Lab2 Report

B07901069 電機四 劉奇聖

Modules Explanation

`Adder.v`, `MUX2.v`, `MUX4.v`, `Sign_Extend.v`, `Control.v`, `ALU_Control.v`, `ALU.v`, `Forwarding_Unit.v`, `Hazard_Detection.v` are the same as lab 1.

`CPU.v`

Compared with lab 1, CPU modules have some input/output to connect to the off-chip memory, including mem_data_i, mem_ack_i, mem_data_o, mem_addr_o, mem_enable_o, mem_write_o. It also replace Data_Memory with dcache_controller and connect cpu_stall_o to PC and pipeline registers.

`Pipeline_Registers.v`

Compared with lab 1, ID_EX_Registers, EX_MEM_Registers, MEM_WB_Registers modules have one more stall_i signal and they update their states only when stall_i is zero.

`dcache_controller.v`

dcache_controller module have 1 bit clock signal and 1 bit reset signal inputs and interfaces to data memory and CPU.

- first TODO (r_hit_data): read hit data is simply the output of the SRAM, so I assign it with sram_cache_data.
- second TODO (cpu_data): We need to convert 256-bit to 32-bit. Since memory accesses are aligned, cpu_offset must be multiple of 4. Therefore we can simply multiply it with 8 to get the index of the least significant bit and take 32 bits from r_hit_data starting at that index.
- third TODO (w_hit_data): As explained previously, (cpu_offset * 8) will be the index of the least significant bit. Although we only need to write 32 bit data, we can only write 256 bit at a time. So we need to assign w_hit_data with r_hit_data first and then replace the 32 bit data starting at index (cpu_offset * 8) with cpu_data_i.
- TODOs in controller FSM
    - frist_TODO: Since next state is WRITEBACK, mem_enable should be 1 and mem_write should be 1. Cache does not need to be updated, so cache_write is 0. write_back should be 1 since next state is WRITEBACK.
    - second TODO: Since next state is READMISS, mem_enable should be 1 to allow us read data from memory. mem_write should be 0 since we only need to read the memory. cache_write should be 0 since we need to wait for memory data is ok. write_back should be 0 since the next state is not WRITEBACK.
    - third TODO: The next state is READMISSOK. mem_enable and mem_write should be 0 since we do not need to read or write the memory. cache_write should be 1 since we already bring the data from the memory and we need to write it to the cache. write_back should be 0 since the next state is not WRITEBACK.
    - forth TODO: Since the next state is IDLE, mem_enable, mem_write, cache_write, write_back should be all 0.

- fifth TODO: Since next state is READMISS, mem_enable should be 1 to allow us read data from memory. mem_write should be 0 since we only need to read the memory. cache_write should be 0 since we need to wait for memory data is ok. write_back should be 0 since the next state is not WRITEBACK.

`dcache_sram.v`

dcache_sram module have 1 bit clock signal and 1 bit reset signal inputs and interface to the dcache_controller. Besides the variables provided by TA, I add 1 bit hit_0 and 1 bit hit_1 wire and array of 1 bit registers of length 16 called lru. hit_0 is 1 when `tag[addr_i][0][24]` is 1 (valid bit is 1) and `tag[addr_i][0][22:0]` is equal to `tag_i[22:0]` (tag matched). hit_1 is 1 when `tag[addr_i][1][24]` is 1 (valid bit is 1) and `tag[addr_i][1][22:0]` is equal to `tag_i[22:0]` (tag matched). hit_o is 1 if hit_0 is 1 or hit_1 is 1.

`lru` array represents which block to be replaced in the same cache line. When hit_0 is 1, tag_o is `tag[addr_i][0]` and data_o is `data[addr_i][0]`. When hit_1 is 1, tag_o is `tag[addr_i][1]` and data_o is `data[addr_i][1]`. When hit_0 and hit_1 are both 0, we find the next block to be replaced is 0 or 1 using `lru[addr_i]` and assign tag_o and data_o correspondingly.

In the always block, if reset signal is 1, then we intitialize all registers to 0. If enable_i is 1 and write_i is 1, if hit_0 is 1, then we write tag_i and data_i to `tag[addr_i][0]` and `data[addr_i][0]` and set `lru[addr_i]` with 1 since block 0 is recently used. If hit_1 is 1, then we write tag_i and data_i to `tag[addr_i][1]` and `data[addr_i][1]` and set `lru[addr_i]` with 0 since block 1 is recently used. If hit_0 and hit_1 are both 0, then we must evict a block by LRU policy, so we assign tag_i and data_i to the block indicated by `lru[addr_i]` and invert `lru[addr_i]` since it is recently used, which means the other is least recently used.

If enable_i is 1 but write_i is 0, then it means that we are reading the cache. If hit_0 is 1, then block 0 is currently used and we have to set `lru[addr_i]` to 1. If hit_1 is 1, then block 1 is currently used and we have to set `lru[addr_i]` to 0.

Difficulties Encountered and Solutions in This Lab

Difficulty 1

The testcases provided by TA do not test the write back functionality.

Solution 1

I wrote custom testcases to test the write back functionality.

Difficulty 2

In the previous lab, there are serveral stages. We can test the correctness of our program before moving to the next stage. In this lab, we need to finish all modules before we test the correctness. I think it is harder to debug.

Solution 2

Use gtkwave to debug and carefully trace each signal.

Development Environment

My OS is "Linux Mint 20.2 Cinnamon". The compiler is `iverilog` version 10.3. I use VSCode to write this homework.