Design for Minor 1

Shrey J. Patel
2019CS10400

March 2021

1 Aim

To extend and enhance the functionality of the MIPS simulator(written in C++) prepared in Assignment 3. In particular, the main objectives are:

1. To develop a DRAM memory model and integrate it with the existing simulator.
2. To increase the efficiency of the DRAM based memory model by implementing non-blocking memory access to the simulator.

2 I/O Specifications

- **Input:**
  1. MIPS assembly language program in a text file.
  2. DRAM specifications: ROW_ACCESS_DELAY and COL_ACCESS_DELAY.
- **Output:**
  1. After every instruction:
     - Clock cycle(s) required for that instruction.
     - Modified registers, if any.
     - Modified memory locations, if any.
     - Activity on DRAM.
  2. After completion of execution:
     - Total number of clock cycles used, including those required for write back on last DRAM operation.
     - Total number of row buffer updates.
     - Number of executions for each instruction.
     - Final list of modified memory locations.

- **Execution Instructions:**
  - The code is divided into two separate files for the two parts of the assignment. Both of them can be compiled together with the command '*make*' in the command line.
  - The executable corresponding to each are output1 and output2 respectively. The executable can be run by typing the following in command line:

    ./output1 input_file_name row_delay col_delay
  - Note that row and column delays will use their default values of 10 and 2, if not explicitly provided.
  - The executable files can be removed using the command *make clean*.

3 Approach

3.1 Part 1

The simplified DRAM memory can be modelled as a 2-dimensional array of rows and columns, which is divided in cells of 4 bytes each. In our model, this 2D array can store a total of 2^{20} bytes of memory. Additionally, a row buffer is also required for loading values from the memory or storing values onto this memory. So, I have used a 2D array for storing the main memory and a single 1D array of the size same as the number of columns of main memory, to store the temporary buffer.

The most important thing, however, in this memory model is the amount of delay associated with fetching the correct row from the memory onto the row buffer and due to this a single lw/sw use more number of clock cycles as compared to other instructions.

3.2 Part 2

As mentioned above, a single load or store operation takes multiple clock cycles to execute. This delay is actually the time required by the DRAM memory to load or update data from/to the desired memory cell. But during this time delay, the rest of the instructions are waiting to be executed, i.e. the operations related to DRAM essentially 'block' other instructions from being executed. This results in large amount of delays in the simulation of whole program, and thus the efficiency of the simulator decreases drastically.

So, I have implemented an algorithm to schedule other instructions 'while' some other load/store instructions are running in such a way that the usage of time can be optimized, while preventing any conflicts, some of which are discussed later. This is called non-blocking memory access, as it doesn't stop or block other instructions and allows for some kind of parallel computation. Though multiple threads/cores are not required as the given DRAM memory model is simple enough so that the scheduling is linear.

4 Code Design and Implementation

The code for this assignment is divided into two parts: code1.cpp for DRAM implementation and code2.cpp for non-blocking memory access.

Note that because this assignment is an extension of Assignment 3, I have only described the features that I have added additionally, not the ones already explained in Assignment 3.

4.1 DRAM Implementation

4.1.1 Data Structures

- A 2-dimensional array called **memory** which implements the DRAM memory block having a capacity of total 2^{20} bytes divided into 1024 rows and 256 columns where each cell can store 4 bytes of memory i.e. 32 bits.
- A 1-dimensional array called **ROW_BUFFER** of fixed size of 256. This temporary buffer stores the current active row of the DRAM memory. This buffer serves as the interface between ISA and the memory block as the lw/sw instructions act on this buffer only. The updates/changes are first reflected on this buffer, and these changes are later pushed on the main memory after some delay.

4.1.2 Algorithm

- Whenever a load/store request is made to the DRAM through an lw/sw instruction, primarily two steps are involved, the first is to check if the requested address id the present in the current state of the row buffer(Case 1), in which case all we need to do is to store or load the desired value directly from the buffer.
- But, if the current row buffer doesn't contain the requested address location(Case 2), then it must fetch the particular row from the main memory before any further operation. But before that too, the row buffer should update/store the contents of the row buffer into the main memory. And both these operations require some amount of delay.
- **Delay Calculation:**
  1. **Case 1:** Total delay = Time required to load/store from the current row buffer = COL_ACCESS_DELAY.
  2. **Case 2:**
     Total delay = Time required to store the current row buffer onto main memory + Time required to fetch a new row onto the buffer + Time required to load/store from the current row buffer =
     2 * ROW_ACCESS_DELAY + COL_ACCESS_DELAY.

4.2 Non-blocking Memory Implementation

4.2.1 Data Structures

In addition to those in part 1:

- A queue named **queuedRegs** which stores the ordered list of registers which are involved in current or upcoming lw/sw operations. Each entry is stored as an ordered pair of the form (string,int) where the string corresponds to the register name while the integer stores the the ending clock cycle of that register.
- An unordered map named **engagedRegs** which stores the currently engaged registers i.e. register overhead due to memory operations. Each entry in the map is of the form (string,queue of int) where string is the name of a register and the corresponding queue stores their end clock cycles in order of their appearance.

Although both the above data structures store the same information, both are required as the queue maintains the order in which the registers are freed, while the map is useful for efficient finding or deletion of an engaged/queued register along with its clock cycles.

4.2.2 Algorithm

- I implemented this part using two different methods, one more complex but more efficient than the other.
- **Implementation 1:** The first(less efficient) implementation simply tries to run normal instructions parallel to a running memory instruction, but stops as soon as another memory request is made. Consequently, the remaining time of the current memory operation is still lost as delay, which in worst case can be equal to the the complete time required for a memory operation, and such a case arises when there are consecutive lw/sw operations.
- **Implementation 2:** This is the more efficient (but more complex) implementation which works on the same principle as described above, in that it tries to schedule other normal instructions while a memory instruction is running in the background, but the major difference here is I have implemented an algorithm to 'bypass' adjacent lw/sw instructions, so that we don't have to waste the whole time in delays. For this, I have used a queue to maintain an ordered list of registers which are to be used for memory operations in current or upcoming clock cycles. This arrangement allows us to keep references of consecutive lw/sw operations, while still executing normal instructions simultaneously. The only time when we have to stop normal instruction from executing is when that instruction itself uses a register which is still engaged in a running memory operation.

- This algorithm also takes care of safe and unsafe instructions. Those normal instructions i.e. anything other than lw/sw are unsafe which themselves makes use of registers which are already engaged/queued in a DRAM operation. Adjacent

In both of the above implementations, three things remain invariant:

1. All instructions are executed sequentially i.e. even though some instructions seem to be bypassed, they are actually executed before the subsequent instructions.
2. Only one instruction can be processed at a time in a processor i.e. no two instructions actually take place on the same clock cycle which is why a DRAM request is executed in a separate clock cycle.
3. Only one instruction request can be processed at a time in a DRAM, so in both of the implementations above, I have kept every memory instruction separate. So, in first part, next DRAM instruction is executed only after a running DRAM task is completely over. While, in second part too, I have maintained the subsequent DRAM instructions in a queue for sequential but separate execution.

5 Testing Strategy

For testing both the parts, I have used the given test cases, both in the minor exam folder as well as from the minor preparatory folder. Additionally, I have manually prepared some test cases to handle various corner cases and to test various extents of difference in the efficiency of both the parts.

I also compared the number of cycles required for the complete execution of the program for both the codes, one with only DRAM while the other containing DRAM with non blocking memory.

In particular, I have used five types of test cases:

1. Test cases given in minor preparatory folder.
2. Test cases given in this assignment statement.
3. Manual test cases
4. Loop test cases
5. Corner cases

All types of cases yield a significant difference in the number of cycles of execution in the two types of code.

Example: Code with only DRAM: 241 cycles Code with non blocking memory: 199 cycles

6 Exception Handling

I have thrown exceptions/errors at various places, some of which are:

- Incorrect format for row and column delay values
- Defining an instruction set for labels twice.
- Defining label names as reserved keywords.
- Colon required at the end of label name.
- Invalid instruction(s).
- Jumping to an invalid label name.
- Register name not valid or doesn't start with $.
- Immediate values are not given as integers for addi.
- Incorrect or unaligned memory address for lw and sw.
- Memory limit exceeded.
- Syntax errors for inappropriate usage of commas, newlines, tabs or whitespaces, thrown while lexing the input file.

7 Features of the simulator

7.1 Strengths

1. With the primary goal of having better time efficiency, the code does fairly well by saving a lot of cycles as evident in the testing strategy. This is achieved by an efficient and modular scheduling mechanism which eliminates most waiting times for normal instructions, and also eliminates most of the desired delays.
2. It also satisfies all of the three invariants mentioned above, in that it achieves efficiency while simultaneously maintaining the sequence in which the program is executed.
3. This implementation of non blocking memory access also automatically takes care of identifying safe and unsafe instructions, and once identifying them, does further computation suitably.

7.2 Weaknesses

1. The code being complex and intensive, requires a lot of debugging and testing for establishing complete correctness.

2. For making the code time efficient, I have used numerous data structures, which may pose a problem in case of a very large input MIPS program. In such cases, the C++ program may throw an error for a large memory overhead. But according to the best of my ability, I have made sure that the code handles at least those memory constraints that have been provided i.e. 2^{20} bytes. In this way, the code is not so space efficient.

3. Because of the high amount of complexity in the code, it might be difficult to integrate it with hardware and optimising it to run on a particular hardware implementation efficiently can be even more challenging.