CSE561/SCE412 Project #1

Modeling Out-of-Order Execution Core

Due 11:59PM, April 30th
TA: Wonkyo Choe

1. Overview

The goal of this assignment is to help you understand the basic organization of out-of-order execution core. Your tasks for this assignment are to 1) model an out-of-order execution core based on issue queue, renaming (map table), reorder buffer (ROB), and 2) analyze a set of traces to find how various ILP parameters (issue width, reservation station size, and ROB size) affect processor performance and how much ILPs are available for the given traces.

2. Microarchitecture to be modeled

A core model for this assignment processes an input instruction trace, and simulates the backend of out-of-order execution core. *We assume the front-end parts (BTB, branch prediction, icache etc) are perfect*, so the front-end can always feed the backend with N instructions (on a correct execution path) every cycle, as long as the back-end can process them. *We also assume that the dcache is also perfect*.

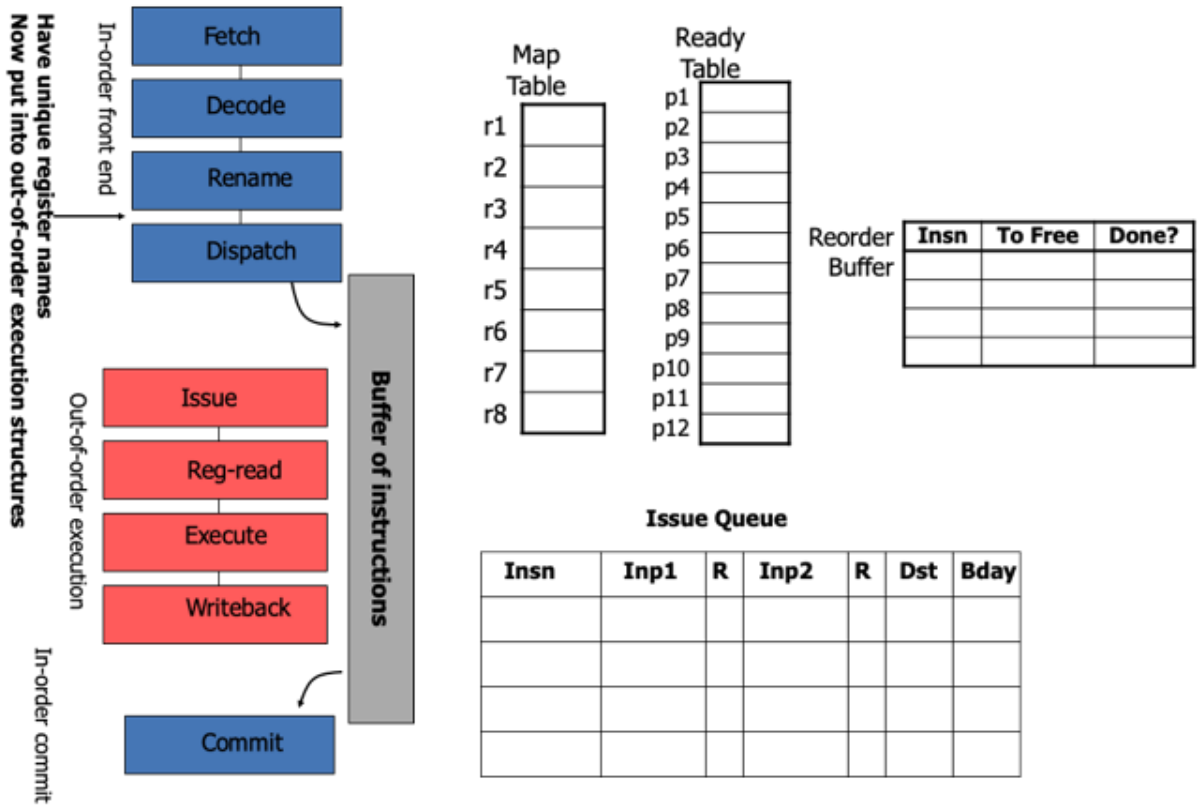

Figure 1.

2.1 Parameters

1. Number of architectural registers: The number of architectural registers specified in the ISA is 67 (`r0` ~ `r66`). The number of architectural registers determines the number of entries in the Rename Map Table (`RMT`). Mostly, the number of physical registers in Ready Table is more than that of architectural registers. In this project, you can simply determine the number of physical registers as twice as that of architectural registers.
2. `WIDTH`: This is the superscalar width of all pipeline stages, in terms of the maximum number of instructions in each pipeline stage. The one exception is Writeback: the number of instructions that may complete execution in a given cycle is not limited to `WIDTH` (this is explained below).
3. `IQ_SIZE`: This is the number of entries in the Issue Queue (`IQ`).
4. `ROB_SIZE`: This is the number of entries in the Reorder Buffer (`ROB`).

2.2 Functional Units

There are `WIDTH` number of *universal, pipelined* function units (FUs). Each FU can execute any type of instruction (hence the term "universal"). The operation type of an instruction indicates its execution latency: Type 0 has a latency of 1 cycle, Type 1 has a latency of 2 cycles, and Type 2 has a latency of 5 cycles. Each FU is fully pipelined. Therefore, a new instruction can begin execution on each FU every cycle.

2.3 Pipeline Registers

The pipeline stages shown in Figure 1 are separated by pipeline registers. In general, this specification names a pipeline register based on the stage that it feeds into. For example, the pipeline register between Fetch and Decode is called DE because it feeds into Decode. A *bundle* is the set of instructions in a pipeline register. For example, if DE is not empty, it contains a *decode bundle*. Table 1-1 lists the names of the pipeline registers used in this spec. It also provides a description of each pipeline register and its size (max # instructions). In Table 1-2, you can find the required queues in modeling the out-of-order execution pipeline.

Table 1-1. Descriptions of the pipeline registers

Pipeline Register	Description	Size (max # instructions)
DE	Pipeline register between the *Fetch* and *Decode* stages	WIDTH
RN	Pipeline register between the *Decode* and *Rename* stages	WIDTH
DI	Pipeline register between the *Rename* and *Dispatch* stages	WIDTH
RR	Pipeline register between the *Issue* and *Register Read* stages	WIDTH
execute_list	execute_list represents the pipeline register between the	WIDTH * 5

	Register Read and *Execute* stages, as well as all sub-pipeline stages within each function unit	There are WIDTH universal function units each with a maximum latency of 5 cycles. Hence, there can be as many as WIDTH*5 instructions in-flight within the Execute stage.
WB	Pipeline register between the *Execute* and *Writeback* stages *To maintain a non-blocking Execute stage, there is no constraint on the number of instructions that may complete in a cycle*	WIDTH * 5 This is a conservative upper bound. If each function unit's 5-stage pipeline is full with a 1-cycle (youngest), 2-cycle, 3-cycle, 4-cycle, and 5-cycle (oldest) operation, then all 5 instructions will complete in the same cycle. Multiply that by the number of such function units (WIDTH).

Table 1-2. Descriptions of the queues

Pipeline Register	Description	Size (max # instructions)
IQ	Instruction queue between the *Dispatch* and *Issue* stages	IQ_SIZE
ROB	Queue guarantees an order of inst until *Commit* stages	ROB_SIZE

A note about register values:
For the purpose of determining the number of cycles it takes for the microarchitecture to run a program, the simulator does not need to use and produce actual register values. This is why the initial Architectural Register values are not provided and the instruction opcodes are omitted from the trace. All that the simulator needs, to determine the number of cycles, is the microarchitecture configuration, execution latencies of instructions (by operation type), and register specifiers of instructions (to determine true, anti, and output dependencies).

2.4 Guide to Implementing your Simulator

This section provides a guide to implementing your simulator. Call each pipeline stage in reverse order in your main simulator loop, as follows. The comments indicate tasks to be performed.

```
do {
       commit();     // Commit up to WIDTH consecutive "ready" instructions from
```

```
                // the head of the ROB. Note that the entry of ROB should be
                // retired in the right order.

writeback();    // Process the writeback bundle in WB: For each instruction in
                // WB, mark the instruction as "ready" in its entry in the ROB.

execute();      // From the execute_list, check for instructions that are
                // finishing execution this cycle, and:
                // 1) Remove the instruction from the execute_list.
                // 2) Add the instruction to WB.

regRead();      // If RR contains a register-read bundle:
                // then process (see below) the register-read bundle up to
                // WIDTH instructions and advance it from RR to execute_list.
                //
                // How to process the register-read bundle:
                // Since values are not explicitly modeled, the sole purpose of
                // the Register Read stage is to pass the information of each
                // instruction in the register-read bundle (e.g. the renamed
                // source operands and operation type).

                // Aside from adding the instruction to the execute_list, set a
                // timer for the instruction in the execute_list that will
                // allow you to model its execution latency.

issue();        // Issue up to WIDTH oldest instructions from the IQ. (One
                // approach to implement oldest-first issuing is to make
                // multiple passes through the IQ, each time finding the next
                // oldest ready instruction and then issuing it. One way to
                // annotate the age of an instruction is to assign an
                // incrementing sequence number to each instruction as it is
                // fetched from the trace file.)
                //
                // To issue an instruction:
                // 1) Remove the instruction from the IQ.
                // 2) Wakeup dependent instructions (set their source operand
                // ready flags) in the IQ, so that in the next cycle IQ should
                // properly handle the dependent instructions.

dispatch();     // If DI contains a dispatch bundle:
                // If the number of free IQ entries is less than the size of
                // the dispatch bundle in DI, then do nothing. If the number of
                // free IQ entries is greater than or equal to the size of the
                // dispatch bundle in DI, then dispatch all instructions from
                // DI to the IQ.

rename();       // If RN contains a rename bundle:
                // If either DI is not empty (cannot accept a new register-read
                // bundle) then do nothing. If DI is empty (can accept a new
                // dispatch bundle), then process (see below) the rename
                // bundle and advance it from RN to DI.
                //
                // How to process the rename bundle:
                // Apply your learning from the class lectures/notes on the
                // steps for renaming:

                // (1) Rename its source registers, and
                // (2) Rename its destination register (if it has one). If you
                // are not sure how to implement the register renaming, apply
```

```
                    // the algorithm that you've learned from lectures and notes.
                    // Note that the rename bundle must be renamed in program
                    // order. Fortunately, the instructions in the rename bundle
                    // are in program order).

        decode();   // If DE contains a decode bundle:
                    // If RN is not empty (cannot accept a new rename bundle), then
                    // do nothing. If RN is empty (can accept a new rename bundle),
                    // then advance the decode bundle from DE to RN.

        fetch();    // Do nothing if
                    // (1) there are no more instructions in the trace file or
                    // (2) DE is not empty (cannot accept a new decode bundle)
                    //
                    // If there are more instructions in the trace file and if DE
                    // is empty (can accept a new decode bundle), then fetch up to
                    // WIDTH instructions from the trace file into DE. Fewer than
                    // WIDTH instructions will be fetched and allocated in the ROB
                    // only if
                    // (1) the trace file has fewer than WIDTH instructions left.
                    // (2) the ROB has fewer spaces than WIDTH.


} while (advance_cycle());
                    // advance_cycle() performs several functions.
                    // (1) It advances the simulator cycle.
                    // (2) When it becomes known that the pipeline is empty AND the
                    // trace is depleted, the function returns "false" to terminate
                    // the loop.
```

3. Instruction Traces

The simulator reads a trace file in the following format:

```
<PC> <operation type> <dest reg #> <src1 reg #> <src2 reg #>
<PC> <operation type> <dest reg #> <src1 reg #> <src2 reg #>
…
```

Where:

- `<PC>` is the program counter of the instruction (in hex).
- `<operation type>` is either "0", "1", or "2".
- `<dest reg #>` is the destination register of the instruction. If it is -1, then the instruction does not have a destination register (for example, a conditional branch instruction). Otherwise, it is between 0 and 66.
- `<src1 reg #>` is the first source register of the instruction. If it is -1, then the instruction does not have a first source register. Otherwise, it is between 0 and 66.
- `<src2 reg #>` is the second source register of the instruction. If it is -1, then the instruction does not have a second source register. Otherwise, it is between 0 and 66.

For example:

```
ab120024 0 1 2 3
```

```
ab120028 1 4 1 3
ab12002c 2 -1 4 7
```

Means:

```
"operation type 0" R1, R2, R3
"operation type 1" R4, R1, R3
"operation type 2" -, R4, R7 // no destination register!
```

4. Simulator Output

The simulator outputs the following measurements after completion of the run:

1. Total number of instructions in the trace.
2. Total number of cycles to finish the program.
3. Average number of instructions retired per cycle (IPC).

The simulator also outputs the timing information for every instruction in the trace, in a format that is used as input to the `scope` tool. The `scope` tool's input format is described in a later section

5. Helping you debug and validate

We provide a `scope` tool that allows you to display pipeline timing diagrams similar to the timing diagrams used in class.

```
Usage: scope <input-file> <output-file>
```

The tool has quite a bit of error checking to make sure you comply with formatting and usage, however, beware it is not error-proof. The input to `scope` must be a text file that encodes the timing of each instruction in the program. Your simulator must dump this timing information and generate the input-file (needed anyway to match validation output). There should be one line for each instruction in the program, and instructions must be dumped in program order. The format of each line is as follows. Note: you must substitute an integer everywhere there is a <> pair.

`<seq_no>` **fu{**`<op_type>`**} src{**`<src1>,<src2>`**} dst{**`<dst>`**} FE{**`<begin-cycle>,<duration>`**} DE{…} RN{…} DI{…} IS{…} RR{…} EX{…} WB{…} CM{…}**

`<seq_no>` is the line number in trace (i.e., the dynamic instruction count), starting at 0. Substitute 0, 1, or 2 for the . , , and are register numbers (include –1 if that is the case). For each of the pipeline stages, indicate the first cycle that the instruction was in that pipeline stage followed by the number of cycles the instruction was in that pipeline stage. The tool automatically does some consistency checks and exits if there is a problem, *e.g.*, begin cycle of DE should equal begin cycle of FE plus duration of FE. The output generated by `scope` contains the timing diagram.

6. Compiling, running, and further validating your simulator

6.1 Cloning the repo to get the traces provided

```
$] git clone https://github.com/csl-ajou/cse561-project1
```

6.2 Running the simulator

Your simulator must accept command-line arguments as follows:

```
$] ./cse561sim <ROB_SIZE> <IQ_SIZE> <WIDTH> <tracefile>
```

6.3 Validating the simulator output

Sample simulation outputs will be provided in the git repository.

Each validation run includes:

1. Timing information for every instruction. The format is described in Section 4.
2. The microarchitecture configuration.
3. All measurements as described in Section 5.

You must run your simulator and debug it until it matches these provided validation outputs. Your simulator must print outputs to the console (i.e., to the screen). Your output must match both numerically and in terms of formatting because the TA will literally "diff" your output with the correct output. Therefore, redirect the console output of your simulator to a temporary file. This can be achieved by placing "`> <your_output_file>`" af
ter the simulator command:

```
$] ./cse561sim <ROB_SIZE> <IQ_SIZE> <WIDTH> <tracefile> > <output_file>
```

You can test whether or not your outputs match the expected output, by running this unix command:

```
$] diff -w <your_output_file> <posted_output_file>
```

The "`-w`" flag tells `diff` to ignore case (uppercase vs. lowercase) and whitespace. Therefore, just make sure there is some whitespace (such as a tab) where you see whitespace in the validation output. If the above command returns without printing anything to the screen, your validation was successful.

7. Hand-in

You are supposed to submit your code (please make an archive file) and a report describing your design and implementation. Also, the report should include the analysis result of how various ILP parameters (issue width, reservation station size, and ROB size) affect processor performance and how much ILPs are available for the given traces.

Note: You can choose the programming language among C, C++, and Python. If you would like to use a different language, please contact the instructor (jsahn@ajou.ac.kr). For the assignment questions, our TA, Wonkyo Choe, can help you in Ajou BB. Please post questions on AjouBB.