EENG260 (Microcontrollers) Lab 3:

Build a 4-bit binary up/down counter with external LEDs controlled by a Tiva C evaluation board using ARM Assembly instructions

Tasks for this lab

- Connect the provided LED hardware to GPIO Port D pins 0-3 and ground, by inserting wires into the appropriate sockets on the bottom of your Tiva C board.
- Create a program in assembly to produce a visible 4-bit count on the provided LEDs that meets the following requirements:
    - Use a core register to store a count value, set to 0 at start-up.
    - Use the LEDs to show the low nybble (4 bits) of your count.
        - Since we are only using the count to produce a display, there is no need to worry about bounds-checking your count (so, having 16 display as 0 is allowed).
        - These LEDs draw more current when lit than the on-board LEDs (3mA), so you will need to configure for that using Port D's GPIODR4R register.
    - Clicking SW1 will increase the count by 1.
    - Clicking SW2 will decrease the count by 1. As mentioned in the previous lab, additional steps are required to properly configure the pin this switch connects to. Specifically, you will need use the port's GPIOLOCK and GPIOCR registers to enable changes to that pin, before any changes you try to make in some other registers (including GPIOPUR and GPIODEN) will "take".
- Add a function to your program to provide a switch debounce delay using the SysTick timer.
    - BL and BX are the instructions you need to make and return from a function call.
    - The SysTick registers you will need to use are STCTRL, STCURRENT, and STRELOAD.
    - A 10ms delay should be produced when a switch changes state, which will allow the physical switch hardware to stabilize.
    - Since you are targeting a specific time delay, you should use the most precise system clock you have available, the crystal-driven Main Oscillator (MOSC), which you can enable and select using the RCC register.

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
- LEDs and Pots.pdf

Part 1 – Build and test your counter

Start a new assembly project targeted to your hardware, by the method of your choice.

For this project, rather than using the LEDs on your Tiva C evaluation board, you will be using the provided "LEDs and Potentiometers" board to display the four bits of output for your counter. A link to the schematic is included above in the Reference Materials section. The potentiometers on the board can be safely ignored for this lab.

First, you need to connect your hardware. Technically, almost any four GPIO port pins would work, but there are some limitations. Mainly, those pins should not otherwise be in use, which eliminates port F (which has the onboard switches and LEDs hard-wired in) and the lower bits of port C (which is how the CCS programmer and debugger talks to your board). Also, the pins need to be available to connect to at the sockets the board provides. Given those limits, I decided we should use Port D pins 0-3 for this lab, which are all located next to each other on one of the sockets. Connect the LED circuits by plugging the leads into the sockets on the bottom of your evaluation board, with circuit ground to one of the GND jacks and the four LED lines to Port D pins 0-3.

With your hardware set up, and your requirements in the second bullet of the "Tasks for this lab" section, you are ready to build your program. To meet those requirements, your main program loop will need to do something similar to:

- Check the state of the switches
- If any switches have changed since the last time through the loop:
    - Determine which switch changed
    - If that switch was just pressed (that bit changed from 1 to 0), increase/decrease count
    - Set LEDs to match the current count

Code up your program, build and test… and it will *almost* work, except that your switch presses will rarely, if ever, make the count go up or down by exactly one. You may have run into this sort of problem in your digital circuits classes, called "switch bounce", where physical vibration of the switch contacts is detected by an input as multiple changes to the state of a switch. There are different ways to handle this issue, one popular choice being a "software debounce", where you basically wait out the vibrations after you detect an initial change.

Part 2 – Adding a debounce delay

If we're going to wait out switch vibrations, we need to know how long to wait. Just as a rule of thumb (good enough for this lab at least), a 10ms delay should work. On some microcontrollers, you might have to rig some combination of do-nothing assembly instructions to waste a chunk of time, then run that code a certain number of times to get the right amount of delay. Instead, our microcontroller has a number of timers built in, including a fairly simple timer called SysTick (registers are described in section 3.3 of the microcontroller manual).

With SysTick, you get a simple count-down timer that decrements by 1 every clock cycle. When the count reaches 0, it sets a flag bit in the STCTRL register. Once you have configured the counter, you can just keep checking that bit until it changes to a 1, at which point your wait is over. So, if we want a 10ms wait, what do we load the timer with for a start value? That depends on the clock source we provide to the SysTick timer.

Our microcontroller includes a "Precision Internal Oscillator (PIOSC)" that serves as the clock source for the microcontroller's system clock, as well as various peripherals, at startup. It operates at 16MHz +/-

1%, and is perfectly suitable for many purposes. By default, SysTick gets that clock signal divided by 4, so 4MHz. But, we can change that clock to feed directly from the system clock in STCTRL, which gives us more options to work with.

In particular, our evaluation board comes equipped with a crystal to provide a precision frequency reference for the microcontroller's "Main Oscillator (MOSC)". This also provides a 16MHz clock frequency, but with a 20 parts-per-million (ppm) variance (as compared to the 10,000ppm PIOSC). Since we have the hardware, we can put it to use, for more accurate wait-times.

There are many more clock options available (section 5.2.5 of the microcontroller manual has full details) that we could employ, depending on how much computing power we need vs. how much electricity we're willing to use, but, for this lab, just driving the system clock directly from the MOSC will serve our needs. You will need to change a couple of bits in the System Control register RCC to make that happen, but that's all (although you will definitely want to change only those bits, so loading the register contents, making changes, then storing the changed value back to RCC is the way to go).

So, a 10ms delay using a counter fed by a 16MHz clock, a little math says we need to wait 160,000 cycles to reach our goal (0x27100 if you prefer hex). We know what we need to do, so we could just add code to our existing program in the right places and make it work. However, if your code looks like mine, you likely have a fair amount of code in place already, making it hard to read what's going on with it. Meanwhile, the thing you want to add (a 10ms delay after any button is pushed) is a well-defined function. Let's make that an actual function, so we can just insert a function call at the appropriate point in our code.

To make a function, all you need is a label to jump to (like main: is used in our main program, you could use something like wait: instead), followed by the code for your function, ending with a "BX LR" to return back to the next instruction after your function call. To call your wait: function would just take a "BL wait" at the appropriate point in your code. Throw in a couple of assembler directives, and you might end up with something like:

```
.thumb
.global main
; your constants
main:       .asmfunc
; your initialization code
loop:       ; your code for checking switch states
            BL wait ; function call to wait: after switch change
            ; your code for handling switch state changes
            B loop
.endasmfunc
.align
; constants specific to wait:
wait:       .asmfunc
            ; SysTick timer setup code
waitLoop:   ; constantly check if the timer has finished
            BEQ waitLoop
            BX LR
.endasmfunc
.end
```

Oh, that new directive, .align? That's a preventive measure. Different assembly instructions and constants take up different numbers of bytes in ROM, and you can end up with a mismatch between where a label points and what the microcontroller will accept as a legitimate location for fetching an instruction (each byte is addressable, but only every other byte is fair game for the location of an assembly instruction – it's a quirk of the microcontroller). .align will ensure that your addresses are properly aligned at that point.

One last note on function calls – while you're in a function, you are still using the same registers in the hardware, so you want to be sure not to accidentally change something you're relying on in your main program while you're inside the function! There are different ways to accomplish this, some better than others. For the simple program we are building here, and with the number of internal registers available, I would suggest just using registers you don't touch in main: while you are within your wait function.

Now that we've got the details of what we need to accomplish nailed down, here's the list of what you need to make happen:

- Add code to your main program to enable MOSC and use it for your system clock (System Control register RCC)
- Add a wait function to your main.asm file, that will return after 10ms have elapsed
    - Disable SysTick counter and configure to use the system clock (STCTRL)
    - Prepare the SysTick counter for its countdown (STRELOAD, STCURRENT)
    - Enable the SysTick counter (STCTRL)
    - Wait for the countdown to finish (STCTRL)
- Call your wait function after a switch change has been detected

With those additions, now, when you press either switch on your development board, your count should change by one each time.

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.