Bridge of Life Education

SOC Design
Processor - SuperScalar

Jiin Lai

©BOLEDU

Topics

1. The Reasoning of RISC Processor Design
2. ISA & Single Cycle Processor
3. Multi-Cycle Processor
4. Pipelined Processor
5. Superscalar

©BOLEDU

Parallelisms in Processor

- Instruction-Level Parallelism – ILP
  - SuperScalar
  - Dynamic Execution – Out-of-Order
- Data-Level Parallelism – SIMD/Vector Processing
- Multitasking
  - Multicore
  - Multithreading

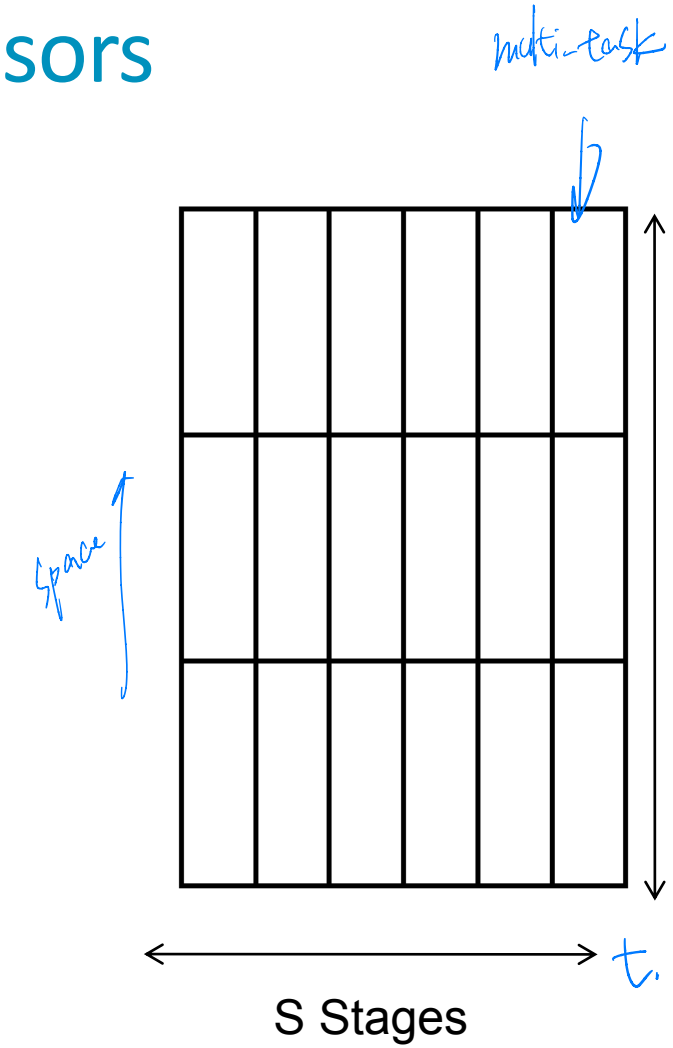
©BOLEDU

SuperScalar

Ref: “processor-ilp-ref.pdf:

©BOLEDU

Superscalar Processors



A Superscalar Processor

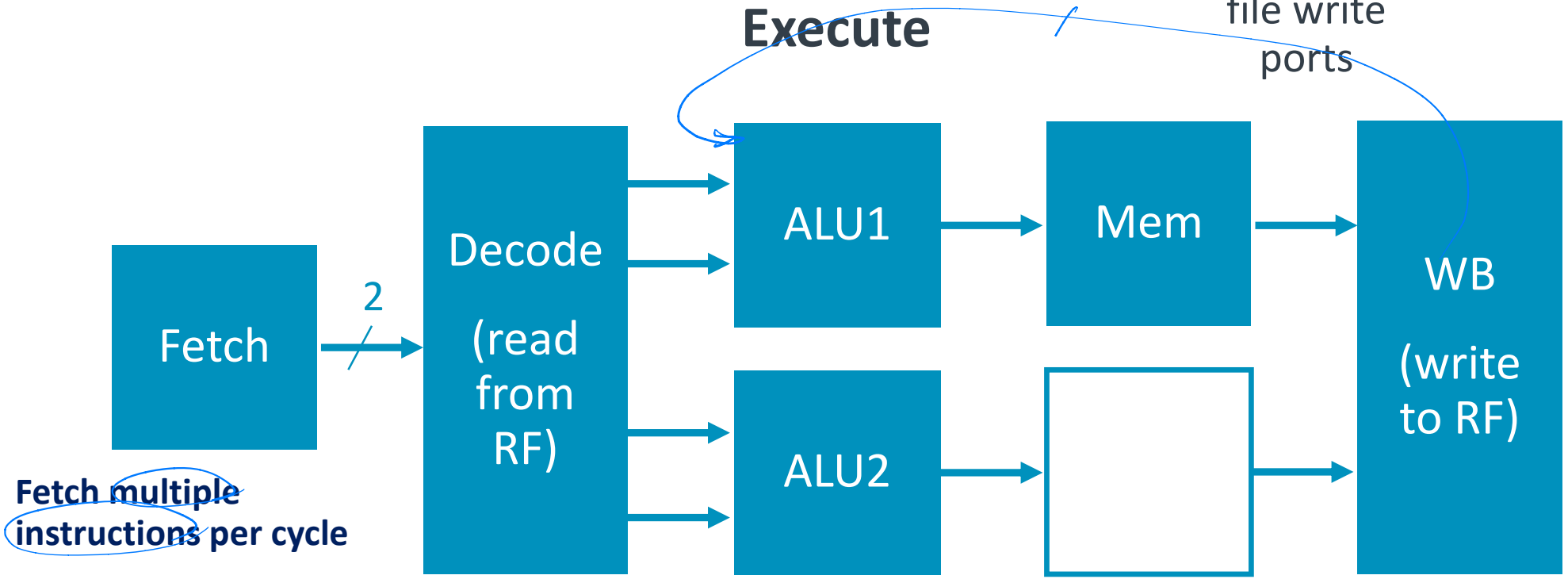
P instructions are processed in each pipeline stage.

IPC ≤ P

Clock Period = T/S+C

©BOLEDU

Simple In-order Superscalar Processor



Fetch multiple instructions per cycle

Decode multiple instruction and check for dependency

Increase # of ports of Register file

Increase # function units

Additional data forwarding paths required from and to both ALUs.

©BOLEDU

Exposing and Exploiting More ILP

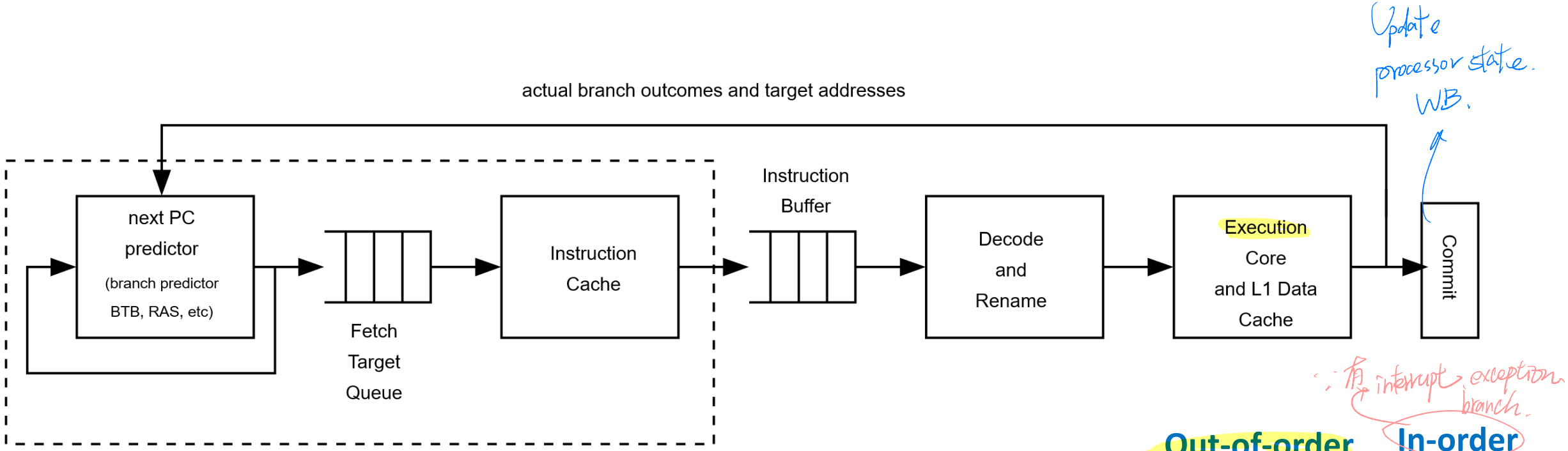
To expose more ILP, we need to consider:

- Branch prediction and speculative execution
- Removing name (or false) data dependencies
- Dynamic instruction scheduling

©BOLEDU

Superscalar Processors

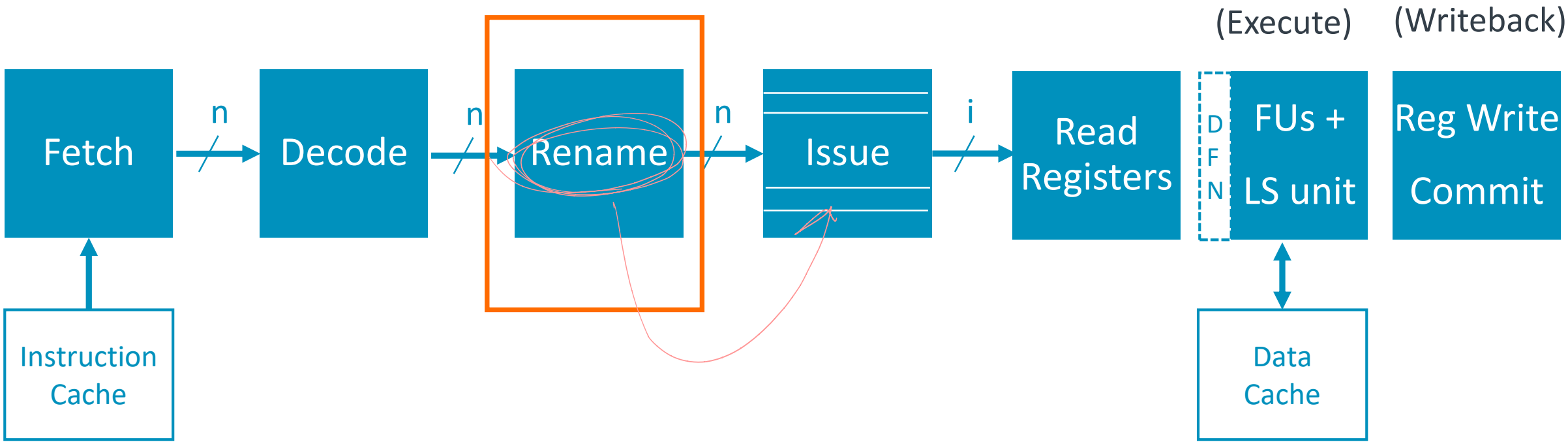
- Our instruction fetch (front-end) decoupled from the processor, run ahead, fill the instruction buffer, and keep execution units fed.



Note: each block may represent multiple pipeline stages

©BOLEDU

A Generic Superscalar Processor



LS unit = Load/Store unit

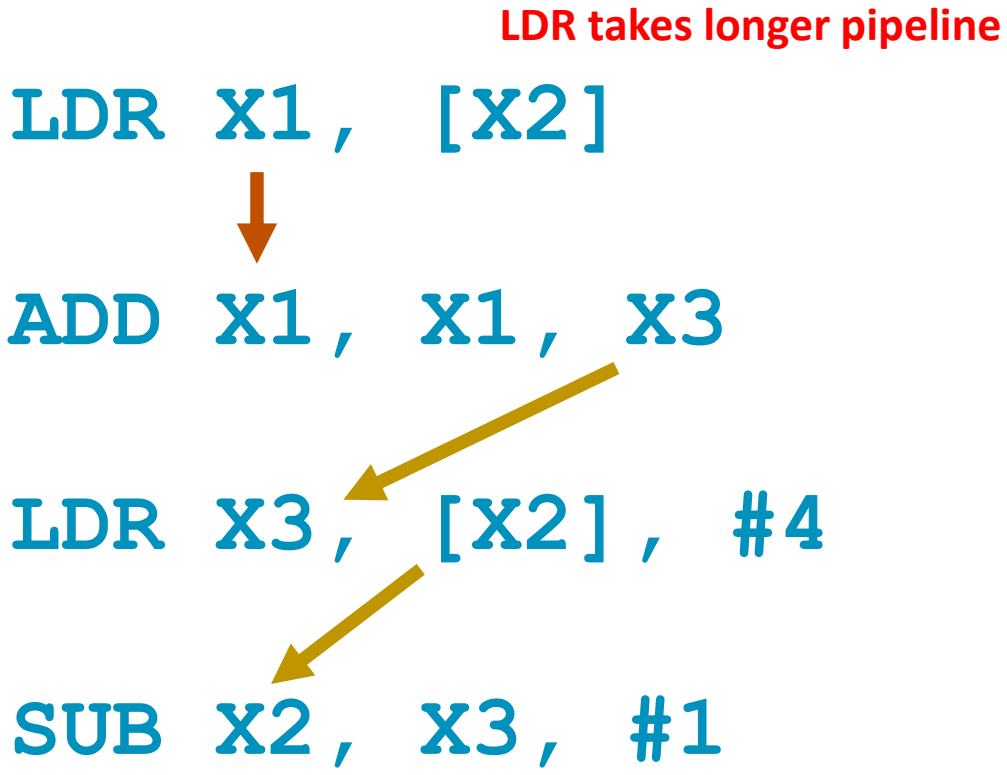
DFN = Data Forwarding Network

©BOLEDU

Data Dependencies – Name Dependencies

Name dependencies may also exist when two instructions refer to the same register. Unlike true data dependencies, no data are communicated:

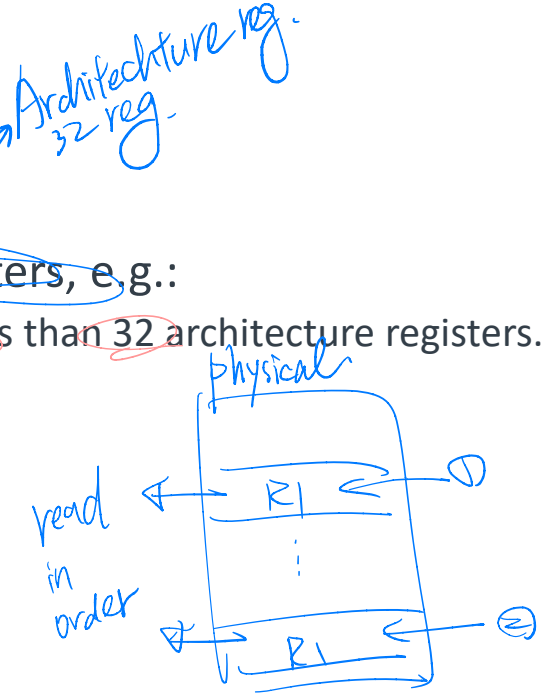
- Output dependencies Write-after-Write (red arrow)
- Anti-dependence Write-after-Read (gold arrow)



©BOLEDU

Register Renaming

- Name (or false) dependencies limits out-of-order instruction issue.
- The processor has many more physical registers than architectural registers, e.g.:
  - A higher performance Arm processor may provide 128 or more physical registers than 32 architecture registers.
- register renaming (in hardware)
  - change register names to eliminate WAR/WAW hazards



key: think of architectural registers as names , not locations

- can have more locations than names
- dynamically map names to locations
- map table holds the current mappings (name→location)
  - write: allocate new location and record it in map table
  - read: find location of most recent write by name lookup in map table
  - minor detail: must de-allocate locations appropriately

https://en.wikipedia.org/wiki/Tomasulo%27s_algorithm

©BOLEDU

Register Renaming Examples

- names: r1, r2, r3, locations: 11, 12, 13, 14, 15, 16, 17
- original mapping: r1→11, r2→12, r3→13 (14-17 "free")

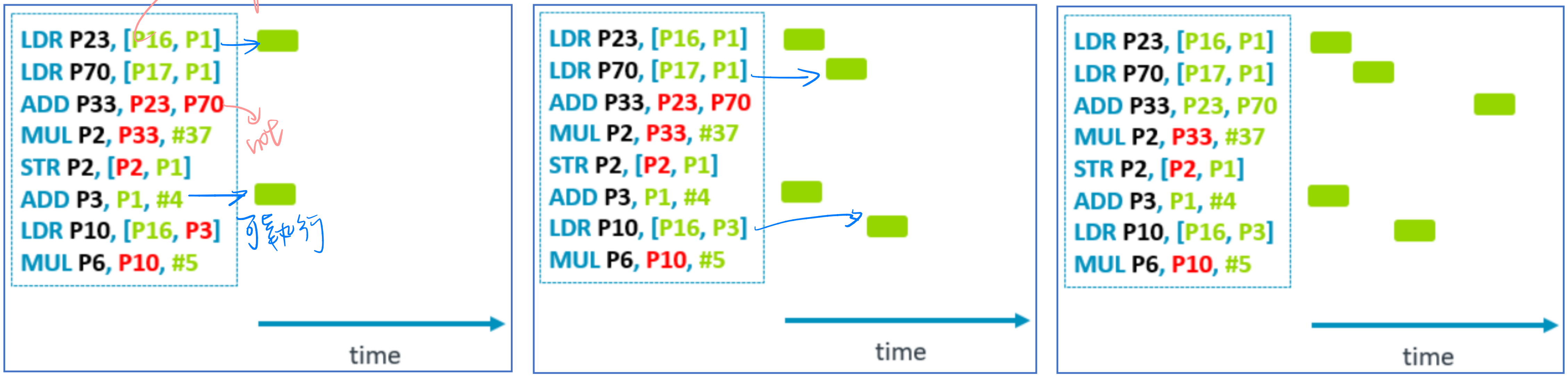
raw instructions	map table r1	map table r2	map table r3	free locations	renamed instructions
	11	12	13	14, 15, 16, 17	
add r1, r2, r3	14	12	13	15, 16, 17	add 14, 12, 13
sub r3, r2, r1	14	12	15	16, 17	sub 15, 12, 14
mul r1, r2, r3	16	12	15	17	mul 16, 12, 15
div r2, r1, r3	16	17	15		div 17, 16, 15

- **Each instruction has a unique physical destination register.**
- **All name dependencies removed.**
- **The processor is free to issue an instruction as soon as its operands are ready and an appropriate FU is free.**

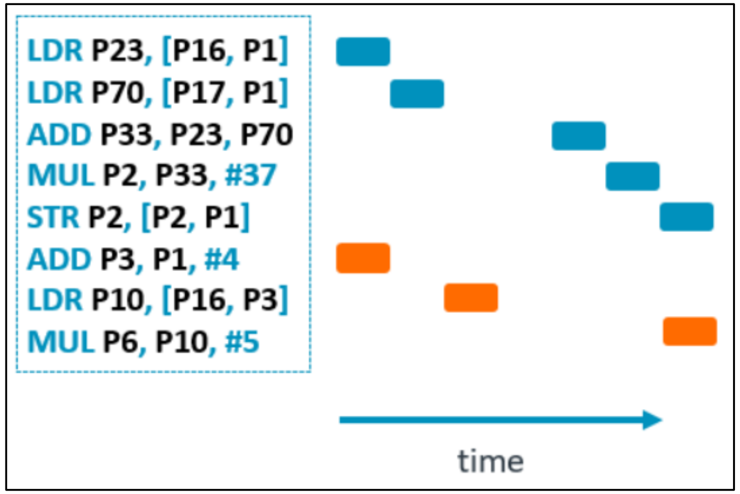
©BOLEDU

Out-of-Order Issues

Operands ready (green)
not available (red;)
destination registers (black)



- The processor can issue an instruction as soon as its operands are ready and an appropriate FU is free.
- When instruction issued, destination register is broadcasted
- Wakeup phase: waiting instructions compare the broadcast destination registers with their own operands. When the register identifiers match, the operand is marked as ready.
- **Selection and issue phase**: select as many ready instructions as possible and issue them to waiting FUs.

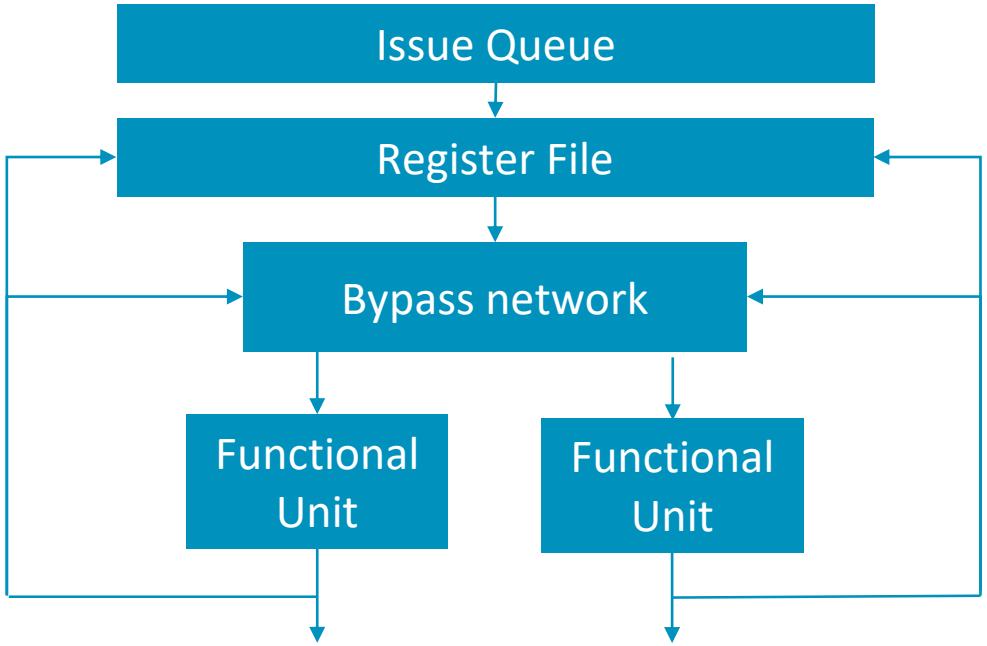


https://en.wikipedia.org/wiki/Out-of-order_execution

©BOLEDU

Superscalar Processors: Data Forwarding (Bypass) Network

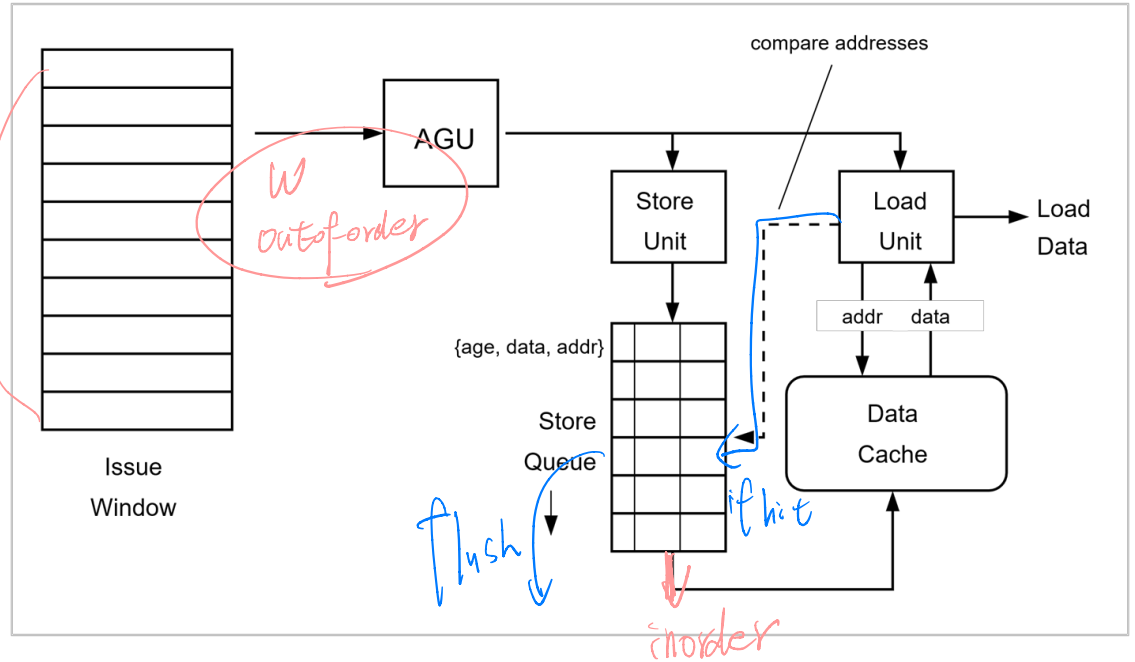
- Data forwarding in a scalar pipeline is relatively simple, consisting of a few extra buses and multiplexers.
- In a superscalar processor, we have many parallel functional units and may need to forward any recently generated results to the input of any functional unit. For example:



©BOLEDU

Loads and Stores

- Memory-carried data dependency
- Store operations cannot be undone.
  - Can not do “speculative” store.
  - Ensure Precise exceptions
- Execute stores in program order.
  1. Issue load/stores out-of-order to Address Generation Unit (AGU).
  2. Buffer stores and only execute them in program order.
  3. For loads, check **all addresses** of older stores. If any match or addresses are unknown, stall load; otherwise, it may access the data cache (**load-bypassing**).



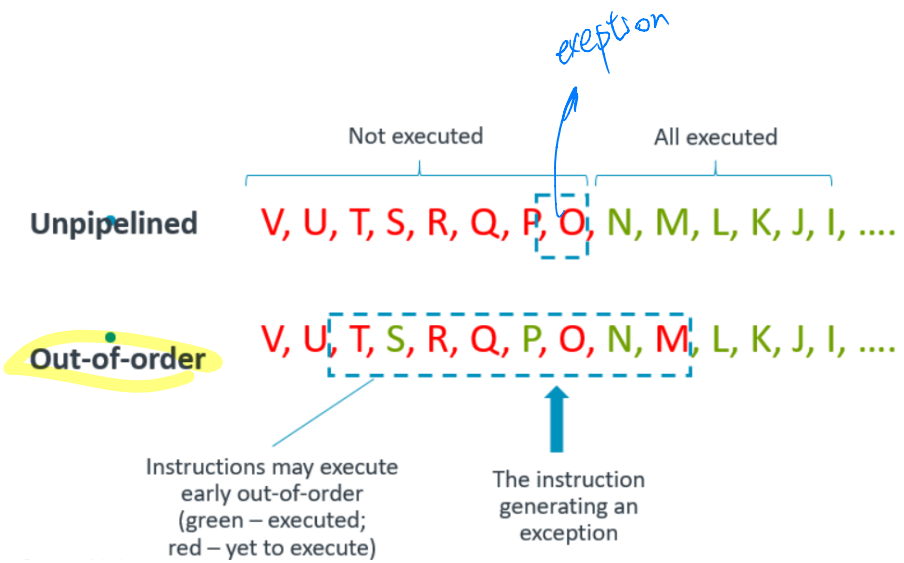
Advanced Features

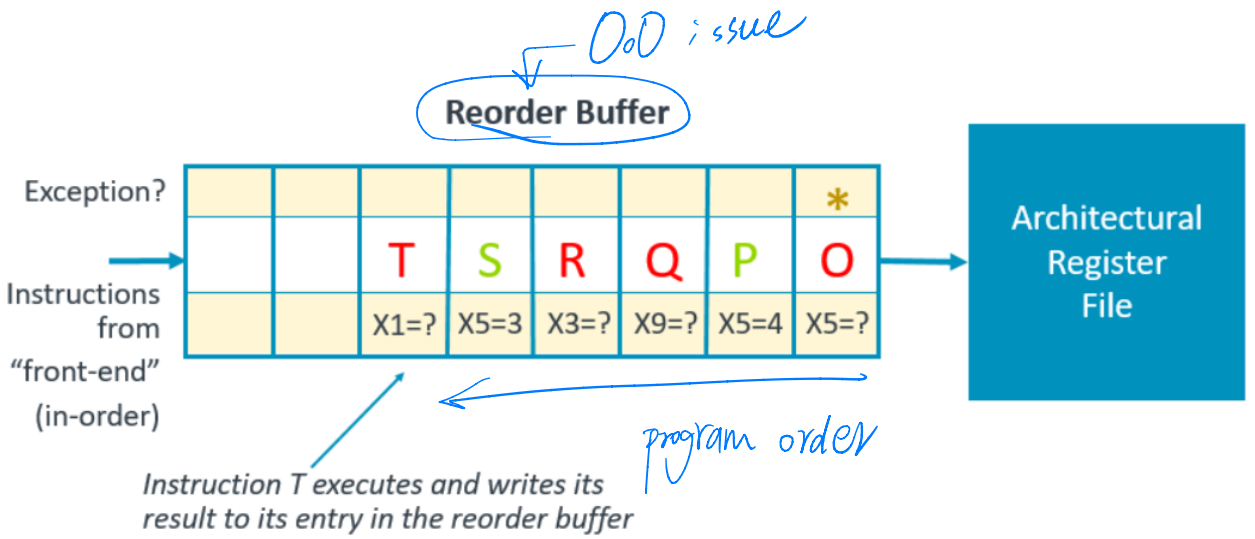
- Store-to-load forwarding
  - Allows data to be forwarded directly from a pending store to a load instruction
- Speculative loads
  - Allow loads to access the data cache speculatively even when there are older stores that have not calculated their addresses

©BOLEDU

Handling Mispredicted branches and Precise Exceptions

- Mispredicted branches and exceptions force us to roll back state.
  - Track the architectural state of the processor. i.e., the state corresponding to the **in-order** execution of instructions
- Reorder Buffer: Buffer results generated out-of-order



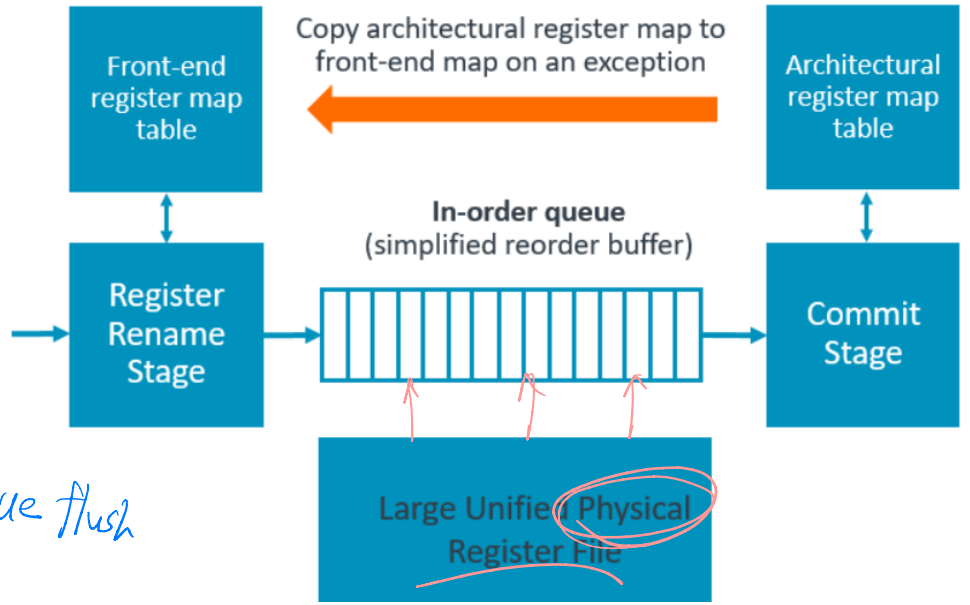


©BOLEDU

The Reorder Buffer – Committing Instructions

When an instruction reaches the end of the reorder buffer, we know all earlier instructions have completed. At this point, we can:

- Update our (architectural) register file. (Note: we have rename registers to reorder buffer entries)
- Check if branches have been mispredicted.
  - If so, flush the reorder buffer and re-execute the branch.
- Check if the instruction needs to raise an exception.
  - If it does, flush the reorder buffer and raise the exception.
- Signal that store operations can write to the data cache.



©BOLEDU

Example: Putting It All Together (Cortex-A77, 2019)

- The Cortex-A77 can fetch and decode 4 instructions/cycle.
- It can issue (dispatch) up to 10 uops/cycle to the integer, FP, and load/store units.
- The branch mispredict penalty is 10 cycles in the best case.
- The out-of-order windows size and reorder buffer hold 160 instructions.
- The target clock frequency is between 2.6 and 3 GHz.

©BOLEDU

Limits to Superscalar Processors

Ultimately, the performance of a superscalar processor is limited by:

- Increasing hardware cost of extracting more ILP
- Memory bandwidth
- Limits to branch prediction and caches
- Interconnect scaling
- Power consumption

©BOLEDU

Data Parallelism - SIMD

Ref: “processor-simd-ref.pdf”

©BOLEDU

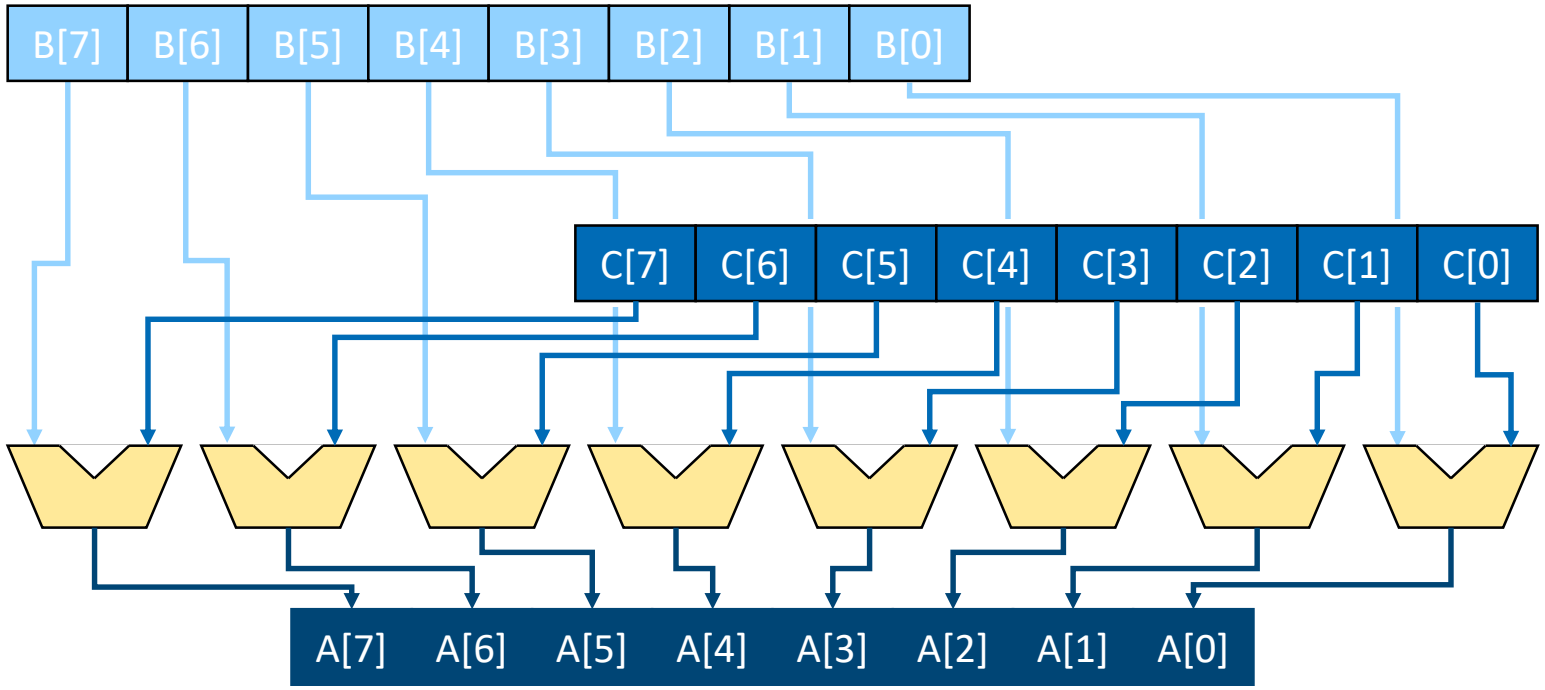
SIMD

- Exploit data-level parallelism
  - Perform operation on array of data together, i.e. performing many register-register with the same opcode
  - Provide energy-efficient computation (amortizing the costs of fetch and decode)
- SIMD also provides more efficient processing for certain codes.
  - E.g., multimedia workloads, such as image recognition and object detection, where operations are on pixel color values that are 8 or 16 bits in length
  - Scalar execution would put each value in its own 32-bit register.
  - SIMD packs them into a vector register (e.g., 256-bit vector of 16 * 16-bit data elements).
    - Then operates on each element independently
- SIMD typically operates on all elements concurrently.

©BOLEDU

SIMD Execution

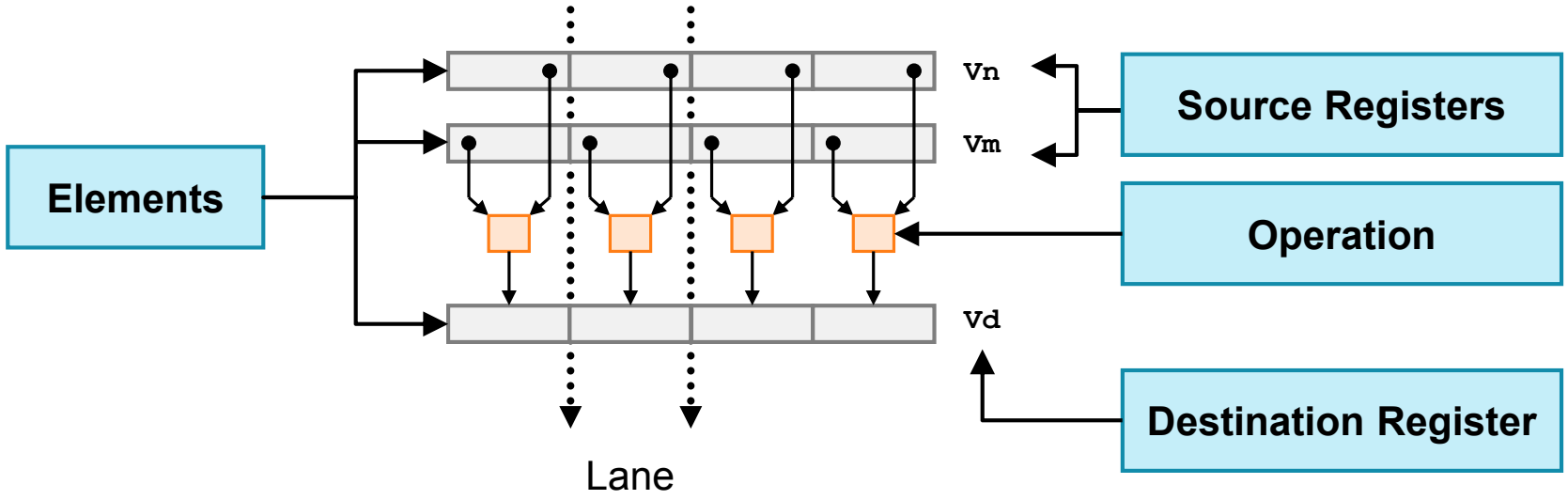
- In SIMD, operations on all data elements occur at the same time.
- The processor provides as many FUs as there are data elements in a vector.



©BOLEDU

Case Study: AArch64 NEON

- NEON is a wide SIMD architecture developed for multimedia applications.
- Registers are considered as vectors of elements of the same data type (128 bits in size).
  - Integer signed and unsigned 8-bit, 16-bit, 32-bit, 64-bit
  - Floating point half, single and double precision
  - Instructions usually perform the same operation in all lanes.



©BOLEDU

Multithreading

Ref: “processor-multithread-ref.pdf”

©BOLEDU

Multicore (Multi-tasking) v.s. Multithreading

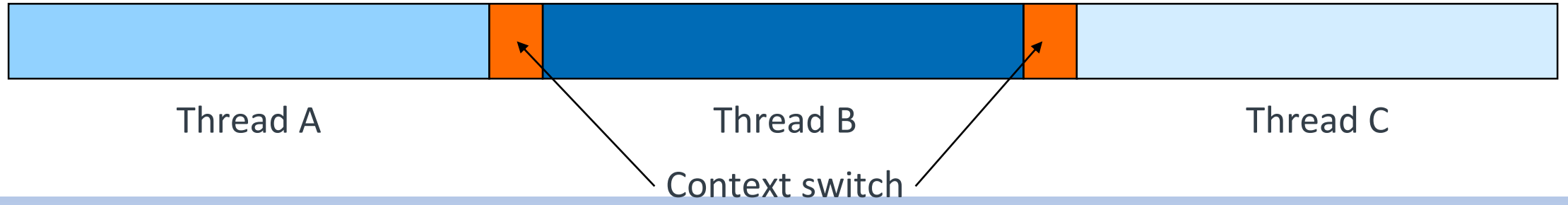
- Multicore processors
  - Allow thread-level parallelism (within a program) and process-level parallelism (across applications).
  - Simply replicate the core several times on the chip.
  - Add the logic for implementing cache coherence, etc.
  - Simple replication of the core increases Area/Power cost
- Multithreading

  - Add extra storage to the core to hold the context of the thread
  - Context switch is faster
  - Switch granularity – fine-grain or course-grain

©BOLEDU

Multitasking

- Each core shared by multiple threads – controlled by the OS
- Context switch – OS saves the thread's state & replaces with the state of another thread
  - Contents of the (architectural) register file
  - Program counter (PC)
  - Memory-management information, such as the page table base register
  - Usually, 100 s of cycles - **Very slow**
- Can we improve it with few cycles for context switching and less cost ?
  - Multi-threading



©BOLEDU

Fine-grained Multithreading

- Switch to a new thread every cycle - each pipeline stage is an instruction from a different thread.

≃ multi-cycle processor

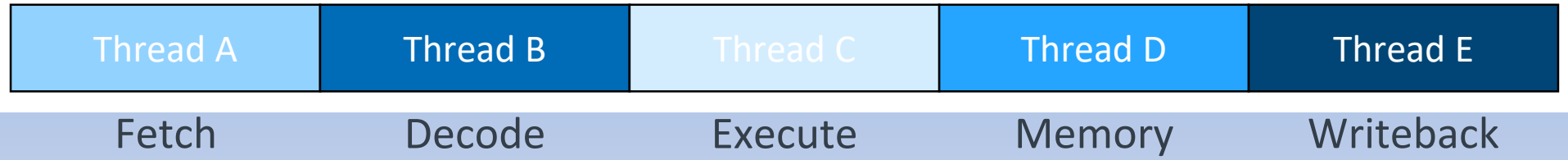
Benefits

- No thread-switch overhead
- **No need for pipeline interlocking** no dependency
  - No two instructions will have a data dependency through registers (they might through memory).
- **No need for branch prediction logic**
  - All branches executed before the next instruction are fetched from the same thread.
- Improved core throughput

Downsides

- Need to support a large number of threads
  - So as to hide most short-latency pipeline bubbles and avoid nops
  - This comes at an area cost.
- Single-threaded performance suffers.
  - Since threads only get an instruction fetched every N cycles
  - But this can bring strong performance guarantees

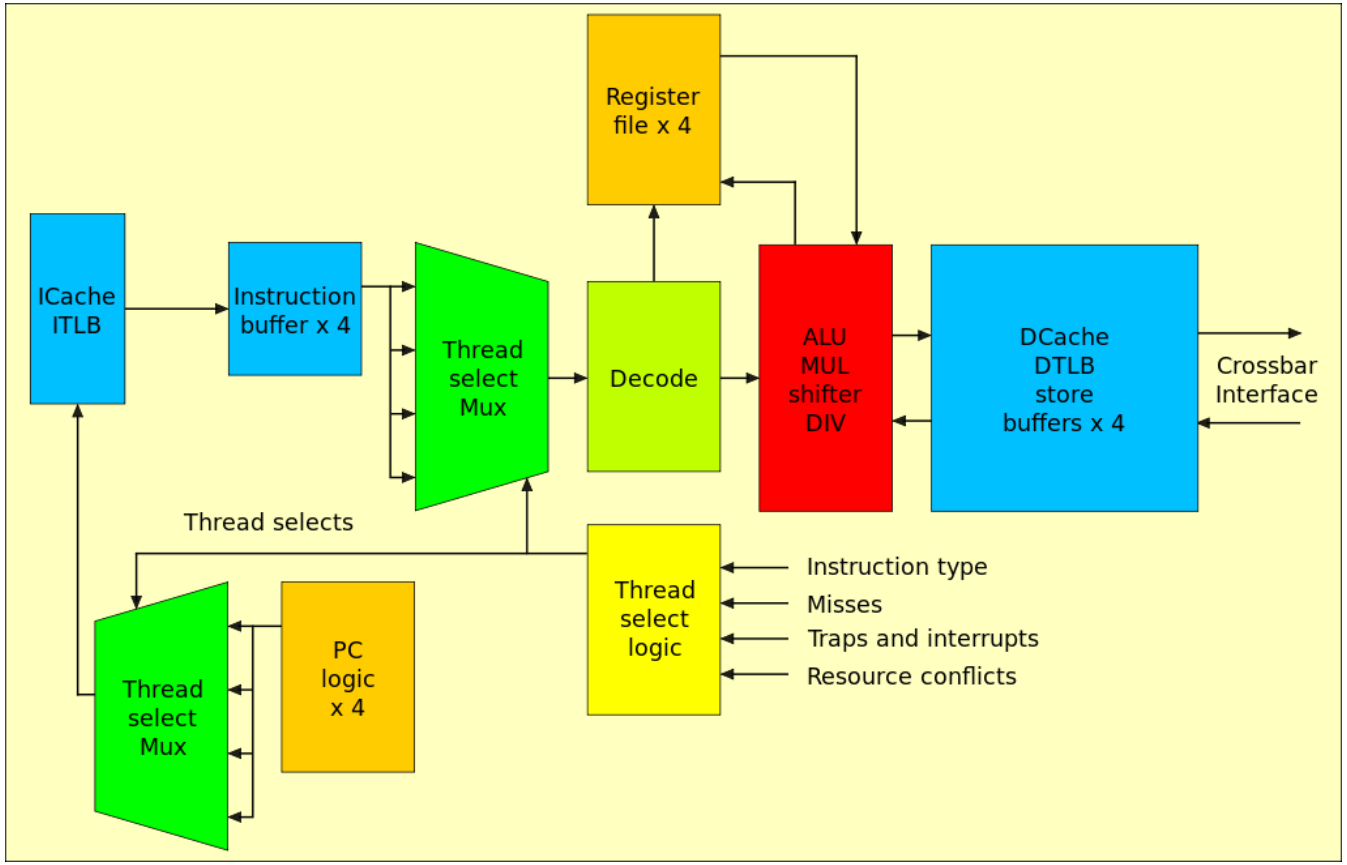
Improve Throughput by Dynamic Thread Selection



©BOLEDU

Case Study: Sun's Niagara (UltraSPARC T1)

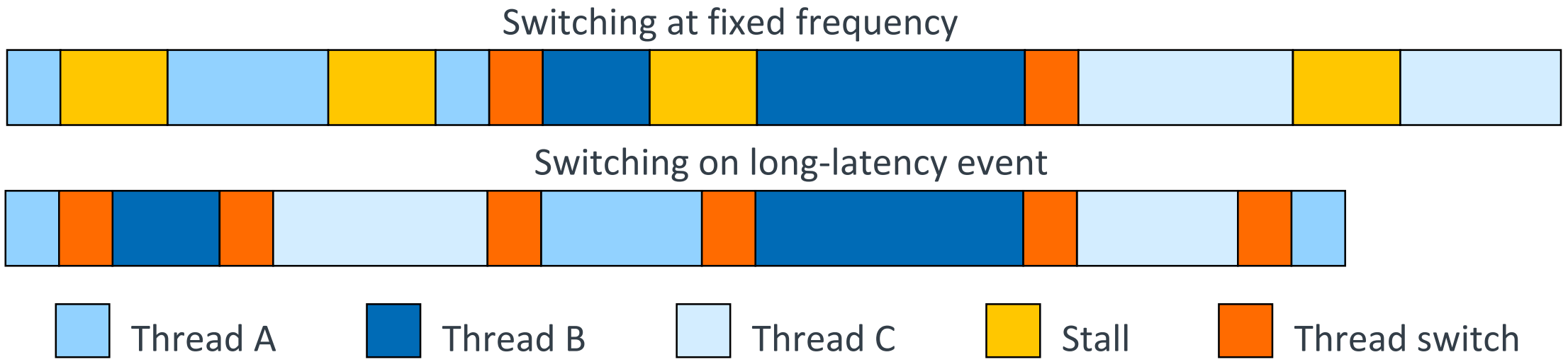
- Each core of eight supports four concurrent threads.
- Threads removed from selection on long-latency events
- This allows provision of only 16KiB instruction and 8KiB data caches.



©BOLEDU

Coarse-grained Multithreading

- Switching less often (and flushing when doing so) – e.g. Switch when stalled
- To avoid the design complexity of handling instructions from different threads in the pipeline at any given time, the pipeline has to be flushed before switching.



©BOLEDU

Coarse-grained Multithreading

Reasons for switching threads

- L1 or L2 cache miss
  - Accessing main memory takes hundreds of cycles, and if the thread-switch penalty is small enough, even an L1 miss can be partially hidden.
- Complex ALU operation
  - A floating-point divide may take tens of cycles and may not be pipelined.
- Timeout
  - This ensures fairness for compute-bound threads.
- Higher priority thread becomes ready.

Methods to reduce thread-switch penalty

- Implement a short pipeline.
  - This reduces the time until the pipeline becomes full again.
- Add a thread-switch buffer
  - Essentially prefetch a few instructions from each thread into a buffer so that they can be fetched (for real) immediately after a switch.
- Provide pipeline registers for each thread.
  - So now no instructions need to be squashed on a long-latency event, but added complexity.

©BOLEDU

Case Study: Intel Montecito (Itanium 2)

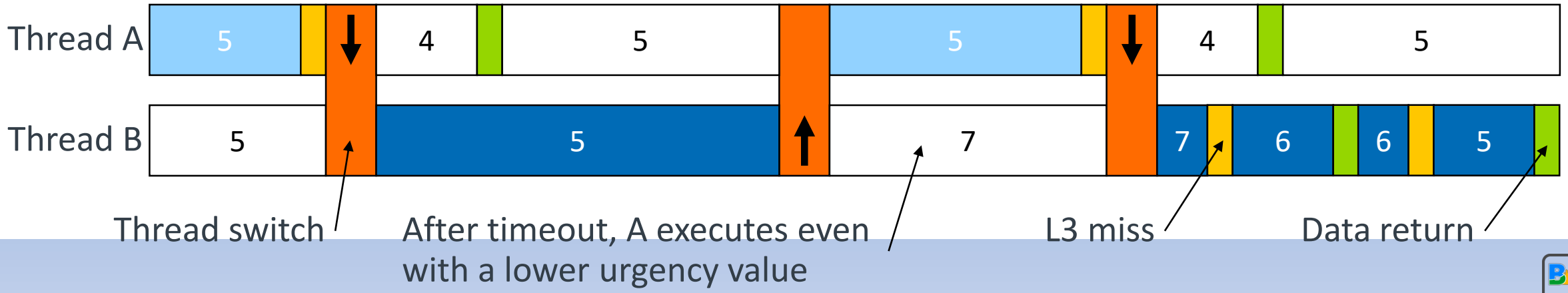
- Support for two threads per core
  - Duplicates all architectural and some microarchitectural state
- Each thread is given an “urgency” value.
  - Larger urgency means higher priority.
  - Urgency is dynamically updated based on system events.

Thread switching possibly occurs on

- L3 cache miss or data return
- Timeout, software hint, or other system event

Diagram shows two threads with events.

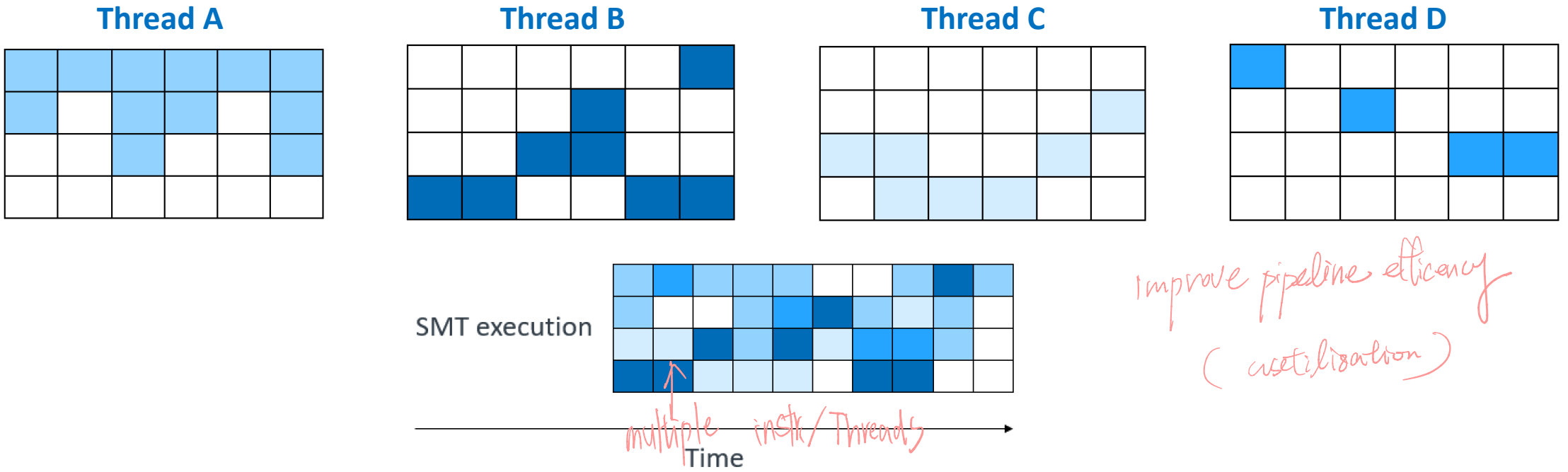
- Colored when switched in, white when switched out
- Urgency value inside boxes



©BOLEDU

Simultaneous Multithreading

- Instructions from multiple threads can occupy a pipeline stage at the same time
- Microarchitectural resources throughout the pipeline are shared between threads.
- Duplicate architectural state. - PC, register file, page table base register, etc.
- Fetch instructions from multiple threads in each cycle – tag theads in the pipeline



©BOLEDU

SMT Policies

Fetch policies

- Flexibility in how instructions enter the pipeline
- Fetch from one thread only in each cycle
  - Round-robin selection
  - Prioritize one thread over others, for example, the one with the fewest in-flight instructions
- Fetch from multiple threads
  - Needs additional logic to fetch from multiple cache lines in the same cycle

Issue policies

- Flexibility in how instructions are issued to the functional units when ready
- Issue from all threads in each cycle
  - Skip threads that have no ready instructions
- Prioritize instructions according to
  - Their age (oldest first)
  - Whether they are speculative (e.g., following an unresolved predicted branch)

©BOLEDU

SMT Downsides

- There are two main drawbacks with SMT, and fine-grained multithreading.
- First, single-threaded performance may suffer.
  - If two or more threads execute, they will contend for resources.
  - Given the extra logic required within the core, the clock rate may not be as high as it could be otherwise.
- Second, the complexity of the core increases.
  - Extra hardware required throughout the pipeline, as well as within the TLB
  - This requires more design, test, and verification time.
  - The load/store queue, in particular, requires careful attention so as to respect the memory consistency model.

©BOLEDU

SMT vs Multicore

SMT

- Higher utilization of core structures through sharing
- Single-threaded performance can suffer.
- For a modest area and power increase, this provides support for multiple threads.
- Memory-bound applications are a good fit because idle cycles can be filled.
- CPU-bound applications are a poor fit because they are likely to slow down.

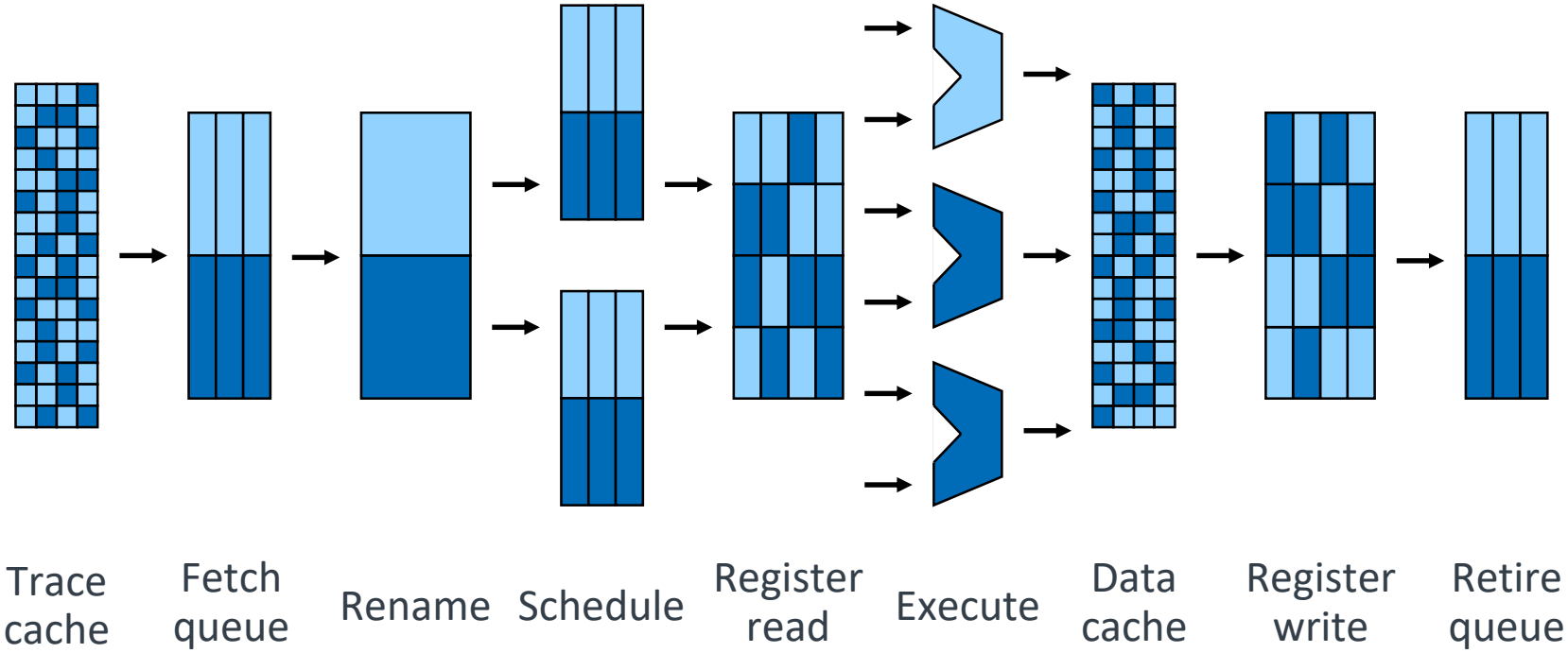
Multicore

- Cores are fully duplicated, so threads have dedicated resources to use.
- Single-threaded performance maintained
- Power and area at least doubled through core duplication
- Memory-bound applications don’t make best use of the core’s resources.
- CPU-bound applications maintain their high IPC.

©BOLEDU

SMT Case Study

- Intel’s Netburst microarchitecture supports 2 threads.
- Performance boosted by 25%
- Die area increased only 5%
- Three types of sharing for microarchitectural structures



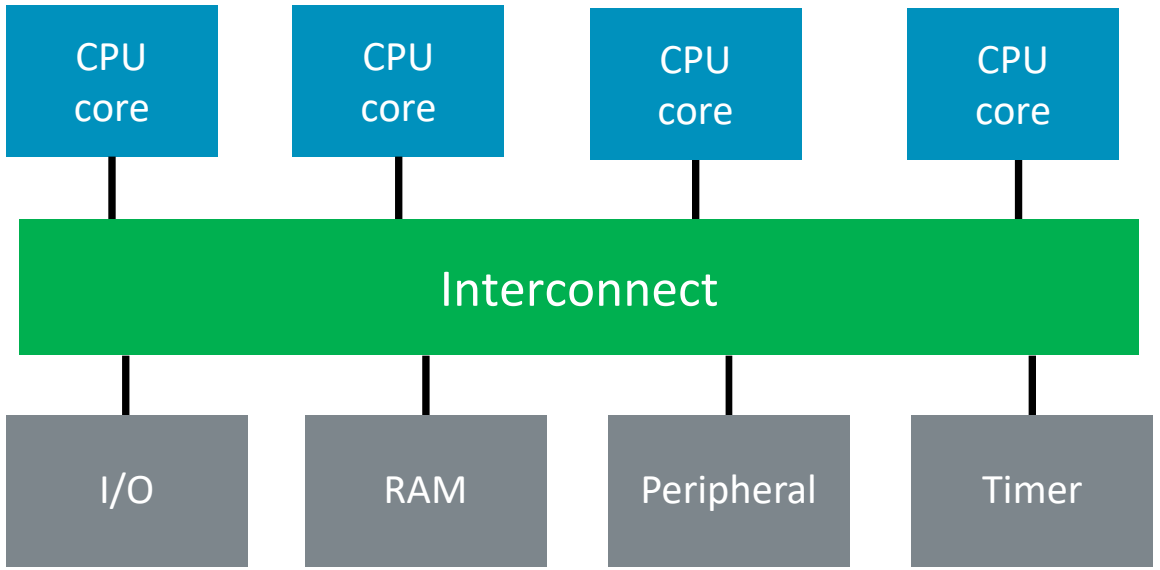
©BOLEDU

Multicore

©BOLEDU

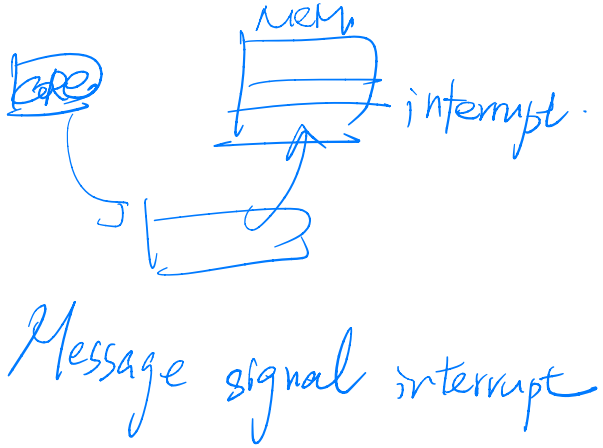
Motivation

- Moore's law - transistors available to an architecture keeps increases
- Performance improved through speculation - sublinear improvement
- Multicore architectures are an efficient way of using these transistors instead.
  - Performance comes from parallelism, specifically thread-level or process-level parallelism.
  - Note that we had multi-processor systems long before Dennard scaling failed; this just pushed them to mainstream.



©BOLEDU

Design Issues with Multicore



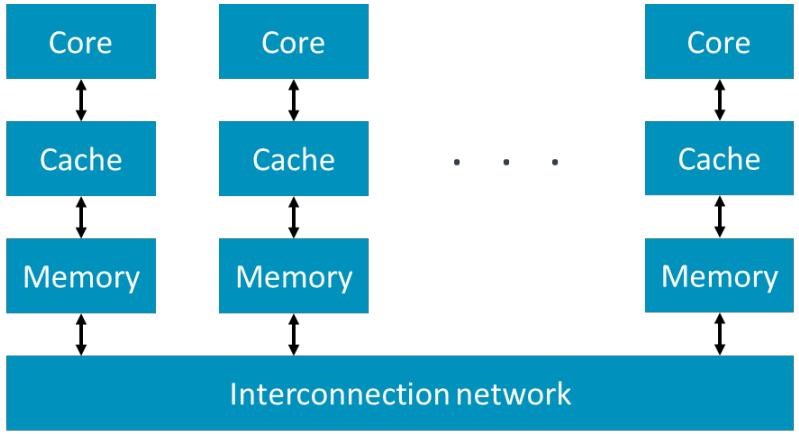
- How do cores communicate with each other?
  - **Shared Memory and Message Passing**
- How is data synchronized? How do we ensure that cores don't get stale data when it's been modified by other cores?
  - **Cache Coherence**
- How do cores see the ordering of events coming from different cores?
  - **Memory Consistency**

©BOLEDU

Communication

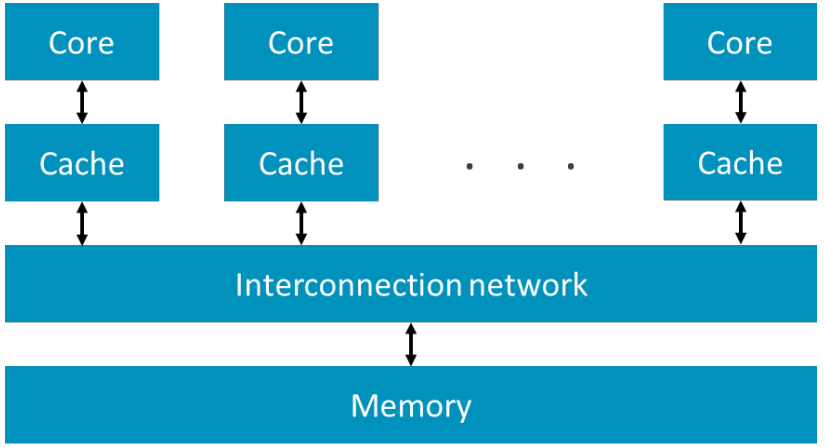
Message Passing

- Do not reply on shared memory space
- Each PE has its core, data, I/O
  - Explicit I/O to communicate with other core
  - Explicit communication via *send* and *receive* operations
- Advantage
  - Less hardware, easier to design
  - Focuses attention on costly non-local operations



Shared Memory

- Shared memory address space accessible to all cores
- Cores may have caches holding data.
  - Communication by read/write to memory.
  - Synchronization via atomic operations to modify memory
- Advantages
  - Matches the programmer's view of the system
  - Hardware handles communication implicitly



©BOLEDU

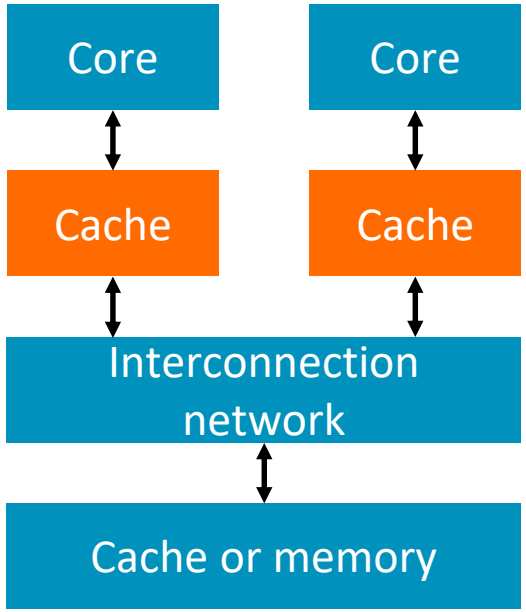
Cache Coherence

MESI Protocol – Refer “processor-multicore-ref.pdf”

©BOLEDU

Cache-coherence Protocol

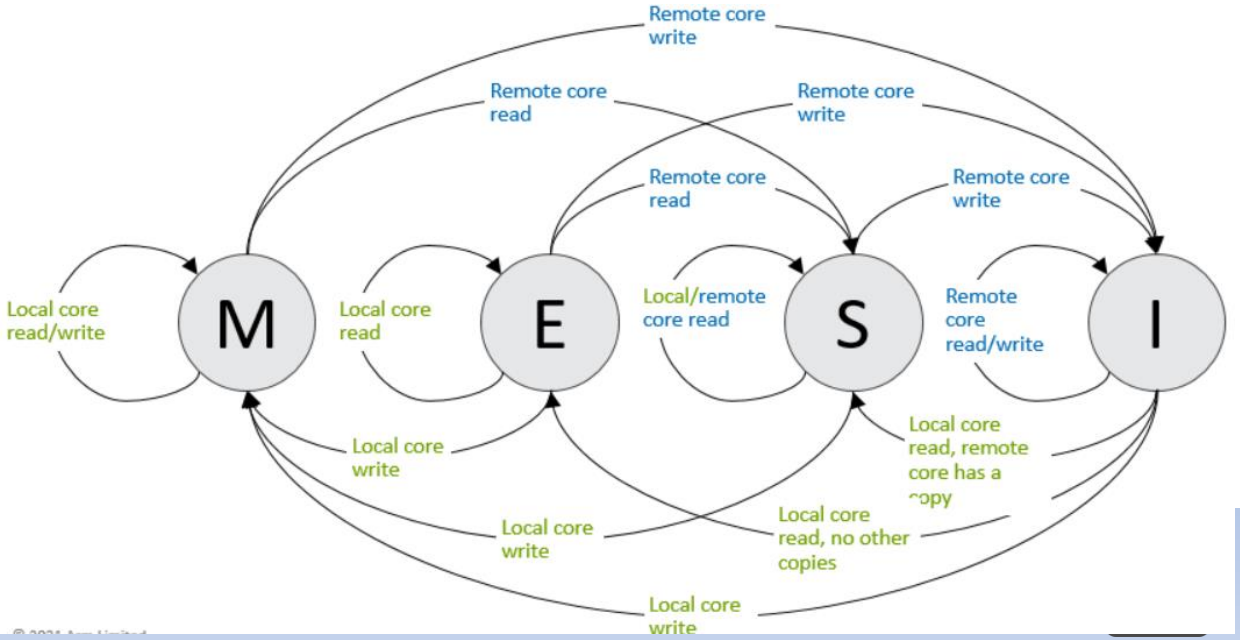
- The cache-coherence protocol ensures that cores always read the most up-to-date values.
  - This means a core can always find the most recent value for some data, no matter where it is.
  - It describes the actions to take on seeing certain events from other cores.
- Cache coherence is required when caches share some memory.
- The protocols rely on caches seeing events from other cores.
  - In particular, reads and writes to the shared memory
  - Often this is realized through snooping operations on the interconnect.
- The protocol runs on each block in the cache independently.
  - This is the granularity commercial implementations track the state information.
- It runs in the private caches that have a shared ancestor.
  - The orange caches in the diagram



©BOLEDU

MESI Cache-coherence Protocol

- **Allocate-on-write**: New data are loaded into the cache on write misses
- **Write-back cache**: If Write hit, write to cache only.
- **Write-Invalidate**: Writing to shared region, invalidated the caches of all other cores.
- The MESI protocol defines states for each cache block and transitions between them.
  - M: Modified/UniqueDirty (UD) – the line is in only this cache and is dirty.
  - E: Exclusive/UniqueClean (UC) – the line is in only this cache and is clean.
  - S: Shared/SharedClean (SC) – the line is possibly in other caches and is clean.
  - I: Invalid/Invalid (I) – the line is not in this cache.



©BOLEDU

Memory Consistency

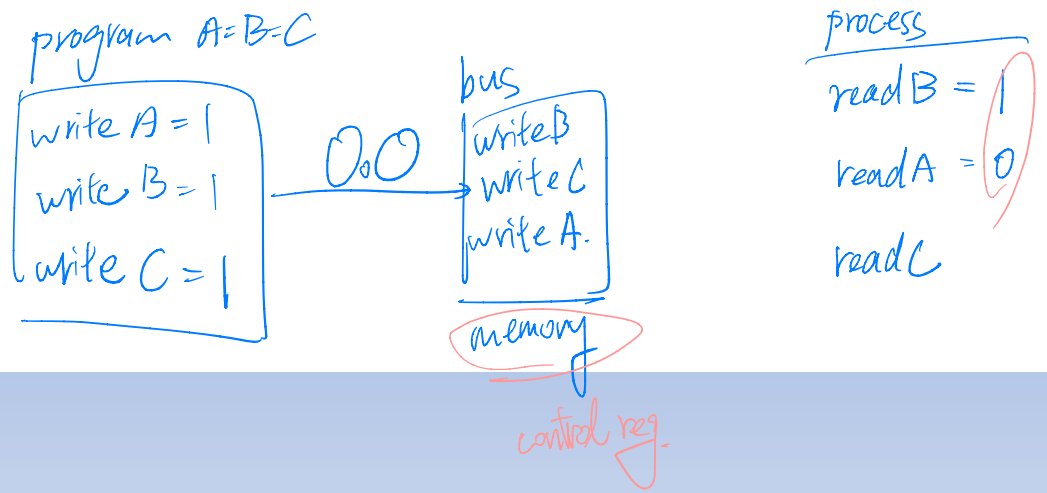
https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf

©BOLEDU

Cache Coherence v.s. Memory Consistency

- **Cache coherence** is to ensure a global order for write to a location are seen in the same order by all processor. (run independently on each block of data – cache line)
  - So when data are written by one core, all other cores can later read the correct value.
  - When a core attempts to write, others know that their copies are stale.
  - When a core attempts to read, others know they must provide their data, if modified.

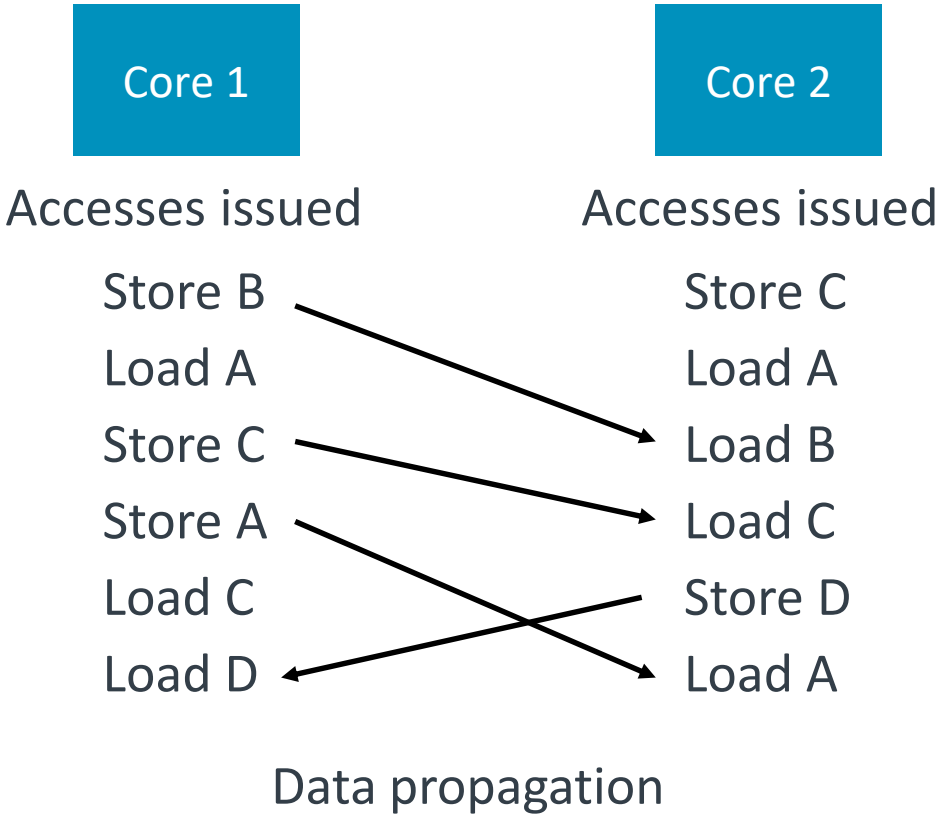
- **Memory Consistency** – ordering of operations to multiple locations wrt all processors.
  - Processor reorders memory operations (avoid stalling read)
  - In what order do other cores see them? What is the effect ?
  - Define rules for the apparent order and visibility of updates



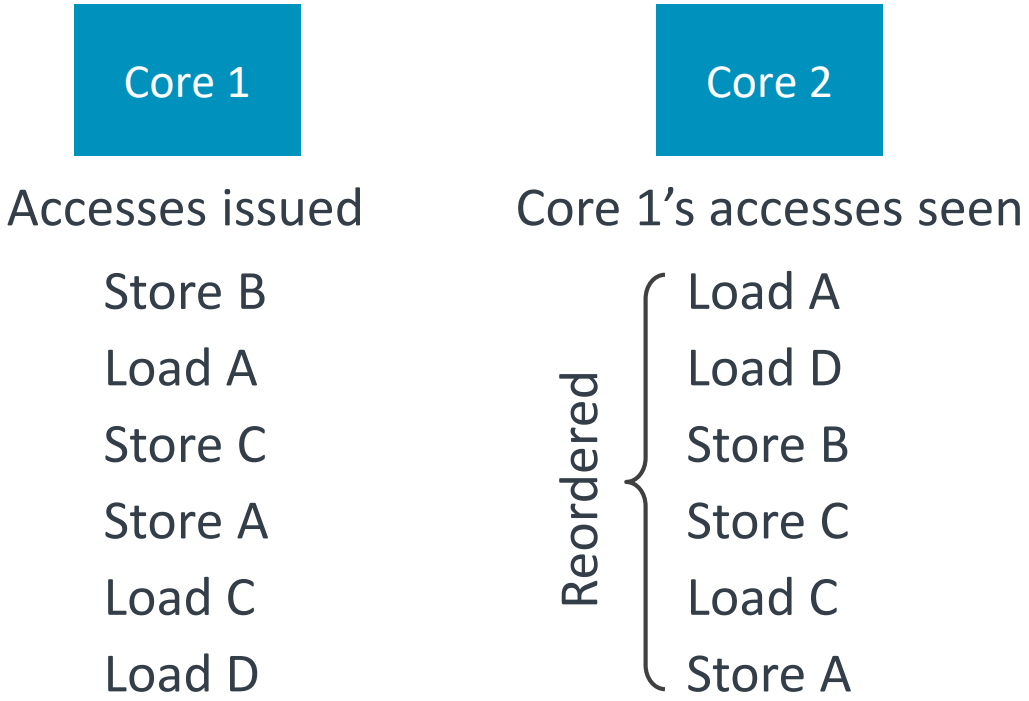
©BOLEDU

Memory Consistency vs Cache Coherence

Cache coherence



Memory consistency



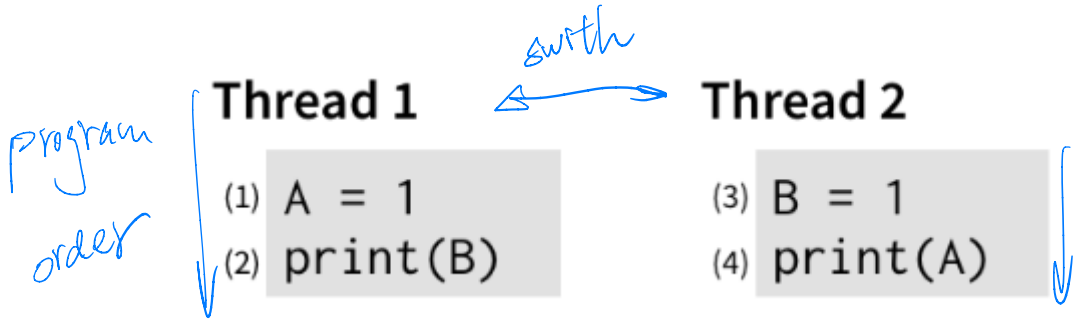
©BOLEDU

Memory Consistency

- The memory consistency model defines valid outcomes of sequences of accesses of the different cores.
- **Sequential consistency (SC)** is the strongest and most intuitive model.
  - The operations of each core occur in program order, and these are interleaved (at some granularity) across all cores. (no loads or stores can by pass other loads or stores)
  - SC is overly strong, and constrained. Prevent useful optimization.
- **Total store order (TSO)** is widely implemented (e.g., x86 architectures).
  - Allow to use store buffer to hide write latency.
  - The same as sequential consistency but allows a younger load to observe a state of memory in which the effects of an older store have not yet become observable (buffered)
- **Weak Ordering** have been adopted (e.g., Arm and PowerPC architectures).
  - Allow almost any operation to be reordered.
  - Use synchronization operation (barrier/fence – forces all memory operation before it to complete before any memory operation after it can begin)

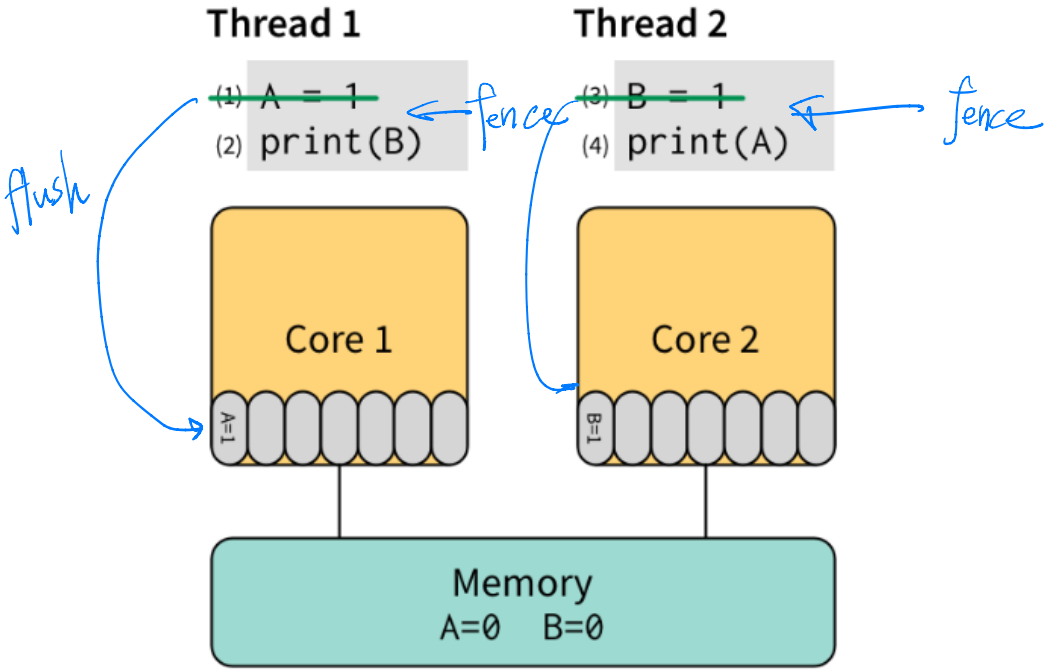
©BOLEDU

Sequential Consistency (SC)



1->2->3->4 => 01
3->4->1->2 => 01
1->3->2->4 => 11
1->3->4->2 => 11
It can not print 00

Total Store Ordering (TSO))



1 (buffered) -> 2 -> 3 (buffered) -> 4 => 00

©BOLEDU

Exceptions

© 2021 Arm Limited

Exceptions

- In some situations, we are required to interrupt a program's execution and take some action. These conditions or system events are called *exceptions*. The necessary action is taken by privileged software, i.e., the *exception handler.*
- Exceptions may occur for many different reasons, e.g.,
  - A page fault, breakpoint, I/O device request, floating-point errors, memory protection violation, etc.

© 2021 Arm Limited

Exceptions

Types of exceptions (for Arm):

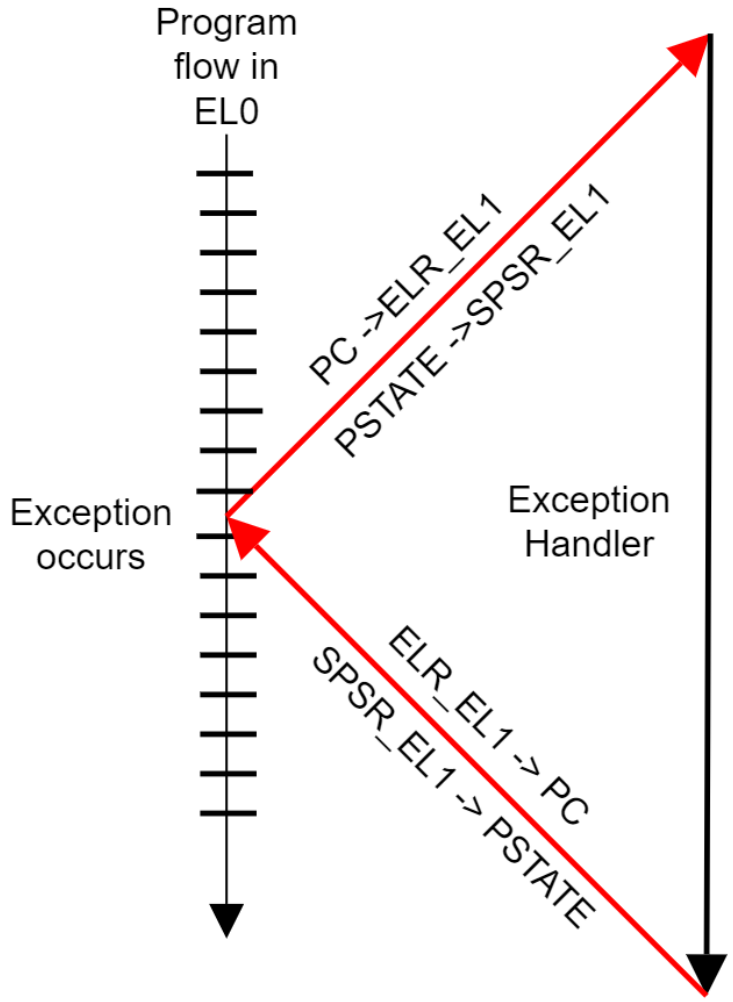
- Interrupts
- Aborts
- Reset
- Exception generating instructions

© 2021 Arm Limited

Exceptions

An exception will cause the processor to perform the following:

1. Save the Processor State (PSTATE), e.g., processor flags, interrupt mask bits, exception level, etc.
2. Save the return address (current PC).
3. Branch to handler specified in vector table
4. Save registers, execute handler code, restore registers.
5. Return from exception (ERET instruction).



© 2021 Arm Limited

Exceptions

The intention is to temporarily interrupt program execution, deal with the exception, and then to resume execution.

A good way to ensure we can easily resume is to ensure that the architectural state is consistent with the sequential model of program execution before the exception is taken, i.e., if the instruction that causes the exception is instruction E:

1. All instructions prior to E should have completed and updated their destination registers. All exceptions caused by these instructions should have been handled.
2. Any instructions after E in program order should not have completed and not have modified any processor state.
3. Whether E should complete or not will depend on the exception.

These are called "**precise exceptions.**"

© 2021 Arm Limited

Precise Exceptions and Pipelining

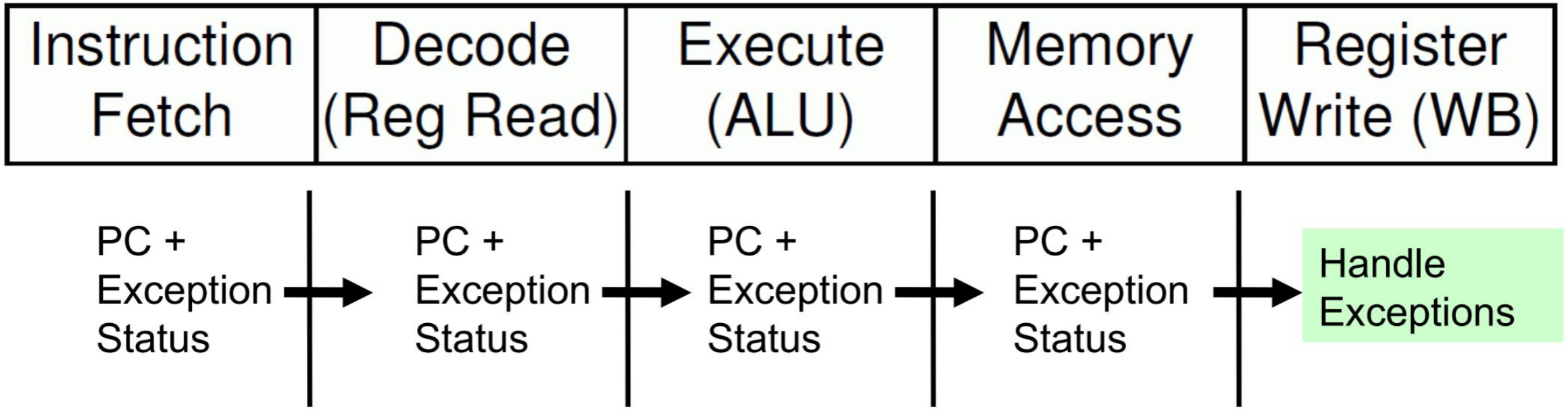
The requirements on the previous slide are trivial in the case of an unpipelined processor, but more complex for a pipelined processor.

Unpipelined

Pipelined

© 2021 Arm Limited

Precise Exceptions



© 2021 Arm Limited
arm