2006-12: A CLASS PROJECT FOR LOW-POWER CACHE MEMORY ARCHITECTURE

Yul Chu, Mississippi State University

A Class Project for Low-Power Cache Memory Architecture

Abstract

This paper presents a class project for a graduate-level computer architecture course. The goal of the project is to let students (two or three students per team) understand the concept of computer hardware and how to design a simple low-power cache memory for future processors. The project consists of three different tasks: 1) Design - Designing a low-power cache memory (instruction or data) at the abstract level after literature research; 2) Code - Writing a simulation program on top of a simulator (e.g., Simplescalar); and 3) Test - Running a test program to evaluate the low-power cache memory by using performance metrics, such as power consumption, cache miss rate, execution time, etc. For the first task, students are required to design their own low-power cache memory. For the second task, they need to write (or modify) a simulation program to implement their design. Finally, they should run benchmark programs through the program to evaluate their cache memory.

1. Introduction

A simulation program has been an important tool to verify the functions for logically designed computer hardware before chip fabrication [1]. A graduate-level computer architecture course, in general, deals with designing a low-power cache memory, branch predictor, superscalar, VLIW, or multi-processors at the abstract level instead of the circuit level [2][3]. After the logical design, students are required to simulate the design with the benchmark programs to inspect whether or not it works properly. This paper presents in detail how to design a low-power cache memory for a graduate-level computer architecture course.

Simulation programs are useful for many computer-engineering courses since they can help students to develop and evaluate their ideas with less hardware costs [4][5]. However, some detailed simulators used to discourage students with many options for selection and lengthy lines of code [6]; students can just repetitively implement the fixed, limited operations of the simulators; therefore, it makes difficult for the students to design a new function logic.

To implement a low-power cache memory, students are required to design a mapping function, replacement policy, write policy, and low-power cache memory architecture at the abstract level [1][3][7]. After that, they can write (or modify) a simulation code for their cache memory and test it to check whether or not working properly.

This paper is set out explained as follows: Section 2 introduces the procedures for designing a low-power cache memory; Section 3 discusses how to grade the project and provides students' evaluation; and Section 4 gives the conclusions.

2. Project Procedures

Three major procedures for the class project are design, code, and test. This section shows each procedure in detail. The first step is to design an efficient mapping function for a cache memory to improve system performance.

Figure 1 shows three types of cache misses: compulsory misses, capacity misses, and conflict misses. Compulsory misses come from the cold start, the first time accesses; capacity misses depends on the cache sizes; conflict misses are caused by competing one location in a cache

memory. In general, conflict misses are very critical for a small cache size, especially level-one on-chip cache memory, and directly affect the power consumption, system performance, and costs. Therefore, the goal of the project is mainly on designing an efficient cache memory to reduce conflict misses and power consumption.

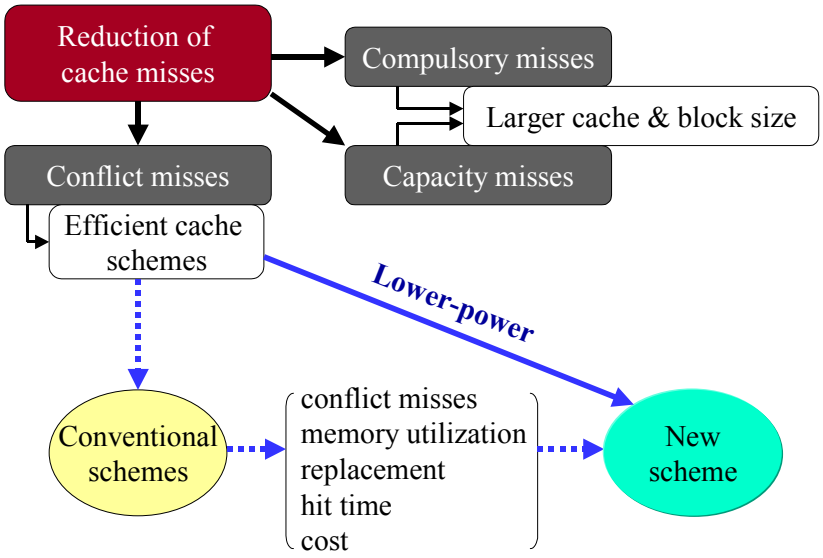

Figure 1. Three types of cache misses and a low-power cache scheme

2.1 Design Procedure

There are five main factors to design a cache memory [3]. Those are the cache size, block size, mapping function, replacement policy, and writing policy. In this paper, we focus on small on-chip cache sizes (e.g., 32KB or 64 KB) with the popular block sizes (e.g., 32bytes or 64 bytes) and include one more factor, power consumption, to design a low-power cache memory for embedded systems. Therefore, there are four factors to design a cache memory, except cache and block sizes: 1) Design a mapping function; 2) Design a replacement policy; 3) Design a writing policy; and 4) Design a low-power cache memory. Since there are many cache specifications to determine, we recommend students to work as a team, 2 to 3 students per team, to discuss many possible topics. Through the discussion, students can build strong and clear concepts for designing a low-power cache memory.

2.1.1 Design a mapping function

For the first step, each team is required to design a mapping function to access the cache memory in an efficient way.

For example, Figure 2 shows two different mapping functions: 2-way set-associative (conventional) and 2-way skewed-associative (more efficient one). Each scheme has two banks and each bank has its own mapping function. For Figure 2-a, the mapping functions F_0 and F_1 are the same. Therefore, if three instructions (A0, A1, and A2) access to the same location in the

bank 0 and bank 1, there should be a conflict since they are located in only two banks. Meanwhile, if the mapping functions F0 and F1 are different like Figure 2-b, the conflict can be resolved since three instructions can be placed into three different locations in the Bank 1. Therefore, the mapping function is an important factor to reduce cache misses. Each team can design any kinds of mapping function to reduce conflict misses by dispersing instructions in a bank.

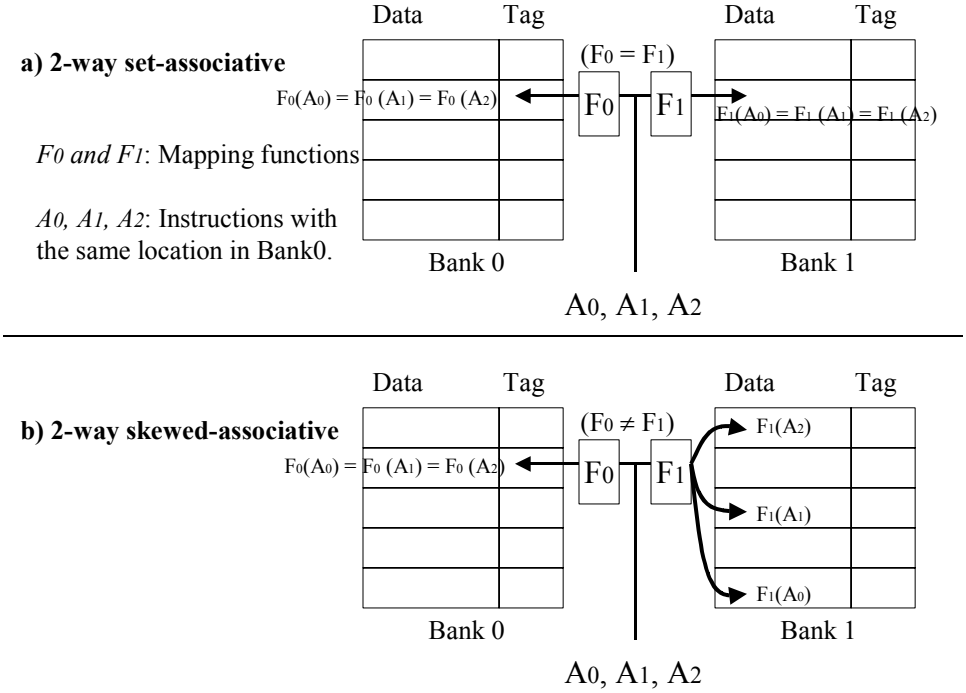

Figure 2. Mapping functions for 2-way set-associative and 2-way skewed-associative

2.1.2 Design a replacement policy

After designing the mapping function, students need to design a replacement policy to place/replace data effectively. For example, if there is a cache miss, it is necessary to bring a data (or instruction) from a lower-level memory (memory) and place (or replace) that data into the cache memory before sending it to the processor. At that time, if we replace the data, which will be used in a near future, it can deteriorate the performance since it should cause another cache misses later. Therefore, it is important to design an efficient replacement policy to increase chances to be referenced by the processor again.

Figure 3, as an example, shows the PLRU (Pseudo Least Recently Used) replacement policy for 2-way skewed-associative: "Whenever there is a cache miss, the flag in the bank 0 would be checked: if the flag bit in the bank 0 is '0', then replace data in the bank 0 and set the flag to 1; otherwise, replace data in the bank 1 and set the flag to 0 [13]."

After designing a replacement policy, each team needs to implement the replacement policy with the mapping function by using a small data manually to make sure for the possible worst cases.

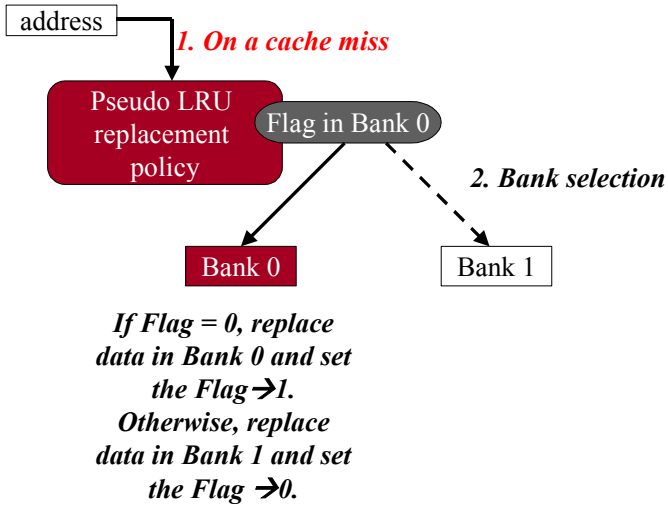

Figure 3. Replacement policy for 2-way skewed-associative

2.1.3 Design a writing policy

There are two types of the writing policy: write-hit policy and write-miss policy. For the write-hit policy, the write-back policy is popular since it updates data only in the cache memory until it is replaced with the data from the lower-level memory (memory). If it is replaced because of a cache miss, the data should be updated to the memory before it is replaced in the cache memory. It works well to reduce memory access time, which is much slower than the cache memory, compared to the write-through policy. The write-through policy updates the cache memory and memory at the same time for any cache write-hits. It would take more time compared to the write-back policy since it should access the memory for every write-hit case.

For the write-miss policy, there are also two general policies, such as the write-allocate (to update the cache memory first and memory later) and the write-no-allocate (to update memory only) for a write-miss. Each team can design write policies based on the conventional write policies to accommodate future references effectively.

2.1.4 Design a low-power cache memory

Figure 4 shows hardware-complexity comparison between two cache memories since cost is also an important factor for cache memory design. In Figure 4, the cost of 2-way skewed-associative would be more expensive than 2-way set-associative because 2-way skewed-associative uses xor mapping functions, which is more complex than 2-way set-associative. However, since the xor mapping functions can reduce cache misses effectively, there can be a tradeoff between two cache memories regarding the performance and cost.

To reduce power consumption for a cache memory, it has been necessary to reduce the frequency of memory accesses by developing techniques such as line buffering. Some research has indicated that a small buffer line between CPU and an L1 (Level-one) cache memory helps in reducing the accesses to the L1 cache and thus reduces the energy dissipation in the L1 cache.

Contents		2-way set-associative	2-way skewed-associative
Logic and Indexing	Replacement	LRU, etc.	PLRU, etc.
	Indexing	Lower part of address	XOR mapping
Hardware	Banks	2	2
	Flag	Y (Bank 0)	Y (Bank 0)
	Bank design	Classical design	Classical design + XOR gates (mapping)
	Counter	N	N
Access time		Same	Same (slightly increased by XOR gates)
Hardware complexity		Same	Almost Same (Only several XOR gates are added to the 2-way set-associative)
Cache miss ratio		High	Medium

'**Access time**' and '**Hardware complexity**' from 'A. Seznec, A case for two-way skewed associative Caches, Proc. Of the 20th ISCA, May 1993, pp169-178'.

Figure 4. Comparison hardware complexity for 2-way cache memories

For example, the CoC (Caching on Cache) is a small-sized sub-cache of the L1 cache [10]. The CoC is accessed first for every memory reference: If the reference is a hit, the lines in the L1 cache are disabled (no access to L1 cache). Otherwise, the L1 cache is accessed normally. The access operation involves maintaining a tag and data of the cache lines and also adds latency to the normal cache access. For another example, the Filter cache is also a small cache between the CPU and the conventional L1 cache (the conventional L1 cache used to be treated as a L2 cache) [11]. The Filter cache contains data with high hit probability based on locality. Therefore, a miss on the filter cache directs the references to the L2 cache, which used to be a conventional L1 cache. The power savings are earned by maintaining a small-size L1 cache (Filter cache) instead of a regular L1 cache. These techniques require additional data and tag arrays that consume power with every reference. Each team can design a low-power cache memory like the above examples.

2.2 Code Procedure

Since the advanced computer architecture class is a graduate-level course, programming languages like C/C++ or VHDL/Verilog would be prerequisites for the class in general [4].

In the previous section, each team could design the mapping function, replacement policy, writing policy, and low-power cache memory for their own purpose. The next step is to write a code for their cache memory. In other words, after students write a code according to the procedures, they should port the code into a simulator like SimpleScalar. Figure 5 shows five SimpleScalar standard simulator models as an example.

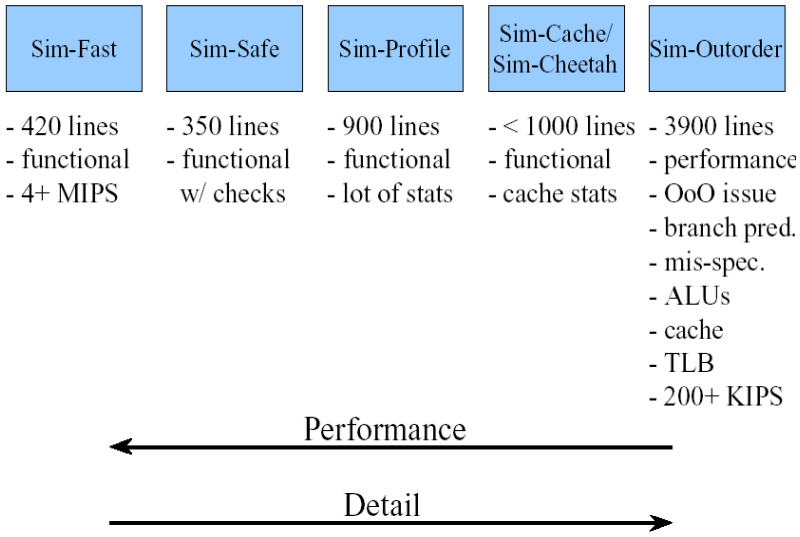

Figure 5. SimpleScalar standard models [8]

```
/************************DATA CACHE ACCESS FUNCTION ****************************/

/* access a cache, perform a CMD operation on cache CP at address ADDR,
   places NBYTES of data at *P, returns latency of operation if initiated
   at NOW, places pointer to block user data in *UDATA, *P is untouched if
   cache blocks are not allocated (!CP->BALLOC), UDATA should be NULL if no
   user data is attached to blocks */
unsigned int                            /* latency of access in cycles */
cache_skew_access(struct cache *cp,             /* cache to access */
              enum mem_cmd cmd,         /* access type, Read or Write */
              SS_ADDR_TYPE addr,        /* address of access */
              void *vp,                 /* ptr to buffer for input/output */
              int nbytes,               /* number of bytes to access */
              SS_TIME_TYPE now,         /* time of access */
              char **udata,             /* for return of user data ptr */
              SS_ADDR_TYPE *repl_addr)  /* for address of replaced block */
{
  char *p = vp;
  SS_ADDR_TYPE tag = CACHE_TAG(cp, addr);
  SS_ADDR_TYPE set, tempset;
  SS_ADDR_TYPE bofs = CACHE_BLK(cp, addr);
  struct cache_blk *blk, *repl, *tempblk, *blk1, *blk2;
  int lat = 0,flag=1;
#ifndef __alpha__
  extern long random(void);
#endif

  /*------------------------- These are my variable declaration--------------*/
  SS_ADDR_TYPE offset ;
  SS_ADDR_TYPE tagset;
  SS_ADDR_TYPE xoredaddr =0x00000000;       /* address of xor mapping access*/
  SS_ADDR_TYPE tempaddr =addr,x=0,y=0;  /*y=286331153;   //y=372964681;*/

  SS_ADDR_TYPE iset = CACHE_SET(cp, addr);
```

Figure 6. Example: a data cache access function in the SimpleScalar [9]

SimpleScalar is an open-source simulator to implement the computer architecture to evaluate devices and performance [8]. In Figure 5, SimpleScalar has several standard models for the architecture and each team can use the 'Sim-Cache/Sim-Cheetah' model for the low-power cache memory project. To use those standard models, each team should: 1) Install SimpleScalar into each team's working directory; and 2) Compile and link to generate the binary files according to [9]. After each team installs the SimpleScalar into the working directory, they can add the code for the mapping function, replacement policy, write policy, and low-power cache memory into the SimpleScalar before compiling and linking. After that, they can compile and link it to evaluate the cache memory.

Figure 6 shows an example of the data cache access function code to be added into the SimpleScalar model. Like Figure 6, each team can write codes for a cache memory and port them into the SimpleScalar to implement.

2.3 Test Procedure

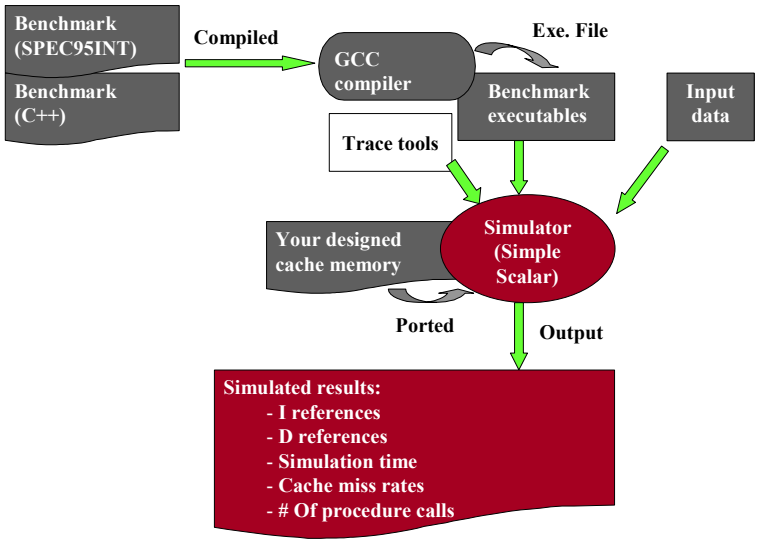

Figure 7. Test procedure with benchmark programs

After students complete the design and code procedures, they need to test their architecture with the benchmark programs.

Figure 7 shows the procedure how to test the cache memory with benchmark programs. The benchmark programs should be compiled and linked to produce the binary, which is the real input for the simulator. Since the simulator is a virtual architecture, it has its own instruction sets; and it is necessary to compile the benchmark programs with the compiler provided by the SimpleScalar to run Benchmark executables (e.g., compiled binaries) [9].

After you create the input binaries, you can implement the simulator with the input data by using the following command line in Figure 8.

Figure 8 shows the command line with many options, such as a level-one data cache (dl1), 32 lines in a cache (32), associativity (1 as a direct-mapped cache), replacement policy (1), the binary input program (/bin/test.ss), etc. Each team can add more options after modifying some procedures in the SimpleScalar models.

After you complete the simulation with benchmark programs, like the Figure 7, you can get the simulation results, such as I references (# of instructions), cache miss rates, etc. With the results, you can evaluate your cache memory by using some metrics, such as cache miss rate, execution time, IPC (Instructions per cycle), power, etc. Those metrics are [1]:

- CPU Execution time = IC × Effective CPI × t – (1)
- Effective CPI = Ideal CPI + Average memory stalls per instruction – (2)
- AMAT = Hit time + Miss rate × Miss Penalty – (3)
  ** IC (Instruction Count), CPI (Cycle Per Instruction), AMAT (Average Memory Access Time).

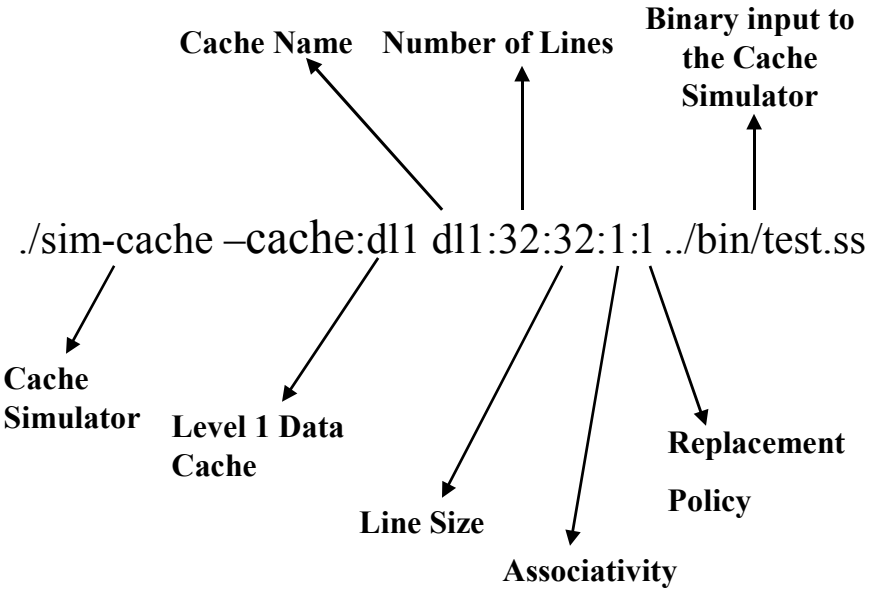

Figure 8. SimpleScalar command line to run a test program

Cacti can be used for computing power consumption for cache memory architecture [12]. After completing the project procedures, students are required to prepare and submit a final project report. The final report would include the followings:

- An explanation of the mapping function, replacement policy, and write policy;
- An explanation of the low-power cache memory architecture;
- A discussion of how to test the architecture;
- A discussion of errors in the architecture;
- A discussion of how to optimize the errors; and
- Simulation results such as cache misses, IPC, power consumption, etc.

3. Grading projects and students' evaluation

The grading for the project mainly depends on the work from the three procedures. For the design procedure, we need to check the efficiency of the designed mapping function, replacement policy, write policy, and low-power cache memory. For the code procedure, the major point is to check whether each part of codes works properly or not. For the test procedure, the whole test procedure would be checked with the results. In addition to the grading, it is necessary to check the discussions among team members since the goal of the project is to share ideas and get clear concepts through their discussions.

As a case study, Figure 9 shows the project grading for the project during fall semesters in 2004 and 2005 at the Mississippi State University. There were 4 teams, 2 students per team, and their final reports were graded based on efficiency (30%), correctness (20%), testing (10%), results (35%), and discussions (10%).

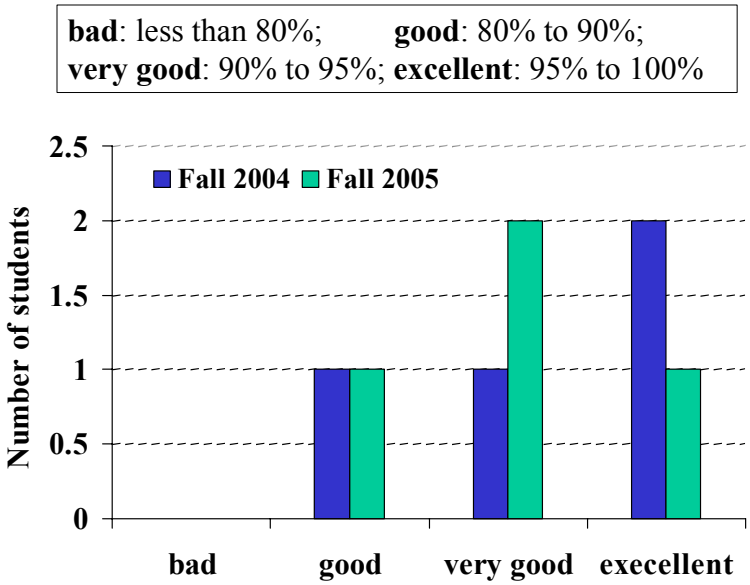

Figure 9. Example: grading projects (Fall 2004 / Fall 2005)

Figure 9 shows that three teams (75%) got grade A (very good and excellent), and one team (25%) had grade B (between 80% and 90%) in 2004 and 2005. Therefore, we could say the class project was successful to let students understand the concepts and design process for the low-power cache memory.

In addition, we submitted two final reports (from 2004), which have excellent grades, to the IEEE Southeast Conference 2005 (referred conferences), and they were accepted as regular technical papers. And one of them also accepted as a short paper for the IEEE IPCCC '05 (International Performance Computing and Communications Conference). Currently, we are preparing two papers to submit to a journal and a conference with the Fall 2005 reports.

From the students' evaluation, we found that the class project worked well for letting students understand how to design a cache memory. However, since there was a one-week break (e.g., Thanksgiving break) during the project term (from Oct. 22 to Nov. 28), most students were short of time to finish the project on time. Therefore, it would be better to start the project one week earlier than Oct. 22 for every fall semester (from the students' comments).

4. Conclusions

There have been so many software tools developed to teach computer architecture classes. Traditionally, those tools have many options to select for any proper operations or consist of lengthy lines of code to figure out. Therefore, students are required to figure out the options first and then learn the proper operations. In addition, since the tools used to have limited functions to operate, it is difficult to design a new function logic with the tools. Therefore, those tools let students understand only the limited operations instead of creative design since they lack experience of the designing process.

This paper introduces a class project for designing a cache memory with the three procedures: 1) Designing a cache memory; 2) Coding the designed cache memory and porting them into the

SimpleScalar simulator; and 3) Testing the architecture through the simulator with the benchmark programs.

According to the grading and students' evaluation from Mississippi State University, we found that these procedures worked successfully for the graduate-level advanced computer architecture class since all students who participated in the class project had As and Bs for their grades and two teams' final reports (out of four teams) were accepted at a conference in 2005. Therefore, we believe that students could learn fundamental concepts and the design process clearly for the advanced computer architecture class and gain confidence in the area of low-power cache memory design.

Bibliography

[1] David A. Patterson & John L. Hennessy, Computer organization and design: the hardware/software interface, *third edition, Morgan-Kaufmann,* San Francisco, California, 2005.

[2] Vincent P. Heuring and Harry F. Jordan, Computer Systems Design and Architecture, *second edition*, Prentice Hall, Upper saddle river, New Jersey, 2004.

[3] John L. Hennessy & David A. Patterson, Computer Architecture: A Quantitative Approach, *third edition*, *Morgan-Kaufmann,* San Francisco, California, 2003.

[4] Lillian Cassel et al, "Distributed Expertise for Teaching Computer Organization & Architecture", *Working Group Reports in the 5th Annual Conference on Innovation and Technology in Computer Science Education*, Helsinki, Finland, July 2000.

[5] D. Ellard, D. Holland, N. Murphy, and M. Seltzer, "On the Design of a New CPU Architecture for Pedagogical Purposes", *in Proc. WCAE 02 – workshop on Computer Architecture Education, on 29th International Symposium on Computer Architecture*, Anchorage, AK (USA), 2002, pp.28-34.

[6] Christopher T. Weaver, Eric Larson, and Todd Austin, "Effective Support of Simulation in Computer Architecture Instruction", *Workshop on Computer Architecture Education (WCAE02) held in conjunction with the 29th International Symposium on Computer Architecture*, Anchorage, AK, May 2002.

[7] Daniel C. Hyde, "Teaching Design In a Computer Architecture Course", *IEEE Micro, Volume 20, Number 3*, May/June 2000, pp23-28.

[8] Todd Austin, SimpleScalar Hacker's Guide Version 2, *SimpleScalar LLC*, http://www.simplescalar.com/docs/hack_guide_v2.pdf.

[9] D. Burger and T. M. Austin, The Simplescalar tool set, version 2.0, Technical Report CS-1342, University of Wisconsin-Madison, 1997.

[10] Hung-Cheng Wu, et al., Energy Efficient Caching on Cache Architectures for Embedded Systems, Journal of Information Science and Engineering, Vol. 19 No. 5, pages 809-825, 2003.

[11] Kin Johnson, et al., The Filter Cache: An Energy Efficient Memory Structure, 30th International Symposium on Microarchitecture (MICRO), Research Triangle Park, North Carolina, USA, Dec. 1997.

[12] Premkishore Shivakumar and Norman P. Jouppi, CACTI 3.0: An Integrated Cache Timing, Power, and Area Model, Western Research Laboratory, WRL-2001-2, Dec. 2001.

[13] A. Seznec, A case for two-way skewed-associative cache, the 20th International Symposium on Computer Architecture (IEEE-ACM), San Diego, May 1993.

Biographical Information

Yul Chu received the Ph.D. degree in Electrical and Computer Engineering from the University of British Columbia, BC, Canada in 2001, M.S.E.E. degree in Electrical Engineering from Washington State University in 1995, and Bachelor in applied electronics from KwangWoon University, Seoul, Korea in 1984. Since 2001, he has been with the Department of Electrical and Computer Engineering at the Mississippi State University and currently is an assistant professor. His current research interests are high performance computer architecture, low-power embedded systems, computer networking, parallel processing, cluster and high-available architectures, Telematics, digital design, etc.