Kite: Architecture Simulator for RISC-V Instruction Set

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

April 2023 (Version 1.11)

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 for educational purposes 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 the precise timing model described in the book. It supports most of the basic instructions introduced in the book, such as `add`, `slli`, `ld`, `sd`, and `beq` instructions. Users can easily compose RISC-V assembly programs and execute them through the pipeline model for entry-level architecture studies. Kite implements several basic functionalities, including instruction dependency checks (i.e., data hazards), pipeline stalls, data forwarding (optional), branch predictions (optional), data cache structures, etc. The remainder of this document describes the Kite implementation and its usage.

2. Prerequisite, Download, and Installation

Kite implements the five-stage pipeline model in C++. Since most computer architecture simulators are written in C/C++, it gives students hands-on experiences in computer architecture simulations using a simple, easy-to-use framework. The simple implementation of Kite is easy to install. It requires only a g++ compiler to build and does not depend on other libraries or external tools. It was 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 v1.11. To obtain a copy of Kite v1.11, use the following `git` command in the terminal. Then, enter the `kite/` directory and build the simulator using a `make` command.

```
$ git clone --branch v1.11 https://github.com/yonsei-icsl/kite
$ cd kite/
$ make -j
```

3. Program Code, Register and Memory States

Kite takes three input files to run, i) program code, ii) register state, and iii) data memory state. A program code is a text file containing RISC-V assembly instructions. In the main directory of Kite (i.e., `kite/`), you can find an example code named `program_code` that has the following RISC-V instructions.

```
# 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)
```

A # symbol is reserved for comment in the program code. Everything after the # sign until the end of the line is regarded as a comment and not read. Each instruction in the program code is sequentially stored in the memory, starting from PC = 4. For example, `beq` is the first instruction in the example code above, given the PC value of 4. The next instruction, `remu`, has PC = 8 because every RISC-V instruction is 4 bytes long [1]. The code size is 24 bytes for six instructions, spanning the lowest-address memory region from 4 to 28. Data access to the code segment is forbidden and will throw an error.

Instructions in the program code are sequentially executed from the top unless a branch or jump instruction alters the next PC. The program ends when the next PC naturally goes out of the 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. Labels, instructions, and register operands are case-insensitive. It means that `beq`, `Beq`, `BEQ` are all treated identically. The current version of Kite supports the following RISC-V instructions. Instructions in the table are listed in 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`
	`ld rd, imm(rs1)`	`rd = memory[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`
	`ori rd, rs1, imm`	`rd = rs1 | imm`
	`xori rd, rs1, imm`	`rd = rs1 ^ imm`
S type	`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 << 12;`
UJ type	`jal rd, imm`	`rd = pc + 4, pc = pc + imm<<1`
No type	`nop`	No operation

The program code shown in the previous page implements Euclid's algorithm that finds the 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 when `x11 == 0`. The last code line stores the `x10` value in the data memory at address `2000` (or at index `250` if the 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 initial values of 32 integer registers. The register state is loaded when a simulation starts. The following shows a part of the register state file after the comment lines.

```
# Kite register state
x0 = 0
x1 = 0
x2 = 4096
x3 = 0

...

x10 = 21
x11 = 15

...

x30 = 0
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 executes the instructions 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 the initial values of the data memory. Every address in the memory state file must be a multiple of 8 to align with doubleword data. The following shows an example of the memory state file.

```
# Kite memory state
0 = -31
8 = -131
16 = -134
24 = -421
32 = 59

...
```

Kite creates 4KB data memory by default, and it is modifiable in `proc_t::init()` in `proc.cc`. Each memory block is 8 bytes long, and accesses to the data memory must obey the 8-byte alignment. The 8-byte alignment rule precludes non-doubleword memory instructions such as `lhu` and `sb`. This restriction will be lifted in future releases. In each line of the memory state, 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. The memory state file does not have to list all memory addresses. The values of unlisted memory addresses will be set to zero by default.

The minimum size of the data memory is 4KB, and the lowest-address region is used as a code segment. The size of the code segment is determined by the program code. For instance, an example in the `program_code` file has six instructions. Since every RISC-V instruction is 4 bytes long, the program code is 24 bytes in size. The program code is assumed to span the lowest memory addresses from 4 to 28, and this region is prohibited from data access.

4. Running Kite Simulations

A Kite simulation runs a program code based on the register and memory states. After the program execution, Kite prints out the simulation results, including pipeline statistics (e.g., total clock cycles, number of instructions), cache statistics (e.g., number of loads and stores), and the final state of the register file and data memory. The following shows the results of the Kite simulation executing `program_code`.

```
$ ./kite program_code

*********************************************************
* Kite: Architecture Simulator for RISC-V Instruction Set *
* Developed by William J. Song                          *
* Intelligent Computing Systems Lab, Yonsei University    *
* Version: 1.11                                         *
*********************************************************

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 = 4096
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 does not make branch prediction and data forwarding. The pipeline is stalled on every conditional branch instruction until its outcome is resolved. The lack of data forwarding also stalls the pipeline when an instruction depends on its preceding instructions. The pipeline stall is lifted as soon as the preceding instructions write their results to the 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) can be observed as follows.

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

*********************************************************
* Kite: Architecture Simulator for RISC-V Instruction Set *
* Developed by William J. Song                          *
* Intelligent Computing Systems Lab, Yonsei University    *
* Version: 1.11                                         *
*********************************************************

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 = 4096
x3 = 0

...
```

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

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

*********************************************************
* Kite: Architecture Simulator for RISC-V Instruction Set *
* Developed by William J. Song                          *
* Intelligent Computing Systems Lab, Yonsei University    *
* Version: 1.11                                         *
*********************************************************

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 the 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 clock cycle #1, and `fetch` means that the instruction is currently in the fetch stage. `pc=4` shows the PC of the `beq` instruction. The remainder of the debug message shows the instruction format. The last argument of the `beq` instruction is an immediate value, which is the distance to the branch target in the unit of two bytes. The `beq` instruction in `program_code` branches to label `exit`, which is translated to immediate value 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 advances to the decode stage at cycle #2. The value of a destination operand register (if any) becomes available when an instruction reaches the execute stage. As explained, turning on the debugging option shows the detailed progress of instruction executions in the pipeline.

5. Implementation

This section explains more advanced content about the Kite implementation. Kite consists of the following files in the `kite/` directory.

```
$ 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` has the `main()` function that creates, initializes, and runs Kite.
- `defs.h` defines 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. Kite supports multi-cycle ALU executions, and `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 the 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 starts. Each assembly instruction in the character string is converted into the `inst_t` class that carries the instruction information, including the 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 the instruction memory. It supplies instructions to the processor pipeline, directed by the program counter (PC). In particular, the instruction memory creates a copy of an instruction and returns a pointer to it when fetching it. The processor pipeline works on the instruction pointer instead of a copy since passing the pointer is much 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 using the `inst_t` class. The instruction memory provides 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, which makes the pipeline stop fetching instructions. The size of the code segment is determined by the program code.
- `pipe_reg.h/cc` files define the `pipe_reg_t` class that models a pipeline register connecting two neighboring stages such as IF/ID. The class has the following four functions. 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. Removing the instruction from the pipeline register requires an explicit invocation of the `clear()` function. The `write()` function writes an instruction to the pipeline register, and `is_available()` tests if the pipeline register has no instruction inside and thus is free.

- `reg_file.h/cc` files implement a register file, `reg_file_t`. The register file contains 32 64-bit integer registers (i.e., `x0`-`x31`). When a simulation starts, the register file loads a state file (i.e., `reg_state`) that defines the initial values of the registers. Notably, dependency-checking logic is embedded in 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 misprediction 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 support type-aware operations, the operand of an unsigned instruction such as `divu` is cast to `uint64_t` to produce the correct result. It is assumed that instructions cannot be pipelined inside an ALU since doing so can potentially cause out-of-order executions. If the ALU gets a multi-cycle instruction such as `div` or `mul`, all subsequent instructions stall until the ALU becomes free.

- `data_memory.h/cc` files implement the 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 the defined values. To align with 64-bit registers, the data memory requires that the memory address of a load or store instruction is a multiple of 8. The default size of the data memory is 4KB, which is modifiable in `data_memory_t()`. Memory addresses belonging to the code segment are inaccessible because the code segment is prohibited from data access.

- `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 implements a 1KB direct-mapped cache with 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 branch predictor and BTB are left nearly empty, again for students to work as programming assignments. The default branch predictor makes always-not-taken branch predictions, so the BTB is not technically used.

- `proc.h/cc` files implement a five-stage processor pipeline. The processor is defined as the `proc_t` class, which contains datapath components including the instruction 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`), 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, which repeats as long as there are in-flight instructions in the pipeline. Note that the pipeline stages are executed backward from the writeback to fetch stages, not from the fetch to writeback stages. Each stage function (e.g., `fetch()`, `decode()`) performs necessary operations that need to take place in 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 state of the register file and data memory is also displayed. Only the accessed memory addresses and their final values are printed for the data memory; other untouched addresses are not shown.

6. Contact

If you notice a bug or have a question regarding 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.