Project Part 3:
Working with the Full Simulator

Version 1.0
Due Date: Friday, October 26, 11:59 PM

ECE 721: Advanced Microarchitecture
Fall 2012, Prof. Rotenberg

- READ THIS ENTIRE DOCUMENT.
- Late policy: 1 point deduction for each hour the project is late (that's 24 points/day).
- Even if you do not finish this part by the due date, you must build on it for later parts.
- Sharing of code between students is considered cheating and will receive appropriate action in accordance with University policy (see the section titled "Academic integrity" in the course syllabus). The TAs will scan source code (from current and past semesters) through various tools available to us for detecting cheating. Source code that is flagged by these tools will be dealt with severely.
- A Wolfware message board will be provided for posting questions, discussing and debating issues, and making clarifications. It is an essential aspect of the project and communal learning. Students must not abuse the message board, whether inadvertently or intentionally. Above all, never post actual code on the message board (unless permitted by the TAs/instructor). When in doubt about posting something of a potentially sensitive nature, email the TAs and instructor to clear the doubt.

1. Introduction

In this part of the 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 or other experimental framework to carry out your independent research project.

You have three tasks (also see Section 4).

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 Part 2 into the simulator (renamer.h, renamer.cc). If you did not complete that part of the project, you must come see me about fixing the problems with your renamer. If it is beyond repair, you can use object code of a working renamer.

If you do a thorough job on this part of the project, the next part of the project (research component) will be much more rewarding. You will be able to focus more time on ideas and experiments, and less time on debugging.

2. Overview of simulator framework

Recall from the first part of the project that the simulator is actually composed of two simulators. The *functional simulator* models only the functionality of the program. The *processor simulator* models both functionality and timing. Pairing a functional simulator with the processor simulator enables validation, debugging, and oracle prediction modes. The processor simulator is checked by comparing the results of committed instructions to their counterparts in the functional simulator (the functional simulator is much simpler and has been verified). Futhermore, since the functional simulator is kept a certain number of instructions ahead of the processor simulator, the functional simulator can supply perfect branch and value predictions to the processor simulator for various oracle prediction studies.

The *Thread* class (LibSS_smt/include/Thread.h) defines the functional simulator and the *processor* class (processor.h) defines the processor simulator. For a single core, we instantiate only one Thread-processor pair. For multiple cores, multiple Thread-processor pairs are instantiated to model a multi-core architecture. Therefore, in the general case, we need an array of *Thread* objects and an array of *processor* objects, as indicated below.

```
Thread *THREAD[];
processor *PROC[];
```

In the case of only a single Thread-processor pair, *THREAD[0] is the functional simulator and *PROC[0] is the processor simulator. (THREAD[0] is a pointer to the functional simulator and PROC[0] is a pointer to the processor simulator.) Note that THREAD and PROC are global variables, so the functional simulator and processor simulator can be accessed from any scope.

3. class *processor* (processor.h)

The *processor* class (processor.h) models the entire pipeline. The new 721sim (developed in the Fall 2011 semester) 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.

3.1. 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 *processor* 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 *processor* 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 Fetch Stage.
   ////////////////////////

   SS_INST_TYPE inst;           // The simplescalar instruction.
   unsigned int pc;             // The instruction's PC.
   unsigned int next_pc;        // The next instruction's PC. (I.e., the PC of
                                // the instruction fetched after this one.)
   unsigned int pred_tag;       // If the instruction is a branch, this is its
                                // index into the Branch Predictor's
                                // branch queue.

   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 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.

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

   unsigned int flags;          // Operation flags: can be used for quickly
                                // deciphering the type of instruction
                                // (we used this in Project Part 1).
   enum ss_fu_class fu;         // Operation functional unit class (ignore:
                                // not currently used).
   unsigned int latency;        // Operation latency (ignore: not currently
                                // used).
   bool split;                  // Instruction is split into two.
   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 checkpoint;             // If 'true', this instruction is a branch
                                // that needs a checkpoint.
   bool split_store;            // Floating-point stores (S_S and S_D) are
                                // implemented as split-stores because they
                                // use both int and fp regs.

   // Source register A.
   bool A_valid;                // If 'true', the instruction has a
                                // first source register.
   bool A_int;                  // If 'true', the source register is an
                                // integer register, else it is a
```

```
                              // floating-point 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.
    bool B_int;               // If 'true', the source register is an
                              // integer register, else it is a
                              // floating-point 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.
    bool C_int;               // If 'true', the destination register is an
                              // integer register, else it is a
                              // floating-point register.
    unsigned int C_log_reg;   // The logical register specifier of the
                              // destination register.

    // IQ selection.
    sel_iq iq;                // The value of this enumerated type indicates
                              // whether to place the instruction in the
                              // integer issue queue, floating-point issue
                              // queue, or neither issue queue.
                              // (The 'sel_iq' enumerated type is also
                              // defined in this file.)

    // 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;                // LWL or SWL instruction.
    bool right;               // LWR or SWR instruction.

    ////////////////////////
    // 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.

    // 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_int;  // Index into integer Active List.
    unsigned int AL_index_fp;   // Index into floating-point 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;         // Execution lane chosen for the instruction
                                // (to be implemented later).

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

   // Source values.
   DOUBLE_WORD A_value;         // If there exists a first source register (A),
                                // this is its value. To reference the value as
                                // an unsigned long long, use "A_value.dw".
   DOUBLE_WORD B_value;         // If there exists a second source register (B),
                                // this is its value. To reference the value as
                                // an unsigned long long, use "B_value.dw".

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

   // Load/store address calculated by AGEN unit.
   unsigned int addr;

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

   // Destination value.
   DOUBLE_WORD C_value;         // If there exists a destination register (C),
                                // this is its value. To reference the value as
                                // an unsigned long long, use "C_value.dw".

} payload_t;
```

3.2. Pipeline Stages

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

Figure 1 illustrates the frontend pipeline stages. Shaded rectangles with rounded corners correspond to functions of the *processor* class that implement the pipeline stages. For example, as shown in the first shaded rectangle, the Fetch Stage is implemented with the function *processor::fetch()* in the file fetch.cc.

The frontend pipeline stages are Fetch, 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.

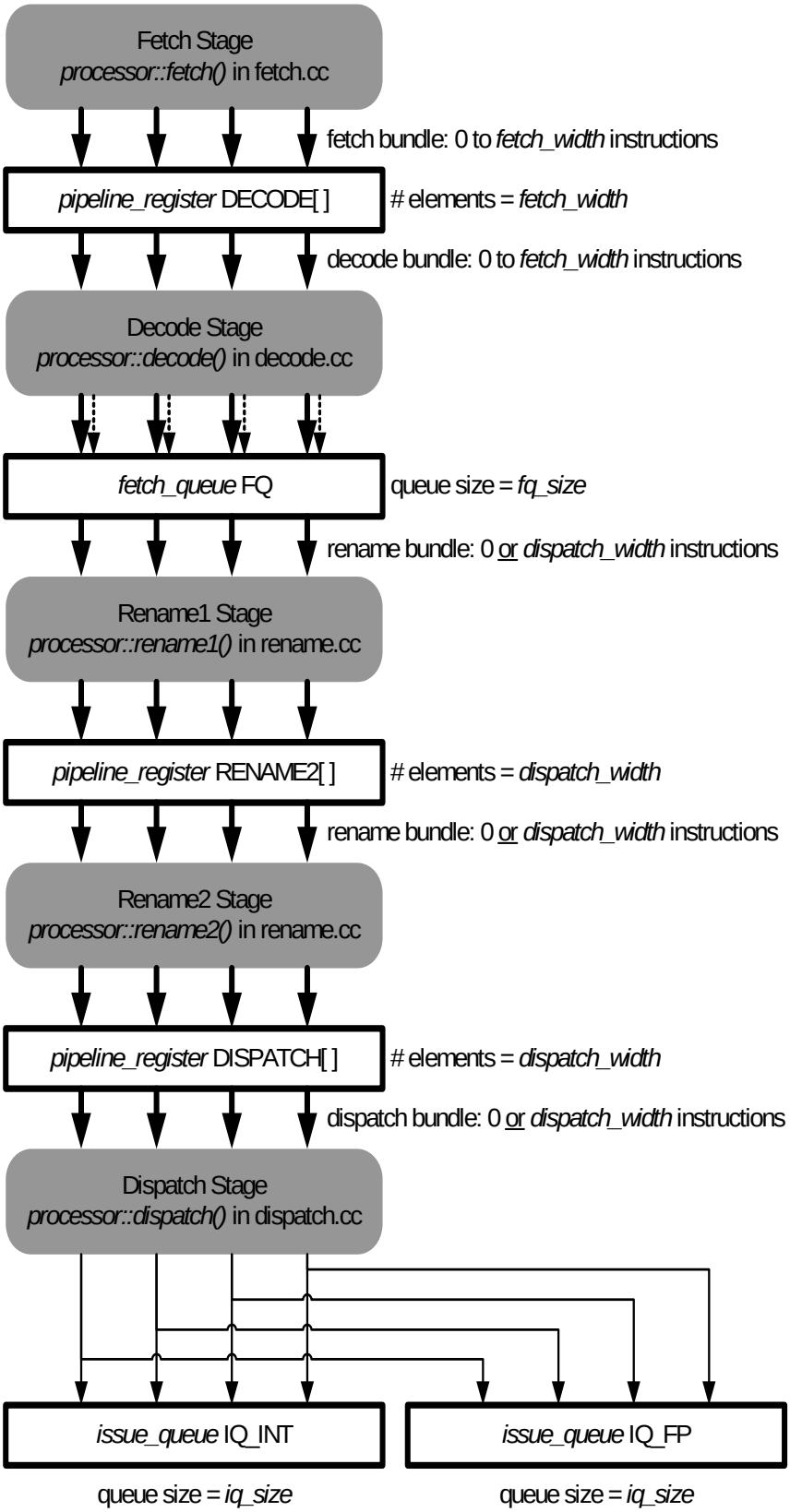

Figure 1. Description of frontend pipeline stages: Fetch, Decode, Rename, and Dispatch.

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

};
```

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 3.1, 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 squashing instructions throughout the pipeline.

In a given cycle, the Decode Stage decodes the decode bundle in the DECODE[] pipeline register, if there is one. While it is not common, some instructions get split into two instructions, indicated in Figure 1 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, 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 Part 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 rename bundle is renamed and "clocked" into the pipeline register between the Rename2 and Dispatch Stages, called 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 integer and floating-point Active Lists (in your renamer module from Project Part 2), the integer and floating-point Issue Queues, and the Load and Store Queues.

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

Figure 2 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.

Both the integer Issue Queue (IQ_INT) and the floating-point Issue Queue (IQ_FP) are 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.
unsigned long long branch_mask;

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

// 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

} issue_queue_entry_t;
```

The Schedule Stage, implemented by *processor::schedule()* in file schedule.cc, is fairly concise. It calls the *select_and_issue()* function of both the integer Issue Queue (IQ_INT) and floating-point Issue Queue (IQ_FP). This is shown in Figure 2: *IQ_INT.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 2.

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 of Execute Stage
                pipeline_register wb;   // pipeline register of Writeback Stage

                lane(); // constructor
};
```

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 2, next to the rr, ex, and wb pipeline registers.

The Register Read, Execute, and Writeback Stages are implemented by the functions *processor::register_read(…)* in file register_read.cc, *processor::execute(…)* in file execute.cc, and *processor::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 integer and floating-point Active Lists (all instructions), the Load and Store Queues (only loads and stores, respectively), and the branch predictor's prediction queue (branches only), until the instruction retires.

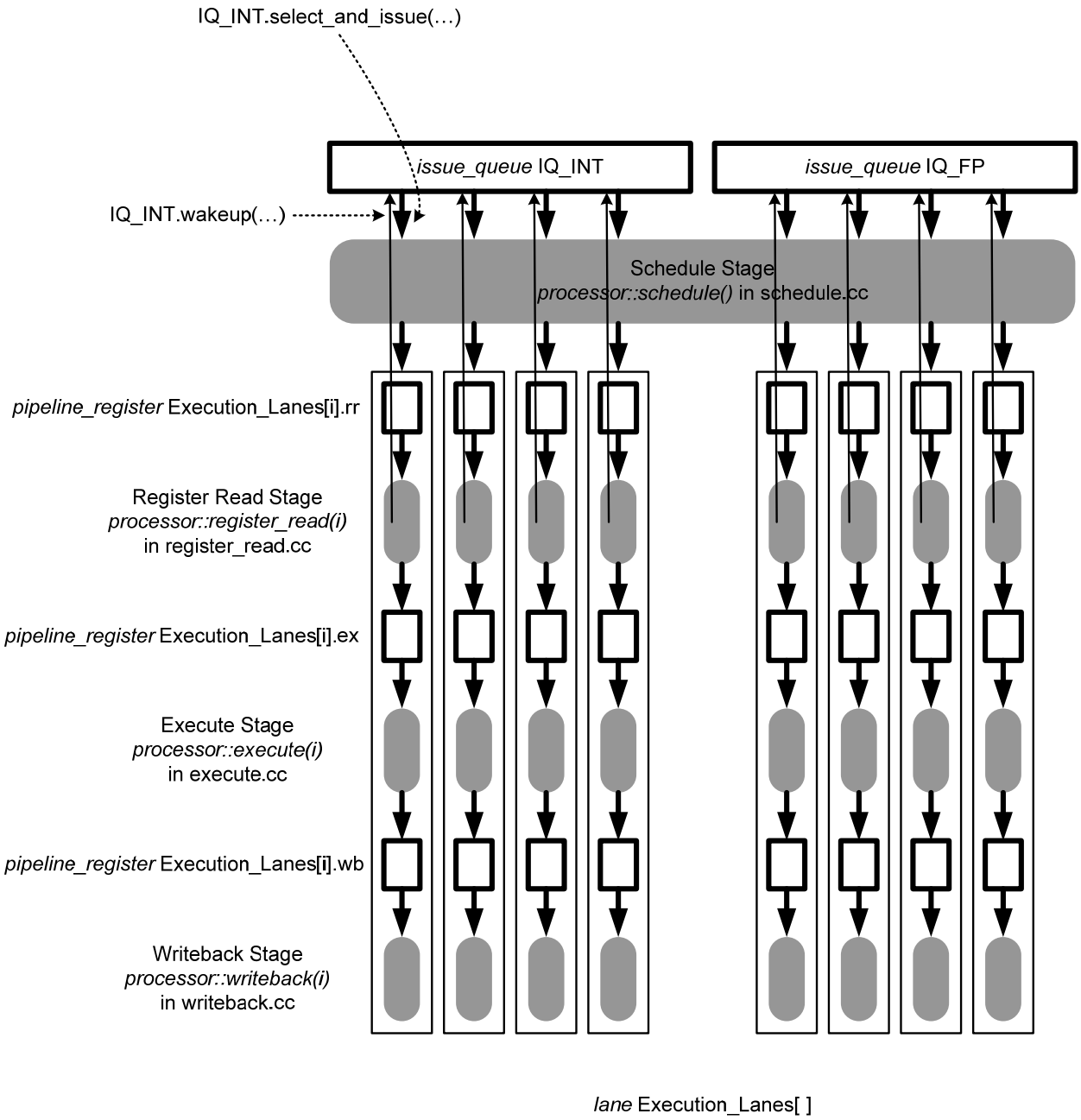

Figure 2. Description of the processor backend: The Schedule Stage and multiple Execution Lanes supplied by the Schedule Stage. Each execution lane is one instruction wide and has three canonical pipeline stages: Register Read, Execute, and Writeback.

3.3. Register Files and Memory

The *processor* class declares three objects to implement the register files and memory:

- *renamer* *REN_INT: This is a pointer to an instance of your *renamer* class. This instance is for management of the integer Physical Register File. Since REN_INT is a pointer, make calls to functions of the *renamer* class using REN_INT->xxx() where xxx is the function name.

- *renamer* *REN_FP: This is a pointer to an instance of your *renamer* class. This instance is for management of the floating-point Physical Register File. Since REN_FP is a pointer, make calls to functions of the *renamer* class using REN_FP->xxx() where xxx is the function name.
- *mem_interface* LSU: This is the Load and Store Unit. It is comprised of the Load Queue, Store Queue and L1 Data Cache. The LSU also maintains the entire architectural memory state of the running program in a memory table.

I plan to improve the LSU in a future simulator release as follows:

(1) Currently, there is no speculative memory disambiguation mode. In the default configuration, loads stall if there are prior unknown store addresses. Fortunately, there is an oracle configuration (-o simulator flag) that is a good substitute for modeling speculative memory disambiguation. In the oracle configuration, stores place their addresses into the Store Queue in the Dispatch Stage instead of waiting until execution; the early store addresses are obtained from the functional simulator.

(2) Once there is a speculative memory disambiguation mode in place, it will be necessary to implement a memory dependence predictor to guide the decision to speculate versus stall. Also, various load misprediction recovery models should be supported.

4. Tasks for Project Part 3

In this project, you will 1) read this document and read through all source files carefully to learn about the simulator, 2) write segments of code that I removed from the simulator, and 3) insert your renamer module into the simulator (renamer.h, renamer.cc).

4.1. Getting started

The materials (source code + this document) can be obtained from the project website.

1. Download the zip file **src3rel.zip** to your project directory that contains src1/ and src2/.
2. unzip src3rel.zip
3. After step 2, a directory called src3_release/ is created and this new directory should be *parallel* with src1/ and src2/. You may rename the new directory to src3/ if you wish, just to have consistent naming.
4. Copy your renamer module (two files, renamer.h and renamer.cc) from your previous project to the src3_release/ directory. If your renamer module does not work, see Section 1 for directions.
5. It doesn't hurt to recompile libSS_smt_*.a at this time. In LibSS_smt/lib.src, do this:
   make clean; make
6. In src3_release/BPRED, do this:
   make clean; make
7. In src3_release/MEM, do this:
   make clean; make
8. In src3_release/UTILS, do this:
   make clean; make
9. In src3_release, do this:
   make clean; make

(The simulator will compile even with the code segments removed, as long as you remember to copy over your renamer files.)

4.2. YOUR TASKS

1. If your renamer from Project Part 2 works, then you should not have to do anything for the renamer class. Otherwise, see Section 1 for directions.
2. I have intentionally omitted segments of code in six files: rename.cc (Rename Stage), dispatch.cc (Dispatch Stage), register_read.cc (Reg. Read Stage), execute.cc (Execute Stage), writeback.cc (Writeback Stage), and retire.cc (Retire Stage). Search for "FIX_ME" and you will see where the omissions are. There are a total of 18 omissions among the six files. They are numbered (*e.g.*, FIX_ME #10) and several FIX_MEs have multiple parts (*e.g.*, FIX_ME #10a, #10b, …). 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.

4.3. Debugging

You should use gdb (or other debugger) to step through your code. Another useful feature embedded within processor.h and processor.cc is the BREAKPOINT variable. The user is prompted for a BREAKPOINT cycle when simulation begins. The function which advances to the next cycle checks to see if "cycle" has reached BREAKPOINT. When the BREAKPOINT cycle is reached, the simulator asks you to enter a new BREAKPOINT cycle. In gdb, before entering the new BREAKPOINT cycle, you can ^C the simulator to browse simulator state.

IMPORTANT: The BREAKPOINT functionality is currently commented out. To enable, you must comment out the surrounding #if 0 / #endif macros. You must do this everywhere BREAKPOINT is used in processor.h and processor.cc.

Asserts will go off in the simulator if your code segments have bugs. There are both 1) miscellaneous asserts throughout the simulator and 2) explicit comparisons between the functional simulator and processor simulator, in *processor::checker()* (checker.cc) which is called from *processor::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.

4.4. What to hand in

Hand in your modified files (rename.cc, dispatch.cc, register_read.cc, execute.cc, writeback.cc, retire.cc) for testing. Hand-in directions will be posted on the course website.

4.5. Grading

- 0% – You don't hand in anything.
- 25% – You submit a legitimate attempt at an implementation (we will inspect the code), but simulator does not compile or run.
- 50% – Simulator runs for > 1,000 instructions but < 10,000 instructions.
- 60% – Simulator runs for > 10,000 instructions but < 100,000.
- 70% – Simulator runs for > 100,000 instructions, but does not complete.
- 80% – Simulator completes but performance differs from instructor's version by > 10%.
- 100% – Simulator completes and performance matches within 10%.