ADDRESS BUFFER

Overview

- What the purpose of having an address buffer?
  - Memory Disambiguation is required to solve RAW hazards on memory operands. Before we can issue a “LW” to the data cache we need to make sure that there is NO pending senior “SW” with the same address. This requires that we compare the address of “LW” with the address of every “SW” ahead of it in the LSQ. This means that we can’t release the LSQ entry of “SW” until it writes its data to the cache. This causes the LSQ to fill up very quickly and may cause the entire pipeline to stall.
  - To solve this problem, we add the store address buffer. The store address buffer allows every ready “SW” to leave the LSQ and join the address buffer given two conditions:
    - The effective address of all senior load instructions is known.
    - The address buffer is not full.

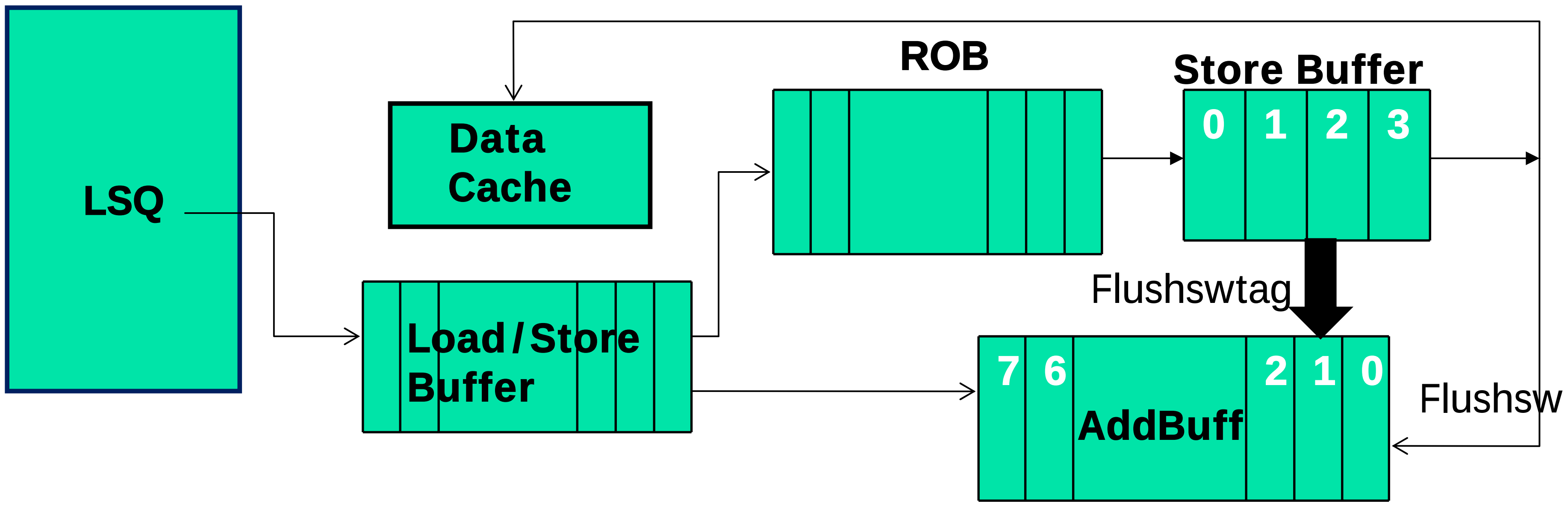
    However, this requires every “LW” to keep count of the number of junior “SW” instructions with the same address that bypass it, lets call the counter “junior count”. Every “LW” needs to compare its address with the address of every “SW” ahead of it in LSQ and all “SW” instructions in the address buffer. If the number of matches exceeds the “junior count” then “LW” can NOT be issued to cache.

Overview

- Address buffer has eight entries indexed from 7 down to 0. Each entry is sub-divided into five fields as shown below.
- New entries always join at entry(7), and every time an entry is released we shift the entries upward to it one location downward. Similar to what we have done in LRU stack in the divider with cache lab.

Valid (1 bit)	Address (32 bit)	Rob Tag (5 bit)	SB Tag (2 bit)	TagSel (1 bit)

Block Diagram

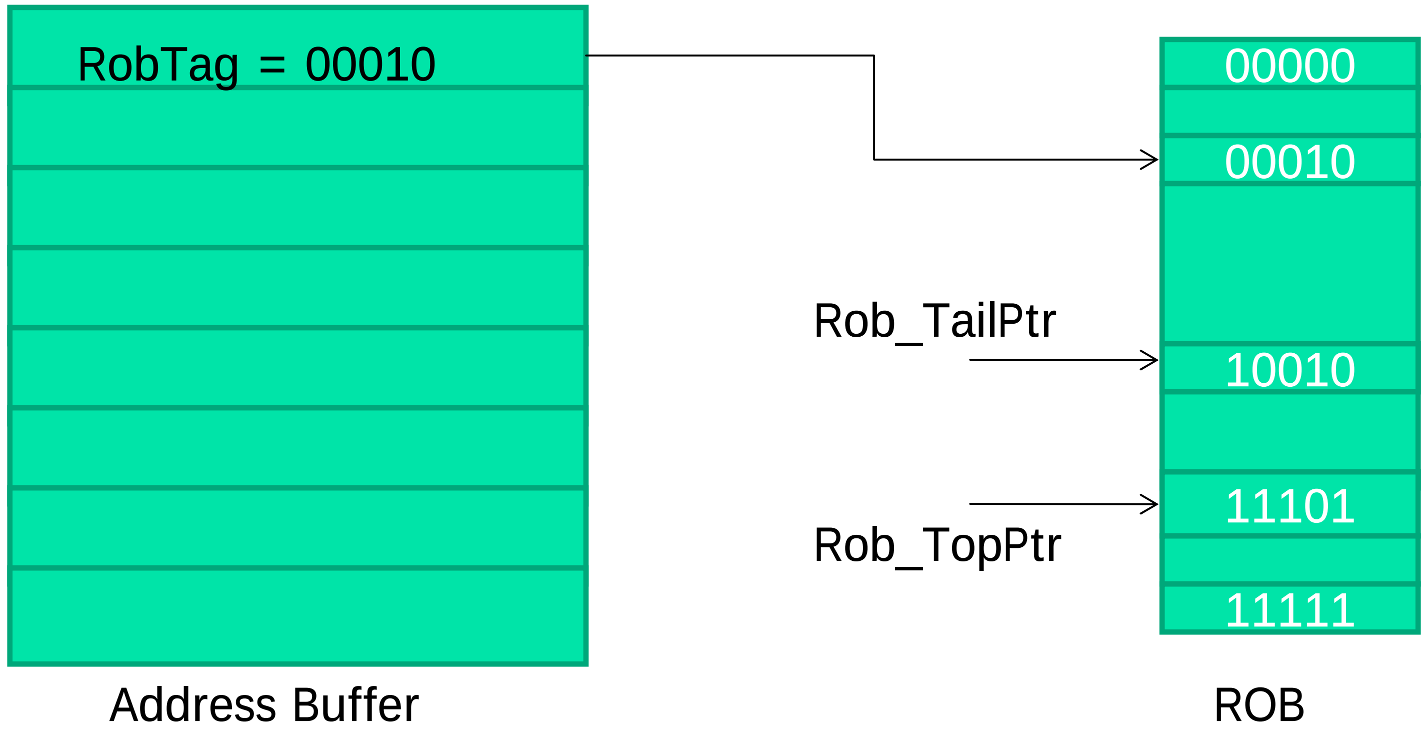


Operation

- Memory Disambiguation:
  - Associative Search: For every entry in the LSQ, we compare its address field with all entries in address buffer and record the number of matches and we set a single bit to indicate that at least one match was found. The associative search is pure combinational.

- Rob depth Calculation:
  - For every entry in address buffer, we calculate the Rob depth. The Rob depth is the distance between the Rob entry of the “SW” in the address buffer and the Rob top pointer taking into consideration the fact that ROB is implemented as a circular buffer.
  - This information is required to perform selective flushing due to branch misprediction.

Example


RobTag = 00010
00000
00010
Rob_TailPtr
10010
11101
Rob_TopPtr
11111
Address Buffer
ROB

Operation

- Flushing Address Buffer Entries:
  - Selective flushing due to branch misprediction provided that the entry Rob depth is larger than that of the mispredicted branch. When a branch (or Jr$31) instruction is mispredicted, we need to flush all instruction younger than the mispredicted branch.
  - When a “SW” writes to the cache and leaves the store buffer, we need to flush the corresponding entry of that store in the address buffer. In this case, we need to use the SBTag to identify the address buffer entry that must be flushed.
- SBTag and TagSel:
  - SBTag: Every time a “SW” completes and reaches the top of the ROB, it joins the store buffer (given that it is not full). At that instance, a 2-bit SBTag is assigned to the store buffer entry. The SBTag is also conveyed to the address buffer to save it in the corresponding entry.
  - TagSel: When the SBTag is saved in the address buffer entry, the TagSel bit of that entry is set to ‘1’.

Operation

- There are two reasons why we need SBTag and TagSel:
  - When a “SW” reaches the top of the ROB and is ready to commit, at that instance the “SW” releases its Rob entry and can no longer uses the RobTag to flush the address buffer entry. Because it could be the case that another “SW” instruction is assigned to the released ROB entry, and this junior “SW” may leave the LSQ generating an entry in the address buffer before the senior “SW” writes to the cache. This will result in 2 entries in the address buffer that contain the same RobTag.
  - The second reason is that we perform selective flushing of the address buffer entries in case of branch misprediction. Every store that is younger than the mispredicted branch must be flushed. The SW instruction who was the senior most instruction when it was at the TOP of the ROB, suddenly becomes the junior most instruction when it leaves ROB (because the ROB TOP pointer is incremented). Hence, if any branch currently in the ROB is mispredicted (before this store word instruction gets out of the store buffer), all younger instructions' entries in the address buffer get flushed from the address buffer and our store word entry gets definitely flushed.