ECE-395 RISC-V Update

2. Microprocessor Experiment Platform

Microprocessor used for this experiment is FE310-G002 utilizing RISC-V core on HiFive 1 rev B board.

HiFive 1 rev B features:

- SiFive E31 Core Complex up to 320MHz
- Flexible clocking options including internal PLL, free-running ring oscillator and external 16MHz crystal.
- 1.61 DMIPs/MHz, 2.73 Coremark/MHz
- 8kB OTP Program Memory
- 8kB Mask ROM
- 16kB Instruction Cache
- 16kB Data SRAM
- Wireless communication
- External RESET pin
- JTAG, SPI I2C, and UART interfaces.
- Requires 1.8V and 3.3V supplies
- Hardware Multiply and Divide

Software development tools that support HiFive 1:

- PlatformIO
- Arduino IDE (Linux/Mac)

For this particular course PlatformIO ecosystem will be used to setup the toolchain and Visual Studio Code will be the IDE.

Visual Studio Code (with PlatformIO) feutures:

- Support for over 700 different development boards
- Support for C, C++ and assembly
- PIO Unified Debugger

2.1 Initial Tool Setup

The following set up is used only for Windows PC in order to develop and debug FE310-G002

Step 1: From Visual Studio website, download and install VSCode (https://code.visualstudio.com/)

Step 2: Install C++ and PlatformIO extensions.

Open VSCode.: Extensions□Type “C++” □ Install C++ (Figure 1). Conduct the same procedure to install “PlatformIO IDE”

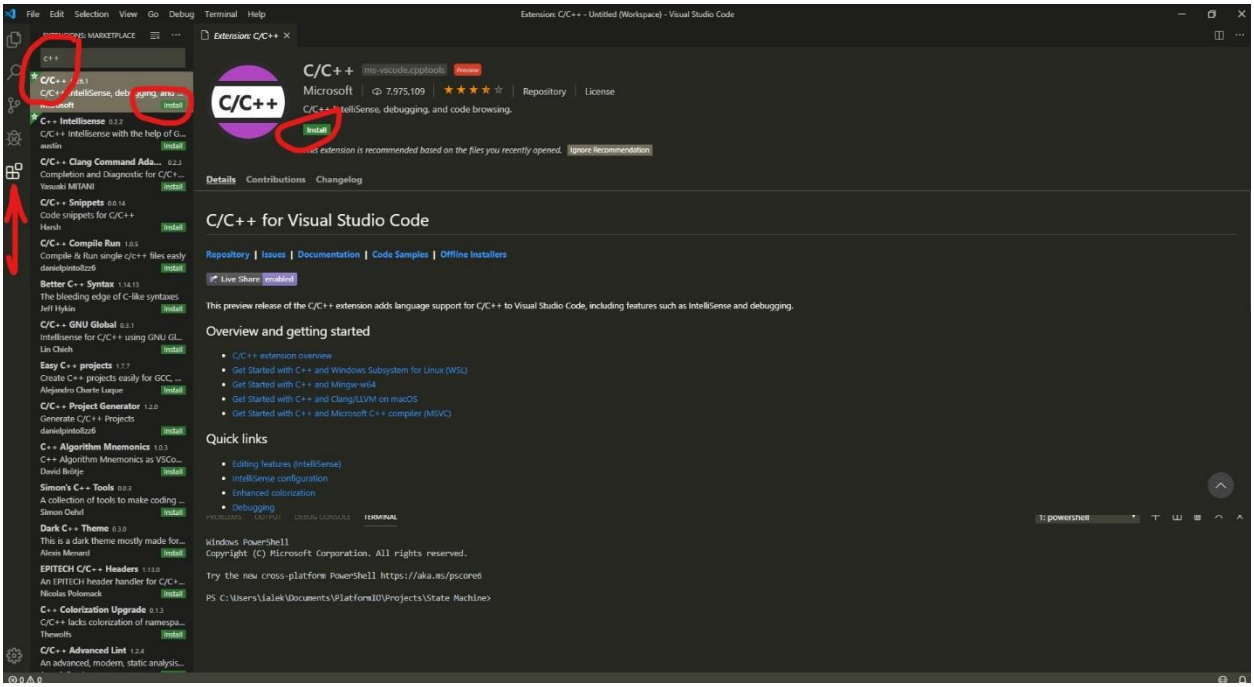


Figure 1

Step 3: Make sure J-Link is up to date and installed. (PlatformIO occasionally "forgets" to install J-Link drivers). Navigate to C:\Users\USERNAME\.platformio\packages\tool-jlink\USBDriver\x64 and execute "dpinst_x64".

Step 4: Project creation. Navigate to PlatformIO homepage. (Figure 2). New Project > Enter project name > Select HiFive 1 rev B boards > Finish

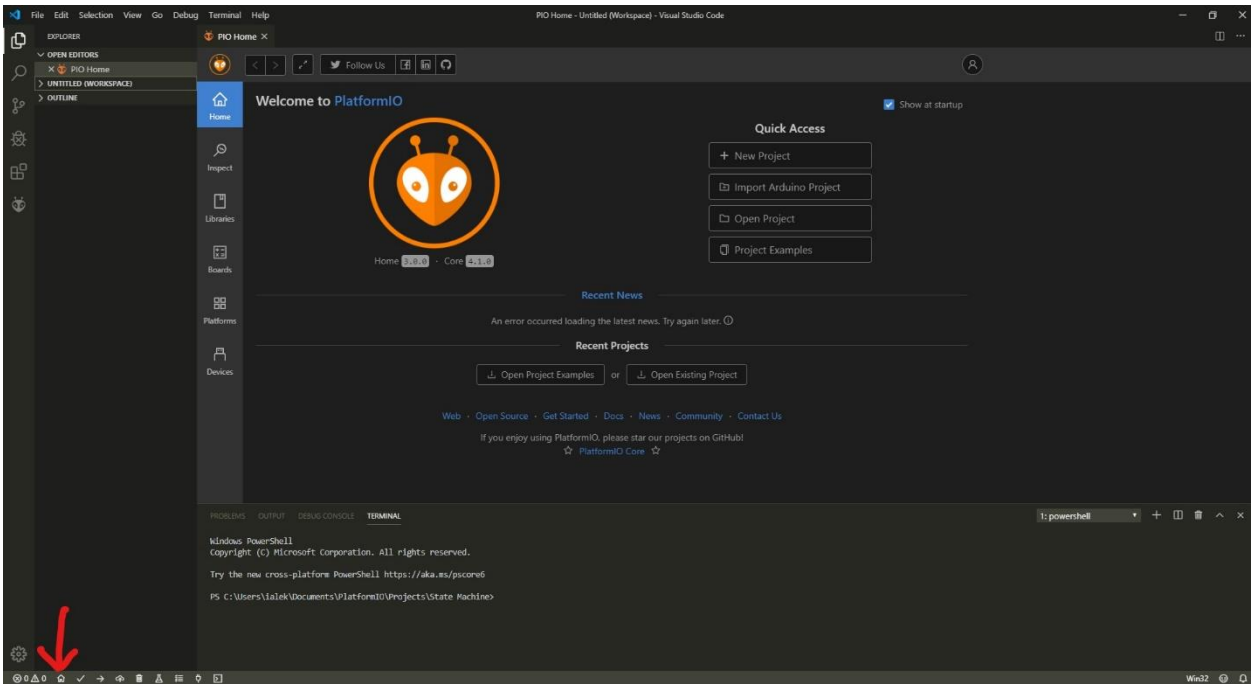


Figure 2

3. Experiment

3.1 Lab 1 – Microprocessor Operation

Lab Objective

- To learn to create a PlatformIO project and write, assemble and debug code
- To observe and document operation of microprocessor core as it executes code

Procedure

Part 1: First a project must be created. In order to do so, navigate to the PlatformIO homepage and follow New Project > Enter project name > Select HiFive 1 rev B boards > Finish. Upon creating of the project, a project folder will be shown in the explorer. Navigate to the "src" folder and create *main.c* and *asm_main.S* in that folder (right click on "src" and then "create a new file"). Copy the following code into the files.

This step demonstrates the basics of memory access and moving data withing the processor.

Add the shown code for *main.c* and *asm_main.S*

.section .text is required command for writing code in assemby, it specifies that the text bellow is code. **.align 2** is used for performance purposes and is part of good programing practice in RISC-V

.globl *asm_main* makes *asm_main* function accesible from *main.c*

.equ offset, 0x80000000 is the same as **#define** in C++, it makes **offset** (name) equal to the hexadecimal value and is a valuable tool for making your code more readable.

The first load is **li,** it moves the hexadecimal value into the **a1** register. Then the **offset** is loaded into the **a2** using **li** command. (note that **data** and **offset** are just hexadecimal values that were assigned those name using **.equ** command earlier). Next the **sw** command will be used to load **a1** into the momory location specified by the **a2**.

lw will load the word from memory to the **a3**, **lh** will load half word to **a4**, and **lb** will load a byte from memory to **a5**. Note the difference in values that are loaded from the same memmory location.

The last two move instructions show how to coppy values between registers. First one copies register value from **a1** to **a2**. The second instruction copies **x0** into **a1**. **x0** is a special register that is always grounded (zero), therefore by copying it into any other register it errases the value of the destination register. It is important to point out that **mv** instruction is a pseudoinstruction, meaning that it uses base instruction to accomplish the task. **mv** instruction is actually **addi rd, rs, 0**. It adds 0 to the source register and saves it in the destination register.

For each part of this lab, use provided worksheet and record required values using a debugger.

Remember that the value in registers and memory updates only AFTER the instruction.

```c
#include <stdio.h>
void asm_main();

int main()
{
    while(1)
    asm_main();
return 1;
}
```

main.c

```
.section .text
.align 2
.globl asm_main

.equ offset, 0x80000000
.equ data,   0xDEADBEEF
# add program code here
asm_main:
   li a1, data              # load immediate (32-bit)

   li a2, offset            # load offset address to a2
   sw a1, (a2)              # save a1 into the memory location specified by a2
   lw a3, (a2)              # load word (32-bit) from mem
   lh a4, (a2)              # load half word (16-bit) from mem
   lb a5, (a2)              # load byte (8-bit) from mem

   mv a2, a1               # copy a1 to a2
   mv a1, x0               # clear a1. x0 is always '0'
 ret
```

asm_main.S

USE OF A DEBUGGER

First build the program, at the bottom of VSCode environment, right next to the PIO home button press the Build button to build the project  (PIO home is the house icon, build function is the checkmark and upload to board is the arrow). Next upload the project to a **CONNECTED** board

Next navigate to the debugger and run it for this project 

using green square (make sure no other project is open, if it is, close project and reload VSCode)

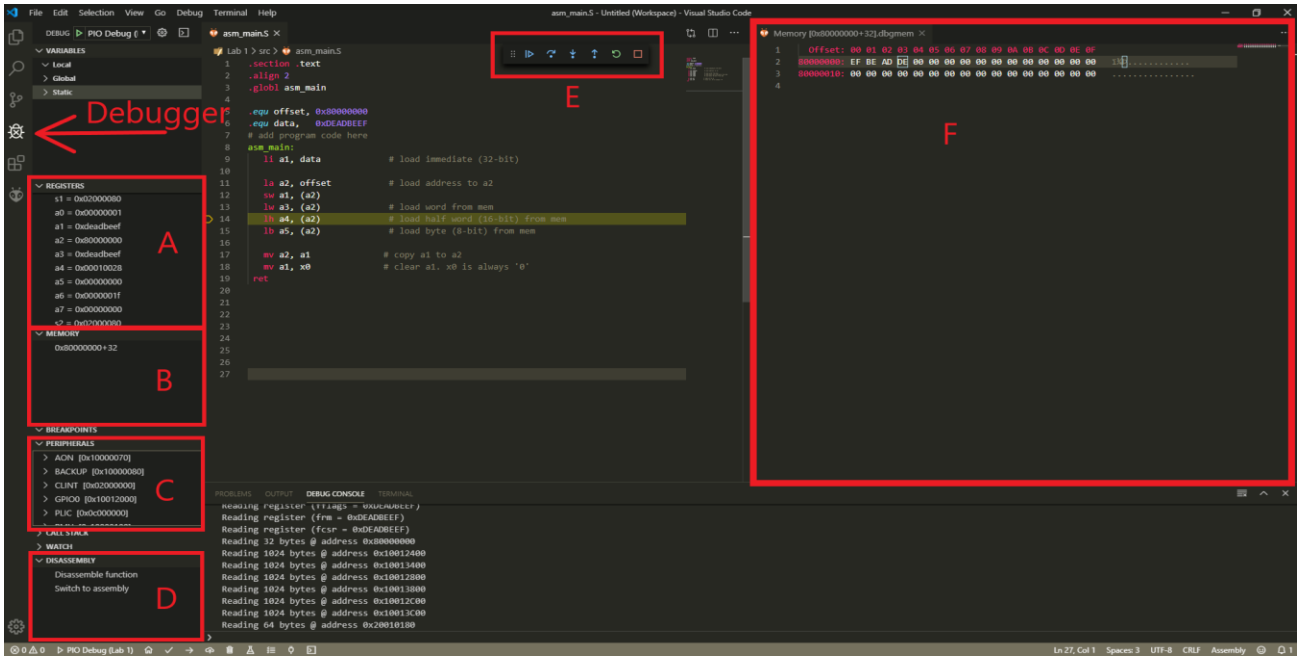


Figure 3

Figure 3 shows the typical debug window display.

Section A lists the board's registers.

Section B is the memory look up. It can be used by adding the memory address and specifying byte offset (in order to see one memory location at a time use 4 as byte offset). The content of the memory location will be displayed in the **Section F**.

Section C contains all the memory addresses for the onboard peripherals for easy access.

Section D contains disassembly, it will disassemble the C code into assembly. The assembly code will remain the same.

Stepping through code with the debugger can be achieved using the debug command gadget, **Section E**.

Play sign is to pause or continue the debugging session. Next to it is the step over function, it will step over the next functions. The down arrow will step into the next function and go through its code line by line. The down arrow will be used the most in this lab since we do not need to step over or go back. The up arrow is used to get out of the function, it will not go to the previous line but rather exit the current function. Restart function will restart the debugging session and the stop button will exit the session.

Accesing memory location is a very important part of the debugging procces. In order to do so the debugging session must be active. In the memory part of the debugger 

press the plus sign and in the prompted window enter the desired adress in hexadecimal format (ex. 0xDEADBEEF) and press enter. In next prompted window enter offset (for one location enter ‘4’).

Part 2: This step demonstrates some of the basic arithmetic and logic operations

Erase previous code in *asm_main.S* and copy the code bellow

```
.section .text
.align 2
.globl asm_main

.equ data1,  0x01010101
.equ data2,  0x10101010
# add program code here
asm_main:
    li a1, 100              # load 100 into a1
    li a2, 150              # load 150 into a2
    add a3, a2, a1          # add a1 and a2 and place it in a3
    sub a4, a2, a1          # subtract a2-a1 and place it in a4
    addi a1, a1, 100        # add immediate to a1 and place the result into a1

    li a1, data1            # load immediate
    li a2, data2
    and a3, a2, a1          # AND both a2 and a1 and place it into a3
    or a4, a2, a1           # OR a2 and a1
    xor a5, a2, a1
 # xor is widely used in microprocesors for masking the registers and insulating
 # the bit that is of interest

 ret
```

Part 3: In this part use of unconditional branches will be shown.

Each label (i.e. spot1, spot2, spot 3 and spot4) has a memory address associated with the instruction following the label. When the branch instruction (i.e. spot3) executes occurs, the program counter is changed to reflect the address associated with the label.

Erase previous code from *asm_main.S* and copy the following code

```
.section .text
.align 2
.globl asm_main



# add program code here
asm_main:
    j spot1

spot1:
    j spot4

spot2:
    j exit

spot3:
    j spot2

spot4:
    j spot3

exit:
    ret
```

Worksheet requires values for PC (program counter) which can be found during debugging in register section of the debugger.

Part 4: This section demonstrates conditional branching

Unlike the unconditional branch demonstrated in the previous step, the conditional branch uses the state of the processors flags to control the flow of the program. The branch is only taken if the condition for the specific branch instruction is met.

Erase previous code from *asm_main.S* and copy the following code

After executing and debugging the following code, change **a1** to 0x0 or 0x1 and observe the difference in branching

```
.section .text
.align 2
.globl asm_main



# add program code here
asm_main:
    li a1, 0xdeadbeef       # keep changing a1 to observe different branching
    li a2, 0xdeadbeef
    beq a1, a2, spot1       # branch if a1 and a2 are equal
    beqz a1, spot2          # branch if a1 is 0
    bne a1, a2, spot3       # branch if a1 is not equal to a2

spot1:
    ret

spot2:
    ret

spot3:
    ret
```

Part5: This step demonstrates the use of jump and link instruction.

The jump and link instruction are used the most in subroutine calls. When the **jal** is executed, a return address is saved to the return address stack and **x1**. When the destination function calls **ret** it returns to the address that is on the top of the return address stack which is the address that **jal** was called from thus resuming normal operation after a subroutine.

Erase previous code from *asm_main.S* and copy the following code

```
.section .text
.align 2
.globl asm_main

.equ value1, 0x80000000     # value1 and value2 are treated as addresses
.equ value2, 0x80000010
# add program code here
asm_main:
    li a2, value1
    jal change_value        # call change value for value1

    li a2, value2
    jal change_value        # call change value for value2

    j asm_main              # do it again

change_value:
    lw a3, (a2)             # load the content of the memory

# operating on retrieved date from memory location a2
    addi a3, a3, 1          # increment the content
    xor a3, a3, a2          # XOR the adress with the incremented content
    li a4, 0xFF             # create a mask
    and a4, a4, a3          # AND mask and previous result in a3

# these operations are just arbitrary set of operations that do not
# accomplish anything meaningful. It only changes the value
# retrieved from memory location
    sw a4, (a2)             # save the result back to the memory
    ret
# will use saved return address to resume operation in asm_main
```

3.2 Lab 2 – General Purpose Inputs and Outputs

Lab Objectives

- To learn how to configure MCU internal peripherals
- To learn how to operate the GPIO pins

Background

When creating a design using a processor, it is common to need inputs or outputs that operate in a binary (on/off) fashion. These signals are used for monitoring user inputs (switches or pushbuttons), driving indicators (lights or audible), controlling actuators, monitoring/driving discrete control lines from/to other circuits in the design, or for a variety of other purposes. Microcontrollers typical will have pins that can be configured as either outputs that can be driven by or inputs that can be monitored by the processor. These signals are commonly referred to as general purpose inputs and outputs (GPIO's).

The HiFive 1 board provides 24 GPIO pins that can be used for their assigned special functions. If a specific peripheral which is tied to a given pin is not used in a design, the pin is available for use as a GPIO. When selecting pins to use as a GPIO, it is important to avoid pins that are assigned to a special function that will also be used in the design. For example, if the UART was to be used in a design, pins associated with the UART functions could not be used as GPIO.

For the FE310 used in HiFive 1, section 17 of SiFive FE310-G002 Manual lists all functions that the GPIO complex has. The manual also provides with all the base addresses, address offsets and bitfield offsets for the FE310.

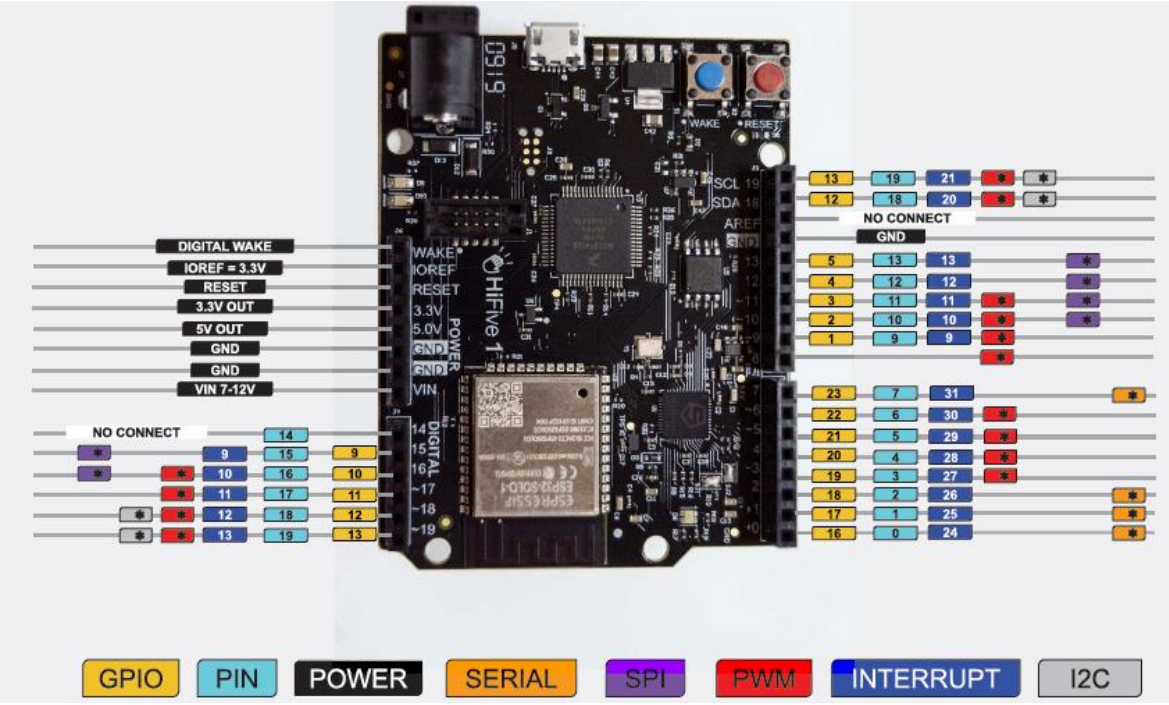


Figure 5

From SiFive HiFive1 Rev B Getting Started Guide

Once a GPIO is selected, several registers must be properly configured in the processor to allow it to be used as an input or output.

1. Each GPIO must have either input or output enabled. SiFive FE310-G002 Manual provides us with the GPIO base address and all the address offsets for registers that will modify the GPIO

Offset	Name	Description
0x00	input_val	Pin value
0x04	input_en	Pin input enable*
0x08	output_en	Pin output enable*
0x0C	output_val	Output value
0x10	pue	Internal pull-up enable*
0x14	ds	Pin drive strength
0x18	rise_ie	Rise interrupt enable
0x1C	rise_ip	Rise interrupt pending
0x20	fall_ie	Fall interrupt enable
0x24	fall_ip	Fall interrupt pending
0x28	high_ie	High interrupt enable
0x2C	high_ip	High interrupt pending
0x30	low_ie	Low interrupt enable
0x34	low_ip	Low interrupt pending
0x40	out_xor	Output XOR (invert)

From SiFive FE310-G002 Manual

2. Each pin is controlled by their respected bitfield offset in the GPIO control register. Some of these offsets for the control registers are already in the *GPIO.inc* and *memoryMap.inc. GPIO.inc* holds all the needed offsets for the GPIO registers. *memoryMap.inc* holds all the base addresses of the memory.
3. For the output operation the pin output enable must be set, output value must be cleared to eliminate randomness after the boot and, for convenience, Output XOR can be enables in order to exercise a more intuitive output. For inputs the input enable must be asserted and either the internal pull up enables or outside pulldown resistors must be added. For the input, all the pin values are stored in the 1st register from the GPIO base address

There is an LED on the HiFive 1 is a tri-color red/green/blue device. The common anode is tied to VDD. The three cathodes are tied through resistors to GPIO's as listed in Table 1. Because the LED is wired with the common anode to VDD, the GPIO's must be driven low to run on the LED color and driven high to turn off the LED.

LED Color	GPIO offset
RED	22
GREEN	19
BLUE	21

Required Equipment and Parts

• Solderless breadboard
• Pushbutton
• Jumper wires

Procedure

The aim of this part of the lab is to learn how to control an LED. Code is given that blinks the LED red. You will modify it to: explore the various control options, change the LED color, change the blink rate and brightness

Step 1: Create assembly include files to store addresses. All source files must be created in "src" folder of the project workspace. Create following files: *"GPIO.inc"*, *"memoryMap.inc"*, *"main.c"* plus a header file .h that will be discussed downstream. This file set up is not exhaustive but will be used in majority of labs in this course. The *.inc* files include addresses and offsets for various board functions and peripherals. It is an equivalent of header file .h in C++.

Step 2: Fill in the *"GPIO.inc"* and *"memoryMap.inc"*. Both include files are not complete but are enough for the rest of the course. If needed new addresses and offsets can be added to both. Remember, these files do not add any new functionality to your code. The reason for separating the addresses and equating them with words is for readability purposes only. All addresses and offsets can be manually entered in assembly program.

```
.equ GPIO_OUTPUT_EN,  0x008     # Enable Output to selected pins
.equ GPIO_OUTPUT_VAL, 0x00C     # Set Output Value
.equ GPIO_OUTPUT_XOR, 0x040     # inverse logic on selected pins
.equ GPIO_INPUT_EN,   0x04      # Set Input enable for selected pins
.equ GPIO_INPUT_VAL,  0x00      # Read Input value of selected pins
.equ GPIO_INPUT_PULUP, 0x10     # enable pull up

.equ PIN_2,           0x40000    # GPIO pin 2 offset

.equ GPIO_RGB_PINS,   0x680000   # All 3 LED's bit offset
.equ GPIO_RED_LED,    0x400000   # Red LED offset
.equ GPIO_BLUE_LED,   0x200000    # Blue LED offset
.equ GPIO_GREEN_LED,  0x080000   # Green LED offset
```

GPIO.inc

```
.equ GPIO_CTRL_ADDR, 0x10012000
.equ UART_CTRL_ADDR, 0x10013000
.equ MTIME, 0x0200BFF8
.equ MTIME_FREQUENCY, 33
```

memoryMap.inc

Step 3: Load the code shows below into the *main.c*

```c
#include <stdio.h>
#include <header1.h>

int main()
{
    int error = 0;
    int ledNum = 0;
    int colors[NUM_LEDS] = {GREEN_LED, BLUE_LED, RED_LED};

    setupGPIO();

    while(!error)
    {
        setLED(RED_LED, ON);
        delay(DELAY);
        error = setLED(RED_LED, OFF);
        delay(DELAY);
        if(ledNum >= NUM_LEDS)
            ledNum = 0;
    }
return 0;

}
```

main.c

Step 4: Load the following code into *header1.h*

```c
#define DELAY           200
#define ON              0x01
#define OFF             0x00
#define NUM_LEDS        0x03

#define RED_LED         0x400000
#define BLUE_LED        0x200000
#define GREEN_LED       0x080000

void setupGPIO();
int setLED(int color, int state);
void delay(int milliseconds);
```

header1.h

Step 5: Load the following code into *setupGPIO.S*. This file is used to set up all the necessary memory locations in order to use GPIO. Notice that this setup function also sets GPIO for inputs that will be used in the next part of this lab.

```
.section .text
.align 2
.globl setupGPIO

#include "memoryMap.inc"
#include "GPIO.inc"


setupGPIO:
    addi sp, sp, -16
# allocate a stack frame, moves the stack up by 16 bits
    sw ra, 12(sp)

    li t0, GPIO_CTRL_ADDR     # load GPIO base address
    li t1, PIN_2              # load address of pin 2 into t1
    sw t1, GPIO_INPUT_EN(t0)  # set pin 2 for input enable
    sw t1, GPIO_INPUT_PULUP(t0) # enable pull up
    li t1, GPIO_RGB_PINS        # get RGB pins offset
    sw t1, GPIO_OUTPUT_EN(t0)
 # write the GPIO RGP pins to GPIO Enable offset
 # (Enable output on RGP pins so we can write)
    sw t1, GPIO_OUTPUT_XOR(t0)  # set the XOR so the RGB pins are active high
    sw x0, GPIO_OUTPUT_VAL(t0)

    lw ra, 12(sp)               # return the return address
    addi sp, sp, 16             # deallocate the stack
    ret
```

setupGPIO.S

Step 6: Copy following code to *setLED.S*

```
.section .text
.align 2
.globl setLED
#include "memoryMap.inc"
#include "GPIO.inc"
.equ NOERROR,  0x0
.equ ERROR,    0x1
.equ LEDON,    0x1

# which LED to set comes into register a0
# desired On/Off state comes into a1

setLED:
    addi sp, sp, -16
# allocate a stack frame, moves the stack up by 16 bits
    sw ra, 12(sp)               # save return address on stack

    li t0, GPIO_CTRL_ADDR           # load GPIO address
    lw t1, GPIO_OUTPUT_VAL(t0)      # get the current value of the pins

    beqz a1, ledOff             # Branch to ledOff if a1 == OFF (0x0)
    li t2, LEDON                # load up value of LEDON into temp register
    beq a1, t2, ledOn           # branch to ledOn if a1 == ON (0x1)
    li a0, ERROR                # return an error for a bad status request
    j exit

ledOn:
    xor t1, t1, a0              # XOR to only change the value of requested LED
    sw t1, GPIO_OUTPUT_VAL(t0)      # write the new output value to GPIO out
    li a0, NOERROR                  # no error
    j exit

ledOff:
    xor a0, a0, 0xffffffff
# invert everything so that all bits are one except the LED we are turning off
    and t1, t1, a0
# AND a0 and t1 to get the LED we want to turn off
    sw t1, GPIO_OUTPUT_VAL(t0)      # write the new output value
    li a0, NOERROR

exit:
    lw ra, 12(sp)                   # restore the return address
    addi sp, sp, 16                 # deallocating stack frame
    ret
```

setLED.S

Step 7: Finally copy following code into *delay.S*

```
.section .text
.align 2
.globl delay

#include "memoryMap.inc"
# a0 is milliseconds passed through parameter

delay:
    addi sp, sp, -16
# allocate a stack frame, moves the stack up by 16 bits
    sw ra, 12(sp)               # save return address on stack

    li t0, MTIME                # load the timer register
    lw t1, 0(t0)                # load the current value of the timer
    li t2, MTIME_FREQUENCY      # get our approximate clock freq
    mul t2, t2, a0              # multiply milliseconds with freq
    add t2, t1, t2              # the target MTIME is now in t2
1:
    lw t1, 0(t0)                # read M value again
    blt t1, t2, 1b              # keep looping until time out

    lw ra, 12(sp)               # restore return address
    addi sp, sp, 16             # deallocate the stack frame
    ret
```

delay.S

Exercise:

1. Modify main.c code to alternate between red, green and blue LED using the colors array.
2. Record the value of the output_val register at 0x0C for each LED. To do so debugger must be started and the memory location at 0x0C must be observed (use offset of 4 bytes for lookahead).
3. The LED blink rate is controlled by the value in the equate DELAY in *header1.h* which is used to set how long the time between on and off will be. Run the original code and **measure the blink rate** by counting the number of times the LED blink in a given interval (Hint: Use the stopwatch feature on a phone to time 30 or 60 seconds).
   Change increase the DELAY by 50%. **Measure the blink rate.**
   Change increase the DELAY by another 50%. **Measure the blink rate.**
4. Modify the code to turn on an external LED connected to one of the GPIO ports. You may use pin 2 with bitfield offset of 0x40000, if any other pin is desired refer to FE310 manual.

Part 2: GPIO's as Inputs

The objective for this lab is to process an input to the GPIO pin and provide a certain output. A pushbutton will be wired to the GPIO. The HiFive1 will recognize the input and turn an LED on.

Step 1: Replace the loop in the *main.c* with code bellow. Connect a normally open pushbutton between ground pin and **Pin 2** on HiFive1. Add `int checkBot();` to the *header1.h*

```
#include <stdio.h>
#include"header1.h"
int main()
{
    setupGPIO();

    while(1)
    {
        if(checkBot()) setLED(0x400000,ON);
        else setLED(0x400000,OFF);

    }
    return 0;
}
```

Step 2: In order to check the status of the input pin a separate function must be written. Create checkBot.S, this file will have the functions needed to check the status of the pin and to return the found value. Additionally, since we are using **Pin 2,** we must know its bit offset in order to setup the memory locations and to retrieve the pin status. This information can be found in the SiFive FE310-G002 Manual.

Step 3: Copy following code into checkBot.S.

```
.section .text
.align 2
.globl checkBot
#include "GPIO.inc"
#include "memoryMap.inc"

.equ ON, 0x1
.equ OFF, 0x0
checkBot:

addi sp, sp, -16                    # allocating stack frame
sw ra, 12(sp)                       # Saving return address to the stack

li t0, GPIO_CTRL_ADDR               # load GPIO base address
lw t1, GPIO_INPUT_VAL(t0)           # add the offset for the READ register
li t2, PIN_2                        # load the BIT offset of PIN2
and t2, t1, t2                      # AND the READ register with the PIN2 offset

beqz t2, pinOFF
# if t2 is 0x0 the READ register has 0 at the PIN2 offset, branch to pinOFF
li a0, ON                           # return ON comand back to the main
j exit

pinOFF:
li a0, OFF                          # return OFF comand back to the main
j exit

exit:
lw ra, 12(sp)                       # restore the return address
addi sp, sp, 16                     # deallocate the frame
ret
```

checkBot.S

Exercise:

1. Record the value of at the GPIO input memory location with and without the button press. (Remember that the button must be held or released during debug session)
2. Explain what GPIO_INPUT_PULUP address does and how can the pull up be avoided.

3.3 Lab 3 – Annunciator (GPIO application)

Lab Objective

- To apply knowledge learned in lab 2 to real world application

Problem

A maple syrup factory in Vermont has a problem. They have a holding tank that stores their product that overflows from time-to-time. When this happens, an operator in a remote monitoring room is sent to clean up the mess. They have asked you group to implement an "Annunciator" system to monitor the holding tank and report its status to the operator in the monitoring room. The system has two objectives, to notify the operator when the tank is near full (so they can manually turn off the fill valve), then to notify the operator when the tank has over flown (so they can be sent to clean it up).

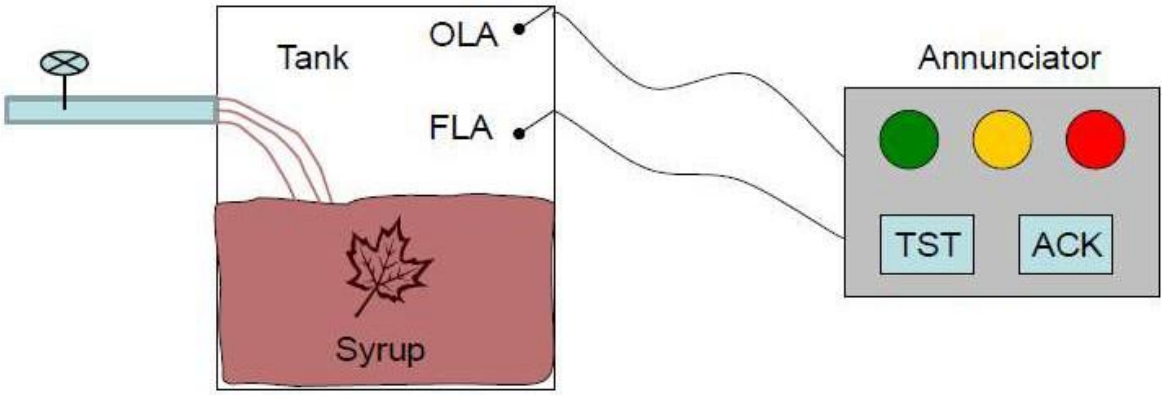


Figure 6 – Annunciator System

There are 4 input to the system. There are two level switches in the tank, full level alarm (FLA) and overflow level alarm (OLA). On the Annunciator box in the control room, there are two momentary push buttons, acknowledge (ACK) and test (TST).

On the Annunciator box, there are 3 outputs from the system, a green ok indicator, a yellow full indicator and a red overflow indicator. The system has 6 states as shown in Figure 7.

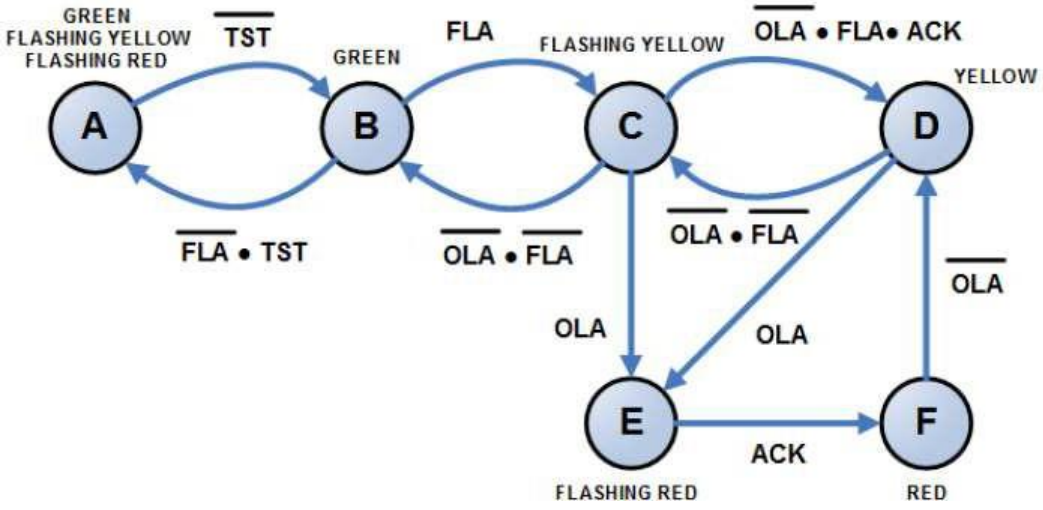


Figure 7 – State Diagram

Required Equipment and Parts:

- **Breadboard**
- **2 Pushbuttons**
- **2 DPI switches**
- **1 Red LED**
- **1 Yellow LED**
- **1 Green LED**
- **3 Resistors (300Ω < R < 1000Ω)**
- **3 Resistors (200Ω)**
- **Jumper wires**

Introduction:

This lab is designed to introduce students to operation of state machines. State machine has finite number of states which it can be in and changes these states depending on the predefined conditions.

During this lab students must utilize their knowledge of GPIO obtained from lab 2 in order to successfully complete this assignment. Some modifications must be made to the assembly files, however most of the work will be done in C++, so a strong knowledge of switch statements and loops is imperative.

Pin selection:

Student must carefully pick pins with which to work. Below is a E310 Pinout table that specifies which GPIO offset corresponds to which pin number, also IO functions and LEDs are specified. Student must pick a pin that does not have any IOF_0 or LED associated with it. Good choices would be Pin 2, 7, 8, 9, 17, 18, 19.

HiFive1 Pin Number	GPIO Offset	IOF0	IOF1	LED
0	16	UART0:RX		
1	17	UART0:TX		
2	18			
3	19		PWM1_1	GREEN
4	20		PWM1_0	
5	21		PWM1_2	BLUE
6	22		PWM1_3	RED
7	23			
8	0		PWM0_0	
9	1		PWM0_1	
10	2	SPI1:SS0	PWM0_2	
11	3	PI1:SD0/MOSI	PWM0_3	
12	4	SPI1:SD1/MISO		
13	5	SPI1:SCK		
14		*Not Connected*		
15	9	SPI1:SS2		
16	10	SPI1:SS3	PWM2_0	
17	11		PWM2_1	
18	12		PWM2_2	
19	13		PWM2_3	

E310 Pinout

(from "HiFive1 getting started v1.0.2"

Bitfield (GPIO Offset)

Bitfield or GPIO Offset is a bit number in 32-bit sequence that corresponds to a pin. For example, student wants to enable input for pin 2. First, the student needs to figure out the GPIO offset of pin 2 which is 18 in this case. The binary 32-bit number with 18th offset bit enabled is below.
0000 0000 0000 0100 0000 0000 0000 0000 which equals to 0x00040000 in hexadecimal(0x4000).
The hexadecimal number is called a "mask" for pin 2. Now when mask is acquired it can be stored to corresponding memory location which activates the input. In E310's case its base GPIO address of 0x10012000 plus *input_en* offset of 0x04 which gives memory location of 0x10012004. The mask for pin 2 can be stored into *input_en* offset by the following assembly commands:

```
Li t0, 0x10012000     ← load GPIO base address into t0
Li t1, 0x04           ← load input_en offset to t1
Li t2, 0x4000         ← load pin 2 mask to t2
Sw t2, t1(t0)         ← store t2 into the address provided by t0 plus the offset of t1
```

Offset	Name	Description
0x00	input_val	Pin value
0x04	input_en	Pin input enable*
0x08	output_en	Pin output enable*
0x0C	output_val	Output value
0x10	pue	Internal pull-up enable*
0x14	ds	Pin drive strength
0x18	rise_ie	Rise interrupt enable
0x1C	rise_ip	Rise interrupt pending
0x20	fall_ie	Fall interrupt enable
0x24	fall_ip	Fall interrupt pending
0x28	high_ie	High interrupt enable
0x2C	high_ip	High interrupt pending
0x30	low_ie	Low interrupt enable
0x34	low_ip	Low interrupt pending
0x40	out_xor	Output XOR (invert)

E310 GPIO Memory Map

(from SiFive FE310-G002 Manual)

Getting input from pin 2 is a similar procedure. First you load the value stored at *input_val* offset into a register. In order to determine if pin 2 is in the high state student must AND the mask and newly stored *input_val*, when the result of the AND operation is 0x0 the pin is low, otherwise its high.

Often it is desirable to enable input or output to many pins at the same time. In that case simple additions of all the masks can be done, since no 1s overlap the result will be a bitmap that covers all the desired pins.

Procedure:

Use the same file setup as in Lab 2 Part 2 (test before proceeding)

1. Select the GPIO's to be used for the inputs and outputs. Use recommended pins. Modify *setupGPIO.S* to enable selected pins for input or output. In *setupGPIO.S* delete **sw t1, GPIO_OUTPUT_XOR(t0)**

2. On the solderless breadboard, wire 2 pushbuttons and 2 of the DIP stitches to the 4 GPIO's selected as the inputs in a pull-up resistor configuration with one side of the switch to the GPIO and the other side to ground.

3. On the solderless breadboard, wire the 3 outputs to the anodes of the red yellow and green LED's. Tie the cathodes of the LED to ground though 200 ohm resistors.

4. Modify code of *checkBot.S* to accept mask for a pin (ex. int checkBot(int PIN)). The parameter will be passed onto the register a0. Before proceeding any further, test this new functionality.

5. In *setLED.S,* change the first line after ledOn: to **or t1, t1, a0.**

6. Add code to *main.c* to handle the states of the Annunciator state machine. When returning value from checkBot(int PIN) do not forget to invert it, since the pullup function is on, pins are in active low state which is not intuitive. (ex TST = !checkBot(PIN_2);)

7. Test and debug the code

8. After fully testing the program, demonstrate it to the course instructor for credit.

3.4 Lab 4 – UART Serial Port

Lab Objective

- To learn how to setup and operate MCU serial ports
- To create functions for serial port initialization and utilization
- To learn how to use oscilloscope to observe serial waveform

Background

Serial communications is a fundamental principal for microprocessor systems. In serial communications, data is transferred sequentially bit-by-bit along a channel in contrast to parallel communications where multiple bits are sent simultaneously over multiple channels. In modern digital systems, there are various protocols which employ serial transmission techniques that are aimed at a variety of applications. Some common examples are: USB (Universal Serial Bus) which is commonly used to interface peripherals to computers; SATA (Serial ATA), which is used to interface storage devices in computers; and Ethernet, which is used for computer networks. Other examples of serial buses are I2C and SPI (Serial Peripheral Interface) buses which are commonly found in embedded processor systems as interfaces busses for memories, DAC's and ADC, etc and CAN Bus (Controller Area Network) which is used to interface various systems in vehicles.

One of the simplest implementations of serial communications is the asynchronous serial port. Historically these were common on personal computers for uses such as interfacing to external modems, peripherals such as mice and computer terminals. On PC's, these serial ports used RS-232 complaint signaling and DB-25 or DE-9 connectors. RS-232 specifies the electrical characteristics of the signals. In the last decade, serial ports on PC's have become rarer features as USB has replaced most of the consumer applications that were previously handled by serial ports.

In embedded systems and industrial controls, asynchronous serial communications is still very common and useful. In one common embedded application, asynchronous serial ports are used for debug console interfaces. Most microcontrollers feature UART (Universal Asynchronous Receiver Transmitters) peripherals internal to the microcontroller. A common implementation would be to connect the microcontroller UART to an RS-232 converter IC (integrated circuit) on the embedded system which would then interface to the RS-232 serial port on a PC.

In newer PC's which do not have built in RS-232 serial ports, a USB-to-RS-232 converter would be used. These converters usually have a DE-9 connector, RS-232 converter IC and a serial-to-USB converter IC with a USB cable to interface to the PC. These serial-to-USB converter IC's are available from a variety of manufactures (Prolific and FTDI are very common). The use of these IC's required a driver to be installed on the PC but typically do not required any custom firmware to use the IC.
In newer embedded systems (in the Arduino for instance), the RS-232 interface is completely removed, and the serial-to-USB IC is directly put on the embedded board. This allows the embedded system to directly connect to a PC without the use of a USB-to-RS-232 converter.

The HiFive uses the same approach as Arduino, however it has a secondary processor as the serial to USB interface. This secondary processor also serves as a programing and debugging interface. It supports both external and internal debugging features.

Then using serial communications, both the transmitter and receiver must use a similar clock rate that are synchronized in some fashion so the receiver can sample and decide if a bit is high or low. In some

serial communications schemes, a clock is sent in parallel with the data to align the transmitter and receiver. In other schemes, the receiver does clock recovery, where it generates a local clock that is aligned to the transitions in the data pattern to provide a sampling clock. The UART uses asynchronous sampling to align the receiver to the transmit stream.
For asynchronous sampling to work, both the transmitter and receiver must be pre-configured to share the same data rate and format. When no data is being sent, the transmitter idles at a fixed level, high in the case of traditional UARTs. When a data byte is to be transmitted, the transmitter starts will a "start" bit, which is always a low. The transmitter then follows with the data bits, which are usually sent LSB (least significant bit) first, high is a "1" and low is a "0". The transmitter closes the transmission with a "stop" bit which is always a high.

When the receiver sees the transition from high (idle) to low (start bit), it knows a data byte is coming. It starts sampling the subsequent bits roughly 1.5 bit periods after the beginning of the start bit. It samples at the bit period for the number of data bits it has been configured for. The presence of the stop bit forces the line to go high so the receiver can observe the next high to low transition. Because the receiver re- synchronizes its sampling after each transmitted byte, differences of up to about +/-5% are possible in the transmit and receive clocks.

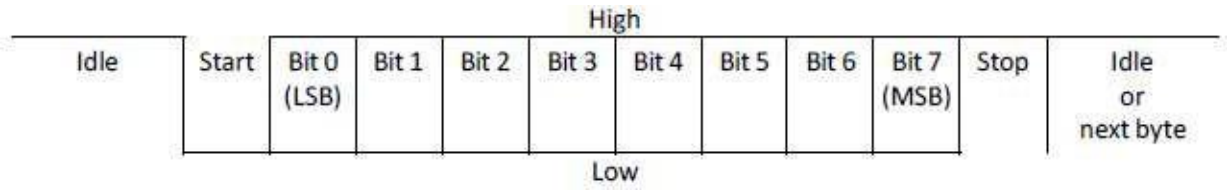


Figure 8 – Asynchronous Serial Transmission

The processor on HiFive1 has two UARTs and both are connected to the secondary microcontroller so both can be used with USB, for simplicity UART0 will be used for this lab. The serial connection is wired to pins PIN0 (RX), PIN1 (TX). In order to use UART0, several memory registers must be altered to set the right flags. Setting up UART on HiFive1 is just a matter of enabling transmit and receive bit at of the UART memory location and the code for it is provided (reference the Freedom E310-G002 Manual for the memory addresses and offsets). Further modification like different baud rate, watermark level, one or two end bits can be accomplished, but are not necessary for this lab. After the UART is configured it can transmit and receive.

Required Equipment and Parts

- Tektronix DPO2012B Oscilloscope (in lab)
- Oscilloscope Probe (from stockroom)
- Tektronix DPO2COMP Computer Serial Module (from stockroom)
- Jumper wires

Procedure

3.4.1 Lab 4, Part1: Reading and writing characters for the UART

The objective of this part is to get familiar with initializing UART and implementing write and read function. HiFive1 UART is a very simple and straightforward system that need minimal set up, besides enabling read and write it does not need any further setup. This part will utilize UART in order to receive input from terminal and echo it back.

Starting point of this project is the setup file for UART, however before writing that a few standard files need to be added. *mamoryMap.inc* needs to be added to the UART project's source folder. The following files need to be created in the source folder: *main.c, UART.inc, setupUART.S, UART_read.S, UART_write.S.* Below is the code for some of these files.

```c
void setupUART();
unsigned char UART_read();
void UART_write(unsigned char a);

int main()
{
    setupUART(); // enable UART
    unsigned char output;

    while(1)
    {

    output = UART_read(); // get UART input
    UART_write(output); // output the received input through UART

    }

return 0;

}
```

main.c

```
.equ UART_txdata, 0x00 # transmit data register
.equ UART_rxdata, 0x04 # receive data register
.equ UART_txctrl, 0x08 # transmit control register
.equ UART_rxctrl, 0x0C # receive control register
.equ UART_ie, 0x10 # UART interrupt enable
.equ UART_ip, 0x14 # UART interrupt pending
.equ UART_div, 0x18 # baud rate divisor
```

UART.inc

```
.section .text
.align 2
.globl setupUART

#include "memoryMap.inc"
#include "UART.inc"

setupUART:
    addi sp, sp, -16
    # allocate a stack frame, moves the stack up by 16 bits
    sw ra, 12(sp)

    li t0, UART_CTRL_ADDR # load UART base address
    li t1, 0x01
    # 0s bit set to 1 for transmit/receive enable, 1st bit set to 0 for 1 stop bi
t
    sw t1, UART_txctrl(t0)
    # load desired parameters into the txctrl mem location
    sw t1, UART_rxctrl(t0)
    # load desired parameters into the rxctrl mem location
    lw ra, 12(sp) # return the return address
    addi sp, sp, 16 # deallocate the stack
    ret
```

setupUART.S

UART_read.S is provided below

```
.section .text
.align 2
.globl UART_read

#include "memoryMap.inc"
#include "UART.inc"

UART_read:
    addi sp, sp, -16
    # allocate a stack frame, moves the stack up by 16 bits
    sw ra, 12(sp)
    li t0, UART_CTRL_ADDR # load base address for UART
    loop:
    lw a0, UART_rxdata(t0) # load rxdata register
    li t1, 0x80000000 # create mask for rxdata to isolate empty bit
    and t3, a0, t1
    # AND the rxdata mask and register to see if empty bit is set
    bnez t3, loop # the data is empty, go back and check again
    # the data is full if the program goes past the branch
    # return will be accomplished via a1 register.
    # (it is the preset return register for RISC-V)
    lw ra, 12(sp)
    addi sp, sp, 16 # deallocate the stack
    ret
```

UART_read.S

Student must write *UART_write.S, b*elow is a template.

```
.section .text
.align 2
.globl UART_write

#include "memoryMap.inc"
#include "UART.inc"

UART_write:
    addi sp, sp, -16
    # allocate a stack frame, moves the stack up by 16 bits
    sw ra, 12(sp)

    # CODE HERE

    lw ra, 12(sp) # load the return address
    addi sp, sp, 16 # deallocate the stack
    ret
```

UART_write.S

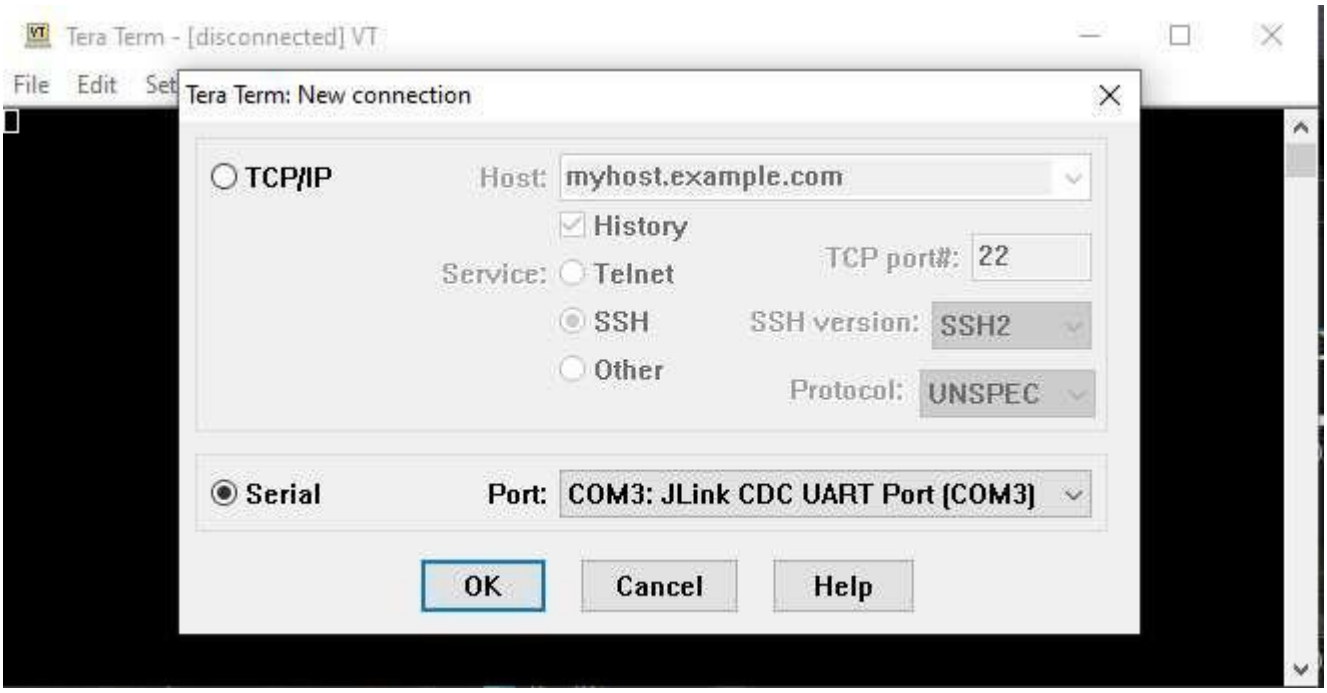
When C calls ***UART_read(),*** the function returns an 8 bit value in a form of an unsigned char, the return is accomplished through register a1. When C calls ***UART_write(a)*** it must pass an unsigned char in place of the 'a'. Inside the ***UART_write()*** function, the argument is received by a0.

The setupUART routine should be completely functional as provided. By default, it configures UART0 to use 8 bit data, 1 stop bit and no parity. Different baud rate can be set, but the board uses a pre-set baud rate of 115200, which is what will be used for this lab.

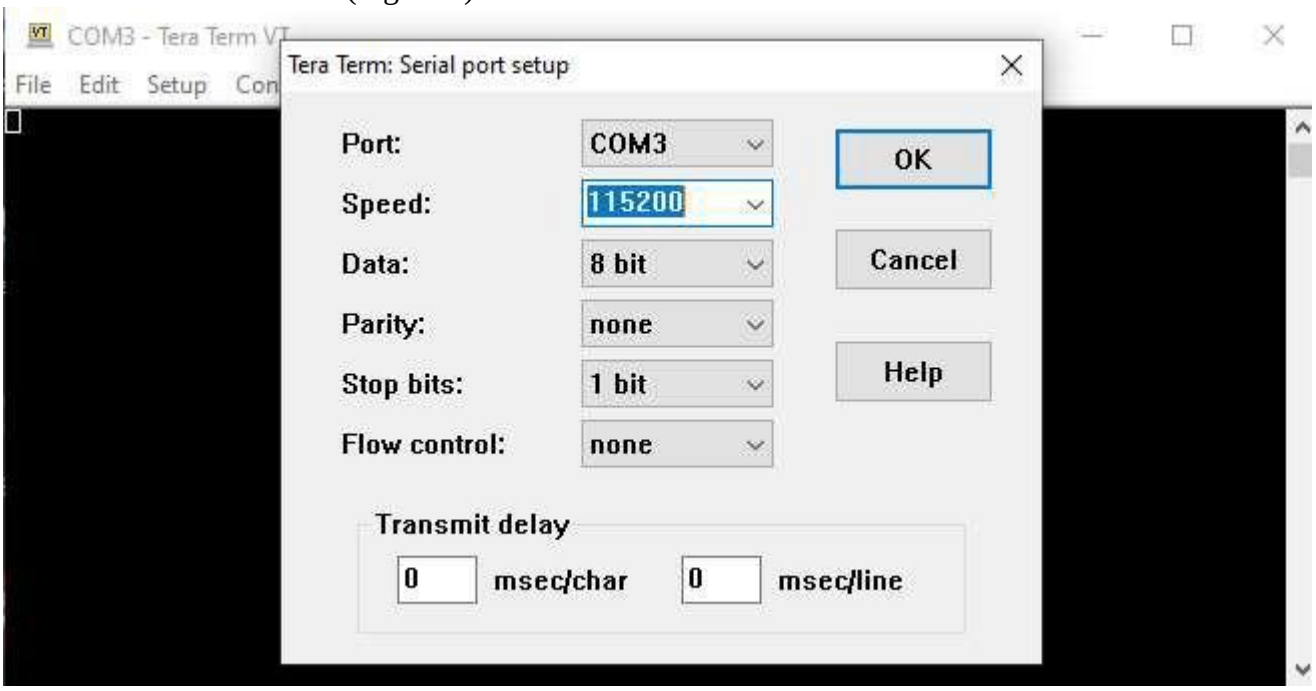
The main function is completely functional as is. It calls setupUART routine, initializes unsigned char variable and loops through read and write routines. The UART_read routine reads the RX data received from USB and returns it to the main and the data gets passed to the UART_write which in turn writes that data back to the TX and back to the USB.

After completing the UART_write routine connect the HiFive1 to the PC. Build, debug and then run the code onto the board as usual. Open Tera Term on the PC that is connected to the board (other terminal programs can be used but Tera Term is recommended).

When Tera Term starts, select New Connection, Serial, then the COM port labeled JLink CDC in the port pull-down (select the first UART port, they are not labeled so trial and error might be needed to determine which one is UART0).Then hit the OK button. (Figure 7)



Next select Setup → Serial port. In the Serial port setup dialog, set the baud rate, data, parity, stop and flow control to match the HiFive1 settings.
In the default case use:
Baud: 115200,
Data: 8 bit,
Parity: none,
Stop: 1 bit and
Flow control: none.
Then click the OK button. (Figure 8)



With the code running on the HiFive1 and Tera Term setup properly, anything typed on the Tera Term console will be echoed back and displayed in the console window. (Figure 9).

Stop the code on HiFive1 and enter a character in the Tera Term. **What happened and why?**

Add a call to UART_write() before and after the loop. **What happens when you add the call before the loop and why? What happens when you add the call after the loop and why?**