G22.2243: High Performance Computer Architecture

SimpleScalar Assignment #2

(Due: October 19, 2004)

This exercise is meant to help you understand different techniques for reducing pipeline stalls resulting from control hazards. These techniques are a critical component of multiple-issue microprocessors, ensuring that the execution engine always has a window of instructions to operate upon.

Assumptions

As the baseline for this exercise, we will use the 5-stage pipeline from Assignment 1 with the following assumptions (you will need to change your Assignment 1 solution to comply with the last assumption):

- Perfect instruction cache (no I-cache misses)
- Perfect data cache (no D-cache misses)
- Split-phase register access for the writeback (**WB**) stage. This means that writes to registers occur in the first half of the clock cycle, and reads occur in the second half.
- Result forwarding, i.e., results of ALU operations are accessible in the immediate next cycle. Loads require a 1-cycle latency.
- Branches are detected in the **ID** stage and branch targets are available at the end of **ID**. However, branch resolution (determining whether a branch is taken or not) does not take place until the end of the **EX** stage.
- While a branch is being resolved, the pipeline *continues* to fetch instructions from the fall-through path. If a branch ends up being taken, then any speculatively fetched instructions are squashed (by setting the busy flag of the appropriate stage latch to FALSE).

Needless to say, this exercise assumes that you have successfully implemented and debugged the hazard resolution and result forwarding code required by Assignment 1. If you are still encountering problems, you should contact me to get a reference implementation of this baseline simulator (the implication of this of course is that you need to have stopped working on Assignment 1).

The Assignment

The assignment consists of three parts, during the course of which you will extend the pipeline of Assignment 1 with support for branch prediction, a branch target buffer, and a return address stack.

NOTE: *The Simplescalar tools come with a branch prediction module,* `bpred.{c,h}`, *which implements all of the following techniques. I encourage you to use this module rather than building your own implementation of each technique. The point of this exercise is not to verify that you can build a table of 2-bit counters (to take an example) but instead to enhance your understanding of how such prediction techniques fit into the overall pipeline.*
Take a look at one of the provided simulators, `sim-bpred.c`, *to see how the branch prediction module is used: you will just need to use the* `bpred_create`, `bpred_lookup`, *and* `bpred_update` *functions in this exercise.*

Branch Prediction

The baseline pipeline continues to fetch instructions along the fall-through path while a branch is being resolved. Since branches are not resolved until the **EX** stage, taken branches incur a penalty of **2 cycles**. Fall-through branches incur no penalty.

Since a branch instruction is identified as such (and its target determined) in the **ID** stage, employing a mechanism for predicting whether the branch will be taken or not permits a reduction in the branch penalty. Specifically, the penalties incurred for various prediction/actual situations become as below:

Prediction	*Actual*	
	Taken	Not Taken
Taken	1 cycle	2 cycle
Not Taken	2 cycle	0 cycle

The 1-cycle penalty for predicted-taken/actual-taken branches results from the fact that the **ID** stage can update the `fetch_pc` value for the **IF** stage, however, this update is only visible in the next cycle.

Modify the baseline pipeline to emulate the effects of a branch predictor in the **ID** stage. Probably the simplest way of doing this is to explicitly update the `fetch_pc` value for the **IF** stage upon detecting a predicted-taken branch. The **EX** stage now needs to update the `fetch_pc` value in the **IF** stage only for *mispredicted* branches (instead of all taken branches).

1. You will need to setup a new target in the `Makefile` to build your simulator. Follow the instructions from the Assignment 1 handout. In addition, you will need to include `bpred.$(OEXT)` in the target lines (look at the `sim-bpred` target).
2. To allow the fetch path to be updated according to the prediction in the ID stage, you will need access to the target address of a branch instruction. This value is available after the functional simulation of an instruction, but requires the following macro (re)definition:
```
/* target program counter */
#undef SET_TPC
#define SET_TPC(EXPR)          (target_PC = (EXPR))
```
   You also need to declare the variable, `target_PC`, of type `md_addr_t` in the `instruction_decode_stage` function.
3. You will be evaluating two kinds of branch prediction algorithms: **2bit** (2-bit saturating counters), and **correlated** (corresponding to the (2,2) correlating predictor discussed in class). These correspond to predictor types **BPred2Bit** and **BPred2Level.** Add a string option "`-bpred`" to your simulator so that the choice of the predictor can be specified from the command line.
4. You will need to *update* the predictor with the actual outcome of the branch: since the outcome is known after the functional simulation of the instruction, both `bpred_lookup` and `bpred_update` calls can be performed in the **ID** stage.
5. To permit meaningful comparisons of the efficacy of the two predictors, the arguments you supply to `bpred_create` should be as follows: (for the **2bit** predictor) `bimod table size = 1024`; (for the **correlated** predictor) `2lev l1 size = 1, 2lev l2 size = 1024, meta table size = 0, history reg size = 2, history xor address = FALSE.`
6. The Simplescalar bpred module does not provide the option of using only a branch direction predictor (without an accompanying prediction for the target). So, when creating the predictor using `bpred_create`, you will need to supply some reasonable values for the following arguments as well: `btb sets = 128, btb assoc = 1.`
7. The return value from `bpred_lookup` should be interpreted as follows: a non-zero value indicates predict taken, while a zero value indicates that the branch is predicted as not being taken.
8. To enable detection of mispredicted branches in the EX stage, you might want to add a field, `prediction`, to the stage latch structure to keep track of whether a branch was predicted as being taken or not.

Branch Target Buffer

The inclusion of a branch predictor in the **ID** stage reduces the penalty for correctly predicted taken branches from 2 cycles in the baseline pipeline to 1 cycle. To reduce this penalty further, we rely on a technique called Branch Target Buffer (**BTB**), which is employed in the **IF** stage.

A BTB is a cache that stores the target addresses corresponding to taken branch instructions. A BTB lookup can result in the following three return values: a valid target address (BTB hit), a value of 1 (BTB miss, but branch is predicted to be taken), and a value of 0 (BTB miss, branch is predicted to be not taken). In the first case, the fetch path can be speculatively updated to fetch instructions from the (new) target address, resulting in a zero-cycle branch penalty if the branch actually ends up being taken. For the other two cases, there isn't enough information to change the fetch path in the **IF** stage; however, the **ID** stage can now update the `fetch_pc` value in **IF** for predicted-taken branches after computing the target. The **EX** stage as before is responsible for resolving the branch and updating the `fetch_pc` value in IF upon detecting a misprediction.

Note that the use of BTBs can result in two kinds of mispredictions. The first kind is where the BTB predicts a branch as being taken, but provides an incorrect target address for this branch. Such (target) mispredictions can be detected in the **ID** stage, which can reset the `fetch_pc` value to the correct target address. The other kind of misprediction arises when the direction of a branch has been mispredicted: these are detected in the **EX** stage.

The above discussion can be summarized in the following table of branch penalties for different BTB return values, direction predictions, and actual branch outcomes.

		Actual Outcome	
BTB result	*Predicted Direction*	Taken	Not Taken
Hit (correct target)	Taken	0 cycle	2 cycle
Hit (mispredicted target)	Taken	1 cycle	2 cycle
Miss	Taken	1 cycle	2 cycle
Miss	Not Taken	2 cycle	0 cycle

Extend the **IF, ID** , and **EX** stages of the branch prediction pipeline above to emulate the effects of having a combined BTB and branch prediction buffer accessed in the **IF** stage. Your implementation should achieve the branch penalties listed in the above table.

1. The Simplescalar BTB implementation requires what are called "predecode" bits, which allows different kinds of branch instructions to be distinguished without requiring that the entire instruction be decoded. In your simulator, you will emulate the availability of "predecode" bits in the **IF** stage by employing the `MD_SET_OPCODE` macro. Since the macro has no side-effects, you can continue to use it as before in the **ID** stage.
2. To help distinguish between BTB hits with correctly predicted and mispredicted targets, you will need to keep track of the target address that was supplied by the BTB. The `NPC` field in the `if_id_s` stage latch is probably the most suitable for this purpose.
3. Note that a single `bpred_create` suffices to create the combined BTB and branch prediction buffer. The BTB functionality is orthogonal to the branch prediction behavior; thus, it makes sense to talk of the **2bit** predictor with and without a BTB and the **correlated** predictor with and without a BTB. You may want to create a simulator option, `"-btb"`, which controls whether or not the BTB is enabled. In each of these four situations, the `bpred_create` call is supplied the following BTB-related arguments: `btb sets = 128, btb assoc = 1.`

Return Address Stack

A BTB is unable to accurately predict jump targets for procedure return instructions because the target address for such jumps depends upon the call site. The Return Address Stack (RAS) structure allows such jump targets to be accurately predicted by keeping track of the return locations in a stack-like structure: calls push the return address onto the stack, and returns pop the address off the stack. Like a BTB, the RAS is employed in the **IF** stage and has the effect of improving BTB accuracy by separating out treatment of return instructions. Note that just like a BTB, a RAS can mispredict the target of a return instruction (when the call depth exceeds the stack size limit).

In this part of the assignment, you will extend your combined BTB-branch predictor simulator with a RAS structure. All this entails is creation of the bpred structures described earlier with a non-zero value in the `ret_addr stack size` argument. For this exercise, we shall use a value of 8.

Create a simulator option "`-ras`" to turn on/off the RAS functionality from the command line.

Sample Test Code and Sample Output

The code distribution for this assignment is available on the department SUNs at
`/home/vijayk/CompArch/assign2.tar`.

1. *Assembly Code Programs*. Two small sample assembly code programs: `loop.S, and loop2.S` have been provided as sample tests for you to use during your simulator development. To compile these, simply use `/home/vijayk/CompArch/bin/ssbig-na-sstrix-gcc`, with the `–nostdlib` flag. The flag prevents the C standard library from being compiled into your code, thus limiting your instruction count to the number of instructions in your assembly code file (makes it easier to assess whether your cycle count is correct).
2. *Sample output*. Detailed stage-by-stage verbose output is provided for the `loop.S and loop2.S` programs for the two types of branch predictors as well as for pipelines augmented with the branch target buffer and the return address stack. The output format is as described in the handout for Assignment 1.
3. In addition the simulator statistics for two larger test programs (`test-printf` and `anagram`) have also been provided. These programs are available in the `tests-pisa/bin.{little,big}/` subdirectory in the simplesim-3.0 tree and are invoked as follows:

```
./sim-pipe-bpred –bpred 2bit –btb –ras
      tests-pisa/bin.big/test-printf
./sim-pipe-bpred –bpred 2bit –btb –ras
      tests-pisa/bin.big/anagram tests-pisa/inputs/words <
            tests-pisa/inputs/input.txt
```

   One test of correctness of your simulators is that they produce the correct output for these two moderately large programs. This is harder than it sounds because your simulator will occasionally be fetching instructions along a wrong path: your code must detect these cases and fix them as appropriate.

   Note that the above reference results were obtained on the SPARC/Solaris implementation of SimpleScalar. You may see slightly different results on the Linux implementation because of endian-ness issues.

Submission Instructions

Please send e-mail to the instructor (vijayk@cs.nyu.edu) by the due date attaching a tar file containing the following pieces: (1) your simulator source code (this should be just the file `sim-pipe-bpred.c`); (2) output generated by your simulator on the above test programs; (3) a brief README file describing your work and any outstanding problems.
Briefly discuss the behavior of the branch prediction algorithms and the BTB and RAS structures in the context of the smaller programs: do your results make sense?