Georgia Institute of Technology

CS4290/CS6290/ECE4100/ECE6100

Assignment #1
Program due: Sunday (1/26) 11:55 pm   (with 50% late penalty, Tuesday 1/28 3pm)
Submitted online at T-square
Prof. Moinuddin Qureshi, Instructor

This is an individual assignment. You can discuss this assignment with other classmates but you should code your assignment individually. You are NOT allowed to see the code of (or show your code to) other students.

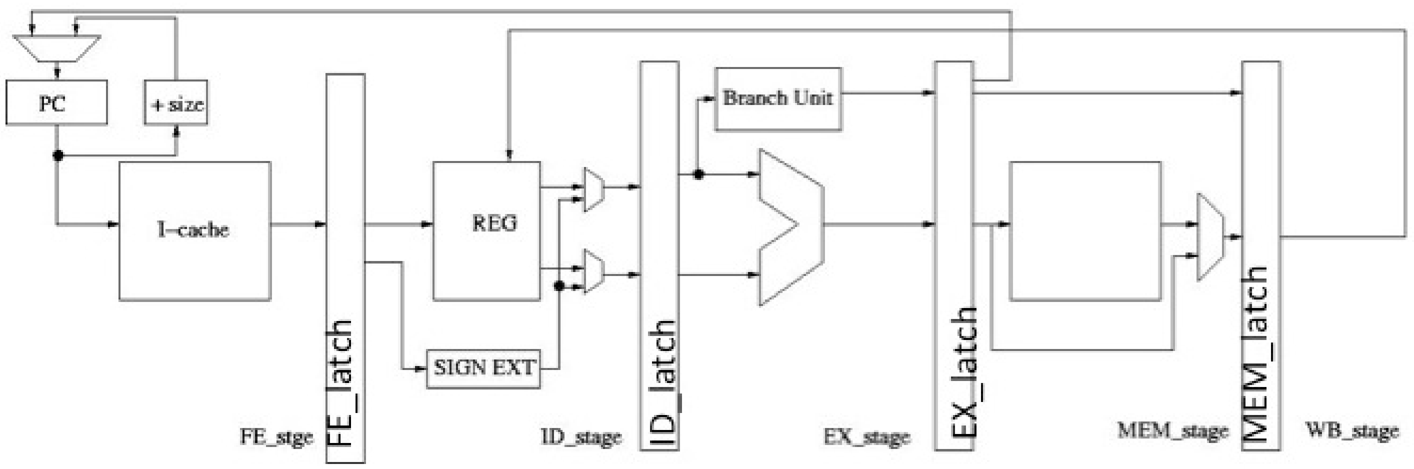
Please follow the submission instructions. If you do not follow the submission file names, you will not receive the full credit. Please check the class homepage to see the latest update. Your code must run on a Linux system with g++4.1. (We will specify the system soon)

Building a Simulator

Introduction

In this assignment, you will implement a simple one-wide in-order 5 stage pipeline trace driven simulator. We provide a simulator frame to help your assignment. Your job is to implement a cycle-level trace-driven simulator that simulates the following architecture. Your final outcome of the simulator is the number of total cycles and other microarchitecture statistics.

Description of the pipeline



This is a simple in order 5-Stage pipeline processor. The simulator reads a trace which resembles RISC type ops. Ops are defined at sim.h file.

The major features of each pipeline are as follows:

- FE_stage: it accesses the I-cache and it reads an op (instruction) from a trace file.

- ID_stage: It reads the register file. If source register files are not ready, there should be a pipeline stall. If an instruction is a control flow instruction, the FE_stage stalls until the branch instruction forwards the target address at the MEM_stage.

- EX_stage: Instructions are executed here. If an instruction takes more than one cycle, (there is the get_op_latency function to check the latency of each instruction), the FE_stage, and ID_stage should stall.

- MEM_stage: It accesses the d-cache. The branch target address is forwarded at this stage.

The front end can fetch a new instruction from the following cycle. Note that the target address is the input of the PC latch
- WB_stage: An instruction is retired at this stage. A destination register is available at this moment. During the first half cycle, it writes value into the register file and the second half cycle, the processor reads the register file. Note that, there is no data forwarding and no branch prediction for this assignment.

Simulator Description

The simulator simulates one processor cycle via the run_a_cycle function. This function calls the following functions, each of which corresponds to one of the stages in the pipeline:

1. FE_stage():

2. ID_stage();

3. EX_stage();

4. MEM_stage();

5. WB_stage();

Your main job is to implement the above five functions which represent each pipeline stage respectively. Note that there are other functions you still need to add more features such as init_structures() but the above 5 functions are the main functions which you need to implement. We have also provided you with interfaces to the instruction and data caches: icache_access and dcache_access functions. Icache_access and dcache_access function always return TRUE for this assignment which models a perfect I-cache and D-cache. However, in the later assignments you will implement a real cache and change the pipeline to handle cache misses.

Grading Schemes
We will run two traces (test traces are not provided to students).
Check IPC values (80%) (0.5% error is acceptable)
Check whether the simulator ends (10%)
Control hazard values (5%) (40% error is acceptable)
Data hazard values (5%) (40% error is acceptable)
Please note that we do not check the latch values. They are provided to help your debugging. **Some important tips**

- Please read these instructions carefully.
- Please read faq before you start.
- You may add more elements in the Op structure. However, if you update more elements, you must update the init_op function whenever you add new elements.

- The simulator must call free_op after an op is retired.

- If you call get_free_op but if you do not use the op, you must call free_op to free the resources

- You may not understand and should not modify the following functions: get_ld_ea, get_st_ea, get_target, ins_decode, write_inst, dprint_inst, init_op_latency, copy_trace_op, init_trace_op, print_stats, print_pipeline. You should not touch *knob* files.

- You may modify init_op, init_op_pool, get_free_op, free_op, get_op, get_op_latency, print_heartbeat, and functions, but it is better for you to look at them.

- We assume a 32 architecture register file system.

- Data structures are in the code: We provide data structures to guide your programming. You can modify data structures to add more data elements.

- You can add your own knobs but please do not modify the existing knob variables. Knob variables are used for grading. (Understanding Knob variables are required to do this assignment.)

- Do not modify print_stats. You must implement the following variables correctly (retired_instruction, data_hazard_count, control_hazard_count, pipeline_latch data structure (op_arrays, op_array_valid), register_file (valid bit)) to receive a full credit. Note that data_hazard_count and control_hazard_count are only incremented at the ID_stage. One op should not increment data_hazard_count more than once at one cycle (i.e. if two source operands cause data hazard, it still increments one).

- In the following assignments, you will extend your first assignment. The code contains some simulator frame structures for future assignments.

- The traces are a simplified version from x86 instructions. The trace does not initialize all the register values. You assume that the beginning of the simulator, all the instructions are ready. Since we truncate some x86 specific information, some uops might not behave as clean as you would expect. This is the real world!.

- Please insert code that prevents from infinite loops. For example, if a simulator waits for a new op for more than 1000 cycles, the simulator should terminate.

How to execute the simulator

Instruction:

copy lab1.tar.gz tar -xvf lab1.tar.gz

cd lab1

make

./sim --print_inst=1 --max_sim_count=10 --print_pipe_freq=1 --trace_file="trace.pzip"

./sim --print_inst=0 --max_inst_count=10 --print_pipe_freq=0

Useful knob variables:

KNOB_OUTPUT_FILE: set the output filename of the print_stats function

KNOB_TRACE_FILE: set the input trace file name

KNOB_MAX_SIM_COUNT: set the maximum cycle_count for the simulation

KNOB_MAX_INST_COUNT: set the maximum inst_count for the simulation

KNOB_PRINIT_PIPE_FREQ: set the frequency of calling print_pipeline() function

KNOB_PRINT_INST: print out debug message while generating traces.

Before Submission

Before you submit, you should check whether your assignment runs until it finishes.
./sim --max_inst_count=0

IPC should be greater than 0. Before you submit, you must run your code at the testing machine (which we haven't figured out yet)

How to count data hazard and control hazard These counter values mimic the behavior of hardware performance counters in real processors. Unlike the text book's example, the behavior of hardware performance counters are very dependent on actual implementation. So we are very lenient about the actual value of the data/control hazard.
Submission Guide

cd lab1
make clean
cd ..
tar cvf lab1.tar lab1/*.h lab1/*.cpp
gzip lab1.tar
Please do not include any trace files

You submit lab1.tar.gz file at T-square.

Grading

We will use the output file from the print_stats function. We might use output of the print_heartbeat function.
If you do not follow the submission file name, or your code does not run at the specified machine, there will be a penalty.
Absolutely no other print messages! Please comment out all your debugging statements before commit!

You must follow the file name convention and director structure. When you untar your lab1.tar.gz (e.g. tar -xvf lab1.tar.gz) you should see lab1 directory. The grading is done by an auto-script. The script cannot recognize different names, different data structures. Because the grading is done by a script, there are several restrictions what you can do such as modifying header files.

Note: The code itself explains more information. Please look at the code and the description together.