

CPU组 Seminar



Issue

北京大学系统结构研究所
2013-05-15



Contents

- Introduction(to issue logic for memory operations)
- Memory Disambiguation
 - Non-speculative Memory Disambiguation
 - Speculative Memory Disambiguation
- Speculative Wakeup of Load Consumer
- Memory Disambiguation on UniCore-3

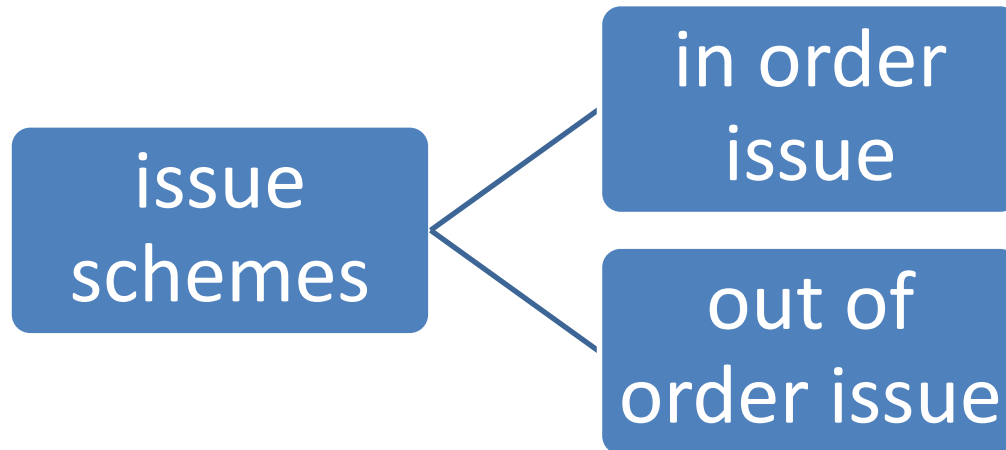


1、 Introduction

- Issue
- Issue for memory operations

Issue(1/2)

- The issue is the pipeline stage in charge of issuing instructions to the functional units for execution.
- Conditions for instruction to be issued
 - source operands are available
 - function units are available
- Issue Schemes

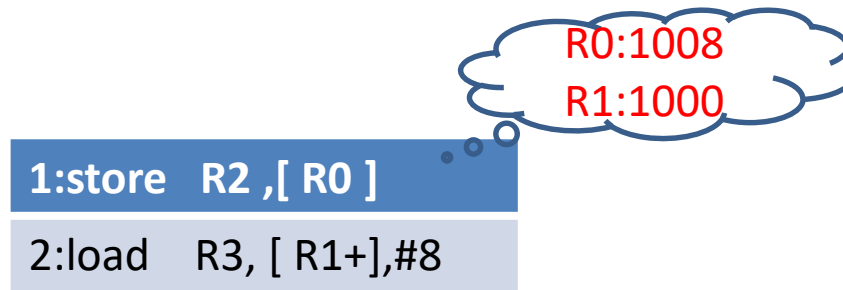


Issue(2/2)

- In order Issue
 - Issues the instructions for execution in the same order they were fetched, and if an instruction is stalled in the pipeline no later instructions can proceed.
- Out of order Issue
 - It allows out-of order execution by issuing instructions to the functional units as soon as their source operands become available.

Issue for memory operations(1/2)

- why the issue logic for memory operations is complex?
 - **Data Dependence**: Conversely to the rest of operations where data dependences are checked at the renaming stage, memory dependences cannot be identified until the memory operations compute their address



存储器: RAW

Issue for memory operations(2/2)

- **Instruction Wakeup**: The latency of a memory operation depends on whether it hits or misses in the data cache or the data TLB, where as the latency of the rest of operations is constant and only depends on the instruction itself
- **Entry Reclamation**: Memory operations often use speculative wakeup techniques which may require delaying the reclamation until we are sure the instruction can be executed, where as once the rest instructions has been selected and its data forwarded to the function unit, its issue queue entry can be safely reclaimed

Memory disambiguation policy

- The mechanism in charge of **handling memory dependences** is called memory disambiguation policy.

		NAME	DESCRIPTION
Memory disambiguation schemes	Non-speculative	Total Ordering	All memory accesses are processed in order.
		Load Ordering Store Ordering	Execution between loads and stores is out of order, but all loads execute in order among them, and all stores execute in order among them.
		Partial Ordering	All stores are processed in order, but loads execute out of order as long as all previous stores have computed their address.
	Speculative	Store Ordering	Stores execute in order, but loads execute completely out of order.



Contents

- Introduction(to issue logic for memory operations)
- **Memory Disambiguation**
 - Non-speculative Memory Disambiguation
 - Speculative Memory Disambiguation
- Speculative Wakeup of Load Consumer
- Memory Disambiguation on UniCore-3

2、 Non-speculative Memory Disambiguation

- Total Ordering
- Load Ordering with Store Ordering
- Partial Ordering

Total Ordering

- In total ordering, all memory operations are executed in order.
- Advantage
 - Simplicity
- Disadvantage
 - Constrains a lot the amount of parallelism

2、 Non-speculative Memory Disambiguation

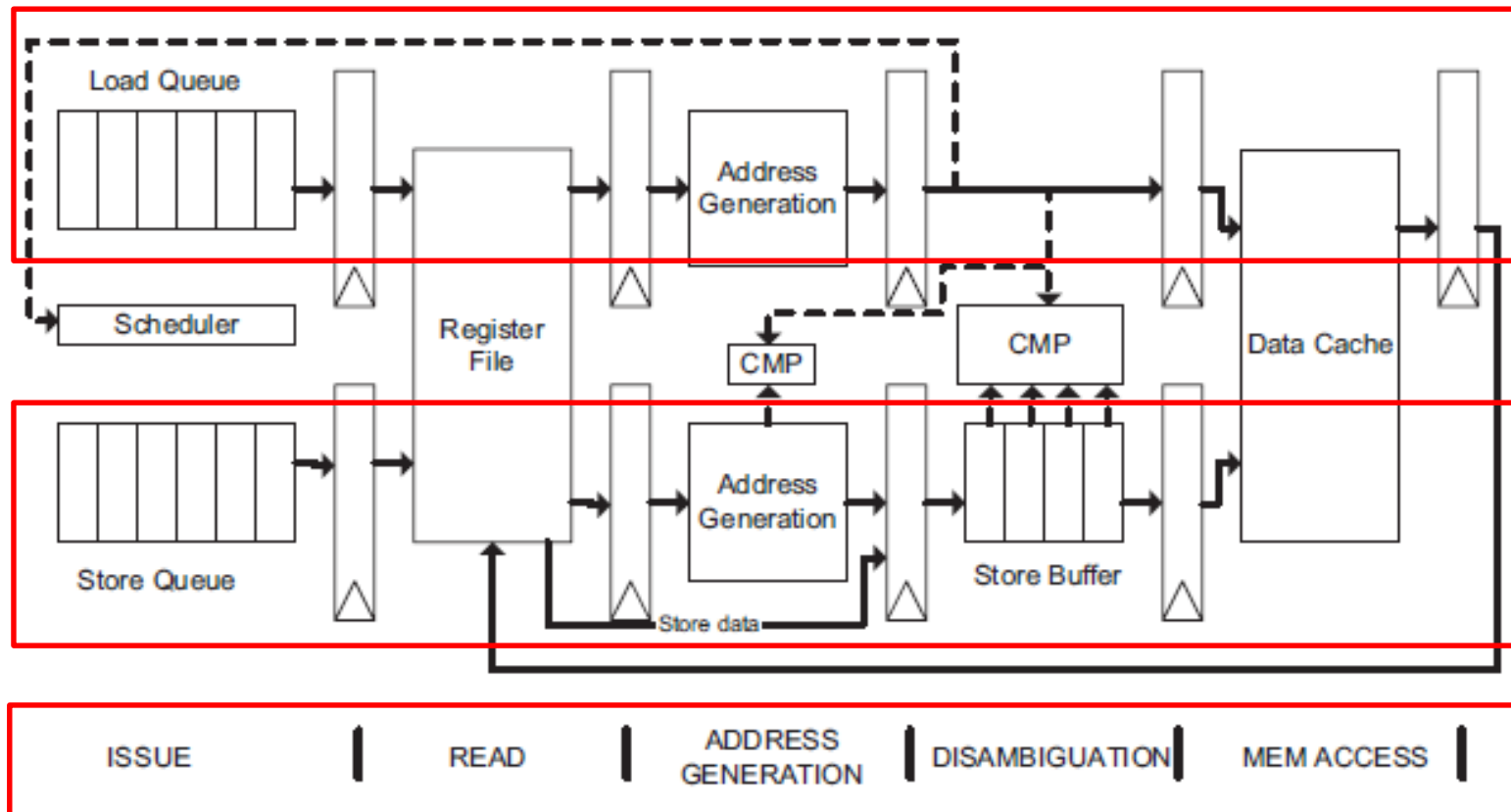
- Total Ordering
- Load Ordering with Store Ordering
- Partial Ordering

Load Ordering with Store Ordering

- Loads proceed in order, and stores proceed in order.
- Loads do not have to wait for previous stores to commit.
- Example: AMD K6

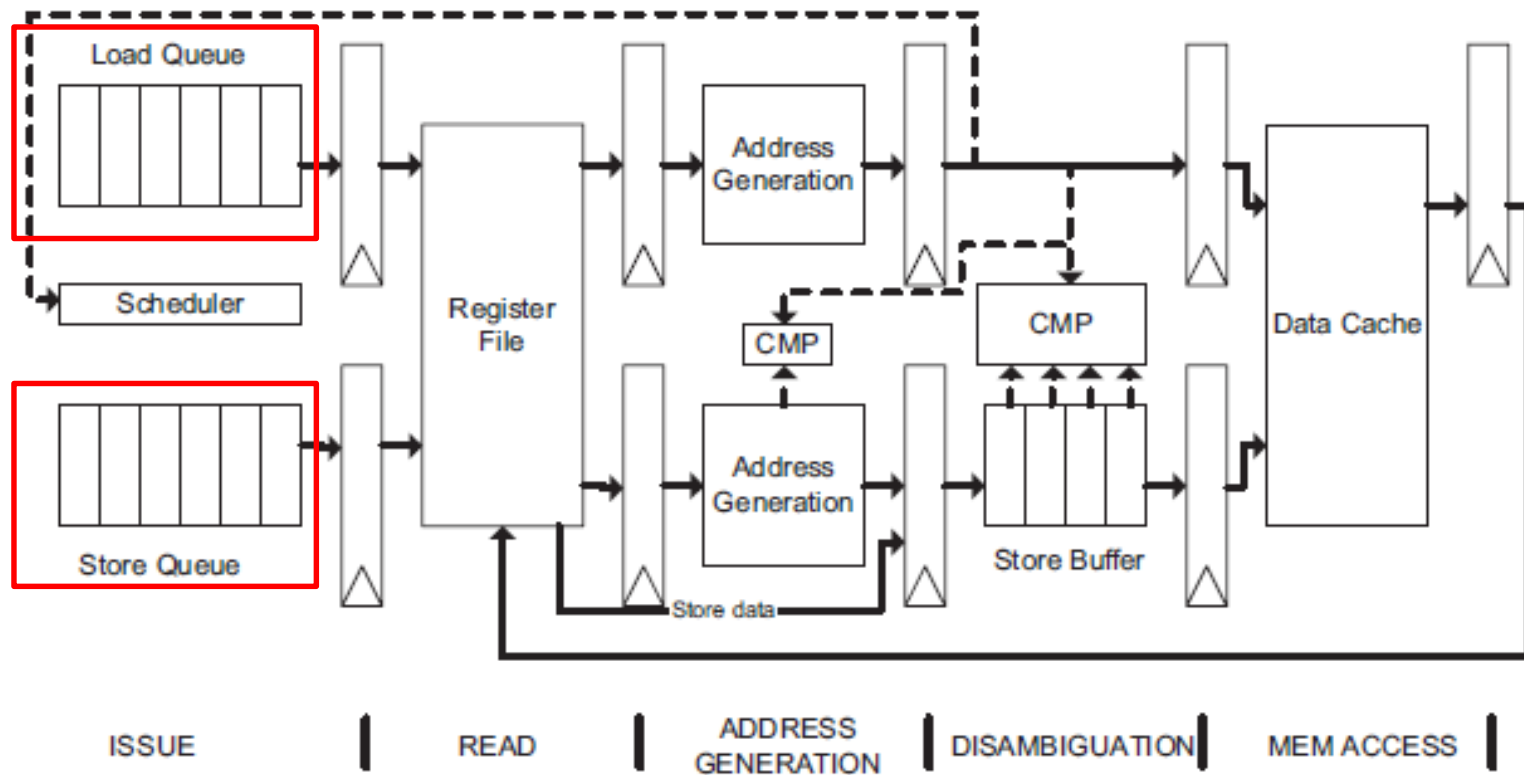
AMD K6 — Overview

- two separate pipelines for load and store operations.
- instructions flow in strict order inside each pipeline.



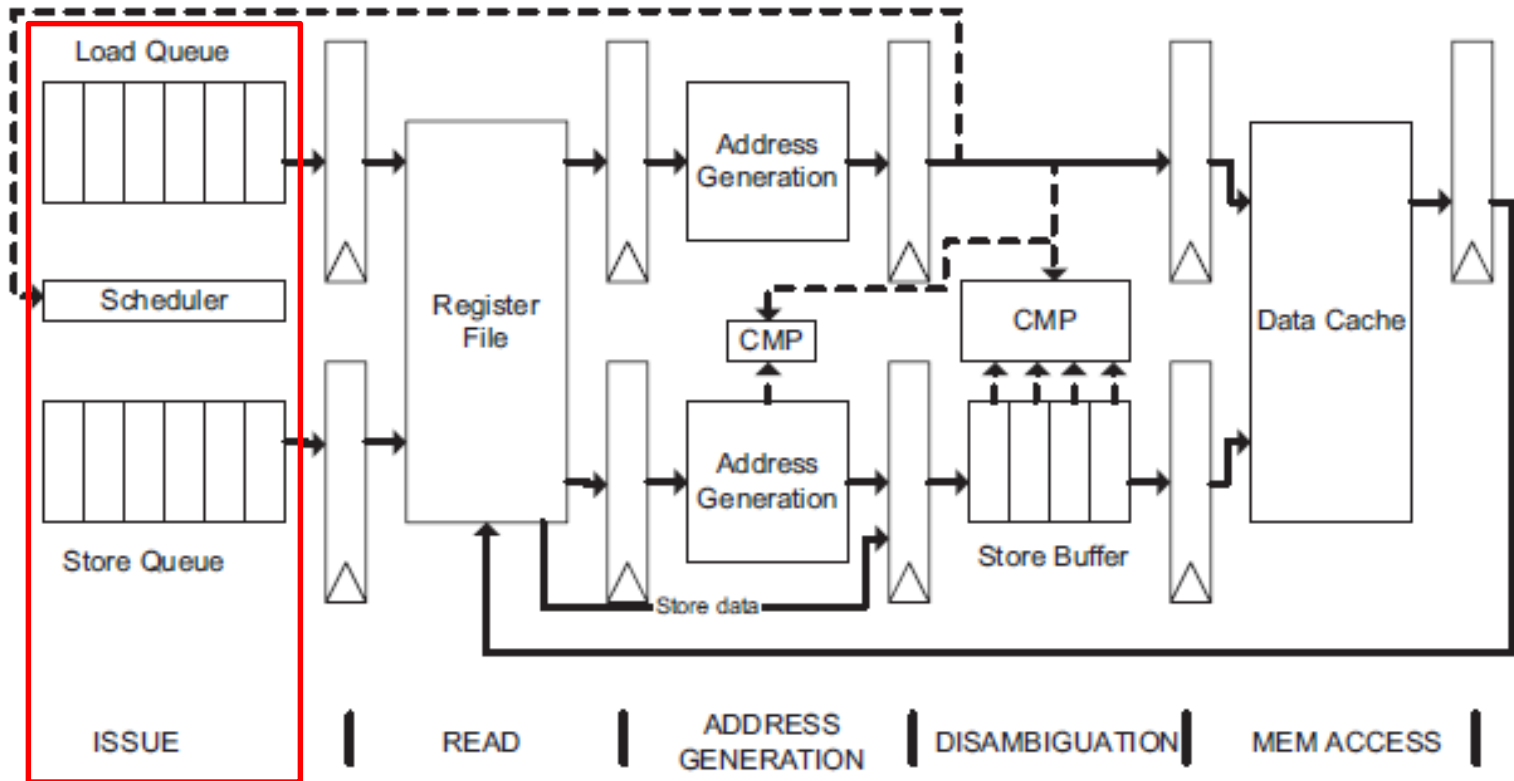
AMD K6 —Execution(1/5)

- Load Queue: Stores the load operations in program order. Loads are inserted in this queue after renaming.
- Store Queue: Stores the store operations in program order.



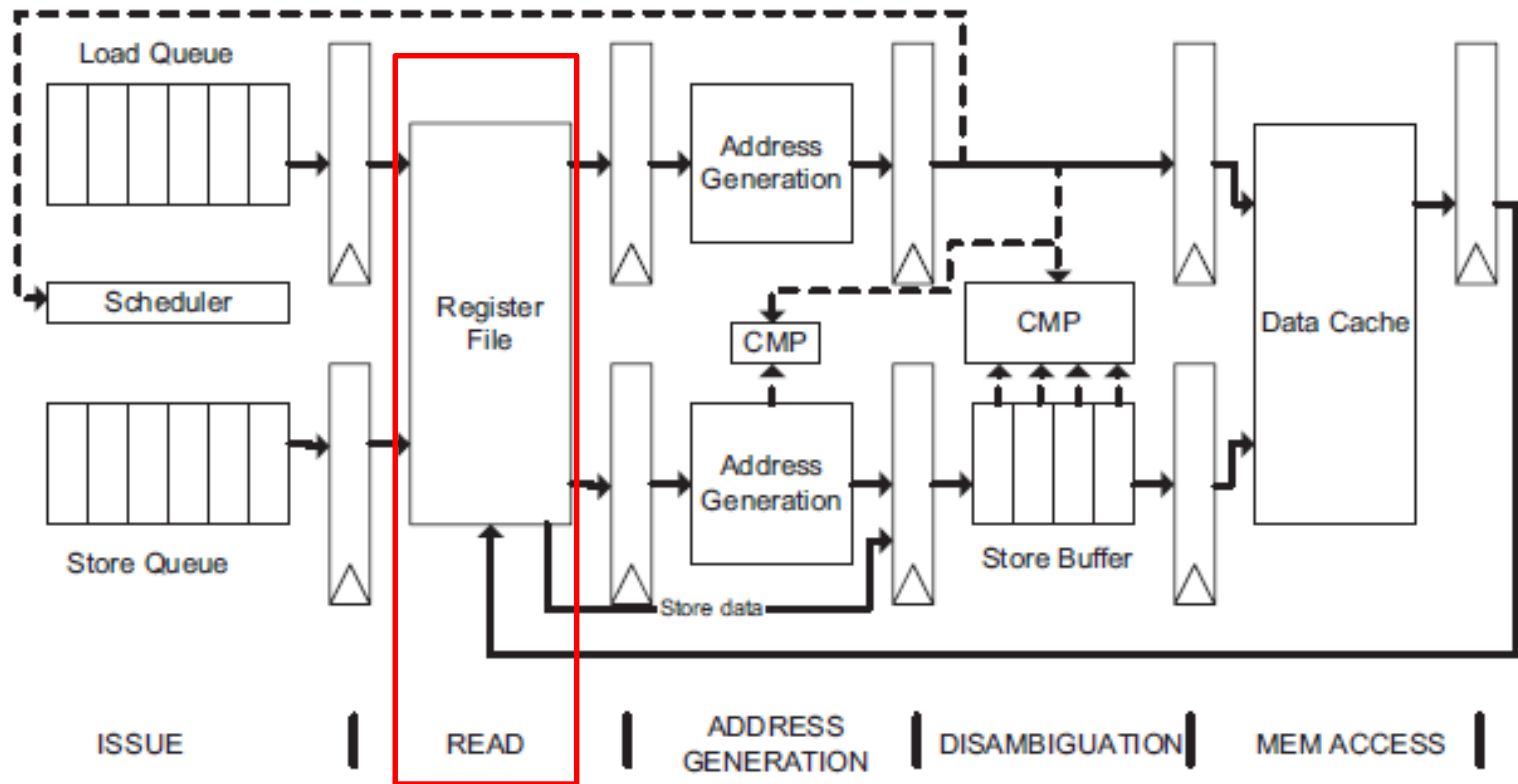
AMD K6 —Execution(2/5)

- If the source operands are ready and it is the oldest instruction on the load and store queue, respectively, issue it.



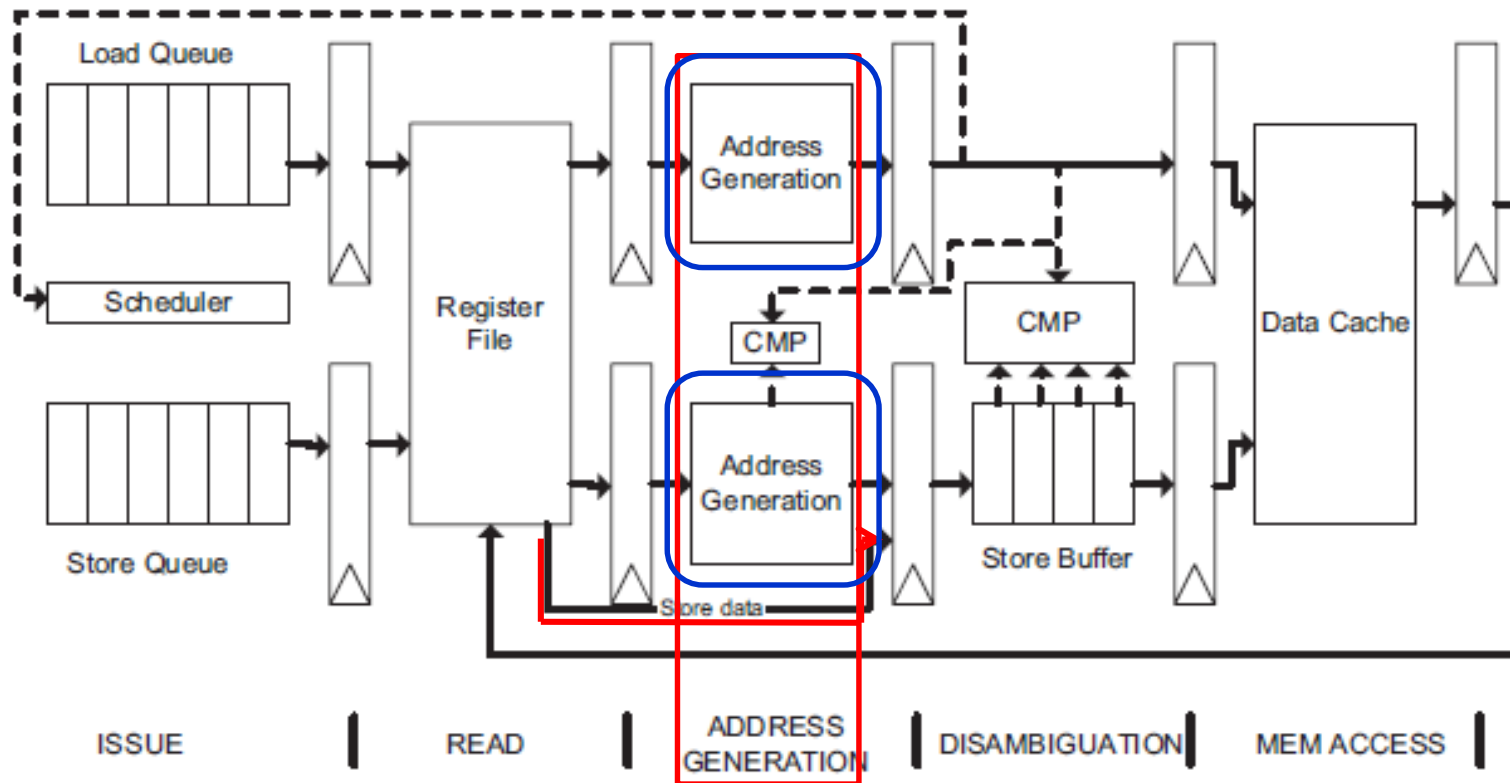
AMD K6 —Execution(3/5)

- Read the source operands from the register file or obtain them from the bypass logic.



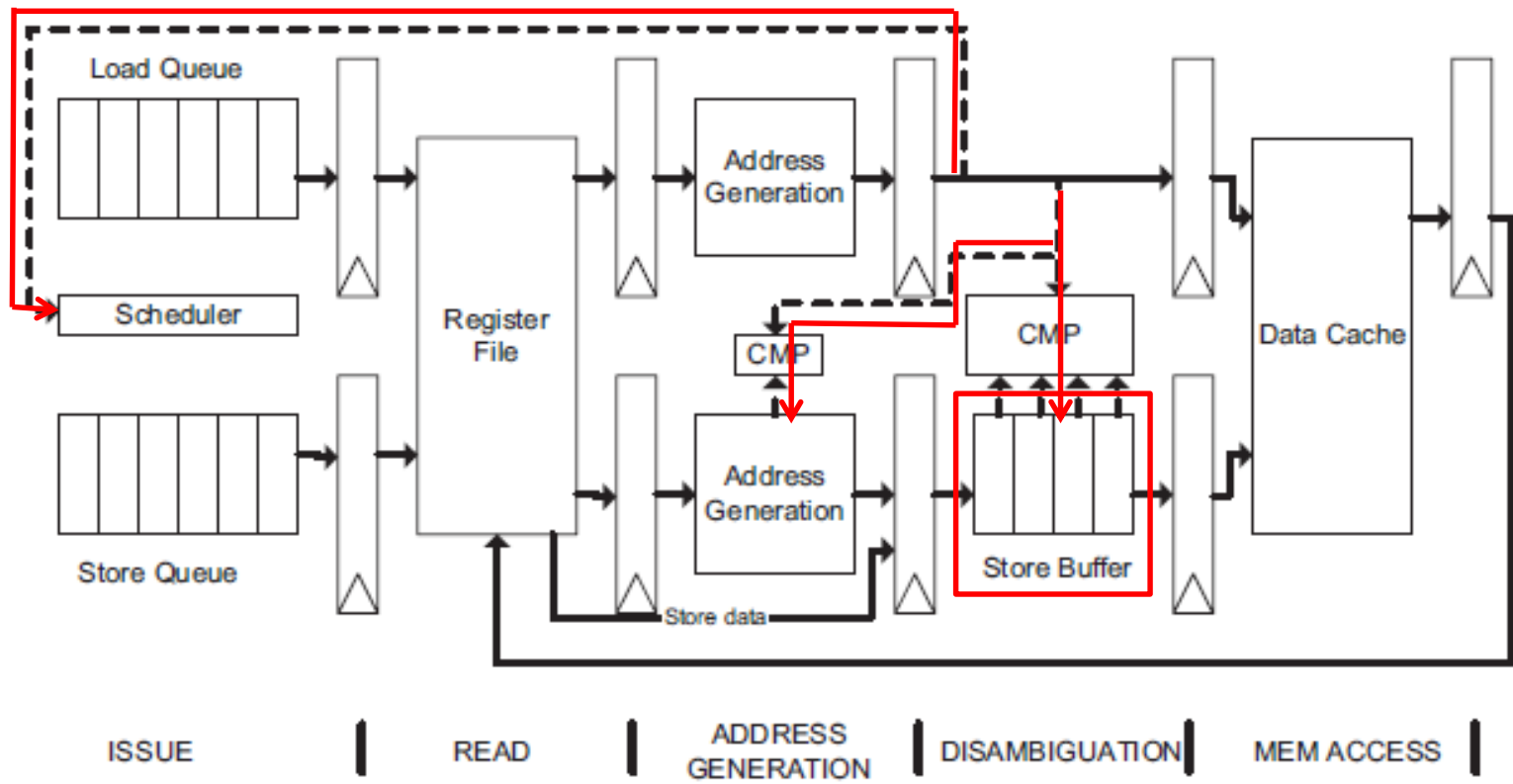
AMD K6 —Execution(4/5)

- Compute the address.
- Store: read data to be stored.



AMD K6 —Execution(5/5)

- Read the source operands from the register file or obtain them from the bypass logic.



2、 Non-speculative Memory Disambiguation

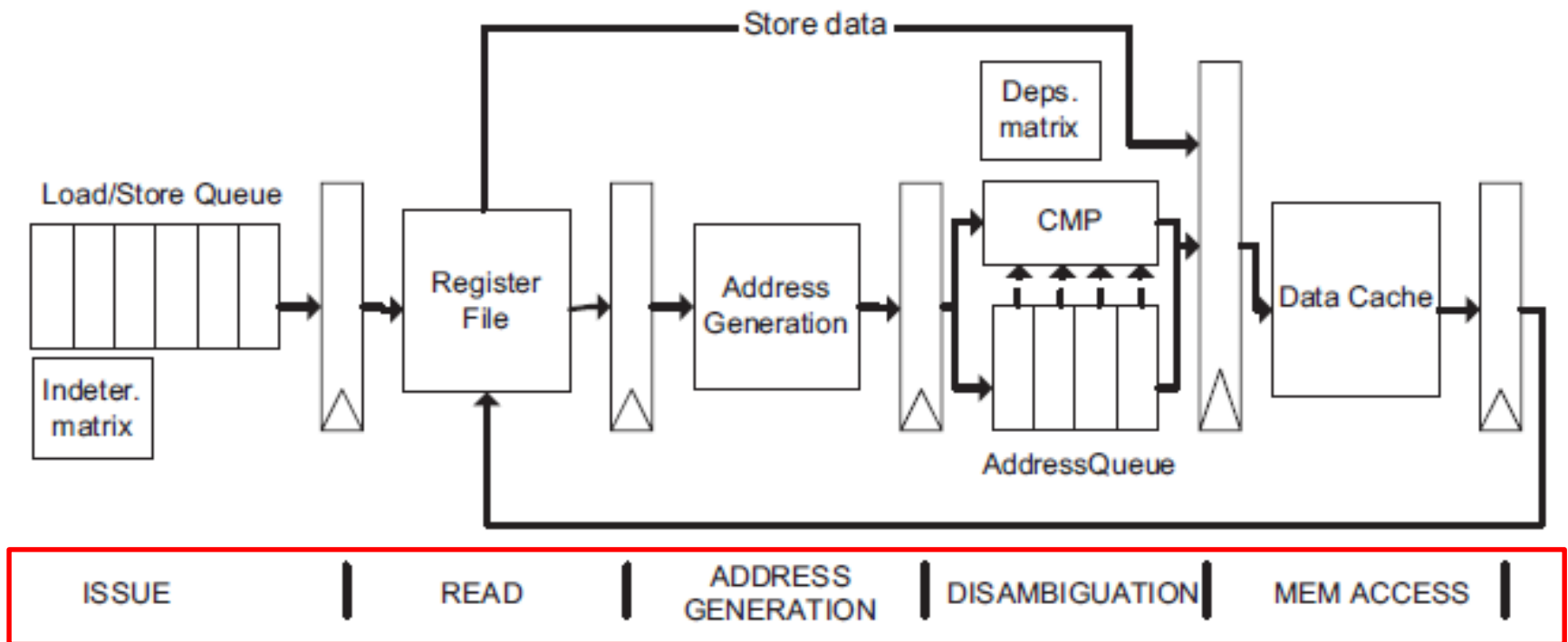
- Total Ordering
- Load Ordering with Store Ordering
- Partial Ordering

Partial ordering

- Loads can be processed out of order
 - a load can be issued as long as it has its source operands ready and all previous stores already have computed its address.
- Examples : MIPS R10000、AMD K8、UniCore-3

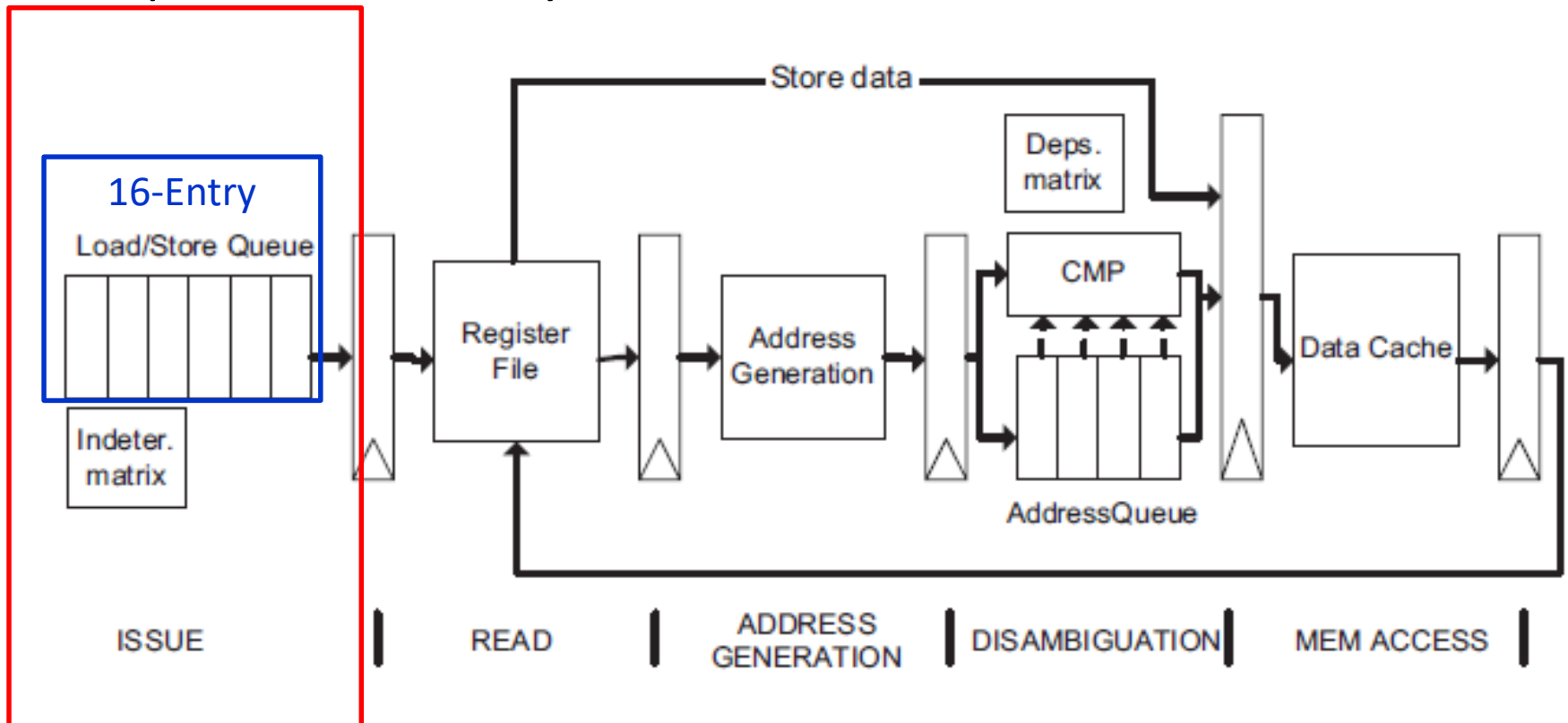
MIPS R10000 — Overview

- Loads can be executed out of order as long as all previous memory operations have computed their address.
- Stores are processed in strict program order.



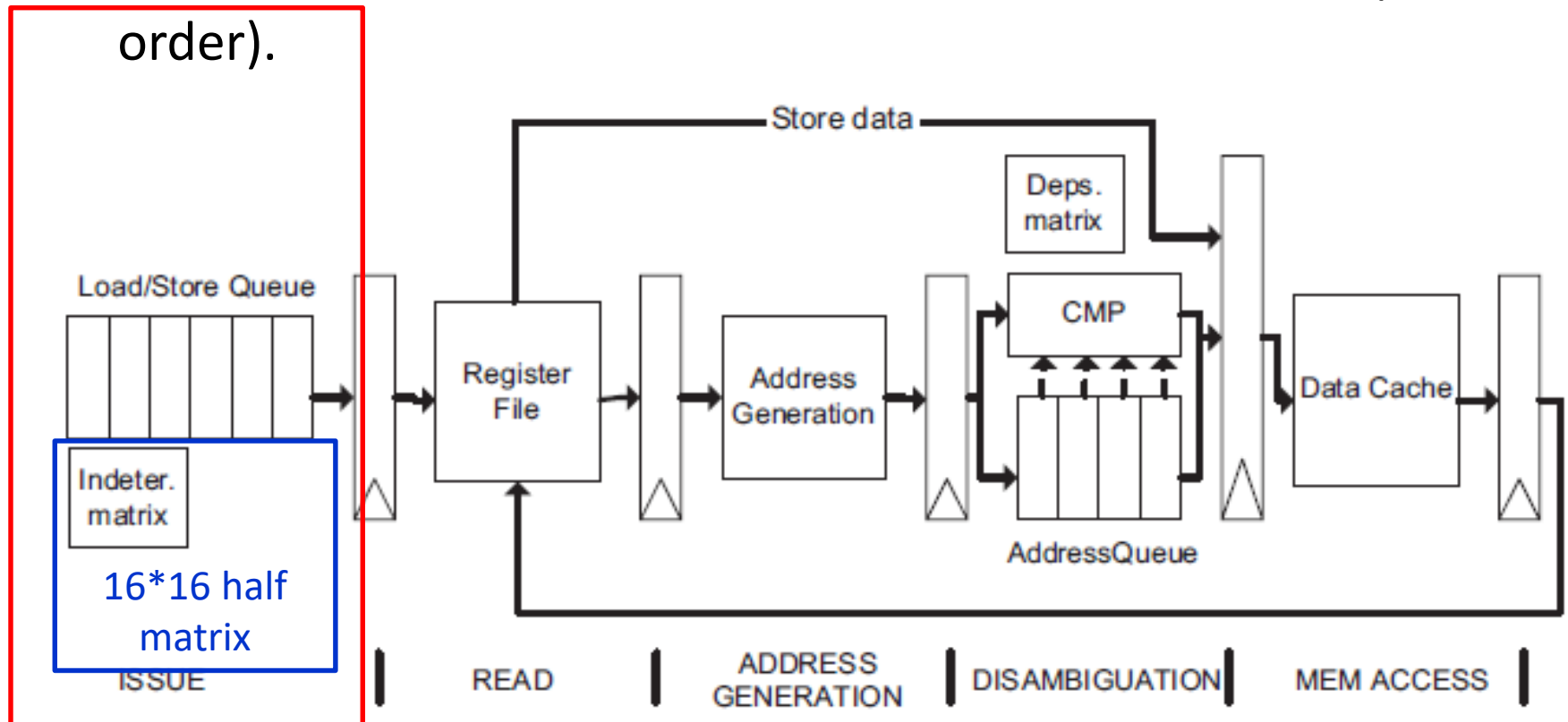
MIPS R10000 — Execution(1/9)

- Loads and store instructions are stored in order.
- Instructions do not leave this queue until their source operands are ready.



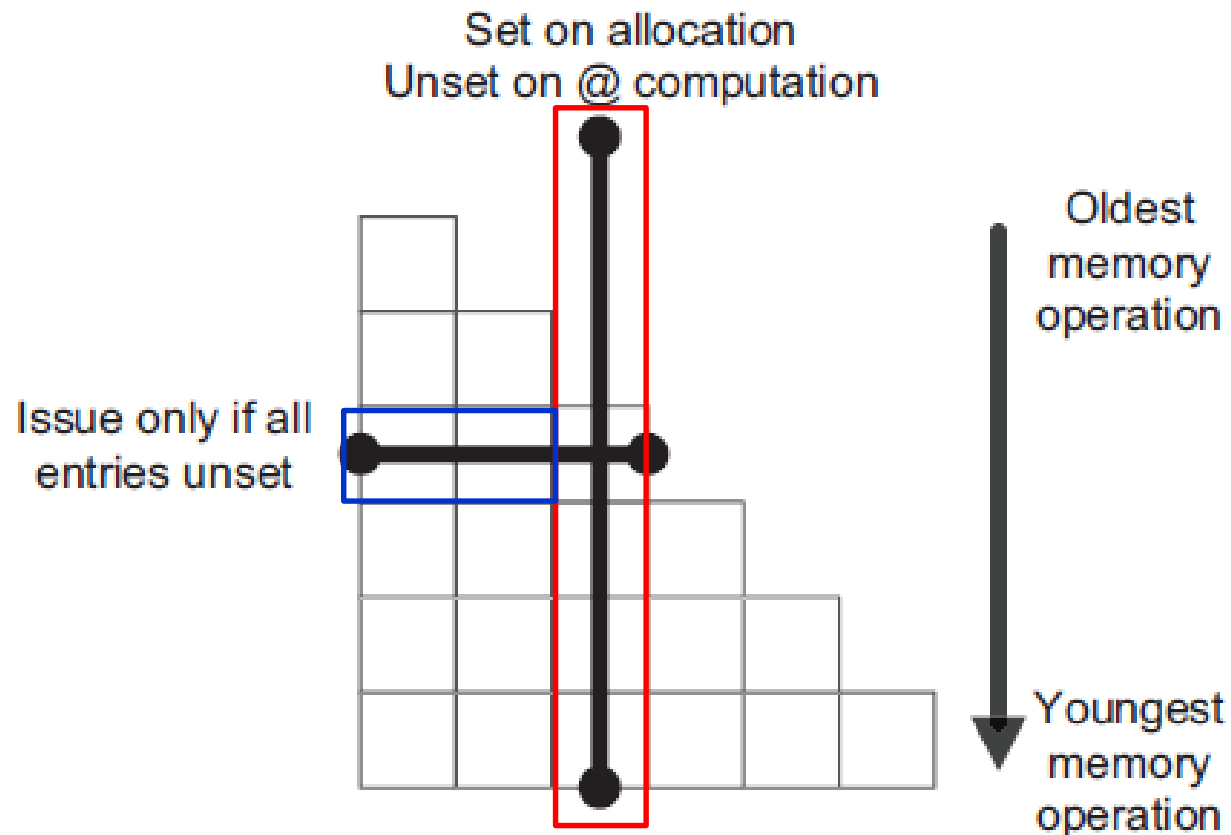
MIPS R10000 — Execution(2/9)

- Every column and row represents an entry on the load/Store queue.
- decide when the load/store instructions can be issued(in order).

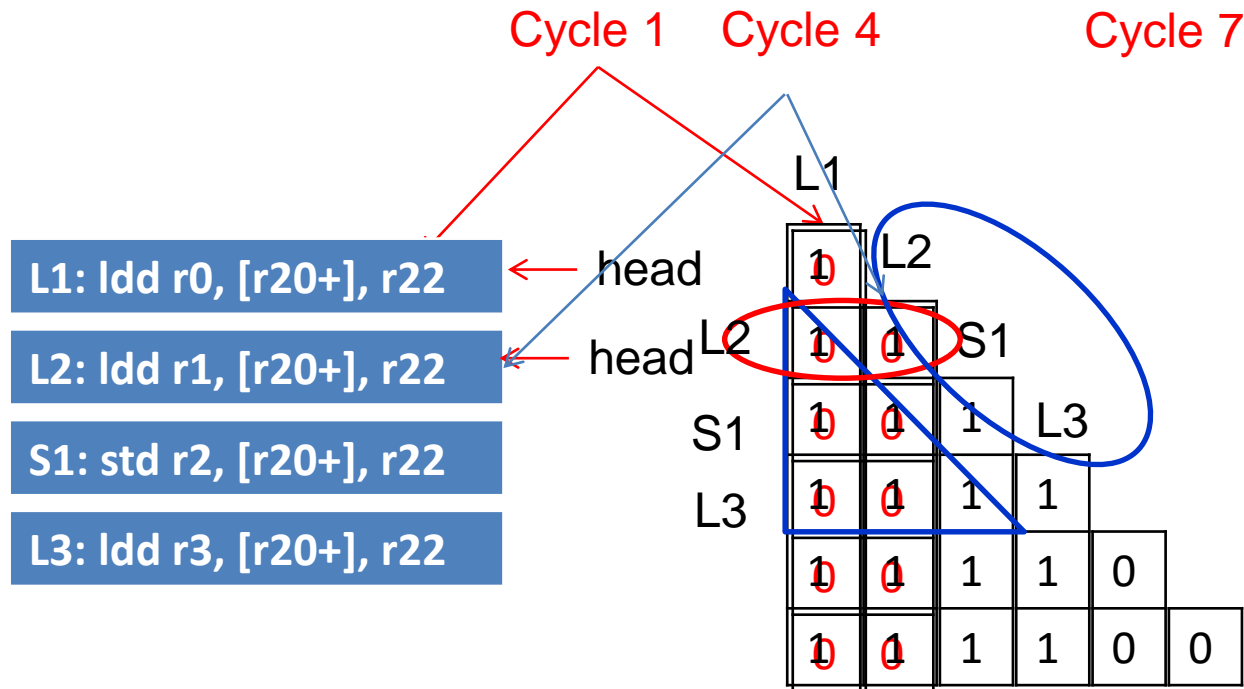


MIPS R10000 — Execution(3/9)

- Example of a 6-entry indetermination matrix.

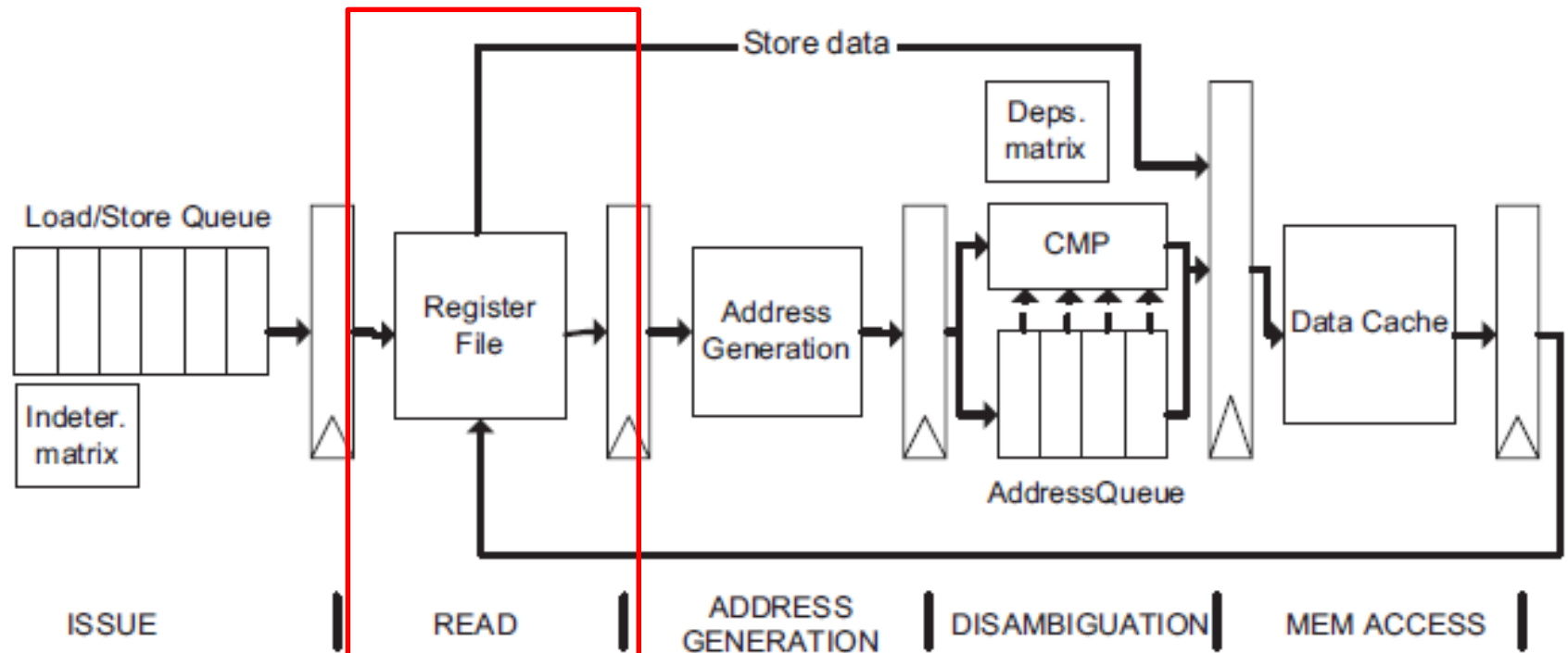


MIPS R10000 — Execution(4/9)



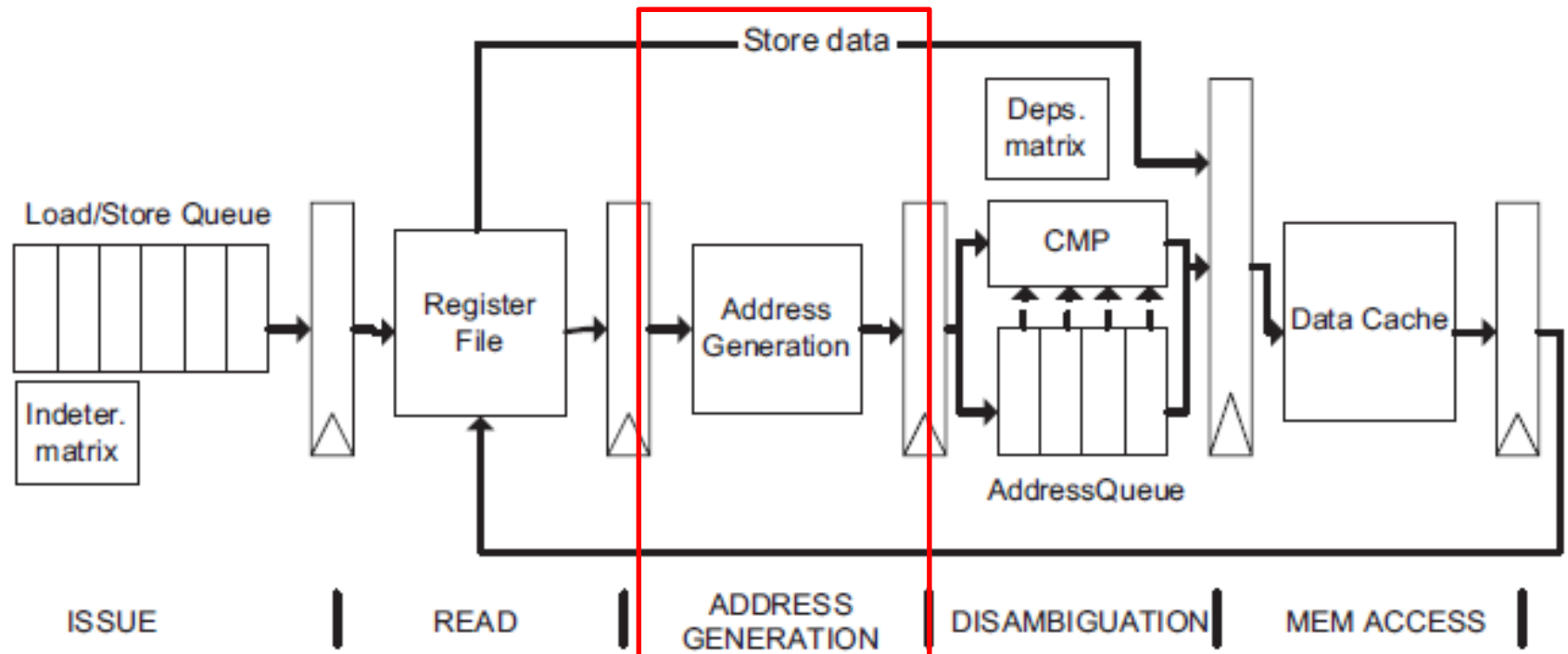
MIPS R10000 — Execution(5/9)

- Read the source operands from the register file.



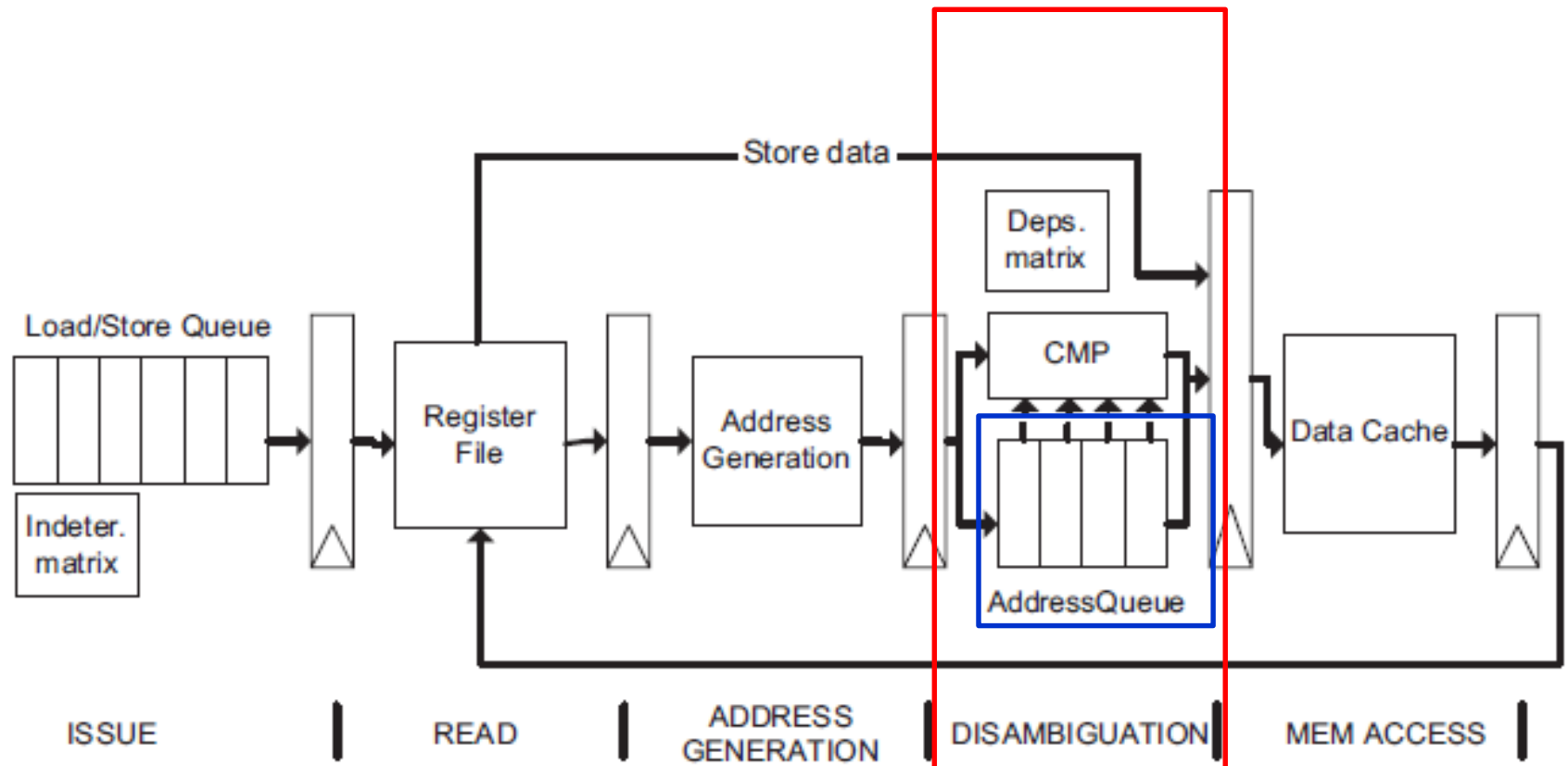
MIPS R10000 — Execution(6/9)

- Read the source operands from the register file.



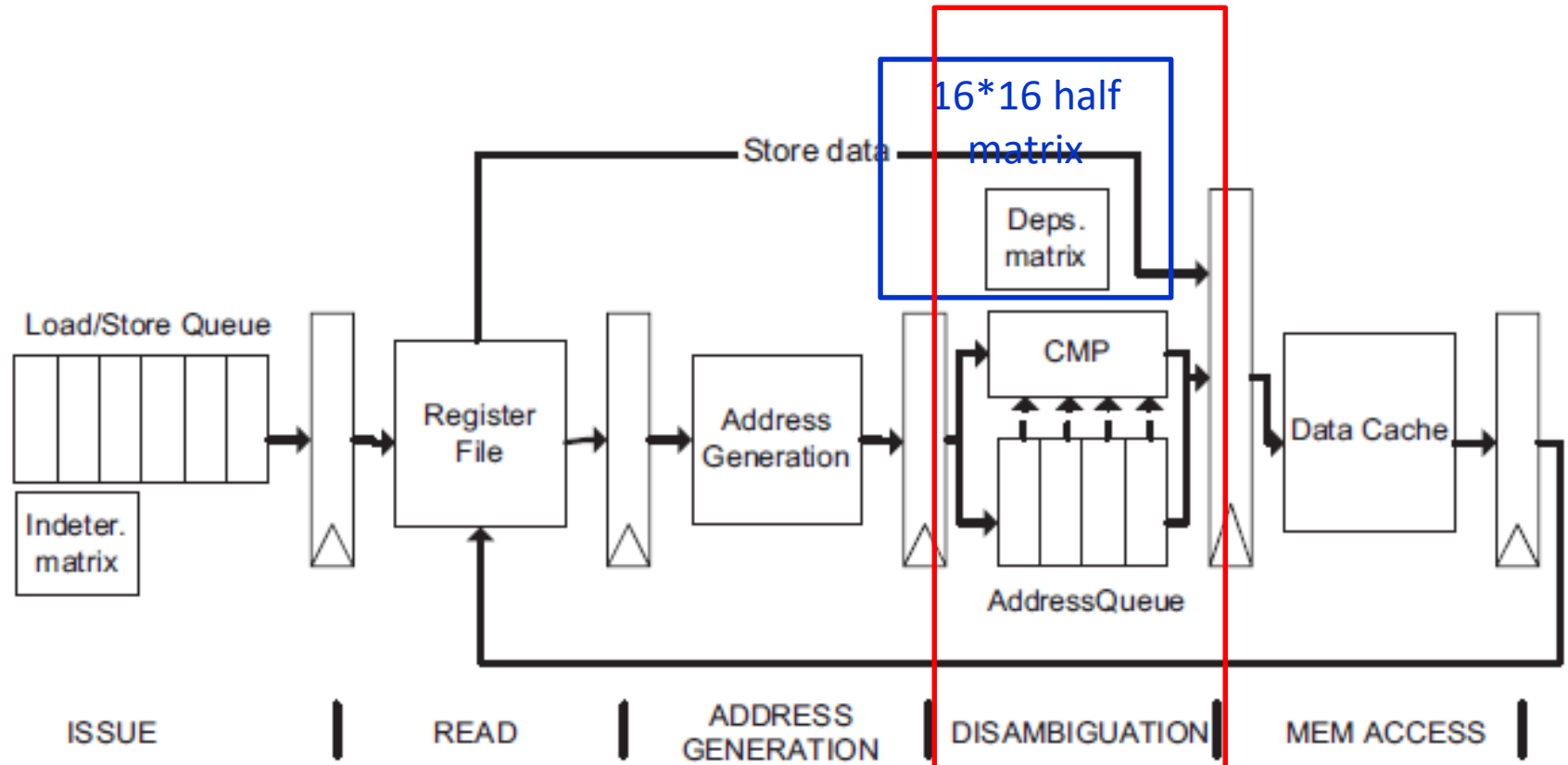
MIPS R10000 — Execution(7/9)

- Keeps the memory address of loads and stores that want to access the cache.



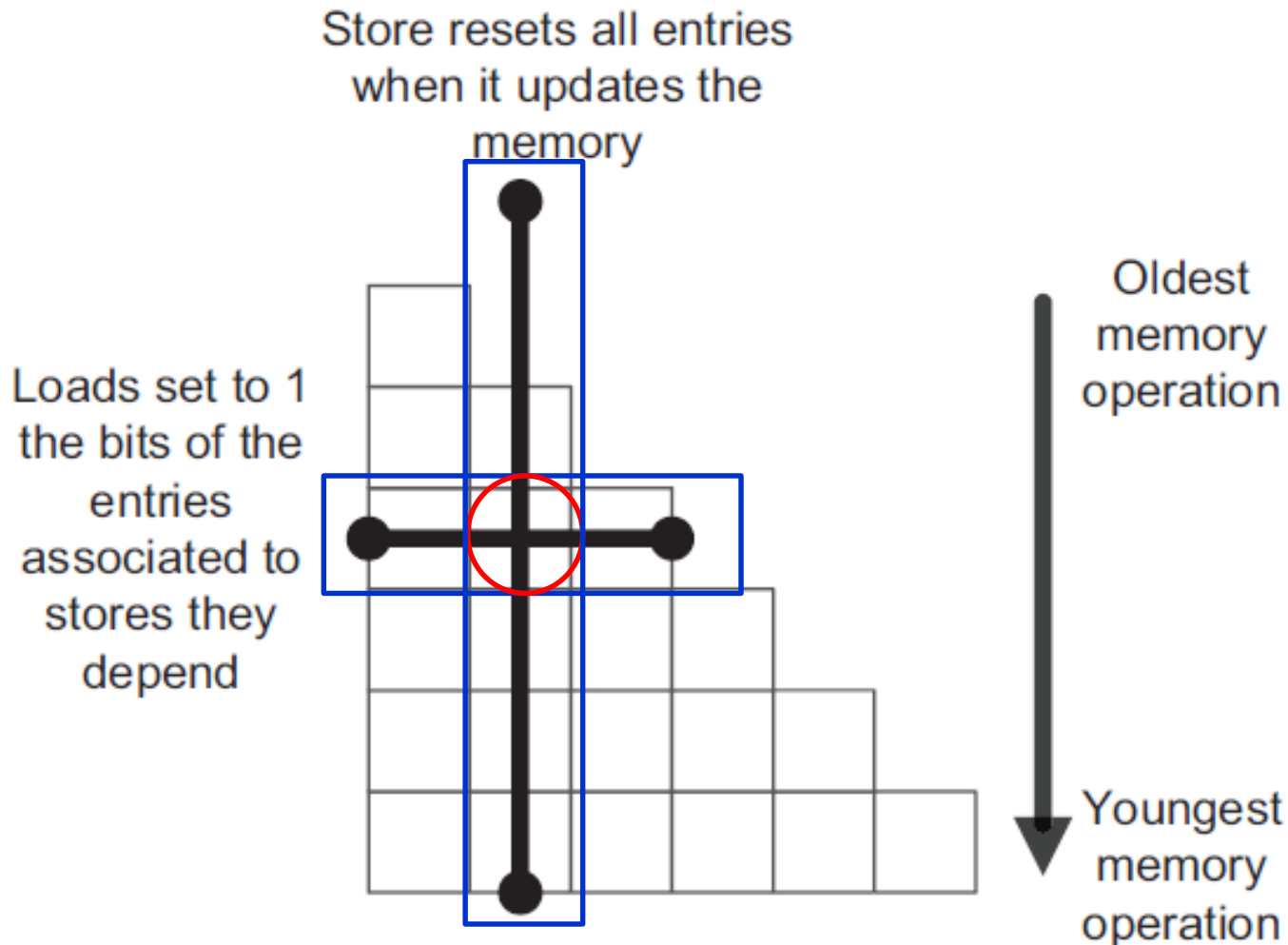
MIPS R10000 — Execution(8/9)

- Every column and row represents an entry on the load/Store queue.



MIPS R10000 — Execution(9/9)

- Example of a 6-entry dependency matrix.





Contents

- Introduction(to issue logic for memory operations)
- **Memory Disambiguation**
 - Non-speculative Memory Disambiguation
 - **Speculative Memory Disambiguation**
- Speculative Wakeup of Load Consumer
- Memory Disambiguation on UniCore-3

3 Speculative Memory Disambiguation

- Introduction
- Example: Alpha 21264

Introduction

- To make sure the program is right executed, memory dependence checking is required.
 - Memory dependence checking can become quite complex if a large number of load/store instructions are involved.
- Processors using speculative memory disambiguation technology boost performance by **speculative issuing loads** that are predicted not to be dependent on any previous in-flight store.
 - load operation do not have to wait for all previous stores to compute its address.
 - require special hardware in order to identify mispredictions and recover the execution.

Alpha 21264—Overview

- Alpha 21264 is a super-scalar microprocessor with out-of-order and speculative execution.
- high level overview of the 21264 pipeline

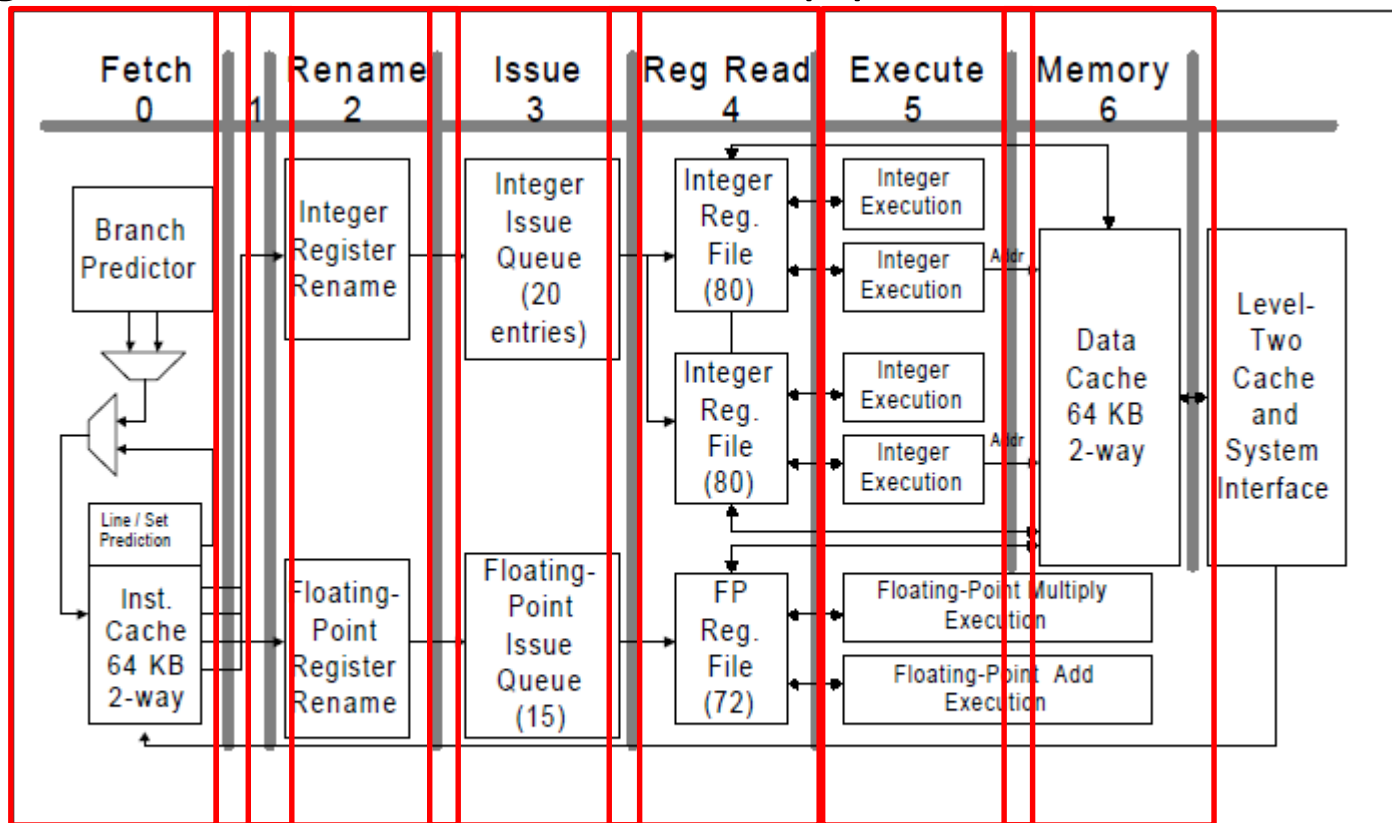
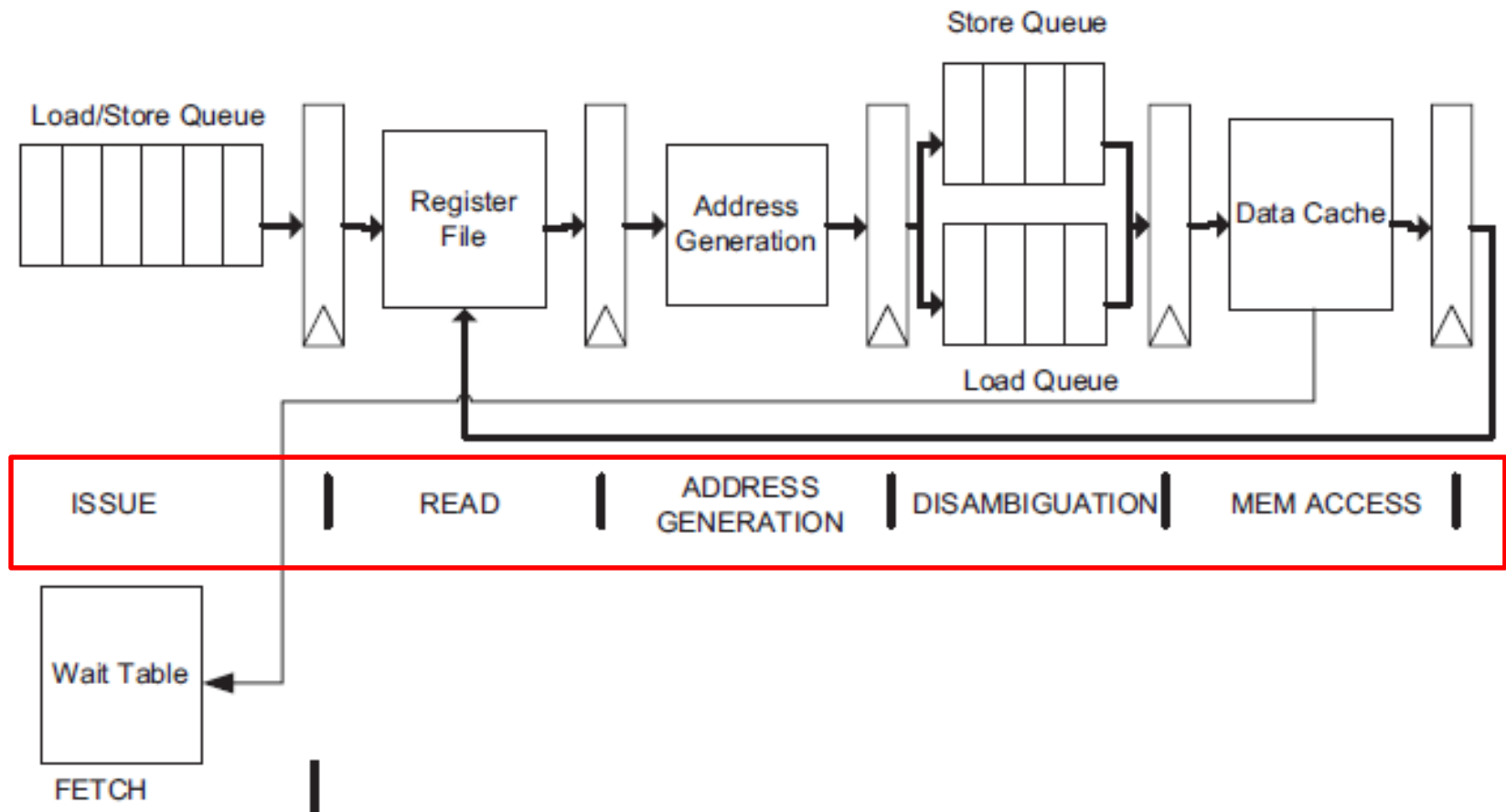
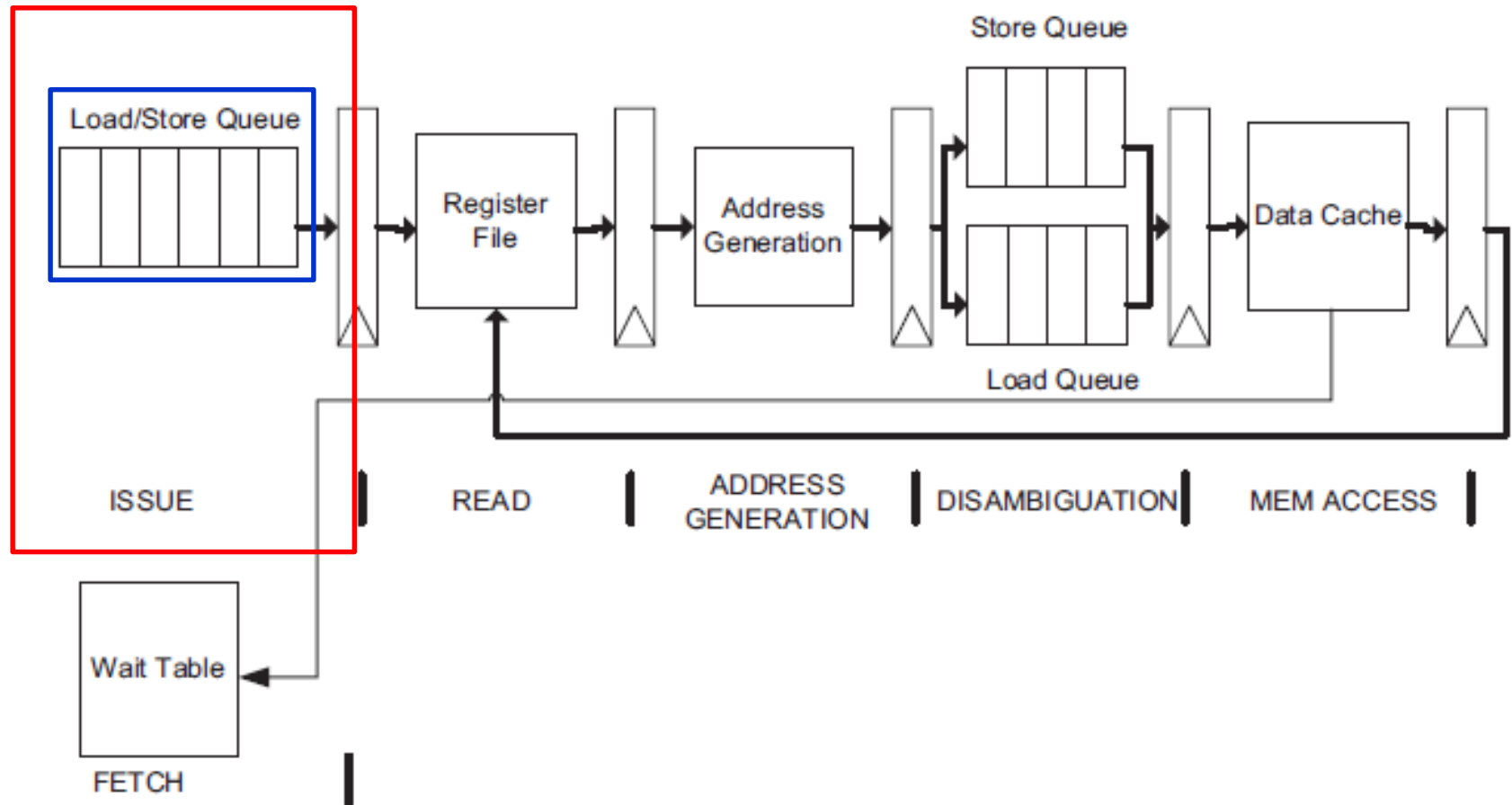


Figure 1 The Basic 21264 Pipeline

Alpha 21264 — Execution




Alpha 21264 — Components(1/4)



Alpha 21264 — Execution

- One load may be mis-speculated for many times.

```
Loop:
    std r2, [r3+], #4
    ....
    ....
    ldd r1, [r3+], #4
    b    Loop
```

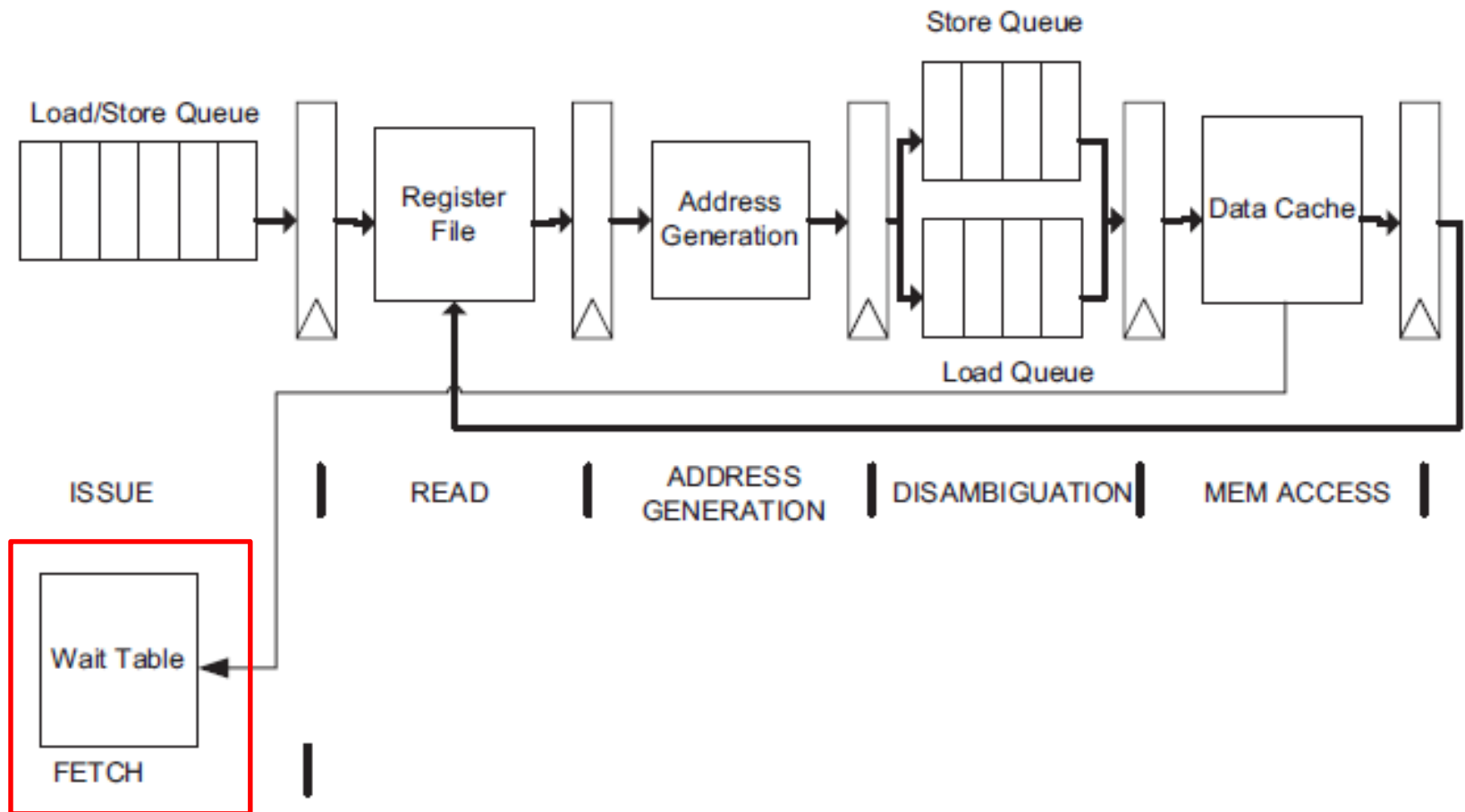


A blue bracket on the right side of the code block groups the store instruction (`std r2, [r3+], #4`) and the load instruction (`ldd r1, [r3+], #4`). To the right of the bracket, the text **RAW** is written in red, indicating a Read-After-Write hazard.

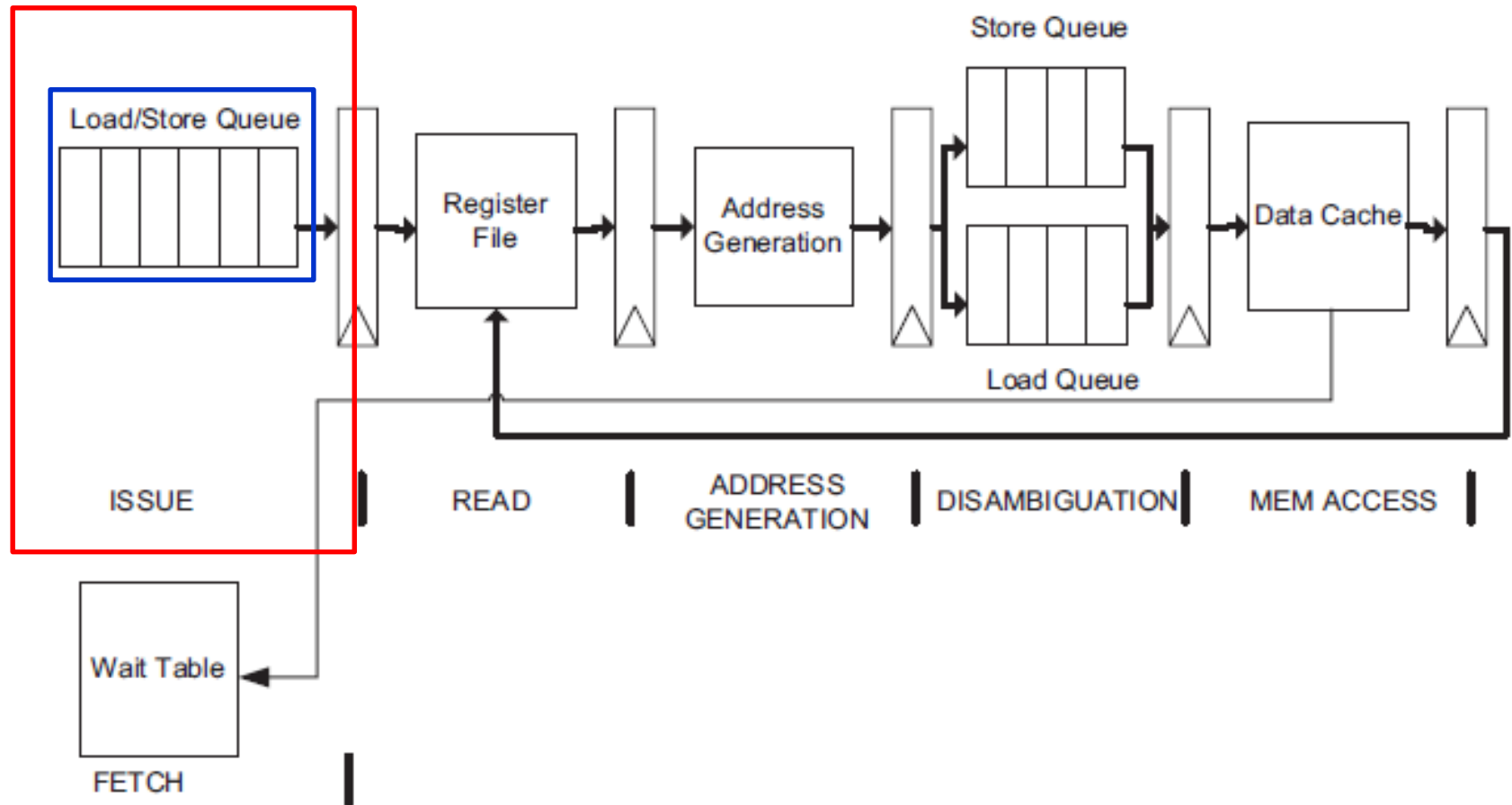
- Use a 'load wait table' to avoid subsequent executions' mis-speculations of the same load.

Alpha 21264 — Components(4/4)

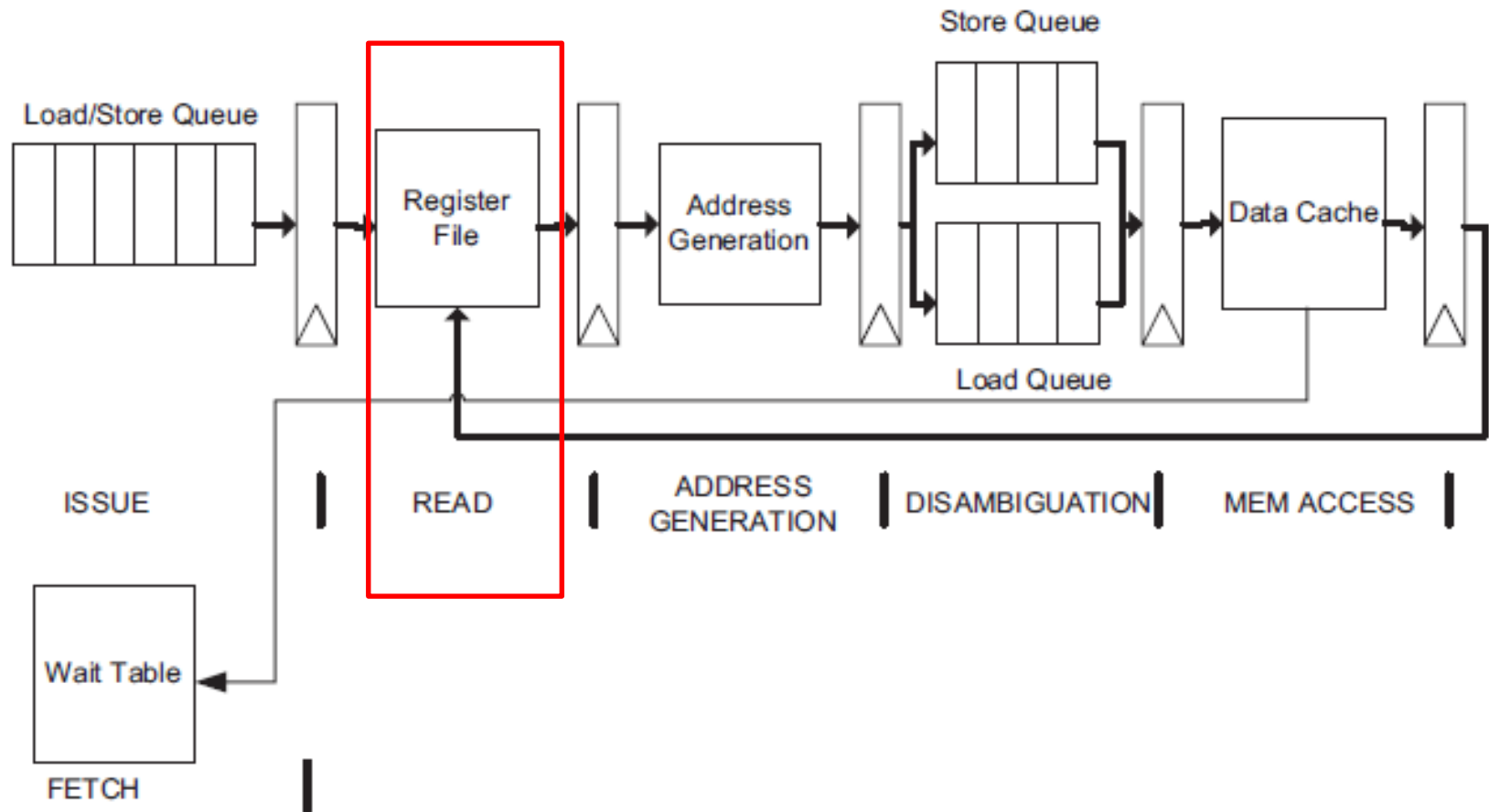
- Used to record a store-load trap.
- 1024 entries of 1bit, indexed by virtual PC.



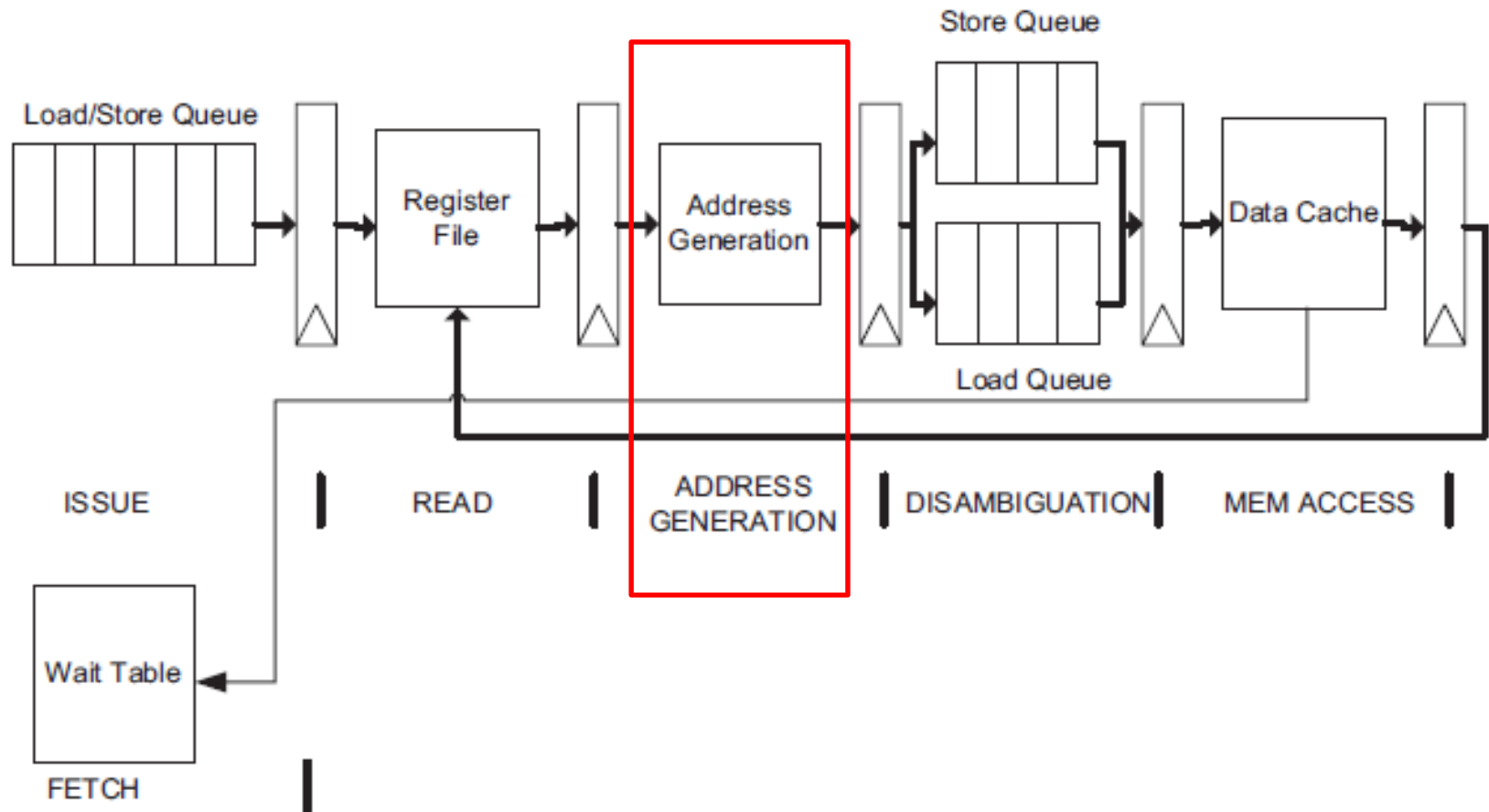
Alpha 21264 — Components(1/4)



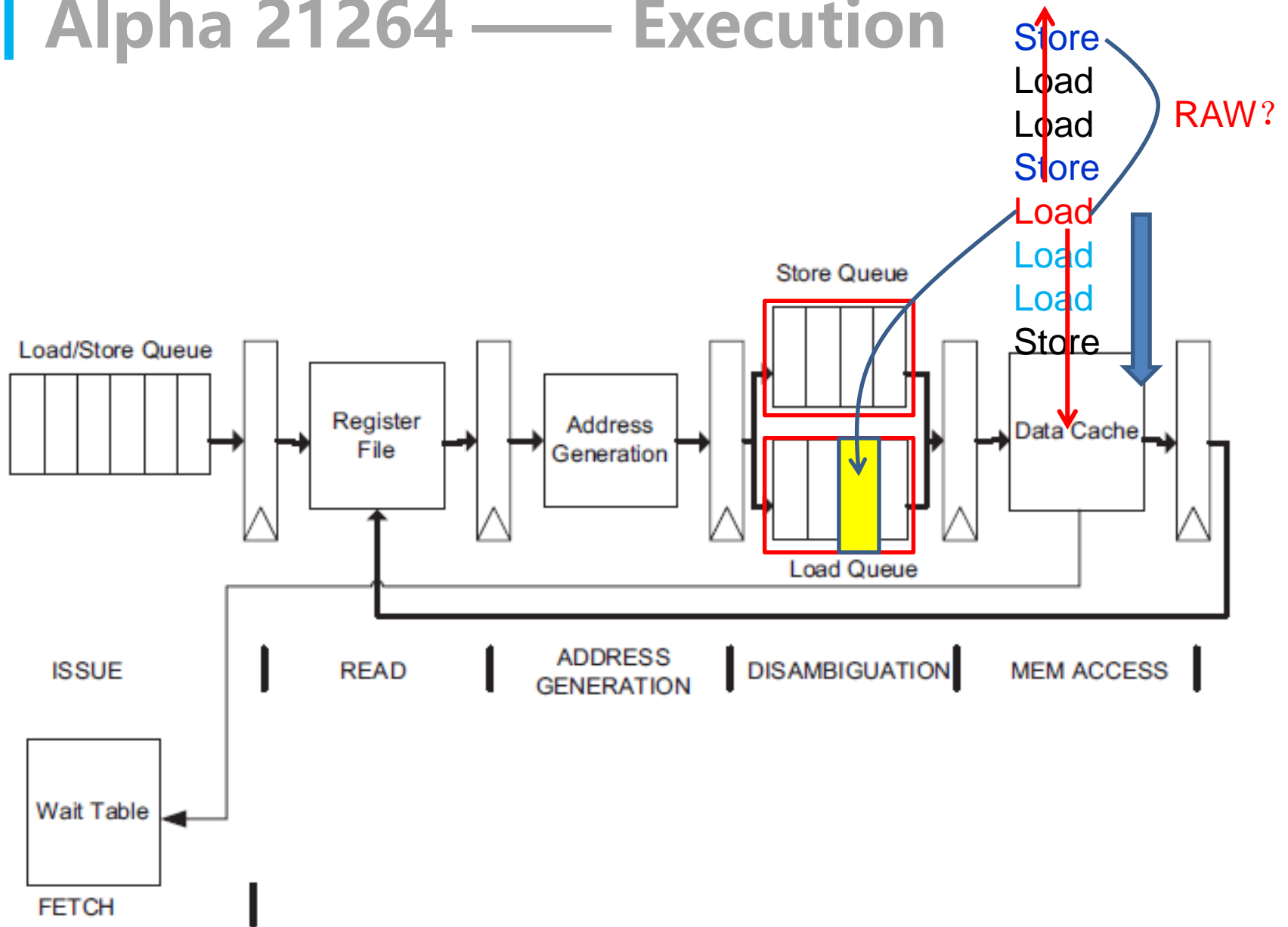
Alpha 21264 — Components(1/4)



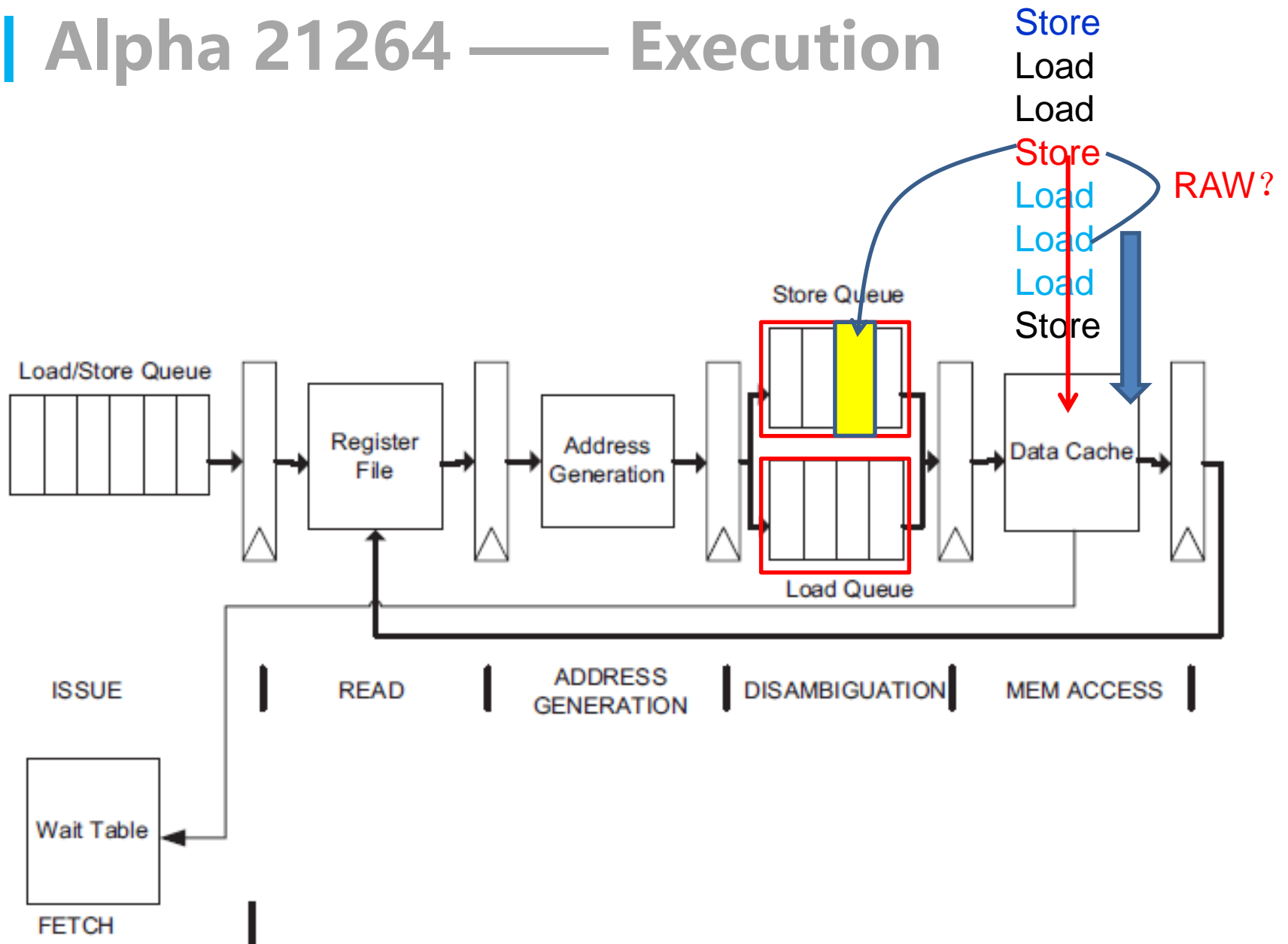
Alpha 21264 — Components(1/4)



Alpha 21264 — Execution



Alpha 21264 — Execution





Contents

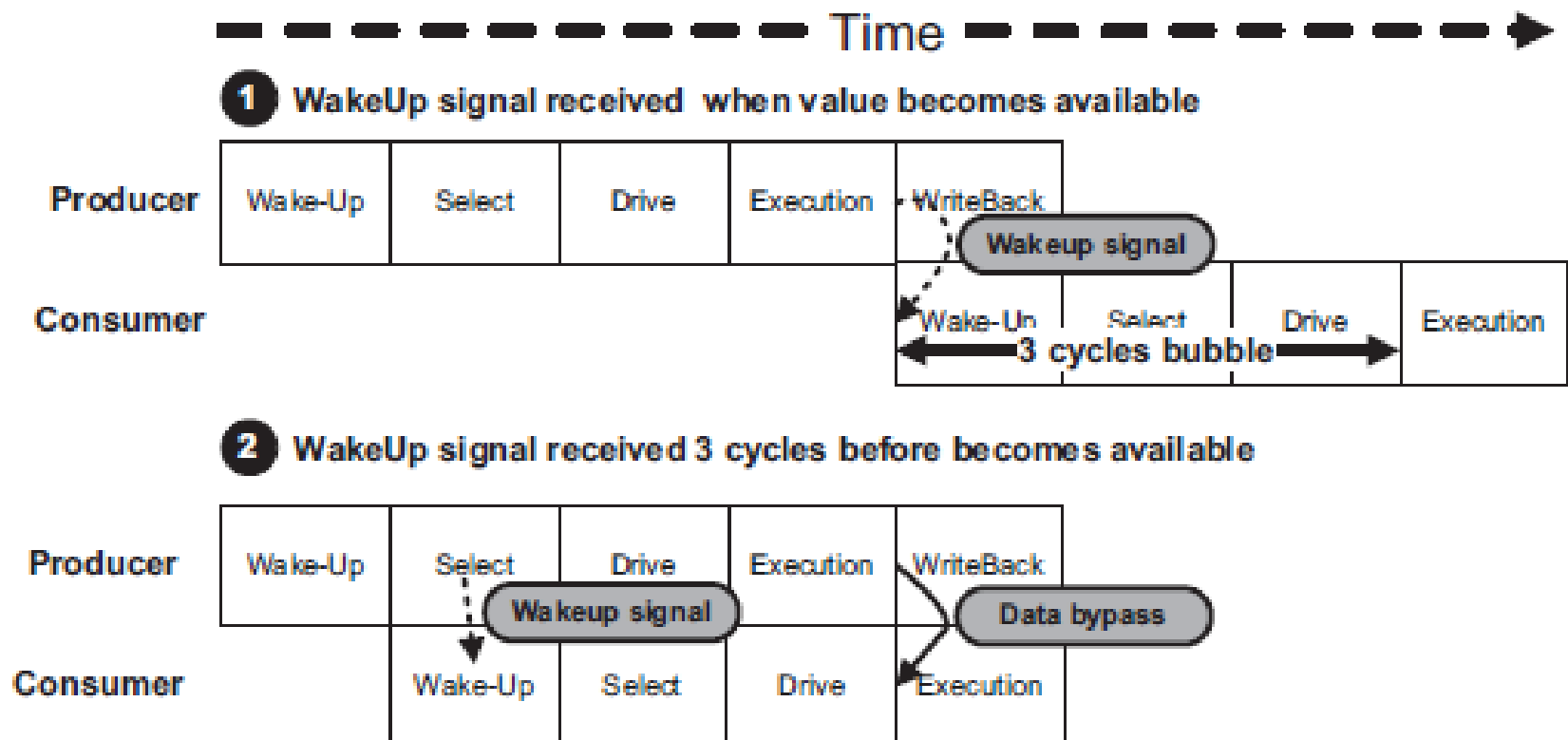
- Introduction(to issue logic for memory operations)
- Memory Disambiguation
 - Non-speculative Memory Disambiguation
 - Speculative Memory Disambiguation
- Speculative Wakeup of Load Consumer
- Memory Disambiguation on UniCore-3

Speculative wakeup of load consumers(1/5)

- **Instruction Wakeup**
 - Wakeup is the event that notifies that one of the source operands has been produced.
 - After the wakeup signal generated, some other instructions which use the source operands produced can be issued.

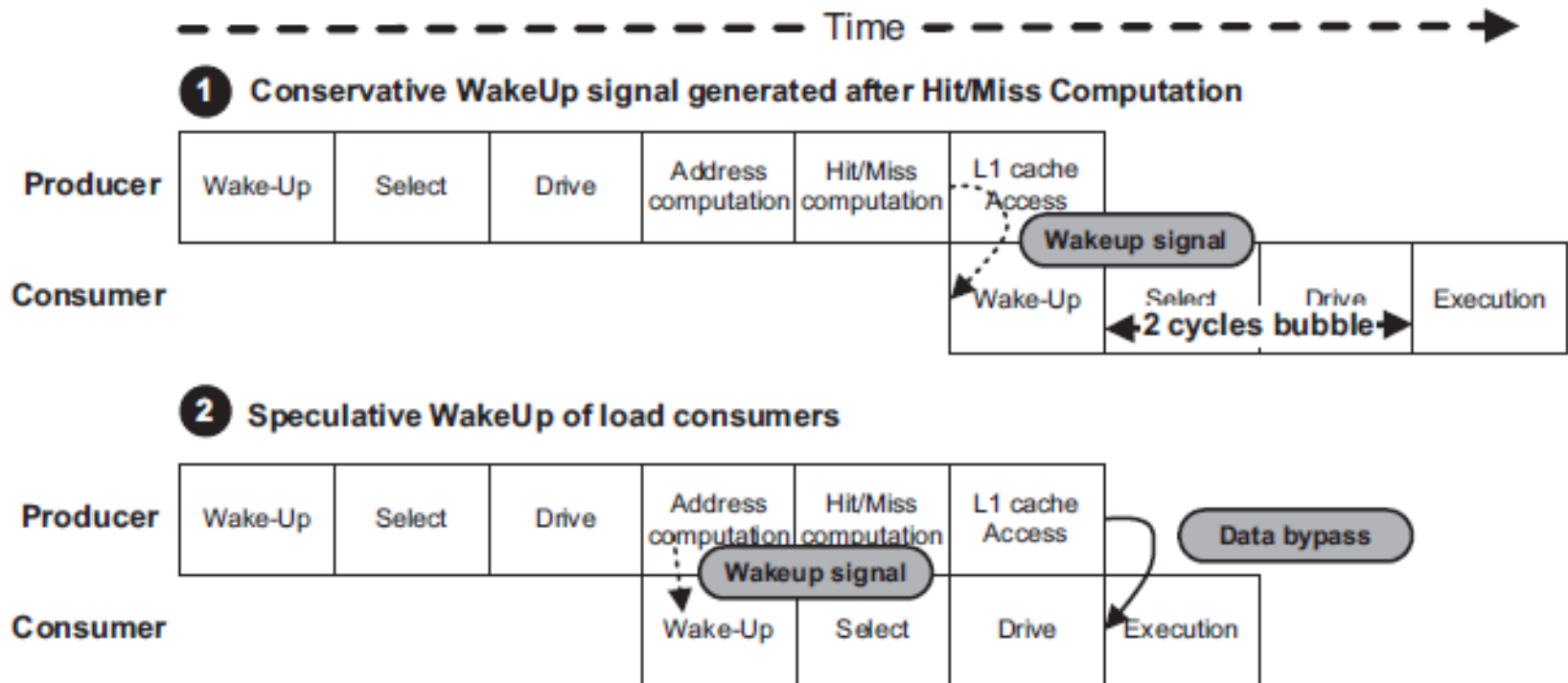
Speculative wakeup of load consumers(2/5)

- Normal operation wakeup



Speculative wakeup of load consumers(3/5)

- Load operation wakeup
 - hit the TLB and data cache
 - miss on the TLB and data cache



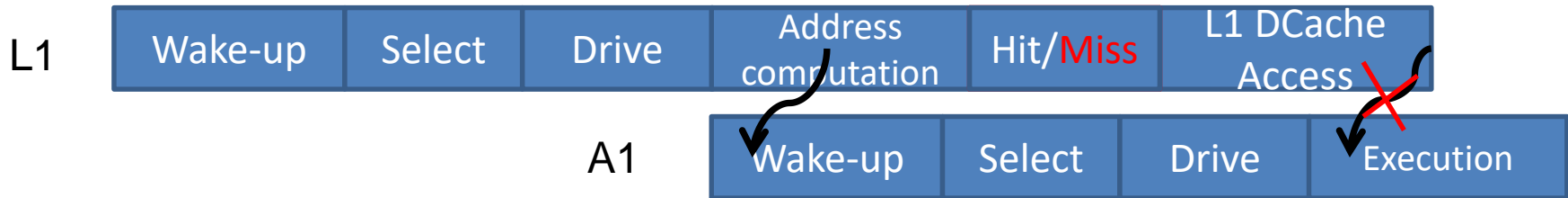
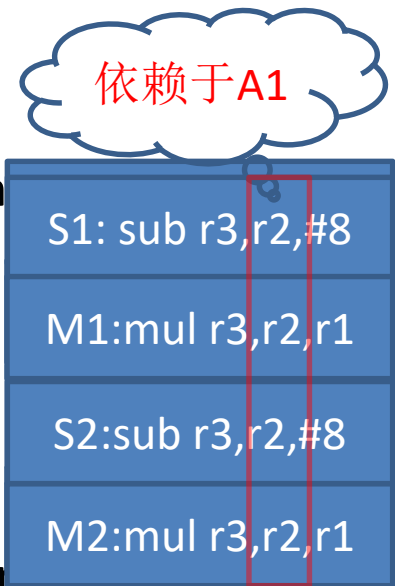
Speculative wakeup of load consumers(4/5)

- **Scenario 1:**

- Obtain a two-cycle bubble between producer and consumer
- do not need back-to-back execution.

- **Scenario 2:**

- A load may miss in cache or its execution is delayed for some reason.
- As a result, the consumers will have to be cancelled and re-executed again.



Speculative wakeup of load consumers(5/5)

- **Solutions to avoid this deadlock**
 - flush all instructions in the pipeline younger than the one be reissued and resume execution from there.
 - ✓ utilization of issue queue entries.
 - ✓ if this situation occurs very often, this scheme may cause significant performance drop.
 - postpone the reclamation of the issue entry allocated by an instruction until we are sure this instruction would not have to be reissued.
 - ✓ reduces the penalty of reissuing instructions compared to the previous one.
 - ✓ Increase the pressure on the issue queue.



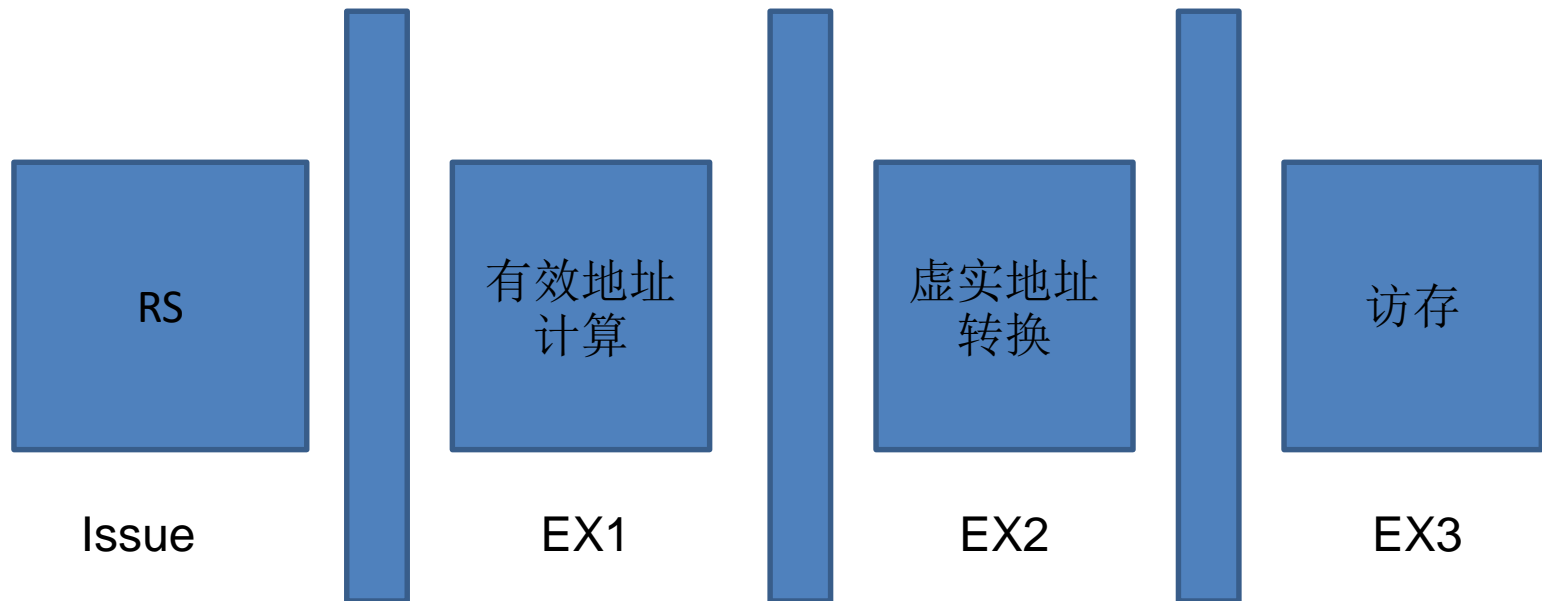
Contents

- Introduction(to issue logic for memory operations)
- Memory Disambiguation
 - Non-speculative Memory Disambiguation
 - Speculative Memory Disambiguation
- Speculative Wakeup of Load Consumer
- Memory Disambiguation on UniCore-3

Memory Disambiguation on UniCore-3

- UniCore-3 processor implements partial ordering.
- Loads can be executed out of order as long as all previous memory operation have computed their address.
- Stores are processed in strict program order.

UniCore-3——Pipeline



回填Cache

开始

一致性操作

Issue

新指令

EX1
(地址计算)

EX1

新指令

Arbiter

仲裁成功的指令

提交后

Replay
Buffer

重执行条件满足
DMMU未就绪

EX2
(地址转换)

写指令Finish

Store
Queue

EX2

重执行条件满足

EX3
(命中判断)

读指令Miss

Load
Miss
Queue

数据取回后

EX3

读写指令Miss

命中

结束

Data Line Fill Buffer

L1DCache

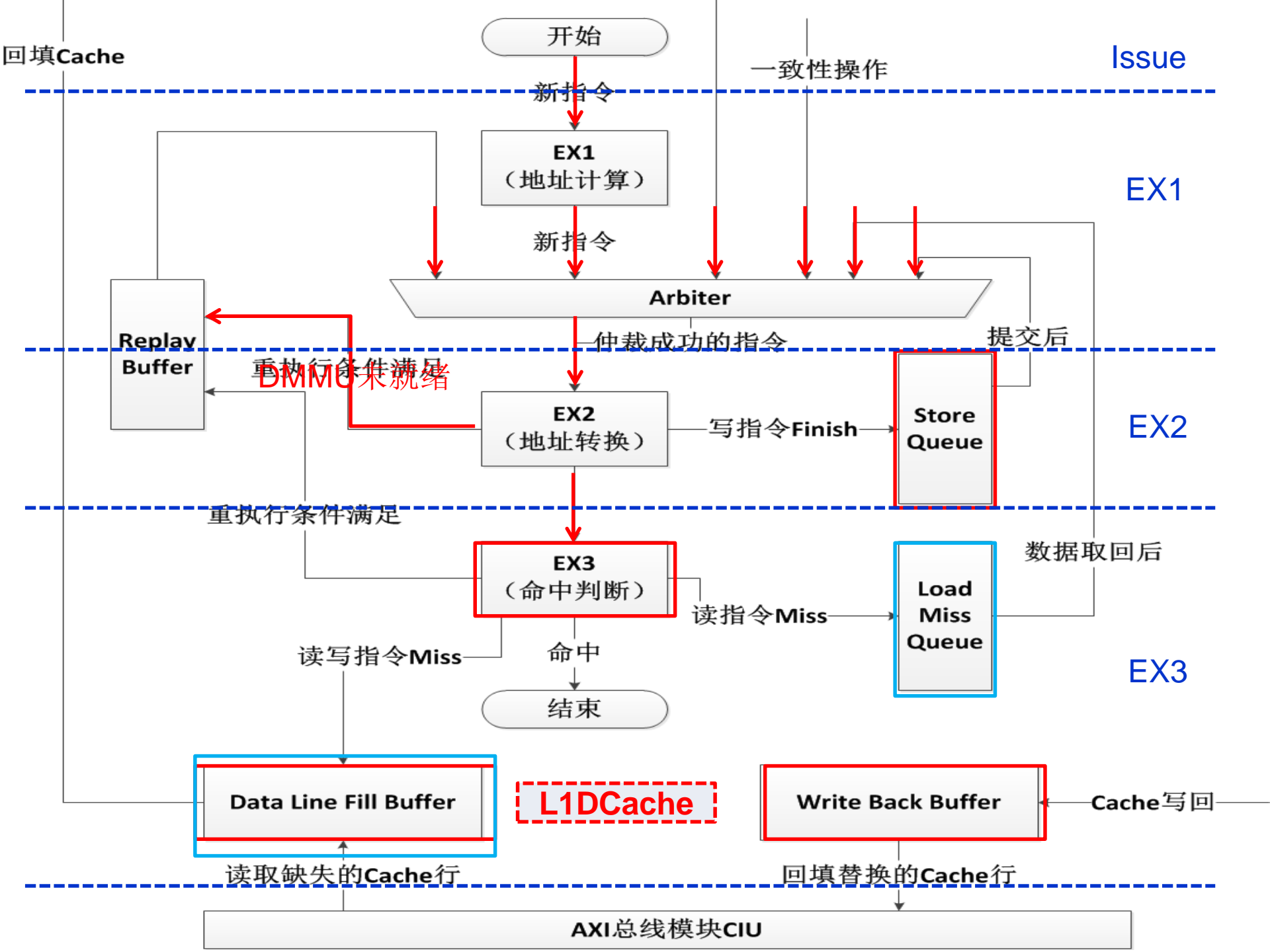
Write Back Buffer

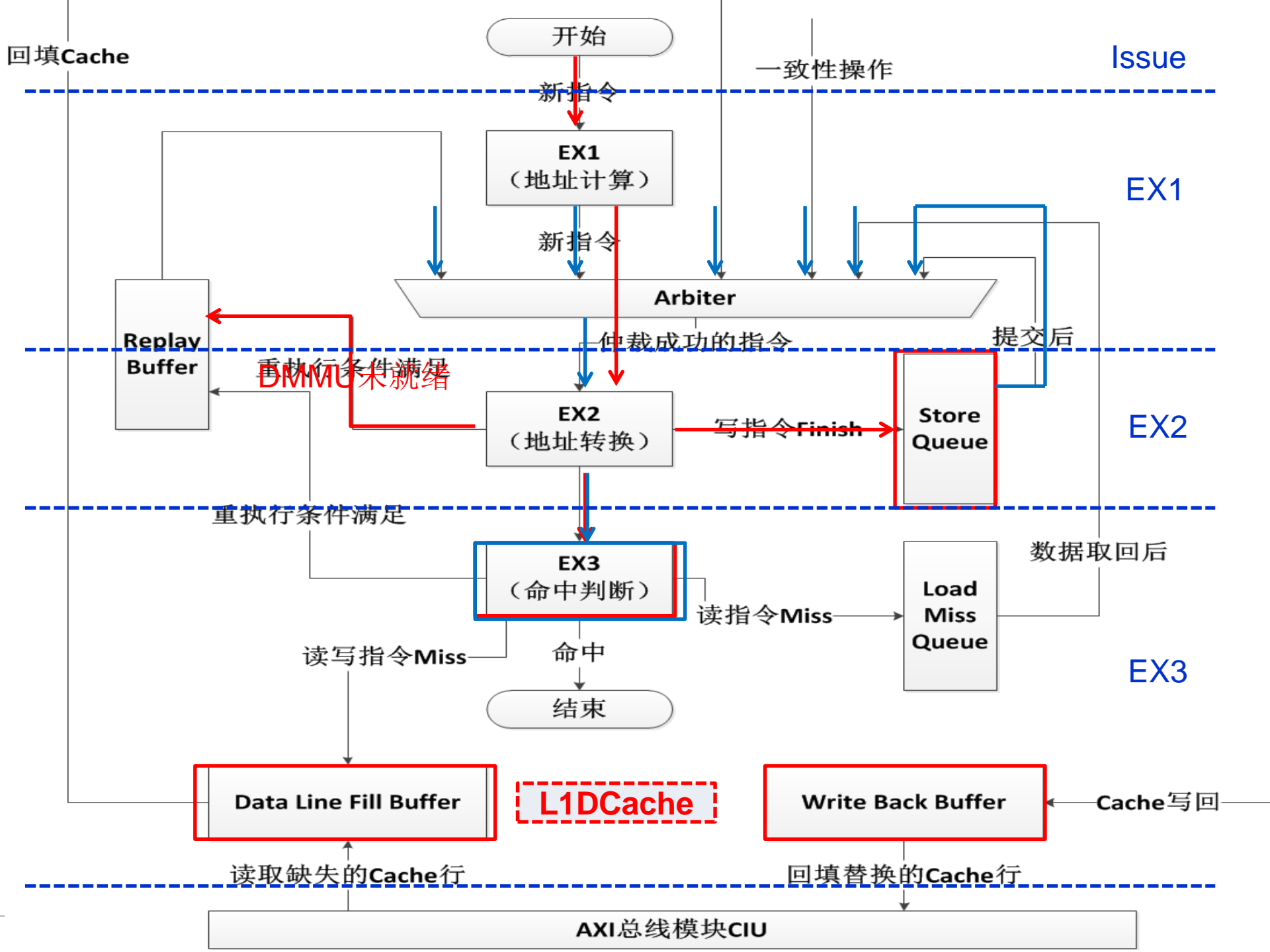
Cache写回

读取缺失的Cache行

回填替换的Cache行

AXI总线模块CIU





instruction sequence of Load/Store

Address	Load/Store	Order
Same Address	Load , Load	In Order
	Store , Load	
	Load , Store	
	Store, Store	
Different Address	Load , Load	Out of order
	Store , Load	Out of order
	Load , Store	In order
	Store, Store	In order

instruction sequence of Load/Store

- Load , Load
- Store , Load
- Load , Store
- Store , Store

回填Cache

开始

一致性操作

Issue

新指令

EX1
(地址计算)

EX1

新指令

Arbiter

仲裁成功的指令

提交后

Replay
Buffer

重执行条件满足

EX2
(地址转换)

写指令Finish

Store
Queue

EX2

Load 2

重执行条件满足

Load 1

EX3
(命中判断)

读指令Miss

Load
Miss
Queue

数据取回后

EX3

读写指令Miss

命中

结束

Miss

Data Line Fill Buffer

L1DCache

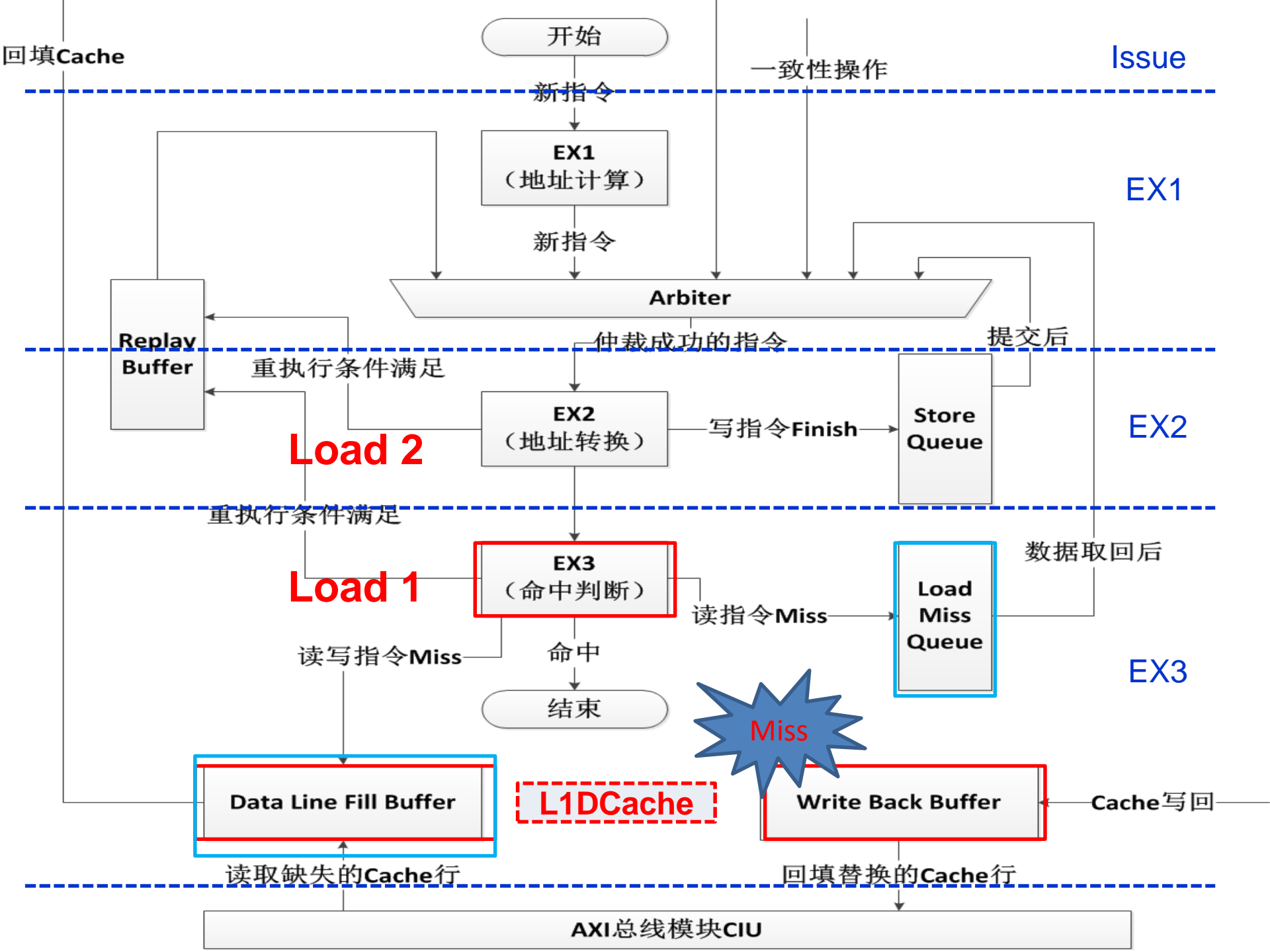
Write Back Buffer

Cache写回

读取缺失的Cache行

回填替换的Cache行

AXI总线模块CIU



回填Cache

开始

一致性操作

Issue

新指令

EX1
(地址计算)

EX1

新指令

Arbiter

仲裁成功的指令

提交后

Replay
Buffer

重执行条件满足

EX2
(地址转换)

写指令Finish

Store
Queue

EX2

重执行条件满足

Load 2

EX3
(命中判断)

读指令Miss

Load
Miss
Queue

数据取回后

EX3

读写指令Miss

命中

结束

Hit

Data Line Fill Buffer

L1DCache

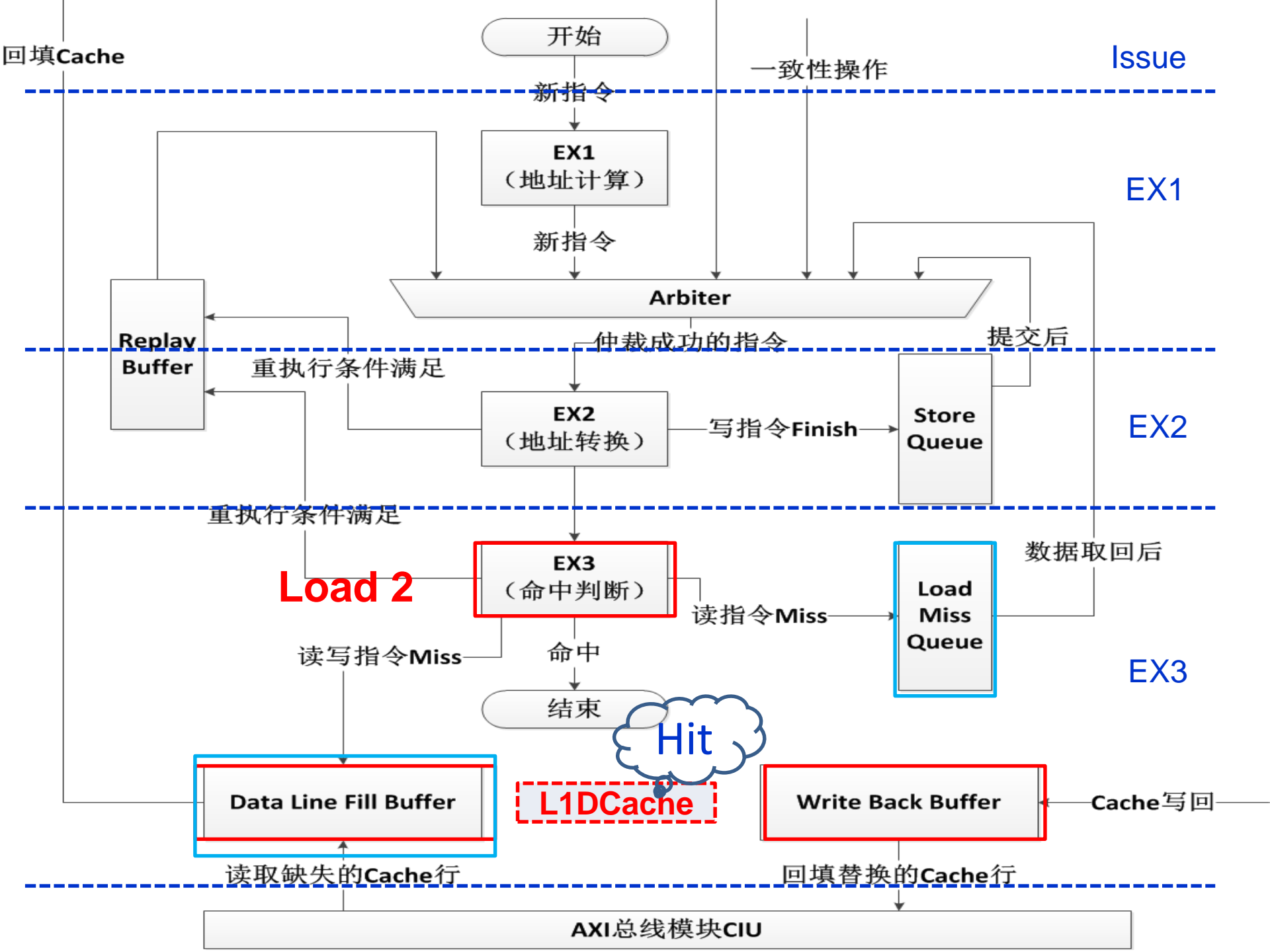
Write Back Buffer

Cache写回

读取缺失的Cache行

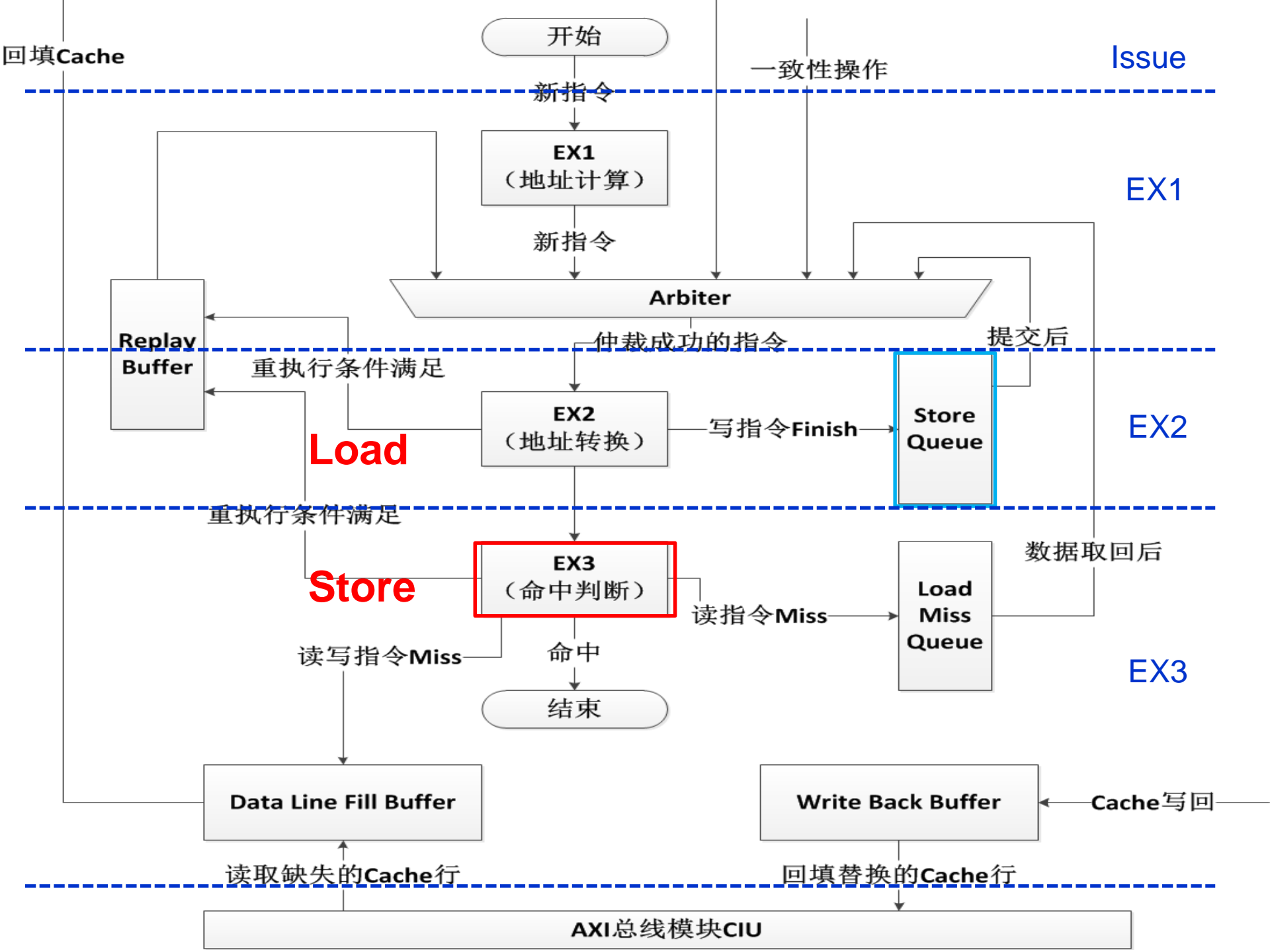
回填替换的Cache行

AXI总线模块CIU



instruction sequence of Load/Store

- Load , Load
- Store , Load
- Load , Store
- Store , Store



回填Cache

开始

一致性操作

Issue

新指令

EX1
(地址计算)

EX1

新指令

Arbiter

! raw hit

提交后

仲裁成功的指令

Replay
Buffer

重执行条件满足

EX2
(地址转换)

写指令Finish

Store
Queue

EX2

重执行条件满足

Load

EX3
(命中判断)

读指令Miss

Load
Miss
Queue

数据取回后

EX3

读写指令Miss

命中

结束

Hit

Data Line Fill Buffer

L1DCache

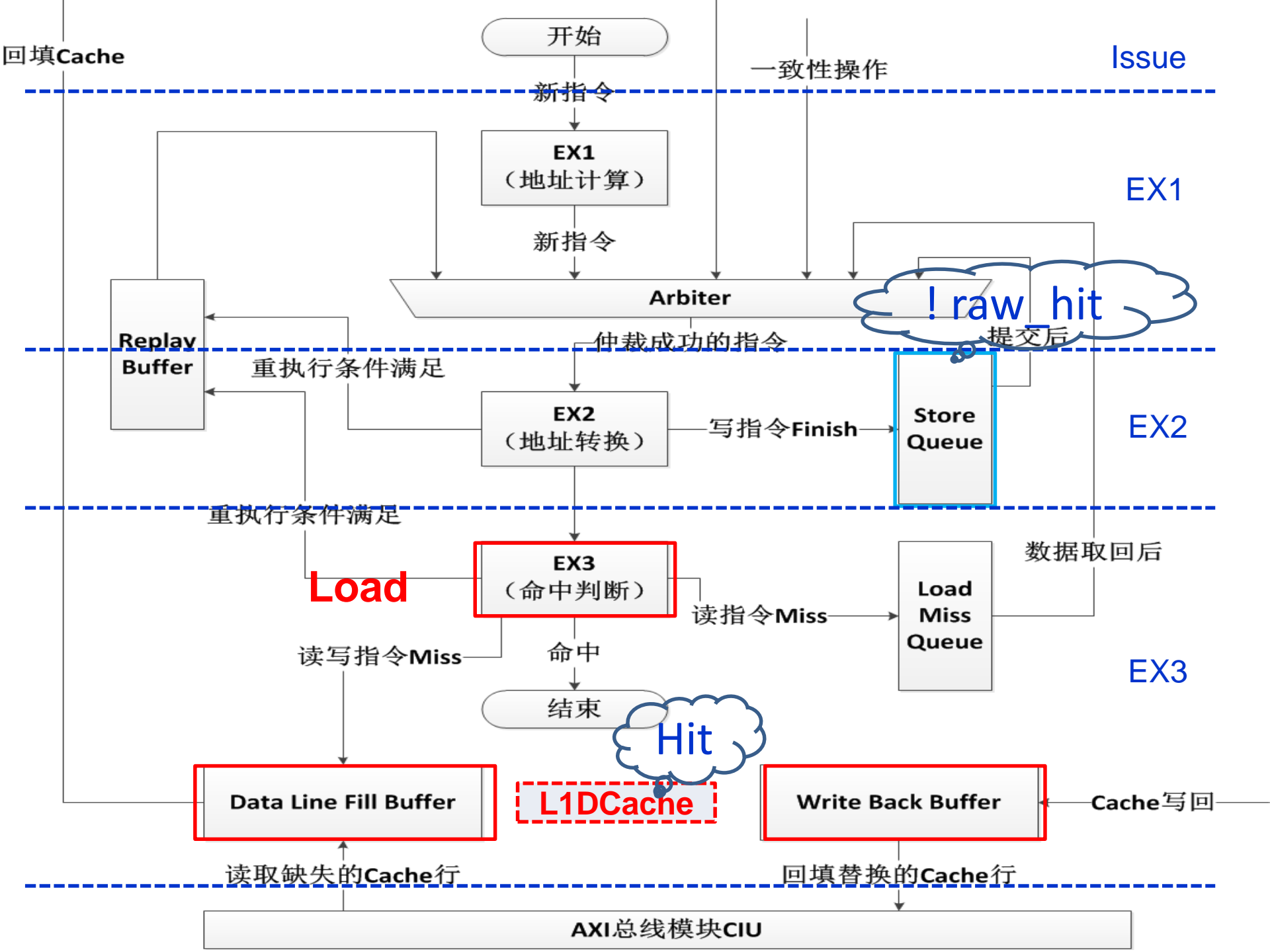
Write Back Buffer

Cache写回

读取缺失的Cache行

回填替换的Cache行

AXI总线模块CIU



instruction sequence of Load/Store

- Load , Load
- Store , Load
- Load , Store
- Store , Store

回填Cache

开始

一致性操作

Issue

新指令

EX1
(地址计算)

EX1

新指令

Arbiter

仲裁成功的指令

提交后

Replay
Buffer

重执行条件满足

EX2
(地址转换)

写指令Finish

Store
Queue

EX2

Store

重执行条件满足

Load

EX3
(命中判断)

读指令Miss

Load
Miss
Queue

数据取回后

EX3

读写指令Miss

命中

结束

Miss

Data Line Fill Buffer

Write Back Buffer

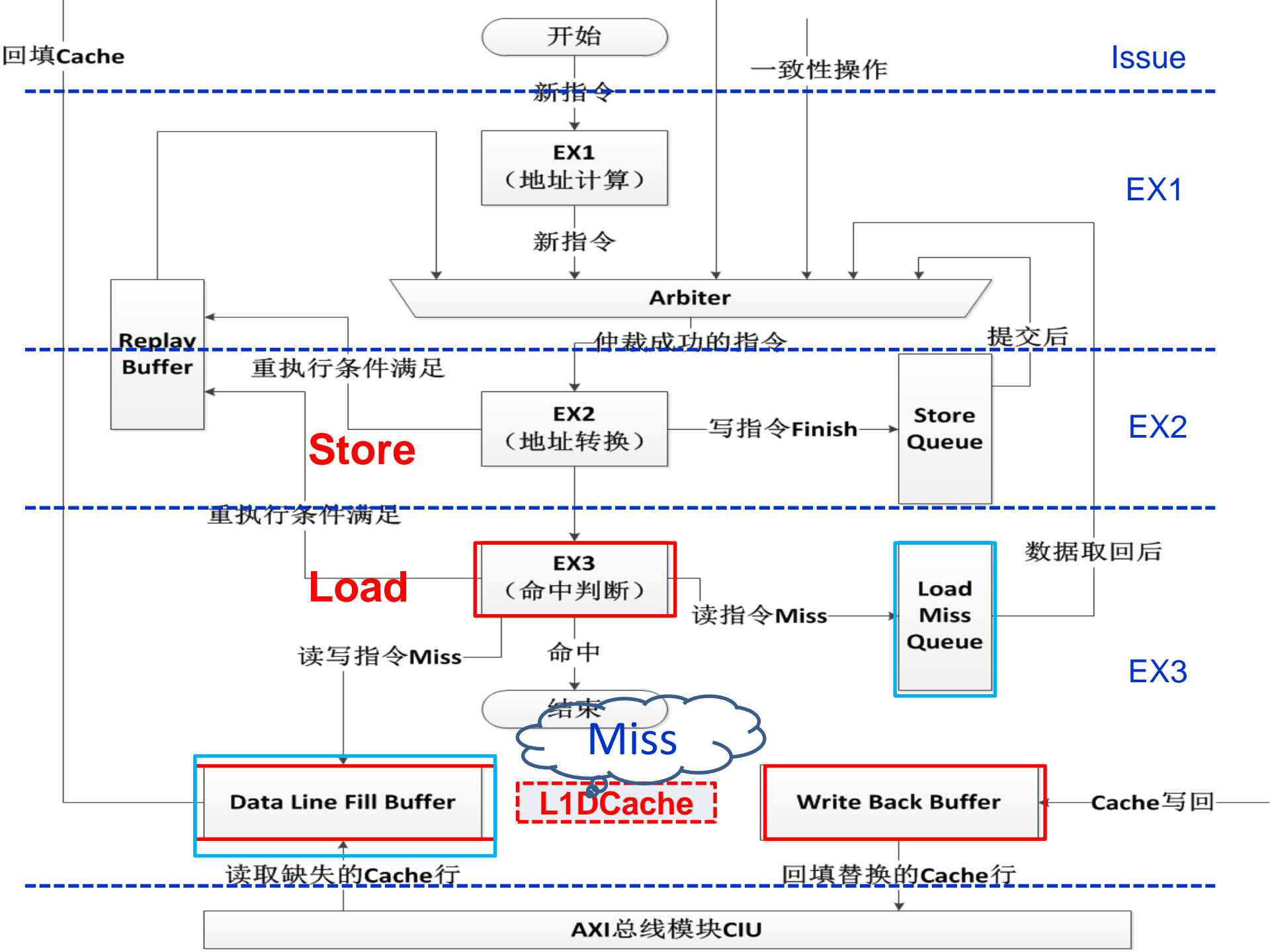
Cache写回

L1DCache

读取缺失的Cache行

回填替换的Cache行

AXI总线模块CIU



回填Cache

开始

一致性操作

Issue

新指令

EX1
(地址计算)

EX1

新指令

Arbiter

仲裁成功的指令

提交后

Replay
Buffer

重执行条件满足

EX2
(地址转换)

写指令Finish

Store
Queue

EX2

重执行条件满足

Store

EX3
(命中判断)

读指令Miss

Load
Miss
Queue

数据取回后

EX3

读写指令Miss

命中

结束

Data Line Fill Buffer

L1DCache

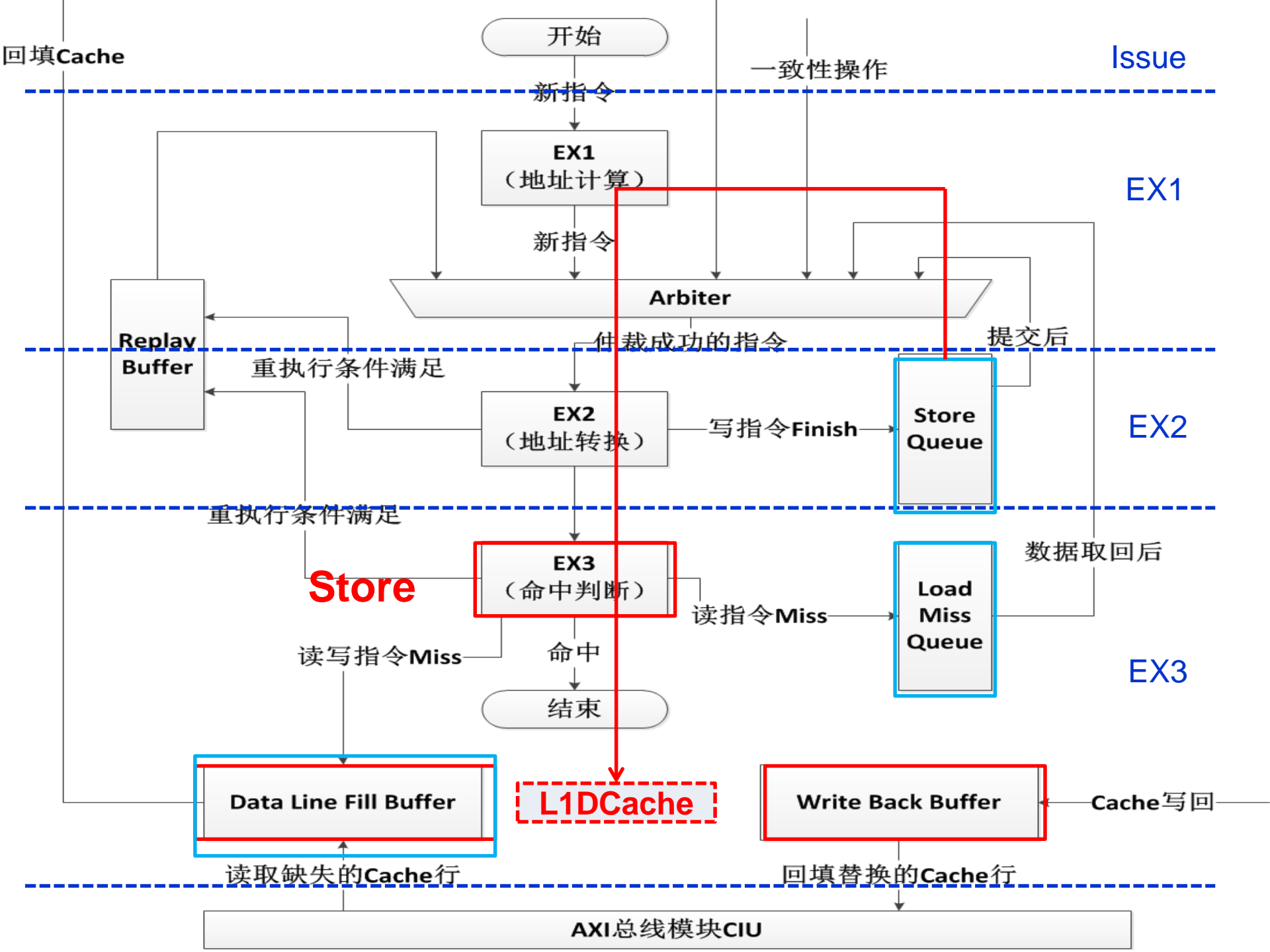
Write Back Buffer

Cache写回

读取缺失的Cache行

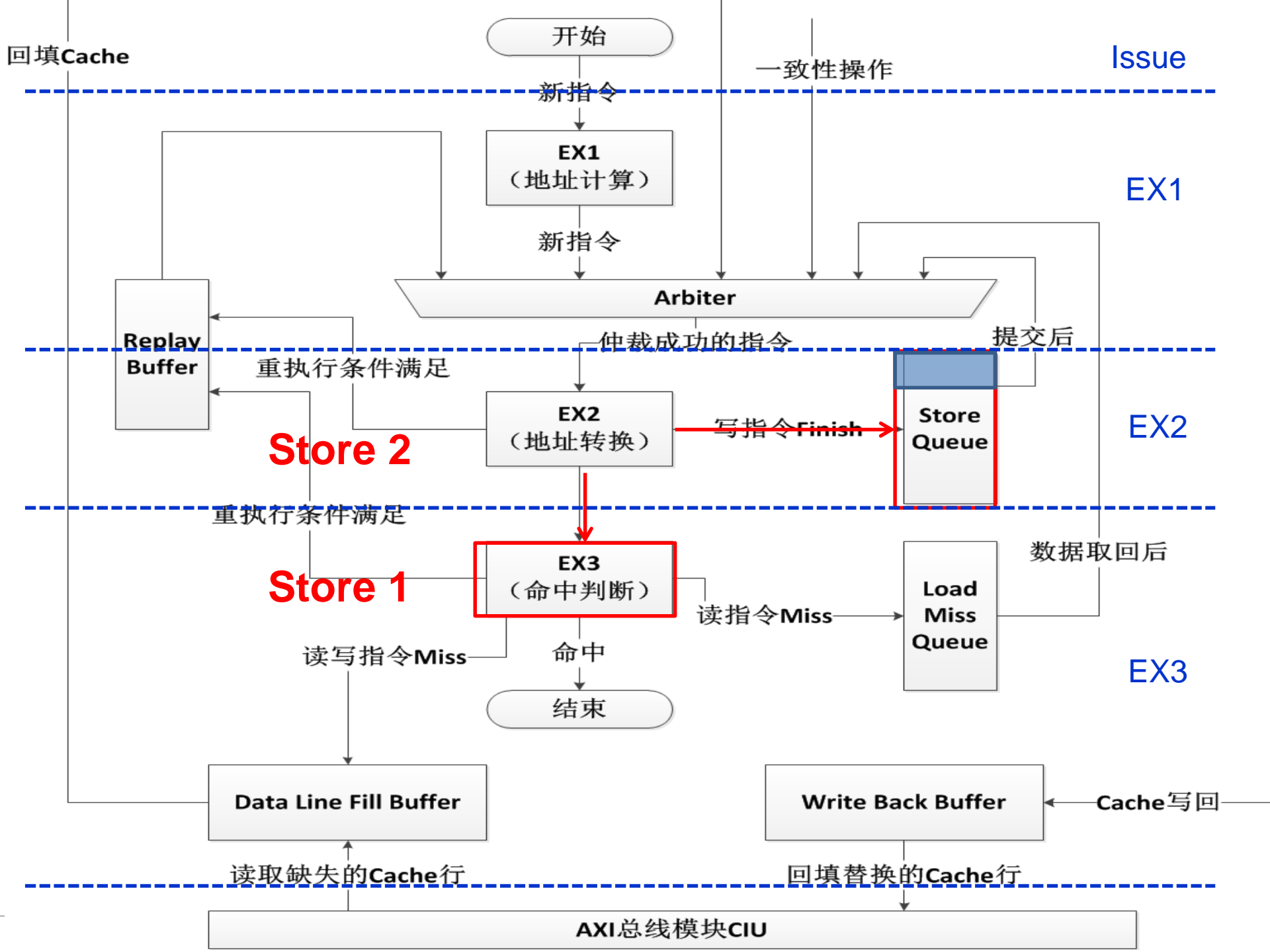
回填替换的Cache行

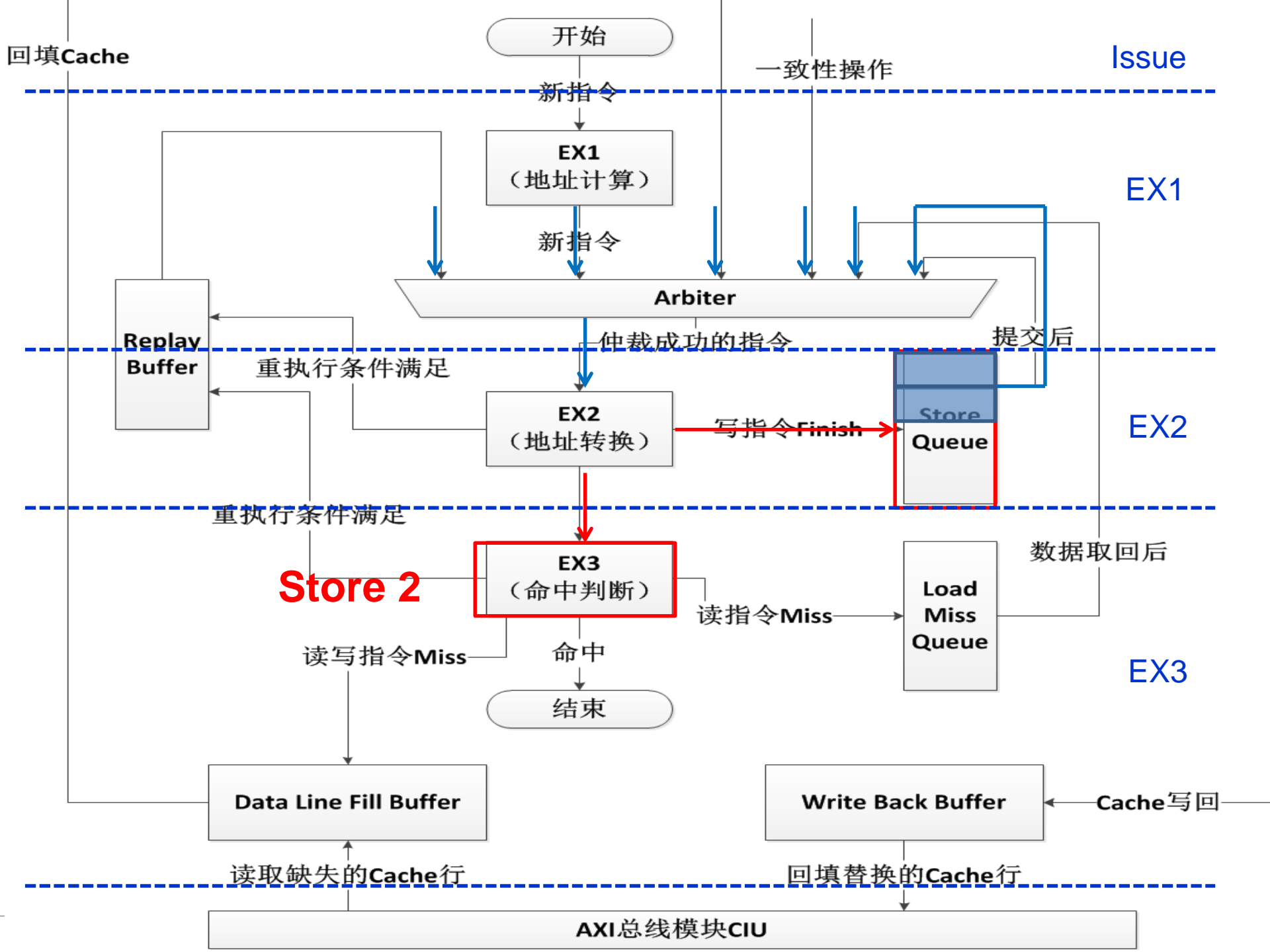
AXI总线模块CIU



instruction sequence of Load/Store

- Load , Load
- Store , Load
- Load , Store
- Store , Store







Thank You