ECE4700J Computer Architecture

Summer 2022

Lab #6 Gem5 Simulation of Spectre V1 Attack

Due: 11:59pm Jul. 15th, 2022 (Beijing Time)

Logistics

- This lab is an individual exercise. You may discuss the specification and help one another. The modifications you submit must be your own.
- There will not be a live demo for this assignment.
- All code and reports (if available) MUST be submitted to the assignment of Canvas.
- Internet usage is allowed and encouraged.
- No late submission is allowed for this lab.

Overview

In this lab, you will study how to setup the gem5 framework, and use this simulator to reproduce the Spectre V1 attack.

- Study how to setup the gem5 simulator.
- Study how to run a program using O3CPU in gem5.
- Study what is Spectre Attack, and reproduce it in gem5.
- Be able to visualize the OoO pipeline in gem5.

Assignments

I. Learning Spectre V1 Attack

Meltdown and Spectre exploit critical vulnerabilities in modern processors. These hardware vulnerabilities allow programs to steal data which is currently processed on the computer. While programs are typically not permitted to read data from other programs, a malicious program can exploit Meltdown and Spectre to get hold of secrets stored in the memory of other running programs. This might include your passwords stored in a password manager or browser, your personal photos, emails, instant messages and even business-critical documents.

To study the Spectre V1 Attack, go to the Spectre paper ([1801.01203] Spectre Attacks: Exploiting Speculative Execution (arxiv.org)). Read the abstract, and sections 1-1.1, 2 -2.5, 3 - 4.2. The paper also includes an example C code of Spectre V1 which we

choose to use in this lab. (Appendix A in that paper). Try to understand that code.

II. Setting Up the Gem5 Simulator

Follow the instruction from "First Steps" in gem5: Getting Started with gem5 to get the code of gem5 and compile it. (Note the default target ISA is X86, if you are an Apple M1 user, you may need to change it to ARM for this lab).

III. Simulate Spectre using Gem5

a) Run the Spectre on your own computer

We have provided you with the C code of Spectre V1 (spectre.c). First, you can compile the proof of concept code on your native machine (**note: I'll be using x86 for all of my examples**). Use gcc to compile the Spectre:

```
gcc spectre.c -o spectre
```

Then directly run it on your native machine (./spectre). If you see the output like this:

```
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffdd76c8... Success: 0x54='T' score=2
Reading at malicious_x = 0xffffffffffdd76c9... Success: 0x68='h' score=2
Reading at malicious_x = 0xffffffffffdd76ca... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffffffffdd76cb... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffdd76cc... Success: 0x4D='M' score=2
Reading at malicious_x = 0xffffffffffdd76cd... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdd76ce... Success: 0x67='g' score=2
Reading at malicious_x = 0xffffffffffdd76cf... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffffffffdd76d0... Success: 0x63='c' score=2
Reading at malicious_x = 0xffffffffffdd76d1... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffdd76d2... Success: 0x57='W' score=2
Reading at malicious_x = 0xffffffffffdd76d3... Success: 0x6F='o' score=2
Reading at malicious_x = 0xffffffffffdd76d4... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffffffffdd76d5... Success: 0x64='d' score=2
Reading at malicious_x = 0xffffffffffdd76d6... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffffffffdd76d7... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffdd76d8... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdd76d9... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffffffffdd76da... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffffffffdd76db... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffdd76dc... Success: 0x53='S' score=2
Reading at malicious_x = 0xffffffffffdd76dd... Success: 0x71='q' score=2
Reading at malicious_x = 0xffffffffffdd76de... Success: 0x75='u' score=2
Reading at malicious_x = 0xffffffffffdd76df... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffffffffdd76e0... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdd76e1... Success: 0x6D='m' score=2
Reading at malicious_x = 0xffffffffffdd76e2... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffffffffdd76e3... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffffffffdd76e4... Success: 0x68='h' score=2
Reading at malicious_x = 0xffffffffffdd76e5... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffdd76e6... Success: 0x4F='O' score=2
Reading at malicious_x = 0xffffffffffdd76e7... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffffffffdd76e8... Success: 0x73='s' score=2
```

```
Reading at malicious_x = 0xffffffffffdd76e9... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffffffffdd76ea... Success: 0x66='f' score=2
Reading at malicious_x = 0xffffffffffdd76eb... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffffffffdd76ec... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdd76ed... Success: 0x67='g' score=2
Reading at malicious_x = 0xffffffffffdd76ee... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffffffffdd76ef... Success: 0x2E='.' score=2
```

It means that your native machine is still vulnerable to Spectre, lol. (The Spectre attack is successful)

b) Run the Spectre on gem5 simulator

To find out if gem5's out of order CPU implementation is vulnerable to Spectre, we need to run the code in gem5. The simplest and fastest way to do this is by running in gem5's syscall-emulation (SE) mode. In SE mode we won't be modeling an OS or any user-mode to kernel-mode interaction, but this okay for Spectre since this proof of concept code is all in user-mode.

So, when running something in gem5, the first step is to create a Python runscript since this is the "interface" to gem5. For this example, what we need is a system with one CPU, an L1 cache, and memory. For simplicity, we are going to modify one of the existing script, specifically the **two_level.py** script from the Learning gem5 tutorial (Learning gem5 Tutorial).

In the file <yourgem5directory>/configs/learning_gem5/part1/two_level.py, simply change the CPU from **TimingSimpleCPU()** to **DerivO3CPU (branchPred=LTAGE())**. Here we also set the O3CPU to use the LTAGE branch predictor instead of the default tournament branch predictor.

Then we simply run it:

```
<yourgem5directory>/build/X86/gem5.opt <yourgem5directory>/configs/learning_gem5/part1/two_level.py spectre
```

And, the output that you get should be like the following, just like above when I run the spectre natively.

```
gem5 Simulator System.  https://www.gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 22.0.0.1
gem5 compiled Jul  4 2022 00:55:28
gem5 started Jul  4 2022 01:08:07
gem5 executing on LAPTOP-R8JA7A7I, pid 23123
command line: gem5-new/gem5/build/X86/gem5.opt gem5-
new/gem5/configs/learning_gem5/part1/two_level.py spectre

Global frequency set at 1000000000000 ticks per second
warn: failed to generate dot output from m5out/config.dot
build/X86/mem/dram_interface.cc:690: warn: DRAM device capacity (8192 Mbytes)
does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7000
```

```
        Beginning simulation!
        build/X86/sim/simulate.cc:194: info: Entering event queue @ 0.   Starting
simulation...
        build/X86/sim/mem_state.cc:443: info: Increasing stack size by one page.
        build/X86/sim/syscall_emul.cc:74: warn: ignoring syscall mprotect(...)
        build/X86/sim/syscall_emul.cc:74: warn: ignoring syscall mprotect(...)
        build/X86/sim/syscall_emul.cc:74: warn: ignoring syscall mprotect(...)
        Reading 40 bytes:
        Reading at malicious_x = 0xffffffffffffdfc8... Success: 0x54='T' score=2
        Reading at malicious_x = 0xffffffffffffdfc9... Success: 0x68='h' score=2
        Reading at malicious_x = 0xffffffffffffdfca... Success: 0x65='e' score=2
        Reading at malicious_x = 0xffffffffffffdfcb... Success: 0x20=' ' score=2
        Reading at malicious_x = 0xffffffffffffdfcc... Success: 0x4D='M' score=2
        Reading at malicious_x = 0xffffffffffffdfcd... Success: 0x61='a' score=2
        Reading at malicious_x = 0xffffffffffffdfce... Success: 0x67='g' score=2
        Reading at malicious_x = 0xffffffffffffdfcf... Success: 0x69='i' score=2
        Reading at malicious_x = 0xffffffffffffdfd0... Success: 0x63='c' score=2
        Reading at malicious_x = 0xffffffffffffdfd1... Success: 0x20=' ' score=2
        Reading at malicious_x = 0xffffffffffffdfd2... Success: 0x57='W' score=2
        Reading at malicious_x = 0xffffffffffffdfd3... Success: 0x6F='o' score=2
        Reading at malicious_x = 0xffffffffffffdfd4... Success: 0x72='r' score=2
        Reading at malicious_x = 0xffffffffffffdfd5... Success: 0x64='d' score=2
        Reading at malicious_x = 0xffffffffffffdfd6... Success: 0x73='s' score=2
        Reading at malicious_x = 0xffffffffffffdfd7... Success: 0x20=' ' score=2
        Reading at malicious_x = 0xffffffffffffdfd8... Success: 0x61='a' score=2
        Reading at malicious_x = 0xffffffffffffdfd9... Success: 0x72='r' score=2
        Reading at malicious_x = 0xffffffffffffdfda... Success: 0x65='e' score=2
        Reading at malicious_x = 0xffffffffffffdfdb... Success: 0x20=' ' score=2
        Reading at malicious_x = 0xffffffffffffdfdc... Success: 0x53='S' score=2
        Reading at malicious_x = 0xffffffffffffdfdd... Success: 0x71='q' score=2
        Reading at malicious_x = 0xffffffffffffdfde... Success: 0x75='u' score=2
        Reading at malicious_x = 0xffffffffffffdfdf... Success: 0x65='e' score=2
        Reading at malicious_x = 0xffffffffffffdfe0... Success: 0x61='a' score=2
        Reading at malicious_x = 0xffffffffffffdfe1... Success: 0x6D='m' score=2
        Reading at malicious_x = 0xffffffffffffdfe2... Success: 0x69='i' score=2
        Reading at malicious_x = 0xffffffffffffdfe3... Success: 0x73='s' score=2
        Reading at malicious_x = 0xffffffffffffdfe4... Success: 0x68='h' score=2
        Reading at malicious_x = 0xffffffffffffdfe5... Success: 0x20=' ' score=2
        Reading at malicious_x = 0xffffffffffffdfe6... Success: 0x4F='O' score=2
        Reading at malicious_x = 0xffffffffffffdfe7... Success: 0x73='s' score=2
        Reading at malicious_x = 0xffffffffffffdfe8... Success: 0x73='s' score=2
        Reading at malicious_x = 0xffffffffffffdfe9... Success: 0x69='i' score=2
        Reading at malicious_x = 0xffffffffffffdfea... Success: 0x66='f' score=2
        Reading at malicious_x = 0xffffffffffffdfeb... Success: 0x72='r' score=2
        Reading at malicious_x = 0xffffffffffffdfec... Success: 0x61='a' score=2
        Reading at malicious_x = 0xffffffffffffdfed... Success: 0x67='g' score=2
        Reading at malicious_x = 0xffffffffffffdfee... Success: 0x65='e' score=2
        Reading at malicious_x = 0xffffffffffffdfef... Success: 0x2E='.' score=2
        Exiting @ tick 102605083000 because exiting with last active thread context
```

c) Visualize the Pipeline on gem5 simulator

i. Generating the Pipeline View (Mandatory)

To generate pipeline visualizations, we first need to generate a trace file of all of the instructions executed by the out of order CPU. To create this trace, we can use the O3PipeView debug flag.

The gem5 also provides tutorial on how to use the pipeline visualizer: gem5: Visualization.

Now, the trace for the O3 CPU can be very large up to many GBs. When creating this trace, you need to be careful to create the smallest trace possible. Also, it's important to dump the trace to a file and not to stdout, which is the default when using debug flags. You can redirect the trace to a file by using the --debug-file option to gem5.

To create the trace file, use the following methodology:

1. Start running spectre in gem5, then hit ctrl-c after the "reading" of first couple of letters. At this point, write down the tick which gem5 exited.
2. Run gem5 with the debug flag O3PipeView enabled.
3. Watch the output and kill gem5 with ctrl-c after two more "reading" of letters appeared than in step 1.

To generate the trace, I ran the following command. Note: you may have a different value for when to start the debugging trace. Also note: when producing the trace gem5 will run much slower.

```
<gem5dir>/build/X86/gem5.opt --debug-flags=O3PipeView --debug-file=pipeview.txt -
-debug-start=1205889000 <gem5dir>/configs/learning_gem5/part1/two_level.py
spectre
```

The tracefile (pipeview.txt) will be more than 600 MB for catching just two letters (T, h) in the output. And producing it is slow (some minutes).

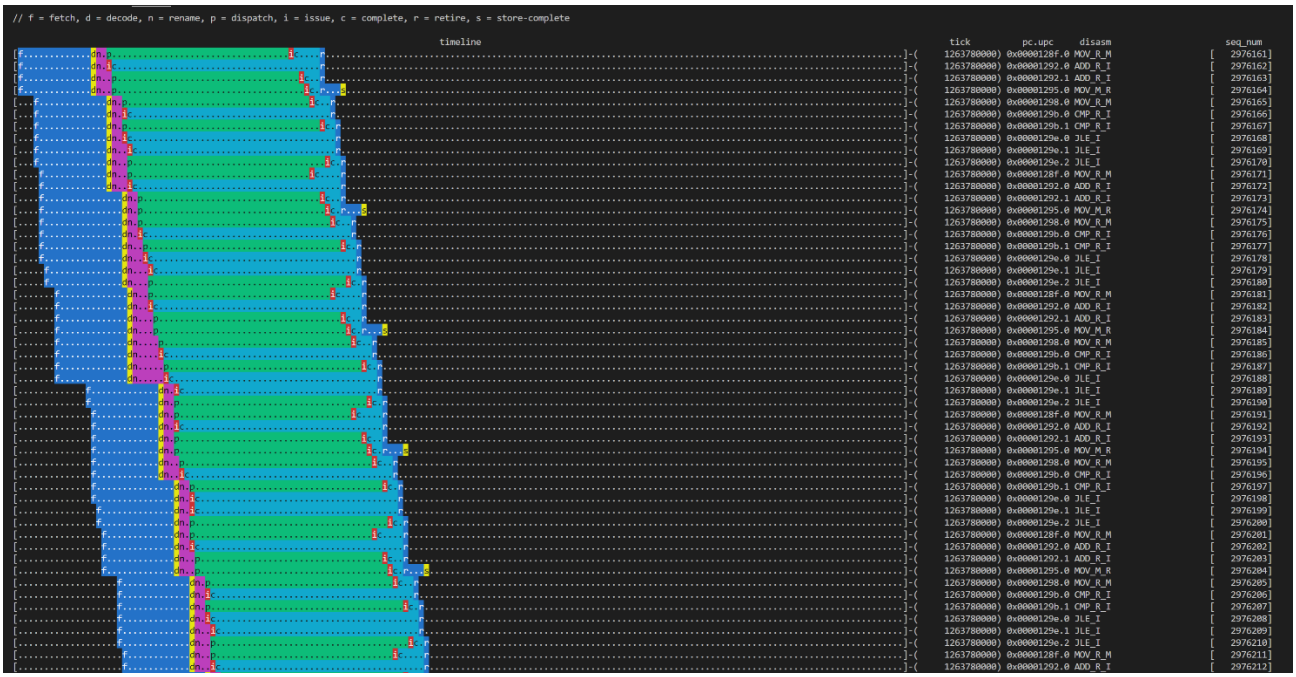
Now, we can process this file to generate the visualization with a script: <gem5dir>/util/o3-pipeview.py. This script requires the path to the file that contains the output generated with the O3PipeView debug flag. Above, we put the output into the file pipeview.txt, and this file was created in the default output directory of gem5 (m5out/).

```
<gem5dir>/util/o3-pipeview.py --store_completions m5out/pipeview.txt --color -w
150
```

In the above command, we want to see when the stores completed (--store_completions) and specified to use color (--color) in the output and use a width of 150 characters (-w 150). Processing a large file like this one of 600 MB may take a few minutes. The output will be in a file called o3-pipeview.out in the current working directory.

You can view this file with `less -r o3-pipeview.out`. You may want to use the -S option with less if your terminal is less than 150 characters wide (or whatever width value you used). Below is a screenshot of the top of my trace. **You are required to also**

take a screenshot of the top of your trace (no matter whether it can show the Spectre attack).



The above image details how to interpret the output from the pipeline viewer. Each "." or "=" represents one cycle of time, which moves from left to right. The "tick" column shows the tick of the leftmost "." Or "=". "=" is used to mark the instructions that were later squashed. The address of the instruction (and the micro-op number) as well as the disassembly is also shown. The sequence number can be ignored as it is always monotonically increasing and is the total order of every dynamic instruction. Finally, each stage of the O3 pipeline is shown with a different letter and color.

ii. Visualize the Spectre Attack (Optional)

Then you can try to find the part that can shows the Spectre attack process. However, doing this is difficult, we don't require you to do this. However, if you can find the exact part, you will be given **20% extra points for your lab6.**

Some hints on this:

1. You can use `gdb` or some other tools to first disassemble your "spectre" program, and find the virtual pc addresses and x86 instructions in the function "victim_function". An example:

```
haoyoung@LAPTOP-R8JA7A7I:/mnt/d/Research/spectre-btb$ gdb spectre
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
```

```
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from spectre...
(No debugging symbols found in spectre)
(gdb) b victim_function
Breakpoint 1 at 0x1815
(gdb) r
Starting program: /mnt/d/Research/spectre-btb/spectre
Reading 40 bytes:

Breakpoint 1, 0x0000555555555815 in victim_function ()
(gdb) disassemble
Dump of assembler code for function victim_function:
=> 0x0000555555555815 <+0>:     endbr64
   0x0000555555555819 <+4>:     push   %rbp
   0x000055555555581a <+5>:     mov    %rsp,%rbp
   0x000055555555581d <+8>:     mov    %rdi,-0x8(%rbp)
   0x0000555555555821 <+12>:    mov    0x27f9(%rip),%eax        # 0x555555558020 <array1_size>
   0x0000555555555827 <+18>:    mov    %eax,%eax
   0x0000555555555829 <+20>:    cmp    %rax,-0x8(%rbp)
   0x000055555555582d <+24>:    jae    0x555555555862 <victim_function+77>
   0x000055555555582f <+26>:    lea    0x280a(%rip),%rdx        # 0x555555558040 <array1>
   0x0000555555555836 <+33>:    mov    -0x8(%rbp),%rax
   0x000055555555583a <+37>:    add    %rdx,%rax
   0x000055555555583d <+40>:    movzbl (%rax),%eax
   0x0000555555555840 <+43>:    movzbl %al,%eax
   0x0000555555555843 <+46>:    shl    $0x9,%eax
   0x0000555555555846 <+49>:    cltq
```

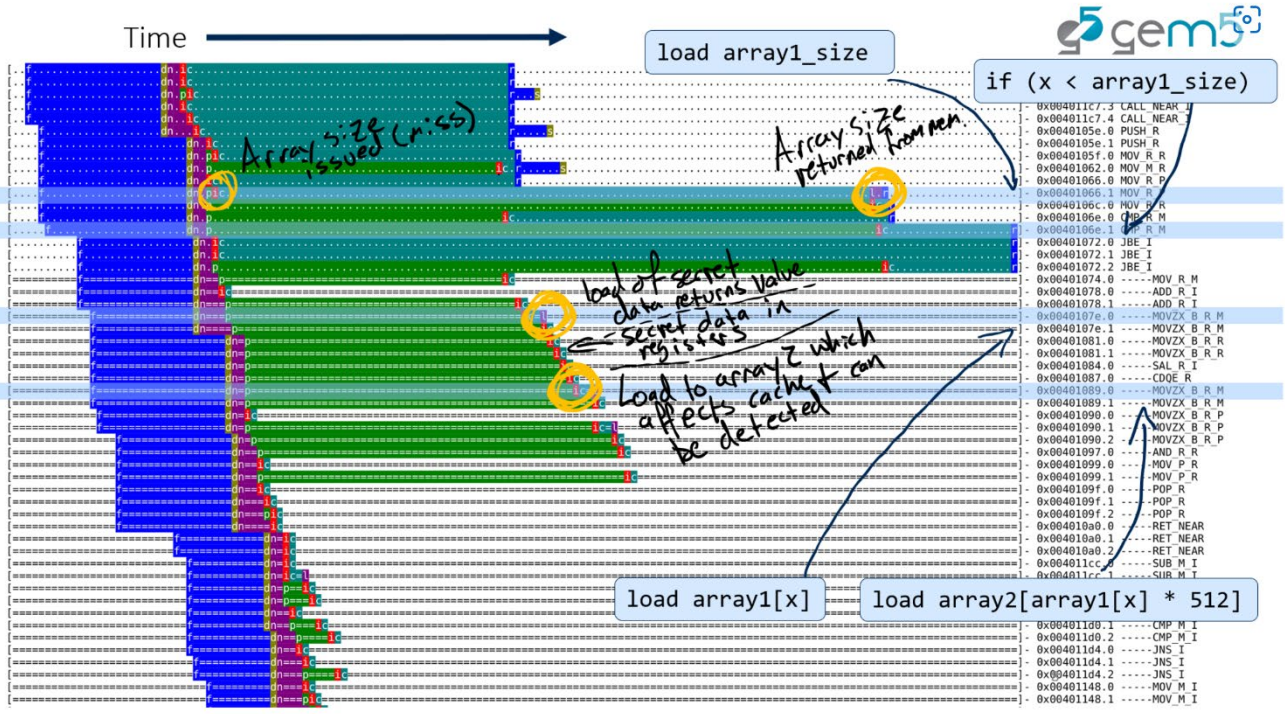
```
   0x0000555555555848 <+51>:    lea    0x2d31(%rip),%rdx        # 0x555555558580 <array2>
   0x000055555555584f <+58>:    movzbl (%rax,%rdx,1),%edx
   0x0000555555555853 <+62>:    movzbl 0x28c6(%rip),%eax        # 0x555555558120 <temp>
   0x000055555555585a <+69>:    and    %edx,%eax
   0x000055555555585c <+71>:    mov    %al,0x28be(%rip)        # 0x555555558120 <temp>
   0x0000555555555862 <+77>:    nop
   0x0000555555555863 <+78>:    pop    %rbp
   0x0000555555555864 <+79>:    retq
End of assembler dump.
```

This shows that the virtual address for "victim_function" is at 0x1815 – 0x1864 (in my computer, may not the same as yours). (It's added to 0x0000555555554000 because of your OS's paging)

2. In the O3-pipeview.out, search the address you just get ("0x00001815" in my computer), to find every time the pipeline execute the "victim_function" part. And then look into the pipeline view and find them one by one, and finally find the Spectre attack V1 (you should first understand what should it be like).

An illustration of the pipeline view of Spectre V1 is shown below (source: [Visualizing Spectre with gem5 (lowepower.com)]):

(# NOTE: the movzbl instruction is MOVZX_B_R_M in gem5.)



If you successfully find the pipeline view of Spectre V1, make a screenshot and submit it. Also notice your TA via Feishu.

d) Submission

You need to submit the screenshot of your pipeline view of any part of the program (named pipeview.png or .jpg), and the "config.json" and "stats.txt" generated in your m5out/ folder.

If you finish the optional part, submit the screenshot of the pipeline view which can show the Spectre Attack, and the "config.json" and "stats.txt" generated in your m5out/ folder.

References:

[1] Kocher, Paul, et al. "Spectre attacks: Exploiting speculative execution." *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[2] Nathan Binkert, et al. 2011. *The gem5 simulator*. SIGARCH Comput. Archit. News 39, 2 (May 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[3] Jason Lowe-Power. "Visualizing Spectre with gem5". http://www.lowepower.com/jason/visualizing-spectre-with-gem5.html

Acknowledgement:

Prof. Jason Lowe-Power, Assistant Professor of Computer Science at UC, Davis.

Deliverables:

- Individual Deliverables:
  - Submit required files of the Assignment.

Grading Policy

Canvas Files Submission and Correctness – 100%