Project #3

Branch Predictor for 5-stage MIPS pipeline

Project Checkoff: see class website

Report Due: see class website

Overview:

In this lab, you will implement a static branch prediction technique and a simple dynamic branch predictor. An excellent discussion of various branch prediction implementations can be found in Chapter 9 from Shen and Lipasti. You have to develop some test-benches and provide a figure which represents the accuracy of each method.

Design Procedure:

Branch flow instructions could amount to about 15 to 20 percent of instructions and the problem is that the next fetch address after a branch instruction is not determined after N (branch resolution latency) cycles in a pipelined processor. This means that if we are fetching X number of instructions (the issue width), then a branch misprediction leads to X times N cycles wasted instruction slots. In general, this is very costly in a very deep pipelined processor.

In the following, we will integrate different branch prediction techniques in our pipeline processor. In the first lab, we implemented early branch technique. We did it by comparing registers, computing branch target address, and updating PC during ID stage. Our conservative assumption was that we have to stall until branch outcome and branch target are resolved at the end of ID stage. This costs us one stall to fetch the correct instruction (which could be either determined by branch target address or sequential instruction flow, i.e. PC+4). What are the other consequences of using early branch? One-cycle-delay for every branch still yields a significant performance loss depending on the branch frequency. By means of branch predictors, we will deal with this loss.

Step1: Here we will implement a static branch prediction technique. For static branch prediction techniques, the idea is to analyze and specify behavior of different branches in a program, and in some cases even change control flow, before running it on a hardware, for example, during compilation of the program.

There are different prediction policies including “always taken”, “always not taken”, “backward taken”, “forward not taken” etc. As the first step, our goal is to implement “always not taken” branch predictor. Here the assumption is that the branch is not taken. (In the context of static prediction methodology, this means relying on compiler to modify the control flow so that branches are typically taken.)

Let us assume the general sequential flow of instructions. If the branch is calculated as not taken at the decode stage, there is no penalty on the performance. Alternatively, if it is calculated as taken, you have to flush the next instruction which is already fetched and restart the execution by fetching the instruction at the branch target address. How does this improve the performance? How do we deal with EX, MEM and WB registers when prediction is wrong? If you have already done this in your original processor, then that's great and you can move on to next step.

Step2: Dynamic prediction is based on two mechanisms. You have to design a predictor, which will predict the outcome of the branch, and a target address buffer, which will store the target address in case of predicted taken branch. Note that the entries of target buffer are addressed by using lower bits of PC. In our project, these modules are used in IF stage, i.e. in parallel to the fetching of the instruction. This will allow to supply the predicted branch address for the next instruction without stalling pipeline. Figures 1 and 2 show how this can be done at IF stage.

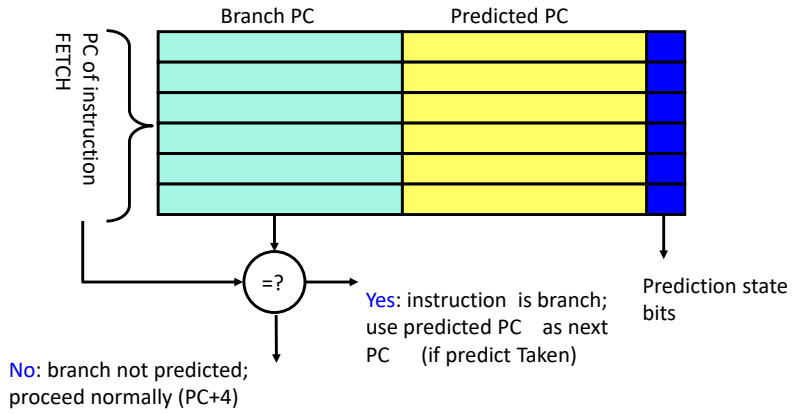

Fig. 1: Branch target buffer with branch state prediction bits.

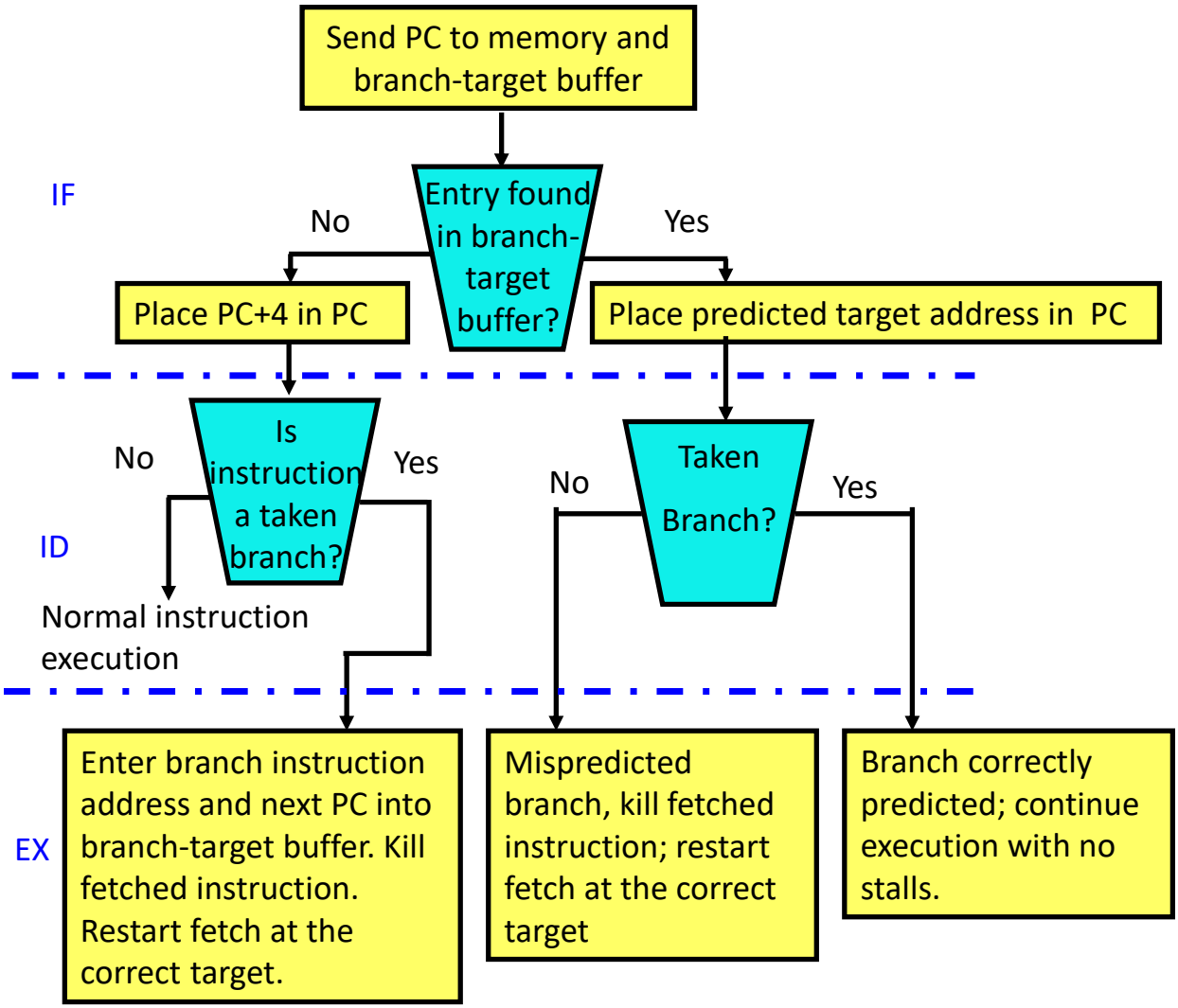

Fig. 2: Steps with branch target buffer for 5-stage pipeline. Note that prediction history table should be updated for all three cases at EX stage.

Let's assume that there are two bits per branch in history table, and there are 2^7 entries in the history table and branch target predictor. For each branch, the two bits are used to encode the four states of finite state machine as shown in figure 3. At the beginning, the BHT is empty (zero) and it corresponds to NN state of the state machine. Note that two bit saturation counter has to be always updated when branch outcome is determined. You should also assume that when branch target buffer is loaded with new addresses, the corresponding branch history table entry is either loaded with updated value for "NN" initial state or with updated value for branch history table bits of the entry which is being kicked out of the buffer.

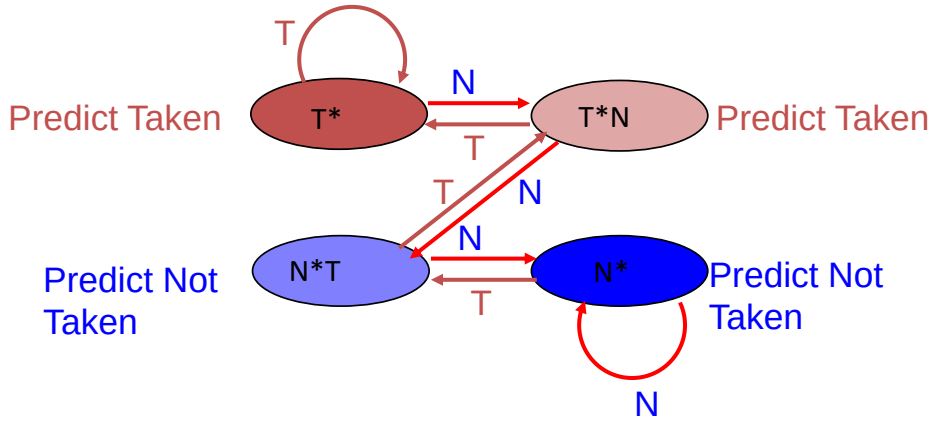

Fig. 3: Two-bit predictor finite state machine.

Step3: In this part, we will add another level of prediction which you learned in the class. The goal is to take advantage of temporal locality of the branch instructions by implementing two-level correlating global predictor (Fig. 4). We will assume that there are two bits in the global history register, and therefore there will be 4 entries (or 8 bits) per each branch (or in our case entries in branch target buffer). Note that now both global history register and corresponding two bits of global history table should be updated by the end of execution stage for all three cases shown in Figure 2.

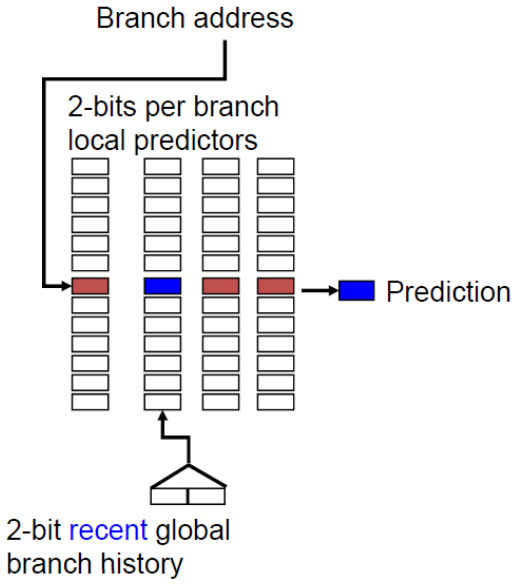

Fig. 4: Two-level correlating “global” branch predictor.

Extra Credit:

1. Modify your design to implement target address cache with 2^3 entries and decoupled branch target history table with 2^{12} entries.

FAQs:

1. Grading:

Your grade is mainly based on correct operation. Functionality is the primary goal. 60% of your score is and the rest of is related to your report.

2. What happens during the checkoff?

You have 10 minutes to present your project. Both of group members must be available during the presentation. You may bring your own laptop or use computers in ECI lab. Everybody has to explain the whole project and answer some questions.

3. What to turn in?

Submit an organized Zip file containing below mentioned files to **zfahimi@ucsb.edu** by the deadline.

Your report is very important. Start with introduction, illustration of instructions, and design methodology. **And then you may focus on each of the steps provided in the manual.** When describing each step, provide the code, the test bench and waveforms. Explain why your waveforms are correct and answer all questions. Organization and completeness of the report determine 40% of your score. Figures should be **readable** and you have to explain them in detail. Mention how many hours you have spent on this lab, your common mistakes in Verilog coding and lessons you learned. Finally, provide a conclusion and wrap up the project. Cite appropriately any references in the report. A folder containing the project files including all source files, test benches and waveforms. **Please heavily comment your code. Poorly commented codes will not be graded.**