Project 2:
Register File and Renaming

Version 1.0
Due Date: Monday, February 10, 5:00 PM

ECE 721: Advanced Microarchitecture
Spring 2020, Prof. Rotenberg

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

1. Introduction

In this part of the project, you will 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 part of the project.

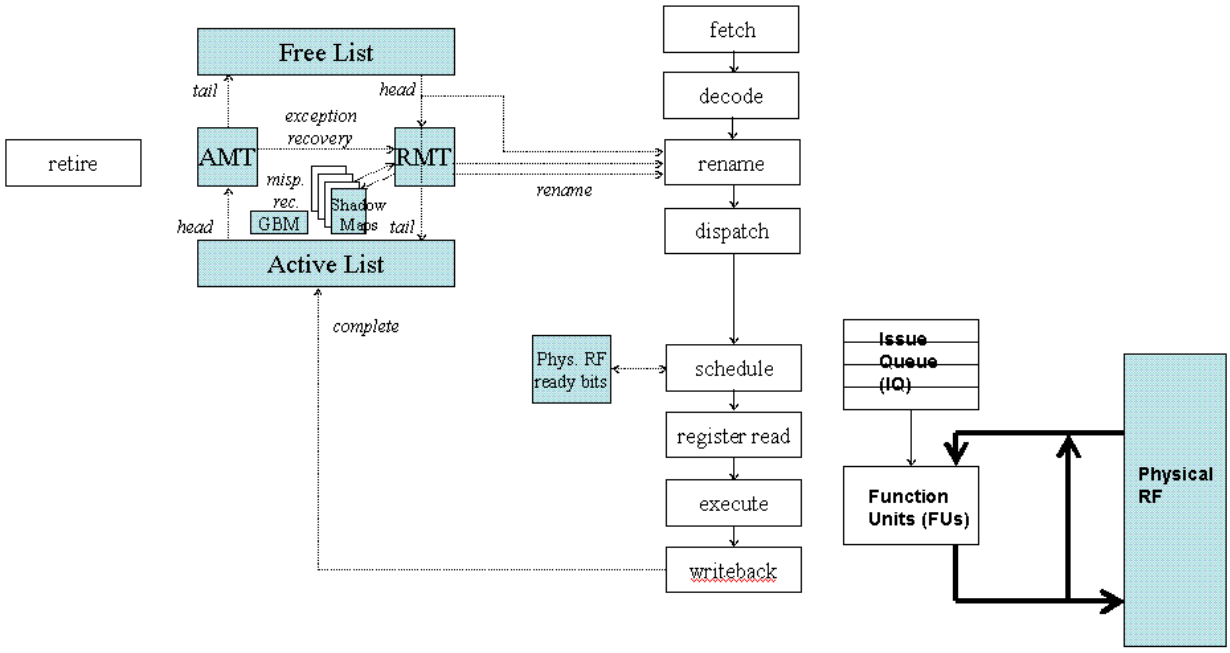

Figure 1. Shaded components are implemented in the C++ renamer class.

3. C++ renamer class specification

```
#include <inttypes.h>

class renamer {
private:
      ////////////////////////////////////////////////////////////////////
      // Put private class variables here.
      ////////////////////////////////////////////////////////////////////

      ////////////////////////////////////////////////////////////////////
      // Structure 1: Rename Map Table
      // Entry contains: physical register mapping
      ////////////////////////////////////////////////////////////////////

      ////////////////////////////////////////////////////////////////////
      // Structure 2: Architectural Map Table
      // Entry contains: physical register mapping
      ////////////////////////////////////////////////////////////////////

      ////////////////////////////////////////////////////////////////////
      // Structure 3: Free List
      //
      // Entry contains: physical register number
      //
      // Notes:
      // * Structure includes head, tail, and possibly other variables
      //   depending on your implementation.
      ////////////////////////////////////////////////////////////////////
```

```
//////////////////////////////////////////////////////////////////
// Structure 4: Active List
//
// Entry contains:
//
// ----- Fields related to destination register.
// 1. destination flag (indicates whether or not the instr. has a
//    destination register)
// 2. logical register number of the instruction's destination
// 3. physical register number of the instruction's destination
// ----- Fields related to completion status.
// 4. completed bit
// ----- Fields for signaling offending instructions.
// 5. exception bit
// 6. load violation bit
//    * Younger load issued before an older conflicting store.
//      This can happen when speculative memory disambiguation
//      is enabled.
// 7. branch misprediction bit
//    * At present, not ever set by the pipeline. It is simply
//      available for deferred-recovery Approaches #1 or #2.
//      Project 2 uses Approach #5, however.
// 8. value misprediction bit
//    * At present, not ever set by the pipeline. It is simply
//      available for deferred-recovery Approaches #1 or #2,
//      if value prediction is added (e.g., research projects).
// ----- Fields indicating special instruction types.
// 9. load flag (indicates whether or not the instr. is a load)
// 10. store flag (indicates whether or not the instr. is a store)
// 11. branch flag (indicates whether or not the instr. is a branch)
// 12. amo flag (whether or not instr. is an atomic memory operation)
// 13. csr flag (whether or not instr. is a system instruction)
// ----- Other fields.
// 14. program counter of the instruction
//
// Notes:
// * Structure includes head, tail, and possibly other variables
//   depending on your implementation.
//////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////
// Structure 5: Physical Register File
// Entry contains: value
//
// Notes:
// * The value must be of the following type: uint64_t
//   (#include <inttypes.h>, already at top of this file)
//////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////
// Structure 6: Physical Register File Ready Bit Array
// Entry contains: ready bit
//////////////////////////////////////////////////////////////////
```

```
//////////////////////////////////////////////////////////////////
// Structure 7: Global Branch Mask (GBM)
//
// The Global Branch Mask (GBM) is a bit vector that keeps track of
// all unresolved branches. A '1' bit corresponds to an unresolved
// branch. The "branch ID" of the unresolved branch is its position
// in the bit vector.
//
// The GBM serves two purposes:
//
// 1. It provides a means for allocating checkpoints to unresolved
//    branches. There are as many checkpoints as there are bits in
//    the GBM. If all bits in the GBM are '1', then there are no
//    free bits, hence, no free checkpoints. On the other hand, if
//    not all bits in the GBM are '1', then any of the '0' bits
//    are free and the corresponding checkpoints are free.
//
// 2. Each in-flight instruction needs to know which unresolved
//    branches it depends on, i.e., which unresolved branches are
//    logically before it in program order. This information
//    makes it possible to squash instructions that are after a
//    branch, in program order, and not instructions before the
//    branch. This functionality will be implemented using
//    branch masks, as was done in the MIPS R10000 processor.
//    An instruction's initial branch mask is the value of the
//    the GBM when the instruction is renamed.
//
// The simulator requires an efficient implementation of bit vectors,
// for quick copying and manipulation of bit vectors. Therefore, you
// must implement the GBM as type "uint64_t".
// (#include <inttypes.h>, already at top of this file)
// The "uint64_t" type contains 64 bits, therefore, the simulator
// cannot support a processor configuration with more than 64
// unresolved branches. The maximum number of unresolved branches
// is configurable by the user of the simulator, and can range from
// 1 to 64.
//////////////////////////////////////////////////////////////////
uint64_t GBM;

//////////////////////////////////////////////////////////////////
// Structure 8: Branch Checkpoints
//
// Each branch checkpoint contains the following:
// 1. Shadow Map Table (checkpointed Rename Map Table)
// 2. checkpointed Free List head index
// 3. checkpointed GBM
//////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////
// Private functions.
// e.g., a generic function to copy state from one map to another.
//////////////////////////////////////////////////////////////////
```

```
public:
      ////////////////////////////////////////
      // Public functions.
      ////////////////////////////////////////

      ///////////////////////////////////////////////////////////////////
      // This is the constructor function.
      // When a renamer object is instantiated, the caller indicates:
      // 1. The number of logical registers (e.g., 32).
      // 2. The number of physical registers (e.g., 128).
      // 3. The maximum number of unresolved branches.
      //    Requirement: 1 <= n_branches <= 64.
      //
      // Tips:
      //
      // Assert the number of physical registers > number logical registers.
      // Assert 1 <= n_branches <= 64.
      // Then, allocate space for the primary data structures.
      // Then, initialize the data structures based on the knowledge
      // that the pipeline is intially empty (no in-flight instructions yet).
      ///////////////////////////////////////////////////////////////////
      renamer(uint64_t n_log_regs,
            uint64_t n_phys_regs,
            uint64_t n_branches);

      ///////////////////////////////////////////////////////////////////
      // This is the destructor, used to clean up memory space and
      // other things when simulation is done.
      // I typically don't use a destructor; you have the option to keep
      // this function empty.
      ///////////////////////////////////////////////////////////////////
      ~renamer();


      ////////////////////////////////////////
      // Functions related to Rename Stage.   //
      ////////////////////////////////////////

      ///////////////////////////////////////////////////////////////////
      // The Rename Stage must stall if there aren't enough free physical
      // registers available for renaming all logical destination registers
      // in the current rename bundle.
      //
      // Inputs:
      // 1. bundle_dst: number of logical destination registers in
      //    current rename bundle
      //
      // Return value:
      // Return "true" (stall) if there aren't enough free physical
      // registers to allocate to all of the logical destination registers
      // in the current rename bundle.
      ///////////////////////////////////////////////////////////////////
      bool stall_reg(uint64_t bundle_dst);
```

```
//////////////////////////////////////////////////////////////////
// The Rename Stage must stall if there aren't enough free
// checkpoints for all branches in the current rename bundle.
//
// Inputs:
// 1. bundle_branch: number of branches in current rename bundle
//
// Return value:
// Return "true" (stall) if there aren't enough free checkpoints
// for all branches in the current rename bundle.
//////////////////////////////////////////////////////////////////
bool stall_branch(uint64_t bundle_branch);

//////////////////////////////////////////////////////////////////
// This function is used to get the branch mask for an instruction.
//////////////////////////////////////////////////////////////////
uint64_t get_branch_mask();

//////////////////////////////////////////////////////////////////
// This function is used to rename a single source register.
//
// Inputs:
// 1. log_reg: the logical register to rename
//
// Return value: physical register name
//////////////////////////////////////////////////////////////////
uint64_t rename_rsrc(uint64_t log_reg);

//////////////////////////////////////////////////////////////////
// This function is used to rename a single destination register.
//
// Inputs:
// 1. log_reg: the logical register to rename
//
// Return value: physical register name
//////////////////////////////////////////////////////////////////
uint64_t rename_rdst(uint64_t log_reg);

//////////////////////////////////////////////////////////////////
// This function creates a new branch checkpoint.
//
// Inputs: none.
//
// Output:
// 1. The function returns the branch's ID. When the branch resolves,
//    its ID is passed back to the renamer via "resolve()" below.
//
// Tips:
//
// Allocating resources for the branch (a GBM bit and a checkpoint):
// * Find a free bit -- i.e., a '0' bit -- in the GBM. Assert that
//   a free bit exists: it is the user's responsibility to avoid
//   a structural hazard by calling stall_branch() in advance.
// * Set the bit to '1' since it is now in use by the new branch.
```

```
// * The position of this bit in the GBM is the branch's ID.
// * Use the branch checkpoint that corresponds to this bit.
//
// The branch checkpoint should contain the following:
// 1. Shadow Map Table (checkpointed Rename Map Table)
// 2. checkpointed Free List head index
// 3. checkpointed GBM
////////////////////////////////////////////////////////////////////
uint64_t checkpoint();

//////////////////////////////////////////
// Functions related to Dispatch Stage. //
//////////////////////////////////////////

////////////////////////////////////////////////////////////////////
// The Dispatch Stage must stall if there are not enough free
// entries in the Active List for all instructions in the current
// dispatch bundle.
//
// Inputs:
// 1. bundle_inst: number of instructions in current dispatch bundle
//
// Return value:
// Return "true" (stall) if the Active List does not have enough
// space for all instructions in the dispatch bundle.
////////////////////////////////////////////////////////////////////
bool stall_dispatch(uint64_t bundle_inst);

////////////////////////////////////////////////////////////////////
// This function dispatches a single instruction into the Active
// List.
//
// Inputs:
// 1. dest_valid: If 'true', the instr. has a destination register,
//    otherwise it does not. If it does not, then the log_reg and
//    phys_reg inputs should be ignored.
// 2. log_reg: Logical register number of the instruction's
//    destination.
// 3. phys_reg: Physical register number of the instruction's
//    destination.
// 4. load: If 'true', the instr. is a load, otherwise it isn't.
// 5. store: If 'true', the instr. is a store, otherwise it isn't.
// 6. branch: If 'true', the instr. is a branch, otherwise it isn't.
// 7. amo: If 'true', this is an atomic memory operation.
// 8. csr: If 'true', this is a system instruction.
// 9. PC: Program counter of the instruction.
//
// Return value:
// Return the instruction's index in the Active List.
//
// Tips:
//
// Before dispatching the instruction into the Active List, assert
// that the Active List isn't full: it is the user's responsibility
```

```
// to avoid a structural hazard by calling stall_dispatch()
// in advance.
/////////////////////////////////////////////////////////////////
uint64_t dispatch_inst(bool dest_valid,
                       uint64_t log_reg,
                       uint64_t phys_reg,
                       bool load,
                       bool store,
                       bool branch,
                       bool amo,
                       bool csr,
                       uint64_t PC);


//////////////////////////////////////////
// Functions related to Schedule Stage. //
//////////////////////////////////////////

/////////////////////////////////////////////////////////////////
// Test the ready bit of the indicated physical register.
// Returns 'true' if ready.
/////////////////////////////////////////////////////////////////
bool is_ready(uint64_t phys_reg);

/////////////////////////////////////////////////////////////////
// Clear the ready bit of the indicated physical register.
/////////////////////////////////////////////////////////////////
void clear_ready(uint64_t phys_reg);

/////////////////////////////////////////////////////////////////
// Set the ready bit of the indicated physical register.
/////////////////////////////////////////////////////////////////
void set_ready(uint64_t phys_reg);


//////////////////////////////////////////
// Functions related to Reg. Read Stage.//
//////////////////////////////////////////

/////////////////////////////////////////////////////////////////
// Return the contents (value) of the indicated physical register.
/////////////////////////////////////////////////////////////////
uint64_t read(uint64_t phys_reg);


//////////////////////////////////////////
// Functions related to Writeback Stage.//
//////////////////////////////////////////

/////////////////////////////////////////////////////////////////
// Write a value into the indicated physical register.
/////////////////////////////////////////////////////////////////
void write(uint64_t phys_reg, uint64_t value);
```

```
//////////////////////////////////////////////////////////////////////
// Set the completed bit of the indicated entry in the Active List.
//////////////////////////////////////////////////////////////////////
void set_complete(uint64_t AL_index);

//////////////////////////////////////////////////////////////////////
// This function is for handling branch resolution.
//
// Inputs:
// 1. AL_index: Index of the branch in the Active List.
// 2. branch_ID: This uniquely identifies the branch and the
//    checkpoint in question.  It was originally provided
//    by the checkpoint function.
// 3. correct: 'true' indicates the branch was correctly
//    predicted, 'false' indicates it was mispredicted
//    and recovery is required.
//
// Outputs: none.
//
// Tips:
//
// While recovery is not needed in the case of a correct branch,
// some actions are still required with respect to the GBM and
// all checkpointed GBMs:
// * Remember to clear the branch's bit in the GBM.
// * Remember to clear the branch's bit in all checkpointed GBMs.
//
// In the case of a misprediction:
// * Restore the GBM from the checkpoint. Also make sure the
//   mispredicted branch's bit is cleared in the restored GBM,
//   since it is now resolved and its bit and checkpoint are freed.
// * You don't have to worry about explicitly freeing the GBM bits
//   and checkpoints of branches that are after the mispredicted
//   branch in program order. The mere act of restoring the GBM
//   from the checkpoint achieves this feat.
// * Restore other state using the branch's checkpoint.
//   In addition to the obvious state ...  *if* you maintain a
//   freelist length variable (you may or may not), you must
//   recompute the freelist length. It depends on your
//   implementation how to recompute the length.
//   (Note: you cannot checkpoint the length like you did with
//   the head, because the tail can change in the meantime;
//   you must recompute the length in this function.)
// * Do NOT set the branch misprediction bit in the active list.
//   (Doing so would cause a second, full squash when the branch
//   reaches the head of the Active List. We don’t want or need
//   that because we immediately recover within this function.)
//////////////////////////////////////////////////////////////////////
void resolve(uint64_t AL_index,
             uint64_t branch_ID,
             bool correct);

////////////////////////////////////////
// Functions related to Retire Stage.   //
```

```
////////////////////////////////////////

//////////////////////////////////////////////////////////////////
// This function allows the caller to examine the instr. at the head
// of the Active List.
//
// Input arguments: none.
//
// Return value:
// * Return "true" if the Active List is NOT empty, i.e., there
//   is an instruction at the head of the Active List.
// * Return "false" if the Active List is empty, i.e., there is
//   no instruction at the head of the Active List.
//
// Output arguments:
// Simply return the following contents of the head entry of
// the Active List.  These are don't-cares if the Active List
// is empty (you may either return the contents of the head
// entry anyway, or not set these at all).
// * completed bit
// * exception bit
// * load violation bit
// * branch misprediction bit
// * value misprediction bit
// * load flag (indicates whether or not the instr. is a load)
// * store flag (indicates whether or not the instr. is a store)
// * branch flag (indicates whether or not the instr. is a branch)
// * amo flag (whether or not instr. is an atomic memory operation)
// * csr flag (whether or not instr. is a system instruction)
// * program counter of the instruction
//////////////////////////////////////////////////////////////////
bool precommit(bool &completed,
       bool &exception, bool &load_viol, bool &br_misp, bool &val_misp,
       bool &load, bool &store, bool &branch, bool &amo, bool &csr,
       uint64_t &PC);

//////////////////////////////////////////////////////////////////
// This function commits the instr. at the head of the Active List.
//
// Tip (optional but helps catch bugs):
// Before committing the head instruction, assert that it is valid to
// do so (use assert() from standard library). Specifically, assert
// that all of the following are true:
// - there is a head instruction (the active list isn't empty)
// - the head instruction is completed
// - the head instruction is not marked as an exception
// - the head instruction is not marked as a load violation
// It is the caller's (pipeline's) duty to ensure that it is valid
// to commit the head instruction BEFORE calling this function
// (by examining the flags returned by "precommit()" above).
// This is why you should assert() that it is valid to commit the
// head instruction and otherwise cause the simulator to exit.
//////////////////////////////////////////////////////////////////
void commit();
```

```
    ////////////////////////////////////////////////////////////////////
    // Squash the renamer class.
    //
    // Squash all instructions in the Active List and think about which
    // sructures in your renamer class need to be restored, and how.
    //
    // After this function is called, the renamer should be rolled-back
    // to the committed state of the machine and all renamer state
    // should be consistent with an empty pipeline.
    ////////////////////////////////////////////////////////////////////
    void squash();

    ////////////////////////////////////////
    // Functions not tied to specific stage.//
    ////////////////////////////////////////

    ////////////////////////////////////////////////////////////////////
    // Functions for individually setting the exception bit,
    // load violation bit, branch misprediction bit, and
    // value misprediction bit, of the indicated entry in the Active List.
    ////////////////////////////////////////////////////////////////////
    void set_exception(uint64_t AL_index);
    void set_load_violation(uint64_t AL_index);
    void set_branch_misprediction(uint64_t AL_index);
    void set_value_misprediction(uint64_t AL_index);

    ////////////////////////////////////////////////////////////////////
    // Query the exception bit of the indicated entry in the Active List.
    ////////////////////////////////////////////////////////////////////
    bool get_exception(uint64_t AL_index);
};
```

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 (which looks like Section 3) is posted on the web page, 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 four items (posted on the project web page) 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. libriscv-base.a: Needed by lib721sim.a. 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. The most complex part of the renamer is checkpoint management. To confirm that a problem stems from this part, comment out your checkpoint management code and run the simulator with perfect branch prediction.
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 project web page 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. What to hand in

Hand in your renamer class (directions are posted on the project web page). We will compile and link it to my simulator to verify it. We will also inspect the code and use automated tools to detect copying/cheating.

4.5. Grading

- 0% — You don't hand in anything.
- 25% — You submit a legitimate attempt at an implementation (we will inspect the code), but simulator does not compile or run.
- 50% — Works with perfect branch prediction (see Section 4.3, item #5). Simulator runs for < 1,000 instructions with real branch prediction.
- 60% — Works with perfect branch prediction (see Section 4.3, item #5). Simulator runs for > 1,000 instructions but < 10,000 instructions with real branch prediction.
- 70% — Works with perfect branch prediction (see Section 4.3, item #5). Simulator runs for > 10,000 instructions but < 100,000 instructions with real branch prediction.
- 80% — Works with perfect branch prediction (see Section 4.3, item #5). Simulator runs for > 100,000 instructions with real branch prediction, but does not complete.
- 90% — Works with perfect branch prediction (see Section 4.3, item #5). Simulator runs to completion with real branch prediction, but performance differs from instructor's version by > 1%.
- 100% — Simulator runs to completion with both perfect and real branch prediction, and performance matches within 1%.