Lab 10: Airbag Trigger

Digital Design and Computer Architecture: RISC-V Edition (Harris & Harris, Elsevier © 2021)

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. Instruction Set

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. The instruction set is summarized in Appendix B of the textbook and in The RISC-V Instruction Set Manual in the chapter titled "RISC-V Assembly Programmer's Handbook" (chapter 25; pages 137-140 in version 20191213, the most recent version as of January 2020).

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 "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 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

- Start the debugger and look at the assembly language code that the PlatformIO 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 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`. Determine which bits of which port are associated with D0, D1, and D2.

A challenge here is to figure out the addresses of the registers controlling these ports. D0, D1, and D2 are the first 3 GPIO pins in the FE310 GPIO complex (see FE310-G002 Manual chapter 17). GPIO0 instance starts at memory address 0x10012000. To write to a pin as an output, we first have to ensure that it is configured as a general IO (iof_en bit set to 0) and configured as an output (output_en bit set to 1). The output_en register begins at an offset of 0x08 from the base address of GPIO0, so to configure as an output we should write the bit corresponding to D2 high at memory address 0x10012008 as shown below.

Pin #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

In binary, this value is 0000 0000 0000 0000 0000 0000 0000 0100; in hexadecimal, it is 0x00000004.

To ensure that the pin is controlled as a GPIO, we also have to make sure it is set to be controlled by GPIO instead of one of the hardware peripherals. Therefore, we need to write 0x00000004 to memory address 0x10012038 which is the location of the iof_en register (0 means controlled by GPIO, 1 means controlled by H/W peripheral). Finally, after these two registers are configured to set the pin up as a GPIO-controlled output, we can write a value to the pin in the output_val register (offset 0x0C from GPIO0 base of 0x10012000). For example, if we wanted to write D2 high, we would write 0x00000004 to memory address 0x1001200C.

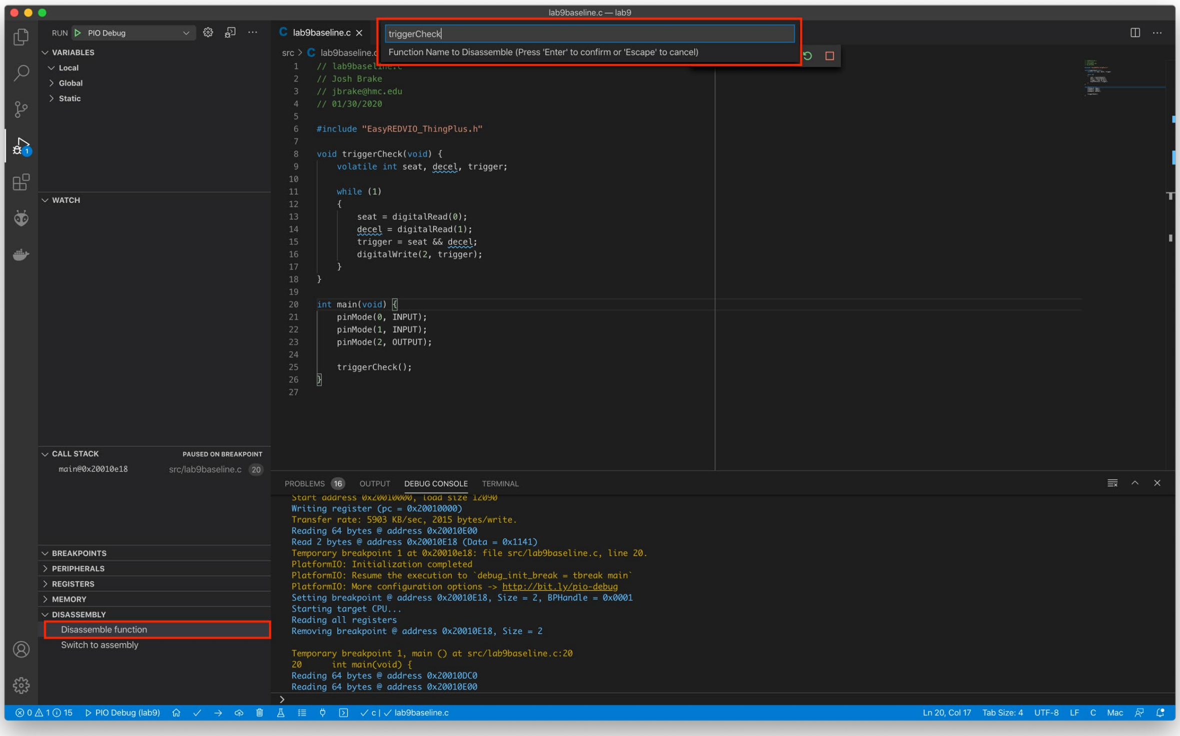
In the same way, if we want to read D0 and D1, we can simply read the value in the input_val register at address 0x1001 xxxx (find xxxx yourself, see the FE310-G002 manual GPIO chapter).

Write an assembly program that reads D0 and D1 and writes to D2. Use the debugger and the oscilloscope to check how your code 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. You can view the generated assembly code using the disassembly features in PlatformIO. When you are debugging, navigate to the disassembly tab in the bottom of the debugging panel and click "Disassemble function". In the text box that opens at the top of the window, type the name of the function you wish to disassemble (in this case "triggerCheck"). This will open up a listing of the assembly instructions generated by the compiler.



```c
// lab9baseline.c
// Josh Brake
// jbrake@hmc.edu
// 01/30/2020

#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();
}
```

```
Start address 0x20010000, load size 12090
Writing register (pc = 0x20010000)
Transfer rate: 5903 KB/sec, 2015 bytes/write.
Reading 64 bytes @ address 0x20010E00
Read 2 bytes @ address 0x20010E18 (Data = 0x1141)
Temporary breakpoint 1 at 0x20010e18: file src/lab9baseline.c, line 20.
PlatformIO: Initialization completed
PlatformIO: Resume the execution to `debug_init_break = tbreak main`
PlatformIO: More configuration options -> http://bit.ly/pio-debug
Setting breakpoint @ address 0x20010E18, Size = 2, BPHandle = 0x0001
Starting target CPU...
Reading all registers
Removing breakpoint @ address 0x20010E18, Size = 2

Temporary breakpoint 1, main () at src/lab9baseline.c:20
20      int main(void) {
Reading 64 bytes @ address 0x20010DC0
Reading 64 bytes @ address 0x20010E00
```

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.

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