Alex Blackson

Yuchao Wang

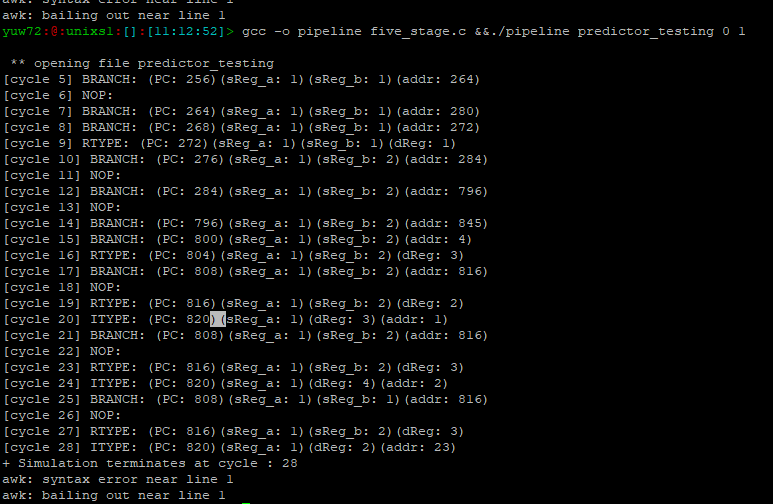
Nick West

Testing – Milestone 1

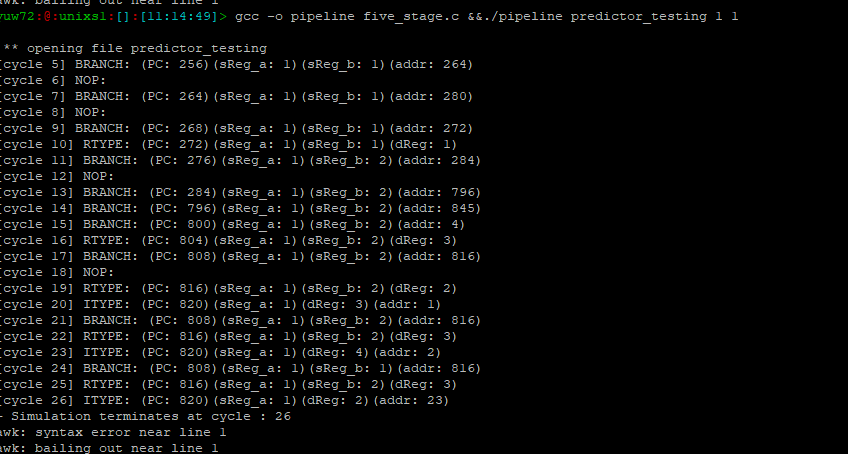
This writeup demonstrates the effect of encountering load-use data hazards and control hazards for both predictions following the “not taken” and branch predictor schemes.

The testing file for control hazards is called “predictor\_testing”. The top half of the below picture uses the “not taken” method. Although the PC is not as large as the one provided in sample.tr, it is still working as long as bits 4 to 9 are the same. Cycle 5 thinks it is not taken, but it is taken (the following instruction’s PC is not the previous PC+4), so there is a NOP to kill the unnecessary instruction. Cycle 7 and cycle 8 predict not taken, and they predict correctly, so there are no NOPs. Thus, it proves our branch “not taken” code is correct.

The lower one follows the branch predictor scheme. I intentionally create a conflict miss so that we can visualize whether the NOPs are inserted correctly. As it shows, the conflict miss is caught and also, the branch instruction in cycle 9 predicts correctly after adjusting its value in hash table. Thus, we can prove the branch predictor is written correctly.

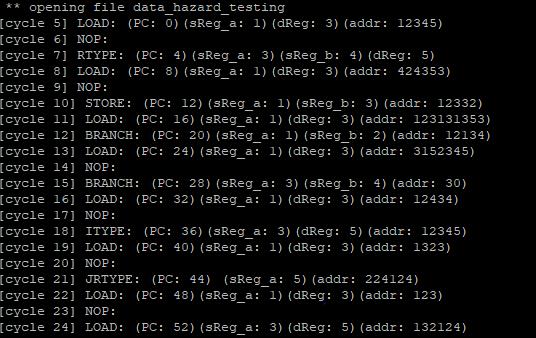


NOT TAKEN



BRANCH PREDICTOR

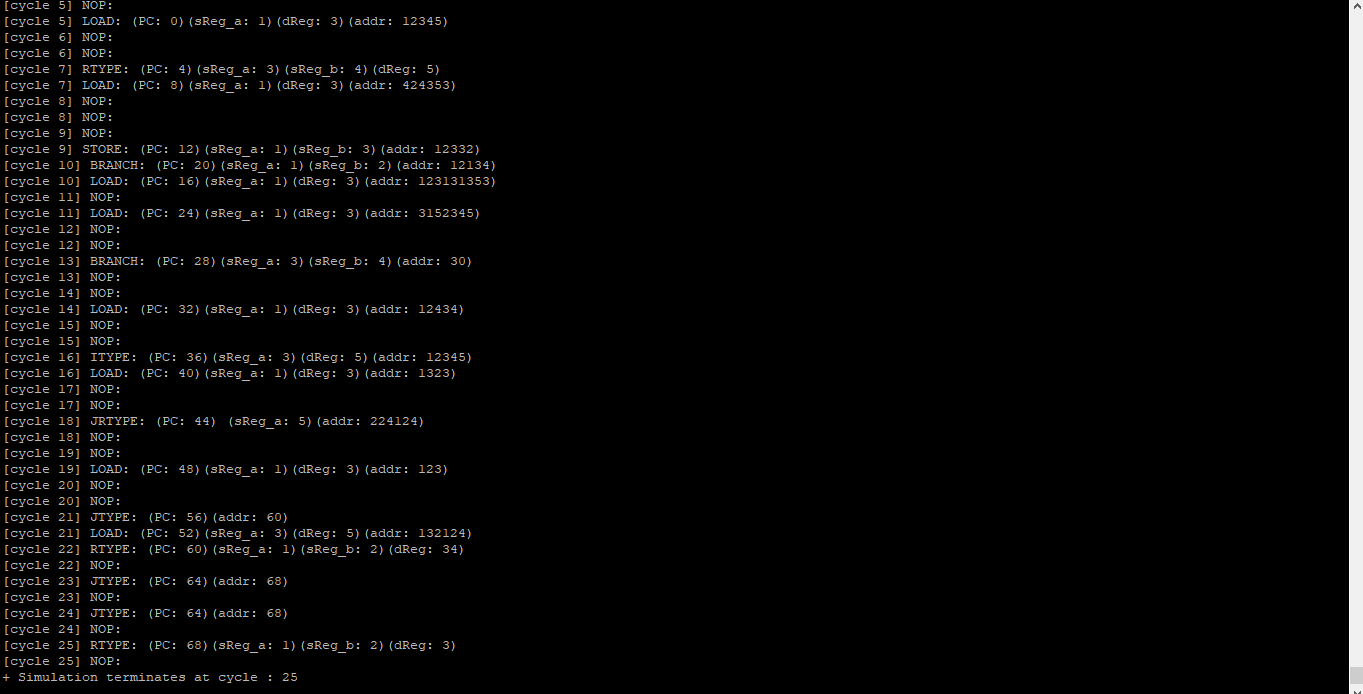
The data hazard detection in our five-stage pipeline is shown below. We create a testing file called “data\_hazard\_testing.” It contains all sorts of scenarios that the pipeline may encounter. For simplicity, we made PC start at 0, and made sReg\_a, sReg\_b and dReg from 1-6. Address doesn’t matter in the data hazard detection. As you can see, from cycle 5 to cycle 15, the data hazard comes from data dependency between the sReg\_a or sReg\_b in the instruction following the load instruction and the dReg in the load instruction. From cycle 16 to cycle 23, data dependency is between dReg and following sReg\_a. When data dependency is detected, a single NOP is added to prevent load-use errors from occurring.



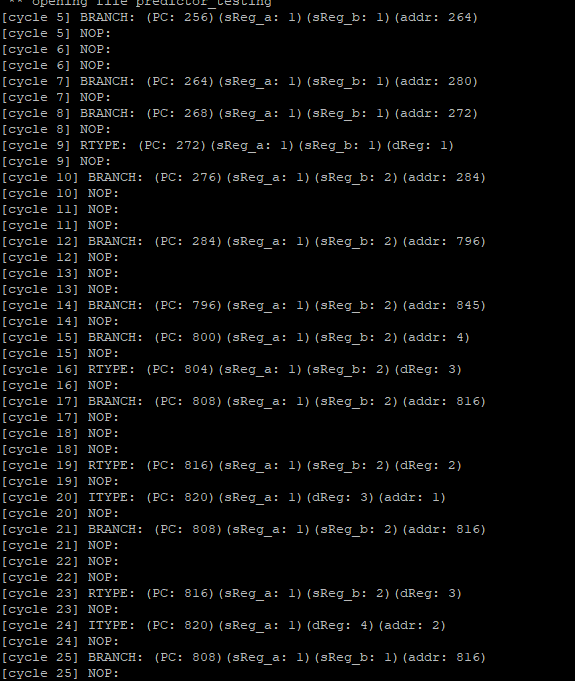
Testing—Milestone 3

By using the same “data\_hazard\_testing” file, we can still cover different situations in superscalar. The picture is shown below. First, the continuous types using the same pipeline cannot stay in the same cycle and need to add a “nop”. The testing below has covered enough examples. Also, J type and JR type and branch type instruction cannot be paired with the instruction following it but can be paired with the instruction preceding it. Cycle 13, cycle 18, cycle 21 have demonstrated this point. We also covered the case in which the last instruction is alone and cannot pair with the other one. In this case, we add a no-op to the other half od the instruction.

Data hazard: By using the same “data\_hazard\_testing” file, we cover all different kinds of types in the testing file to have data dependency from “load” instruction and expect to see it outputs one cycle of “nops” plus a “nop” in the same cycle, unless the load instruction is the second one in the cycle.

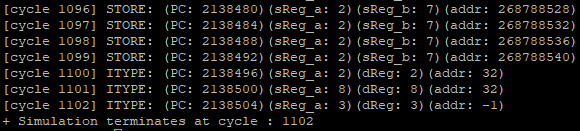


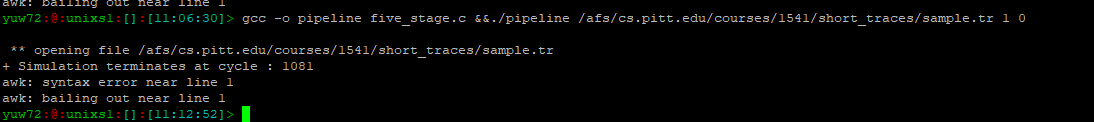
Control hazard: By using the same “predictor\_testing” file, we can see that whenever our program predicts incorrectly, it adds a nop in the same cycle plus one more cycle of nops. Otherwise, it only has one nop in the same cycle.



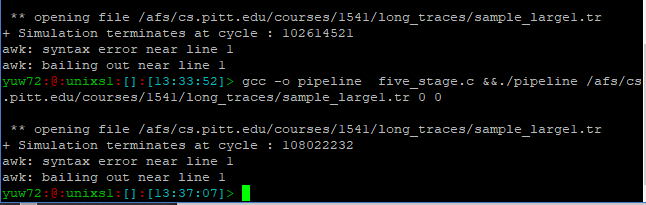
5-Stage No Predictor vs. Predictor Performance

The performance enhancement created by our branch predictor, represented by decreased CPI, is shown below. The first picture shows the number of cycles it took to run the sample.tr file with the branch predictor turned off (1102 cycles). The following picture shows the number of cycles it took to run the same file but with the branch predictor turned on (1081 cycles). This shows that the added intelligence of our branch predictor creates a 21 cycle decrease for the small instruction mix.



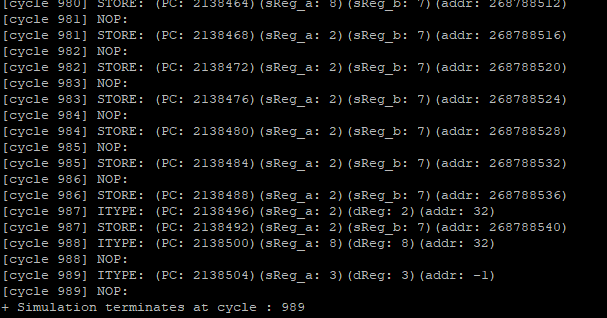


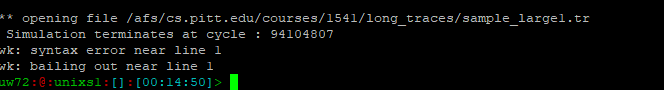
This picture shows the branch predictor’s performance enhancement for the large trace file, sample\_large1.tr. For the no predictor test run, the number of cycles required is 108,022,232, while the predictor test run yields a cycle total of 102,614,521. The predictor reduced the number of cycles by 5,407,711, which is about 5% reduction of total cycles in the large trace file, which is expected due the branch predictor’s use of a hash table to ‘learn’ when and when not to branch.



5-Stage vs. Superscalar Performance

The following two pictures show the results of testing the superscalar implementation on the sample.tr and sample\_large1.tr files respectively. The superscalar, unlike the 5-stage pipeline, fetches two instructions at a time, which it can execute simultaneously if one is a load/store and the other is any other type. However, the superscalar must detect data dependencies, load-use data hazards, and control hazards like the 5-stage pipeline. For the small trace file, the superscalar shows a 100-cycle decrease/improvement from the branch predictor mode of the 5-stage implementation. For the large trace file, the superscalar shows a 12,080,662-cycle improvement from the branch predictor mode of the 5-stage implementation. These improvements are expected, considering the major increase in throughput that the superscalar provides by allowing multiple instructions to execute simultaneously during some cycles.





Implementation Choices

First, in branch predictor, we decide to overwrite the value in an index of hash table once a collision happens.

Second, the output of superscalar follows the rules that the ALU/Branch/Jump instruction prints before load/store instruction in a cycle regardless of their PC order.

Third, when encountering compulsory miss, the branch predictor is default to “not taken” and therefore has no prediction in other words.

We also cover the corner cases where the last instruction in a trace file could have data hazard or control hazard.