Politecnico di Milano
AA 2018-2019

Computer Science and Engineering

Embedded Systems

Cache Design

Dario Ferrari - 900275

Andrea Mazzeo - 895579

1 CACHE ARCHITECTURE

The implemented cache is organized as a 4-way set associative cache. It is composed of 128 lines for each way and each line is composed by 4 words of 32 bits.

The cache is split in two elements: the data cache, which contains the words of data required by the CPU provided by the memory, and the tag cache which allows to manage the lookup to the memory, allowing to determine if one request is miss or hit.

The tag memory is organized as the following table:

Lru_way	way3 valid	way3 dirty	way3 tag	way2 valid	way2 dirty	way2 tag	way1 valid	way1 dirty	way1 tag	way0 valid	way0 dirty	way0 tag

The first column represents the way least recently used and it allows to identify the victim line, that is the line that will be unloaded in case of miss request.

For each way there are three elements, the first is the valid bit that is useful to identify if the data stored in cache are valid, and can be used, or not. The second one is the dirty bit. When a data is dirty it means that when it is unloaded, it will be written in memory, otherwise the write in memory is unnecessary. The last one element is the tag works as identifier for the data in the cache memory.

All these elements are placed in one single line of 94 bits, 2 bits for LRU, 1 for both valid and dirty, and 21 for tag.

Regarding the data cache, there are two possible ways to implement it.

The first is similar to the tag cache, and it is showed in the following representation:

Way0_word0	Way1_word0	Way2_word0	Way3_word0
Way0_word1	Way1_word1	Way2_word1	Way3_word1
Way0_word2	Way1_word2	Way2_word2	Way3_word2
Way0_word3	Way1_word3	Way2_word3	Way3_word3

Instead, the second one is the following:

Way0_word0	Way0_word1	Way0_word2	Way0_word3
Way1_word0	Way1_word1	Way1_word2	Way1_word3
Way2_word0	Way2_word1	Way2_word2	Way2_word3
Way3_word0	Way3_word1	Way3_word2	Way3_word3

The differences between these two schemas are represented by the number of cycles spent to manage several situations.

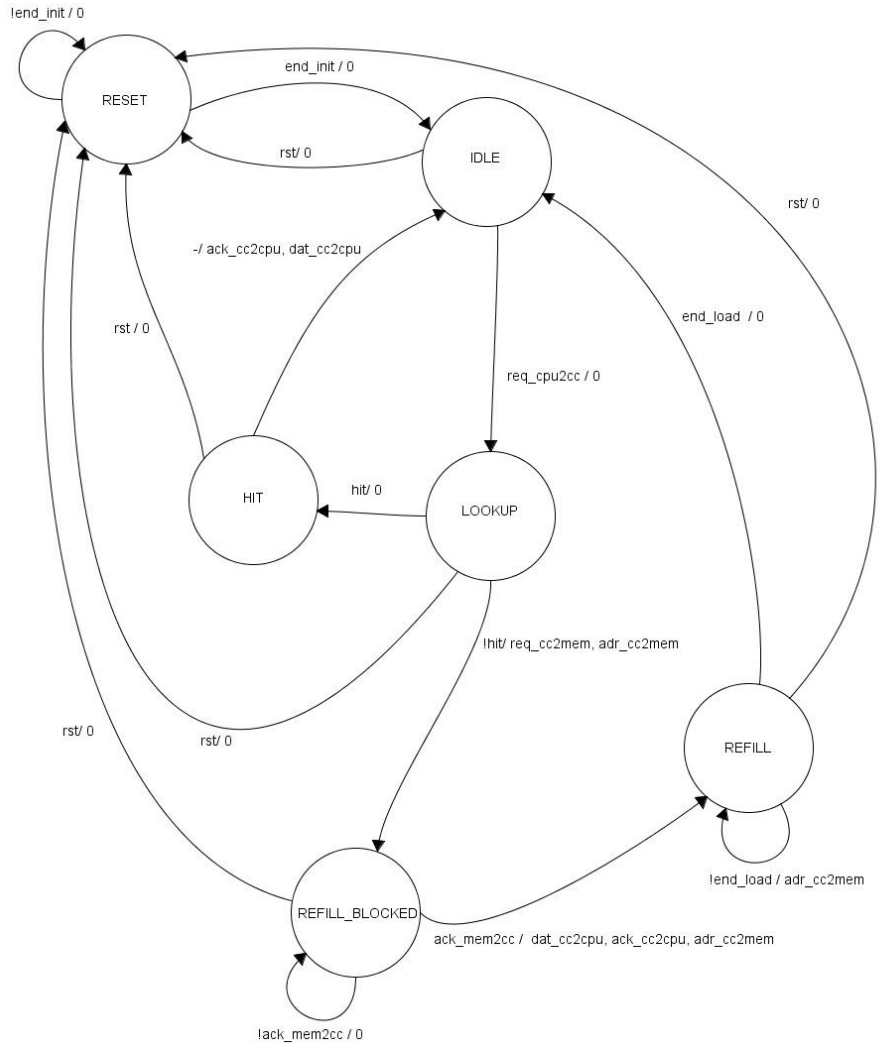
With the first type, when a request is forwarded from the CPU, without knowing which way consider but knowing the word offset is possible to retrieve directly in one cycle the data. With the second organization, this is not possible, because the first cycle is spent for the lookup in tag cache and then, when the way is known it is possible to access to the data cache and retrieve the line. It requires two cycles.

But on the other side, the first method is slower than the second one in case of line unload. The first requires four clock cycles, one for each word. Instead, the second type requires only one cycle to unload the whole line.

This advantage allows to gain three cycles for the CPU. And for this reason, the second layout is preferred.

2 CACHE CONTROLLER BEHAVIOR

The cache controller is implemented as a finite state machine, made up by six states, as showed in the following picture.



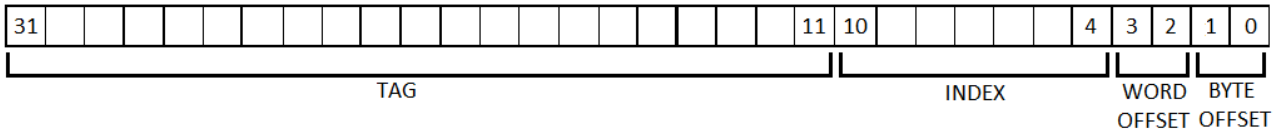
2.1 RESET STATE

The first state is the reset, that works as initialization for the cache. In this state all the lines are filled with 0, both the data cache and tag cache.

2.2 IDLE STATE

The controller is in idle when it is waiting for a request from the CPU.

If a request is detected the address arrived from the CPU is split in four elements, as showed in the following image:



- **Index**: is used to access to the right line in the tag and data cache.
- **Tag**: as explained before is the identifier used in the cache.
- **Word offset**: defines which word is required by the CPU.
- **Byte offset:** defines which byte is required (if a load byte request is processed).

Through these information the access to the tag cache is allowed.

Once that the request is detected, the controller at the next clock cycle will go to the **Lookup state**.

2.3 LOOKUP STATE

Knowing that the memories are implemented by the BRAMs, it is known that every access to these types of memory require one clock cycle.

This means that lookup began during the **Idle state,** the process terminates in the lookup state, and the right line is extracted. After that, is possible to compare the tag of each way with the requested tag and defines if the request is miss or hit.

If the request is a hit then next state will be **Hit state**, otherwise it will be **Refill blocked state**, and in this case a data request towards the central memory is forwarded.

2.4 HIT STATE

In this state, the hit requests from the CPU are completed.

If the request is a read hit, then an acknowledge signal and the data are sent to the CPU.

Otherwise, if the request is a write hit the data to write is sent to the data cache. Also, in this case, the acknowledge signal is risen.

2.5 REFILL BLOCKED STATE

In this state the controller first unloads the victim line in the MSHR and then it waits the first word from the central memory.

When the word is available, it is sent to the MSHR and to the data cache.

Now, is evaluated the CPU request and the acknowledge towards the CPU is risen.

The next state will be the **Refill state**.

2.6 REFILL STATE

In this state, the controller handles the load of the remaining words from the memory.

Since in refill blocked state, the acknowledge signal has been generated, the CPU is now free to continue with its operations. If another request to the cache is issued while the controller is in refill state, then the new request is stalled and dealt when the previous one is completed.

When all the words are loaded and the new line is correctly saved in the data cache, the controller goes in **Idle state**, and waits for another request.

3 TEST CASES

In this section we show the respective test cases in which we leverage the cache controller architecture, covering all the possible cases: Read hit/miss and Write hit/miss.

The test cases done are in sequence:

- Read miss way 0
- Read miss way 1
- Read miss way 2
- Read miss way 3
- Read hit way 0
- Write hit way 3
- Write hit way 1
- Read miss way 2
- Read hit, reads the value written in test 7
- Write hit on the tag of previous read operation
- Write hit on the tag of previous read operation
- Write hit on the tag of previous read operation (at this point the entire line is overwritten)
- Read hit, lbu operation word offset 3 byte offset 0
- Read hit, lb operation word offset 3 byte offset 0
- Read hit, lbu operation word offset 3 byte offset 1
- Read hit, lb operation word offset 3 byte offset 0
- Write miss way 0