Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Computer Architecture and Assembly Language Lab

Spring 2016

Lab 5

Simulating a Cache

Goal

After completing this lab, you will:

- Know the structure of a cache in a MIPS machine
- Learn how caches affect program performance

Preparation

Please read Chapter 5 in the textbook. This knowledge is required for this lab.

Introduction

1. Basics of Cache

The ***cache*** is a small high-speed memory, usually with a memory cycle time comparable to the time required by the CPU to fetch one instruction. The cache is usually filled from main memory when instructions or data are fetched into the CPU. Often the main memory will supply more words to the cache than the CPU requires, to fill the cache more rapidly and to take advantage of spatial locality. The amount of information which it replaces at one time in the cache is called a ***block (or a line)***. This is normally the width of the data bus between the cache and the main memory. A wide block size for the cache means that several instruction or data words are loaded into the cache at one time, providing a kind of prefetching for instructions or data. Since the cache is small, the effectiveness of the cache relies on the following properties of most programs [1]:

- *Temporal locality*: if an item is referenced now, it will tend to be referenced again soon.
- *Spatial locality*: if an item is referenced, items with addresses that are close by will tend to be referenced soon.

When a cache is used, there must be some ways in which the memory controller determines whether the value currently being addressed in memory is available from the cache or not. There are several ways that this can be accomplished. A possible way to implement a cache memory is to use *direct mapping*. Here, part of the memory address (usually the low order bits of the address) is used to address a word in the cache. This part of the address is called the *index*. The remaining high-order bits in the address, called the *tag*, are stored in the cache memory along with the data.

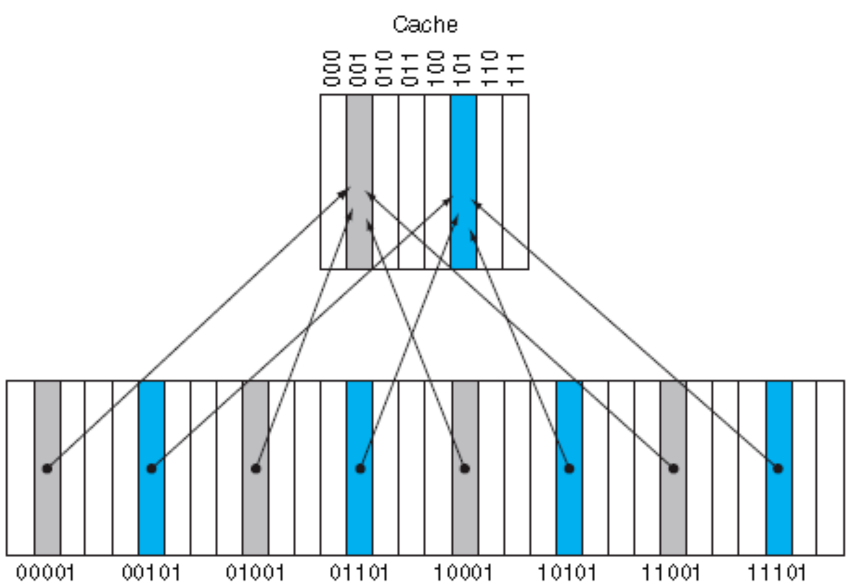

Fig 1. An illustration of the organization of a direct-mapped cache. The first bit in the cache block is a **valid bit** which tells the cache controller if the data in that block are valid or not. Each block in cache is **indexed**, allowing each block to be addressed when the CPU is looking for instructions or data stored in cache. A **tag** identifies which memory location corresponds to that particular block in cache. The tag contains the upper portion of the address, while the lower portion is used in the index. Bits 0 and 1 are not used.

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

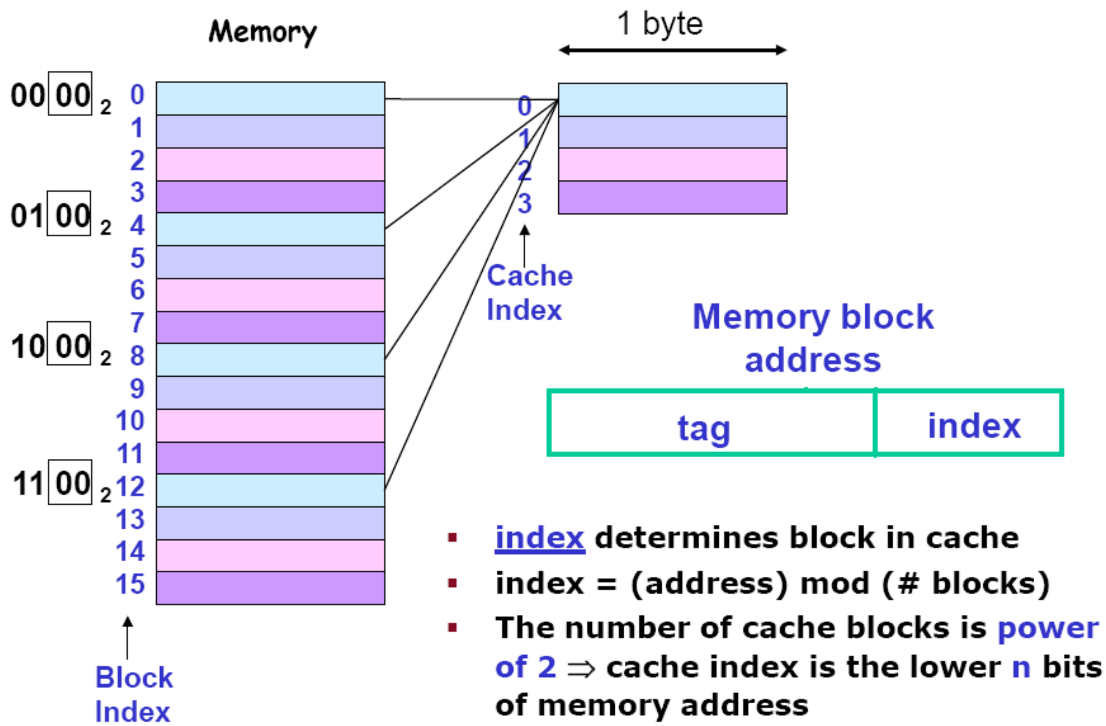

Fig 1. (cont.) Definition of index and tag [4] of a direct-mapped cache (assume one byte per block).

A characteristic of direct mapped caches is that a particular memory address can be mapped to only one cache location. Many memory addresses are mapped to the same cache location (in fact, all addresses with the same index field are mapped to the same cache location.) Whenever a "cache miss" occurs (data needed by the processor is not in cache), the cache block will be replaced by a new block of information from main memory at an address with the same index but with a different tag.

Note that if the program "jumps around" in memory, this cache organization will likely not be effective because the index range is limited. Also, if both instructions and data are stored in cache, it may well happen that both map into the same area of cache, and may cause each other to be replaced very often. This could happen, for example, if the code for a matrix operation and the matrix data itself happened to have the same index values.

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

A more interesting configuration for a cache is the set-associative cache, which uses a *set associative mapping*. In this cache organization, a given main memory location can be mapped to more than one cache location, within a set. Here, each index corresponds to two or more data words, each with a corresponding tag. A set associative cache with n tag and data fields is called an "n-way set associative cache". Usually, for n= 1, 2, 4, 8 are chosen for a set associative cache (n= 1corresponds to direct-mapped cache). Such n-way set associative caches allow interesting tradeoff possibilities: cache performance can be improved by increasing the number of "ways", or by increasing the block size, for a given total amount of memory.

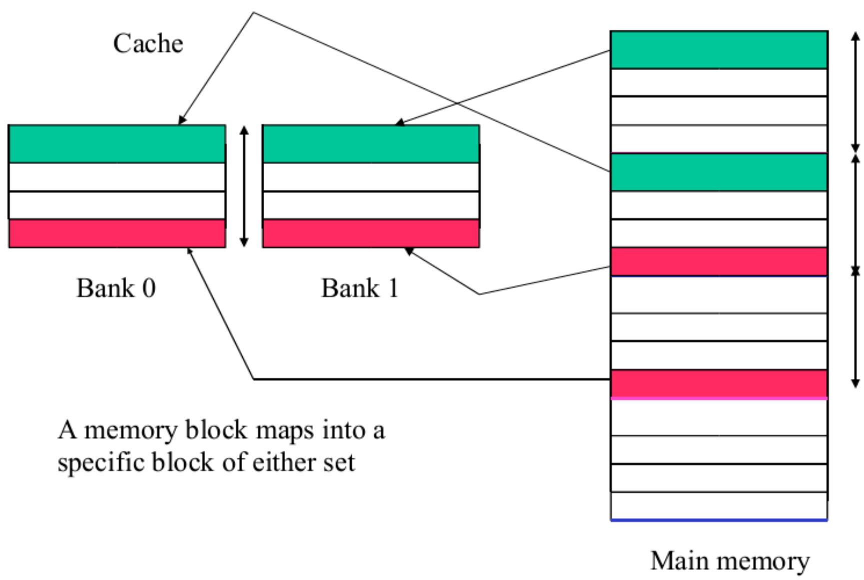

Fig.2 Set-associative cache illustration

In a 2-way set associative cache, if one data word is empty for a read operation corresponding to a particular index, then it is filled. If both data words are already filled, then one must be overwritten by new data. Similarly, in an n-way set associative cache, if all n data and tag fields in a set are filled, then one value in the set must be overwritten, or replaced,. Note that an entire block must be replaced each time (but not the entire set). The most common replacement algorithms are:

- Random --- the location for the value to be replaced is chosen at random from all n of the cache locations at that index position. In a 2-way set associative cache, this can be accomplished with a single modulo 2 random variable obtained, say, from an internal clock.

- Least recently used (LRU) --- here the value which was actually used least recently is replaced. In general, it is more likely that the most recently used value will be the one required in the near future (due to temporal locality). For a 2-way set associative cache, this is readily implemented by setting a special bit called the **`USED bit`** for the other word when a value is accessed while the corresponding bit for the word which was accessed is reset. The value to be replaced is then the value with the **`USED bit`** set. This replacement strategy can be implemented by adding a single **`USED bit`** to each cache location. The LRU strategy operates by setting a bit in the other word when a value is stored and resetting the corresponding bit for the new word. For an n-way set associative cache, this strategy can be implemented by storing a modulo n counter with each data word.

Given an address, if the address is in cache, it is called a **`hit`**; otherwise, it is called a **`miss`**. To check if the address is in the cache, we use the following procedure:

1. Use the set index to determine which cache set the address should reside in.
2. For each block in the corresponding cache set, compare the tag associated with that block to the tag from the memory address that is needed. If there is a match, proceed to the next step. Otherwise, the data is not in the cache.
3. For the block where the data was found, look at the valid bit. If it is 1, the data is in the cache, otherwise it is not.

Cache memories normally allow one of two things to happen when data is written into a memory location for which there already is a value stored in cache:

- **`Write through cache`** -- both the cache and main memory are updated at the same time. This may slow down the execution of instructions which write data to memory, because of the longer write time to main memory. Buffering memory writes can help speed up memory writes if they are relatively infrequent, however.
- **`Write back cache`** --- here only the cache is updated directly by the CPU; the cache memory controller marks the value so that needs to be written back into main memory when the word at that location is removed from the cache. This method is used because a memory location may often be altered several times while it is still in cache without having to write the value into main memory. This method is often implemented using an “**`ALTERED” bit`** (or dirty bit) in the cache. The **`ALTERED bit`** is set whenever a cache value is written into by the processor. Only if the **`ALTERED bit`** is set is it necessary to write the value back into main memory (i.e., only values which have been

altered must be written back into main memory so to insure that the data is not lost). The value should be written back immediately before the value is replaced in cache.

2. *Spim-Cache* Tutorial

You can download this simulator (Spim-Cache Windows Executable) from

http://www.disca.upv.es/spetit/spim.htm

In this lab, we will use a new simulation tool, called ***Spim-Cache***. *Spim-cache* is an educational tool to perform cache simulation. It is based on the Spim MIPS simulator, and it is similar to

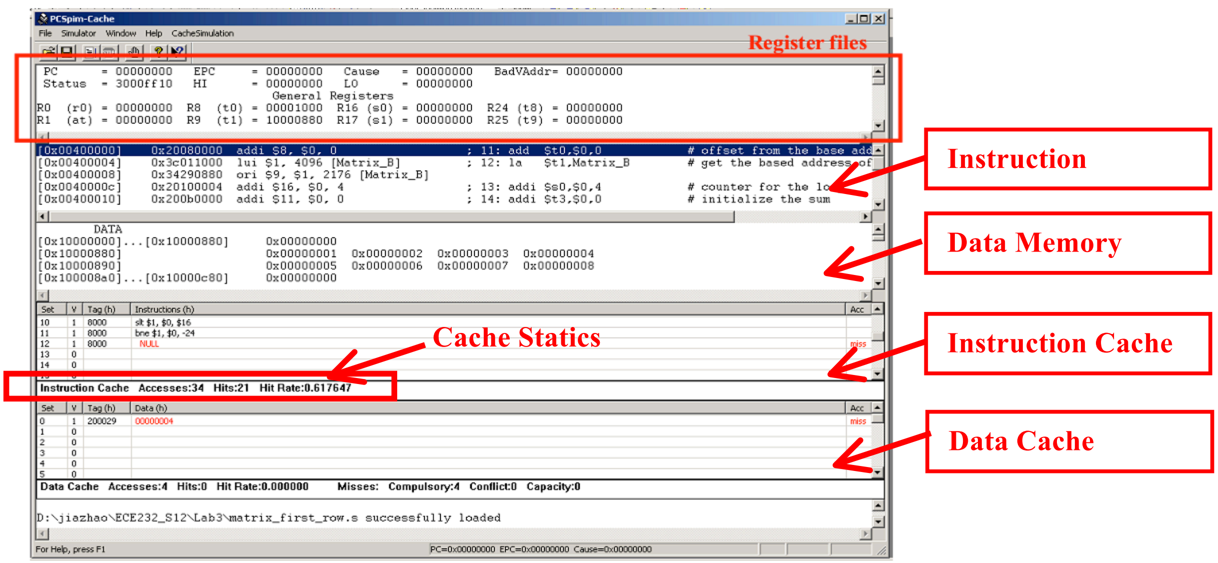

Fig. 3 The window of Spim-Cache

QTSpim. It can adjust the configurable cache parameters and visualizes cache behavior and cache statistics. *Spim-cache* permits the simulation of a data cache, an instruction cache, or both (i.e., Harvard Architecture). To display the contents of the cache or caches being simulated, the user interface extends the main window of *PCSpim* by adding one or two new frames.

To start the cache simulation, students must select the **`Cache Simulation`** option in the **`Cache Settings`** dialog, which pops up after clicking on the **`Settings`** entry of the **`simulator`**. Then, a dialog with different cache configurations is displayed. The **`Cache Settings`** dialog, shown below, allows students to choose the cache configuration, that is, cache size, block size, RAM to cache mapping functions, etc. This dialog is accessible from the **`Cache Simulation`** menu option. If the **`Show Rate`** option is selected, some statistics are displayed in a small frame below the cache frame.

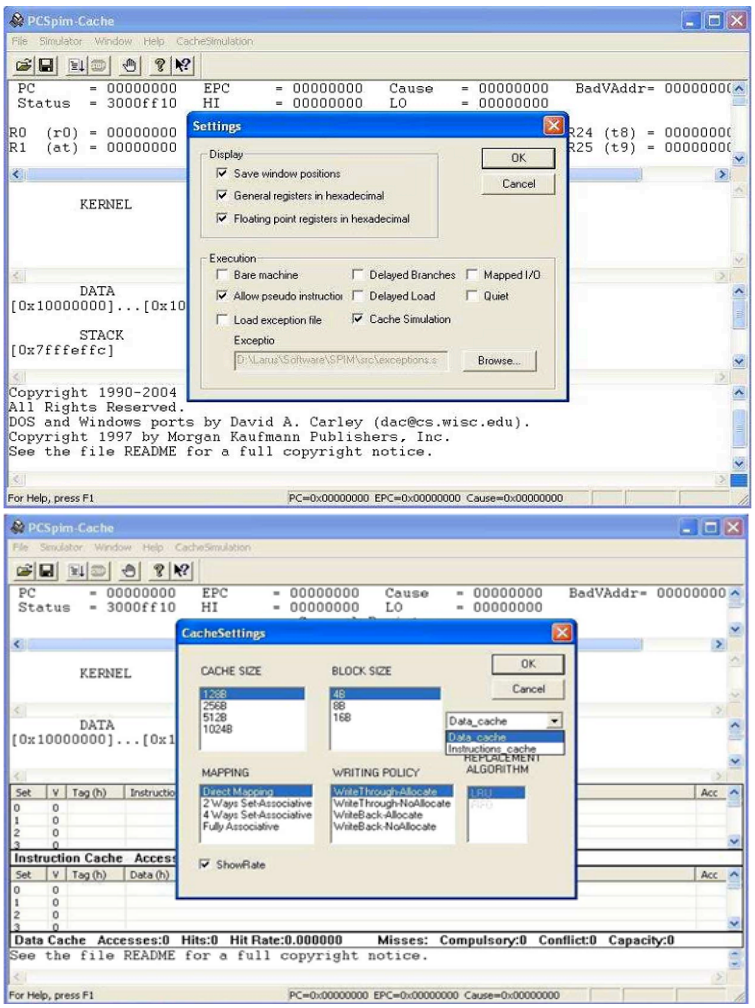

Fig.4 Get started with *Spim-Cache*

To run a program in *Spim-Cache*, you need to do the following:

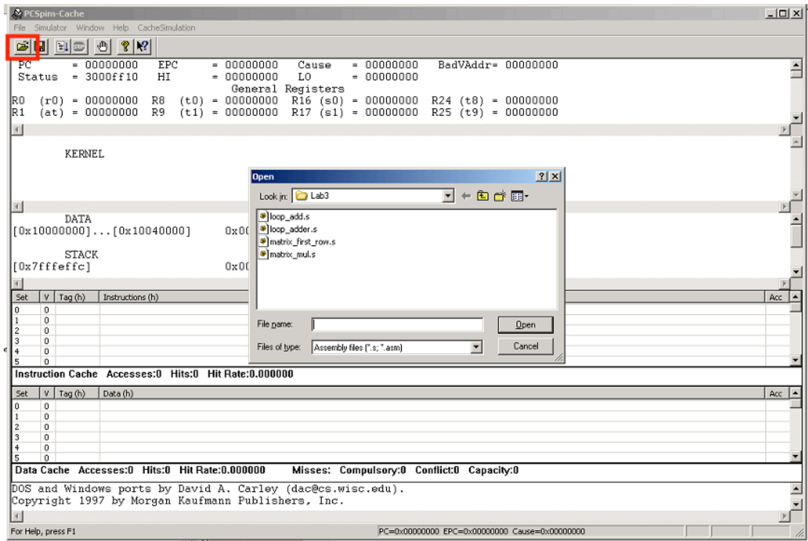

Fig. 5 Open an assembly program on Spim-cache

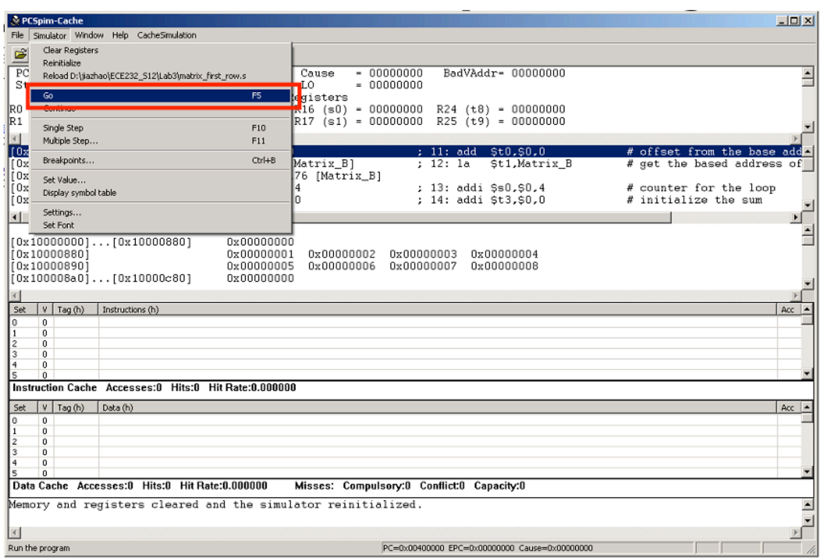

Fig. 6 Run the program

In the basic *PCSpim* simulator a step of execution covers the entire execution of a single instruction. The cache simulation extension modifies the semantic of memory reference instructions, i.e., loads and stores. A memory reference instruction that hits into the cache takes only one step (as a normal instruction) to execute. Nevertheless, because of pedagogical reasons, load and store misses are handled with a different number of steps. A load miss is handled in three steps: 1) detecting and marking the miss in the corresponding set (all blocks in the set are marked); 2) fetching the block from main memory; and 3) loading the requested data into the corresponding register. A store miss is handled in two or three steps depending on the selected write miss policy (allocate or no-allocate). With the no-allocate policy the steps are: 1) detecting and marking the miss; and 2) storing the content of the register in the main memory. With the allocate policy the steps are: 1) detecting and marking the miss in the corresponding set; 2) fetching the block from main memory; and 3) storing the corresponding data into the cache. Finally, with respect to the instruction cache, a miss is handled analogously to a load miss in the data cache. In this case, the third step is the execution of the loaded instruction.

Assignment 1

The following program counts the sum of even numbers inside a given array:

```
# Calculate the sum of even numbers inside a given array
.data 0x10000480
ArrayA:
    .word 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

.text
.globl main
main:
la $t1, ArrayA
li $t2, 0           # count = 0
li $t3, 12          # number of elements in ArrayA
li $t4, 1           # mask
loop:
    lw $t5, 0($t1)
    and $t6, $t5, $t4
    beq $t6, $t4, cont
    add $t2, $t2, $t5
cont:
    addi $t1, $t1, 4
    addi $t3, $t3, -1
    bgt $t3, $0, loop

li $v0, 10
syscall
```

(a) Load the program above into SPIM-CACHE. Configure the cache settings to be 256-B size, 16-B line size, and four-way set associative. Run the program and record the results (i.e. the status of each register).

(b) Write down the contents of data cache after the program is completed. Explain in details how the elements of Array_A are mapped to each slot in the data cache.

(c) Write down the contents of the instruction cache after the program is completed. Explain in detail how the contents are obtained through executing the program.

Assignment 2

Write a program to calculate and print the Kronecker product of two following vectors:

$A = [1, 2, 3]$ $\qquad$ $B = [8,7,6]$

Note that the Kronecker product of two vectors A with size m and B with size n is defined as:

$$C = A \otimes B = [a_1 b_1, \ldots, a_1 b_n, a_2 b_1, \ldots \ldots, a_m b_1, \ldots, a_m b_n]$$

where $C \in \mathrm{R}^{mn}$ and your program should output the value of C[0], C[1],..., C[8] in that order.

A and B are defined in the program as follows:

```
.data 0x10000480
A:
.word 1,2,3
B:
.word 8,7,6
```

Execute the program based on the cache settings of "256-B size, 16-B line size, four-way set associative". After the program finishes, record the results in the user space's memory address. Write down the contents of the data cache and explain in details how the elements of A and B are mapped to each slot in the data cache.

Assignment 3

Write an assembly code program to perform sven-dimensional cross-product of two vectors **X** and **Y**, and store the result into a vector **Z**, i.e.,

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

$$Z = X \times Y = T_X \cdot Y = \begin{bmatrix} 0 & -x_4 & -x_7 & x_2 & -x_6 & x_5 & x_3 \\ x_4 & 0 & -x_5 & -x_1 & x_3 & -x_7 & x_6 \\ x_7 & x_5 & 0 & -x_6 & -x_2 & x_4 & -x_1 \\ -x_2 & x_1 & x_6 & 0 & -x_7 & -x_3 & x_5 \\ x_6 & -x_3 & x_2 & x_7 & 0 & -x_1 & -x_4 \\ -x_5 & x_7 & -x_4 & x_3 & x_1 & 0 & -x_2 \\ -x_3 & -x_6 & x_1 & -x_5 & x_4 & x_2 & 0 \end{bmatrix} \cdot Y$$

where $X = [1,2,3,4,5,6,7]^T$ and $Y = [4,5,6,7,8,9,10]^T$.

Use Spim-Cache simulator to simulate your assembly code program. You will need to run your program multiple times with *different* configurations for the data cache and monitor the performance of the data cache.

All elements in vector **X**, **Y** and **Z** are stored in words. The seven elements in vector **X** are stored in consecutive memory spaces. The base address (address of the first word) is **`0x10000860`**$_{\text{hex}}$ (use **`.data 0x10000860`**).

Calculate the matrix T_x and store the values in the memory (It is ok if you derive the value by hand and write the value to the code file directly). Elements in each row of matrix are stored in consecutive memory spaces, while seven rows in matrix T_x are NOT stored in consecutive memory spaces. The base address of each row is shown below in Table 1. The base address of two consecutive rows is separated by 1024_{10} bytes (400_{16} bytes).

Then Calculate the matrix **Z** using matrix production. The eight elements in vector Z are also stored in consecutive memory spaces. The base address is **`0x10000840`**$_{\text{hex}}$.

In this lab, you will deal with the multiplication of two 32 bit values, for example, the value stored in **`$t0`** and **`$t1`** respectively. The low 32 bits of the result (64 bit in total) is stored in register **Lo** then you need to move it to register **`$t2`**.

```
mult $t0,$t1     # calculate $t0 × $t1, result is 64 bit

mflo $t2         # move the low 32 bit of the result to $t2
```

After running the program, record the **screenshot** of vector **`C`** in the **data memory** from **`0x10000840`** to **`0x10000860`**.

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Part of the assembly code is given below:

```
.data 0x10000860
Vector_X: .word 1,2,3,4,5,6,7
          .data 0x10000880
Vector_Y: .word 4,5,6,7,8,9,10
          .data 0x10000C80
Matrix_T: .word 0,0,0,0,0,0,0
          .data 0x10001080
          .word 0,0,0,0,0,0,0
          .data 0x10001480
          .word 0,0,0,0,0,0,0
          .data 0x10001880
          .word 0,0,0,0,0,0,0
          .data 0x10001c80
          .word 0,0,0,0,0,0,0
          .data 0x10002080
          .word 0,0,0,0,0,0,0
          .data 0x10002480
          .word 0,0,0,0,0,0,0
          .data 0x10002880
Vector_Z: .word 0,0,0,0,0,0,0,0

          .text 0x00400000
          .globl main       # main program starts in the next line
main:

     #Your code starts from here
```

Assignment 4

Write a program to transpose a given matrix A.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 \end{bmatrix}$$

Please write the program to calculate A transpose.

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

The original and destination matrices and their addresses are defined as follows:

```
.data 0x10000800
      OrinRow_0:
            .word 1,2,3,4,5,6
      OrinRow_1:
        .word 7,8,9,10,11,12
      OrinRow_2:
            .word 13,14,15,16,17,18
      OrinRow_3:
            .word 19,20,21,22,23,24
      OrinRow_4:
            .word 25,26,27,28,29,30
      OrinRow_5:
            .word 31,32,33,34,35,36
            .data 0x10000900
      TransRow_0:
            .word 0,0,0,0,0,0
      TransRow_1:
      .word 0,0,0,0,0,0
      TransRow_2:
            .word 0,0,0,0,0,0
      TransRow_3:
            .word 0,0,0,0,0,0
      TransRow_4:
            .word 0,0,0,0,0,0
      TransRow_5:
            .word 0,0,0,0,0,0
    .text 0x00400000
      main:
            #Your code starts from here
```

Assignment 5

Run the previous program on *Spim-Cache* under three different cache configurations: 1) Cache size 128B, block size 4B, direct-mapping; 2) Cache size 128B, block size 8B, direct-mapping; 3) Cache size 128B, block size 16B, direct-mapping.

Record the miss rate of each configuration, compare the results between them and analyze the reasons for your results.

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Experiment report

Write a proper report including your codes, results (snapshot of output) and the conclusion of assignments and convert it to pdf format. Please also attach the code files (*.s,*.asm) to Sakai together with the report. Each lab report is due before the start of the next lab. Please include your name and Student ID in both report and the code.

References

1. Patterson and Hennessy, "Computer Organization and Design: The Hardware / Software interface", 4th Edition.
2. Daniel J. Ellard, "MIPS Assembly Language Programming: CS50 Discussion and Project Book", September 1994.
3. J. Sahuquillo, N. Tomás, S. Petit and A. Pont. Spim-Cache: A Pedagogical Tool for Teaching Cache Memories through Code-Based Exercises. IEEE Transactions on Education, Vol. 50, No. 3, August 2007.
4. "Tutorial on Spim-cache", http://www.ecs.umass.edu/ece232/resource/spim-cache_tutorial.pdf
5. Download link for Spim-cache simulator, http://www.disca.upv.es/spetit/spim.htm
6. http://ecs.victoria.ac.nz/Courses/NWEN242_2014T2/WebHome