EE 277A: Embedded SoC Design

San Jose State University
Department of Electrical Engineering

Sai Kashyap Kurella

Date: 05/18/2023

Lab No: 10

1. INTRODUCTION

In this laboratory, we will develop a Snake game that involves two players. The user's input will be obtained from Tera term via UART, and the visual output will be presented using VGA. We will utilize the C programming language to program the System on Chip (SoC) at a higher level, taking advantage of compilers, software libraries, and tools to simplify low-level management complexities. The CMSIS abstraction layer will further aid in this process. This approach allows us to create more intricate applications efficiently in terms of time and effort. Additionally, we have the flexibility to combine C code with assembly code within the same project or even the same file. By using the C programming language, we can write programs for the Cortex-M0 CPU and effectively control the peripherals.

Our objective in this lab is to create a two-player snake game using a C program. The C program will incorporate UART and timer interrupt handlers to manage and display/detect the snake's coordinates. Additionally, we will keep track of scores and indicate which player is currently winning. Verilog code will be employed to write the CPU and bus interfaces, as well as the on-chip memory and peripheral hardware. These components will be modified and enhanced as necessary to ensure their proper functioning. The System on Chip (SoC) will incorporate interrupt mechanisms for the AHB timer and AHB UART, among other functionalities. Specifically, the software designed for the Cortex-M0 CPU will include interrupt servicing routines tailored for the Cortex-M0 CPU's requirements.

To achieve this, we will integrate the necessary interrupt registers and develop suitable interrupt handlers. This implementation will enable us to incorporate the AHB timer, VGA, and UART protocol mechanisms in both the hardware and software aspects. Throughout the lab, we utilized Vivado, Keil uVision, and Tera Term for our tasks.

2. BACKGROUND AND DESIGN SOLUTIONS

2.1.INTERRUPT HANDLING IN CORTEX -M0

The Cortex-M series processors feature the Nested Vector Interrupt Controller, which manages interrupts by prioritising and masking them. The NVIC has memory-mapped registers that can be programmed to manage interrupts, such as enabling or disabling them and setting priority levels. The priority levels are determined by 8-bit registers, but only the most significant bits are utilised. The Cortex-M0 and Cortex-M0+ processors have four programmable priority levels.

The Nested Vector Interrupt Controller (NVIC) handles nested interrupts automatically in the Cortex-M0 Microprocessor Architecture, as shown in Figure 1. During the execution of an Interrupt Service Routine (ISR), the NVIC prioritises interrupts and blocks out any same or lower priority interruptions that have been configured. If a higher priority interrupt is triggered, the running ISR is paused so that the higher priority ISR can be executed with minimal delay.

Whenever an interrupt is triggered, the processor employs a vector table to compute the beginning address of the corresponding Interrupt Service Routine (ISR). Typically, the vector table is located at the start of the memory space, but it can be relocated to a different address by user software or a bootloader. The vector table houses the reset vector address and the initial value for the Main Stack Point.

In Figure 2, we can observe the handling of nested interrupts, which are categorised into different levels of priority. A higher priority interrupt can occur and be dealt with while a lower priority interrupt is being serviced. This is often referred to as interrupt pre-emption or nested exceptions.

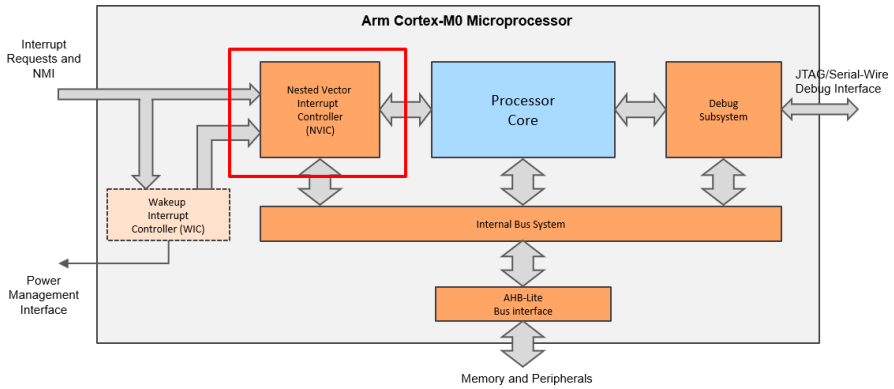

Figure 1. NVIC in Cortex-M0 Microprocessor.

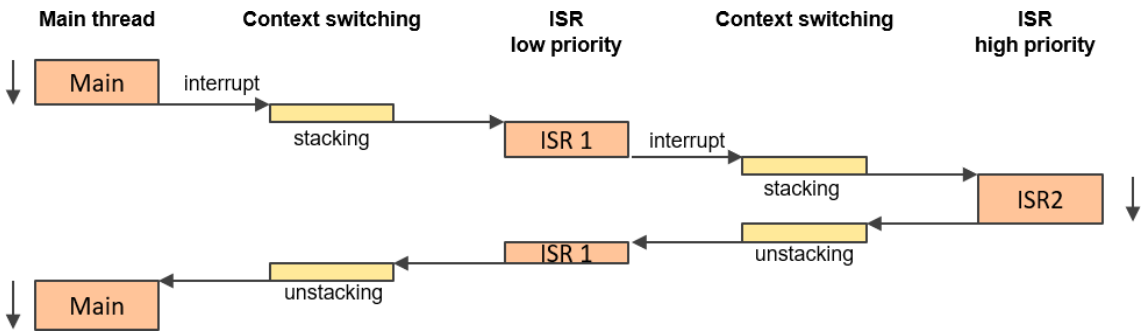

Figure 2. Nested Interrupt Handling

2.2.TIMER PERIPHERAL

The typical hardware timer architecture comprises a Prescaler, Timer Register, Compare Register, Comparator, and Capture Register. The Prescaler receives a clock signal and divides it by a predetermined factor (such as 4, 8, or 16) before outputting the resulting frequency to other components. The Timer Register increments or decrements at a set frequency, driven by the output from the Prescaler (known as ticks). The Compare Register is programmed with a target value that is periodically compared to the Timer Register value. If the two values match, an interrupt is triggered. The Capture Register stores the current Timer Register value at specific events and can also generate interrupts at these events. Table 1 lists the base address and size of the timer registers.

The various timer mode involves Compare, Capture and PWM. Figure 4 depicts the operations performed in Compare mode and Figure 5 depicts the operations performed in Capture mode.

Table 1: Timer Peripheral Registers

Register	Base address	Size
Load value	0x5300_0000	4 bytes
Current value	0x5300_0004	4 bytes
Control value	0x5300_0008	4 bytes
Clear register	0x5300_000C	4 bytes

In order to configure timer interrupt for it to act as counter, an interrupt is generated every time the counter reaches zero. A clear register needs to be added; this is used to clear the interrupt request once the processor finishes its ISR. The timer interrupt signal is depicted in Figure 3.

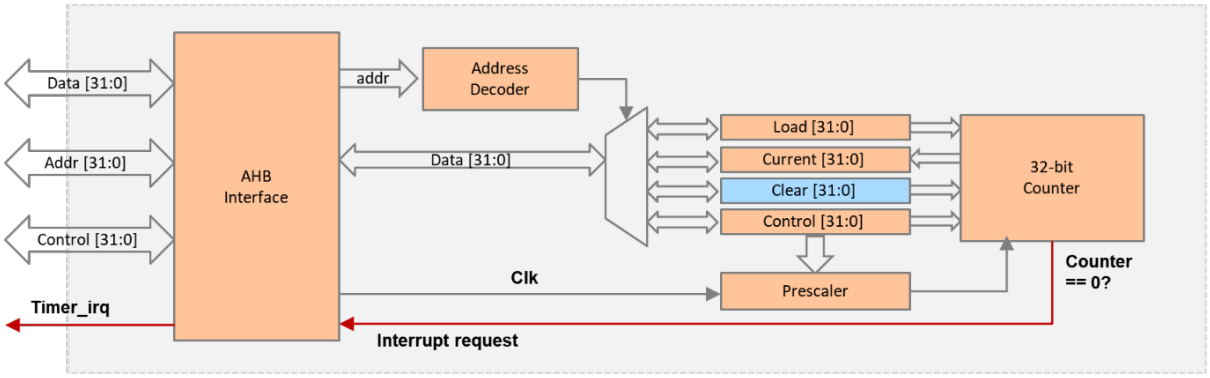

Figure 3: Timer Interrupt Signal

2.3.UART PERIPHERAL

A USB-UART bridge, also known as a USB-to-serial converter, is a device that converts the USB protocol to the UART protocol. It allows a device that only has a UART interface, like the NEXYS A7 board, to communicate with a computer that only has a USB interface.

The NEXYS A7 board features a built-in **USB-UART bridge**, which allows it to communicate with a computer over USB. The USB-UART bridge is implemented using a chip called the FT2232H, which is also used as a JTAG programming interface.

Teraterm is a terminal emulation program that allows you to communicate with devices over a serial interface. It can be used to send and receive data over a serial connection and can display the received data in various formats.

The Nexys A7 contains an FTDI FT2232HQ USB-UART bridge (connected to connector J6) that allows usage of Windows COM port commands to communicate with the board using PC software. USB packets are converted to UART/serial port data using free USB-COM port drivers, which can be found at www.ftdichip.com under the "Virtual Com Port" or VCP header.

A two-wire serial interface (TXD/RXD) and optional hardware flow control (RTS/CTS) are used to communicate with the FPGA. Following the installation of the drivers, I/O commands from the PC can be directed to the COM port to generate serial data traffic on the C4 and D4 FPGA pins. The ***transmit LED (LD20)*** and the ***receive LED (LD21)*** are two on-board status LEDs that provide visual feedback on traffic passing through the port (LD19). Signal designations that indicate direction are from the perspective of the DTE (Data Terminal Equipment), which in this case is the PC.

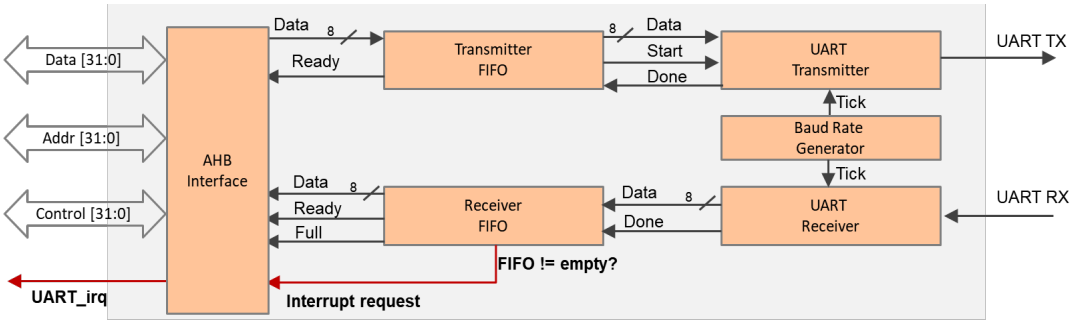

Figure 4. Interrupt enabled UART

2.4. CORTEX MICROCONTROLLER SOFTWARE INTERFACE STANDARD(CMSIS)

CMSIS is a vendor-neutral hardware abstraction layer created for the Cortex-M processor family. It delivers a standard software interface that contains library functions that facilitate the management of the CPU, such as the configuration of the nested vectored interrupt controller (NVIC). The primary objective of CMSIS is to enable the easy movement of software between various Cortex-M microcontrollers and serial processors.

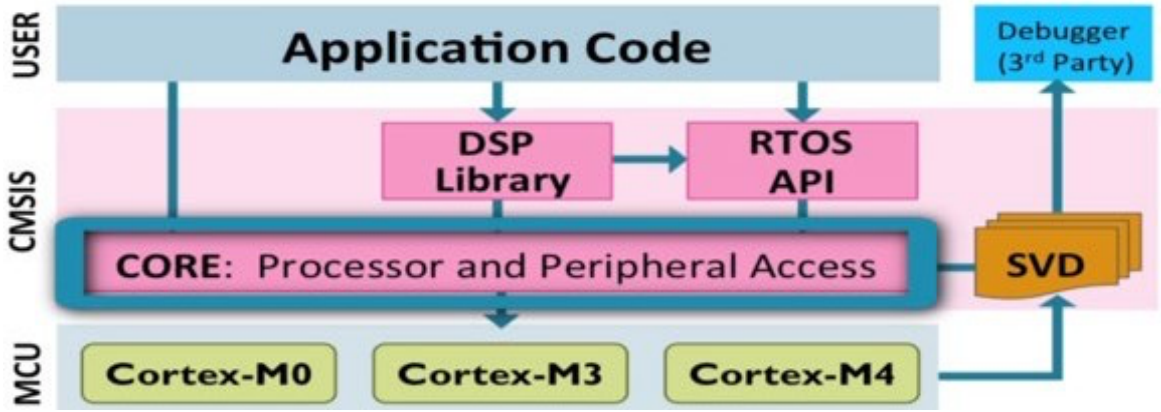

Figure 5. CMSIS Architecture

The structure of CMSIS is depicted in *Figure 5.* CMSIS-CORE is responsible for providing a processor interface and peripheral register for a range of Cortex-M processors, including Cortex-M0, Cortex-M3, Cortex-M4, SC000, and SC300. CMSIS-DSP includes more than 60 functions that support both fixed-point (fractional q7, q15, q31) and single precision floating-point (32-bit) implementation. The CMSIS-RTOS API serves as a standardised programming interface for managing threads, resources, and time in real-time operating systems. Programmer's view of an entire microcontroller system, which includes peripherals, is included in CMSIS-SVD files (System View Description). SVD offers visual design capabilities.

2.5. PROGRAMMING CORTEX-M0 USING CMSIS

There are a number of standardized functions included in the CMSIS; these mainly include access functions for peripherals, registers, and special instructions. The CMSIS includes functions for reading from or writing to core registers. We can also use CMSIS to execute special instructions. The table here lists some of the Cortex-M0 special instructions and their corresponding CMSIS intrinsic functions. CMSIS can also be used for system control and SysTick setup.

There are many standardized functions to access NVIC, system control block (SCB), and system tick timer (SysTick). For example:

- To enable an interrupt or exception use 'NVIC_EnableIRQ (IRQn_Type IRQn)'.
- To set pending status of interrupt us void 'NVIC_SetPendingIRQ (IRQn_Type IRQn)'.

For standardized access of special registers, the following functions can be used:

- Read PRIMASK register: uint32_t get_PRIMASK (void)
- Set CONTROL register: void set_CONTROL (uint32_t value)

For standardized functions to access special instructions following functions can be used:

- System initialization: void System Init(void)

Figure 6 depicts the CMSIS File and Folder structure. The CMSIS core standard consists of the device startup, system C code, and a device header. The device header defines the device peripheral registers and pulls in the CMSIS header files. The CMSIS header files contain all of the CMSIS core functions.

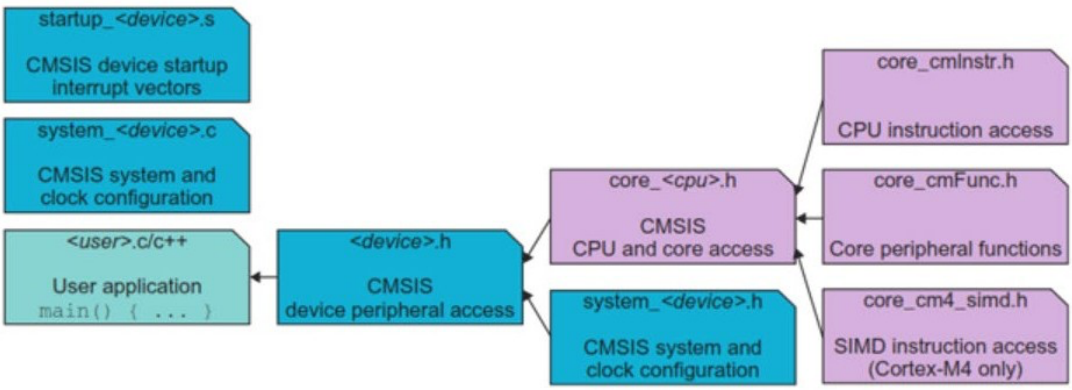

Figure 6. CMSIS File Structure

3. IMPLEMENTATION AND WORKING

This lab involves several tasks that require modification of a C program. First, the processor and NVIC must be initialised using CMSIS functions. Next, interrupts must be enabled in the program using CMSIS functions. The timer should be initialised using timer functions. Additionally, the CMSIS-based timer handler driver function must be modified to perform tasks such as clearing the interrupt request, incrementing a counter, and updating the seven-segment displays. Finally, the CMSIS-based UART handler driver function should be modified to allow reading and writing from/to the UART.

For this, along with the FPGA Verilog files from the previous lab, the software files listed in Table 2 are used.

File name	Description
core_cm0.h	CMSIS Cortex-M0 core peripheral access layer header file
core_cmFunc.h	CMSIS Cortex-M core function access header file
core_cmInstr.h	CMSIS Cortex-M core instruction access header file
cm0dsasm.s	Includes interrupt vectors and other setup assembly code
main.c	Includes the main program and interrupt service routines
EDK_CM0	Defines the interrupt numbers and memory map etc.

Table 2: Software Files used

After performing the restructuring of Keil project as mentioned in the Lab9.docx document and setting the path, the folder structure is as shown in figure 7.

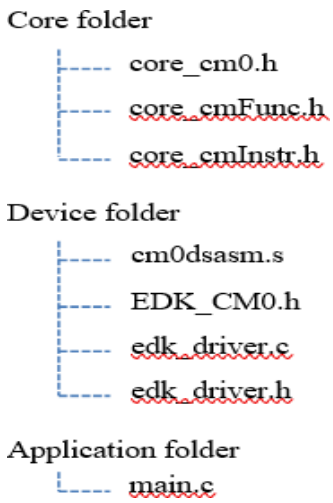

Figure 7: File structure in Keil Project

After performing the restructuring of Keil project as mentioned in the Lab9.docx document and setting the path, the folder structure is as shown in figure 7.The assembly code will initialize the interrupt vector, define heap and stack, define Reset handler internal interrupt that branch to the main code in main.c, define Timer handler interrupt that performs- pushing of registers (e.g., R1 – R4) to the stack, branching to the timer interrupt service routine in main.c, popping of registers from the stack, define UART handler that performs- pushing of registers (e.g., R1 – R4) to the stack, branching to the UART interrupt service routine in main.c, popping of registers from the stack. The C code in main.c performs the following tasks:

- *Main program*
    - Initialization if the processor and the nested vectored interrupt controller (NVIC) using CMSIS functions.
    - Enabling of the interrupts using CMSIS functions: NVIC_EnableIRQ(IRQ_Type).
    - Initialization and starting of the timer using timer driver functions: void timer_init(int load_value, int prescale, int mode), void timer_enable(void).
    - Then the following tasks are repeated:
        - Reading the value from the switches using GPIO functions: int GPIO_read(void).
        - Writing the value to the LEDs using GPIO functions: void GPIO_write(int data).
- Timer interrupt handler
    - Clearing of timer interrupt request using timer driver function: void timer_irq_clear(void).
    - Incrementing the counter.
    - Displaying the counter to the 7-segment in decimals using driver functions: void seven_seg_write(char dig1, char dig2,char dig3,char dig4).

- UART interrupt handler
    - Reading from the UART (from the keyboard) using UART driver functions

3.1. GAME IMPLEMENTATION

Below is the block diagram of the project implementation. We have used a given Verilog file and synthesized the design. The Verilog file has VGA, UART, GPIO, Seven Segment Display and Interrupt code in it. Then we have a code.hex file which is generated using the C programming language. Then we have put the code.hex file into the Vivado directory. Then we are giving the inputs using Teraterm and checking the game on Monitor using VGA Protocol. Using the provided single player snake game as reference, a replica of snake was made and scoring mechanism was manipulated accordingly to accommodate a two-player game.

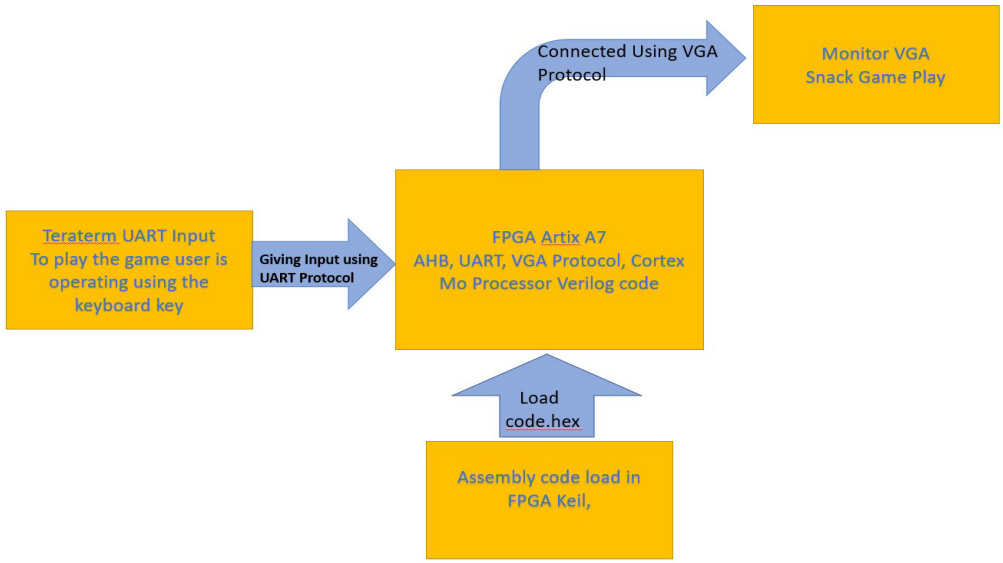

Figure 8. Basic block diagram explaining game operation

4. RESULT

The video of output is included in the Lab10.zip folder. Below are the screenshots of the outputs. Figure 9, Figure10 and Figure 11 depict different scoring scenarios of the game. Few other conditions like ***Snake killing itself by touching its own tail*** is missing in the figures below, since it was very difficult to capture that specific condition.

Figure 9. Player-1 winning situation where player-2 hits the boundary.

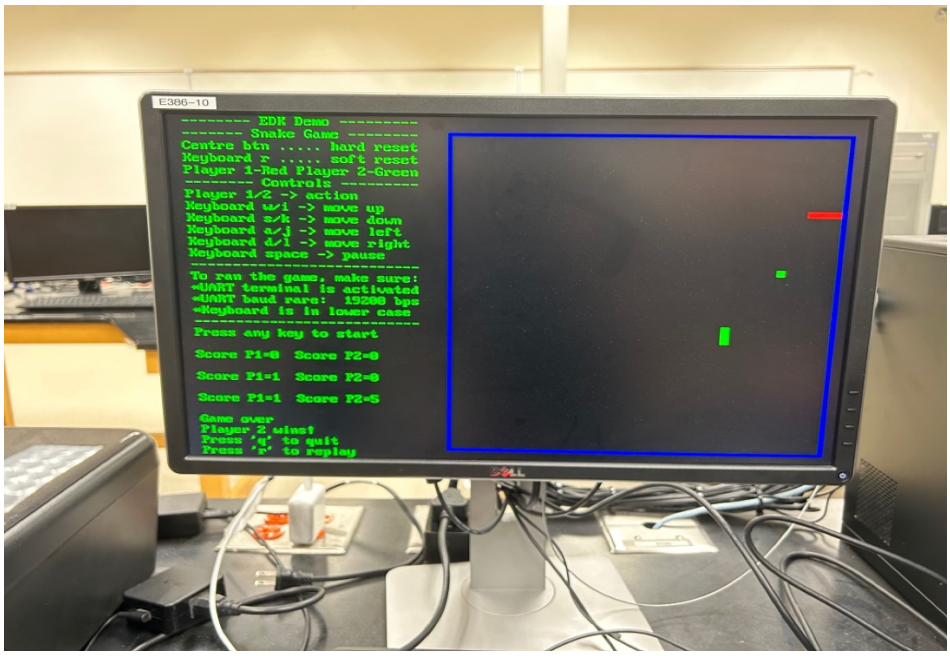

Figure 10. Player-2 winning situation where Player-1 hits the boundary.

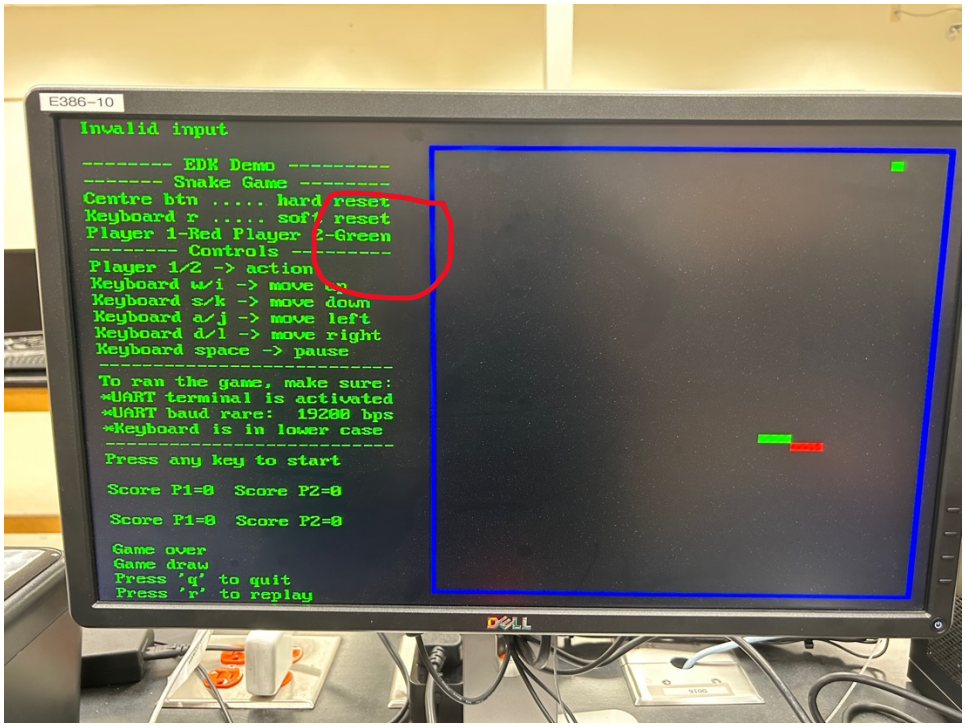

Figure 11. Tie situation where both the snakes collide each other.

5. CONCLUSION

Overall, in this lab, we have implemented a two-player snake game. The color of one snake is red, and the other is green. The score record is maintained for both the snakes. One point is added to the score when corresponding snake eats food (glowing pixel). On the death of any of the snakes, points should be added to the score of opponent snake. If snakes die from a head-on collision, no points should be added to either's score. However, if one snake dies by colliding with another, then points need to be added to the opponent player's score. If any snake dies by colliding with the wall, then points need to be added to the opponent snake's score. The winning player should be displayed with the final score. C programming language was used to program the SoC at a higher level, which will allow compilers, software libraries, and tools to be used to mask the complexity of low-level management. This will be accomplished through the usage of the language. This enables us to create more intricate applications in a manner that is more time and labor efficient. It is always possible to combine C code with assembly code inside the same project, or even the same file. We used the C programming language to write programs for the Cortex-M0 CPU and to control the peripherals.