NC State University

Department of Electrical and Computer Engineering

ECE 463/563 (Prof. Rotenberg)

Project #3: Dynamic Instruction Scheduling (Version 1.0)

Due: Monday, December 2, 2024, 11:59 PM

1. Preliminary Information

1.1. Academic integrity

- Academic integrity:
  - **No collaboration whatsoever:** Each student must work individually on their project. There is zero tolerance for collaboration on the project in any form.
  - **Source code:** Each student must design and write their source code alone. They must do this (design and write their source code) without the assistance of any other person in ECE 463/563 or not in ECE 463/563. They must do this (design and write their source code) without searching the web for past semesters' projects and without searching the web for source code with similar goals (*e.g.*, modeling computer architecture components such as caches, predictors, pipelines, *etc*.), which is strictly forbidden. They must do this (design and write their source code) without looking at anyone else's source code, without obtaining electronic or printed copies of anyone else's source code, *etc*. Use of ChatGPT or other generative AI tools is prohibited.
  - **Explicit debugging:** With respect to "explicit debugging" as part of the coding process (*e.g.*, using a debugger, inspecting code for bugs, inserting prints in the code, iteratively applying fixes, *etc*.), each student must explicitly debug their code without the assistance of any other person in ECE 463/563 or not in ECE 463/563.
  - **Report:** Each student must run their own experiments, collect and process their own data, and write their own report. Plagiarism (lifting text, graphs, illustrations, *etc*., from someone else, whether one adjusts the originals or not) is prohibited. Using someone else's data (in raw form and/or graph form) is prohibited. Fabricating data is prohibited.
  - **Sanctions:** The sanctions for violating the academic integrity policy are (1) a score of 0 on the project and (2) academic integrity probation for a first-time offense or suspension for a subsequent offense (the latter sanctions are administered by the Office of Student Conduct). Note that, when it comes to academic integrity violations, both the giver and receiver of aid are responsible and both are sanctioned. Please see the following RAIV form which has links to various policies and procedures and gives a sense of how academic integrity violations are confronted, documented, and sanctioned: RAIV form.
  - **Enforcement:** The TAs will scan source code (from current and past semesters) through tools available to us for detecting cheating. The outputs from these tools, combined with in-depth manual analysis of these outputs, will be the basis for investigating suspected academic integrity violations. TAs will identify suspected plagiarism and/or data fabrication and these cases will be investigated.

- Reasonable assistance: If a student has any doubts or questions, or if a student is stumped by a bug, the student is encouraged to seek assistance using both of the following channels.
  - Students may receive assistance from the TAs and instructor.
  - Students are encouraged to post their doubts, questions, and obstacles, on the Moodle message board for this project. The instructor and TAs will moderate the message board to ensure that reasonable assistance is provided to the student. To strictly abide by the "no collaboration policy", students may not reply to other students' project queries unless the instructor solicits input from other students for a particular query.

    * An example of reasonable assistance via the message board: Student A: "*I'm encountering the following problem: I'm getting fewer writebacks from L2 to main memory than the validation runs. What might I be doing wrong?*" Instructor/TA: "*Maybe you have implemented evictions in more than one place in your code rather than consolidating it into one place (efficient and less bug-prone). If so, make sure you are checking for and counting evicted dirty blocks at all such locations (or try to implement evictions at one place in the code).*"

    * Another example of a reasonable exchange: Student A: "*I'm unsure how to split the address into tag and index, and how to discard the block offset bits. I've successfully computed the # bits for each but I am stuck on how to manipulate the address in C/C++. Do you have any advice?*" Instructor/TA: "*We suggest you use an unsigned integer type for the address and use bitwise manipulation, such as ANDs (&), ORs (|), and left/right shifts (<<. >>) to extract values from the unsigned integer, the same as one would do in Verilog.*"

    * Another example of a reasonable exchange: Student A: "*I'm unsure how to dynamically allocate memory, such as dynamically allocating a 1D array of structs/classes or (more appropriately for a cache) a 2D array of structs/classes. Can you point me to some references on this?*" Instructor/TA: "*Sure, here is a web site or reference book that discusses dynamic memory allocation including 1D and 2D arrays.*"

1.2. Same project scope for both ECE 463 and ECE 563 students

Unlike previous projects, the scope of this project is the same for both ECE 463 and ECE 563 students. This is because the OOO pipeline must be modeled in its entirety to measure cycles to execute a trace. Note that the project focuses on modeling data dependencies (only through registers), pipeline stages, and structural hazards (Issue Queue and Reorder Buffer). Therefore, we assume perfect branch prediction and perfect caches, and ignore memory dependencies: you will NOT integrate a BTB, conditional branch predictor, instruction cache/TLB, data cache/TLB, or Load Queue/Store Queue.

1.3. Programming languages for this project

You must implement your project using the C, C++, or Java languages, for two reasons. First, these languages are preferred for computer architecture performance modeling. Second, our Gradescope autograder only supports compilation of these languages.

1.4. Responsibility for self-grading your project via Gradescope

<u>You will submit, validate, and SELF-GRADE your project via Gradescope; the TAs will only manually grade the report. While you are developing your simulator, you are required to frequently check via Gradescope that your code compiles, runs, and gives expected outputs with respect to your current progress. This is necessary to resolve porting issues in a timely fashion (*i.e.*, well before the deadline), caused by different compiler versions in your programming environment and the Gradescope backend. This is also necessary to resolve non-compliance</u>

issues (*i.e.*, how you specify the simulator's command-line arguments, how you format the simulator's outputs, *etc.*) in a timely fashion (*i.e.*, well before the deadline).

2. Project Description

In this project, you will construct a simulator for an out-of-order superscalar processor that fetches 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.

3. Inputs to Simulator

3.1. Traces

The simulator reads a trace file in the following format:

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

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 66.
- `<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 66.
- `<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 66.

For example:

```
ab120024  0   1  2  3
ab120028  1   4  1  3
ab12002c  2  -1  4  7
```

Means:

"operation type 0" R1, R2, R3
"operation type 1" R4, R1, R3
"operation type 2" -, R4, R7 // *no destination register!*

Traces are posted on the Moodle website.

3.2. Command-line arguments to the simulator

The simulator executable built by your Makefile must be named "sim" (the Makefile is discussed

in Section 6).

Your simulator must accept command-line arguments as follows:

```
sim <ROB_SIZE> <IQ_SIZE> <WIDTH> <tracefile>
```

The parameters <ROB_SIZE>, <IQ_SIZE>, and <WIDTH>, are explained in Section 5. `<tracefile>` is the filename of the input trace.

4. Outputs from Simulator

The simulator first outputs the timing information for each dynamic instruction in program order, followed by final outputs (simulator command, processor configuration, and simulation results).

See Section 6 regarding the formatting of these outputs and validating your simulator.

4.1. Timing information for each dynamic instruction

The simulator outputs the timing information for each dynamic instruction in the trace, in program order (*i.e.*, in the same order that instructions appear in the trace). The per-instruction timing information is output in the following format:

```
<seq_no> fu{<op_type>} src{<src1>,<src2>} dst{<dst>}
FE{<begin-cycle>,<duration>} DE{…} RN{…} RR{…} DI{…} IS{…} EX{…}
WB{…} RT{…}
```

`<seq_no>` is the line number in trace (*i.e.*, the dynamic instruction count), 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 pipeline stages, indicate the first cycle that the instruction was in that pipeline stage followed by the number of cycles the instruction was in that pipeline stage.

Here is an example instruction from one of the validation runs.

```
5 fu{2} src{15,-1} dst{16} FE{5,1} DE{6,1} RN{7,1} RR{8,1}
DI{9,1} IS{10,3} EX{13,5} WB{18,1} RT{19,1}
```

Notice that the begin-cycle of a given pipeline stage equals the begin-cycle of the immediately preceding pipeline stage plus the number of cycles spent in the immediately preceding pipeline stage. For example, the instruction's first cycle in the **`EX`** stage is cycle 13, which is the first cycle in **`IS`** (10) plus the number of cycles spent in **`IS`** (3).

4.2. Final outputs

The simulator outputs the following after completion of the run:

1. Simulator command.
2. Processor configuration.

3. Simulation results:
    a. Dynamic instruction count. (Total number of retired instructions.)
    b. Cycles. (Total number of cycles to retire all instructions.)
    c. Instructions per cycle (IPC). (item a divided by item b, above)

5. Simulator Specification

5.1. Microarchitecture to be Modeled

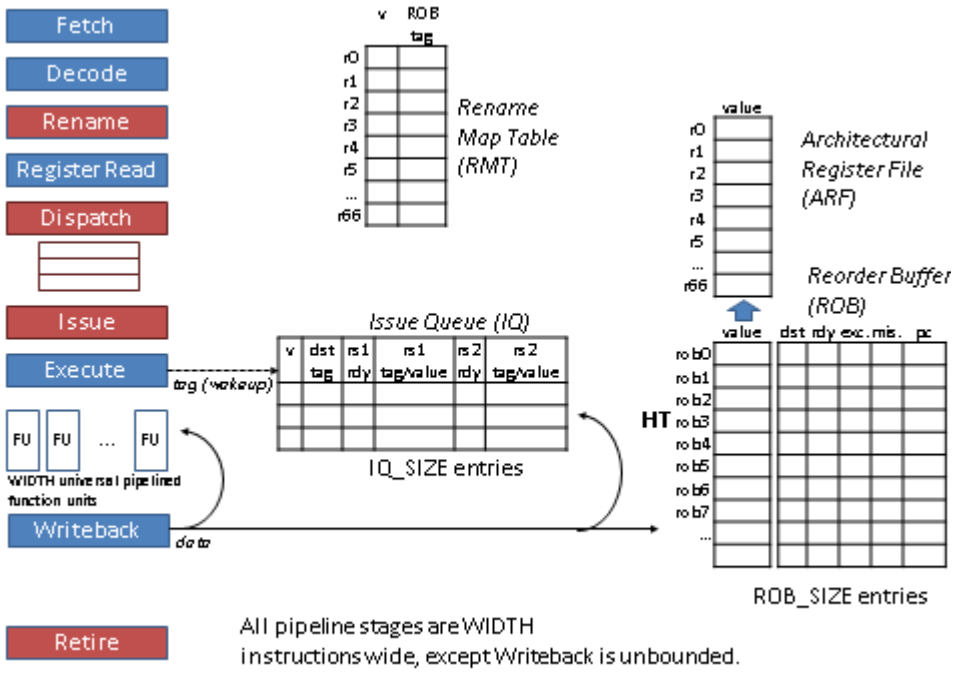

Figure 1. Overview of microarchitecture to be modeled, including the terminology and parameters used throughout this specification.

Parameters:

1. Number of architectural registers: The number of architectural registers specified in the ISA is 67 (r0-r66).[1] The number of architectural registers determines the number of entries in the Rename Map Table (RMT) and Architectural Register File (ARF).
2. WIDTH: This is the superscalar width of all pipeline stages, in terms of the maximum number of instructions in each pipeline stage. The one exception is Writeback: the number of instructions that may complete execution in a given cycle is not limited to WIDTH (this is explained below).
3. IQ_SIZE: This is the number of entries in the Issue Queue (IQ).
4. ROB_SIZE: This is the number of entries in the Reorder Buffer (ROB).

Function units:

There are WIDTH universal pipelined function units (FUs). Each FU can execute any type of instruction (hence the term "universal"). 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

[1] The ISA is MIPS-like: 32 integer registers, 32 floating-point registers, the HI and LO registers (for results of integer multiplication/divide), and the FCC register (floating-point condition code register).

has a latency of 5 cycles. Each FU is fully pipelined. Therefore, a new instruction can begin execution on a FU every cycle.

Pipeline registers:
The pipeline stages shown in Figure 1 are separated by pipeline registers. In general, this spec names a pipeline register based on the stage that it feeds into. For example, the pipeline register between Fetch and Decode is called DE because it feeds into Decode. A "bundle" is the set of instructions in a pipeline register. For example, if DE is not empty, it contains a "decode bundle".
Table 1 lists the names of the pipeline registers used in this spec. It also provides a description of each pipeline register and its size (max # instructions).

Table 1. Names, descriptions, and sizes of all of the pipeline registers.

Pipeline Register	Description	Size (max # instructions)
DE	pipeline register between the Fetch and Decode stages	WIDTH
RN	pipeline register between the Decode and Rename stages	WIDTH
RR	pipeline register between the Rename and Register Read stages	WIDTH
DI	pipeline register between the Register Read and Dispatch stages	WIDTH
IQ	Queue between the Dispatch and Issue stages	IQ_SIZE
execute_list	execute_list represents the pipeline register between the Issue and Execute stages, as well as all sub-pipeline stages within each function unit.	WIDTH*5 There are WIDTH universal function units each with a maximum latency of 5 cycles. Hence, there can be as many as WIDTH*5 instructions in-flight within the Execute stage.
WB	pipeline register between the Execute and Writeback stages *To maintain a non-blocking Execute stage, there is no constraint on the number of instructions that may complete in a cycle.*	WIDTH*5 This is a conservative upper bound. If each function unit's 5-stage pipeline is full with a 1-cycle (youngest), 2-cycle, 3-cycle, 4-cycle, and 5-cycle (oldest) operation, then all 5 instructions will complete in the same cycle. Multiply that by the number of such function units (WIDTH).
ROB	Queue between the Writeback and Retire stages	ROB_SIZE

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 Architectural Register File (ARF) 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 dependencies).

5.2. Guide to Implementing your Simulator

This section provides a guide to implementing your simulator.

Call each pipeline stage in reverse order in your main simulator loop, as follows. The comments indicate tasks to be performed.

```
do {
     Retire();       // Retire up to WIDTH consecutive
                     // "ready" instructions from the head of
                     // the ROB.

     Writeback();    // Process the writeback bundle in WB:
                     // For each instruction in WB, mark the
                     // instruction as "ready" in its entry in
                     // the ROB.

     Execute();      // From the execute_list, check for
                     // instructions that are finishing
                     // execution this cycle, and:
                     // 1) Remove the instruction from
                     //    the execute_list.
                     // 2) Add the instruction to WB.
                     // 3) Wakeup dependent instructions (set
                     //    their source operand ready flags) in
                     //    the IQ, DI (the dispatch bundle), and
                     //    RR (the register-read bundle).

     Issue();        // Issue up to WIDTH oldest instructions
                     // from the IQ. (One approach to implement
                     // oldest-first issuing, is to make multiple
                     // passes through the IQ, each time finding
                     // the next oldest ready instruction and
                     // then issuing it. One way to annotate the
                     // age of an instruction is to assign an
                     // incrementing sequence number to each
                     // instruction as it is fetched from the
                     // trace file.)
```

```
                // To issue an instruction:
                // 1) Remove the instruction from the IQ.
                // 2) Add the instruction to the
                //    execute_list. Set a timer for the
                //    instruction in the execute_list that
                //    will allow you to model its execution
                //    latency.

Dispatch();     // If DI contains a dispatch bundle:
                // If the number of free IQ entries is less
                // than the size of the dispatch bundle in
                // DI, then do nothing. If the number of
                // free IQ entries is greater than or equal
                // to the size of the dispatch bundle in DI,
                // then dispatch all instructions from DI to
                // the IQ.

RegRead();      // If RR contains a register-read bundle:
                // If DI is not empty (cannot accept a
                // new dispatch bundle), then do nothing.
                // If DI is empty (can accept a new dispatch
                // bundle), then process (see below) the
                // register-read bundle and advance it from
                // RR to DI.
                //
                // Since values are not explicitly modeled,
                // the sole purpose of the Register Read
                // stage is to ascertain the readiness of
                // the renamed source operands. Apply your
                // learning from the class lectures/notes on
                // this topic.
                //
                // Also take care that producers in their
                // last cycle of execution wakeup dependent
                // operands not just in the IQ, but also in
                // two other stages including RegRead()
                // (this is required to avoid deadlock). See
                // Execute() description above.

Rename();       // If RN contains a rename bundle:
                // If either RR is not empty (cannot accept
                // a new register-read bundle) or the ROB
                // does not have enough free entries to
                // accept the entire rename bundle, then do
                // nothing.
                // If RR is empty (can accept a new
                // register-read bundle) and the ROB has
```

```
                    // enough free entries to accept the entire
                    // rename bundle, then process (see below)
                    // the rename bundle and advance it from
                    // RN to RR.
                    //
                    // Apply your learning from the class
                    // lectures/notes on the steps for renaming:
                    // (1) allocate an entry in the ROB for the
                    // instruction, (2) rename its source
                    // registers, and (3) rename its destination
                    // register (if it has one). Note that the
                    // rename bundle must be renamed in program
                    // order (fortunately the instructions in
                    // the rename bundle are in program order).

    Decode();       // If DE contains a decode bundle:
                    // If RN is not empty (cannot accept a new
                    // rename bundle), then do nothing.
                    // If RN is empty (can accept a new rename
                    // bundle), then advance the decode bundle
                    // from DE to RN.

    Fetch();        // Do nothing if either (1) there are no
                    // more instructions in the trace file or
                    // (2) DE is not empty (cannot accept a new
                    // decode bundle).
                    //
                    // If there are more instructions in the
                    // trace file and if DE is empty (can accept
                    // a new decode bundle), then fetch up to
                    // WIDTH instructions from the trace file
                    // into DE. Fewer than WIDTH instructions
                    // will be fetched only if the trace file
                    // has fewer than WIDTH instructions left.

} while (Advance_Cycle());
                    // Advance_Cycle performs several functions.
                    First, it advances the simulator cycle.
                    Second, when it becomes known that the
                    pipeline is empty AND the trace is depleted,
                    the function returns “false” to terminate
                    the loop.
```

6. Submit, Validate, and Self-Grade with Gradescope

Sample simulation outputs are provided on the Moodle site. These are called "validation runs". **Refer to the validation runs to see how to format the outputs of your simulator.**

You must submit, validate, and self-grade[2] your project using Gradescope. Here is how Gradescope (1) receives your project (zip file), (2) compiles your simulator (Makefile), and (3) runs and checks your simulator (arguments, print-to-console requirement, and "diff -iw"):

1. **How Gradescope receives your project: zip file.** While you are developing your simulator, you may continuously submit new zip files to Gradescope containing the latest version of your project. The latest submission is the one that is considered for your grade. Gradescope will accept a zip file consisting of three things: your source code, a Makefile to compile your source code, and your project report. In the early stages of your project, before creating the report, your zip file will have only source code and a Makefile. Once the report is completed, your zip file will contain everything.
    1. Report (included in the zip file once available): The report must be a PDF file named "report.pdf" located at the top level of the zip file, because that is what Gradescope looks for when checking completeness of the submission. *See Section 7 and the required report template in Moodle for the required contents of the report.*
    - Makefile: The Makefile must be at the top level of the zip file, because Gradescope runs "make" with the expectation that the Makefile is at the top level.
    - Source code: Whether your source code is at the top level of the zip file or in directories below the top level, your Makefile must be designed to compile your source code, accordingly.

2. **How Gradescope compiles your simulator: Makefile.** Along with your source code, you must provide a Makefile that automatically compiles the simulator. This Makefile must create a simulator named "sim". An example Makefile is posted on the Moodle site, which you can copy and modify for your needs.

3. **How Gradescope runs and checks your simulator: arguments, print-to-console requirement, "diff -iw", and timeout.**
    - Your simulator executable (created by your Makefile) must be named "sim" and take command-line arguments in the manner specified in Section 3.2, because Gradescope assumes these things.
    - Your simulator must print outputs to the console (*i.e.*, to the screen), because Gradescope assumes this.
    - Your output must match the validation runs both numerically and in terms of formatting, because Gradescope runs "diff -iw" to compare your output with the correct output. 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

[2] The mystery runs component of your grade will not be published until we release it. The report will be manually graded by the TAs.

runs have whitespace. Note, however, that extra or missing blank lines are NOT ok: “diff -iw” does not ignore extra or missing blank lines.

- Gradescope’s autograder has a timeout for compiling the simulator and running all tests. The default timeout is 10 minutes. This is usually ample time for this course. If the autograder times out for your project (very inefficient simulator or a bug that causes deadlock), you will see a grade of zero for that submission. Please see Section 9.2 regarding optimizing run time. Also seek advice from the instructor and TAs, as needed.

7. Grading Breakdown, Experiments, and Report

See the required report template in Moodle for the grading breakdown, experiments, and report contents. Use the report template as the basis for the report that you submit (insert graphs, fill in answers to questions, *etc*.).

8. Penalties

Various deductions (out of 100 points):

-1 point for each day (24-hour period) late, according to the Gradescope timestamp. The late penalty is pro-rated on an hourly basis: -1/24 point for each hour late. We will use the "ceiling" function of the lateness time to get to the next higher hour, *e.g.*, ceiling(10 min. late) = 1 hour late, ceiling(1 hr, 10 min. late) = 2 hours late, and so forth. **For this third and final project, Gradescope will accept late submissions no more than one week after the deadline. The goal of this policy is to allow adequate time for the TAs to grade reports and assess partial credit for simulator development effort (for simulators that don't match any validation runs), before final grades are due for the semester.**

See Section 1.1 for penalties and sanctions for academic integrity violations.

9. Advice on backups and run time

9.1. 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 (your NFS B: drive in your NCSU account, home PC, laptop … keep consistent copies in multiple places) or removable media (flash drive, *etc*.).

9.2. Run time of simulator

Correctness of your simulator is of paramount importance. That said, making your simulator *efficient* is also important because you will be running many experiments: many superscalar processor configurations 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 Moodle site 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.

As mentioned in Section 6, another reason for being wary of excessive run times is Gradescope's autograder timeout.

10. Scope Tool: An Optional Visualization Aid

I have written a tool that allows you to display instruction schedules identical to the ones drawn in class. You may use this tool as an optional visualization aid.

- Download the scope tool from the Moodle website.
- Run your simulator and redirect its output to some filename, `<scope-input-file>`.
- Usage: `scope <scope-input-file> <scope-output-file>`
- `<scope-output-file>`: This contains the instruction schedule. It is best displayed in a web browser because *some* web browsers provide horizontal scrollbars when lines are wider than your screen.

Go to the Moodle website to view an example.