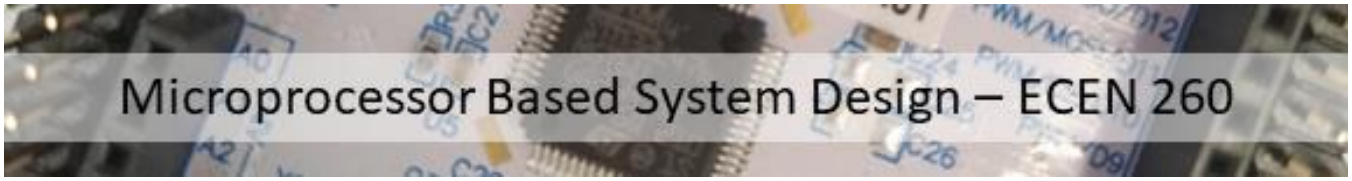


ECEN 260 Lab 04 – Instructions
Switch-Controlled LEDs with Assembly

Lab Objectives

- Use GPIO control registers to read and write to ports connected to switches and LEDs.
- Wire a circuit on a breadboard from a schematic.
- Continue practicing how to use assembly language.
- Write a separate lab report.

Required Parts

- 1 Nucleo-L476RG & cable
- 1 breadboard (protoboard)
- 1 switch
- 1 LED (any color)
- 1 resistor
- 4 male-to-male wires

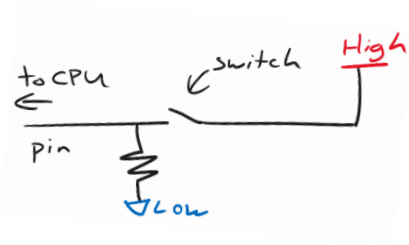
Lab Background

For this lab (lab partner optional), you will be writing and testing ARM assembly code on your Nucleo board to control LEDs based on inputs from button switches. You should have already completed this week's Mini Lab.

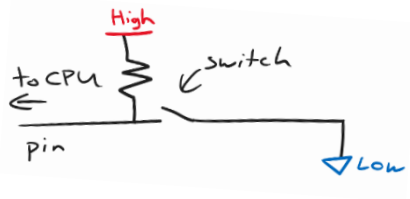
Active-High Switches versus Active-Low Switches

If you need a review on pull resistors, continue reading here. If you feel confident that you understand pull resistors, skip to the next page.

Active-high Switch with Pull-down Resistor: If a switch is wired to a HIGH voltage, then the switch will connect the microcontroller pin to that HIGH voltage when the button is pressed. To detect when the button is *not* pressed, the microcontroller pin is connected to the other voltage level (LOW). To avoid directly connecting HIGH to LOW when the button is pressed, a resistor is placed in between the microcontroller pin and the LOW voltage. So, an *active-high* switch requires a *pull-down* resistor. This configuration sends a HIGH when the button is pressed and a LOW when the button is not pressed.



Active-low Switch with Pull-up Resistor: If a switch is wired to a LOW voltage, then the switch will connect the microcontroller pin to that LOW voltage when the button is pressed. To detect when this type of button is *not* pressed, the microcontroller pin is connected to the other voltage level (HIGH). A resistor is placed in between the microcontroller pin and the HIGH voltage. So, an *active-low* switch requires a *pull-up* resistor. This configuration sends a LOW when the button is pressed and a HIGH when the button is not pressed.



Why Use Active-low Switches? While less intuitive, active low switches are common because two devices that operate on different voltages can communicate with just a ground signal. Another reason is because it reduces the chance of a wire that touches your board accidentally connecting a HIGH and LOW voltage together.

Control Registers:

The I/O control registers – also known as Special Function Registers (SFRs) – are used to configure and communicate with the I/O pins. Each GPIO port has 16 pins, 0 through 15. For example, PC13 is pin 13 of Port C. Some of the control registers have only two options per pin and so they use only 1 bit for each pin and are thus 16-bit registers. Other control registers have four options, requiring 2-bits for each pin, and are 32-bit registers.

The Mode Register (MODER): Each port has a **32-bit** Mode Register – two bits for each pin, with these options:

- 00 = input mode
- 01 = output mode
- 10 = alternate function mode (don't worry about this for now)
- 11 = analog mode (don't worry about this for now)

For each port, the MODER starts at the base address for that port.

Register	bit	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
GPIOx_MODER		MODE15		MODE14		MODE13		MODE12		MODE11		MODE10		MODE9		MODE8		MODE7		MODE6		MODE5		MODE4		MODE3		MODE2		MODE1		MODE0	

The Pull-Up/Pull-Down Register (PUPDR): Each port has a **32-bit** Pull-Up/Pull-Down Register – two bits for each pin, with these options:

- 00 = no pull resistor
- 01 = pull up resistor enabled
- 10 = pull down resistor enabled
- 11 = reserved (don't worry about this for now)

For each port, the PUPDR starts at the base address + 0xC for that port.

Register	bit	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
GPIOx PUPDR		PUPD15		PUPD14		PUPD13		PUPD12		PUPD11		PUPD10		PUPD9		PUPD8		PUPD7		PUPD6		PUPD5		PUPD4		PUPD3		PUPD2		PUPD1		PUPD0	

The Input Data Register (IDR): Each port has a **16-bit** Input Data Register – one bit for each pin:

- 0 = the microcontroller has read a LOW voltage on the pin
- 1 = the microcontroller has read a HIGH voltage on the pin

For each port, the IDR starts at the base address +0x10 for that port.

Register	bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
GPIOx_IDR (where x = A..I)		ID15	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0

The Output Data Register (ODR): Each port has a **16-bit** Output Data Register – one bit for each pin:

- 0 = the microcontroller will write a LOW voltage onto the pin
- 1 = the microcontroller will write a HIGH voltage onto the pin

For each port, the ODR starts at the base address +0x14 for that port.

Register	bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
GPIOx_ODR (where x = A..I)		OD15	OD14	OD13	OD12	OD11	OD10	OD9	OD8	OD7	OD6	OD5	OD4	OD3	OD2	OD1	OD0

Wiring the Circuit

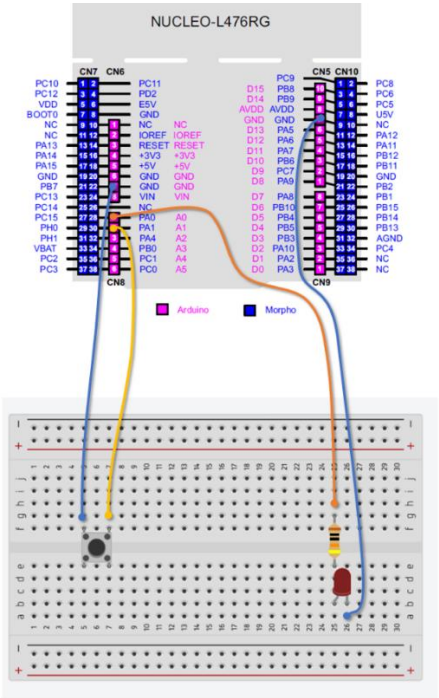
On-board Devices (what is already on your board): Your Nucleo board has a blue user button (Button 1) already wired in an active-low configuration to pin PC13 and connected to an existing on-board pull-up resistor. Your Nucleo board also has a green LED (LED 2) already wired to pin PA5 with an on-board current-limiting resistor.

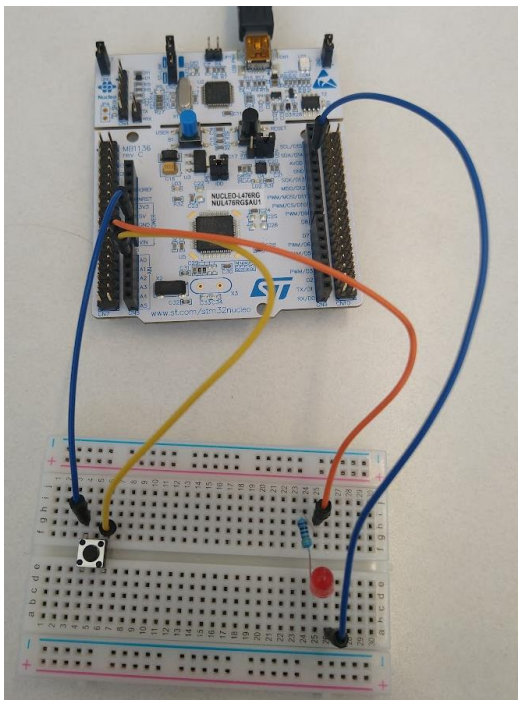
Off-board Devices (what you will connect to your board): In addition to these two on-board devices, you will connect another button and another LED. The additional button will be called "Button 3" because Button 2 is the name for the reset button. The additional LED will be called "LED 4" because LED 1 is the name for the communication LED, and LED 3 is the name for the power LED. (See the wiring diagram below.)

Adding Button 3: You will connect Button 3 to pin PA1 in an active-low configuration. That means it will need a pull-up resistor. However, rather than connecting a pull resistor, enable the *internal* pull-up resistor for that pin using the PUPDR. When you are putting the button into the breadboard, place it across the gap in the middle of the board oriented like the button on the left in the figure below. *If you try to place the button sideways like the button on the right, it will not fit.* When oriented correctly, the two left pins are already connected to each other, and the two right pins are already connected to each other. When the button is not pushed, the left and right pins are disconnected. When the button is pushed, the left and right pins are connected. For the Button 3, connect one side to ground and the other side to PA1.

Adding LED 4: You may choose any color LED for LED 4. Each LED generally requires a resistor to limit the current and prevent the LED from burning out. You will connect LED 4 to pin PA0 with a current-limiting resistor. The 3.3V voltage we are supplying is close enough to the forward voltage of the LED that any resistor value from 50 to 500 ohms is good enough. Your kit comes with 100-ohm resistors. Use one of those. For resistors, the direction does not matter. For LEDs, the direction *does* matter. *The long pin of the LED is the positive side, and the short pin is the negative side.* The negative side of the LED should be the side connected to ground. The long side of the LED should connect to the resistor, and the other end of the resistor should connect to PA0.

Wiring Diagram and Photo





Starting Code

This is some starting code. Work to understand it. It turns on LED2 if Button1 is pushed and off if it is not pushed.

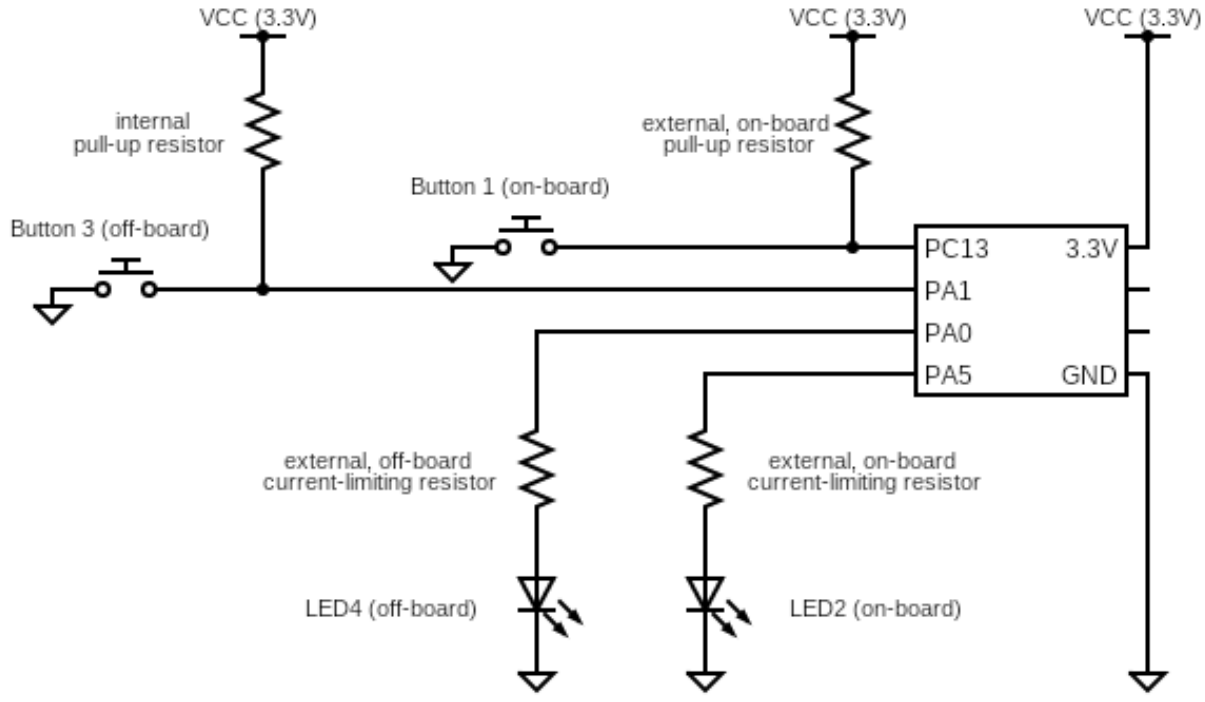
```
    .text
    .global main
    .cpu cortex-m4
    .syntax unified


// This starting code shows how to configure B1 and LED2
// and have LED2 turn on whenever B1 is pushed.
main:

    // Base Address of Port A (PA) = 0x48000000
    MOV  R4, #0x0000
    MOVT R4, #0x4800

    // Base Address of Port C (PC) = 0x48000800
    MOV  R5, #0x0800
    MOVT R5, #0x4800

    // Address of Reset/Clock Control Advanced High-Performance Bus 2 Enable Register
    // RCC AHB2 ENR = 0x4002104C
    MOV  R6, #0x104C
    MOVT R6, #0x4002

    // Enable the I/O Port clocks with RCC AHB2 ENR (R6)
    MOV  R0, #0b00000101 // bit0 for PA, bit2 for PC
    STR  R0, [R6]


    // ***** Configure LED2 (PA5) as output *****
    // Create bitmask to reset bit 11 of PA MODER (bit 10 already initializes to 1)
    MOV  R1, #0x0800  // 0000 1000 0000 0000
    MVN  R1, R1       // flip the bits, mask = 1111 0111 1111 1111
    // Read PA MODER value, reset bit 11, and write back to PA MODER
    LDR  R0, [R4]     // load PA MODER (PA base + 0x0)
    AND  R0, R1       // bitmask to reset bit 11
    STR  R0, [R4]     // store PA MODER


    // ***** Configure Button 1 (PC13) as input *****
    // Create bitmask to reset bits 26 and 27 of PC MODER
    MOV  R1, #0x0000  // 0000 1100 0000 0000 0000 0000 0000 0000
    MOVT R1, #0x0C00  // top half of above number
    MVN  R1, R1       // flip the bits, mask = 1111 0011 1111 1111 1111 1111 1111 1111
    // Read PC MODER value, reset bits 26 and 27, and write back to PC MODER
    LDR  R0, [R5]     // load PC MODER
    AND  R0, R1       // bitmask to reset bits 26 and 27
    STR  R0, [R5]     // store PC MODER

loop:

    // ***** Read status of Button 1 (PC13) *****
    // Create bitmask to read bit 13 pf PC IDR
    MOV  R1, #0x2000     // mask = 0010 0000 0000 0000
    // Read PC IDR value, reset all bits except bit 13
    LDRH R0, [R5, #0x10] // load PC IDR (PC base + 0x10)
    AND  R0, R1          // bitmask to reset all bits except bit 13

    // Check value. Zero = button pushed. Not zero = button not pushed.
    CMP  R0, #0          // compare with zero
    BNE  else            // if not pushed, go to "else", otherwise continue to "if"

if:
    BL turnonLED2  // call turnonLED2 function
    B loop         // return to top of loop
else:
    BL turnoffLED2 // call turnoffLED2 function
    B loop         // return to top of loop
```

(code continues on next page)

```
// This function turns off LED2.
// Note: This function assumes R4 still has PA base address
turnoffLED2:

   // Create bitmask to reset bit 5
   MOV R1, #0x0020 // 0000 0000 0010 0000
   MVN R1, R1      // flip the bits (MOVE NOT), mask = 1111 1111 1101 1111

   // read PA ODR value, reset bit 5, and write back to PA ODR
   LDRH R0, [R4, #0x14] // load PA ODR (PA base + 0x14)
   AND  R0, R1          // reset bit 5 using mask
   STRH R0, [R4, #0x14] // store PA ODR

   // return (end function)
   BX LR

// This function turns on LED2.
// Note: This function assumes R4 still has the PA base address
turnonLED2:

   // Create bitmask to set bit 5
   MOV R1, #0x0020 // mask = 0000 0000 0010 0000

   // read PA ODR alue, set bit 5, and write back to PA ODR
   LDRH R0, [R4, #0x14] // load PA ODR (PA base + 0x14)
   ORR  R0, R1          // set bit 5 using mask
   STRH R0, [R4, #0x14] // store PA ODR

   // return (end function)
   BX LR


   // end the file
   .end
```

Schematic:

A schematic is a wiring diagram that symbolically shows components and indicates how they are connected. In future labs, you will make your own schematic for the circuits you build. For now, the schematic is provided:



Lab Requirements

Getting started: Use the "Starting Code" to get started with the control registers for Button 1 and LED 2.

Adding to the code: Build on the starting code to additionally configure Button 3 as an input with a pull-up resistor and LED 4 as an output.

Completing a 'B' project:

In your loop, add to the code so that Button3 controls LED4 the same way that Button1 controls LED2. A working 'B' project will have a loop in Assembly that functions like the C code loop to the bottom left. Use conditional branches (such as BNE or BEQ) for the if/else blocks. Use linked branches (BL) for the function calls.

Completing a 'A' project:

In your loop, change the code so that LED 2 will turn on only if *both* buttons are pushed and LED 4 will turn on only if *exactly one* button is pushed, but not both. The loop in your code should function similar to the C code at the bottom right, but in ARM assembly. Use conditional branches (such as BNE or BEQ) for the if/else blocks. Use linked branches (BL) for the function calls.

Tips:

Note that the function calls themselves are not conditional but are instead placed inside a conditional block of code. That means that in assembly, you won't add a condition code to a linked branch itself (for example, BLNE is not allowed). Instead, do a conditional branch to a block of code that includes the linked branch. (The starting code has a simple if/else block.)

Equivalent C code loop for 'B' project

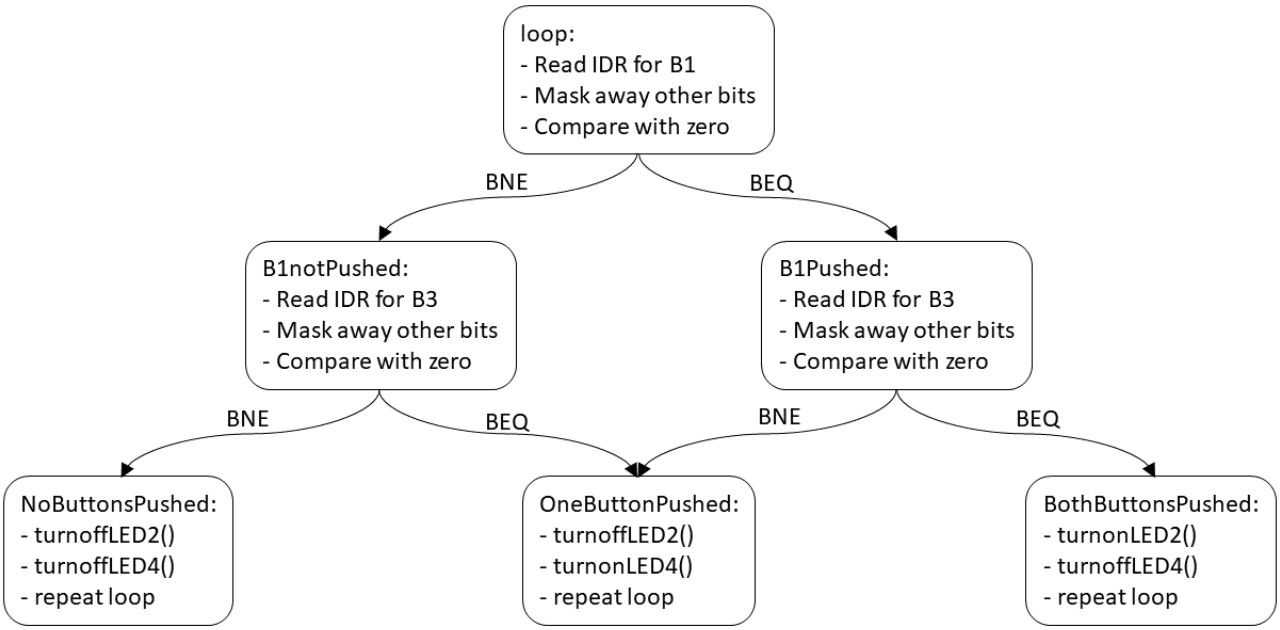
```
while(1){

  // read the status of the buttons
  ... code here ...

  // check status of first button
  if (button1 == pressed){
     turn_on_LED2();
  } else {
     turn_off_LED2();
  }

  /* You already have Assembly code
  ** for the above functionality.
  ** For a 'B' project, add the
  ** functionality below. **/

  // check status of second button
  if (button2 == pressed){
     turn_on_LED4();
  } else {
     turn_off_LED4();
  }

}
```

Equivalent C code loop for 'A' project

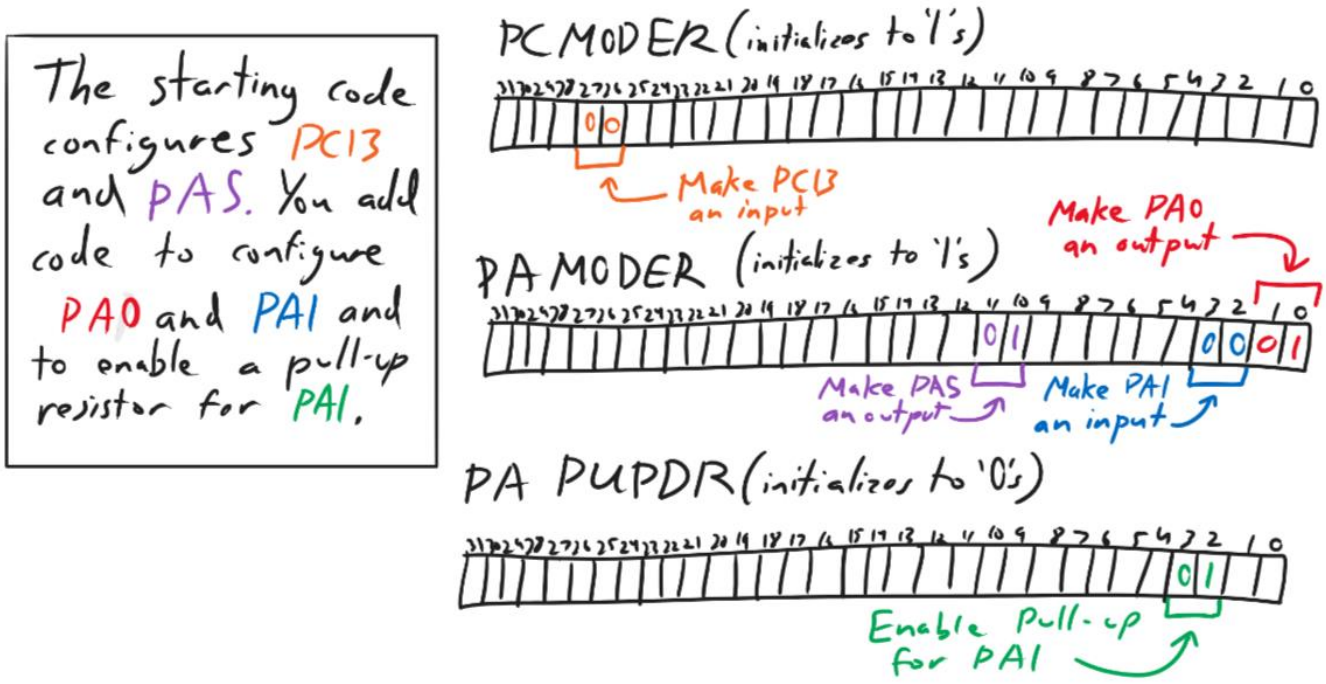
```
while(1){
   // read the status of the buttons
   ... code here ...

   // turn on/off LEDS
   if (button1 == pressed){
      if (button2 == pressed){
         // both buttons pressed
         turn_on_LED2();
         turn_off_LED4();
      } else {
         // one button pressed
         turn_off_LED2();
         turn_on_LED4();
      }
   } else {
      if (button2 == pressed){
         // one button pressed
         turn_off_LED2();
         turn_on_LED4();
      } else {
         // no buttons pressed
         turn_off_LED2();
         turn_off_LED4();
      }
   }
}
```

For the ‘A’ project, it may be simplest to create a unique label for each of the “if” or “else” sections and branch to the correct section based on the result of the comparisons. Below is a flow diagram of one possible way of doing the logic in the loop:

One way to do the logic in the loop:



Bit-level Configuration Visual:



Submission Requirements

Starting with this lab, there will be two I-Learn assignments for each lab: one for your photos and code ("Lab 04 Demo & Code") and a second for your lab report ("Lab 04 Report"), detailed in the next two sections.

Demonstration of Correct Operation

Once your lab is working correctly, take the four following photos.

1. Photo of system with no buttons being pressed.
2. Photo of system with only Button 1 pressed.
3. Photo of system with only Button 3 pressed.
4. Photo of system with both Button 1 & Button 3 pressed.

Then submit those four photos as well as a copy of your main.s file (5 files in total) to the "Lab 04 Demo & Code" assignment in I-Learn.

Lab Report Instructions

The second assignment in I-Learn for this lab is called "Lab 04 Report." Your lab reports will start very basic and progressively become more detailed as the semester progresses.

There is a lab report template provided in the "Lab 04 Report" assignment. For this first report, your report only needs to have three sections:

Overview: Your Overview section should provide the reader with a clear and brief introduction to the lab, effectively communicate the objectives/goals of the lab, and set the appropriate context for the report.

Code: Identify the section of your code that is most important and include just that code in your report.

Conclusion: In the conclusion section of your lab report, write two paragraphs: a one paragraph summary of what you accomplished and learned during this lab, and a second paragraph discussion of how this lab connects to things you learned in class, with things you knew before this course, or with things you hope to learn/do in the future.

Submit your report as a pdf to the "Lab 04 Report" assignment in I-Learn.

Stretch Yourself (optional)

As soon as you finish the lab, feel free to try connecting and wiring more lights and buttons. Play around with different Boolean logic in the code to see how you can make different input states can create different output states.