Lecture 13:
CPU – Load & Store Queue

Hunjun Lee
hunjunlee@hanyang.ac.kr

Remember: There are three pipeline hazards

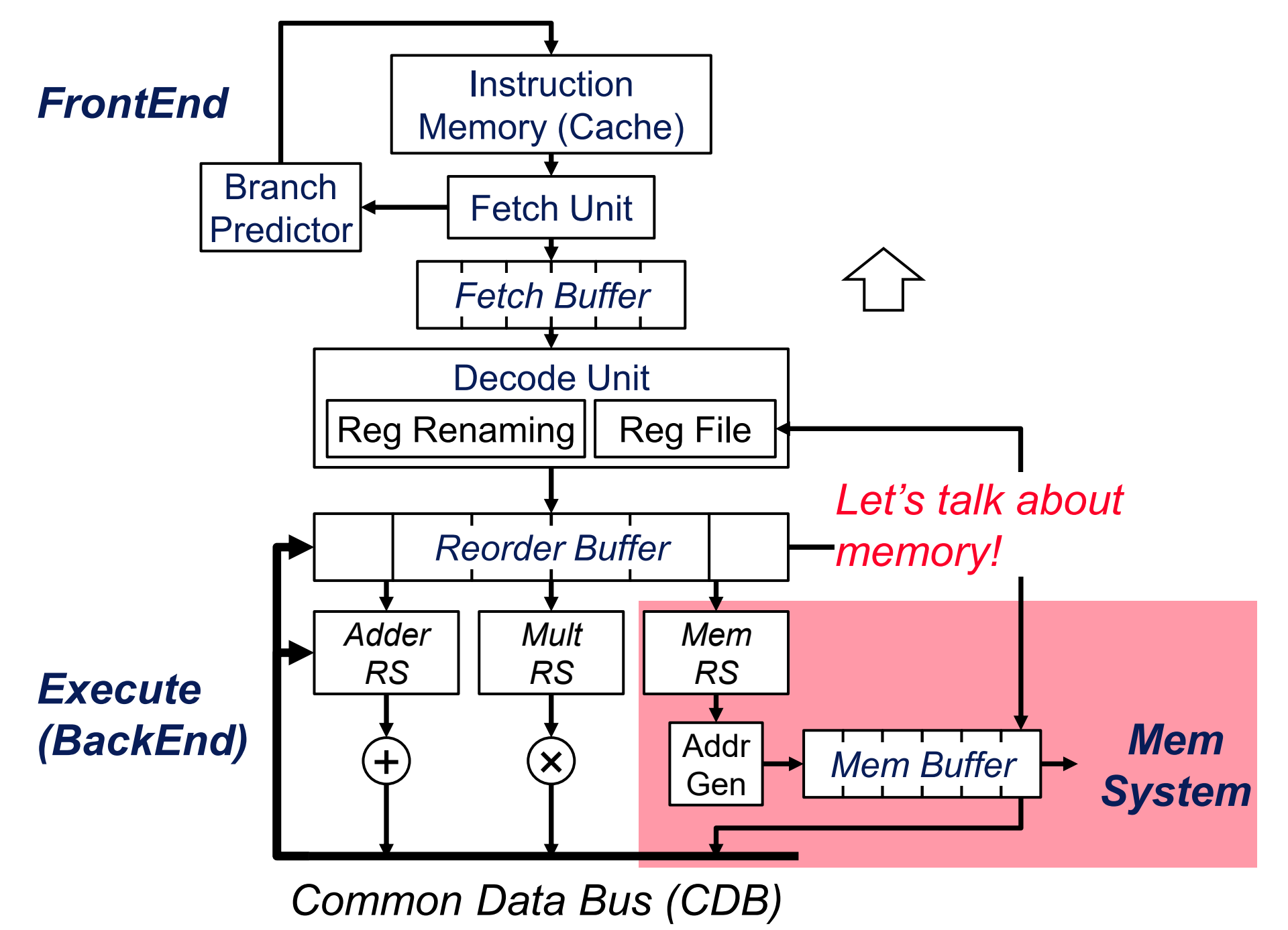
- Data Hazard
  - Data dependency (Read-after-write: RAW)
  - Anti dependency (Write-after-read: WAR)
  - Output dependency (Write-after-write: WAW)
- Control Hazard
  - Data dependency of program counter
- Structural Hazard
  - Due to the lack of resources
    - Ex1) We need to split the instruction and data memory (to simultaneously execute IF and MEM stage
    - Ex2) We may execute multiple instructions in the EX stage (#ALU < # instructions ready)

What about the dependence in memory?

- You've seen the dependence between register accesses (read and write)

- What about the memory accesses? ➔ What if two different instructions load and store to the same address?
  - Register dependence is statically known ⇔ Memory dependence is dynamically known
  - Register state is small ⇔ Memory state is large

- But still, CPU needs to ensure memory dependencies in an out-of-order processor!
  - This is called a **"Memory Disambiguation"**

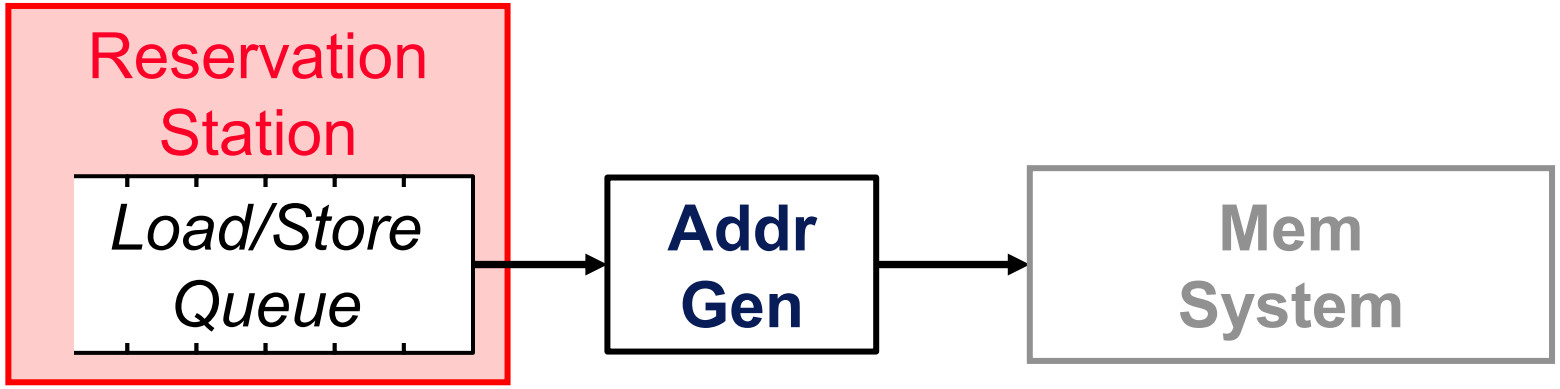


Memory system in CPU pipeline

- **Q1 (Addr Gen).** When do you calculate the memory addresses of the target load store instructions
  - **Option #1:** In order (after all the previous load and store instructions have completed their address calculations)
  - **Option #2:** Out-of-order (as soon as the operands are ready)
- **Q2 (Mem Access).** When do you dispatch the memory instructions to the memory system
  - **Option #1 (Total ordering):** In order (after all the previous load and store instructions have issued the memory requests)
  - **Option #2 (Load ordering / Store ordering):** Execution between loads and stores out of order, but preserve load-load ordering and store-store ordering
  - **Option #3 (Partial ordering):** All stores proceed in order, but loads execute out of order (as long as all previous stores have computed their address)
  - **…**

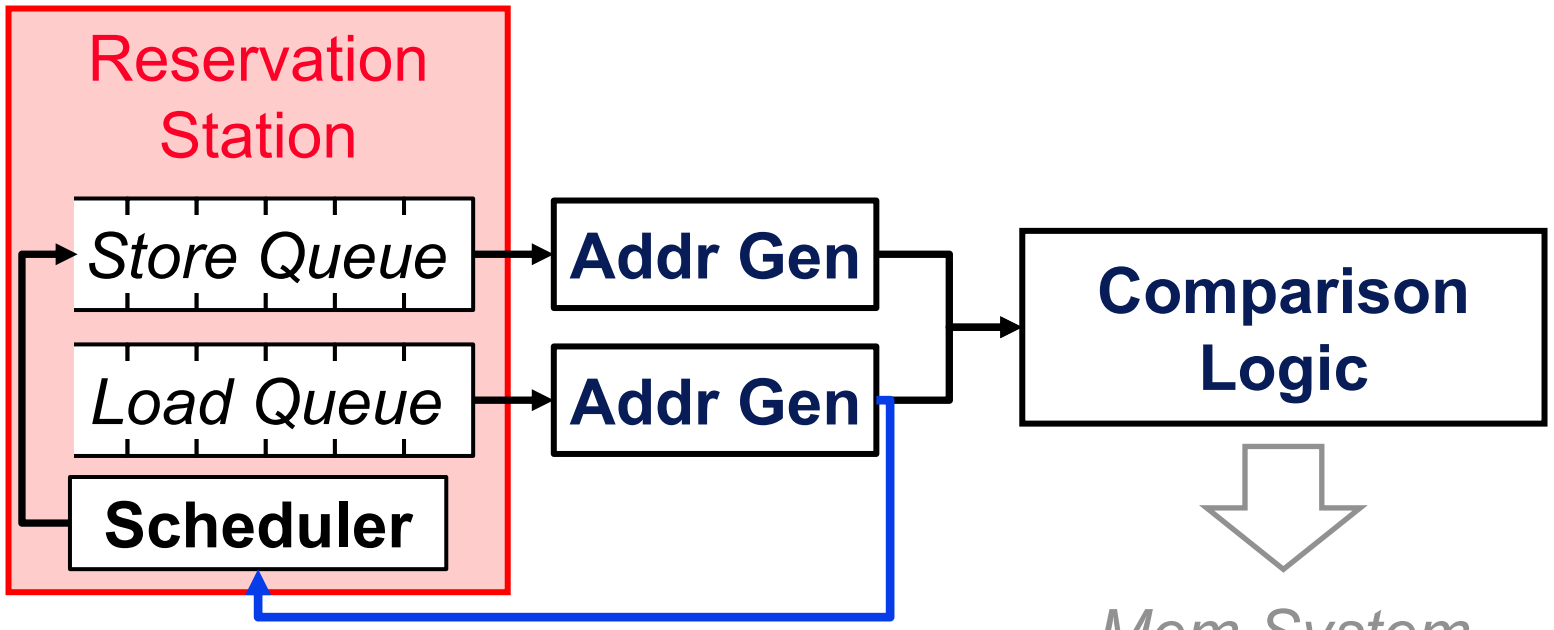
In-order address generation

- Dispatch instructions to the address generation unit only when (1) the operands for the address generation is ready and (2) the instruction is in the oldest entry



Out-of-order address generation

- Dispatch instructions to the address generation unit only when (1) the operands for the address generation is ready and (2) the instruction is in the oldest entry at **each** queue



Loads check if all the older stores have completed address generation

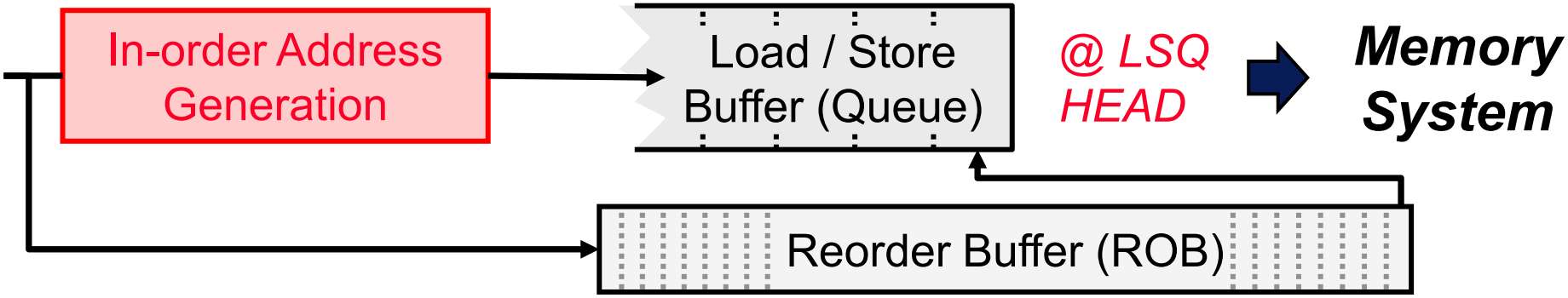
Memory system in CPU pipeline

- **Q1 (Addr Gen).** When do you calculate the memory addresses of the target load store instructions
  - **Option #1:** In order (after all the previous load and store instructions have completed their address calculations)
  - **Option #2:** Out-of-order (as soon as the operands are ready)
- **Q2 (Mem Access).** When do you dispatch the memory instructions to the memory system
  - **Option #1 (Total ordering):** In order (after all the previous load and store instructions have issued the memory requests)
  - **Option #2 (Load ordering / Store ordering):** Execution between loads and stores out of order, but preserve load-load ordering and store-store ordering
  - **Option #3 (Partial ordering):** All stores proceed in order, but loads execute out of order (as long as all previous stores have computed their address)
  - **…**

In-order Load/Stores (Total Ordering)

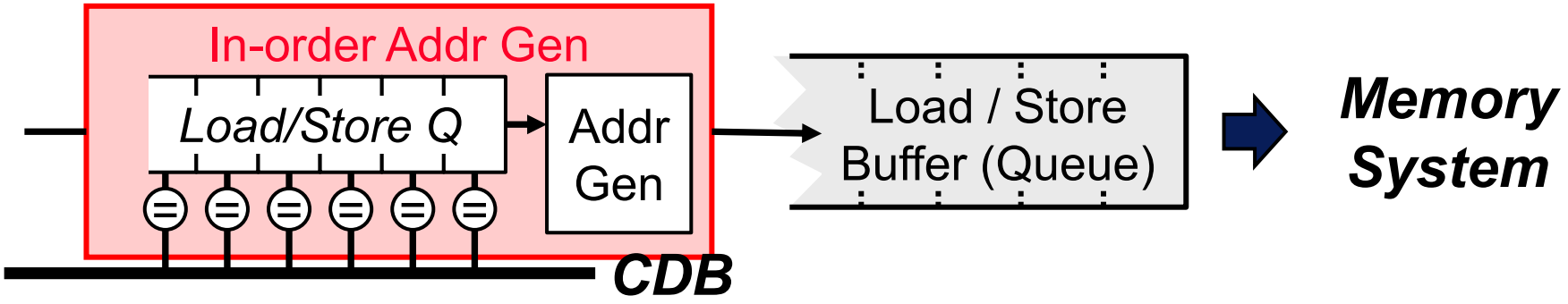
- Perform all loads and stores in-order (in the program order), but non-memory instructions can be executed out-of-order
  - Pessimistic, but guarantees correctness!

- Load & store queue (LSQ) → a circular queue to execute memory operations in order
  - The CPU can execute the instruction at the LSQ head
    - Load: perform load right away!
    - Store: wait until the store is at the ROB head *Precise Exception

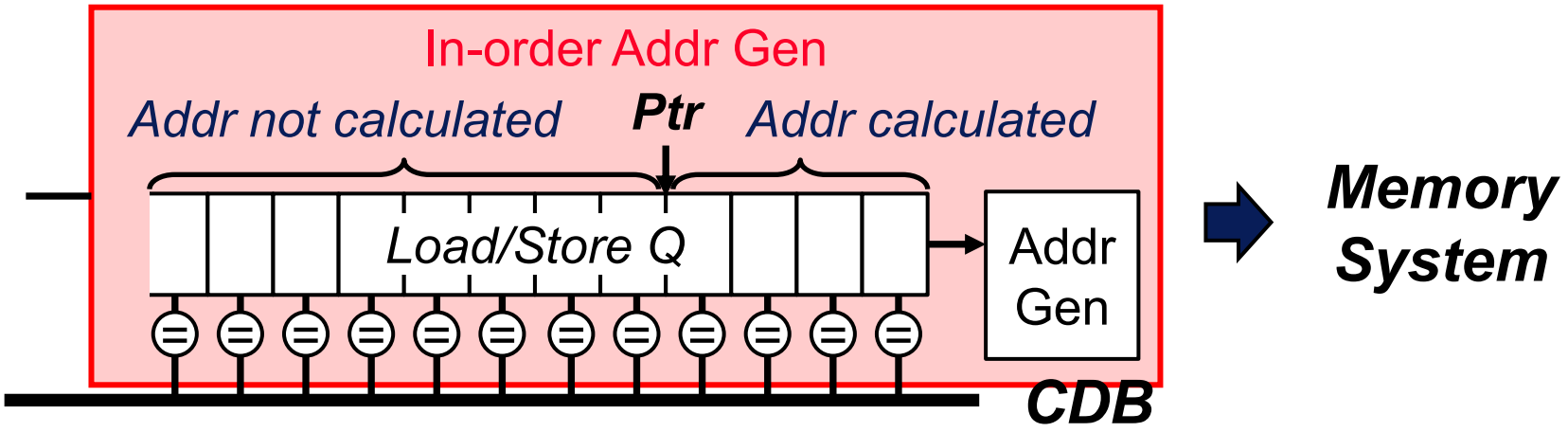


Quick Note: Why Separate Buffers?

- **Option #1:** Make a separate buffer for both load/store Q in address gen + load/store buffer



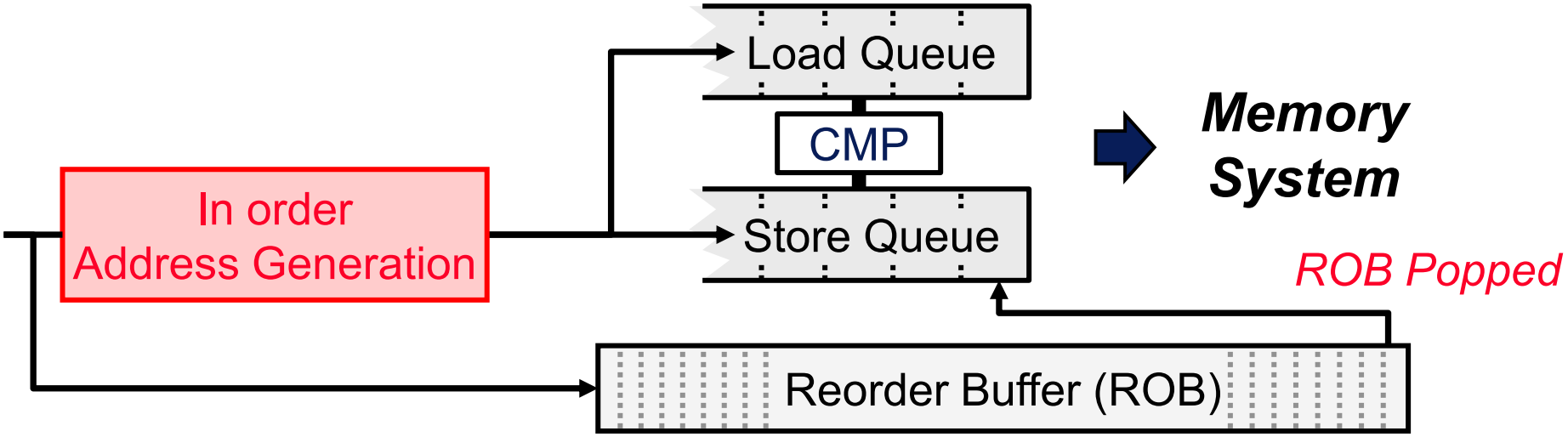
- **Option #2:** Unify the load/store queue (simple … but)



Load Ordering / Store Ordering

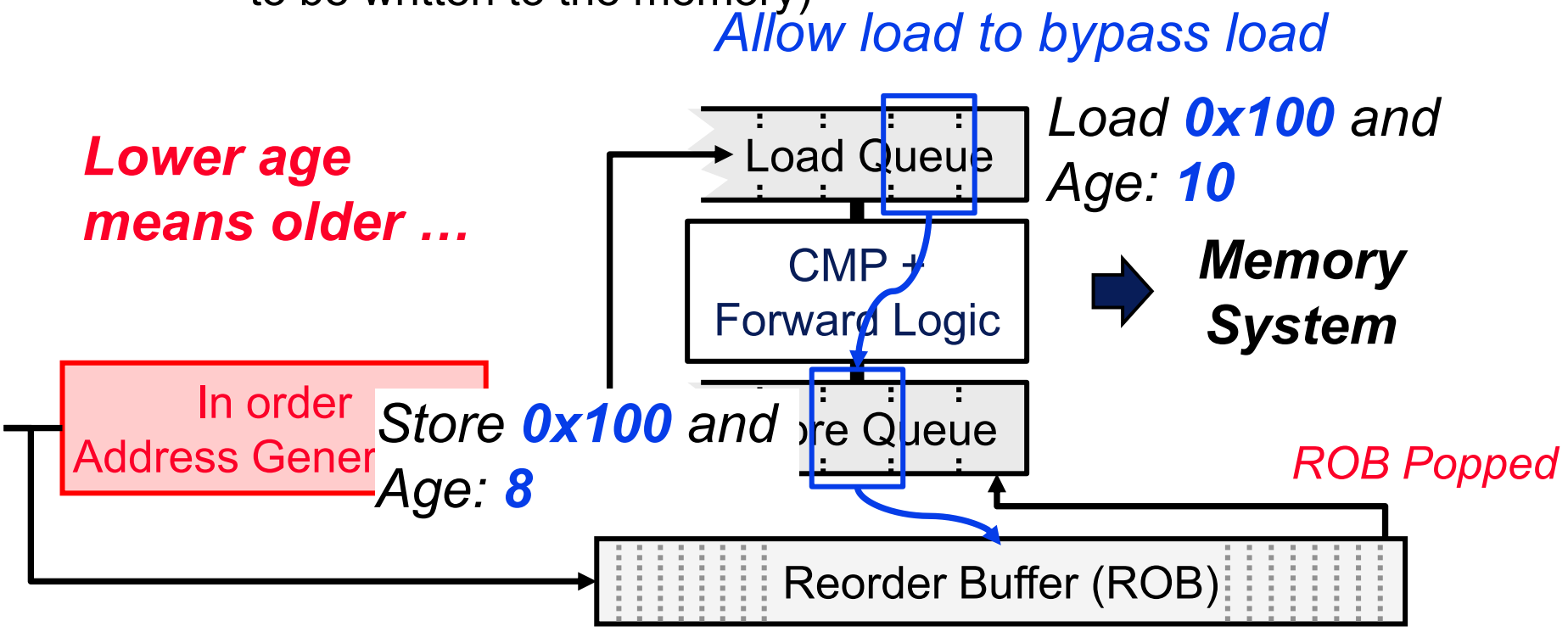
- Perform loads in advance if it is independent with the preceding stores (but the memory addresses are calculated in-order)

- We have a separate load and store queue
  - Load: execute a load instruction (memory operation) if it is independent of all the store instructions
  - Store: wait until the store is at the ROB head



Partial Ordering

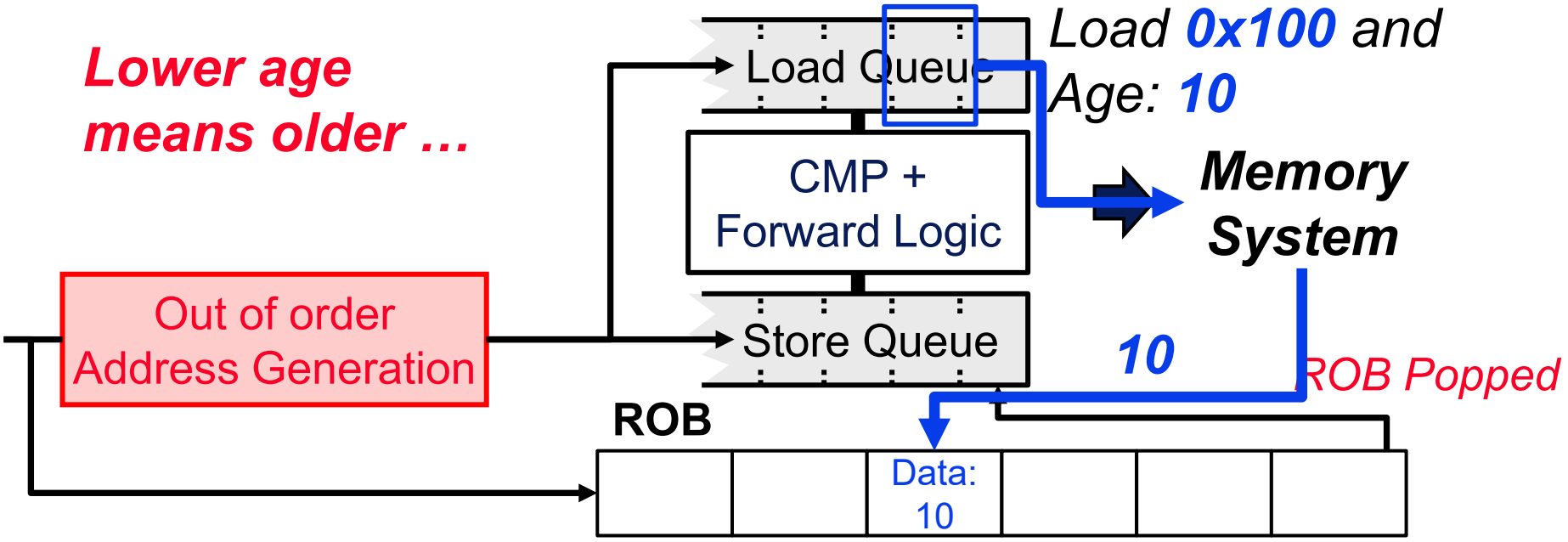
- If there exists a dependent store instruction in the store queue, the load instruction can use the data to store in the memory
  - We can prevent unnecessary idle cycles (to wait for the data to be written to the memory)



Store Ordering

- Stores execute in-order, but loads execute completely out of order (even without waiting for the stores addresses to become available!)
- Only store-to-store ordering is kept!

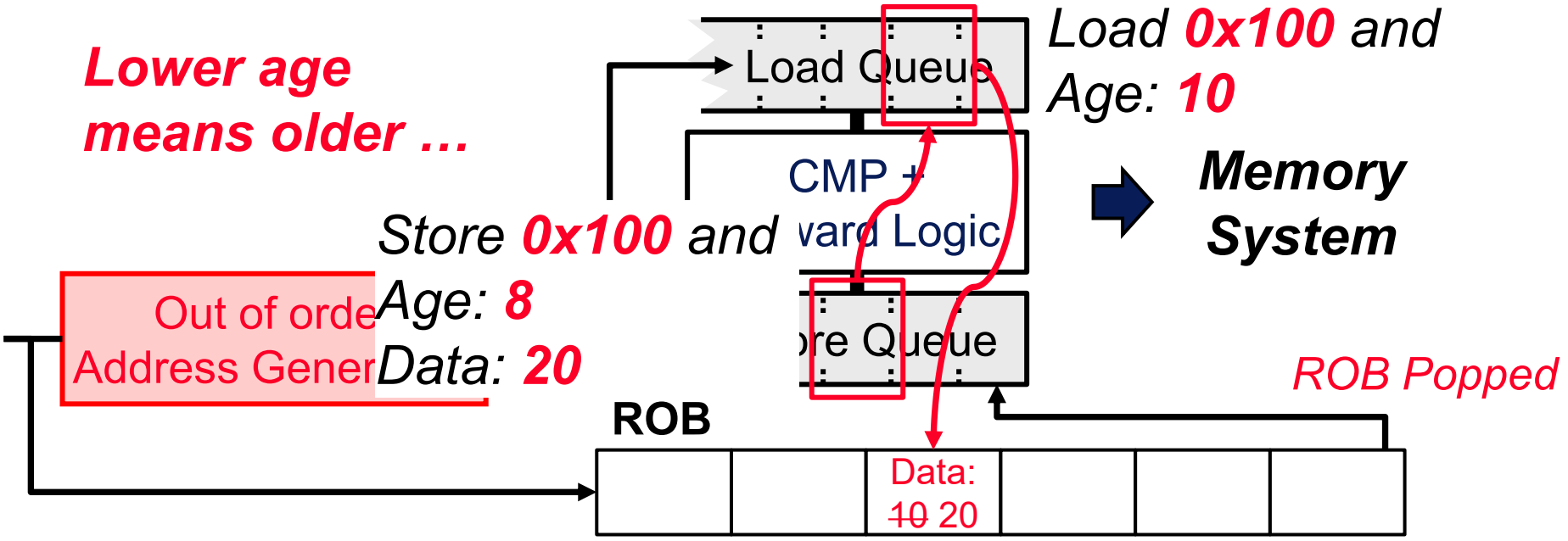
This potentially have memory dependence, but whatever …



Store Ordering

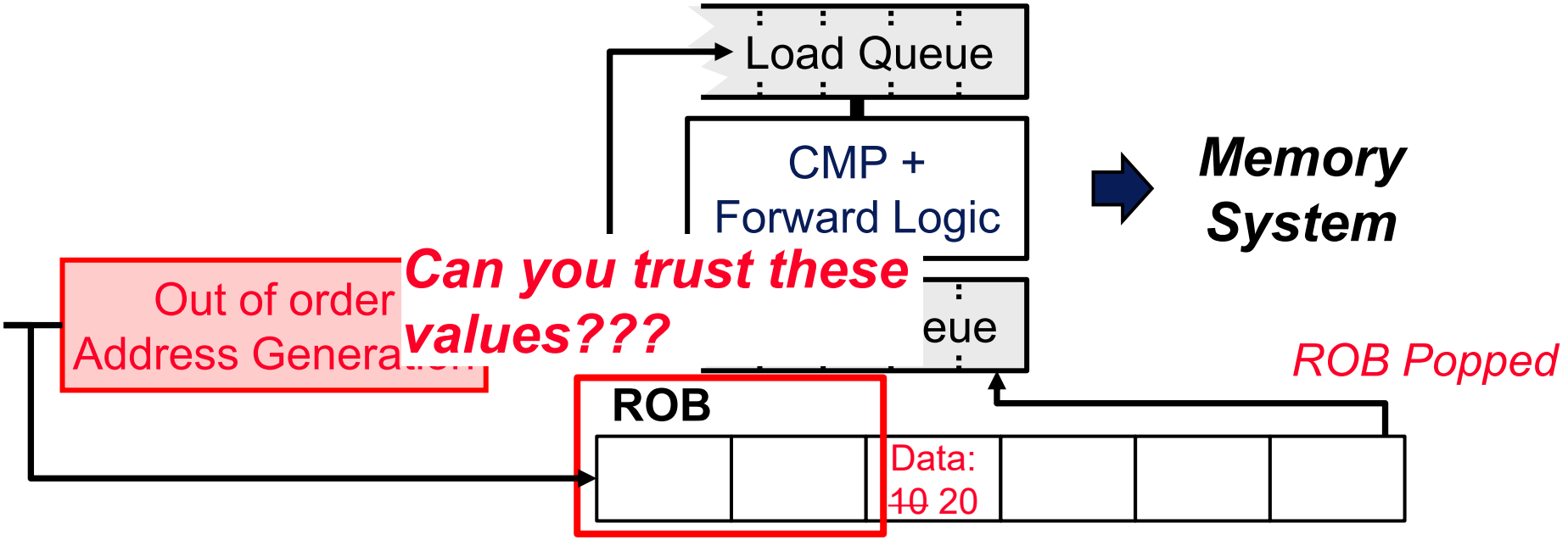
- Stores execute in-order, but loads execute completely out of order (even without waiting for the stores addresses to become available!)
- Only store-to-store ordering is kept!

This potentially have memory dependence, but whatever …



Store Ordering

- Stores execute in-order, but loads execute completely out of order (even without waiting for the stores addresses to become available!)
- Only store-to-store ordering is kept!



Prediction

- We need a mechanism to correct mis-speculated loads. Is this easy?? Mis-prediction propagates!

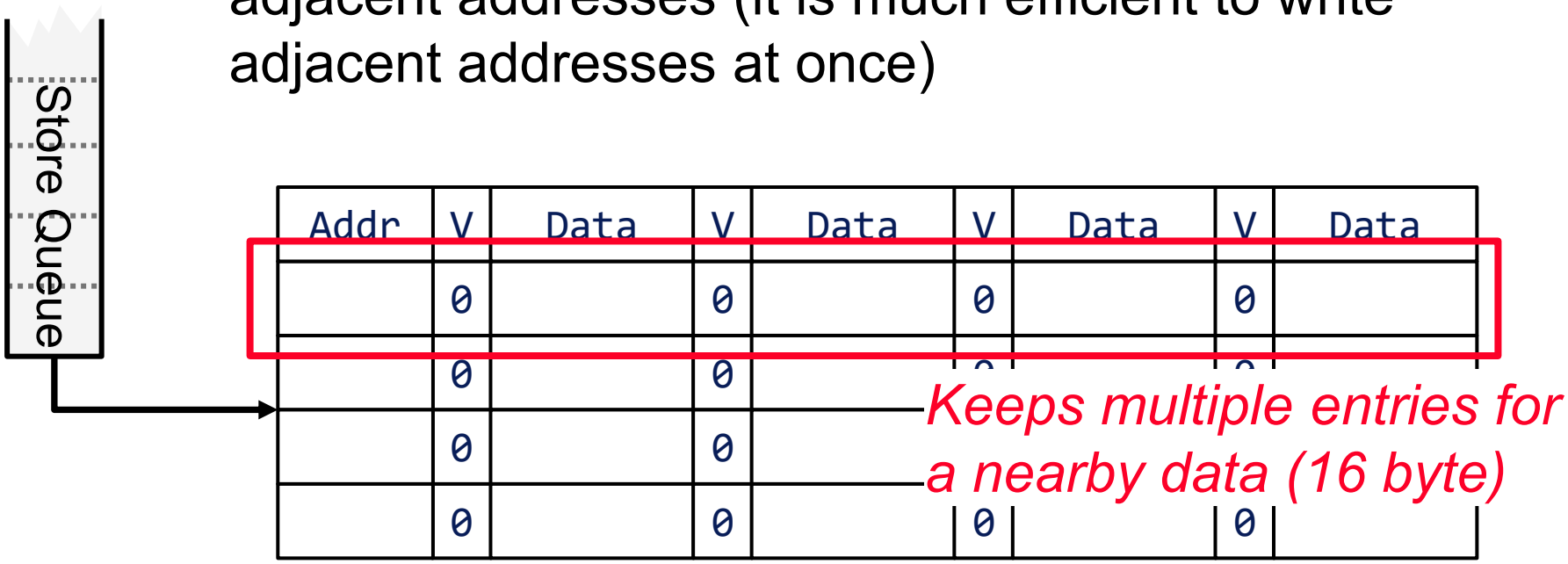
```
sw r1 100(r2)
lw r3 50(r5)     // assume that lw is mis-speculated
add r4 r3 r8     // incorrect r3 in the ROB will be
                 // used to calculate r4 …
```

- **Flush all the ROB entries (starting from the mis-speculated load) ➔ easy but high overhead**
  - Mitigation #1: Re-execute only incorrect instructions
  - Mitigation #2: Predict … (load-store pairs that are highly likely to be dependent)

Modifying Store to Store Order: Merging Write Buffer

- You can adopt a write buffer before writing the data to the memory (or cache as you will learn later on …)

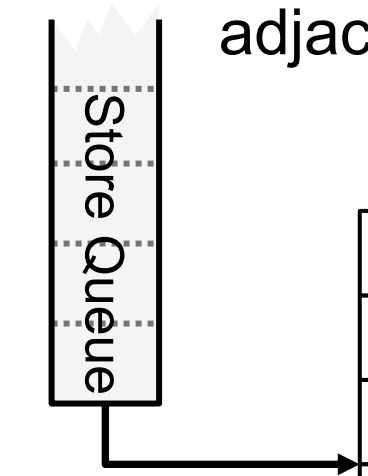
- Write buffer may merge multiple entries in the adjacent addresses (it is much efficient to write adjacent addresses at once)



Modifying Store to Store Order: Merging Write Buffer

- You can adopt a write buffer before writing the data to the memory (or cache as you will learn later on …)

- Write buffer may merge multiple entries in the adjacent addresses (it is much efficient to write adjacent addresses at once)



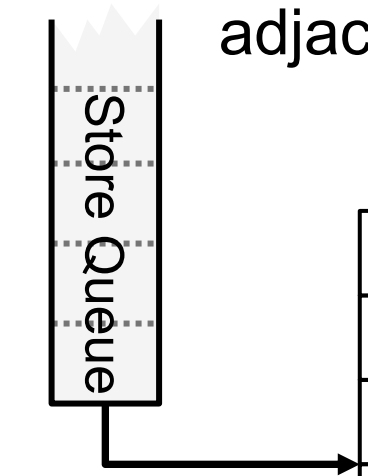
Addr: 0x100
Data: 50

Addr	V	Data	V	Data	V	Data	V	Data
0x100	1	50	0		0		0	
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Modifying Store to Store Order: Merging Write Buffer

- You can adopt a write buffer before writing the data to the memory (or cache as you will learn later on …)

- Write buffer may merge multiple entries in the adjacent addresses (it is much efficient to write adjacent addresses at once)



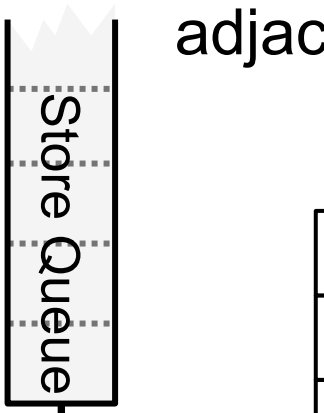
Addr	V	Data	V	Data	V	Data	V	Data
0x100	1	50	0		0		0	
0x200	**1**	**100**	0		0		0	
	0		0		0		0	
	0		0		0		0	

Addr: 0x200
Data: 100

Modifying Store to Store Order: Merging Write Buffer

- You can adopt a write buffer before writing the data to the memory (or cache as you will learn later on …)

- Write buffer may merge multiple entries in the adjacent addresses (it is much efficient to write adjacent addresses at once)



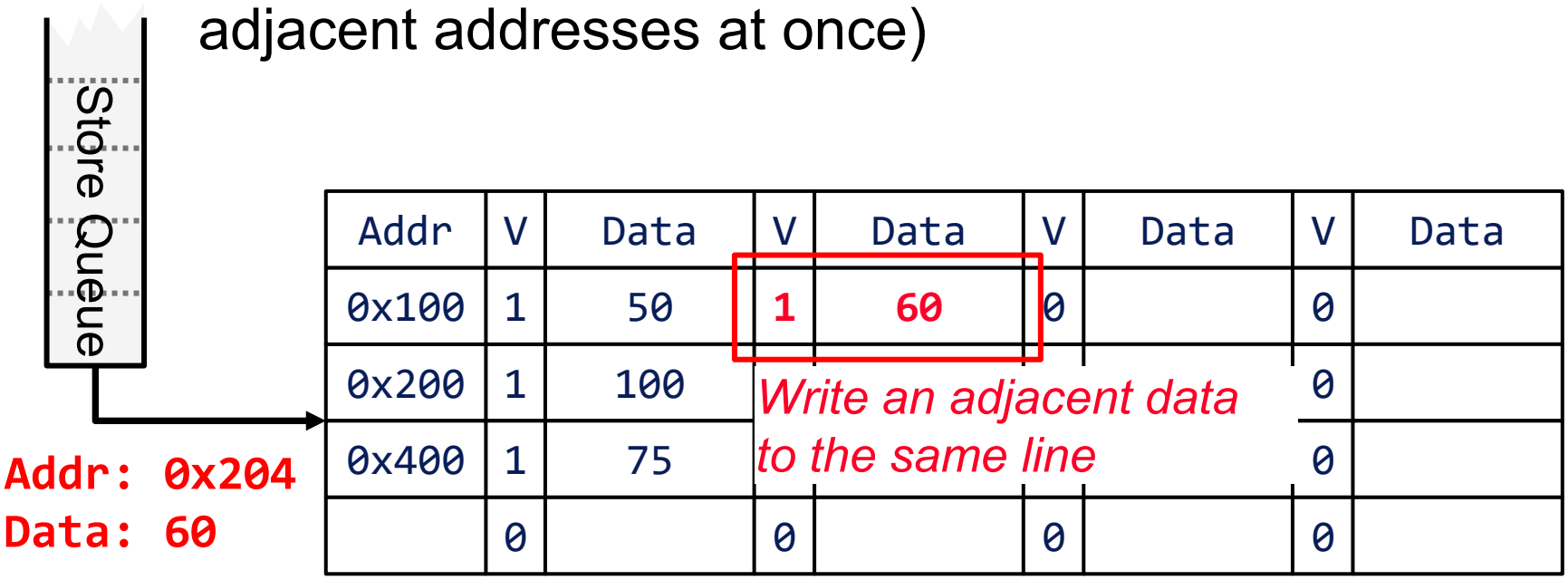
Addr	V	Data	V	Data	V	Data	V	Data
0x100	1	50	0		0		0	
0x200	1	100	0		0		0	
0x400	1	75	0		0		0	
	0		0		0		0	

Addr: 0x400
Data: 75

Modifying Store to Store Order: Merging Write Buffer

- You can adopt a write buffer before writing the data to the memory (or cache as you will learn later on …)

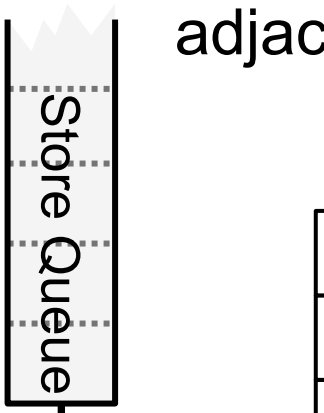
- Write buffer may merge multiple entries in the adjacent addresses (it is much efficient to write adjacent addresses at once)



Modifying Store to Store Order: Merging Write Buffer

- You can adopt a write buffer before writing the data to the memory (or cache as you will learn later on …)

- Write buffer may merge multiple entries in the adjacent addresses (it is much efficient to write adjacent addresses at once)



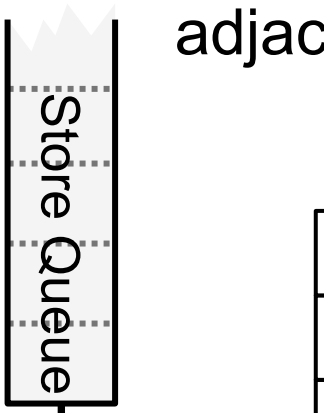
Addr: 0x300
Data: 10

Addr	V	Data	V	Data	V	Data	V	Data
0x100	1	50	1	60	0		0	
0x200	1	100	0		0		0	
0x400	1	75	0		0		0	
0x300	1	10	0		0		0	

Modifying Store to Store Order: Merging Write Buffer

- You can adopt a write buffer before writing the data to the memory (or cache as you will learn later on …)

- Write buffer may merge multiple entries in the adjacent addresses (it is much efficient to write adjacent addresses at once)



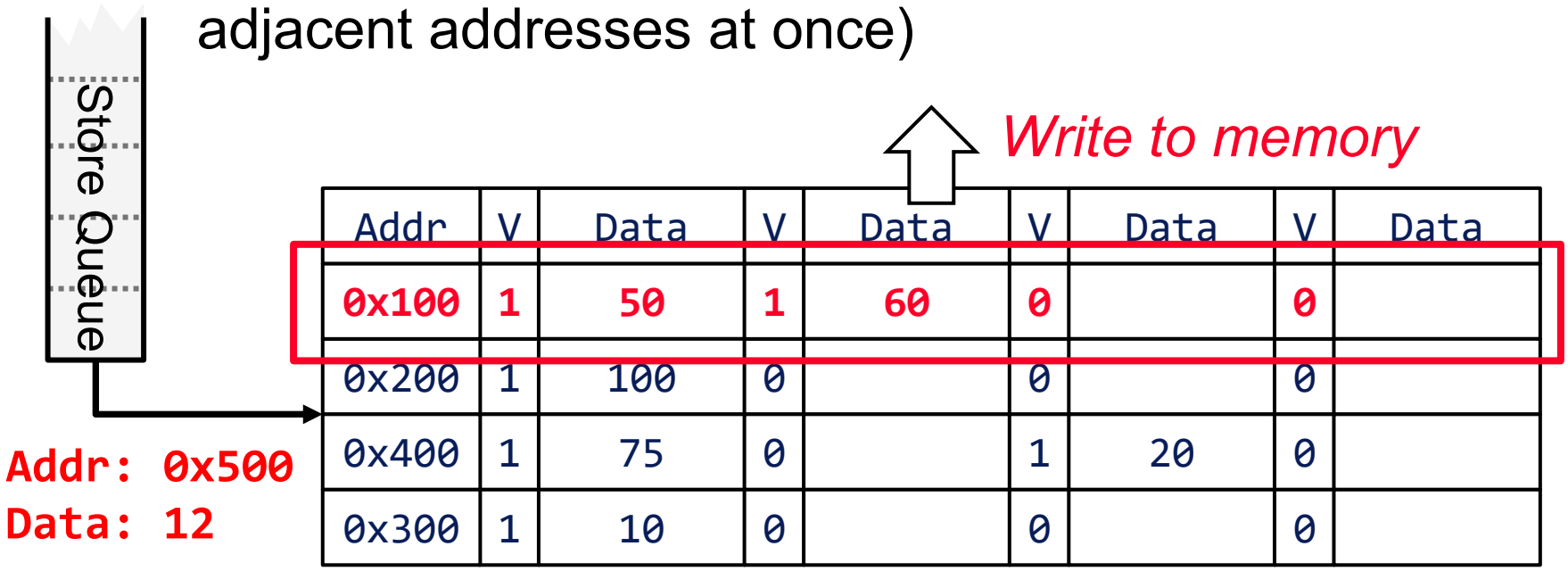
Addr: 0x408
Data: 20

Addr	V	Data	V	Data	V	Data	V	Data
0x100	1	50	1	60	0		0	
0x200	1	100	0		0		0	
0x400	1	75	0		1	20	0	
0x300	1	10	0		0		0	

Modifying Store to Store Order: Merging Write Buffer

- You can adopt a write buffer before writing the data to the memory (or cache as you will learn later on …)

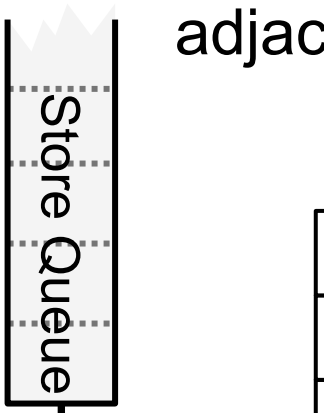
- Write buffer may merge multiple entries in the adjacent addresses (it is much efficient to write adjacent addresses at once)



Modifying Store to Store Order: Merging Write Buffer

- You can adopt a write buffer before writing the data to the memory (or cache as you will learn later on …)

- Write buffer may merge multiple entries in the adjacent addresses (it is much efficient to write adjacent addresses at once)



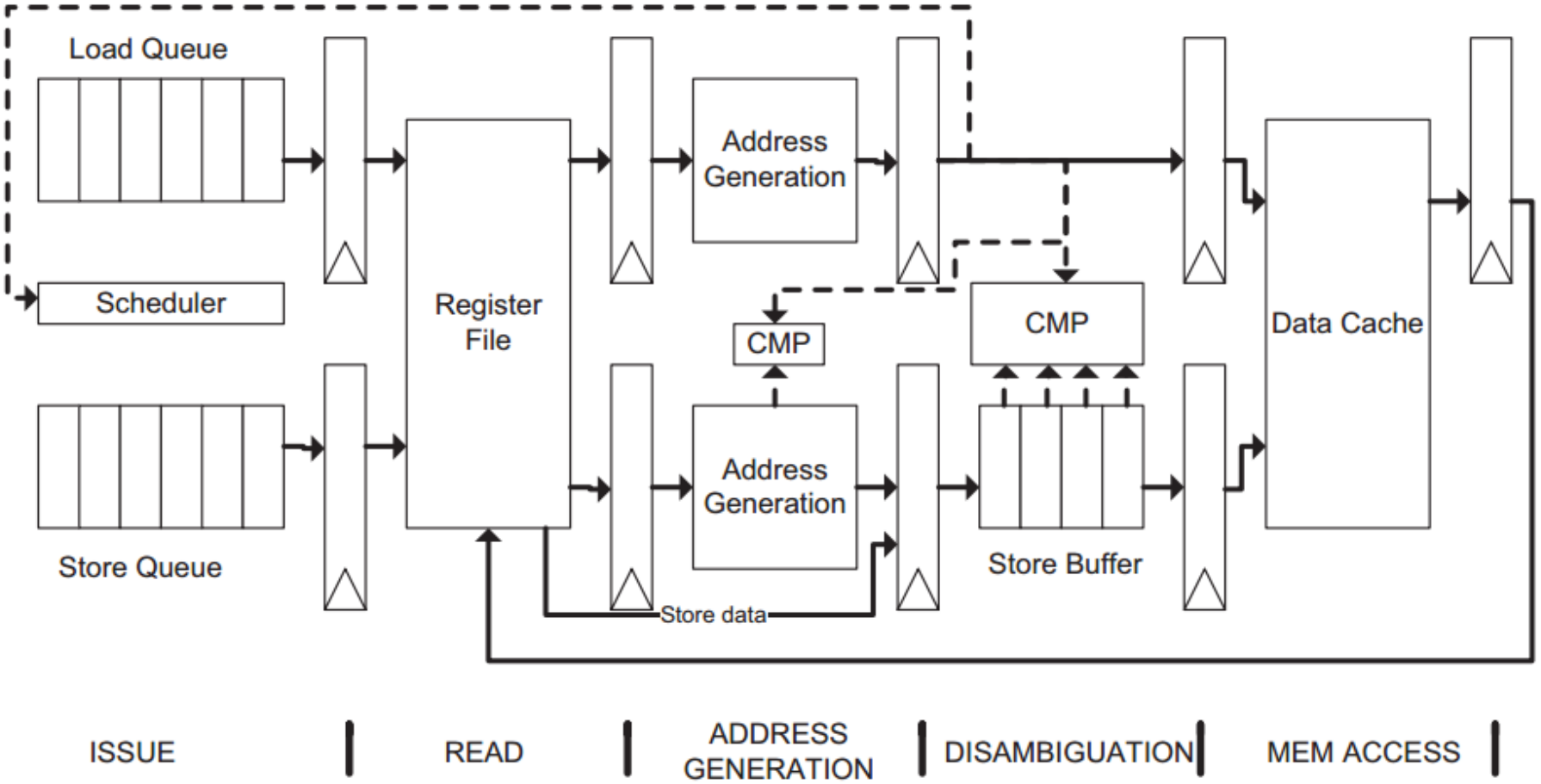
Addr: 0x500
Data: 12

Addr	V	Data	V	Data	V	Data	V	Data
0x200	1	100	0		0		0	
0x400	1	75	0		1	20	0	
0x300	1	10	0		0		0	
0x500	1	12	0		0		0	

Implementation Details

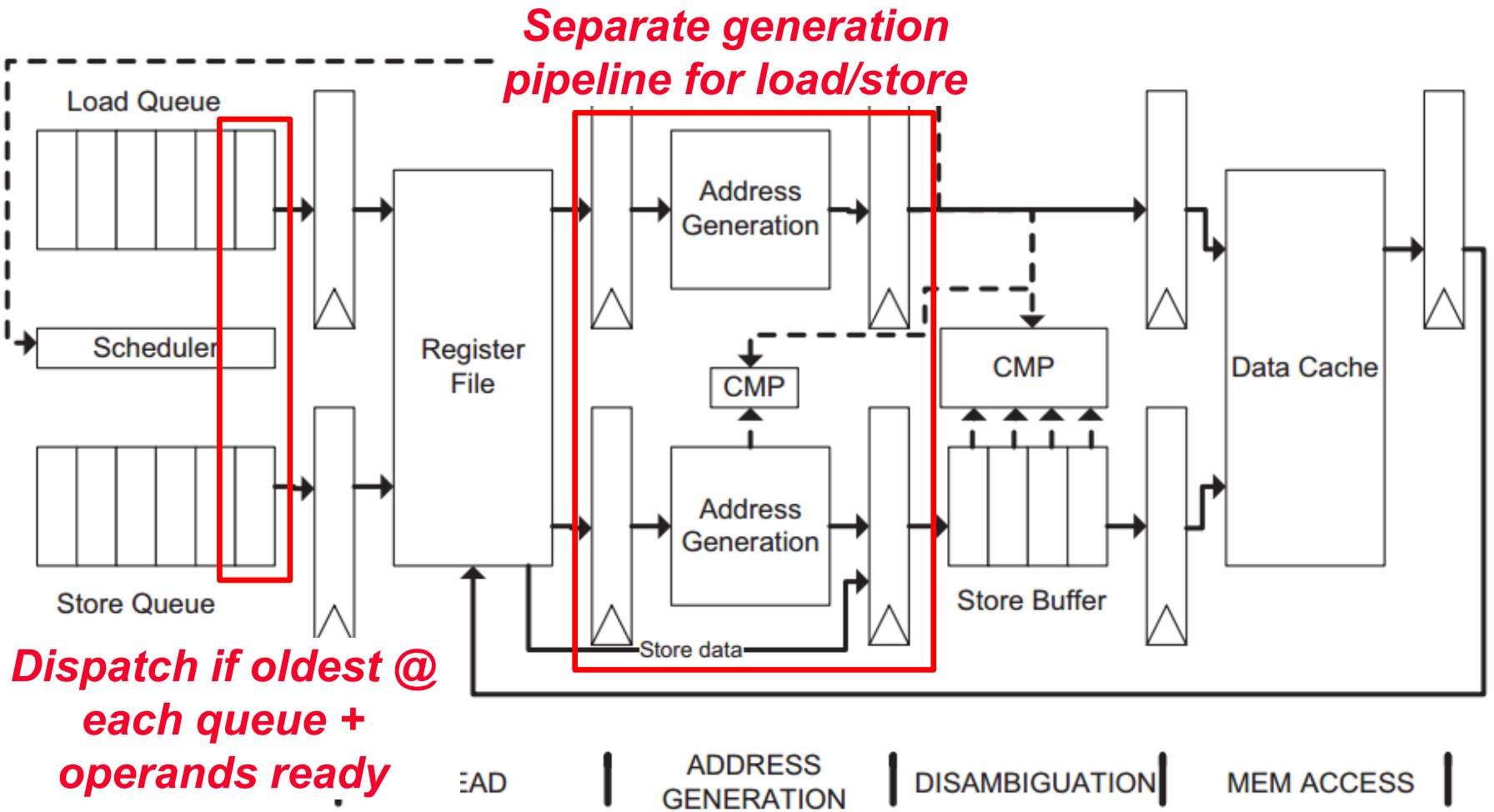
AMD K6

- **AMD K6: load ordering + store ordering**



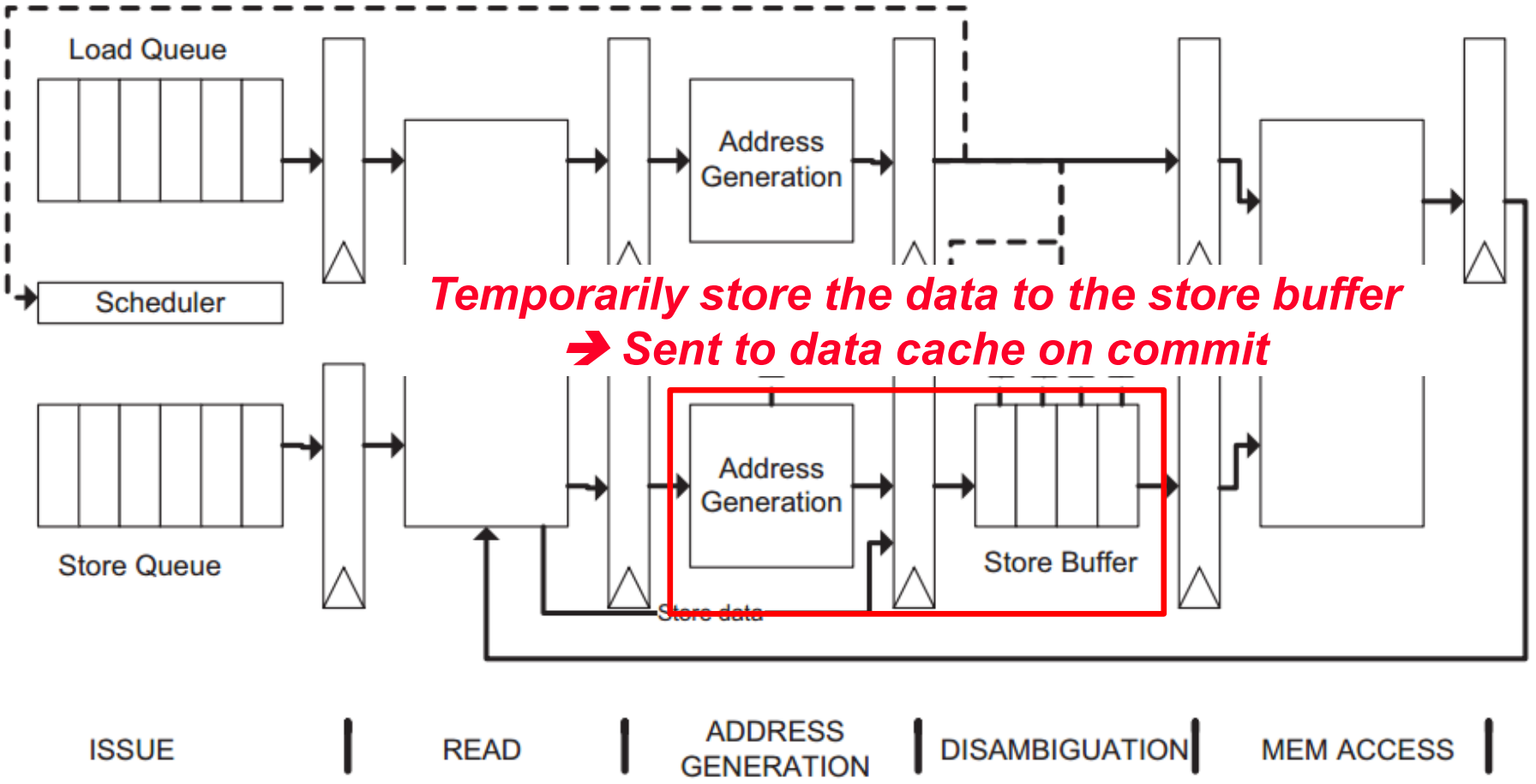
AMD K6

- Execute load operations in-order and store operations in-order (but loads may bypass stores)



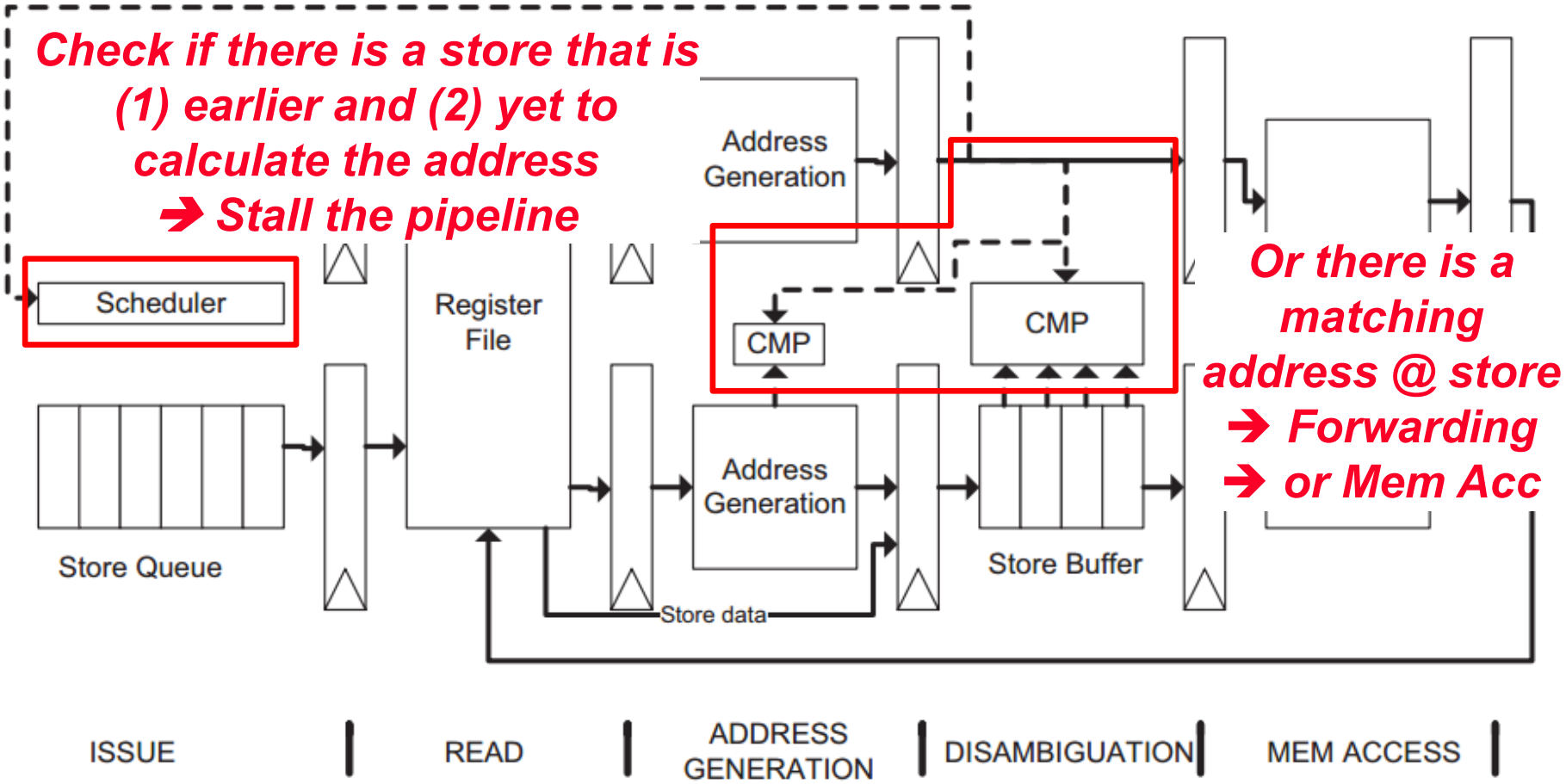
AMD K6

- Execute load operations in-order and store operations in-order (but loads may bypass stores)



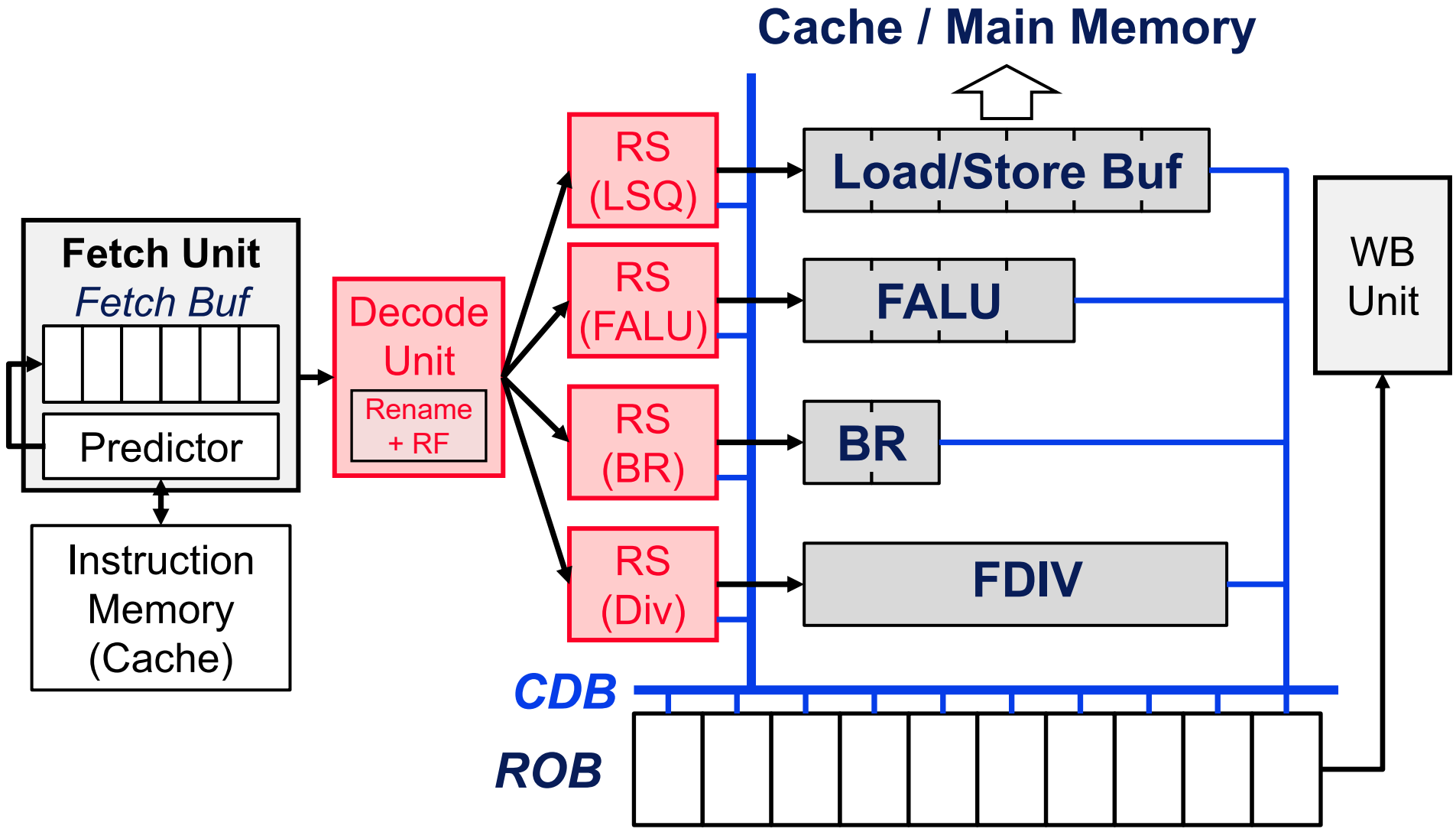
AMD K6

- Execute load operations in-order and store operations in-order (but loads may bypass stores)

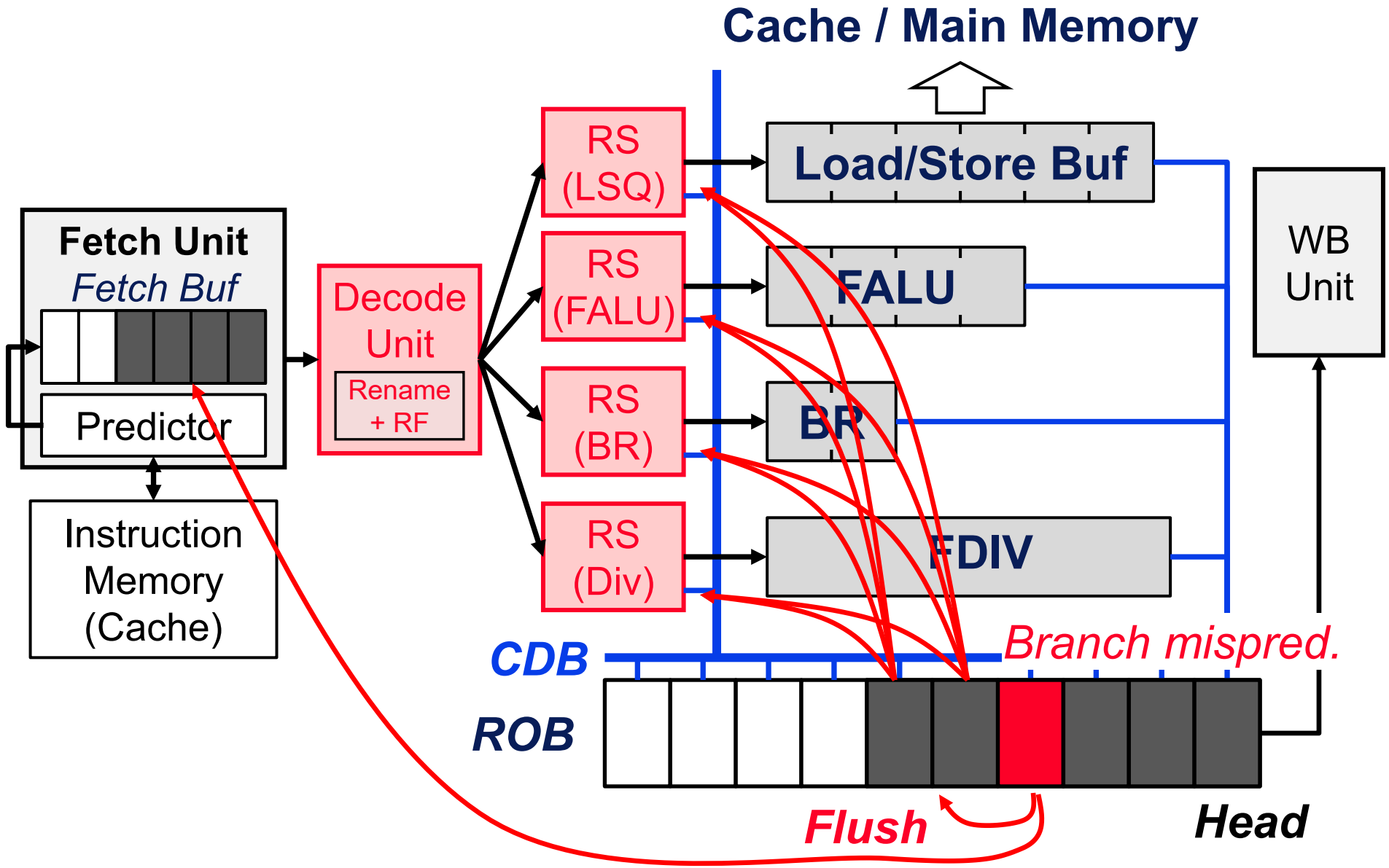


Overview

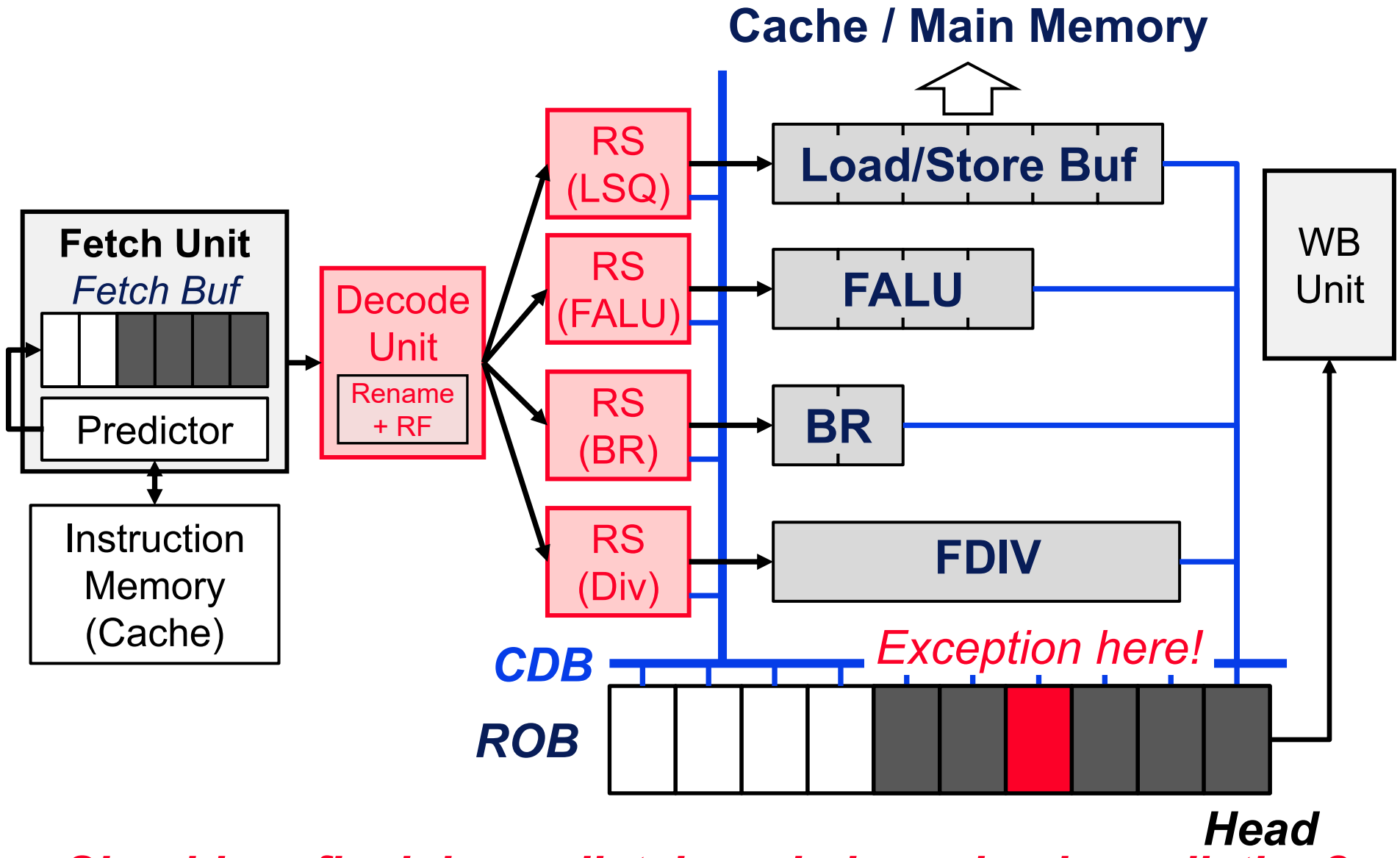
Overall architecture



Overall architecture

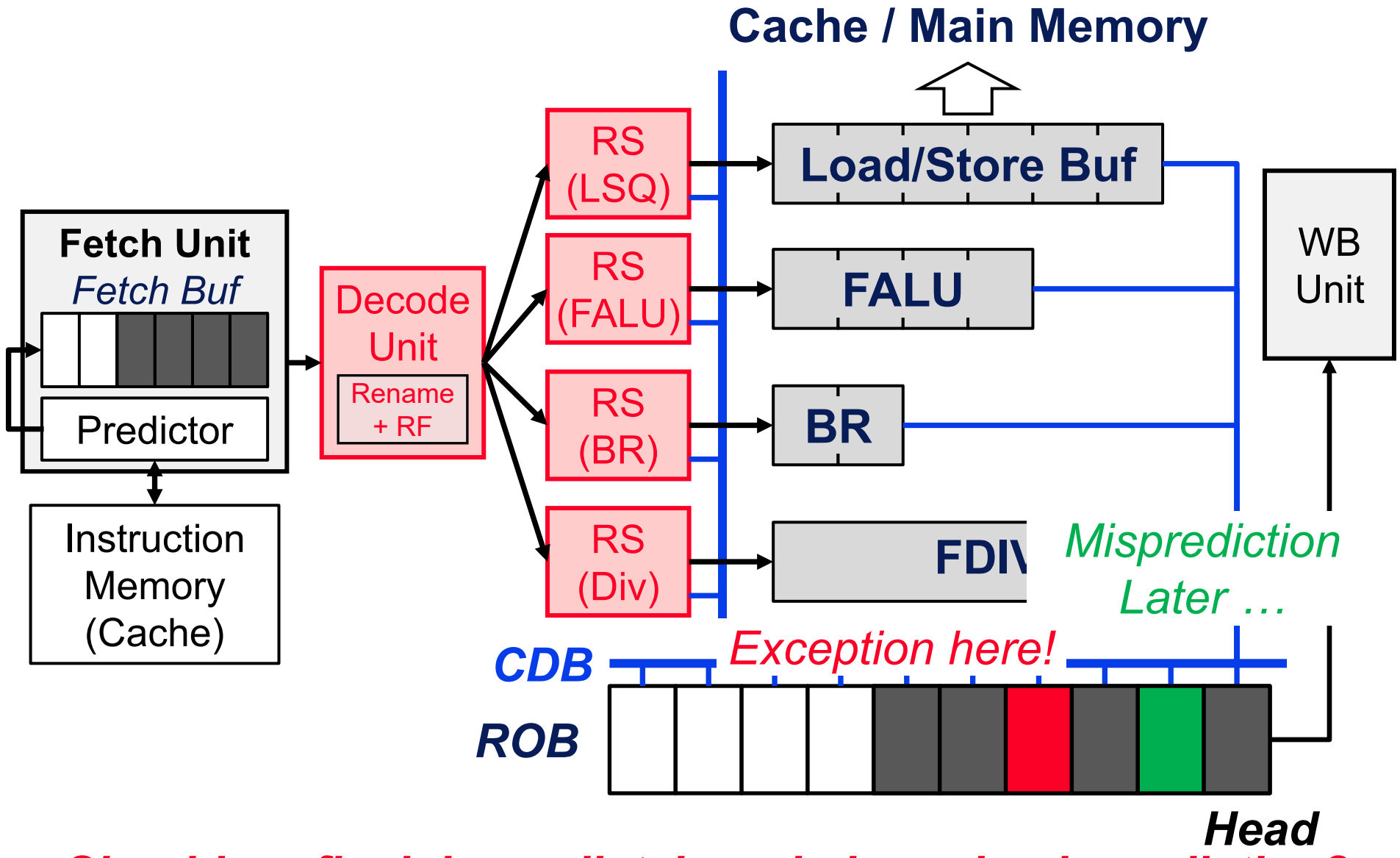


Overall architecture



Should we flush immediately as in branch misprediction?

Overall architecture



Should we flush immediately as in branch misprediction?

Hint for later classes (After midterm exam)

- Can you apply all the optimizations?
  - Out-of-order address generation …
  - Load bypassing
  - Load forwarding
  - Out-of-order load execution …
  - Merging store buffer?

- Why? ➔ You are playing in **the von Neumann Architecture**
  - You reorder or correct the incorrect instruction orders anyway …

- Why not? ➔ Does the assumption hold for multiprocessor?

Question?

Announcements:

Reading: *finish reading P&H Ch.4*

Handouts: *none*