CS {4/6}290 & ECE {4/6}100 - Spring 2023
Project 3: Tomasulo Simulator

Dr. Thomas Conte

Due: April 7th 2023 @ 11:55 PM

Version: 1.1.1

Changelog

1. Version 1.1.1 (2023-03-23): A student noticed that the debug logs did not reflect certain behavior due to a lack of clarity in the description of an instruction. In order to prevent special case behavior, this is handled in an updated driver. Clarified dispatch in debug logs. No PDF changes
2. Version 1.1.0 (2023-03-22): Clarified which loads access cache. Pseudocode for memory disambiguation was incorrect and did not implement the algorithm correctly. Debug outs and Reference outs updated. Changes highlighted.
3. Version 1.0.0 (2023-03-21): 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.
- 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 TAs.
- 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 **C99 or C++17** standards, using only the standard libraries.

2 Introduction

In this project, you will implement a timing simulator for an Out-of-Order FOCO CPU using the PREGS/RAT approach. This CPU is equipped with a Branch Predictor, Instruction Cache, Data Cache, and Store Buffer. In order to enable correct behavior with memory operations in parallel, the CPU uses a memory disambiguation algorithm. Read the following sections carefully for simulator details.

Note: This project MUST be done individually. Please follow all the rules from Section 1 and review Appendix A.

3 Simulator Specifications

3.1 Basic Out of Order Architecture

Your simulator will implement the stages of an out-of-order (FOCO) processor using PREGs and a RAT, and functional support for cache misses and branch mispredictions. Your simulator will take in a handful of parameters that are used to define the processor configuration. The provided code will call your per-stage simulator functions in reverse order to emulate pipelined behavior. Figure 1 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.

3.2 Simulator Parameters

The following command line parameters can be passed to the simulator:

- `-A`: The number of ALU units in the Execute stage
- `-M`: The number of Multiply (MUL) units in the Execute stage
- `-L`: The number of Load/Store (LSU) units in the Execute stage
- `-S`: The number of reservation stations **per** function unit. (You will model a unified scheduling queue with $S * (A + M + L)$ reservation stations)
- `-D`: This flag has no argument; if it is passed, cache misses and mispredictions are disabled by the driver
- `-F`: The "Fetch Width" is the number of instructions fetched/added to the dispatch queue per cycle
- `-P`: The number of Physical registers. There are $P + 32$ ROB entries
- `-I`: The input trace file
- `-H`: Print the usage information

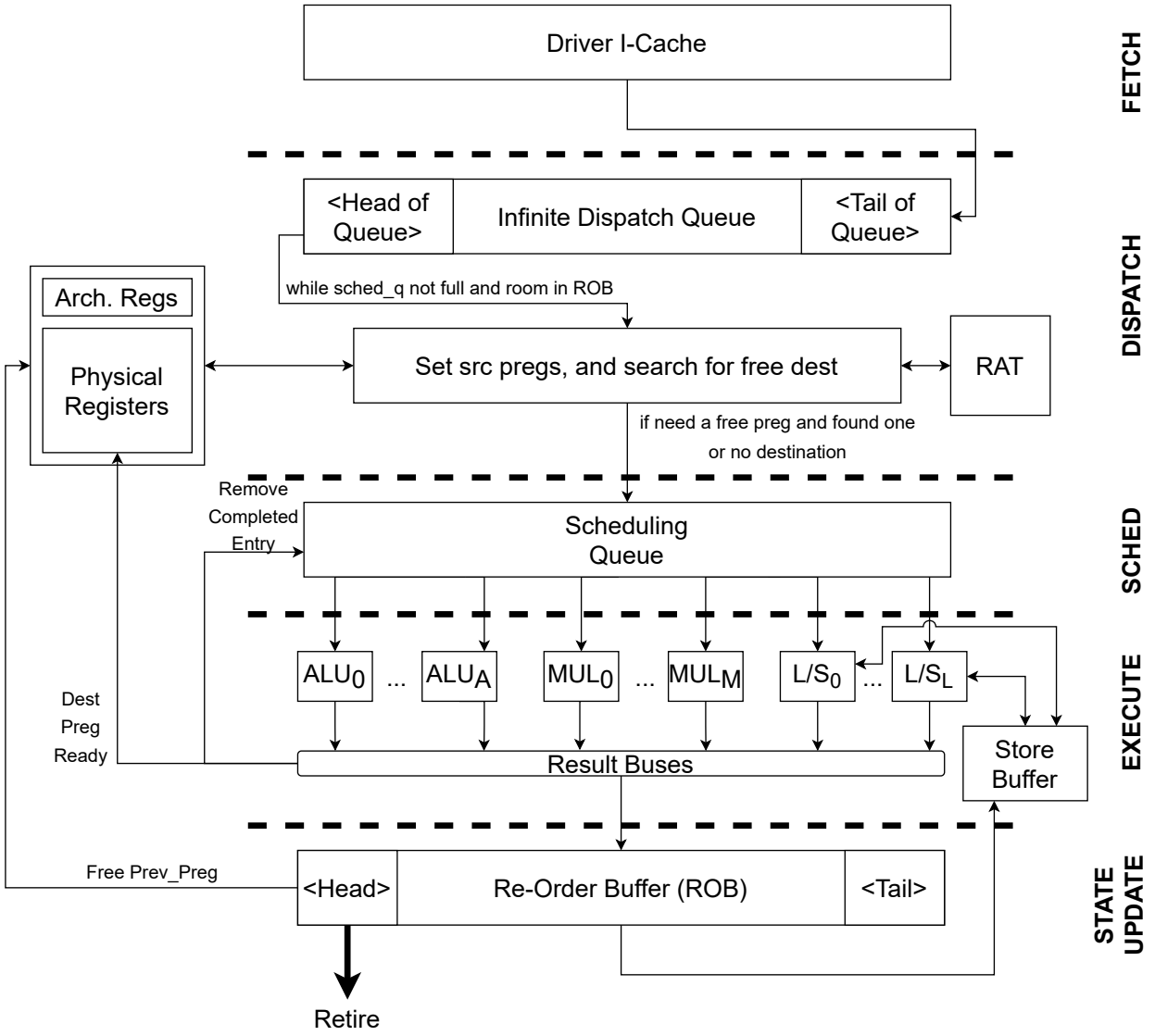

Figure 1: Overview of the Out of Order Architecture. An arrow that crosses a dotted line indicates that the value is latched at the end of a cycle. An arrow that does not cross a dotted line indicates that the associated action occurs within the clock cycle.

3.3 Trace Format

Each line in a trace represents the following:
`<Address> <Opcode> <Dest Reg #> <Src1 Reg #> <Src2 Reg #> <LD/ST Addr> <Dyn Instr Count> <Branch Misprediction> <I-Cache Miss> <D-Cache Miss>`
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 (for example) there is no destination register. The Branch Misprediction, I-Cache Miss, and D-Cache Miss flags are set to 1 when these events occur. The opcodes map to the following operations and Function Units:

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

3.4 The Memory Hierarchy

You will implement a Store Buffer that holds a store between the time it is completed and it is retired. In the Execute Stage, when progressing the Load-Store Units, you will first progress all loads through the load store units, then you will progress all stores. Loads can take a variable number of cycles to execute, more details will be discussed later in this section. Stores always complete in the first cycle and in this cycle, they will add their load/store address to the store buffer.

3.4.1 The Cache

The traces have been annotated with I-Cache misses and D-Cache misses based on a 64 KiB 16-way set-associative cache with 64 byte cache blocks `(C=16, B=6, S=4` with a TLB that never misses. This cache serves as a $1 + L$-ported unified Instruction and Data Cache that imposes no structural hazards. Because of this you do not need to simulate a data cache.

The data cache has a 2 cycle hit time and a misses are repaired from an L2 cache that always hits. As such the cache miss penalty is (`L1_MISS_PENALTY`). While caches typically take 2 cycles with 1 cycle for array-lookup and 1 cycle for tag compare, in your simulation, you can just check `inst_t.dcache_miss` in cycle 2 in the Load/Store Unit. In the event that there is a miss, the Load/Store Unit must stall until cycle (2 + `L1_MISS_PENALTY`) at which point the instruction will complete. Note that due to the existence of a store buffer, you only check the cache for loads that miss in the store buffer.

This means that Stores always take 1 cycle and Loads can take 1, 2, or 12 cycles.

3.4.2 The Store Buffer

The Store buffer exists as a hit-time reduction technique for store instructions. The Store buffer is allowed to have an unbound number of entries, however you will see in this section that it has an upper bound of the number of entries in the ROB.

In the cycle when a store begins execution, the store address is added to the FIFO (First-In-First-Out) store buffer.

In State Update, we keep a count of the number of stores that are retired that cycle. **In the following cycle**, we remove that many entries from the FIFO store buffer. This 1 cycle delay is introduced to maintain the availability of data in a structure until the end of the cycle.

3.5 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()`.

In the event of a mispredicted branch, the driver will return the branch instruction and then 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 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 `NULL` instructions until it resolves the I-Cache miss, at which point it will resume returning valid instructions. This means that your code will only be notified of I-Cache misses once they are resolved. You will need to inspect instructions returned by `procsim_driver_read_inst()` to determine if they experienced an I-Cache misses.

If the driver returns a `NULL` instruction, do not add it to the dispatch queue.

3.6 Dispatch

In dispatch we will attempt to fill the reservation stations in the scheduling queue with instructions from the head of the dispatch queue. Remember that instructions are dispatched in program order. Dispatch will set a reservation station's function unit specifier, will set source pregs using the Register Alias Table, and if an instruction has a destination, it will search the Physical Register File for a free preg and will set the `prev_preg`, set the destination preg, and update the RAT. If you are not able to find a free preg for a destination, dispatch must stall and the instruction cannot be fired.

Note that not every instruction will have all of `src1, src2, dest` specified. You will need to account for this in dispatch and schedule.

The dispatch stage will allocate room in the ROB for the instruction as it places it into a reservation station. If there is not sufficient room in the ROB, Dispatch will stall.

3.6.1 The Physical Register File

The physical register is combined with the Architectural register such that the first `NUM_REGS` indices correspond to the Architectural Registers, and the next `P` registers correspond to physical registers that you can allocate to. When searching for a free preg, dispatch starts at index `NUM_REGS`. Initially, all register file entries are free and ready.

You should use an O(1) access structure for the register file because you will need to access it in multiple places so it is better to access it via an array/vector instead of a list/linked-list.

3.6.2 Register Alias Table

The Register Alias Table (RAT) maps an architectural register to the most recent preg allocated to remove a false dependency. The RAT is initialized to point to the architectural registers and has `NUM_REGS` entries.

3.7 Schedule

The schedule stage searches the scheduling queue for instructions that are ready to fire (their source pregs are ready) and fires them. 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. However, memory operations cannot be naively reordered like this due to the existence of pointers and aliasing...

3.7.1 Memory Disambiguation and Consistency

Because the superscalar processor may have more than one load/store unit, we need to ensure that any out of order memory transactions do not cause program correctness issues. You will need to implement the following memory disambiguation algorithm to ensure that we do not violate program correctness.

- A Load instruction can be fired before any other Load instruction that precedes it in program order completes.
- A Load instruction can **never** be fired before ALL Store instructions that precedes it in program order complete.
- A Store instruction can **never** be fired before ALL Load and Store instructions that precede it in program order complete.
- Only 1 Store can be fired per cycle.

If you have implemented the scheduling queue correctly, any instruction currently in the scheduling queue has not been completed. Example Pseudocode: Fixed condition for Stores

```
fired_store = false
for CurRS in Scheduling_Queue where pregs ready: // Scheduling_Queue sorted in program order
    if ( !CurRS.sources_ready ) continue; // skip if not ready
    // Attempting to fire Reservation Station: CurRS
    // Other Opcodes not shown
    // Loads:
    canfire = true;
    for OtherRS in Scheduling_Queue excluding CurRS:
        if ( (OtherRS is STORE) and (OtherRS preceeds CurRS in program order) ):
            // Cannot fire Load
            canfire = false
        endif
    endfor
    if canfire:
        // look for available LSU and fire if found one
    endif
    // Stores:
    canfire = true;
    for OtherRS in Scheduling_Queue excluding CurRS:        // v1.1.0
        if ( ( ( (OtherRS is LOAD) or (OtherRS is STORE) )  // v1.1.0
            and (OtherRS preceeds CurRS in program order) ) // v1.1.0
            or (fired_store) ):                             // v1.1.0
            // Cannot fire store since already fired store or
            // preceding memory operation is in scheduling queue
            canfire = false
        endif
    endfor
    if canfire:
        // look for available LSU
        if found free LSU:
            // set up lsu
            fired_store = true
        endif
    endif
endfor
```

3.8 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 busses to mark destination pregs as ready and remove entries from the scheduling queue.

3.8.1 ALU

This is a single-stage unit.

3.8.2 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.

3.8.3 LSU

The Load/Store unit is a single-stage unit that will simulate store buffer and data cache accesses.

First, all Loads progress through the pipeline. The following events may occur:

- Cycle 1: check the store buffer. If there is an entry with the same address as the load instruction's load/store address, the load instruction is completed, a result bus entry is allocated, and the unit is marked as free.
- Cycle 2: check if the instruction has a cache miss. If there is no cache miss, the load instruction is completed, a result bus entry is allocated, and the unit is marked as free.
- Cycle (2 + `L1_MISS_PENALTY`): The load instruction is completed, a result bus entry is allocated, and the unit is marked as free.

If there is a store buffer hit, the Load takes 1 cycle, if there is a cache hit, the load takes 2 cycles, if there is a cache miss, the load takes (2 + `L1_MISS_PENALTY`) cycles.

Then, Stores progress through the pipeline:

- Cycle 1: Add the instruction's load/store address into the FIFO store buffer. The Store instruction is completed, a result bus entry is allocated, and the unit is marked as free.

3.8.4 Result Buses

As an instruction finishes executing in its function unit, it will pass its values to the result buses. These values will be used to mark destination pregs as ready. Completed instructions are removed from the scheduling queue. These busses will also place the instruction in the ROB in program order. All of this happens within the same cycle!

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

3.9 State Update

State update performs 2 functions for us, progressing the store buffer and updating architectural state in program order. Every cycle you will perform the following two operations.

First, you will pop off as many entries from the store buffer as stores that were retired in the **previous** cycle.

Second, you will, in program order, retire as many instructions as possible. Retiring an instruction will commit their destination register values (if present) to the Architectural Registers, (since we aren't modeling data, you don't actually need to model this behavior). Retiring an instruction will also free its `prev_preg` in the physical register file. If the `prev_preg` is actually an architectural register, do not mark it as free. You will need to count the number of stores retired so you can do step 1 the following cycle.

If you retire a mispredict, set `retired_mispredict_out` to true and stop retiring any more instructions. (Realistically, since the driver will return `NULL` until you retire the mispredict, there should be no more valid entries in the ROB anyways.

We assume misprediction retirement has side-effects, so instructions cannot begin to fill the pipeline until the following cycle. This is why the `procsim_do_cycle` does not call the other stages in the event of a mispredict retire. Note that you may have stores that retired in the same cycle as the mispredict so you still need to pop those in the following cycle.

3.10 Initial Conditions

All architectural registers in the register file are ready.
All physical registers in the register file are free but not ready.
All function units are empty/ready.
All queues and the ROB are empty.

4 Statistics

You will need to keep track of the following statistics:

- `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
- `instructions_retired`: The number of instructions that exit the ROB
- `branch_mispredictions`: The number of branch mispredictions the processor encounters. (The provided code increments this statistic for you)
- `icache_misses`: The number of I-cache misses encountered by fetch. You should increment this when you see fetch instruction that has the `icache_miss` flag set to true
- `reads`: The number of load instructions encountered in the execution of the program
- `store_buffer_read_hits`: the number of reads that hit in the
- `dcache_reads`: number of reads that go to the L1 cache (only incremented if there is no store buffer hit)
- `dcache_read_misses`: number of read misses in the L1 cache
- `dcache_read_hits`: number of read hits in the L1 cache
- `store_buffer_hit_ratio`: fraction of reads that hit in the store buffer

- `dcache_read_miss_ratio`: fraction of reads to the L1 cache that miss
- `dcache_ratio`: fraction of reads that go to the L1 cache
- `dcache_read_aat`: the AAT for read accesses to the L1 cache
- `read_aat`: the effective AAT for reads: `1 * store_buffer_hit_ratio + dcache_ratio * dcache_read_aat`
- `no_dispatch_pregs_cycles`: Cycles where dispatch was stalled due to not being able to find a free preg
- `rob_stall_cycles`: Cycles in which dispatch stops putting instructions into the scheduling queue only because of the constraint on the maximum number of ROB entries
- `no_fire_cycles`: cycles where nothing in the scheduling queue could be fired
- `dispq_max_size`: The maximum number of instructions in the dispatch queue during execution. You should update this at the end of `procsim_do_cycle()`
- `schedq_max_size`: 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
- `instructions_in_trace`: The number of instructions in the trace (this is handled by the driver)

The details of the statistics are also available in `procsim.hpp`.

5 Implementation Details

You have been provided with the following files:

- `procsim.cpp` - Your simulator 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
- `procsim_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.sh`: A shell script which runs your simulator and compares its outputs with the reference outputs. If you want to test some particular configurations instead of all configurations, you can pass them as arguments, like: `./validate.sh big tiny`. **You do not need to modify this file.**
- `Makefile`: A Makefile that contains the logic needed to compile your code. It has some useful features:
  - `make validate` will compile your code and run `validate.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 (namely, it terminates) 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 3.3. The driver will read these for you
  - `bfs_2_15`: A trace of the kernel for Breadth-First-Search on a graph with 2^{15} vertices of average degree 16
  - `cachesim_gcc`: A trace of the calls to sim_access in the solution to Project 1 with PIPT, (C=15, B=6, S=4). The input to the simulator was the first 1500 accesses of the GCC trace for project 1

  - `perceptron_gcc`: A trace of the calls to predict() and update_predictor() in the solution to Project 2 for a perceptron predictor with N=9. G=32. The input to the simulator was the first 4000 branches of the GCC trace for project 2
  - `tiled_mm`: A trace of the multiplication: A * B = C, where A, B, and C are [64x64], with 3-dimensional tiling, Ti = 16, Tj = 16, Tk = 4.
- `ref_outs/`: A directory containing the output of the TA simulator(s) for some select configurations (`make validate` will print their configuration flags)

5.1 Provided Framework

The provided `procsim.cpp` includes detailed comments for implementing your simulator functions. Look for "TODO" comments, which indicate places you need to add code. In short, we suggest dividing your simulator logic into the provided (but initially empty) functions representing different pipeline stages. The provided implementation of `procsim_do_cycle()`, which we recommend using, invokes these pipeline stage functions in reverse order to prevent you from needing to manage pipeline buffers by hand.

Additionally, the provided header file `procsim.hpp` contains comments describing the meaning of statistics your simulator needs to collect in the `procsim_stats_t` struct. If you are unsure about the meaning of some statistics, please see Section 4 for more details.

5.2 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/`) and extract the provided project tarball and run the `6290docker*` script in the project tarball corresponding to your OS. That should open an 18.04 `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`, `./procsim`, etc.

> **Note**: Using this Docker image is not required if you have a Linux system (or environment) available and just want to make sure your code compiles on 22.04. We will set up a Gradescope autograder that automatically verifies we can successfully compile your submission.

6 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 7 (Experiments).

You can run `make validate` to compare your output with the reference outputs. If you want to test only one configuration for all four benchmarks, you can use the `validate.sh` script directly and pass it a configuration name it understands, like `./validate.sh tiny`.

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 less than a minute or two. (For reference, the TA solution finishes a simulation in a few (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` to compile for validation with optimizations enabled. This brings the runtime of the solution simulator down to 1 second.

6.1 Debug Outputs

We have provided debug outputs for you in a separate tarball on Canvas. We produced these debug traces by running the following configurations (here, we use the 25,000-instruction traces, not the full length traces):

- The `med` configuration for each trace
- The `tiny`, `med`, `med_nomiss`, and `big` configurations for `cachesim_gcc`

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 `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 are not included in `debug_prints.txt`. 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("\t\tUsing preg: P%lu for src1: R%d\n", my_var1, my_var2);
#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 not to match reference outputs.

6.2 Debugging

To debug, please use GDB, Valgrind, and Address Sanitizer as demonstrated in Appendix B and on Piazza. We recommend running a 25,000-instruction trace, e.g., passing `-I traces/cachesim_gcc_25K.trace` to `procsim`, since the full length traces will likely take a while to run under GDB or Valgrind. We also encourage comparing with the debug outputs (the tool `diff` is useful) before coming to office hours for help debugging your code.

7 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/*_full.trace`). This means you will submit 4 configurations. **An optimal pipeline has the best possible IPC (Instructions Per Cycle) with the lowest resource utilization.**

Here are the constraints for your experiments:

- misses are enabled
- The number of ALU units is 1, 2, or 3
- The number of MUL units is 1, or 2
- The number of LSU units is 1, 2, or 3
- The number of reservation stations per FU is 2, 4, or 8
- The number of pregs is 64, 96, or 128
- You may fetch 2, 4, or 8 instructions per cycle

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 and not other file types (no Microsoft Word documents, please).

8 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 it worked properly 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 a subset of 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!

9 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
+5 : Your simulator **completely matches** the `med_nomiss` validation outputs
+25 : Your simulator **completely matches** the all validation outputs
+15 : 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! 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

Preamble: The goal of all assignments in this course is for you to learn. Learning requires thought and hard work. Copying answers thus prevents learning. More importantly, it gives an unfair advantage over students who do the work and follow the rules.

1. As a Georgia Tech student, you have read and agreed to the Georgia Tech Honor Code. The Honor Code defines Academic Misconduct as "any act that does or could improperly distort Student grades or other Student academic records."
2. You must submit an assignment or project as your own work. Absolutely No Collaboration on Answers Is Permitted. Absolutely no code or answers may be copied from others — such copying is Academic Misconduct. NOTE: Debugging someone else's code is (inappropriate) collaboration.
3. Using code from GitHub, via Googling, from Stack Overflow, etc., is Academic Misconduct (Honor Code: Academic Misconduct includes "*submission of material that is wholly or substantially identical to that created or published by another person*").
4. Publishing your assignments on public repositories accessible to other students is unauthorized collaboration and thus Academic Misconduct.
5. Suspected Academic Misconduct will be reported to the Division of Student Life Office of Student Integrity. It will be prosecuted to the full extent of Institute policies.
6. Students suspected of Academic Misconduct are informed at the end of the semester. Suspects receive an Incomplete final grade until the issue is resolved.
7. We use accepted forensic techniques to determine whether there is copying of a coding assignment.
8. If you are not sure about any aspect of this policy, please ask Dr. Conte.

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 with `gdb ./procsim` and then run `run -I traces/cachesim_gcc_25K.trace <more procsim args>` at the gdb prompt
- Valgrind: Valgrind is really useful for detecting memory leaks. Use the following command to track all leaks and errors:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes \
    ./procsim -I traces/cachesim_gcc_25K.trace <more procsim args>
```

- Address Sanitizer: More details will be posted on Piazza soon.