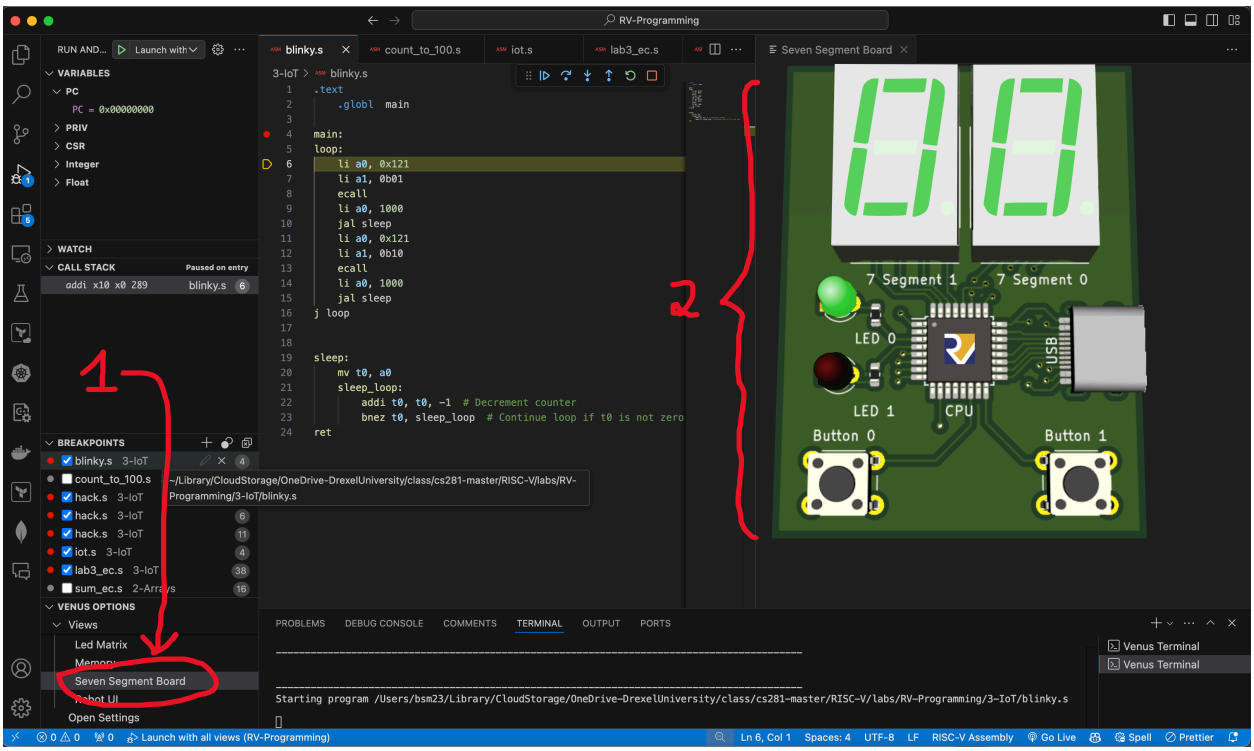
Lab3 – Having Fun with IoT and Hardware Interfacing

Introduction

One of the killer use cases for RISC-V is System on Chip (SoC), and IoT applications. The Venus simulator we are using has a fun integrated SoC microcontroller to play with. In this lab we will be playing with interfacing with push buttons, LEDs and multiple 7-segment digital displays.

PART 1: Bringing up the Simulated Microcontroller

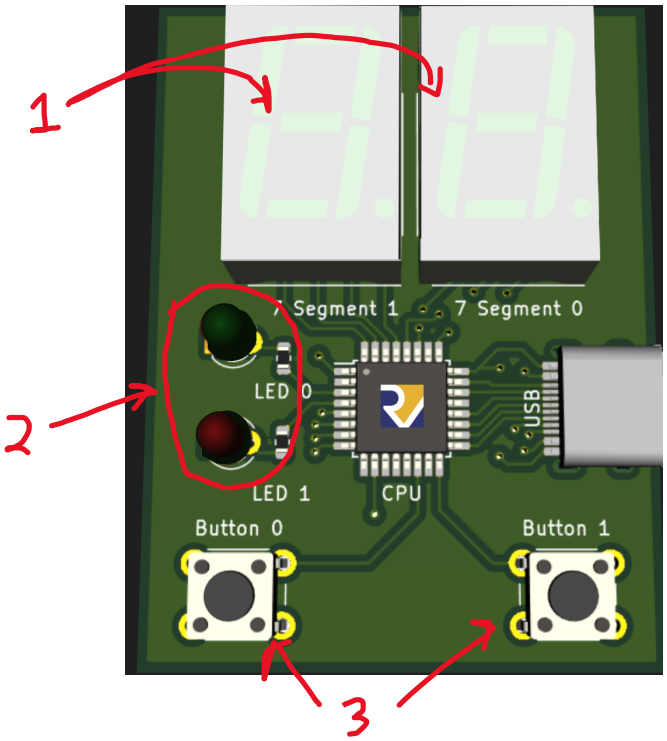
Lets tart by taking a look at the microcontroller and how you launch it from within VSCode:



Look at the picture above:

1. There are a couple of different ways to launch the microcontroller – you can put in your debugger configuration (launch.json), you can start it from the command pallet menu, or what I found to be simplest is to click on the “Seven Segment Board” option under “Venus Options” when the debugger is running.
2. When its running a graphical view of the microcontroller will be shown in another window.

The 7-segment board we will be playing with in this lab has several interesting features:



1. Two seven segment displays that we can control by turning on and off individual LEDs in the display to render numbers
2. Two leds, one is red and one is green, we can turn them on and off
3. Two input buttons, we can attach code to each of them to control operations on the board.

To get started I am providing several sample programs to demonstrate features of interfacing with the microcontroller board. All interactions use `ecalls`, similar to how we wrote to the console in other labs. More detailed directions on this interface can be found on the VSCode plugin page for the Venus simulator here: https://marketplace.visualstudio.com/items?itemName=hm.riscv-venus

The goal of this lab is to execute all of my sample programs, using the debugger and learning about their operation. After you execute my samples and understand them, you can use these samples as the basis for the code you will need to write that is described in part 3 of this lab.

Deliverables for Part 1:

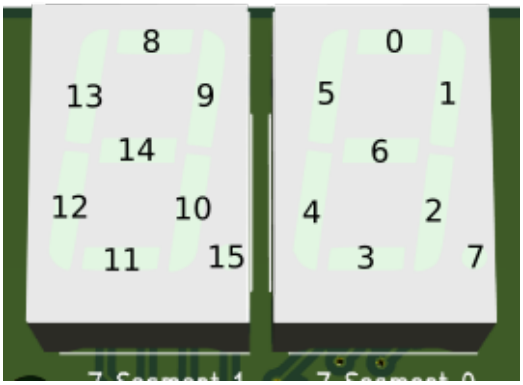
1. None

PART 2: Running the Sample Programs

This section describes several sample programs that I have provided to help you learn about interfacing with the microcontroller. Like most bare metal programming there is a setup process and then a loop that executes forever controlling the hardware behavior. If you have ever done microcontroller programming (e.g., the Arduino platform) you will see this is a common paradigm.

Sample	Demo Focus	Additional Information
blinkey	Blink the LED lights back and forth	Use of ecall 0x121 is used to interface with the led devices. This value must be passed in the a0 register. Register a1 is used to turn on and off the individual LEDs using a bit field. For example 0b00 turns off both LEDs, 0b11 turns them both on, 0b01 turns on the top (green) LED, and finally, 0b10 turns on the bottom (red) LED. Also note that this program injects a delay between switching the LEDs on and off to give a nice presentation. If we did not do this they would flash very quickly. The only way to simulate a delay is to go into a loop that does nothing – see the sleep: function at the bottom. Play with the delay time to see what happens
button_led	Blink a LED based on the button that was pressed	This program introduce ecall 0x122 which reads the status of the push buttons. The push buttons use the same strategy as the as the LEDs. After making the ecall, a0 will contain 0b01 if the left button is pushed, 0b10 if the right button is pushed, 0b00 if no buttons were pushed, and even 0b11 if both buttons were pushed (albeit its hard to press both buttons unless you stop the program in the debugger). To provide feedback on which button was pressed, the return value from ecall 0x122 is then passed as an argument to ecall 0x121 to update a LED.
double_flash_led	Loop and flash the LEDs twice	Another demo to integrate looping and simulating flashing LEDs, this will be useful in the ultimate program you are building for this lab
count_to_100	Update the seven segment display to count continuously from 0 to 100	This program uses the 2 seven segment displays to count from 0 to 100 forever. In introduces several things.

The final program that I provided is called count_to_100. This program uses the two 7-segment LEDs and continuously counts from 0 to 99. After it hits 99 the counter resets and starts back at zero. In introduces a number of things that you will need to master for your lab programming assignment. Before I further explan this program lets take a look at how the 7-segment displays are managed in the simulator.



The picture above from the documentation provides the bit location of each LED in the duel 7-segment display. Notice that the range goes from bit 0 (upper segment on right) to bit 15 (the dot on the left display). To update the display requires 2 separate pieces of data. The first is a sequence of bits that indicates which LEDs in the display to turn on or off. A value of 1 means turn on and a value of 0 means turn off. As an example, lets say you wanted to display “01” on the display.

```
     BIT NUMBER
  |--------------|
  1111110000000000
  5432109876543210
  ----------------
0b0011111100000110
```

The bitfield below represents “01” Notice that bits 1 and 2 are used to form the 1 on the right display, and bits 8-13 are used to create a zero on the left display.

The second field is a bitfield instructing the simulator which bits from your first bitfield should be updated, and which bits can be ignored. For example consider the bitfield:

```
     BIT NUMBER
  |--------------|
  1111110000000000
  5432109876543210
  ----------------
0b1111111111111111
```

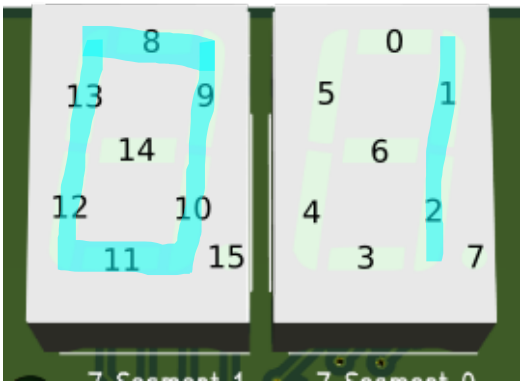
Note that the bitfield shown above contains a series of sixteen bits, all set to the value of 1. This second bit field is taken in conjunction with the first. Specifically, every bit in the second field that is 1 will be used to change the value of a LED in the seven segment display.

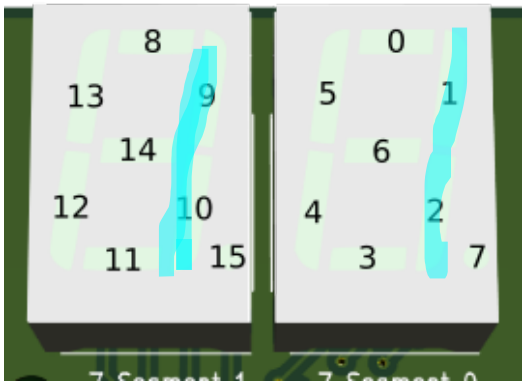
```
     BIT NUMBER
   |--------------|
   1111110000000000
   5432109876543210
   ----------------
0b0011111100000110    #the LEDs to create “01”
0b1111111111111111    #the bitfield
```

Given the above arguments, the bitfield has a 1 in every position, therefore every single bit in the first argument will be examined and adjustments to the display will be made accordingly. This means that each led will be set – let “0” = turn off, and let “x” = turn on:

```
     BIT NUMBER
   |--------------|
   1111110000000000
   5432109876543210
   ----------------
0b0011111100000110    #the LEDs to create “01”
0b1111111111111111    #the bitfield
   ----------------
   ooxxxxxxoooooxxo
```

We can see from the above that all 16 segments were updated, turning each one either on or off. You may be asking yourself why do we have the bitfield?





Take a look at the picture above, lets say we want to transition the value shown on the display from “01” to “11”. First lets calculate the bitfield to display “11”: `0b0000011000000110`. We could now update the display using this new value and set the bitfield to be all 1’s as shown above. However, computers operate on numbers rapidly, but changing the physical state of every LED in the segment takes some time in real hardware. If you look carefully, consider moving from “01” to “11” in terms of which segment values changed:

```
     BIT NUMBER
  |--------------|
  1111110000000000
  5432109876543210
  ----------------
0b0011111100000110    #the LEDs to create “01”
0b0000011000000110    #The LEDS to create “11”
  ----------------
0b0011100100000000    #The bits that changed – a logical XOR
  ----------------
0b0011111100000110    #The final value, (“01” xor “11”) or “11”
```

Carefully examine the transitions shown above. We start with the bitfield of the previous value “01” in this case, and then take the xor function with the new value, “11” in this case. This identifies the bits that have changed. We then take the logical or function with the new value (“11”) to end up with the final bit field value. This value is interpreted as any bit that has a 1 will be updated, and any bit with a 0 will be ignored. In other words, to go from “01” to “11” bits 15,14,7,6,5,4,3, and 0 can be ignored. This is far more efficient, adjusting the state of 8 segments vs adjusting the state of all 16 segments. There will be an extra credit option in your lab to implement this optimization.

Now back to the count_to_100 program. Run the program and observes what happens. Then take a look at the code. Things to call to your attention:

1. Note that in the .data segment there is a label called digits. This is really an array, where `digit[0]` is the bitfield to render a 0 on the right segment, `digit[1]` is the bitfield for 1, and so on.
2. For this program the `digit_msk` variable is hard coded to a 16 bit string of 1’s. Thus we will be re-rendering the entire seven segment display on each pass.
3. We also have variables for `digit_max`, which will be used to control the loop, we reset after 99 and count from 0 to 99 so `digit_max` is 100, and `digit_sz` is 10, which is the size of the digit array.
4. In the `main` function we have a loop. The first thing in this loop is a `blt` check, note that we are keeping track of the current number we need to display in register `s0` so, so if `s0<100` we just go on, but if `s0=100` then we reset `s0=0`. This handles the wrap around situation.

5. The `write_lcd` function does a few things. Lets say our counter is currently at 25. Remember we don't update the segments using the value of 25, we need to separate 25 into the number "2" in the 10's place and the number "5" in the 1's place – this is called a encoded decimal. After all 25 is $2*10^1 + 5*10^0$. This is what the `extract_bcd` function does. If you call it with 25 in register `a0`, it returns with 2 in register `a0` and 5 in register `a1`.
6. The `encode_digit` function takes these 2 variables and calculates the proper bitfield. Keeping with the example of "25". This function receives the value of 2 and the value of 5 as arguments in registers `a0` and `a1` respectively. From there:
    a. It starts with the number 2 in the 10's position.
    b. It looks up the bitfield for the number 2 in the `digit` table. This value is: 0b01011011.
    c. If you look carefully at how the LEDs are defined, you will notice that the left LED has the same structure as the right one, however the bits do not start at position 0 they start at position 8. So if we shift this value to the left by 8 positions using `slli` then it now looks like: 0b0101101100000000.
    d. We then look up the number 5 in the `digit` table. This value is: 0b01101101.
    e. Finally we logically `or` the values from steps (c) and (d) to get the complete bitfield for the number "25": 0b0101101101101101.
    f. After returning to the `write_lcd` function the bitfield calculated in step (e) will be in register `a0`. We then load the value for the `digit_msk` and setup to make the call to update both segments.
    g. To update the segments we use ecall 0x120, which is passed in register `a0`. The bitfield for "25" is passed in register `a1`. And finally, the bitmask containing all 1's to keep things simple is passed in register `a2`.

I spent a significant amount of time in this section explaining the sample programs provided. You will use these samples to write a program that puts all of these concepts together. This will be described in the next section.

There is nothing to hand in for this section of the lab, but making sure you understand these programs will make writing the program for your assignment much easier.

PART 3: What you need to do

We are now ready for you to write some code, and to hopefully have some fun. First start by loading up the scaffolded program that I provided called lab3_starter.s – this will accelerate your effort. Using the starter, implement the following functionality:

1. When the program first starts, reset the 7-segment display to indicate a starting value of "00".
2. When the left button is pressed, increment the value displayed, for example going from "00" to "01".
3. When the right button is pressed, decrement the counter and update the display. For example going from "01" to "00".

4. Handle the boundary conditions. So if you are at “00” you cant decrement, and if you are at “99” you cant increment. If this condition happens, do not update the counter and provide the user feedback about the issue by blinking both LEDs (the green and red ones) twice.

Make sure your code is well documented and is easy to follow. There are TODO instructions for you in the starter.

The above describes the basic requirement for this lab, hand in the code that implements the above and you are all set. If you have interest, I am also offering some extra credit for expanded functionality. Here are some extra credit options:

1. (+3 pts). Modify the code so that if we try to decrement at “00” or increment at “99” the flash_led function will not just hard code 2 blinks. Instead it will receive a parameter in register `a0` and blink that number of times. Hand in your code so that your program blinks 5 times during these error conditions. Do not hard code 5 blinks, flash_led(5) will take an argument and loop.
2. (+3 pts). Modify the code so that the “red” LED will be on if the current value is an odd number and the “green” LED will be on if the current value is an even number. Hint: all odd numbers have their least significant bit as a 1, and all even numbers have their least significant bit as a 0. Think about a logical function you can use to test the value of the least significant bit.
3. (+4 pts). This last extra credit option can be done standalone, or in conjunction with part 2. Basically the goal is to examine the current value and turn on BOTH LEDs if the current value is a power of 2. Hint: All numbers that are a power of 2 have exactly 1 bit on. There is a vary fast way to check for this:

   Algorithm:

   ```
   if num == 0 then NOT_A_POWER_OF_2 else
   if (num bitwise_and (num-1)) != 0 then NOT_A_POWER_OF_2
   else NUM_IS_A_POWER_OF_2
   ```

Deliverables for Part 3:

Please hand in the source code showing your completed program– after you have tested it. If you attempted any of the extra credit options, clearly indicate this at the top of your program in a comment.