[Lab 3] Branch Prediction Hardware

Introduction

Branch prediction plays an important role in improving the performance of modern CPUs. The goal of this lab is to improve the performance of your pipelined CPU by adding branch hardware to your design.

Part 0: Lab 3 Set Up

Lab 3 builds on your previous design from Lab 2. First, clone this repo to your local machine and copy the following files from Lab 2 to Lab 3.

```
- src/simple_cpu.v
- src/modules/control/*.v
- src/modules/operation/*.v
- src/modules/memory/data_memory.v
- src/modules/memory/*_reg.v
```

The data memory size needs to be increased for Lab 3. Change `MEM_ADDR_SIZE` to 14 in `data_memory.v`

```
module data_memory #(
  parameter DATA_WIDTH = 32, MEM_ADDR_SIZE = 14
)(
```

Now, if you can compile Lab 3 via `make baseline`, you are ready to move onto Part I. Before that you may need to check if your code can pass the unit test up to `task5` by running `test.py` as below.

```
Linux> python test.py --test=unit
```

Part I: Enable Full RV32I Support

In this lab, you will run real RV32I programs on your CPU design. Isn't it exciting? :) To do so, the first step is to implement the U-type instructions (`lui` and `auipc`) in your pipelined CPU from the previous lab.

Below are the U-type instruction format and the semantics of each instruction. Please refer to the RISC-V materials that we had provided before for further details.

```
opcode: LUI (0110111) / AUIPC (0010111)
------------------------------------------
|      imm[31:12]      |  rd  |  opcode  |
------------------------------------------
|          20          |   5  |    7     |
```

- *LUI (Load Upper Immediate)*: places the 20-bit immediate value in the top 20 bits of the `rd` register and fills in the lowest 12 bits of `rd` with zeros.

  - rd = imm_{20} << 12
- *AUIPC (Add Upper Immediate to PC)*: forms a 32-bit immediate value as above and places the `PC + the 32-bit offset` into the `rd` register.
  - rd = PC + (imm_{20} << 12)

You will need to modify the following files to support the U-type instructions. Note that you need to add control signals & muxes to your previous design to complete Part I.

```
- src/module/simple_cpu.v
- src/module/control/control.v
- src/module/operation/immediate_generator.v
- src/module/memory/idex_reg.v
```

Once you implement the U-type instructions, you will be able to run `unit_tests/task6`.

Part II: Measuring Baseline CPU Performance

The objective of Part II is to (1) make sure your pipelined CPU design from Part I (i.e., without a branch predictor) works okay for a set of real RV32I workloads and (2) measure the baseline CPU performance.

Benchmark for Lab 3

The benchmark consists of the following workloads, which we will use to test out your submission; the unit tests are provided just for your convenience.

```
* bst_array
* fibo
* matmul
* quicksort
* spmv
* spconv
```

We have already included the workloads in `test.py`, so that you can simply run the benchmark using the test script as below.

```
Linux> python test.py --test=benchmark
```

Note that your pipelined CPU (from the previous lab) may fail to run some of the workloads in the benchmark. If that happens, you must fix the bugs of your design.

Measuring CPU Performance

To measure some of the statistics of the CPU, we also provide you with a simple hardware counter (`src/module/utils/hardware_counter.v`). You can use the hardware counter like the example shown below.

```
<add below code to simple_cpu.v>
///////////////////////////////////////////////////////////////////////////////
// Hardware Counters
///////////////////////////////////////////////////////////////////////////////
wire [31:0] CORE_CYCLE;
```

```
hardware_counter m_core_cycle(
  .clk(clk),
  .rstn(rstn),
  .cond(1'b1),

  .counter(CORE_CYCLE)
);
```

Now, if you uncomment the `$display` tasks in `src/riscv_tb.v` as shown below, the test script will print out the CORE_CYCLE statistics.

```
  $display($time, " HARDWARE COUNTERS");
  $display($time, " CORE_CYCLE: %d", my_cpu.CORE_CYCLE);
  /*
  $display($time, " NUM_COND_BRANCHES: %d", my_cpu.NUM_COND_BRANCHES);
  $display($time, " NUM_UNCOND_BRANCHES: %d", my_cpu.NUM_UNCOND_BRANCHES);

  $display($time, " BP_CORRECT: %d", my_cpu.BP_CORRECT);
  $display($time, " BP_INCORRECT: %d", my_cpu.NUM_COND_BRANCHES - my_cpu.BP_CORRECT);
  */
```

After Part II, your baseline CPU needs to run all the workloads in the benchmark and report the performance numbers when running `test.py`. You should add the performance numbers in the report for submission.

> *You may want checkpoint the baseline version for the performance comparison later.*

Part III: Add Branch Hardware to CPU

Now, you will need to implement the branch hardware in your CPU design. In Part III, you will first implement the `gshare` branch predictor and a (direct-mapped cache style) branch target buffer (BTB).

You will need to modify the following files to add branch hardware to your CPU.

```
- module/branch/branch_hardware.v
- module/branch/branch_target_buffer.v
- module/branch/gshare.v
- and others...
```

Branch Hardware

The branch hardware needs to be accessed in the instruction fetch (IF) stage when the instruction is a conditional branch or a (direct/indirect) jump. To do so, you simply need to peek at the opcode of the instruction in the IF stage (sort of pre-decoding). Then, you need to access the branch predictor and the BTB.

> *In the reference design, as in the H&P textbook, the actual branch target address is computed in the EX stage, but is latched to the EX/MEM register. So, a branch is resolved in the `MEM stage` in the next cycle. You should resolve the branch in `MEM` to be consistent with the statistics in the reference design.*

GSHARE BRANCH PREDICTOR

The gshare branch predictor consists of the global branch history register (BHR) and the pattern history table (PHT). Recall that the BHR captures the history of the last `N` branches (i.e., actual branch outcomes), which is

XORed with PC to index into the PHT. Each entry in the PHT has a 2-bit saturating counter, and the number of entries in the PHT is 256 by default. Note that PC[1:0] is not used for indexing.

Initialization

For an active low reset,

> *Each 2-bit counter in the PHT must be initialized to `weakly NT (01)`.*
> *All the entries in the BHR must be initialized to zero (i.e., not taken).*

Updating Branch Predictor

- Recall that the branch predictor must be updated with the actual branch outcome after the branch is resolved. In the implementation, you need to update the predictor according to the `update`, `actually_taken`, and `resovled_pc` signals.
- The branch predictor (BHR and PHT) must be updated only for conditional branches; no updates for unconditional branches (i.e., jumps).

BRANCH TARGET BUFFER (BTB)

The branch target buffer stores the branch target address for a branch PC. Note that only taken branches (or jumps) are stored in the BTB. Each entry in the BTB consists of a `valid` bit, `tag` bits, and a 32-bit `branch target` address. The number of entries in the BTB is 256 by default.

Our BTB is essentially a direct-mapped cache. For a branch PC, if we cannot find the entry in the BTB (i.e., a BTB miss), we just do PC+4 even if the branch predictor says it is 'predicted taken'.

Initialization

For an active low reset,

> *All the entries in the BTB must be invalid (0).*

Updating BTB

- The BTB module updates the entry when the `update` signal is asserted.
- BTB must be updated with the branch target address for all types of branches (conditional, jumps, etc.).
- If the conditional branch is actually not taken, we do not update the BTB.

Next PC Selection

With a branch predictor, now there will be four possible next PC values, which you need to mux with.

- PC
- PC + 4
- Predicted Taken PC (from BTB)
- Misprediction recovery PC (actually taken PC)

Measuring Branch Statistics

After completing Part III, you will need to measure the following branch statistics along with CPU cycles.

- `NUM_COND_BRANCHES`: # of conditional branch instructions
- `NUM_UNCOND_BRANCHES`: # of unconditional branch instructions

- `BP_CORRECT` : # of correct predictions for conditional branches

To do so, uncomment the rest of `$display` tasks in `src/riscv_tb.v` , and implement them accordingly.

```
    $display($time, " HARDWARE COUNTERS");
    $display($time, " CORE_CYCLE: %d", my_cpu.CORE_CYCLE);

    $display($time, " NUM_COND_BRANCHES: %d", my_cpu.NUM_COND_BRANCHES);
    $display($time, " NUM_UNCOND_BRANCHES: %d", my_cpu.NUM_UNCOND_BRANCHES);

    $display($time, " BP_CORRECT: %d", my_cpu.BP_CORRECT);
    $display($time, " BP_INCORRECT: %d", my_cpu.NUM_COND_BRANCHES - my_cpu.BP_CORRECT);
```

We will check the branch statistics to see if your branch hardware works correct.

How to Compile Part III

You can compile the CPU with the gshare branch predictor via `make gshare`. Please make sure your gshare branch predictor works okay before moving onto Part IV.

Part IV: Implement a Modern Branch Predictor

Now, you will need to implement another branch predictor called the `perceptron` predictor. To do so, you need to modify the following files.

```
- module/branch/branch_hardware.v
- module/branch/perceptron.v
- and others...
```

> *You can simply use the BTB and other structures that you have implemented in Part III.*

The perceptron branch predictor uses the simplest form of neural networks (perceptron), instead of using two-bit counters. As we have learned during the class, it is one of the state-of-the-art branch predictors and is employed in some commercial CPUs (e.g., AMD Zen microarchitectures).

Reference Paper

- Daniel A. Jiminez and Calvin Lin "Dynamic Branch Prediction with Perceptrons" HPCA 2001.

PERCEPTRON BRANCH PREDICTOR

Before you start implementing the perceptron predictor, you should read the paper thoroughly and understand how it works. At a high level, the perceptron predictor consists of an `N-bit` BHR and the perceptron table. We use the `lower bits of PC` to index into the perceptron table; note that PC[1:0] is still not used for indexing. Each entry in the table has `N + 1` weights, and each weight is a `signed integer` value in two's complement representation.

We train the weights during the program execution (like the two-bit counters), and we make a branch prediction by performing the dot product of the weights and the BHR. See Section 3 in the paper for further details. The number of entries in the perceptron table is 32 by default.

Initialization

For an active low reset,

> *Each weight in the perceptron table must be initialized to zero.*
> *All the entries in the BHR must be initialized to zero (i.e., not taken).*

Updating Branch Predictor (Training Perceptrons)

- The perceptron predictor must also be updated after a branch is resolved, which we can do according to the `update`, `actually_taken`, and `resovled_pc` signals. See Section 3.3 to understand how to exactly train perceptrons.
- Similar to the two-bit counters used in the gshare predictor, each perceptron weight must be saturated at the MIN and MAX values.

Measuring Branch Statistics

You will also need to measure and report the branch statistics for Part IV. Please refer to the description in Part III.

How to Compile Part IV

To compile the CPU with the perceptron branch predictor, use the `make perceptron` command. Note that the `src/modules/branch_hardware.v` file has the macros to guard the implementation of different branch predictors.

Submission

#DUE: 5/23 (TUE) 11:59PM

Late Policy

- 10% discounted per day (5/24 12:00 AM is a late submission)
- After 05/27 (SAT) 12:00AM, you will get zero score for the assignment

What to Submit

- Your code that we can compile and run
- Submit all the files/directories (do `make clean` first)
- 1-page report that includes the following for the benchmark workloads:
  - performance comparison between the baseline CPU (without branch hardware) and Lab 3
  - branch statistics: # of branch instructions, # of correct predictions, and branch prediction accuracy (# of correct predictions / # of branches instructions)

How to Submit

- Upload your compressed file (zip) to eTL
- Format: YourStudentID_YOURLASTNAME_lab#
  - e.g., 2020-12345_KIM_lab3.zip
- Please make sure you follow the format (10% penalty for the wrong format)