University of Connecticut
CSE 4302: Computer Architecture
Fall 2022

Programming Assignment 2: 5-Stage Pipelined Simulator

Due November 4, 2022 (Friday) @ 11:59 PM on HuskyCT

Introduction

In this programming assignment, you will build a 5-stage pipelined simulator implementing the MIPS-based *riscy-uconn* ISA.

First, ensure that your git repository is up-to-date by executing `git pull` within the `cse4302` directory. This will create a new `pa2` directory in the repository root that contains the materials for this programming assignment. The directory and file structure of `pa2` is the same as `pa1`. **As before, you may only modify `sim_stages.c`.**

The objective of this programming assignment is to modify `sim_stages.c` to implement a fully functional 5-stage pipelined CPU simulator for the *riscy-uconn* ISA. The necessary modifications to your non-pipelined implementation are described in the subsequent sections of this document.

To receive full credit for this assignment, your simulator implementation must be fully functional and correct **both with and without forwarding** for all 10 unit tests in the `unittests` directory.

Once you have completed this programming assignment, submit your `sim_stages.c` file via HuskyCT by the posted deadline. **To receive credit for this assignment, you must also schedule a 10–15 minute code review meeting with the TA. You have up until 2 weeks after the HuskyCT deadline to complete the code review.**

1 5-Stage Pipelined Simulator

The objective of this programming assignment is to extend the non-pipelined simulator implementation of PA1 into a fully pipelined implementation that supports forwarding.

1.1 Simulator Structure

The simulator structure is mostly the same as the non-pipelined simulator in PA1 with the following differences:

1. Instead of executing each CPU stage in subsequent cycles, the simulator now executes each stage in the same cycle like a typical pipelined machine. The machine state is now updated in two phases.

   In the first phase, the `writeback()`, `memory_stage()`, `execute()`, `decode()`, and `fetch()` functions are invoked in this order to evaluate the corresponding stage for each in-flight instruction. The input of each pipeline stage is the current state, and the output of each pipeline stage is the desired state for the next cycle. The next cycle state for each pipeline stage is stored in the corresponding `*_n` variables found in the `process_instructions()` function.

   The second phase is responsible for actually advancing the pipeline and moving the machine's current state to the next state. To do this, the global cycle counter is incremented and the current pipeline state for each stage is set to the next cycle state evaluated in the first phase.

2. Pipeline-related control variables have been added to `sim_core.h` and listed in Figure 1. The subsequent sections detail when and how to use these new control variables.

Pipeline Control Variables
`forwarding_enabled`
`pipe_stall`
`j_taken`
`br_mispredicted`
`lw_in_exe`
`we_exe, ws_exe, dout_exe`
`we_mem, ws_mem, dout_mem`
`we_wb, ws_wb, dout_wb`

Figure 1: Pipeline-related control variables.

1.2 Forwarding

The simulator executable now accepts a program argument that controls whether forwarding is enabled or not:

```
$ ./simulator OUT_FILE FORWARDING_ENABLED
```

where `OUT_FILE` is an assembled *riscy-uconn* program and `FORWARDING_ENABLED` is either 0 or 1 to indicate that forwarding is disabled or enabled respectively. The forwarding enabled status is visible in the `sim_stages.c` file through the `forwarding_enabled` variable declared in `sim_core.h`.

When forwarding is enabled, the machine will enable all forwarding paths from the execute, memory, and write-back stages to the decode stage. When forwarding is not enabled, the machine will stall when a hazard is detected.

1.3 Modifications

The following sections describe the required modifications for each stage of the simulator. Each stage should use your PA1 implementation as its starting point. It is assumed that your PA1 implementation is correct.

1.3.1 Fetch Stage

Every cycle, the `fetch_in` argument of the fetch stage is set to the fetch output of the previous cycle by the simulator core. Consequently, this variable is used to hold the fetch output when the pipeline is stalled. The global variable `pipe_stall` is used to determine when to hold fetch output. The fetch stage re-executes the current instruction when the decode stage asserts (sets to 1) the pipeline stall condition.

The fetch stage must also inject a `NOP` into the pipeline if the control flow changes. The global variables `j_taken` and `br_mispredicted` are used to determine when to return a `NOP`, or `0x00000000`. The decode stage asserts `j_taken` when an unconditional branch instruction changes the program control flow. Similarly, the execute stage asserts `br_mispredicted` when a conditional branch is taken.

Finally, when the pipeline is not stalled or standard program control flow is not changed, the fetch stage performs the instruction memory lookup using the current `PC` (i.e., PC/4 as address) like in the non-pipelined simulator.

1.3.2 Decode Stage

The decode stage must first check if a `BNE` or `BEQ` instruction in the execute stage resolved to "branch taken." The `br_mispredicted` flag is set whenever this occurs, since the current pipeline always predicts `PC+4` for conditional branches in decode. If a "branch taken" is detected, then the decode stage must inject a `NOP` into the pipeline. If not, decode proceeds.

There are two local variables in the decode function, `rs_enabled` and `rt_enabled`, that are asserted when the decoded instruction reads the first and/or second register operand respectively. Explicitly, `rs_enabled` is asserted when the decoded instruction reads the the first register operand (`RS`) and `rt_enabled` is asserted when the decoded instruction reads the second register operand (`RT`). **Note:** be careful when asserting these variables when register `$0` (`$zero`) is read by an instruction; since register `$0` should never actually be written to during normal program operation, any data dependencies on this register should be ignored.

The decode function must also perform the pipeline interlock checks. These checks must use 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 write-back in a later stage. 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 that stage.

Following the interlock checks, the decode stage must determine whether to (i) forward data from the execute, memory or write-back stage if forwarding is enabled, or (ii) stall the pipeline by asserting the `pipe_stall` signal if forwarding is not enabled. If forwarding is enabled, the decode stage should use the forwarded data from the appropriate stage. `dout_exe`, `dout_mem`, or `dout_wb` contain the forwarded data for the corresponding stage. **Note:** a load (`LW`) instruction in the execute stage does not have dependent data to forward to the decode stage. Thus, the decode must assert `pipe_stall` when `lw_in_exe` is asserted and the corresponding dependency is detected.

When `pipe_stall` is asserted, the decode stage must override the current instruction and inject a `NOP`. Moreover, the decode stage must return the modified structure **without** modifying the program counter so that the fetch stage unit re-fetches the same instruction in the next cycle.

If no data dependencies are detected (`pipe_stall` is not asserted), the decode stage must perform the necessary program counter update logic like in PA1. **Note:** the `pc` variable should no longer be written to directly and the `pc_n` variable should be used instead; this change is already reflected in the `advance_pc()` function. In addition, `j_taken` must be asserted for the JR, J, and JAL instructions.

For BEQ and BNE instructions, PC+4 is predicted for the next instruction address. Similar to PA1, br_addr should be set to PC+(sign-extended IMM) and the PC is advanced by 4 bytes with advance_pc(4) function.

1.3.3 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 the register specifier that will be written to. When appropriate, dout_exe must be set to the ALU output value that may be forwarded to the decode stage.

For BNE and BEQ instructions, if the branch is taken, then pc_n is set to br_addr and the br_mispredicted flag is set to indicate that earlier stages need to be flushed. Otherwise, nothing is done for these instructions.

If a load instruction is being executed, lw_in_exe must be asserted. This ensures that the decode stage can appropriately stall the instruction dependent on this load instruction when forwarding is enabled.

1.3.4 Memory Stage

we_mem must be asserted for instructions that will update the register file. When this is the case, ws_mem must be set to the the register specifier that will be written to. When appropriate, dout_mem must be set to the output that may be forwarded to the decode stage.

1.3.5 Write-back Stage

we_wb must be asserted for instructions that will update the register file. When this is the case, ws_wb must be set to the the register specifier that will be written to. When appropriate, dout_wb must be set to the output that may be forwarded to the decode stage.