CS152 Laboratory Exercise 2 (Version 1.0.2)

Professor: Krste Asanović
TA: Donggyu Kim and Howard Mao
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

February 16, 2018

1 Introduction and goals

The goal of this laboratory assignment is to conduct memory hierarchy experiments by running realistic workloads on RocketChip [1]. To enable RTL-based simulation using FPGAs, we have already generated performance simulators by automatically transforming RocketChip [2, 3]. We will run these performance simulators in Amazon EC2 F1 instances (`https://aws.amazon.com/ec2/instance-types/f1`) to collect RocketChip's memory system stats for a subset of the SEPC2006int benchmark suite (`https://www.spec.org/cpu2006/`). You will also make architectural design decisions based on the results.

The lab has two sections, a directed portion and an open-ended portion. Everyone will perform the directed portion in the same way, and grades will be assigned based on correctness. The open-ended portion will allow you to pursue more creative investigations, and your grade will be based on the effort made to complete the task or the arguments you provide in support of your ideas.

For both the directed potion and the open-ended portion of this lab, you must work in *a group of two (not three)*. You are also encouraged to discuss solutions to the lab assignments with other groups, but must run through the lab by yourselves and turn in *a hard copy of your own lab report in the beginning of the lecture on March 5th.*

You are only required to do one of the open-ended assignments. These assignments are generally starting points or suggestions. Alternatively, you can propose and complete your own open-ended project as long as it is sufficiently rigorous. If you feel uncertain about the rigor of a proposal, feel free to consult the instructor or the TAs.

1.1 Graded Items

You will turn in a hard copy of your results to the instructor or TA. Please label each section of the results clearly. The directed items need to be turned in for evaluation. You only need to turn in *one* of the problems found in the open-ended portion.

1. (Directed) Section 3.5: Analysis on Cache Statistics
2. (Directed) Section 3.6: Performance Modeling with Microarchitectural Events
3. (Open-ended) Section 4.1: Design a Memory Hierarchy with a 5mm^2 Area Budget
4. (Open-ended) Section 4.2: Validation of Memory Hierarchies

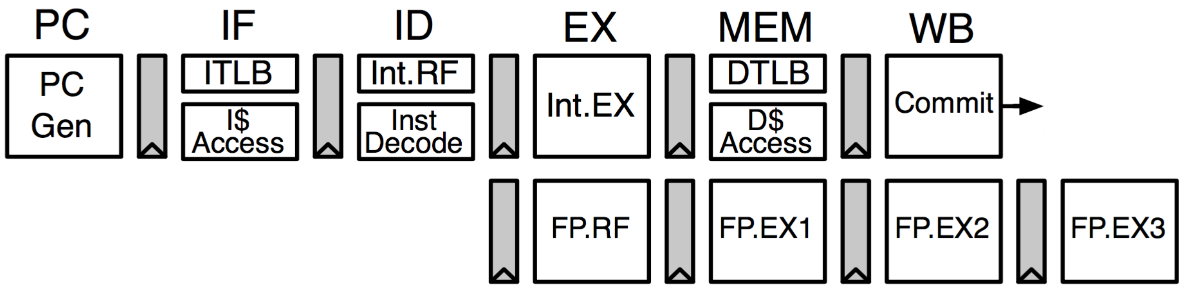

Figure 1: The Rocket Core Pipeline.

5. (Directed) Section 5: Feedback

Lab reports must be in *readable* English and not raw dumps of log-files. It is *highly* recommended that your lab report be typed. Charts, tables, and figures - when appropriate - are great ways to succinctly summarize your data.

2 Background

2.1 Rocket Chip

RocketChip [1] is an open-source SoC generator suitable for research and industrial purposes. Rather than being a single instance of an SoC design, RocketChip is a hardware design generator, capable of producing many design instances from a single piece of Chisel [4] source code. Multiple industry products as well as silicon prototypes are manufactured using RocketChip. A RocketChip instance generally consists of three major components: processors, a cache hierarchy, and an uncore.

Rocket Chip instantiates an in-order processor, Rocket, by default, but also supports various core implementations. Rocket is a 5-stage in-order processor (Figure 1) that implements the RISC-V ISA [5, 6]. Its cache hierarchy includes L1 instruction caches, L1 blocking data caches, and fully associative L1 TLBs and a direct-mapped L2 TLB with configurable sizes, associativities, and replacement policies. In this lab, we provide pre-built FPGA images for various configurations. For now, RocketChip does not provide an L2 cache implementation yet, so we adopt an abstract L2 cache model instead [3]. This cache model is runtime configurable, so we do not need different FPGA images for different L2 cache parameters.

2.2 The SPEC CPU2006 Benchmark Suite

The SPEC CPU2006 benchmark suite (https://www.spec.org/cpu2006/) *was* widely used to evaluate real systems as well as design ideas in computer architecture research. This benchmark suite includes real-world application which execute *trillions of instructions* with their reference inputs. In this lab, we will only use a subset of benchmarks with their test inputs as follows:

- `400.perlbench`: Cut-down version of Perl v5.8.7, the popular scripting language.
- `401.bzip2`: Modified Julian Seward's bzip2 version 1.0.3, where all compression and decompression happens entirely in memory.

- `403.gcc`: C language compiler gcc version 3.2, which generates code for an AMD Opteron processor.
- `429.mcf`: Combinatorial optimization for single-depot vehicle scheduling in public mass transportation.

SPEC CPU2006 will retire soon. Why not SPEC CPU2017? (Un)fortunately, SPEC CPU2017 is more realistic, requiring longer execution times. We just wanted to save time and money. However, we selected benchmarks that also exist in SPEC CPU2017.

2.3 FPGA-based Performance Simulation with Amazon EC2 F1 Instances

We evaluated various pipelines for very small benchmarks using software-based performance simulators for Lab1. However, software-based simulation does not provide sufficient performance to evaluate complex hardware designs for realistic software applications.

Instead, any RTL designs can be directly mapped and emulated in the FPGA at speed. However, we need more accurate and runtime configurable timing models for memory systems and devices that will be implemented as an abstract RTL model or a software model. For this reason, RocketChip RTL implementations are automatically transformed and instrumented to generate performance simulators running in the FPGA [2, 3], enabling efficient communications between FPGA and software.

There are increasing interests in using FPGAs for application-specific accelerators. As a result, cloud service providers decided to offer FPGA cloud instances such as Amazon EC2 F1 instances. We use these services to quickly evaluate real-world hardware designs for real-world software applications.

We pre-built FPGA-based performance simulators for various cache parameters and provide them as Amazon FPGA images (AFIs) that can be loaded into any F1 instances. We also provide the Amazon machine image (AMI) that contains necessary software binaries and scripts to run simulations.

3 Directed Portion (30%)

3.1 Launching an F1 Instance

We will launch an F1 instance to start this lab. First, log in to an instructional machine (`icluster{6-9}.eecs.berkeley.edu`). Then, to enable commands to control your F1 instances, run:

```
# You can add this line in ~/.bash_profile
inst$ source ~cs152/sp18/cs152.lab2.bashrc
```

Next, let's launch an F1 instance:

```
inst$ launch-f1 | tee <file name>
wait until running
running now!
Instance ID: i-07ab2ee7a526ccacb
```

```
IP Address : 34.227.49.244
Keep Instance ID & IP Address
Please shut the instance down in 8 hours.
wait for initialization
initializing
...
initializing
ok
It's time to SSH into your instance
```

It will take for a while (~3 min) for initialization, so please be patient. Also, note that you will need the instance id and the IP address across this lab. Most importantly, **each student** is allowed to launch **single instance at a time** as many times as possible, but the total instance-hours should not exceed 40 hours. (Therefore, *80 hours are allowed for each group in total.*) If you violate any rule, you will get a severely penalty. (50 % with the first violation and 100 % with the second violation. Therefore, you don't have to do this lab if you violated the rules twice.)

3.2 Linux Boot and Hello World

Now, let's SSH into the F1 instance:

```
inst$ ssh centos@<IP Address>
```

Once you SSH into your F1 instance, move into `cs152-lab2`:

```
$ cd cs152-lab2
```

This directory contains necessary files to conduct this lab. First of all, we should load an AFI image:

```
# 16KiB L1$, L1 TLB reach = 128KiB, L2 TLB reach = 4MiB
$ ./load-fpga.sh agfi-0aa6f0423f0ff7843
```

This will load a RocketChip simulator for 16KiB L1 I/D caches, fully-associative L1 I/DTLBs with 32 entries, a direct-mapped L2 TLB with 1024 entries.

We provide you a make file command to conveniently run RISC-V binary images:

```
# Default argument values
# L1_SIZE ?= 16KB (design parameter)
# L1_TLB_REACH ?= 128KB (design parameter)
# L2_TLB_REACH ?= 4MB (design parameter)
# L2_WAY_BITS ?= 2 (2^2 = 4 ways, runtime parameter)
# L2_SET_BITS ?= 12 (2^12 = 4096 sets, runtime parameter)
# L2_BLOCK_BITS ?= 6 (2^6 = 64 Bytes, runtime parameter)
$ make BIN=<RISCV binary image>
```

This command creates a directory named `output/<cache parameters>`, runs the image in the simulation, pipes `stdout` and `stderr` to `output/<cache parameters>/<RISCV binary image>.{out, err}`, and dumps the cache (and branch) statistics to `output/<cache parameters>/<RISCV binary`

`image>.stat`. (Read `Makefile` for details.) This will be useful when you run simulations for various cache configurations.

Now, let's boot Linux and print `Hello CS152!` in the simulator. Run:

```
$ make BIN=images/bblvmlinux-hello
```

You can see linux boot messages and `Hello CS152!` in the screen. It also prints performance counter values, which are also saved in `outputs/16KiB-128KiB-4MiB-2-12-6/bblvmlinux-hello.stat`:

```
##  cycles             = 19024725
##  instret            = 8995954
##  loads              = 674618
##  stores             = 430790
##  L1 I$ misses       = 95952
##  L1 D$ misses       = 68632
##  L2$ misses         = 20942
##  ITLB misses        = 1674
##  DTLB misses        = 2450
##  L2 TLB misses      = 2765
##  branches           = 390234
##  branch mispredicts = 87680 // Mispredicts for branch directions (taken / not taken?)
##  target mispredicts = 71764 // Mispridicts for control flow target addresses
```

Can you compute the CPI, the miss rates and the misses per kilo instructions (MPKIs) of miss events for `bblvmlinux-hello`?

Warning: `make claen` will delete *all* generated output files. So, ask yourself carefully if you really want to delete all the files before you run into a disaster.

3.3 Cache Parameter Sweep for SPEC CPU2006

You will collect cache and branch stats for SPEC CPU2006 by running `images/bblvmlinux-{400.perlbench, 403.gcc, 429.mcf}`. Note that these benchmarks run on top of Linux where TLBs play an important role for system performance.

To automate simulation runs for various cache configurations, we will run a script, `sweep.py`. This script loads AFI images for each cache configuration and runs simulations for each benchmark sequentially. Table 1 shows AFIs for different cache configurations. Also, for `agfi-09d2f1c836f0ad278` (L1 \$ size = 32 KiB, L1 TLB reach = 128 KiB, L2 TLB reach = 4 MiB) in Table 1, we apply L2 cache parameters in Table 2. Otherwise, we only have the 1MiB 8-way L2 cache with 128 byte lines (the second parameter in Table 2). Please take a look and figure out how this script runs simulations using the makefile command.

Once you understand what is going to happen, it is time to run the script to sweep cache parameters:

```
# To prevent simulations gone while you are out
$ tmux
# Also, good to measure how long it took
$ time ./sweep.py
```

AFI	L1 $ assoc.	L1 $ sets	L1 $ block size	L1 TLB reach	L2 TLB reach
`agfi-09161206020478060`	4	32	64 bytes	128 KiB	4MiB
`agfi-0aa6f0423f0ff7843`	4	64	64 bytes	128 KiB	4MiB
`agfi-09d2f1c836f0ad278`	8	64	64 bytes	128 KiB	4MiB
`agfi-0e31b1450453a3d68`	4	64	64 bytes	128 KiB	2MiB
`agfi-09dfe6ac51b81c7f6`	4	64	64 bytes	128 KiB	No L2 TLB
`agfi-0c0aced8dc57ef135`	4	64	64 bytes	64 KiB	No L2 TLB
`agfi-0815d9f8a9ed5ef16`	4	64	64 bytes	32 KiB	No L2 TLB

Table 1: AFIs for various cache configurations

Size	Associativity	Sets	Block Size (Bytes)
1MiB	8	512	256
1MiB	8	1024	128
1MiB	8	2048	64
1MiB	8	4096	32
1MiB	2	4096	128
1MiB	4	2048	128
512KiB	8	512	128
2MiB	8	2048	128
4MiB	8	4096	128

Table 2: L2 cache configurations simulated with `agfi-09d2f1c836f0ad278` in Table 1

It will run for a long time (~7 hours). You can hang out while the machine is working, but make sure you come back on time. (Otherwise, you will blow credits out and have Donggyu go bankrupt!) If you want to return to the screen in the middle, SSH into F1 instance again and run:

```
$ tmux a
```

3.4 Terminating Your Instance

Once, you finish the experiments. Copy the output files to the instructional machine as follows:

```
inst$ scp -r centos@<IP Address>:~/cs152-lab2/outputs ./
```

You will have the whole output files in the current directory. Next, shut down your instance with the following command in **an instructional machine**:

```
inst$ terminate-f1 <Instance Id>
```

Once again, you should terminate an instance on time to avoide a penalty.

3.5 Analysis on Cache Statistics

Now, it's time to analyze the output files of long simulations. We provide you a script to convert all output files into CSV files for each benchmark. Run:

```
cd outputs
cp ~cs152/sp18/lab2/analyze.py ./
python analyze.py
```

Using the CSV files, answer the following questions. Assume the L1 cache access time is 1 cycle, the L2 cache access time is 23 cycles, and the main memory access time is 100 cycles. You may want to modify the script for each question.

1. How does the L1 cache size affect system performance? Report miss rates, MPKIs, AMATs (in cycles), and CPIs for different L1 cache sizes by fixing other parameters.
2. How does the L2 cache size affect system performance? Report miss rates, MPKIs, AMATs (in cycles), and CPIs for different L2 cache cache sizes by fixing other parameters.
3. How does the L2 cache associativity affect system performance? Report miss rates, MPKIs, AMATs (in cycles), and CPIs for different L2 cache associativities by fixing other parameters.
4. How does the L2 cache block size affect system performance? Report miss rates, MPKIs, AMATs (in cycles), and CPIs for different L2 cache block sizes by fixing other parameters.
5. L1 TLBs cannot be as large as you want because it is fully-associative. How important their reaches are if there is no L2 TLB?
6. Do you think a direct-mapped L2 TLB is crucial for system performance? Present the evidence.

3.6 Performance Modeling with Microarchitectural Events

We can approximate CPI with the following equation:

$$CPI = CPI_{base} + \sum_{e \in \{events\}} MPI_e \times PENALTY_e$$

where MPI_e is misses per instruction for event e and $PENALTY_e$ is the miss penalty for event e. (You can easily see why we computed MPKIs in Section 3.5). Table 3 shows the miss penalties for microarchitectural events.

Note that RocketChip supports three-level page tables (Section 4.3 in [6]). Thus, when there is an L2 TLB miss, the hardware page table walker (PTW) accesses a cache for page table entries (PTEs) first (let's assume its hit rate is 2/3) and the L2 cache and main memory if there is a miss in the cache. The PTW repeats it three times to access a leaf table to obtain the physical page number.

The Rocket core also predicts branch conditions as well as target addresses for control flow instructions. When these predictions are wrong, PC is redirected from the memory stage, and thus their penalties are 3 cycles in most cases.

Assuming $CPI_{base} = 1.3$ (why not 1?), predict CPIs for various cache parameters across benchmarks using the data from Section 3.3 and compare them against the actual CPIs from the simulations. How close are the predicted CPIs to the actual CPIs? What information do you want to know for more accurate performance predictions?

Microarchitectural events	Miss penalty (cycles)
L1 cache miss	23 (L2 cache access latency)
L2 cache miss	100 (DRAM access latency)
L1 TLB miss	2
L2 TLB miss	$3 \times (2/3 \times 1 + 1/3 \times 123)$
Branch condition mispredict	3
Target address mispredict	3

Table 3: Miss Penalties for microarchitectural events

4 Open-ended Portion (70%)

Pick *one* of the following questions. The open-ended portion is worth a large fraction of the grade of the lab, and the grade depends on how comprehensive the process to your conclusion is. Therefore, please spend the appropriate amount of time and energy on it. Also, have fun with it!

4.1 Design a Memory Hierarchy with a 5mm^2 Area Budget

For this question, we want to figure out how we can make the best use out of our 5mm^2 area budget for caches. (Chips cannot be arbitrarily large!) Unlike the directed potion, cycle time matters now.

We will assume the critical path is on L1 caches or TLBs. (In reality, this is very likely). Therefore, the access times for L1 caches and TLBs determine the cycle time of the CPU. L2 cache can be accessed in multiple cycles. Of course, you can just decide L2 cache must be accessed in one cycle by slowing down the cycle frequency.

We will use CACTI [7] to estimate the access times and the areas for given cache configurations.

4.1.1 Finding Access Times and Areas for Caches

You can use CACTI in your computer, but we recommend instructional machines. First, clone the CACTI repository and compile CACTI:

```
inst$ git clone https://github.com/donggyukim/cacti.git
inst$ cd cacti
inst$ make
```

In `cs152-sp18-configs`, there are `L1Icache.cfg`, `L1Dcache.cfg`, `L2cache.cfg`, `TLB.cfg`, which are configuration files for L1 instruction caches, L1 data caches, L2 caches, and L1 TLBs, respectively. In this question, we do not have L2 TLBs anymore.

Edit these files for different cache configurations. Table 4 shows available configurations for the L1 instruction and data caches and the fully-associative L1 TLBs. Table 5 shows available parameters for L2 caches.

Once you edit the configuration file for each cache, run the following command to obtain the access time and the area:

```
inst$ make run CFG=<configuration file>
```

The result will be saved in `cs152-sp18-outputs/`. Note that results are appended to the output file when you repeat it for the same configuration file.

AFI	L1 $ size	L1 $ assoc.	L1 $ sets	L1 $ block size	L1 TLB reach
agfi-069fb09e3da3fa862	8 KiB	4	32	64 bytes	32 KiB
agfi-03568b559c45b97db	8 KiB	4	32	64 bytes	64 KiB
agfi-08c58a468273a0b0d	8 KiB	4	32	64 bytes	96 KiB
agfi-07e41bc03370a114e	8 KiB	4	32	64 bytes	128 KiB
agfi-0b8262a4e576cb828	16 KiB	4	64	64 bytes	32 KiB
agfi-0e5ecbb2f1d6e3f4b	16 KiB	4	64	64 bytes	64 KiB
agfi-0710d00dec992ca63	16 KiB	4	64	64 bytes	96 KiB
agfi-0712256c0497219a9	16 KiB	4	64	64 bytes	128 KiB
agfi-080e11351b52378e9	32 KiB	8	64	64 bytes	32 KiB
agfi-085646b4525462028	32 KiB	8	64	64 bytes	64 KiB
agfi-0aaa8cdf3586864fd	32 KiB	8	64	64 bytes	96 KiB
agfi-0f0209296119ea2c2	32 KiB	8	64	64 bytes	128 KiB

Table 4: Available L1 cache and TLB configurations

Parameters	Values
Associatively	1, 2, 4, 8
Number of Sets	Up to 4096 (power of 2)
Block Size	Up to 256 bytes (power of 2)

Table 5: Available L2 cache parameters

Compute the cycle time assuming the critical path is on L1 caches or TLBs. L2 cache can be accessed in multiple cycles, so you should compute the L2 cache latency in cycles by dividing its access time by the cycle time. (Of course, it is one cycle if you decide the cycle time be the L2 cache access time.)

If you want to flush all output files, run:

```
inst$ make clean
```

4.1.2 Running Simulations

Once you decide cache configurations from CACTI, it's time to validate your design ideas through simulations. Launch an F1 instance as in Section 3.1 and SSH into the instance.

We provide you a script, `run.py`, to run a single simulation for each benchmark. Open the script and edit it if necessary. In the script, you are supposed to select an AFI for L1 cache and TLB configurations (Table 4) and change L2 cache parameters. **You must change L2 cache latency in cycles to the value that is computed from Section 4.1.1**.

After editing the script, run a simulation for each benchmark with the following command:

```
$ ./run.py <benchmark image path>
```

4.1.3 Submission

The goal is to find the cache configurations that minimize CPI × cycle time. (Recall the Iron Law.) You should submit a report that explains how you can get the optimal cache configurations. Your argument should be supported by simulations through all 4 SPEC benchmarks used in this lab. In addition, please submit the configuration files from Section 4.1.1 to Donggyu (`dgkim@eecs.berkeley.edu`).

4.2 Validation of Memory Hierarchies

For this question, we will validate memory hierarchies of simulated systems. We have some motivations for this validation:

- How do we know cache configurations if you are not told in advance?
- Can we believe the simulated systems correctly implement memory hierarchies we desire?
- (Most importantly,) How can we trust Donggyu's L2 cache model? We should suspect his one-person show in this lab.

4.2.1 Getting Started and Easy Questions

To validate the memory hierarchy, we will use the `caches` benchmark in `ccbench` [8] developed by Christopher Celio, who is also the author of BOOM used in Lab 3.

We provide you a pre-compiled binary for the `caches` benchmark in the instance. Launch an instance as in Section 3.1 and SSH into the instance. Next, move to `cs152-lab2/ccbench` and find `run.py`. Open it and figure out what this script does.

Whenever ready, run:

```
$ ./run.py <AFI image>
```

The AFI image can be any image in Table 1 and 4. This will execute the cache benchmark multiple times with different arguments and generate an output file named `caches.report.txt`.

`ccbench` also provides a script to visualize the output file. First, clone the repository in *an instructional machine*:

```
inst$ git clone https://github.com/donggyukim/ccbench.git
inst$ cd ccbench/caches/
```

Then copy the output file to the instructional machine and run a script:

```
inst$ scp centos@<ip address>:~/cs152-lab2/ccbench/cache.report.txt reports/
inst$ ./run-test.py -r reports/cache.report.txt
```

This will generate a graph in `plots`. Figure 2 is an example graph from the `caches` benchmark. Can you answer the following questions with `caches.report.txt`?

- What is the L1 cache size?
- What is the L1 cache latency?
- What is the L2 cache size?
- What is the L2 cache latency?
- What is the DRAM latency?

To answer the following questions, you need to understand the source code (`caches.c`) and rerun simulations with a modified script:

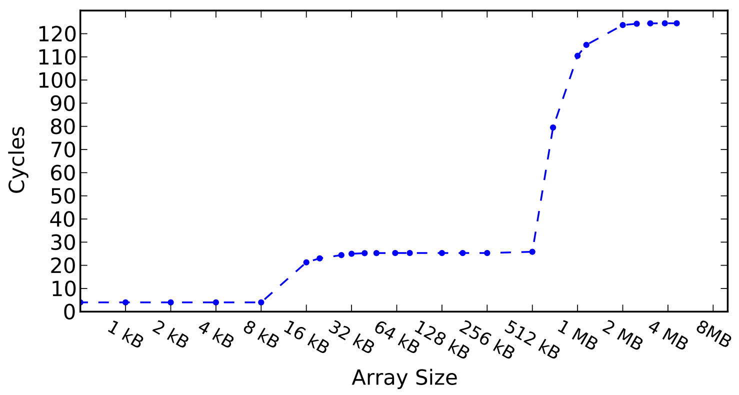

Figure 2: Memory timing validation with `ccbench`

- What is the L1 cache block size?
- What is the L2 cache block size?

Read `docs/CelioCaulin_CS267_finalReport.pdf` for more information.

Finally, can you find any bug in Donggyu's L2 cache timing model? (You will get an extra point if it is a real bug.)

4.2.2 Hard Questions

Do your best to solve hard questions in this section. We do not expect you to solve all the questions but you should at least show your initiative and effort to answer the following questions:

- What is the page size?
- What is the TLB reach?
- What is the miss penalty of TLBs?
- (Extra Points) What is the L1 instruction cache size?

You will need to modify `caches.c` to answer the above questions. Then, compile it in *an instructional machine*:

```
inst$ source ~cs152/sp18/cs152.lab2.bashrc
inst$ cd ccbench/caches
inst$ make ARCH=riscv
```

Next, copy the binary file to the F1 instance. (We assume you have launched an instance.) Make sure you backed up the previous binary.

```
inst$ scp caches centos@<ip address>:~/cs152-lab2/ccbench/
```

Now, you can run the benchmark with the script in the instance.

4.2.3 Submission

Submit a report that explains how you can figure out cache configurations using `ccbench`. Specify what AFI images and what cache parameters you used. We recommend you include graphs that show the answers for each question. For hard question grading, please send `caches.c` to Donggyu (`dgkim@eecs.berkeley.edu`).

5 Feedback

To best of our knowledge, this is the world's first (hopefully not last) lab using AWS F1 instances to explore various memory hierarchies with FPGA-based simulations running real-world applications. Thus, we need your feedback to continue this lab in the future. Please fill out the survey form at `http://tinyurl.com/cs152-sp18-lab2-survey`.

How many hours did the directed portion take you? How many hours did you spend on the open-ended portion? Was this lab a lot of fun or just boring? Did you learn anything interesting? Is there anything you would like to change? Feel free to write as little or as much as you want.

6 Acknowledgments

We thank Amazon for their credit donation for this project. This lab is inspired by the previous set of CS 152 labs written by Henry Cook, Yunsup Lee and Andrew Waterman, which targeted functional simulators such as Simics and Spike.

References

[1] K. Asanović, R. Avizienis, 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," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, apr 2016.

[2] D. Kim, A. Izraelevitz, C. Celio, H. Kim, B. Zimmer, Y. Lee, J. Bachrach, and K. Asanović, "Strober : Fast and Accurate Sample-Based Energy Simulation for Arbitrary RTL," in *ISCA*, 2016.

[3] D. Kim, C. Celio, D. Biancolin, J. Bachrach, and K. Asanović, "Evaluation of RISC-V RTL with FPGA-Accelerated Simulation," in *First Workshop on Computer Architecture Research with RISC-V*, 2017.

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *DAC*, 2012.

[5] A. Waterman, Y. Lee, D. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual: User-level ISA Version 2.1," Tech. Rep. UCB/EECS-2016-118, 2016.

[6] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10," May 2017.

[7] S. Wilton and N. Jouppi, "CACTI: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 677–688, may 1996.

[8] C. Celio, "The ccbench micro-benchmark collection (https://github.com/ucb-bar/ccbench/wiki)," 2010.