Computer Systems Architecture 2019/20
Laboratory Exercise – Cache memory simulator

1. Introduction

In this laboratory exercise you will write a C program to simulate the operation of a fully associative cache memory on an embedded processor. You will use your simulation program to analyse the performance of the cache when executing cross correlation and bubble sort algorithms. You will also explore the impact of memory block size on cache performance.

You will work individually on this laboratory exercise; submitting a C program, a results file and an individual report.

2. Embedded system memory architecture

The embedded processor is interfaced to a 128 Ki x 16-bit external data memory using a 20-bit address bus and a 16-bit data bus. The embedded processor contains a fully associative 256 x 16-bit cache memory for data accesses, and the cache can be configured in one of the 8 modes listed in Table 1.

The cache memory uses a *FIFO* block replacement policy, and a *write-back*, *write-allocate* policy for cache write misses.

Mode ID	Cache block size (16-bit words)	Number of cache lines
1	1	256
2	2	128
3	4	64
4	8	32
5	16	16
6	32	8
7	64	4
8	128	2

Table 1. Cache memory configuration modes

3. Memory trace files

You are provided with two memory trace files that have been created by recording the read and write memory accesses when executing cross correlation and bubble sort algorithms on the embedded processor.

The source code for the cross correlation algorithm is presented in Appendix A, and was executed with input signal length of 500 samples. The bubble sort algorithm is presented in Appendix B, and was executed using an array of 700 random 16-bit values.

The memory trace files only record array accesses, and all other variables are stored in processor registers.

The memory trace files are ASCII text files. Each line begins with an 'R' or 'W' character to signify a memory read or memory write respectively. The 'R' or 'W' character is followed by a single space, and the memory address accessed. The address is represented as a hexadecimal number.

Individual memory trace files have been produced for each student, and they can be found in in the *Laboratory Materials/Memory Trace Files* content item on Blackboard. They are also available via the Dropbox link http://tiny.cc/wmvrjz

The filenames of your specific trace files can be found in the *Cross_Correlation_Trace_File* and *Bubble_Sort_Trace_File* columns in Blackboard Gradebook.

A small test trace file, *test_trace.trc*, has been provided to help you test your trace file read function.

4. Cache simulator

Write a program in C to simulate the operation of the data cache in the embedded processor. You may wish to first write a program to simulate the configuration where the cache has 256 lines and a block size of 1 16-bit word. You can then enhance this program to simulate the other cache configuration modes shown in Table 1.

The cache simulator must be written as a single source file using ANSI standard C. The program must include a header which includes your name and student ID number. The program must be clearly commented. The bubble sort algorithm presented in Appendix B is a good example of how to comment a program.

Your program must be suitable to be run from a terminal window, providing its formatted output to the terminal window.

The simulator must output its results in the following comma separated variable (CSV) format:

trace_file_name, *mode_ID*, *NRA*, *NWA*, *NCRH*, *NCRM*, *NCWH*, *NCWM*

trace_file_name	The name of the trace file being analysed (without the folder path)
mode_ID	The ID number of the cache configuration mode (1 … 8)
NRA	Total number of read accesses to the external memory
NWA	Total number of write accesses to the external memory
NCRH	Number of cache read hits
NCRM	Number of cache read misses
NCWH	Number of cache write hits
NCWM	Number of cache write misses

The *mode_ID*, *NRA*, *NWA*, *NCRH*, *NCRM*, *NCWH* and *NCWM* values should all be formatted as unsigned integers.

5. Cache performance simulations

For each of the configuration modes listed in Table 1, you should simulate the performance of the cache memory for the cross correlation and bubble sort algorithms using your allocated memory trace files.

6. Assessment

There are 4 elements to the assessment of the laboratory exercise:

- laboratory attendance and engagement,
- the cache simulation program,
- the results from the cache simulation program,
- a short written report presenting the cache performance results and their analysis.

6.1. Laboratory attendance and engagement

You will have been timetabled to attend two 3-hour laboratory sessions. These sessions are to support you in writing the cache simulator and analysing the performance results. You are required to attend both sessions, and to demonstrate that you are making satisfactory progress with the laboratory exercise. Failure to attend both scheduled laboratory sessions and demonstrate satisfactory progress may result in a mark of zero for the laboratory exercise.

You should ensure that you have discussed your progress with one of the postgraduate demonstrators before leaving the computer cluster, and in good time before the end of each session.

6.2. Cache simulation program

The cache simulation program must be submitted to Blackboard by midnight on the day of your second scheduled laboratory attendance. The submission link can be found in the *Laboratory Materials* folder in Blackboard. The program will be submitted to a source code plagiarism detection tool. The program will be compiled and run to verify that it executes correctly, and so please ensure that this is possible from the single source file submitted.

The criteria for the assessment of the program, together with indicative percentage marks, are given below:

- Use of appropriate structured programming techniques 10%
- Clearly commented 10%

6.3. Results from the cache simulation program

The formatted results from the cache simulation program must also be submitted to Blackboard by midnight on the day of your second scheduled laboratory attendance. The results must be submitted as a single ASCII file (with 16 lines) formatted as described in section 4, and using the file extension .csv The submission link can be found in the *Laboratory Materials* folder in Blackboard.

The criteria for the assessment of the results, together with indicative percentage marks, are given below:

- Numerical correctness 30%

6.4. Cache performance report

You are required to write a short report presenting the performance results from the cache simulations together with your analysis of these results. The report must be submitted as a PDF document (via Turnitin) to Blackboard by 5pm on 21st April 2020.

The report must present the performance results in both tabular and graphical formats.

The formatting requirements of the PDF document are described below:

- Paper size: A4
- Margins: no smaller than 2cm
- Font: Arial, Calibri or Verdana with the font size no smaller than 11pt
- Line spacing: no less than 1.0
- Length: no more than 4 A4 sides

The report must include your name and your student ID number. There is no requirement for a title page.

The criteria for the assessment of the report, together with indicative percentage marks, are given below:

- Clarity of results presentation (tabular and graphical) 10%
- Analysis of performance results 30%
- Overall quality of presentation and adherence to formatting requirements 10%

When analysing the results you will need to refer to the pattern of memory accesses when executing cross correlation and bubble sort algorithms on the embedded processor.

You may also wish to consider the impact of the access times of the main memory and the cache memory. For the purposes of this analysis you should assume that the main memory has access time of 100 ns, and the cache memory has an access time of 10 ns.

Peter R Green
9th February 2020.

Appendix A – Source code for Cross Correlation Algorithm

```
/*
 * Filename:      CrossCorrelation.c
 * Author:        Peter Green
 * Date:          22 February 2019
 *
 */

void CrossCorrelation (short int *signal1, short int *signal2,
                        short int *correlation, int signal_length) {

      // array index variables for two input signals and the correlation output
      int overlap;
      int signal1_index, signal2_index
      int correlation_index = 0;

      // compute the first half of the correlation function
      for (overlap = 1; overlap <= signal_length; overlap++) {

            correlation[correlation_index] = 0;
            signal2_index = signal_length - overlap;

            for (signal1_index = 0; signal1_index < overlap; signal1_index++) {

                  correlation[correlation_index] += signal1[signal1_index] *
                                                      signal2[signal2_index];
                  signal2_index++;
            }
            correlation_index++;
      }


      // compute the second half of the correlation function
      for (overlap = signal_length - 1; overlap > 0; overlap--) {

            correlation[correlation_index] = 0;

            signal2_index = 0;

            for (signal1_index = signal_length - overlap;
                    signal1_index < signal_length; signal1_index++) {

                  correlation[correlation_index] += signal1[signal1_index] *
                                                      signal2[signal2_index];

                  signal2_index++;
            }
            correlation_index++;
      }
}
```

Appendix B – Source code for Bubble Sort Algorithm

```
/*
 * Filename:      BubbleSort.c
 * Author:        Jack Andrews
 * Student ID:    123456789
 * Date:          20 February 2019
 *
 */

void BubbleSort(short int *array, int length) {

      // Temporary variable used for swapping of elements
      short int temp;

      // Loop counters
      int i, j;

      /* Pass over the whole array on the first iteration. On subsequent
       * iterations, ignore the already sorted upper elements, achieved by
       * decrementing i on each iteration of the outer for() loop.
       */
      for (i = length-1; i >= 0; i--) {

            /* The inner for() loop iterates over the remaining array elements,
             * comparing each and swapping if necessary.
             */
            for (j = 0; j < i; j++) {

            /* Compare the value at index j in the array with the value at
             * index j+1. If array[j] > array[j+1], then swap the elements.
             */
                  if (array[j] > array[j+1]) {

                        // Swap the array elements
                        temp = array[j];
                        array[j] = array[j+1];
                        array[j+1] = temp;

                  }

            }
```