UChicago CMSC-22200 Computer Architecture, Autumn 2016
Lab 3: Simulating Branch Prediction

Instructor: Prof. Yanjing Li
TAs: Xin-Chuan (Ryan) Wu, Maria Hyun, Mohammad Reza Jokar, Gushu Li

Assigned: Thur., 10/20, 2016
Due: **11:59 pm, Thur., 10/27, 2016**

Introduction

For this assignment, you will write a C program which is a branch predictor and employ that in the pipelined ARM simulator that you developed in Lab 2. (The required instructions are the same as Lab 1 and Lab 2. See the last two pages of the writeup for the list of required instructions). We will fully specify the behavior of the branch predictor. Your job is to extend the simulator in Lab 2 so that it implements branch predictor as specified.

The source code for Lab 3 is provided in the course website. In the `src/` directory, we provide two files (`shell.c` and `shell.h`) that already implement the shell and another four files (`pipe.h pipe.c bp.h bp.c`) that you are allowed to change. You have already implemented the pipelined simulator in `pipe.h pipe.c`. You should implement branch predictor in `bp.h and bp.c` and employ the branch predictor in your pipelined simulator (`pipe.h pipe.c`).

Use the simulator that you developed in Lab 2 as a starting point to implement the pipelined simulator with branch prediction. Your simulator in Lab 2 can be used as a reference to verify that your simulator (with branch predictor) functions correctly and explore how the branch predictor can improve the performance (IPC should be increased). In addition, you should write your own input files that can leverage the benefits of branch predictor.

The Shell

There is a little change in `shell.c` in Lab 3. The `rdump` command will also print the number of cycles that have passed so far.

Branch Predictor

Organization. The *branch predictor* consists of *(i)* a *gshare* predictor and *(ii)* a *branch target buffer*.

Note that the PC in this lab document, as well as `CURRENT_STATE.PC` in the lab source code, refer to the PC in the fetch stage.

Gshare. The gshare predictor uses an **8** bit global branch history register (GHR). The most recent branch is stored in the **least-significant-bit** of the GHR and a value of **'1'** denotes a taken branch. The predictor XORs the GHR with bits `[9:2]` of the PC and uses this 8 bit value to index into a **256-entry** pattern history table (PHT). Each entry of the PHT is a **2 bit** saturating counter: a taken branch increments whereas a not-taken branch decrements; the four values of the counter correspond to strongly not-taken (00), weakly not-taken (01), weakly taken (10), strongly taken (11).

Branch Target Buffer. The branch target buffer (BTB) contains **1024 entries** indexed by bits `[11:2]` of the PC. Each entry of the BTB contains *(i)* an address tag, which is the full 64 bits of the fetch stage PC; *(ii)* a valid bit (1 means the entry is valid, 0 means the entry is not valid); *(iii)* a bit indicating whether this branch is *unconditional* (1 means the branch is conditional, 0 means the branch is unconditional); and *(iv)* the target of the branch, which is 64 bits, with the low two bits always equal to **2'b00**. Note, in actual hardware implementation only bits 63:12 of the PCs are used as tags. However, we use the full 64 bits just to keep the code clean. Also, a valid bit is not strictly required, but we ask you to implement in this lab for better prediction accuracy.

Prediction. At every fetch cycle, the predictor indexes into both the BTB and the PHT. If the predictor misses in the BTB (i.e., address tag != PC or valid bit == 0), then the next PC is predicted as **PC+4**. If the predictor hits in the BTB, then the next PC is predicted as the **target** supplied by the BTB entry when either of the following two conditions are met: *(i)* the BTB entry indicates that the branch is unconditional, or *(ii)* the gshare predictor indicates that the branch should be taken. Otherwise, the next PC is predicted as **PC+4**.

Update. The branch predictor structures are always updated in the execute stage, where all branches are resolved. The update consists of: *(i)* updating the PHT, which is indexed using the current value of the GHR and PC, *(ii)* updating the GHR, and *(iii)* updating the BTB. Unconditional branches *do not* update the PHT or the GHR, but only the BTB (setting the *unconditional* bit in the corresponding entry). When you need to add a new PC/branch target to BTB, but the BTB entry was already taken by another PC with the same lower address bits, you should always replace the entry with the new PC.

Direct and Indirect Branch. In this lab, you need to use BTB to predict indirect branches as well as direct branches. However the accuracy of prediction may be low for indirect branches. Indirect Branches are hard to predict because of the undetermined branch destination.

Initial State. All branch predictor structures are initialized to 0.

The Simulation Routine

As mentioned, there are two files, (`bp.h bp.c`) where you will implement the branch predictor. In the `bp.h` file, we have an struct named `bp_t`, representing the current status of the branch predictor. You should add variables to represent Gshare and BTB in this struct. We also have two function prototypes here, `bp_predict and bp_update`. You should implement these two functions in `bp.c`. You are allowed to add any parameters to these two functions but do not change the prototypes. Then, in your pipelined simulator in `pipe.c`, you must use `bp_predict` to determine the address of the next instruction and `bp_update` to update the status of the branch predictor when a branch decision is made.

```
typedef struct
{
    /* gshare */
    /* BTB */
} bp_t;

void bp_predict(/*......*/);
void bp_update(/*......*/);
```

Flushing the Pipeline

When resolving a branch, the pipeline is flushed under any of the following conditions:

- The instruction is a branch, but the predicted direction does not match the actual direction.
- The instruction is an indirect branch, and it is taken, but the predicted destination (target) does not match the actual destination.
- The instruction is a branch, but it was not recognized as a branch (i.e., BTB miss).

  Note that, we will never incorrectly predict a non-branch instruction as a branch instruction.

Steps to Complete This Lab

We suggest that you should develop the pipelined simulator with branch prediction in 3 steps.

Step 1: Implement the branch predictor structure (Gshare and BTB).

Step 2: Implement `bp_update` to update the branch predictor status and `bp_predict` to predict the next PC.

Step 3: Employ the branch predictor in the pipelined simulator that you developed in Lab 2 and handle the branch recovery for misprediction cases.

Testing Your Code

We provide a few test inputs and their outputs to test your code. In order to test that your branch predictor is able to improve the performance, you should also write more test cases on your own. (Hint. Loops with a large number of iterations may lead to significant performance difference between simulators with/without branch prediction.) You can use the rdump/mdump command to verify that the state of the machine is updated correctly after each cycle.

We will provide you with a reference simulator on the course website so that you can check your branch predictor implementation. There is a new command named `bp_dump` in the reference simulator.

`bp_dump <pht_low> <pht_high> <btb_low> <btb_high>`: dump the contents of PHT and BTB, from entry # low to entry # high to the screen (but not to the dumpfile). It also shows the contents of GHR and fetch-stage PC.

Extra Credits

Tournament Predictor. For extra credits, you need to implement a tournament predictor similar to what you learned in lecture. Try to use both the gshare predictor and a local branch predictor and design your own selector to select which prediction outcome to use. You must also write a document explaining your design, and compare the tournament predictor with gshare. Extra credit is 5%. If you want to do this optimization and implement this tournament predictor, you must make a separate folder with the name Lab3-optimized in your repository, that includes all your src files and executable for the optimized simulator, and also a document explaining your optimizations as mentioned above.

Lab Files

From the course website, you will find a source code distribution with three subdirectories `src/`, `inputs/` and `outputs/`.

In `src/`, we are providing you with the simulator skeleton as described above. You can compile the simulator with the provided `Makefile`.

In `inputs/`, we have written some input files for you. You should write more input files in order to be confident that your simulator is correct.

In `outputs/`, we are providing you with the outputs of the given input files.

Handin

You will be working in groups of 2, and only one copy of the code needs to be submitted for one group, but `please make sure you write down both names and cnetid at the top of the pipe.c and bp.c file.`

You should electronically hand in your code (all files in the `src/` directory) into lab3 folder of your own SVN repository. Contact the TAs if your handin SVN repository does not exist. Your code should be readable and well-documented. In addition, please turn in additional test cases that you used in `inputs/` subdirectory.

You should stick with the same partner as Lab2, and we will take the submission from the student whose name comes first in alphabetical order.

Key Reminders

1. You only need to modify four files, (`pipe.h pipe.c bp.h bp.c`) and you are not allowed to change shell.h and shell.c. You must use mem_read_32 and mem_write_32 functions to implement loading and storing 8-bit, 16-bit, and 64-bit data, and you are not allowed to modify mem_read_32 and mem_write_32.

2. Follow the 3 steps as mentioned previously to complete this lab. After completing all 3 steps, you will have a fully-functioning pipelined simulator with branch prediction.

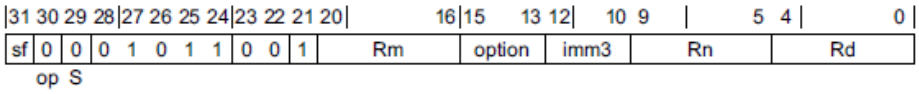
3. You should write more input files in order to be confident that your simulator is correct.

4. Use the same method as Lab1 to assemble a ARM program (with script asm2hex).

LAB3 Required Instructions: (Only need to implement 64-bit variant)

Except for LSL and LSR, you can assume the shift amounts is 0 for all instructions.
ADD (Extended Register, Immediate)
ADDS (Extended Register, Immediate)
CBNZ
CBZ
AND (Shifted Register)
ANDS (Shifted Register)
EOR (Shifted Register)
ORR (Shifted Register)
LDUR
LDURB
LDURH
LSL (Register, Immediate)
LSR (Register, Immediate)
MOVZ
STUR
STURB
STURH
STURW
SUB (Extended Register, Immediate)
SUBS (Extended Register, Immediate)
MUL
SDIV
UDIV
HLT
CMP
BL
BR
B
BEQ
BNE
BGT
BLT
BGE
BLE

1. ADD extended instruction format listed in the reference manual as below



But if you compile ADD Xd, Xn, Xm, you will get a machine code bit_21 is 0. When you compile the format ADD Xd, Xn, Xm, `<extend> <amount>`, the bit_21 is 1. Please make sure ADD Xd, Xn, Xm can work with your simulator. The same thing happens in ADDS, SUB, and SUBS.

2. ADD extended instruction format is shown like this.

```
ADD <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

In lab3, we don't expect you to implement the {`, <extend> #<amount>`} part. You're only required to implement ADD Xd, Xn, Xm. ADDS, SUB and SUBS are the same.

3. For add function, though the reference manual use addWithCarry in the description, you don't need to implement addWithCarry. Simple add is fine.

4. For ADD immediate, defines as below

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12:Zeros(12), datasize);
    when '1x' ReservedValue();
```

No need to implement ReservedValue().

5. For load/store, no need to do CheckSPAlignment().