NC State University

Department of Electrical and Computer Engineering

ECE 463/521: Fall 2014

Project #3: Dynamic Instruction Scheduling (Version 1.0)

Due: Tuesday, Dec. 2, 2014, 11:59 PM

1. Ground Rules

1. All students must work alone.
2. Sharing of code between students is considered cheating and will receive appropriate action in accordance with University policy. 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.
3. 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.
4. You *must* do all your work in the C/C++ or Java languages. Exceptions must be approved. The C language is fine to use, as that is what many students are trained in. Basic C++ extensions to the C language (*e.g.*, classes instead of structs) are encouraged (but by no means required) because it enables more straightforward code-reuse.
5. Use of the Eos Linux environment is *required*. This is the platform where the TAs will compile and test your simulator. (WARNING: If you develop your simulator on another platform, get it working on that platform, and then try to port it over to Eos Linux at the last minute, you may encounter major problems. Porting is not as quick and easy as you think unless you are an excellent programmer. What's worse, malicious bugs can be hidden until you port the code to a different platform, which is an unpleasant surprise close to the deadline.)

2. Project Description

In this project, you will construct a simulator for an out-of-order superscalar processor based on Tomasulo's algorithm that fetches, dispatches, and issues *N* instructions per cycle. Only the dynamic scheduling mechanism will be modeled in detail, *i.e.*, perfect caches and perfect branch prediction are assumed. For bonus part, a superscalar integrated with two level caches will be modeled.

3. Inputs to Simulator

The simulator reads a trace file in the following format:

```
<PC> <operation type> <dest reg #> <src1 reg #> <src2 reg #> <mem address>
<PC> <operation type> <dest reg #> <src1 reg #> <src2 reg #> <mem address>
...
```

Where:

- `<PC>` is the program counter of the instruction (in hex).
- `<operation type>` is either "0", "1", or "2".
- `<dest reg #>` is the destination register of the instruction. If it is **-1**, then the instruction does not have a destination register (for example, a conditional branch instruction). Otherwise, it is between 0 and 127.
- `<src1 reg #>` is the first source register of the instruction. If it is **-1**, then the instruction does not have a first source register. Otherwise, it is between 0 and 127.
- `<src2 reg #>` is the second source register of the instruction. If it is **-1**, then the instruction does not have a second source register. Otherwise, it is between 0 and 127.
- `<mem address>` is the memory address for memory access instructions. If it is **0**, then it's not a memory access instruction. Otherwise, it is a hex address.

For example:

```
ab120024  0   1  2  3  0
ab120028  1  -1  1  3  0
ab12002c  2   1  4 -1  ffe04540
```

Means:

"operation type 0" R1, R2, R3, 0
"operation type 1" -, R1, R3, 0 // no destination register!
"operation type 2" R1, R4, - , ffe04540 // memory access instruction

Note:
`<mem address>` is only used for bonus part (integration of data caches). You can ignore the last part if no data caches are modeled.

4. Outputs from Simulator

The simulator outputs the following measurements after completion of the run:

1. Total number of instructions in the trace.
2. Total number of cycles to finish the program.
3. Average number of instructions completed per cycle (IPC).

The simulator also outputs the timing information for every instruction in the trace, in a format that is used as *input* to the *scope tool*. The scope tool's input format is described in a later section.

For the *bonus part* (integration of data caches), the simulator also outputs

1. Number of L1 accesses
2. Number of L1 misses
3. Contents of L1 cache
4. Number of L2 accesses (if L2 cache size is not 0)
5. Number of L2 misses
6. Contents of L2 cache

5. Simulator Specification

5.1. Microarchitecture to be Modeled

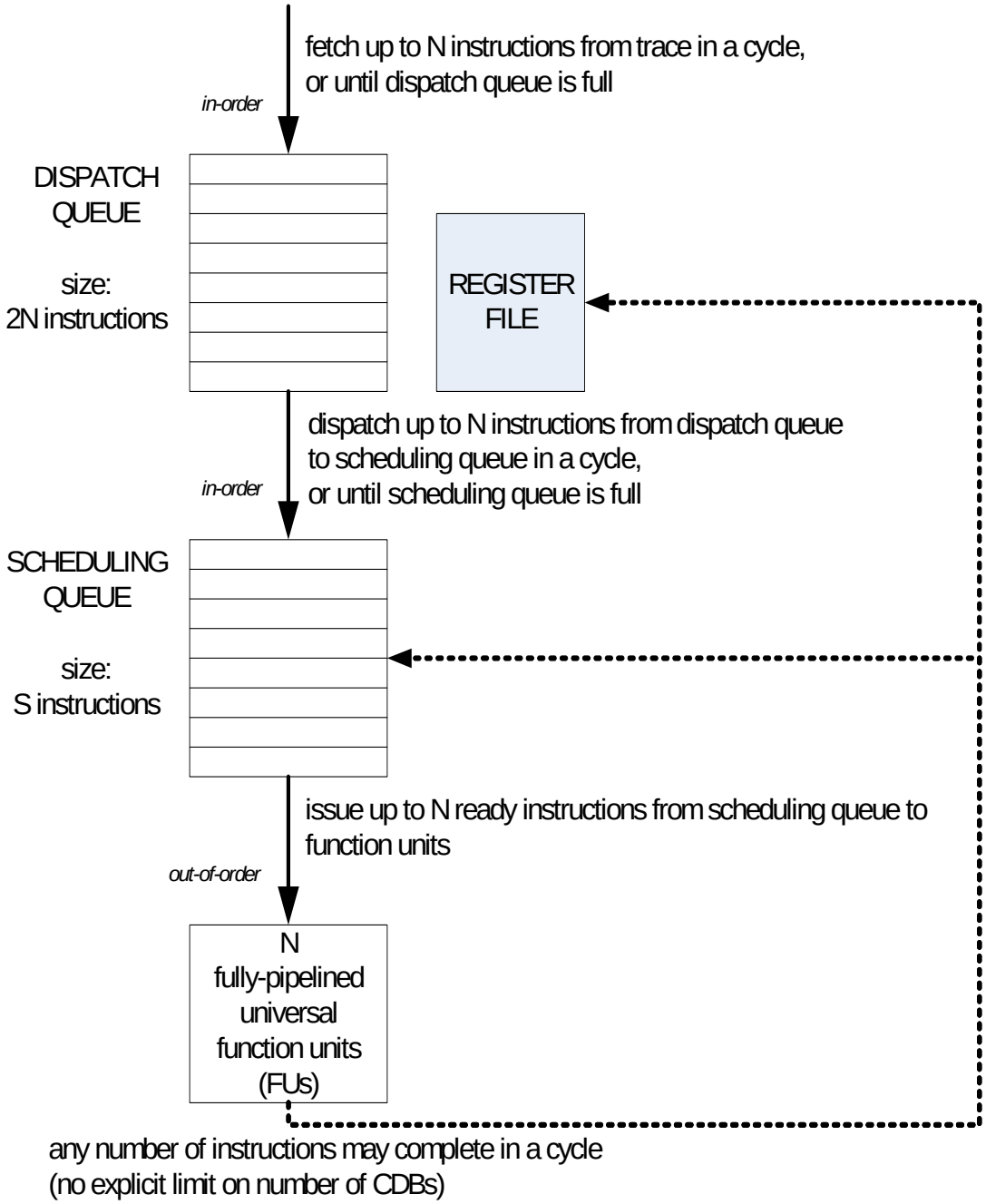

Figure 1. Overview of microarchitecture to be modeled, including the terminology and parameters used throughout this specification.

Tags:
In a simulator, a simple way to generate tags for instructions is to assign increasing sequence numbers. Assign a tag of 0 to the first instruction in the trace, a tag of 1 to the second instruction in the trace, and so on. The traces are sufficiently short, that you should not need to reuse tags.

Superscalar microarchitecture:
The microarchitecture is N-way superscalar. This means that up to N instructions can be fetched, dispatched, and issued each cycle. In a given cycle, fewer than N instructions may be fetched, dispatched, or issued, however, due structural hazards or data dependences.

In a given cycle, up to N instructions are fetched into the Dispatch Queue. Fetching stalls if the Dispatch Queue is full. Fetching stops altogether after the last instruction in the trace is fetched.

The Dispatch Queue is 2N instructions in size. The reason is that the Dispatch Queue abstractly models the pipeline registers for the instruction fetch and instruction dispatch stages. That is, it models the effect of two pipeline stages that are each N instructions wide. The details of how this effect is simulated, is explained later.

In a given cycle, up to N instructions (in program order) are dispatched from the Dispatch Queue to the Scheduling Queue. Dispatching stalls if the Scheduling Queue is full. As the instructions are dispatched, their source and destination registers are renamed using Tomasulo's algorithm. Use the tags that were assigned as described above.

In this microarchitecture, there is one centralized pool of reservation stations, called the Scheduling Queue. The Scheduling Queue is S instructions in size. Thus, S is the size of the window.

As instructions in the Scheduling Queue become ready (all source operands are ready), they are candidates for issuing from the Scheduling Queue to the Function Units. In a given cycle, up to N ready instructions can be issued. Priority among ready instructions should be based on program order, from oldest instruction (lowest tag) to newest instruction (highest tag). Tags can be used to establish priority since they were derived from sequence numbers. While the chosen priority scheme is arbitrary, it is essential for matching your simulator with ours. An instruction is removed from the Scheduling Queue when it issues to a Function Unit.

There are N function units (FUs). Each FU can execute any type of instruction (sometimes referred to as a "universal function unit" in the literature). The operation type of an instruction indicates its execution latency: Type 0 has a latency of 1 cycle, Type 1 has a latency of 2 cycles, and Type 2 has a latency of 5 cycles (The latency of Type 2 is different for bonus part, see Section 5.3 for details). Each FU is fully pipelined. Therefore, a new instruction can begin execution on a FU every cycle.

Due to different execution latencies and the fact that the FUs are fully pipelined, it is possible for more than N instructions to complete in the same cycle. As a simplification, you do not have to limit the number of instructions that can complete in a given cycle. This is tantamount to having as many result buses ("common data buses", or CDBs) as the maximum number of instructions

that can possibly complete in the same cycle. You do not have to determine what that number is. Instead, just permit all instructions that finish execution in a given cycle to advance from the FUs to the writeback stage. When an instruction completes, it broadcasts this fact to the Scheduling Queue (to wake-up dependent instructions) and the Register File (to possibly update its state).

About register values:
For the purpose of determining the number of cycles it takes for the microarchitecture to run a program, the simulator does not need to use and produce actual register values. This is why the initial Register File values are not provided and the instruction opcodes are omitted from the trace. All that the simulator needs, to determine the number of cycles, is the microarchitecture configuration, execution latencies of instructions (operation type), and register specifiers of instructions (true, anti, and output dependences).

Imprecise interrupts:
The microarchitecture does not maintain precise interrupts. There is no Reorder Buffer. Thus, instructions update the Register File as soon as they complete.

5.2. Guide to Implementing your Simulator

You can implement your simulator in any way. To help you match the spec and also increase the efficiency of your simulator, however, I recommend you organize your simulator, at a high-level, as follows.

1. Define 5 states that an instruction can be in (*e.g.*, use an enumerated type): IF (fetch), ID (dispatch), IS (issue), EX (execute), WB (writeback).
2. Define a circular FIFO that holds all active instructions in their program order. Conceptually, this is like a ROB, but it is a "fake ROB" since it is NOT used by the simulator to model in-order retirement. Instead, the fake-ROB is (i) a matter of convenience and efficiency for simulator implementation and (ii) necessary for using the scope tool that will be provided for debugging and validation. Regarding (i), the fake-ROB can serve as a single place where everything about an instruction is maintained, so that the instruction information doesn't need to literally move through the pipeline. Each entry in the fake-ROB should be a data structure containing per-instruction information, for example, state of the instruction (which stage it is in), operation type, operand state, sequence number (tag), *etc*. An instruction is added to the fake-ROB when it is fetched from the trace and removed when it has reached the WB state and all prior instructions have been removed from the fake-ROB. Regarding (ii), the fake-ROB is useful for printing out stuff in program order for use by the scope tool described later. Since we aren't really modeling a ROB, make the fake-ROB large enough that it never overflows – I suggest 1024 entries.
3. Define 3 lists:
    a. dispatch_list: This contains a list of instructions in either the IF or ID state. The dispatch_list models the Dispatch Queue. (By including both the IF and ID states, we don't need to separately model the pipeline registers of the fetch and dispatch stages.)
    b. issue_list: This contains a list of instructions in the IS state (waiting for operands or available issue bandwidth). The issue_list models the Scheduling Queue.

c. execute_list: This contains a list of instructions in the EX state (waiting for the execution latency of the operation). The execute_list models the FUs.

4. Call each pipeline stage in reverse order in your main simulator loop, as follows. The detailed comments indicate tasks to be performed. The order among these tasks is important.

```
do {
     FakeRetire();  // Remove instructions from the head of
                    // the fake-ROB until an instruction is
                    // reached that is not in the WB state.

     Execute();     // From the execute_list, check for
                    // instructions that are finishing
                    // execution this cycle, and:
                    // 1) Remove the instruction from
                    //    the execute_list.
                    // 2) Transition from EX state to
                    //    WB state.
                    // 3) Update the register file state
                    //    (e.g., ready flag) and wakeup
                    //    dependent instructions (set their
                    //    operand ready flags).

     Issue();       // From the issue_list, construct a
                    // temp list of instructions whose
                    // operands are ready - these are the
                    // READY instructions. Scan the READY
                    // instructions in ascending order of
                    // tags and issue up to N of them.
                    // To issue an instruction:
                    // 1) Remove the instruction from the
                    //    issue_list and add it to the
                    //    execute_list.
                    // 2) Transition from the IS state to
                    //    the EX state.
                    // 3) Free up the scheduling queue
                    //    entry (e.g., decrement a count
                    //    of the number of instructions in
                    //    the scheduling queue)
                    // 4) Set a timer in the instruction's
                    //    data structure that will allow
                    //    you to model the execution
                    //    latency.

     Dispatch();    // From the dispatch_list, construct a
                    // temp list of instructions in the ID
                    // state (don't include those in the
                    // IF state - you must model the
                    // 1 cycle fetch latency). Scan the
                    // temp list in ascending order of
                    // tags and, if the scheduling queue
                    // is not full, then:
                    // 1) Remove the instruction from the
```

```
                    //    dispatch_list and add it to the
                    //    issue_list. Reserve a schedule
                    //    queue entry (e.g. increment a
                    //    count of the number of
                    //    instructions in the scheduling
                    //    queue) and free a dispatch queue
                    //    entry (e.g. decrement a count of
                    //    the number of instructions in
                    //    the dispatch queue).
                    // 2) Transition from the ID state to
                    //    the IS state.
                    // 3) Rename source operands by
                    //    looking up state in the register
                    //    file; rename destination
                    //    operands by updating state in
                    //    the register file.
                    //
                    // For instructions in the
                    // dispatch_list that are in the IF
                    // state, unconditionally transition
                    // to the ID state (models the 1 cycle
                    // latency for instruction fetch).

    Fetch();        // Read new instructions from the
                    // trace as long as 1) you have not
                    // reached the end-of-file, 2) the
                    // fetch bandwidth is not exceeded,
                    // and 3) the dispatch queue is not
                    // full.  Then, for each incoming
                    // instruction:
                    // 1) Push the new instruction onto
                    //    the fake-ROB. Initialize the
                    //    instruction's data structure,
                    //    including setting its state to
                    //    IF.
                    // 2) Add the instruction to the
                    //    dispatch_list and reserve a
                    //    dispatch queue entry (e.g.,
                    //    increment a count of the number
                    //    of instructions in the dispatch
                    //    queue).

} while (Advance_Cycle());
                    // Advance_Cycle performs several functions.
                    First, if you want to use the scope tool
                    (below), then it checks for instructions that
                    changed states and maintains a history of
                    these transitions. When an instruction is
                    removed from the fake-ROB -- FakeRetire()
                    function -- you can dump out this timing
                    history in the format read by the scope tool.
                    Second, it advances the simulator cycle.
                    Third, when it becomes known that the
```

```
fake-ROB is empty AND the trace is depleted,
the function returns ‘‘false’’ to terminate
the loop.
```

5.3. Bonus part: Integration of Data Caches (optional)

Integrate data caches with the microarchitecture modeled in Section 5.1 (Figure 2):

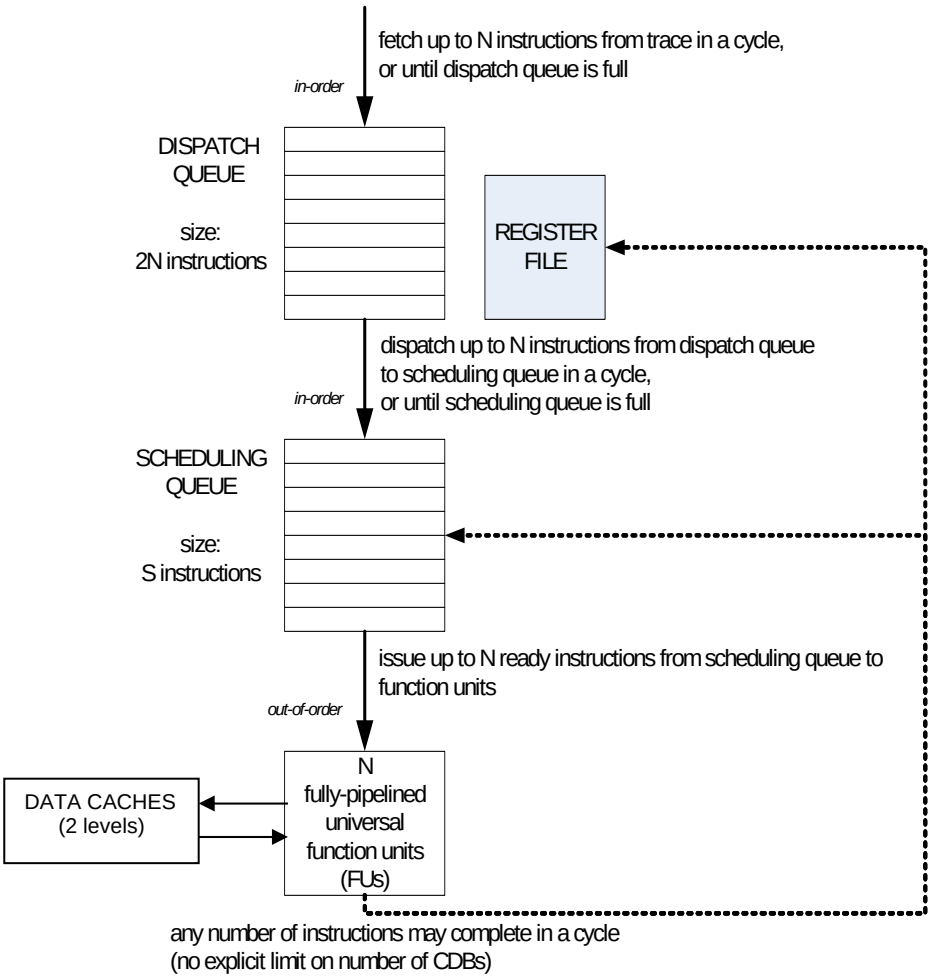

Figure 2 Overview of microarchitecture to be modeled, including the integration of data caches

For this part, instructions of Type 2 in the traces are memory operations. Consider all memory accesses to be read accesses. In the execute stage, for every instruction, access the data from the data caches. If it hits in the L1cache, the execution latency will be 5 cycles. If it misses in L1 cache but hits in L2 cache, the latency will be 10 cycles. If it misses both in L1 cache and L2 cache, the latency will be 20 cycles. If the L2 cache size is 0 (no L2 cache), the execution latency for a cache hit is 5 cycles, the latency for a cache miss is 20 cycles, which is the miss penalty of the data cache.

The cache configuration to be modeled is set-associative, using LRU replacement policy. For this part, all caches in the memory hierarchy will have the same BLOCKSIZE.

Note:
You can reuse your code from Project 1 to model the data caches.

*Note regarding simulator output: When printing out the final contents of data caches, you **don't** need to print the blocks from MRU block to LRU block (different from project 1). Just keep them unsorted so that your output is consistent with the output of the TAs' simulator.*

6. Helping you Debug and Validate: A Scope Tool

A tool is provided to you so as to allow you to display pipeline timing diagrams identical to the timing diagrams drawn in class.

- Location of tool: see website for location of tool.
- Usage: `scope <input-file> <output-file>`
- The tool has quite a bit of error checking to make sure you comply with formatting and usage, however, beware it is not error-proof. Warning messages will sometimes direct you in the right direction (often it just spits out my email address ☺).

You must provide an input file that encodes the timing of each instruction in the program. Your simulator dumps this timing information. There should be one line for each instruction in the program, and instructions must be dumped in program order. The format of each line is as follows. Note: you must substitute an integer everywhere there is a <> pair.

```
<seq_no> fu{<op_type>} src{<src1>,<src2>} dst{<dst>}
IF{<begin-cycle>,<duration>} ID{…} IS{…} EX{…} WB{…}
```

<seq_no> is the unique tag of the instruction (line number in trace, starting at 0). Substitute 0, 1, or 2 for the <op_type>. <src1>, <src2>, and <dst> are register numbers (include –1 if that is the case). For each of the IF, ID, IS, EX, and WB states, indicate the first cycle that the instruction was in that state followed by the number of cycles the instruction was in that state. The tool automatically does some consistency checks and dies if there is a problem, *e.g.*, begin cycle of ID should equal begin cycle of IF plus duration of IF.

Two output files are created:

- <output-file>: This contains the timing diagram. It is about ½ MB and is best displayed as a web page because web browsers provide scrollbars, hence…
- <output-file>.html: This is a very brief html shell that you can edit or keep as is. To view the html file, in a web browser type:
  - **file:**<full-path>**/**<output-file>**.html**
  - you can now view the timing diagram and compare it to mine

Go to the course website to view samples.

7. Validation and Other Requirements

7.1. Validation requirements

Sample simulation outputs will be provided on the website. These are called "validation runs". You must run your simulator and debug it until it matches the validation runs. (**No** debug runs will be provided for this project since the validation runs themselves contain debug information.)

Each validation run includes:
1. Timing information for every instruction. The format is described in Section 6.
2. The microarchitecture configuration.
3. All measurements as described in Section 4.

Your simulator must print outputs to the console (i.e., to the screen). (Also see Section 7.2 about this requirement.)

Your output must match both numerically and in terms of formatting, because the TAs will literally "diff" your output with the correct output. You must confirm correctness of your simulator by following these two steps for each validation run:
1) Redirect the console output of your simulator to a temporary file. This can be achieved by placing `> your_output_file` after the simulator command.
2) Test whether or not your outputs match properly, by running this unix command:
`diff –iw <your_output_file> <posted_output_file>`
The –iw flags tell "diff" to treat upper-case and lower-case as equivalent and to ignore the amount of whitespace between words. Therefore, you do not need to worry about the exact number of spaces or tabs as long as there is some whitespace where the validation runs have whitespace.

7.2. Compiling and running simulator

You will hand in source code and the TAs will compile and run your simulator. As such, you must meet the following strict requirements. Failure to meet these requirements will result in point deductions (see section "Grading").

1. You must be able to compile and run your simulator on Eos Linux machines. This is required so that the TAs can compile and run your simulator. If you are logging into an Eos machine remotely and do not know whether or not it is Linux (as opposed to SunOS), use the "uname" command to determine the operating system.
2. Along with your source code, you must provide a **Makefile** that automatically compiles the simulator. This Makefile must create a simulator named "sim". The TAs should be able to type only "make" and the simulator will successfully compile. The TAs should be able to type only "make clean" to automatically remove object (.o) files and the simulator executable. An example Makefile will be posted on the web page, which you can copy and modify for your needs.

3. Your simulator must accept command-line arguments as follows:

```
sim <S> <N>
    <BLOCKSIZE> <L1_size> <L1_ASSOC> <L2_SIZE> <L2_ASSOC>
    <tracefile>
```

   - `S`: Scheduling Queue size.
   - `N`: Peak fetch, dispatch, and issue rate.
   - `BLOCKSIZE`: Positive integer. Block size in bytes. (Same block size for all caches in the memory hierarchy.)
   - `L1_SIZE`: Positive integer. L1 cache size in bytes.
   - `L1_ASSOC`: Positive integer. L1 set-associativity (1 is direct-mapped).
   - `L2_SIZE`: Positive integer. L2 cache size in bytes. L2_SIZE = 0 signifies that there is no L2 cache.
   - `L2_ASSOC`: Positive integer. L2 set-associativity (1 is direct-mapped).
   - `tracefile`: is the filename of the input trace.

   Note:
   BLOCKSIZE, L1_size, L1_ASSOC, L2_SIZE, and L2_ASSOC are all 0 if no data caches are modeled.

4. Your simulator must print outputs to the console (i.e., to the screen). This way, when a TA runs your simulator, he/she can simply redirect the output of your simulator to a filename of his/her choosing for validating the results.

7.3. Keeping backups

It is good practice to frequently make backups of all your project files, including source code, your report, etc. You can backup files to another hard drive (Eos account vs. home PC vs. laptop … keep consistent copies in multiple places) or removable media (Flash drive, etc.).

7.4. Run time of simulator

Correctness of your simulator is of paramount importance. That said, making your simulator *efficient* is also important for a couple of reasons.

First, the Tas need to test every student's simulator. Therefore, we are placing the constraint that your simulator must finish a single run in 2 minutes or less. If your simulator takes longer than 2 minutes to finish a single run, please see the Tas as they may be able to help you speed up your simulator.

Second, you will be running many experiments: many superscalar configurations (S, N) and multiple traces. Therefore, you will benefit from implementing a simulator that is reasonably fast.

One simple thing you can do to make your simulator run faster is to compile it with a high optimization level. The example Makefile posted on the web page includes the –O3 optimization flag.

Note that, when you are debugging your simulator in a debugger (such as gdb), it is recommended that you compile without –O3 and with –g. Optimization includes register allocation. Often, register-allocated variables are not displayed properly in debuggers, which is why you want to disable optimization when using a debugger. The –g flag tells the compiler to include symbols (variable names, etc.) in the compiled binary. The debugger needs this information to recognize variable names, function names, line numbers in the source code, etc. When you are done debugging, recompile with –O3 and without –g, to get the most efficient simulator again.

7.5. VCL Information

In addition to using *grendel* cluster, and other eos linux machines, VCL allows you to reserve linux machines. The following website describes how to use VCL:

http://vcl.ncsu.edu/

From the website:
1) select “Reservation System > Make a Reservation”,
2) select “NCSU WRAP” and “Proceed to Login”,
3) login,
4) for the environment, select “Linux Lab Machine (Realm RHEnterprise 4)” or “Linux Lab Machine (Realm RHEnterprise 5)”,
5) you may choose to reserve a shell for up to 4 hours beginning now or later (with the possibility to extend reservation before it expires),
6) click “Create Reservation” button,
7) wait until the screen updates with a reservation and click “Connect!”,
8) you will be given an internet address that you can ssh to, using putty or other ssh clients, the same way that you remotely and securely login to other linux machines.

8. Tasks, Grading Breakdown

8.1. VALIDATION

Your simulator must match the validation outputs that we will post on the project website. (1) The final measurements must match. (2) The timing information of every instruction must match.

8.2. RUNS

For each trace on the website, use your simulator as follows:

1. Graphs: For each benchmark, make a graph with IPC on the y-axis and S (Scheduling Queue size) on the x-axis. Use S = 8, 16, 32, 64, 128, and 256. Plot 4 different curves (lines) on the graph: one curve for each of N=1, N=2, N=4, and N=8.
2. Discussion: (a) Discuss trends for each benchmark individually. Give possible explanations for the trends. (b) Compare and contrast results from different benchmarks. Give possible explanations for differences among benchmarks.

   Sample discussion topics:

   - What happens to IPC as Scheduling Queue size (S) increases? (Does the answer depend on N?)
   - What happens to IPC as peak issue rate (N) increases? (Does the answer depend on S?)
   - What is the relationship between S and N? Explain…
   - Do some benchmarks show higher or lower IPC than other benchmarks, for the same microarchitecture configuration? Why might this be the case?

8.3. GRADING BREAKDOWN

0	You do not hand in anything by the due date.
+50	Your simulator does not compile, run, and work, but you hand in significant commented code.
+30	Your simulator matches the 2 validation outputs posted on the website. Additional 2 mystery runs will be made to check the correctness of your simulator.
+10	You ran all experiments outlined above and generated the graphs (*your simulator must be validated first*).
+10	You presented analysis and discussion of graphs as outlined above.
+10	Bonus part (optional): Your simulator matches the 2 validation outputs posted on the website.

9. What to Submit via Wolfware

You must hand in a single zip file called **project3.zip** that is no more than 1MB in size. Please respect the size limit on behalf of fellow students, as the Wolfware submission space is limited and exceeding your quota will cause problems come submission time. (Notify the TAs

beforehand if you have special space requirements. However, a zip file of 1MB should be sufficient.)

Below is an example showing how to create **project3.zip** from an Eos Linux machine. Suppose you have a bunch of source code files (*.cc, *.h), the Makefile, and your project report (report.doc).

```
zip project3 *.cc *.h Makefile report.doc
```

project3.zip must contain the following (any deviation from the following requirements may delay grading your project and may result in point deductions, late penalties, etc.):

1. **Project report**. This must be a single MS Word document named **report.doc** *OR* a single PDF document named **report.pdf**. The report must include the following:
    - A cover page with the project title, the Honor Pledge, and your full name as electronic signature of the Honor Pledge. A sample cover page will be posted on the project website.
    - All experiments, analysis, and discussion, as described in Section 8.
2. **Source code**. You must include the commented source code for the simulator program itself. You may use any number of .cc/.h files, .c/.h files, etc.
3. **Makefile**. See Section 7.2, item #2, for strict requirements. If you fail to meet these requirements, it may delay grading your project and may result in point deductions.

10. Penalties

Various deductions (out of 100 points):

-1 point for each hour late, according to Wolfware timestamp. **TIP**: Submit whatever you have completed by the deadline just to be safe. If, after the deadline, you want to continue working on the project to get more features working and/or finish experiments, you may submit a second version of your project late. If you choose to do this, the TA will grade both the on-time version and the late version (*late penalty applied to the late version*), and take the maximum score of the two. Hopefully you will complete everything by the deadline, however.

Up to -10 points for not complying with specific procedures. Follow all procedures very carefully to avoid penalties. Complete the **SUBMISSION CHECKLIST** that is posted on the project website, to make sure you have met all requirements.

Cheating: Source code that is flagged by tools available to us will be dealt with according to University Policy. This includes a 0 for the project and other disciplinary actions.