Modified Pipeline MIPS Processor with delayed branches ad multi-cycle memory access

Anish Teja Bramhajosyula (IMT2024029) & Akshay KM (IMT2024014)

EGC - 121
Computer Architecture

Note

This is a two-file project. One is the `src.py` file which has the source code which implements the modified MIPS Pipeline design. The other is the `program.asm` file, which contains the MIPS code. It will be read by the Python file.

This project is intended to showcase the core concept of branch delay handling, prediction, resolution and memory access cycles demonstration. Hence, only a select few instructions (pseudo and otherwise) are supported. Also, the registers are numbered and not in their usual conventional nomenclature.

`halt` is used to indicate the end of the program. This is not part of the MIPS standard. It has been incuded for our convenience. It may trigger compilation errors in MIPS compilers.

The `random` module in Python has been used to generate random numbers.

The `Processor` class

This is the only class in the program which contains data members to hold various parameters and member functions to simulate the flow of a processor. Note that this is an abstraction and is only a depiction of a portion of the functionality of the actual MIPS processor.

The following are the data members of the `Processor` class:

1. `regs` - A list of the values for 32 registers
2. `ins_mem` - Instruction memory
3. `data_mem` - Data memory
4. `pc` - Program counter
5. `cycles` - Keeps track of the total clock cycles
6. `branch_taken` - Flag to check if a branch is taken
7. `stall_next_cycle` - Flag to check if a `NOP` should be inserted in the next cycle to stall
8. `if_id`, `id_ex`, `ex_mem`, `mem_wb` - Pipelined registers
9. `stall_cycles` - Keeps track of the number of stall cycles
10. `instructions_executed` - Keeps track of the total number of instructions executed
11. `load_stalls` - Keeps track of the number of load stalls introduced

12. `branch_count` - Keeps track of the number of branch statments executed

13. `mem_delay_cycles` - Keeps track of the number of memory dellay cycles introduced

The follwing are the member functions of the `Processor` class:

1. `load_program()` - Reads the `program.asm` file and turns each line a string and stores the program in a list. It also parses through the read lines and collects addresses for branching and/or jumping. In the next pass, it collects all the instructions and loads them onto the instruction memory

2. `if_stage(stall : bool)` - Fetches the instruction from the previously loaded instruction memory and modifies the program counter. The process is carried out, keeping in mind whether or not there is a stall.

3. `id_stage(stall : bool)` - Introduces a stall if there is a branch statement and uses the delay slot fore executing the very next instruction. It parses each line and sets the `$rs`, `$rt`, `$rd` and immediate value (whichever is applicable).

4. `ex_stage()` - Checks for the instruction for every line and the register values. It then performs the appropriate operation. If the instruction requires data memory access, i.e., it is a `lw` or a `sw`, then we generate either **2** or **3** randomly and stall the processor for those many cycles, until the memory access is complete.

5. `mem_stage()` - Checks if all the reseved memory cycles have been exhausted and stalls if not. It passes on the calculated values to the appropriate registers. It also writes to the data memory for `sw`, and reads from the memory for `lw`.

6. `wb_stage()` - It writes back to the appropriate registers for operations requiring it.

7. Additional functions are used to print and test the output and also show the statistics (which includes total cycles, branch cycles, memory access cycles, etc.). The `run` function executes the pipelined stages in reverse. This is because, during a single clock cycle, the write operations are executed in the first half, and the read operations are executed in the second half.