Kite: Architecture Simulator for RISC-V Instruction Set

William J. Song
School of Electrical Engineering, Yonsei University
wjhsong@yonsei.ac.kr

June, 2022 (Version 1.9)

1. Introduction

Kite is an architecture simulator that models a five-stage pipeline of RISC-V instruction set [1]. The initial version of Kite was developed in 2019 primarily for an educational purpose as a part of EEE3530 Computer Architecture. Kite implements the five-stage pipeline model described in *Computer Organization and Design, RISC-V Edition: The Hardware and Software Interface* by D. Patterson and J. Hennessey [2]. The objective of Kite is to provide students with an easy-to-use simulation framework and precise timing model as described in the book. It supports most of basic instructions introduced in the book such as `add`, `slli`, `ld`, `sd`, `beq` instructions. Users can easily compose RISC-V assembly programs and execute them through the provided pipeline model for entry-level architecture studies. The pipeline model in Kite provides several functionalities including instruction dependency checks (i.e., data hazards), pipeline stalls, data forwarding or bypassing (optional), branch predictions (optional), data cache structures, etc. The remainder of this document describes the details of Kite implementations and usages.

2. Prerequisite, Download, and Installation

The five-stage pipeline model in Kite is implemented in C++. As pointed out earlier in the introduction of this document, Kite was developed primarily for an educational purpose. Its objective is to have the apprentices of computer architecture experience architecture simulations with a simple, easy-to-use framework. If you join the computer architecture world in your career (either product development or research), you will certainly have to use some sorts of architecture simulators for your work. The majority of architecture simulators are written in C/C++ since this programming language is the most suitable one to interface between computer hardware and software. If you are not familiar with C++ programming in Linux, this will be a perfect chance to have hands-on experiences with Kite.

The simple implementation of Kite makes it really easy to install. It requires only g++ compiler to build, and it does not depend on any other libraries or external tools to run. It has been validated in Ubuntu 16.04 (Xenial), 18.04 (Bionic Beaver), 20.04 (Focal Fossa), and Mac OS 10.14 (Majave), 10.15 (Catalina), 11 (Big Sur), 12 (Monterey). The latest release of Kite is v1.9. To obtain a copy of Kite v1.9, use the following `git` command in terminal. Then, enter the `kite/` directory and build the simulator using a `make` command.

```
$ git clone --branch v1.9 https://github.com/yonsei-icsl/kite
$ cd kite/
$ make -j
```

3. Program Code, Register and Memory States

Kite needs three input files to run, i) program code, ii) register state, and iii) data memory state. A program code refers to a plain text file containing RISC-V assembly instructions. In the main directory of Kite (i.e., `kite/`), you should find a file named `program_code` that contains the following RISC-V instructions after some comment lines.

```
# Kite program code
loop:   beq  x11, x0,  exit
        remu x5,  x10, x11
        add  x10, x11, x0
        add  x11, x5,  x0
        beq  x0,  x0,  loop
exit:   sd   x10, 2000(x0)
```

In the program code, a # symbol is reserved to indicate the beginning of comment. Everything after # until the end of line is treated as a comment and not read. Each instruction in the program code is sequentially assigned an address starting from PC = 4. For example, `beq` is the first instruction in the code above, and it is given the PC value of 4. The next instruction, `remu`, has PC = 8 because every RISC-V instruction is 4 bytes long [1]. The size of the code segment is defined as 1KB (or 255 instructions). A program size greater than 1KB will throw an error.

Instructions in the program code are sequentially loaded from the top, unless branch or jump instructions alter the next PC to fetch. The program ends when the next PC naturally goes out of code segment where there exists no valid instruction, or when the next PC is set to zero (i.e., PC = 0). The PC value of zero is reserved as an invalid address. Note that labels, instructions, and register operands are case-insensitive. It means that `beq`, `Beq`, `BEQ` are all identical in the program code. The current version of Kite supports the following list of RISC-V basic instructions. Instructions in the table are listed in the alphabetical order in each category. Pseudo instructions (e.g., `mv`, `not`) and ABI register names (e.g., `sp`, `a0`) are not supported.

Types	RISC-V instructions	Operations in C/C++
R type	`add rd, rs1, rs2`	`rd = rs1 + rs2`
	`and rd, rs1, rs2`	`rd = rs1 & rs2`
	`div rd, rs1, rs2`	`rd = rs1 / rs2`
	`divu rd, rs1, rs2`	`rd = (uint64_t)rs1 / (uint64_t)rs2`
	`mul rd, rs1, rs2`	`rd = rs1 * rs2`
	`or rd, rs1, rs2`	`rd = rs1 | rs2`
	`rem rd, rs1, rs2`	`rd = rs1 % rs2`
	`remu rd, rs1, rs2`	`rd = (uint64_t)rs1 % (uint64_t)rs2`
	`sll rd, rs1, rs2`	`rd = rs1 << rs2`
	`sra rd, rs1, rs2`	`rd = rs1 >> rs2`
	`srl rd, rs1, rs2`	`rd = (uint64_t)rs1 >> rs2`
	`sub rd, rs1, rs2`	`rd = rs1 - rs2`
	`xor rd, rs1, rs2`	`rd = rs1 ^ rs2`
I type	`addi rd, rs1, imm`	`rd = rs1 + imm`
	`andi rd, rs1, imm`	`rd = rs1 & imm`
	`jalr rd, imm(rs1)`	`rd = pc + 4, pc = (rs1 + imm) & -2`
	`slli rd, rs1, imm`	`rd = rs1 << imm`
	`srai rd, rs1, imm`	`rd = rs1 >> imm`
	`srli rd, rs1, imm`	`rd = (uint64_t)rs1 << imm`
	`ori rd, rs1, imm`	`rd = rs1 | imm`
	`xori rd, rs1, imm`	`rd = rs1 ^ imm`
I type	`lb rd, imm(rs1)`	`rd = (int8_t)memory[rs1 + imm]`
	`lbu rd, imm(rs1)`	`rd = (uint8_t)memory[rs1 + imm]`
	`lh rd, imm(rs1)`	`rd = (int16_t)memory[rs1 + imm]`
	`lhu rd, imm(rs1)`	`rd = (uint16_t)memory[rs1 + imm]`
	`lw rd, imm(rs1)`	`rd = (int32_t)memory[rs1 + imm]`
	`lwu rd, imm(rs1)`	`rd = (uint32_t)memory[rs1 + imm]`
	`ld rd, imm(rs1)`	`rd = memory[rs1 + imm]`
S type	`sb rs2, imm(rs1)`	`memory[rs1 + imm] = (uint8_t)rs2`
	`sh rs2, imm(rs1)`	`memory[rs1 + imm] = (uint16_t)rs2`
	`sw rs2, imm(rs1)`	`memory[rs1 + imm] = (uint32_t)rs2`

	`sd rs2, imm(rs1)`	`memory[rs1 + imm] = rs2`
SB type	`beq rs1, rs2, imm`	`pc = (rs1 == rs2 ? pc + imm<<1 : pc + 4)`
	`bge rs1, rs2, imm`	`pc = (rs1 >= rs2 ? pc + imm<<1 : pc + 4)`
	`blt rs1, rs2, imm`	`pc = (rs1 < rs2 ? pc + imm<<1 : pc + 4)`
	`bne rs1, rs2, imm`	`pc = (rs1 != rs2 ? pc + imm<<1 : pc + 4)`
U type	`lui rd, imm`	`rd = imm << 20;`
UJ type	`jal rd, imm`	`rd = pc + 4, pc = pc + imm<<1`
FR type	`fadd.d rd, rs1, rs2`	`rd = (double)rs1 + (double)rs2`
	`fdiv.d rd, rs1, rs2`	`rd = (double)rs1 / (double)rs2`
	`fmul.d rd, rs1, rs2`	`rd = (double)rs1 * (double)rs2`
	`fsub.d rd, rs1, rs2`	`rd = (double)rs1 - (double)rs2`
FI type	`fld rd, imm(rs1)`	`rd = (double)memory[rs1 + imm]`
FS type	`fsd rs2, imm(rs1)`	`memory[rs1 + imm] = (double)rs2`
No type	`nop`	No operation

Kite supports non-doubleword memory accesses such as `lb`, `lhu`, `sw` for both loads and stores. These instructions must have their addresses aligned with their data lengths. For instance, the address of `lw` must be a multiple of 4 for a word-sized data element. The alignment restriction ensures that a memory access is served within a single cache line. Kite also support a few basic floating-point instructions such as `fadd.d`, `fdiv.d`, `fmul.d`, and `fsub.d`, where the tailing `.d` indicates that the instructions are double-precision. Memory instructions for double-precision data (i.e., `fld`, `fsd`) are supported as well. Supports of other floating-point instructions including single-precision data type (e.g., `fadd.s`, `flw`) main as future work.

The exemplary program code shown in the previous page implements Euclid's algorithm that finds a greatest common divisor (GCD) of two unsigned integer numbers in `x10` and `x11`. The equivalent program code in C/C++ is shown below. For instance, suppose that `x10 = 21` and `x11 = 15`. In the first iteration, the remainder of `x10 / x11` (i.e., modulo % operation) is stored in the `x5` register, which is `x5 = 6`. Then, `x11` and `x5` are copied to `x10` and `x11`, respectively. The second iteration makes `x5 = 3` by taking the remainder of `15 / 6`, and `x10 = 6` and `x11 = 3`. In the third iteration, `x5 = 0`, `x10 = 3`, and `x11 = 0`. The loop reaches the end for `x11 == 0`. The last line of code stores the `x10` value containing the GCD of `21` and `15` in data memory at the address `2000` (or at index `250` if the data memory is viewed as a doubleword array).

```
/* Equivalent program code in C/C++ */
while(x11 != 0) {       // loop: beq  x11, x0,  exit
    x5 = x10 % x11;     //       remu x5,  x10, x11
    x10 = x11;          //       add  x10, x11, x0
    x11 = x5;           //       add  x11, x5,  x0
}                       //       beq  x0,  x0,  loop
memory[250] = x10;      // exit: sd   x10, 2000(x0)
```

Register state refers to the `reg_state` file that contains the state (i.e., values) of 32 integer registers of RISC-V. For newly added supports of double-precision floating-point instructions, the `reg_state` file should list 32 floating-point registers as well (i.e., `f0`-`f31`). The register state is loaded when a simulation starts and used for initializing the registers of processor pipeline. You may find the following after comment lines in the register state file.

```
# Kite register state
# 64-bit integer registers
x0 = 0
x1 = 0
x2 = 8
x3 = 0

...

x10 = 21
```

```
x11 = 15

...

x30 = 0
x31 = 0

# Double-precision floating-point registers
f0 = 0.0
f1 = 0.0
f2 = 2.71828
f3 = 3.14159265359

...
```

The register state file must list all 32 integer registers from `x0` to `x31`. The left of an equal sign is a register name (e.g., `x10`), and the right side defines its initial value. The example above shows that `x10` and `x11` registers are set to 21 and 15, respectively. When a simulation starts, the registers are initialized accordingly. A program code will initiate instruction executions based on the defined register state. Since the `x0` register in RISC-V is hard-wired to zero [2], assigning non-zero values to it is automatically discarded.

Memory state refers to the `memory_state` file that contains initial values of data memory at discrete memory addresses. Similar to the register state, the memory state is used for initializing the contents of data memory. Every address in the memory state file must be a multiple of 8 for 64-bit alignment, which implies that the memory cannot be initialized with non-doubleword data. If a value at a memory address is given in a floating-point format such as `1.23`, it will be saved in the memory address as the double-precision data format. The following shows an example of memory state file.

```
# Kite memory state
1024 = 5
1032 = -8
1040 = -4
1048 = 7

...
```

The baseline code of Kite creates a 4KB data memory, and its size is easily modifiable in `proc_t::init()` in the `proc.cc` file. The minimum size of memory is defined as 2KB, where the first 1KB is reserved for the code segment. Each memory block is 8 bytes (or 64 bits) long, and accesses to the data memory must obey the 8-byte alignment. In other words, the memory address of a load or store instruction always have to be a multiple of 8. The memory address alignment is strictly enforced, and a failure will terminate a simulation. The 8-byte alignment rule restricts the scope of memory data handling since byte-granular instructions such as `lbu` and `sb` are not supported. So, the byte-granular memory instructions are not in the list of supported instructions. This restriction will be lifted in future releases. At each line of the memory state shown above, the left of an equal sign is a memory address in multiple of 8, and the right side defines a 64-bit value stored at the memory address. Notably, the memory state file does not have to list all the memory addresses. The values of unlisted memory addresses will be set to zero by default.

4. Running Kite Simulations

Running a Kite simulation invokes a program code associated with register and memory states. After the program execution is done, Kite prints out simulation results including pipeline statistics (e.g., total clock cycles, number of instructions), cache statistics (e.g., number of load and stores), and the final states of registers and data memory. The following shows an example of Kite simulation executing `program_code`.

```
$ ./kite program_code

*******************************************************
* Kite: An architecture simulator for five-stage      *
* pipeline modeling of RISC-V instruction set         *
* Developed by William J. Song                        *
* School of Electrical Engineering, Yonsei University *
* Version: 1.9                                        *
*******************************************************

Start running ...
Done.

======== [Kite Pipeline Stats] =========
Total number of clock cycles = 48
Total number of stalled cycles = 6
Total number of executed instructions = 17
Cycles per instruction = 2.824

Data cache stats:
    Number of loads = 0
    Number of stores = 1
    Number of writebacks = 0
    Miss rate = 1.000 (1/1)

Register state:
x0 = 0
x1 = 0
x2 = 8
x3 = 0
x4 = 0
x5 = 0
x6 = 0
x7 = 0
x8 = 0
x9 = 0
x10 = 3
x11 = 0
x12 = 0
x13 = 0
x14 = 0
x15 = 0
x16 = 0
x17 = 0
x18 = 0
x19 = 0
x20 = 32
x21 = 400
x22 = 720
x23 = 0
x24 = 0
x25 = 0
x26 = 0
```

```
x27 = 0
x28 = 0
x29 = 0
x30 = 0
x31 = 0

Memory state (all accessed addresses):
(2000) = 3
```

The default pipeline model of Kite does not include branch prediction and data forwarding features. The pipeline is stalled on every conditional branch instruction until its outcome is resolved (i.e., taken or not-taken branch direction and branch target address). The lack of data forwarding also stalls the pipeline when an instruction depends on its preceding instructions. The pipeline stall is lifted when the preceding instructions write their results to registers that the stalled instruction depends on.

Kite offers branch prediction and data forwarding as optional features. These options can be enabled by recompiling Kite with `OPT="-DBR_PRED -DDATA_FWD"` flags, where `-DBR_PRED` enables branch prediction, and `-DDATA_FWD` enables data forwarding, respectively. When both features are enabled, noticeable performance improvements (i.e., cycles per instruction) should be observed as follows.

```
$ make clean
$ make -j OPT="-DBR_PRED -DDATA_FWD"
$ ./kite program_code

*******************************************************
* Kite: An architecture simulator for five-stage      *
* pipeline modeling of RISC-V instruction set         *
* Developed by William J. Song                        *
* School of Electrical Engineering, Yonsei University *
* Version: 1.9                                        *
*******************************************************

Start running ...
Done.

======== [Kite Pipeline Stats] =========
Total number of clock cycles = 36
Total number of stalled cycles = 3
Total number of executed instructions = 17
Cycles per instruction = 2.118
Number of pipeline flushes = 4
Number of branch mispredictions = 4
Number of branch target mispredictions = 0
Branch prediction accuracy = 0.429 (3/7)

Data cache stats:
    Number of loads = 0
    Number of stores = 1
    Number of writebacks = 0
    Miss rate = 1.000 (1/1)

Register state:
x0 = 0
x1 = 0
x2 = 8
```

```
x3 = 0

...
```

Kite provides a debugging option that prints out the detailed progresses of instructions at every pipeline stage and at every clock cycle. To enable the debugging option, Kit has to be compiled with a `-DDEBUG` flag as follows.

```
$ make clean
$ make -j OPT="-DDEBUG"
$ ./kite program_code

*******************************************************
* Kite: An architecture simulator for five-stage      *
* pipeline modeling of RISC-V instruction set         *
* Developed by William J. Song                        *
* School of Electrical Engineering, Yonsei University *
* Version: 1.9                                        *
*******************************************************

Start running ...
1 : fetch : [pc=4] beq x11, x0, 10(exit) [beq 0, 0, 10(exit)]
2 : decode : [pc=4] beq x11, x0, 10(exit) [beq 15, 0, 10(exit)]
3 : execution : [pc=4] beq x11, x0, 10(exit) [beq 15, 0, 10(exit)]
4 : memory : [pc=4] beq x11, x0, 10(exit) [beq 15, 0, 10(exit)]
5 : writeback : [pc=4] beq x11, x0, 10(exit) [beq 15, 0, 10(exit)]
5 : fetch : [pc=8] remu x5, x10, x11 [remu 0, 0, 0]
6 : decode : [pc=8] remu x5, x10, x11 [remu 0, 21, 15]
6 : fetch : [pc=12] add x10, x11, x0 [add 0, 0, 0]
7 : alu : [pc=8] remu x5, x10, x11 [remu 6, 21, 15]
7 : decode : [pc=12] add x10, x11, x0 [add 0, 15, 0]
7 : fetch : [pc=16] add x11, x5, x0 [add 0, 0, 0]
8 : execution : [pc=8] remu x5, x10, x11 [remu 6, 21, 15]
8 : decode : [pc=12] add x10, x11, x0 [add 0, 15, 0]
8 : fetch : [pc=16] add x11, x5, x0 [add 0, 0, 0]
9 : memory : [pc=8] remu x5, x10, x11 [remu 6, 21, 15]
9 : execution : [pc=12] add x10, x11, x0 [add 15, 15, 0]
9 : fetch : [pc=16] add x11, x5, x0 [add 0, 0, 0]
10 : writeback : [pc=8] remu x5, x10, x11 [remu 6, 21, 15]
10 : memory : [pc=12] add x10, x11, x0 [add 15, 15, 0]
10 : decode : [pc=16] add x11, x5, x0 [add 0, 6, 0]
10 : fetch : [pc=20] beq x0, x0, -8(loop) [beq 0, 0, -8(loop)]

...
```

The example above shows that turning on the debugging option discloses the details of instruction executions in the pipeline. For instance, the first instruction of program code, `beq`, is fetched at clock cycle #1. The debug message shows `1 : fetch : [pc = 4] beq x11, x0, 10(exit) [beq 0, 0, 10(exit)]`. The leading `1` indicates the clock cycle 1, and `fetch` means that the instruction is currently at the fetch stage. `pc=4` shows the PC of the `beq` instruction. The remainder of the debug message shows the instruction format. Since the last argument of `beq` instruction is an immediate value (i.e., distance to a branch target in unit of two bytes), `10(exit)` means that the label `exit` is replaced with an immediate value of `10`. The repeated instruction format inside the squared brackets shows the actual values of register operands. Since the registers are not yet read in the fetch stage, the values of `x11` and `x0` registers are simply shown as zeros. The correct values of source operand registers appear when the

instruction moves to the decode stage, and the value of destination operand register (if any) becomes available when the instruction reaches the execute stage for arithmetic-logic instructions or memory stage for memory instructions. As described, turning on the debugging option discloses the details of instruction executions in the pipeline.

5. Implementation

This section describes more advanced contents about the implementations of Kite codes. Kite is comprised of the following files in the `kite/` directory. They are explained in an order to better describe them rather than in the alphabetical order.

```
$ ls | grep ".h\|.cc"

alu.cc           data_cache.cc   defs.h          inst_memory.h  proc.cc
alu.h            data_cache.h    inst.cc         main.cc        proc.h
br_predictor.cc  data_memory.cc  inst.h          pipe_reg.cc    reg_file.cc
br_predictor.h   data_memory.h   inst_memory.cc  pipe_reg.h     reg_file.h
```

- `main.cc` contains a `main()` function that creates, initializes, and runs Kite.
- `defs.h` declares the `enum` lists of RISC-V instructions (e.g., `op_add`, `op_ld`), instruction types (e.g., `op_r_type`, `op_i_type`), integer registers (e.g., `reg_x0`, `reg_x1`), and ALU latencies of all supported instructions. Kite supports multi-cycle ALU executions, and the default setting in `defs.h` defines that `div`, `divu`, `mul`, `remu`, `remu`, `fdiv.d`, and `fmul.d` instructions take two cycles for an ALU to execute. All other instructions take one clock cycle. `defs.h` provides several macros near the end of file to convert strings to `enum` lists (e.g., `get_op_opcode()`, `get_op_type()`), and vice versa.
- `inst.h/cc` files define a C++ class that represents a RISC-V instruction. RISC-V instructions in a program code (e.g., `program_code`) are read when a simulation initiates, and each assembly instruction in character string is converted into the `inst_t` class that carries necessary instruction information such as program counter (i.e., `pc`), register numbers (e.g., `rs1_num`), register values (e.g., `rs1_val`), immediate value (i.e., `imm`), memory addresses (i.e., `memory_addr`), control directives (e.g., `alu_latency`, `rd_ready`), etc.
- `inst_memory.h/cc` files implement an instruction memory. It provides a processor pipeline with instructions directed by program counters (PCs). In particular, the instruction memory creates a copy of an instruction and returns a pointer to it when fetching the instruction. The processor pipeline works on the pointer to the instruction rather than the actual copy of it since passing the pointer is much lighter and faster than passing the sizable `inst_t` class. When a simulation starts, the instruction memory loads a program code that contains RISC-V assembly instructions. It parses the program code and stores the instructions in `inst_t` classes. The instruction memory keeps supplying the processor pipeline with instructions until the PC goes out of a code segment where there exists no valid instruction. PC = 0 is reserved as invalid, and it makes the pipeline stop fetching instructions. The size of the code segment is set to 1KB by default at the bottom of the address space.
- `pipe_reg.h/cc` files define a class named `pipe_reg_t` that models a pipeline register connecting two neighboring stages such as IF/ID. The class has four functions to manipulate the pipeline register. The `read()` function probes the pipeline register and returns a pointer to an instruction if it holds one. This function does not remove the instruction from the pipeline register. It requires an explicit invocation of `clear()` function to remove the instruction from it. The `write()` function writes an instruction into the pipeline register, and `is_available()` tests if the pipeline register contains no instruction inside and thus is free.
- `reg_file.h/cc` files implement a register file as a class, `reg_file_t`. The register file contains 32 64-bit integer registers (i.e., `x0`-`x31`) and 32 double-precision floating-point registers (i.e., `f0`-`f31`). It can be written by `write()` function by specifying a register number to access and supplying a datum to write into it. When a simulation starts, the register file loads a state file (i.e., `reg_state`) that defines the initial values of registers. Notably, dependency checking logic is embedded into the register file, instead of probing pipeline registers to detect data hazards. Checking data dependency is simply done by maintaining a table that traces which instruction is the

last producer of each register. When a branch mis-prediction occurs, the table is flushed to discard the information of wrong-path instructions.

- `alu.h/cc` files model an arithmetic-logical unit (ALU) that executes instructions based on their operation types. To implement type-aware operations, the operands of unsigned instructions such as `divu` are cast to `uint64_t` to produce correct calculation results. Floating-point instructions read register operands via `read_fp()`, which reads register values in the double-precision format. It is assumed that instructions cannot be pipelined within the ALU since doing so can potentially cause out-of-order executions and resource conflicts. If an instruction such as `div` and `mul` takes multiple cycles in the ALU, all the subsequent instructions must stall until the ALU becomes available.

- `data_memory.h/cc` files implement a data memory that holds data values. When a simulation starts, the data memory loads a state file (i.e., `memory_state`) to set memory addresses to store defined values. For easier alignment with 64-bit registers, the data memory requires that the memory address of a load or store instruction must be a multiple of 8. The default size of data memory is 4KB, and the size is easily modifiable by entering a different size in the `data_memory_t()` constructor. Since the first 1KB of the address space is reserved for the code segment, the memory address of a load or store instruction must be greater than 1KB.

- `data_cache.h/cc` files implement a data cache that stores a subset of memory blocks. A cache block can be any multiple of 8 bytes (e.g., 16-byte cache block). The default cache model in Kite implements an 1KB direct-mapped cache of 8-byte cache blocks. Other cache configurations such as set-associative caches can be easily extended from the default cache model, but they are not included in the baseline code for students to work as programming assignments.

- `br_predictor.h/cc` files are used when the optional branch prediction is enabled. They define two classes, `br_predictor_t` and `br_target_buffer_t`, representing a branch predictor and branch target buffer (BTB), respectively. The member functions of both branch predictor and BTB are left nearly empty, again for students to work as programming assignments. A default branch predictor makes always-not-taken branch predictions, and thus the BTB are not really used in this case.

- `proc.h/cc` files implement a five-stage processor pipeline. The processor is defined as `proc_t` class, and it includes datapath components including encompassing memory (`inst_memory_t`), register file (`reg_file_t`), ALU (`alu_t`), data cache (`data_cache_t`), data memory (`data_memory_t`), branch predictor (`br_predictor_t`) and BTB (`br_target_buffer_t`), and four pipeline registers (`pipe_reg_t`) of IF/ID, ID/EX, EX/MEM, and MEM/ WB. The `run()` function of `proc_t` executes the five-stage pipeline in a while-loop that repeats as long as there are in-flight instructions in the pipeline. Note that the pipeline stages are executed backwards from the writeback to fetch stages, not from the fetch to writeback stages. Each stage function (e.g., `fetch()`, `decode()`) defines operations to perform at the corresponding stage. At the end of a program execution, `proc_t` prints out the summary of execution results including the total number of clock cycles, stalled cycles, and executed instructions, followed by cycles per instruction, number of pipeline flushes and branch prediction accuracy (if branch prediction is enabled), and data cache statistics. The final states of registers and data memory are also displayed. For the data memory, only accessed memory addresses and their final values are printed, and other untouched addresses are silenced.

6. Contact

In case you notice a bug or have a question regarding the use of Kite, please contact Prof. William Song by email at wjhsong@yonsei.ac.kr.

References

[1] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA," *SiFive Inc.* and *University of California, Berkeley*, June, 2019.

[2] D. Patterson and J. Hennessey, "Computer Organization and Design RISC-V Edition: The Hardware Software Interface," *Morgan Kaufmann*, 1st Ed., Apr. 2017.