IIT Madras
Department of Computer Science and Engineering

CS6600: July-Nov '24
Project 3: Dynamic Scheduling
Due: Sunday, November 10 at 11:59 PM

1. Ground rules

1. All students must work alone for this project.
2. Sharing of code between students and/or copying code from public git repositories is considered cheating. The TAs will scan source code through various tools available to us for detecting cheating. Source code that is flagged by these tools will receive the following actions:
   - Zero credits for the project.
   - Final grade will be two grades lower than the actual grade obtained by the student.
3. Students are encouraged to engage in white board discussions, which is an essential aspect of the project.
4. It is recommended that you do all your work in the C or C++ languages. Exceptions (in rare cases) must be approved by the faculty.
5. Students must get the code to work on the provided Docker.

2. Project Description

In this project, you will construct a simulator for an out-of-order superscalar processor based on Tomasulo's algorithm (discussed in class) that fetches, dispatches, and issues N instructions per cycle. The primary goal is to only model the dynamic scheduling mechanism in detail. Hence, 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, which is specified in hexadecimal.
- `<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. 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.

Example trace:

```
bc020064  0   1  2  3
bc020068  1   4  1  3
bc02006c  2  -1  4  7
```

Means:
"operation type 0" R1, R2, R3
"operation type 1" R4, R1, R3
"operation type 2" -, R4, R7

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).

In addition, the simulator should also print the timing information for every instruction in the trace.

5. Specification of the Simulator

5.1 Superscalar Microarchitecture

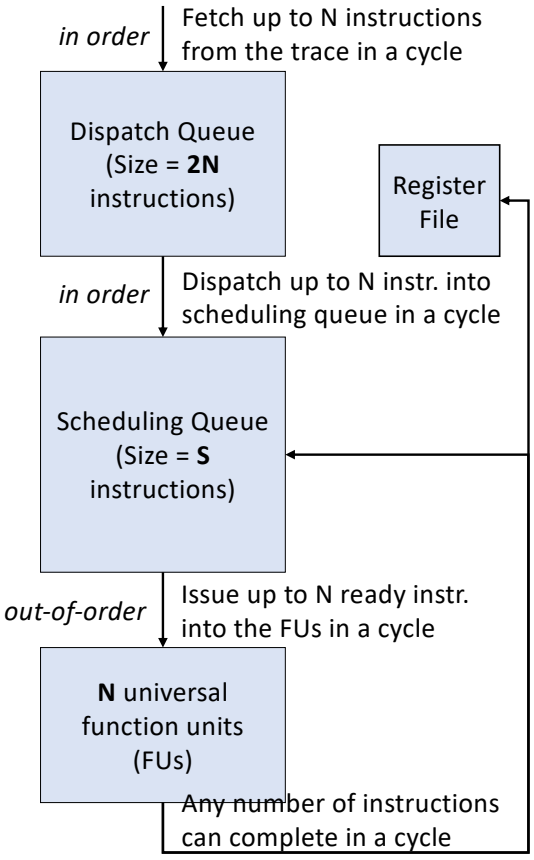

Figure 1. Overview of the superscalar microarchitecture to be modeled.

The microarchitecture is N-way superscalar. This means that up to N instructions can be fetched, dispatched, and issued each cycle, as shown in Fig. 1. However, in a given cycle, fewer than N instructions may be fetched, dispatched, or issued, due structural hazards or data dependences.

Fetch:
In a given cycle, up to N instructions are fetched into the Dispatch Queue. Fetching stalls if either the Dispatch Queue is full or the last instruction in the trace has been fetched. The Dispatch Queue is 2N instructions in size.

Dispatch:
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. In a simulator, a simple way to generate tags for instructions is to assign increasing sequence of numbers. For example, you can 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.

Issue:
In this microarchitecture, there is one centralized pool of reservation stations, called the Scheduling Queue, which is S instructions in size. As instructions in the Scheduling Queue become ready, they are possible 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). An instruction must be removed from the Scheduling Queue when it issues to a Function Unit.

Execute:
There are N Function Units (FUs), each of which can execute any type of instruction, and hence, is referred to as a *universal function unit*. 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*.
- Type 2 has a latency of *10 cycles*.

Assume that each FU is fully pipelined. Therefore, a new instruction can begin execution on a FU every cycle. Due to different execution latencies, it is possible for more than N instructions to complete in the same cycle. You do not have to limit the number of instructions that can complete in a given cycle. 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 update the state if needed).

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 produce actual register values. All that the simulator needs is the microarchitecture configuration, execution latencies of instructions, and register

specifiers of instructions (to determine true, anti, and output dependences). Hence, the initial Register File values and the instruction opcodes are omitted from the trace.

Imprecise interrupts:
The microarchitecture does not maintain precise interrupts. Hence, instructions can update the Register File as soon as they complete.

5.2 Implementation Guide

It is recommended that your simulator is organized as follows.

1. Define 5 states that an instruction can be in (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 is NOT used by the simulator to model in-order retirement. Instead, this ROB is a matter of convenience for simulator implementation. It 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 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, tag, etc. An instruction is added to the ROB when it is fetched from the trace and removed when it has reached the WB state and all prior instructions have been removed. Ensure that the ROB is large enough and that it never overflows since it is intended to only serve as a bookkeeping data structure.

3. Define 3 lists:
    - **dispatch_list:** This contains a list of instructions in either the IF or ID state. The *dispatch_list* models the Dispatch Queue in Fig. 1. By including both the IF and ID states, we don't need to separately model the pipeline registers of the fetch and the dispatch stages.
    - **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 in Fig. 1.
    - **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 in Fig. 1.

4. Call each pipeline stage in reverse order in your main simulator loop, as follows. The order among these tasks is important.

```
do {
    Retire();
    Execute();
    Issue();
    Dispatch();
    Fetch();
} while (Advance_Cycle());
```

- **Retire():** Remove instructions from the head of the 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 perform the following tasks:
  a) Remove the instruction from the *execute_list*.
  b) Transition from EX state to WB state.
  c) Update the register file state (e.g., ready flag) and wakeup dependent instructions (by setting their operand ready flags).

- **Issue():** From the *issue_list*, construct a temporary list of instructions whose operands are ready. These are referred to as the READY instructions. Scan the READY instructions in ascending order of tags and issue up to N of them. Perform the following tasks to issue an instruction:
  a) Remove the instruction from the *issue_list* and add it to the *execute_list*.
  b) Transition from the IS state to the EX state.
  c) 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 since 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:
  a) Remove the instruction from the *dispatch_list* and add it to the *issue_list*.
  b) Transition from the ID state to the IS state.
  c) Rename source operands by looking up state in the register file.
  d) Rename destination operands by updating state in the register file.

- **Fetch():** Read new instructions from the trace as long as you have not reached the end-of-file, the fetch bandwidth is not exceeded, and the dispatch queue is not full. Then, for each incoming instruction:
  a) Push the new instruction onto the ROB. Initialize the instruction's data structure, including setting its state to IF.
  b) Add the instruction to the *dispatch_list*.

- **Advance_Cycle():** It advances the simulation cycle. Finally, when it becomes known that the ROB is empty and the trace is depleted, the function returns "false" to terminate the loop.

6. Validation Requirements

Sample simulation outputs will be provided to validate the correctness of your simulator. These are called validation runs. You must run your simulator and debug it until it passes all the given validation runs.

Each validation run includes:

1. Timing information for every instruction. The format is described below.

2. The microarchitecture configuration.
3. All measurements as described in Section 4.

Your simulator output must match both numerically and in terms of formatting, because the TAs will only *diff* your simulator's output with the correct output. You must confirm the 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> <provided_validation_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.

6.1 Simulator Compilation and Execution Requirements

You will hand in source code, and the TAs will compile and run your simulator. You must meet the following requirements without fail:

1. You must be able to compile and run your simulator on the provided Docker.
2. You must provide a *Makefile* that automatically compiles the simulator. This *Makefile* must create a simulator named "*ooosim*". The TAs should be able to type only "*make*" and the simulator will successfully compile. An example *Makefile* will be provided to you. Please feel free to modify it based on your needs.
3. Your simulator must accept the following command-line arguments in the specified order.
   ./ooosim <N> <S> <trace_file>
   - N : Superscalar bandwidth.
   - S : Schedule queue size.
   - trace_file : Character string specifying the full name of trace file.

6.2 Compiling and Running the Simulator on the Docker

The following are the steps to be followed to compile and run the simulator on Docker.

Step 1 : Install Docker. For example, if you are installing on Ubuntu based machine, you can follow this: https://docs.docker.com/engine/install/ubuntu/. Docker desktop can also be used; in the latest version it comes with an integrated terminal in which the following commands can be run.

Step 2 : Build the docker image. For this, go to the folder containing Dockerfile and run:
$ docker build -t assign3 .
Where, *assign3* is the name for the docker image.

Step 3 : Create a container and run. The command for it is:
$ docker run -it assign3 /bin/bash

Step 4 : Now, we will get a shell inside the docker container. If you run make, it should build the simulation executable.

$ cd Assignment_files
$ make
$ ./ooosim 8 64 gcc_trace.txt

Note: Docker container is a temporary instance, so any changes made is not saved.

7. Project Experiments and Report

Plots:

For each trace (provided to you), plot IPC on the y axis and Scheduling Queue size (S) on the x axis for S = 8, 16, 32, 64, 128, and 256. Produce a curve each for N = 2, 4, and 8. Clearly label the axes and display the legend.

Discussion:

1. Explain the trend in IPC as the Scheduling Queue size (S) increases.
2. Explain the trend in IPC as the peak issue bandwidth (N) increases.
3. Describe the interaction between S and N.

8. Submission Requirements

You must submit a single zip file, named <StudentID>_assignment3.zip, which should contain the following:

- **Source code.** You must include all the source code files (.cc/.h or .c/.h)
- **Makefile.** The Makefile needs to be updated to compile all your source code files.
- **Project report.** This should be a single *report.doc* or *report.pdf* file, which contains the plots and the discussions.

9. Grading Policy

The grading policy for this project is described in the following table (Table 1).

Description	Points
Substantial programming effort that results in a simulator that builds, but reports inaccurate statistics.	30
Working simulator that reports correct statistics and timing information for the validation runs (provided *a priori*)	30
Working simulator that reports correct statistics and timing information for the mystery runs (created by the TAs)	20
Project report	20

Table 1. Grading policy for the dynamic scheduling simulator project.