ECE 385

Fall 2024

Final Project

Space Invaders using Microblaze (SoC)

Sara Sehgal (sehgal5)
Matthew Curran (mc159)

I. Introduction

For the final project of this course, I chose to implement the arcade game **Space Invaders** on the **Spartan-7 FPGA** with a **MicroBlaze System-on-Chip (SoC)**. Space Invaders is a 2D shooting game that features a player-controlled ship at the bottom of the screen tasked with shooting down rows of moving invaders while avoiding their projectiles.

Introducing the basic idea of the game, the player controls the ship using a keyboard interface for left and right movement and fires projectiles to eliminate the invaders. The invaders move as a group, bouncing horizontally across the screen, and gradually descend over time. The game gets more challenging as invaders move faster when their numbers decrease. The player's objective is to destroy all the invaders.

To achieve this, I utilized the Spartan-7 FPGA and integrated a MicroBlaze soft-core processor to manage input, game state, and timing. Real-time graphics for the player, invaders, and bullets are generated using a VGA controller, while input controls are handled through a USB keyboard interface.

II. Design Goals

The goal of this project is to combine concepts of digital logic design, hardware-software co-design, and real-time rendering to implement an interactive game. Key objectives include:

1. **Player Control:** Smooth movement and firing capabilities based on keyboard input.
2. **Invader Behavior:** Group movement, bouncing logic, and descent as the game progresses.
3. **Collision Detection:** Detecting interactions between player projectiles and invaders.
4. **VGA Display:** Rendering the player, invaders, bullets, and background in real-time.
5. **System Integration:** Efficient integration of all game components using SystemVerilog modules.

The Spartan-7 FPGA provides sufficient resources for implementing these functionalities in hardware, ensuring smooth and responsive gameplay. The MicroBlaze SoC is used for higher-level game state control and coordination of events.

III. Game Overview

1. Player Ship:
   - Moves horizontally across the bottom of the screen using keyboard inputs.
   - Fires projectiles upward to destroy invaders.
2. Invaders:
   - Move horizontally as a group and reverse direction upon hitting the screen edges.
   - Gradually descend toward the bottom of the screen.
3. Player Projectiles:
   - Fired vertically upward to interact with invaders.
4. Game Progression:
   - Invaders increase their speed as the game progresses, creating additional challenge.

IV. Microblaze Description

The game is implemented with a **MicroBlaze SoC**, which is used to handle communication between a USB peripheral (keyboard) and the FPGA. This setup is implemented in a similar manner to Lab 6 and enables smooth interaction between the player and the game hardware. The keyboard input serves as the control interface for the player ship, allowing it to move left, right, and shoot projectiles.

To enable USB keyboard communication, I used the SPI protocol to connect the master device (FPGA) and the slave device (MAX3421E chip), which acts as the USB host interface. The SPI protocol is a full-duplex communication standard, and its bus consists of four channels. The **SS** (Slave Select)is an active-low signal that selects the MAX3421E chip for communication. When SS is low, the device is enabled, and the master (FPGA) can initiate data transfer. The **MISO** (Master In Slave Out) transfers data from the slave (MAX3421E) to the master (FPGA). **MOSI** (Master Out Slave In) transfers data from the master (FPGA) to the slave (MAX3421E). Lastly, the **CLK** (Clock) synchronizes communication between the master and slave. Since SPI is full duplex, when one channel is transferring, the other channel must be filled with dummy byte.

Other than that, there are still several blocks/modules that are used with the processor, which will be discussed in the next section.

V. Module Description

Ball.sv (player module)

```
input  logic        Reset,
   input  logic        frame_clk,
   input  logic [7:0]  keycode0,
   input  logic [7:0]  keycode1,

   output logic [9:0]  PlayerX,
   output logic [9:0]  PlayerY,
   output logic [9:0]  PlayerS
```

The player module is responsible for controlling the player ship in the Space Invaders game. It manages the player's horizontal movement based on keyboard inputs (left and right) and ensures the ship remains within defined screen boundaries. The vertical position and size of the player remain fixed, simplifying the control logic.

Color_Mapper.sv

```
input  logic [9:0] PlayerX, PlayerY, DrawX, DrawY, Player_size,
input  logic [9:0] InvaderX[50], InvaderY[50],
input  logic [9:0] ShotX, ShotY, ShotW, ShotH,
input  logic       ShotOn,
input  logic [49:0] InvaderOn,
input  logic [6:0]  offset,
output logic [3:0] Red, Green, Blue
```

This module is used to map color (RGB value) to every pixel on the VGA canvas. It takes in the coordinates of the player ship, invaders, and the player's shot to render their respective sprites on the screen based on their positions. The module dynamically checks whether the current pixel being drawn matches the position of the player, active invaders, or the projectile, and assigns the appropriate color. It also integrates a scrolling background effect using an offset signal to create visual depth. The priority of rendering ensures that projectiles appear on top of all other elements, followed by the player ship, invaders, and the background. By taking DrawX and DrawY as inputs, the module maps each pixel coordinate to its respective color, ensuring real-time visual updates of the game components. The output consists of the RGB color value for the current pixel.

Modules Instantiated

- DrawInvader: Handles the rendering of each individual invader sprite based on its position and size.
- DrawBackground: Generates the scrolling background effect using DrawX, DrawY, and offset.
- Pallette: Maps indices to specific RGB values for rendering sprites and the background.
- DrawPlayer: Renders the player ship sprite based on its coordinates and size.

DrawBackground.sv

```
input  logic [9:0] DrawX,
input  logic [6:0] DrawY,
input  logic [6:0] offset,
output logic [2:0] PalInd
```

The DrawBackground module is responsible for generating a scrolling background effect on the VGA display. It calculates the color palette index for each pixel based on its coordinates (DrawX and DrawY) and a vertical offset. By dynamically shifting the row value with the offset, the module creates the illusion of a moving background, adding depth to the game.

DrawInvader.sv

```
input  logic [9:0] DrawX,
input  logic [9:0] DrawY,
input  logic [9:0] InvaderX,
input  logic [9:0] InvaderY,
input  logic [4:0] Size,
output logic       invader_on
```

The DrawInvader module is responsible for rendering individual invader sprites on the VGA display based on their coordinates and sizes. It takes as input the current pixel coordinates (DrawX, DrawY), the position of the invader (InvaderX, InvaderY), and the size of the invader (Size). Using combinational logic, the module determines whether the current pixel lies within the boundaries of the invader sprite. The invader is rendered by calculating the column (col) and row (row) of the pixel relative to the invader's position and checking these values against predefined patterns for different sprite sizes (Size values of 8, 9, and 10). These patterns

determine which pixels form the visible shape of the invader. If the current pixel matches part of the invader's pattern, the invader_on signal is set to 1, indicating that the pixel belongs to the invader sprite; otherwise, it is set to 0. This modular approach enables efficient and customizable rendering of invaders of varying sizes and positions, ensuring visual clarity and dynamic sprite behavior during gameplay.

DrawPlayer.sv

```
input  logic [9:0] DrawX,
input  logic [9:0] DrawY,
input  logic [9:0] PlayerX,
input  logic [9:0] PlayerY,
input  logic [4:0] Size,
output logic       player_on
```

The DrawPlayer module is responsible for rendering the player ship sprite on the VGA display based on its coordinates and size. It takes as inputs the current pixel being drawn (DrawX, DrawY), the player's position (PlayerX, PlayerY), and the player's size (Size). Using combinational logic, the module calculates the relative column and row of the pixel with respect to the player's position. It then checks these values against predefined conditions that represent the shape of the player sprite. If the current pixel lies within the boundaries of the player sprite, the output signal player_on is set to 1, indicating that the pixel belongs to the player ship; otherwise, it is set to 0.

Hex_driver.sv

```
input   logic         clk,
input   logic         reset,
input   logic  [3:0]  in[4],
output  logic  [7:0]  hex_seg,
output  logic  [3:0]  hex_grid
```

The hex_driver module displays 4-digit hexadecimal values on a seven-segment display using multiplexing. It takes as inputs a 4-bit nibble array (in[4]), a clock signal (clk), and a reset signal (reset). The outputs are segment control signals (hex_seg) to illuminate segments and digit enable signals (hex_grid) to select which digit is active.
The module uses an internal submodule, nibble_to_hex, to convert each 4-bit nibble into its corresponding seven-segment encoding (0–F). A 17-bit counter cycles through the four digits by enabling one grid at a time while sending the appropriate segment data.
In the reset state, all outputs are turned off. During operation, the display multiplexes through all digits so quickly that all four appear active simultaneously.

<u>Modules Instantiated:</u>

- nibble_to_hex: Converts a 4-bit binary nibble into a seven-segment display encoding.

InstantiateInvaders.sv

```
input  logic       reset_ah,
```

```
input  logic       vsync,
input  logic [7:0]  keycode,
input  logic [9:0]  ShotX,
input  logic [9:0]  ShotY,
input  logic [9:0]  ShotW,
input  logic [9:0]  ShotH,
input  logic [49:0] Collision,

output logic [9:0]  InvaderX[50],
output logic [9:0]  InvaderY[50],
output logic [49:0] InvaderOn
```

The InstantiateInvaders module is responsible for instantiating and managing a grid of 50 invaders for the Space Invaders game. It handles invader positioning, activation status, and interaction with the player's shot. The inputs include the reset signal, vertical sync clock (vsync), keyboard input for movement, coordinates and dimensions of the player's shot (ShotX, ShotY, ShotW, ShotH), and an array indicating collision statuses for each invader (Collision). The outputs include arrays for invader positions (InvaderX and InvaderY) and their activation states (InvaderOn).
Internally, it calculates the first and last active columns of invaders using combinational logic for efficient boundary detection, which helps optimize motion and updates. The invaders are grouped into rows, each with different starting positions and sizes, and instantiated through multiple calls to the invader module. The InvaderSpeed module determines the delay between invader movements to regulate gameplay speed dynamically.
This module plays a central role in creating and controlling the behavior of the invader swarm, including managing their movement, collision detection, and activation status.

<u>Modules Instantiated:</u>

- **Invader:** Controls individual invader movement, collision status, and position updates.
- **InvaderSpeed:** Dynamically adjusts the delay for invader motion based on game state.

Invaders.sv

```
input  logic       Reset,
input  logic       frame_clk,
input  logic [7:0]  keycode,
input  logic [9:0]  ShotX,
input  logic [9:0]  ShotY,
input  logic [9:0]  ShotW,
input  logic [9:0]  ShotH,
input  logic [9:0]  Invader_X_Start,
input  logic [9:0]  Invader_Y_Start,
input  logic [9:0]  InvaderS,
input  logic [2:0]  index,
input  logic [5:0]  Last_Active_Col,
input  logic [5:0]  First_Active_Col,
input  logic       Collision,
```

```
input  logic [3:0]  Delay,

output logic [9:0]  InvaderX,
output logic [9:0]  InvaderY,
output logic        InvaderOn
```

The invader module is responsible for managing the behavior and movement of a single invader in the Space Invaders game. It takes inputs such as the reset signal, frame clock, keycode, and the player's shot coordinates (ShotX, ShotY, ShotW, ShotH). Additionally, it uses parameters like the invader's starting position (Invader_X_Start, Invader_Y_Start), size (InvaderS), and indices for active columns (First_Active_Col, Last_Active_Col). The output includes the current position of the invader (InvaderX, InvaderY) and its activation status (InvaderOn).
The module calculates horizontal motion for the invader, ensuring it bounces at predefined boundaries (Invader_X_Min, Invader_X_Max). The motion direction is updated dynamically based on its position. A delay counter regulates the movement speed for smoother animation. If a collision with the player's shot occurs, the invader is marked inactive by resetting InvaderOn. This module ensures that individual invaders move, bounce within boundaries, and deactivate upon collisions, contributing to the coordinated behavior of the invader grid.

InvaderSpeed.sv

```
input  logic [49:0] InvaderOn,
output logic [3:0]  Delay
```

The InvaderSpeed module dynamically adjusts the movement delay for the invaders based on the number of active invaders remaining in the game. It takes as input a 50-bit vector (InvaderOn), where each bit represents whether an invader is active. The output is a 4-bit delay signal (Delay) that determines the movement speed of the invaders.
Internally, the module counts the number of active invaders by iterating through the InvaderOn vector. As the number of invaders decreases, the delay reduces, thereby increasing the speed of the invaders. The delay value is capped at a maximum of 15 to ensure a controlled speed progression.
This module provides an adaptive mechanism to increase game difficulty over time, mimicking the classic Space Invaders gameplay where invaders move faster as their numbers dwindle.

Mb_usb_hdmi_top.sv

```
    input logic Clk,
    input logic reset_rtl_0,

    //USB signals
    input logic [0:0] gpio_usb_int_tri_i,
    output logic gpio_usb_rst_tri_o,
    input logic usb_spi_miso,
    output logic usb_spi_mosi,
    output logic usb_spi_sclk,
    output logic usb_spi_ss,
```

```
//UART
input logic uart_rtl_0_rxd,
output logic uart_rtl_0_txd,

//HDMI
output logic hdmi_tmds_clk_n,
output logic hdmi_tmds_clk_p,
output logic [2:0]hdmi_tmds_data_n,
output logic [2:0]hdmi_tmds_data_p,

//HEX displays
output logic [7:0] hex_segA,
output logic [3:0] hex_gridA,
output logic [7:0] hex_segB,
output logic [3:0] hex_gridB
```

The mb_usb_hdmi_top module serves as the top-level design for the Space Invaders project, integrating multiple key components to facilitate input/output operations, display rendering, and overall game management. It includes USB communication, VGA-to-HDMI conversion, key processing, and instantiation of game logic modules.
The module takes in several signals, including a clock (Clk), reset signal, USB inputs, and UART connections. It outputs VGA/HDMI video signals, HEX display data, and controls the RGB color mapping for the game display. The MicroBlaze SoC handles USB keyboard inputs via the SPI protocol to process keycodes, while clock generation is managed by a clock wizard module producing multiple frequencies required for video rendering.

Modules Instantiated

- **hex_driver:** Displays keycodes on HEX displays for debugging.
- **mb_block:** Manages USB communication using the MicroBlaze SoC.
- **clk_wiz_0:** Generates required clock frequencies.
- **vga_controller:** Produces synchronization signals and pixel coordinates.
- **hdmi_tx_0:** Converts VGA signals to HDMI-compatible outputs.
- **player:** Handles player movement based on key inputs.
- **InstantiateInvaders:** Generates invader instances and updates their positions.
- **shot:** Manages player shots, collision detection, and projectile motion.
- **color_mapper:** Maps pixel coordinates to RGB values for display rendering.

Pallette.sv

```
input  logic [2:0] index,

output logic [3:0] red,
output logic [3:0] green,
output logic [3:0] blue
```

The Pallette module maps a 3-bit index input to corresponding RGB color values. It outputs 4-bit red, green, and blue signals to represent specific colors used in the game display. The unique

color mapping allows differentiation of game elements like the background, player, invaders, and projectiles.

Shot.sv

```
input  logic        Reset,
   input  logic        frame_clk,
   input  logic [7:0]  keycode0,
   input  logic [7:0]  keycode1,
   input  logic [9:0]  PlayerX,
   input  logic [9:0]  PlayerY,
   input  logic [9:0]  PlayerS,
   input  logic        Player, // Indicates whether this is a player shot
   input  logic [9:0]  InvaderX[50],
   input  logic [9:0]  InvaderY[50],
   input  logic [49:0] InvaderOn,

   output logic [9:0]  ShotX,
   output logic [9:0]  ShotY,
   output logic [9:0]  ShotW,
   output logic [9:0]  ShotH,
   output logic        ShotOn,
   output logic [49:0] Collision
```

The shot module handles the functionality of projectiles in the game, including firing shots, updating their position, and detecting collisions. It accepts inputs such as the player's coordinates, invader positions, shot dimensions, keypresses, and the Player signal to distinguish between player and invader shots. The module outputs the shot's coordinates (ShotX, ShotY), its dimensions (ShotW, ShotH), its active state (ShotOn), and a Collision signal array indicating which invader has been hit.

When the fire key (e.g., keycode 44) is pressed, a shot is launched from the player's position. The shot moves upward if fired by the player and downward if fired by an invader. The module detects collisions by comparing the shot's position with the positions and sizes of invaders, deactivating the shot upon impact and setting the corresponding Collisionsignal. The projectile also deactivates when it reaches the screen boundaries. This design ensures efficient and accurate shot movement and collision detection.

VGA_controller.sv

Input: pixel_clk,
reset,
Output: hs,
vs,
active_nblank,sync,
[9:0] drawX,
[9:0] drawY

This module is used to generate various VGA sync functionalities, which includes horizontal sync (hsync) and vertical sync (vsync) which is useful in this lab.

IP Blocks:

mb_block

This is the IP block that wraps my integrated Microblaze processor, which includes several components, which I will list below.

MicroBlaze: This is the MicroBlaze processor itself; just like the slc-3 module in lab5, this include the ISA of the processor.

MicroBlaze Debug Module (MDM): this is the debugging module of the processor, which can enable us to run debugging routine on MicroBlaze.

AXI Interrupt Controller: This module receives interrupts from all peripherals and merge them into a single interrupt signal and output it to MicroBlaze.

Concat: This module concatenates several 1-bit inputs to a multiple-bit output.
Processor System Reset: This module provides customizable reset to all systems, including the processor, interconnect, and peripherals.

Clocking wizard: This module can customize input clock by configuring its buffering, feedback, and timing parameters.AXI GPIO: As its name suggests, the General Purpose IO provides an Input/Output interface
for the peripherals to communicate with the processor. It can support input/output from 1 to 32 bits.

AXI Timer: This module serves as a timer/counter for the processor. It can be used to provide reliable timing.

AXI Uartlite: The module provides the controller with unsynchronous serial data transfer abilities.

AXI Quad SPI: The module serves as an interface for the processor to connect to the peripherals; its protocol enables data transfer between master/slave devices.

AXI Interconnect: This module connects one or more AXI memory mapped master devices to one or more AXI memory mapped slave devices.

clk_wiz_0

This module is used to provide a 25MHz clock and a 125MHz clock for the VGA to HDMI converter.

hdmi_tx_0

This module converts VGA signal to HDMI signal, so that we can plug the FPGA via an HDMI cable.

VI. Top Level Block Diagram

Because our logic relied on the information regarding each invader being accessible in each module simultaneously, there are a lot of wires between the modules, making the block diagram hard to understand.

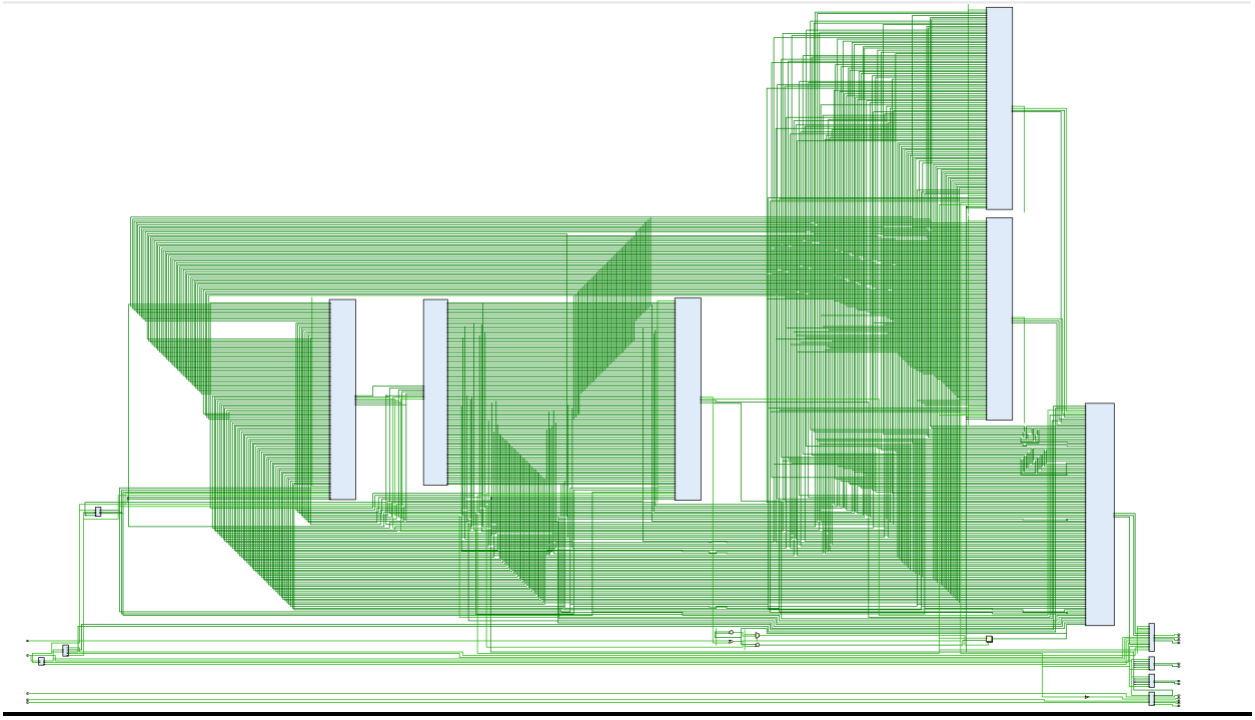

Figure 1, Real Block diagram

For the purpose of simplicity, I rewrote the code to only connect the location of the first invader between the modules. Note that this diagram will not work, with the only difference being that the InvaderX and InvaderY signals will be duplicated an additional 49 times.

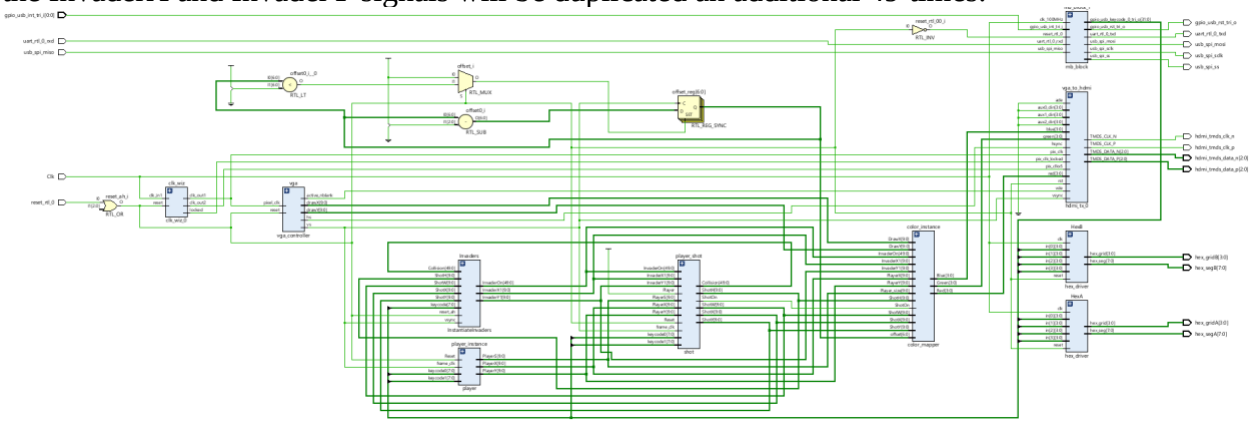

Figure 2, Simplified Block diagram

VII. Simulation

Since this design is a game that is highly based on graphics, the simulation cannot show much other than the working logic of some counters/ROMs. As a result, the entirety of the testing process took place on the HDMI output, by exporting the project hardware to Vitis.

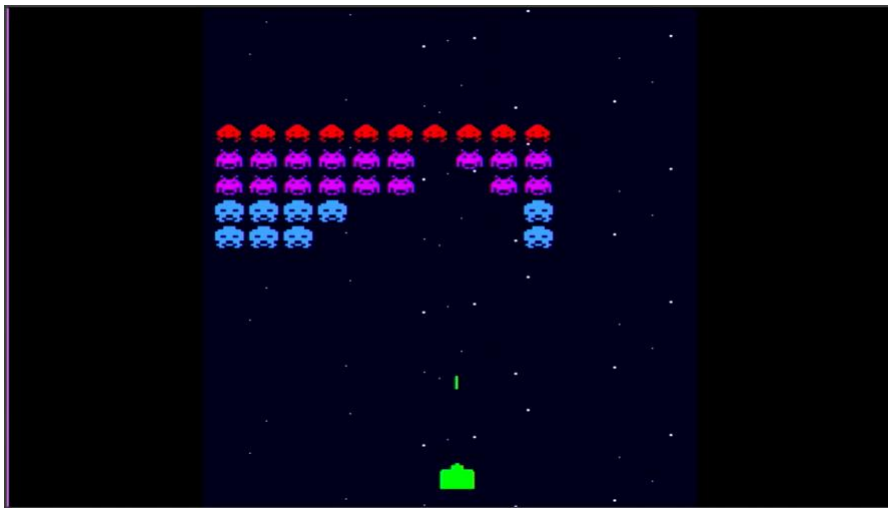

Figure 3, HDMI output

Key features to note here are the different sprites and colors for the different rows of invaders, the player character sprite and shot, which are the same color, and the starry background, which scrolls as the game is played.

VIII. Design Statistics

LUT	13841
DSP	3
BRAM	32
FLIP-FLOP	4326
LATCHES	0
FREQUENCY	135.7 MHz
STATIC POWER	0.593 W
DYNAMIC POWER	0.516 W
TOTAL POWER	0.077 W

Table 1, Design Statistics

IX. Conclusion & Potential Improvements

The implementation of Space Invaders on the Spartan-7 FPGA with a MicroBlaze SoC successfully demonstrates the integration of hardware modules to create a functional video game system. The project uses VGA signaling to display game graphics, a USB interface for keyboard inputs, and modular Verilog designs to control gameplay elements such as the player, invaders, and projectiles. The key features include smooth player movement, collision detection between the player's shots and invaders, dynamic invader behavior, and real-time updates of the display. By carefully orchestrating synchronization signals, clock generation, and game logic, the system delivers responsive and visually accurate gameplay.

The modular approach to designing components such as the player, invaders, shot handling, and color mapping ensures scalability and maintainability. The VGA controller effectively drives the

display, while the color_mapper and associated sprite modules handle the visual representation of game elements. The use of a scrolling background and multiple invaders adds to the visual complexity and enhances the gaming experience.

Overall, the game design has been a success, and I was able to implement all the key elements that make the original Space Invaders game unique and enjoyable. This final project serves as a fitting conclusion to the semester, providing an excellent opportunity to apply and test the hardware programming skills acquired throughout the course.

Some features that could be added to make this game more interactive and challenging are:

I. **Enhanced Invader Behavior:** The invader movement is currently linear with predefined boundaries. Introducing more complex patterns, such as zigzag or randomized motions, would make the game more challenging and engaging.
II. **Multiple Player Shots:** Currently, the system supports a single active player shot. Adding the ability to fire multiple shots simultaneously would provide a more dynamic gameplay experience.
III. **Player Lives and Score System:** Implementing a scoring mechanism and a player lives counter would allow for a complete game experience, enabling players to track their progress and strive for higher scores.
IV. **Audio Feedback:** Integrating sound effects for events like shooting, invader elimination, and game-over conditions using an additional audio module would greatly enhance user engagement.
V. **Game Difficulty Progression:** Gradually increasing the speed of the invaders or introducing new types of enemies as the game progresses would improve replayability and challenge.
VI. **Texture-Based Sprites:** Replacing the pixel-based shapes with texture-based sprites for the player, invaders, and projectiles would provide a more polished and visually appealing game display.
VII. **Improved USB Input Handling:** Optimizing the keyboard input interface to handle multiple simultaneous key presses could allow for advanced player controls, such as diagonal movements.