



# Danmarks Tekniske Universitet

02321 Hardware/Software Programming

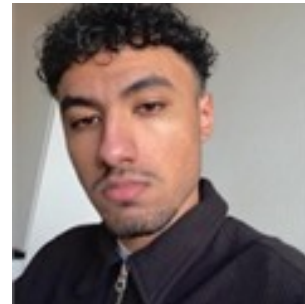
3 Weeks Project: Tetris



Younes Humadi  
S235771



Benjamin Hadziosmanovic  
S235764



Ali Walid al-Sayad  
S235792

Januar 23, 2025

# Contents

<b>1</b>	<b>Contribution to the project work</b>	<b>3</b>
<b>2</b>	<b>Artificial Intelligence</b>	<b>3</b>
<b>3</b>	<b>Requirement specification</b>	<b>3</b>
3.1	Initial Objective . . . . .	3
3.2	Minimum Requirements . . . . .	3
3.3	Initial Extensions . . . . .	4
3.4	Changes to Requirements . . . . .	4
3.5	Refined Requirements . . . . .	4
<b>4</b>	<b>Analysis</b>	<b>5</b>
4.1	Object review . . . . .	5
4.2	Technical analysis . . . . .	5
4.3	Comparison . . . . .	5
4.4	Changes and Impact . . . . .	6
<b>5</b>	<b>Risk Analysis</b>	<b>7</b>
<b>6</b>	<b>Design</b>	<b>8</b>
6.1	Block Diagram for Block Design . . . . .	9
6.2	Flowdiagram overall . . . . .	11
6.3	Flowdiagram for Overall Functionality . . . . .	13
6.4	Flowchart for Gameplay . . . . .	15
6.5	Flowchart for Button Mapping . . . . .	17
6.6	Flowchart for SD Card Image . . . . .	18
<b>7</b>	<b>Implementation</b>	<b>19</b>
7.1	Overview of Implementation . . . . .	19
7.2	System Initialization . . . . .	19
7.3	Drawing into the Monitor . . . . .	23
7.4	Game Logic . . . . .	25
7.4.1	Collision check . . . . .	25
7.4.2	Spawn block . . . . .	26
7.4.3	Clear Row . . . . .	27
7.4.4	Score Box . . . . .	27
7.5	Rendering . . . . .	28
7.6	Challenges and Solutions . . . . .	29
<b>8</b>	<b>Tests</b>	<b>30</b>
<b>9</b>	<b>Discussion</b>	<b>30</b>
<b>10</b>	<b>Conclusion</b>	<b>31</b>
<b>11</b>	<b>User manual</b>	<b>31</b>
11.1	Bootng from SD . . . . .	32
11.2	Bootng from SDK . . . . .	32

# 1 Contribution to the project work

As a group, we have been really good at collaborating on the various tasks, but there have been periods where specific people have focused on individual tasks. Benjamin has had a little more focus on the Drawing to the monitor and sd card than the others, Ali has had more focus on design and Younes has had more focus on the breadboard itself and its effect on our game. However, the rest of the project has been group work that took place both online and at school. We have used the expertise and experiences of the various group members to build our project.

## 2 Artificial Intelligence

The entire project used Artificial Intelligence as a helpful tool to facilitate the smooth running of processes and improved development outcomes. It supported the us with technical insights, refining ideas, and helping in complex problem-solving scenarios. AI was very helpful in optimizing core functionalities, such as rendering logic, collision detection, system integration, but also smoothing out the debugging and troubleshooting processes. This was also contributory to the shaping of the documentation, with suggestions on how to make it clear and coherent, which ensured that the report communicated well the goals and achievements of the project. With AI in application, the development process was much quicker and contributed toward the quality and reliability of the final product.

## 3 Requirement specification

Before we started with our current project, we had to develop a plan and a project description for the game that we wanted to create. The description outlined the following objectives and requirements:

### 3.1 Initial Objective

Here the aim was to create a fully functional tetris game with:

- Display on monitor via. HDMI
- Real time user interaction with zybo buttons, zybo switch and or huzzah buttons
- Core Tetris functionalities: block movements, rotations, line clearing, and score-tracking

### 3.2 Minimum Requirements

Our minimum requirements for our game were the following:

- Two IP cores
  - HDMI IP cores
  - Custom tetris logic for game rules
  - Other IP cores

- Hardware interrupts for responsive user input.
- At least one I/O device
  - GPIO for button, LED's, switch and controller
  - HDMI output for game display
  - We will use UART and GPIO as I/O devices
  - CDIO element would possibly come from Introduction to Statistics or Operating Systems

### 3.3 Initial Extensions

Video and Audio with possibly Ethernet

Our own IP cores

Advanced C code

### 3.4 Changes to Requirements

We initially presented our requirements, but as the project progressed, our understanding evolved, and some requirements were modified. In most cases we felt we needed to change the requirements based on the new information that we got throughout the coding experience. Therefore some of our requirements have been changed and more requirements were added. The following are the changes and additions to the initial requirements.

### 3.5 Refined Requirements

There were no major changes in the initial objective of the project, other than adding some functionalities such as: ghost block, fast switch, instant landing and so on. However instead of the huzzah board we used a breadboard, which we connected to the zybo board. For the HDMI part, we also added the small requirement of being able to boot up the game by only using the zybo and sd card.

As for the minimal requirements we added some changes and added addition logic. The first change was the implementation of the Custom tetris logic for game rules. We removed the requirement, because our understanding of the project was limited to our current understand, and for this reason we removed this requirement. However instead of this we added the requirement of additional buttons for the controller buttons. Meaning that we had to change the given HDMIOUT Vivado block design (which only handled 4 buttons), to handle an extra of 5 buttons. We did not create an extra ip core however we added an extra input port for the btn\_axi\_GPIO called **ctrl.btns**. One minor change that we added was the removal of the LED's as we figured that they were practically useless for our project. We also added the minimal requirement of double buffering.

For the extensions we realised that we did not need an audio or music for that matter. However we changed this for an SD card. And we did not configure Ethernet. Again the need for an extra IP core was practically baseless since we just added an extra port for

the new buttons. However we could have added an extra AXI GPIO for the new buttons, however the ports, for us, was a better option. Also we added the extension of booting the game from an SD card in the zybo.

## 4 Analysis

As we have cleared up the changes and additions to our tetris project, we can summarize and fully understand the objective for our 3 weeks project.

### 4.1 Object review

The main objective of the project was to create a fully functioning Tetris game that mimicks the real Tetris game or at least mostly looks and functions like the actual game. To achieve this, we agreed on three main objectives: HDMI output, user interaction, and core Tetris functionalities. These were essential to our project and achieving our task of creating the Tetris game. By making these 3 requirements our main task, we would achieve our goal. To achieve this we would also create the minimal requirements that would help us achieve our goal. Included in these minimal requirements were the usage of HDMI IP cores in Vivado, hardware interrupts and I/O devices. Finally the last objective was to ensure we had Video, Sd card image display and advanced C code designed to fit the project.

### 4.2 Technical analysis

This section discusses the challenges we encountered during the project and the solutions we implemented. Our biggest challenge was implementing the boundary/collision check for falling blocks. This meant ensuring that blocks couldn't go beyond the grid lines, fall below the lowest grid point, or fail to detect already-landed blocks. However after a few tries and constantly failing we finally got it right and the functionality worked as expected.

Another problem we encountered was the audio output part of our game. Initially, we planned to include audio; however, as mentioned earlier, we decided it was unnecessary for the project. and instead chose to display an start game image and a in game image by using the SD card.

### 4.3 Comparison

This section covers the comparison between the objective and requirements and the final product. The following are descriptions and explanations of whether we succeeded in implementing each requirement:

- We displayed on monitor via HDMI
- User interaction with btns, switch and controller buttons
- Core tetris functionalities, included but not limited to
  - Block falling

- Rotation
  - Collision check
  - Line clearing
  - Score tracking
  - Ghost block
  - Fast falling
  - Instant falling
  - And much more
- Btms and switches for input and HDMI for output.
- UART for connection between pc and zybo
- CDIO elemtns
- Double buffer
- Video display
- SD card image display
- Custom input port for btms
- Advanced C code
- Boot from zybo

As seen above all of our requirements were met and this resulted in what we believe to be a fully functioning Tetris game.

## 4.4 Changes and Impact

Throughout the project, several changes were made to the initial requirements to adapt to new challenges and insights. One of the most important changes was the scrapping of the audio functionality to allow for the implementation of SD card support for displaying game images. This decision improved the visual appeal of the game and streamlined development but meant sacrificing an initially planned feature.

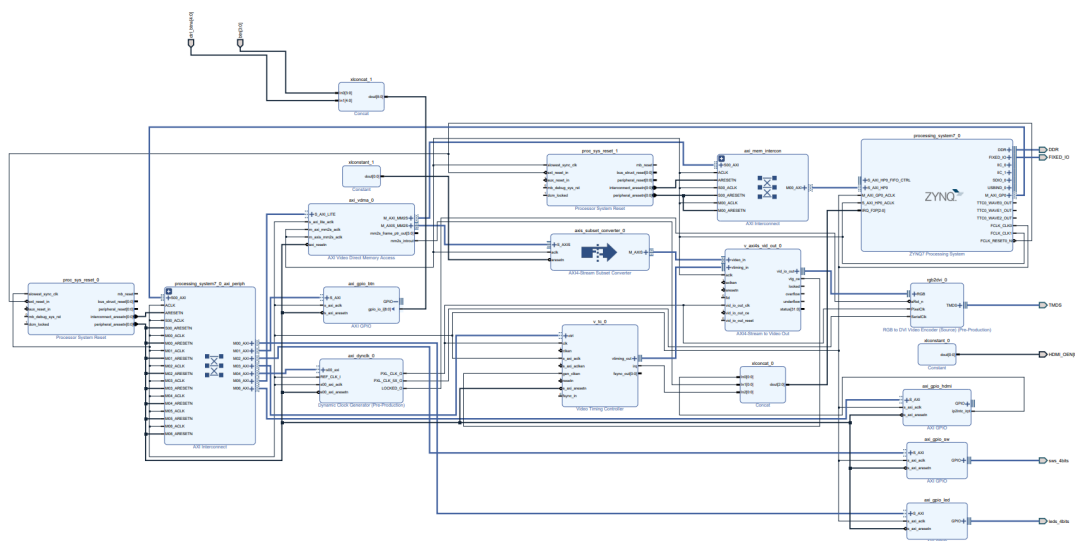
Other important changes were the extension of the GPIO configuration to allow more buttons without the need to design new IP cores. In this way, unnecessary complexity is avoided, and integration with the already existing system is smoother. This allowed us to present a more refined and functional product while still keeping the project within the confines of hardware and time constraints.

## 5 Risk Analysis

What can go wrong	How likely is it to go wrong	How serious are the consequences if it goes wrong	Overall risk
HDMI output stops working	Medium – Minor misconfigurations in Vivado settings or wiring issues could cause this.	High – Game display will not function	High
Button inputs are unresponsive	Low – GPIO configuration and hardware are generally stable.	High – Gameplay would be unplayable	Medium
Collision detection fails	Medium – Complex logic for boundary checks can introduce bugs.	High – Game logic would break	High
SD card fails to load images	Low – SD card integration was tested and is straightforward.	Medium – Affects visual appeal but not core gameplay	Low to Medium
Interrupts fail to trigger	Low – Interrupts are well-supported in the chosen architecture.	High – Delays in gameplay responsiveness	Medium
Double-frame buffer causes tearing	Low – Implemented buffer switching has been tested.	Medium – Visual quality is reduced	Low to Medium
Power supply instability	Low – Power supply is reliable if connections remain intact.	High – Hardware might shut down	Medium to High

Table 1: Risk Analysis

At the start of the project we added the given handout meaning the HDMI OUT Vivado project. We added the project to our Vivado and started analysing the different components in the project. This included the blockdesign, xdc and wrapper file. For each part we thoroughly read and understood the contents, such as each IP core in the blockdesign and the available pins in the .xdc file We changed the blockdesign slightly to include 9 buttons instead of 4 buttons. We achieved this by adding an extra port **ctrl\_btns** and a concat.



The block design for the HDMI output system of the Zybo board couples the ARM-based processing system with video and GPIO peripherals. The main elements are rgb2dvi IP for the HDMI conversion, v\_axi4s\_vid\_out for the video data, and v\_tc for the timing control. The block design also includes AXI GPIO modules, which are configured to handle input from 9 buttons connected to the board. These provide user control for gameplay, with each GPIO port mapped to a unique button input. The AXI GPIO will make for seamless integration with the ARM processing system for responsive and reliable input handling. This was an important configuration for the implementation of gameplay mechanics effectively.



## 6.1 Block Diagram for Block Design

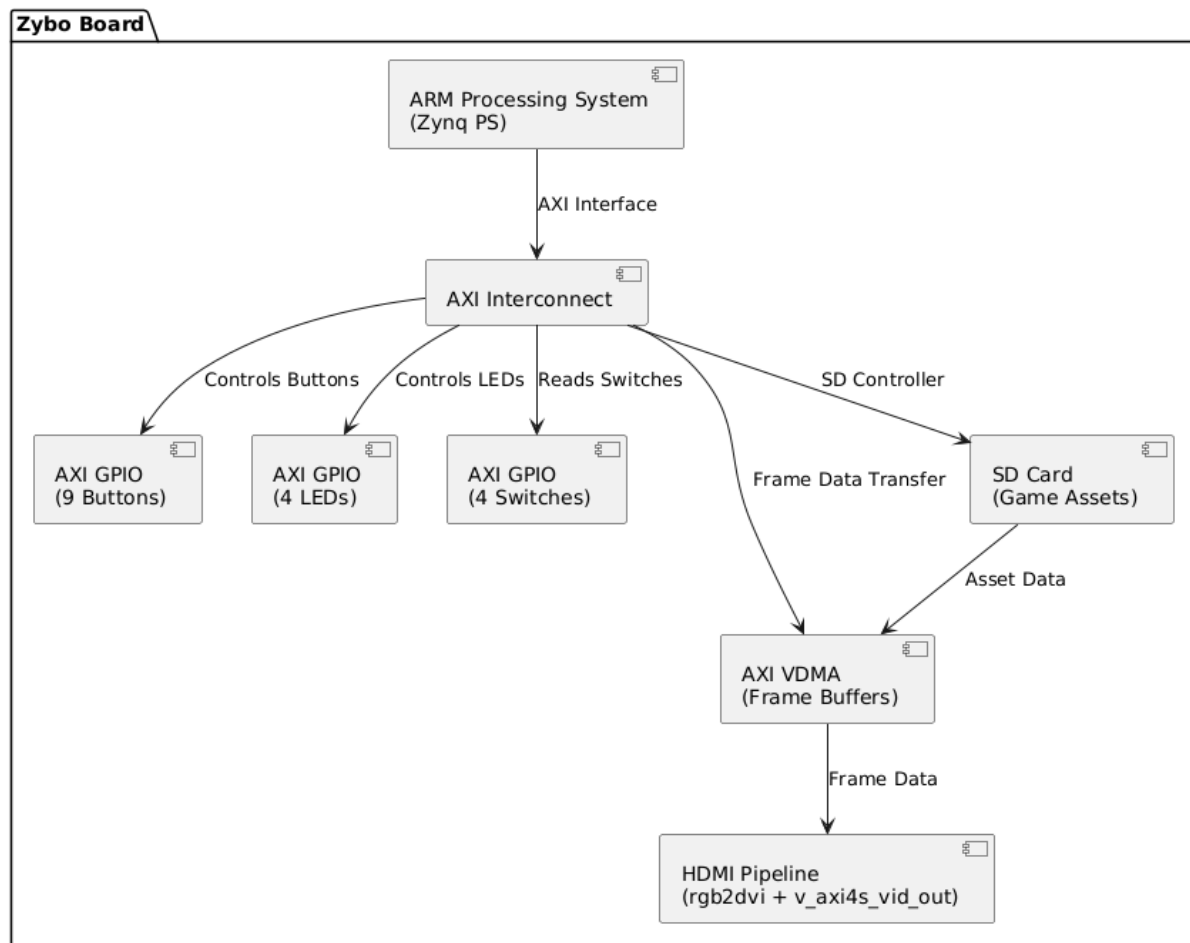


Figure 2: Caption

In This block diagram we see a complete representation of the architecture in a system that's implemented on a Zybo Board. The zybo boards integrates an ARM Processing System, known as Zynq PS, with programmable logic through an AXI Interconnect. The heart of the system is based on the Zynq Processing System, which plays the main controller role in managing data flow and communicating with different peripheral components. These peripherals are further interfaced with the processing system using an AXI interconnect, which provides a high-bandwidth bridge for efficient communication. The system consists of several peripheral components connected via AXI GPIO interfaces. These include a 9-button module for the reception of user input, a 4-LED module for displaying output signals or status, and a 4-switch module-which provides additional control inputs, such as toggles. Each of these peripherals provides some way for the user to interact with the system, and the Zynq PS processes the corresponding input or output data.

The controller also includes an SD card controller to enable data exchange between the system and the SD card.

Another important part of the design is the AXI VDMA that will be responsible for the transfer of frame data from memory to the display pipeline. The VDMA acts as a bridge to fetch the frame buffers held in the memory and then push them out to the HDMI

pipeline. The HDMI pipeline consists of two modules: `rgb2dvi`, which converts the frame data into HDMI-compatible signals, and `v_axi4s_vid_out`, which sends the video output processed out to an external HDMI-compatible display.

From there, the general flow of the data starts with control signals from buttons, switches, and LEDs. Those are dealt with by the Zynq PS over the AXI Interconnect. From there, frame data gets processed and passed along the HDMI pipeline, which will yield a graphical representation on the monitor's end. This design would be perfect for an interactive application or game; the buttons and switches provide user input, LEDs indicate status, and the HDMI pipeline outputs graphical content based on the input and asset data.

## 6.2 Flowdiagram overall

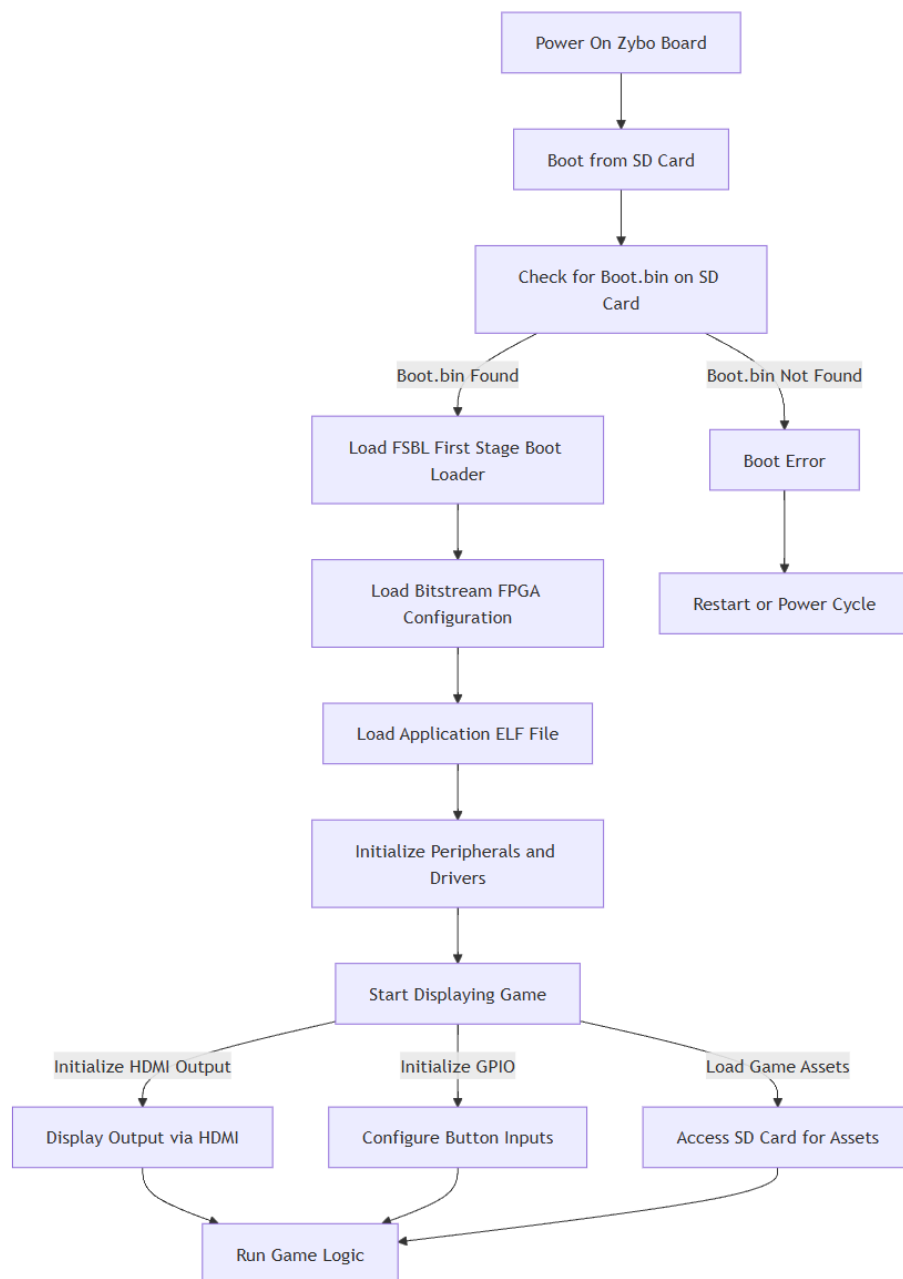


Figure 3: Caption

This flowchart illustrates the Zybo Board's boot and initialization process for a game application. It is further elaborated here:

first we have to boot, the Zybo Board attempts to start up using an SD card. It searches the SD card for 'boot.bin', a file that contains all the necessary bootloader and configuration data. If 'boot.bin' is not present, then the system cannot boot and the user must take action: that is, a restart or power cycle to the board to attempt again.

The 'boot.bin' initiates the booting process of the \*\*FSBL: First Stage Boot Loader, and then the FSBL will start. The FSBL mainly initializes the board hardware and also prepares a bitstream file that needs configuration within an FPGA. Further, this is

followed by the loading of an **\*\*Application ELF file, which\*\*** carries the main application executable code when the FPGA has finished its configuration.

Following the ELF of the application loading, the system then initializes all required peripherals/drivers: GPIO for user interaction, SD card interface, and HDMI output. Following this initialization, the system reaches its main state; first, it will show the game in its running condition.

It now performs three simultaneous actions: initializing the HDMI output to display game graphics on an attached monitor; configuring the GPIO to handle button inputs from a user; and access to the SD card for loading game assets such as textures, levels, and other data that may be requested by an application.

HDMI output, GPIO are the inputs with game assets being treated as one too; these go to the game logics that runs in an unbroken loop for the entire session of game - processing user input, changing game states and rendering resulting graphics that would go via HDMI.

This flowchart summarizes the successive boot process and parallel operations involved in running a graphical application, such as a game, on the Zybo Board. The SD card serves as a storage device for the bootloader, configuration, and game assets, and there is much interaction between the initialization of hardware and real-time execution of the game.

### 6.3 Flowdiagram for Overall Functionality

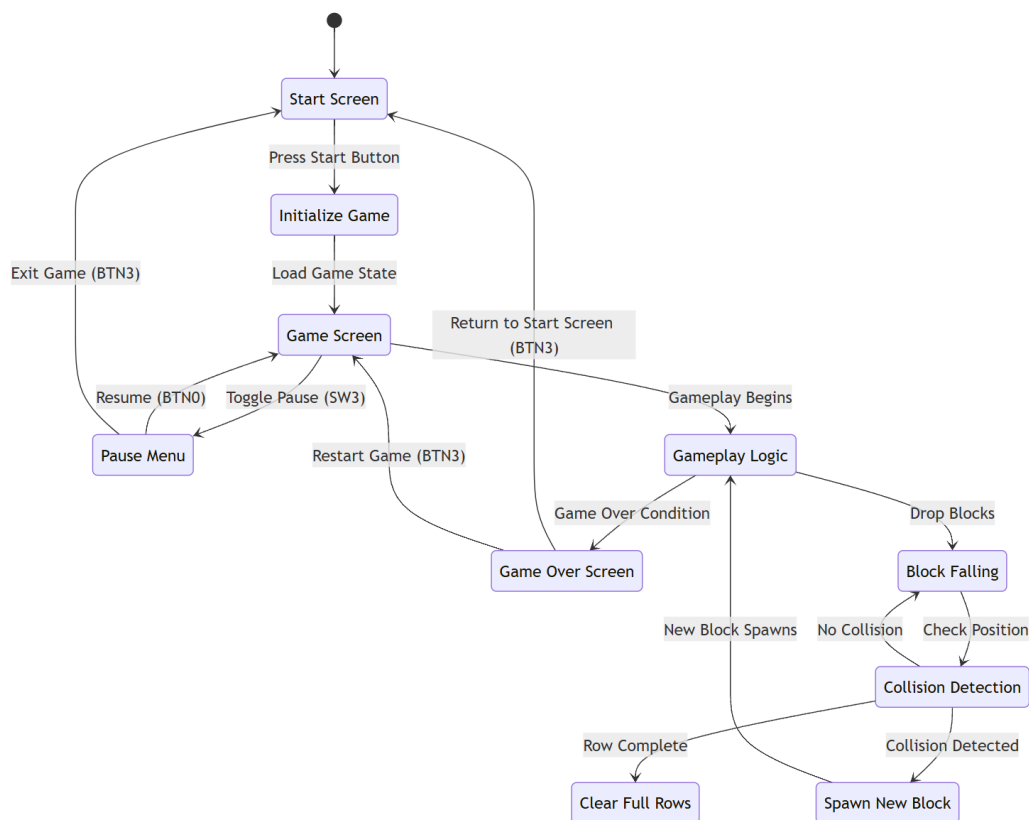


Figure 4: Caption

This flowchart describes the gameplay flow of the Tetris game, implemented on the Zybo Board. The process flow of the game can be described below in detail:

The game begins on the Start Screen, which includes a start button that the player activates to begin the game. This opens the game, prepares the environment, and loads the previous or default state of the game. The game now transitions to the Game Screen, at which point active gameplay occurs.

From the Game Screen, the player can end

**Pause Game:** At each switch on/off of a certain switch, SW3, the game goes into a pause menu. In pause, the player is able to resume the game using BTN0 or perform other actions associated with the pause. **Restart Game:** The player is able to restart the game using BTN3 and go back to the start of the current session. **Exit Game:** BTN3 also is used to exit the game into the Start Screen.

**Gameplay Logic:** The basic treatment and interaction of the blocks in the game emanates from here. It starts with the gameplay state, which deals with logics such as drop block, detects collision, manages states of the game.

**Falling Blocks:** Blocks are constantly falling downwards while checking for their collision with other blocks or the bottom of the game field. **Collision Detection:** If no collision was detected, then the block continued to fall. On detecting a collision, it locks the block and at the top of the screen, a new block is spawned. If in a process of a row without gaps is filled, then this row is being cleared from a field and the player progress or score updated. The game will then enter an infinite loop from the above steps until reaching a

Game Over Condition, which would include the inability to spawn new blocks anymore since the top contains blocks blocking the way. At this stage, the game will head into the Game Over Screen where it goes back to the Start Screen via BTN3 to fresh-start. The following flowchart will show how the game states, player interaction, and core game-play mechanics come together in a single, fluid process to create a structured interactive experience.

## 6.4 Flowchart for Gameplay

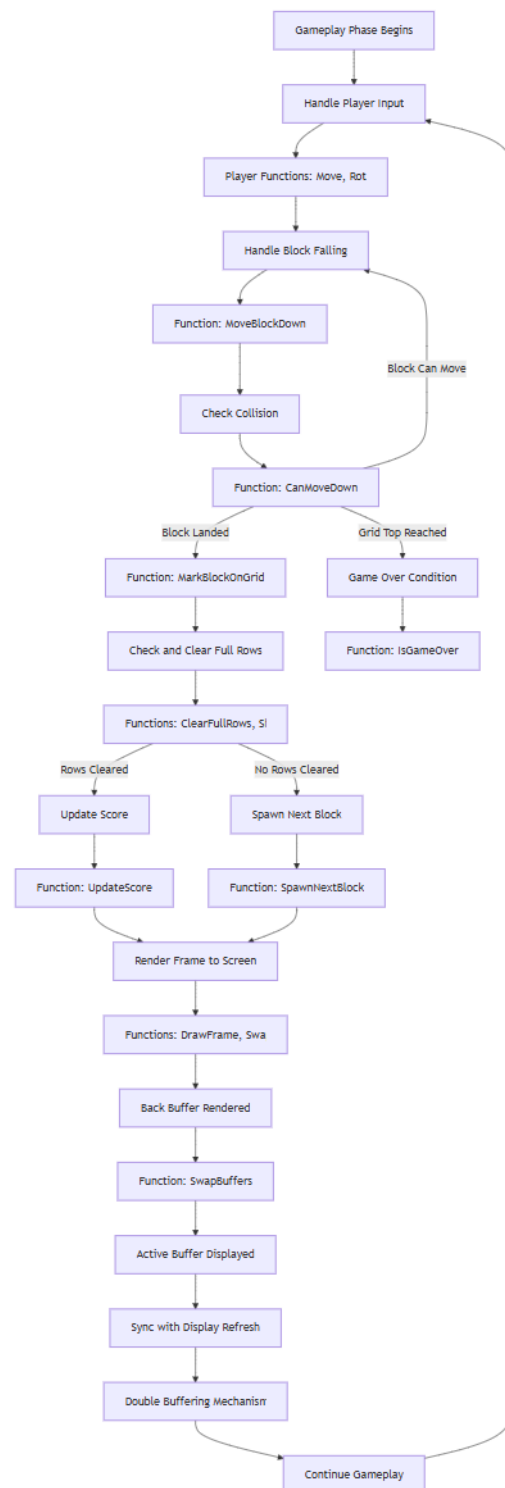


Figure 5: Caption

This flowchart represents the operational flow for tetris on a Zybo Board: from the **\*\*Start Screen\*\*** to gameplay, including pauses, restarts, and game-over scenarios.

The game will start on the **\*\*Start Screen\*\***; a player can push the start button to initialize the game. On initialization, the game sets up the environment and loads in the

game state, transitioning to the **Game Screen** where active gameplay occurs.

On the **Game Screen**, the player may take several actions: 1. The player can **pause the game** using the designated switch (SW3). This takes the player to the **Pause Menu**, where they can either resume gameplay via BTN0 or stay paused while analyzing the current state. 2. If the player presses BTN3, he will **restart the game** to its previous state and go back to the initial phase of the game. 3. The player can also use BTN3 to **exit the game** and be taken back to the Start Screen, which would end the session in course.

Next, it executes the **Gameplay Logic**, where the entire control of moving blocks and the progress of the game takes place. First, there is a continuous falling of blocks onto the field of the game: - **Block Falling**: Blocks fall incrementally downwards; their position is regularly checked for possible collisions. - **Collision Detection**: If there is no detection of collision, the block keeps falling. Upon detection of collision-either with other blocks or the bottom of the game field-it will lock. A new block then spawns at the top of the screen to continue the gameplay. - **Row Completion**: As blocks continue to pile up, the completely filled rows disappear from the playing field. This counts for the player's score or progress.

Game logics run in a loop until **Game Over Condition** is achieved: Once spawning of new blocks is blocked - the field is full, then the game is moved to **Game Over Screen** where a player is informed that the game is over. On Game Over Screen press of BTN3 takes player to the Start Screen for a new session.

The above flowchart focuses on how best and interactively to structure this gameplay through balanced user and automated game-playing logics that give a great flair for its dynamics.



## 6.5 Flowchart for Button Mapping

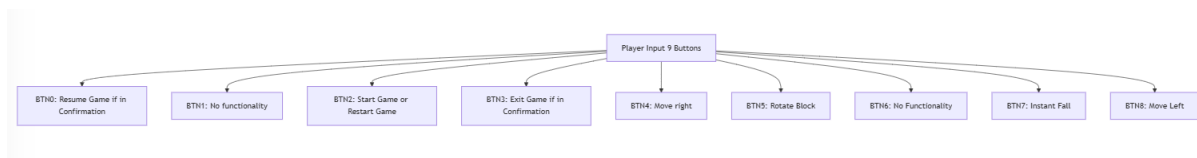


Figure 6: Caption

This is a smaller flowchart or diagram that shows what each button does.

- btn0: Resume game if in Confirmation
- btn1: No functionality
- btn2: Start game or restart game
- btn3: Exit game if in Confirmation
- btn4: Move right
- btn5: Rotate block
- btn6: No functionality
- btn7: Instant Fall
- btn8: Move left

## 6.6 Flowchart for SD Card Image

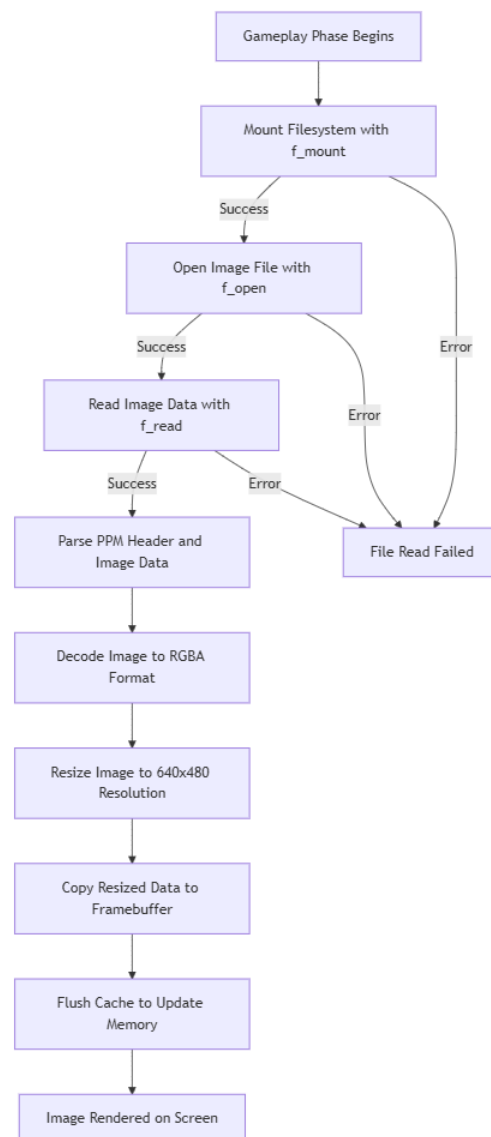


Figure 7: Caption

The sd flowchart shows mostly the functionality of the function **LoadAndDisplayImage**. This function mainly includes the functions from the Fatfs: Generic FAT Filesystem Module. This part happens after the game has been booted up and is initialized. First it mounts the filesystem by using the `f_mount` function, then if there is an error, the file read fails, however if there is success the image file opens with the `f_open` function. If the image file is opened and does not fail, which will lead to failure, the image data is read using the `f_read` function. Then if everything is good, the system parses the PPM header and Image data, which then the system decodes the Image to RGBA format, which is for displaying the image on the monitor. After the image is resized to 640x480 resolution, which we have picked based on the resolution of the monitor. We then copy the resized data to the framebuffer, flush the cache and the image renders to the screen. This function is used for our 2 images that we use, the start screen image "start3.ppm" and "ingame.ppm".

## 7 Implementation

This block design was the basis for the rest of the project, which included the c code implementation, however before we get to the implementation we need to understand the structure of the C code, which we have written in Vivado SDK.

### 7.1 Overview of Implementation

The implementation of our project was split into several areas. These included the Vivado Blockdesign part, enabling the buttons and pins in the .xdc file, coding the actual game in Vivado SDK and making sure they were compatible with each other. That is, the SDK was based on the Vivado blockdesign and project, so our game was built on layers with the Vivado part being the first block and SDK C code being the second layer. And the SD card BOOT.BIN being the top layer.

For starters the bitstream file was created in the Vivado project, then the hardware (wrapper and bitstream), was exported to the SDK project. From here the SDK could use the hardware design from the Vivado to fx, launch on HDMI, configure buttons and interact with the SD card and more. This enabled us to combine the Vivado and SDK for our game. Proceeding with our implementation we could create our functions and game logic and much more in the SDK.

### 7.2 System Initialization

We modified the block design and Vivado later in the project, after nearly completing the Tetris game. The changes in the Vivado included the addition of a concat IP and a new input port named **ctrl\_btns**. This handled the configuration of the new buttons on the controller. This would be connected to the axi\_gpio\_btn, which configured the btns. Firstly the AXI GPIO was changed to handle 9 inputs instead of 4, and the concat was used to combine the 2 ports into a single bus to then be connected to the AXI GPIO.

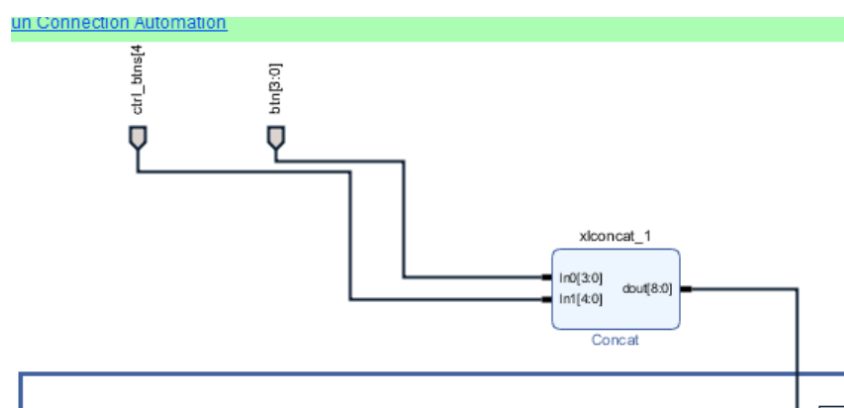


Figure 8: Caption

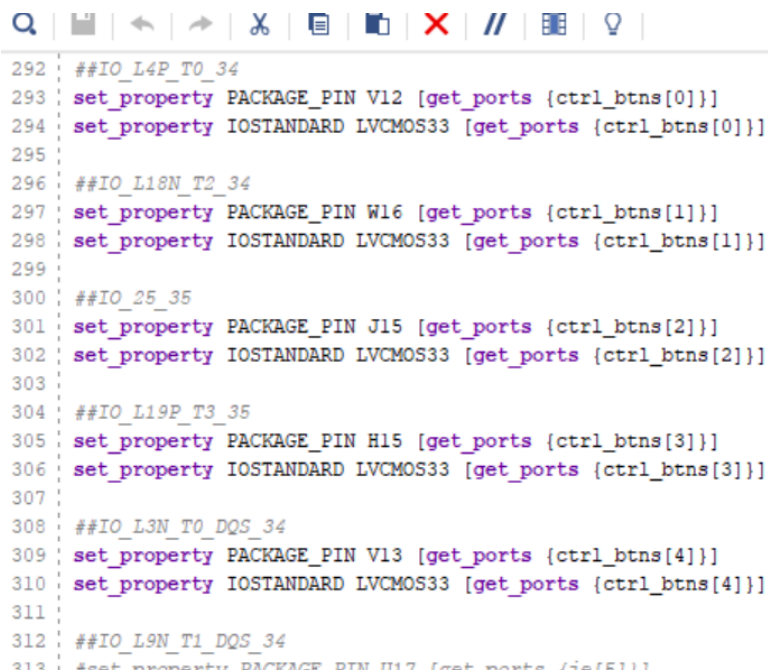
The next step was to edit the **ZYBO\_Master.xdc** which is the constraints file used in the Vivado project. Here we enabled the 4 btns and the 5 pins which corresponded to the 5 buttons:

```

28
29 ##Buttons
30 ##IO_L20N_T3_34
31 set_property PACKAGE_PIN R18 [get_ports {btn[0]}]
32 set_property IOSTANDARD LVCMOS33 [get_ports {btn[0]}]
33
34 ##IO_L24N_T3_34
35 set_property PACKAGE_PIN P16 [get_ports {btn[1]}]
36 set_property IOSTANDARD LVCMOS33 [get_ports {btn[1]}]
37
38 ##IO_L18P_T2_34
39 set_property PACKAGE_PIN V16 [get_ports {btn[2]}]
40 set_property IOSTANDARD LVCMOS33 [get_ports {btn[2]}]
41
42 ##IO_L7P_T1_34
43 set_property PACKAGE_PIN Y16 [get_ports {btn[3]}]
44 set_property IOSTANDARD LVCMOS33 [get_ports {btn[3]}]
45

```

Figure 9: Caption



```

292 ##IO_L4P_T0_34
293 set_property PACKAGE_PIN V12 [get_ports {ctrl_btns[0]}]
294 set_property IOSTANDARD LVCMOS33 [get_ports {ctrl_btns[0]}]
295
296 ##IO_L18N_T2_34
297 set_property PACKAGE_PIN W16 [get_ports {ctrl_btns[1]}]
298 set_property IOSTANDARD LVCMOS33 [get_ports {ctrl_btns[1]}]
299
300 ##IO_25_35
301 set_property PACKAGE_PIN J15 [get_ports {ctrl_btns[2]}]
302 set_property IOSTANDARD LVCMOS33 [get_ports {ctrl_btns[2]}]
303
304 ##IO_L19P_T3_35
305 set_property PACKAGE_PIN H15 [get_ports {ctrl_btns[3]}]
306 set_property IOSTANDARD LVCMOS33 [get_ports {ctrl_btns[3]}]
307
308 ##IO_L3N_T0_DQS_34
309 set_property PACKAGE_PIN V13 [get_ports {ctrl_btns[4]}]
310 set_property IOSTANDARD LVCMOS33 [get_ports {ctrl_btns[4]}]
311
312 ##IO_L9N_T1_DQS_34
313 set_property PACKAGE_PIN H17 [get_ports {ctrl_btns[5]}]

```

Figure 10: Caption

After we exported the bitstream we had to initialize the components. This was done the same way as the project 1 earlier in the course. The hardware was initialized in the "hardware\_init.c and interrupts.c respectively. The buttons and switches are initialized using this part in those file:

Listing 1: Code snippet of c code

```
// Initialize buttons as input
XGpio_Initialize(&BTNInst, BTNS_DEVICE_ID);
XGpio_SetDataDirection(&BTNInst, 1, 0xFF); // Configure all
      button bits as input

// Initialize switches as input
XGpio_Initialize(&SWInst, SWITCHES_DEVICE_ID);
XGpio_SetDataDirection(&SWInst, 1, 0xFF); // Configure all switch
      bits as input
```

In interrupts.c, GPIO interrupts are configured to handle button presses. The IntcInitFunction() connects the GPIO interrupt handler (BTN\_Intr\_Handler) and enables global interrupts.

Listing 2: Code snippet of c code

```
// Connect GPIO interrupts
status = XScuGic_Connect(&INTCInst, INTC_GPIO_INTERRUPT_ID,
                        (Xil_ExceptionHandler) BTN_Intr_Handler,
                        (void *) &BTNInst);

// Enable GPIO interrupts
XGpio_InterruptEnable(&BTNInst, BTN_INT);
XGpio_InterruptGlobalEnable(&BTNInst);
```

To integrate the hardware setup with the game logic, the buttons and switches are initialized in the main file using the XGpio\_Initialize function. This step ensures the software can access the GPIO configurations defined in Vivado. Proper error handling is implemented to detect initialization failures, providing clear feedback during debugging.

Listing 3: Code snippet of c code

```
// Initialize Button GPIO
int status = XGpio_Initialize(&BTNInst,
    XPAR_AXI_GPIO_BTN_DEVICE_ID);
if (status != XST_SUCCESS) {
    xil_printf("Button_GPIO_Initialization_Failed\r\n");
    return XST_FAILURE;
}
XGpio_SetDataDirection(&BTNInst, 1, 0xFF); // Set buttons as
      input

// Initialize Switch GPIO
status = XGpio_Initialize(&SWInst, XPAR_AXI_GPIO_SW_DEVICE_ID);
if (status != XST_SUCCESS) {
    xil_printf("Switch_GPIO_Initialization_Failed\r\n");
    return XST_FAILURE;
}
```

```
XGpio_SetDataDirection(&SWInst, 1, 0xFF); // Set switches as
input
```

The first 4 buttons are accessed using XGpio\_DiscreteRead on channel 1. Each button is mapped to specific actions in the game, such as starting the game or toggling states. Debounce logic is applied to ensure actions are only triggered once per button press.

Listing 4: Code snippet of c code

```
if (!simulationStarted && !gameOver) {
    int btn_value = XGpio_DiscreteRead(&BTNInst, 1);
    double elapsedButtonTime = ((double)(currentTime -
        lastButtonTime)) / COUNTS_PER_SECOND;

    if (btn_value & 0x2 && elapsedButtonTime >= DEBOUNCE_TIME) {
        // BTN1 toggled
        ShowStartScreen(stride, &currentLevel);
        lastButtonTime = currentTime; // Update button debounce
        time
    }
}
```

The extra 5 buttons are accessed using a masked gpioData read. Each bit in the mask corresponds to a specific button, allowing for distinct actions like moving blocks or rotating them. The debounce mechanism ensures reliable input handling, even if a button is pressed multiple times in quick succession.

Listing 5: Code snippet of c code

```
// Button Debounce Logic
double elapsedButtonTime = ((double)(currentTime - lastButtonTime
)) / COUNTS_PER_SECOND;
if (elapsedButtonTime >= DEBOUNCE_TIME) {
    if (gpioData & 0x10) { // BTN0: Move Right
        if (CanMoveHorizontal(blockX, blockY, currentShape,
            currentRotation, 1)) {
            blockX += BLOCK_SIZE;
        }
    }
    if (gpioData & 0x100) { // BTN2: Move Left
        if (CanMoveHorizontal(blockX, blockY, currentShape,
            currentRotation, -1)) {
            blockX -= BLOCK_SIZE;
        }
    }
    if (gpioData & 0x20) { // BTN1: Rotate Block
        int nextRotation = (currentRotation + 1) % 4;
        if (CanRotate(blockX, blockY, currentShape, nextRotation)
        ) {
            currentRotation = nextRotation;
        }
    }
    lastButtonTime = currentTime; // Update button time
}
```

This initialization and handling of buttons and switches form the backbone of user interaction in the Tetris game. By combining GPIO reads, debounce logic, and dedicated actions for each button, the system ensures responsive and precise control during gameplay. The integration of extra buttons expanded functionality, allowing for more intuitive and advanced gameplay mechanics.

DemoInitialize initializes the screen, and sets up hardware including the configuration of frame buffers that will be needed. It populates the frame pointers to reference allocated frame buffers, which hold pixel data during rendering. Similarly, a call is made to retrieve the video DMA (VDMA) configuration via XAxiVdma.LookupConfig and initializes the VDMA with its configuration. An error message will appear if configuration or initialization goes wrong. This program begins with the initialization of the display controller, using the function DisplayInitialize to set up the display including VDMA, a timing controller, dynamic clock, and frame buffers. Finally, it starts the display through DisplayStart, allowing the system to start rendering onto the screen. Failure in any one of these cases will lead to the appropriate output of error messages showing the issues that occurred.

### 7.3 Drawing into the Monitor

After the initialization process we could finally start to draw to the screen, which would be essential to our project, here we used a variety of functions which helped us to complete our task. The main drawing function which all other drawing functions use is the DrawToTheMonitor function, which is the function that all other drawing functions use.

Listing 6: Code snippet of c code

```
void DrawToTheMonitor(u8 *frame, u32 xStart, u32 yStart, u32
    width, u32 height, u32 stride, u8 red, u8 green, u8 blue)
{
    u32 xcoi, ycoi;
    u32 iPixelAddr;

    for (ycoi = yStart; ycoi < yStart + height; ycoi++)
    {
        for (xcoi = xStart; xcoi < xStart + width; xcoi++)
        {
            iPixelAddr = (xcoi * 4) + (ycoi * stride);
            frame[iPixelAddr] = red;
            frame[iPixelAddr + 1] = green;
            frame[iPixelAddr + 2] = blue;
        }
    }
}
```

In cases where a given rectangle should be rendered to any of the monitors using drawing, which goes straight into a frame buffer, one would consider taking a frame buffer pointer/frame starting address, then a rectangular starting co-ordinate, with their width-height stride and rgb value as DrawToTheMonitor argument.

This function iterates over the region defined by nested loops, the outer one moving vertically ycoi;, the inner vertically xcoi;. For each input pixel, it calculates the memory address in the frame buffer for the formula:

Listing 7: Code snippet of c code

```
iPixelAddr = (xcoi * 4) + (ycoi * stride);
```

Here, xcoi \* 4 takes care of the 4 bytes RGBA per pixel, and ycoi \* stride aligns the row based on the stride of the frame.

It then sets the red, green, and blue components of the pixel at the computed address by the assignments to frame[iPixelAddr], frame[iPixelAddr + 1], and frame[iPixelAddr + 2], respectively. This would effectively change each pixel of the rectangle to the RGB value specified, allowing for dynamic and precise rendering on the monitor.

An example usage of this function is the DrawBlock function:

Listing 8: Code snippet of c code

```
void DrawBlock(u8 *frame, u32 xStart, u32 yStart, u32 blockSize,
              u32 stride, u8 red, u8 green, u8 blue) {
    if (xStart >= GRID_X_START && xStart + blockSize <=
        GRID_X_START + GRID_WIDTH &&
        yStart >= GRID_Y_START && yStart + blockSize <=
            GRID_Y_START + GRID_HEIGHT) {
        u32 borderThickness = 1; // Thickness of the border

        // Draw the border
        DrawToTheMonitor(frame, xStart, yStart, blockSize,
                        blockSize, stride, 0, 0, 0); // Black border

        // Draw the inner block
        DrawToTheMonitor(frame, xStart + borderThickness, yStart
                        + borderThickness,
                        blockSize - 2 * borderThickness, blockSize
                        - 2 * borderThickness,
                        stride, red, green, blue);
    }
}
```

The DrawBlock function is designed to draw a square block with a border and a filled interior on the frame buffer. It takes as input the frame buffer pointer, the starting coordinates of the block, its size, the stride of the frame, and the RGB values for the color of the inner block. It will check, using starting coordinates and size first, whether the block fits within the grid boundaries that have been predefined.

If so, it defines a border thickness and then calls the DrawToTheMonitor function twice, once to paint the area of the block black - to draw the block's black border - and a second time to paint the block's colored interior by filling in the area of the block interior to the border with the color defined by the RGB values. This will ensure that the block is visually a block, having a boundary and an inner color.



Drawing a block to the screen

## 7.4 Game Logic

The game handles many functions that each has a unique and specifik funtionality, so we will explain a example game and what functions are used.

After the initialization part the game starts on the start screen where ShowStartScreen is called, this function includes the:

- DrawTetris(): Which draws each letter of the Tetris word, then the function calls decide the color and position of the letter.
- DrawLevel(): Which draws each letter of the LEVEL word, then the function calls decide the color and position of the letter.
- LoadAndDisplayImage(): Calls the sd image to display the start3.ppm file in the background of the letters
- DrawDigitLevel(): Draws the next number incementet by the btn1 press

After the btn1 press the in game loads, here most of the important functions happen. For starters the backround from the SD is display, the grid, the Scorebox, the next block box, and the active block is shown on the display.

- LoadAndDisplayImage(): For the ingame.ppm background
- RenderFrame(): Which clears the grid area, redwars the grid, draws the landed and active blocks and draws the score and next block boxes.
- SpawnNextBlock(): Which spawns the block in a random order
- ClearFullRows(): Constantly checks for full rows and if this happens the row is cleared and blocks on top of the cleared row fall

If the player wants to leave the game the sw3 is triggered, the confirmation box appears and the player decides if he wants to exit to the start screen or resume the game.

This was the display part now we focus on what goes on behind the scenes of the display.

### 7.4.1 Collision check

The game handles the blocks in 2 ways how it draws them and how the game understands them, the draw block draws it on the display but the other makes makes it so it now where the blocks are and how they are interpreted. We created a 4x4 grid called SHAPE DATA to represent blocks in all rotations, which is essential for collision detection.. As it tells the other functions how they look and where they are. CanMoveDown(), MarkBlockOnGrid() and CanMoveHorizontal uses this data to understand the blocks current position and can function based on this data.

Logic that runs behind the scene ensures that block positions, block movements, and collisions are performed correctly within a game grid; key functionalities include CanMoveDown(), MarkBlockOnGrid, and CanMoveHorizontal:

### CanMoveDown()

A helper function returning whether a given block can make one step downward in the given grid. That happens based on current position and its current form-the bottom cells are unobstructed and within grid bounds.

Example: If a block at row 4, column 5 wants to move down, CanMoveDown() will check for collision or out-of-bound errors at row 5, column 5.

Listing 9: Code snippet of c code

```
if (grid[row + 1][col] == 1 || row + 1 >= GRID_ROWS) {  
    return 0; // Can't move down  
}
```

### MarkBlockOnGrid()

This function updates the grid by marking the cells that are occupied by a landed block. It also assigns the color of the block to these cells for display-related purposes.

Example: For a block that falls to row 6, column 3, MarkBlockOnGrid() changes grid[6][3] = 1 and the related RGB color values.

Listing 10: Code snippet of c code

```
grid[rowStart + row][colStart + col] = 1;  
gridColors[rowStart + row][colStart + col][0] = SHAPE_COLORS[  
    currentShape][0];
```

### CanMoveHorizontal()

This function shows whether a block can shift left or right by looking into the cells left and right of it. It's checking for collision against the grid and boundary cases.

Example: If a block is at column 5 and wants to go right, CanMoveHorizontal() searches column 6 for any collisions.

Listing 11: Code snippet of c code

```
if (grid[row][col + direction] == 1 || col + direction >=  
    GRID_COLUMNS) {  
    return 0; // Can't move  
}
```

These three functions work all together to achieve the interaction of blocks with the grid properly along with the logic and rules in the game.

## 7.4.2 Spawn block

The SpawnNextBlock() function manages to spawn a new block at the top when the previous block has landed. This function leverages a shuffled "bag" of shapes in order to select up-and-coming blocks for good random balance. Sets random rotation also and puts the block at the initial position top-centered in the grid.

Once all of the shapes have been used, the bag of shapes is reshuffled to keep the variety going throughout the game. The new block is rotated by a random amount from 0° to 270° prior to placement.

### 7.4.3 Clear Row

The `ClearFullRows()` function takes care of the detection and removal of the full rows occurring on the game grid. In case a row is completely filled with blocks, it gets removed, and all the rows above it move down. This function is very crucial in the advancement of the game, along with updating the score.

Then, the function loops through each row in the grid, looking for those that are completely filled. If a fully filled row—that is, one whose every column contains blocks—the function would, then, clear that row by setting its cells to zero, shifting all rows above down to fill in the space that was opened.

Detecting a Full Row: If row 18 contains blocks in all columns:

Listing 12: Code snippet of c code

```
for (int col = 0; col < GRID_COLUMNS; col++) {
    if (grid[18][col] == 0) {
        isFull = 0; // Row is not full
        break;
    }
}
if (isFull) {
    ClearRow(18); // Clear the row
}
```

Clearing a Row and Shifting: When row 18 is full, it is cleared, and rows 0 to 17 are shifted down:

Listing 13: Code snippet of c code

```
for (int row = 18; row > 0; row--) {
    for (int col = 0; col < GRID_COLUMNS; col++) {
        grid[row][col] = grid[row - 1][col]; // Shift down
    }
}
memset(grid[0], 0, sizeof(grid[0])); // Clear the top row
```

Updating the Score: For every cleared row, the score is incremented based on a predefined value:

Listing 14: Code snippet of c code

```
score += 100; // Add points for clearing a row
```

This function is crucial for maintaining the flow of gameplay by ensuring rows are removed and the grid remains organized.

### 7.4.4 Score Box

Score box: It will be a display region that depicts visually the current score of the player. During runtime, it will be changed by a step value of 100 in every refreshing step. `DrawScoreBox()` handles the visual layout of the box while the number inside the box is drawn using the `DrawNumber()` function.

It draws a box-shaped rectangle using the `DrawToTheMonitor()` method, filling up the area with a background color. This is because the score would come in steps of 100- that

is, 100, 200, 300, and so on. Say when a player clears one row, the score comes to be 100 and the DrawNumber() draws the digits in their respective positions inside the box. Say if it is 600 then the function would align properly to center the number in their respective places:.

This makes the score box refreshed within the game loop to be able to keep the current score. This feature gives the players a view that is clear enough and will help to visually represent their progress in this game.

## 7.5 Rendering

The rendering uses a double-buffering scheme to maintain visual smoothness by avoiding screen tearing. It does so by using two frame buffers, one for the current frame that is on-screen and another for preparing the next one. When the next frame is prepared, the buffers are swapped and seamless updates with no disturbance of the active display are made.

pFrames holds an array of pointers to the frame buffers and currentFrame identifies the index of the currently active buffer. So for instance, if currentFrame points to frame 0, then rendering of the next frame will be done in frame 1. When rendering is completed, the buffers are swapped, and the cache is flushed so that the updates become visible on screen.

Assigning the next frame for rendering:

Listing 15: Code snippet of c code

```
u8 *nextFrame = pFrames[(currentFrame + 1) % DISPLAY_NUM_FRAMES];
RenderFrame(nextFrame, blockX, blockY, blockActive, currentShape,
             currentRotation);
```

Swapping buffers and flushing the cache:

Listing 16: Code snippet of c code

```
currentFrame = (currentFrame + 1) % DISPLAY_NUM_FRAMES; // Move
                to the next frame
SwapGridBuffers(); // Swap the display buffers
Xil_DCacheFlushRange((unsigned int)pFrames[currentFrame],
                     dispCtrl.vMode.height * dispCtrl.stride); // Flush cache
```

Adding a delay to control frame timing:

Listing 17: Code snippet of c code

```
usleep(1000); // Introduce a delay for consistent FPS
```

When a new frame is available, RenderFrame builds the visuals in the next buffer. The currentFrame index is incremented circularly to toggle between buffers. Because only the grid area needs updating for performance reasons, the SwapGridBuffers function does an update of that area. Finally, Xil\_DCacheFlushRange flushes the cache of the active buffer, making the updates visible. This delay created by usleep makes the visuals flow smoothly while running the game. This is the continuous repetition of every game loop.

## 7.6 Challenges and Solutions

**1. Implementation of Double Buffering Challenge:** Smooth rendering without flickering required efficient double buffering. The major problem was how to handle the synchronization of the current and the next frame while drawing everything correctly before showing that frame.

The solution idea will be to make use of the `SwapGridBuffers` function that clears off the cache for the area of the grid so that modifications to the next frame show instantaneously; thereby in each cycle of rendering, write to the back buffer, whereas after the end of rendering, flip the buffers to show up: More smoothing of frames come out this way.

### 2. Block-Collision Detection And Moves:

The challenges presented are the accurate collision detection of moving blocks, the falling of blocks within the grid area, consideration of rotation, and detection of whether a block can go further down.

**Solution:** The functions `CanMoveDown`, `CanMoveHorizontal`, and `CanRotate` were implemented to handle collisions and ensure the validation of block movements. These functions check for conflicts in the grid matrix and ensure that no illegal move is made through it. Any collision updates the grid array to mark the position of the block finally.

### 3. Shape Drawing and Rotation:

**Problem:** To be able to represent different Tetris shapes with rotations but still keep it clear. Each shape had to render correctly and adjust to rotations.

**Solution:** Shapes were kept in the array `SHAPE_DATA` as 4x4 matrices; there was one for each different rotation. These were then used by functions like `DrawShapeT_Rotated` and `DrawShapeL_Rotated` to correctly position blocks within a shape and make sure each shape and its rotation looks and fits as it should within the grid.

### 4. Button Mapping and Debouncing:

**Problem:** Getting proper, reliable input from 9 buttons without getting incidental multiple presses as a result of hardware bounce.

**Solution:** Every button had its action through the usage of `XGpio_DiscreteRead`. In building a debounce logic, time checks were performed for every press of the button to ensure intervals; that was pretty helpful in making sure spurious inputs did not occur.

### 5. SD Card Integration for Image Loading:

**Problem Statement:** Reading large-size images from SD card into a frame-buffer required precise management of memory with good file reading mechanisms.

**Solution:** To access the .ppm image files on the SD card, the `LoadAndDisplayImage` function was utilized. By converting the images directly into pixel data mapped onto the frame buffer using DMA and calling `Xil_DCacheFlushRange` to make sure the cache is flushed, images can be efficiently rendered.

### 6. Grid Clearing and Scoring:

**Problem:** Full row detection within the grid-clearing logic that resets rows while the rest of the game remains intact.

**Solution:** The `ClearFullRows` function identifies and clears full rows, shifting all rows above down using `ShiftRowsDown`. In this operation, the score is updated, and the grid

state changes dynamically to keep the game going.

Each problem was solved with a separate function and logical abstractions, keeping the code modular and maintainable. It provided a solid implementation for the complex nature of the Tetris game.

## 8 Tests

Test case	Expected result	Actual result	Remarks
Button Input Responsiveness	All buttons respond correctly to press events (e.g., moving blocks, rotating, dropping).	Matches expected	Button presses reliably control block actions after implementing debounce logic.
Block Rendering	Blocks are drawn in correct positions with accurate colors and borders.	Matches expected	Accurate block placement with visible borders achieved using the 'DrawBlock' function.
Shape Rotations	Shapes rotate correctly, following predefined configurations for each type.	Matches expected	All shapes rotate as intended, even near grid boundaries.
Collision Detection	Blocks stop moving when colliding with other blocks or grid boundaries.	Matches expected	Collision logic prevents blocks from overlapping or leaving the grid.
Row Clearing Mechanism	Full rows are cleared, and rows above shift downward correctly.	Matches expected	Cleared rows and downward shifting are visually accurate.
Score Display	Current score updates correctly and is displayed on-screen.	Matches expected	Scores update dynamically, and the 'DrawScoreBox' function renders them accurately.
Level Display	Current level is displayed and increments after a certain number of cleared rows.	Matches expected	Level increments after clearing the required rows, displayed using the 'DrawDigitLevel' function.
Game Over Condition	The game stops and displays a "Game Over" screen when blocks reach the top.	Matches expected	Game Over screen appears as intended; gameplay halts upon reaching the top row.
Double Buffering Effectiveness	Smooth rendering with no screen flickering during gameplay.	Matches expected	Double buffering ensures seamless frame updates without visible flicker.
SD Card Functionality	Images and assets are loaded correctly from the SD card.	Matches expected	SD card initialization and file loading work reliably for game assets.

Table 2: Test cases and results for the Tetris implementation.

A few things that were not up to standard, were the next block and some rendering. Meaning tha the grid and the scorebox renders when it should not, and they should only render when an action happens, this was a hard thing to implement. Also if the gameover happens and u press btn2 once, a weird screen comes up, however if u press it again, the game restarts correctly.

## 9 Discussion

The development of the Tetris game on the Zybo board brought in opportunities hand in glove with challenges, testing both technical skills and project management capability. This was generally a great success, since it integrated hardware and software components into one. In other words, visually pleasing displays were available through HDMI output, and using GPIO for user inputs ensured responsiveness in gameplay. A few salient features include double buffering that ensures smooth rendering without screen tearing, also guaranteeing a consistent frame rate.

Though, at the same time, there have been a few roadblocks in this project. Collision detection of falling blocks initially was pretty not-straightforward because a lot of combinations of grid interactions along with boundary checks were there. Other minor issues were unnecessary redrawing of the static elements and inconsistent behavior on a game-over state. For example, updating of the grid and scoreboard when they have no reason to update is inefficient; it will be much better not to. Other than this, it could also include a Next Block Box that displays the next block to be played in order for the player to strategize and provide better planning in their moves and give more likeness to the classic game of Tetris.

Considering the resource constraints, a few design decisions had to be made-for example, both the limited time and hardware. In streamlining the project, implementation of audio and Ethernet connectivity was omitted, adding SD card functionality and button mapping really fleshed it out. This was an evident prioritization of core gameplay over secondary features very much about keeping the project focused and manageable.

In the end, the iterative approach to problem-solving allowed us to overcome most of the issues, showed adaptability, and teamwork. Refinements from the initial objectives, such as the rendering logic and more UI elements, actually fleshed out the completed product. It is very easy to iterate upon this with greater optimization of performance, among other features that enhance gameplay.

## 10 Conclusion

In the Tetris project, it was able to reach its major objectives in creating a working and entertaining game with the implementation on the Zybo board. It was also in line with the integration of HDMI output, GPIO for user input, and the game mechanics involved. The features include line clearing, block rotation, and scoring, which were implemented successfully in creating a smooth and quick responsive game.

The ability to add enhancements for SD card functionality and button mapping to the reasons that contributed to overall success, showing the our adaptation ability. Audio functionality and advanced network features have been excluded from this version due to the limitation of time and resources, but the core mechanics we focused on helped them deliver a polished and playable game. Other enhancements could be the rendering logic and inclusion of a Next Block Box, which would enhance the overall usability and performance of the game.

This project has been a great learning experience, reinforcing skills in hardware-software integration, debugging, and modular design. It provided insight into the challenges and opportunities of embedded systems development. We were able to walk a good line between complexity and practicality; the product can definitely show the potential of the Zybo board in being used with interactive applications. Future versions can build on this groundwork and give a more polished and feature-filled game experience.

## 11 User manual

There is 2 ways the user can start the game

## 11.1 Booting from SD

The first way is to boot from sd what is required for this to function is to add the BOOT.BIN file on the sd and insert in zybo board. After this the user turns on the zybo and the game starts on the monitor.

## 11.2 Booting from SDK

The second way to start the game is to boot from the SDK. This is done by installing the .zip file in and programming the FPGA with the hdmi\_out\_wrapper\_hw\_platform\_1. Then the user has to run as GDB.

Additionally the controller has to be connected to the zybo in the following order:

- btn4 to V12
- btn5 to W16
- btn6 to J15
- btn7 to H15
- btn8 to V13

If the buttons are not configured correctly, the game will assume that its being pressed constantly and the game will not work as intended.

The game functions are previously explained in the flowcharts above.