CS2323- Pre-Final Status Report

Name- Vemula Siddhartha
Roll number- ee23btech11063
No group, done solo.

Previously Implemented (Summary):

- Previously, a single-cycle RISC-V 32/64 I processor was implemented, only for R-type, I-type and S-type instructions.
- A Data Memory was emulated using Verilog code, to verify the working of load and store instructions.
- Implemented ALU, Control Unit, Immediate generation block and a Register file.

Newly Implemented (Summary):

- The previously implemented single-cycle processor was first extended to support the whole Base Integer Instruction Set. Newly added support for B-type, U-type and J-type instructions.
- An Instruction Memory was emulated using Verilog, and the working of branch and jump instructions was verified appropriately.
- The Processor was then divided into five stages and a pipelined implementation was then adapted.
- The pipeline followed is the Standard RISC-V 5 stage pipeline, taught in the class and explained in the book (Patterson's Computer Architecture).
- To implement a pipelined processor, 4 pipeline registers are placed between the 5 main stages of the processor, to decrease the clock time period, and maintain throughput of around 1 instruction per cycle. The latency of each instruction, however, increases to 5 cycles.
- The working of all the different types of instructions was first verified, without any forwarding or hazard detection. The results matched the ones obtained from Ripes (Mode: 5-stage processor with no forwarding or hazard detection/elimination).
- Then, a forwarding block was implemented, which is used for eliminating the stalls usually needed for consecutive R/I-type instructions with dependencies. Its working was verified by matching its results with the one obtained on Ripes (Mode: 5-stage processor w/o hazard detection).
- Then, a hazard detection block was implemented. This block gives two outputs, `stall` and `flush`. `stall` is outputted when the registers have to stall. This is useful when a load instruction loads a value to a register which is operated upon in the next instruction. Without this block, a nop should be added to avoid wrong results, but with this block, no nop has to be added by the programmer,

the hardware automatically detects this and stalls one cycle. `flush` is outputted when the values in the pipeline registers are to be flushed. This is useful for branch/jump instruction, since the values in the pipeline registers have to be discarded and have to wait till the branch condition is evaluated. Both these outputs are given as inputs to the IF/ID and ID/EX pipeline registers, as well as the PC register.

Verification Procedure:

- The disassembled instructions are taken from Ripes and are written to `instructions.txt` file.
- Then, the testbench writes the instructions from this file to the instruction memory (emulated as of now).
- At the start, a reset signal is also asserted, to initialize all the registers.
- The data memory is also initialized at the start of the testbench simulation, using an external file which gives the initial state of the data memory.
- Then, the reset signal is deasserted, and every clock cycle, a PC value is inputted to the Instruction Memory block, which gives an instruction every clock cycle, based on the PC value.
- The processor is made to run for a given number of cycles and the simulation is stopped.
- The registers are printed every cycle, and the output is verified against the output obtained from a similar processor configuration on Ripes.
- A `run.sh` bash script is used which includes all the Verilog module files to be included for simulation and runs the Verilog codes.
- A vcd dumpfile is also used to verify the waveforms obtained for different instructions using *GTKWave*.

Module Explanation:

- `pc_reg`: This module is the register which stores the value of PC.
- `pc_next_mux`: This module is a mux which selects what the next value of PC is supposed to be. Based on the select inputs, it either chooses PC+4, or PC value to which the program is supposed to *branch* or the PC value to which the program is supposed to *jump* to.
- `pc_4_add`: This module is an adder which just adds 4 to the current PC value.
- `instruction_mem`: This module is used to emulate the instruction memory.
- `if_id_reg`: This module is the pipeline register between the IF and ID stages of the processor.
- `reg_file`: This module is the register file. Contains registers `x0` to `x31`, and two read and one write ports.

- `imm_gen`: This module is the immediate generation block. Generates the immediate value required based on the type of instruction.
- `control_unit`: This module is the control unit. This gives control signals based on the type of instruction and the operation to be performed by all other components. This is sort of the *brain* of the processor.
- `id_ex_reg`: This module is the pipeline register between the ID and EX stages of the processor.
- `alu_op1_mux`: This module is a mux which chooses between `rs1` and PC based on the instruction.
- `alu`: This module is the ALU block. This performs all the arithmetic and logical operations (obviously :)).
- `branch_taken`: This module finds whether the branch is to be taken or not. It finds whether or not to branch based on the register values and the type of branch instruction.
- `pc_jump`: This module finds the effective value of PC if a jump is encountered.
- `pc_add_imm`: This module finds the effective branch address if a branch were to be taken.
- `ex_mem_reg`: This module is the pipeline register between EX and MEM stages of the processor.
- `data_mem`: This module emulates the data memory.
- `mem_wb_reg`: This module is the pipeline register between MEM and WB stages of the processor.
- `wb_mux`: This module is a mux which selects between ALU result and Memory Read data based on the type of instruction. This outputs a value which is the value to be written back to the destination register.
- `forwarding_unit`: This module implements the data forwarding procedure. This checks for dependencies and gives control signals as to which value is to be forwarded and to which register. Outputs two control signals, one for `rs1` and one for `rs2`.
- `forward_mux`: This module is a mux which selects which value goes to `rs1` and `rs2` based on the control signals given by the `forwarding_unit` block. The outputs are the forwarded data values.
- `hazard_unit`: This module is the hazard detection/elimination block. It gives `stall` and `flush` outputs which tell the pipeline registers (IF/ID and ID/EX registers) and PC register to stall or flush their contents whenever there is a hazard detected and applicable.

Further Implementation:

- Streamline the disassembly procedure.
- There are many redundant vectors/bits which are being propagated through the pipeline registers. They will be removed wherever necessary.
- There seems to be a bug while forwarding for store instruction dependencies. This is the first priority target.
- Flash the processor onto the FPGA and verify its hardware working. Get its resource utilization and timing reports.

Expected Challenges:

- FPGA :(
- On a serious note, the processor might not work as expected on real hardware. I will have to debug the problems. This debugging will take considerable time.
- Come up with a foolproof set of instructions, which verify the proper working of the processor.

Code:

The code to both the single-cycle processor and the five-stage pipelined processor can be found on my GitHub.

github.com/vs00007/RISC-V-Processor

The codes are (I tried to) mostly commented to explain the working of each of the modules.

Final Remarks:

- Most of the features proposed initially have already been implemented. I plan to work on integrating the M-extension too, though it may be challenging at this point.