

CS 301, Course Project - Report

Abhishek Raj (180010002), Harsh Raj (180010017), Rishit Saiya (180010027)

October 7, 2020

1 Abstract

The report touches upon an [ARM](#) project where we are given an array of unpacked BCD numbers are stored starting from memory location U_BCD. We have converted the unpacked BCD numbers to packed BCD numbers and stored the array starting from memory location P_BCD.

We have also explained upon some the problem statements explaining in detail about parameters like Number of Static & Dynamic Instructions, Instruction Counts under various constraints, Code Coverage, Execution Time, etc. It also explains in details of our previous versions/ideas and the breakpoints where we chose a different idea to enhance and achieve the results.

Note about Codes:

main.s contains the main ARM code (This code is written without conditional execution).
main_thumb.s contains the code using THUMB instructions.

2 Simulator & Software Dependencies

ARM Assembly Language Programming is used to code in the **.asm** file. We are using [NXP's ARM Cortex M3 – LPC1768](#) for simulation of our program. All the statistics mentioned in this report are in state-of-art to the software packs for LPC1768 updates to this date. We have performed simulation using [Keil SDE](#) which provides a wide range of ARM Cortex-M based microcontroller devices.

3 Q.1 - Static & Dynamic Instructions

The basic definitions of Static & Dynamic Instructions are as follows:

Static Instruction: The number of instructions of the program will constitute number of Static Instructions.

The number of Static Instructions of our code is **20**.

Dynamic Instruction: The actual number of instructions executed by the CPU for a specific program execution will constitute as number of Dynamic Instructions.

The generalised expression for number of Dynamic Instructions is as follows:

$$\text{Dynamic Instructions} = \frac{5n^2 + 29n}{2} + 9$$

where n is the number of unpacked BCD numbers in an array

4 Q.2 - Instruction Counts under various Constraints

- (a) The number of Instruction Counts when code without conditional execution was involved is **20**.
- (b) By usage of Conditional Execution in our code, we observed that there was no change in number of instructions than previous but there were instances where it could increase. So we avoid that.
- (c) We observed that when code written with THUMB instructions was implemented, the code execution time dropped which is a sign of an efficient implementation. Upon seeing the reason, we came to know that it happens so due to reduced number of instructions along with less variation in values of flags. The number of instructions in this case was **17**.

Figure 1 further approves towards the fact that with Thumb Instruction, the code execution took **3.833 μ s** whereas without Thumb Instructions, the code execution took **5.250 μ s**.

5 Q.3 - Code Coverage, Code Execution Time, No. of Calls Executed

- (a) The code coverage and its contents were migrated into a file **mainlog** whose contents are as follows:

```
\\project\RTE\Device\LPC1768/startup_LPC17xx.s\Reset_Handler -  
100% (2 of 2 instructions executed)  
\\project\RTE\Device\LPC1768/startup_LPC17xx.s\NMI_Handler -  
0% (0 of 1 instructions executed)  
\\project\RTE\Device\LPC1768/startup_LPC17xx.s\HardFault_Handler -
```


*** error 63: logfile already active

Code Coverage References - [Code Coverage](#), [Logs](#)

- (b) The Code Execution time and related details were taken from the **mainlog** file and they are as follows:

```
Time spent in code without debug info - Time: 217.500 µs (97%)
Functions sorted by execution time consumption:
'__asm_0x100' - Time: 5.250 µs (2%), Calls: 1
'Reset_Handler' - Time: 0.417 µs (0%), Calls: 1
'BusFault_Handler' - Time: 0 µs (0%), Calls: 0
'SVC_Handler' - Time: 0 µs (0%), Calls: 0
'PendSV_Handler' - Time: 0 µs (0%), Calls: 0
'Default_Handler' - Time: 0 µs (0%), Calls: 0
'SysTick_Handler' - Time: 0 µs (0%), Calls: 0
'DebugMon_Handler' - Time: 0 µs (0%), Calls: 0
'UsageFault_Handler' - Time: 0 µs (0%), Calls: 0
'MemManage_Handler' - Time: 0 µs (0%), Calls: 0
'HardFault_Handler' - Time: 0 µs (0%), Calls: 0
'NMI_Handler' - Time: 0 µs (0%), Calls: 0

Execution times by Modules and Functions:
App: 'project', Time: 5.667 µs (3%)
Module: 'RTE/Device/LPC1768/startup_LPC17xx.s', Time: 0.417 µs (0%)
Function: 'Reset_Handler', Time: 0.417 µs (0%), Calls: 1
Function: 'NMI_Handler', Time: 0 µs (0%), Calls: 0
Function: 'HardFault_Handler', Time: 0 µs (0%), Calls: 0
Function: 'MemManage_Handler', Time: 0 µs (0%), Calls: 0
Function: 'BusFault_Handler', Time: 0 µs (0%), Calls: 0
Function: 'UsageFault_Handler', Time: 0 µs (0%), Calls: 0
Function: 'SVC_Handler', Time: 0 µs (0%), Calls: 0
Function: 'DebugMon_Handler', Time: 0 µs (0%), Calls: 0
Function: 'PendSV_Handler', Time: 0 µs (0%), Calls: 0
Function: 'SysTick_Handler', Time: 0 µs (0%), Calls: 0
Function: 'Default_Handler', Time: 0 µs (0%), Calls: 0
Module: 'RTE/Device/LPC1768/startup_LPC17xx.s', Time: 0 µs (0%)
Module: 'main.s', Time: 5.250 µs (2%)
Function: '__asm_0x100', Time: 5.250 µs (2%), Calls: 1
```

Code Execution Time References - [µVision Debugger](#), [Execution Profiler](#)

- (c) The number of calls for our code *main.s* is *1*. The information regarding the same is given in above snippet.