

# **EE 277A – EMBEDDED SoC DESIGN**

## **DEPARTMENT OF ELECTRICAL ENGINEERING**



### **API and Final Application for Snake Game**

**Under the guidance of**

**Dr. Shrikant Jadhav**

#### **Authors-**

Pinky Mathew (015949310)

Pushkar Deodhar (015266914)

Shreya Kulkarni (015219451)

#### **Submission date –**

05/23/2022

## Table of Contents

1. Introduction	4
2. Learning Objectives	5
3. Software & Hardware Used	5
3.1 Verilog files used for hardware designing	5
3.2 Software Files used in the application	6
4. Background	7
4.1 Application Programming Interface (API)	7
4.2 CMSIS	7
4.3 Interrupt Handling	9
4.4 Timer peripheral	10
4.5 UART peripheral	11
5. Implementation	12
5.1 Assembly code	12
5.2 The C code in main.c performs the following tasks:	12
5.3 Tasks for Single Player:	12
5.4 Tasks for Double Player Snake Game:	16
6. Power Consumption	21
6.1 Approach 1: Power optimization in Hardware design (using Verilog)	21
6.2 Approach 2: Power optimization using C programming:	26
7. CONCLUSION	27
8. RESULT	28
8.1 Link for single player:	28
8.2 Link for double player:	28

## List of Figures

Figure 1 SoC Architecture	4
Figure 2 Verilog files	5
Figure 3 keil files used in the application	6
Figure 4 CMSIS Architecture	7
Figure 5 CMSIS File Structure	8
Figure 6 NVIC in Cortex-M0 Microprocessor	9
Figure 7 Nested Interrupt Handling	10
Figure 8 Timer control block	11
Figure 9 UART Interrupt Signal	11
Figure 10 Flowchart of Single player snake application	13
Figure 11 Body Hit for Single Player Game	14
Figure 12 Boundary Hit for Single Player	15
Figure 13 Flowchart of Double player snake application	16
Figure 14 Self Body Hit (Green Snake) for two player snake game	18
Figure 15 Boundary Hit (Green Snake) for two player Snake game	19
Figure 16 Head-Head Collision for Two Player Snake Game	19
Figure 17 Body to Head Collision for two player Snake game	20
Figure 18 Clock gating reduces power consumption of our design	21
Figure 19 Program using if-else statements	22
Figure 20 Program using case statements	23
Figure 21 Schematic file after synthesis of design	24
Figure 22 Power optimization settings in Vivado 'pre-place'	24
Figure 23 Power optimization settings in Vivado 'post-place'	25
Figure 24 Power analysis before optimization	25
Figure 25 Power analysis after optimization	26
Figure 26 Task switching	27

## List of Tables

Table 1 Timer peripheral registers	10
Table 2 Boundary hit functions for single player	14
Table 3 Boundary hit functions for two player Snake game	18

# 1. Introduction

In this lab, we will create a snake game application which will be implemented using multiple high-level and low-level software drivers' application is for the single and double players implemented on ARM cortex-M0 CPU. Then, with the help of APIs, we'll build the essential communication interface between high-level software and low-level hardware. We'll also construct an API that takes advantage of the features provided by software drivers and CMSIS to enable more generic and user-friendly application development services. We'll make a final application to demonstrate the Snake game on the SoC. We'll also use the sleep mode option to reduce our application's power consumption. In summation, this project will teach us how to create low-level hardware in 'Verilog,' as well as high-level software applications in 'C,' which can be implemented on the SoC.

Figure 1 depicts the Architecture of a System on Chip.

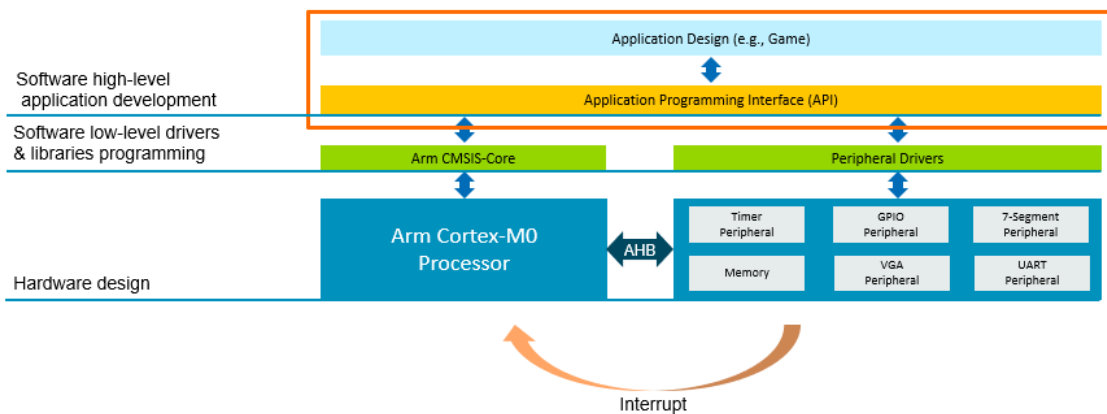


Figure 1 SoC Architecture

In this lab, we will create drivers for all three levels depicted in the figure 1 above. Firstly, we will develop a high-level snake gaming software program. Then we will create an API for communication between the ARM CMSIS-core and peripheral drivers in the application game. We will also create a low-level software driver to communicate between our core application and hardware design, which is our ARM Cortex M0 CPU. We will use an IP core provided by ARM for the hardware design of the ARM Cortex M0 CPU. We will also create hardware modules in Verilog to access different peripherals which are available on SoC like VGA, memory access block, UART block, etc. In conclusion, this lab will teach us a thorough understanding of entire application design, from higher-level software creation in C and assembly language programming to lower-level processor hardware development in Verilog.

## 2. Learning Objectives

The objective of the project is to make reusable and simple-to-use APIs to create a single player and two-player snake game in C utilizing UART and timer interrupt handlers to control, display, and detect the snake's positions.

## 3. Software & Hardware Used

1. **Keil:** We have used Keil uVision 5 for C and assembly language i.e. for snake game application development.
2. **Tera term:** Used tera term software to send (UP, DOWN, RIGHT, LEFT) commands via UART to the SoC.
3. **Vivado:** We used Xilinx Vivado software for the hardware designing of the processor and other peripherals.
4. **Nexys A7:** we have used SoC board Nexys A7 manufactured by the Xilinx

### 3.1 Verilog files used for hardware designing

Figure 2 depicts the file structure of verilog files in Vivado.



Figure 2 Verilog files

### 3.2 Software Files used in the application

Figure 3 depicts the file structure of c and assembly files 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

*Figure 3 Software File Structure*

## 4. Background

### 4.1 Application Programming Interface (API)

An API is a software abstraction layer that allows application developers to use a standard programming interface. Most operating systems, for example, have their own APIs to make it easier for programmers to create apps. Base services, visual interface, network services, and other interface services can all be provided using an API. Commercial APIs such as Java API, Windows API, Google AJAX APIs, and others are available on the market.

We have created a basic API for developing snake game applications, which will be used in this project. By merging functions from CMSIS and peripheral drivers, the API may provide general, easy-to-use functionalities for the end-user. For example, we reset both the CPU and the peripherals using a SoC startup method.

### 4.2 CMSIS

The Cortex Microcontroller Software Interface Standard (CMSIS) is a hardware abstraction layer for the Cortex-M processor series that is vendor-independent. CMSIS provides a standardized software interface, including library functions that make it easier to control the CPU, such as configuring the nested vectored interrupt controller (NVIC). The main goal is to make software more portable between multiple Cortex-M serial processors and microcontrollers.

Figure 4 shows the structure of CMSIS.

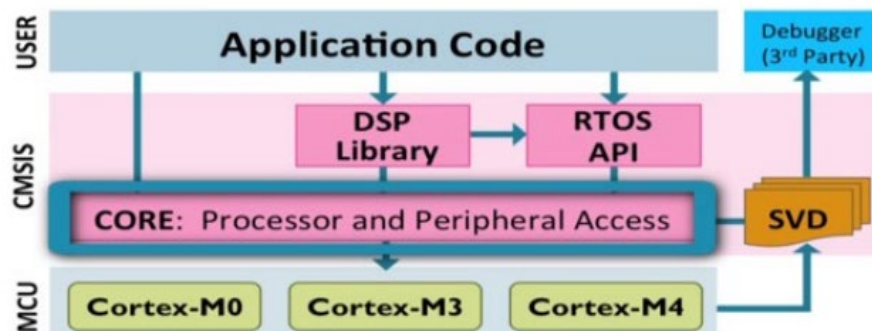


Figure 4 CMSIS Architecture

CMSIS-CORE is a peripheral register and processor interface for the Cortex-M0, Cortex-M3, Cortex-M4, SC000, and SC300 processors. Over 60 functions in fixed point (fractional q7, q15, q31) and single precision floating-point (32-bit) implementation are included in CMSIS-DSP. CMSIS-RTOS API is a standardized programming interface for thread control, resource management, and time management in real-time operating systems. CMSIS-SVD files (System View Description) contain the programmer's view of a whole microcontroller system, including peripherals. Visual design is provided by SVD.

### 4.2.1 Programming of Cortex M0 using CMSIS

There are several 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:

- REV: uint32\_t \_\_REV(uint32\_t int value)
- NOP: void \_\_NOP(void)

For standardized name of system initialization functions following function can be used:

- System initialization: void SystemInit(void)

Figure 5 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 CMSISheader files contain all the CMSIS core functions.

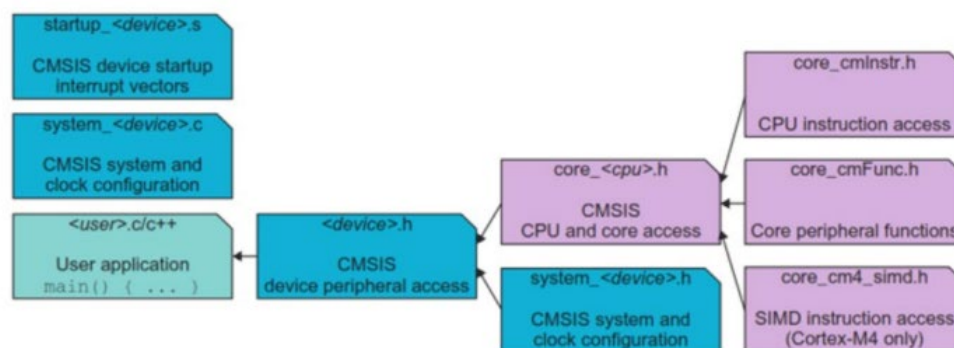


Figure 5 CMSIS File Structure



### 4.3 Interrupt Handling

The Cortex-M series processors include an interrupt controller called the Nested Vector Interrupt Controller for interrupt handling such as interrupt prioritization and interrupt masking. The NVIC contains programmable registers for interrupt management such as enable/disable, and priority levels. These registers are memory mapped. The priority levels are defined by 8-bit width registers, but only the MSB bits are implemented. In the Cortex-M0 and Cortex-M0+ processors, there are four programmable priority levels. The NVIC handles nested interrupts automatically. The Cortex-M0 Microprocessor Architecture is depicted in Figure 6. When an Interrupt Service Routine is operating, the NVIC handles interrupt prioritization and masks out same or lower priority interruptions after the priority levels of each interrupt have been set. If a higher priority interrupt occurs, the running ISR will be pre-empted so that the higher priority ISR can be executed as quickly as possible.

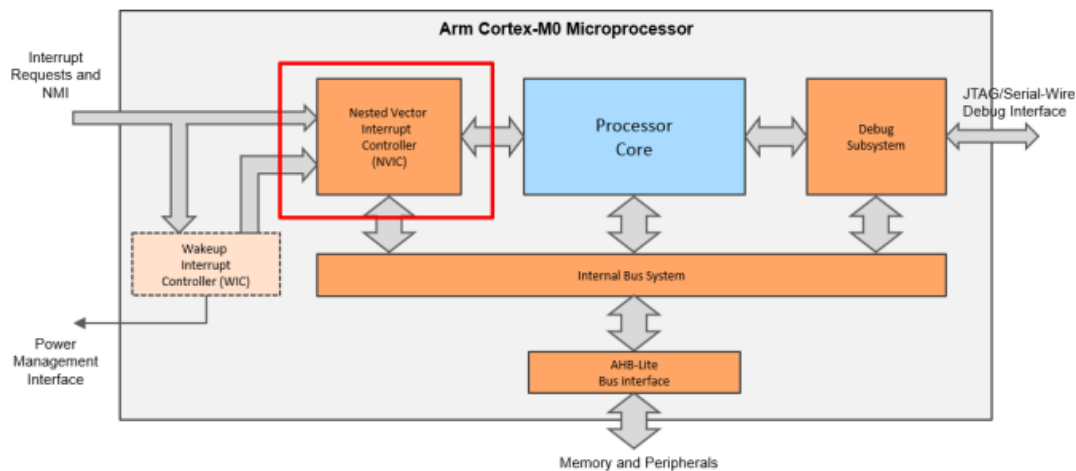


Figure 6 NVIC in Cortex-M0 Microprocessor

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 7 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.

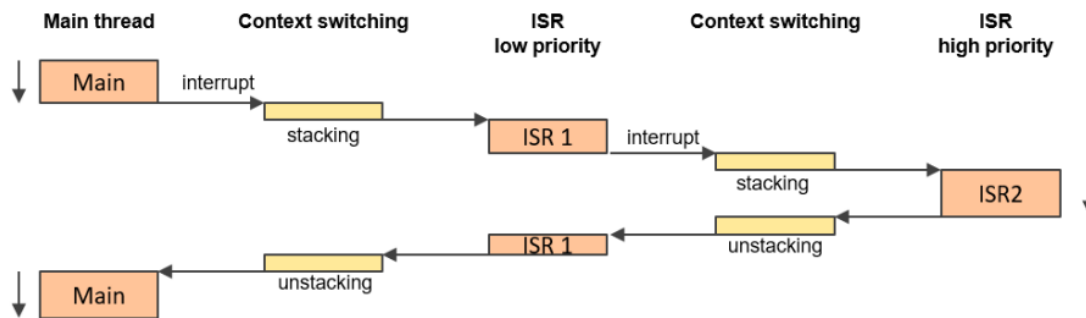


Figure 7 Nested Interrupt Handling

## 4.4 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 9.

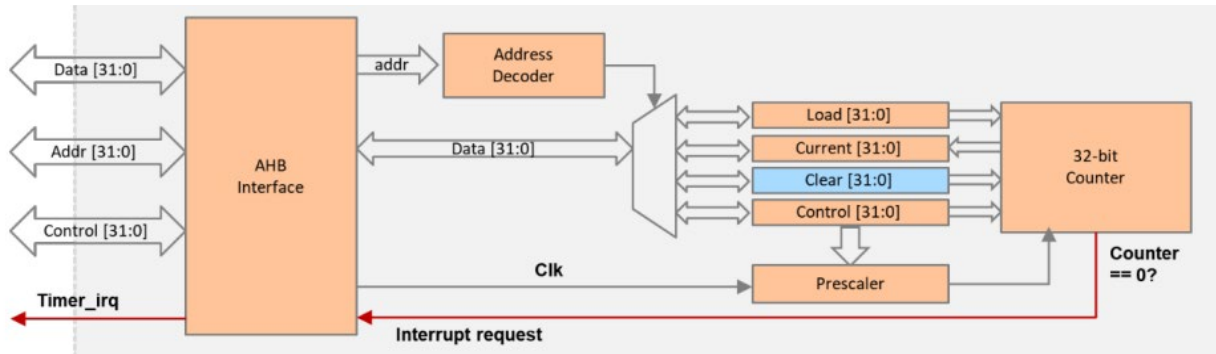


Figure 8 Timer control block

## 4.5 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](http://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.

The UART interrupt signal is depicted in Figure 10.

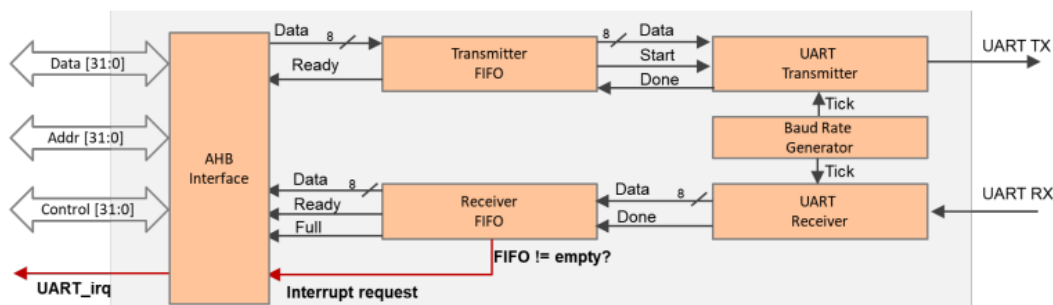


Figure 9 UART Interrupt Signal

## **5. Implementation**

### **5.1 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.

### **5.2 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 ().

### **5.3 Tasks for Single Player:**

The figure 10 explains the flow chart of basic application flow of our code for single player snake game.



Figure 10 Flowchart of Single player snake application

### Task 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.

### Task 2: Draw the target

VGA\_plot\_pixel(target.x, target.y, WHITE) is used to draw the target on the picture console.

### Task 3: Detect if the snake hits itself and end game if it does

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.' Figure 12 shows that the red snake has hit itself and game was over.

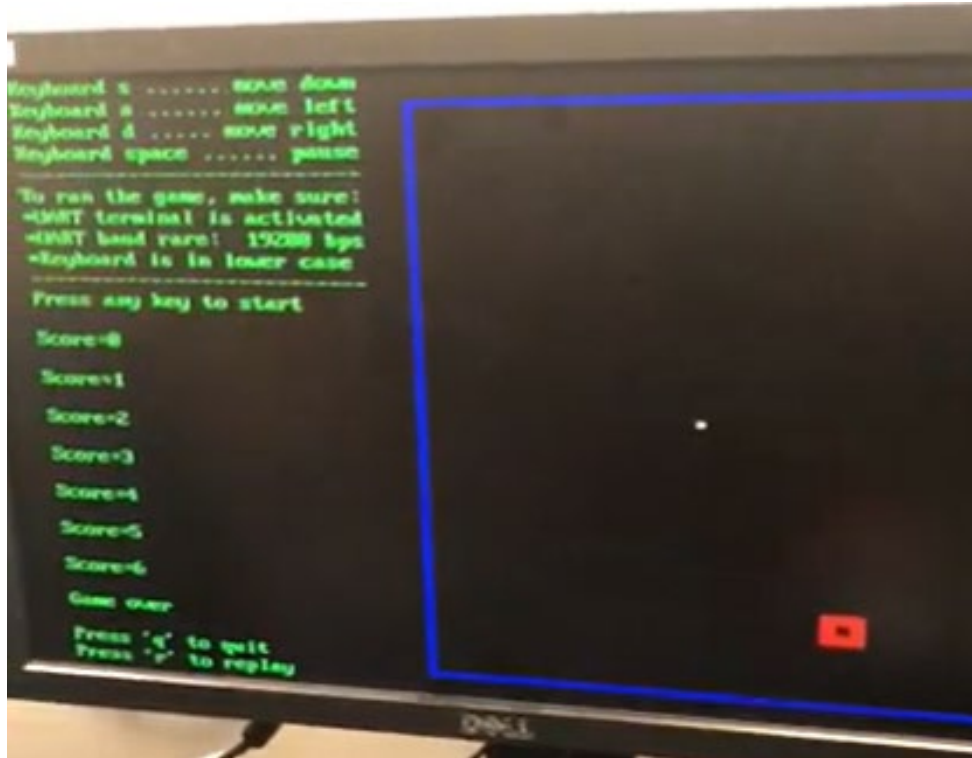


Figure 11 Body Hit for Single Player Game

#### Task 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. Table 2 explains API's, which are used to check the boundary hit conditions:

Table 2 Boundary hit functions for single player

API	Use to check boundary condition of
<code>boundary_hit(left_boundary, top_boundary, right_boundary, top_boundary+boundary_thick)</code>	Top boundary
<code>boundary_hit(left_boundary, top_boundary, left_boundary + boundary_thick, bottom_boundary)</code>	Left boundary
<code>boundary_hit(left_boundary, bottom_boundary, right_boundary, bottom_boundary)</code>	Right boundary
<code>boundary_hit(right_boundary, top_boundary, right_boundary, bottom_boundary + boundary_thick)</code>	Bottom boundary

Figure 12 shows that the red snake has hit the wall and game was over.

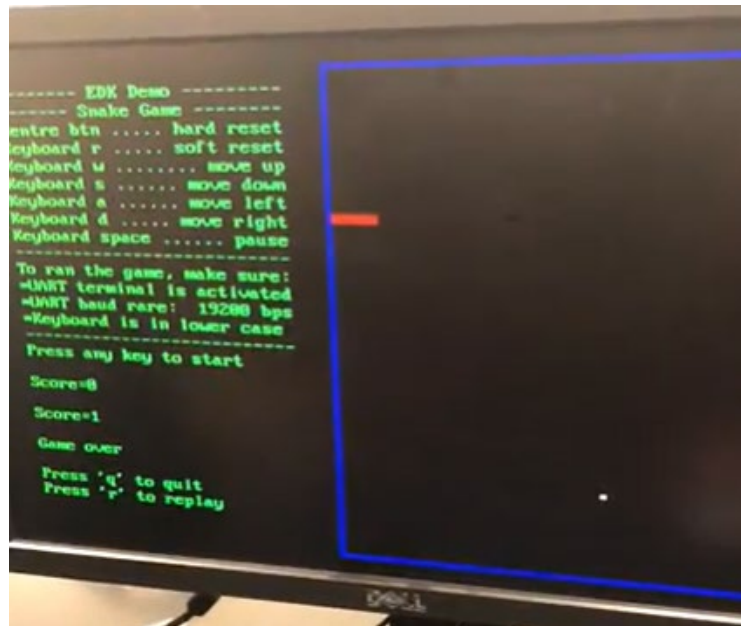


Figure 12 Boundary Hit for Single Player

## 5.4 Tasks for Double Player Snake Game:

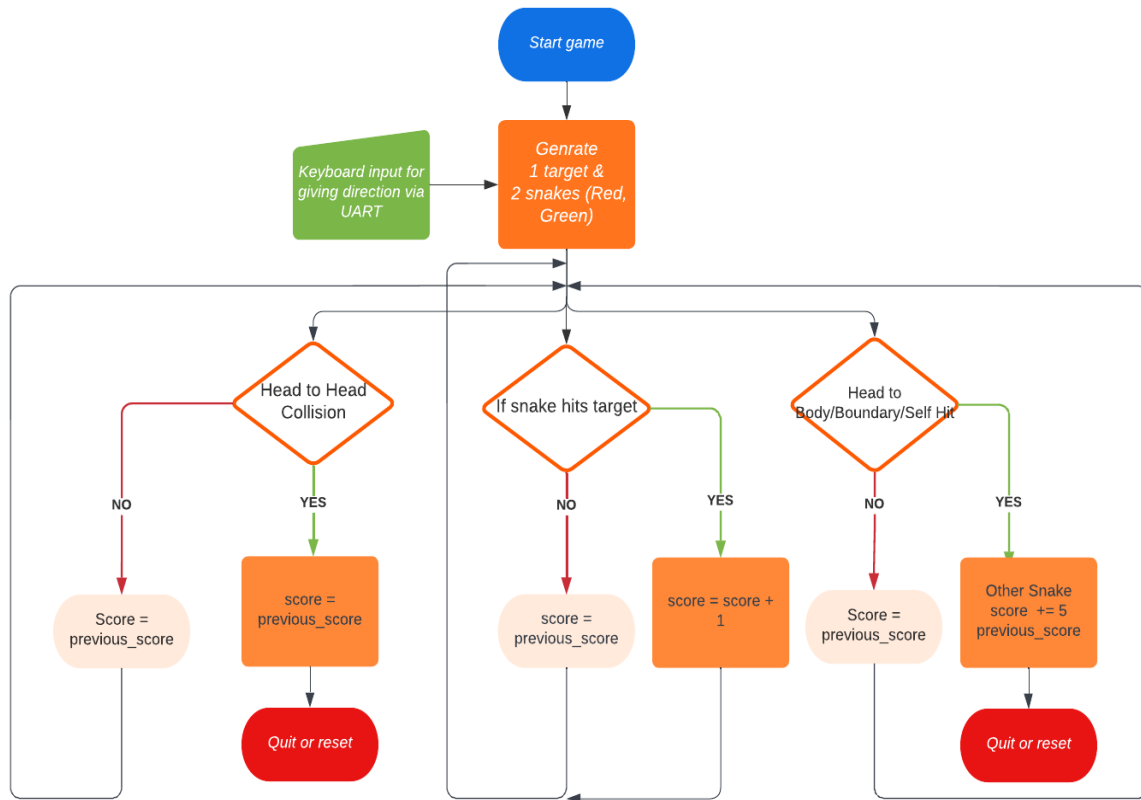


Figure 13 Flowchart of Double player snake application

Above figure 13, flowchart explains basic working flow of the code for double player snake game.

### Task 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.

### Task 2: Draw the target

VGA\_plot\_pixel(target.x, target.y, WHITE) function is used to draw the target on the picture console.



### **Task 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).

### **Task 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.

### **Task 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.

### **Task 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.

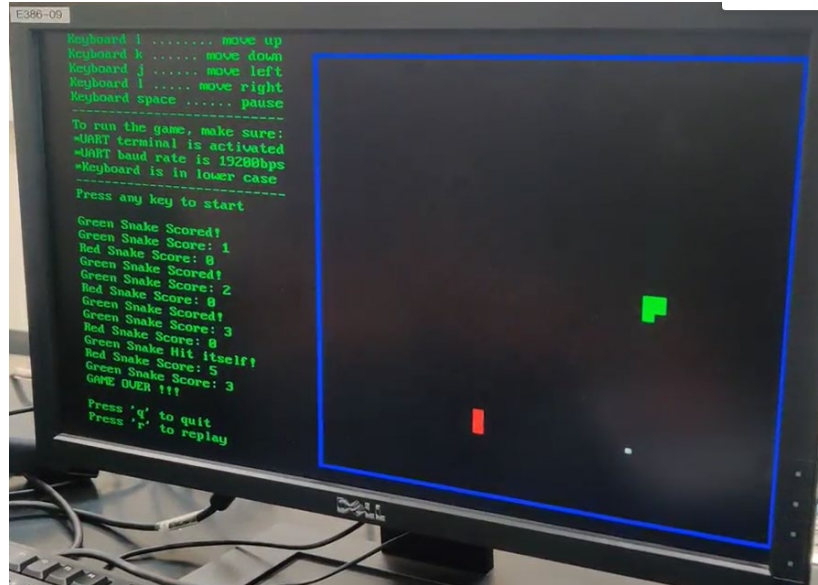


Figure 14 Self Body Hit (Green Snake) for two player snake game

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

#### Task 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:

Table 3 Boundary hit functions for two player Snake game

API	Use to check boundary condition
<code>boundary_hit(left_boundary, top_boundary, right_boundary, top_boundary+boundary_thick)</code>	Top boundary
<code>boundary_hit(left_boundary, top_boundary, left_boundary + boundary_thick, bottom_boundary)</code>	Left boundary
<code>boundary_hit(left_boundary, bottom_boundary, right_boundary, bottom_boundary)</code>	Right boundary
<code>boundary_hit(right_boundary, top_boundary, right_boundary, bottom_boundary + boundary_thick)</code>	Bottom boundary

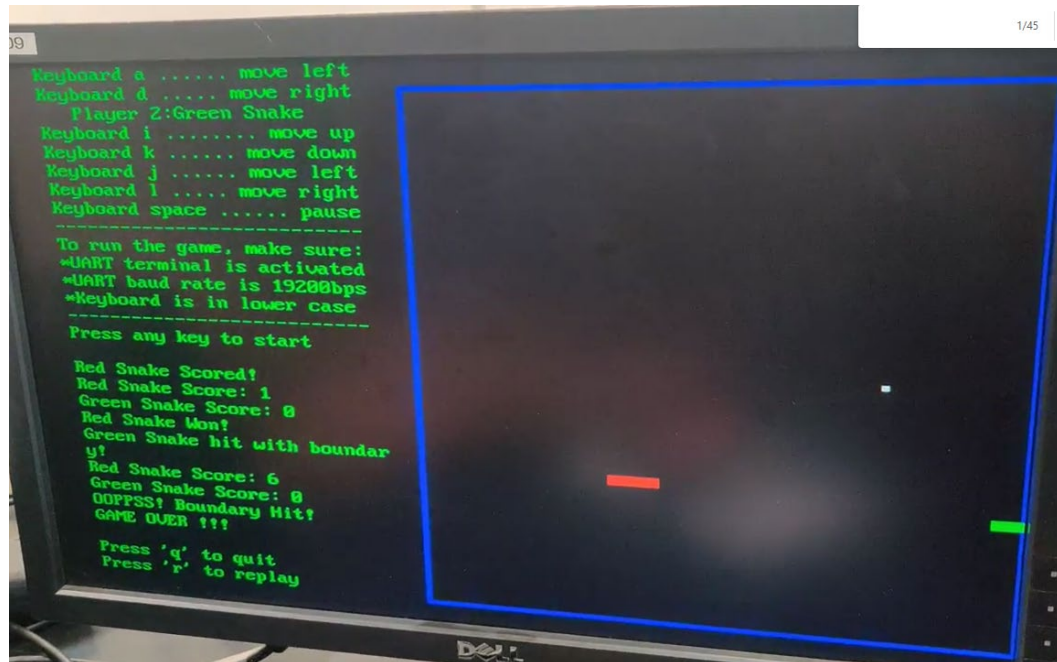


Figure 15 Boundary Hit (Green Snake) for two player Snake game

As we can see in above photo, in figure 15 green snake hits wall and 5 points get added in the score of red snake.

### Task 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.

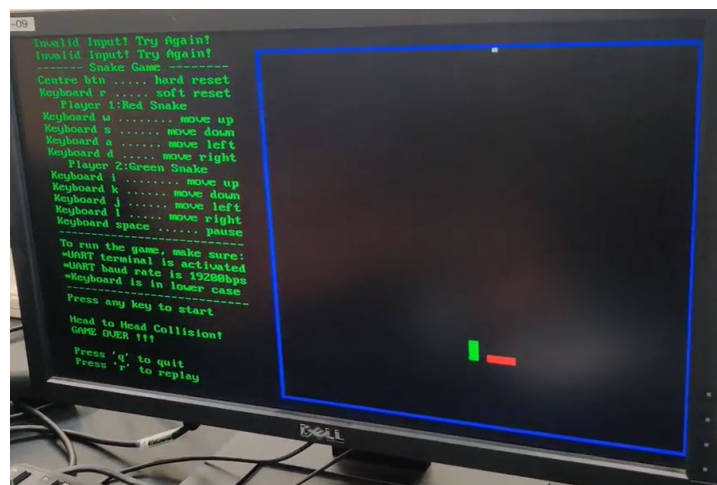


Figure 16 Head-Head Collision for Two Player Snake Game

As we can see in above photo, figure 16 green snake's head hit the head of red snake's head and no points gets added in the score of any snake.

### Task 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.

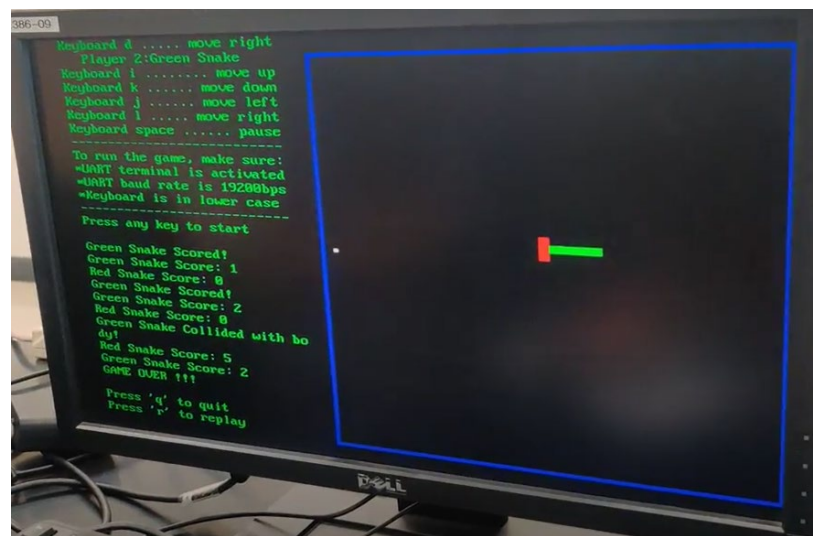


Figure 17 Body to Head Collision for two player Snake game

As we can see in above photo, figure 17 green snake hits body of red snake and 5 points get added in the score of red snake.

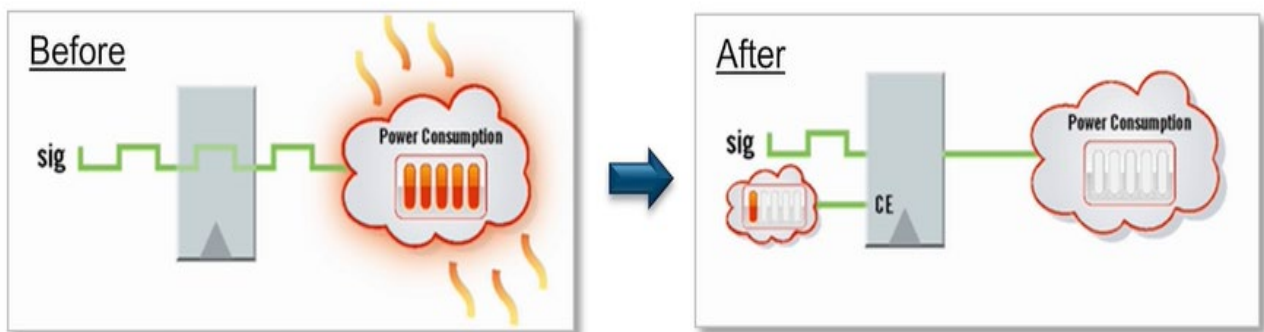
## 6. Power Consumption

### 6.1 Approach 1: Power optimization in Hardware design (using Verilog)

There are mainly 2 types of power analysis in case of hardware design first is Static and second is dynamic power usage. Static power dissipation is when the leakage current flows through the transistors after switching on the circuit/board and dynamic power dissipation is when charging discharging of capacitance in the transistors happen during the normal operation.

The Xilinx's Vivado software which we are using for this project itself has power optimization tool using which we can analyze our design right after its synthesis. We have analyzed and mentioned few power optimization scenarios as follows:

Vivado performs automatic 'Clock gating' over entire design, which prevents unnecessary switching between block ram, registers, and other such blocks in a FPGA, (keeping the original functionality of our design unchanged)



*Figure 18 Clock gating reduces power consumption of our design*

As we can see in the above image, figure 18 after using the clock gating, we can reduce the power consumption drastically, which will reduce power consumption of our overall application as well. Clock gating is a power-saving feature in semiconductor microelectronics that enables switching off circuits. Many electronic devices use clock gating to turn off buses, controllers, bridges, and parts of processors, to reduce dynamic power consumption.

```

s:/user/Desktop/LAB-10_Project/vivado_ver1/project_1_if_else/project_1.srcs/sources_1/imports/FPGA_prev_labs/AHBDCD.v

assign HSEL_NOMAP = dec[15]; //REST OF REGION NOT COVERED ABOVE

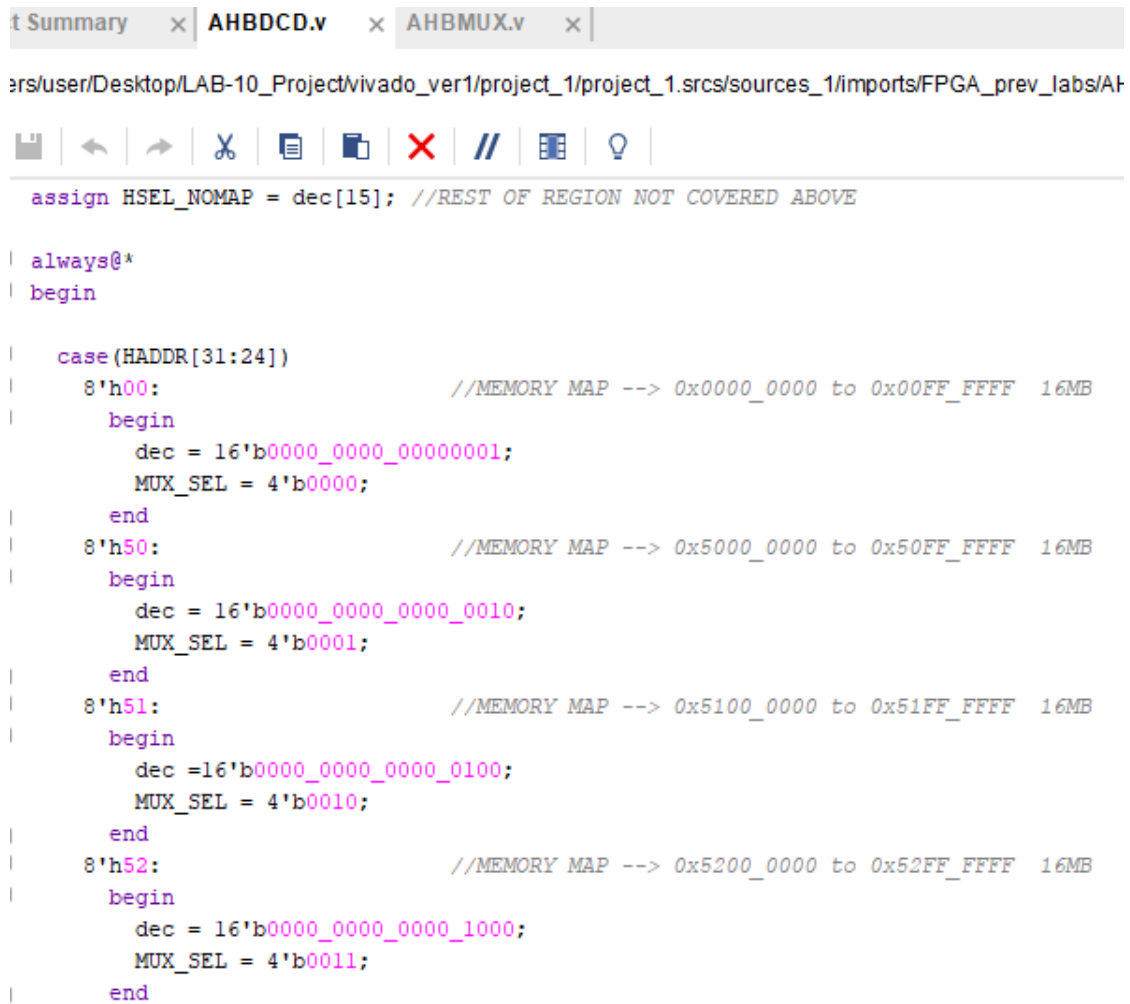
always@*
begin

    if(HADDR[31:24] == 8'h00) //MEMORY MAP --> 0x0000_0000 to 0x00FF_FFFF 16MB
    begin
        dec = 16'b0000_0000_00000001;
        MUX_SEL = 4'b0000;
    end
    else if(HADDR[31:24] == 8'h50) //MEMORY MAP --> 0x5000_0000 to 0x50FF_FFFF 16MB
    begin
        dec = 16'b0000_0000_0000_0010;
        MUX_SEL = 4'b0001;
    end
    else if(HADDR[31:24] == 8'h51) //MEMORY MAP --> 0x5100_0000 to 0x51FF_FFFF 16MB
    begin
        dec = 16'b0000_0000_0000_0100;
        MUX_SEL = 4'b0010;
    end
    else if(HADDR[31:24] == 8'h52) //MEMORY MAP --> 0x5200_0000 to 0x52FF_FFFF 16MB
    begin
        dec = 16'b0000_0000_0000_1000;
        MUX_SEL = 4'b0011;
    end
    else if(HADDR[31:24] == 8'h53) //MEMORY MAP --> 0x5300_0000 to 0x53FF_FFFF 16MB

```

Figure 19 Program using if-else statements

Also, Power can be optimized by changing the coding style by the programmer. If we minimized asynchronous reset signals from our design which are especially used for Flip flops and block RAM which will also lead to the power saving of the circuit, also the programmer can use case statements in place of 'if – else' or other serial statements, so that program optimization, which will eventually lead to the power optimization. We have also made changes in the Verilog code to analyze the case statement and if-else scenarios, mentioned in figures 19 and 20.



```
Summary x AHBDLCD.v x AHBMUX.v x |
ers/user/Desktop/LAB-10_Project/vivado_ver1/project_1/project_1.srscs/sources_1/imports/FPGA_prev_labs/At

assign HSEL_NOMAP = dec[15]; //REST OF REGION NOT COVERED ABOVE

always@*
begin

    case(HADDR[31:24])
        8'h00: //MEMORY MAP --> 0x0000_0000 to 0x00FF_FFFF 16MB
            begin
                dec = 16'b0000_0000_00000001;
                MUX_SEL = 4'b0000;
            end
        8'h50: //MEMORY MAP --> 0x5000_0000 to 0x50FF_FFFF 16MB
            begin
                dec = 16'b0000_0000_0000_0010;
                MUX_SEL = 4'b0001;
            end
        8'h51: //MEMORY MAP --> 0x5100_0000 to 0x51FF_FFFF 16MB
            begin
                dec = 16'b0000_0000_0000_0100;
                MUX_SEL = 4'b0010;
            end
        8'h52: //MEMORY MAP --> 0x5200_0000 to 0x52FF_FFFF 16MB
            begin
                dec = 16'b0000_0000_0000_1000;
                MUX_SEL = 4'b0011;
            end
    end
end
```

Figure 20 Program using case statements

Vivado also offers pre-place for power optimization and post place for the timing analysis of our hardware design. The below image is the snippet from schematic file which we have got after running synthesis of the project. And as we can in the figure 22 the block reset block is using signal 'CE' which is continuously high, means it is requires power for every clock cycle and there are other such signals which are consuming power while running the whole application.

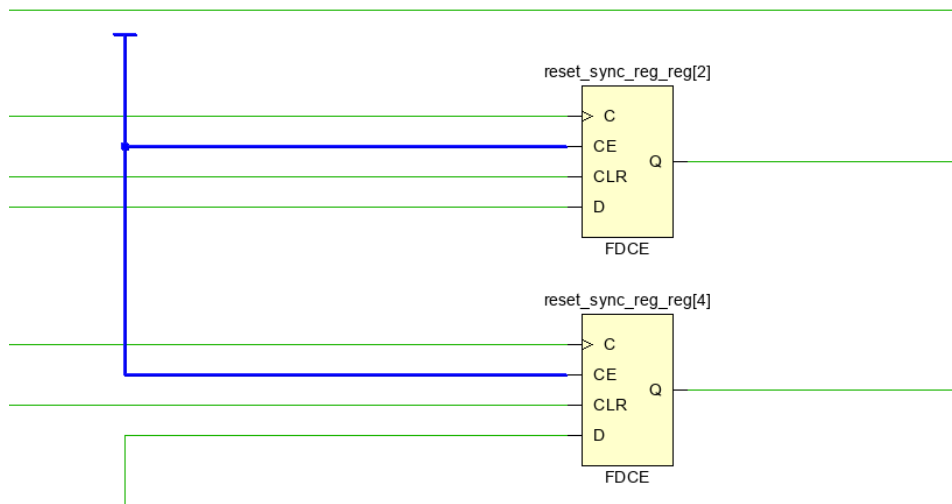


Figure 21 Schematic file after synthesis of design

So to optimize power in such cases Vivado offers post and pre power optimization option which is very useful for the designers to analyze power like mentioned in figure 22

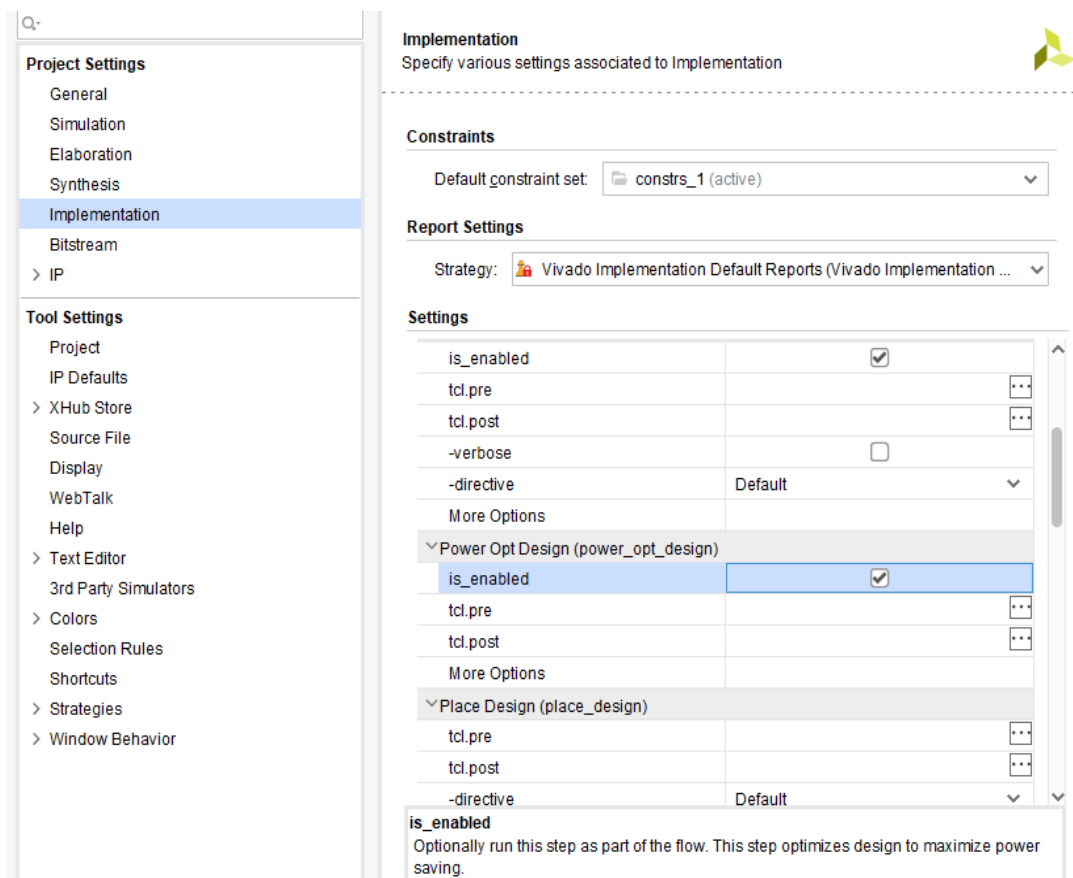


Figure 22 Power optimization settings in Vivado 'pre-place'



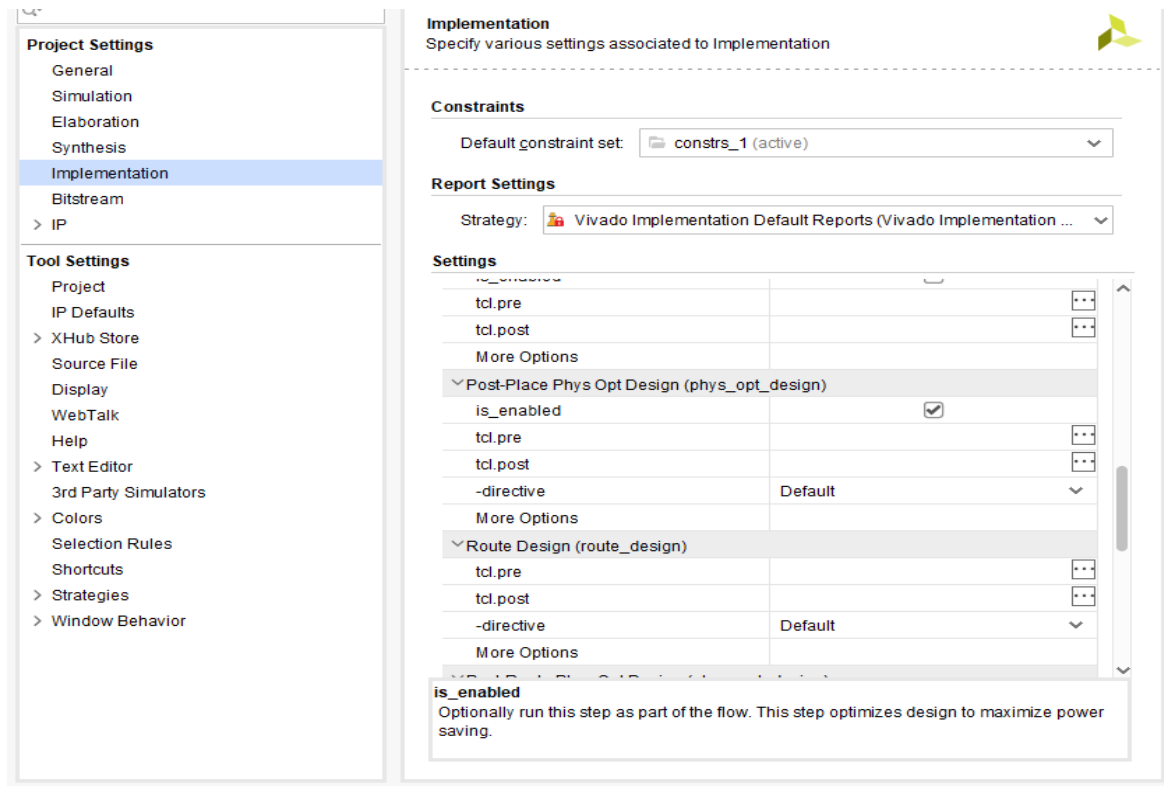


Figure 23 Power optimization settings in Vivado 'post-place'

Below we have attached the image, figure 24 before making changes in the power optimization option. In that case power consumption was around 0.161W and, we can see in the image the detailed power consumed by the different blocks.

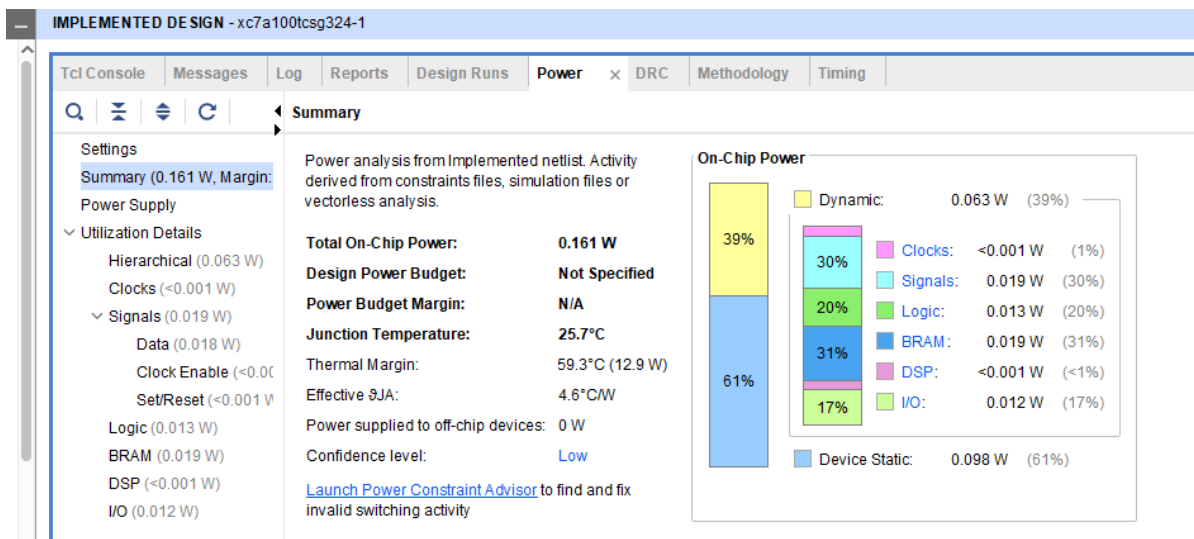


Figure 24 Power analysis before optimization

Below we have attached the image, figure 26 after making changes in the power optimization option. In that case power consumption was around 0.154W and, we can see in the image the detailed power consumed by the different blocks. So, in the summer we have optimized around  $0.161 - 0.154 = 0.007\text{W}$  of power, it looks small number but if we consider the case where our application is running for several hours than collectively, we will save a lot of power using hardware optimization techniques, and good programming practice.

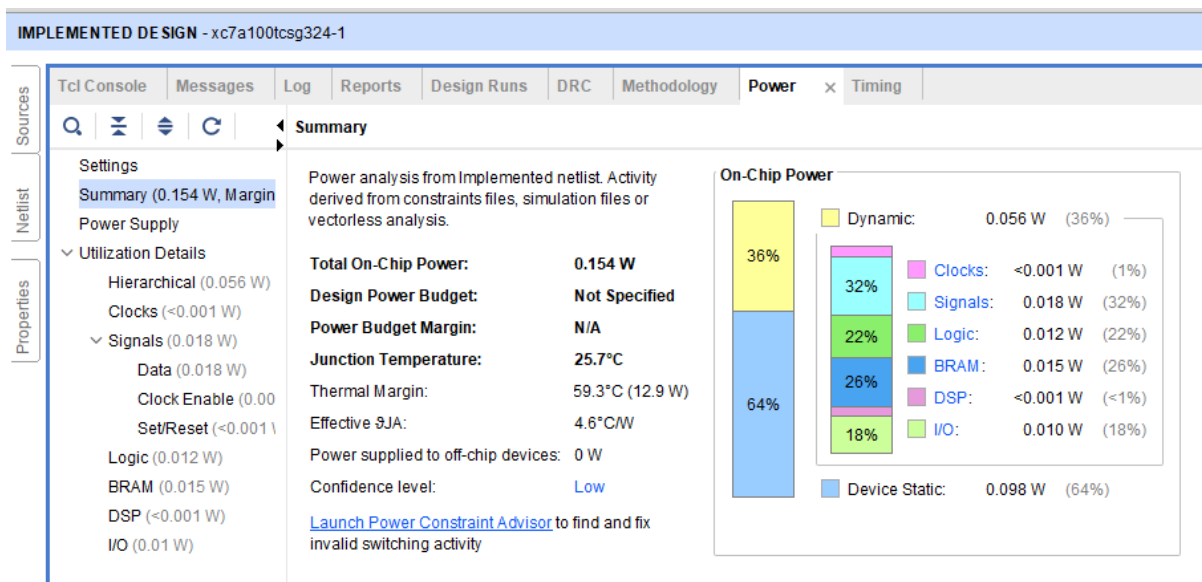


Figure 25 Power analysis after optimization

## 6.2 Approach 2: Power optimization using C programming:

The sleep-on-exit functionality instructs the CPU to enter sleep mode after all exception handlers have been handled. Because unneeded applications are not executed in the main thread, as well as needless stacking and unstacking operations are avoided, this feature helps minimize CPU power usage. Sleep-on-exit is commonly used in interrupt-driven programs because it extends the amount of time the processor may be in sleep mode.

Normal sleep mode and deep sleep mode are supported by the Cortex-M0 CPU. The actual meaning and behavior of the two modes are determined by the microcontroller's implementation. For instance, Turn off part of the clock signals for normal sleep. Deep sleep: lower memory block voltage supply and turn off other components. WFE or WFI instructions can be used to enter sleep mode. If anything happens, the CPU will wake up. WFE (wait for event) instructions, WFI (wait for interrupt) instructions, and the sleep-on-exit functionality are the three ways to enter sleep mode. If anything happens, the CPU will wake up. After the application returns from ISR, the sleep-on-exit feature does not perform the unstacking procedure. The power consumption is minimized since superfluous applications are not executed in the main thread. The need for wasteful stacking and unstacking is eliminated.

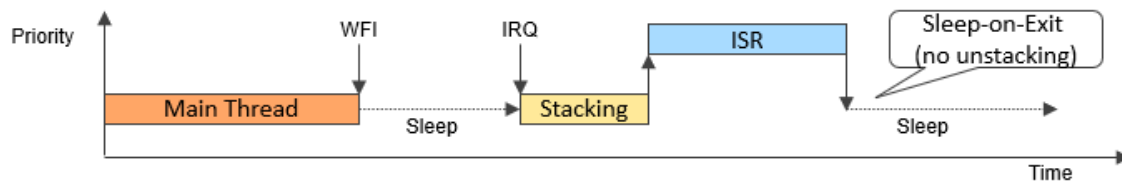


Figure 26 Task switching

As shown in Figure 26, the main thread is running and waiting for an interrupt, i.e. WFI mode (wait for interrupt), after which interrupt request (IRQ) functions are added to the stack and performed in order of priority. The CPU returns to sleep mode after completing the task. In conclusion, the CPU only wakes up when an interrupt occurs; otherwise, it is in sleep mode, conserving power. For example, the ISR calls the UART function only when a key is pushed on the keyboard; after that, further activities such as the timer function and snake movement are executed.

## 7. CONCLUSION

In this project, the SoC and peripherals were programmed at a higher level to create a snake game application using CMSIS drivers and higher-level software drivers. A timed interrupt was used to detect various circumstances such as the snake's boundary and body hits, as well as the score increment operation and show the results on the VGA monitor. By sending various characters to the SoC through a PC or laptop, a UART interrupt was developed to control the snake's orientation. Both interruptions are implemented using CMSIS' driver functions. The actions stated above are completed by calling ISRs written in embedded C code from assembly code.

## **8. RESULT**

### **8.1 Link for single player:**

1. Boundary hit condition- [link](#)
2. Body hit condition- [link](#)

### **8.2 Link for double player:**

1. Boundary hit- [link](#)
2. Self-hit- [link](#)
3. Head- body and head-head collusion- [link](#)

Instruction to play the snake game is included in the 'readme.txt' file in 'SnakeGame' folder.