Lab 4B: Fine-Grained Multithreading

Assigned: Mon., 2/24; Due: **Fri., 3/21** (Midnight)

Instructor: Onur Mutlu
TAs: Rachata Ausavarungnirun, Varun Kohli, Paraj Tyle, Xiao Bo Zhao

1. Introduction

In this lab, you will extend your pipelined ARM machine to support the control flow instructions that we deferred from the previous lab (conditional and unconditional branches). You will also implement fine-grained multithreading to prevent stalling. Along the way, you will learn about the trade-offs for using fine-grained multithreading, and after you are finished, you will have a complete pipeline that can run four contexts[1] without hardware stalling.

2. Additions to the ARM Machine

2.1. Architecture

- **Instruction Set.** You should use the five-stage pipeline ARM core with **branches** from **Lab 4A** as a starting point. However, you should *remove* any branch prediction mechanism. The machine should still support all ARM instructions specified in Lab 1. The behavior of your pipeline should be match that of your (now correct) implementation of the Lab 1 simulator for the 25 instructions in the following table. The only **exception** is the instruction `SWI`, which we are redefining for this lab only (see section below).

ADC	ADD	AND	B	BIC
BL	CMN	CMP	EOR	LDR
LDRB	MLA	MOV	MUL	MVN
ORR	RSB	RSC	SBC	STR
STRB	SUB	TEQ	TST	SWI

- **System Call Instruction.** If the bottom 24 bits are `0xA`, then it terminates the program (same as Lab 3) running in the current context. Once a context terminates, there is no way to resume it. Whenever this context is executed again, a `NOP` equivalent instruction should be inserted into the pipeline. Once all contexts are terminated, the simulation ends. Finally, for debugging and grading purposes, when simulation ends you must ensure that the contents of the register file for each context are dumped out in the **same format** as Lab 3.
  If the bottom 24 bits are `0x5`, then the thread id of the current thread should be loaded into the register `R0`. This instruction should work similarly to a move, i.e. `PC` is incremented by 4 as usual and data dependencies are handled properly.

- **Exceptions.** No support (same as Lab 3).

2.2. Microarchitecture

Pipeline Modifications.

In this lab you will create **four** different execution contexts and run them in a fine-grained multi-threading fashion. Each context should contain its own set of registers(`R0-15` and `CPSR`). Also, each context's `PC` should be initialized to the same address, 0x400000, and all other registers should be

[1] For this lab, we will use the words context and thread interchangeably.

initialized to 0. The number of pipeline stages and the number of execution units should not change. Data forwarding is still required for some type of instructions. The remaining mechanisms from previous labs (branch predictor and unnecessary data forwarding or stalling) should be removed since they are obsolete.

Memory.

Memory is shared between all **four** contexts. This creates opportunities for writing interesting multithreaded programs. Basic software primitives such as locking and message passing can all be implemented using shared memory. You are encouraged to write some complex test cases to verify the correctness of your multithreaded machine.

Fine-Grained Multithreading.

At each cycle the machine should fetch the instruction from a different context and insert it into the pipeline. As a result, each thread is only executed once every fourth cycle. The pipeline should always remain full, and no hardware stalling should occur.
The four context should have identifiers 0-3. Thread 0 should execute first, followed by 1, 2, and 3. When all contexts are executed once, the machine should fetch the next instruction for context 0. Therefore, the writeback stage and the fetch stage should always be executing from the same context.
In order to facilitate testing and grading, we are providing you with a new register file. You can find this at `/afs/ece/class/ece447/labs/lab4b/447src`. This new register file allows you to specify a thread id through the parameter `id`, which defaults to 0. Once halted, the content of the registers will be dumped to a log identified using `id`. Therefore, **it is important that you instance each of the register files with the correct** `id`.
Once a thread terminates by executing `SWI #0xA`, all write signals to its register file should be permanently asserted low. This means that its `PC` should increment one last time and point to the next instruction (`PC + 4`). This behavior should match the one in Lab 1 and Lab 2. Finally, you should ensure that all register files are halted together. The `halt` signal for the register files should be asserted only when all threads terminate.

3. Submission

3.1. Lab Section Checkoff

So that the TAs can check you off, please come to any of the lab sections *before* Sat., 4/1. **Note that you must be checked off for both Lab 4a and Lab 4b by then.** You can get them checked off separately in different lab sections or all in one sitting. Please come *early* during the lab section. During the Lab Section, the TAs may ask you:

- to answer questions about your implementations,
- to simulate your implementations using various test inputs (some of which you may have not seen before),
- to show that your implementation can execute all four contexts fine-grainedly, each terminating at different times,
- to explain the trade-offs between using fine-grained multithreading and fancy branch predictors.

3.2. Source Code

Make sure that your source code is readable and documented. Please submit the lab by executing the following commands. **Make sure you format your folders as specified. We will penalize wrong submissions.**

Fine-Grained Multithreading.

```
$ cp -r src /afs/ece/class/ece447/handin/lab4/andrewID/threaded
$ cp -r inputs /afs/ece/class/ece447/handin/lab4/andrewID/threaded
```

3.3. README

In addition, please submit a `README.txt` file. To submit these files, execute the following command.

```
$ cp README.txt /afs/ece/class/ece447/handin/lab4/andrewID/threaded/README.txt
```

The `README.txt` file must contain the following three pieces of information.

1. A high-level description of your design (including what are the initial values of your components).
2. How does fine-grained multithreading stack up agaisnt branch predictors? How does each one perform? What are some limitations for using each approach? How are areas and critical paths affected?
3. Did you try writing any interesting multithreaded program? Could you write the same program using one thread? If so, does it run faster using fine-grained multithreading or using branch predictors?

It may also contain information about any additional aspect of your lab.

3.4. Late Days

We will write-lock the handin directories at midnight on the due date. For late submissions, please send an email to `447-instructors@ece.cmu.edu` with tarballs of what you would have submitted to the handin directory.

```
$ tar cvzf lab4_threaded_andrewID.tar.gz src inputs README.txt
```

Remember, you have only 7 late lab days for the entire semester (applies only to lab submissions, not to homeworks, not to anything else). If we receive the tarball within 24 hours, we will deduct 1 late lab day. If we receive the tarball within 24 to 48 hours, we will deduct 2 late lab days... During this time, you may send updated versions of the tarballs (but try not to send too many). However, once a tarball is received, it will immediately invalidate a previous tarball you may have sent. You may not take it back. We will take your very last tarball to calculate how many late lab days to deduct. If we don't hear from you at all, we won't deduct any late lab days, but you will receive a 0 score for the lab.