UChicago CMSC-22200 Computer Architecture, Autumn 2016

Lab 5: Multicore

Instructor: Prof. Yanjing Li
TAs: Xin-Chuan (Ryan) Wu, Maria Hyun, Mohammad Reza Jokar, Gushu Li

Assigned: Tue., Nov. 15th, 2016
Due: **11:59 pm, Thur., Dec. 1st, 2016**

Objective

There are two main objectives in this lab.

In the first part of this lab, we will test all functionalities of the ARM simulator from previous labs, which include:

1. 40 required ARMv8 instructions (lab1)
2. 5-stage pipeline implementation with support for data dependency handling and control dependency handling (lab2)
3. Branch prediction (lab3)
4. Level-1 (L1) instruction and data caches (lab4)

Please refer to previous lab assignments for detailed descriptions of the requirements. We will provide *all* test cases. Your job is to produce output files (by using the rdump and mdump commands) that *exactly match the provided reference simulator for every cycle.* If you have implemented the required functionalities correctly in all four previous labs, you should be able to finish this part with minimal effort.

In the second part, you will extend your simulator in the first part to model a *multicore* system. In the following sections, we will describe how the functional interface to the simulator should change in order to support multiple independent threads, as well as the microarchitectural specification. Cache coherence is not required for this lab. Therefore, we will not use test cases where two processor share the same memory locations.

We will also offer extra credits in a separate document.

Multicore: Functional Changes

Before we model the microarchitectural details of a multicore system, we need to extend the *functional interface* of the system so that the user's program can make use of multiple threads. In this lab, we make the following changes to the functional (ISA-level) interface:

- The system has four CPUs (labeled CPU 0 through CPU 3). All CPUs have private states (register files and program counters), but share memory.
- All CPUs are independent expect for the shared memory. You should replicate the pipeline, branch predictor, and L1 caches for each CPU.
- Initially, only CPU 0 is running. CPU 0 implements the full functional behavior of the Reference Simulator in part 1 of this lab. Thus, if a single-threaded program is run, it should behave exactly the same as if it were run on the simulator for part 1 of this lab.
- To start a thread on another CPU, we define a new *system call* with the ARMv8 ERET instruction to spawn the thread. The machine code of ERET is 0xd69f03e0. When this instruction is executed with X30 = 1, 2, or 3, then a new thread is spawned on CPU 1, 2, or 3. To spawn the thread, the program counter of the respective CPU is set to the PC of the instruction following the syscall (PC + 4), and the X29 register on the respective CPU is set to 1. The X29 register on the CPU that invoked the syscall is set to 0. Hence, from the program's point of view, execution continues after the system call on both the initiating CPU and the newly-started CPU, with the X29 register distinguishing which

thread is the new thread and which is the original thread. When a new thread is spawned, execution begins in the subsequent cycle. (We will only test cases where the spawn-thread syscall is invoked to spawn a thread onto a CPU that is not currently executing and has no instructions in its pipeline.)

- To help us test programs more easily, we add a new *console output* system call. When ERET is invoked with $X30 = 11, then the simulator takes the value in $X29 as a syscall argument, and prints a line such as "OUT (CPU 2): 1234fffe" on standard output, where 1234ffe is the value in register X29.
- The HLT instruction now halts only the current CPU. Simulation continues until all CPUs are halted.

Simulator Shell Changes

For the second part of the lab, you will need to modify the shell in `shell.c` and `shell.h`. We will specify the exact output that your simulator must produce (**please ensure that your turned-in simulator does not emit any debug output; any additional output will be construed as an incorrect result**).

We only require your lab5 part 2 simulator shell to support the following commands:

- `go`: execute until all CPUs are halted. Output should occur only when console-out system calls are executed.
- `run N`: run for N cycles. As above, only console output should be printed. If execution completes before N cycles, the simulator should halt at that point instead.
- `i X Y`: set register X on CPU 0 to value Y.
- `rdump`: Dump register state for all four CPUs. Your simulator should EXACTLY print a line "CPU n:" followed by lines for "X0: 0x0" through "X30: 0xdeadbeef", "FLAG_N: 1", and "FLAG_Z: 0" (with appropriate values substituted). After all four CPU register dumps, there should be a line "No. of Cycles: n". There should be no blank lines.

Note that commands `go`, `run` and `i` work as before. Only `rdump` must be modified to show register state for all CPUs.

When standard input reaches EOF, the simulator should quit without printing any additional output.

Multicore Timing Specifications

Your job for part 2 of this lab is to implement the following behavior in a cycle-accurate manner, starting with your simulator from part 1 of lab5.

- CPUs are simulated in ascending order (CPU 0, CPU 1, CPU 2, CPU 3), one cycle at a time. Hence, cycle 0 of CPU 0 is simulated, then cycle 0 of CPU 1, etc.
- For this part, branch recoveries can cancel the instruction cache miss stalls (similar to lab4).
- The ERET instruction should *serializes* the pipeline. In other words, when ERET is in the decode stage, it must stall in the decode stage if any instructions are in subsequent stages (execute, memory, writeback). When ERET is in the execute, memory or writeback stages, the two instructions following it must stall in the decode and fetch stages until ERET leaves the pipeline. The ERET's action takes effect when the ERET instruction reaches the writeback stage.

Getting Started

There is no lab handout directory for this lab. Instead, you should begin with your Lab 4 simulator.

Grading

We will provide reference simulators for both parts of the lab. Unlike previous labs, we will also provide all test cases we'll be using for grading. Note that, however, this set of tests is not meant to be exhaustive, and it's always good practice to think about other tests that exercise all aspects of the simulator.

1. 70%: Single-core lab5 simulator. We will provide 6 test cases, and each test case is 11.66%.

2. 30%: Multi-core lab5 simulator. We will provide 3 test cases, and each test case is 10%.

In both parts, we will check your code cycle by cycle for each test. Your output must match ours *EXACTLY* (i.e., using a diff command produces no output). *No* partial credit will be given for any test cases.

Handin

As in previous labs, please hand in your lab electronically in your SVN, and name the hand-in directory lab5-part1 for the first part of the lab, and lab5-part2 for the second part.

You may modify any file in the simulator source (and/or add new files). Since we have specified the external program interface for this lab, there are no restrictions on how you may modify any source file.