Objective

1. To help you develop the ability to apply knowledge in computer engineering learned in the course.

2. To help you develop the ability to use the techniques, skills and modern engineering tools needed for engineering practice.

3. To help you obtain the knowledge of related topics in computer science discipline.

The objective of this final project is to give you an opportunity to learn how to design and implement a real computer architecture. Through this project, we hope to consolidate and refine your understanding of the RISC processor core that you have learned in the lectures of this class.

In the project, you will use a contemporary, widely used simulation tool called SimpleScalar. I think you have accumulated enough experience on this simulator through the previous homeworks and labworks.

You will be given a very simple version of the SimpleScalar simulator, which is called sim-fast, a functional simulator that simulate an unpipelined version of the architecture core. You need to hack the source code of the simulator and become familiar with the implementation details of the simulator. Then, in the first phase of the project, you will be asked to add the implementation of an instruction in the ISA. In the second phase, your mission is to extend the sim-fast module of the SimpleScalar to a 5-stage pipelined execution core with automatic hazard detection and forwarding. In the last phase, you should implement the cache simulation module in sim-fast simulator.

As before, this is a team project. It will "simulate" a real engineering project in open source community. Team members should cooperate to download documents and source code from the website, read those documents and hack the source code of the software, make clear the goal of the project, write down you design document clearly, and implement your design efficiently. Team should discuss to divide the project to small items to make sure the workload is evenly distributed on every member. Team should learn how to communicate effec-
tively, learn how to negotiate with each other, learn how to make decision, and learn how to assume responsibility.

**Project Description: Phase III**

After you finished the 2nd phase of the project, i.e. developing a 5-stage pipeline in sim-fast, I think you should understand clearly how instructions are executed in a processor. Actually, what you developed in the last phase is a multi-cycle datapath of the CPU. In your implementation, so far, all instructions are executed in 5 stages and each stage takes one clock cycle. Therefore, every instruction takes the same number of cycles, i.e. 5 cycles, to finish its execution, no matter it is a "load", a "branch", or a "floating point operation". This is a simplified model for you to understand and implement the 5-stage pipeline easily. In real world, things are more complicated than our course project. In fact, for a commercial processor, e.g. Alpha, Sparc, and IA-64, executing different instructions takes different number of clock cycles. An "add" operation may take 5 cycles, while a "floating point division" may take 20 or more cycles. For a "load" operation, sometimes it can finish in 3 or 5 cycles, and sometimes it will cost you 15 or 30 cycles. The difference is introduced by cache, a very important component in modern architecture. As you know, compared with the speed of CPU, memory access is very slow. Therefore, there is a deep and wide gap between memory and CPU. cache is such a kind of component invented to eliminate this gap. I will not elaborate the advantages and principles of cache here. Please reference the 7th chapter of Patterson & Hennesy's textbook for details. In the 3rd phase of the project, you are required to add a cache simulation module in the SimpleScalar simulator. To be more specific, your task is to modify the sim-fast simulator to let it simulate the behavior of cache.

1. **How to simulate cache?**

As defined by Patterson & Hennesy, cache is just the memory block that occupies the higher level in the memory hierarchy in computer architecture. Physically, there is no difference between cache and main memory, except that cache is faster and smaller than main memory. Functionally, cache is a subset of main memory. It stores the data that are referenced by the program frequently.

According to what has been discussed above, you can reserve a chunk of memory in your program, i.e. sim-fast, to simulate cache. Just as the author of SimpleScalar reserving host memory to simulate target machine's memory. What you need to do is to lookup data in the cache first. If the datum you want is found in cache, return it. Otherwise, continue to look for it in the main memory.

In file "machine.def", line 412 to 421 defines the implementation of "lw" instruction of MIPS. Pay attention to the macro "READ\_WORD" in line
This is the exact function/macro that memory access begins. You can wrap this macro and insert your cache lookup code before it. There are many implementation choices. Choose the one you like most.

2. The difference between cache lookup and memory reference

As I suggested in the last section, both memory and cache of the target machine are simulated by the host memory. Therefore, you may wonder, what is the difference between cache lookup and memory reference? My answer is, you should simulate the difference. As you know, cache is smaller but faster than main memory. So, there are cache miss and cache hit. When cache hit occurs, "load" instruction can quickly return the data in 1 or 2 cycles; while in case of cache miss, you should help the "load" instruction to get its data from main memory, which will take 10 or 20 cycles to finish. This performance penalty can be expressed in the total cycle number of program been executed. So, you must keep a cycle counter in the simulator. If cache hit occurs, add 1 or 2 cycles (its up to you to define how many. For simplicity, you can just specify 1 cycle) to the cycle counter. If cache miss, add 10 or 20 cycles. Then, print out the total cycle number of the running program. You can repeat the experiment on the simulator that has no cache, or cache is disabled (therefore, every memory access cause cache miss). You will get another cycle number. The difference between the cycles numbers manifest the difference of cache and memory.

3. The organization of cache

In this section, I will give some suggestions about the organization of cache.

a. Multi-level vs. Single-level

In modern commercial processor, there are several levels of cache in the system. L1, L2, L3 cache, or on-chip, off-chip cache. You are not required to implement a real CPU. So, a single level cache is enough.

b. Associativity

There three different ways to organize the cache lines: direct mapped, set associative, and fully associative. Please reference page 496 of your textbook for detailed description of these three methods. You are required to implement a fully associative cache in this project.

c. I-Cache vs. D-Cache

In modern computer architecture, the cache is divided into instruction cache and data cache. Instruction cache, or I-cache, is used to provide instructions for execution in instruction fetch stage. Data cache, or D-cache, is used to provide data in memory access stage. In
this project, you do not need to differentiate the type of the memory access. That means, your implementation is a unified cache.

d. Virtual Tag vs. Physical Tag
   Since the original SimpleScalar simulator didn’t simulate the memory management unit, you don’t need to worry about this problem. Just use the upper bits of the memory address as the tag of the cache line.

e. Cache Line Size and Cache Capacity
   The cache line size should be 32 bytes. And the cache should have 32 cache lines in total. Therefore, the capacity of the cache is 1k.

f. Valid Bit
   You should have a valid bit in the cache line to indicate whether the data in the cache line is currently valid or not.

g. Replacement Algorithm
   You should implement the LRU replacement algorithm (see page 504, P&H). "LRU" is the abbreviation of "Least Recently Used". It means that you should kick off the cache line that is "least recently used" when you can’t find an empty slot for the coming new cache line. Therefore, you need to maintain a "life" counter for each cache line. The "life" counter indicates how long did this cache line live in cache after the last reference (i.e. cache hit at it). It’s up to you to choose which method to implement it. The main point here is that the "oldest" cache line should be kick out from cache when replacement happens.

h. Write Policy
   There are two write policies: write back and write through. Write back means the information is written only to the cache line in cache when memory write happens. The modified cache line is written to memory only when it is replaced. Write through means the information is written to both the cache line in cache and the main memory. Therefore, the main memory always have the most update data. In this project, you should implement the write back policy.

i. Dirty Bit
   The cache line should have a dirty bit to indicate whether the data in it need to be written back to memory in case of replacement.

4. Test Case and Test Environment
   Below is a piece of assembly code, mmm.s

   1 .data
   2 .align 5
   3 .comm a,1024
   4 .comm b,1024
.comm c,1024
.text
.align 5
.globl __start
__start:
    la $16, a  # $16 = &a[0,0];
    la $17, b  # $17 = &b[0,0];
    la $18, c  # $18 = &c[0,0];

    addi $4, $0, 16  # $4 = 16
    sll $5, $4, 2    # $5 = 64
    sll $6, $5, 4    # $6 = 64 * 16

    move $8, $0     # i = 0
__loop0:
    beq $8, $6, __exit0  # i < (16 * 4 * 16) ?
    move $9, $0  # j = 0
__loop1:
    beq $9, $5, __exit1  # j < (16 * 4)?
    move $10, $0  # k = 0
    move $11, $0  # k = 0
__loop2:
    beq $10, $5, __exit2  # k < (16 * 4)?
    add $20, $16, $8  # &a + i * 16
    add $20, $20, $10 # &a + i * 16 + k
    lw $12, 0($20)  # load a[i,k]
    add $21, $17, $11 # &b + k * 16
    add $21, $21, $9  # &b + k * 16 + j
    lw $13, 0($21)  # load b[k,j]
    mul $14, $12, $13 # a[i,k] * b[k,j]
    add $22, $18, $8  # &c + i * 16
    add $22, $22, $9  # &c + i * 16 + j
    lw $15, 0($22)  # load c[i,j]
    add $15, $15, $14 # c[i,j] += a[i,k]*b[k,j]
    sw $15, 0($22)
    addi $10, $10, 4  # (k++)*4
    addi $11, $11, 64 # (k++)*16*4
    j __loop2
__exit2:
    addi $9, $9, 4    # (j++)*4
    j __loop1
__exit1:
    addi $8, $8, 64   # (i++)*16*4

for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++)
        for (k = 0; k < 16; k++)
            c[i * 16 + j] += a[i * 16 + k] * b[k * 16 + j];

This piece of code is performing Matrix Multiplication. The two URLs below provide thorough description of matrix multiplication and its algorithm.

http://mathworld.wolfram.com/MatrixMultiplication.html
http://www-unix.mcs.anl.gov/dbpp/text/node45.html

Because of the perfect data layout of matrix, matrix multiplication becomes a very good method to test the performance of cache. Follow the steps below to setup your test environment and compile the test case.

a. Update the loader.c in the target-pisa directory with the one we provided on our website.
   You can look at the difference between the new copy of loader.c and its original version. Try to figure out the reason of the changes if you are interested. Then rebuild you simulator.
   
   make clean
   make config-pisa
   make sim-fast

b. Download the test program mmm.s from our website or here.

c. Assemble, link and run the test code:
   ~/simplescalar/bin/sslittle-na-sstrix-as -o mmm.o mmm.s
   ~/simplescalar/bin/sslittle-na-sstrix-ld -o mmm mmm.o
   ~/simplescalar/simpleisim-3.0/sim-fast mmm

5. Implementation Choices and Requirements

   a. You can implement the cache simulation module base on the original version of the sim-fast simulator(single cycle datapath) or on the revised 5-stage(multi-cycle datapath) pipeline version from the 2nd phase of the course project.
b. The cache miss latency is 10 cycles and cache hit latency is 1 cycles. This is true for both single cycle datapath and multi-cycle datapath.

c. The cache simulation can be easily turn on or off. When cache is turned off, every memory access latency is 10 cycles.

d. You program should be able to print out some important statistic data. They are:
   1. total number of memory accesses
   2. total number of cache hits
   3. total number of cache misses
   4. total number of cache line replacements
   5. total number of cache line writebacks
   6. total number of clock cycles of the running

e. Because there are only two kinds of memory access operations in the test case mmm.s, you only need simulate the cache behavior for these two memory access instructions. They are "lw" and "sw".

6. What you should hand in

   a. A detailed design report. The report should clearly describe the organization of the cache.

   b. Critical data structures with clear comments.

   c. Provide source code of code that you wrote or modified

   d. Statistical data after running the matrix multiplication program on your simulator.

7. Rubric:

   1. 60% will be based on your report and code modifications
   2. 20% will be based on your demonstration that the program works
   3. 20% will be on your verbal descriptions and operation of the simulator

8. Expected Functionality

   a. The test case mmm.s should be executed on the enhanced sim-fast simulator that has your cache simulation module in it without error.

   b. The enhanced sim-fast simulator should be able to collect these statistic data(see 5.d of the project document or below) when executing mmm.s.

       * Total number of memory accesses issued by the program;

       * Total number of cache hits;
* Total number of cache misses;
* Total number of cache line replacements;
* Total number of cache line writebacks;
* Total number of clock cycles of the running;

c. Show me your source code and explain it to me how it works.

d. The wordload distribution among group members.

9. Hints

a. You can imagine the cache as an array. Each element in the array is a cache line. They looks like this:

```c
struct cache_line {
    unsigned int data[8];
    unsigned int tag : 27;
    unsigned int dirty : 1;
    unsigned int valid : 1;
    unsigned int ref_count : 19;
};

struct cache_line cache[32];
```

b. You can reference the code in file: cache.c and sim-cache.c in SimpleScalar.

Team Work and Project Management

You need to establish a team methodology as in a real world design project. It is mandatory that you elect a project team leader whose function, among other things, is to call the design meetings.

Ethics for Team Work

Although discussion of the project in general terms is to be expected, members of different groups should not exchange source code or project implementation details. Also, it is expected that the contributions of each member of group will be clearly detailed, so care should be taken to make sure that each person contributes a fair amount. Therefore, your team strategy should be geared to prevent bottlenecks where group progress is entirely dependent on one person. A well-considered strategy and a clearly defined interface may mitigate such difficulties.

Reading List

http://www.capsl.udel.edu/~jmanzano/files/