Project 3:
Working with the Full Simulator

Version 1.0

ECE 721: Advanced Microarchitecture
Spring 2023, Prof. Rotenberg

Due: Saturday, March 11, 2023, 11:59pm
Late projects will only be accepted until: Saturday, March 18, 2023, 11:59pm

- READ THIS ENTIRE DOCUMENT.
- Academic integrity:
    - **Source code:** Each student must design and write their source code alone. They must do this (design and write their source code) without the assistance of any other person in ECE 721 or not in ECE 721. They must do this (design and write their source code) without searching the web for past semester's projects, which is strictly forbidden. They must do this (design and write their source code) without looking at anyone else's source code, without obtaining electronic or printed copies of anyone else's source code, *etc*.
    - **Explicit debugging:** With respect to "explicit debugging" as part of the coding process (*e.g.*, using a debugger, inspecting code for bugs, inserting prints in the code, iteratively applying fixes, *etc*.), each student must explicitly debug their code without the assistance of any other person in ECE 721 or not in ECE 721.
    - **Sanctions:** The sanctions for violating the academic integrity policy are (1) a score of 0 on the project and (2) academic integrity probation for a first-time offense or suspension for a subsequent offense (the latter sanctions are administered by the Office of Student Conduct). Note that, when it comes to academic integrity violations, both the giver and receiver of aid are responsible and both are sanctioned. Please see the following RAIV form which has links to various policies and procedures and gives a sense of how academic integrity violations are confronted, documented, and sanctioned: RAIV form.
- Reasonable assistance: If a student has any doubts or questions, or if a student is stumped by a bug, the student is encouraged to seek assistance using both of the following channels.
    - Students may receive assistance from the TA(s) and instructor.
    - Students are encouraged to post their doubts, questions, and obstacles, on the Moodle message board for this project. The instructor and TA(s) will moderate the message board to ensure that reasonable assistance is provided to the student. Other students are encouraged to contribute answers so long as no source code is posted.

      * An example of reasonable assistance via the message board: Student A: "*I'm encountering the following assertion/problem, has anyone else encountered something like this?*" Student B: "*Yes, I encountered something similar and you might want to take a look at how you are doing XYZ in your renamer, because the problem has to do with such-and-such.*"

* Another example of a reasonable exchange: Student A: "*I'm unsure how to size my Free List based on the other parameters.*" Instructor/TA/Student B: "*You can reference the lecture notes on this topic but I'll also answer here. The key is that the PRF has a specified number of physical registers and, at any given time, a fixed number of these are committed registers. The number of committed registers is the number of logical registers. The committed registers cannot be free, ever. From that, you should be able to infer an upper bound required for the size of the Free List. For example, if the PRF size is 160 and the # logical registers is 32, then at most there can be 128 free registers. Again, also refer back to the lecture notes.*"

- **The intent of the academic integrity policy is to ensure students code and explicitly debug their code by themselves. It is NOT our intent to stifle robust, interesting, and insightful discussion and Q&A that is helpful for students (and the instructor and TA(s)) to learn together. We also would like to help students get past bugs by offering advice on where they may be doing things incorrectly, or where they are making incorrect assumptions, *etc*., from an academic and conceptual standpoint.**

1. Introduction

In this project, you will learn how to work with the complete simulator. View this as preparation for the research phase of the course, in which you will modify the simulator to carry out your independent research project.

You have three tasks (also see Section 3).

1. Read this document and read through all source files carefully to learn about the simulator.
2. Write segments of code that I intentionally omitted from the simulator.
3. Integrate your renamer module developed in Project 2 into the simulator (renamer.h, renamer.cc). If you did not complete Project 2, you must come see me about fixing the problems with your renamer. If it is beyond repair, I can provide a lib file of a working renamer along with necessary files and instructions for linking with it.

2. class *pipeline_t* (pipeline.h)

The *pipeline_t* class (pipeline.h) models the entire pipeline. It models the canonical pipeline presented in the lecture notes. Moreover, the canonical pipeline is modeled explicitly. In particular:

1. Instructions are explicitly moved through the pipeline, as they would move in a real pipeline. In turn, this means that…
2. Pipeline registers and all queues are explicitly modeled.
3. Execution lanes are explicitly modeled.

2.1. class *payload* (payload.h)

All of the payload information associated with each in-flight instruction is held in a data structure called PAY (class *payload*, payload.h), instantiated within the *pipeline_t* class. PAY does not correspond to a real pipeline structure. Rather, PAY corresponds to payload information generated in pipeline stages and passed forward from one pipeline stage to the next. Instead of moving an instruction's entire payload from one pipeline stage to the next, which would slow

down the simulator tremendously, the simulator moves the instruction's index into PAY, *i.e.*, its index into a buffer called PAY.buf[]. Therefore, if you ever need to reference payload information for an instruction from within a pipeline stage (the logic for each pipeline stage is implemented with a dedicated member function of the *pipeline_t* class), often a local variable called *index* will be present and you can reference the instruction's payload information as follows: PAY.buf[index].* where * is any field of the payload. The struct which defines the payload itself is called *payload_t* (payload.h). So PAY.buf[] is an array where each element is of type *payload_t*, the payload of an individual instruction. Carefully study the comments for *payload_t*, below, to learn about all of the information associated with an instruction in the pipeline. Also take note of which pipeline stages generate which payload information. Alternatively, you can study the file payload.h, directly.

```
typedef struct {

   ////////////////////////
   // Set by Fetch1 Stage.
   ////////////////////////

   insn_t inst;                 // The RISCV instruction.
   reg_t pc;                    // The instruction's PC.
   reg_t next_pc;               // The next instruction's PC. (I.e., the PC of
                                // the instruction fetched after this one.)
   bool branch;                    // This instruction was identified as a branch,
                                   // by the BTB (if bundle came from instr. cache) or
                                   // by the trace cache.
   btb_branch_type_e branch_type;  // If the instruction was identified as a branch,
                                   // this is its type.
   uint64_t branch_target;         // If the instruction was identified as a branch,
                                   // this is its taken target (not valid for indirect
                                   // branches).

   bool good_instruction;       // If 'true', this instruction has a
                                // corresponding instruction in the
                                // functional simulator. This implies the
                                // instruction is on the correct control-flow
                                // path.

   debug_index_t db_index;      // Index of corresponding instruction in the
                                // functional simulator
                                // (if good_instruction == 'true').
                                // Having this index is useful for obtaining
                                // oracle information about the instruction,
                                // for various oracle modes of the simulator.

   // FIX_ME: not currently set/incremented
   uint64_t sequence;           // Unique sequence number for speculatively
                                // fetched instructions.  Helpful for
                                // logging (debug traces).

   ////////////////////////
   // Set by Fetch2 Stage.
   ////////////////////////

   unsigned int pred_tag;       // If the instruction is a branch, this is its
                                // index into the Fetch Unit's branch queue.

   ////////////////////////
   // Set by Decode Stage.
   ////////////////////////
```

```
unsigned int flags;          // Operation flags: can be used for quickly
                             // deciphering the type of instruction.
fu_type fu;                  // Operation function unit type.
cycle_t latency;             // Operation latency (ignore: not currently used).

bool checkpoint;             // If 'true', this instruction is a branch
                             // that needs a checkpoint.

// Note: At present, the decode stage does not split RISCV instructions
// into micro-instructions.  Nonetheless, the pipeline does support
// split instructions.
bool split;                  // Instruction is split into two micro-ops.
bool upper;                  // If 'true': this instruction is the upper
                             // half of a split instruction.
                             // If 'false': this instruction is the lower
                             // half of a split instruction.
bool split_store;            // Instruction is a split-store.

// Source register A.
bool A_valid;                // If 'true', the instruction has a
                             // first source register.
unsigned int A_log_reg;      // The logical register specifier of the
                             // source register.

// Source register B.
bool B_valid;                // If 'true', the instruction has a
                             // second source register.
unsigned int B_log_reg;      // The logical register specifier of the
                             // source register.

// ** DESTINATION ** register C.
bool C_valid;                // If 'true', the instruction has a
                             // destination register.
unsigned int C_log_reg;      // The logical register specifier of the
                             // destination register.

// ** SOURCE ** register D.
// Floating-point multiply-accumulate uses a third source register.
bool D_valid;                // If 'true', the instruction has a
                             // third source register.
unsigned int D_log_reg;      // The logical register specifier of the
                             // source register.

// IQ selection.
sel_iq iq;                   // The value of this enumerated type indicates
                             // whether to place the instruction in the
                             // issue queue, skip it, or skip it with
                             // an exception.
                             // (The 'sel_iq' enumerated type is also
                             // defined in this file.)

uint64_t CSR_addr;           // System register address, for privileged
                             // instructions that reference and/or modify
                             // a specified system register.

// Details about loads and stores.
unsigned int size;           // Size of load or store (1, 2, 4, or 8 bytes).
bool is_signed;              // If 'true', the loaded value is signed,
                             // else it is unsigned.
bool left;                   // Relic of PISA ISA - no longer used.
bool right;                  // Relic of PISA ISA - no longer used.
```

```
    ////////////////////////
    // Set by Rename Stage.
    ////////////////////////

    // Physical registers.
    unsigned int A_phys_reg;     // If there exists a first source register (A),
                                 // this is the physical register specifier to
                                 // which it is renamed.
    unsigned int B_phys_reg;     // If there exists a second source register (B),
                                 // this is the physical register specifier to
                                 // which it is renamed.
    unsigned int C_phys_reg;     // If there exists a ** DESTINATION ** register (C),
                                 // this is the physical register specifier to
                                 // which it is renamed.
    unsigned int D_phys_reg;     // If there exists a third ** SOURCE ** register (D),
                                 // this is the physical register specifier to
                                 // which it is renamed.

    // Branch ID, for checkpointed branches only.
    unsigned int branch_ID;      // When a checkpoint is created for a branch,
                                 // this is the branch's ID (its bit position
                                 // in the Global Branch Mask).

    ////////////////////////
    // Set by Dispatch Stage.
    ////////////////////////

    unsigned int AL_index;       // Index into Active List.
    unsigned int LQ_index;       // Indices into LSU. Only used by loads, stores,
                                 // and branches.
    bool LQ_phase;
    unsigned int SQ_index;
    bool SQ_phase;

    unsigned int lane_id;        // Execution lane chosen for the instruction.

    ////////////////////////
    // Set by Reg. Read Stage.
    ////////////////////////

    // Source values.
    union64_t A_value;           // If there exists a first source register (A),
                                 // this is its value. To reference the value as
                                 // uint64_t, use "A_value.dw".
    union64_t B_value;           // If there exists a second source register (B),
                                 // this is its value. To reference the value as
                                 // uint64_t, use "B_value.dw".
    union64_t D_value;           // If there exists a third ** SOURCE ** register (D),
                                 // this is its value. To reference the value as
                                 // uint64_t, use "D_value.dw".

    ////////////////////////
    // Set by Execute Stage.
    ////////////////////////

    // Load/store address calculated by AGEN unit.
    reg_t addr;

    // Resolved branch target. (c_next_pc: computed next program counter)
    reg_t c_next_pc;

    // Destination value.
    union64_t C_value;           // If there exists a ** DESTINATION ** register (C),
```

```
                                // this is its value. To reference the value as
                                // uint64_t, use "C_value.dw".

   uint32_t fflags;             // If it is a FP instruction,
                                // this is the new fflags bits it will post

   // If there was an exception during execution, the trap is stored here.
   trap_storage_t trap;

} payload_t;
```

2.2. Pipeline Stages

2.2.1. Frontend pipeline stages: Fetch, Decode, Rename, and Dispatch

Figure 2 illustrates the frontend pipeline stages. Shaded rectangles with rounded corners correspond to functions of the *pipeline_t* class that implement the pipeline stages. For example, as shown in the first shaded rectangle, the Fetch Stage is implemented with the function *pipeline_t::fetch()* in the file fetch.cc.

The frontend pipeline stages are Fetch[1], Decode, Rename and Dispatch. Rename is sub-pipelined into Rename1 and Rename2. Instructions flow through the frontend pipeline stages in "bundles". Pipeline stages are separated by pipeline registers that hold the instruction bundles. The one exception is that the Decode and Rename Stages are separated by the Fetch Queue (FQ), which is a fifo queue that accumulates decoded instructions.

The frontend has two separately-specified widths: the Fetch and Decode Stages are of width *fetch_width* and the Rename and Dispatch Stages are of width *dispatch_width*. It is because of the intervening Fetch Queue that we can have two different widths, if desired.

In a given cycle, the Fetch Stage fetches 0 to *fetch_width* instructions (the fetch bundle). It fetches 0 instructions if it is stalled for an instruction cache miss. If not stalled, it may fetch fewer than the full *fetch_width* due to predicted-taken branches ending the fetch bundle early. If the subsequent Decode Stage is not stalled, the fetch bundle is "clocked" into the pipeline register between the Fetch and Decode Stages, called DECODE[]. DECODE[] is an array of elements that are of type *pipeline_register*. Here is the definition of the class *pipeline_register*, which you can also view in the file pipeline_register.h:

```
class pipeline_register {

public:

   bool valid;                          // valid instruction
   unsigned int index;                  // index into instruction payload buffer
   unsigned long long branch_mask;      // branches that this instruction depends on

   pipeline_register(); // constructor

};
```

[1] Fetch is sub-pipelined into Fetch1 and Fetch2, which is not shown in Figure 2.

As you can see, the class *pipeline_register* corresponds to only a single instruction. To model a pipeline register that contains a bundle of instructions, an array of type *pipeline_register* is used, as in the case of DECODE[].

As explained in Section 2.1, while it is true that the simulator moves instructions through the pipeline from one pipeline stage to the next, it does not move an instruction's entire payload – this would be too slow. Rather, the instruction's index into PAY.buf[] is what is moved. Now refer to the fields of *pipeline_register*, above. The *valid* flag indicates whether or not the pipeline register contains an instruction at all. If it does, then *index* is the instruction's index into PAY.buf[] which can be used to obtain all the information about the instruction that is available up to this point in the pipeline. There is a third and final field, *branch_mask*. This is the bit vector that identifies all prior unresolved branches that the instruction depends on. This is used for selectively squashing instructions in the Issue Queue and Execution Lanes.

In a given cycle, the Decode Stage decodes the decode bundle in the DECODE[] pipeline register, if there is one. While not currently applied to any RISCV instruction opcodes, the 721sim user may split instructions into two instructions, indicated in Figure 2 with a dotted arrow beside each solid arrow. The Decode Stage stalls if there is not enough space in the Fetch Queue for the worst-case number of instructions that may need to be inserted: two times the number of instructions in the decode bundle.

The Fetch Queue, FQ, is a fifo queue of type *fetch_queue* and of size *fq_size*. The size should be at least 2**fetch_width* (otherwise the Decode Stage may stall indefinitely due to splitting *fetch_width* instructions), and may need to be even larger for good performance. For details about the class *fetch_queue*, refer to file fetch_queue.h.

In a given cycle, the Rename1 Stage tries to obtain a full rename bundle – *dispatch_width* instructions – from the Fetch Queue. The Rename1 Stage stalls if either (1) the Fetch Queue has fewer than *dispatch_width* instructions[2], or (2) the Rename2 Stage is stalled. On the other hand, if a full rename bundle can be obtained from the Fetch Queue and the Rename2 Stage is not stalled, a new rename bundle is removed from the Fetch Queue and "clocked" into the pipeline register between the Rename1 and Rename2 Stages, called RENAME2[].

In a given cycle, the Rename2 Stage renames the rename bundle in the RENAME2[] pipeline register, if there is one. The Rename2 Stage may stall for any of three reasons: (1) there is no rename bundle in RENAME2[], or (2) the subsequent Dispatch Stage is stalled, preventing advancement of the rename bundle to the next stage, or (3) the renamer module (your renamer module from Project 2) indicates there aren't sufficient resources for renaming the whole rename bundle (not enough free physical registers for logical destination registers or not enough free checkpoints for branches). On the other hand, if there is a rename bundle in RENAME2[], and if it can advance to the Dispatch Stage, and if there are sufficient renaming resources, then the

[2] The Fetch Unit disables the Fetch1 stage when the Fetch2 stage detects a (very rare) serializing instruction at the end of a fetch bundle. Fetch1 is re-enabled when the serializing instruction retires. Only when Fetch1 is disabled in this manner, Rename1 must relax its requirement of a full rename bundle from the FQ. Thus, there could be a partial rename bundle (and subsequently a partial dispatch bundle) corresponding to the bundle containing a serializing instruction. This scenario is rare and is not depicted in Figure 2.

rename bundle is renamed and "clocked" into the pipeline register between the Rename2 and Dispatch Stages, called DISPATCH[].

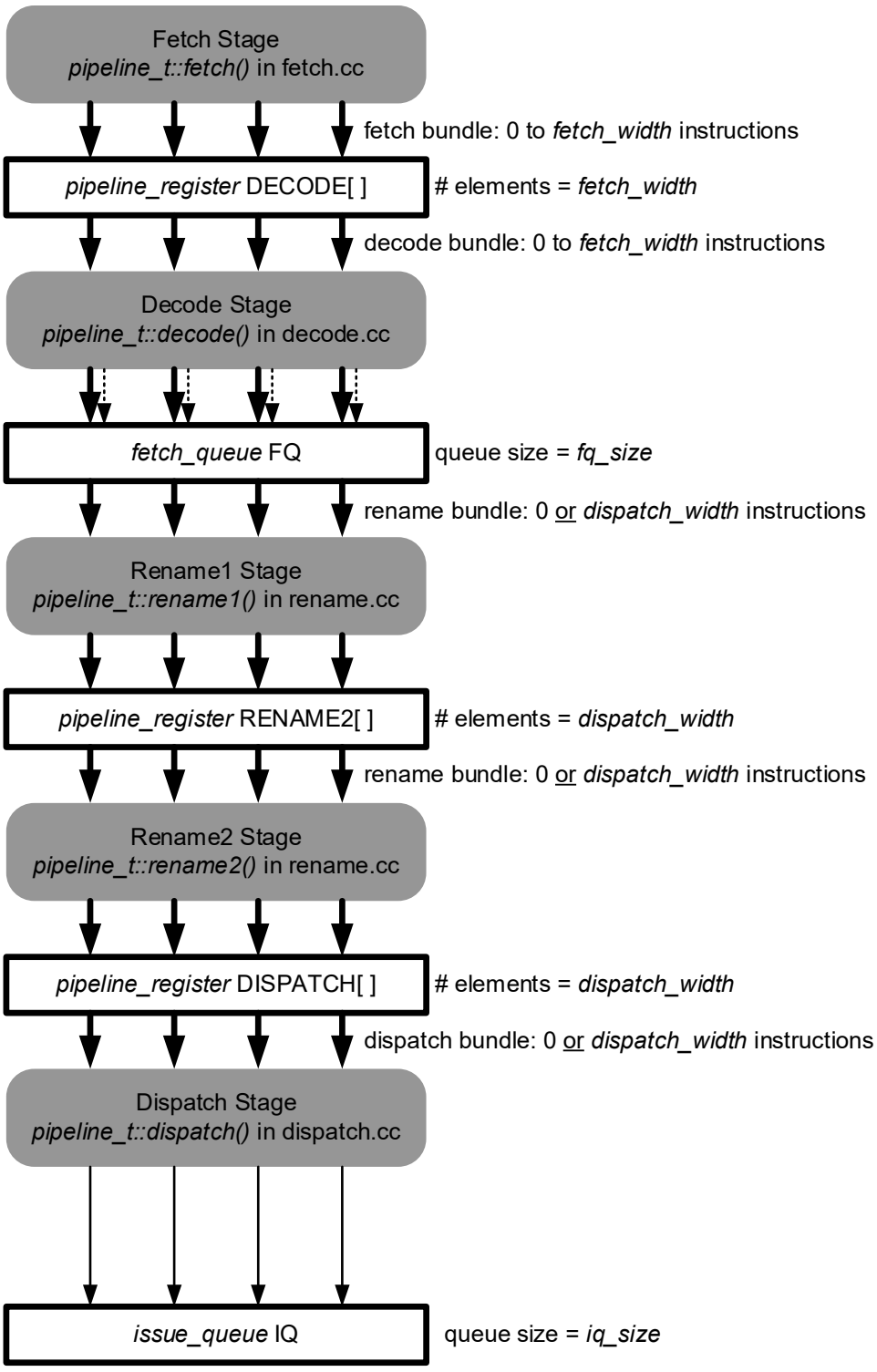

Figure 1. Description of frontend pipeline stages: Fetch, Decode, Rename, and Dispatch.

In a given cycle, the Dispatch Stage dispatches the dispatch bundle in the DISPATCH[] pipeline register, if there is one. The Dispatch Stage may stall if either (1) there is no dispatch bundle in DISPATCH[], or (2) there aren't enough resources to dispatch the whole dispatch bundle. Resources include the Active List (in your renamer module from Project 2), the Issue Queue, and the Load and Store Queues.

2.2.2. Backend pipeline stages: Schedule and Execution Lanes comprised of Register Read, Execute, and Writeback

Figure 3 illustrates the processor backend. It is comprised of the Schedule Stage and multiple Execution Lanes supplied by the Schedule Stage. Each Execution Lane is one instruction wide and has three canonical pipeline stages: Register Read, Execute, and Writeback.

The Issue Queue (IQ) is of type *issue_queue*. The class *issue_queue* is defined in file issue_queue.h. The file issue_queue.h also defines the data structure for a single entry in the Issue Queue CAM, called *issue_queue_entry_t*. Here is the definition of *issue_queue_entry_t*:

```
typedef
struct {

// Valid bit for the issue queue entry as a whole.
// If true, it means an instruction occupies this issue queue entry.
bool valid;

// Index into the instruction payload buffer.
unsigned int index;

// Branches that this instruction depends on.
uint64_t branch_mask;

// Execution lane that this instruction wants.
unsigned int lane_id;

// Valid bit, ready bit, and tag of first operand (A).
bool A_valid;           // valid bit (operand exists)
bool A_ready;           // ready bit (operand is ready)
unsigned int A_tag;     // physical register name

// Valid bit, ready bit, and tag of second operand (B).
bool B_valid;           // valid bit (operand exists)
bool B_ready;           // ready bit (operand is ready)
unsigned int B_tag;     // physical register name

// Valid bit, ready bit, and tag of third operand (D).
bool D_valid;           // valid bit (operand exists)
bool D_ready;           // ready bit (operand is ready)
unsigned int D_tag;     // physical register name


// Support for ideal age-based priority.
int prev;       // IQ index of previous-oldest instruction still in the IQ.
int next;       // IQ index of next-oldest instruction still in the IQ.
} issue_queue_entry_t;
```

The Schedule Stage, implemented by *pipeline_t::schedule()* in file schedule.cc, is fairly concise. It calls the *select_and_issue()* function of the Issue Queue (IQ). This is shown in Figure 3:

IQ.select_and_issue(…). The arguments to *select_and_issue(…)* specify the Execution Lanes that are to receive issued instructions.

A single Execution Lane is of type *lane*. Therefore, multiple Execution Lanes are implemented by an array whose elements are of type *lane*:
lane Execution_Lanes[]
This declaration is also shown at the bottom of Figure 3.

The class *lane* is actually fairly simple: (you can also refer to file lane.h)

```
class lane {
public:
        pipeline_register rr;   // pipeline register of Register Read Stage
        pipeline_register *ex;  // pipeline register(s) of Execute Stage
        pipeline_register wb;   // pipeline register of Writeback Stage

        unsigned int ex_depth;  // number of sub-stages in the Execute Stage

        lane(); // constructor
        void init(unsigned int ex_depth);
};
```

As you can see, the class *lane* consists of three pipeline registers, each just one instruction wide. The first pipeline register, rr, supplies the Register Read Stage. The second pipeline register, ex, supplies the Execute Stage. The third pipeline register, wb, supplies the Writeback Stage. As an example, to reference the rr pipeline register of the i^{th} Execution Lane, one would use:
Execution_Lanes[i].rr
This usage is also illustrated in Figure 3, next to the rr, ex, and wb pipeline registers.

The Register Read, Execute, and Writeback Stages are implemented by the functions *pipeline_t::register_read(…)* in file register_read.cc, *pipeline_t::execute(…)* in file execute.cc, and *pipeline_t::writeback(…)* in file writeback.cc. The only argument to these functions is a lane number that specifies which Execution Lane is to be processed by the function.

An instruction leaves the pipeline after the Writeback Stage, although it still occupies entries in the Active List (all instructions), the Load and Store Queues (only loads and stores, respectively), and the Fetch Unit's branch queue (branches only), until the instruction retires.

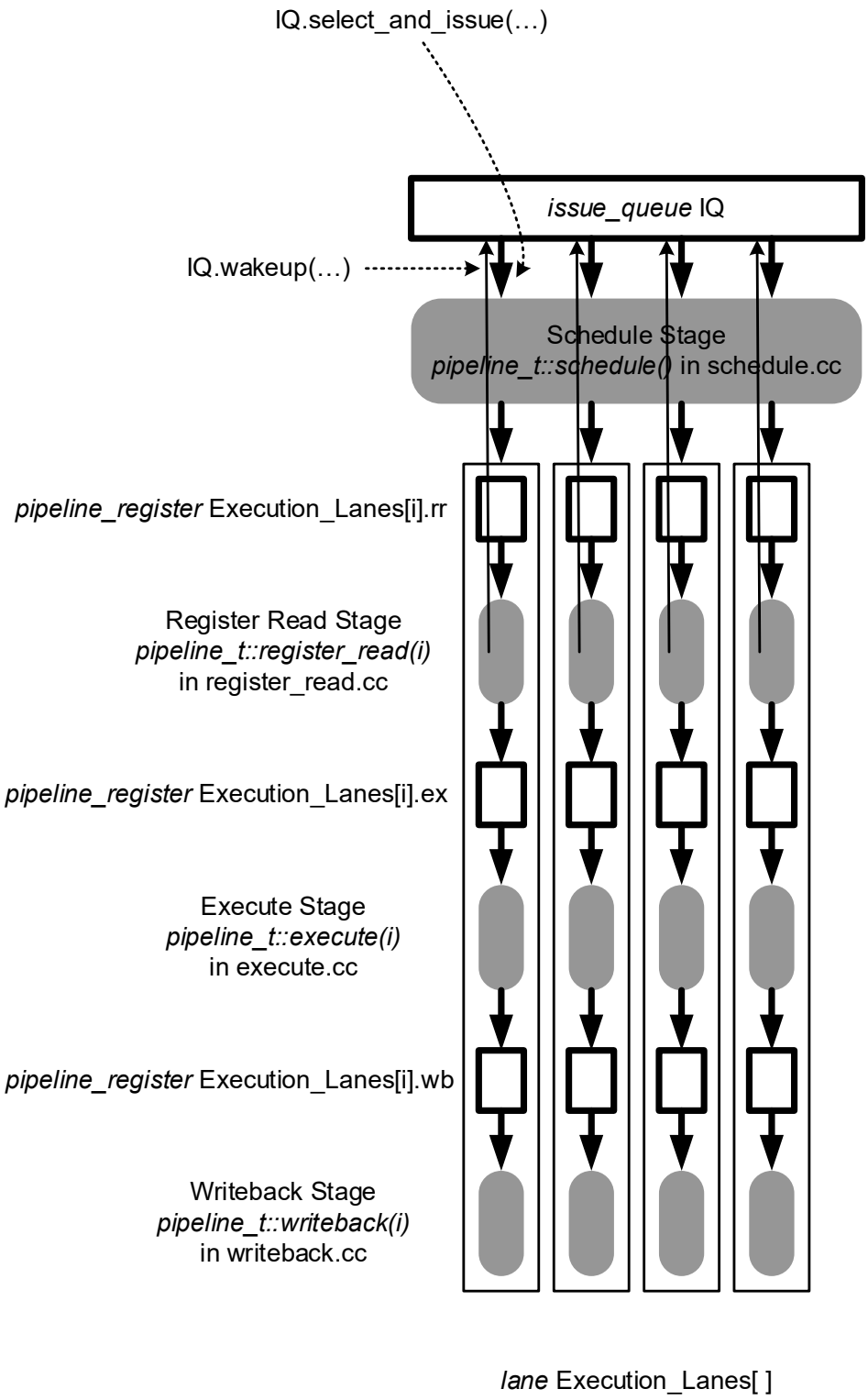

Figure 2. Description of the processor backend: The Schedule Stage and multiple Execution Lanes supplied by the Schedule Stage. Each execution lane is one instruction wide and has three canonical pipeline stages: Register Read, Execute, and Writeback.

2.3. Register File and Memory

The *pipeline_t* class declares two objects to implement the register file and memory:

- *renamer* *REN: This is a pointer to an instance of your *renamer* class. This instance is for management of the Physical Register File (a unified physical register file for 64-bit integer and floating-point values). Since REN is a pointer, make calls to functions of the *renamer* class using `REN->xxx()` where xxx is the function name.
- *lsu* LSU: This is the Load and Store Unit. It is comprised of the Load Queue, Store Queue, and cache hierarchy (the cache hierarchy models only timestamps; the mmu models architectural memory state). The LSU also maintains the architectural memory state of the running program (via an instance of the mmu class from the riscv-base library).

3. Tasks for Project 3

3.1. Getting started

721sim is built using cmake, which allows for building the simulator executable outside of the source tree. It is recommended you use distinct and parallel directories for your source code, builds, and runs, such as the following:

`~/ece721/project3/721sim` → This is where you will clone the 721sim source code repo.

`~/ece721/project3/run` → This is where you can create benchmark subdirectories, use atool-simenv to set up these subdirectories, and launch runs.

`~/ece721/project3/build` → This is where you will build the 721sim executable using cmake. Note that you can have multiple build directories (`build-release`, `build-debug`, *etc*.) which is convenient if you want multiple 721sim executables (*e.g.*, with and without -g, or different 721sim executables for different git branches, *etc*.).

This is just an example structure. The key point is that it is good practice to keep your source repo, builds, and benchmark runs separate.

3.1.1. Make a top-level Project 3 directory

Make a top-level directory for your Project 3, such as `~/ece721/project3`. Sections that follow will refer to this example Project 3 directory. If you chose a different path name and directory name, substitute accordingly.

3.1.2. Obtain the (partial) 721sim source code

1. `cd ~/ece721/project3`
2. `git clone /mnt/designkits/riscv-toolchains/ece721/721sim`
3. The clone command should have created a subdirectory `721sim/`. Thus, your new repo will be under: `~/ece721/project3/721sim/`. The repo has two subdirectories, `riscv-base/` and `uarchsim/`. You will implement your tasks under the `uarchsim/` subdirectory. This is where you will put your Project 2 renamer source code. This is where the source files that you need to modify reside. These tasks are specified in Section 3.2.

3.1.3. How to build the 721sim executable

Note: You should be able to test building the 721sim executable, even before coding Project 3, as long as you place your Project 2 renamer files (`renamer.h`, `renamer.cc`, and any supporting files you created for the renamer) in `uarchsim/`. But 721sim will not work until you code and debug the missing parts.

1. `cd ~/ece721/project3`
2. `mkdir build`
3. `cd build`
4. `cmake -DCMAKE_BUILD_TYPE=RelO3 ../721sim`
   Note: The last argument should be the path to your repo. It can be a relative path, as shown, or an absolute path.
   Note: cmake creates a build directory and does not itself compile 721sim. Actual compilation happens in the next step.
5. `make -j$(nproc)`
   Note: The `-j` flag causes make to run with maximum parallelism.
6. If compilation succeeds, the 721sim executable should be located here:
   `./uarchsim/721sim`
   i.e., at the absolute path:
   `~/ece721/project3/build/uarchsim/721sim`
   You can do a quick test (just to print out 721sim's "help"):
   `./uarchsim/721sim -h`
7. You can either copy the 721sim executable to your run directory or create a symbolic link to it. I recommend a symbolic link to save space and to always know which build directory you are referencing.

3.2. Your tasks

1. Copy your Project 2 renamer files (`renamer.h`, `renamer.cc`, and any supporting files you created for the renamer) to `~/ece721/project3/721sim/uarchsim/`. See Section 1 for directions if your Project 2 renamer doesn't work.
2. I intentionally omitted segments of code in seven files under `~/ece721/project3/721sim/uarchsim/`: rename.cc (Rename Stage), dispatch.cc (Dispatch Stage), register_read.cc (Reg. Read Stage), execute.cc (Execute Stage), writeback.cc (Writeback Stage), retire.cc (Retire Stage), and squash.cc (complete squash function, branch resolve function). Search for "FIX_ME #" and you will see where the omissions are. There are a total of 18 omissions among the seven files. They are numbered (*e.g.*, FIX_ME #10) and several FIX_MEs have multiple parts (*e.g.*, FIX_ME #10a, #10b1, #10b2, …). Detailed comments indicate what to implement at each FIX_ME location. Often, the comments are much longer than the actual code that you will write, so don't be alarmed by the volume of comments.

3.3. Debugging

You should use gdb (or other debugger) to step through your code.

Asserts will go off in the simulator if your code segments or renamer module has bugs. There are both 1) miscellaneous asserts throughout the simulator and 2) explicit comparisons between the

functional simulator and processor simulator, in *pipeline_t::checker()* (checker.cc) which is called from *pipeline_t::retire()* (retire.cc).

You may have bugs in your code segments that do not cause the simulator to assert/die, but cause the measured IPC to be way off. I will post results from my simulator so you can compare performance.

In any case, after reading this document, browsing through the source files, and gaining enough knowledge to modify the simulator, you should be able to track down most problems. If you implement the missing code segments properly, you should not have to change any of the existing code. If you feel the need to change existing code, please contact me first.

3.4. Submitting and self-grading your project

You will submit just your renamer (`renamer.h`, `renamer.cc`, and optionally any other supporting `.h/.cc` files that you may have created for the renamer) and the seven modified source files (`rename.cc`, `dispatch.cc`, `register_read.cc`, `execute.cc`, `writeback.cc`, `retire.cc`, and `squash.cc`) to Gradescope for automatic grading. Just as you had done during development, Gradescope will build the simulator, test it on the validation runs, and compute your score. Your final score on this project is whatever score Gradescope shows for your latest submission before the deadline.

3.5. Project scoring

BASE:
If you submit a legitimate attempt at an implementation (we will inspect the code), ***BASE*** = 30; otherwise ***BASE*** = 0.

N: total number of validation runs.

validation_run_score$_i$:
Score for each validation run:

- 10: Simulator runs for > 1,000 instructions but < 10,000 instructions (faults, asserts, or deadlocks before 10,000 instructions).
- 20: Simulator runs for > 10,000 instructions but < 100,000 instructions (faults, asserts, or deadlocks before 100,000 instructions).
- 30: Simulator runs for > 100,000 instructions but < 1,000,000 instructions (faults, asserts, or deadlocks before 1,000,000 instructions).
- 40: Simulator runs for > 1,000,000 instructions but does not complete (faults, asserts, or deadlocks before completion).
- 60: Simulator runs to completion, but IPC differs from instructor's version by > 1%.
- 70: Simulator runs to completion, and IPC matches within 1%.

The project score is calculated as follows:

$$project\ score = BASE + \frac{\sum_{i=1}^{N} validation\_run\_score_i}{N}$$

3.6. Late policy

-1 point for each day (24-hour period) late, according to the Gradescope timestamp. The late penalty is pro-rated on an hourly basis: -1/24 point for each hour late. We will use the "ceiling" function of the lateness time to get to the next higher hour, *e.g.*, ceiling(10 min. late) = 1 hour late, ceiling(1 hr, 10 min. late) = 2 hours late, and so forth.

Gradescope will accept late submissions no more than one week after the deadline. The goal of this policy is to encourage forward progress for other work in the class.