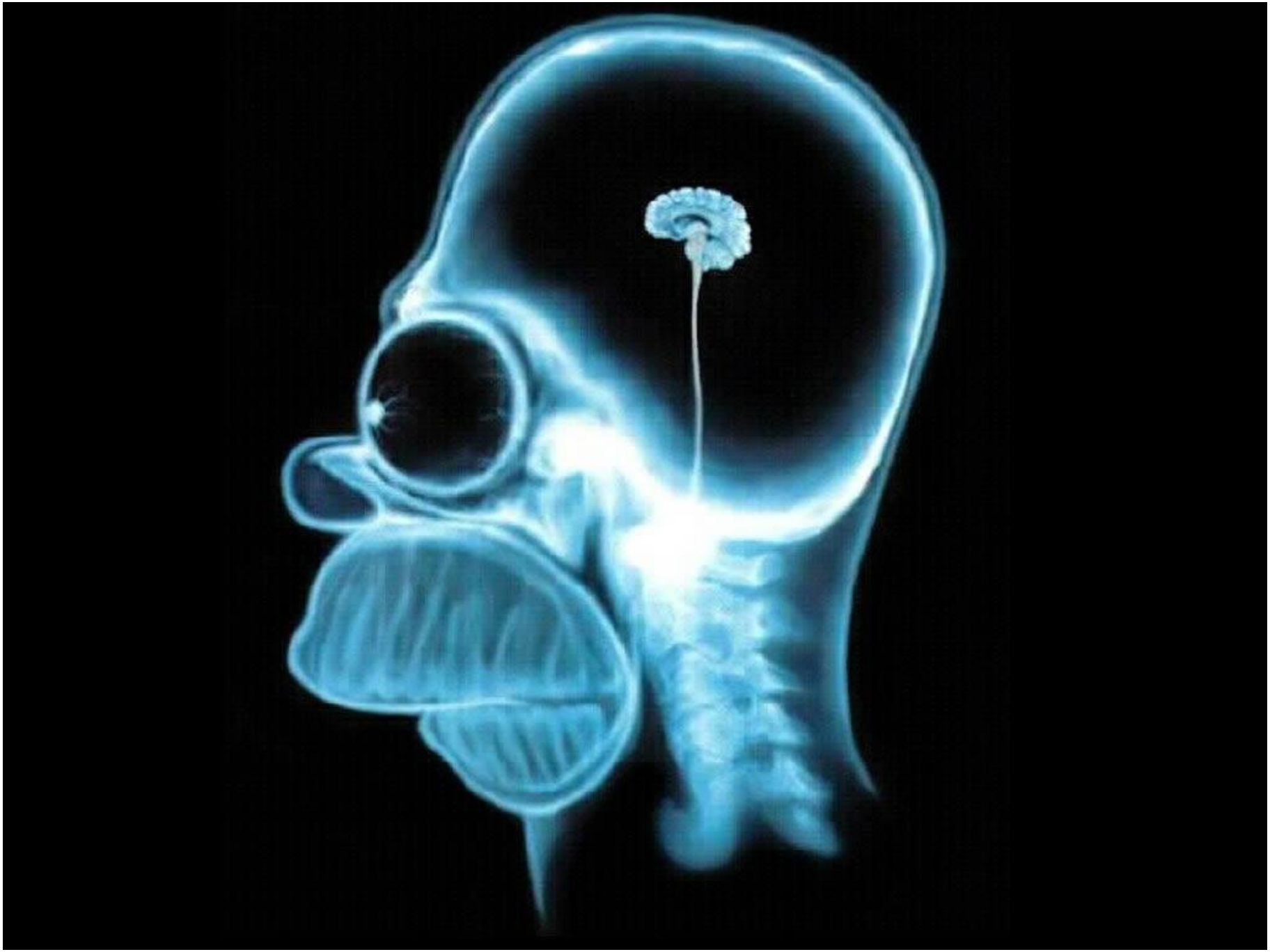
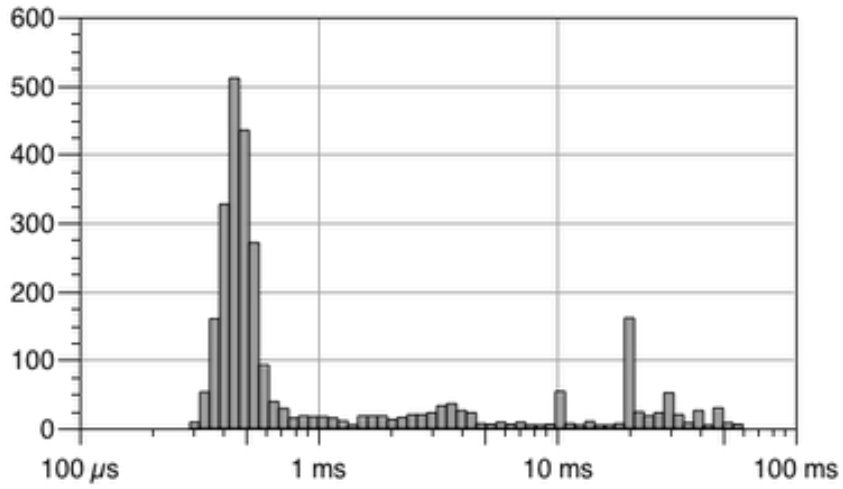
Lock-Free Algorithms In Java

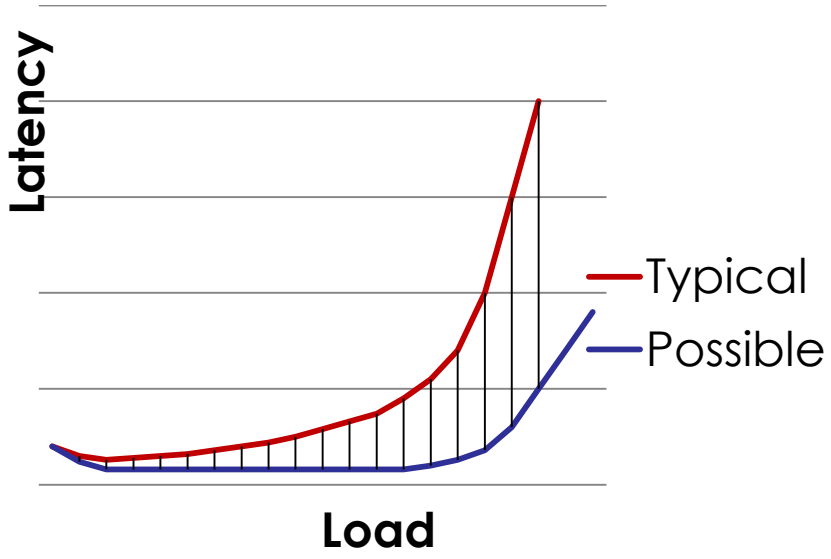
Martin Thompson - @mjpt777



Low-Latency & High-Throughput

- Low and predictable latency
- Consistent behaviour under increasing load
- Ability to handle burst traffic
- Design for the 10X worst case



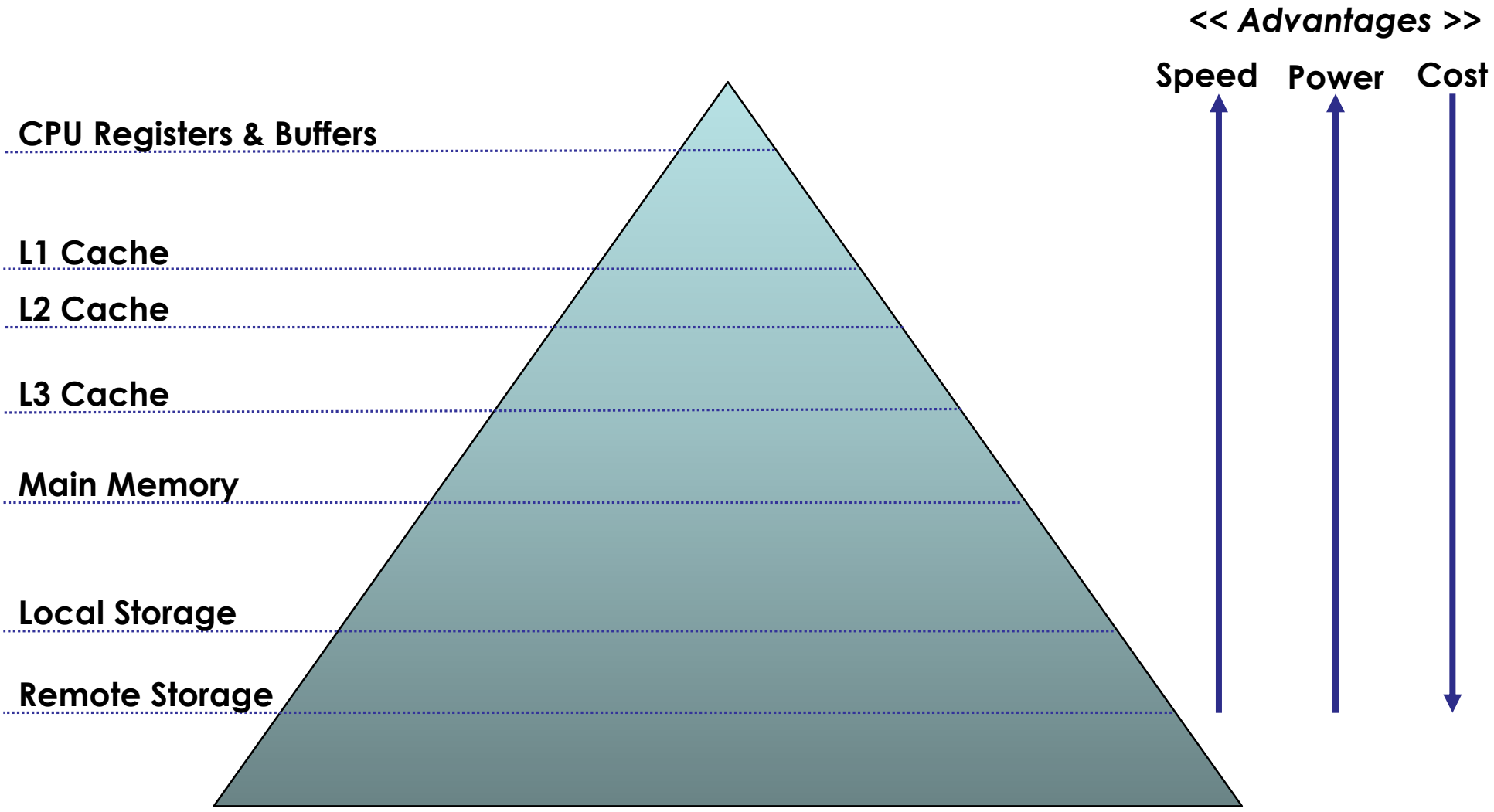


Modern Hardware

"The real design action is in the memory sub-systems – caches, buses, bandwidth, and latency"

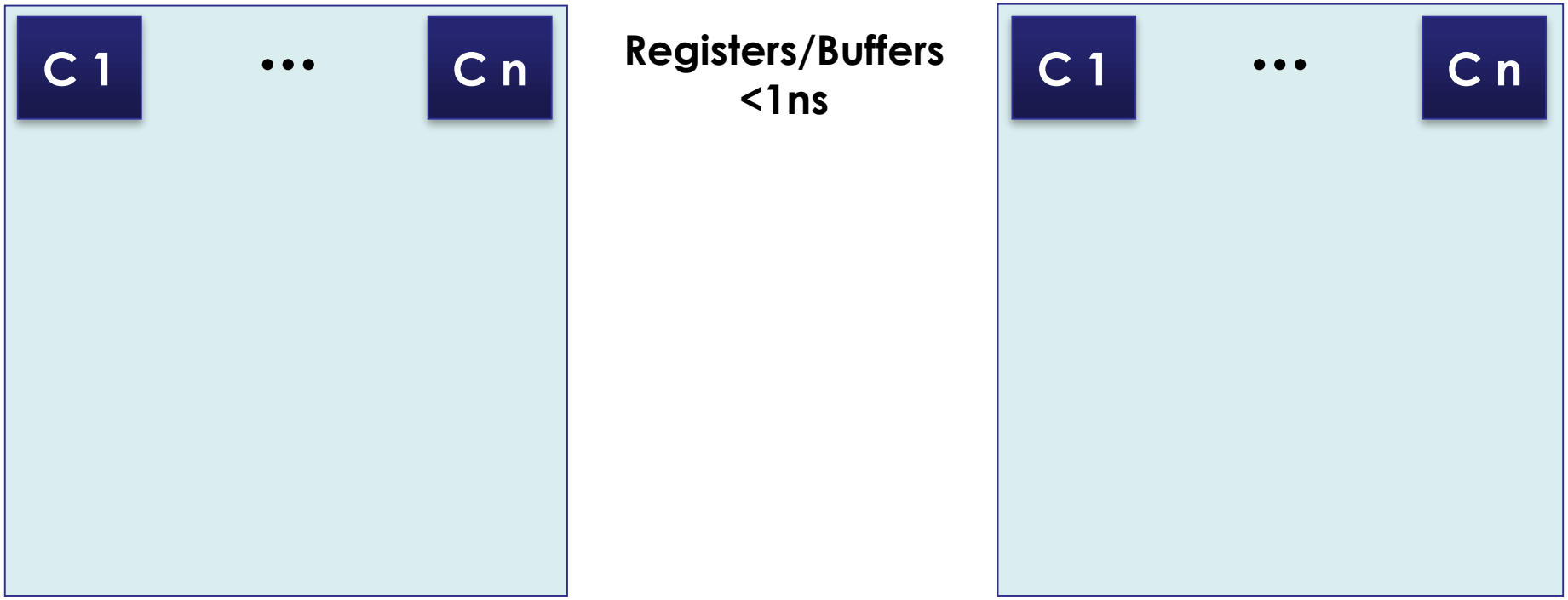
– Richard Sites (DEC Alpha Architect)

The Memory Hierarchy



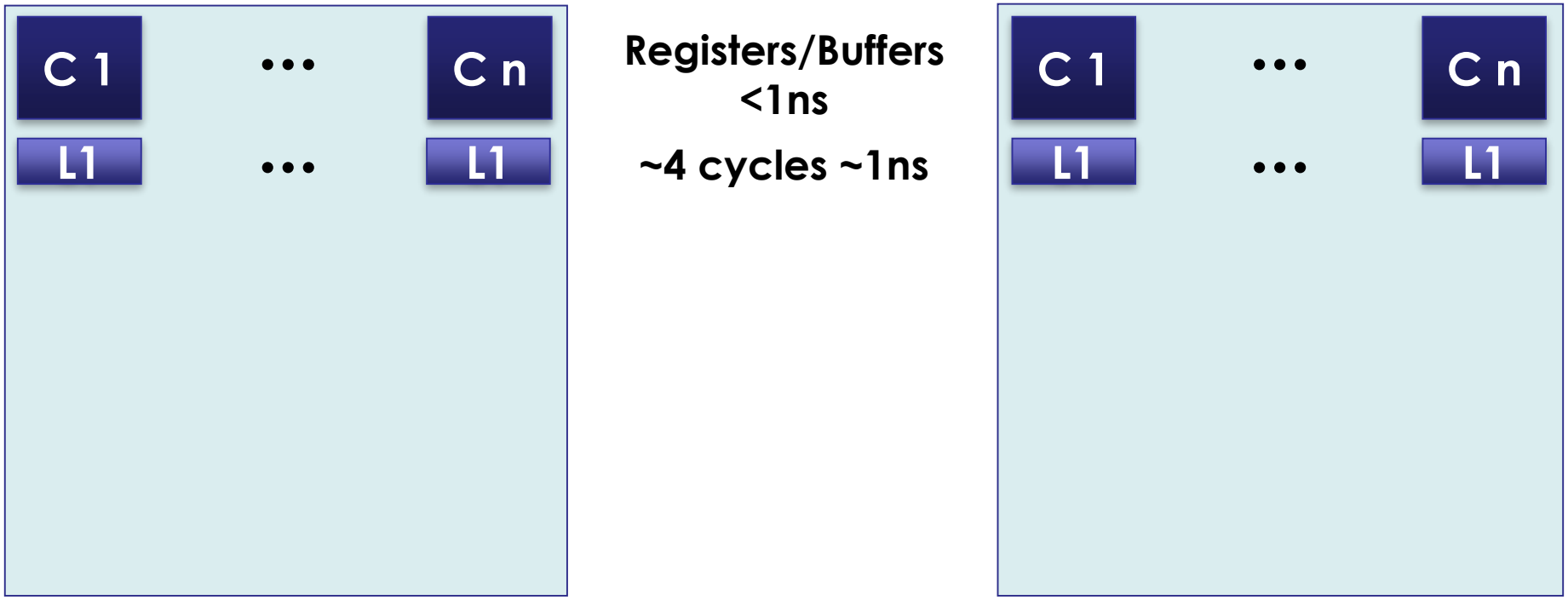
Big Picture

Modern Hardware (Intel Sandy Bridge EP*)



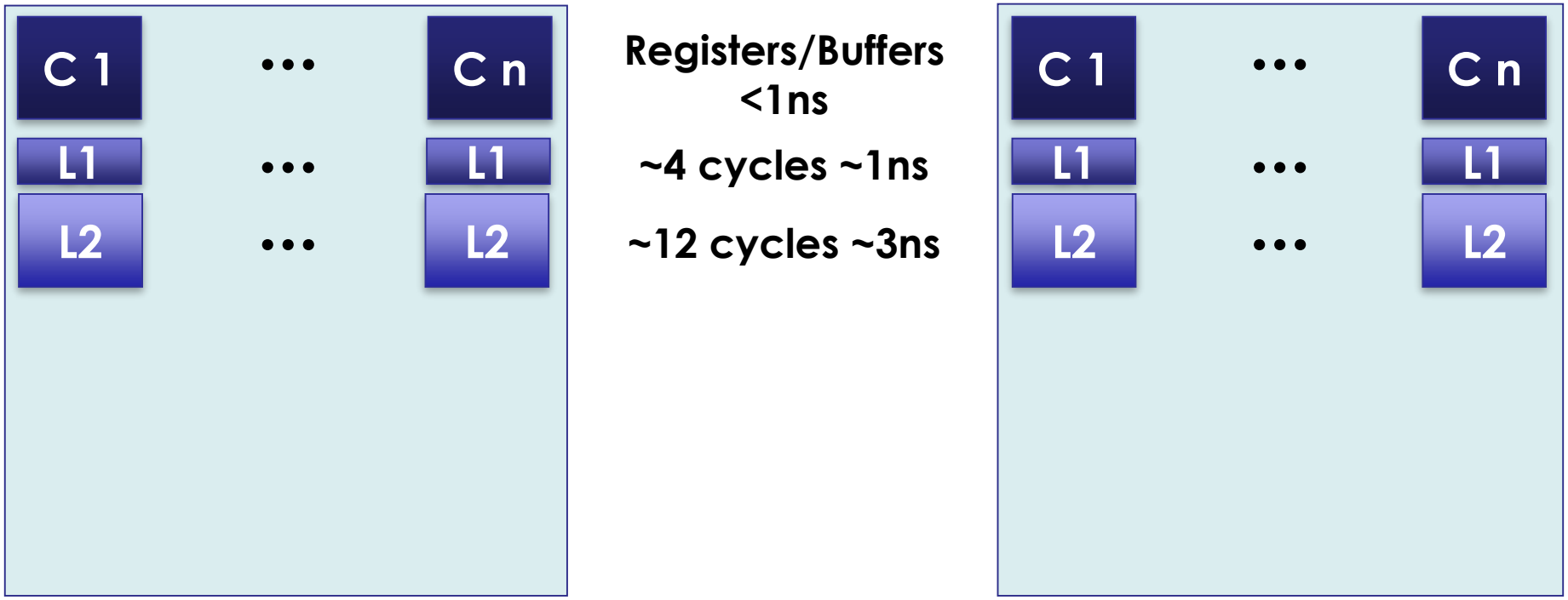
* Assumption: 3GHz Processor

Modern Hardware (Intel Sandy Bridge EP*)



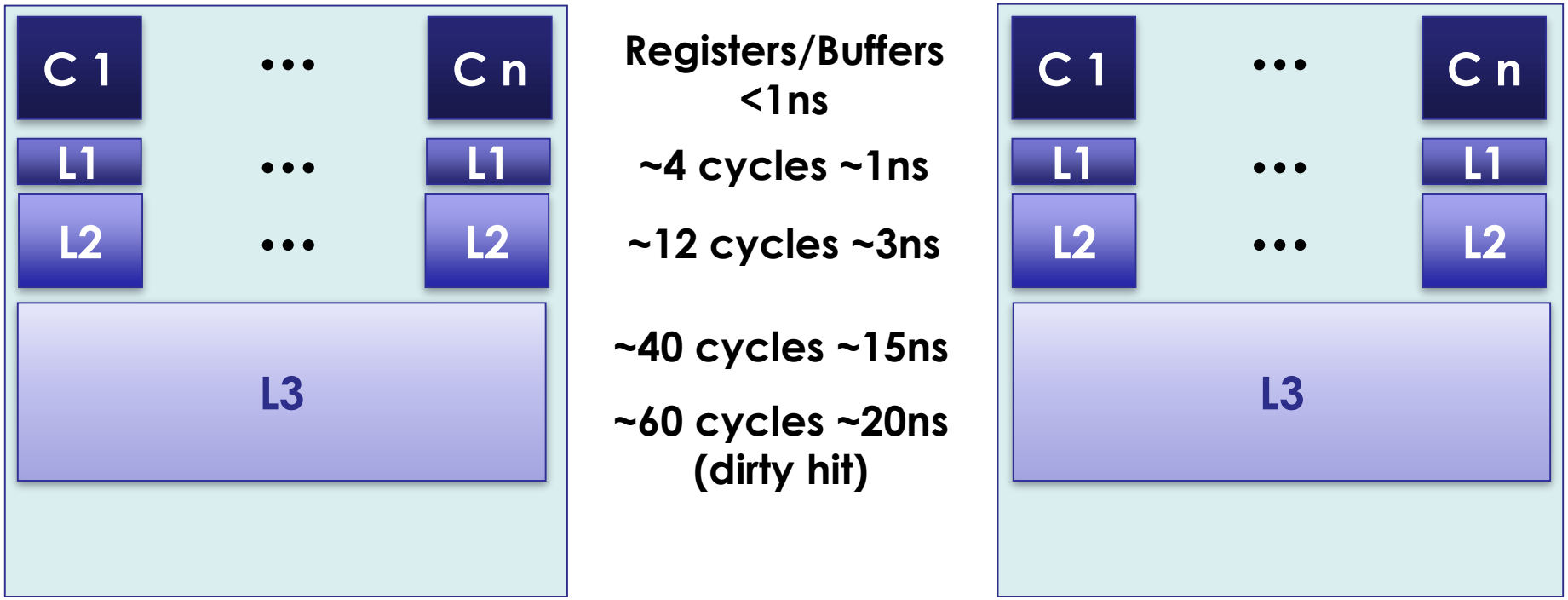
* Assumption: 3GHz Processor

Modern Hardware (Intel Sandy Bridge EP*)



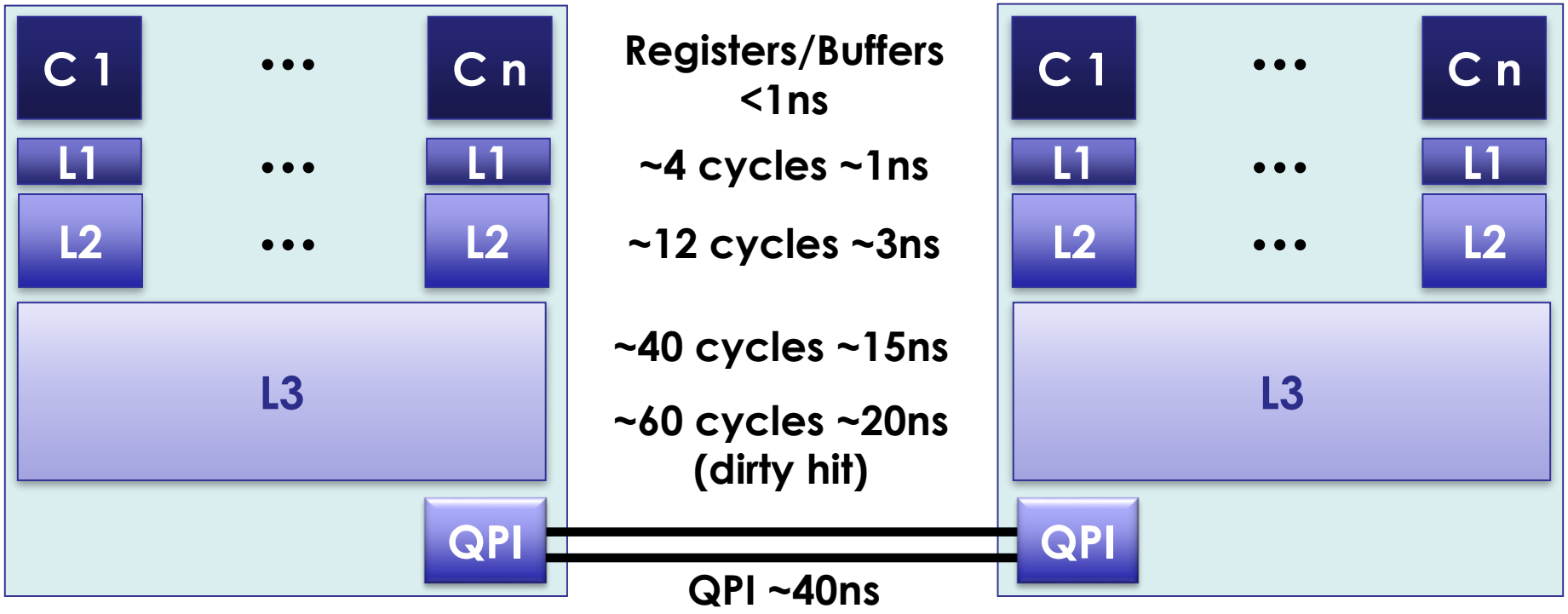
* Assumption: 3GHz Processor

Modern Hardware (Intel Sandy Bridge EP*)



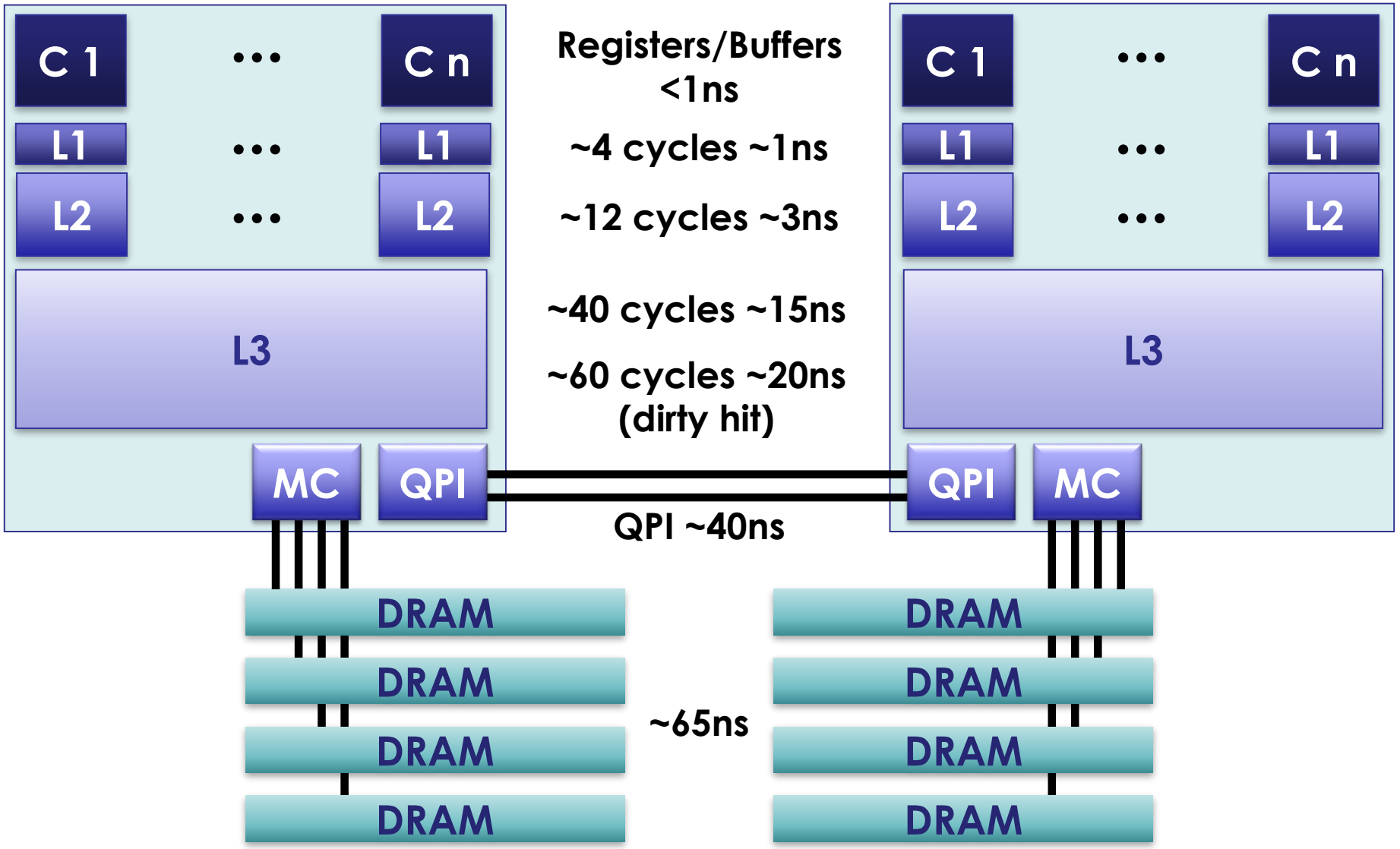
* Assumption: 3GHz Processor

Modern Hardware (Intel Sandy Bridge EP*)



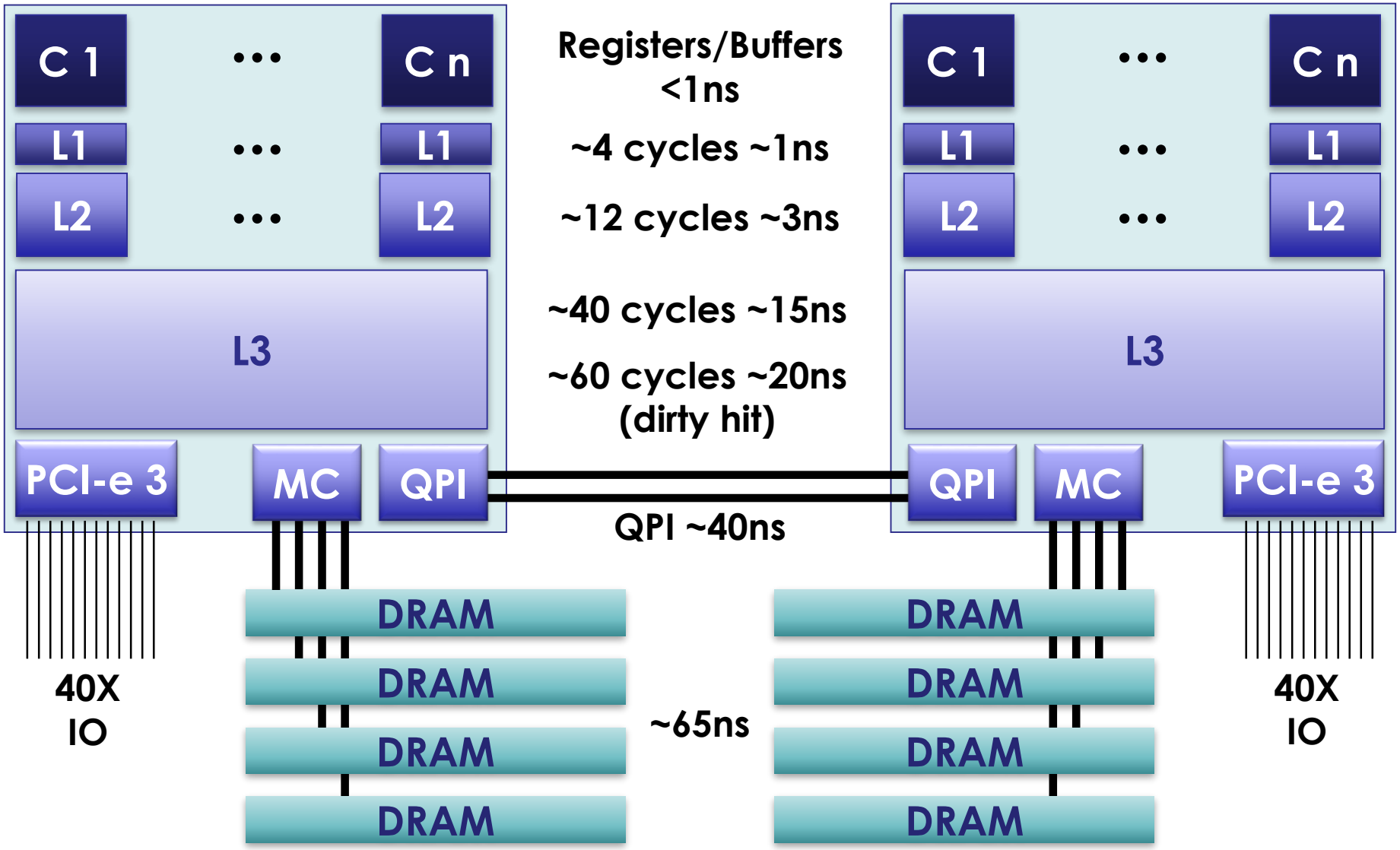
* Assumption: 3GHz Processor

Modern Hardware (Intel Sandy Bridge EP*)



* Assumption: 3GHz Processor

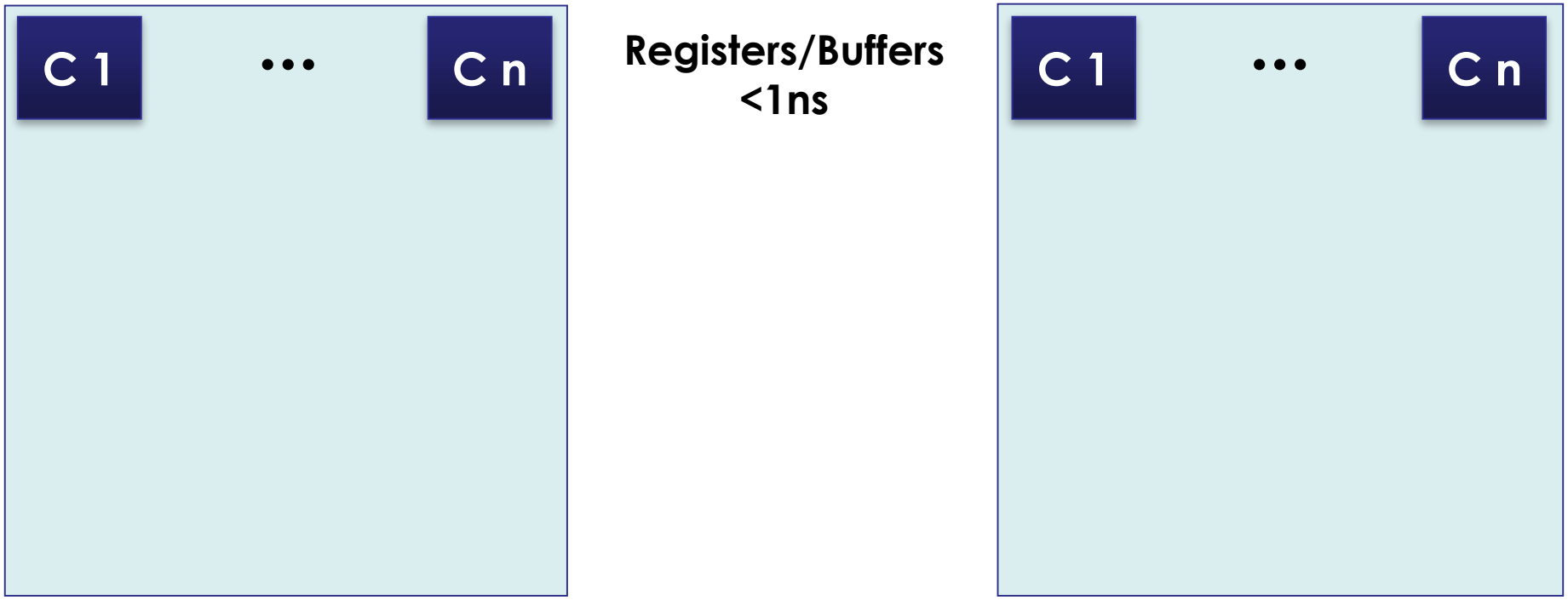
Modern Hardware (Intel Sandy Bridge EP*)



* Assumption: 3GHz Processor

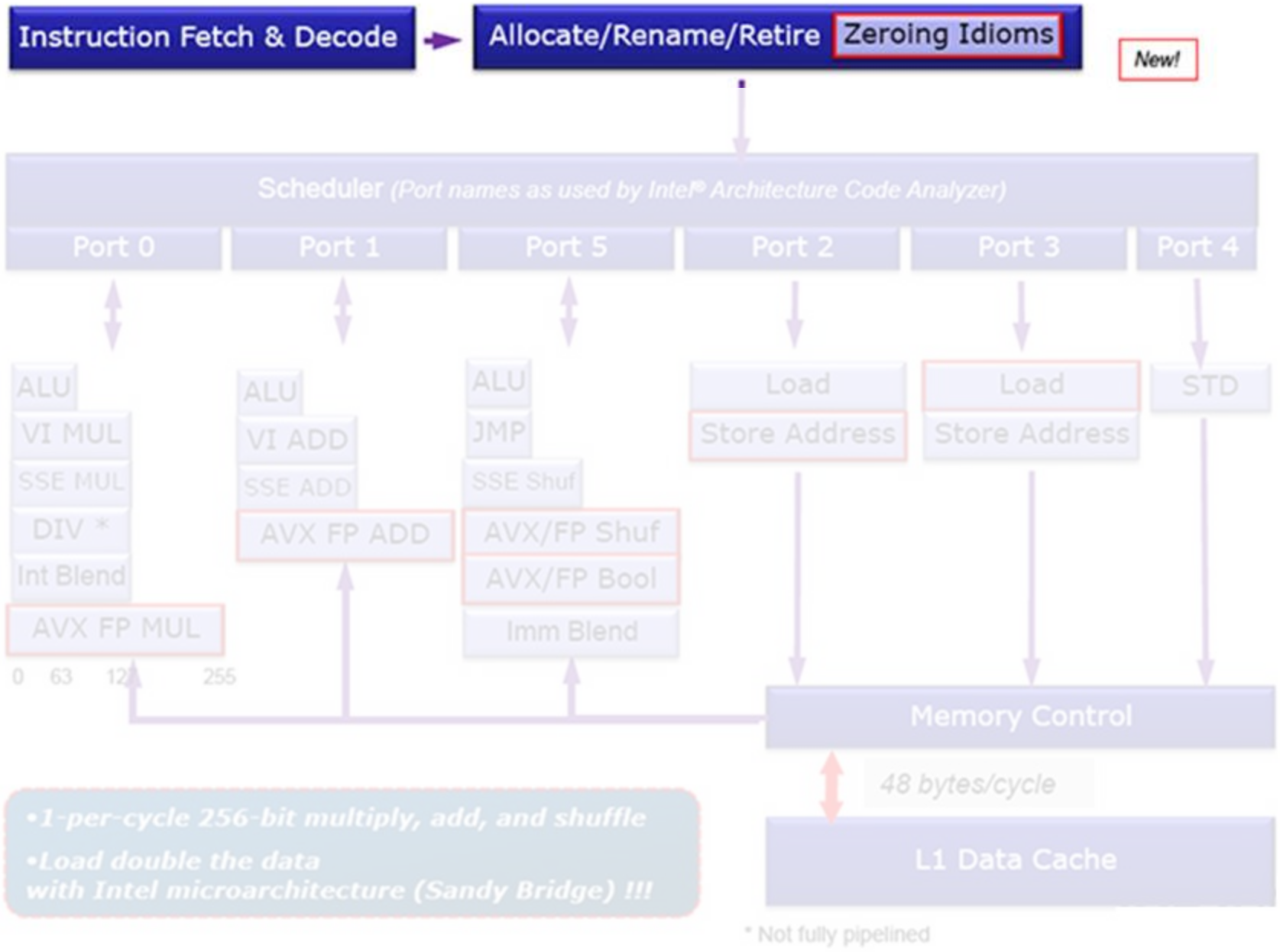
Inside the Processor

Modern Hardware (Intel Sandy Bridge EP*)

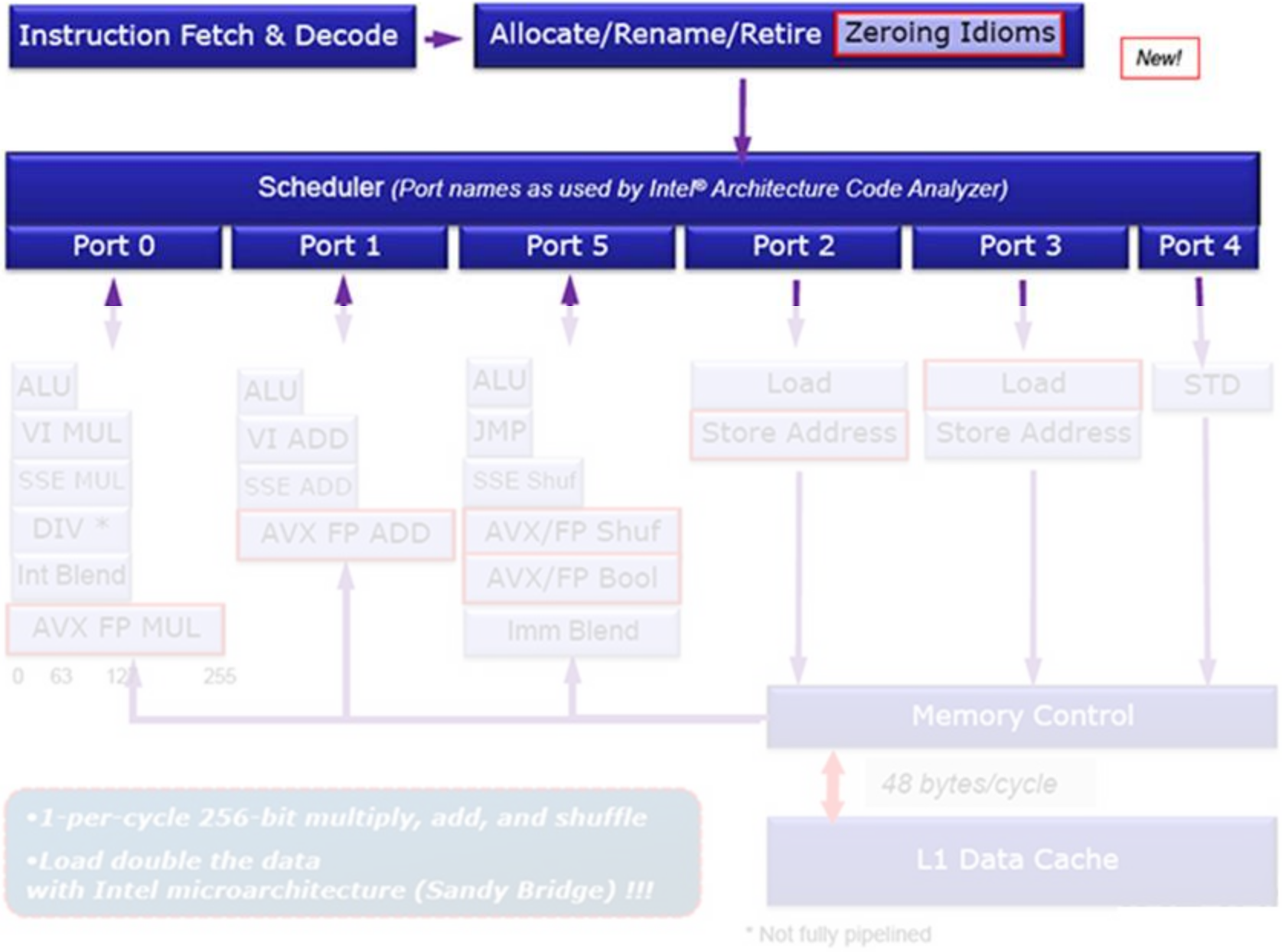


* Assumption: 3GHz Processor

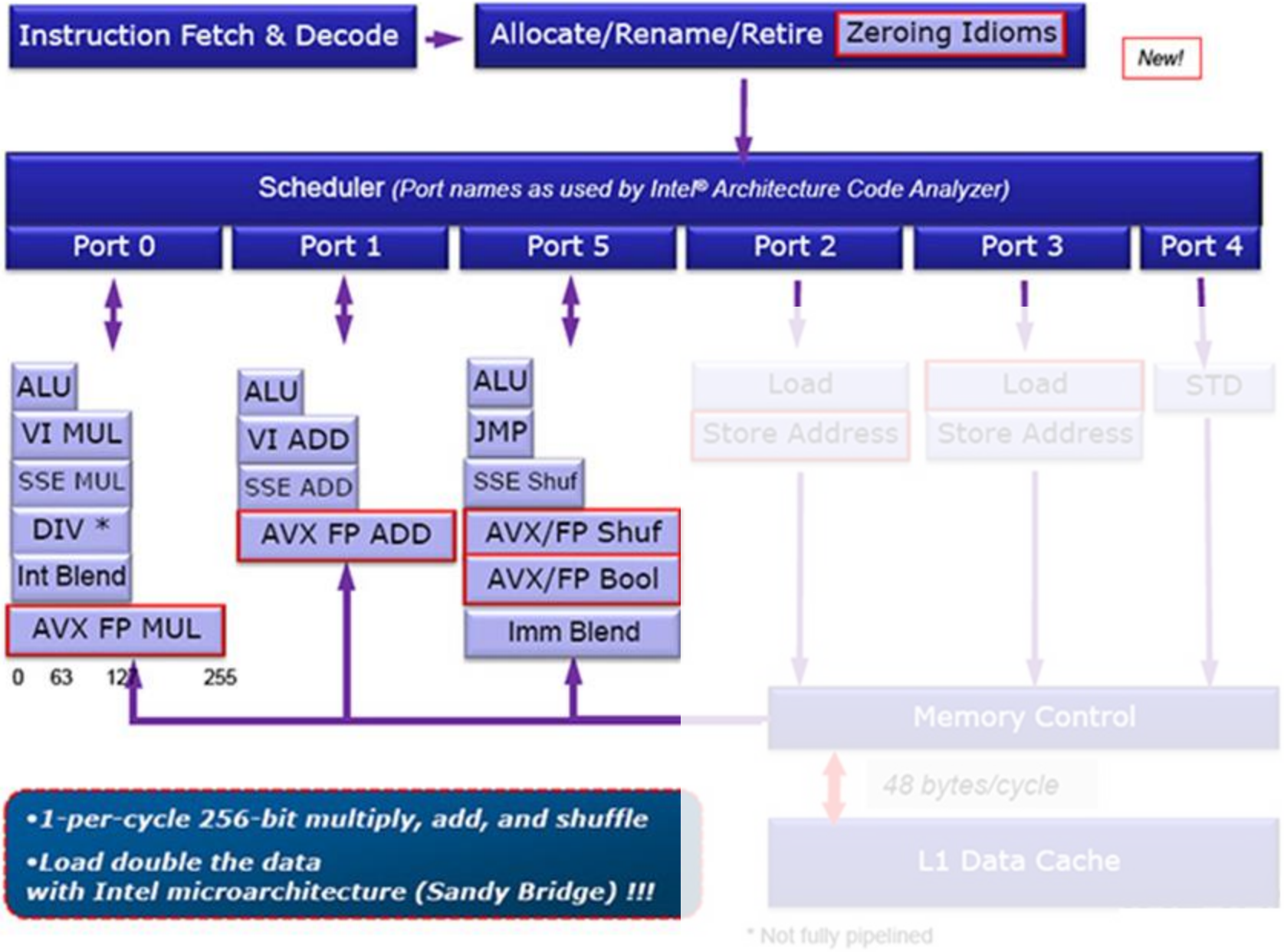
Intel® Microarchitecture (Sandy Bridge) Highlights



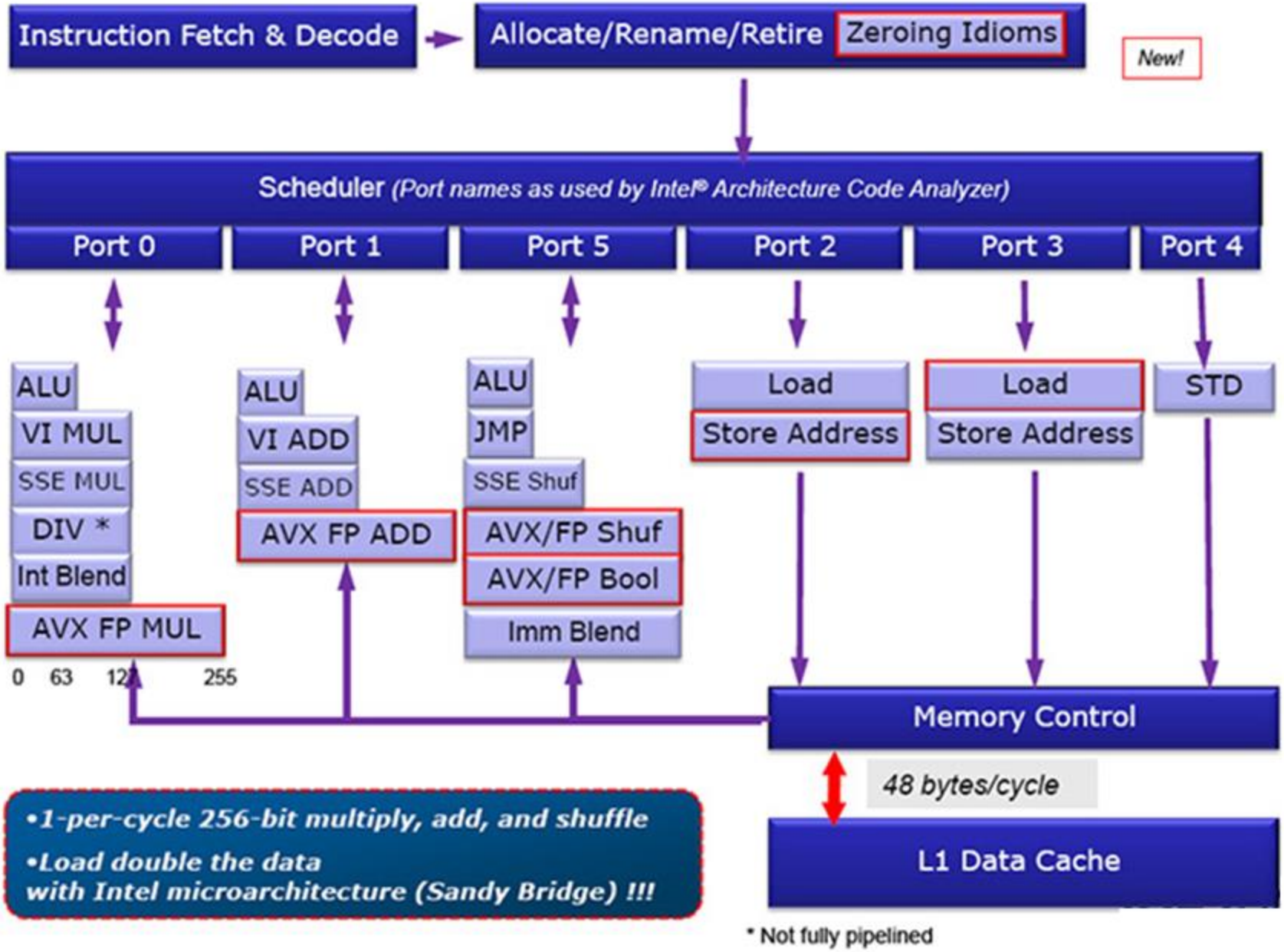
Intel® Microarchitecture (Sandy Bridge) Highlights



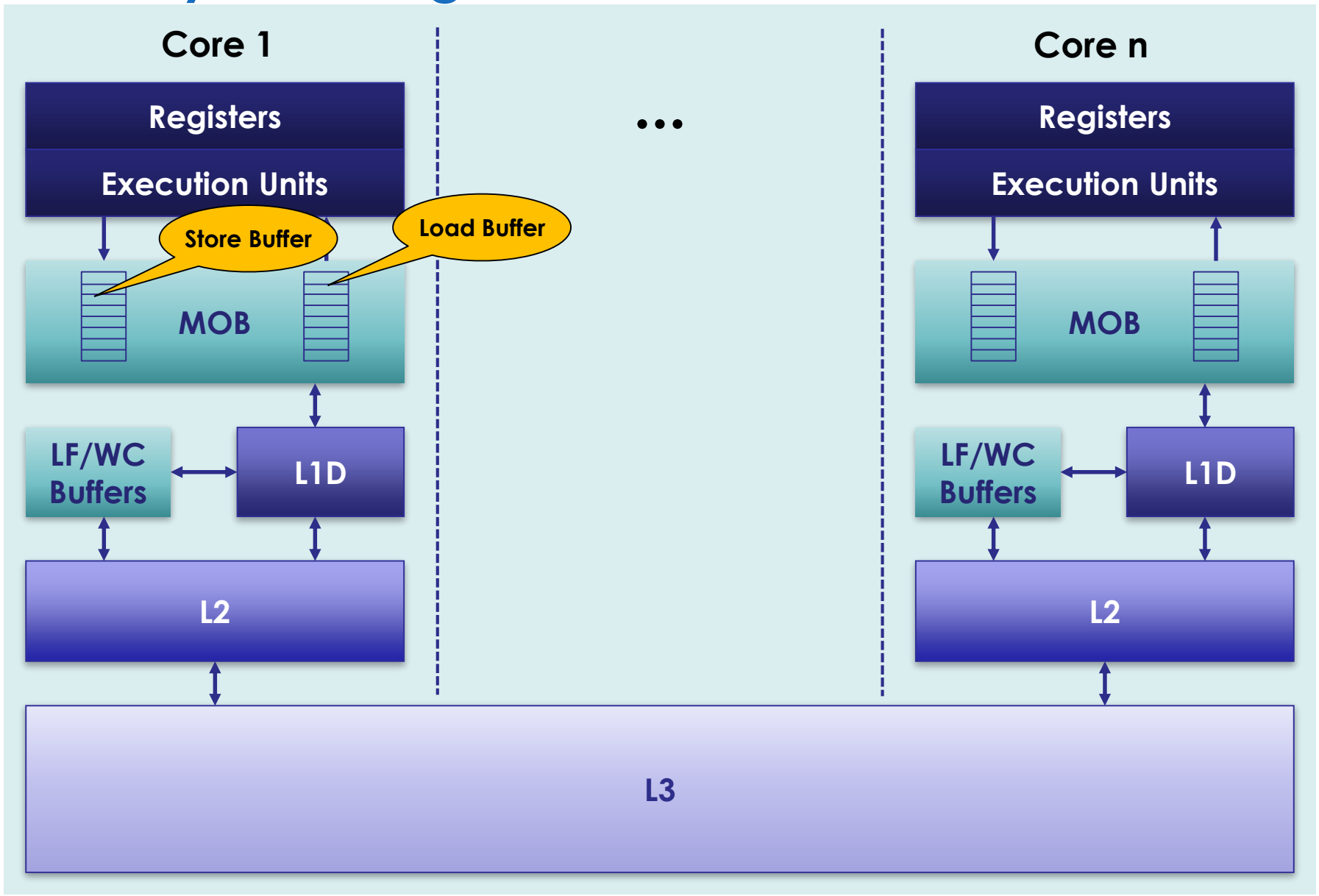
Intel® Microarchitecture (Sandy Bridge) Highlights



Intel® Microarchitecture (Sandy Bridge) Highlights

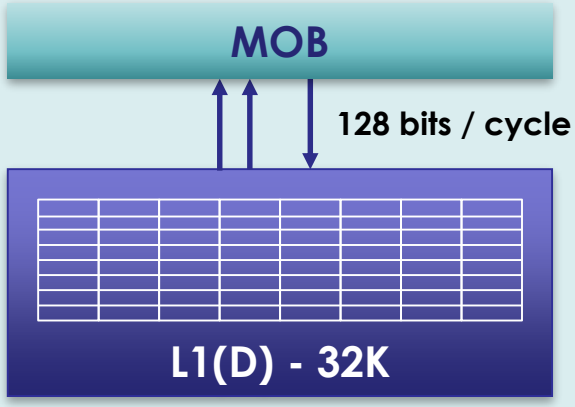


Memory Ordering

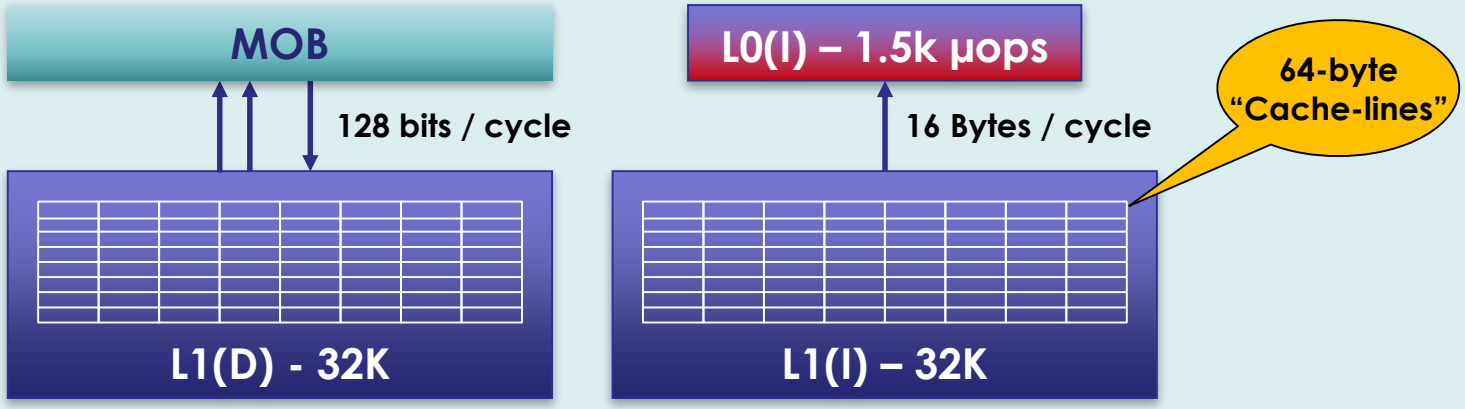


The Cache System

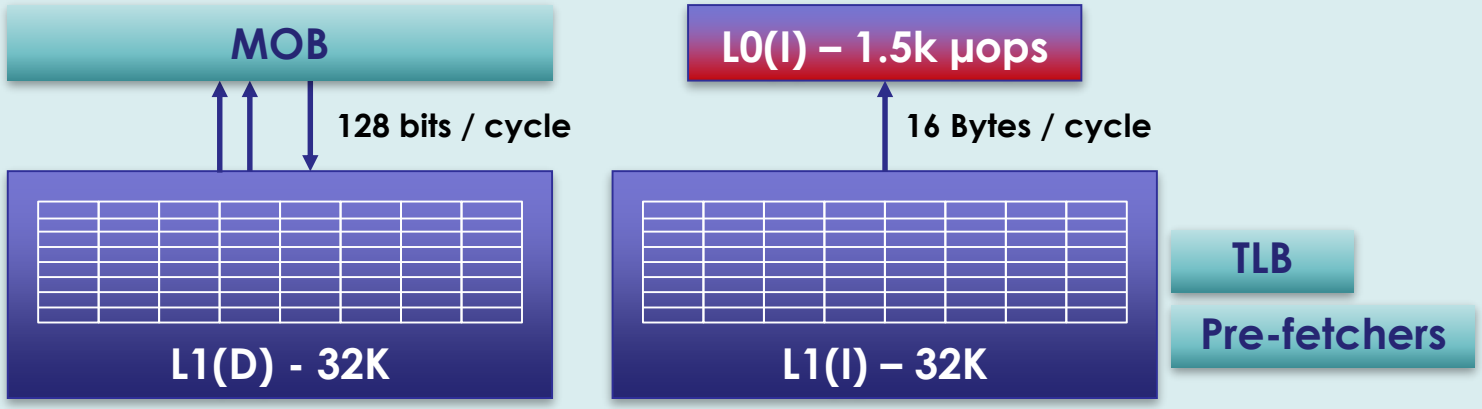
Cache Structure & Coherence



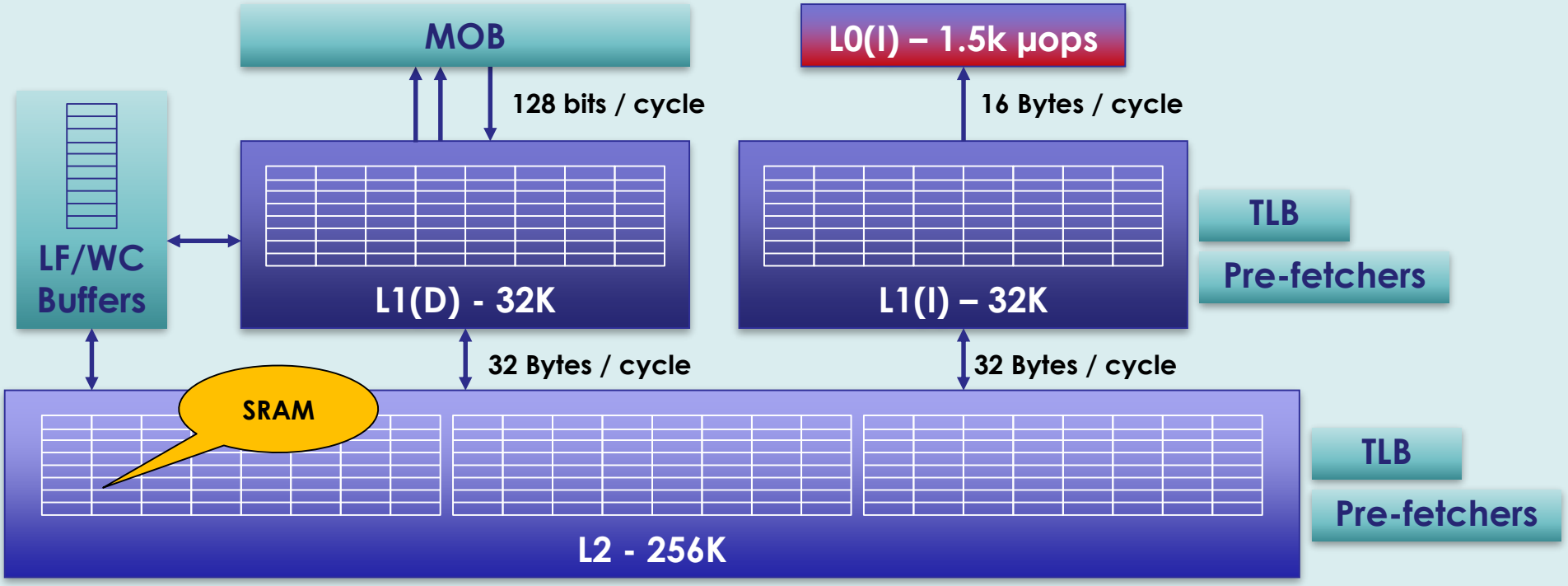
Cache Structure & Coherence



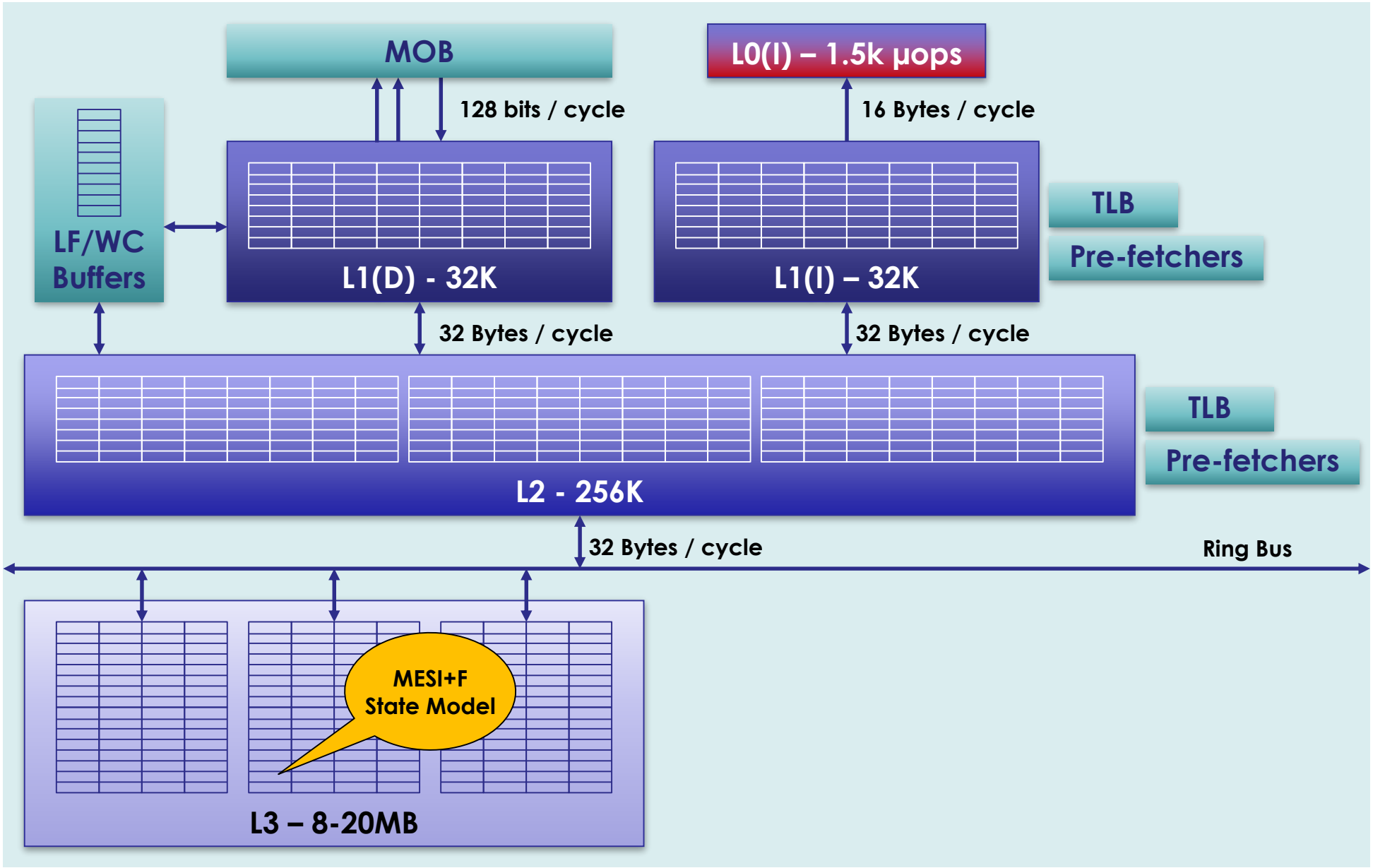
Cache Structure & Coherence



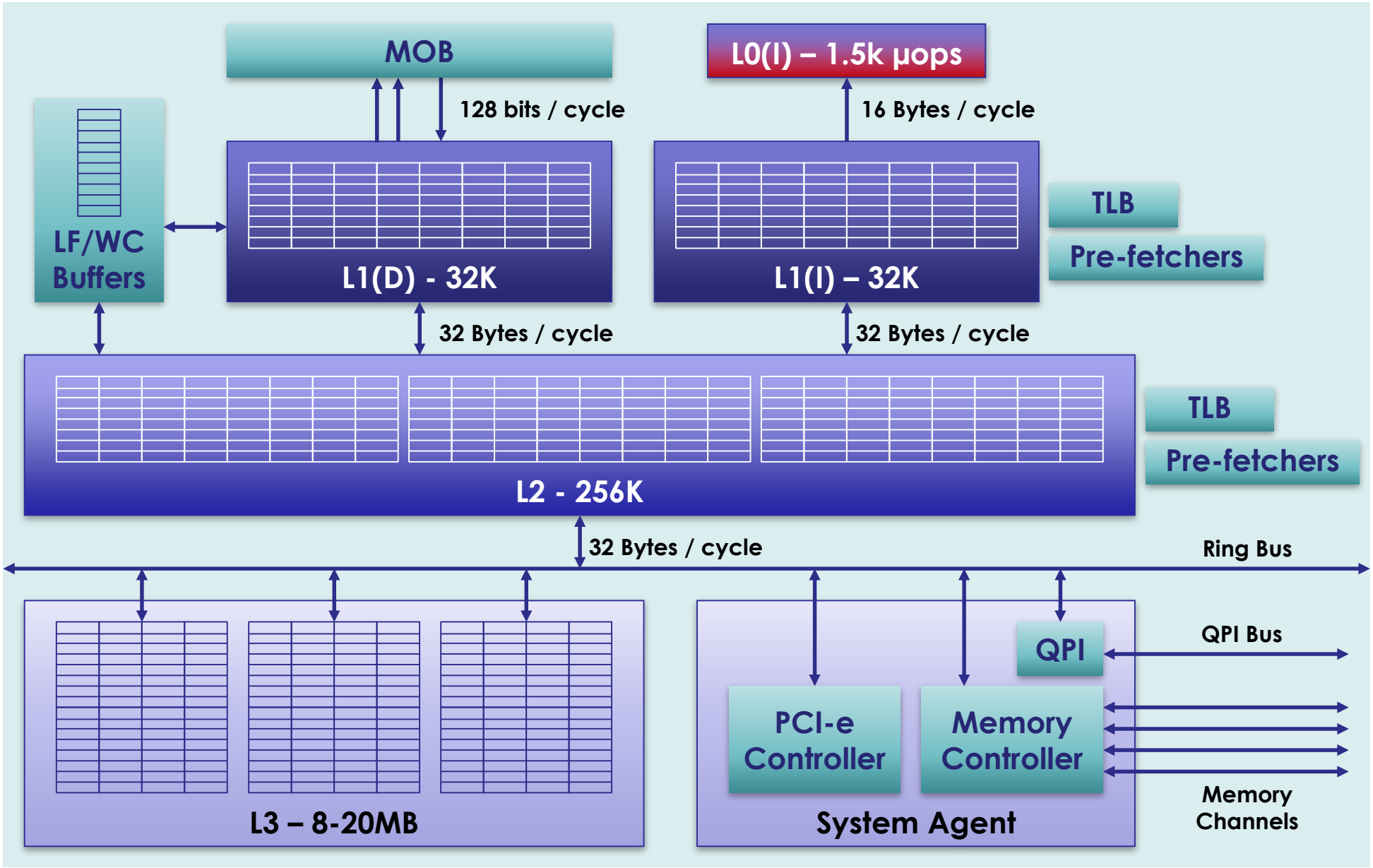
Cache Structure & Coherence



Cache Structure & Coherence

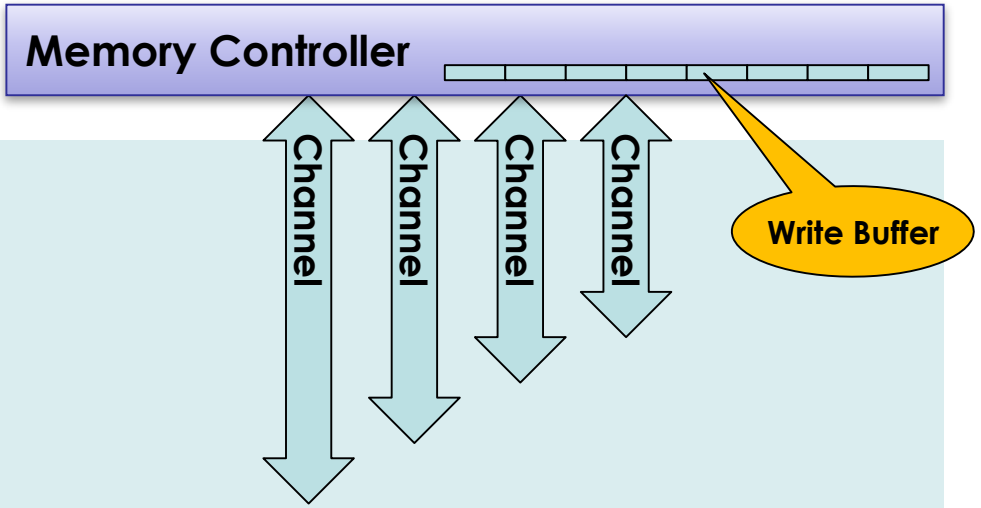


Cache Structure & Coherence

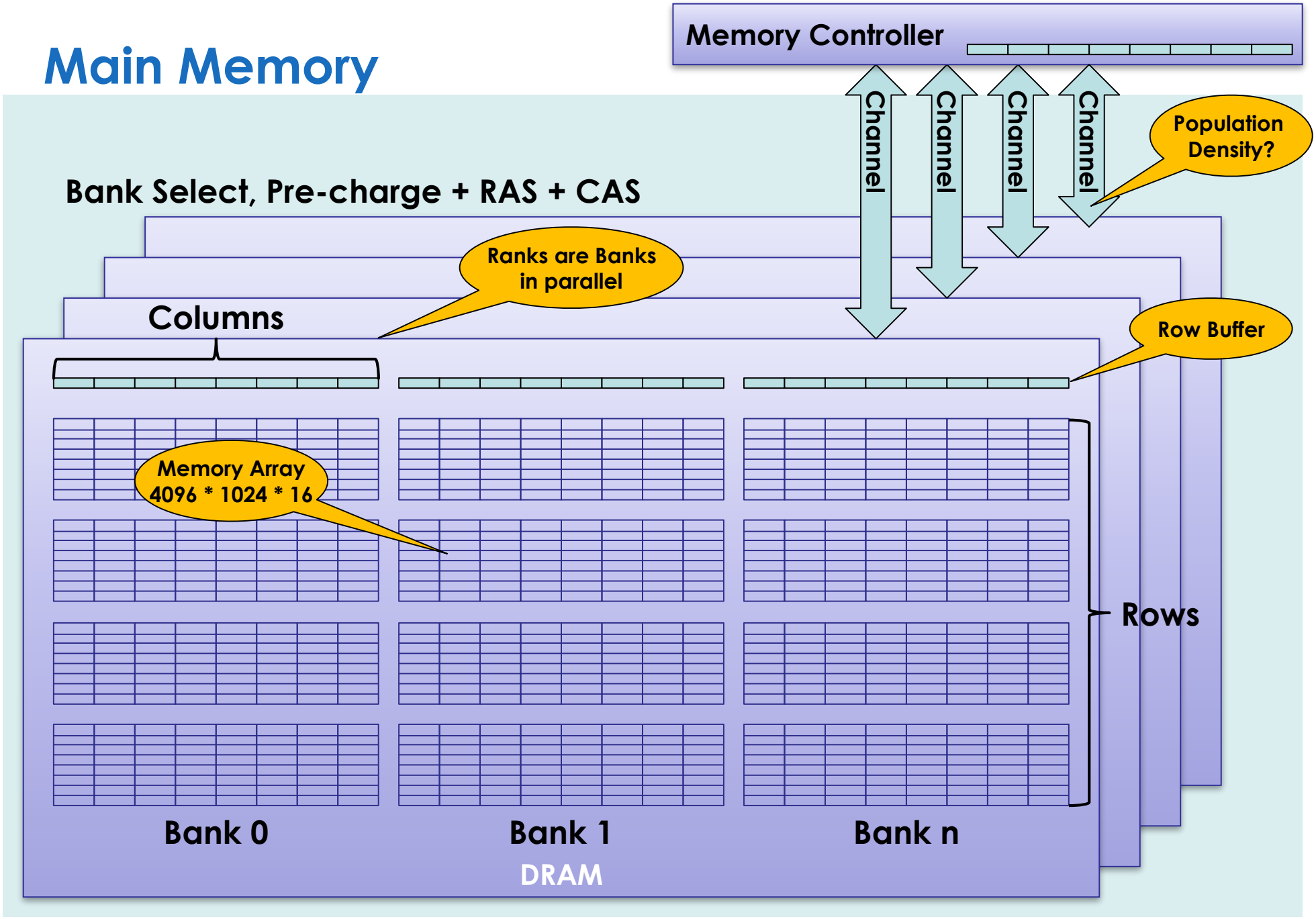


Main Memory

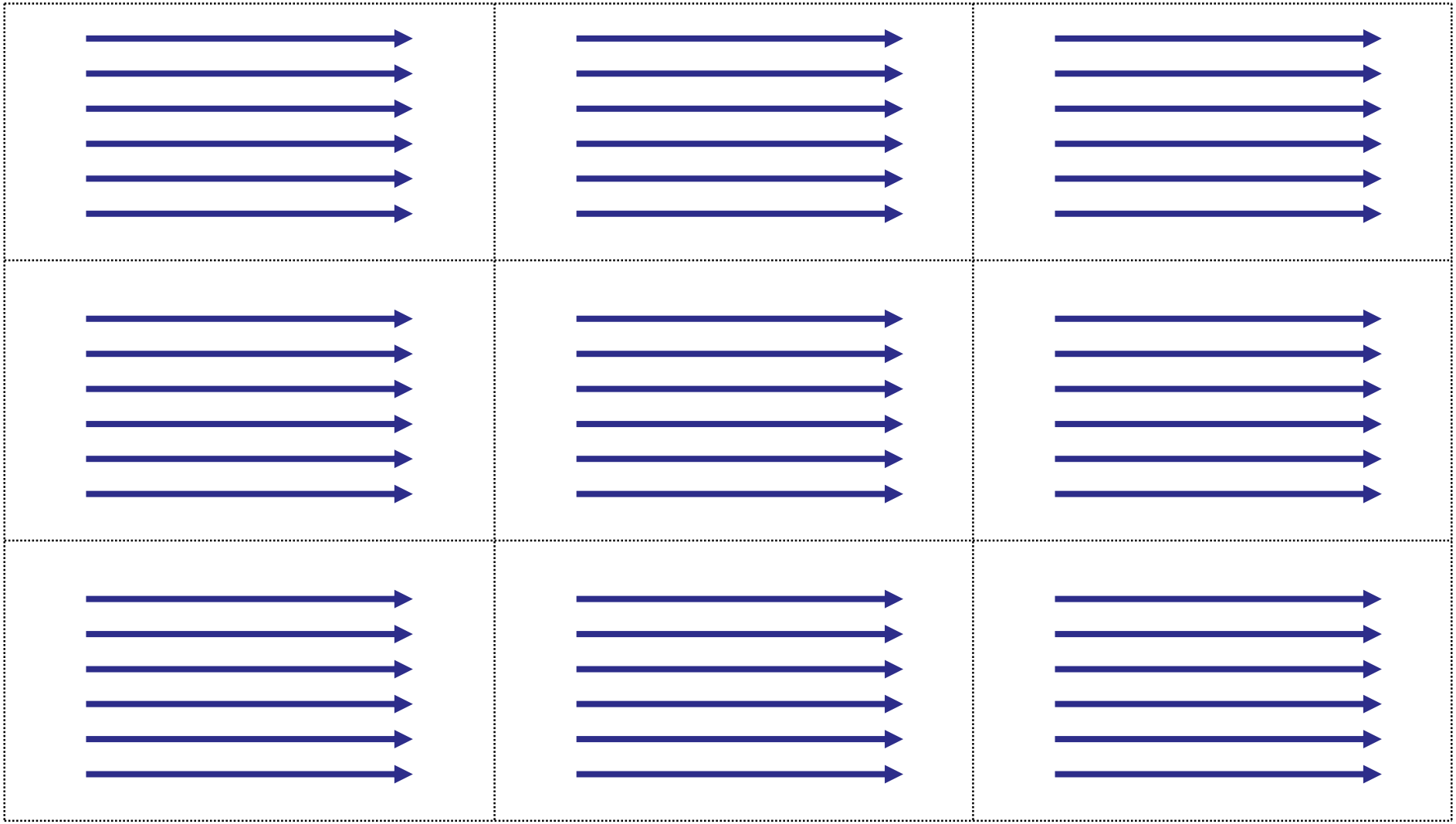
Main Memory

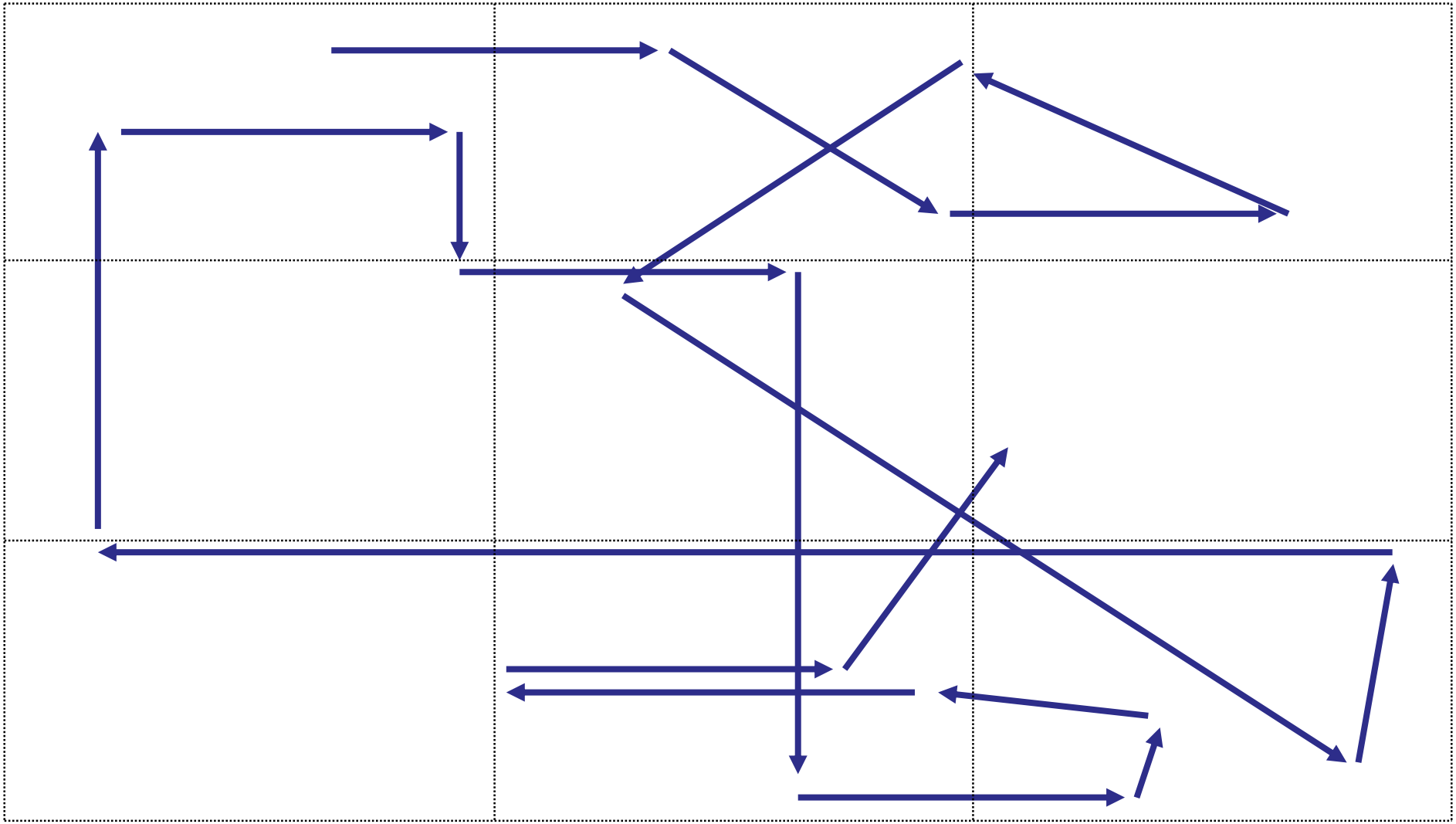

Memory Controller
Channel
Channel
Channel
Channel
Write Buffer

Main Memory


Memory Controller
Channel
Channel
Channel
Channel
Population Density?
Bank Select, Pre-charge + RAS + CAS
Ranks are Banks in parallel
Columns
Row Buffer
Memory Array 4096 * 1024 * 16
Rows
Bank 0
Bank 1
Bank n
DRAM

Memory Access Patterns





Latencies measured by SiSoftware

Intel i7-3960X (Sandy Bridge E)

	L1D	L2	L3	Memory
Sequential	3 clocks	11 clocks	14 clocks	6.0 ns
In-Page Random	3 clocks	11 clocks	18 clocks	22.0 ns
Full Random	3 clocks	11 clocks	38 clocks	65.8 ns

Measuring Interesting Things

Ping Pong

Latency Measurement Pattern

Echo Receiver

Throughput Measurement Pattern

Exercise 1
Ping Pong

Memory Models

Hardware Memory Models

Rules for how threads interact via shared memory

– so we can reason about our software!!!

Hardware Memory Models

Rules for how threads interact via shared memory

– so we can reason about our software!!!

- **Program Order (PO) for a single thread**

Hardware Memory Models

Rules for how threads interact via shared memory

– so we can reason about our software!!!

- Program Order (PO) for a single thread
- **Sequential Consistency (SC) [Lamport 1979]**
  - What you expect a program to do! (for race free)

Hardware Memory Models

Rules for how threads interact via shared memory

– *so we can reason about our software!!!*

- Program Order (PO) for a single thread
- Sequential Consistency (SC) [Lamport 1979]
  - What you expect a program to do! (for race free)
- **Strict Consistency (Linearizability)**
  - Some special instructions

Hardware Memory Models

Rules for how threads interact via shared memory

– so we can reason about our software!!!

- Program Order (PO) for a single thread
- Sequential Consistency (SC) [Lamport 1979]
  - What you expect a program to do! (for race free)
- Strict Consistency (Linearizability)
  - Some special instructions
- **Total Store Order (TSO)**
  - Sparc model that is weaker than SC

Hardware Memory Models

Rules for how threads interact via shared memory

– so we can reason about our software!!!

- Program Order (PO) for a single thread
- Sequential Consistency (SC) [Lamport 1979]
  - What you expect a program to do! (for race free)
- Strict Consistency (Linearizability)
  - Some special instructions
- Total Store Order (TSO)
  - Sparc model that is weaker than SC
- **x86/64 is TSO + (Total Lock Order & Causal Consistency)**
  - http://www.youtube.com/watch?v=WUfvvFD5tAA

Hardware Memory Models

Rules for how threads interact via shared memory

– so we can reason about our software!!!

- Program Order (PO) for a single thread
- Sequential Consistency (SC) [Lamport 1979]
  - What you expect a program to do! (for race free)
- Strict Consistency (Linearizability)
  - Some special instructions
- Total Store Order (TSO)
  - Sparc model that is weaker than SC
- x86/64 is TSO + (Total Lock Order & Causal Consistency)
  - http://www.youtube.com/watch?v=WUfvvFD5tAA
- **Other Processors have weaker models**

Intel x86/64 Memory Model

http://www.multicoreinfo.com/research/papers/2008/damp08-intel64.pdf

http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html

1. Loads are not reordered with other loads.
2. Stores are not reordered with other stores.
3. Stores are not reordered with older loads.
4. Loads may be reordered with older stores to different locations but not with older stores to the same location.
5. In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).
6. In a multiprocessor system, stores to the same location have a total order.
7. In a multiprocessor system, locked instructions have a total order.
8. Loads and stores are not reordered with locked instructions.

Language/Runtime Memory Models

Some languages/Runtimes have a well defined memory model for portability:

- Java Memory Model (Java 5)
- C/C++ 11
- Erlang
- GO

Language/Runtime Memory Models

Some languages/Runtimes have a well defined memory model for portability:

- Java Memory Model (Java 5)
- C/C++ 11
- Erlang
- GO

For most languages we are at the mercy of the compiler

- Instruction reordering
- C “volatile” is inadequate
- Register allocation for caching values
- No mapping to the hardware memory model
- Fences/Barriers need to be applied
- Many have bugs that need to be worked around!

Java Memory Model

Introduced with Java 5 as JSR 133

- Visibility of change in the correct order is key

Java Memory Model

Introduced with Java 5 as JSR 133

- Visibility of change in the correct order is key

- **"*within-thread-as-if-serial*"** for program order

Java Memory Model

Introduced with Java 5 as JSR 133

- Visibility of change in the correct order is key

- "***within-thread-as-if-serial***" for program order
- "***happens-before***" clarification on program order for locks and **`volatile`** variables
  - Release before acquire of a lock
  - Write of a **`volatile`** field before the read to fix store order

Java Memory Model

Introduced with Java 5 as JSR 133

- Visibility of change in the correct order is key

- "***within-thread-as-if-serial***" for program order
- "***happens-before***" clarification on program order for locks and `volatile` variables
  - Release before acquire of a lock
  - Write of a `volatile` field before the read to fix store order
- "***initialisation safety***" `final` fields must be visible at the end of a constructor

Java Memory Model

Introduced with Java 5 as JSR 133

- Visibility of change in the correct order is key

- "***within-thread-as-if-serial***" for program order
- "***happens-before***" clarification on program order for locks and **`volatile`** variables
  - Release before acquire of a lock
  - Write of a **`volatile`** field before the read to fix store order
- "***initialisation safety***" **`final`** fields must be visible at the end of a constructor
- Fields declared **`volatile`** are read and written as **atomic** operations even when **`double`** and **`long`**

Exercise 2
Condition Variables

Performance Testing

Performance Testing

Minefield of mistakes!!! – Goal is to achieve consistency

- **JVM behaviours**
  - **Classloading**
  - **Compilation**
  - **Garbage Collection**
  - **Optimisation Race Conditions**
  - **“Fake” warmups**
  - **Eliminated code**

Performance Testing

Minefield of mistakes!!! – Goal is to achieve consistency

- JVM behaviours
  - Classloading
  - Compilation
  - Garbage Collection
  - Optimisation Race Conditions
  - "Fake" warmups
  - Eliminated code

- **Performance tests need to be faster than target code**
  - **Test against stubs**
  - **Make separate "correctness tests"**

Performance Testing

Minefield of mistakes!!! – Goal is to achieve consistency

- JVM behaviours
  - Classloading
  - Compilation
  - Garbage Collection
  - Optimisation Race Conditions
  - "Fake" warmups
  - Eliminated code

- Performance tests need to be faster than target code
  - Test against stubs
  - Make separate "correctness tests"

- **Micro benchmarking is difficult and can be very misleading!**

Exercise 3
IP 1C Queue

Mechanical Sympathy
"Making Progress"

Mechanical Sympathy

Is it really “Turtles all the way down”?

Mechanical Sympathy

Is it really “Turtles all the way down”?

What is under all these layers of abstraction?

Mechanical Sympathy

Is it really "Turtles all the way down"?

What is under all these layers of abstraction?

"The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry."

- Henry Peteroski

Mechanical Sympathy – *"Making Progress"*

Do we really need to flush the store buffer in our 1P1C Queue?

- `java.util.concurrent.AtomicLong.lazySet()`

Mechanical Sympathy – *"Making Progress"*

Do we really need to flush the store buffer in our 1P1C Queue?

- `java.util.concurrent.AtomicLong.lazySet()`

How can we avoid the stalling and expensive division operation to get a remainder?

- **Let's go back to good old binary mathematics:**
  - **Buffer as a power of 2 in size**
  - **Mask for the remainder with size - 1**

Exercise 4
Improved IP 1C Queue

Why Contention Is The Enemy

Contention

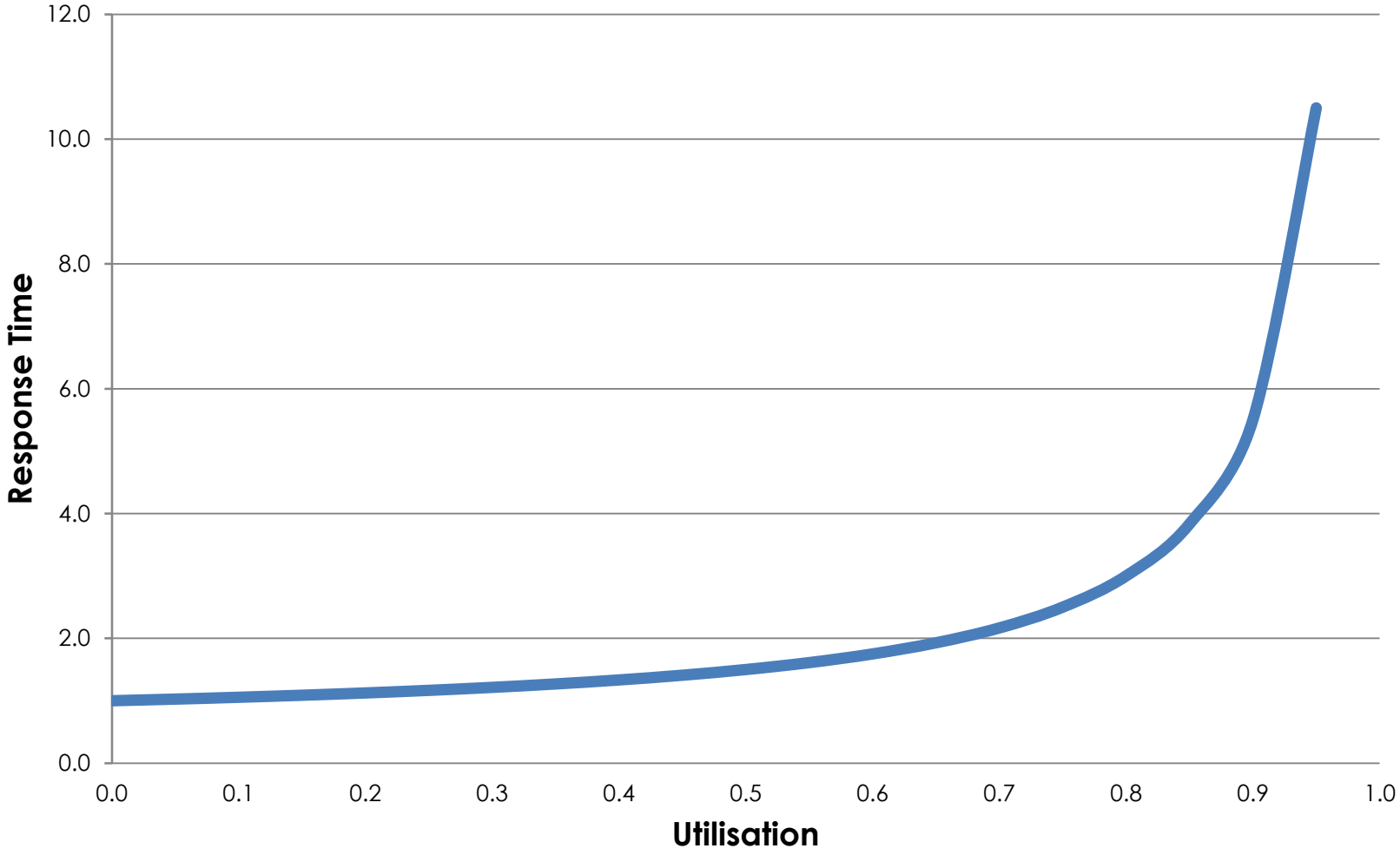
- Managing Contention
  - Locks
  - CAS Techniques
- Little's & Amdahl's Laws
  - $L = \lambda W$
  - *Sequential Component Constraint*

Contention

- Managing Contention
  - Locks
  - CAS Techniques
- Little's & Amdahl's Laws
  - $L = \lambda W$
  - *Sequential Component Constraint*

- **Single Writer Principle**
- **Shared Nothing Designs**

Queuing Theory



Kendall Notation

M/D/1

Queuing Theory

$$r = s(2 - \rho) / 2(1 - \rho)$$

r = mean response time

s = service time

ρ = utilisation

Queuing Theory

$$r = s(2 - \rho) / 2(1 - \rho)$$

r = mean response time

s = service time

ρ = utilisation

Note: $\rho = \lambda * s$

Little's Law

L = λW : WIP = Throughput * Cycle Time

Queue Length = Average effective arrival rate * Average time in the System

Little's Law

L = λW : WIP = Throughput * Cycle Time

Queue Length = Average effective arrival rate * Average time in the System

? L = 4 per µs * 8µs

? L = 0.001 per µs * 8µs

? L = 100 per µs * 8µs

Little's Law

L = λW : WIP = Throughput * Cycle Time

Queue Length = Average effective arrival rate * Average time in the System

? L = 4 per µs * 8µs

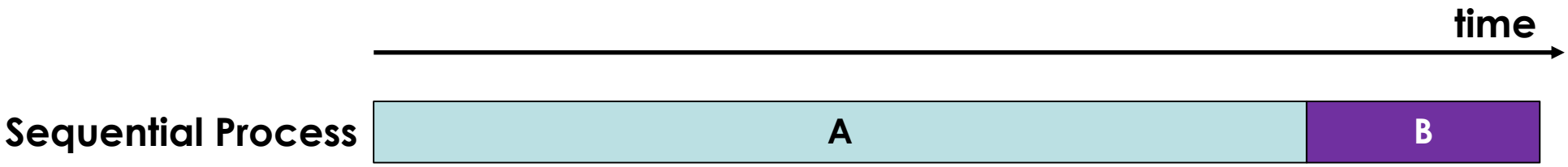
? L = 0.001 per µs * 8µs

? L = 100 per µs * 8µs

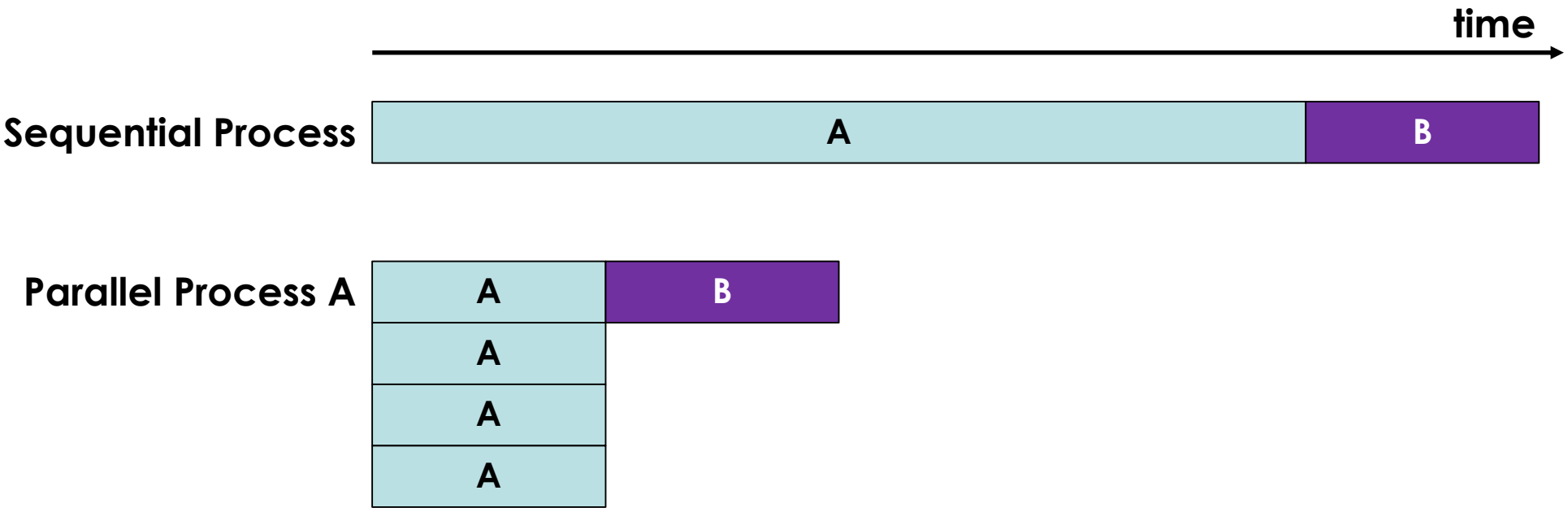
Max Latency = Cycle Time * Queue Length

? ML = 8µs * 100

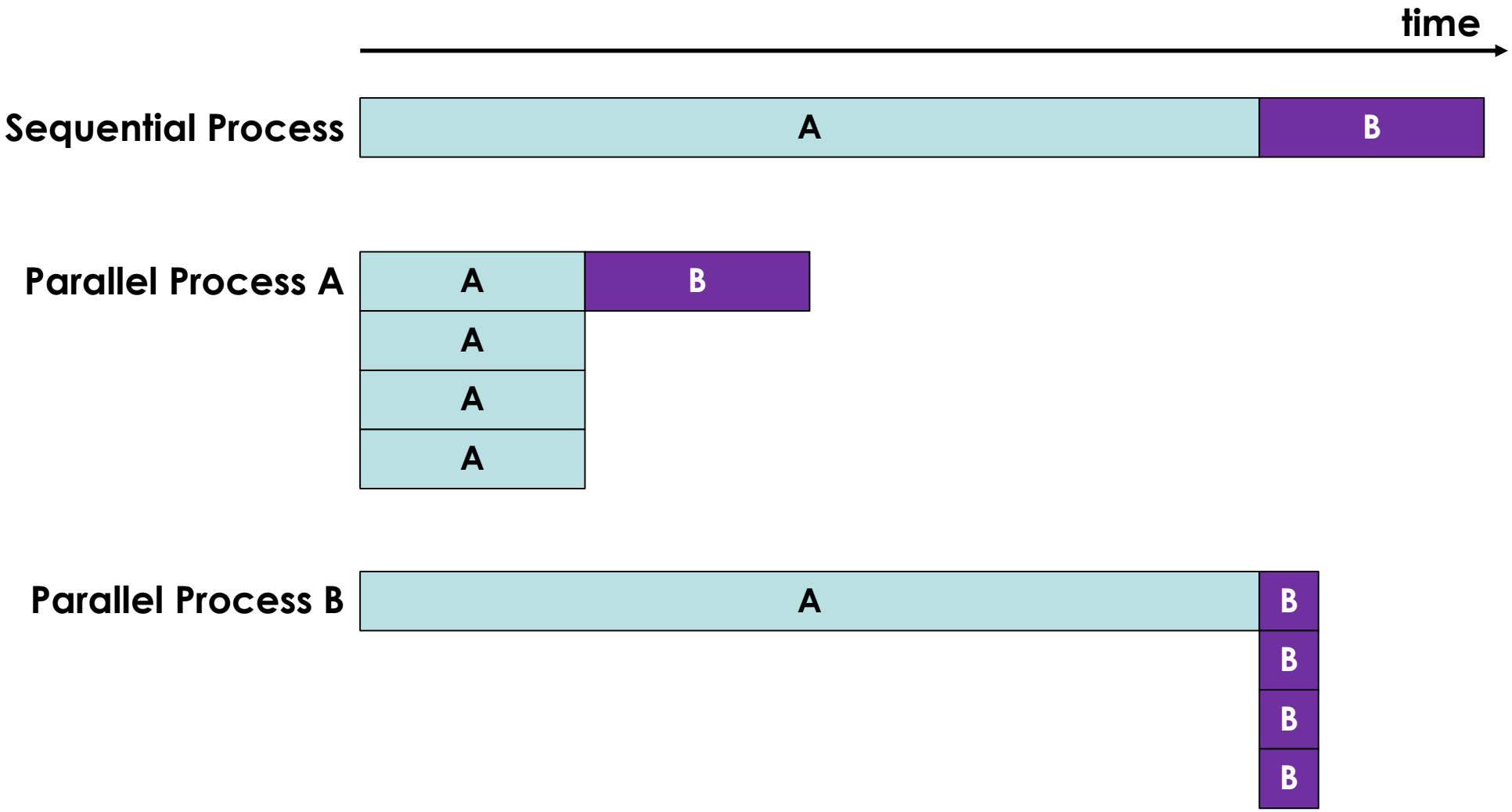
Amdahl's Law



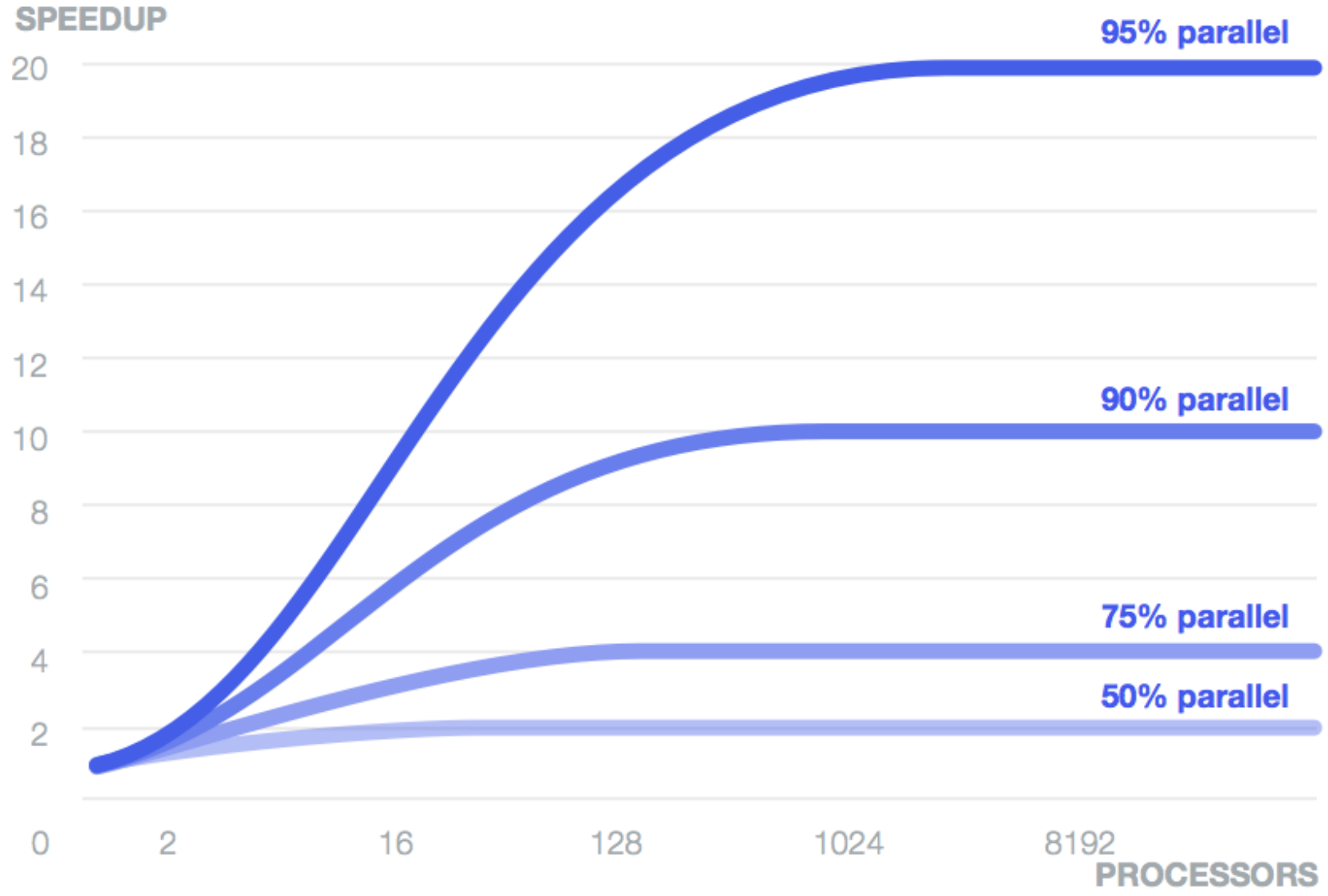
Amdahl's Law



Amdahl's Law


time
Sequential Process
A
B
Parallel Process A
A
B
A
A
A
Parallel Process B
A
B
B
B
B

Amdahl's Law


SPEEDUP
20
18
16
14
12
10
8
6
4
2
0
2
16
128
1024
8192
PROCESSORS
95% parallel
90% parallel
75% parallel
50% parallel

Universal Scalability Law

$$C(N) = N / (1 + \alpha(N - 1) + ((\beta * N) * (N - 1)))$$

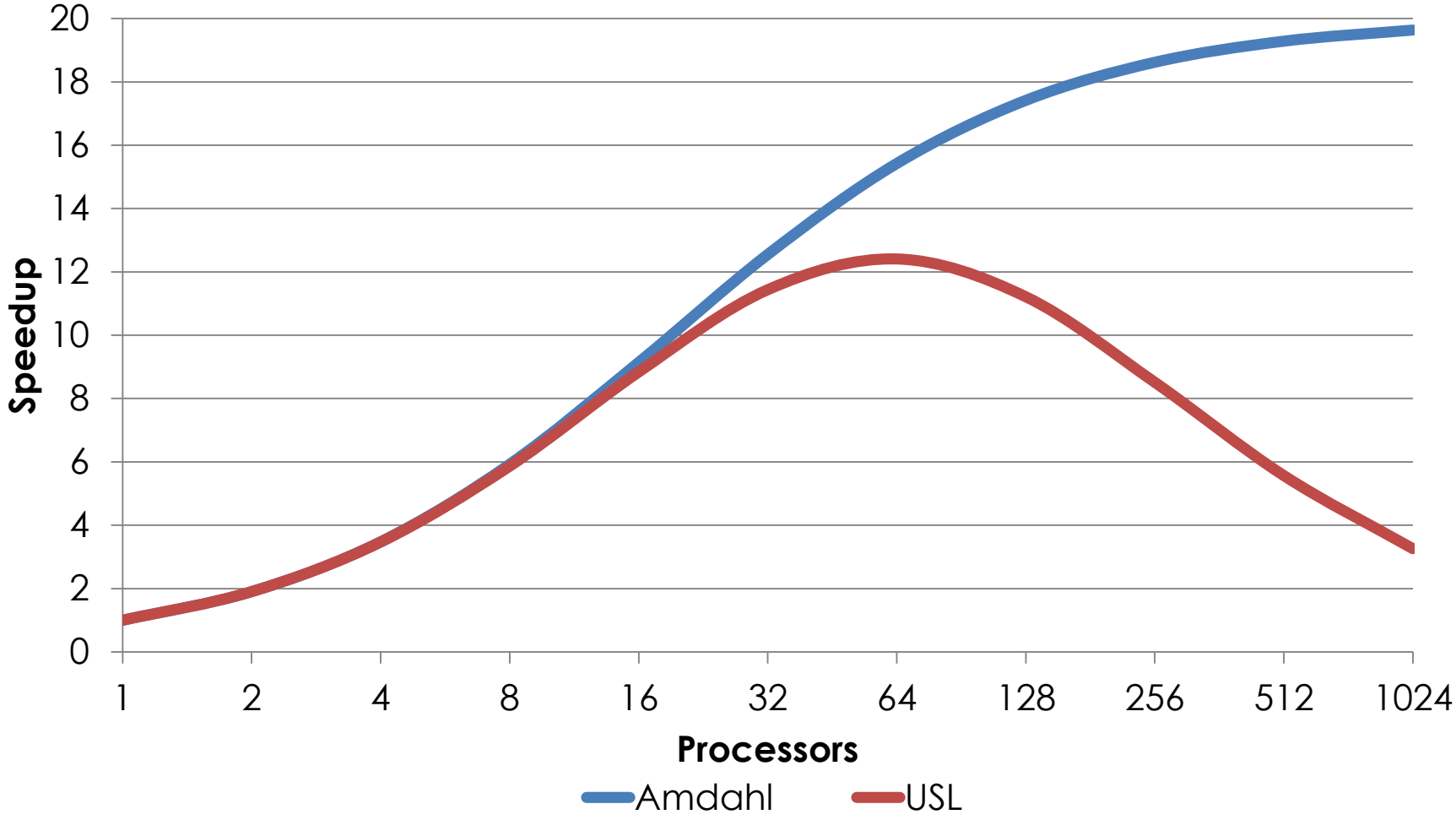
C = capacity or throughput

N = number of processors

α = ***contention*** penalty

β = ***coherence*** penalty

Universal Scalability Law



Managing Contention

Managing Contention In User Space

Sometimes it is necessary for doing configuration changes such as adding or removing a subscriber, or implementing a sequence or counter

Managing Contention In User Space

Sometimes it is necessary for doing configuration changes such as adding or removing a subscriber, or implementing a sequence or counter

- On x86/64, LOCK instructions are available and are commonly known as CAS or Atomic instructions
- They use to lock the memory bus for serialisation but since Nehalem they work on just exclusive cache lines
- Tend to be ~15 cycles in cost
- The have full fence semantics and wait on draining the store buffers

Managing Contention In User Space

Sometimes it is necessary for doing configuration changes such as adding or removing a subscriber, or implementing a sequence or counter

- On x86/64, LOCK instructions are available and are commonly known as CAS or Atomic instructions
- They use to lock the memory bus for serialisation but since Nehalem they work on just exclusive cache lines
- Tend to be ~15 cycles in cost
- The have full fence semantics and wait on draining the store buffers
- **They are available to C/C++ via Intel "*intrinsics*" or *GNU Atomic Builtins* with the same API**

Managing Contention In User Space

Sometimes it is necessary for doing configuration changes such as adding or removing a subscriber, or implementing a sequence or counter

- On x86/64, LOCK instructions are available and are commonly known as CAS or Atomic instructions
- They use to lock the memory bus for serialisation but since Nehalem they work on just exclusive cache lines
- Tend to be ~15 cycles in cost
- The have full fence semantics and wait on draining the store buffers
- They are available to C/C++ via Intel "*intrinsics*" or *GNU Atomic Builtins* with the same API
- **X86/64 has the rather nice LOCK XADD for counters**
  - **Coming in Java 8 ☺**

Managing Contention Using Atomic*

Java 5 added `java.util.concurrent.Atomic*`

- CAS bases updates to `int`, `long` and references
- CAS is a Compare And Set/Swap of a value conditionally
- Can be applied in a loop until success

Managing Contention Using Atomic*

Java 5 added java.util.concurrent.Atomic*

- CAS bases updates to int, long and references
- CAS is a Compare And Set/Swap of a value conditionally
- Can be applied in a loop until success

They are based on a set of operations hidden in Unsafe

- **These are replaced as *intrinsics* by the runtime optimiser**
- **Unsafe is not available to normal programs**
  - **However! With reflection we can get to it...**

Moving Between Known Good States

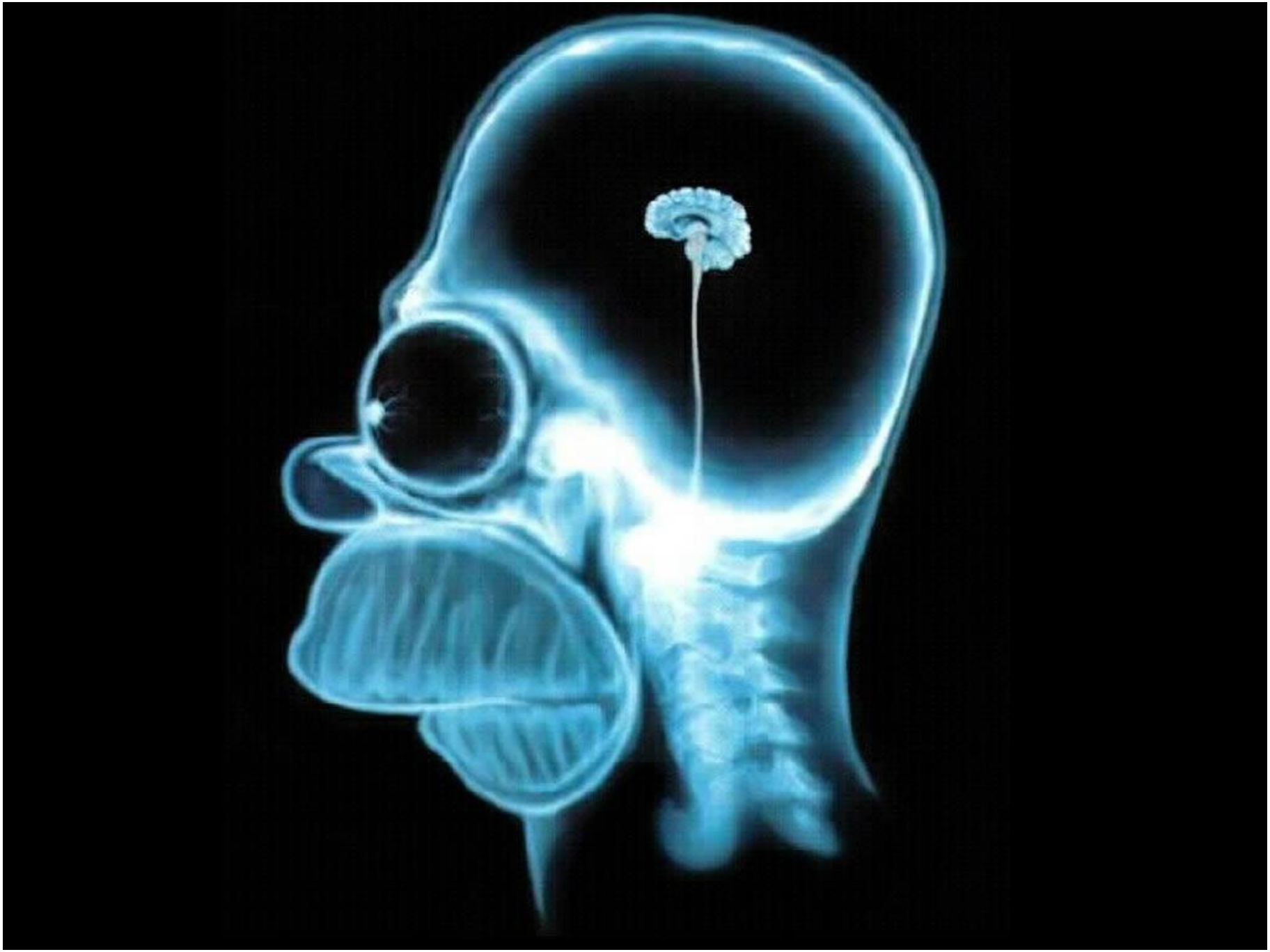
Contended access is best considered as state machines

Moving Between Known Good States

Contended access is best considered as state machines

- CAS based steps that are atomic
- Be prepared for failure and to re-try
- Try to make progress but also be prepared to back out

Exercise 5
MPSC Queue



Questions?

Blog: http://mechanical-sympathy.blogspot.com/

Email: martin@real-logic.co.uk

Twitter: @mjpt777

Discussion: https://groups.google.com/forum/#!forum/mechanical-sympathy

"Any intelligent fool can make things bigger, more complex, and more violent.

It takes a touch of genius, and a lot of courage, to move in the opposite direction."

- Albert Einstein