ECE 566A Modern System-on-Chip Design, Spring 2017
Lab 3: Cache Design
Due: Mar 24, 5:00 pm

1. Overview

In Lab3, you need to read source code of the baseline cache design and then your need to design your own direct-mapped and two-way associative cache using Verilog HDL based on these open source codes. After that, you will go through the ASIC design flow you did in Lab1 on these cache controllers you designed. So, in this lab, you need to know how to use ASIC design flow tools including Synopsys VCS, Design Compiler, and Cadence Encounter by following the tutorials that posted on our course website. In addition, you will compare the performance of different caches in your report.

2. Background knowledge

2.1 Cache mapping techniques

Cache mapping is the method by which the contents of main memory are brought into the cache and referenced by the CPU. The mapping method used directly affects the performance of the entire computer system.

1) Direct mapping

Main memory locations can only be copied into one location in the cache. This is accomplished by dividing main memory into pages that correspond in size with the cache. (Fig. 2.1)

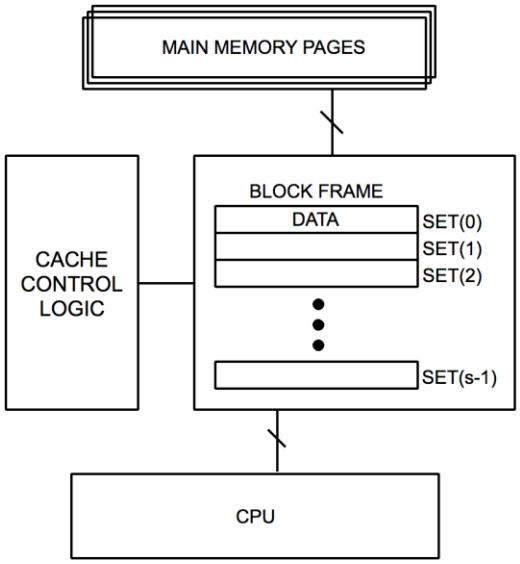

Figure 2.1 Example of direct mapping used in cache memory

2) Fully associative mapping

Fully associative cache mapping is the most complex, but it is most flexible with regards to where data can reside. A newly read block of main memory can be placed anywhere in a fully associative cache. If the cache is full, a replacement algorithm is used to determine which block in the cache gets replaced by the new data. (Fig. 2.2)

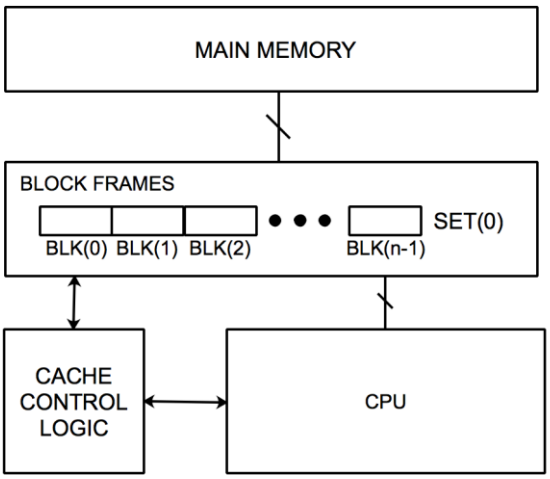

Figure 2.2 Example of fully associated mapping used in cache memory

3)Set associative mapping

Set associative cache mapping combines the best of direct and associative cache mapping techniques. As with a direct mapped cache, blocks of main memory data will still map into as specific set, but they can now be in any N-cache block frames within each set. (Fig. 2.3)

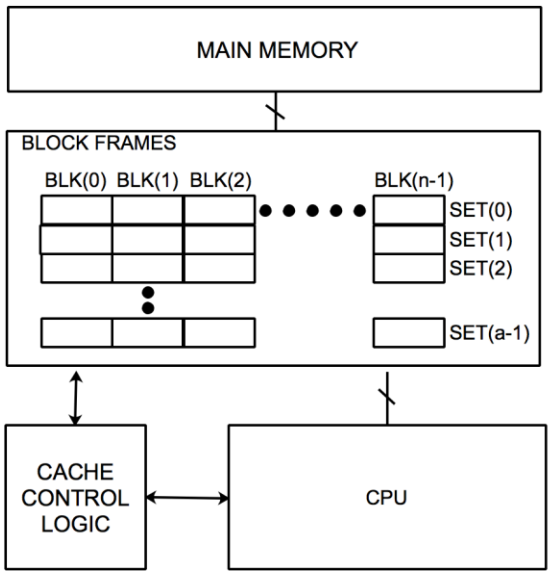

Figure 2.3 Example of set association mapping used in cache memory

2.2 Cache write

Since the cache contents area duplicate copy of information in main memory, writing (instructions to enter data) to the cache must eventually be made to the same data in main memory. This is done in two ways as follows:

1) Write-through cache

Writing is made to the corresponding data in both cache and main memory.

2) Write-back cache

Main memory is not updated until the cache page is returned to main memory. A write-back cache is more complex to implement, since it needs to track which of its locations have been written over, and mark them as dirty for later writing to the backing store. The data in these locations are written back to the backing store only when they are evicted from the cache.

No data is returned on write operations, thus there are two approaches for situations of write-misses:

- Write allocate (also called fetch on write): data at the missed-write location is loaded to cache, followed by a write-hit operation. In this approach, write misses are similar to read misses.
- No-write allocate (also called write-no-allocate or write around): data at the missed-write location is not loaded to cache, and is written directly to the backing store. In this approach, only the reads are being cached.

3. Cache description for the lab

We have provided you with a functional-level model of a cache, which essentially just passes all cache requests through to the memory interface, and passes all memory responses through to the cache response interface. While this might not seem useful, the functional-level model will enable us to develop many of our test cases with the test memory before attempting to use these tests with the baseline and alternative designs.

Based on the baseline design, you need to design both a direct-mapped, write-back cache and a two-way associative, write-back cache with a total capacity of 256 bytes -- 16 cache lines, and 16 bytes per cache line.

The datapath for the alternative direct-mapped design is shown in Fig. 3.1. The blue signals represent the control/status signals for communicating between the datapath and the control unit. The mkaddr block simply concatenates the tag and index plus some zeros like this: { tag, 4’b0000 }.

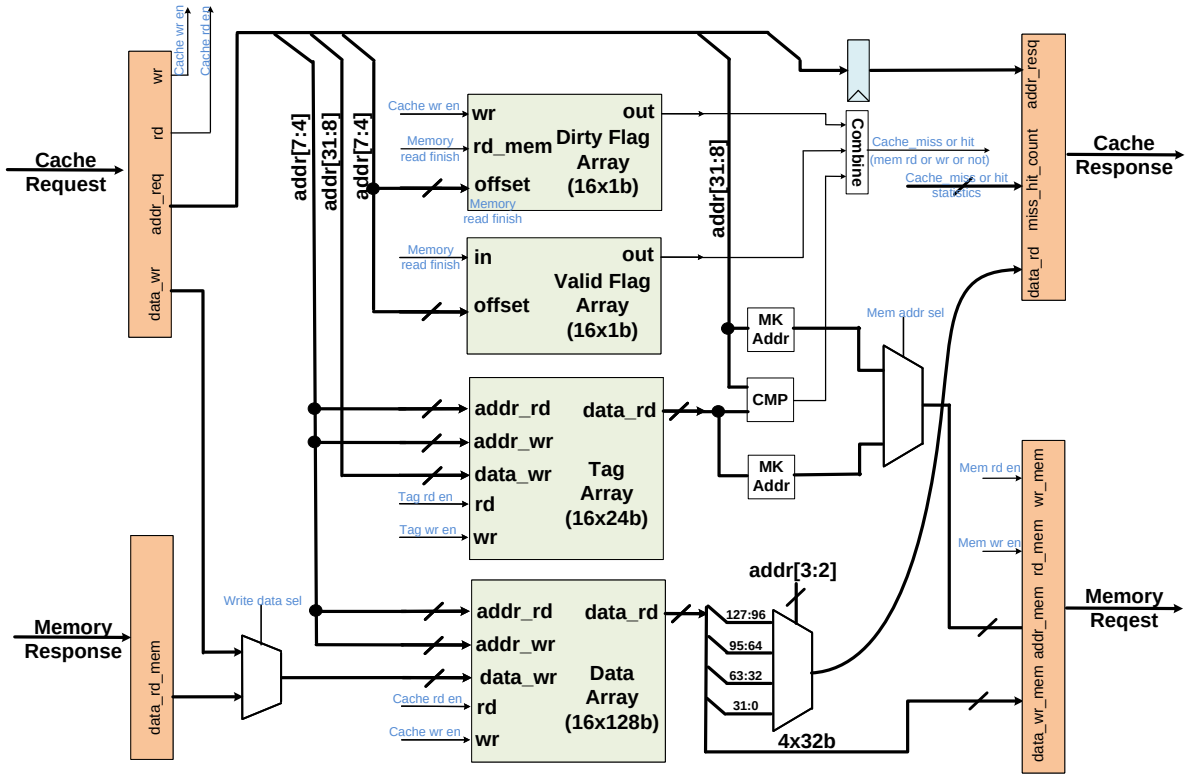

Figure 3.1 Direct-Mapped Cache datapath

The direct-mapped design is direct mapped with 16-byte cache lines and a total capacity of 256 bytes (i.e., 16 cache lines). So, we need four bits for the byte offset and four bits for the index leaving 24 bits for the tag.

The FSM for the direct-mapped design is shown in Fig. 3.2. The control unit should include valid and dirty bits to track the state of each tag entry. And Fig. 3.3 shows the address format when access.

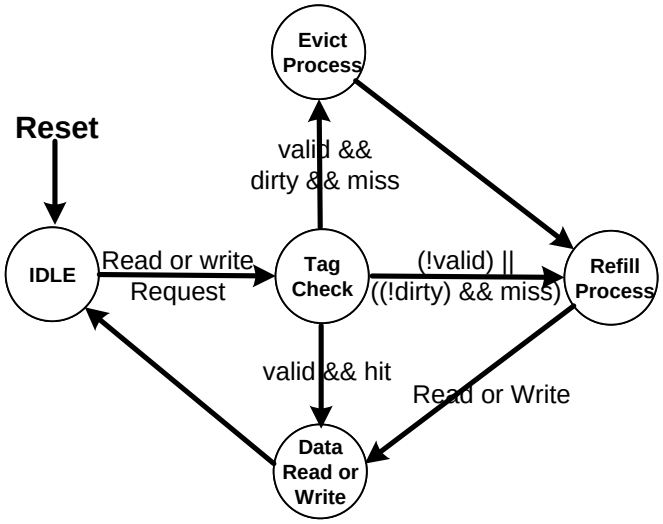

Figure 3.2 Direct-Mapped cache FSM control unit
(It also could be used in Two-way Associative cache)

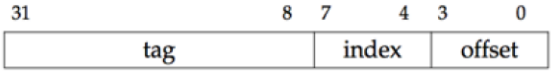

Figure 3.3 Address formats when access the cache

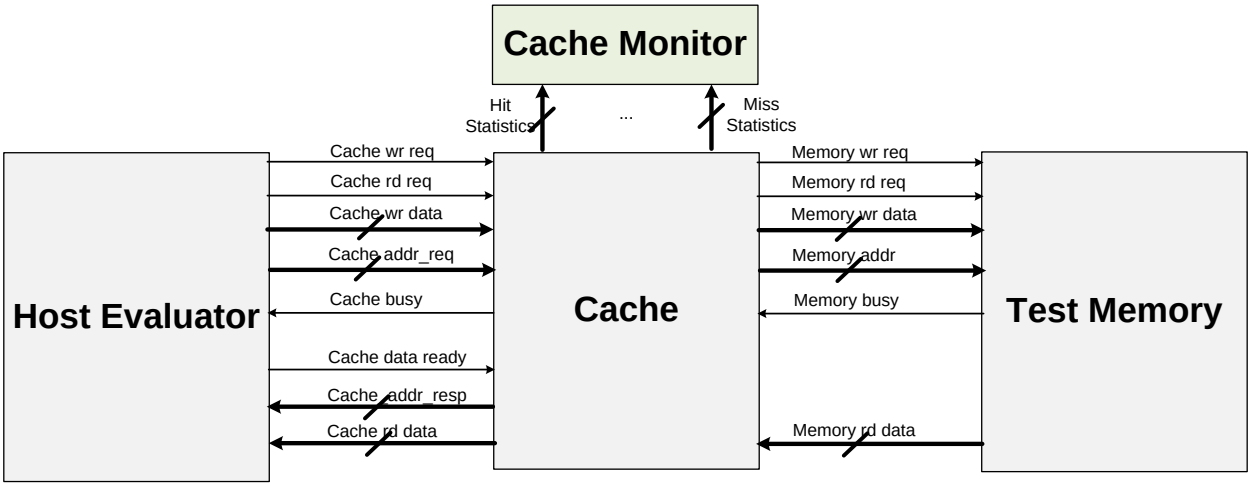

Figure 3.4 Example block-level diagram used in cache design

Fig. 3.4 shows a block-level diagram illustrating how the direct-mapped, baseline, and alternative designs are integrated with a host evaluator, cache, and test memory for testing and evaluation. We will load data into the test memory before resetting the cache. Once we start the execution, the host evaluator will send read or write requests into the cache, and eventually the cache will send memory responses to the host evaluator. If the cache needs to access main memory, then the cache will send memory requests to the test memory, and eventually the test memory will send memory responses back to the cache.

Note that while the memory request or response format is quite flexible, our cache designs will only support 4-byte cache requests and 16-byte memory requests. The data field can contain an arbitrary value in a write memory request, however the data field must contain all zeros in a write memory response. This simplifies creating reference responses when testing.

You can find the Verilog version code of baseline cache in the folder *Baseline* on our classroom Github of Lab3_src. Detailed explanation of the folder content is in section 4.

Based on the baseline design, you will need to design your own verilog code of direct-mapped and two-way associative cache for this lab. And you can find the template code in the folder *DirectMap* and *TwoWayAssociative* on our classroom Github of Lab3_src. Detailed explanation of the folder content is in section 4.

4. How to get started

You are expected to accept the lab assignment in the link by click the button “Accept this assignment”.

https://classroom.github.com/assignment-invitations/fd9053e0018022b5a51083520633cd4c

If you are the first time to use github, you should generate your own public key and add it to your github account, or you will have permission denied when you git clone the repository form our github classroom. When you add the ssh public key, you can follow the link below:

https://help.github.com/articles/connecting-to-github-with-ssh/

And then create a folder in your own linux server account and git clone your lab assignment repository. There are the command lines you may refer below:

% mkdir name_folder
% cd name_folder
% git clone git@github.com:wustl-ese566/lab3-your_user_name.git
% cd lab3-your_user_name/ (i.e., cd lab3-YunfeiGu/)

This repository lab3-your_user_name/ contains two folders:

- Baseline: a folder contains the source code of baseline Cache;
- DirectMap: a folder contains the source code of direct-mapped Cache;
- TwoWayAssociative: a folder contains the source code of two way associative Cache;

There are five files in each folder:

- CacheController.v: Cache controller implementation code;
- CacheController_tb.v: Test bench to test the correctness of the cache controller;
- CacheController_tb1.v: Test bench to test the miss rate of the cache;
- ram.bin: Memory initial data;
- Makefile: Automated compile support file.

For each CacheController.v, the inputs and outputs are:

- rst: Reset;
- clk: Clock input;
- wr, rd: Cache operation request signals;
- data_rd: Data returned from cache (Cache to host);
- data_wr: Data written to cache (Host to cache);
- addr_req: Cache request address (Host to cache);
- addr_resp: The data address of cache response (Cache to host, cache controller keeps the address of cache request in a buffer when cache miss happens);
- rdy: Cache ready;
- busy: Cache busy;
- wr_mem, rd_mem: Memory operation request signals;
- busy_mem: Memory busy;
- data_rd_mem: Data returned from memory (Memory to cache);
- data_wr_mem: Data written to memory (Cache to memory);
- addr_mem: Memory access address (Cache to memory);

- cache_miss_count: Cache miss statistics;
- cache_hit_count: Cache hit statistics;

The direction of each signal is defined in the provided Verilog code.

(Note: In the lab we mainly focus on the cache controller, so we assume *busy_mem* is always de-asserted).

Please be sure to source the class setup script using the following command before compiling your source code: module add ese461.

5. Tasks

5.1 Direct-mapped Cache design

You should implement a direct-mapped cache with the same interface used in the baseline design (Code template is in the folder *DirectMap* on github).

The size of cache is 16 lines with 16 bytes for each line. So, the total cache size is 16x16 bytes. Please implement the *cache_miss_count* and *cache_hit_count and* present them into your reports. They will be used to evaluate the performance of your cache. Also, please show the content of the cache in your report to show that your cache work properly.

5.2 Two-way associative Cache design

You should implement a two way associative cache with the same interface used in the baseline design (Code template is in the folder *TwoWayAssociative* on github).

The size of cache is 16 lines with 16 bytes for each line. So, the total cache size is 16x16 bytes. Please implement the *cache_miss_count* and *cache_hit_count and* present them into your reports. They will be used to evaluate the performance of your cache. Also, please show the content of the cache in your report to show that your cache work properly.

5.3 Compare the performance of different Caches

In this section,you will compare the performance of direct-mapped and two-way associative Cache. And you need to go through all the steps you did in lab1:

- Use Synopsys VCS to compile the Verilog source code of different cache controllers;
- Use Design Compiler to do the synthesis of different cache controllers;
- Use Cadence Encounter to do the place and route of different cache controllers.

Then in your report, compare the power, area, timing and other facts that you think are important of these different Cache and briefly explain the result. It is better to visualize your experimental results(tables and/or plots).

6. Submission

Please submit your lab assignment on Github. You are expected to submit your report, *TwoWayAssociative/CacheController.v* file, and lab3_dc.tcl file for each *Cache*. If you modify the test benches or create new ones, submit them too.

To submit your job, execute the following command:
(Note: the first two commands just need to be done once for the entire semester.)

```
% git config --global user.name "your_user_name"
% git config --global user.email "your_email_for_github"
% cd directory_of_your_lab_assignment/lab3-your_user_name/
% git add DirecMap/CacheController.v
% git add TwoWayAssociative/CacheController.v
% git add DirectMap/lab3_dc.tcl
% git add TwoWayAssociative/lab3_dc.tcl
% git add Lab3-report.pdf
% git add DirectMap/CacheController_tb.v.v          (optional)
% git add DirectMap/CacheController_tb1.v          (optional)
% git add TwoWayAssociative/CacheController_tb.v   (optional)
% git add TwoWayAssociative/CacheController_tb1.v  (optional)
% git commit -m "your commits"
% git push -u origin master
```

You also should submit anything else you think may help us understand your code and result.

Please do not submit files like compiling result(*simv*) or simulation data(*.vpd*).

7. Acknowledgement

[1] http://www.csl.cornell.edu/courses/ece4750/handouts/ece4750-lab3-mem.pdf