Name: Gaurav Jain Entry Number: 2019CS10349

COL216: Minor Exam Assignment

Introduction:

This assignment aims to improve the interpreter that we have designed in Assignment 3 to handle a subset of MIPS assembly language instructions by maintaining the memory structure (DRAM). This would give us a basic simulator. The two instructions among the considered instructions that require memory access are *lw* and *sw* instructions.
The DRAM memory model is a 2D model in which the data is stored in a 2D array. A memory address in the DRAM could be thought of as consisting of a ROW ADDRESS and COLUMN ADDRESS.

The *lw* instruction corresponds to the READ operation on the DRAM works as:

1. Activate the ROW corresponding to this address by COPYING the row to a ROW BUFFER at the bottom of the structure. The time required for this operation: ROW_ACCESS_DELAY.
2. Copy the data at the column offset from the row buffer to the REGISTER. Time for this operation: COL_ACCESS_DELAY.

We first check whether the corresponding row is already located in the ROW BUFFER for subsequent READ operations. If yes, then just copy the data at the column offset to the data bus after time COL_ACCESS_DELAY (no need for copying the row into the row buffer since it is already present). However, if the row is different from the one currently located in the row buffer, then:

1. First, copy the row buffer back to its row (Time for this operation: ROW_ACCESS_DELAY).
2. Access the data from a new row going through the READ operation protocol described earlier.

The *sw* instruction corresponds to the WRITE operation on the DRAM, which works as:
If the row is present in the row buffer, update the row buffer data (at the column offset) in time COL_ACCESS_DELAY.
If the row is not present, then:

1. Copy the row buffer back (Time for this operation: ROW_ACCESS_DELAY).
2. Copy the new row to the row buffer (Time for this operation: ROW_ACCESS_DELAY).
3. Update the data in the row buffer in time COL_ACCESS_DELAY.

Name: Gaurav Jain Entry Number: 2019CS10349

Approach to the code for PART1:

For the first part of this assignment, not much change is required in the code. We have the original memory structure, which is a 1D array of size 2^20. The MIPS instruction text file is read line by line, and instructions are separated based on operator type (*add,addi,lw,sw*) and corresponding arguments in the *ins* structure. A map is used to store the ins structure and their key. This key corresponding to each instruction is put in the memory structure.
Now all the instructions are executed one by one like the previous assignment. But, special conditions are added in the *lw* and *sw* instructions. There are two global counters, *PREV* and *CURR*, to know the row number used as the row buffer.
Based on the values of these variables, we can know whether:

1. Row buffer is initialized (PREV = -1)
2. Row buffer is updated (PREV != CURR)
3. Row buffer is unchanged (PREV = CURR)

Based on this, we can add the extra clock cycles required in the execution. Accordingly, we print out the required information. The execution takes place sequentially and, only one instruction is executed at a given cycle.
This approach's strengths are that it can be used to understand how real systems work with 2D DRAM as a memory model. How row access delay and column access delay affects the execution time can be explained using this piece of code. We can implement assembly programs more efficiently by minimizing the delay and reducing the execution time. This will improve the performance.
We have assumed that instructions are executed sequentially in this approach, but we can improve this by executing independent instructions parallelly. This improvement is shown next.

Non-Blocking Memory Access:

The non-blocking memory access procedure can only be applied to two instructions in our ISA- *lw* and *sw*. When either one of them is under execution, we can just execute *add* and *addi* instructions provided they are independent of the original execution.
The notion of the independence of instructions is different for *lw* and *sw* instructions.
For *lw* instruction, the value at a particular address in the memory is loaded in the first register. So during the entire period of this instruction execution, there should not be any *add* or *addi* instruction that uses this register. Otherwise, we can not proceed further.
For *sw* instruction, the value of a particular register is stored in the memory, so there should not be any updates on this register during the execution period. This register can be used as a source for *add* or *addi* instructions.
Thus, independent instructions are executed until we encounter either an *lw/sw* instruction, dependent *add/addi* instruction, or the execution period is over of the original *lw/sw* instruction.

Approach to the code for PART2:

Like the first part, we proceed with the execution of instructions, but when we encounter an *lw* or *sw* instruction, we call an extra function that executes the following instructions until it meets a dependent *add/addi* instruction or *lw/sw* instruction or when the number of instructions exceeds the number of clock cycles required by the original instruction. In all these cases, we exit this function and return the new program counter. This program counter is helpful for further execution as this counter tells us the last executed instruction.
We proceed with further execution sequentially.

This approach is far better than the previous approach. We can have improved the performance by decreasing the number of clock cycles. We can add more instructions to the ISA and see a good amount of performance improvement. This approach takes us closer to real systems.
There are some restrictions on the instructions that we can execute parallelly. This can be improved by lifting some of the restrictions on registers and memory. Memory instructions are always executed sequentially in this approach. We can either increase the number of row buffers, or if both instructions require the same row buffer, we can do them parallelly.

NOTE-
At the end of the execution, the *row buffer update* counter is printed. This counter can have two meanings-

1. The number of times the row buffer is updated or replaced by a completely new row.
2. The number of times the row buffer is replaced by a completely new row or a single address in the row buffer. (I have used this)

Testing Strategy:
Our testing strategy involved thinking and figuring out what can happen when a user interacts with our program and then implementing it to handle such cases. Cases like consecutive row buffer updates, no row buffer updates and a combination of them are tested for the first part.
For the second part, cases like where the number of independent instructions exceeds the execution period are tested. Cases in which there are dependent instructions after lw/sw instruction are tried to see the instructions' parallel execution break.
The testing strategy also involved trying out the different scenarios using various test cases that test all the different scenarios that can transpire while running the program. The different kinds of test cases that were used are included along with the document.

To run the code, read the README file.