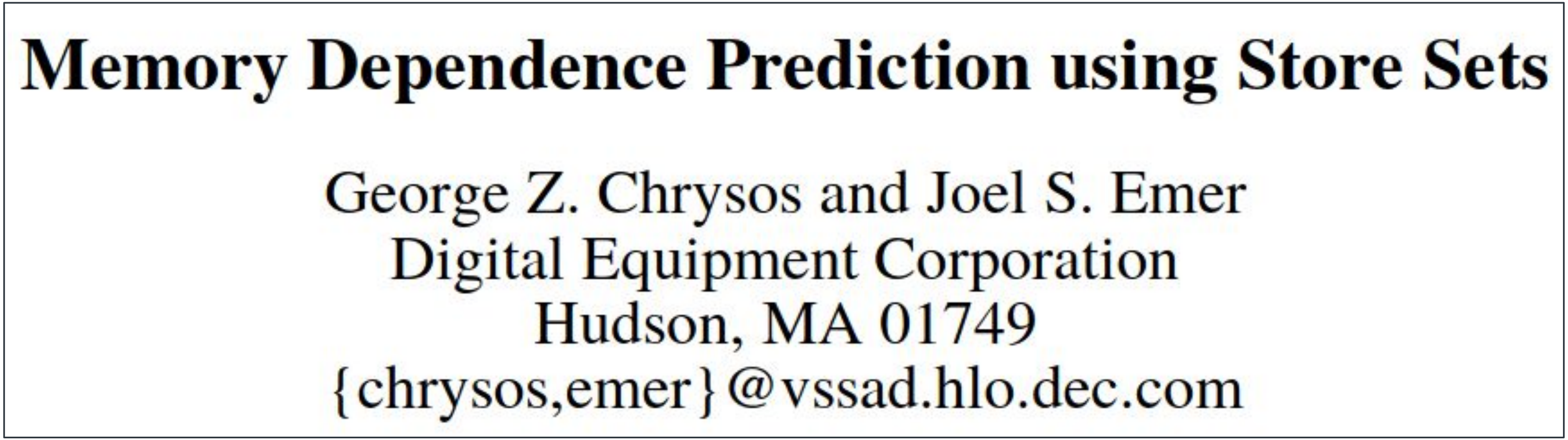
UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Asier Fernández De Lecea
Javier Salamero

Memory Dependence Prediction using Store Sets

Memory Dependence Problem



Memory Dependence Prediction using Store Sets

George Z. Chrysos and Joel S. Emer
Digital Equipment Corporation
Hudson, MA 01749
{chrysos,emer}@vssad.hlo.dec.com

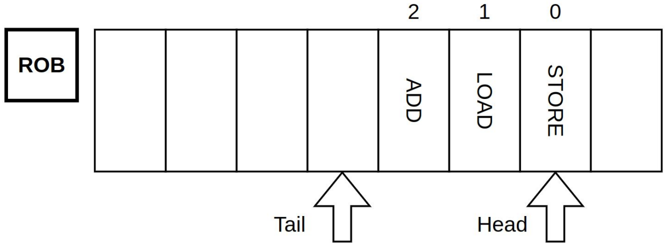
Memory Dependence Problem

Two types of data dependencies (also known as true dependencies or RAW dependencies):

- Register dependencies
  - Can be determined as soon as instructions are decoded

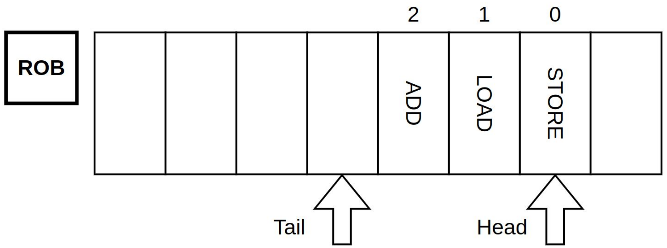
- Memory dependencies:
  - Unknown until addresses are computed (memory disambiguation), a problem in OoO execution schemes

Memory Dependence Problem



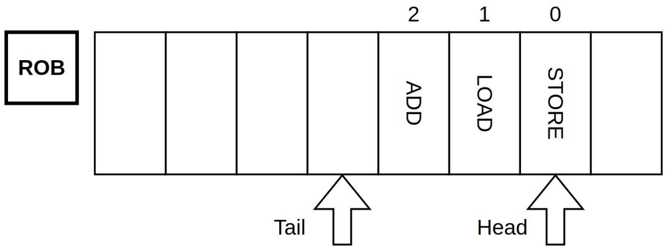
```
STORE   R1  R2   IMM       # mem[R2+IMM] <- R1

 LOAD   R3  R4   IMM       # R3 <- mem[R4+IMM]

  ADD   R5  R3    R3       # R5 <- R3+R3
```

Memory Dependence Problem



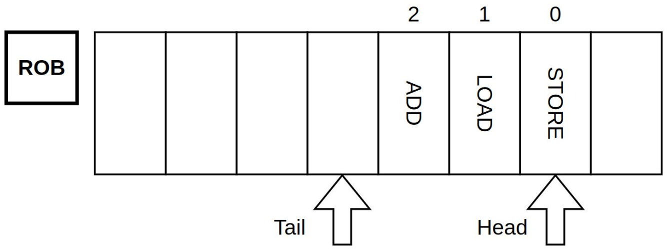
```
STORE   R1  R2   IMM      # mem[R2+IMM] <- R1

LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3   R3       # R5 <- R3+R3
```

Memory Dependence Problem



```
STORE   R1 R2  IMM     # mem[R2+IMM] <- R1
LOAD    R3 R4  IMM     # R3 <- mem[R4+IMM]
ADD     R5 R3  R3      # R5 <- R3+R3
```

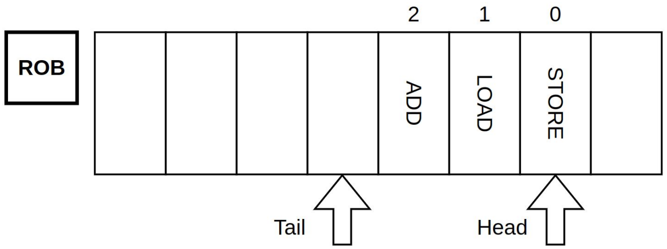
Memory Dependence Problem



It could be the case that both **<u>CALCULATED</u>** addresses interfere with each other…

```
STORE   R1 R2  IMM    # mem[R2+IMM] <- R1

LOAD    R3 R4  IMM    # R3 <- mem[R4+IMM]

ADD     R5 R3  R3     # R5 <- R3+R3
```

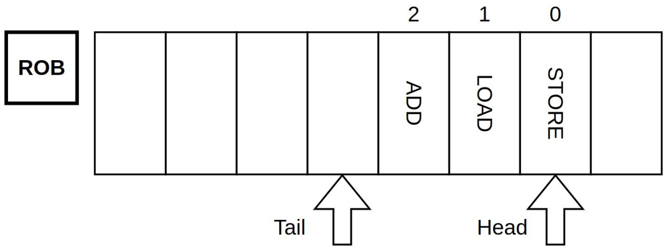
Memory Dependence Problem



Let’s see how that looks…

```
STORE   R1 R2  IMM    # mem[R2+IMM] <- R1

LOAD    R3 R4  IMM    # R3 <- mem[R4+IMM]

ADD     R5 R3  R3     # R5 <- R3+R3
```

Memory Dependence Problem

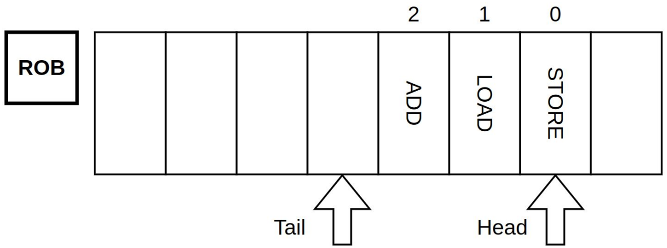


If source operands for address calculation **are ready/known**, two options:...

Ready/known

```
STORE   R1  R2   IMM      # mem[R2+IMM] <- R1

LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3    R3      # R5 <- R3+R3
```

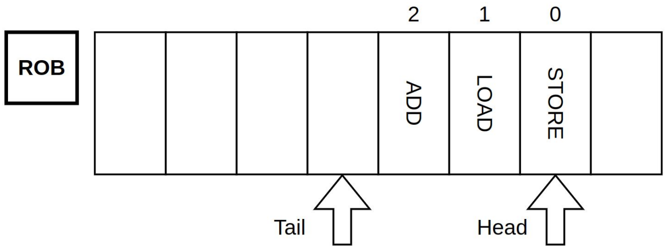
Memory Dependence Problem



Option #1: addresses **do not interfere** with each other…

```
STORE   R1  R2   IMM      # mem[ Addr A ] <- R1

LOAD    R3  R4   IMM      # R3 <- mem[ Addr B ]

ADD     R5  R3   R3       # R5 <- R3+R3
```

Memory Dependence Problem

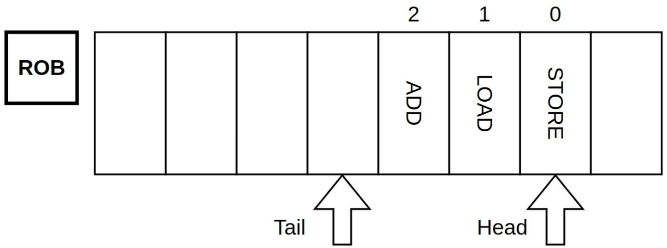


Option #1: addresses **do not interfere** with each other…

Exploit MLP, LOAD-then-ADD

```
                  ✓
STORE   R1  R2   IMM       # mem[Addr A] <- R1
                  ✓
 LOAD   R3  R4   IMM       # R3 <- mem[Addr B]

 ADD    R5  R3    R3       # R5 <- R3+R3
```

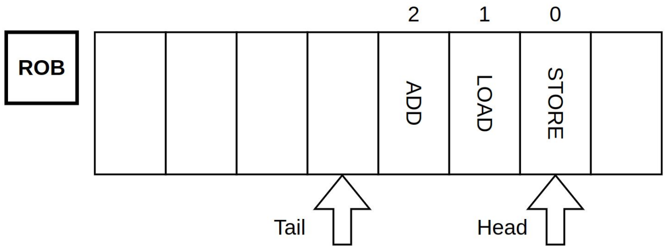
Memory Dependence Problem



Option #2: addresses **DO interfere** with each other…

```
STORE   R1  R2   IMM      # mem[ Addr A ] <-  R1

 LOAD   R3  R4   IMM      # R3  <-  mem[ Addr A ]

  ADD   R5  R3    R3      # R5  <-  R3+R3
```

Memory Dependence Problem

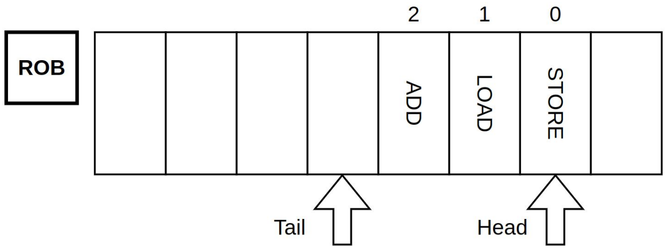


Option #2: addresses **DO interfere** with each other…

STORE, then LOAD, then ADD

```
STORE   R1  R2   IMM        # mem[Addr A] <- R1

LOAD    R3  R4   IMM        # R3 <- mem[Addr A]

ADD     R5  R3   R3         # R5 <- R3+R3
```

Memory Dependence Problem

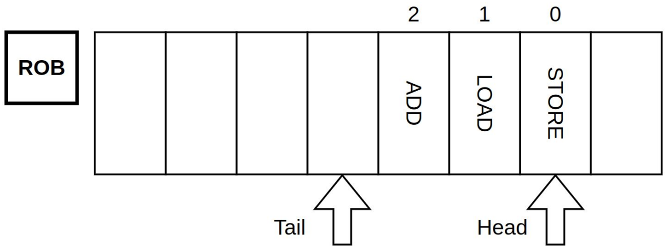


If the source operand of the STORE is not known, but the LOAD's is…

Speculate or don't

```
STORE   R1  R2   IMM        # mem[ Addr ??? ] <- R1
LOAD    R3  R4   IMM        # R3 <- mem[ Addr A ]
ADD     R5  R3   R3         # R5 <- R3+R3
```

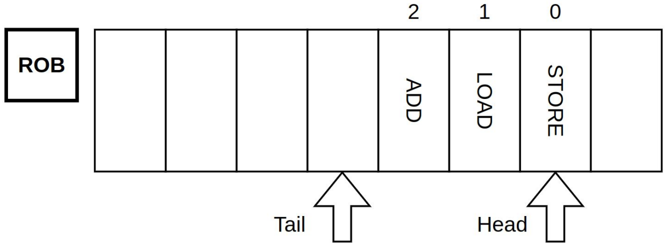
Memory Dependence Problem



No speculation implies that any LOAD **and its dependent instructions** should wait for the STORE’s resolution…

```
STORE   R1  R2   IMM      # mem[ Addr ??? ] <- R1
LOAD    R3  R4   IMM      # R3 <- mem[ Addr A ]
ADD     R5  R3   R3       # R5 <- R3+R3
```

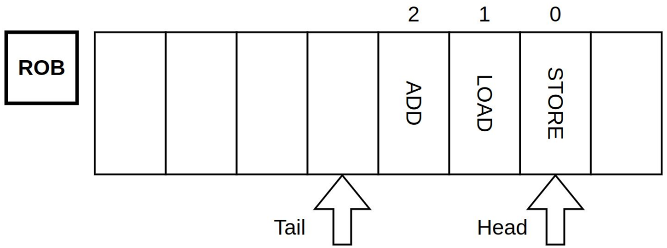
Memory Dependence Problem



…EVEN IF IT JUST SO HAPPENS THAT THERE ARE NO INTERFERENCES!!!

```
STORE   R1  R2   IMM        # mem[ Addr B ]<-  R1
LOAD    R3  R4   IMM        # R3 <-  mem[ Addr A ]
ADD     R5  R3   R3         # R5 <-  R3+R3
```

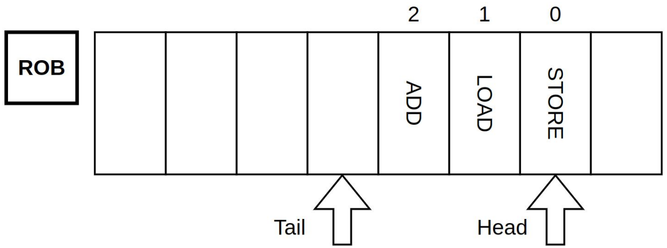
Memory Dependence Problem



Speculation implies that the an address-ready LOAD executes, **hoping no interferences arise afterwards**…

```
STORE   R1  R2 ✗  IMM      # mem[ Addr ??? ] <- R1
              ✓
LOAD    R3  R4    IMM      # R3 <- mem[ Addr A ]
ADD     R5  R3     R3      # R5 <- R3+R3
```

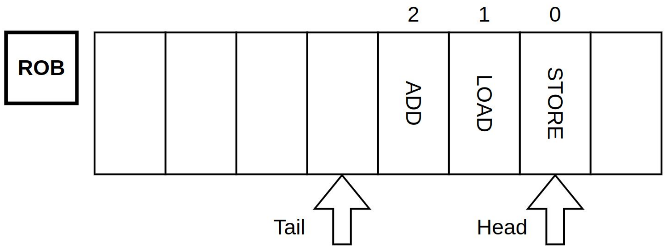
Memory Dependence Problem



If NO interferences, **AWESOME**!!!

```
STORE   R1  R2 ✓  IMM     # mem[ Addr B ] <-  R1
LOAD    R3  R4 ✓  IMM     # R3 <-  mem[ Addr A ]
ADD     R5  R3    R3      # R5 <-  R3+R3
```

Memory Dependence Problem



If YES interferences, **<u>must UNDO</u>** erroneous instructions after the LOAD…

```
                 ✓
🤦 STORE  R1  R2  IMM    # mem[Addr A] <- R1
                 ✓
💩 LOAD   R3  R4  IMM    # R3 <- mem[Addr A]
💩 ADD    R5  R3  R3     # R5 <- R3+R3
```

Background

To face the memory disambiguation problem, there are two conventional approaches:

No speculation: blocking a LOAD until all previous STOREs are ready and issued. This creates many false dependencies, but removes memory-order violations.

- For some LOADs, it is not a problem as this added delay is hidden in OoO execution
- Others are in the critical path, and this conservative approach degrades performance

Background

To face the memory disambiguation problem, there are two conventional approaches:

Naive speculation: LOADs always speculate, assuming no dependencies with any previous STOREs and are marked as ready as soon as register dependencies are fulfilled. Thus, there are no false dependencies...

- LOADs can cause memory-order violations and require a re-execution mechanism to produce the correct results
- Better performance than no-speculation scheme

Background

In **Naive Speculation**, re-execution of LOADs after a memory-order violation can be implemented with different strategies, here are two of them:

- **Trap the LOAD** and squash ALL younger instructions in the pipeline
  - All younger instructions are killed, not only the dependence chain of the LOAD
  - Recovery latency of throwing away all the work done by the discarded instructions can be high

Background

In **Naive Speculation**, re-execution of LOADs after a memory-order violation can be implemented with different strategies, here are two of them:

- ▷ **Replaying only dependent instructions** reduces the penalty, but typically those instructions must remain in the instruction queue (IQ) until the LOAD retires. It has some drawbacks:
  - ○ It fills the IQ with issued instructions waiting until the LOAD commits; if the IQ is full, you cannot extract more ILP
  - ○ Issuing “to-be-replayed” LOADs may delay other useful instructions in the IQ that could issue

Proposal of the paper

This issues reveal the need of memory dependence prediction (MDP), this proposal has the following goals:

1. Predict the LOAD instructions that if allowed to execute would cause a memory-order violation
2. Delay the execution of these LOADs only as long as is necessary to avoid a such a violation.

If the prediction is accurate, the re-execution of the memory order violating LOADs will be very unlikely and it will not have great impact on performance.

Store Set

MDP is based on **the concept of a store set**.

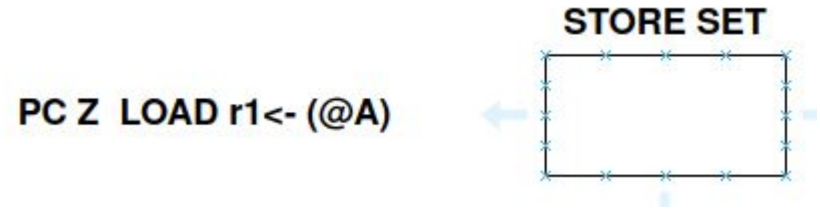
The store set of a LOAD consists of all the STOREs upon which a LOAD has ever depended.

- ▷ STOREs identified by PC
- ▷ Store set will be used to know which STOREs need to be executed before the LOAD

Store Set

Simple example with a concrete LOAD:

1. Initially its store set is empty, and the LOAD always speculates.



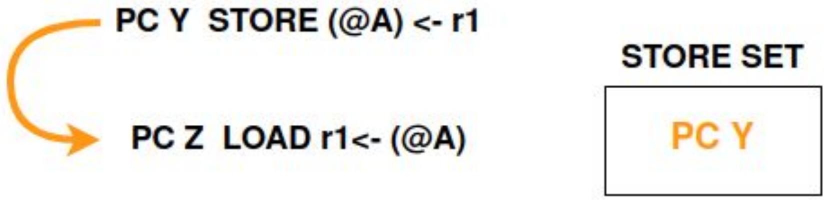
PC Z LOAD r1<- (@A)

STORE SET

Store Set

Simple example with a concrete LOAD:

1. Initially its store set is empty, and the LOAD always speculates.
2. If a STORE detects a memory order violation by any LOAD, the STORE is added to the LOAD's set and the <u>LOAD</u> is re-executed.



PC Y STORE (@A) <- r1

PC Z LOAD r1<- (@A)

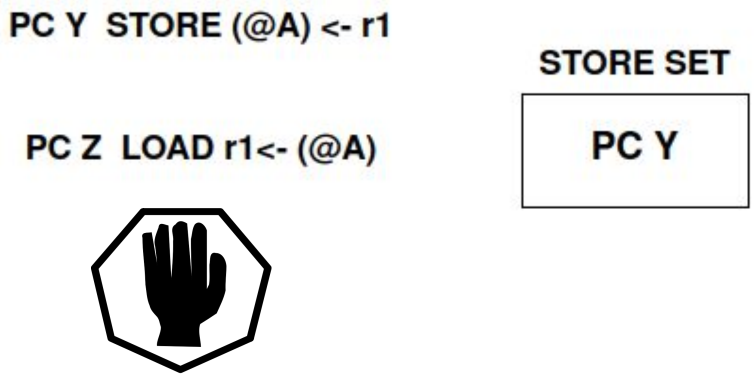
STORE SET

PC Y

Store Set

Simple example with a concrete LOAD:

1. Initially its store set is empty, and the LOAD always speculates.
2. If a STORE detects a memory order violation by any LOAD, the STORE is added to the LOAD's set and the LOAD is re-executed.
3. Every time the LOAD is fetched it will be halted in the IQ until all the recently fetched STOREs in the set are issued before it.



PC Y STORE (@A) <- r1

PC Z LOAD r1<- (@A)

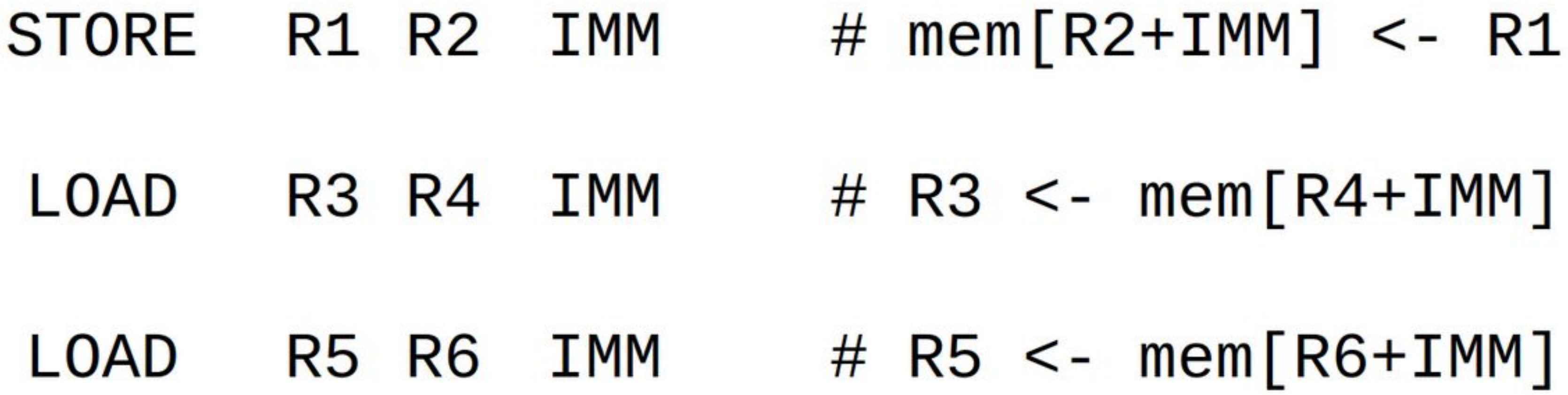
STORE SET

PC Y

Store Set

A STORE could produce a memory dependence:

- In multiple LOADS:

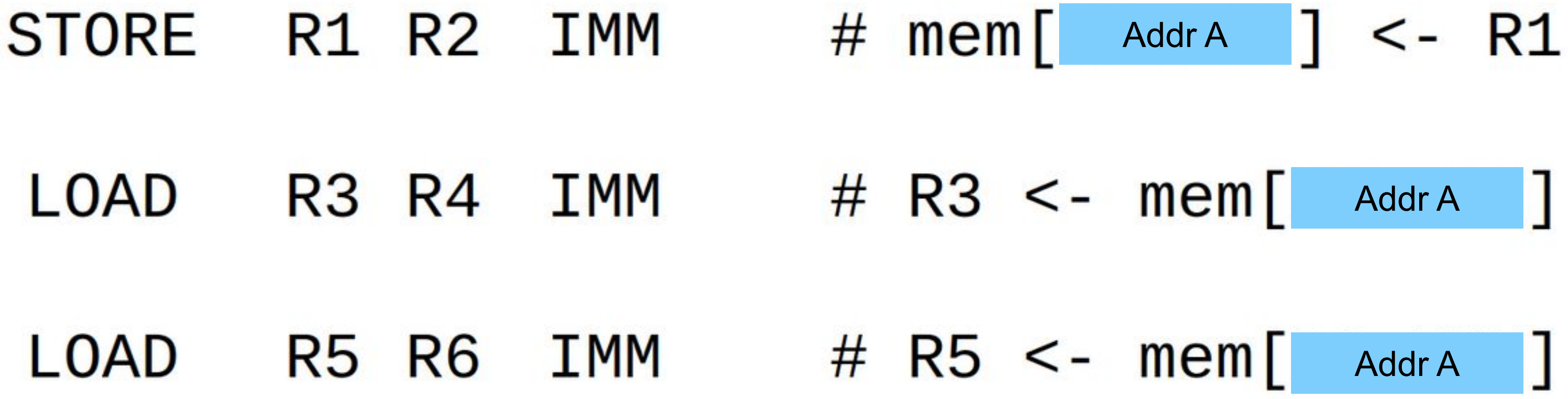


```
STORE   R1  R2   IMM       # mem[R2+IMM] <- R1

 LOAD   R3  R4   IMM       # R3 <- mem[R4+IMM]

 LOAD   R5  R6   IMM       # R5 <- mem[R6+IMM]
```

Store Set

A STORE could produce a memory dependence:

- In multiple LOADS:

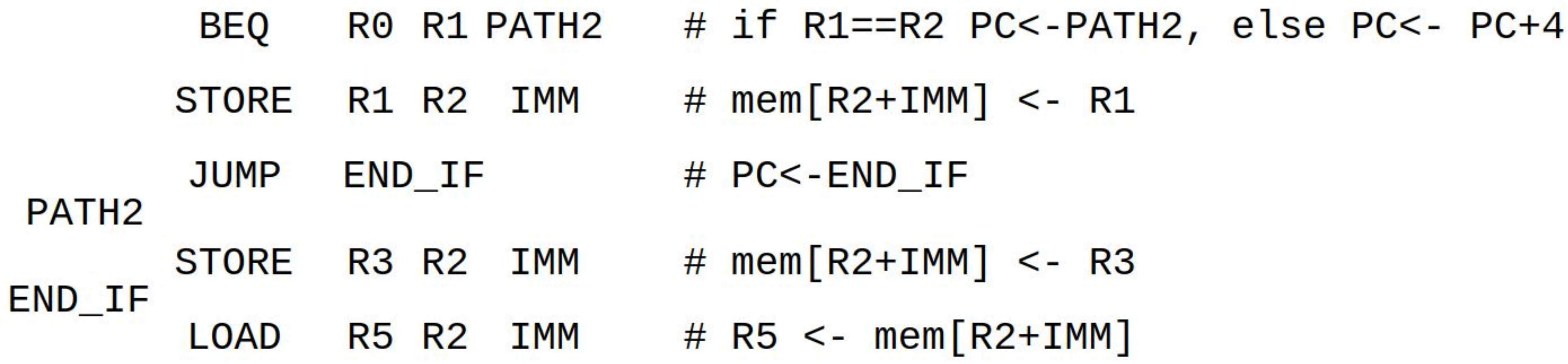


```
STORE  R1  R2  IMM     # mem[ Addr A ] <- R1

LOAD   R3  R4  IMM     # R3 <- mem[ Addr A ]

LOAD   R5  R6  IMM     # R5 <- mem[ Addr A ]
```

Store Set

A STORE could produce a memory dependence:

- Along with other STOREs to the same LOAD address:
  - The LOAD depends on STOREs from different paths

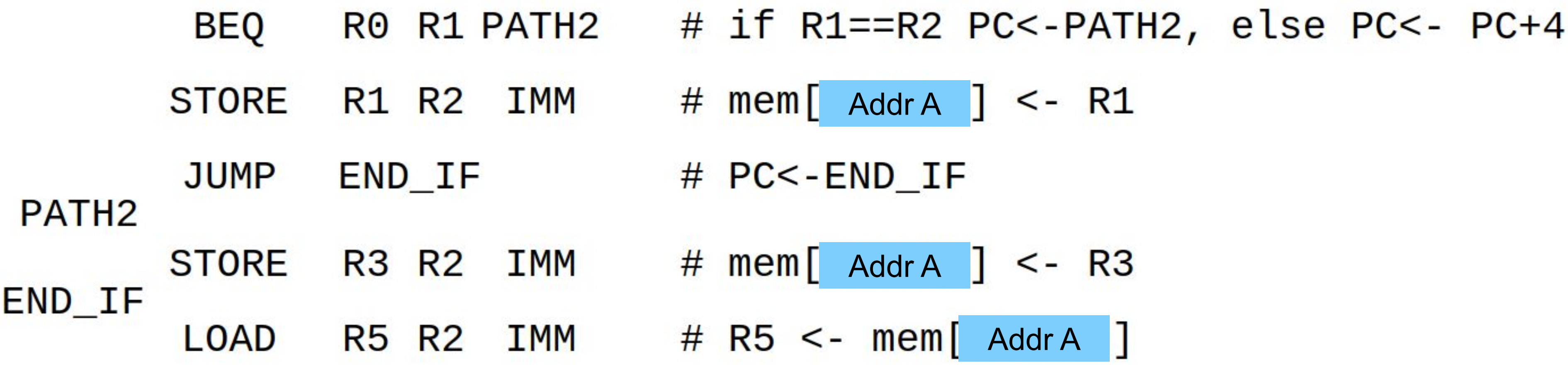


```
       BEQ    R0 R1 PATH2    # if R1==R2 PC<-PATH2, else PC<- PC+4
      STORE   R1 R2  IMM     # mem[R2+IMM] <- R1
       JUMP   END_IF         # PC<-END_IF
PATH2
      STORE   R3 R2  IMM     # mem[R2+IMM] <- R3
END_IF
       LOAD   R5 R2  IMM     # R5 <- mem[R2+IMM]
```

Store Set

A STORE could produce a memory dependence:

- Along with other STOREs to the same LOAD address:
  - The LOAD depends on STOREs from different paths

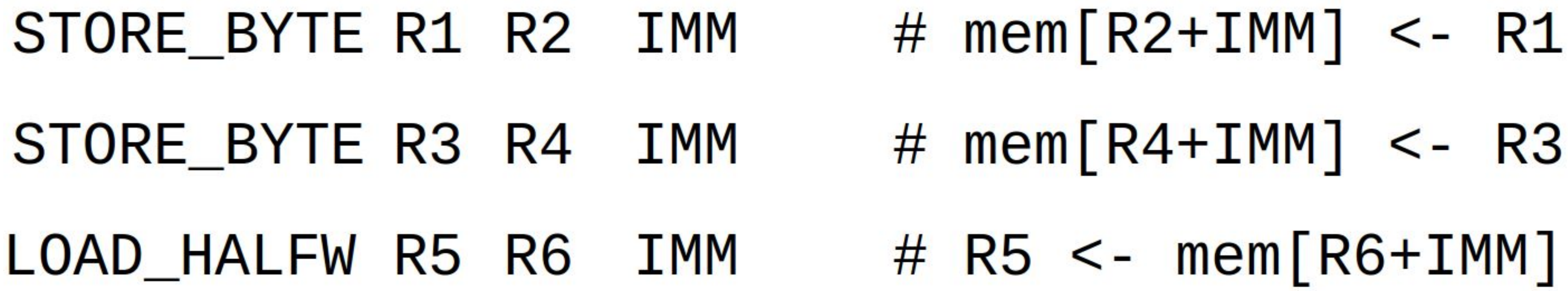


```
       BEQ     R0  R1 PATH2     # if R1==R2 PC<-PATH2, else PC<- PC+4
      STORE    R1  R2  IMM      # mem[ Addr A ] <- R1
       JUMP    END_IF           # PC<-END_IF
PATH2
      STORE    R3  R2  IMM      # mem[ Addr A ] <- R3
END_IF
       LOAD    R5  R2  IMM      # R5 <- mem[ Addr A ]
```

Store Set

A STORE could produce a memory dependence:

- If LOAD depends on multiple STOREs that write in portions of a data word that is read by a single LOAD

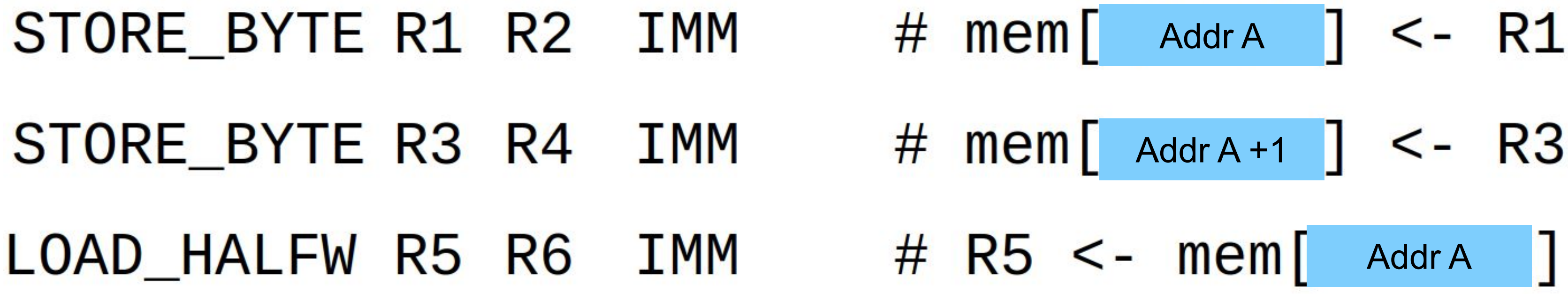


```
STORE_BYTE R1  R2  IMM      # mem[R2+IMM] <- R1
STORE_BYTE R3  R4  IMM      # mem[R4+IMM] <- R3
LOAD_HALFW R5  R6  IMM      # R5 <- mem[R6+IMM]
```

Store Set

A STORE could produce a memory dependence:

- If LOAD depends on multiple STOREs that write in portions of a data word that is read by a single LOAD

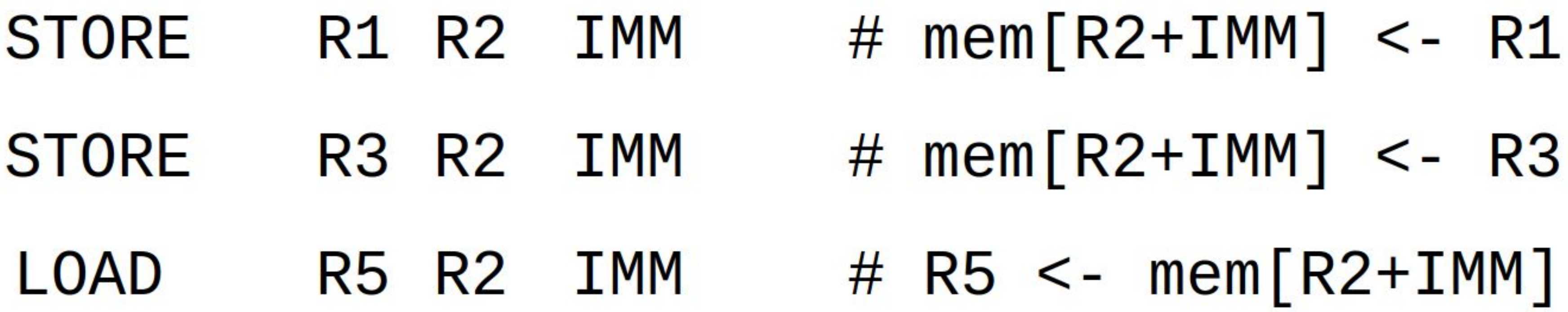


```
STORE_BYTE R1  R2   IMM      # mem[Addr A] <- R1
STORE_BYTE R3  R4   IMM      # mem[Addr A +1] <- R3
LOAD_HALFW R5  R6   IMM      # R5 <- mem[Addr A]
```

Store Set

A STORE could produce a memory dependence:

- If WAW hazards are treated as dependencies, a LOAD can depend on a series of STOREs to the same location

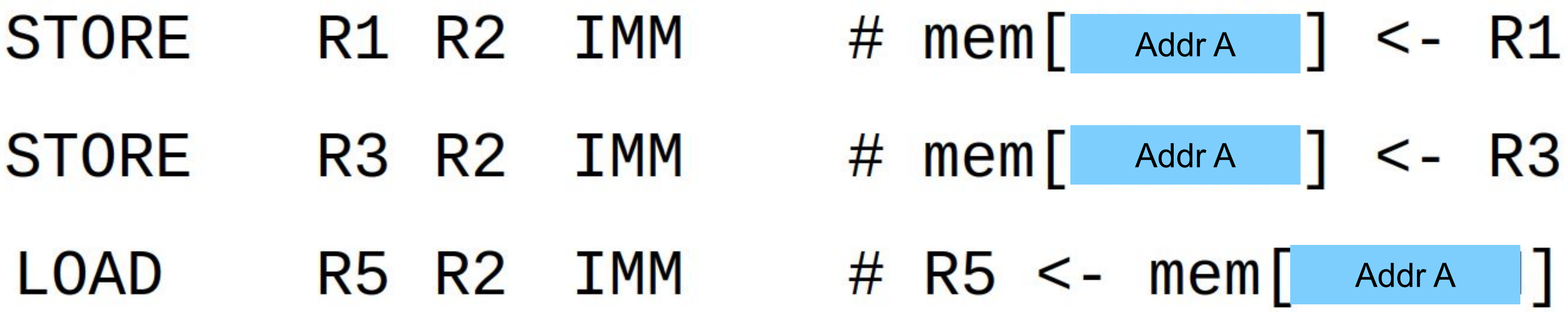


```
STORE    R1  R2   IMM      # mem[R2+IMM] <-  R1
STORE    R3  R2   IMM      # mem[R2+IMM] <-  R3
 LOAD    R5  R2   IMM      # R5 <-  mem[R2+IMM]
```

Store Set

A STORE could produce a memory dependence:

- If WAW hazards are treated as dependencies, a LOAD can depend on a series of STOREs to the same location



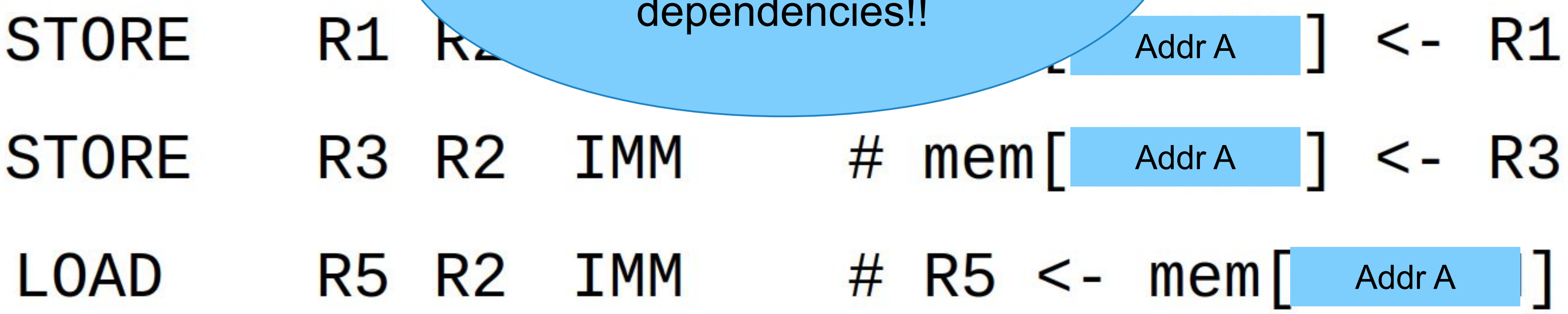
```
STORE    R1  R2   IMM      # mem[ Addr A ] <-  R1
STORE    R3  R2   IMM      # mem[ Addr A ] <-  R3
LOAD     R5  R2   IMM      # R5 <-  mem[ Addr A ]
```

Store Set

A STORE could produce a memory dependence:

- If WAW hazards are t a LOAD can depend on a series of STO

In essence, store sets should be able to account for all these dependencies!!



```
STORE    R1  R               [ Addr A ] <-  R1
STORE    R3  R2  IMM    # mem[ Addr A ] <-  R3
LOAD     R5  R2  IMM    # R5 <-  mem[ Addr A ]
```

Store Set

The concept of MDP with the store set approach is based on two assumptions:

1. The historic behavior of memory-order violations is a good approach to avoid future memory-order violations

2. It is important to predict dependencies of LOADs where one LOAD is dependent on multiple STOREs or multiple LOADs depend on the same STORE

Simulation Environment

At the time of publication of the paper, SoA was Alpha 21264, MIPS R10000, HP-PA8000 and Intel Pentium Pro.

Memory-order violations are only relevant in the context of OoO execution. In particular, in processors of sizeable issue width and large instruction windows.

CPU model used for simulation used double the cache sizes and issue width of the Alpha 21264, as well as a 128-entry IQ…

Simulation Environment

At the time of publication of the paper, SoA was Alpha 21264, MIPS R10000, HP-PA8000 and Intel Pentium Pro.

Memory-order violations are only relev
execution. In particular, in processors of s
instruction windows.

Why such disparity (doubling!) of numbers?

CPU model used for simulation used double the cache sizes and issue width of the Alpha 21264, as well as a 128-entry IQ…

Simulation Environment

At the time of publication of the paper, SoA was Alpha 21264, MIPS R10000, HP-PA8000 and Intel Pentium Pro.

Memory-order violations are only relev
execution. In particular, in processors of s
instruction windows.

Make the problem of memory-order violations as big as possible and show how good this technique is at mitigating it…?

CPU model used for simulation used double the cache sizes and issue width of the Alpha 21264, as well as a 128-entry IQ…

Simulation Environment

At the time of publication of the paper, SoA was Alpha 21264, MIPS R10000, HP-PA8000 and Intel Pentium Pro.

Memory-order violations are only relev
execution. In particular, in processors of s
instruction windows.

Due considerations also as to the representativeness, realism, feasibility of the model's characteristics…

CPU model used for simulation used double the cache sizes and issue width of the Alpha 21264, as well as a 128-entry IQ…

Simulation Environment

At the time of publication of the paper, SoA was Alpha 21264, MIPS R10000, HP-PA8000 and Intel Pentium Pro.

Memory-order violations are only relev
execution. In particular, in processors of s
instruction windows.

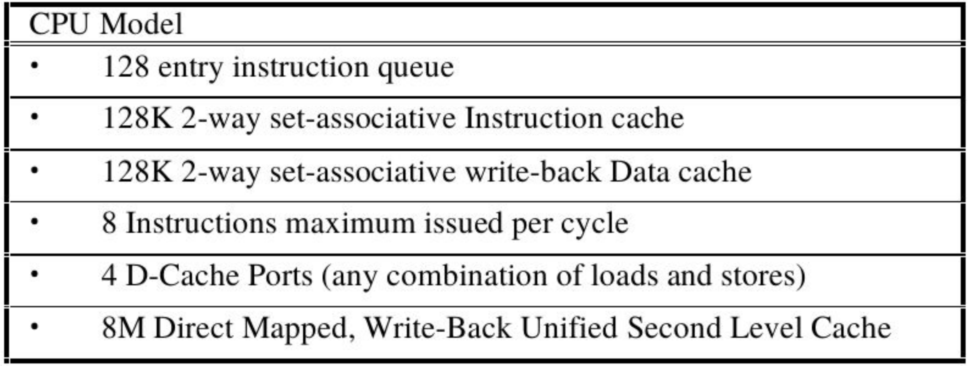
Also, what CPU model? Gem5? Don’t think so…

CPU model used for simulation used double the cache sizes and issue width of the Alpha 21264, as well as a 128-entry IQ…

Simulation Environment

Other CPU model points:

- ▷ Aggressive fetch unit capable of fetching multiple basic blocks in a cycle
- ▷ Large McFarling-style choosing branch predictor



CPU Model
• 128 entry instruction queue
• 128K 2-way set-associative Instruction cache
• 128K 2-way set-associative write-back Data cache
• 8 Instructions maximum issued per cycle
• 4 D-Cache Ports (any combination of loads and stores)
• 8M Direct Mapped, Write-Back Unified Second Level Cache

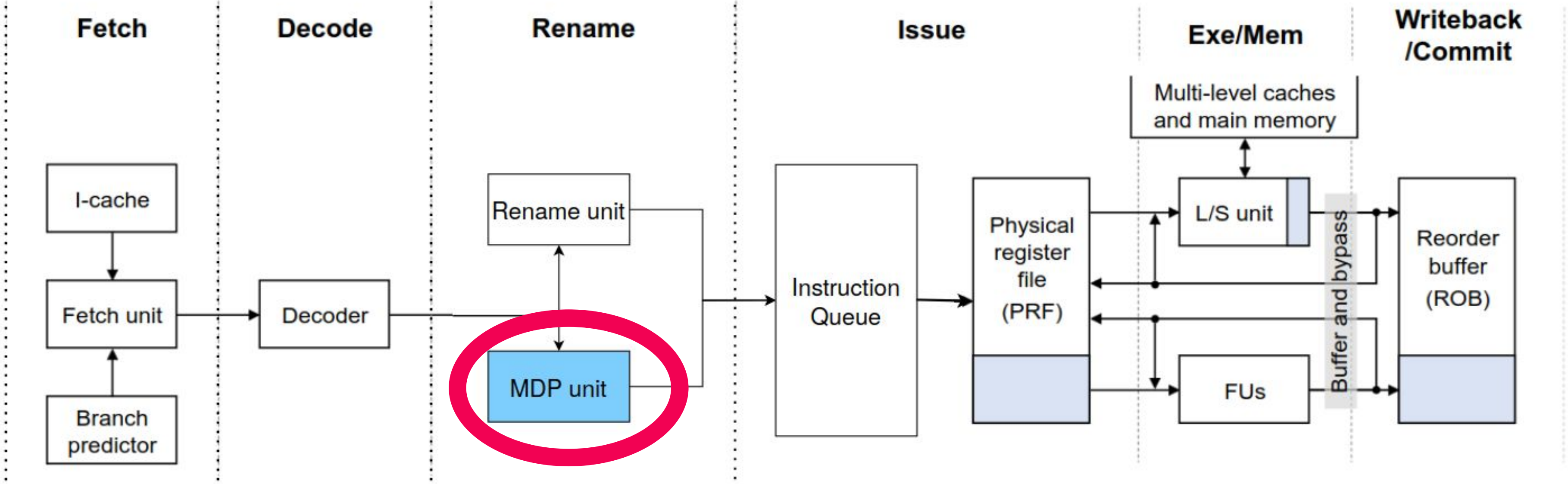
Implementation

Hardware is limited, having an infinite number of store sets (one per each LOAD with an unlimited number of STOREs in each set) is not feasible.

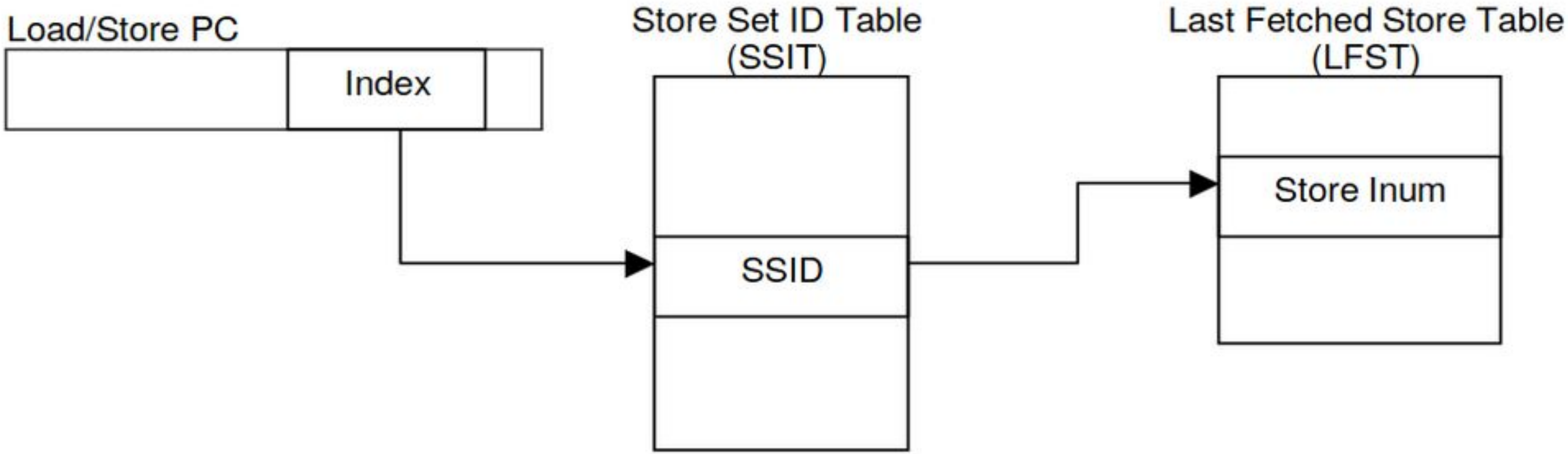
For a low-cost implementation, we need to relax the structure requirements:

- Limited number of store sets, some LOADs will have to share store sets
- A STORE’s PC will be only in a single set
- Store sets will have a single STORE saved

Where does this go?

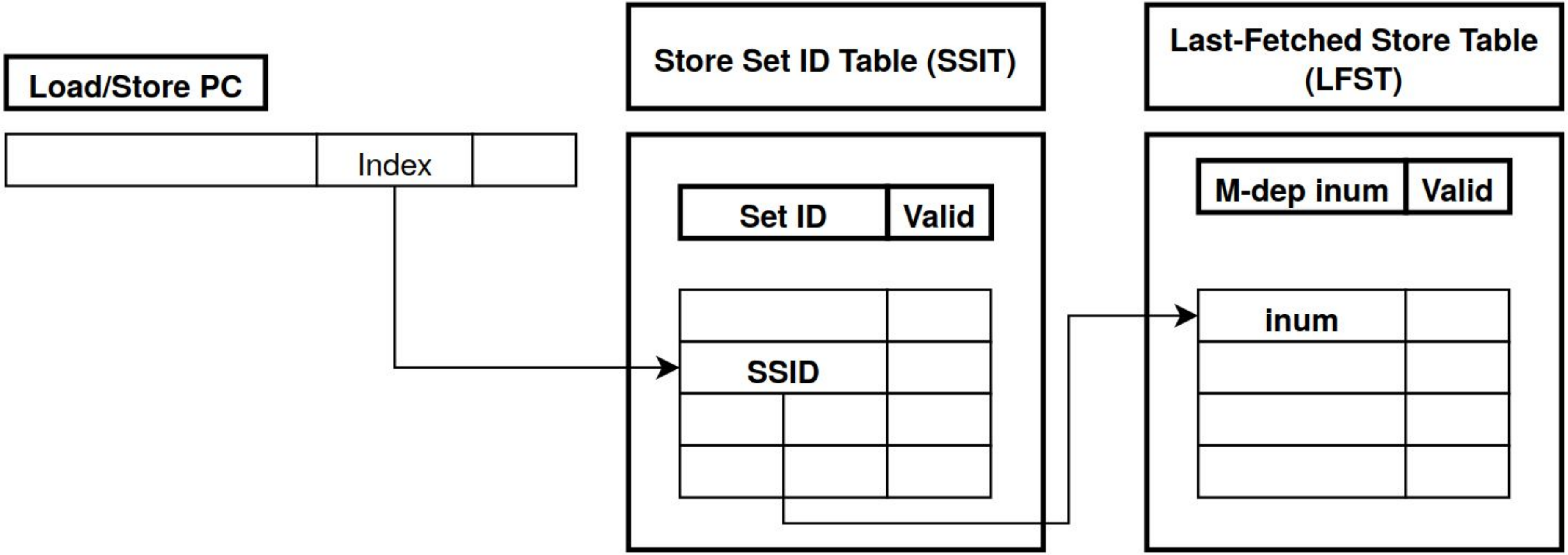


Implementation



Implementation

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

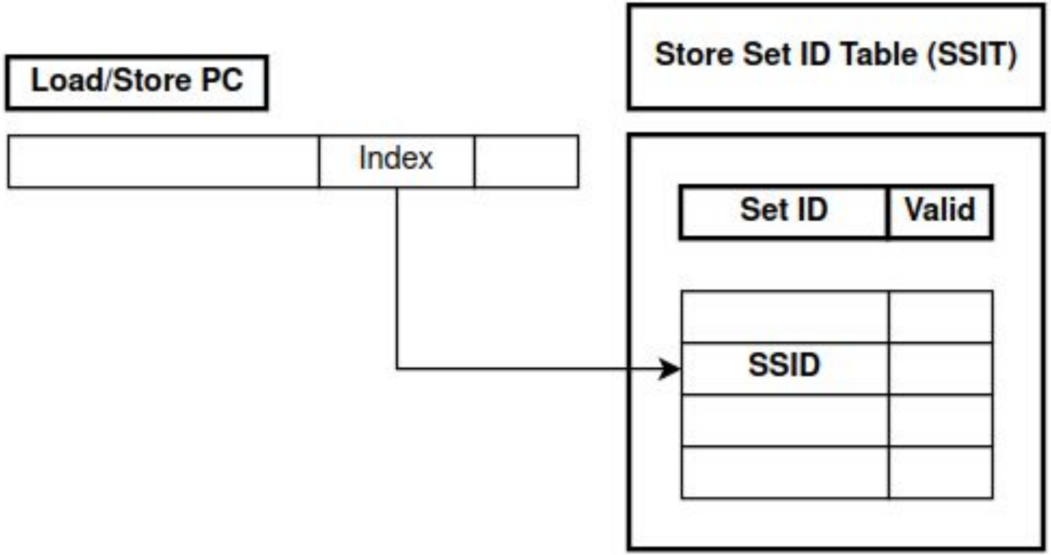


Implementation

The first table of the proposed solution is the Store Set ID Table (SSIT).

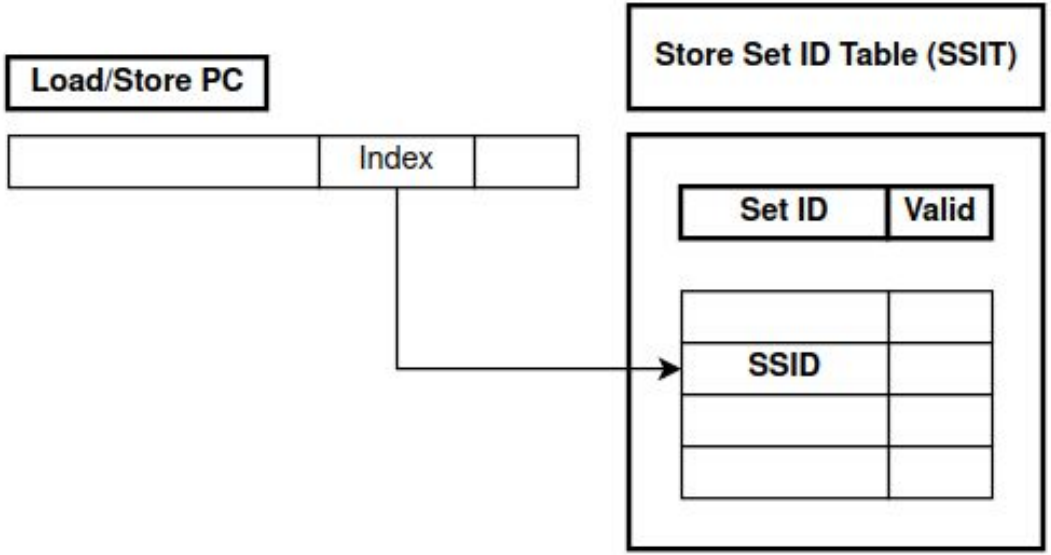
It is indexed by the PC of LOADs/STOREs.

It contains the store set ID to which a LOAD/STORE belongs if it has previously committed a memory-order violation.



Implementation

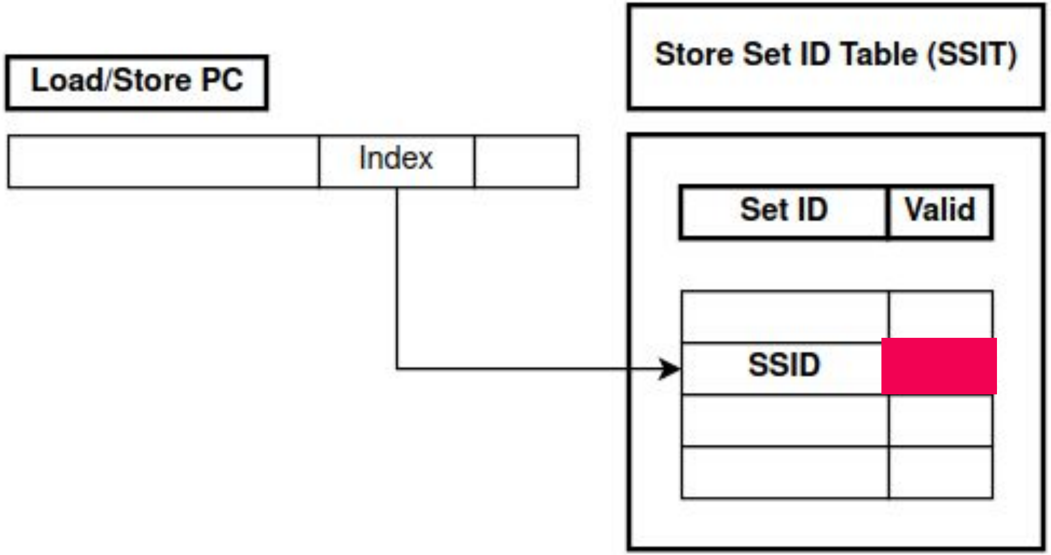
The SSID of a store set is created by XOR-ing the PCs of the first LOAD-STORE pair to commit a memory-order violation.



Implementation

If a LOAD/STORE indexes the SSIT table and finds its entry to be INVALID…

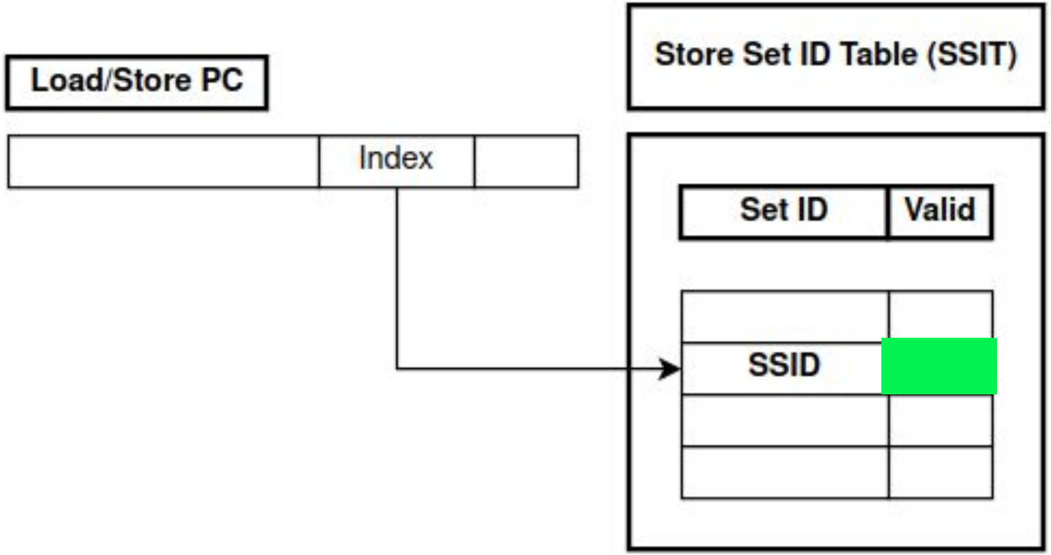
This means that it does not belong to any store set, so there are no memory dependencies to worry about.



Implementation

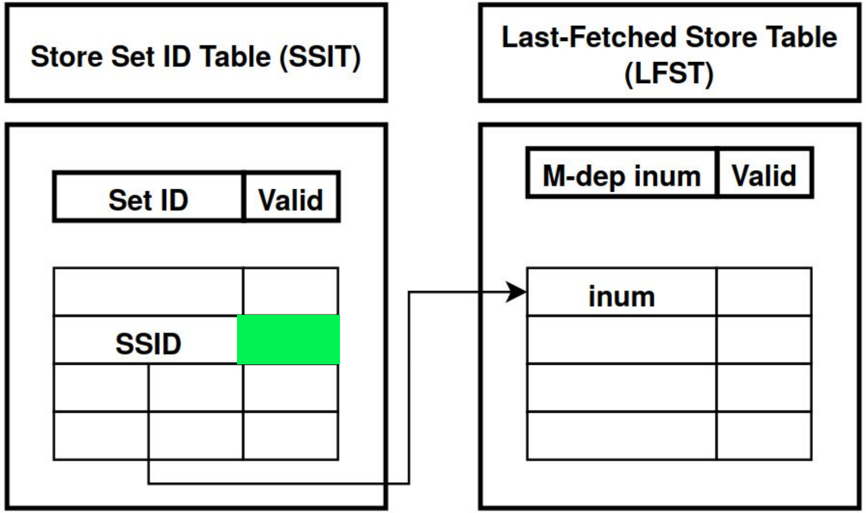
If a LOAD/STORE indexes the SSIT table and finds its entry to be VALID...

This means that it belongs to store set “SSID”, so THERE ARE memory dependencies to worry about.



Implementation

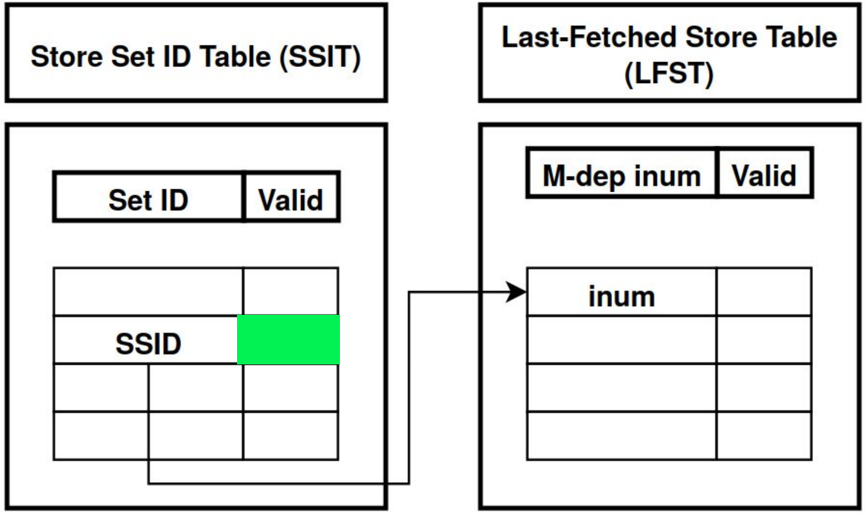
The next table that the paper introduces is the Last-Fetched STORE Table (LFST).



Implementation

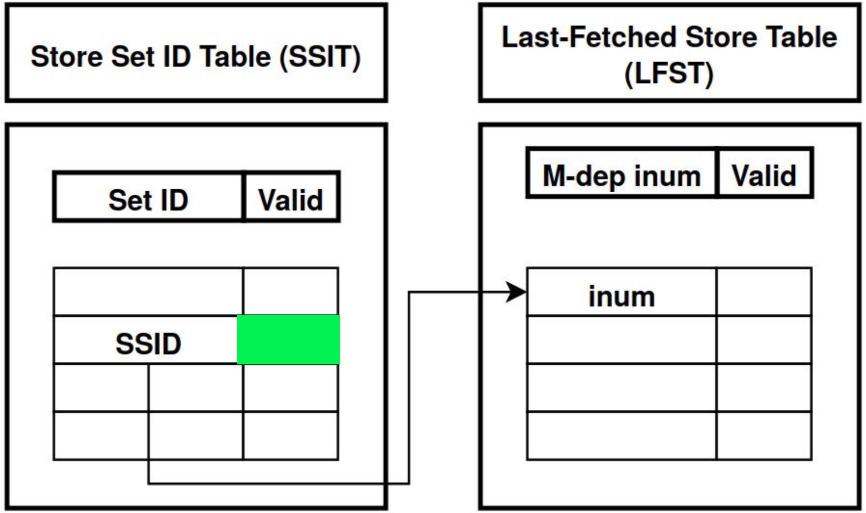
It is indexed by the SSID of a LOAD/STORE that is in a valid STORE Set.

It contains information about the last STORE that was fetched, i.e. which STORE operation is to be waited-for next by the LOAD/STORE instruction.



Implementation

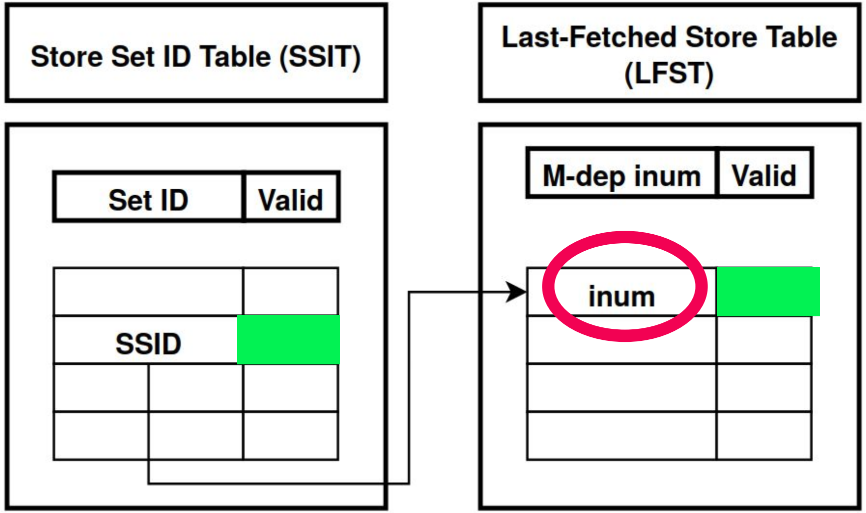
In particular, the LFST contains a unique identifier (we assume ROB ID) of the last STORE to have been fetched.



Implementation

If the LFST entry is VALID…

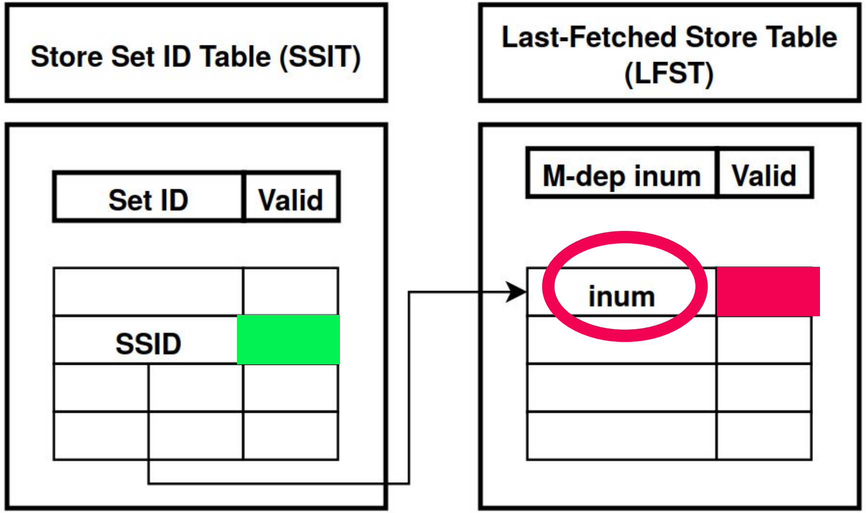
This means that the memory dependence must be respected (needing a wake-up operation), and thus the LOAD/STORE instruction indexing the SSIT will have to wait for the last STORE "inum" to finish.



Implementation

If the LFST entry is INVALID…

This means that there is no memory dependence to be respected, and thus the LOAD/STORE instruction indexing the SSIT is free to go.

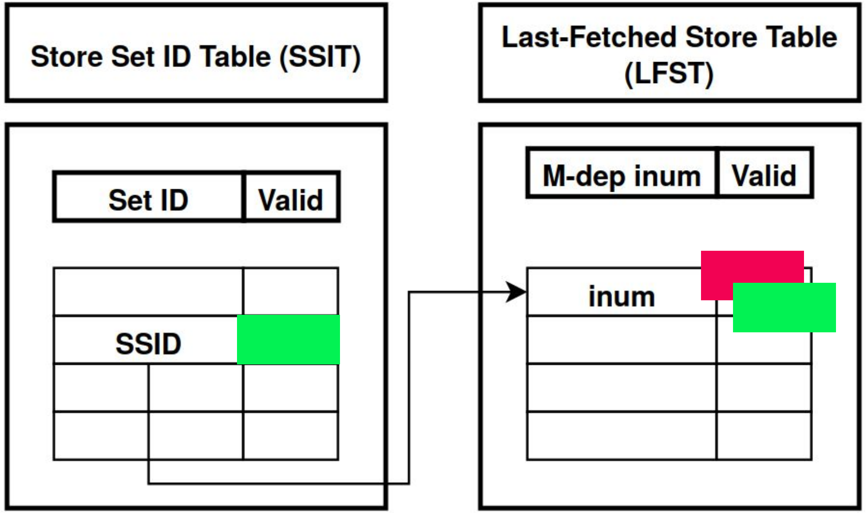


Implementation

In the case of STORE instructions, they replace the existing “inum” with their own (regardless of whether it was valid or invalid), so that the next instruction waits for the…

…Last-Fetched…

…STORE instruction!



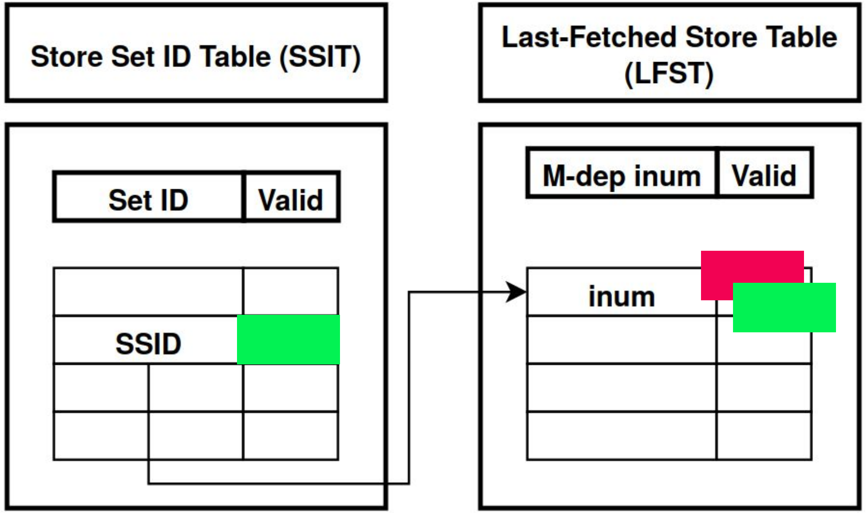
Implementation

In the case of STORE instructions, they replace the existing “inum” with their own (regardless of whether it was valid or invalid), so that the next instruction waits for the…

…<u>Last-Fetched</u>…

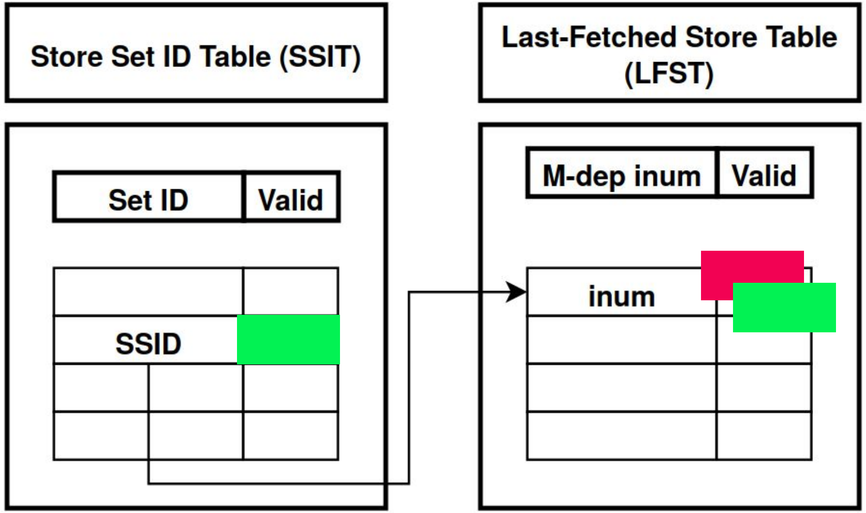
…STORE instruction!

(don’t forget to validate the LFST entry too…)

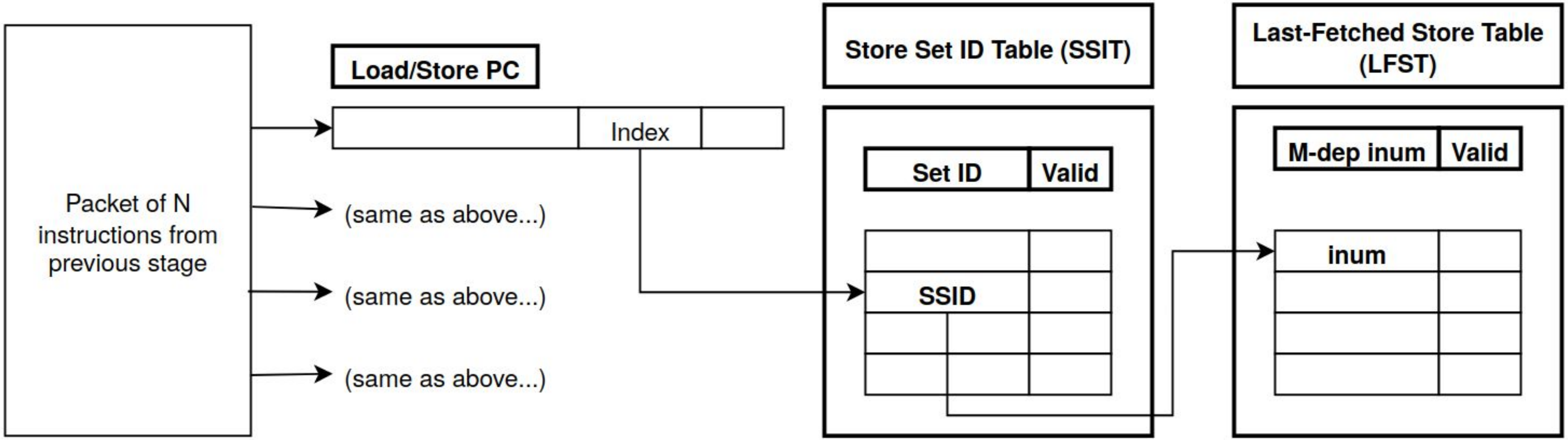


Implementation

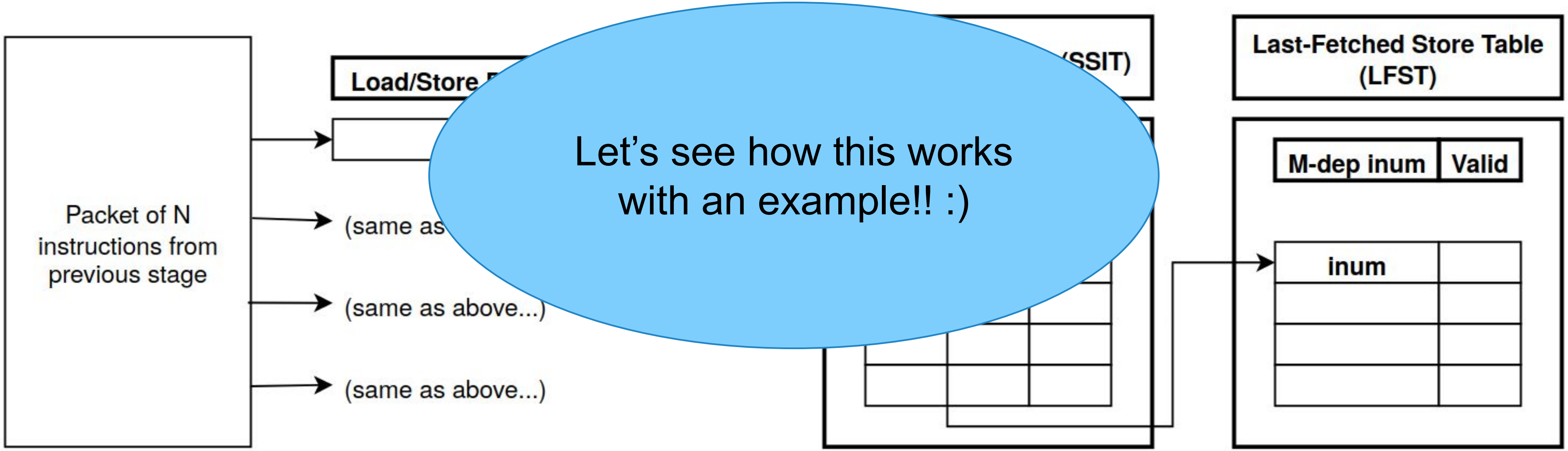
Later, in the IQ, M-dependencies will need the same wake-up logic as with R-dependencies, but now with this "inum" business...



Implementation

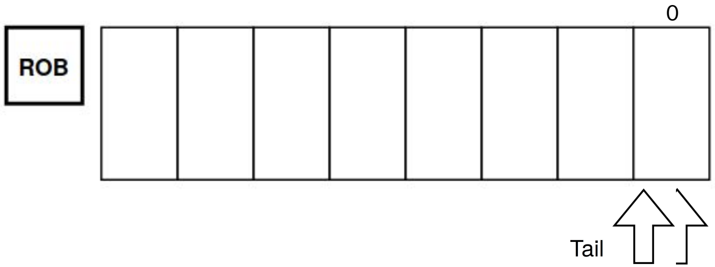


Implementation



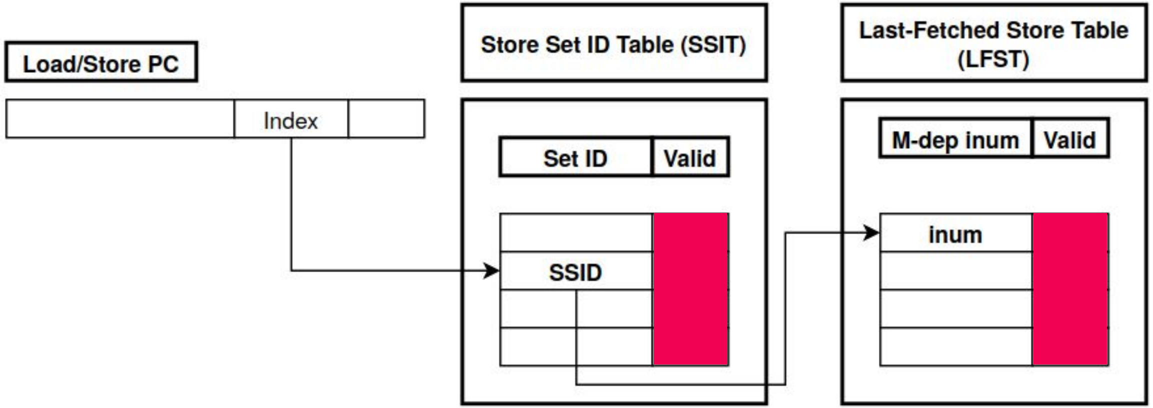
Let’s see how this works with an example!! :)

```
STORE   R1  R2   IMM     # mem[R2+IMM] <- R1
 (…)
 LOAD   R3  R4   IMM     # R3 <- mem[R4+IMM]

 ADD    R5  R3   R3      # R5 <- R3+R3
```

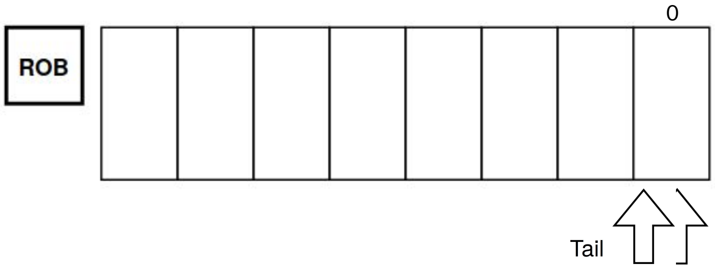


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready

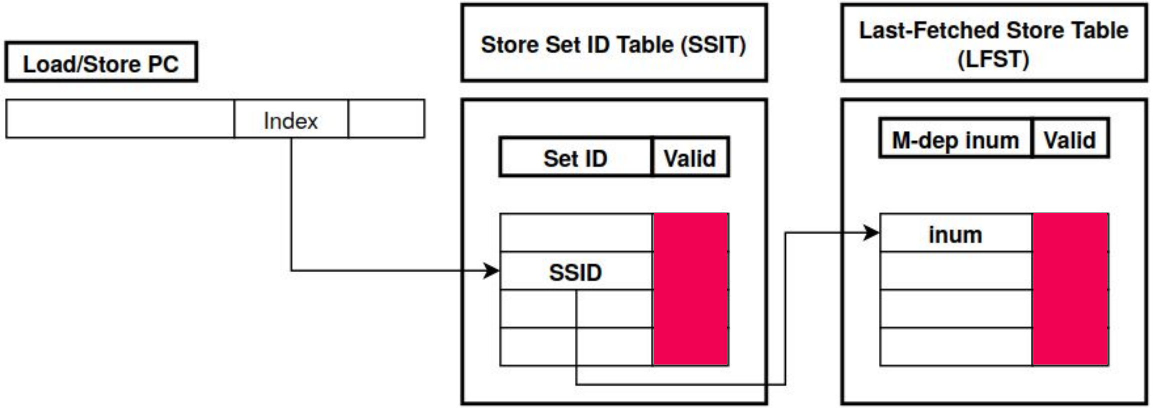


```
➡ STORE   R1  R2   IMM      # mem[R2+IMM] <- R1
   (…)
   LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

   ADD     R5  R3   R3       # R5 <- R3+R3
```

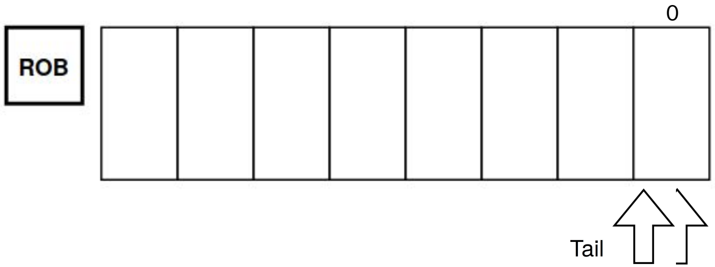


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready

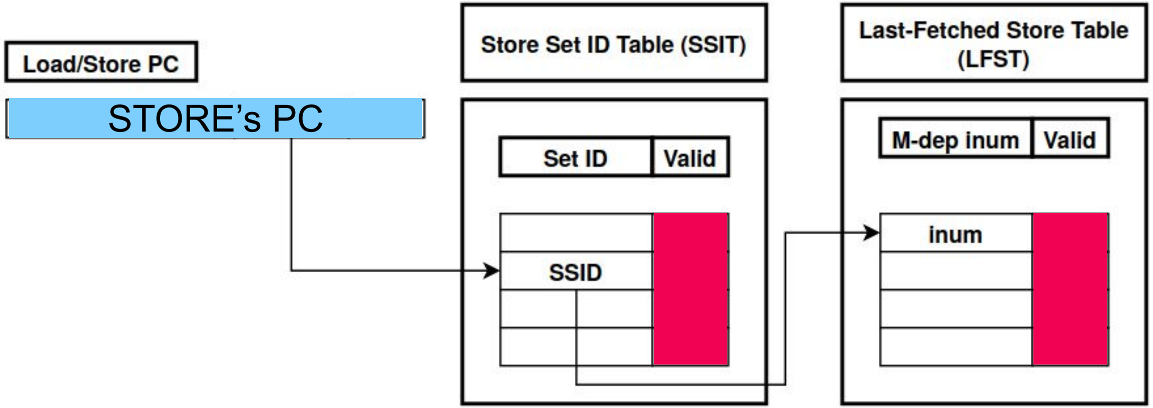


```
STORE   R1  R2   IMM      # mem[R2+IMM] <- R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3   R3       # R5 <- R3+R3
```

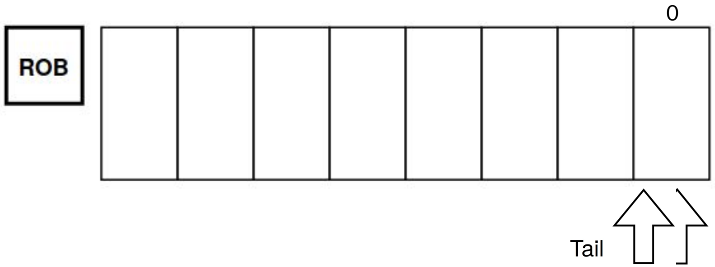


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready

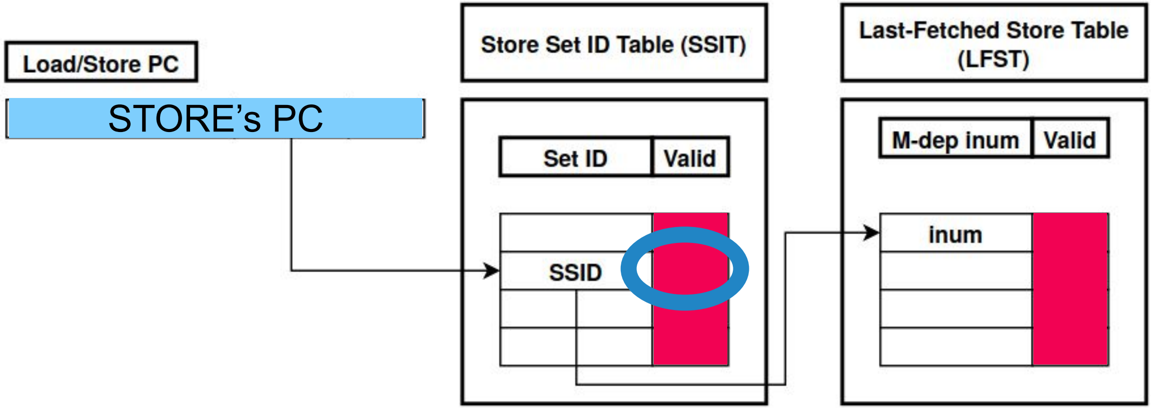


```
STORE   R1  R2  IMM     # mem[R2+IMM] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3  R3      # R5 <- R3+R3
```

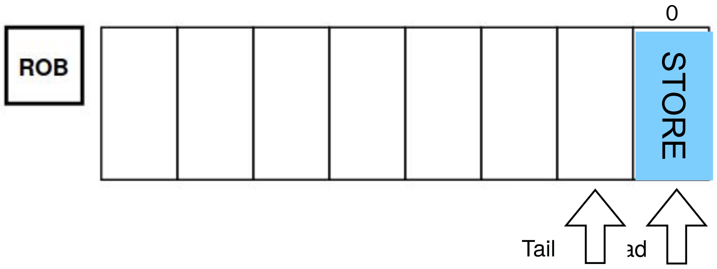


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready

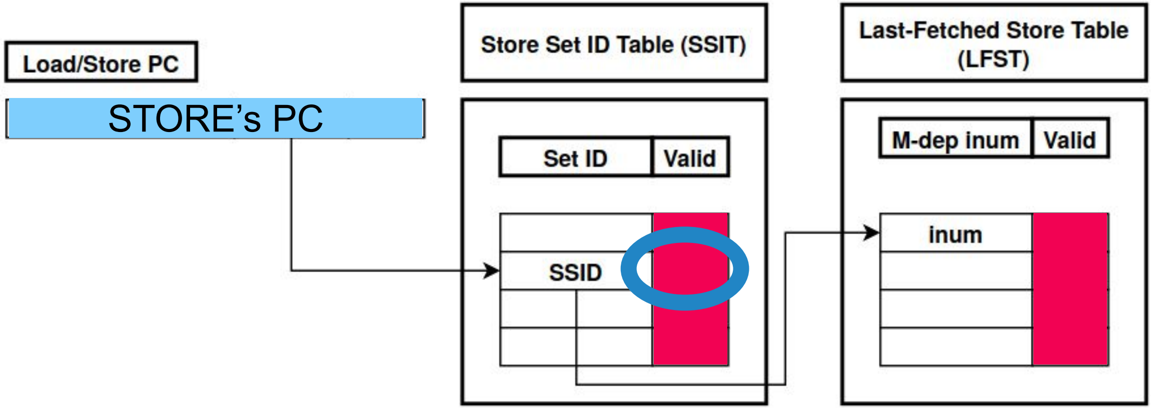


```
STORE   R1  R2   IMM      # mem[R2+IMM] <- R1
 (…)
 LOAD   R3  R4   IMM      # R3 <- mem[R4+IMM]

 ADD    R5  R3   R3       # R5 <- R3+R3
```

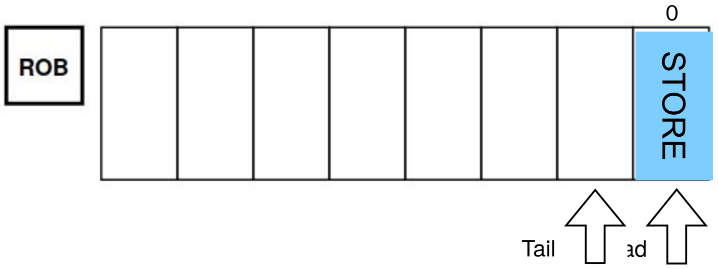


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩

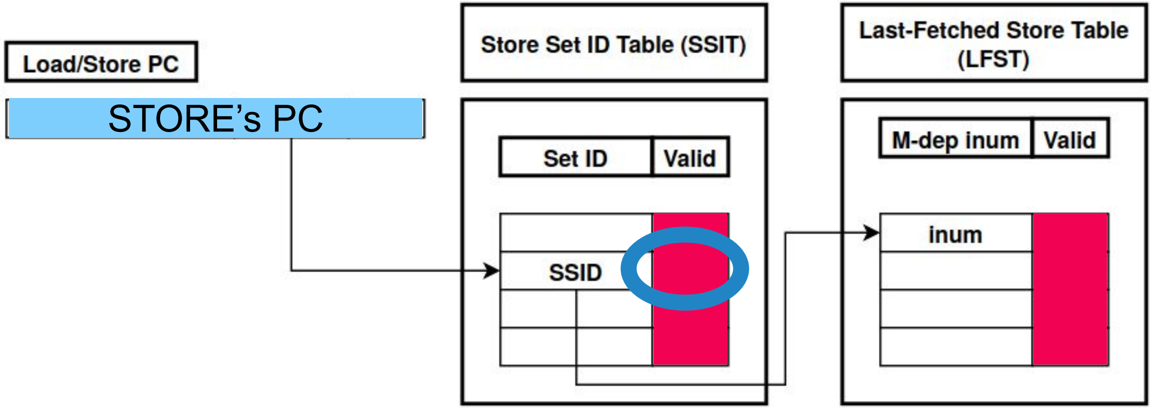


```
STORE   R1  R2  IMM     # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3   R3     # R5 <- R3+R3
```

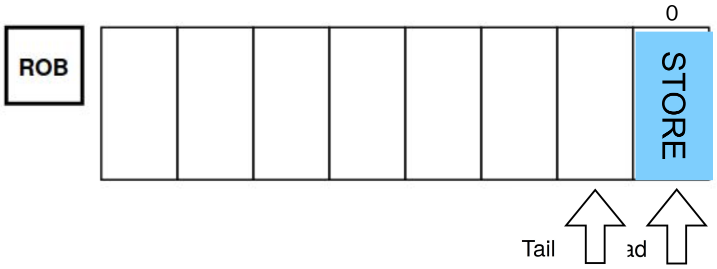


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩

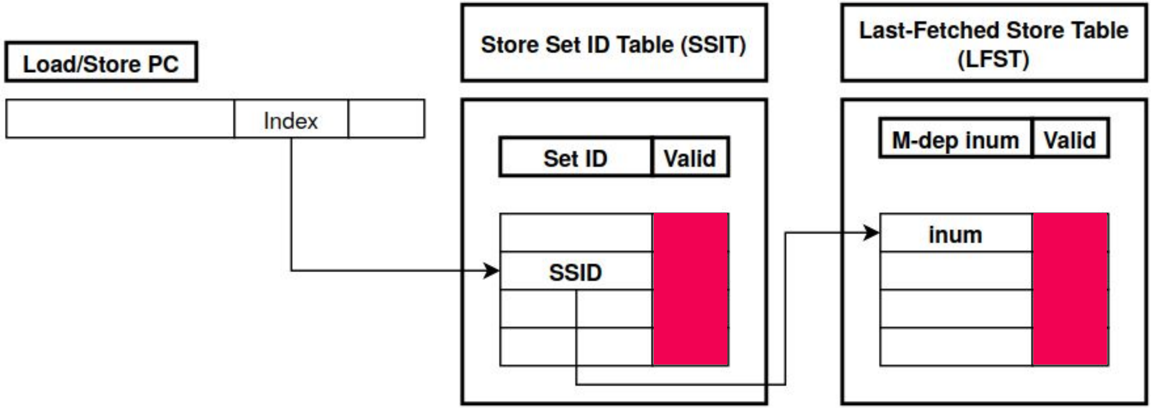


```
STORE   R1  R2   IMM      # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3   R3       # R5 <- R3+R3
```

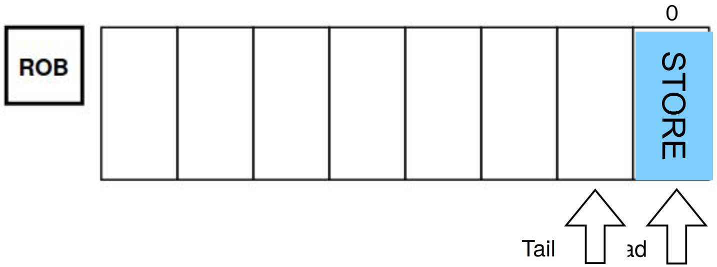


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩



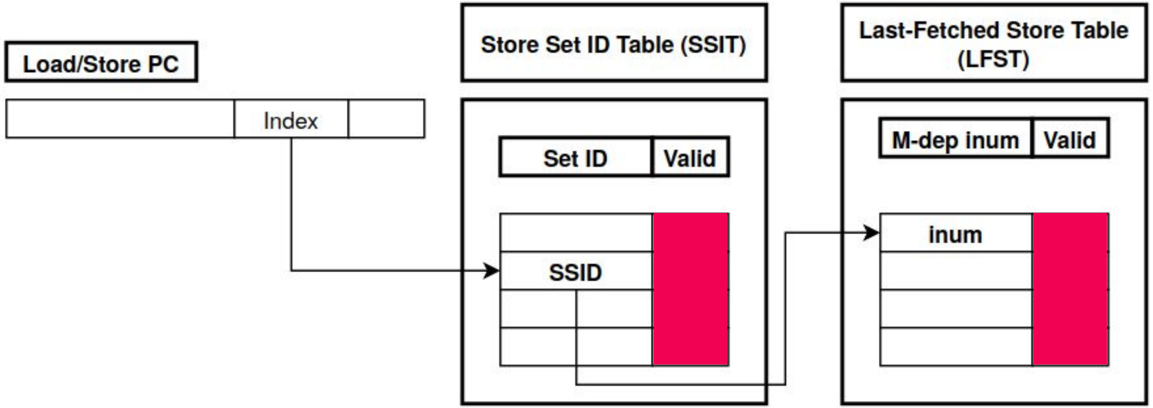
```
STORE   R1  R2  IMM     # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3   R3     # R5 <- R3+R3
```



OoO Issue Queue

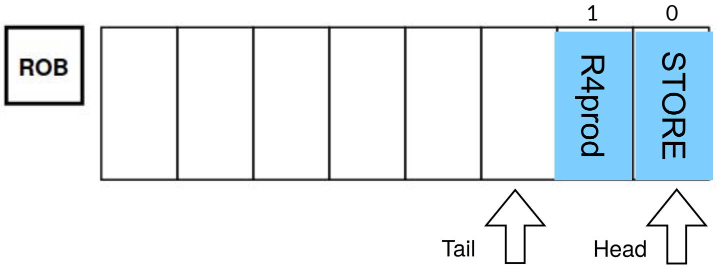
Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩

R4-producer instr.



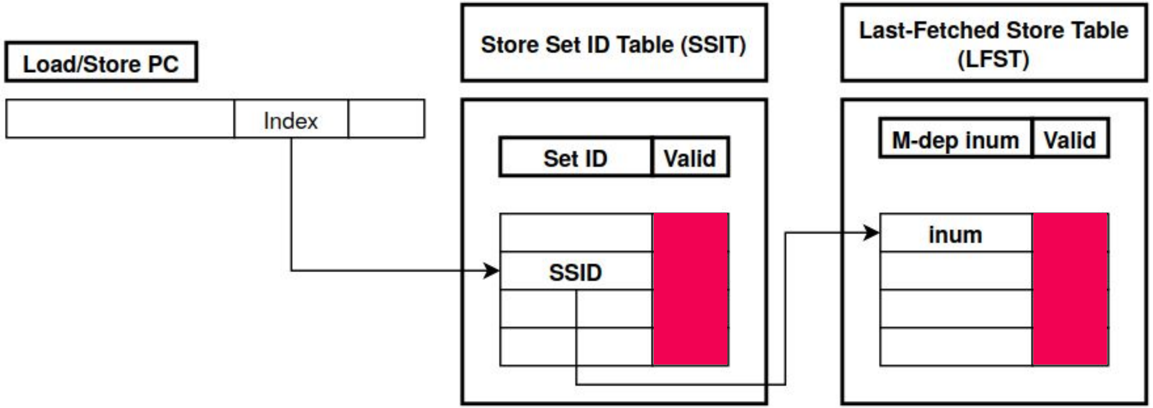
```
STORE   R1  R2   IMM     # mem[ Addr ??? ] <- R1
 (…)
LOAD    R3  R4   IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3   R3      # R5 <- R3+R3
```

R4-producer instr.

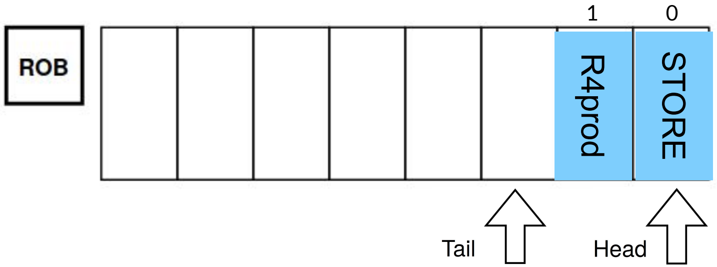


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩

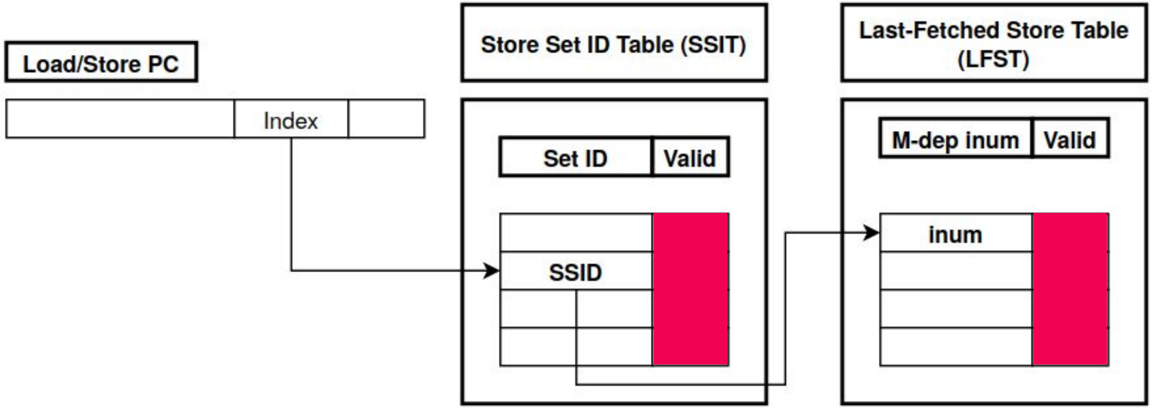


```
STORE   R1  R2  IMM     # mem[ Addr ??? ] <- R1
 (...)
LOAD    R3  R4  IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3   R3     # R5 <- R3+R3
```

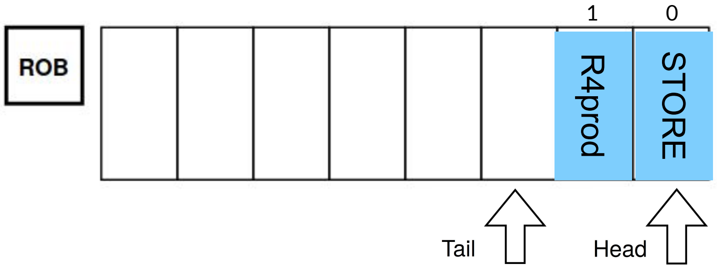


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩

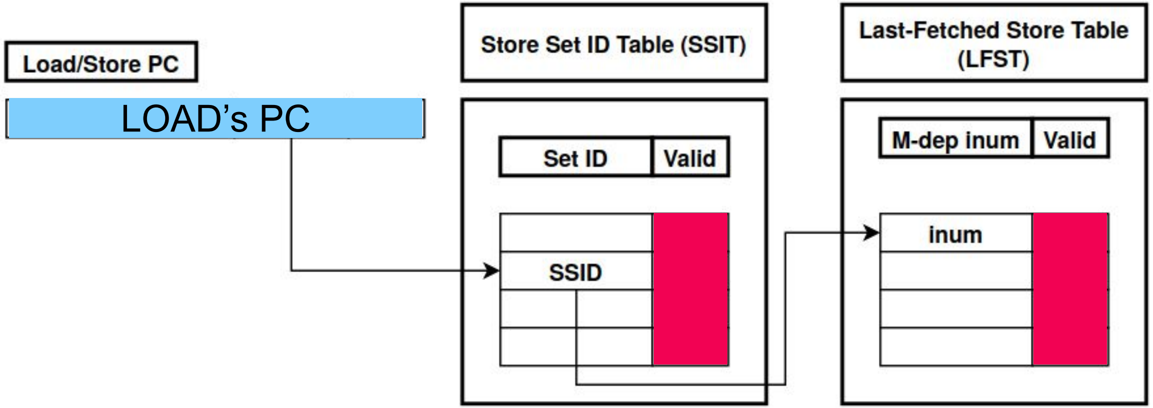


```
STORE   R1  R2   IMM      # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

 ADD    R5  R3   R3       # R5 <- R3+R3
```

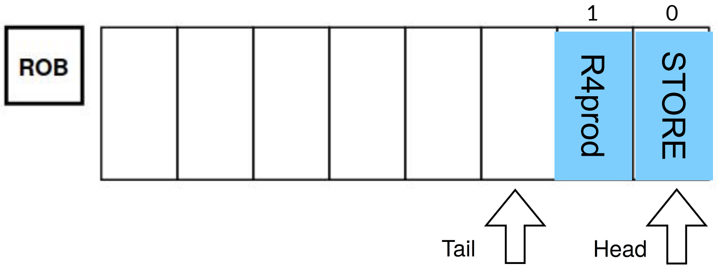


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩

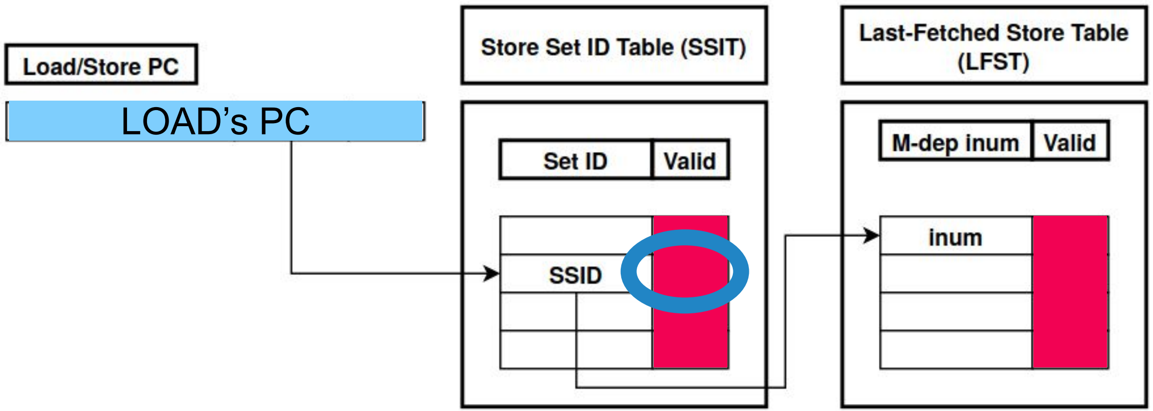


```
STORE   R1  R2   IMM      # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3   R3       # R5 <- R3+R3
```

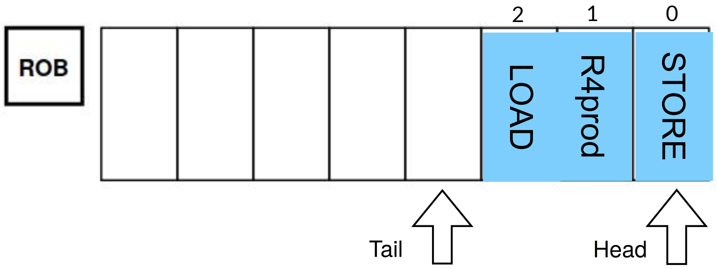


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩

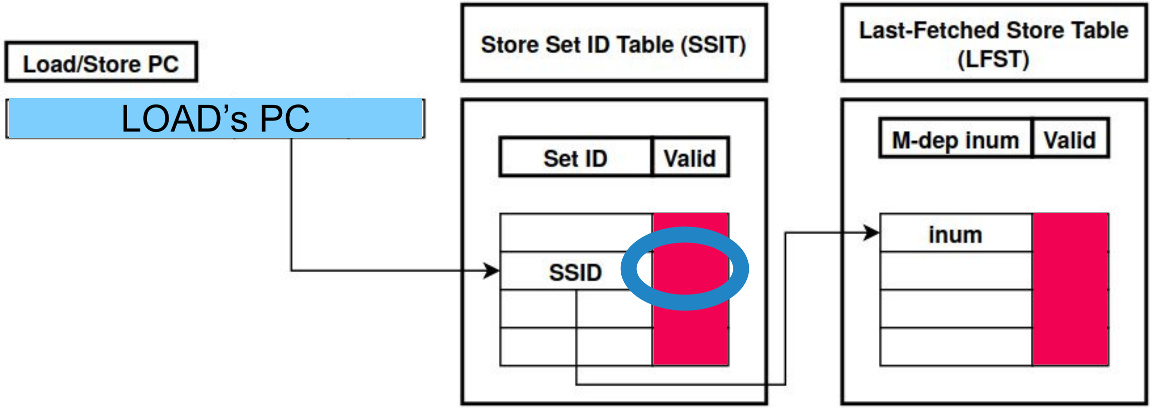


```
STORE   R1  R2  IMM      # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4  IMM      # R3 <- mem[R4+IMM]

 ADD    R5  R3   R3      # R5 <- R3+R3
```

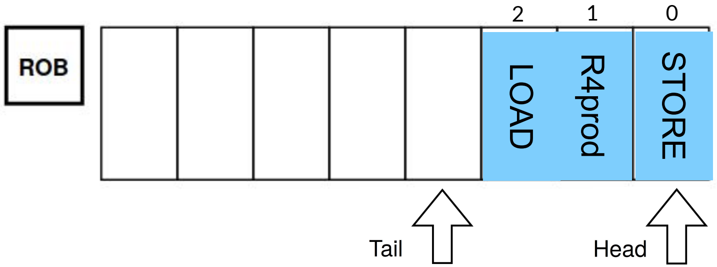


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
LOAD	2	🟥	X	🟩

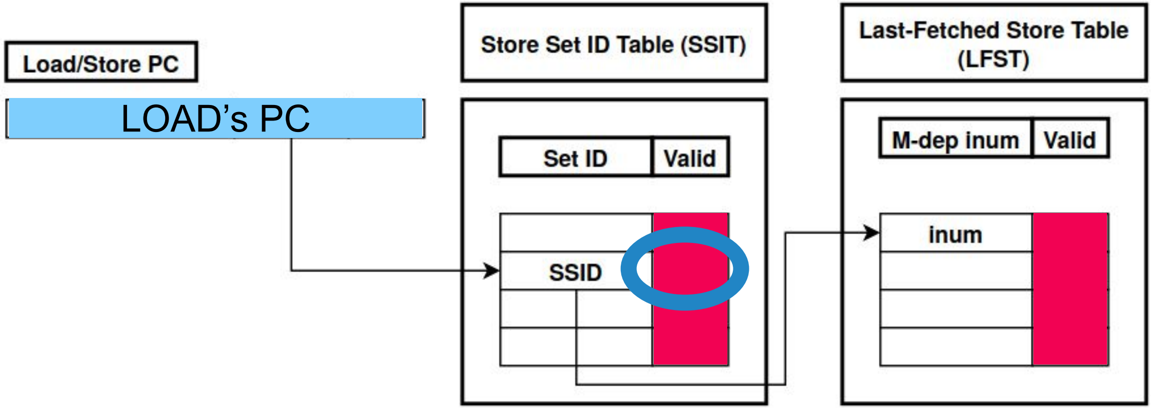


```
STORE   R1  R2   IMM      # mem[ Addr ??? ] <-  R1
 (…)
LOAD    R3  R4   IMM      # R3 <- me[ Addr ??? ]

ADD     R5  R3   R3       # R5 <- R3+R3
```

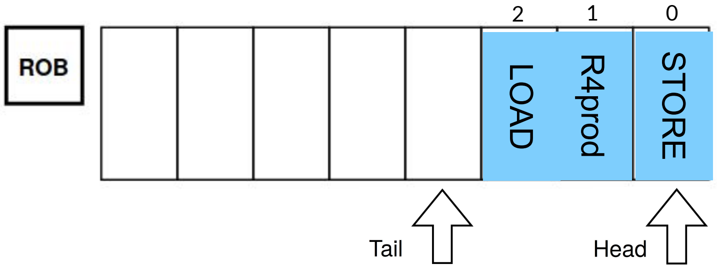


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
LOAD	2	🟥	X	🟩

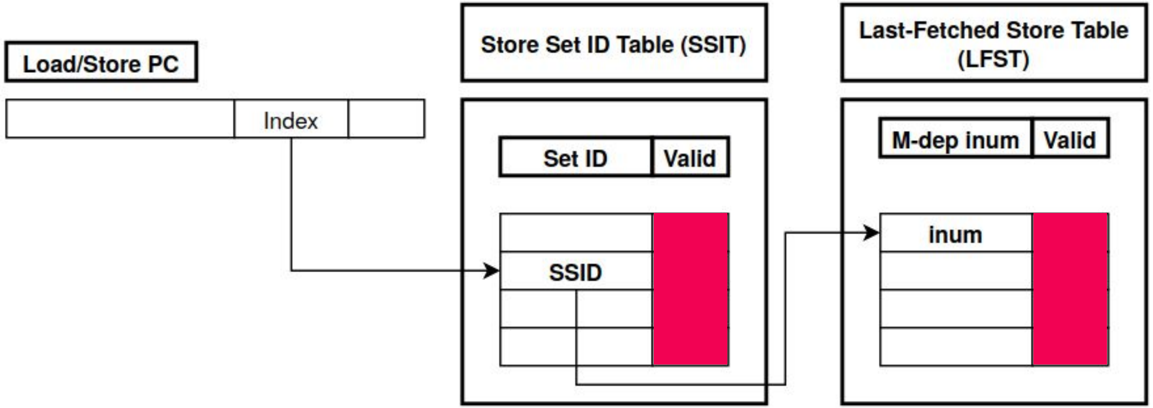


```
   STORE   R1  R2  IMM     # mem[ Addr ??? ] <- R1
   (…)
   LOAD    R3  R4  IMM     # R3 <- mem[ Addr ??? ]

➡  ADD     R5  R3  R3      # R5 <- R3+R3
```



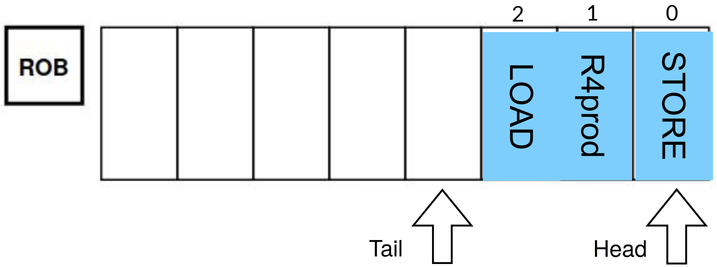
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
LOAD	2	🟥	X	🟩



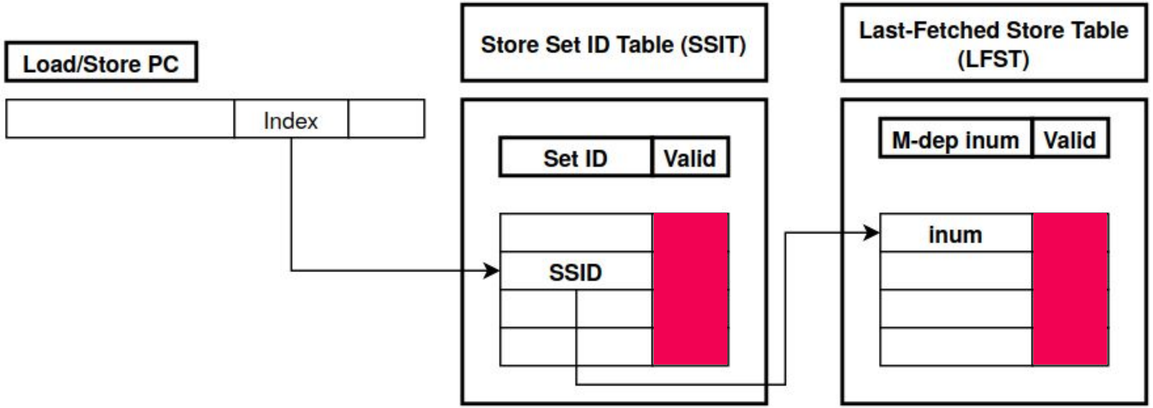
```
STORE   R1  R2  IMM      # mem[ Addr ??? ] <- R1
 (…)
LOAD    R3  R4  IMM      # R3 <- mem[ Addr ??? ]

ADD     R5  R3  R3       # R5 <- R3+R3
```

ADD



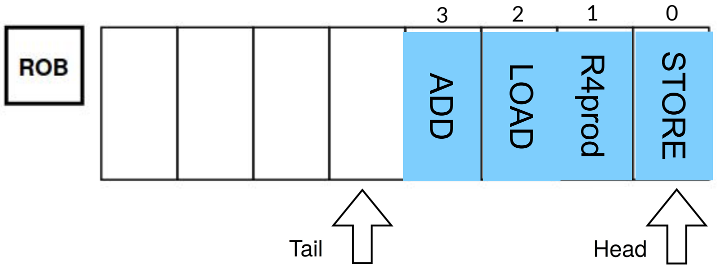
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
LOAD	2	🟥	X	🟩



```
STORE   R1  R2  IMM     # mem[ Addr ??? ] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[ Addr ??? ]

➡ ADD   R5  R3  R3      # R5 <- R3+R3
```

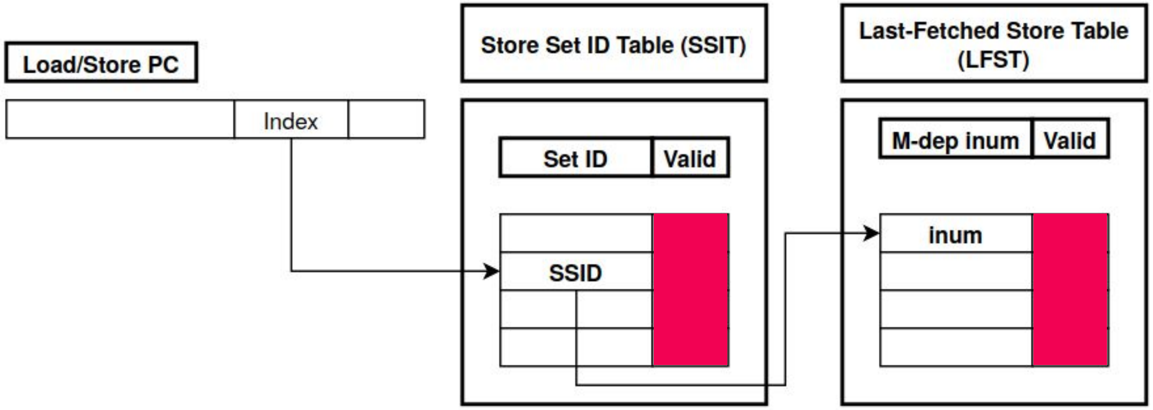
ADD



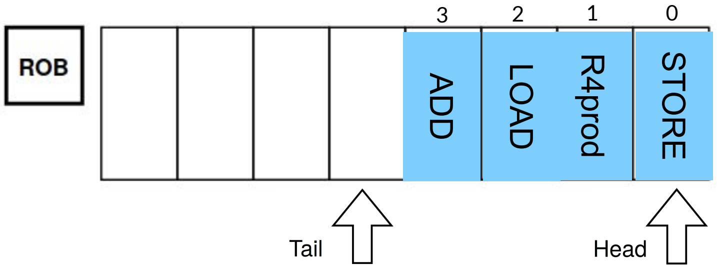
				3	2	1	0
				ADD	LOAD	R4prod	STORE

OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
LOAD	2	🟥	X	🟩
ADD	3	🟥	X	🟩

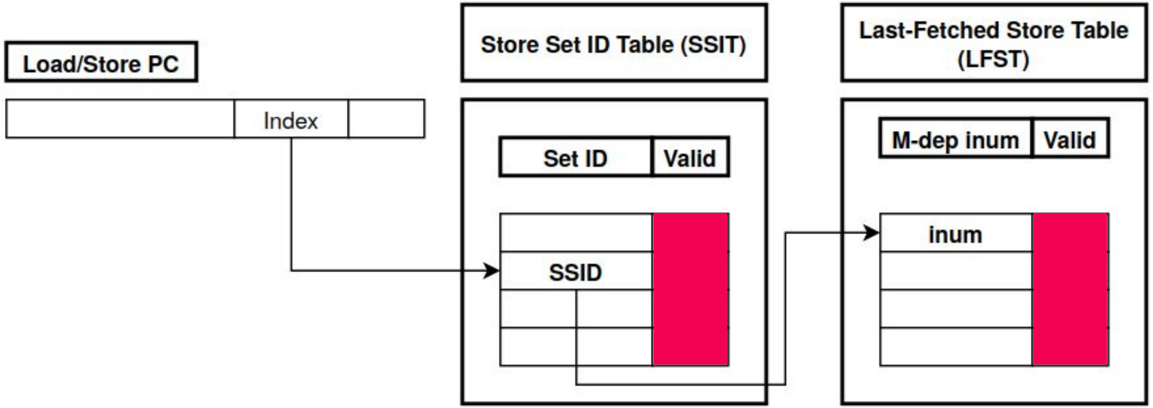


```
STORE   R1  R2  IMM      # mem[ Addr ??? ] <- R1
 (…)
LOAD    R3  R4  IMM      # R3 <- mem[ Addr ??? ]

ADD     R5  R3  R3       # R5 <- R3+R3
```

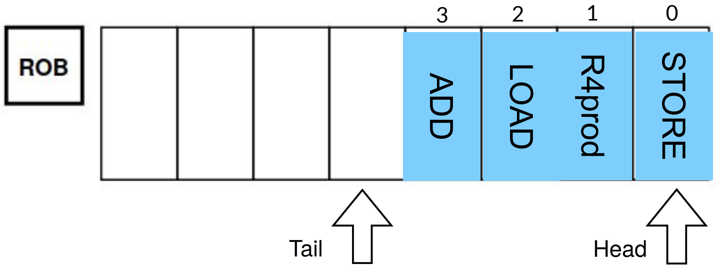


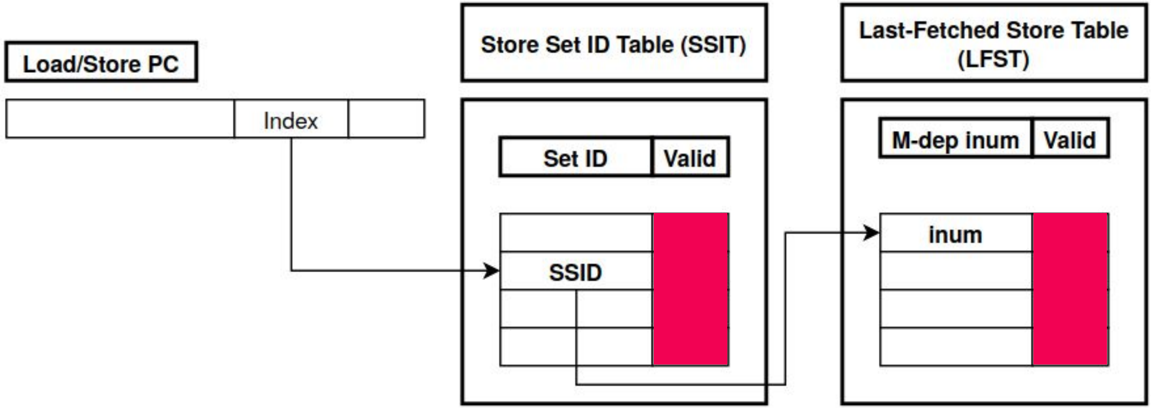
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
LOAD	2	🟥	X	🟩
ADD	3	🟥	X	🟩



```
STORE   R1  R2   IMM      # mem[ Addr ??? ] <- R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[ Addr ??? ]

ADD     R5  R3   R3       # R5 <- R3+R3
```

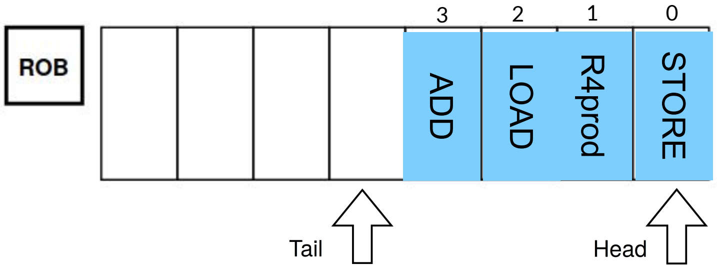


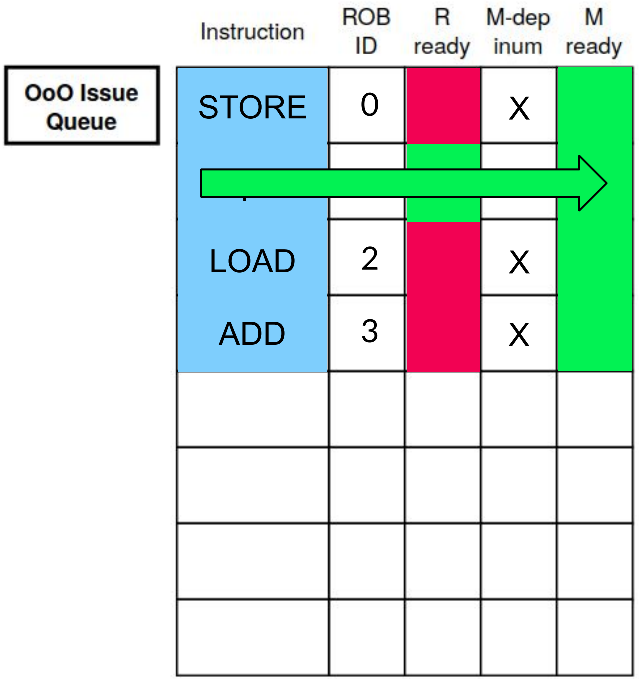


OoO Issue Queue

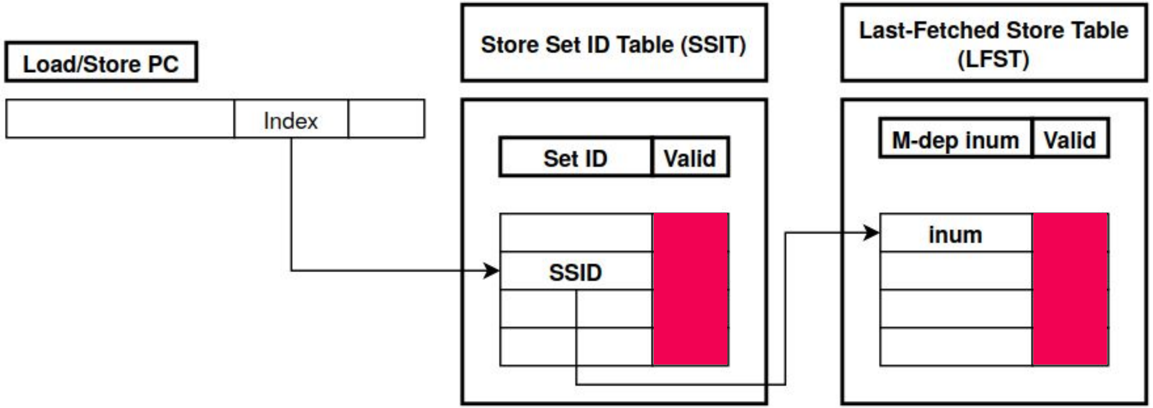
Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X !!	🟩
R4prod	1	🟩	X	🟩
LOAD	2	🟥	X	🟩
ADD	3	🟥	X	🟩

```
STORE   R1  R2  IMM      # mem[ Addr ??? ] <- R1
 (...)
LOAD    R3  R4  IMM      # R3 <- mem[ Addr ??? ]

ADD     R5  R3   R3      # R5 <- R3+R3
```

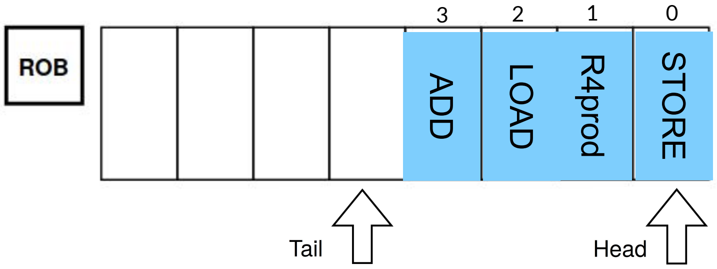


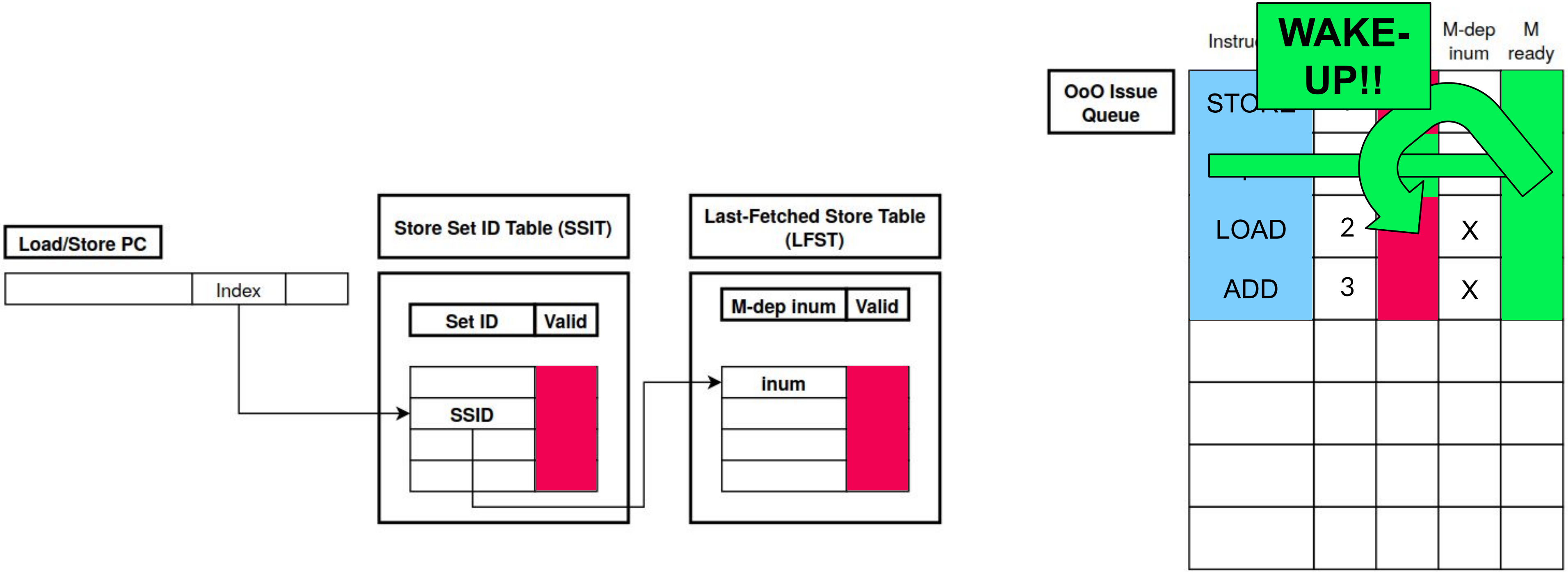


Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
		🟩		🟩
LOAD	2	🟥	X	🟩
ADD	3	🟥	X	🟩

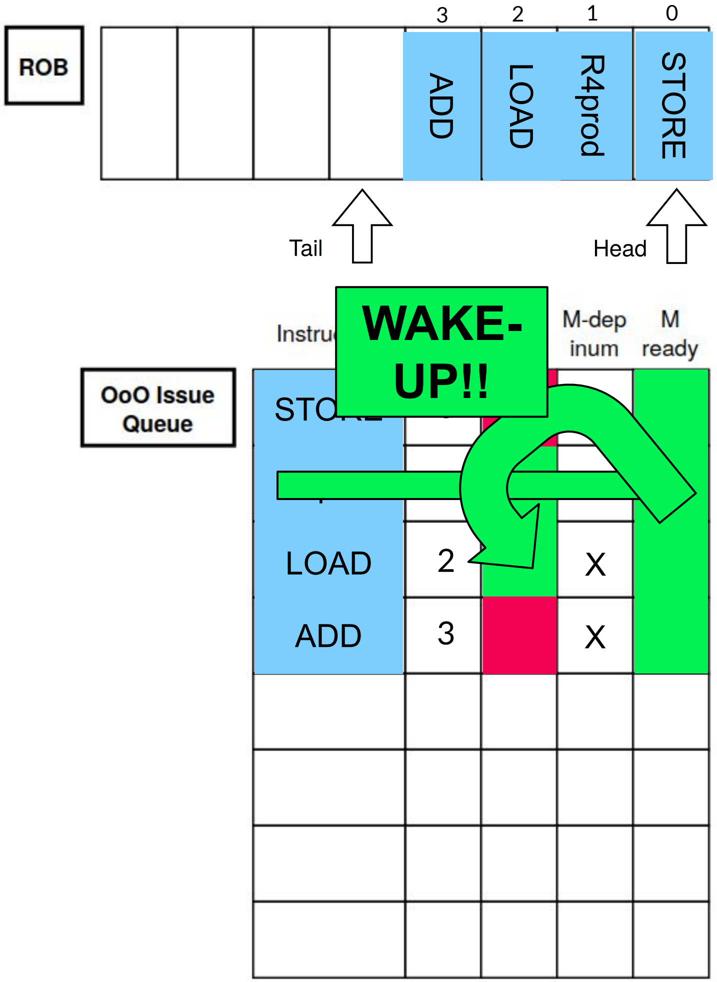


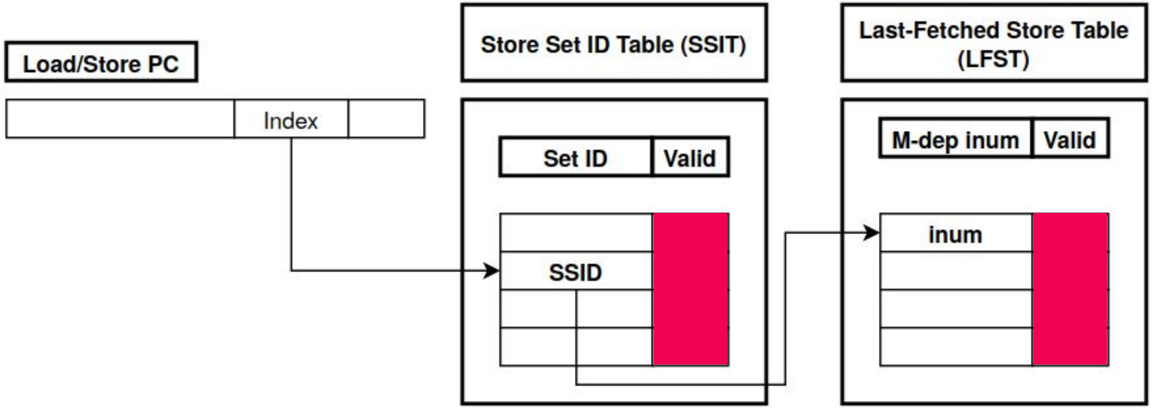
```
STORE    R1  R2   IMM       # mem[ Addr ??? ] <-  R1
 (…)
LOAD     R3  R4   IMM       # R3 <-  mem[ Addr ??? ]

ADD      R5  R3   R3        # R5 <-  R3+R3
```



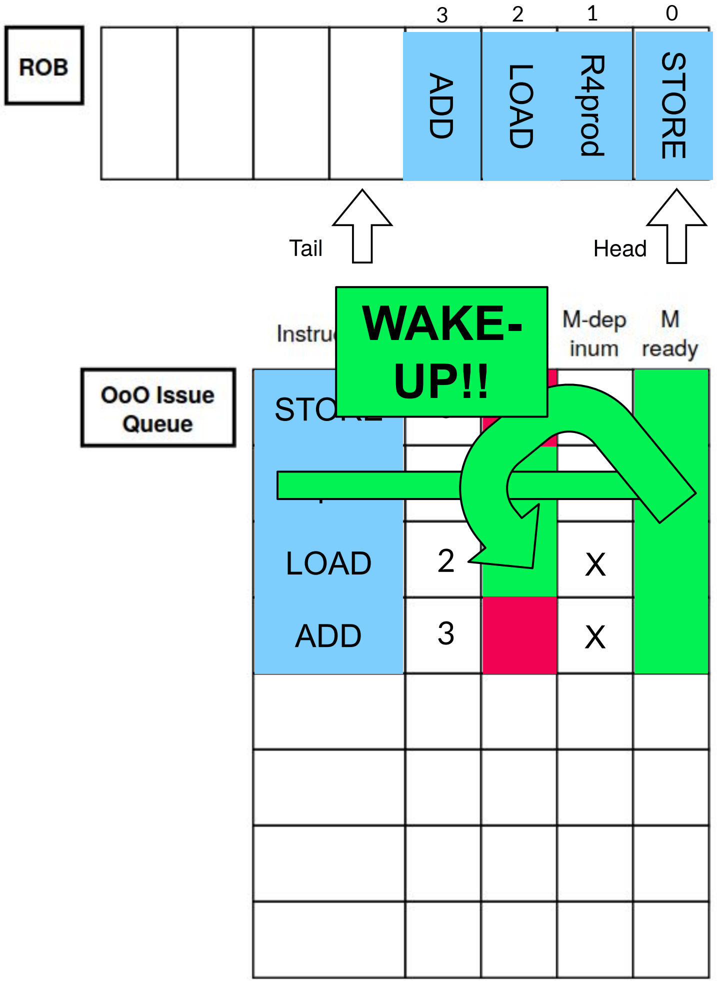


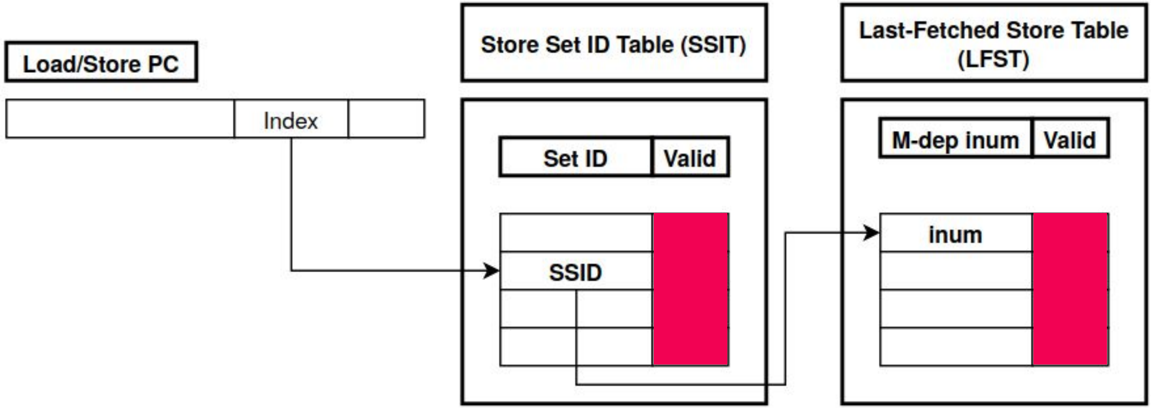
```
STORE    R1  R2   IMM      # mem[ Addr ??? ] <-  R1
 (…)
LOAD     R3  R4   IMM      # R3 <-  mem[ Addr ??? ]

ADD      R5  R3   R3       # R5 <-  R3+R3
```





```
STORE    R1  R2   IMM       # mem[Addr ???] <-  R1 !!
 (...)
LOAD     R3  R4   IMM       # R3 <- mem[Addr A]

ADD      R5  R3   R3        # R5 <- R3+R3
```





```
STORE   R1  R2   IMM      # mem[Addr ???] <- R1 !!
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[Addr A]

 ADD    R5  R3    R3      # R5 <- R3+R3
```

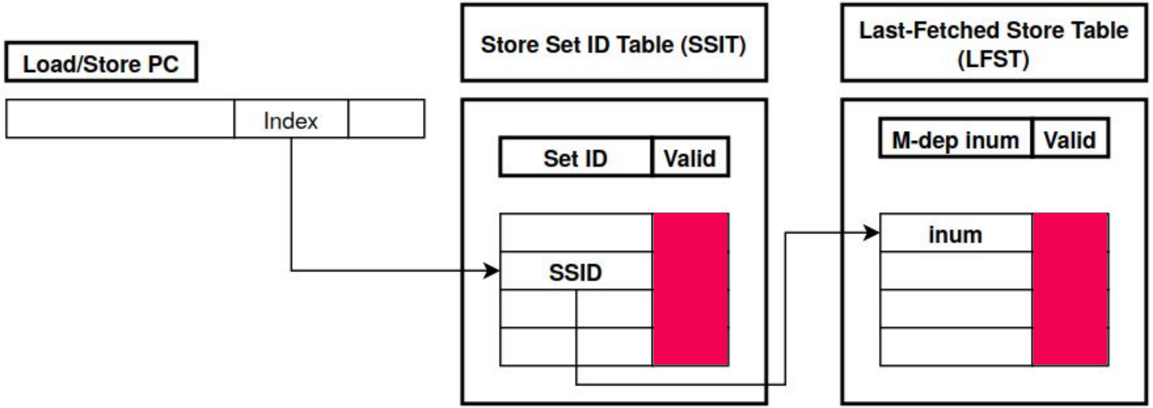
ROB

				3	2	1	0
				ADD	LOAD	R4prod	STORE

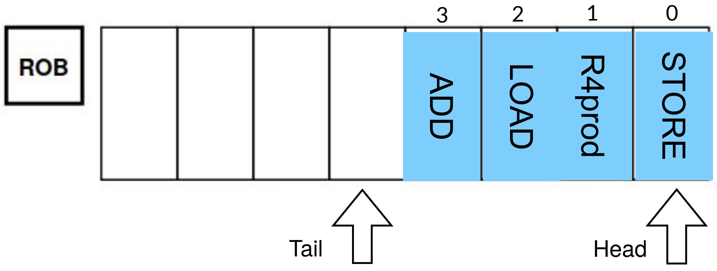
Tail (slot 3), Head (slot 0)

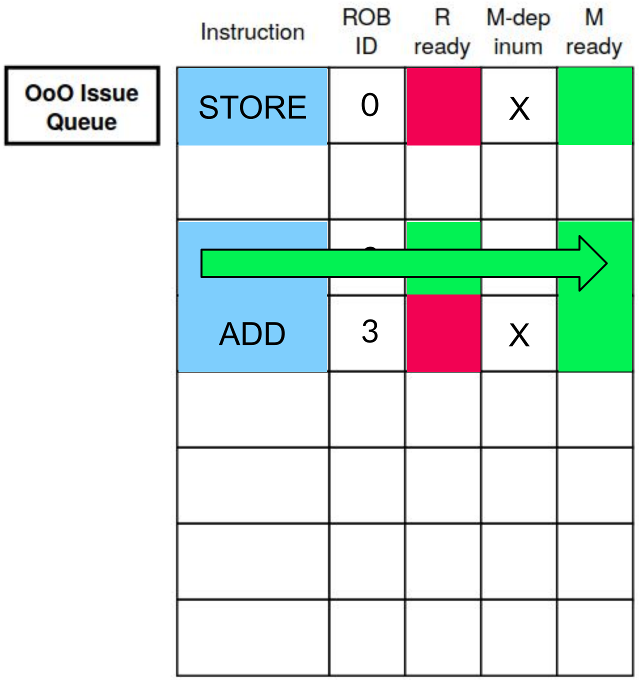
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
LOAD	2	🟩	X	🟩
ADD	3	🟥	X	

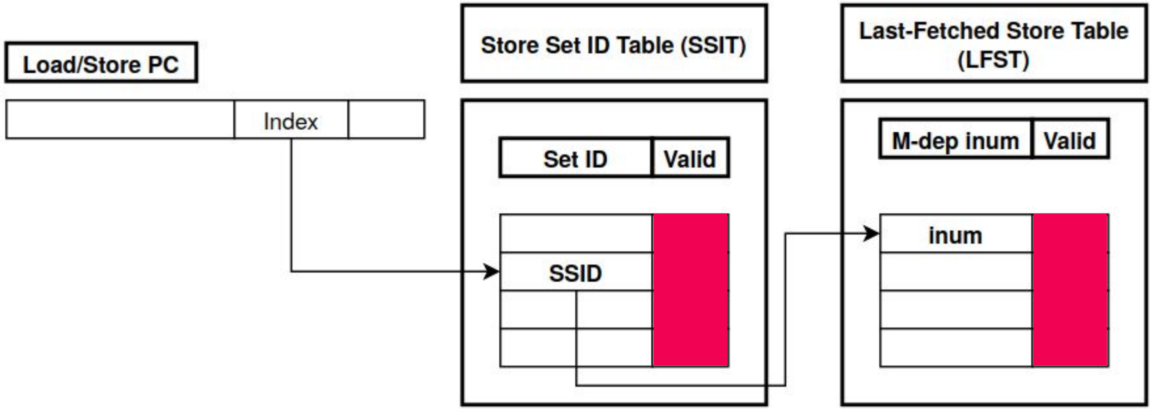


```
STORE   R1  R2  IMM      # mem[Addr ???] <- R1 !!
(…)
LOAD    R3  R4  IMM      # R3 <- mem[Addr A]

ADD     R5  R3  R3       # R5 <- R3+R3
```

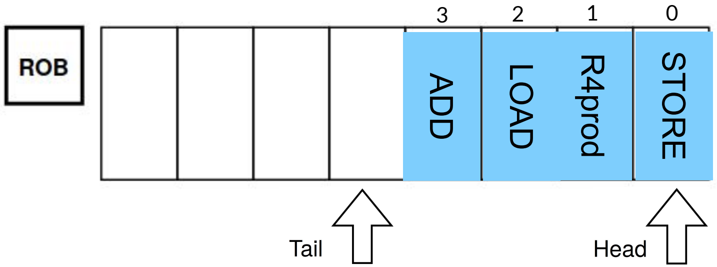


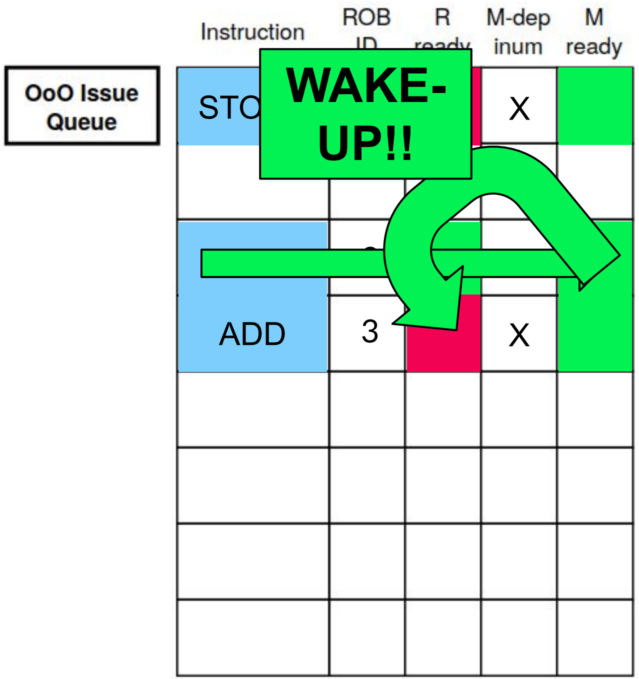


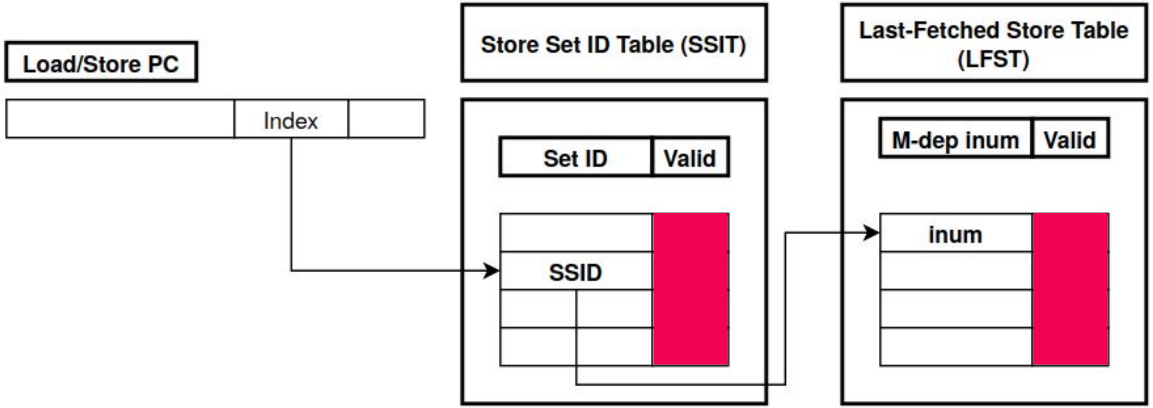
Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
		🟩		🟩
ADD	3	🟥	X	🟩



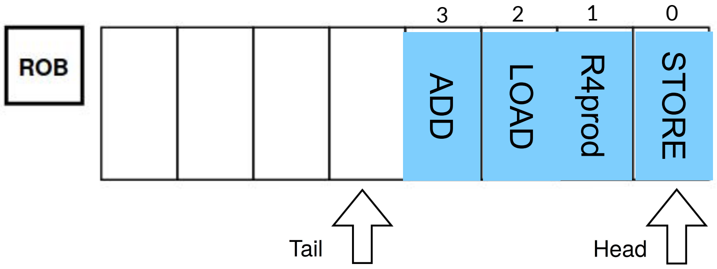
```
STORE   R1  R2  IMM     # mem[Addr ???] <- R1!!
(…)
LOAD    R3  R4  IMM     # R3 <- mem[Addr A]

ADD     R5  R3  R3      # R5 <- R3+R3
```

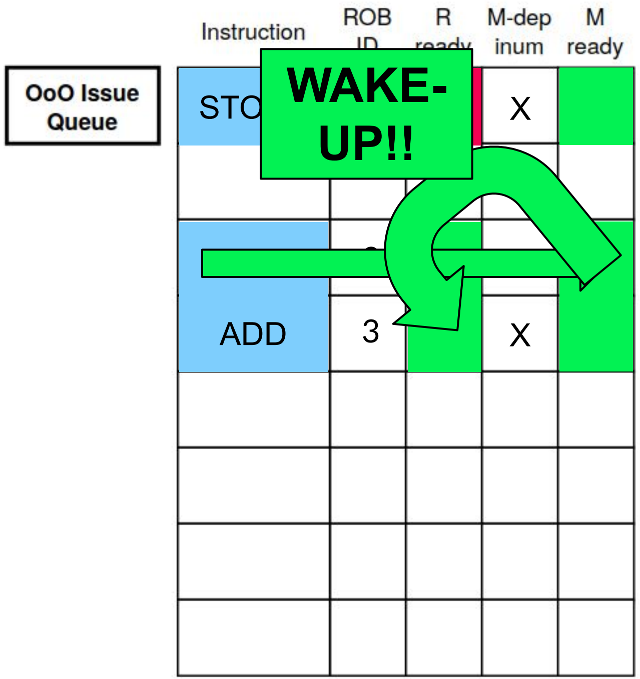




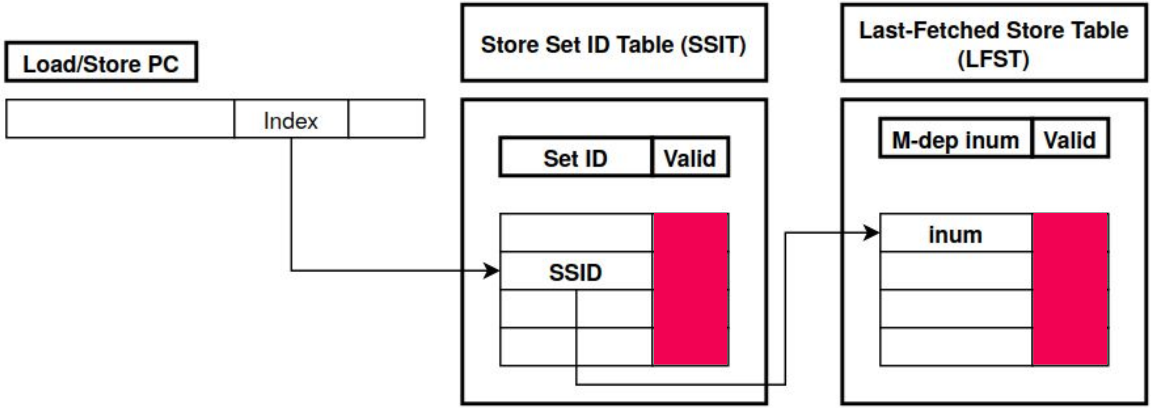


```
STORE   R1  R2  IMM     # mem[ Addr ??? ] <- R1 !!
(...)
LOAD    R3  R4  IMM     # R3 <- mem[ Addr A ]

ADD     R5  R3  R3      # R5 <- R3+R3
```

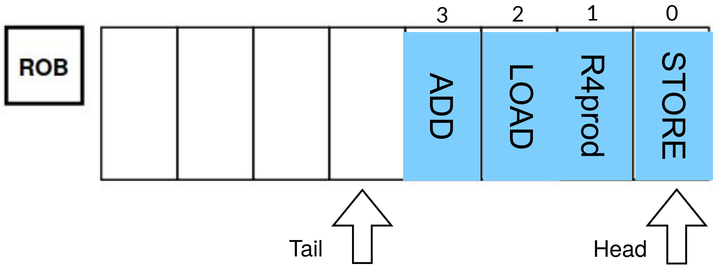




Instruction	ROB ID	R ready	M-dep inum	M ready
STO		🟥	X	🟩
ADD	3	🟩	X	🟩

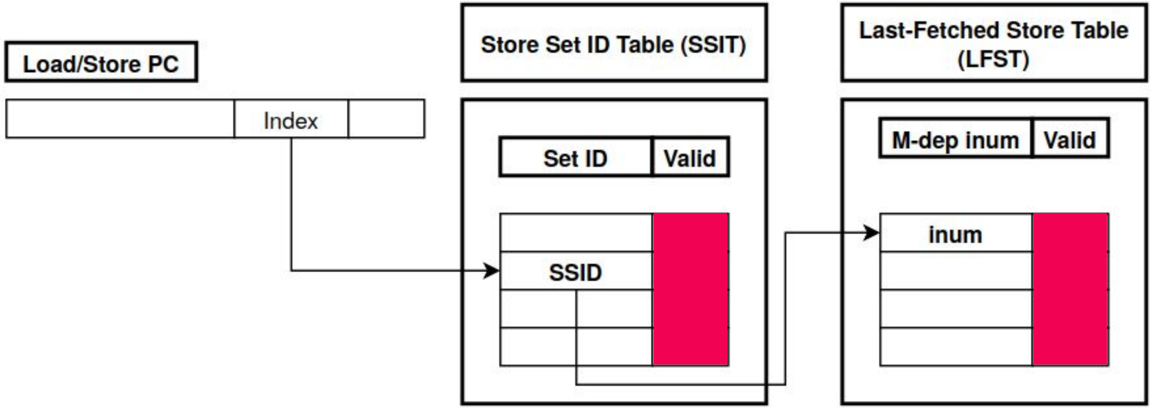


```
STORE   R1  R2  IMM      # mem[Addr ???] <- R1 !!
(…)
LOAD    R3  R4  IMM      # R3 <- mem[Addr A]

ADD     R5  R3  R3       # R5 <- R3+R3
```

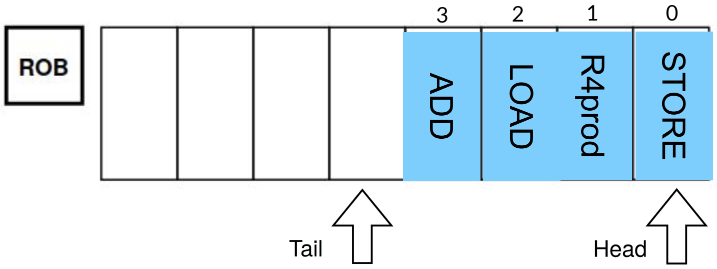


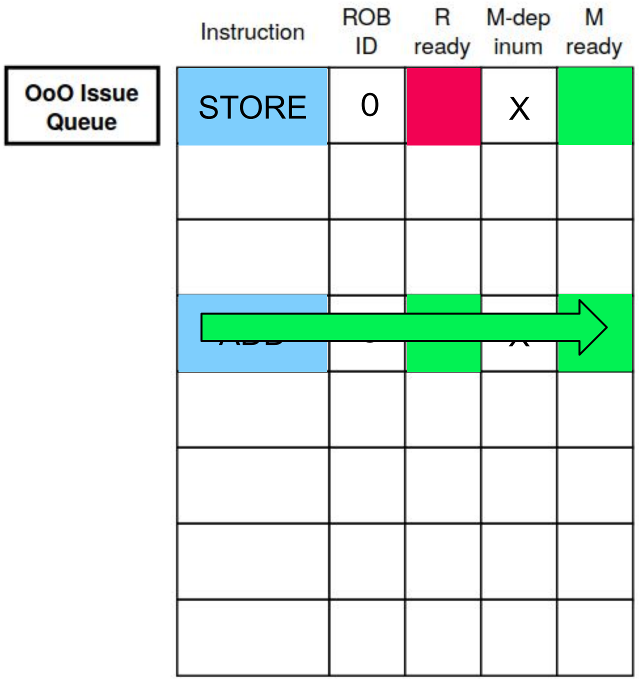
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
ADD	3	🟩	X	🟩

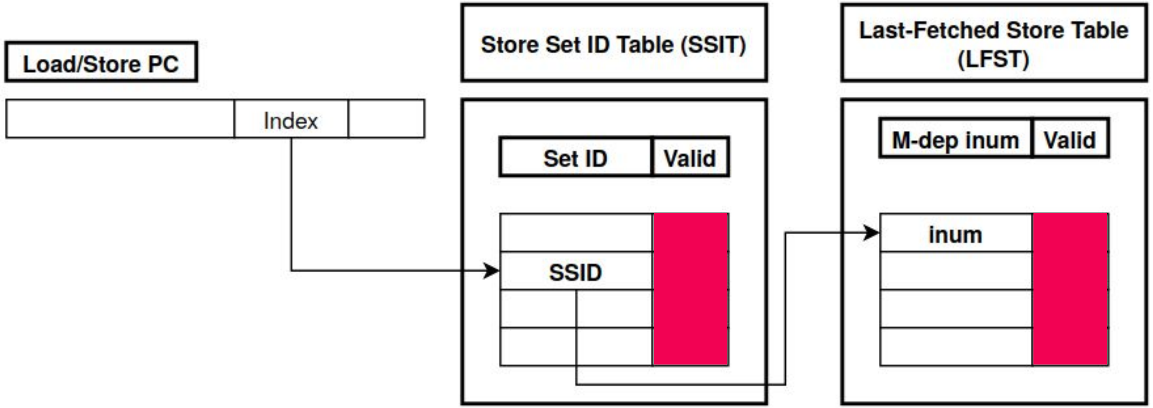


```
STORE   R1  R2   IMM      # mem[Addr ???] <- R1 !!
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[Addr A]

ADD     R5  R3   R3       # R5 <- R3+R3
```

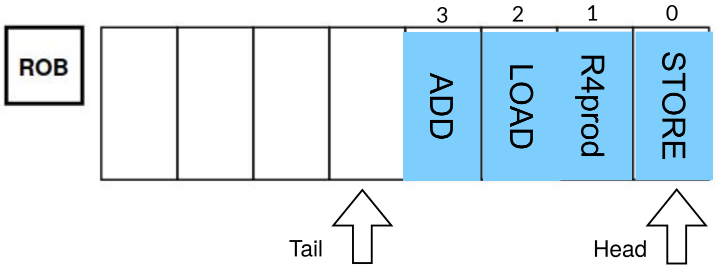




Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
ADD		🟩	X	🟩

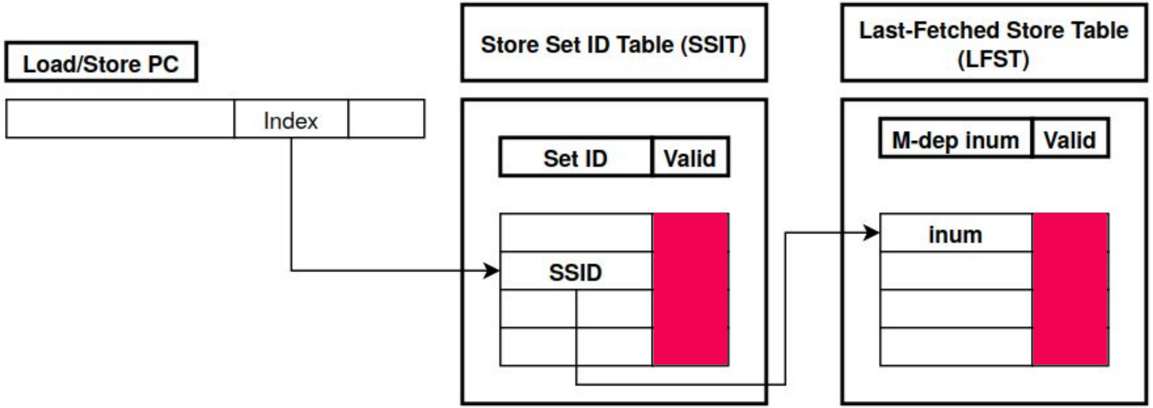


```
STORE   R1  R2  IMM     # mem[Addr ???] <- R1 !!
 (...)
LOAD    R3  R4  IMM     # R3 <- mem[Addr A]

ADD     R5  R3  R3      # R5 <- R3+R3
```

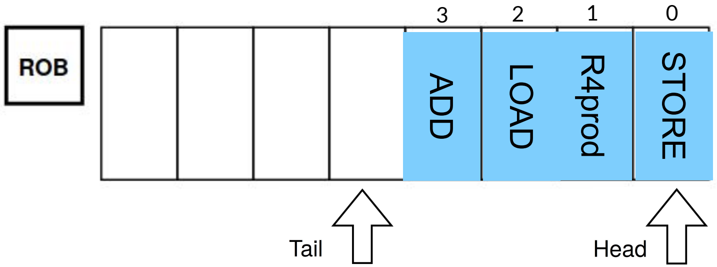


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩



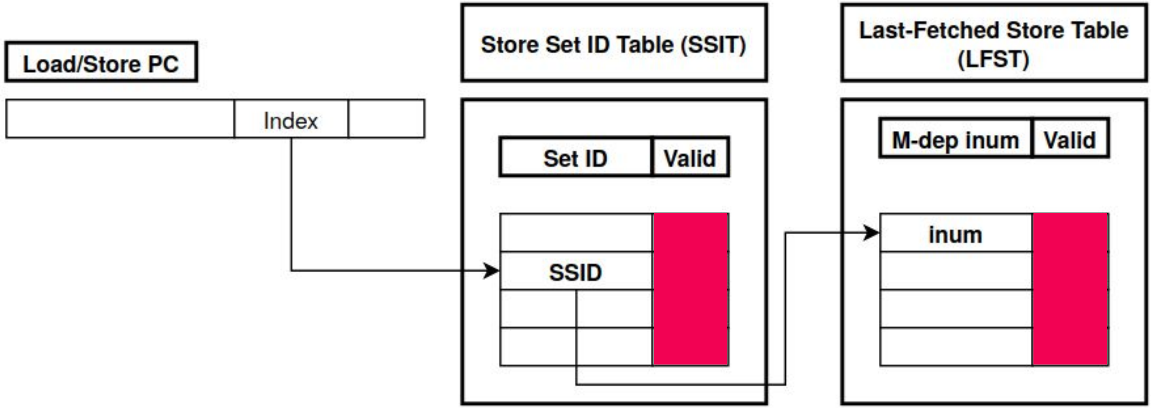
```
STORE   R1  R2  IMM      # mem[Addr ???] <- R1 !!
 (...)
 LOAD   R3  R4  IMM      # R3 <- mem[Addr A]

 ADD    R5  R3   R3      # R5 <- R3+R3
```



OoO Issue Queue

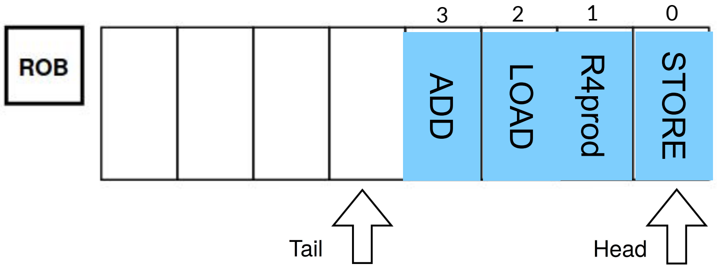
Instruction	ROB ID	R	M-dep inum	M ready
STORE		🟥		🟩

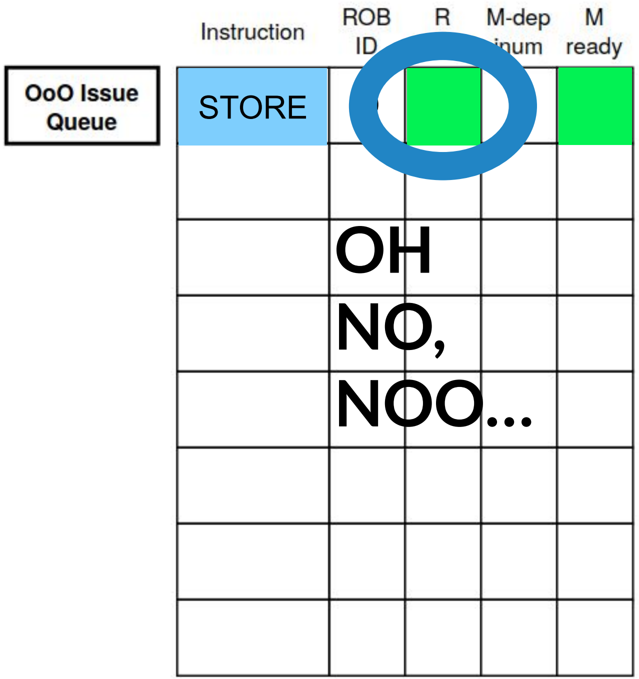
OH NO...



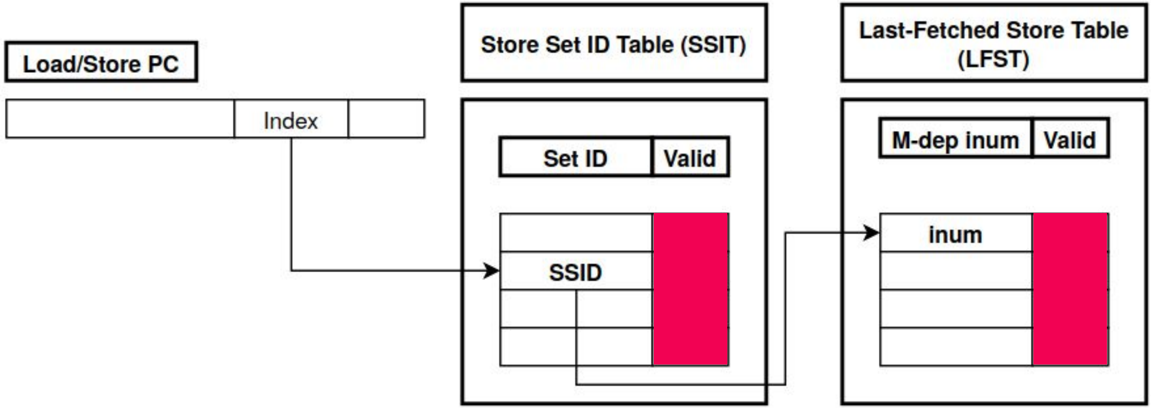
```
STORE    R1  R2   IMM      # mem[Addr ???] <-  R1
 (...)
LOAD     R3  R4   IMM      # R3 <-  mem[Addr A]

ADD      R5  R3   R3       # R5 <-  R3+R3
```

!!

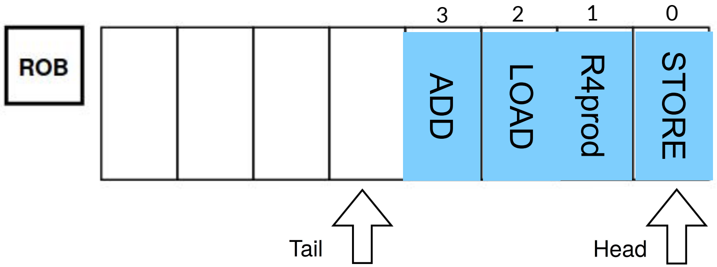




OH NO, NOO...



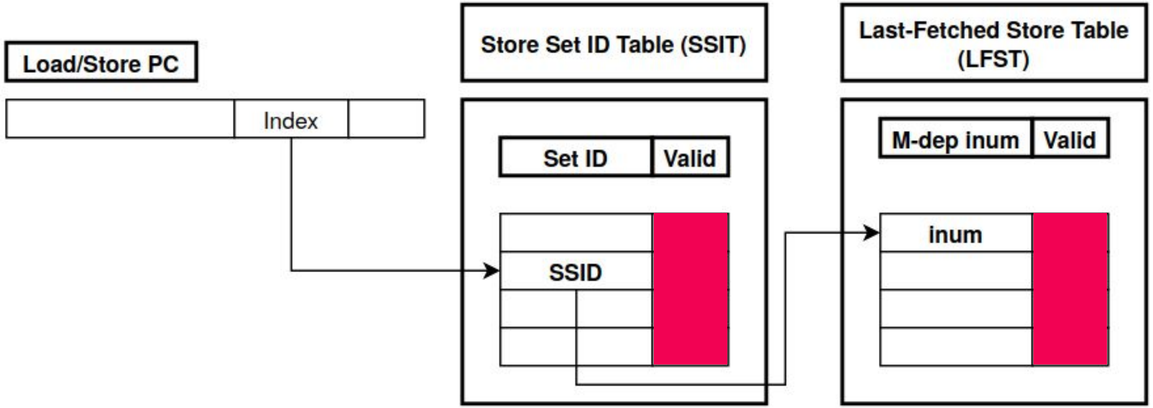
```
STORE    R1  R2   IMM      # mem[Addr ???] <- R1!!
 (…)
LOAD     R3  R4   IMM      # R3 <- mem[Addr A]

 ADD     R5  R3   R3       # R5 <- R3+R3
```



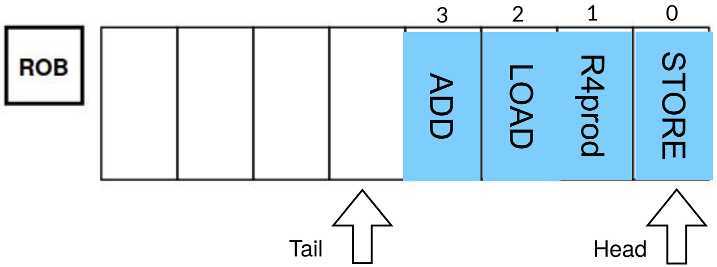
OoO Issue Queue

Instruction	ROB ID	R	M-dep inum	M ready
STORE		🟩		🟩

OH
NO,
NOO,
NOOO…



```
STORE   R1  R2   IMM      # mem[Addr A] <- R1!!
 (...)
LOAD    R3  R4   IMM      # R3 <- mem[Addr A]

ADD     R5  R3   R3       # R5 <- R3+R3
```



OoO Issue Queue

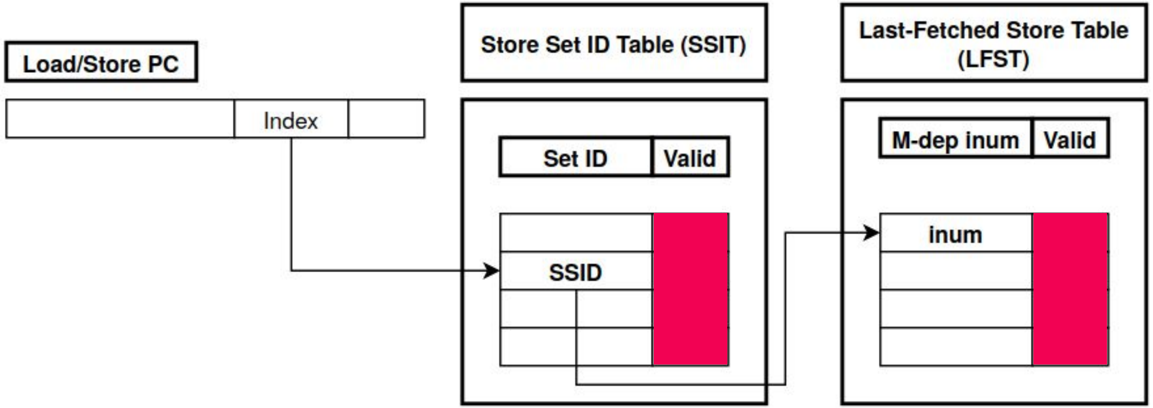
Instruction	ROB ID	R	M-dep inum	M ready
STORE		🟩		🟩

OH

NO,

NOO,

NOOO...



```
STORE   R1  R2   IMM      # mem[Addr A] <- R1 !!
(…)
LOAD    R3  R4   IMM      # R3 <- mem[Addr A]

ADD     R5  R3   R3       # R5 <- R3+R3
```

ROB					3	2	1	0
					ADD	LOAD	R4prod	STORE

Tail

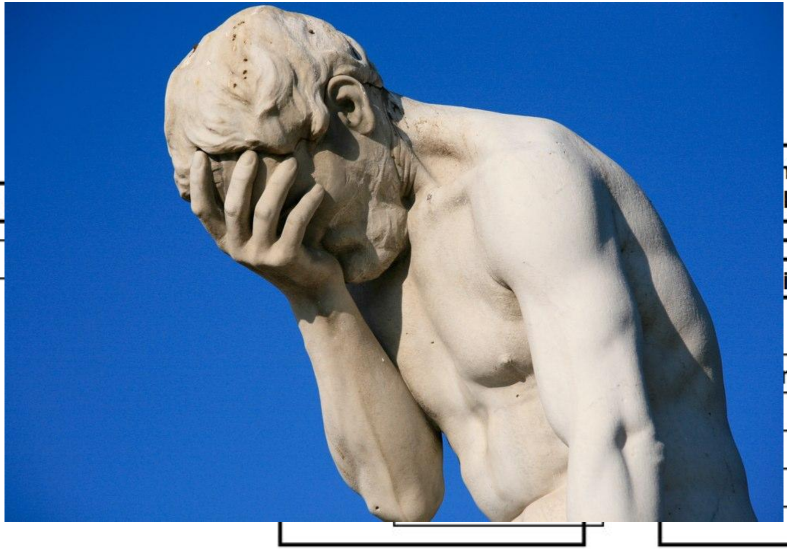
Head

OoO Issue Queue

Instruction	ROB ID	R	M-dep inum	M ready
STORE		🟩		🟩

OH
NO,
NOO,
NOOO…

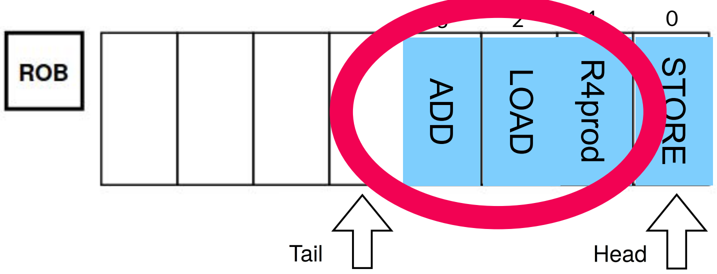
Load/Store PC



ned Store Table (LFST)

inum	Valid
m	🟥
	🟥
	🟥
	🟥

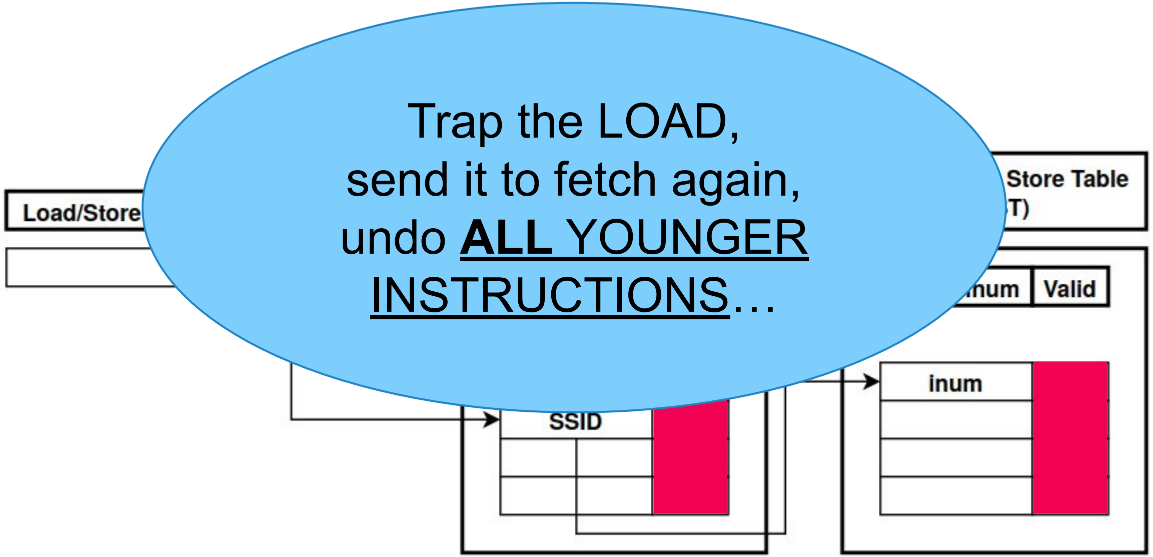
```
STORE   R1  R2  IMM     # mem[Addr A] <- R1 !!
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[Addr A]

ADD     R5  R3   R3     # R5 <- R3+R3
```



OoO Issue Queue

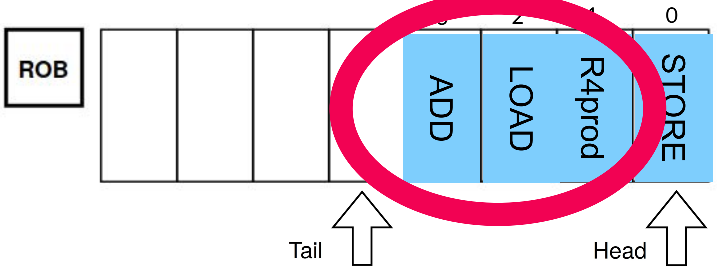
Instruction	ROB ID	R	M-dep inum	M ready
STORE		🟩		🟩

OH NO, NOO, NOOO…



Trap the LOAD,
send it to fetch again,
undo **ALL YOUNGER INSTRUCTIONS**…

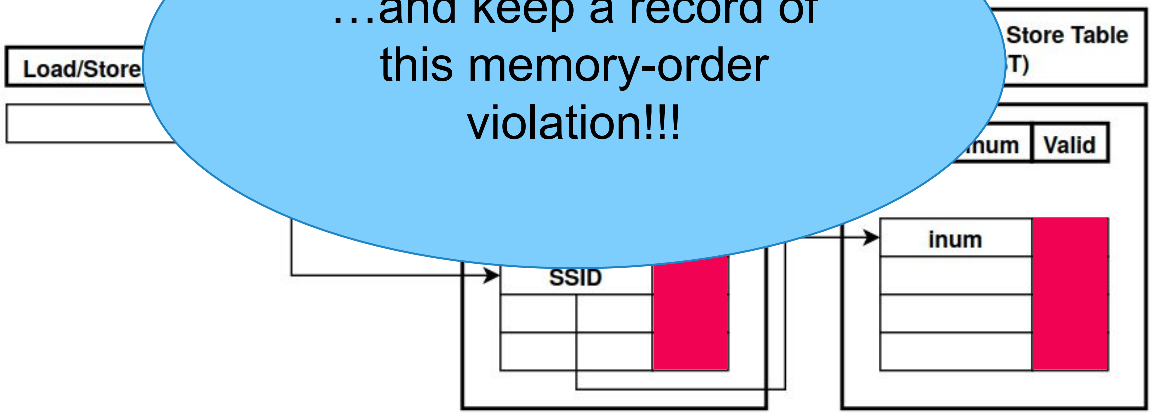
```
STORE   R1  R2  IMM     # mem[Addr A] <- R1 !!
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[Addr A]

ADD     R5  R3  R3      # R5 <- R3+R3
```



OoO Issue Queue

Instruction	ROB ID	R	M-dep inum	M ready
STORE		🟩		🟩

OH NO, NOO, NOOO…



…and keep a record of this memory-order violation!!!

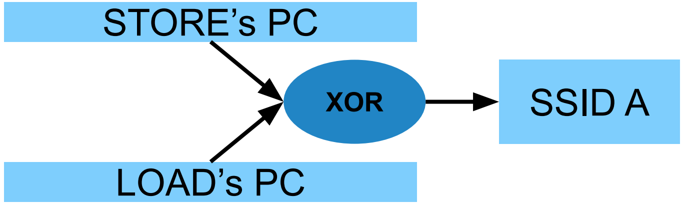
```
STORE   R1  R2  IMM      # mem[ Addr A ] <- R1
 (...)
LOAD    R3  R4  IMM      # R3 <- mem[ Addr A ]

ADD     R5  R3   R3      # R5 <- R3+R3
```

ROB

				3	2	1	0
				ADD	LOAD	R4prod	STORE

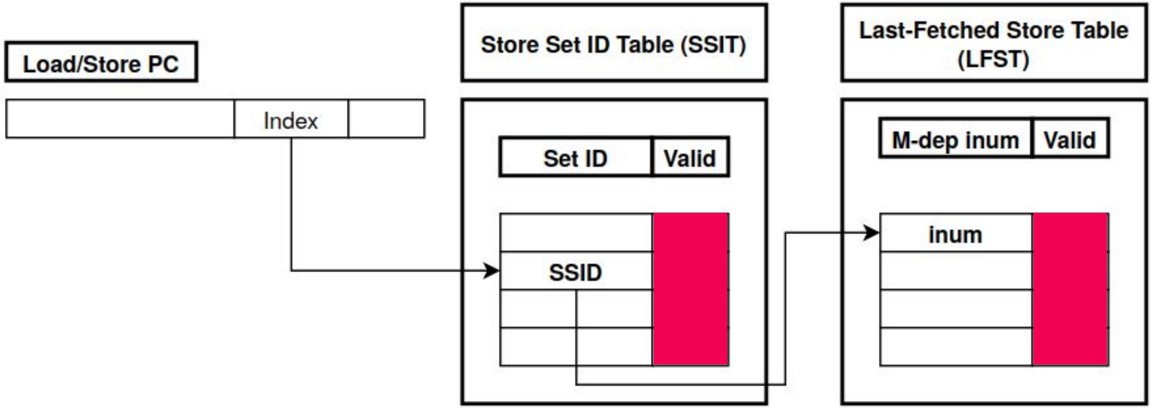
Tail

Head

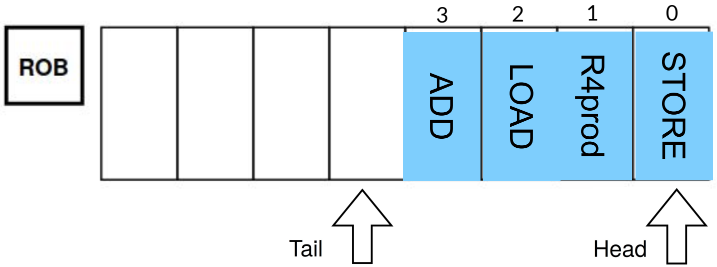


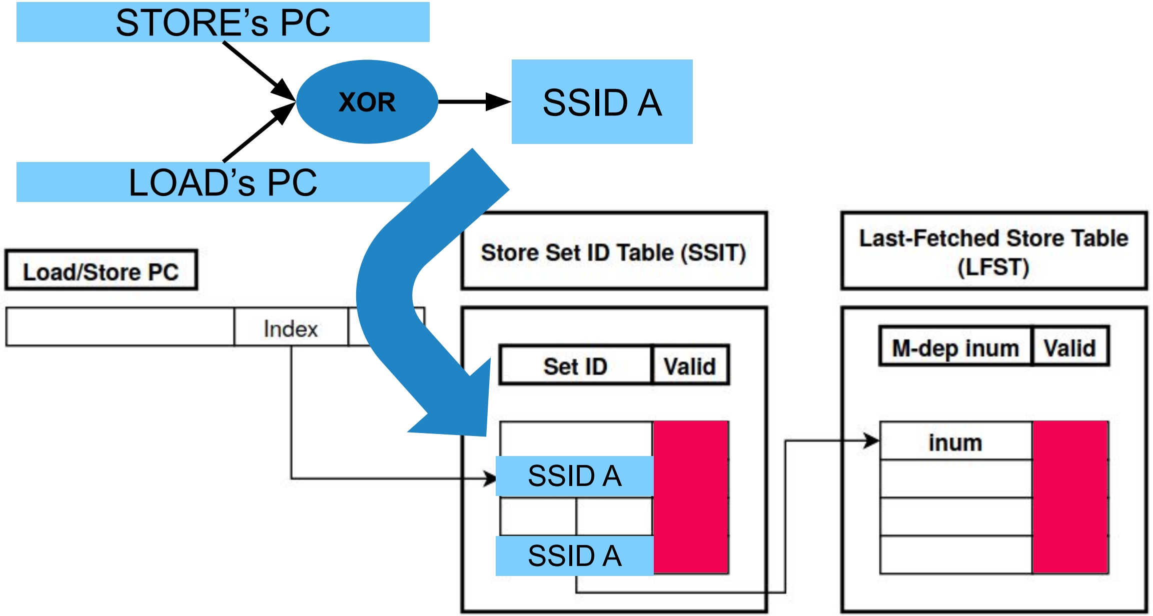
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟩	X	🟩



```
STORE   R1  R2   IMM      # mem[ Addr A ] <- R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[ Addr A ]

ADD     R5  R3   R3       # R5 <- R3+R3
```





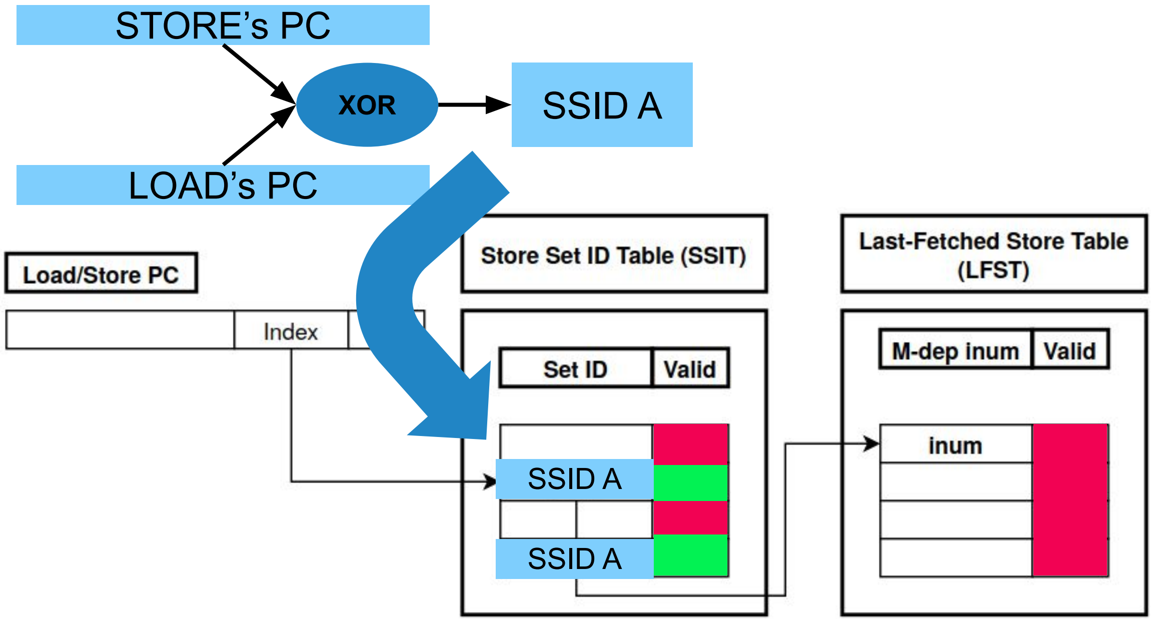
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟩	X	🟩

```
STORE   R1  R2   IMM      # mem[ Addr A ] <-  R1
 (…)
LOAD    R3  R4   IMM      # R3 <-  mem[ Addr A ]

ADD     R5  R3   R3       # R5 <-  R3+R3
```

ROB

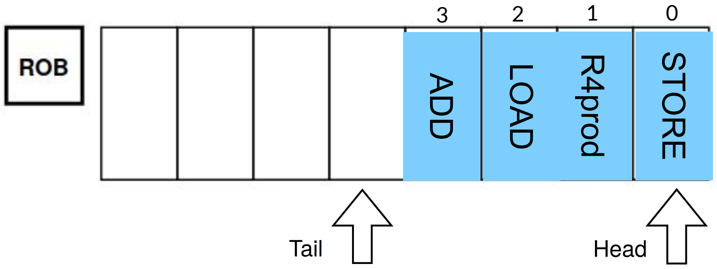
				3	2	1	0
				ADD	LOAD	R4prod	STORE
			Tail				Head

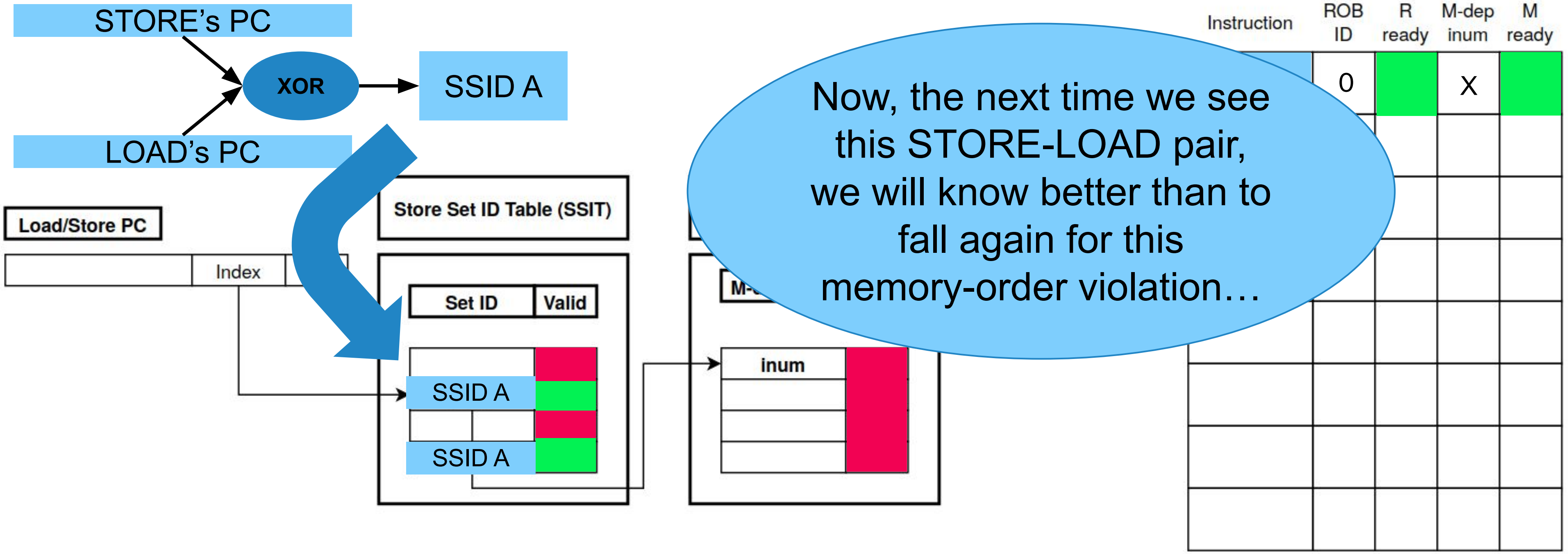


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟩	X	🟩

```
STORE   R1  R2   IMM      # mem[ Addr A ] <- R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[ Addr A ]

ADD     R5  R3   R3       # R5 <- R3+R3
```

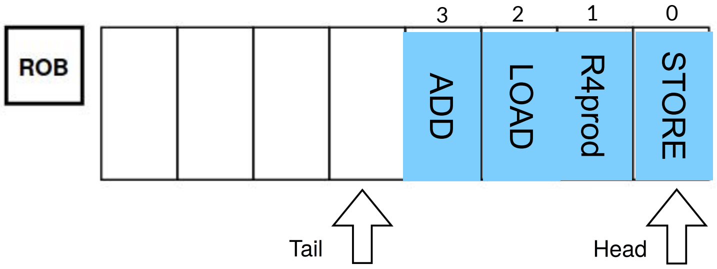


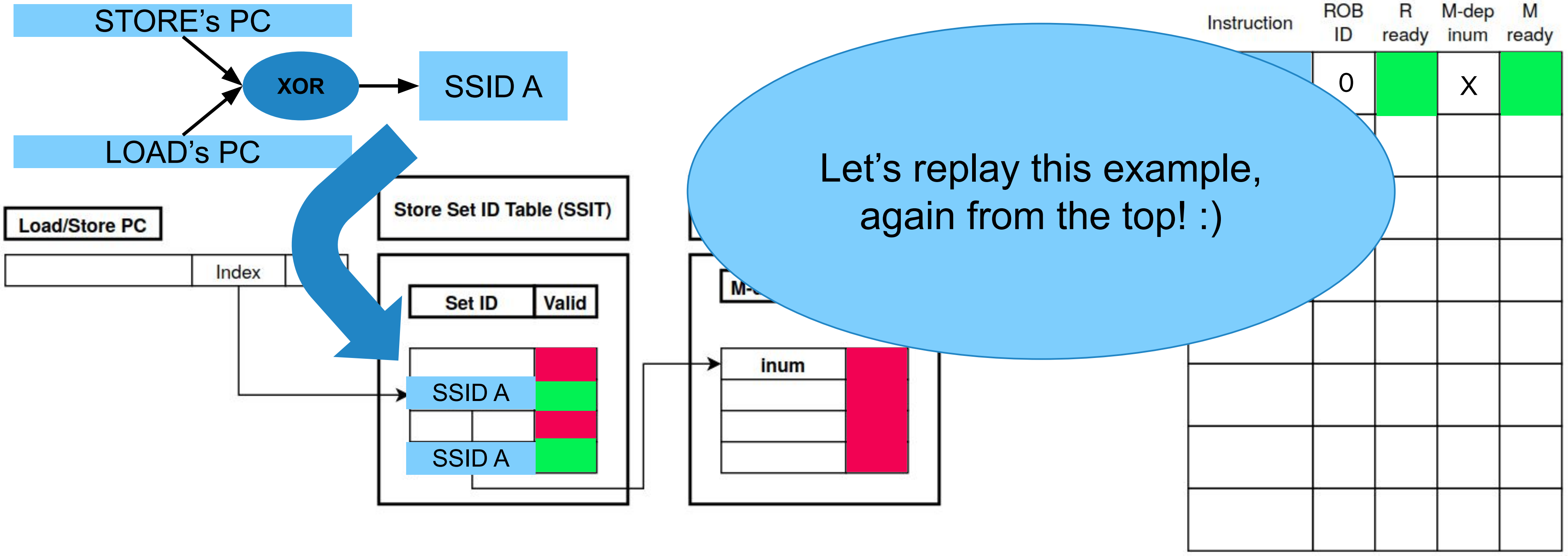


Instruction	ROB ID	R ready	M-dep inum	M ready
	0	🟩	X	🟩

Now, the next time we see this STORE-LOAD pair, we will know better than to fall again for this memory-order violation…

```
STORE    R1  R2   IMM      # mem[ Addr A ] <-  R1
 (…)
LOAD     R3  R4   IMM      # R3 <-  mem[ Addr A ]

ADD      R5  R3    R3      # R5 <-  R3+R3
```

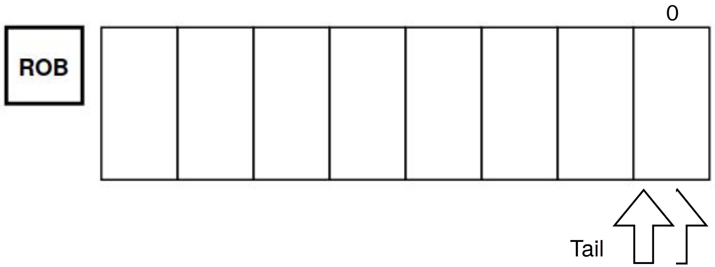




Instruction	ROB ID	R ready	M-dep inum	M ready
	0	🟩	X	🟩

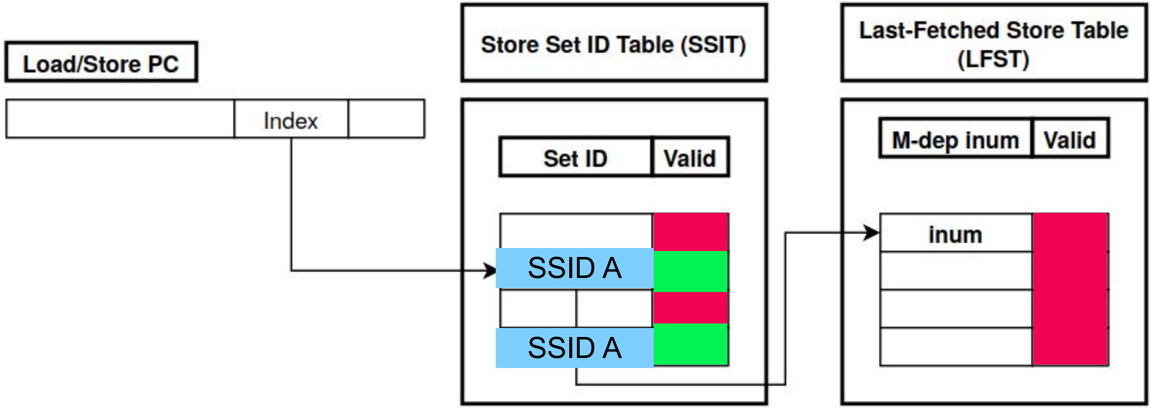
Let's replay this example, again from the top! :)

```
STORE   R1  R2   IMM     # mem[R2+IMM] <- R1
 (…)
LOAD    R3  R4   IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3   R3      # R5 <- R3+R3
```

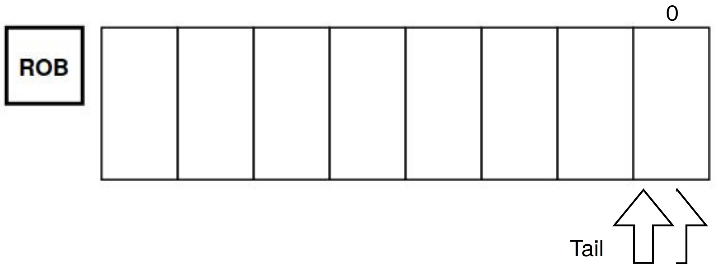


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready

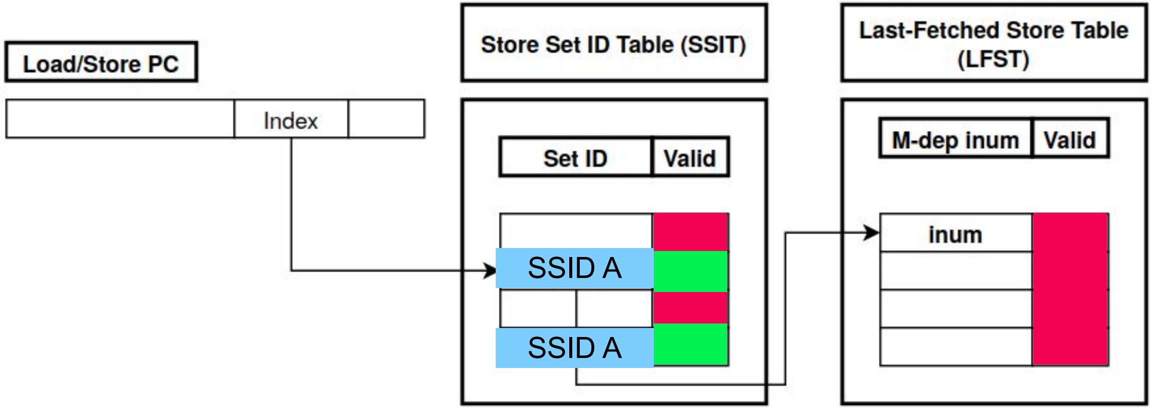


```
STORE   R1  R2   IMM      # mem[R2+IMM] <- R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3   R3       # R5 <- R3+R3
```

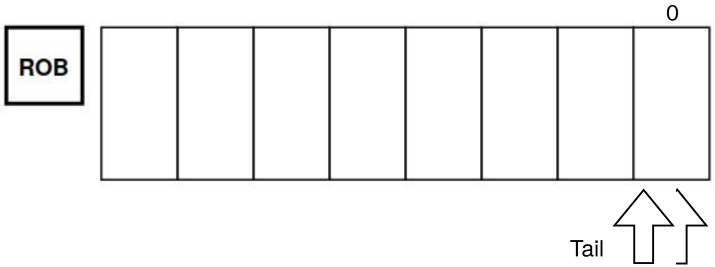


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready



```
STORE   R1  R2   IMM      # mem[R2+IMM] <- R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

 ADD    R5  R3   R3       # R5 <- R3+R3
```



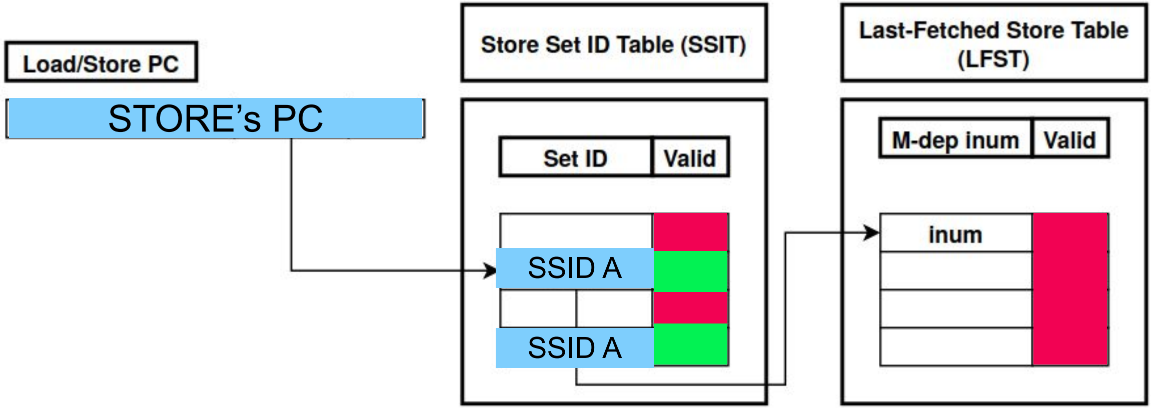
ROB

0

Tail

OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready



Load/Store PC

STORE’s PC

Store Set ID Table (SSIT)

Set ID | Valid

SSID A

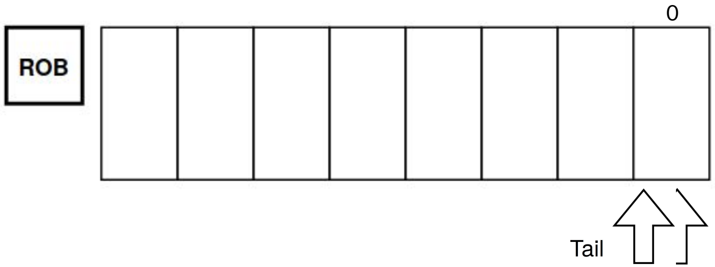
SSID A

Last-Fetched Store Table (LFST)

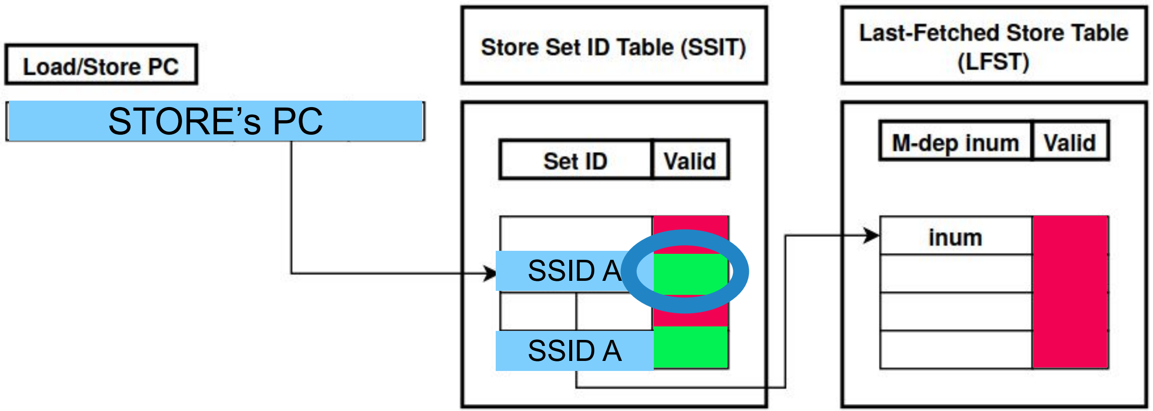
M-dep inum | Valid

inum

```
STORE   R1  R2   IMM     # mem[R2+IMM] <- R1
 (…)
 LOAD   R3  R4   IMM     # R3 <- mem[R4+IMM]

 ADD    R5  R3   R3      # R5 <- R3+R3
```

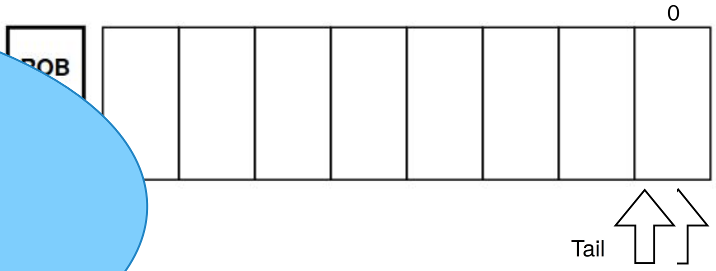


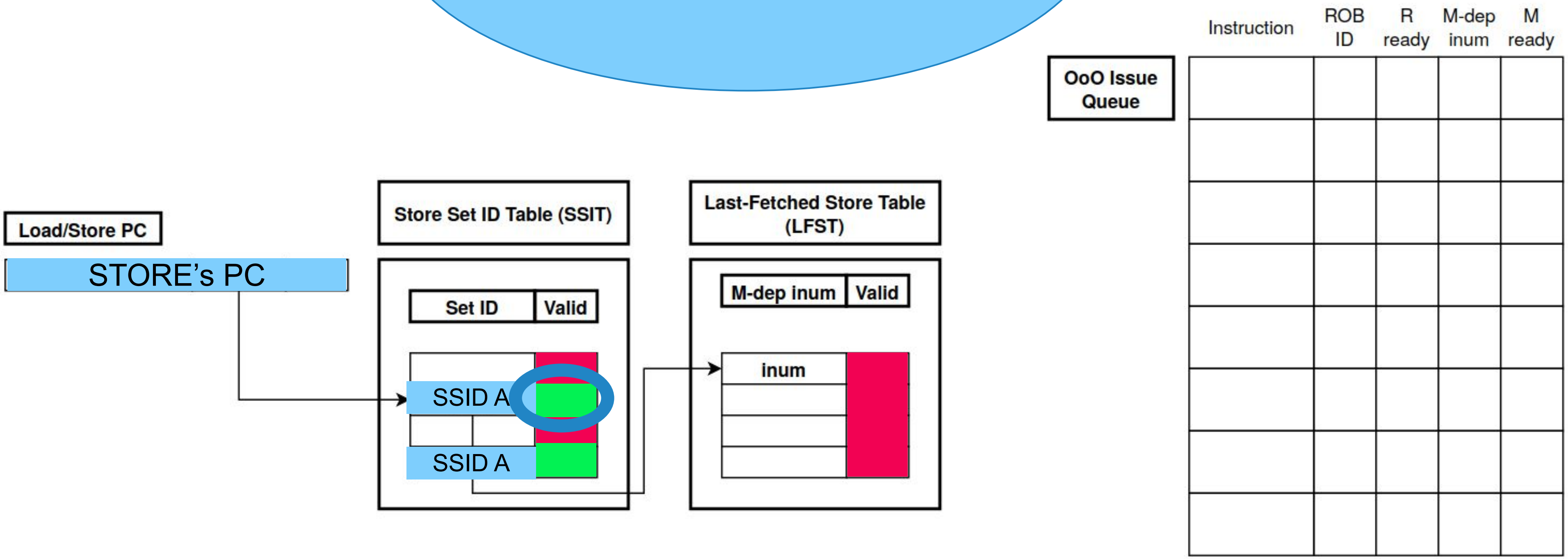
	Instruction	ROB ID	R ready	M-dep inum	M ready
OoO Issue Queue					



```
STORE   R1  R2  IMM
 (…)
 LOAD   R3  R4  I

  ADD   R5  R3
```

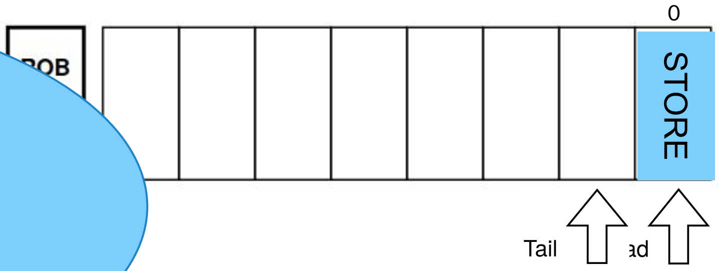
The STORE must now **notify** that it was the last STORE to be fetched…





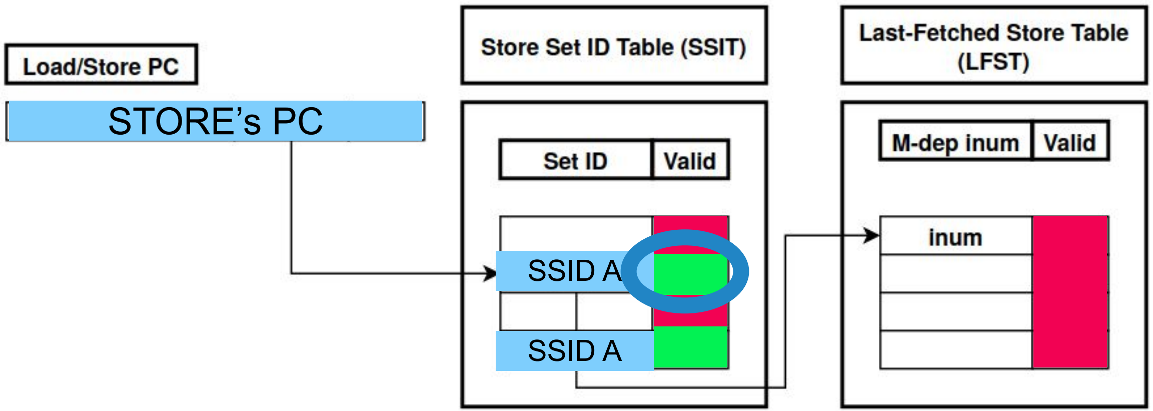
```
STORE   R1  R2  IMM
 (…)
 LOAD   R3  R4  I

 ADD    R5  R3  I
```

The STORE must now **notify** that it was the last STORE to be fetched…



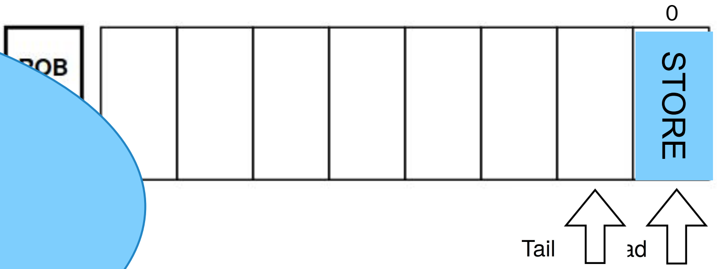
Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩

OoO Issue Queue



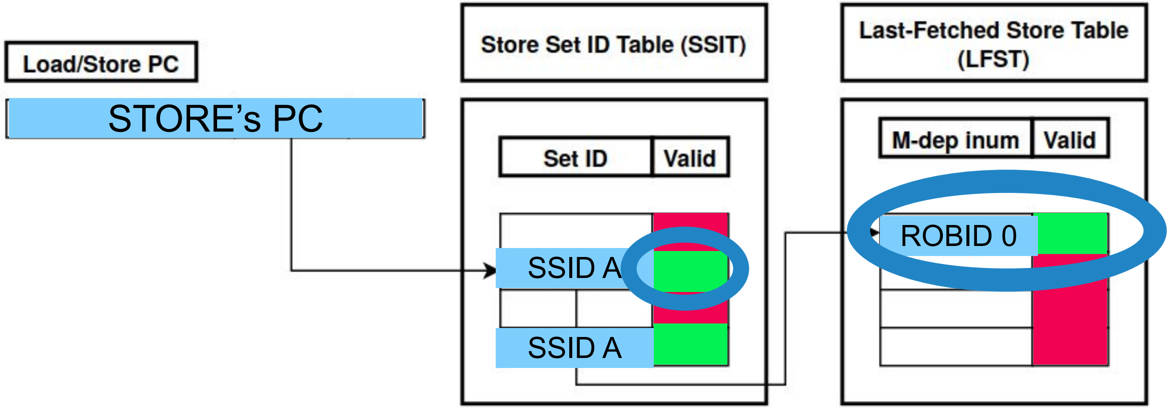
```
STORE   R1  R2  IMM
 (…)
 LOAD   R3  R4  I

 ADD    R5  R3
```

The STORE must now **notify** that it was the last STORE to be fetched…



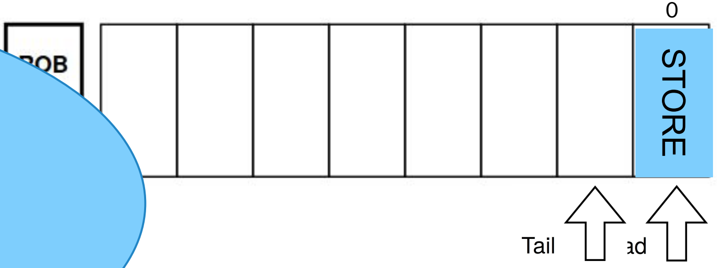
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩



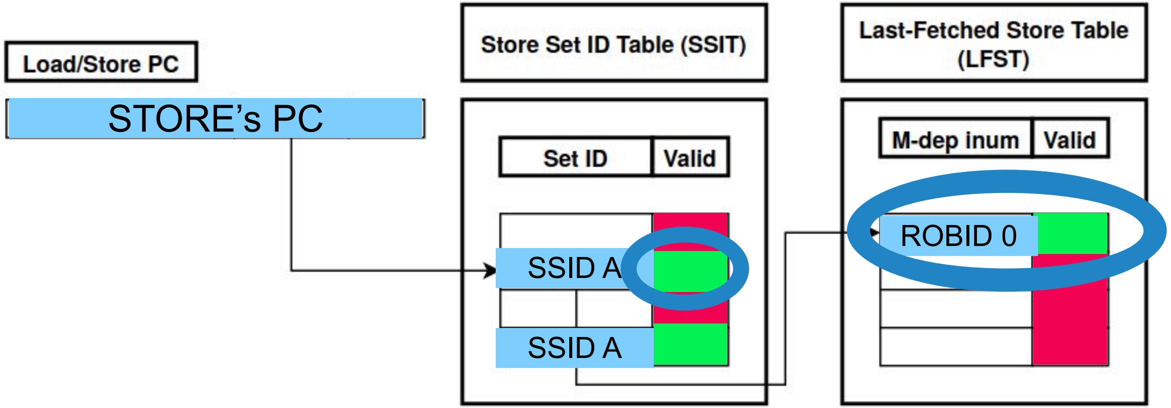
```
STORE   R1  R2  IMM
 (…)
LOAD    R3  R4  I

ADD     R5  R3  I
```

Now the LOAD will know it should wait for a wake-up of the STORE whose ROB ID is #0…



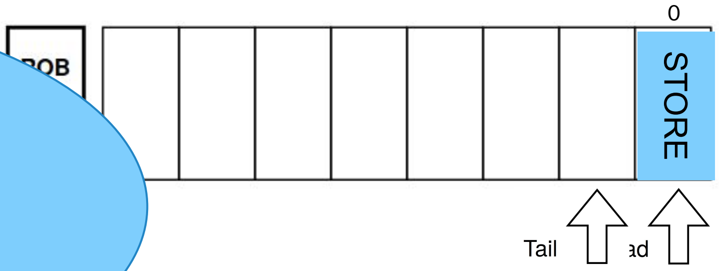
OoO Issue Queue

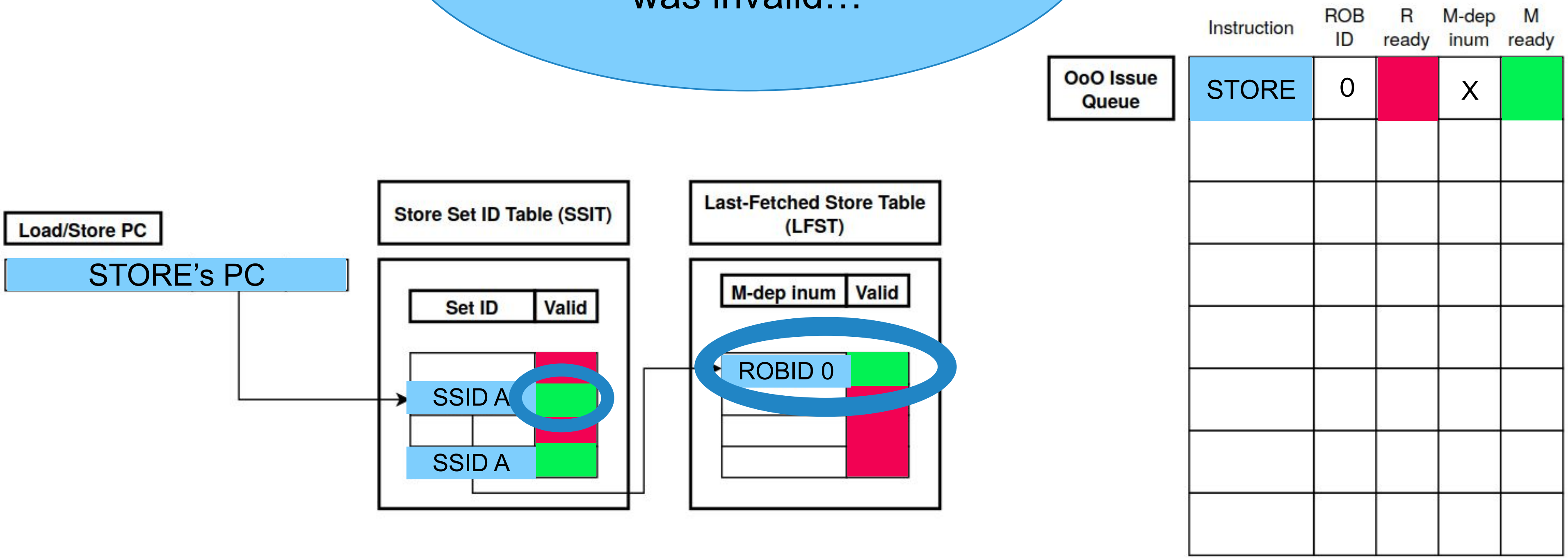
Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩



```
STORE  R1  R2  IMM
 (…)
 LOAD  R3  R4  I
 ADD   R5  R3
```

Notice also that the STORE is M-ready since the previous LFST entry was invalid…



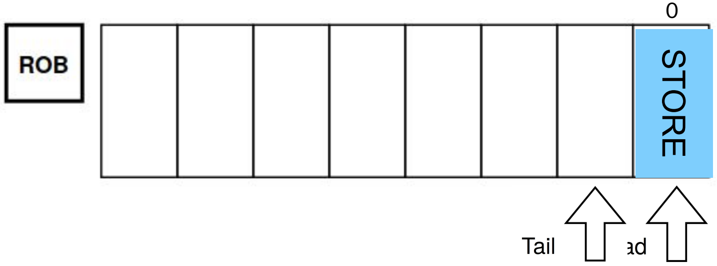


Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

M-dep inum	Valid
ROBID 0	🟩
	🟥
	🟥
	🟥

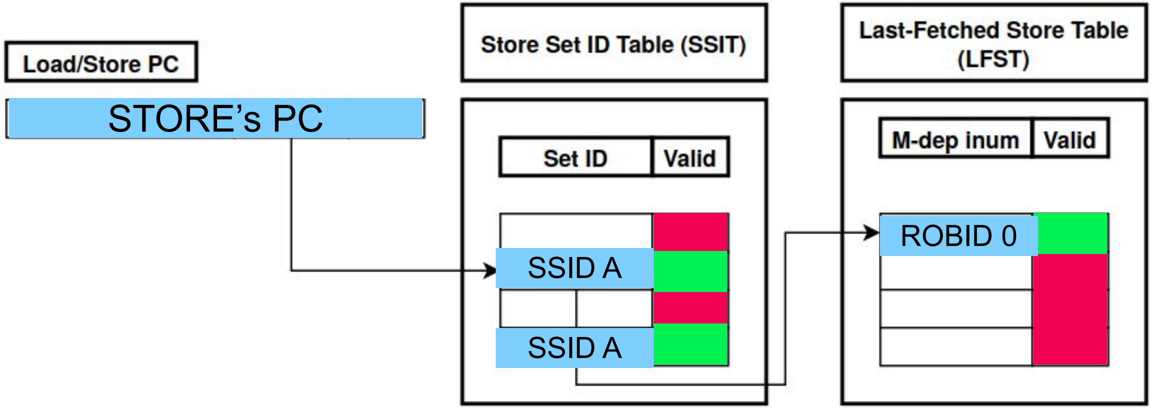
Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩

```
STORE   R1  R2   IMM      # mem[Addr ???] <-  R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3   R3       # R5 <- R3+R3
```

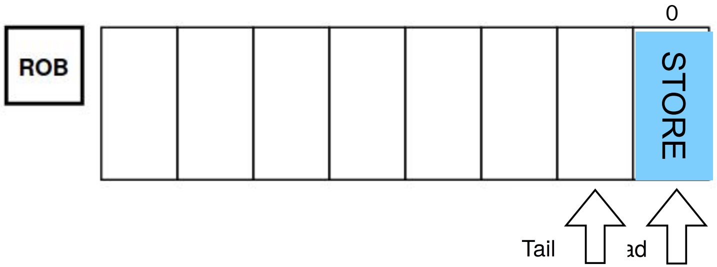


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩



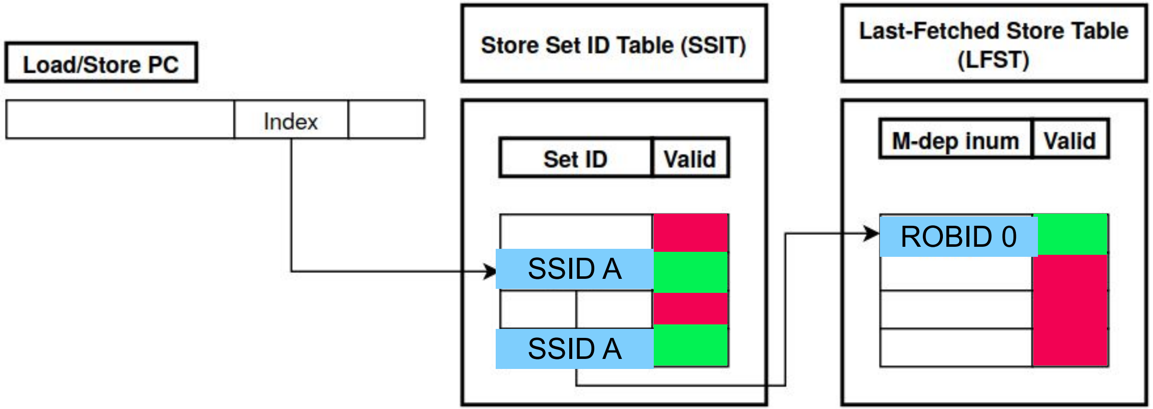
```
STORE   R1  R2  IMM     # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3   R3     # R5 <- R3+R3
```



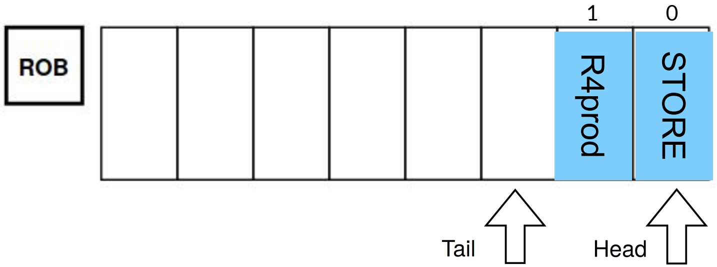
R4-producer instr.

OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩



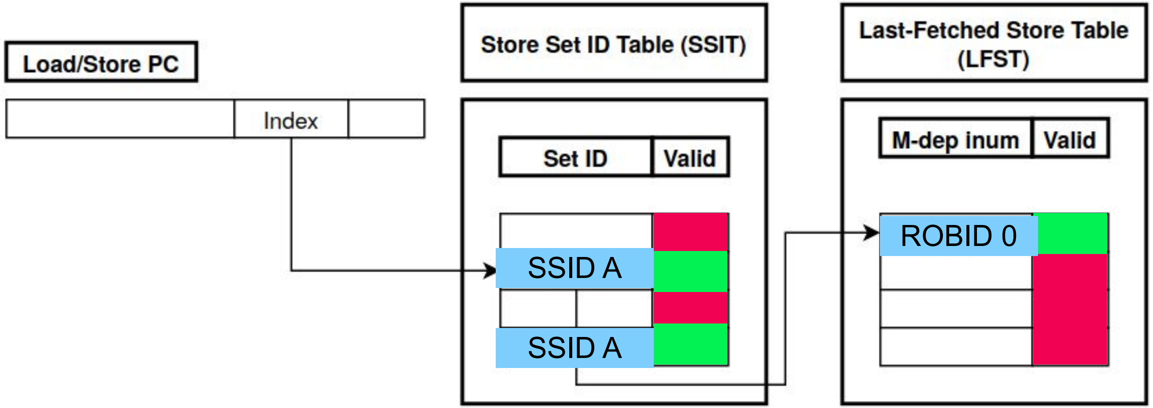
```
STORE   R1  R2  IMM     # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3  R3      # R5 <- R3+R3
```



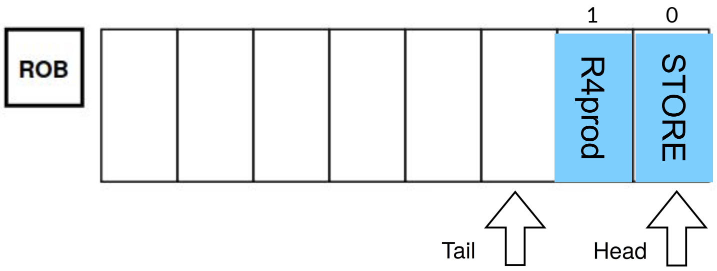
R4-producer instr.

OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩

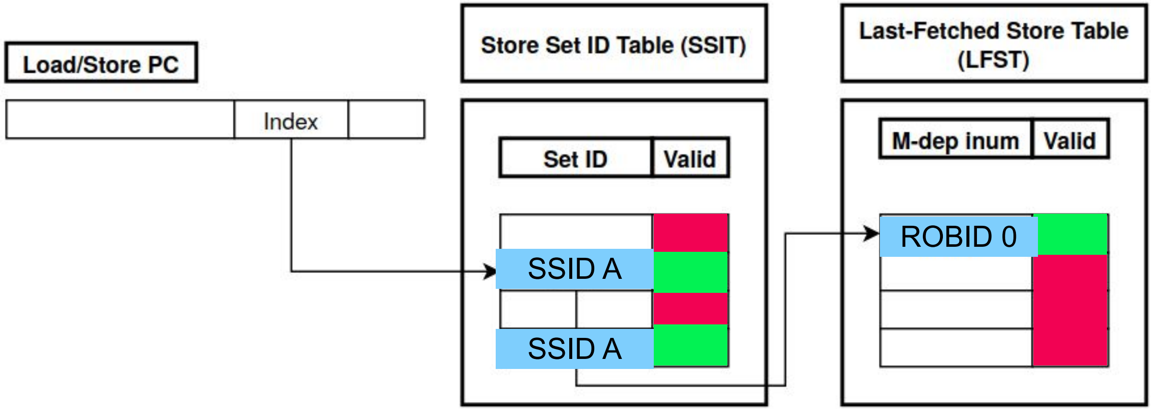


```
STORE   R1  R2  IMM     # mem[ Addr ??? ] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3  R3      # R5 <- R3+R3
```



OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩



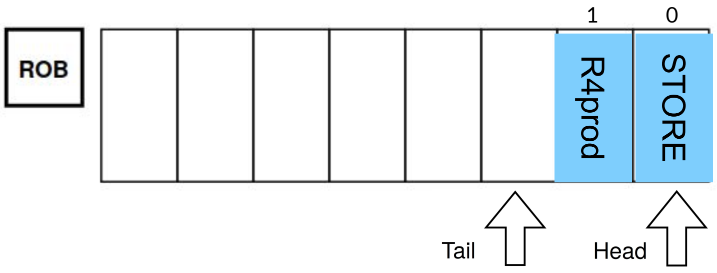
Store Set ID Table (SSIT)

Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

Last-Fetched Store Table (LFST)

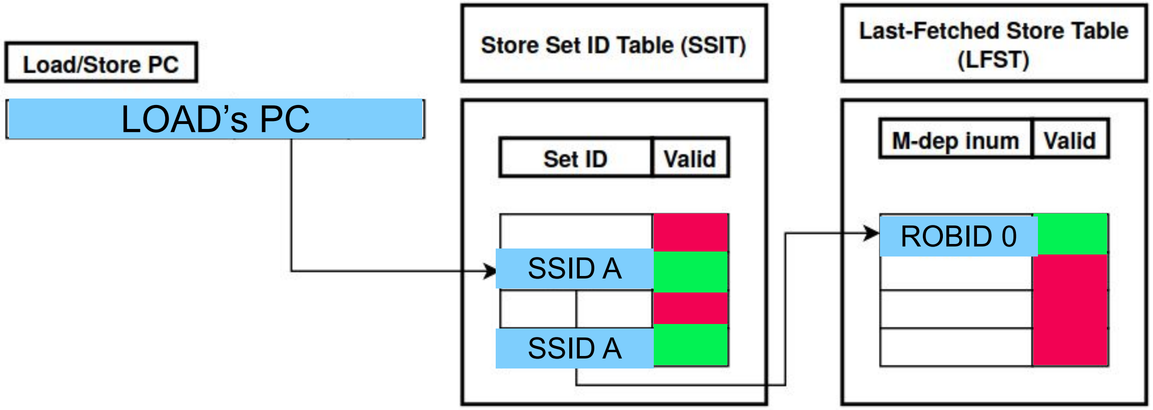
M-dep inum	Valid
ROBID 0	🟩
	🟥
	🟥
	🟥

```
STORE   R1  R2  IMM     # mem[ Addr ??? ] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3  R3      # R5 <- R3+R3
```

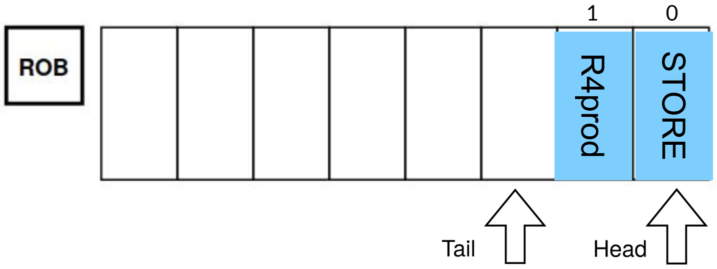


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩

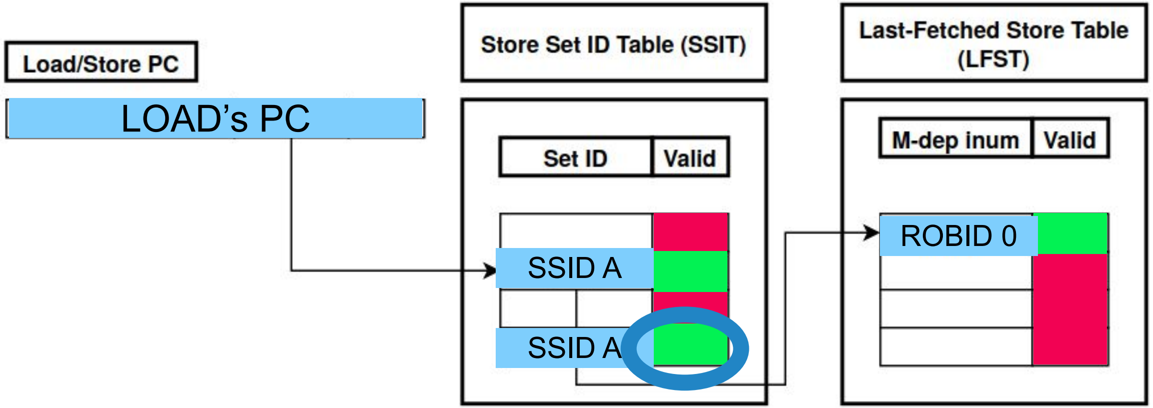


```
STORE   R1  R2  IMM     # mem[Addr ???] <- R1
 (...)
LOAD    R3  R4  IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3  R3      # R5 <- R3+R3
```

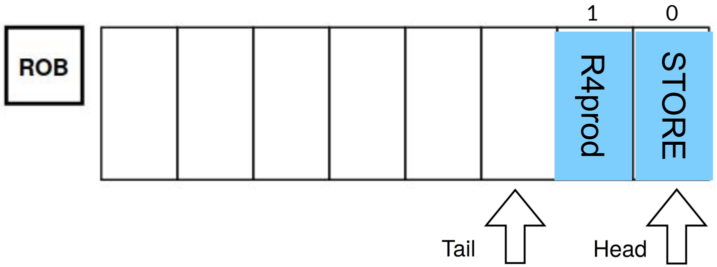


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩

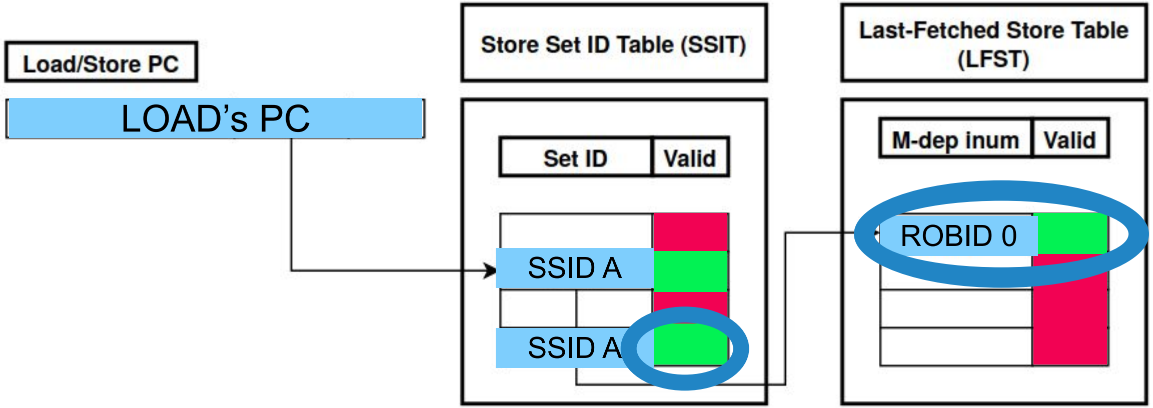


```
STORE    R1  R2  IMM     # mem[Addr ???] <- R1
 (…)
LOAD     R3  R4  IMM     # R3 <- mem[R4+IMM]

ADD      R5  R3  R3      # R5 <- R3+R3
```

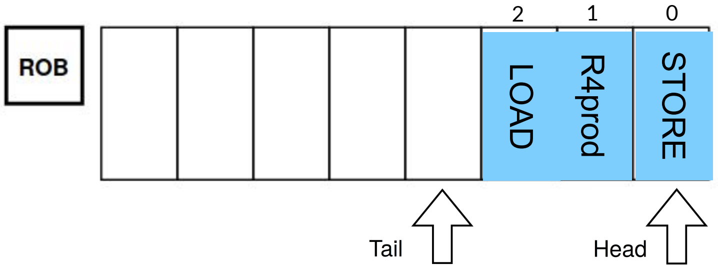


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩

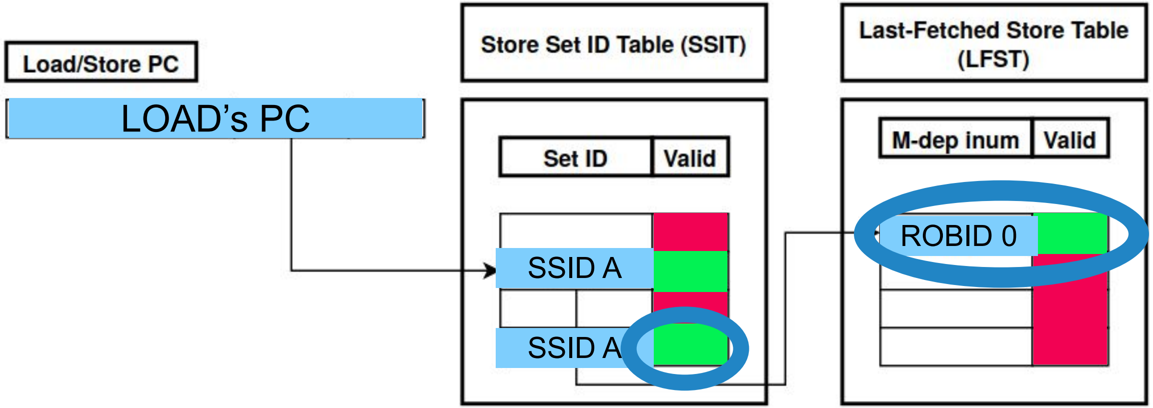


```
STORE   R1  R2   IMM      # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3   R3       # R5 <- R3+R3
```

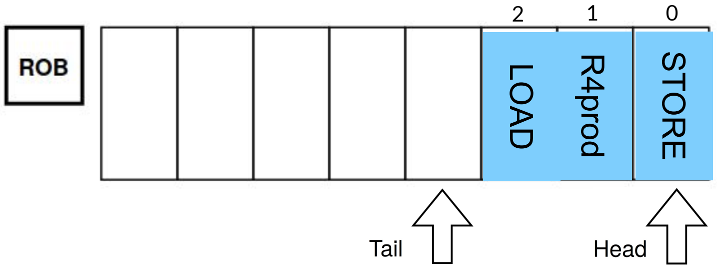


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
LOAD	2	🟥	0	🟥

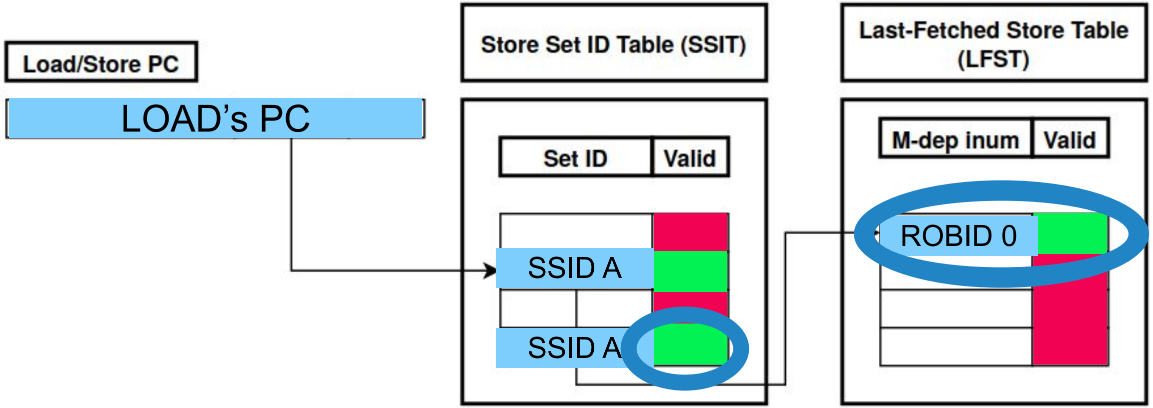


```
STORE   R1  R2   IMM      # mem[Addr ???] <-  R1
 (…)
LOAD    R3  R4   IMM      # R3 <- me[Addr ???]

ADD     R5  R3   R3       # R5 <- R3+R3
```

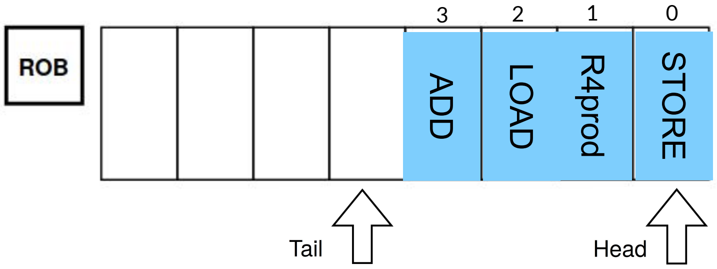


OoO Issue Queue

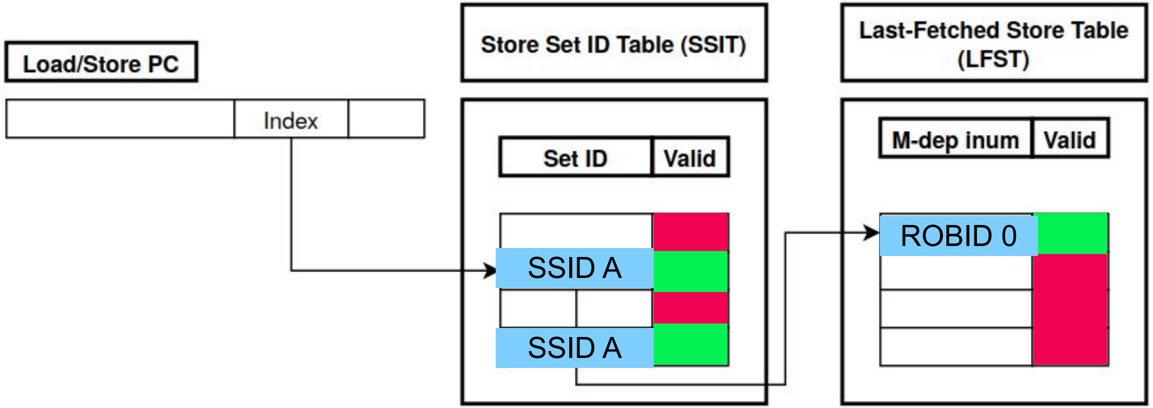
Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
LOAD	2	🟥	0	🟥



```
   STORE   R1  R2  IMM     # mem[ Addr ??? ] <- R1
   (…)
   LOAD    R3  R4  IMM     # R3 <- mem[ Addr ??? ]

➡  ADD     R5  R3  R3      # R5 <- R3+R3
```



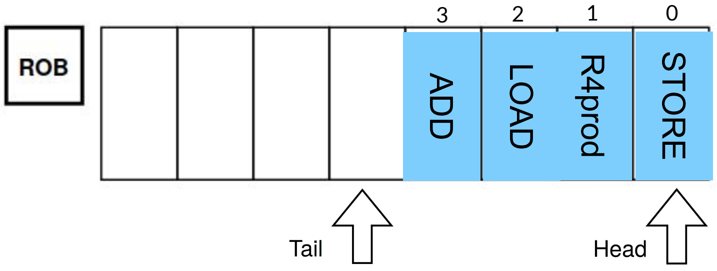
ADD



OoO Issue Queue

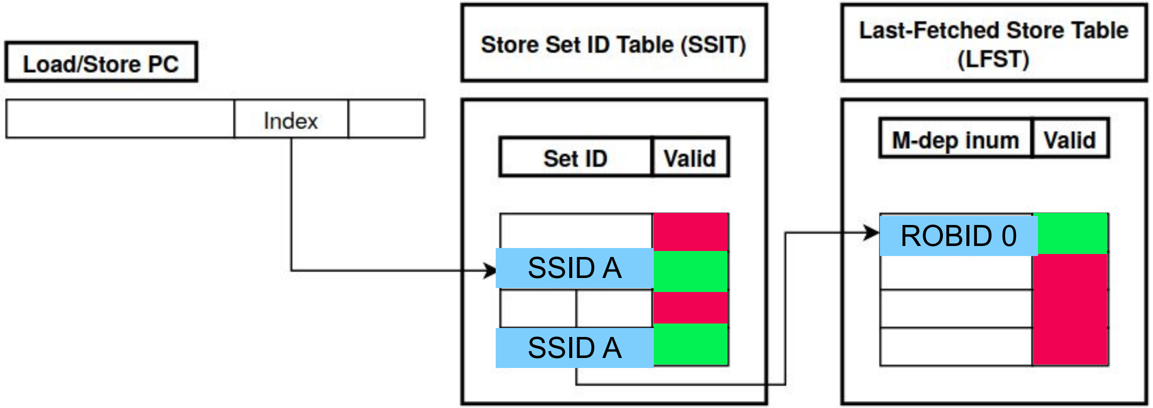
Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
LOAD	2	🟥	0	🟥
ADD	3	🟥	X	🟩

```
STORE   R1  R2  IMM      # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4  IMM      # R3 <- mem[Addr ???]

ADD     R5  R3  R3       # R5 <- R3+R3
```

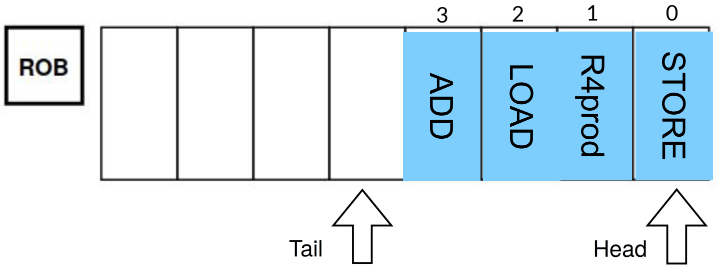


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
LOAD	2	🟥	0	🟥
ADD	3	🟥	X	🟩

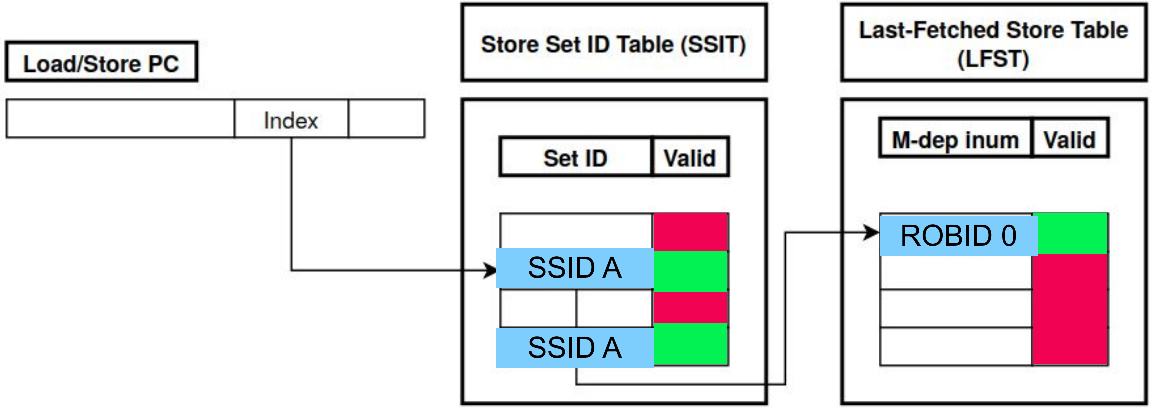


```
STORE   R1  R2  IMM     # mem[ Addr ??? ] <- R1
 (...)
LOAD    R3  R4  IMM     # R3 <- mem[ Addr ??? ]

ADD     R5  R3  R3      # R5 <- R3+R3
```



OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X !!	🟩
R4prod	1	🟩	X	🟩
LOAD	2	🟥	0	🟥
ADD	3	🟥	X	🟩



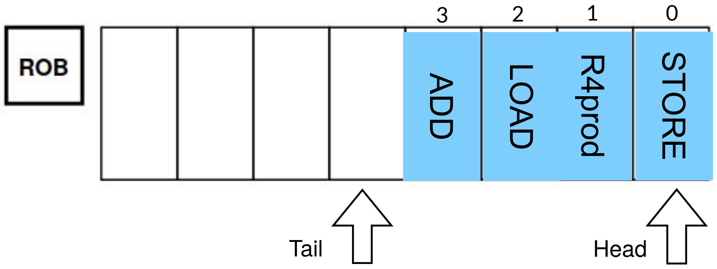
Store Set ID Table (SSIT)

Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

Last-Fetched Store Table (LFST)

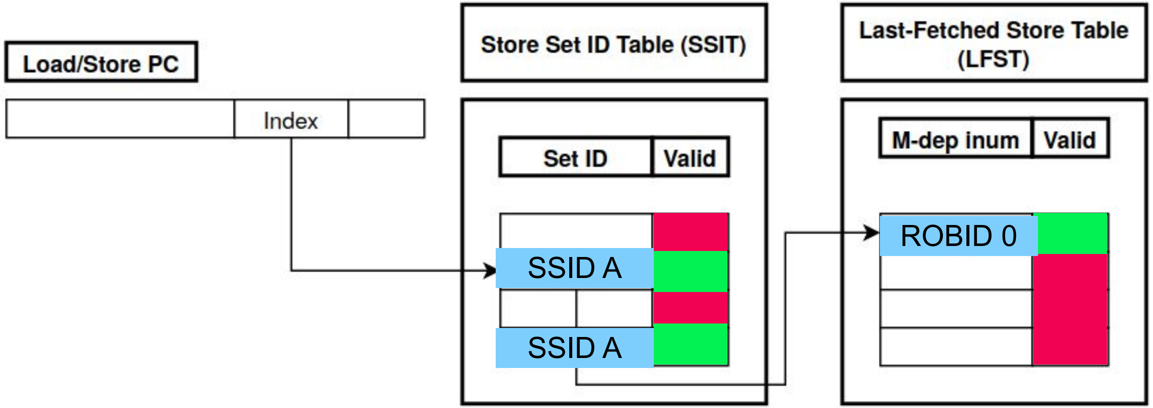
M-dep inum	Valid
ROBID 0	🟩
	🟥
	🟥
	🟥

```
STORE   R1  R2  IMM     # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[Addr ???]

ADD     R5  R3  R3      # R5 <- R3+R3
```



OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
		🟩		🟩
LOAD	2	🟥	0	🟥
ADD	3	🟥	X	🟩



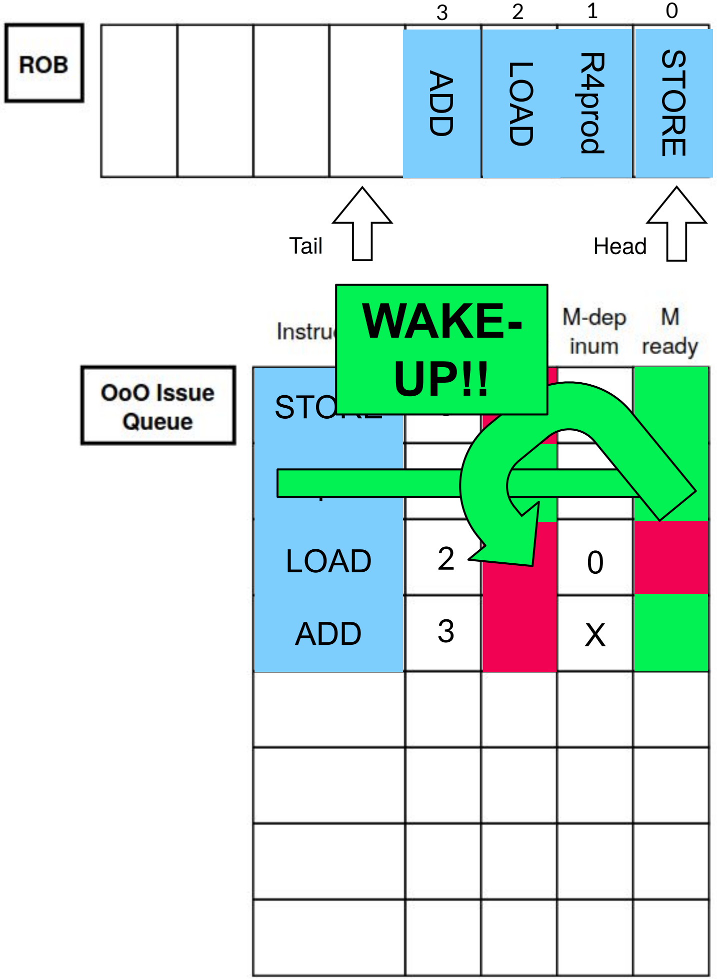
Store Set ID Table (SSIT)

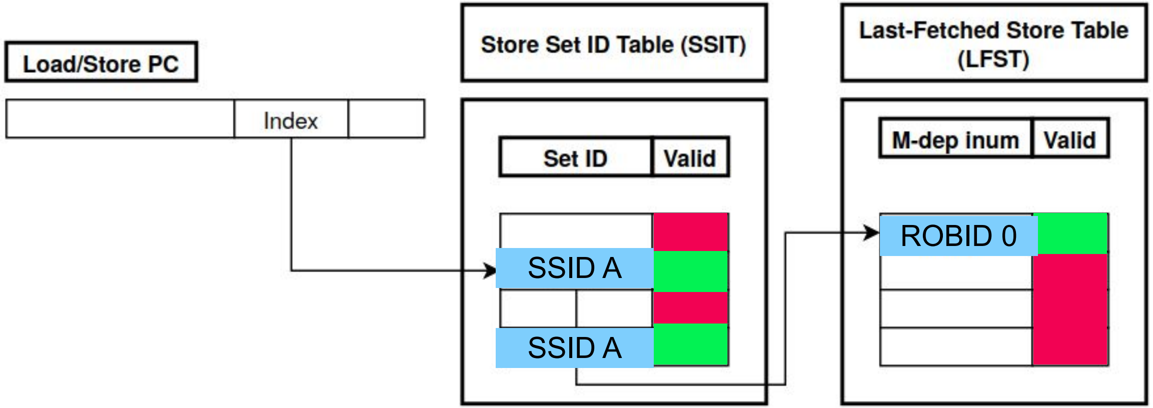
Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

Last-Fetched Store Table (LFST)

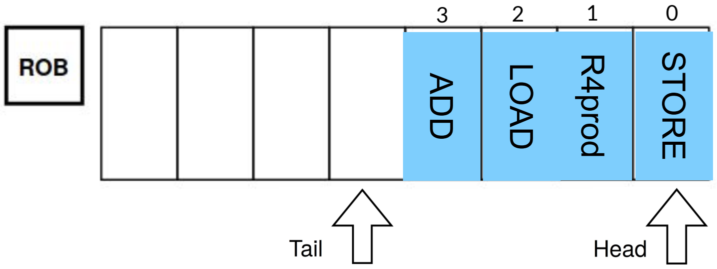
M-dep inum	Valid
ROBID 0	🟩
	🟥
	🟥
	🟥

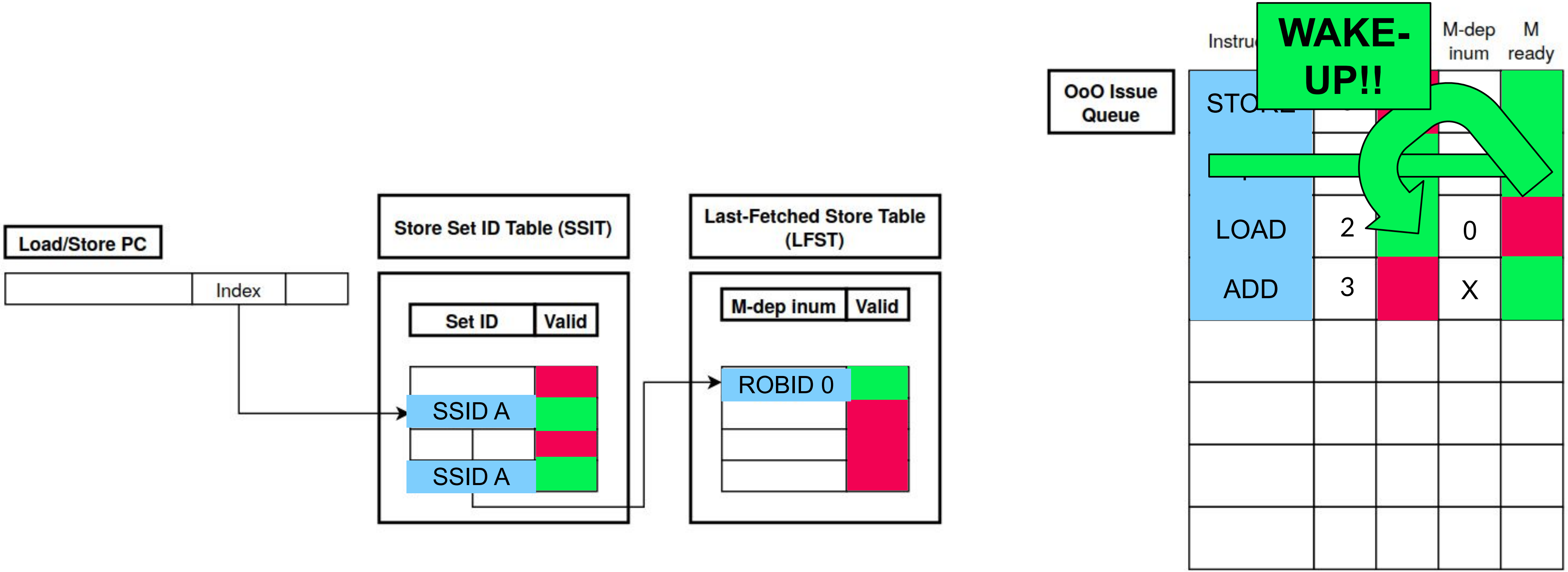
```
  STORE   R1  R2  IMM     # mem[ Addr ??? ] <- R1
  (…)
  LOAD    R3  R4  IMM     # R3 <- mem[ Addr ??? ]

➡ ADD     R5  R3  R3      # R5 <- R3+R3
```



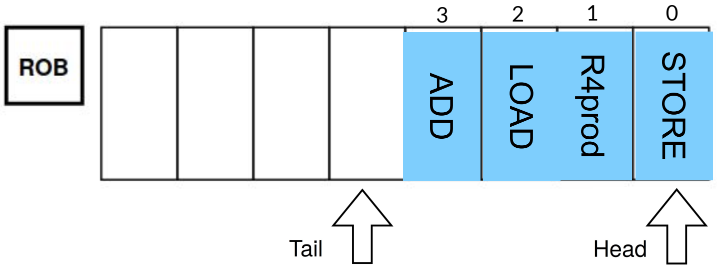


```
STORE    R1  R2   IMM      # mem[Addr ???] <- R1
 (…)
LOAD     R3  R4   IMM      # R3 <- mem[Addr A]

ADD      R5  R3   R3       # R5 <- R3+R3
```



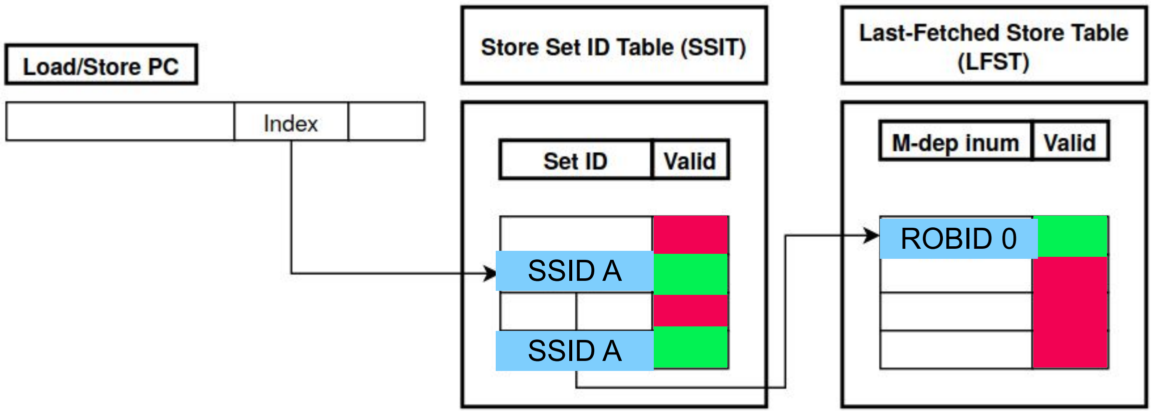


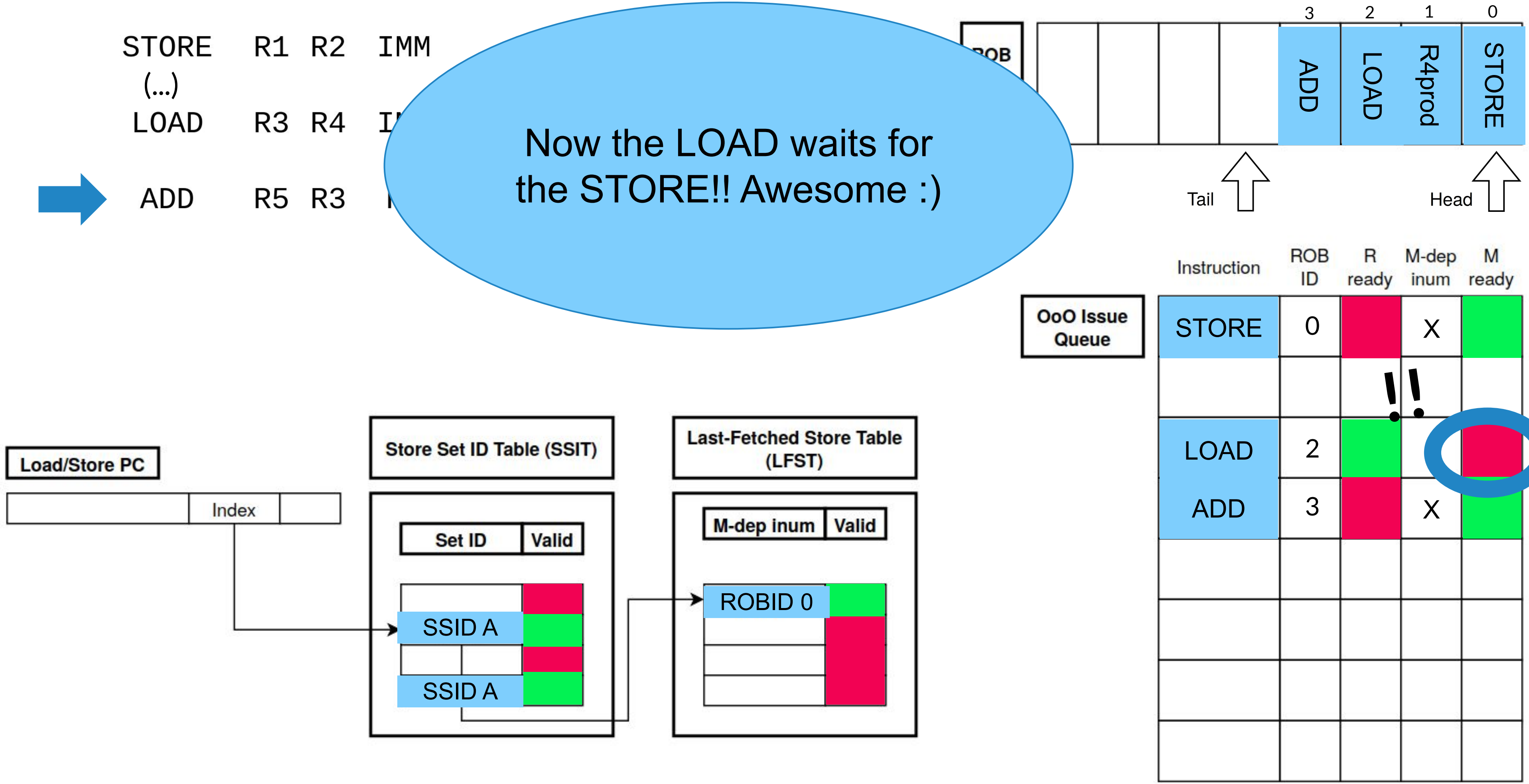
```
STORE   R1  R2  IMM     # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[Addr A]

ADD     R5  R3  R3      # R5 <- R3+R3
```



OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
			!!	
LOAD	2	🟩		🟥
ADD	3	🟥	X	🟩





```
STORE    R1  R2  IMM
 (…)
LOAD     R3  R4  I

ADD      R5  R3
```

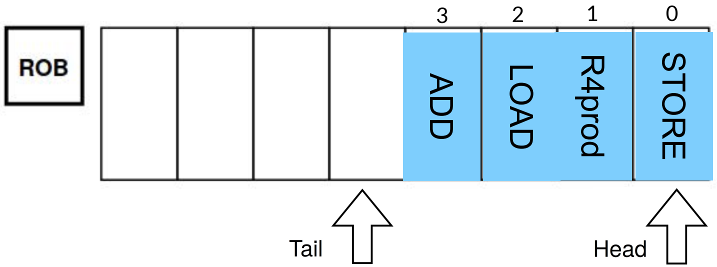
Now the LOAD waits for the STORE!! Awesome :)

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
LOAD	2	🟩		🟥
ADD	3	🟥	X	🟩

Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

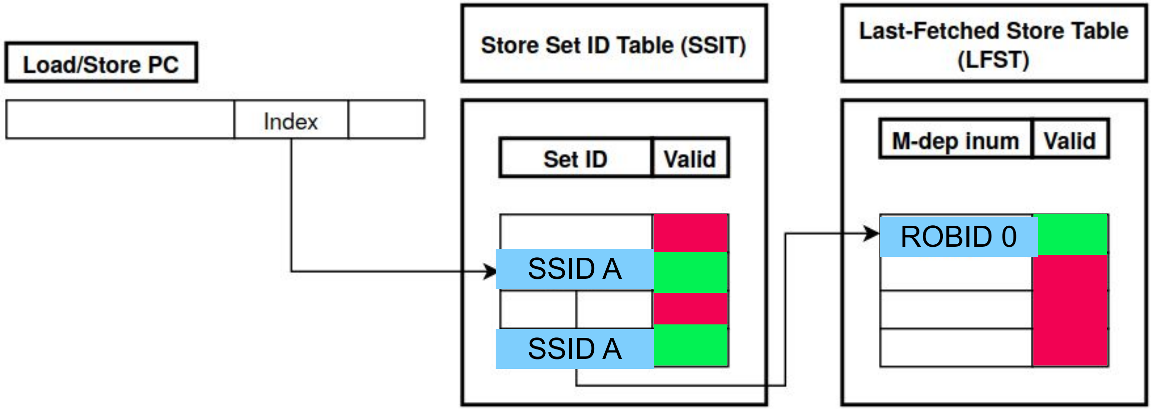
M-dep inum	Valid
ROBID 0	🟩
	🟥
	🟥
	🟥

```
  STORE   R1  R2  IMM      # mem[Addr ???] <- R1
   (…)
  LOAD    R3  R4  IMM      # R3 <- mem[Addr A]

➡ ADD     R5  R3  R3       # R5 <- R3+R3
```

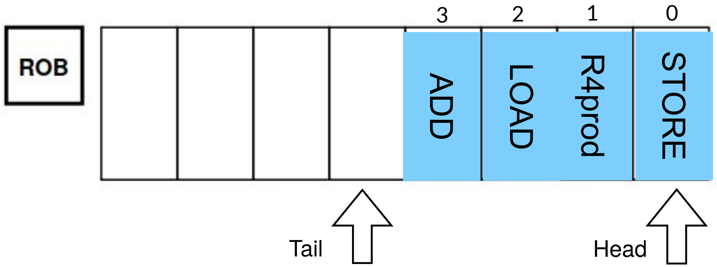


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
LOAD	2	🟩	0	🟥
ADD	3	🟥	X	🟩



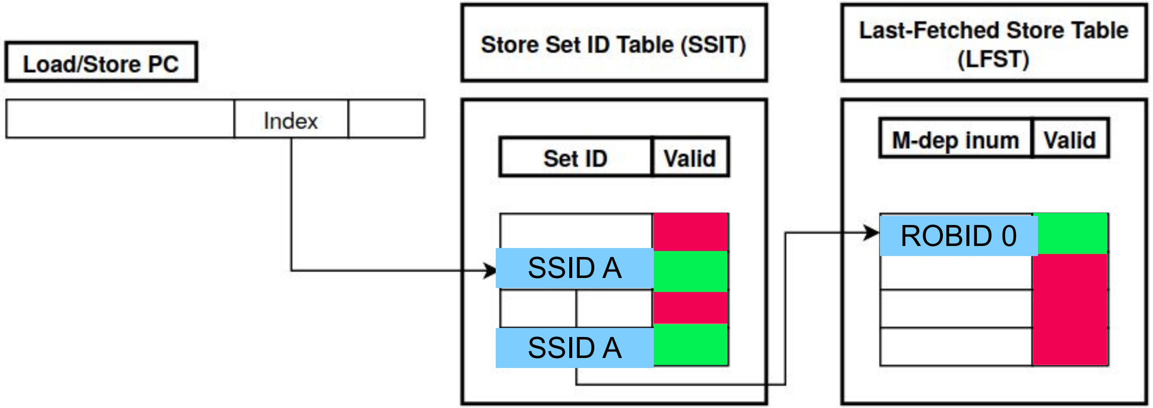
```
STORE   R1  R2   IMM      # mem[ Addr A ] <- R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[ Addr A ]

ADD     R5  R3   R3       # R5 <- R3+R3
```



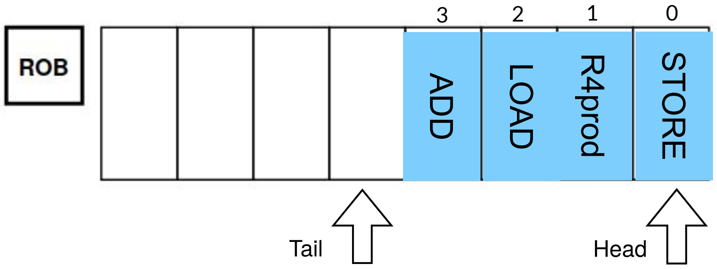
				3	2	1	0
				ADD	LOAD	R4prod	STORE

OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟩	X	🟩
LOAD	2	🟩	0	🟥
ADD	3	🟥	X	🟩

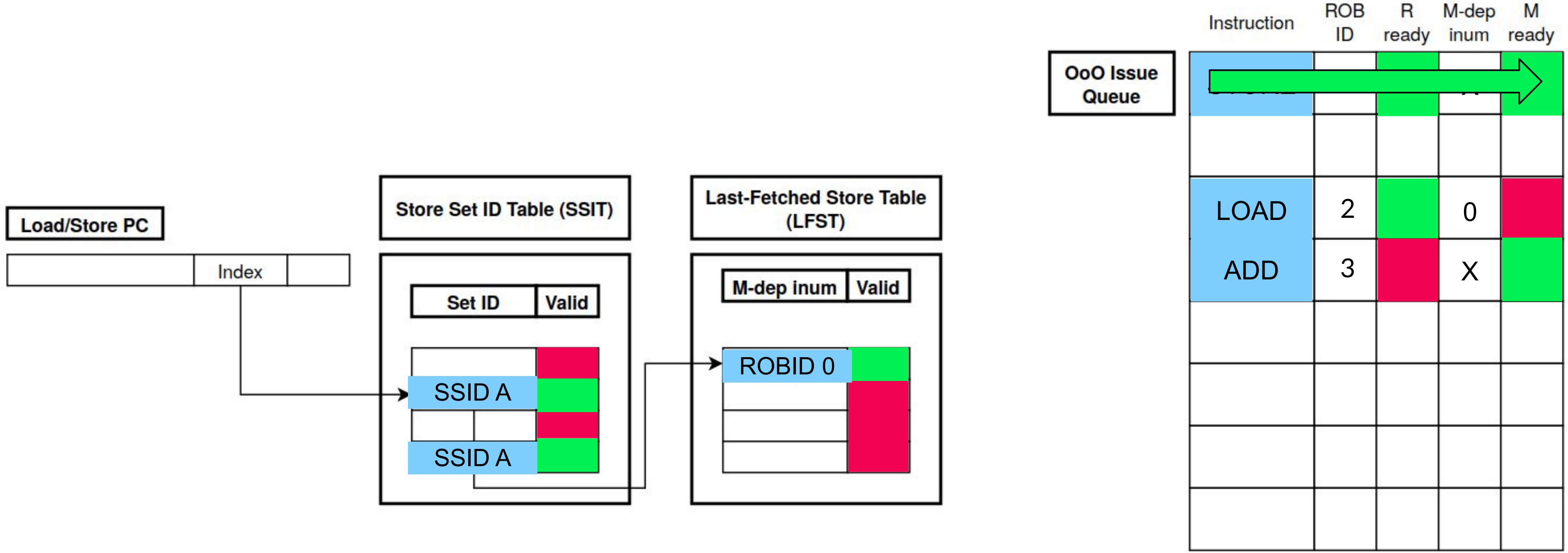


```
STORE   R1  R2  IMM      # mem[Addr A] <- R1
 (…)
LOAD    R3  R4  IMM      # R3 <- mem[Addr A]

ADD     R5  R3  R3       # R5 <- R3+R3
```



ROB

3	2	1	0
ADD	LOAD	R4prod	STORE



OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE		🟩	X	🟩
LOAD	2	🟩	0	🟥
ADD	3	🟥	X	🟩

Load/Store PC

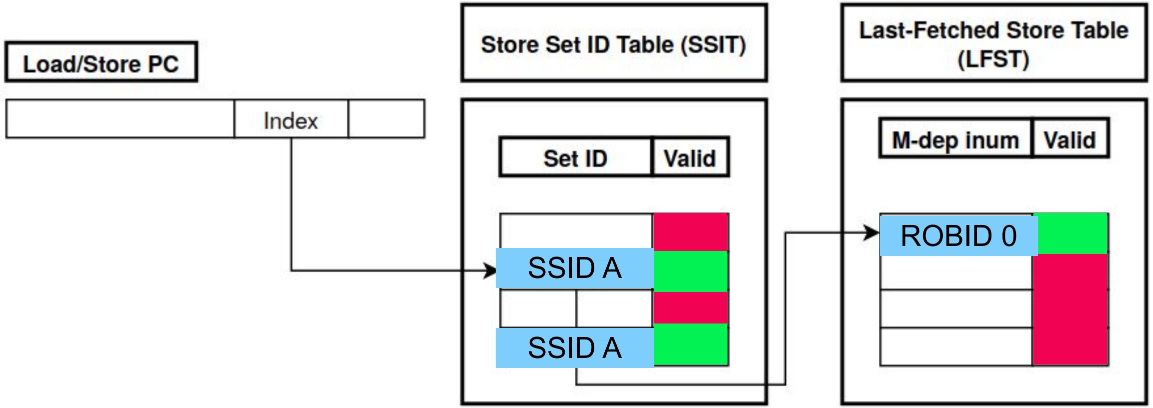
Store Set ID Table (SSIT)

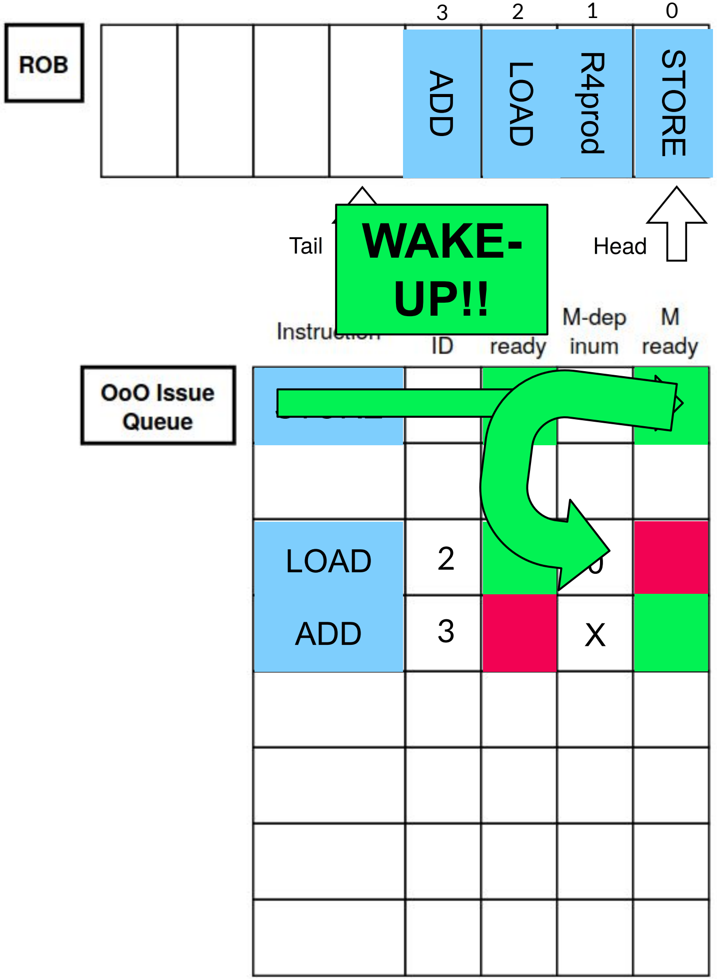
Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

Last-Fetched Store Table (LFST)

M-dep inum	Valid
ROBID 0	🟩
	🟥
	🟥
	🟥

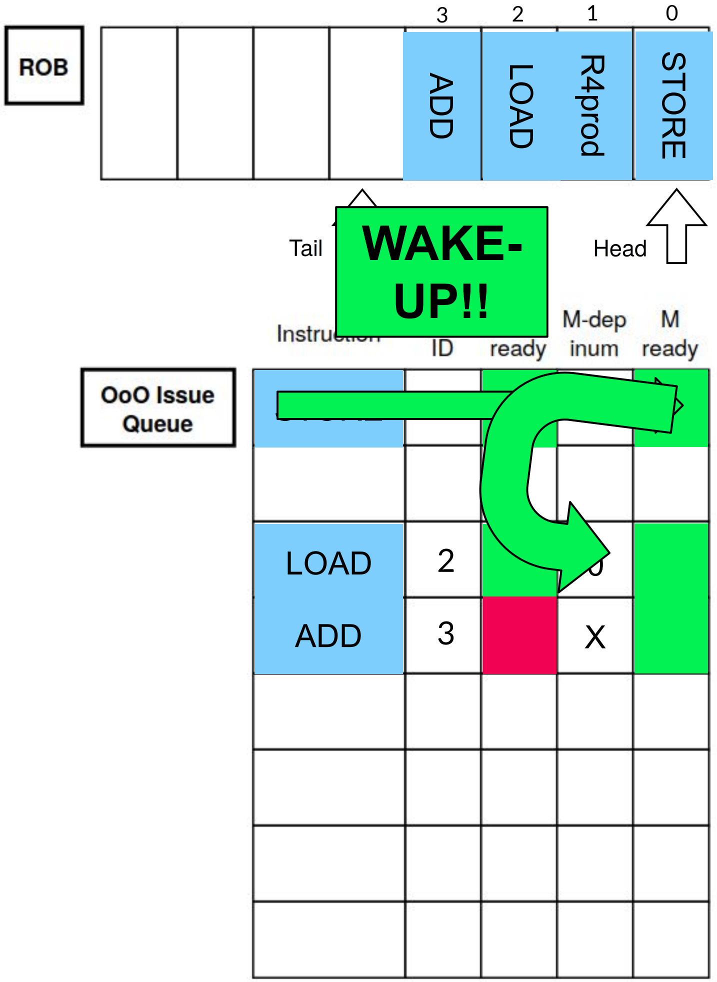
```
   STORE   R1 R2  IMM     # mem[Addr A] <- R1
   (…)
   LOAD    R3 R4  IMM     # R3 <- mem[Addr A]

➡  ADD     R5 R3  R3      # R5 <- R3+R3
```

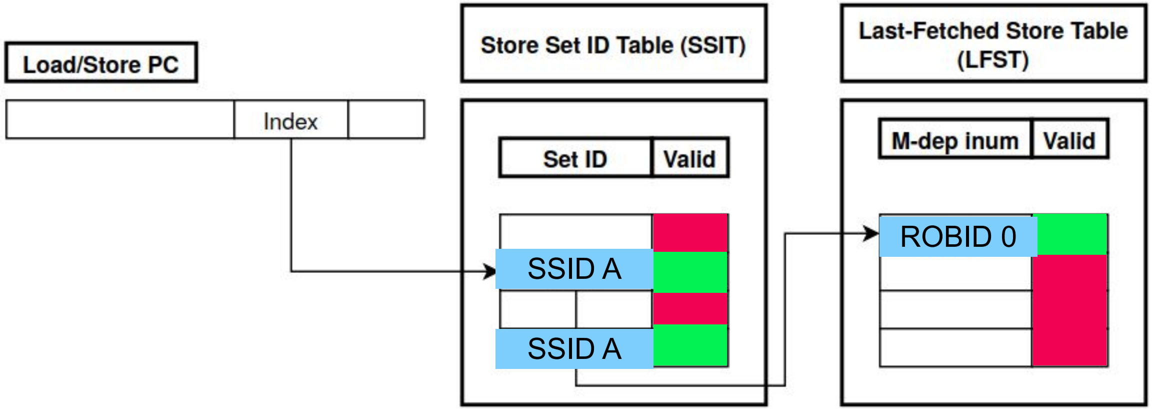




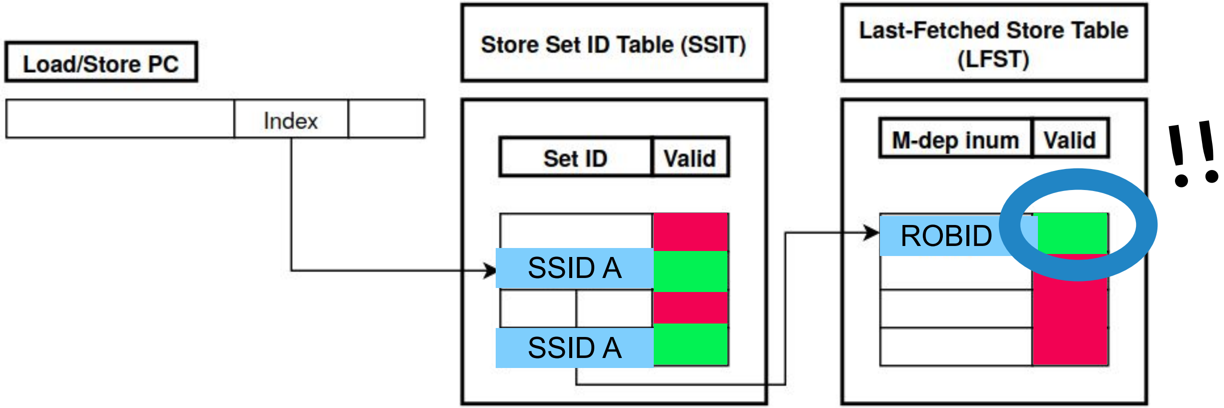
Instruction	ID	ready	M-dep inum	M ready
STORE		🟩		🟩
LOAD	2	🟩	0	🟥
ADD	3	🟥	X	🟩

```
    STORE    R1  R2   IMM      # mem[ Addr A ] <-  R1
     (…)
    LOAD     R3  R4   IMM      # R3 <-  mem[ Addr A ]

➡   ADD      R5  R3   R3       # R5 <-  R3+R3
```



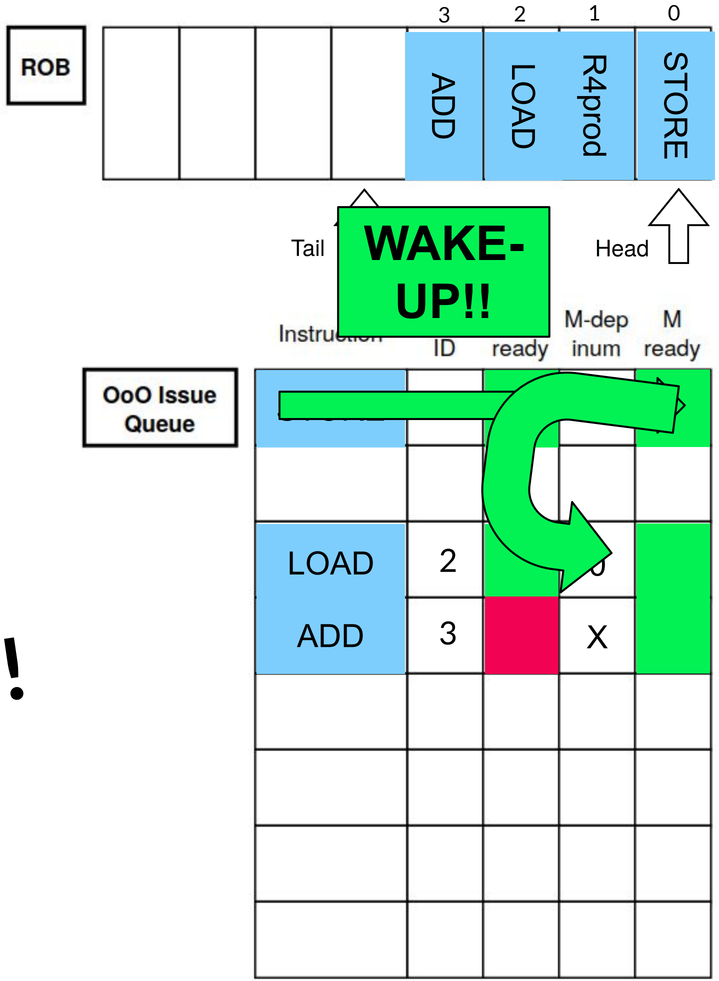


```
STORE   R1  R2  IMM      # mem[Addr A] <- R1
 (…)
LOAD    R3  R4  IMM      # R3 <- mem[Addr A]

ADD     R5  R3  R3       # R5 <- R3+R3
```



Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

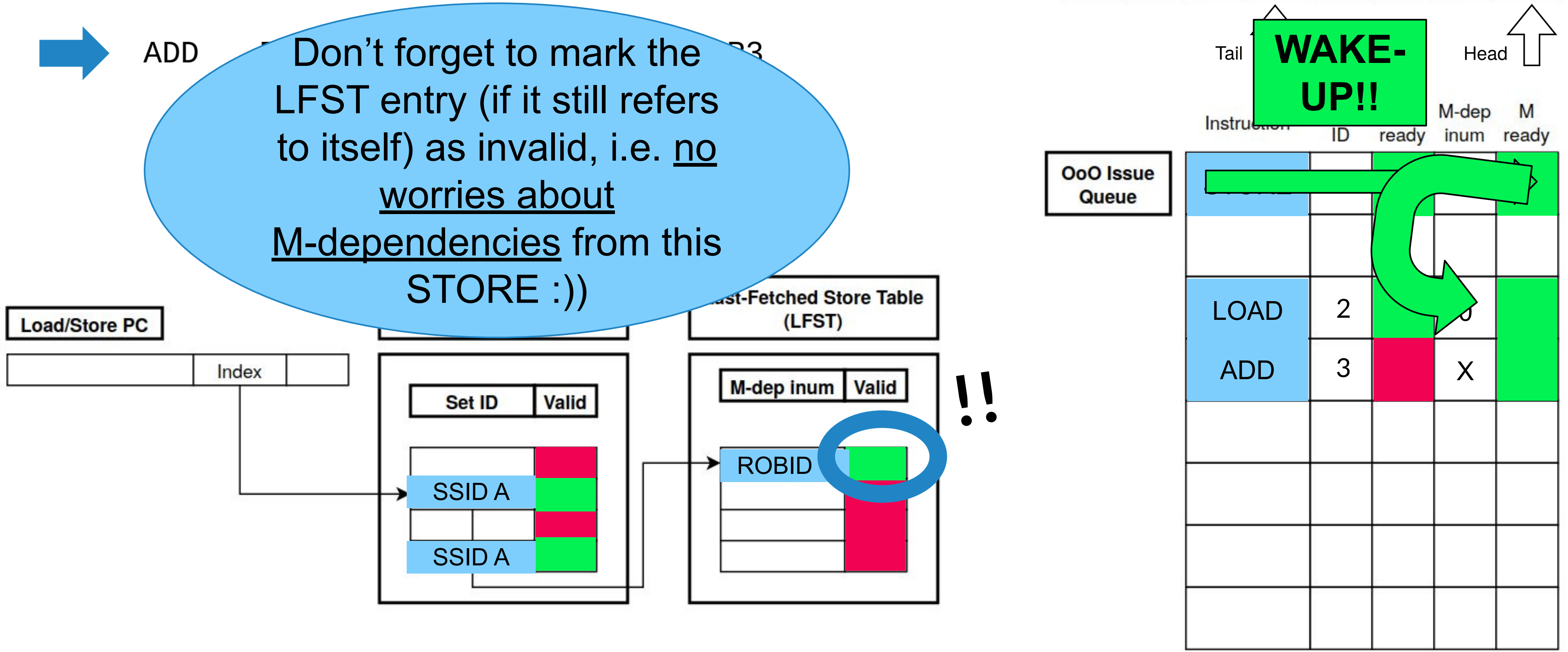
M-dep inum	Valid
ROBID	🟩
	🟥
	🟥
	🟥



				3	2	1	0
				ADD	LOAD	R4prod	STORE

Instruction	ID	ready	M-dep inum	M ready
		🟩		🟩
LOAD	2	🟩	0	🟩
ADD	3	🟥	X	🟩

```
STORE    R1  R2   IMM       # mem[ Addr A ] <- R1
 (…)
LOAD     R3  R4   IMM       # R3 <- mem[ Addr A ]

ADD                                R3
```



Don’t forget to mark the LFST entry (if it still refers to itself) as invalid, i.e. <u>no worries about M-dependencies</u> from this STORE :))

	3	2	1	0
ROB	ADD	LOAD	R4prod	STORE

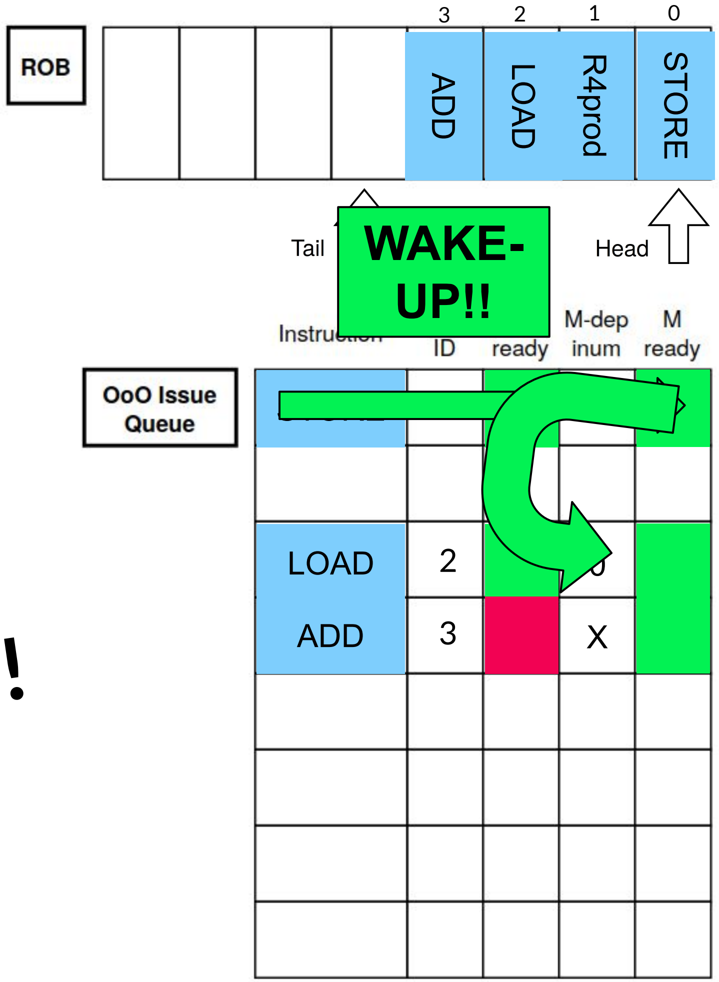
Instruction	ID	ready	M-dep inum	M ready
STORE		🟩		🟩
LOAD	2	🟩	0	🟩
ADD	3	🟥	X	🟩

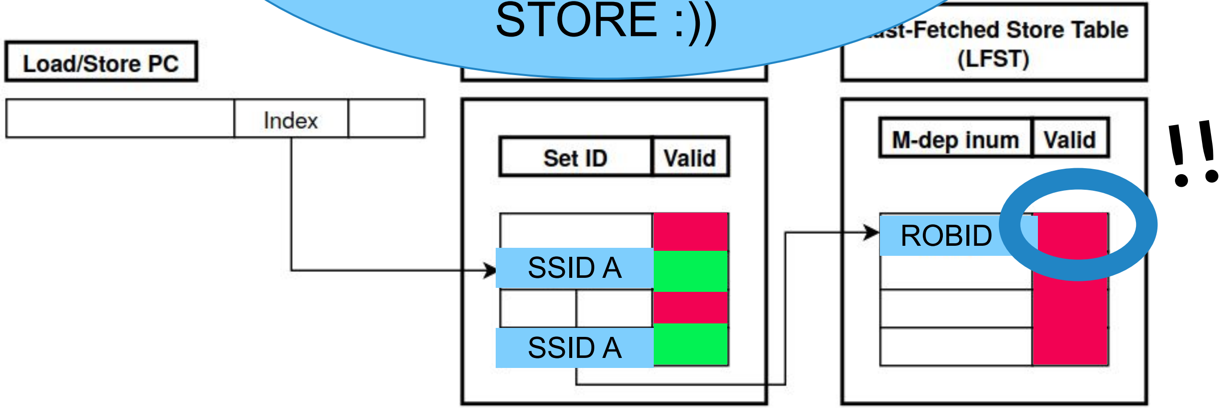
Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

M-dep inum	Valid
ROBID	🟩
	🟥
	🟥
	🟥

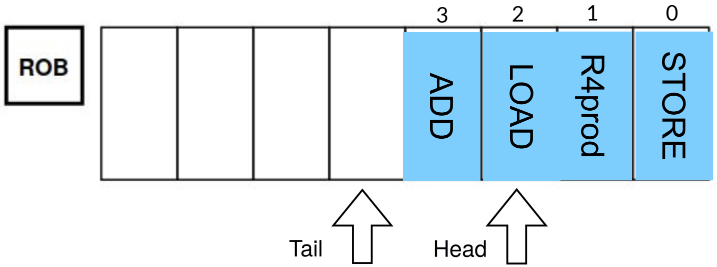
```
STORE   R1  R2  IMM     # mem[ Addr A ] <-  R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[ Addr A ]

ADD                                   R3
```

Don't forget to mark the LFST entry (if it still refers to itself) as invalid, i.e. no worries about M-dependencies from this STORE :))



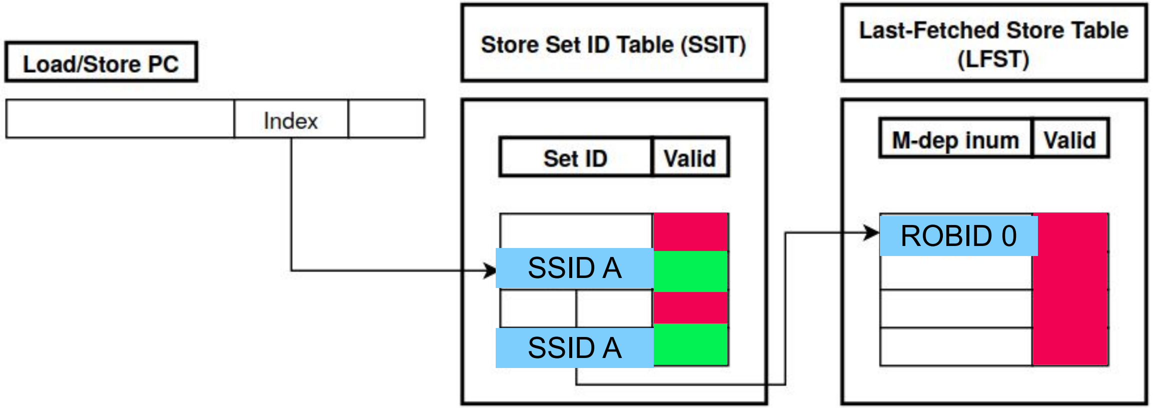


```
STORE   R1  R2  IMM     # mem[Addr A] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[Addr A]

ADD     R5  R3  R3      # R5 <- R3+R3
```

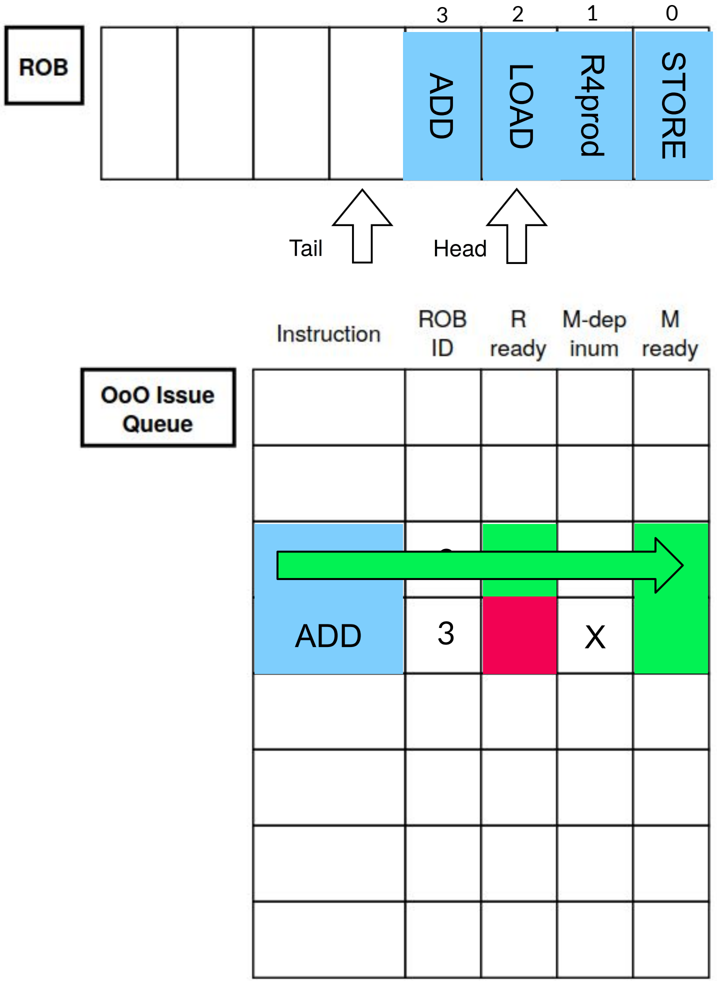


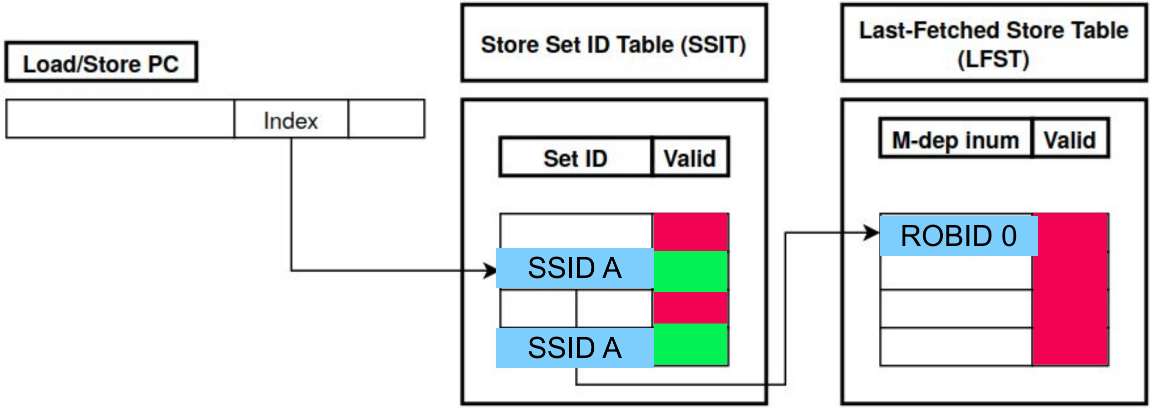
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
LOAD	2	🟩	0	🟩
ADD	3	🟥	X	🟩

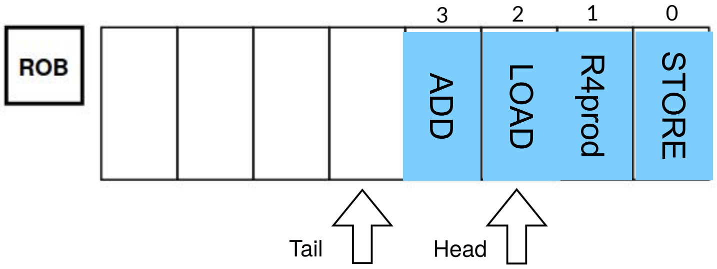


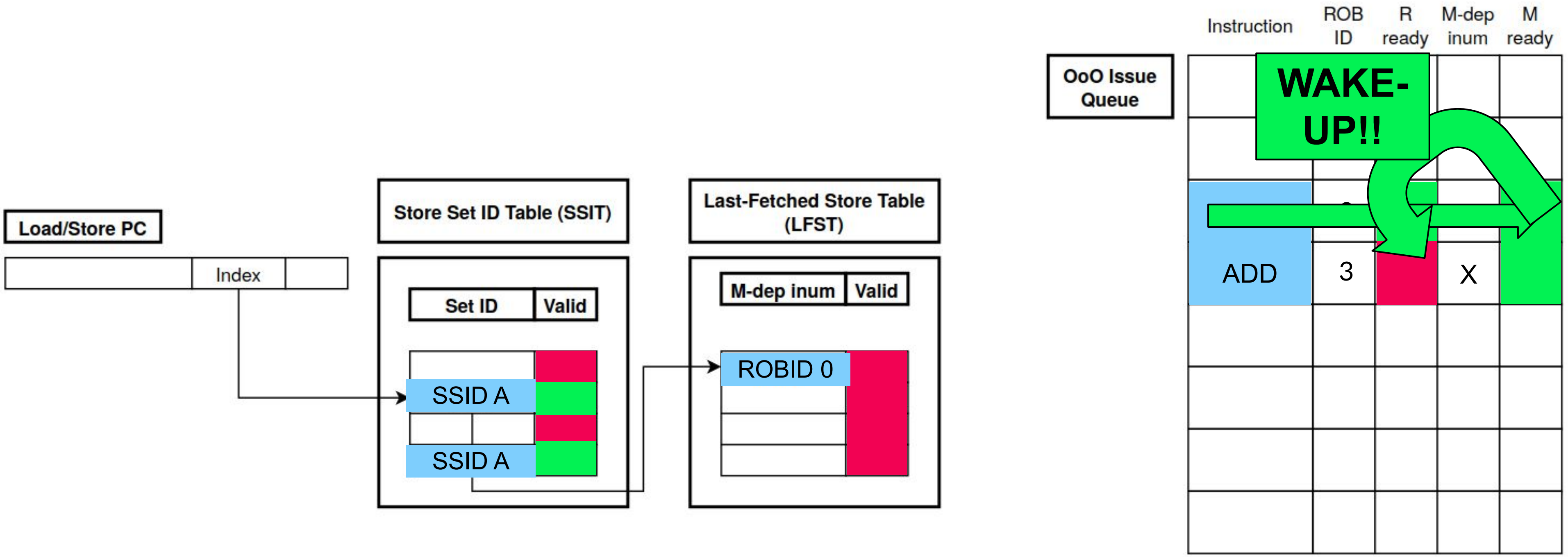
```
STORE    R1  R2  IMM      # mem[ Addr A ] <- R1
 (...)
LOAD     R3  R4  IMM      # R3 <- mem[ Addr A ]

ADD      R5  R3   R3      # R5 <- R3+R3
```





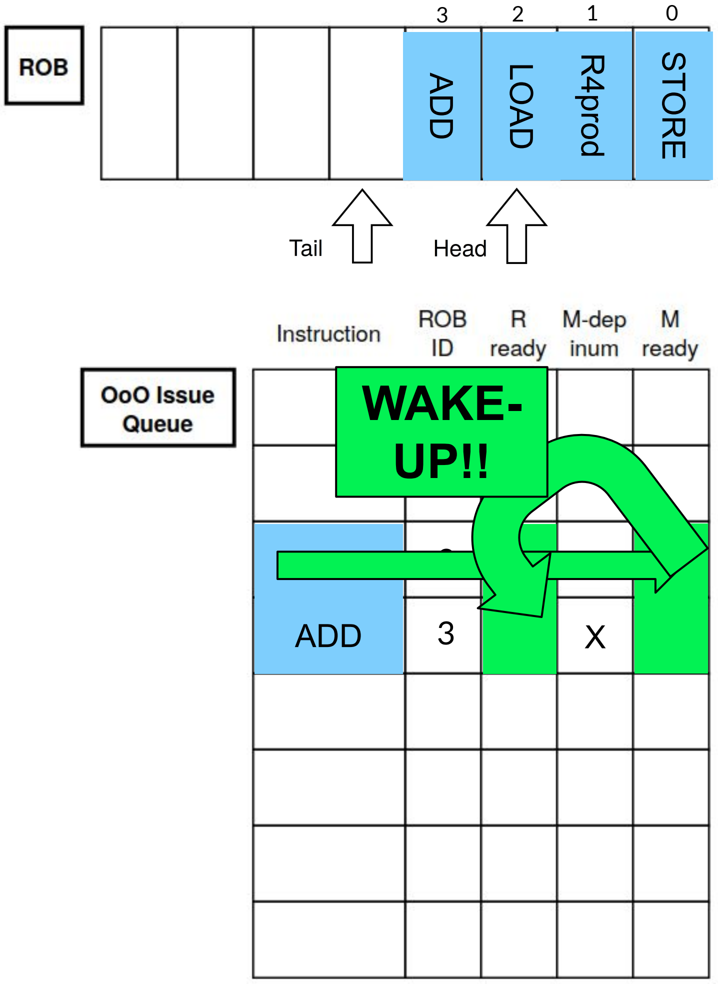
```
  STORE    R1  R2   IMM       # mem[ Addr A ] <-  R1
   (…)
  LOAD     R3  R4   IMM       # R3 <-  mem[ Addr A ]

➡ ADD      R5  R3   R3        # R5 <-  R3+R3
```



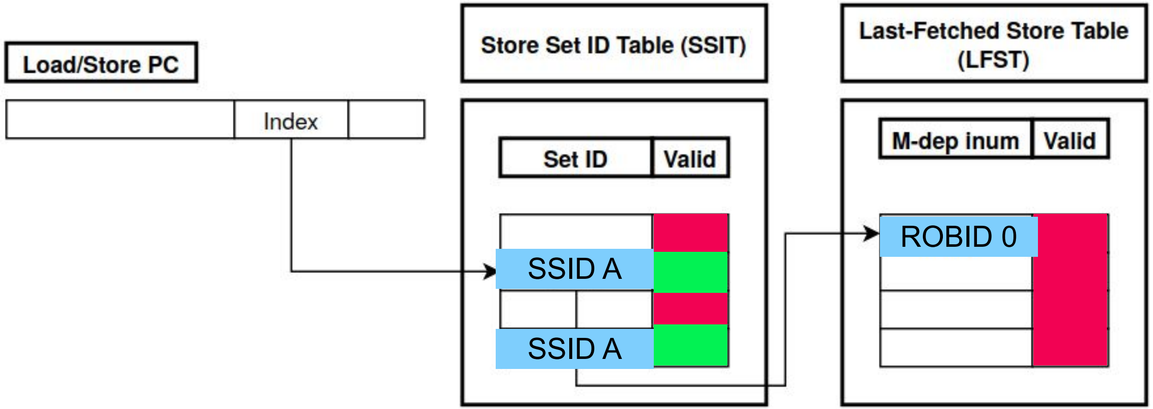


Instruction	ROB ID	R ready	M-dep inum	M ready
ADD	3	🟥	X	🟩

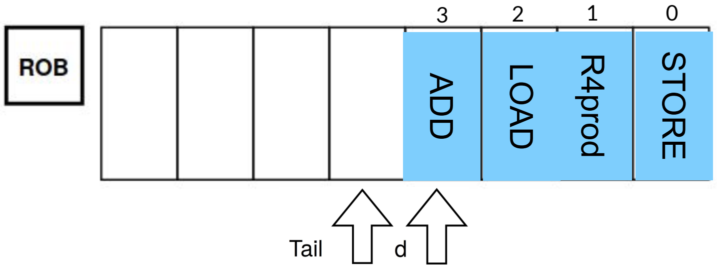
```
STORE   R1  R2  IMM     # mem[Addr A] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[Addr A]

ADD     R5  R3   R3     # R5 <- R3+R3
```



Instruction	ROB ID	R ready	M-dep inum	M ready
ADD	3	🟩	X	🟩

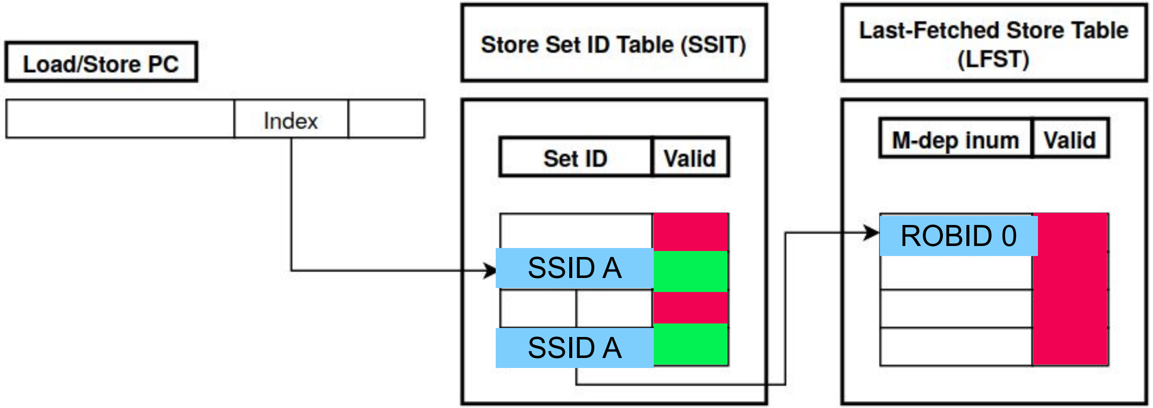


```
  STORE   R1  R2  IMM     # mem[Addr A] <- R1
   (…)
  LOAD    R3  R4  IMM     # R3 <- mem[Addr A]

➡ ADD     R5  R3  R3      # R5 <- R3+R3
```

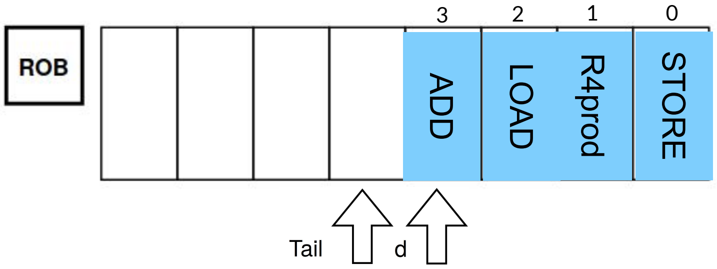


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
ADD	3	🟩	X	🟩

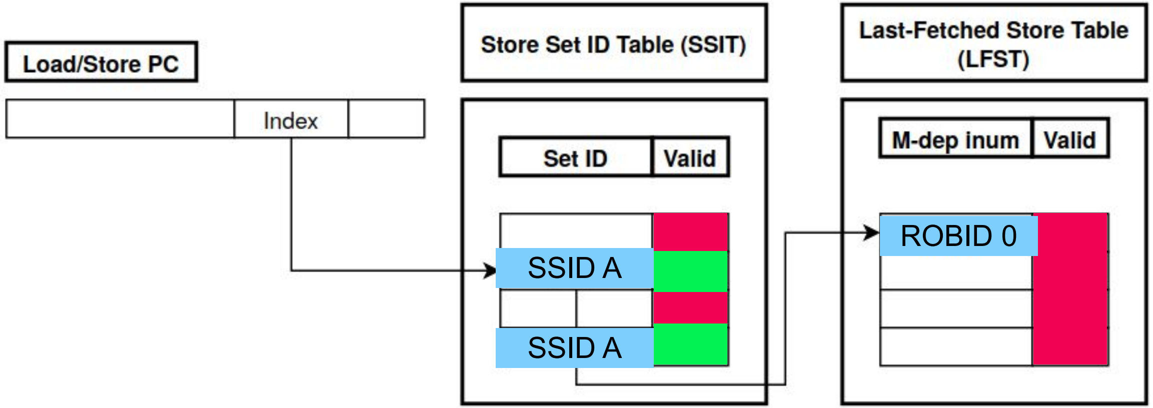


```
        STORE   R1  R2  IMM    # mem[Addr A] <- R1
         (…)
        LOAD    R3  R4  IMM    # R3 <- mem[Addr A]

➡       ADD     R5  R3   R3    # R5 <- R3+R3
```

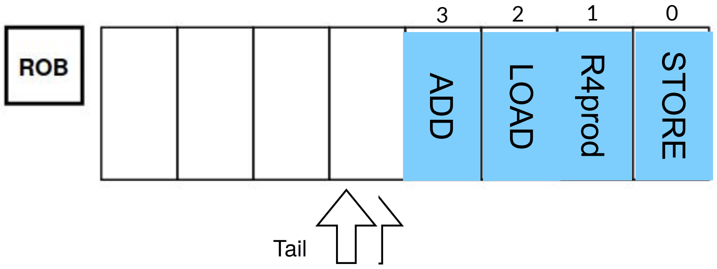


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
ADD	3	🟩	X	🟩

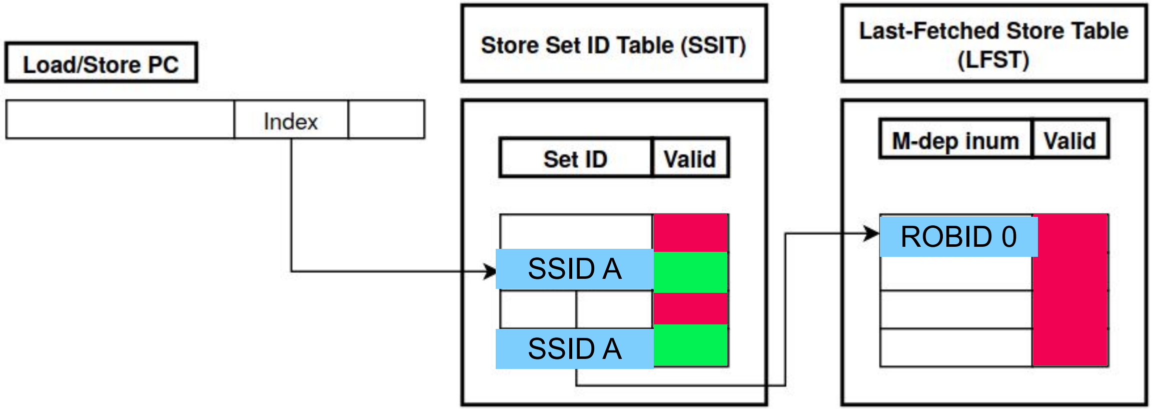


```
   STORE   R1  R2   IMM     # mem[ Addr A ] <-  R1
    (…)
   LOAD    R3  R4   IMM     # R3 <-  mem[ Addr A ]

➡  ADD     R5  R3   R3      # R5 <-  R3+R3
```



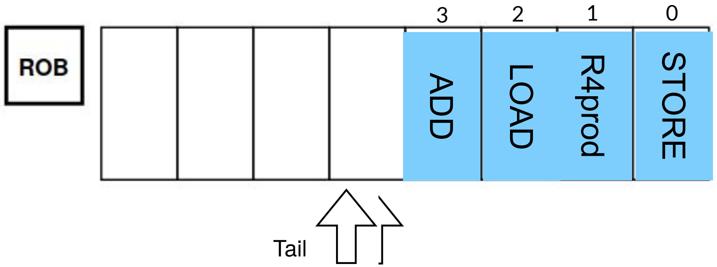
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready



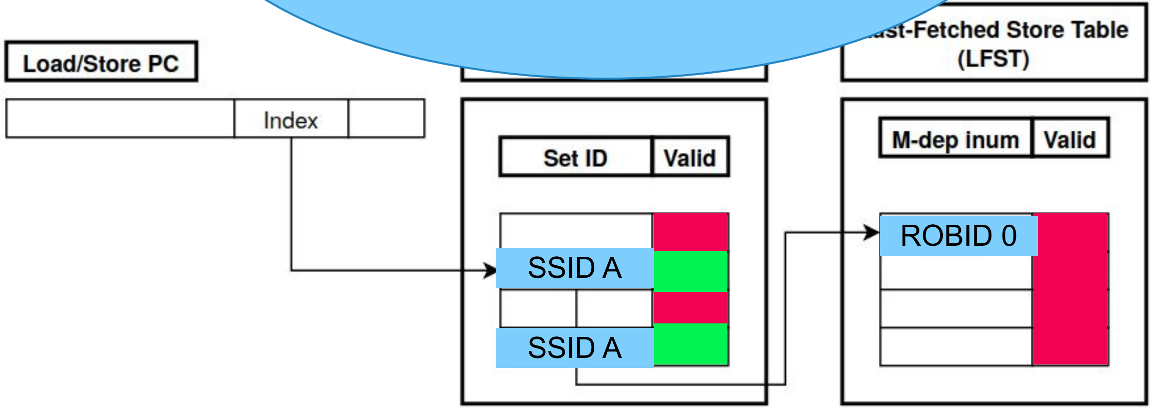
```
STORE   R1  R2  IMM     # mem[Addr A] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[Addr A]

ADD                                  R3
```

Alright! Now, for our last trick of the night…



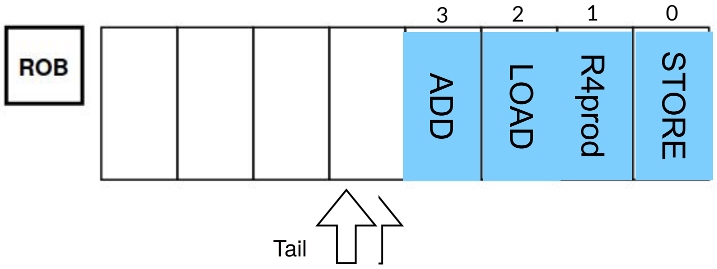
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready



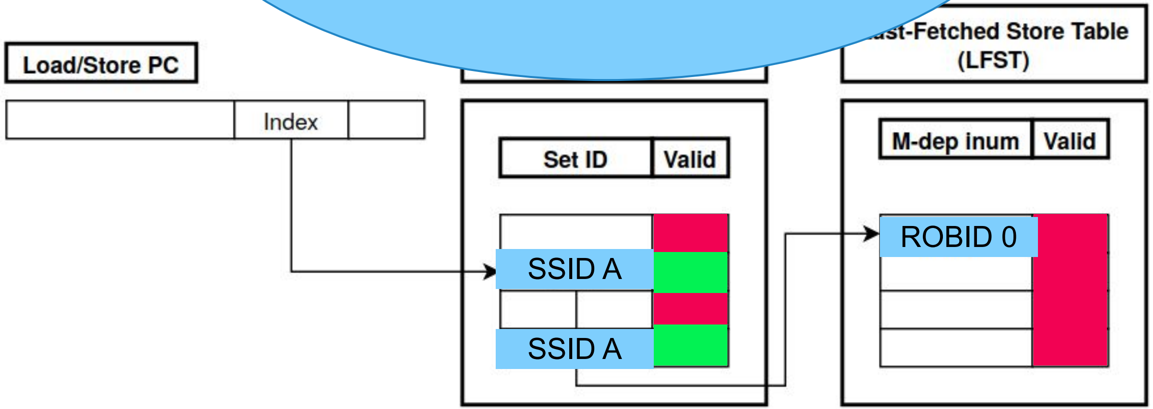
```
STORE   R1  R2   IMM     # mem[Addr A] <- R1
 (…)
LOAD    R3  R4   IMM     # R3 <- mem[Addr A]

ADD                          R3
```

…**WHAT IF** there was another, new STORE in between the LOAD-STORE pair??

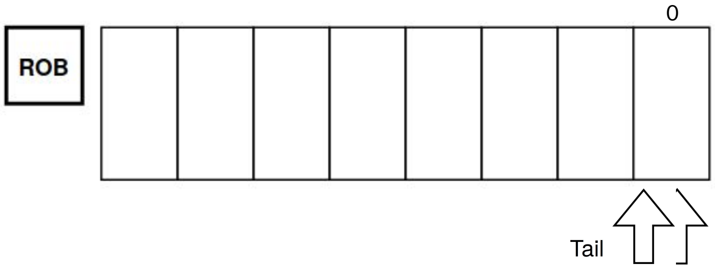


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready

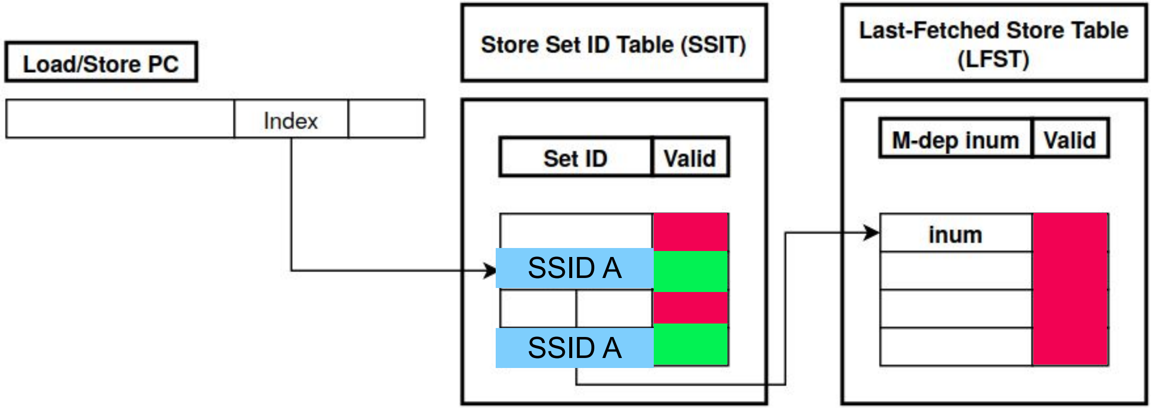


```
STORE   R1  R2  IMM     # mem[R2+IMM] <- R1
 (...)
 LOAD   R3  R4  IMM     # R3 <- mem[R4+IMM]

 ADD    R5  R3   R3     # R5 <- R3+R3
```

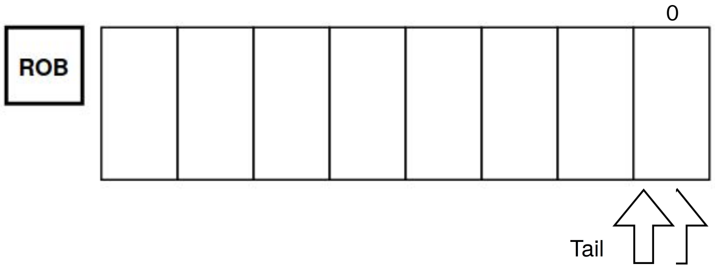


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready

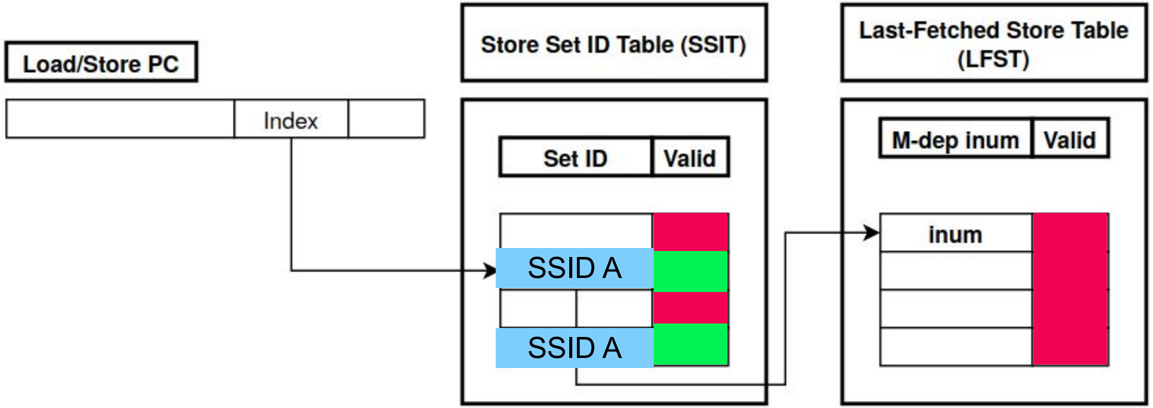


```
STORE   R1  R2  IMM     # mem[R2+IMM] <- R1
 (…)
 LOAD   R3  R4  IMM     # R3 <- mem[R4+IMM]

 ADD    R5  R3   R3     # R5 <- R3+R3
```

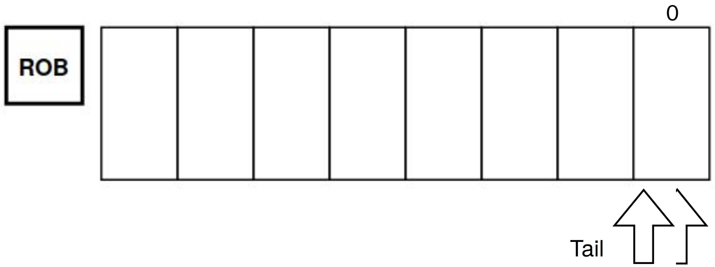


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready

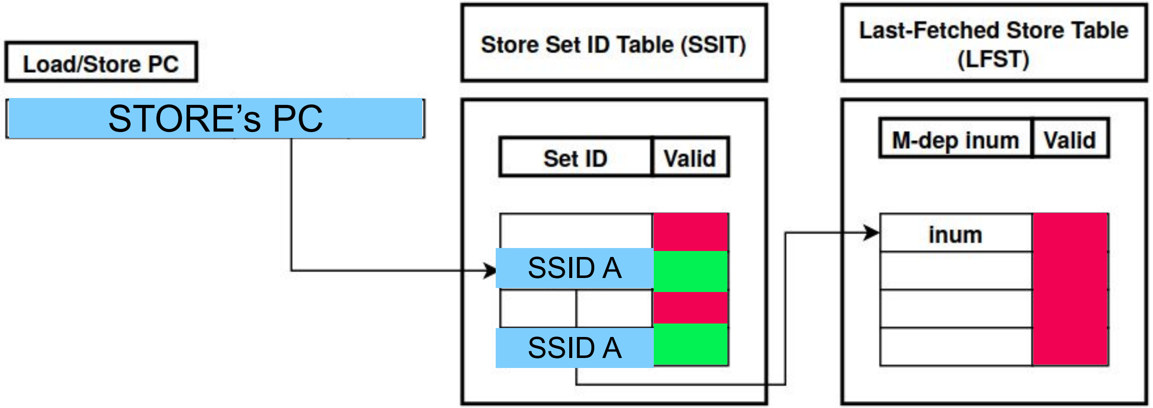


```
STORE   R1  R2   IMM      # mem[R2+IMM] <- R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3   R3       # R5 <- R3+R3
```

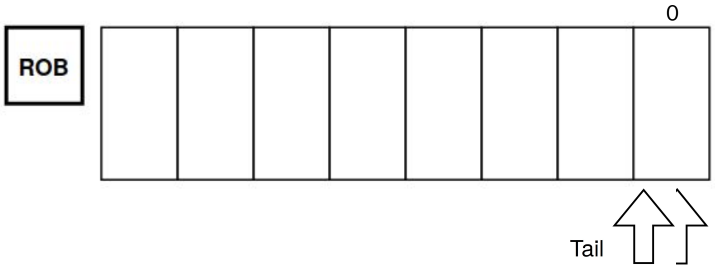


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready

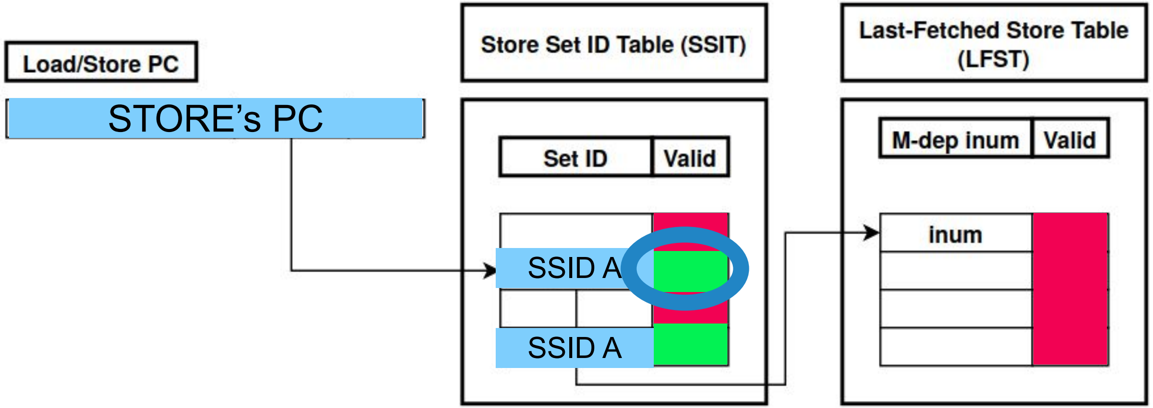


```
STORE   R1  R2  IMM     # mem[R2+IMM] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3  R3      # R5 <- R3+R3
```

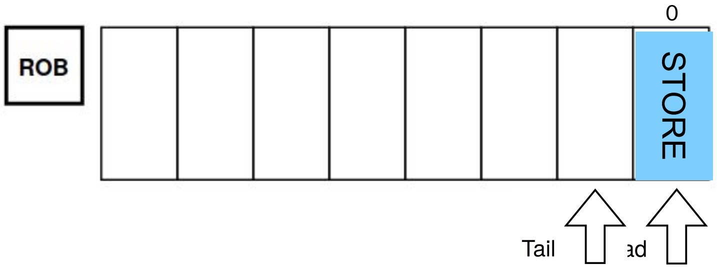


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready

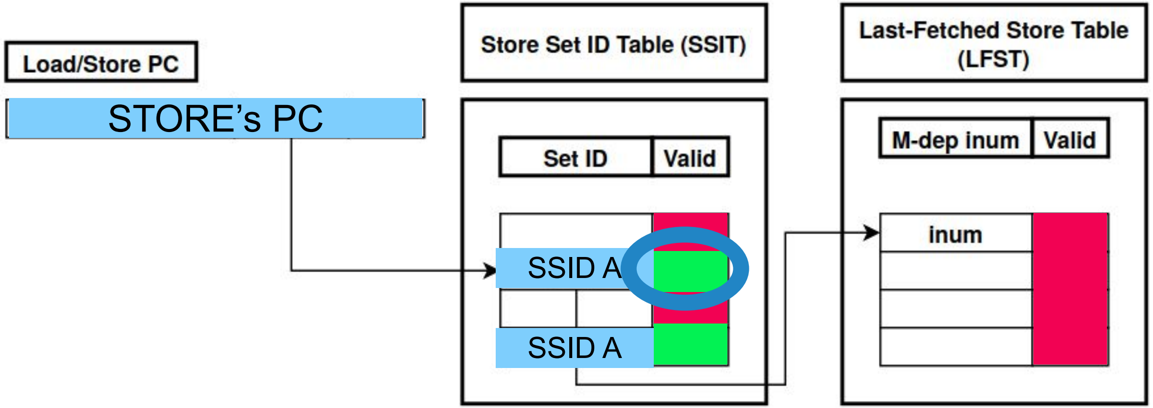


```
STORE   R1  R2  IMM      # mem[R2+IMM] <- R1
 (…)
LOAD    R3  R4  IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3  R3       # R5 <- R3+R3
```

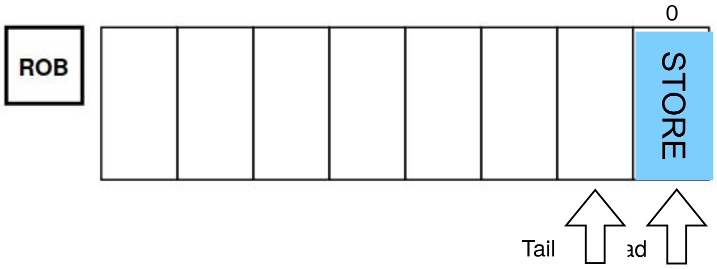


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩

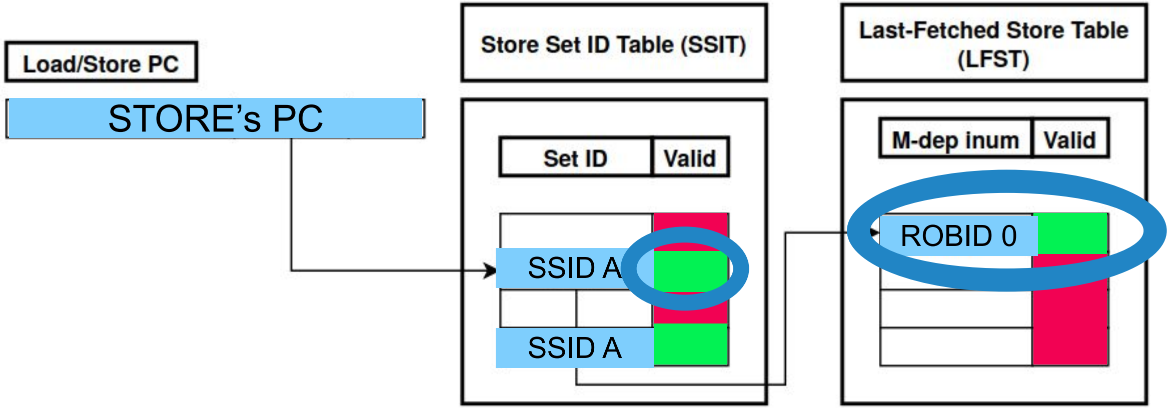


```
STORE   R1  R2   IMM      # mem[R2+IMM] <- R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3   R3       # R5 <- R3+R3
```

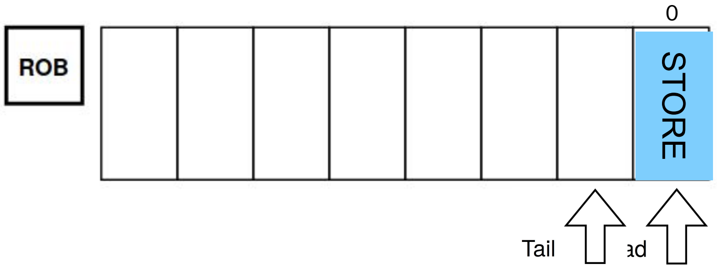


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩

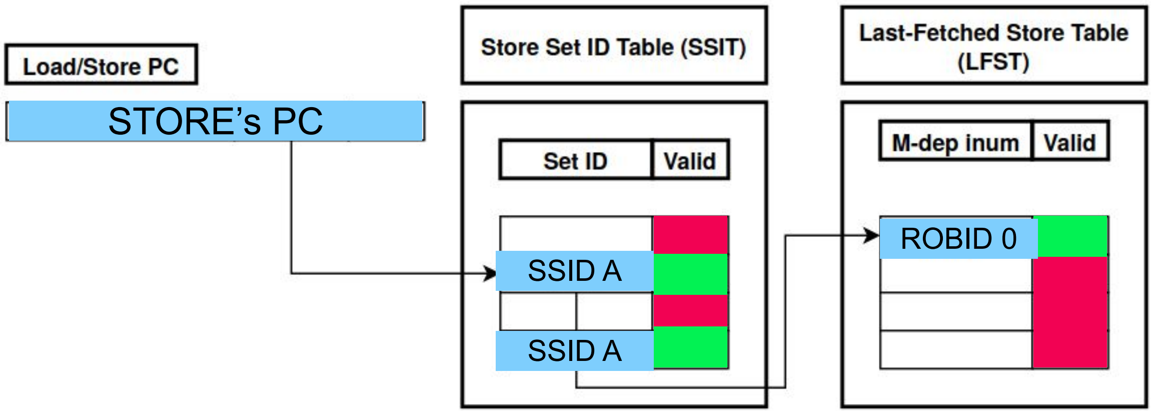


```
STORE   R1  R2  IMM     # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3  R3      # R5 <- R3+R3
```



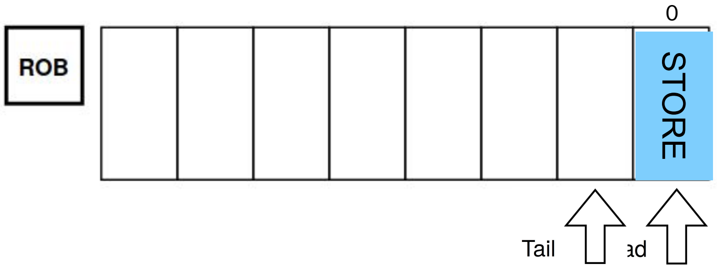
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩



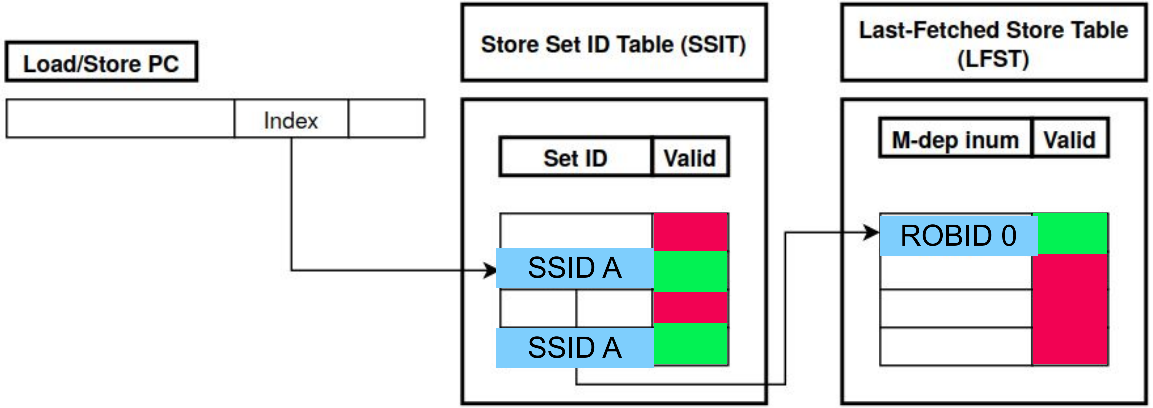
```
STORE   R1  R2  IMM     # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3  R3      # R5 <- R3+R3
```

R4-producer instr.

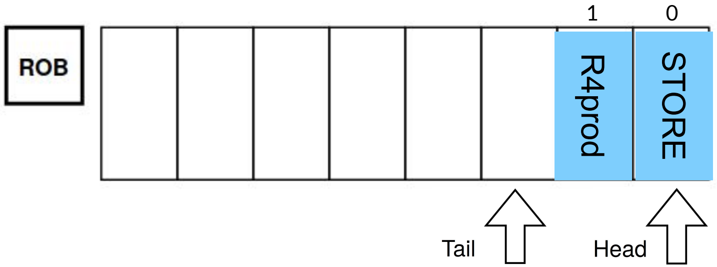


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩



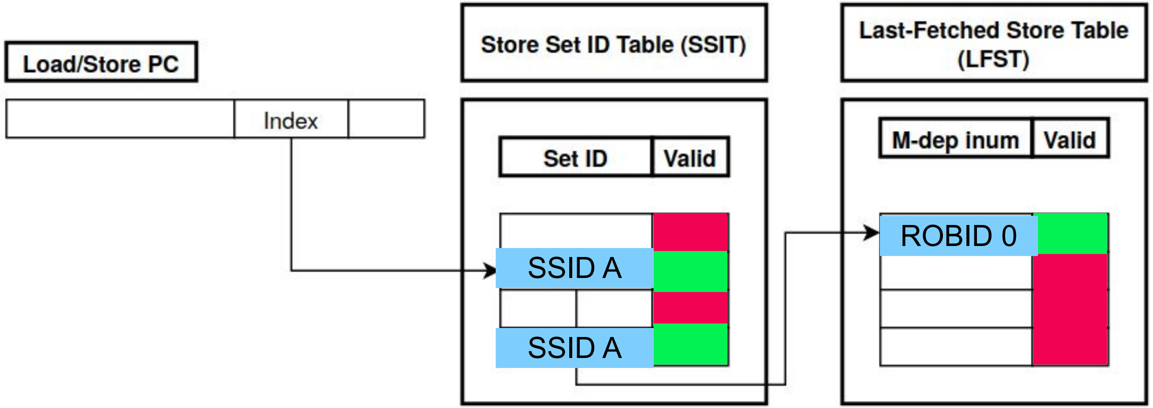
```
STORE   R1  R2  IMM     # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4  IMM     # R3 <- mem[R4+IMM]

ADD     R5  R3  R3      # R5 <- R3+R3
```



R4-producer instr.

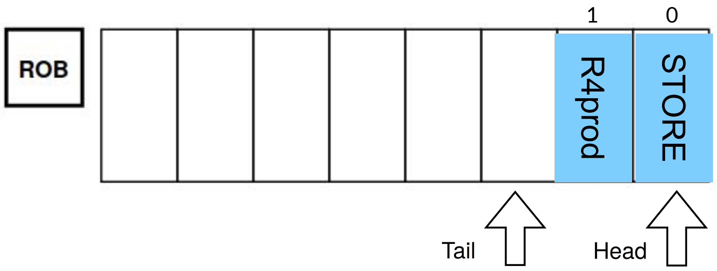
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩



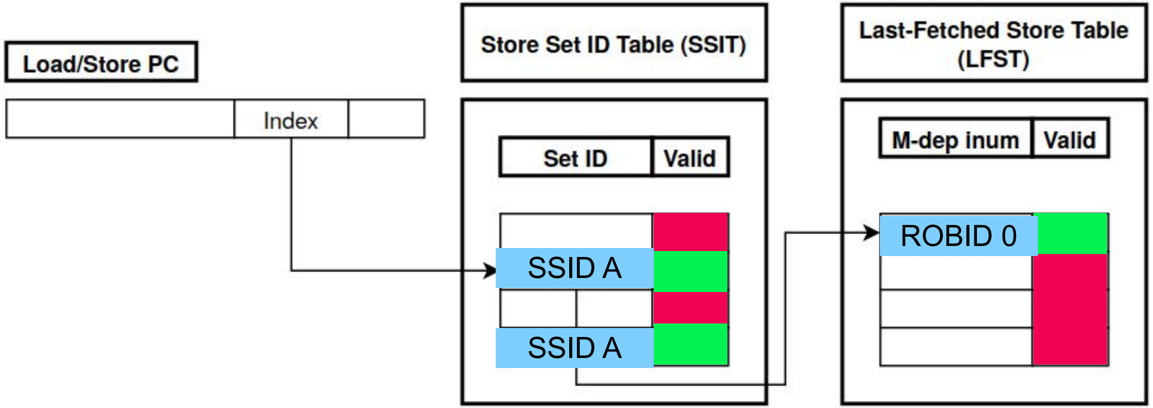
```
STORE   R1  R2   IMM      # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3   R3       # R5 <- R3+R3
```

NEW STORE!!

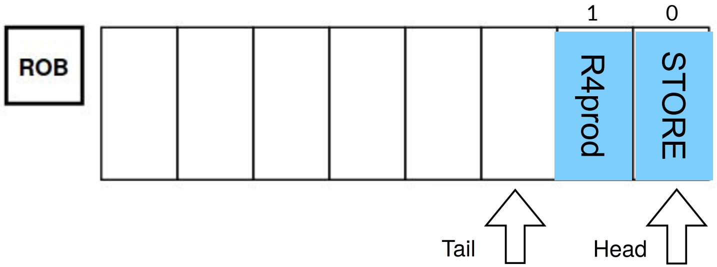


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩

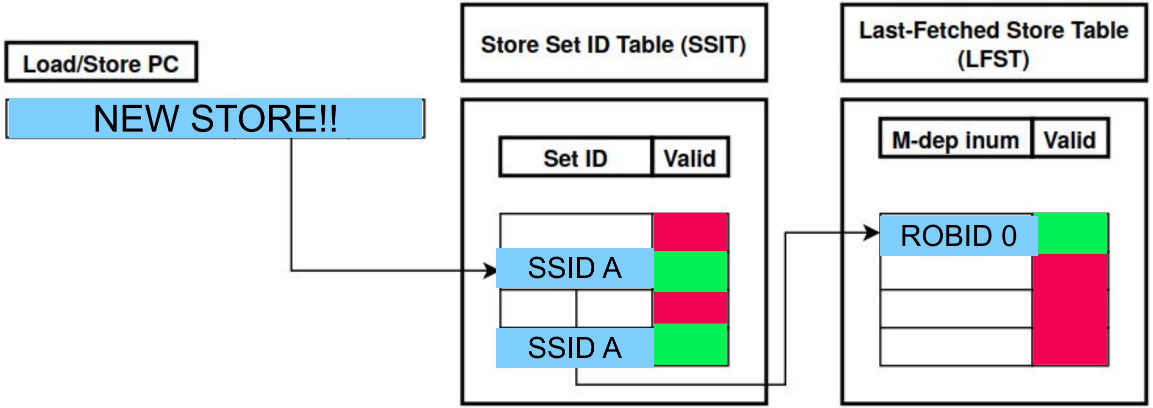


```
STORE   R1  R2  IMM      # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4  IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3  R3       # R5 <- R3+R3
```

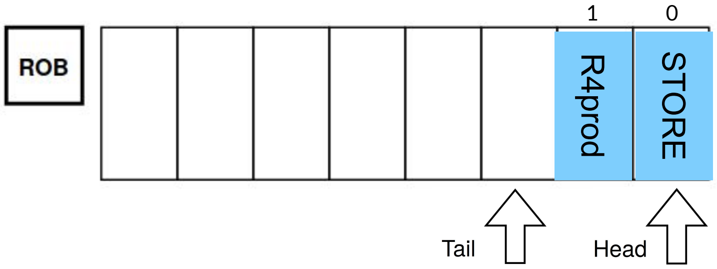


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩

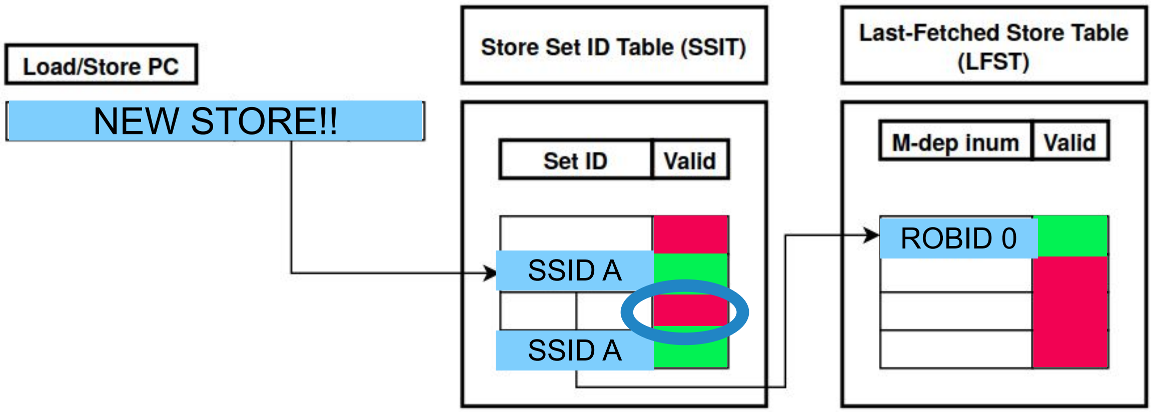


```
STORE   R1  R2  IMM      # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4  IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3   R3      # R5 <- R3+R3
```

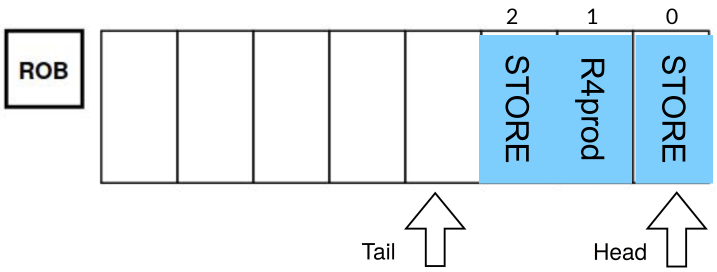


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩

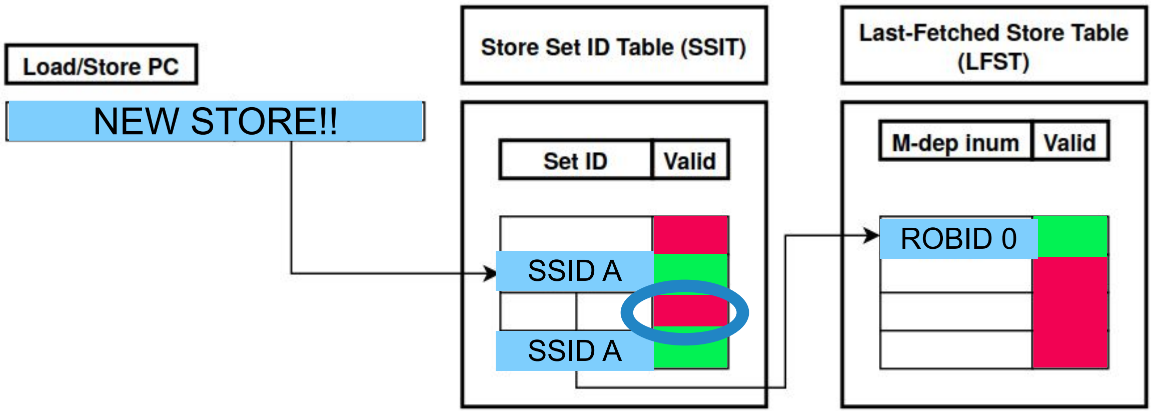


```
STORE   R1  R2  IMM      # mem[Addr ???] <- R1
 (…)
LOAD    R3  R4  IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3  R3       # R5 <- R3+R3
```

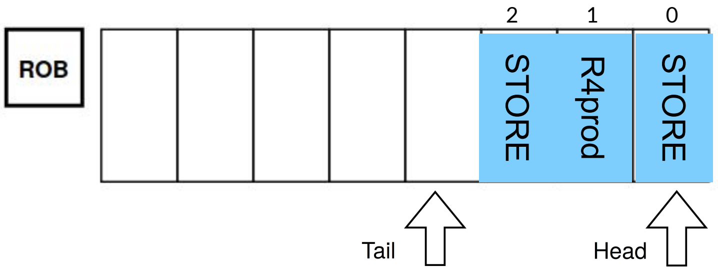


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
STORE	2	🟥	X	🟩

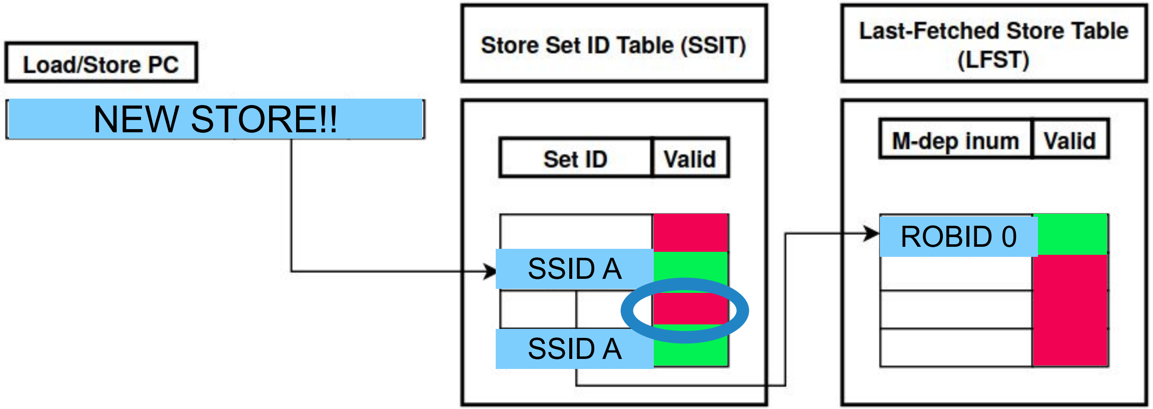


```
STORE    R1  R2   IMM       # mem[Addr ???] <- R1
(...)  STORE   R1  R8  IMM      # mem[Addr ???] <- R1
LOAD     R3  R4   IMM       # R3 <- mem[R4+IMM]

ADD      R5  R3   R3        # R5 <- R3+R3
```

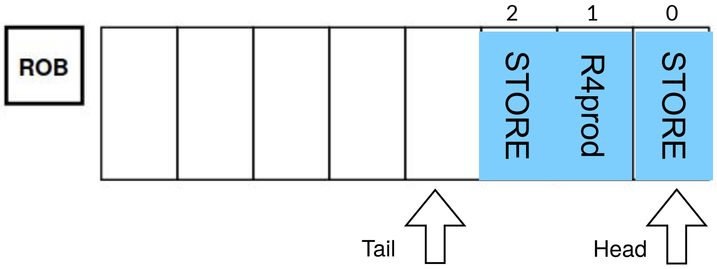


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
STORE	2	🟥	X	🟩

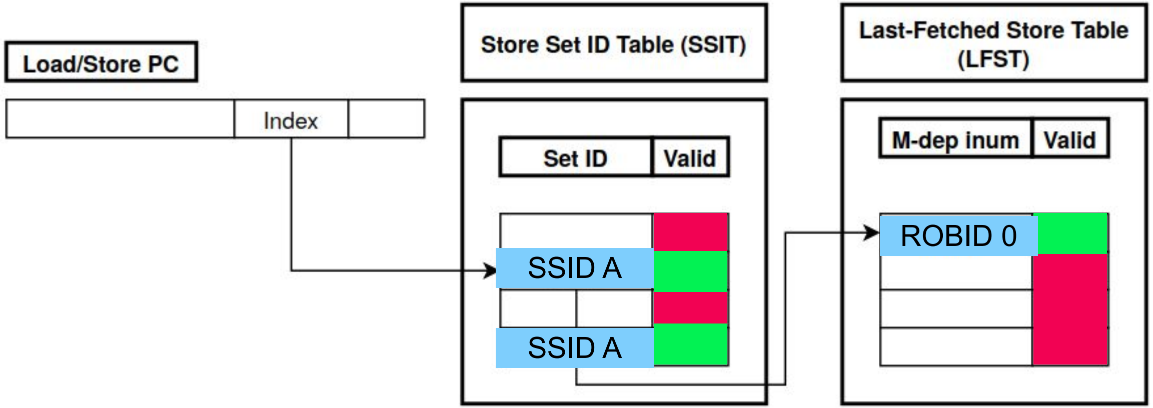


```
      STORE   R1  R2   IMM      # mem[ Addr ??? ] <- R1
      (…)  STORE   R1   R8   IMM      # mem[ Addr ??? ] <- R1
➡     LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

      ADD     R5  R3   R3       # R5 <- R3+R3
```

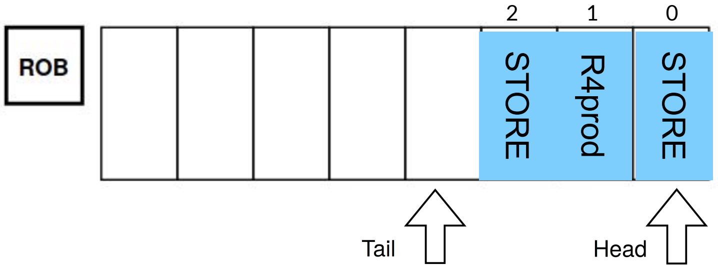


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
STORE	2	🟥	X	🟩

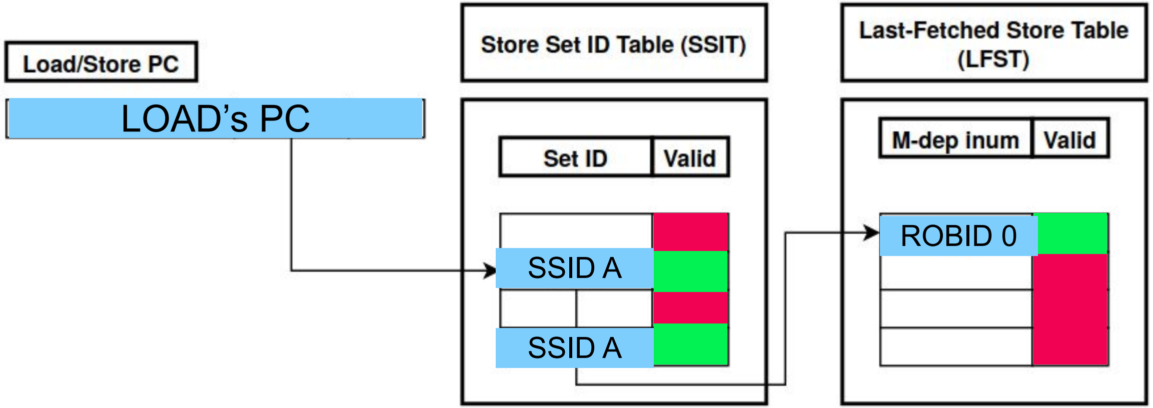


```
   STORE   R1  R2   IMM      # mem[ Addr ??? ] <-  R1
   (…)  STORE   R1  R8  IMM     # mem[ Addr ??? ] <-  R1
➡  LOAD    R3  R4   IMM      # R3 <-  mem[R4+IMM]

   ADD     R5  R3   R3       # R5 <-  R3+R3
```

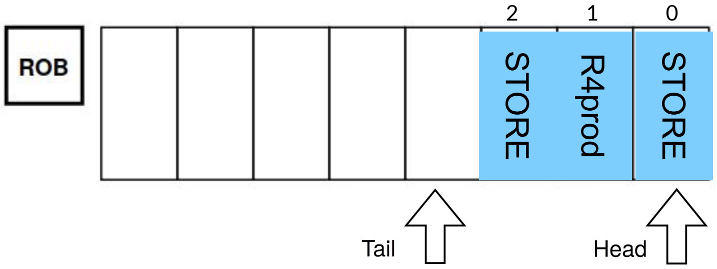


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
STORE	2	🟥	X	🟩

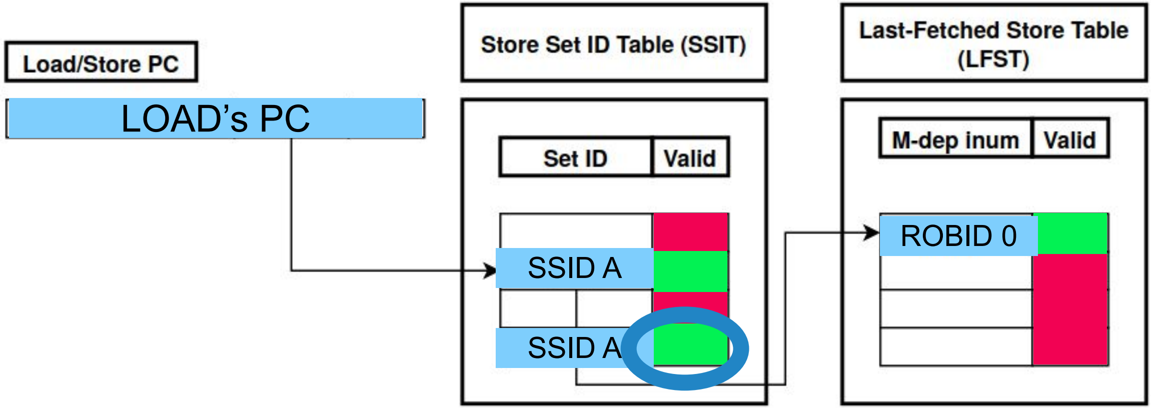


```
      STORE   R1  R2   IMM      # mem[ Addr ??? ] <- R1
(…)   STORE   R1  R8   IMM      # mem[ Addr ??? ] <- R1
      LOAD    R3  R4   IMM      # R3 <- mem[R4+IMM]

      ADD     R5  R3   R3       # R5 <- R3+R3
```

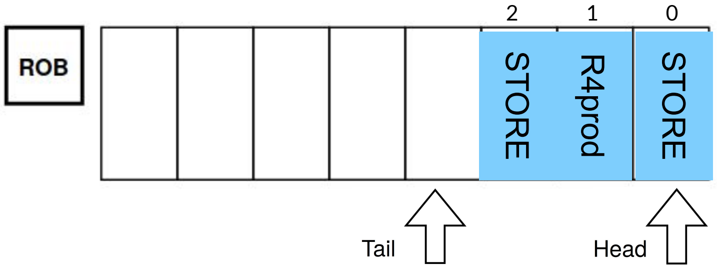


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
STORE	2	🟥	X	🟩

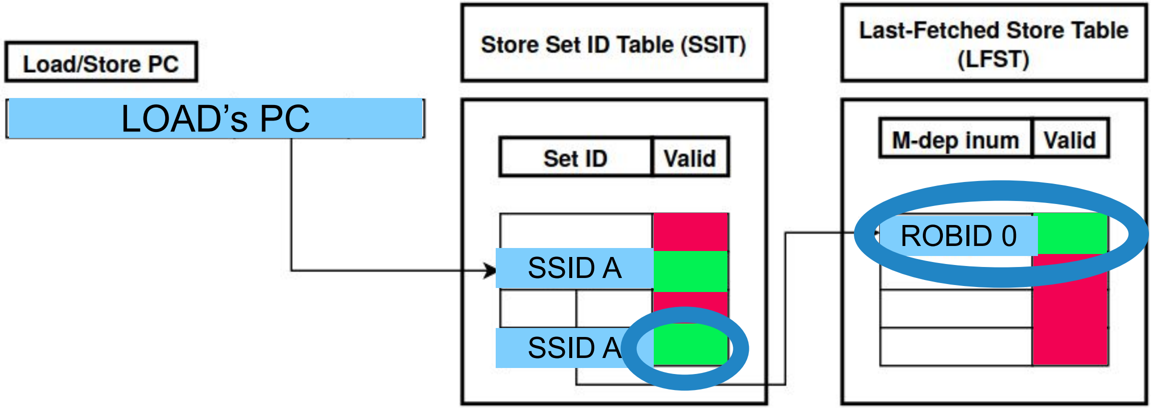


```
STORE    R1  R2   IMM       # mem[Addr ???] <- R1
 (…)  STORE   R1  R8  IMM      # mem[Addr ???] <- R1
LOAD     R3  R4   IMM       # R3 <- mem[R4+IMM]

ADD      R5  R3   R3        # R5 <- R3+R3
```



OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
STORE	2	🟥	X	🟩



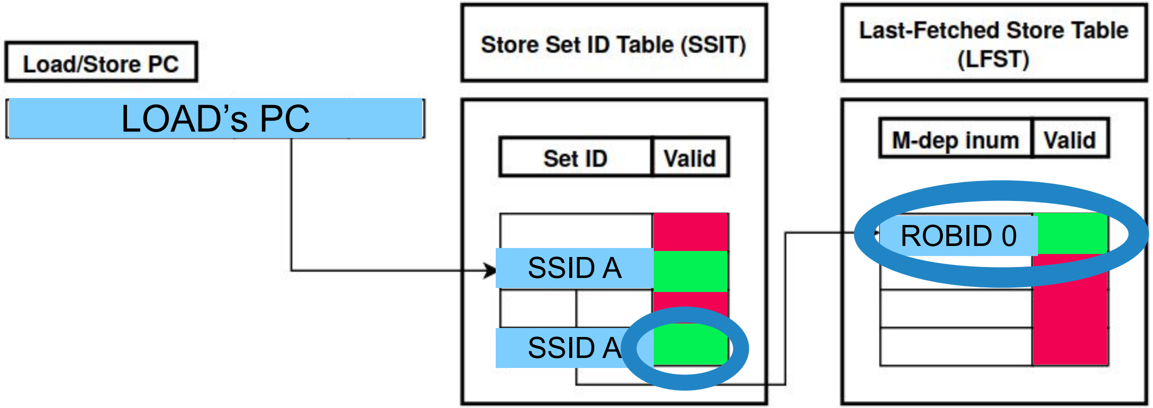
```
STORE   R1  R2  IMM      # mem[ Addr ??? ] <-  R1
 (…)  STORE   R1  R8  IMM      # mem[ Addr ??? ] <-  R1
LOAD    R3  R4  IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3  R3       # R5 <- R3+R3
```

ROB

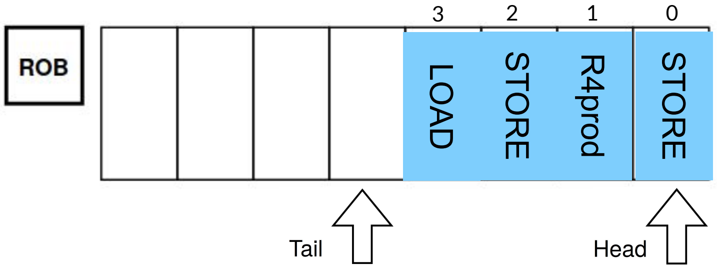
				3	2	1	0
				LOAD	STORE	R4prod	STORE
			Tail				Head

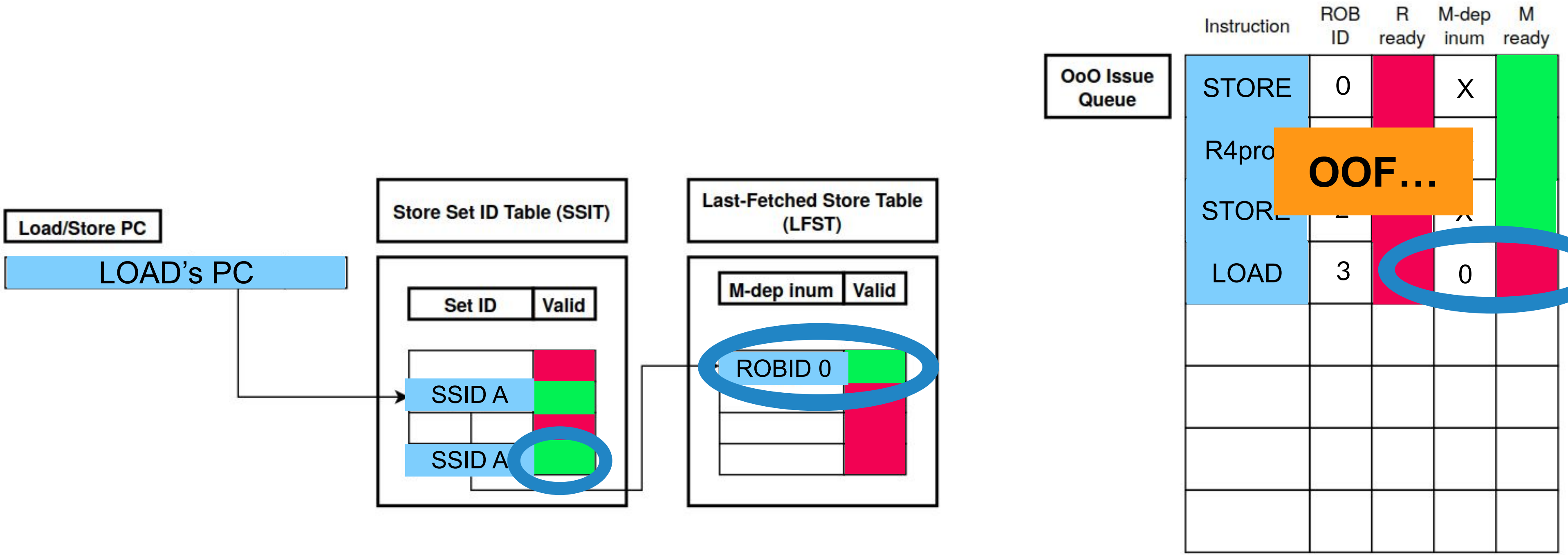
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
STORE	2	🟥	X	🟩
LOAD	3	🟥	0	🟥



```
STORE   R1  R2  IMM      # mem[Addr ???] <- R1
(…)  STORE  R1  R8  IMM      # mem[Addr ???] <- R1
LOAD    R3  R4  IMM      # R3 <- mem[R4+IMM]

ADD     R5  R3  R3       # R5 <- R3+R3
```





Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

M-dep inum	Valid
ROBID 0	🟩
	🟥
	🟥
	🟥

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4pro				🟩
STORE	2	🟥	X	🟩
LOAD	3	🟥	0	🟥

```
STORE   R1  R2   IMM      # mem[ Addr ??? ]  <-  R1
 (…)  STORE   R1  R8  IMM      # mem[ Addr ??? ]   R1
LOAD    R3  R4   IMM      # R3 <-  mem[ Addr ??? ]

 ADD    R5  R3   R3       # R5 <-  R3+R3
```

ROB

				3	2	1	0
				LOAD	STORE	R4prod	STORE

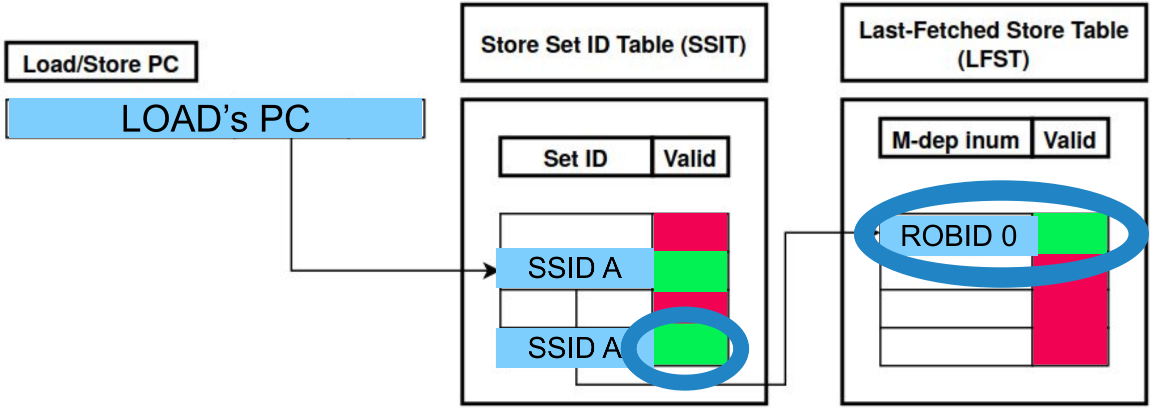
Tail

Head

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4pro				🟩
STORE	2	🟥	X	🟩
LOAD	3	🟥	0	🟥

OoO Issue Queue

OOF…



Load/Store PC

LOAD's PC

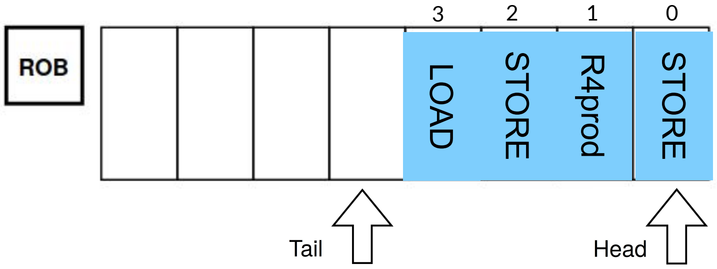
Store Set ID Table (SSIT)

Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

Last-Fetched Store Table (LFST)

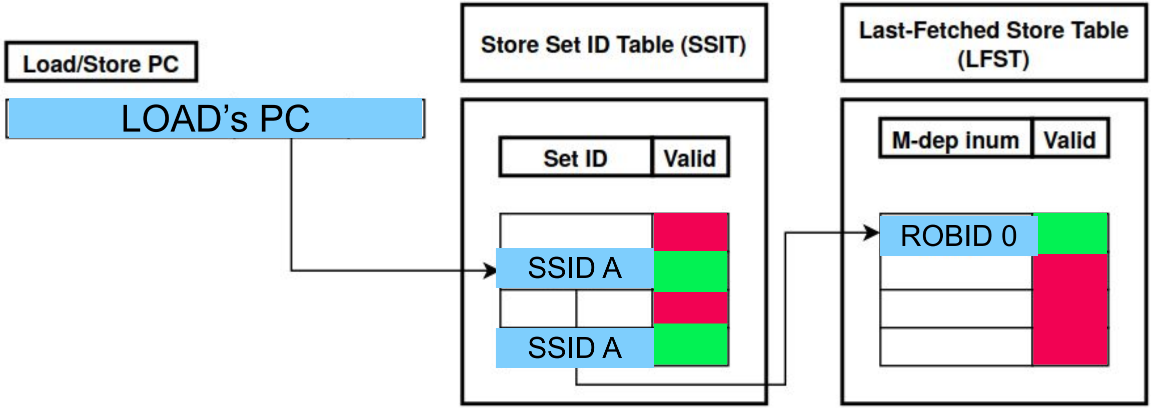
M-dep inum	Valid
ROBID 0	🟩
	🟥
	🟥
	🟥

```
STORE   R1  R2   IMM      # mem[ Addr ??? ] <-  R1
(…)  STORE   R1   R8  IMM      # mem[ Addr ??? ] <-  R1
LOAD    R3  R4   IMM      # R3 <-  mem[ Addr ??? ]

ADD     R5  R3   R3       # R5 <-  R3+R3
```

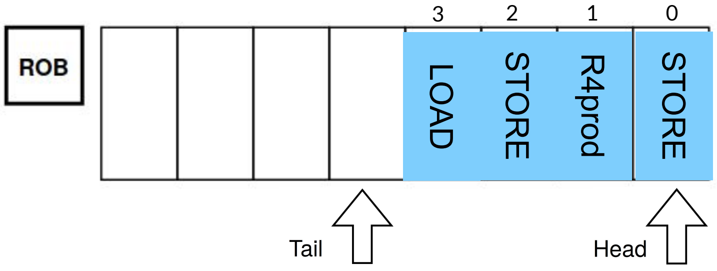


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
STORE	2	🟥	X	🟩
LOAD	3	🟥	0	🟥



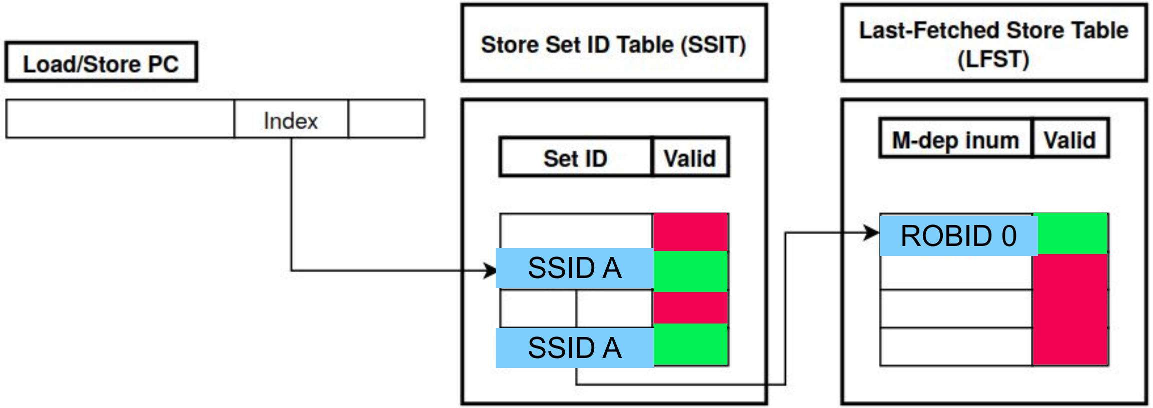
```
STORE   R1  R2   IMM      # mem[Addr ???] <- R1
(…)  STORE   R1  R8  IMM     # mem[Addr ???] <- R1
LOAD    R3  R4   IMM      # R3 <- mem[Addr ???]

ADD     R5  R3   R3       # R5 <- R3+R3
```



ADD

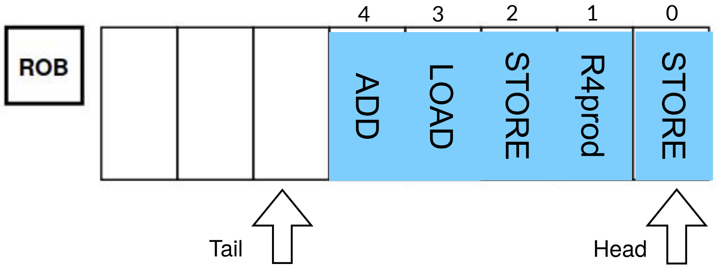
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
STORE	2	🟥	X	🟩
LOAD	3	🟥	0	🟥



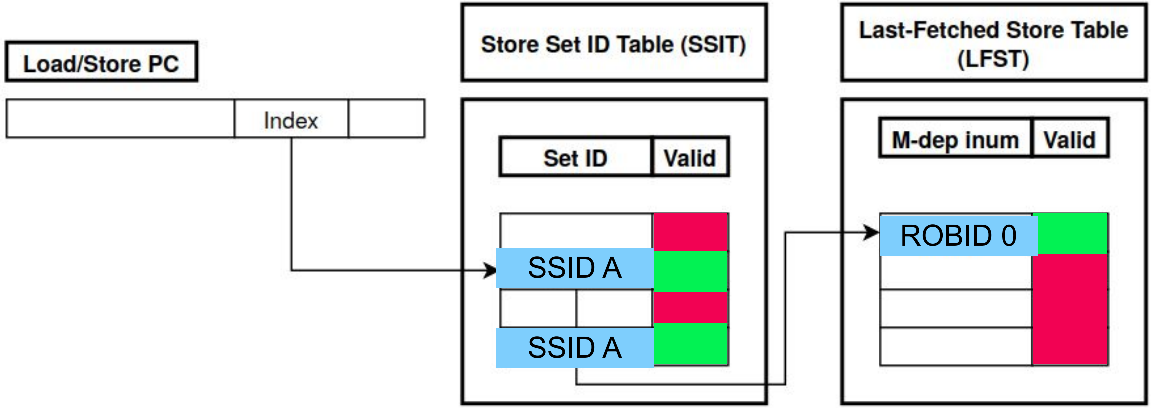
```
STORE   R1  R2  IMM      # mem[Addr ???] <-  R1
(...)  STORE   R1  R8  IMM      # mem[Addr ???] <-  R1
LOAD    R3  R4  IMM      # R3 <-  mem[Addr ???]

ADD     R5  R3  R3       # R5 <-  R3+R3
```

ADD

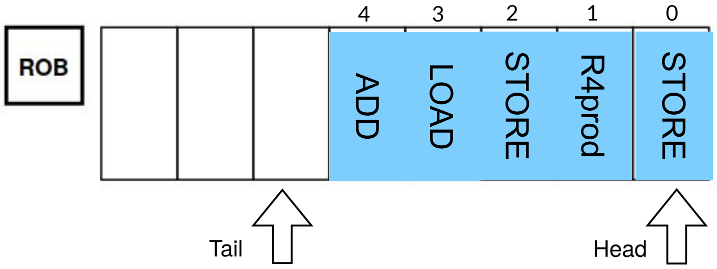


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
STORE	2	🟥	X	🟩
LOAD	3	🟥	0	🟥
ADD	4	🟥	X	🟩

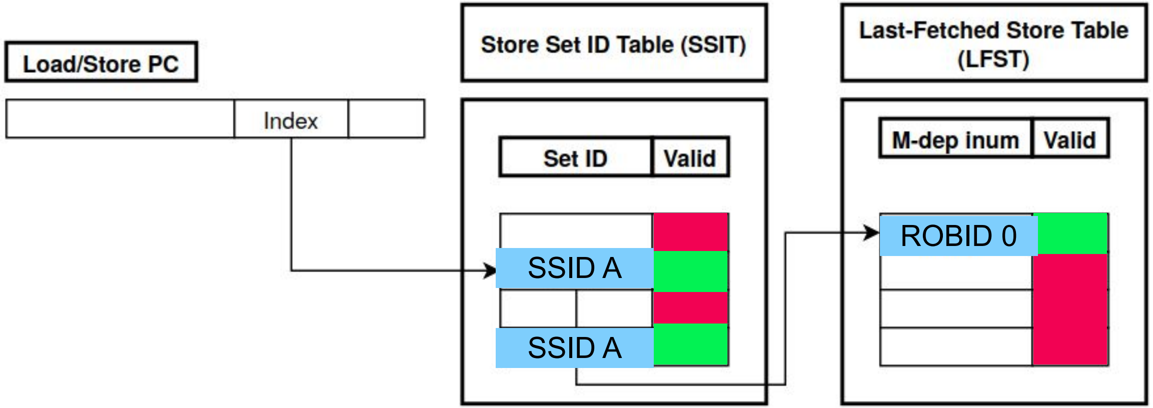


```
   STORE   R1  R2  IMM     # mem[Addr ???] <- R1
   (…)  STORE   R1  R8  IMM     # mem[Addr ???] <- R1
   LOAD    R3  R4  IMM     # R3 <- mem[Addr ???]

➡  ADD     R5  R3  R3      # R5 <- R3+R3
```

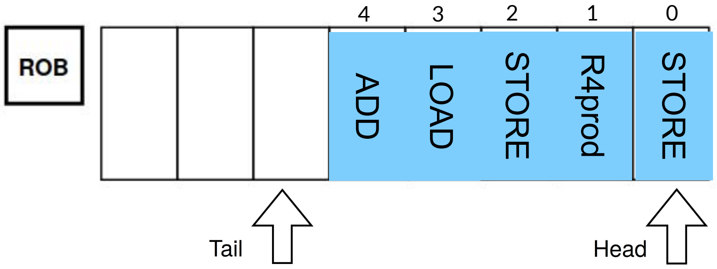


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
R4prod	1	🟥	X	🟩
STORE	2	🟥	X	🟩
LOAD	3	🟥	0	🟥
ADD	4	🟥	X	🟩

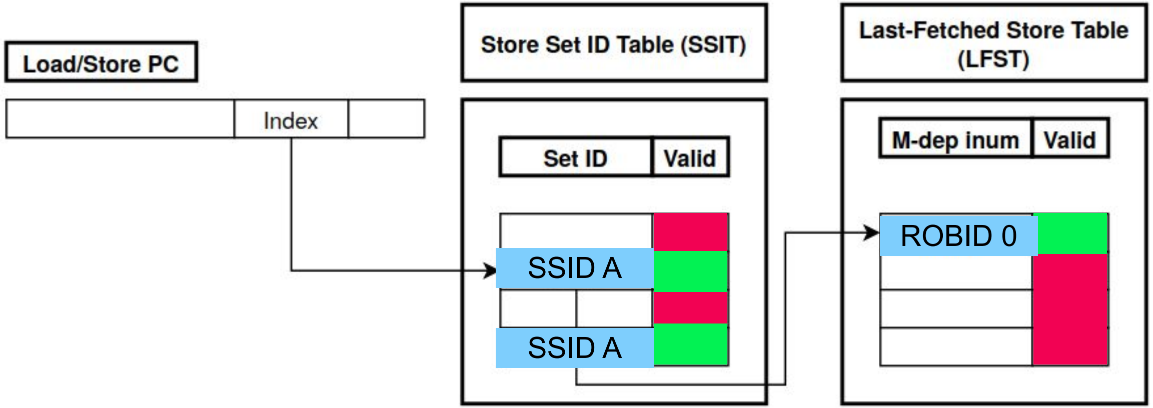


```
STORE    R1  R2   IMM       # mem[ Addr ??? ] <-  R1
(…)  STORE   R1  R8   IMM      # mem[ Addr ??? ] <-  R1
LOAD     R3  R4   IMM       # R3 <-  mem[ Addr ??? ]

ADD      R5  R3   R3        # R5 <-  R3+R3
```



OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X!!	🟩
R4prod	1	🟩	X	🟩
STORE	2	🟥	X	🟩
LOAD	3	🟥	0	🟥
ADD	4	🟥	X	🟩



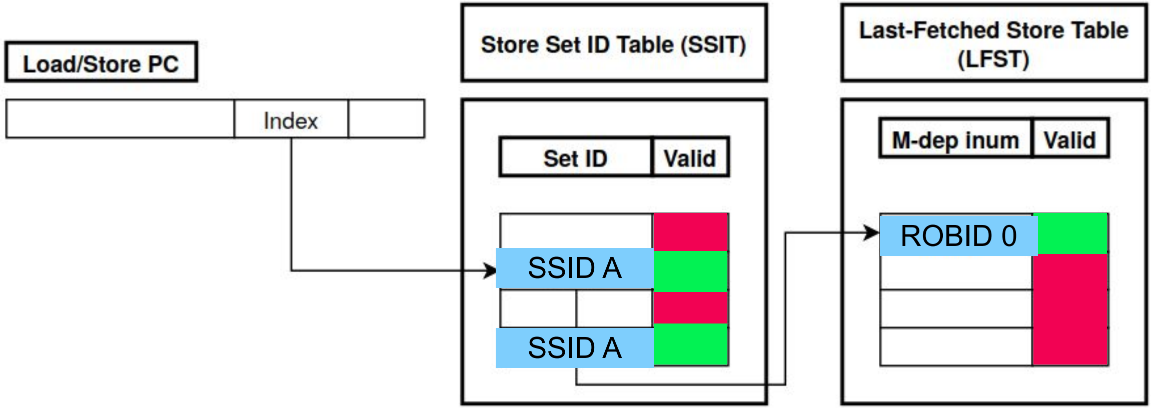
```
        STORE   R1  R2  IMM      # mem[Addr ???] <- R1
        (...)  STORE  R1  R8  IMM      # mem[Addr ???] <- R1
        LOAD    R3  R4  IMM      # R3 <- mem[Addr ???]

➡      ADD     R5  R3  R3       # R5 <- R3+R3
```

ROB

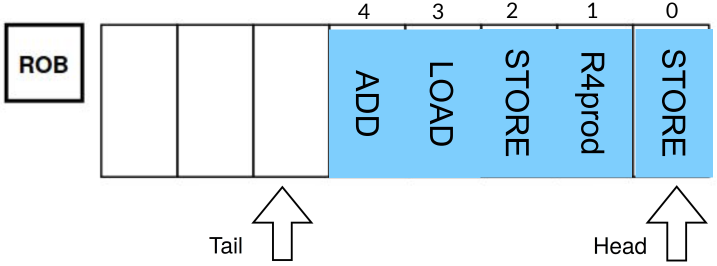
			4	3	2	1	0
			ADD	LOAD	STORE	R4prod	STORE
		Tail					Head

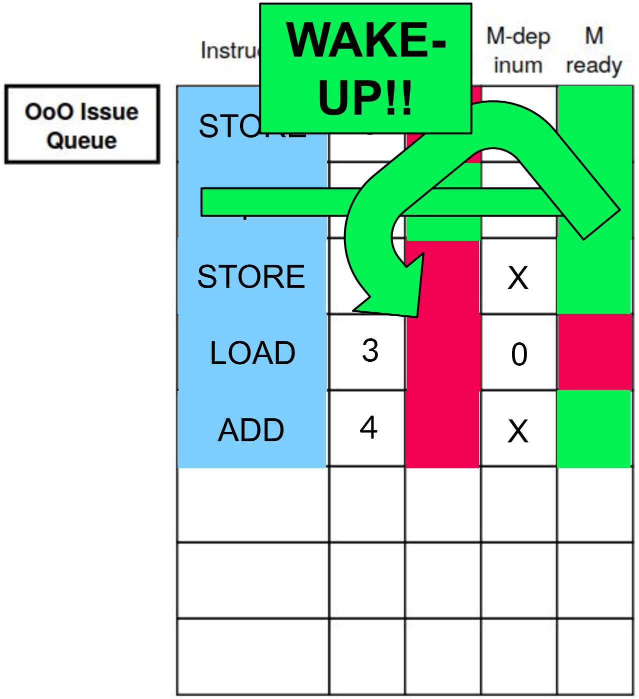
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
		🟩		🟩
STORE	2	🟥	X	🟩
LOAD	3	🟥	0	🟥
ADD	4	🟥	X	🟩

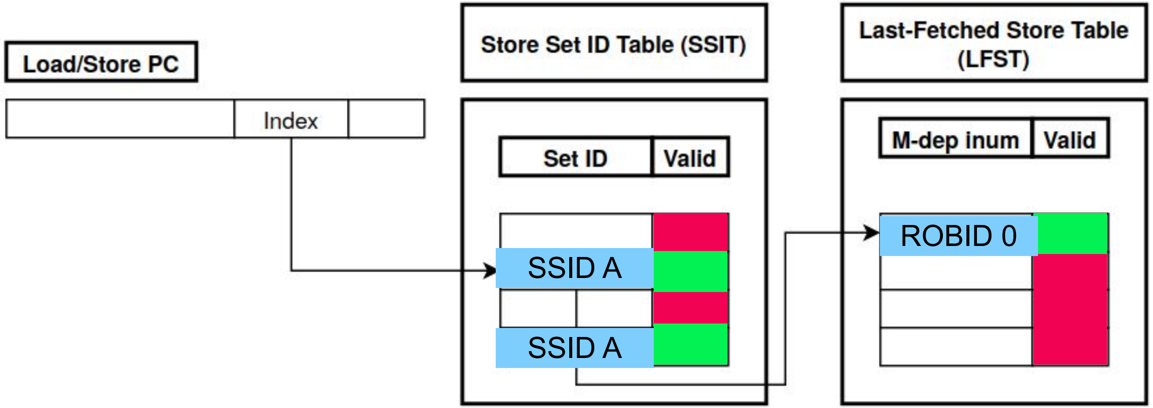


```
STORE   R1  R2   IMM      # mem[ Addr ??? ] <-  R1
(…)  STORE   R1   R8   IMM      # mem[ Addr ??? ] <-  R1
LOAD    R3  R4   IMM      # R3 <-  mem[ Addr ??? ]

ADD     R5  R3   R3       # R5 <-  R3+R3
```

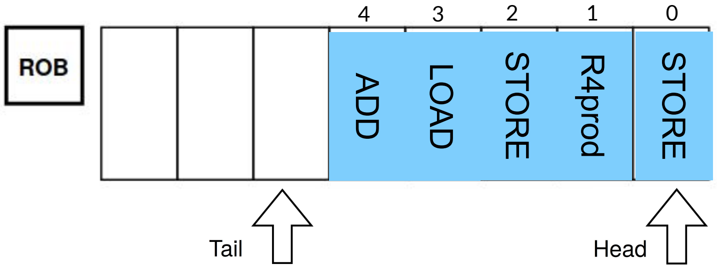


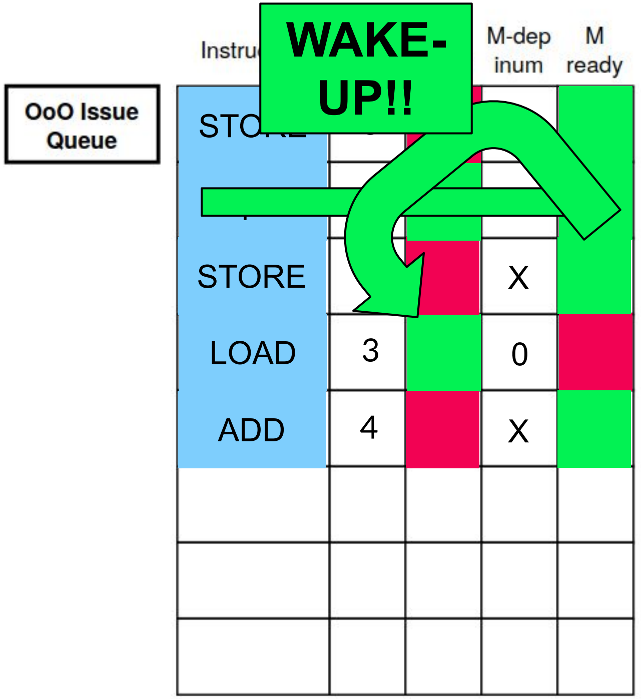


Instru			M-dep inum	M ready
STO		🟥		🟩
		🟩		🟩
STORE		🟥	X	🟩
LOAD	3	🟥	0	🟥
ADD	4	🟥	X	🟩

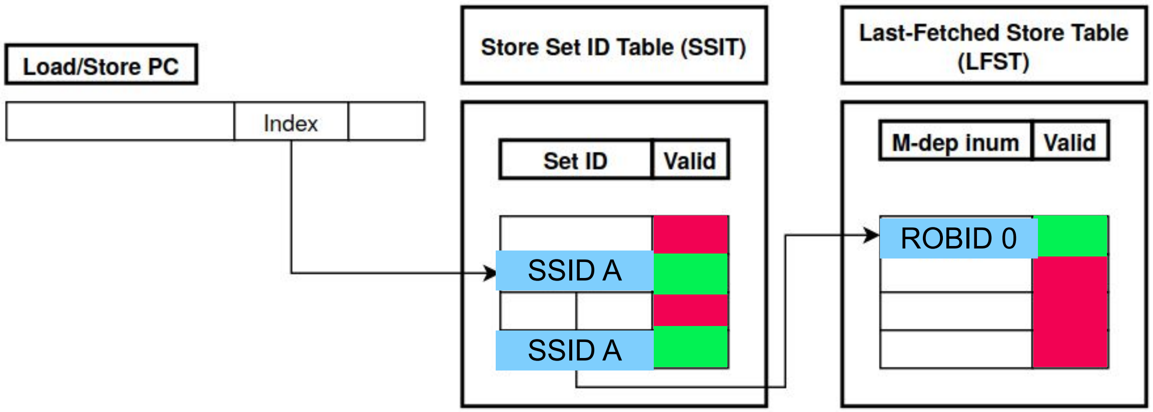


```
STORE    R1  R2   IMM      # mem[Addr ???] <-  R1
(…)  STORE   R1   R8   IMM     # mem[Addr ???] <-  R1
LOAD     R3  R4   IMM      # R3 <-  mem[Addr A]

ADD      R5  R3   R3       # R5 <-  R3+R3
```





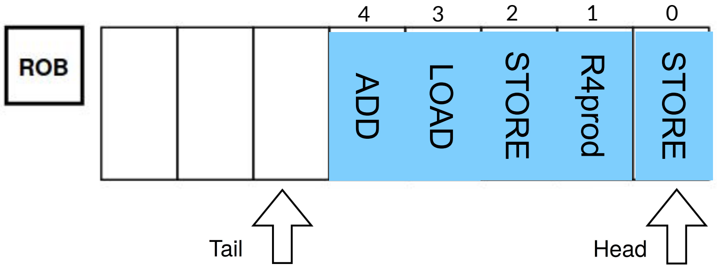
Instru			M-dep inum	M ready
STORE		🟥		🟩
		🟩		🟩
STORE		🟥	X	🟩
LOAD	3	🟩	0	🟥
ADD	4	🟥	X	🟩



Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

M-dep inum	Valid
ROBID 0	🟩
	🟥
	🟥
	🟥

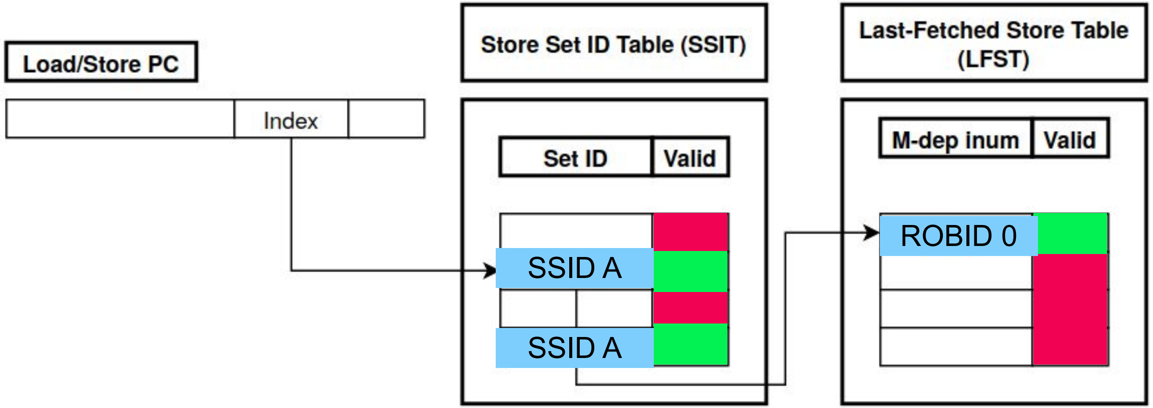
```
     STORE   R1  R2  IMM     # mem[ Addr ??? ] <-  R1
     (…)  STORE   R1  R8  IMM     # mem[ Addr ??? ] <-  R1
     LOAD    R3  R4  IMM     # R3 <-  mem[ Addr A ]

➡    ADD     R5  R3  R3      # R5 <-  R3+R3
```



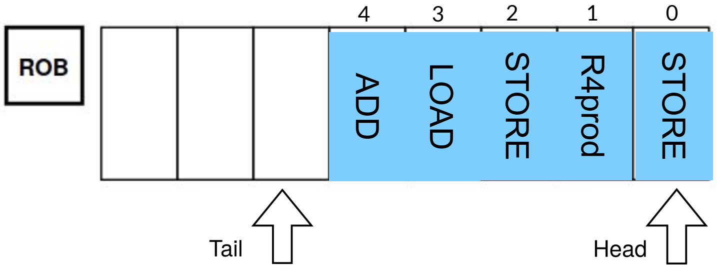
			4	3	2	1	0
			ADD	LOAD	STORE	R4prod	STORE

OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
STORE	2	🟥 !	! X	🟩
LOAD	3	🟩		🟥
ADD	4	🟥	X	🟩

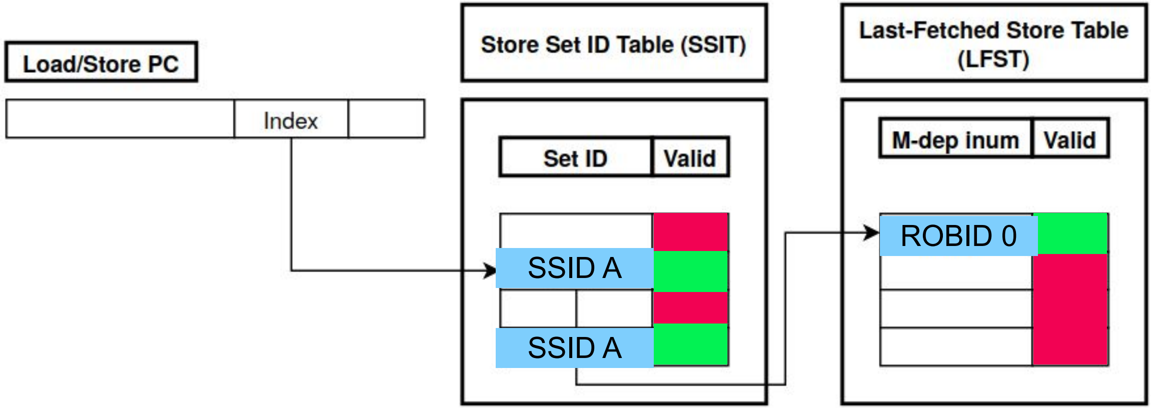


```
     STORE   R1  R2  IMM     # mem[Addr ???] <- R1
     (...)  STORE  R1  R8  IMM    # mem[Addr ???] <- R1
     LOAD    R3  R4  IMM     # R3 <- mem[Addr A]

➡    ADD     R5  R3  R3      # R5 <- R3+R3
```



OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟥	X	🟩
STORE	2	🟥	X	🟩
LOAD	3	🟩	0	🟥
ADD	4	🟥	X	🟩



Store Set ID Table (SSIT)

Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

Last-Fetched Store Table (LFST)

M-dep inum	Valid
ROBID 0	🟩
	🟥
	🟥
	🟥

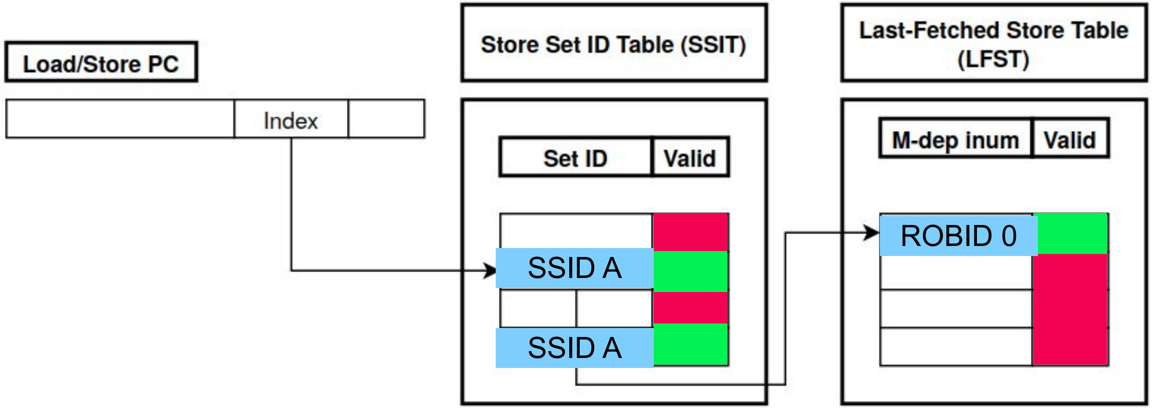
```
      STORE    R1  R2   IMM       # mem[ Addr A ] <-  R1
      (…)  STORE   R1   R8   IMM      # mem[ Addr ??? ] <-  R1
      LOAD     R3  R4   IMM       # R3 <-  mem[ Addr A ]

➡     ADD      R5  R3   R3        # R5 <-  R3+R3
```

ROB

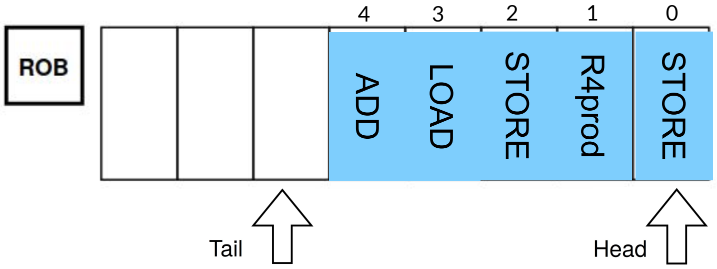
			4	3	2	1	0
			ADD	LOAD	STORE	R4prod	STORE
		Tail ⇧					Head ⇧

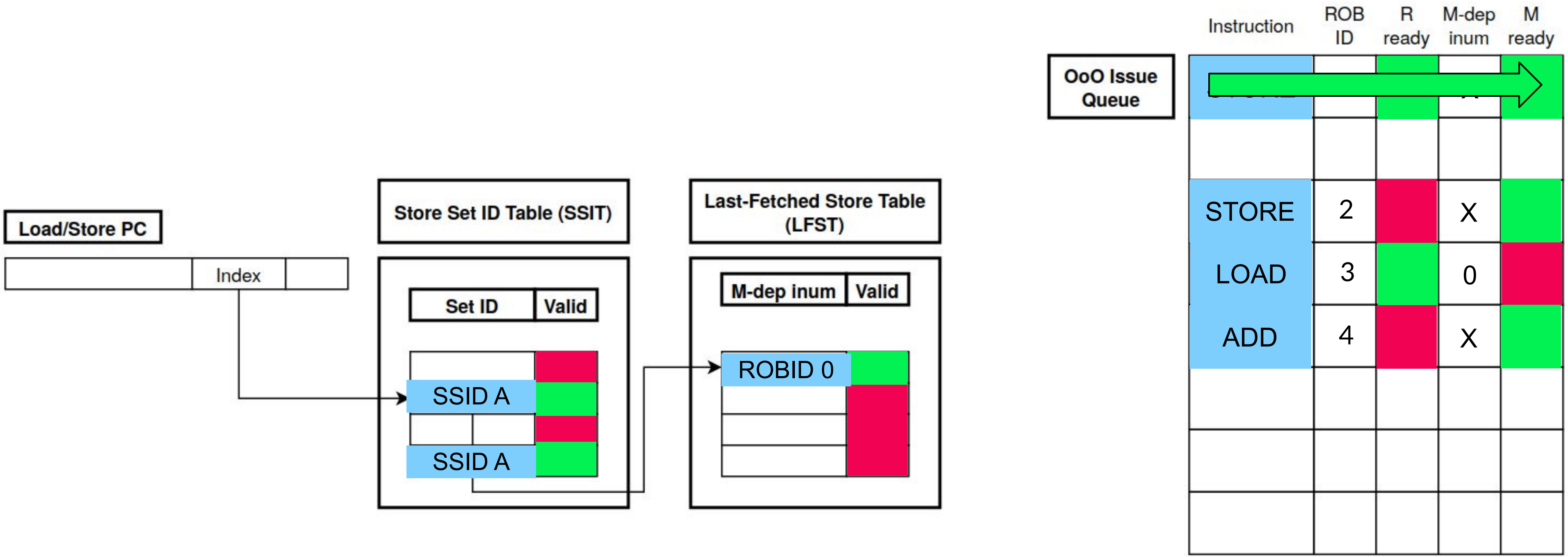
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	0	🟩	X	🟩
STORE	2	🟥	X	🟩
LOAD	3	🟩	0	🟥
ADD	4	🟥	X	🟩



```
STORE    R1  R2   IMM      # mem[ Addr A ] <-  R1
(…)  STORE   R1  R8  IMM      # mem[ Addr ??? ]  <-  R1
LOAD     R3  R4   IMM      # R3 <-  mem[ Addr A ]

ADD      R5  R3   R3       # R5 <-  R3+R3
```



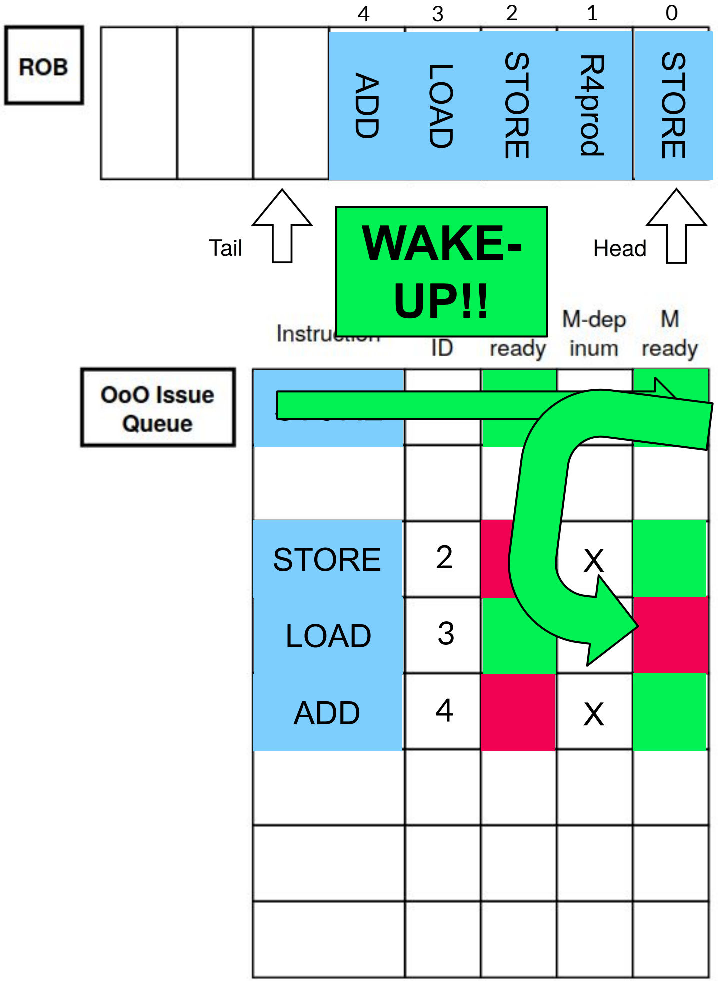


Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

M-dep inum	Valid
ROBID 0	🟩
	🟥
	🟥
	🟥

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE		🟩	X	🟩
STORE	2	🟥	X	🟩
LOAD	3	🟩	0	🟥
ADD	4	🟥	X	🟩

```
STORE   R1  R2  IMM     # mem[Addr A] <- R1
(…)  STORE  R1  R8  IMM     # mem[Addr ???] <- R1
LOAD    R3  R4  IMM     # R3 <- mem[Addr A]

ADD     R5  R3  R3      # R5 <- R3+R3
```

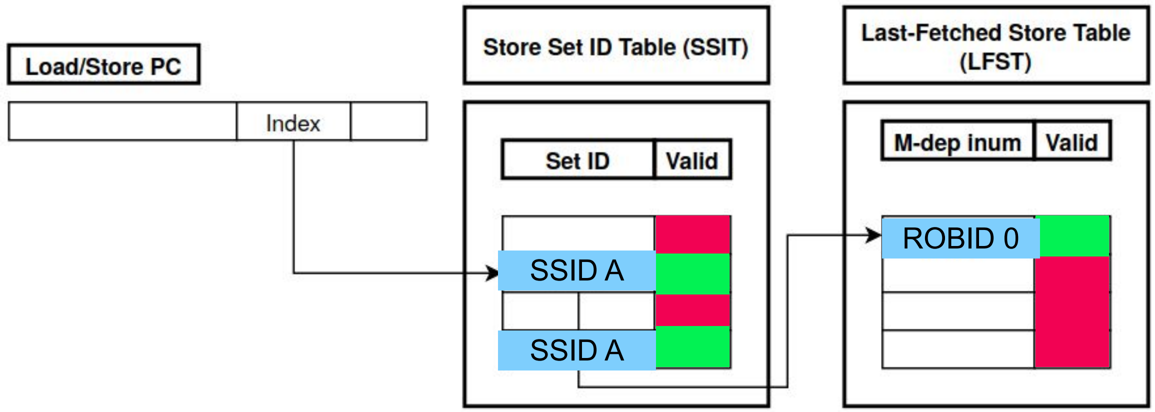


ROB

4	3	2	1	0
ADD	LOAD	STORE	R4prod	STORE

OoO Issue Queue

Instruction	ID	ready	M-dep inum	M ready
STORE		🟩		🟩
STORE	2	🟥	X	🟩
LOAD	3	🟩		🟥
ADD	4	🟥	X	🟩



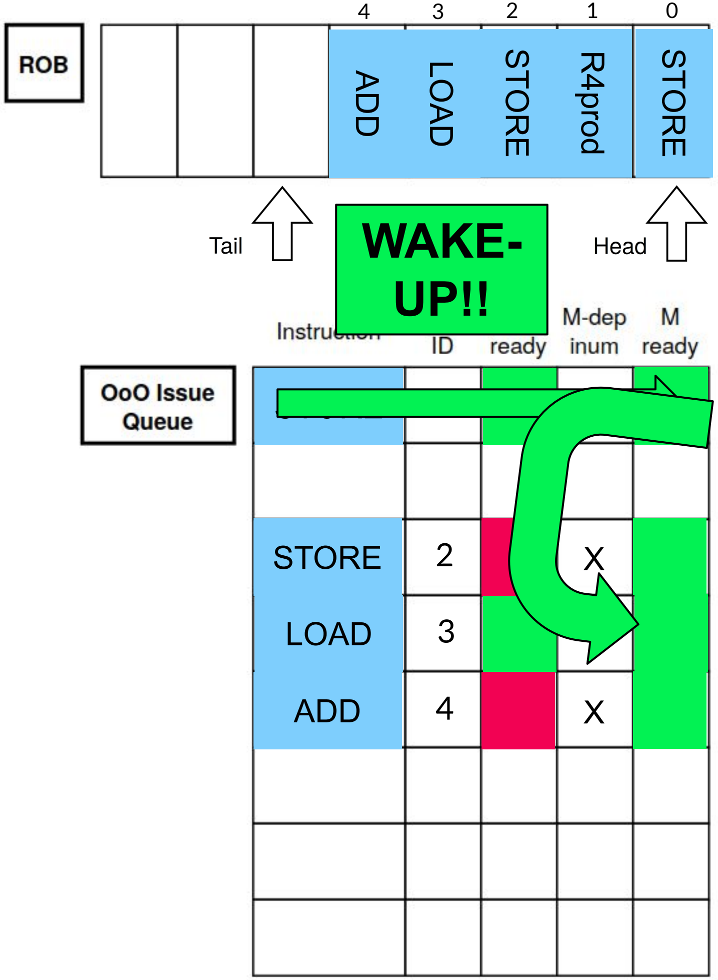
Store Set ID Table (SSIT)

Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

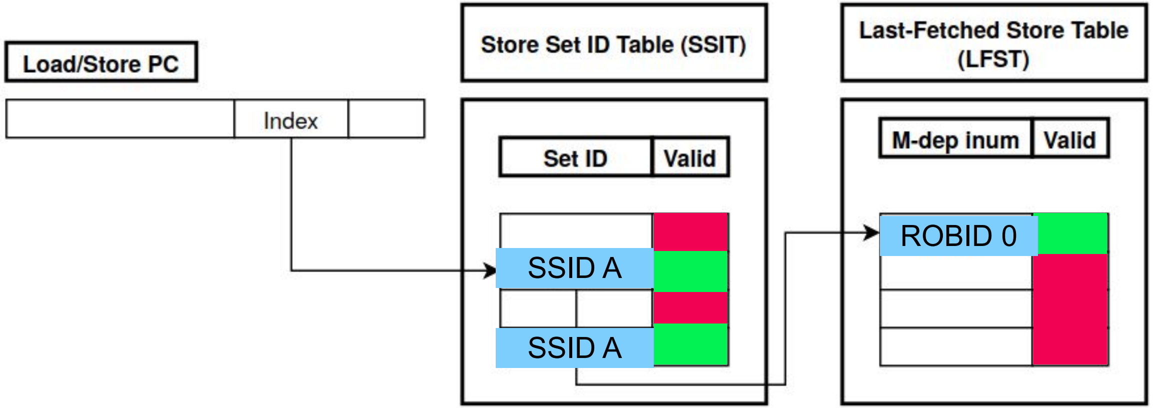
Last-Fetched Store Table (LFST)

M-dep inum	Valid
ROBID 0	🟩
	🟥
	🟥
	🟥

```
STORE   R1  R2  IMM     # mem[ Addr A ] <- R1
(…)  STORE   R1  R8  IMM     # mem[ Addr ??? ] <- R1
LOAD    R3  R4  IMM     # R3 <- mem[ Addr A ]

ADD     R5  R3  R3      # R5 <- R3+R3
```



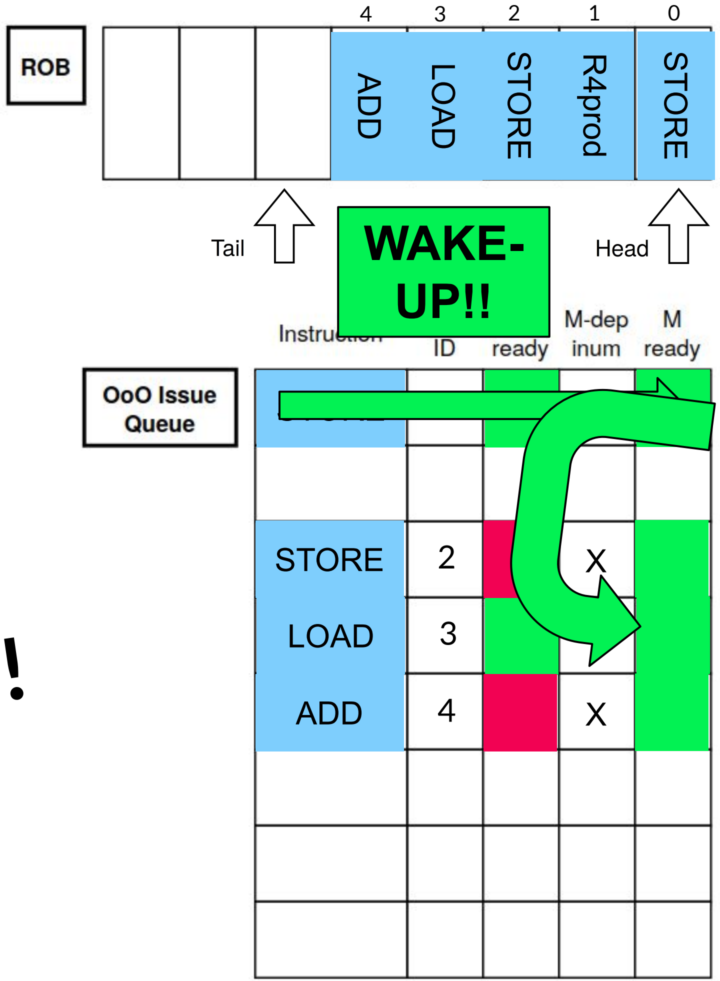
Instruction	ID	ready	M-dep inum	M ready
STORE		🟩		🟩
STORE	2	🟥	X	🟩
LOAD	3	🟩		🟩
ADD	4	🟥	X	🟩



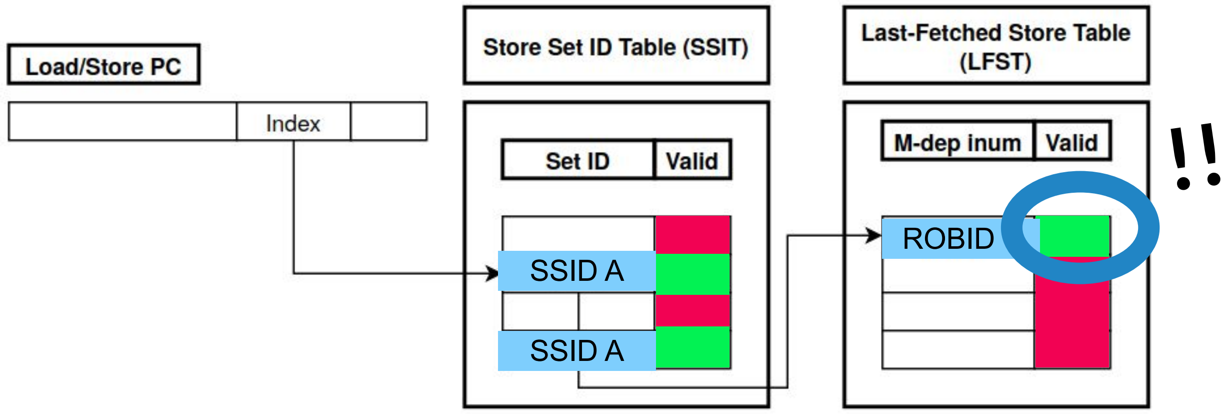
Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

M-dep inum	Valid
ROBID 0	🟩
	🟥
	🟥
	🟥

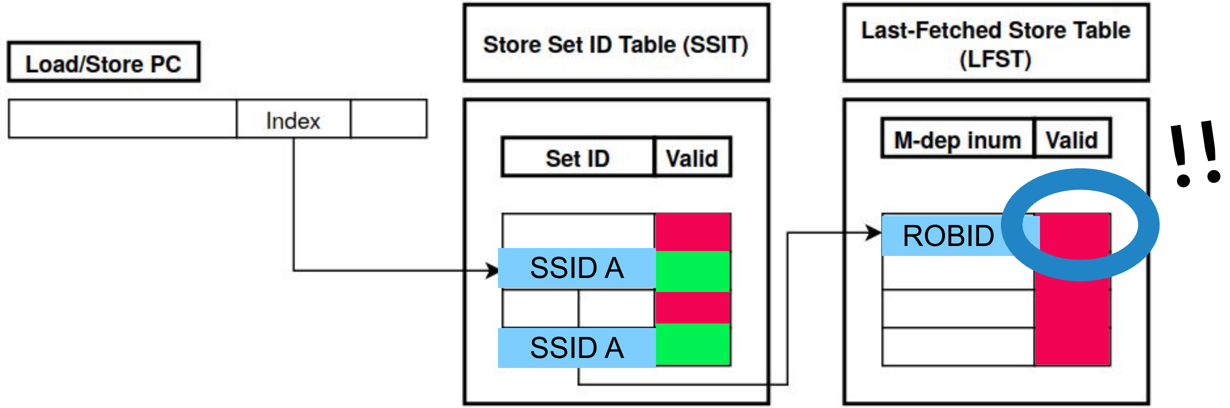
```
STORE    R1  R2   IMM      # mem[ Addr A ] <-  R1
(…)  STORE   R1   R8   IMM      # mem[ Addr ??? ]  <-  R1
LOAD     R3  R4   IMM      # R3 <- mem[ Addr A ]

ADD      R5  R3   R3       # R5 <-  R3+R3
```



Instruction	ID	ready	M-dep inum	M ready
STORE		🟩		🟩
STORE	2	🟥	X	🟩
LOAD	3	🟩		🟩
ADD	4	🟥	X	🟩

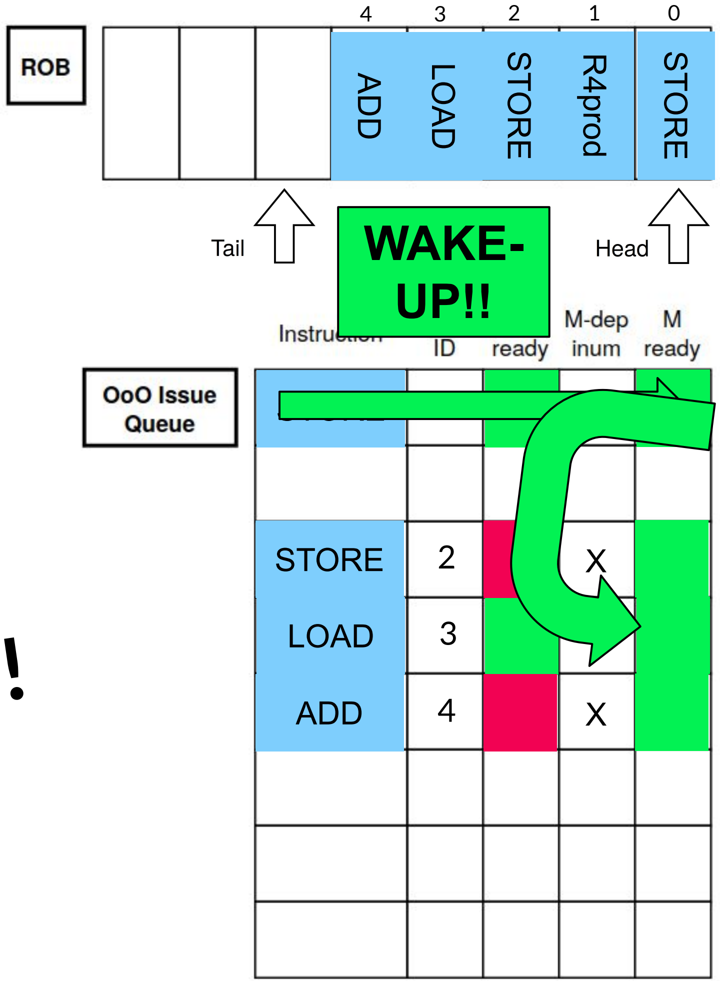


```
STORE    R1  R2   IMM       # mem[Addr A] <-  R1
(...)  STORE   R1   R8  IMM       # mem[Addr ???] <-  R1
LOAD     R3  R4   IMM       # R3 <-  mem[Addr A]

ADD      R5  R3   R3        # R5 <-  R3+R3
```



Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

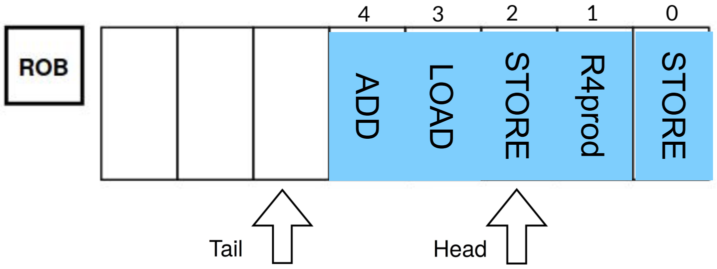
M-dep inum	Valid
ROBID	🟥
	🟥
	🟥
	🟥



			4	3	2	1	0
			ADD	LOAD	STORE	R4prod	STORE

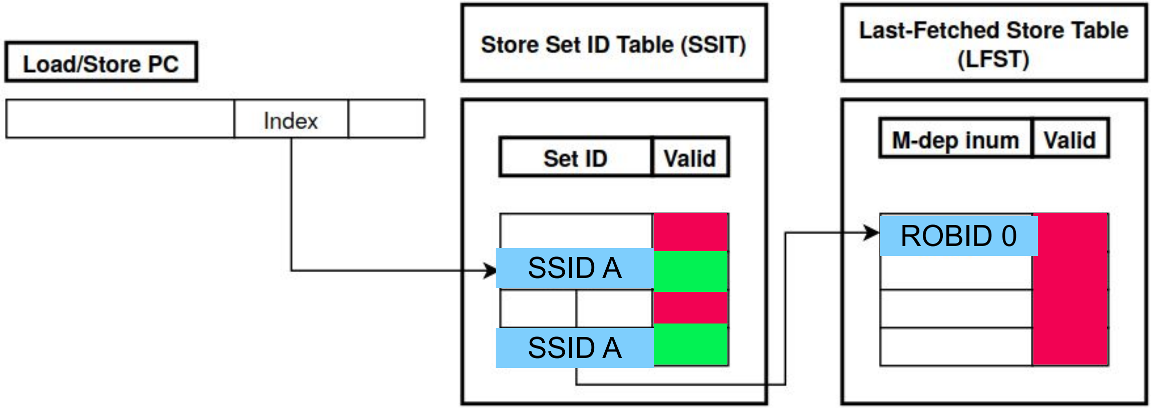
Instruction	ID	ready	M-dep inum	M ready
STORE		🟩		🟩
STORE	2	🟥	X	🟩
LOAD	3	🟩		🟩
ADD	4	🟥	X	🟩

```
STORE   R1  R2  IMM      # mem[ Addr A ] <- R1
(…)  STORE   R1  R8  IMM      # mem[ Addr ??? ] <- R1
LOAD    R3  R4  IMM      # R3 <- mem[ Addr A ]

➡ ADD     R5  R3  R3       # R5 <- R3+R3
```

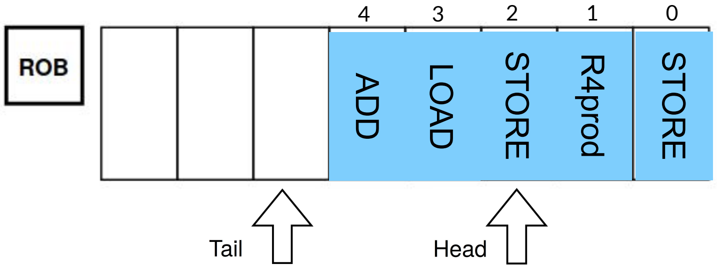


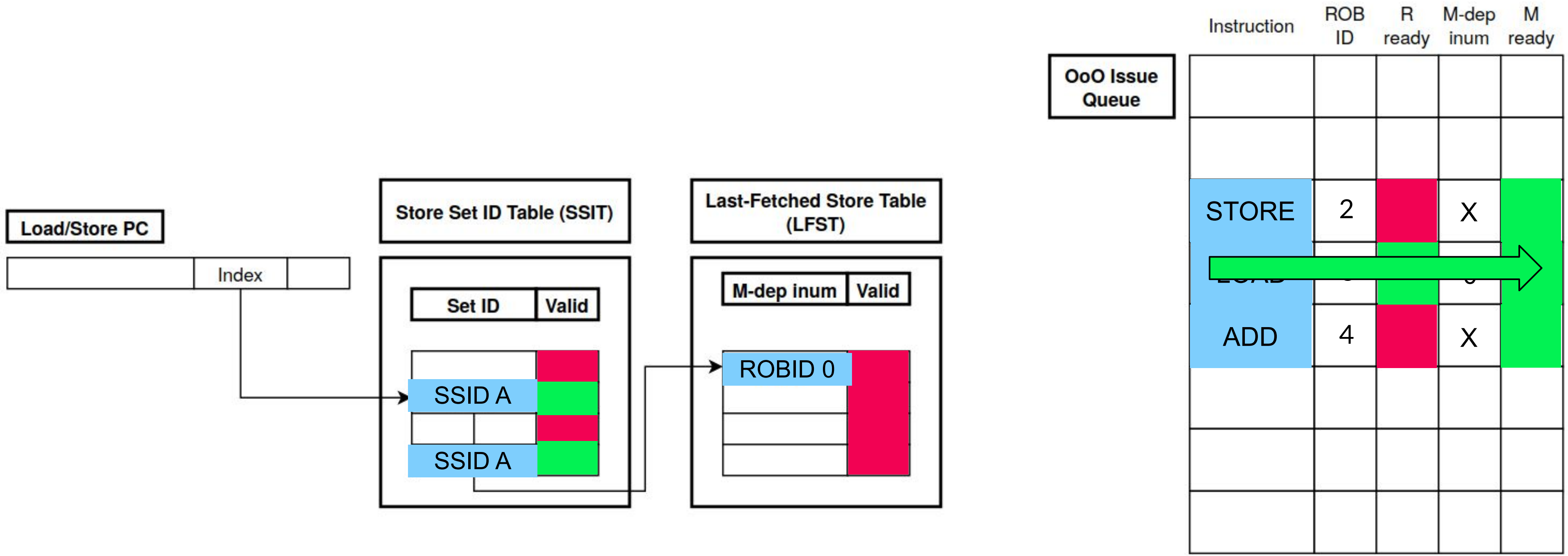
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	2	🟥	X	🟩
LOAD	3	🟩	0	🟩
ADD	4	🟥	X	🟩



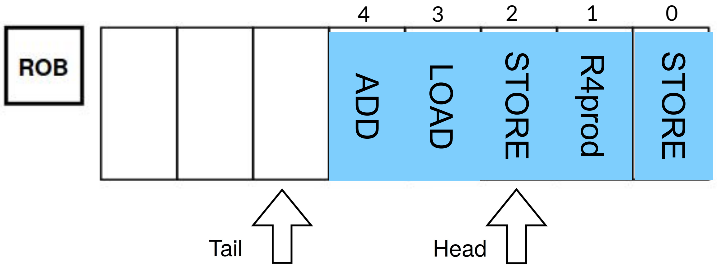
```
STORE   R1  R2  IMM     # mem[Addr A] <- R1
(...)  STORE  R1  R8  IMM    # mem[Addr ???] <- R1
LOAD    R3  R4  IMM     # R3 <- mem[Addr A]

ADD     R5  R3  R3      # R5 <- R3+R3
```

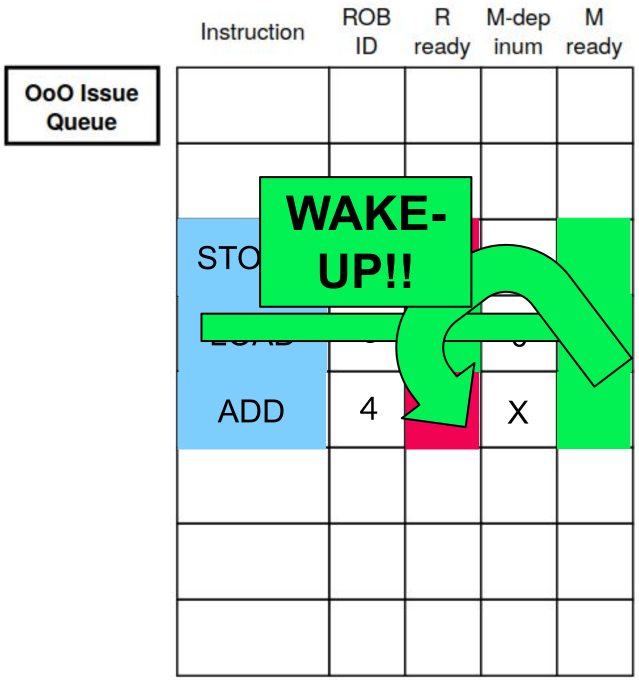




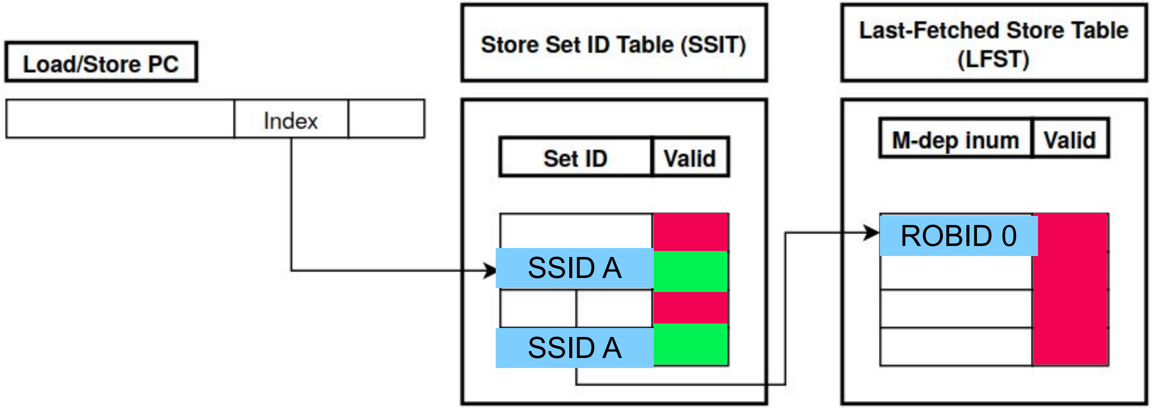
Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	2	🟥	X	🟩
LOAD	3	🟩	0	🟩
ADD	4	🟥	X	🟩

```
   STORE   R1  R2   IMM        # mem[ Addr A ] <- R1
   (…)  STORE   R1  R8   IMM       # mem[ Addr ??? ] <- R1
   LOAD    R3  R4   IMM        # R3 <- mem[ Addr A ]

➡  ADD     R5  R3   R3         # R5 <- R3+R3
```

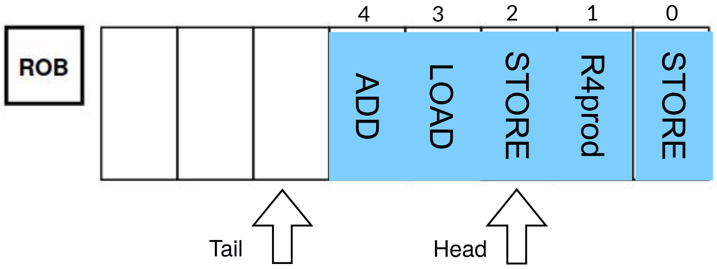


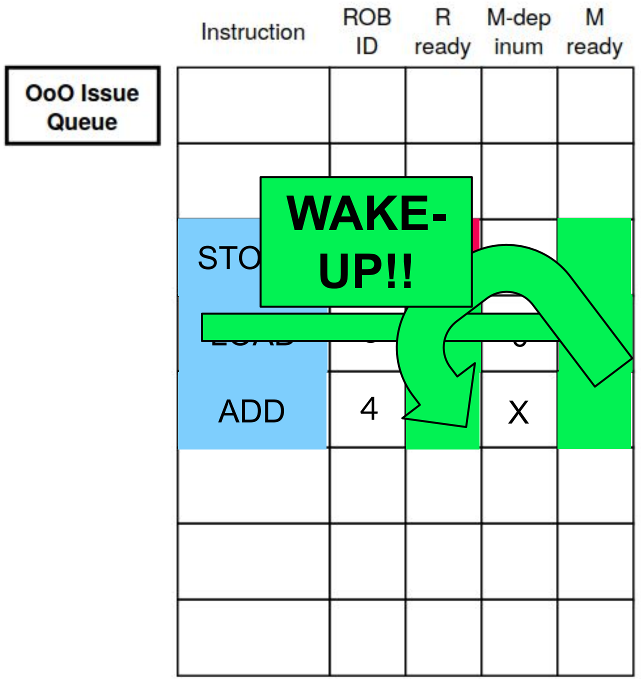


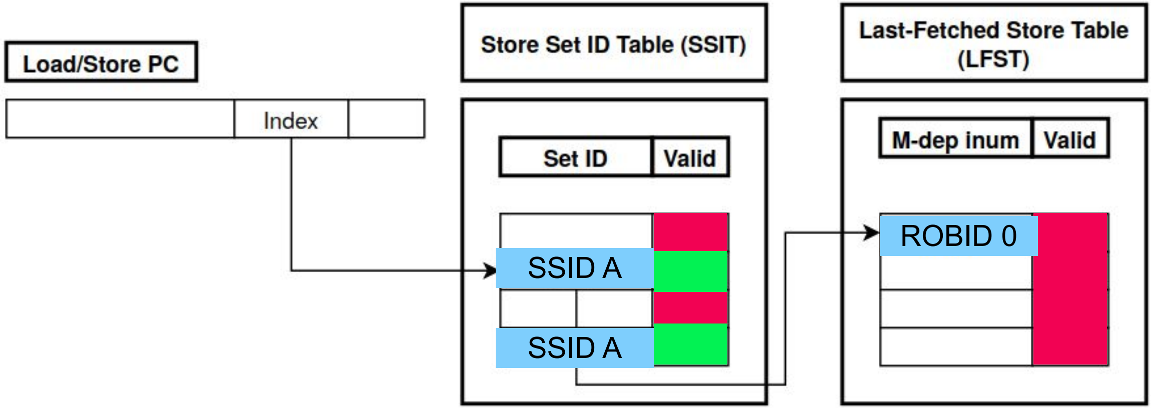
Instruction	ROB ID	R ready	M-dep inum	M ready
STO		🟥		🟩
LOAD				🟩
ADD	4	🟥	X	🟩



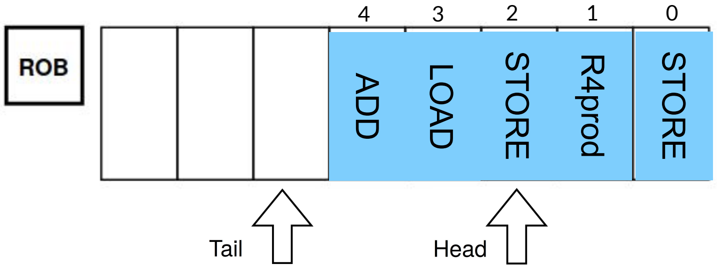
```
STORE    R1  R2   IMM       # mem[ Addr A  ] <-  R1
 (…)  STORE   R1   R8   IMM       # mem[ Addr ??? ] <-  R1
 LOAD    R3  R4   IMM       # R3 <-  mem[ Addr A  ]

 ADD     R5  R3   R3        # R5 <-  R3+R3
```





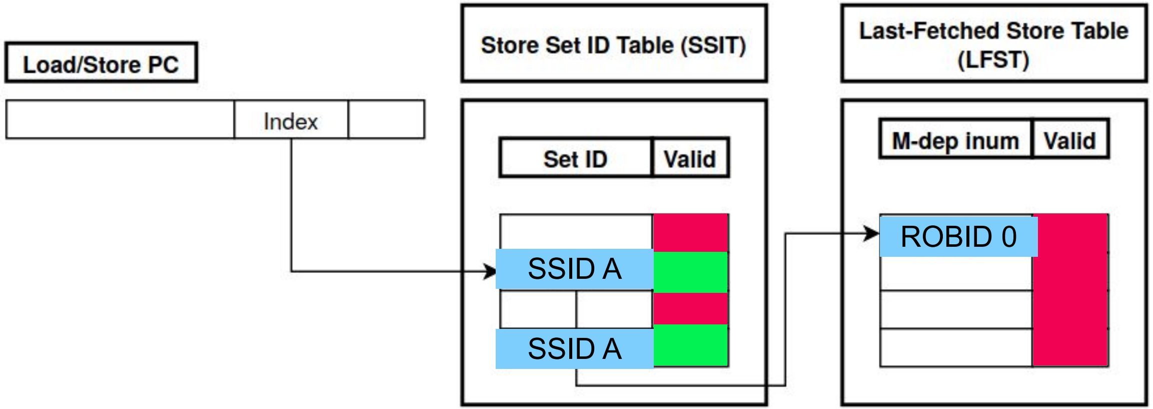


```
STORE   R1  R2   IMM      # mem[Addr A] <- R1
(…)  STORE   R1  R8  IMM      # mem[Addr ???] <- R1
LOAD    R3  R4   IMM      # R3 <- mem[Addr A]

ADD     R5  R3   R3       # R5 <- R3+R3
```

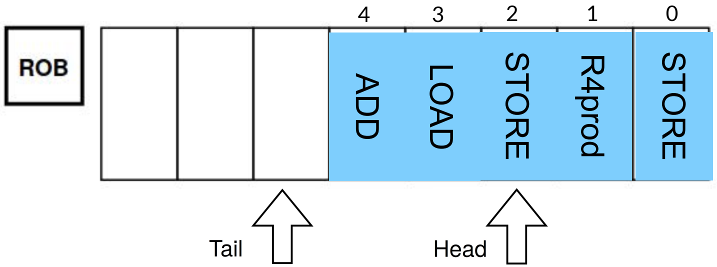


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	2	🟥	X	🟩
ADD	4	🟩	X	🟩

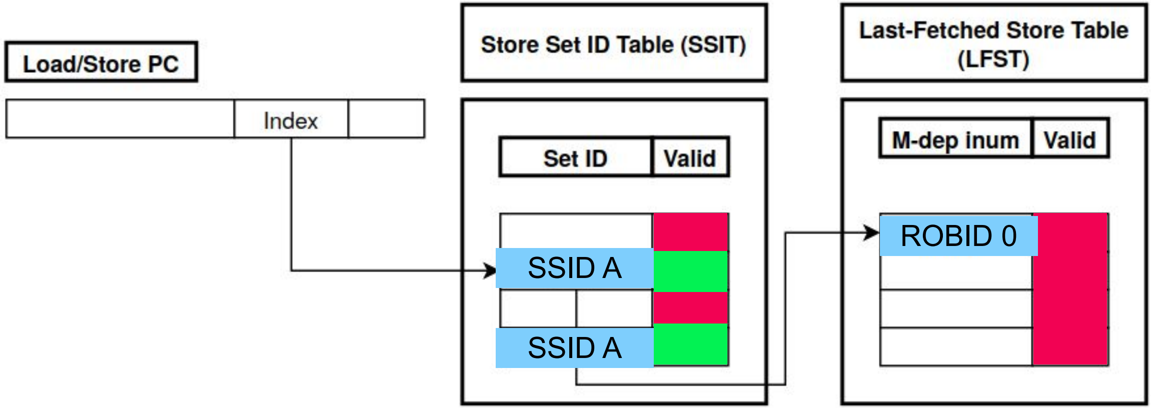


```
STORE   R1  R2   IMM      # mem[ Addr A ] <-  R1
(…)  STORE   R1   R8   IMM      # mem[ Addr ??? ] <-  R1
LOAD    R3  R4   IMM      # R3 <-  mem[ Addr A ]

➡ ADD     R5  R3   R3       # R5 <-  R3+R3
```

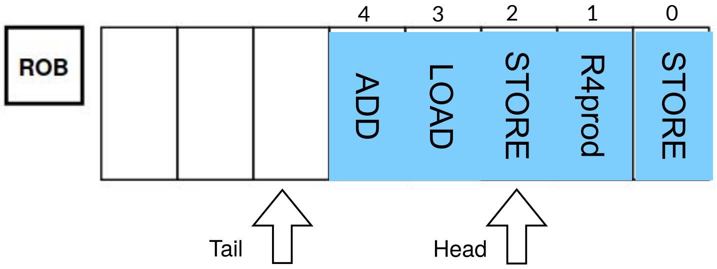


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	2	🟥	X	🟩
[illegible]	[illegible]	🟩	[illegible]	🟩

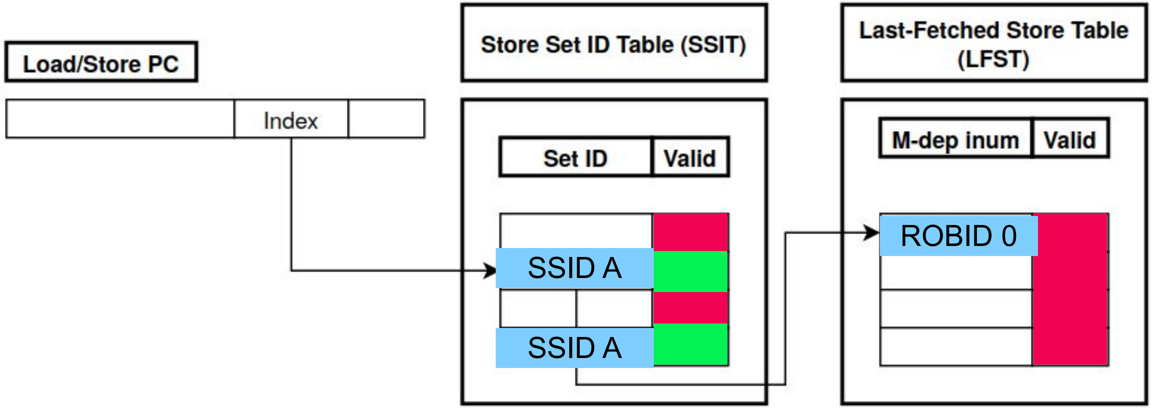


```
      STORE   R1  R2  IMM      # mem[Addr A] <- R1
      (…)  STORE   R1  R8  IMM      # mem[Addr ???] <- R1
      LOAD    R3  R4  IMM      # R3 <- mem[Addr A]

      ADD     R5  R3  R3       # R5 <- R3+R3
```

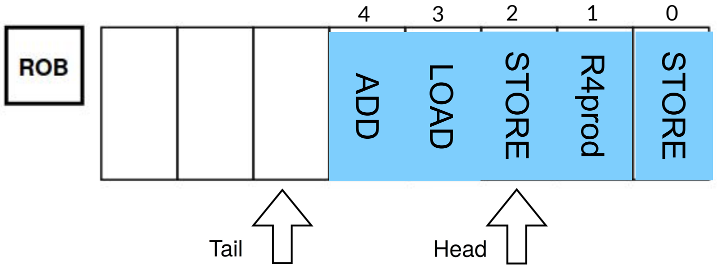


OoO Issue Queue

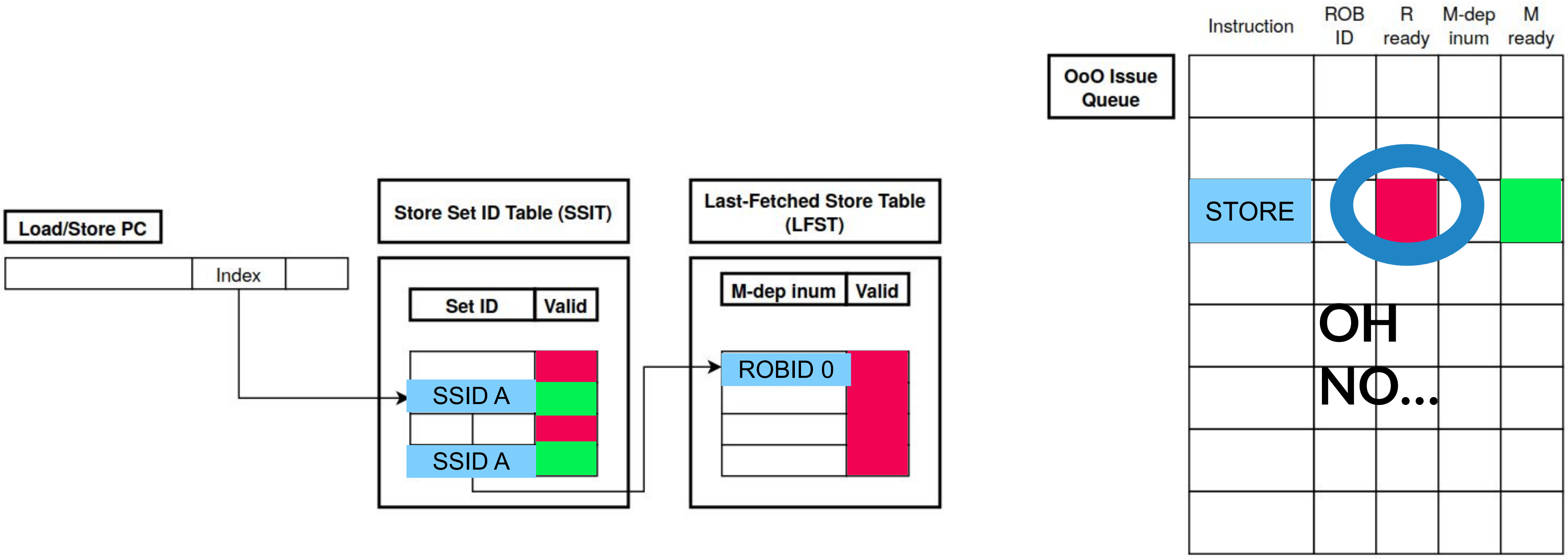
Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	2	🟥	X	🟩



```
STORE    R1  R2   IMM       # mem[ Addr A  ] <-  R1
(…)  STORE   R1   R8   IMM      # mem[ Addr ??? ] <-  R1
LOAD     R3  R4   IMM       # R3 <-  mem[ Addr A  ]

ADD      R5  R3   R3        # R5 <-  R3+R3
```



			4	3	2	1	0
			ADD	LOAD	STORE	R4prod	STORE



Store Set ID Table (SSIT)

Set ID	Valid
	🟥
SSID A	🟩
	🟥
SSID A	🟩

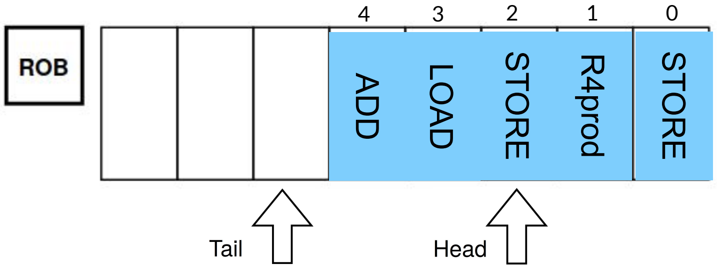
Last-Fetched Store Table (LFST)

M-dep inum	Valid
ROBID 0	🟥
	🟥
	🟥
	🟥

OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE		🟥		🟩

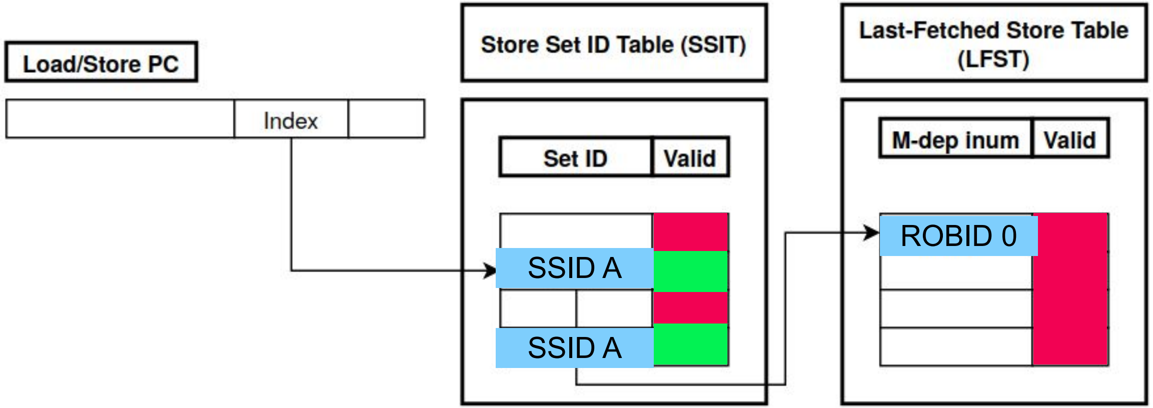
```
STORE    R1  R2   IMM       # mem[Addr A] <- R1
(…)  STORE   R1  R8  IMM       # mem[Addr ???] <- R1
LOAD     R3  R4   IMM       # R3 <- mem[Addr A]

➡ ADD      R5  R3   R3        # R5 <- R3+R3
```



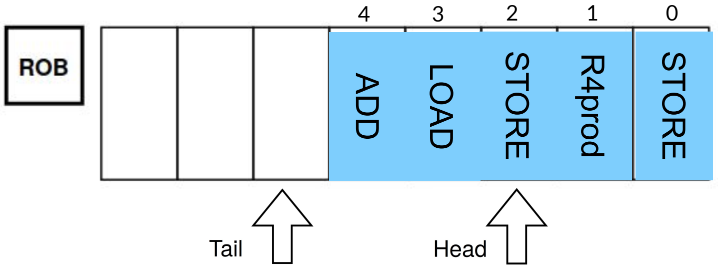
OoO Issue Queue

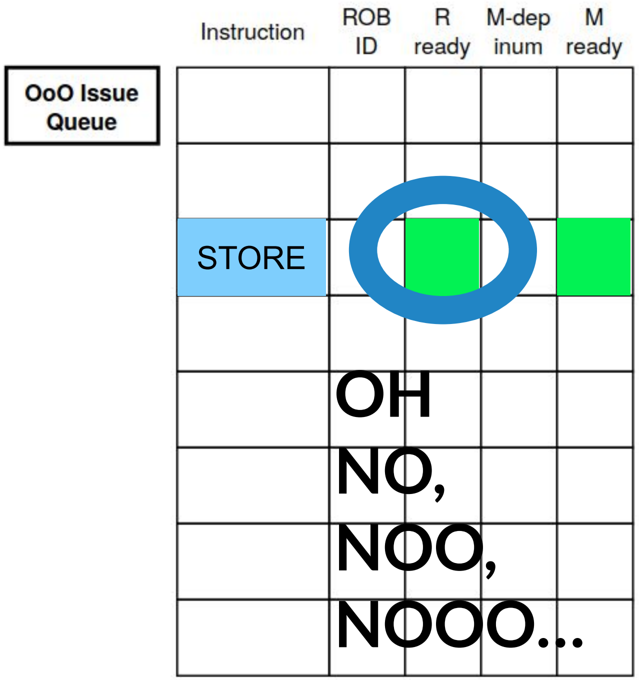
Instruction	ROB ID	R ready	M-dep inum	M ready
STORE		🟩		🟩

OH NO, NOO...

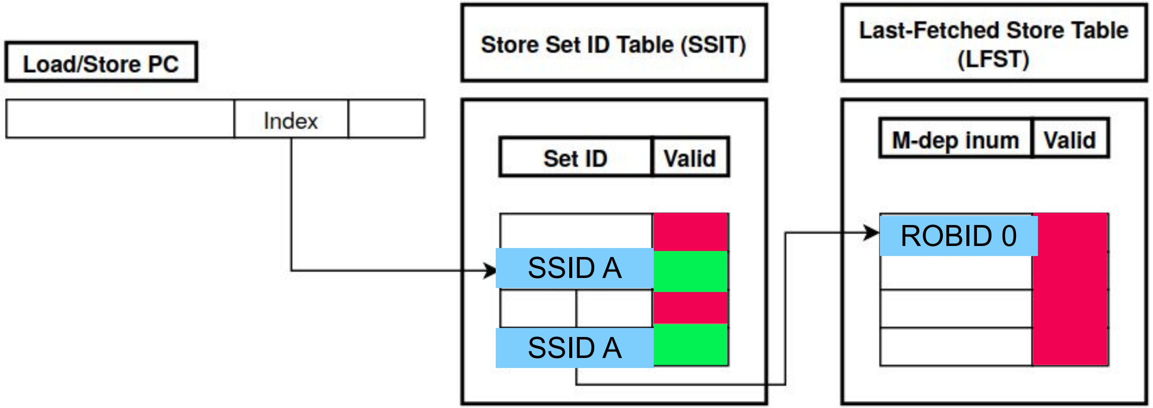


```
      STORE   R1  R2  IMM      # mem[Ad    ] <- R1
       (…)  STORE  R1  R8  IMM     # mem[Addr ???] <- R1
      LOAD    R3  R4  IMM      # R3 <- mem[Addr A]

      ADD     R5  R3  R3       # R5 <- R3+R3
```

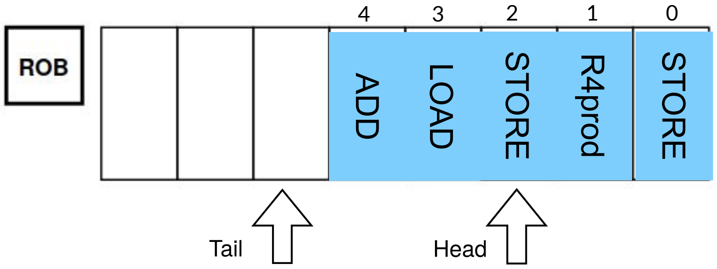




OH NO, NOO, NOOO...



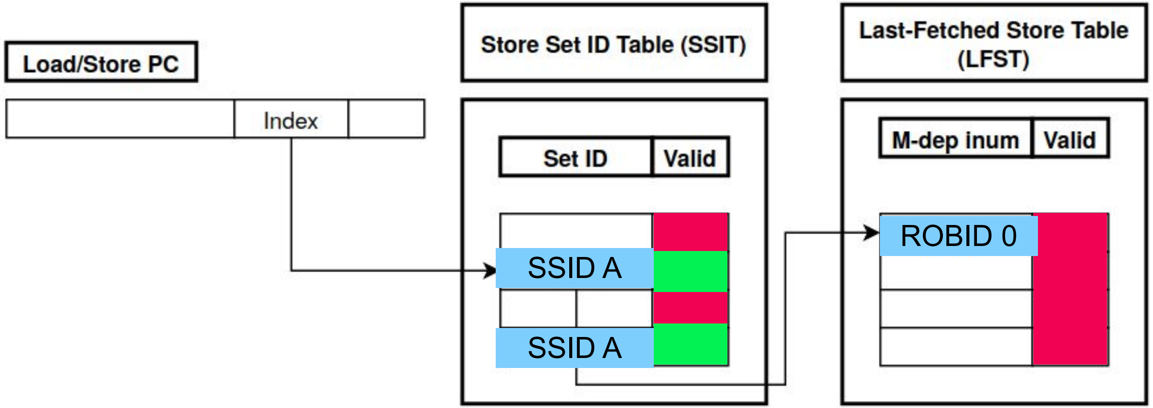
```
STORE    R1  R2   IMM       # mem[Addr A] <-  R1
(…)  STORE   R1   R8   IMM      # mem[Addr A] <-  R1
LOAD     R3  R4   IMM       # R3 <-  mem[Addr A]

ADD      R5  R3   R3        # R5 <-  R3+R3
```



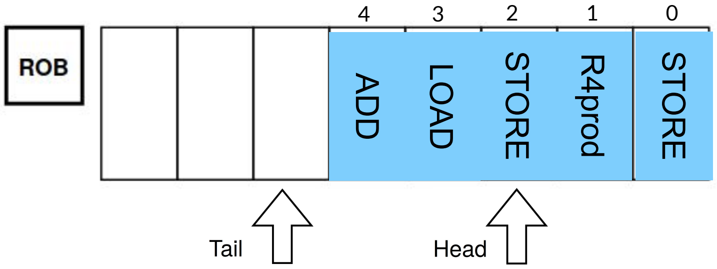
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE		🟩		🟩

OH NO, NOO, NOOO...



```
STORE   R1  R2   IMM      # mem[Ad     ] <- R1
 (…)  STORE   R1   R8   IMM     # mem  Addr A  ] <- R1
LOAD    R3  R4   IMM      # R3 <- mem[ Addr A ]

ADD     R5  R3   R3       # R5 <- R3+R3
```

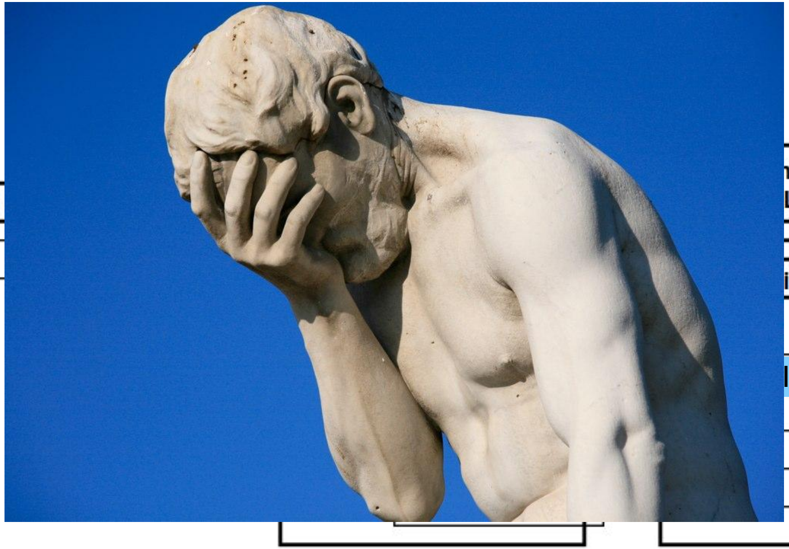


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE		🟩		🟩

OH NO, NOO, NOOO…

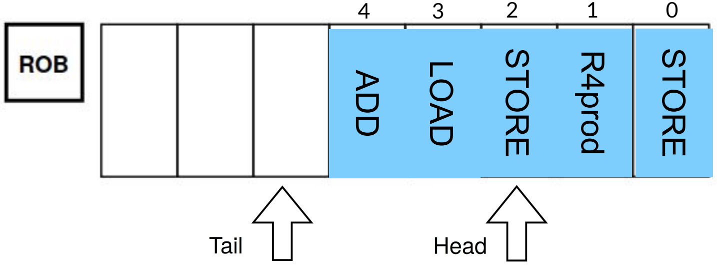
Load/Store PC

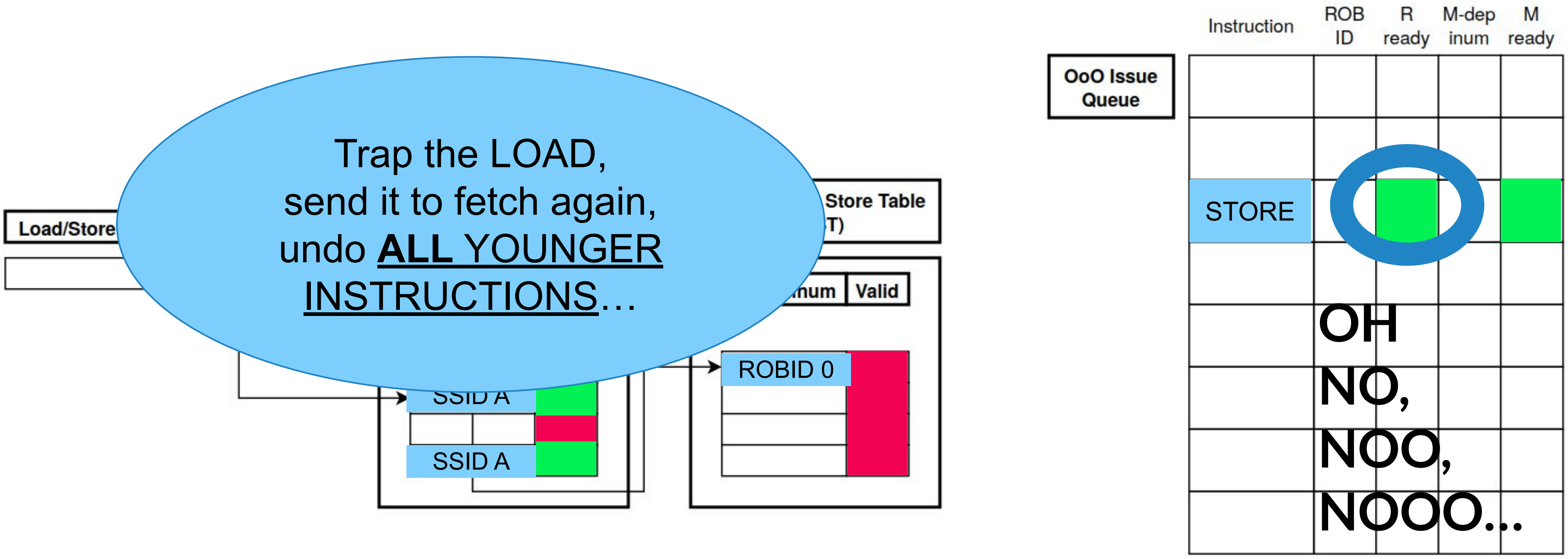


ed Store Table (LFST)

inum	Valid
ID 0	🟥
	🟥
	🟥
	🟥

```
    STORE   R1  R2   IMM       # mem[Ad    ] <-  R1
    (…)  STORE   R1  R8  IMM     # mem[Addr A] <-  R1
    LOAD    R3  R4   IMM       # R3 <-  me   [ Addr A ]

➡   ADD     R5  R3   R3        # R5 <-  R3+R3
```

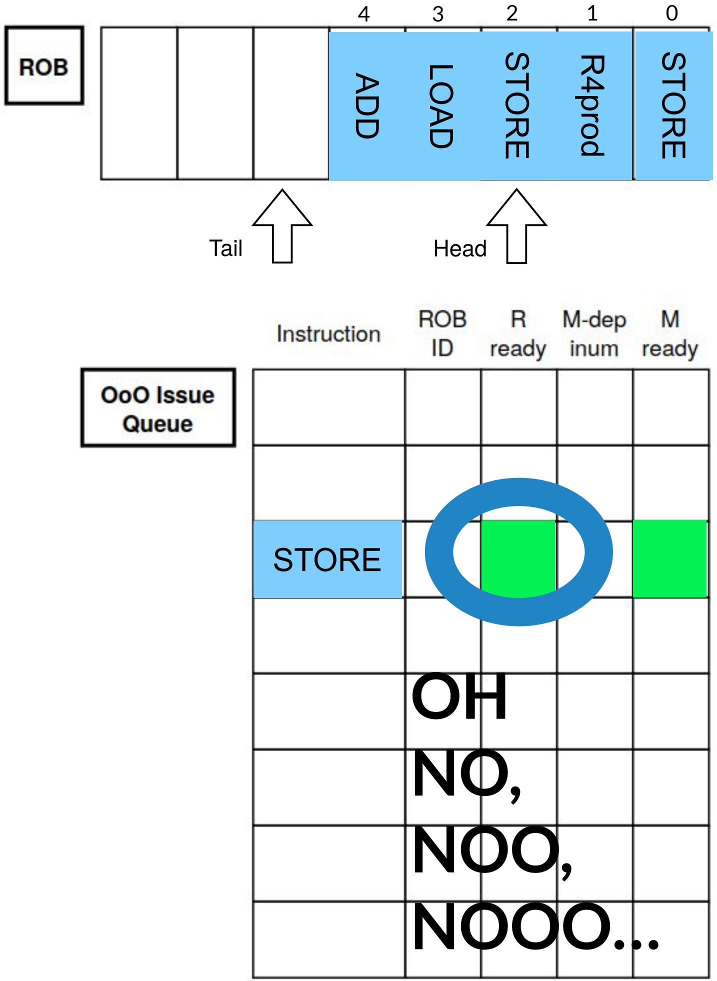




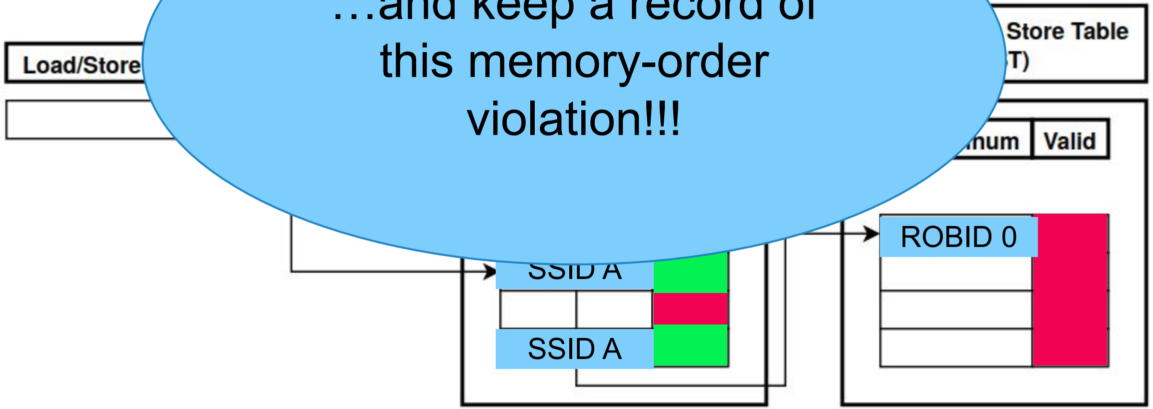
Trap the LOAD,
send it to fetch again,
undo ***ALL YOUNGER INSTRUCTIONS***…

OH NO, NOO, NOOO...

```
STORE   R1  R2  IMM      # mem[Ad   ] <- R1
(…)  STORE   R1  R8  IMM      # mem[Addr A] <- R1
LOAD    R3  R4  IMM      # R3 <- mem[Addr A]

ADD     R5  R3  R3       # R5 <- R3+R3
```



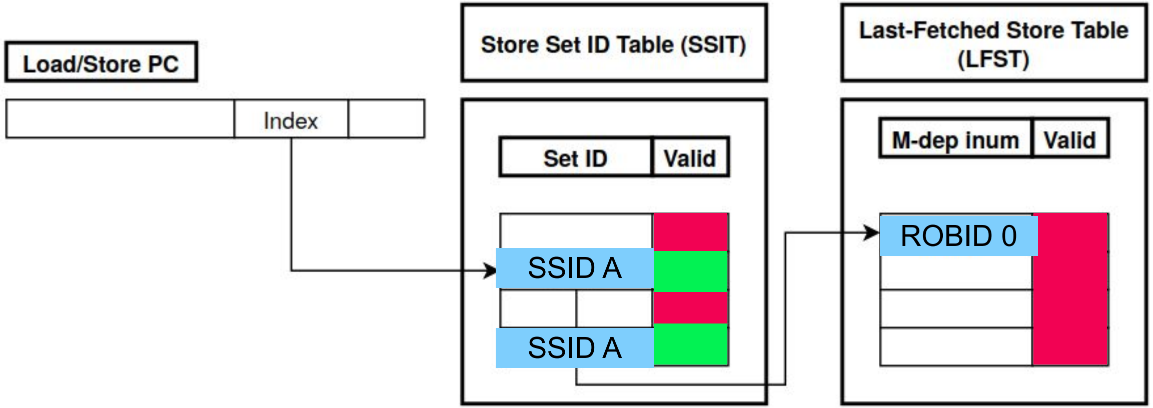
…and keep a record of this memory-order violation!!!



```
STORE    R1  R2   IMM      # mem[Addr A] <- R1
(…)  STORE   R1  R8  IMM      # mem[Addr A] <- R1
LOAD     R3  R4   IMM      # R3 <- mem[Addr A]

ADD      R5  R3   R3       # R5 <- R3+R3
```

NEW STORE's PC → SSID A

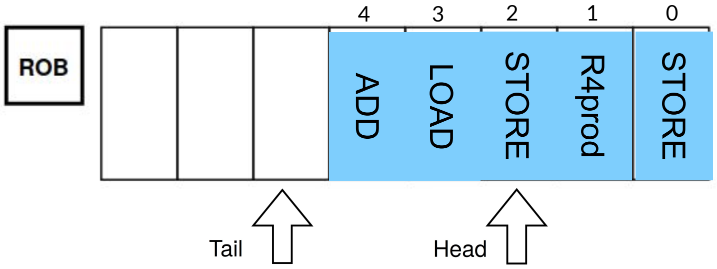
ROB				4	3	2	1	0
				ADD	LOAD	STORE	R4prod	STORE
			Tail			Head		

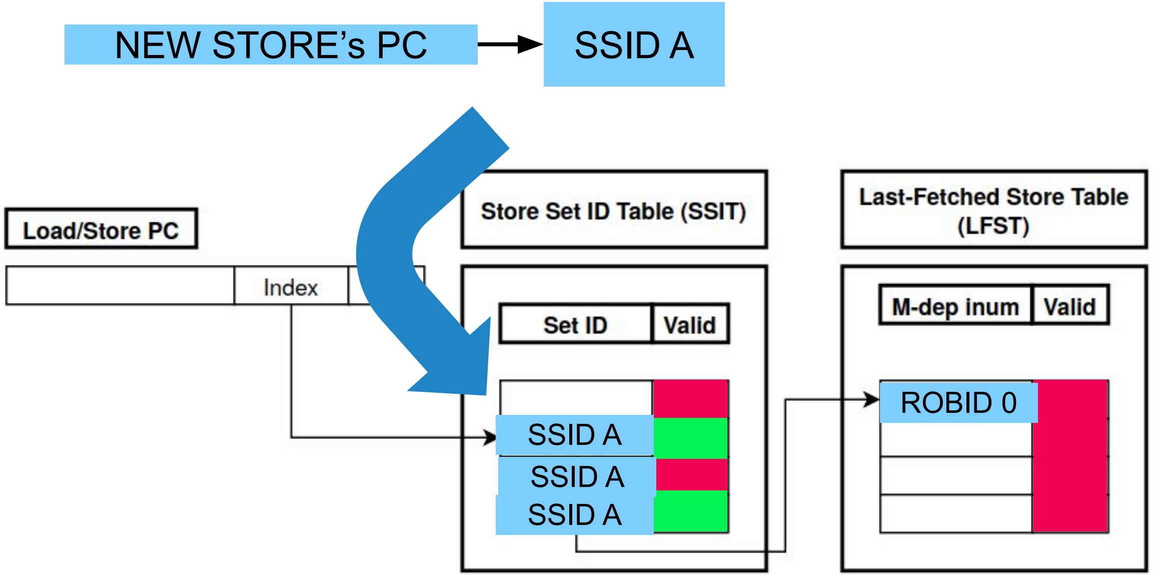


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	2	🟩	X	🟩

```
STORE   R1  R2  IMM      # mem[Ad    ] <- R1
(…)  STORE  R1  R8  IMM      # mem[Addr A] <- R1
LOAD    R3  R4  IMM      # R3 <- mem[Addr A]

ADD     R5  R3  R3       # R5 <- R3+R3
```

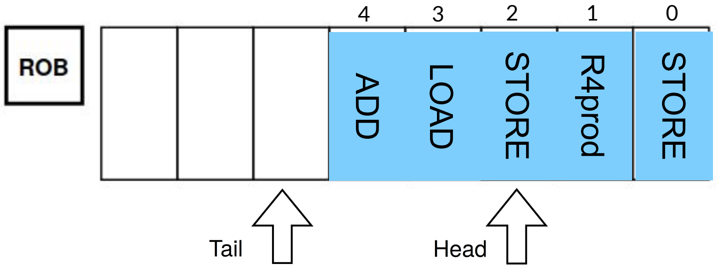




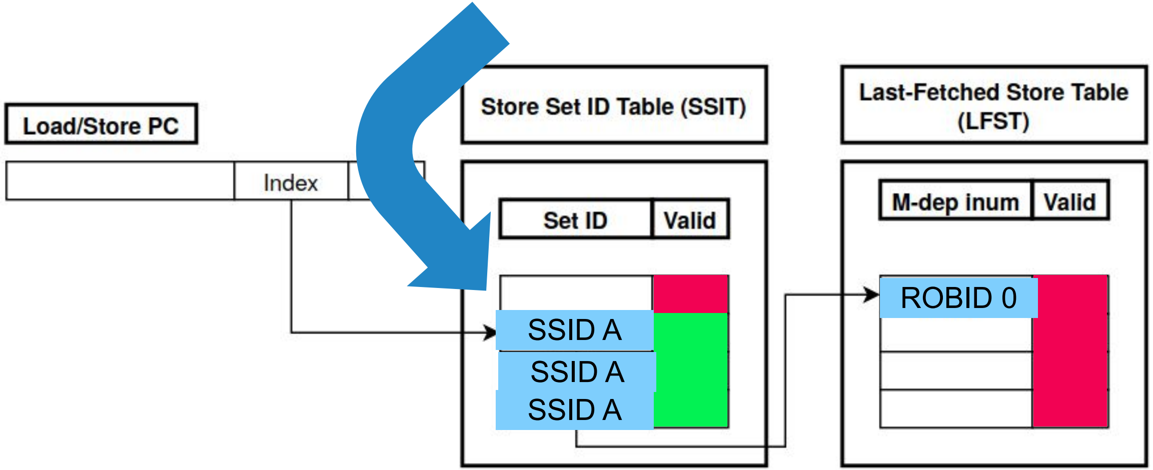
OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	2	🟩	X	🟩

```
STORE   R1  R2   IMM      # mem[Addr A] <- R1
(...)  STORE   R1  R8  IMM      # mem[Addr A] <- R1
LOAD    R3  R4   IMM      # R3 <- mem[Addr A]

ADD     R5  R3   R3       # R5 <- R3+R3
```



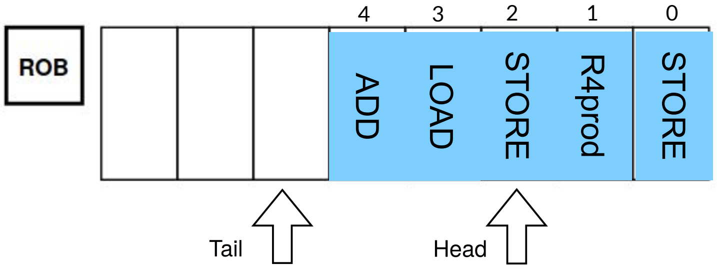
NEW STORE's PC → SSID A

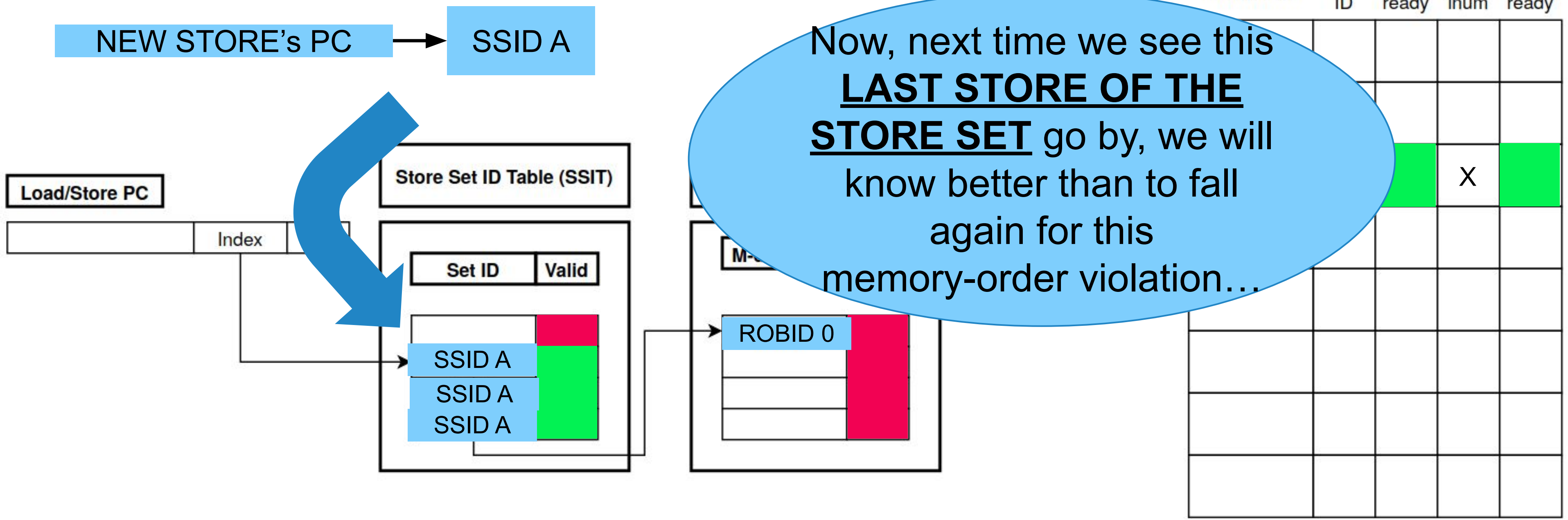


OoO Issue Queue

Instruction	ROB ID	R ready	M-dep inum	M ready
STORE	2	🟩	X	🟩

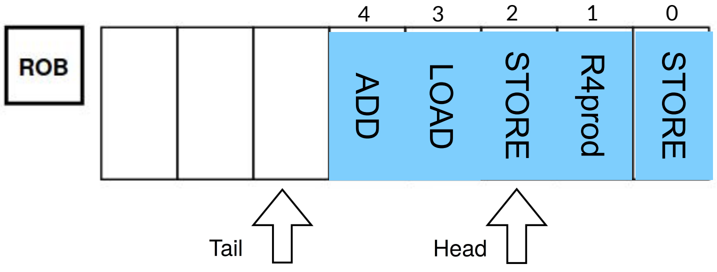
```
STORE    R1  R2   IMM      # mem[ Ad ] <-  R1
(…)  STORE   R1   R8   IMM      # mem[ Addr A ] <-  R1
LOAD     R3  R4   IMM      # R3 <-  mem[ Addr A ]

ADD      R5  R3   R3       # R5 <-  R3+R3
```

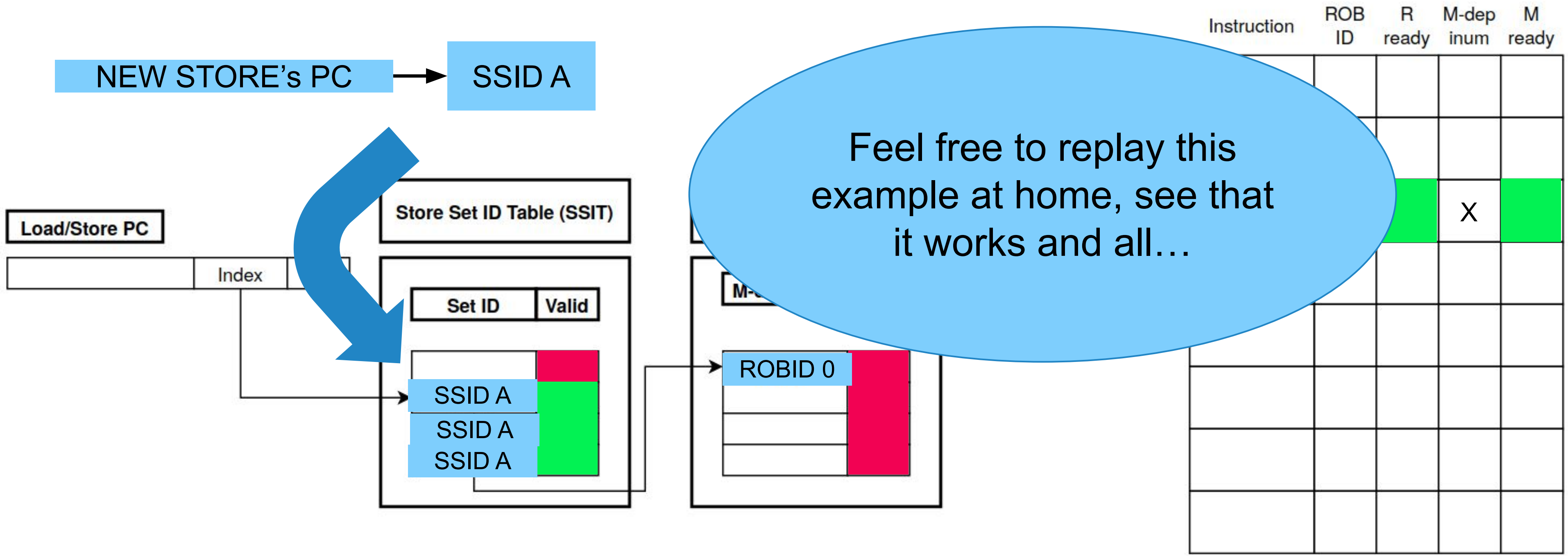




Now, next time we see this **LAST STORE OF THE STORE SET** go by, we will know better than to fall again for this memory-order violation…

```
STORE   R1  R2  IMM     # mem[Ad    ] <- R1
(…)  STORE   R1  R8  IMM     # mem[Addr A] <- R1
LOAD    R3  R4  IMM     # R3 <- mem[Addr A]

ADD     R5  R3  R3      # R5 <- R3+R3
```





Feel free to replay this example at home, see that it works and all…

Assignment Rules

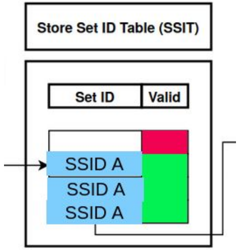
1. If neither the LOAD nor the STORE been assigned a store set, a SSID is generated and assigned to both instructions

2. If only the LOAD has been assigned a store set, the STORE is assigned to the LOAD’s store set

3. If only the STORE has been assigned a store set, the LOAD is assigned to the STORE store set

4. If both the LOAD and the STORE have already been assigned to store sets, one of the two store sets is declared the "winner" and the other inherits the winner SSID.

Clear conditions

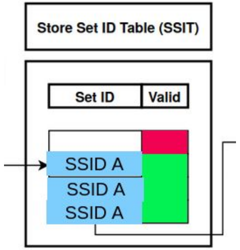


A SSIT entry is validated and an SSID is assigned to the corresponding entry when a LOAD or a STORE is involved in a memory-order violation. However, these entries in the SSIT remain valid for the rest of the program.

Unrelated LOADs or STOREs can share a SSIT entry after time, causing undesired dependences, a method of invalidating those entries is needed, authors propose two:

- ▷ …

Clear conditions



Unrelated LOADs or STOREs can share a SSIT entry after time, causing undesired dependences, a method of invalidating those entries is needed, authors propose two:

- Clear SSIT after an arbitrary amount of cycles, i.e. every million
- 2-bit counter per entry in a two-state branch-prediction fashion, where the MSB indicates if the entry is valid.

  Every time a LOAD-STORE dependence is enforced, it is checked if it generates a false dependency, then, the counter of the STORE is updated

Table sizes

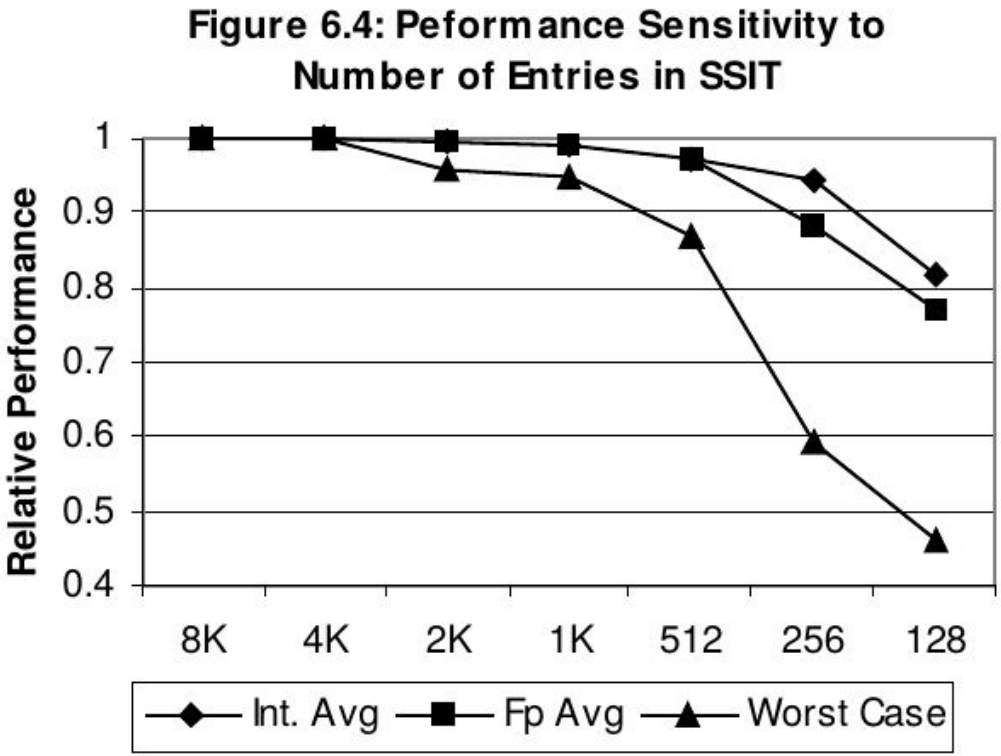


Figure 6.4: Peformance Sensitivity to Number of Entries in SSIT
Relative Performance
1
0.9
0.8
0.7
0.6
0.5
0.4
8K
4K
2K
1K
512
256
128
Int. Avg
Fp Avg
Worst Case

Thoughts about the paper

- The store set invalidation mechanisms seems on the simpler side, counters could interfere in some patterns.

- They mention the benefit of requiring executing STOREs in store sets in order eliminates write-after-write hazard detection mechanisms, but what happens two STOREs to the same address do not have a LOAD causing a memory-order violation??

- In the example of two STOREs followed by a LOAD to the same address, if this piece of code is inside of a loop, what limits the first STORE in a second iteration to be issued before the LOAD of the first iteration (RAW memory hazard)?

Eskerrik asko!