6.S195: Lab 7
Complex Branch Predictors

November 1, 2013

Due: Tuesday November 12, 2013

Note: This lab is a direct continuation of Lab 6. You will start with the 6-stage pipeline you created for that lab.

1 Introduction

In this lab you will be expanding the branch predictors in your processor from lab 6 to include direction prediction in addition to target prediction. At the end of lab 6, the SMIPS processor used a branch target buffer (BTB) to perform simple target prediction. This scheme has good performance for smaller programs and tight loops, but it is limited due to the large memory footprint of the BTB per branch.

To predict more branches with less memory per branch, a direction predictor can be used. Instead of storing target PC's, a direction predictor stores some information about the history of the branch in question. The predictor uses this history to make a guess about the direction of the branch: taken or not taken.

Unfortunately, a direction prediction can not be used compute the next PC at the instruction fetch stage like a target predictor can do. Instead, direction predictors are used in the decode stage, where branch targets can first be calculated. This increases the amount of logic needed to track wrong path instructions, but it enables branch prediction tracking for many more branches at a time than a traditional target predictor allows for.

In a complex processor pipeline, target predictors and direction predictors to work together to provide an aggressive and accurate branch prediction scheme. By the end of this lab, you will have a 6 stage SMIPS pipeline with a target predictor and a direction predictor. In the process of implementing this branch prediction scheme, you will learn much about the required details to implement these predictors.

2 Starting Off

To start this lab, pull your personal git repository for lab 7 by running the following command.

```
$ git clone $GITROOT/lab7.git
```

This will create a folder called `lab7` with some support code for this lab. You will copy the rest of the code for this lab will from your lab 6 submission.

Next, copy all the `*.bsv` files from `lab6/src` to `lab7/src`. Add all your source files to the git repo by running the following commands.

```
$ cd lab7
$ git add src/*
$ git commit -am "Initial code from lab 6."
$ git push
```

You should have a 6 stage pipeline with a BTB that is properly trained with feedback from the execute stage. You should also ensure that all 6 stages can fire concurrently by either inserting `$display` statements in each stage and running `run_assembly_loud`, or by running the schedule analysis tool in the Bluespec GUI as mentioned in tutorial 5.

3 Meeting FPGA Timing Requirements

At the end of Lab 6, you ran FPGA synthesis on your processor for the first time. You were not required to meet the provided timing constraints since it was just an introduction to the flow. To begin this lab, you will make any necessary modifications to meet timing requirements if your processor did not already meet them.

One way to meet timing is to iteratively reduce the critical path of the processor. First you synthesize your design for an FPGA using the following commands:

```
$ cd scemi/fpga
$ xilinx build -v xupv5_dut
    .. wait for a long time (about 50 minutes) ..
```

Next you will find the critical path from the log files (refer to lab 6's handout for a description of the log files outputted by the synthesis flow). Finally make modifications to the Bluespec implementation of your processor to reduce the critical path of your processor until it meets the timing requirements.

Exercise 1 (5 Points): Synthesize for FPGA, and modify the processor to meet timing requirements. Compute the instructions per second (IPS) for each benchmark using the clock frequency reported by the Xilinx tools. Report the IPS for each benchmark in discussion.txt.

4 Branch Target Buffer

If you look at the current implementation of the branch target buffer (BTB) in `AddrPred.bsv`, you will notice that it is only updated on mispredictions of taken branches. This causes problems in the following case:

```
cycles = getTime();
insts = getInsts();
for( i = 0 ; i < DATA_SIZE; i++) {
    if( i != 0 ) {
        printChar(',');
    }
    printInt(result_data[i]);
}
printChar('\n');
cycles = getTime() - cycles;
insts = getInsts() - insts;
printStr("Cycles = "); printInt(cycles); printChar('\n');
printStr("Insts  = "); printInt(insts); printChar('\n');
```

You can try adding this code to the bottom of qsort's source code just before the return to see how it performs for yourself. The loop in the above code compiles to the below assembly.

```
   pc:      inst    // assembly
------------------------------------------------
000013b8: 00008021  // move $s0,$zero
000013bc: 27b10010  // addiu $s1,$sp,16
000013c0: 12000002  // beqz $s0,13cc <main+cc>
000013c4: 2404002c  // li $a0,44
000013c8: 0c00040b  // jal 102c <printChar>
000013cc: 8e240000  // lw $a0,0($s1)
000013d0: 26310004  // addiu $s1,$s1,4
000013d4: 26100001  // addiu $s0,$s0,1
000013d8: 0c000408  // jal 1020 <printInt>
000013dc: 2a0200fa  // slti $v0,$s0,250
000013e0: 1440fff7  // bnez $v0,13c0 <main+c0>
```

The `if( i != 0 )` conditional is implemented by the assembly instruction `beqz $s0,13cc`. This branch will be taken for the first iteration of the loop, but it will not be taken again. The first time the processor reaches that instruction, it will mispredict because it has no previous information about the target. After the execute stage detects the misprediction, it will send the corrected PC to the instruction fetch stage, and that stage will include the target for that branch in the BTB. Due to the implementation of the update method in the BTB, that target will never be removed until there is an address collision with another branch. That means the BTB will provide the branch target as the next pc the rest of the time the branch is reached, but it will always be incorrect because it is never taken after the first iteration of the loop. That means the BTB reaches zero percent accuracy for prediction of this branch. That is worse than every other possible branch prediction algorithm, so lets fix that.

To solve this problem, you can add some history bits to the BTB to keep track of whether or not these branches are actually taken. With proper training, two history bits for each index can be used to keep track of how often the branch at the index is taken or not taken to provide a prediction if the branch will be taken or not the next time the instruction is reached. The two history bits make up a saturating counter and provides hysteresis for the prediction. On each branch taken, the history counter for the specific branch should be incremented by one (unless the history counter is already 3). On each branch not taken, the history counter should be decremented by one (unless the history counter is already 0).

This history counter will be used to predict the direction of the branch the next time it is reached. A counter value of 2 or 3 relates will predict taken, and a counter value of 0 or 1 will predict not taken. The predictions 0 and 3 are often referred to as "strongly not taken" or "strongly taken," and the predictions 1 and 2 are often referred to "weakly not taken" or "weakly taken."

Exercise 2 (10 Points): In a new module named `mkBtbHist`, implement a BTB with a two bit saturating counter for each index to predict direction. Make sure this module is trained properly with feedback from the execute stage (not just mispredictions). You may need to run your processor with a large BTB to see a benefit from your counter since small BTBs only keep targets for a short bit of time, not long enough to benefit from history bits.

5 Decoupling Target Prediction from Direction Prediction

The BTB with added history bits provides better branch prediction performance because it includes target prediction and direction prediction. Direction prediction does not need to be coupled with target prediction if you do not mind waiting a cycle until the decode stage to do branch prediction.

For branch instructions, the potential target address is known in the decode stage. If you added a dedicated direction predictor to the decode stage, then you could predict many more branch than is possible with the BTB.

A branch history table (BHT) is a direction predictor that uses a two bit saturating counter just like the one added to the BTB in the previous exercise. Its behavior and implementation have been described in lecture. Its integration in the smips pipeline has also been described in lecture, but as a reminder: you will need to include a new epoch, `dEpoch`, and you will include a way to get training information from execute to the BHT (all taken and not taken branches, not just mispredicted branches).

Exercise 3 (10 Points): Implement a parameterized BHT in the module mkBht and integrate it in your 6 stage SMIPS pipeline along with an 8 entry BTB (no history bits). Make sure the BHT is trained properly and wrong path instructions are dealt with in an efficient manner.

6 Final Synthesis

Now that you have a more sophisticated branch prediction method, you should be getting better IPC on average (assuming your initial BTB wasn't too large and your BHT is large enough). Now it is time to see if you have improved your IPS.

Exercise 4 (5 Points): Synthesize the final processor and do any modifications to meet timing. Compute the instructions per second (IPS) for each benchmark using the clock frequency reported by the Xilinx tools. What is the speed up (or slow down) for each benchmark in instructions per second. Report your answers in discussion.txt.

7 Submitting the lab

When you are done with this lab, commit all your changes and push your changes back to be graded by running the following commands:

```
$ git commit -am "Done with lab 7"
$ git push
```

Bonus: Correcting jr Instructions

This section is not required for the lab, but it introduces some techniques for you to try to improve your processor performance.

There is a lot of research done on branch predictors. At this point, you can try out almost any branch prediction algorithm in your current pipeline to test its performance. Unfortunately many of these algorithms don't work well for the small sized SMIPS benchmarks we are running. Luckily there are still gains that can be made, particularly with `jr` (jump to a register) instructions.

`jr` instructions require reading from a register, and then redirecting the PC based on the value of the register. In your current pipeline, the read is done in the register fetch stage, but redirection for `jr` misprediction is not done until the execute stage. This can be improved upon by adding PC redirection in the register fetch stage and adding a corresponding fetch epoch. **Try this:** Include redirection for `jr` instructions in the register fetch stage.

If you look at the generated assembly code for the SMIPS benchmarks, you will notice that `jr` is almost always used to return from functions that were called using `jal` (jump and link). `jal` always writes to register 31 (`$ra`), so `jr` is almost always jumping to the PC stored in register 31. Why not predict that the next pc after `jr` is always register 31? This could be done in the decode stage if you had a way to read register 31. **Try this:** Include a predictor for `jr` in the decode stage that predicts it will jump to the PC in register 31. To do this, you may want to add an extra read port to the register file that always outputs register 31.

Bonus 2: Running on an FPGA

Warning (10/31/2013): Currently the FPGA boards are hung up. You will probably not be able to run your designs on an FPGA until the machines are fixed. This section is here for later reference once the boards are running again.

If the design compiled without errors, then your design met the timing requirements and should work if programmed to an FPGA. If you want to try running it, log into a vlsifarm computer with an available FPGA board (vlsifarm-03.mit.edu, vlsifarm-04.mit.edu, and vlsifarm-06.mit.edu) and run the following:

```
$ cd scemi/fpga
$ build -v tb
$ programfpga
$ ./run_assembly
$ ./run_benchmarks
```

You should see the same output as simulation if the FPGA is working.