# Computer Architecture 2021 Fall Lab 2

TA: 陳炫均

## 1. Problem Description

In this homework, you are going to extend your Lab 1 to have memory hierarchy. In Lab 1, we still assume memory read/write can be done in a cycle. However, in reality, data memory is several order slower than CPU cycle. In this lab, we use a 1KB L1 cache, whose read/write latency is still the same as CPU cycle, with a larger off-chip data memory, which requires 10 cycle for a read/write operation. We will examine the correctness of your implementation by dumping the value of each register and data memory after each cycle.

### 1.1. System Architecture

The major difference from Lab 1 is that we have an off-chip module in this lab, Data_Memory. CPU has its cache within it; however, Data_Memory does not reside in CPU. Instead, it is connected to CPU in testbench. As in Figure 1, the cache controller carrys several control signals and data over CPU to the off-chip memory.

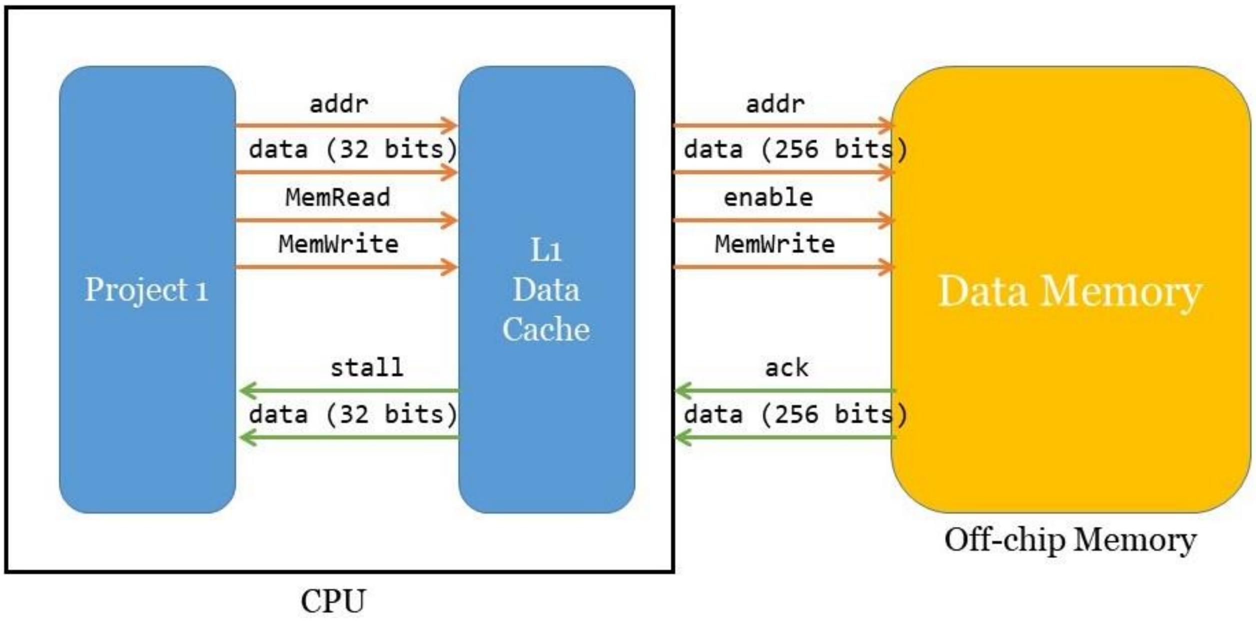

*Figure 1 System Architecture*

For short, you have to replace the Data_Memory in your Lab 1 with the cache controller, then complete the cache controller so that it can correctly handle cache hit and miss to decrease the latency of memory read/write.

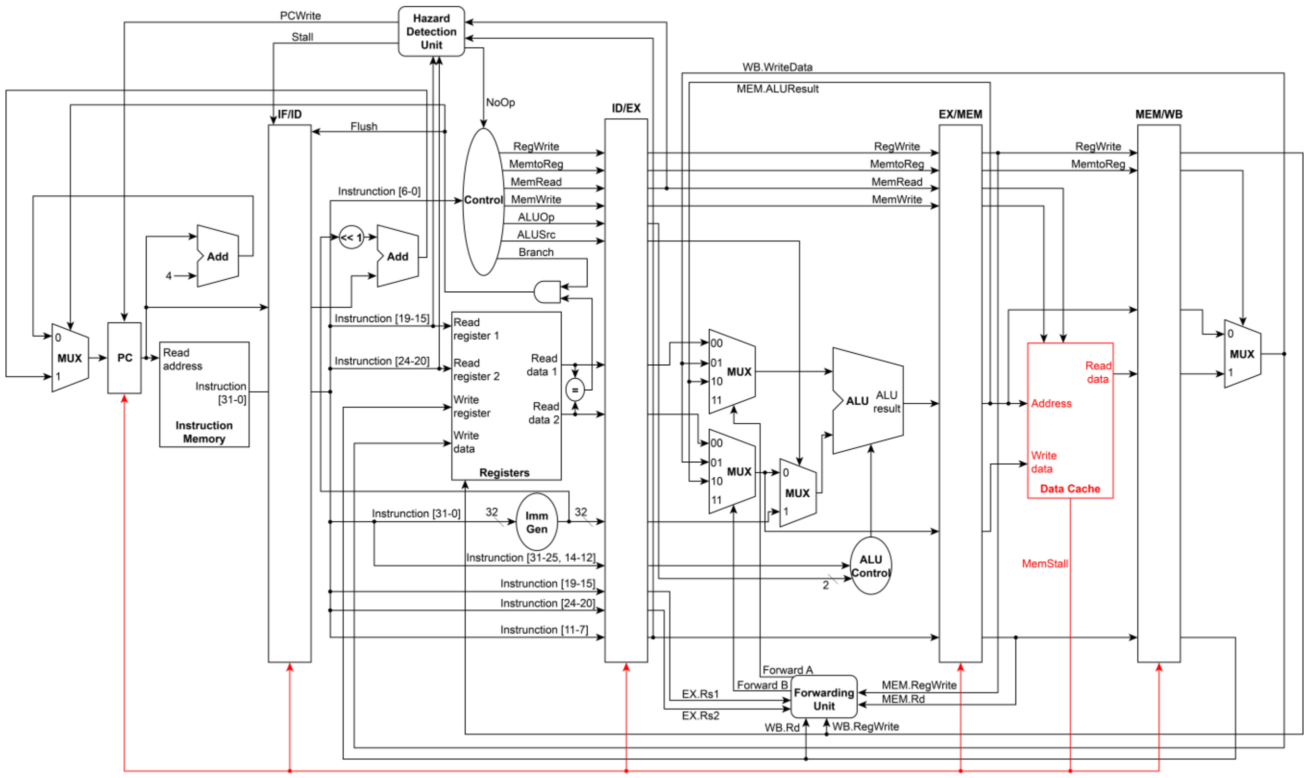

*Figure 2 Datapath: you need to replace Data_Memory with dcache, and connect MemStall signal to sequential circuit elements*

## 1.2. Specification of Cache and Data Memory

In this lab, we still have 32-bit memory address, and the registers are also 32-bit. The cache capacity is 1KB, and 32-byte (256-bit) per cache line. The cache is two-way associative, with replacement policy being LRU (least recently used). Therefore, for the 32-bit address, the cache controller will treat it as 23-bit tag, 4-bit block index, and 5-bit byte offset. Note that you do not have to handle unaligned read/write in this lab. The cache applies “write back” policy to handle write hit, and “write allocate” as its write miss policy.

| | hit | miss |
|---|---|---|
| read | Fetch data from cache | Evict a block by LRU policy. Then bring the data from data memory into the cache. |
| write | Write only to the cache, and set up the dirty bit. | Evict a block by LRU policy. Then bring the data from data memory into the cache. Write to the cache, and set up the dirty bit. |

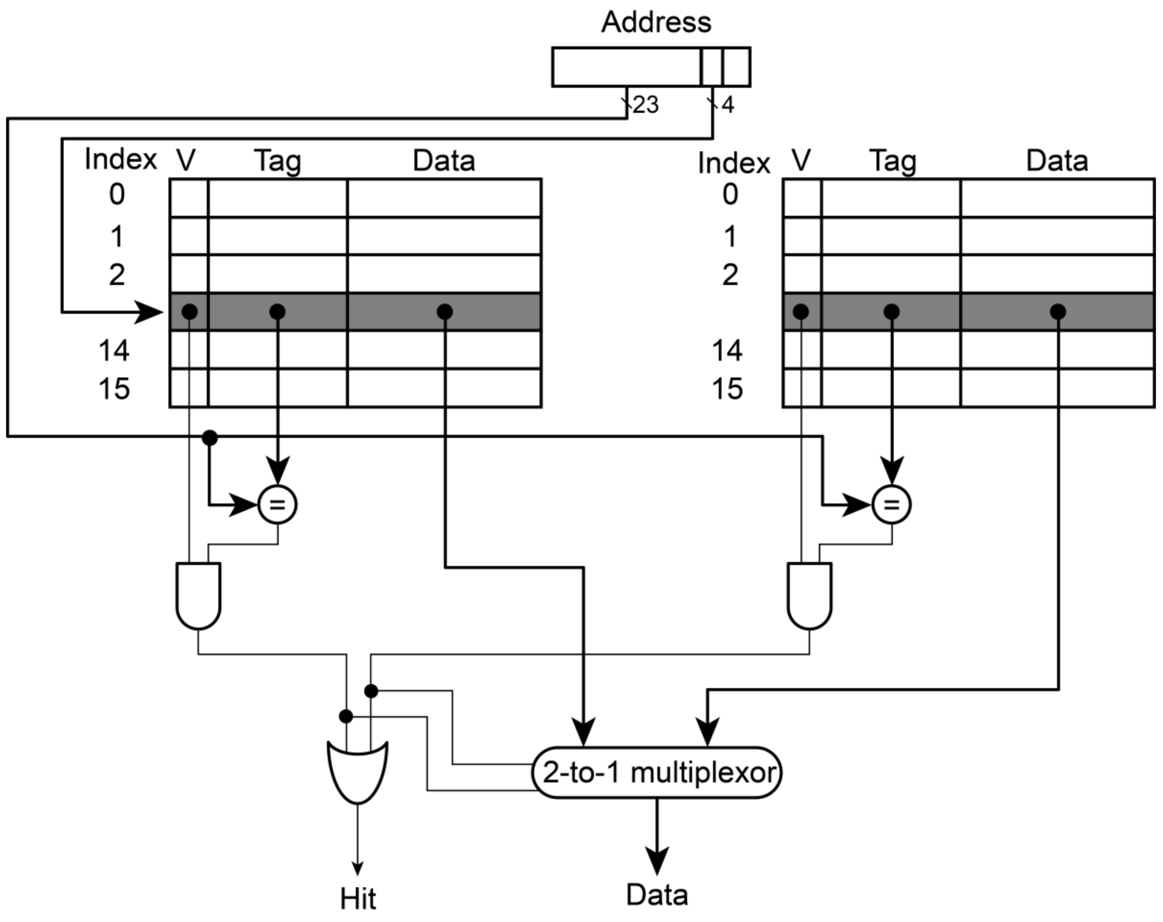

*Figure 3 2-way associative cache*

The access latency of off-chip Data_Memory is 10 cycles. When the enable signal of Data_Memory is turned on, the Data_Memory will start accessing the data, and send back an ack signal and data of corresponding address after 10 cycles.

## 1.3. Instructions (same as Lab 1)

| **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 |

## 1.4. Input / Output Format

Besides the modules listed above, you are also provided "testbench.v" and "instruction.txt". After you finish your modules and CPU, you should compile all of them including "testbench.v". A recommended compilation command would be

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

Then by default, your CPU loads "instruction.txt", which should be placed in the same directory as CPU.out, into the instruction memory. This part is written in "testbench.v". You don't have to change it. "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.

Also, if you include unchanged "testbench.v" into the compilation, the program will generate a plain text file named "output.txt", which dumps values of all registers and data memory at each cycle after execution. The file is self-explainable.

A difference from Lab 1 is that there are two output files in this lab, `output.txt` and `cache.txt`. `output.txt` dumps values of registers and some selected data memory at each cycle. And `cache.txt` records each load/store operations, and whether it is a hit or miss.

Note that your output do not have to be 100% the same as the one of our reference program. **<u>We will only check the values of the last cycle in `output.txt`, and numbers and orders of hit and miss in `cache.txt`.</u>**

## 1.5. Modules You Need to Add or Modify

### 1.5.1. dcache_controller

The controller determines whether the upcoming load/store is a hit or miss. Then according to the write back and write allocate policy, properly interact with CPU and Data_Memory.

### 1.5.2. dcache_sram

This module stores tags and data of the cache. You should add some additional codes to support 2-way associative and LRU replacement policy.

### 1.5.3. testbench

As in Lab 1, You have to initialize reg in your pipeline registers before any instruction is executed. If you initialize your pipeline registers in testbench in Lab 1, please remember to copy those codes into testbench here. Except for registers initialization, please do not change the output format (`$fdisplay` part) of this file.

1.5.4. Others

You can add more modules than listed above if you want. You are free to change some details as long as your CPU can perform correctly.

1.5.5. CPU

Replace the Data_Memory part in your Lab 1 with dcache_controller.

## 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. You have to write more detail than Section 1.5.

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
```

You can draw a FSM (Finite State Machine) diagram to explain how your cache works.
**You need to explain in detail for your cache controller, which is the core of this lab.**

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 20.04) 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. Submission Rules

Put all your Verilog codes into a directory named "codes", then put "codes" and your report (should be named in format "studentID_lab2_report.pdf") into a directory named "studentID_lab2". **Note that you have to REMOVE Instruction_Memory.v, Registers.v, PC.v, instruction.txt, output.txt, which are provided by TA, in your submission.** Finally, zip this directory, and upload this zip file onto NTU COOL before **1/18/2022 (Tue.) 23:59.**

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

```
$ unzip studentID_lab2.zip
```

in Linux terminal:

- `studentID_lab2/`
    - `studentID_lab2/codes`
        - `studentID_lab2/codes/CPU.v`
        - `studentID_lab2/codes/ALU.v`
        - …
    - `studentID_lab2/ studentID_lab2_report.pdf`

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

## 4. Evaluation Criteria

We will compile your program in following command:

```
studentID_lab2/codes/ $ iverilog *.v ../../*.v –o CPU.out
```

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

4.1. Programming Part (80%)

We will have several instruction files to test the following features.

- Basic pipeline operations (as in lab 1): 20%
- Read miss
    - Detection (10%)
    - Handle (10%)
- Write miss
    - Detection (10%)
    - Handle (10%)
- Read hit (10%)
- Write hit (10%)
- We will have a demonstration session on both labs. The time slots will be announced around the deadline of this lab. You have to come up to briefly explain how your program works to get the credits of the programming part.

4.2. Report (20%)

4.3. Other

- Minor mistakes (examples below) causing compilation error: -10 pts
    - wrong usage of "`include"
    - submitting unnecessary files (listed in Section 3)

        - 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.