MEMORY REQUEST REORDERING

Anirudha Kulkarni : 2019CS50421

Pratyush Saini : 2019CS10444

Input:

MIPS instruction set with add, addi, sw and lw operations

Build Instructions:

Program can be run easily with make instructions:

1. To Build: `make build2`
2. To run executable from build `make run2`
3. To remove build file `make clean2`

All these steps can be performed by `make nba` which will remove previous build, create new one and give output.

Custom `ROW_ACCESS_DELAY` and `COL_ACCESS_DELAY` can be provided by navigating to `build` directory and

```
./main ../input.txt 100 45
```

which will execute the program with `ROW_ACCESS_DELAY` as 100 and `COL_ACCESS_DELAY` as 45 Provide input in input.txt or give it as command line argument. Defaults used when `make all` is used are:

```
`ROW_ACCESS_DELAY`: 10
`COL_ACCESS_DELAY`: 2
input location : 'input.txt'
```

Code with DRAM Reordering can be run by:

```
make dram
```

Approach:

We used the same DRAM Implementation as In Minor Examination. The Memory block being 2 dimensional 1024 x 1024 sized memory block.

The first half of the DRAM space is reserved for storing instructions from input MIPS assembly file. The next half of the space is reserved for storing data values in Memory (where each data value occupies 4 units of space).

We redesigned our DRAM implementation to handle the problem of repeatedly writing and loading rows from DRAM to the Row Buffer.

We implemented our idea using a Queue Data Structure.

Algorithm:

We start executing all the statements sequentially. According to the type of instruction seen:

i) If the statement is lw/sw type, we push the statement to the Queue, wherein the loading / writing process starts from the DRAM memory unit.

ii) If the statement is ALU type instruction (except lw/sw), we first check if this statement is dependent on the statement in the waiting queue of DRAM. In case this is true, we perform the function complete_dram_activity (it's semantics are mentioned in the next unit), and then execute this instruction when all it's dependencies are resolved.

iii) In case the current statement is independent of the instructions in the waiting queue of DRAM, we execute the instruction in place and increment our Program counter.

Simple DRAM implementation vs Non Blocking Implementation-

We implemented the Simulator with both Simple DRAM implementation (as in Part A of Minor Examination) and NBA (as in Part B of Minor Examination).

In our NBA implementation, the execution of the instructions present in the queue occurs in parallel with the ALU type instructions, using which we were able to reduce the number of cycles by a fair count.

For example there are two instruction present in the waiting queue of DRAM:

```
lw $t0, 1000

sw $t1, 1028
```

In our Simple DRAM approach, we would issue two DRAM requests in the first two cycles and then start the loading / storing process from the third cycle.

While in our NBA Implementation, we would rather issue a DRAM request of first instruction and start it's activity in DRAM from the second cycle. The DRAM request for second instruction is issued in the second cycle and it's activity starts in DRAM after the activity of first instruction is completed.

Resolving Dependencies (Complete_DRAM_Activity Function)

If we encounter some dependent statement while sequentially executing the program, we execute the instructions in the waiting queue of the DRAM.

To increase the efficiency , we order the execution of requests in the waiting queue of DRAM, subject to the condition that we do not compromise with the semantics of our Simulator. We perform the recording based on the following approach.

1. Identify the requests in the DRAM waiting queue which are independent of the request preceding it in the queue and store them separately, which maintain the dependent one's in a separate data structure.
2. We first fulfill all the independent requests for the row already loaded in the DRAM and check for the corresponding dependent request for the same row (if any of those dependent ones gets resolved now after fulfilling the previous request).
3. We then write back this row and load other rows in the Row buffer and perform the same operation, till all the independent requests get fulfilled.
4. After this, we are only left with the dependent requests, which are executed sequentially.

Checking dependency

For the lw type instructions, if any instruction succeeding it either reads or writes value in the destination register of this lw instruction then it is considered a dependent statement.

For the sw type instructions, if any instruction succeeding it writes value in the destination register, it is then considered a dependent statement , other independent.

Checking dependencies among requests waiting in the queue while reordering:

For any sw type instruction, if some instruction following it tries to access the same memory location corresponding to the current sw instruction, then the instruction is considered a dependent statement.

For any lw type instruction, if some instruction following it tries to access the same destination register corresponding to the current sw instruction, then the instruction is considered a dependent statement.

Empty the Waiting Queue or Just Resolve dependency?

We have two approaches for the DRAM complete activity function. Either we could wait till all the requests in the waiting queue in the DRAM gets fulfilled beforing moving forward, or just fulfill those requests which solve the dependency of the current instruction (corresponding to PC).

Both approaches have their own strengths and weaknesses. If we just solve the dependencies, we do not have ample power to handle the lookaheads so it might create conflicts in the future and unnecessary increase clock cycles.

If we fulfill all the requests in the waiting queue, we might need to load/write back rows proportional to the number of times we call the complete_dram_activity function.

Assumptions:

1. Registers are accessed only during `COL_ACCESS_DELAY` duration
2. `ROW_ACCESS_DELAY` > `COL_ACCESS_DELAY` and both are non zero
3. Instructions dont consume cycle to be accessed from memory
4. Maximum instructions and data values are 2** 19 so that overall memory will not exceed 2 ** 20

Strengths:

1. The DRAM sits idle if and only if there is no independent instruction remaining and that can not be optimized. 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.

Weakness:

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