Computer Architecture Project 2

1 Introduction

Since you've finished the first project, i.e. developing a 5-stage pipeline in sim-pipe, you should understand clearly how instructions are executed in a pipelined processor. In your implementation, so far, all instructions are executed in 5 stages and each stage takes one clock cycle. Therefore every instruction except which encounters a hazard takes the same number of cycles, i.e. 5 cycles, to finish its execution, no matter it is a "load", a "store" 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 or 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 component invented to eliminate this gap. I will not elaborate the advantages and principles of cache here. Please refer to Patterson & Hennessy's textbook for details. In this project, you are required to add a cache simulation module in SimpleScalar simulator. To be more specific, your task is to modify the sim-pipe simulator which you developed in the 2nd project to let it simulate the behavior of cache.

2 Design Considerations

2.1 How to Simulate Cache

As defined by Patterson & Hennessy. 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-pipe, 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 data 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 PISA. Pay attention to the macro "READ_WORD" in line 417. This is the exact function/macro at memory access begins. You can wrap this macro and insert you cache lookup code before it. There are many implementation choices. Choose your favorite.

2.2 The difference between Cache Lookup and Memory Reference

As 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 that 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 cache miss occurs, 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 executed. So, you must keep a cycle counter in the simulator. If cache hit occurs, add 1 or 2 cycles (it’s up to you to define how many. For simplicity, you can just specify 1 cycle) to the cycle counter. Otherwise for cache miss, add 10 or 20 cycles (For simplicity, you can just specify 10 cycle). 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 causes cache miss). You will get another cycle number. The difference between the cycle numbers manifests the difference of cache and memory.

2.3 Cache Organization

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

2.3.1 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 leve***l cache is enough.

2.3.2 Associativeness

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

2.3.3 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***.

2.3.4 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.

2.3.5 Cache Line Size and Cache Capacity

The cache line size should be ***16 bytes***. And the cache should have ***16 sets*** in total. Therefore the capacity of the cache is ***1k bytes***.

2.3.6 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.

2.3.7 Replacement Algorithm

You should implement the ***FIFO*** replacement algorithm. FIFO is to replace the earliest cache line in the cache when there is no empty cache line in the cache set. You can implement this algorithm by using a queue or some other methods, all depend on you.

2.3.8 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 in a later time. Write through means the information is written to the cache line in cache and the main memory immediately. Therefore, the main memory always has up-to-date data. In this project, you should implement the ***write back*** policy.

2.3.9 Dirty Bit

The cache line should have a ***dirty bit*** to indicate whether the data in it needs to be written back to memory in case of replacement.

3 Your Mission

1. You should implement the cache simulation module based on the 5-stage pipeline version (sim-pipe) developed in the 1st project.
2. The cache miss latency is 10 cycles and cache hit latency is 1 cycle.
3. The cache simulation should be easily to turn on or off. When cache simulation is turned off, all memory access latency is 10 cycles.
4. Your program should be able to print out some important statistic data as the followings:
   a) Total number of clock cycles of the running.
   b) Total number of memory accesses.
   c) Total number of cache hits.
   d) Total number of cache misses.
   e) Total number of cache line replacements.
   f) Total number of cache line write-backs.
5. Because there are only two kinds of memory access operations in the test case mmm.s. You only need simulate the cache behavior for two memory access instructions: ”lw” and ”sw”.

4 Testcase and Test Environments

```
        .data
        .align   5
        .comm    a,1024
        .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, 4        # $5 = 16 * 16
        sll $6, $5, 4        # $6 = 16 * 16 * 16

        move $8, $0          # i = 0
__loop0:
        beq $8, $6, __exit0 # i < (16 * 16 * 16) ?
        move $9, $0          # j = 0
__loop1:
        beq $9, $5, __exit1 # j < (16 * 16)?
        move $10, $0         # k = 0
        move $11, $0         # k = 0
__loop2:
        beq $10, $5, __exit2      # k < (16 * 16)?
        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++)*16
        addi $11, $11, 64    # (k++)*16*16
        j   __loop2
__exit2:
        addi $9, $9, 4       # (j++)*16
        j   __loop1
__exit1:
        addi $8, $8, 64           # (i++)*16*16
        j   __loop0
__exit0:
        li $2, 1
        syscall
```

Above is a piece of assembly code, mmm.s. The corresponding C code looks like this:

```
#define DIM 16

int a[DIM*DIM];
int b[DIM*DIM];
int c[DIM*DIM];

int
main()
{
    int i,j,k;

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

    return 0;
}
```

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:

1) Download the loader.c file which we provided to the target-pisa directory. And then rebuild you simulator:

```
make clean
make config-pisa
make sim-pipe
```

2) Download the test program mmm.s.

3) 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/simplesim-3.0/sim-pipe mmm
```

5 Implementation Hints

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

```
struct cache_line{
    unsigned int data[4];
    unsigned int tag:27;
    unsigned int dirty:1;
    unsigned int valid:1;
    unsigned int ref_count:19;
};
struct cache_line cache[16];
```

2. You can refer to the code in file: cache.c and sim-cache.c in SimpleScalar.

6 Hand in

1. A detailed design report. The report should clearly describe the organization of the cache. This file should be in this form: student number_DesignDoc.pdf

2. Statistic data after running the matrix multiplication program on your simulator. This file should be in this form: student number_Result.txt

3. Files that you changed: such as sim-pipe.c. Please do not upload files that you have not changed!!