# EENG260 (Microcontrollers) Lab 2:

Control an LED with a switch on the Tiva C evaluation board using ARM Assembly instructions

## Tasks for this lab

- Build a program using assembly instructions that turns on one of the LEDs on your Tiva C board.
- Then, modify your program to use switch SW1 on your Tiva C board to control that LED (SW1 pressed makes the LED turn on, SW1 released makes the LED turn off).

## Reference Materials

- ARM Assembly spnu118u.pdf
- CortexM_InstructionSet.pdf
- Tiva TM4C123GH6PM Microcontroller spms376e.pdf
- Tiva C Series TM4C123G LaunchPad spmu296.pdf
- Partial main.asm file

## Part 1 – Light an LED

First, start a new project for your lab in CCS. You *could* follow all the steps from the previous labs to get to the point of having a working project targeted at your board with a simple main.asm file imported, but I would recommend just doing a copy/paste on your previous project folder in the Project Explorer pane, renaming your new project accordingly, then deleting the bits you added to the main.asm file last time.

Now, we're ready to start coding, once we've worked out just what we need to do. For this lab, we want to turn on one of the LEDs on the Tiva C board. If you look in the "Tiva C Series TM4C123G LaunchPad" document, the schematics are included, starting on page 20. All three LEDs are shown on that page, with their associated drive circuitry and control lines (LED_R, LED_G, and LED_B for red, green, and blue). Further up the page, those lines come back to the microcontroller, to pins labeled PF1, PF2, and PF3. Those are abbreviations for General Purpose I/O (GPIO) Port F pins 1, 2, and 3. This is a parallel port on the microcontroller, so all we need to do is make one of those pins go high, to turn on the drive transistor for the LED.

On some microcontrollers, that might be just as simple as storing a value at a particular memory location, but, due to the range of features this microcontroller supports, we have to do some configuration work first. From here on out, you will be referencing the "Tiva TM4C123GH6PM Microcontroller" document.

Since we want to use GPIO Port F, we need to look up what we need to do to make that happen. According to the Contents, section 10 covers GPIO, and, in particular, section 10.3 (starting on page 656) covers initialization and configuration, including step-by-step instructions that cover every possible thing you could want to do with GPIO ports – which, for us, is overkill at this point. Before the list, there's mention of having the option of using the Advanced Peripheral Bus (APB) or the Advanced High-Speed

Bus (AHB). Normally, you might think "use whichever one sounds best", but, in this case, you definitely want to use APB, since that's what the CCS debugger uses to view register contents.

Let's go over the list and pick out the useful bits:

1. Enable the clock to the port by setting bit(s) in RCGCGPIO, one of the System Control registers (section 5). Yes, we need to do that – no clock would mean we can't use the port at all.
2. Set the direction of the port pins by setting bits in GPIODIR, one of the GPIO registers. Yes, we're trying to use one of the port pins as an output, so we'll need to do that.
3. Configure registers for alternate pin functions. By default, the GPIO pins are configured as "regular" pins, rather than any alternate function, so we can ignore this one.
4. Set the drive strength for the pins. The pins default to a 2mA drive strength, which is plenty to drive the circuit included on the Tiva C board, so we don't need to worry about this one either.
5. Program pull-up, pull-down, etc. We're just driving an LED from the pin, so we can skip this.
6. Configure pins as digital IO by setting bit(s) in GPIODEN, another GPIO register. We do want a digital output to drive our LED, so we will need to do that.
7. Program registers for interrupts. We're not doing anything with interrupts (that will be a later lab), so we can ignore this for now.
8. Optionally, lock pin configurations using GPIOLOCK. Not an option we need, so we'll skip it.

So, to prepare GPIO Port F for use, we need to enable it by giving it a clock (RCGCGPIO), set the port direction on the pin we've chosen to be an output (GPIODIR), then set that pin as digital I/O (GPIODEN). Once we've done that, we should just need to store a 1 in the appropriate bit of GPIODATA (the data register for the port), and the light should turn on. With those register names in hand, you can look up the details of which bits control which options in their entries in the Register Description subsections of the System Control and GPIO sections of the document.

Of course, there's one more twist. When working with parallel ports, as a rule, you want to be certain to only change the bits you intend to change (in larger projects, you may not even be the only one programming your system). Often, this is done by reading the contents of a register, modifying the bits you want to change, then writing the altered bytes back to the register. On this microcontroller, an alternate method is used, called "register offset addressing", where an offset to the base register address is used to specify which bit(s) you want to alter. Details are listed in section 10.2.1.2, on page 654.

Now, you just need to make your additions to the main.asm file for constants and assembly instructions to make those register alterations happen. All those registers are mapped to the memory space of the microcontroller, so you can use LDR or MOV to get the values you need stored in the core registers (R0, R1, etc.), then use the contents of those registers to STR new values to the System Control and GPIO registers. Since you are dealing with known, fixed memory locations for those registers, and the documentation is written showing all the register locations as offsets to a base address, you can let the microcontroller handle the math for you. For example, to enable GPIO Port F, you will need to write a value to the System Control register RCGCGPIO. If you load R0 with the base address for the System Control registers (maybe using a LDR of a value you set up as a constant with a .field assembler directive), R1 with the value you want to store, and have a constant set up for the offset to that register using a .equ assembler directive (let's use the label oRCGCGPIO for this example), then you could use the instruction " STR R1, [R0,#oRCGCGPIO]" to store the contents of R1 in the RCGCGPIO register.

Build, debug, and verify that the LED you selected turns on, and you will be done with part 1.

## Part 2 – Control the LED with a switch

Now that you can turn the LED on, the next thing to do is to control the LED with one of the onboard switches, specifically SW1 (located in the lower-left of the Tiva C board). SW2 is equally workable, but, due to one of the alternate features on that pin, there is extra configuration work that has to be done to use that pin as a digital input, so we will stick with SW1 for now.

Go back to the board schematic, and find SW1, then trace it back to the microcontroller. You should end up back at another GPIO Port F pin, which means some of the work you need to do to configure the pin for input is already done. Note that one side of the switch ties directly to ground, while the other ties directly to the port F pin. To cause a change in voltage at the pin when the switch is pressed, there will need to be a pull-up resistor involved – and, since it's not on the schematic, you will need to configure the microcontroller to include that pull-up resistor on that input pin.

Review the list of configuration steps in section 10.3 of the microcontroller manual, to find the register you need to alter to give that pin a pull-up resistor. While you're at it, are there any other registers that need to be altered to use that pin as an input?

Once you have identified the registers you need to alter, give it a try. Properly configured, when you read the GPIO Port F GPIODATA register, that pin should read 0 when the switch is pressed, and 1 when the switch is released. Of course, that "register offset addressing" from section 10.2.1.2 that you had to do to make the LED bit writeable is also necessary to properly read the bit for that switch as well.

Finally, now that you have the switch and LED working, you have to write some code to make the switch control the LED. That would go inside the "loop:" area, since you want to constantly "poll" the switch, then make changes to the LED as appropriate. How you do that is up to you… but, since we're not using the data from that switch to do anything other than control that LED, the simplest way would be to read the switch state from port F into an internal register, use a couple of assembly instructions to invert and move the data bit from the switch to the proper position for controlling the LED, then write that data back to port F to control the LED.

## To turn in

Make a .zip file with the following contents and upload it to Canvas:

- A brief lab report, in Word or PDF format. What issues did you run into? What did you learn? Keep it under a page, if you can.
- A copy of your main.asm file. You can export your file to the file system by right-clicking on it in the Project Explorer pane and selecting Export.