Project 4:
CPR implementation and study

Version 1.0

ECE 721: Advanced Microarchitecture
Spring 2023, Prof. Rotenberg

Due: Friday, April 21, 2023, 11:59pm
Late projects will only be accepted until: Friday, April 28, 2023, 11:59pm

- READ THIS ENTIRE DOCUMENT.
- Academic integrity:
  - **Source code:** Each team 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 team 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 team has any doubts or questions, or if a team is stumped by a bug, the team is encouraged to seek assistance using both of the following channels.
  - Teams may receive assistance from the TA(s) and instructor.
  - Teams 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 team. 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 teams 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 teams 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. Overview and Teams

In this project, you will implement Checkpoint Processing and Recovery (CPR) in 721sim in several phases. Students will work in teams of two students. Exceptions to two-student teams will require approval of the instructor, a strong justification, and an appropriate plan for the project scope.

2. Key aspects to be modeled

There are four key aspects for modeling CPR:

1. Coarse-grain retirement and aggressive register reclamation
2. Checkpoint-based recovery including complications of rolling back prior to the offending instruction (guaranteeing forward progress)
3. Confidence-based checkpoint placement
4. Hierarchical store queue

Rather than tackle all of these at once, it is judicious to implement them in phases and in the order above. Section 3 discusses judicious implementation phases.

3. Implementation phases

3.1. Phase 1: coarse-grain retirement and aggressive register reclamation

In this phase, you will focus on getting coarse-grain retirement and aggressive register reclamation working. Ideally, we would like to decouple this phase from complications of checkpoint-based recovery: the complication of ensuring forward progress despite rolling back to an instruction that is much older than the offending instruction that caused the rollback. Section 3.1.1 discusses how we can minimize and simplify recoveries using oracle information for *exact checkpoint placement*. This sets the stage for focusing on the task at hand: Section 3.1.2 discusses implementing coarse-grain retirement, aggressive register reclamation, and recovery based on exact checkpoint placement.

3.1.1. Avoiding complications of checkpoint-based recovery: exact checkpoint placement

We can avoid complications of checkpoint-based recovery in two ways: (1) eliminate or minimize offending instructions and (2) ideally place checkpoints at offending instructions. Thus, there will be two subphases, Phase 1A and Phase 1B:

- **Phase 1A: Minimize offending instructions and ideally place checkpoints at remaining offending instructions:** Run with perfect branch prediction (eliminate branch mispredictions

as offending instructions) and oracle memory dependence prediction (eliminate load violations as offending instructions). There may still be dynamic exceptions (*e.g.*, instruction fetch page fault, load page fault, store page fault, *etc.*) and serializing instructions (system calls, atomic memory operations, CSR instructions, *etc.*). For dynamic exceptions, we will exploit ideal information from the leading functional-only simulator, leveraging linkages between fetched instructions and counterparts in the Debug Buffer (these linkages are already created in the Fetch Unit which calls payload::map_to_actual()). Given these linkages, we can ideally know which instructions are going to cause dynamic exceptions (because the leading functional-only simulator should have encountered the same exceptions) and insert a checkpoint right before the offending instruction. Serializing instructions are known by their opcode *a priori* and we will simply insert checkpoints before and after a serializing instruction. Summarizing, in Phase 1A: (1) There are no branch mispredictions or load violations, (2) place a checkpoint before an instruction that we know, using ideal knowledge from its counterpart in the Debug Buffer, will post an exception, and (3) place checkpoints before and after a serializing instruction (which is true even for a real implementation in subsequent phases).
- **Phase 1B: Ideally place checkpoints at mispredicted branches ("oracle confidence") and other offending instructions:** Run with real branch prediction but still rely on oracle memory dependence prediction. To ensure uncomplicated rollback to just after a mispredicted branch, we can ideally place a checkpoint right after a mispredicted branch using "oracle confidence": we can identify a mispredicted branch *a priori*, by comparing its predicted next PC against the next PC of its counterpart in the Debug Buffer (this can only be done for branches that are themselves on the correct path, because they have valid linkages to the Debug Buffer, which is fine because we only need to ensure placing a checkpoint right after the oldest mispredicted branch in the window… branches and other instructions after it are not consequential). We must still assume oracle memory dependence prediction because it may not be possible to identify load violations *a priori* in the frontend (load violations depend on dynamic timing among stores and loads in the window). Summarizing, in Phase 1B: (1) There are no load violations, (2) place a checkpoint before an instruction that we know, using ideal knowledge from its counterpart in the Debug Buffer, will post an exception, (3) place checkpoints before and after a serializing instruction (which is true even for a real implementation in subsequent phases), and (4) place a checkpoint after a mispredicted branch that is itself on the correct path (it is the first mispredicted branch in the window), using oracle confidence (knowledge that its predicted next PC differs from the next PC of its counterpart in the Debug Buffer).

In Phases 1A and 1B, we easily ensure forward progress because checkpoints are placed exactly where they need to be for rollbacks.

How to obtain information about an instruction *a priori* from the Debug Buffer (assume "index" is the index of the instruction in the PAY buffer):

```
// The Debug Buffer only has counterparts for instructions on the correct
// control-flow path.  Thus, db_index is only valid if good_instruction is true.
db_t * actual;  // "actual" is a pointer to the Debug Buffer entry of
                // the instruction, assuming it has a linkage.
if (PAY.buf[index].good_instruction) {
   actual = get_pipe()->peek(PAY.buf[index].db_index);
   // You can now look at anything about the instruction a priori
   // (in an oracle manner). See debug.h, struct db_t, to learn which information
   // you can discover about the instruction.  For example, actual->a_next_pc is
   // the PC of the next dynamic instruction after this instruction.  For a
   // branch instruction, you know which is the correct control-flow path after
   // the branch by looking at actual->a_next_pc.  Thus, you can determine
   // whether or not a branch is mispredicted by comparing PAY.buf[index].next_pc
   // (the PC of the instruction that was speculatively fetched after the
   // the branch) against actual->a_next_pc (the PC of the instruction that
   // should have been fetched after the branch).
   // As another example, you can know a priori if an instruction if going to
   // post an exception, by examining actual->a_exception.  If the counterpart in
   // the functional-only simulator raised an exception, then so should this
   // instruction in the pipeline simulator.  (Note that instruction fetch and
   // decode exceptions are known early in the pipeline anyway; in this case you
   // can confirm these same exceptions occurred in the functional-only
   // simulator.)
}
```

3.1.2. Implementing coarse-grain retirement, aggressive register reclamation, and recovery based on exact checkpoint placement

Here is a brainstorm of changes. These are comprehensive but may not be exhaustive. You will figure out details during implementation of Phase 1.

- **Checkpoint buffer:** Replace the Active List and AMT with a checkpoint buffer. In your Project 2, you already implemented a buffer of checkpoints, but you allocated checkpoints to branches informed by the GBM and implemented OOO reclamation of checkpoints based on branch resolution. In the baseline CPR implementation, manage the checkpoint buffer as a FIFO of checkpoints with head and tail pointers, and only the head (oldest) checkpoint can be reclaimed and this is signaled by the coarse-grain retirement logic. Note that there is always at least one checkpoint allocated (the head checkpoint), even for an empty pipeline, that corresponds to the committed register state (analogous to the AMT).
- **Maximum number of instructions between checkpoints:** As discussed in Section 3.1.1, in Phase 1, we are implementing exact checkpoint placement at criteria instructions to simplify recovery. Thus, you may think it unnecessary to create a new checkpoint after a maximum threshold of dynamic instructions since the last checkpoint (*e.g.*, 256 instructions in the CPR paper). Yet, practically speaking, we are constrained by the PAY buffer size (a simulator-only construct). In the baseline simulator, the PAY buffer is NOT a structural hazard (as it shouldn't be, because it is a simulator-only construct). It is auto-sized based on the Active List size ("rob_size") and maximum number of instructions in the frontend stages. In pipeline.cc, the pipeline::pipeline() constructor, you will see an upper bound for the number of in-flight instructions passed into the PAY() constructor:
  ```
  PAY(2*fetch_width + fq_size /* FETCH2, DECODE, FQ */ + 2*dispatch_width + rob_size
  /* RENAME2, DISPATCH, ROB */)
  ```
  We should keep this formulation, both from a practical simulator standpoint (bounded PAY buffer) and from an academic standpoint (CPR's large, yet still bounded, *virtual window size*). Conceptually, CPR decouples its *virtual window size* (for example, 8 checkpoints * maximum of 256 instructions between checkpoints, or 2,048 instructions) from its *physical window size* (PRF size). Contrast this with a ROB-based processor: its virtual window size (Active List size) and physical window size (PRF size) are tightly coupled (every instruction in the Active List that needs a physical register holds onto that physical register). Even though in CPR,

there is no longer an Active List, we can take the meaning of "rob_size" to mean the maximum virtual window size of CPR. What this means for you: (1) retain the rob_size parameter to signify CPR's virtual window size, (2) continue to use the rob_size parameter for auto-sizing the PAY buffer (we also use it for auto-sizing the Fetch Unit's branch queue, for convenience), and (3) calculate the maximum threshold of dynamic instructions between checkpoints as: max_instr_bw_checkpoints = (rob_size / num_chkpts), and abide by this threshold. Thus, we don't need a new simulator command-line flag for max_instr_bw_checkpoints, rather, we derive it from existing parameters and command-line flags that set these parameters (--al, --cp).

- **Creating checkpoints:** In the rename2() stage, create checkpoints according to guidance about criteria instructions discussed in Section 3.1.1 for Phase 1A and Phase 1B. Also create a checkpoint if the number of dynamic instructions renamed since the last checkpoint is max_instr_bw_checkpoints, as explained above. Accordingly, here are some key changes to rename2() and the renamer class:
    - Add a member variable to the pipeline_t class for keeping track of the number of instructions renamed since the last checkpoint, for example:
      `uint64_t instr_renamed_since_last_checkpoint`
      Manage it properly: initialize it to zero in the pipeline_t constructor, reset it in rename2() after inserting a checkpoint, increment it for each renamed instruction, reset it after a rollback, *etc*.
    - Replace `bool stall_branch(uint64_t bundle_branch)` and
      `bool stall_dispatch(uint64_t bundle_inst)` with:
      `bool stall_checkpoint(uint64_t bundle_chkpts)`
    - Modify rename2(), which is the caller of renamer::stall_checkpoint(), to count the number of checkpoints (bundle_chkpts) needed by the rename bundle according to the criteria discussed in Section 3.1.1. Note that a single instruction may satisfy multiple criteria (*e.g.*, both exception and amo, or both exception and csr, *etc*.), in which case the precedence order is: amo or csr (2 checkpoints: 1 checkpoint before and 1 checkpoint after), exception (1 checkpoint before), mispredicted branch (1 checkpoint after). Within this same counting loop, rename2() must determine if instr_renamed_since_last_checkpoint *will* reach max_instr_bw_checkpoints, within or right after the rename bundle (without literally incrementing instr_renamed_since_last_checkpoint: that must be done only as instructions are renamed, *i.e.*, when the rename bundle is deemed to advance). If so, bundle_chkpts should include this additional checkpoint criterion. rename2() will then pass in bundle_chkpts to renamer::stall_checkpoint(), and the latter will return whether or not to stall the rename bundle due to insufficient checkpoints. Note, because Phase 1 demands exact checkpoint placement at criteria instructions for simplified recovery, you must stall rename2() if there are not enough checkpoints. (Keep the counting of bundle_dst for passing into renamer::stall_reg(), as usual.)
    - renamer::checkpoint() must be adjusted, including no longer returning a branch ID since checkpoints are not "owned" by any particular instruction (void renamer::checkpoint() instead of uint64_t renamer::checkpoint()). When the rename bundle is allowed to advance (sufficient renaming resources: checkpoints and physical registers), modify rename2() to call renamer::checkpoint() before and/or after each criteria instruction with the following precedence:

```
if (amo || csr) {insert checkpoints before and after}
else if (exception) {insert checkpoint before}
else if (misp. branch || (instr_renamed_since_last_checkpoint == max_instr_bw_checkpoints))
{insert checkpoint after}
```

- In renamer::checkpoint(), as per the CPR paper: (1) Checkpoint the current RMT and the current unmapped bits of physical registers. (2) Increment the usage counter of each physical register in the newly created checkpoint.[1]
  Do not bother checkpointing the Free List head pointer and head phase bit as that style of Free List recovery doesn't work with CPR.[2]

- **Four different instruction counters associated with each checkpoint:** We will need four different instruction counters affiliated with each checkpoint.
  - (1) Uncompleted instruction counter: This is the total number of uncompleted instructions between this checkpoint and the next one, that have been renamed but not yet completed. This is the same instruction counter described in the paper and class lectures, quizzes, and exams. An instruction increments its affiliated checkpoint's uncompleted instruction counter when it is renamed and decrements it when it is completed. It is used by the retirement unit to determine when the oldest checkpoint can be reclaimed, which is when there exists a next checkpoint and when the uncompleted instruction counter is 0.
  - (2) Load counter, (3) store counter, and (4) branch counter: These counters are needed by the retirement unit to know how many loads, stores, and branches to commit from the Load Queue (LQ), Store Queue (SQ), and Branch Queue (BQ) (which is in the Fetch Unit), respectively, when the retirement unit is implementing so-called bulk commit of loads, stores, and branches. Since we got rid of the Active List and the Active List previously signaled load, store, and branch flags to guide LQ, SQ, and BQ commit calls, we need a substitute for this functionality. By recording the total number of loads, stores, and branches affiliated with each checkpoint, then the retirement unit knows how many calls to LQ, SQ, and BQ commit functions to make, when performing so-called bulk-commit (reclaiming the oldest checkpoint and advancing the committed register state to the next oldest checkpoint). Note that the load, store, and branch counters, affiliated with a checkpoint, are incremented when the instruction is renamed but NOT decremented when it completes (unlike the uncompleted instruction counter), because we need to record the number of such instructions and pass that information to the retirement unit via a revised renamer::precommit() function as will be discussed below.
    These three counters may also be used to restore the LQ tail, SQ tail, and BQ tail, when there is a rollback to a checkpoint. Checkpointing these queue pointers is tricky because checkpoints are created in rename2(), whereas the BQ is pushed in fetch2() and the LQ/SQ are pushed in dispatch(). Thus, it is recommended that these structures

[1] We'll assume the usage counters of all physical registers in the newly created checkpoint can be incremented in the current cycle. This is inconsistent with our modeling of a certain rate of decrementing usage counters when the checkpoint is freed later (see bullet "Course-grain retirement").

[2] Because of aggressive register reclamation (OOO pushing of free registers at the Free List tail), our traditional trick of bulk recovery of squashed instructions' registers doesn't work, *i.e.*, checkpointing and restoring the Free List head and head phase bit doesn't work in CPR. Squashed instructions will need to indirectly cause freeing of physical registers in a manner analogous to instruction completion, which is to say, whether an instruction completes execution or is selectively squashed, it decrements the usage counters of its source registers.

be rolled back using the known number of loads, stores, and branches between the oldest checkpoint and the rollback checkpoint. The LQ, SQ, and BQ tail pointers can be restored to offsets from their head pointers, the offsets equal to the total number of loads, stores, and branches prior to the rollback checkpoint (sum all prior checkpoints' counters, excluding the rollback checkpoint). This strategy should work for both exact checkpoint placement (Phase 1) and inexact checkpoint placement (Phase 2).

- **amo, csr, and exception flags associated with each checkpoint:** For the same reason that we associated load, store, and branch counters with each checkpoint, we must also record whether a checkpoint has an affiliated amo or csr instruction or exception. This functionality was previously supported by per Active List entry flags.
- **Each instruction inherits a checkpoint ID; each instruction conditionally increments the four instruction counters and conditionally sets the amo/csr flags of the corresponding checkpoint:** In rename2(), each instruction must inherit a checkpoint ID corresponding to the checkpoint that it is affiliated with, *i.e.*, the nearest prior checkpoint. This is also a convenient point at which to conditionally increment the four instruction counters and conditionally set the amo/csr flags of the checkpoint. Accordingly, add the following function to the renamer class:

  ```
  uint64_t renamer::get_checkpoint_ID(bool load, bool store, bool branch,
  bool amo, bool csr)
  ```

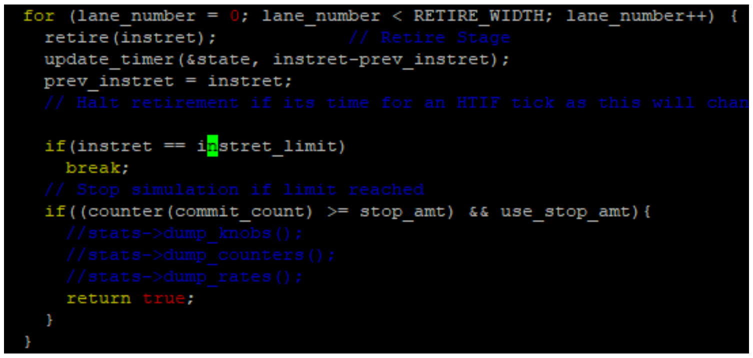
  This function should be called in rename2(). **You must correctly order the calls to renamer::get_checkpoint_ID() and renamer::checkpoint() (both of which are called in rename2()):** (1) For an amo or csr instruction, renamer::get_checkpoint_ID() should be called *after* the first call to renamer::checkpoint() and *before* the second call to renamer::checkpoint(), because the amo or csr instruction is affiliated with the checkpoint inserted before it. (2) For an exception instruction, renamer::get_checkpoint_ID() should be called *after* renamer::checkpoint(), because the exception instruction is affiliated with the checkpoint inserted before it. (3) For a mispredicted branch or reaching the instruction threshold, renamer::get_checkpoint_ID() should be called *before* renamer::checkpoint(), because the checkpoint is inserted after these criteria instructions.

  renamer::get_checkpoint_ID() returns the ID (*i.e.*, checkpoint number, *i.e.*, index into the checkpoint buffer) of the nearest prior checkpoint. Each instruction keeps its checkpoint ID in its payload for at least three reasons: (1) to decrement the uncompleted instruction counter of its affiliated checkpoint when it completes; (2) for a mispredicted branch to signal which checkpoint to roll back to (actually the next checkpoint, which is guaranteed to be right after the branch due to exact checkpoint placement in Phase 1); and (3) to selectively squash instructions in the IQ and execution lanes based on checkpoint IDs (for immediate branch misprediction recovery[3]). The caller passes in the type of the instruction via Boolean flags – load, store, branch, amo, and csr – so that the renamer class can conditionally increment the load, store, and branch counters and conditionally set the amo and csr flags of the

[3] An interesting observation is that, in principle, immediate recovery could be applied even to load violations and exceptions because any instruction may trigger a rollback to the nearest prior checkpoint and selective squashing is based on checkpoint IDs. Thus, in Phase 2, we should be able to support immediate load violation recovery (and avoid repeating the load violation by forcing the load to stall under ambiguity). While immediate exception recovery is possible mechanistically, the fact that we don't rename system registers mandates that exception handling be deferred until the offending instruction is non-speculative (pipeline_t::take_trap () modifies committed system register state).

corresponding checkpoint. Note that this function also unconditionally increments the uncompleted instruction counter of the corresponding checkpoint.

- **Posting exceptions, marking completion, *etc.*:** Modify renamer::set_exception() and every call to this function, to use the instruction's checkpoint ID (from its payload) instead of its defunct Active List index. Similarly, modify renamer::set_complete() and every call to this function, to use the instruction's checkpoint ID and decrement the uncompleted instruction counter of that checkpoint.
- **Preliminary simulator changes to prepare for coarse-grain retirement:** To prepare for coarse-grain retirement, we must first restructure the retirement process in the simulator. In particular, the retirement loop inside `pipeline_t::step_micro()` – which loops up to RETIRE_WIDTH times – must be replaced with a single call of the `pipeline_t::retire()` function and the `pipeline_t::retire()` function requires an additional argument.
    - Old: `void pipeline_t::retire(size_t& instret)`
      New: `void pipeline_t::retire(size_t& instret, size_t instret_limit)`
    - Old: The image below shows the retirement loop in `pipeline_t::step_micro()`.



```
for (lane_number = 0; lane_number < RETIRE_WIDTH; lane_number++) {
  retire(instret);              // Retire Stage
  update_timer(&state, instret-prev_instret);
  prev_instret = instret;
  // Halt retirement if its time for an HTIF tick as this will chan

  if(instret == instret_limit)
    break;
  // Stop simulation if limit reached
  if((counter(commit_count) >= stop_amt) && use_stop_amt){
    //stats->dump_knobs();
    //stats->dump_counters();
    //stats->dump_rates();
    return true;
  }
}
```

      New: Replace the loop, above, with the following *non-looping* code. (Note: subsequent text will explain where and how to place some of the displaced code into `pipeline_t::retire()` itself.). Note the additional argument passed into `retire()`.

```
    retire(instret, instret_limit);
    // Stop simulation if the instruction limit is reached.
    if ((counter(commit_count) >= stop_amt) && use_stop_amt) {
       return true;
    }
```

- **Coarse-grain retirement:** Augment the `pipeline_t` class, and modify the `renamer::precommit()`, `renamer::commit()`, and `pipeline_t::retire()` functions, as follows.
    - Add a new member variable to the `pipeline_t` class that represents the retirement state, since we will model coarse-grain retirement over one or more cycles with a state machine. Something like this (example names are based on their meaning, and a struct is use to conveniently aggregate all of the state variables):

```
typedef
enum {RETIRE_IDLE, RETIRE_BULK_COMMIT, RETIRE_FINALIZE}
retire_state_e;

typedef
struct {
   retire_state_e state;
   // The following seven variables should be passed by reference to
   // the corresponding arguments of the new renamer::precommit()
   // function.
   uint64_t chkpt_id;
   uint64_t num_loads_left, num_stores_left, num_branches_left;
   bool amo, csr, exception;
   // Keep track of the next logical register to process in the
   // oldest checkpoint.
   uint64_t log_reg;
} retire_state_t;

// This is the aggregated retirement state variable.
retire_state_t RETSTATE;
```

- `renamer::precommit()` must be modified as follows:
  `bool renamer::precommit(uint64_t &chkpt_id, uint64_t &num_loads, uint64_t &num_stores, uint64_t &num_branches, bool &amo, bool &csr, bool &exception)`
  It returns true to signal that so-called bulk-commit of the instructions logically between the oldest and next oldest checkpoints may proceed. That is, it returns true if there exists a checkpoint after the oldest checkpoint and if all instructions between them have completed.

  Here's an example of how to call it:

```
bool proceed = REN->precommit(RETSTATE.chkpt_id,
RETSTATE.num_loads_left, RETSTATE.num_stores_left,
RETSTATE.num_branches_left, RETSTATE.amo, RETSTATE.csr,
RETSTATE.exception);
```

- We will model a certain rate of decrementing usage counters of physical registers in the to-be-freed oldest checkpoint. Accordingly, modify the `renamer::commit()` function as follows:
  `void renamer::commit(uint64_t log_reg)`
  For example, to tell the renamer to decrement the usage counter of the physical register mapped to logical register 7 in the oldest checkpoint:
  `REN->commit(7).`

- The `pipeline_t::retire()` function should implement a new state machine with the following states:

  RETIRE_IDLE: In this state, call renamer::precommit(). If proceed is false, just do nothing and return from the retire() function.

  On the other hand, if proceed is true, you must first check the `RETSTATE.exception`, `RETSTATE.amo`, and `RETSTATE.csr` flags, and follow a similar process as exists in the baseline simulator with respect to these flags. Adapting the baseline code:

```
// Sanity checks of the 'amo' and 'csr' flags.
assert(!RETSTATE.amo || IS_AMO(PAY.buf[PAY.head].flags));
assert(!RETSTATE.csr || IS_CSR(PAY.buf[PAY.head].flags));
if (RETSTATE.amo || RETSTATE.csr) {
   // There should be only 1 instruction - the amo or csr -
   // between the oldest and next oldest checkpoint.
   // So the following assertions should succeed.
   assert(RETSTATE.num_loads_left <= 1);
   assert(RETSTATE.num_stores_left <= 1);
   assert(RETSTATE.num_branches_left == 0);
   // load and store are declared as local variables (bool)
   load = (RETSTATE.num_loads_left > 0);
   store = (RETSTATE.num_stores_left > 0);
}
if (!RETSTATE.exception) {
   if (RETSTATE.amo && !(load || store)) { // amo, excluding load-with-reservation
(LR) and store-conditional (SC)
      RETSTATE.exception = execute_amo();
   }
   else if (csr) {
      RETSTATE.exception = execute_csr();
   }

   // This is probably optional.
   // Just doing it out of completeness and adapting existing code.
   if (RETSTATE.exception)
      REN->set_exception(RETSTATE.chkpt_id);
}
```

Note that we are still in the RETIRE_IDLE state. After all the above logic, if `RETSTATE.exception` is true, then handle it in much the same way (if not exactly the same way) as the existing exception case code in the baseline simulator's retire() function (see the final else in the baseline pipeline_t::retire() function). Modify renamer::squash(), called from pipeline_t::squash_complete(), to properly implement a complete rollback (*e.g.*, rollback the RMT and unmapped bits using the oldest checkpoint). While it may be easier for you to reinitialize your Free List and physical register usage counters in a literal manner within renamer::squash(), another alternative that may reflect hardware is for squashed instructions (from within pipeline_t::squash_complete()) and squashed checkpoints (from within renamer::squash()) to decrement physical register usage counters and indirectly free certain physical registers in the process, using the existing aggressive register reclamation hardware. After exception recovery, return from the retire() function, still in the RETIRE_IDLE state.

On the other hand, if `RETSTATE.exception` is false, then transition from the RETIRE_IDLE state to the RETIRE_BULK_COMMIT state. The variables `RETSTATE.num_loads_left`, `RETSTATE.num_stores_left`, and `RETSTATE.num_branches_left`, have been initialized properly via renamer::precommit(). Initialize `RETSTATE.log_reg` to 0 upon transitioning from RETIRE_IDLE to RETIRE_BULK_COMMIT. This state variable keeps track of the number of logical registers processed in the oldest checkpoint so far.

RETIRE_BULK_COMMIT: We may remain in this state for one or more cycles, to model a certain rate of committing loads, stores, and branches from the LQ, SQ, and BQ, respectively, and to model a certain rate of decrementing usage counters of physical registers contained in the oldest checkpoint that is being freed. Each cycle that retire() is called, if the retirement state is RETIRE_BULK_COMMIT:

(1) Call `LSU.train(true /*load*/)` and `LSU.commit(true /*load*/, RETSTATE.amo`[4]`)` up to RETIRE_WIDTH times and decrement `RETSTATE.num_loads_left` for each such call, stopping this process entirely when `RETSTATE.num_loads_left` reaches 0 (note that it may initially be 0). Also capture and assert amo_success as in the baseline retire() code.
(2) Call `LSU.train(false /*store*/)` and `LSU.commit(false /*store*/, RETSTATE.amo)` up to RETIRE_WIDTH times and decrement `RETSTATE.num_stores_left` for each such call, stopping this process entirely when `RETSTATE.num_stores_left` reaches 0 (note that it may initially be 0). Also capture and assert amo_success as in the baseline retire() code.
(3) Call `FetchUnit->commit()`[5] up to RETIRE_WIDTH times and decrement `RETSTATE.num_branches_left` for each such call, stopping this process entirely when `RETSTATE.num_branches_left` reaches 0 (note that it may initially be 0).
(4) Call `REN->commit(RETSTATE.log_reg)` up to RETIRE_WIDTH times and increment `RETSTATE.log_reg` for each such call, stopping this process entirely when `RETSTATE.log_reg` reaches NXPR+NFPR, the total number of logical registers for RISCV.

We are ready to transition to the next state, when `RETSTATE.num_loads_left` is 0, `RETSTATE.num_stores_left` is 0, `RETSTATE.num_branches_left` is 0, and `RETSTATE.log_reg` is NXPR+NFPR. Note that these conditions may be reached in different cycles. Only when all loads, stores, branches, and logical registers have been processed, are we ready to transition to the next state.

But first, we must free the oldest checkpoint (note: we already decremented the usage counters of all physical registers contained therein, via repeated calls to `REN->commit(RETSTATE.log_reg)`). Create a new function, `void renamer::free_checkpoint()`, that tells the renamer to increment the head pointer of its checkpoint buffer, thereby freeing the oldest checkpoint and advancing the committed register state to the next oldest checkpoint. Call `REN->free_checkpoint()` just prior to transitioning to the next state.

Then, transition the state from RETIRE_BULK_COMMIT to RETIRE_FINALIZE.

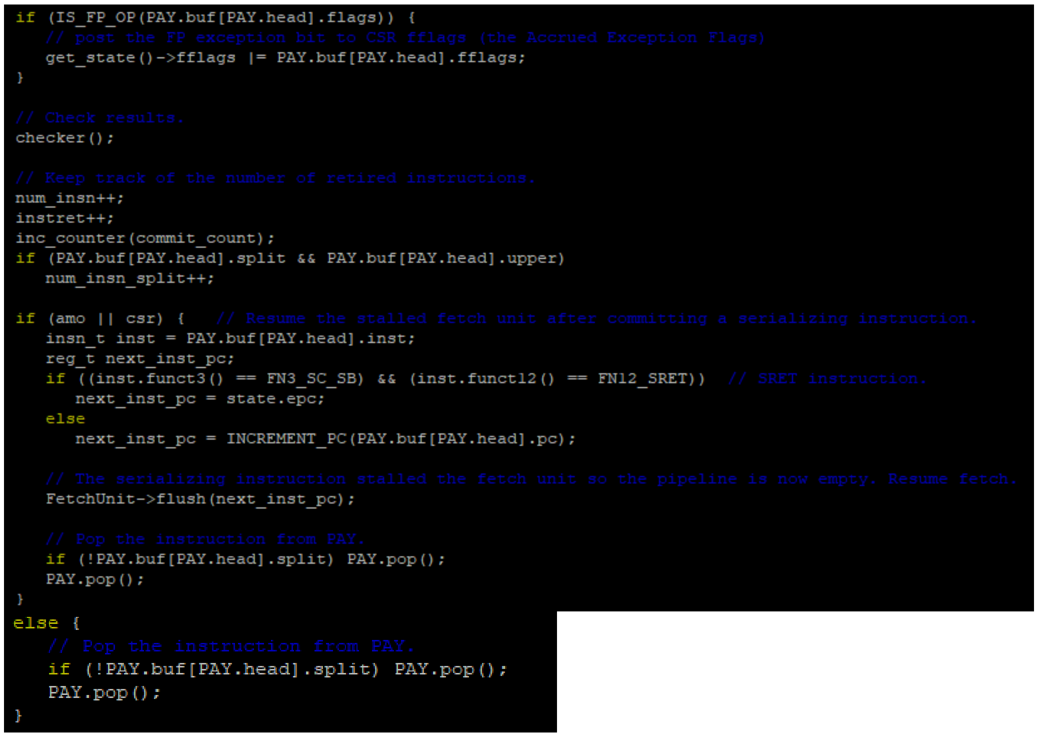
RETIRE_FINALIZE: We generally spend only one cycle in this state, except to pause for an HTIF tick (an idiosyncrasy of the RISCV base tools). This state is dedicated to (1) managing FP accrued exception flags, (2) checking all instructions affiliated with the oldest checkpoint, (3) counting retired instructions and pausing for HTIF ticks, (4) resuming the fetch unit with the correct PC as a final step for serializing instructions (amo and csr), and (5) popping instructions from PAY.

Implement a while loop to sequence through instructions in PAY, for as long as the instructions have a checkpoint ID that matches `RETSTATE.chkpt_id` (recall, the latter

[4] In the RETIRE_IDLE state, we (approximately) asserted that if there is an amo, that there is only one instruction affiliated with the oldest checkpoint. Thus, it should be ok to use a fixed amo flag here: if amo is true, there should be at most one call to LSU.commit().
[5] For simplicity, we need to get rid of the argument of fetchunit::commit(), which is used to assert that the committing branch's pred_tag matches the BQ head. It's a useful assert, but bulk-commit makes it inconvenient to locate this branch's entry in the PAY buffer.

was properly set via renamer::precommit()). Another stop condition for this loop is reaching the PAY tail – no more instructions in PAY, which implies no instructions fetched after the next checkpoint – which is possible due to serializing instructions (amo, csr) stalling the fetch unit until they retire, or an instruction cache miss after a checkpoint-criterion instruction, *etc*.

The purpose of this loop is to perform all of the remaining steps that are in the baseline retire() code, which you should reference and reuse. The only difference is that the baseline retire() code does it for a single retired instruction whereas here we are looping through all instructions affiliated with the newly freed checkpoint. As shown in the image below, the remaining steps for each instruction are: (a) manage FP accrued exception flags (fflags), (b) check the instruction (pipeline_t::checker()), (c) count the retired instruction, (d) resume the Fetch Unit after the amo/csr, and (e) pop the instruction from PAY. Academically speaking, you shouldn't interpret this loop as "cheating with a fake Active List". Steps (b), (c), and (e) are simulator artifacts, not real hardware[6]. (d) is hardware-implementable by affiliating not just the amo/csr flags with each checkpoint, but also the related payload information (same concept applies to a real Active List). I believe step (a) is for accumulating floating-point exceptions and deferring their handling in bulk, which is a useful concept for offloading work to accelerators without worry of aborting them in the middle (it's hard or impossible to reconstruct a precise state in the middle), and actually dovetails with the coarse-grain nature of CPR.



```
if (IS_FP_OP(PAY.buf[PAY.head].flags)) {
   // post the FP exception bit to CSR fflags (the Accrued Exception Flags)
   get_state()->fflags |= PAY.buf[PAY.head].fflags;
}

// Check results.
checker();

// Keep track of the number of retired instructions.
num_insn++;
instret++;
inc_counter(commit_count);
if (PAY.buf[PAY.head].split && PAY.buf[PAY.head].upper)
   num_insn_split++;

if (amo || csr) {   // Resume the stalled fetch unit after committing a serializing instruction.
   insn_t inst = PAY.buf[PAY.head].inst;
   reg_t next_inst_pc;
   if ((inst.funct3() == FN3_SC_SB) && (inst.funct12() == FN12_SRET))  // SRET instruction.
      next_inst_pc = state.epc;
   else
      next_inst_pc = INCREMENT_PC(PAY.buf[PAY.head].pc);

   // The serializing instruction stalled the fetch unit so the pipeline is now empty. Resume fetch.
   FetchUnit->flush(next_inst_pc);

   // Pop the instruction from PAY.
   if (!PAY.buf[PAY.head].split) PAY.pop();
   PAY.pop();
}
else {
   // Pop the instruction from PAY.
   if (!PAY.buf[PAY.head].split) PAY.pop();
   PAY.pop();
}
```

[6] Real processors have performance counters for measuring IPC, *etc*. Retired instruction counting can be done in CPR properly by noting how many total instructions were affiliated with the checkpoint, analogous to the load and store counts we affiliated with each checkpoint.

NOTES FOR THE ABOVE IMAGE:

- Replace "amo" and "csr" with `RETSTATE.amo` and `RETSTATE.csr` and make some sanity-check assertions with IS_AMO() and IS_CSR().
- Also, you can shift the redundant PAY popping logic out of the if/else to control-independent code.
- Remember to place the required `while ()` loop around the code in the image above, to sequence through PAY for as long as the `PAY.head` instruction has a checkpoint ID that matches `RETSTATE.chkpt_id` or until `PAY.tail` is reached.
- Moreover, you must append the following code *inside* the RETIRE_FINALIZE `while ()` loop, after popping the instruction from PAY:

```
update_timer(&state, 1);  // Update timer by 1 retired instr.
// Pause, but remain in the RETIRE_FINALIZE state for
// the next cycle, if it's time for an HTIF tick,
// as this will change state.
if (instret == instret_limit)
   return;   // Pause and remain in the state RETIRE_FINALIZE.
```

After the RETIRE_FINALIZE `while ()` loop, transition the state from RETIRE_FINALIZE to RETIRE_IDLE.

- **A note about incrementing/decrementing usage counters for *both* source and destination operands:** The CPR paper says that a *consumer* of a physical register should be reflected in the physical register's usage counter. The fact is, even a *producer* of a physical register should be reflected in the physical register's usage counter. For example, for the sequence below, p10's usage counter should be 3 (for the producer of p10 and two consumers of p10) instead of just 2 (for the two consumers of p10):

```
r5(p10) = …          // producer of p10
        = r5(p10)    // consumer of p10
        = r5(p10)    // consumer of p10
r5(p11) = …
```

Why should the producer of a physical register also be included in its usage counter? It is because of the possibility of "dynamically dead" producers. Sometimes a compiler generates code like this:

```
A: r5 = …
B: branch cond, D
C: r5 = …
D: … = r5   // D will get either A's (cond true) or C's (cond false) version of r5
```

At run-time, if `cond` is false, C's version of r5 kills A's version of r5 *without there being a consumer of A's version of r5*. That is, in this dynamic scenario, A is dynamically dead. Here's what could happen. When C's version of r5 is renamed, it will unmap A's version of r5. A's version's usage counter, based on the paper, will be 0 (no instructions source r5). Thus, as soon as A's version is unmapped, it will be aggressively freed… possibly (and likely) before producer A issues from the IQ. When A does finally issue, it could clobber the physical register that used to be assigned to it, but is now assigned to some future younger instruction, yielding an incorrect result. The solution is to include the producer, A, as a user of its own physical register, so that it will be held by A until it writes the physical register, even if there are no consumers of the physical register.

- **Branch misprediction recovery with exact checkpoint placement:**
  - Replace renamer::resolve() with renamer::rollback().
    `uint64_t renamer::rollback(uint64_t chkpt_id, bool next, uint64_t &total_loads, uint64_t &total_stores, uint64_t &total_branches)`
    The first argument, chkpt_id, is the checkpoint ID inherited by the offending instruction, *i.e.*, the nearest prior checkpoint. Offending instructions include mispredicted branches in Phases 1B and 2 and load violations in Phase 2. The second argument controls whether we rollback to chkpt_id (next = false) versus the checkpoint after chkpt_id (chkpt_id + 1, modulo the checkpoint buffer size) (next = true). If the checkpoint insertion logic intentionally inserted a checkpoint right after a branch (for example, for unconfident branches, which in Phase 1B is any mispredicted branch; or for confident branches in Phase 2 that fortuitously get a checkpoint right after it due to the maximum instruction threshold), then the branch will call renamer::rollback() with next = true. In Phase 1B, all mispredicted branches will call renamer::rollback() with next = true because we force checkpoints immediately after all mispredicted branches. For load violations in Phase 2, mispredicted loads will always call renamer::rollback() with next = false because the load itself must be squashed, refetched, and reexecuted.

    Regardless of which checkpoint we rollback to (chkpt_id or the next one), we'll refer to that checkpoint as the *rollback checkpoint*.

    In renamer::rollback():

    - Determine the rollback checkpoint from chkpt_id and next.
    - Assert the rollback checkpoint is valid/allocated, *i.e.*, it exists between the checkpoint buffer head and tail.
    - Restore the RMT and unmapped bits from the rollback checkpoint.
    - Generate a uint64_t "squash mask", wherein bits are set corresponding to checkpoint IDs of (1) the rollback checkpoint and (2) all allocated younger checkpoints (logically after the rollback checkpoint in program order), prior to squashing/freeing the latter younger checkpoints. For example, suppose we currently have five checkpoints out of eight maximum, where the oldest checkpoint has ID 2, the youngest checkpoint has ID 6, and the rollback checkpoint has ID 4. In this example, we want to generate a squash mask that will signal any instruction belonging to checkpoints 4, 5, or 6, to squash themselves. Considering only the low eight bits (upper bits should be 0), the squash mask = `01110000`. The renamer::rollback() function returns its generated squash mask.
    - The rollback checkpoint should be preserved (not squashed/freed). On the other hand, all allocated younger checkpoints should be squashed/freed. Decrement the usage counter of each physical register mapped in each squashed/freed checkpoint.

    - Reset the uncompleted instruction count, load count, store count, and branch count, all to 0, of the rollback checkpoint. Reset the amo, csr, and exception flags of the rollback checkpoint.
    - Calculate the total number of load instructions between the oldest checkpoint and the rollback checkpoint, by summing all the load counts affiliated with the oldest checkpoint (inclusive) through the rollback checkpoint (not inclusive, although it is moot because you reset its load count anyway). Do the same for stores and branches, using the store counts and branch counts, respectively. The total number of loads, total number of stores, and total number of branches, should be returned to the caller via the corresponding pass-by-reference arguments of renamer::rollback().

- In the IQ and backend execution lanes' pipeline registers, replace the instruction's branch mask (no longer inherited) with the instruction's inherited checkpoint ID.
- Replace `void issue_queue::squash(unsigned int branch_ID)` with `void issue_queue::squash(uint64_t squash_mask)`
  Change the way selective squashing is performed in the IQ. Instead of checking for a passed-in branch ID in the instruction's inherited branch_mask, check if the instruction's inherited chkpt_id is within the passed-in squash_mask (test the squash_mask bit corresponding to chkpt_id). Squash the instruction if its chkpt_id is covered by the squash_mask. This means not only removing it from the IQ, but also decrementing the usage counters of its source and destination physical registers. You can identify all physical registers used by the squashed instruction, using its PAY index available in its IQ entry to reference its PAY buffer entry.
- Replace `void pipeline_t::resolve(unsigned int branch_ID, bool correct)` with `void pipeline_t::selective_squash(uint64_t squash_mask)`.
  Unlike before, this function is only called when there is a rollback. As before, full-squash the frontend pipeline stages and selectively squash the IQ and execution lanes. Changes: (1) Use the updated method for selective squashing, based on the squash_mask covering the instruction's inherited chkpt_id (see the preceding bullet). (2) For each squashed instruction, decrement the usage counters of its source and destination physical registers (its PAY index is available in the pipeline register type).
- Modify the writeback() stage to only do something for a mispredicted branch. I don't believe anything needs to be done for a correctly predicted branch. Adjust the branch misprediction recovery process as follows:
    - Replace the call `REN->resolve()` with `REN->rollback()`, pass in appropriate arguments to it, and record its return value (squash mask).
    - Replace the call to `pipeline_t::resolve()` with a call to `pipeline_t::selective_squash()`, passing the squash mask to it.
- For Phase 2, you will need to modify three function calls (and the implementations of these functions) in the writeback() stage's recovery code. These changes are not necessary in the case of exact checkpoint placement, as explained below, but you may want to begin these changes. The changes are with respect to the calls `FetchUnit->mispredict()`, `LSU.restore()`, and `PAY.rollback()`, to accommodate different arguments (discussed below).

    - `fetchunit_t::mispredict()`: Technically speaking, in Phase 1B, you can still use the branch's pred_tag to correctly rollback the Fetch Unit's BQ. This is due to exact checkpoint placement. Nonetheless, in preparation for inexact checkpoint placement, I recommend adjusting this function to take in total_branches (gotten from renamer::rollback()) instead of the branch's pred_tag. Within this function, you can infer which BQ entry to rollback to using the BQ head entry and total_branches. Perhaps you can start with no change, get Phase 1B working, and then make the above change and get it working for Phase 1B. Additional complicated changes will be needed to `fetchunit_t::mispredict()` and the fetch1() stage in the context of inexact recovery (*e.g.*, to distinguish exact vs. inexact recovery cases, to reuse branch predictions/outcomes from the BQ between the rollback point and the old BQ tail, *etc*.).
    - `lsu::restore()`: Technically speaking, in Phase 1B, you can still use the branch's LQ/SQ indices to correctly rollback the LSU. This is due to exact checkpoint placement. Nonetheless, in preparation for inexact checkpoint placement, I recommend adjusting this function to take in total_loads and total_stores (obtained from renamer::rollback()) instead of the branch's LQ/SQ indices. Within this function, you can restore the LQ/SQ tails using the LQ/SQ heads and total_loads/total_stores. Perhaps you can start with no change, get Phase 1B working, and then make the above change and get it working for Phase 1B.
    - `payload::rollback()`: Technically speaking, in Phase 1B, you can still use the branch's PAY index to correctly rollback the PAY buffer. This is due to exact checkpoint placement. Nonetheless, in preparation for inexact checkpoint placement, I recommend adjusting this function to take in a checkpointed PAY index instead of the branch's PAY index. TODO: add a checkpointed PAY index to each checkpoint. Perhaps you can start with no change, get Phase 1B working, and then make the above change and get it working for Phase 1B.

- **Aggressive register reclamation:**
  - It is recommended that you add the following public member functions to the renamer class, for incrementing and decrementing physical register usage counters. If they are public, they can be called from inside and outside the renamer class. The table below is an attempt to comprehensively document all functions that should call `renamer::inc_usage_counter()` or `renamer::dec_usage_counter()`, and for which physical registers.

```
// Increment the usage counter of physical register "phys_reg".
void renamer::inc_usage_counter(uint64_t phys_reg);

// Decrement the usage counter of physical register "phys_reg".
// Before decrementing, assert the usage counter > 0.
// After decrementing, check if phys_reg's unmapped bit is 1 and
// usage counter is 0; if so, push phys_reg onto the Free List.
void renamer::dec_usage_counter(uint64_t phys_reg);
```

function	Which functions call it (I may be missing some… you will figure it out during implementation).
`renamer::inc_usage_counter()` (could be a private function?)	`renamer::rename_rsrc()`, for sources `renamer::rename_rdst()`, for destinations `renamer::checkpoint()`, for checkpointed physical registers

`renamer::dec_usage_counter()`	`renamer::read()`, for sources
	`renamer::write()`, for destinations
	`renamer::commit(log_reg)`, for the physical register mapped to log_reg in the oldest checkpoint
	
	// Functions involved in a complete squash:
	`pipeline_t::squash_complete()`, for sources and destinations of squashed instructions in the Dispatch stage and execution lanes
	➔ `renamer::squash()`, for physical registers mapped in squashed checkpoints
	➔ `issue_queue::flush()`, for sources and destinations of squashed instructions in the IQ
	➔ `lsu::flush()`, for destinations of squashed loads that haven't written the PRF yet because they stalled (LQ[i].addr_avail && !LQ[i].value_avail) // TRICKY CASE
	
	// Functions involved in a selective squash:
	`renamer::rollback()`, for physical registers mapped in squashed checkpoints
	`lsu::restore()`, for destinations of squashed loads that haven't written the PRF yet because they stalled (LQ[i].addr_avail && !LQ[i].value_avail) // TRICKY CASE
	`pipeline_t::selective_squash()`, for sources and destinations of squashed instructions in the Dispatch stage and execution lanes
	➔ `issue_queue::squash()`, for sources and destinations of squashed instructions in the IQ

- It is recommended that you add the following private member functions to the renamer class, for managing the unmapped bits of physical registers. The table below is an attempt to comprehensively document all functions that should call `renamer::map()` or `renamer::unmap()`, and for which physical registers.

```
// Clear the unmapped bit of physical register "phys_reg".
void renamer::map(uint64_t phys_reg);
```

```
// Set the unmapped bit of physical register "phys_reg".
// Check if phys_reg's usage counter is 0; if so,
// push phys_reg onto the Free List.
void renamer::unmap(uint64_t phys_reg);
```

Function	Comment
`renamer::rename_rdst()`	Call `renamer::map()` for the physical register popped from the Free List and entered into the RMT. Call `renamer::unmap()` for the physical register that was displaced in the RMT.

`renamer::squash()`
`renamer::rollback()`
	While looping through the checkpointed unmapped bits (from the oldest checkpoint for `renamer::squash()`; from the rollback checkpoint for `renamer::rollback()`):
	• If the unmapped bit of a given physical register should be restored to 0, call `renamer::map()` for that physical register.
	• If the unmapped bit of a given physical register should be restored to 1, call `renamer::unmap()` for that physical register.

3.1.3. Phase 1 validation, experiments, and analysis

3.2. Phase 2: Inexact checkpoint placement and recovery (complications with forward progress)

3.3. Phase 3: Confidence mechanism

3.4. (stretch goal) Phase 4: Hierarchical store queue

3.5. Other stretch goals

- Drain-based squashing of instructions
- Detailed constraint-based modeling of aggressive register reclamation and checkpointing
- OOO checkpoint reclamation

4. Grading

4.1. Submitting and self-grading your simulator

4.2. Simulator scoring

4.3. Report

4.4. 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. This is a necessity due to end-of-semester constraints.