NC State University

Department of Electrical and Computer Engineering

ECE 463/563: Fall 2018 (Rotenberg)

Project #3: Dynamic Instruction Scheduling (Version 1.0)

Due: Friday, December 14, 5:00 PM

1. Groundrules

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: 0 on the project and referral to the Office of Student Conduct for sanctions.
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. It is recommended that you do all your work in the C, C++ or Java languages. Exceptions must be approved by the instructor.
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. 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 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

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:

- o `<PC>` is the program counter of the instruction (in hex).
- o `<operation type>` is either “0”, “1”, or “2”.
- o `<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.
- o `<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.
- o `<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 wolfware website.

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 retired 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.

5. Simulator Specification

5.1. Microarchitecture to be Modeled

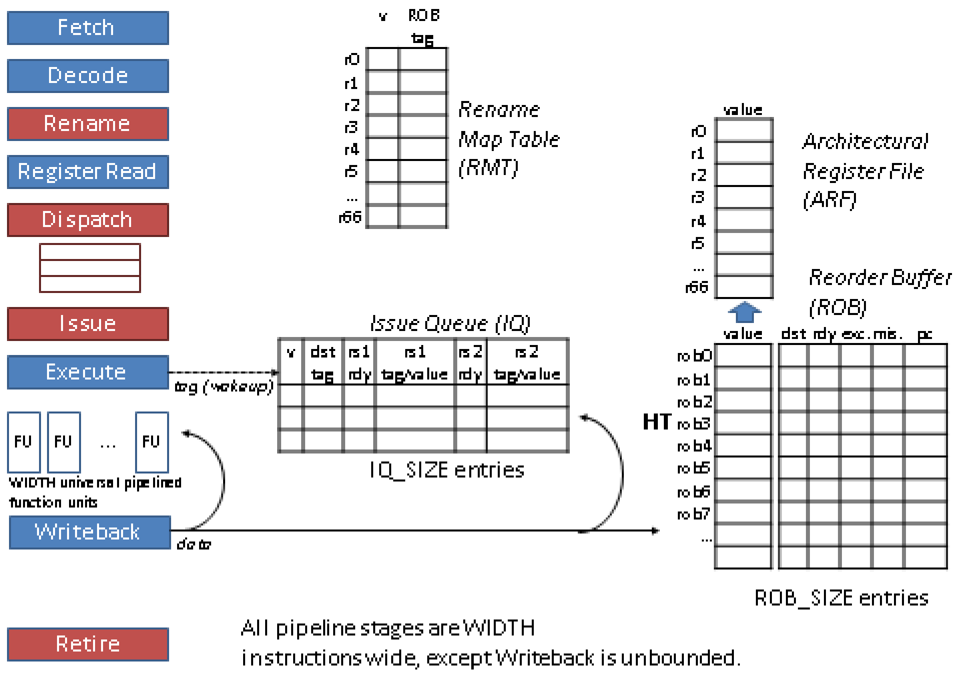

Figure 1. Overview of microarchitecture to be modeled, including the terminology and parameters used throughout this specification.

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. Helping you Debug and Validate: A Scope Tool

I have written a tool that allows you to display pipeline timing diagrams identical to the timing diagrams drawn in class.

- Download the tool from the wolfware website.

- 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 or tell you to email me.

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>}
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. The tool automatically does some consistency checks and exits if there is a problem, *e.g.*, begin cycle of DE should equal begin cycle of FE plus duration of FE.

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.

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 <u>numerically</u> and in terms of <u>formatting</u>, 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.
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 <ROB_SIZE> <IQ_SIZE> <WIDTH> <tracefile>
```

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 (ROB_SIZE, IQ_SIZE, WIDTH) 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. Test your simulator on Eos linux machines

You must test your simulator on Eos linux machines such as: *remote.eos.ncsu.edu* and *grendel.ece.ncsu.edu*.

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

8.2.1. Large ROB, effect of IQ_SIZE

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

1. **Graphs [5 points]:** Keep ROB_SIZE fixed at 512 entries so that it is not a resource bottleneck. For each benchmark, make a graph with IPC on the y-axis and IQ_SIZE on the x-axis. Use IQ_SIZE = 8, 16, 32, 64, 128, and 256. Plot 4 different curves (lines) on the graph: one curve for each of WIDTH = 1, 2, 4, and 8.
2. **Graph Analysis [5 points]:** Using the data in the graph, for each WIDTH (1, 2, 4, and 8), find the minimum IQ_SIZE that still achieves within 5% of the IPC of the largest IQ_SIZE (256). This exercise should give four optimized IQ_SIZE's per benchmark, one optimized for each of WIDTH = 1, 2, 4, and 8. Tabulate the results of this exercise as follows:

	"Optimized IQ_SIZE per WIDTH" Minimum IQ_SIZE that still achieves within 5% of the IPC of the largest IQ_SIZE		
	Benchmark 1	Benchmark 2	…
WIDTH = 1			
WIDTH = 2			
WIDTH = 4			
WIDTH = 8			

3. **Discussion [5 points]:**
   - The goal of a superscalar processor is to achieve an IPC that is close to WIDTH (which is the peak theoretical IPC of the processor). Given this goal, what is the relationship between WIDTH and IQ_SIZE? Explain.
   - Do some benchmarks show higher or lower IPC than other benchmarks, for the same microarchitecture configuration? Why might this be the case?

8.2.2. Effect of ROB_SIZE

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

1. **Graphs [5 points]:** For each benchmark, make a graph with IPC on the y-axis and ROB_SIZE on the x-axis. Use ROB_SIZE = 32, 64, 128, 256, and 512. Plot 4 different curves (lines) on the graph: one curve for each of WIDTH = 1, 2, 4, and 8. For a given WIDTH, use the optimized IQ_SIZE for that WIDTH, as obtained from the table in Section 8.2.1.

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 validation outputs posted on the website. Additional mystery runs will be made to check the correctness of your simulator.
+20	Report (graphs, analysis, discussion). **(*your simulator must be validated first*)**

9. What to Submit via Wolfware

You must hand in a single zip file called **project3.zip**.

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.pdf).

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

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 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 referral to the Office of Student Conduct.