Datapath Simulator

Ufuk Arslan & Selman Berk Özkurt

RV32I Emulation

We built a RV32I instruction emulator first, to trace the execution, using which we can infer the hazards. Data memory and instruction memory are implemented separately, the latter being immutable by the emulated processor itself. Other than that, we can fully emulate a RISC-V core with its registers and full data memory (taking into account alignment and endianness issues as well) without an operating system.

Input File

We did not program a fully functional assembler becuse it is beyond the scope. What we did was a simple parser that takes RISC-V instructions in different lines and stores them in a virtual instruction memory in the emulator. We do not support pseudoinstructions, empty lines, labels and comments in our input code. Yet, it is possible to easily implement many algorithms. Notice that all forms of register naming is implemented to make programming easier. One example input file (`inp.a`) is provided.

Usage

In a terminal type `python main.py inputfile` to run the code for `inputfile`. The results will be written to the `stdout`. Python 3 was used during development.

Dump Format

In order to make analysis of the execution more explicit, in accordance with the main aim of this study, All the instructions emulated are tallied with relevant information such as the instruction, source and destination registers and their values, other arguments, program counter for that instruction, the clock cycle and the number of stalls for that instruction.

Hazard and Timing Considerations

There are three types of hazards we may face when we are creating a pipelined structure. Remember we assume *no branch prediction* and data *forwarding*.

Structural Hazards

They occur when our hardware can not handle some combinations of instructions simultaneously, which means that we have resource conflicts. We do not take these cases into consideration in our project.

Control Hazards

They occur when we have instructions that change the PC, the main reason of this hazard is the pipelining of branches. Noting that we did not implement any branch prediction techniques, we considered the best way we can use without them. So, we need to know whether the branch is taken or not as early as possible. Inside ID step if we find out that we have a branch type instruction, we use a simple adder at hardware level to compute the effective address of the target instruction of the branch. We also include a simple comparator at ID step so that we find the result of whether the branch is going to be taken before going through ALU in the EX step. These are the optimizations we can do at hardware level, so in normal cases we need to add ONE STALL after each branch type instruction. However, since we decide the result of branch instructions at ID step, when the registers we use have dependencies over instructions coming before them, after forwarding, we are forced to do the comparisons at EX step, which means that we need to add TWO STALLS after those types of branch instructions.

Data Hazards

They occur when previous instruction has dependancy on the current instruction. This causes conflicts in the registers or memory where we read or write data at cycles that we are no supposed to do. There are 3 cases which may cause problems and they are:

- **RAW (read after write):** Consumer instruction reads the data before producer instruction writes that data. These are the most common data hazards, but luckily we can overcome most of them by using forwarding. So, we don't need to worry about all the cases of RAW when we are adding stalls. There is one specific case we need to handle and that occurs when we load data from memory to a register, then use that register in the next instruction, and for this case we need to add ONE STALL between those two instructions.
- **WAW (write after write):** First instruction writes after second instruction writes to the same memory or register. This case does not happen in the pipeline structure we use.
- **WAR (write after read):** Latter instruction writes the data before former instruction reads that data. This case also does not happen in the pipeline structure we use.

Tallies Calculated

After printing all the execution as decribed in the dump format section in addition to the memory and registers, the program calculates and prints number of instructions executed, number of clock cycles and number of stalls added.