EE 277A EMBEDDED SoC DESIGN

DEPARTMENT OF ELECTRICAL ENGINEERING

Lab 10: API and Final Application

Kunal Sawant

(015948764)

1. Introduction

In this laboratory, our objective is to develop a snake game application. We will utilize various high-level and low-level software drivers to implement the game for both single and double players on the ARM cortex-M0 CPU. To establish a connection between the high-level software and the low-level hardware, we will utilize APIs. Additionally, we will create an API that leverages the capabilities of software drivers and CMSIS to facilitate the development of more versatile and user-friendly applications. As a final step, we will showcase the Snake game on the SoC by creating a complete application. Moreover, we will incorporate the sleep mode option to minimize power consumption. Overall, this project will provide us with valuable knowledge in creating low-level hardware using 'Verilog' and developing high-level software applications in 'C' that can be executed on the SoC.

Figure 1 depicts the Architecture of a System on Chip.

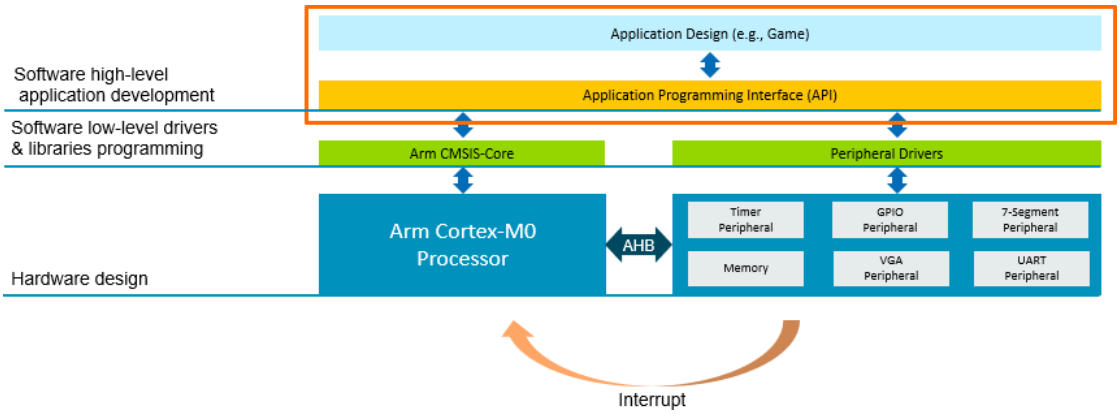

Figure 1 SoC Architecture

In this laboratory, our primary focus is on developing drivers for all three levels illustrated in Figure 1. To begin, we will design a high-level software program specifically for a snake game. Following that, our next task is to construct an API that facilitates communication between the ARM CMSIS-core and peripheral drivers utilized in the game application. Additionally, we will create a low-level software driver responsible for establishing communication between our core application and the hardware design, which in this case is the ARM Cortex M0 CPU. The ARM-provided IP core will be utilized for the hardware design of the ARM Cortex M0 CPU. Furthermore, we will develop hardware modules in Verilog to interface with various peripherals available on the SoC, such as VGA, memory access block, UART block, and more. By the end of this lab, we will have gained comprehensive knowledge and understanding of the complete application design process, encompassing the creation of higher-level software in languages like C and assembly, as well as the development of lower-level processor hardware using Verilog.

2. Verilog files used for hardware designing

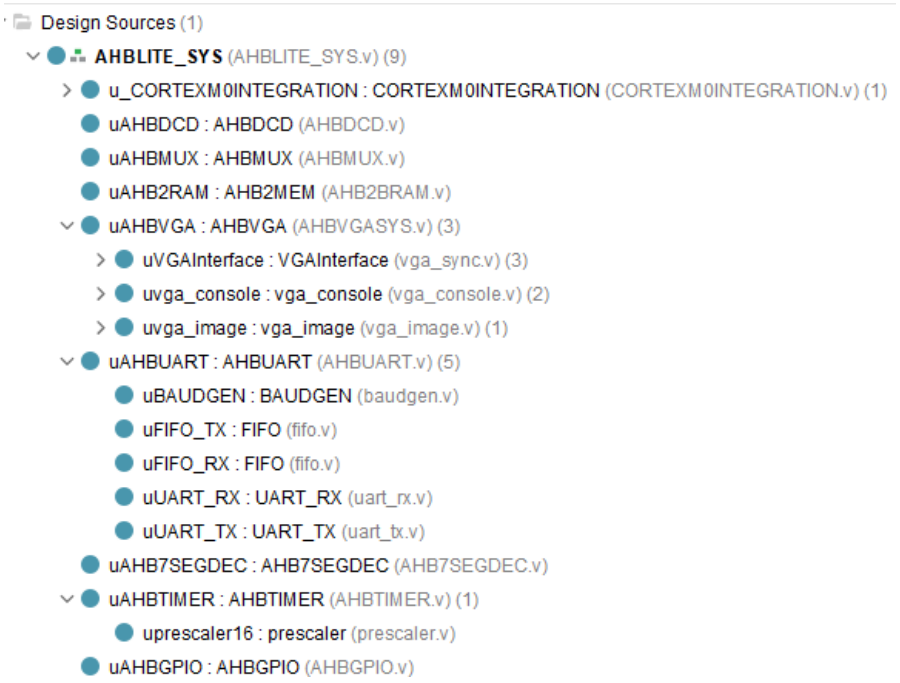

Figure 2 Verilog files

Figure 2 depicts all the files that were used for the hardware code implementation.

2.1 Software Files used in the application

Below is the group structure in Keil

Core folder

- 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

Device folder

- cm0dsasm.s
- EDK_CM0.h: used to specify Interrupt Number Definition
- edk_driver.c: functions definition for VGA, timer, 7 segments, GPIO peripherals
- edk_driver.h: Peripheral driver header file
- edk_api.c: Application Programming Interface (API) functions
- edk_api.h: initialization of parameters and functions used for VGA, UART, rectangle
- etc. retarget.c: Retarget functions for ARM DS-5 Professional / Keil MDK, allows us to use print library functions

Application folder

- main.c : tasks performed game initialization settings, boundary hit condition, target generation, UART, Timer

3. Theoretical Understanding

3.1 Application Programming Interface (API)

An API serves as a software layer that acts as an intermediary between application developers and the underlying programming resources. Its purpose is to provide a standardized interface that simplifies the process of creating applications. Various operating systems have their own APIs, which enable programmers to develop apps more efficiently. These APIs offer a range of services, including core functions, graphical user interfaces, network services, and other interface-related functionalities. Commercial APIs like Java API, Windows API, Google AJAX APIs, and others are readily available in the market.

In the context of our project, we have developed a fundamental API specifically designed for building snake game applications. This API combines functionalities from CMSIS and peripheral drivers, resulting in a user-friendly and versatile set of tools for the end-user. For instance, we have implemented a SoC startup method within the API, which allows for the convenient resetting of both the CPU and peripherals.

3.2 CMSIS

The Cortex Microcontroller Software Interface Standard (CMSIS) is a hardware abstraction layer specifically designed for the Cortex-M series of processors. It is independent of any particular vendor and aims to provide a standardized software interface. This interface includes a collection of library functions that simplify the management of the CPU, such as configuring the nested vectored interrupt controller (NVIC). The primary objective of CMSIS is to enhance the portability of software across various Cortex-M serial processors and microcontrollers.

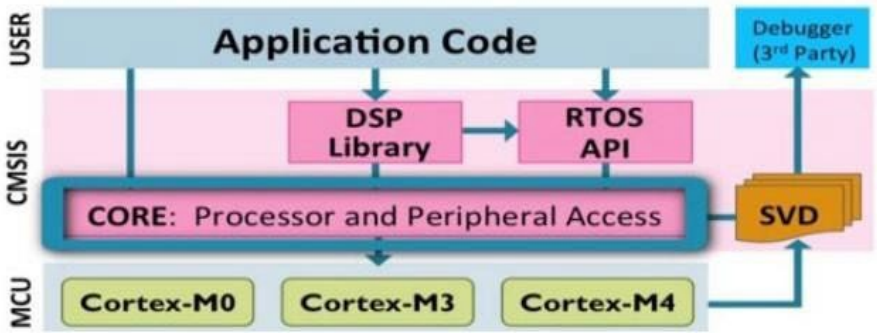

Figure 3 CMSIS Architecture

Figure 3 illustrates the structure of CMSIS, showcasing its different components. CMSIS-CORE serves as the peripheral register and processor interface for processors like Cortex-M0, Cortex-M3, Cortex-M4, SC000, and SC300. It encompasses over 60 functions, offering support for fixed-point (fractional q7, q15, q31) and single precision floating-point (32-bit) implementations through CMSIS-DSP. CMSIS-RTOS API, on the other hand, provides a standardized programming interface for tasks such as thread control, resource management, and time management in real-time operating systems. Finally, CMSIS-SVD files (System View Description) contain a programmer's view of the entire microcontroller system, including its peripherals. These SVD files facilitate visual design and understanding of the system.

3.2.1 Programming of Cortex M0 using CMSIS

CMSIS offers a variety of standardized functions that serve different purposes. These functions primarily include access functions for peripherals, registers, and special instructions. For instance, CMSIS provides functions for reading from and writing to core registers, as well as executing special instructions. The provided table showcases a selection of special instructions specific to the Cortex-M0 processor and their corresponding CMSIS intrinsic functions. Additionally, CMSIS can be utilized for system control and configuring the SysTick timer.

When it comes to accessing specific components like NVIC, system control block (SCB), and system tick timer (SysTick), CMSIS provides standardized functions. For example, to enable an interrupt or exception, the 'NVIC_EnableIRQ (IRQn_Type IRQn)' function can be used. Similarly, to set the pending status of an interrupt, the 'NVIC_SetPendingIRQ (IRQn_Type IRQn)' function is available. For accessing special registers in a standardized manner, CMSIS offers functions such as 'get_PRIMASK (void)' to read the PRIMASK register and 'set_CONTROL (uint32_t value)' to set the CONTROL register.

Furthermore, CMSIS includes standardized functions for accessing special instructions. Functions like 'REV(uint32_t int value)' for reversing the order of bits in a value and 'NOP(void)' for executing a no-operation instruction can be employed. To access system initialization functions with standardized names, the function 'SystemInit(void)' can be utilized.

Figure 4 provides an illustration of the CMSIS File and Folder structure. The core standard of CMSIS consists of the device startup, system C code, and a device header. The device header defines the peripheral registers for the specific device and includes the necessary CMSIS header files. These CMSIS header files encompass all the core functions of CMSIS.

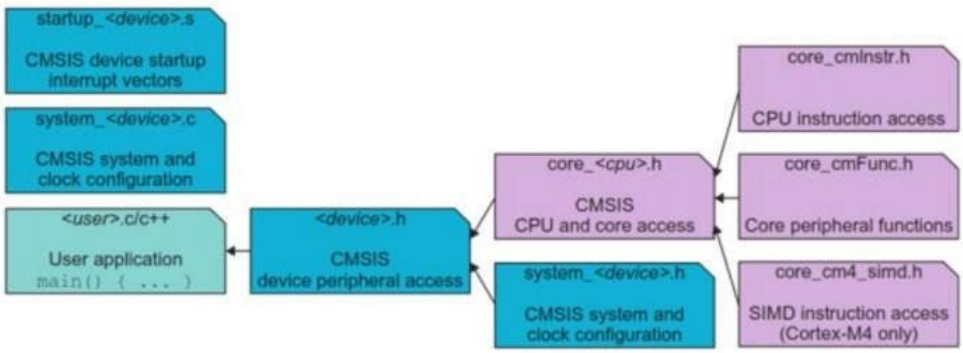

Figure 4 CMSIS File Structure

3.3 Interrupt Handling

The Cortex-M series processors feature the Nested Vector Interrupt Controller (NVIC) to manage interrupts, including prioritization and masking. The NVIC consists of programmable registers that handle interrupt-related tasks such as enabling or disabling interrupts and assigning priority levels. These registers are mapped to the memory space. The priority levels are determined by 8-bit width registers, although only the most significant bits (MSB) are implemented. In processors like Cortex-M0 and Cortex-M0+, there are four programmable priority levels available. The NVIC is responsible for managing nested interrupts automatically.

Figure 5 depicts the Cortex-M0 Microprocessor Architecture. During the execution of an Interrupt Service Routine (ISR), the NVIC takes charge of interrupt prioritization and masks out interruptions with the same or lower priority after setting the priority levels for each interrupt. If a higher priority interrupt occurs, the currently running ISR is preempted, allowing the higher priority ISR to be executed promptly. This ensures that higher priority interrupts are handled with minimal delay.

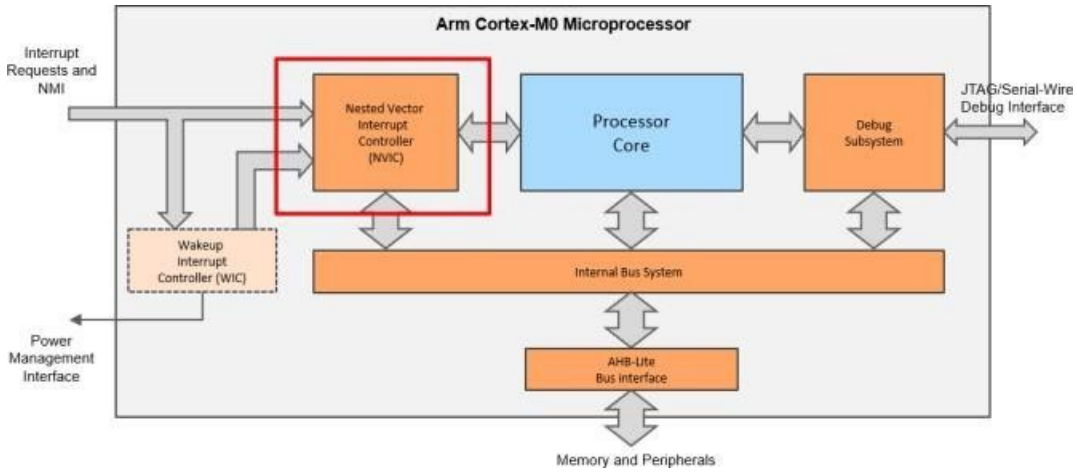

Figure 5 NVIC in Cortex-M0 Microprocessor

When an interrupt occurs, the processor uses a vector table to automatically calculate the ISR's start address. The vector table is initially positioned at the beginning of the memory space, but it can be moved by a bootloader or user software to a different address location. The reset vector address and the starting value for the Main Stack Point are stored in the vector table. Figure 6 depicts servicing of nested interrupts. The exceptions (or interrupts) are commonly divided into multiple levels of priorities. A higher priority exception can be triggered and serviced during a lower priority exception. This is commonly known as a nested exception, or interrupt preemption.

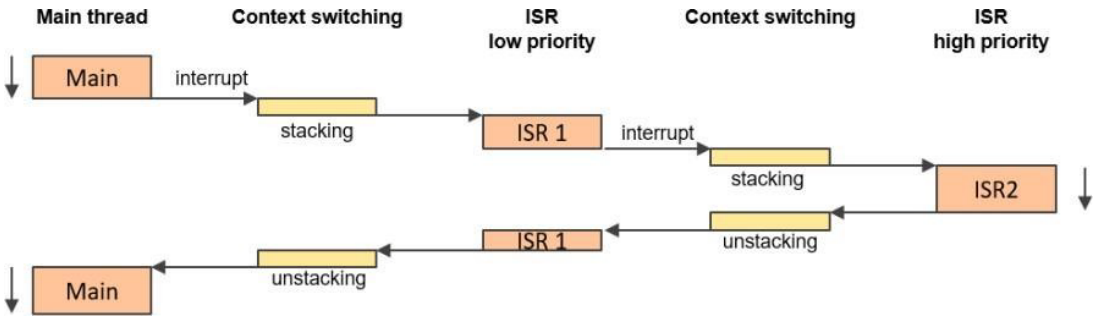

Figure 6 Nested Interrupt Handling

3.3 Timer peripheral

The standard architecture of HW timers includes a Prescaler, Timer Register, Compare Register, Comparator and Capture Register. The Prescaler takes the clock source as its input, divides the input frequency by a predefined value (e.g., 4, 8, 16) and outputs the divided frequency to the other components. The timer register increases or decreases at a fixed frequency and is driven by the output from the Prescaler; often referred to as ticks. The Compare register is preloaded with a desired value, which is periodically compared with the value in the timer register. If the two values are the same, an interrupt is generated. The Capture Register loads the current value from the timer register upon the occurrence of certain events and can also generate an interrupt upon the occurrence of certain events. Table 1 shows base address and size of the timer registers.

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

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 7.

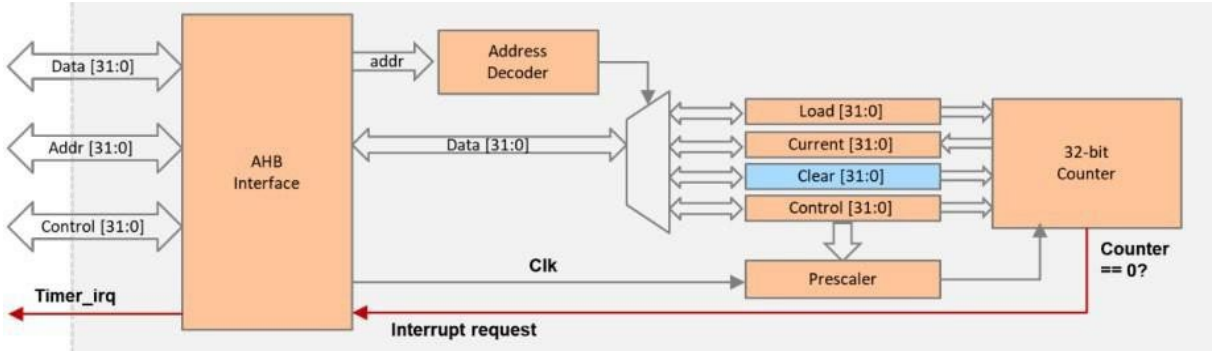

Figure 7 Timer control block

3.4 UART peripheral

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. To configure UART interrupt to send characters to a PC or laptop, mechanism to generate interrupt if the receiver FIFO is not empty can be implemented. Figure 8 shows the block diagram of UART.

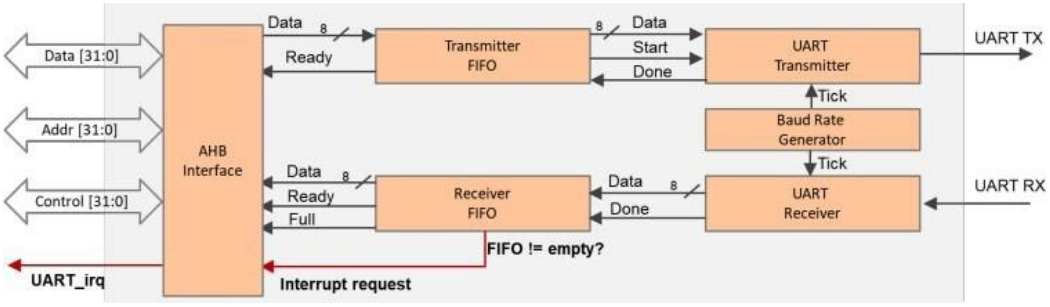

Figure 8 UART Interrupt Signal

4 Implementation

4.1 Assembly code

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.

4.2 The C code in main.c performs the following tasks:

Game_Init() handles all of the initialization for the rectangle boundary, score, snake speed, snake creation, and timer, as well as calling UART interrupts. We use Timer_ISR() to check whether the snake has hit itself, if it has touched the border, and to increase the score. We utilize UART to take commands from the user's keyboard and append snake movement in UART_ISR ().

4.3 Tasks for Single Player:

The figure 9 explains the flow chart of basic application flow of our code for single player snake game.

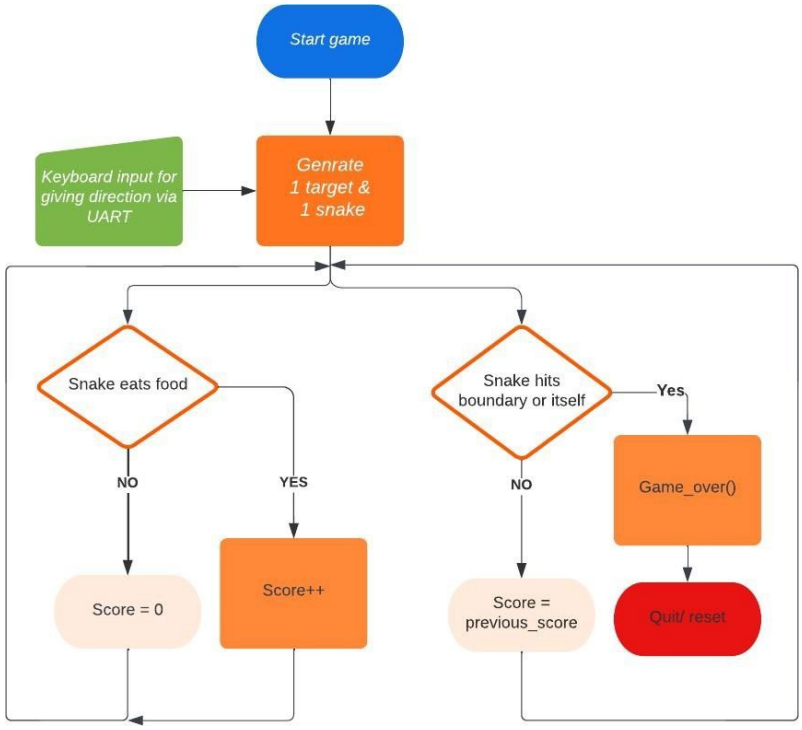

Figure 9 Flowchart of Single player snake application

Step 1: Generate Target

Target_gen() is called from Timer ISR() to produce a random target, with a constant check to see if the target's x and y coordinates coincide with the snake's body. If the target overlaps, the function is called repeatedly until a non-overlapping target is found.

Step 2: Draw the target

VGA_plot_pixel(target.x, target.y, WHITE) is used to draw the target on the picture console.

Step 3: Detect if the snake hits itself and end game

To see if the snake has hit itself, we check if the snake's head, which has coordinates (x=0, y=0), overlaps with any other snake coordinates other than (x,y) = (0,0). If overlap is discovered, i.e. the snake's head coordinates are equal to any other snake coordinates, we call the GameOver() function, which ends the game and allows the user to continue by hitting 'r' or quit by pressing 'q.'

Step 4: Detect if the snake hits the boundary

To identify the boundary hit situation, we created the API boundary_hit(), which has four arguments that represent the top, bottom, left, and right end points of each line of the boundary. We are testing if the snake's head is striking anywhere between the rectangle's coordinates in this API. If it succeeds, the value 1 will be returned. A value of 0 is returned if no hit is detected. If it returns 1 (ret GameOver = 1), we call GameOver(), and the user can press 'r' to continue playing or 'q' to exit the game.

4.4 Tasks for Double Player Snake Game:

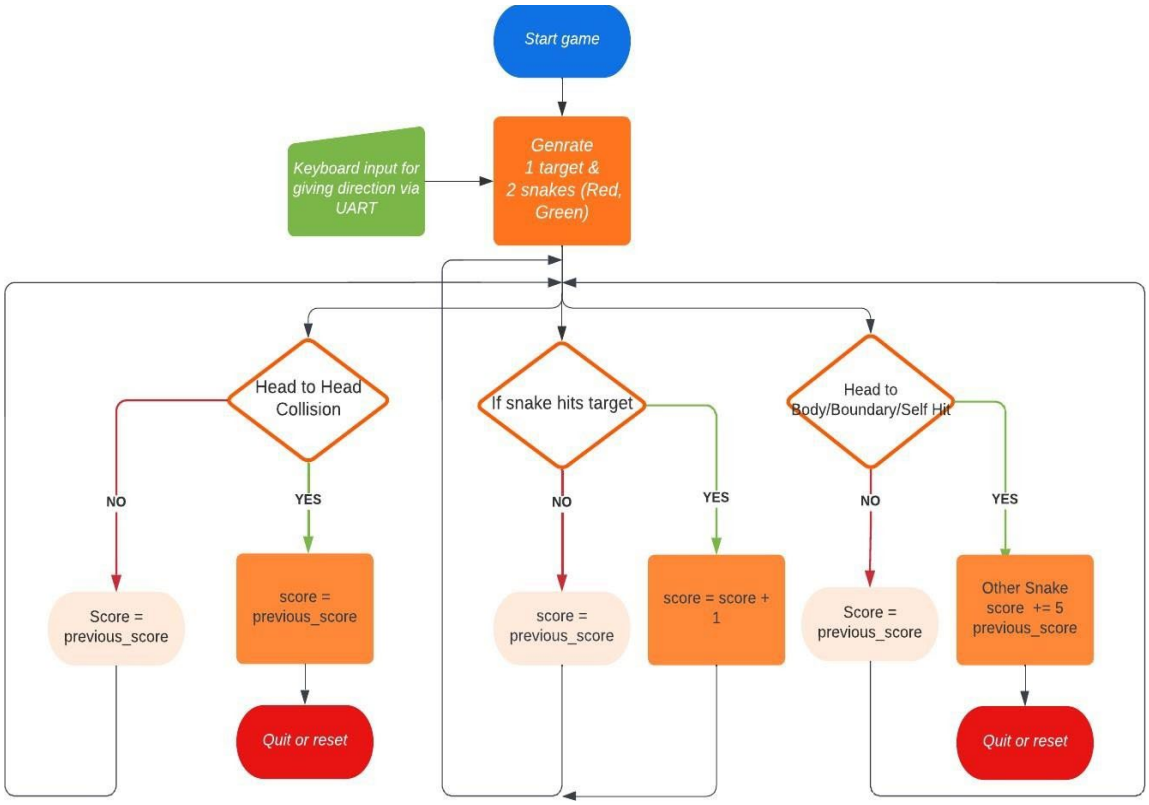

Figure 10 Flowchart of Double player snake application

Above figure 10, flowchart explains basic working flow of the code for double player snake game.

Step 1: Generate 2 Targets

target gen() is called from Timer ISR() to produce a random target, with a constant check to see if the target's x and y coordinates coincide with the snake1 (Red) and snake2 (Green) bodies. If the target overlaps, the function is called repeatedly until a non-overlapping target is found.

Step 2: Draw the target

VGA_plot_pixel(target.x, target.y, WHITE) function is used to draw the target on the picture console.

Step 3: Initialization of Parameters

To make a two-player game, construct two structural variables, snake1 and snake2, each with x and y coordinates, direction, and a node that represents the snake's length. Two global variables, score1 and score2, are defined to keep track of scores of both snakes and are initialized to 0 (the starting value).

Step 4: Control signals for snake_1 and snake_2

Both snakes start with a node size of 4, and snake1 is red in color, with start and end coordinates of (60,80), (62,80), respectively. Snake 2 is a green snake with (80,100), (82,100) as its start and finish coordinates. We use the control signals “W, S, A, D” from user’s keyboard to provide instructions to the red snake. We are using “I, J, K, L” from user’s keyboard input for snake green. The ascii values of these keys are defined as macros in the file edk_api.h to do this.

Step 5: When snake eats food

To see if the snake has struck the target, we keep checking to see if the target's x and y coordinates are overlapped by the snake's head. If snake 1 strikes the target, score_1 is increased by 1, and if snake 2 hits the target, score_2 is increased by 1. We must boost the pace of both snakes if one of them has hit the goal. We are doing this to see which score is the highest. The snakes' highest score determines the speed from the speed_table[] array.

Step 6: To detect if the snake has hit itself

We're seeing if the snake's head, which has coordinates of x=0, y=0, overlaps with any other snake’s coordinates other than (x,y) = (0,0). If an overlap is detected, GameOver() is called, and the user may either continue playing or leave the game by hitting 'r'. If one of the snakes strikes itself, the other snake receives an extra 5 points.

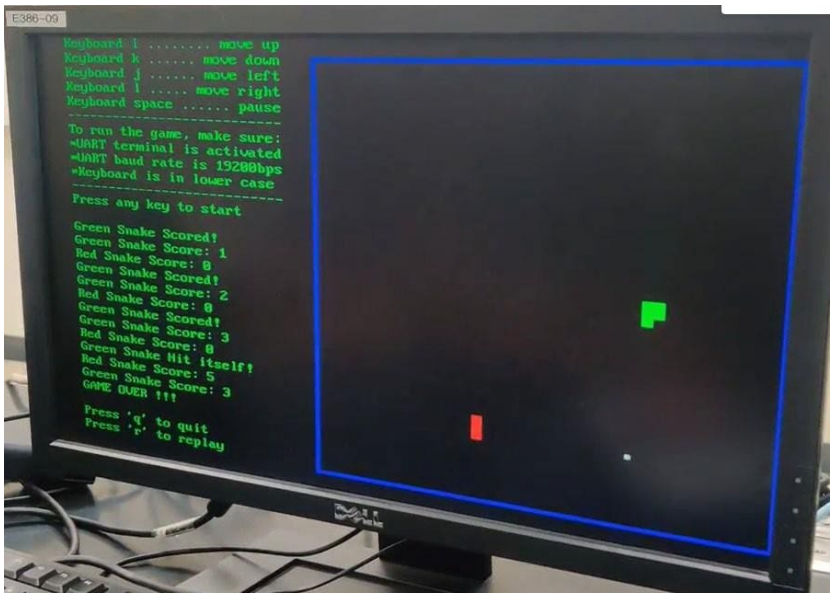

Figure 11 Self Body Hit (Green Snake) for two player snake game

As we can see in above photo figure 11, green snake hits itself and 5 points gets added in the score of red snake.

Step 7: To detect if the snake has hit boundary

To identify the boundary hit situation, we created the API boundary_hit(), which has four arguments that represent the top, bottom, left, and right end points of each line of the boundary. We're testing if the snake red and green head's is striking anywhere between the rectangle's coordinates in boundary_hit() api. If it succeeds, the value 1 will be returned. A value of 0 is returned if no hit is detected. If it returns 1 (ret GameOver = 1), we call GameOver(), and the user can press 'r' to continue playing or 'q' to exit the game. If one of the snakes touches the boundary, the other snake receives an extra 5 points this program is also added in boundary_hit() function. Following API's, we have used to check the boundary hit conditions:

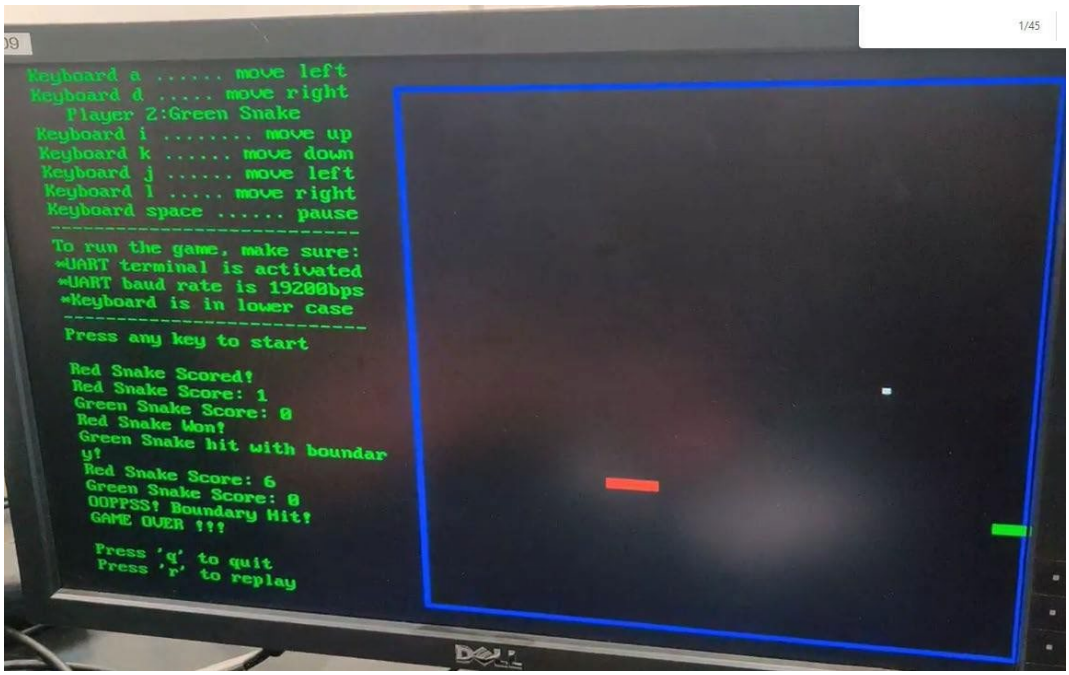

Figure 12 Boundary Hit (Green Snake) for two player Snake game

As we can see in above photo, in figure 12 green snake hits wall and 5 points get added in the score of red snake.

Step 8: Head-to-head collusion occurs

To check for head-to-head collisions, we constantly check if the coordinates of snake 1's head intersect with the coordinates of snake 2's head. If a head-to-head collision is detected, a message is presented, and GameOver() is called, with the user having the option to continue playing by hitting 'r' or to exit the game by pressing 'q'. No snake earns a point here.

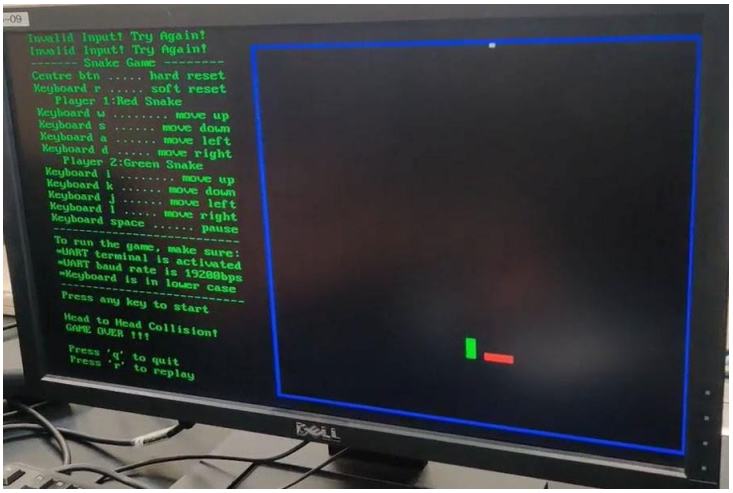

Figure 13 Head-Head Collision for Two Player Snake Game

As we can see in above photo, figure 13 green snake's head hit the head of red snake's head and no points gets added in the score of any snake.

Step 9: Detect head to body collusion of 2 snakes

We continuously check if the ordinates of one snake's head are striking the coordinates of another snake other than head coordinates to see if a snake has struck the body of another snake. If one of the snake's slams into the body of another, the other snake receives an extra 5 points. and we call GameOver(), where the user can hit 'r' to continue playing or 'q' to exit the game.

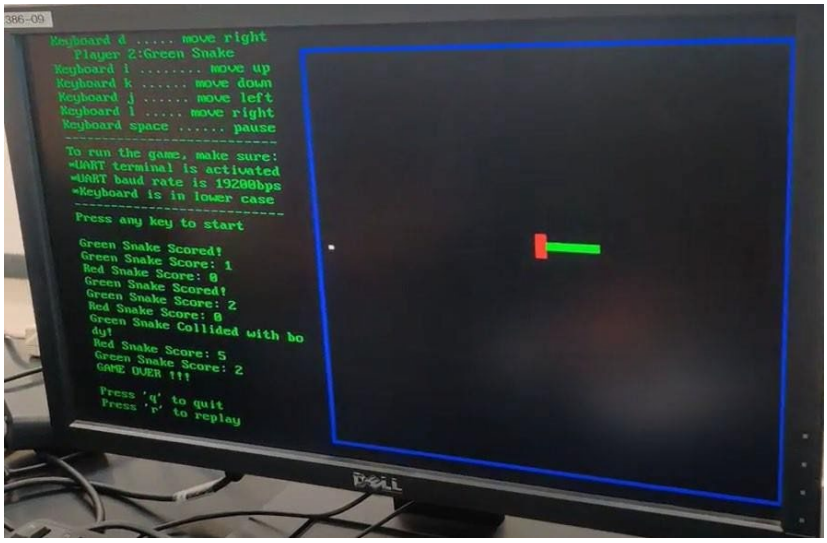

Figure 14 Body to Head Collision for two player Snake game

As we can see in above photo, figure 14 green snake hits body of red snake and 5 points get added in the score of red snakes.

5. RESULT

Result is uploaded as a video on canvas and for both single and two player snake game.

6. CONCLUSION

In this project, we employed higher-level programming techniques to develop a snake game application on the SoC (System-on-Chip) and its peripherals. We utilized CMSIS drivers and higher-level software drivers to accomplish this task. To detect specific conditions, such as when the snake hits its boundaries or body segments, and to handle the score increment and display the results on the VGA monitor, we utilized a timed interrupt. Additionally, we implemented a UART interrupt, which allowed us to control the snake's orientation by sending different characters from a PC or laptop to the SoC. Both interrupts were implemented using the driver functions provided by CMSIS. To execute the desired actions, we called Interrupt Service Routines (ISRs) written in embedded C code from assembly code.

Video Links

https://drive.google.com/file/d/1YlMjbqvv1p3DzVdPUEUiTIJoWaTwL_oD/view?usp=sharing

https://drive.google.com/file/d/15COV_5I3O-RWaf1HuFz1EaeomRYtRMyN/view?usp=sharing

https://drive.google.com/file/d/1vipWU3IMVR0TPxBq9XdvaVhuUY-DkJyy/view?usp=sharing