## Approach:

The main idea is that we will make 1024(for each row) deques which will store what to execute. So like we will keep enqueueing in the respective structure with values and when there is any blocking because of any register or memory than then program will look in “regRowGroup” to find which bus to execute to end blocking. At last, if all execution is done and then program will check that, is every bus is empty else make buses empty by executing (this checking and execution is done in an O(instruction) time using a vector row and queue DRAMreq which helps in determining the non-empty buses).

## New struct:

**DRAMqueue:** It is a structure which contains (ins,reg,regValue,mem,memValue}.

**regRowGroup:** It is a vector storing 32 integers, these are the row groups in which the register is busy for DRAM execution.

**DRAMreq:** Store the lw/sw instruction when they are executed.

**row:** Store for each row group how many instructions are executed/cancelled already which are not removed from the DRAMreq yet.

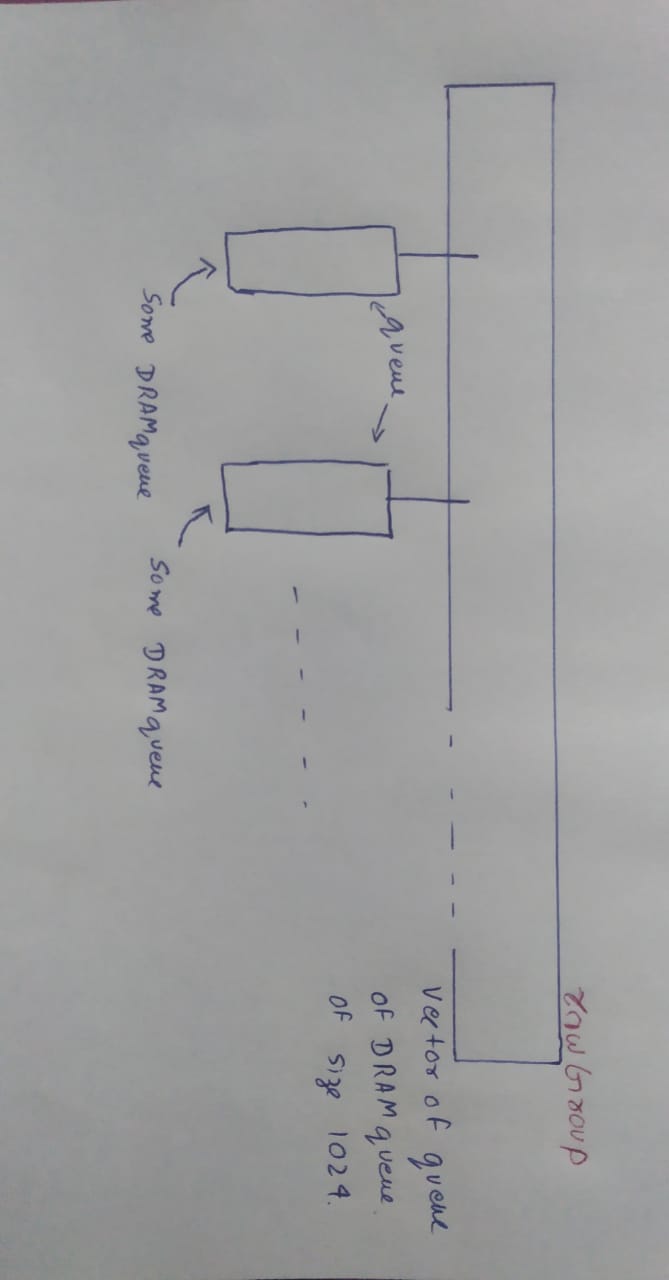
## New functions:

**transferFromQueueToDRAM:** It will pop from queue and put row in rowbuffer and buffer updates will be updated accordingly.

**nextDRAM:** This function is used to decide the next instruction to be executed from the waiting deques in an efficient manner in an attempt to reduce writeback and load of rows in row buffer.

**executeDRAM:** It will first check whether any register or memory address is blocking or not if blocking then clear the blocking by popping from respective queue and executing the instructions in DRAM in an efficient manner else continue program.

**putLw\_SwInQueue:** The main purpose of this function is to enqueue in DRAMqueue in respective data bus deques. But for optimizing purpose we have also done skip kind of things in it like lw then lw at same register than we have skipped first lw.



It’s a follow up of Minor exam and other aspects have been explained earlier.

## **Strengths:**

1. This is a feasible linear time algorithm with respect to the number of instructions to which reduces the overall cycles requirement by reordering the DRAM servicing of instructions and reducing the number of writebacks and loading of rows required.
2. When an instruction is encountered which is unsafe to execute than the registers are serviced in an efficient manner in DRAM. For example, if there is an add instruction with $t1, $t2, $t3 involved in it and let’s say the load instruction on these registers are waiting in queue to be serviced as follows: $t3 in 1000 mem (grp 1), $t2 in 2000 mem(grp2) and $t3 in 1004 mem(grp1) and the current process in DRAM being done in 2004 mem(grp2). So first $t2 is freed, then $t1 and $t3 depending on which came first in queue.
3. If a pair of instruction in queue gets consecutive to each other such that first one is and sw and second one is a lw from same memory (and register not busy anywhere else) then a store to load transfer happens and the second instruction is put in queue to be serviced in DRAM.
4. If a pair of instruction in queue gets consecutive to each other such that first instruction is lw and second instruction is also a lw at same register then the previous lw is popped from the queue as it would be overwritten later.
5. If a pair of instruction in queue gets consecutive to each other such that first instruction is sw and the second is a sw at same memory location then the previous instruction is popped from the queue and is not serviced DRAM.
6. If a pair of instruction in queue gets consecutive to each other such that first instruction is lw and the second is a sw at same memory location using the same register as in load instruction then the current instruction is not put in queue as it does not change anything.

## **Weakness:**

* Could have made more efficient but didn’t do that because that would make program to run O(n^2) where n=total instruction (or would require a map large enough to store all memory locations as key and an integer value corresponding to them to keep time complexity O(n)).

Eg.

Sw $t0, 1004

Sw $t1, 1000

Sw $t2, 1004

Sw $t3, 1000

In this case even though storing $t0 at 1004 doesn’t make any sense because later we are changing it to the value at $t2.

* First sw/lw will not be skipped when the first instruction and later instruction has same memory address.

Eg.

sw $t1, 1000

sw $t2, 1000

This happened because there is no look ahead of upcoming instruction as soon as the first instruction comes it starts getting processed in the DRAM and when the second instructions come it waits in the queue. A possible solution would be to force stop the ongoing DRAM processing force stopping a loading of row from memory to rowBuffer or a writeback of memory to rowBuffer or updating a register value from the rowBuffer have the potential to corrupt the memory or register values temporarily (i.e., put wrong values in them) hence we had not implemented such forced stopping of DRAM processing.

* All sw/lw will be executed when current and later instruction have same register and different memory.

Eg.

lw $t1, 1000

lw $t1, 1004

lw $t1, 1008

This is done so as not to do any force stopping of DRAM process or to involve look aheads of instructions which would make the complexity O(n\*n).