#cec32xNotes, cec32x_lab_16_Lab2-advanced_topics_of_Keil

Lab 2. Advanced Topics of Keil

Introduction

We will be learning assembly programming soon in the class. One of the advantages of assembly programming is that it is faster than C, especially for applications using specific hardware of the microprocessor, as will be shown later in the course.

We mentioned in the class that a good approach to learning programming is to read code first and then try to figure out the meaning of each statement.

Due to the above two points, 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. We will introduce the algorithms and their implementation later in the class. Here, we just give the implementation first. 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 lab02_c_functions.c and lab02_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 as much as possible.) 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 on 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

- Download the zipped project file and unzip it to your lab project folder. Change the folder name and file names as needed.
- 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 for Performance Evaluation

Task 1. Determining the Execution Time of Each Statement Using the Execution Profiler

(15 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("Results from C Alg 1:\n");
printf("  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

(25 points)

As we mentioned before, the **Execution Profiler** does not include the time used by the function called in a C statement. To find 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` function 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.

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

Lab Tasks for Programming

Task 3. Counting the Number of Figures in a 32-bit Number

(40 points)

We have learned that we can calculate the number of 1's in a binary number. With the number of total bits and the number of 1's in the binary number, we can easily calculate the number of 0's in that number. Now, we extend the problem to count the number of occurrences of 0, 1, 2, ... up to 9 in a 32-bit number (`uint32_t`).

To be consistent, please use the following prototype of the function, which has to be added to the header file:

```
void C_number_of_0_to_9s(uint32_t x, uint32_t result_arr[10]);
```

You can declare an array of 10 to save the results, as shown below:

```
uint32_t array_n_of_0_to_9[10];
```

Task 4. Printing the Counting Results

(10 points)

To verify your function, you can try with the input parameter being `55320`. Please print your results similar the following (it does not have to match exactly):

```
Counting for the decimal digits in 55320:
#0 : 1, #1 : 0, #2 : 1, #3 : 1, #4 : 0
#5 : 2, #6 : 0, #7 : 0, #8 : 0, #9 : 0
```

Please also print the result for the input parameter being `0`. Print out the results similar to the above as well.

Submission

(10 points)

Submit the following on Canvas:

- Report: A **PDF** containing your measured results, screenshots of the code you wrote for Task 4.
- Code: A **ZIP** containing the Keil project and source code for this lab assignment.

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.