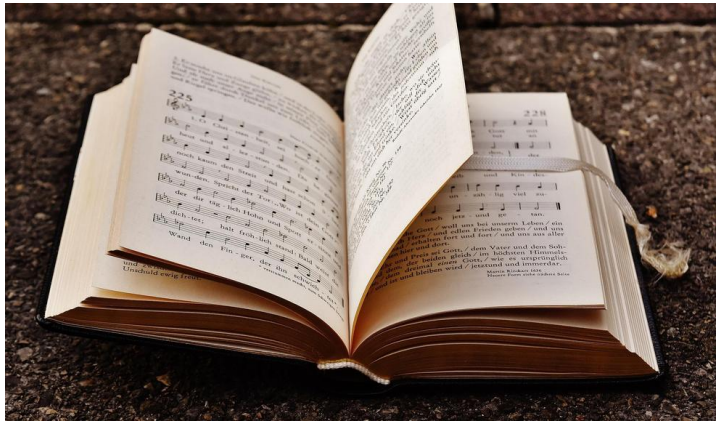
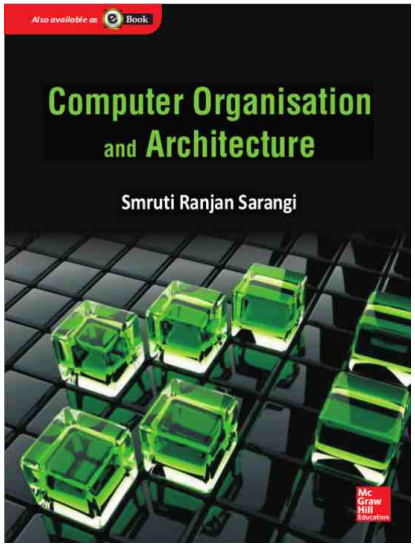


Chapter 4:
e Issue, Execute, and Commit Stages

Background Required to Understand this Chapter

- In-Order Pipelines
- Precise Exceptions
- OOO Pipelines: Basic idea



http://www.cse.iitd.ac.in/~srsarangi/archbooksoft.html

Contents

1. Instruction Renaming
2. Instruction Dispatch, Wakeup, Select
3. The Load-Store Queue (LSQ)
4. Instruction Commit

Renaming (architectural registers ⇨ physical registers)

Consider a 4-issue processor

- 4 instructions need to be renamed each cycle
- 4 instructions have 8 read operands, and 4 write operands
- Each read operand, needs to read its value from a physical register
- Each write operand needs to be assigned a new physical register
- Dependences between the instructions need to be taken care of

There are two main approaches

- Renaming with a physical register file
- Renaming with reservation stations

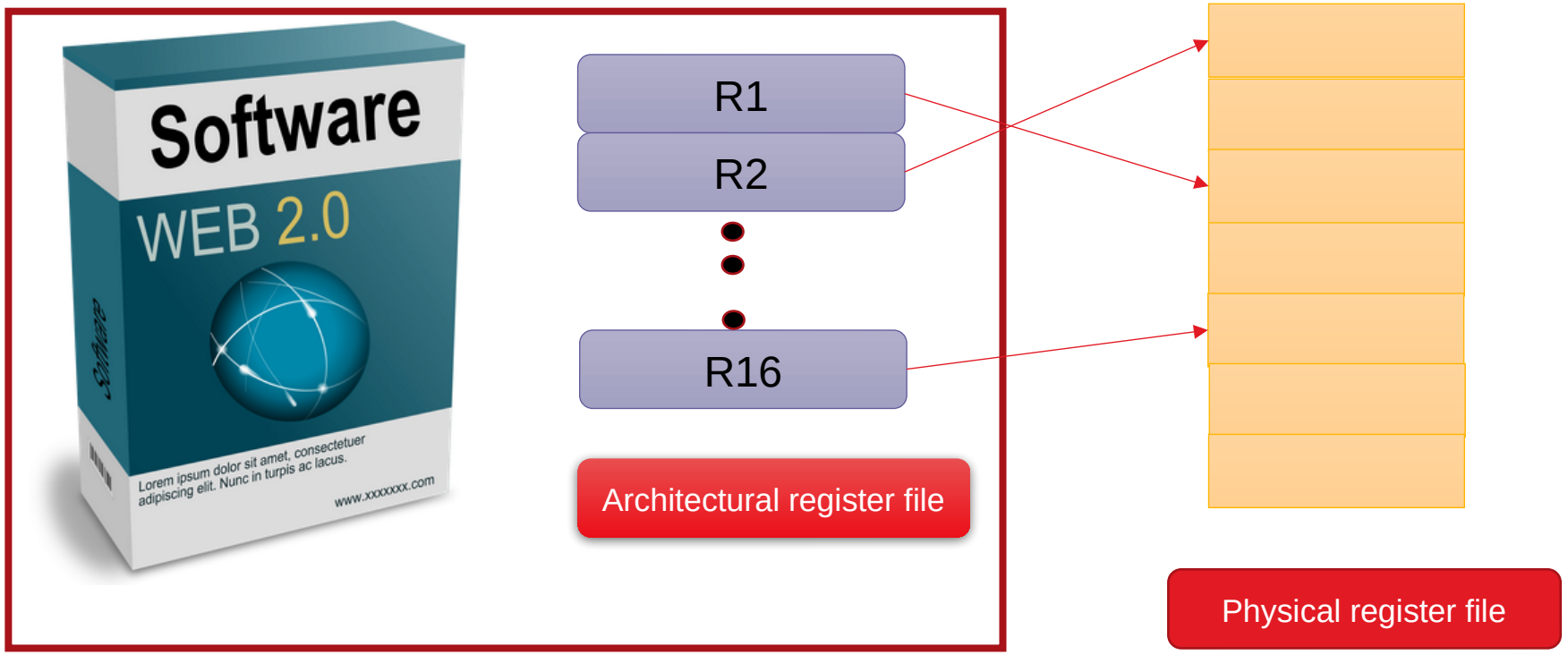
Renaming with a physical register file

Every ISA has a set of registers that are visible to software

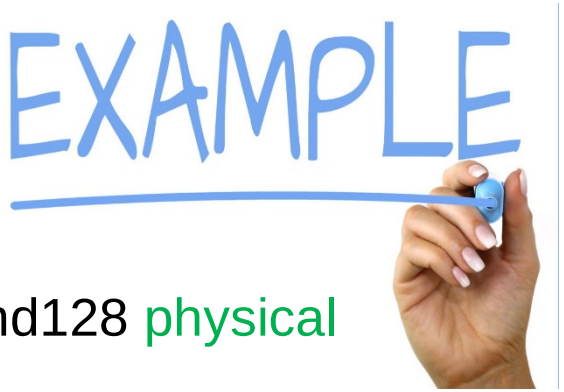
- These are called architectural registers
- x86 has 8
- x86-64 has 16
- ARM has 16
- MIPS has 32

- To ensure precise exceptions, the architectural registers should appear to be updated **in-order**
- However, inside the processor we need to do renaming to eliminate output and anti dependences

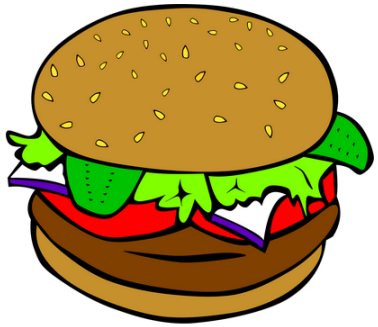
Physical Register File and the Process of Renaming



Physical Register File



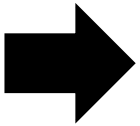
- Consider a system with16 architectural registers and128 physical registers
- In this case, architectural registers exist in principle. There is no separate architectural register file.
- Each architectural register is mapped to one and only one physical register at any point in time.
- 112 physical registers are unmapped. They either are empty or contain a value that will be overwritten by a later instruction



Why are 112 registers unmapped?

Example

```
mov r1, 1
add  r1, r2, r3
add  r4, r1, 1
mov r2, 5
add r6, r2, r8
mov r1, 8
add r9, r1, r2
```



```
mov p11, 1
add  p12, p2, p3
add  p41, p12, 1
mov p21, 5
add p61, p21, p8
mov p13, 8
add p91, p13, p21
```

- *r* series ⊏ architecture, *p* series ⊏ physical
- At the beginning *ri* is mapped to *pi*
- With time, the mapping changes
- *p12* contains the value of *r1* till the value is overwritten. *p13* contains the new value. At this point, *p12* contains an interim value

Example - II

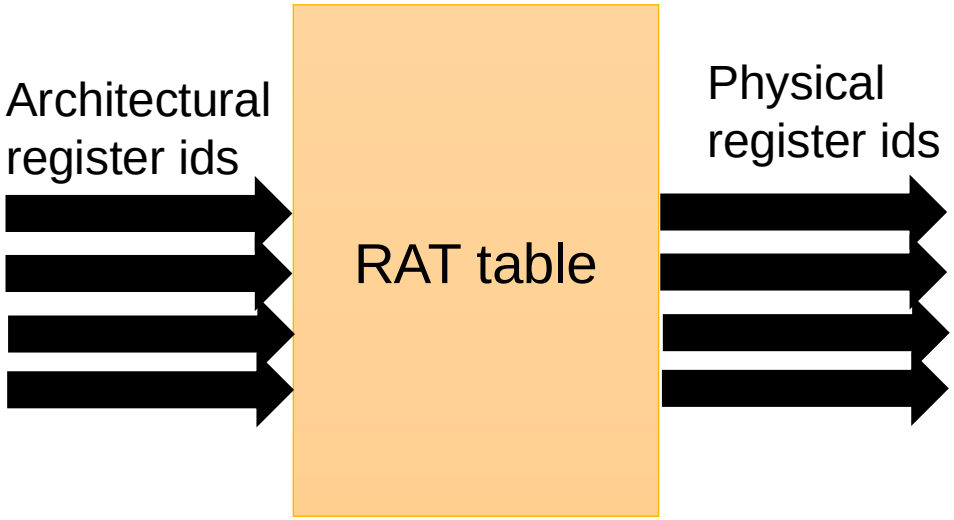
- We need 16 physical registers to contain the latest values of architectural registers
- 112 physical registers can possibly contain interim values that might be required by instructions in the OOO pipeline (also called in-flight instructions like p12)
- More is the number of physical registers ⇒ higher is the number of instructions that can simultaneously be in flight ⇒ more is the ILP
- Renaming: convert architectural register id ⇒ physical register id

Rename Stage

Three Structures

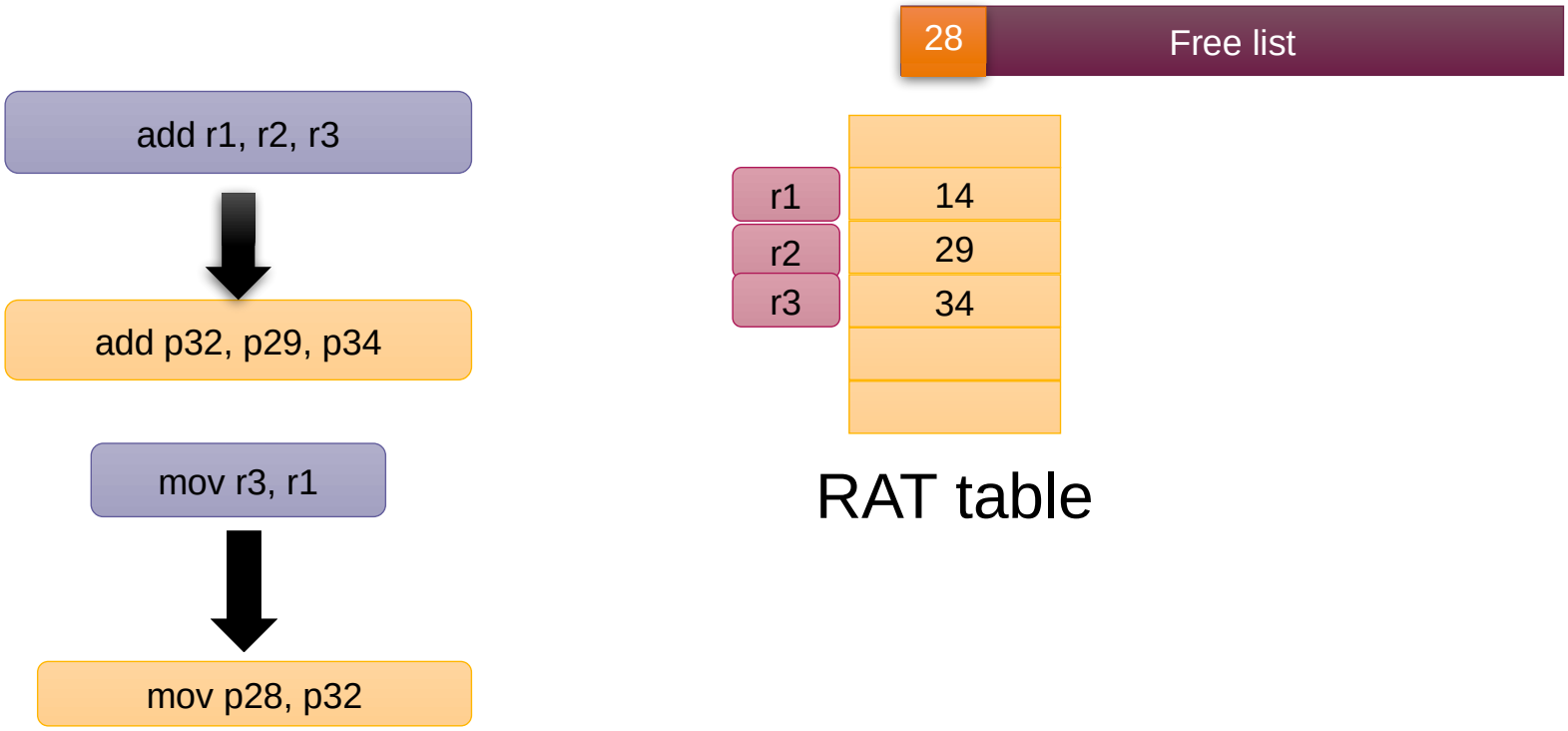
- Register Alias Table (RAT) ⎵ Translate architectural register id to physical register id
- Dependence Check Logic (DCL) ⎵ Take care of dependences between instructions renamed in the same cycle
- Free List or Free Queue ⎵ Maintain a list of unmapped physical registers

Register Alias Table



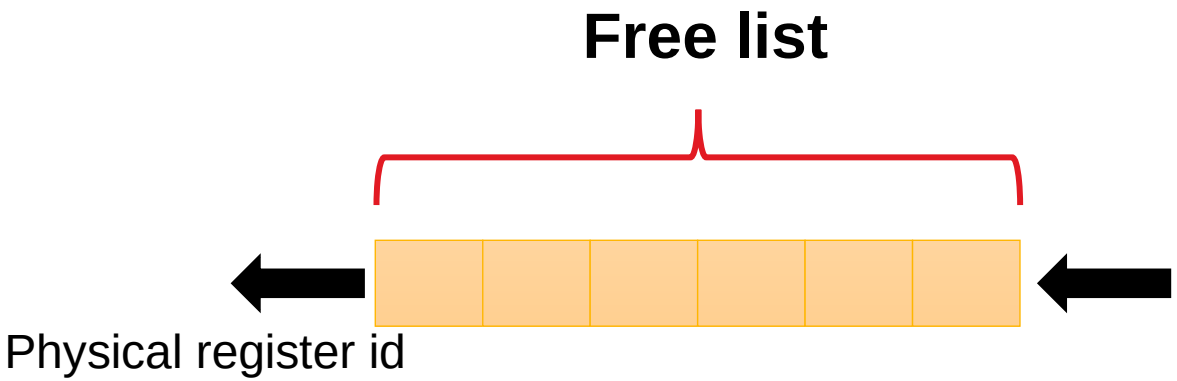
- Rename multiple instructions at once
- We get a stream of physical register ids
  - Eliminates WAW and WAR hazards

Operation of the RAT Table



Free List

- It is a queue of unmapped physical register file ids
- An entry is dequeued from it when we need to assign it to an architectural register id
- When is an entry added to it? **LATER**
- **Update the RAT with the new mapping**

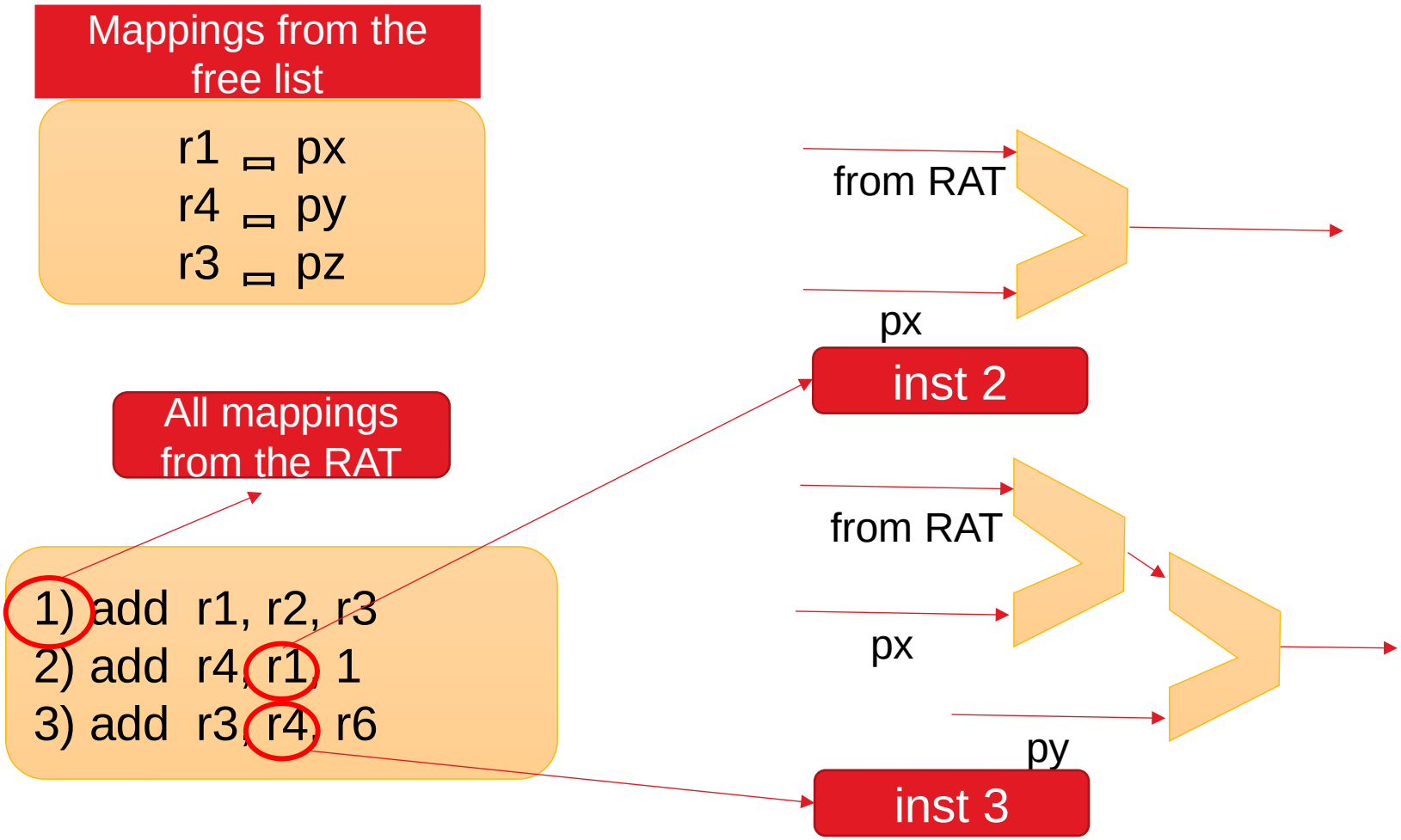


Dependence Check Logic

```
add  r1, r2, r3
add  r4, r1, 1
```

- Let us say *r1* is mapped to *p11*
- The second instruction needs to read the mapping of *r1* as *p11*
- What if both instructions are sent to the RAT together?

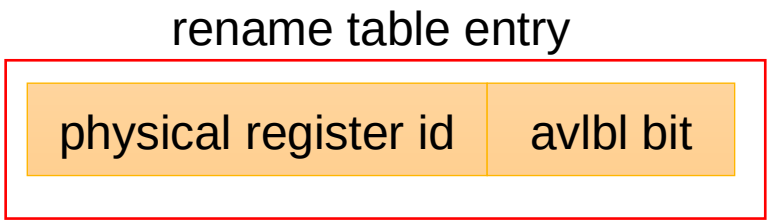
Dependence Check Logic - II



Available bit in the rename table entry

We need some more information in each entry

- An available bit (*avlbl)*
- It indicates whether the result can be found in the physical register file and the rest of the pipeline or not.
  - If not, the instruction has to **wait** in the instruction window.



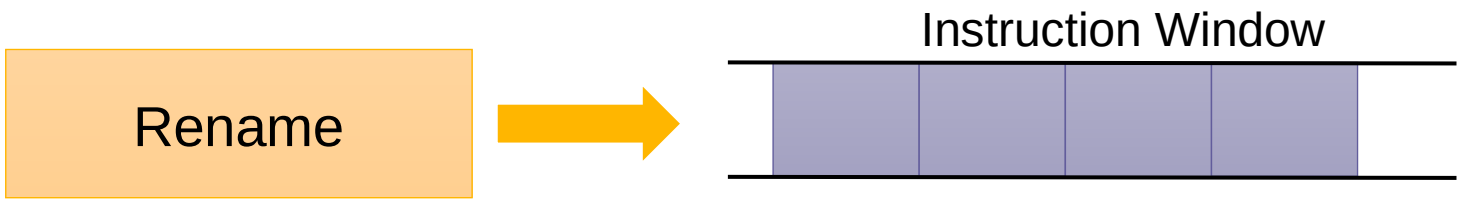
Contents

1. Instruction Renaming
2. Instruction Dispatch, Wakeup, Select
3. The Load-Store Queue (LSQ)
4. Instruction Commit

Dispatch

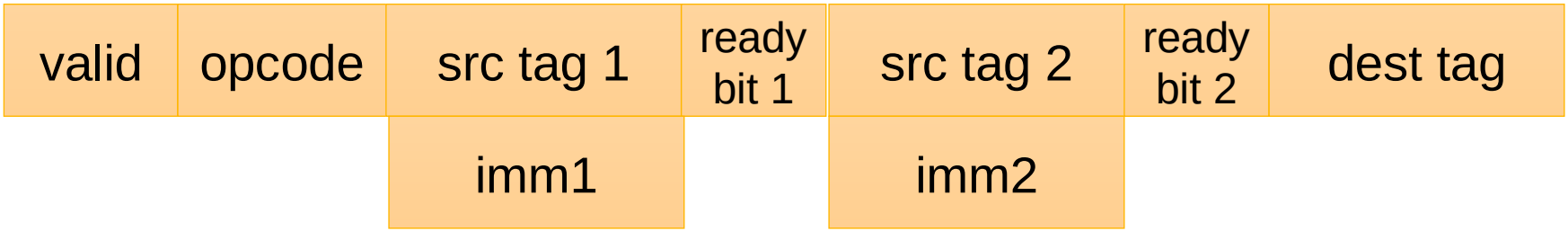
Till the rename stage, instructions proceed in-order.

Dispatch ⎵ Send the instruction to the instruction window (IW)



Instruction Window

Status of an instruction in the window



opcode ▭ type of operation: add, subtract, multiply

tag ▭ physical register id

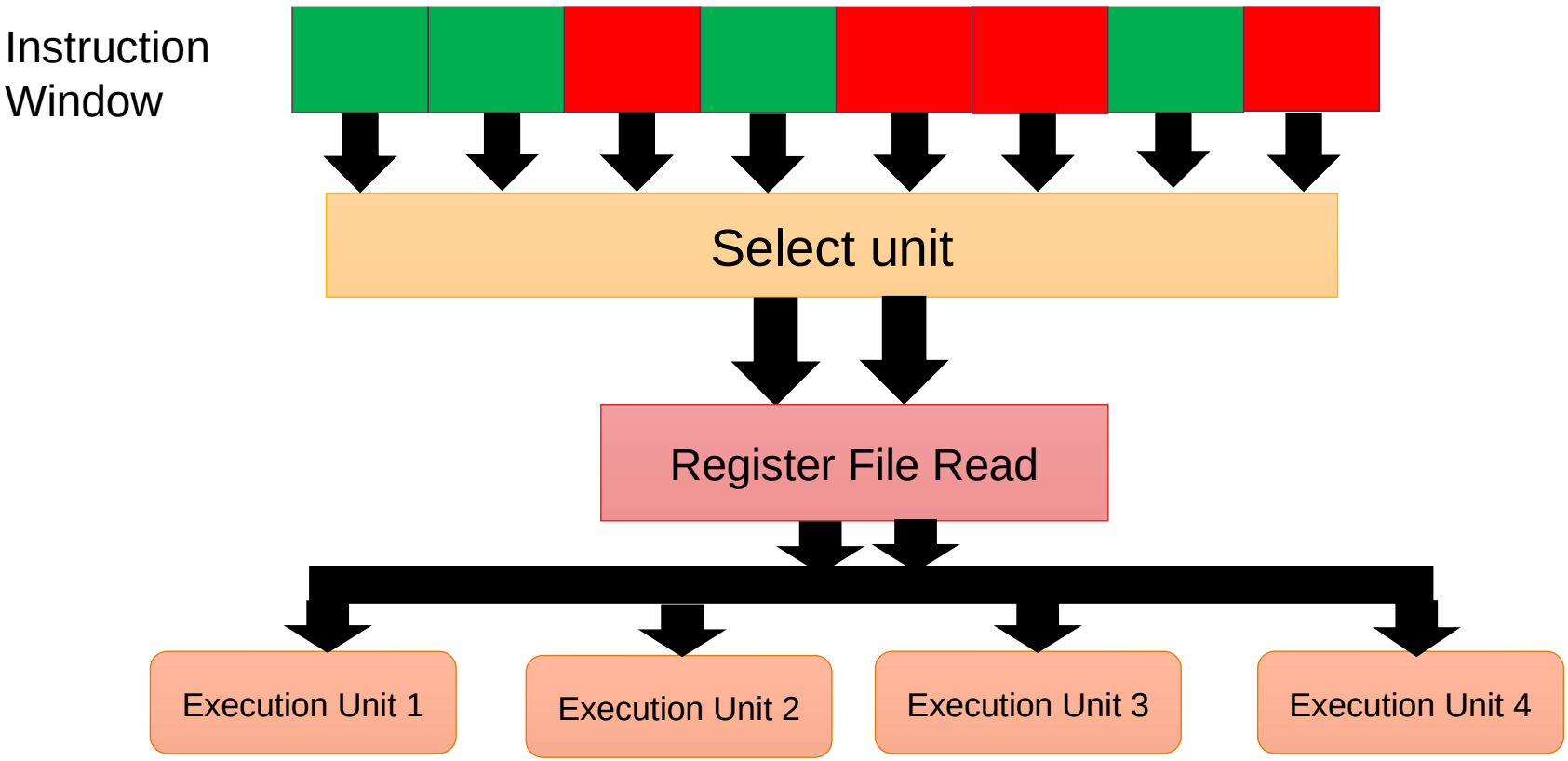
ready bit ▭ does the physical register file contain the corresponding value

At any point in the instruction window

Instructions waiting for an operand to be ready

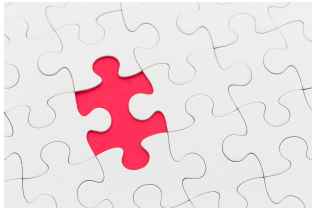
Instructions whose operands are ready

Select and Execute



Wakeup

- How does an instruction in the IW know that its operands are ready?

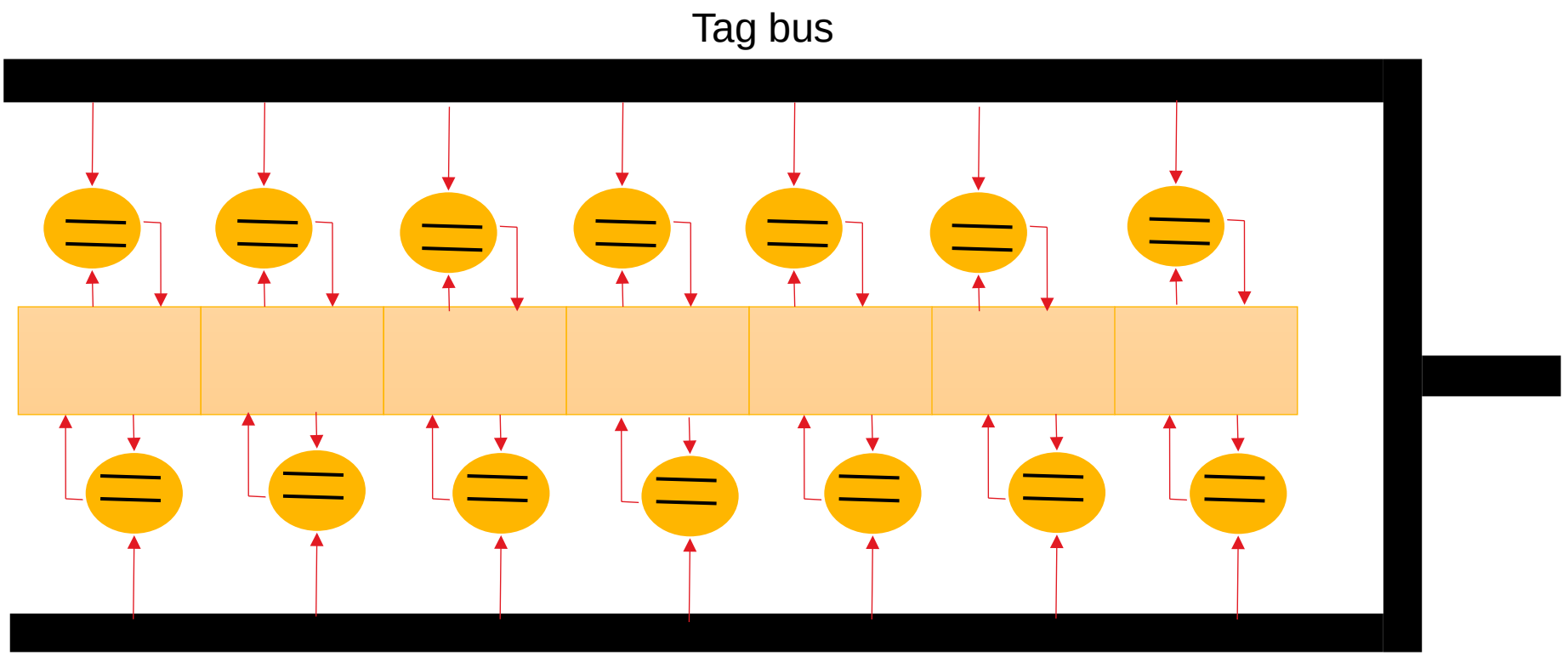


The producer instructions need to let it know

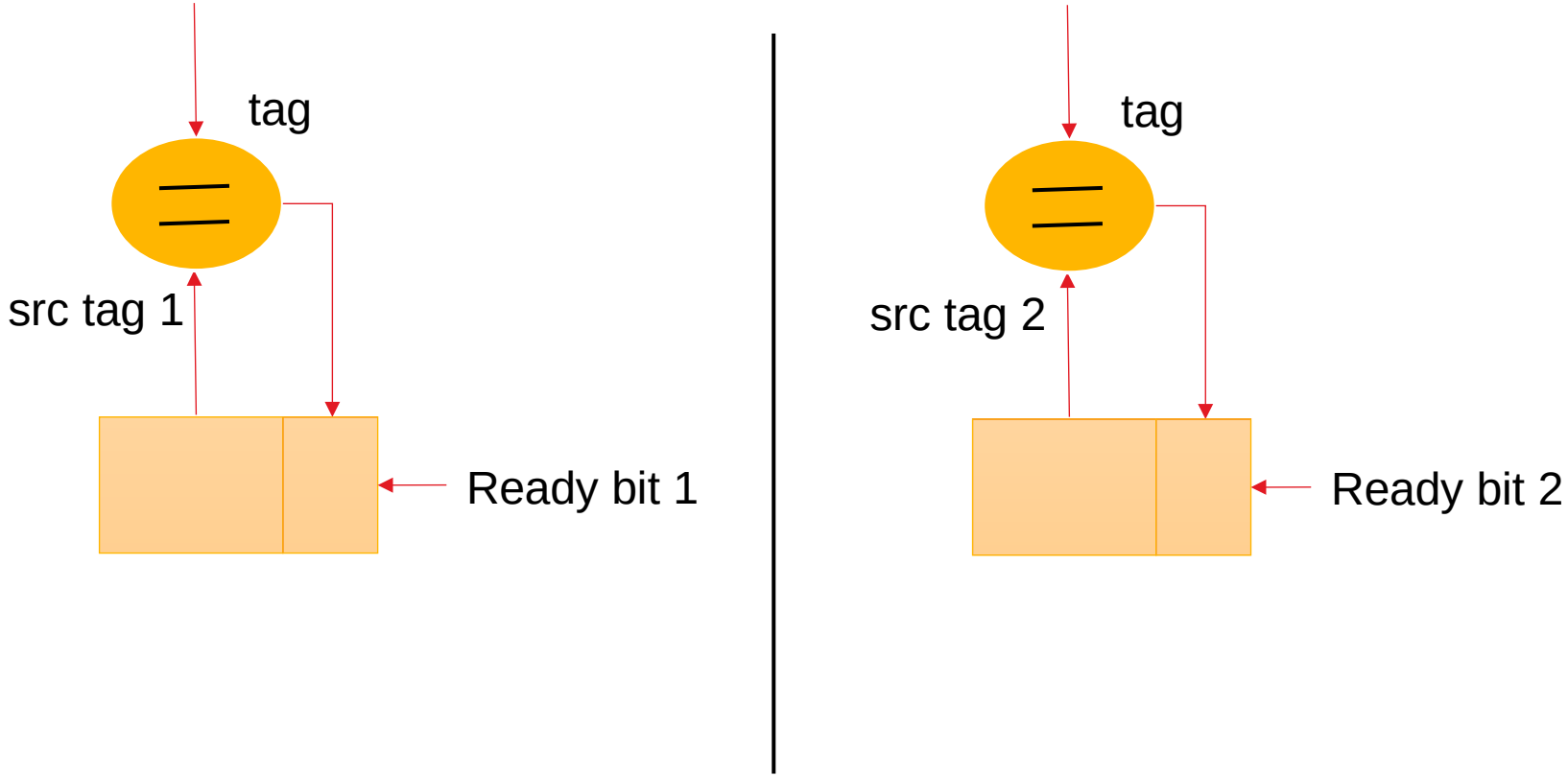
Once a producer finishes executing

- Broadcasts the tag (destination physical register id) to the entries of the IW
- Each entry marks its corresponding source operand as ready if the tag matches
- This is called **wakeup**

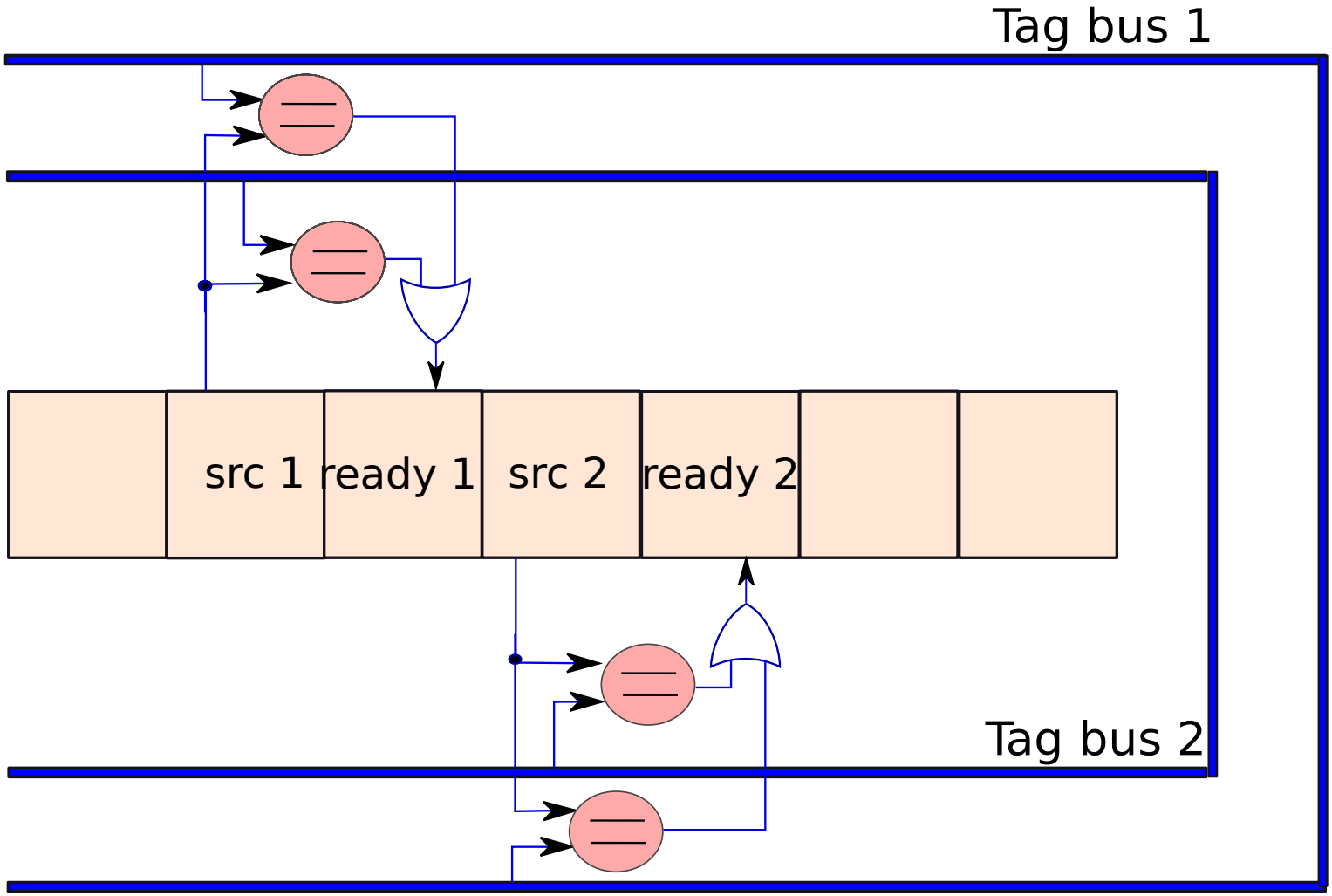
Instruction Window



Instruction Window - II

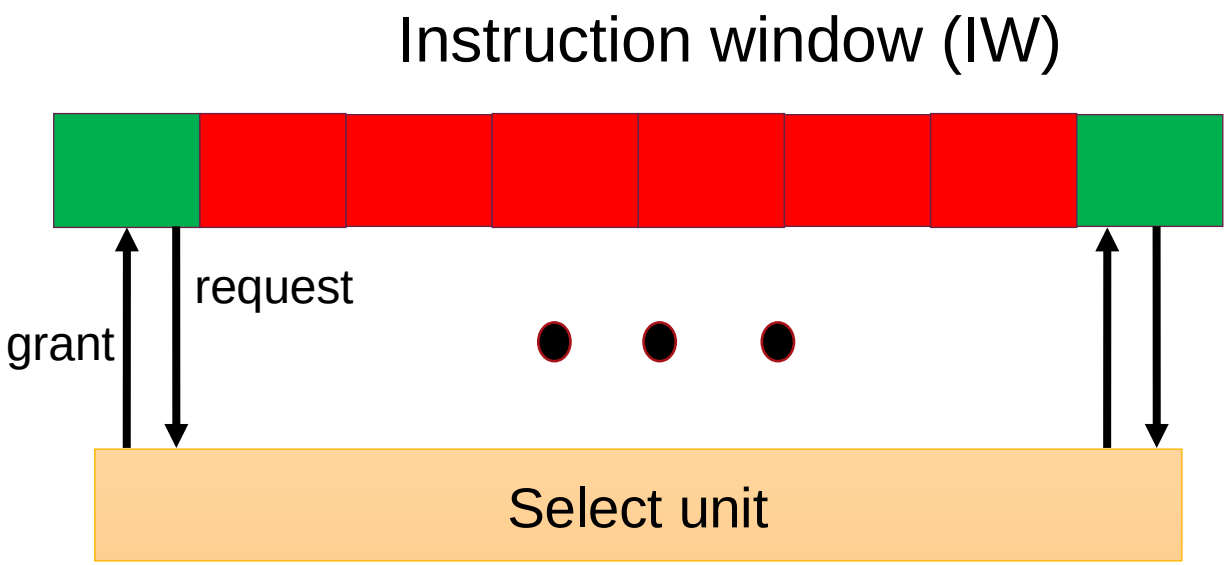


Broadcast multiple tags at once



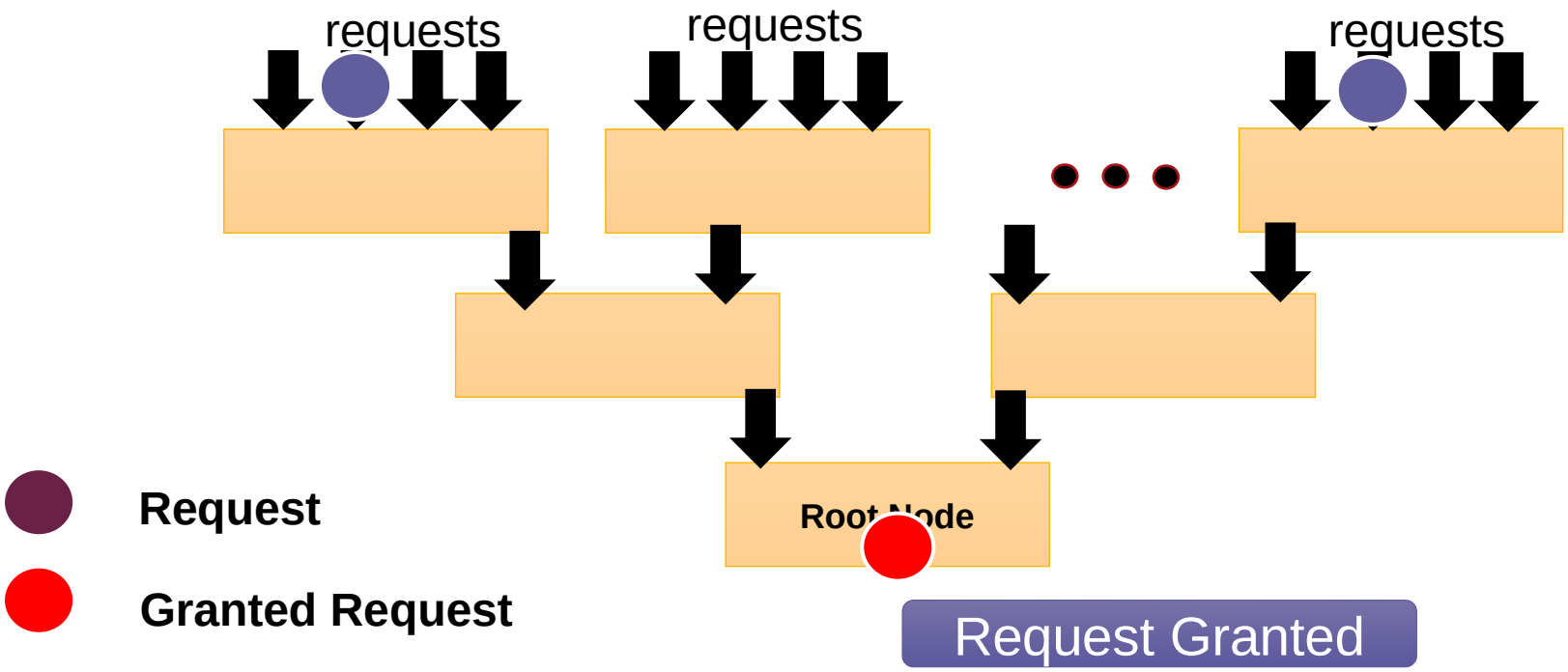
Instruction Select

Select Unit



- Once an IW entry has all its operands ready, it is marked as ready.
- It asserts its attached grant line
- The select unit chooses one among the ready instructions

Tree Structured Select Unit: Each node is a choice box

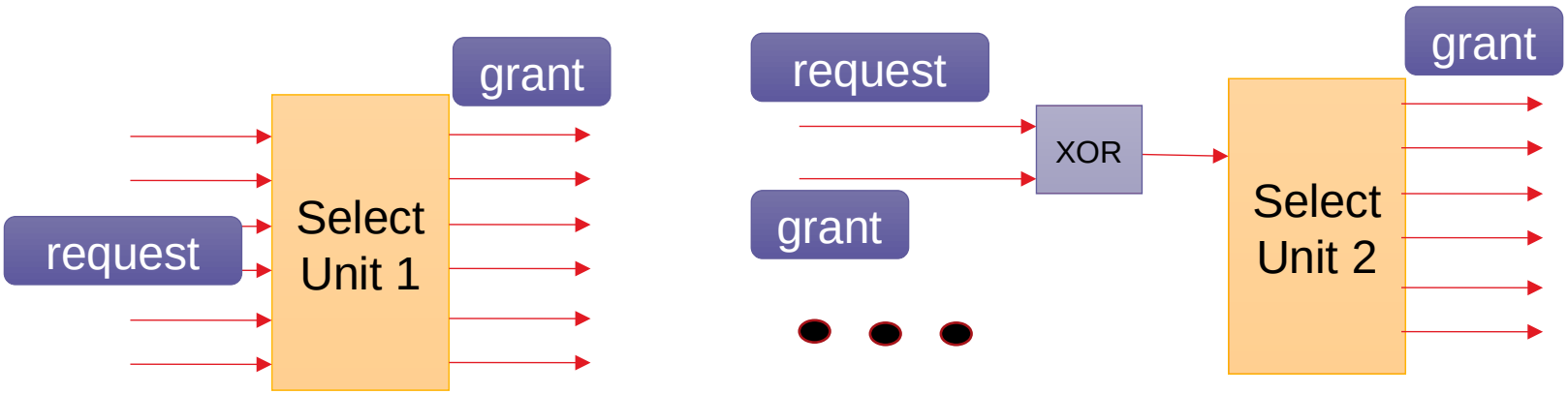


Select Unit for Multiple Resources

What if we have 2 adders, 2 multipliers, and 1 divider?

Solution:

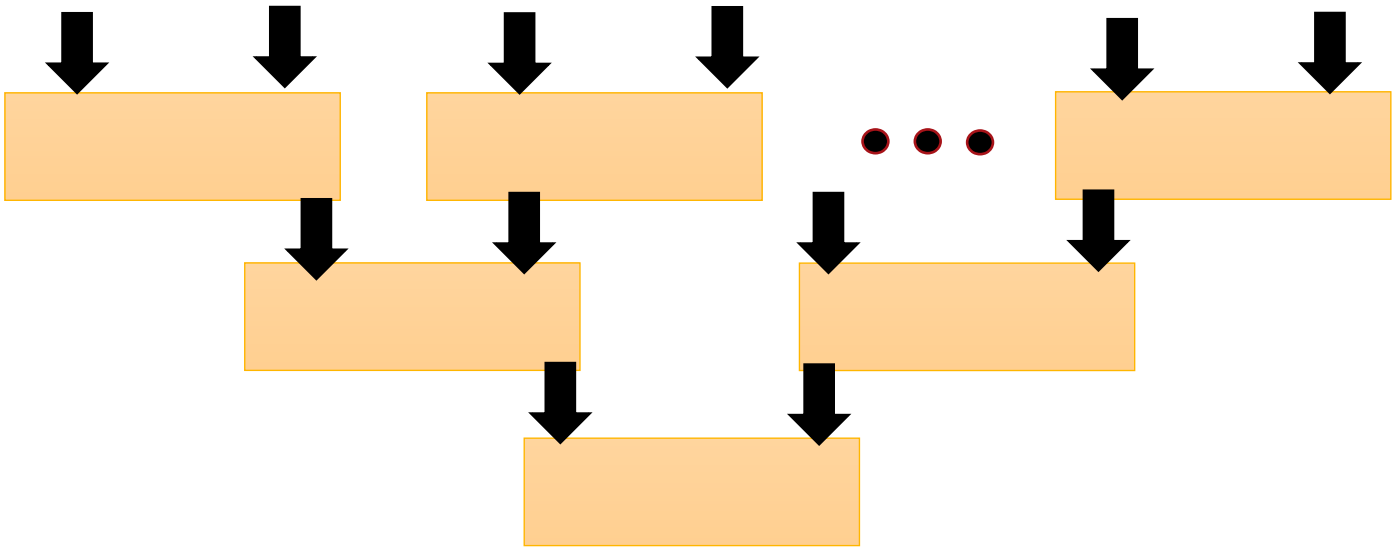
- Have a separate select unit for each class of instructions
- For a select unit that caters to multiple FUs (functional units)
- Three options: Option 1 ⎵ chain select units



Option 2: Multiple request signals

- Send multiple request signals (up to *n*)
- Every level sends at most *n* requests to the root
- The root selects at most *n* requests, and issues grant signals
- This circuit is more complicated than serial select circuits
  - The choice boxes need more elaborate logic

Option 3: Asymmetric Select Units



- Have two select units with different policies.
- Select unit 1:
  - Each choice box gives priority to its left input
- Select unit 2:
  - Each choice box gives priority to its right input

Select Policies

The select policy has an important implication on performance. We should select those instructions that are the most performance critical.

- Random
- Oldest First

Type of opcode

- Load instructions typically have higher priority than stores.
- Other priorities can also be learnt based on execution patterns.

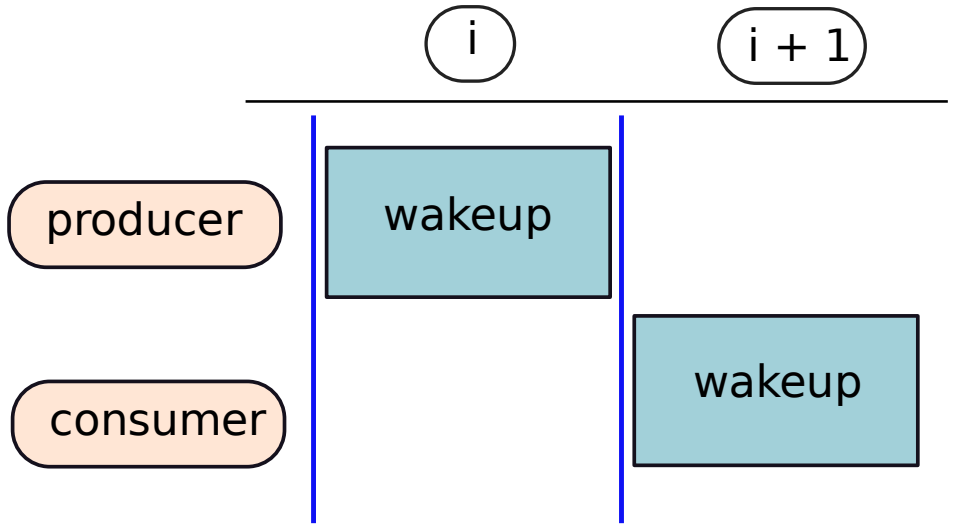
Tag Broadcast

RAW Dependences in our Pipeline

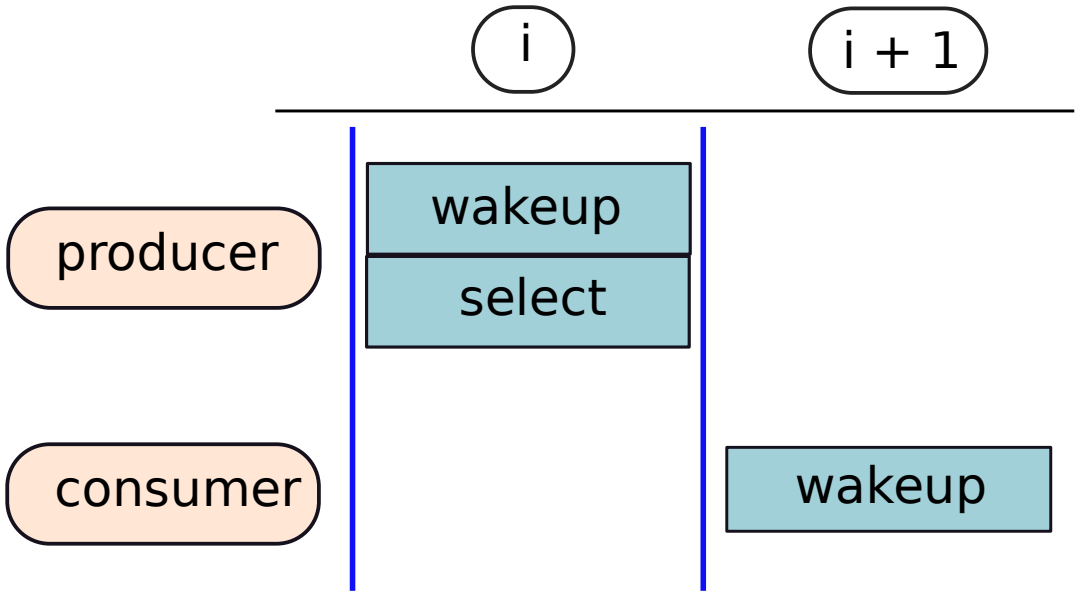
Consider consecutive instructions with RAW dependences

1) add r1, r2, r3
2) add r4, r1, r5

We want back-to-back execution – Execution in consecutive cycles.

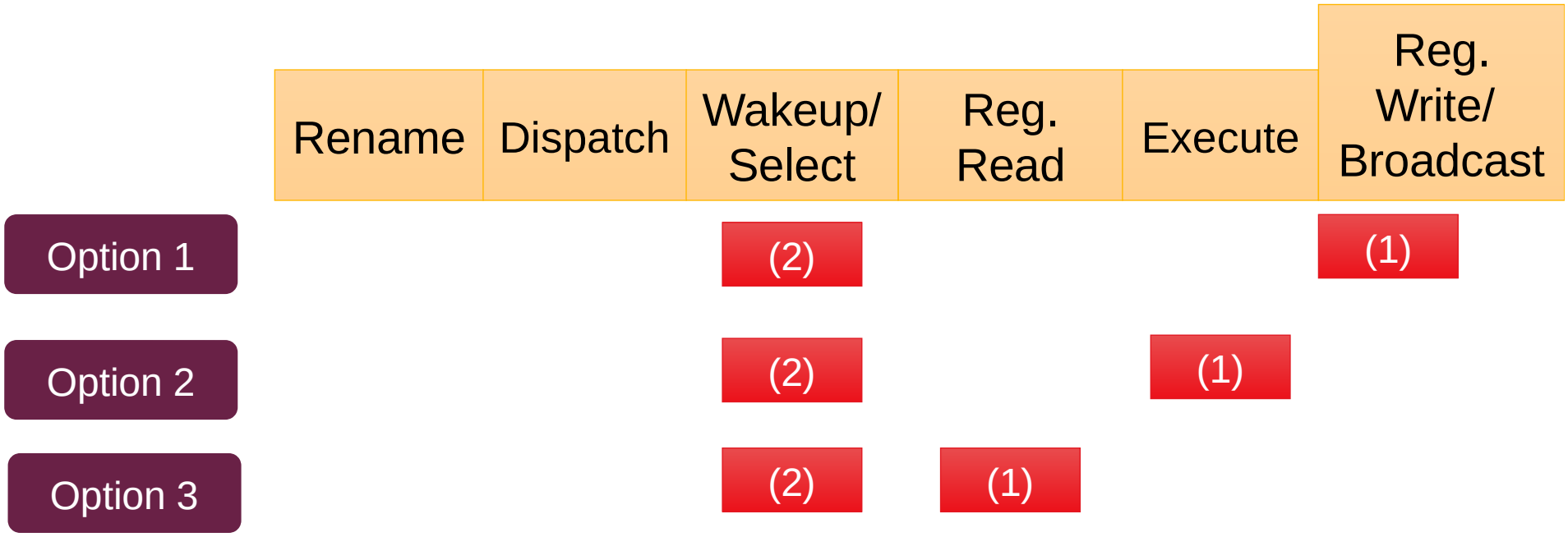


What does this imply?



Assume we have slightly optimized our pipeline to have wakeup and select in the same cycle.

When should we broadcast the tag?



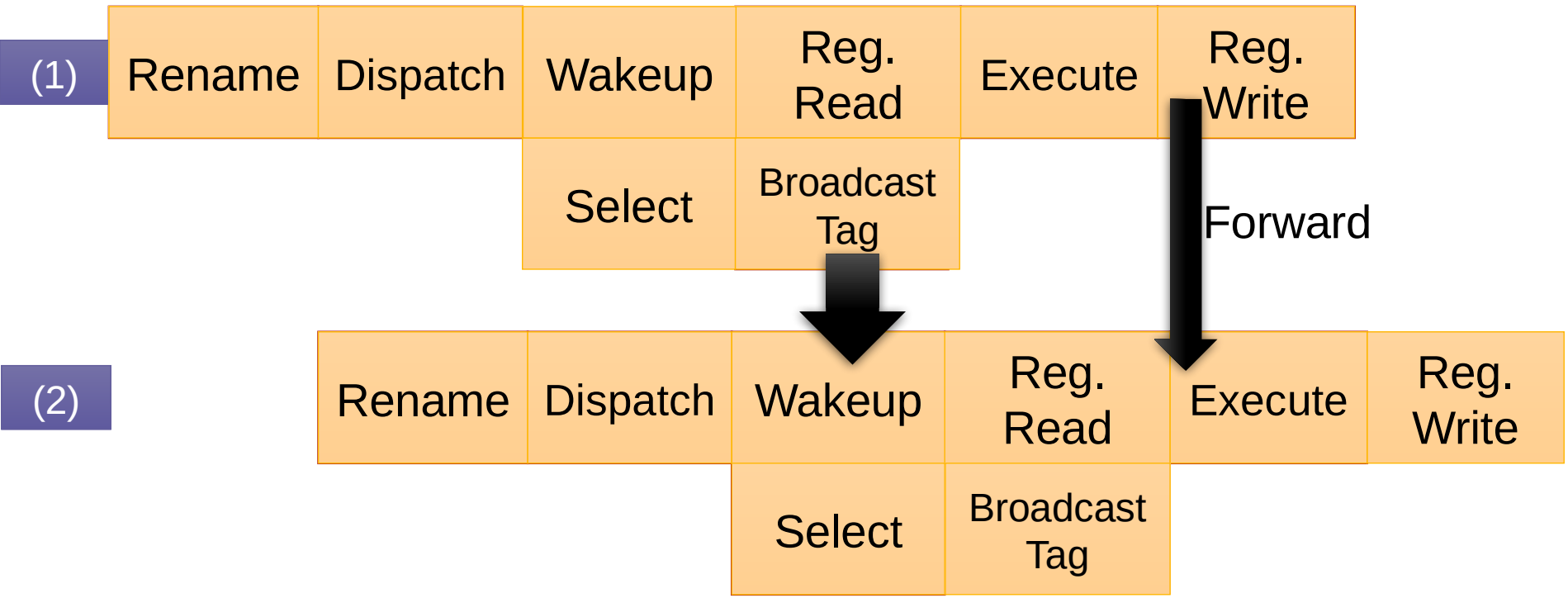
Let us consider Option 3 – Early Broadcast

- Allows back-to-back (in consecutive cycles) execution

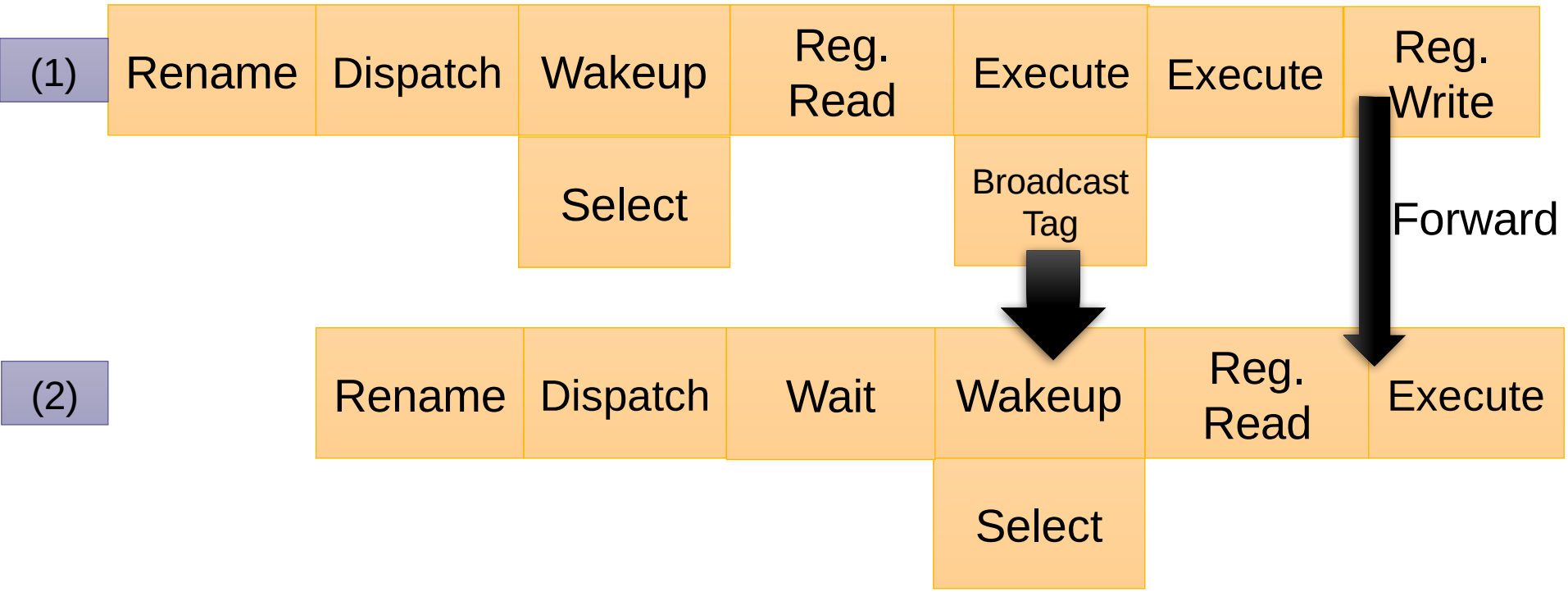
What needs to be done:

- When (1) is in the Reg. read stage, (2) needs to be in the select stage
- This is possible if we broadcast the tag for (1) when it is in the reg-read stage
- (2) picks up the tag, wakes up, and gets selected for the next cycle

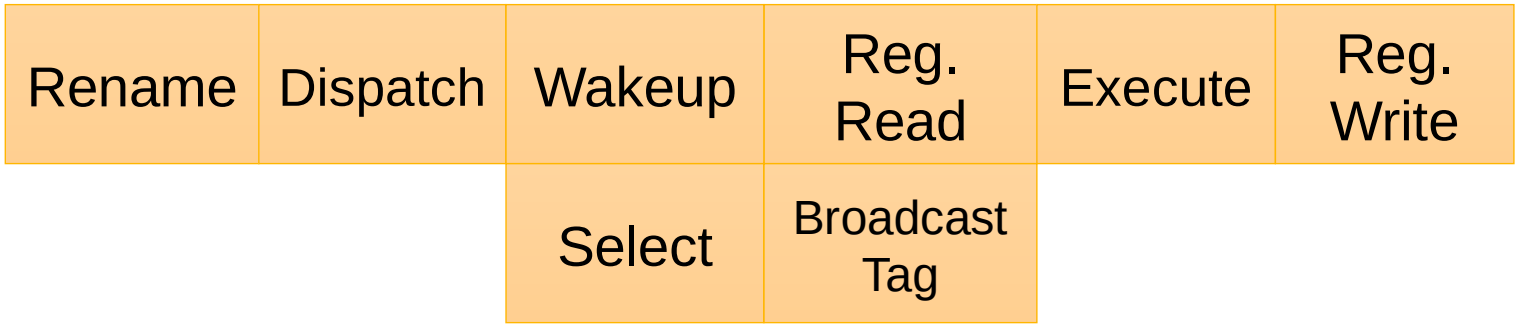
Classic Example of Forwarding in an OOO Pipeline



2-cycle latency for producer: Load-use Hazard



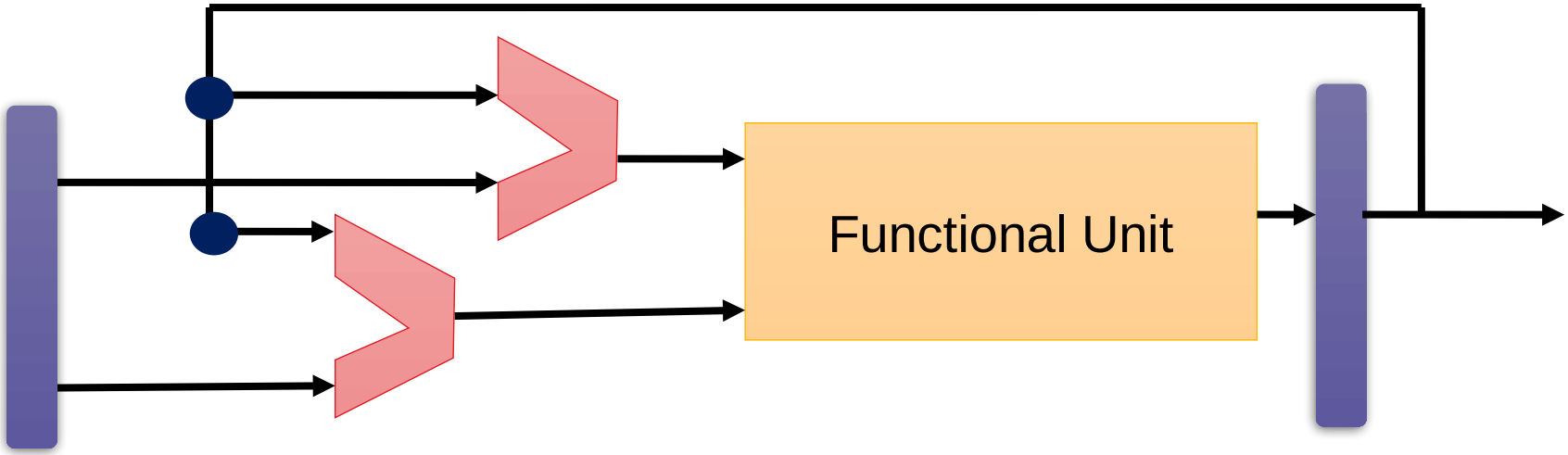
Structure of the Pipeline



- For different instructions, we broadcast the tags at different points of time.
- If the execution requires *k* cycles, we broadcast the tag *k* cycles after getting selected.

Forwarding Logic in the Execution Unit

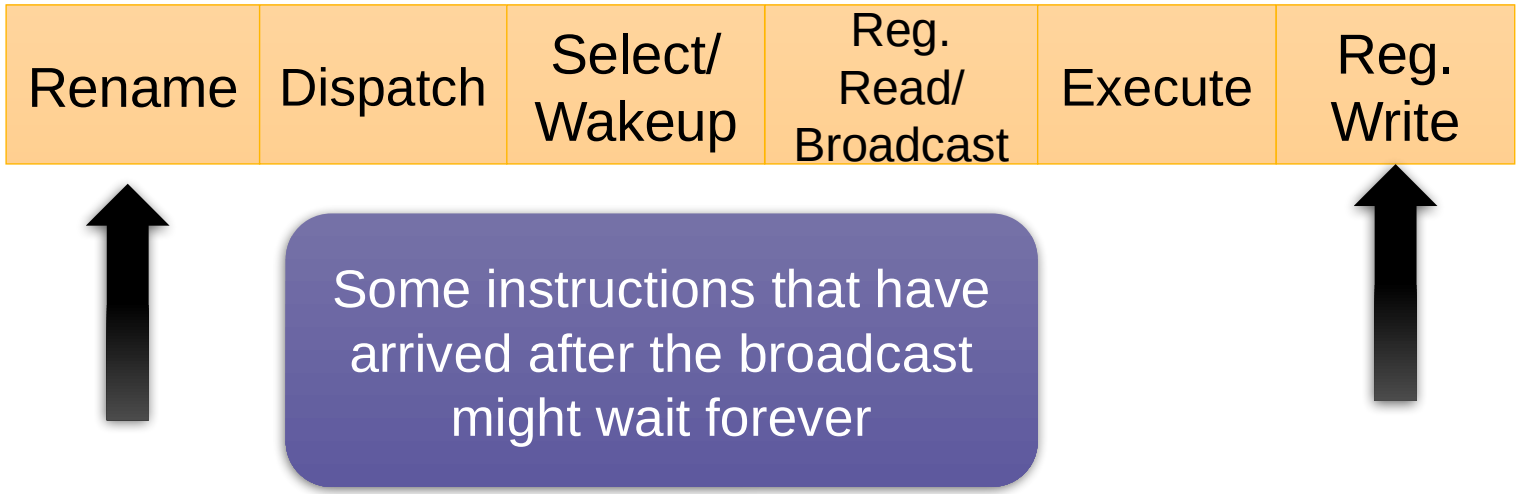
Also called bypass paths



from the
Register File

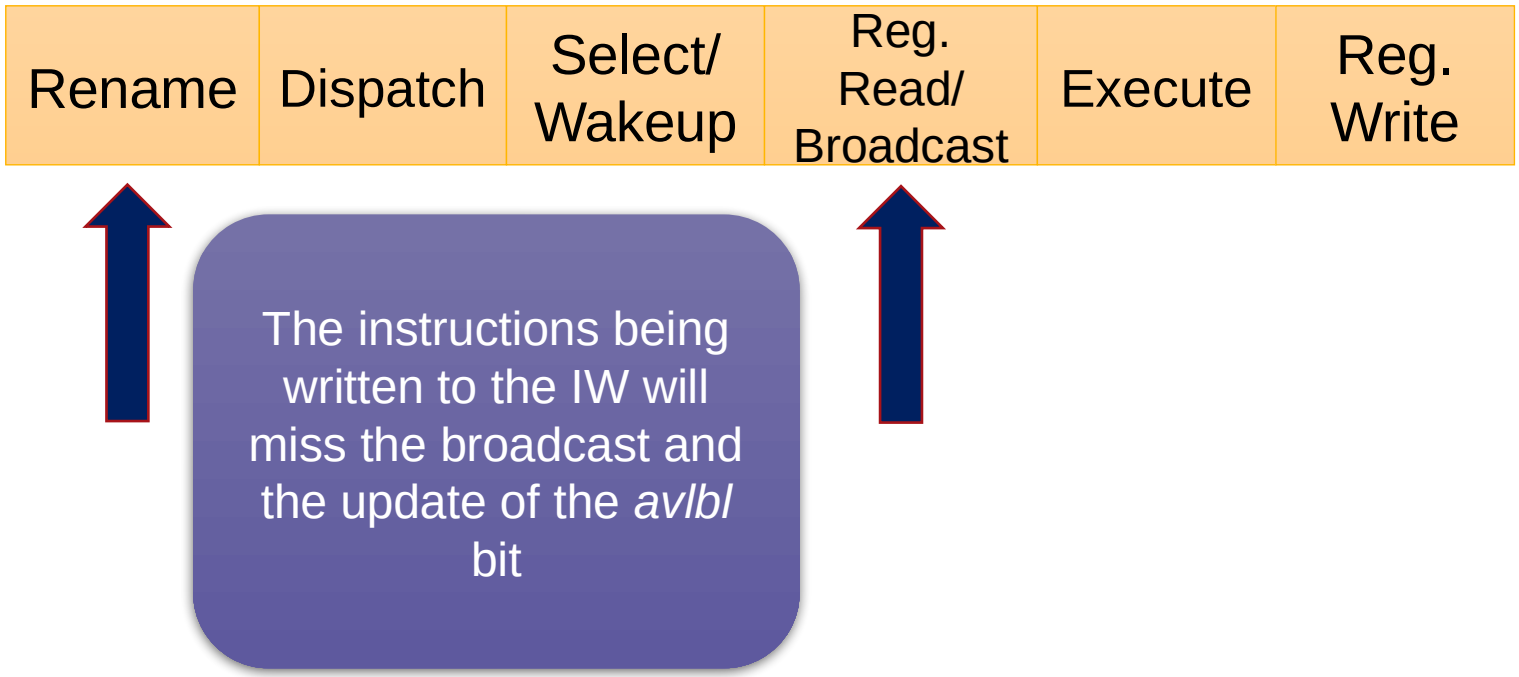
When should we mark the *avlbl* bit?

Option 1: Along with the register write

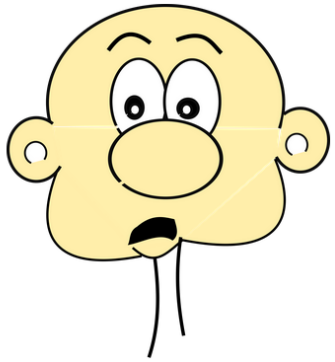


When should we mark the available bit?

Option 2: Along with the broadcast



What to do?



Let us fix solutions 1 and 2

Basic features of a new solution:

- Realization: **Instructions** being written to the IW will miss the broadcast
- These instructions would have also read the avlbl bit as 0
- As a result: they will wait forever

What does the teacher do if some students in the class are sleeping?

- ANSWER: Take one more class

What is the solution here:

- Double broadcast

Double Broadcast

Both solutions 1 and 2 can be fixed, if two broadcast messages are sent

Let us look at one solution

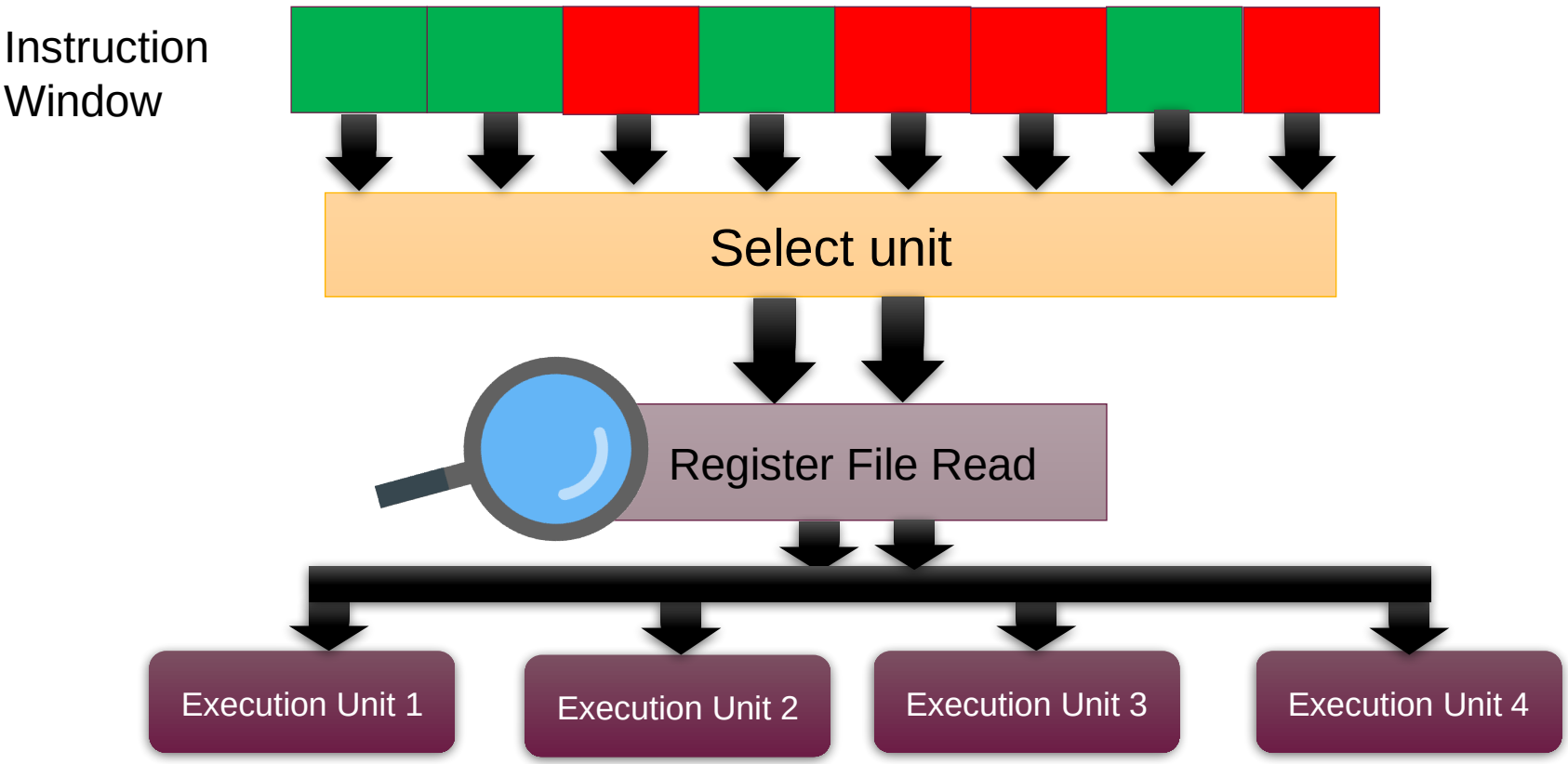
- Set the avlbl bit to 1 along with sending the Broadcast message
- There might be some instructions that miss the broadcast
- **Log** the Broadcasted tags in a temporary structure
- Also record all the instructions that are being dispatched in a separate structure called the *dispatch buffer*

In the next cycle

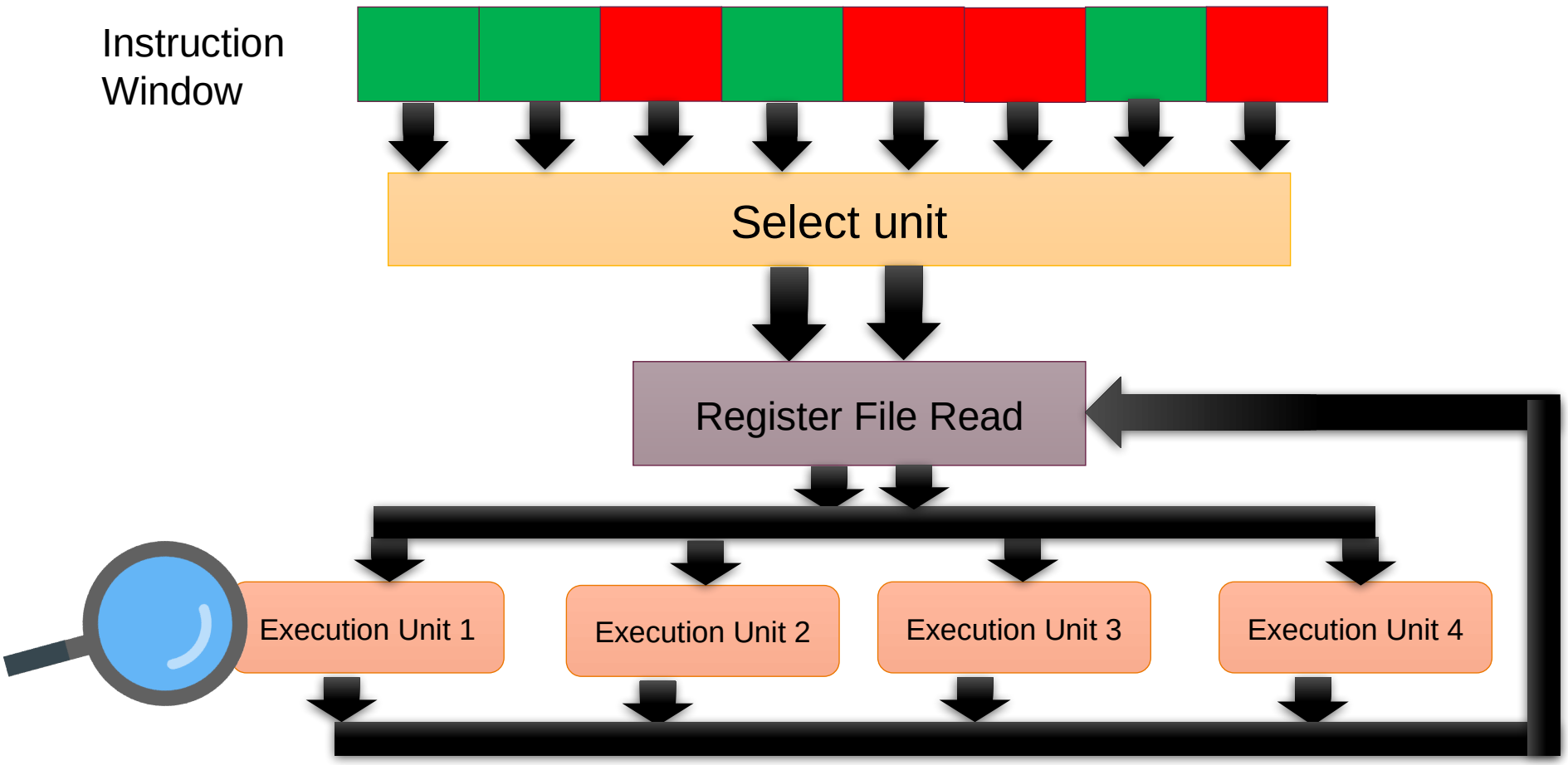
- Match the tags with entries in the dispatch buffer.
- If there is a match, update the corresponding IW entry
- Clear the entries for the last cycle in the *dispatch buffer*

Register Read and Execute

Read operands from the register file and Issue to the execution units



Execute


Instruction Window
Select unit
Register File Read
Execution Unit 1
Execution Unit 2
Execution Unit 3
Execution Unit 4

Contents

1. Instruction Renaming
2. Instruction Dispatch, Wakeup, Select
3. The Load-Store Queue (LSQ)
4. Instruction Commit

Case of Load and Store Instructions

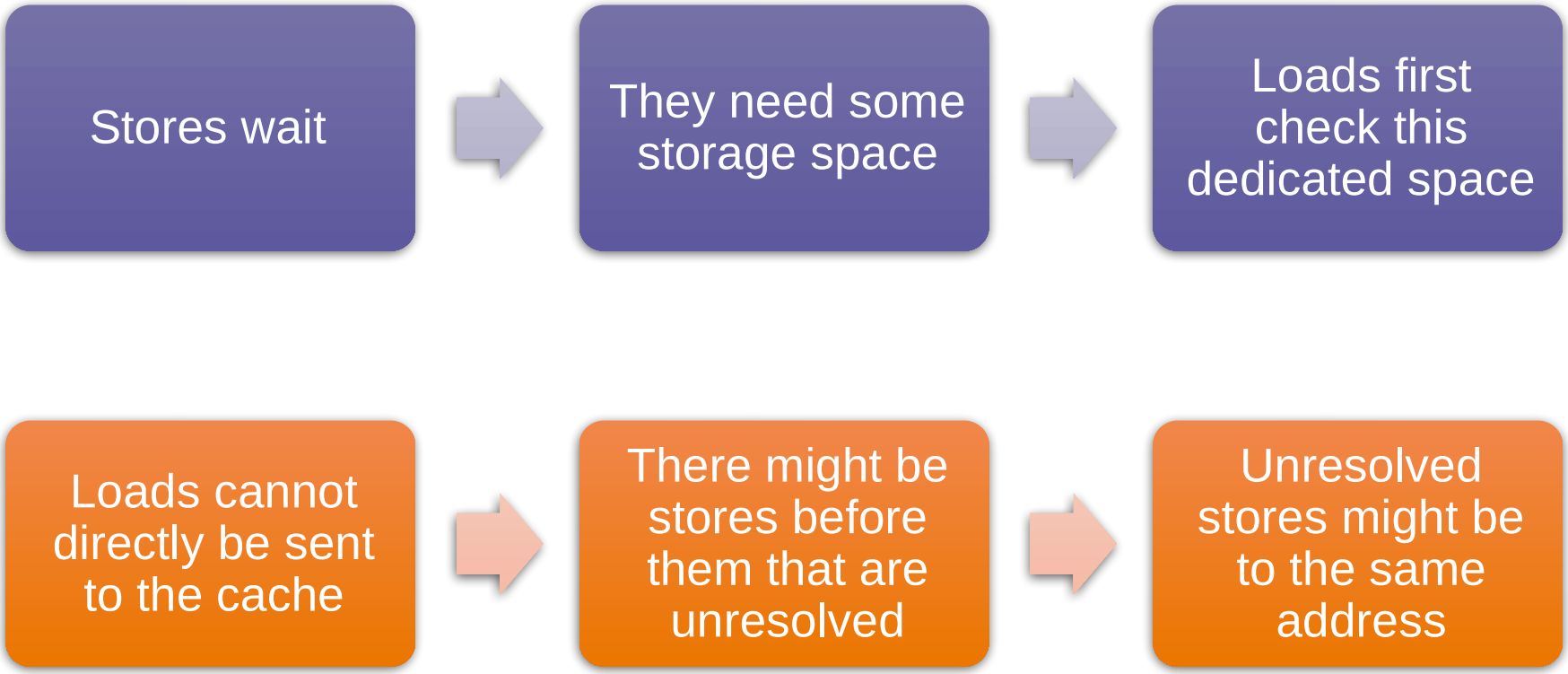
```
ld r1, 4[r3]
st r2, 10[r5]
```

First, the instructions are sent to the adder to compute the effective address. In this case: 4 + (contents of *r3*), 10 + (contents of *r5*)

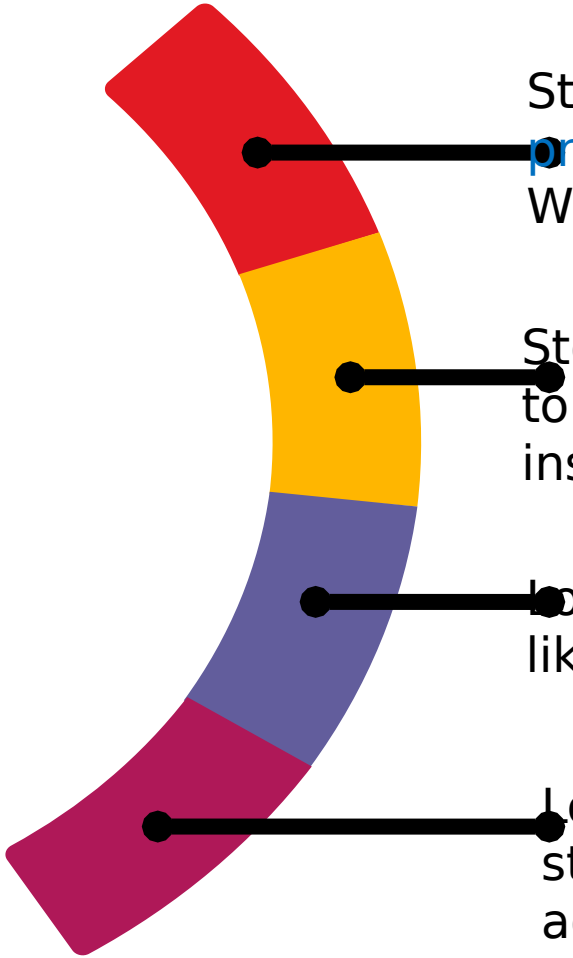
Next, they are sent to a load-store unit

- Sends the loads and stores to the data cache
- Loads can execute immediately
- Stores update the processor's state (remember precise exceptions)
- We cannot afford to write the value of stores on the wrong path
  - Hence, for stores we wait.

Requirements: Loads and Stores



Key requirements



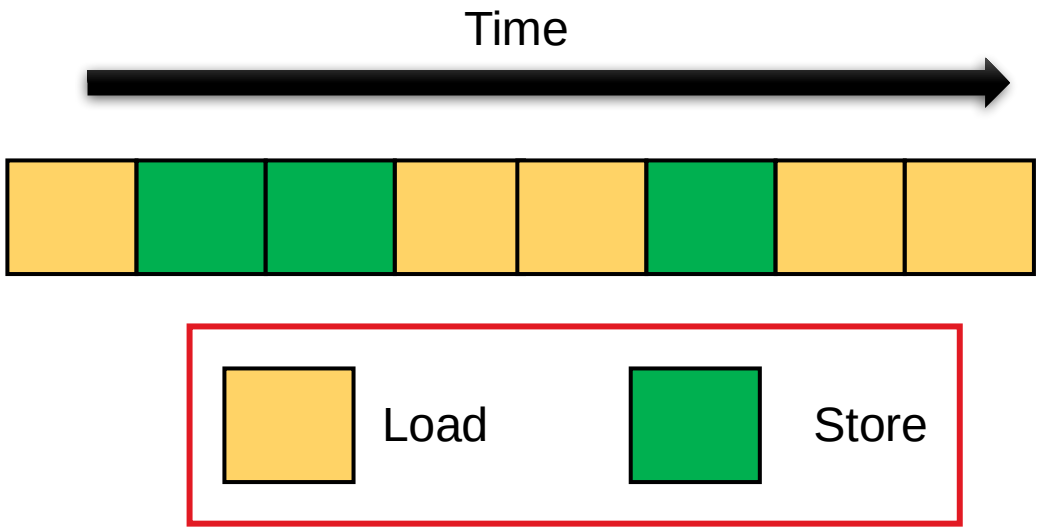
Stores have to be sent to the memory system in program order. This is needed for precise exceptions. We need dedicated storage space ▭ store queue

Stores can only be sent when they are guaranteed to be on the correct path. A store needs to be the earlie instruction in the pipeline when it is sent.

Loads first check the store queue. The store queue is like a cache.

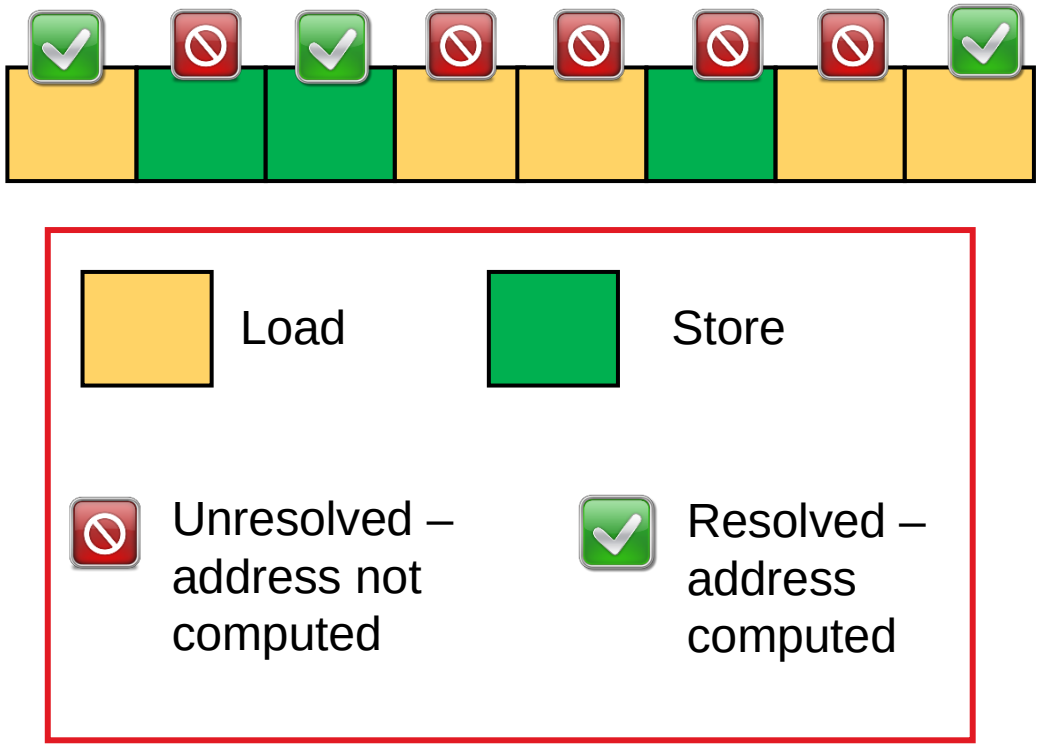
Loads cannot be sent to the memory system until all th stores before it are resolved and do not write to the sar address.

Load-Store Queue (LSQ)

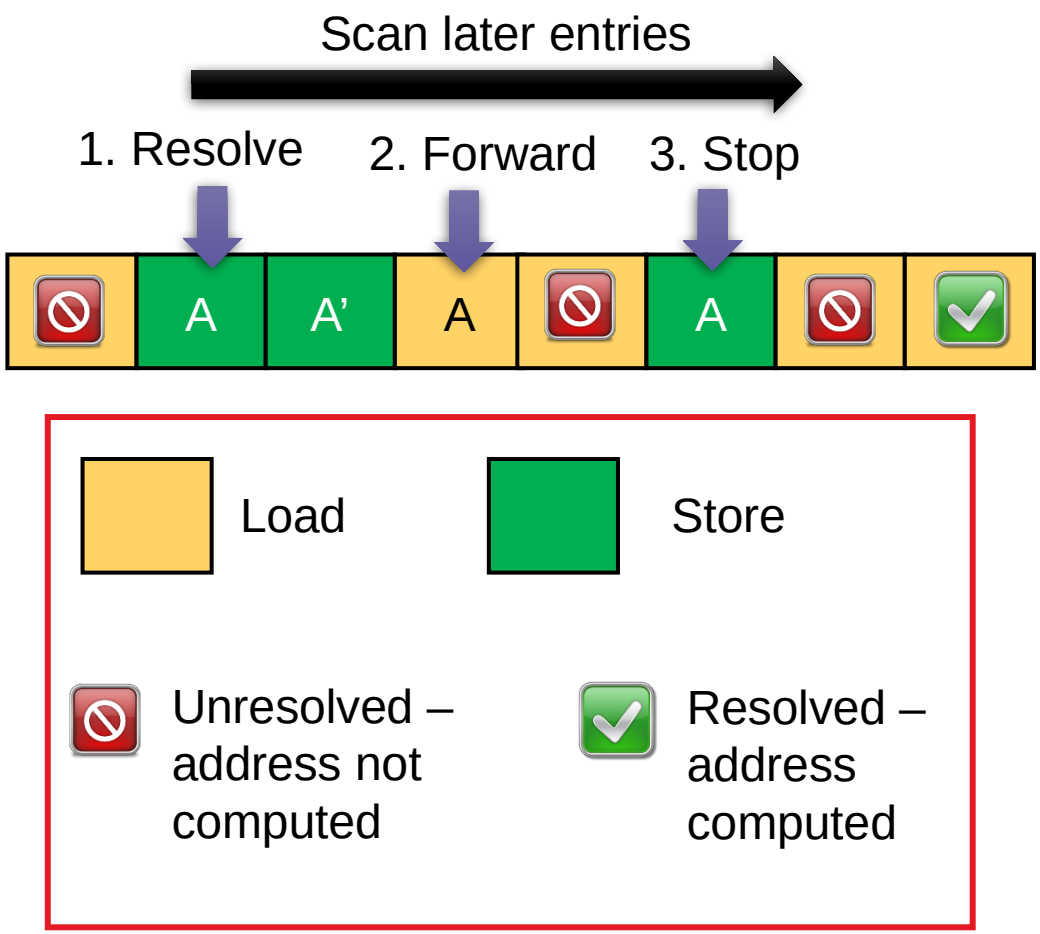


- Allocate an entry at decode time (allocated in order)
- Don’t ask when to deallocate an entry (ask later)
- When the effective address is computed, update the entry

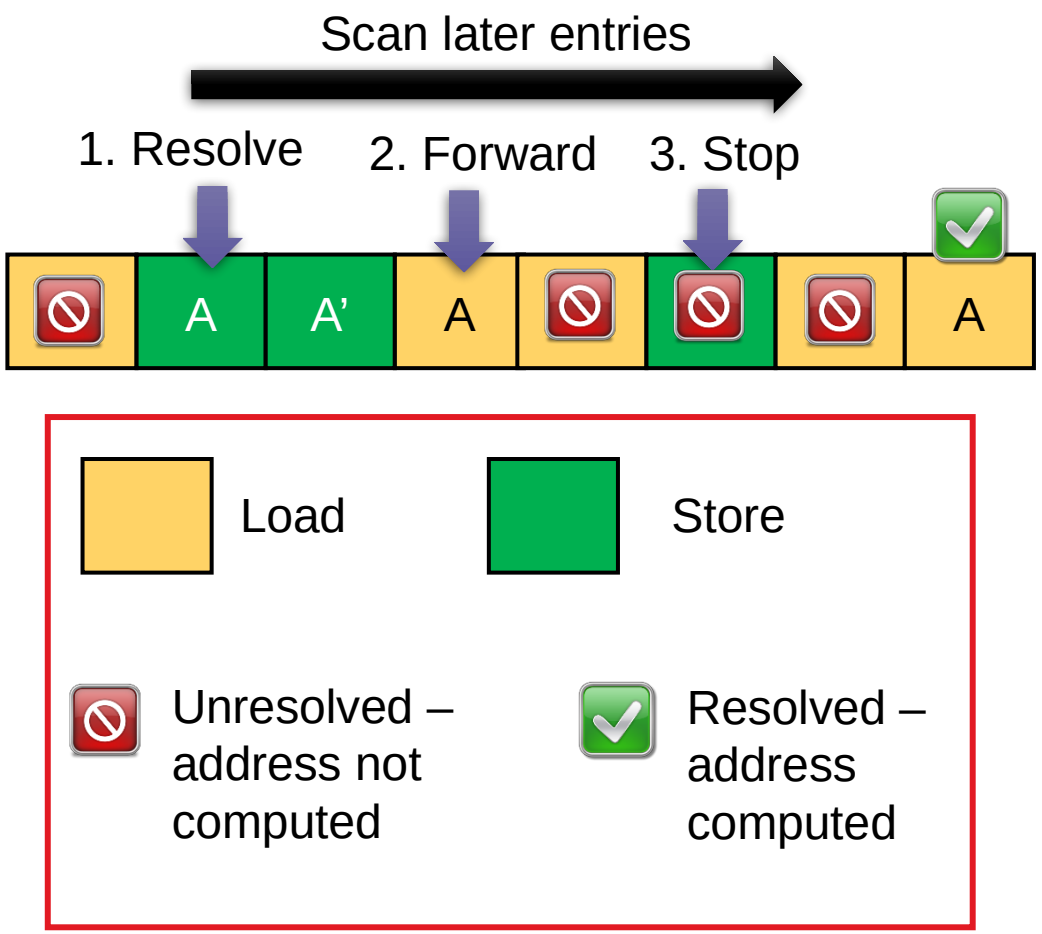
Resolved and Unresolved Eentries



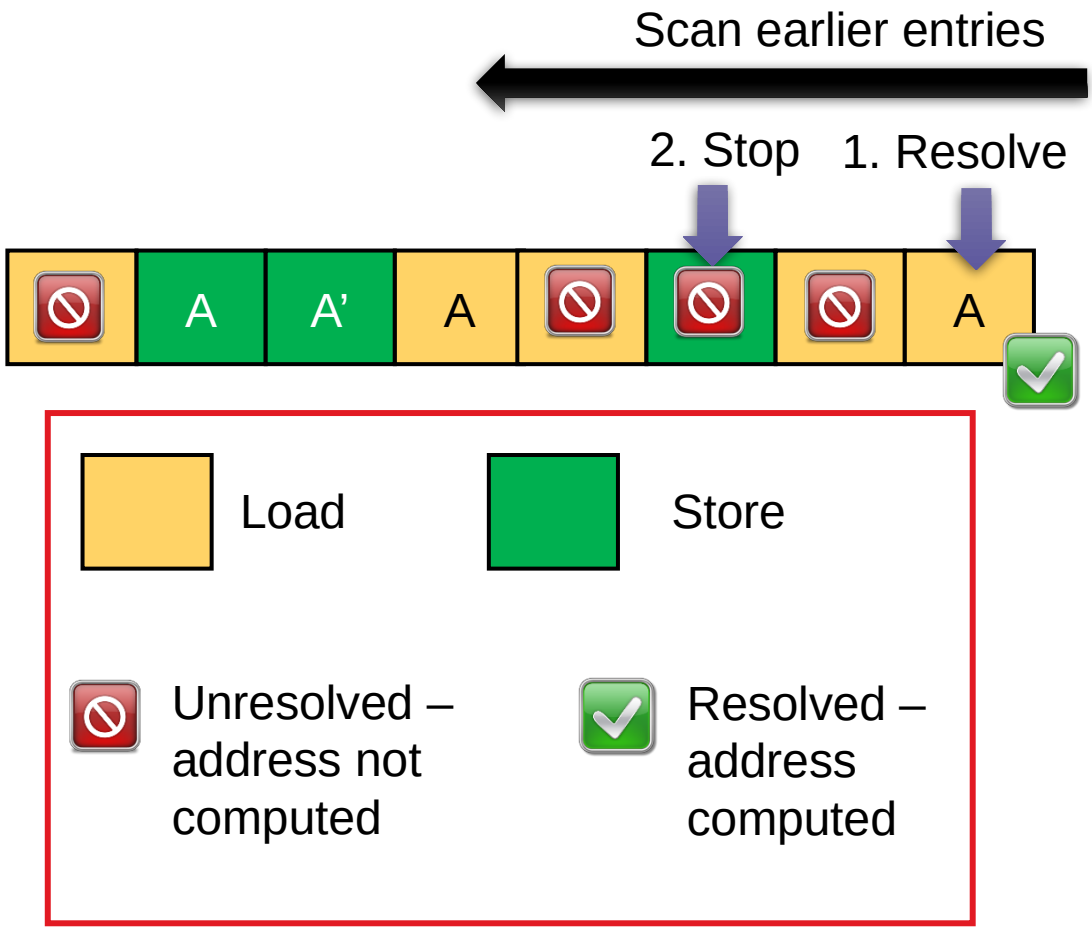
Compute Address of a Store – Case I



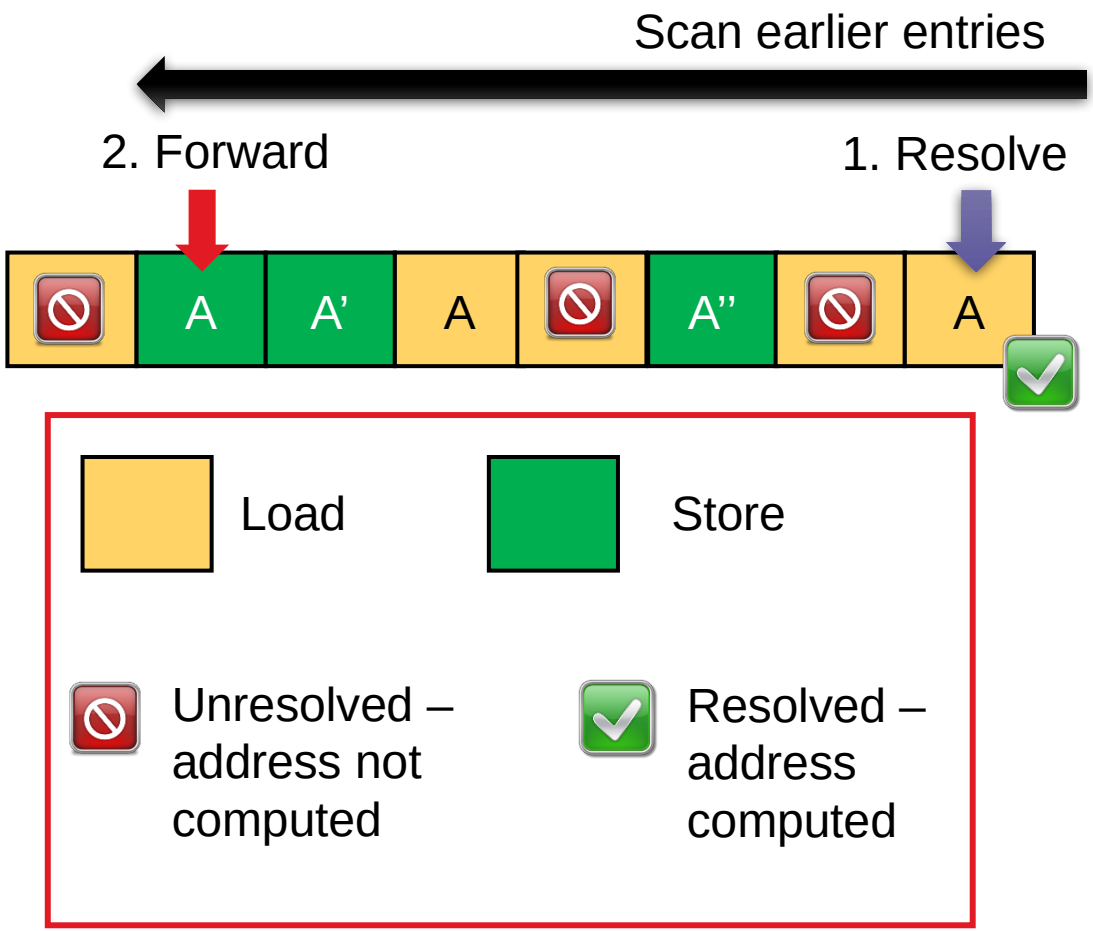
Compute Address of a Store – Case II



Compute Address of a Load – Case I



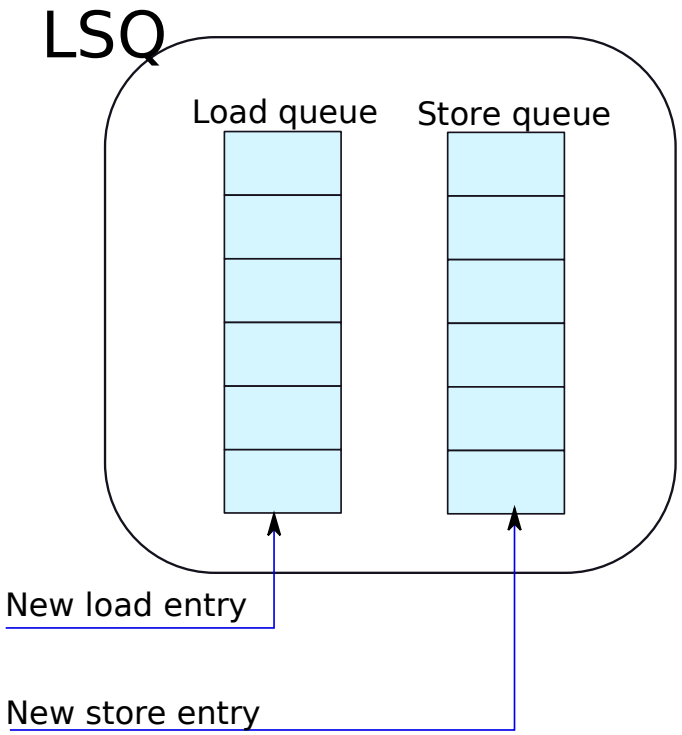
Compute Address of a Load – Case II



Summary of the Discussion

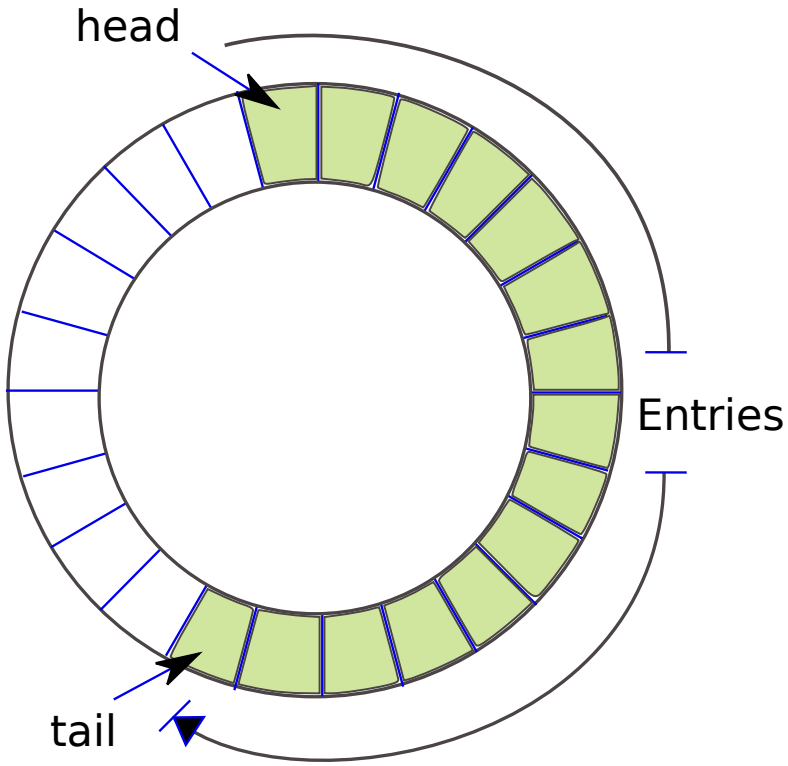
Type	Search direction	Action: Condition
Loads	Search all the stores before it	1. Terminate and forward value: Store to the same address 2. Terminate: Store with an unresolved address 3. Go to d-cache: Conditions (1) and (2) not met
Stores	Search all the loads and stores after it	1. Terminate: Store to the same address 2. Terminate: Store with an unresolved address 3. Forward value: Load from the same address

Actual Implementation

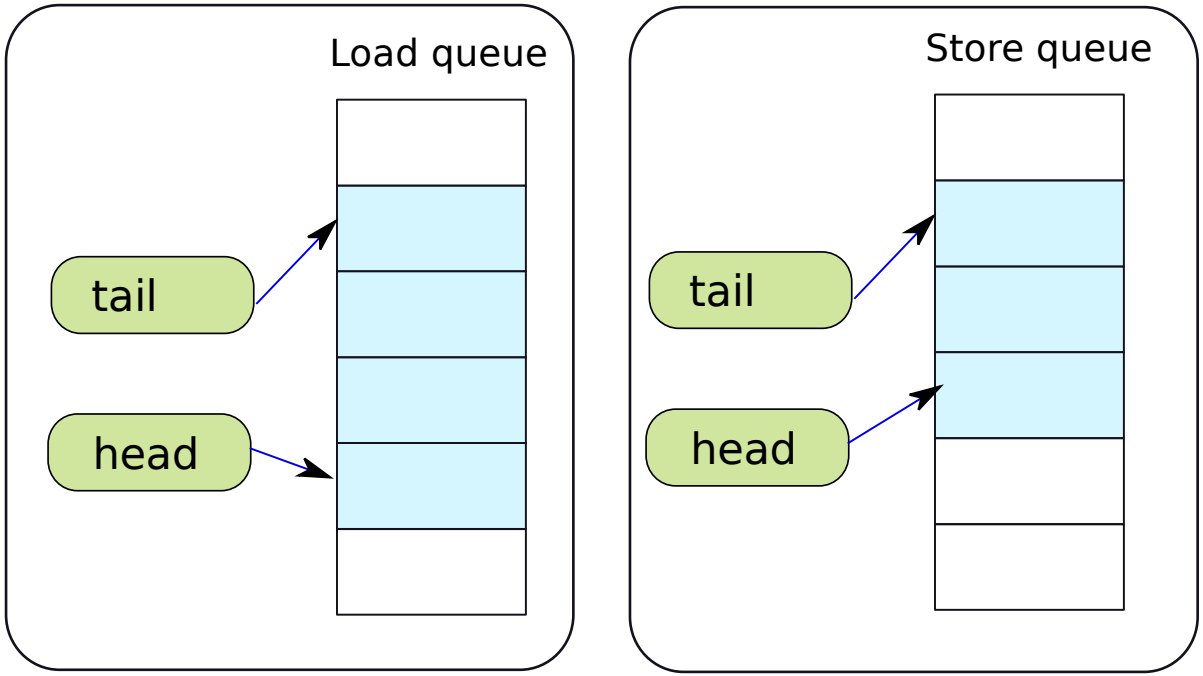


- For the purpose of efficiency, have two queues:
  - One load queue
  - One store queue

Both are arranged as a circular queue



Organization



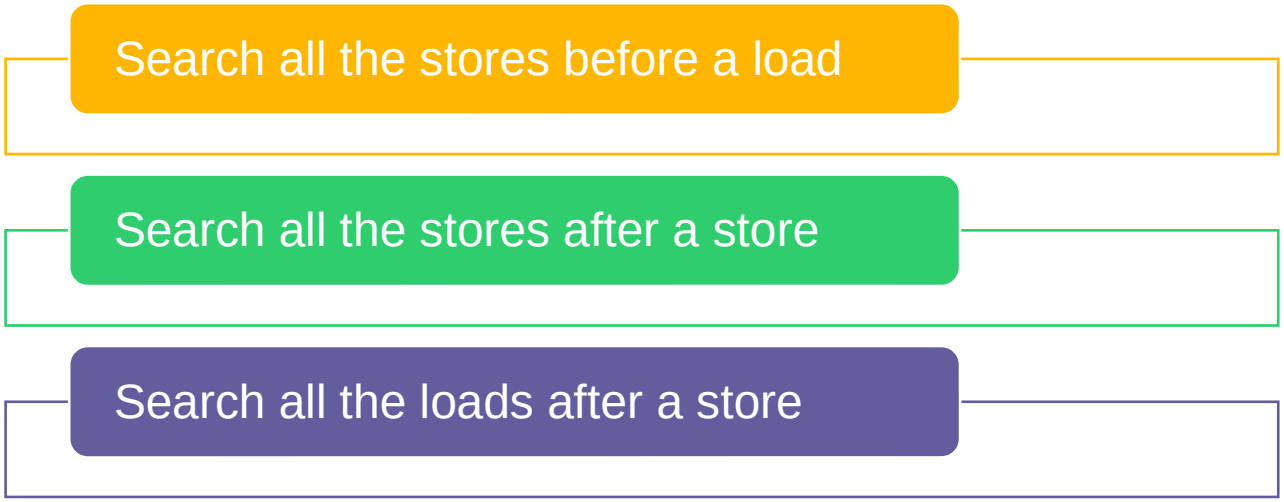
Load queue entry

1. Load address
2. Index of the *tail* ptr in the store queue when the entry was added

Store queue entry

1. Memory address and value
2. Index of the *tail* ptr in the load queue when the entry was added

Basic Search Operations



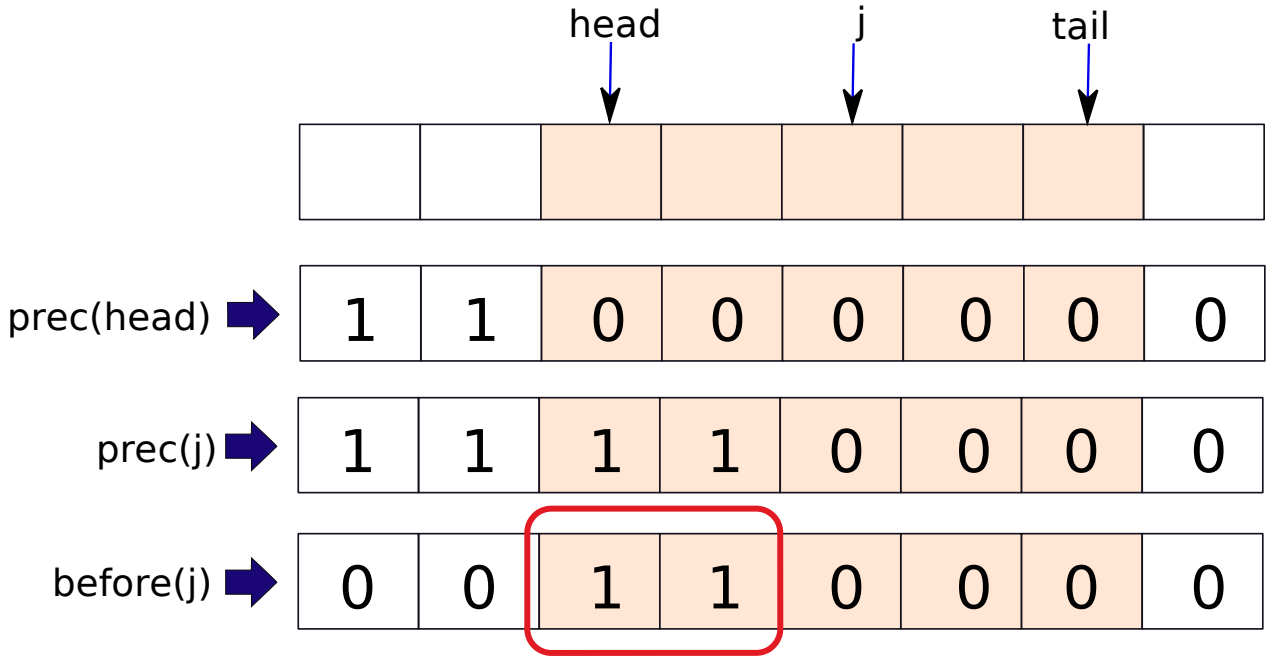
- Represent locations as a bit vector. If a queue has *N* entries, we have *N* bits: one bit per each entry.
- Create a bit vector called *prec* – all the locations before a given location after set to 1.

Implementation of functions: *before*

$j \geq head$

$$before(j) = \overline{prec(head)} \wedge prec(j)$$ be

Compute quickly in HW



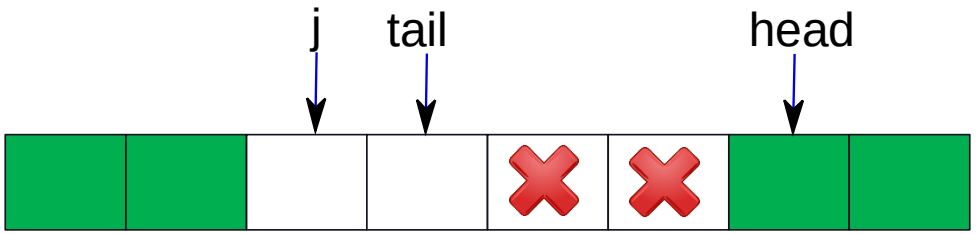
Entries before *j* in the queue

Expressions for other cases

$j < head$

Wraparound

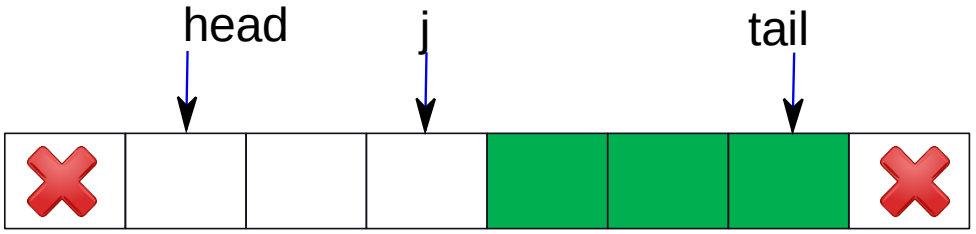
$$before(j) = \overline{prec(head)} \vee prec(j)$$



after function

$j \leq tail$

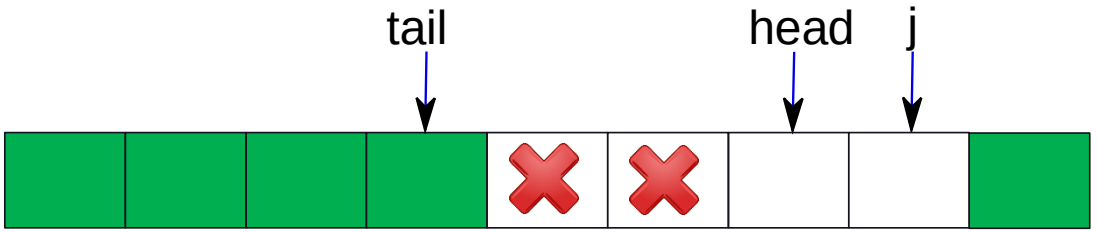
$$after(j) = \overline{prec(j)} \wedge \overline{map(j)} \wedge (prec(tail) \vee map(tail))$$



$j > tail$

Wraparound

$$after(j) = (\overline{prec(j)} \vee prec(tail) \vee map(tail)) \wedge \overline{map(j)}$$



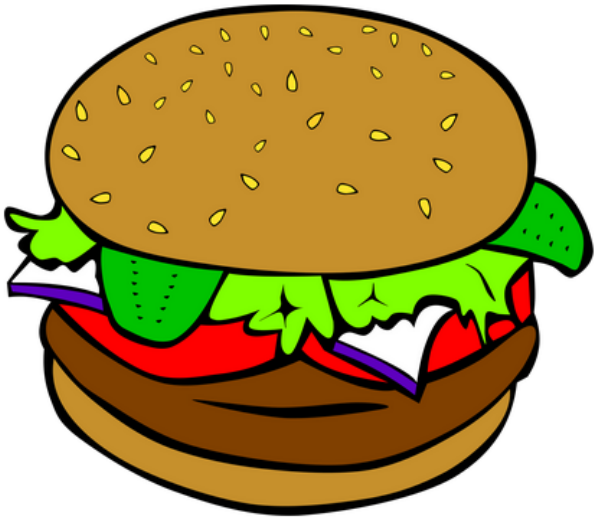
Operation of the LSQ

Type	Search direction	Action: Condition
Loads	Search all the stores before it	1. Terminate and forward value: Store to the same address 2. Terminate: Store with an unresolved address 3. Go to d-cache: Conditions (1) and (2) not met
Stores	Search all the loads and stores after it	1. Terminate: Store to the same address 2. Terminate: Store with an unresolved address 3. Forward value: Load from the same address

- We need to maintain a few more bit-vectors
  - *resvd:* bit-vector to store whether an entry is resolved or unresolved
  - *match*: all the entries that match a given address (we can use a CAM array for this purpose, Chapter 7)
- *Example:* All matching/unresolved stores before a load:

Wrap-up

- We can compute various functions and find a set of locations of interest.
- We need to find either the first or last location.
- How do we do this quickly in hardware?

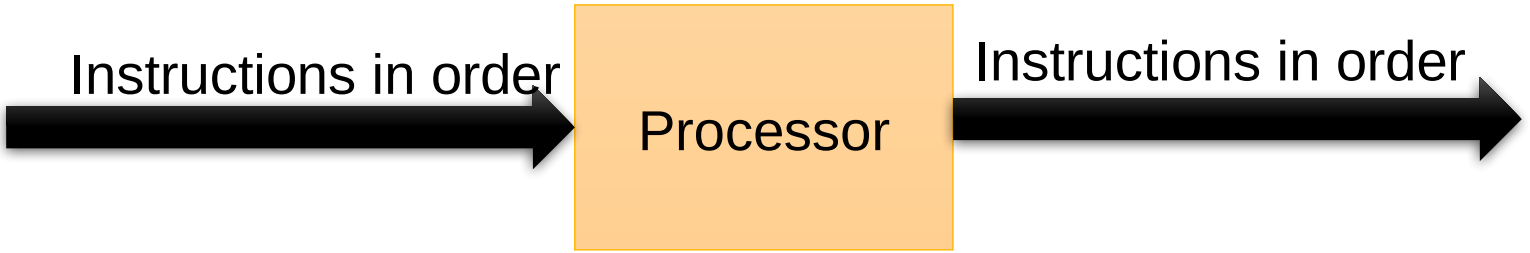


Can we use a select unit?

Contents

1. Instruction Renaming
2. Instruction Dispatch, Wakeup, Select
3. The Load-Store Queue (LSQ)
4. Instruction Commit

Precise Exceptions



For an external observer

- Instructions need to appear to execute in order
- At any point of time, the instruction flow can be paused, and resumed later.

New Structure: Reorder Buffer (ROB)

Reorder Buffer (entry)

- Contains 1 entry for every instruction that has been fetched
- After decoding an instruction, we enter it in the ROB
- If there are no free entries, the **OOO pipeline** stalls
- Instructions are entered into the ROB in program order
- The ROB is essentially an in-order queue

Updating an entry

- Whenever an instruction finishes its execution, we update the entry of the ROB
- Mark it **ready**

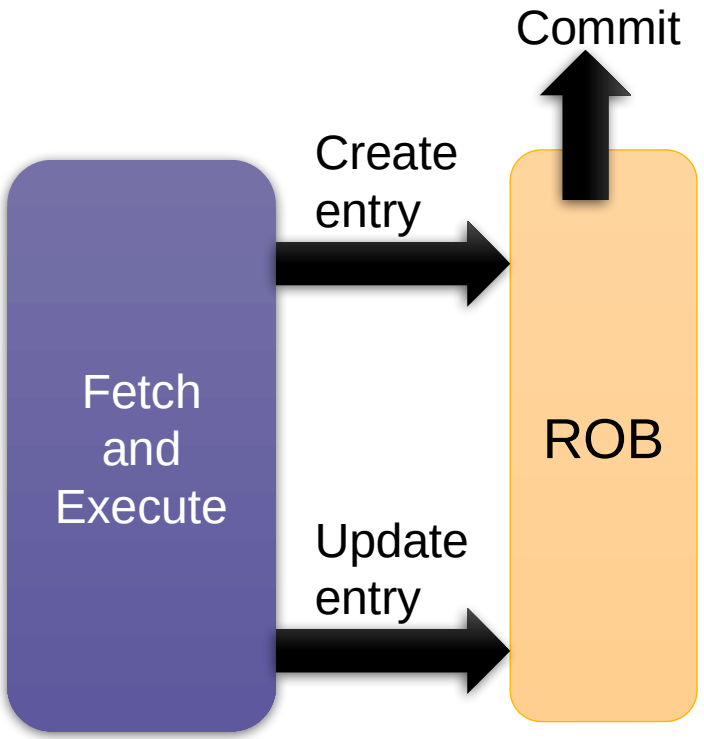
Removing an entry

- It is called retiring or committing
- This happens in program order (keep reading ...)

Removing Entries from the ROB (Commit/Retire)

Every processor has a commit width

- Number of instructions to commit per cycle
- Let the commit width be **W**
- 2, 4, or 6 (typical values)
- Start from the top of the ROB and access the next W entries
- Stop at the first entry, which is not ready or continue till we reach the W^{th} entry
- Commit all the entries that we find to be ready



What does it mean to commit an entry?

Remove it from the ROB

AND

Depends on the type of instruction

Consider instructions with a register destination

- Inst J: r1 = r2 + r3
- Assume that r1 is mapped to *py*
- Before this instruction, *r1* was mapped to *px*

When can we free *px*?

- After *J* commits, there are no instructions earlier than *J* in the pipeline
- There is no instruction that requires the value in *px*
- *px* can be reclaimed, and added to the free list

Updates to renaming

Whenever we see an instruction of the form:

- Instruction J: r1 = ...
- Remember the previous mapping of *r1* by reading the rename table
- Assume that *r1* was mapped to *px*
- Save this mapping in the ROB entry for instruction *J*
- When *J* is committed, unmap *px,* and return it to the free list

This can be done for all instructions that have a destination operand as a register: ALU instructions, load instruction

Load and Store Instructions

When should a store be written to memory?

- ANS: When we are sure that it is not in the wrong path of a branch, or no interrupt will come before it
- Only possible at commit time.
- Solution: Send stores to memory at commit time.
- At the same time remove them from the store queue

When should a load be removed from the LSQ?

- At commit time ^

What do we in the case of an interrupt?

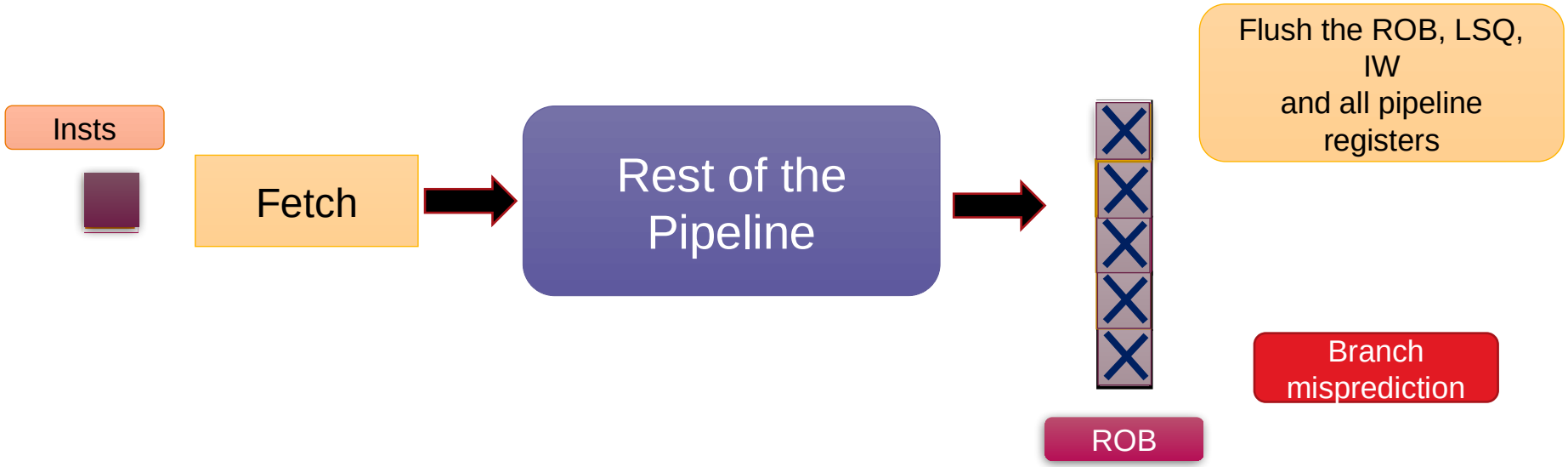
Let us treat interrupts, exceptions, and branch mispredictions in the same category

We will clearly have some instructions in the pipeline that are on the wrong path. They should not be committed

Use the ROB:

- Mark the instruction that has had an exception, or suffered from a misprediction, in the ROB
- In the case of an interrupt mark the topmost entry in the ROB
- Wait till the marked instruction retires
- Initiate the recovery sequence
- More on the next slide ▭

Flushing the Pipeline



Recovery Sequence

- Flush the pipeline
- Remove the entries from the IW, LSQ, and ROB
- Remove entries from all pipeline registers and other temporary structures (not the rename table or register file though)

Restore the state

- Architectural register state for the last committed instruction and the PC
- The contents in the rest of the physical registers does not matter

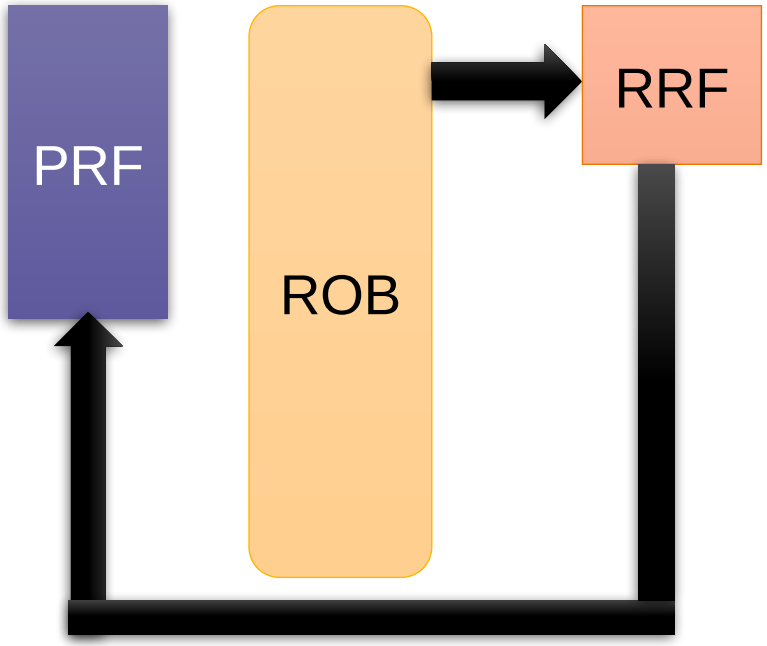
Approach 1: Retirement Register File (RRF)

RRF ⊔ Maintain the state of all architectural registers at commit time

Each entry of the ROB gets augmented with the value of the destination register

Write the value of the destination register at commit time.

The RRF maintains a checkpoint at commit time



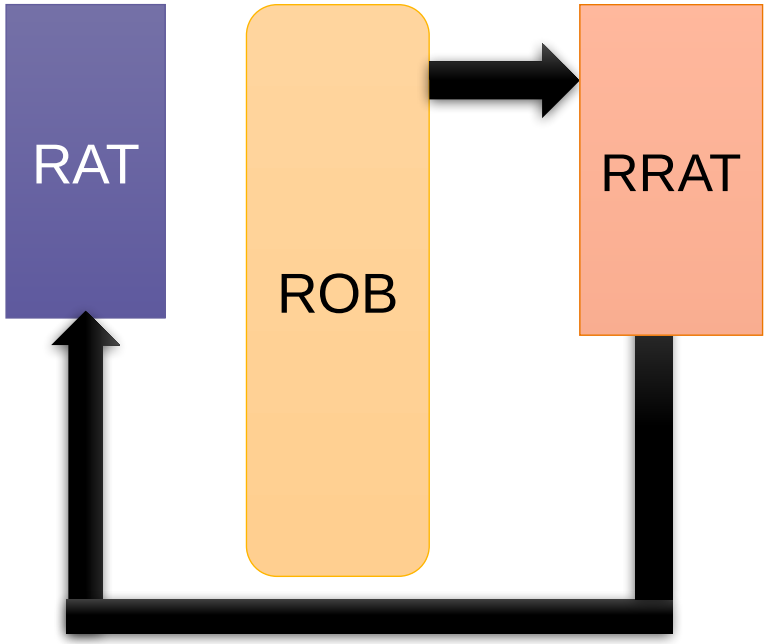
Approach 2: Retirement RAT (RRAT)

Why do we need to write the values into the RRF?

Aren't just the mappings sufficient.

Maintain a copy of the mapping of the destination register in the ROB entry

- Eg: r1 = r2 + r3, *r1* gets mapped to *py*
- Maintain *r1* = *py* in the ROB entry
- Update the RRAT with this mapping



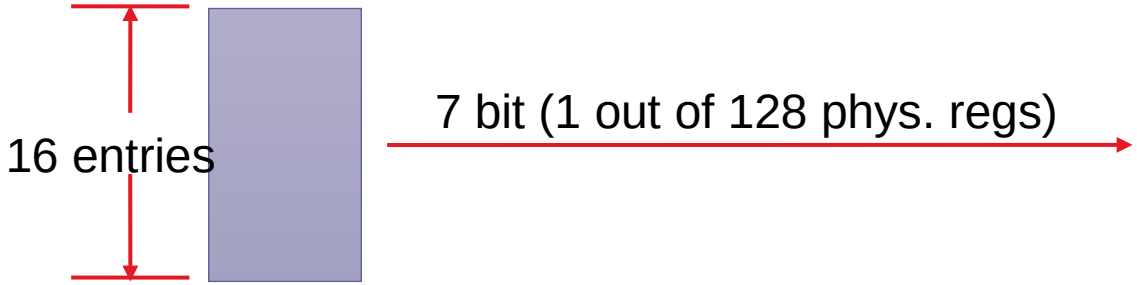
At the time of recovery

- Transfer the mappings in the RRAT to the RAT

Prove that this can be done.

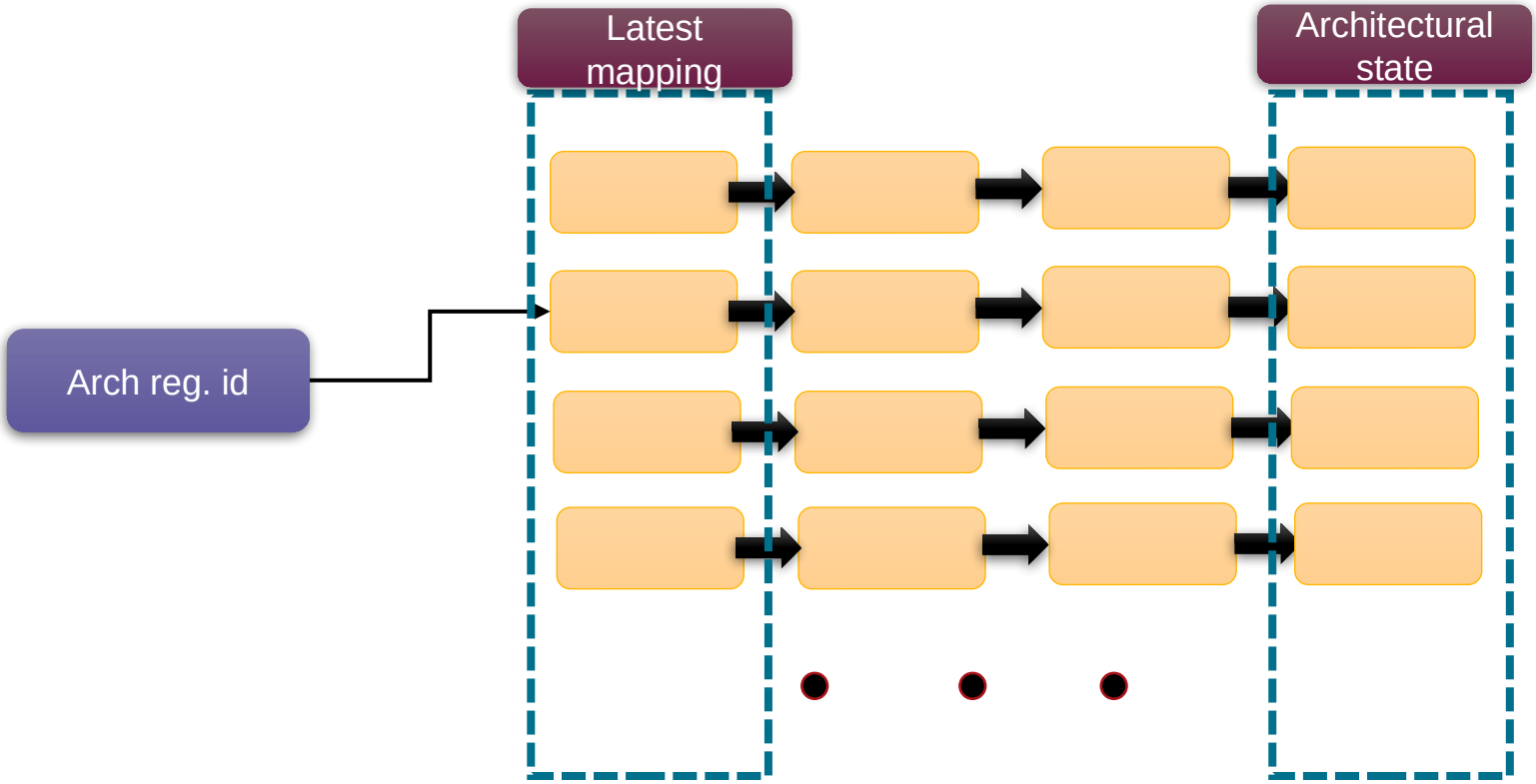
Approach 3: RAT Checkpointing (SRAM based)

Consider the following design



- At every branch let us maintain a checkpoint
- Set of 16, 7-bit ids
- In each entry of the RAT, have a shift register, where each entry of the shift register is 7 bits wide
- To create a checkpoint, shift the entries in each row.
- The last entries of each shift register represent the current **architectural state**
- Remove the last entries once the corresponding instructions commit

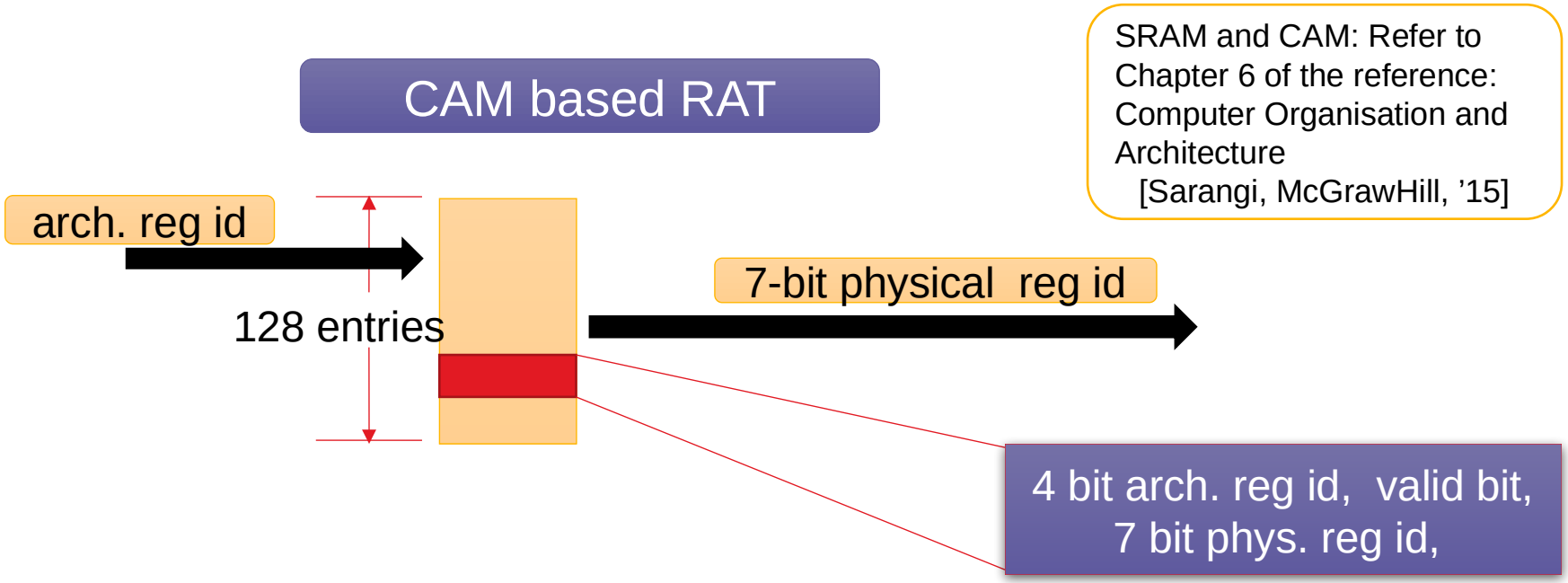
RAT



Approach 4: RAT Checkpointing (CAM based)

A CAM (content addressable memory) is like a hash table

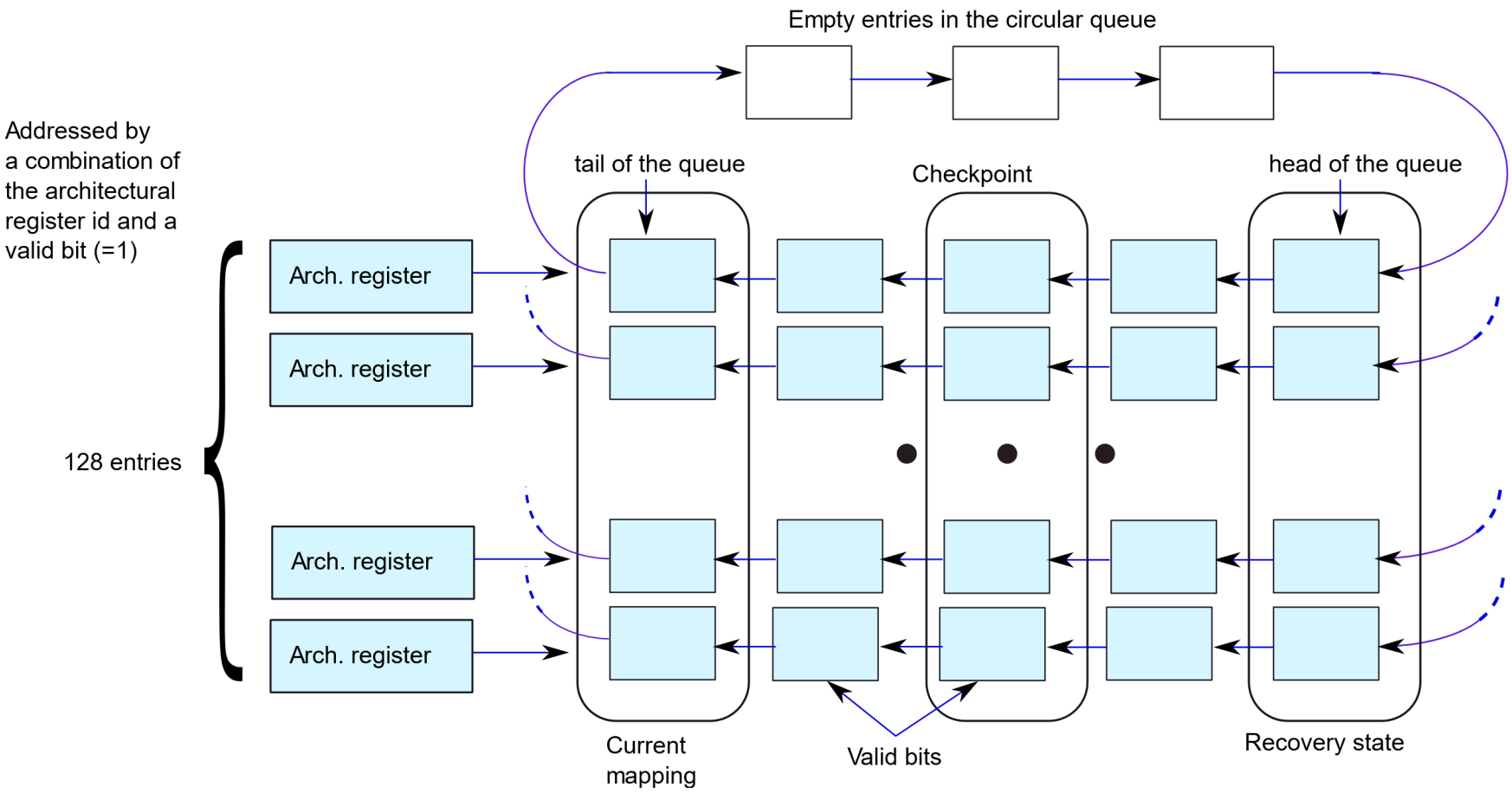
- You address each element not by an index, but by a subset of its content
- It is slower than a structure that is addressed by an index



CAM based RAT

- At any point of time, 16 bits are set to 1. They contain the current mapping (latest state)
- The architectural state is also a set of 16 entries in the RAT table.
- We can save a checkpoint by saving the set of valid bits in a 128 bit vector.
- For every branch, we can take a snapshot of the valid bits.
- The best way to achieve this is to have a shift register in each row of the RAT table, where each entry is 1 bit wide
- Use the same idea as the SRAM based RAT

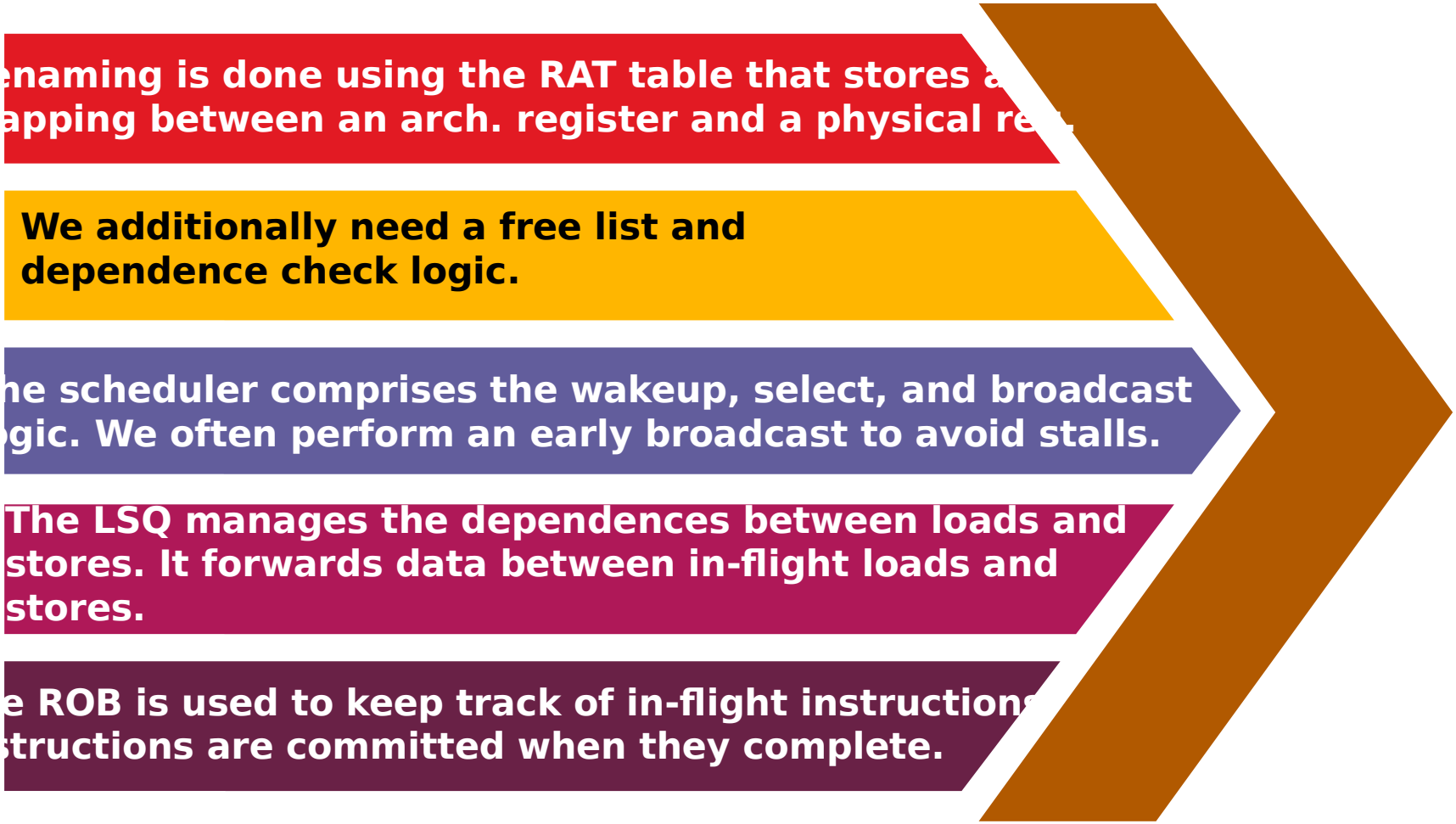
CAM based checkpointing, one 1 bit per row



Pros and Cons

RRF	+ point ▭ Simple to implement. Transferring the checkpoint is easy - point ▭ Extra register writes every cycle. More power - point ▭ Need to save the value in the ROB. More space.
RRAT	+ point ▭ Requires less space in the ROB than the RRF - point ▭Has activity every cycle.
SRAM based	+ point ▭ Activity only on a branch. Shift operation. - point ▭ Each row of the RAT is wider. Cannot resume from non-branch points.
CAM	+ point ▭ Shift operations are easy. Shift only 1 bit. - point ▭ The CAM per se is a slower structure than an SRAM.

Conclusion


enaming is done using the RAT table that stores a
apping between an arch. register and a physical re
We additionally need a free list and dependence check logic.
he scheduler comprises the wakeup, select, and broadcast gic. We often perform an early broadcast to avoid stalls.
The LSQ manages the dependences between loads and stores. It forwards data between in-flight loads and stores.
e ROB is used to keep track of in-flight instructions
tructions are committed when they complete.

The End