CS520, Spring 2018: Homework 6

Due:	End of exam week
Late penalty after:	End of exam week
Cutoff:	Hardly any flexibility. We are limited by when I have to turn in grades.

This is your final coding project for the course. I am giving you as much time as is practical to complete it, but I cannot give you hardly any slack with regard to turning it in late. A few hours past the deadline is okay. But even a whole day late is going to require justification **in advance**. I have a deadline for turning in grades, and the graders need significant time to perform the grading or so many submissions.

Therefore you should set your own personal deadline that is days earlier than mine. You can easily compute this by taking the max of the number of days late you have been on all other assignments and then subtract that from my deadline to get yours.

Under special circumstances, there is the option to give you a Incomplete grade, but I also don't want to have to fill out a bunch of grade-change forms over the summer.

Main objective

Allow branch instructions to pass through Decode without all of their dependencies satisfied.

How to affect your score by being proactive

I'm going to propose a little "game" to encourage proactive work on this assignment. If I get too many complaints about this being unfair or something, then I'll rescind this suggestion.

First of all turning in a working assignment on time gets you an automatic perfect score. The only way you could shoot yourself in the foot is if you plagiarized your code from other students.

The graders have been running MOSS on earlier assignments and have found evidence of what appears to be inappropriate code sharing with heavy refactoring in an attempt to hide the fact that the copying. Once that analysis complete, those students are in for some serious trouble, unless they contact us before we contact them.

This plagiarism analysis will be performed very carefully for this final assignment as well.

The “game” here is meant as a way to reward proactive students, and it is meant to get struggling students past major hurdles. The rules are as follows:

- For me to find out about any of this, you have to properly ping me in Discord using @theosib. Asking questions in public is highly encouraged, although I may answer some in private.
- The “points” I’m talking about here are generally small but could make a significant difference in your grade, especially if
- Asking good questions (anywhere, but tagged) will earn points in your favor. Points are higher for question asked **earlier**. Good questions asked **publicly** will also earn more points.
- Dumb questions (asked directly to me) are generally neutral, but if you could find the answer by searching this document or any of course slides, that could actually count against you. Translation: READ STUFF and don’t waste my time.
- When students get stuck and need help, I will try to help them. Help provided towards the start of exam week counts in your favor. Help provided towards the end is neutral. Remember, if you don’t ask for help, you may not finish the assignment, and you score will be much worse.
- Sometimes, the only way to get past a hurdle is for me to paste code to you in private. If I have to do this, it will count against you (more so as time passes). But that’s way better than failing.

I’m not actually going to be assigning numerical points here. I’m going to keep records of specifically what you did and refer to them when making grading decisions.

Requirements

- Branch prediction -- Have Fetch predict conditional branches and send them on Decode and the IQ speculatively.
- Branch Resolution Unit (BRU) -- Functional unit that computes whether or not a conditional branch **should** have been taken.
- Reorder buffer -- Queue that receives instructions in program order, receives completions out of order, and retires in-order. Unified with physical register file (PRF).
- “Precise state” architectural register file (ARF) to which instructions get retired.
- Load/Store Queue -- Uncommitted stores and data forwarding from LOAD to STORE.
- [need to add missing text here, but everything is described in detail below.]

Changes to go from HW5 to HW6

General notes

When setting properties on a Latch, use the setProperty() method. When setting properties in Globals, it is almost always correct to use the setClockedProperty() method instead.

Infrastructure

CpuCore

CpuCore.putRetiredSet() allows the Retirement stage to communicate retired instructions (each cycle) to other stages. This is currently just used by the memory unit to know which STORE instructions it can finally commit. **CpuCore.getRetiredSet()** read returns this set.

InstructionBase

InstructionBase now has fields to keep track of assigned ROB index, fault code, branch prediction and branch resolution. Before an instruction's dest reg is renamed (and assigned a ROB index), instructions converted to String will indicate the program file address; afterwards, the ROB index is printed instead. New methods:

- public void setReorderBufferIndex(int ix) { rob_index = ix; }
- public int getReorderBufferIndex() { return rob_index; }
- public void setBranchPrediction(EnumBranch p) { branch_prediction = p; }
- public EnumBranch getBranchPrediction() { return branch_prediction; }
- public void setBranchResolution(EnumBranch p) { branch_resolution = p; }
- public EnumBranch getBranchResolution() { return branch_resolution; }
- public void setFault(EnumFault f) { fault = f; }
- public EnumFault getFault() { return fault; }

PipelineStageBase and IPipeStage

PipelineStageBase.registerFileLookup() can how handle looking into the retirement ARF (the register file into which results are moved when an ROB entry is retired). When a RAT entry contains a negative number (specifically -1), it means that the architectural register renames to its own index in the retirement ARF.

Normally, if you have a pipeline stage with 0 or 1 inputs and 0 or 1 outputs, the **compute(Latch,Latch)** will be called. Otherwise, the **compute()** method with no arguments will be called. If you want to force a stage to always use the no-argument **compute()** method, you can call **disableTwoInputCompute()** in your constructor.

IProperties and PropertiesContainer

A new property type is supported, an array of instructions:

- public InstructionBase[] getPropertyInstructionArr(String name);

In my implementation, the ROB and PRF are separate arrays that represent different information about the same instruction at the same index. The PRF is of type RegisterFile (with its physical mode flag set), while the ROB is just an array of instructions.

In properties containers, properties are accessed by String name. Unfortunately, having to type the same strings in multiple places risks introducing typos. So I have put lots of pre-defined property names into IProperties. Some are used in Globals, while others are used in Latches, to communicate between pipeline stages.
Common Latch properties:

- public static final String RESULT_VALUE = "result_value";
- public static final String RESULT_FLOAT = "result_float";
- public static final String RESULT_FAULT = "result_fault";
- public static final String LOOKUP_BRANCH_TARGET = "lookup_branch_target";

Globals:

- public static final String PROGRAM_COUNTER = "program_counter";
- public static final String MAIN_MEMORY = "main_memory";
- public static final String REG_BRANCH_TARGET = "reg_branch_target";
- public static final String FETCH_BRANCH_STATE = "fetch_branch_state";
- public static final String CPU_RUN_STATE = "cpu_run_state";
- public static final String REGISTER_FILE = "register_file";
- public static final String REGISTER_ALIAS_TABLE = "register_alias_table";
- public static final String ARCH_REG_FILE = "arch_reg_file";
- public static final String REORDER_BUFFER = "reorder_buffer";
- public static final String ROB_USED = "rob_used";
- public static final String ROB_HEAD = "rob_head";
- public static final String ROB_TAIL = "rob_tail";
- public static final String RECOVERY_PC = "recovery_pc";
- public static final String RECOVERY_TAKEN = "recovery_taken";

When copying properties between latches, you can pass RESULT_PROPS_SET as the second argument to copyAllPropertiesFrom() to copy only results fields.

For reasons explained below, there are three branch-related states for the Fetch stage:

- Fetch is fetching instructions normally:
- public static final int BRANCH_STATE_NULL = 0;
- Fetch encountered a branch instruction whose program address comes from a register, so Fetch is waiting for Decode to look up that register's value:
- public static final int BRANCH_STATE_WAITING = 1;
- Decode is responding to Fetch with a target address (in REG_BRANCH_TARGET):
- public static final int BRANCH_STATE_TARGET = 2;

There is also a set of global execution states for the CPU:

- Not started:
- public static final int RUN_STATE_NULL = 0;
- Running normally:
- public static final int RUN_STATE_RUNNING = 1;
- A fault (mispredicted branch) has been detected. Stop fetching NEW instructions until the CPU is finished being reset to the precise state of the mispredicted branch. Meanwhile, the rest of the CPU has to keep executing to complete instructions that fall between the faulting instruction and the head of the ROB:
- public static final int RUN_STATE_FAULT = 2;
- The Retirement stage has reached the faulting instruction at the head of the ROB. Components of the CPU are now being requested to discard instructions that came AFTER the faulting instruction. This includes dropping instructions from IQ and LQS and early termination of long-running instructions like divides.
- public static final int RUN_STATE_FLUSH = 3;
- This state is entered once the CPU has been cleared of all pending instructions. Along with setting this state, the Retirement stage has to inform Fetch about where it must start executing again. Global properties RECOVERY_PC and RECOVERY_TAKEN are used to inform Fetch as where to restart (a branch instruction, specifically) and the correct resolution of that branch.
- public static final int RUN_STATE_RECOVERY = 4;
- A HALT instruction has reached the head of the ROB.
- public static final int RUN_STATE_HALTING = 5;
- Memory STORE retirement/commitment lags that of retirement, so everything stops while in the RUN_STATE_HALTING state, except the memory unit, which is given time to commit any remaining STOREs before stopping and switching the state to HALTED, which stops the simulation.
- public static final int RUN_STATE_HALTED = 6;

RegisterFile and IRegFile

The “RENAMED” flag has had its name changed to “UNMAPPED.” Registers also now have RETIRED and FAULT flags. There are also new methods for getting/setting those flags:

- public void markFault(int index, boolean has_fault);
- default public boolean isUnmapped(int index) { return isRenamed(index); }
- public boolean isRetired(int index);
- public boolean hasFault(int index);
- default public void markUnmapped(int index, boolean is_unmapped) {
-     markRenamed(index, is_unmapped);
- }
- public void markRetired(int index, boolean is_retired);

Operand

Operand now has a **getOrigRegisterNumber()** method to return the original architectural register number for an operand that was renamed to a physical register.

Descriptions of changes

Fetch

The Fetch stage is now responsible for all branch handling. Whether or not a branch is taken is determined either by the type of branch (JMP and CALL are always taken) or a branch predictor (for BRA).

On the other hand, the branch **target** must be known precisely. If the target is a label, it is already known. If the target is a register, my solution is to put Fetch into a paused state. It sends the branch along to Decode and stalls, waiting for Decode to respond with the target address is read from the register file or picked up from forwarding. My solution was to mark the instruction being send to Decode with a property (LOOKUP_BRANCH_TARGET) that indicates which operand of the branch instruction corresponds to the target address (0 for JMP, 1 for CALL and BRA).

When the Fetch stage gets a HALT instruction, it must permanently cease fetching any more instructions.

When Fetch observes that the global CPU_RUN_STATE is set to RUN_STATE_FAULT, it must cease fetching instructions until the run state changes to RECOVERY.

When Fetch observes that the global CPU_RUN_STATE is set to RUN_STATE_RECOVERY, it must begin fetching instructions again, starting at the address in the global RECOVERY_PC. This will only ever happen for a mispredicted conditional BRA instruction. To get the correct resolution, use the global RECOVERY_TAKEN.

In my implementation, Fetch is responsible for changing the run state from RUN_STATE_RECOVERY back to RUN_STATE_RUNNING.

If a conditional branch (BRA) instruction has a **label** target, use the following prediction rules:

- If the label address is less than the current PC, predict taken.
- Otherwise predict NOT taken.

If a conditional branch has a **register** target, always predict taken. Since the target is unknown (a prediction would have to wait until Decode responded with the target address), and that would take too much coding effort. Besides, none of our code examples have conditional branches with register targets.

Decode

The Decode stage now has one input and two outputs, so it will naturally implement the no-argument compute() method. It will fetch its input by doing **Latch input = readInput(0);**

When in the following run states, Decode should consume its input and return:

- HALTING, HALTED, FAULT

After checking run states, Decode should check to see if there is room in the ROB. If the ROB is full, Decode should use setResourceState to indicate the reason it is stalling and return (without consuming its input). Optionally, you can keep track of the number of used ROB entries in the Global ROB_USED. Alternatively, you can compare ROB_HEAD and ROB_TAIL indices:

- The ROB is empty if ROB_HEAD and ROB_TAIL have the same value.
- The ROB is full if the ROB_TAIL is one behind the ROB_HEAD, as in:
    - int rob_head = globals.getPropertyInteger(ROB_HEAD);
    - int rob_tail = globals.getPropertyInteger(ROB_TAIL);
    - boolean rob_full = ((rob_tail + 1) & 255) == rob_head;
- Note that the second method reduces the ROB and PRF capacity by 1, but I’m pretty sure it never gets full while running our benchmarks.

Decode determines the new destination physical register number for each instruction by retrieving the value of the Global ROB_TAIL. After duplicating the Latch and after selecting the ROB/PRF index, use the method setReorderBufferIndex to inform the instruction as to where it will end up in the PRF and ROB.

Decode no longer needs to support squashing of fall-through instructions. In fact, one of only two ways that Decode treats branches as special is to look up branch register targets (LOOKUP_BRANCH_TARGET property exists on the input Latch). The source register be valid or cannot be captured through forwarding. In this case, it is easiest to have Decode just stall until the appropriate source operand value becomes available. (There are alternatives, but they won't benefit the workloads we are running.) This should be done **after** calling registerFileLookup and forwardingSearch on the duplicate of the input Latch.

Decode has two outputs. Memory instructions go to the output leading to the load/store queue (LSQ). All others go to the IQ. Use the **newOutput(int output_num)** method to get a new output Latch. **If the appropriate output cannot accept work, then Decode must stall without consuming input or taking any other action.**

The other special case for branches pertains to CALL instructions. Change the operands so that the branch resolution unit (BRU) will add 1 to the PC, just as you have seen in earlier implementations:

```
if (opcode == EnumOpcode.CALL) {
   Operand pc_operand = Operand.newRegister(Operand.PC_REGNUM);
   pc_operand.setIntValue(ins.getPCAddress());
   ins.setSrc1(pc_operand);
   ins.setSrc2(Operand.newLiteralSource(1));
}
```

If the instruction requires a writeback, then rename the destination register as follows:

- Retrieve the PRF and RAT from Globals.
- Remember the architectural register number and use this to index the RAT and PRF.
- Remember the old physical register number stored in the RAT at that index.
- If the old physical register number is negative, then the architectural register had been mapped to its eponymous entry in the ARF.
- If the old physical register number is positive, then mark it as UNMAPPED in the PRF.
- Claim the register, setting USED and INVALID and clearing FLOAT, UNMAPPED, RETIRED, and FAULT by using the RegisterFile method **markNewlyAllocated()**.

If the instruction DOES NOT require a writeback, it is still necessary to properly claim the physical register whose index is the same as the ROB tail by calling the **markNewlyAllocated()** method. However, none of the other steps above should be done. For instance, for a STORE instruction, there is no destination register, so no architectural register should be renamed.

The final steps are:

- Store the instruction into the output Latch
- Call the write() method on the output Latch
- Store the instruction into the ROB entry indexed by the ROB tail.
- Increment the ROB tail index. Wrap back to zero if the index goes past the end of the array.
- Update the ROB_USED value if you choose to use that variable.
- Call the **consume()** method on the input Latch.

Branch Resolution Unit

Branch instructions are dispatched to the IQ like most other instructions. When all inputs are available, they are then issued from the IQ to the BRU. The BRU is a new single pipeline stage specifically for resolving conditional branches. Additionally, to avoid contention for other compute resources, JMP and CALL instructions are sent through the BRU (instead of Execute).

JMP instructions pass through with no special action taken.

CALL instructions must compute a result value. This is the sum of the values of src1 and src2 operands and is stored in the output latch using the **setResultValue()** method.

BRA instructions compute the actual branch resolution (taken or not taken) based on the comparison and the value of oper0. Use the InstructionBase method **setBranchResolution()** to specify the correct outcome, EnumBranch.TAKEN or EnumBranch.NOT_TAKEN.

When the branch prediction (**ins.getBranchPrediction()**) and the branch resolution (**ins.getBranchResolution()**) do not match, mark the instruction as having a fault (**ins.setFault(InstructionBase.EnumFault.BRANCH)**).

Floating point and integer divide units

Div units should check the global CPU_RUN_STATE. If the state is set to RUN_STATE_FLUSH, then they should set their stall counter so that any pending divide instruction terminates immediately. The instruction must still pass through to Writeback, but prematurely.

When the processor is in the FLUSH state, all executing instructions are to be flushed and discarded, and Writeback knows when that is the case when core.numCompleted() and core.numIssued() return the same value.

GlobalData

All of my changes to this class will be provided to you.

IssueQueue

Branch instructions are sent to the BRU. No memory instructions are sent through the IQ.

After reading the input Latch, the IQ should check the CPU_RUN_STATE. In RUN_STATE_FAULT and RUN_STATE_FLUSH, inputs should be consumed without further processing. However, the IQ cannot simply bail out at this point. In the FAULT state, there may be IQ entries that precede the faulting instruction in program order. The easiest way to fool subsequent code into thinking that there is no new instruction coming in is to call the setInvalid() method on the input Latch **after** having called consume().

When in the state RUN_STATE_FLUSH, all IQ entries must be cleared.

Load/Store Queue (LSQ)

The LSQ is a 32-entry instruction queue similar to is similar to the IQ in many ways, but there are key differences as well:

- The LSQ contains only LOAD and STORE instructions.
- LSQ entries are maintained in the original program order. If an instruction must be removed, succeeding entries must be shifted to fill the gap.
- LOAD instructions are deleted as soon as they are issued.
- STORE instructions are removed only in program order and only upon **retirement**.
- The LSQ is the first stage of the MemUnit functional unit. The remaining two stages are meant to model the latency of accessing a data cache, so we shall refer to those stages as “**The DCache**.” The first stage of the DCache computes the address (especially important for forwarding address operands to LOAD instructions), and the second accesses memory.

Tips:

- The LSQ stage has a single input and a single output, but we need to receive new instructions independently of sending out older ones. Therefore, call **disableTwoInputCompute()** in the constructor and use the no-argument **compute()** method.
- In the compute() method, maintain a boolean flag to remember whether or not you’ve already written to the output.

- Always remember that you need to call the **write()** method on an output Latch in order to submit it to the next stage.
- To retrieve the input Latch, do this:
    - Latch input = readInput(0);
- To allocate an output Latch, do this:
    - Latch output = newOutput(0);

The following steps are listed in the order I implemented them in.

The CpuCore has a method **getRetiredSet()** that contains the ROB indices of all instructions just retired. Use this to mark all retired STORE instructions as retired also in the LSQ. Retired STOREs are the sent one-by-one through to the remaining stages of the MemUnit to be committed (this is the time when they are actually written to main memory).

Next, the LSQ should check the CPU_RUN_STATE. In RUN_STATE_FAULT, RUN_STATE_FLUSH, RUN_STATE_HALTING, the input Latch should be consumed and invalidated just like with the IQ. Also, in the states RUN_STATE_FLUSH and RUN_STATE_HALTING, all LSQ entries should be deleted **except** those STORE entries marked "retired."

When in the run state RUN_STATE_HALTING, if there are still "retired" STORE entries, leave the CPU in this state and continue processing the remaining retired stores. When in the run state RUN_STATE_HALTING, it there are **zero** remaining retired STORE entries, change the processor state to RUN_STATE_HALTED.

STORE instructions are only removed when they retire. As an optional optimization, if Decode wants to dispatch an instruction but the LSQ is full, check to see if the first (oldest) entry in the LSQ is a retired STORE. If so, you can commit that store immediately and delete it from the LSQ. This will free up a slot for the new memory instruction. See below on how to commit a STORE.

When Decode wants to dispatch a memory instruction (when !input.isNull()), check to see if there is a free entry in the LSQ. If there is, place the new memory instruction into the last entry of the queue and keep track of the fact that the number of used entries has increased. Also, don't forget to call **consume()** on the input Latch. If the LSQ is full, then simply decline to call **consume()** on the input Latch.

Use forwarding to capture results from completions to any LSQ entries requiring those registers as inputs.

After that is a good point to fill a hashmap listing all instructions in the LSQ, for diagnostic purposes. I used the reorder buffer index (**getReorderBufferIndex()**) as the key and the string representation of the instruction the value.   I also concatenated to the instruction string other status information for each instruction (ready, completed, retired).

The next thing to do is look for an instruction to issue, and there is a priority order. Before you can do that, check to make sure you **can** issue an instruction to the remaining stages of the MemoryUnit (the DCache). If you have already issued an instruction OR the next stage indicates that it cannot accept new work, set the activity string and return:

```
setActivity(String.join("\n", doing.values()));
return;
```

Optional optimization: The only LOAD instruction that can be issued with forwarding to the DCache is one that is in the first slot of the LSQ. (All other LOADs must have fully-known addresses in order to check them against STORE addresses.) If the first LSQ slot contains a LOAD, and all of the LOAD's register input operands are either valid now or will be on the next cycle, then you can issue that to the next stage. For any operands that will be available on the next cycle, set the appropriate "forward#" properties on the output Latch. Also, see below for information on how to specify to the DCache that this load must access memory.

If you successfully found a LOAD to issue, call setActivity() as shown above and return.

Next, look for a LOAD instructions in the LSQ that can be issued. LOADs can have dependencies so should be prioritized over STOREs. STOREs can wait a long time in the LSQ without hurting performance, as long as the LSQ doesn't fill completely. Loop over all entries in the LSQ and do the following:

- Skip past any STOREs and any instruction with an unknown address (a register source operand needed to compute the address does not have a value).
- Starting from the LSQ index one less than the LOAD and looping backwards:
    - Skip past any LOADs
    - If a STORE is encountered with an unknown address, this is a **failure**. Exit the inner loop and skip past the LOAD in the outer loop. We do not implement speculative LOADs in this design.
    - If a STORE is encountered with an address that is both known **and** the same as that of the LOAD, this is a **match**. Exit the inner loop. In the outer loop, check to make sure that the STORE also has a known **value** (what it wants to write to memory).
        - If the value is unknown, this is a failure: Skip past this LOAD instruction.
        - If the value is known, record the indices of the LOAD and matching STORE for later processing. Exit the outer loop.
    - If you finish the inner loop and find that there are no STOREs preceding the LOAD, or all STOREs preceding the load have known addresses do not match the load, then record the index of the LOAD and the fact that there is no matching STORE. Terminate the outer loop.

After executing the above, if an issuable LOAD has been found, the we will send it to the DCache:

- If there **is** a matching store, then:
    - Copy the data value from the matching STORE to the LOAD:
        - output.setResultValue(store_oper0.getValue(), store_oper0.isFloat());
    - Set a property on output latch instructing the DCache that this is a BYPASS LOAD, meaning that it will pass through the DCache **without** accessing memory.
- If there **is not** a matching store, then:
    - Set a property on the output latch, indicating that this is an ACCESS LOAD, which will make the DCache fetch the LOAD's data value from main memory.
- Delete the LOAD from the LSQ, shifting succeeding entries to fill the gap and decrementing the count of used entries.

If you successfully found a LOAD to issue, call setActivity() as shown above and return.

Next, see if there are any STORE instructions that can be completed. Completing a STORE **does not** access main memory, and completing a store **does not** remove it from the LSQ. For a store, "completion" merely indicates to Writeback that the STORE has receives all of its inputs and is therefore eligible to retire. Loop over all LSQ entries and do the following:

- Skip past all LOAD instructions, all instruction not ready (some input operands have not been received), and all instructions marked as completed or retired.
- Set the instruction on the output Latch.
- Set a property on the output latch indicating that this is an ISSUE STORE so that its data value **is not** written to memory. An ISSUE STORE is passed through the DCache simply to mark its ROB entry as completed/valid.

If you successfully found a STORE to issue, call setActivity() as shown above and return.

Finally, check to see if the **first** entry of the LSQ has been marked retired. If this is the case, then perform the following:

- Set the instruction in the output Latch.
- Set a property on the output Latch indicating that this is a COMMIT STORE. A COMMIT STORE has its data written to memory, but **it is not passed through to writeback**.
- Delete the STORE from the LSQ, shifting all remaining entries to fill the gap.

Finally, if you're gotten to this point, call setActivity() as shown above and return.

MemUnit and DCache

Most of this is explained above, but I will repeat some key points.

- The MemUnit is comprised of three stages: LSQ, DCache1, and DCache2.

- DCache1 accepts posted forwarding and computes the address. The address is passed to DCache2 through a property on the output Latch.

There are four different actions DCache2 can take:

- LOAD with BYPASS
  - Data value already retrieved from a matching STORE. Pass through without accessing memory.
- LOAD with ACCESS
  - Data value must be retrieved from memory.
- STORE with ISSUE
  - Store is passed through DCache without modifying memory.
- STORE with COMMIT
  - Store modifies memory but **is not passed on to Writeback.**

MyCpuCore

In initProperties(), no flags are set on the PRF. Instead:

- Set every RAT entry to -1, to indicate that the value is stored in the ARF.
- On every ARF entry, set the USED flag and clear the INVALID flag.

In runProgram():

- Upon entering set the global CPU_RUN_STATE to RUN_STATE_RUNNING.
- Loop as long as CPU_RUN_STATE is not equal to RUN_STATE_HALTED.

There are various other changes to the pipeline that should be obvious from the reference trace.

Writeback

The Writeback stage loops over all of its input latches, accepting completed instructions. For each valid input, do the following:

- Store the completing instruction into its associated ROB entry.
- Call core.incCompleted();
- If the CPU run state is RUN_STATE_FLUSH **and** core.numCompleted()==core.numIssued(), then change the run state to RUN_STATE_RECOVERY.
- If the instruction needs a writeback, set its result value in the PRF.
- If the instruction does not need a writeback, simply mark its PRF entry as valid.
- If the CPU run state is RUN_STATE_RUNNING **and** the instruction has a fault:
  - Mark the PRF entry as having a fault
  - Change the CPU run state to RUN_STATE_FAULT.
- Don’t forget to consume the input.

Note: When setting the CPU_RUN_STATE, **always** use the setClockedProperty method.

Retirement

The Retirement stage has zero inputs and outputs.

Upon entering the compute() method, check the CPU run state. If the state is RUN_STATE_FLUSH, then:

- Reset the ROB head to the same value as the ROB tail.
- Reset all RAT entries to -1.

Create a Set<Integer> to keep track of all ROB entries getting retired on this clock cycle. This is used by the LSQ to know when to retire STOREs.

The main retirement loop will retire as many ROB entries as possible. While there are ROB entries and the head of the ROB is **completed** (PRF entry marked valid):

- If the head instruction is HALT, set the CPU run state to RUN_STATE_HALTING.
- If the head instruction has a fault:
    - Set the CPU run state to RUN_STATE_FLUSH
    - Set the global RECOVERY_PC to the program address of the faulting instruction (ins.getPCAddress()).
    - Set the global RECOVERY_TAKEN to **true** if **ins.getBranchResolution()** indicates that the branch was resolved taken (otherwise set it to **false**).
    - Exit the retirement loop
- Add the head instruction's ROB index (which happens to be the same as the head index itself) to the set of retired entries.
- Mark the corresponding PRF entry as retired. (for diagnostic purposes)
- If the instruction requires a writeback:
    - Get the original architectural register number of the destination operand by calling ins.getOper0().getOrigRegisterNumber()
    - Copy the value from the head of the PRF into this ARF entry.
    - If the corresponding RAT entry points to the head of the ROB, then set that RAT entry to -1.
- Increment the ROB head pointer (wrapping around the end). Also decrement the used-entry count if you have chosen that option.

Upon exiting compute(), call **core.putRetiredSet()** with the set of all instructions retired this cycle.