E85: Digital Electronics 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 non-optimized C version.

0. FE310 Board

You will do this lab on the RED-V ThingPlus board, which contains a Freedom E310 (FE310) system-on-a-chip (SoC). The FE310 features SiFive's E31 CPU Coreplex, a high-performance, 32-bit core. The FE310 supports RV32IMAC core.

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, you 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 pins 0 and 1 and the output is connected to pin 2. 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 "EasyREDVIO_ThingPlus.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) {
    pinMode(0, INPUT);
```

```
    pinMode(1, INPUT);
    pinMode(2, OUTPUT);

    triggerCheck();
}
```

- Implement this code on your microcontroller.
- Tie pin 0 to 1 and apply a pulse on pin 1 to simulate a sudden deceleration for an occupied seat.
- Measure the latency from pin 1 rising to pin 2 rising using two channels of an oscilloscope.
- Repeat your experiment 10 times and find the average, maximum, and standard deviation.

2. Interpreting Assembly Language

- Start the debugger and look at the assembly language code that the compiler produces for the baseline `triggerCheck()` code, as well as the `digitalRead` and `digitalWrite` functions it calls. If the Disassembly pane is not open by default, select View->Disassembly or hit Ctrl-F12. Study it until you understand how each line relates to the C code. You can step through the assembly by selecting "Step Over" or hitting F10 while focusing the Disassembly pane.

What is the largest number of instructions that might occur from the time that pin 1 rises until pin 2 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 (remove the .txt file from led.S.txt after downloading). The C code contains a prototype for the `led` function, and the assembly language code implements it. The argument a is passed in a0.

```
// flash.c

#include "EasyREDVIO_ThingPlus.h"

#define DELAY_MS 500

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

int main(void) {
    pinMode(5, OUTPUT);

    while(1) {
        led(0);
        delayLoop(DELAY_MS);
        led(1);
        delayLoop(DELAY_MS);
    }
```

```
}
```

```
// led.S
// turn LED on GPIO 5 on or off

.section .text                  // define this file as code
.align 2                        // make sure code aligns on word boundaries
.globl led                      // declare LED to be called externally

.equ GPIO_OUTPUT_VAL, 0x1001200C

// Our led output value passed into a0

led:
    addi sp, sp, -16            // Setup our stack frame
    sw ra, 12(sp)               // Save return address

    li t1, GPIO_OUTPUT_VAL      // Put address of GPIO0 output_val register in t1
    lw t2, 0(t1)                // Store current state of output_val register in  t2
    li t3, 0x20                 // Put a 1 in the 6th bit corresponding to GPIO 5
    beqz a0, ledoff
ledon:
    or t2, t2, t3
    j finish
ledoff:
    not t3, t3
    and t2, t2, t3
finish:
    sw t2, 0(t1)
    lw ra, 12(sp)               // Restore the return address
    addi sp, sp, 16             // Deallocate stack frame
    ret
```

- Create a new project and add both files. (Note that you will have to choose All Files or ASM Source File in the file type pulldown to select led.S when adding the file.) Compile it and run it on the RED-V board and verify that the LED flashes. Single-step through the assembly language code and watch how it works.

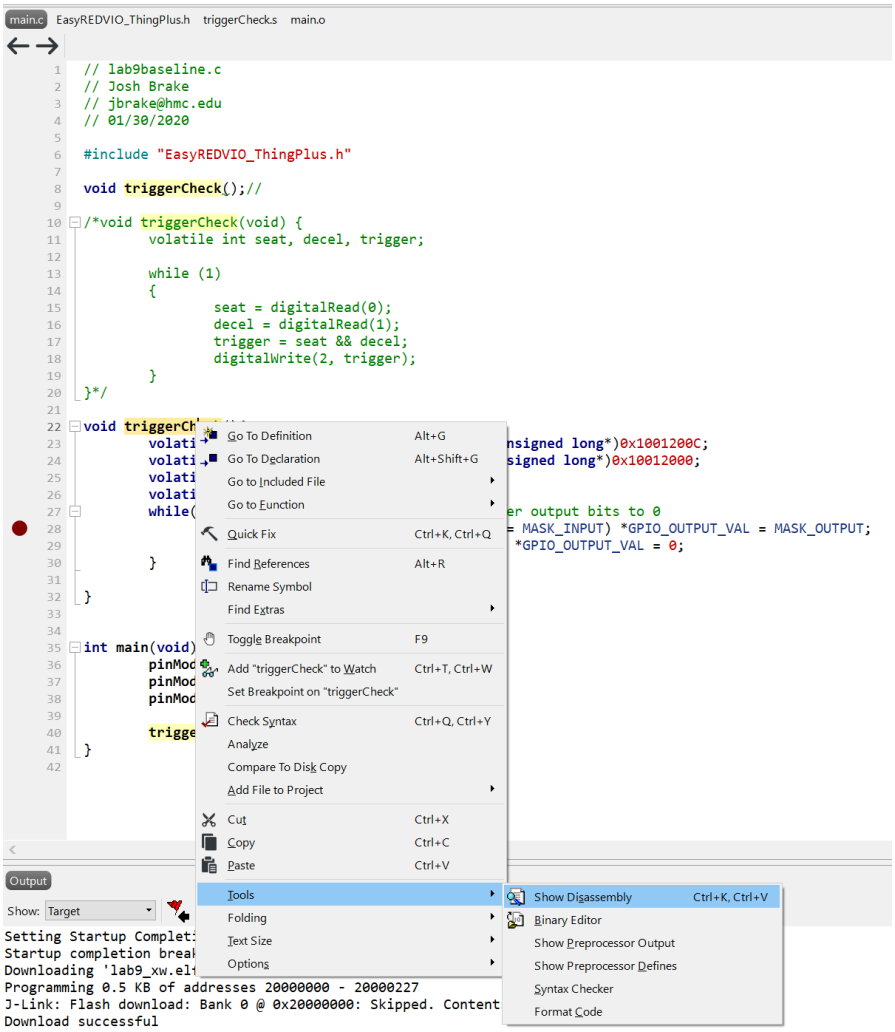
4. Assembly Language Implementation

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`. Optimize your code to minimize the latency.

Repeat your count of the largest number of instructions that might occur from pin 1 to pin 2 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. You can view the generated assembly code using the disassembly features. There are two ways to view the disassembled code. While debugging, right-click on the function you want to view in assembly and select "Go to Disassembly" from the context menu. You can also view the assembly of a function in a separate window by right clicking the name of a function while not debugging, navigating to tools in the context menu and selecting "Show Disassembly".



Repeat your count of the largest number of instructions that might occur from pin 1 to pin 2 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.

Please indicate any bugs you found in this lab manual, or any suggestions you would have to improve the lab.