High Performance Computer Architecture (CS60003)

Dept. of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Spring 2023

Load Store Queue (LSQ)

What is really Out-of-Order?

- Fetch
- Decode
- Issue
- Exec
- Broadcast
- Commit

Inorder

Out of Order respecting true dependencies

Inorder

(RETIRE)

Memory Access Ordering

- We have seen OOO processors track when one instruction uses register values produced by another instruction.
- However loads and store instructions can access same memory locations, without using the same register!
  - How do we handle them in the OOO processor?
  - Does loads and stores need to be done strictly in program order?
  - Can we reorder them?

Handling Dependencies

- Control Dependencies: Branch Prediction
- False Dependencies: Register Renaming
- True Dependencies: Respected by Tomasulo's Algorithm
- What about memory dependencies?
  - sw R1, 0($R3)
  - Lw R2, 0($R4)

There could be a dependency if the addresses are same ⇒ in which case we have to do inorder.

Q: How do we handle them?

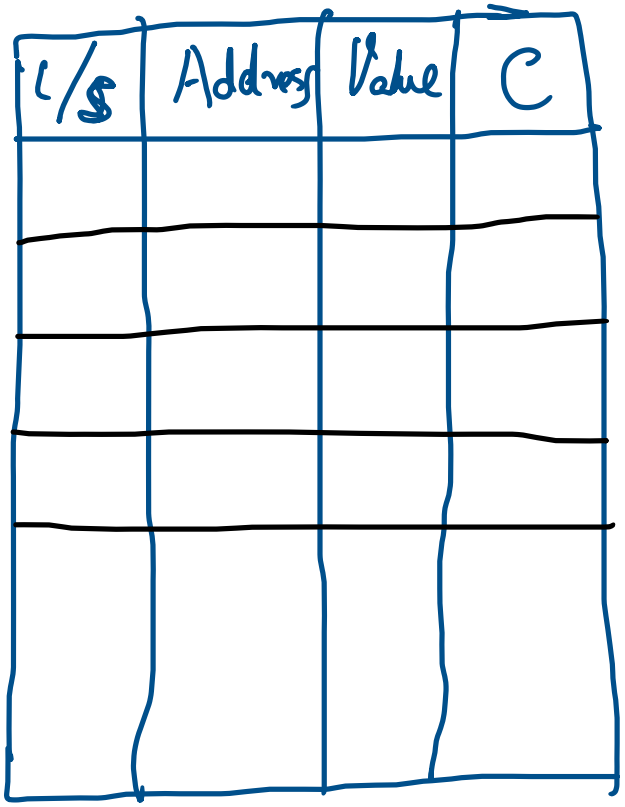
When do we allow stores to write?

- During Commit:
  - It is unsafe to write memory before it is committed.
  - Because in case of exception/branch miss if it is cancelled, we have to undo the effect of a memory write.
  - This is unstable, as we have seen.
- But then does the load have to wait till the stores are committed?
  - That would be very late.
  - Ideally, we would like to get data loaded into registers as soon as possible, to be used in other following instructions.
  - For this we shall be introducing another hardware unit, called the load-store queue (LSQ).

Load Store Queue (LSQ)

- We need this unit to get the load instructions to get data as soon as possible, while the store does not write to memory until commit.

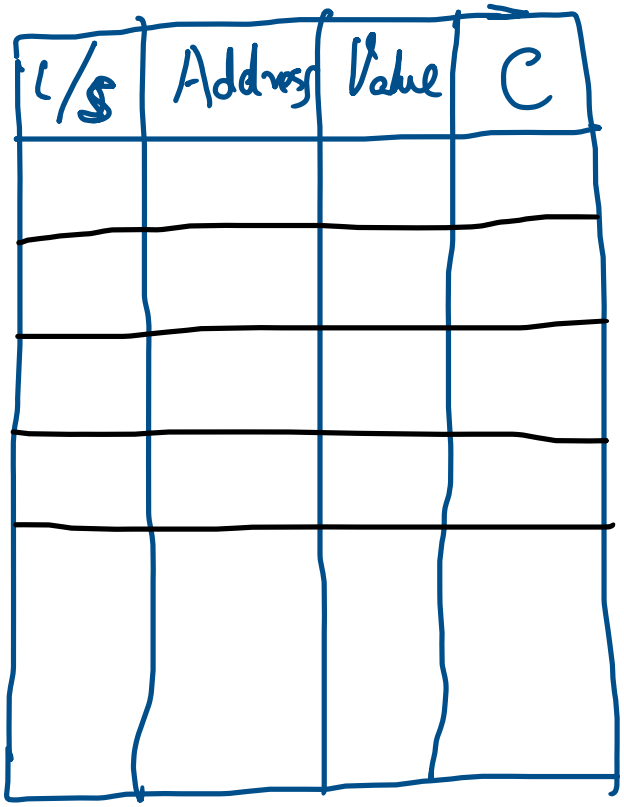
Structure just like ROB
- we put things in-order,
- remove also at commit in-order.



Load Store Queue (LSQ)

- We need this unit to get the load instructions to get data as soon as possible, while the store does not write to memory until commit.

lw R1, 0(R1)
sw R2,0(R3)
lw R4, 0(R4)
sw R5, 0(R0)
lw R5, 0(R8)

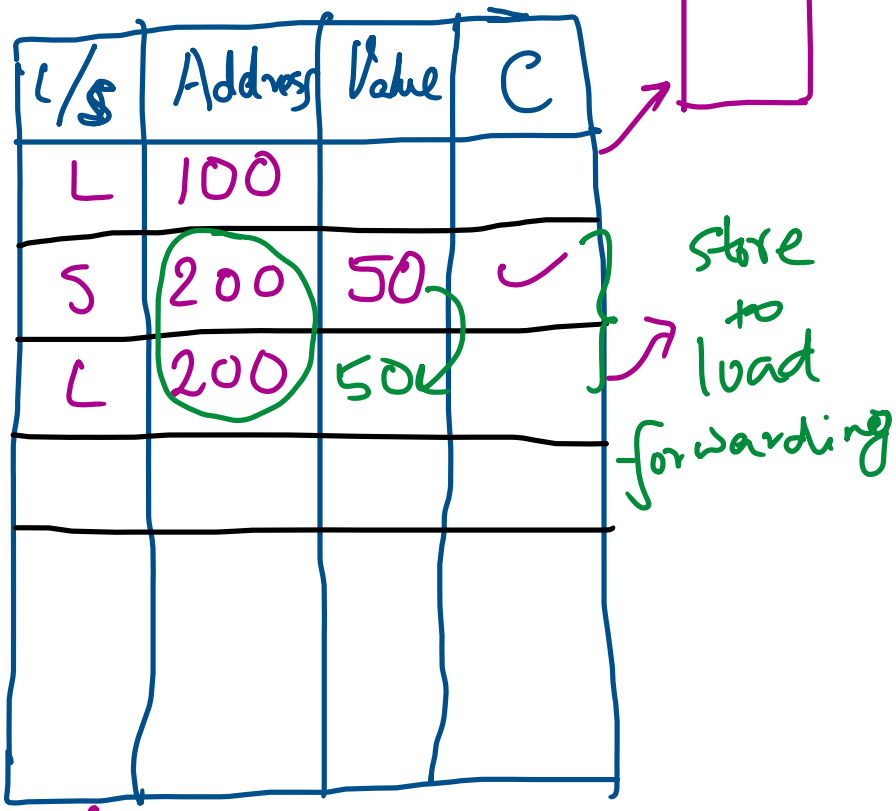


Load Store Queue (LSQ)

- We need this unit to get the load instructions to get data as soon as possible, while the store does not write to memory until commit.

lw R1, 0(R1)
sw R2,0(R3)
lw R4, 0(R4)
sw R5, 0(R0)
lw R5, 0(R8)

Program Order



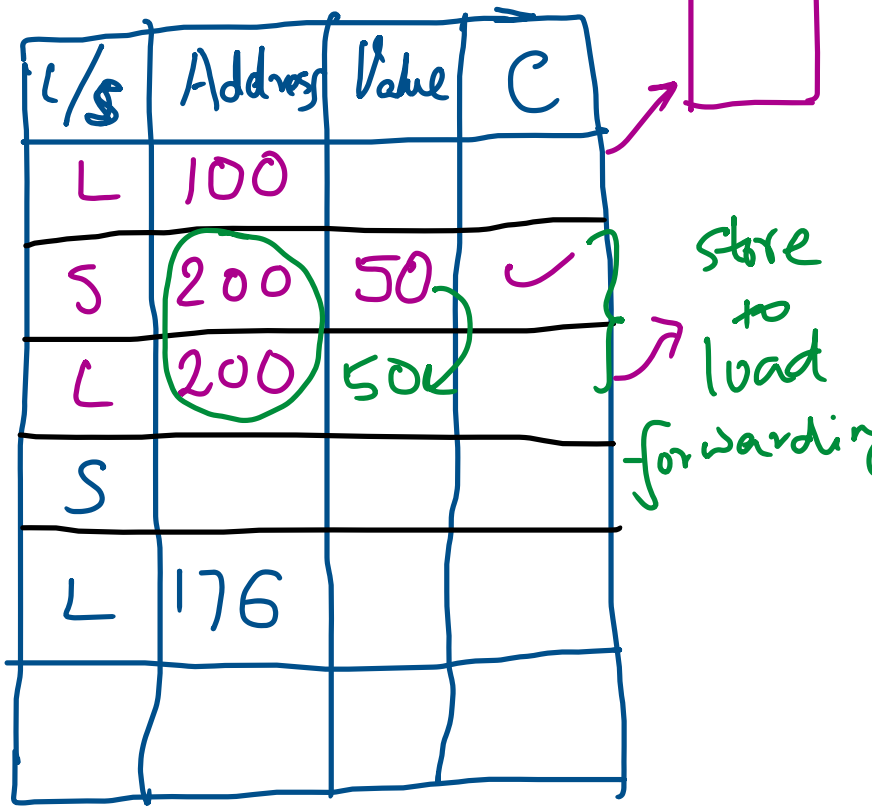
For every load, we check for in-flight stores with same address. If no match, we go to memory, else just forward.

Load Store Queue (LSQ)

- We need this unit to get the load instructions to get data as soon as possible, while the store does not write to memory until commit.

lw R1, 0(R1)
sw R2,0(R3)
lw R4, 0(R4)
sw R5, 0(R0)
lw R5, 0(R8)

Program order



We assumed previous store's Value & address are ready. It may happen the store isn't complete. What do we do?

Options

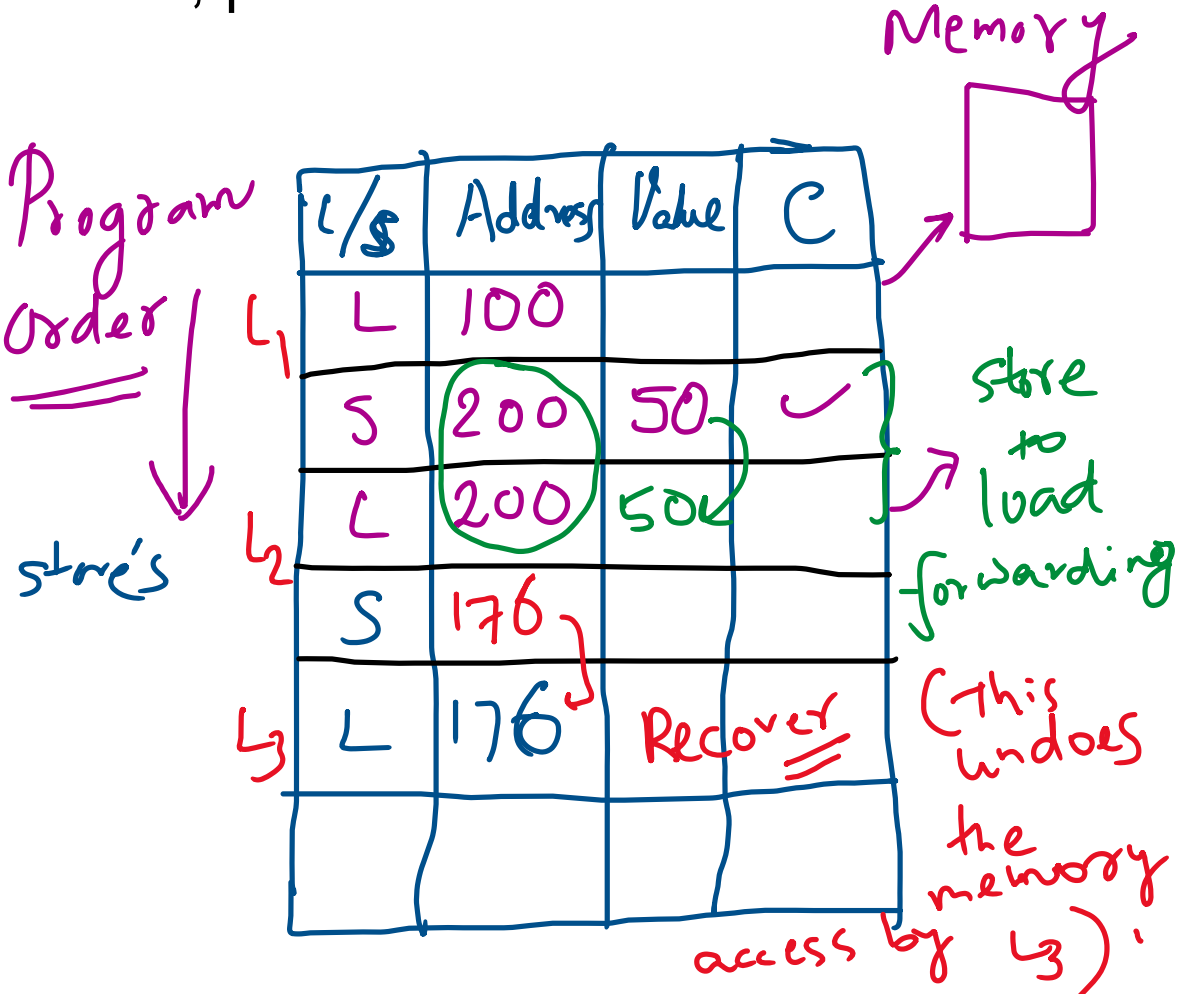
- In-order: Make loads and stores in-order
  - But that means say a load takes a lot of time due to a cache miss all the following load/store instructions wait.
- Wait for all Previous Store Addresses:
  - We check for if address matches with any previous store.
  - If none, load address memory,
  - if yes it matches but store is not ready we wait for it to complete.
- Go Anyway (most aggressive!)
  - When we have the address of the load, we check if it matches with any of the previous stores we wait.
  - If none, we ignore that some may match and just go for the memory.

Recovery

- Two situations can occur:
  - Store address is some address, but not 176 (the load address).
  - In this case, the load was correct.
- What if the store address is 176?
  - In this case, the load can possibly get wrong result
  - It needs to recover!
  - Thus, after the store we check for all following load instructions and matching addresses.
  - We redo the loads.

Most Modern Processors do Go Anyway!

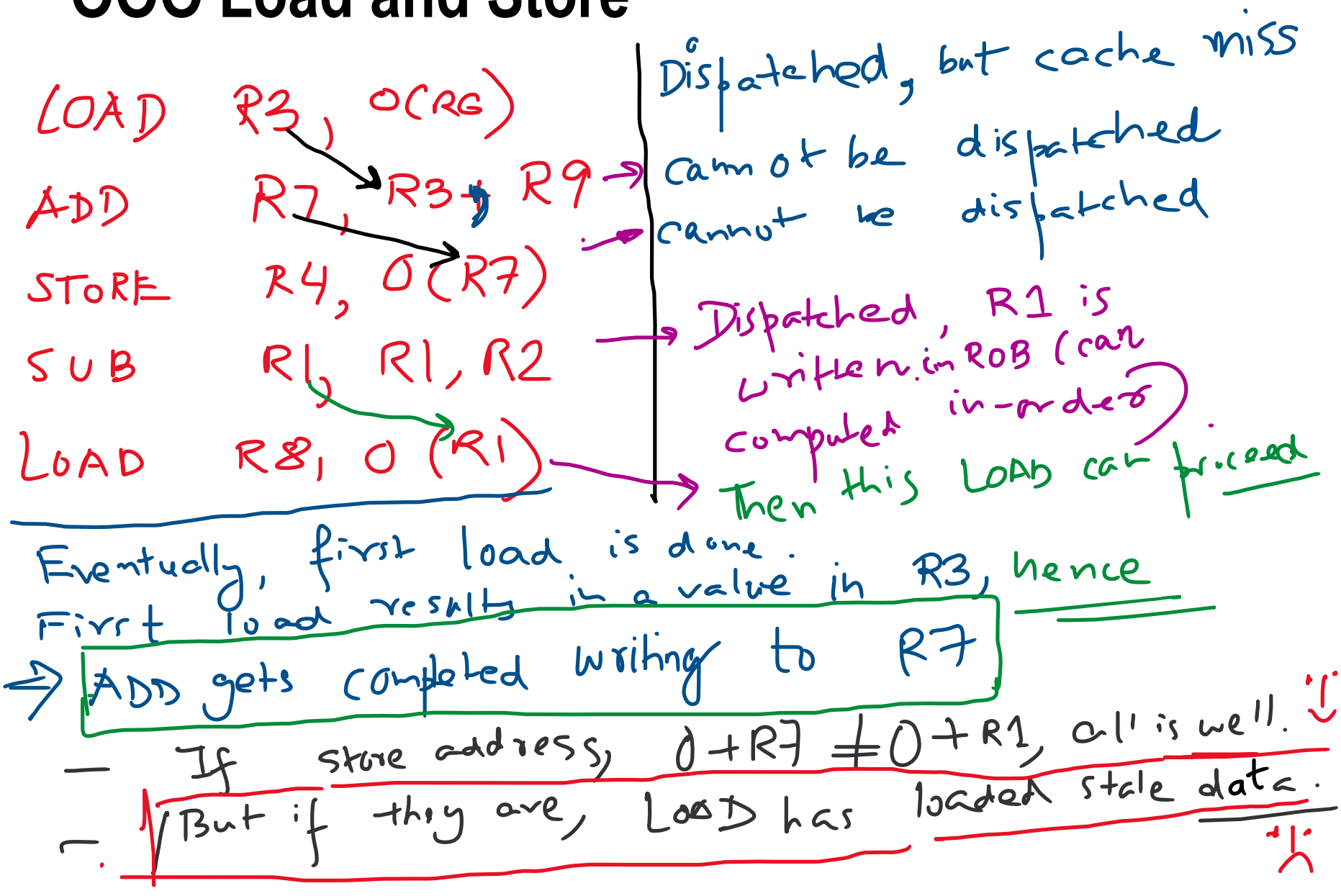
- Based on the fact that most often the address will not match, this strategy is best, performance wise.



OOO Load and Store

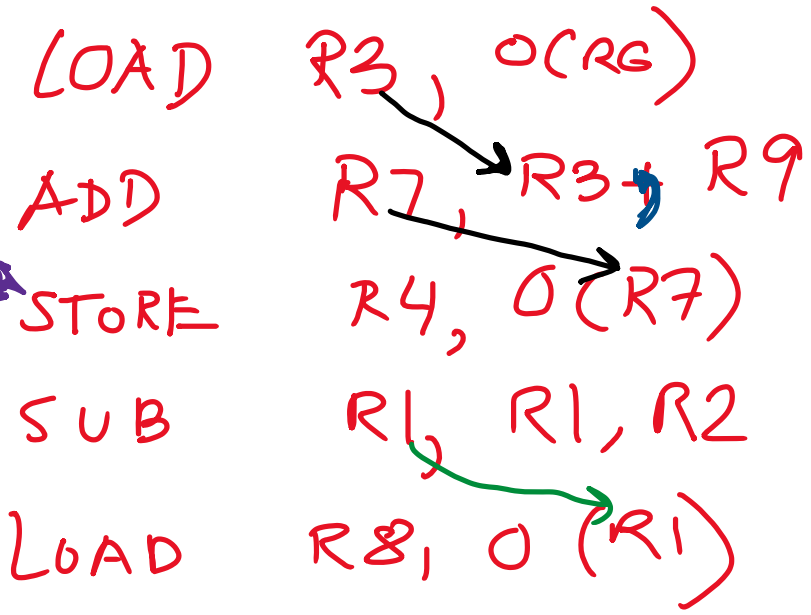
```
LOAD   R3, 0(R6)
ADD    R7, R3+R9
STORE  R4, 0(R7)
SUB    R1, R1, R2
LOAD   R8, 0(R1)
```

OOO Load and Store



In order loads & stores

- The store is considered done when it knows the address and the value of R4 is ready (not when it commits).
- **At this point the load can proceed!**
- It checks for address match with any prior store.
  - Thus, most instructions are Out of order, but loads and stores are in-order!
  - Clearly, not an optimal strategy.
  - Most often (0+R7)≠(0+R1).



↳ We may needlessly delay the load
It can result also in a cache miss, and increase the penalty!

Memory Ordering Quiz

	In Memory	Out Memory
LW R1, 0(R2)		
SW R1, 4(R2)		
LW R1, 0(R3)		
SW R1, 4(R3)		
LW R1, 0(R4)		
SW R1, 4(R4)		

→ We allow O-O load - store execution.
→ All loads are say cache misses, needing 90 cycles.
→ Stores are waiting for preceding loads to write to the registers.

Memory Ordering Quiz

* If say, (4 + R2) = (0 + R3) 2nd LW needs to be redone.

	In Memory	Out Memory
LW R1, 0(R2)	1	41
SW R1, 4(R2)		→ 42 (done)
LW R1, 0(R3)	2	42
SW R1, 4(R3)		→ 43 (done)
LW R1, 0(R4)	3	43
SW R1, 4(R4)		→ 44 (done)

→ We allow O-O load - store execution.
→ All loads are say cache misses, needing 40 cycles.
→ Stores are waiting for preceding loads to write to the registers.

Memory Ordering Quiz

state cycles in OOO in-order execution

	In Memory	Out Memory
LW R1, 0(R2)	1	41
SW R1, 4(R2)	In cycle 42, this store knows R1 & can be considered done.	
LW R1, 0(R3)	43	83
SW R1, 4(R3)	Likewise for cycle 84.	
LW R1, 0(R4)	85	125
SW R1, 4(R4)	126	

Store to Load Forwarding

- Load: Which earlier store do we get the value from?
- Store: Which later load do I give my value to?
- Lets consider an example.

LSQ Example

oldest →

L/S	PC	Seq	Addr	Value
L	0xF048	41773	0x3290	
S	0xF04C	41774	0x3410	
S	0xF054	41775	0x3290	
L	0xF060	41776	0x3418	
L	0xF840	41777	0x3290	
L	0xF858	41778	0x3300	
S	0xF85C	41779	0x3290	
L	0xF870	41780	0x3410	
L	0xF628	41781	0x3290	
L	0xF63C	41782	0x3300	

→ youngest

0x3290	42
0x3300	1
0x3410	38
0x3418	1234

LSQ Example

Commit of load: Writing to Reg file
Commit of Store: Writing to cache.

oldest →

L/S	PC	Seq	Addr	Value
L	0xF048	41773	0x3290	42
S	0xF04C	41774	0x3410	25 [Does not write to cache]
S	0xF054	41775	0x3290	-1
L	0xF060	41776	0x3418	
L	0xF840	41777	0x3290	
L	0xF858	41778	0x3300	
S	0xF85C	41779	0x3290	
L	0xF870	41780	0x3410	
L	0xF628	41781	0x3290	
L	0xF63C	41782	0x3300	

→ Youngest

0x3290	42
0x3300	1
0x3410	38
0x3418	1234

LSQ Example

Commit of load: Writing to Reg file
Commit of Store: Writing to cache.

oldest →

L/S	PC	Seq	Addr	Value
L	0xF048	41773	0x3290	42
S	0xF04C	41774	0x3410	25 [Does not write to cache]
S	0xF054	41775	0x3290	-17
L	0xF060	41776	0x3418	1234
L	0xF840	41777	0x3290	-17 (Forward)
L	0xF858	41778	0x3300	1
S	0xF85C	41779	0x3290	
L	0xF870	41780	0x3410	
L	0xF628	41781	0x3290	
L	0xF63C	41782	0x3300	

match

0x3290	42
0x3300	1
0x3410	38
0x3418	1234

→ Youngest

Before loading check if address matches with any previous store. Here NO. So goto memory.

LSQ Example

Commit of load: Writing to Reg file
Commit of Store: writing to cache.

oldest

L/S	PC	Seq	Addr	Value
L	0xF048	41773	0x3290	42
S	0xF04C	41774	0x3410	25 [Does not write to cache]
S	0xF054	41775	0x3290	-17
L	0xF060	41776	0x3418	1234
L	0xF840	41777	0x3290	-17 (Forward)
L	0xF858	41778	0x3300	1
S	0xF85C	41779	0x3290	0 [Does not write to cache]
L	0xF870	41780	0x3410	25
L	0xF628	41781	0x3290	0
L	0xF63C	41782	0x3300	1

match

youngest

0x3290	42
0x3300	1
0x3410	38
0x3418	1234

before loading check if address matches with any previous store. Here NO. So goto memory.

LSQ Example-Commits

Commit of load: Writing to Reg file
Commit of Store: Writing to cache.

oldest

L/S	PC	Seq	Addr	Value
L	0xF048	41773	0x3290	42
S	0xF04C	41774	0x3410	25
S	0xF054	41775	0x3290	-17
L	0xF060	41776	0x3418	1234
L	0xF840	41777	0x3290	-17
L	0xF858	41778	0x3300	1
S	0xF85C	41779	0x3290	0
L	0xF870	41780	0x3410	25
L	0xF628	41781	0x3290	0
L	0xF63C	41782	0x3300	1

youngest

Register File

0x3290	42 -17 0
0x3300	1
0x3410	38 25
0x3418	1234

write to cache

RF

Commit of store only allows write => if there is an exception we have precise exception

LSQ, ROB, and RS

- Issue Load/Store:
  - A ROB entry
  - A LSQ entry
- Issue Non_Load/Store:
  - A ROB entry
  - A RS
- Execute Load/Store:
  - Compute Address
  - Produce Value
- Write Result (Broadcast)
  - Only Load. Its value is used by following instructions.
  - Store does not write, except in the LSQ
- Commit Load/Store:
  - Free ROB and LSQ entries
- Commit Store:
  - Write data to memory
- Commit Load:
  - Write data to Register File

Quiz

sw R1, 0(R2)

lw R2, 0(R2)

Does lw access Cache or memory?

Quiz

sw R1, 0(R2)

lw R2, 0(R2)

Does lw access Cache or memory?

No!

Quiz

sw R1, 0(R2)

lw R2, 0(R2)

It gets it value from:

- A Result Broadcast
- A Reservation Station
- A ROB entry
- A LSQ entry.

Quiz

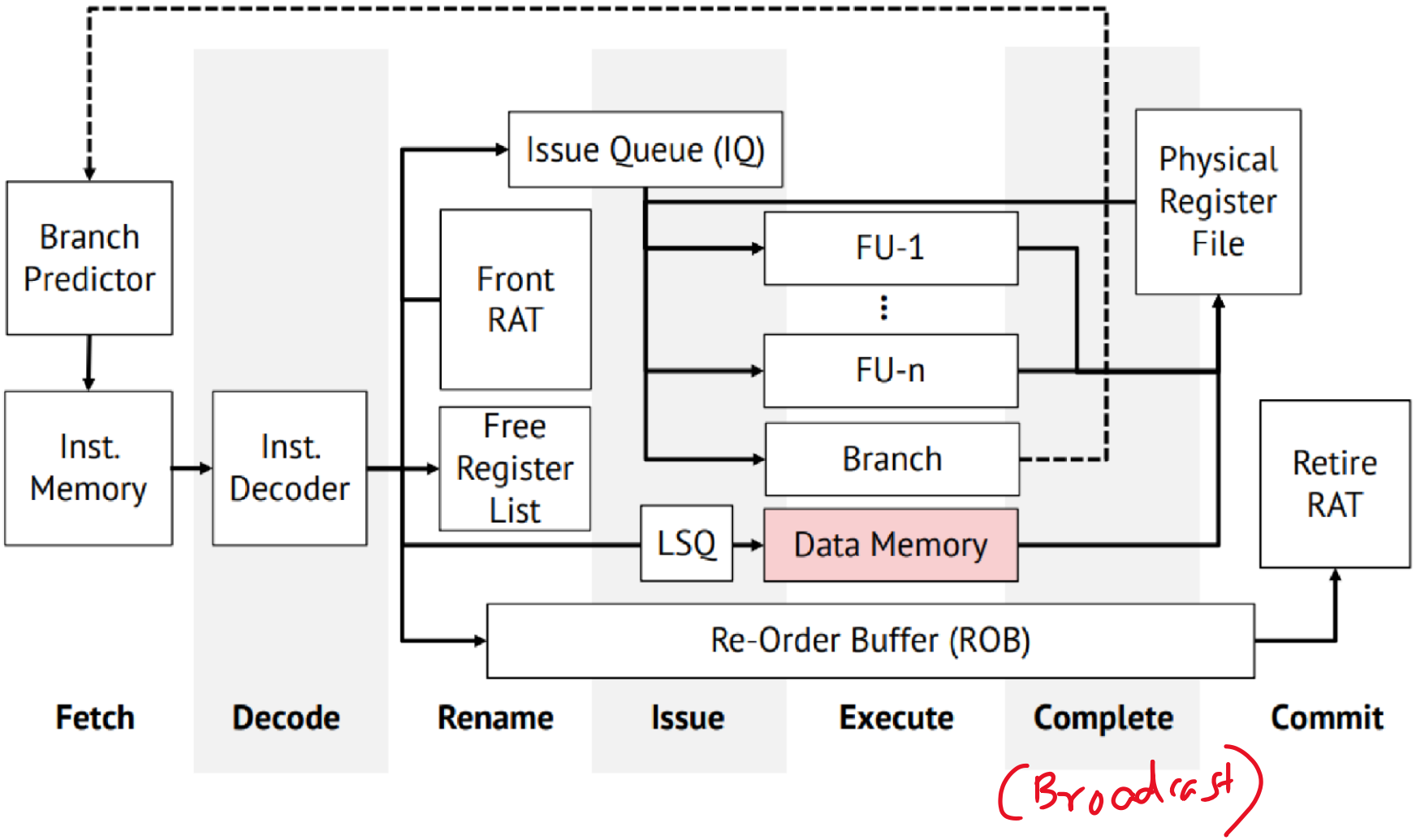
sw R1, 0(R2)

lw R2, 0(R2)

It gets it value from:

- A Result Broadcast
- A Reservation Station
- A ROB entry (as store is not a register producing instruction, it does not write the value in the ROB. It is written only in LSQ for forwarding, and later commit to memory.)
- A LSQ entry

Almost a Modern Processor!



Thank You!