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

Programming Assignment 4: Out-of-Order Execution in Pipelined RISC-V Simulator

Due December 2, 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 `pa4` 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. Your simulator needs to work with two configuration flags: `ooo_enabled` and `forwarding_enabled`. This assignment must support all four cases for these flags. The `dump_all.sh` script will automatically execute each assembled unit test and gather the required outputs in a single `pa4_out.txt` file. A `pa4_out_gold_WO_prf_ctr.txt` is provided to you to test the register and memory state outputs. 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.

This programming assignment implements an *out of order execution and commit* of instructions in addition to the functionality supported in PA3. Similar to PA3, the **execution** stage executes all non load/store instructions in a single cycle. However, **execution_ld_st** and **execution_2nd_ld_st** stages execute up to two different load/store instructions concurrently. Each load/store unit takes multiple cycles (6 cycles for each load/store instruction execution) in the execution phase. The **writeback** stage receives an executed instruction from one or more of the three execution stages in a given cycle, and performs the register write-back operation based on the instruction type. The pipeline executes and commits instructions *out-of-order*. Therefore, the Non Load-Store Instructions that are not dependent on the load-store instructions in the **execution_ld_st** or **execution_2nd_ld_st** stages execute in the pipeline and commit out of order. Moreover, load/store instructions that are allowed to enter the two load/store execution stages execute and commit out of order. Therefore, up to three instructions may complete writeback in a given cycle.

The additional load/store execution stage maintains its own structs. Moreover, in addition to the state structs from PA3, the writeback stage tracks the three possible commits using explicit structs for each execution unit. Similarly to PA3, the microarchitecture state is also updated in the process_instructions() function in `sim_core.c`.

In addition to the variables already present in PA3, the following global variables are added to PA4 to support out of order execution:

- `ooo_enabled` enables/disables the Out-of-Order execution of Instructions.
- `we_wb`, `we_ld_st_wb`, and `we_ld_st_2_wb` track register file write enables for instructions in the writeback stage.

- `ws_wb`, `ws_ld_st_wb`, and `ws_ld_st_2_wb` track register file index for instructions in the write-back stage.
- `dout_wb`, `ws_ld_st_wb`, and `dout_ld_st_2_wb` hold the value of forwarded data for instructions in the writeback stage.
- `we_mem2`, `ws_mem2`, and `dout_mem2` are added to track the second load/store execution unit.
- `dmem_busy2` and `dmem_cycles2` are added to support the second load/store unit.

decode	**execute**	**execute_ld_st**	**execute_2nd_ld_st**	**writeback**
`non LD/ST`	RAW_{reg}	$LD : RAW_{reg}$ $LD : WAW_{reg}$	$LD : RAW_{reg}$ $LD : WAW_{reg}$	RAW_{reg}
`LD`	RAW_{reg}	$LD : RAW_{reg}$ $ST : RAW_{mem}$	$LD : RAW_{reg}$ $ST : RAW_{mem}$	RAW_{reg}
`ST`	RAW_{reg}	$LD : RAW_{reg}$ $LD : WAR_{mem}$ $ST : WAW_{mem}$	$LD : RAW_{reg}$ $LD : WAR_{mem}$ $ST : WAW_{mem}$	RAW_{reg}
`terminate`		LD or ST	LD or ST	

The **decode** stage must support the PA4 functionality for out of order execution by tracking conditions when an instruction is allowed to proceed into the execution pipeline. The table shows various scenarios for stall conditions for an instruction being processed in the decode stage. The `ooo_enabled` flag must be used to ensure out of order execution works with and without forwarding. When this flag is zero, your implementation must perform the in-order execution of PA3 and only allocate load and store instructions to the first load/store unit. The instruction State struct is modified to support `ld_st_unit` variable to track the assigned load/store unit during execution. This flag value of 1 implies the instruction is scheduled to execute using execution_ld_st and its value of 2 implies the load/store instruction is allocated to execution_2nd_ld_st. Only during out of order execution, both load/store units may be allowed to execute in parallel.

When any instruction except terminate type instruction is in the decode stage, it may stall if any of its operand(s) detect a RAW hazard with another instruction's destination register in any of the execute stages or the writeback stage. Note that writeback stage during OOO can commit up to three instructions simultaneously.

For terminate instruction (`addi zero,zero,1`) in the decode stage, a valid load or store in the load/store units indicates another older instruction that must proceed to its final execute cycle before the terminate instruction is allowed to enter the execution stage. Otherwise, terminate instruction may commit before an older instruction and terminate the program prematurely.

When a non load/store instruction is in decode stage and there is a register WAW hazard for a valid instruction in the load/store units, the decode must stall since otherwise the non load/store instruction may commit its value ahead of an older register write to the same register.

When a load is in the decode stage and there is a RAW hazard in memory with another store instruction in one or both of the load/store units, the decode must stall. Since decode does not know the address of its load instruction, this stall is enforced whenever a load type instruction is active in the load/store unit(s), i.e., not completed execution.

When a store is in the decode stage and there is a WAR hazard in memory with another load

instruction in one or both of the load/store units, the decode must stall. Since decode does not know the address of its store instruction, this stall is enforced whenever a load type instruction is active in the load/store unit(s), i.e., not completed execution.

When a store is in the decode stage and there is a WAW hazard in memory with another store instruction in one or both of the load/store units, the decode must stall. Since decode does not know the address of its store instruction, this stall is enforced whenever a store type instruction is active in the load/store unit(s), i.e., not completed execution.

When a load is in the decode stage and older load is in the load/store unit(s), the decode does not stall.

Note that for all previous stalls due to the load/store units, the load/store unit busy signal must be asserted when a load or store enters execution. During its last execution cycle, the busy flag is de-asserted and the store to memory completes, or a load from memory completes with data forwarded if forwarding is enabled. The above-mentioned stalls apply when the load/store unit(s) are busy. A load or store in its final execution cycle may not stall decode since it is considered to have completed execution in this cycle. Do not make any assumptions about when in the execution stage a load or store to memory is initiated and memory is being read or written to. The only known fact about the load/store units is that a load or store completes execution in its final cycle of execution.

Once an instruction is allowed to proceed past the decode stage, it is executed in one of the three execution units. This assignment must support the appropriate functionality needed for the **execute()**, **execute_ld_st()**, and **execute_2nd_ld_st()** stages. Note that `ld_st_unit` flag must dictate whether a load or store executes in its allocated execution stage.

The **writeback** stage must be modified to support up to three instruction writebacks (and data forwarding) in a given cycle when `ooo_enabled` flag is set.

When you have completed the programming assignment, submit your `sim_stages.c` and `pa4_out.txt` files via HuskyCT.