EEE3096S: Practical on Embedded ARM Assembly Programming

1st Bhiwajee Kaushal
BHWKAU001

2nd Ngari Marshall
NGRMAR007

***Abstract*—This report presents the methodology, implementation, and results of an Embedded ARM Assembly Programming practical. The practical involved generating ARM assembly code to program an STM32F0 to perform a binary count using its LEDs. The results discuss the successes and inefficiencies within the code generated while the conclusion offers a summary of the practical and possible areas of improvement.**

I. Introduction

This report details the programming of the STM32F0 in ARM Assembly. The practical task was to implement a binary count using LEDS and various push button inputs. The practical required a detailed report of the design process as well as a physical demonstration of the functioning code.
The report holds details on the design process, the practical's results and discussions, the conclusions drawn as well as possible improvements, and the resources used to complete the practical.
The project files are available on GitHub via the following link:
https://github.com/Bhiwaa/EEE3096S-Group-39.git
(see `Practical 2 → Core → Src → assembly.s`).

A. Aim

The aims of the practical were as follows:

1) The LEDs should automatically count in binary with an increment of 1 every 0.7 seconds.
2) When the `SW0` button is held down, the increment should increase to 2 every 0.7 seconds.
3) When the `SW1` button is held down, the increment timing should change to 0.3 seconds instead of 0.7 seconds.
4) When the `SW2` button is held down, the LED pattern should display `0xAA`. Once released, the counting should resume from the counter value prior to pressing `SW2`.
5) When the `SW3` button is held down, the current LED pattern should freeze. The counting should resume only after `SW3` is released.
6) A detailed report of the design process.

II. Methodology

The following procedure was followed in the programming of the STM32F0 in assembly:
Firstly, the practical task was partitioned into four independent tasks, namely:

1) Lighting LEDs in a binary count format.
2) Implementing a delay on the STM32F0.
3) Checking for push button press.
4) Performing unique actions based on button press.

The first two tasks were practiced on Oaksim [4] to cultivate an understanding of register manipulation and basic assembly code. This practice culminated in implementing a simple `ADD` instruction to increment the LED count. A do-nothing loop was used to introduce a delay on the STM32F0. Following this, the methods `write_leds`, `led_counter_Norm`, `load_delay_slow`, and `delay` were written. The value of the delay was initially set to a random number to allow for testing of other tasks.

Thereafter, research was conducted on the address locations of various registers on the STM32F0, with focus on the Input Data Register for GPIOA [3]. Additionally, research on conditional statements and branching was conducted to aid in the execution of task 4 [1]. Once completed, the method `button_check` was written to check for button presses that directly impact which LEDs come on, while conditional checks were added to `write_leds` to ensure the correct delay is applied.

Furthermore, the methods `write_leds_SW3`, `load_delay_fast`, `led_counter_SW0`, and `write_leds_SW2` were written to perform the corresponding action based on button press. With the overall practical code complete, the delay values were adjusted to meet the set criteria. This involved computing the loop iterations required to achieve the desired delay. The code was progressively benchmarked against the draft solution provided to ensure compliance [5].

Below is the formula used:

$$\text{Loop iterations} = \frac{\text{time}_{\text{desired}} \times f_{\text{clock}}}{\text{cycles per instruction}}$$

where $\text{time}_{\text{desired}}$ is the target delay in seconds, f_{clock} is the microcontroller clock frequency (48 MHz for the STM32F0), and cycles per instruction represents the number of clock cycles the loop consumes. The adjustable value was the number of clock cycles per loop iteration.

This method was, however, very inefficient as multiple iterations of the method were required. Lastly, the code was formatted, the entire program benchmarked against the demo, and comments were added. (Note: All assembly methods mentioned are found in Section VI of this report).

III. Results and Discussions

The final program (VI) met the set requirements, however, its implementation bore some minor discrepancies.
Firstly, both delays in the program did not precisely match the set criteria. This was as a result of using a do-nothing loop to provide a delay. The number of loop iterations was computed via trial and error and comparing with a provided draft solution [5]. This method was inefficient since many iterations were needed and visual confirmation of success is prone to error. The discrepancy was, however, very small hence the delays were considered to be accurate in that regard.
Additionally, the current program fails to account for switch bounce which occurred moderately during testing. As a result, a single button press or release occasionally produced unintended behavior. Due to the random and low frequency nature of this problem, no solution was implemented. A potential solution, checking for a button press then applying a short delay before checking again, was discussed though its addition would impact the normal timing of the system hence why it was not implemented.
Despite these minor discrepancies, the program demonstrated practical reliability and produced practically accurate results. The precision flaw in the delay value did not produce a visual mismatch in the LED display according to the draft solution hence why no further action was applied to improve it. Furthermore, the same can be said for the button bounce flaw which did not have a high enough frequency to impact the general operation of the program. However, to ensure portability of the program as well as enhanced precision, these flaws would need to be resolved at the expense of program complexity.

IV. Conclusion

In conclusion, the practical was executed successfully with the assembly program producing practically accurate results despite minor discrepancies. The benchmarking process against the provided demo showed the current draft of the program was logical with the discrepancies having minimal impact on performance. In terms of findings, knowledge on register manipulation, STM32F0 address values and assembly instructions with emphasis on `.thumb` instruction set was acquired. Lastly to improve the program the following actions can be performed:

1) To mitigate the influence of button bounce, button checks can be performed twice with a small delay, approximately 20ms, between checks. However, this will affect the general timing of the binary count display hence should only be done if timing intervals are not important.
2) To accommodate scalability of the program, the current program can be adapted to use stacks since its current state uses 7 of the 8 allocated registers. This will be at the expense of execution speed but will minimize overwriting of registers.
3) To improve the delay implementation, the program can be adapted to use one of the Timers present on the STM32F0.

V. AI Clause

During the Practical, AI models such as ChatGPT were used to clarify the syntax of ARM Assembly programming, particularly the `.thumb` instruction set. Additionally, they were used to sift the internet for relevant resources on STM32F0 address values and available instructions in `.thumb` mode.

VI. Appendix

The following is the Assembly code used in the programming of the STM32F0:

```
/*
 * assembly.s
 *
 */

 @ DO NOT EDIT
      .syntax unified
   .text
   .global ASM_Main
   .thumb_func

@ DO NOT EDIT
vectors:
      .word 0x20002000
      .word ASM_Main + 1

@ DO NOT EDIT label ASM_Main
ASM_Main:

      @ Some code is given below for you to start
          with
      LDR R0, RCC_BASE   @ Enable clock for GPIOA
          and B by setting bit 17 and 18 in
          RCC_AHBENR
      LDR R1, [R0, #0x14]
      LDR R2, AHBENR_GPIOAB @ AHBENR_GPIOAB is
          defined under LITERALS at the end of the
          code
      ORRS R1, R1, R2
      STR R1, [R0, #0x14]

      LDR R0, GPIOA_BASE  @ Enable pull-up resistors
           for pushbuttons
      MOVS R1, #0b01010101
      STR R1, [R0, #0x0C]
      LDR R1, GPIOB_BASE  @ Set pins connected to
          LEDs to outputs
      LDR R2, MODER_OUTPUT
      STR R2, [R1, #0]
      MOVS R2, #0  @ NOTE: R2 will be dedicated to
          holding the value on the LEDs

@ TODO: Add code, labels and logic for button checks
     and LED patterns

main_loop:
      B button_check @Branch to check for button
          press

button_check:
      LDR R3, [R0, #0x10] @ Load IDR to check for
          SW0 press
      MOVS R4, #1
      ANDS R3, R3, R4
      CMP R3, #0
      BEQ led_counter_SW0
```

```
        LDR R3, [R0, #0x10] @ Load IDR to check for
            SW2 press
        MOVS R4, #4
        ANDS R3, R3, R4
        CMP R3, #0
        BEQ write_leds_SW2

        LDR R3, [R0, #0x10] @ Load IDR to check for
            SW3 press
        MOVS R4, #8
        ANDS R3, R3, R4
        CMP R3, #0
        BEQ write_leds_SW3

        B led_counter_Norm

led_counter_Norm:
   ADDS R2, R2, #1 @ Implement normal led increase
   B write_leds

led_counter_SWO:
        ADDS R2, R2, #2 @ Implement fast-mode led
            increase(+2)
        B write_leds

write_leds_SW2:
        MOVS R5, #0xAA
        STR R5, [R1, #0x14] @ Display 10101010 upon
            pressing SW2
        B main_loop

write_leds_SW3:
        STR R2, [R1, #0x14] @ Hold the display to most
             recent value
        B main_loop

write_leds:
        STR R2, [R1, #0x14] @ Display the current led
            pattern
        LDR R3, [R0, #0x10] @ Load idr data to check
            if SW1 is pressed
        MOVS R4, #2
        ANDS R3, R3, R4
        CMP R3, #0
        BEQ load_delay_fast @ Implement the
            corresponding delay depending on button
            press
        B load_delay_slow

load_delay_fast:
        LDR R5, =SHORT_DELAY_CNT @ Load corresponding
            delay data
        LDR R6, [R5]
        B delay

load_delay_slow:
        LDR R5, =LONG_DELAY_CNT @ Load corresponding
            delay data
        LDR R6, [R5]
        B delay

delay:
        subs R6, R6, #1 @ Implement a do nothing look
            to act as a delay
        BNE delay
        B main_loop

@ LITERALS; DO NOT EDIT
        .align
RCC_BASE:    .word 0x40021000
AHBENR_GPIOAB:   .word 0b1100000000000000000
GPIOA_BASE:   .word 0x48000000
GPIOB_BASE:   .word 0x48000400
MODER_OUTPUT:   .word 0x5555

@ TODO: Add your own values for these delays
LONG_DELAY_CNT: .word 1400000
SHORT_DELAY_CNT: .word 600000
```

References

[1] Azeria Labs, "ARM Conditional Execution and Branching (Part 6)," *Azeria Labs*, [Online]. Available: https:\azeria-labs.com/arm-conditional-execution-and-branching-part-6/. [Accessed: 16-Aug-2025].

[2] Elsevier, "Thumb Instruction Set," *ScienceDirect*, [Online]. Available: https://www.sciencedirect.com/topics/computer-science/thumb-instruction-set. [Accessed: 16-Aug-2025].

[3] STMicroelectronics, *RM0455: STM32H7A3/7B3 and STM32H7B0 Value Line Advanced Arm-based 32-bit MCUs Reference Manual*, STMicroelectronics, 2020. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0455-stm32h7a37b3-and-stm32h7b0-value-line-advanced-armbased-32bit-mcus-stmicroelectronics.pdf. [Accessed: 16-Aug-2025].

[4] Wunkolo, *OakSim: Online ARM Assembler and Emulator*, 2017. [Online]. Available: https://wunkolo.github.io/OakSim/. [Accessed: 24-Aug-2025].

[5] Anthropic, *Claude AI Public Benchmark Demo*, 2025. [Online]. Available: https://claude.ai/public/artifacts/e77499fc-fd54-469c-aa5c-6009b609f283. [Accessed: 25-Aug-2025].