EEE3530-01 **Spring 2024**

Assignment 1: RISC-V Assembly

Due: Sunday, Apr. 7, 2024, 11:59PM

1 Installation

- Instructions in this document assume that Ubuntu is already installed on your machine. This document does not cover how to install and use Linux. If you have not installed Linux yet, follow the supplementary materials in `linux_guide.pdf` posted on LearnUs.
- If you just installed a fresh copy of Ubuntu, update the system first before proceeding. Type the following commands in the terminal. The `sudo` commands will require your account password. Updating the system will take a while.

```
$ sudo apt update
$ sudo apt upgrade
```

- Then, install the basic libraries and a C++ compiler. These libraries could have already been installed during the update process.

```
$ sudo apt install build-essential gdb g++
```

- Get a copy of Kite v1.12 from GitHub using the following command. The leading `$` sign indicates that it is a terminal command. Do not type this symbol in the terminal.

```
$ git clone --branch v1.12 https://github.com/yonseicasl/kite
```

- Go to the `kite/` directory, and build it.

```
$ cd kite/
$ make
g++ -g -Wall -O3  -o data_cache.o -c data_cache.cc
g++ -g -Wall -O3  -o reg_file.o -c reg_file.cc
g++ -g -Wall -O3  -o inst_memory.o -c inst_memory.cc

...

g++ -g -Wall -O3  -o kite data_cache.o reg_file.o inst_memory.o br_predictor.o
main.o alu.o proc.o pipe_reg.o inst.o data_memory.o
$
```

- Kite is now ready.

2 Kite: Architecture Simulator for RISC-V Instruction Set

- Kite is a home-grown architecture simulator that models the five-stage pipeline of the RISC-V instruction set.
- This document only presents the minimum information you need for the assignment. For more detailed documentation, refer to `https://github.com/yonseicasl/kite/blob/master/doc/kite.pdf`.
- 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 (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.
- 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, 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.
- The code size is 24 bytes for six instructions, spanning the lowest memory addresses 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 no valid instruction exists or when the next PC is set to zero. PC = 0 is reserved as an invalid address.
- Labels, instructions, and register operands are case-insensitive. For instance, `beq`, `Beq`, `BEQ` are all identical.
- The current version of Kite supports the following RISC-V instructions. 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

- A register state refers to the `reg_state` file containing the initial values of 32 64-bit 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 lists 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. The program code executes the instructions based on the defined register state.
- Since the `x0` register in RISC-V is hard-wired to zero, assigning a non-zero value to it is automatically discarded.
- A memory state refers to the `mem_state` file containing 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

...
```

- Each memory block is 8 bytes long, and access 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`.
- 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 address.
- The default memory size 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.

- A Kite simulation runs the program code based on the register and memory states. After the program execution is done, Kite prints out the simulation results. The following shows the results for executing the example code in `program_code`.

```
$ ./kite program_code

************************************************************
* Kite: Architecture Simulator for RISC-V Instruction Set  *
* Developed by William J. Song                             *
* Computer Architecture and Systems Lab, Yonsei University *
* Version: 1.12                                            *
************************************************************

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

...

x9 = 0
x10 = 3
x11 = 0

...

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

- To trace the line-by-line execution of the program, you can turn on the debugging option and rebuild Kite. Then, executing the program prints out the detailed progress of the program execution.

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

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 : execute : [pc=4] beq x11, x0, 10(exit) [beq 15, 0, 10(exit)] (NT)
4 : memory : [pc=4] beq x11, x0, 10(exit) [beq 15, 0, 10(exit)] (NT)
5 : writeback : [pc=4] beq x11, x0, 10(exit) [beq 15, 0, 10(exit)] (NT)
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]

...
```

- Since you only need the outcome of individual instructions in this assignment, you can use a `grep` command in the terminal to filter the output.

```
$ ./kite program_code | grep writeback

5 : writeback : [pc=4] beq x11, x0, 10(exit) [beq 15, 0, 10(exit)] (NT)
10 : writeback : [pc=8] remu x5, x10, x11 [remu 6, 21, 15]
11 : writeback : [pc=12] add x10, x11, x0 [add 15, 15, 0]
13 : writeback : [pc=16] add x11, x5, x0 [add 6, 6, 0]
14 : writeback : [pc=20] beq x0, x0, -8(loop) [beq 0, 0, -8(loop)] (T)
18 : writeback : [pc=4] beq x11, x0, 10(exit) [beq 6, 0, 10(exit)] (NT)
23 : writeback : [pc=8] remu x5, x10, x11 [remu 3, 15, 6]
24 : writeback : [pc=12] add x10, x11, x0 [add 6, 6, 0]
26 : writeback : [pc=16] add x11, x5, x0 [add 3, 3, 0]

...
```

- The debugging trace shows that the first instruction is `beq x11, x0, exit`. The PC of this instruction is PC = 4, and the values of register operands are `x11 = 15` and `x0 = 0`. Label `exit` is translated to an immediate value of 10, which means that the instruction jumps to five instructions away if the branch is taken.
- Similarly, the second instruction is `remu x5, x10, x11` whose PC = 8 and register values are `x5 = 6`, `x10 = 21`, and `x11 = 15`.

3 Background

- In this assignment, you will write an assembly code that performs the matrix multiplication of C += A * B, where matrix A is $M \times K$ (i.e., rows×columns), B is $K \times N$, and C is $M \times N$ as shown in Fig. 1.

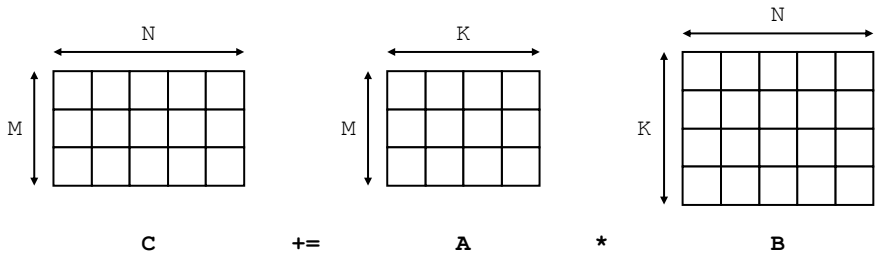

Fig. 1. The matrix multiplication of C += A * B, with M, N, K dimensions.

- The most common way of calculating the matrix multiplication follows the illustration in Fig. 2.
  - ① To calculate the first output element, it performs the inner product of the first row of matrix A and the first column of matrix B (i.e., the sum of element-wise multiplications).
  - ② The second output element is calculated similarly by performing the inner product of the first row of matrix A and the second column of matrix B.
  - ③ The rest of the matrix multiplication follows the same procedure.
- Notably, this method sweeps a row and column of input matrices, A and B, for the same output element. Such a calculation method is called *output-stationary* (OS).

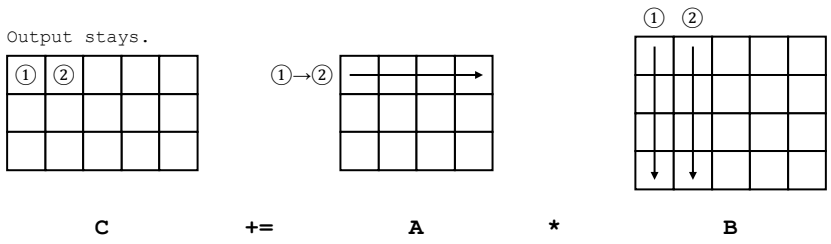

Fig. 2. The matrix multiplication of C += A * B following the output-stationary dataflow.

- In a processor, the output element can be kept in a register while A and B matrices' elements are loaded from the memory. The output value is stored only once in the memory after its calculation is done.
- An alternative to the output-stationary is *input-stationary* (IS) or *weight-stationary* (WS), where an input (A) or weight (B) element is fixed during the multiplication.
- Fig. 3 shows the IS dataflow to calculate the same matrix multiplication.
  - ① The first row of the output matrix is calculated as the vector-dot product of the first input element and the first row of the weight matrix. Since the output row is incomplete, it is called a partial sum.
  - ② The first row of the output matrix is accumulated with the vector-dot product of the second input element and the second row of the weight matrix.
  - ③ The rest of the matrix multiplication follows the same procedure.

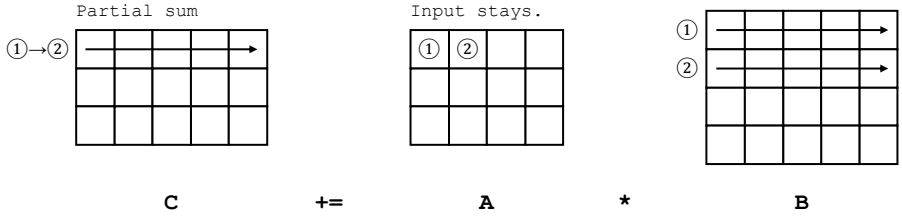

Fig. 3. The matrix multiplication of C += A * B following the input-stationary dataflow.

- The IS method keeps the same input element while it sweeps the row of the output and weight matrices.
- In a processor, the input element is kept in a register while B and C matrices' elements are loaded from the memory. Once the use of the input element is done, it will never be accessed again during the multiplication.
- Instead, it requires writing output elements back to the memory in every iteration because it does not reuse the output values.
- The IS dataflow generates more memory instructions than the OS because of storing partial sums, but it actually performs better (i.e., fewer total clock cycles) since it better utilizes the cache space; related discussions will be covered later in the semester.

4 Implementation

- The goal of this assignment is to practice RISC-V assembly programming using Kite.
- You will have to write a RISC-V assembly code in the `program_code` file that performs the matrix multiplication of C += A * B following the input-stationary dataflow.
- Before starting the assignment, download and execute a script named `assembly.sh` inside the `kite/` directory to update the `program_code`, `reg_state`, and `mem_state` files for this assignment.

- The script will ask you to enter your 10-digit student ID.

```
$ wget https://casl.yonsei.ac.kr/teaching/eee3530/assembly.sh
$ chmod +x assembly.sh
$ ./assembly.sh
Enter your 10-digit student ID:
```

- In the updated `reg_state` file, the `x10 = 1024`, `x11 = 2016`, and `x12 = 3008` registers have the base address of matrix A, B, and C, respectively.

- The `x13`, `x14`, and `x15` registers are initialized with matrix dimensions; M = 5, N = 8, and K = 4.

```
# Kite register state
...

x10 = 1024
x11 = 2016
x12 = 3008
x13 = 5
x14 = 8
x15 = 4

...
```

- The `mem_state` file defines the initial memory state of A, B, and C matrices organized as 1-D arrays in the row-major fashion.

```
# Kite memory state
...

1024 = -3
1032 = 0
1040 = 3
1048 = -1
...

2016 = 1
2024 = -2
2032 = 0
2040 = 3
...

3008 = 0
3016 = 0
3024 = 0
3032 = 0
...
```

- In the mathematical form, the matrix multiplication of C += A * B calculates the following.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} += \begin{bmatrix} -3 & 0 & 3 & -1 \\ 5 & 2 & 0 & -4 \\ 0 & 1 & -5 & 0 \\ 2 & 0 & -4 & 1 \\ -1 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & -2 & 0 & 3 & 0 & 4 & 2 & 4 \\ 0 & -2 & 0 & 2 & -2 & 0 & 1 & -1 \\ -1 & 0 & -4 & 1 & 4 & 3 & 0 & -1 \\ -4 & -1 & 5 & -2 & 0 & 0 & -1 & -1 \end{bmatrix}$$

- In this assignment, you may use any registers to handle intermediate values during the matrix multiplication.

- This assignment does not involve function calls, so there is no need to manipulate stack pointers and keep saved registers and return addresses, etc.

5 Submission

- After implementing the input-stationary matrix multiplication, execute `program_code` using Kite.

```
$ make
$ ./kite program_code
...

Memory state (only accessed addresses):
...

(3008) = -2
(3016) = 7
(3024) = -17
...
```

- You may disregard pipeline stats, such as the total number of instructions and clock cycles in the output.
- As far as the memory state shows the correct matrix multiplication result, the assignment is complete.
- When the assignment is done, execute the `tar.sh` script in the `kite/` directory. It creates a tar file named after your student ID, e.g., `2024143530.tar`. Upload the tar file on LearnUs. Do not rename the file.

```
$ ./tar.sh
$ ls *.tar
2024143530.tar
```

6 Grading Rules

- The following is the general guideline for grading. A 30-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change; a grader may add a few extra rules without notice for a fair evaluation of students' efforts.

  -5 points: The tar file includes redundant tags such as a student name, `hw1`, etc.

  -5 points: The code has insufficient comments. Comments in the skeleton code do not count. You must clearly explain what each part of your code does.

  -5 points: The matrix multiplication code is not written in the `program_code` file.

  -15 points: Matrix multiplication does not follow the input-stationary dataflow.

  -25 points: The program code does not produce a correct multiplication result. However, the values are close, and `program_code` shows substantial efforts.

  -30 points: No or late submission.

  Final grade = F: The submitted tar file is copied from someone else. All students involved in the incidents will get Fs for the final grade.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect, discuss your concerns with the TA. Always be courteous when contacting the TA. If no agreements are made between you and the TA, elevate the case to the instructor to review your assignment. Refer to the course website for the contact information of the TA and instructor: `https://casl.yonsei.ac.kr/eee3530`
- Arguing for partial credits for no valid reasons will be regarded as a cheating attempt; such a student will lose the assignment scores.