CS 152 Laboratory Exercise 2

Professor: John Wawrzynek
TAs: Josh Kang and Hasan Genc
Department of Electrical Engineering & Computer Sciences
University of California, Berkeley

February 3, 2022

Revision History

Revision	Date	Author(s)	Description
1.0	2022-02-03	hngenc	Initial release

1 Introduction and Goals

The goal of this laboratory assignment is to study processor memory hierarchy design by conducting experiments on realistic RISC-V implementations. You will be running simulations of silicon-proven RTL along with a DRAM model.

1.1 Graded Items

All reports are to be submitted through **Gradescope**. Please label each section of the results clearly. All directed items need to be turned in for evaluation. Your group only needs to submit *one* of the problems in the open-ended portion.

- (Directed) Problem 3: Matrix Transposition Case Study
- (Open-ended) Problem 4.1: Validation and Reverse Engineering of Memory Hierarchies
- (Open-ended) Problem 4.2: Design Your Own Hardware Prefetcher
- (Open-ended) Problem 4.3: Design Your Own Replacement Policy and Victim Cache
- (Directed) Problem 5: Feedback

! → Lab reports must be written in *readable* English; avoid raw dumps of logfiles. **Your lab report must be typed, and the open-ended portion must not exceed six (6) pages.** Charts, tables, and figures – where appropriate – are excellent ways to succinctly summarize your data.

2 Background

As with Lab 1, this lab is based on the **Chipyard** framework being actively developed UC Berkeley. However, we will be exploring more sophisticated hardware designs than the rudimentary Sodor processors from the previous lab.

2.1 Chipyard

Chipyard is an integrated design, simulation, and implementation framework for agile development of systems-on-chip (SoCs). It combines Chisel, the Rocket Chip generator, and other Berkeley projects to produce a full-featured RISC-V SoC from a rich library of processor cores, accelerators, memory system components, and I/O peripherals. Chipyard supports several hardware development flows, including software RTL simulation, FPGA-accelerated simulation (FireSim), and automated VLSI methodologies (Hammer).

! → Chipyard documentation: `https://chipyard.readthedocs.io/en/latest/`

2.2 Rocket Chip

Rocket Chip [1] is an open-source SoC generator originally developed at UC Berkeley. It leverages Chisel to compose a library of highly parameterized generators for cores, caches, and interconnects into an integrated SoC. It has been the basis of numerous silicon-proven designs in both research and industry.

Rocket Chip can generate a practically unbounded space of *instances*, including many parameter sets that are impractical or suboptimal. In this lab, we will examine a variety of design points, each with a different memory hierarchy, to explore the concepts described in class. All Rocket Chip instances used in this lab have three major components: processor cores, a cache hierarchy, and an outer memory system.

2.2.1 Rocket Microarchitecture

Rocket Chip derives its name from the Rocket core that it instantiates by default: a 5-stage, single-issue, in-order RISC-V processor. The instances of Rocket used in this lab implement the RV64IMAFDC instruction set variant[1], which refers to the 64-bit RISC-V base ISA (RV64I) along with a set of useful extensions [2]: M for integer multiply/divide, A for atomic memory operations, F and D for single- and double-precision floating-point, and C for 16-bit compressed representations of common instructions

Rocket also supports the RISC-V privileged architecture [3] with machine, supervisor, and user modes. It has an MMU that implements the Sv39 virtual memory scheme, which provides 39-bit virtual address spaces with 4 KiB pages. As such, these designs are capable of booting mainstream operating systems such as Linux.

Rocket has been extensively optimized for efficient ASIC implementation, resulting in specific microarchitectural adaptations that differentiate it from the classic 5-stage RISC pipeline normally seen in educational settings. In particular, the overall design is mainly

[1] Also known as RV64GC, with G (“general-purpose”) being the canonical shorthand for “IMAFD”

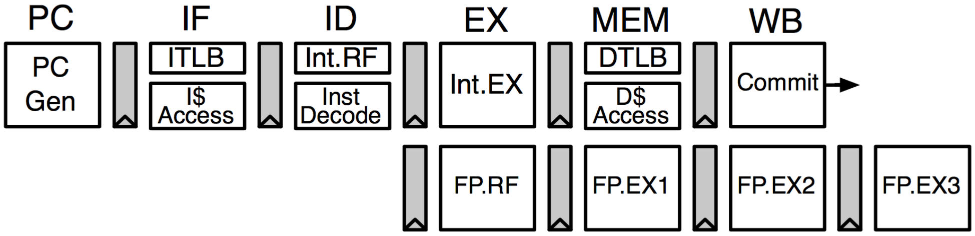

Figure 1: Rocket pipeline

concerned with (1) minimizing high-fanout stall signals and (2) restructuring pipeline logic to cope with long clock-to-Q delays of compiler-generated SRAMs. Several factors contribute to improved reduction in critical paths compared to more naive approaches:

1. Instructions are not permitted to stall except in the ID stage for data and known structural hazards.
2. Most hazards that arise in EX or later stages are handled by replaying (re-fetching and re-executing) the instruction upon reaching WB (not unlike how exceptions propagate down the pipeline). One notable case is load-hit speculation, in which an instruction that depends on a load result can be issued before it is known whether the load is a cache hit.
3. Branch conditions are resolved in EX, but the PC is redirected in MEM. The 3-cycle mispredict penalty is mitigated by branch prediction provided by a configurable branch target buffer (BTB), branch history table (BHT), and a return address stack (RAS).
4. Bypass muxes are moved into EX with the selects precomputed in ID; bypass data comes directly from pipeline registers to the extent possible.
5. Some variable-latency operations (e.g., L1D miss, divide) use a scoreboard to track pending register writes. This enables instructions to complete out of program order so that a long-latency operation does not halt the pipeline for subsequent instructions. Consequently, with a non-blocking L1 data cache, multiple misses can be serviced simultaneously.

2.2.2 Cache Hierarchy

The basic unit of replication for a core in Rocket Chip is a *tile*.[2] Each tile consists of one core (Rocket) and a portion of the inner cache hierarchy that is private to each core:

- L1 instruction cache (L1I)
- L1 data cache (L1D) of either a blocking or a non-blocking design
- fully-associative L1 instruction and data TLBs
- optional unified direct-mapped L2 TLB
- hardware page table walker

[2] Although Rocket Chip can generate multi-core instances, this lab will feature only single-tile instances.

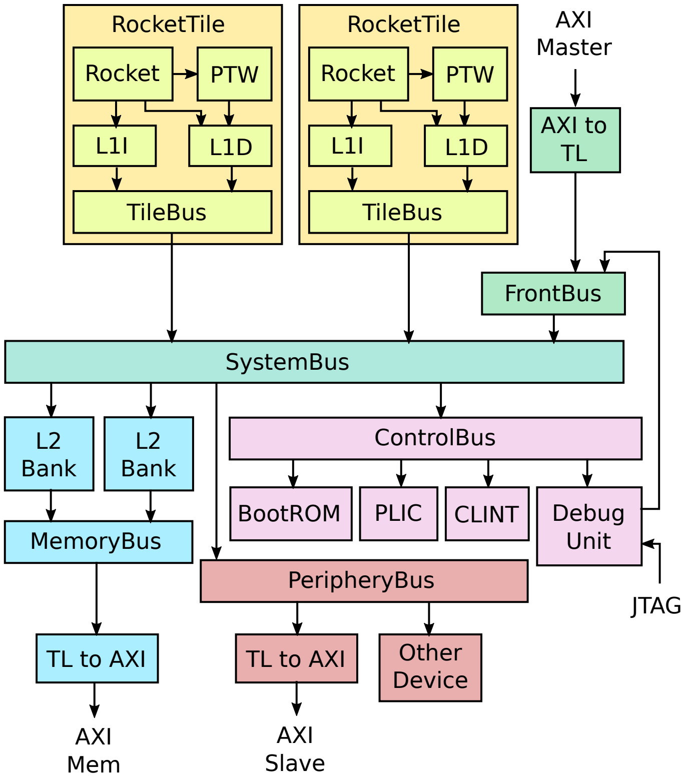

Figure 2: A generic Rocket Chip instance

SoC instances can optionally be configured with a unified, inclusive, multi-banked L2 cache as a last-level cache shared between tiles. If an L2 cache is not present, an L2 broadcast hub is instantiated in its place to maintain coherence between the L1 caches. Each of these structures exposes various parameters such as capacity, associativity, replacement policy, and cache line size, which are set through a Scala-based configuration system at elaboration time.

2.2.3 Outer Memory System

The L2 coherence agent (either the L2 cache or broadcast hub) makes requests to an outer memory system through a AXI4 master port. This top-level port would typically interface with a DRAM controller, but since an actual DRAM controller implementation is not openly available, we instead attach a model that simulates the functional and timing behaviors of a DDR3 memory system. The default SoC configuration presents a single memory channel, but the system can be configured to use multiple channels for greater bandwidth.

3 Directed Portion (30%)

3.1 Terminology and Conventions

Throughout this course, the term *host* refers to the machine on which the simulation runs, while *target* refers to the machine being simulated. For this lab, an instructional server will act as the host, and the RISC-V processors will be the target machines.

UNIX shell commands to be run on the host are prefixed with the prompt "`eecs$`".

3.2 Setup

To complete this lab, `ssh` into an instructional server with the instructional computing account provided to you. The lab infrastruture has been set up to run on the `eda{1..8}.eecs.berkeley.edu` machines (`eda-1.eecs`, `eda-2.eecs`, etc.).

Once logged in, source the following script to initialize your shell environment so as to be able to access to the tools for this lab. Run it before each session.

```
eecs$ source ~cs152/sp22/cs152.lab2.bashrc
```

First, clone the lab materials into an appropriate workspace and initialize the submodules.

! → **It is highly recommended** to work in the local `/scratch` partition to avoid issues with filesystem performance and quotas. Even simulations of modest length (few hundred thousand cycles) can produce a few gigabytes of logs and waveform dumps. Do not use your NFS home directory to avoid slowing down the simulation. Remember that `/scratch` is *not* backed up automatically.

```
eecs$ mkdir -m 0700 /scratch/${USER}
eecs$ cd /scratch/${USER}
eecs$ git clone ~cs152/sp22/lab2.git
eecs$ cd lab2
eecs$ LAB2ROOT="$(pwd)"
eecs$ ./scripts/init-submodules-no-riscv-tools.sh
```

The remainder of this lab will use `${LAB2ROOT}` to denote the path of the `lab2` working tree. Its directory structure is outlined below:

`${LAB2ROOT}`	
`lab/`	
`directed/`	Source code for Lab 2 directed portion
`open1/`	Source code and tools for Problem 4.1
`open2/`	Source code and tools for Problem 4.2
`open3/`	Source code and tools for Problem 4.3
`generators/`	Library of RTL generators
`chipyard/`	SoC configurations

`rocket-chip/`	Rocket Chip generator
`sifive-cache/`	Open-source inclusive L2 cache from SiFive
`sims/`	
`verilator/`	Verilator simulation directory
`generated-src/`	Generated Verilog after Chisel elaboration
`output/`	Simulation logs and traces

3.3 Matrix Transposition Case Study

The directed portion will lead you through a simple case study of a matrix transposition kernel with these objectives:

- Illustrate some basic cache optimization techniques
- Conduct a brief design-space exploration of cache configurations using the Rocket Chip parameterization system
- Familiarize you with the RTL simulation flow

We begin with a naive implementation of matrix transposition in `${LAB2ROOT}/lab/directed/transpose.c` that is derived directly from the mathematical definition. Take a moment to understand the source code. Note that both the 256×64 input matrix and 64×256 output matrix are stored in row-major order. The matrix elements are 64-bit integers.

Compile it into a bare-metal binary:

```
eecs$ TESTDIR=${LAB2ROOT}/lab/directed
eecs$ cd ${TESTDIR}
eecs$ make
```

Next, navigate to the Verilator directory and build the simulator. Notice that the `CONFIG` variable selects the top-level SoC design to generate – what exactly this means will be described in the next section. The Chisel design is elaborated into Verilog RTL, which is then compiled into a cycle-accurate simulator.

```
eecs$ SIMDIR=${LAB2ROOT}/sims/verilator
eecs$ cd ${SIMDIR}
eecs$ make CONFIG=CS152RocketConfig -j4
```

! → This particular configuration contains a 4 KiB direct-mapped L1 data cache and a 4 KiB direct-mapped L1 instruction cache, both with 64-byte cache lines.

Next, run the naive matrix transposition kernel on the simulator:

```
eecs$ make CONFIG=CS152RocketConfig run-binary-hex BINARY=${TESTDIR}/transpose.riscv
```

This will involve a few minutes of waiting, as the entire program takes approximately 2 million target cycles to execute.

The program prints a snapshot of several hardware performance counters in the processor.[3] Use this information to answer the following questions:

(3.3.a) How many cycles does the transpose operation take?

(3.3.b) What is the miss rate of the L1 data cache?

(3.3.c) Why does the naive transpose code result in non-ideal cache performance?

(3.3.d) Which memory access in the code incurs the most misses, and why?

3.4 Cache Blocking

Rewrite the transpose code to employ *cache blocking* (loop tiling) using $B \times B$ blocks. Experiment with a few values of B to determine which factor yields the best performance for a 256×64 input matrix. To maximize B, you may also find that it is necessary to apply a simple *loop interchange* within a block.

(3.4.a) For the given matrix dimensions, what is the optimal blocking factor B?

(3.4.b) What is the performance improvement using cache blocking over the naive code?

(3.4.c) It turns out that the block size $B \times B$ which yields the lowest miss rate is much smaller than what one might expect based solely on the 4 KiB capacity of the L1 data cache. What is the reason for this? (**Hint**: Consider the access pattern within a block, particularly how the rows of the rectangular matrices map to cache sets and which type of cache misses dominate for larger B.)

You can first run your code in a software ISA simulator to more quickly test for correctness, before running in Verilator simulation to gather performance data.

```
eecs$ spike ${TESTDIR}/transpose.riscv
```

This is normally sufficient for debugging software. However, in the unlikely case that a bug manifests only in Verilator, a verbose simulation trace[4] can be found at `${SIMDIR}/output/chipyard.TestHarness.CS152RocketConfig/transpose.out`.

3.5 Cache Parameters

Navigate to `${LAB2ROOT}/generators/chipyard/src/main/scala/config/CS152Configs.scala` and examine the definition of `CS152RocketConfig`.

In Rocket Chip, a *Config* is a Scala class that sets one or more generator parameters to specific values. Configs are additive, can override each other, and can be composed of other Configs. `CS152RocketConfig` is an example of a Config that combines other Configs through the `++` operator. The constituent Configs are applied from right to left (or from

[3] Note that in-flight and recently retired instructions may or may not be reflected when reading the performance counters.

[4] These prints show signals from Rocket's writeback stage each cycle; refer to line 932 of `${LAB2ROOT}/generators/rocket-chip/src/main/scala/rocket/RocketCore.scala` to identify each field.

bottom to top) in the chain, by reverse order of precedence. Thus, a Config appearing to the left of (or above) another Config overrides any parameters previously set by the latter. For more information on the Rocket Chip parameter system, read through the Chipyard documentation.[5]

In `CS152RocketConfig`, change the associativity of the L1 data cache to 2 by modifying `WithL1DCacheWays` parameter, while also adjusting `WithL1DCacheSets` to keep the overall cache size constant. Rebuild the simulator and re-run your blocked matrix transpose version. Repeat this with 4 and 8 ways.

(3.5.a) How does performance and miss rate change when associativity is increased?

(3.5.b) Explain why higher associativity is or is not beneficial for this particular kernel.

3.6 Multi-level Caches

We will continue our experimentation with the `CS152RocketL2Config` design, which is derived from `CS152RocketConfig` but adds a 64 KiB 8-way inclusive L2 cache. (Remember to change the L1 configuration back to a 4 KiB direct-mapped cache.)

Modify your transposition code to introduce another level of cache blocking for the L2. Simulate on `CS152RocketL2Config` and answer the following:

```
eecs$ make CONFIG=CS152RocketL2Config run-binary-hex BINARY=${TESTDIR}/transpose.riscv
```

(3.6.a) Does adding another level of cache blocking improve performance compared to your previous code from 3.5? Why or why not? (**Hint**: Consider whether there is any locality left for the L2 cache to exploit.)

[5] https://chipyard.readthedocs.io/en/latest/Chipyard-Basics/Configs-Parameters-Mixins.html

4 Open-ended Portion (70%)

Select *one* of the following questions per team. The open-ended portion is worth a considerable fraction of the grade for the lab, and the grade depends on how comprehensive the process to your conclusion is.

4.1 Validation and Reverse Engineering of Memory Hierarchies

In this problem, we will try to infer fundamental parameters of a memory system by running user code and measuring execution latency. This is useful for a variety of reasons:

- To help guide application optimizations when the underlying microarchitecture is unknown or undisclosed.
- To validate memory system performance before tape-out. Some of the most insidious bugs in computer system design are performance bugs, since applications still execute correctly but only more slowly. We would like to catch these bugs before committing a design to silicon, but without a performance model of the target machine, they may go undiscovered.
- Using the same principle as above, to help validate simulation models. The common approach of split timing and functional modeling makes it possible to build highly complex cache and memory models – but it is very easy to write "correct" yet fundamentally broken timing models.[6]

We have provided a mystery Rocket Chip configuration for you to characterize, aptly named `CS152RocketMysteryConfig`.

4.1.1 Cache Sizes and Access Latency

We will first run the `caches` micro-benchmark, which comes from the ccbench suite developed by Christopher Celio, to determine the cache sizes and access latency at each level of the cache hierarchy.

`caches` executes a single-threaded pointer chase on an array of a given size, in which each 4-byte array element yields the index of the next element to access.

```
start_cycles = get_cycles();
for (uint32_t k = 0; k < g_num_iterations; k++) {
        idx = arr_n_ptr[idx];
}
end_cycles = get_cycles();
```

As the array exceeds the size of a cache, the sharp increase in misses is observable from the longer time that the benchmark takes to run. To accurately isolate the load latency of an individual element, both spatial locality and memory prefetching must be defeated. This involves striding the indices to point to different cache lines and randomly sorting

[6] For example, in one FASED bug, a write address was mistakenly being used for a read access to the tag array of the last-level cache model. While this would cause a real cache to return incorrect data, it simply manifested as small timing aberration in the model.

the indices within a virtual page (so TLB locality is still maintained).

We have provided a Makefile to automate running ccbench in simulation and visualizing the result. First, sweep across a predefined range of array sizes:

```
eecs$ cd ${LAB2ROOT}/lab/open1
eecs$ make sim -j4
eecs$ make ccbench-sweep -j4
```

This should take approximately **an hour** to finish when using a couple of parallel jobs. Each benchmark run outputs a log statement in the form:

```
App:[caches],NumThreads:[0],AppSize:[1024],Time:[4.01507],TimeUnits:[Cycles Per
    ↪ Iteration],NumIterations:[30000],RunType:[0]
```

"AppSize" records the array size in terms of 4-byte elements, and "Time" records the average cycles spent per iteration. Run the following to extract these lines into a consolidated report file and invoke the plotting script from ccbench:

```
eecs$ make ccbench-plot
```

This generates a plot of cycles per iteration versus array size. Open `${LAB2ROOT}/lab/open1/ccbench/caches/plots/plot-CS152RocketMysteryConfig.pdf` and answer the following:

- L1 D cache size
- L1 D cache latency
- L2 cache size
- L2 cache latency
- DRAM latency

Save the plot for your report.

4.1.2 Other Parameters

Finally, try to empirically deduce some more subtle parameters:

- L1 D cache associativity (or equivalently, cache line size)
- L1 D cache replacement policy
- L1 I cache size
- L1 I cache associativity
- L1 I cache replacement policy
- L1 I TLB reach
- L1 D TLB reach
- L2 TLB reach
- L2 TLB hit latency
- DRAM page policy (open or closed)
- Aggregate DRAM page size (ranks × banks)
- Number of DRAM ranks (you can assume there are 8 banks)

- DRAM Column Address Strobe latency (CAS)
- DRAM Row Address to Column Address delay (RCD)
- DRAM Row Precharge time (RP)

In `${LAB2ROOT}/lab/open1/test/`, two templates are provided to help you begin writing your own micro-benchmarks:

`bmark-p.c` : This executes in a "bare-metal" environment with physical addressing. It is usually the quicker option since bare-metal programs can be loaded directly into the simulation without the initialization overheads of user mode.

`bmark-v.c` : This executes in a user-mode environment with virtual addressing. An initial supervisor program called the *proxy kernel* (`pk`) is required to load the user program and set up paging.

In general, you should avoid accessing arbitrary memory locations that have not been properly allocated, either statically or dynamically with `malloc()` or `mmap()`.

For a rudimentary timer, the templates also define a function that returns the value of the `cycle` CSR, which counts the number of cycles after reset:

```
static inline unsigned long rdcycle(void) {
    unsigned long cycles;
    __asm__ __volatile__ ("rdcycle %0" : "=r" (cycles));
    return cycles;
}
```

`printf()` and other C `stdio.h` functions can be used to print results to stdout/stderr.[7]

Edit the Makefile to add other programs of your own. To build all programs:

```
eecs$ TESTDIR=${LAB2ROOT}/lab/open1/test
eecs$ cd ${TESTDIR}
eecs$ make
```

To run the `bmark-p.riscv` program:

```
eecs$ SIMDIR=${LAB2ROOT}/sims/verilator
eecs$ cd ${SIMDIR}
eecs$ make CONFIG=CS152RocketMysteryConfig run-binary-hex BINARY="${TESTDIR}/bmark-p.riscv"
```

To run the `bmark-v.riscv` program:

```
eecs$ make CONFIG=CS152RocketMysteryConfig run-pk PAYLOAD="${TESTDIR}/bmark-v.riscv"
```

! → Note that the compiled binaries are **not** interchangeable, since different linker options are applied that are specific to whether virtual memory is enabled.

Feel free to test your code on other known configurations, such as the one used in the directed portion or any new ones that you define.

[7] Formatting of floating-point numbers is supported by newlib, the embedded C library.

4.1.3 Submission

For the given `CS152RocketMysteryConfig` instance, report the cache capacities, access latencies, and block sizes as indicated by the `caches` benchmark. (Include the plots from ccbench.)

Then provide your best estimate for as many of the other parameters as possible – aim for *at least five*. For each of those parameters, explain how you measured it, referring to your code as necessary (the code is not counted towards the page limit). If you are not certain that you can accurately determine enough parameters, still provide your code and explain what you tried. More credit will be awarded for a measured and analyzed negative result than an ill-justified guess which might be correct by coincidence. If you have data or plots to show that your code works on known instances, include them in your justification.

Feel free reach out to your GSI if you need help understanding ccbench, Rocket Chip, or anything else regarding this problem.

4.2 Design Your Own Hardware Prefetcher

In this problem, we will build a hardware prefetcher (in either Chisel or C++) in hopes of improving the performance on various benchmarks.

We will use a Rocket Chip instance that has a 16 KiB 4-way set-associative L1 data cache and 64-byte cache lines. Unlike the directed portion, this configuration uses the *non-blocking data cache*, so that Rocket can take advantage of hit-under-miss while a prefetch is being serviced.

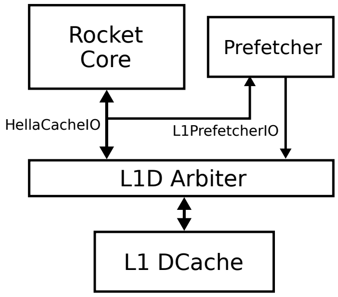

Figure 3: L1 prefetcher integration

4.2.1 Interfaces

Partly for convenience and modularity, the prefetcher is integrated with the tile as a separate module from the cache itself. It can be considered another client of the data cache, like the core.

Start by navigating to `${LAB2ROOT}/generators/rocket-chip/src/main/scala/rocket/L1Prefetcher.scala`. This Chisel file contains our generator framework for creating prefetchers. All prefetcher modules inherit from the `L1Prefetcher` base class, which specifies a common set of I/O ports (grouped together in the `L1PrefetcherIO` bundle).

To observe the stream of memory requests and misses, the prefetcher snoops on the `HellaCacheIO` interface between the core and L1 data cache. A curated subset of signals from the `HellaCacheReq` bundle is presented to the prefetcher:

Input Signal	Description
`io.cpu.req.valid`	Asserted when Rocket's execute stage sends a request to the L1D cache
`io.cpu.req.bits.addr`	Virtual address of the access
`io.cpu.req.bits.cmd`	Memory operation type (e.g., 0=load, 1=store, etc.)
`io.cpu.req.bits.size`	Logarithm of access size (e.g., 0=1 byte, 3=8 bytes)
`io.cpu.miss`	Asserted when a cache miss is being reported to Rocket's writeback stage (**Note**: This is delayed by two cycles after the original request)

The prefetcher has a simplified outgoing interface through which it can inject prefetch requests into the L1D. This is a decoupled interface that uses ready/valid hand-shaking[8] to coordinate the source and sink: Both `io.dmem.req.valid` and `io.dmem.req.ready` must be high during the same cycle to initiate a prefetch.

[8] https://inst.eecs.berkeley.edu/~cs150/Documents/Interfaces.pdf

Input Signal	Description
`io.dmem.req.ready`	Asserted when the L1D can accept a request
`io.dmem.nack`	Asserted when a prefetch request from two cycles ago is rejected, either because all MHSRs are occupied or the request is a secondary miss (**Note**: This could be used to replay a request or throttle the prefetcher)
Output Signal	Description
`io.dmem.req.valid`	Indicates that the request is valid
`io.dmem.req.bits.addr`	Virtual address to prefetch (**Note**: This must be aligned to an 8-byte boundary)
`io.dmem.req.bits.write`	Indicates intent to write

The single port to the L1D is arbitrated between the core, prefetcher, and the page table walker, with the prefetcher being given the lowest priority so as to avoid blocking actual memory requests. However, for more aggressive prefetching schemes, it may be desirable to implement some form of throttling to ensure that prefetches do not excessively occupy the MSHRs (miss status handling registers). For example, prefetches could be rate-limited based on a fixed interval or a feedback loop that adapts to miss rate.

For reference, the `ExampleL1Prefetcher` module is provided as a demonstration on how to use the `L1PrefetcherIO` interface described above. This is a naive implementation of the one-block prefetch-on-miss scheme from lecture, but its simplistic design actually turns out to be quite ineffectual, causing a moderate performance degradation more often than it helps. Hopefully yours is a superior solution!

! → Once you understand how the interfaces work, implement your own hardware prefetcher within the empty `CustomL1Prefetcher` module provided in `L1Prefetcher.scala` (search for a TODO comment).

4.2.2 C++ Modeling

As an alternative to writing Chisel, you also have the option of implementing your prefetcher in C++ as a software model that is co-simulated with Rocket Chip. This method uses SystemVerilog DPI (Direct Programming Interface) to enable a Verilog wrapper module to call C++ functions.

Start by navigating to `${LAB2ROOT}/generators/rocket-chip/src/main/resources/csrc/L1Prefetcher.cc`. This C++ file contains two functions to work with:

Function	Description
`prefetcher_init()`	This is called at the beginning of simulation and can be used to initialize global state.
`prefetcher_tick()`	This is called for each clock cycle and is where the bulk of your prefetcher logic will reside. The function signature matches the I/O ports described above. Values for output signals (`dmem_req_*`) are assigned by dereferencing the argument pointers. Read the important note below about output signal timing.

! → Treat assigning to `*dmem_req_valid`, `*dmem_addr`, and `*dmem_req_write` as if the values are being latched by registers, and avoid combinationally coupling them with `dmem_req_ready`. If, for example, `dmem_req_valid` is asserted only when `dmem_req_ready` is true, the outputs will be improperly delayed by one cycle relative to `dmem_req_ready`, which will cause the prefetch request to be ignored by the cache.

4.2.3 Building

To use your custom hardware prefetcher, first modify the `WithL1Prefetcher` Config defined in `${LAB2ROOT}/generators/chipyard/src/main/scala/config/CS152Configs.scala`. Replace the default `ExampleL1Prefetcher` instantiation with the appropriate module, either `CustomL1Prefetcher` (Chisel) or `ModelL1Prefetcher` (C++), like so:

```
class WithL1Prefetcher extends Config((site, here, up) => {
  case BuildL1Prefetcher =>
    Some((p: Parameters) => Module(new CustomL1Prefetcher()(p)))
})
```

A specific top-level configuration for this problem (`CS152RocketPrefetchConfig`) has already been prepared for you, which includes the `WithL1Prefetcher` Config.

To build the simulator:

```
eecs$ SIMDIR=${LAB2ROOT}/sims/verilator
eecs$ cd ${SIMDIR}
eecs$ make CONFIG=CS152RocketPrefetchConfig -j4
```

4.2.4 Simulating

First see how your prefetcher performs on the matrix transposition kernel from the directed portion. You may choose to use the naive code or your L1 cache-blocked version.

```
eecs$ cd ${SIMDIR}
eecs$ make CONFIG=CS152RocketPrefetchConfig run-binary-hex BINARY="${LAB2ROOT}/lab/
    ↪ directed/transpose.riscv"
```

Next test your prefetcher on the suite of benchmarks from Lab 1.[9] These also print a snapshot of the hardware performance counters – note that L1D misses from regular and prefetch requests are counted separately.

```
eecs$ make CONFIG=CS152RocketPrefetchConfig run-bmark-tests
```

Lastly, as an example of a more complex application, we have also included the Graph Algorithm Performance Benchmark Suite (GAPBS) [4], which consists of portable, high-performance implementations for six fundamental graph algorithms developed by Scott Beamer. Specifically, our focus is on the direction-optimizing variant of Breadth-First Search (BFS). Smaller inputs (Kronecker graphs with 2^{10} vertices) will be used here, as the reference inputs such as real social network graphs are too intensive in memory requirements and simulation time.

```
eecs$ cd ${LAB2ROOT}/lab/open2
eecs$ make
eecs$ cd ${SIMDIR}
eecs$ make CONFIG=CS152RocketPrefetchConfig run-bfs
```

As in the directed portion, the simulation traces can be found at `${SIMDIR}/output/chipyard.TestHarness.CS152RocketPrefetchConfig/*.out` based on benchmark name.

4.2.5 Debugging

To dump waveforms from simulation, run the debug versions of the `make` targets:

```
eecs$ make CONFIG=CS152RocketPrefetchConfig run-binary-debug-hex BINARY="
    ↪ ${LAB2ROOT}/lab/directed/transpose.riscv"
eecs$ make CONFIG=CS152RocketPrefetchConfig run-bmark-tests-debug
eecs$ make CONFIG=CS152RocketPrefetchConfig run-bfs-debug
```

Waveform dumps (which can become quite large) are written to `${SIMDIR}/output/chipyard.TestHarness.CS152RocketPrefetchConfig/*.vpd`. The prefetcher instance is found under `TOP.TestHarness.chiptop.system.tile_prci_domain.tile_reset_domain.tile.prefetchOpt` in the module hierarchy. Waveforms can be viewed on the instructional servers with the DVE application (requires X11 forwarding over `ssh` or X2Go):

```
eecs$ dve &   # ‘&’ backgrounds the process
```

4.2.6 Submission

Report performance metrics and cache statistics from running the various benchmarks with the prefetcher enabled. Compare these to results gathered from running on the baseline `CS152RocketNoPrefetchConfig` system, which omits the prefetcher but is otherwise identical. Include the source code for your implementation in an appendix (not counted towards the page limit).

[9] You can run the benchmarks in parallel by adding the `-j` *N* flag to the `make` command, but refrain

In your report, describe your design and any implementation challenges in detail. Here are some suggestions to consider in your evaluation:

- What memory access patterns or instances of locality were you targeting?
- Explain your design rationale and the various approaches that you considered. What worked and what did not?
- Analyze the impact on miss rate and CPI. Were any results surprising?
- Optionally, see if you can characterize your prefetcher on the set of metrics introduced in Lecture 7:

  accuracy = useful prefetches / total prefetches

  coverage = total prefetches / total unique accesses

  timeliness = number of prefetches arriving on time / total prefetches

  It may be useful to instrument your prefetcher with Chisel `printfs`[10] or C++ `std::cout` statements to log certain events and parse the trace with a script.

A negative result is perfectly acceptable so long as you reason about why the outcomes differed from your expectations. (Designing an effective prefetcher is a non-trivial task! This exercise is partly meant to underscore the challenges of prototyping an idea.)

Feel free reach out to your GSI if you need help understanding Chisel, Rocket Chip, or anything else regarding this problem.

from spawning an excessive number of jobs so as to be fair to other users. $N = 4$ is probably acceptable.

[10] `https://github.com/chipsalliance/chisel3/wiki/Printing-in-Chisel`

4.3 Design Your Own Replacement Policy and Victim Cache

For this problem, we would like to investigate whether a different cache replacement policy, combined with a victim cache, would improve the performance of five selected SPEC benchmarks compared to random replacement.

Assume you are designing for a 16 KiB 4-way set-associative L1 data cache, where the backside is connected to DRAM. You will model your cache modifications in `spike`, a functional ISA simulator for RISC-V that has been extended with a basic cache model. The simulator feeds memory addresses through a simulated cache (with a given size, associativity, and block size) to compute the number of accesses, hits, and misses. While `spike` does *not* model microarchitectural timings and is therefore not cycle-accurate, its speed lets us execute much longer programs ordinarily infeasible in RTL simulation.

! → The only constraints are that you can only add *less than 2^{11} bits* of state (in either flip-flops or SRAM) to support your new replacement policy and *less than 2^{13} bits* for the victim cache. Assume that physical addresses are 56 bits wide.

4.3.1 SPEC CPU2006

The SPEC CPU2006 package is a former[11] industry-standard benchmark suite for evaluating general-purpose processors, memory systems, and compilers [5]. You will be running five benchmarks from SPECint (integer) and SPECfp (floating-point) on smaller *test* inputs.[12] Brief descriptions of them, taken from the SPEC documentation, follow:

- `401.bzip2` is based on bzip2 version 1.0.3, modified to perform compression and decompression in memory instead of file I/O.
- `429.mcf` is derived from MCF, a program used for single-depot vehicle scheduling in public mass transportation. It features a specialized version of the simplex algorithm for network flow problems.
- `450.soplex` is based on SoPlex 1.2.1. It solves a linear program using a simplex algorithm and sparse linear algebra.
- `458.sjeng` is based on Sjeng 11.2, a program that plays chess and several chess variants. It attempts to find the best move via a combination of alpha-beta or priority proof number tree searches, advanced move ordering, positional evaluation and heuristic forward pruning.
- `470.lbm` implements the "Lattice Boltzmann Method" to simulate incompressible fluids in 3D.

[11] It has since been replaced by SPEC CPU2017; however, we still opt to use SPEC CPU2006 for being much simpler to cross-compile for RISC-V.

[12] Each benchmark with reference inputs generally require a day to run on a typical Rocket Chip instance mapped to an FPGA. In this case, we are more interested in stressing the caches than reporting valid benchmark scores, so minor adjustments have been made to limit simulation time.

To build spike and simulate all benchmarks using its cache model:

```
eecs$ cd ${LAB2ROOT}/lab/open3
eecs$ make spike -j4
eecs$ make run
```

The benchmarks should take around a total of 15 minutes to execute when run serially. For quicker testing, individual programs can be re-run with make run-X, where X is the name of the benchmark without the numerical prefix (e.g., run-bzip2, run-mcf, etc.).

The simulation output is recorded in ${LAB2ROOT}/lab/open3/CPU2006/build.riscv/*.out, and the compiled SPEC binaries can also be found in that same directory. Removing these *.out files forces make to re-run the simulations later:

```
eecs$ make clean-run
```

A full rebuild can be triggered by purging all generated files:

```
eecs$ make clean
```

4.3.2 Modifying the Cache Simulator

Navigate to ${LAB2ROOT}/lab/open3/riscv-isa-sim/riscv/cachesim.cc, where you will find the definition for the cache_sim_t C++ class instantiated in spike. After taking some time to understand how the current cache model operates, modify the cache_sim_t::victimize() function to implement your custom replacement policy.

To rebuild the simulator without running any benchmarks:

```
eecs$ cd ${LAB2ROOT}/lab/open3
eecs$ make spike
```

4.3.3 Adding a Victim Cache

Generally, the associativity of a cache (number of ways) presents a trade-off between access time and conflict misses. In order to reduce conflict misses without affecting access times, N. Jouppi proposed *victim caching* [6] in which a small fully-associative secondary cache, called a victim cache, is added to a direct-mapped L1 cache to hold recently evicted cache lines.

We are interested in whether a victim cache would be worthwhile to implement even for a set-associative cache with a moderate number of ways. To get a sense of the scope for potential improvement, you may want to first augment cache_sim_t to track the number of conflict misses. Then implement your own victim caching scheme inside the cache model. You will likely need to modify the cache_sim_t::access() function, which calls cache_sim_t::victimize().

4.3.4 Submission

In your report, describe how your cache replacement policy and victim cache work, using visual aids (e.g., block diagrams) where appropriate to illustrate their operation. Include a diff of your modifications to the cache model in an appendix (not counted towards the page limit).

Report cache statistics from running the SPEC benchmark suite on your modified cache, and compare them to the original cache. Try to explain your results as best you can. Here are some suggestions to consider in your analysis on the effectiveness of your design:

- Based on the number of memory accesses, misses, and instructions retired, what effect on AMAT and CPI do you think your new cache would have?
- Estimate the cost in resources if you were to implement your design in hardware. How much additional state would be required in terms of bits?
- For the given set of benchmarks, how problematic are conflict misses compared to compulsory or capacity misses? Would the addition of a victim cache be justifiable?
- What is the minimum number of victim cache entries that you would recommend? Are there diminishing returns to increasing victim cache size?
- Program behavior can sometimes be characterized by distinct execution "phases". Does the miss rate vary over time within a benchmark?

Feel free to reach out to your GSI if you need help understanding the ISA simulator, the cache model, or anything else regarding this problem.

5 Feedback Portion

In order to improve the labs for the next offering of this course, we would like your feedback. Please append your feedback to your individual report for the directed portion.

- How many hours did the directed portion take you?
- How many hours did you spend on the open-ended portion?
- Was this lab boring?
- What did you learn?
- Is there anything that you would change?

Feel free to write as much or as little as you prefer (a point will be deducted only if left completely empty).

5.1 Team Feedback

In addition to feedback on the lab itself, please answer a few questions about your team:

- In a few sentences, describe your contributions to the project.
- Describe the contribution of each of your team members.
- Do you think that every member of the team contributed fairly? If not, why?

6 Acknowledgments

This lab was heavily inspired by the previous set of CS 152 labs developed by Henry Cook, Yunsup Lee, and Andrew Waterman, which targeted functional simulators such as Simics and Spike. More recent iterations of this lab, developed by Donggyu Kim, David Biancolin, and Albert Magyar, used FireSim to run FPGA-based simulations on Amazon EC2 F1.

References

[1] K. Asanović, R. Aviúzienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016.

[2] A. Waterman and K. Asanović, Eds., *The RISC-V instruction set manual, volume I: User-level ISA*, Version 20191213, RISC-V Foundation, Dec. 2019. [Online]. Available: `https://riscv.org/specifications/`.

[3] A. Waterman and K. Asanović, Eds., *The RISC-V instruction set manual, volume II: Privileged architecture*, Version 20190608-Priv-MSU-Ratified, RISC-V Foundation, Jun. 2019. [Online]. Available: `https://riscv.org/specifications/privileged-isa/`.

[4] S. Beamer, K. Asanović, and D. Patterson, *The GAP benchmark suite*, 2015. arXiv: `1508.03619 [cs.DC]`. [Online]. Available: `http://gap.cs.berkeley.edu/benchmark.html`.

[5] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, 1–17, Sep. 2006. DOI: `10.1145/1186736.1186737`.

[6] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *SIGARCH Computer Architecture News*, vol. 18, no. 2SI, 364–373, May 1990. DOI: `10.1145/325096.325162`.