Group 35: Gyro Racer Final Report

Fiona Kuang, Nicholas Furlan, Yushun Tang, Junze Cao

Overview

Gyro Racer is a racing game using non-traditional controllers as inputs. The player's actions are calculated from a continuous stream of data from a gyroscope. The player can also control their character using the FPGAs buttons or a keyboard as well.
This project is similar to Mario Kart on the Wii. While the Wii remote has buttons for input, steering is determined by tilting the remote. One advantage is that the Wii remote is not directly connected to the Wii, so the players have greater freedom of motion. In contrast, in this project the gyroscope will be connected to the FPGA board via wires, thus the player will be somewhat limited in motion based on wire length.
The main goal of this project is to create a multiplayer racing game using gyroscopes as the game controller. A custom hardware block was created to process the input data quickly so that the game can be played in real time. Additionally, a custom sprite controller block was created to efficiently render the moving cars on the screen at the rate of 60 Hz, so that the frame rate is consistent.

Functional Requirements & Features

This project aims to:

- Take input from up to 2 players using gyroscopes
- Synchronize the inputs through a hardware block, which will deliver these inputs to the MicroBlaze processor
- Run all game logic through the MicroBlaze processor
- Display output at a consistent framerate through the VGA output
- Utilize sound output through the on-board audio output port

Acceptance Criteria

To fulfill requirements, this project must:

- Allow for 2 players to play and display the correct game logic
- Display the correct player movement according to player input
- Display correct game logic
- Achieve at least 30 frames per second graphics
- Output sound effects according to gameplay

Block Diagram

The following Figure is the final high level block diagram for Gyro Racer.

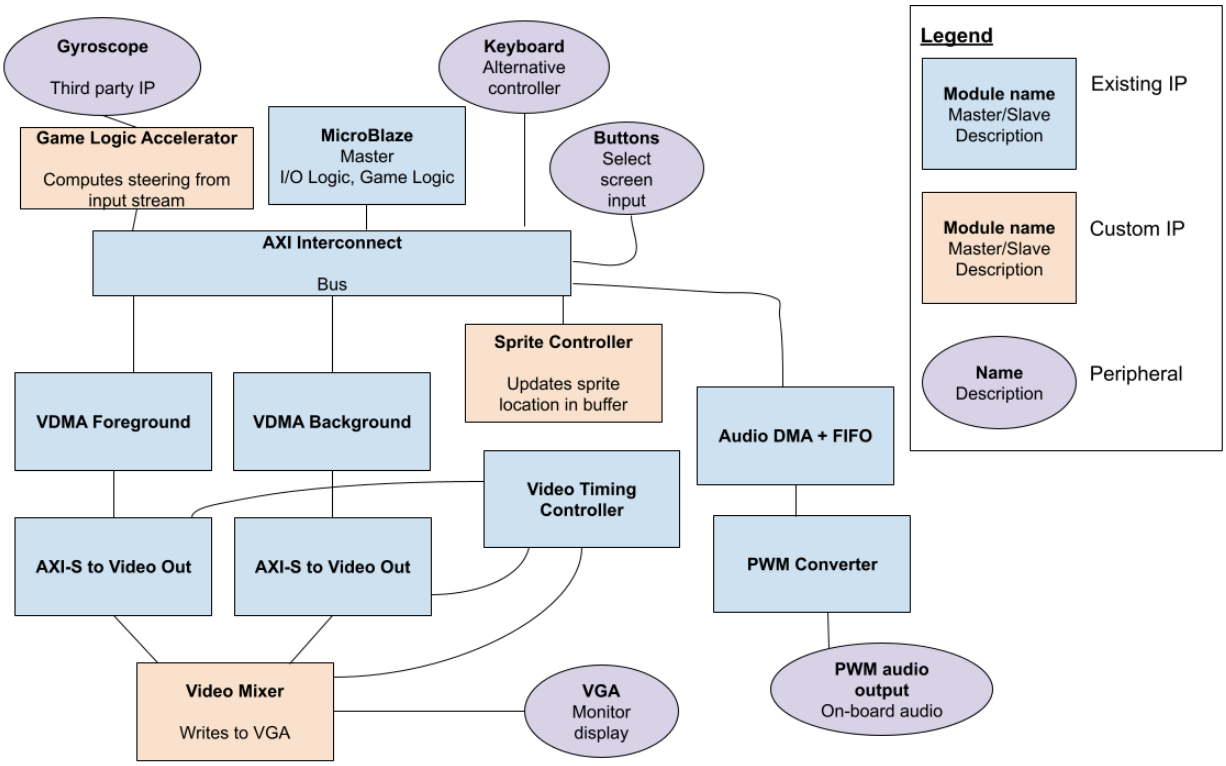

Figure 1: Gyro Racer's block diagram

Many of the IPs used in this project were from Vivado's library. MicroBlaze handles the game logic and I/O such as buttons and keyboard excluding the gyroscope. MicroBlaze draws images into the frame buffer upon initialization and sends signals when audio should be played. The custom sprite controller block updates the sprite locations in the frame buffer during the game time. A third party IP reads the gyroscope's internal registers and sends it to the custom gyroscope angle conversion IP to calculate the resulting angle.

Outcome

Our project was successful in implementing our goals, as listed above. Our project is capable of being controlled by up to 2 players, either both using gyroscope controls or with mixed controls (including the keyboard, buttons, and gyroscope). These inputs are read through hardware, with specialized hardware designed to compute the current angle measured by the gyroscope before being sent to the MicroBlaze processor. From there, the processor handles the game logic. We are able to get a consistent frame rate through a combination of the MicroBlaze processor and a hardware sprite controller. The processor controls the background (which does not change throughout the course of a race), while our hardware sprite controller controls sprite drawing

and movement, allowing us to achieve a consistent 30 frames per second framerate at the VGA standard resolution. Lastly, we got audio output working, allowing us to have a background music track play during gameplay, as well as sound effects for making a selection and finishing a race. A difference between our proposed and final features is that we originally considered other forms of input for the project, such as an accelerometer and joystick, but we ultimately decided that adding these would add a lot of complexity for minimal gain. If we had a chance to restart our project, we would try to ensure we started with the Nexys Video board in order to utilize higher resolution graphics, and we would aim to implement sound earlier to better integrate it into our system. Despite this, we believe that our project was overall excellent in meeting all of the goals we set, and implemented all features we wanted.

Future Work

One location we could improve upon is our implementation of sound in our project. While our sound controller uses DMA to load sound effects on demand, it was implemented late into development, so we didn't have an opportunity to use it to the fullest. In the future, additional sound effects could be implemented for different situations, such as a unique startup and ending sound effect, or a collision sound effect, as well as multiple background music tracks for each course.
Another place we could improve in the future is the race course complexity. While we have unique courses, they do not have any mechanics unique to themselves. A possible change would be to implement different terrain types, such as ice and sand, each with their own physics. As well, we could implement moving obstacles, in order to make each run through of a track less predictable. As the sprite controller can handle drawing many more sprites than it currently does, these changes should not require any large hardware rewrites.
A final change we could make is using an SD card for storage. This would allow us to store more courses, sprites, and sound effects, which would be necessary if the above changes were implemented. This would also allow us to store higher definition graphics and audio, potentially leading to a possible upgrade if we could get a Nexys Video board in the future. Lastly, an SD card would allow us to save persistent information across sessions, which we could use to keep track of a persistent leaderboard for best times.

Project Schedule

The following table compares the proposed milestones and the actual progress during the milestones.

Milestone	**Proposed**	**Actual**
1	● Do preliminary research	● Researched IP blocks, PMODs, game logic

	○ Existing IP blocks that can be used ○ Potential PMOD for secondary input	
2	● Get required hardware for the project: ○ Gyroscope input ○ Speaker (if needed, lab workstations may have 3.5mm output) ○ Secondary inputs (ex accelerometer) ○ Wires (must be long enough for both players comfortably control with and move) ● Swap to the Nexys Video board if possible (for higher resolution output and better performance) ● View controller & keyboard inputs through the board (though not yet interpreting) ● Display basic video output ● Play basic audio output	● Received PMODS ● Interfaced gyroscope with MicroBlaze ● Interfaced VGA with MicroBlaze and got graphical output ● Interfaced keyboard to board and got LED output ● Could not get a Nexys Video board
3	● Basic gameplay output: can display and move around a point on screen with the keyboard, some form of collision detection implemented ● Audio synced to gameplay (although only simple sounds like beeps implemented) ● Interpreting direction and intensity given by all planned inputs	● Formula to convert angular velocity to angle in MicroBlaze ● Started creating custom IP for gyroscope calculations ● Interfaced keyboard with MicroBlaze ● Improved VGA output
4	● More advanced game logic: control speed and acceleration of player, implement more	● Added angle conversion to custom IP and added AXI-lite slave interface ● Game logic for movement and boundary collision implemented

	obstacles and at least one racetrack ● Basic multiplayer (even with just two keyboard inputs) ● Improve graphics output of game: higher quality sprites, more animations ○ Try to have some of these graphical improvements done for the mid-project demo ○ Low / inconsistent frame rate is fine for now, will be optimized later ● Synchronize all inputs in hardware: should be simple past this point to interpret these signals in the game logic	● Mid-project integration ● Started creating a custom IP to control the sprites
5	● Optimization: move game logic to hardware where possible, aim for a consistent frame rate ● Additional track design - move beyond the bare minimum of functionality for tracks ● Improved audio: multiple sound effects, potential background tracks, overlapping sounds ● Playable with all inputs, including combinations of different inputs	● Confirmed custom IP for gyro calculations works with MicroBlaze ● Fixed drift issues in gyro calculations ● Created multiplayer game logic ● Improved hardware sprite controller custom IP
6	● Should be playable at a consistent frame rate with consistent, synced audio, with all forms of supported input ○ Not much added compared to Milestone 5: gives us time to catch up if we fell behind earlier, and time	● Implemented sound effects and background music ● Created new racetracks for the game ● Tested multiplayer game logic ● Completed sprite controller block ● Added reset to the gyro calculation block ● Integrated software and hardware components

	for general polish and debugging otherwise	

Table 1: Proposed and Actual Milestones

Discussion

The final milestones diverged from the proposed milestones early on after receiving feedback from our TA. The TA advised focusing on working on the gyroscope as it was the most critical aspect of the project, thus the optional feature of audio was postponed for later milestones, and we stopped trying to work with the accelerometer or joystick PMODs. Creating the custom IP block for gyroscope calculations also took more time than expected, causing a delay in other features such as game logic that depend on the completion of input functionality. The reason for the delay with the gyroscope was because the initial plan to access the gyroscope's internal registers through Digilent's gyroscope IP only worked with MicroBlaze and not with a custom AXI master IP block. An alternative method to accessing the gyroscope PMOD was only found during milestone 4. Another deviation to the milestones was integration issues with the keyboard. It would work fine as a standalone project, however when integrating with the main project it would not work. This was an ongoing task that was resolved in milestone 6.

Detailed Description of the IP Blocks

The IP modules used in this project can be roughly divided into several categories: Controller Input, Video Rendering and Output, Audio Output, Processor Interfaces and Software & Game Logic. For each category, if there is software closely related to the category (such as drivers, API libraries, etc.), the description will be given together; otherwise, the general software logic will be given in the software part.

Controller Input

This project supports multiple input methods for game control and menu navigation.

Gyroscope

Our primary way of controlling the movement of the car is with the two gyroscope blocks in our project. Our gyroscope block functions by taking the angular velocity measured at each axis, and outputs the current angle at each axis. The angular velocity is read directly from the PMOD gyro using a controller found at [1]. This controller uses SPI communication to constantly read registers from the gyro, and will constantly output the angular velocity measured by the gyro at any point in time.

On boot or reset, our gyro module reads 255 values from the gyro, then divides the result by 255, getting an average rest value of the gyro at each axis. While this value should be 0, a gyro may have slight variations in what it outputs, so this calibration is required to minimize gyro drift.

We also have a minimum reading value, below which any angular velocity readings will be filtered out. Both of these features ensure we have minimal drift on the gyro during gameplay. If any drift is found anyways, we have reset for the gyro blocks tied to two switches on the board, allowing any drift to be reset away for either gyro (and allowing players ensure the neutral gyro position is where they expect).

In order to convert angular velocity into an angle, our hardware block performs integration. Every cycle, a measurement is taken for all three axes, and for axes where this exceeds the minimum filter value (adjusted with the average measured at calibration), we add (or subtract) this value from our running sum. As this sum will rapidly grow large due to repeated addition, we use a 64-bit register to store it, while angular velocity measurements are 16 bits. Our output right shifts this register to output the most significant 16 bits to our AXI controller, where the angle can be read by the MicroBlaze processor. While this angle is unitless, we do not need a unit for our purposes since calculating the speeds of the cars does not require a unit.

Keyboard

The keyboard is one of the ways to control the movement of the car. The IP block that communicates with the keyboard is axi_ps2 from Digilent. Our keyboard supports the PS/2 protocol, which uses a two-wire serial bus (clock and data) to communicate with a host. The keyboard uses 11-bit words that include a start bit, data byte (LSB first), odd parity, and stop bit. It uses scan codes to communicate key press data. Each key is assigned a code that is sent whenever the key is pressed. When a key is released, an F0 key-up code is sent, followed by the scan code of the released key. [2]

The 4 keys we keep track of are “w”, “a”, “s” and “d”. The typical keyboard behaviour is, when two keys are long pressed, the later one will hijack the sequence. To prevent this unintended behavior, we flag the status when it is pressed, and unflag when it is released. This way more than one key press signal can be sent at the same time, enabling force impulse in multiple directions.

On-board Buttons and Switches

The on-board buttons serve as the primary control method for menu navigation and an alternative way of game input. Five buttons (up, down, left, right, OK) can be used for menu selection and vehicle control. The switches `SW[15:0]` are mainly used for debugging: `SW[15]` and `SW[0]` reset gyroscopes to zero points, `SW[8]` resets the MicroBlaze processor, and `SW[7]` forces return to the title screen. These inputs are connected to the AXI Interconnect via Xilinx AXI GPIO interface and polled by the MicroBlaze processor whenever the game state is updated.

Video Rendering and Output

Color Representation, Dual Buffer and Video Mixer

The Nexys 4 DDR board features an on-board VGA connector with RGB channels, each supporting 4-bit color depth. To optimize memory alignment, this project employs the RGBA4444 color format. However, for efficient resource utilization, the alpha channel serves a binary purpose rather than representing opacity: a non-zero alpha value indicates an opaque pixel, while a zero value indicates transparency.

The video buffer architecture implements a dual-buffer system, separating foreground and background elements for optimal rendering performance:

- The background buffer stores static elements such as the track and environment
- The foreground buffer manages dynamic elements including player vehicles and moving objects

This enables independent updates of foreground elements without affecting the background content, eliminating the need for full screen redraws and significantly enhancing rendering performance.

To combine the two frame buffers, a custom Video Mixer IP is implemented. The mixer follows a simple compositing rule: if the foreground pixel is opaque, it outputs the foreground pixel; if the background pixel is opaque, it outputs the background pixel; otherwise, it outputs black. This straightforward approach ensures minimum resource utilization, efficient pixel selection while maintaining visual clarity.

The output pins of the Video Mixer are connected to the external VGA connector. To specify the I/O standard and the package pin for the VGA connectors, a few rules in the only constraint file Nexys-4-DDR-Master.xdc are specified.

Hardware Graphics - Sprite Controller

The Custom IP Sprite Controller enables high-speed rendering of player-controlled cars by reading sprite information and rendering it onto the video buffer. Sprites are 2D bitmap images that can be moved independently on screen, enabling efficient animation without full screen redraws.

In this project, sprites are either 16x16 or 32x32 pixels in size, with each sprite represented by a 32-bit unsigned integer. The bits are defined as follows:

- Bits 0-11: X-coordinate of the sprite on screen
- Bits 12-23: Y-coordinate of the sprite on screen
- Bits 24-30: Index of the sprite in the corresponding 32x32 or 16x16 tile list
- Bit 31: Flag indicating whether the sprite is 32x32 (1) or 16x16 (0)

If all bits of a sprite information are set to 1, the Sprite Controller considers it an invalid sprite and ignores it.

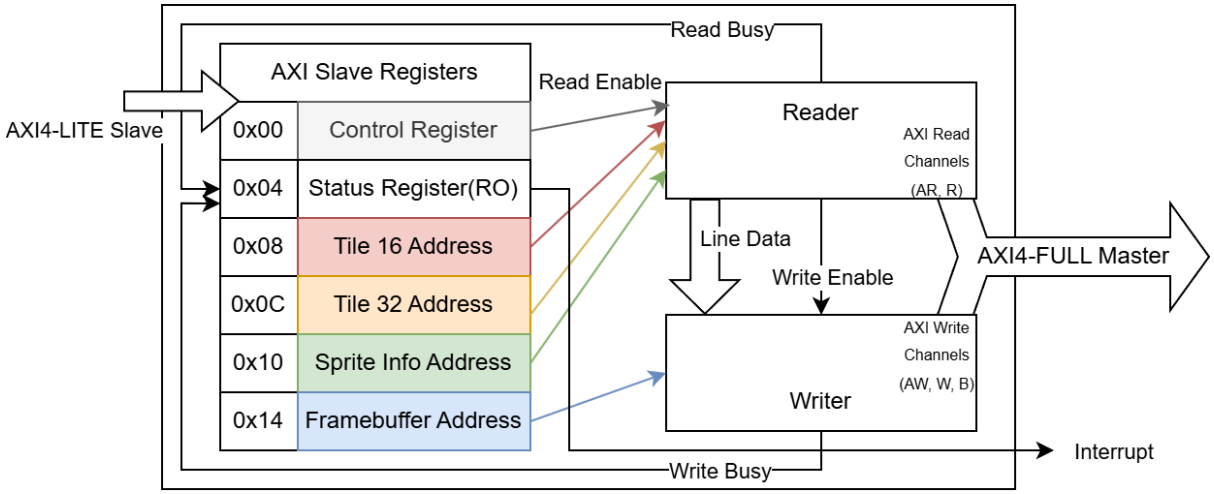

Figure 2. Block Diagram of the Sprite Controller

The figure above shows the diagram of the block. This module has an AXI4-Lite slave interface for register access and an AXI-Full master interface for memory reading/writing. The registers for addresses are used by the internal reader/writer block, while the control registers can be written to start an operation, enable/disable interrupt, acknowledge an interrupt request, and the status register shows the busy status of the writer and the reader. The writer block can be considered as controlled by the reader.

The IP supports rendering up to 32 sprites simultaneously. These sprite informations are stored as an array at a specific memory address. The MicroBlaze provides the Sprite Controller with a pointer to the sprite information. After initialization, the Sprite Controller reads and caches all 32 sprite informations from this address.

The Sprite Controller accesses pattern data from separate memory spaces for 16x16 and 32x32 sprites, with pointers provided by the MicroBlaze. Based on the sprite index, it reads pattern pixels and writes them to the buffer at the calculated memory address.

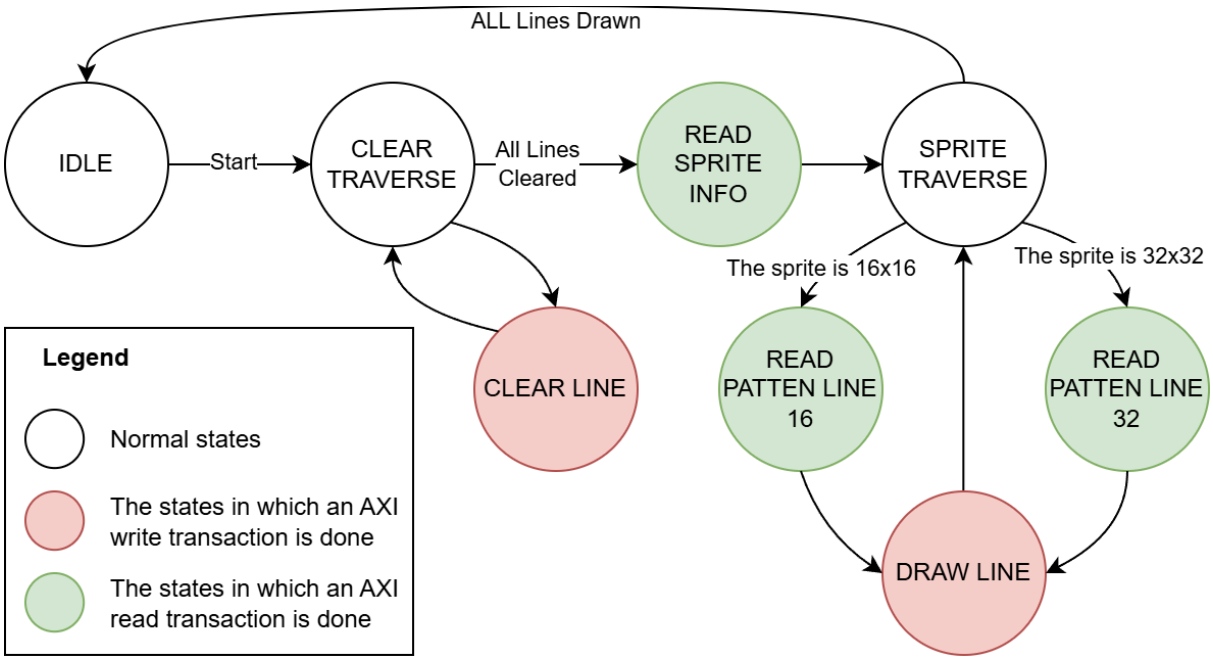

Figure 3. FSM of the Sprite Controller

The state machine manages the rendering process with a CLEAR-READ-WRITE loop to draw all sprites onto the frame buffer.

Software Graphics Library

For implementation simplicity, static images (background images, tracks, etc.) are drawn by the MicroBlaze during initialization. To facilitate game logic development, this project implements a series of graphics processing APIs. The key components include:

- **Color**: Implements the RGBA4444 color representation mentioned earlier
- **Image**, **ImageView**, **ImageSlice**:
  - **Image**: Represents a complete image
  - **ImageView**: Represents an image without memory ownership
  - **ImageSlice**: Represents a rectangular slice of an image
- **Font**: A class for storing and managing font data
- **Draw**: A drawing class that supports rendering various shapes and elements on images or any graphics buffer, including:
  - Points
  - Lines
  - Rectangles
  - Circles
  - Text
  - Other images
- **Display**: An abstract representation of video buffer memory

Video Output Dataflow

The system employs several Xilinx IP cores:

1. **Video Timing Controller (VTC)**: Generates 800x600@60Hz video synchronization signals
2. **AXI Video DMA**: Performs DMA operations to transfer video buffer data through the AXI4-Stream interface
3. **AXI4-Stream to Video Out**: Outputs pixel color values to the Video Mixer IP at the correct timing based on VTC synchronization signals

Due to the dual-buffer architecture, there are two independent data paths, both controlled by the same VTC for synchronization.

Additionally, the Video DMA supports interrupt signals and can be configured to generate an interrupt request after completing one frame transfer. This interrupt signal (60Hz) serves as the synchronization signal for the entire game loop, ensuring that game logic updates and rendering occur every 1/60 second.

Audio Output Dataflow

The Nexys 4 DDR board features an on-board audio connector supporting mono audio output driven by a PWM signal. The audio system consists of three main components:

- **Xilinx AXI DMA**: Configured in Simple DMA mode to read continuous audio sample data from DRAM and output through the AXI4-Stream interface.
- **Xilinx AXI-Stream Data FIFO**: Temporarily stores audio samples and provides programmable full/empty outputs through AXI GPIO ports to the MicroBlaze for congestion control feedback.
- **Custom IP Blocks**:
    - **axis_acker8k**: Features an AXI4-Stream slave interface and an 8-bit audio sample output port. When TVALID is high (indicating data input), it outputs TREADY at 2000 Hz, accepting one 32-bit sample every 1/2000 second. The system uses PCM 8-bit unsigned format, with each sample occupying 8 bits, thus the 2000 Hz data rate results in an 8000 Hz sample output frequency.
    - **PWM Generator**: Converts input sample data into PWM signal at a frequency of clock frequency (100MHz)/32, thus supporting 32 duty cycles for acceptable audio quality.

The software implementation on the MicroBlaze processor manages audio sample transmission at 10 Hz. It monitors the FIFO's sample status to determine whether to skip the current transmission:

- If the programmable full signal is active, the software suspends sample transmission until the programmable empty signal appears
- If the programmable empty signal is active, the software transmits 896 samples at 10Hz (greater than 8000 Hz/10 Hz to prevent audio device starvation) until the programmable full signal appears.

Processor Interfaces

This section describes the general IP blocks needed to build up the processor system.

MicroBlaze IP

The system is built around a MicroBlaze soft processor core, configured with:

- Maximum Performance Predefined Configuration
    - This enables almost everything that could make the processor run faster, including extended FPU, integer multiplier and divider, branch target cache, etc.
- 32KB instruction and data caches
    - The instruction and data caches are useful when the code and data is stored on external DRAM, which significantly boost the performance.
- 16KB local memory
- Trace Bus Interface
    - Although this trace interface is unused at the end, it is very useful for debugging with ILA when sometimes the GDB doesn't work, e.g. the reset vector and interrupt vector were overwritten by some illegal memory access, and the processor is in some undefined state.
- MicroBlaze Debug Module connected

Interconnects

The processor system has two isolated interconnects. The first is an AXI SmartConnect which is used to connect all AXI master ports (the MicroBlaze Instruction/Data Cache port, the Video DMA, the Audio DMA and the sprite controller) to the DRAM, and the second is an AXI Interconnect which is used to connect the MicroBlaze Peripheral Data Port with all peripherals. This isolation is implemented for several reasons:

- Memory access and peripheral register space access are typically unrelated operations
- Some DMA devices only need memory access and don't interact with peripherals
- This design reduces the load on each arbiter, improving system performance

The AXI Smartconnect is said to have better performance for high throughput and low latency requirements by some AMD adaptive support answers, so we applied it as the memory interconnect without verifying if the SmartConnect does have better performance.

Peripherals

The MicroBlaze is connected to the following peripheral IPs:

- AXI Interrupt Controller
- AXI Timer
    - The timer is actually not useful after the whole system is built. It is used to request MicroBlaze to refill the Audio DMA at the rate of 10 Hz, but this process can actually be synchronized with Video DMA which operates at 60 Hz.
- AXI Uartlite

Software and Game Logic Implementation

This section explains some of the key components in Software and Game Logic, namely Game State Management, Race Logic, Car Mechanics, Input Handling and Rendering.

Game State Management

The game consists of 4 distinct states: GameStart, GameOption, GamePlay and GameEnd. The transition is outlined in the figure below:

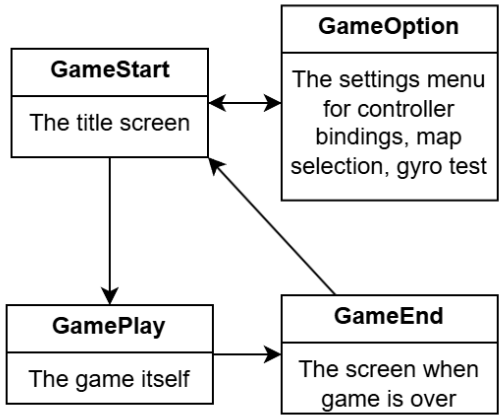

Figure 4: Game State Transition

1. GameStart: The initial menu screen displaying the project name, group member names, option for single player and multiplayer.
2. GameOption: Configuration menu for customizing controls, tracks, and race settings. A user can select from the following input: gyroscope, button and keyboard. For both gyroscopes, a user can test the sensitivity as well as resetting them. Other options include selecting 1 of 3 available track maps and the number of laps.
3. GamePlay: Actual track map. Car mechanics will be elaborated in the next section.
4. GameEnd: Post-race results screen with race statistics and exit options.

Race Logic (gameplay.hpp)

- Lap Progression: For a user to complete a lap, a car needs to cross the halfway checkpoint, and then cross the finish line. A halfway checkpoint is designed so that small movement around the finish line does not trigger a lap increment.
- Time Elapsed: Because we keep the frame rate at 60FPS, keeping track of the frame count is enough to calculate the time elapsed.
- Winning: For a 2-player game, when the slower player crosses the finish line, the game is done and the game state moves to GameEnd.

Car Mechanics (car.hpp)

- Acceleration: An input, regardless of from which sensor, is considered a force. According to Newton's second laws of motion, F = ma, the force then becomes acceleration given

constant mass. We consider the force in a cartesian plane, so we break it down in the x and y direction. To calculate the velocity, we use
$v = (v + a) \times (1 - \text{FRICTION})$
where FRICTION=0.01.

- Collision Detection: We consider 2 types of collision: Car-to-Car collision where two cars collide and map collision where a car runs into the boundary. Either way, we use a collision box in a car to check.
    - Car-to-Car collision: Check if the two collision boxes have any overlaps. If there are, assuming conversation of momentum, we set both velocities to the mean of the two, and then check the Map collision in the next step.
      $v = (v1 + v2) / 2$
    - Map collision: Check if the collision box at the new position has any overlaps with the boundary. If there are, velocity is set to 0 in that direction.

Input Handling (controller.hpp and drivers under ./platform)

The following input methods are available to users to control the car. We discuss the software aspect of the input in this section.

- Gyroscope: Initialize and calibrate both gyroscopes. In the event of drift during game play, a built-in switch on the DDR board for each gyroscope can be used to zero out the drift to reset.
- Keyboard: Initialize the keyboard which configures the interrupt and enable it. Manage the sequence of interrupts needed to complete a keystroke.
- Button: Simple GPIO input which will not be discussed here.

Once the inputs are recorded, we feed them as impulses to control the car, based on bindings.

Rendering (under ./graphics)

- Sprite Controller: Microblaze sends a signal to the Sprite IP, which draws the 32x32 sprite based on the x and y coordinates. Previous positions are erased before drawing. This way microblaze acts as DMA and hardware acceleration is achieved.
- Direction Handling: Preload 8-directional car sprites, and render them based on direction calculated using vx and vy.

Design Tree

Github Repository: https://github.com/nicholas-furlan/ece532_gyro_racer
The design tree structure is organized into the following parts:

- `assets/`: Resource files (track image, BGM, font, ...) and utility python scripts for converting binary files (image, music) to C/C++ header.

- `docs/`: Final project report, presentation slides, and resources referenced in the README Markdown document.
- `final/gyro_racer_proj/`: The Vivado 2018.3 project directory.
    - `gyro_racer.srcs/sources_1/`: The RTL code for all custom IPs and the 3rd-party dependencies.
    - `gyro_racer.srcs/constrs_1/`: The constraint file used in the project.
- `ip/`: The packaged IP location
- `ip_keyboard/`: The AXI PS/2 IP from Digilent Inc. for keyboard interfacing.
- `software/`: The software C++ files for this project.
- `stash/`: The temporary files used during development, which can be safely ignored.

For detailed version of the tree view of the repository, see `README.md` in the Github Repository

Tips and tricks

- Not all gyro IPs available online will have the same functionality. The gyro IP we initially had success with (found through the tutorial at [3]) worked well with the Vivado SDK, but we were unable to get it to communicate with hardware blocks. While the IP we ended up using in the final project [1] is perfect for our needed, it only allows access to the angular velocity registers, so modifications are required if you want to access any of the other registers (such as configuration)
- The video output implementation is based on the design example from online tutorial for HDMI/VGA framebuffer on Artix-7 FPGA board [4]. This is a very detailed and useful tutorial to make a basic working video subsystem.
- Try to understand the AXI standard as early as you can. The AXI is a widely used interface between IPs provided by Xilinx, if you have a deep understanding of the standard protocol you will save a lot of time in the design and debugging of the system.

Video

A video demonstrating Gyro Racer can be found at the following link:
https://www.youtube.com/watch?v=LTuoZn6Tse0

References

[1] https://forum.digikey.com/t/gyro-l3g4200d-pmod-controller-vhdl/23030
[2] https://digilent.com/reference/programmable-logic/nexys-4-ddr/reference-manual
[3] https://digilent.com/reference/learn/programmable-logic/tutorials/pmod-ips/start
[4] https://numato.com/kb/simple-hdmi-vga-framebuffer-design-example-on-neso-artix-7-fpga-board