Computer Architecture Project 2 Report

Team: 單生狗聯盟(WannaLaugh)

2.1 Module Explanation

dcache_controller.v

In **dcache_controller.v**, there are
8 inputs:
  clk_i, **rst_i**, **mem_ack_i**, **cpu_MemRead_i** and **cpu_MemWrite_i** which all 1 bit.
  cpu_data_i and **cpu_addr_i** which both 32 bits.
  mem_data_i which is 256 bits.
6 outputs:
  mem_enable_o, **mem_write_o** and **cpu_stall_o** which all 1 bit.
  mem_addr_o and **cpu_data_o** which both 32 bits.
  mem_data_o which is 256 bits.

There are several codes that needed to add to **dcache_controller.v**:

1. **r_hit_data**

```
assign r_hit_data = (hit) ? sram_cache_data : mem_data_i;
```

where
  sram_cache_data is the data read from the **dcache_sram.v**
  mem_data_i is the data read from the **Data_Memory.**

If **hit** == 1, it means that the data is in the cache, the 256 bits data should be chosen from the cache. Otherwise, data should be chosen from the data memory.

2. **cpu_data**

```
cpu_data <= r_hit_data[(cpu_offset*'d8) +: 'd32];
```

where **cpu_data** is the read data that requested.

Since the block size is 32 bytes (256 bits) and the data requested is in size of 32 bits: $\frac{256}{32} = 8$.

Thus, the block offset should multiply by 8 to get the base offset of the data and extend by 32 bits to get the requested data.

3. **w_hit_data**

```
w_hit_data <= r_hit_data;
 w_hit_data[(cpu_offset*'d8) +: 'd32] <= cpu_data_i;
```

where **w_hit_data** is the write data that should write to the cache or the data memory.

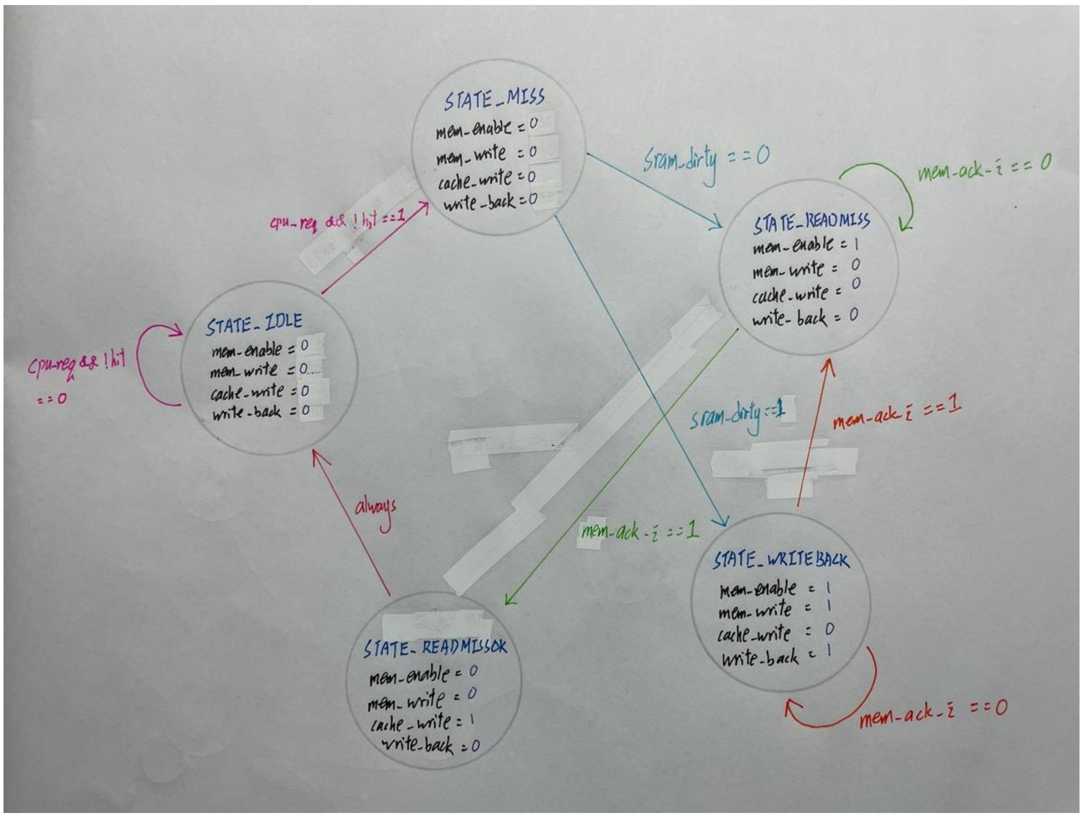
Since by TA code, the write process will be
write allocate: write miss = read miss + write hit

Thus, the 256 bits data that needed to write will first held in **r_hit_data**. Passed 256 bits **r_hit_data** to **w_hit_data** and write the modified 32 bits data to **w_hit_data**.

The method of base offset calculation is the same as in **section "2. cpu_data"**

w_hit_data is now ready to be write to cache or data memory.

4. **controller** (actions and states)
Following is the FSM for the controller:



When changing from a state to another, the four variables should also change to the value in the next state as shown in the image above.
Note that after entering **STATE_READMISSOK**, the next state always be **STATE_IDLE**.

There are four variables used to determine the actions of dcache_sram.v and Data_Memory.v.

mem_enable
: used to determine whether the memory access on data memory is allowed.

mem_write
: used to determine whether the write action is allow on data memory when there is data needed to write to the data memory.

cache_write
: used to determine whether the write action is allow on cache when there is data needed to write to the cache.

write_back
: used to determine whether there is a write back action due to the replacement of a dirty cache in the cache.

There are five states with different value for the four variables stated above that complete the function of controller.

STATE_IDLE

```
mem_enable  <= 1'b0;
mem_write   <= 1'b0;
cache_write <= 1'b0;
write_back  <= 1'b0;
```

Wait for request, no actions needed in cache and data memory.

If cpu_reg && !hit == 1, means that there is a request (either read or write) and there is a miss. Thus, the controller state will change from STATE_IDLE to STATE_MISS.

Else the controller state remains in STATE_IDLE.

STATE_MISS

```
mem_enable  <= 1'b0;
mem_write   <= 1'b0;
cache_write <= 1'b0;
write_back  <= 1'b0;
```

No action needed in cache and data memory.

If sram_dirty == 1, means that there is dirty cache been replaced by the LRU policy and write back action is needed. Thus, the controller state will change from STATE_MISS to STATE_WRITEBACK.

Else the controller state will change from STATE_MISS to STATE_READMISS.

STATE_READMISS

```
mem_enable  <= 1'b1;
mem_write   <= 1'b0;
cache_write <= 1'b0;
write_back  <= 1'b0;
```

When in STATE_READMISS, mem_enable = 1’ b1 in order to allow memory access in data memory to read data from data memory.

If mem_ack_i == 1, means that the data is already read from the data memory. Thus, the controller state will change from STATE_READMISS to STATE_READMISSOK.

Else, the controller state remains in STATE_READMISS until the data is read from data memory and an ACK is receive from data memory.

STATE_READMISSOK

```
mem_enable  <= 1'b0;
mem_write   <= 1'b0;
cache_write <= 1'b1;
write_back  <= 1'b0;
```

When in STATE_READMISSOK, cache_write = 1’ b1 so that the data can be written into the cache

The controller state is then change from STATE_READMISSOK to STATE_IDLE and wait for next request.

STATE_WRITEBACK

```
mem_enable  <= 1'b1;
mem_write   <= 1'b1;
cache_write <= 1'b0;
write_back  <= 1'b1;
```

When in STATE_WRITEBACK,

mem_enable = 1’ b1 so that memory access is allowed in data memory

mem_write = 1’ b1 so that that data in the dirty cache is write to the data memory

write_back = 1’ b1 to indicate write back action occur.

If mem_ack_i == 1, means that the data is already write to the data memory. Thus, the controller state change from STATE_WRITEBACK to STATE_READMISS.

Else the controller state remains in STATE_WRITEBACK until the data is write to data memory and an ACK is receive from data memory.

dcache_sram.v

變數解釋

這個 module 有 7 個 input 和 3 個 output，input 分別是 clk_i,rst_i,addr_i,tag_i,data_i,enable_i 和 write_i,output 分別是 tag_o,data_o 和 hit_o。這份 module 會使用 tag 和 data 這兩個 register 去儲存 cache 裡的 tag 和 data，然後使用 LRU 去紀錄每一個 set 當前的 LRU 來實作 2-way associative cache。LRU 是個 16bit 的 register，每一個 bit 的 value 代表著每一個 set 的 LRU 是第 0 筆資料還是第 1 筆資料。

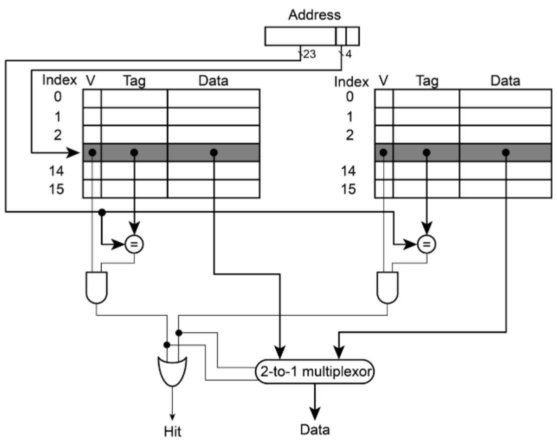
初始化

當 rst_i 為 1 的時候，tag,data 跟 LRU 會初始化為 0。

write data

當 enable_i 和 write_i 都是 1 的時候，根據 addr_i 查詢到的 tag[addr_i]如果跟 tag_i 相同，就會把 data_i 寫進去 data[addr_i]，寫入 data 的同時，也會把相對應的 tag 替換，並且 set up dirty bit，否則就會寫入在 data[addr_i]的 LRU 的那筆 data。

read data



根據上圖，我們可以透過 addr_i 找到 tag[addr_i]的 23-bit tag 和 1-bit valid bit 來決定是否 hit，如果 hit 變把 hit_o 設成 1，tag_o 設成 hit 的 tag，data_o 設成 hit 的那筆 data，否則，我們就把 hit_o 設成 0，tag_o 設成 LRU 的 tag，data_o 設成 LRU 的 data。此外，每次 hit 了之後我們都會更新 LRU。

CPU.v

In project 2, CPU.v have

Five inputs:

clk_i, rst_i, start_i and mem_ack_i which is 1 bit.

mem_data_i which is 256 bits.

Four outputs:

mem_data_o which is 256 bits.

mem_addr_o which is 32 bits.

mem_enable_o and mem_write_o which is 1 bit.

A new module, **dcache_controller dcache()** is added and connected with wires to interact with **Data Memory** and **dcache_sram**.

Register_IFID.v & Register_IDEX.v & Register_EXMEM.v & Register_MEMWB.v

In project 2, there is a small modification for all four pipeline registers.
Every pipeline register has a new input called **stall_i**.
If **stall_i** == 1, then the register should undergo stall operation and all the outputs maintain the same value as outputs in previous cycle.

2.2. Members & Teamwork

Member	Works
B07902091 周俊廷	主要負責 dcache_controller.v。 更新 pipeline registers（共 4 個） CPU.v 最終修正 撰寫上述 module 報告
B07902089 李智源	負責 dcach_sram.v 撰寫 dcach_sram 報告
B07902059 陳君翰	Debug、資料測試、報告彙整
B06902098 李恩慈	CPU.v 初步撰寫（由 project 1 CPU.v 改寫）

2.3. Difficulties & Solutions

B07902091 周俊廷	1. read data 和 write data 的查找方式： 在 block offset 的使用方式上產生了困難 a. 錯誤：$cpu\_offset * d'32 +: 'd32$ b. 正確：$cpu\_offset * d'8 +: 'd32$ 2. state 的改變： 對 mem_enable, mem_write, cache_write 和 write_back 的值理解錯誤 a. 錯誤：根據目前的 state 去更改這四個變數 b. 正確：把四個變數更改成 next state 時的值 在本報告中的 dcache_controller.v 的部分有作更詳盡的說明，並附上 Finite State Machine 圖做為參考
B07902089 李智源	1. 難以直觀地理解助教程式碼，因此不了解 controller 如何運作，導致不了解 sram 具體要處理什麼 2. 判斷 hit 時 tag 的比對： a. 錯誤： 25bit 全部比對 b. 正確：只需使用 tag[22:0]，共 23bit 即可

	3. LRU 的實作問題： 應在每次 CPU read 時即更新
B07902059 陳君翰	雖自認對於實作方法有足夠理解，但難以直觀地理解助教程式碼，因此有很長一段時間不知從何處下手
B06902098 李恩慈	none

2.4. Development Environment

The OS used : CSIE Workstation

The Compiler used : iverilog