Computer Architecture COL216

MIPS Simulator - Assignment 5

Arpit Chauhan, 2019CS10332
Aman Gupta, 2019CS10673
17/5/2021

Description

In this assignment, we have written a C++ program that reads a MIPS assembly language program as input and executes it as per MIPS ISA by maintaining internal data structures representing processor components such as Register File and Memory. We have developed a model for the Main Memory which can be thought of as a 2-dimensional array of ROWS and COLUMNS with DRAM timing values ROW_ACCESS_DELAY and COL_ACCESS_DELAY in cycles. The memory access has been made non-blocking so that subsequent instructions don't always wait for the previous instructions to complete . DRAM Request Manager has been incorporated in this assignment to efficiently handle the execution of multiple input programs at the same time in Multicore Processors .

Input / Output

- The input to the program will be N text files (equal to number of cores) each containing a MIPS program , along with 3 arguments , ROW_ACCESS_DELAY, COL_ACCESS_DELAY and M (Simulation Time in cycles) .
- Each instruction will be in a new line.
- The allowed instructions are: **add, sub, mul, beq, bne, slt, j, lw, sw, and addi.**
- If any instruction does not follow MIPS convention, then a corresponding error is thrown along with the line number.
- If all the instructions are syntactically correct, then the program starts executing the operations starting from the first instruction.
- The labels must also follow MIPS convention i.e. must start with a letter, maybe followed by letters, numbers, or "_", must be terminated by ":" and should be unique.
- At every clock cycle , we print the executed instruction , modified registers(if any) , Data Memory , and activity on the DRAM such as row buffer updates .
- After the execution completes , relevant statistics such as total execution time in cycles , total number of row buffer updates , Data memory used , Instruction memory used etc are consoled out along with the Integer Registers and Data Memory.

Approach / Algorithm

- Each instruction occupies 4 bytes and is executed in one clock cycle except for lw and sw .

- For lw(READ) and sw(WRITE) operations , activating the corresponding row takes zero, one or two times ROW_ACCESS_DELAY cycles depending upon the current row in the row buffer . Then copying/updating the data in the row buffer takes COL_ACCESS_DELAY cycles.
- But the program does not wait for the lw/sw instruction to finish , it continues executing the following instructions unless there is some dependency on the ongoing lw/sw instruction.
- While some lw/sw instruction is being processed in the DRAM , if the program encounters another lw/sw request which has no dependency on any previous instruction not completed yet , the new DRAM request is pushed into a **Request Manager** which then does the reordering and decides which instruction is to be passed into DRAM next for execution.
- In the request manager , we have separate **finite sized queues** for a maximum of 16 rows at once i.e. each queue contains instructions belonging to the same row.
- sw-lw **forwarding** and sw-sw/lw-lw **redundancy** are checked in the Request Manager and handled there itself accordingly .
- We maintain an iterator named **Program Counter**: pc, that always contains the address of the next instruction to be executed (one for each file). This counter is updated after each instruction execution.
- The program ends after the Simulation time (M cycles) even though all the instructions may not have completed .

--- <u>Strengths</u>

- Whenever a lw/sw request is being processed in the DRAM , the remaining queue is **reordered** such that any instruction which requires the same row buffer is executed next on priority basis.
- This design does not copy back a row if only read operations are performed on it thereby saving a significant amount of time. This is done using a **dirty bit**.
- This design also checks if we can reorder the queue such that any **blocking register gets freed earlier** than usual to save time.
- **Redundant sw-sw** and **lw-lw** instructions are removed from the Dram Queue to reduce the execution time.
- **Forwarding** of **sw-lw** instruction is done in the Memory Request Manger.
- The model is implemented keeping in mind **hardware constraints** like finite size of memory queue and two read, one write port of register file.
- Separate part of total Memory is provided to each core to **avoid corruption** of data of one core by another. Though, total memory is fixed.
- The queue is reordered during the execution time; **no additional lookahead** or storage is needed at the time of compilation of the code.
- The simulator outputs a **detailed log** of the execution of the MIPS program which is very useful for the user.

--- <u>Weaknesses</u>

- Due to the finiteness of the Dram queue, after filling 16 different rows, the request manager will stop accepting instructions until any row is freed.
- A single core can lead to filling most of the rows of the queue thereby causing starvation as instruction of other cores may not be added.
- There might be cases when executing the current blocking register row of the queue would lead to a larger throughput.
- The latency of individual memory instruction may increase, though the overall throughput decreases.
- Memory Request Manager may take up quite a significant amount of time during which execution might stall.
- Use of lookahead during runtime or additional storage during compilation might help in reduction of total clock cycles, which is not considered in this design.
- MIPS convention only accepts Immediate value upto 2^16 only, which makes the memory beyond the limit inaccessible.
- The design can be made even cheaper and more organised by making the code modular so that it is easier to test and refactor.

<u>ERROR HANDLING</u>

All errors are reported with the appropriate line number and reason for the user to debug.

1. **Syntax Error**
   A syntax error is thrown in the following cases:
   - If an invalid instruction is provided to the code.
   - If the operands of any instruction do not follow the convention specified by MIPS ISA.
   - If the integer value of an I-type immediate value is larger than 2^16 bits.
   - If a label is not in the specified format or is repeated twice.
   - If the total number of instructions exceeds the allotted space for Instruction memory.

   Note: The extra whitespaces or indentation are ignored by the program
2. **Execution Error**
   An Execution error occurs in the following cases:
   - The instructions beq, bne, or j asks for a label that has not been initialized.
   - If the data memory exceeds the allotted space for data. Or asks for an invalid memory address. (Like not a multiple of four or negative)
   - If an instruction tries to modify the value of the zero register.

TESTING

- We have done extensive testing as a part of the assignment to check the validity of the MIPS simulator. The test cases have been added to the Zip folder.
- All the allowed instructions with labels are included in the test cases to check the validity of each of them.
- Details of execution performed in each cycle along with the state of Registers and Data Memory are stored in the output which help to check the validity of the MIPS simulator.
- We have attempted to handle syntax errors and execution errors of almost every type and tested them rigorously.
- **TEST CASES DOCUMENTATION:**
  - T1 - Multi-core functionality with cores having all types of Instruction
  - T2 - Checking blocking/Non-blocking nature of processor.
  - T3 - Validating characteristics of processors on inputs with branching
  - T4 - Checking correctness of output on very large test cases
  - T5 - Multi-core functionality with non-memory instructions
  - T6 - Checking lw-lw/sw-sw redundancy and sw-lw forwarding
  - T7 - Ensuring hardwares constraints like finite queue size, maximum of one write and two read on a clock cycle in one core, etc. remain true.
  - T8 - Ensuring finiteness of individual queue.
  - T9 - Monitoring throughput efficient, delay in memory manager and throughput- latency trade off.
  - T10 - Edge test cases (Latency Increases)
  - T11 - Analysing the tradeoff between which operation should MRM perform first assigning to DRAM or adding an issued request.
  - T12 - Gain-loss of assigning rows of executing instruction based on current blocking register.

X----------------------------END-----------------------------------X