CS {4/6}290 & ECE {4/6}100 - Spring 2024
Project 2: Tomasulo & Branch Predictor Simulator

Dr. Thomas Conte and Pulkit Gupta
Due: April 2nd @ 11:55 PM
Version: 1.0

Changelog

1. Version 1.0.0 (2024-03-14): Initial release

1 Rules

- **This is an individual assignment. ABSOLUTELY NO COLLABORATION IS PERMITTED.** All cases of honor code violations will be reported to the Dean of Students. See Appendix A for more details.
- The due date at the top of the assignment is final. Late assignments will not be accepted for full credit. Please see the syllabus for the late policy for this course.
- Please use office hours for getting your questions answered. If you are unable to make it to office hours, please email the Head TA/Instructors.
- This is a tough assignment that requires a good understanding of concepts before you can start writing code. **Make sure to start early.**
- Read the entire document before starting. Critical pieces of information and hints might be provided along the way.
- Unfortunately, experience has shown that there is a high chance that errors in the project description will be found and corrected after its release. **It is your responsibility to check for updates on Canvas, and download updates if any.**
- Make sure that all your code is written according to **C11 or C++20** standards, using only the standard libraries.

2 Introduction

This project is split into two parts; the Branch Predictors and the Out-of-Order Processor. Please refer to Section 3 for information on implementing the Branch Predictors, and refer to Section 4 for information on implementing the Out-of-Order Processor. In this section, we will discuss simulator parameters and trace files, which are shared by both parts of the project.

2.1 Simulator Parameters

The following command line parameters can be passed to the simulator. Details on these parameters can be found in the corresponding sections listed in Section 2:

- `-x`: When enabled, only run the Branch Predictors.
- `-i`: The path to the input trace file
- `-h`: Print the usage information.

2.1.1 Branch Predictor Specific Parameters

- `-b`: Select the Branch Predictor to use.
  - 0: Always Taken (**provided**)
  - 1: Two-bits Smith Counter (**provided**)
  - 2: Yeh-Patt (**for students in ECE 6100 & CS 6290**)
  - 3: GShare (**for everyone**)
- `-p`: Log_2 of the number of entries in the Pattern Table for Yeh-Patt, or the number of 2-bit counters in GShare.
  - **Restriction:** $P \geq 9$, small pattern tables are not very performant.
- `-H`: Log_2 of the number of entries in the History Table for Yeh-Patt.
  - **Restriction:** $H \geq 9$, small history tables are not very performant.

2.1.2 Out-of-Order Processor Specific Parameters

- `-a`: The number of Addition (ALU) function units in the Execute stage.
- `-m`: The number of Multiply (MUL) function units in the Execute stage.
- `-l`: The number of Load/Store (LSU) function units in the Execute stage.
- `-s`: The number of reservation stations per function unit. The scheduling queue is unified, which means that $s \times (a + m + l)$ reservation stations are shared by all function units.
- `-f`: The "Fetch Width", which is the number of instructions fetched/added to the Dispatch queue per cycle.
- `-d`: This flag has no argument; if it is passed, cache misses and mispredictions are disabled by the driver.

Note that the upper bounds for some parameters are not defined. You will need to constraint these in the experiments according to Section 8.

2.2 Trace Format

Each line in a trace represents the following:
`<Address> <Opcode> <Dest Reg #> <Src1 Reg #> <Src2 Reg #> <LD/ST Addr> <Branch Taken> <Branch Target> <I-Cache Miss> <D-Cache Miss> <Dynamic Instruction Count>`

That is, the instructions are effectively already decoded for you. Each register number is in the range $[0, 31]$, except when a register is -1, which indicates there is no destination register or that source register is not needed. The Branch Misprediction, I-Cache Miss, and D-Cache Miss flags are set to 1 (or otherwise appropriately) when the corresponding events occur. The opcodes map to the following operations and Function Units:

Opcode	Operation	Function Unit
2	Add	ALU unit
3	Multiply, Divide, or Float	MUL unit
4	Load	LSU unit
5	Store	LSU unit
6	Branch	ALU unit

3 Branch Predictor Specifications

Stalls due to control flow hazards and the presence of instructions such as function calls, direct and indirect jump, returns, etc. that modify the control flow of a program are detrimental to a pipelined processor's performance. To mitigate these stalls, various branch prediction techniques are used. In fact, branch prediction is so vital that it continues to be an ongoing area of research and many new ideas are proposed even today. To better understand branch prediction and various existing techniques, in this project you will implement various branch predictors: a two level Local History branch predictor (Yeh-Patt), and a GShare branch predictor.

Recall from lectures that branch prediction has 3 W's: Whether to branch, Where to branch (if the branch is taken) and Which instruction is a branch. In this project we will concern ourselves with only the first W (i.e. Whether a branch is taken or not).

We have provided you with a framework that reads in branch traces generated from the execution of various benchmarks (see Section 6) and drives your predictors. The simulator framework supports configurable predictor parameters. Your task will be to write functions called by the driver that initialize the predictor, perform a branch direction prediction, update the predictor state, and finally update the statistics for each branch in the trace.

3.1 Basic Branch Predictor Architecture

Your simulator should model a GShare predictor (for everyone) and a Yeh-Patt predictor (for students in ECE 6100 & CS 6290). Figures 1 & 2 show the architecture diagrams of the GShare and Yeh-Patt branch predictors, respectively. The driver for the simulation chooses the predictor based on the command line inputs, and calls the appropriate functions for prediction and update.

3.2 The GShare Predictor (everyone)

GShare is a cheap branch predictor discussed in class which hashes the PC and GHR to index into a table of Smith counters [1, 2].

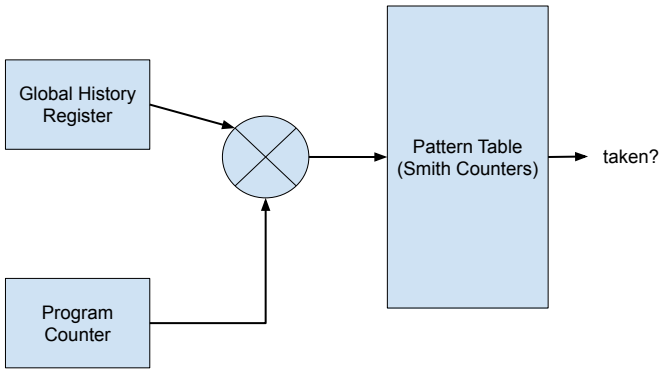

Figure 1: The architecture of the GShare branch predictor

- The Global History Register is a P bits wide shift register. The most recent branch is stored in the *least-significant-bit* of the GHR. A value of 1 denotes a taken branch and a value of 0

denotes a not taken branch. The initial value of the GHR is 0.

  - When you update the GHR, shift the GHR left by 1 and set the lowest bit based on the branch outcome.

- The Table contains 2^P smith counters. The Predictor hashes the GHR with the branch PC to index into the table.
- The simulator will use the hash function: Hash(GHR, PC) = GHR $\bigoplus$ PC$[2 + P - 1 : 2]$, where $\bigoplus$ is the XOR operator. For example, when $PC = 0x000C and P = 2$, $PC[2 + P - 1 : 2] = PC[3 : 2] = 0b11$.
- Each Smith Counter in the Table is 2-bits wide and initialized to $0b01$, the Weakly-Not-Taken state.

3.3 The Yeh-Patt Predictor (ECE 6100 & CS 6290)

A two level adaptive branch predictor [2].

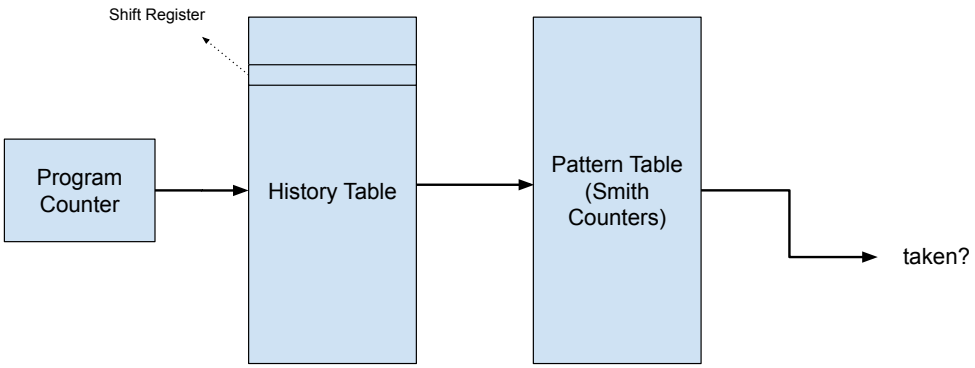

Figure 2: The architecture of the Yeh-Patt branch Predictor

- The History Table (HT) contains 2^H, P-bit wide shift registers. The most recent branch is stored in the *least-significant-bit* of these history registers (based on the branch pc). A value of 1 denotes a taken branch and a value of 0 denotes a not taken branch.
  - When you update a history table entry, shift it left by 1 and set the lowest bit based on the branch outcome.
- The predictor uses bits $[2 + H - 1 : 2]$ of the branch address (PC) to index into the History Table. All entries of the history table are set to 0 at the start of the simulation.
- The Pattern Table (PT) contains 2^P smith counters. The value read from the History Table is used to index into the pattern table.
- Each Smith Counter in the PT is 2-bits wide and initialized to $0b01$, the Weakly-Not-Taken state.
- This makes the total predictor size for the Yeh-Patt Predictor $2^H * P + 2 * 2^P$ bits.

3.4 Simulator Operation

The simulator operates in a trace driven manner and follows the below steps:

- The appropriate branch predictor initialization function is called, where you setup the predictor data structures, etc.
- Branch instructions are read from the trace one at a time and the following operations ensue for each line of the trace:
  - The branch address is used to index into the predictor, and a direction prediction is made. The branch interference statistics are also updated at this time. The prediction is returned to the driver (true - TAKEN, false - NOT-TAKEN). Note that the steps to index into the predictor table(s) may be different for each predictor
  - The driver calls the update prediction stats function. Here the actual behavior of the branch is compared against the prediction. A branch is considered correctly predicted if the direction (TAKEN/NOT-TAKEN) matches the actual behavior of the branch. Stats for tracking the accuracy of the branch predictor are updated here.
  - The branch predictor is updated with the actual behavior of the branch.
- A function to cleanup your predictor data structures (i.e. The destructor for the predictor) and a function to perform stat calculations is called by the driver.

Please find more details of the provided framework for the branch predictor in Section 6.

4 Out-of-Order CPU Specifications

In this section, you will implement a simulator for a CPU using tagged Tomasulo. Your simulator will take in a handful of parameters that are used to define the processor configuration; see Section 2.1.2 for more information. The provided code will call your per-stage simulator functions in reverse order to emulate pipelined behavior. For details on the provided framework, see Section 6.2. Please read the following subsections carefully for information on simulator functionality.

4.1 Basic Out of Order Architecture

Your simulator will implement the stages of a fire out-of-order complete out-of-order (FOCO) processor using Tomasulo's Algorithm with tags. In order to enable correct behavior with memory operations in parallel, the CPU uses a memory disambiguation algorithm.

Figure 3 is a diagram of the general architecture of the model you will simulate. All edges passing through a dotted line will write to their relevant buffers/latches on the clock edge.

4.2 Fetch and The Driver

The driver exposes the instruction cache (I-cache) to you via the function `procsim_driver_read_inst()`. The driver will return 1 instruction per call to `procsim_driver_read_inst()`. Every cycle, you read `-f` instructions from the driver and append them to the dispatch queue (if there is room).

In the event of a mispredicted/faulting instruction, the driver will first return the instruction, but subsequent calls will return `NULL` (a NOP) until you set `retired_mispredict_out` to `true` in `stage_state_update()`. When you do this, `procsim_do_cycle` will update the appropriate statistic. This models the time it takes for the misprediction/fault to be corrected. The instructions from the wrongly predicted path are not provided by the trace, so we fetch NOPs instead.

In the event of an I-Cache miss, the driver will handle this for you in terms of emulating correct behavior. The driver will return NOP instructions until it resolves the I-Cache miss, at which point it will resume returning valid instructions. In the event of an I-Cache miss, NOP instructions should be added to the dispatch queue as if they were a normal instruction. You can get the cause of the NOP using the value of `driver_read_status_t` set by `procsim_driver_read_inst`.

4.3 Dispatch

In Dispatch, we will attempt to fill as many reservation stations in the scheduling queue as possible with instructions from the head of the dispatch queue. You should also set values in the memory alias table for certain instructions (see Section 4.3.1). Remember that instructions are dispatched in program order.

The Dispatch stage can continue to dispatch instructions (remove instructions from the dispatch queue) until one of the following condition is met:

- If there is nothing in the dispatch queue, you must stall.
- If the scheduling queue is full, you must stall.
- The Dispatch stage will allocate room in the ROB for the instruction as it places it into a reservation station, so if there is not sufficient room in the ROB, then Dispatch should stall.

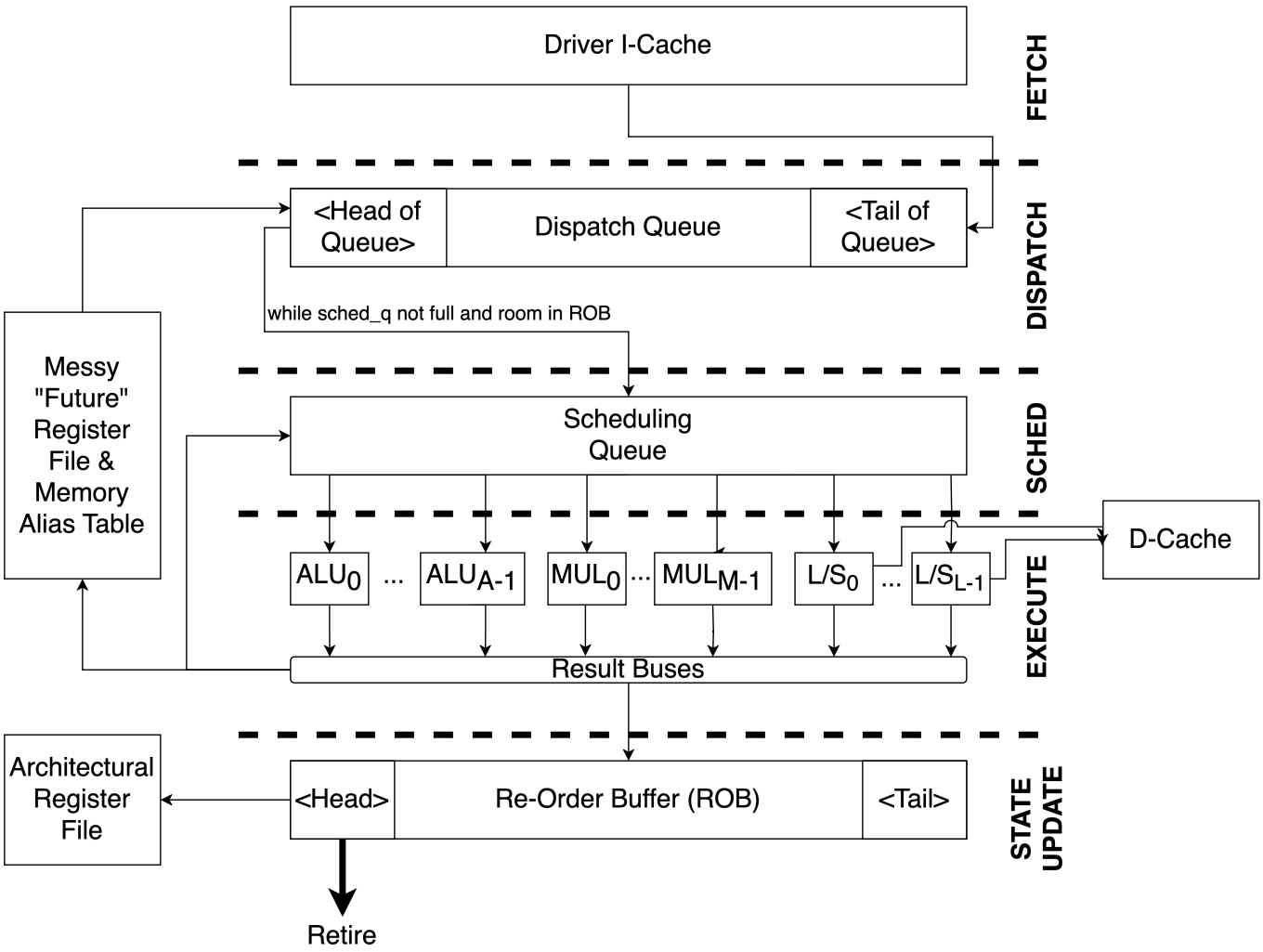

Figure 3: Overview of the Out of Order Architecture. An arrow that crosses a dotted line indicates that the value is latched at the end of a cycle. An arrow that does not cross a dotted line indicates that the associated action occurs within the clock cycle.

- If you already dispatched `procsim_conf_t.dispatch_width` NOPs in this cycle, you must stall.
- If you already dispatched `procsim_conf_t.dispatch_width` instructions that are not NOPs in this cycle, you must stall.

For source registers of an instruction, dispatch will use the tags and ready bits from the messy register file to set up the reservation station in the scheduling queue. For the destination register, dispatch allocates a new tag for it even if the instruction actually doesn't have destination. If it does have a destination register, you clear the ready bit of the destination register in the messy register file and assign this new tag to that register, otherwise you don't. Dispatch also allocates a second new tag for the "memory destination register" of each instruction (see Section 4.3.1 for details).

Note, you may find it unnecessary to generate two new tags for each instruction. The reason we're telling you to do this is so that your debug out matches ours (if you opt to implement the debug

out). To match the debug out, you should also (1) represent the tag using an `uint64_t` (big enough for this project); (2) initialize the global tag counter to 96 (after you initialize the register file and the memory alias table) and increase the counter by 1 after generating a new tag; (3) generate the new tag for the "memory destination register" before generating for the destination register.

Note that not every instruction will have all of `src1, src2, dest` specified. A register value of -1 indicates that register is not specified (not needed). You will need to account for this in Dispatch and Schedule.

When you initialize the dispatch queue, its length should be equal to `f` $\times$ 32, where `f` is fetch width and 32 is the number of registers.

4.3.1 Memory Disambiguation

In Tomasulo's algorithm, register numbers cannot capture data dependence in the memory. This causes program correctness issues. For example, consider the following sequence of memory instructions:

```
1: SW r5, 0(r6)  # (no dest, src1=r5, src2=r6) mem[r6 + 0] = r5
2: LW r7, 0(r8)  # (dest=r7, src1=r8, no src2) r7 = mem[r8 + 0]
```

If r6 and r8 hold the same value, there is a read after write data dependency at `mem[r6+0]`. Because instructions 1 and 2 do not share any registers, the original Tomasulo's Algorithm cannot capture the potential data dependency for this specific case. You will need to implement the following memory disambiguation algorithm to ensure that we do not violate program correctness.

Provided memory traces contain `load_store_addr` for all load/store instructions. The memory disambiguation algorithm adds extra "memory source registers (ms)" and "memory destination registers (md)" to load/store instructions. Since both load and store will only operate on one address at a time, ms and md are always the same. The register number is determined by the hash of the `load_store_addr`, which is `load_store_addr[11:6]`.

For example, assume both load/store instructions access memory location 0x01f0 in the following code sequence:

```
1: SW:   (no dest) | src1=r5 , src2=r6   # SW r5, 0(r6), addr=0x01f0
2: ADD:  dest=r9   | src1=r10, src2=r11  # ADD r9, r10, r11
3: LW:   dest=r7   | src1=r8 , (no src2) # LW r7, 0(r8), addr=0x01f0
```

The address `0x01f0` hashes to 7. That means we need to add "memory register 7". Now the code becomes the following:

```
1: SW:   md=m7  , (no dest) | ms=m7  , src1=r5 , src2=r6   # SW r5, 0(r6), addr=0x01f0
2: ADD:  (no md), dest=r9   | (no ms), src1=r10, src2=r11  # ADD r9, r10, r11
3: LW:   md=m7  , dest=r7   | ms=m7  , src1=r8 , (no src2) # LW r7, 0(r8), addr=0x01f0
```

To implement this memory disambiguation algorithm, we need a "memory alias table" that stores 64 "memory registers". The data structure should be same as the normal register file, where each entry contains a tag and a ready bit. We also need extra fields in each entry of the scheduling queue: `md_tag`, `ms_tag`, and `ms_ready`. The result buses should also update corresponding entries in the register file and the scheduling queue.

By hashing the load/store addresses to "memory registers", we can treat these registers as if they were real registers and run Tomasulo's algorithm. For example, when scheduling load/store instruc-

tion, you do something like (lets say `rs` is a scheduling queue entry and `MAT` is the memory alias table):

```
rs.ms_tag = MAT[inst.ms].tag
rs.ms_ready = MAT[inst.ms].ready
rs.md_tag = unique_tag()
```

Note that is just part of the code and you still need to handle the regular registers. Also notice that even though a load instructions just reads memory, we consider it to have a md and assign a new tag to it. This way when a load instruction is followed by a store instruction that writes the same address, the store instruction will have a dependency on the load instruction, thus preserving the load/store order.

4.4 Schedule

The schedule stage searches the scheduling queue for instructions that are ready to fire (have all of their operands ready) and fires them. Make sure to initialize the scheduling queue to the proper size ($= s \times (a + m + l)$) based on the processor parameters.

In order to ensure determinism and match the solution, if multiple instructions can be fired at the same time, they are fired in program order (but still within the same cycle if possible). This means that if 2 instructions are both ready for a single function unit, the instruction that comes first in program order would fire first. Use `inst_t.dyn_instruction_count` as a mechanism to determine program order. It will also be helpful to ensure that dispatch only adds instructions to the "end" of the scheduling queue.

4.5 Execute

In the execute stage you will move every instruction through its associated function unit and onto a result bus when it completes. Then you will use the result buses to update processor state.

4.5.1 ALU

This is a single-stage unit.

4.5.2 MUL

This is a 3-stage unit that is pipelined, meaning that up to 3 instructions may be worked on at any given time in a single MUL unit.

4.6 LSU

This is a single-stage unit. We assume the Load/Store units are not pipelined.

The traces have been annotated with I-Cache misses and D-Cache misses.

In a real processor, the Load/Store units would access the data cache (dcache) in-order to access or manipulate data. However, our traces are pre-labeled with I-Cache and D-Cache misses (see Section 2.2), meaning that the cache accesses have already been emulated for you.

You will need to stall the LSU units depending on the trace labels and the instruction type. Because we do not model a store buffer or load/store queue, stores will never stall, and instead only take a

single cycle. However, loads will take a variable number of cycles, depending on the cache access result `inst_t.dcache_hit`:

- `CACHE_LATENCY_L1_HIT`: 2 cycles (`L1_HIT_CYCLES`)
- `CACHE_LATENCY_L2_HIT`: 10 cycles (`L2_HIT_CYCLES`)
- `CACHE_LATENCY_L2_MISS`: 100 cycles (`L2_MISS_CYCLES`)

Noted that `L2_HIT_CYCLES` is the total number of cycles that includes the time to miss in L1 and hit in L2. Similarly, `L2_MISS_CYCLES` includes the time to miss in L1 and L2 and get data from the memory.

You don't need to change the stats of I-Cache misses and D-Cache misses because the driver tracks it for you. These values are defined in `procsim.hpp`.

4.6.1 Result Buses

As an instruction finishes executing in its function unit, it will pass its values to the result buses. These values will then be used to update the messy "future" register file and update all of the scheduling queue entries. Note that due to Memory disambiguation, some instructions will have two destinations, so you'll also need to properly broadcast the second destination to update the memory alias table and the scheduling queue.

Completed instructions are removed from the scheduling queue. These busses will also place the instruction in the ROB in program (tag) order. All of this happens within the same cycle!

Note that there are `A + M + L` result busses, thus instructions never stall waiting for a free result bus.

4.7 State Update

State Update is where we will update the architectural state of the processor. Every cycle, you will, in program order, retire a number of instructions up to "retire width" from the ROB. The retire width is equal to the fetch width (defined by `-f`).

In a real processor, you would also commit their destination register values (if present) to the Architectural Register File. However, since we aren't modeling data in our pipeline, you don't actually need to do this!

You will need to call the provided `procsim_driver_update_predictor` function to update your branch predictor when you retire a branch instruction. Also, if you retire an instruction with the `inst_t.mispredict` field set to true, set `retired_mispredict_out` to true and stop retiring any more instructions.

When you create the ROB, make sure its size (`num_rob_entries`) is equal to the fetch width (`-f`) times the number of registers (which is 32 in our scenario).

4.8 Initial Conditions

The simulated pipeline should initialize to the following conditions:

- Registers in the register file all marked as ready with incrementing tags. R0 has tag 0; R1 has tag 1, ..., R31 has tag 31.

- MAT also initialized to ready with incrementing tags. M0 has tag 32; M1 has tag 33, ..., and M63 has tag 95.
- All function units are empty/ready.
- All queues and the ROB are empty.

4.9 Queue sizes summary

- Size of the dispatch queue is equal to the size of ROB.
- Size of the scheduling queue is given by `s` $\times$ (`a` + `m` + `l`) where `s, a, m, l` are parameters.
- Size of the ROB is given by `f` $\times$ 32, where `f` is fetch width and 32 is the number of registers.

5 Statistics

You will need to keep track of the following statistics. They are split up into statistics for the Branch Predictors and Out-of-Order Processor.

5.1 Branch Predictor Statistics

More details into these statistics can be found in `branchsim_stats_t` struct in `branchsim.hpp`:

- `total_instructions`: Total number of instructions. We will trace this for you in the driver.
- `num_branch_instructions`: Total number of branch instructions.
- `num_branches_correctly_predicted`: Total number of branches that were predicted correctly.
- `num_branches_mispredicted`: Total number of branches that were mispredicted.
- `prediction_accuracy`: The accuracy of the Branch Predictor.

5.2 Out-of-Order Processor Statistics

More details into these statistics can be found in `procsim_stats_t` struct in `procsim.hpp`:

- `cycles`: The number of cycles that have passed since the processor was started (the provided code increments this statistic for you).
- `instructions_fetched`: The number of instructions fetched by the Fetch stage, regardless of whether or not they retire. Do not count NOPs.
- `instructions_retired`: The number of instructions that exit the ROB.
- `no_fire_cycles`: Cycles where nothing in the scheduling queue could be fired.
- `rob_no_dispatch_cycles`: Cycles in which Dispatch stops putting instructions into the scheduling queue only because of the constraint on the maximum number of ROB entries.
- `dispq_max_usage`: The maximum number of instructions in the dispatch queue during execution. You should update this at the end of `procsim_do_cycle()`.
- `schedq_max_usage`: The maximum number of instructions in the scheduling queue during execution. You should update this at the end of `procsim_do_cycle()`.

- `rob_max_usage`: The maximum number of instructions in the ROB during execution. You should update this at the end of `procsim_do_cycle()`.
- `dispq_avg_size`: The average number of instructions in the dispatch queue during execution. You should update this at the end of `procsim_do_cycle()`.
- `schedq_avg_size`: The average number of instructions in the scheduling queue during execution. You should update this at the end of `procsim_do_cycle()`.
- `rob_avg_size`: The average number of instructions in the ROB during execution. You should update this at the end of `procsim_do_cycle()`.
- `ipc`: The average number of instruction retired per cycle.
- `i_cache_misses`: **You do not need to modify this.** The number I-Cache misses. The simulator driver tracks this for you.
- `d_cache_misses`: **You do not need to modify this.** The number D-Cache misses. The simulator driver tracks this for you.
- `num_branch_instructions`: **You do not need to modify this.** The number of branch instructions. The simulator driver tracks this for you.
- `branch_mispredictions`: **You do not need to modify this, but you need to set `(*retired_mispredict_out)` correctly within the `stage_state_update` function to get this number correct.** The number of branch mispredictions the processor encounters. The provided `procsim_do_cycle()` function tracks this parameter for you.
- `instructions_in_trace`: **You do not need to modify this.** The number of instructions in the trace. The simulator driver tracks this for you.

6 Implementation Details

You have been provided with the following files:

- `branchsim.cpp` - Your Branch Predictor implementations will go here.
- `branchsim.hpp` - **You do not need to modify this file.** Header file containing useful definitions and function declarations for the branch predictors.
- `Counter.cpp` - **You do not need to modify this file.** Implementation of the Counter struct functions.
- `Counter.hpp` - **You do not need to modify this file.** Header file containing Counter struct and function definitions.
- `procsim.cpp` - Your processor implementation will go here.
- `procsim.hpp` - **You do not need to modify this file.** Header file containing definitions shared between your simulator and the driver.
- `proj2_driver.hpp` - **You do not need to modify this file.** Provided code that reads a trace, maintains a pointer to the next instruction in the trace, and allows your simulator code to read it via `procsim_driver_read_inst()`. The simulator invokes your `procsim_do_cycle()` function repeatedly until all trace instructions have been retired.
- `run.sh`: A shell script that invokes your simulator. You only need to change this if you have made the highly discouraged choice to write your simulator without the provided framework.
- `validate_grad.sh` (**for students in ECE 6100 & CS 6290**): **You do not need to modify this file.** A shell script which runs your simulator and compares its outputs with the reference outputs.
- `validate_undergrad.sh` (**for students in ECE 4100 & CS 4290**): **You do not need to modify this file.** A shell script which runs your simulator and compares its outputs with the reference outputs.
- `Makefile`: A Makefile that contains the logic needed to compile your code. It has some useful features:
  - `make validate_grad` will compile your code and run `validate_grad.sh`.
  - `make validate_undergrad` will compile your code and run `validate_undergrad.sh`.
  - `make submit` will generate a submission tarball for Gradescope.
  - `make clean` will clean out all compiled files.
  - `make FAST=1` will compile with `-O2` (You should run `make clean` first).
  - `make DEBUG=1` will compile with the preprocessor definition `DEBUG` defined. (You should run `make clean` first). This will cause code gated with `#ifdef DEBUG ... #endif` to compile, which you may find useful for generating your own debug traces.
  - `make PROFILE=1` will compile your code for use with `gprof`. (You should run `make clean` first). If your simulator is working correctly but runs slowly, this may help diagnose where the bottleneck is. After you compile with this flag and run your simulator as normal, you can run `gprof procsim` to see profiling results.

  - `make SANITIZE=1` will compile your code for use with `Address-Sanitizer`. (You should run `make clean` first). If your simulator is seg-faulting but valgrind cannot find the error, address sanitizer is a stricter framework that is capable of finding memory bugs in global/stack space and the heap.

- `traces/`: A directory containing execution traces from real programs run on a RISC-V simulator; their format is detailed above in Section 2.2. The driver will read these for you:
  - `mcf505`: A trace from a program for vehicle scheduling in mass public transportation. It consist mostly of integer arithmetic.
  - `deepsjeng531`: A trace for a program based on the 2008 Computer Speed-Chess Champion. It estimates the best move using multiple heuristics; including alpha-beta tree search, forward pruning, and move ordering.
  - `leela541`: A trace for a program for a Go playing engine. It uses Monte Carlo position estimation and selective tree search.
  - `xz557`: A trace from a program based on XZ Utils 5.0.5; a data compression and decompression utility.
  - `nab544`: A trace from a program based on Nucleic Acid Builder (NAB), which is a molecular modeling application that performs the types of floating point intensive calculations that occur commonly in life science computation. The calculations range from relatively unstructured "molecular dynamics" to relatively structured linear algebra.
- `ref_outs/`: A directory containing the output of the TA simulator(s) for some select configurations (`make validate` will print their configuration flags).

6.1 Provided Framework for Branch Predictor

We have provided you with a comprehensive framework where you will add data structure declarations and write the following functions for each branch predictor (We have also provided implementations for 2 predictors for you for your reference):

- `void <predictor_name>_init_predictor(branchsim_conf *sim_conf)`

  Initialize class variables, allocate memory, etc for the predictor in this function

- `bool <predictor_name>_predict(branch *branch)`

  Predict a branch instruction and return a `bool` for the predicted direction {true (TAKEN) or false (NOT-TAKEN)}. The parameter `branch` is a structure defined as:

```
typedef struct branch_t {
    uint64_t ip;            // Branch address (PC)
    uint64_t inst_num;      // Instruction count of the branch
    bool is_taken;          // Actual branch outcome
} branch;
```

- `void <predictor_name>_update_predictor(branch *branch)`

  Function to update the predictor internals such as the history register and Smith counters, etc. based on the actual behavior of the branch

- `void <predictor_name>_cleanup_predictor()`

Destructor for the branch predictor. De-allocate any heap allocated memory or other data structures here.

Apart from the per predictor functions, you will need to implement some general functions for book-keeping and final statistics calculations:

- ```
  void branchsim_update_stats(bool prediction, branch_t *br,
      branchsim_stats_t *sim_stats);
  ```

  Function to update statistics based on the correctness of the prediction (you can use the `is_taken` field of the `branch` to check if the branch was actually taken or not).

- ```
  void branchsim_finish_stats(branchsim_stats_t *sim_stats);
  ```

  Function to perform final calculations such as misprediction rate, etc

6.2 Provided Framework for OOO Processor

For the Out-of-Order processor, we have provided multiple empty functions representing different pipeline stages. Please refer to Section 4 for information on pipeline stage functionality. There are also empty function definitions for initializing your pipeline variables and calculating statistics. All of these functions can be found in `procsim.cpp`, with detailed comments explaining what each function should do. Fill in any function that has a TODO comment.

However, we do provide the implementation for `procsim_do_cycle()`, which invokes the pipeline stage functions in reverse order to prevent you from needing to manage pipeline buffers by hand.

You will need to keep track of certain statistics for your pipeline. The provided header file `procsim.hpp` contains the `procsim_stats_t` struct, which you will need to fill with the proper values. Please see Section 5 for details of what each statistic should track/represent.

6.3 Docker Image

We have provided an Ubuntu 22.04 LTS Docker image for verifying that your code compiles and runs in the environment we expect — it is also useful if you do not have a Linux machine handy. To use it, install Docker (`https://docs.docker.com/get-docker/`), extract the provided project tarball, and run the `6290docker*` script in the project tarball corresponding to your OS. You do not need to use the graphical application that comes with Docker. Running the previous script should open an 22.04 `bash` shell where the project folder is mounted as a volume at the current directory in the container, allowing you to run whatever commands you want, such as `make`, `gdb`, `valgrind`, `./proj2sim`, etc.

Note: Using this Docker image is not required. If you have a Ubuntu 22.04 system or environment available, you can verify that your code compiles on that system. We will also set up a Gradescope autograder that automatically verifies if we can successfully compile your submission, so if your submission passes the Gradescope compilation step, then there is no need to verify with Docker or an Ubuntu 22.04 system.

7 Validation Requirements

You must run your simulator and debug it until the statistics from your solution **perfectly (100 percent)** match the statistics in the reference outputs for all test configurations. This requirement must be completed before you can proceed to Section 8 (Experiments). You can run `make validate_undergrad` or `make validate_grad` to compare your output with the reference outputs.

We do not have a hard efficiency requirement in this assignment, but please make sure your simulator finishes a simulation for one of the provided traces in a few minutes (For reference, the TA solution finishes a simulation in 3-7 seconds.) Otherwise, it will be difficult for the TAs to verify your code produces matching outputs. You can also `make clean` followed by `make FAST=1 validate_undergrad` to compile for validation with optimizations enabled. This brings the runtime of the simulator down to about 1 second.

7.1 Debug Outputs

We have provided debug outputs for you in the `debug_outs` directory. Note that we only used the shorter traces (50K), not the full length traces (2M). We produced these debug traces by running `med_nomiss`, `med_always_taken`, `med`, `big`, and `tiny` for all 5 benchmarks.

These are the flags for each of the above configurations:

- `med_nomiss` (branch mispredictions and cache misses disabled): -b 3 -p 15 -s 5 -a 3 -m 2 -l 2 -f 4 -d
- `med_always_taken` (branch predictor always predicts taken): -b 0 -s 5 -a 3 -m 2 -l 2 -f 4
- `med`: -b 3 -p 15 -s 5 -a 3 -m 2 -l 2 -f 4
- `big`: -b 3 -p 15 -s 8 -a 4 -m 3 -l 2 -f 8
- `tiny`: -b 3 -p 15 -s 4 -a 1 -m 1 -l 1 -f 2

Note that for the sake of brevity (and disk space), the debug outputs may not include the contents of all data structures used in the TA solution.

We have also provided `branchsim_cpp_debug_printfs.txt` and `procsim_cpp_debug_printfs.txt`, which contains all of the print statements in the simulator used to generate the debug outputs. The print statements already written in the provided template code is not included. To enable debug output generation in your own code, use `make DEBUG=1` (you should `make clean` first!) which will allow you to use preprocessor directives to control whether or not your code generates print statements, like this:

```
#ifdef DEBUG
   printf("Yeh-Patt: Creating a history table of %" PRIu64 " entries of length %"
   PRIu64 "\n", myvar1, myvar2);
#endif
```

Please do not submit code that always generates debug outputs (that is, it prints without the `#ifdef DEBUG` ... `#endif` directives), as this will break the autograder and cause you to not match reference outputs.

7.2 Debugging

To debug, please use GDB and Valgrind as demonstrated in Appendix B. However, the full length traces (2M) will likely take a while to run under GDB or Valgrind, so we recommend debugging with the shorter length traces (50K). We also encourage comparing with the debug outputs (the tool `diff` is useful) before coming to office hours for help debugging your code.

8 Experiments

Once your simulator has been validated and matches the `ref_outs`, you will need to find the optimum pipeline configuration for each trace (Only consider the full length traces: `traces/*_2M.trace`). This means you will submit 5 configurations. **An optimal pipeline has the best possible IPC (Instructions Per Cycle) with the lowest resource utilization.**

Here are the constraints for your experiments. See Section 2.1 for a detailed explanation of the meaning of these parameters:

- Don't pass `-x` or `-d`.
- If `-b` is 2 (Yeh-Patt):
  - `-p`: 14-15.
  - `-h`: 11 when `-p` is 14, 12 when `-p` is 15.
- If `-b` is 3 (Gshare):
  - `-p`: 15-16.
- `-s`: 4-8.
- `-a`: 1-4.
- `-m`: 1-3. Cannot be larger than `-a`. Additionally, must be at least 2 if `-a` is 3 or greater.
- `-l`: 1-2. Must be 2 if `-a` is greater than 2.
- `-f`: 2, 4, or 8. Must be greater than or equal to the maximum value of `-a`, `-m`, and `-l`.

While we do not require you to search the entire space of configurations, you will need to search a large portion of it to ensure you are not missing information that is crucial to your decision making process.

Within these constraints, you are expected to find a configuration that performs at least 90% as well as the best performer and uses as few resources as possible. You should consider the impact of each architectural knob on the architecture as a whole. When evaluating whether a configuration is optimal, consider the statistics you are calculating beyond just IPC. You are responsible for defining "resources" however you see fit. So long as your justification and reasoning are sound, many interpretations will be accepted. Your report must contain a justification for why your chosen configuration is ideal. Include evidence and analysis in terms of plots, explanations, research, and logic.

Ensure that the report is in a file named `<last name>_report.pdf`. Please submit a `PDF` file: do not submit any other file types (ex. Microsoft Word).

9 What to Submit to Gradescope

Please run `make submit` and submit the resulting tarball (`tar.gz`) to Gradescope. Do not submit the assignment PDF, traces, or other unnecessary files (using `make submit` avoids this). Running `make submit` will include PDFs in the project directory in the tarball, but **please make sure** that the command ran successfully and your experiments report PDF is present in your submission tarball. We will create a simple Gradescope autograder that will verify that your code compiles and matches the 10 million–branch `gcc` trace and a subset of the other reference traces. This autograder is a smoke test to check for any incompatibilities or big issues; it is not comprehensive.

Make sure you untar and check your submission tarball to ensure that all the required files are present in the tar before making your final submission to Gradescope!

10 Grading

You will be evaluated on the following criteria:

+0 : You don't turn in anything (significant) by the deadline.
+50 : You turn in well commented significant code that compiles and runs but does **not** match the validation.
+10 : Your simulator **completely matches** the validation outputs for the branch predictors.
+25 : Your simulator **completely matches** the all validation outputs.
+10 : You ran experiments and found the optimum configuration matching the constraints in the Experiments section and presented sufficient evidence and reasoning.
+5 : Your code is well formatted, commented and does not have any memory leaks or violations! We may also run some special cases for these points. Check out Appendix B for some useful tools.

Points for the experiments and/or the memory leak check cannot be awarded without first matching all validation traces. This is non-negotiable.

Appendix A - Plagiarism

We take academic plagiarism very seriously in this course. Any and all cases of plagiarism are reported to the Dean of Students. We use accepted forensic techniques to determine whether there is copying of a coding assignment. You may not do the following in addition to the Georgia Tech Honor Code:

- Copy/share code from/with your fellow classmates or from people who might have taken this course in prior semesters.
- Publish your assignments on public repositories, github, etc, that are accessible to other students.
- Submit an assignment with code or text from an AI assistant (e.g., ChatGPT).
- Look up solutions online. Trust us, we will know if you copy from online sources.
- Debug other people's code. You can ask for help with using debugging tools (Example: Hey XYZ, could you please show me how GDB works), but you may not ask or give help for debugging the cache simulator.

- You may not reuse any code from earlier courses even if you wrote it yourself. This means that you cannot reuse code that you might have written for this class if you had taken it in a prior semester. You must write all the code yourself and during this semester.

Using AI Assistants:

Anything you did not write in your assignment will be treated as an academic misconduct case. If you are unsure where the line is between collaborating with AI and copying AI, we recommend the following heuristics:

Heuristic 1: Never hit "Copy" within your conversation with an AI assistant. You can copy your own work into your own conversation, but do not copy anything from the conversation back into your assignment. Instead, use your interaction with the AI assistant as a learning experience, then let your assignment reflect your improved understanding.

Heuristic 2: Do not have your assignment and the AI agent open at the same time. Similar to the above, use your conversation with the AI as a learning experience, then close the interaction down, open your assignment, and let your assignment reflect your revised knowledge. This heuristic includes avoiding using AI directly integrated into your composition environment: just as you should not let a classmate write content or code directly into your submission, so also you should avoid using tools that directly add content to your submission.

Deviating from these heuristics does not automatically qualify as academic misconduct; however, following these heuristics essentially guarantees your collaboration will not cross the line into misconduct.

Appendix B - Helpful Tools

You might the following tools helpful:

- gdb: The GNU debugger will prove invaluable when you eventually run into that segfault. The Makefile provided to you enables the debug flag which generates the required symbol table for gdb by default.
  - You can invoke gdb for the with `gdb ./proj2sim` and then run `run -i traces/<trace> <more proj2sim args>` at the gdb prompt.
- Valgrind: Valgrind is really useful for detecting memory leaks. Use the following command to track all leaks and errors for the simulator:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes \
    ./proj2sim -i traces/<trace> <more proj2sim args>
```

References

[1] S. McFarling, "Combining Branch Predictors," 1993.

[2] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, ser. MICRO 24. New York, NY, USA: Association for Computing Machinery, 1991, p. 51–61. [Online]. Available: https://doi.org/10.1145/123465.123475