2019 Computer Architecture Project 2

Member

- 資工三 b06902021 吳聖福
    - Connect all the modules
    - Implement unaligned access
- 資工三 b06902026 吳秉柔
    - Implement cache controller
- 資工三 b06902093 王彥仁
    - Implement tag comparator and SRAM interface

Design & implementation

We used our Project 1 code in most of this project. Thus, we only describe the new or modified modules below. For other details, please refer to the report of Project 1 (inside the branch `hw4`).

Execution environment

Run `make` instead of `iverilog *.v` to compile the modules. Otherwise, the compilation may fail because of the order of modules (some macros for memory sizes are placed in `CPU.v`).

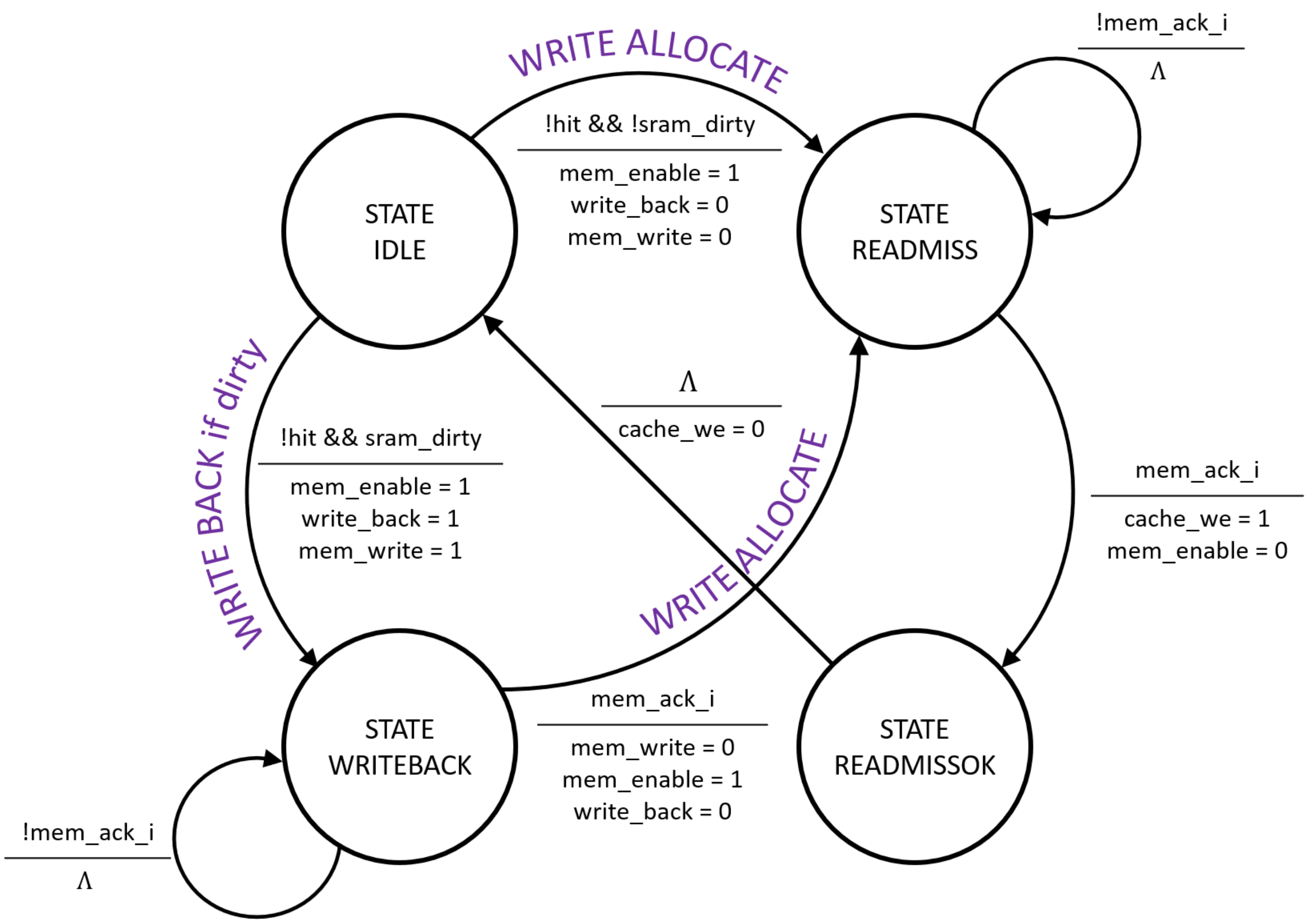
`DCache_SRAM.v`
We merged the provided `dcache_tag_sram` and `dcache_data_sram` modules into a module named `DCache_SRAM`.

`Data_Memory.v`
We merged two `always` clauses for simpler code. We also deleted the unnecessary assignment to output if `write_i` is set.

`DCache_Top.v`

Cache controller FSM

The following graph shows the FSM of our cache controller:



Our cache controller containes four states:

- `STATE_IDLE`: This state waits for read or write request from the CPU. If the read/write is a hit, the data is fetched/written. If the read/write is a miss and the cache line is clean, the FSM simply moves to `STATE_READMISS`. If the cache line is dirty, the dirty block should be written back to memory, so the FSM moves to `STATE_WRITEBACK`.
- `STATE_READMISS`: This state waits for the memory to allocate a new block to the cache. The cache controller waits until the memory send an ACK signal and the FSM moves to `STATE_READMISSOK`.
- `STATE_READMISSOK`: The allocation of the new block is finished. The cache controller sets some signals to default, and the FSM returns to `STATE_IDLE`.

- `STATE_WRITEBACK`: This state waits for the memory to write back the dirty cache block. The cache controller waits until the memory sends an ACK signal and the FSM moves to `STATE_READMISS`.

(The cache hits if the tag of the request data is equal to the tag of the cache line.)

SRAM interface

In a cache read, the result is some consecutive bits of the SRAM output; in a cache write, the data to be written to the SRAM is the SRAM output with some consecutive bits replaced by the new data. The bits to be read/write is decided by the cache offset.

Unaligned access

If a cross-cache-line access (an unaligned memory access covers two cache lines) happens, two cache operations must be done to read/write the desired data. Thus, we implemented it as following:

- Add a wire `is_unalign` to indicate whether the access is a cross-cache-line access.
- Add a state bit `unalign_second` to mark the cache line being accessed (`0` represents the cache line correponding to the address, and `1` represents the next cache line).
- Once the FSM is in the state `STATE_IDLE`, we flip the `unalign_second` bit if the cache hits and `is_unalign=1`. Thus, if the first cache access finished, it will start the second cache access at the next cycle.
- The `mem_stall` flag will be set if `is_unalign=1` and `unalign_second=0` (which indicates that the cross-cache-line access haven't finished).

`testcase/generate_rand_insr.cpp`

We modified our testdata generator to generate load/store instructions accessing positions near cache line boundaries to test if the cache replacement and unaligned access mechanism works properly.

The original `mytest.v` was moved to `testcase/`, but it still needs to be compiled by `make` inside `code/`.

Difficulties encountered and solutions of this projects

We got some minor bugs during implementation:

- Accidentally deleted the code setting `write_back` on writing back dirty cache lines, which results in an infinite loop when simulating. (However, we didn't figure out the exact reason that caused the infinite loop.)
- Forgot to set `cache_we` when the new block is allocated, so the tag bits and dirty bit aren't updated.

The bugs are easy to fix once we figured it out.