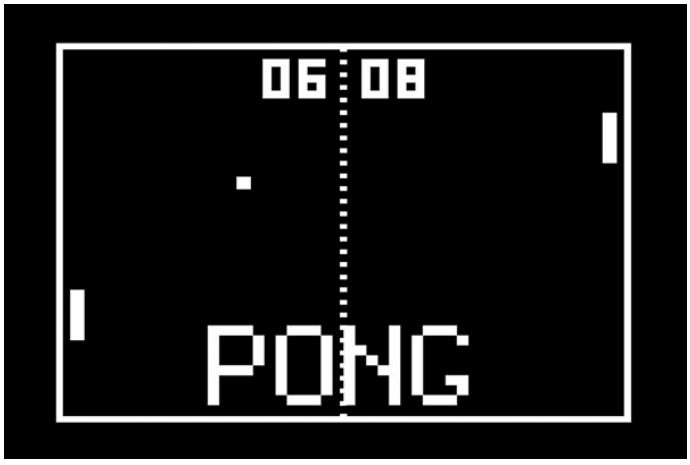
University of Toronto
Faculty of Applied Science & Engineering
ECE532

Group Report

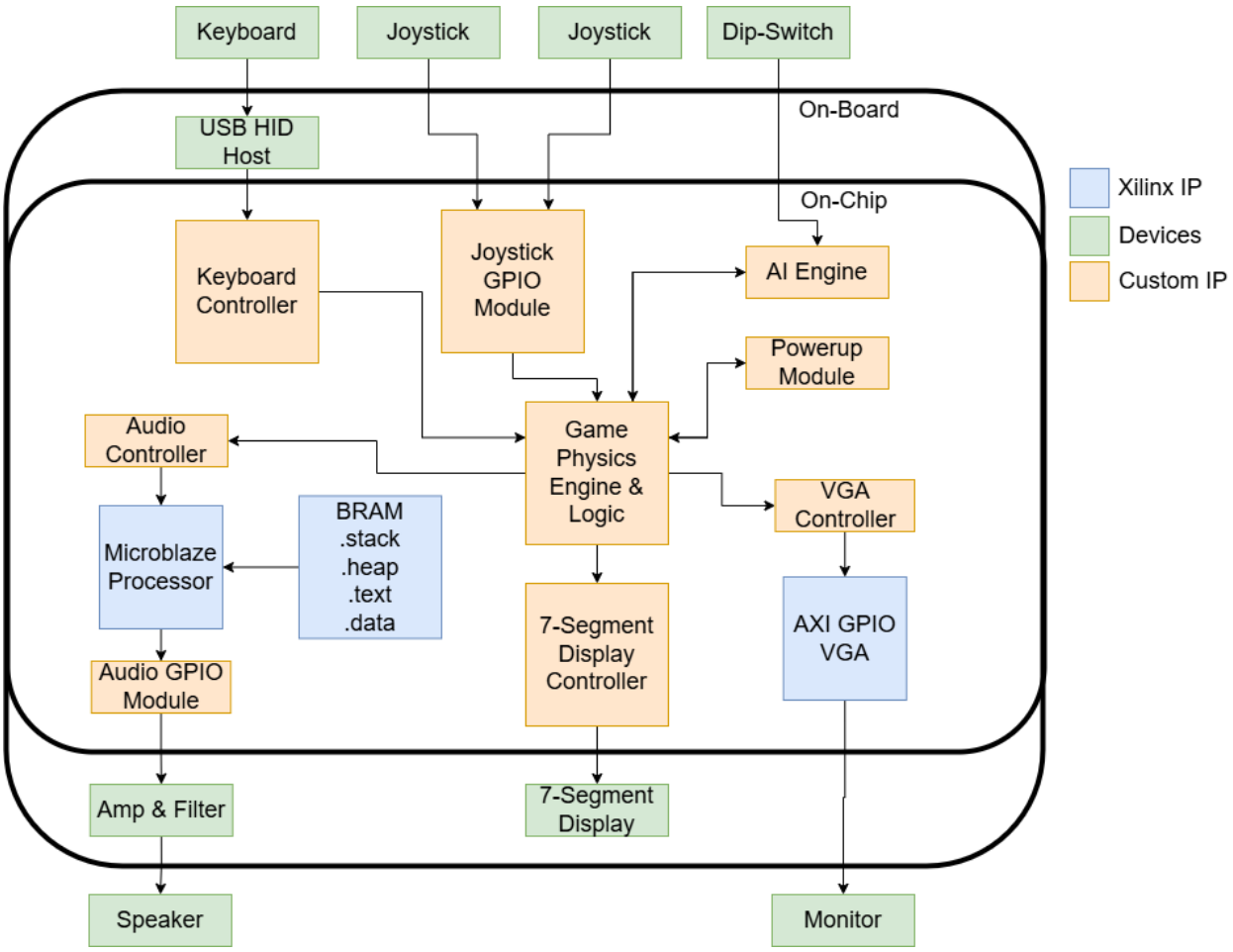
Project:	Pong	Date	April 7th, 2025

Group	27
Student(s)	Michael Acquaviva Andrew Moser Antoine Vilain Alan Cao



Overview

Our team wanted to create an interactive and engaging project. Everyone on the team enjoys video games so we explored different game ideas such as Pacman, Tetris, chess etc. The team decided on the classic video game Pong. This project allowed us to combine our interest in gaming with FPGA development while learning how to integrate key hardware components like VGA display, audio processing and input controllers. The motivation behind this project was to explore FPGA-based game development while integrating various hardware peripherals. Implementing Pong on the Nexys4 DDR board gave us experience with VGA output, audio processing and real-time system design. The team also worked with various AXI-based IP blocks, memory management, and MicroBlaze soft-core processing. We decided to make the game into a 2-player version with dual joysticks to incorporate more players. The team decided to set the goal to be the normal base game, and if time permits, to then add additional features to the game. After the team achieved the baseline goal, the added additional features include a powerup to change paddle sizes, a title and game over screens, and an AI mode with different difficulty levels.



The system is divided into three layers:

- **Devices (green)**: These include external components like the keyboard, joysticks, monitor, speaker, and 7-segment display.
- **Custom IP (orange)**: These are Verilog-based modules created for core game functionality, such as the game physics engine, input controllers, powerup module, audio controller, and 7-segment display logic.
- **Xilinx IP (blue)**: Prebuilt IP blocks like MicroBlaze (a soft-core processor), BRAM for memory, and AXI GPIO for VGA are used for control and interfacing.

Key Data Flow:

- **Input Devices**: The keyboard and joysticks feed into respective controllers via USB HID and SPI. These modules forward player input to the Game Physics Engine.
- **Game Logic**: The Game Physics Engine handles paddle movement, ball physics, collisions, powerups, and scoring and game over. It communicates with the 7-Segment Display Controller to update scores.
- **Video Output**: Game state is rendered via the VGA Controller and sent through an AXI GPIO block to a monitor.
- **Audio Output**: The Audio Controller receives game events from the Game Engine and passes sound data to the MicroBlaze and Audio GPIO, which drive the speaker through an amplifier and filter.
- **MicroBlaze Processor**: It manages system tasks, runs software logic, and interacts with BRAM memory where code and data are stored. Specifically, it is in charge of driving sound output in our implementation.

This modular design ensures a separation between game control, processing, and I/O, while using both hardware and software components.

Outcome

Overall, the project exceeded our expectations. The team created a smooth improved game compared to the original classic game. The main objectives of the project were to:

1. Develop a real-time Pong game with two-player functionality.
2. Use an FPGA to handle game logic, rendering, and input processing.
3. Create a graphical display using VGA output.

4. Design a simple and user-friendly interface for players.

The team was able to achieve all of these goals therefore we moved onto additional features. The team incorporated audio, powerups, a title and game over screen and single player AI mode. The additional features were all achieved which really made the game unique, especially the power-up feature. Moreover, many of the important features of the game are implemented completely in hardware! This made the game very different from many of the other games any of us had ever tried to design before and added an extra level of accomplishment.

Beginning with the audio output we implemented various sounds for paddle contact, intro music and powerup sounds. This was definitely the most difficult part of the project. The first idea was to send the data to the PMOD DA2 using I2C. Some audio was able to be played, however, it was too slow for real time audio. It was hard to achieve continuous playback without buffering. This method was first used because it has better audio quality than PWM. Afterwards, the team decided to try the PWM method because we prioritized efficiency, and it did not require any additional hardware. This method used an AXI timer to pass a 44.1 kHz signal followed by a low pass filter and amplifier. The low-quality noise turned out great for this project, however given more time an I2S DAC method can be used to produce higher quality output. In the future if we were to restart this project, we would first try to achieve the lowest possible audio quality (PWM) then proceed to methods that produce higher quality output.

Next, the power-ups were implemented to enhance gameplay and add variety to the classic Pong experience. We introduced a powerup that could dynamically increase or decrease paddle size when collected. This impacts a player's defensive or offensive advantage. The powerups are triggered when a ball collides with randomly spawning powerup blocks on the screen. The powerup system works well and is fully functional in real time with smooth visual feedback via the VGA output. However, we could improve the system by adding more variety in powerup types, such as speed modifiers, multi-ball features, inverted controls, or more. Another idea would be to create better visual indicators to make it clearer when a power-up is active. More strategy could also be introduced with players being able to collect powerups and activate them with a separate input, as opposed to immediately triggering after collection. This allows the player to strategically hold onto and use powerups at opportunistic times. If we could start over, we would begin developing the power-ups earlier in the project to allow more time for testing.

Finally, the AI component was a final late addition to our project. It was difficult to test the AI based on debugging its natural movement. The team implemented a switch that allows instantly switching from single player AI mode to 2 player mode. This AI had both an easy

and hard mode allowing for players to play against different bots. The AI can be improved by being incorporated into a training mode that allows users to practice hitting various difficult shots. This gives players the opportunity to improve their skill, for example in controlling the ball spin to more accurately collect powerups. This would be extremely unique and expand on our primary goal of developing a user-friendly fun game. Starting from scratch, the team would practice implementing multiple algorithms to see which works best based on a specific game mode.

If somebody were to take over the project, the next steps could be to implement a better AI or a greater variety of different powerups. Alternatively, since the game uses 2-axis joysticks it could be very interesting if the paddles could move horizontally as well as vertically, although this presents some additional challenges.

Project Schedule

Based on the team's original project proposal, we outlined the goal of implementing a real-time, two-player Pong game on the Nexys 4 DDR FPGA board. The baseline requirements set were features such as paddle control, ball physics, collision detection, scoring, and VGA output. After exceeding these expectations, the team moved onto additional features such as powerups, audio output and an AI component.

Our weekly milestones followed a structured progression, focusing first focusing on the core components.

High Level General Overview

Week #	**Area of Focus**	**Completion**	**Original Plan**	**Reasoning**
Week 1	VGA Setup	Completed	Completed as planned	Team did thorough research before starting the project.
Week 2	Input Control	Completed	Completed as planned	Team worked through week and made progress on core function
Week 3	Game Logic	Completed	Completed as planned	Everyone completed their task
Week 4	AI Opponent	Completed by week 8	Slightly delayed to week 5	Focus shifted towards audio
Week 5	Audio Output	Completed by week 7	Slightly delayed to week 7	Very difficult task had to change method several times

Week 6	Powerups and Title Screen	Completed	Completed as planned	The powerup adds a unique fun feature to the game
Week 7	Debugging and Integration	Completed	Completed as planned	Good time management
Week 8	Final Demo	Completed	Completed as planned	The project is fully functioning

The team initially began conducting research on the project so coming into week one we already had our goals established. This research included brainstorming which peripherals we should sign out, how the project should be divided between hardware and software, what existing Vivado IPs we could use and what we needed to custom write. Week 1 focused on VGA output, which was achieved successfully. The team established the use of double buffering, which was implemented for smoother rendering. Week 2 the team established the controls used for the game. The keyboard is used to operate the reset and start of game. The joysticks control the paddle movement, and a 7-segment display was used for the score. All of this was done during the second week, so we were able to fast-track other tasks. Moving into Week 3 the game logic was completed perfectly as well. The team configured the game so that certain shots impact ball speed and ball physics was also completed. Week 4 the team faced a minor setback due to fully integrating the AI opponent however this issue was resolved and looked at into later weeks. Week 5 the team decided to incorporate the MicroBlaze component with audio. This was a really difficult task. The initial plan was to use I2C protocols, but the bus ended up being too slow for real time audio to be played. This part was stretched over multiple weeks due to trying various methods and debugging. Week 6 the team decided to add a title screen and powerups. This was difficult but due to good time management this task was finished accordingly. Lastly, in Week 7, all modules were fully integrated and tested for the demo.

Overall, the team followed the original plan set for the demo. Following closely the original plan ensured we were able to complete the task both efficiently and manage our time wisely. Some tasks were finished early like the input control and powerups allowing for more time to be allocated to additional features. The only minor delay was shifting AI development by one week, which didn't affect overall progress. Strong initial planning, task ownership and early testing helped us stay ahead and reduce last-minute issues. The team was very flexible whenever we needed to allocate more time for difficult components such as audio and testing.

System Design Overview

Input Peripherals

The two input peripherals used were the PMOD JSTK2 and an USB keyboard. Both these modules were implemented in custom IP. The joystick module communicates with the PMOD using the SPI protocol and assembles the standard 5-byte data packet. One FSM uses SCLK to read the MOSI and MISO data lines. Another FSM compiles that data into the standard 5-byte packet. The 10-bit X and Y positions are masked and output to the game logic engine. The USB keyboard connects to the HID Controller on the Nexys4 DDR, which emulates a PS/2 style bus. The module uses the pdata and pclk to read and assemble the 1-byte scancode, which is then stored onto a shift register. When the appropriate scancode is released, the module raises the corresponding output flag to the game logic engine.

Game Logic

The Pong module is a custom IP that tracks the game status, including but not limited to paddle positions, ball positions and speed, and player scores. It computes the game state at 30 FPS, and implements game-over logic, ball and paddle/wall collisions, scoring, ball spin, and ball and paddle movements. The powerup module dynamically changes the size of the paddles, and pseudo-randomly generates the powerup's spawn time and spawn position. The powerup is given depending on whether the ball is moving left and right, thus whoever hit the ball last. The player scores are output to the seven-segment display, and almost all signals are output to the VGA module to render the game. The scoring module does use a binary-to-decimal converter which was initially written to debug the joystick module, so if desired, game score can be 10 or higher.

To not have an obscenely high frame rate, the game logic module does not slow down the clock. Instead it uses a game_update register which becomes high once every few million cycles and is gating every update loop within the game logic. This is not a power efficient way to do things but saved us from any potential clock synchronization issues, which for our application was a worthwhile tradeoff.

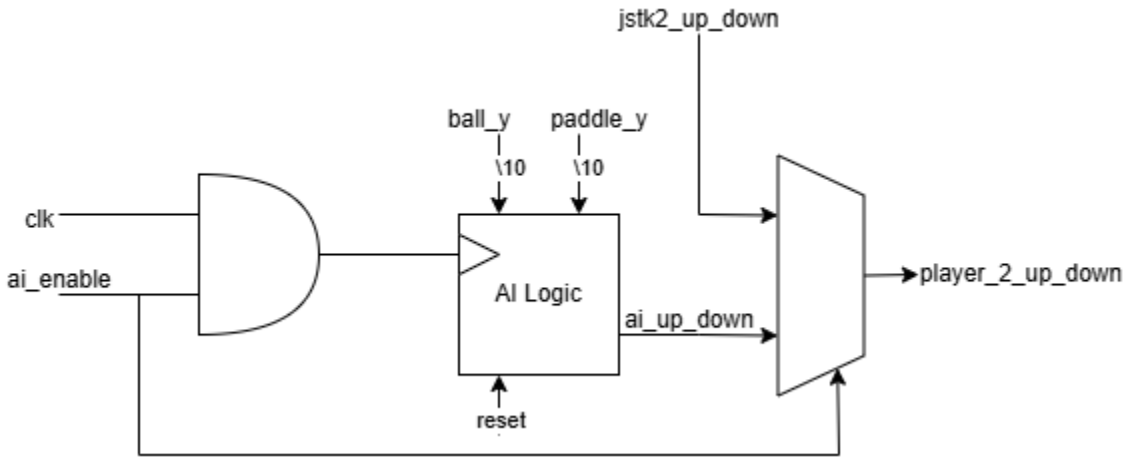
VGA Output

The VGA module is implemented entirely in hardware, and operates by reading in the ball, paddle, and powerup positions and sizes from the game logic engine and renders them onto the display at 30 FPS. A multiplexer muxes the VGA output between the title, game,

and game over screens. A VGA sync module generates the appropriate vsync and hsync signals for the VGA protocol. The game is entirely black and white for simplicity.

Single Player AI Logic

A simple AI logic module was integrated into the design to allow for single-player gameplay. Since this was integrated after the joysticks and two-player version, the method for implementing single-player was as follows. A multiplexer was added between the output signal of the second-player joystick processing unit and the AI-logic. We used a dipswitch on the board to enable/disable the AI logic and to serve as the multiplexer select signal. Depending on the position of the dipswitch, the second player paddle up/down signals were relayed over from either the AI logic or the joystick. This is shown in the figure below:



Developing the AI algorithm for the game ended up being an involved task with several rounds of testing. There was a clear trade-off between making the AI easy to play against vs difficult. In the first iteration, the AI was a perfect player: it took into account the ball's exact trajectory and bounces off of the walls. This made it impractical to play against, since the player could never win. Contrasting this with a simple ball-tracker made the AI too easy to play against, since you could always score if the ball speed was faster than the paddle speed (which was the case with the vertical spin feature).

To resolve this, I sampled two consecutive movements of the ball to compute the speed and trajectory. With this information, it was impossible to score on the AI when the ball moved straight across the screen with no spin, but the AI did get "confused" when there was a sharp or steep trajectory of the ball. It was still difficult to play against (we never won a game against the AI), but the scores were fair (i.e. we played a game where the AI beat us

7-5). With more time, we could have used the dipswitches to also select a level of AI difficulty.

Audio Synthesis with Microblaze

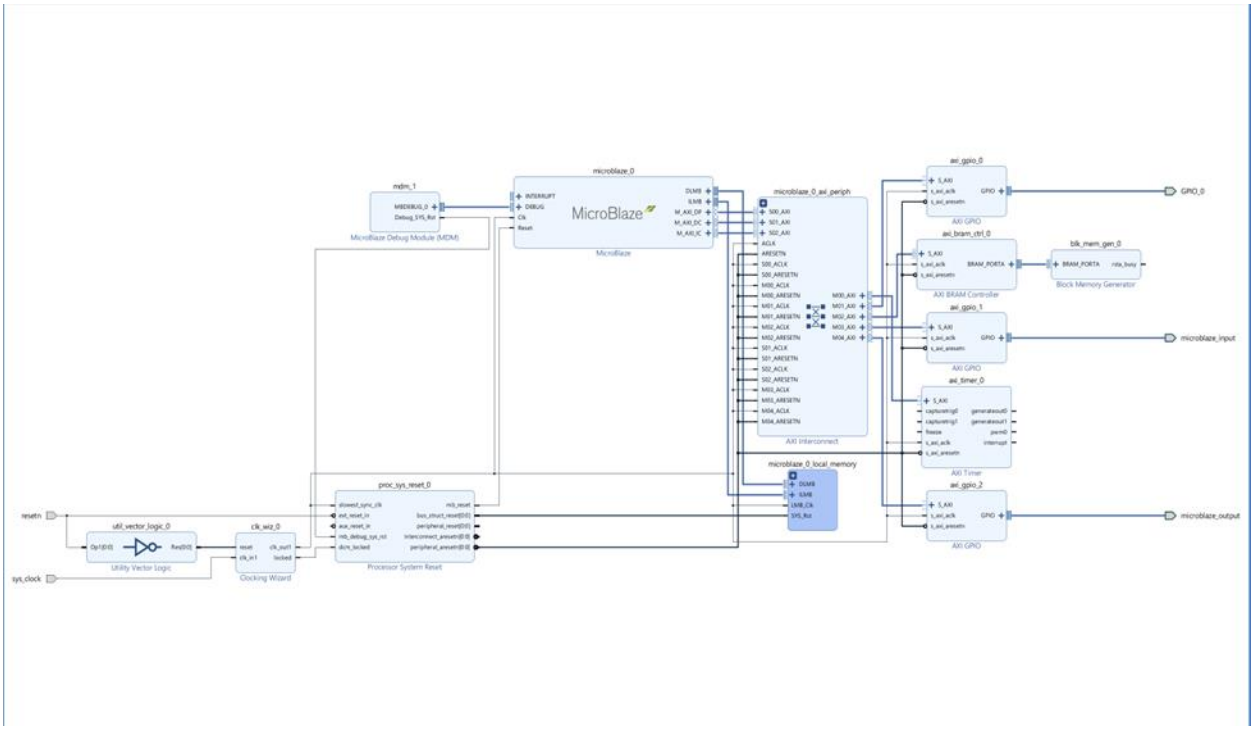
The block diagram below shows our implementation of the audio synthesis hardware in Vivado.

The main component driving this system was the Microblaze processor, which ran the code generating the audio signal. The overall system was also comprised of a few devices over an AXI bus as well:

1. A timer to determine the timing of each note, and for PWM frequency synthesis
2. A GPIO to output the audio signal to the audio port
3. A GPIO to communicate with the game logic
4. A GPIO to turn on the board's low pass filter
5. A BRAM to hold the code and data

This block diagram was wrapped in an HDL wrapper so that it could be integrated into the game.

The method of synthesizing sound using this setup was as follows. The Microblaze, as defined by software, toggles a pin with a square wave at a given frequency. The toggling period is defined by a timer interrupt. The pin will continue to toggle at this frequency for a given duration (also defined by the timer interrupt) – this serves as the duration of the note being played. This form of audio is rudimentary and only uses 1-bit. It is not high-quality, but the sound it produces sounds like a classic video game.



We used a 16-bit bus to encode the sound effects. By asserting a number on this bus, the game logic module could request a specific melody to be played. For example, the theme song was encoded as 0x0001, while the ball-bounce effect was encoded as 0x0002. We used one-hot encoding for the sound effects, since this made it easy to debug by using the dip-switches on the board as test drivers for the bus, before full integration into the game logic.

This method of requesting sound effects required minimal changes in the top level. Since there were already sections of the FSM which dealt with states like “game-over”, “ball bounce”, or “point-scored”, it was easy to add signal assertations for a single clock cycle into these parts of the game.

Design Tree

https://github.com/antoinevilain001/ece532-project/tree/main

Tips and Tricks

For future students we suggest that starting early is essential. Working with hardware like the Nexys4 DDR is very easy to make mistakes. Debugging can be time-consuming and integration between software and hardware rarely works on the first try. Break the project down into small testable modules and verify each part independently before integrating. Also, what helped our group is the use of multiple block diagrams to map out the design early. This helps align the whole team and makes debugging easier. Lastly, make sure to choose a project you’re excited about and it makes you really want to work on it.

One very specific debugging trick that we would also like to recommend is to give your MicroBlaze access to one or more debugging lights as a GPIO. The reason to do this is because it is so easy to read and write to GPIO with the software that anytime we had reason to suspect that anything was going wrong with the MicroBlaze, we would write to one or more lights to check. If we were not sure if the MicroBlaze is stuck in reset, we wrote to a light and see if it turns on. If we were not sure if the for loop is running, we wrote the index of the loop to the lights and see how fast they flicker. If we were not sure if our function was being run, we turned the light on at the beginning of the function to know right away. This small thing that we first did just to test our MicroBlaze ended up saving us countless hours of debugging down the line and we highly recommend everybody to do this, especially since setting up UART output on MicroBlaze is nontrivial so the traditional debugging via print statements takes much more effort.