Instruction-Level Power Consumption Simulator for Modeling Simple Timing and Power Side Channels in a 32-bit RISC-V Micro-Processor

by

Gloria (Yu Liang) Fang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2021

Author .....
Department of Electrical Engineering and Computer Science
December 25, 2020

Certified by .....
Anantha Chandrakasan
Vannevar Bush Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Instruction-Level Power Consumption Simulator for Modeling Simple Timing and Power Side Channels in a 32-bit RISC-V Micro-Processor

by

Gloria (Yu Liang) Fang

Submitted to the Department of Electrical Engineering and Computer Science
on December 25, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We create a Python based RISC-V simulator that is capable of simulating any assembly code written in RISC-V, and even perform simple power analysis of RISC-V designs. The power consumption of non-privileged RISC-V RV32IM instructions are measured experimentally, forming the basis for our simulator. These instructions include memory loads and stores, PC jumps and branches, as well as arithmetic instructions with register values. The object-oriented simulator also supports stepping and debugging. In the context of designing software for hardware use, the simulator helps assess vulnerability to side channel attacks by accepting input power consumption values. The power consumption graph of any disassembled RISC-V code can be obtained if the power consumption of each instruction is given as an input; then, from the output power consumption waveforms, we can assess how vulnerable a system is to side channel attacks. Because the power values can be customized based on what's experimentally measured, this means that our simulator can be applied to any disassembled code and to any system as long as the input power consumption of each instruction is supplied. Finally, we demonstrate an example application of the simulator on a pseudorandom function for simple side channel power analysis.

Thesis Supervisor: Anantha Chandrakasan
Title: Vannevar Bush Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Professor Chandrakasan for all the support, guidance, and feedback. I feel extremely lucky to contribute to this lab, and grateful for the opportunity to research under one of the most encouraging and accomplished professors in the field.

I would also like to thank Utsav for all his guidance and help. Utsav is the most amazing and knowledgeable mentor, and I truly respect him and am very grateful for his help through all the challenges in research. Utsav has taught me so much over the past two semesters of MEng, as well as in previous semesters of UROP. I really look up to Utsav as a mentor and as a great friend.

I want to thank the MTL labs, and I want to thank the Department of EECS for their continuing support over the past five years.

I also want to thank all the professors in every class I've taken here at MIT. I appreciate all the work professors put into teaching and helping me learn, and I also want to thank all the TA's for their time and help.

Finally, I'd like to thank all my friends and family for their endless support. Their positivity and friendship get me through any hurdle, and have made the past few years at MIT so unforgettable and valuable.

Contents

1 Introduction **15**

1.1 Motivation 15

1.2 Background 17

1.3 Objectives 18

1.4 Key Steps 21

2 Setup and Methods **25**

2.1 RISC-V Instruction Set Architecture Summary 25

2.1.1 Register Type Instructions (REG) 25

2.1.2 Immediate Type Instructions (IMM) 26

2.1.3 Branch Instructions (BRANCH) 26

2.1.4 Load Instructions (LOAD) 26

2.1.5 Store Instructions (STORE) 26

2.1.6 Jump Instructions (JUMP) 26

2.1.7 PC Instructions 27

2.1.8 Other Instructions 27

2.2 Overall Setup and Control Parameters, and Scaling to Other Environments 27

2.3 Hardware Setup 28

2.4 Coding Setup and Justification 29

2.4.1 Repeating Measurements 31

2.5 Arithmetic and Logic Instructions 32

2.5.1 REG 33

2.5.2 IMM 34

2.5.3 SHAMT . . . 35
2.6 Conditional Instructions . . . 35
2.7 Memory Instructions . . . 37
2.7.1 Alignment . . . 37
2.7.2 Load Setup . . . 38
2.7.3 Store Setup . . . 38
2.8 Further Setup Improvements and Modifications to Handle Noise . . . 39
2.8.1 Assessing the Effect of Noise . . . 41
2.9 Final Average Current Measurements . . . 43

3 Design of Instruction-Level Power Simulator 45
3.0.1 Inputs and Initial Values . . . 46
3.0.2 Outputs . . . 46
3.0.3 Operation . . . 46
3.0.4 Register Reads and Writes . . . 48
3.0.5 End of Execution . . . 48
3.0.6 Additional Options . . . 49
3.0.7 Memory Class . . . 52
3.1 Other Pre-Processing and Container Scripts . . . 53
3.1.1 Memory Pre-processing . . . 53
3.1.2 RISC-V Code Pre-processing . . . 54
3.1.3 Command line container file . . . 54

4 Evaluation and Application of Simulator 57
4.1 Comparison of Simulated and Experimental Results . . . 57
4.2 Evaluation of Simulator . . . 58
4.2.1 Power Analysis using Simulator . . . 60
4.3 Evaluation of Simulator for Simple Power Analysis (SPA) Side-Channel 64
4.3.1 Naive Discrete Gaussian Sampler . . . 65
4.3.2 Modified Discrete Gaussian Sampler . . . 67
4.4 Limitations . . . 69

5 Conclusion and Future Work **73**

A Code for Data Collection **75**

A.1 Example main.c File 75

A.2 Python Code to Generate main.c files 78

B Code for RISC-V Simulator **87**

B.1 Preprocessing Assembly Code 87

B.2 Preprocessing Memory Inputs 88

B.3 Commandline Wrapper for RISC-V Simulator 89

B.4 Register Name File (`reg_name.txt`) 92

B.5 Python based RISC-V Simulator 94

B.6 Sample RISC-V Code: Pseudorandom Number Generator 132

B.7 Example Neural Network in C for Simulator and Experimental Measurement Comparison 136

List of Figures

2-1 Figure 1: Three example instruction formats that are used within arithmetic instructions. 33

2-2 Current measurements with 5 minute wait times between successive measurements. 40

2-3 Current measurements with 10 minute wait time between successive measurements. 41

4-1 Waveform of current for the first few cycles. 60

4-2 Waveform of current. This sample reflects the majority of cycles. . . . 61

4-3 Waveform of current in the final cycles. 61

4-4 Plot of the number of cycles between each call of `random` versus the value of xlen. 63

4-5 Excerpt of assembly code corresponding to `random` function. 64

4-6 C code for the naive discrete Gaussian sampler. 65

4-7 Histogram of output values for naive approach of discrete Gaussian sampler. 66

4-8 Histogram of cycle count for naive discrete Gaussian sampler. 66

4-9 Histogram of average current values for naive Gaussian sampler. . . . 68

4-10 C code for the modified discrete Gaussian sampler. 68

4-11 Output value distribution of modified discrete Gaussian sampler. . . . 69

4-12 Histogram of average current for the modified Gaussian sampler. . . . 70

4-13 Histogram of cycle count for modified Gaussian sampler. 70

List of Tables

1.1 List of all supported instruction types and instructions. 19

1.2 Register names, alternative names, and common usage. Adapted from `https://inst.eecs.berkeley.edu//čs61c/fa17/img/riscvcard.pdf` 20

2.1 RISC-V instruction type groupings. 32

2.2 Current measurements for the multiply (mul) instruction after successive wait times. 40

2.3 Percent difference of average current for selected arithmetic instructions, with 5 minute and 10 minute wait times. (Percent difference values are relative to 10 min average current). 42

2.4 Measuring the extent of noise for selected arithmetic instructions. . . 43

2.5 Average current for each instruction. 44

4.1 Experimental and simulated results for average current and timing. . 58

4.2 The 10 other anomalies (in addition to the start and end) where the waveform pattern is broken. Their timesteps (cycles) correspond to 16207, 33987, 54967, 80747, 112927, 153107, 202887, 263867, 337647, 425827. 62

4.3 Cycle and current values for each class of outputs, using naive discrete Gaussian sampler. 67

4.4 Cycle count and average current for all outputs, using modified Gaussian sampler. 69

Chapter 1

Introduction

1.1 Motivation

The motivation behind creating a Python-based, instruction-level RISC-V power consumption simulator starts from the difficulty of designing efficient software for hardware. Chip design is a detailed process, and one important step is designing software for the corresponding hardware. The software needs to work well in accordance with the chip. In particular, some requirements include chip security and energy efficiency. For instance, the software design affects the power consumption of the hardware.

If the software isn't designed with the hardware in mind, problems could arise. For example, an attacker who observes power consumption waveforms may be able to infer details about the software execution [15, 10]. This example of a side channel attack reveals the importance of considering the hardware while designing corresponding software.

This, however, is difficult in practice. Software designers don't always have the access, or ability, to work with hardware equipment. Such equipment includes setups for measuring power consumption, or analyzing physical electromagnetic emanations. To measure power consumption, for instance, one would need an oscilloscope and a power source, and be able to connect a differential amplifier. Although some setups may be small and portable, they are rarely used by software developers, and thus not

always available.

Our solution is to create a Python simulator that can model the expected power consumption output, when given the software input code. Although there are other simulators for ARM processors such as JouleTrack [18, 6], to our knowledge this will be the first RISC-V [20] instruction-level power consumption simulator.

The simulator will be based on experimental data, rather than building up from pure theoretical analysis. This is because a thorough theoretical analysis - for instance, at a lower level e.g. circuit connections like netlists - will be computationally expensive in order to adapt to any input. The simulation will be at the assembly instruction level, which provides enough detail while also still being computationally practical. For instance, we profile power consumptions of various assembly instructions. When given software input, we can piece together the corresponding assembly code as well as applying further analysis, to determine the power consumption of the entire code. Note the simulator does not model inter-instruction dependencies, a limitation that can be expanded on in future work.

The benefits of such a simulator are plenty. Firstly, it will eliminate the need for a lab setup (e.g. hardware, with oscilloscope) in order to perform a quick instruction-level simulation. Instead, simply the input code is needed to generate a resulting power consumption waveform.

Secondly, the power consumption from the simulator is quicker and deterministic, unlike the data from lab setups which may vary depending on the specific equipment and environment. Deterministic output is important because it allows users to compare outputs for different code, without worrying about noise or human error. In turn, such comparisons will help software designers make choices when deciding on factors like energy efficiency.

Finally, being able to quickly generate the power consumption output could also help make general predictions for side channel attacks. Side channel attacks often leak in the form of EM radiation or power consumption waveforms [8, 9]. Fortunately, while collecting power waveforms for an instruction set to create the simulator, we may also get a general guideline on which instructions leak the most data. Once we

learn about how instructions leak data, this information will impact security related decisions made by software designers.

With these benefits, a simulator will help facilitate the software designing process, by eliminating the need to separately gather hardware data.

1.2 Background

Various power consumption simulators exist for the ARM processor [5, 13], but not for RISC-V processors. There are existing instruction simulators [14, 4], but they do not simulate power consumption. However, existing ARM instruction simulators provide some background for various approaches [18, 11, 17].

Simulators exist for many purposes. Some simulators primarily serve for data modelling, observing behavior, or tracing memory. Architectural level simulators simulate processor parts, making them useful for hardware logic designers. Direct execution simulators will recompile and replace parts of the code so that it will run locally. Instruction set simulators will model the result of executing each instruction [17]. Other simulators will follow the process of data being stored in registers, so that a programmer can observe each step for debugging.

Since we're looking for the power consumption of software programs, we're mostly concerned with instruction set simulators. When it comes to power consumption simulators, existing ARM instruction set simulators come in three types: black box, white box, and grey box. Black box simulators do not require knowledge of the implementation. Instead, they use statistical analysis to estimate the Gaussian distribution parameters, or linear regression to fit to unknown data (such as modelling inside vector spaces) [16]. Although these models are computationally fast, such models are too general for our purposes as they do not account for enough detail.

White box simulators, in contrast, model detailed circuit connections. They may specify the connections between hardware components (e.g. transistors), making them costly to compute if we were to model an entire board. Furthermore, this level of

detail does not account for noise, such as when communication channels interfere with each other in crosstalk.

Grey box simulators are intermediate between black box and white box simulators. Though many grey box simulators use hamming weight assumptions [19] or averaged total cost, McCann et. al. preserve the effect of instruction sequences by using a different method by combining linear-regression approaches, as well as considering how consecutive instructions interact [11].

Since we desire both the computational speed of black box simulators, as well as a certain level of detail, we will be creating an instruction-level simulator. We select the most relevant RISC-V instructions among the entire instruction set, and profile their power consumptions. We then aggregate their profiles in a way that will simulate input software code.

1.3 Objectives

Ultimately, our simulator aims to simulate the instruction level power consumption of an input RISC-V code, allowing for simple power analysis. To do so, we need to achieve the following objectives.

First, we measure the power consumption of RISC-V instructions included in the RV32IM instruction set. The instructions supported by our simulator include arithmetic and logic instructions, memory instructions (including loads and stores), conditional branches or jumps, and program counter instructions. This covers all non-privileged instructions that don't modify system-configuration registers.

Secondly, since we are supporting a multitude of instructions which interact with the program counter, memory addresses, and registers, we need to support each of these components. We need to track the program counter in order to support conditional statements - including jumps and branches.

We also need to simulate memory in order to support loads and stores, and be able to output the final memory values. In some cases of disassembled C code, memory

Type of Instruction	Instructions
Loads	lb, lh, lw, lbu, lhu
Stores	sb, sh, sw
Shifts	sll, slli, srl, srli, sra, srai
Arithmetic	add, addi, sub, lui, auipc
Logical	xor, xori, or, ori, and, andi
Compare	slt, slti, sltu, sltiu
Branches	beq, bne, blt, bge, bltu, bgeu
Jump and Link	jal, jalr
Multiply	mul, mulh, mulhsu, mulhu
Divide	div, divu
Remainder	rem, remu

Table 1.1: List of all supported instruction types and instructions.

values are initialized before the assembly code is run; therefore, we also need to support user-input memory initialization. The main challenge of supporting memory is that there are many addresses, however usually accesses only happen at 2, 4, or 8 bit offsets. Additionally, since many memory addresses may be empty, this needs to be designed carefully to ensure that we do not end up using more space in our Python simulator than needed.

We also need to support registers. There are 32 registers, and register values are each 32 bits long. The registers are named `x0` through `x31`, and many registers have conventional use cases. For example, register `x0`, also known as `zero`, is fixed to hold the constant value 0; registers `a0` through `a7` are alternate names for `x10` through `17` and are typically used for function arguments and return values. Since there are fewer registers than memory addresses, supporting them is less challenging. However, we need to support alternative naming, since many registers have equivalent names in RISC-V.

Another challenge is deciding on the format of data stored. In our Python-based simulator, we need to consider the format of registers because they may require a different amount of overhead and space - whether it is in bit-string format, which requires more overhead to convert; or hexadecimal string format; or integer format.

Register Name	Alternata Name(s)	Common Usage
x0	zero	Constant zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporary registers
x8	s0, fp	Saved register, frame pointer
x9	s1	Saved register
x10-x17	a0-a7	Function arguments and return values
x18-x27	s20-s11	Saved registers
x28-x31	t3-t6	Temporary registers

Table 1.2: Register names, alternative names, and common usage. Adapted from `https://inst.eecs.berkeley.edu//čs61c/fa17/img/riscvcard.pdf` .

The overhead of our data is important because as we apply our simulator to larger programs, we may easily run over millions of steps, magnifying the time it takes to perform computations. This is further complicated by the fact that some numbers are treated as two's complement, while others may be treated as unsigned - for example, instructions like `sltiu` specifically require sign extending and reinterpreting values as unsigned. Thus, extra support and casework is needed to support these unique number formats.

And while storing the data format internally is a challenge, the input RISC-V code can also provide ambiguous data formats. Depending on the disassembly and conversion process, the assembly code could use either hexadecimal or decimal constants. This means we need to support both, converting them to a consistent format internally. The input assembly code may also use either relative or absolute addressing for jumps and branches, however the RISC-V instruction manual describes relative addressing; as a result, we need to provide pre-processing to maintain consistency of relative-addressing.

Since we will inevitably run into errors, we also prefer to support instruction-level debugging. This not only helps debug any handwritten assembly code (though

use cases are small as most assembly code probably comes from the compiled and disassembled C code), it also helps debug errors within the simulator. Thus we need to support debug statements that are specific enough to inform the type of error, and the location (line number) of errors. In line with debug statements, we also aim to support an option for verbose and non-verbose print statements, which print the status after running each instruction. For ease of use, we also need to support single-stepping the simulator, as well as running for a given number of steps, and also running until completion. This helps trace the sequence of instructions through the assembly code.

Once the simulator supports all the above, it is capable of stepping through assembly code. Finally, since our goal is to ultimately simulate power consumption waveforms, we need to support plotting, and allow the user to input power values for each instruction. This means we need to accumulate a list of power values as we step through every instruction. Once all these objectives are achieved, we will have a working Python-based RISC-V power consumption simulator.

1.4 Key Steps

In order to achieve these objectives, and demonstrate the functionality of the simulator, there are several crucial steps.

The first step is designing a hardware setup. This setup needs to be designed to amplify current changes, so that we can visualize the power consumption of a test chip on an oscilloscope. For our setup, we also need to verify its resistance to noise. For example, we need to confirm that noise due to setup is not significant enough to affect the validity of our measurements.

Additionally, we need to design a strategy for collecting measurements. Since the chip tends to heat up over time, we need to factor this in and add enough wait time between measurements. Once the setup design is available, can start gathering power data for all supported RISC-V instructions.

One challenge is designing the structure of the C code for gathering repeated measurements. Because the structure of a for-loop destroys our measurements (we will

describe this in further detail in future chapters), we need to hard code repetitions to a certain extent. However, the test chip is limited in memory, thus we need to ensure we have enough repetitions for measurement, but not too many that we cannot store the code on the chip. Thus we need to design the C code carefully for measurements.

Once we generalize our C code, we need to repeat this for every supported instruction. Since this is difficult to repeat manually, we will use Python to generate the C code based on instruction type. To do so, we need to classify our instruction types. This classification is used for data measurement, and has a different purpose than the classification used for the simulator (although the categories tend to be similar).

The workflow for measuring a specific instruction is therefore using C code, converting it to RISC-V assembly, loading it to the chip, and measuring the results on the oscilloscope.

After analyzing and gathering data, we apply them to the Python RISC-V simulator. The next objective (not necessarily chronological) is to achieve a fully functional simulator, which supports all non-privileged instructions, and plotting power inputs.

Once these milestones are achieved, we evaluate and demonstrate the use of the simulator. We evaluate correctness by creating an example function in C. We compile the C code using gcc to determine the final memory values; and we compare these values to the equivalent RISC-V assembly code fed through the simulator. We show that it works correctly using a pseudorandom number generator that even has 32-bit overflow in some arithmetic steps.

After verifying correctness, we demonstrate using the simulator's generated power consumption waveform for simple power analysis. We show that the waveform exposes information about the input or output values in a function called in the assembly code. We choose a discrete Gaussian sampler to illustrate this application, as the security of these samplers are crucial for their use case in lattice-based cryptography [7, 3, 12].

By achieving each of these key steps, we have collected experimental power consumption data, and created a Python RISC-V simulator that uses this data to generate a power consumption waveform. With such a simulator, it is now possible to perform

instruction-level power consumption analysis of any future RISC-V assembly code, without needing to physically prepare an entire hardware setup.

Chapter 2

Setup and Methods

In this chapter, we describe methods for measuring the power consumption for each supported instruction. These power values will be used as inputs for the Python simulator described in later chapters.

2.1 RISC-V Instruction Set Architecture Summary

We support all non-privileged instructions from the RISC-V instruction set architecture (ISA). A brief overview is included below; further detail can be found in the RISC-V ISA specifications [1].

2.1.1 Register Type Instructions (REG)

The format of these instructions is "inst rd, rs1, rs2". An operation is done on the values held in registers rs1 and rs2, and then stored in rd. This category includes arithmetic and logic instructions, shifts, multiply, divide, and remainder instructions. We exclude immediate type, auipc, and lui, which we analyze separately.

[1] https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

2.1.2 Immediate Type Instructions (IMM)

These instructions take the format "inst rd, rs1, imm" where the immediate value "imm" is a constant. Most immediate instructions are identical to their REG equivalents except they use the constant rather than the value in a register. For simplicity we'll abbreviate this category of instructions as "IMM".

2.1.3 Branch Instructions (BRANCH)

Branch instructions execute a program counter branch if a given condition is true. These include six branch instructions, some of which are unsigned.

2.1.4 Load Instructions (LOAD)

Load instructions retrieve values from memory addresses. The format is "inst rd, offset(rs1)". Depending on the specific load instruction, a different number of bits are read from the memory.

2.1.5 Store Instructions (STORE)

Store instructions come in the format "inst rs1, offset(rd)". The opposite of load instructions, store instructions capture the value saved in register rs1, and save it to the address in register rd with offset.

2.1.6 Jump Instructions (JUMP)

There are two types of jumps: JAL and JALR[2]. Both kinds of jumps are often used when function calls are disassembled, as the return address of a function would be saved.

[2]In our simulator, we eventually handle these in separate functions, but we group them for discussion.

2.1.7 PC Instructions

Finally, AUIPC and LUI are often used to modify the PC. We group them separately for analysis. AUIPC performs a relative PC increment, and this value is saved in register rd. LUI can be used to perform an absolute address change - rd saves the value of the constant left shifted by 12 bits.

2.1.8 Other Instructions

The RISC-V instruction set also includes privileged instructions[21]. Privileged instructions are only executed by the operating system in certain (e.g. supervisor) modes. Since these instructions are very specific and require an operating system, and since we are considering general input software code, we will not consider them in this study.

There are also several C standard instructions for compressed instructions[3], which are compressed 16 bit instructions convenient for commonly used 32-bit instructions (for example, if using the 8 most common registers, or if the values are small). Since many of these are used in specific cases, our analysis of the base instructions will be a sufficient guideline for general RISC-V assembly code. We also do not need to cover alternative instructions which are already pseudoinstructions, as they can simply be disassembled into existing instructions.

2.2 Overall Setup and Control Parameters, and Scaling to Other Environments

In order to measure power consumption, we first set up a differential amplifier and measure the incoming current. This setup is consistent among all our instruction types. The setup uses a V_{DD} value of 1.1 V. Thus the power will be the average current times

[3]Chapter 12 of RISC-V manual, https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

the voltage.

$$P = i_{avg} V_{DD}$$

For each instruction in the RISC-V ISA, we take the average current measurement for i_{avg}. In general, the setup involves repeating an instruction thousands of times. However, depending on the instruction type, there may be additional steps in the setup, which are described in the following sections.

Since power consumption is also proportional to frequency, our measurements can be extended to other processor frequencies by simply multiplying with the ratio.

$$P = CV^2 f$$

Our setup is at $f = 40MHz$. Our results can quickly generalize to any different setup at the same voltage, using the following ratios:

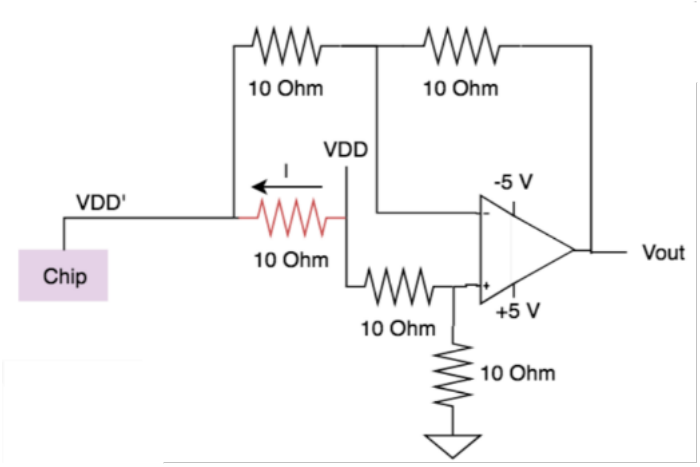
$$P_2 = \frac{P_1 f_2}{f_1}$$

Similarly, we can use the squared ratio of voltages to generalize to any setup with a different voltage. Note that this is an approximate model, as memory does not scale with V^2.

2.3 Hardware Setup

The hardware setup is shown below. The test chip used for measuring power consumption is a prototype [2, 1] fabricated in a 40nm low-power technology node. It contains a 32-bit RISC-V micro-processor supporting the RV32IM instruction set, with 32KB of instruction memory and 64KB of data memory.

For each RISC-V instruction, and over many iterations, the current measured is across the highlighted resistor, and an oscilloscope reads the output. The same hardware setup is used for measuring all instructions.



The current passes through a resistor before entering the V_{DD} pin on the chip. The voltage on this resistor is proportional to the current drawn by the chip. Therefore, we can amplify the voltage on this resistor with a non-inverting differential amplifier.

The test board provides this 10 Ohm resistor, and the amplifier is wired using an AD8001 op-amp chip. We fix the chip, and connect it to a differential amplifier. The output is read by a 2.5 GS/s Tektronix MDO3024 mixed domain oscilloscope.

2.4 Coding Setup and Justification

To collect current measurements, we need thousands of iterations of a single instruction. This will help generate a consistent waveform of the current over a small window of time, from which we can find the average current.

We generate a file written in C that contains these repeated instructions. Note that this file first needs to set up the hardware connections between the chip and our computer which collects the data - for example, designating the correct gates, ports, and voltage settings.

Once we are done the hardware connections, and we start repeating instructions for our measurements, there are still a few additional steps to prepare. This is where the setup varies from instruction to instruction; when we describe the setup for ALU or logic instructions, conditionals, and memory instructions in the following sections, this is the step we are referring to.

Generally, however, in the repeated instructions, we have a for-loop iterating thousands of times. To maximize iterations, within the for-loop we hard-code the assembly instruction thousands of times as well. Thus, we have an outer loop wrapped around hard coded repetitions.

There are two reasons for this outer loop and inner repetition setup. The first reason is for our measurement accuracy. Note that we cannot simply run a single for loop with one instruction, such as in the following manner:

```
for (i = 0; i < 1000; i++)
        asm volatile ("inst  a2, a0, a1;" : : : "a2");
}
```

This would result in incorrect current measurements: when compiled to assembly, the PC will be instructed to move and extra steps will be done for the counter after each instruction "inst a2, a0, a1". Thus, the current we measure will be affected by both the instruction "inst" and changing the counter.

However, by repeating the instruction within the for loop, we minimize the impact of incrementing the for-loop counter. For instance, when setting up for ALU instructions, when the inner loop (i.e. hardcoded repetitions) repeats 7500 times, and the outer loop counter increments once. This repeats until the outer loop reaches iteration 1000. This means that for every 7500 times the instruction is run, there will be only one counter incrementation and PC movement - a relatively small impact on the power consumption waveform.

The second reason for this double loop setup is to conserve memory, while allowing for as many current measurements as possible. Since the chip's instruction memory is limited, we cannot unroll the inner loop into hundreds of thousands of iterations; the for-loop allows for more iterations, while still maintaining the integrity of our measurements.

For certain instructions, there may also be additional setup steps; in general, by minimizing the number of setup steps compared to the actual instruction repetitions, we ensure the impact of the setup steps on the measurements is small.

Overall, the code prepares the hardware ports and connections, as well as the iterated assembly instructions. Due to the thousands of repeated lines, we use a separate Python script to generate this main C code. After generated, the C code is compiled into a bitfile which can be fed to the processor.

2.4.1 Repeating Measurements

We now describe this double loop setup in more detail. A Python script is responsible for encoding the number of loops over the tested instruction. The script sets the value of 7500 for inner_loop, and 1000 for outer_loop. Inner_loop refers to the number of lines inside the C code where the assembly instruction is repeated. Outer_loop refers to the number of times the loop is run, while inner_loop refers to the number of copies of the instruction we want to test. For example, if we set outer_loop = 3, inner_loop = 2, we would have:

```
int main (void)
{        ...
    for (i = 0; i < 3; i++)
    {
        ...
        asm volatile ("inst  a2, a0, a1;" : : : "a2");
        asm volatile ("inst  a2, a0, a1;" : : : "a2");
        ...
    }
}
```

Note that the hardcoded `asm volatile` line repeates 2 times, and the for loop iterates 3 times. The elipses represent a constant number of instructions that contribute to overhead but are not related to the instruction being measured. Additionally, the example above uses register instructions, however the same concept applies to other categories (with variations described in each section below).

In practice, we want to iterate as many times as possible to cancel out the effect of noise. We cap the value of inner_loop at 7500 to prevent the output code from being too large for the chip to store in memory. The outer_loop is set to 1000 in order for

all the iterations to complete after a reasonable amount of time, while still collecting enough data. This outer versus inner loop setup ensures we can maximize the number of measurements taken (since the total number of iterations is multiplicative), while still ensuring that the code is small enough to be uploaded to the chip.

2.5 Arithmetic and Logic Instructions

As mentioned previously, we group our instructions into several types: arithmetic and logic instructions, conditional instructions, memory instructions, and program counter instructions. The list of instructions belonging to each type is shown in the table below.

Type	Category	RISC-V Instructions
Arithmetic and Logic	Register Instructions (REG)	xor, add, sub, mul, mulh, mulhu, mulhsu, div, divu, rem, remu, and, or, slt, sltu
	Immediate value Instructions (IMM)	xori, addi, andi, ori, sltiu, slti
	Shifts (SHAMT)	sll, srl, sra, slli, srli, srai
Memory	Load instructions (LOAD)	lw, lh, lb, lbu, lhu
	Store Instructions (STORE)	sw, sh, sb
Conditional Branches and Jumps	Branch instructions (BRANCH)	beq, bne, blt, bge, bltu, bgeu
	Jump instructions (JUMP)	jal, jalr
Program Counter (PC)	PC instructions (PC_INST)	auipc, lui

Table 2.1: RISC-V instruction type groupings.

The setup for each of these types are different; in this section we detail the ALU and logic instructions.

For measurement purposes, ALU and logic instructions can be further separated

into three subtypes, which we will call REG, IMM, and SHAMT.

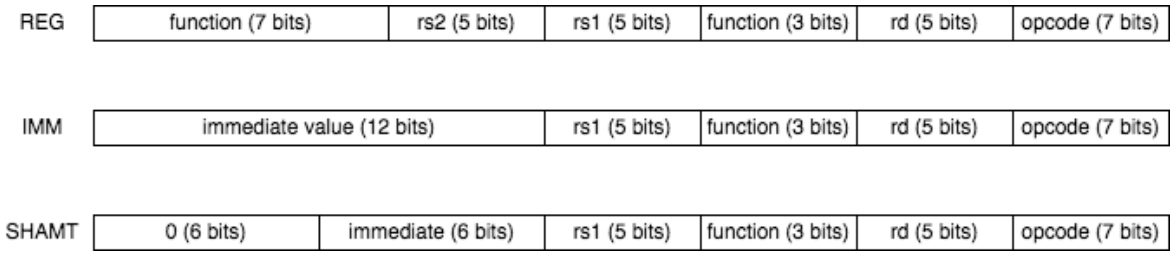


Figure 2-1: Figure 1: Three example instruction formats that are used within arithmetic instructions.

2.5.1 REG

REG instructions use three registers, rd, rs1, rs2. Here, rd is the destination register, and rs1, rs2 are two input registers. For example, "sub" is a REG instruction for subtraction, so "sub a2, a0, a1" takes the value stored in register a0, subtracts the value stored in register a1, and stores this difference in register a2.

Before we can repeat a REG instruction, we need to set up the values in rs1 and rs2. To do so, we first load register a0 (this is rs1), and a1 (this is rs2):

```
    mv a0, [random value 1]
    mv a1, [random value 2]
```

The random values are set by a random number generator. For reference, the C file used to generate this same block of assembly uses

```
randx = random(randx);
asm volatile ("mv a0, %0"   : : "r" (randx) : "a0");
randx = random(randx);
asm volatile ("mv a1, %0"  : : "r" (randx) : "a1");
```

Note that "random(randx)" is the following LCG random number generator:

```
unsigned long random (unsigned long x)
{
    return ((1103515245 * x + 12345) % 2147483648);
}
```

Note that we use random values for a0 and a1 because we want our measurements to be for the REG instruction, i.e. independent of the values stored in the register. As a result, we need to measure over many random input values. This is done at each iteration of the for-loop, for a total of 1000 iterations.

Note that if we generate a random input each time we take a measurement, then we need to perform the two mv instructions into a0 and a1 each time - overwhelming the REG instruction itself and making it measurements inaccurate. This resolved when we hard-code all our repeats within the for loop, as mentioned earlier. There are 7500 hard-coded iterations.

Thus, inside the for-loop, once registers a0 and a1 are loaded, we hard-code instruction repeats. We use a2 as our rd destination register.

Cycle Count

For most supported instructions, each instruction takes a single cycle to execute. However, the `div` instruction requires 32 cycles. Division is usually a complex problem in hardware (except for dividing by powers of two which are equivalent to shifts). Thus we also expect it to have higher power consumption in addition to taking more cycles.

2.5.2 IMM

IMM instructions take input rd, rs1, imm. Like before, rd is the destination and rs1 is an input register. imm refers to an immediate value - a constant. For example, the IMM instruction "addi a2, a0, 2615" takes the value stored in register a0, adds 2615, and stores the result in a2.

The setup for IMM is almost the same as for most REG instructions. However, since IMM instructions have one less input register, we only need to move a value into rs1 in our setup. The immediate value is randomized - this is to ensure that our measurements are for the instruction, rather than dependent on the actual values.

2.5.3 SHAMT

SHAMT (shift-amount) instructions are a special type of instruction, designating how much to shift a value's bits. For example, the instructions "sll" and "slli" refer to left-shift (where each left shift doubles the value), while "srl" and "srli" refer to right shift (where each right shift halves the value).

Although shift instructions really include both register and immediate versions, we classify them separately for data collection, because the shifted value needs to be capped to the lower 5 bits[4]. This is because there is a limit to how much we can shift the bits of a number stored in a register.

In the register version of shift instructions, the inputs are "rd, rs1, rs2", where rd is the destination, rs1 is the register storing the value to shift, and rs2 holds the value to shift by. For example, "sll a3, a1, a4" tells us to take the value stored in a1, left shift the binary value by however many digits specified in register a4, and store the outcome in register a3.

In the immediate value version of shift instructions, the inputs are "rd, rs1, imm". Thus, "slli a3, a1, 3" tells us to shift the value in a1 by 3 bits left, then store the outcome in register a3.

In both cases, the setup is similar to the normal REG and IMM setups, with an added step to check that the shift value (whether immediate or stored in rs2) is capped.

2.6 Conditional Instructions

There are several variants of the branch instruction; each of them takes in two registers rs1, rs2, and an offset value. For instance, BEQ (branch if equal) moves the PC to the next instruction (PC + 4), if the two registers' values are equal. BNE (branch if not equal) increments the PC if the values in the registers are not equal. BLT branches if rs1 is less than rs2, and BGE branches if rs1 is greater than rs2.

Unlike logic and arithmetic operations, control sequence operations will affect

[4]According to the official RISC-V instruction manual, Section 2.4

the location of the PC (program counter) in the next operation. Thus, to test a control sequence operation such as BEQ (branch-if-equal), we must customize the setup differently.

First, we set the values of the two compared registers, rs1 and rs2, to be equal. This ensures the branching will occur. Next, we need to ensure this operation is repeated, in order to take multiple measurements. To do so, we copy the BEQ instruction on the next line, and simply branch to PC+4, which corresponds to the next instruction. This ensures that we perform a branch, while also iterating over BEQ.

Note that for each branching instruction variant, the compared registers rs1 and rs2 are set to the necessary values to ensure that branching occurs. The table below summarizes the values used for the signed variants (the unsigned BLTU, BGEU are set up similar to their signed counterparts). Additionally, the offset is set to 4 for all cases, in order to branch to the next instruction at PC+4, since RISC-V uses PC-relative addressing.

Branch Instruction	rs1	rs2	Offset
BEQ	0	0	4
BNE	0	1	4
BLT	0	1	4
BGE	1	0	4

Finally, after all iterations are completed, we terminate with a NOP. The NOP is a pseudoinstruction that expands to ADDI x0, x0, 0, where x0 is the read-only register storing zero [5]. When analyzing, we may need to consider the implications of performing an add instruction on the power consumption. In general, our data will not be significantly different, as a single NOP will be overshadowed by the thousands of branch instruction measurements. As a result, this small difference will not be reflected in the average power consumption.

[5]Page 20 of https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

2.7 Memory Instructions

Memory instructions (loads and stores) have a slightly more elaborate setup than other instructions. This is because in order to load or store at certain addresses, we need to load the value of that address into another register - an extra level of indirection.

2.7.1 Alignment

There are 3 types of alignments among loads and stores: load/store word, load/store half-word, and load/store byte. A "word" is 4 bytes, a "half word" is two bytes, a single byte is 4 bits.

For example, the load half word instruction "lh a2, 0(a1)" would load the value from memory address "a1 + 0" (where 0 is the offset) to register a2. Note that "a1 + 0" should be aligned to half words, because we are using "lh".

It is important to ensure these addresses are all aligned when we use corresponding word, half-word, or byte instruction. Although RISC-V actually supports misaligned addresses [6], they are not guaranteed to run quickly as they are not always atomic processes. Thus, in our measurements, we will be measuring aligned addresses.

Because these three types of loads and stores align differently, and our range of memory addresses is limited, this means that when we divide these memory addresses according to the alignment intervals we have a different number of slots. For example, suppose we set our base address at 0x8000, and we have a maximum offset of around 2047. In other words, we can use up to address 0x87FF. If we use the "sb" instruction, we store to any of the 2048 addresses and they would be aligned to bytes. But if we use the "sw" instruction, we can only store to addresses that are multiples of 4 - namely, only $\lfloor 2047/4 \rfloor = 511$ word-aligned addresses.

Because the number of available addresses differs for different alignments, we need to adjust our for-loop iterations accordingly. For previous instruction types, we total $1000 \times 7500 = 7500000$ repetitions. To keep the total repetitions around a similar amount, we use the following number of repeats for inner and outer loop. Note the

[6] Section 2.6 of the official RISC-V manual, Load and Store Instructions

precise number of repetitions doesn't matter since we are obtaining an average current measurements, but we'd like to be somewhat close, in order to be consistent between instructions.

Instruction	Inner Loop (Hardcoded Repeats)	Outer Loop (For Loop)
lw	511	16000
lh	1023	8000
lb	2047	4000
sw	511	16000
sh	1023	8000
sb	2047	4000

2.7.2 Load Setup

In all instructions (load and store), we first store the base memory address 0x8000 in a1. Then, we load values into a1 + offset for various offsets.

To prepare for load instructions, we need to prepare our values before the for-loop. We first move a random value into register a2. We then use the store instruction with the same alignment[7] to store a2 into offsets of a1.

Once the values are loaded at the relevant memory addresses, we start the power. Then, we start the for-loop iterating the instruction, and hard-code repeats for the load instruction in the for loop (see Appendix).

2.7.3 Store Setup

To prepare for store instructions, within the for-loop, we move a random value into a2. Inside the rest of the for-loop, we repeatedly load this random value into a1 + offset for various offsets.

Note that here, in order to make our measurement independent of the value being stored itself, we randomize the value in register a2 at each iteration of the for-loop. This differs from the setup for load instructions: since each call to a load instruction

[7]For example, if we use lh, we will use sh; if we use lw, we use sw.

requires a corresponding store instruction (to first store a value at that same memory address), there are just as many loads as stores in the setup. Thus, to ensure we are only measuring the load instructions, all the setup must be moved before the for-loop.

2.8 Further Setup Improvements and Modifications to Handle Noise

As described earlier, a bitfile is created per instruction we want to measure current for. To eliminate errors coming from noise, we average over 10 measurements.

Between each instruction, it is important to note that the chip may heat up. This means that over several consecutive measurements, the later measurements will have accumulated more heat. Thus, we need to modify our setup, to ensure enough time for heat dissipation.

Ideally, a device would be allowed to cool indefinitely at room temperature; however, this is not always possible given time limitations. Thus, there's a tradeoff between how much we've accounted for heat dissipation (by waiting between measurements), versus how much time it takes to collect all data. We want to minimize the total time for data collecting while still maintaining consistent measurements, so we consider finding optimal time whereby the device has cooled enough, and the wait time is reasonable. After fixing a wait time, we simply need to ensure it is consistent across all instruction measurements.

We approximate the time needed to dissipate heat by taking measurements with a wait time of 1 minute, then 2 minutes, up to 10 minutes.

The wait time refers to the time between measurements for different instructions. We wait 5 minutes rather than running continuously because we want to ensure the effect of temperature to be minimal. Otherwise, the chip will heat up and measurements will be inaccurate for later instructions.

Note that at the 5 minute mark, we use several arithmetic and logic instructions to analyze this effect. As a result, running one instruction (bitfile) takes about 1 minute

plus 5 minutes of waiting time for heat dissipation. In total this takes 2.6 hours to run.

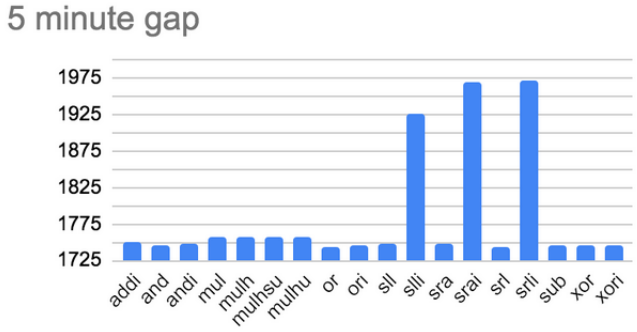
In particular, if we track the `mul` instruction, we see the following results for the first five minutes:

Wait Time (minutes)	Current (micro amps)
1	1760.52
2	1759.39
3	1759.22
4	1760.32
5	1760.69

Table 2.2: Current measurements for the multiply (mul) instruction after successive wait times.

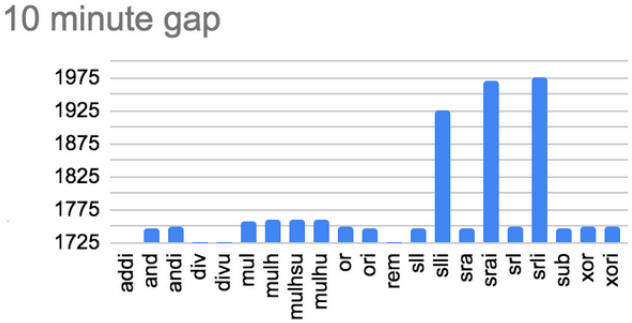
The maximum measurement (1760.69) is within 0.8 percent of the minimum measurement, 1759.22. This suggests that the wait time has very little impact on the measurement. We'd like to confirm this across the other instructions, and also check whether this also holds from 5 to 10 minutes.

Figure 2-2: Current measurements with 5 minute wait times between successive measurements.



The figures above show the current measurements at 5 minutes, and 10 minutes. A cursory examination shows us that the current measurements for each instruction between 5 and 10 minutes are similar. A detailed comparison (see table below) shows

Figure 2-3: Current measurements with 10 minute wait time between successive measurements.



that the percentage difference is not more than 1 percent. (The most positive and most negative percent changes are highlighted.) Noise alone can account for up to 0.5 percent difference in measurement; thus, our results suggest that any amount of wait time from 5 to 10 minutes is a good choice for the setup, as the measurements are within 1 percent.

2.8.1 Assessing the Effect of Noise

Noise contributes to fluctuations in our measurements. As a result, all our current values are averages over 10 consecutive measurements. Nevertheless, we'd like to book-keep the noise fluctuations and discover possible patterns. The following measurements were taken with the 5-minute wait time between different instructions. We include two metrics: maximum measurement divided by minimum measurement, to see the percentage change; and standard deviation. The percentage change is meaningful as we can compare them across different instructions. We see, across all instructions, that the ratio of maximum to minimum current is within 0.5 percent, and standard deviation is within 3.2. We conclude that the extent to which noise affects our measurements is minimal.

Instruction	Average current (μA) 5 min	Average current (μA) 10 min	Percent Difference
add	1749.85	1740.59	0.529
addi	1750.45	1726.52	1.367
and	1746.03	1748.22	-0.125
andi	1748.59	1749.08	-0.028
mul	1757.36	1758.37	-0.057
mulh	1757.1	1759.25	-0.122
mulhsu	1758.07	1761.39	-0.189
mulhu	1757.98	1760.96	-0.170
or	1744.44	1749.33	-0.280
ori	1745.47	1748.24	-0.159
sll	1747.28	1746.82	0.026
slli	1926.32	1925.89	0.022
sra	1747.34	1748.27	-0.053
srai	1969.52	1971.63	-0.107
srl	1744.38	1749.93	-0.318
srli	1972.56	1976.86	-0.218
sub	1744.99	1748.05	-0.175
xor	1745.74	1748.9	-0.181
xori	1745.51	1748.9	-0.194

Table 2.3: Percent difference of average current for selected arithmetic instructions, with 5 minute and 10 minute wait times. (Percent difference values are relative to 10 min average current).

Instruction	Average Current (μA)	Ratio of Max Current to Min Current	Standard Deviation
add	1749.851	1.002	0.895
addi	1750.449	1.003	1.756
andi	1748.587	1.002	1.330
and	1746.030	1.003	1.366
mulhsu	1758.072	1.002	0.875
mulhu	1757.980	1.001	0.667
mulh	1757.099	1.002	0.967
mul	1757.360	1.003	1.800
ori	1745.465	1.002	0.873
or	1744.438	1.004	1.913
slli	1926.319	1.005	3.123
sll	1747.280	1.003	1.396
srai	1969.521	1.003	2.286
sra	1747.356	1.002	0.955
srli	1972.554	1.002	1.331
srl	1744.382	1.003	1.572
sub	1744.993	1.003	1.515
xori	1745.507	1.002	1.204
xor	1745.735	1.002	1.306

Table 2.4: Measuring the extent of noise for selected arithmetic instructions.

2.9 Final Average Current Measurements

The table below shows the current measurements for each instruction. Results have been averaged using a 10 minute wait time. These values will be used in the final simulator.

Notably the arithmetic and logic instructions under the REG category consume around 1400 to 1900 microamps, with the exception of division and remainder. This is expected since division consumes more power and also requires more cycles. Notably the branch and jump instructions have higher current, closer to 3000 microamps. Generally, instructions taking an immediate value also consume less than the same instruction taking in a register (for example, `andi` takes around 1 microamp less than `and`); however this appears to be the opposite for shift instructions compared to their immediate-instruction counterparts (`sll`, `srl`, `sra`).

Instruction	Instruction Type	Average Current (μA)
sll	REG	1747.28
srl	REG	1744.38
sra	REG	1747.34
xor	REG	1745.74
add	REG	1749.85
sub	REG	1744.99
mul	REG	1757.36
mulh	REG	1757.10
mulhsu	REG	1758.07
mulhu	REG	1757.98
div	REG	2942.86
divu	REG	1476.47
rem	REG	2932.05
remu	REG	1465.62
slt	REG	1750.97
sltu	REG	1749.58
and	REG	1749.85
or	REG	1744.44
slli	IMM	1926.32
srli	IMM	1972.56
srai	IMM	1969.52
xori	IMM	1745.51
addi	IMM	1750.45
andi	IMM	1748.59
ori	IMM	1745.47
sltiu	IMM	1750.24
slti	IMM	1749.31
beq	BRANCH	2904.98
bne	BRANCH	2924.13
blt	BRANCH	2936.27
bge	BRANCH	2914.36
bgeu	BRANCH	2920.13
bltu	BRANCH	2933.06
lw	LOAD	2434.12
lh	LOAD	2465.97
lb	LOAD	2453.45
lhu	LOAD	2458.88
lbu	LOAD	2450.49
sw	STORE	2727.06
sh	STORE	2896.19
sb	STORE	1838.31
jal	JUMP	2919.94
jalr	JUMP	2919.94
auipc	PC_INST	1751.49
lui	PC_INST	1749.08

Table 2.5: Average current for each instruction.

Chapter 3

Design of Instruction-Level Power Simulator

We create a RISC-V Python simulator which can take in any RISC-V assembly code and output an approximate power consumption waveform. This Python simulator supports all non-privileged instructions which don't modify system-configuration registers, supports alternative register names (for example, "zero" is equivalent to "x0"), two's complement negative values, and both hexadecimal and decimal constants.

Note that in our workflow, we compile C code to binary then disassemble it to get assembly code, and finally feed it to the Python simulator in conjunction with our power measurement data. As such, pseudoinstructions will be replaced in the dissassembling stage, and we do not need to support them separately. (Although not needed for our purposes, theoretically if a pseudoinstruction decomposes to multiple smaller instructions, we would need to support this as a single instruction, because replacing it with multiple instructions would offset any future jump or branch instructions).

After stepping all instructions through our simulator, we output the approximate power waveforms. These instruction-level waveforms will suffice for simple power analysis (SPA).

3.0.1 Inputs and Initial Values

The inputs to the Python simulator include instruction to power mappings, and memory address to value mappings. For example, the user can specify the power consumption value for every supported instruction. Not every instruction needs to have a specified power, however these unspecified values will be left blank in the final output. Furthermore, since these are simply numerical values, the user can choose to use current instead of power - the simulator will simply plot them chronologically according to which instruction is executed.

The user can also specify, in any order, values inside memory addresses. The user has the option to specify addresses that they want to have randomized. By default, unspecified memory addresses are set to zero. Similarly, all register values are initialized to zero.

3.0.2 Outputs

During the simulation, the user has the option of a verbose output which prints out every instruction the simulator steps through. This can be useful for debugging assembly code.

After the simulator steps through all instructions, it outputs the power consumption waveform, using the values from the input for each instruction's power. The output is a chronological graph. The simulator also has the option to save the final memory state separately. This is useful if the user wants to check memory values, or use the ending state for a future simulation.

3.0.3 Operation

During operation, the program counter starts from the first instruction (PC = 0). Typically, when there are no branches or jumps, the PC increments by 4. If there are conditional statements, the PC will behave accordingly. All behaviors were checked with the ISA instruction manual.

Initially, the simulator reads the input assembly code and memory. After stripping

any blank lines from the input code, and initializing a `MemFile` instance to represent the memory, the simulator can be run.

The simulator runs by stepping through each line of assembly code. There is an option to step through line by line, or run a specified number of lines, or run the full code until completion. Fundamentally, these instructions operate the same way by calling the same `parse_and_exec()` instruction to parse and execute each instruction. All of these instructions contain a `verbose` option allowing for detailed execution descriptions, which is useful for debugging.

Additionally, the `run()` function includes a check to ensure the next step will not put the PC out of bounds of the input assembly code.

To parse each instruction, we use regex matching to determine the class of instruction. There are three types of string formats for the instructions:

- `inst rs1, rs2, rs3`
- `inst rs1, offset(rs2)`
- `inst rs1, offset`[1]

Based on these regex format and operation similarities, we group our instruction classes. For example, all REG instructions are grouped because they use the same "inst r1, r2, r3" format, read from two registers, and write to a register. This results in the following groupings - note they are similar to the power consumption groupings mentioned in earlier chapters, however the categorization here aims to help modularize the Python simulator.

Each instruction type corresponds to a separate Python function. Thus, after an instruction is classified, its corresponding function handles the instruction by case, according to the RISC-V ISA Manual's description (see previous chapters).

During operation, all register values are stored and updated from a dictionary, which maps register name to value. Numbers and constants will be passed through `HelperFunctions` to obtain the true, 32-bit 2's complement value.

[1]This is seen in jump instructions.

3.0.4 Register Reads and Writes

Before we can handle register reads and writes, we first need consistent register naming.

In order to support alternative register names (as mentioned in a previous section) in the input assembly code, we load all the equivalent names into a dictionary `self.reg_name` mapping a name to the corresponding register. Within the simulator, all names are standardized to use `x0` through `x32` - this means that if an instruction uses a different register such as `s0`, we need to call `get_reg_name[s0]` to obtain the register name `x8`.

Rather than hard-coding the contents of this register-name dictionary in Python (which would be tedious), we import them from a text-file. In this text file, each line maps a name to the register name (i.e. names in the form of `xi`). Note that this is a one-time read that saves all values into a dictionary, thus it is not computationally expensive. It is also beneficial if we ever want to support different register names.

Now that all names use `x0` through `x32`, we can handle reads and writes consistently. A dictionary `registers` maps each of these standardized register name to a value in hexadecimal. (All values are stored hexadecimal for consistency, and converted to decimal when needed for arithmetic). If a register is not initialized, by using python's dictionary `get(index, default)` function, we assign the default value to zero. Note that all writes to register `x0` are ignored, and all reads will load the value zero. (Equivalently, this means when our simulator runs, the dictionary `registers` will never need a key for `x0`, because the default value zero will be automatically assigned to registers that are not keys).

3.0.5 End of Execution

Execution stops in three cases: end of file, error, or `wfi` (wait for interrupt). The first two are intuitive (and custom debug statements will print in the case of errors); the `wfi` suggests waiting for an external interrupt outside of this thread. This instruction appears after we disassemble C code.

3.0.6 Additional Options

Plotting

Since the main use case will be checking the final power consumption plot, the user has the option to set `plot=True` when calling `run()`. After running, the user can call `show_power_plot()` to display the final plot (with options to specify the min and max of the x, y axes).

Reset

At any point, the simulator can also be reset by calling the `reset()` function, which zeroes all registers, re-reads the input code file, and re-sets memory according to the memory file.

Debugging

Debugging is also included, which will throw exceptions specifying which line of assembly code resulted in error. We currently detect invalid instructions, invalid instruction formats, and invalid pointers (which are not multiples of four).

PC Pointer

In general, the PC is only allowed to access positions that are multiples of four. This prevents illegal reads and writes. As a result, our Python simulator is built on this assumption - we do not support accessing the bit level instructions, or shifting the reading frame.

One exception is in certain load and store instructions; however these are finely constrained to their specific offsets following the ISA manual's description.

Helper Functions

To handle positive and negative (two's complement) numbers, as well as identifying hexadecimal from decimal numbers, we have three separate helper functions - one to convert integer to hex string; another to convert hex string to integer; and finally

one that identifies whether it is hex or integer, and outputs the corresponding integer. These functions are handled separately to keep the code modular; furthermore, they are implemented as static functions that can be called directly without initializing the class.

We need to identify hexadecimal from decimal numbers in many situations - one example occurs when reading inputs. The input assembly code - generated from the C file - often contains immediate values expressed in decimal, as well as branch or jump offsets expressed in hexadecimal. (A preprocessing script - described in following sections - will ensure that the RISC-V code and memory addresses, every hex number is appended with `0x`.)

We need to support two's complement because constant values can be either positive or negative. Furthermore, our functions are constructed to perform correctly in the case of overflows. For example, `int_to_hex_string(hex_string_to_int(0x1FFFFFFFF))` gives the correct output -1 - this is because `0x1FFFFFFFF` overflows 32 bits, so the top bit would be truncated.

Although there may be existing libraries to handle such computations, it is more lightweight to implement them directly using a combination of built in functions like `int()`, `hex()`, and modulus. We don't need to import an entire library; furthermore, we can customize it to support 2's complement, and our desired bit length. Few existing libraries support both of these two properties.

Additionally, we operate at the granularity of hexstring rather than bitstring for two reasons: first, hexstring suffices for our arithmetic - we can support integer to hex conversion and vice versa, described below. Second, any non-decimal arithmetic will need to be handled as a Python string; thus an 8-character hexstring is more memory efficient than a 32-character bitstring.

Python's `hex()` function directly converts any integer to a hexadecimal value, however there are two problems: firstly, it does not limit the number of bits, and secondly, it does not support two's complement - negative numbers are simply the same hexadecimal value, with a negative sign appended.

Thus, to achieve two's complement with 32-bit numbers, we need to write our own

function. First, when converting an integer to hex string, we need to find the integer's value mod 2^{32}, as this would give us the lowest 32 bits. If we have a positive integer, this would simply be a case of checking if the integer overflows 2^{32}, and truncating any overflows. If we have a negative integer, it would be equivalent to adding 2^{32} first, then taking the modulus. To see why, consider -1: this becomes $2^{32} - 1$. If we *then* find the hex value of $2^{32} - 1$ using Python's `hex()` function, we get `0xffffffff` because Python's `hex()` treats all inputs and outputs as positive values.

The above is summarized in the following code, which generates a 2's complement, 8-bit hex string from an integer value:

```
intvalue = (2**32 + intvalue)%2**32
hex_str = hex(intvalue)[2:]
return '0x' + '0'*(8-len(hex_str)) + hex_str
```

We also need to handle the opposite case: generating the integer value from a hex string `hexstr`. Though Python has a function `int(hexstring, 16)` which converts an integer into hexadecimal, again it does not support fixing a bit length or two's complement. Thus, we add another step to calculate the actual value:

```
actualvalue = (intvalue - 2**31 + 1) %(-2**32) + 2**31 - 1
```

To illustrate how this works, consider positive `intvalue` integers that come from positive hexstrings. Subtracting 2^{31} will give a negative number. In python, the negative modulus of a negative number is negative, so we add back 2^{31}. Similarly, adding and subtracting 1 also cancels out.

Now consider the other case: `intvalue` integers that come from negative hexstrings. Because Python's `int()` function only generates positive integers, it won't apply 2's complement: instead, we'll get positive integers greater than or equal to 2^{31}. Then, by subtracting 2^{31}, we still have a positive value. In Python, the negative modulus of this positive number will yield a negative value (for example, `2%-3 = -1`). After the modulus, we can add 2^{31} back and we will still have a negative value.

The extra $+1$ is needed to handle the `hexstr = 0x80000000` edge case. In 2's complement, this should be equal to -2^{31}. When we apply Python's `int(hexstr,16)`

directly, this gives an `intvalue` of 2^{31}. If we do not add and subtract 1, then plugging in this `intvalue`, $2^{31} - 2^{31} \mod (-2^{32}) + 2^{31} = 2^{31}$, instead of -2^{31}. By adding 1 and subtracting it after, we ensure that `hexstr = 0x80000000` gets mapped to -2^{31}.

Usage of HelperFunctions Class

In addition to converting between integer and hexadecimal, one common use case is to double-wrap the input in order to truncate any overflow. For example, in some instructions with immediate values, the immediate (constant) could be any integer (in particular, this could happen if the assembly code is hand-written as opposed to generated by C disassembly). To ensure that we cap it within 32 bits, we can do the following:

```
hex_string_to_int(int_to_hex_string(const))
```

Though there exist alternative ways (such as checking bounds, taking the modulus, checking the sign, etc), these two steps encapsulate all these checks in one line. The inner function converts the constant to a 32-bit 2's complement hex string, effectively truncating any overflow. The second step converts this hex string back to an integer within our desired range, and we can proceed to perform arithmetic with this integer value.

Note that the HelperFunctions treat numbers as 2's complement, and therefore, signed values. If we want to treat numbers as unsigned, we will need to use `int(hexstr,16)` directly to get the positive value. This is the case for example in the SLTIU instruction, where both the register's value, and the constant, are compared as unsigned values.

3.0.7 Memory Class

For modularity, a separate memory class is constructed in Python to hold memory values, and an instance is initialized in the simulator. This memory class stores a dictionary mapping memory address to its value. This dictionary was chosen for several reasons. Firstly, it allows the user to provide memory values in any order.

Dictionaries also allow O(1) access (due to hashing). Although a list implementation would be valid, often the user may provide sparsely filled memory cells that are far apart, separated by stretches of unspecified or empty cells. It would be very space-expensive to index all these empty cells in a list.

Secondly, we chose a dictionary over writing to an external text file. This is because at the end of the program, we may not want to overwrite the initial memory input file, but we still want the option to choose our final save location. With a dictionary, this is convenient because we simply have to write its contents to an output location.

Additionally, the memory class is able to handle parsing, updates, and reading from uninitialized locations. Handling these miscellaneous tasks in one class ensures that we don't need to sprinkle checks in our main simulator. We want to handle parsing because when the memory file is first provided (i.e. the initial values at each memory address), the raw text file may have whitespaces we'd like to skip.

Updates to memory locations (for example, after the simulator runs a *store* instruction) are done by converting the address and the data into hex values. Note that the memory class stores all values in hexadecimal to ensure consistency, so we clean all inputs before storing them.

Finally, the memory class handles writing and reading, which is as simple as updating or reading the memory dictionary. Reading from uninitialized memory locations defaults to 0.

3.1 Other Pre-Processing and Container Scripts

Because the Python simulator is just one step in our entire workflow, some pre-processing steps are needed to ensure the input script is compatible with the simulator.

3.1.1 Memory Pre-processing

When we disassemble C code, initial memory addresses and values can be generated. The pre-processing script ensures all memory addresses are formatted in hex-string format. Python requires the hex string in `int(hexstr,16)` to be prefixed with `0x`:

our pre-processing script appends this to all values.

3.1.2 RISC-V Code Pre-processing

The disassembled C code consists of a medley of instructions using absolute addressing. This is an issue because the RISC-V ISA - upon which the simulator is based - describes jumps and branches as relative-addressed. Although we could add a relative or absolute address option to the simulator, another approach is to keep the simulator consistent with the official RISC-V ISA definitions, and pre-process the RISC-V code to be relative-addressed. To do so, for every jump or branch related instruction, we simply subtract the current address from the absolute address.

In the assembly code, there were actually inconsistencies in relative versus absolute addressing. Upon close inspection, the `jalr` instruction was disassembled as relative-addressed, while all other jump related instructions were absolute. This is probably due to how pseudoinstructions were substituted. Thus, the pre-processing code was adapted to handle this inconsistency. These inconsistencies are another reason why handling pre-processing separately is better than setting the simulator to use *only* relative or *only* absolute addressing, as not all lines use the same type of addressing.

Finally, another inconsistency in the disassembled code is in the numerical values. Some numerical values were in decimal, while others were in hexadecimal. After inspecting all instructions, it was clear that branch related instructions used hexadecimal but were not prefixed by `0x`; other instructions' immediate values were all in decimal. The pre-processing script also replaced these hexadecimal values with the correct `0x` prefix.

For efficiency, the above steps are handled in a single loop that walks through the raw RISC-V code line-by-line.

3.1.3 Command line container file

Though the simulator can be used as an object and imported, a container file is created so that we can also run it in the command line. This. is a very specific use case thus

the file supporting this is created separately from the main simulator class.

An example command would be:

`python3 command_line_py_riscv.py memlistfile=rand_mem_locations.txt codefile=code.txt powerfile=pwr.txt outfolder=out_folder`

`command_line_py_riscv` is the name of this container file, `rand_mem_locations.txt` specifies memory locations we want to initialize with random values, `code.txt` specifies the assembly code file, and `pwr.txt` specifies the file listing power consumption for each instruction. This way, users preferring the command line can run the simulator directly, or write bash scripts.

Chapter 4

Evaluation and Application of Simulator

4.1 Comparison of Simulated and Experimental Results

An effective simulator will yield results that are comparable to experimental measurements. To verify this, we simulate the execution of a simple neural network[1]. The network has three layers with three to four neurons each[2]. This neural network has pretrained weights based on the iris-dataset[3]. The networks parameters were converted to fixed point with final weights scaled and rounded to integer values, and biases also scaled to match the weights in each layer (in other words, scaled by 100 in the first layer, by 100^2 in the second layer, and so on, because neighboring layers are fully connected; the final prediction value is not affected since it corresponds to the maximum index, not the numerical value.). This was done to avoid the use of floating point (since our test chip and simulator do not natively support floating point instructions) and to ensure clarity of the generated assembly code (compared to using

[1]See Appendix for full code in C.

[2]Network structure inspired by https://towardsdatascience.com/building-your-first-neural-network-using-keras-29ad67075191

[3]https://archive.ics.uci.edu/ml/datasets/iris

floating point numbers) as we can then quickly determine which memory locations have been modified. Along with validating functional correctness of the simulator, our primary goal is verifying the simulator's ability to estimate average power and timing.

	Total Time (μs)	Average Current (μA)
Simulated	4.08	1977.6
Experimental	4.15	1921.8

Table 4.1: Experimental and simulated results for average current and timing.

The simulated number of cycles was 163, and the simulated average current was 1977.6 μA. Experimental results are measured at 1.1V and 40MHz. The average current is within around 2.9 percent of the experimental, and the simulated time is within 1.7 percent of the experimental. Thus the simulator's timing and average current values are are a very good estimate of experimental values.

Three extra cycles are required for the additional instructions used to experimentally measure execution timing, and this accounts for the 0.07 us difference between simulation and experiment.

4.2 Evaluation of Simulator

To evaluate and validate the simulator, our first candidate function is a pseudo-random number generator written in C:

```
unsigned int random( int xlen )
{
    unsigned int randx = 4017786737;
    while (xlen > 0)
    {
        randx = (1103515245 * randx + 12345) % 4294967295;
        xlen--;
    }
    return randx;
}
```

Note that 4294967295 = `0xffffffff` - this will put our simulator to the test: it needs to address overflows, and also treat this value with an unsigned comparison using SLTIU.

In `main`, `random()` is called 11 times different input values, where `a[11] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}` and `b[11] = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100}`:

```
void main( void )
{
    int i;
    for (i = 0; i < 11; i ++)
    {
        c[i] = random(2020 + (a[i] * b[i]));
    }
}
```

The code is then translated into RISC-V assembly, and the initial memory values are also saved into a text file for the simulator to read.

Note that the assembly code has three parts (see Appendix): the start (which saves the stack pointer), followed by `main()`, which then jumps to `random()` and returns after the jump. In particular, note that the corresponding a[i] and b[i] values are retrieved from memory using load instructions. Some pseudoinstructions like `mv` can be replaced by equivalent instructions (such as `addi` in this case).

If we walk through the RISC-V assembly code in detail, there are a few notable registers. In main(), s4 stores the address pointing to the top of the a[i] array, s3 stores the top of the b[i] array, and s2 stores the top of the c[i] array. These are used to help load the correct input (2020+a[i]*b[i]) into register a0, the input of the random() function.

Once we enter random(), a3, a4 and a5 are used for calculating the arithmetic. a3 stores the constant 12345. a4 (the constant 1103513245) is multiplied with a5 then this value is stored in a5. Thus, a5 effectively stores randx. a0 stores the counter xlen.

The simulator performs correctly: first we run the C code direcly by compiling with gcc, adding a line to print out every `c[i]`. This gives us the correct outputs that we want. Then we check the simulator's memory output - we find at the addresses occupied by array `c`; all values match. Thus we've confirmed the simulator performed correctly on a medium-sized assembly code file.

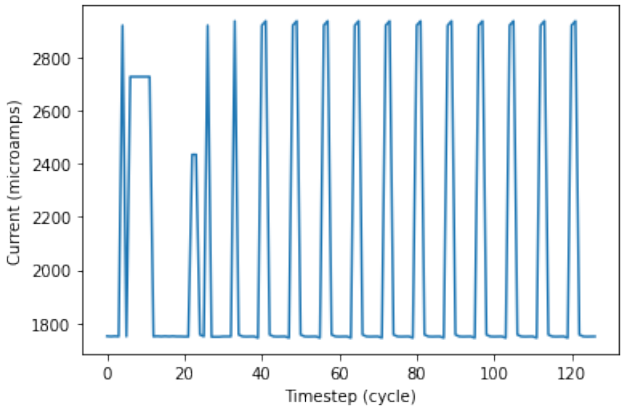
4.2.1 Power Analysis using Simulator

For this particular pseudorandom function, we can explore how much the simulator can help with simple power analysis.

Using the current values for each instruction (collected in previous chapters), we obtain an average current of 2044.975μA over all 530008 instruction cycles to terminate running the pseudorandom number generator. Plotting the current waveforms, we observe a few notable points that show where the code could potentially be vulnerable for attack. (Note that the power consumption waveform will be proportional to the current waveform, because $P = iV$; so in most cases it suffices to observe the current waveform).

Because the simulated measurements span 530008 cycles - too large to display in its entirety - we will zone in on specific points of interest, as much of it is repetitive. In the beginning, the waveform is noticeably different. This is expected because the program counter has to first step through several instructions before jumping into the `main` function.

Figure 4-1: Waveform of current for the first few cycles.



Throughout most of the timesteps, the waveform has a regular pattern shown in the figure below. Near the end, the waveform shows an irregular pattern as it exits out of the `main` block - again, this is expected since there are several `load` instructions and a `jump` before the final `wfi` stop instruction is reached.

Figure 4-2: Waveform of current. This sample reflects the majority of cycles.

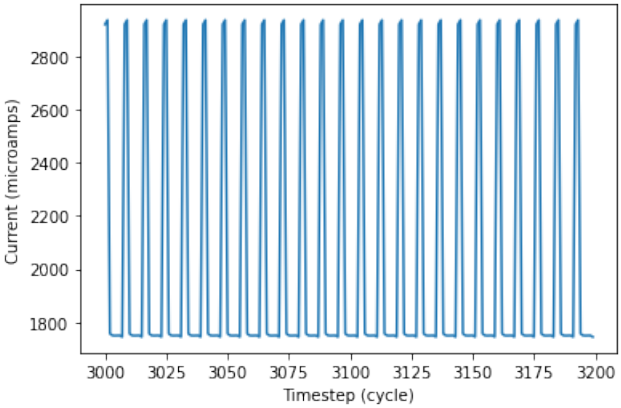
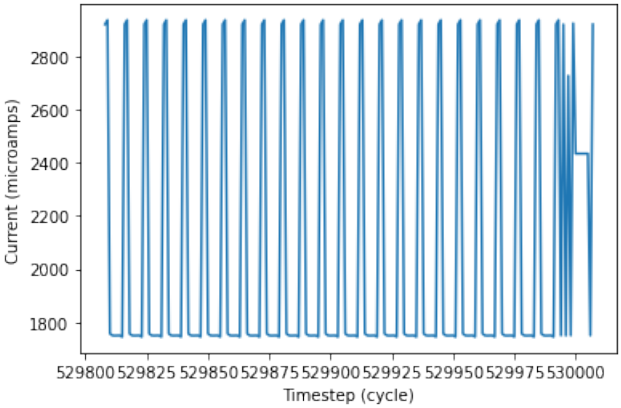


Figure 4-3: Waveform of current in the final cycles.



However, there are a few exceptions to this regular pattern, in addition to the start and end. At around timestep 16000 - 17000, there is another noticeable shift in the waveform pattern. Similarly, around timestep 34000, there is another shift in waveform pattern. In fact, using the output waveform, if we examine all 530008 cycles, we find exactly 12 cases of these anomalies, happening around cycles 27, 16207, 33987, 54967, 80747, 112927, 153107, 202887, 263867, 337647, 425827, and near cycle 530008 where the code terminates. This immediately exposes the fact that we initialized arrays `a, b` with 11 values, representing the 11 intervals between these 12 anomalies.

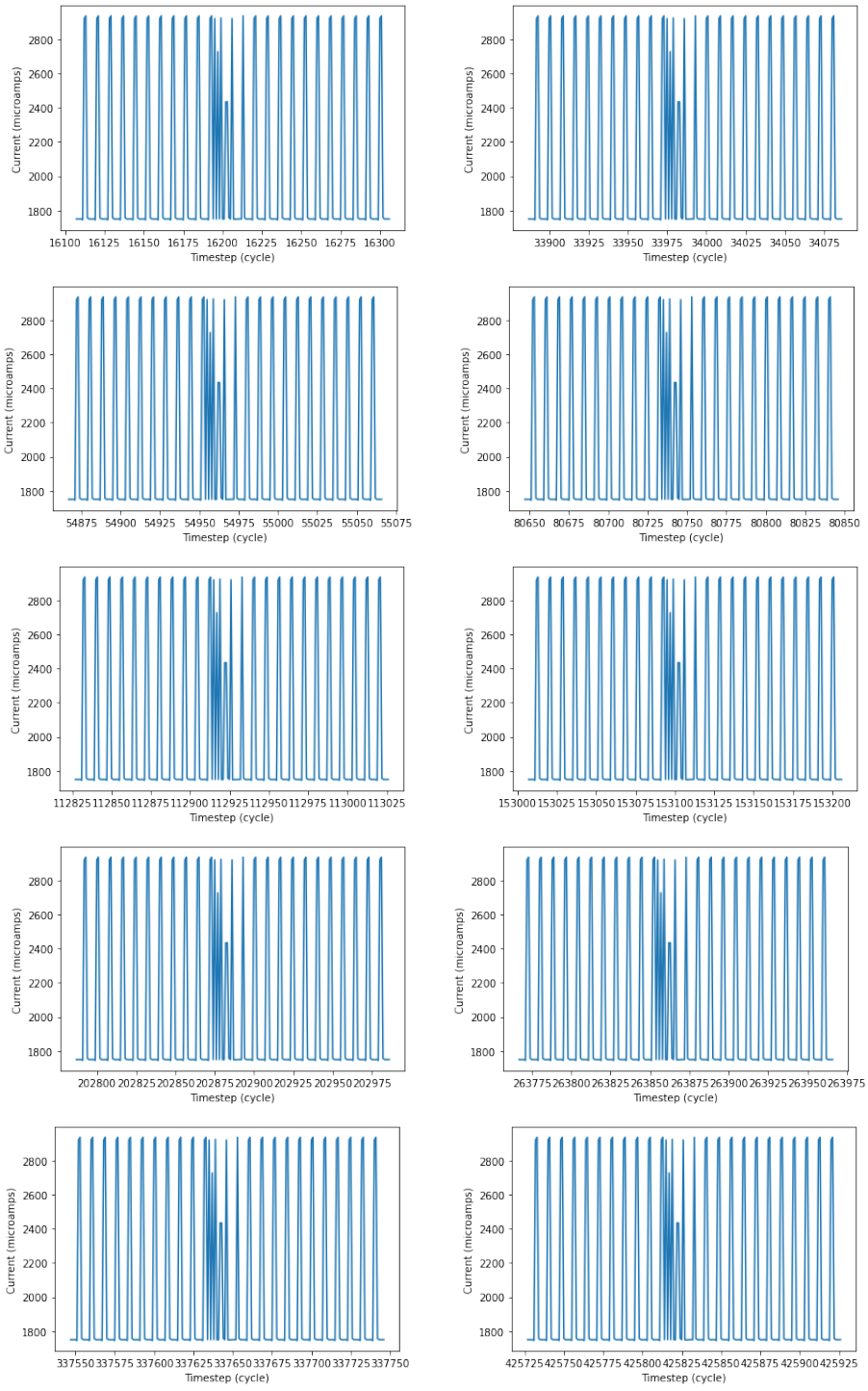
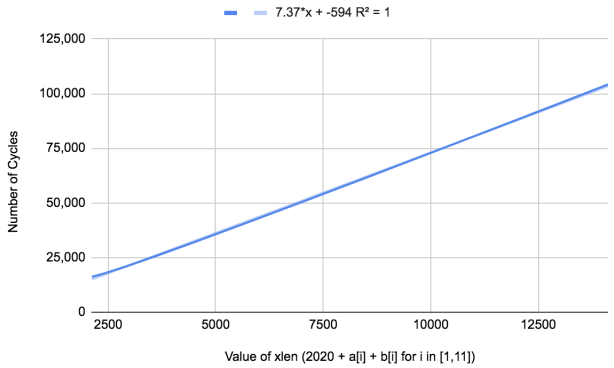


Table 4.2: The 10 other anomalies (in addition to the start and end) where the waveform pattern is broken. Their timesteps (cycles) correspond to 16207, 33987, 54967, 80747, 112927, 153107, 202887, 263867, 337647, 425827.

Furthermore, if we look at the difference between each of these anomalous time steps, we find that they reveal the value of `xlen` used by the pseudo-random number generator. The 11 seed values entering the pseudo-random number generator are $1 \times 100, 2 \times 200, ...11 \times 1100$ based on the arrays `a, b`. Similarly, the number of time steps between each of these anomalous cycles is exactly proportional to these `xlen` values, corresponding to 16180, 17780, 20980, 25780, 32180, 40180, 49780, 60980, 73780, 88180, and 104181 cycles. We can see this proportionality in the figure below.

Figure 4-4: Plot of the number of cycles between each call of `random` versus the value of xlen.



The ratio happens to be 7.37 time steps per `xlen` value according to this graph (with a y-intercept of -594). In fact, this aligns with our expectations, because in the assembly code, each iteration inside the `random` function takes 8 steps (the exact value 7.37 is due to extra instructions which happen before and after entering `random`).

In particular, the following lines of assembly correspond to the loop in `random`. Note addresses are relative. We can see that the first branch instruction almost always skips to `mul` (except when `random` is finished, in which case `a0` is equal to 0 and the jump on the third line takes the `pc` back to `main`). After that, 7 more instructions ensue until the final `jal` is reached, which jumps back to the first `blt`.

Thus, if an observer knows this value is 7, or if they know any one out of the 11 `xlen` values and its index, then they are able to infer both the number of steps per iteration of `random`, as well as all other input `xlen` values. Fortunately, the actual computation and modulus is not exposed, and thus the observer would not know the final output value; however we can imagine a case where knowing the input value of

```
blt zero,a0,0xc
addi a0,a5,0
jalr zero,ra,0x0
mul a5,a5,a4
addi a0,a0,-1
add a5,a5,a3
sltiu a2,a5,-1
addi a2,a2,-1
sub a5,a5,a2
jal zero,-0x24
```

Figure 4-5: Excerpt of assembly code corresponding to `random` function.

a function - based on the number of cycles it takes - could already be exposing too much information.

In other words, within the power consumption waveforms - which are related to these current waveforms - the difference between anomalous cycles actually exposes information on the relative size of the input, `xlen`.

4.3 Evaluation of Simulator for Simple Power Analysis (SPA) Side-Channel

The simulator can be used to detect simple side channel leakages at the instruction level, either in power consumption or in timing.

To illustrate this, we run the simulator on two different versions of a discrete Gaussian sampler: a naive approach, and a modified approach that counters simple power analysis and timing attacks.

We have chosen the discrete Gaussian sampler because it performs an important step in lattice-based cryptography: it generates vectors with coefficients in the discrete Gaussian distribution. Thus, ensuring its safety against side channel attacks is crucial to security.

For our sampler, the output is always a number in the range of -5 through 5 inclusive, and the frequency of each of these outputs matches a Gaussian distribution

```
for (i = 0; i < CDF_TABLE_LEN; i++)
{
    if (prnd <= CDF_TABLE[i]){
        break;
    }
}
sample = (unsigned int) i;
s = ((-sign) ^ sample) + sign;
return s;
```

Figure 4-6: C code for the naive discrete Gaussian sampler.

centered at 0.

4.3.1 Naive Discrete Gaussian Sampler

The first version of this sampler uses a naive approach, while the second has been adjusted to be safe from simple power analysis and timing attacks. For each approach, we run the Gaussian sampler 1000 times with randomized input seed values.

Consider the following C code excerpt for the naive approach.

`CDF_TABLE` is a list of the following increasing values `9142, 23462, 30338, 32361, 32725, 32765`, where our value `i` ranges from 0 through 5 inclusive. `prnd` holds the last 16 digits of a random seed value, right shifted by 1 bit.

Running this experiment 1000 times with uniformly randomized seed values, we can plot a histogram final value of register `a0` (the value returned by the Gaussian sampler). As expected, this follows a Gaussian distribution for values between -5 and 5, verifying correctness.

While the outputs are numerically correct, in this naive approach, we discover that the number of cycles depends on the output value. In fact, if we compare the total cycle count, we discover 6 peaks, corresponding to 6 classes of outputs in increasing order: $0, \pm 1, \pm 2, \pm 3, \pm 4, \pm 5$. Similarly, when include current measurements into our simulator, we see 6 distinct peaks of the average current drawn. This can be seen in the following table and figures.

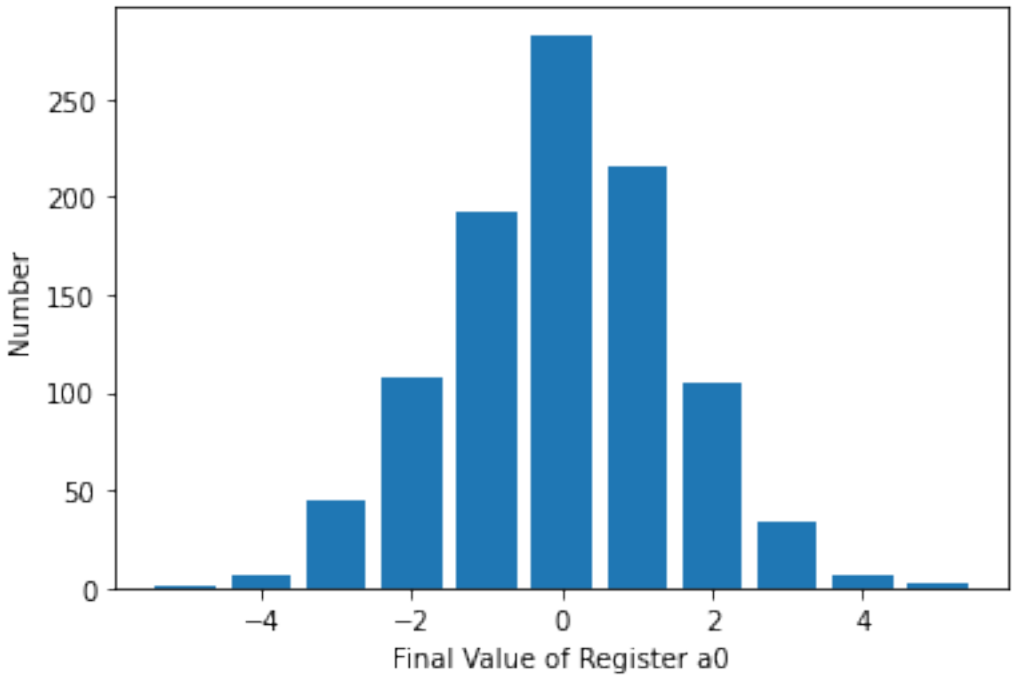


Figure 4-7: Histogram of output values for naive approach of discrete Gaussian sampler.

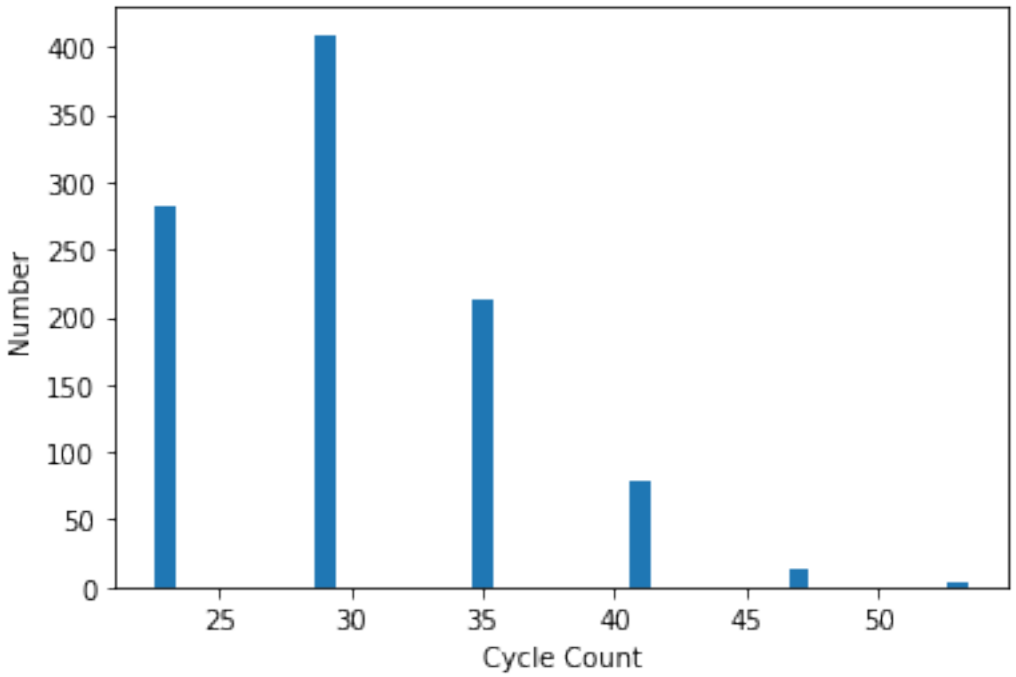


Figure 4-8: Histogram of cycle count for naive discrete Gaussian sampler.

Output Class	Cycle Count	Average Current (μA)
0	23	2037.83
± 1	29	2088.78
± 2	35	2122.25
± 3	41	2145.94
± 4	47	2163.57
± 5	53	2177.21

Table 4.3: Cycle and current values for each class of outputs, using naive discrete Gaussian sampler.

Because both the average current and the timing show distinct peaks, this means both are leaking information.

Since this current is proportional to the power consumption, by simply observing power consumption, an attacker can narrow down the output to 2 value in the worst case (one of the 5 classes $\pm 1, \pm 2, \pm 3, \pm 4, \pm 5$), and 1 value in the best case (if it were the peak corresponding to output 0).

Similarly, an attacker can measure the time it takes to for the function to complete, and from the cycle graph and the period per cycle, narrow down the output to 1 or 2 possible values out of all 11 possibilities.

Thus, with our simulator, we've exposed the potential side channel attack - both in timing and in power analysis - associated with this naive approach.

4.3.2 Modified Discrete Gaussian Sampler

Consider the following modified code, designed to ensure the same number of cycles and instructions are called regardless of the input seed or output. Instead of breaking out of the for loop - which causes the execution timing to change - we design a special shift instruction, and then ensure this same instruction is executed for CDF_TABLE_LEN iterations regardless of the output value of `sample`.

We can verify that the output register `a0` produces the expected Gaussian distribution. This can be seen in the histogram of the final value in register `a0`.

Using our simulator, we observe that both the average current (and thus power

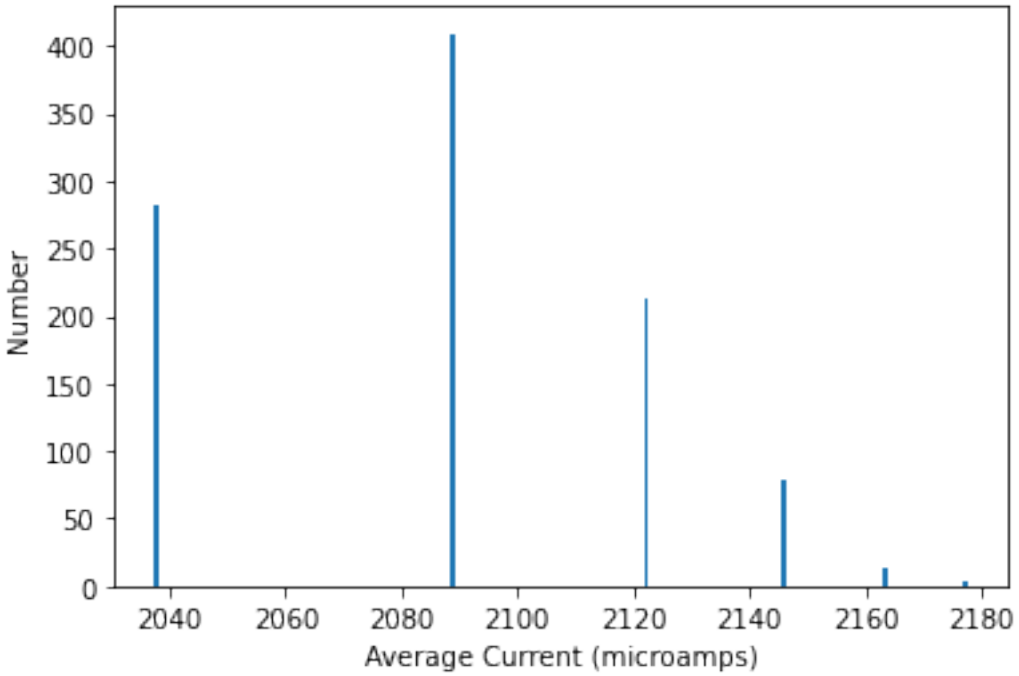


Figure 4-9: Histogram of average current values for naive Gaussian sampler.

```
for (i = 0; i < CDF_TABLE_LEN; i++)
{
    sample += (unsigned int)(CDF_TABLE[i] - prnd) >> 31;
}
s = ((-sign) ^ sample) + sign;
return s;
```

Figure 4-10: C code for the modified discrete Gaussian sampler.

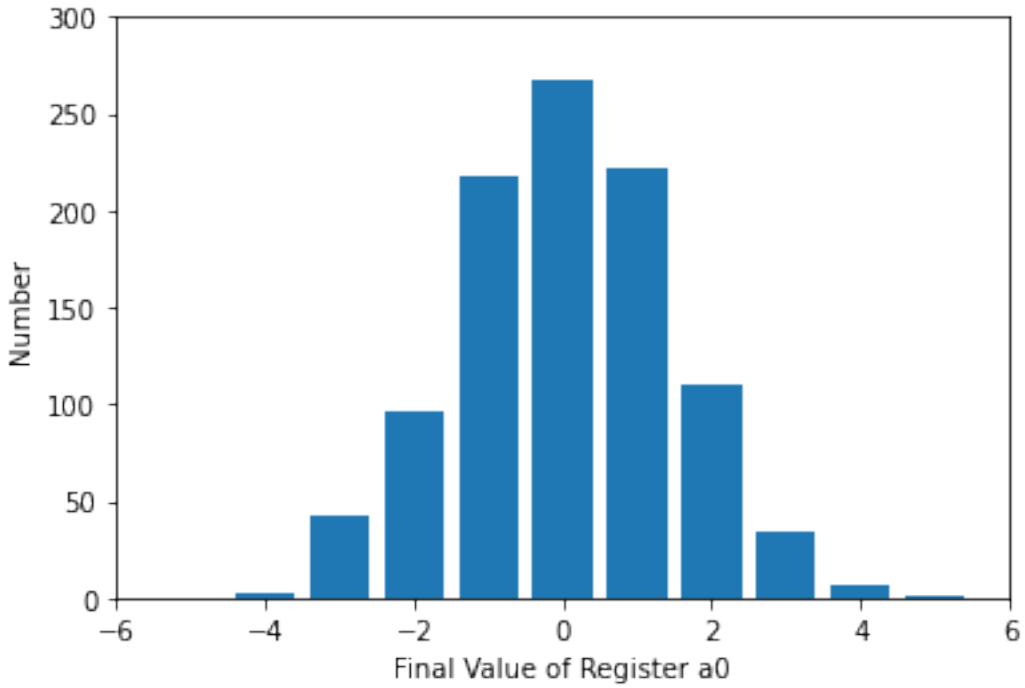


Figure 4-11: Output value distribution of modified discrete Gaussian sampler.

consumption) and cycle count are indistinguishable, regardless of the output value. In the figures below, we see a single peak, rather than several discrete peaks. This is consistent with the fact that the approach is designed to be safe from power and timing attacks at the instruction level. Thus, our simulator confirms the improvements from this modified Gaussian sampler.

Output Class	Cycle Count	Average Current (μA)
$0, \pm1, \pm2, \pm3, \pm4, \pm5$	68	2016.24

Table 4.4: Cycle count and average current for all outputs, using modified Gaussian sampler.

4.4 Limitations

Although all peaks generated by the simulator are highly discrete and clear-cut, in reality, small variations depending on different reg values can still occur. Hence, if we were to experimentally collect the same data using hardware, we'd expect the distributions to be less discrete.

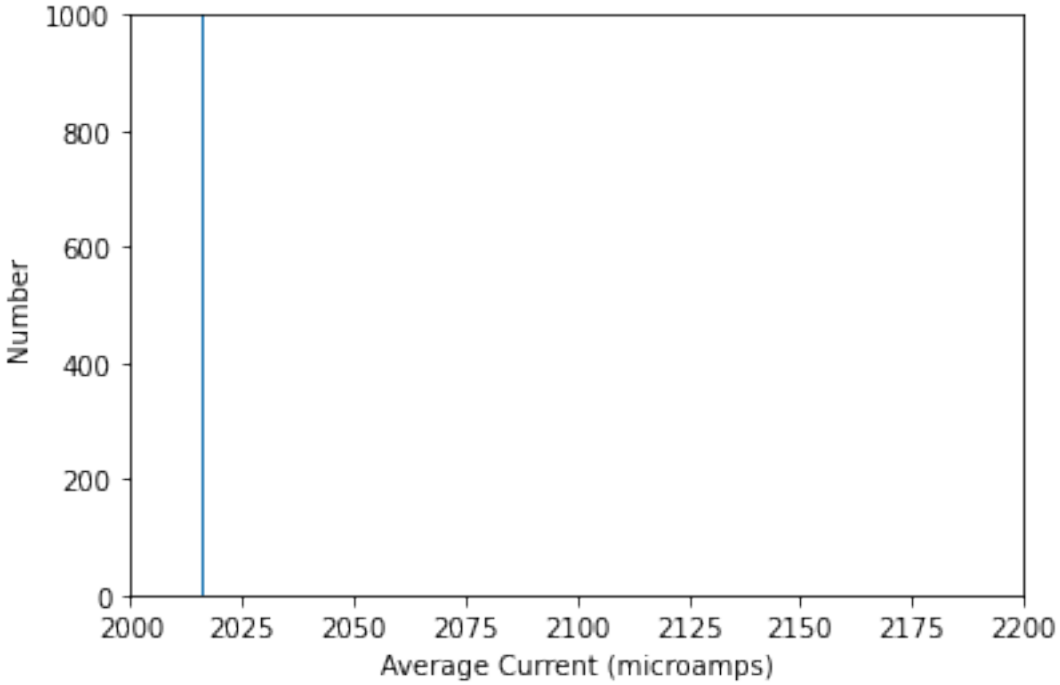


Figure 4-12: Histogram of average current for the modified Gaussian sampler.

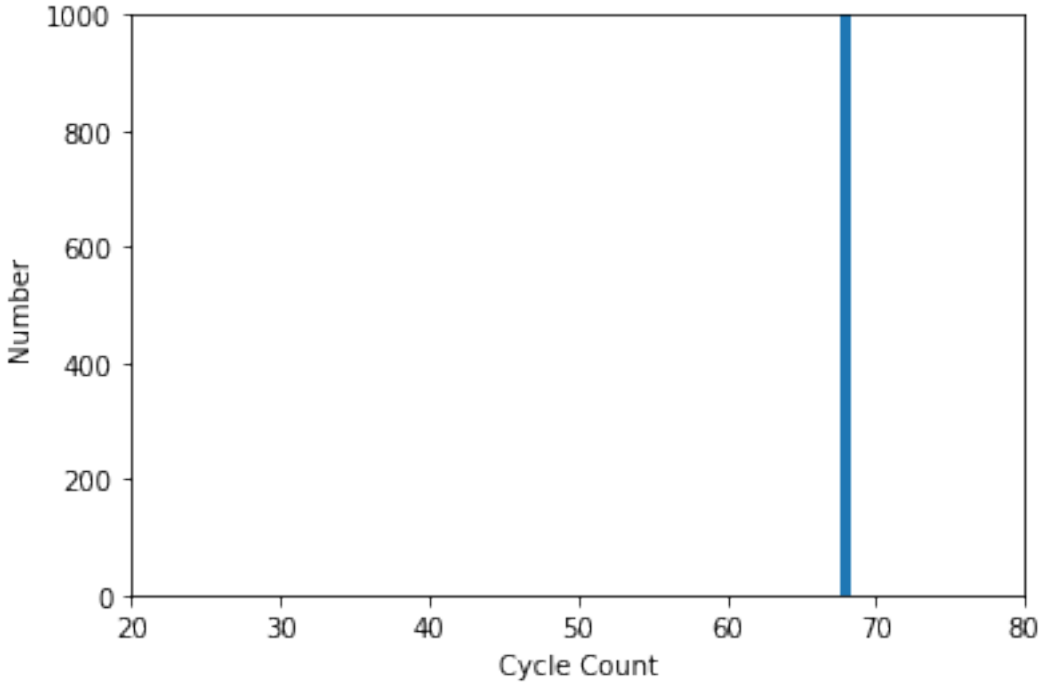


Figure 4-13: Histogram of cycle count for modified Gaussian sampler.

Furthermore, our simulator is not intended to be a fool-proof guarantee that an approach is safe from side channel attacks, because many other factors and variations emerge when the same code is implemented in hardware. However, our simulator does combine the ease of simulating quickly without requiring a hardware setup, and the instruction-level detail that can expose side channel vulnerabilities to a limited extent.

Chapter 5

Conclusion and Future Work

Being able to estimate power consumption and detect simple power analysis and timing-based side-channel leakage from assembly code is important for efficient and secure software design. We have designed a Python-based instruction level power consumption simulator for 32-bit RISC-V micro-processor that helps facilitate power and cycle count estimation and also preliminary side-channel analysis, specifically at the level of simple power analysis and side channel attacks that are related to the sequence (timing) of instructions.

To achieve this, we performed hardware measurements for all 45 non-privileged instructions in the RISC-V instruction set architecture. For each instruction, Python templates were used to generate C code, which was then converted to RISC-V assembly. Each instruction's current was measured by averaging repeated measurements in order to reduce effects of noise.

After obtaining data for each instruction, the simulator can execute a given RISC-V assembly code and output a simulated waveform over all time steps. We have evaluated using an example pseudo-random function and discovered that even this simulated waveform exposes potential ways for side-channel attacks, such as estimating the relative size of an input value based on the anomalous cycles in the waveform, or exposing the output value of a discrete Gaussian sampler. Possible applications of this simulator are listed below:

- Estimate power consumption for any input RISC-V assembly code,
- Preliminary detection of simple power analysis (SPA) side-channel leakage,
- Detection of timing-based side-channel leakage,
- Modelling for software design, memory usage, and helping debug RISC-V code.

Although our simulator is able to uncover simple side-channel attacks, many complex attacks are also possible, such as differential power analysis which involves using statistical analysis to uncover secret operand values. Our simulator is not yet feasible for this purpose, because we only consider the average power consumption per instruction and do not include the effect that different input operands have on the power consumption. In practice, power consumption differs for each occurrence of the same instruction due to inter-instruction dependencies [11].

Indeed, differential power analysis requires knowledge of how each instruction changes with different register and operand values. To extend our simulator, we would require more exhaustive data collection for each operand value (rather than averaging over many random operand values), and determine the power consumption of each instruction as a function of these values. Due to time and physical constraints, this was not included in our current simulator, and this can be explored in future work.

We also do not support the effect of using different operands, and can only model simple timing attacks. Future extensions could expand on correlation power, using statistical measurements to determine dependencies on input data variations.

There are many other possible extensions of the RISC-V simulator. For instance, we might learn whether data is leaked through the first or second operator in the instruction, or through transitioning from one instruction to the next [11]. With enough data, we can also imagine a machine learning model that generates an estimated power waveform from a given sequence of RISC-V instructions. These extensions will require collecting and processing significantly larger amounts of data, and hence this is left as future work.

Appendix A

Code for Data Collection

A.1 Example main.c File

Below is an example main.c file generated using the instruction "beq".

```
// Test RV32IM

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "interrupt.h"
#include "printk.h"
#include "rtc.h"

unsigned long random (unsigned long x)
{
    return ((1103515245 * x + 12345) % 2147483648);
}

volatile unsigned long * const lwe_clk_gate = (unsigned long *) 0xA00AF004;
```

```
volatile unsigned long * const aes_clk_gate = (unsigned long *) 0xA0000034;
volatile unsigned long * const chacha_clk_gate  = (unsigned long *) 0xA0010044;
volatile unsigned long * const trivium_clk_gate = (unsigned long *) 0xA0020034;
volatile unsigned long * const gpio_ddr     = (unsigned long *) 0xAAAB0000;
volatile unsigned long * const gpio_port    = (unsigned long *) 0xAAAB0004;

int main ( void )
{
    unsigned int i;
    unsigned long out;

    // PRNG Seed
    unsigned long randx = 80817752;

    *lwe_clk_gate = 1;
    *aes_clk_gate = 1;
    *chacha_clk_gate = 1;
    *trivium_clk_gate = 1;

    *gpio_ddr  = 0x0000FFFF;
    *gpio_port = 0x000000F0;

    // Clear Data-Path

    asm volatile ("li a0, 0" : : : "a0");
    asm volatile ("li a1, 0" : : : "a1");
    asm volatile ("add a2, a0, a1;" : : : "a2");
    asm volatile ("sub a2, a0, a1;" : : : "a2");
    asm volatile ("mul a2, a0, a1;" : : : "a2");
    asm volatile ("and a2, a0, a1;" : : : "a2");
```

```
    asm volatile ("or a2, a0, a1;" : : : "a2");
    asm volatile ("xor a2, a0, a1;" : : : "a2");
    asm volatile ("sll a2, a0, a1;" : : : "a2");
    asm volatile ("srl a2, a0, a1;" : : : "a2");

    // Power Measurement Start
    *gpio_port = 0x00000000;
    *gpio_port = 0x00008000;
    *gpio_port = 0x00000000;

    for (i = 0; i < 1000; i++)
    {
        // Initialize Variables / Registers Start

        // Test Code Start

        asm volatile ("mv a0, 0"   : : : "a0");
        asm volatile ("mv a1, 1"   : : : "a1");
// Initialize Variables / Registers End
        asm volatile ("beq a0, a0, 4;" : : : "a0");
        asm volatile ("beq a0, a0, 4;" : : : "a0");
        asm volatile ("beq a0, a0, 4;" : : : "a0");
        ... // Omitted for length, this repeats 7500 times (hardcoded)

        asm volatile ("beq a0, a0, 4;" : : : "a0");

        asm volatile ("addi  x0, x0, 0;" : : : "a0");
        // Test Code End
    }
```

```
    *gpio_port = 0x00000AAA;
    *gpio_port = 0x00008AAA;
    *gpio_port = 0x00000AAA;

    // Power Measurement End
}
```

A.2 Python Code to Generate main.c files

```
#! /usr/bin/python
import sys
import random
#1000 outer loop, 7500
def gen_main(command, outer_loop, inner_loop, instr="REG", immediate=None):

    target = open('main.c', 'w')

    ################
    ## write headers
    ################

    target.write("""// Test RV32IM

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "interrupt.h"
#include "printk.h"
#include "rtc.h"
```

```
unsigned long random (unsigned long x)
{
    return ((1103515245 * x + 12345) % 2147483648);
}

volatile unsigned long * const lwe_clk_gate       = (unsigned long *) 0xA00AF004;
volatile unsigned long * const aes_clk_gate       = (unsigned long *) 0xA0000034;
volatile unsigned long * const chacha_clk_gate    = (unsigned long *) 0xA0010044;
volatile unsigned long * const trivium_clk_gate   = (unsigned long *) 0xA0020034;
volatile unsigned long * const gpio_ddr           = (unsigned long *) 0xAAAB0000;
volatile unsigned long * const gpio_port          = (unsigned long *) 0xAAAB0004;
""")


    target.write("""
int main ( void )
{
    unsigned int i;
    unsigned long out;

    // PRNG Seed
    unsigned long randx = 80817752;

    *lwe_clk_gate = 1;
    *aes_clk_gate = 1;
    *chacha_clk_gate = 1;
    *trivium_clk_gate = 1;

    *gpio_ddr  = 0x0000FFFF;
```

```
*gpio_port = 0x000000F0;

// Clear Data-Path

asm volatile ("li a0, 0" : : : "a0");
asm volatile ("li a1, 0" : : : "a1");
asm volatile ("add a2, a0, a1;" : : : "a2");
asm volatile ("sub a2, a0, a1;" : : : "a2");
asm volatile ("mul a2, a0, a1;" : : : "a2");
asm volatile ("and a2, a0, a1;" : : : "a2");
asm volatile ("or a2, a0, a1;" : : : "a2");
asm volatile ("xor a2, a0, a1;" : : : "a2");
asm volatile ("sll a2, a0, a1;" : : : "a2");
asm volatile ("srl a2, a0, a1;" : : : "a2");

// Clear Data-Path""")
if (instr=="LD") or (instr == "ST"):
    target.write("""

// move the value of the base address into a1
asm volatile("li a1, 0x8000;":::"a1");
""")
if instr=="LD":
    target.write("""
randx = random(randx); //pick a random within 2^11 to save
//move the random value into register a2
asm volatile ("mv a2, %0;"   : :"r"(randx) : "a2");
//store a2 at the address in a1""")

    if command == "lw":
```

```
            for i in range(inner_loop):
                target.write("""
    asm volatile("sw a2, """ + str(i*4) + """(a1);":::);""")

        if command == "lh":
            for i in range(inner_loop):
                target.write("""
    asm volatile("sh a2, """ + str(i*2) + """(a1);":::);""")

        if command == "lb":
            for i in range(inner_loop):
                target.write("""
    asm volatile("sb a2, """ + str(i) + """(a1);":::);""")


    #logic to deal with outer_loop, for loads and stores:
    #Note, the .csv file is where we input the
    #values for outer loop. so we expect those
    #to be calculated/ confirmed in CSV.

    target.write("""// Power Measurement Start

    *gpio_port = 0x00000000;
    *gpio_port = 0x00008000;
    *gpio_port = 0x00000000;

    for (i = 0; i < """ + str(outer_loop) + """; i++)
    {
        // Initialize Variables / Registers Start
"""")
```

```
    if instr=="REG":
     # set a0, a1 to different things
        target.write("""
        randx = random(randx);
asm volatile ("mv a0, %0"   : : "r" (randx) : "a0");
        randx = random(randx);
asm volatile ("mv a1, %0"  : : "r" (randx) : "a1");
// Initialize Variables / Registers End""")

    elif (instr == "IMM") or (instr == "SHAMT") or (instr == "ST"):
     # only use a0, assuming other is immediate
     # branch sets its own values
        target.write("""
        randx = random(randx);
asm volatile ("mv a0, %0"   : : "r" (randx) : "a0");
// Initialize Variables / Registers End""")

    target.write("""
        // Test Code Start
 """)

    if instr=="REG":
        for i in range(inner_loop):
            target.write("""
asm volatile (\""""" + command + """  a2, a0, a1;" : : : "a2");""")

    elif instr=="IMM":
        for i in range(inner_loop):
            target.write("""
```

```
        asm volatile (\""" + command + """  a2, a0, """
            + str(random.randint(0, 2**12-1)) + """;" : : : "a2");""")

    elif instr=="SHAMT":
        for i in range(inner_loop):
            target.write("""
        asm volatile (\""" + command + """  a2, a0,"""
            + str(random.randint(0, 2**5-1)) + """;" : : : "a2");""")

    elif instr=="BRANCH":
     # set a0 and a1 accordingly
        target.write("""
        asm volatile ("mv a0, 0"   : : : "a0");
        asm volatile ("mv a1, 1"   : : : "a1");
// Initialize Variables / Registers End""")
        assembly_line = ""

        if command == "beq":
            assembly_line = """
        asm volatile ("beq a0, a0, 4;" : : : "a0");"""
        # 4 is the offset, which means move to next instruction (PC+4)

        if command == "bne":
            assembly_line = """
        asm volatile ("bne a0, a1, 4;" : : : "a0");"""

        if command == "blt":
            assembly_line = """
        asm volatile ("blt a0, a1, 4;" : : : "a0");"""
```

```
    if command == "bge":
        assembly_line = """
    asm volatile ("bge a1, a0, 4;" : : : "a0");"""

    for i in range(inner_loop):
        target.write(assembly_line)

    target.write("""

    asm volatile ("addi  x0, x0, 0;" : : : "a0");""")
    # equivalent to NOP pseudoinstruction, see manual

elif instr == "LD":
    if command == "lw":
        for i in range(inner_loop):
            target.write("""
    asm volatile ("lw a2, """ + str(i*4)+""""(a1);"   : : : );""")
            #4092
            #if lw, outer_loop *8
    if command == "lh":
        for i in range(inner_loop):
            target.write("""
    asm volatile ("lh a2, """ + str(i*2)+""""(a1);"   : : : );""")
            #4094
            #if lh, outer loop *4
    if command == "lb":
        for i in range(inner_loop):
            target.write("""
    asm volatile ("lb a2, """ + str(i)+""""(a1);"   : : : );""")
            #4095
```

```
    # if lb, outer loop * 2.
    #reason is because only 4092, but we want 7000, 1000
    #so total 7000*1000
    #previously, inner was 7000. But now inner is 4000. so outer needs
    #to double. This is because IMM only has 12 bits, so largest 4095.
elif instr == "ST":
    if command == "sw":
        for i in range(inner_loop):
            target.write("""
asm volatile("sw a0, """ + str(i*4) + """(a1);":::);""")

    if command == "sh":
        for i in range(inner_loop):
            target.write("""
asm volatile("sh a0, """ + str(i*2) + """(a1);":::);""")

    if command == "sb":
        for i in range(inner_loop):
            target.write("""
asm volatile("sb a0, """ + str(i) + """(a1);":::);""")

else:
    raise SystemExit("INVALID INSTRUCTION TYPE.")

target.write("""
    // Test Code End
}

*gpio_port = 0x00000AAA;
*gpio_port = 0x00008AAA;
```

```
    *gpio_port = 0x00000AAA;

    // Power Measurement End
}""")

    target.close()


if __name__ == "__main__":
    print sys.argv
    command = sys.argv[1]
    command = command.lower()
    outer_loop = int(sys.argv[2])
    inner_loop = int(sys.argv[3])

    instr = "REG"
    if len(sys.argv) > 4: #has index 4, length 5
        instr = (sys.argv[4]).strip()
        instr = instr.upper()

    gen_main(command, outer_loop, inner_loop, instr=instr)
```

Appendix B

Code for RISC-V Simulator

B.1 Preprocessing Assembly Code

```
# prefixes all the hex numbers in code2.txt with '0x'.
# also changes all the absolute addresses to current addresses
def clean_code():
    f = open("code.txt", "r")
    lines = f.read().split("\n")
    f.close()
    g = open("code_cleaned_rel_address.txt", "w")
    line_pc = 0

    for nextline in lines:
        if nextline.strip() == "":
            continue
            # line_pc is not incremented.
        inst = nextline.split() [0]
        comma_split_line = nextline.split(",")
        current_instruction = inst.strip().lower()
        if (current_instruction in ["jal", "jalr", "beq", "bne", "blt",
            "bltu", "bgt", "auipc", "lui"]):
```

```
            num = comma_split_line[-1] # this will never be negative.
            if num[0:2] != "0x":
                num = "0x" + num # append hex

            relative_address = num
            # also subtract current pc from number
            # note: only for jumps/branches.

            # note: inconsistency in dumpfile, "jalr" uses relative
            if current_instruction in ["jal", "beq", "bne",
                "blt", "bltu", "bgt"]:
                absolute_address = int(num, 16)
                current_address = line_pc
                relative_address = hex(absolute_address - current_address)

            line = ",".join(comma_split_line[0:-1] + [relative_address] )
        else: # don't change if not special instruction
            line = nextline
        print(line)
        g.write(line + "\n")

        line_pc += 4
    g.close()
```

B.2 Preprocessing Memory Inputs

```
# prefixes all the hex numbers in code2.txt with '0x'.
f = open("mem.txt", "r")
lines = f.read().split("\n")
```

```
f.close()
g = open("mem_cleaned.txt", "w")
for nextline in lines:
    if nextline.strip() == "":
        continue
    addr, value = nextline.split()
    line = "0x" + addr + "\t 0x" + value
    print(line)
    g.write(line + "\n")
g.close()
```

B.3 Commandline Wrapper for RISC-V Simulator

```
#!/usr/bin/env python
import sys
sys.path.insert(1, '../py-riscv-package/py_riscv')
import riscv_compiler
import random
import os

# disable/enable print
default_stdout = sys.stdout
def disable_print():
    sys.stdout = open(os.devnull, 'w')
def enable_print():
    if (sys.stdout != default_stdout):
        sys.stdout.close()
        sys.stdout = default_stdout

# specify number of iterations.
```

```
# specify which memory locations will be random (list format).
# mem locations need to be hex strings.

def repeated_runs(iters, random_mem_locations, code_filepath,
    power_filepath, output_filepath):
    enable_print()
    for i in range(iters):
        print("====== Iteration " + str(i+1) + " of " + str(iters) + " ======")
        # create mem file with random vals, name it "mem_iter_i.txt"
        with open(os.path.join(output_filepath,
            "mem_iter_" + str(i) + ".txt"), "w") as f:
            for loc_index in range(len(random_mem_locations)):
                random_number = random.randint(0, 2**32-1) # 32 bit
                line = str(random_mem_locations[loc_index])
                    + "\t" + str(random_number) + "\n"
                f.write(line)
            f.close()

        disable_print()
        riscv_instance = riscv_compiler.RISCV(code_filepath,
            os.path.join(output_filepath, "mem_iter_" + str(i) + ".txt"))
        riscv_instance.use_power_data(power_filepath)
        riscv_instance.run(plot=True)

        power_output = riscv_instance.get_power_consumption_list()

        # save in output_power_iter_i.txt
        with open(os.path.join(output_filepath,
            "output_power_iter_" + str(i) + ".txt"), "w") as outfile:
            for j in range(len(power_output)):
```

```
                inst, pwr = power_output[j]
                line = str(inst) + "\t" + str(pwr) + "\n"
                outfile.write(line)
            outfile.close();
        enable_print()
        print("Output written to", os.path.join(output_filepath,
            "output_power_iter_" + str(i) + ".txt"))

        print(riscv_instance.registers)
        # registers store hex values.

# example command line run
# python3 command_line_py_riscv.py memlistfile=rand_mem_locations.txt
#   codefile=code.txt powerfile=pwr.txt outfolder=out_folder
if __name__ == "__main__":
    arg_dict = dict(arg.split('=') for arg in sys.argv[1:])
    iters = arg_dict.get("iters", 1) #default to 1 iter
    try:
        memlistfile = arg_dict["memlistfile"]
    except:
        print("Please provide memlistfile.")
        sys.exit()
    try:
        powerfile = arg_dict["powerfile"]
    except:
        print("Please provide powerfile.")
        sys.exit()
    try:
        codefile = arg_dict["codefile"]
    except:
```

```
        print("Please provide codefile.")
        sys.exit()

    outfolder = arg_dict.get("outfolder", "")
    iters = int(iters) # default as string

    print("Using iters:", iters)
    print("Using randomizing the memory addresses listed in file:", memlistfile)
    print("Using codefile:", codefile)
    print("Using power file:", powerfile)
    print("Using out folder:", outfolder)

    # expects random memory addresses listed with whitespaces
    with open(memlistfile, "r") as m:
        mem_list = m.read().split()
        m.close()
    repeated_runs(iters, mem_list, codefile, powerfile, outfolder)
```

B.4 Register Name File (reg_name.txt)

This file lists all equivalent register names and standardizes them.

```
zero x0
ra x1
sp x2
gp x3
tp x4
t0 x5
t1 x6
t2 x7
s0 x8
```

```
fp x8
s1 x9
a0 x10
a1 x11
a2 x12
a3 x13
a4 x14
a5 x15
a6 x16
a7 x17
s2 x18
s3 x19
s4 x20
s5 x21
s6 x22
s7 x23
s8 x24
s9 x25
s10 x26
s11 x27
t3 x28
t4 x29
t5 x30
t6 x31
x0 x0
x1 x1
x2 x2
x3 x3
x4 x4
x5 x5
```

```
x6 x6
x7 x7
x8 x8
x9 x9
x10 x10
x11 x11
x12 x12
x13 x13
x14 x14
x15 x15
x16 x16
x17 x17
x18 x18
x19 x19
x20 x20
x21 x21
x22 x22
x23 x23
x24 x24
x25 x25
x26 x26
x27 x27
x28 x28
x29 x29
x30 x30
x31 x31
```

B.5 Python based RISC-V Simulator

```
import re
```

```
import matplotlib.pyplot as plt

# nov 10 updates:
# supports verbose
# supports negative hex: -0x1 format
# hex helper function checks if input integer - casts to int first in case
if string
class RISCV:
    def __init__(self, codefilepath, memfilepath, verbose=False):
        self.codefilepath = codefilepath
        self.steppable = True
        self.ended = False #0 if reaches wfi
        self.memfilepath = memfilepath
        self.memfile = MemFile()
        self.memfile.load(memfilepath)
        self.registers = {} # x1 thru x31, x0 is 0
        self.cycle_counter = 0
        self.verbose = verbose

        # initializes the alternate register names
        self.reg_name = {}
        self._load_reg_names()

        assert self.codefilepath != memfilepath, "Memory and code should
have different filepaths."

        codefilelines = open(codefilepath, 'r').readlines()
        self.codefilelines = []
        for line in codefilelines:
            if line.strip() == "":
```

```
                continue
            self.codefilelines.append(line)
        ## we want to clean this to ensure no blank lines, for line counting
purpose

        self.pc = 0
        self.instruction_power_dict = {}
        self.previous_inst = ""
        self.inst_line_power_list = [] #(inst, power) for plotting later
                            # history of instructions run
        self.plot = False

        self.REG = ['sll', 'srl', 'sra', 'xor', 'add', 'sub', 'mul', 'mulhu',
'mulh', 'mulhsu', 'div', 'divu', 'rem', 'remu', 'and', 'or', "sltu",
                    'slt']
                # slt, divu, remu, mulhu,| mulh, mulhsu
        self.IMM = ['slli', 'srli', 'srai', 'xori', 'addi', 'andi', 'ori',
'sltiu', 'slti'] #TODO: subi is not an inst.
                # slti

        self.BRANCH = ['beq', 'bne', 'blt', 'bge', 'bltu', 'bleu',
                       "bgeu", "bgt", "bgtu", "ble"] #just beq, bne, blt,
bge, bltu, bgeu. other ones are pseudo substitutable.
        self.LOAD = ['lw', 'lh', 'lb', 'lbu', 'lhu'] #lbu, lhu
        self.STORE = ['sw', 'sh', 'sb']
        self.JUMP = ['jal', 'jalr']
        self.PC_INST = ['auipc', 'lui']


        #TO CHECK: added auipc, lui
```

```
        # is the immediate value given ALREADY left shifted by 12 bits?
or do we need to do shift?
        # do we add current PC or next +4 pc?
        # i assume it takes the imm, left shifts, and adds current pc. (or
zero if lui)

    def get_reg_name(self, reg):
        """Interprets a string reg name as one of x0 through x31.
        for example, a0 is x10."""

        # if reg is already x0, through x31, it will not find it and simply
return itself.
        # if reg is invalid, it will throw an error.
        try:
            return self.reg_name[reg.strip().lower()]
        except:
            raise SyntaxError("Invalid register: " + reg)

    def _load_reg_names(self, reg_name_filepath="reg_name.txt"):
        """Loads file that tells us mapping of register names.
        For example file would have a0 x10\na1 x11 ..."""
        with open(reg_name_filepath, 'r') as f:
            name_list = f.readlines()
        for line in name_list:
            if line.strip() == "":
                continue
            name, xname = line.split()
            # make sure value is 32 bits
            # reads the '0x0' string into binary, truncates last 32,
            # then puts back into integer.
```

```
            self.reg_name[name] = xname

### functions for plotting power ###
    def use_power_data(self, filepath, func=lambda x: x):
        """ Uses power data specified by filepath.
        filepath: string filepath with format 'instruction   power\ninst2
 power'
        func: function to apply to numerical power value (for example,
        if we want to convert current to power)
        """
        # reads file and updates self.instruction_power_dict
        with open(filepath, 'r') as f:
            inst_power_list = f.readlines()
        for line in inst_power_list:
            if line.strip() == "":
                continue
            inst, power = line.split()
            inst = inst.lower()
            power = float(power)

            # apply function on x. by default, function returns self.
            # this is useful in case we have e.g. current data instead.
            self.instruction_power_dict[inst] = func(power)

    def get_power_for_instruction(self, inst):
        """ Gets numerical power value associated with instruction (e.g.
'add').
        Returns None (which will plot blank) if instruction's power is unspecified.
        """
        return self.instruction_power_dict.get(inst.lower(), None)
```

```
    def _add_point(self, inst, power_value):
        """ Adds instruction and corresponding power value to plot.
        inst: string instruction
        power_value: number for power
        """
        self.inst_line_power_list.append((inst, power_value))

    def get_power_consumption_list(self):
        """ Returns the current list of (instruction, power) in
        timestep order."""

        return self.inst_line_power_list

    def show_power_plot(self, minstep=None, maxstep=None, minpower=None,
maxpower=None):
        """ Shows current power plot of all instructions.
        If plotting is not turned on, it will be blank.
        "None" will be shown as gap in plot.
        minstep, maxstep specifies the range (x) of steps to plot.
        minpower, maxpower specifies the range (y) of power to plot.
        """
        assert len(self.inst_line_power_list) != 0, "Nothing to plot. (Did
you use run(plot=True)? )"

        # note this is albe to plot NONE points (just gap in plot)
        inst_list, power_list = zip(*self.inst_line_power_list)

        steps = list(range(len(inst_list)))
        plt.plot(steps, power_list, 'b-') # blue line graph
```

```
        # we need to use steps otherwise plt will autogenerate x axis,
        # making it hard to set minstep

        xlabels = [str(inst) + '\n' + str(step) for inst,step in zip(inst_list,
steps)]
        plt.xticks(steps, xlabels) # labels x axis with instructions

        plt.ylabel("Power")
        plt.title("Power Consumption Over Instruction Time Steps")

        # set axis range
        plt.axis(xmin=minstep, xmax=maxstep, ymin=minpower, ymax=maxpower)
        plt.show()

### functions for running ###
    def check_steppable(self):
        """Checks if code is steppable (pc is between 0 and end of file)."""
        self.steppable = (0 <= self.pc/4 < len(self.codefilelines)) and
(not self.ended)
        return self.steppable

    def run(self, max_steps=None, plot=False, verbose=False):
        """Runs code until completion, or until specified max_steps steps.
        If plot=True, power values will be saved."""
        self.verbose = verbose
        self.plot = plot
        if self.plot:
            assert self.instruction_power_dict != {}, "To plot, please specify
power data with use_power_data()."
```

```
    # can specify max limit number of inst to run
    counter = 0
    if max_steps is None:
        while self.check_steppable():
            if self.cycle_counter % 1000000 == 0:
                print(self.cycle_counter)
            # execute stuff
            self.step(verbose=verbose)
            counter += 1
        print("Done. End of code file.")
    else:
        for i in range(max_steps):
            if self.check_steppable():
                self.step(verbose=verbose)
                counter += 1
            else:
                print("Done. End of code file.")
                break
        #print("Executed", str(counter), "steps.")
    return

def step(self, verbose=False):
    """Executes one step (next instruction)."""
    self.verbose = verbose
    assert self.pc%4 == 0 # pc is always 4*
    if self.verbose:
        print("--Current pc:", self.pc)
    instruction = self.codefilelines[self.pc//4]
    prev_pc = self.pc
```

```
        try:
            new_pc = self.parse_and_exec(instruction)
            # note that when wfi is reached, it cannot step. pc stays same.
        except:
            print("error stepping - cannot parse_and_exec")
            self._raise_error()

        return True

    def parse_and_exec(self, inst_string):
        """Parses and executes instruction based on string."""
        old_pc = self.pc # used for debugging (prints error)
        inst_params = self._match_instruction(inst_string) # could be length
2, 3, etc list
        if self.verbose:
            print("\n> Executing instruction:", inst_string, inst_params)

        inst = inst_params[0].strip() #string of the instruction itself

        #########
        if (inst == 'wfi'):
            self.ended = True
            return self.pc # not change pc
        #########

        # Note pc is updated after any of the following.
        if inst in self.REG:
            if self.verbose:
                print("--REG")
            dest = self.get_reg_name(inst_params[1])
```

```
        rs1 = self.get_reg_name(inst_params[2])
        rs2 = self.get_reg_name(inst_params[3])
        self.reg(inst_params[0], dest, rs1, rs2)

    elif inst in self.IMM:
        if self.verbose:
            print("--IMM")
        dest = self.get_reg_name(inst_params[1])
        rs1 = self.get_reg_name(inst_params[2])
        # parse logic in case hex or int input
        const = HelperFunctions.int_or_hex_to_int(inst_params[3])

        self.imm(inst_params[0], dest, rs1, const)

    elif inst in self.BRANCH:
        if self.verbose:
            print("--BRANCH")
        # parse logic in case hex or int
        offset = HelperFunctions.int_or_hex_to_int(inst_params[3])
        rs1 = self.get_reg_name(inst_params[1])
        rs2 = self.get_reg_name(inst_params[2])
        self.branch(inst_params[0], rs1, rs2, offset)



    elif inst in self.LOAD:
        if self.verbose:
            print("--LOAD")
        # parse logic in case hex or int
        offset = HelperFunctions.int_or_hex_to_int(inst_params[2])
        xto = self.get_reg_name(inst_params[1])
```

```
            xmem = self.get_reg_name(inst_params[3])
            self.load(inst_params[0], xto, offset, xmem)


        elif inst in self.STORE:
            if self.verbose:
                print("--STORE")
            # parse logic in case hex or int
            offset = HelperFunctions.int_or_hex_to_int(inst_params[2])
            xval = self.get_reg_name(inst_params[1])
            xmem = self.get_reg_name(inst_params[3])
            self.store(inst_params[0], xval, offset, xmem)

        elif inst in self.JUMP:
            if self.verbose:
                print("--JUMP")
                print(inst)
            # parse logic in case hex or int
            # jal and jalr different:
            if inst == "jal":
                #jal rd, immJ
                rd = self.get_reg_name(inst_params[1]) #params0 is instruction
itself
                immj = HelperFunctions.int_or_hex_to_int(inst_params[2])
                # call self.jump
                self.jump_jal(inst, rd, immj)
            elif inst == "jalr":
                # jalr rd, rs1, immI
                rd = self.get_reg_name(inst_params[1])
                rs1 = self.get_reg_name(inst_params[2])
```

```
                immj = HelperFunctions.int_or_hex_to_int(inst_params[3])
                self.jump_jalr(inst, rd, rs1, immj)
                # TODO check this works

        elif inst in self.PC_INST:
            if self.verbose:
                print("--PC_INST")
                print(inst)
            dest = self.get_reg_name(inst_params[1])
            # parse logic in case hex or int input
            const = HelperFunctions.int_or_hex_to_int(inst_params[2])
            self.pc_inst(inst_params[0], dest, const)

        else:
            # design question - should this reset() or not?
            # double check if the behavior of this is equivalent to the
commented one.
            print("error inside parse_and_exec")
            self._raise_error()
            # raise SyntaxError('Invalid instruction on line '+ str(old_pc//4
+ 1),
            #                   (self.codefilepath, old_pc//4 + 1, 0, inst_string))

        # increment cycle counter
        if inst == "div":
            self.cycle_counter += 32
        else:
            self.cycle_counter += 1

        # if plotting turned on, add point in power plot
```

```
        if self.plot:
            if inst != "div":
                self._add_point(inst, self.get_power_for_instruction(inst))
            else:
                assert inst == "div"
                divpwr = self.get_power_for_instruction("div")
                for i in range(32):
                    self._add_point("div", divpwr)
            # need to check the cycle counter is working correctly.
            assert self.cycle_counter == len(self.inst_line_power_list),
"Error with cycle counter."

        return self.pc

    def _match_instruction(self, inst_string):
        """Matches single line instruction string to either of two formats.
        Returns list of parts in instruction.
        """
        string = inst_string.lower().strip()

        ############
        if (string == "wfi"):
            return ["wfi"] # no groups, just itself.
        #     sys.exit(0) didn't work bc throws error.
        ############

        word = "([^\s]+)"
        space = "[\s]+"
        maybespace = "[\s]*"
        s_word_s = maybespace + word + maybespace
```

```
        # type1: inst rs1, rs2, rs3
        # regular inst
        type1 = "^" + word + space + word + maybespace + "," + s_word_s
+ "," + s_word_s + "\$"

        # type 2: inst rs1, offset(rs2)
        # memory inst
        type2 = "^" + word + space + word + maybespace + "," + s_word_s
+ "\(" + s_word_s + "\)" + maybespace + "\$"

        # type 3: inst rs1, offset (we see this with jumps especially)
        type3 = "^" + word + space + word + maybespace + "," + s_word_s
+ "\$"

        # try to match; return groups
        regular_inst = re.search(type1, string)
        if regular_inst:
            return regular_inst.groups()
        else:
            mem_inst = re.search(type2, string)
            if mem_inst:
                return mem_inst.groups()
            else:
                jump_inst = re.search(type3, string)
                if jump_inst:
                    return jump_inst.groups()
                else:
                    print("error in _match_instruction")
                    self._raise_error()
```

```
### convenience functions ###
    def get_total_lines(self):
        """Returns total lines in code. Not necessarily total steps."""
        return len(self.codefilelines)

    def reset(self):
        """Resets registers, pc, and memory.
        This can be run after updating the codefile/memory file,
        as it will re-read their filepaths."""
        self.__init__(self.codefilepath, self.memfilepath)
        return

    def set_pc(self, value):
        """Sets current pc to value. Must be multiple of 4."""
        assert value%4 == 0
        self.pc = int(value)
        return

    def get_pc(self):
        """Returns current pc (multiple of 4)."""
        return self.pc

### functions to handle different inst ###
    def reg(self, inst, dest, xval1, xval2):
        """Executes xval1 inst xval2, saves in dest.
        inst is a string instruction.
        dest, xval1, xval2 are strings of register names.
        pc is updated.
        """
```

```
        if dest == "x0":
            self.pc += 4
            return

        reg1 = HelperFunctions.hex_string_to_int(self.registers.get(xval1,
'0x00000000'))
        reg2 = HelperFunctions.hex_string_to_int(self.registers.get(xval2,
'0x00000000'))



        # https://github.com/riscv/riscv-v-spec/issues/12
        # answer to shifting.
        # what happens if shift by negative amount?
        # what happens if we shift by > 32? will this take modulo 32?
        # what happens if we overflow the shifting for a negative number
in srai? will it become 1000000...0?
        # Is there a largest amount we can shift by? somewhere it said low
6 bits - does that mean 32 is the limit for shift amount?

        # so take low and mod 32
        # sxxW is for RV32, so they require top bit of imm to be 0
        if inst == "sll":
            # puts zeros in gaps
            # get lower 6 bits of reg2
            shift_binstring = bin(int(self.registers.get(xval2, '0x00000000'),
16))[2:][-6:]
            # note - unclear behaviour - will this mod 32?
            shift_val = int(shift_binstring, 2) %32
```

```
            # only low 6 bits of shift amount is used
            # we do not use hex_to_int because don't want truncate negatives
            result = reg1*2**(shift_val)

        elif inst == "srl":
            # shifts zeros into gaps
            #string to be shifted is converted to binary of 32 bits
            reg1_hexstring = self.registers.get(xval1, '0x00000000')
            reg1_raw_binstring = bin(int(reg1_hexstring, 16))[2:] #remove
'0b'
            reg1_binstring = '0'*(32-len(reg1_raw_binstring)) + reg1_raw_binstring
            # we do not use hex_to_int because it is off by 1 for shifting
negative nums

            # now we get shift amount
            shift_binstring = bin(int(self.registers.get(xval2, '0x00000000'),
16))[2:][-6:]
            shift_val = int(shift_binstring, 2) %32
            filler = '0'
            shifted = reg1_binstring[:-shift_val] # all bits except last
shift_val bits
            result = int('0b' + filler*(32 - len(shifted)) + shifted, 2)

        elif inst == "sra":

            reg1_hexstring = self.registers.get(xval1, '0x00000000')
            reg1_raw_binstring = bin(int(reg1_hexstring, 16))[2:] #remove
'0b'
            reg1_binstring = '0'*(32-len(reg1_raw_binstring)) + reg1_raw_binstring
            # we do not use hex_to_int because it is off by 1 for shifting
```

```
negative nums
            filler = reg1_binstring[0] # find highest bit

            # now we get shift amount
            shift_binstring = bin(int(self.registers.get(xval2, '0x00000000'),
16))[2:][-6:]
            shift_val = int(shift_binstring, 2) %32
            shifted = reg1_binstring[:-shift_val] # all bits except last
shift_val bits
            result = int('0b' + filler*(32 - len(shifted)) + shifted, 2)

        elif inst == "sltu":
            # if rs1<rs2, write 1 to rd.
            # else, write 0 to rd.
            # except, if rs1 is x0:
              # if rs2 is not 0, set rd=1 (because x0 < anything)
              # else, set rd=0.
            result = int(int(HelperFunctions.int_to_hex_string(reg1),16)
< int(int(HelperFunctions.int_to_hex_string(reg2),16)))

        elif inst == "slt":
            # if rs1<rs2, write 1 to rd.
            # else, write 0 to rd.
            result = reg1 < reg2

        elif inst == "xor":
            result = reg1^reg2
        elif inst == "add":
            result = reg1 + reg2
        elif inst == "sub":
```

```
            result = reg1 - reg2
        elif inst == "and":
            result = reg1 & reg2
        elif inst == "or":
            result = reg1 | reg2
        # todo: implement mul, div, rem
        elif inst == "mul":
            result = reg1 * reg2
        elif inst == "mulhu": # multiply, high bits, unsigned
            result = int(HelperFunctions.int_to_hex_string(reg1),16) *
int(int(HelperFunctions.int_to_hex_string(reg2),16)) //2**32
        elif inst == "mulh":
            result = reg1 * reg2 //2**32
        elif inst == "mulhsu": # multiply, high bits, signed * unsigned
            result = reg1 * int(int(HelperFunctions.int_to_hex_string(reg2),16))
//2**32

        elif inst == "div":
            # div performs signed division
            # (divu performs unsigned)
            # according to RISCV manual, dividing by 0 sets all bits to
1
            # (which is -1 in div, and 2**length-1 for divu),
            if reg2 == 0:
                result = -1
            else:
            # Additionally, the only way to overflow div (signed) is
            # -2**(32-1)/-1. Here, the result should be -2**(31).
            # this is already supported when we use hex_string_to_int etc.
                result = reg1 // reg2
```

```
elif inst == "divu":
    # unsigned division
    reg1_u = int(HelperFunctions.int_to_hex_string(reg1), 16)
    reg2_u = int(HelperFunctions.int_to_hex_string(reg2), 16)
    if reg2_u == 0:
        result = 2**32-1 #32 bit, all 1's
    else:
        result = reg1_u // reg2_u

elif inst == "remu":
    reg1_u = int(HelperFunctions.int_to_hex_string(reg1), 16)
    reg2_u = int(HelperFunctions.int_to_hex_string(reg2), 16)
    if reg2_u == 0:
        reg1_u
    else:
        result = reg1_u % reg2_u

elif inst == "rem":
    # according to RISCV manual page 44,
    # sign of remainder is sign of dividend (reg1)
    # also, x/0 gives remainder x
    # and overflow (-2**31/-1) gives remainder 0 (already supported)
    if reg2 == 0:
        result = reg1
    else:
        # ensure remainder is the right sign
        if reg1 >= 0:
            result = reg1 % reg2
        else:
```

```
                result = reg1 % reg2 - reg2
        else:
            print("Unsupported reg instruction.")
            self._raise_error()

        result_hexstring = HelperFunctions.int_to_hex_string(result)
        if self.verbose:
            print("--Result of " + inst, result_hexstring)
        if dest != 'x0':
            self.registers[dest] = result_hexstring
        self.pc += 4

    def imm(self, inst, dest, xval, const):
        """Executes xval inst const, saves result in dest.
        inst is a string instruction.
        dest, xval1 are strings of register names.
        const is integer constant.
        pc is updated.
        """

        if dest == "x0":
            self.pc += 4
            return

        # capping the value of the constant.
        const = HelperFunctions.hex_string_to_int( HelperFunctions.int_to_hex_string
( const))

        reg_xval = HelperFunctions.hex_string_to_int(self.registers.get(xval,
'0x00000000'))
```

```
        if inst == 'slli':
            shift_binstring = bin(int(HelperFunctions.int_to_hex_string(const),
16))[2:][-6:]
            # this needs ^ to be done in case const is -, then we want the
hex string and extract lower bits
            shift_val = int(shift_binstring, 2) %32
            result = int(reg_xval * 2**(shift_val))

        elif inst == 'srli':
            # print("SRLI INSTRUCTION")
            # print("xval", xval)
            reg1_hexstring = self.registers.get(xval, '0x00000000')
            # print("reg1_hextring", reg1_hexstring)
            reg1_raw_binstring = bin(int(reg1_hexstring, 16))[2:] #remove
'0b'
            # print("reg1_raw_binstring", reg1_raw_binstring)
            reg1_binstring = '0'*(32-len(reg1_raw_binstring)) + reg1_raw_binstring
            # print("reg1_binstring", reg1_binstring)
            # we do not use hex_to_int because it is off by 1 for shifting
negative nums

            # now we get shift amount
            shift_binstring = bin(int(HelperFunctions.int_to_hex_string(const),
16))[2:][-6:]
            # this needs ^ to be done in case const is -, then we want the
hex string and extract lower bits

            shift_val = int(shift_binstring, 2) %32
            # print("shift_val", shift_val)
```

```
            filler = '0'
            shifted = reg1_binstring[:-shift_val] # all bits except last
shift_val bits
            # print("shifted", shifted)

            result = int('0b' + filler*(32 - len(shifted)) + shifted, 2)

            # print("result", result)

        # slli x1, x2, offset
        # shifts whatever in x2 by offset, stores result in reg x1
        # insert 0 shift, binary representation
        # srai copies the bit sign of highest bit
        # the cornell implemented this incorrectly
        elif inst == "srai":
            reg1_hexstring = self.registers.get(xval, '0x00000000')
            reg1_raw_binstring = bin(int(reg1_hexstring, 16))[2:] #remove
'0b'
            reg1_binstring = '0'*(32-len(reg1_raw_binstring)) + reg1_raw_binstring
            # we do not use hex_to_int because it is off by 1 for shifting
negative nums

            # now we get shift amount
            shift_binstring = bin(int(HelperFunctions.int_to_hex_string(const),
16))[2:][-6:]
            # this needs ^ to be done in case const is -, then we want the
hex string and extract lower bits
            shift_val = int(shift_binstring, 2) %32
            filler = reg1_binstring[0]
```

```
            shifted = reg1_binstring[:-shift_val] # all bits except last
shift_val bits
            result = int('0b' + filler*(32 - len(shifted)) + shifted, 2)

        elif inst == "xori":
            # xor 32 bit binary of xval with constant,
            # store result in dest
            result = const ^ reg_xval

        elif inst == "addi":
            result = reg_xval + const

        # elif inst == "subi":
        #     result = reg_xval - const

        elif inst == "andi":
            result = reg_xval & const

        elif inst == "ori":
            result = reg_xval | const

        elif inst == "sltiu":
            # returns true if the register value is less than unsigned (sign
extended constant)
            #print("reg_xval", int(HelperFunctions.int_to_hex_string(reg_xval),16))
            #print("const", int(HelperFunctions.int_to_hex_string(const),16))
            # we need to use int(,16) to get sign extended version - unsigned
comparison of both
            # otherwise we can just use HelperFunctions.hex_string_to_int()
wrapped around int_to_hex_string().
```

```
            result = int(int(HelperFunctions.int_to_hex_string(reg_xval),16)
< int(HelperFunctions.int_to_hex_string(const),16))

        elif inst == 'slti':
            # same as sltiu but signed
            result = reg_xval < const

        else:
            print("Unsupported imm instruction.")
            self._raise_error()

        result_hexstring = HelperFunctions.int_to_hex_string(result)
        if self.verbose:
            print("--Result of " + inst, result_hexstring)
        if dest != 'x0':
            self.registers[dest] = result_hexstring
        self.pc += 4
        return

    def pc_inst(self, inst, dest, const):
        const = HelperFunctions.hex_string_to_int( HelperFunctions.int_to_hex_string
(const))

        if inst == "auipc":
            # upper immediate is constant left shift by 12 bits
            result = self.pc + const*(2**12)

        elif inst == "lui":
            result = const*(2**12)
```

```
        result_hexstring = HelperFunctions.int_to_hex_string(result)
        if self.verbose:
            print("--Result of " + inst, result_hexstring)
        if dest != 'x0':
            self.registers[dest] = result_hexstring
        self.pc += 4

    def branch(self, inst, x1, x2, offset):
        """Executes branch instruction. If x1 inst x2, then pc moves by
offset.
        inst is string instruction.
        x1, x2 are string register names.
        offset is integer, must be multiple of 4.
        pc is updated.
        """
        assert offset%4 == 0

        do_branch = False
        x1_value = HelperFunctions.hex_string_to_int(self.registers.get(x1,
'0x00000000'))
        x2_value = HelperFunctions.hex_string_to_int(self.registers.get(x2,
'0x00000000'))

        if inst == "beq":
            do_branch = x1_value == x2_value
        elif inst == "bne":
            do_branch = x1_value != x2_value
        elif inst == "blt":
            do_branch = x1_value < x2_value
        elif inst == "bge":
```

```
            do_branch = x1_value >= x2_value
        elif inst == "bltu":
            # unsigned comparison
            do_branch = int(self.registers.get(x1, '0x00000000'),16) < int
(self.registers.get (x2, '0x00000000'),16)
        elif inst == "bleu":
            # unsigned less than or equal to
            do_branch = int(self.registers.get(x1, '0x00000000'),16) <=
int(self.registers.get(x2, '0x00000000'),16)
        elif inst == "bgeu":
            do_branch = int(self.registers.get(x1, '0x00000000'),16) >=
int(self.registers.get(x2, '0x00000000'),16)
        elif inst == "bgt":
            do_branch = x1_value > x2_value
        elif inst == "bgtu":
            do_branch = int(self.registers.get(x1, '0x00000000'),16) > int
(self.registers.get (x2, '0x00000000'),16)
        elif inst == "ble":
            do_branch = x1_value <= x2_value
        # "bgeu", "bgt", "bgtu", "ble",
        else:
            print("Unsupported branch instruction.")
            self._raise_error()

        if do_branch:
            self.pc += offset
        else:
            # no branch, go to next instruction
            self.pc += 4
        return
```

```
    def jump_jal(self, inst, rd, imm):
        """Jumps to offset imm more than pc, and saves current pc+4"""
        # save pc+4 in rd
        if rd != 'x0':
            self.registers[rd] = HelperFunctions.int_to_hex_string(self.pc
+ 4)
        # move pc to pc + imm
        self.pc += imm #imm already parsed to int
        return

    def jump_jalr(self, inst, rd, rs1, imm):
        """Jumps to (address saved in rs1) + imm, zeroing the last bit
        saves pc+4 in rd."""
        if rd != 'x0':
            self.registers[rd] = HelperFunctions.int_to_hex_string(self.pc
+ 4)
        rs_address = self.registers.get(rs1, "0x00000000")
        self.pc = HelperFunctions.hex_string_to_int(rs_address) + imm #regs
store hex
        return

    def store(self, inst, xvalue, offset, xmem_reg):
        """Executes store instruction. Value inside register xvalue is stored
at:
        (address specified in register xmem) + offset.
        inst is string instruction
        xvalue, xmem are string register names.
        offset is integer number, must be correct multiple (depending on
inst).
```

```
        Note that depending on the type of inst, the number of bytes saved
varies.
        pc is updated.
        """
        # example:
        # sw x1, 8(x2)
        # stores value saved in x1 reg into location (x2 reg) + 8

        # the total offset is the offset conributed by
        # offset input, and the amount 'xmem' is off from a multiple of
4.
        xmem = self.registers.get(xmem_reg, "0x00000000")
        # print('--xmem initially', xmem)
        xmem_rem = HelperFunctions.hex_string_to_int(xmem)%4 # offset from
multiple of 4 memory address
        # print('--xmem_rem, offset from multiple of 4', xmem_rem)
        xmem = HelperFunctions.int_to_hex_string(HelperFunctions.hex_string_to_int
(xmem) - xmem_rem)
        # print('--xmem-xmem_rem', xmem)
        offset += xmem_rem
        # print('--offset',offset)

        # offset might be a multiple of 4.
        # note this is compatible with negative offset.
        whole_lines = int(offset)//4
        # print('--whole lines within offset', whole_lines)
        rem = int(offset)%4
        # print('--true remainder after offset', rem)

        location_int = HelperFunctions.hex_string_to_int(xmem) + whole_lines*4
```

```
# need to mul by 4
        location_hex = HelperFunctions.int_to_hex_string(location_int)
        # print('--location hex, whole lines and xmem', location_hex)

        mem_string = self.memfile.read(location_hex)
        # print('--value read from location', mem_string)

        word_to_store = self.registers.get(xvalue, '0x00000000')
        # print('--value from reg to store', xvalue, word_to_store)
        INTERVAL = 0

        # one byte is 2 hex digits, or 8 bits.
        # depending on instruction, read different length.
        if inst == 'sb':
            INTERVAL = 2
        elif inst == 'sh':
            assert rem == 0 or rem == 2, "sh supports 16 bit/2 byte/4 hex
digit offsets."
            INTERVAL = 4
        elif inst == 'sw':
            assert rem == 0, "sw supports 32 bit/4 byte/8 hex digit offsets."
            INTERVAL = 8
        else:
            print("Unsupported store instruction.")
            self._raise_error()

        # extract length of store string based on inst
        word_to_store = word_to_store[-INTERVAL:]
        if self.verbose:
            print("--Storing", word_to_store)
```

```
            print("--To address", location_hex)
        new_mem_string = mem_string[0:len(mem_string)-(rem+INTERVAL//2)*2]
+ word_to_store + mem_string[len(mem_string)-(rem+INTERVAL//2)*2+INTERVAL:]

        # update memory
        self.memfile.update(location_hex, new_mem_string)
        if self.verbose:
            print("--Result of store", new_mem_string)
        self.pc += 4
        return

    def _raise_error(self):
        """Internal function to raise syntax error on current line."""
        line = self.pc//4
        raise SyntaxError('Invalid instruction on line ' + str(line+1),
                          (self.codefilepath, line + 1, 0, self.codefilelines
[line]))

    def load(self, inst, xinto, offset, xmem_reg):
        """Loads what's saved in xmem (with offset) into register xinto.
        inst: string of instruction
        xmem: string of memory location
        offset: integer number (should align with memory, depending on inst)
        xinto: string of register name
        pc is updated.
        """
        # example:
        # lw x1, 8(x2)
        # loads into reg x1, value from location (x2 reg) + 8
```

```
        if xinto == "x0":
            self.pc += 4
            return

        # the total offset is the offset conributed by
        # offset input, and the amount 'xmem' is off from a multiple of
4.
        xmem = self.registers.get(xmem_reg, "0x00000000")
        # print('--xmem original', xmem)
        xmem_rem = HelperFunctions.hex_string_to_int(xmem)%4 # offset from
multiple of 4 memory address
        xmem = HelperFunctions.int_to_hex_string(HelperFunctions.hex_string_to_int
(xmem) - xmem_rem)
        # print('--xmem ', xmem)

        offset += xmem_rem
        # print('--offset',offset)

        # offset might be a multiple of 4.
        # note this is compatible with negative offset.
        whole_lines = int(offset)//4
        rem = int(offset)%4

        # print('--whole_lines in offset', whole_lines)
        # print('--true rem', rem)

        location_int = HelperFunctions.hex_string_to_int(xmem) + whole_lines*4
 # need to mul by 4!
        location_hex = HelperFunctions.int_to_hex_string(location_int)
```

```
        # print('--location', location_hex)
        mem_string = self.memfile.read(location_hex) #location points to
very last bit in the line
        # print('--mem_string at location', mem_string)

        INTERVAL = 8
        # setting intervals
            # 0 offset: -2 because read 2 bits
            # 1 offset: -4 4 bits from the end
            # 2 offset: -6
            # 3 offset: -8

        if inst == 'lb' or inst == 'lbu': # include the unsigned variant
            INTERVAL=2

        elif inst == 'lh' or inst == 'lhu': # include unsigned variant
            assert rem == 0 or rem == 2, "lh supports 16 bit/2 byte/4 hex
digit offsets."
            INTERVAL = 4

        elif inst == 'lw':
            assert rem == 0, "lw supports 32 bit/4 byte/8 hex digit offsets."
            INTERVAL = 8

        else:
            print("Unsupported load instruction.")
            self._raise_error()

        value = mem_string[
            len(mem_string)-(rem+INTERVAL//2)*2: len(mem_string)-(rem+INTERVAL//2)
```

```
*2+INTERVAL
            ]

        # print('--final value loaded', value)
        if self.verbose:
            print('--Loaded bits', value)

        if xinto != 'x0':
            # updated with sign extending.
            if inst == 'lbu' or inst == 'lhu':
                self.registers[xinto] = '0x' + (8-len(value))*'0' + value
            else: # sign extending
                self.registers[xinto] = '0x' + (8-len(value))*(value[0])
+ value

        # print('--reg saved to', xinto)
        if self.verbose:
            print("--Saved as", self.registers.get(xinto, '0x00000000'))
# in case if it is x0, save nothing

        self.pc += 4
        return

    def save(self, output_location):
        """Saves memory into output_location. Does not save registers. Does
not reset."""
        self.memfile.save(output_location)
        return True

class MemFile:
```

```
"""Class representing a memory file.
Operations to load, update, read, and save.
Operations don't modify original file."""
# assumes memory file is separated by whitespace
def __init__(self):
    self.filepath = ""
    self.lines = {} # maps address number to value

def load(self, filepath):
    """loads memory specified in filepath as memory.
    Expects each line to be formatted like "address   value"
    For example: 0x00000004 0xabababab
    """
    self.filepath = filepath
    with open(filepath, 'r') as f:
        inst_list = f.readlines()
    for line in inst_list:
        if line.strip() == "":
            continue
        address, value = line.split()
        # make sure value is 32 bits
        # reads the '0x0' string into binary, truncates last 32,
        # then puts back into integer.
        value = HelperFunctions.int_to_hex_string(HelperFunctions
            .int_or_hex_to_int(value))
        self.lines[address] = value

def update(self, line, data):
    """Updates the data saved at address "line".
    line, data are string hex values."""
```

```
        # ensure values are 32 bit
        line = HelperFunctions.int_to_hex_string(HelperFunctions.hex_string_to_int
(line))
        data = HelperFunctions.int_to_hex_string(HelperFunctions.hex_string_to_int
(data))

        self.lines[line] = data

    def read(self, line):
        """Reads memory from address "line"."""
        ## line must be in hex string
        ## add functionality for lines that aren't multiples of 4 exactly.
        return self.lines.get(line, '0x00000000')

    def save(self, filepath):
        """Save current memory into location filepath."""
        with open(filepath, 'w') as outfile:
            for i in self.lines.keys():
                outfile.write(i + "\t" + self.lines[i].strip() + "\n")
        print("Memory saved to", filepath)
        return True


class HelperFunctions:

    @staticmethod
    def int_to_hex_string(intvalue):
        """Converts an integer into 32 bit hex string (8 digits)
        Implements two's complement and truncates."""
        # intvalue up to 2**31-1
```

```
        # 2**31 is 1000...0 in binary, or 800..00 in hex
        # negative int is -2**31
        intvalue = int(intvalue) # in case input is a string
        intvalue = (2**32 + intvalue)%2**32
        hex_str = hex(intvalue)[2:]
        return '0x' + '0'*(8-len(hex_str)) + hex_str

    @staticmethod
    def hex_string_to_int(hexstr):
        """Converts hex string into integer value.
        Implements two's complement and assumes 32 bit (or less) input."""
        intvalue = int(hexstr, 16)
        actualvalue = (intvalue - 2**31 + 1) %(-2**32) + 2**31 - 1
        return actualvalue

    @staticmethod
    def int_or_hex_to_int(num):
        """Interprets num as an integer, where num is either an integer,
        string integer, or hex string."""
        try:
            if type(num) == int:
                return num
            elif type(num) == str:
                if num[0:2] == "0x" or num[0:3] == "-0x":
                    num = HelperFunctions.hex_string_to_int(num)
                else:
                    num = HelperFunctions.hex_string_to_int(HelperFunctions.
int_to_hex_string(int(num)))
                    # this is needed to ensure handling of twos complement
                return num
```

```
        except:
            raise SyntaxError("Invalid constant: " + str(num) + " is not
a number or hex string.")
```

B.6 Sample RISC-V Code: Pseudorandom Number Generator

```
auipc gp,0x10000
addi gp,gp,88
auipc sp,0x10010
addi sp,sp,-8
jal ra,0x48
wfi
lui a5,0xef7a9
lui a4,0x41c65
lui a3,0x3
addi a5,a5,-143
addi a4,a4,-403
addi a3,a3,57
blt zero,a0,0xc
addi a0,a5,0
jalr zero,ra,0x0
mul a5,a5,a4
addi a0,a0,-1
add a5,a5,a3
sltiu a2,a5,-1
addi a2,a2,-1
sub a5,a5,a2
jal zero,-0x24
addi sp,sp,-32
sw s0,24(sp)
sw s1,20(sp)
sw s2,16(sp)
sw s3,12(sp)
```

```
sw s4,8(sp)
sw ra,28(sp)
addi s0,zero,0
auipc s4,0x10000
addi s4,s4,-120
auipc s3,0x10000
addi s3,s3,-84
auipc s2,0x10000
addi s2,s2,-48
addi s1,zero,44
add a4,s4,s0
add a5,s3,s0
lw a5,0(a5)
lw a0,0(a4)
mul a0,a0,a5
addi a0,a0,2020
jal ra,-0x94
add a5,s2,s0
sw a0,0(a5)
addi s0,s0,4
bne s0,s1,-0x28
lw ra,28(sp)
lw s0,24(sp)
lw s1,20(sp)
lw s2,16(sp)
lw s3,12(sp)
lw s4,8(sp)
addi sp,sp,32
jalr zero,ra,0x0
```

Below is the corresponding memory initialization file:

```
0x10000000  0x00000000
0x10000004  0x00000001
0x10000008  0x00000002
0x1000000c  0x00000003
0x10000010  0x00000004
0x10000014  0x00000005
0x10000018  0x00000006
0x1000001c  0x00000007
0x10000020  0x00000008
0x10000024  0x00000009
0x10000028  0x0000000a
0x1000002c  0x00000064
0x10000030  0x000000c8
0x10000034  0x0000012c
0x10000038  0x00000190
0x1000003c  0x000001f4
0x10000040  0x00000258
0x10000044  0x000002bc
0x10000048  0x00000320
0x1000004c  0x00000384
0x10000050  0x000003e8
0x10000054  0x0000044c
0x10000058  0x0000000a
0x1000005c  0x0000000a
0x10000060  0x0000000a
0x10000064  0x0000000a
0x10000068  0x0000000a
0x1000006c  0x0000000a
0x10000070  0x0000000a
0x10000074  0x0000000a
```

```
0x10000078  0x0000000a
0x1000007c  0x0000000a
0x10000080  0x0000000a
```

B.7 Example Neural Network in C for Simulator and Experimental Measurement Comparison

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// pre trained iris weights, when randomize seed 98
const int weight[3][3][4] =
{
    {
        {-3, -39, -29, -40},
        {-38, -19, 32, 44},
        {-13, 36, -8, -32}
    },
    {

        {28, -10, 12},
        {-42, 48, 0},
        {8, 20, -56}
    },

    {

        {-43, 15, 45},
        {-24, -12, -19},
        {-50, -1, -35}

    }
};
```

```
const int bias[3][3] =
{
    {-2, -28, -49},
    {4756, 2415, 946},
    {33400, -132200, -18600}
};

int layer1[3] = {3,3,3};
int layer2[3] = {5,5,5};
int layer3[3] = {7,7,7}; // output layer

int x[4] = {51,35,14,02};

int relu(int x) {
int out;
if (x>0) {
out= x;
} else
{
out= 0;
}
return out;
}

void main( void )
{
    int l1i;
    for (l1i=0; l1i < 3; l1i++) {
     layer1[l1i] = relu(
```

```
 x[0]*weight[0][l1i][0]
 + x[1]*weight[0][l1i][1]
 + x[2]*weight[0][l1i][2]
 + x[3]*weight[0][l1i][3]
 + bias[0][l1i]
 );
}

int l2i;
for (l2i = 0; l2i < 3; l2i++) {
 layer2[l2i] = relu(
 layer1[0]*weight[1][l2i][0]
 + layer1[1]*weight[1][l2i][1]
 + layer1[2]*weight[1][l2i][2]
 + bias[1][l2i]
 );
}

int l3i;
for (l3i = 0; l3i < 3; l3i++) {
 layer3[l3i] = layer2[0]*weight[2][l3i][0]
 + layer2[1]*weight[2][l3i][1]
 + layer2[2]*weight[2][l3i][2]
 + bias[2][l3i]
 ;
}

}
```

Bibliography

[1] U. Banerjee, T. Ukyab, and A. P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR TCHES*, 2019:17–61, August 2019.

[2] U. Banerjee, A. Wright, C. Juvekar, M. Waller, Arvind, and A. P. Chandrakasan. An energy-efficient reconfigurable dtls cryptographic engine for securing internet-of-things applications. *IEEE JSSC*, 54(8):2339–2352, August 2019.

[3] Joppe Bos, Craig Costello, Leo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! Practical, Quantum-Secure Key Exchange from LWE. Cryptology ePrint Archive, Report 2016/659, 2016. https://eprint.iacr.org/2016/659.

[4] Cornell. RISC-V Interpreter. https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/.

[5] Yann Le Corre, Johann Grosschadl, and Daniel Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on arm cortex-m3 processors. Cryptology ePrint Archive, Report 2017/1253, 2017. https://eprint.iacr.org/2017/1253.

[6] Mark A. Finlayson. *Introducing the ARM Architecture.* ARM Limited.

[7] Tim Guneysu, Vadim Lyubashevsky, and Thomas Poppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012*, pages 530–547, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[8] N. Homma, T. Aoki, and A. Satoh. Electromagnetic information leakage for side-channel analysis of cryptographic modules. In *2010 IEEE International Symposium on Electromagnetic Compatibility*, pages 97–102, 2010.

[9] A. Krieg, J. Grinschgl, C. Steger, R. Weiss, and J. Haid. A side channel attack countermeasure using system-on-chip power profile scrambling. In *2011 IEEE 17th International On-Line Testing Symposium*, pages 222–227, 2011.

[10] N. Lawson. Side-channel attacks on cryptographic software. *IEEE Security Privacy*, 7(6):65–68, 2009.

[11] D. McCann, E. Oswald, and C. Whitnall. Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages. *26th USENIX Security Symposium*, August 2017.

[12] Chris Peikert. A decade of lattice cryptography. Cryptology ePrint Archive, Report 2015/939, 2015. https://eprint.iacr.org/2015/939.

[13] B. K. Reddy, M. J. Walker, D. Balsamo, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett. Empirical cpu power modelling and estimation in the gem5 simulator. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, 2017.

[14] riscv.org. Spike RISC-V ISA Simulator. https://github.com/riscv/riscv-isa-sim, 2020.

[15] Sahbuddin Abdul Kadir, A. Sasongko, and M. Zulkifli. Simple power analysis attack against elliptic curve cryptography processor on fpga implementation. In *Proceedings of the 2011 International Conference on Electrical Engineering and Informatics*, pages 1–4, 2011.

[16] W. Schindler, K. Lemke, and C. Paar. A stochastic model for differential side channel cryptanalysis. *LNCS*, 3659:30–46, 2005.

[17] A. Shah. Armsim: An instruction-set simulator for the arm processor.

[18] A. Sinha and A. P. Chandrakasan. JouleTrack - A Web based Tool for Software Energy Profiling. *IEEE/ACM DAC*, pages 220–225, June 2001.

[19] C. Thuillet, P. Andouard, and O. Ly. A smart card power analysis simulator. *CSE 2009*, page 847–852, 2009.

[20] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*, December 2019.

[21] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, June 2019.