CDA 5106: Spring 2022
Machine Problem 3: Dynamic Instruction Scheduling

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 through various tools available to us for detecting cheating. Source code that is flagged by these tools will be dealt with severely.
3. A Webcourses discussion will be created for posting questions, discussing and debating issues, and making clarifications. It is an essential aspect of the project and communal learning. Never post actual code on the discussion thread. 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 Linux environment is required. This is the platform where the TAs will compile and test your simulator.

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.

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

- <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.
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!

4. Output 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.

5. Simulator Specification

5.1 Microarchitecture to be Modeled

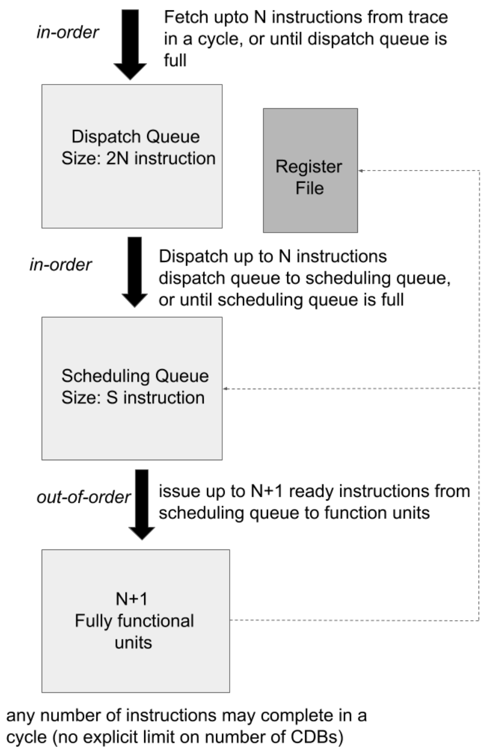
in-order
Fetch upto N instructions from trace in a cycle, or until dispatch queue is full
Dispatch Queue Size: 2N instruction
Register File
in-order
Dispatch up to N instructions dispatch queue to scheduling queue, or until scheduling queue is full
Scheduling Queue Size: S instruction
out-of-order
issue up to N+1 ready instructions from scheduling queue to function units
N+1 Fully functional units
any number of instructions may complete in a cycle (no explicit limit on number of CDBs)

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

5.2 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.

5.3 Superscalar microarchitecture

The microarchitecture is N-way superscalar. This means that up to N instructions can be fetched, dispatched, and up to N+1 instructions can be 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+1 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+1 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.

5.4 Fully Pipelined Functional Units

Students will implement only the fully pipelined architecture. The number of FUs is N+1. 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).

5.5 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).

5.6 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.

6. 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, we 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 a matter of convenience and efficiency for simulator implementation. 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. Since we aren't really modeling a ROB, make the fake-ROB large enough that it never overflows – we 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+1 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.
                                        //It advances the simulator
                                        //cycle. Also, when it becomes known that the
                                        //fake-ROB is empty AND the trace is depleted,
                                        //the function returns "false" to terminate
                                        //the loop.
```

7. Validation and Other Requirements

7.1 Validation Requirements

Sample simulation outputs are provided, and they 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.
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).

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 linux 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.

No debug runs will be provided for this project, the validation runs contain enough debug information.

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.

1. You must be able to compile and run your simulator on 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.
3. Your simulator must accept command-line arguments as follows:

   sim <S> <N> <tracefile>

   ...where <S> is the Scheduling Queue size, <N> is the peak fetch and dispatch rate, issue rate will be up to N+1 and <tracefile> is the filename of the input trace.
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 Validation File Format

The validation file names are of the following format:
pipe_<S>_<N>_<trace >.txt
For Example
pipe_2_8_gcc.txt

The timing information in the validation files are provided as follows:
<tag> fu{<operation type>} src{<src1 reg#>,<src2 reg#>} dst{<dest reg#>} IF{<cycle>,<duration>} ID{<cycle>,<duration>} IS{<cycle>,<duration>} EX{<cycle>,<duration>} WB{<cycle>,<duration>}
For Example
```
0 fu{0} src{29,14} dst{-1} IF{0,1} ID{1,1} IS{2,1} EX{3,1} WB{4,1}
1 fu{2} src{29,-1} dst{14} IF{0,1} ID{1,1} IS{2,1} EX{3,5} WB{8,1}
```
Etc.

8. Tasks, Grading Breakdown

8.1 Validation

Your simulator must match the validation outputs that are posted. (1) The final measurements must match. (2) The timing information of every instruction must match.

8.2 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.
+25	Your simulator matches the validation outputs posted on Webcourses.
+25	Your simulator matches the mystery runs outputs. Mystery runs will be made to check the correctness of your simulator.

9. What to Submit via Webcourses

You must hand in a single zip file called project3.zip.
Below is an example showing how to create project3.zip from a Linux machine.
Suppose you have a bunch of source code files (*.cc, *.h) and the Makefile.
zip project3 *.cc *.h Makefile

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. 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.
2. 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. Submission Checklist for Machine Problem3

Requirement	**Section**
Your simulator should output total number of instructions, total number of cycles and IPC along with the timing information for every instruction in the required format.	Section 4

Your simulator output should match the validation files provided exactly. Check by doing diff -iw <your_output><trace provoded>	Section 7.1
Your simulator must compile without errors.	-
Your simulator must accept command-line arguments as follows: sim <S> <N> <tracefile>	Section 7.2
Your simulator must print outputs to the console	Section 7.2
Submit all of your source code	Section 9
Your source code and Makefile should be zipped into project3.zip	Section 9