CS-470: Homework Set #1
A Cycle-by-Cycle Simulator of an Out-of-Order Processor

1 Introduction

This homework consists in writing a simulator of a simple out-of-order processor called OoO470. The processor runs a minimal subset of the RISC-V instruction set and is implemented with internal data structures similar to the MIPS R10000, studied in Lab #1. The simulation is meant to be cycle-exact and must update the internal data structures precisely as the real processor. The simulator reads an assembly program; it dumps the internal data structures at the end of every cycle until the complete execution terminates. For details on which language you can use, please refer to Section 6.1.

2 Instruction Set

Our processor has thirty-two 64-bit general purpose registers, named `x0` to `x31`. For simplicity, we only consider arithmetic instructions. The processor implements the following subset of the RISC-V instruction set:

Mnemonic	Semantic
`add dest, opA, opB`	dest = opA + opB
`addi dest, opA, imm`	dest = opA + (signed) imm
`sub dest, opA, opB`	dest = opA - opB
`mulu dest, opA, opB`	dest = (unsigned) opA * (unsigned) opB
`divu dest, opA, opB`	dest = (unsigned) opA / (unsigned) opB
`remu dest, opA, opB`	dest = (unsigned) opA % (unsigned) opB

Here is a sample input JSON file:

```
[
  "add x0, x1, x2",
  "addi x1, x2, 10",
  "sub x2, x3, x4",
  "mulu x1, x2, x3",
  "divu x1, x2, x3",
  "remu x1, x2, x3"
]
```

The file defines an array containing instructions whose PC starts from 0 and increases by 1 for each instruction. For instance, the PC of `add` is 0 and the PC of `addi` is 1.

Among all instructions in the list, both `divu` and `remu` can trigger an exception when the divisor operand is zero.

The processor handles exceptions precisely. That is, when an instruction triggering an exception is being committed, the processor should do the following:

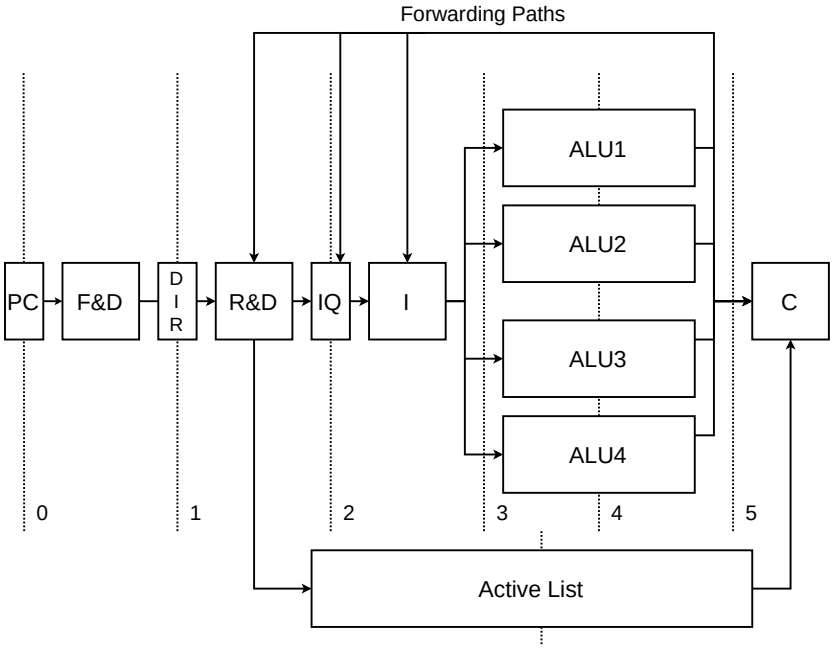

Figure 1: The pipeline structure of the processor. Dashed lines suggest where the pipeline registers and clocked data structures are placed. We denote by *PC* the **Program Counter**, *F&D* the **Fetch and Decode** stage, *DIR* the **Decoded Instruction Register**, *R&D* the **Rename and Dispatch stage**, *IQ* the **Integer Queue**, *I* the **Issue stage**, *ALU* the **Artithmetic Units**, and finally by *C* the **Commit** stage.

- Set the Exception Program Counter (ePC) to the instruction trigger exception.
- The architectural registers should be the same as an in-order processor.
- The program counter is set to `0x10000`.

The simulation should be stopped either when all instructions are committed or when an exception is detected and exception recovery (see more in Section 3.5) is finished.

3 Microarchitecture

Our processor contains 64 physical 64-bit registers and its high level structurs is displayed in Figure 1. The **Fetch and Decode** unit takes four instructions per cycle or less. Fetching and decoding an instruction takes a single cycle. The **Rename and Dispatch** unit takes the four instruction per cycle coming from the Fetch and Decode unit, performs the renaming in a single cycle, and allocate entries in both the **Active list** and the **Integer Queue**. The **Issue** unit picks instructions with all operands ready, and issue them to available ALUs. There are four **Arithmetic** units (**ALUs**) and they can all start a new operation every cycle. The latency of all arithmetic operations is two cycles. The **Commit** unit can commit up to four instructions. All pipeline stages are displayed in Figure 1 with dashed lines. The **Active List** is accessed by both the **Rename and Dispatch** and the **Commit** stage, and therefore has its own register.

3.1 Fetch and Decode Stage

The **Fetch and Decode** stage gets instructions from program memory and passes them to the **Rename and Dispatch** stage, if it accepts new instructions (i.e., does not apply backpressure). Initially, the **PC** is reset to 0 and incremented at each cycle by the number of instructions that are actually fetched and decoded in the cycle. Up to four instructions per cycle are fetched from program memory and decoded. If there are less than four instructions in the program, only those available are processed. If the **Rename**

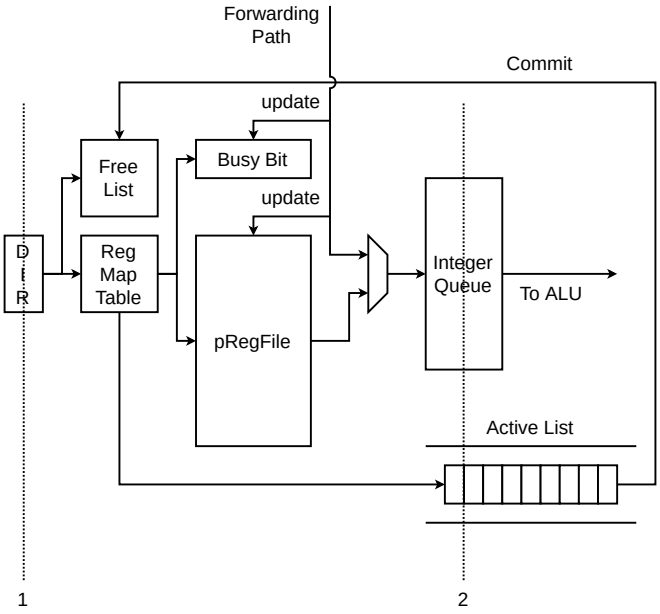

Figure 2: Important data paths in the **Rename and Dispatch** stage. Dashed lines show the same pipeline registers and clocked data structures in Figure 1. For simplicity, other pipeline stages and the delays are ignored.

and Dispatch stage applies backpressure, no instructions are processed. When the **Commit** stage indicates that an exception is detected, the **PC** is set to 0x10000.

3.2 Rename and Dispatch Stage

Figure 2 shows the data structures and data paths in the **Rename and Dispatch** stage. The function of this stage is as follows:

- Check if there are enough physical registers, enough entries in the **Active List**, and enough entries in the **Integer Queue**. If not, apply back pressure to the previous stage, and no instruction will be renamed and dispatched. In this case, instructions already fetched by the previous stage stay in the Decoded Instruction Register. For simplicity, you can treat the instructions in the Decoded Instruction Register atomically: either all are renamed and dispatched or none.
- If there are enough physical resources, rename the instructions decoded from the previous stage and update the **Register Map Table** and **Free List** accordingly.
- Determine the state of the operands required by each instruction. Each operand can be either (a) ready in the physical register file, (b) ready from the forwarding path, or (c) not produced yet. Similar to MIPS R10000, a **Busy Bit Table** can record whether a physical register is available.
- Allocate newly renamed entries in the **Active List** and the **Integer Queue**. **Integer Queue** entries are allocated after accessing the physical register file, which is similar to the Reservation Station used by the Tomasulo algorithm but differs from what is presented in the R10000 paper.
- Observe the results of all functional units through the forwarding paths and update the physical register file as well as the **Busy Bit Table**.

Similar to MIPS R10000, this processor has 64 physical registers, 32 entries in the **Active List**, and 32 entries in the **Integer Queue**.

3.3 Issue Stage, Execution Stage, and Forwarding Paths

The **Issue** stage issues ready instructions from the **Integer Queue**, which buffers the instructions requiring resolving RAW hazard and structure hazard. Since there are four ALUs, during each cycle, the **Issue** stage scans the **Integer Queue**, picks and issues at most four instructions whose operands are ready. For simplicity, when more than four instructions are ready, the oldest instructions (with smaller PCs) are issued first and the newest ones wait for execution.

Similar to the physical register file, the **Integer Queue** observes the results of all functional units through forwarding paths and updates the status of any corresponding entry accordingly. The **Issue** unit can issue both instructions with all operands noted as ready in the **Integer Queue** and ones with operands not yet ready but provided by a forwarding path. In both cases, any issued instruction is removed from the **Integer Queue** in the next cycle.

All ALUs have two-cycle latency and consist of the **Execution** stage. Accordingly, there are four forwarding paths. Results will be broadcast to the forwarding path on the second ALU cycle, driven by the pipeline register 4 in the figure 1.

3.4 Commit Stage

Besides the **Rename and Dispatch** stage, the **Commit** stage also maintains the **Active list** by (1) marking instructions done or exception on receiving results from forwarding paths, (2) retiring or rolling back instructions, and (3) recycling physical registers and push them back to the **Free List**.

During each cycle, the Commit unit scans the **Active list** in program order and picks instructions for retirement until any of the following happens:

- four instructions are already picked,
- an instruction is met that is not completed yet, or
- an instruction is met that is completed but triggers an exception. In this case, the processor will enter the Exception mode in the next cycle.

The Commit unit then removes instructions picked for retirement from the **Active list** and accordingly frees their old destination physical register.

3.5 Exception Recovery

The processor enters the **Exception Mode** once a completed instruction triggering an exception is being committed in the current cycle. The processor tracks the exception status in the **Exception Flag** register. During the **Exception Mode**, the processor halts fetching and decoding and rolls back all instructions following the exception instruction.

More precisely, in the **Exception Mode**, the **Commit** stage should do this:

- Record the PC of the instruction with the exception in the Exception PC register and set the Exception Flag register.
- Notify the **Fetch and Decode** stage that no instruction should be decoded and supplied during the Exception Mode. The **Fetch and Decode** stage should set the PC to 0x10000 and clear the Decoded Instruction Register on the same cycle.
- Reset the **Integer Queue** and the **Execution** stage.
- Similar to MIPS R10000, in every cycle, pick (up to) 4 instructions in reverse program order from the **Active list** bottom, and use it to recover the **Register Map Table**, the **Free List**, and the **Busy Bit Table**.

The **Commit** stage quits the Exception Mode when the **Active List** becomes empty.

3.6 Implementation Details

Here is some information regarding resource allocation as well as the data structure update timing:

- All data structures (the physical register file, the **Active list**, the **Integer Queue**, the **Register Map Table**, the **Free List**, and the **Busy Bit Table**) are implemented in asynchronous-read register files. This means that, after applying a read address, you can get a result in the same cycle. Another way of expressing this point is that the read path through any of these data structures is a combinational path. For instance, in the same cycle, you can access the **Busy Bit Table** and, based on its result, decide if you can use the value from the physical register file or wait for the value to become available.
- The updates to all queues (the **Active list**, the **Integer Queue**, and the **Free List**) in a cycle can be used by the incoming instructions in the same cycle. Another way of expressing this point is that there is a combinational path between the write ports and the read ports of all queues. For instance, if no other physical registers are available, and the **Commit** stage just releases four physical registers in the current cycle, you can immediately allocate these released registers for the incoming decode instructions without waiting for the next cycle.
- There is a strict order of physical registers when recycling them. For normally committed instruction, the physical register attached to the instruction that was earlier in program order will be earlier appended into the **Free List**. For exception recovery, the physical register attached to the instruction that was later in program order, will be earlier appended into the **Free List**.

4 Data Structures

The microarchitectural state of the processor is completely defined by the following data structures, closely mimicking hardware storage:

- Program Counter
- Physical Register File
- Decoded Instruction Register
- Exception Flag
- Exception Program Counter
- Register Map Table
- Free List
- Busy Bit Table
- Active List
- Integer Queue

You are not required to log the values of the pipeline registers 3 and 4, as well as forwarding paths. You can define your own data structure for them. For instance, you may consider using a FIFO to model a shift register used by the two-cycle delay of the **ALUs** stage. **No other data structure is allowed.** At the beginning of every cycle, you have to need to output in a JSON file **only** the state of these data structures.

We describe each of the JSON fields in the next sections. We also provide a simple program together with the expected JSON object from the reference simulator for you to double check your understanding of the output format.

Additionaly, we provide a visualisation tool to help you with the debugging process and make it available on Moodle. You can load any JSON output in it and get a cycle by cycle evolution of state of the processor.

4.1 Program Counter

The Program Counter is an unsigned integer pointing to the next instruction to fetch. Its value is initialized to zero on reset.

In the JSON object, it takes an entry named `PC`:

```
"PC": 0,
```

4.2 Physical Register File

The physical register file is an array with 64 elements. Each element is an unsigned 64-bit integer. All physical registers are initialized to zero on reset.

In the JSON object, it takes an entry named `PhysicalRegisterFile`, whose value type is an array with 64 elements.

```
"PhysicalRegisterFile": [
    0, 0, 0, 0, ...
],
```

4.3 Decoded Instruction Register

The Decoded Instruction Register is an array that buffers instructions that have been decoded but have not been renamed and dispatched yet. It is basically register 1 in Figure 1. On initialization, the value of the array is empty since no instruction is waiting for renaming and dispatching.

In the JSON object, you are only required to record the PC of instructions in this register. Therefore, the Decoded Instruction Register is recorded with the key `DecodedPCs`:

```
"DecodedPCs": [
    0, 1, 2, 3
],
```

4.4 Exception Flag and Exception PC

The Exception Flag is a Boolean variable, indicating whether the processor is under the Exception Mode. While the Exception PC is an unsigned integer storing the value of the program counter of the instruction that triggers an exception. On initialization, the value of the Exception Flag is false and the value of the Exception PC is set to 0.

In the JSON object, the Exception Flag is recorded with the key `Exception` and the Exception PC is recorded with the key `ExceptionPC`:

```
"ExceptionPC": 0,
"Exception": false,
```

4.5 Register Map Table

The **Register Map Table** records the mapping relationship between architectural and physical registers. Since we have 32 architectural registers, the table is an array with 32 elements. On initialization, all architectural registers are mapped to physical registers with the same id. For instance, Register `x0` is mapped to `p0`. Therefore, the **Register Map Table** is initialized to an array containing 32 integers from 0 to 31.

In the JSON object, the **Register Map Table** is recorded with the key `RegisterMapTable`:

```
"RegisterMapTable": [
    0, 1, 2, ...
],
```

4.6 Free List

The **Free List** keeps all physical registers that are available for renaming. It is a FIFO structure, but we use an array to represent it. On initialization, physical registers from `p32` to `p63` are free, so the **Free List** should be reset to an array with 32 elements from 32 to 63.

In the JSON object, the **Free List** is recorded with the key `FreeList`:

```
"FreeList": [
    32, 33, 34, ...
],
```

4.7 Busy Bit Table

The **Busy Bit Table** indicates whether the value of a specific physical register will be generated from the **Execution** stage, so it is a table with 64 Boolean values. On initialization, no physical registers have their value being generated because no instruction is in the pipeline. As a result, the **Busy Bit Table** should be an array with 64 `false`.

In the JSON object, the **Busy Bit Table** is recorded with the key `BusyBitTable`:

```
"BusyBitTable": [
    false, false, false, ...
],
```

4.8 Active List

The **Active list** keeps all instructions renamed and being executed. It is modeled with an array, and every entry records the information of a specific instruction. On initialization, the **Active list** should be empty. Meanwhile, it is never possible to see more than 32 entries in the **Active list**.

In the JSON object, the **Active list** has a key named `ActiveList`, with the value an array with some JSON objects representing its entries:

```
"ActiveList": [
    {
        "Done": false,
        "Exception": false,
        "LogicalDestination": 0,
        "OldDestination": 0,
        "PC": 0,
    },
],
```

The meaning of each key in the object is explained in the following:

- `Done`: Whether the instruction is ready to commit.
- `Exception`: Whether the instruction will trigger an exception.

- `LogicalDestination`: The id of the destination architectural register. For instance, instruction `add x0, x1, x2` uses `x0` as its destination. When allocating an entry for this instruction, this field should be 0, which represents `x0`.
- `OldDestination`: The id of the physical register previously mapped to the destination architectural register. For instance, instruction `add x0, x1, x2` uses `x0` as its destination. Before renaming, the `x0` is mapped to `p0`, so the value of this field is 0.
- `PC`: The PC of the instruction.

4.9 Integer Queue

The **Integer Queue** keeps all instructions awaiting issuing. It is also modeled with an array and every entry records the operation information of a specific instruction. On initialization, the **Integer Queue** should be empty. Similar to the **Active list**, it is impossible to have more than 32 entries in the **Integer Queue**.

In the JSON file, the **Integer Queue** has a key named `IntegerQueue`, and its value is an array with some JSON objects representing its entries:

```
"IntegerQueue": [
    {
        "DestRegister": 32,
        "OpAIsReady": true,
        "OpARegTag": 0,
        "OpAValue": 0,
        "OpBIsReady": true,
        "OpBRegTag": 0,
        "OpBValue": 0,
        "OpCode": "add",
        "PC": 0
    },
],
```

The meaning of each field in the entry is explained in the following:

- `DestRegister`: The id of the destination physical register.
- `OpAIsReady`: Whether the operand A has its value in the entry. This field is true when operand A is read from the physical register file or a **Forwarding Paths** during the **Rename and Dispatch** stage, or from **Forwarding Paths** while waiting in the **Integer Queue**.
- `OpARegTag`: The physical register id of operand A by which the **Integer Queue** can listen to Forwarding Paths and get the value. When the value of operand A is available within the **Integer Queue**, we don't care about the value of this field.
- `OpAValue`: The value of operand A. When the value of operand A is not available yet, we don't care about the value of this field and it can stay blank.
- `OpBIsReady`, `OpBRegTag`, `OpBValue`: Similar fields for the operand B. Note that for instruction `addi`, operand B is the sign-extended immediate.
- `OpCode`: The operation code. It is a string value among these options: `"add"`, `"sub"`, `"mulu"`, `"divu"`, and `"remu"`.
- `PC`: The PC of the instruction.

5 Simulation Hints

In case you have never written any cycle-accurate simulator, here are some general suggestions to help you organize your program. Please consider that the state of the processor is (almost) completely defined

by the data structures (registers) listed in the previous section. This means that essentially your program will implement the combinational logic of the processor in a `propagate` stage and mimic the behaviour of the registers on a clock edge in a `latch` stage. In other words, you are essentially writing the Register Transfer Level implementation of the processor (as you would do in VHDL or Verilog) but hopefully in an extremely friendlier programming language.

During the `propagate` stage, you should first make a copy of all data structures to prepare the next state of the processor. You should then update the value of these copies according to the functionality of all units (which correspond to combinational logic). All units will read the current state of the processor except those that access data structures that can be updated and read in the same cycle (e.g., queues). During the `latch` stage, you can replace the current value of the registers with the copy representing the next state and prepared during the `propagate` stage.

For instance, in terms of the **Free List**, during the `propagate` stage, you should follow these steps:

- Make a copy of the original **Free List**.
- Check the **Active List** to see if any instruction will be committed. Append all free physical registers you collect into the copy.
- Do renaming, where you will pop some physical registers from the copy.

And later, during the `latch` stage, you can simply replace the current value of the **Free List** with the copy you made.

Finally, you can find in Figure 3 a possible pseudocode for the simulator.

```
// 0. parse JSON to get the program
parseInstructions();

// 1. dump the state of the reset system
dumpStateIntoLog();

// 2. the loop for cycle-by-cycle iterations.
while(not (noInstruction() and activeListIsEmpty())){

    // do propagation
    // if you have multiple modules, propagate each of them
    propagate();

    // advance clock, start next cycle
    latch();

    // dump the state
    dumpStateIntoLog();
}

// 3. save the output JSON log
saveLog();
```

Figure 3: Simulator Pseudocode

6 Homework Guidelines

6.1 Programming Language and Running Environment

We provide you with a Ubuntu 22.04 container where your code will be compiled and run from for grading. You have root access on the container. You can run root commands without a password (e.g., running `sudo`). The container image contains the following programming language toolchains:

- C 17 / C++ 17 (gcc, g++)
- Python 3.10 (python3, pip3)
- Go 1.20 (go)
- Rust 1.67 (cargo, rustc)
- Java (OpenJDK 17)
- Scala 3.2.2 (sbt)
- JavaScript (node 18.14, npm)

If your favorite language is not on the list, you can come and talk to the TAs. We will try to accommodate your case to the extent possible.

6.2 Submission Instructions

You must provide a build script with the following name responsible for compiling your code and installing the required library dependencies (if any). If you do not use a compiled language, please provide an empty build.sh file. You can also use `apt` in `build.sh` to install any additional libraries. Please check that your code compiles and runs properly within that environment before the deadline.

```
./build.sh
```

You must also provide a run script with the following interface responsible for running your simulator using the input program (input.json) and producing the output schedule (output.json in the following snippet).

```
./run.sh </path/to/input.json> </path/to/output.json>
```

The two scripts and your code must be contained within a single **zip** file (with both scripts at the root of the archive).

6.3 Testing Guidelines

We provide 9 basic tests cases for you to test your code, which are under the test folder. You can check desc.txt under each folder to see their description.

Assuming you already have a valid build and run script, you can run generate the schedules corresponding to all test cases using.

```
./runall.sh
```

And you can test your code against the reference using.

```
./testall.sh
```

6.4 Reproducing the Grading Environment

We provide you with a docker image containing the environment we will use for grading. Please ensure that your code compiles and runs within the docker environment before submitting.

To install docker on your system please follow the instructions on the docker website. We will assume over the next instructions that you have a valid docker installation. You can check that your docker installation is working with:

```
sudo docker run hello-world
```

The grading environment is defined in a configuration file called Dockerfile. With it, we can build the docker image docker containing the actual grading environment and register it to docker. We can do so with two commands.

From the root of the project (where the Dockerfile image is), to build the docker image simply run.

```
sudo docker build . -t cs470
```

After which, you should see the image under the name cs470 when running.

```
sudo docker image ls
```

You docker setup is now complete. From this point, you only need to run the following command to run the grading environment (from the root of the homework project).

```
sudo docker run -it -v $(pwd):/home/root/cs470 cs470
```

Where we mount the root of the project in /home/root/cs470. You can then check that your code runs in the environment as follows.

```
cd /home/root/cs470
./runall.sh
./testall.sh
```

If you have any questions when installing / running docker, please talk to a TA, we would be happy to help.

6.5 Grading

- We provide a set of tests for you to check the general correctness of your code and of the produced files. Appropriate functional testing is your responsibility. For grading, we use a significantly expanded set of test cases.
- We will generate the output of your code automatically using the docker image and the build and run scripts described above.
- We will check your generated schedules against the reference using the same `compare.py` script as the one used in the `testall.sh` script, but, we will grade your code semiautomatially. If the comparison script fails to find the equivalence between some reference schedule and your schedule, we will manually check the equivalence. We will only manually accept a schedule if it has similar (or better) performance as the reference and if it it produces correct output.
- If your code does not pass a test and you think it is correct but are not sure, you can always come and talk to a TA and we will check the schedule with you.