University of Connecticut
CSE 4302 / CSE 5302 / ECE 5402: Computer Architecture
Fall 2024

Programming Assignment 3: In-order Pipelined RISC-V Simulator with Multi-cycle Execution

Due October 29, 2024 @ 11:59 PM on HuskyCT

Ensure that your git repository is up-to-date by executing `git pull` within the `cse4302` directory. This will create a new `pa3` directory in the repository root that contains the materials for this programming assignment. There are several source code files in the `src` directory, but you will **only** modify `sim_stages.c` for this assignment; **you are not allowed to modify any other files in the `src` and `unittests` directories or the two bash scripts.**

You will modify `sim_stages.c` to implement a fully functional pipelined processor simulator that supports in-order execution with multi-cycle latency for the *riscv-uconn* ISA. Your simulator implementation must be functional and terminate for all unit tests in the `unittests` directory. The `dump_all.sh` script will automatically execute each assembled unit test and gather the required outputs in a single `pa3_out.txt` file. A `pa3_out_gold.txt` is provided to you to test the register and memory state output of your assignment. This file does not provide the performance counters data and thus must be ignored using -I 'TOTAL CYCLES SIMULATED' and -I 'AVERAGE CPI' flags when using the `diff` utility for comparisons.

The simulator now features a 4-stage pipeline in `sim_stages.c`. The **fetch** and **decode** stages must process all RISC-V instructions and capabilities supported in PA2. The execution and memory stages are now two parallel pipelines. The **execution** stage executes all non load/store instructions in a single cycle. However, **execution_ld_st** stage executes the load and store instructions in multiple cycles (6 cycles in this assignment). The **writeback** stage receives an executed instruction from one of the execution stages and performs the register write-back operation based on the instruction type. The pipeline must process all instructions *in-order*. Therefore, the **decode** stage must stall when the **execution_ld_st** stage is busy processing a multi-cycle load or store instruction. To support this additional interlock check, a global variable *dmem_busy* is asserted in the **execution_ld_st** stage to cause instructions in previous stages to stall.

The pipelined parallel execution is supported using the process_instructions() function in `sim_core.c`. The pipeline stages are executed in backward order to capture the current cycle's state in the committed_inst and pc_n variables, and the wb_out_n, ex_out_n, ex_ld_st_out_n, decode_out_n, and fetch_out_n state structs together referred as *_n microarchitecture next state of the pipeline stages. The *_n state structs are globally visible in `sim_stages.c`. At the end of each cycle, all microarchitecture state is updated in the process_instructions() function. This is represented by pc and the wb_out, ex_out, ex_ld_st_out, decode_out, and fetch_out state structs that are also globally visible in `sim_stages.c`.

In the `sim_stages.c`, you are expected to implement the functionality of all RISC-V instructions and capabilities supported in PA2 using the new 4-stage pipeline with multi-cycle execution of load and store instructions. Each stage does not receive explicit arguments, but all the microarchitecture stage is accessible to the pipeline stages for ensuring their intended functionality that is described next.

In each cycle, the pipeline stages execute their functionality to exploit instruction level parallelism. The following global variables are visible to the `sim_stages.c` in addition to the *_out_n and *_out state structs for the fetch, decode, execute, execute_ld_st and writeback stages.

- `forwarding_enabled` enables/disables the forwarding paths.
- `j_taken` tracks an unconditional branch.
- `br_mispredicted` tracks a conditional branch.
- `we_exe`, `we_mem`, and `we_wb` track register file write enables for the execute, execute_ld_st, and writeback pipeline stages respectively.
- `ws_exe`, `ws_mem`, and `ws_wb` track register file index to be written to for the execute, execute_ld_st, and writeback pipeline stages respectively.
- `dout_exe`, `dout_mem`, and `dout_wb` hold the value of forwarded data for the execute, execute_ld_st, and writeback pipeline stages respectively.
- `dmem_busy` tracks if a valid load or store instruction is being executed in the execute_ld_st stage. It must be set in each valid cycle of a load or store instruction and reset during the last cycle of the multi-cycle execution.
- `dmem_access_cycles` is a constant variable that holds the number of cycles a load or store instruction spends in the execute_ld_st stage. This variable is set to 6 for this assignment.
- `dmem_cycles` is a counter to track the number of elapsed cycles for a load or store instruction in the execute_ld_st stage.

The **fetch** stage returns the state struct in one of the following conditions every cycle. A *nop* is returned if a control flow instruction instructs pipeline flush. This is possible for J-Type and B-Type instructions in this cycle. The fetch_out is returned when the current instruction is fetch gated in case the pipeline is stalled. In this pipeline, both pipe_stall and dmem_busy indicate a stall condition. The fetch_out_n is returned when a new instruction is fetched from the instruction memory. This struct must be populated with the necessary fetch stage metadata (the next state) before returning.

The **decode** stage returns the state struct in one of the following conditions every cycle. When no pipeline flush or stall condition is detected, the decode functionality must be performed based on the instruction type being passed on from the fetch stage. The fetch_out struct is first copied into decode_out_n. Then, the decode functionality is performed, and the decode_out_n struct is populated with the necessary decode stage metadata before returning. If a control flow instruction instructs pipeline flush, a *nop* is returned. This is possible for B-Type instruction in this cycle. Furthermore, when pipe_stall or dmem_busy is detected to indicate a stall condition, this pipeline stage also returns a *nop*.

The **execute** stage performs the ALU functionality for all non load and store type instructions in a single cycle. It reads the decode_out struct as an input to compute its next state output, ex_out_n. If the decode_out indicates *nop*, or a load or store instruction type, then it returns a *nop* state struct. Otherwise, a valid instruction is received and processed for execution. The ex_out_n struct must be populated with the necessary execute stage metadata before returning. To support pipeline interlocking and data forwarding, the we_exe, ws_exe, and dout_exe global variables must be updated correctly in this stage.

The **execute_ld_st** stage performs the multi-cycle execution of load or store instruction types. To ensure, in-order execution and writeback of the pipeline, the dmem_busy flag must be asserted during the execution of these long latency instructions to stall the fetch and decode stages. The execute_ld_st stage first reads the decode_out struct as an input to compute its next state output. If the decode_out indicates *nop*, or a non load or store instruction type then it must de-assert dmem_busy to '0', and return a *nop* state struct. If the dmem_busy is '0' and a load or store instruction is received from the decode_out, it indicates the first cycle of the 6-cycle load or store operation. In this first cycle, the memory address metadata is computed for the load or store instruction, and ex_ld_st_out_n state struct is returned. The dmem_busy must be set to '1' in this cycle to indicate the start of a long latency memory operation to the pipeline. The cycles when dmem_busy is detected as '1', the ex_ld_st_out is returned to capture the multi-cycle operation of the load or store instruction by holding on to its state. In these cycles, the dmem_cycles counter must be updated to track the correct latency of the load and store instructions. During the last (6th) cycle of this multi-cycle load or store instruction execution, the actual data memory access is performed and the dmem_busy, dmem_cycles, we_mem, ws_mem, and dout_mem global variables are updated before returning ex_ld_st_out state struct.

The **writeback** stage returns the state struct in one of the following conditions every cycle. A *nop.inst* is returned if both ex_out and ex_ld_st_out are *nop*. In case ex_ld_st_out is a valid load or store instruction type but dmem_busy is '1', the *nop.inst* is returned as there is no writeback since the load or store has not reached its last cycle of execution. In all other scenarios, there is a valid instruction to be committed. Note that ex_out and ex_ld_st_out state structs must never send a valid instruction to the writeback stage in the same cycle as the pipeline ensures in-order execution. Based on the instruction type being processed, the ex_out or ex_ld_st_out state struct is read to receive and process writeback functionality by updating the wb_out_n state struct. The we_wb, ws_wb, and dout_wb global variables are updated before returning wb_out_n.inst to indicate a committed instruction to the process_instructions() function in `sim_core.c`.

When you have completed the programming assignment, submit your `sim_stages.c` and `pa3_out.txt` files via HuskyCT.