Processor Lab

Overview of this lab

In this lab, you are going to build your own RISC-V processor and simulate programs. We are going to dive into the implementation of Ripes and implement our own processor model. We will also get a taste of the security issues in modern processors.

Before starting this lab, you are supposed to be equipped with the following:

- The same environment as Lab 1 (the assembly lab)
- Basic C++ programming language
- Basic RISC-V assembly

After this lab, you will learn the following:

- Understanding the 5-stage pipelined processor model
- Understanding the implementations of pipeline hazards, forwarding, and branch prediction
- Understanding the security issues of branch predictor units
- Better understanding about simulating, especially Ripes, a RISC-V simulator

During this lab, you are suggested to refer to the following materials:

- RISC-V ISA

  Quick reference card

  Specification, Volume 1

- Linux cheat sheet

  https://www.linuxtrainingacademy.com/linux-commands-cheat-sheet/

- Papers about branch prediction attacks

  On the Power of Simple Branch Prediction Analysis (ASIACCS'07)

  BranchScope: A New Side-Channel Attack on Directional Branch Predictor (ASPLOS'18)

Provided Infrastructure

Inside this repository, we provide the following files in the tree:

```
├── lab2
    ├── lab2-handout.pdf --- This document
    ├── testcases --- The folder contains testcases
            ├── hazard_testcases --- The folder contains testcases used to
                                     test the correctness of hazard
detection
                                     implementation.
            ├── branch_testcases --- The folder contains testcases used to
                                     test the correctness of branch
prediction
                                     implementation.
            ├── performance_testcases  --- The folder contains the
testcases
                                           to test performance of the
overall
                                           processor.
      ├── riscv-card.pdf
      ├── branch-prediction.pdf
      ├── asiaccs07.pdf
      ├── asplos18.pdf
```

Tutorial of Designing Processors in Ripes

This is a basic tutorial of how processors are implemented in Ripes. In addition to this article, **reading the code directly** is always a great way to learn exactly what happens. We highly recommend you to read the code in `Ripes/src/processors/RISC-V/rv5s_hz`, especially `rv5s_hz.h`, on Git branch `lab2_v1.2_bp`, and `Ripes/src/processors/RISC-V/rv5s_eb`, especially `rv5s_hz.h`, on Git branch `lab2_v2.0_eb`. There are other processor models in `Ripes/src/processors/RISC-V/`. Reading the code of those models may also help you finish the lab, but it should be enough to read branch `lab2_v2.0_eb` since it corresponds to the latest Ripe version.

The following content can also be found in the comments in the code, especially `rv5s_hz.h`.

Ports in Ripes

Just like the real hardware, processors in Ripes are composed of many components. Components communicate with each other through ports and the links that connect the ports. For a component, data and controls flow into its input ports and the results flow out through the output ports.

Input Ports

Input ports are the ports that data flow into a component. For example, in the hazard unit, we need the index of the register as input. Then we can define an input port in the following way: (as in `Ripes/src/processors/RISC-V/rv5s_hz/rv_hz_hzunit.h`)

```
INPUTPORT(id_reg1_idx, RV_REGS_BITS);
```

`id_reg1_idx` is the name of the port. To get the value of it, use `id_reg1_idx.uValue()`. `RV_REGS_BITS` is the width of the port.

Output Ports

Output ports are the ports that data flow out of a component. In the branch unit, for example, we need to output the result of the branch. Then we can define an output port in the following way: (as in `Ripes/src/processors/RISC-V/rv5s_hz/rv_branch_id.h`)

```
OUTPUTPORT(taken, 1);
```

`taken` is the name of the port. To output the value to the port, add the following code to the constructor of the component:

```
taken << [=] {
    return branchTaken();
};
```

The expression after `taken` is a lambda expression in C++ 11. You do not need to know more about lambda expression. Just return the value that you want to output to the port. All the member variables and functions can be used inside the lambda expression.

Connect Between Ports

A pair of connected ports means data are transferred from the first port (e.g., an output port of a component) to the second (e.g., an input port of another component). Ripes overloads the `>>` operator to represent there is a link between the two ports linked by `>>`. `<<` operator is also overloaded to represent there is a combinational logic (C++ lambda expr) on the right side and the logic output is assigned to the left side.

You do not need to know what is operation overloading in C++. You only need to know how to use `>>`.

For example `Component1->PortA >> Component2->PortB` means there is a link from A to B, and data can be transferred from A to B through this link. The port on the left of `>>` must be an output port, and the port on the right must be an input port. Notice that there should be one and only one link for each input port, while an output port can link to many links. For example:

Correct: `PortA >> PortB; PortA >> PortC;`

Error!: `PortX >> PortZ; PortY >> PortZ;`

Useful Components in Ripes

Multiplexer

A multiplexer (MUX), also known as a data selector, is a device that selects between several input signals and forwards the selected input to a single output. The multiplexer you will use in Ripes is `EnumMultiplexer`.

An `EnumMultiplexer` is a multiplexer that chooses from two values of two ports using the specified `Enum` type to select the port. For example,

```
(1) Enum(ValueSrc, Src0, Src1);
(2) SUBCOMPONENT(mp, TYPE(EnumMultiplexer<Value, ValueWidth>));
(3) src0->out >> mp->get(ValueSrc::Src0);
```

```
(4) src1->out >> mp->get(ValueSrc::Src1);
(5) ValueSrc srcSelect() { return ValueSrc::Src0; }
(6) srcSelect() >> mp->select;
```

As we can see in the example, line 1 defines an Enum type. Line 2 declare an EnumMultiplexer. Line 3 and line 4 inputs the values that the multiplexer chosen from. Line 6 inputs the choosing standard to the multiplexer. In this code snippet, `mp` will always choose the `Src0` as defined in line 5.

Then the value of `mp->out` will be `src0->out`.

Gates

A logic gate is an idealized model of computation or physical electronic device implementing a Boolean function, a logical operation performed on one or more binary inputs that produces a single binary output.

In Ripes, there are four kinds of gates that can be used (`Or`, `Xor`, `And`, `Not`). They implement the corresponding functions just as their names suggest.

```
SUBCOMPONENT(x_y_or, TYPE(Or<1, 2>))
x >> *x_y_or->in[0]
y >> *x_y_or->in[1]
```

Then `x_y_or->out` should be the value of `x | y`. Note that `x_y_or->in[0]` and `x_y_or->in[1]` need to be dereferenced here because they are C++ pointers to ports.

Registers

For a pipelined processor model, we need registered storage between two stages. For example, in `Ripes/src/processors/RISC-V/rv5s_hz/rv5s_hz_ifid.h`, we use the macro `REGISTERED_CLEN_INPUT(NAME, WIDTH)` to define registers, then we use two ports called `NAME_in` and `NAME_out` which are to set the register and to get the value from the register. For connecting them, we use `CONNECT_REGISTERED_CLEN_INPUT(NAME, CLEAR_PORT, ENABLE_PORT)` as the file shows.

Also you can directly define registers as sub-componenets, use `SUBCOMPONENT(NAME, Register<WIDTH>);` or `SUBCOMPONENT(NAME, RegisterClEn<WIDTH>)` to define registers, and use `NAME->in/out` to connect.

More

If you want to learn more about the hardware components used in Ripes, you may refer to the document and code of VSRTL (`Ripes/external/VSRTL`), which is the core dependency of Ripes.

Basic Architecture of the Processor in `rv5s_hz.h` on Git Branch `lab2_v1.2_bp`

We are going to implement a processor with hazard detection and branch prediction in `rv5s_hz.h`. The components in each stage are as follows:

IF	ID	EX	MEM	WB
PC, Instruction Memory, Branch Predictor (Prediction), Branch Target Buffer (Prediction)	Branch, Control, Decode, Register File, Immediate, Branch Predictor (Update), Branch Target Buffer (Update)	ALU	Data Memory	

And there are one set of pipeline registers between every two adjacent stages, just as the processor model in the lecture. They are `ifid_reg`, `idex_reg`, `exmem_reg`, and `memwb_reg`. There is also a Hazard Unit across stages that handle hazard.

Basic Architecture of the Processor in `rv5s_eb.h` on Git Branch `lab2_v2.0_eb`

This processor is similar to the one above, with two differences: there is a Forwarding Unit across stages; there is no branch predictor.

Tasks

In Lab 2, you are asked to implement two separate 5-stage pipelined RISC-V processor architectures, one with hazard/forwarding handling when resolving branch earlier (`lab2_v2.0_eb`), and the other with branch prediction when disabling forwarding (`lab2_v1.2_bp`). To avoid interference between the two implementations, we decouple the tasks into two Git branches. More specifically,

- Implement early branch resolving and handle data hazard and forwarding. (`lab2_v2.0_eb`)
- Run experiments to see how much programs can benefit from early branch resolving. (`lab2_v2.0_eb`)
- Implement branch prediction. (`lab2_v1.2_bp`)
- Demonstrate an attack based on branch prediction when running a program (`lab2_v1.2_bp`).
- Write a design document.

You are supposed to implement your own processor in `Ripes/src/processors/RISC-V/rv5s_hz` on Git branch `lab2_v1.2_bp` and in `Ripes/src/processors/RISC-V/rv5s_eb` on Git branch `lab2_v2.0_eb`. You can modify any file in these two folders. The files out of these folders should remain unchanged.

There are other models in Ripes that are different from the processor you are going to implement. Feel free to borrow code from those processors. But you need to make sure you know what you are doing and implement your processor correctly. Copying the code of other processors directly may cause errors that are hard to debug. Notice that there is no forwarding mechanism on `lab2_v1.2_bp`, and the branch resolving should be moved to an earlier stage on `lab2_v2.0_eb`. These are different from the processor you used to simulate your Gaussian filter implementation in Lab 1.

Don't be scared by so many files. Modular design is always a good habit for understandability. You will find that most of the files are short and simple after you finish the lab.

1. Setup Environment

First make sure you have the identical directory structure as in the section [Provided Infrastructure](#Provided Infrastructure).

If you encounter errors like `server certificate verification failed`, add `GIT_SSL_NO_VERIFY=1` in front of git command, e.g., `GIT_SSL_NO_VERIFY=1 git pull`.

For the first two tasks

```
$ cd ~/yao-archlab-f22/Ripes/
$ git pull
$ git checkout --recurse-submodules lab2_v2.0_eb
$ mkdir -p build && cd build
$ cmake ../
$ make -j4
```

Note that at this point, your compilation would *fail*, because you have not fully implemented the necessary functionalities. If you want to compile and see how the latest Ripe works (which is almost the same as in old Ripes), you can checkout to the previous commit `f2b8382af807dd343b5314e1ba869123339a9cc8` and add some lines in the `Ripes/CMakeLists.txt` (you can run `git diff f2b8382af807dd343b5314e1ba869123339a9cc8 lab2_v2.0_eb` and see the changes on the `CMakeLists.txt`).

For remaining tasks

```
$ cd ~/yao-archlab-f22/Ripes/
$ git pull
$ git checkout --recurse-submodules lab2_v1.2_bp
$ mkdir -p build && cd build
$ cmake ../
$ make -j4
```

Username: yao-archlab-f22

Password: 7U6-zJHEn-_6sj8iiHc8

Note: You must checkout the correct Git branches and finish the corresponding tasks on them, respectively; if you have some changes left uncommit, stash/commit/reset before `git pull` or `git checkout`.

When running with the processor you implement, **select the processor named "5-stage RISC-V processor with early branch resolving" and "5-stage RISC-V processor with hazard detection", respectively**. (The link seems chaotic but this is fine.)

2. Implementing Earlier Branch Resolving

The original 5-stage pipelined processor `rv5s` in Ripes does not use the earlier branch resolving design introduced in the lecture. With earlier branch resolving, the register operands needed to determine the branch direction or the jump destination are read at the ID stage but are written at the WB stage. This raises an issue when two nearby instructions access the same register with a read-after-write (RAW) dependency.

In this part, you are required to implement **earlier branch resolving with data hazard detection and forwarding**. Because our processor is in-order, the only data hazard that you need to consider is the RAW hazard. Review the lecture notes if you forget anything about data hazards. Note that the original `rv5s` processor has implemented its own hazard detection and forwarding, but when we resolve branch earlier, such logic need to be adjusted.

- You will need to finish all the **TODO**s in `rv5s_eb`.
- Your implementation is restricted in the directory `rv5s_eb`.
- You are allowed to completely re-write the files in `rv5s_eb`, not following the annotated **TODO**s. But this would be more difficult and we do not advice you to do so.
- The grading of this task depends on both the correctness of your implementation and the expected performance. For correctness, you should detect all hazard cases. For performance, you are expected to implement all commonly used forwarding mechanisms (see the lecture notes and/or the implementation of `rv5s` for reference).

Checkpoint 1: After you finish this part, you will be able to pass all the testcases in `hazard_testcases` and `branch_testcases`. Note that branch prediction does not affect the correctness of the processor.

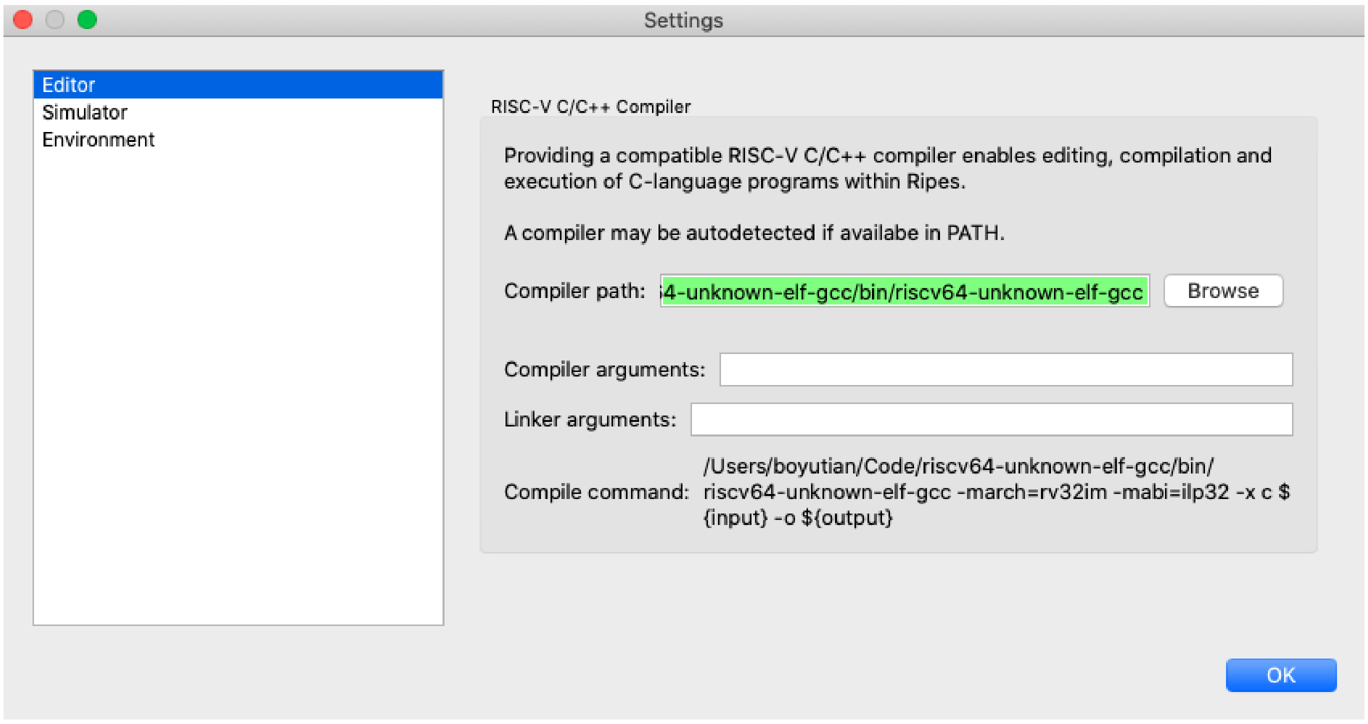
3. Profiling Benefits of Earlier Branch Resolving

You do not need to write any code in this part. You will need to run some testcases to learn the benefits of earlier branch resolving. You need to compare the performance of the following two processor models:

- The 5-stage pipelined processor with earlier branch resolving and hazard detection and forwarding, labeled as **5-Stage Processor with Earlier Branch Resolving**, i.e., the one you just implemented;
- The 5-stage pipelined processor with hazard detection and forwarding, labeled as **5-Stage Processor**, which is the standard 5-stage processor in Ripes and the implementation is in `rv5s`.

Run all the testcases in `branch_testcases` and `performance_testcases` to compare the results. You should explain why the cycle counts differ in the two processor models.

When you run the C codes in `performance_testcases`, you need to delete all the compiler arguments and linker arguments (by default, no optimization is involved). **This is different from what we did in Lab 1.** The setting window should look like this:



RISC-V C/C++ Compiler

Providing a compatible RISC-V C/C++ compiler enables editing, compilation and execution of C-language programs within Ripes.

A compiler may be autodetected if availabe in PATH.

Compiler path: 4-unknown-elf-gcc/bin/riscv64-unknown-elf-gcc

Compiler arguments:

Linker arguments:

Compile command: /Users/boyutian/Code/riscv64-unknown-elf-gcc/bin/riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 -x c ${input} -o ${output}

4. Implementing Branch Prediction

Branch prediction is a technique used in processor design that attempts to guess which way a branch will go before this is known definitively. See the lecture notes for more information about branch prediction.

In this part, you are required to implement the branch prediction policy and control the branch prediction module. You should now switch to Git branch `lab2_v1.2_bp`.

To simplify the implementation, we do NOT separately handle jump (`JAL`, `JALR`) and branch instructions (`BEQ`, `BNE`, `BGE`,`BLT`,`BGEU`,`BLTU`). They are all uniformly handled by `branch`, `branch_predictor` and `branch_target_buffer` in `rv5s_hz.h`. And we do not have return address stack (RAS) design. The function return are handled just as normal jumps.

We have provided you a basic implementation framework to implement branch prediction. The branch prediction modules (`branch_predictor` and `branch_target_buffer`, at the IF stage) and the branch module (`branch`, at the ID stage) can both modify the PC value. There is a multiplexer named `pc_src` to select the final PC value from them. Obviously, the branch module outputs the actual accurate branch result and the branch prediction modules output the predicted result. You need to implement the selection of `pc_src`, to ensure that the branch prediction can benefit the execution when the prediction is right, and the correct PC can be recovered when the prediction is wrong.

You can also have other designs to implement branch prediction. This is totally OK. But it may take your more time. If you use the given implementation framework, You will need to finish the following **TODO**s:

`rv5s_hz/rv_branch_id.h:`

- Implement the `wrongPredictPc()` function. This function returns 1 if the predicted result of this instruction is wrong (later connected to `ifid_reg->clear` to undo an instruction) and returns 0 if the prediction is right.
    - *Hint: you may need to add some input ports to decide whether the prediction is wrong. For example, at least you need the result of prediction. Still, you need to connect the ports you added*

    in `rv5s_hz.h` to make them valid.
  - We name the branch module as `rv_branch_id` because it is in the ID stage, which is different from other processor models in Ripes.
- Implement the value generation for the output port `branch_final_select`. It outputs `PcSrcFinal::ACTUAL` when the PC should take the value from the branch module and outputs `PcSrcFinal::PREDICT` if takes the value from the prediction modules.

It is worth noting that, although the branch predictor and the branch target buffer can only affect the performance, the processor will not work correctly unless the branch module is implemented correctly.

`rv5s_hz/rv_branch_predictor.h`:

- Implement the `branchPredict()` function. This function returns 1 if the branch is predicted taken and returns 0 if predicted not taken.
- Design a branch history table (BHT) to maintain the branch history information, with initialization if necessary. The BHT should be updated in the lambda expression connected to `update_wire->out`. You can initialize the BHT in the constructor of `BranchPredictor`. You can use the design in the lecture, which maintains a saturate counter for each branch instruction.
  - The output value of `update_wire->out` and the `endpoint` component is dummy. They are just for updating the predictor. Just implement the actual update in the lambda expression after `update_wire->out`.
  - The size of branch history table here is not constrained, which is not true for the real-world processors. In Lab 2, you can use any data structures you like. But keep in mind that hardware resources are limited in real-world processors.
  - You are **not allowed** to only implement a branch prediction scheme which is too simple (any scheme without using branch history information, such as always-predict-taken or always-predict-not-taken). There are performance bars for the performance testcases. **You can get the performance score only if you exceed the bars. **
    - performance_1_matrix.c: 642333 cycles
    - performance_2_ranpi.c: 667945 cycles
- Implement the `reset` function **(If you need to)**.
  - This function is called every time you clicked the 'Reset' button in Ripes. This function is for reset the branch history table to its initial blank status. You may want to do things such as clearing a map `M` using `M.clear()`.
  - If you do not need to clear anything, you can leave the function empty.
- (**Bonus**) Use a **global** branch history to not only maintain history for a single instruction, but also for a global pattern for more accuracy prediction. For an example which could benefit from this, you can refer to the matrix performance test: only one of the two `if`s in the nested loops could be executed, so that global pattern will help to predict.
  - *Hint: you can refer to the provided material on branch prediction and WikiPedia.*

`rv5s_hz/rv_branch_target_buffer.h`:

- Implement the value generation of the output port `target_address`. This port outputs the predicted target address of the branch instruction.

- Design a branch target buffer (BTB) to maintain the historical target address information. It is a table to use the address of the branch instruction as the index and to get the predicted address as the content data. The BTB should be updated in the lambda expression connected to `update_wire->out`. You can initialize the BTB in the constructor of `BranchTargetBuffer`.
  - The output value of `update_wire->out` and the `endpoint` component is dummy. They are just for updating the target buffer. Just implement the actual update in the lambda expression after `update_wire->out`.
  - Same as the branch history table in the branch predictor, the size of branch target buffer here is not constrained.
- Implement the `reset` function. **(If you need to)**
  - Same as the `reset` function in branch predictor. If you do not need to clear anything, you can leave the function empty.

`rv5s_hz/rv5s_hz.h`

- We also mark a **Part 4 TODO** in `rv5s_hz.h`. It is to modify the PC4 source of the branch address generation at the branch module when using branch prediction. Think about why and how to change the source of PC4 here.
  - *Hint: If a non-taken branch was incorrectly predicted as taken, and the branch module now need to correct it to the next instruction of the branch instruction, what will happen?*

Checkpoint 2: After you finish this part, you will be able to pass all the testcases in `branch_testcases` and your processor should run the programs in `performance_testcases` correctly.

After you finish all the parts above, run the testcases in `performance_testcases` and see how good you can do.

5. Demonstrating Branch-Prediction-Based Attacks

Modern processors are complex hardware systems, and complex systems can easily exhibit security vulnerabilities. Microarchitectural side channels are one of the most difficult security issues to defend in today's processors. In this part, you will be demonstrating one of such attacks based on the branch predictor you just implemented. You do not need to implement any Ripes code in this part, but may write some assembly code.

Read the two given papers (or more at your choice) about the attacks on branch predictors. The principle of this kind of attacks lies in that **the attacker and the victim sharing the same physical resources**, e.g., the branch predictor unit. The attacker first primes the unit in some way, to cause a potential resource conflict. After the victim executes its code, some states may be changed, including the state that is initialized by the attacker. Then the attacker can observe this change by probing. For example, assume at first the attacker initializes the predicting branch direction/BTB in some way. If the branch predictor changes the predicting direction/BTB due to the execution of the victim code, the attacker could observe a longer latency when it executes its own code later due to the wrong prediction. In Ripes, the attacker can easily get the precise latency, since Ripes provides the cycle count.

Your task is to demonstrate an example with carefully designed attacker and victim code, so that the attacker can distinguish some secret data (e.g., one bit of 0 or 1) in the victim code, through observing its own code execution time. Show the attacker and victim code your designed, run them with different secret data on Ripes, and show the time difference. Explain how your attack demonstration utilizes the hardware states, e.g., information in the branch predictor and/or the branch target buffer.

There is one more problem to solve. Usually the attacker and the victim are two parties, each of whom runs a separate process. But we have only been running a single program on Ripes so far. You will have to simulate the case of multiple processes. *Hint*: In reality, the two processes are running on a single physical core with time-multiplexing, so their execution can be seen as interleaved in time. When you finish running the program, see if Ripes keeps the execution states.

Write-up

You should write a design document for Lab 2. There is no specific format required, but you should demonstrate how your processor works, in a clear way. You should at least include the following parts:

Methodology

List the ports you added to implement the earlier branch resolving logic, the hazard unit, the forwarding unit, the branch module, the branch predictor and the branch target buffer. Explain the functionality of each port.

Describe the solution of detecting RAW hazards. Does your implementation work correctly? If not, do you have any reasons or observations?

Describe the branch prediction policy you used. Describe the data structure you used to keep the branch history and branch target history.

How did you finish the lab? Show your debugging process if there is anything interesting.

Results

Part 3: Show the total cycle counts and the average CPI values when running the given testcases on your implementation with earlier branch resolving and on the basic processor model. Explain your results.

Part 4: Show the total cycle counts and the average CPI values when running the given testcases on your implementation with branch prediction. Also show the accuracy of your branch predictor. If you have designed several branch predictions (not required), you can show the results separately. Explain your results. How much can programs benefit from your policies? If you implement a global-based prediction, also explain and show your results.

Part 5: Show the example attacker and victim code, and how you organize the code execution. Demonstrate the execution time difference the attacker can observe when running with different victim secret data. Explain how your code can successfully exploit the hardware states in branch prediction.

Question Answering

- How are stalls implemented in Ripes? Find out the mechanism and explain how it works.
- How should we choose the sources of PC? When can the value from the prediction modules be used, and when should the PC be recovered by the branch module? You can draw a diagram or table to show the different conditions.
- How are mispredicted branches handled? Find out the mechanism and explain how it works.

- What are the pros and cons of resolving branches earlier at the ID stage? You can consider stalls, hardware cost, branch prediction, etc.
- How to defend against branch-prediction-based attacks?
- Can we implement an out-of-order processor using this design? If we can, how should the lab framework be modified? If not, please briefly write down the major reasons.
- (**Bonus**) The prediction is in the IF stage while updating is in the ID stage. Think about two continuous branch instructions: the first one is in the ID stage, and the second is in the IF stage. What is the order of updating the first result and querying the second prediction? How to control the order? How do local-based and global-based prediction algorithms be affected by the order?

Submission

Create a folder with the name of `lab2-<student ID>-<first name>-<last name>`, e.g., `lab2-2020123456-xiaoming-wang`. Put the following files in it. You should generate the patch files by using `git diff` against the initial state on *all* files you have changed (make sure you are not missing any code changes). Compress the folder as a `.zip` package and upload it to `learn.tsinghua.edu.cn` (网络学堂) by **Nov 25 (Friday)**. You may submit for multiple times, and we will grade based on the latest submission.

```
+-- lab2-design-document.pdf --- Your Lab 2 design document.
+-- rv5s_eb.patch --- Your implementation.
+-- rv5s_hz.patch --- Your implementation.
```

Grading policy

Plagiarism is **strictly** forbidden. Contact the TAs if you are in trouble.

- 24% - Functionality of earlier branch resolving: correctness and minimum performance.
    - This part is composed of 8 testcases. And every testcase account for 3% points.
    - The detailed grading policy for each testcase is given in [Task Part 2](#2. Implementing Earlier Branch Resolving).
- 16% - Functionality of branch prediction: correctness and minimum performance.
    - When prediction is wrong, the processor should be able to recover to the correct PC.
    - The branch prediction policy should not be a naive one such as always taken or always not-taken.
- 10% - Design document: result analysis of earlier branch resolving.
- 10% - Design document: the branch prediction policy you used and how you update the predictor. Analyze the performance of your predictor.
- 10% - Design document: construction of attack case.
- 20% - Design document: question answering.
- 5% - Design document: others.
- 5% - Performance: the accuracy of your branch prediction.
    - Compare the results with or without branch prediction, you could get all 5 points if you can beat the performance bars (the minimum cycles of always predicting taken/not taken) of the performance testcases (matrix and ranpi).
        - performance_1_matrix.c: 642333 cycles
        - performance_2_ranpi.c: 667945 cycles

- 5% (bonus) - Global-history-based prediction is **correctly** implemented, and the corresponding question (the last) is answered.