E85: Digital Design and Computer Engineering
Lab 9: Airbag Trigger

Objective

The purpose of this lab is to learn to write performance-optimized code in C and assembly language and to understand the relationship between the two. Specifically, you will design a microcontroller circuit for an airbag trigger that can respond to a simulated impact as quickly as possible, and will compare your optimized assembly and C versions to a nonoptimized C version.

0. Instruction Set

You will do this lab on the NUCLEO board, which contains a Cortex-M0 microprocessor. The Cortex-M0 runs the ARM v6M version of the Thumb instruction set, which differs from the ARM v4 version we have emphasized in class. The instruction set is summarized in Table 3.3 of the Cortex-M0 Technical Reference Manual, available on the class web page. Thumb instructions are packed into 16 bits to better use the limited program memory on a small microcontroller. Most instructions only take two operands, with the first being both the destination and first source. Most instructions only access registers R0-R7. Most instructions only come in the S variant.

1. Airbag Trigger

An airbag trigger should deploy an airbag as fast as possible for each occupant of a vehicle when a collision occurs. For the sake of simplicity, let us model the inputs to the trigger as two digital signals, one indicating that a seat is occupied, and the second indicating that a high-G deceleration event has occurred. In this lab, we will design a microcontroller-based system to monitor the two inputs and assert a trigger output when both inputs are TRUE. Assume that the inputs come on D0 and D1 and the output is connected to D2. Assume that none of the other pins are configured as outputs.

2. Baseline Code

The following baseline code (on the web page) is logically correct but not as efficient as possible. The variables are declared volatile to discourage the compiler from optimizing much.

```
// lab9baseline.c

#include "EasyNucleoIO.h"

void triggerCheck(void) {
      volatile int seat, decel, trigger;

      while (1) {
            seat = digitalRead(0);
            decel = digitalRead(1);
            trigger = seat && decel;
            digitalWrite(2, trigger);
      }
}

int main(void) {
      EasyNucleoIOInit();
      pinMode(0, INPUT);
      pinMode(1, INPUT);
      pinMode(2, OUTPUT);

      triggerCheck();
}
```

Implement this code on your microcontroller. Tie D0 to 1 and apply a pulse on D1 to simulate a sudden deceleration for an occupied seat. Measure the latency from D1 rising to D2 rising using two channels of an oscilloscope. Repeat your experiment 10 times and find the average, maximum, and standard deviation.

2. Interpreting Assembly Language

Look at the assembly language code that the Keil compiler produces for the baseline `triggerCheck()` code, as well as the `digitalRead` and `digitalWrite` functions it calls. Study it until you understand how each line relates to the C code.

What is the largest number of instructions that might occur from the time that D1 rises until D2 rises (while the program is in the `triggerCheck` loop)?

3. Tutorial: Mixing C and Assembly Language

It is not hard to mix C and assembly language programs. For example, the following `flash.c` and `led.s` files are available on the class web page. The C code contains a prototype for the `led` function, and the assembly language code implements it. The argument a is passed in R0.

```
// flash.c
#include "EasyNucleoIO.h"

// prototype for assembly language function
void led(int a);

int main(void) {
      EasyNucleoIOInit();
      pinMode(13, OUTPUT);
      while(1) {
            led(0);
            delayLoop(200);
            led(1);
            delayLoop(200);
      }
}
```

```
; led.s
; turn LED on D13 / PB3 on or off
      AREA |.text|, CODE, READONLY     ; define this file as code
      EXPORT led              ; declare LED to be called externally

GPIOB_ODR EQU 0x48000414          ; define constant address of port

led
      PUSH {R4}               ; save R4 on stack before changing it
      LDR R4, =GPIOB_ODR          ; put point to GPIOB ODR in R4
      LDR R1, [R4]                ; R1 = GPIO_ODR
      MOVS R3, #0x08          ; 1 in 3rd bit (LED is on PB3)
      CMP R0, #0              ; check if we should turn off LED
      BEQ ledoff                  ; yes: skip to ledoff
ledon
      ORRS R1, R1, R3         ; otherwise set PB3 to 1
      B finish                ; and skip
ledoff
      MVNS R3, R3                 ; clear PB3 to 0
      ANDS R1, R1, R3
finish
      STR R1, [R4]                ; store R1 back in PORTB_ODR
      POP {R4}                ; restore R4 from stack
      BX LR                   ; equivalent to MOV PC, LR, preferred now

      ALIGN                       ; make sure code ends on word boundary
      END                     ; bye bye.  Have a nice day.
```

Create a new project and add both files. Compile it and run it on the Nucleo board and verify that the LED flashes. Single-step through the assembly language code and watch how it works.

4. Assembly Language Implementation

Write your own hand-optimized airbag `triggerCheck` in assembly language. Comment out the baseline `triggerCheck()` function and call your assembly language function instead.

A suggested approach is to adapt `led.s`. Determine which bits of which port are associated with D0, D1, and D2. Figure out the addresses of the registers controlling these bits. Write a simple program that reads D0 and D1 and toggles D2, and use the debugger and some wires and oscilloscope to check that it works. Then optimize your code.

Repeat your count of the largest number of instructions that might occur from D1 to D2 and your physical measurements of average and standard deviation in latency. How much improvement did you achieve?

5. Optimized C Implementation

Rewrite the baseline `triggerCheck()` function as efficiently as you can in C. Look at the assembly language output by the compiler, and optimize until you are satisfied.

Repeat your count of the largest number of instructions that might occur from D1 to D2 and your physical measurements of average and standard deviation in latency. How do your results compare with the baseline and with your assembly language code?

What to Turn In

1. Please indicate how many hours you spent on this lab. This will be helpful for calibrating the workload for next time the course is taught.
2. Your assembly language implementation.
3. Your optimized C implementation.
4. A table of instruction count and average, max, and standard deviation of latency for each of the three implementations.

If you have suggestions for further improvements of this lab, you're welcome to include them at the end of your lab.