Computer Architecture Project 2 Report

B07902031 黃永雯、B07902043 許喬茵、B07902069 李哲宇

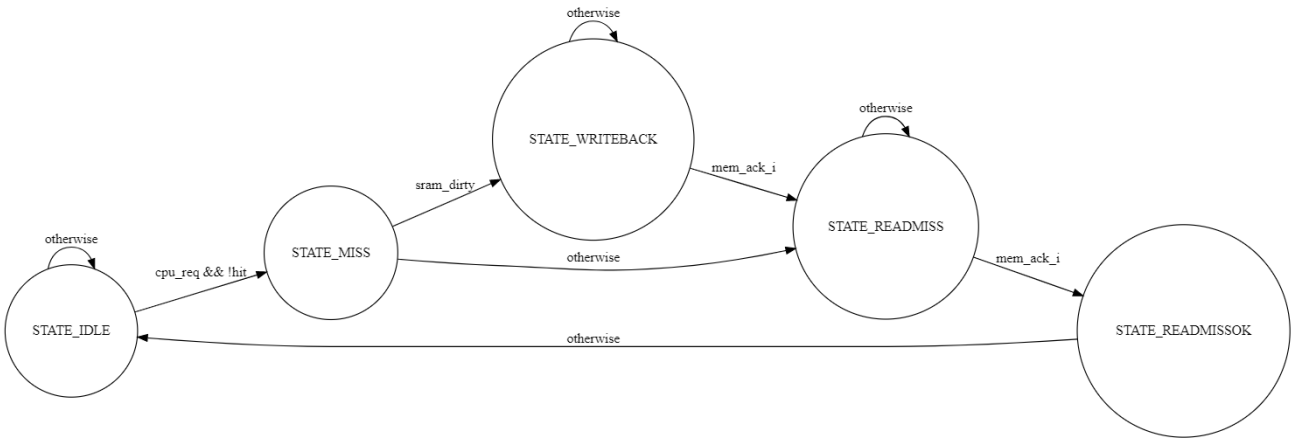
Modules Explanation

dcache_controller.v

This module interacts with CPU and Data Memory, determining the hit/miss.

`r_hit_data` will be set to the value of `sram_cache_data` if `hit` equals to 1; otherwise, it will be set to `mem_data_i`, we set the register `cpu_data` to `r_hit_data[cpu_offset*8+:32]` (which is 32 bits after the `cpu_offset` *8) whenever `cpu_offset` changes or `r_hit_data` changes (reading data from 256-bit to 32-bit). And we set `w_hit_data` to `r_hit_data` and `w_hit_data[cpu_offset*8+:32]` to the cpu data input when we write data from 32-bit to 256-bit.

Then we decide the values of four variables: write_back, mem_write, mem_enable and cache_write, and decide the state of different situations acccording to a state diagram shown as below:



There are different states that are needed to be considered:

- When we reset, the state is `STATE_IDLE`, and the four variables are all set to 0.
- When the state is `STATE_MISS`, we will check whether the `sram_dirty` is on. If the sram dirty bit equals to 1, we set `write_back, mem_write, mem_enable` to 1 (since in this case, it will write back), and `cache_write` is set to 0, then the state is changed to `STATE_WRITEBACK`. On the other hand, if the sram dirty bit equals to 0, we only set

`mem_enable` to 1, and `write_back, mem_write, cache_write` are all set to 0, then the state is changed to `STATE_READMISS`.

- When the state is `STATE_READMISS`, we check whether the `mem_ack_i` bit is on. If the bit is on, we will set `cache_write` to 1 and `mem_enable` to 0, then the state will be changed to `STATE_READMISSOK`.
- When the state is `STATE_READMISSOK`, we set all four variables to 0, and the state change back to `STATE_IDLE`.
- When the state is `STATE_WRITEBACK`, we check whether the `mem_ack_i` is on. If is, we set `mem_enable` to 1, and `mem_write`, `cache_write`, `write_back` to 0.

dcache_sram.v

This module will get signals from dcache_controller for tags and data storing of the cache, and also deal with 2-way aassociative and LRU replacement.

We let a signal `block_hit` indicate if any of the two blocks in a set hits.
That is, `block_hit[j]`, where `j` = 0, 1, will be set to 1 if the value of the input tag `tag_i` equals to that of the tag stored in the cache block `j` (`tag[addr_i][j]`). Of course, `tag[addr_i][j]` must be valid.

In the always block, if it is to write data to cache blocks, i.e. the input of enable and write equals to 1, it writes to the cache.
It will first check the value of `hit_o`. If `hit_o` equals to 1, indicating write hit, it will then check which side hits. In detail, if block 0 hits (`block_hit[0] == 1`), the value of correspoding data in block 0 (`data[addr_i][0]`) will be set to `data_i`, and the valid bit and dirty bit will both be set to 1. On the other hand, if block 1 hits (`block_hit[1] == 1`), the value of corresponding data in block 1 (`data[addr_i][1]`) will be set to `data_i`, and the valid bit and dirty bit will both be set to 1.
If the value of `hit_o` equals to 0, we got a read miss / write miss and we should choose either block 0 or block 1 to be replaced according to LRU policy.
To implement LRU, we use an array `last` to record the most recently modified block of each set. If `last[addr_i]` is 1, which implies the last used one is block 1 and that block 0 is the one that least recently used. So we set `data[addr_i][0]` to `data_i`, `tag[addr_i][0][22:0]` to `tag_i[22:0]` (the tag part) and set `last[addr_i]` to 0.
If `last[addr_i]` is 0, we set `data[addr_i][1]` to `data_i`, `tag[addr_i][1][22:0]` to `tag_i[22:0]` (the tag part) and set `last[addr_i]` to 1.
In both the cases the valid bit will be set to 1 and the dirty bit will be set to 0.

We also consider the read hit case, if block 0 hits, the last used will be set to block 0, i.e., `last[addr_i] <= 0`. On the other hand, if block 1 hits, the last used will be set to block 1, i.e., `last[addr_i] <= 1`.

As for the output ports, `data_o` and `tag_o` depend on whether we got a hit or a miss. First we check if `enable_i` is 0. If it is, then both the outputs will be assigned to 0. If not, then we go on.
If block 0 hits, then `data_o` will be assigned to the data in block 0 (`data[addr_i][0]`) and `tag_o` will be assigned to the tag in block 0 (`tag[addr_i][0]`).
If block 1 hits, then `data_o` will be assigned to the data in block 1 (`data[addr_i][1]`) and `tag_o` will be assigned to the tag in block 1 (`tag[addr_i][1]`).
Otherwise we got a miss, `data_o` and `tag_o` will be assigned to the corresponding values in the block which is to be replaced, i.e., `data_o = data[addr_i][~last[addr_i]]` and `tag_o = tag[addr_i][~last[addr_i]]` .
Lastly, `hit_o` will be assigned to 1 if `enable_i` is 1 and either block 0 or block 1 hits; otherwise, it will be assigned to 0.

CPU.v

`CPU` Module connects modules with wires, which means that some of the output will be the input of other modules. The modifications are the wire connections of `dcache_controller` module and some wires in the pipeline latches that are connected to `dcache_controller` module.

Pipeline Latches

The four pipeline latches, `IF_ID` , `ID_EX` , `EX_MEM` and `MEM_WB` is as the same as they are in Project 1, except for all of them will stall (do nothing) when their new input port, `MemStall_i` equals to 1.

testbench.v

In addition to what has been written by TAs, we also added the initialization of all registers in the 4 pipeline latches, i.e., we set all of their values to 0. Moreover, the register `last[0:15]` in `dcache_sram` is set to 1, to let the first time deal with block 0 (the left block).

Members & Teamwork

- B07902031 黃永雯：codes of dcache_sram and dcache_controller, debugging, report writing
- B07902043 許喬茵：codes of CPU and dcache_sram, debugging, report writing
- B07902069 李哲宇：codes of dcache_controller, debugging, report writing

Difficulties Encountered and Solutions in This Project

- The first problem we met is that we used the whole tag in `dcache_sram` . We then found out that the component of the tag is `[valid bit, dirty bit, tag]` . Therefore, we only

use `tag_i[22:0]` when the value of tag memory needed to be changed. And we give the valid bit and dirty bit by ourselves.

- The second problem we met is that we forget to consider the write miss condition in `dcache_sram` module. The probelm is then fixed by checking the value `hit_o`.
- Another problem we met is that when both block 0 and block 1 miss, we give the wrong value of `data_o` and `tag_o`. We then set them to the corresponding values in `~last[addr_i]` when the two blocks miss to solve this problem.

Development Environment

The OS we use include MacOS and Windows, and the compiler we use is iverilog.