Parallel Programming and Heterogeneous Computing

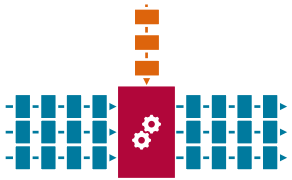
Shared-Memory Hardware

Max Plauth, Sven Köhler, Felix Eberhardt, *Lukas Wenzel* and Andreas Polze

Operating Systems and Middleware Group

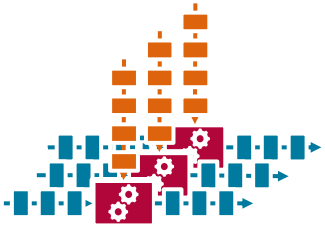
Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Recap:
Types of Parallelism



- **Data Level Parallelism**

  The same operation is applied in parallel to multiple units of data.



- **Task Level Parallelism**

  Multiple operations are executed in parallel.

  - **Instruction Level Parallelism (ILP)**

    ... between operations in a task

  - **Thread Level Parallelism (TLP)**

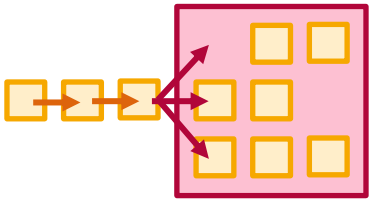
    ... between multiple tasks within a workload

  - **Request Level Parallelism**

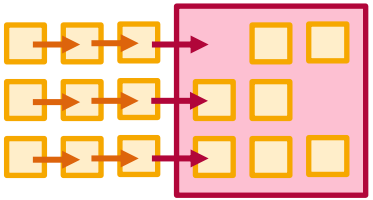
    ... between multiple workloads

Shared-Memory Hardware
Exploiting Instruction Level Parallelism

- ILP arises naturally within a workload
  - Programmers think in terms of a single instruction sequence
- TLP is explicitly encoded within a workload
  - Programmers designate parallel operations using multiple tasks



ILP



TLP

Why consider ILP in a parallel programming lecture?
Knowledge of common ILP mechanisms and assumptions enables performance optimization on single-thread granularity!

Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Pipelining

- Instruction execution phases (e.g. Instruction Fetch, Decode, Execute, Memory Access, Writeback) employ distinct hardware units
  - Without pipelining only one unit would operate each clock cycle
- Pipelining increases throughput by utilizing all units in every cycle
- Latency per instruction remains the same

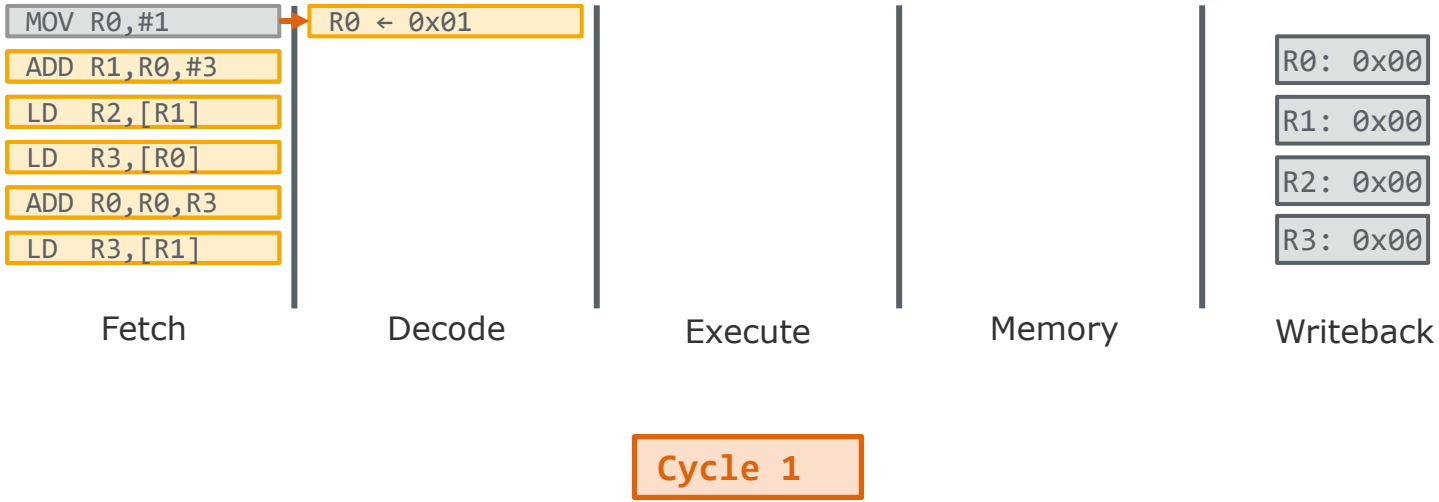
15 Cycles

20% Utilization

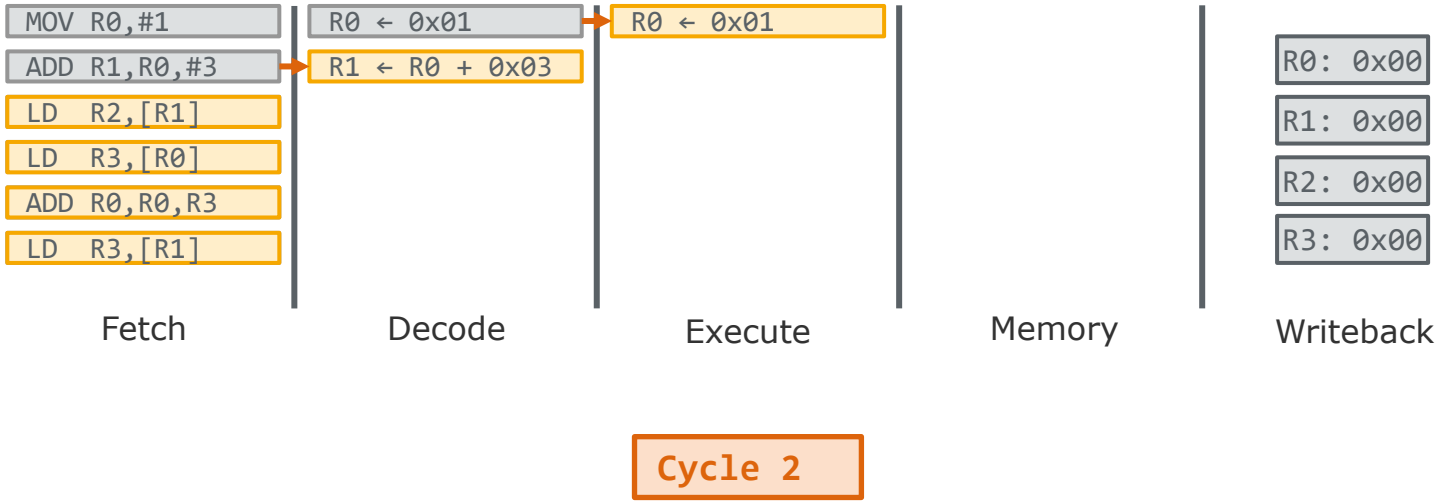
7 Cycles

Approaching 100% Utilization

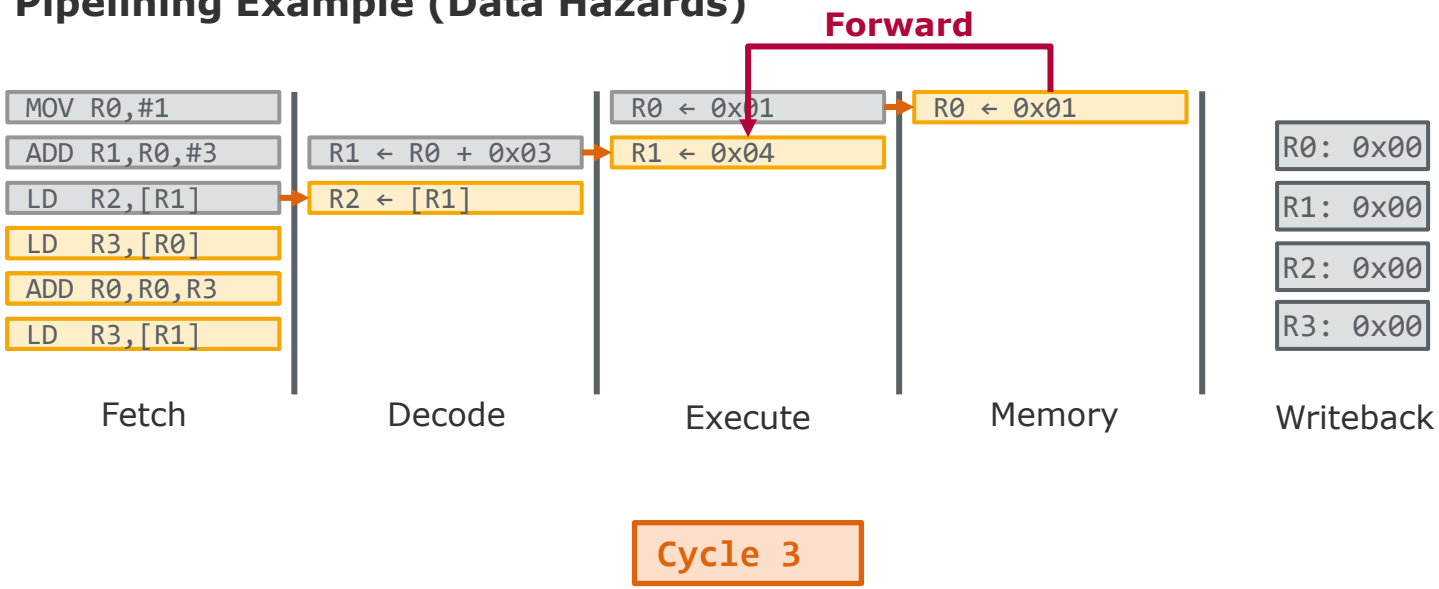
Pipelining Example (Data Hazards)



Pipelining Example (Data Hazards)

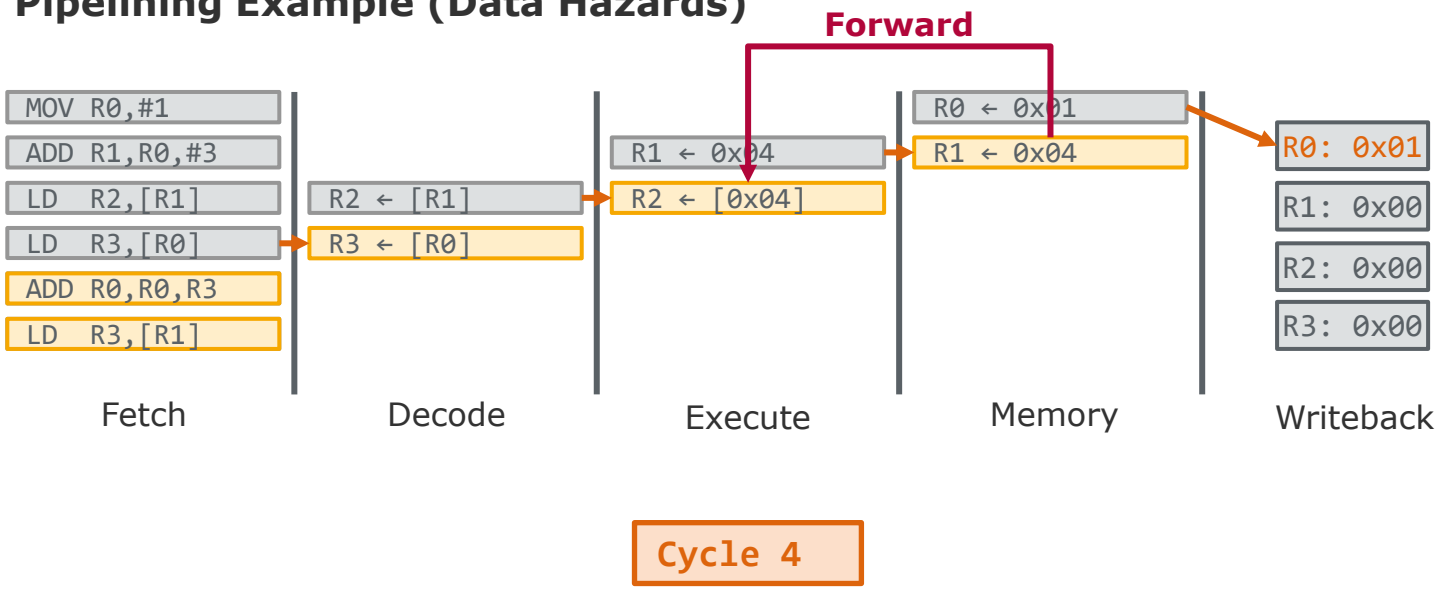


Pipelining Example (Data Hazards)

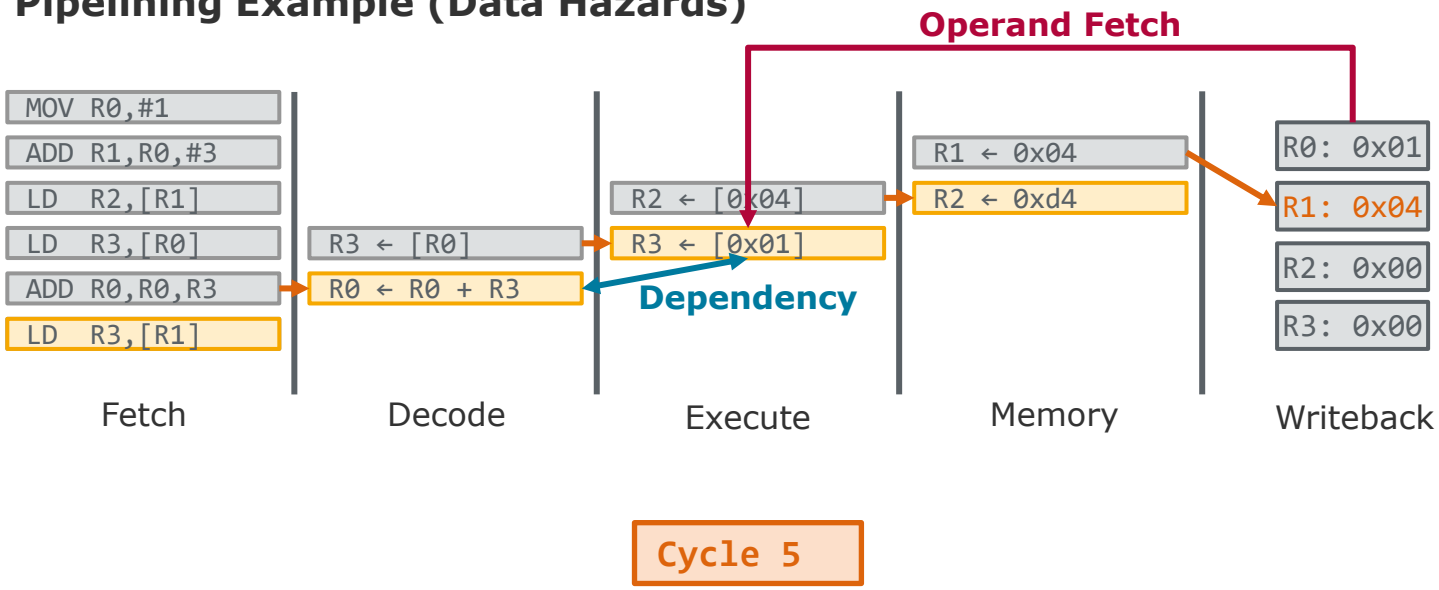


HPI Hasso Plattner Institut

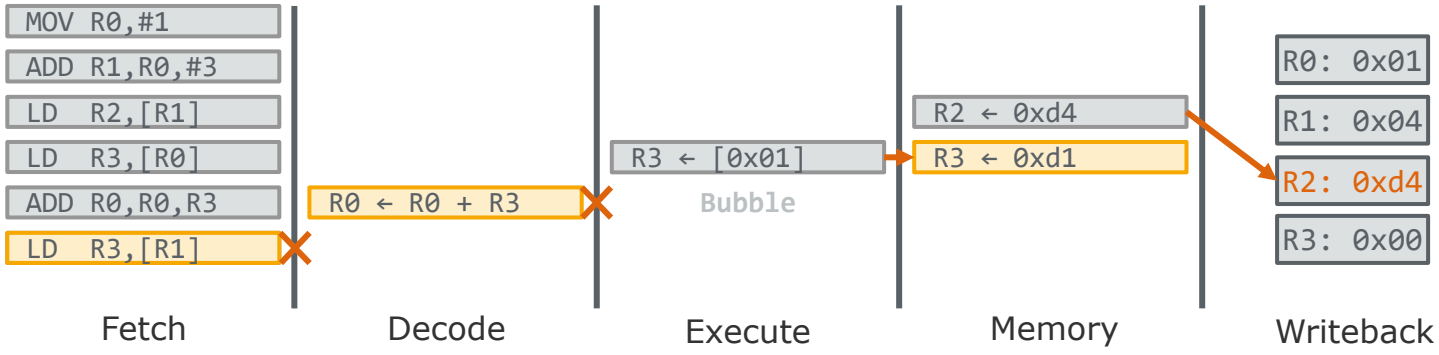
Pipelining Example (Data Hazards)



Pipelining Example (Data Hazards)



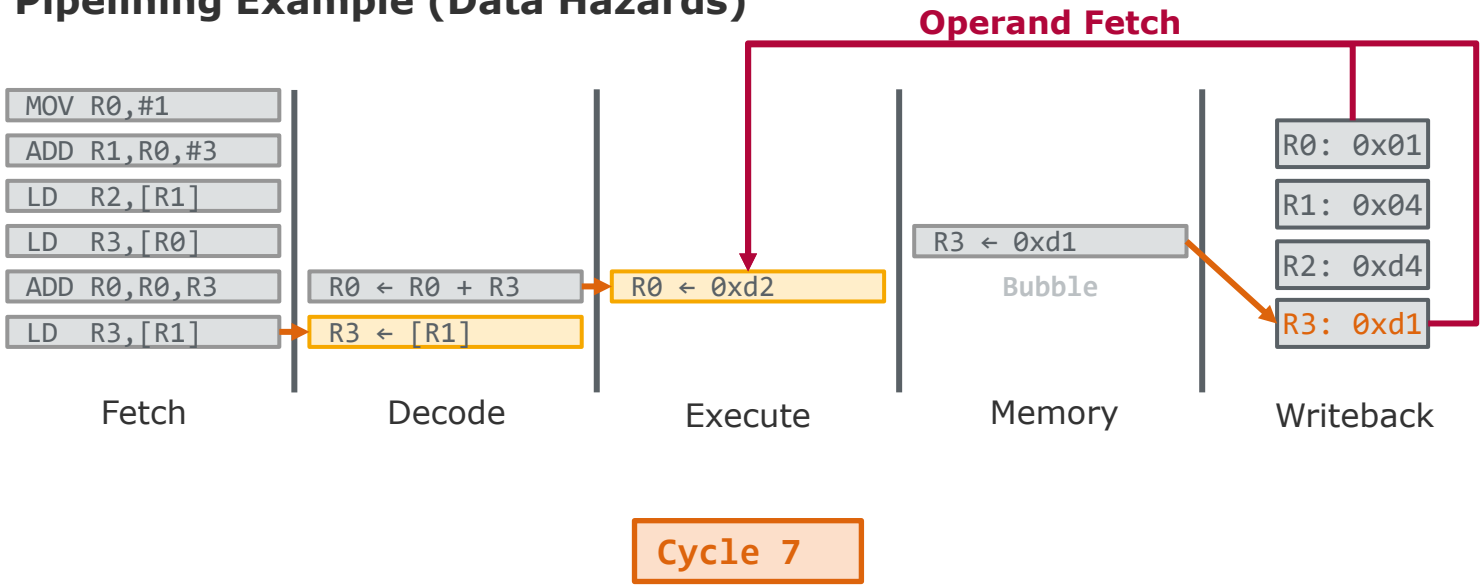
Pipelining Example (Data Hazards)



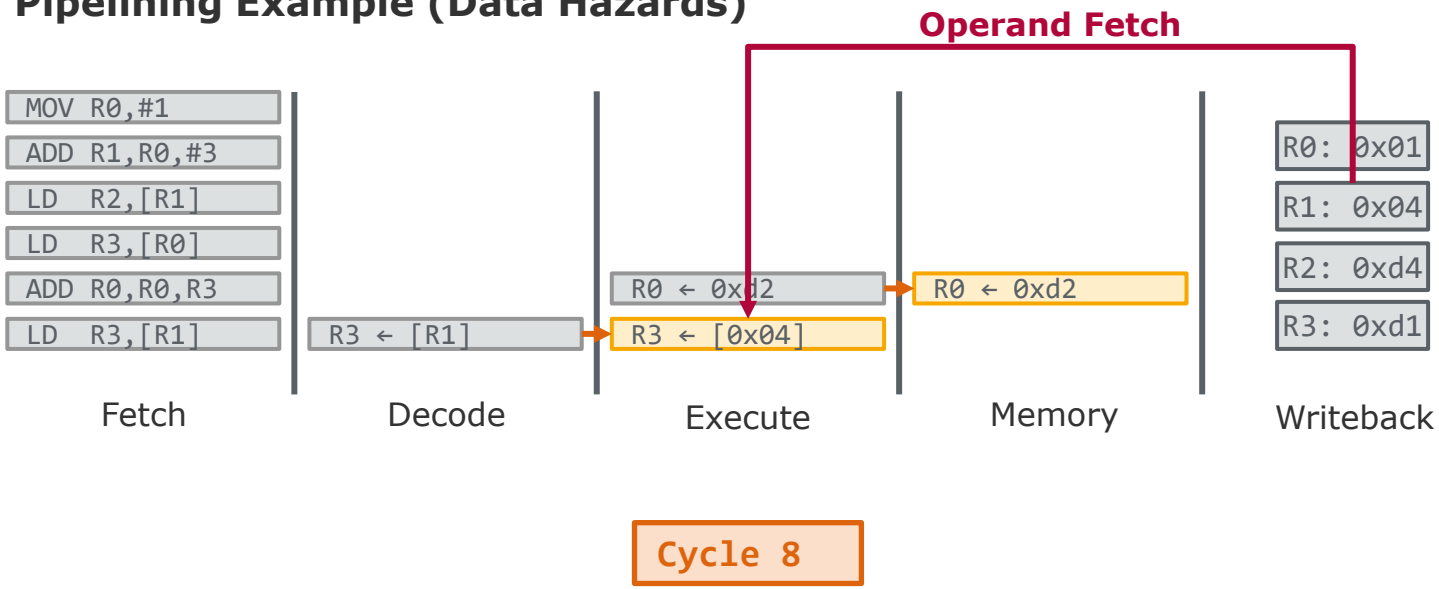
Cycle 6

HPI Hasso Plattner Institut

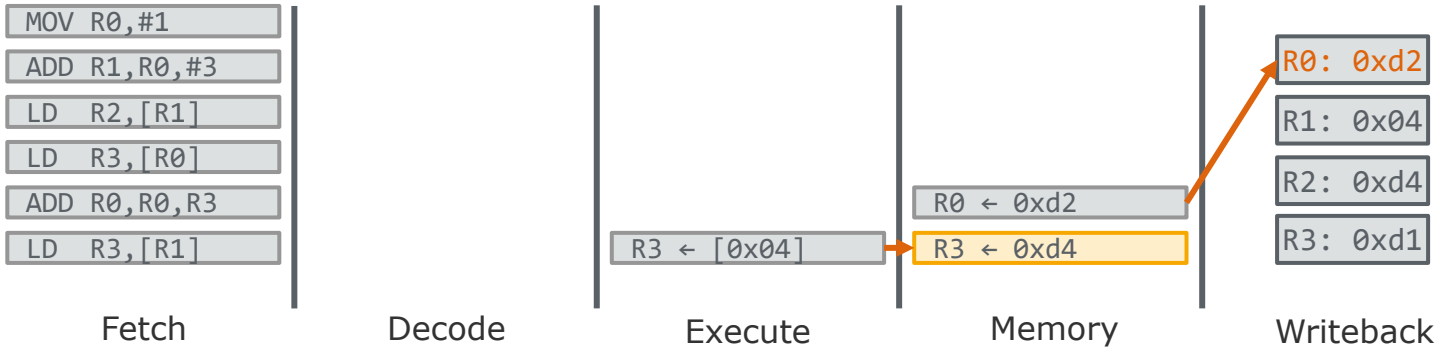
Pipelining Example (Data Hazards)



Pipelining Example (Data Hazards)

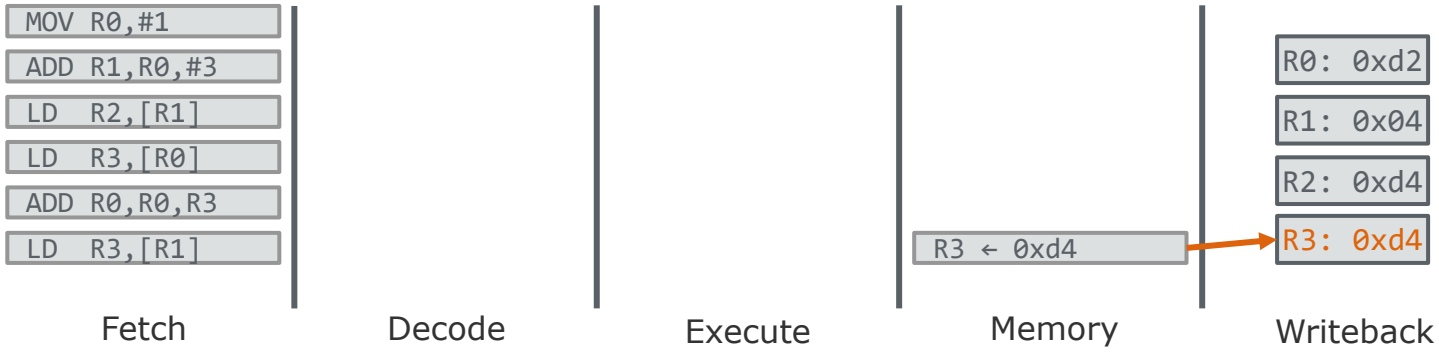


Pipelining Example (Data Hazards)



Cycle 9

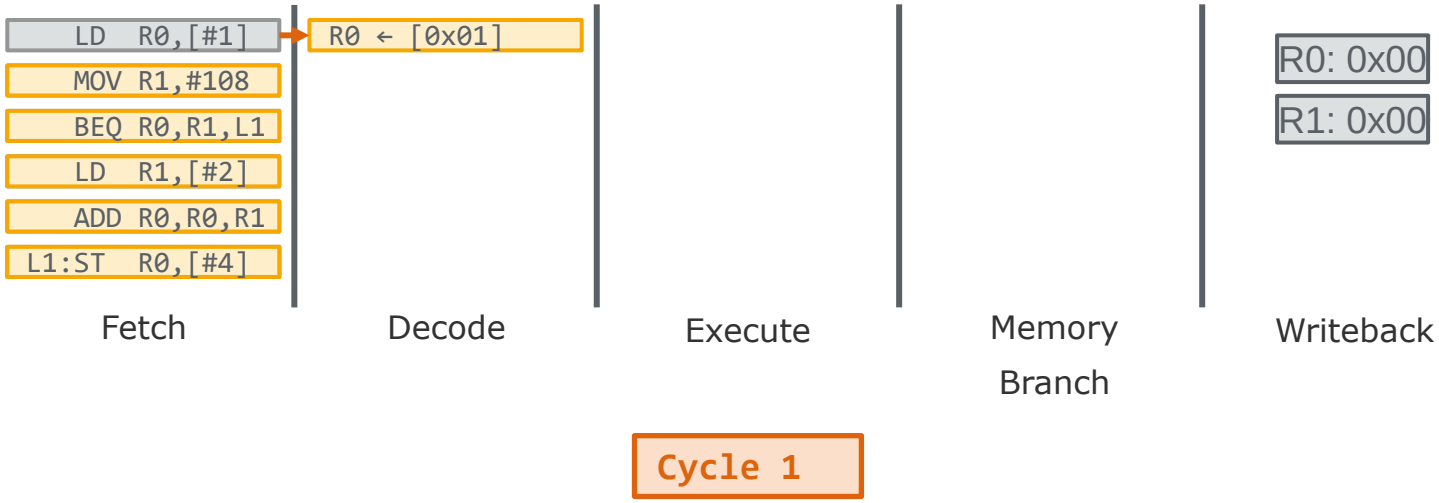
Pipelining Example (Data Hazards)



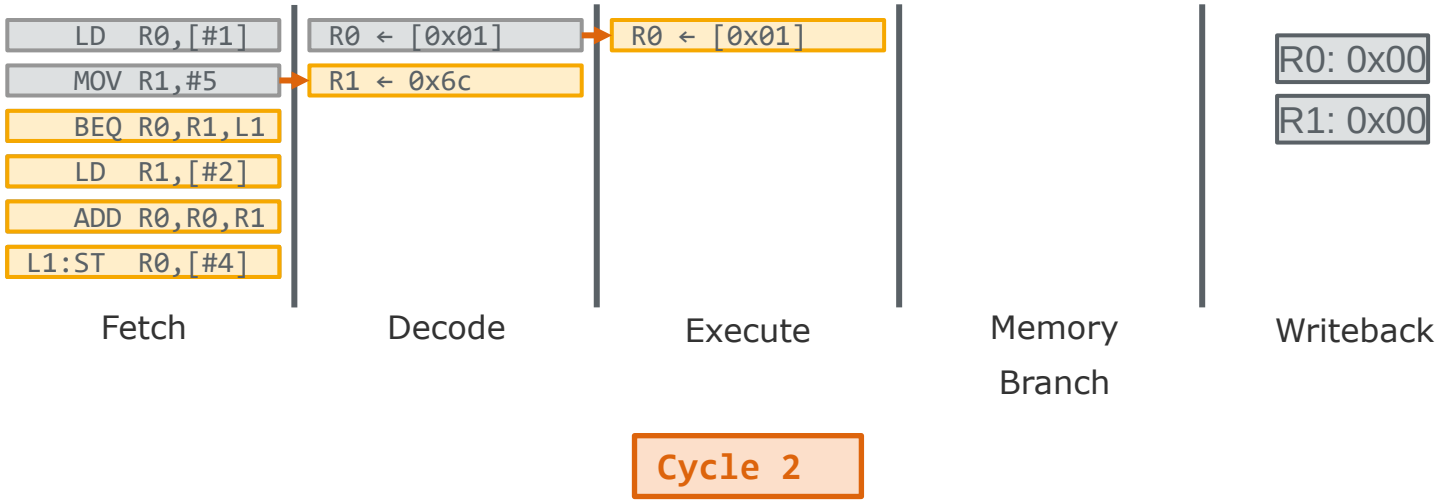
Cycle 10

HPI Hasso Plattner Institut

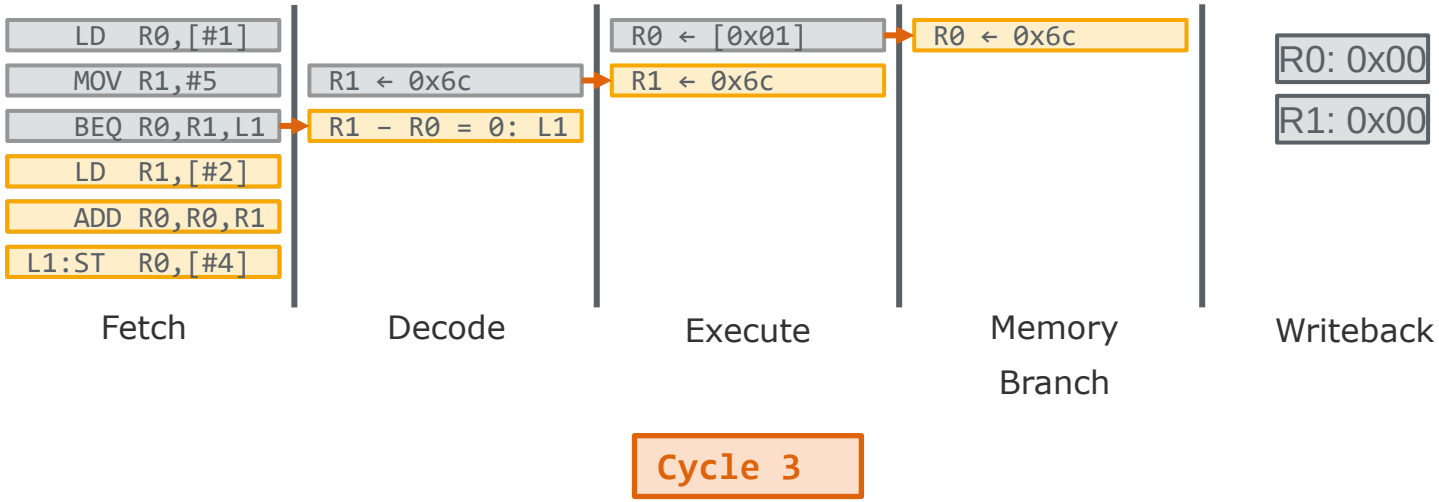
Pipelining Example (Control Hazard)



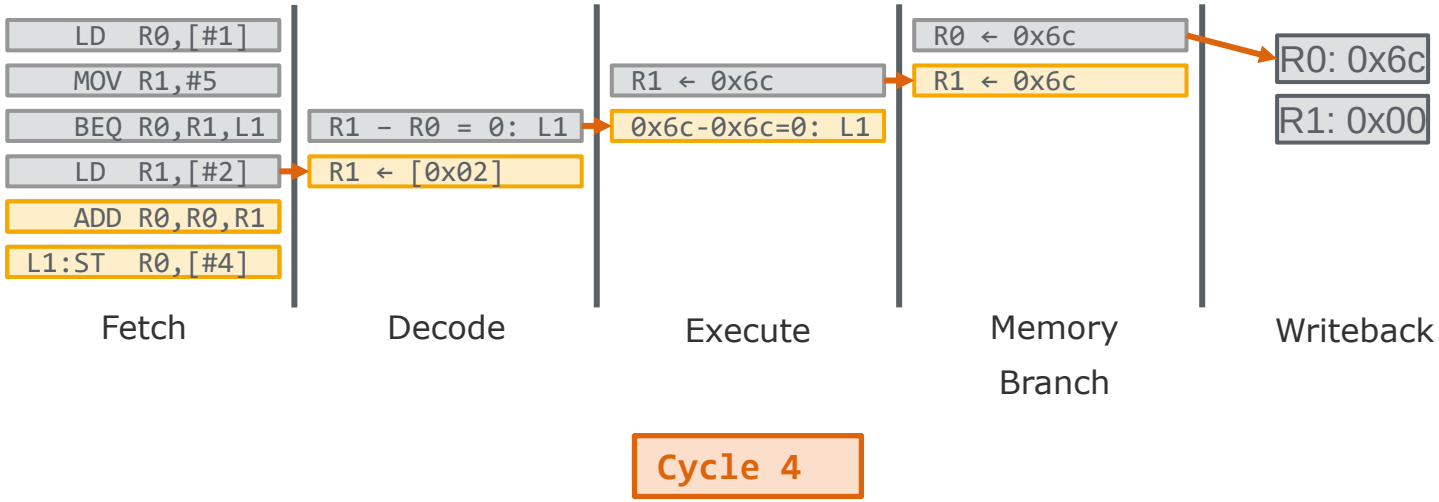
Pipelining Example (Control Hazard)



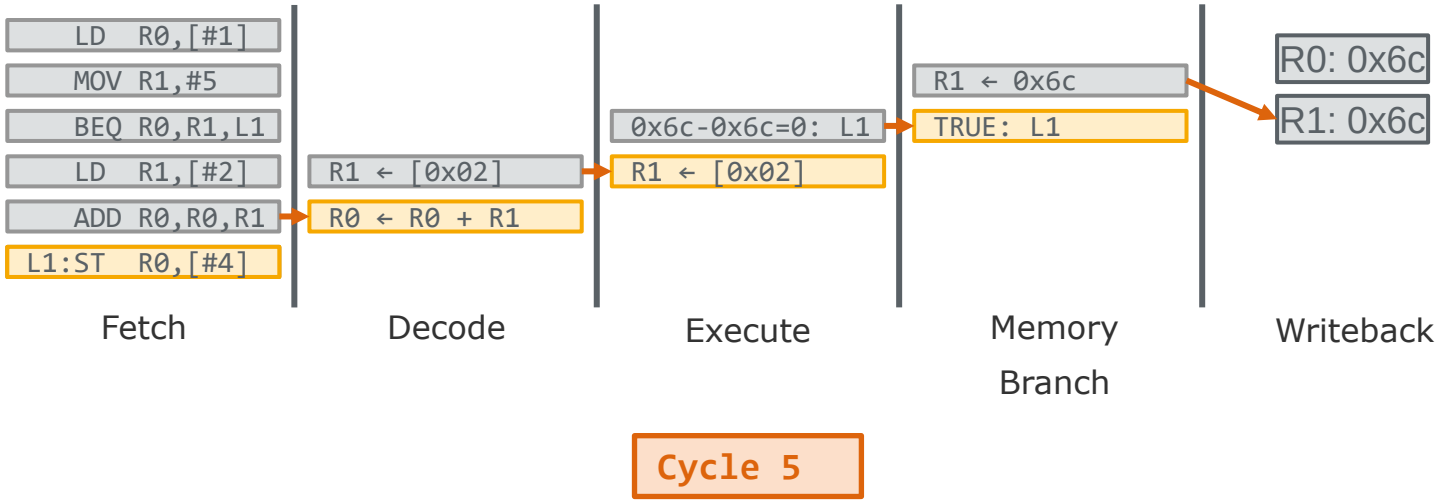
Pipelining Example (Control Hazard)



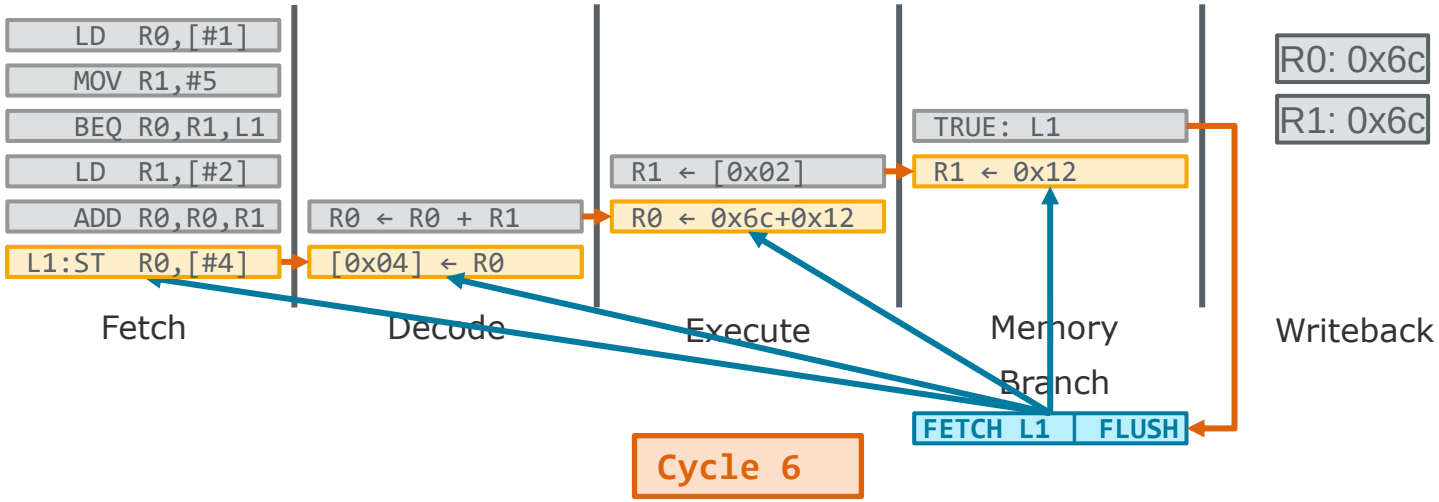
Pipelining Example (Control Hazard)



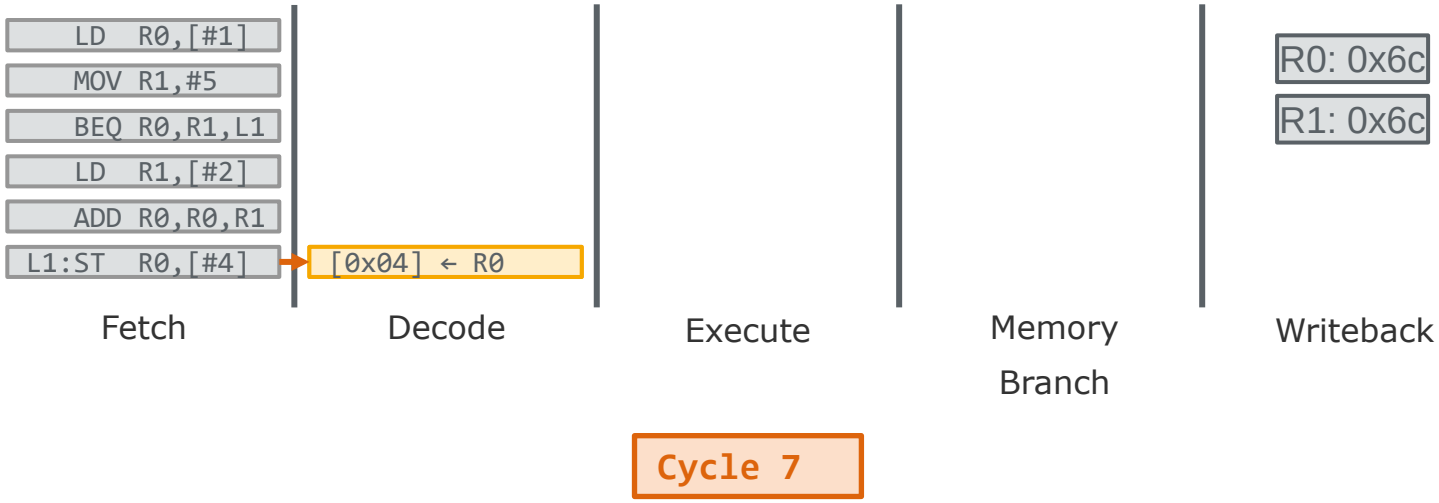
Pipelining Example (Control Hazard)



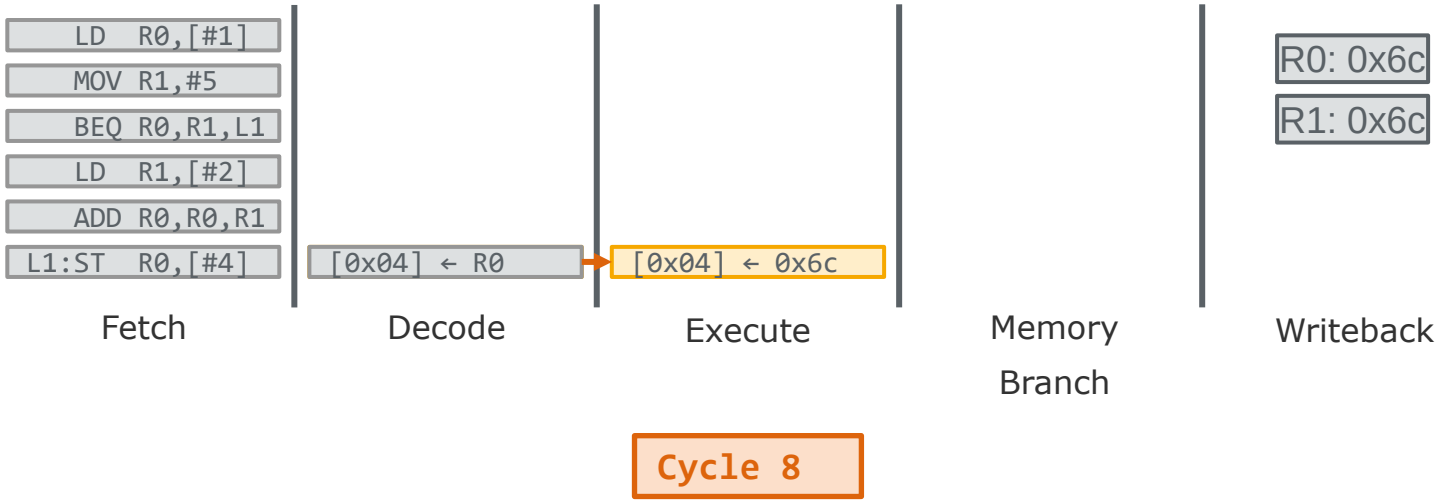
Pipelining Example (Control Hazard)



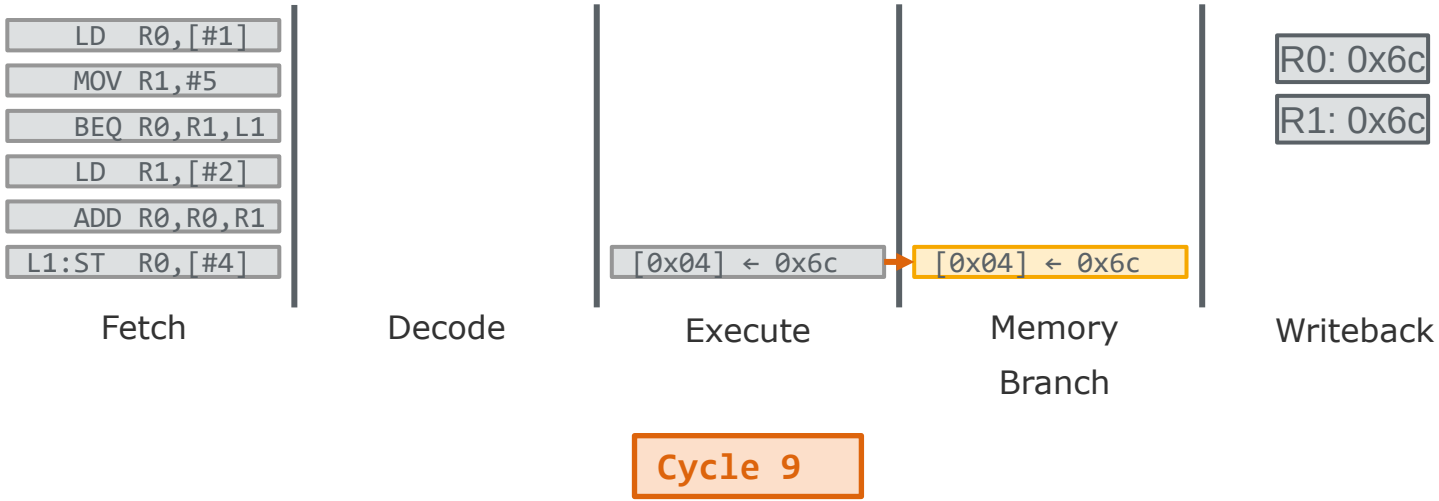
Pipelining Example (Control Hazard)



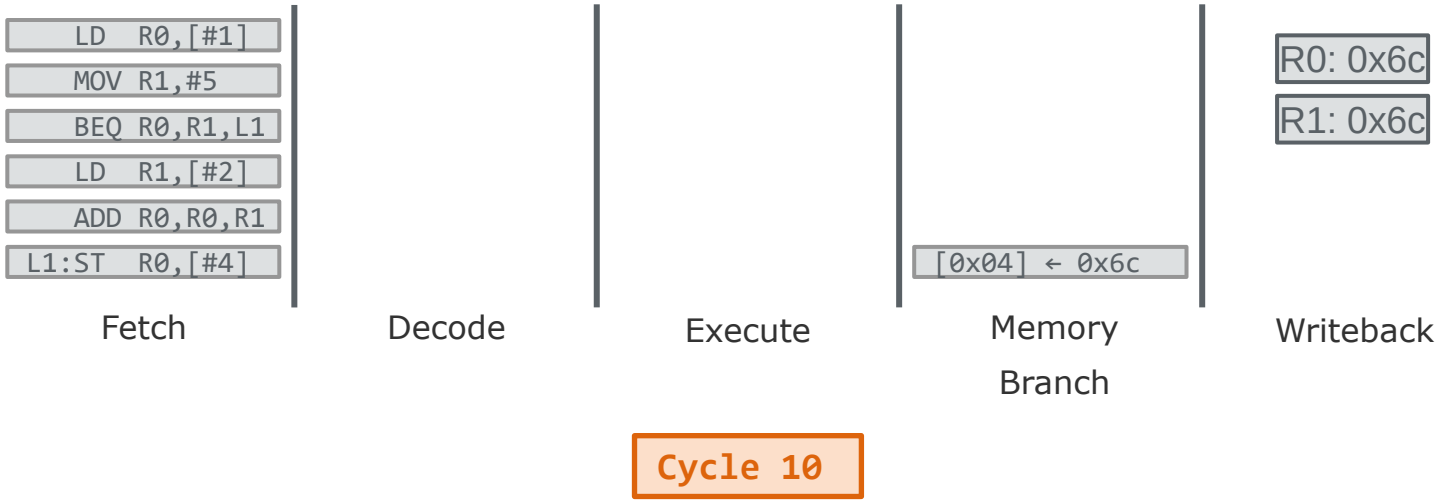
Pipelining Example (Control Hazard)



Pipelining Example (Control Hazard)



Pipelining Example (Control Hazard)



Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Hasso Plattner Institut

Pipelining Problems

- Data Hazard: Instruction requires operand that is not yet written back, Solutions:
  - Forwarding: Shortcut writeback path and transfer operands directly from intermediate stage
  - Generate Bubbles : Insert NOPs and halt preceeding stages until operand is available
- Control Hazard: Conditional branches may divert instruction stream, subsequent Fetches depend on branch completion, Solutions:
  - Generate Bubbles: Fetch stage halts after issuing a branch inserting NOPs, continues after branch target is computed
  - Branch Prediction: Fetch stage predicts branch target and continues issuing instructions, on misprediction all intermediate instructions are flushed

Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Superpipelining

- Finer subdivision of stages (~ 20-30) decreases combinatorial path depth to achieve higher clock frequencies
  - Approach taken with Intel NetBurst microarchitecture, introduced 2000 and abandoned in 2008 due to Power Wall
  - Control Hazards can degrade performance towards that of a coarser pipeline (relies on accurate Branch Predictor)

Shared-Memory Hardware
Exploiting Instruction Level Parallelism

HPI Hasso Plattner Institut

Superscalar Architecture

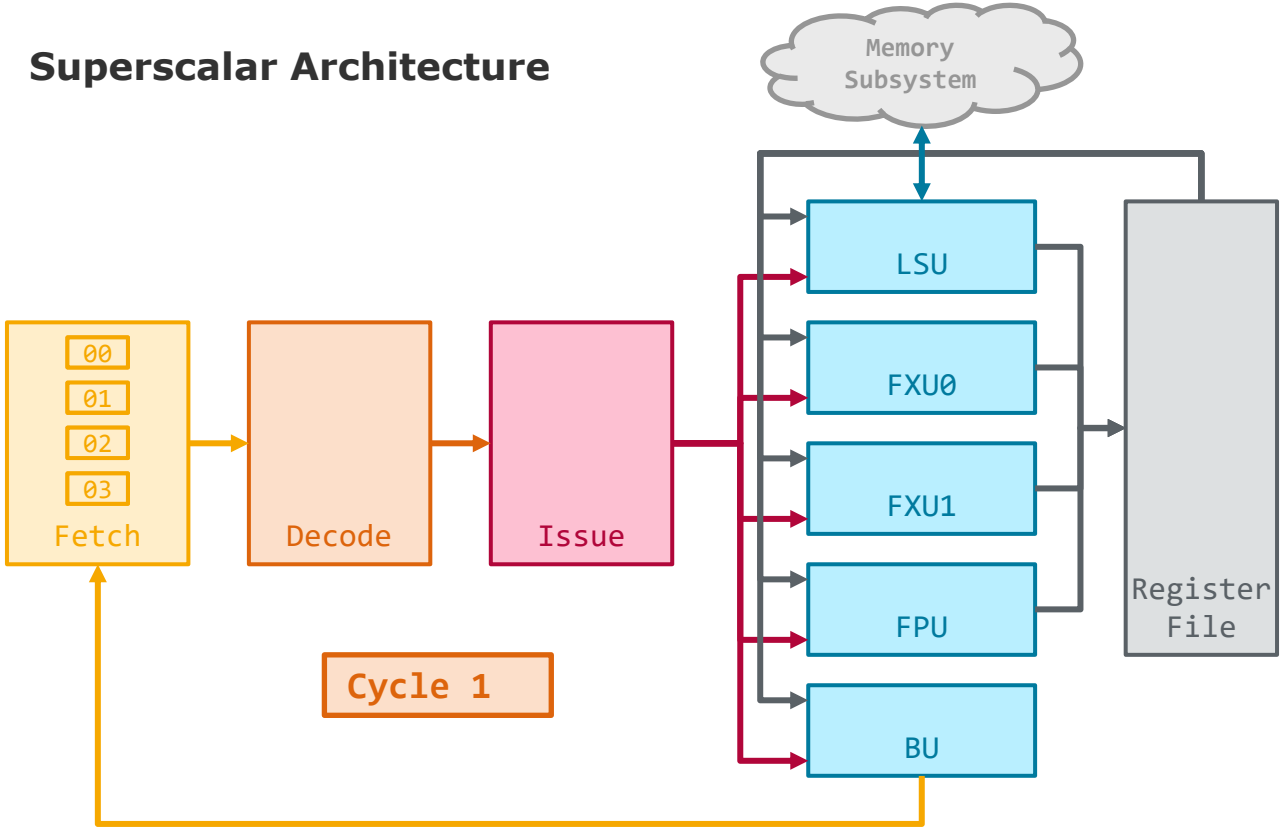
- Scalar pipelines can not exceed 1 IPC (instruction per cycle)
- ➢ Duplicate execution units to handle more than one independent instruction per cycle

Instructions are issued if no previous instruction is blocked and dependencies are met (operands and execution unit available)

- Frontend: Instruction Fetch and Decode
  - Can be duplicated to supply multiple decoded instructions per cycle, as all instructions are independent at that stage
- Issue Queue: Schedules decoded instructions for execution
- Backend: Registers and various execution units (EU)
  - Fixed-Point units (FXU), Load-Store units (LSU), Floating-Point Units (FPU), Branch Units (BU)

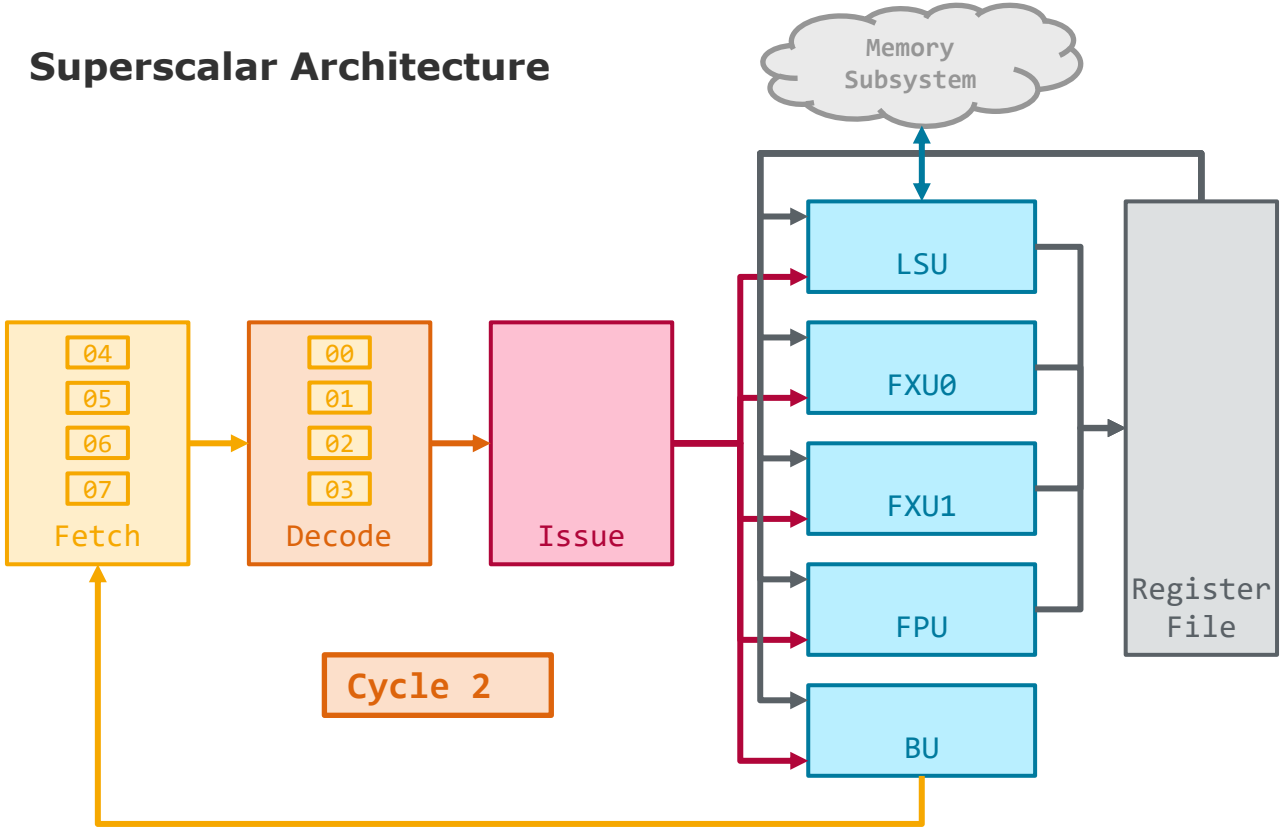
Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Superscalar Architecture



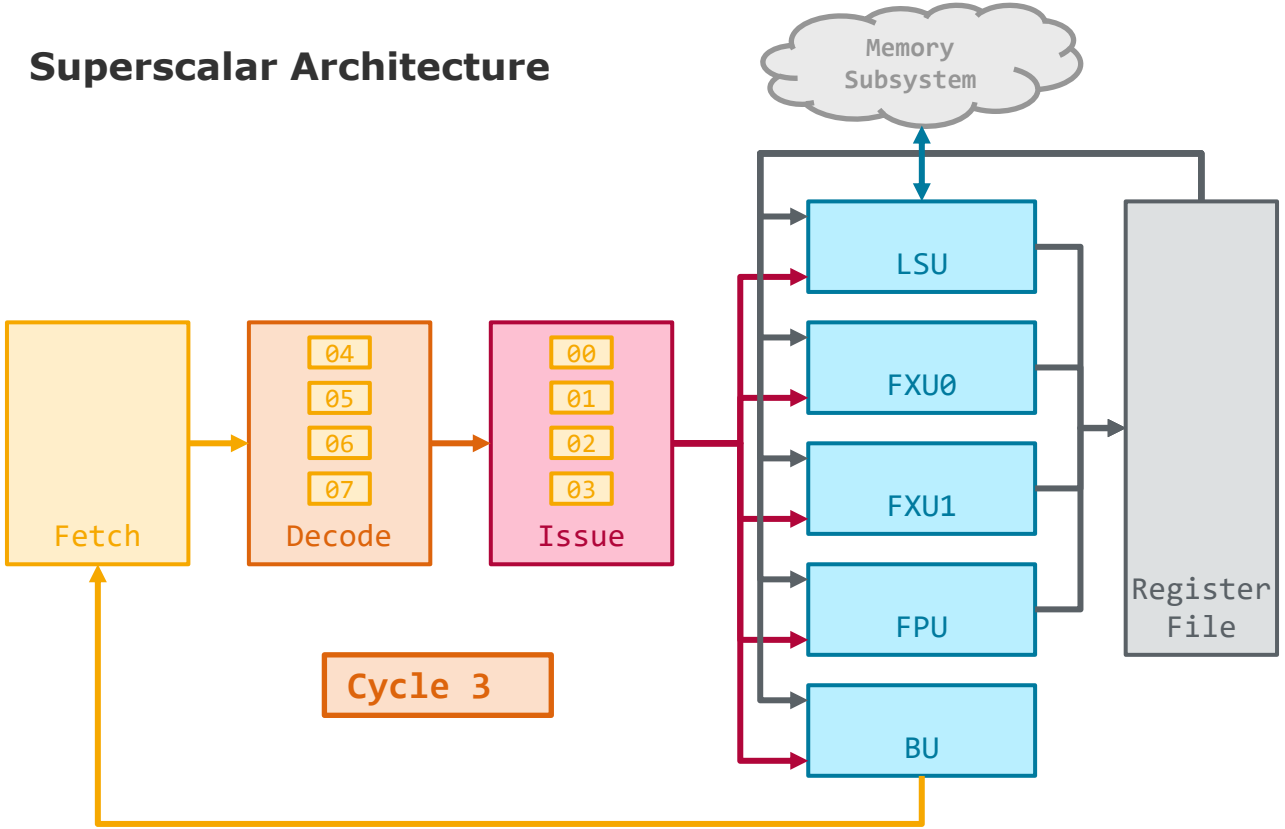
Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Superscalar Architecture



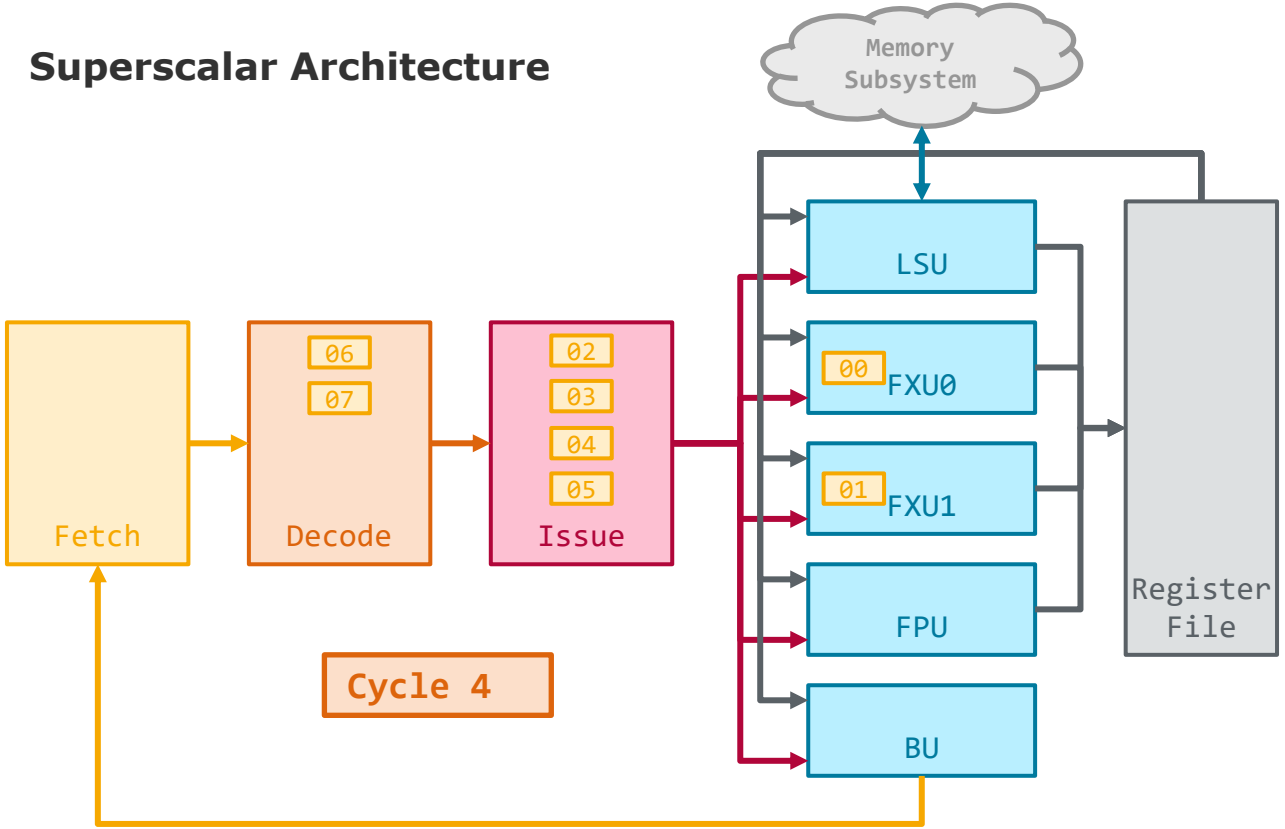
Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Superscalar Architecture



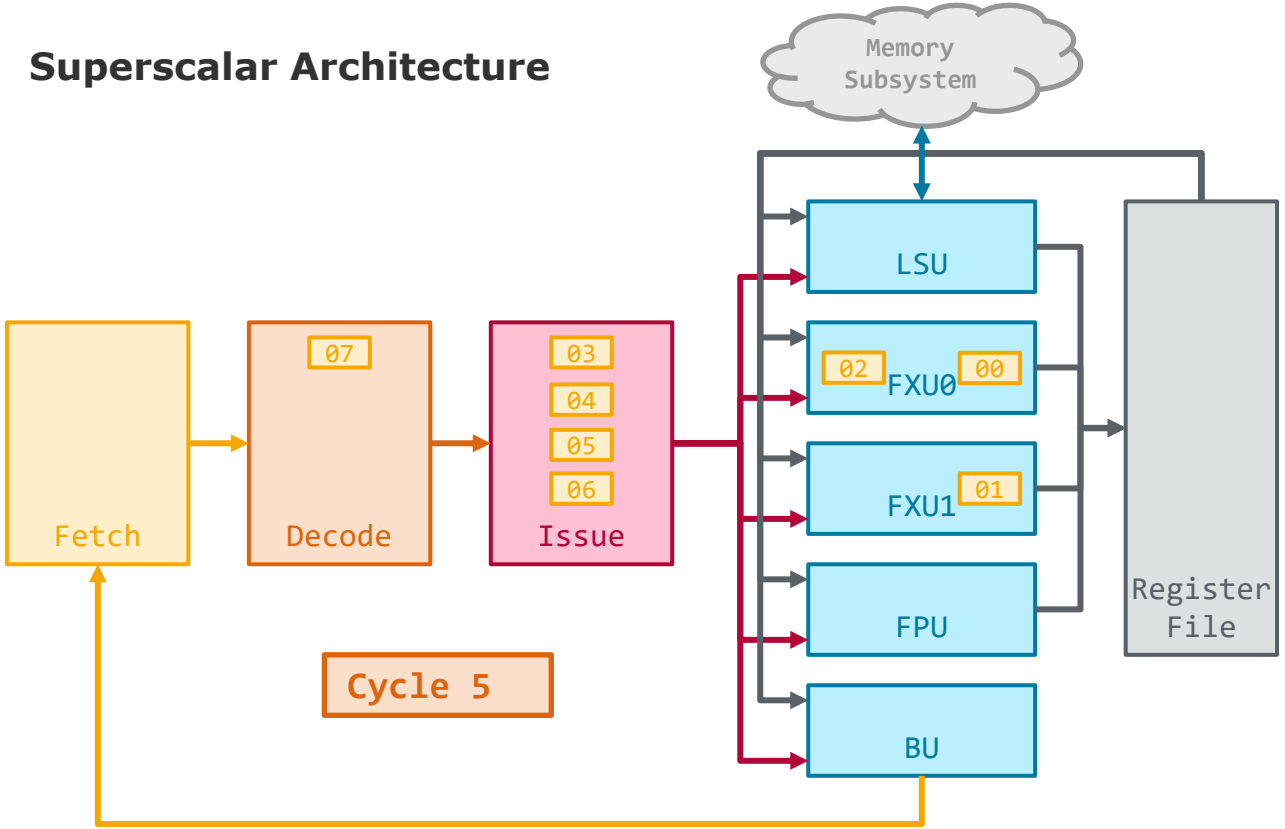
Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Superscalar Architecture



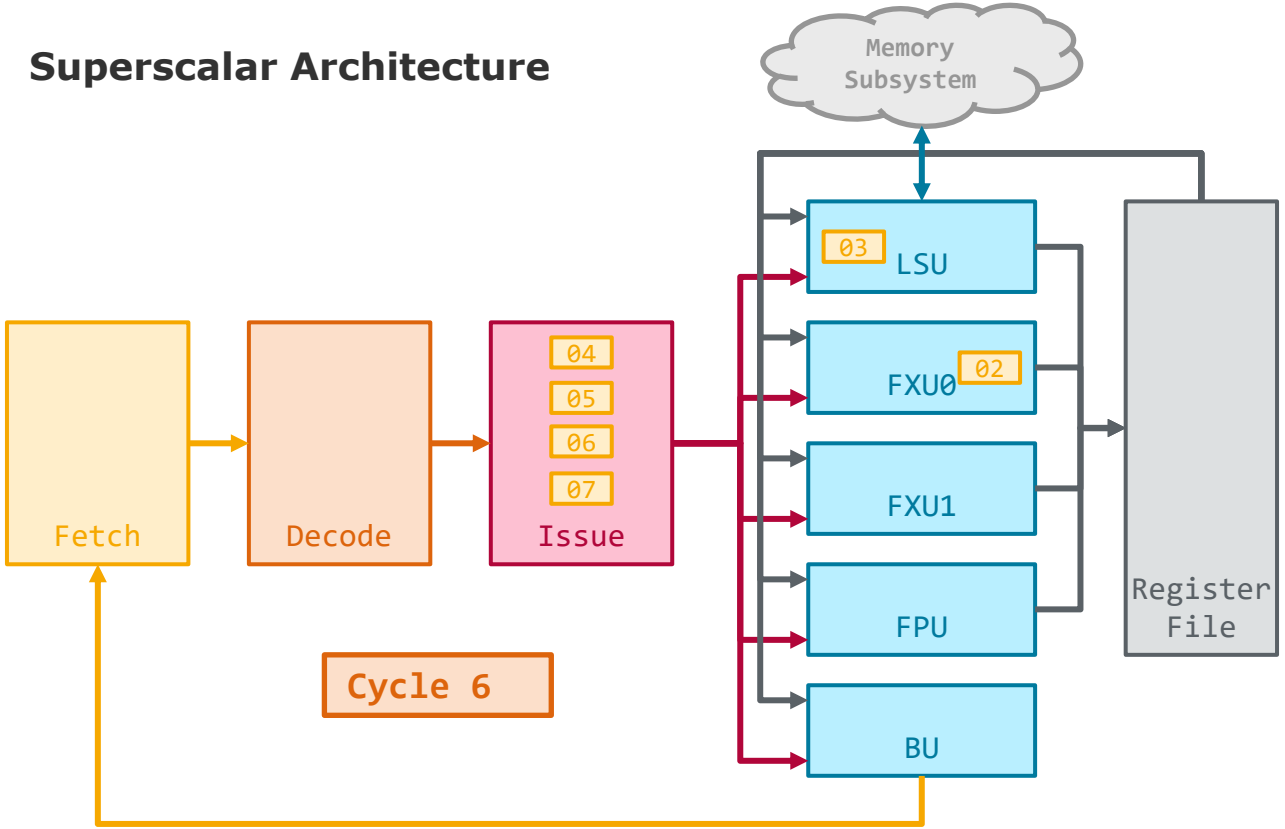
Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Superscalar Architecture



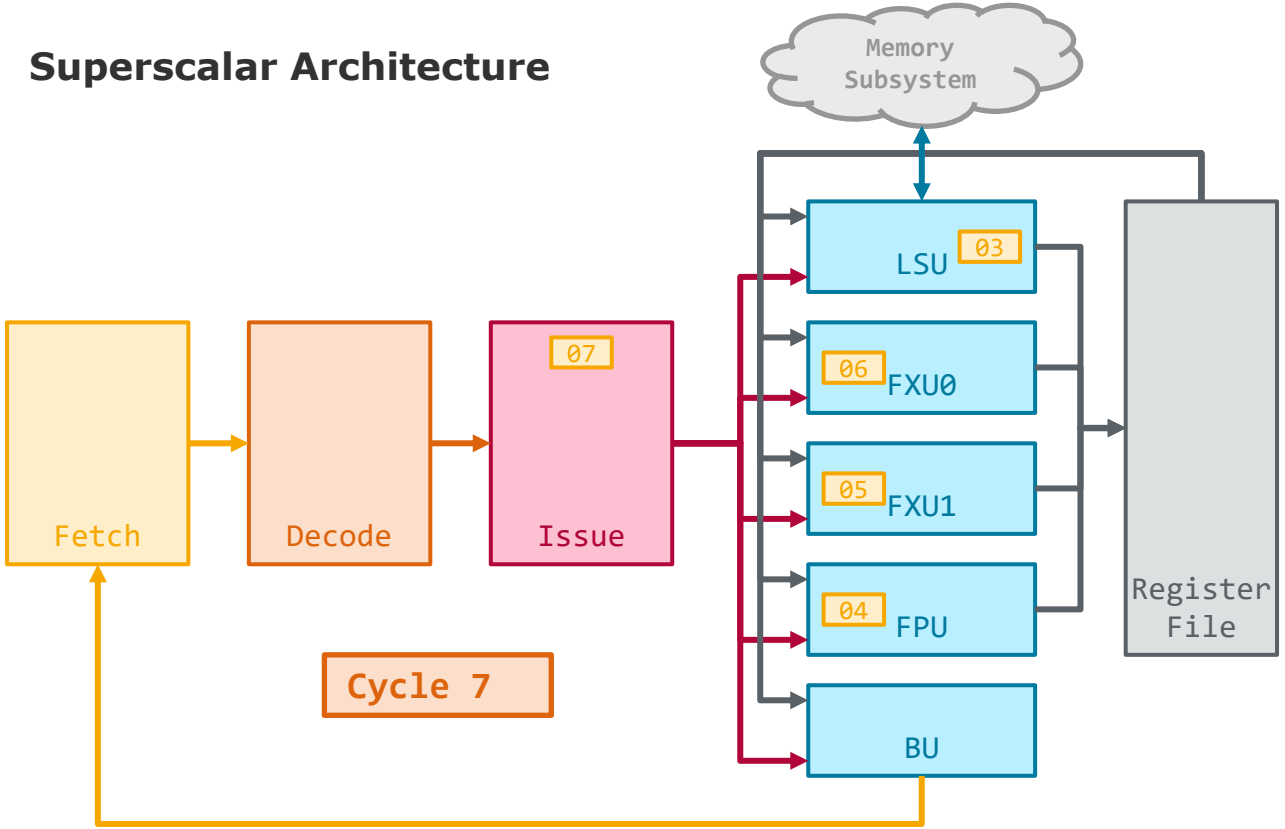
Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Superscalar Architecture



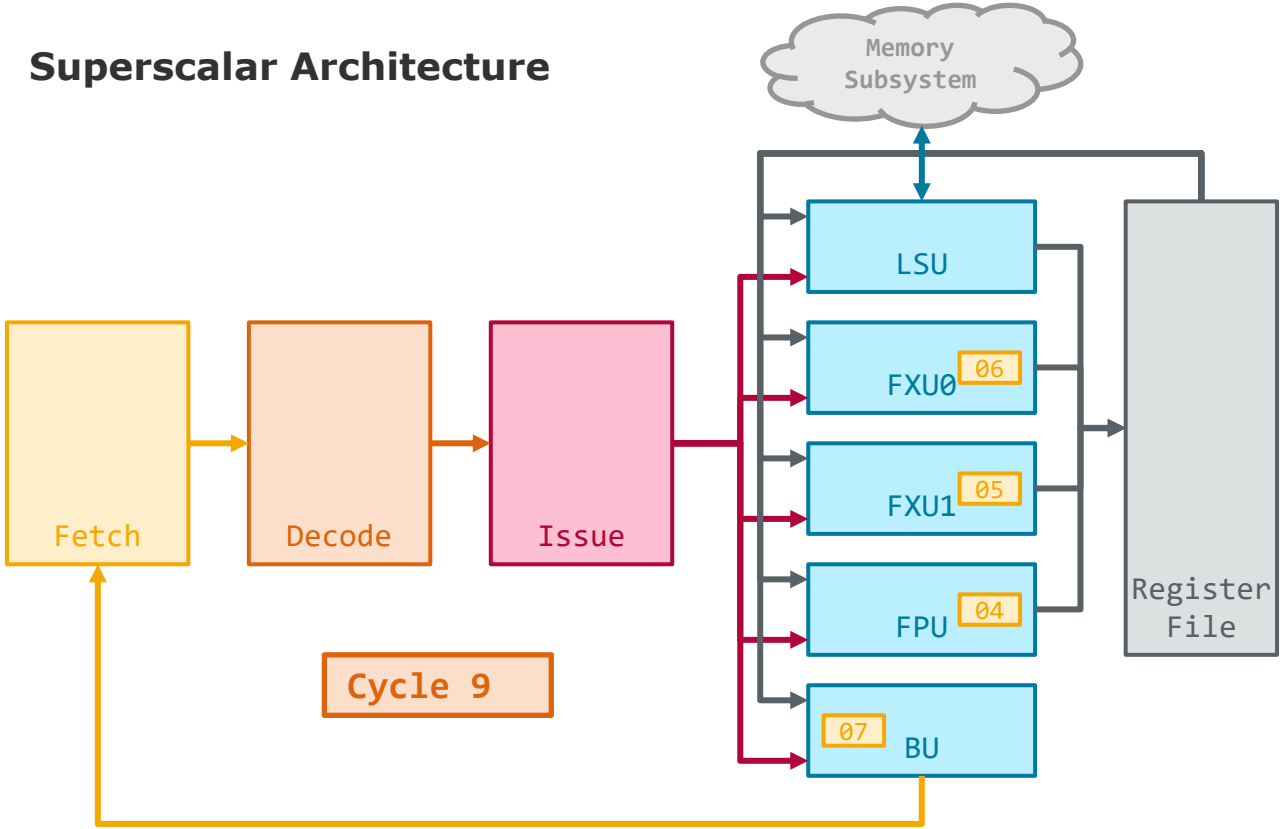
Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Superscalar Architecture



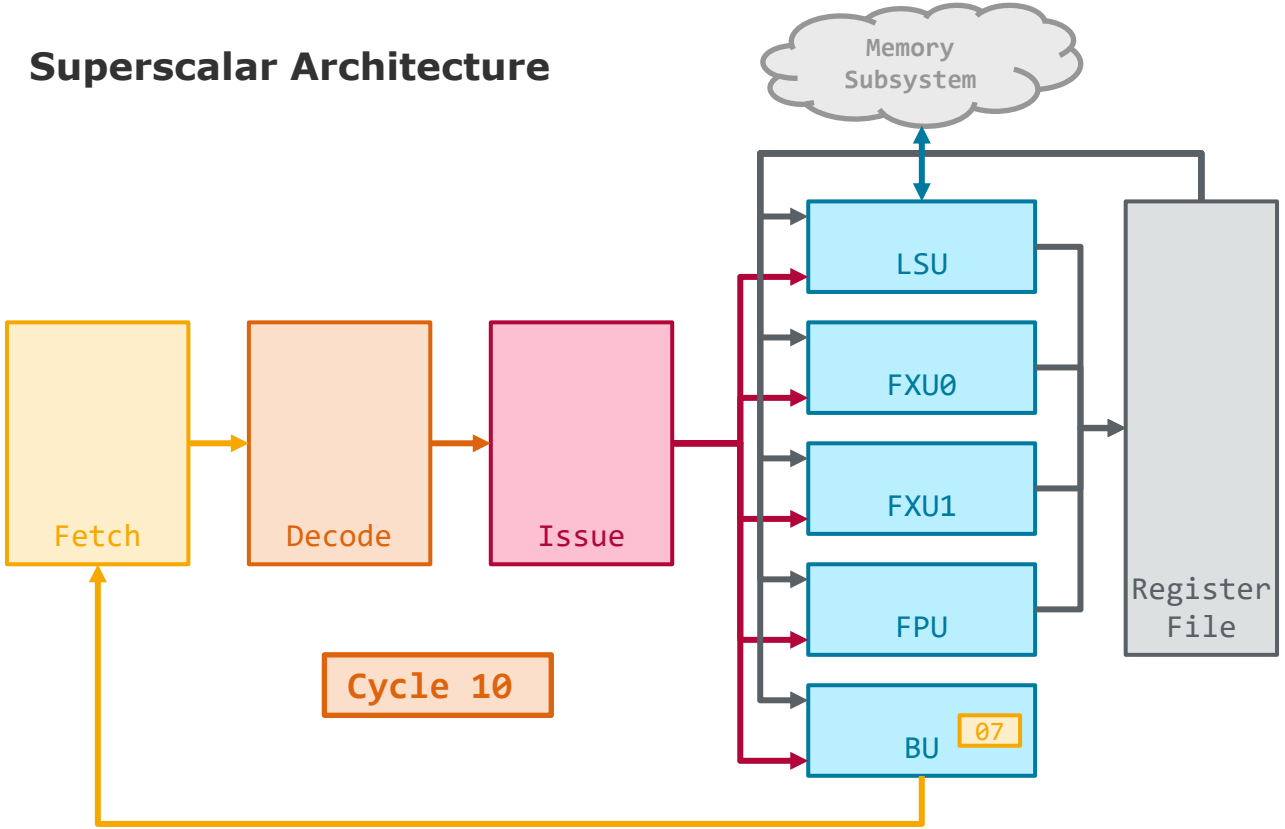
Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Superscalar Architecture



Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Superscalar Architecture



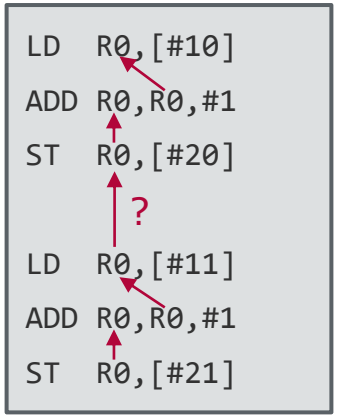
Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Superscalar Architecture

- Frontend and execution units in backend are usually pipelined

- In-Order Execution: Issue queue must keep order of instruction stream:
  - Even independent subsequent instructions can not complete before a delayed previous instruction (e.g. Load with cache miss)
  - Entire Backend is stalled for a single stalled execution unit

- False Dependencies: Write-after-Read conflicts on registers
  - Instruction might wait for destination register to become free, even though both operands and execution unit are available already



Superscalar Optimization: Out of Order

- Execution must *appear* sequential,

  Instructions can be executed out of program order if:

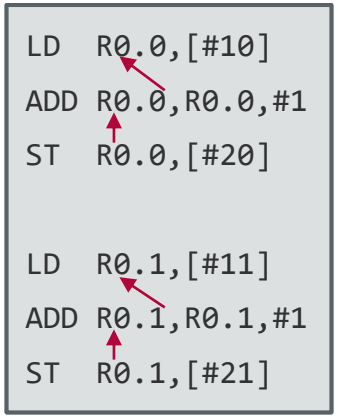
  - Dependency tracking in issue queue ensures that instruction is only executed once operands are available
  - Architecturally visible effects of instruction are held back until previous instructions have applied their effects (commit)

- Add Reorder Buffer (ROB) to track computed but not yet committed results

HPI Hasso Plattner Institut

Superscalar Optimization: Register Renaming

- Architectural Registers are likely to be reused within the execution window, creating false dependencies
  - Add more physical registers to accommodate conflicting usage
- ➢ Issue Queue maps architectural register numbers to a pool of physical registers



Superscalar Optimization: Speculative Execution

- Analogous to scalar pipelines:
  - Branch instructions could act as barriers to Issue Queue
  - Efficient alternative: Branch Prediction continues instruction stream speculatively, on misprediction speculative instructions are nullified
- Out of order architecture can accommodate speculative execution:
  - Speculative instructions appear in Reorder Buffer after the branch instruction they depend on
  - Once branch commits, dependent instructions can be identified and if necessary discarded from Reorder Buffer
- Current Branch Predictors can achieve >95% accuracy!
- Problem: Not all instruction effects are nullified
  - Non-architectural state (e.g. in caches) sometimes can not be and usually is not rolled back, opening the way to side-channel attacks

Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Even though programmers think in terms of sequential instruction streams, awareness of instruction level parallelism opens optimization potential.

Shared-Memory Hardware
Exploiting Instruction Level Parallelism

Very-Long-Instruction-Word (VLIW) /
Explicitly-Parallel-Instruction-Computer (EPIC)

- Alternative to dynamic instruction scheduling in superscalar architectures
  - Requires programmer or compiler to explicitly designate parallelizable instructions (static schedule)
  - Greatly simplifies hardware implementation
  - Burden on Compilers to statically determine instruction dependencies and optimal execution schedules
- Static analysis can not capture runtime effects like cache behaviour
  - *One* static execution schedule may not be optimal in *all* runtime situations
- Prominent example IA-64 architecture from 2001, not widely adopted
- Also some embedded architectures, DSPs, older GPUs

Shared-Memory Hardware
Instruction Level Parallelism

Hasso Plattner Institut

Limits of Instruction Level Parallelism

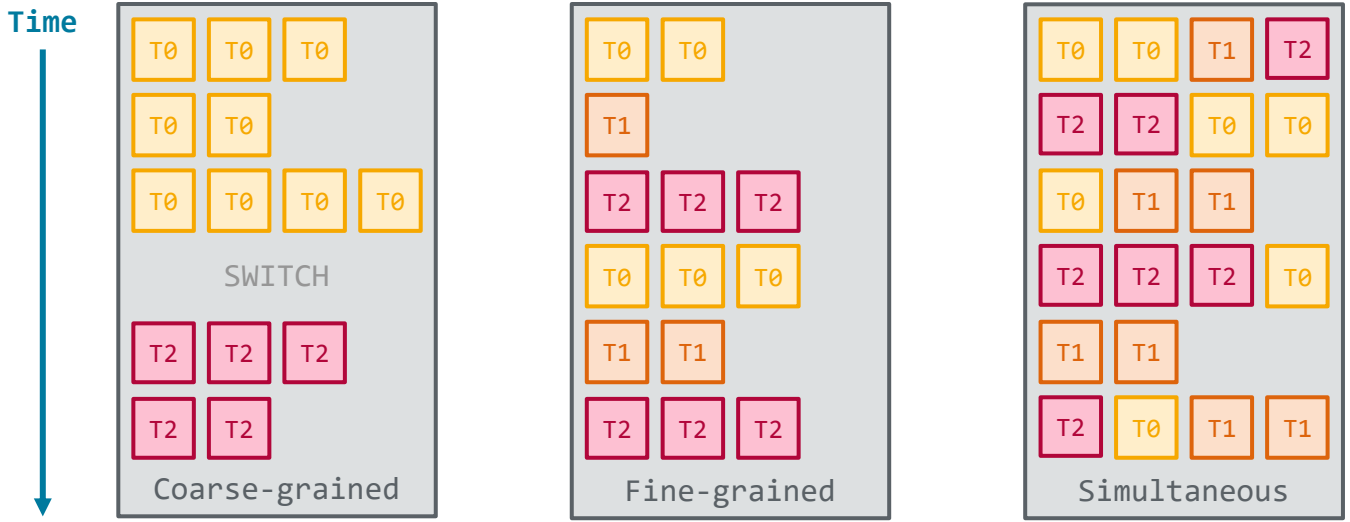
- ILP in a single instruction stream is limited due to dependencies
- Larger execution windows quickly become infeasible due to prohibitively complex dependency checking logic between executing instructions
- ➢ ILP exploitation techniques have reached stage of diminishing returns for general workloads

Executing multiple independent instruction streams offers new potential for parallelization!

Shared-Memory Hardware
Thread Level Parallelism

Single-Core Multithreading

- Threads are the smallest units of parallelism under programmers' explicit control
- There are different execution schemes for multiple threads on a single core:



Shared-Memory Hardware
Thread Level Parallelism

Simultaneous Multithreading (SMT)

- Superscalar Out of Order cores already have much of the logic required for SMT (i.e. dependency tracking, register renaming)
- SMT Support: Duplicate architectural state per hardware thread and tag instructions with thread number

- Issue Queue gains additional dependency domains
- Higher utilization of execution units

Shared-Memory Hardware
Thread Level Parallelism

SMT never increases but might decrease singlethread performance, if execution units are congested by other threads.

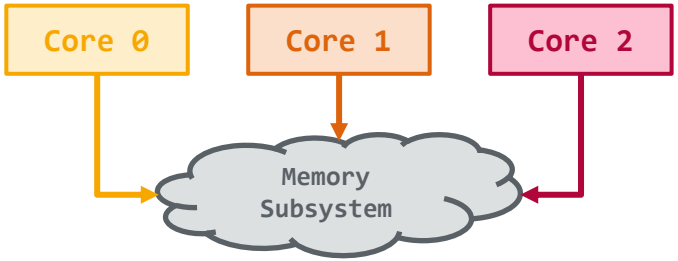
SMT never decreases but can increase core utilization and thus overall throughput.

Shared-Memory Hardware
Thread Level Parallelism

Multicore Machines

- Workloads with high degree of TLP can cause contention of the execution units available in a single core
- ➢ Distribute workload on multiple cores!

- Cores are self contained (do not share execution units or frontend logic)
- Cores share access to a memory subsystem



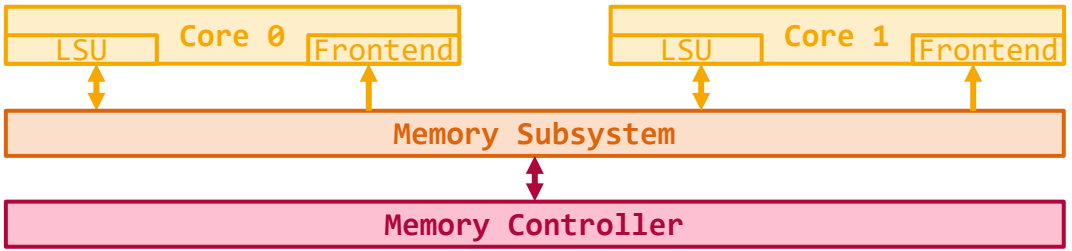
HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

And now for a break and a cup of Darjeeling.

*or beverage of your choice

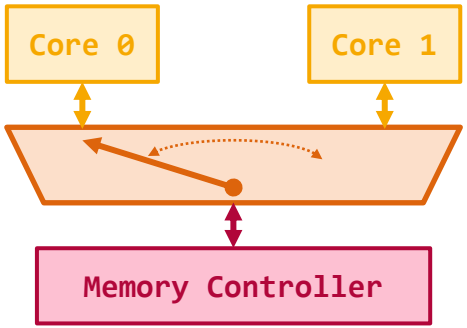
Shared-Memory Hardware
Memory Consistency Models

- Cores initiate two types of memory operations:
  - Instruction Fetches through the Frontend
  - Data Loads/Stores through the Load-Store Units
- Multiple Cores are serviced by a shared memory subsystem, which performs main memory accesses via a memory controller



Shared-Memory Hardware
Memory Consistency Models

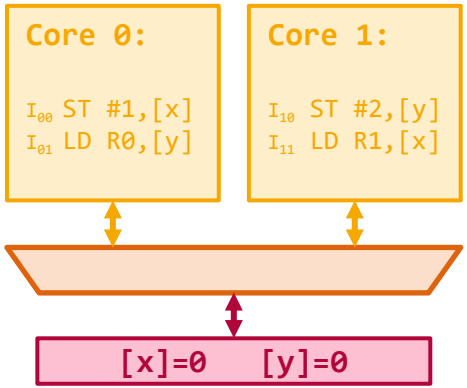
- 1st Simplification: Model memory subsystem as a multiplexer
  - One core at a time has exclusive access to memory controller
- 2nd Simplification: Disregard Instruction Fetches
  - Fetches are not explicitly initiated by instructions
  - ➢ Not covered by the consistency model



Shared-Memory Hardware
Memory Consistency Models

What happens, if multiple (in-order) cores concurrently access memory?

- Sequential Consistency
  - Multiplexer might switch at arbitrary times
  - The global instruction order $<_M$ arises from interleaving the local instruction orders $<_{C0}$ and $<_{C1}$
- Only guarantee: If two instructions are issued in a particular order by the core, they can not be reversed in the global memory order

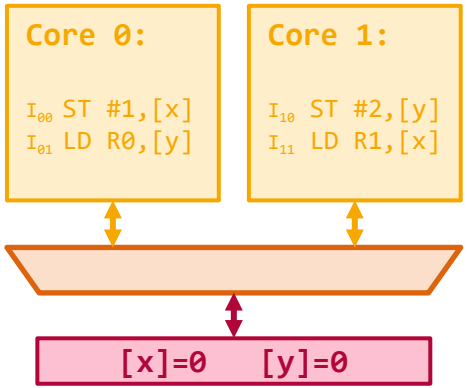


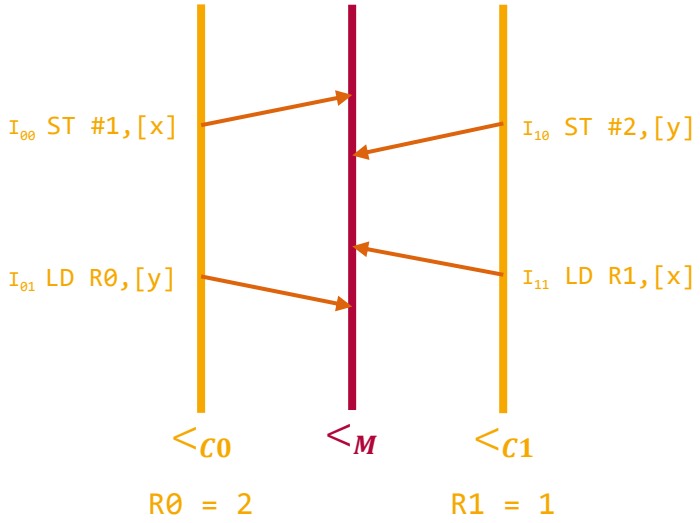
$$I_a <_{Cx} I_b \Rightarrow I_a <_M I_b$$

Shared-Memory Hardware
Memory Consistency Models

What happens, if multiple (in-order) cores concurrently access memory?

- Sequential Consistency

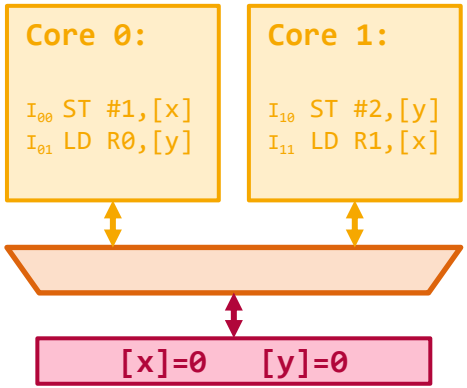


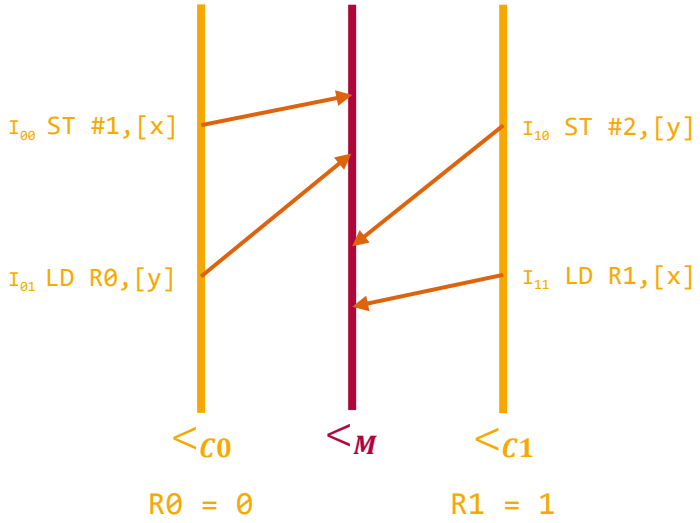


Shared-Memory Hardware
Memory Consistency Models

What happens, if multiple (in-order) cores concurrently access memory?

- Sequential Consistency

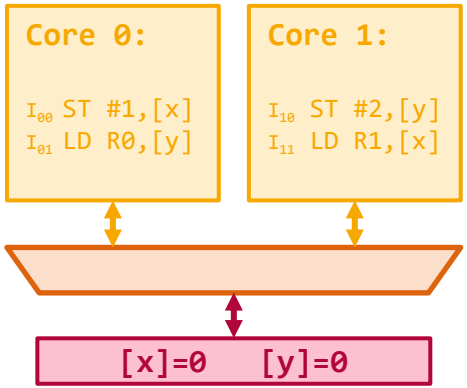


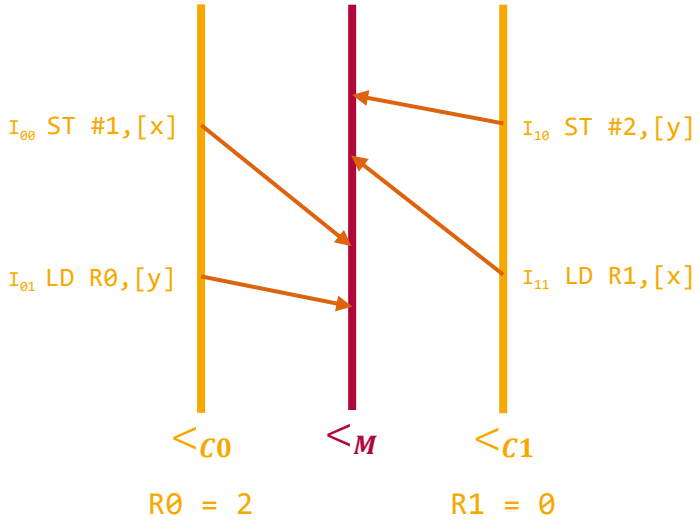


Shared-Memory Hardware
Memory Consistency Models

What happens, if multiple (in-order) cores concurrently access memory?

- Sequential Consistency

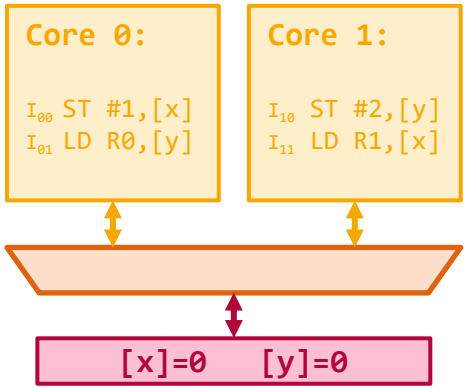


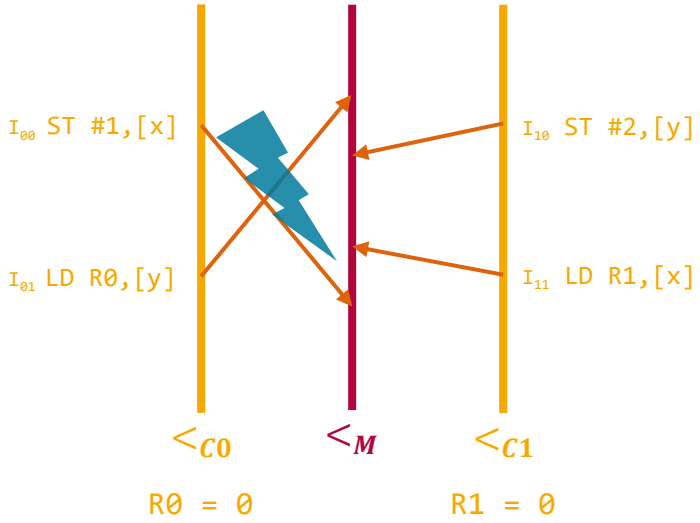


Shared-Memory Hardware
Memory Consistency Models

What happens, if multiple (in-order) cores concurrently access memory?

- Sequential Consistency



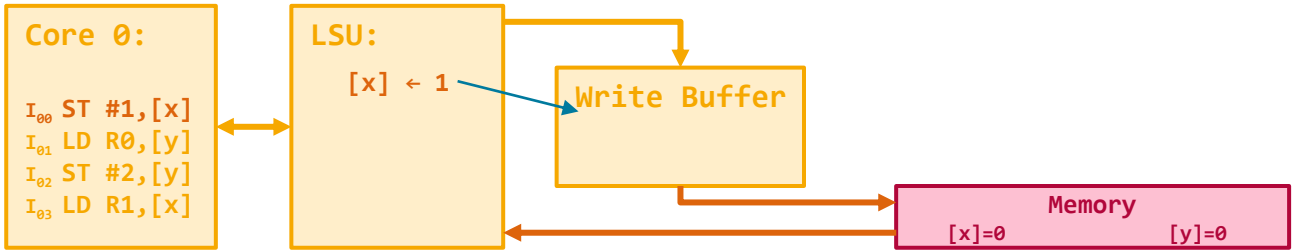


$I_{00} <_{C0} I_{01}$ contradicts $I_{01} <_{M} I_{00}$

Shared-Memory Hardware
Memory Consistency Models

Excursion: Write Buffers

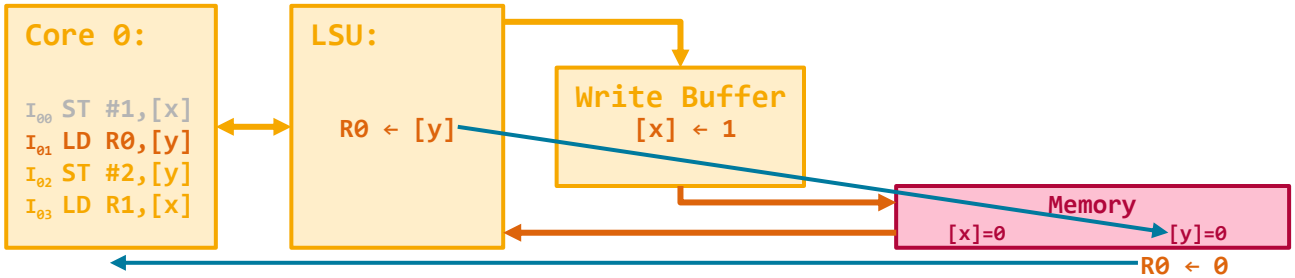
- **Load** instructions must wait for results from memory
- **Store** instructions produce no results for subsequent instructions
  - LSU does not need to wait for an issued **Store** instruction to complete
- This optimization is implemented using Write Buffers, i.e. FIFO memories storing address and data of pending **Store** operations
- To maintain in-order illusion, subsequent **Loads** to addresses present in Write Buffer must return most recent buffered data (Bypass)



Shared-Memory Hardware
Memory Consistency Models

Excursion: Write Buffers

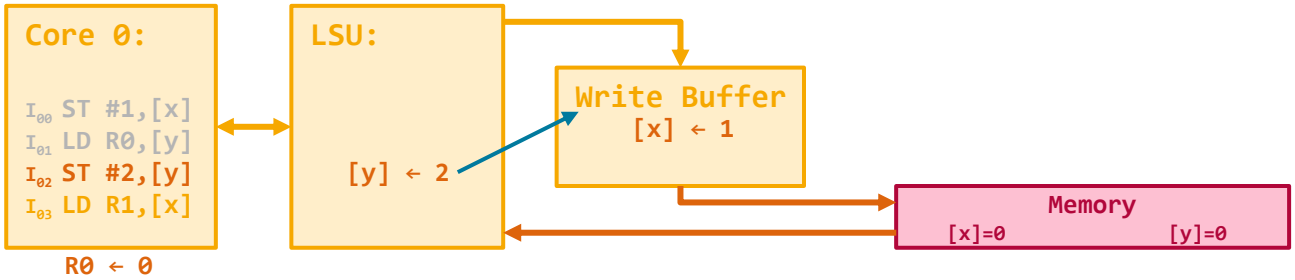
- **Load** instructions must wait for results from memory
- **Store** instructions produce no results for subsequent instructions
  - LSU does not need to wait for an issued **Store** instruction to complete
- This optimization is implemented using Write Buffers, i.e. FIFO memories storing address and data of pending **Store** operations
- To maintain in-order illusion, subsequent **Loads** to addresses present in Write Buffer must return most recent buffered data (Bypass)



Shared-Memory Hardware
Memory Consistency Models

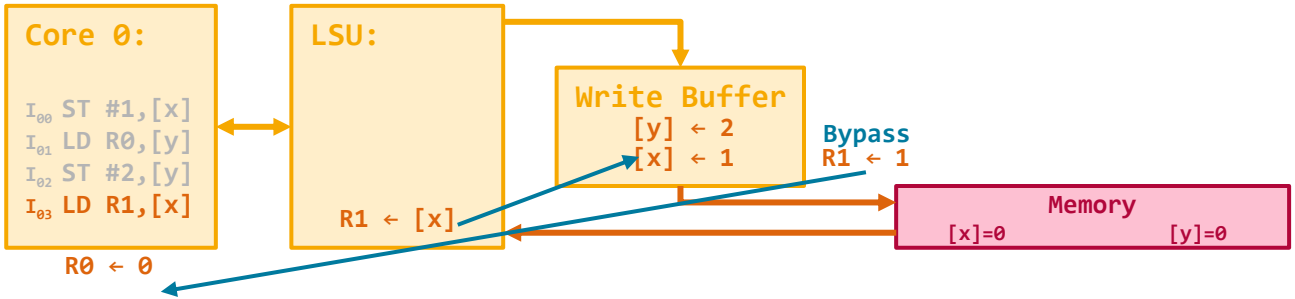
Excursion: Write Buffers

- **Load** instructions must wait for results from memory
- **Store** instructions produce no results for subsequent instructions
  - LSU does not need to wait for an issued **Store** instruction to complete
- This optimization is implemented using Write Buffers, i.e. FIFO memories storing address and data of pending **Store** operations
- To maintain in-order illusion, subsequent **Loads** to addresses present in Write Buffer must return most recent buffered data (Bypass)



Excursion: Write Buffers

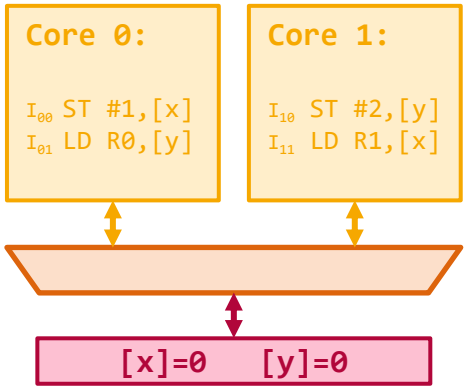
- **Load** instructions must wait for results from memory
- **Store** instructions produce no results for subsequent instructions
  - LSU does not need to wait for an issued **Store** instruction to complete
- This optimization is implemented using Write Buffers, i.e. FIFO memories storing address and data of pending **Store** operations
- To maintain in-order illusion, subsequent **Loads** to addresses present in Write Buffer must return most recent buffered data (Bypass)

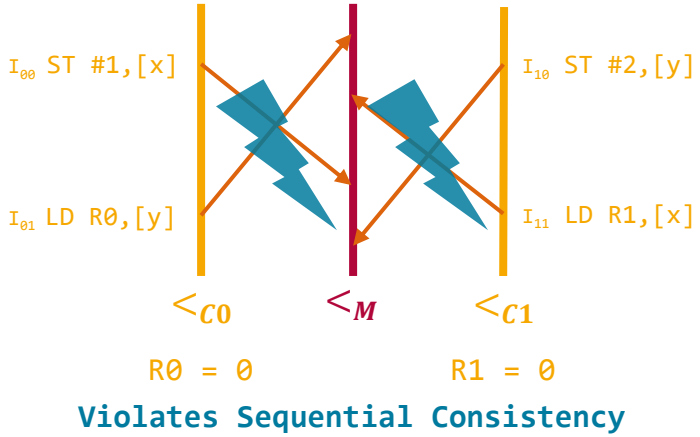


Shared-Memory Hardware
Memory Consistency Models

What happens, if multiple (in-order) cores with <u>write buffers</u> concurrently access memory?

- Total Store Order
- **Stores** I_{00} and I_{10} wait in write buffer, while **Loads** I_{01} and I_{11} can proceed
- **Store-Load**-Reordering violates Sequential consistency
- Define new consistency model to accommodate write buffers





Total Store Order

- Abbreviation: $I_x \overset{M}{\leftarrow} I_y$ means that consistency model M guarantees:

$$I_x <_C I_y \Rightarrow I_x <_M I_y$$

- Formal description of TSO considers **Load** and **Store** instructions separately:
  - Maintain **Load-Load** order: $L_a \overset{TSO}{\longleftarrow} L_b$
  - Maintain **Load-Store** order: $L_a \overset{TSO}{\longleftarrow} S_b$
  - Maintain **Store-Store** order: $S_a \overset{TSO}{\longleftarrow} S_b$
  - No clause requiring **Store-Load** order

Total Store Order

- If required, Store-Load reordering can be explicitly forbidden by interposing a **Fence** instruction
  - **Fence** effectively flushes the write buffer before performing any more **Load** instructions
- Additional clauses to formalize **Fence** (transitively ensures order between preceding and subsequent instructions):

$$L_a \overset{TSO}{\longleftarrow} F \quad ; \quad S_a \overset{TSO}{\longleftarrow} F$$

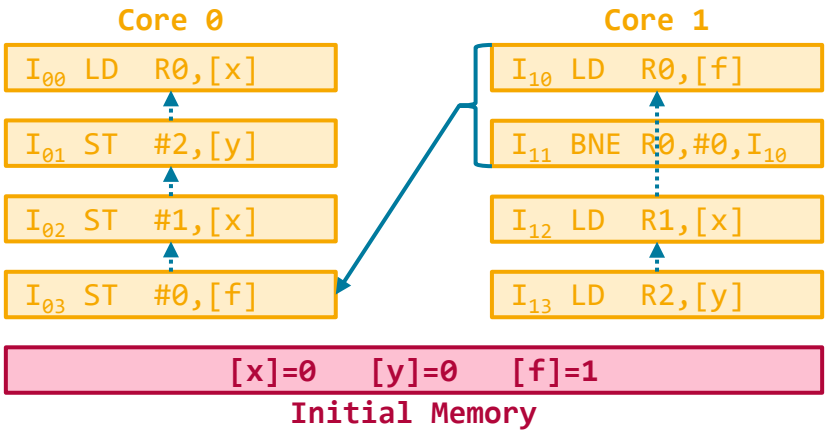
$$F \overset{TSO}{\longleftarrow} F$$

$$F \overset{TSO}{\longleftarrow} L_a \quad ; \quad F \overset{TSO}{\longleftarrow} S_a$$

Shared-Memory Hardware
Memory Consistency Models

Total Store Order

- Widely implemented in architectures like SPARC and x86
- Missing **Store-Load** order is not problematic for most programming idioms:
  - Example: Guard access to variables using a flag



Shared-Memory Hardware
Memory Consistency Models

Atomic Operations

- To make flag from previous example a proper lock, the acquire operation must atomically check and set it (i.e. using Test and Set instruction, **`TAS`**)
- New instruction type: Read-Modify-Write (**`RMW`**)
  - Combination of a **`Load`** and subsequent **`Store`** to the same address
  - No other accesses to that address may happen in between
- For consistency model, **`RMW`** acts as both **`Load`** and **`Store`**
  - SC and TSO maintain order between **`RMW`** and any other instruction type
- Possible **`RMW`** implementation: Flush write buffer, then lock the memory multiplexer not to switch between the **`Load`** and **`Store`** part of the instruction

Shared-Memory Hardware
Memory Consistency Models

Weak Consistency

- TSO **Fence** demonstrates explicit request of ordering guarantees by programmer

- Many orderings are not required but enforced by strong consistency models like SC and TSO
- To release optimization potential, define consistency model that gives only explicitly requested ordering guarantees
  - **Fences** indicate required order
- Only guarantee without **Fence** is ordering between operations on the same address

Weak Consistency

- Formalization:
  - Maintain order of operations on the same address:

$$L_a \overset{WC}{\leftarrow} L_a \quad ; \quad L_a \overset{WC}{\leftarrow} S_a \quad ; \quad S_a \overset{WC}{\leftarrow} L_a \quad ; \quad S_a \overset{WC}{\leftarrow} S_a$$

  - Force order between operations on different addresses with **Fence**:

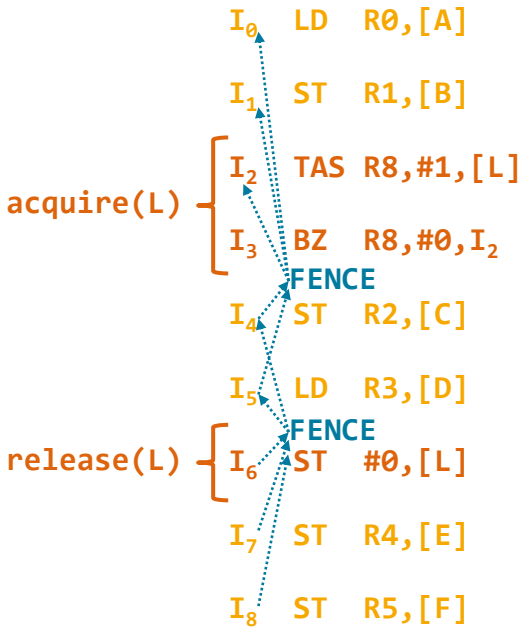
$$L_a \overset{WC}{\leftarrow} F \quad ; \quad S_a \overset{WC}{\leftarrow} F$$

$$F \overset{WC}{\leftarrow} F$$

$$F \overset{WC}{\leftarrow} L_b \quad ; \quad F \overset{WC}{\leftarrow} S_b$$

Weak Consistency

- Critical section example:
  - Section consists of I_4 and I_5
  - Guarded by lock L



Shared-Memory Hardware
Memory Consistency Models

Release Consistency

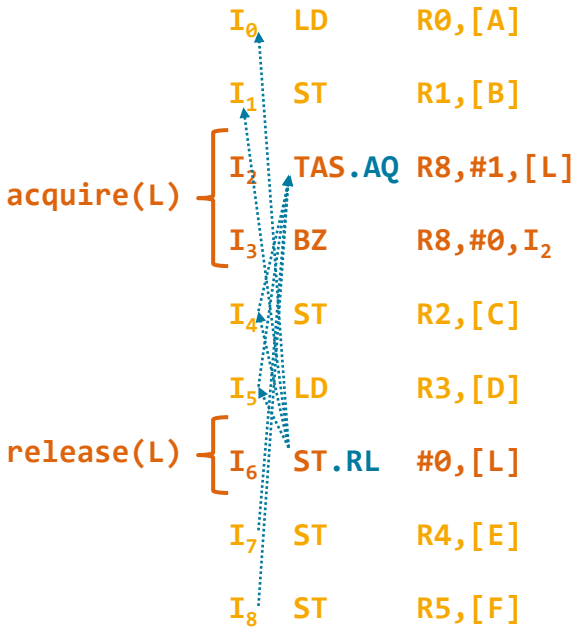
- Minimum requirement for correct critical section implementation:
  - Instructions in CS execute after **`acquire()`**
  - Instructions in CS execute before **`release()`**
- Full **`Fences`** for **`acquire()`** and **`release()`** also ensure:
  - Instructions before CS execute before **`acquire()`**
  - Instructions after CS execute after **`release()`**
  - ➢ Unnecessary guarantees sacrifice optimization potential!
- ➢ Instead use half **`Fences`**, that order in one, not both directions:
  - **`Acquire`** orders itself before subsequent instructions: $A \overset{RC}{\leftarrow} L_a \,;\, A \overset{RC}{\leftarrow} S_a$
  - **`Release`** orders preceding instructions before itself: $L_a \overset{RC}{\leftarrow} R \,;\, S_a \overset{RC}{\leftarrow} R$
  - Maintain order among **`Acquire`** and **`Release`**: $A \overset{RC}{\leftarrow} A \,;\, A \overset{RC}{\leftarrow} R \,;\, R \overset{RC}{\leftarrow} A \,;\, R \overset{RC}{\leftarrow} R$

Shared-Memory Hardware
Memory Consistency Models

Release Consistency

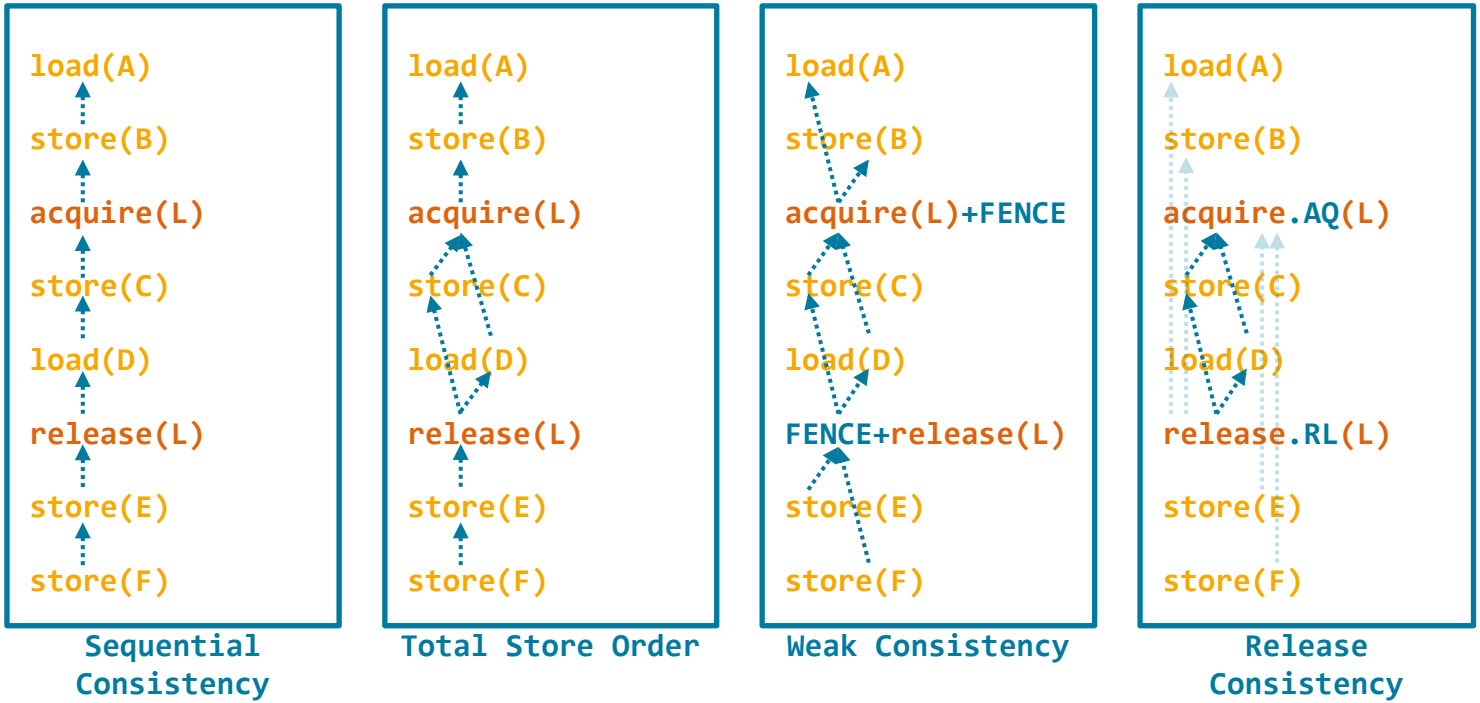
- **`Acquire`** and **`Release`** semantics can be attached to:
  - Regular **`Fence`** instructions
  - **`Load`**, **`Store`** and **`RMW`** instructions

- Allows **`acquire(L)`** and **`release(L)`** to have no ordering effect on instructions outside critical section
- Or do they?



Shared-Memory Hardware
Memory Consistency Models

Overview

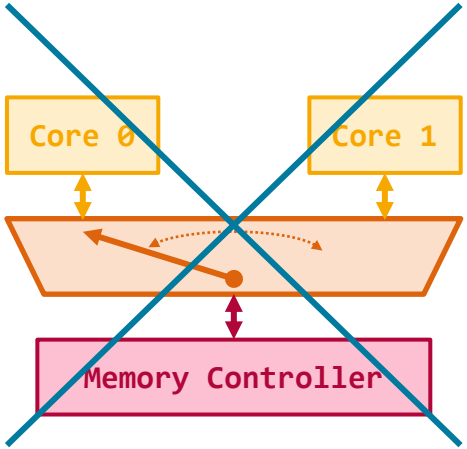


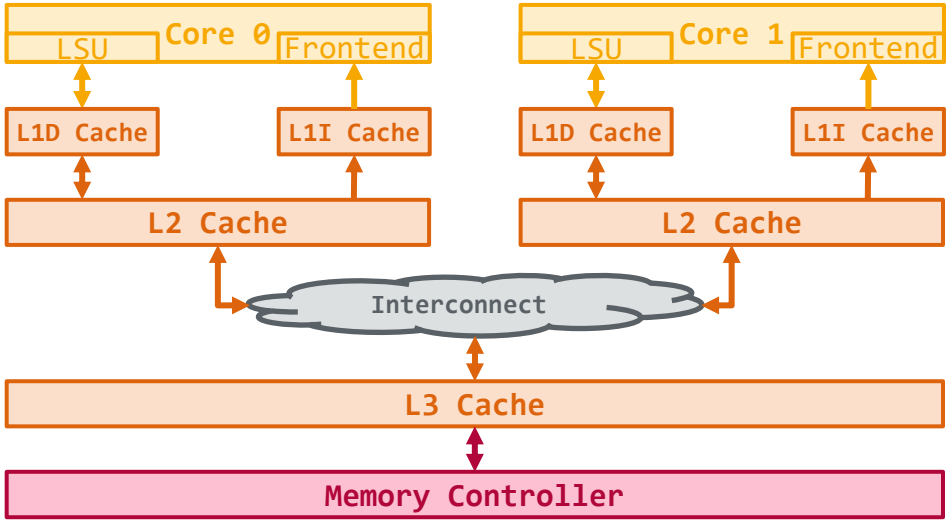
And now for a break and another cup of Darjeeling.

*or beverage of your choice

Shared-Memory Hardware
Coherent Cache Hierarchy

- Current conception of memory subsystem is inaccurate:
  - Not a multiplexer granting exclusive access to memory controller
  - Instead: Hierarchy of caches striving to reduce memory operations reaching levels closer to memory controller





Shared-Memory Hardware
Coherent Cache Hierarchy

Hasso Plattner Institut

Caches

- Store copies of recently used main memory regions (cache lines)
  - If present (cache hit) core can operate on cached copy instead of main memory

Cache lines are the basic unit of the memory subsystem!

Any access (even single byte) will fetch an entire line (64-128 byte) into the cache.

- Caches are orders of magnitude smaller that main memory
  - Faster implementation: Lower access latency and higher throughput
- Resulting performance approaches that of the cache for a high hit ratio

$$Latency_{avg} = Latency_{Cache} \cdot HitRatio + Latency_{MainMem} \cdot (1 - HitRatio)$$

  - ➢ Illusion of memory with cache speed and main memory size

- Requires high hit ratio
  - ➢ Based on temporal and spatial locality

Shared-Memory Hardware
Coherent Cache Hierarchy

Prefetching

- Technique to improve the cache hit ratio:

  Hardware predicts or software indicates cache lines that will be accessed in the near future and fetches them proactively.
- Software: Explicit prefetch instructions like **`PREFETCHxx`** (x86) or **`DCBT`** (Data Cache Block Touch, Power)

- Tradeoff: Aggressive, erroneous or premature prefetching may defeat its purpose by evicting still used cache lines or wasting memory bandwidth.
  - **Coverage**: Ratio of accessed locations that were successfully prefetched
  - **Accuracy**: Ratio between prefetched locations that were and were not accessed

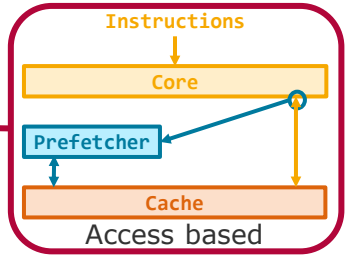
Shared-Memory Hardware
Coherent Cache Hierarchy

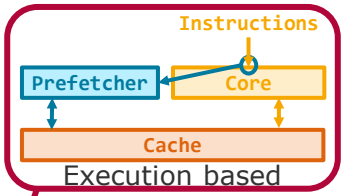
Prefetching

- Access based Prefetchers observe all memory accesses or only cache misses

  Each access/miss might trigger a prefetch

  Prefetched address is predicted using information associated with access/miss (address, program counter, history of addresses or offsets)
  - Temporal Correlation: Record sequence of accessed addresses
  - Spatial Correlation: Record data layout of accessed structures (relative offsets)
    - Stride Prefetcher: Recognizes layouts with fixed relative offsets
- Execution based Prefetchers analyze instruction stream directly to predict locations it might access





Shared-Memory Hardware
Coherent Cache Hierarchy

Caches

- Distort global visibility of memory operations by cores!
  - Delayed propagation of **`Stores`** to main memory
  - Stale results from **`Loads`** by missing updates to main memory

- Order is restored by establishing the Single-Writer-Multiple-Reader (SWMR) invariant between caches:
  - A cache can only service a **`Store`** operation on a cache line if no other cache can service **`Loads`** or **`Stores`** from the same cache line
  - Multiple caches can service **`Loads`** on their local cache line copies as long as no **`Stores`** to the same cache line occur
- Caches obeying the SWMR invariant are called coherent
  - Mechanisms to maintain the SWMR invariant are coherence protocols

Shared-Memory Hardware
Coherent Cache Hierarchy

MSI Coherence Protocol

- MSI is a simple coherence protocol, based on a state machine
- Seen from a particular cache, each cache line is in one of three states:
  - Invalid: The cache line is not present in the cache, this cache may service neither **`Load`** nor **`Store`** operations
  - Shared: The cache line is present in this and probably other caches, this cache may service **`Load`** operations
  - Modified: The cache line is only present in this cache, this cache may service **`Load`** and **`Store`** operations

Shared-Memory Hardware
Coherent Cache Hierarchy

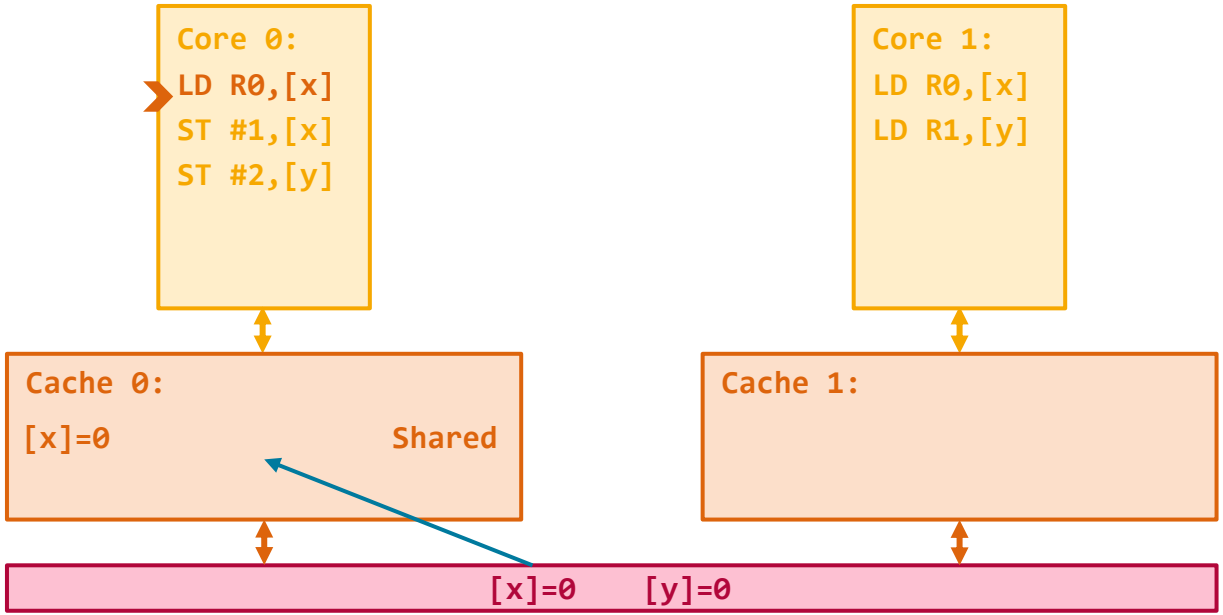
MSI Coherence Protocol

- Transitions may occur for two reasons:
  1) Required for servicing **`Loads`** or **`Stores`** from core
  2) Reacting to observed behavior of other caches (Snooping)

- Examples:
  1) *If the cache needs to service a Write operation on a Shared line, it must broadcast an invalidation message to all caches to ensure it holds the only copy before marking its line Modified.*
  2) *If a cache holds a Modified line, it must snoop accesses to this line by other caches, if necessary write back its updates to memory and transition to Invalid state.*

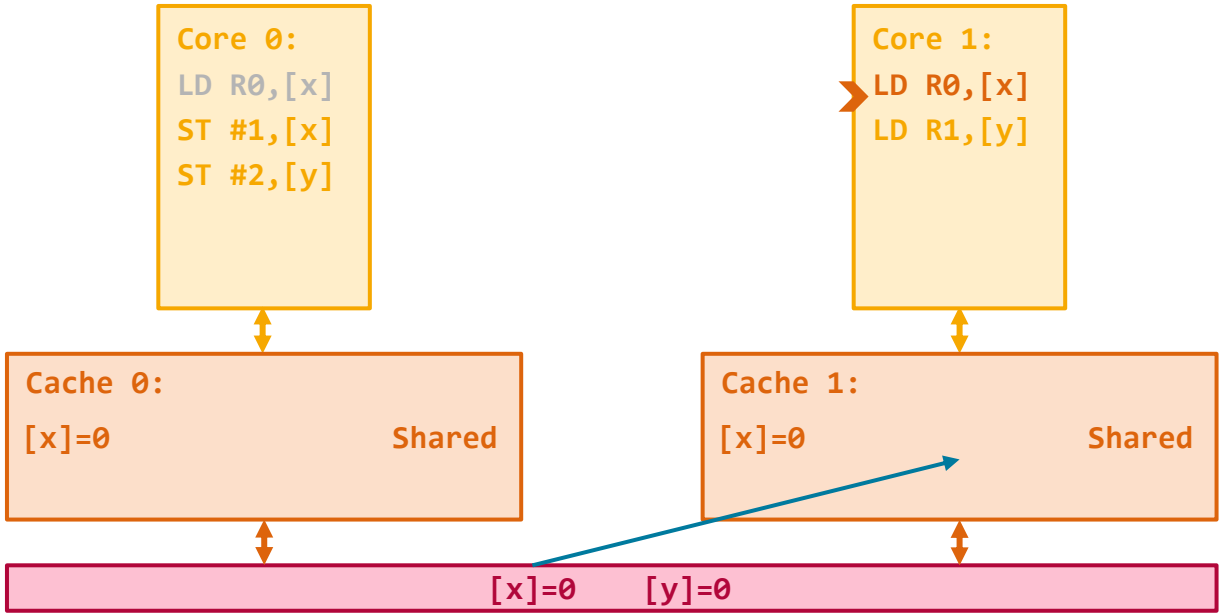
Shared-Memory Hardware
Coherent Cache Hierarchy

MSI Coherence Protocol



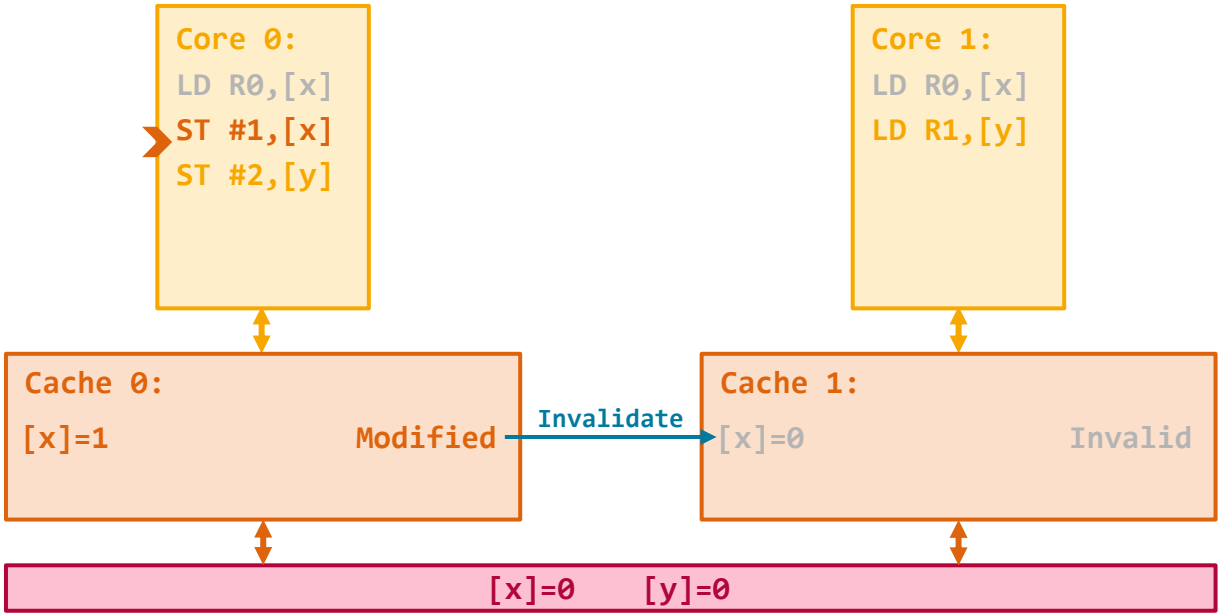
Shared-Memory Hardware
Coherent Cache Hierarchy

MSI Coherence Protocol



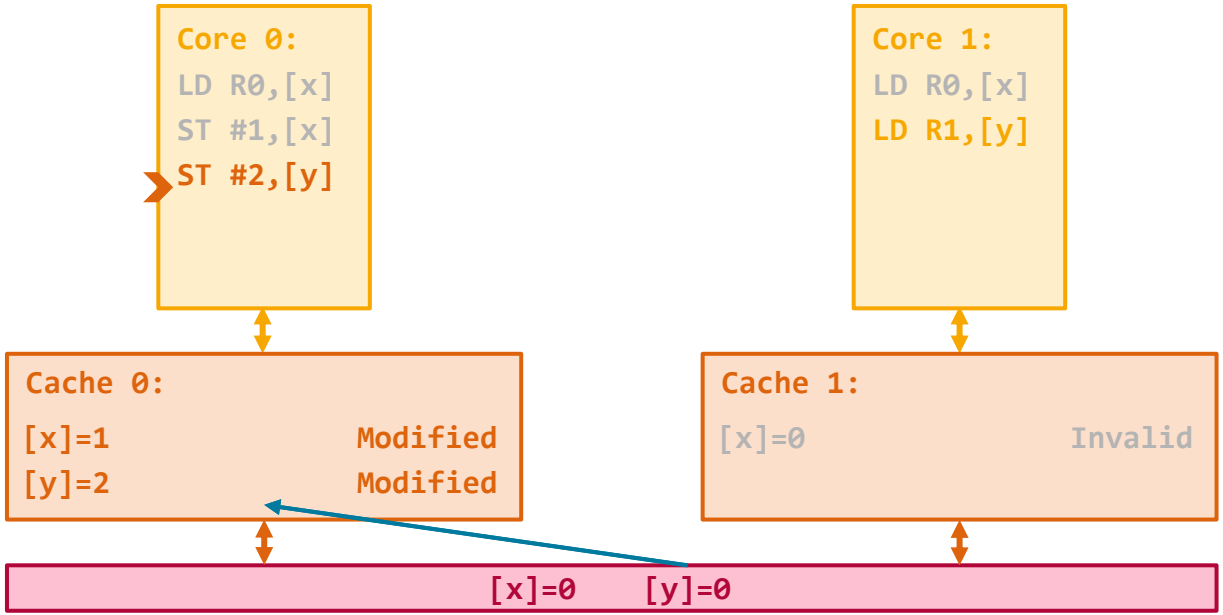
Shared-Memory Hardware
Coherent Cache Hierarchy

MSI Coherence Protocol



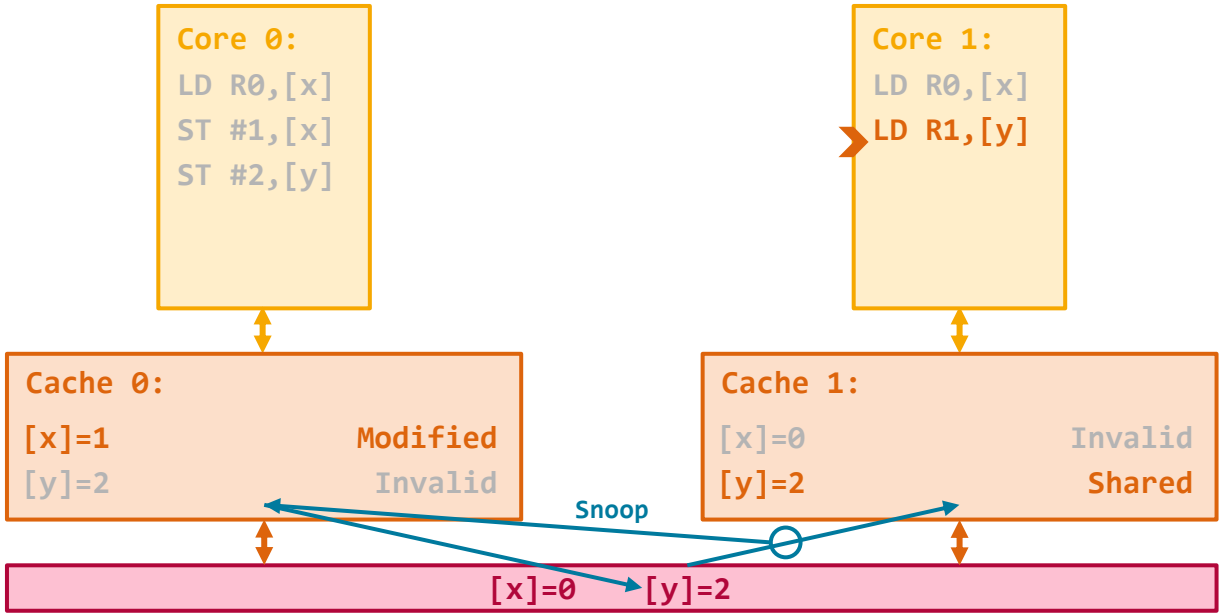
Shared-Memory Hardware
Coherent Cache Hierarchy

MSI Coherence Protocol



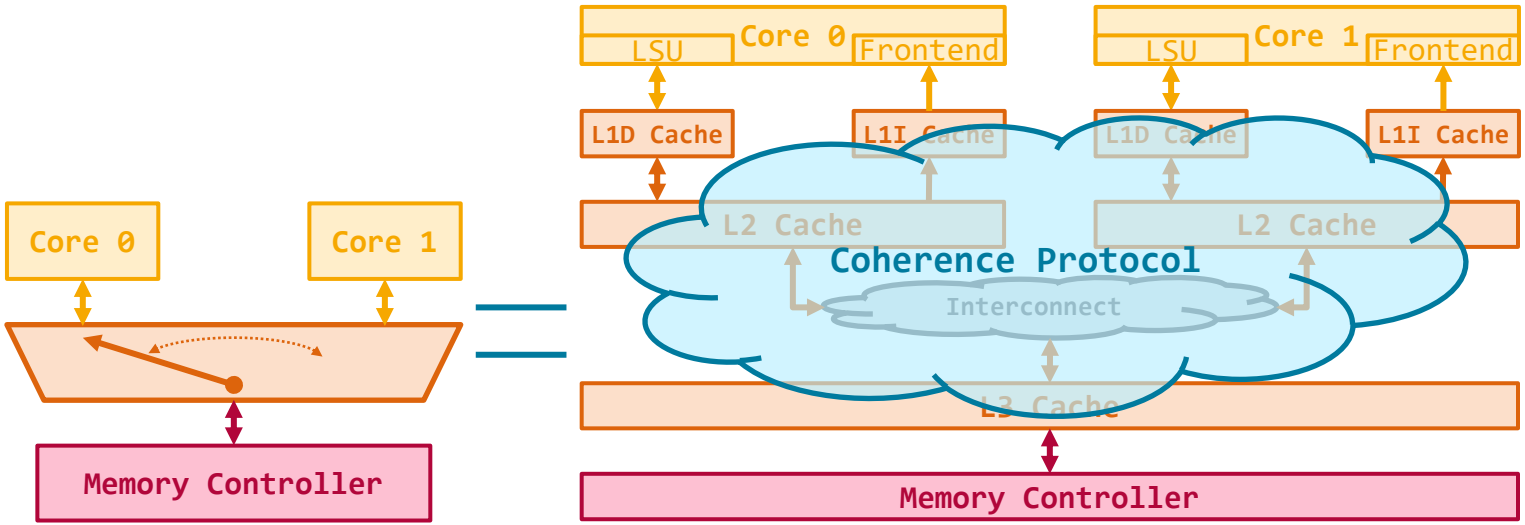
Shared-Memory Hardware
Coherent Cache Hierarchy

MSI Coherence Protocol



Shared-Memory Hardware
Coherent Cache Hierarchy

A coherent cache hierarchy reestablishes sequential consistency equivalent to the original multiplexer model!



Shared-Memory Hardware References

- **„Computer Architecture, A Quantitative Approach"**. Hennessy, John and Patterson, David. Sixth Edition. Morgan Kaufmann Publishers. 2018.

- **„A Primer on Memory Consistency and Coherence"**. Sorin, Daniel and Hill, Mark and Wood, David. First Edition. In „Synthesis Lectures on Computer Architecture". Morgan & Claypool Publishers. 2011.
- **„A Primer on Hardware Prefetching"**. Falsafi, Babak and Wenisch, Thomas. First Edition. In „Synthesis Lectures on Computer Architecture". Morgan & Claypool Publishers. 2014.
- **„Multithreading Architecture"**. Nemirovsky, Mario and Tullsen, Dean. First Edition. In „Synthesis Lectures on Computer Architecture". Morgan & Claypool Publishers. 2012.

Can be accessed from the Uni Potsdam network at:

https://www.morganclaypool.com/toc/cac/1/

HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

And now for a break and the last cup of Darjeeling.

*or beverage of your choice