COS/ECE 375
Fall 2025

Project 2/Final Project: RISC-V ISA Cycle-Accurate Simulation.
You must work in groups of two, three, or four people. All members must contribute. All members are responsible for the result. Submit via Canvas.
We may choose to randomly sample a set of teams for interviews about their code.

Introduction

In Project 1, you created a functional simulator for the RISC-V ISA. While your functional simulator can execute RISC-V programs and give correct results, it does not simulate the *performance* of an ISA implementation. As you know, implementations of a given ISA may differ widely in performance based on their use of pipelining, their cache size and organization etc. Please read this document in its entirety before attempting the assignment.

In this project, you will create a cycle-accurate simulator for a five-stage pipelined implementation of the RISC-V ISA. Your simulator will capture how instructions flow through the pipeline and will also generate performance statistics for executed programs e.g. number of cache misses. As in Project 1, the simulator is to be written in C++, though you may write a C-style implementation in C++ if you wish. The simulator must be capable of handling all the instructions required for the functional simulator from Project 1.

Note: Some sections from the Project 1 specification may be useful as a reference as you work on this project. In particular, the sections on test case structure, test case compilation, the memory abstraction, and how the simulation was broken down into functional steps.

Overview

In contrast to Project 1, where you executed an instruction without any notion of timing, in this project you will model how the instructions would actually flow through a five-stage pipelined implementation. As seen in class, the five stages of this pipeline are Fetch (IF), Decode (ID), Execute (EX), Memory (MEM), and Writeback (WB). Your simulator must model cycle-by-cycle timing - the fetching, decoding, executing, memory accessing, and writing back. In order to properly model the execution of instructions in the pipeline, you will need to keep track of hazards and cache misses.

We provide you with an initial test case, and the corresponding output from the reference solution. You should study the reference output, especially the pipeline state, to understand exactly how to model the timing of different situations. You should certainly create your own test cases too. You are encouraged to write and share test cases. When using other people's test cases, acknowledge the authors. You will need to submit all test cases you used to test your simulator. The test cases will not be graded. But at least some test cases should be ones you created yourself, and they should have unique coverage that go beyond those you obtained from others. You may find it helpful to write test cases and compare their output from the functional simulator to that from your cycle-accurate simulator. Remember – your cycle-accurate simulator should generate results identical to a correct functional simulator. However, this is the starting point. The timing must also be calculated correctly.

Simulation Requirements

Your simulator must accurately model the *timing* of the following:

- Full forwarding and stalls
    - This includes forwarding from a load to the data input of a store (i.e., register rt) without a stall cycle in between (in other words, a WB->MEM forward from the load to the store). See lecture slides or page 313 in the textbook for an overview of such forwarding. All other cases of forwarding to a store should forward when the store is in the EX stage.
    - Forwarding to ID for the execution of branches. You may need to add stall cycles between the forwarding instruction and the branch in ID for the forwarding to work; see page 326 in the text for further details.
    - Your simulator should implement the following stalls: one-cycle load-use stall, one-cycle arithmetic-branch stall, two-cycle load-branch stall. A stall injects a bubble (nop) to hold the blocked instruction in place while allowing all previous instructions to flow down the pipeline.
    - Note that dependencies on the zero register shouldn't cause a stall
- An always-not-taken branch prediction policy. Recall that control transfer instructions are resolved in the ID stage. Before the target address is calculated and/or resolved, in the IF stage, fetch the next instruction at PC + 4. You must properly squash this speculative instruction once the target address is resolved to be different from PC + 4 (it should never enter ID stage).
    - If you are bored after completing all the requirements, you can implement some more advanced branch prediction policies and write about your findings in Partners.md. But your submission by default should still conform to always-not-taken.
- Cache hits and misses (details provided in the "Caches and Cache Configuration Section)
- Exceptions (details provided in the "Exceptions" section)
- 0xfeedfeed
    - The RISC-V ISA does not contain a HALT instruction. So, as in Project 1, we use the code 0xfeedfeed to signify the end of the code section of a program. You should halt execution when the 0xfeedfeed has completed its WB stage.

Other requirements:

- Must track the correct status of a pipeline stage (bubble, speculative, squashed, etc.), refer to the provided pipeline state format.
- Your simulator must be capable of running for a specified number of cycles and providing statistics about the execution to that point (details provided in the "Simulation Statistics" section)
- The final architectural state (registers and memory) of your cycle-accurate simulator for a given program must match those of a correct functional simulator from Project 1.

TODO

To provide you with a starting point for your cycle-accurate simulator, use the template provided. You should explore the code provided and understand how it all ties together. Refer to the "Provided Files" section for more information on the files.

A suggested roadmap of how you can tackle this project is:

- Pipeline stage simulation (simulator.cpp, simulator.h): We provide functional simulation functions (e.g. simFetch, simAddrGen, etc.) in simulator.cpp. They are solutions to project 1's functional simulator. To simulate pipeline stages, complete the pipeline stage simulation functions (e.g. simIF, simMEM, etc.) by calling the functional simulation functions.
- Stalls and forwarding in the pipeline (cycle.cpp): Once you have all simulation functions ready from simulator.cpp, you can modify runCycles in cycle.cpp to model pipeline behavior like stall and forwarding.
- Cache behavior (cache.cpp, cache.h): This is a rather standalone portion of the assignment. Refer to the "Caches and Cache Configuration" Section for more information.
- Timing of cache misses (cycle.cpp): Put together cache and pipeline simulations.
- Handling of exceptions (simulator.cpp, simulator.h, cycle.cpp): Add detection of exception to simulator.cpp and correctly model timing in cycle.cpp. Refer to the "Exceptions" section.
- Simulation statistics (cycle.cpp): Modify finalizeSimulator in cycle.cpp to report correct statistics. Refer to the "Simulation Statistics" section.

This roadmap is only a suggestion. You can split up various sections of the project amongst group members, but each member should contribute equally. Each group member should also understand all parts of the project. Regardless of the order you do the project in, you must ensure that all the above requirements are met. We may choose to randomly sample a set of teams for interviews about their code.

Caches and Cache Configuration

You must implement one level of I and D caches in your simulator. These caches must be capable of simulating the timing of a variety of cache configurations. They **do not** need to keep track of data in each cache line. The cache size, block size, ways (direct-mapped is a 1-way cache), and latency of each cache is specified in a text configuration file. The caches must implement a **true** LRU replacement policy. Assume cache is write-allocate and stalls on misses (both read and write misses). No dirty bit is needed since the cache is write-through and writes to memory are assumed to not impact timing. You may assume that the block size will always be a power of 2 (in bytes), and that the cache size will always be a multiple of (block size * number of ways). You may assume that a single load/store instruction accesses only a single block. You are required to keep track of the hits and misses in each cache during execution. You may do this in any way you choose, but you must provide the final tally of hits and misses to the dumpSimStats(..) function through a SimulationStats struct.

The fetch stage accesses the I cache, and the memory stage accesses the D cache. A hit completes within the fetch or memory stage. The **additional** latency of a cache miss (i.e., the

miss penalty) is given in the cache configuration file. A reasonable starting point for debugging would be to set the miss penalty to 5 cycles. As an example, if an instruction misses in the I cache, and the I cache miss penalty is 5 cycles, then that instruction should appear in the IF stage 6 times before proceeding to the ID stage.

During a miss, instructions before the missing instruction should proceed down the pipe with NOPs inserted as necessary. Instructions after the missing instruction, if any, should stall in their respective stages. In a typical processor today, the L1 I and D caches would merge their miss streams into a single stream of requests for a unified on-chip L2. Because we are not asking you to simulate multiple levels of caches, you may assume that main memory can accept up to two requests in any single cycle (one from the I cache and one from the D cache) and service them simultaneously.

Note that inserted NOPs, e.g. while modeling exception timing, shouldn't cause a cache miss. You may assume that your simulator will never be tested with self-modifying code. In other words, instructions will never be modified by store instructions.

The cache configuration text file contains eight lines each starting with one number. Text after the number is ignored. The first four lines specify the I cache parameters, and the second four lines specify the D cache parameters. These parameters are cache size in bytes, block size in bytes, number of ways, and miss latency. For example:

```
2048        #  2K Instruction Cache
16          #  16 byte block size (I-Cache)
2           #  2-way set associative (I-Cache)
5           #  5 cycle miss penalty (I-Cache)
4096        #  4K Data Cache
16          #  16 byte block size (D-Cache)
4           #  4-way set associative (D-Cache)
8           #  8 cycle miss penalty (D-Cache)
```

Exceptions

Your simulator must handle two types of exceptions: illegal instructions and memory exception. Exception raises a trap to transfer control to a handler in the operating system. To give a simple exercise of exception handling, we ask that your simulated core should jump to address 0x8000 when any exception is detected. For illegal instructions, you do not need to go beyond what is already detected in the provided simulator: we only look at illegal opcode and illegal funct3 here.

Note that, initially, the addresses 0x8000 and beyond only contain zeros. Jumping to that address and executing instructions from that point will yield a succession of illegal instruction exceptions, causing an infinite loop. To place meaningful instructions at 0x8000, you can use the .org directive to place code at a specified address. To assemble an illegal instruction, you can pick one outside of our supported rv64i subset. To trigger a memory exception, you can access an address outside of the range of the memory.

When an exception occurs, you should NOT update state for the instruction triggering the exception. For instance, if an ADD triggers arithmetic overflow, you should not update the destination register. An illegal instruction should be detected in ID and squashed before it reaches EX. An instruction causing memory exception should be detected in MEM and squashed before it reaches WB. All instructions that were in the pipeline before the excepting instruction must complete execution successfully. All instructions after the excepting instruction must be squashed. IF should be fetching from the exception handler address in the next cycle.

Simulation Statistics

Your simulator must report statistics once the simulation is complete. These statistics include: the number of dynamic instructions executed, the number of cycles executed, I cache and D cache hits/misses, and load stalls. The first two statistics are provided in the starter code. Note that load stalls includes both load-use stalls and load-branch stalls.

Even though a load-branch dependency might require two stall cycles, this would still count as one load stall for our statistics. Additionally, if a single instruction depends on two loads both of which cause stalls, then we would need to increment load stalls twice.

The statistics are output by the finalizeSimulator function in cycle.cpp. You will need to complete this function.

Provided Files

To allow you to concentrate on key aspects of the cycle-accurate simulator, we provide you with these files. >_< indicates the files you will need to modify and submit.

>Partners.md<
Makefile

bin/
- riscv64-elf-as
- riscv64-elf-objcopy
- riscv64-elf-objdump

src/
- MemoryStore.h
- MemoryStore.cpp
- RegisterInfo.h
- Utilities.h
- Utilities.cpp
- >simulator.cpp<
- >simulator.h<
- funct.h
- funct.cpp
- sim_funct.cpp
- >cache.h<
- >cache.cpp<
- cycle.h
- >cycle.cpp<

sim_cycle.cpp
test/
fib.s
fib_cycle_*.ref
illegal.s
illegal_cycle_*.ref
cache_config.txt

riscv64-elf-as, riscv64-elf-objcopy, riscv64-elf-objdump, MemoryStore.{h,cpp}, RegisterInfo.h – What you've already seen in project 1.

Utilities.{h,cpp} – More utilities dealing with dumping state.

>simulator.{h,cpp}< – [Modify these files] Most of Project 1. This provides functions to functionally simulate instructions step by step, and templates you'll need to implement for pipeline stage simulation. **You will need to modify simulator.cpp and simulator.h to implement pipeline stage simulation.**

funct.{h,cpp} – The rest of Project 1. It calls functional simulation functions in **simulator.cpp** to load programs and execute them to completion.

sim_funct.cpp – A main used for building a functional simulator. It calls functions in **funct.cpp**.

>cache.{h,cpp}< - [Modify these files] As given to you, models a cache with random hits and misses. **You will need to modify cache.{h,cpp} to implement real cache hit/miss behavior.**

>cycle.{h,cpp}< – [Modify only cycle.cpp] The part of the simulator that models the pipeline timing and behavior. It does this with calls to the pipeline stage simulation functions and the cache simulator. As given to you, it is a 5-stage pipeline with instructions freely flow through with no regards of hazards. **You will need to modify cycle.cpp to implement real pipeline behavior with hazards, exceptions, and cache misses.**

sim_cycle.cpp – A main used for building a cycle accurate simulator. It calls functions in **cycle.cpp**.

fib.s – A sample RISC-V assembly test program.

fib_cycle_*.ref – The reference state dumps of running fib.s to completion using sim_cycle with cache_config.txt. **fib_cycle_*_ideal_cache.ref** provides reference state dumps of running fib.s to completion using sim_cycle but with ideal cache. This is useful when you have not yet implemented a cache model.

illegal.s – A sample RISC-V test program with an illegal instruction.

illegal_cycle_*.ref – The reference state dumps of running illegal.s to completion using sim_cycle with cache_config.txt. **illegal_cycle_*_ideal_cache.ref** provides those for an ideal cache.

Makefile – Build rules for the executables and assembly test cases. For more details on how to use make, refer to the project 1 description.

>Partners.md< - [Modify this file] A markdown file that needs to be filled out and turned in along with the necessary source files.

Functional Simulator Compilation and Execution

The functional simulator, which does the same thing as Project 1, has its main in sim_funct.cpp. Refer to the provided Makefile for instructions on how to build it. **This functional simulator is fully complete.**

The sim_funct binary takes one command-line argument indicating the name of the binary file to be executed.

```
sim_funct <file.bin>
```

Running sim_funct will create output files with names containing the word "funct". An output file is created for the final reg state, mem state, and simulation statistics. These output files will not be used for grading. The mem and reg state produced by sim_funct may be a useful sanity check to ensure that your cycle-accurate simulator, sim_cycle, reaches the correct, final reg and mem state.

Cycle-Accurate Simulator Compilation and Execution

Your cycle-accurate simulator has its main in sim_cycle.cpp. Refer to the provided Makefile for instructions on how to build it. **This is the simulator you are building in this assignment.**

The sim_cycle binary takes two command-line arguments. The first is the filename of the binary to be executed, and the second is the filename of the cache configuration to be used.

```
sim_cycle <file.bin> <cache_config.txt>
```

Running sim_cycle will create output files with names containing the word "cycle". In addition to the reg state, mem state, and simulation statistics, there is also an output file for pipeline state. These output files will be used for grading and you should ensure that your cycle-accurate simulator produces the correct output for each of these files.

Noteworthy Functions

Status dumpPipeState(PipeState& state, const std::string &base_output_name)
Located in Utilities.{h,cpp}. You do not need to modify this function. The parameters to it provide the current pipeline information in the PipeState struct and start of the filename to use (i.e., the
<basename>). The format of the output is similar to the example output below:

```
Cycle:       51 || Inst at 0x38 (spcu)     | blt zero, t1, -24
| NOP (bubble)            | addi t1, t1, -1         | NOP
(bubble)            |
```

The dumpPipeState function appends this output to the <basename>_cycle_ pipe_state.out file in the current working directory.

All stages of the pipeline have instructions, but sometimes those instructions are microarchitecturally inserted NOPs. NOPs are accompanied by a reason, whether it is an inserted bubble, idling at the start of the execution, or squashed due to misprediction. Speculative instructions in the IF stage should be labeled as so. Please refer to StageStatus in Utilities.h.

Status initSimulator(CacheConfig& icConfig, CacheConfig& dcConfig, MemoryStore *mainMem, const std::string &base_output_name)
Located in cycle.{h,cpp}. When called, this function sets up and initializes all state required by your simulator, including an I-cache and a D-cache configured as specified in icConfig and dcConfig respectively (see the "Caches" section), but does NOT begin execution. The main memory that you should use is provided as a pointer to a MemoryStore. The return value indicates success or failure to aid in debugging.

Status runCycles(uint64_t cycles)
Located in cycle.{h,cpp}. When called, this function runs your simulation for the number of cycles specified (including stalls). It also calls the dumpPipeState function to output the instructions present in the pipeline at the **last** cycle run. For example, when runCycles(40) is called at the start of simulation, the simulator should run for at least 40 cycles (i.e., cycles 0 through 39), and then call dumpPipeState which will emit the instructions present in the pipeline at cycle number 39.

If the halt instruction 0xfeedfeed completes its writeback stage before the simulator has run for the specified number of cycles, your simulator should call dumpPipeState after the cycle in which the halt instruction finishes its writeback stage, and then return from runCycles(..).

When the cycles parameter is 0, the simulator runs until it halts.

If the specified number of cycles were simulated, the function should return SUCCESS. If the end of the program was reached prior to the total number of cycles being simulated, the function should return HALT. You may use other return values for debugging purposes.

Status runTillHalt()
Located in cycle.{h,cpp}. This function simply calls runCycles(1) repeatedly to simulate execution until the halt instruction reaches its writeback stage. Calling runCycles(1) means pipe state is output each cycle. It does not call runCycles(0), which would cause pipe state to only be output at the end.

As in initSimulator, the return value may be used for debugging your simulator.

Status finalizeSimulator()
Located in cycle.{h,cpp}. When called, finalizeSimluator does the following:

1. Calls dumpSimStats(..) with number of cycles simulated and other statistics in the SimulationStats struct. The printSimStats function is provided.
2. Calls dumpRegisterState with an initialized RegisterInfo struct (as you did at the end of simulation in Project 1).
3. Calls dumpMemoryState with MemoryStore representing main memory. As in Project 1, the dumpMemoryState function will dump the contents of a region of memory that will **vary** on a **test-by-test basis**.

What to Submit

Use Canvas to submit your modified cycle.cpp, simulator.h, simulator.cpp, cache.h, cache.cpp, Partners.md, and any test cases you used to test your simulator. You should place all these files in a folder named netid1_netid2_netid3_netid4. Place test cases in netid1_netid2_netid3_netid4/test. If your group is smaller than 4, adjust the file name as necessary. Zip the folder and submit the .zip file. One submission per group is sufficient. You can zip the folder using:

```
zip -r netid1_netid2_netid3_netid4.zip
netid1_netid2_netid3_netid4
```

Grading will be done on Nobel. If we cannot compile your source files with the provided Makefile, we cannot grade your assignment, and you will have to resubmit; the delay will be taken as late days. TAs in office hours cannot devote time for debug code that is not on Nobel.

Grading

Grading will consist mainly of running test RISC-V binaries (including but not limited to any sample binaries we give you) through your simulator. These test binaries will thoroughly test the operation of instructions. Particular attention will be paid to "corner cases". Correctness will be determined by the final register values, memory results, and statistics of the simulator's program. Grading will also consider snapshots of pipeline state at various cycles during the execution. This allows you to earn partial credit even if your results are not completely correct.

We may look at your source code to determine whether instructions are implemented correctly. Code which is difficult for us to understand may lose some points.

Finally, efficiency matters. Excessively slow or pointlessly inefficient simulators will be penalized.

Note:
Please read the specifications more than once. It will save you a considerable amount of time.