Final Project

EE277A: Embedded SoC Design

San Jose State University Electrical Engineering Department

Team Members:

- Zeel Jatinkumar Lia
- Vishwajeetsinh Varnamiya
- Sai Kashyap Kurella

Degree: MSEE

Date: 05/18/2023

1. INTRODUCTION

This report provides an overview of the development of a Tic Tac Toe game implemented on a Cortex-M0 microcontroller using UART, 7-segment,GPIO, and VGA peripherals. The objective of the project was to create a simple and interactive game that allows two players to compete against each other in a classic game of Tic Tac Toe.

The C programming language was typically 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 further facilitated by using the CMSIS abstraction layer. This enables us to create more intricate applications in a manner that is more time and labour efficient. It is always possible to combine C code with assembly code inside the same project, or even the same file. Use the C programming language to write programs for the Cortex-M0 CPU and to control the peripherals. We have used Vivado, Keil uVision and TeraTerm for this Project.

2. HARDWARE DESIGN AND PERIPHERALS USED

2.1. BUILIDING HARDWARE SOC

This sub-unit will introduce the fundamentals of C and assembly programming, as well as instruction on how to utilize C and assembly to write programs for the Cortex-M0 CPU.

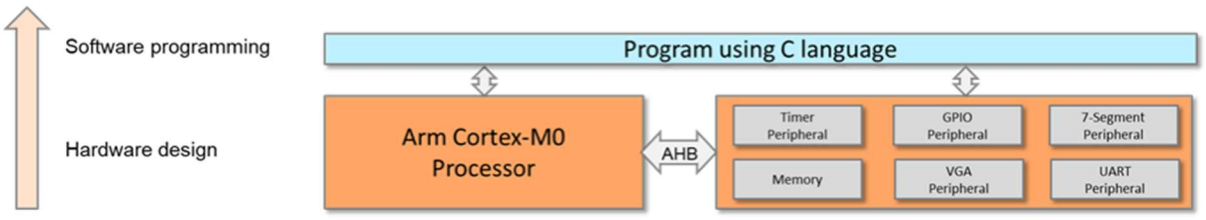


Figure 1. NVIC in Cortex-M0 Microprocessor.

The C programming language, assembly code, or a combination of the two can be used to write software for the Cortex-M0 CPU. The utilization of the C programming language and assembly language is compared in the following table. Although the C programming language is easy to learn, it only offers limited access to the core registers and the stack. It also runs on a range of platforms and makes it simple to manage complex data structures.

In addition, the creation of instruction sequences will not be under the direct control of the C programmer. Although assembly language is more challenging to learn and maintain, it offers direct control over every instruction and every bit of memory. Assembly language learning takes more time. Programmers developing SoCs frequently need to combine the two languages. The embedded assembler found in Arm C compilers makes it simple to incorporate assembly functions into C program code. Similar to this, most other C compilers feature an inline assembler that may be used to incorporate assembly code inside of a C program file.

2.2. PERIPHERALS USED

2.2.1. VGA PERIPHERAL

The Diligent Nexys VGA Module is intended to give the Diligent Nexys board a VGA interface. It can display up to 4096 colors on a VGA monitor using a 12-bit interface. The module can be linked to a VGA monitor using a common 15-pin VGA cable and is attached to the Nexys board through a 16-pin header at J8.

The VGA interface on the Nexys board consists of five standard signals, namely Red, Green, Blue, Horizontal Sync (HS), and Vertical Sync (VS), which are connected directly from the FPGA to the VGA connector. To achieve a 12-bit interface capable of producing 4,096 colours, the FPGA routes four signals for each of the standard VGA colour signals.

Each of the video signals has a series resistor that, when paired with the 75-ohm termination resistance of the VGA display, creates a divider circuit and ensures that the video signals do not exceed the maximum voltage defined by the VGA standard. These straightforward circuits ensure that the color signals are either completely on (0.7V), completely off (0V), or halfway between. Figures 1 and 2 show how the NEXYS A7 VGA Interface works.

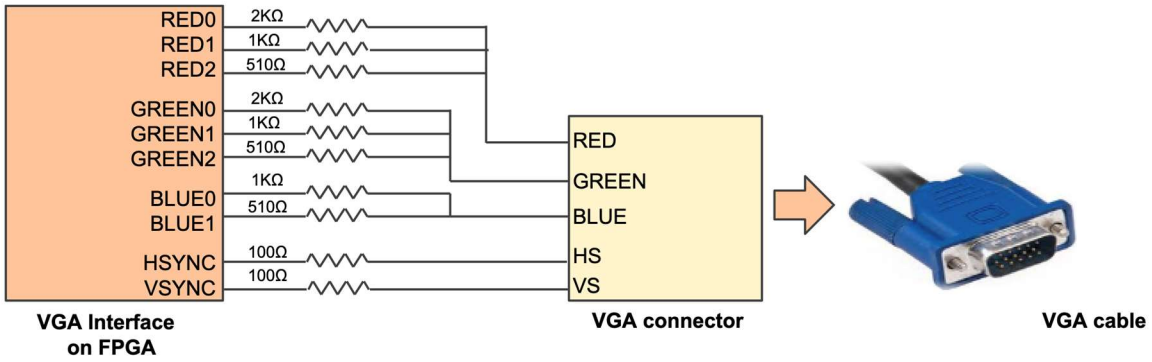
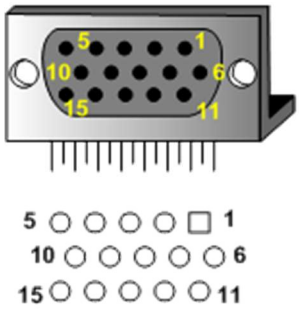


Figure 1. Block level demonstration of VGA.



VGA pin assignments

Pin	Signal	Pin	Signal
1	Red	9	NC
2	Green	10	GND
3	Blue	11	NC
4	NC	12	NC
5	GND	13	HS
6	GND	14	VS
7	GND	15	NC
8	GND		

VGA signal color mapping

Color \ Signal	Red	Green	Blue
Black	0	0	0
Blue	0	0	1
Green	0	1	0
Cyan	0	1	1
Red	1	0	0
Purple	1	0	1
Yellow	1	1	0
White	1	1	1

HD DB-15 connector, PCB hole pattern, pin assignments, and color-signal mapping

Figure 2. VGA pin diagram and explanation

A VGA cable connects the VGA peripheral to a monitor, allowing it to display text and graphics. An AHB interface, a VGA interface, an image buffer for showing images and a text console module for displaying words make up the system's five components. Texts and images can be shown on a monitor via a VGA cable with the help of the suggested VGA peripheral. Here is a diagram of the VGA peripheral block diagram. This system consists of five parts: an AHB interface, a text console module, an image buffer, and a multiplexer.

VGA peripheral can display texts and images on a monitor through a VGA cable. It consists of 5 components: , an AHB interface, a VGA interface, an image buffer for displaying images, a text console module for displaying texts, and a multiplexer. The digital outputs from the FPGA can be converted to analog and connected to the VGA connector using resistor-divider circuits. The example below utilizes 10 signals, including 8-bit colour and two standard sync signals; thus, 256 colour levels can be presented.

Master: The master (e.g., a processor) selects one peripheral (or one register) by giving the address to the address bus. At the same time, it sets control signals, such as read or write and transfer size.

AHB-Lite Master Interface: The AHB-Lite manager provides address and control information to initiate read and write operations. The processor also receives the response from the peripheral, including data and ready and response signal.

AHB-Lite Peripheral Interface: An AHB-Lite subordinate responds to transfer initiated by the manager in the system. The signal HSELx is the output from the address decoder, which is used to select one subordinate at a time.

VGA interface: Generates synchronization signals to the VGA port which is directly connected to external pins of the VGA port and outputs the address of the current pixel. VGA interface consist of three modules, frequency divider, horizontal address counter, vertical address counter.

Image buffer: Stores the colour information of all pixels in the image region, which is implemented on a dual-port memory, In this lab we have used (16M- 4)byte to store the image pixels.

Text console: Displays texts in the text region, and implemented on hardware logics. For this lab we have used 1 word (4 byte) space to print a character.

Some chips do not have a large on-chip memory, such as on-chip SRAM. In such a case, the resolution can be reduced by mapping multiple pixels to a single data in the memory. For example, a 4×4 pixel region can be presented by one single data in the image buffer.

Name	Description
vga_red[2:0]	3-bit red signal
vga_green[2:0]	3-bit green signal
vga_blue[1:0]	2-bit blue signal (less sensitive to eyes)
Hsync	Horizontal synchronization signal: **one single zero** pulse indicates the start of the next line.
Vsync	Vertical synchronization signal: **one single zero** pulse indicates the start of the next frame.

Table 1: VGA signal used

Above table contains the name and description of some of the VGA signals. Some chips do not have a large on-chip memory, such as on-chip SRAM. In such a case, the resolution can be reduced by mapping multiple pixels to a single data in the memory. For example, a 4×4 pixel region can be presented by one single data in the image buffer.

2.2.2. AHB UART PERIPHERAL

The Nexys A7 includes an FTDI FT2232HQ USB-UART bridge (connected to connector J6) that enables PC software to interact with the board using common Windows COM port commands. Using a two-wire serial interface (TXD/RXD) that can optionally enable hardware flow control (RTS/CTS), the FPGA is communicated with. The C4 and D4 FPGA pins will generate serial data traffic once the drivers have been installed, which enables the PC to issue I/O commands to the COM port.

Two status LEDs are incorporated inside the port: "LD20" for transmission and "LD21" for receive. These LEDs offer visual feedback on the port traffic, which is signalled by LD19. Signal designations indicating direction are decided from the viewpoint of the Data Terminal Equipment (DTE), in this circumstance, the PC.

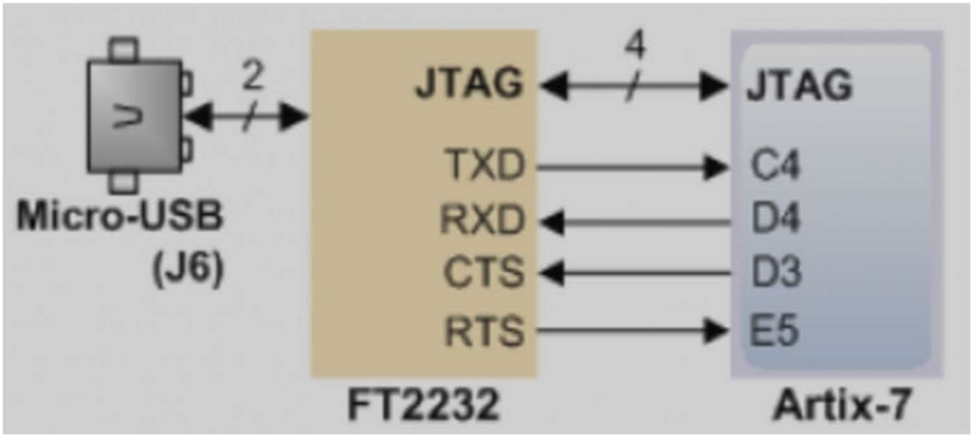


Figure 3. UART Peripheral demonstration

The basic blocks of the AHB Protocol, such as Master, Slave, decoder, and multiplexers, are developed. It is possible to use the AMBA-AHB protocol in any application, provided that the design is AHB-compliant. Peripherals from AHB All of these features are implemented utilizing on-chip memory blocks, and the processor will communicate over the AHB light bus with the VGA and UART peripherals, which may communicate with each other and display data from a host or PC.

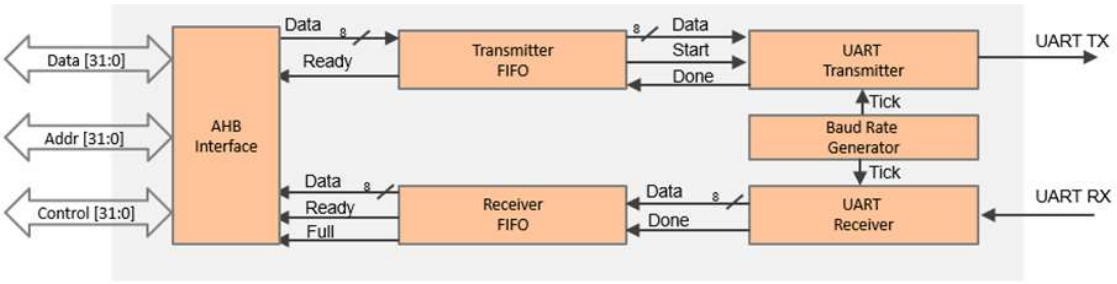


Figure 4. Low-Level Hardware design of UART

Transmitter FIFO: A more powerful clock frequency is used by the processor, such as 50,000,000 Hz. UART data is sent at a significantly lower frequency, such as 19,200 Hz. A lot of time is lost if the processor waits for the UART. FIFOs are therefore employed to raise system effectiveness.

Receiver FIFO: Used to provide the processor with additional time to deal with interrupt signals. The term "FIFO" describes a data buffer, such as a data queue, that outputs the data that was first received. A buffer that produces its most recent input data, such as the program stack, is known as last in first out (LIFO), in contrast. A synchronized FIFO Both reading and writing are done using the same clock. With asynchronous FIFO for writing as well as reading, several clocks are utilized.

2.2.3. PERIPHERAL: GPIO

The fundamental registers in GPIO are the direction register, data in, and data out. It has no extra functions or a mask register. The hardware architecture of AHB GPIO is shown in Figure 7.

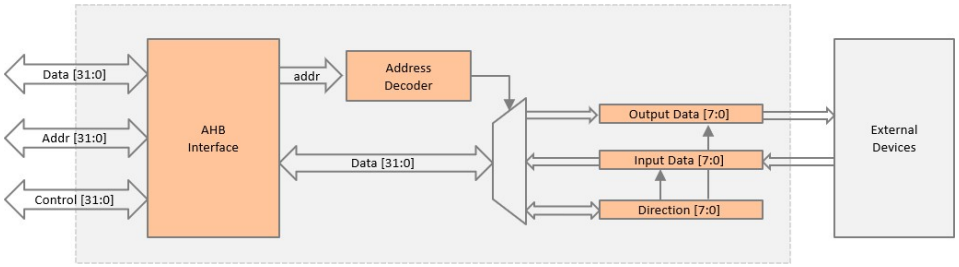


Figure5. Block level demonstration of GPIO peripheral

The GPIO peripheral registers include a Direction register, which regulates whether the operation of reading or writing is being performed, and Data registers, which hold input and output data (i.e., data that is being read from or transmitted to external devices). The base address of GPIO registers is displayed in Table 1.

Register	Address	Size
GPIO base address	0x5300_0000	
Data	0x5300_0000	4 Byte
Direction	0x5300_0004	4 Byte

Table2. Base Addresses corresponding to GPIO registers

2.3. INTERRUPT HANDLING IN CORTEX-M0

The Cortex-M series processors are equipped with the Nested Vector Interrupt Controller (NVIC) to handle interrupts through prioritization and masking. The NVIC provides memory-mapped registers that can be configured to manage interrupts, including enabling or disabling them and defining priority levels.

The Nested Vector Interrupt Controller (NVIC) in the Cortex-M0 Microprocessor Architecture efficiently handles nested interrupts. The NVIC prioritizes interrupts and blocks any preset interruptions with the same or lower priority when an Interrupt Service Routine (ISR) is being executed. When a higher priority interrupt comes in, the currently running ISR is momentarily put on hold so that the higher priority ISR can be executed without delay. Figure 9 provides an illustration of this behaviour.

The CPU employs a vector table to find the initial address of the associated Interrupt Service Routine (ISR) whenever an interrupt is generated. The vector table is typically placed at the start of the memory area. However, user software or a bootloader have the ability to relocate it to a new address. The reset vector position and the Main Stack Point's starting value are both located in the vector table.

If you look at Figure 9, 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.

In Figure 10, 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 lesser priority interrupt remains being serviced. This is often referred to as interrupt pre-emption or nested exceptions.

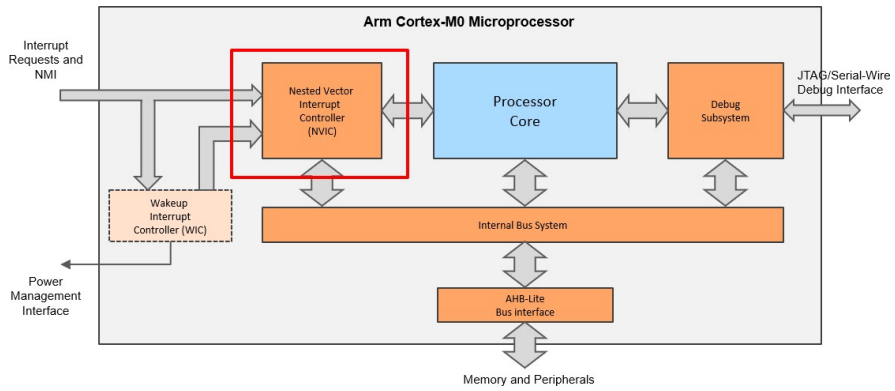


Fig 9. NVIC: Cortex-M0 Microprocessor.

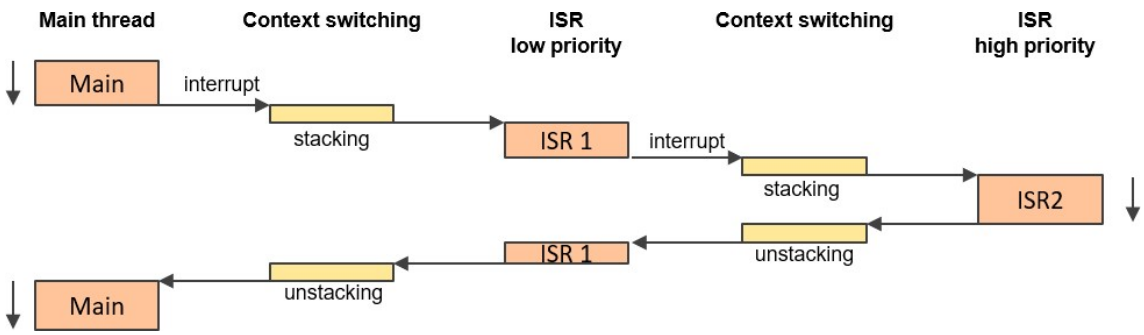


Fig 10. "Nested Interrupt Handling"

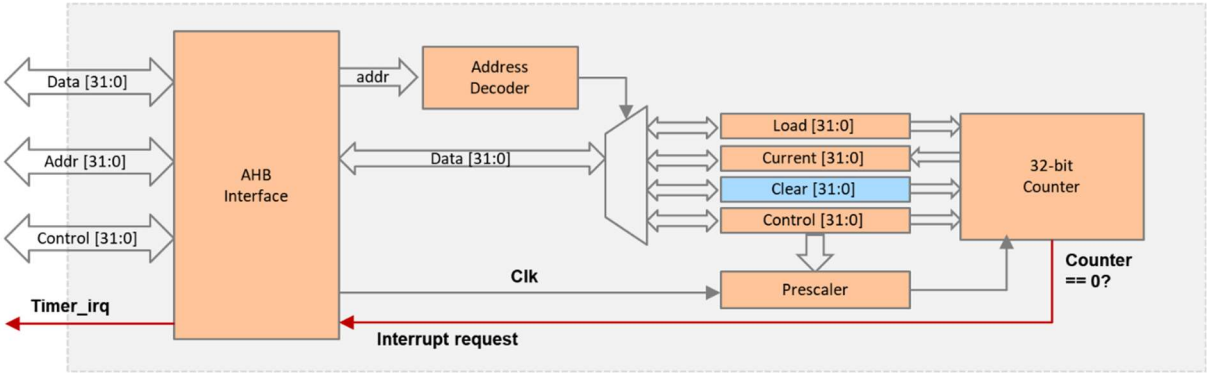


Fig 11: Interrupt Enabled Timer Peripheral

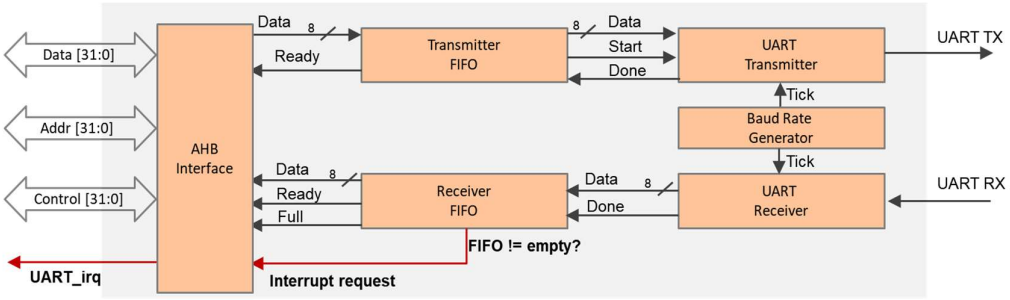


Fig 12. Interrupt enabled UART

2.4. CORTEX MICROCONTROLLER SOFTWARE INTERFACE STANDARD(CMSIS)

CMSIS is a hardware abstraction layer designed specifically for the Cortex-M processor family, aiming to provide a consistent software interface across different vendors. It offers a set of library functions that simplify CPU management tasks, including the configuration of the nested vectored interrupt controller (NVIC). The main goal of CMSIS is to enable seamless software portability between different Cortex-M microcontrollers and serial processors.

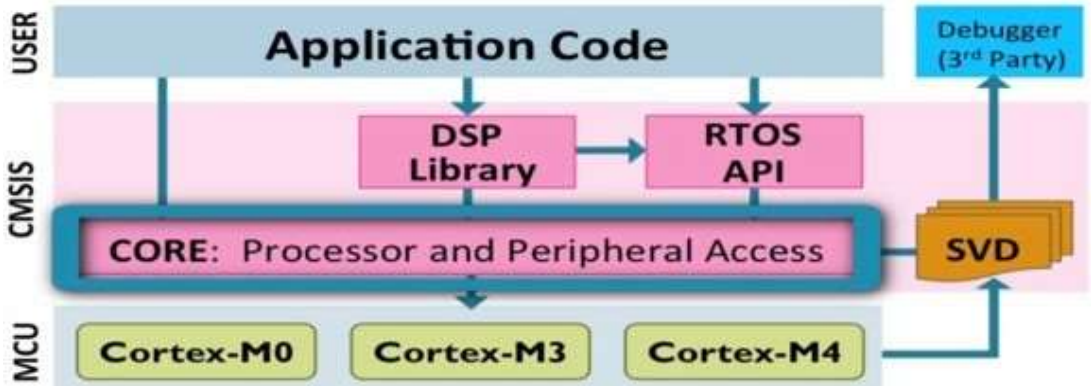


Fig 13. Architecture of CMSIS

Figure 5 illustrates the structure of CMSIS. CMSIS-CORE is responsible for providing a standardized interface for processors and peripheral registers across a range of Cortex-M processors, including Cortex-M0, Cortex-M3, Cortex-M4, SC000, and SC300. CMSIS-DSP incorporates over 60 functions that support both fixed-point (fractional q7, q15, q31) and single precision floating-point (32-bit) implementations. The CMSIS-RTOS API serves as a standardized programming interface for managing threads, resources, and time in real-time operating systems. CMSIS-SVD files (System View Description) offer a programmer's view of the complete microcontroller system, including peripherals, while also providing visual design capabilities.

2.4.1. USING CMSIS TO PROGRAMME MICROPROCESSOR(CORTEX-M0)

CMSIS offers a range of standardized functions that cover access to peripherals, registers, and special instructions. These functions allow for reading from and writing to core registers, as well as executing special instructions. Moreover, CMSIS provides intrinsic functions that are tailored for specific Cortex-M0 special instructions, which are clearly specified in a table. Furthermore, CMSIS can be employed for system control and configuring the SysTick timer.

Numerous standardized functions are available for accessing the NVIC, system control block (SCB), and system tick timer (SysTick). For instance, some of these functions include:

- To activate an interrupt or exception, utilize the function 'NVIC_EnableIRQ (IRQn_Type IRQn)'.
- To set the pending status of an interrupt, utilize the function 'NVIC_SetPendingIRQ (IRQn_Type IRQn)'.

To access special registers in a standardized manner, the following functions can be employed:

- To read the PRIMASK register: use 'uint32_t get_PRIMASK(void)'
- To set the CONTROL register: use '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.

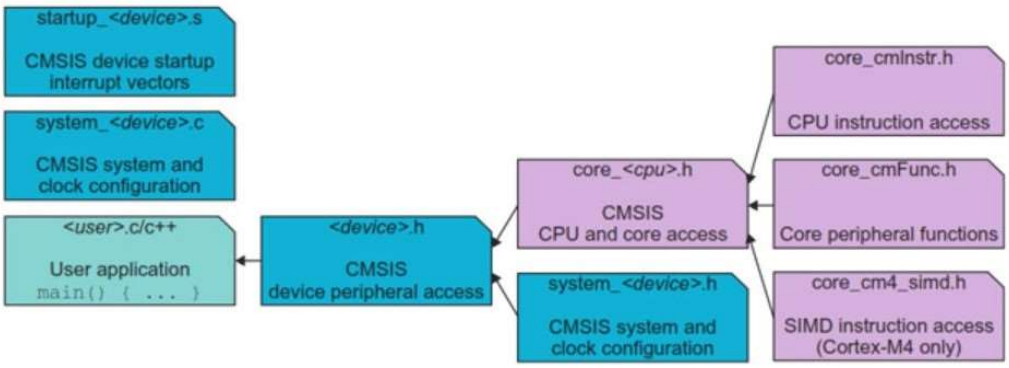


Fig 14. CMSIS File Structure

3. IMPLEMENTATION AND WORKING

We have used C programming to develop this tic-tac-toe game. 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 3: 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.

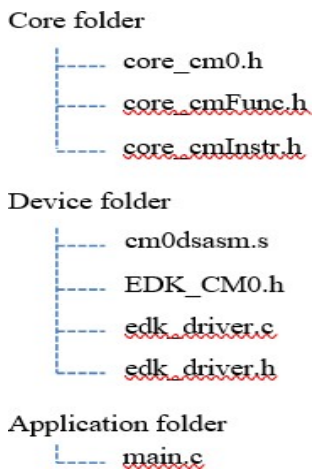


Figure 15: File structure in Keil Project

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.

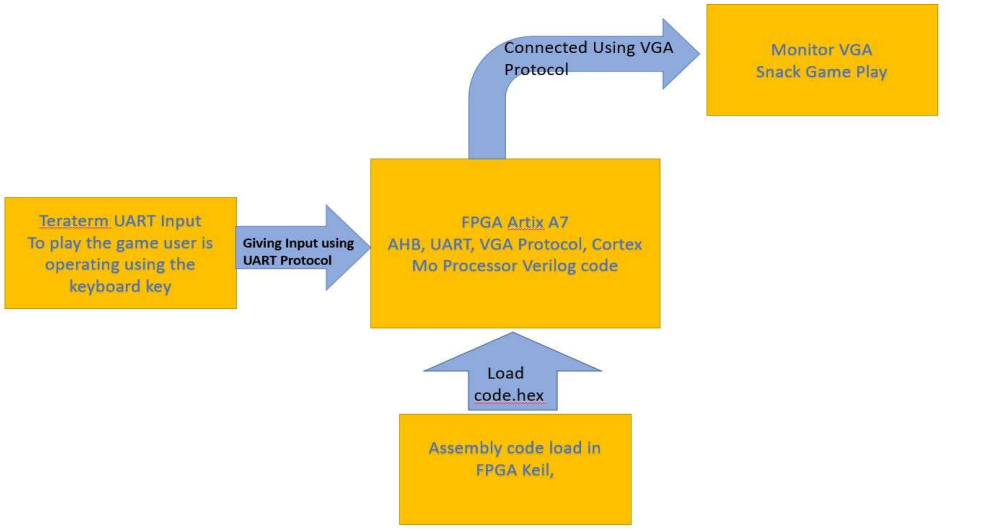


Figure 16. Basic block diagram explaining game operation

3.2.FLOW CHART AND GAME LOGIC EXPLANATION

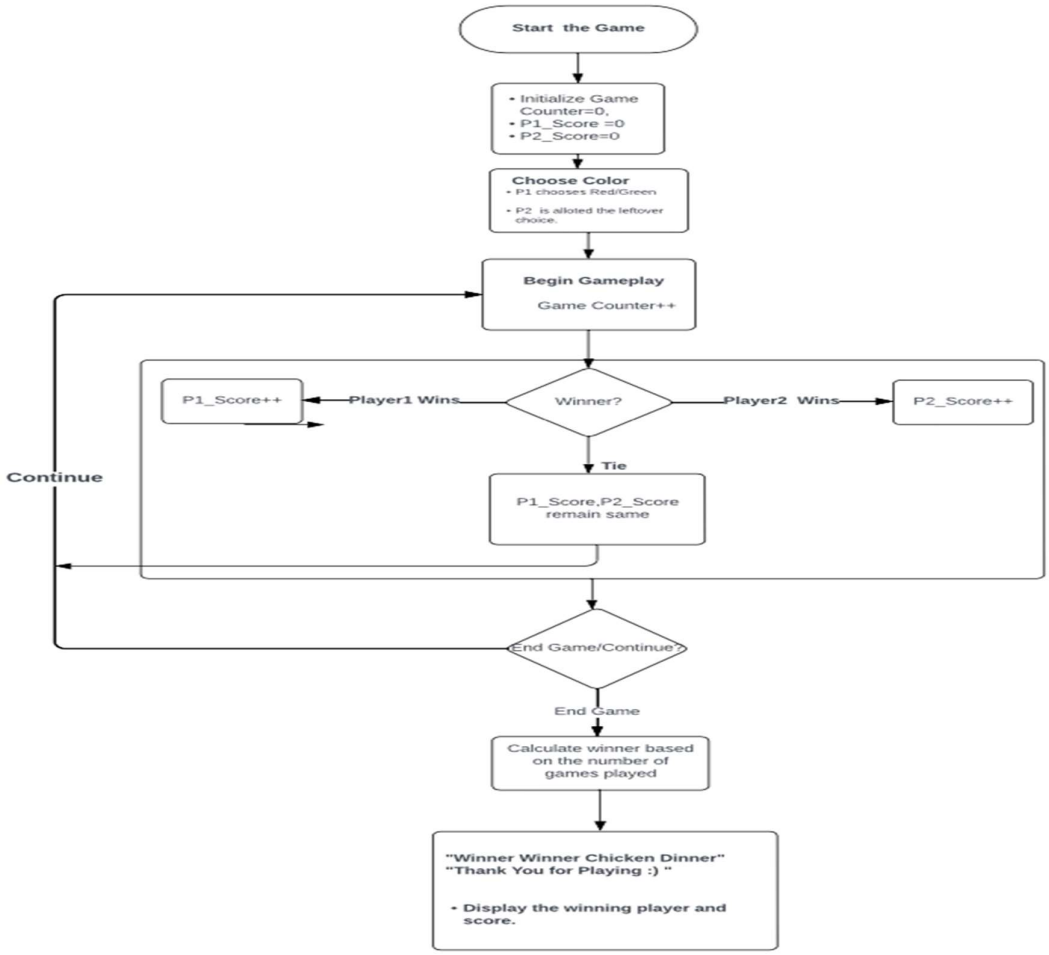


Figure 16. Flow chart demonstrating logic behind gameplay

The game begins with the initialization of the game counter, which keeps track of the number of games played. Additionally, the scores for PLAYER 1 (P1__Score) and PLAYER 2 (P2__Score) are set to zero at the start. Next, the players are given the option to choose their preferred color, either Red or Green. Player 1 (P1) selects their color first, and Player 2 (P2) is assigned the remaining color.

Once the color selection is complete, the gameplay begins. The game counter is incremented, indicating the start of a new game. If Player 1 wins the game, their score (P1_Score) is incremented, and the winner is determined. Conversely, if Player 2 wins, their score (P2_Score) is incremented. In the case of a tie, where neither player wins the game, the scores remain the same.

After each game, the players have the option to continue playing or end the game. If they choose to continue, the game counter is incremented, and a new game begins. However, if they decide to end the game, the overall winner is calculated based on the total number of games played and the scores of both players.

Upon reaching the end of the game, a message is displayed congratulating the winner with the phrase "Winner Winner Chicken Dinner." Additionally, a final message is shown to express gratitude to the players for participating, stating "Thank You for Playing :)". All messages are shown on text console of the VGA display. Overall, this flow represents the sequence of events in the Tic Tac Toe game, including initializing scores, allowing color selection, conducting gameplay, tracking scores, determining the winner, giving the option to continue or end the game, and finally, displaying messages to acknowledge the winner and thank the players for their involvement.

4. SIMULATION AND RESULT

The picture below show multiple scenarios of the game play. Figure.17 shows Player-1 winning scenario, where Player-1 gets a point.Figure.18 shows Player-2 winning scenario wherein Player-2 gets the point and Figure-19 shows the tie situation where neither Palyer-1 nor Player-2 gets a point and the score of each remain the same.

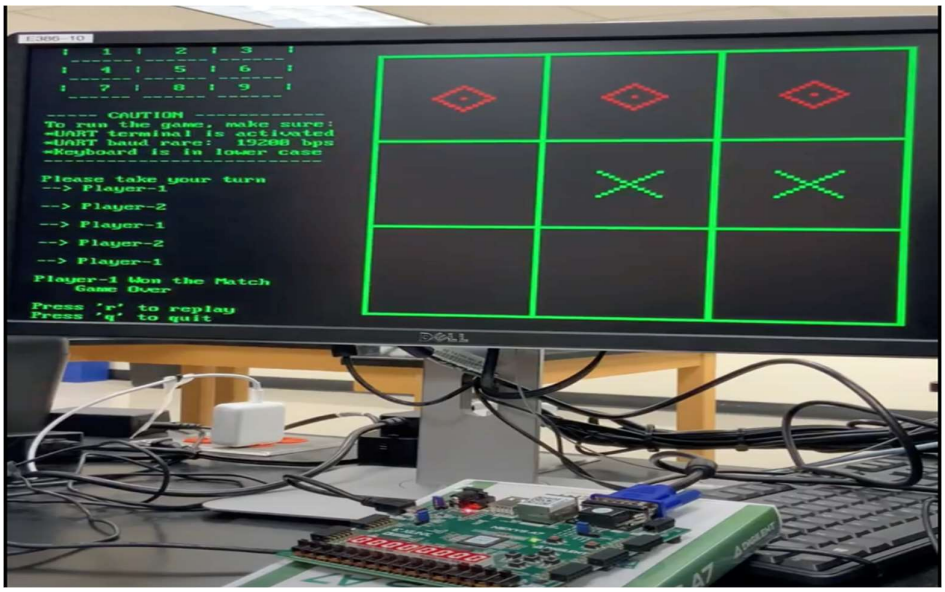


Figure 17. Player-1(RED) winning condition

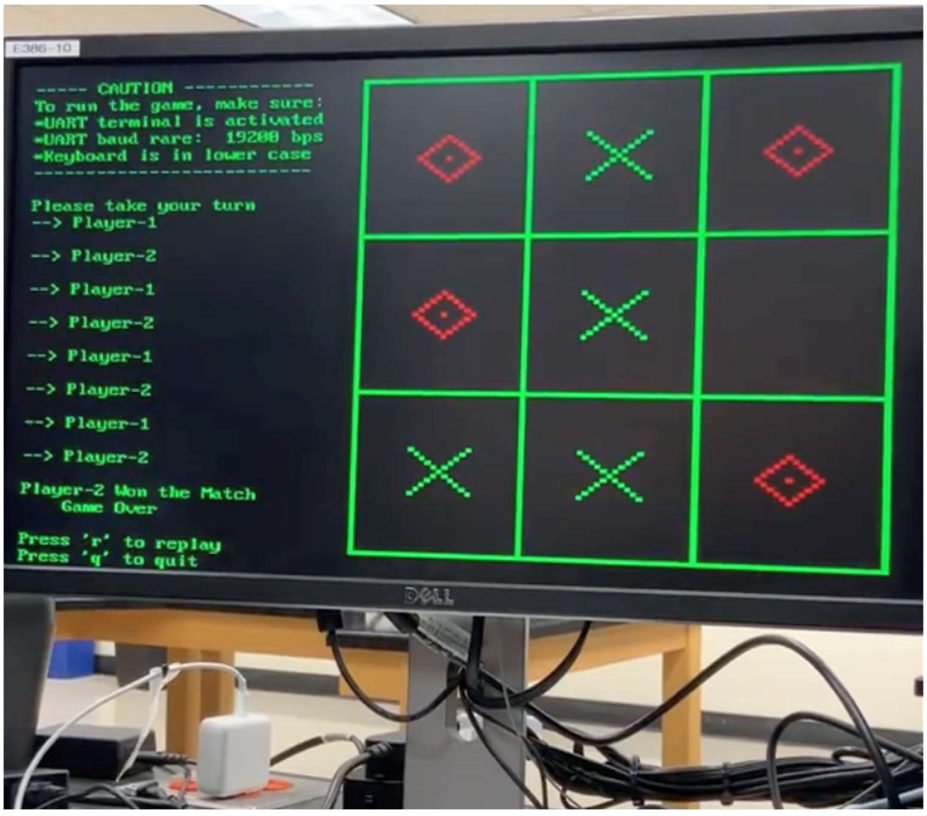


Figure 18. Player-2(GREEN) winning condition

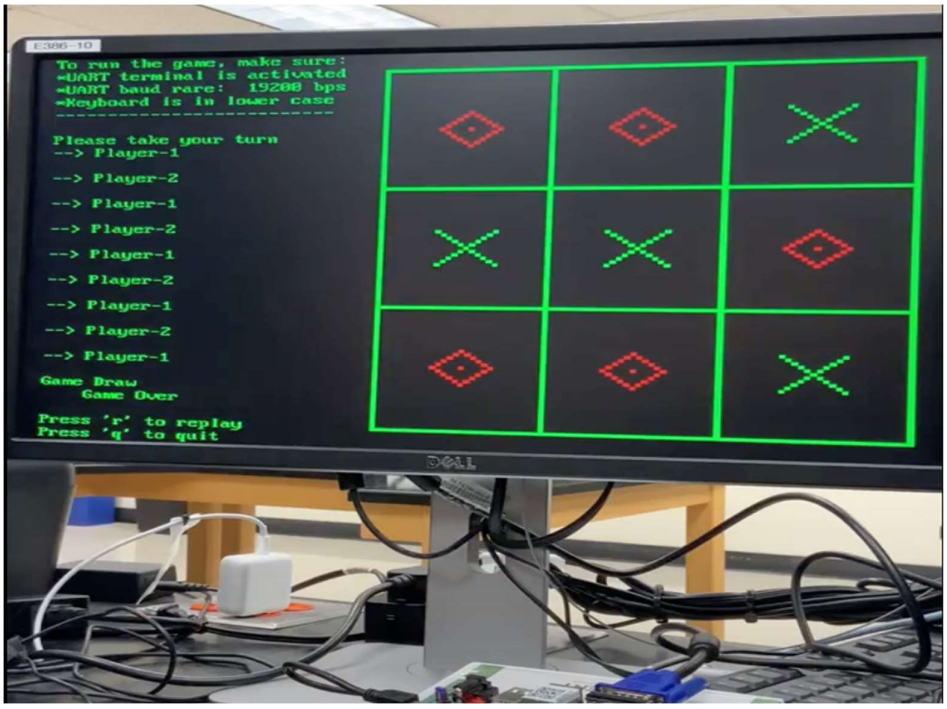


Figure 19. Player-2(GREEN) winning condition

5. CONCLUSION

In this project, we have implemented a two-player tic-tac-toe game. The colour of one player should be red, and the other should be green. The score is maintained for both the players. Points are added to the score of the player who occupies three column in a row. If all the cells on the grid are filled, and neither player has achieved a winning condition, the game is considered a draw or a tie. The winning player should be displayed with the final score. C 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 labour 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.

All the project files and video recording can be found in the link below:
https://drive.google.com/drive/folders/1YnytWMUSdkiYxxjQIdpzMSmfT9lF_xh5?usp=share_link