Approach: Part1:- In this, first i had created an matrix of 2^10x2^10 for DRAM memory and a buffer array of size 2^10 for the row buffer. Now whenever a add or addi instruction is there it is executed and the registers are changed accordingly. When there is lw/sw instruction then i check if address is in direct form like 1004 or in offset form like 4($t2) , then i got the addresss and i check if address is in multiples of 4 or not, as i am storing word if not then i return error and stop. Now with this address i have row number = int(address/2^10) and column number = address%2^10 . Then i check for three conditions :

1. row buffer is empty: In this case i load the row(= row number) from DRAM memory into row buffer(this takes ROW_ACCESS_DELAY cycles) and then look for column (= column number) in row buffer (this takes COLUMN_ACCESS_DELAY cycles) and do the operation corresponding to lw/sw.

2. row buffer have same row as row number: In this case i look for column (= column number) in row buffer (this takes COLUMN_ACCESS_DELAY cycles) and do the operation corresponding to lw/sw.

3. row buffer have diffferent row as row number: In this case first i wirte the current row in row buffer into DRAM memory (this takes ROW_ACCESS_DELAY cycles) and then i load the row(= row number) from DRAM memory into row buffer(this takes ROW_ACCESS_DELAY cycles) and then look for column (= column number) in row buffer (this takes COLUMN_ACCESS_DELAY cycles) and do the operation corresponding to lw/sw.

One important point is that i have considerd total memory as row buffer memory and matrix memory, so i didnot write back the remaining row buffer back into matrix memory as it will add up to extra ROW_ACCESS_DELAY cycles. So in case of row same as row buffer , row buffer values will be considered. Also i have initialized memory values to zero considered as empty memory.

Strengths:- In this approach we have maitained a row buffer which helps us to minimize the total cycles in execution, as we didnot have row buffer then we have to look for value in memory every time and accesssing a memory takes much time then accessing a row buffer as we can see the general values of ROW_ACCESS_DELAY is higher than COLUMN_ACCESS_DELAY, so we didn't have to add ROW_ACCESS_DELAY everytime we need to access memory as we already have the corresspoding row and we just need to look for the column which takes much less time.

Weakness:- we have to wait for lw/sw to finish then only we can execute next instruction, whole code is run into serial but we could have done some instructions parallely to lw/sw. There is increases in space complexity as we have to create an extra row buffer which takes its own space of 2^10. Part2:- In this, basic approach is same as part1 but there are some changes also.

In part2, i had executed the next instructions of add/addi which are not dependent on lw/sw executed. When instruction add/addi dependent on lw/sw is encountered before the previous lw/sw instruction is finished, i finish the lw/sw then i execute instruction encountered. If number of add/addi instructions (after a lw/sw instruction) not dependent on this lw/sw are greater than number of cycles required to finish lw/sw then in between i print the cycle when the lw/ sw is finished and in this same cycles i executed the independent add/addi instruction. If there is another lw/sw instruction after a lw/sw then i wait for the previous lw/sw instruction to finish then i proceed. Now to clarify when instructions are dependent and when they are not:-

if we are using a register $r0 in {lw $r0 address } and there is also $r0 anywhere in add/addi then they are dependent else independent as we have not to change $r0 until whole DRAM accessing or writing into memory/register is done.

Strengths:- In this approach we are executing some more independent instructions also in parallel we are accessing the DRAM memory. So there is decrease in total cycles also for some cases like instructions which doesnot contains any lw/sw instruction we can run it otherwise we have to wait for lw/sw instruction to finish, so if there is only one lw/sw instruction we can just issue a request and then proceed on with our code.

Weakness:- We have to still wait for lw/sw to execute before we can execute another lw/sw instruction. If there are only two lw/sw instruction then we have to wait for one to finish then execute another lw/sw. But if we need output of part of code which is indepent of these two lw/sw instructions then we cannot do anything but wait. This interpreter cannot ignore lw/sw instructions and instructions dependent on these and execute other independent instructions. We can do many lw/sw that are in same row simultaneously, but our interpretor cannot do so. Input:- input is a filename and ROW_ACCESS_DELAY and COLUMN_ACCESS_DELAY. For part1, run on terminal $ a3p1.cpp filename ROW_ACCESS_DELAY COLUMN_ACCESS_DELAY For part2, run on terminal $ a3p2.cpp filename ROW_ACCESS_DELAY COLUMN_ACCESS_DELAY Output:- It will show what is happening in each cycle and at end total number of cycles and number of row buffer updates.