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

# Programming Assignment 1: Pipelined RISC-V Simulator with Interlocks and No Control-Flow Instructions

Due September 17, 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 `pa1` directory in the repository root that contains the materials for this programming assignment.

Navigate to the `assembler` and `make` it again. Periodically, the assembler may be updated with new functionality, so it is important to ensure it is current between assignments.

The following is a brief description of the relevant materials in the `pa1` directory:

| | |
|---|---|
| `src/` | Simulator source code |
| `unittests/` | Simulator unit tests (test programs) |
| `README.md` | Simulator and unit test build instructions |
| `assemble_all.sh` | Bash script to automatically assemble the tests in `unittests/` |
| `dump_all.sh` | Bash script to automatically gather the outputs for all unit tests |

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 5-stage pipelined processor simulator for the subset of *riscv-uconn* ISA described in this document. Your simulator implementation must be functional and terminates 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 `pa1_out.txt` file. You are encouraged to write and test your own unit tests, but they will not contribute to your grade for PA1.

A basic implementation of the pipeline stages (`fetch()`, `decode()`, `execute()`, `memory()`, and `writeback()`) that support the ADDI instruction are in `sim_stages.c`. You will modify this file to support the necessary implementations for this assignment.

**You are expected to modify `sim_stages.c` to support the following instructions.**

1. R-Type Instructions: ADD, SUB, AND, OR, XOR, SLT, SLL, SRL
2. I-Type Instructions: LW, ADDI, ANDI, ORI, XORI, SLTI, SLLI, SRLI
3. S-Type Instructions: SW
4. U-Type Instructions: LUI

**Support for Non Control-flow RISC-V instruction types:** Implement the code for all PA1 instruction types in the `sim_stages.c` **`decode()`, `execute()`, `memory()`, and `writeback()`** functions. Each stage must only use the provided `State` structure to populate the necessary dynamic metadata for each instruction.

The pipeline implementation given to you as a starting point does not support interlocking in the presence of hazards. However, individual unit tests, **ITYPE_*.asm**[1], **RTYPE_*.asm**, **STYPE_STORE.asm**,

[1] * denotes the various instruction types in this category. Consult the `unittests` directory for more details.

and **UTYPE_LUI.asm** are sufficient to test the functionality without requiring the support for dependent instructions in the pipeline. You are encouraged to use these tests to complete this part of the assignment.

**Support for Pipeline Interlocks:** This component of the assignment assumes each instruction individually and functionally executes through the pipeline. **The pipeline must handle data and structural hazards** when multiple instructions co-exist with data dependencies. Note that control hazards are not an issue since control-flow instructions are not supported in PA1.

Pipeline-related control variables are provided in `sim_core.h` and listed in Figure 0.1. These global variable are accessible to each pipeline stage and must be updated correctly to implement pipeline interlocks.

| Pipeline Control Variables | Description |
|---|---|
| `pipe_stall` | Set when the pipeline must stall |
| `we_exe, we_mem, we_wb` | Set when an instruction performs a writeback |
| `ws_exe, ws_mem, ws_wb` | Specifies the register index to be written to |

**Figure 0.1: Global variables used to control the pipeline.**

In the start of `fetch stage`, the `fetch_out` struct must be returned if the `pipe_stall` is `1`. This ensures that the fetch stage does not advance `pc` to avoid structural hazard in the pipeline's fetch stage. When the pipeline is not stalled, advance `pc_n` sequentially to the current `pc` + 4.

The `decode stage` must detect all **data hazards**[2] to ensure correct execution of the pipeline. This check uses the global pipeline control variables `we_exe`, `ws_exe`, `we_mem`, `ws_mem`, `we_wb`, and `ws_wb` to determine if the decoded instruction is dependent on a pending register update in a later stage of the pipeline. The write enable signals (`we_*`) are set when the instruction in the corresponding stage writes to a register, and the write select signals (`ws_*`) contain the register *index* of the register to be written to in the corresponding stage. The check also uses local variables in the decode stage that assert when the decoded instruction reads the first and/or the second register operand respectively. Care must be taken when asserting these variables when register `x0` (`zero`) is read by an instruction. Since register `x0` is not allowed to be written to during normal program operation, any data dependencies on this register should be ignored.

The data hazard interlock check must force the pipeline to stall by asserting `pipe_stall` until the necessary data dependency is resolved in the writeback stage. On asserting `pipe_stall`, a `nop`[3] must be returned for the decode stage to ensure that the current dependent instruction does not propagate in the pipeline.

The pipeline stages after decode do not stall due to data or structural hazards. However, they need to support the global pipeline control variables needed to implement the interlock check in the decode stage. In the `execute` stage, `we_exe` must be asserted for instructions that will update the register file. When this is the case, `ws_exe` must be set to the register specifier that will be written to. Similarly, `we_mem` and `ws_mem` must be updated in the `memory` stage, and `we_wb` and `ws_wb` in the `writeback` stage.

**When you have completed the programming assignment, submit your `sim_stages.c` and `pa1_out.txt` files via HuskyCT.**

[2]In the 5-stage in-order pipeline, only read-after-write (RAW) hazards need to be supported.

[3]The `nop` is provided in `sim_core.c` as a `Struct` constant, and you must use it when returning a `nop`. The encoding of a `nop` is `addi x0, x0, 0`.