Kite: An Architecture Simulator for RISC-V Instruction Set

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

April 2020 (Version 1.7)

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. Simulator users can easily compose RISC-V assembly programs and execute them through the provided pipeline model for entry-level architectural 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 the usages of architecture simulations with the simple, easy-to-use framework. If you join the computer architecture world for your career (either product development or research), you will certainly have to use some sort of architecture simulators for your work or research. 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 moment for you 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), Ubuntu 18.04 (Bionic Beaver), Mac OS 10.14 (Majave), and Mac OS 10.15 (Catalina). The latest release of Kite is v1.7. To obtain a copy of Kite v1.7, use the following `git` command in terminal. Then, enter the `kite/` directory and build the simulator using a `make` command.

```
$ git clone --branch v1.7 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, 400(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]. Instructions in the program code are sequentially loaded from top to bottom unless branch or jump instructions alter the next PC to fetch. Note that labels, instructions, and register operands are case-insensitive, which means that `beq`, `Beq`, `BEQ` are all identical. The program ends when the next PC naturally goes out of code segment where there exists no valid instruction. The PC value of zero (PC = 0) is reserved as invalid. The current version of Kite supports the following list of RISC-V basic instructions. Instructions in the table are sorted in alphabetical order in each type. Pseudo instructions (e.g., `nop`, `mv`) 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`
	`slli rd, rs1, imm`	`rd = rs1 << imm`
	`srai rd, rs1, imm`	`rd = rs1 >> imm`
	`srli rd, rs1, imm`	`rd = (uint64_t)rs1 << imm`
	`ld rd, imm(rs1)`	`rd = memory[rs1 + imm]`
	`ori rd, rs1, imm`	`rd = rs1 | imm`
	`xori rd, rs1, imm`	`rd = rs1 ^ imm`
S type	`sd rs2, imm(rs1)`	`memory[rs1 + imm] = rs1`
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)`
UJ type	`jal rd, imm`	`rd = pc + 4, pc = pc + imm<<1`
No type	`nop`	No operation

The exemplary program code shown above implements an 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 easier understanding. 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 400 (or at index 50 if the data memory is viewed 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[50] = x10;       // exit: sd   x10, 400(x0)
```

Register state refers to the reg_state file that contains the state (i.e., values) of 32 integer registers of RISC-V. 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
x0 = 0
x1 = 0
x2 = 8
x3 = 0

...

x10 = 21
x11 = 15

...

x31 = 0
```

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 throws an error.

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. The following shows an example of memory state file.

```
# Kite memory state
0 = 0
8 = 4
16 = 5
24 = 0
32 = 4

...
```

The baseline code of Kite creates an 1KB data memory, and its size is easily modifiable in proc_t::init() in the proc.cc file. 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` cannot be fully supported. But, the byte-granular instructions are not in the list of supported instructions anyways. This restriction may 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 in the 1KB space. Values of unlisted memory addresses will be set to zero (i.e., 64-bit zero) by default.

4. Running Kite Simulations

Running a Kite simulation invokes a program code associated with register and memory states. After a program run, 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 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: 1.7                                        *
*******************************************************

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):
(400) = 3
```

The default pipeline model of Kite does not include branch prediction and data forwarding features. The pipeline stalls at every encounter of 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 re-compiling 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: 1.7                                        *
*******************************************************

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
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 progress 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: 1.7                                        *
*****************************************************

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 reveals the details of instruction executions along 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 corresponding instruction. The remainder of debug message follows the instruction format. Since the last argument of `beq` instruction is an immediate value (i.e., offset to a branch target in unit of two bytes), the debug message shows `10(exit)` which means that the immediate value is `10` and it replaces the `exit` label in the program code. 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. 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 implementation of Kite codes in case you have to modify them. You can find that Kite is comprised of following files. Explanations of them will follow in an order better to explain 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` apparently includes a `main()` function that creates, initializes, and runs Kite. `defs.h` declares the `enum` list 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. Importantly, Kite supports multi-cycle ALU executions. `defs.h` defines that `div`, `divu`, `mul`, `remu`, and `remu` 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 an instruction class used in Kite. RISC-V instructions in a program code (e.g., `program_code`) are read when a simulation initiates, and the assembly instructions of character strings are converted to `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`), memory addresses (e.g., `memory_addr`), control directives (e.g., `alu_latency`, `rd_ready`). The instructions are initially stored in instruction memory, and they are fetched and executed by being passed from a pipeline stage to another.

`inst_memory.h/cc` files implement an instruction memory model. It provides processor pipeline with instructions indicated by program counter (PC). Precisely, the instruction memory creates a copy of an instruction and returns a pointer to it. Thus, the processor pipeline works on the pointer to instruction rather than actual copy of it. When a simulation starts, instruction memory loads a program code (e.g., `program_code`) that contains RISC-V assembly instructions. It parses the program code and stores the instructions in the `inst_t` class. Instruction memory keeps supplying the processor pipeline with instructions until the PC goes out of code segment where there exists no valid instruction. Note that PC = 0 is reserved as invalid, and it makes the pipeline stop fetching instructions.

`pipe_reg.h/cc` define a class `pipe_reg_t` that models a pipeline register connecting two adjacent stages such as IF/ID. The class has four functions to manipulate the pipeline register. `read()` function probes the pipeline register and returns a pointer to an instruction if it holds one. This function does not automatically remove the instruction from the pipeline register, instead it requires explicitly invoking `clear()` function to remove the instruction from it. `write()` function writes an instruction in the pipeline register, and `is_available()` tests if the pipeline register is free and thus contains no instruction inside.

`reg_file.h/cc` implements a register file in a class named `reg_file_t`. The register file includes 32 64-bit integer registers (i.e., `x0-x31`). It can be read or written via `read()` or `write()` functions by specifying a register number to access. When a simulation starts, the register file load the state file (i.e., `reg_state`) that contains the initial values of registers. Notably, dependency checking logic is embedded into 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 latest 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` model an arithmetic-logical unit (ALU) that executes instructions based on their operation types. To be type-safe, operands of unsigned integers such as `divu` are cast to `uint64_t` to produce correct results. It is assumed that instructions cannot be pipelined within an ALU, or otherwise out-of-order executions may possibly occur. Since some instructions (e.g., `div`, `mul`) take multiple cycles to run by the ALU, subsequent instructions following the multi-cycle instructions stall until the ALU becomes free.

data_memory.h/cc define data memory that holds data values. When a simulation starts, the data memory loads the state file (i.e., memory_state) to set memory addresses to contain defined values. To align with 64-bit registers, 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 1KB, and the size is easily modifiable by entering a different size in data_memory_t constructor. All the memory addresses of load and stores instructions must fall into the defined address space.

data_cache.h/cc include a data cache that contains a subset of memory blocks. Although the size of memory block is 8 bytes in the data memory to align with 64-bit registers, a cache block can be of any length as far as it is greater than 8 bytes and a power of two. The default cache model in Kite implements 1KB direct-mapped cache of 8-byte cache blocks, but it also has been tested with other configurations such as set-associative caches with different-sized cache blocks. However, those models are not included in the baseline code for students to work as a programming assignment.

br_predictor.h/cc define two classes, br_predictor_t and br_target_buffer_t, that represent branch predictor and branch target buffer (BTB), respectively. The functions of both branch predictor and BTB are left nearly empty, again for students to work as a programming assignment. The default branch predictor always guesses that branches are not taken, and thus the BTB are not really utilized in the simulation.

proc.h/cc implement a processor pipeline model. The processor is defined as proc_t class that contains datapath components including instruction memory (inst_memory_t), register file (reg_file_t), ALU (alu_t), data cache (data_cache_t) and 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 as long as there are in-flight instructions in the pipeline. Note that the pipeline stages are executed backwards from writeback to fetch stages, not from fetch to writeback stages. Each stage function such as fetch() or decode() defines the behaviors of corresponding stage to process incoming instructions. At the end of program execution, proc_t prints out the summary of execution results including 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 such as number of loads, stores, writebacks, and miss rate. Following the pipeline statistics shows the final state of register file and data memory. For the data memory, only accessed memory addresses are printed, and other untouched addresses are not shown.

6. Contact

In case you notice a bug or have a question regarding the use of Kite simulator, please feel free to contact Dr. William Song via email, 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.