COL216: Minor

Himanshu Gaurav Singh, 2019CS10358

MIPS simulator with DRAM timing model

1. Program specification

a. Input Format:
A .txt file supposedly containing a sequence of MIPS commands and optional values of ROW_ACCESS_DELAY and COLUMN_ACCESS_DELAY(by default, 10 and 2).

On the terminal, "make all" generates the executables for both parts, "make nba" for Non-Blocking Memory and "make sdram" for Standard DRAM

b. Output Format:

For every cycle/set of cycle, the change in the state of the registers, the DRAM and the ROW BUFFER are displayed along with the instruction executed. At the end of the simulation, the total number of cycles, number of buffer updates and the frequency of each instruction is displayed.

c. Assumptions:

1. All the assumptions regarding input/output constraints in A3 are applicable.

2. Approach

A. Implementation of the DRAM

a. Memory elements

At the level of implementation, DRAM is modelled as a two-dimensional memory object. In continuation with my A3 implementation, the architecture is word-addressable and hence, has dimensions 1024*256 (note that this is just a description of the model of the DRAM).

The ROW_BUFFER is modelled as a linear memory object with dimensions of 1024 bytes(256 words). The index of the current row of DRAM placed in the buffer is maintained(-1 if uninitialized).

The first 2^16 words are reserved for storing the instructions in a row-major order. In case this bound is crossed an error "Instruction Memory overflow" is raised. Analogously an error "Data Memory overflow" is raised as well.

```
struct mem{
    int inst ;      //  inst=0 indicates data is stored , inst=1 indicated instruction is stored
    string kind; // stores type of instruction
    string label; // stores label in case of bne,beq and j
    int args[3]; // stores appropriate register number,
    //  offset (incase of sw and lw), constant (addi)
    // and 32 bit signed integer in case of data
};

vector < vector < mem > > DRAM ;
//   vector<mem>Memory;
int current_row_in_BUFFER = -1;
vector<mem> ROW_BUFFER;
```

Code snippet, modelling the memory

b. READ-WRITE instructions and timing characteristics

The steps of READ and WRITE as mentioned in the problem statement have been implemented word-by-word by manipulating the DRAM and the ROW_BUFFER. The cycle-count is manipulated step by step as the partial steps of the execution proceeds.

An offset is added to the memory address(at the level of implementation) to keep the space for instructions unaltered.

```
void copy_to_dram()
{
    DRAM[current_row_in_BUFFER] = ROW_BUFFER;
    clock_cycles+=ROW_ACCESS_DELAY;
}
```

BUFFER copied to DRAM(in case a different request is initiated)

```
void copy_to_buffer(int x)
{
    if(x != current_row_in_BUFFER)
    {   cout<<clock_cycles+1<<"-";
        if(current_row_in_BUFFER != -1)copy_to_dram();
        buffer_updates++;
        ROW_BUFFER = DRAM[x];
        current_row_in_BUFFER = x ; //cout<<x<<"\n";
        clock_cycles+=ROW_ACCESS_DELAY;
        cout<<clock_cycles<<":"<<"Buffer updated to row "<<x<<" of DRAM\n\n";
    }
}
```

Ensures that the BUFFER contains the required row after (2*ROW_DELAY+COLUMN_DELAY) or (COLUMN_DELAY) as the case maybe.

```
int read(int x)
{

    copy_to_buffer(x/256);
    cout<<clock_cycles+1<<"-";
    clock_cycles+=COLUMN_ACCESS_DELAY;
    return ROW_BUFFER[x%256].args[0];
}

void write(int x , int z)
{
    // copy_to_dram();
    copy_to_buffer(x/256);
    cout<<clock_cycles+1<<"-";
    clock_cycles+=COLUMN_ACCESS_DELAY;
    ROW_BUFFER[x%256].args[0] = Register[z];
    cout<<clock_cycles<<":"<<"Row buffer corresponding to DRAM row "<<x/256<<" at column "<<x%256<<" is updated
        to "<<Register[z]<<"\n\n";
}
```

The READ and WRITE procedures

B. Non-blocking memory access

This part is aimed at improving the timing parameters of the simulator by exploiting the parallel execution of "load" and "store" instructions on the DRAM and the other instructions in the Register File.

This scope of improvement has several constraints related to dependencies of the Register File operations on the DRAM operations and vice versa.

Dependency related constraints:

1. Before the execution of a register file instruction, all the "load" and "store" instructions involving the registers same as that in RF instruction that occur before it should be executed on DRAM.
2. All store and load operations of the file should be executed in the exact sequence on the DRAM.

Solution :

The second constraint of sequential execution on DRAM motivates a QUEUE data structure to be used for DRAM instructions.

The Algorithm :

Concretely stating,

1. The program counter iterates over the instructions in the file and maintains a queue of all the unexecuted load and store instructions that have occurred till then.

2. As the program counter moves to a (new) instruction and finds it to be a logical/arithmetic instruction, it checks if this instruction has some unresolved dependencies(in terms of unexecuted corresponding load and store instructions).

3. If such an issue exists, then the instructions in the queue are dequeued and executed on DRAM till all such dependencies are resolved(i.e. all preceding "load/store" instructions that had registers overlapping with the current instruction are executed.

4. The current instruction is executed.

5. If the current instruction is a “load/store” instruction, then the instruction is enqueued into the queue.

6. During the whole execution , the DRAM is idle only if the queue is empty, otherwise the load and store instructions are being executed in parallel with register file instructions.

Advantages of the Approach. :

1. The DRAM sits idle if and only if there are no more “load\store” instructions to execute. This improves timing efficiency.
2. The choice of the data structure is directly motivated by the execution constraints. Evidently, the algorithm provides the **best “worst-possible” running time** of execution.
3. The above approach disallows the possibility of simultaneous “store “operation and an arithmetic/logical operation, which avoids collisions.

Weaknesses in the Approach :

1. It is a general programming practice to ensure that the registers storing memory locations and those storing data are kept separate throughout.(Although, not strictly required). If this is assumed, we can remove the dependencies of the arithmetic/logic instructions on the “store” instructions as well. In this case, the only dependency in relation to the register instructions is the one to “load” instructions.

2. If there are consecutive “load” instructions in the file with the same register as destination, a more clever simulator will load once according to the last instruction. Similar is the case if there are several consecutive “store” instructions with the same memory locations as the destination.

3. The algorithm necessitates the use of extra space for maintaining the queue data structure and also for efficient resolution(O(1) per instruction amortized) of dependencies.

Testing Strategy

1. Small sets of instructions are executed with the first part of the program and the total number of cycles and the states of the register file , DRAM and ROW BUFFER are verified by matching with a manual dry run. Example : *testcase1.txt , tc3.txt, tc4.txt , tc5.txt,tc6.txt.*

2. Randomly generated test-cases are run on both the standard DRAM implementation and the one with non-blocking memory access to ascertain the difference in the timing parameters with the states of the registers and main memory being same. Example : *random_input.txt, random_input_large_memlocs.txt, random_input_mixed.txt, input_medium.txt.*
3. The random testcases are generated by a minimally modified version of the file “helper.cpp” submitted by my pair with A3.

Observations from Testing

It is observed that the approach gives sufficiently better results for randomly generated testcases and the magnitude of advantage increases with increase in size of instruction

set. However, an adversary can create a testcase in which the advantage given by non-blocking access won't be that prominent. Nevertheless, such cases will be rare in practice.
Note: The code base for the assignment is the submitted code for A3 by my pair.