Lab 10. Evaluating the performance of C and assembly functions

Introduction

We have claimed in class before that assembly functions can be used to improve a program's performance in terms of execution time. In this lab, we will look at the performance evaluation problem of the C and assembly functions implementing the same algorithm. We will also evaluate the performances of different versions of C and assembly functions performing the same job using different algorithms.

We use two approaches to evaluate performances of C/assembly functions using the ARM simulator in Keil (MDK-ARM):

- Enabling the *Execution Profiler* using the *Show Time* in Keil to see the time spent on each statement.
- Inspecting the *Internal States* of the microprocessor, which is, in fact, the number of clock cycles recorded by the ARM simulator in Keil.

To perform the performance evaluation on a fairground, we use mainly the functions provided for this lab. Please download the enclosed `.zip` file and unzip it in your project folder.

All the functions for which we will evaluate performance in this lab are for the calculations of *the number of 1's* in a *uint32_t* word. Some of these functions have been exposed to you in the class lectures and other labs. We will consider the following three algorithms here:

- Algorithm 1—checking one bit in each loop.
- Algorithm 2—checking two bits in each loop.
- Algorithm 3—eliminating a single *1* in each loop.

The above algorithms are implemented in both C and assembly, as seen in the lab10_c_functions.c and lab10_asm_functions_prob.s files, which can be found in the `source` folder of the given files of the lab. (You are encouraged to read these functions carefully to try to understand them fully.) To facilitate prototyping these functions, the corresponding header files are created and are also included in the `source` folder.

Two approaches of performance evaluation in Keil

There are many different ways of evaluating the performance of the statements and functions in Keil.

The easiest approach may be the *Execution Profiler*, which can provide the execution time for each line of code shown beside the source code.

- To enable the Execution Profiler, we need to click the following chain of choices in the Debug mode of Keil: **Debug → Execution Profiling → Show Time**.

- To see the Execution Profiler, we just need to run the code and stop it. The time spent on each line of code will be displayed in the left of the source code in the debug window.

The Execution Profiler can provide a quick view of the efficiency for each statement, without considering the time spent on the execution of the statement if the statement calls a function. To count the number of clock cycles used for each function, we can use the *Internal States* of the simulator.

- To enable the display of the Internal States in the debug mode, we need to expand the **Internal** class in the **Register** tab on the left pane.
- To read the number of clock cycles of a statement, we need to set up a breakpoint in front of the statement and another one immediately after it. When the program halts at the statement, we read the value of the States; when the program halts at the next statement, we read another value of the States. The difference between the two States is the number of clock cycles that the statement/function takes.

We will use both the Execution Profiler and the Internal States in this lab.

Lab preparation

lab10_performance_evaluation.zip

- Download the zipped project file and unzip it to your lab project folder.
- Build the project to see if there are any errors or warnings; if you see them, please let the TA/instructor know ASAP.
- Make sure that the clock of the MCU is set up to 1.0 MHz in **Options for Target… → Xtal (MHz)**. With this setup, ONE microsecond will correspond to ONE clock cycle, making the counting easy.

Lab tasks

Task 1. Determining the execution time of each statement using the Execution Profiler

(20 points) In this task, we use the Execution Profiler to see the efficiency of the C statements—the efficiency of the implementation of C statements using assembly language by the compiler.

- Enter the debug mode.
- Enable the Execution Profiler. (Under the Debug menu)
- Run the program without using breakpoints.
- Stop the program and see the time spent on each statement.
- Record in your lab report the time spent on the following lines:

```
// Test C functions
    num_of_1a = C_number_of_1s_alg1(A);
    num_of_1b = C_number_of_1s_alg1(B);
    num_of_1c = C_number_of_1s_alg1(C);
    printf("C Alg 1: A: %d, B: %d, C: %d \n", num_of_1a, num_of_1b, num_of_1c);
```

- Determine how many clock cycles are used in the four executable statements in the above code snippet without considering the time spent on the execution of the function itself. Report the results in your lab report.

Task 2. Determining the execution time of a block of code including the functions within the block using the Internal States

(30 points) As we mentioned before, the Execution Profiler does not include the time used by the function called in a C statement. To the function execution time, we need to use the Internal States. Note that the execution time of some of the above functions is data dependent. Hence, we need to check their performance for different data; this is the reason why we need to run each function three times, with three different input variables.

- Enter the debug mode.
- Enable the Internal States.
- Set up a breakpoint before each function call in the `Evaluate C functions` and `Evaluate assembly functions` sections; the total should be 18. Also, set up a breakpoint at each `printf` command in the above sections.
- Record the Internal States at each breakpoint. Then calculate the number of clock cycles used for each function call. Provide your results in the lab report using a table like the one listed below.
- Observe that
    - The C functions corresponding to different algorithms have different efficiencies.
    - An assembly function using a certain algorithm is more efficient than it's C counterpart. Sometimes the efficiency improvement is very significant.
- Record the efficiency improvements of the assembly functions over their C counterparts in your lab report for all the three algorithms. Clearly define your rubric for the improvements.

Run this counting of the execution-clock cycles one more time after the programming assignments below, as detailed later.

	Input A	Input B	Input C
Alg1, C			
Alg1, asm			
Alg2, C			
Alg2, asm			
Alg3, C			
Alg3, asm			

Task 3 Programming in C and assembly to do a more detailed performance comparison of the above functions

(30 points) The code we (have) run so far only supports our claims in special cases of input data. To be more convincing, we need to run the above functions for a large amount of input data, which should be random in nature. To that end, we need to write some supporting code running the above functions over the dataset. The supporting code in C for the C function of Algorithm 1 is given already. You need to write the corresponding test code for the C functions of Algorithm 2 and 3 as your C assignment. You also need to write test code in assembly for the assembly function of Algorithm 2. The placeholder code snippets for these test code are given already.

1. C assignments

(10 points each) The instructions of the C assignment are given in C program assignments A and B in the `lab_main_prob.c` file of the lab. Follow them to finish your assignment in the file.

Copy the C code you write, including the `for` loop statements, into your lab report.

2. Asm assignments

(10 points) You need to write an assembly function, the name of which is given in the code, to perform the same test for the assembly function for Algorithm 3 as the C tests—you have to call `asm_number_of_1s_alg3` `NUM_ELE` times and save the results of the number of 1's obtained in each input data to the corresponding location of `array_nof1d`. An example is given for calling `asm_number_of_1s_alg2` `NUM_ELE` times. Finish your assignment in the `lab10_asm_functions_prob.s` file. A placeholder is given. Please change the number of registers in the `PUSH` and `POP` instructions according to your real usage of the registers.

Also, copy your code (the entire block of code of `asm_number_of_1s_alg3_asm_caller`) into your lab report.

Task 4. Counting the execution clock cycles for each of the testing block code given/produced above.

(20 points) Perform the following for this task:

- Make sure that you have the same results for all the FOUR test cases using the `printf` results.
- Create breakpoints on those lines of `breakPointHere++;` to calculate the execution time for the above four cases. List your results in a table in the report similar to the one given above.

Note that for a real MCU, we can use the CYCCNT register to count the number of cycles used for executing a function. For details see https://stackoverflow.com/questions/32610019/arm-m4-instructions-per-cycle-ipc-counters [https://stackoverflow.com/questions/32610019/arm-m4-instructions-per-cycle-ipc-counters].