INDIAN INSTITUTE OF TECHNOLOGY DELHI

COL216

Report

Minor

MIPS simulator with DRAM timing model

Abstract

A simulator is a software that emulates the actions of an entity without actually utilising the entity. Here we attempt to create a cross platform MIPS simulator that emulates all the hardware instructions supported by MIPS. This simulator takes as input a MIPS assembly program that translates it into instructions executed by MIPS.

Harsh Agrawal | 2019CS10431

Contents

1 Modules

We implemented a simulator in `C++` for executing MIPS instructions from assembly code. The program was divided into `compiler` ,`hardware`, and a `DRAM` module. These modules perform different functions as outlined below

1.1 Compiler

- This module takes as input the assembly program and parses the tokens according to the syntax specifications of MIPS assembly programs.
- This module assembles the program into a sequence of instructions that can be executed according to the hardware specifications.
- It also generates appropriate syntax errors if an erreneous expression or an unidentified token is encountered.
- The compiler checks the encodibility of instructions into 4 bytes (without actually encoding it) using the size of operands involved.

1.2 Hardware

- This module emulates a subset of instructions provided by MIPS. They are `add, mul, sub, slt, addi, bne, beq, j, lw, sw`.
- The module maintains a record of all the register and memory values and modifies them according to the instruction specifications.
- It generates appropriate exceptions on performing prohibited actions like out of bounds memory access and reserved register access.

1.3 DRAM

- This module implements the DRAM model in the form of a `2D` array.
- It contains a row buffer to simulate an actual `DRAM`. A row is first copied into this row buffer and the module uses the buffer as a write-back cache for further read and write operations.

2 Approach

The approach used to simulate the DRAM timing model is described in this section

2.1 DRAM

- The `DRAM` maintains a `2D` array to simulate the memory and stores a row buffer array to simulate the cache memory.
- In case of a `read` operation if the memory address belongs to the row stored in the row buffer, a column access can yield the result. However in the other case, the row buffer needs to be written back to the actual memory consuming a time equal to the row delay of DRAM. The new row is loaded into the row buffer and the address is then served from row buffer using column access.
- Similarly, the `write` operation involves writing to the row buffer if the row of the address matches the row-buffer else a write back is required.

2.1.1 Advantages

- Abstracting out the `DRAM` methods in a separate module helped modularise the code and an easier implementation of the `lw, sw` instructions
- The row buffer in `DRAM` helps to reduce the access time of memory addresses by caching an entire row. This reduces the time taken to access nearby memory addresses consecutively.
- Implementing `DRAM` in the form of a `2D` array helped implement a cache mechanism. Storing the data linearly increases the complexity by deciding on the appropriate data to be cached for better performance.

2.1.2 Disadvantages

- Access of several distant memory addresses consecutively leads to an additional overhead of writing back the memory buffer and reloading it with a fresh row. Lots of cache misses can cause the implementation to be slower than a standard no-cache implementation.

- Copying a row into a row buffer seems like an unnecessary overhead as `C++` already provides constant time random access to any element of the `2D` array. Although random access in a `2D` array could speed up our simulator significantly but it would foreshadow the actual hardware implementation of the `DRAM`.
- The row buffer implemented as a write back cache does not include a dirty bit for the cache. If the dirty bit is unset the DRAM can skip writing back the row buffer to memory.

2.2 Non Blocking processor

A non blocking processor does not block on data that requires a bus read-/write. In this case the `lw, sw` instructions are responsible for storing and loading from the main memory(`DRAM`). This involves a data transfer over the bus and the processor need not block on bus operation to execute instructions that only require register computations.

2.2.1 Specifications

- When data is loaded/stored from/to the main memory the processor always attempts to execute the next instruction in the cycle immediately following the issue of a DRAM request
- If the next instruction does not involve any register used in the `lw, sw` instructions, the instruction is safely executed.
- If the next instruction is itself a `lw, sw` instruction the processor blocks until the previous bus transfer is complete, irrespective of the registers and memory addresses involved.
- If the next instruction involves a register that is to be loaded from the memory, the processor blocks, waiting for the bus transfer to complete.
- However if the register was simply used to store a value into the main memory; it is not necessary to block, the processor continues execution.

2.2.2 Advantages

- A non blocking processor reduces the number of clock cycles to execute a set of instructions, reducing the average `CPI`. Instructions invloving a bus transfer do not block enabling other instructions to be executed.
- The non blocking processor is very efficient for programs that consist majorly of register computations and less frequent bus transfers. It allows execution of `lw, sw` instructions in a single clock cycle which would otherwise consume large clock cycles in blocking mode.
- The processor blocks on instructions that use registers to be loaded from a pending `lw` call. This ensures that the program output matches with a program executed in blocking mode.
- The optimization to not block on register used as the source of `sw` instruction reduces unnecessary blocking of the processor.
- The approach can never be slower than a blocking program, i.e. there are no additional overheads introduced in the implementation.

2.2.3 Disadvantages

- The approach is equivalent to a blocking program in case several `lw, sw` instructions are executed consecutively.
- The approach does not group consecutive `lw, sw` instructions that are unrelated to each other as a single parallel instruction, instead it waits for each one of them to complete before the next can be executed.
- A further optimization could help in case a register is consecutively loaded from `lw` instruction followed by a register operation. The `lw` instruction could be eliminated completely as the next register operation overwrites any value loaded from the memory.
- The memory accesses could be rearranged to always access the most nearby address thus reducing the additional overheads incurred by cache misses in `DRAM`. This involves reordering of unrelated instructions.

3 Testing

The testing method was completely manual. The test cases written manually are included in the `./tests/input` directory. The test cases were run with different parameters of the program and the outcomes were compared.

- `load_and_store.in` : This test case includes consecutive load and store instructions with different offsets to test the correctness of the `DRAM` implementation.
- `load_and_use.in` : This test case loads a value from the memory and uses it in the next instruction. This tests whether the program blocks for pending load instruction when a register is to be used.
- `load_continuous.in` : This test case was designed to execute `lw` instructions in a single clock cycle. This test case invloves multiple unrlated arithmetic operations after a load instruction. This test cases whether the program does not block when not necessary.
- `store_and_use.in` : This test case stores value from a register to the main memory and then uses it in the next operation. Here no blocking is required and the register value is safe to use without blocking for `sw` to complete.
- `store_continuous.in` : This test cases executes multiple arithmetic operations after each `sw` instruction and checks wheter `sw` are executed in a single clock cycle.

3.1 Result

The program output was observed and all the register, memory and row buffer values were inspected for correctness manually. In each of the test case the expected outputs matched. The program blocked when necessary and executed parallely when blocking was not required.

The output logs of the test cases are available in the `./tests/output` directory. The test cases were run with different configuration options and each test case's blocking and non blocking version is included.

4 Assumptions

- We have assumed that the maximum memory available to any user program is 2^{20} bytes. This does not include the memory required for storing instructions.
- We have assumed 32 registers for MIPS each of them storing 32 bits. We have hardwired the register `$0` to the value 0 as is the case in MIPS hardware. We have also restricted use of kernel reserved registers `$26, $27`.
- We have assumed that exactly 1 clock cycle is required for each of the instructions except `lw, sw` to display the execution statistics.
- We have enforced tight syntax rules to enforce good coding practice. We disallow certain instructions like (a) `lw $5, ($29)` and (b) `lw $5, 40231`. The first instruction skips supplying the offset to the register and the second one involves raw memory access.
- Our implementation also enforces using numeric registers instead of their named counterparts. For example `$5` is allowed but its named counterpart `$a1` is disallowed
- We have enforced the use of a whitespace between the operand and its arguments. For example `add$4,$5,$6` is disallowed as there is no whitespace between `add` and `$4`
- We have implemented branching instructions like `beq, bne, j` to take only labels and instruction numbers as the last argument.