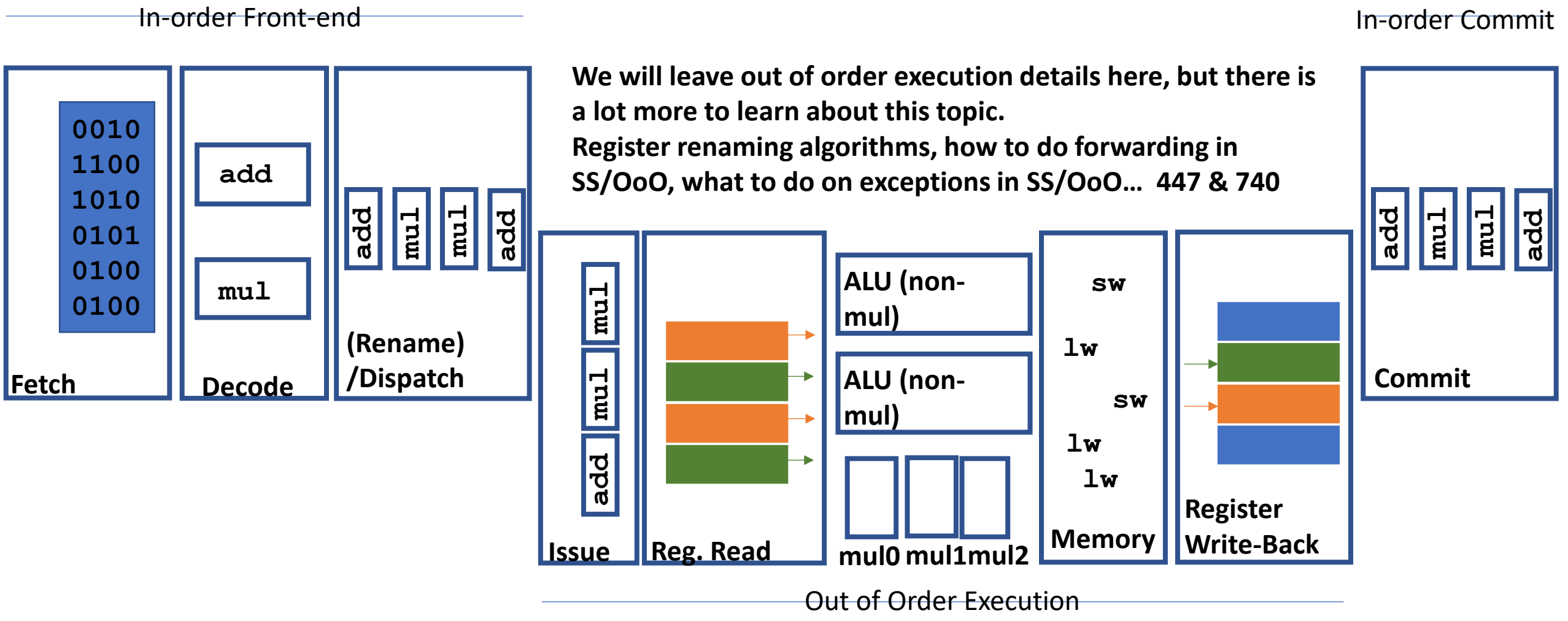
Stalled at decode/reg. read



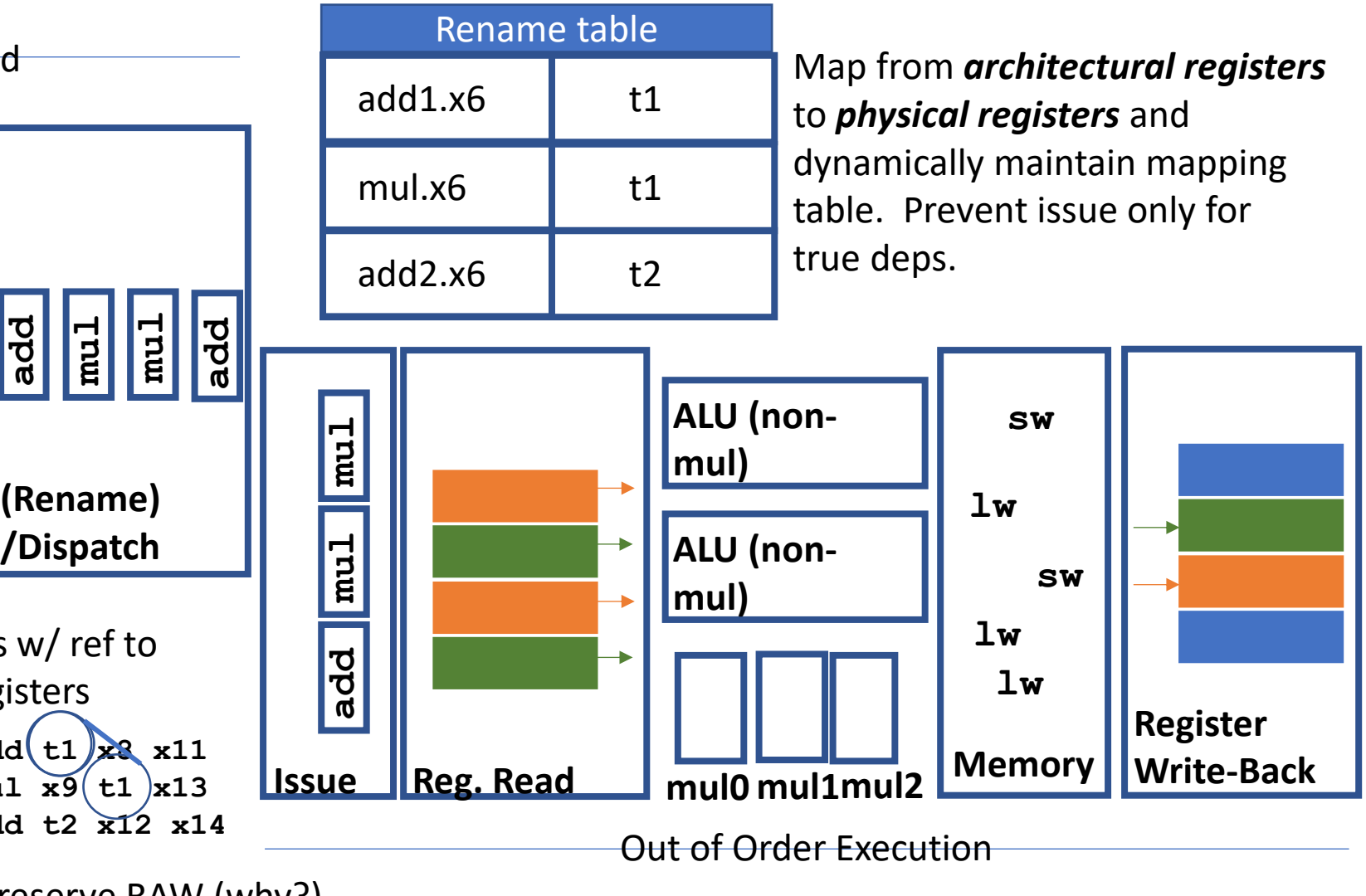
Completes quickly and writes reg.

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*



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

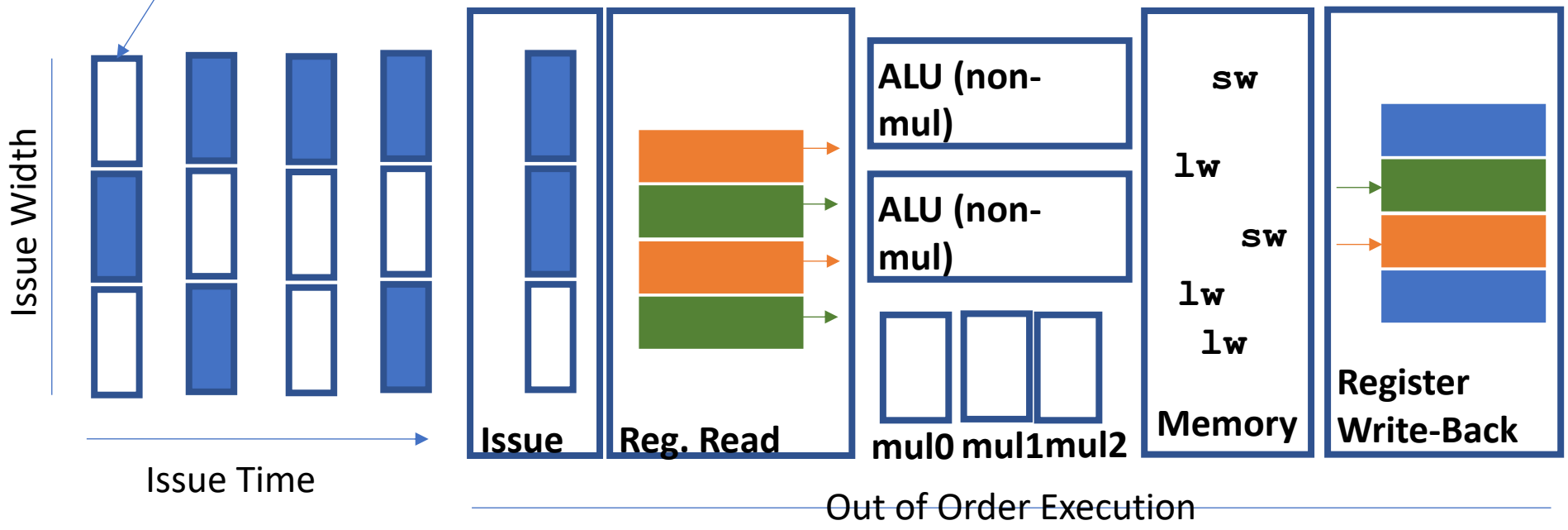


Superscalar execution exploits ILP to increase IPC

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

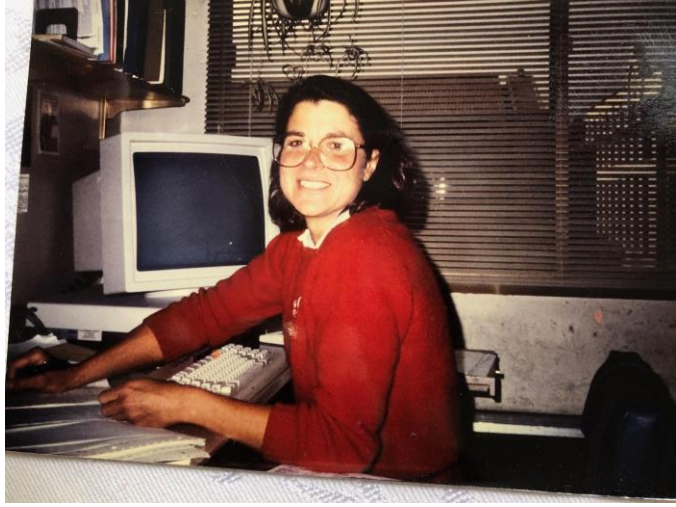
Performance in a superscalar processor depends on the existence of ILP in the program.

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



Simultaneous Multi-Threading (SMT)

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



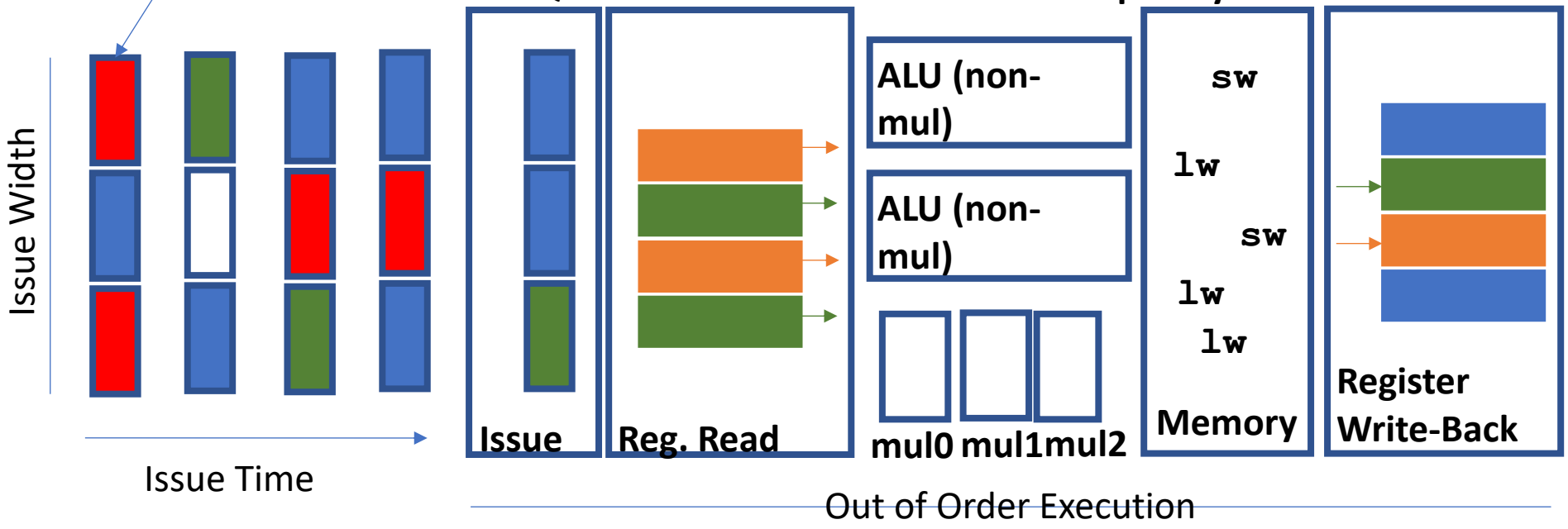
Susan Eggers, inventor of SMT, ca. 1980

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.

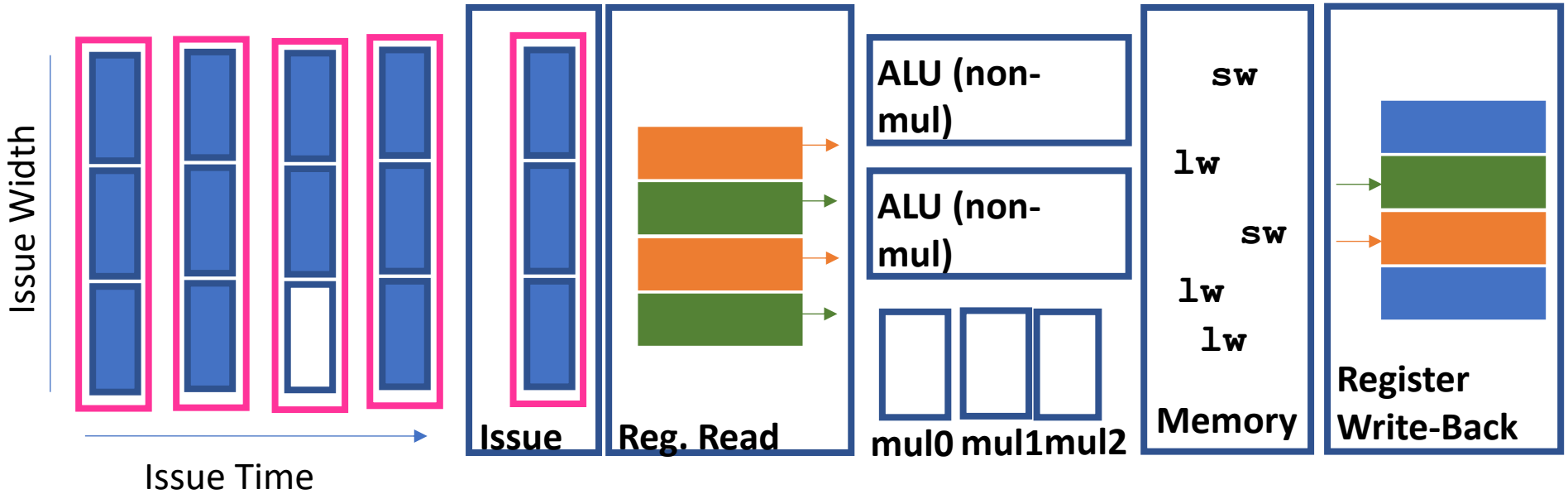
Question: Sources of hardware complexity for SMT?



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.

Today: More Advanced Architecture Concepts

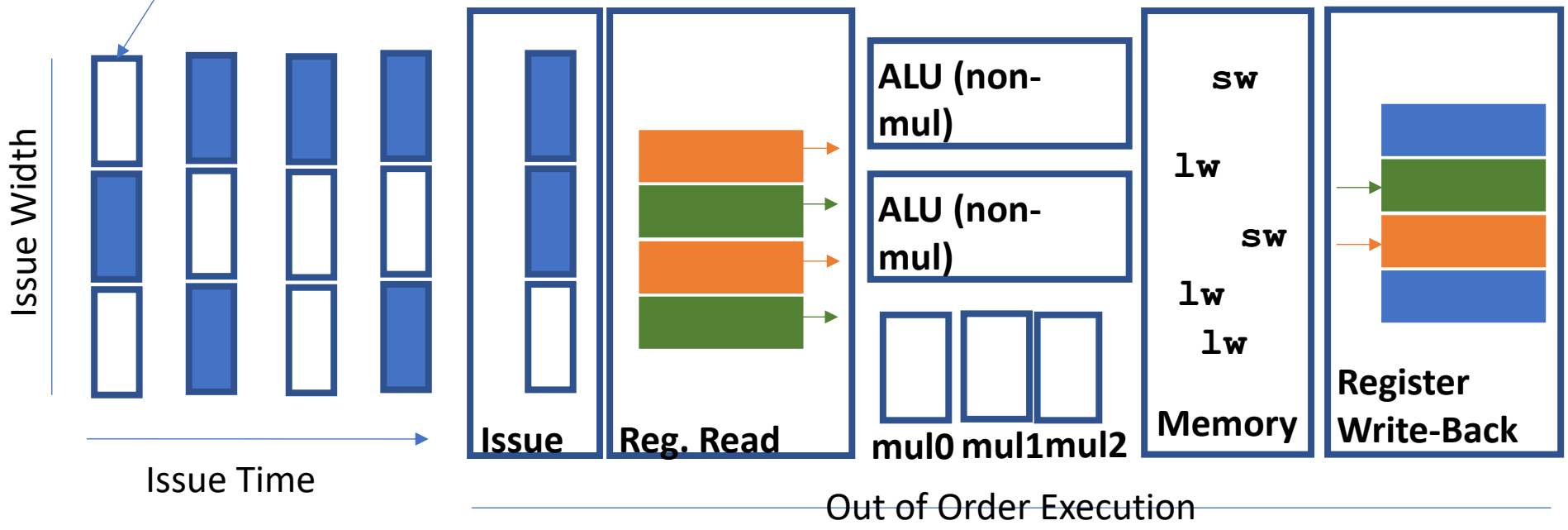
- (more) VLIW
- Vector machines & SIMD
- Dataflow as a hardware/software boundary design problem
- Systolic Array Architectures (if time)

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?

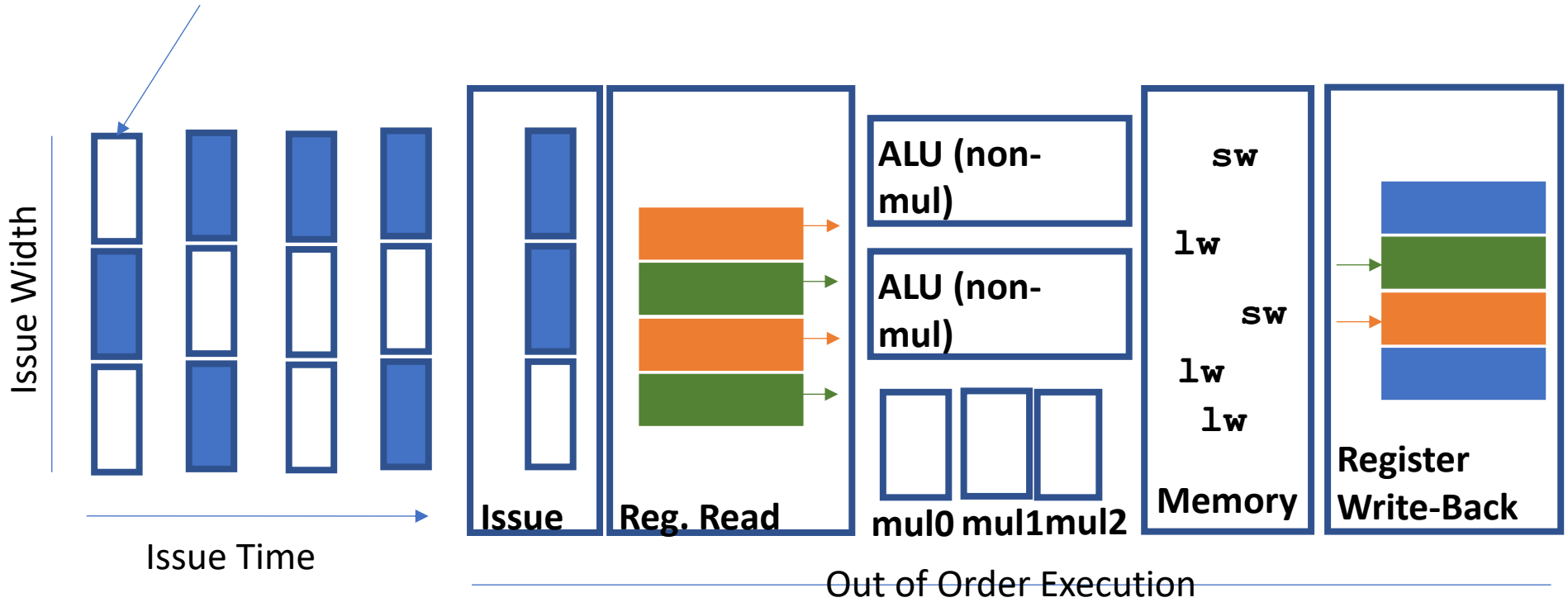


Superscalar execution exploits ILP to increase IPC

Question: how to avoid issue slot waste?

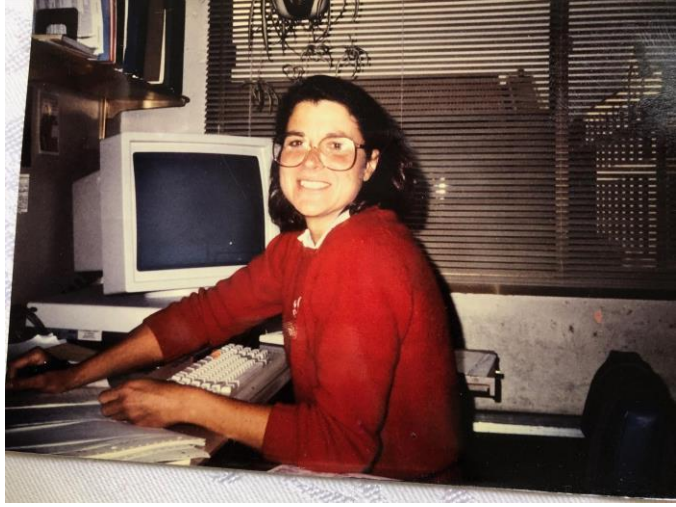
- *Schedule code in program to avoid dependences*
- *Schedule code in loops to align with fetch granularity*
- *Schedule code to avoid oversubscribing functional units (i.e., a sequence of consecutive multiplies can't issue together)*

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.



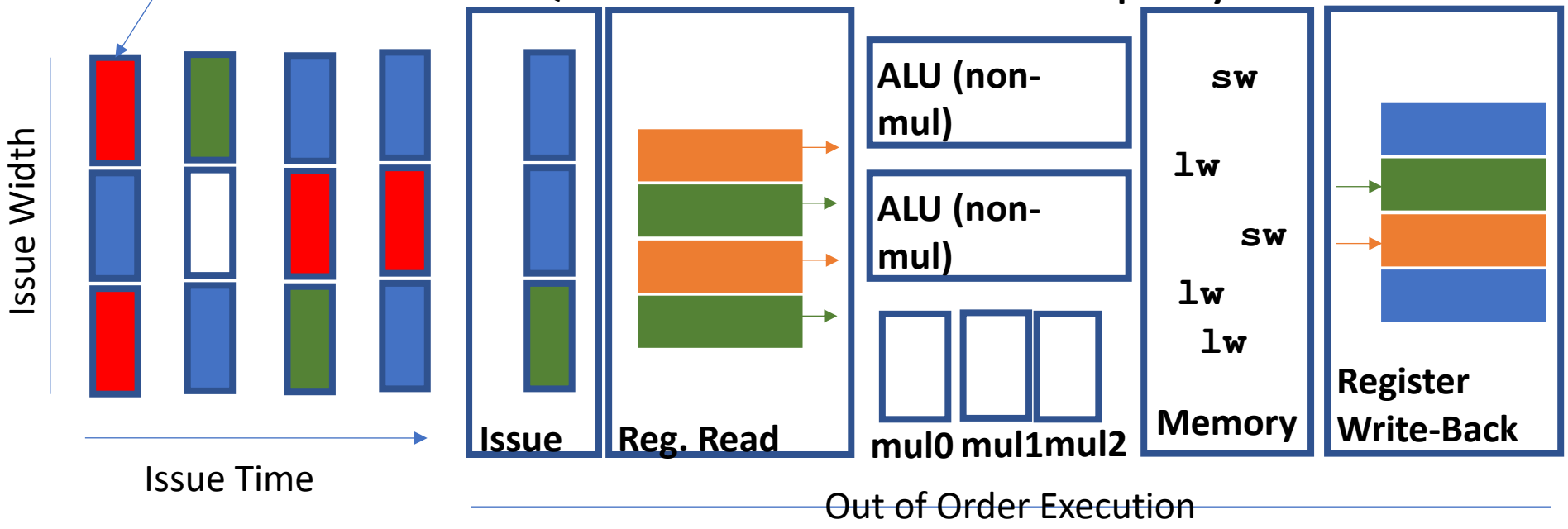
Susan Eggers, inventor of SMT, ca. 1980

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?

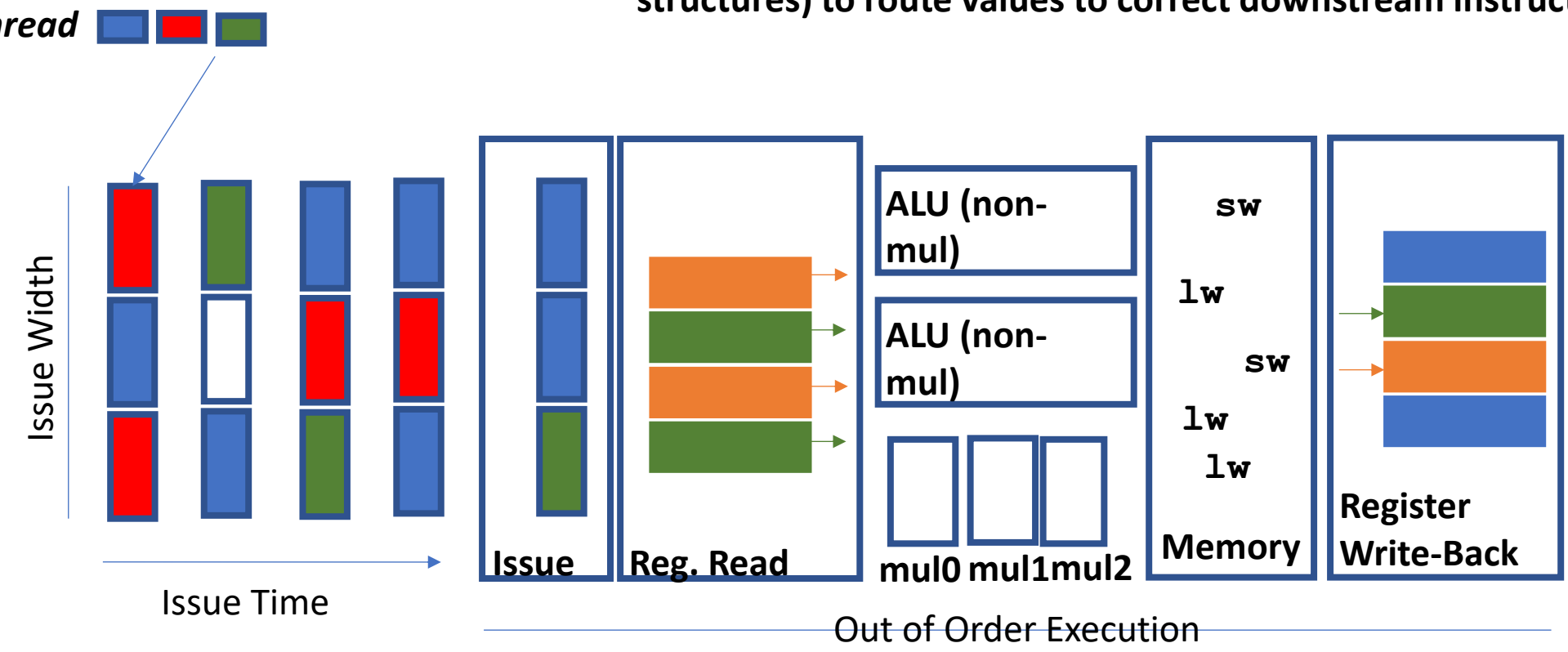


Simultaneous Multi-Threading (SMT)

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

Question: Sources of hardware complexity for SMT?

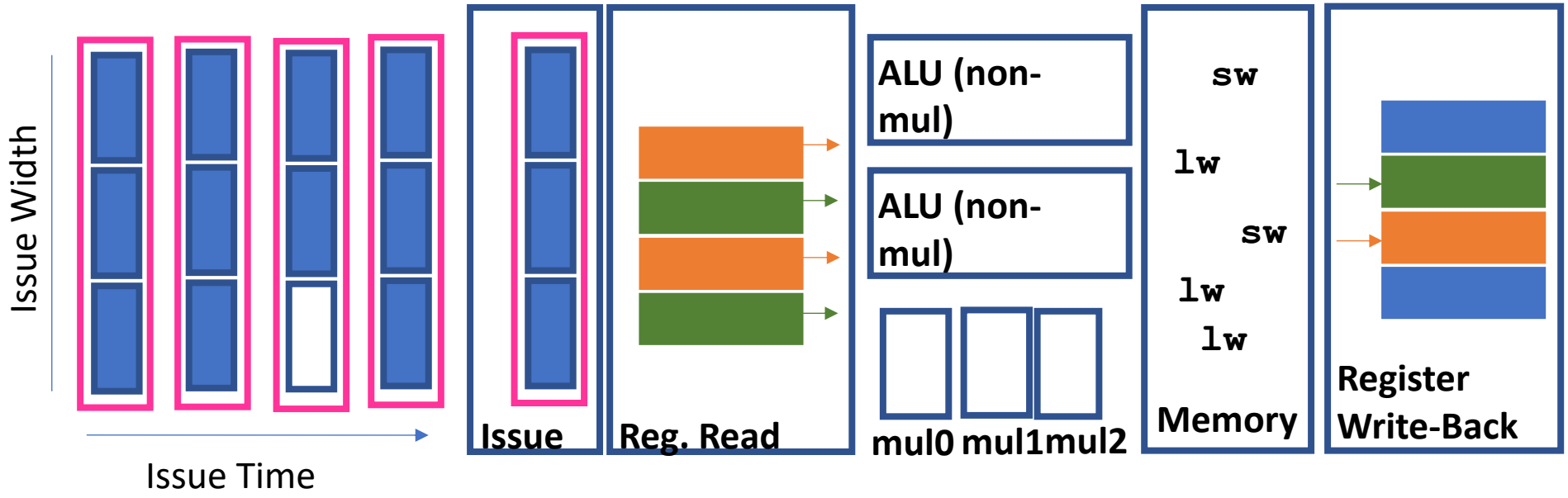
- **Need fetch to support multiple streams (including branch prediction logic…)**
- **Need to tag functional units, rename table entries, ROB entries (and other structures) to route values to correct downstream instructions**



Very Large Instruction Word (VLIW) and the EPIC Architecture (Explicit Parallel Instruction Computer)

Change the *ISA*! In VLIW, the ISA exposes issue width architecturally. Each fetch / issue is on a *bundle* of instructions that are independent

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



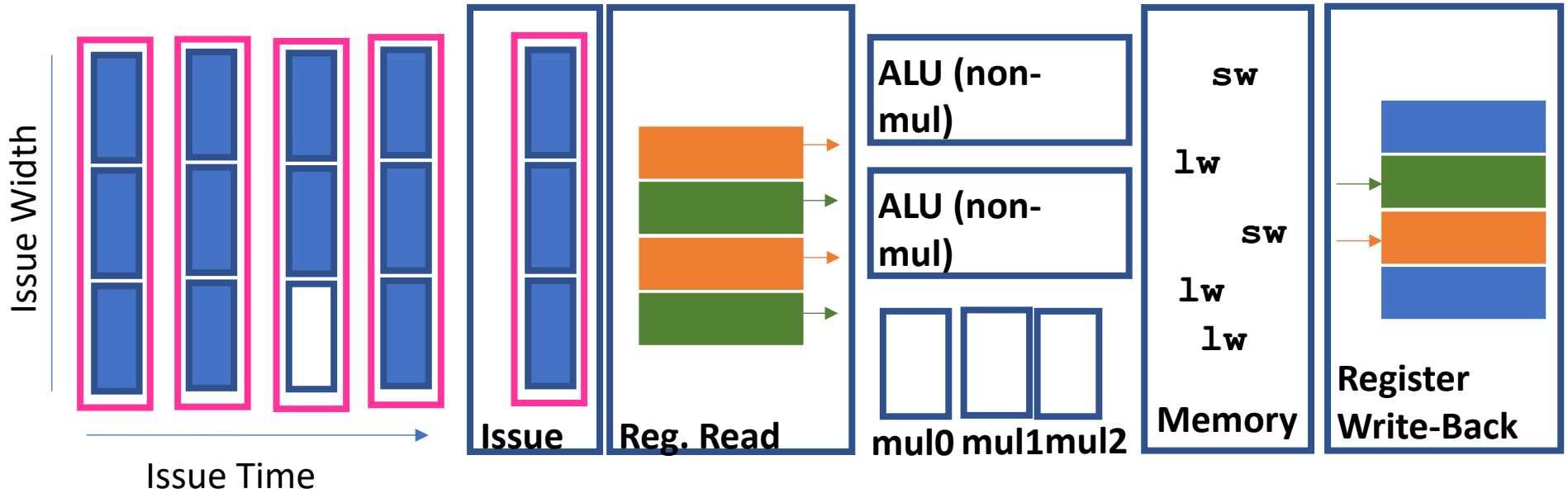
Type:
Mem, Float,
Int, Long Imm.
Branch
e.g.,
MMI, IIF, MMI
MM/I, M/MI

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

Very Large Instruction Word (VLIW) and the EPIC Architecture (Explicit Parallel Instruction Computer)

What do we rely on for VLIW to work? What assumptions do we depend on for VLIW to work and be efficient?

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



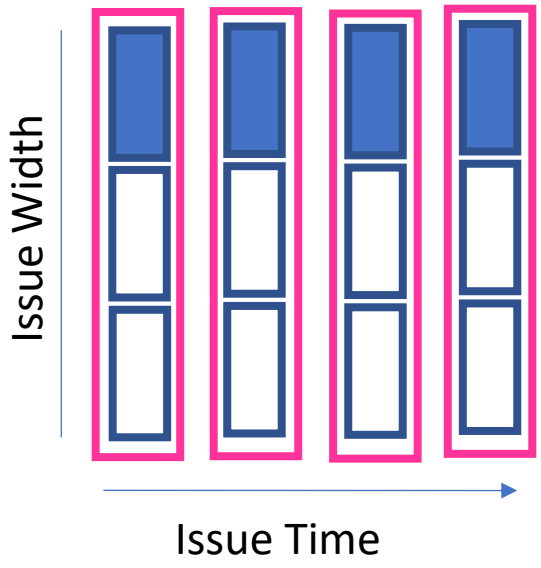
Type:
Mem, Float,
Int, Long Imm.
Branch
e.g.,
MMI, IIF, MMI
MM/I, M/MI

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

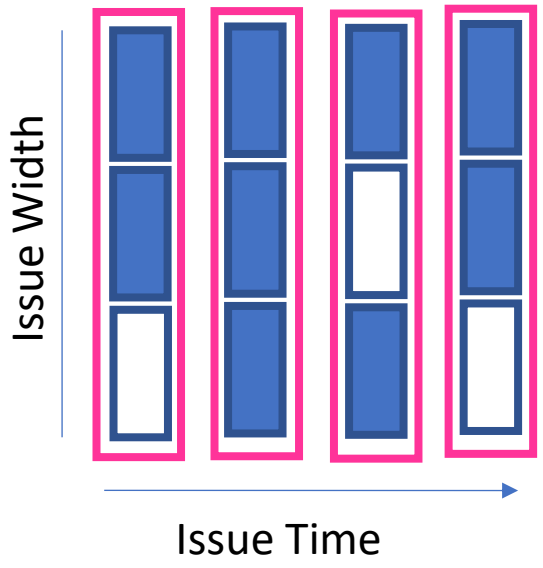
Very Large Instruction Word (VLIW) and the EPIC Architecture (Explicit Parallel Instruction Computer)

Software-constructed (compiler-constructed) bundles of instructions can come from anywhere

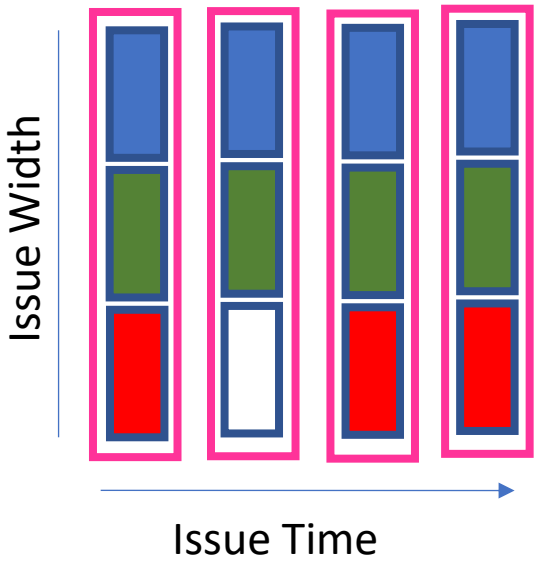
EPIC assumes ***in-order execution*** (static scheduling) and presence of ILP exploiting features, e.g., branch pred., load speculation



Like single-issue scalar execution



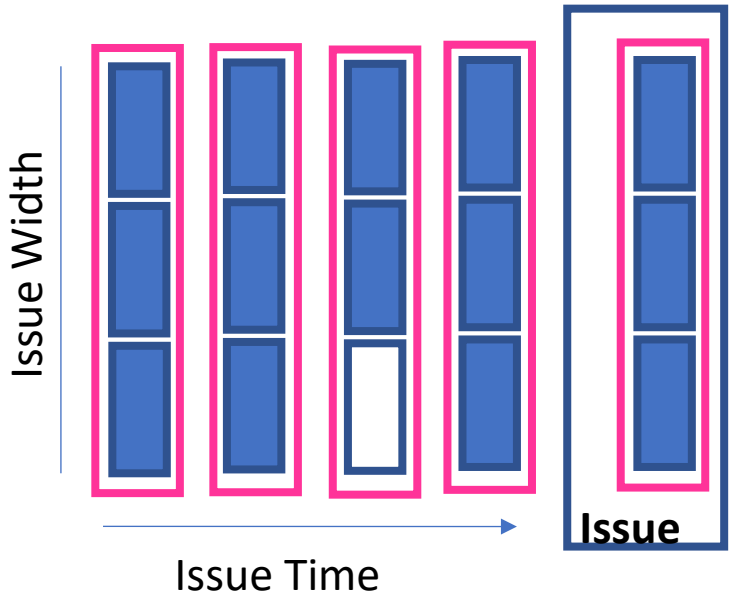
Like multi-issue superscalar execution



Like SMT superscalar, exploiting thread-level parallelism in prog.

Superscalar OoO is great at finding ILP to reduce CPI, but EPIC eliminates dynamic scheduling. Why?

Question: how can static scheduling be good enough to justify eliminating dynamic scheduling & SS/OoO?

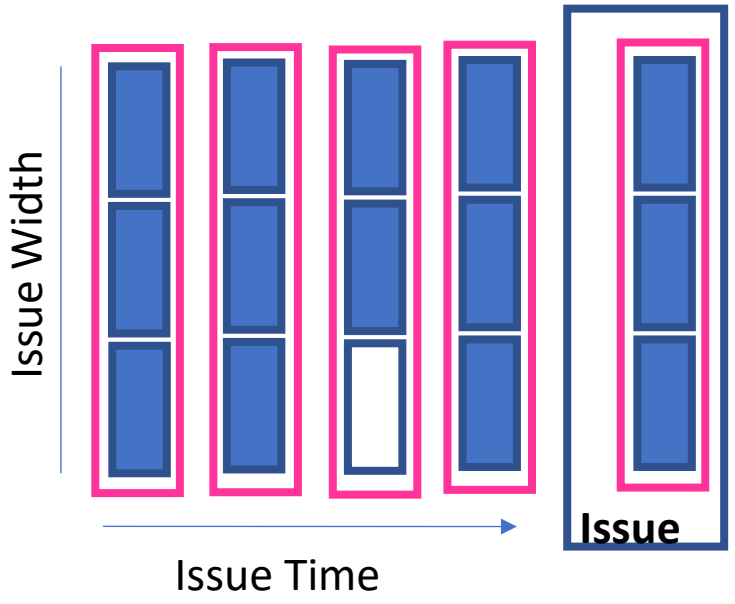


Goal for static & dynamic scheduling: Find instructions to keep the issue window full at all times.

```
ld x11 (x8)
add x2 x3 x4
mul x4 x10 x11
mul x10 x8 x9
st x10 (x8)
add x6 x8 x11
mul x9 x6 x13
add x6 x12 x14
```

Dynamic Scheduling vs. Static VLIW

Question: how can static scheduling be good enough to justify eliminating dynamic scheduling & SS/OoO?



Goal for static & dynamic scheduling: Find instructions to keep the issue window full at all times.

Dynamic scheduling has a limited scope for analysis and optimization

Short window limits reordering distance

```
ld x11 (x8)
add x2 x3 x4
mul x4 x10 x11
mul x10 x8 x9
st x10 (x8)
add x6 x8 x11
mul x9 x6 x13
add x6 x12 x14
```

Static scheduling has *global* scope for reordering / optimization

Long window allows long reorderings

```
ld x11 (x8)
mul t1 x10 x11
mul x10 x8 x9
... //other ops taking t_mul cycles
add x2 x3 x4
st x10 (x8)
```

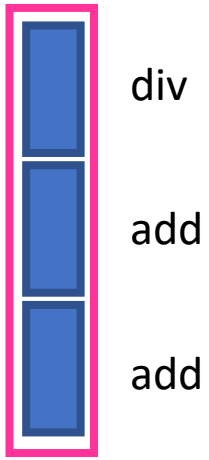
Latency = t_mul

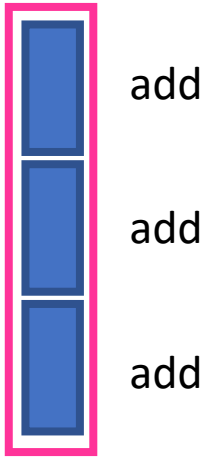
At this point, muls are done and we keep rolling, overlapped latency.

Effective scheduling relies on approximately equal execution latency for all instructions

- If some instructions in a bundle are long-latency and others short-latency, the longs delay the shorts
  - *Scheduling same-latency ops together keeps the machine moving*
- **What about unpredictable latency instructions like memory operations?**

not awesome → (why?)





very awesome

Effective scheduling relies on approximately equal execution latency for all instructions

- If some instructions in a bundle are long-latency and others short-latency, the longs delay the shorts
  - *Scheduling same-latency ops together keeps the machine moving*
- **What about unpredictable latency instructions like memory operations?**
  - **Unpredictable stalls in the pipeline waiting for memory operations**.
  - Can tolerate latencies by scheduling same-latency operations together, if compiler has an expectation about memory latency, cache structure, and producer / consumer relationships.
    - **This is a very difficult compilers problem!!**

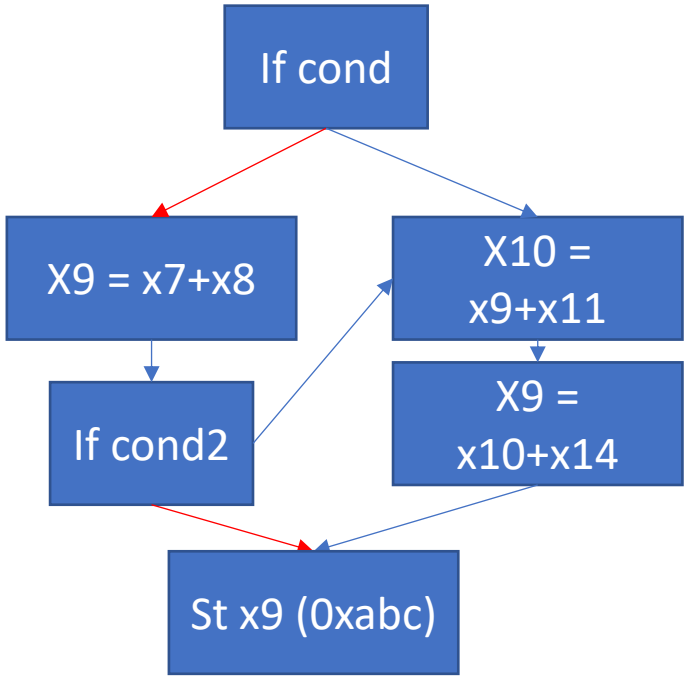
Branch instructions in EPIC

- **EPIC / VLIW does branches differently than in SS/OoO**
- **Option 1:** Waste space in a bundle, run branches like in SS/OoO
  - if taken, grab taken bundle, if not, grab sequentially next bundle
- **Option 2:** Predication
  - run both sides of branch and commit only insns with true predicate
- Predication takes pressure off of control logic & branch prediction (why?)
  - Do we need a branch predictor?
- Costs of predication?

If !P *nullify* insn

If conversion

```
bne cond, pc+12
Add x9 x7 x8
bne cond2 pc+12
Sub x10 x9 x11
Add x9 x10 x14
St x9 (0xabc)
```



The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel® Itanium™ Processor

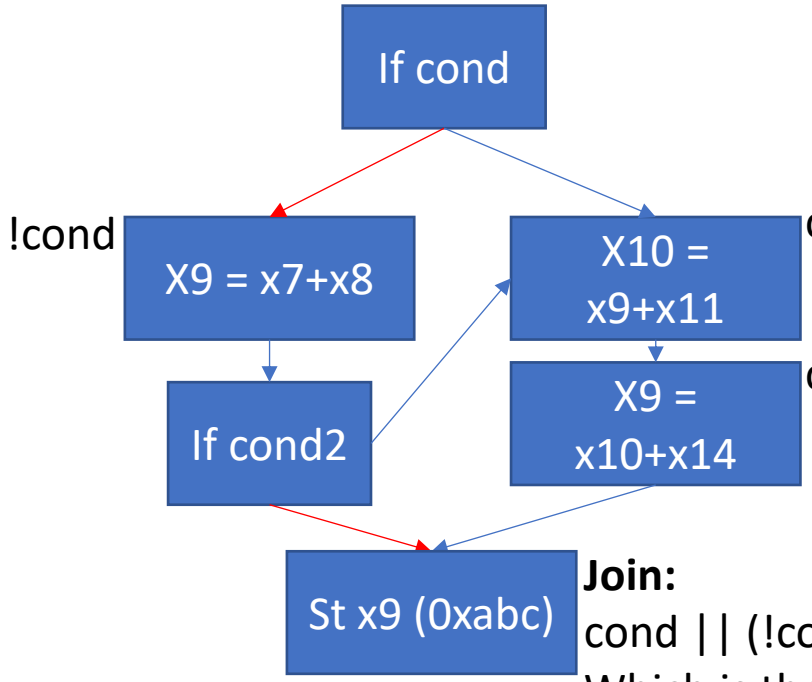
Youngsoo Choi, Allan Knies, Luke Gerke, Tin-Fook Ngai
Intel Corporation, MS SC12-304
2200 Mission College Blvd
Santa Clara, CA 95052
{youngsoo.choi, allan.knies, luke.c.gerke, tin-fook.ngai}@intel.com

If conversion

The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel® Itanium™ Processor

Youngsoo Choi, Allan Knies, Luke Gerke, Tin-Fook Ngai
Intel Corporation, MS SC12-304
2200 Mission College Blvd
Santa Clara, CA 95052
{youngsoo.choi, allan.knies, luke.c.gerke, tin-fook.ngai}@intel.com

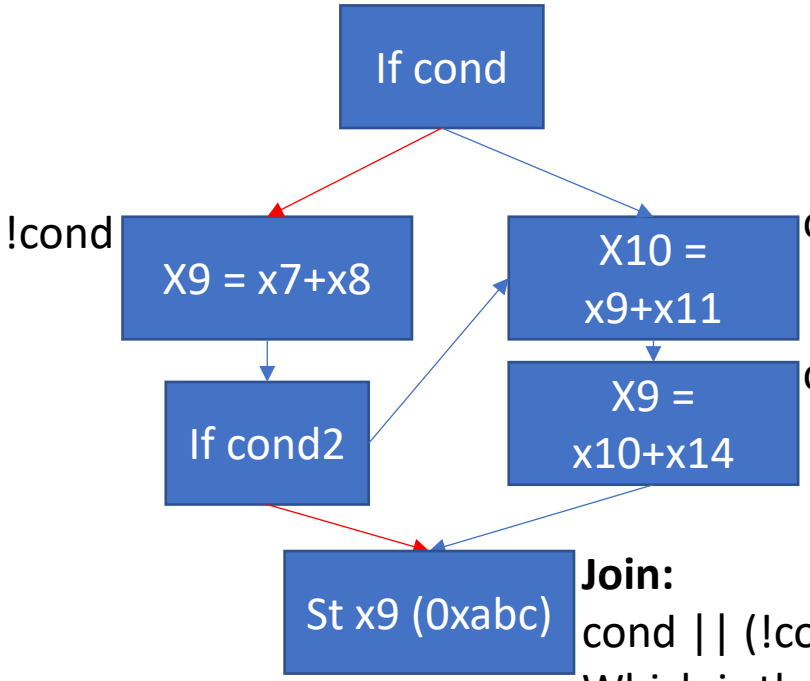
```
bne cond, pc+12
Add x9 x7 x8
bne cond2 pc+12
Sub x10 x9 x11
Add x9 x10 x14
St x9 (0xabc)
```



Join:
cond || (!cond && cond2) || (!cond && !cond2)
Which is the same as:
cond || !cond, i.e., unconditional

If conversion

```
bne cond, pc+12
Add x9 x7 x8
bne cond2 pc+12
Sub x10 x9 x11
Add x9 x10 x14
St x9 (0xabc)
```



cond || (!cond && cond2)

cond || (!cond && cond2)

Join:
cond || (!cond && cond2) || (!cond && !cond2)
Which is the same as:
cond || !cond, i.e., unconditional

The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel® Itanium™ Processor

Youngsoo Choi, Allan Knies, Luke Gerke, Tin-Fook Ngai
Intel Corporation, MS SC12-304
2200 Mission College Blvd
Santa Clara, CA 95052
{youngsoo.choi, allan.knies, luke.c.gerke, tin-fook.ngai}@intel.com

Store predicate results in explicit predicate registers

```
P1=!cond
P2=cond||(!cond&&cond2)
(P1)Add x9 x7 x8
(P2)Sub x10 x9 x11
(P2)Add x9 x10 x14
St x9 (0xabc)
```

Add explicit predicates to the code that executes. Predicates evaluate **dynamically** using predicate registers.

No branch instructions here!
Microarchitectural implication?

Pipeline Characteristics

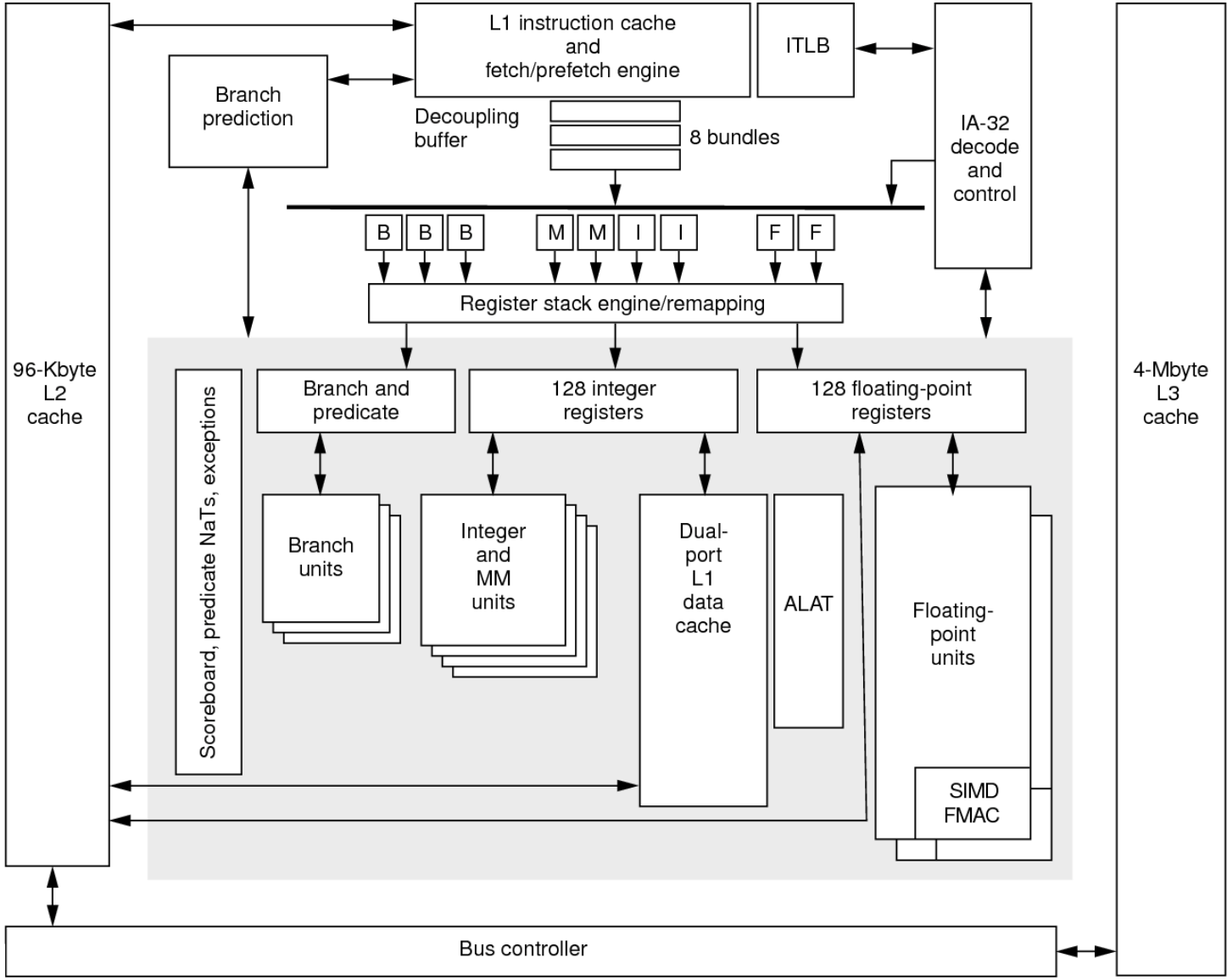
- Execute 2 bundles (6insns) per cycle
- 10 stage pipeline
- 4 Integer Units (2 of which do Ld/St)
- 2 Floating Point Units
- 3 Branch Units
- Issue in order, execute in order
- Simple register dependence tracking using a “scoreboard”

Control Characteristics

- Predication *and* sophisticated two-level branch predictor (why?)
- Instruction queues connect fetch to execute units hiding some fetch bubble latency with execute latency (how?)

Register File

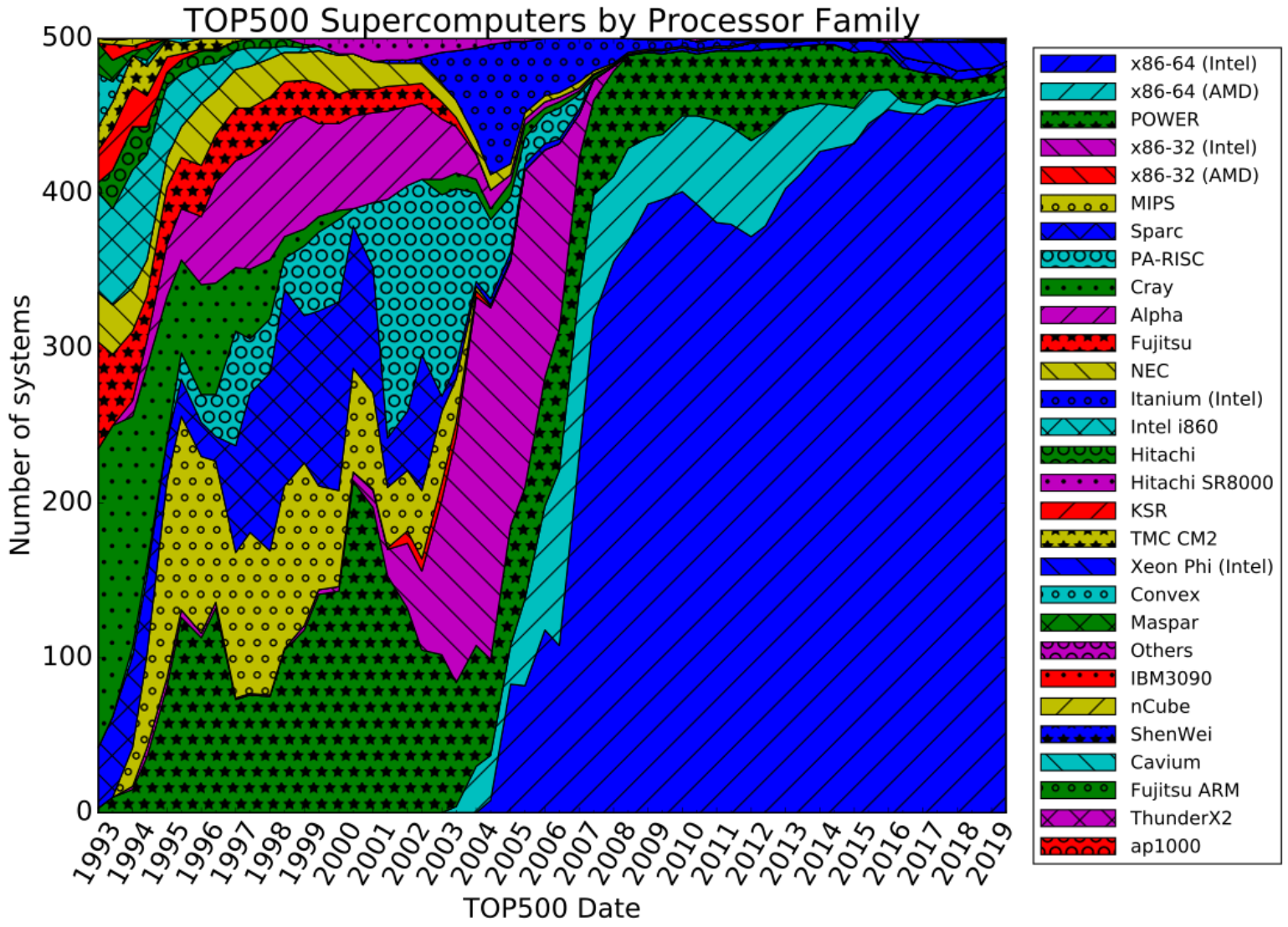
- Fairly complex and highly abundant
- Separate predicate / branch, int, and FP regs
- “Register stack engine” efficiently doles out physical registers, avoiding structural hazard



Intel Itanium EPIC Architecture

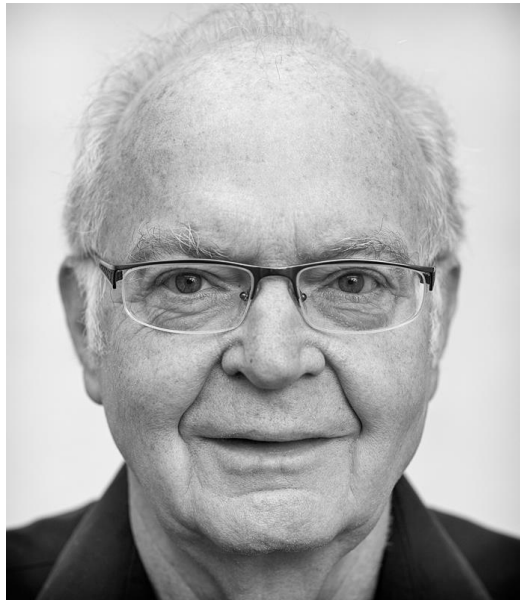
VLIW / EPIC is a Very Cool HW/SW Interface!

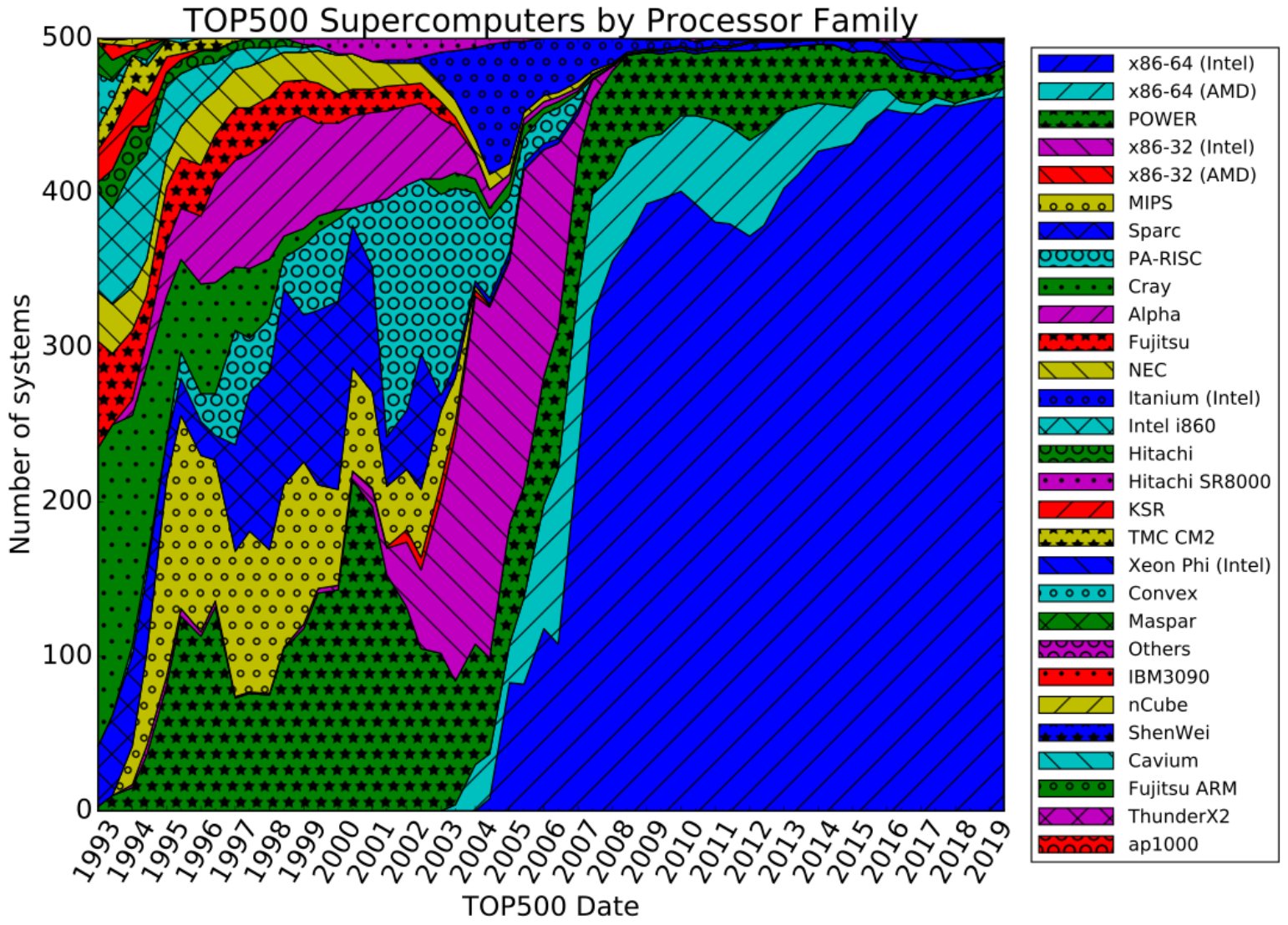
- Why did Itanium not seize the (any?) market as Intel anticipated?
- (In the top500 supercomputers, we mostly have x86-64, not IA64)



VLIW / EPIC is a Very Cool HW/SW Interface!

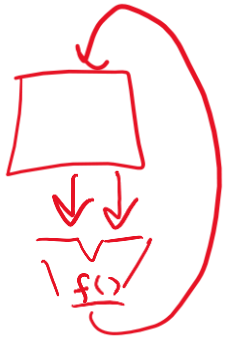
- Donald Knuth: *“the "Itanium" approach [was] supposed to be so terrific—until it turned out that the wished-for compilers were basically impossible to write”*





Parallelism Beyond ILP

Flynn's Taxonomy of Parallel Architectures



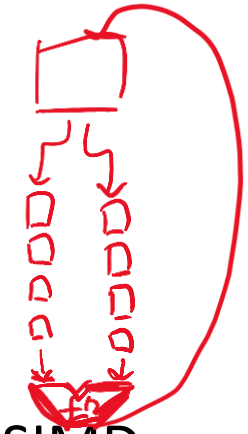
"sequential"

incl. SS/OoO

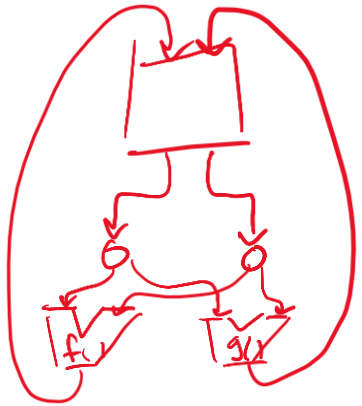
SISD

"vector"

"array"



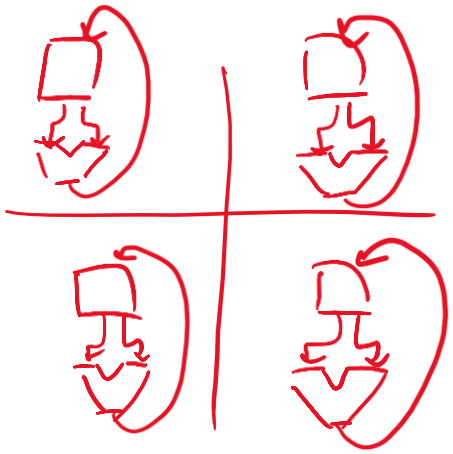
SIMD



"streaming"

"replicated"

MISD



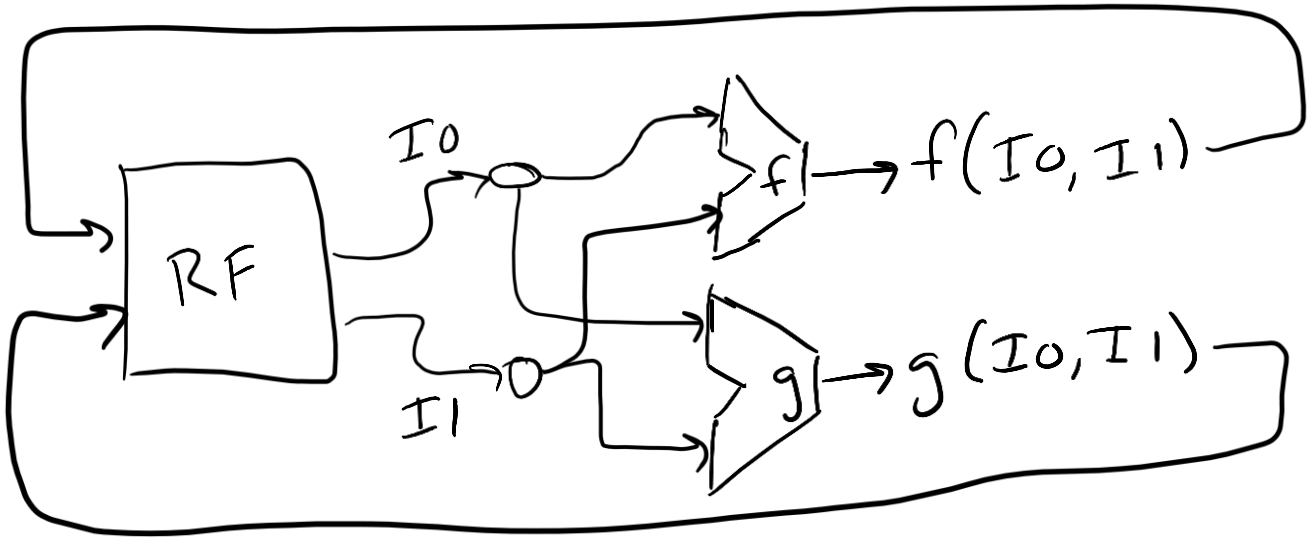
"multicore"

Later this sem.

MIMD

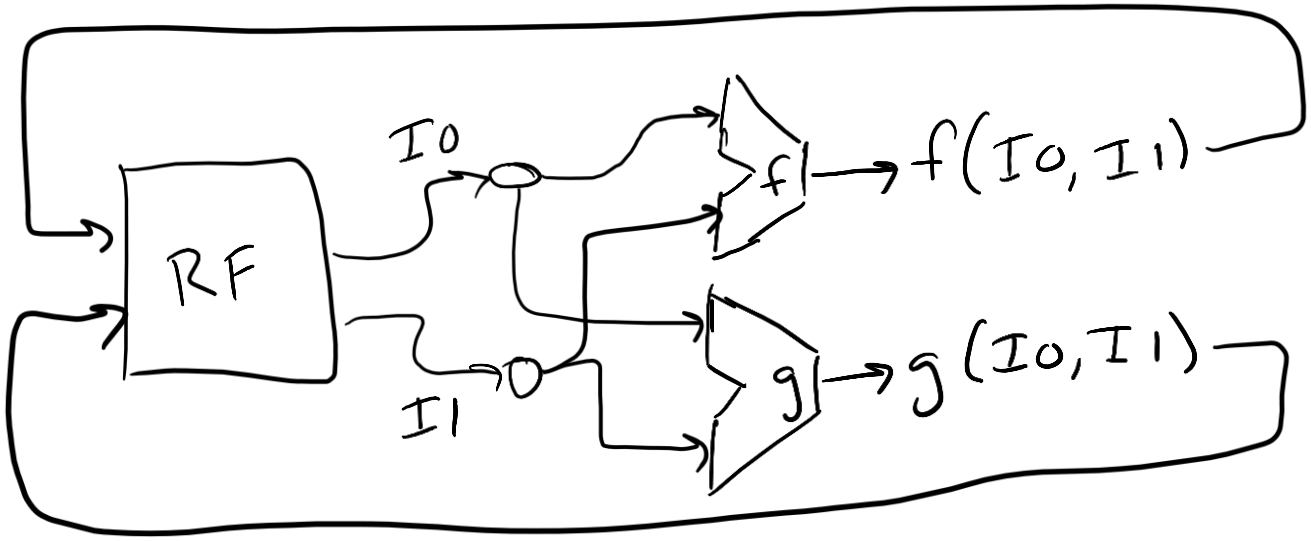
MISD – Multiple Instruction Single Data

- Send same inputs (logically) simultaneously to multiple functions
- Used for …what?



MISD – Multiple Instruction Single Data

- Send same inputs (logically) simultaneously to multiple functions
- Rare, sometimes used in DSP, filter signal using multiple programs
- Modular **redundancy**, replicated hardware for execution (if f == g)



SIMD – Single Instruction Multiple Data

Apply instruction to many data: Single instruction (fetched, decoded, etc) applies its operation to a large number of data elements

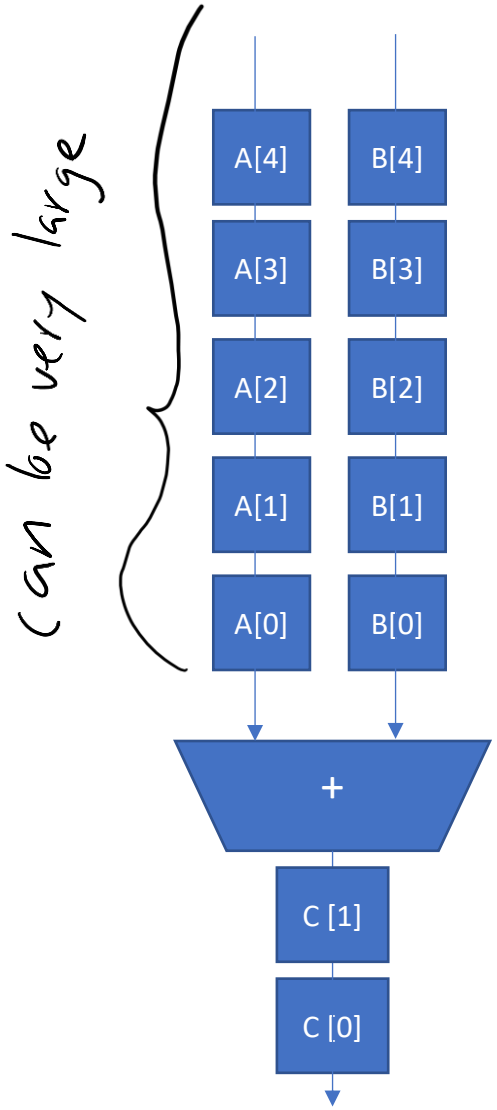
Amortizes Instruction Costs: Each instruction corresponds to a large number of operations

A Few Flavors: most notable/effective are ***Vector*** machines

- Also: historically, ***Array Machines*** but these are not widely used anymore

Data-oriented, Not Necessarily Parallel: Instructions specify what do to to each element of data, but not how to do it (i.e., parallel, partially parallel, sequential)

Vector Machines



`setvl`: tell machine length of input vector. Actual in-memory length can be *thousands* of elements! Machine returns max it can handle in a vector register (varies by implementation, can be tens of elements)

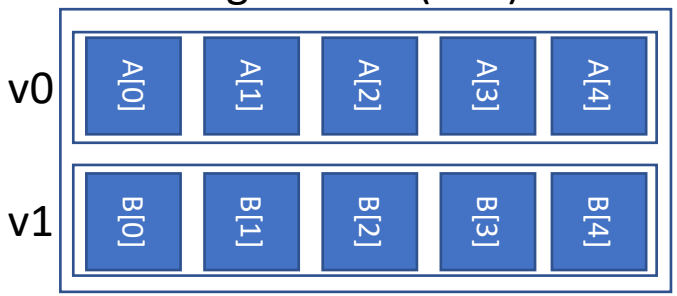
`vld <vector register>, <mem>`: load vector of length `vl` starting at memory location <mem>

`vst <vreg>, <mem>`: store elems of <vreg> to memory starting from memory location <mem>

`vadd <vreg1> <vreg2>`: add element-wise store into vreg1

Assumes explicit vector register file that can temporarily store vector operands

vector register file (VRF)




```
setvl 5
vld v0, a
vld v1, b
vadd v0, v1
vst v0, c
```

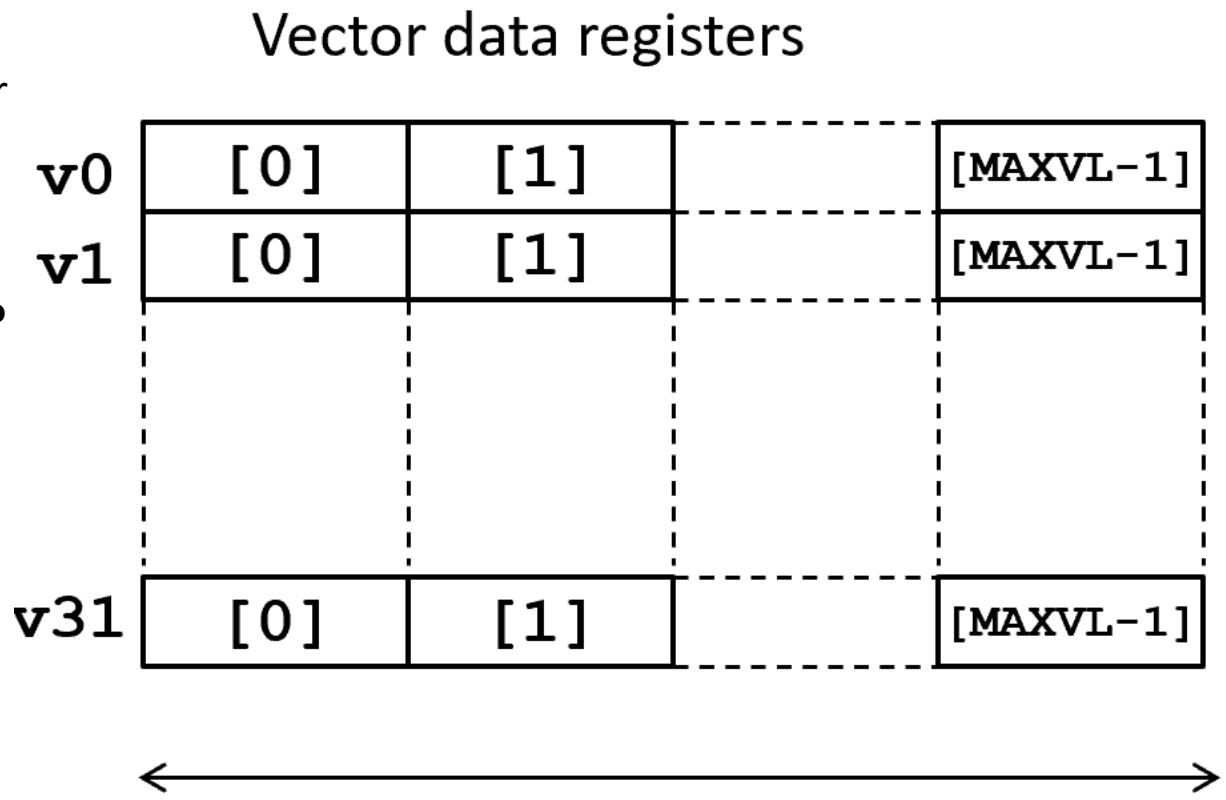
Vector register file

Large, performance-critical structure accessed potentially many times per-cycle during vector operation. **How large? How critical?**

maxvl is an implementation-dependent parameter. **How do we (architects) set maxvl?**

If **setvl** sets **vl** to greater than **maxvl,** then **vl** gets set to **maxvl. HW/SW consequence?**

If **setvl** sets **vl** to less than **maxvl,** then the excess vectors get set to **0** during ops



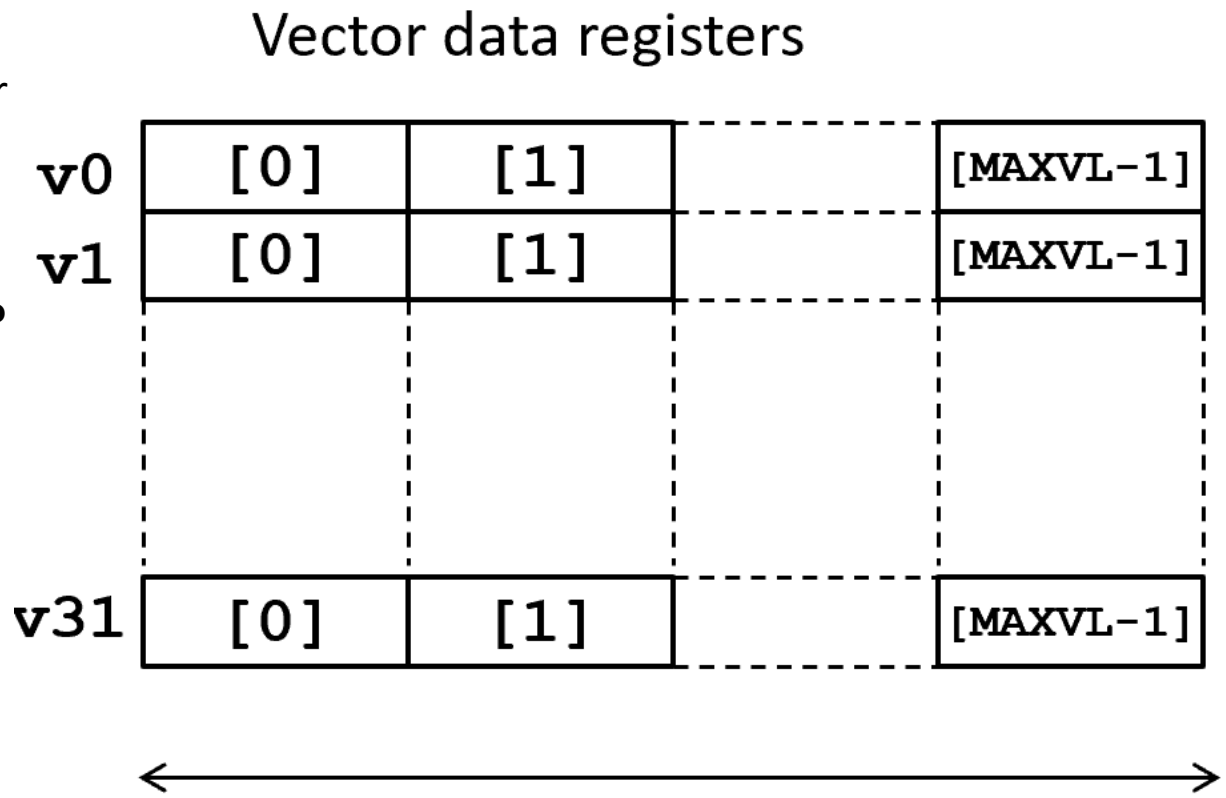
Vector register file

Large, performance-critical structure accessed potentially many times per-cycle during vector operation. **How large? How critical?**

maxvl is an implementation-dependent parameter. **How do we (architects) set maxvl?**

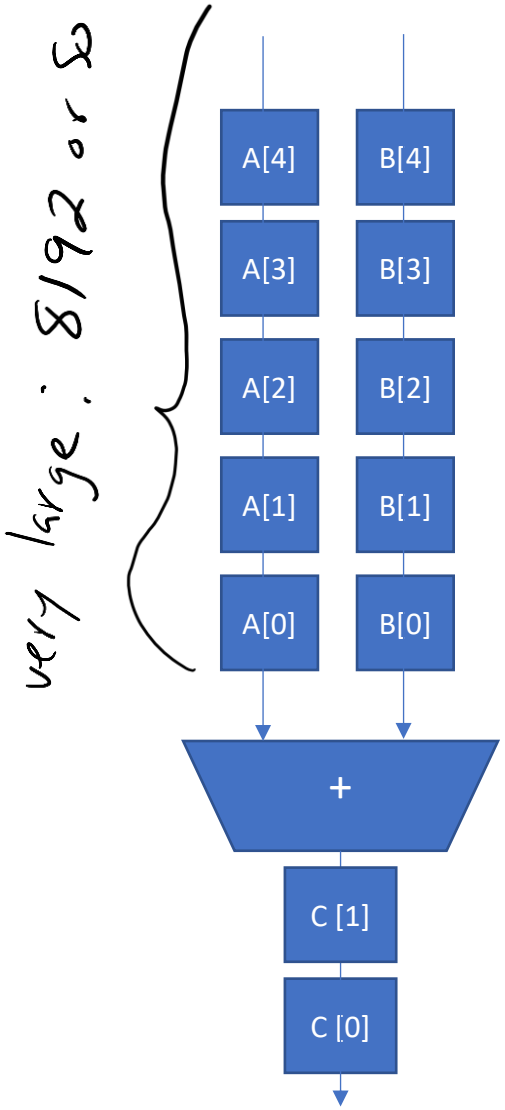
If **setvl** sets **vl** to greater than **maxvl,** then **vl** gets set to **maxvl. HW/SW consequence?**

If **setvl** sets **vl** to less than **maxvl,** then the excess vectors get set to **0** during ops

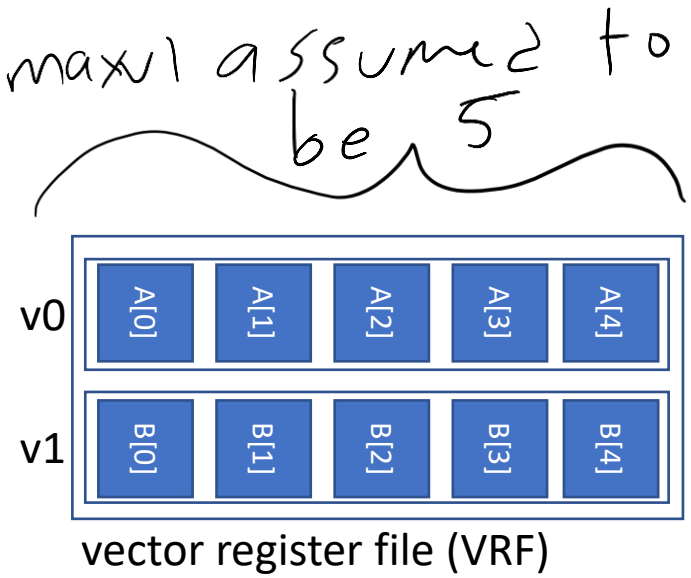


8B / word * 16 words / VRF entry * 32 VRF entries per VRF = 4kB!
Larger than many L1 Caches

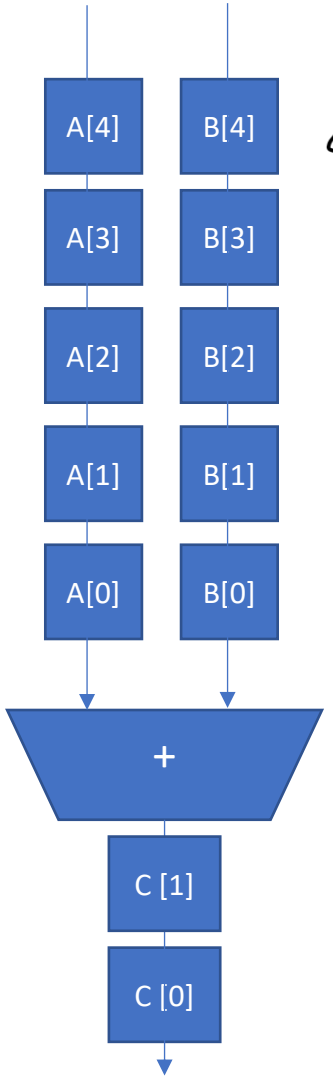
Dealing with limited vector size is easy in SW




```
setvl 8192 //machine says "vl = 5, actually"
for(i = 0…(8192 / vl) ): //loop vl at a time
  vld v0, a + i*vl
  vld v1, b + i*vl
  vadd v0, v1
  vst v0, c + i*vl
```

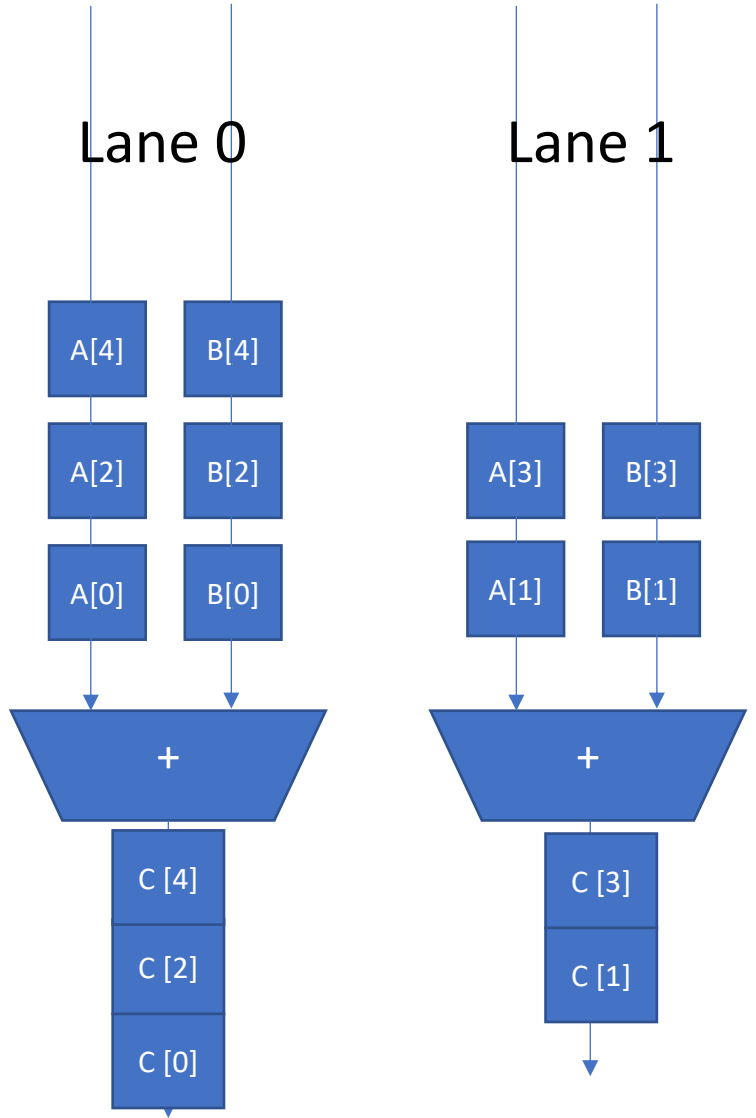


Vector Machines are Easily Parallelizable

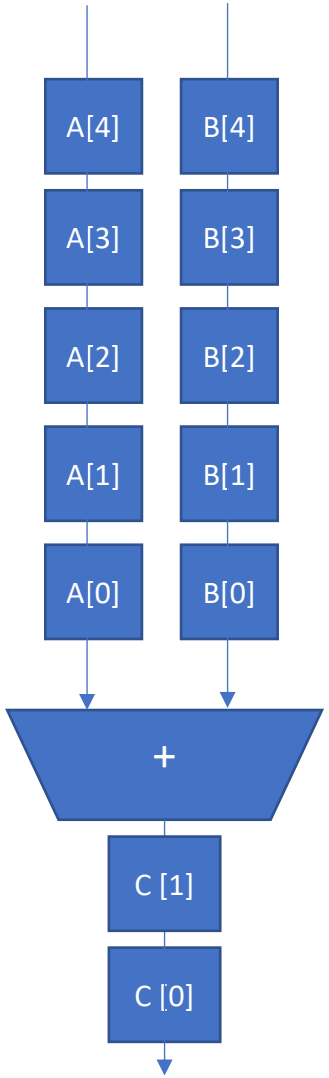


Abstraction: execute an instruction's operation over an entire vector of data

Implementation: Parallel functional units each process parts of a vector, producing a vector output. **Why simple?**



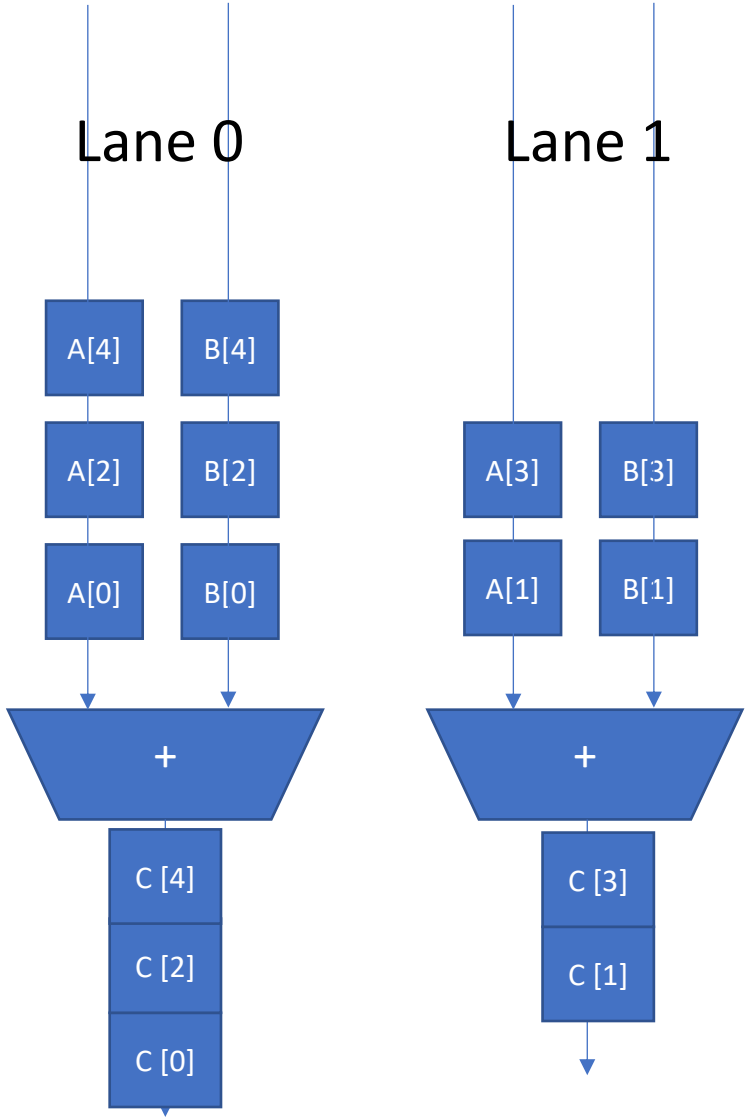
Vector Machines are Easily Parallelizable



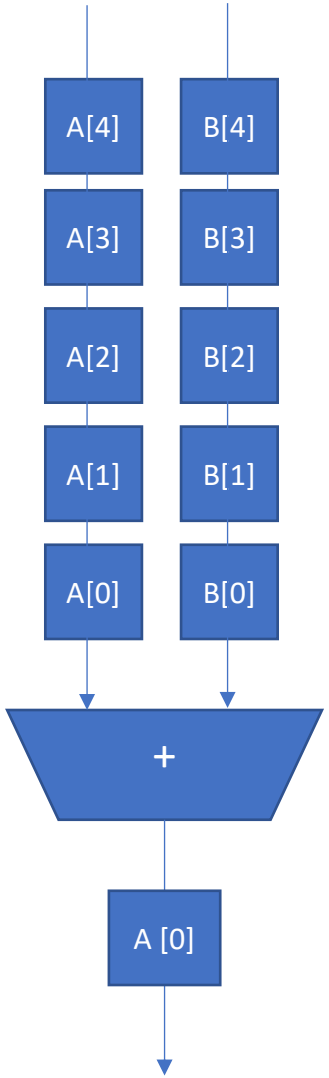
Simple: Vector instruction operates on v0[**i**] and v1[**i**] ***not*** v0[**i**] and elem *v.

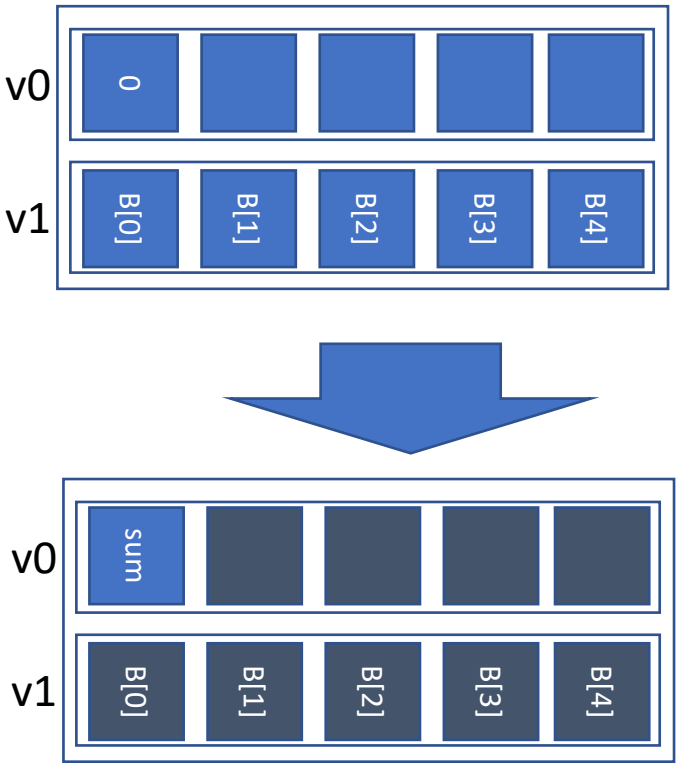
Very simple operand matching logic, no need to track complex producer consumer relationships across inputs of operations.

Primary cost?



Reduction Operations






```
setvl 5
vld v0, a
vld v1, b
vredsum v0, v0, v1, vm
```

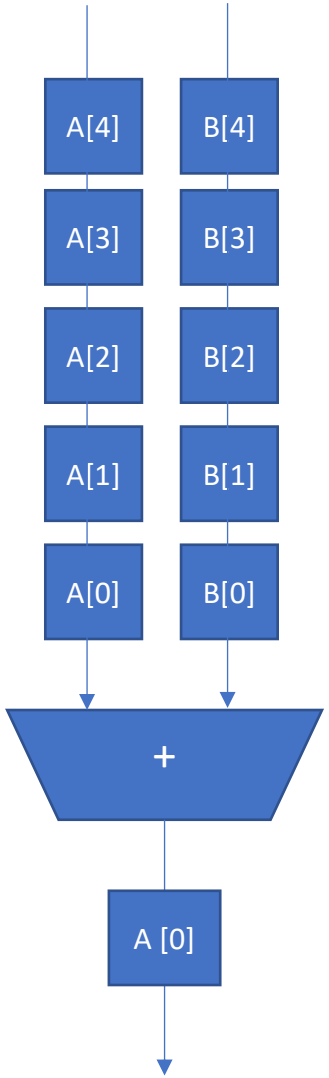
dest vreg [0]

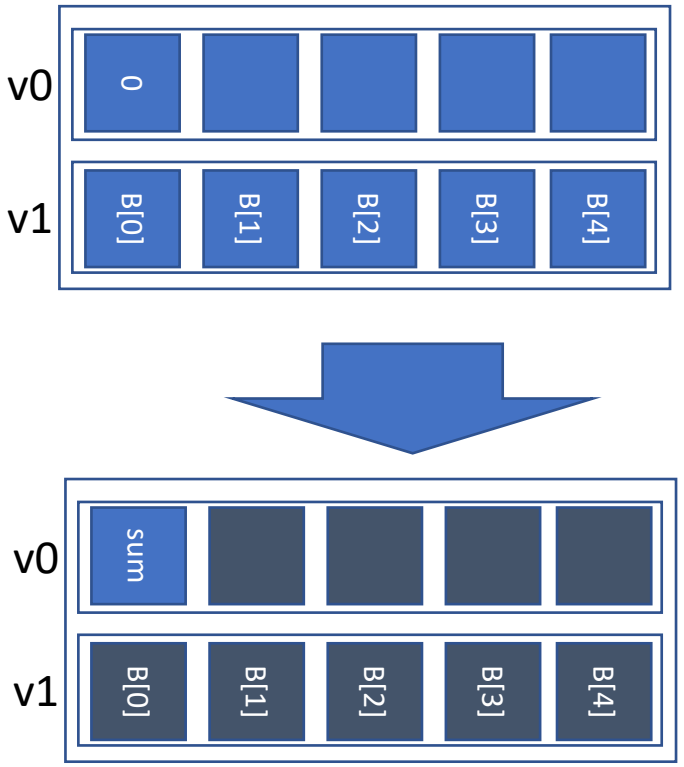
vreg to reduce

initial value

$v0[0] = v0[0] + \Sigma_i v1[i]$

Reduction Operations






```
setvl 5
vld v0, a
vld v1, b
vredsum v0, v0, v1, vm
```

v0[0] =

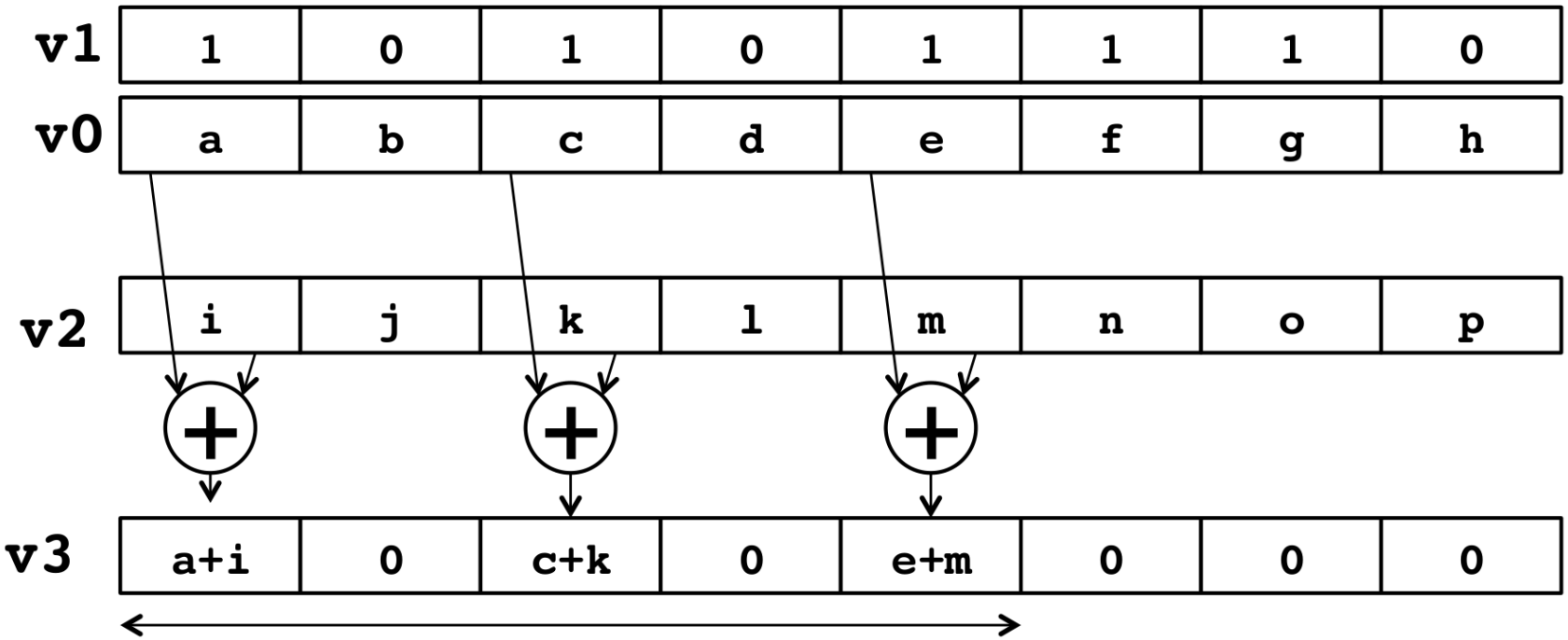
v0[0] + Σ_i v1[i]

Vector Masking

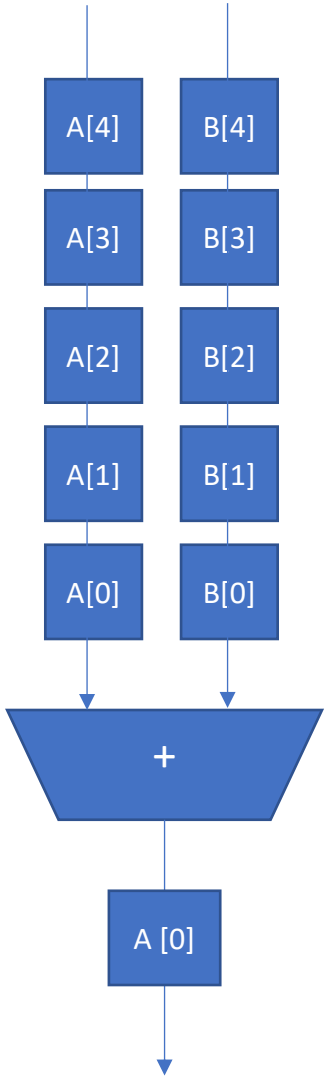
`vadd v3, v0, v2, v1.t`

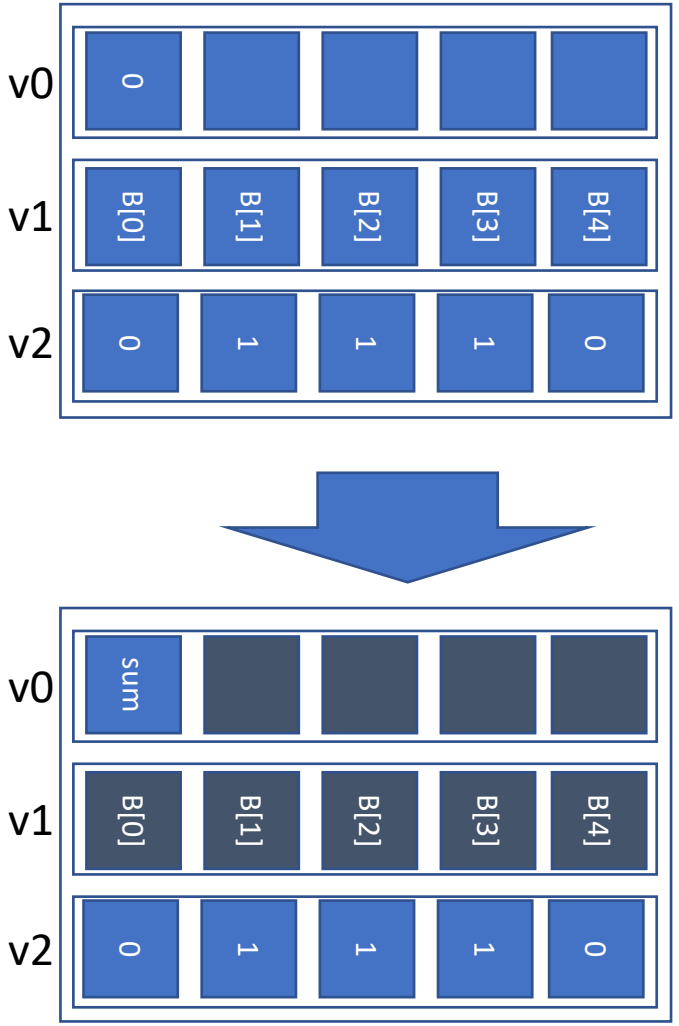
Behavior of a masked vector operation: For elements up to vl in v3, add elements from v0 and v2 if that element in v1's LSB is set to 1, set other v3 elems to 0

What high-level programming concept does this get used to implement?



Reduction Operations with a vector mask






```
setvl 5
vld v0, a
vld v1, b
vredsum v0, v0, v1, v2
```

input vec

dst

init val

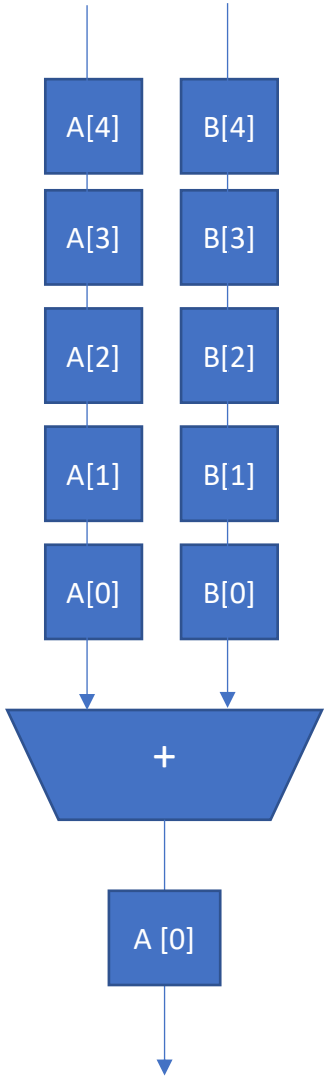
mask

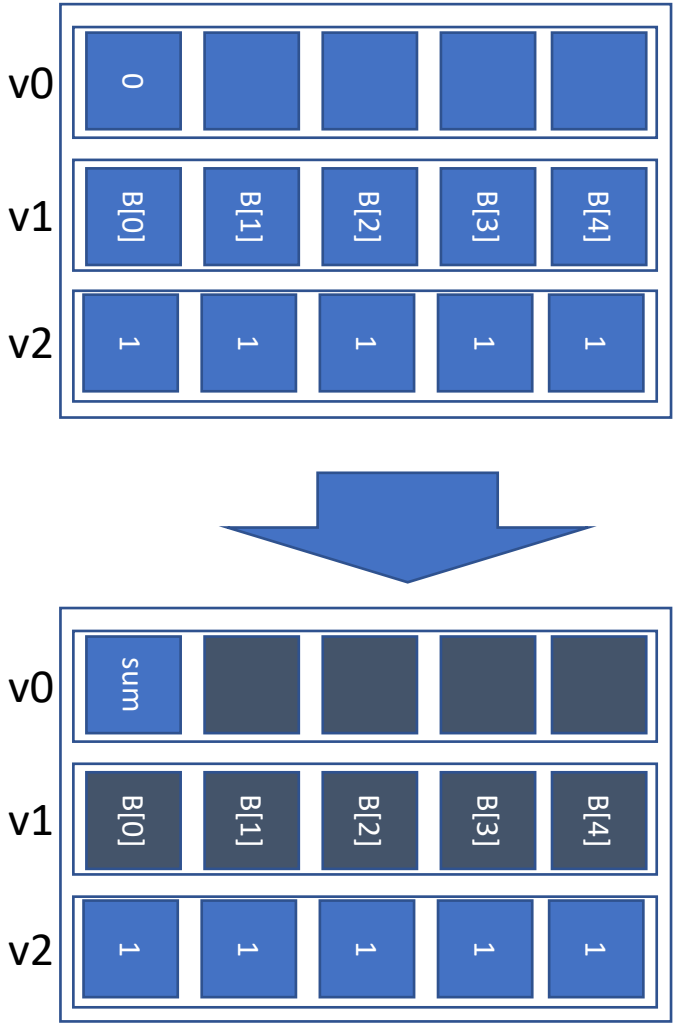
Reduction operations accumulate the result of an operation on a vector into the first element of a destination vector

Uses for reduction?

```
v0[0] =
v0[0] +
v1[1] + v1[2] + v1[3]
```

Reduction Operations with a vector mask



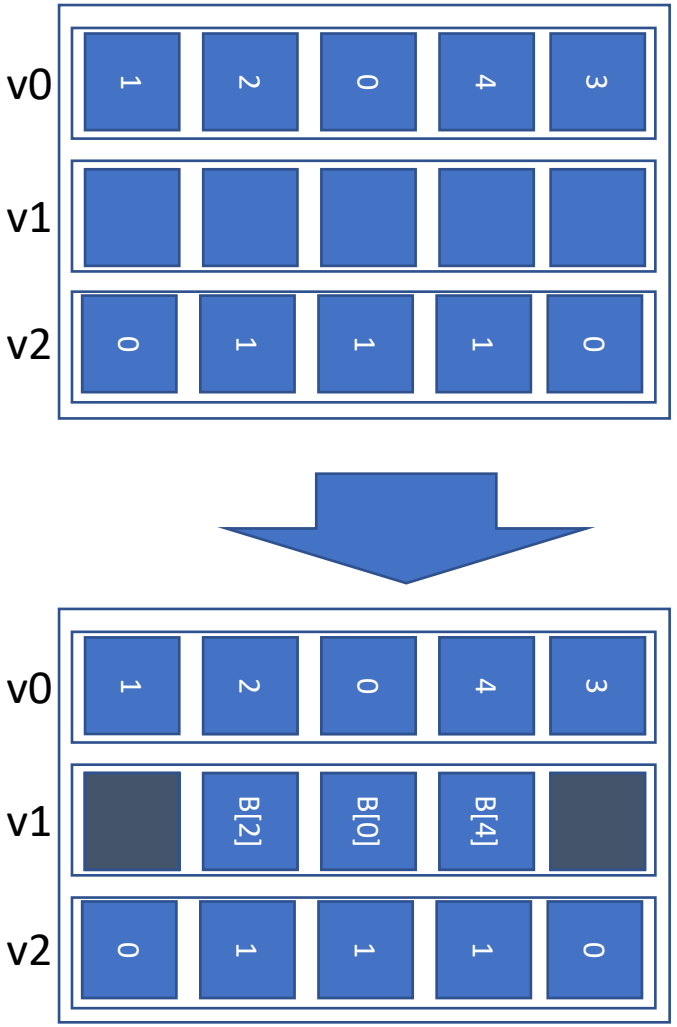


Uses for reduction?
Dot product, e.g.,

```
setvl 5
vld v0, a
vld v1, b
vmul v0, v1
vredsum v0, v0, v1, v2
```

```
for( i = 0..len){
  v[i] += a[i] * b[i]
}
```

Indexed Memory Accesses (Scatter/Gather)



index vector

dest

```
vluxei64 v1, (&B), v0, v2
```

base addr

mask

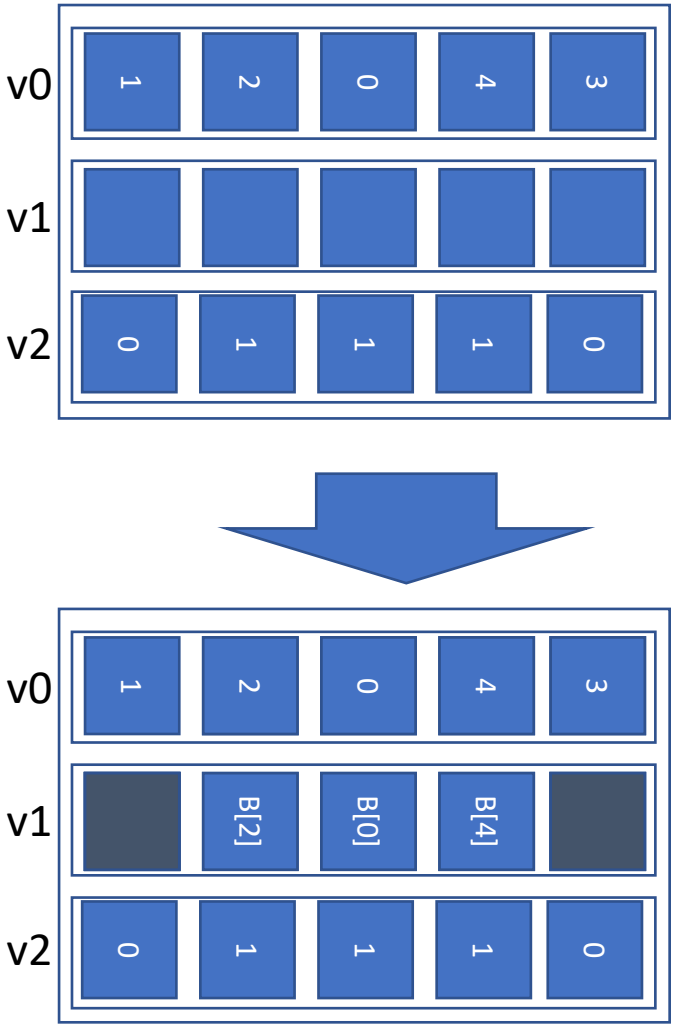
Indexed memory loads **"gather"** elements from all over memory into a contiguous vector register.

Indexed memory stores **"scatter"** elements from a contiguous vector register into locations all over memory

Uses?

```
v1[i] = v2[i] ? B[v0[i]] : v1[i]
```

Indexed Memory Accesses (Scatter/Gather)



dest

index vector

```
vluxei64 v1, (&B), v0, v2
```

base addr

mask

Common Use: indirect array accesses. Common in graph analytics

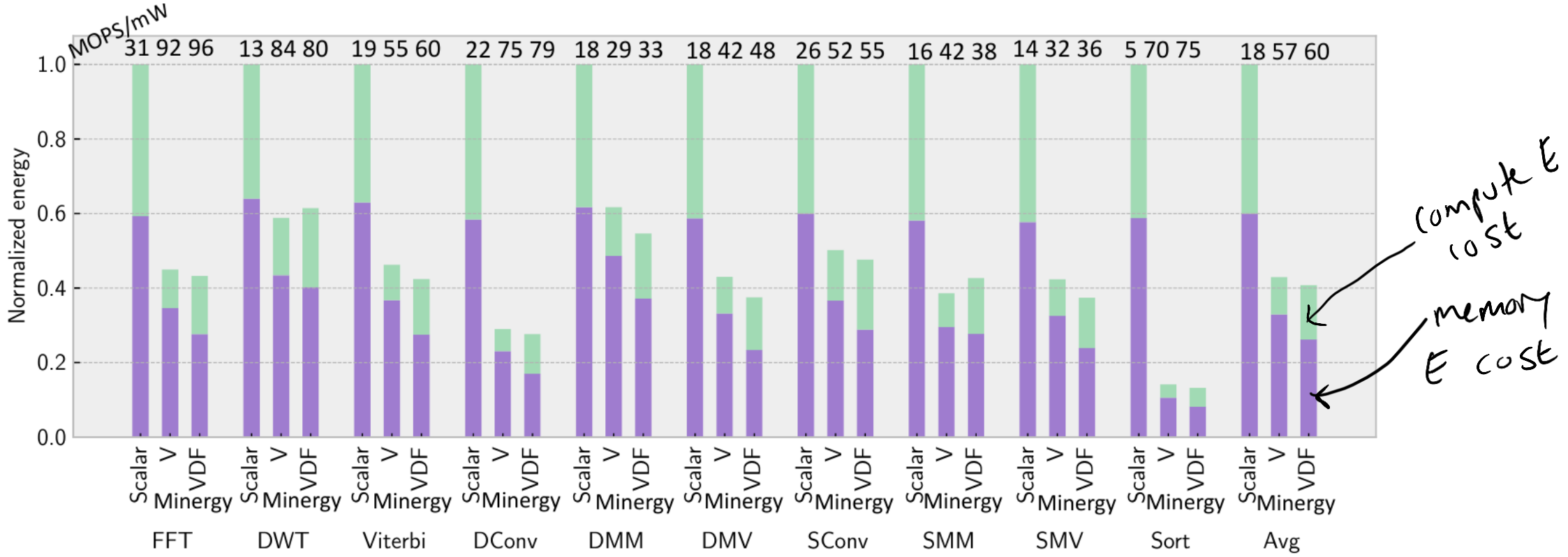
```
for( src in 0 .. n ){
  for( dst in 0..ind[src].len() ){
    data[ ind[src][dst] ]++;
  }
}
```

```
v1[i] = v2[i] ? B[v0[i]] : v1[i]
```

Summary of Benefits: Vector Architectures

- **Compared to scalar architectures:**
  - **Single instruction performs many operations:** one instruction is the equivalent of executing an entire loop of a program!
  - **Control is simpler:** no loops, no branches, no misprediction/misspeculation
  - **Vector interface makes data-independence across vector elements explicit:** simplifies implementations and eliminates complex dependence logic
  - **Dependence checking of vectors, not elements:** what dependence tracking is required pertains to entire vector registers, not individual elements, amortizing its cost significantly
  - **Easy to express data parallelism:** avoids software complexity of multithreading on a multiprocessor (i.e., MIMD)
  - **Maximize value of memory bandwidth:** contiguous/strided vector fetch operations are a good match for highly-banked memories
  - **Energy efficiency**: instruction & data fetch amortize costs across vector saving energy
  - **Require vector programming style, which means changing all of your code. Code doesn't match vector style well? Can't use the vector architecture without lots of extra work!**

Vector execution model saves energy (and time) over scalar processing



Taken from a very recent research project about optimizing for minimum energy by using a new vector processor (**V** bars in the plot) and a customized variant (**VDF** bars in the plot). **V/VDF** use RISCV vector insns., **scalar** plain RISCV insns.

Key take-away: vector processing cuts energy by ***more than half*** compared to scalar processing.

What did we just learn?

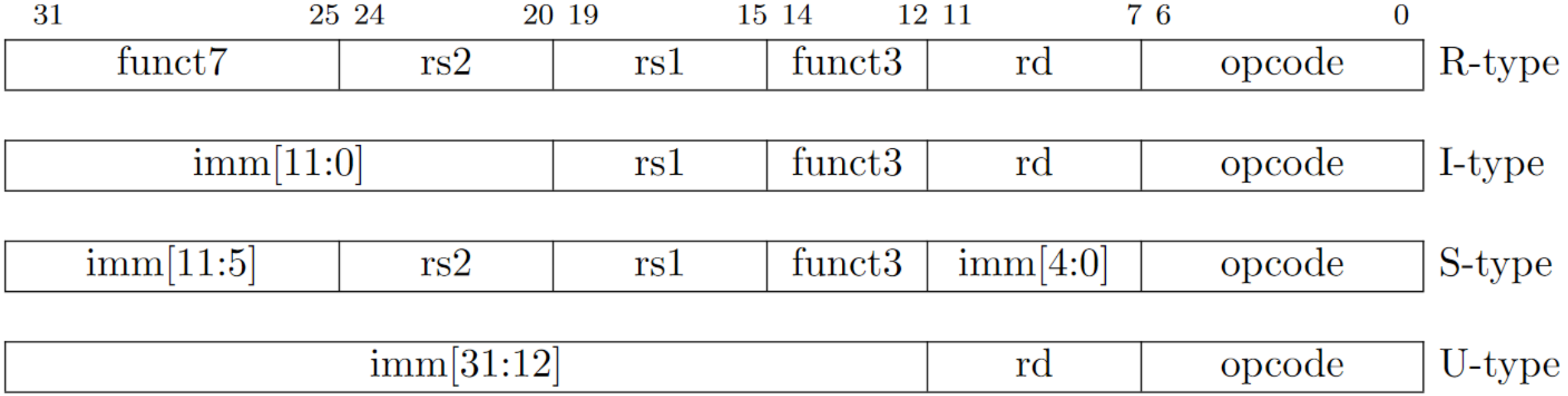
- We learned about how VLIW and Vector processing are two different takes on the hardware software boundary that admit more parallelism than SS/OoO's ILP focus allows
- VLIW did not take over, vector has been a consistent background hum
- Both approaches require the programmer and the compiler to make big changes to code to work well with these new hardware/software interfaces.

What to think about next?

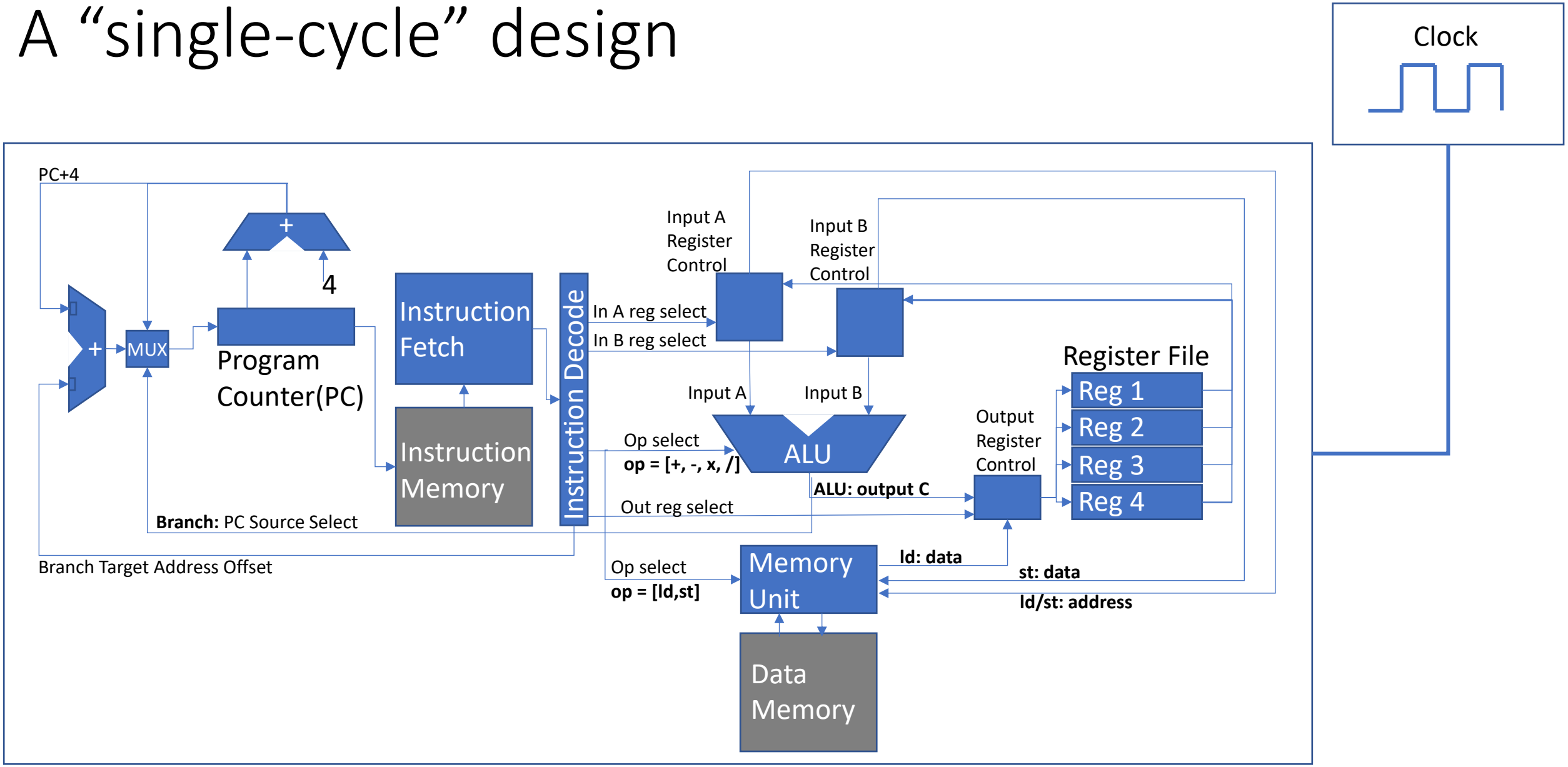
- Lab 3 out Thursday
- Next we look at Virtual Memory as an abstraction
- Also look at the underlying mechanisms and options for implementing virtual memory in a modern CPU

RISCV-RV32I Specification

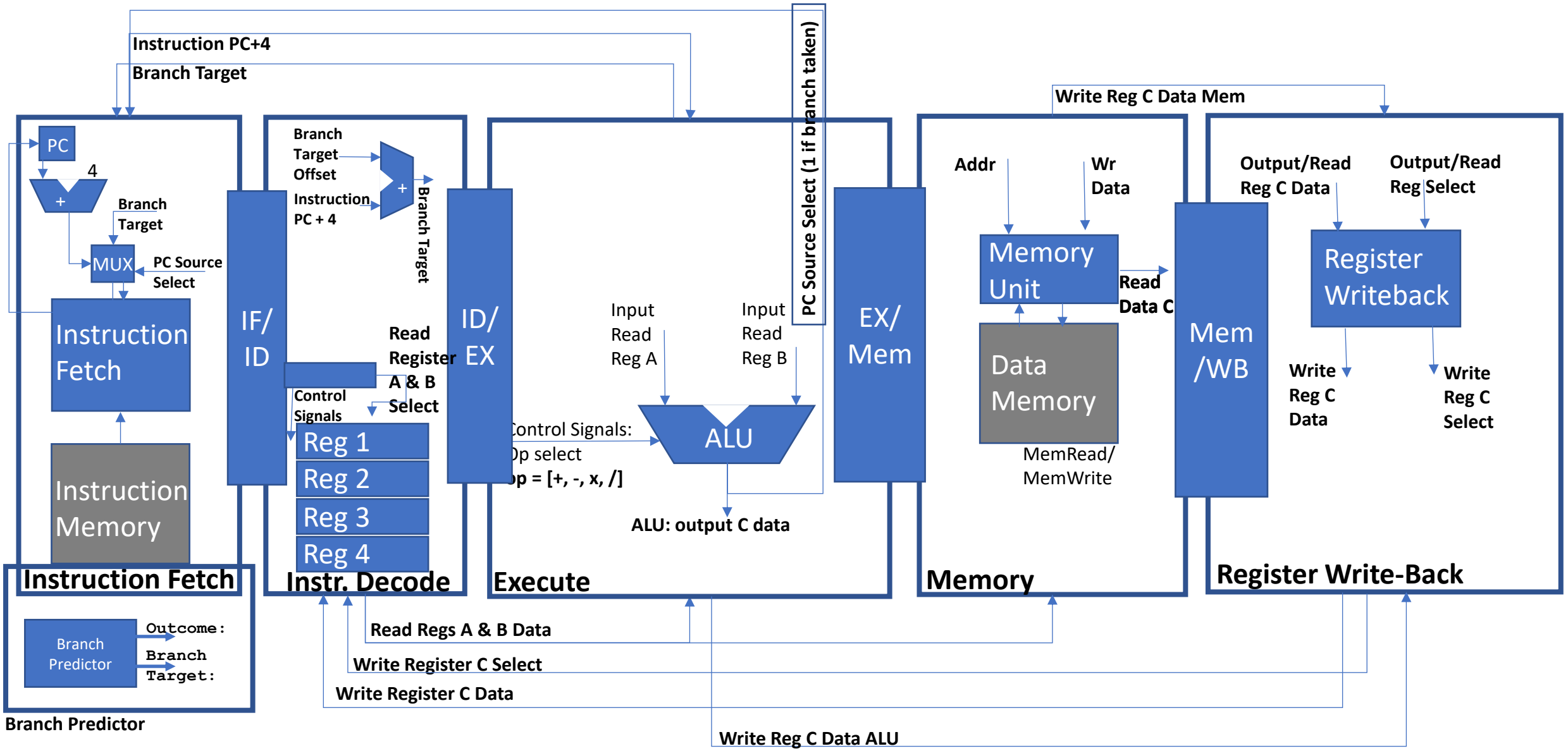
- Four base instruction encoding formats
  - R(egister), I(mmediate), S(tore), U(pper Immediate)
  - Mnemonics are non-binding and formats get flexibly used



A “single-cycle” design



Pipeline with Branch Predictor



Types of Data Hazards

```
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
```

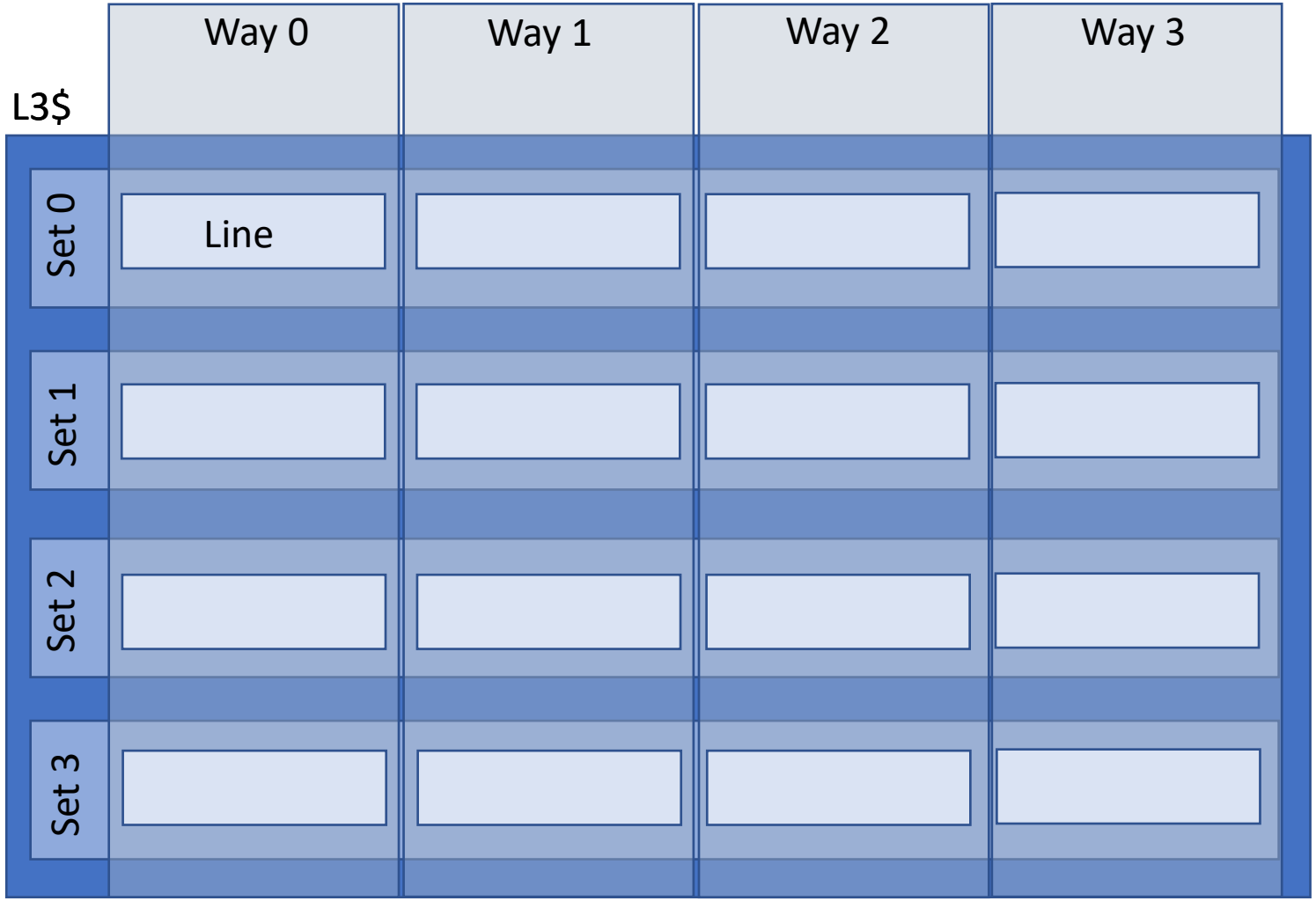
Write-After-Read (WAR)

```
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

How many bits in tag/index/offset?




```
lb x6 0x7fff0053
```

set index

0x01111111111111110000000010100 11

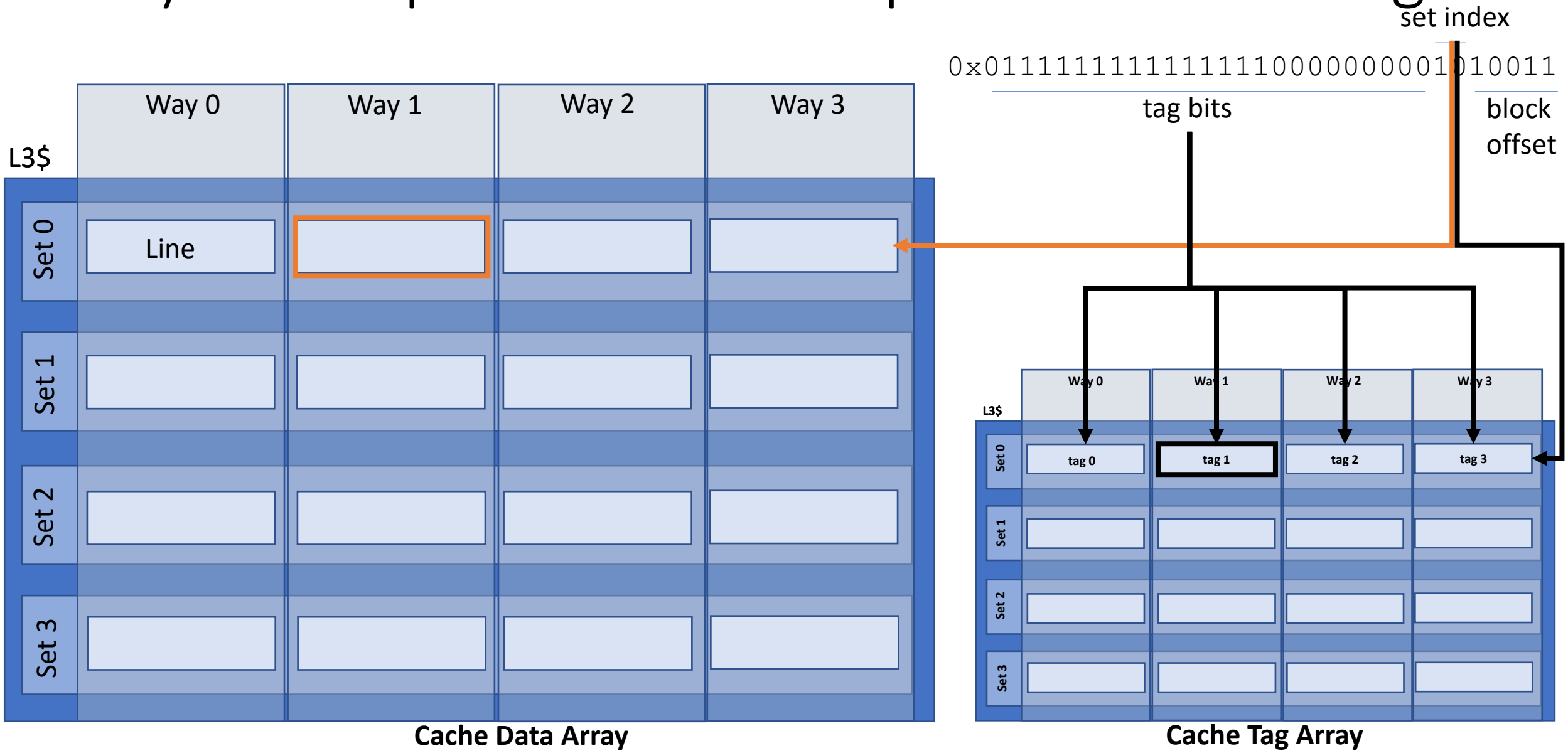
tag bits | block offset

Enough **block offset** bits to count block bytes
Enough **set index** bits to count the ways
All left-over bits are **tag** bits
Question: what do tag bits mean?

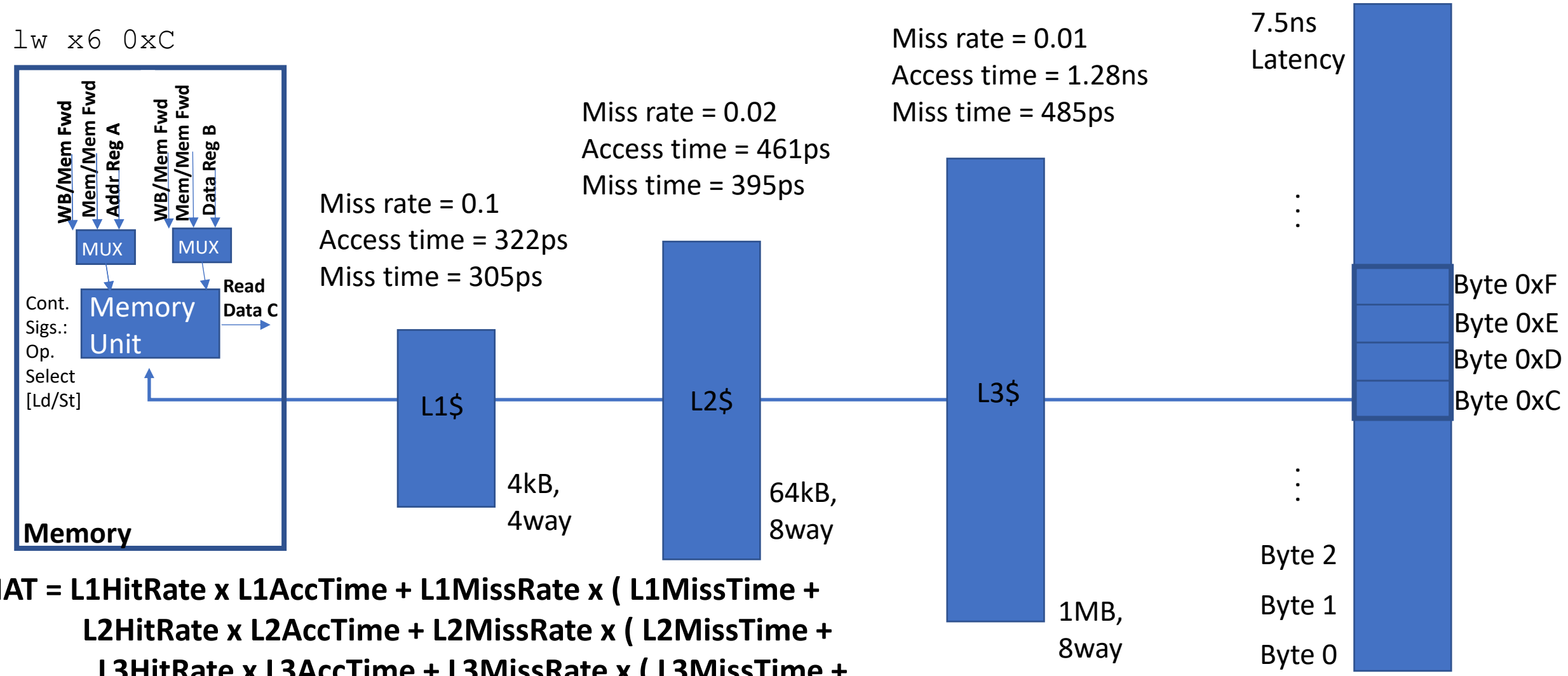
Valid	Dirty	Tag	32 bytes data

Total cache size = 32B x 4 sets x 4 ways = 512B

Physical implementation separates data & tags

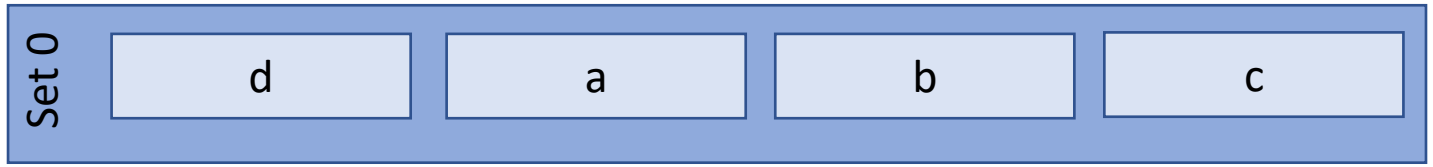


Average Memory Access Time (AMAT): Measuring the performance of a memory hierarchy



AMAT = L1HitRate x L1AccTime + L1MissRate x (L1MissTime +
L2HitRate x L2AccTime + L2MissRate x (L2MissTime +
L3HitRate x L3AccTime + L3MissRate x (L3MissTime +
DRAM Latency)))

Belady's MIN Algorithm for Optimal Replacement

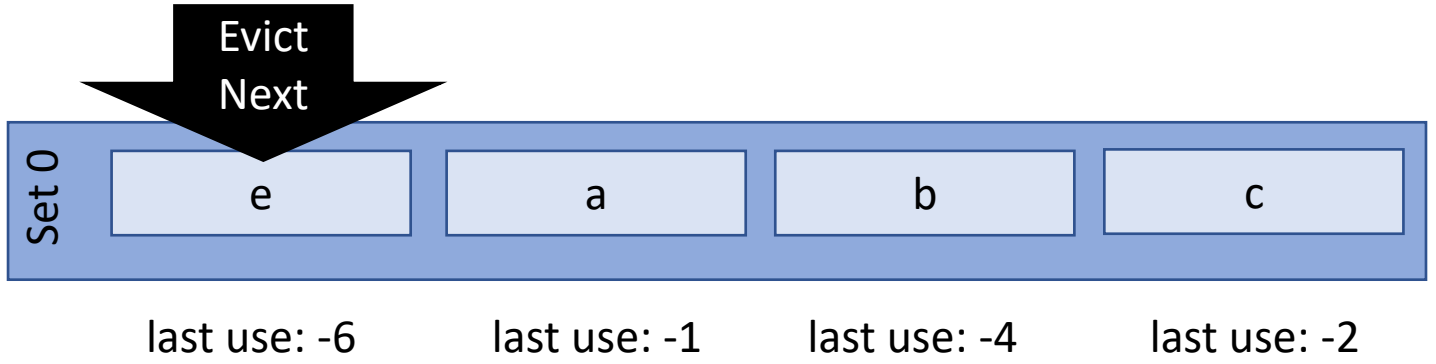



```
lb x6 0xe   Miss
lb x6 0xb   Hit
lb x6 0xc   Hit
lb x6 0xd   Hit
lb x6 0xa   Miss
```

Bélády László:
"What defines optimality for a cache replacement algorithm?"
Evict the cached element that will be used furthest in the future.

Least-Recently Used (LRU) Replacement

Evict the block that was used the furthest in the execution's past



LRU's Gamble: "Haven't used block 0xe for longest, probably won't use it again any time soon, either"

```
lb x6 0xe
lb x6 0xa
lb x6 0xb
lb x6 0xa
lb x6 0xc
lb x6 0xa
lb x6 0xd
lb x6 0xa
lb x6 0xe
lb x6 0xa
lb x6 0xb
lb x6 0xe
```

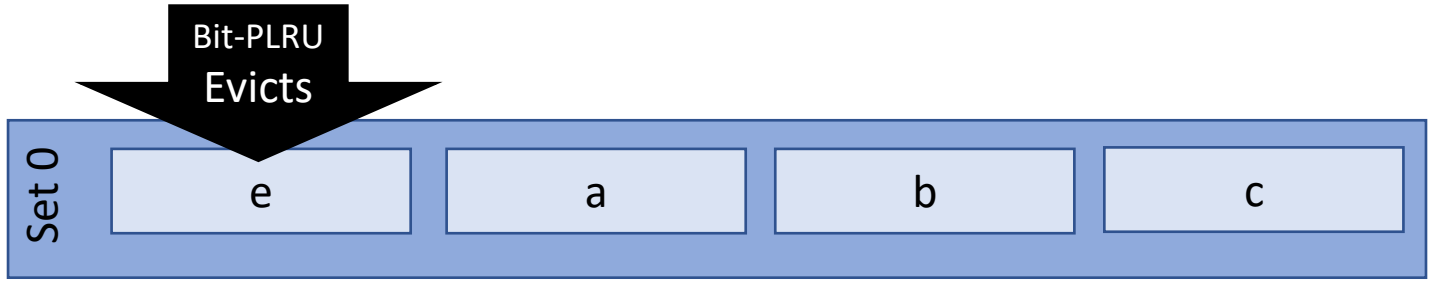
knowable

guessable

If a block was **not** used recently, it will **not** be used again soon

Bit-Pseudo-Least-Recently Used (Bit-PLRU)

Evict a block that was definitely not most recently used



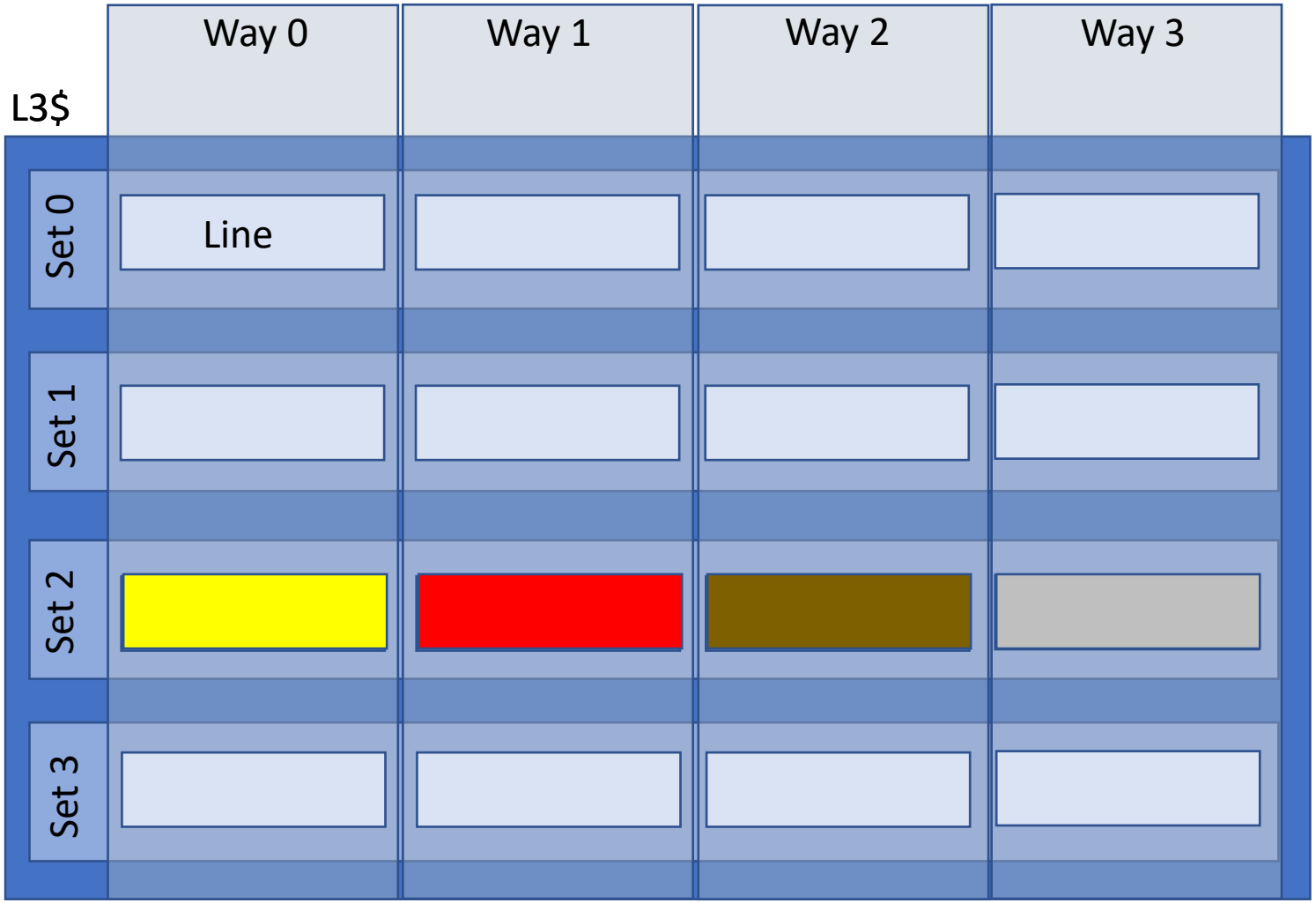

```
lb x6 0xe
lb x6 0xa
lb x6 0xb
lb x6 0xa
lb x6 0xc
lb x6 0xa
lb x6 0xd
lb x6 0xa
lb x6 0xe
lb x6 0xa
lb x6 0xb
lb x6 0xe
```

knowable

guessable

Set MRU bit when block is used (most recently), clear all MRU bits when all MRU bits are set, evict the left-most block with unset MRU bit

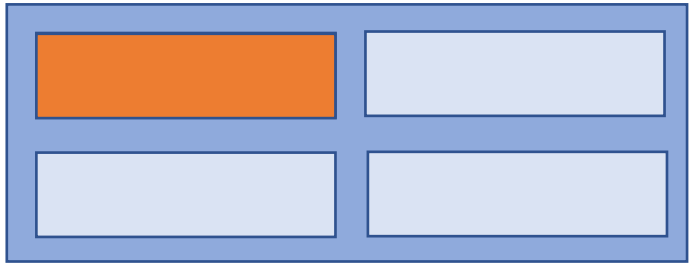
Victim Caches/Buffers



Block evicted from cache goes into (usually fully associative, small) victim buffer.

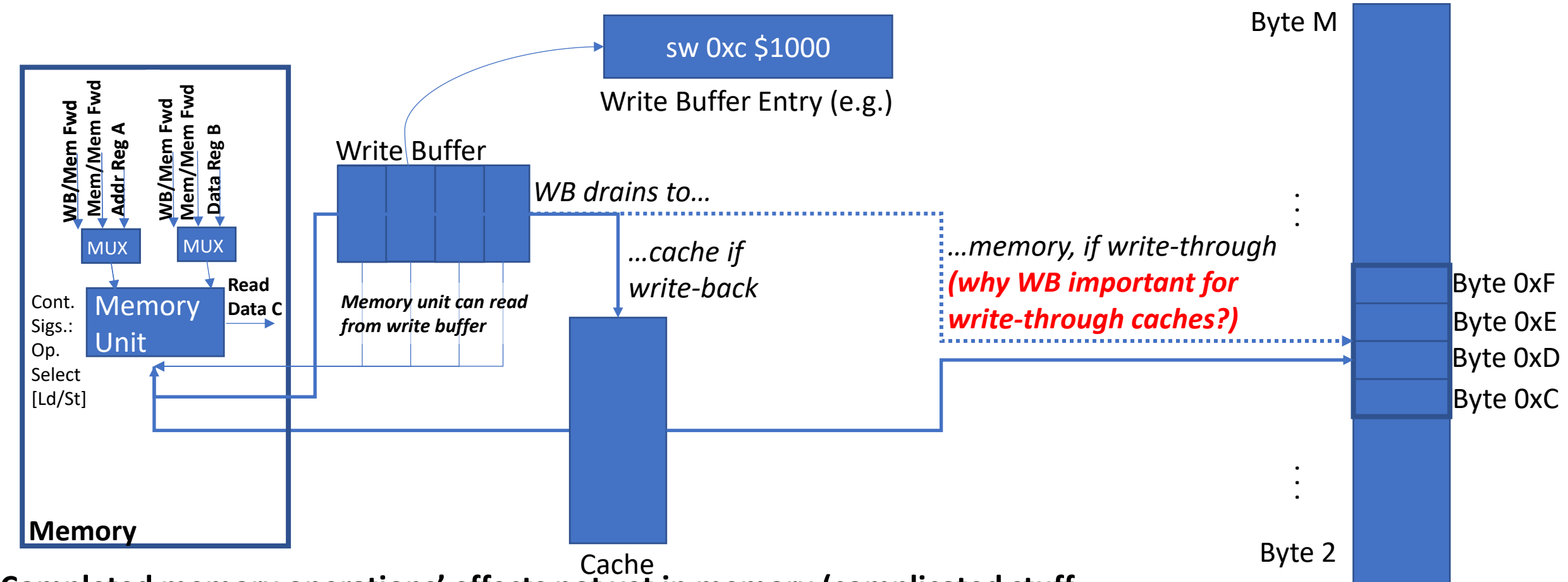
On next access, "victim" can be re-cached without going down the hierarchy.

What problem does a victim cache solve?



Victim Cache

Non-blocking Writes & Write Buffering

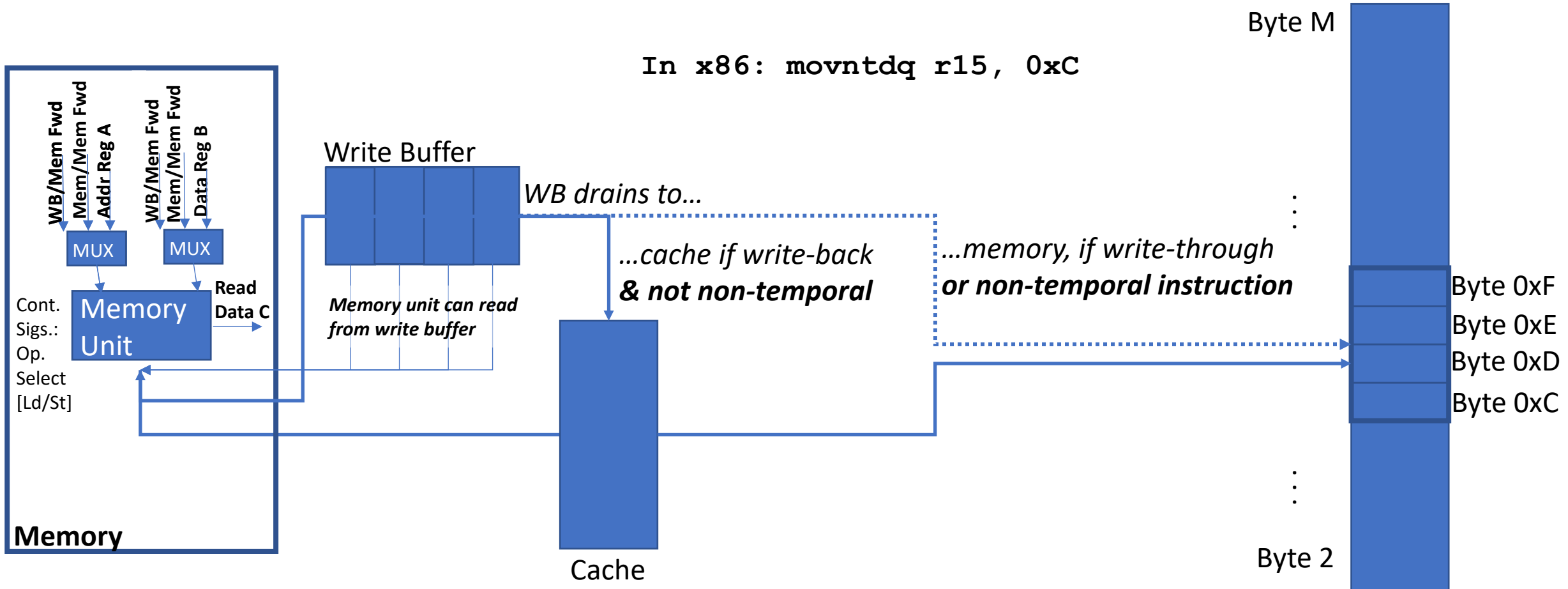


Completed memory operations' effects not yet in memory (complicated stuff, later in the semester…)

What is the latency of a write if it ends up buffered?

***Unpredictable write completion latency*. Need ordering logic.**

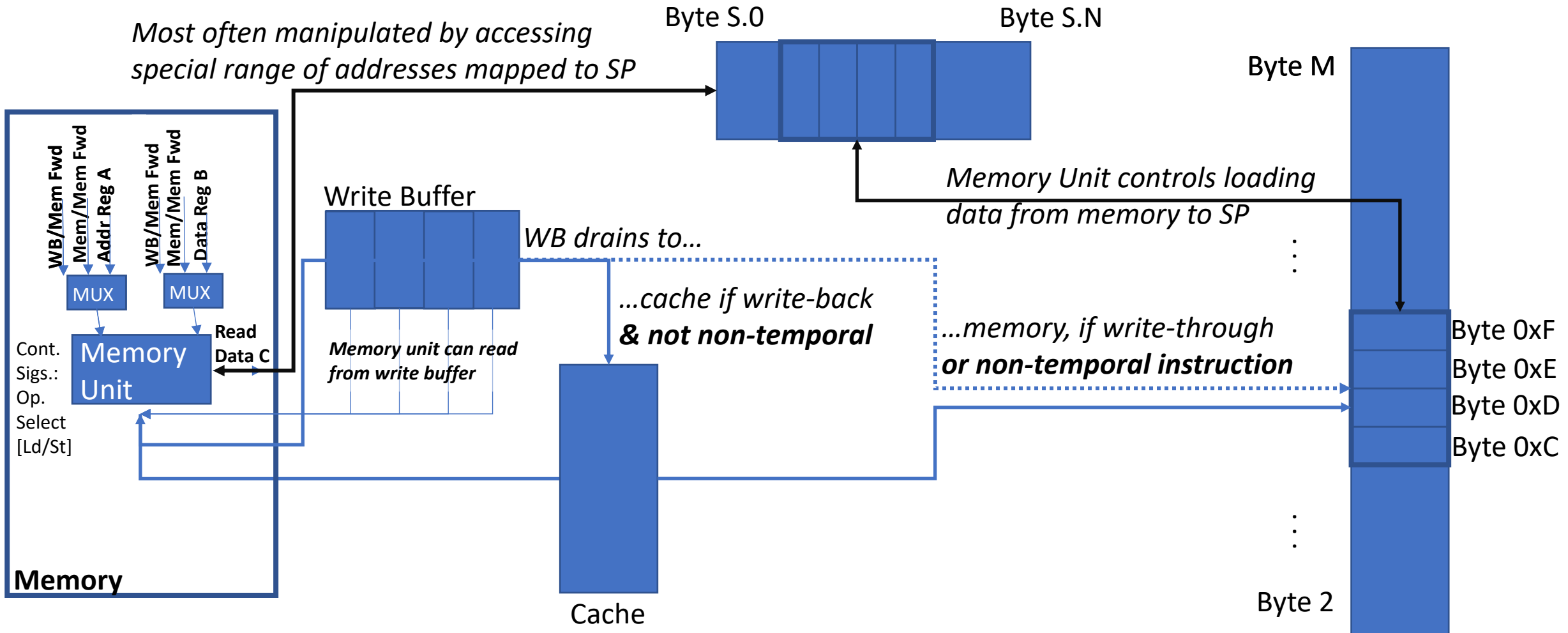
Non-temporal/Streaming Stores



Non-temporal Store: Acts like no-write-allocate+write-through even if write-allocate+write-back for rest of cache.

When would you use a non-temporal store instruction?

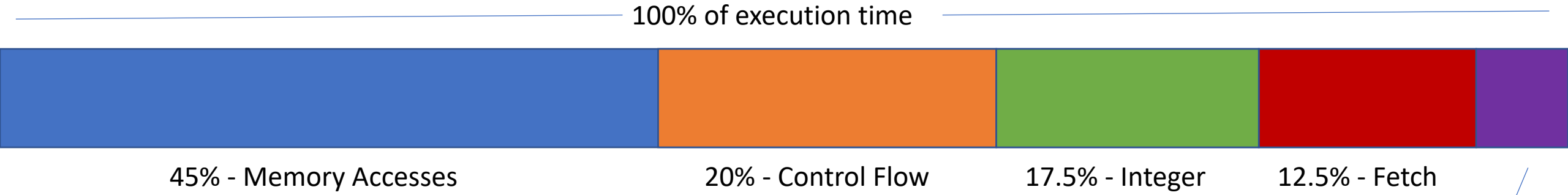
Scratchpad Memories



Most important thing about scratchpads:

Software control is as good (or bad) as the programmer.

Amdahl's Law



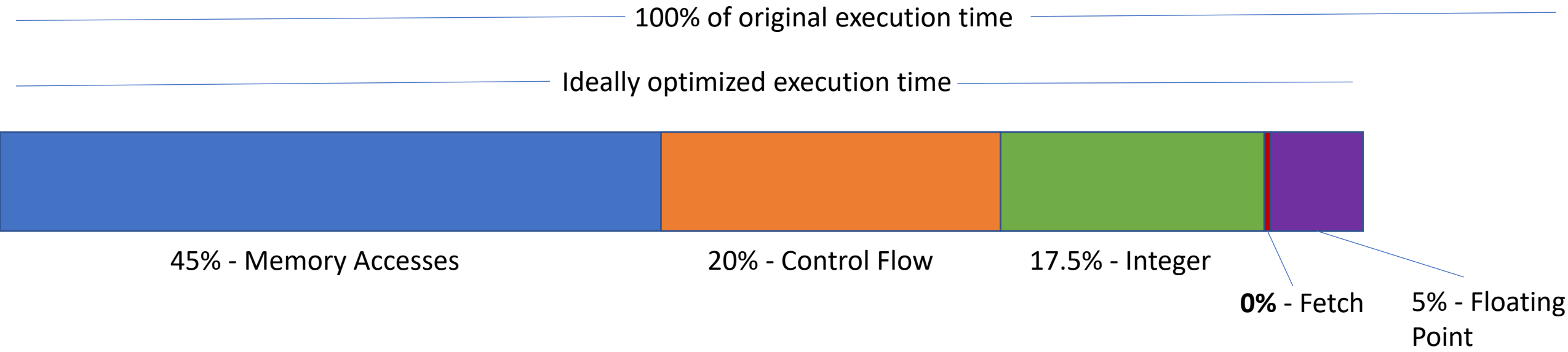
Amdahl's Law:

optimized time = [1-p x time / 1.0] + [p x time / speedup]

Or equivalently:

speedup = 1 / [(1 – p) / 1.0 + p / speedup]

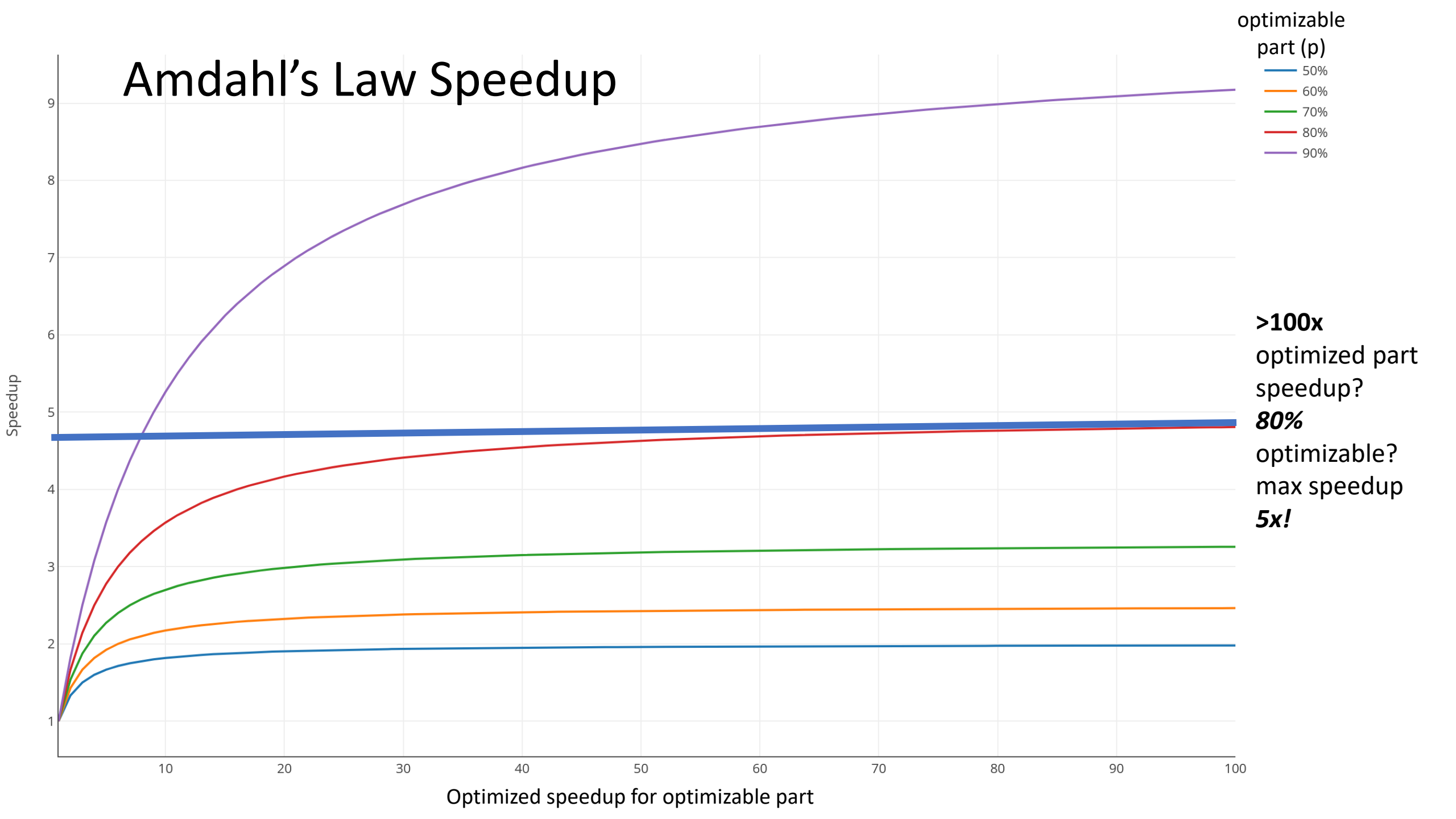
Limit Cases for Amdahl's Law



Amdahl's Law with *infinite* speedup:

optimized time = [1-p x time / 1.0] + [p x time / *infinity*]

Amdahl's Law Speedup

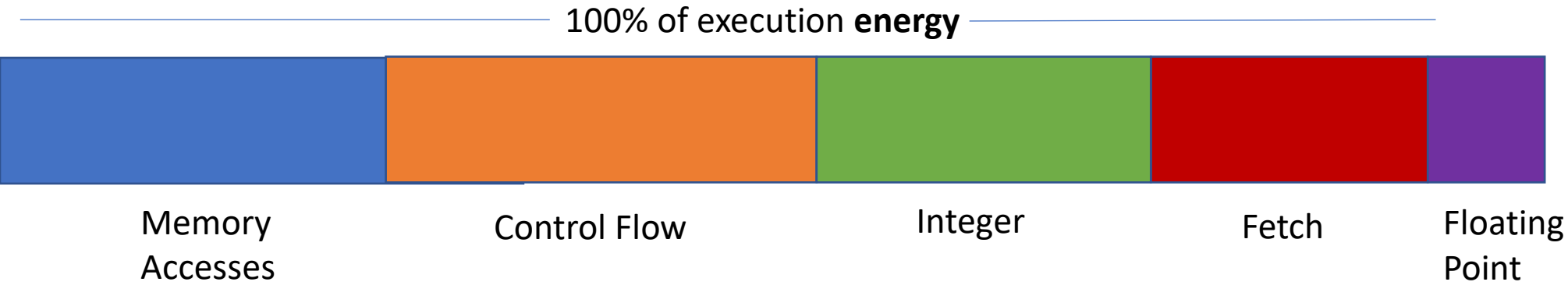


>100x optimized part speedup?

80% optimizable?

max speedup ***5x!***

Another view of the world: Gustaffson's Law



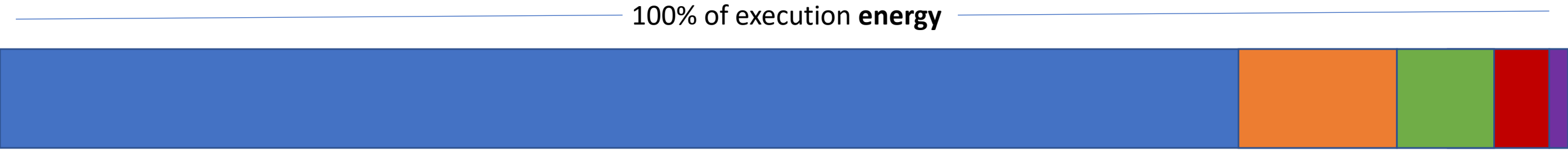
Idea: find an *optimizable* part of your system and make it *bigger*

Here, we have already optimized memory by 2x, so we know that memory is optimizable by 2x. Can we do more memory accesses?

Another view of the world: Gustaffson's Law

Q: How to change a system to be bottlenecked by one thing instead of another?

(We will return to this for lab4.)



90% - Memory Accesses

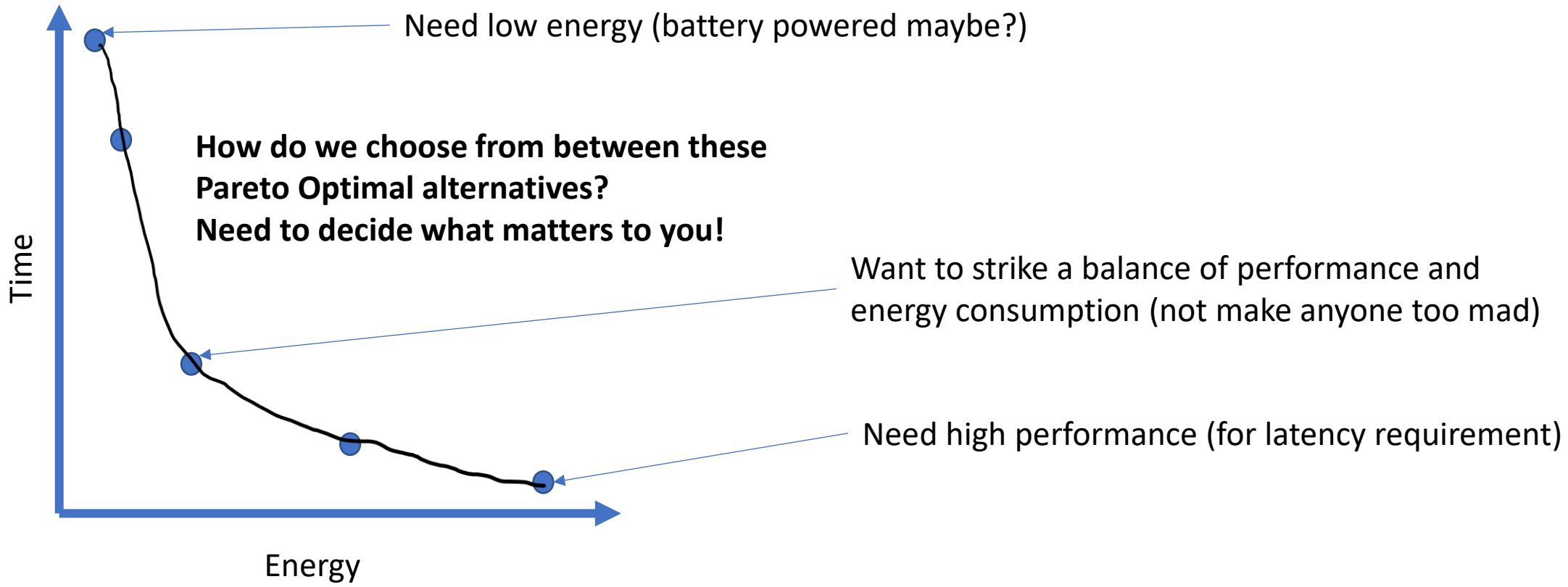
This idea *assumes* you have a clever way to optimize memory

```
data_size = 50
data[data_size] = {…}
if(…){ }
…//18 more of these conditionals
if(…){ }
for d in 0..data_size{ d++ }
```

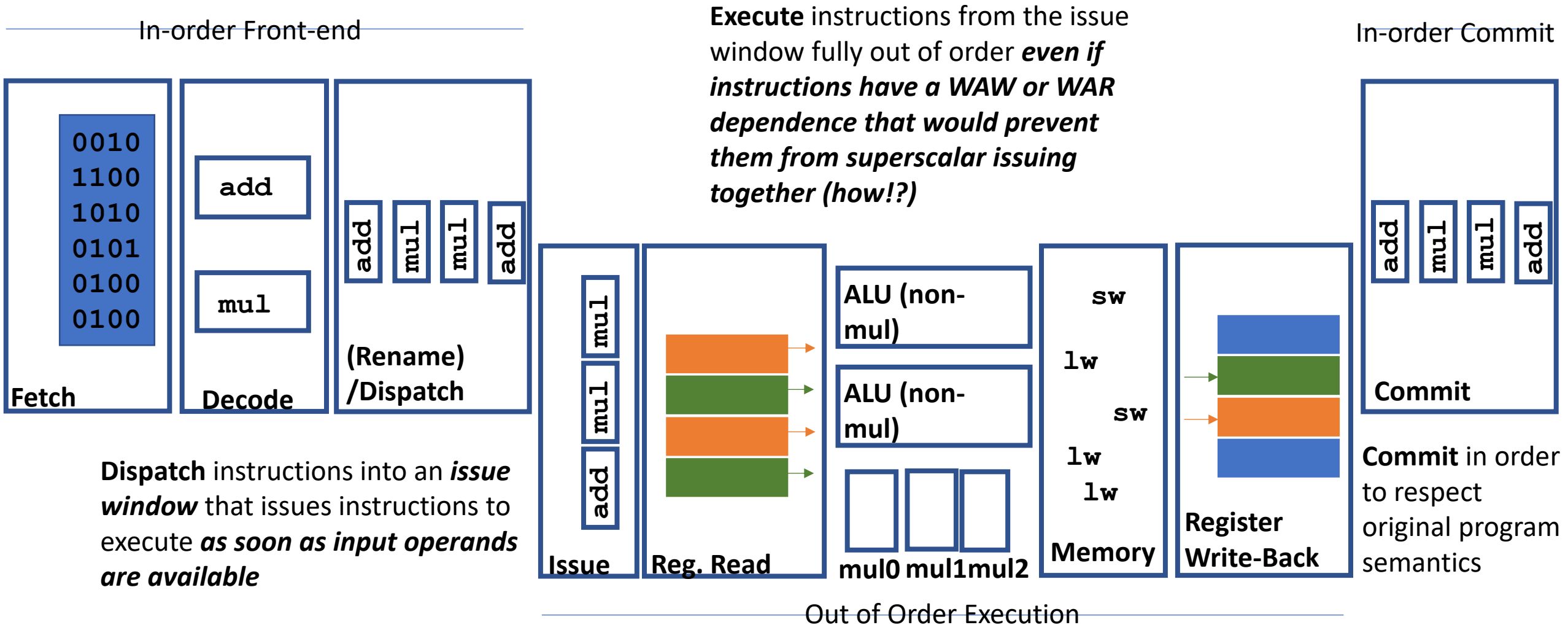
Gustaffson!

```
data_size = 100
data[data_size] = {…}
if(…){ }
…//18 more of these conditionals
if(…){ }
for d in 0..data_size{ d++ }
```

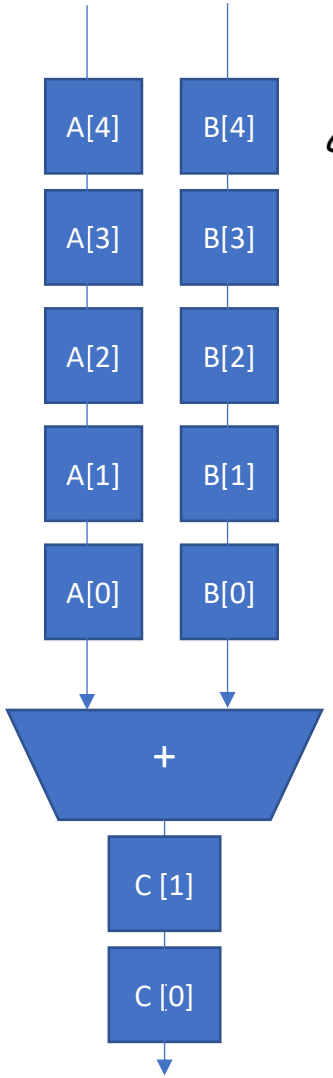
Design Consequence of Pareto Optimality



Out of Order Execution

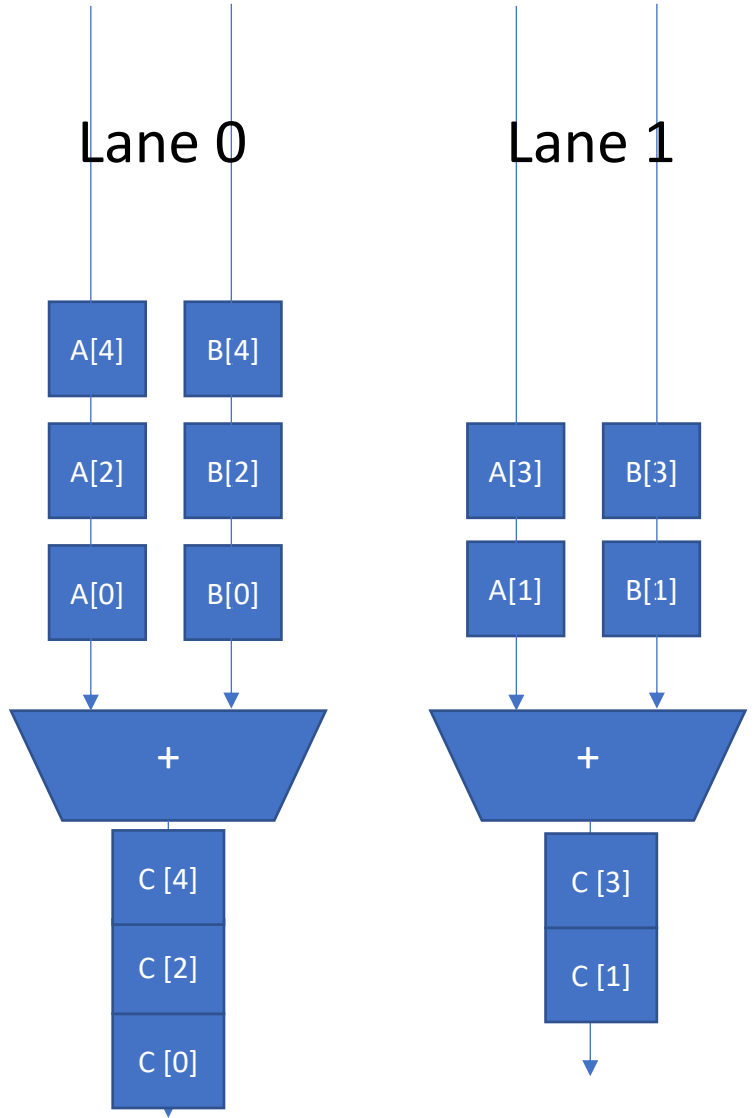


Vector Machines are Easily Parallelizable



Abstraction: execute an instruction's operation over an entire vector of data

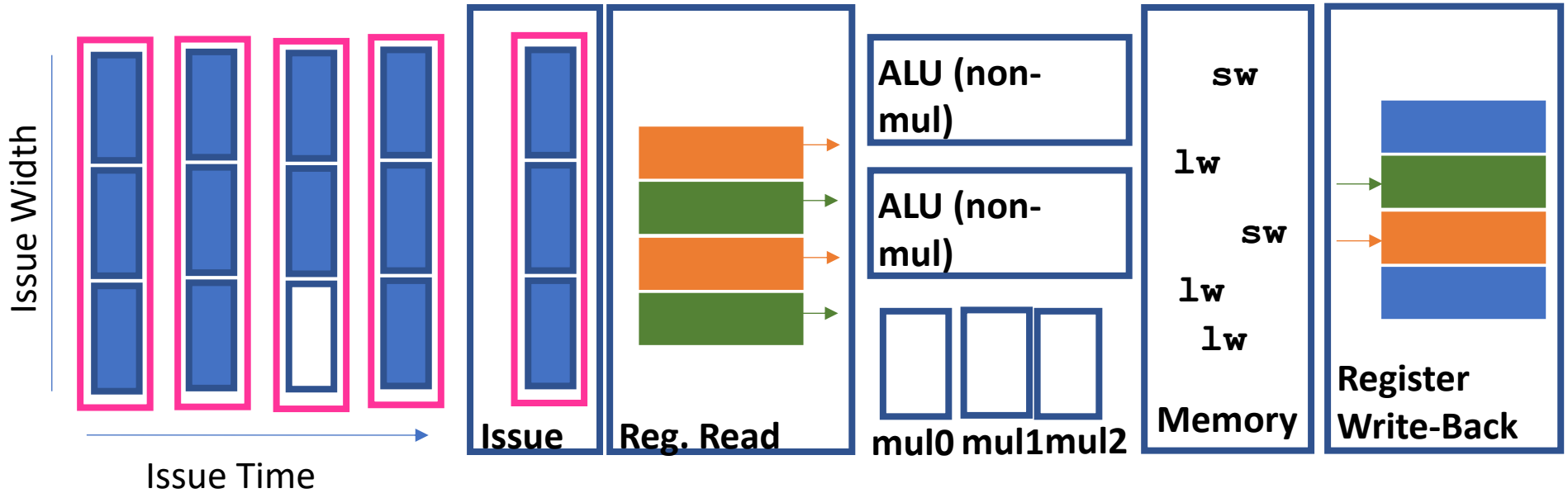
Implementation: Parallel functional units each process parts of a vector, producing a vector output. **Why simple?**



Very Large Instruction Word (VLIW) and the EPIC Architecture (Explicit Parallel Instruction Computer)

Change the *ISA*! In VLIW, the ISA exposes issue width architecturally. Each fetch / issue is on a *bundle* of instructions that are independent

EPIC/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.