Computer Architecture - Spring 2018

ARM Thumb Simulator

All submissions must be received by June 8, 2018, 11:59pm

Access the repo at:
https://classroom.github.com/g/EP6PBNiG
In this project you will build an ARM Thumb simulator to examine performance metrics of some given programs. You are provided the support code for the simulator; and you will complete the simulator and collect statistics with it.
You may **work in pairs** on this project.
It would be best to **overestimate the amount of time this project will take**. The learning curve is higher than the lab assignments have been. You will spend a lot of time reading the ARMv7 Architecture Reference Manual, particularly Chapters A5 and A7.

Tasks

The simulator should handle a subset of the ARM Thumb instruction set. The "Shang" benchmark on which you will do measurements has **the following instructions that you must support:** `push`, `pop`, `sub` (sp minus immediate), `sub` (immediate), `sub` (register), `add` (sp plus register), `add` (sp plus immediate), `add` (register), `add` (immediate), `cmp` (register), `cmp` (immediate), `mov` (immediate), `mov` (register), `str`, `strb`, `stmia`, `ldr`, `ldrb`, `ldr` (literal), `ldmia`, `b` (unconditional), `bcc`, `bcs`, `beq`, `bge`, `bhi`, `ble`, `bls`, `blt`, `bne`, `bl`, `lsl` (immediate), `neg` (also called `rsb`).
Pay particular attention to the byte memory operations. The C++ implementation of the simulator only reads and writes memory on a word granularity (4 bytes at a time), so make sure you work within that constraint.
The most challenging instructions tend to be `push`, `pop`, `ldm`, `stm`, and the unconditional branch `b`. Follow exactly the specification in the ARM v7 manual for these.
Your tasks are as follows:

- Complete the simulator so it successfully runs the Shang benchmark (details below). The simulator, as provided, implements only a handful of instructions. You need to add support for other instructions.

Here is the RIGHT way to do this project:

- **Make test cases other than Shang to check specific instruction implementations.**
- **Learn how to utilize unit tests for C++ (check out cppunit and gtest).**
- **Make test cases other than Shang to check specific instruction implementations.**
- **Test your statistics on programs smaller than Shang.**
- **Start Early!**
- **Learn how to use GDB!**

- Shang takes a little while to run in simulation. It's difficult to debug on a large program like Shang, and if you don't write any smaller tests to make sure your statistics and individual instructions work, you will be very frustrated the night before the assignment is due when Shang does not run correctly and you don't know why.
- You also need to collect statistics on the benchmark that you run to answer the questions in the writeup. Please measure dynamic (runtime) statistics for:

    - Number of instructions
    - Number of Memory Reads and Memory Writes (push, pop, ldm and stm count 1 for each register handled by the instruction).
    - Number of Conditional Branches, both forward and backward, taken and not taken in each direction. Do not include unconditional branches (including procedure calls or returns) as branches.
    - Cache performance. For a 256-byte direct-mapped cache, what is the best block (line) size in bytes? You could choose to have 64 entries in the cache of 4 bytes each, or instead 1 entry of 256 bytes, or anything in between. What has the best hit rate?

You will measure statistics for different optimization levels of the Shang benchmark. Example outputs from the instructor's simulator for the fib and shang benchmarks are included in the sample outputs directory.

Getting Started

You'll want to start with the "fib" test program that is supplied. It should return the (hex) value 59 in `r0`. However, **when you first run it, it will fail because not all the instructions are implemented, and you should add those first until the program runs correctly**. A complete output of thumbsim with the -i and -d flags is included, along with the assembly file for fib.
Next, you will want to collect statistics and fix the Cache::access routine.
You should make sure your tool-flow works properly and have a working fib, without statistics, by the end of Week 1 of the project. You should have correct statistics including cache for fib and some of the additional instructions for shang by the end of Week 2 at the latest. **If you don't meet these milestones, you're in danger of not finishing the assignment.**

Deliverables

You have two deliverables. First, you will submit your source code through GitHub. You will also submit a one-page PDF writeup (no longer than one page!) also through GitHub, that presents your conclusions to the following questions: **Base your answers on the statistics you gather from Shang.**

1. If you are building a processor and have to do static branch prediction (meaning you have to assume at compile time whether a branch is taken or not), how should you do it? You can make a different decision for branches that go forward or backward.
2. If you are building a 256-byte direct-mapped cache, what should you choose as your block (line) size?
3. What conclusions can you draw about the differences between compiling with no optimization and -O2 optimization?

Compiling ARM Thumb Programs

This process is rather involved, and involves the use of a Raspberry Pi. For now, use the provided files. When you need to generate your own test cases, I will post detailed instructions on our Piazza page on how to create the simulator input files.

The Simulator

You get five source files (plus a header file). You should have to only modify three of them. Here is what they are and what you should do:

- **decode.cpp** associates opcodes with their string equivalents, and prints them if the flags are set. You will need to modify this file to decode the new instructions you encounter. The purpose of decode is to print instructions and to return the correct instruction types to the execute function.
- **execute.cpp** is the major file you should change. The simulator calls execute() for each iteration representing an instruction, fetches the current instruction, decodes and evaluates that instruction, changing the machine state (the data memory `dmem`, the register file `rf`, and the program counter `pc`). I have left a few instructions as examples in this file, including some of the trickiest ones to implement. You need to fill in the rest of the instructions and also capture all necessary statistics.
- **main.cpp** contains the main routine and parses command-line arguments which may be helpful in debugging:

    - -p dumps the parsed program at the start of the simulation.
    - -d dumps the contents of data memory (all non-zero data memory entries) and the register file after the end of the simulation.
    - -i prints every instruction as it executes.
    - -w prints every write to data memory.
    - -s prints statistics at the end of the program.
    - -c # (fill in # with a size in bytes) enables caches of size #. For this assignment, -c 256 is probably most appropriate.
    - -f *simfilename* runs the sim file specified. This option must be specified.

    You should not have to change this file.

- **parse.cpp** parses the sim file. You shouldn't need to change it.
- **thumbsim_driver.cpp** (and **thumbsim.hpp**) contain the core data structures. You should only need to change one routine in this file, Cache::access, which currently returns a cache miss (false) for every cache access. You will need to enter tags into the cache ("entries") on a miss and check tags on every access.

    If you call "thumbsim -c 256" the system automatically instantiates several caches: 256B, 4-byte cache lines; 256B, 8-byte cache lines, etc. (up to 256B, 256-byte cache lines). Every time you access memory (ldr, str, push, pop, etc.), *YOU* call caches.access(addr). The system automatically calls Cache::access on each cache. *YOU* also need to write Caches::access(), which

    - Keeps track of cache tags (not data, not valid, ...)

- Determines hit or miss
- Keeps stats: hits++, misses++

Start by typing 'make' to create the `thumbsim` executable.

Technical Notes About the Simulator

The simulator is written in C++ and is written for the g++ compiler. You are welcome to run the supplied code on any machine with any compiler, but I will only support g++, and have only tried the code on x86 and x86_64 Linux.
For reasons your instructor has not yet learned, the simulator does not work correctly when compiled and run on the Raspberry Pis. For this reason, you should not use them for this assignment!
In writing the simulator, special attention was given to the behavior of bit-fields and how they are laid out by the compiler.

The Shang Benchmark

Shang is code that solves a puzzle (solution) from PuzzleHunt II (now apparently offline). ARM Thumb programs place their return code in `r0` and if your code successfully runs the Shang program, it will return a 1 in `r0`; if you don't have a 1 in `r0` then you have implemented your simulator incorrectly. You should be able to locate the strip numbers that are the solution on the stack if your program successfully completes. Note that the presence of a 1 in `r0` by itself does not mean your simulator is working. **The only way to verify correctness is to match statistics and memory contents of your instructor's output.**

Submission

Submit the following files via GitHub:
Report in PDF format (no Docs!):
All source code
Any test files you wrote/used.

Extra Credit Opportunity

For an extra 15%, Compute the accurate cycle count according to the cycle-timings in the Raspberry Pi Manual, and include it in the stats output. This will involve identifying data and control hazards in the pipeline.