Project 2:
Register File and Renaming

Version 1.0

ECE 721: Advanced Microarchitecture
Spring 2026, Prof. Rotenberg

Due: Friday, February 20, 2026, 11:59pm
Late projects will only be accepted until: Friday, February 27, 2026, 11:59pm

- READ THIS ENTIRE DOCUMENT (and the "`renamer.h`" specification referenced in Sec. 3).
- Academic integrity:
    - **Outside of legal participation in message board Q&A (see the section on "Reasonable assistance", below), there shall be no collaboration whatsoever:** Each student must work individually on their project. There is zero tolerance for collaboration on the project in any form.
    - **Source code:** Each student must design and write their source code alone. They must do this (design and write their source code) without the assistance of any other person in ECE 721 or not in ECE 721. They must do this (design and write their source code) without searching the web for past semesters' projects and without searching the web for source code with similar goals (*e.g.*, modeling superscalar processors), 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*. Use of ChatGPT or other generative AI tools is prohibited.
    - **Explicit debugging:** With respect to "explicit debugging" as part of the coding process (*e.g.*, using a debugger, inspecting code for bugs, inserting prints in the code, iteratively applying fixes, *etc*.), each student must explicitly debug their code without the assistance of any other person in ECE 721 or not in ECE 721.
    - **Sanctions:** The sanctions for violating the academic integrity policy are (1) a score of 0 on the project and (2) academic integrity probation for a first-time offense or suspension for a subsequent offense (the latter sanctions are administered by the Office of Student Conduct). Note that, when it comes to academic integrity violations, both the giver and receiver of aid are responsible and both are sanctioned. Please see the following RAIV form which has links to various policies and procedures and gives a sense of how academic integrity violations are confronted, documented, and sanctioned: RAIV form.
- Reasonable assistance: If a student has any doubts or questions, or if a student is stumped by a bug, the student is encouraged to seek assistance using both of the following channels.
    - Students may receive assistance from the TA(s) and instructor.
    - Students are encouraged to post their doubts, questions, and obstacles, on the Moodle message board for this project. The instructor and TA(s) will moderate the message board to ensure that reasonable assistance is provided to the student. Other students

are encouraged to participate in the message board Q&A 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.*"

1. Introduction

In this project, you will implement the register file and register renaming mechanism for a modern superscalar microarchitecture. The renamer module is a C++ class. Interface functions and primary data structures are pre-defined since everyone's renamer class must interface to the same pipeline. The implementation of the renamer class is totally up to you, as long as the interface and other key requirements are met.

You will test your renamer class by linking it to the instructor's pipeline simulator. You will be provided with a pre-compiled library that implements the pipeline. A Makefile is provided to link your renamer class to the pipeline, forming a test simulator. Your renamer class must be correct (the simulator must run to completion) and must closely match the performance of the instructor's renamer class.

2. Components

Figure 1 shows the organization of a modern superscalar microarchitecture. You will implement the shaded components in this project.

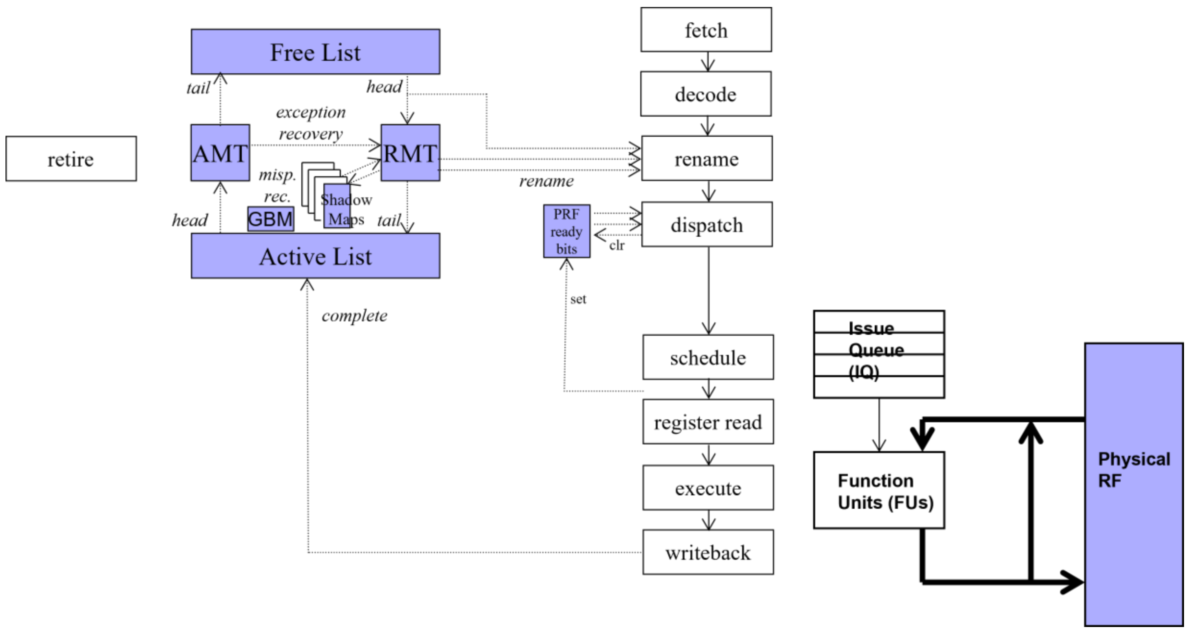

Figure 1. Shaded components are implemented in the C++ renamer class.

3. C++ renamer class specification

Thoroughly read the file "`renamer.h`", which is the formal C++ renamer class specification. It is posted on the Moodle site under the Project 2 topic.

4. Tasks for Project 2

In this project, you will (1) implement the renamer class and (2) compile your class and link it with the provided library to form a full simulator.

4.1. Getting started

Create a dedicated directory for this course under your home directory: ~/ece721/. Under this dedicated course directory, create a sub-directory for this project called proj2/ (~/ece721/proj2/).

Within proj2/, you must create two new files: renamer.h and renamer.cc. These two files represent a self-contained C++ class for the renamer. A blank renamer.h file is posted on the Moodle site (and serves as the renamer class specification as mentioned in Section 3), which you can fill in with data structures of your choosing. It is good coding style to place actual function code in a separate file, renamer.cc, and the Makefile (below) assumes this file exists.

You must download three items (posted on the Moodle site) to proj2/ in order to build your simulator:

1. Makefile: This file allows you to just type "make" and your renamer class will be compiled and linked in with the simulator library. This will create a full processor simulator called "721sim".
2. glue.cc: This is a piece of glue-code that is needed for the simulator to interface with your class. You should not do anything with glue.cc other than download it to proj2/. The Makefile assumes it exists.

3. lib721sim.a: This is a library archive containing pre-compiled object code for the rest of the processor simulator. The only thing you need to do is download it to proj2/. The Makefile assumes it exists.

4.2. What measurements to collect and print out

The simulator will automatically print out information (IPC and a few other things). You do not have to modify the output. Your task is to get the renamer to work correctly.

4.3. Debugging

Debugging tips:

1. The best advice is prevention. Avoid many bugs through careful design, up-front planning, and efficient coding. Fully understand the interface and the management of your data structures. When there is a bug, having a deep understanding of the code will enable you to locate it by re-inspecting the code.
2. You may be tempted to ask me to print out pipeline information. In fact, you can do this yourself. Simply print out a trace of calls to each renamer function.
3. Use software asserts whenever possible. For example, when renaming a destination register, assert the free list is not empty. When freeing a register, assert the free list isn't already full. Check the man page (type: man assert) if you haven't used asserts before.
4. Use the *gdb* debugger (or a graphical debugger) to step through your code. If you don't know how to use *gdb*, now is a good time to learn. At some point, every computer engineer has to use a debugger.
5. It is recommended that you implement Project 2 in three phases:
    - **Phase 1: Get perfect branch prediction working, without implementing *renamer::checkpoint()* and *renamer::resolve()*.** If you are targeting just perfect branch prediction, you can defer implementing *renamer::checkpoint()* and *renamer::resolve()*, since there are no branch mispredictions. IPC *may* differ in some perfect branch prediction runs (those with fewer branch checkpoints) if you don't implement these two functions, since 721sim won't stall the rename stage on limited branch checkpoints, but at least you'll verify running to completion.
    - **Phase 2: Get perfect branch prediction working, with your implementations of *renamer::checkpoint()* and *renamer::resolve()*.** This will test your branch checkpoint management (allocating and freeing branch checkpoints), except for aspects impacted by branch misprediction recovery since there are no branch mispredictions with perfect branch prediction. If you are successful with Phase 2, all perfect branch prediction runs should complete and match IPCs of corresponding validation runs.
    - **Phase 3: Get real branch prediction working.** If you completed Phases 1 and 2, it may be easier to attribute Phase 3 bugs to flawed branch misprediction recovery.
6. **I highly recommend you use straightforward arrays for your various data structures, when applicable.** Sometimes, using sophisticated C++ classes for structures that are more naturally implemented as straightforward arrays leads to overcomplication, many bugs, and

code that is hard to debug. That said, you are free to use whatever style you wish as long as it works.

The Moodle site will list benchmarks to run and the corresponding output of my simulator. First, get your simulator to run to completion. Then, compare your IPC to mine.

4.4. Submitting and self-grading your project

You will submit just your source files – renamer.h, renamer.cc, and optionally any other supporting .h/.cc files that you may have created – to Gradescope for automatic grading. Just as you had done during development, Gradescope will compile and link your renamer with the rest of the simulator, test it on the validation runs, and compute your score. Your final score on this project is whatever score Gradescope shows for your latest submission before the deadline.

4.5. Project scoring

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

N: total number of validation runs.

validation_run_score$_i$:
Score for each validation run:

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

The project score is calculated as follows:

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

4.6. Late policy

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

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