Computer Architecture 2023 Fall Lab 3

1. Problem Description

In this lab, you are extending your Lab 2 to a pipelined CPU with a 2-bit dynamic global branch predictor. In Lab 2, we let the actual branch decision be determined at ID stage with an "equal" module added to the datapath. However, in this lab, we are replacing that "equal" module with a global branch predictor and utilizing the original resource of ALU (i.e., operation of subtraction) in the EX stage to examine the actual branch decision. We will test the correctness of your implementation by dumping the values specified in the provided reference "testbench.v" after each cycle.

1.1.2-bit Dynamic Branch Predictor

The major difference from Lab 2 is that we are replacing the branch decision at ID stage with a global branch predictor and verify the actual branch decision at EX stage using the original resource of ALU (i.e., operation of subtraction). All branches share the same predictor, that is, all branches receive the predictions from the same predictor and update the predictor with their actual branch decisions according to the state transition diagram shown in Figure 1.

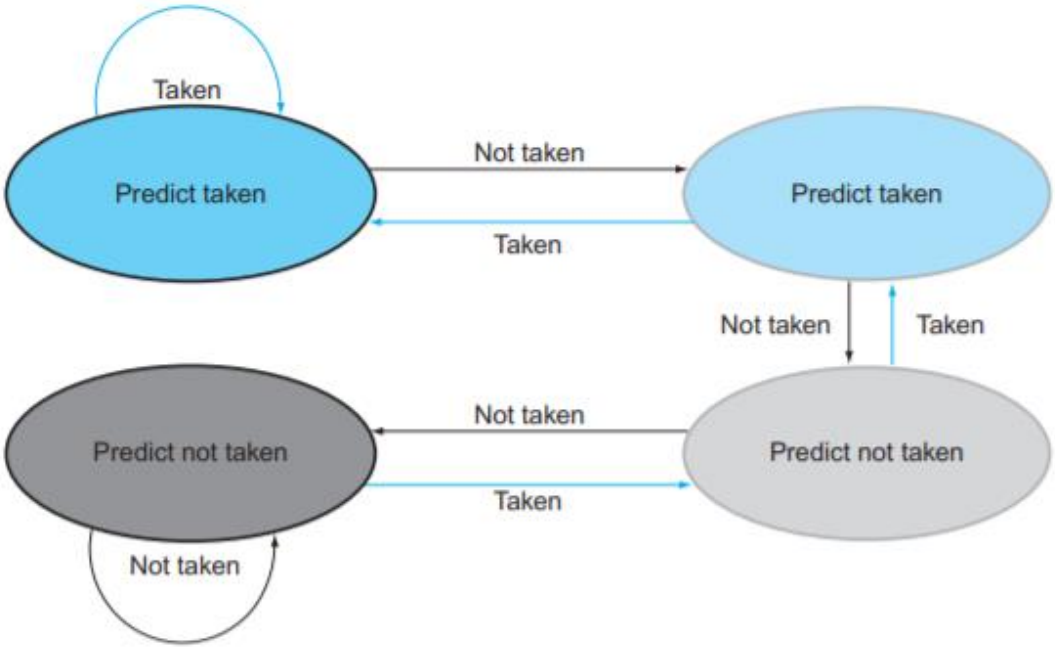

Figure 1: state transition diagram used for the 2-bit dynamic branch predictor

1.2.Instructions

Instructions needed to be support in this lab are the same as Lab 2, where the machine codes are as follows:

funct7	rs2	rs1	funct3	rd	opcode	function
0000000	rs2	rs1	111	rd	0110011	and
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000001	rs2	rs1	000	rd	0110011	mul
imm[11:0]		rs1	000	rd	0010011	addi
0100000	imm[4:0]	rs1	101	rd	0010011	srai
imm[11:0]		rs1	010	rd	0000011	lw
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw
imm[12,10:5]	rs2	rs1	000	imm[4:1,11]	1100011	beq

1.3.Input / Output Format

After you finish your modules and CPU, you should compile all of them including your "testbench.v". A recommended compilation command would be

```
$ iverilog  -o CPU.out *.v
```

Then by default, your CPU loads "instruction.txt", which should be placed in the same directory as CPU.out, into the instruction memory. "instruction.txt" is a plain text file that consists of 32 bits (ASCII 0 or 1) per line, representing one instruction per line. For example, the first 3 lines in "instruction.txt" are

```
0000000_00000_00000_000_01000_0110011 //add  $t0,$0,$0
000000001010_00000_000_01001_0010011  //addi $t1,$0,10
000000001101_00000_000_01010_0010011  //addi $t2,$0,13
```

Note that underlines and texts after "//" (i.e. comments) are neglected. They are inserted simply for human readability. Therefore, the CPU should take "00000000000000000000010000110011" and execute it in the first cycle, then "00000000101000000000010010010011" in the second cycle, and "00000000110100000000010100010011" in the third, and so on.

1.4. Modules You Need to Add or Modify

1.4.1. branch_predictor

The predictor gives the prediction of whether the upcoming branch is a taken or not taken action. Then, the predictor is updated by the actual branch decision examined at the EX stage according to the state transition diagram shown in Figure 1. Initial state of the branch predictor is strongly taken (Top left of Figure 1).

1.4.2. ALU_Control & ALU

You have to add additional control signals in the ALU_Control and ALU modules to support the examination of the `beq` condition.

1.4.3. Additional Flushes

Since the the actual branch decision is determined at the EX stage now, you need to add proper flushes to the pipeline registers (e.g., ID/EX) if any misprediction occurs.

1.4.4. Others

You can add more modules than listed above if you want. You are also required to change any detail to let your CPU perform correctly.

2. Report

2.1. Modules Explanation

You should briefly explain how the modules you implement work in the report. You have to explain them in human-readable sentences. Either English or Chinese is welcome, but no Verilog. Explaining Verilog modules in Verilog is nonsense. Simply pasting your codes into the report with no or little explanation will get zero points for the report.

Take “PC.v” as an example, an acceptable report would be:

> PC module reads clock signals, reset bit, start bit, and next cycle PC as input, and outputs the PC of the current cycle. This module changes its internal register “pc_o” at the positive edge of the clock signal. When the reset signal is set, PC is reset to 0. And PC will only be updated by next PC when the start bit is on.

And following report will get zero points.

> The inputs of PC are clk_i, rst_i, start_i, pc_i, and ouput pc_o.
>
> It works as follows:
>
> ```
> always@(posedge clk_i or negedge rst_i) begin
>     if(rst_i) begin
>         pc_o <= 32'b0;
> ```

```
    end
    else begin
        if(start_i)
            pc_o <= pc_i;
        else
            pc_o <= pc_o;
    end
end
```

2.2. Difficulties Encountered and Solutions in This Lab

Write down the difficulties if any you encountered in doing this lab, and the final solution to them.

2.3. Development Environment

Please specify the OS (e.g. MacOS, Windows, Ubuntu) and compiler (e.g. iverilog) or IDE (e.g. ModelSim) you use in the report, in case that we cannot reproduce the same result as the one in your computer.

3. Environment

Requirement: Docker

After you unzip the supplied file, we will get a directory in the following structure

- Lab3/
  - Lab3_spec.pdf
  - Lab3_slide.pdf
  - Lab3/
    - Makefile
    - code/
      - src/
        - <Implement your CPU here>
      - supplied/
        - <supplied codes, don't modify them>
    - Docker-compose.yml
    - Dockerfile
    - Judge.yaml
    - Testcases/
      - <sample testcases>
    - Log/
      - <logs>

You should modify the codes in "code/src" only and do not modify anything in "code/supplied".

The sample testcases are listed in the directory “testcases”.

After implemented your CPU, use $make or $docker-compose up to execute your code and get the log and report. Docker-compose will build up the environment for you.

You can find the log, text and waveform, in the directory “log” after each execution.

If you don’t have docker in your environment. The alternative is using the following commands

$cp testcases/instruction_n.txt instruction.txt

$iverilog -o cpu code/src/*.v code/supplied/*.v

$vvp cpu

And compare the file “output.txt” with the corresponding output_n.txt

We recommend develop under the provided environment, otherwise, please make sure your code run correctly under **ubuntu 22.04** with **iverilog=11.0-1.1**. **You will get 0 point if your code cannot be executed correctly on our environment.**

4. Submission Rules

Put all your Verilog codes into a directory named “codes”, then put “codes” and your report (should be named in format “studentID_lab3_report.pdf”) into a directory named “studentID_lab3”. **Note that you have to REMOVE Instruction_Memory.v, Data_Memory.v, Registers.v, PC.v, testbench.v, instruction.txt, output.txt, which are provided by TA, in your submission. And don’t modify testbench.v.** Please use ASCII-printable characters only. Finally, zip this directory, and upload this zip file onto NTU COOL before **12/25/2023 (Mon.) 23:59.**

In short, we should see a single directory like the following structure after we type

```
$ unzip studentID_lab3.zip
```

in Linux terminal:

- studentID_lab3/
  - studentID_lab3/codes
    - studentID_lab3/codes/CPU.v
    - studentID_lab3/codes/ALU.v
    - …

- `studentID_lab3/studentID_lab3_report.pdf`

Make sure you remove the following files before submission: Instruction_Memory.v, Data_Memory.v, PC.v, Registers.v, testbench.v, instruction.txt, output.txt. And make sure the testbench.v reads instructions from "instruction.txt" and output to "output.txt".

5. Evaluation Criteria

We will compile your program in following command:

```
studentID_lab3/codes/ $ iverilog  –o CPU.out *.v ../../*.v
```

, where `../../*.v` includes `Instruction_Memory.v, Data_Memory.v, PC.v, Registers.v`, testbench.v and the working directory is in your "codes/" directory. That is, some modules are provided outside the directory you submit.

5.1. Programming Part (80%)
- We will have a demonstration session on both labs. You have to come up to briefly explain how your program works to get the credits of the programming part.

5.2. Report (20%)

5.3. Other
- Minor mistakes (examples below) causing compilation error: -10 pts
  - wrong usage of "`include"
  - submitting unnecessary files (those provided by TA except CPU.v)
  - other mistakes that can be fixed within 5 lines
- Major mistakes causing compilation error: 0 pts on programming part
- No show up at demonstration: 0 pts on programming part
- Wrong directory format: -10 pts
- Wrong I/O paths: -10 pts
- Late submission: -10 pts per day
- Plagiarism: 0 pts.

email to eclab.ca.ta@gmail.com if you have any questions.